DS4300 - Large-Scale Storage and Retrieval

Distributed Data: Replication and Partitioning

What is a distributed system?

"A collection of independent computers that appear to its users as one computer."

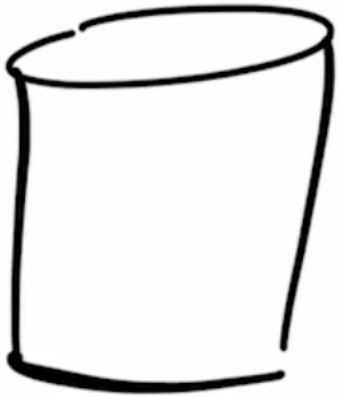
- Andrew Tannenbaum

Three characteristics of distributed systems

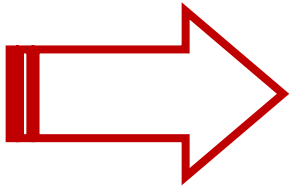
- The computers operate concurrently
- The computers fail independently
- There is no shared global clock

Distributed Storage

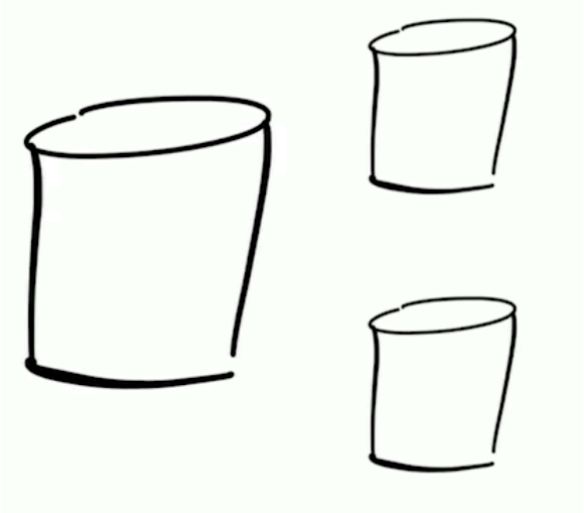
These problems are well-understood when we are dealing with a single computer / single thread.



Single Master Storage



Replication:

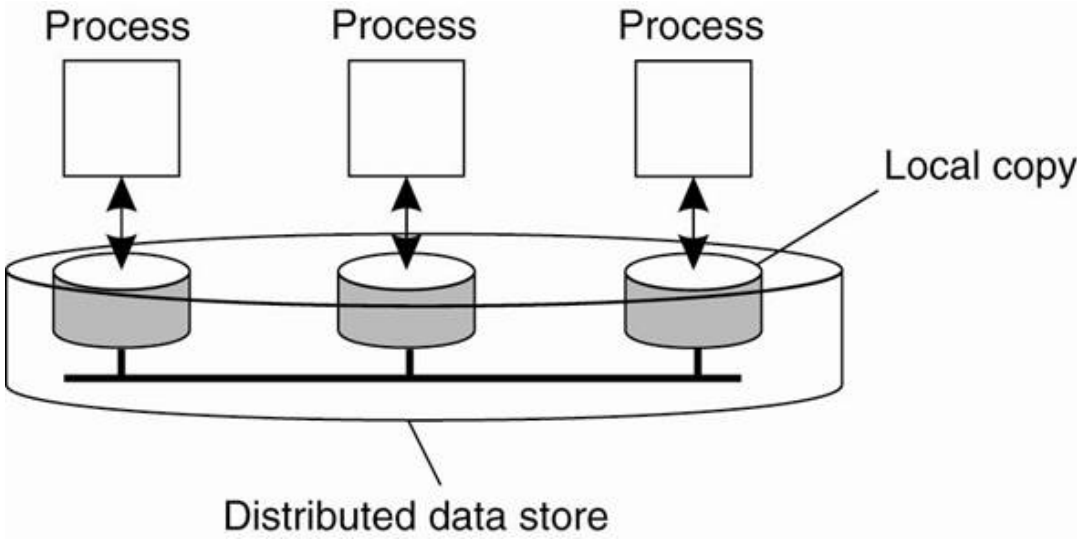


Sharding:

CAP Theorem

Distributed Data Stores

- Information is stored on more than one node, often in a replicated fashion (i.e., each data block is available on N nodes)
- Distributed databases are typically non-relational
- May have rich querying semantics or limited to query by key (e.g., key-value stores) such as Google BigTable, AmazonDB, and Windows Azure Storage.



Network Partitions…are inevitable!

- No distributed system is protected from network failures, therefore, applications must be designed to tolerate network failures

- **Partition Tolerance:** The ability of a data processing system to continue processing data even if a network partition causes communication errors between subsystems.

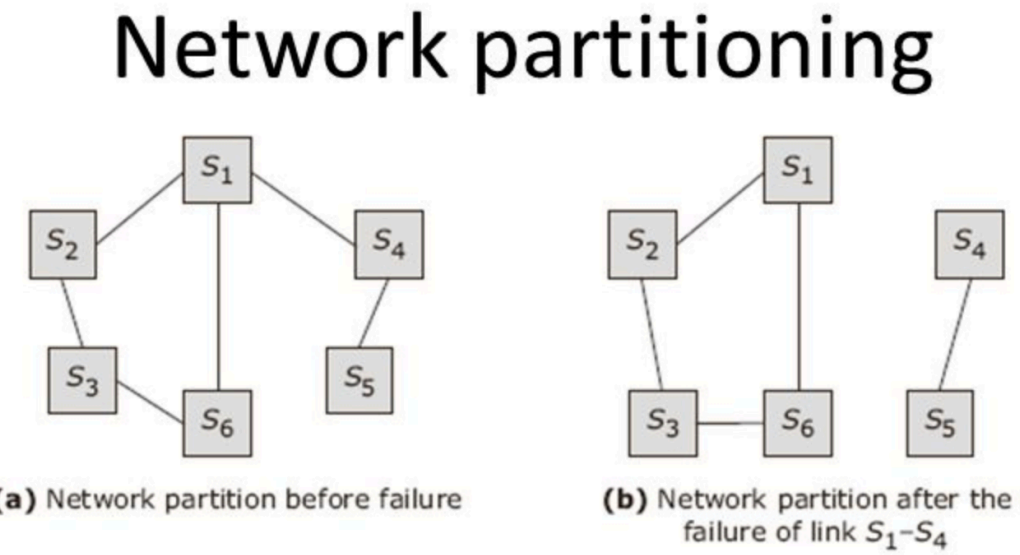
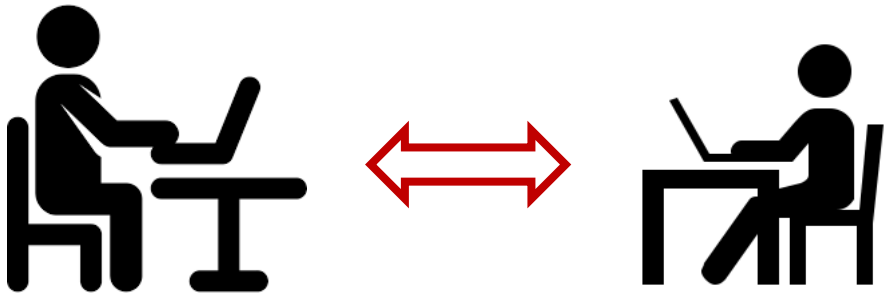


Figure 12-18 Network partitioning

If the network of nodes has failed, any one of the reasons may exist

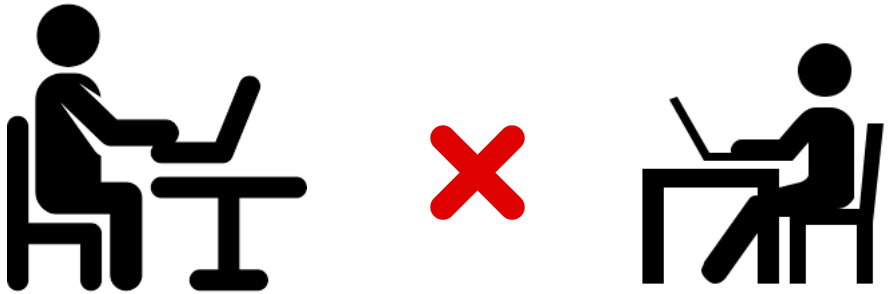
- The site has crashed or the network is gone.
- The communication link has failed.
- The network is partitioned.
- Site S_4 is very busy and cannot respond.

Example: Pair programming

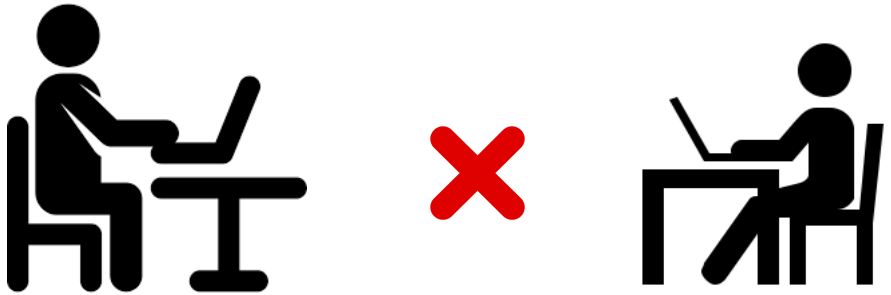


Manager
How's it going?

You
Fine!



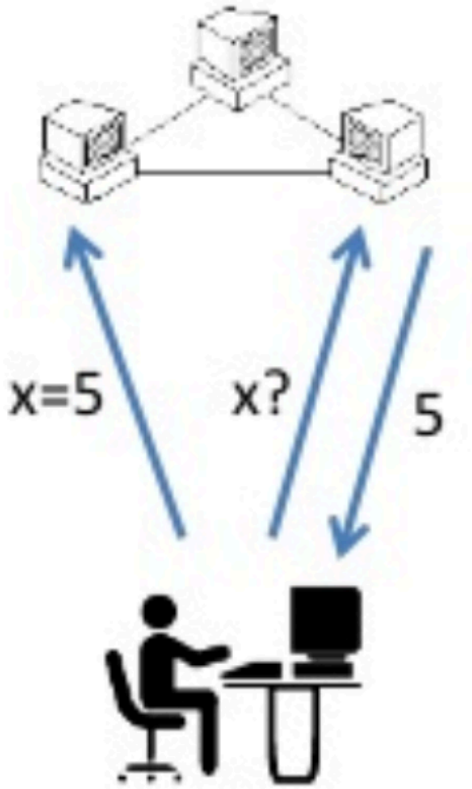
You
Ask me later.



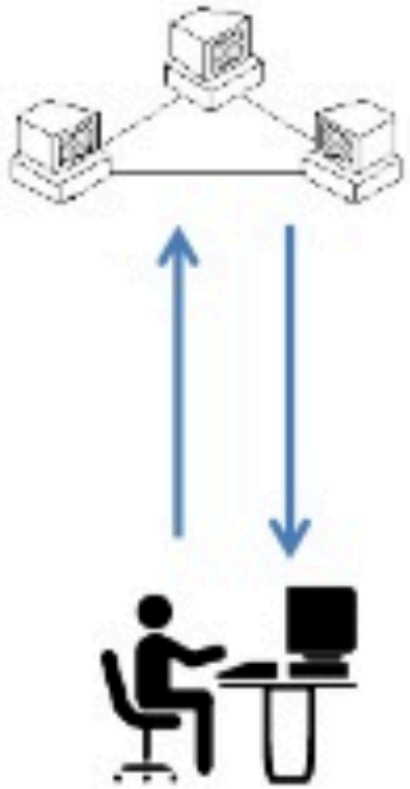
You
Here's my report.
(Might be wrong.)

CAP Theorem

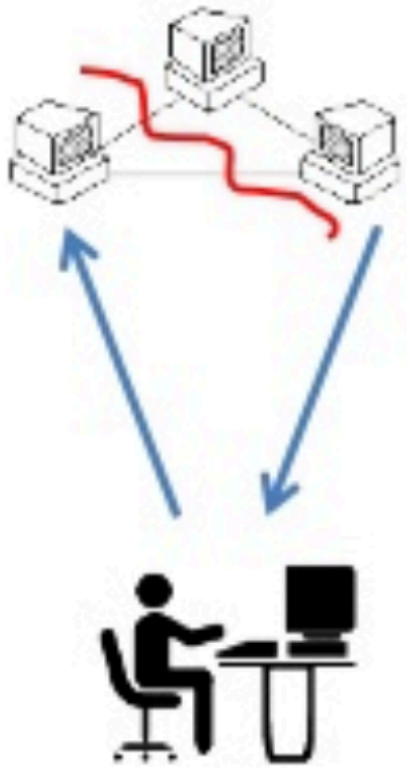
Consistency



Availability



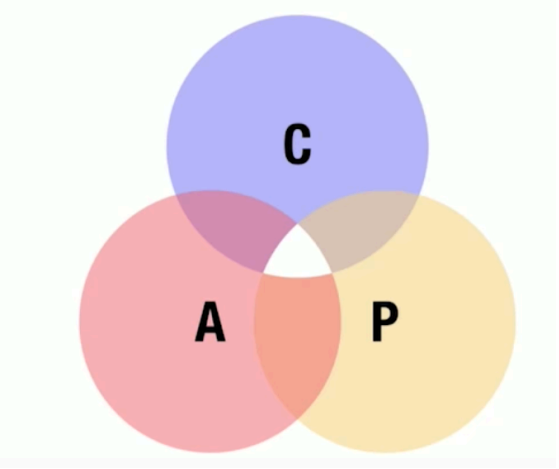
Partition tolerance



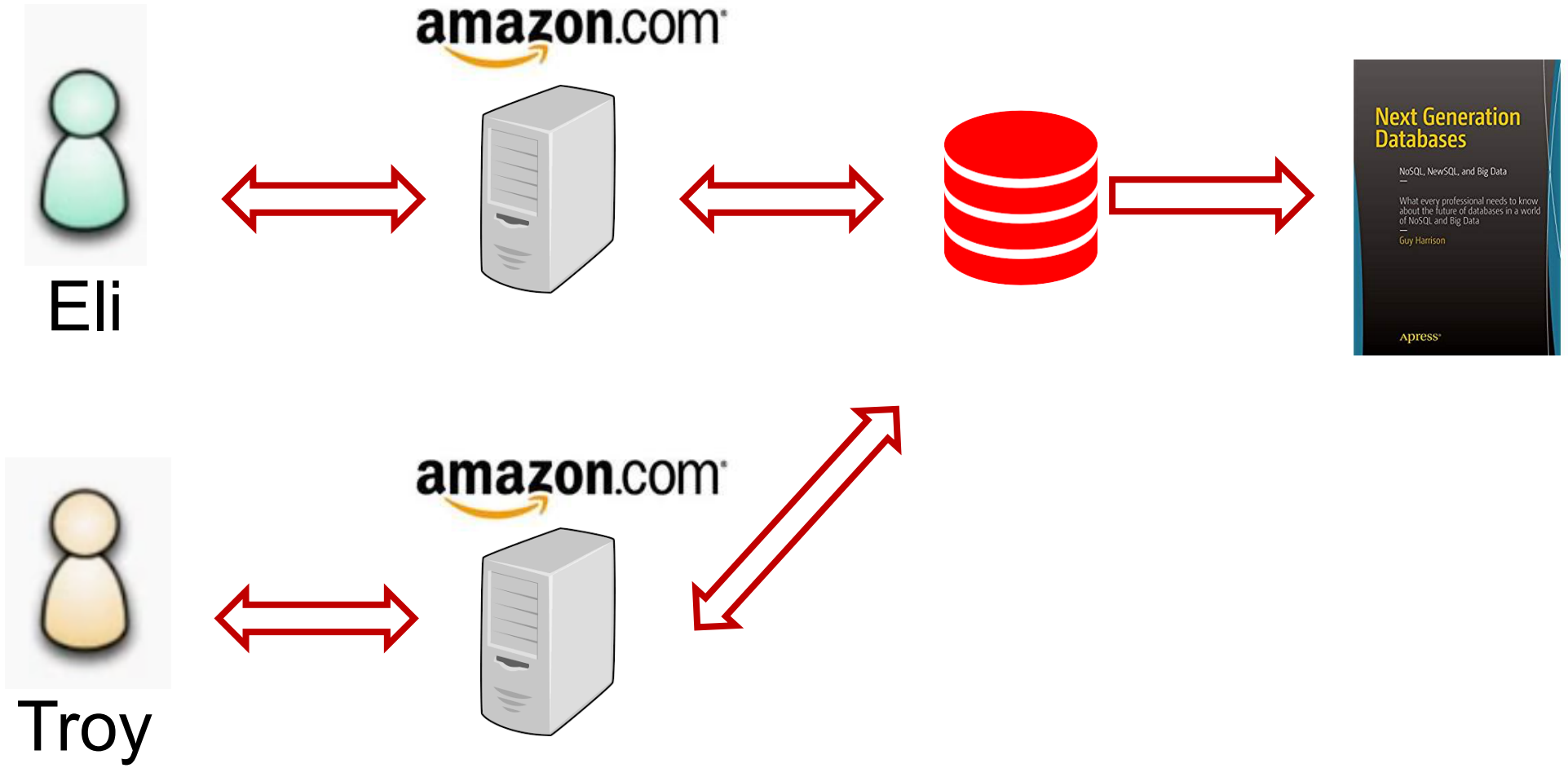
The CAP Theorem

The **CAP theorem**, also named **Brewer's theorem** after computer scientist Eric Brewer states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

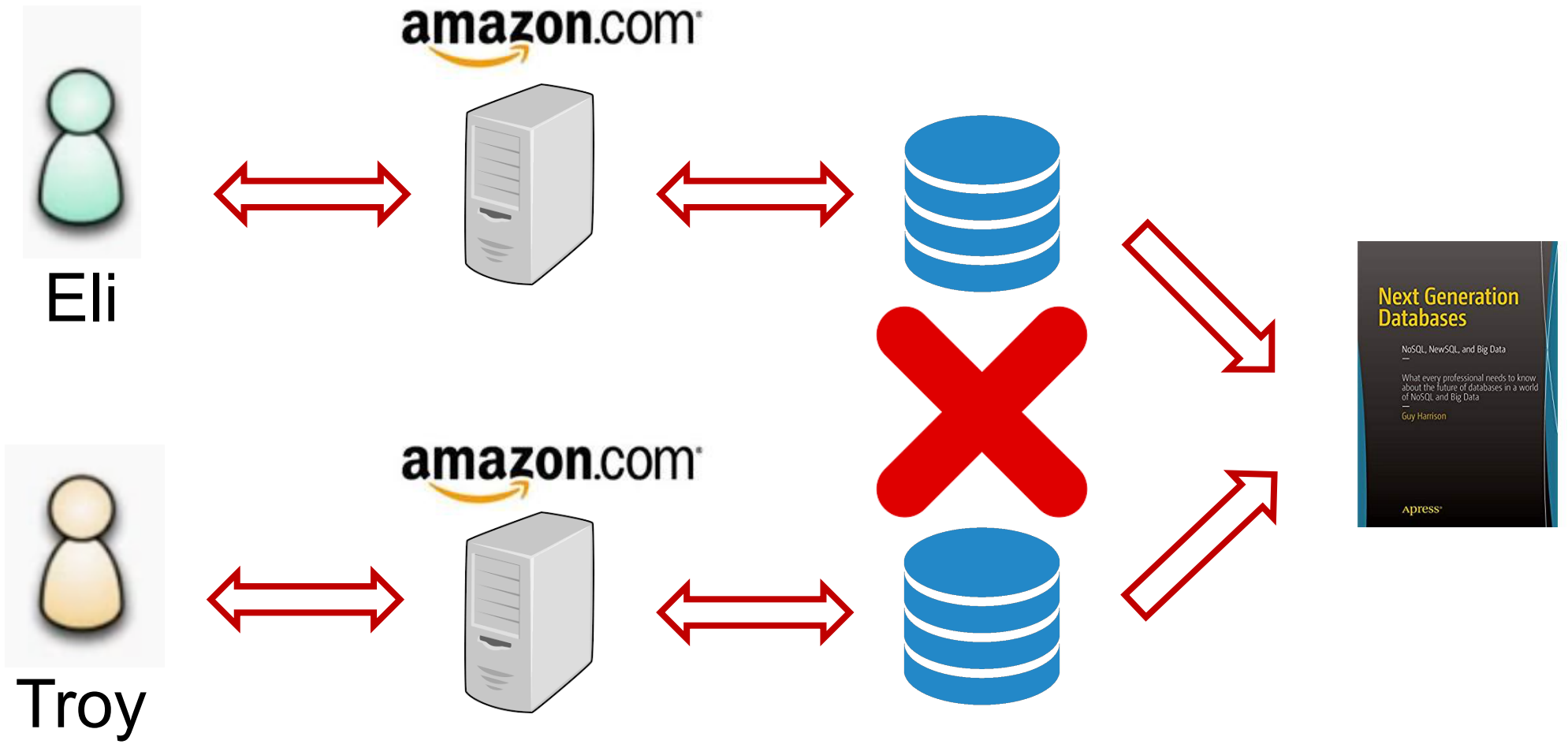
Consistency	***Availability***	***Partition tolerance***
Every read receives the most recent write or an error	Every request receives a (non-error) response – without guarantee that it contains the most recent write	The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes



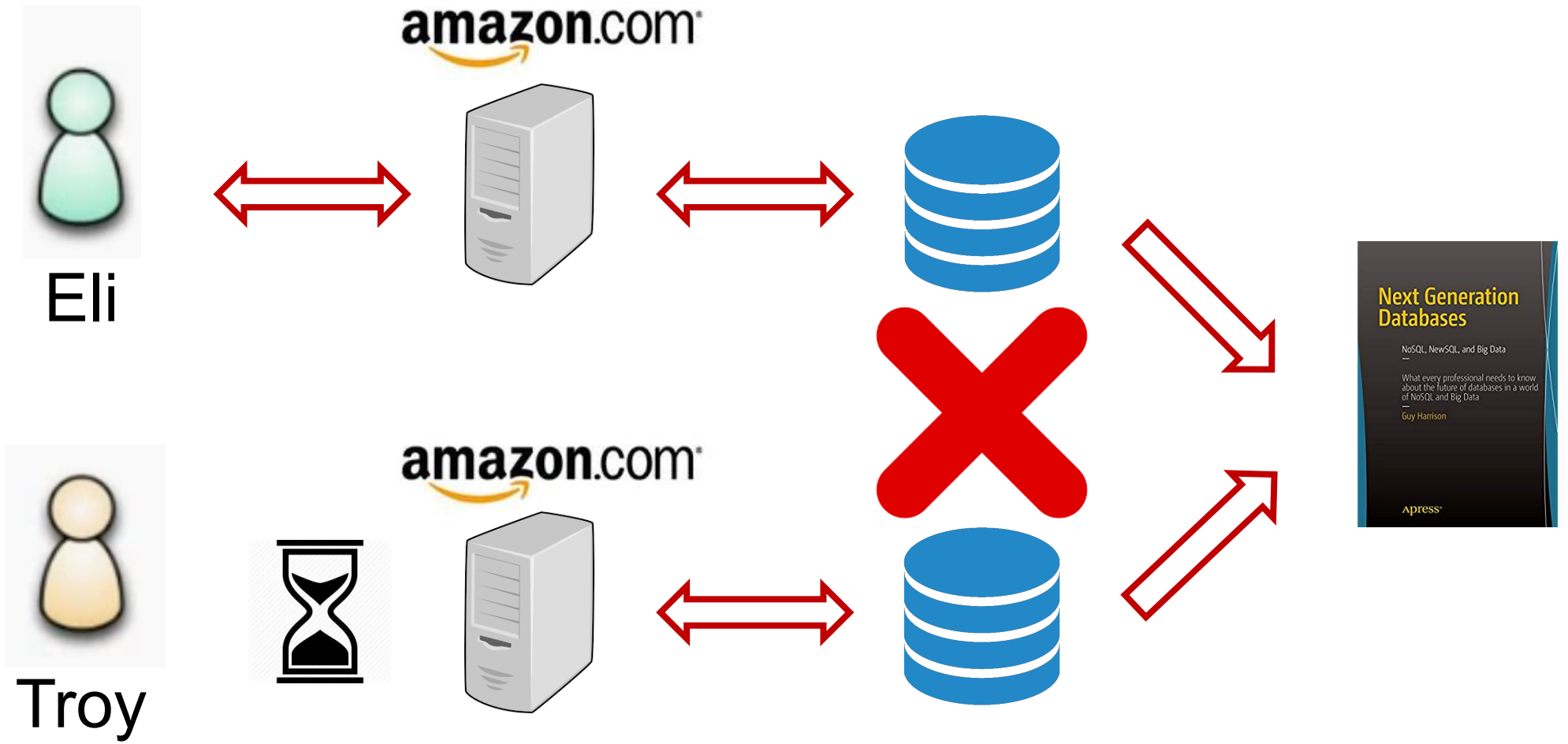
Consistency



Consistency vs. Availability

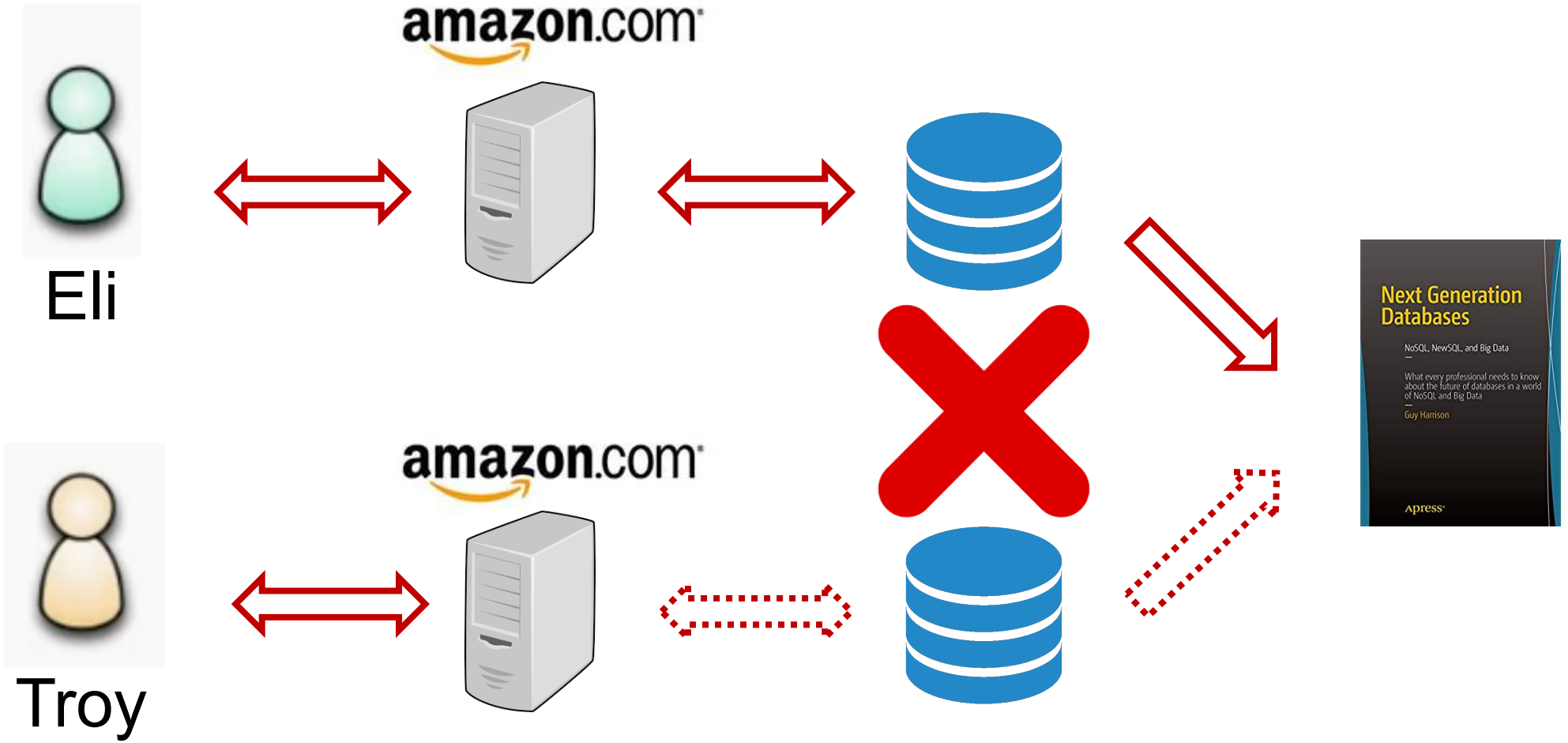


Favor Consistency: Give up availability



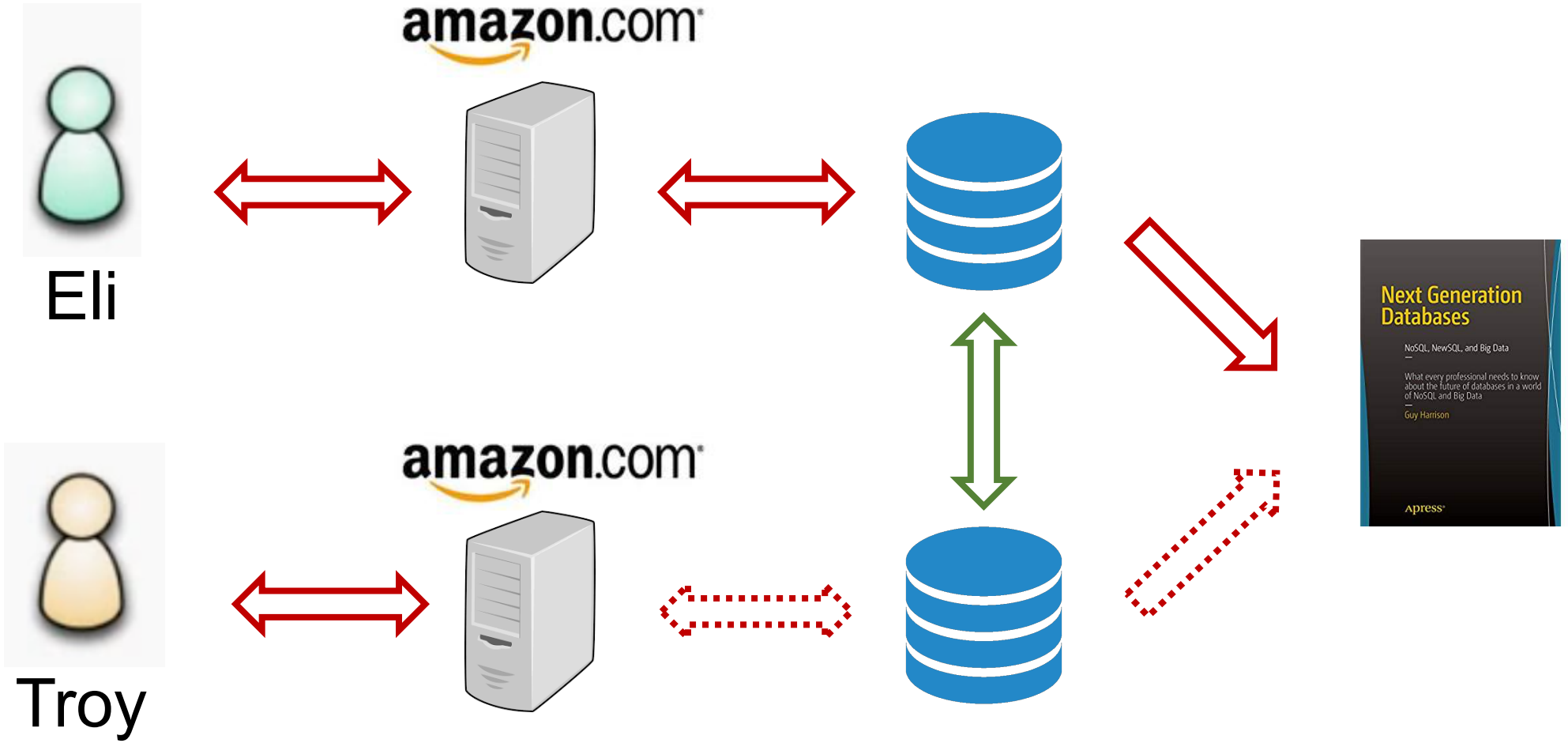
"Please try again later. We want to make sure our information is correct."

Favor Availability: Give up consistency



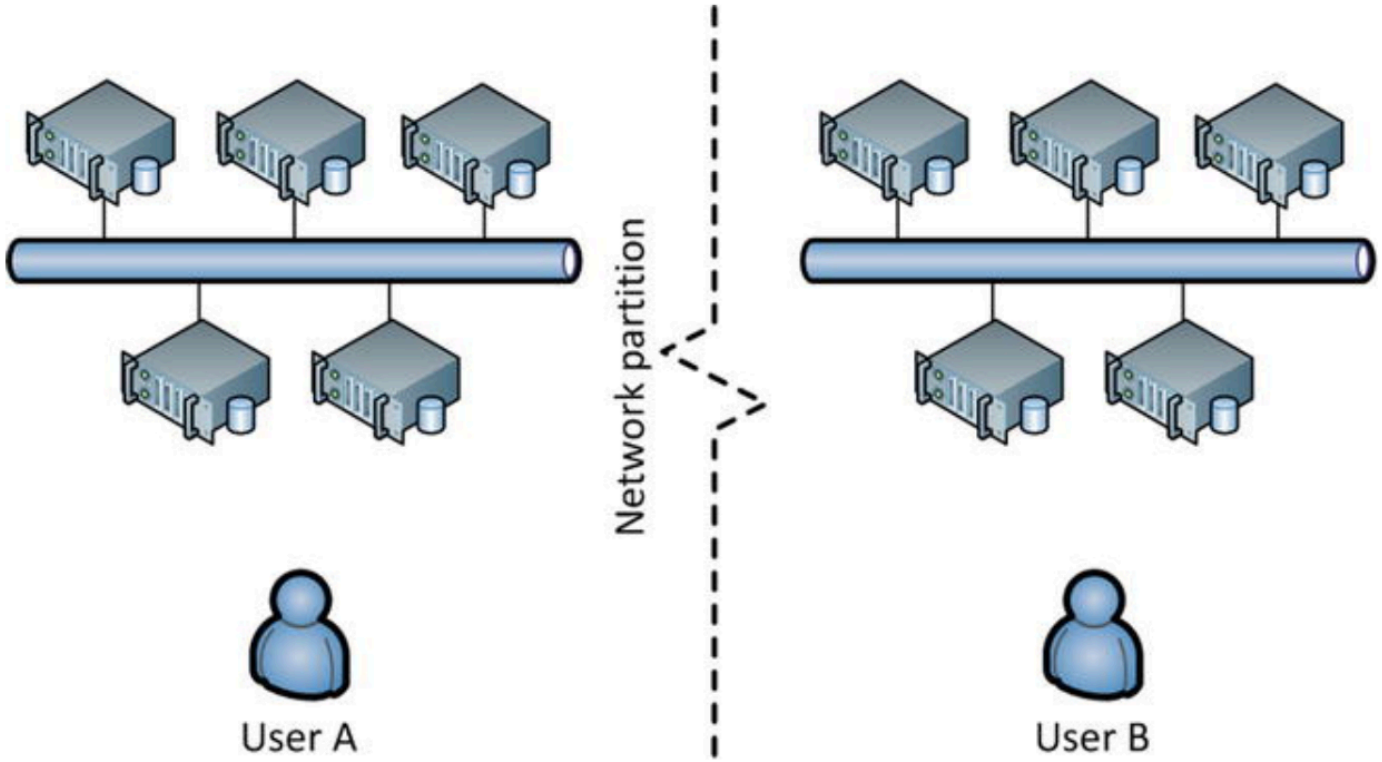
"Last time we checked, we still had the book! Buy it – we'll let you know if there is a delay."

We're consistent AND available!



"We are open for business!
(as long as the network stays alive.)"

Network Partitions



We can:

Favor **consistency** by making the system unavailable for one of the users
(return an error!)

OR

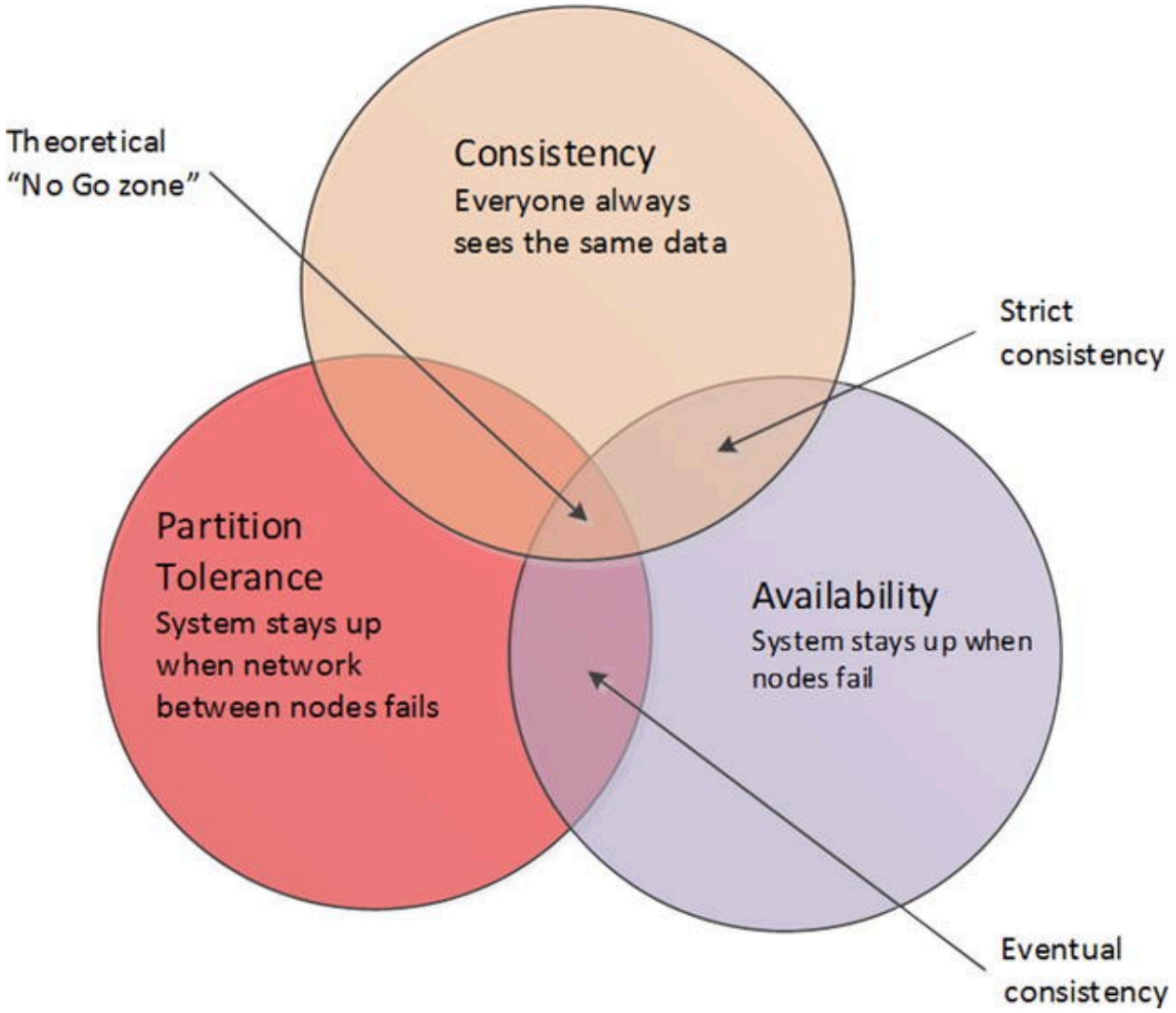
Favor **availability** by allowing each user to have different views of the data.
(return a response!)

The CAP Theorem from a database perspective

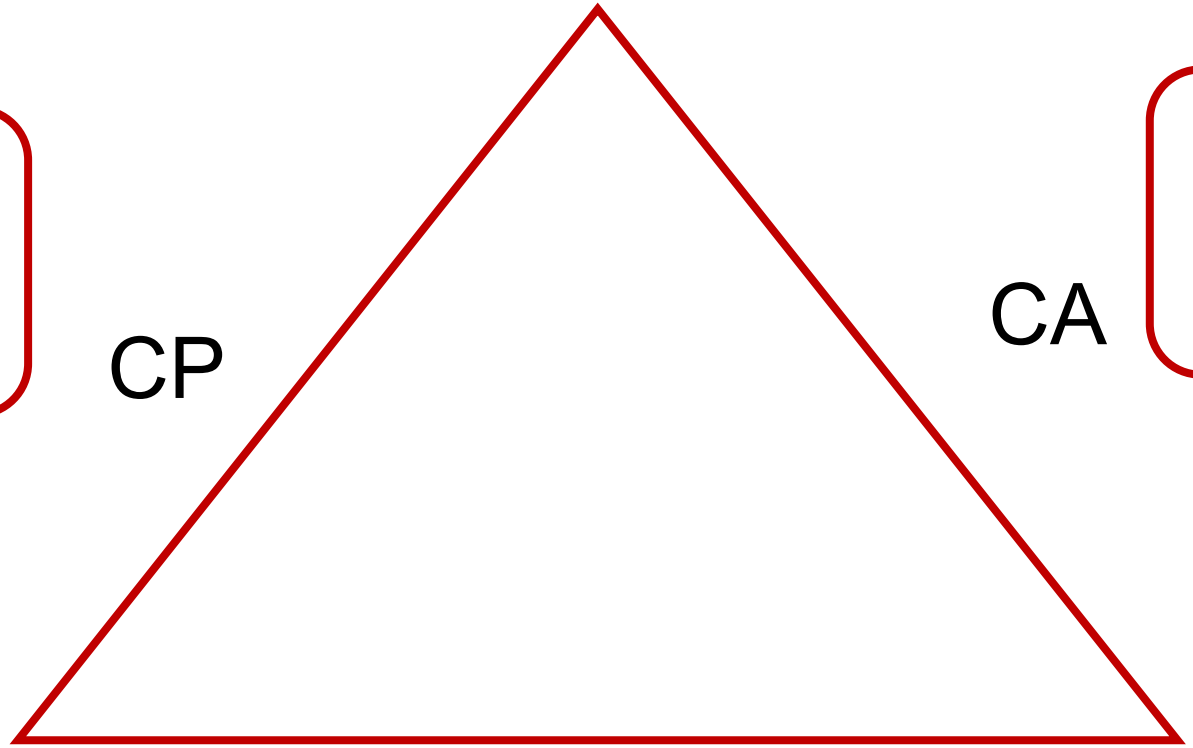
Consistency: Every user of the database has an identical view of the data at any given instant.

Availability*:* In the event of a failure, the database remains operational.

Partition tolerance: The database can maintain operations in the event of the network's failing between two segments of the distributed system.



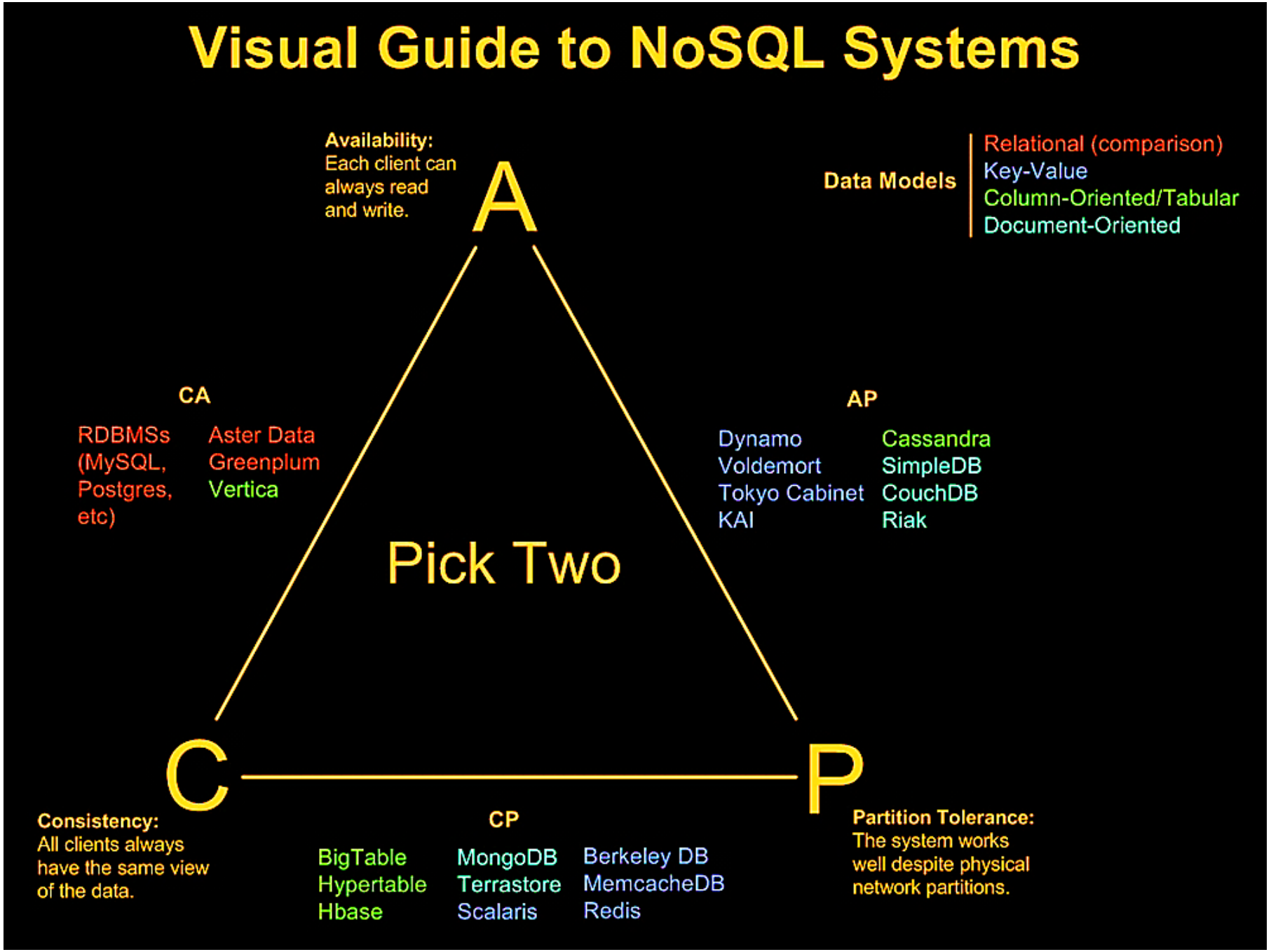
CAP Triangle



MongoDB
Hbase
Memcache
Redis

Oracle
MySQL
SQL Server
Etc.

Cassandra CouchDB
DynamoDB Riak



http://blog.nahurst.com/visual-guide-to-nosql-systems

CAP theorem for NoSQL

What the CAP theorem really says:

If you cannot limit the number of faults and requests can be directed to any server and you insist on serving every request you receive then you cannot possibly be consistent

Eric Brewer 2001

How it is interpreted:

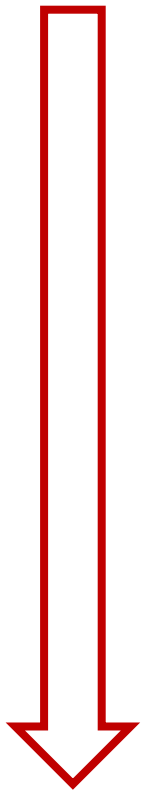
You must always give something up: consistency, availability or tolerance to failure and reconfiguration

Tunable Consistency

Levels of Consistency

Database consistency is concerned with guaranteeing that a database always appears truthful to the user. Every operation must take the database from one consistent state to another.

- **Strict**: Changes to data are atomic and appear instantaneous
- **Sequential**: Clients see changes in the same order they were applied
- **Causal**: All causally-related changes are observed in the same order by all clients
- **Eventual**: All updates will eventually propagate through the system and all replicas will be consistent
- **Weak**: No propagation guarantees. Changes may appear out of order for some clients

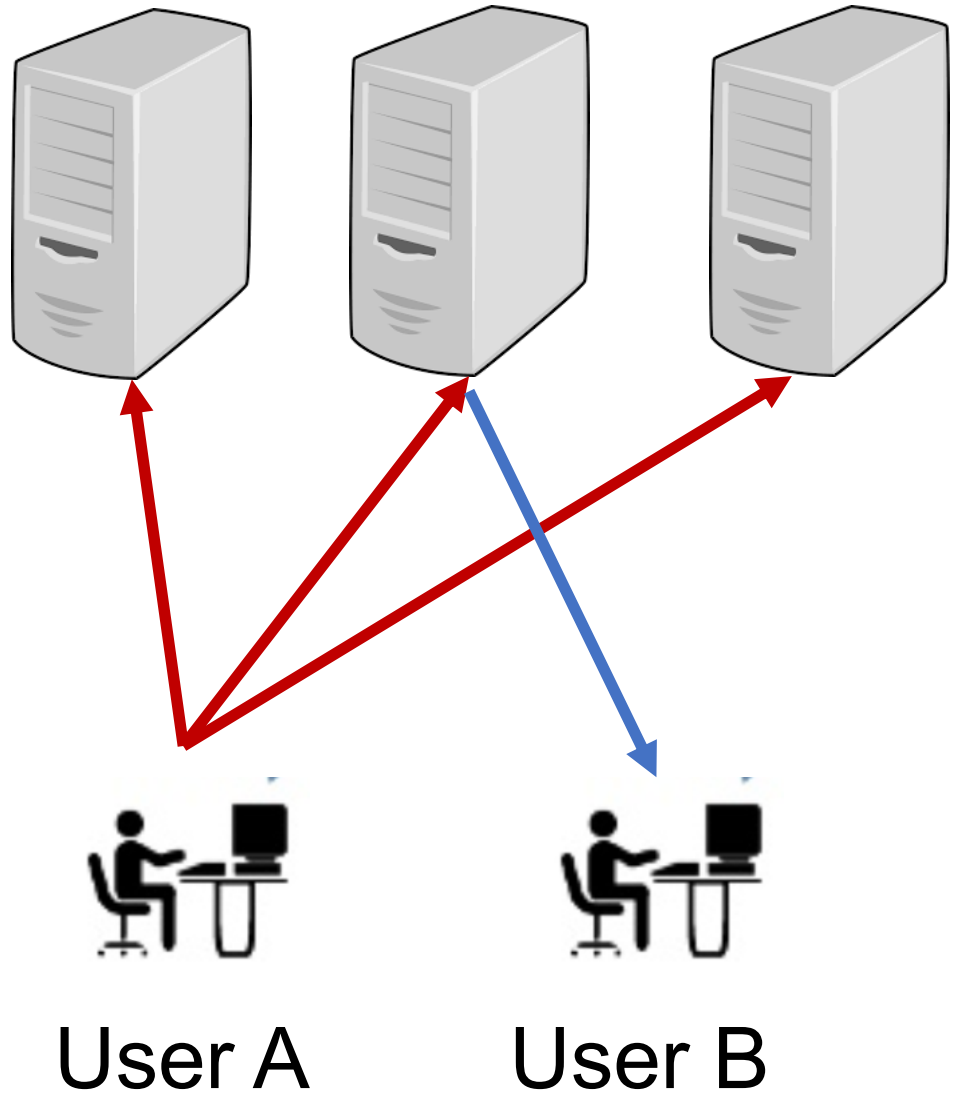


Tunable Consistency (NWR Notation)

N is the number of copies of each data item that the database will maintain.

W is the number of copies of the data item that must be written before the write can complete and control is returned to the application.

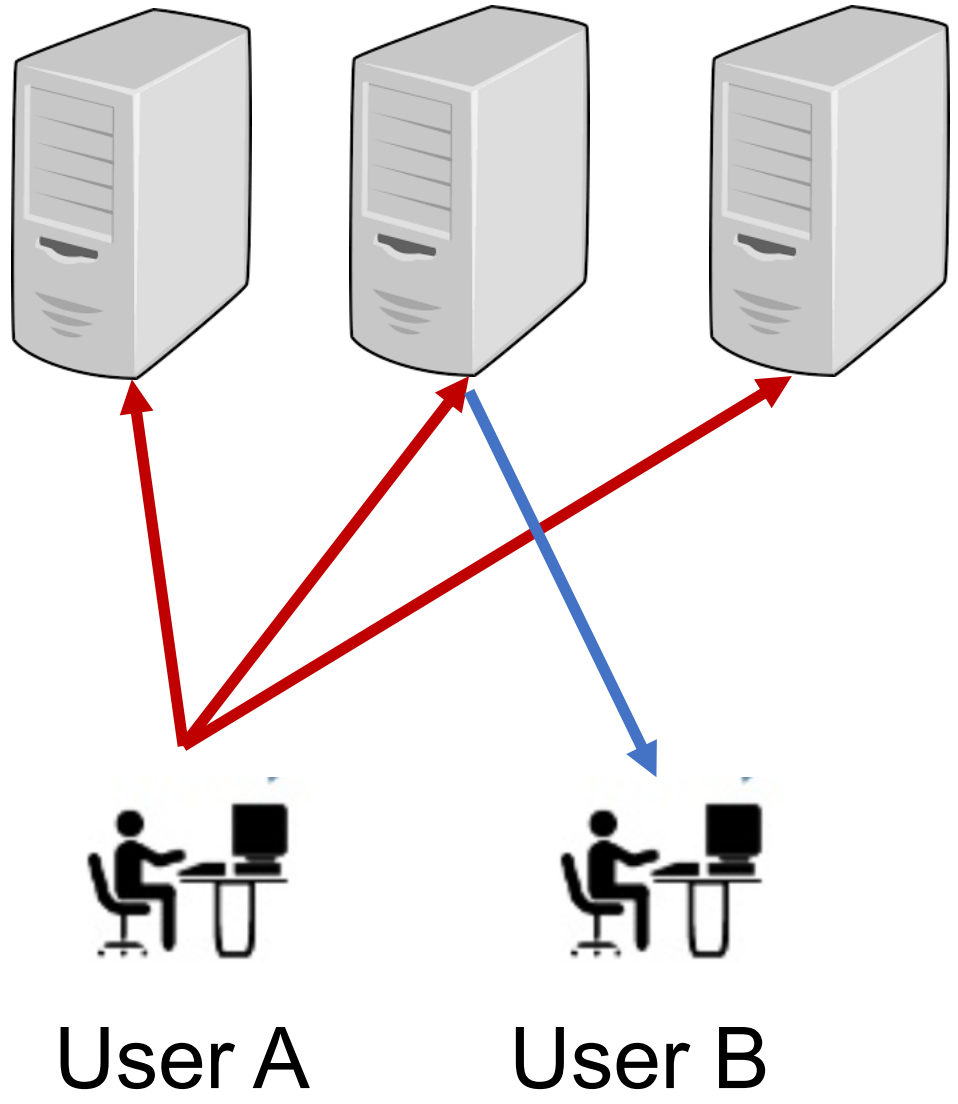
R is the number of copies that the application will access when reading the data item.



Tunable Consistency (W = N, R = 1)

N = 3
W = 3
R = 1

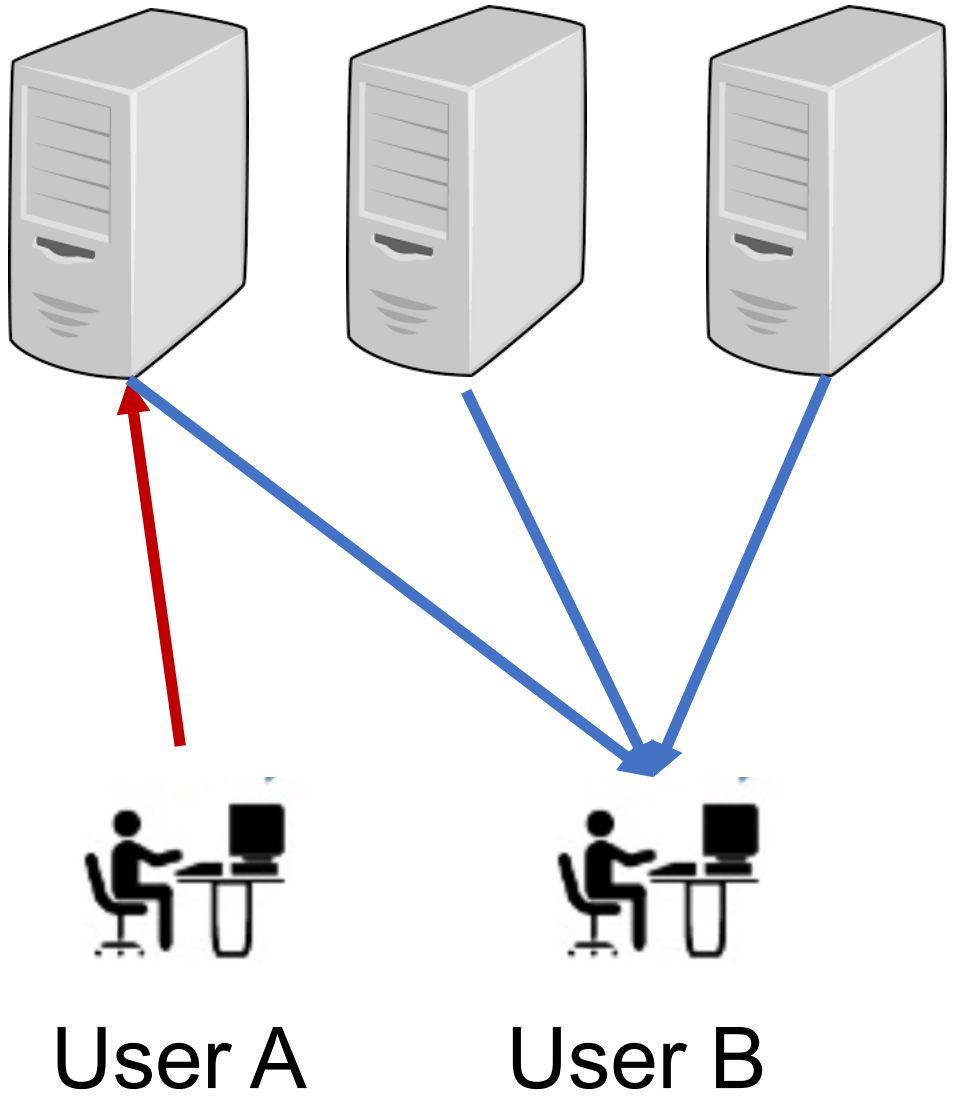
- Slow writes, Fast reads
- Consistent
- “Synchronous replication”



Tunable Consistency (W = 1, R = N)

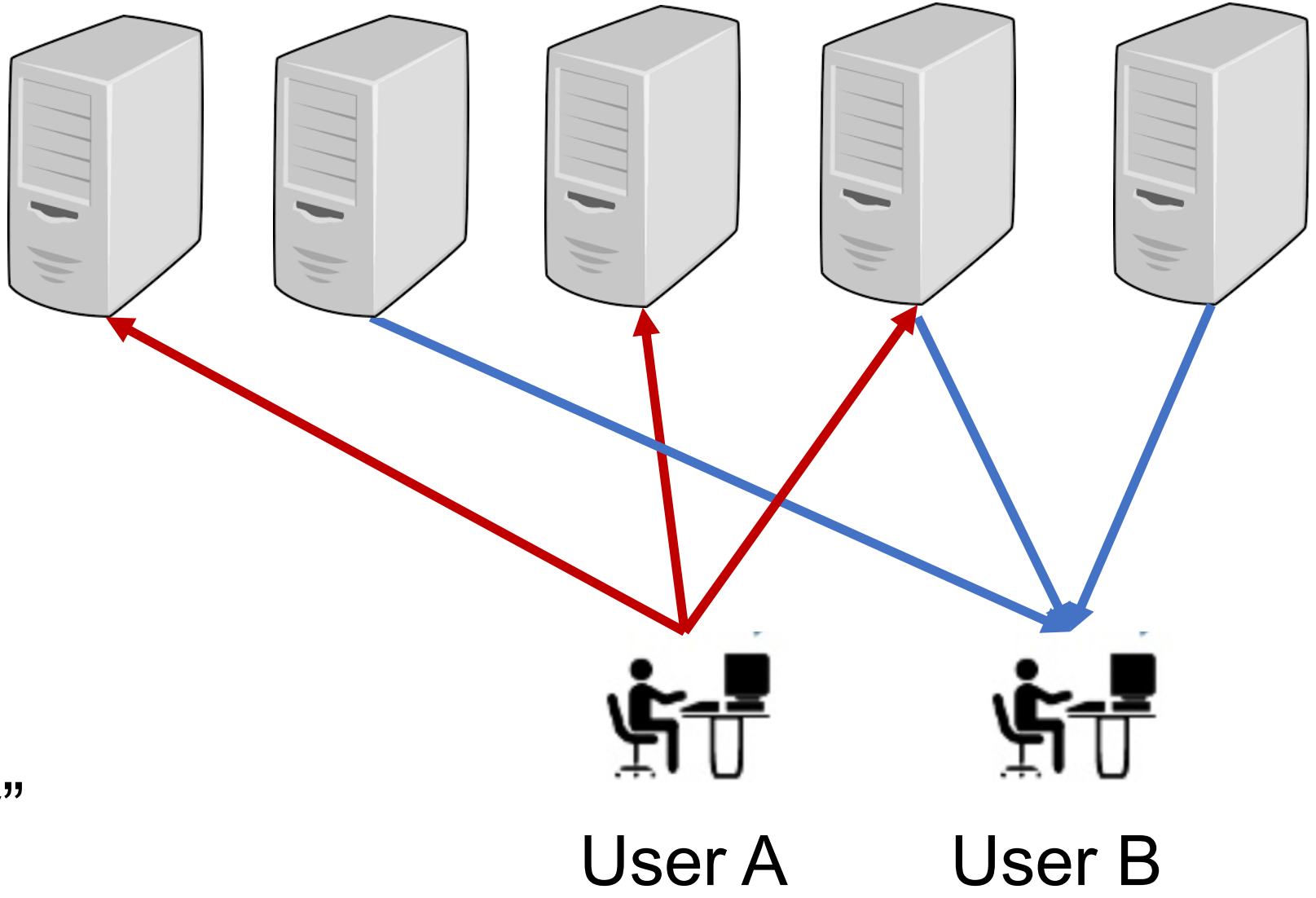
N = 3
W = 1
R = 3

- Fast writes, Slow reads
- Consistent



Tunable Consistency (W + R > N)

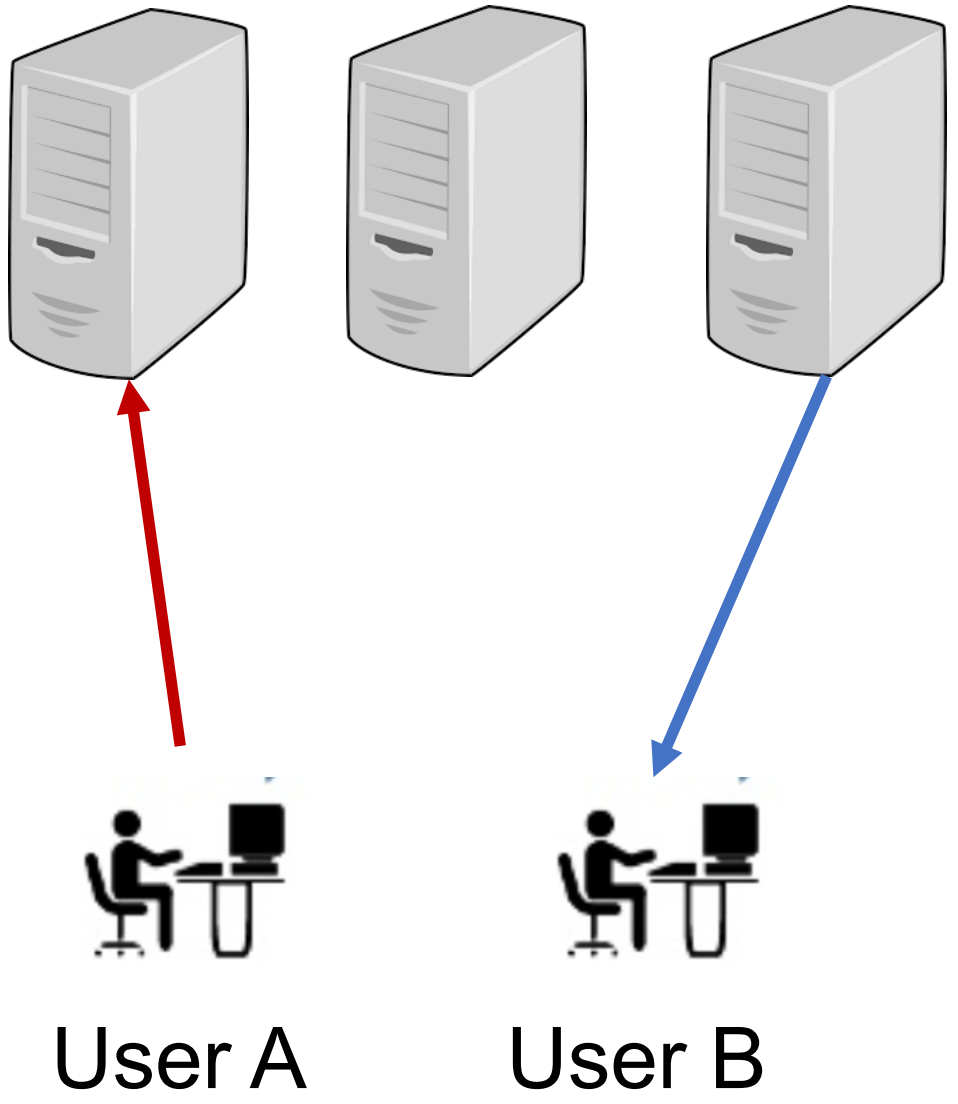
e.g.,
N = 5
W = 3
R = 3



- Mixed Read/Write Performance
- Consistent
- “Quorum Assembly”

Tunable Consistency (W = 1, R = 1)

N = 3
W = 1
R = 1



- Fast writes, Fast reads
- Eventual consistency
- Strict consistency lost

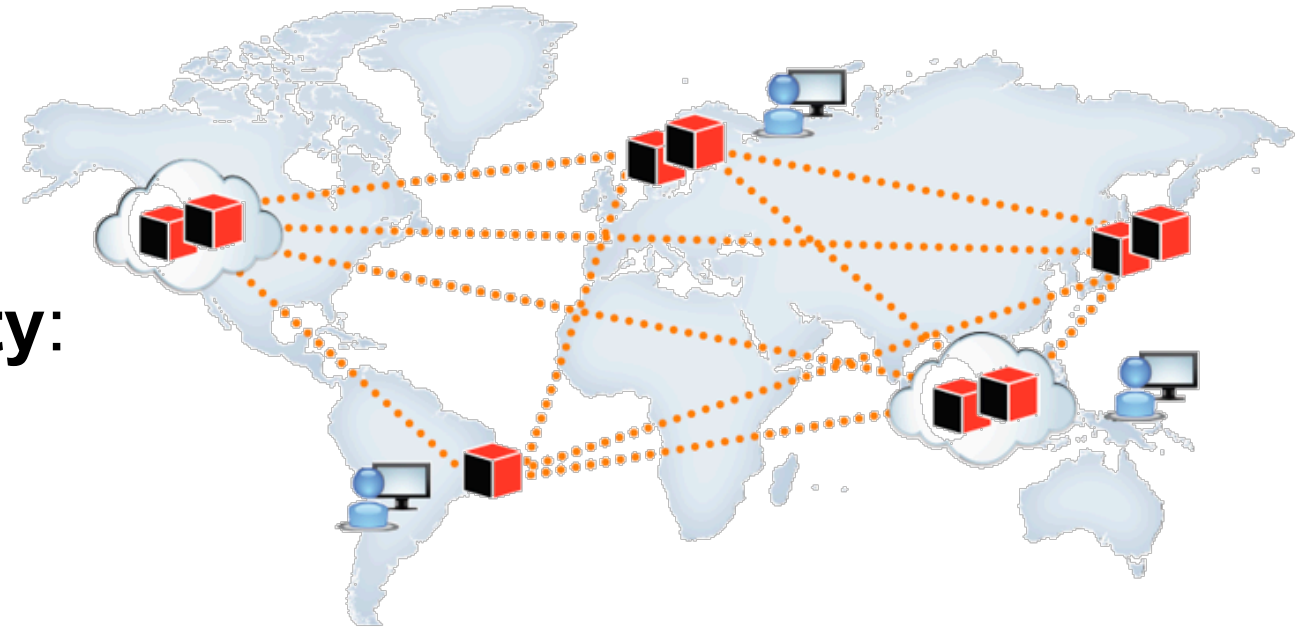
Replication

Benefits of Distributed Data

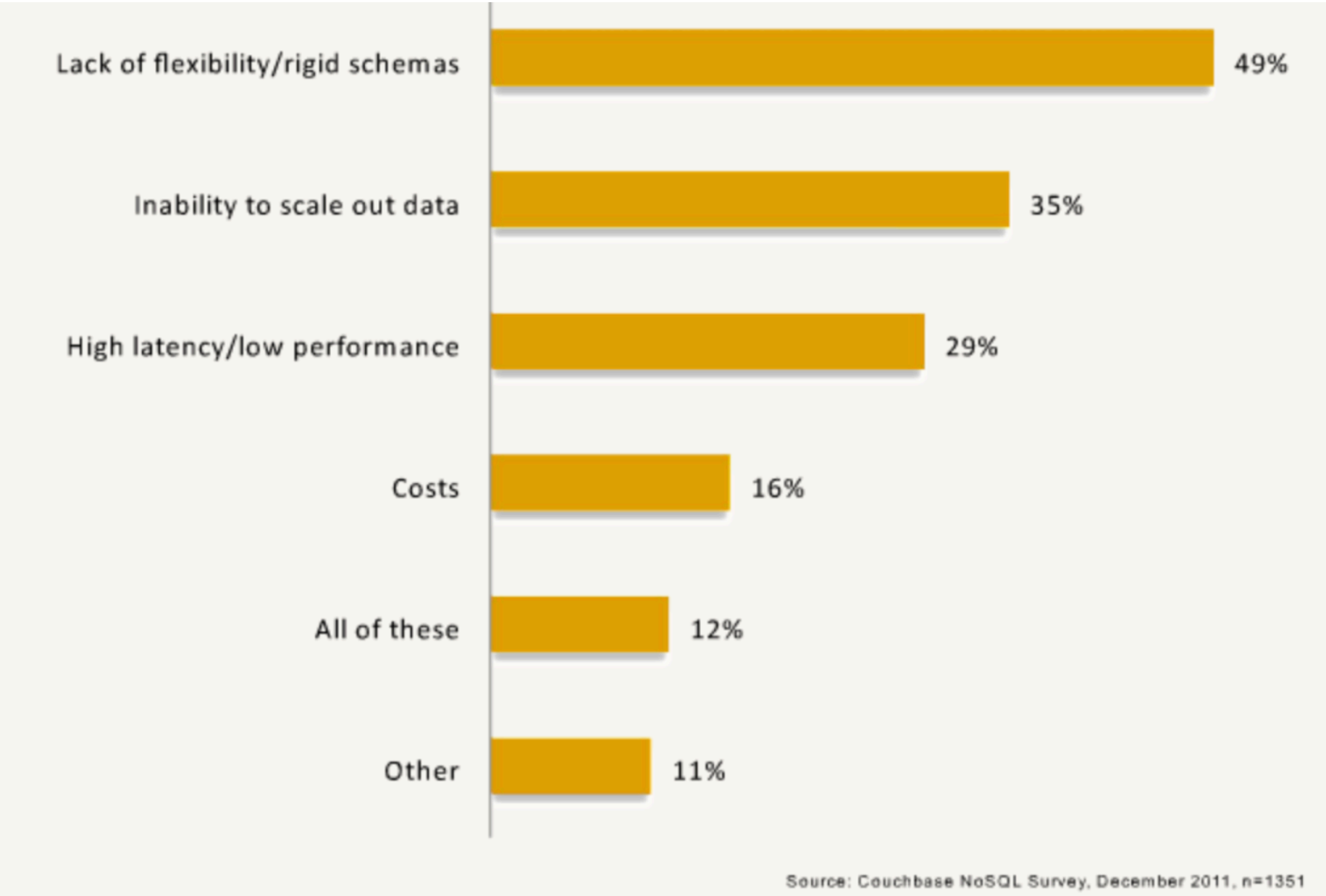
- **Scalability / High throughput**: Data volume or Read/Write load grows beyond the capacity of a single machine

- **Fault Tolerance / High Availability**: Your application needs to continue working even if one or more machines goes down.

- **Latency**: When you have users in different parts of the world you want to give them fast performance too.

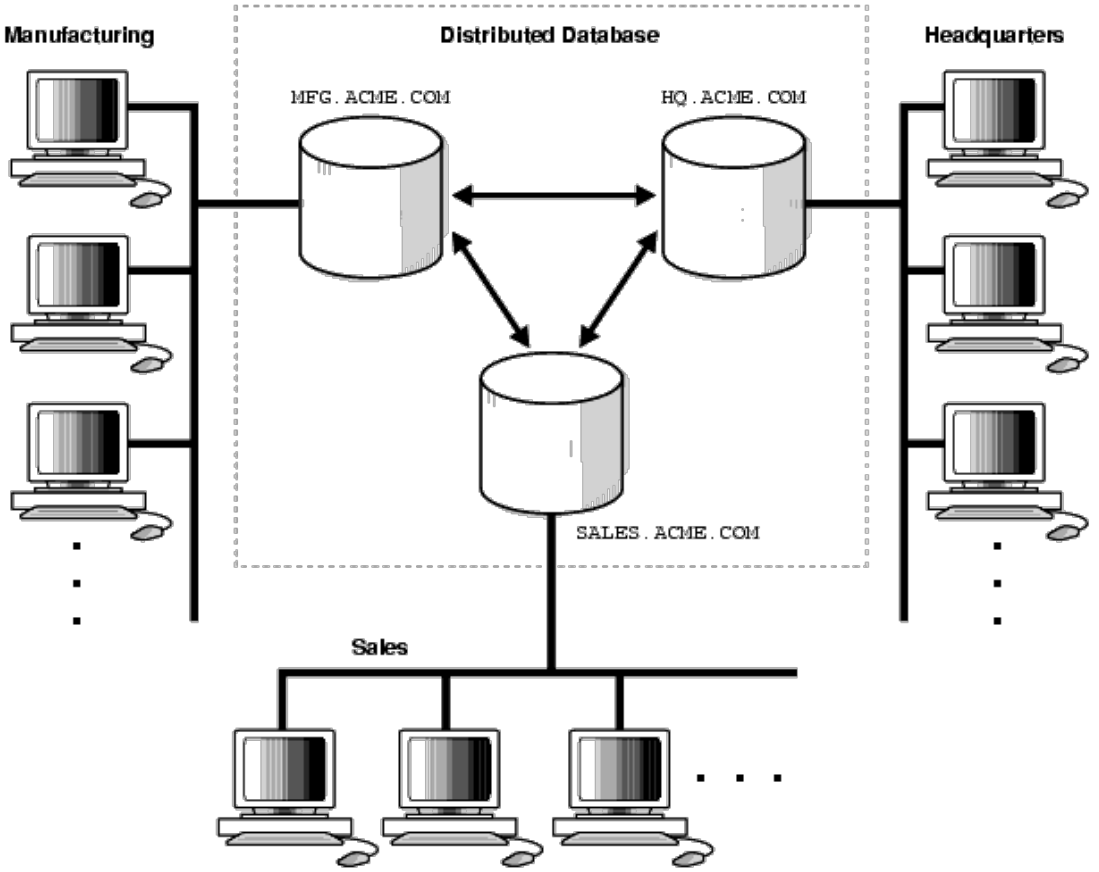


Drivers of NoSQL



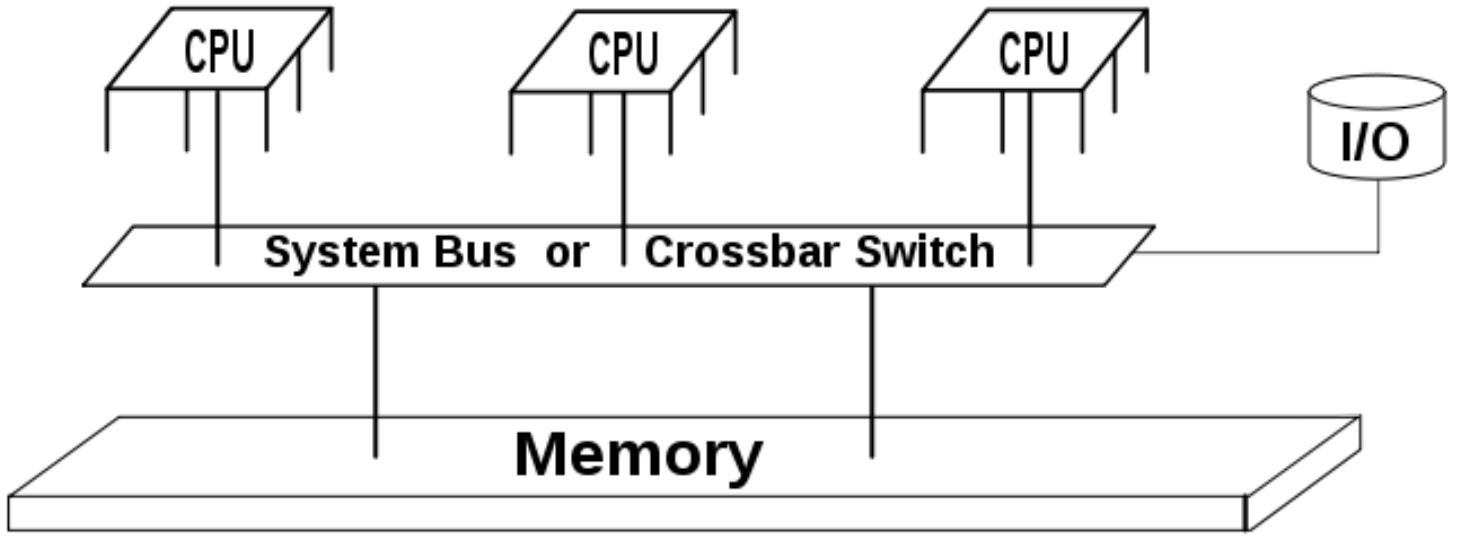
Challenges of Distributed Data

- **Consistency**: Updates have to be propagated across the network.
- **Application Complexity**: Responsibility for reading and writing data in a distributed environment often falls to the application.



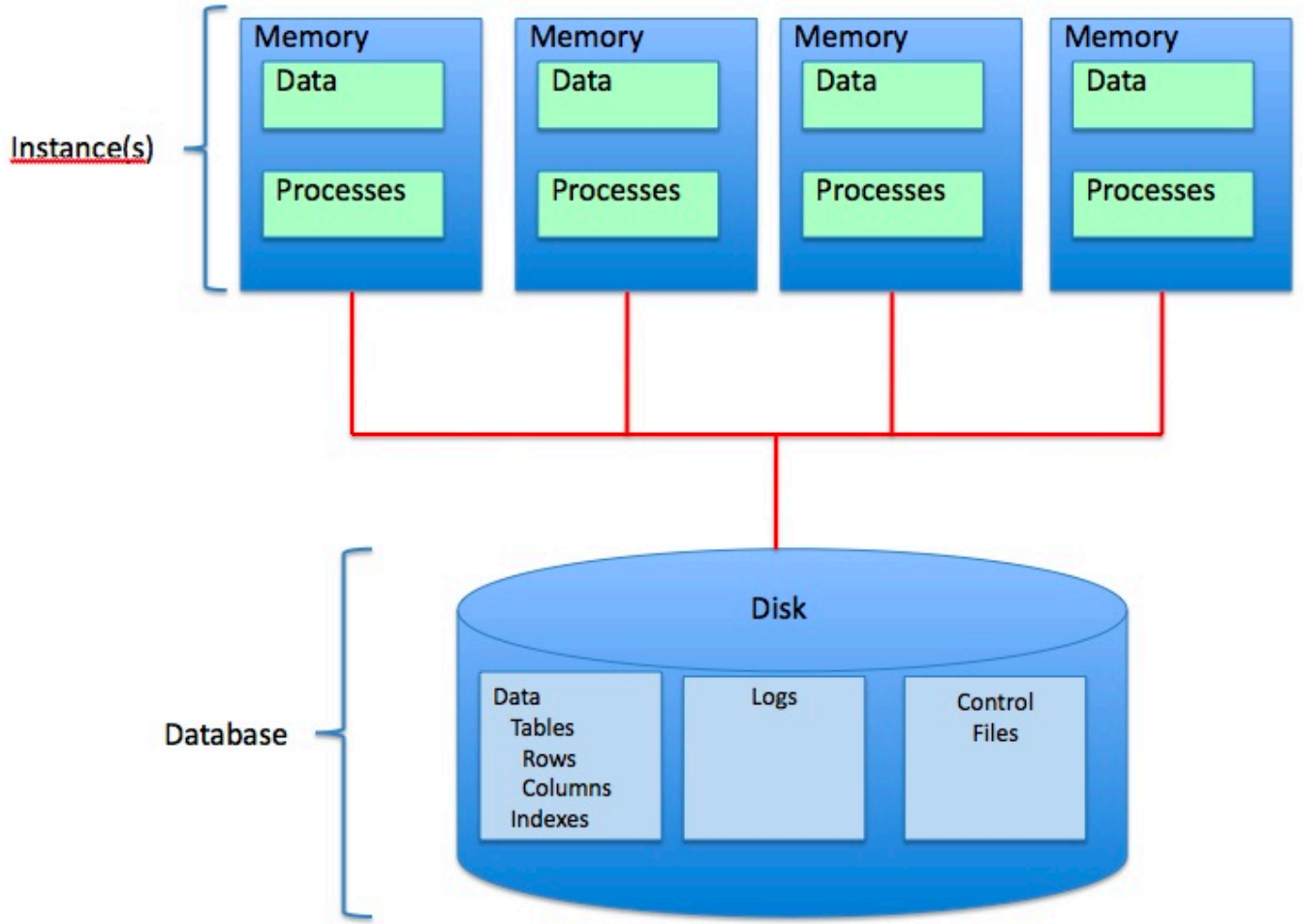
Shared-memory architectures (Vertical scaling)

- Geographically Centralized server
- Some fault tolerance (via hot-swappable components)



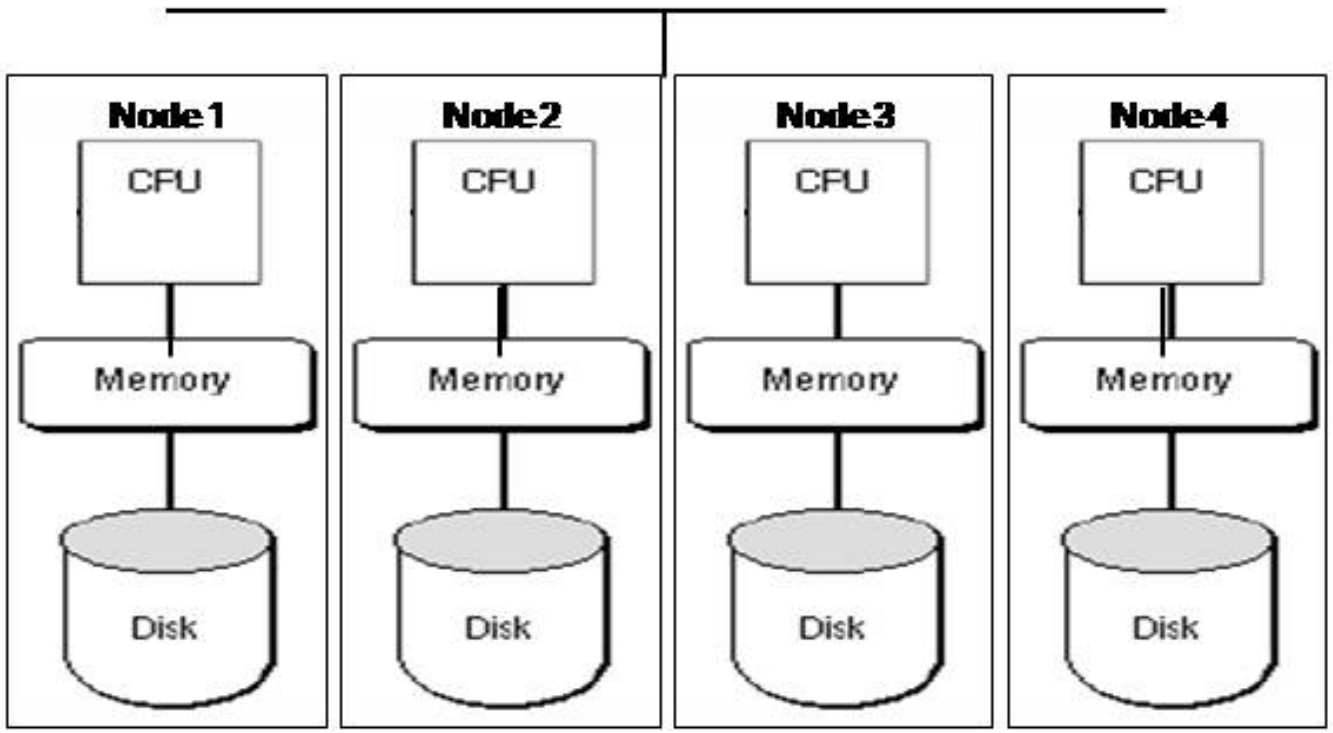
Shared-disk architectures (Vertical scaling)

- Machines are connected via a fast network
- Contention and the overhead of locking limit scalability (high-write volumes) but ok for Data Warehouse applications (high read volumes)
- Oracle RAC (Real Application Cluster)



Shared-nothing architectures (horizontal scaling)

- Each node has its own CPU, memory, and disk
- Coordination via application layer using conventional network
- Geographically distributed
- Commodity hardware

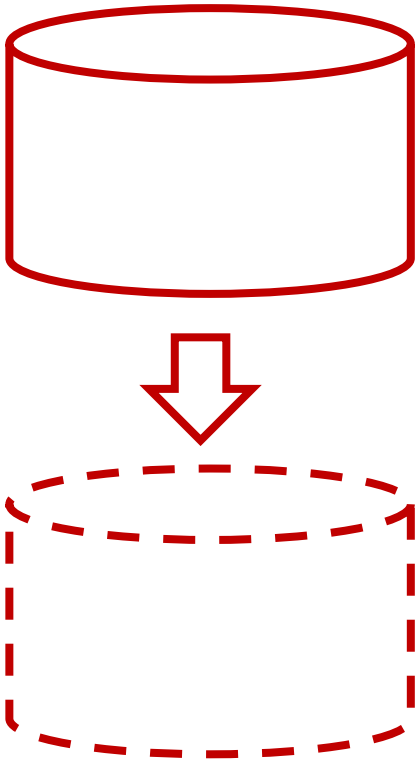


Common Algorithms for Replication

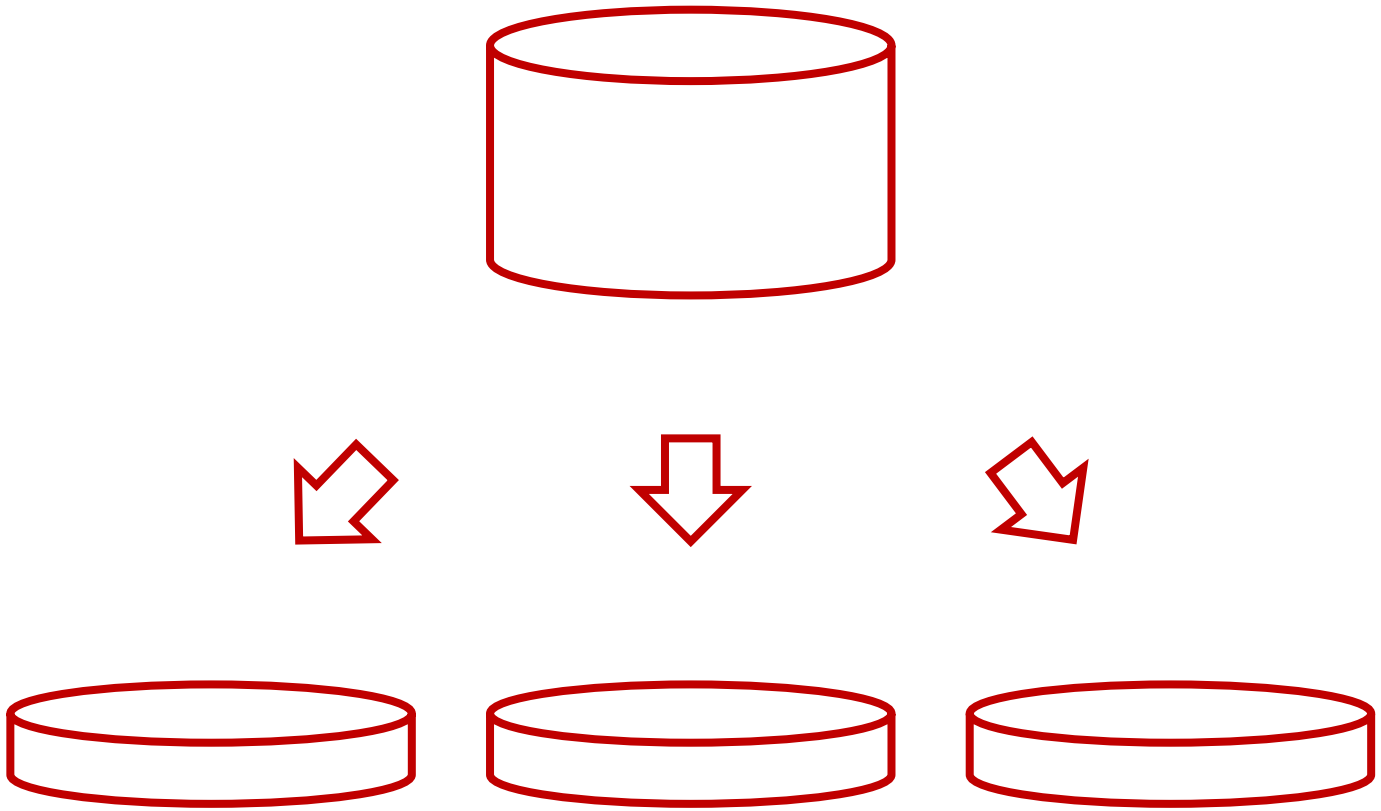
- Single leader
- Multiple leader
- Leaderless

Most distributed databases use one of these approaches. They each have tradeoffs.

Replication vs. Partitioning / “Sharding”

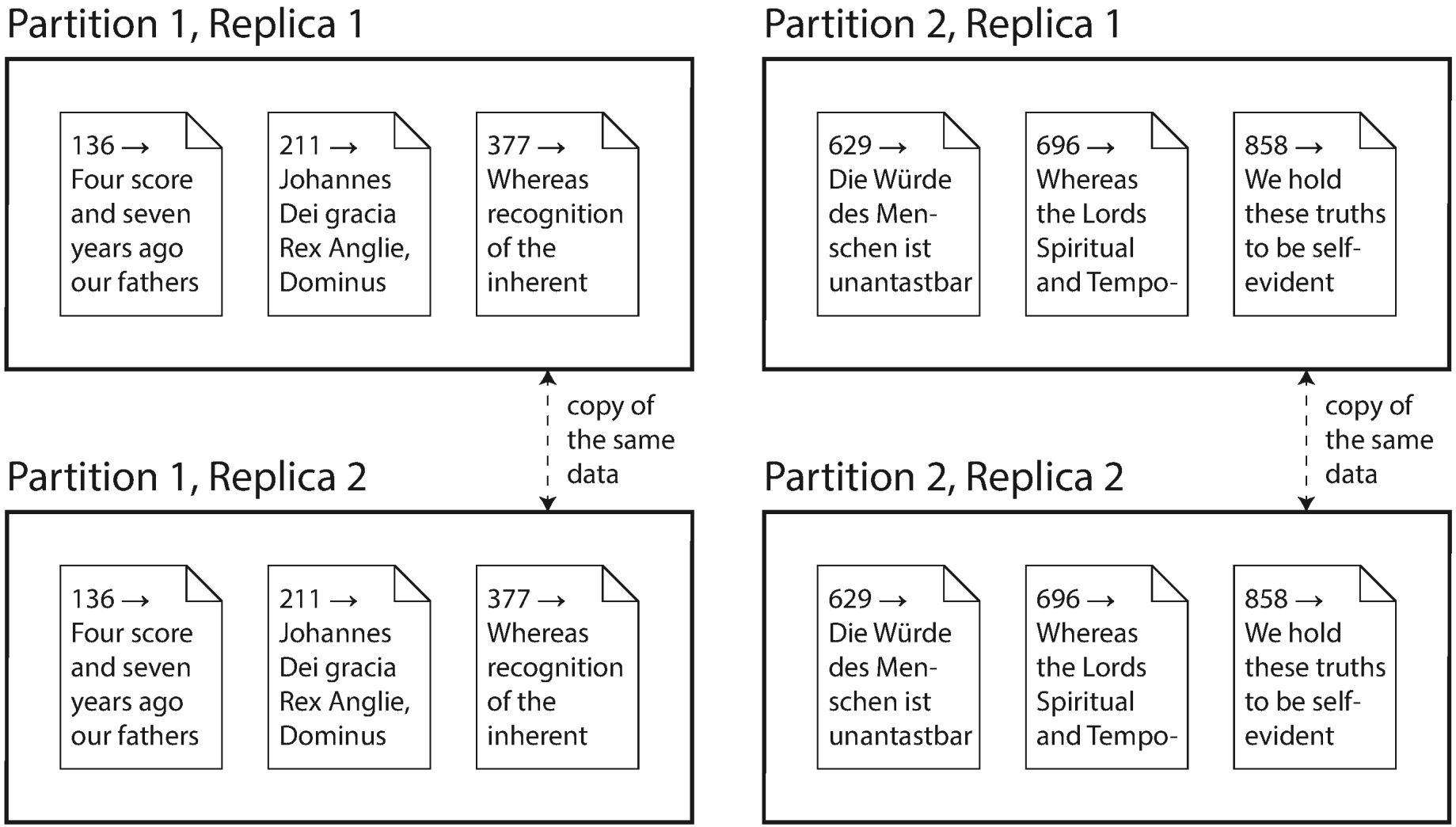


Each node holds a copy of the whole DB

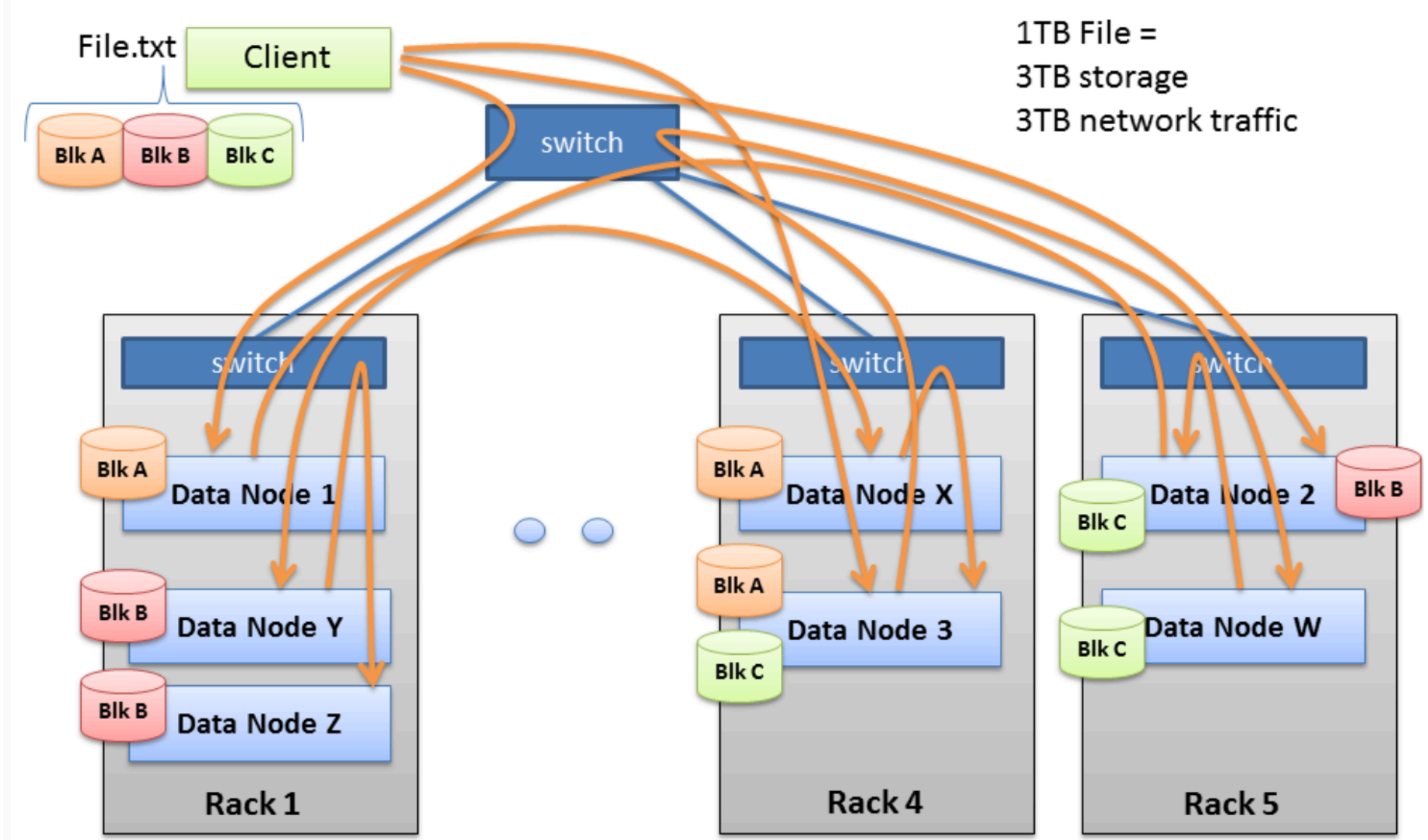


Parts of the DB are spread across nodes

Combining replication AND partitioning is common

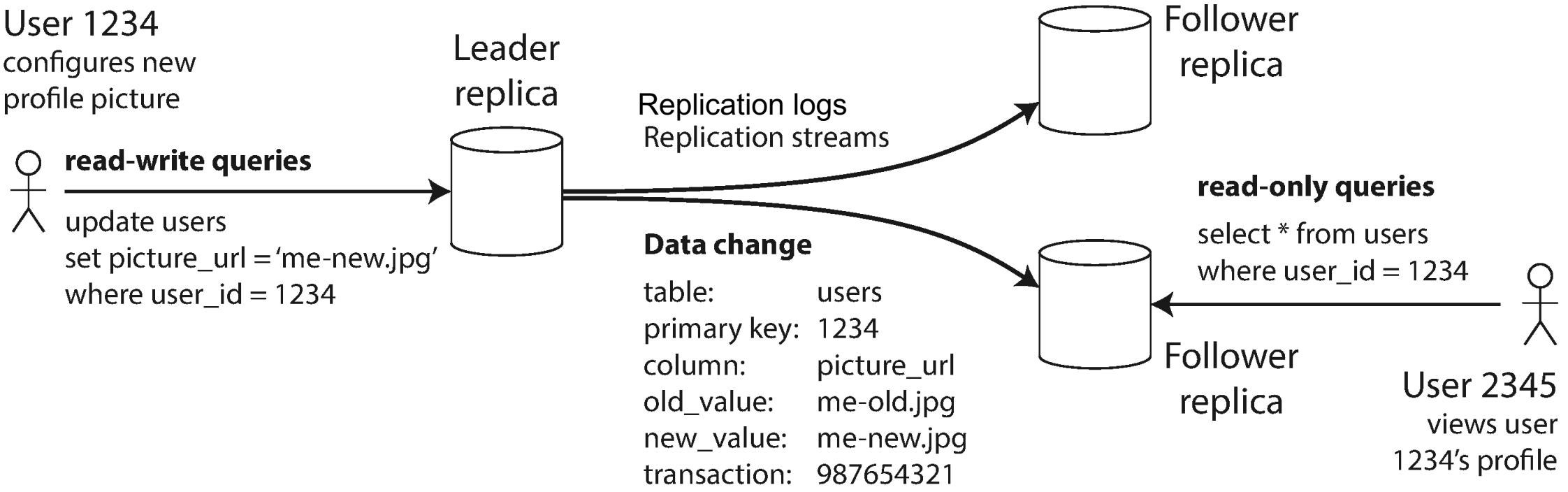


Hadoop Distributed File System (HDFS)



Leader-based replication (aka active/passive or master/slave)

1. Write to leader (ONLY)
2. Leader sends replication stream to followers
3. Followers process stream
4. Clients read from leader or any follower



Who uses leader based replication?

Relational: MySQL, Oracle, SQL Server, PostgreSQL

NoSQL:

MongoDB,
RethinkDB (realtime web apps),
Expresso (LinkedIn)

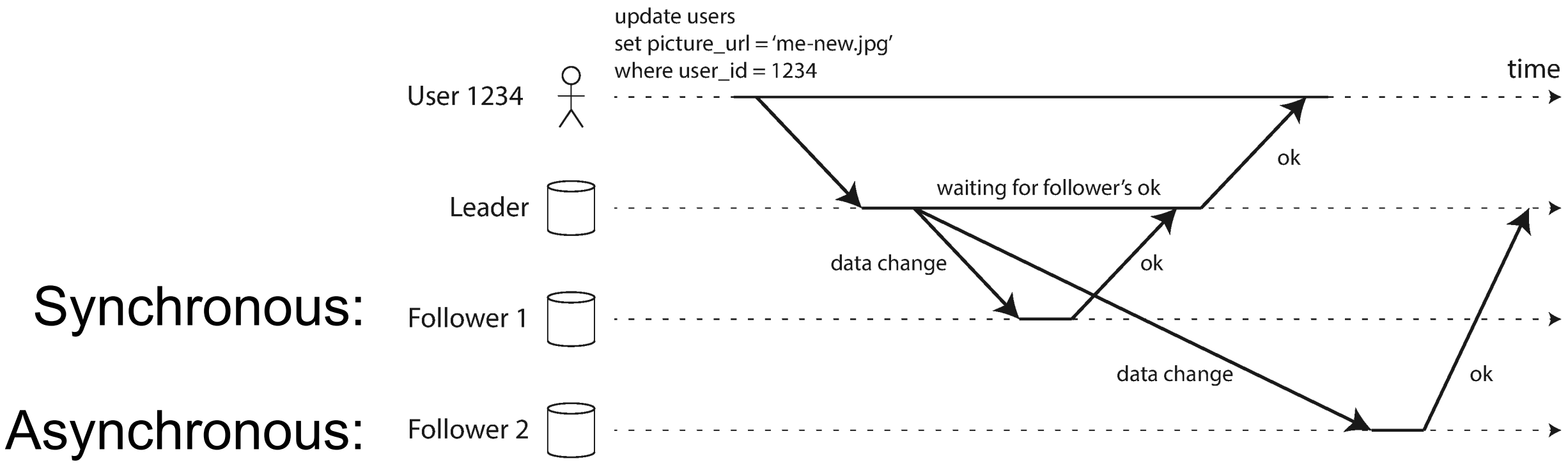
Messaging Brokers: Kafka, RabbitMQ

Replication Methods

Replication Method	Description
Statement-based	Send INSERT, UPDATE, DELETEs to replica. Simple but error-prone due to non-deterministic functions like now(), trigger side-effects, and difficulty in handling concurrent transactions.
Write-ahead Log (WAL)	A byte-level specific log of every change to the database. Leader and all followers must implement the same storage engine and makes upgrades difficult.
Logical (row-based) Log	For relational DBs: Inserted rows, modified rows (before and after), deleted rows. A transaction log will identify all the rows that changed in each transaction and how they changed. Logical logs are decoupled from the storage engine and easier to parse.
Trigger-based	Changes are logged to a separate table whenever a trigger fires in response to an insert, update, or delete. Flexible because you can have application specific replication, but also more error prone.

Synchronous vs. Asynchronous Replication

Synchronous: Leader waits for a response from the follower
Asynchronous: Leader doesn't wait for confirmation.



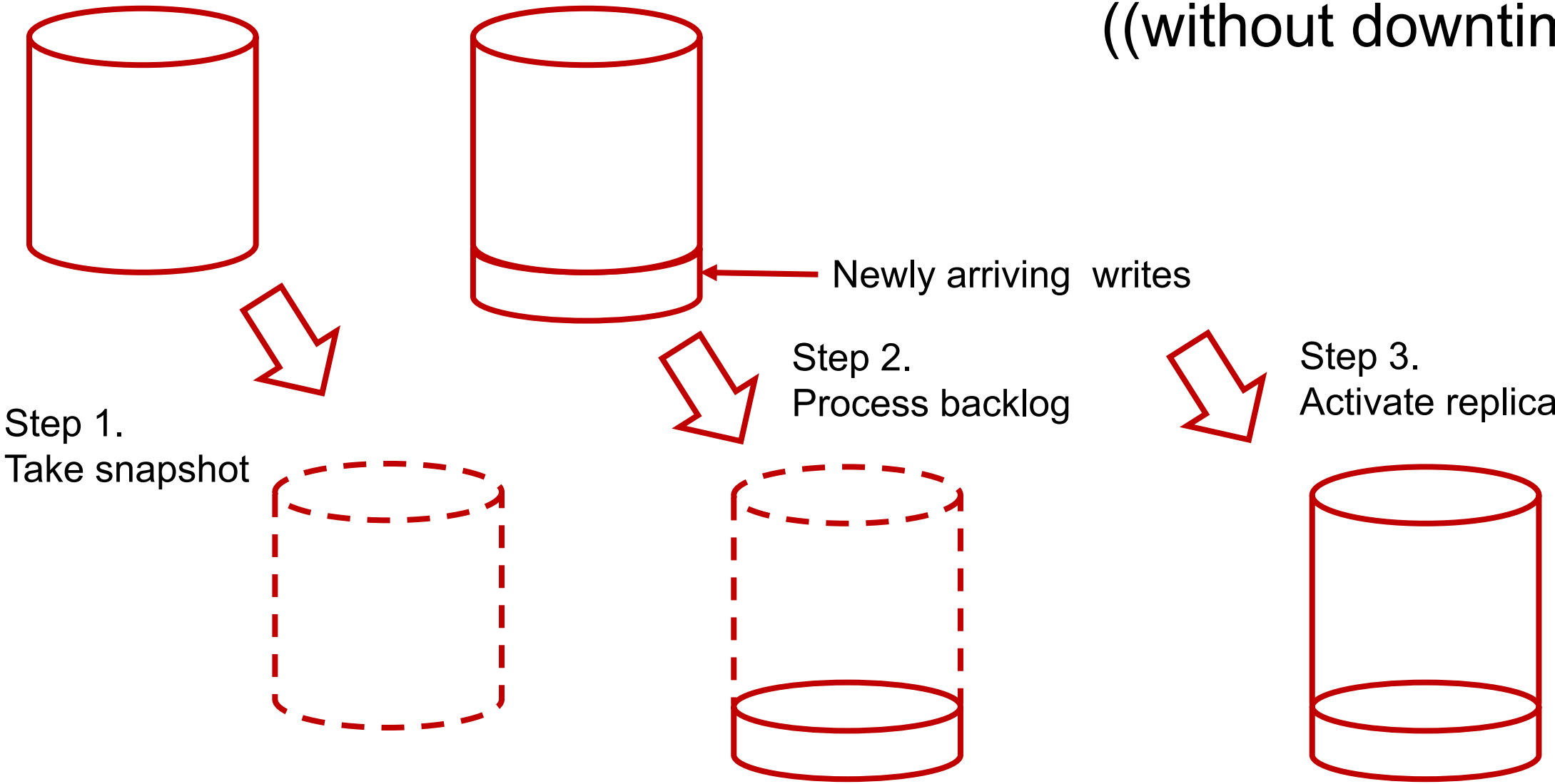
Synchronous vs. Asynchronous

	Synchronous	Asynchronous
Leader waits for response?	**YES**	**NO**
Guaranteed Consistency	**YES** – We confirm writes to every follower	**NO** – We proceed without confirming that the update was received. It might not have been!
Resilient to Follower Failure?	**NO** – If we demand a response / acknowledgement we might be endlessly waiting due to a node failure	**YES** – We accept that the replica might not respond if it's failed and might therefore be out-of-date.
Resilient to Leader Failure?	**YES** – A follower can be made the new Leader	**NO** – We might have lost some writes. Writes are not durable.

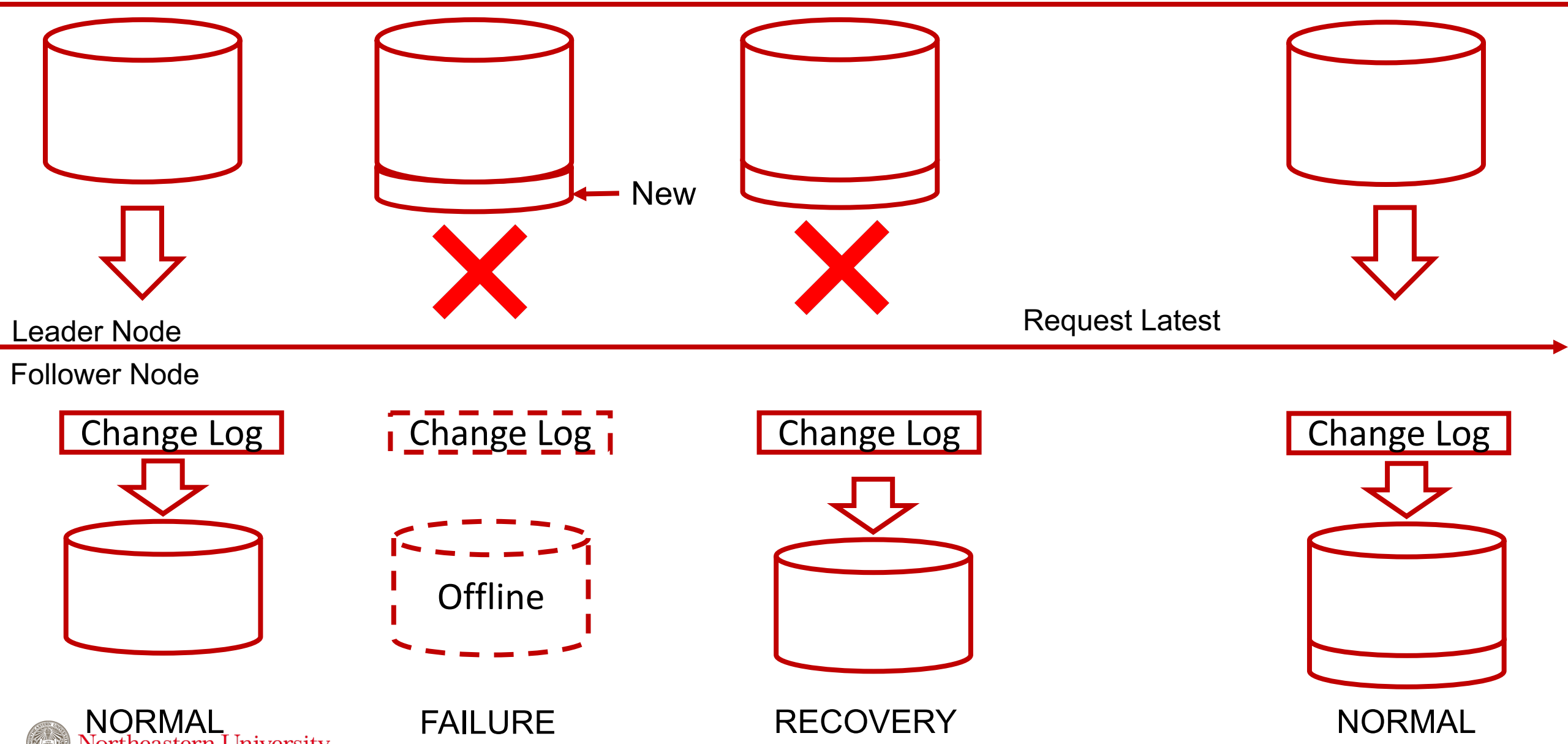
Semi-Synchronous: Designate one follower as synchronous and the rest as asynchronous. If synchronous follower fails another follower takes its place.

Adding new replicas (and keeping everything consistent)

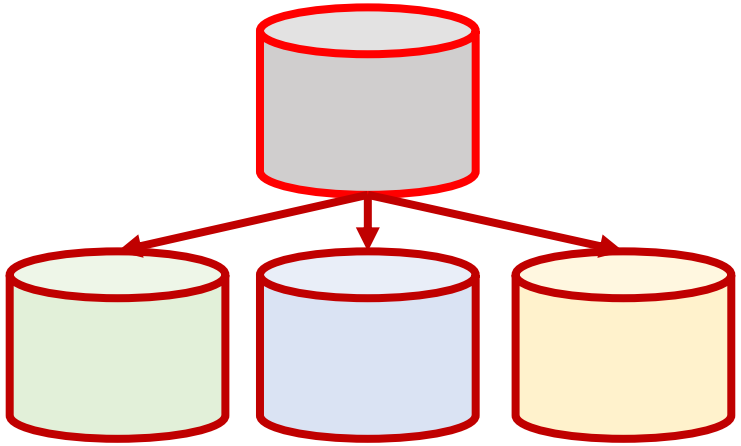
((without downtime!))

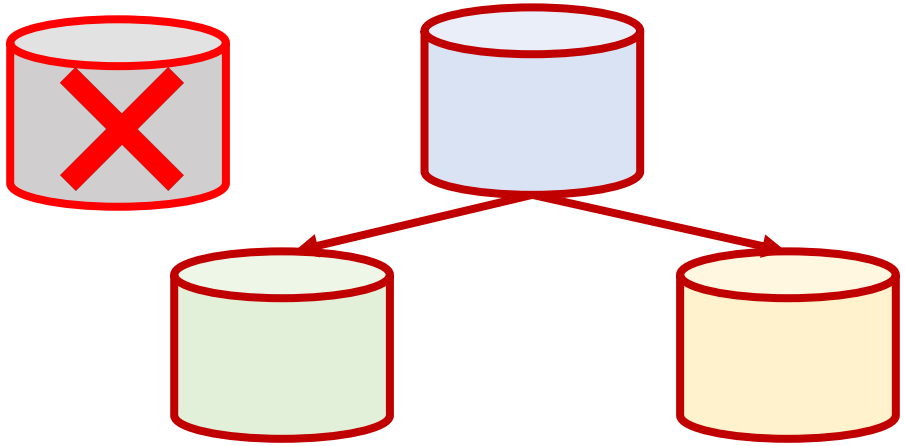


Follower Failure. (Recovery)



Leader Failure. (Failover)





Challenges:

a) How do we pick a new leader?
   - Consensus strategy – perhaps based on who has the most updates?
   - Use a controller node to appoint new leader

b) How do we configure clients to start writing to the new leader?

Leader Failure. (Failover)

More Challenges:

a) If asynchronous replication is used, new leader may not have all the writes
How do we recover the lost writes? Or do we simply discard?

b) After the old leader recovers how do we avoid having multiple leaders receiving conflicting data? (Split brain: no way to resolve conflicting requests.)

c) Leader failure detection. Optimal timeout is tricky.

Replication Lag has implications

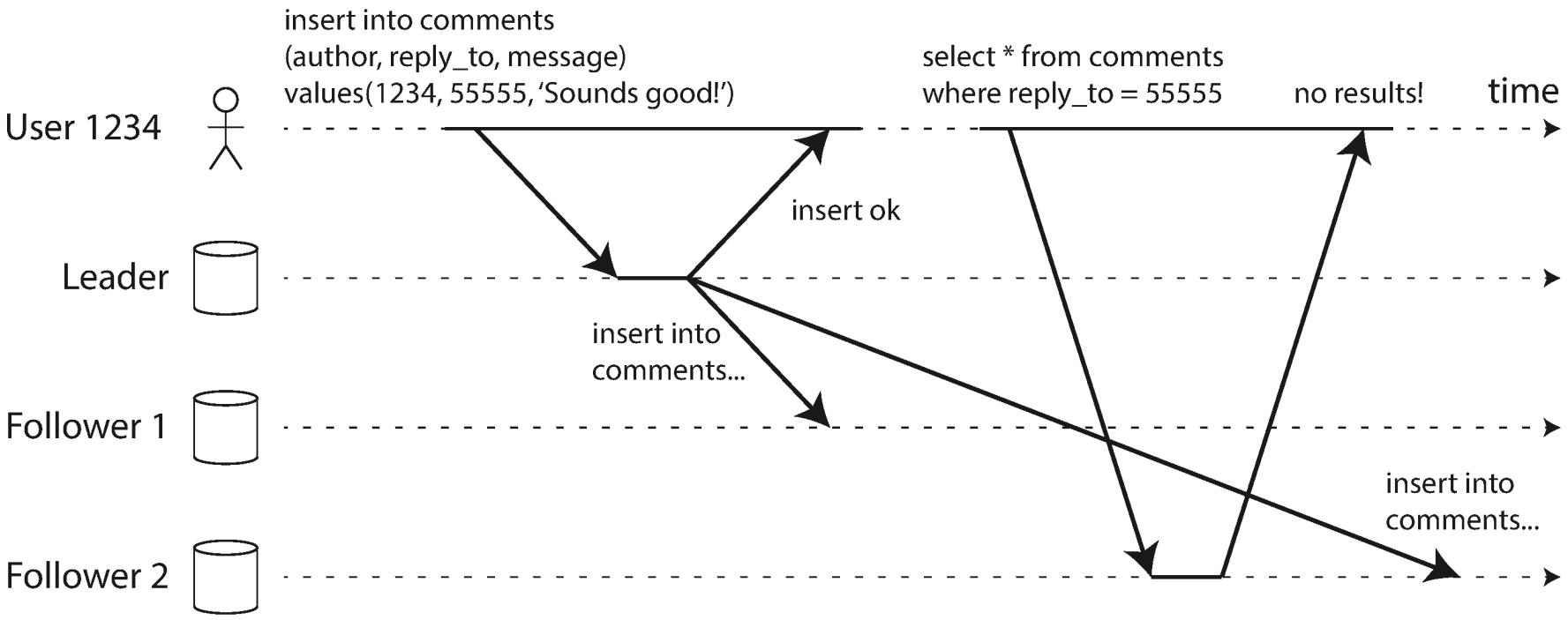
Replication Lag refers to the time it takes for writes on the leader to be reflected on all of the followers.

Synchronous replication: Replication lag causes writes to be slower and the system to be more brittle as we add more and more followers.

Asynchronous replication: We maintain availability but at the cost of delayed or *eventual* consistency. This delay is called the *inconsistency window*.

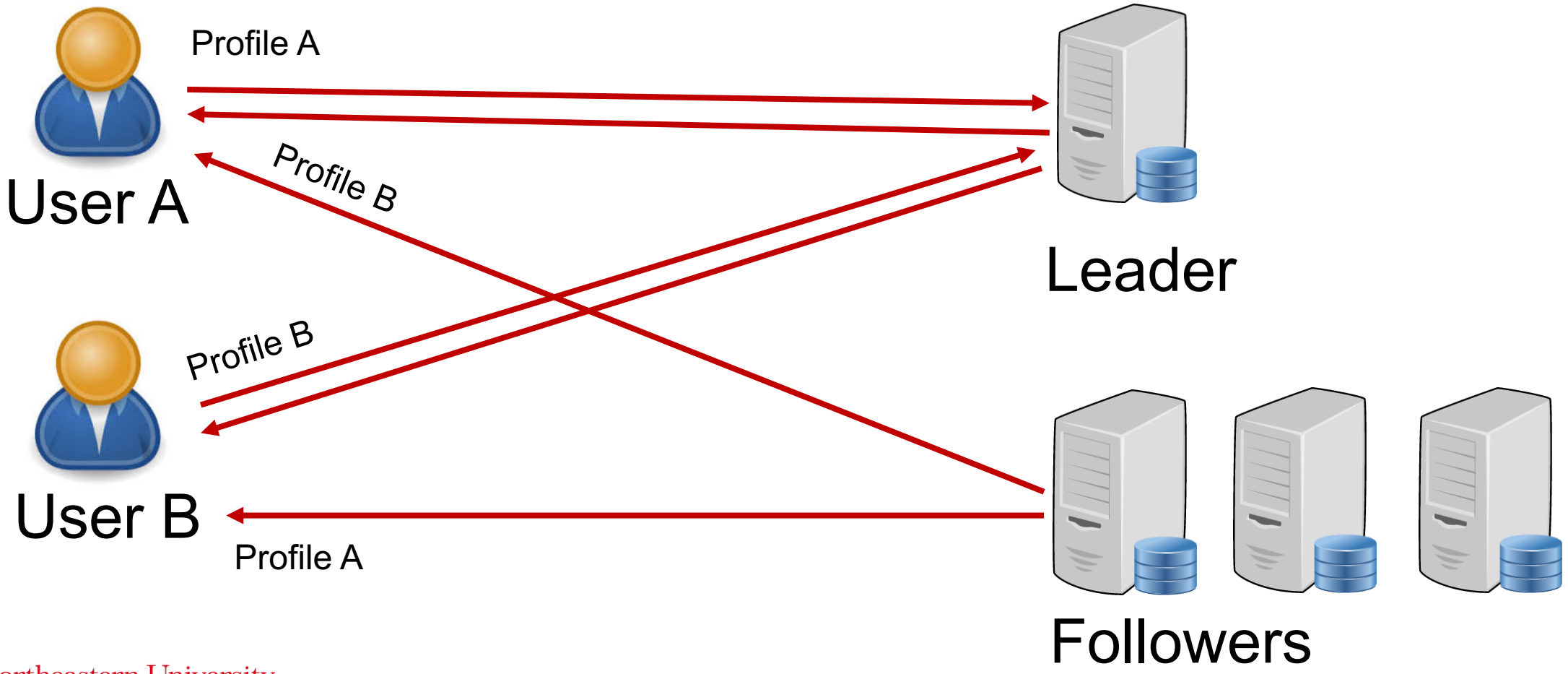
Read-after-write Consistency

Users should see writes they themselves made (even if other users are delayed in seeing those same writes.)



Implementing read-after-write consistency

Method 1. Modifiable data is read from the Leader



Implementing read-after-write consistency

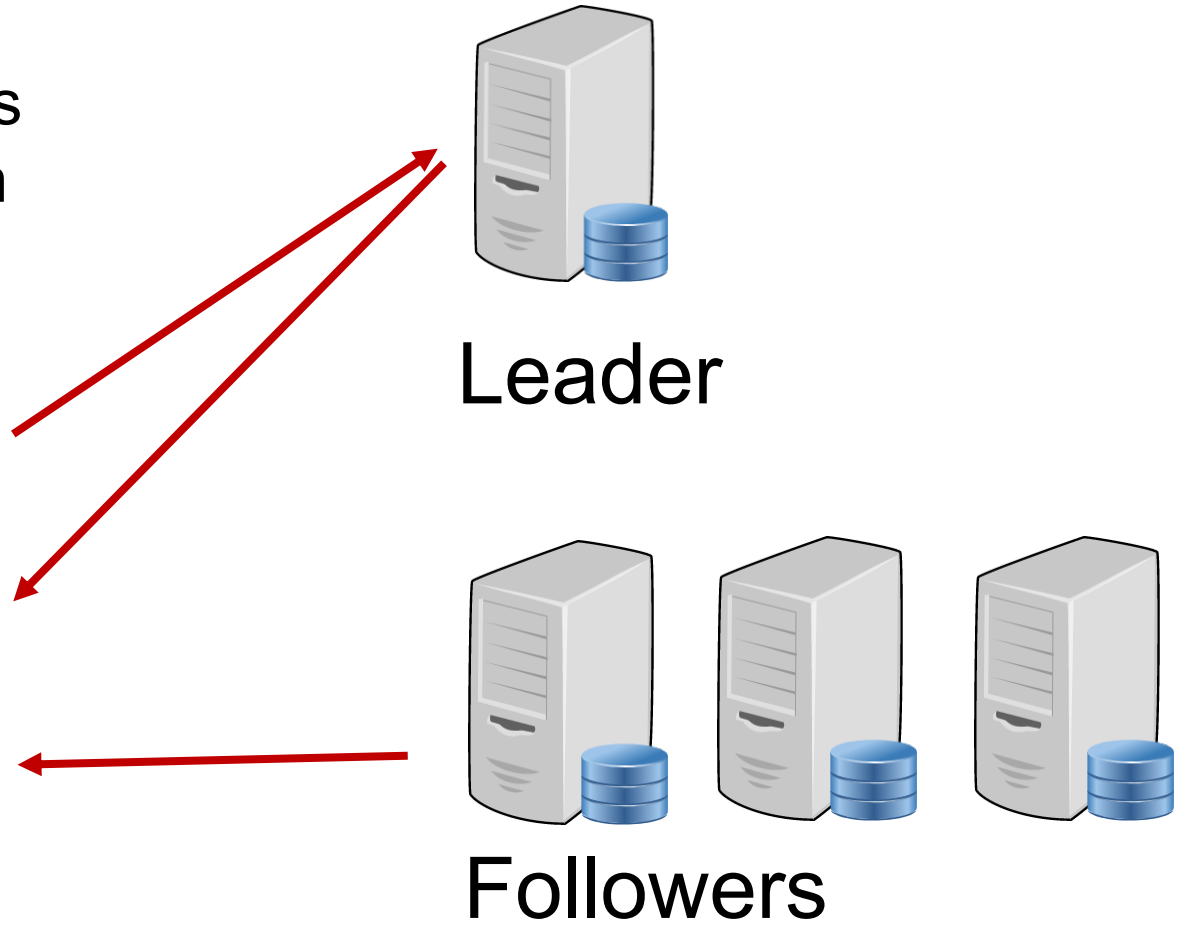
Method 2. Dynamically switch to reading from Leader for "recently updated" data.

For example, have a policy that all requests within one minute of last update come from Leader

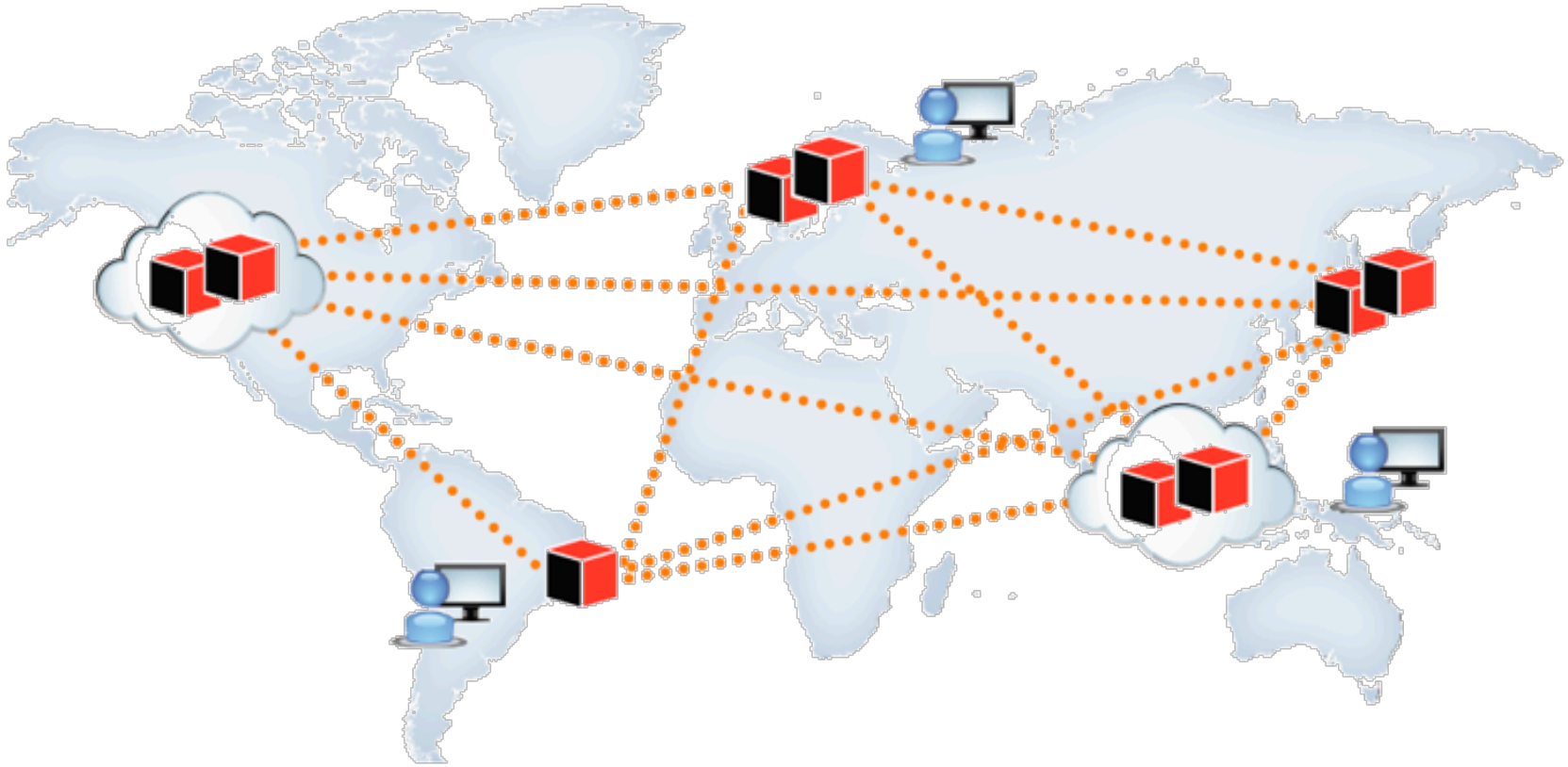
Update X @ 14h 22m 55.783s

Read X @ 14h 23m 12.199s

Read X @ 14h 24m 29.635s



What if the leader is geographically remote?

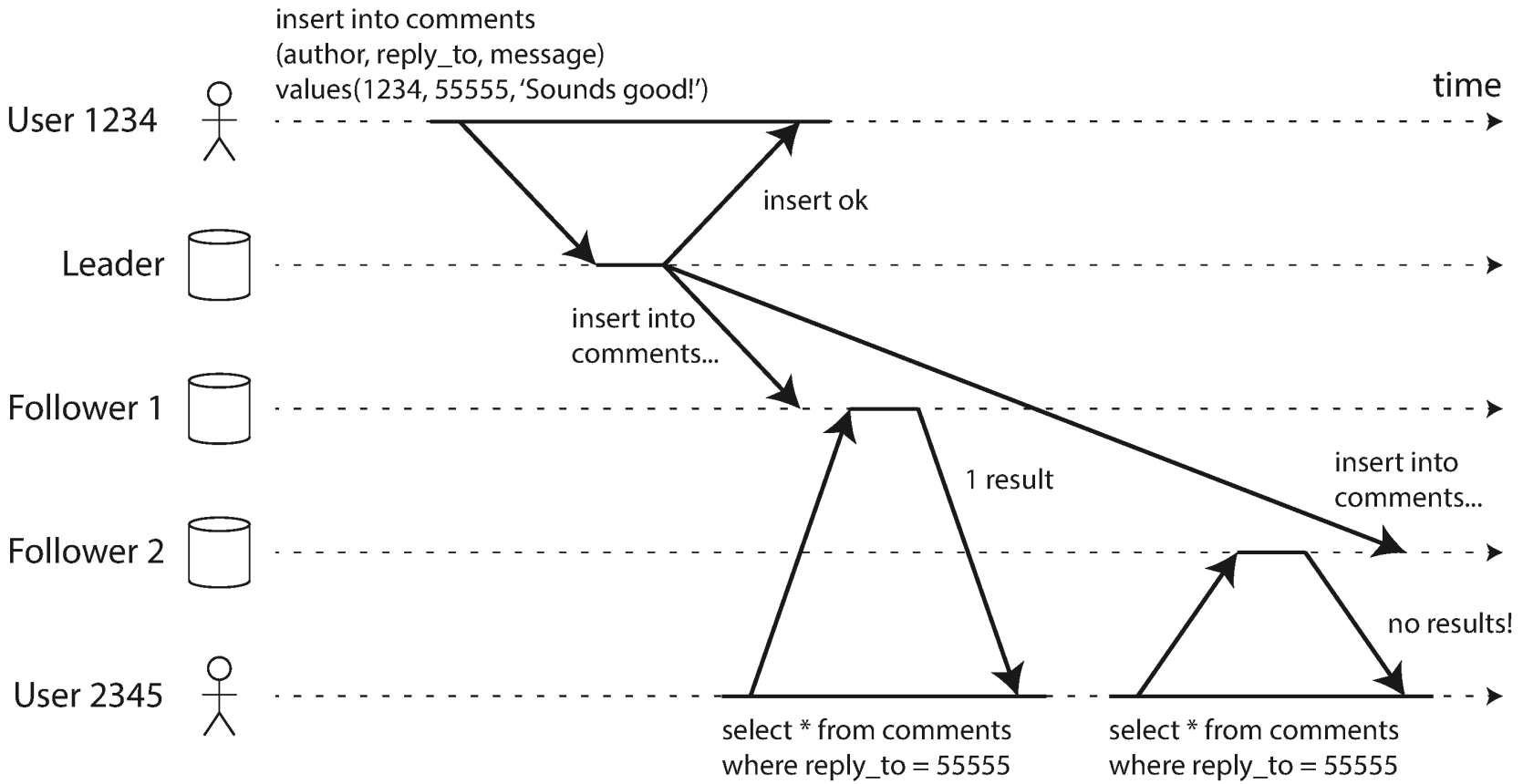


We created followers so they would be proximal to users.
But now we have to route requests to distant leaders.

Monotonic read consistency

Monotonic read anomalies occur when a user reads values out of order from multiple followers.

Monotonic read consistency ensures that when a user makes multiple reads, they will not read older data after previously reading newer data.

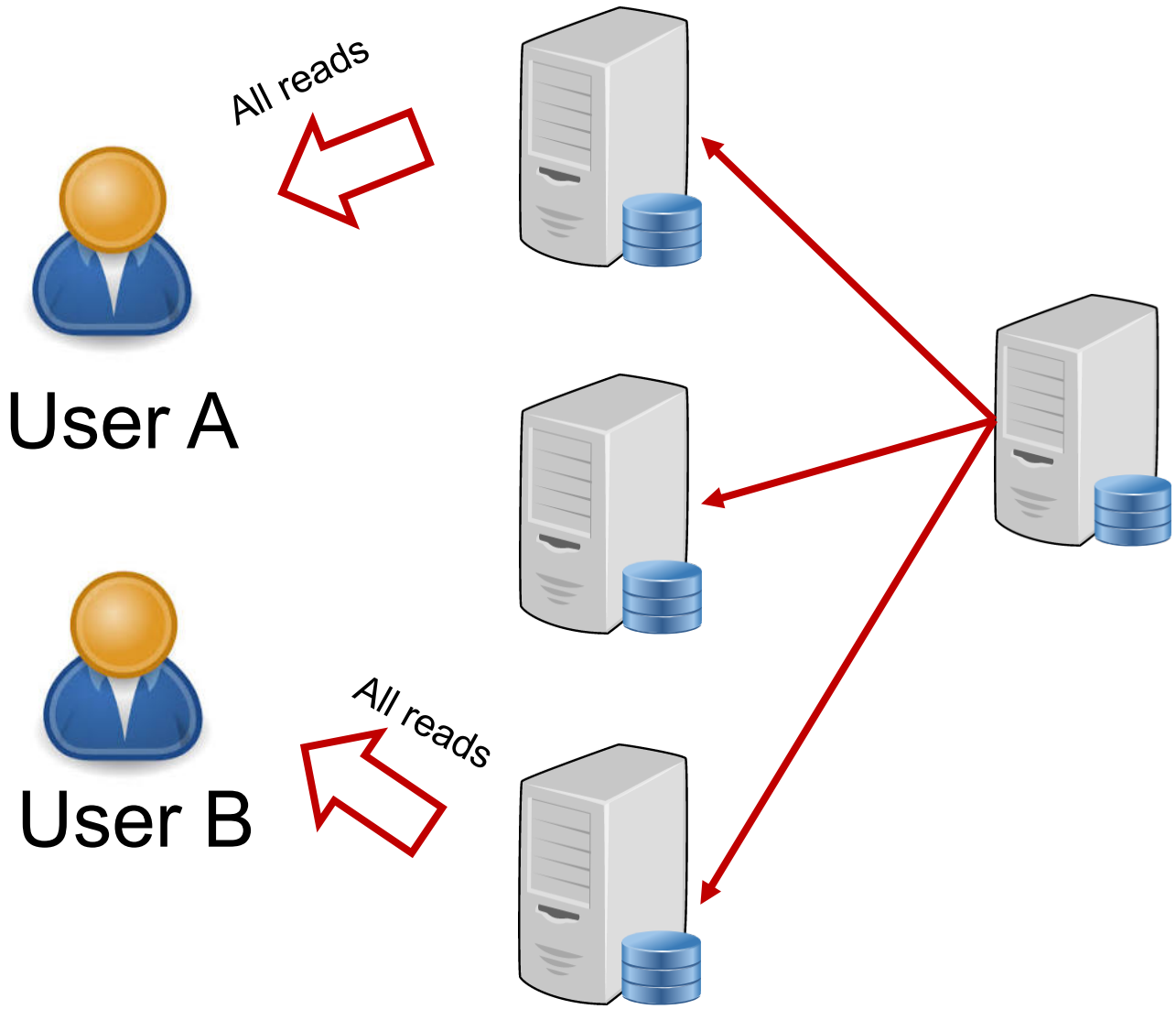


Eventual < Monotonic < Strong

Monotonic read consistency

Monotonic read anomalies occur when a user reads values out of order from multiple followers.

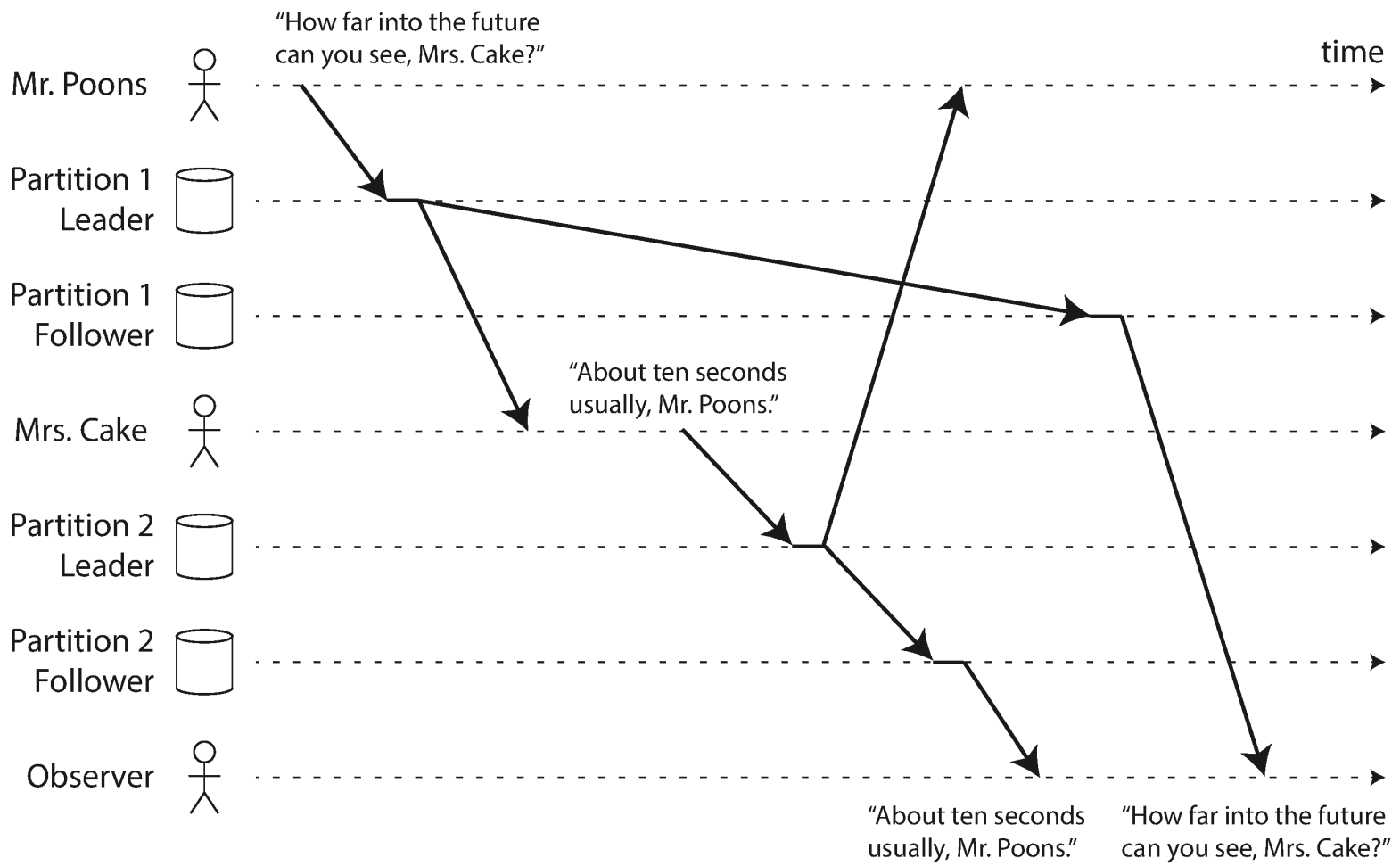
Monotonic read consistency ensures that when a user makes multiple reads, they will not read older data after previously reading newer data.



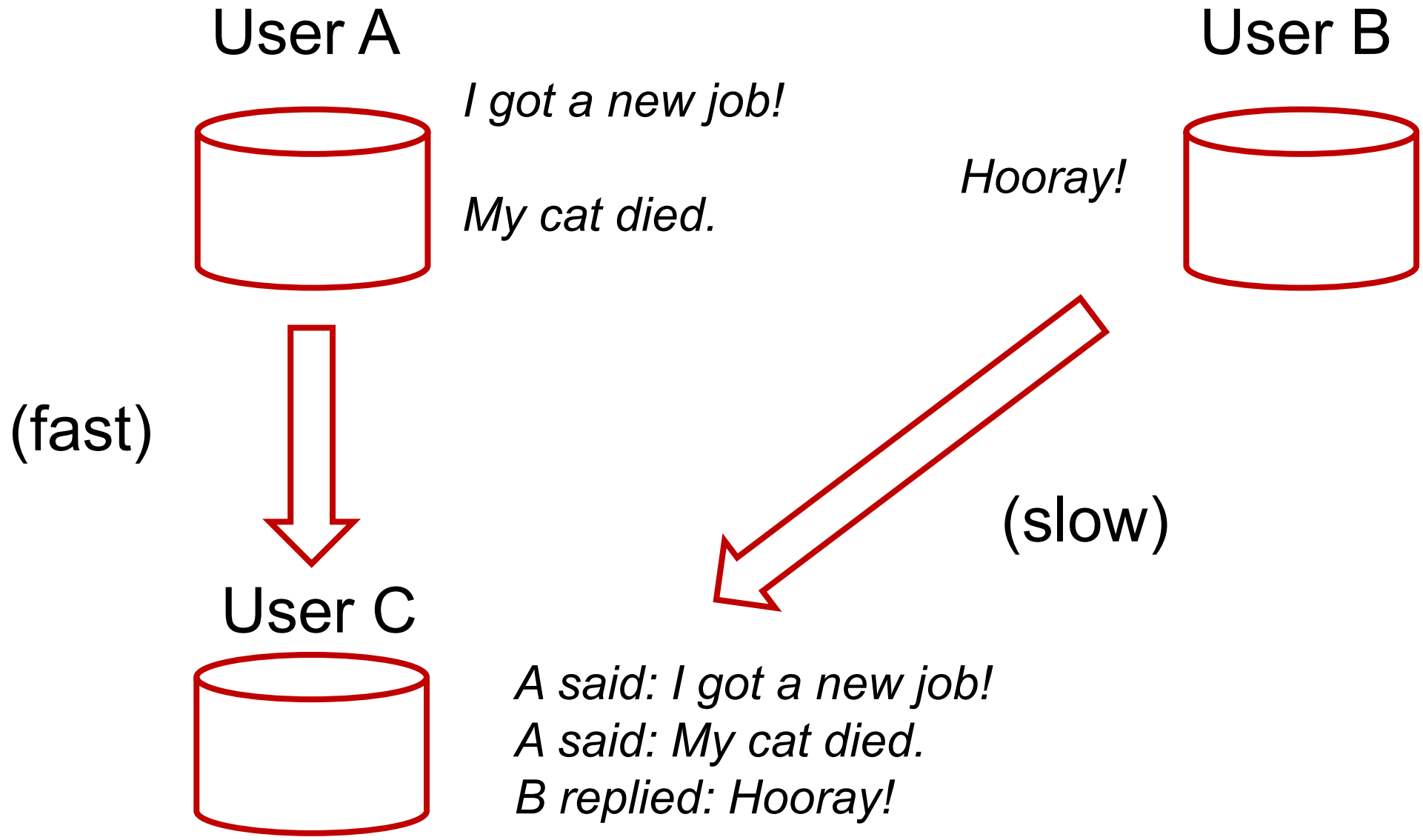
Consistent prefix reads (Causal consistency)

Reading data out of order can occur if different partitions replicate data at different rates. There is no global write consistency.

Consistent prefix read guarantee ensures that if a sequence of writes happens in a certain order, anyone reading those writes will see them appear in the same order.



A social feed

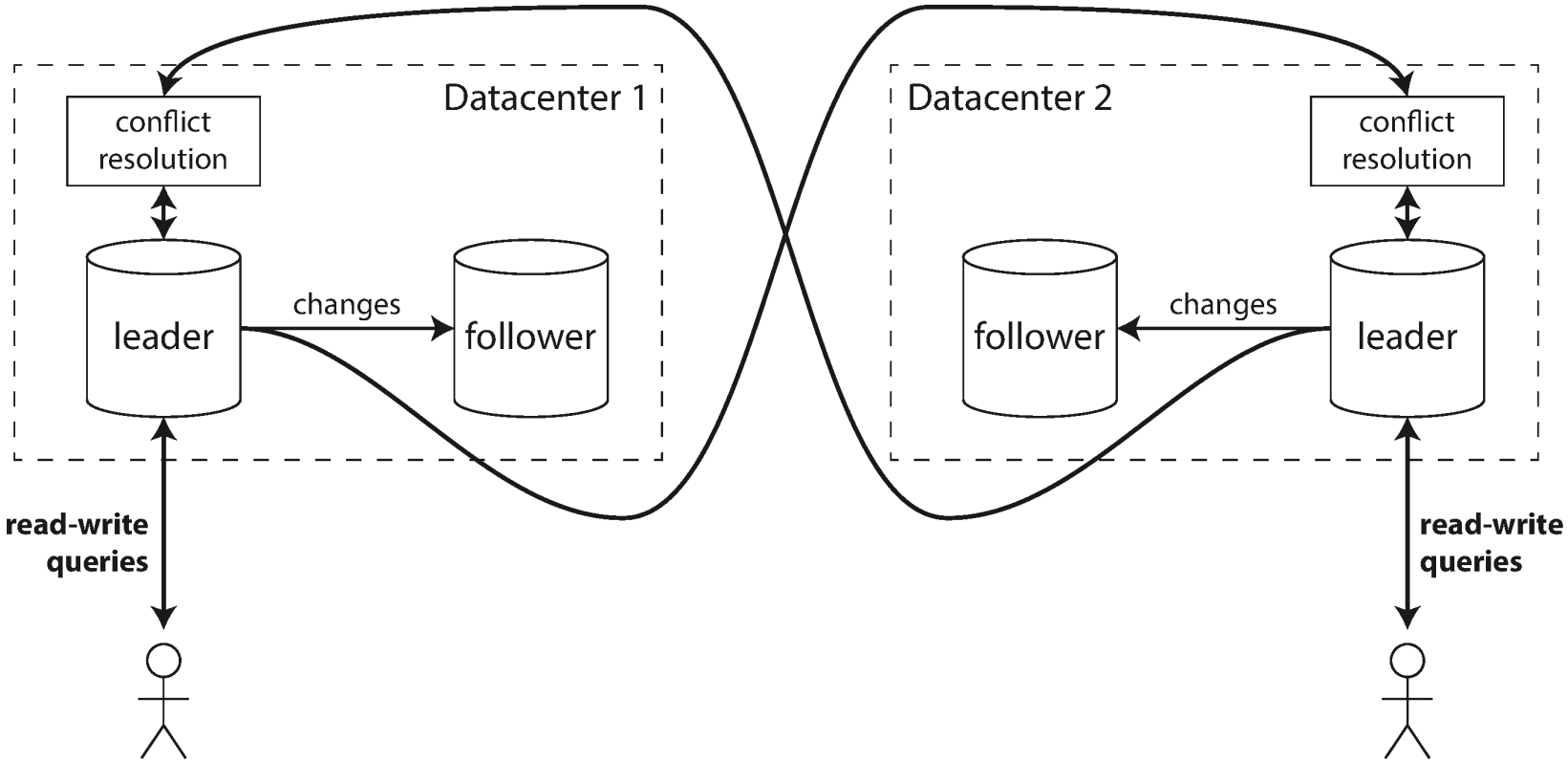


3 Levels of consistency

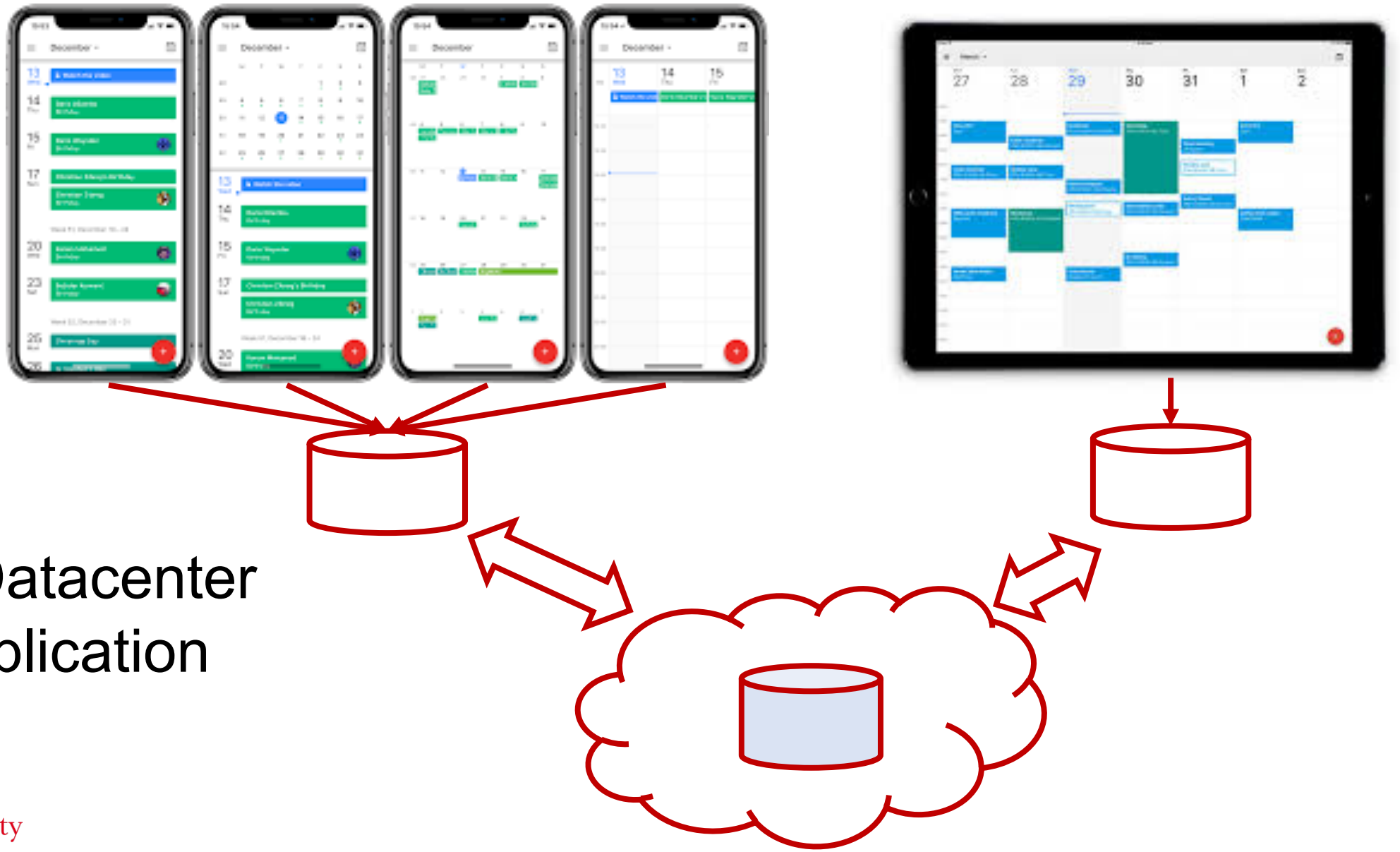
Consistency Promise	Description	Solution
Read After Write	Users should always see data that they submitted themselves.	Users reads data they can modify from the leader. Or read from leader for some interval after an update.
Monotonic Reads	After users have seen the data at one point in time, they shouldn't later see the data from some earlier point in time.	Require that a user always read from the same follower
Consistent Prefix Reads	Users should see the data in a state that makes causal sense: for example, seeing a tweet and a retweet in the correct order.	Causally connected data is written to the same partition or have applications manage causal dependencies.

Multi-Leader Replication (active/active, master-master)

- Each data center has its own leader, supporting **reduced latency**.
- Leaders communicate with each other to stay in sync.
- **Greater tolerance** of intra-datacenter networking problems
- Downside: **Conflict-resolution** protocols are needed – e.g., how to handle something simple like auto increments?



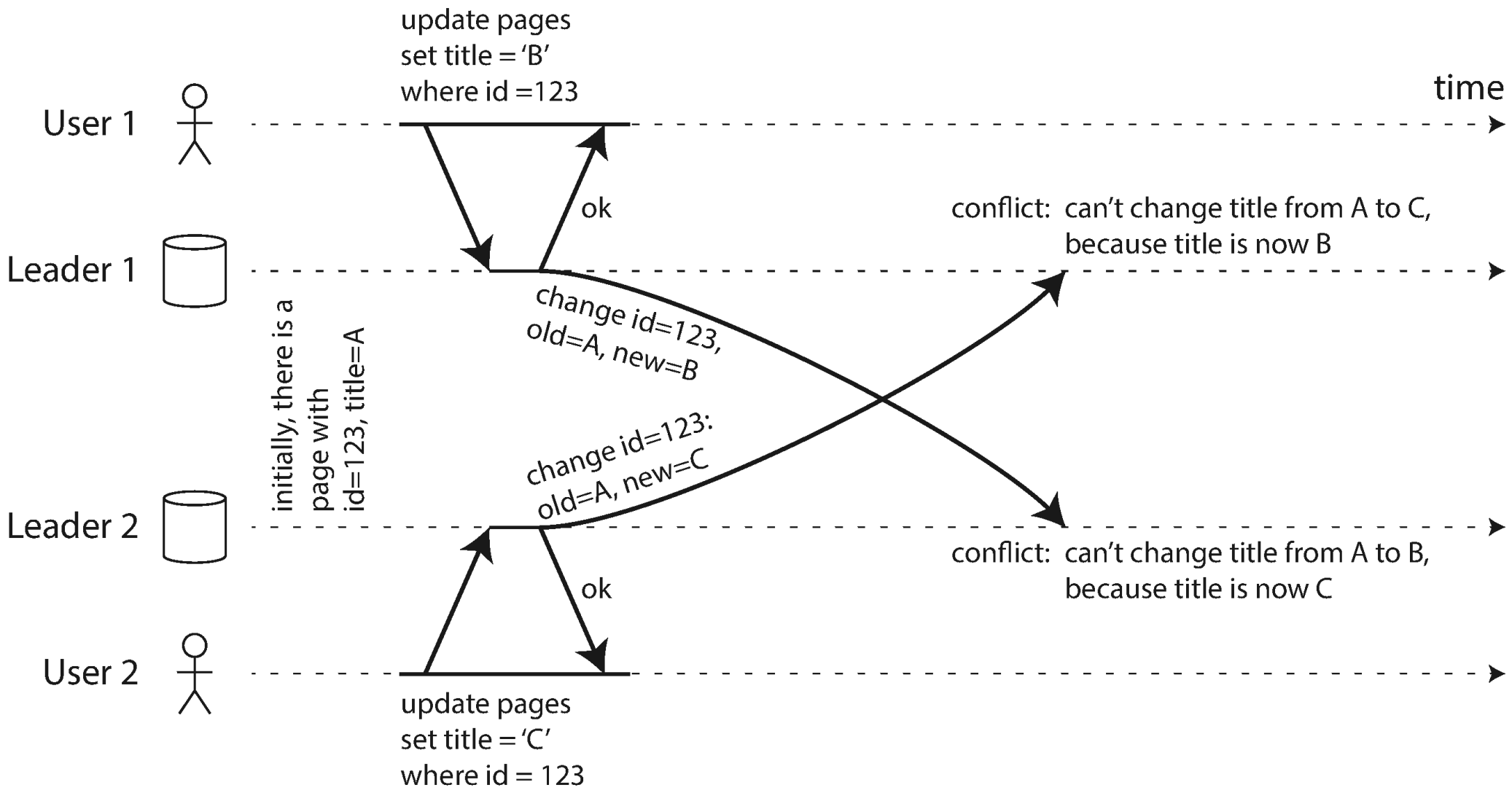
SaaS applications with offline support



Device ~ Datacenter

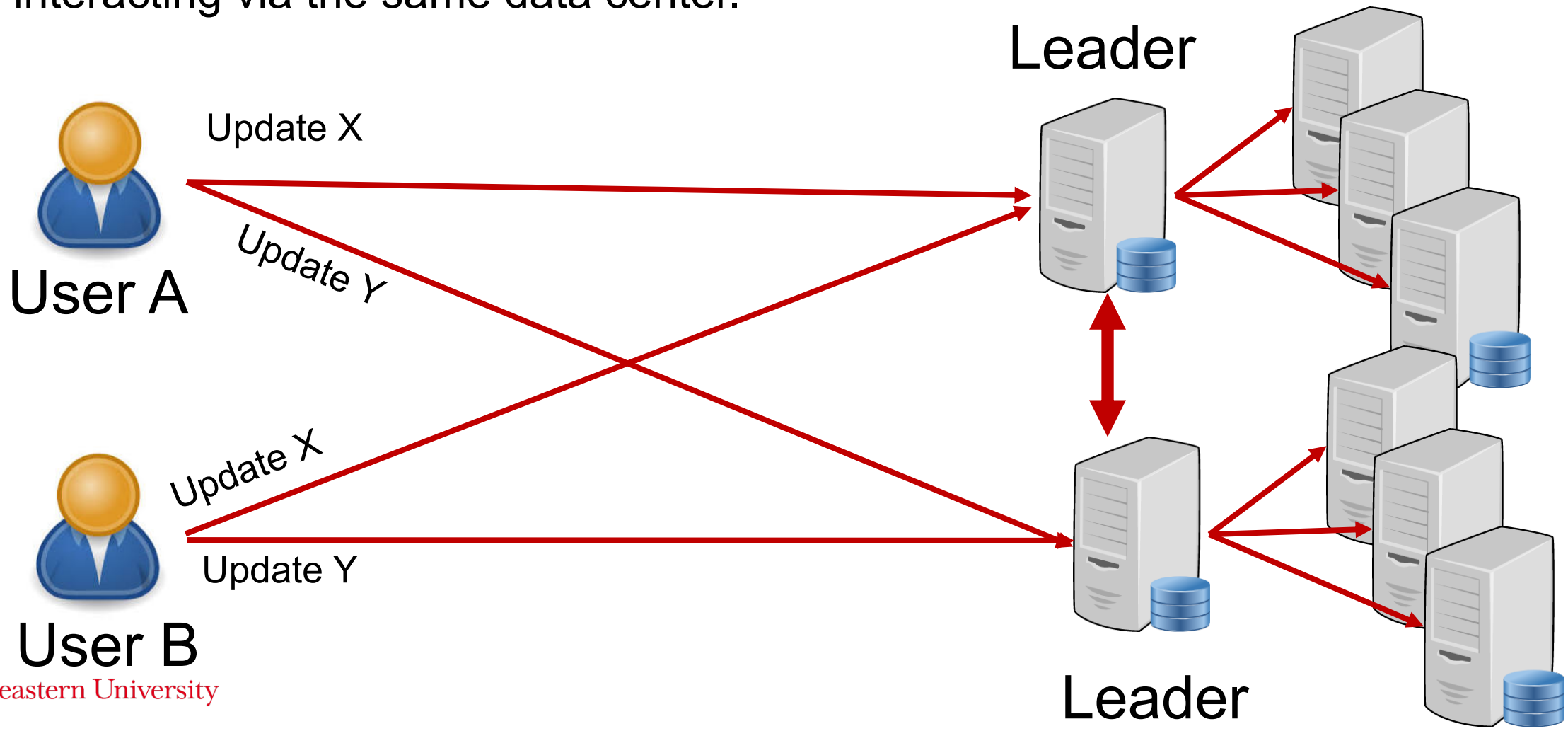
Sync ~ Replication

An example conflict



Handling Write Conflicts via Avoidance

Avoidance: Require all writes for a particular record to go through the same leader. E.g., have a particular user always interacting via the same data center.



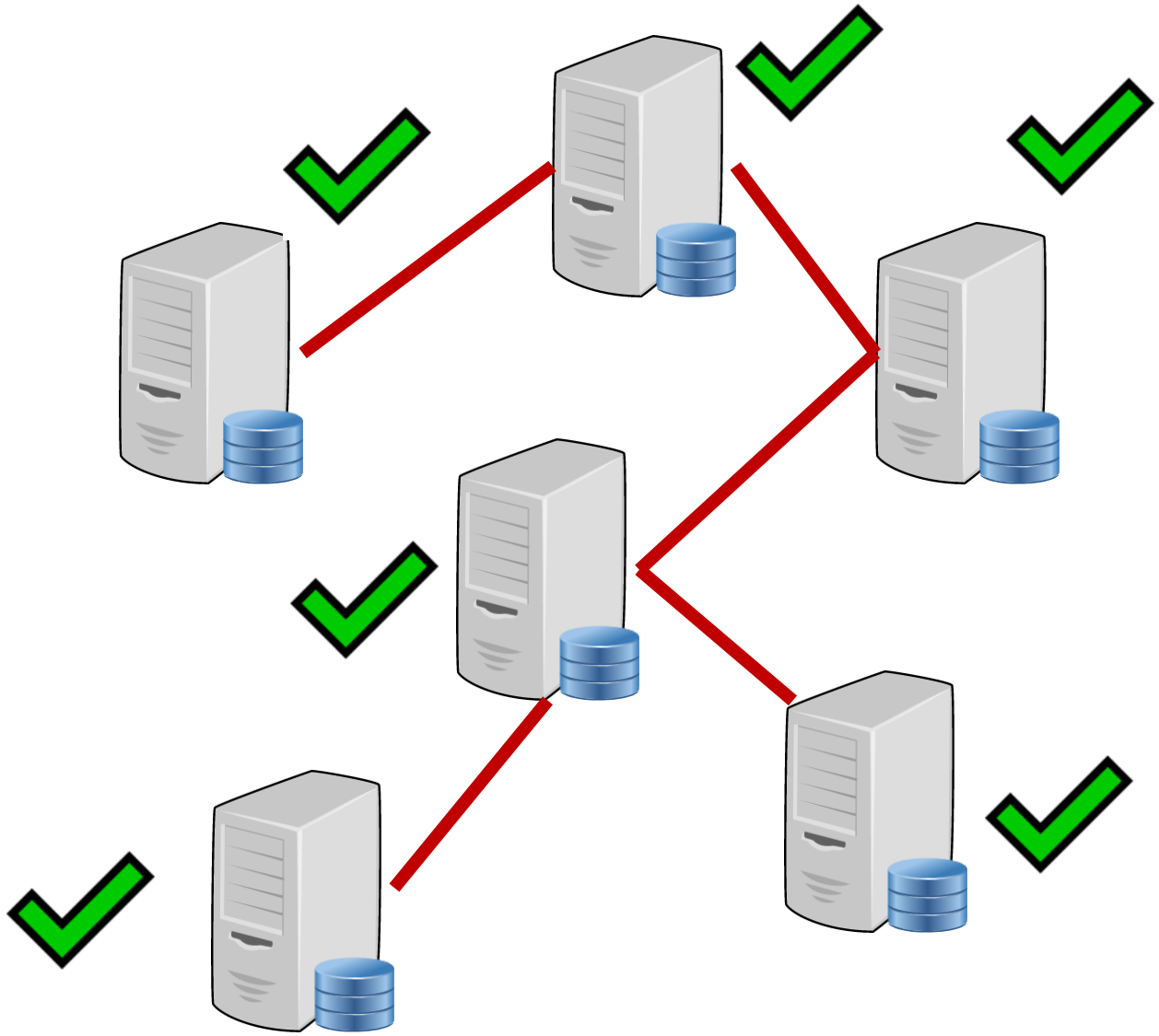
Handling Write Conflicts via Convergence

Convergence: Allow the conflict to occur but converge on a common outcome for all replicas.

- Last write wins (LWW): Attach a timestamp to each write.
- Replica ranking: High-ranked replicas win over low-ranked replicas
- Merging: Combine writes somehow
- Pass-the-buck: Log the conflict and leave it to application code to handle.
- Custom database specific algorithms. For example, Google uses *operational transformation* algorithms to resolve write conflicts in Google Docs
- Conflict-free replicated data types (CRDTs).

Trivial example: Raising a flag

- Raising a Boolean flag to signal an event (But never allow the flag to be unraised)
- It doesn't matter which leader signals the event, we can reliably converge on a consistent value
- Simple protocol:
  a) neighboring compare
  b) if one has a TRUE value, they both become true

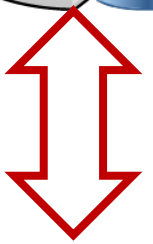


Incrementing a simple counter on multiple nodes

"like"

10 likes

likes = likes + 1



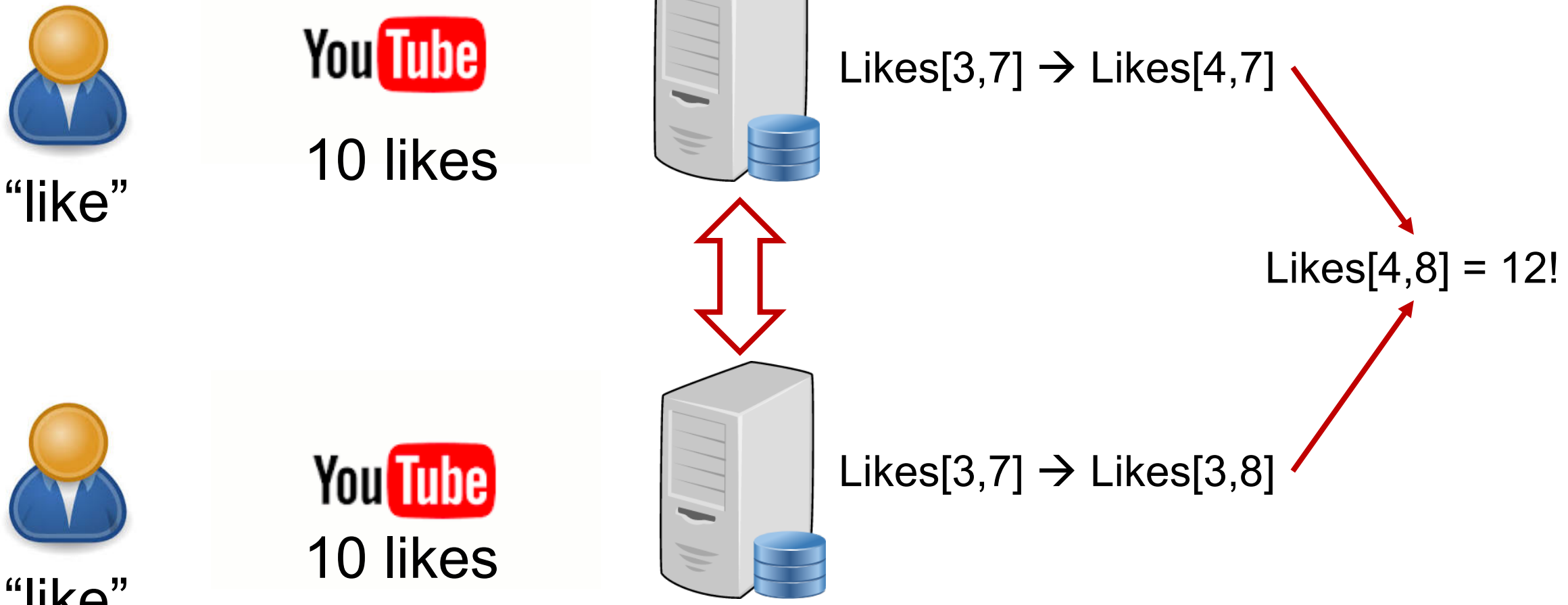
"likes is now 11!"

"like"

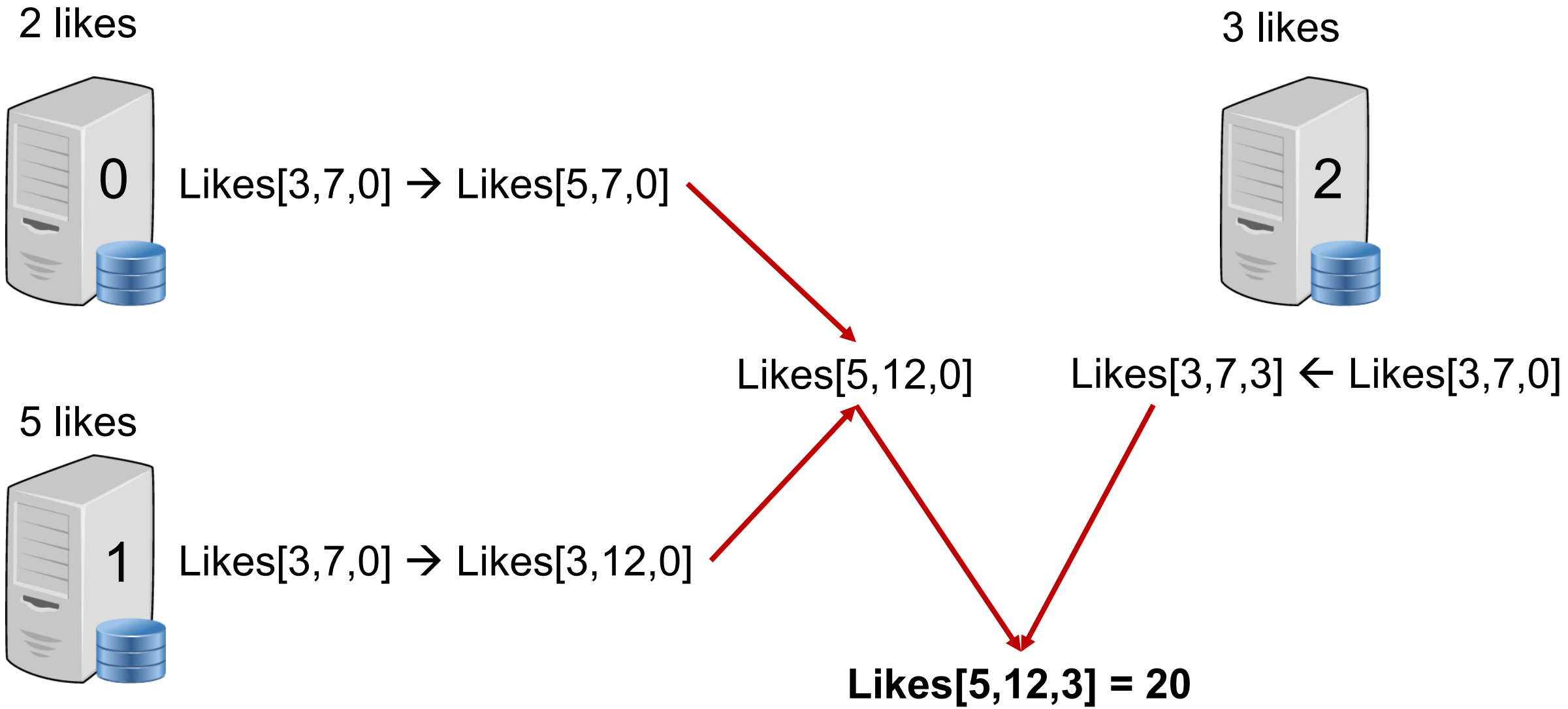
10 likes

likes = likes + 1

Incrementing a simple counter on multiple nodes



It works with as many leaders as you like!



Conflict-free replicated datatypes

Conflict-free replicated datatypes (CRDTs) are a family of data structures for sets, maps, ordered lists, counters, etc. that can be concurrently edited by multiple users, and which automatically resolve conflicts in sensible ways.

G-Counter (Grow-only Counter) [edit]

```
payload integer[n] P
    initial [0,0,...,0]
update increment()
    let g = myId()
    P[g] := P[g] + 1
query value() : integer v
    let v = ∑_i P[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0, n - 1] : X.P[i] ≤ Y.P[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0, n - 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

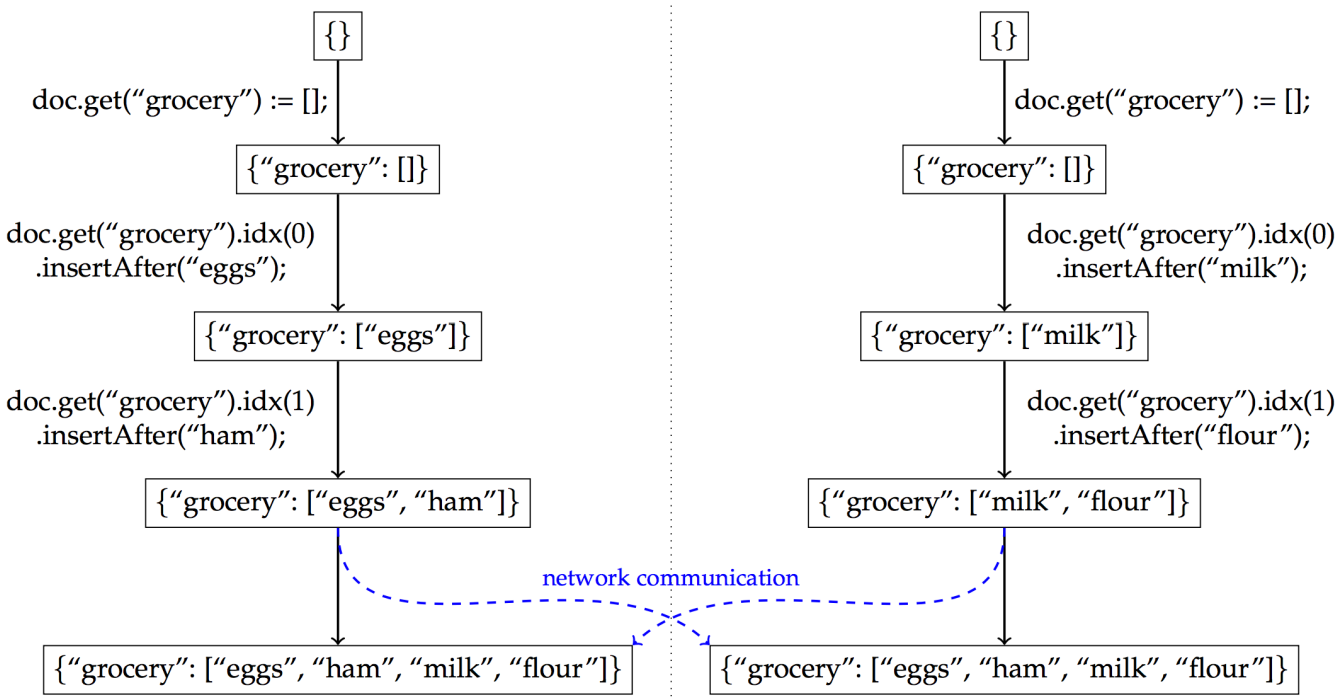
Source: Wikipedia

A Conflict-Free Replicated JSON Datatype

Martin Kleppmann and Alastair R. Beresford

Abstract—Many applications model their data in a general-purpose storage format such as JSON. This data structure is modified by the application as a result of user input. Such modifications are well understood if performed sequentially on a single copy of the data, but if the data is replicated and modified concurrently on multiple devices, it is unclear what the semantics should be. In this paper we present an algorithm and formal semantics for a JSON data structure that automatically resolves concurrent modifications such that no updates are lost, and such that all replicas converge towards the same state (a conflict-free replicated datatype or CRDT). It supports arbitrarily nested list and map types, which can be modified by insertion, deletion and assignment. The algorithm performs all merging client-side and does not depend on ordering guarantees from the network, making it suitable for deployment on mobile devices with poor network connectivity, in peer-to-peer networks, and in messaging systems with end-to-end encryption.

Index Terms—CRDTs, Collaborative Editing, P2P, JSON



An active area of research and development!

Automerge

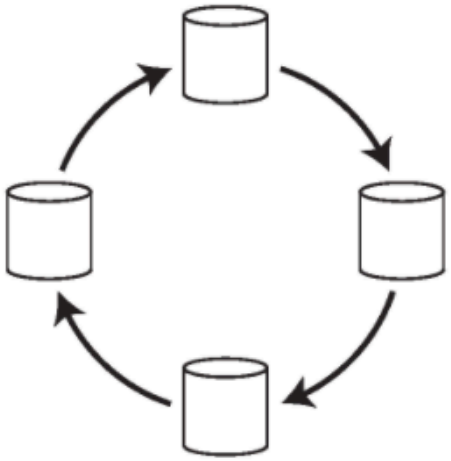
Join the Automerge Slack community

Automerge is a library of data structures for building collaborative applications in JavaScript.

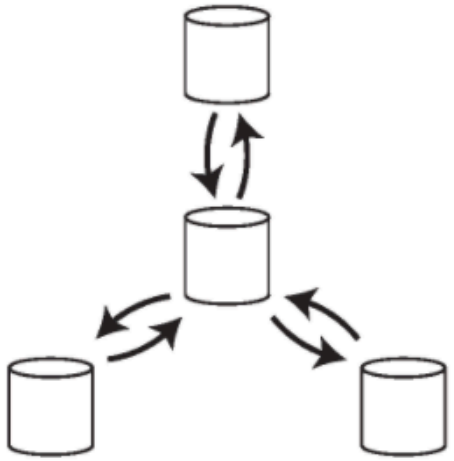
Automatic merging. Automerge is a so-called Conflict-Free Replicated Data Type (CRDT), which allows concurrent changes on different devices to be merged automatically without requiring any central server. It is based on academic research on JSON CRDTs, but the details of the algorithm in Automerge are different from the JSON CRDT paper, and we are planning to publish more detail about it in the future.

(developed by Martin Kleppmann – your textbook author!)

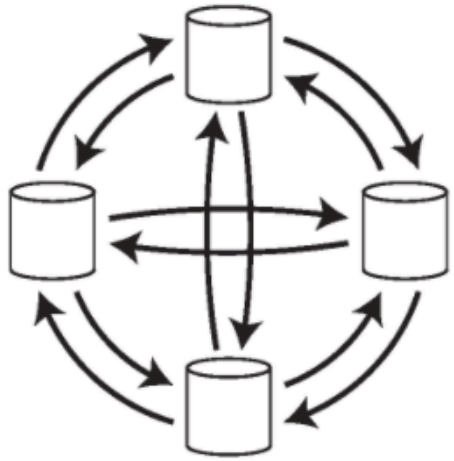
Multi-Leader Topologies



(a) Circular topology



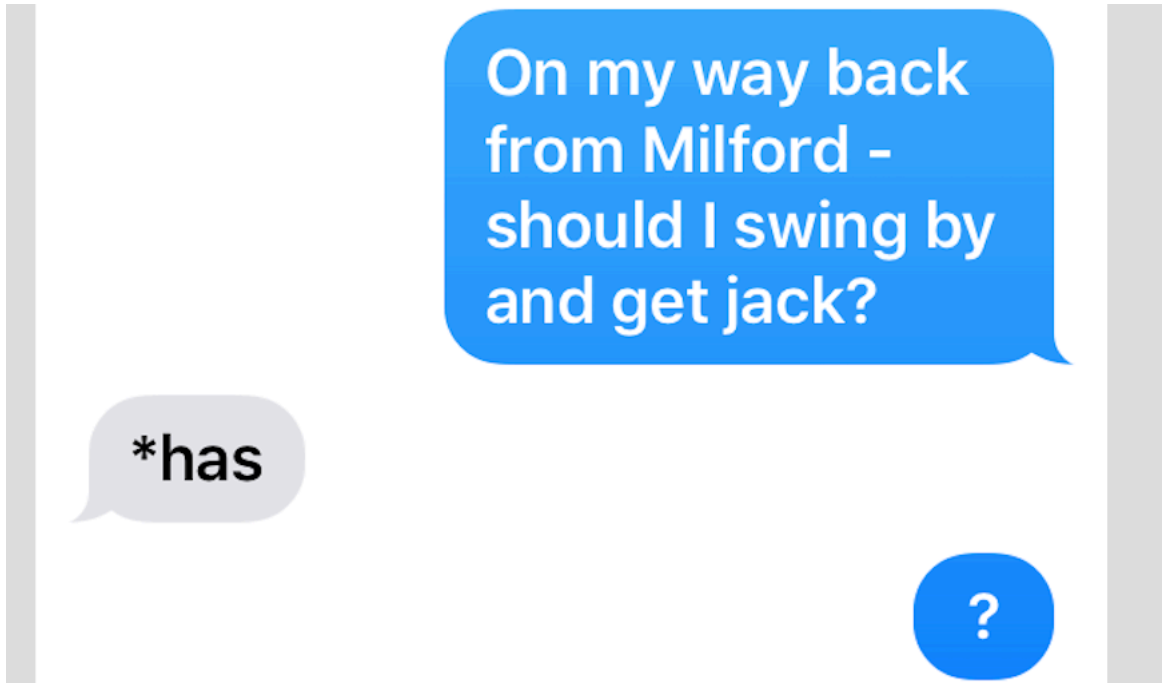
(b) Star topology



(c) All-to-all topology

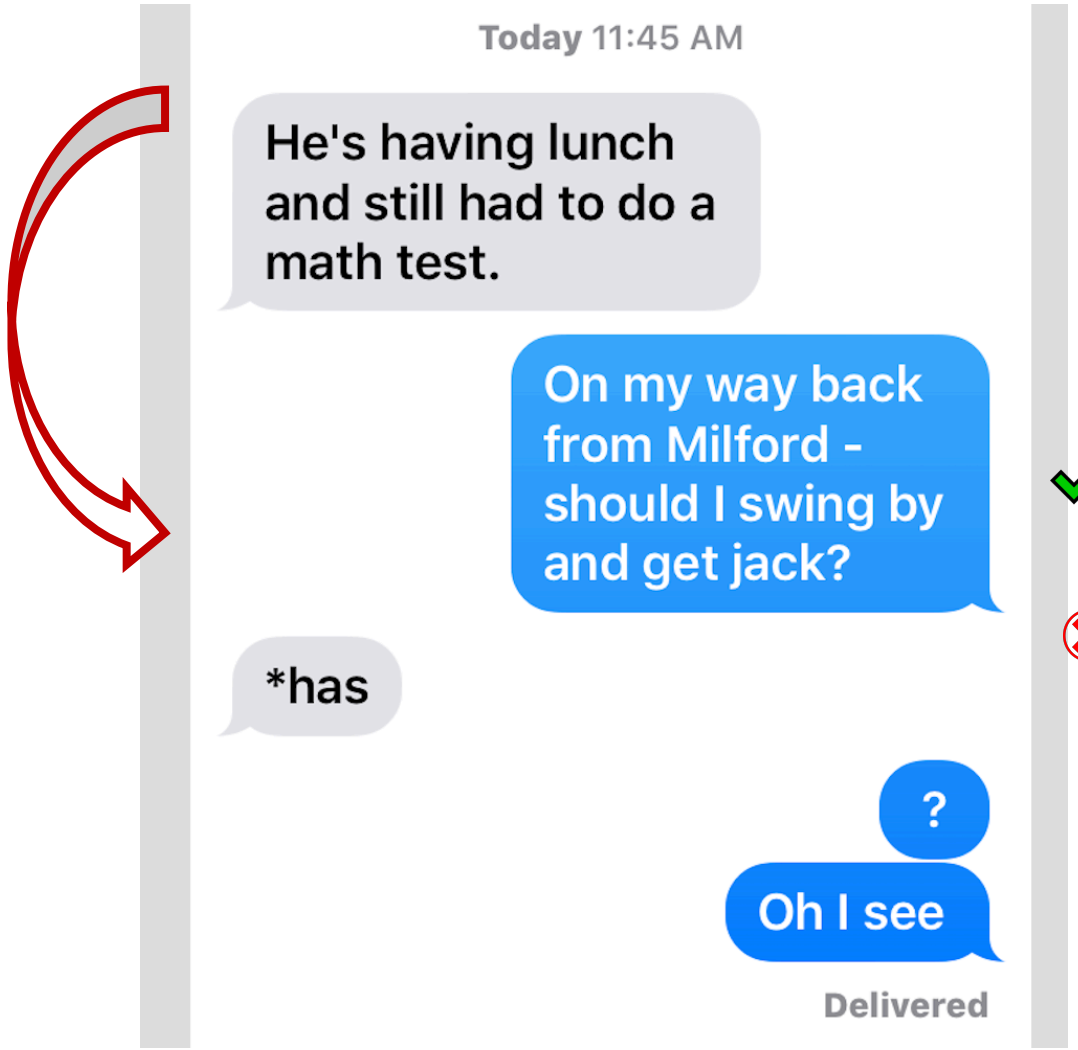
- All-to-all is more robust (no single point of failure)
- But having multiple replication paths also creates the potential for anomalies
  (i.e., if one message overtakes another)

A real-world example: text messaging

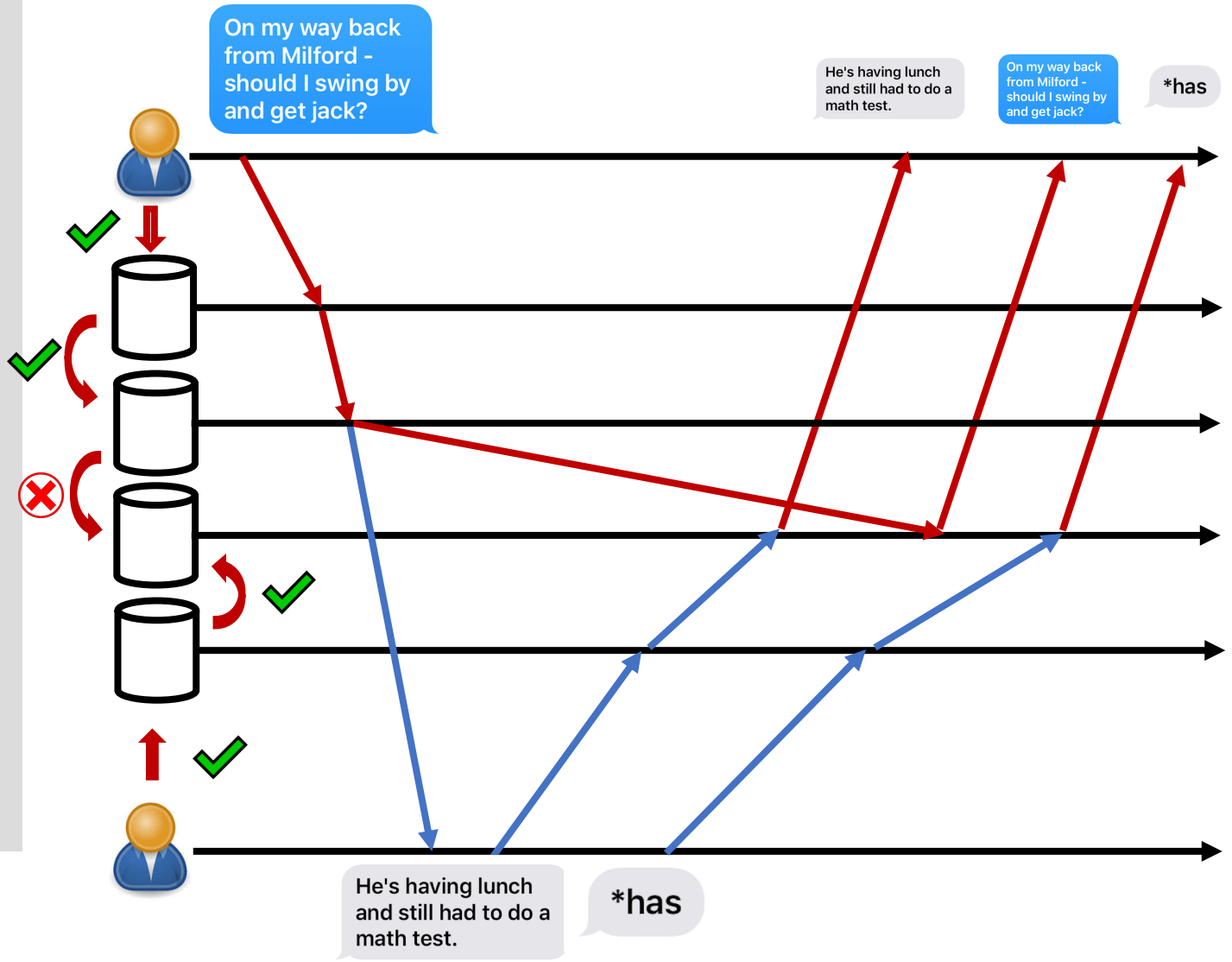


Prefix-read inconsistency causes me some confusion!

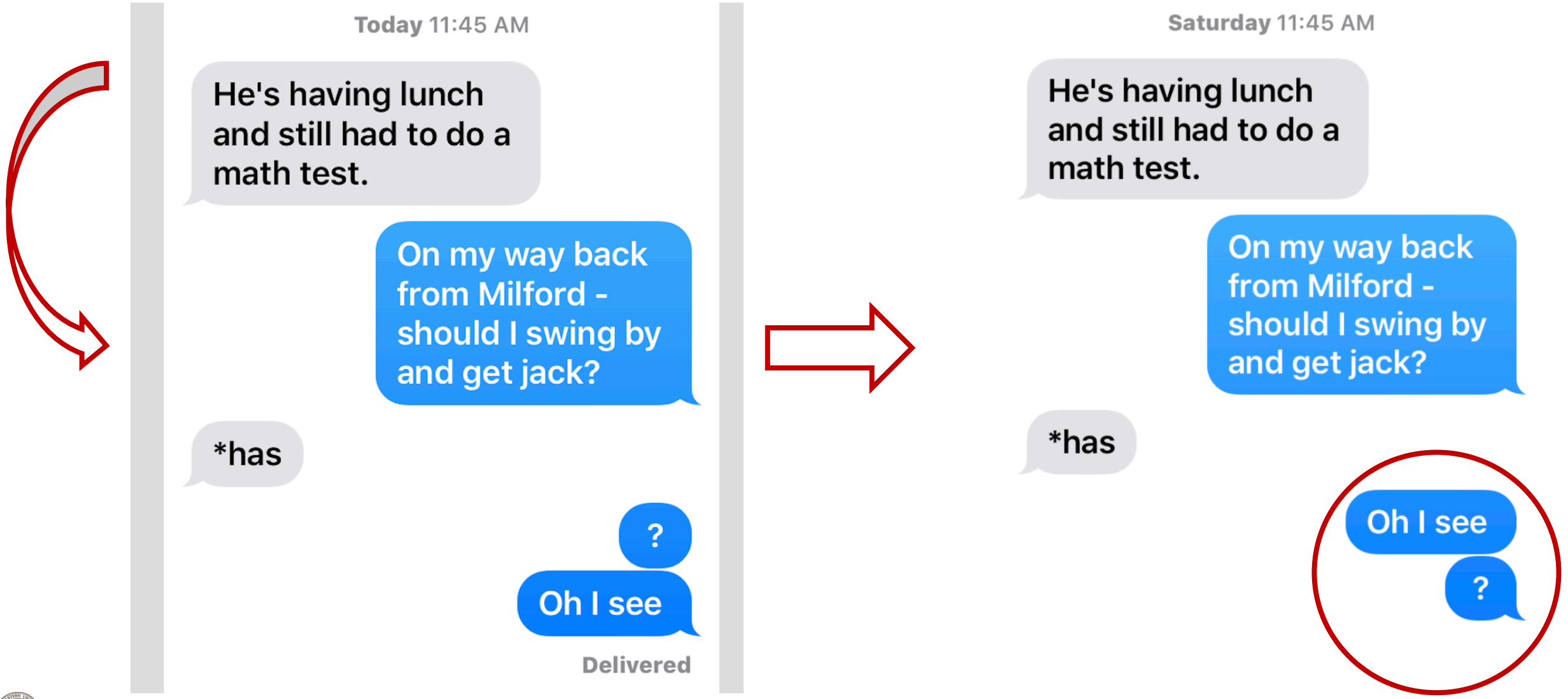
A real-world example: replication lag



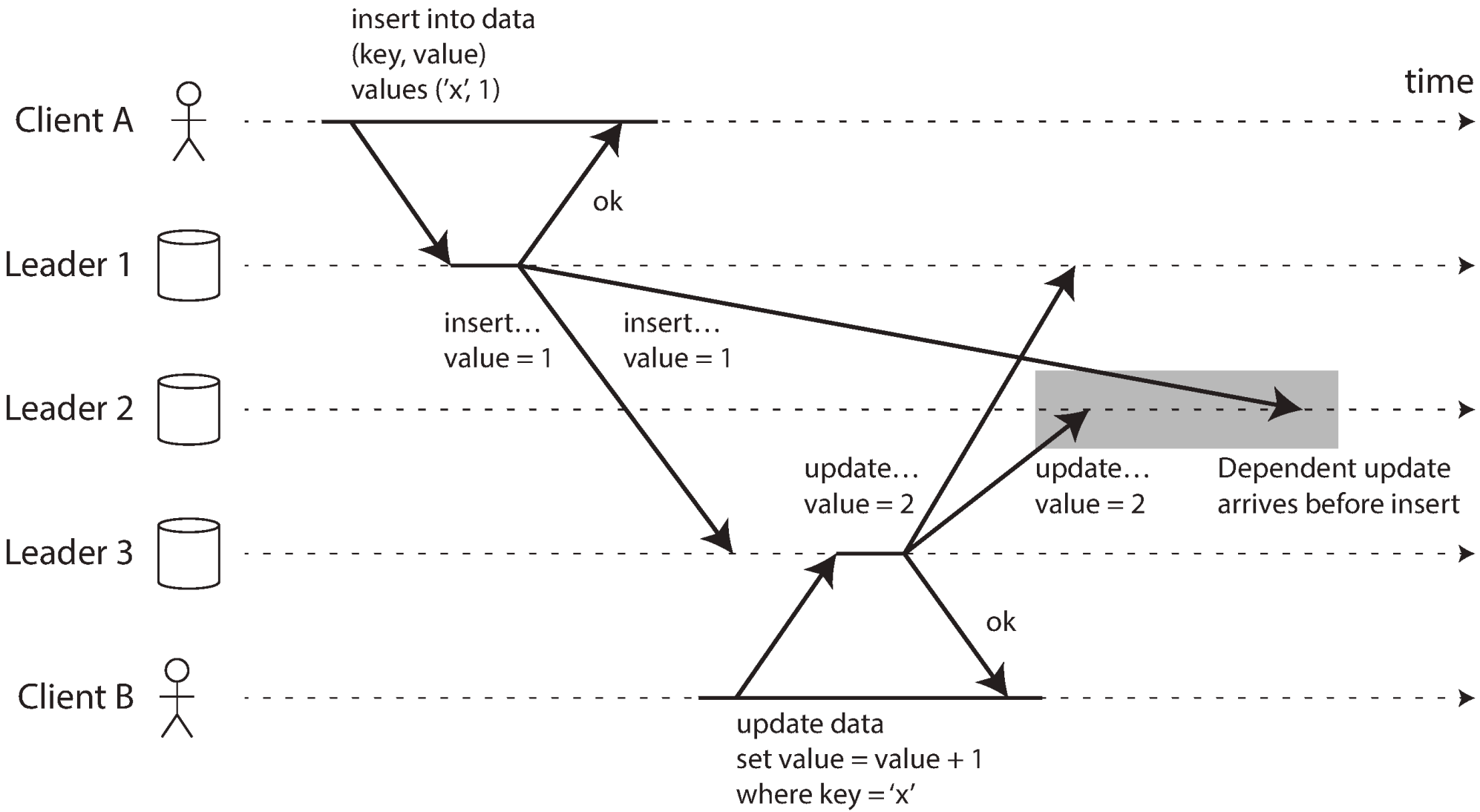
One possibility



A real-world example: text messaging



Multi-leader replication anomalies with All-to-All

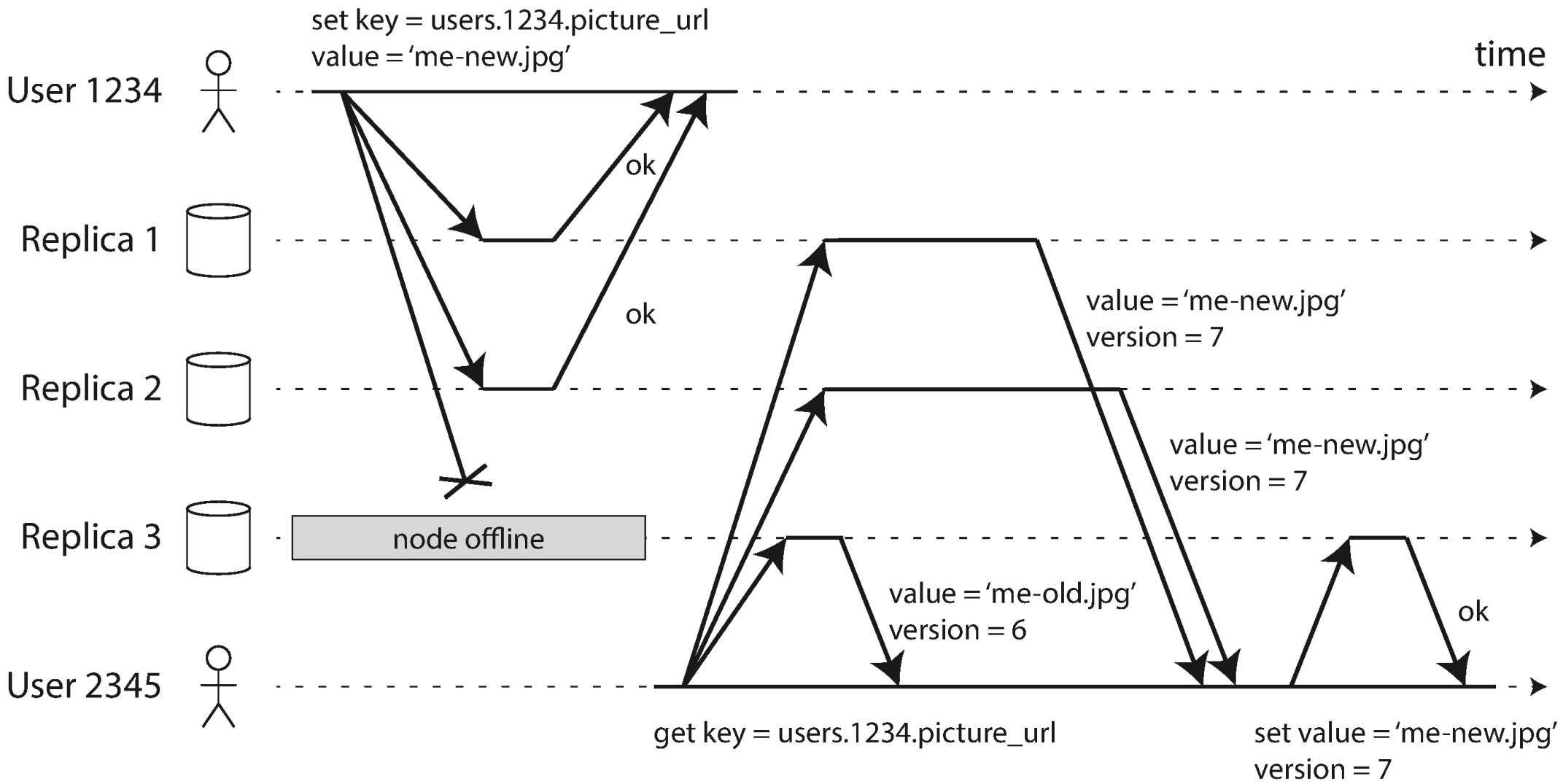


Leaderless Replication

- There is no designated leader
- Writes may be made to one or more replicas *in any order*
- Sometimes called “Dynamo-style” after Amazon’s internal Dynamo key-value store which implements the notion of tunable consistency using the NWR model.

  Note: *DynamoDB (publically available in AWS) is different. It uses single-leader replication.*

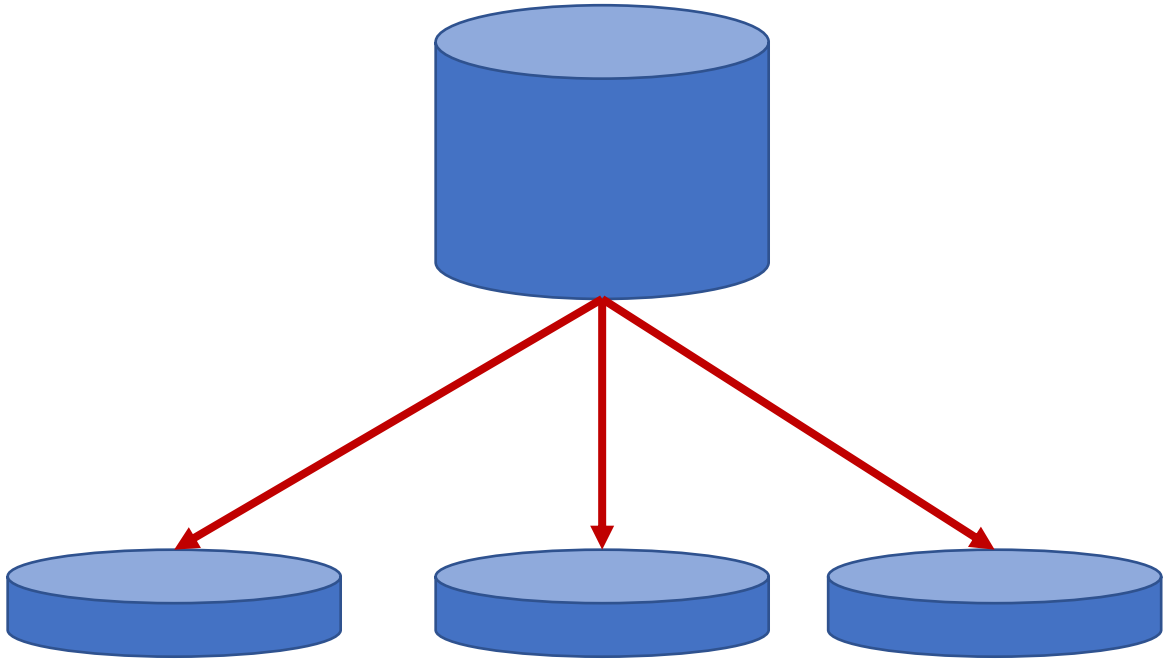
Leaderless replication with a failed node



Partitioning

Partitioning

- Split data across multiple database servers or nodes
- Each partition is its own database
- Cross-partition operations are more difficult and slower but possible
- Why partition? *Scalability*! (Multiple nodes can share the load, although we must take steps to ensure that the partitions are comparably-sized.)

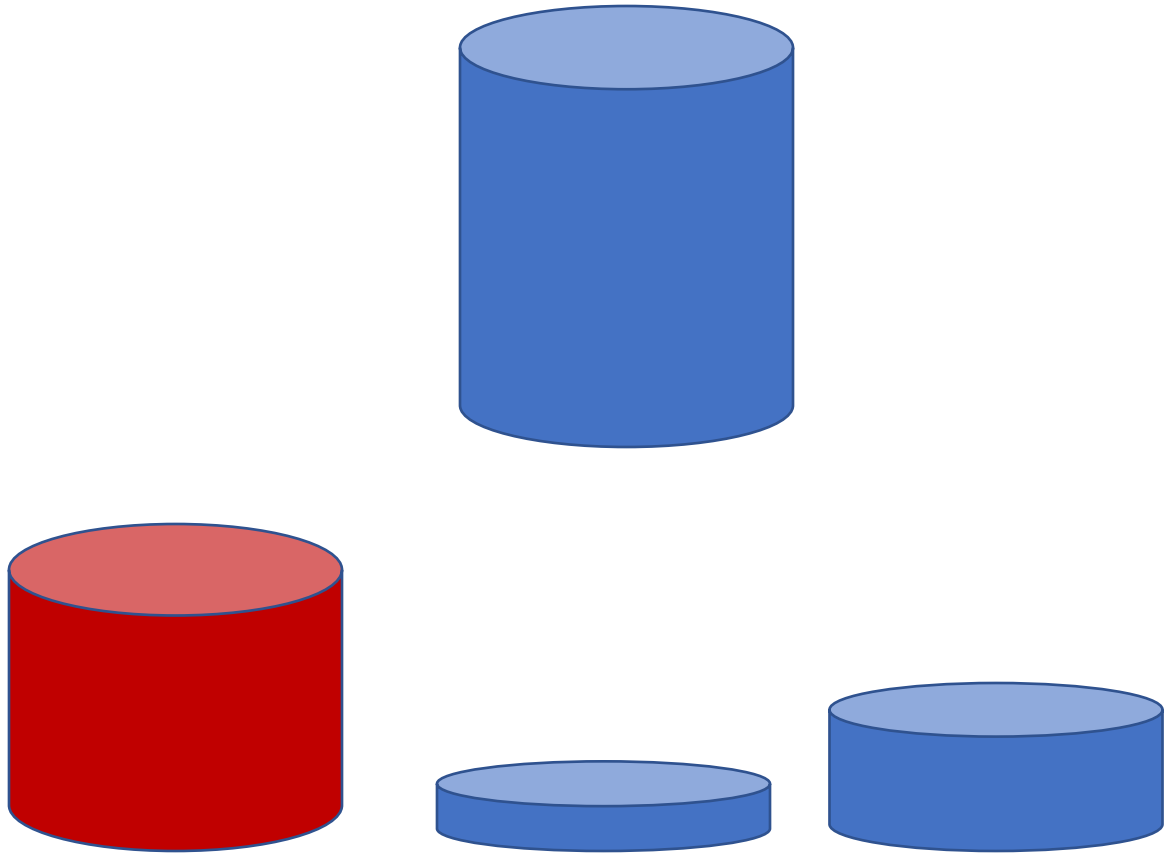


Question #1: How should we partition?

What scheme do we use to partition our data?

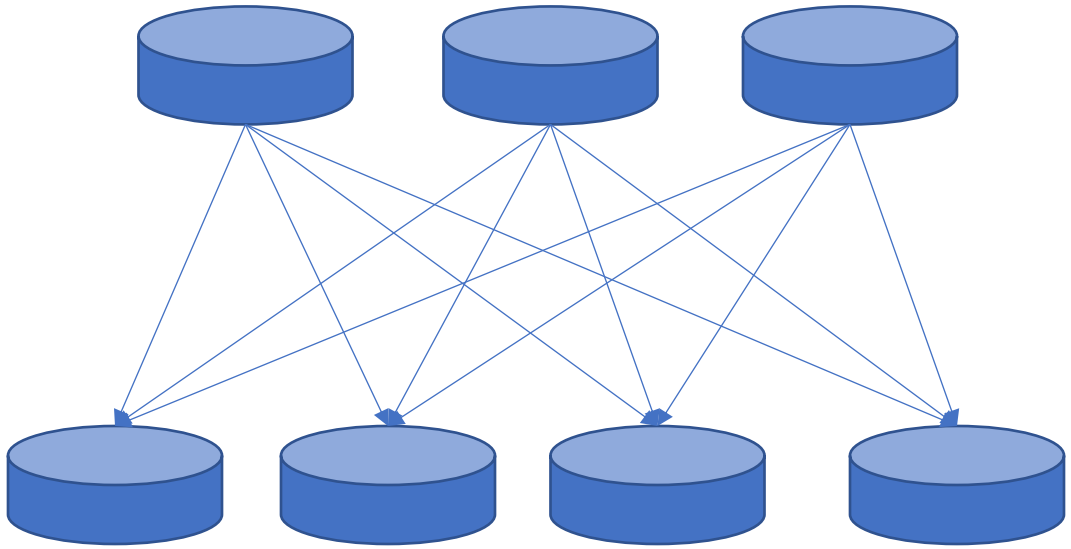
Data that is queried together belongs together to avoid expensive cross-partition joins.

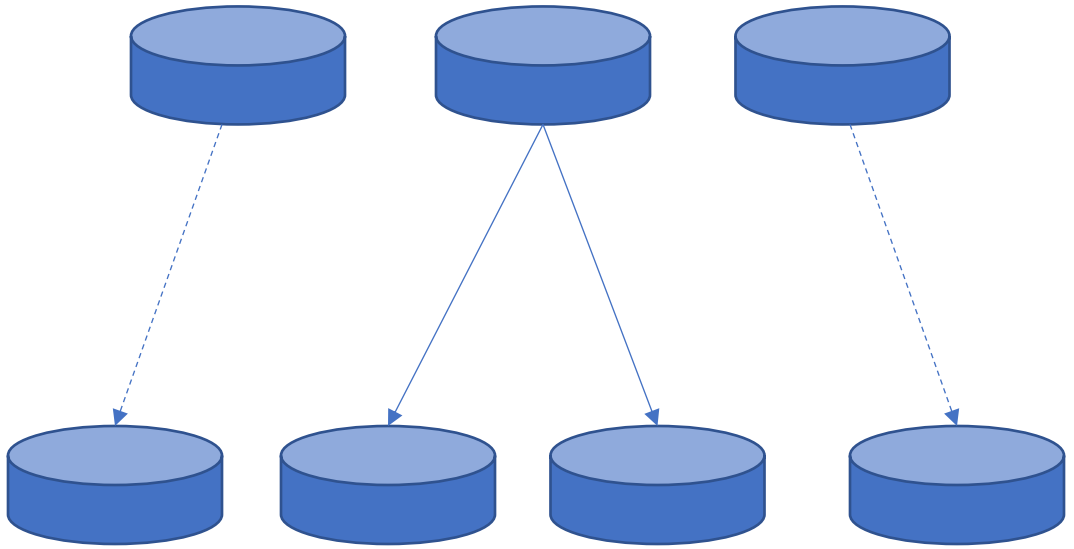
How do we avoid data *skew, hot spots, and bottlenecks?*



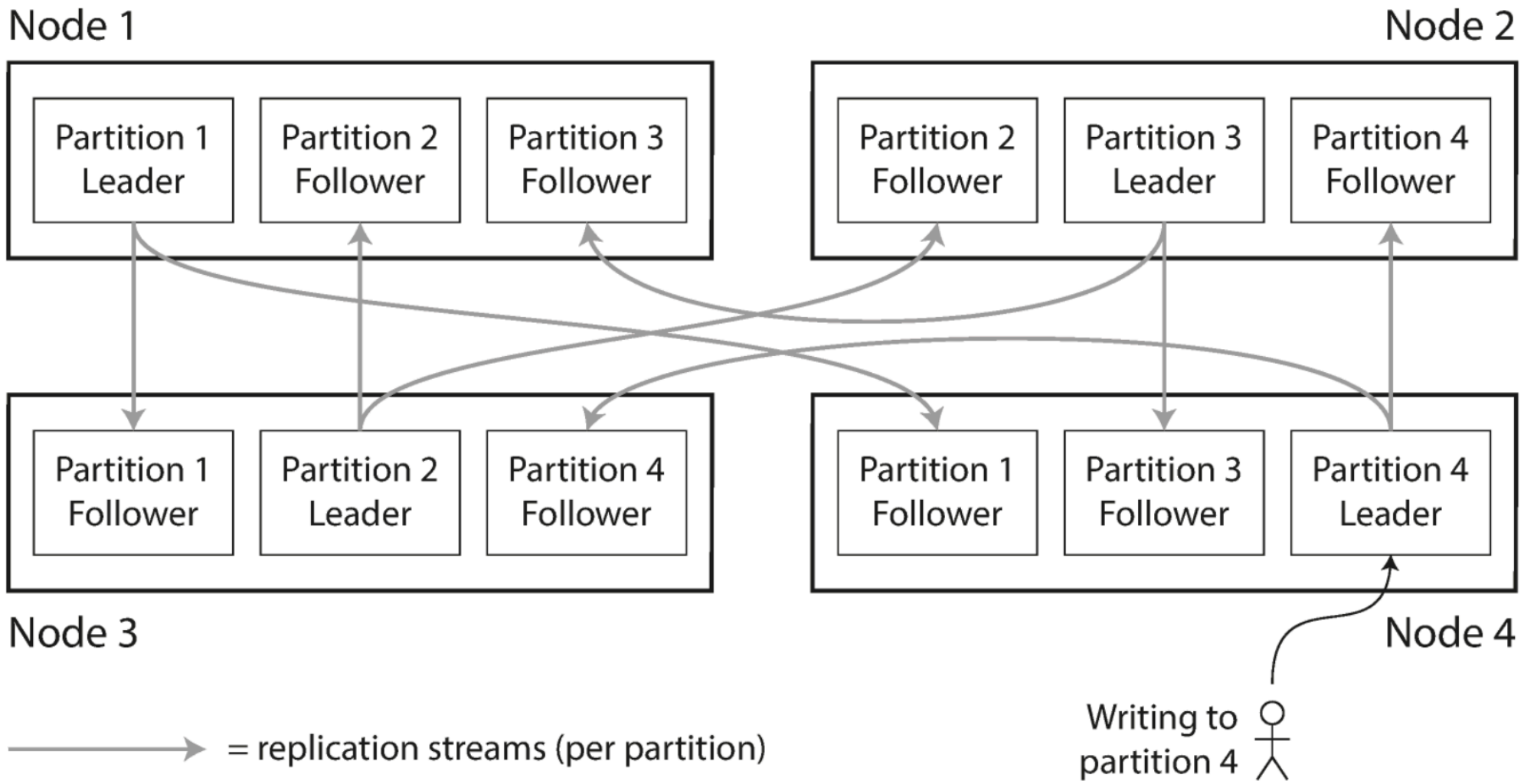
Question #2: How do we efficiently rebalance?

Ideally, we should be able to add or remove partitions without creating a lot of downtime and without moving a lot of data.

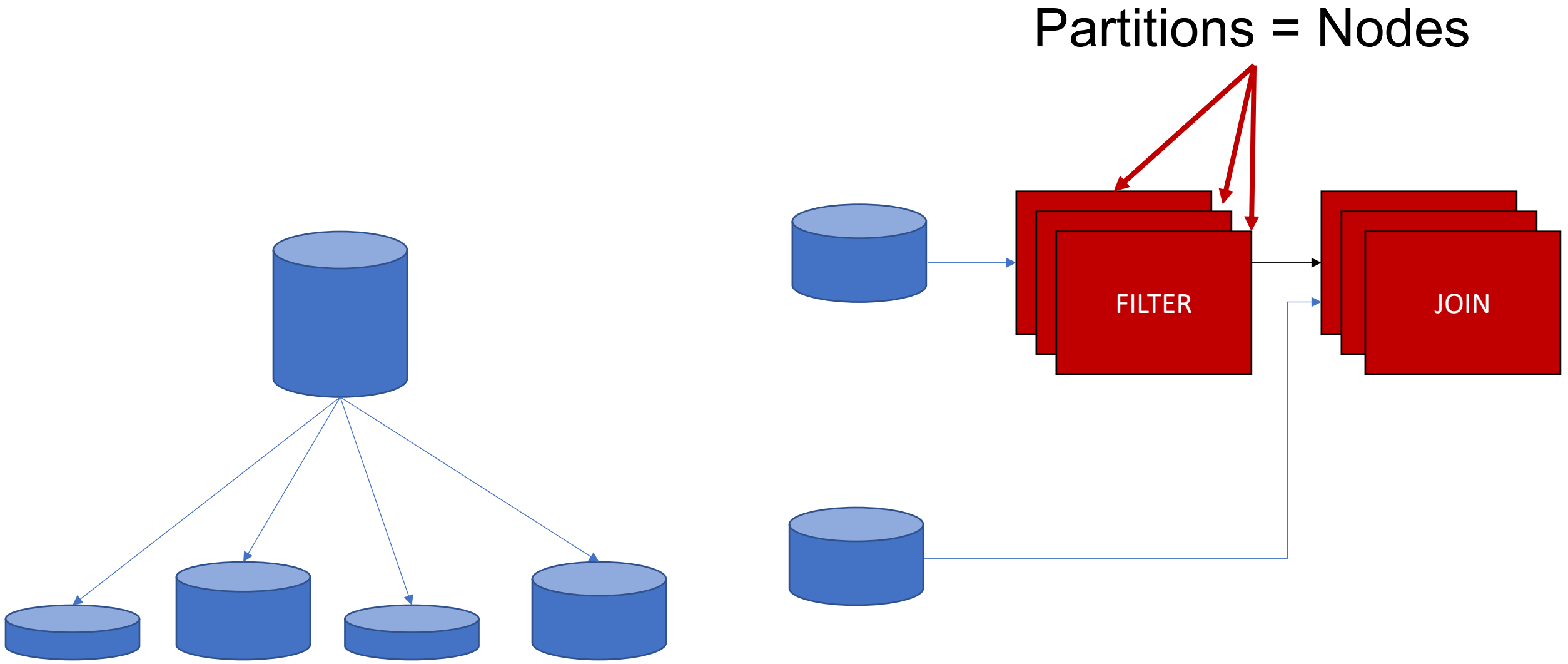




Partitioning with Replication



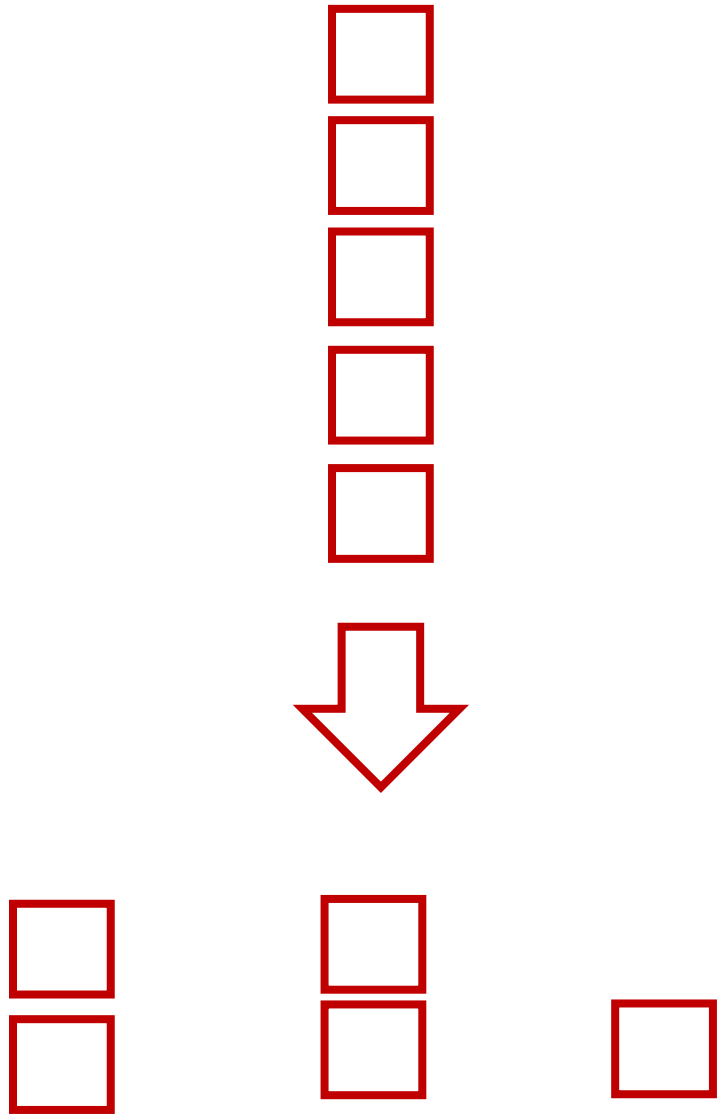
Partitioning contexts: Storage vs. Data flow



Partitioning Randomly or by Round-Robin

Deal items to each node (or pick a destination node randomly)

Advantage: Guarantees even or nearly even distribution of data, avoiding hotspots and data skew

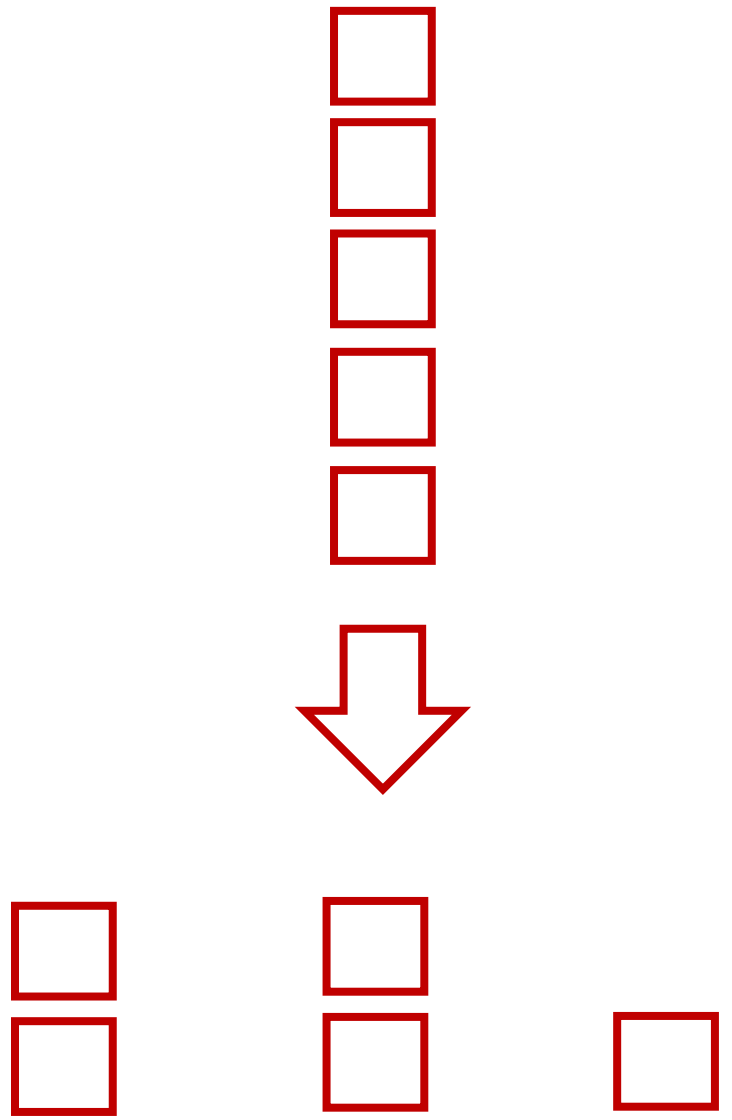


Partitioning Randomly or by Round-Robin

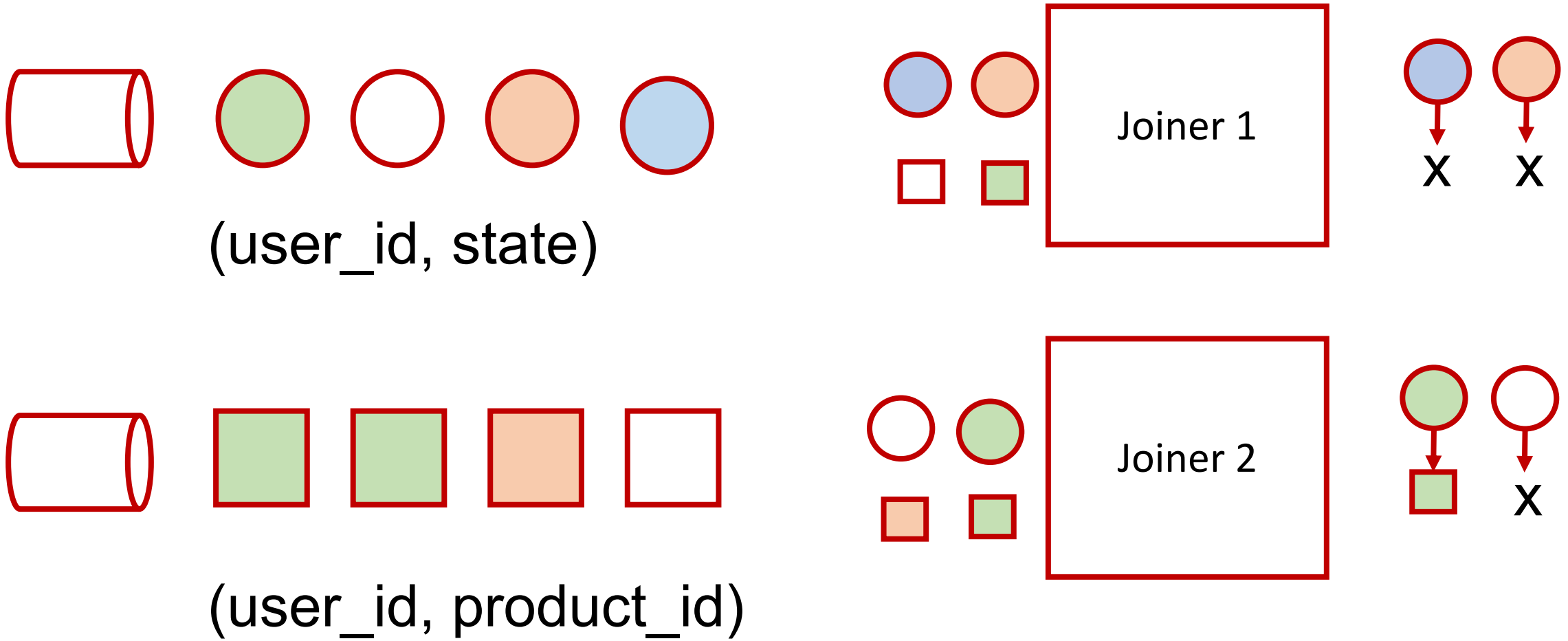
Disadvantage:

STORAGE CONTEXT: We don't know which partition or shard contains our data so we have to query each shard in parallel.

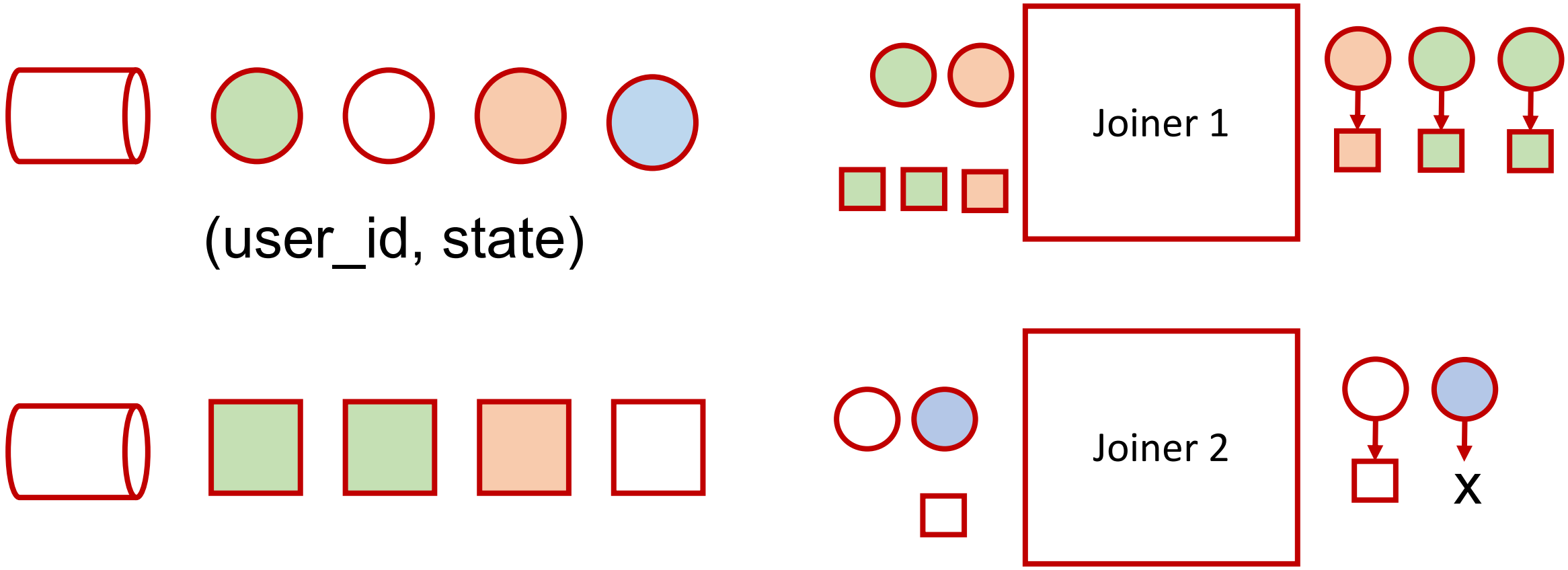
DATA FLOW CONTEXT: Certain operations (like joins) no longer work! Depending on the type of join you might end up with too many or two few records.



Joining Data: Partitioned randomly (Lost Records!)

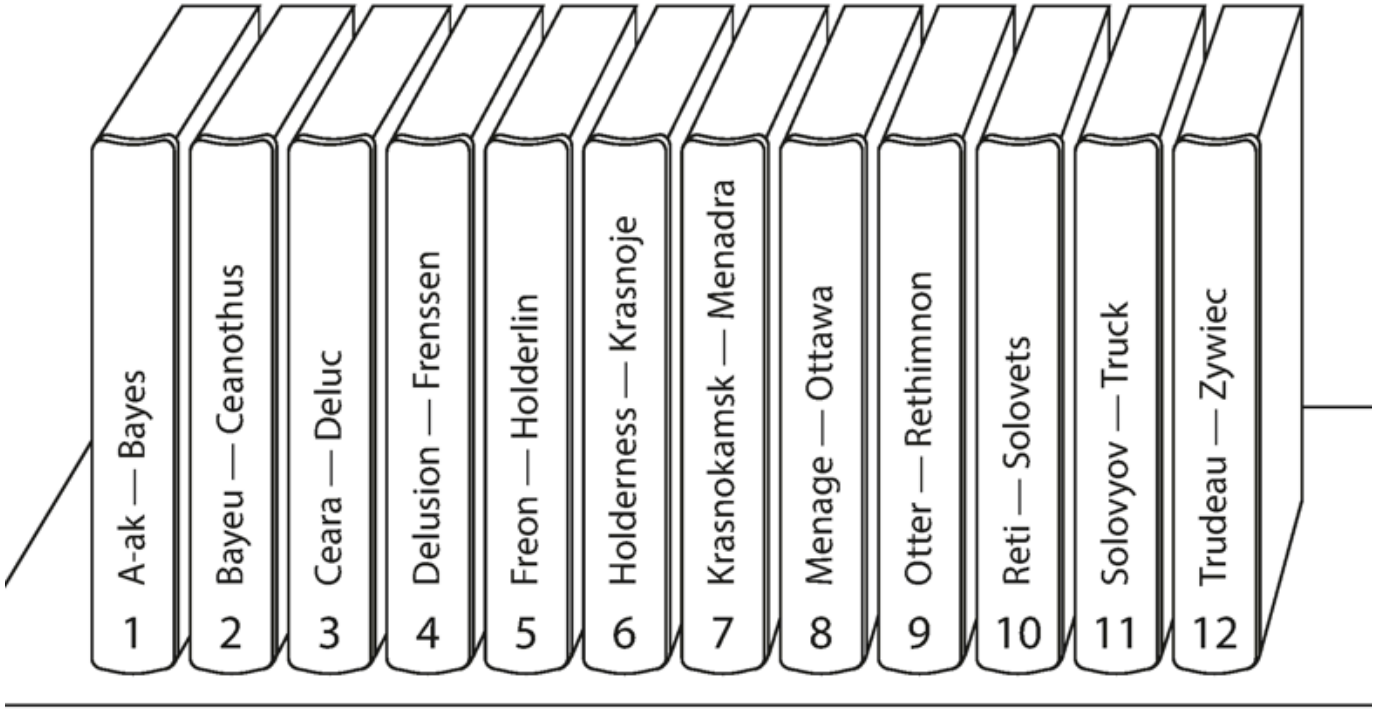


Joining Data: Partitioned by a common key



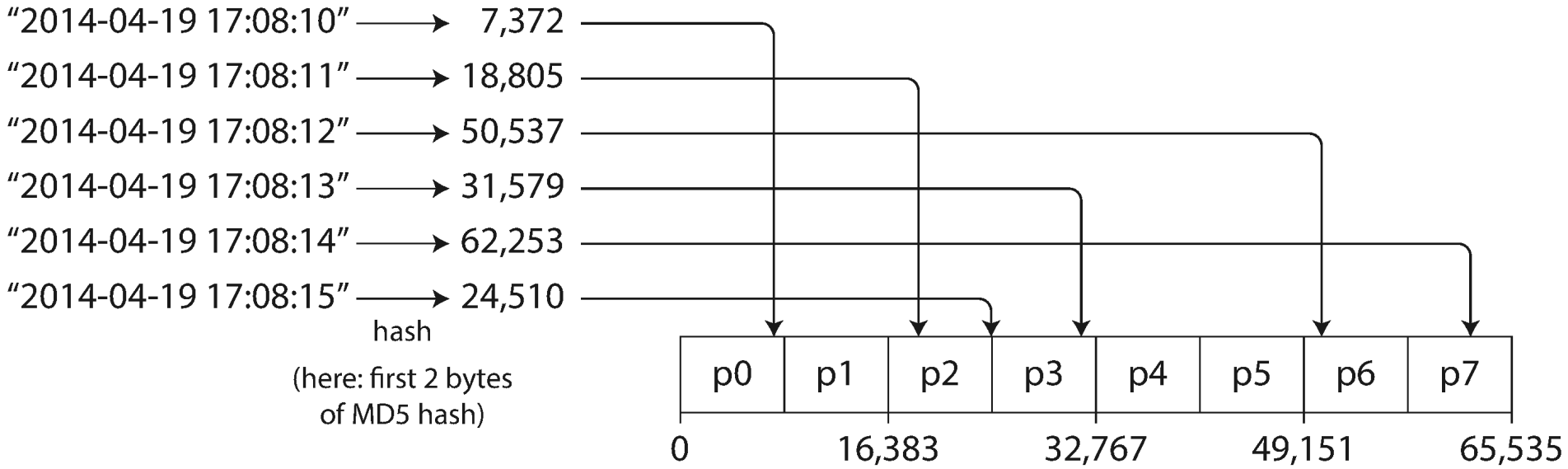
Partitioning by Key Range

- Achieves balancing if key range boundaries are chosen appropriately
- Eliminates the need to query multiple partitions in parallel. You know exactly which partition holds the value for your key
- If data segments are sorted by key, (as we saw with SSTables, for example) range queries are especially efficient.
- **Use case**: Sensor data keyed on date and you want to retrieve all data for a particular month. Efficient Read BUT Writes can become a bottleneck. Why?



Partitioning by hash(key) range

Hash: *String* → $[0..2^{31} - 1]$



Why? Hash function will evenly distribute keys. For example, even very similar strings will have very different hash values.

But what is the disadvantage? **You lose the ability to do range queries on the original key!**

Partitioning by Key: *hash*(key) mod N

Key → hash(key) % N, for N nodes
Assigns each key / value to a node: 0..N-1

ADVANTAGES

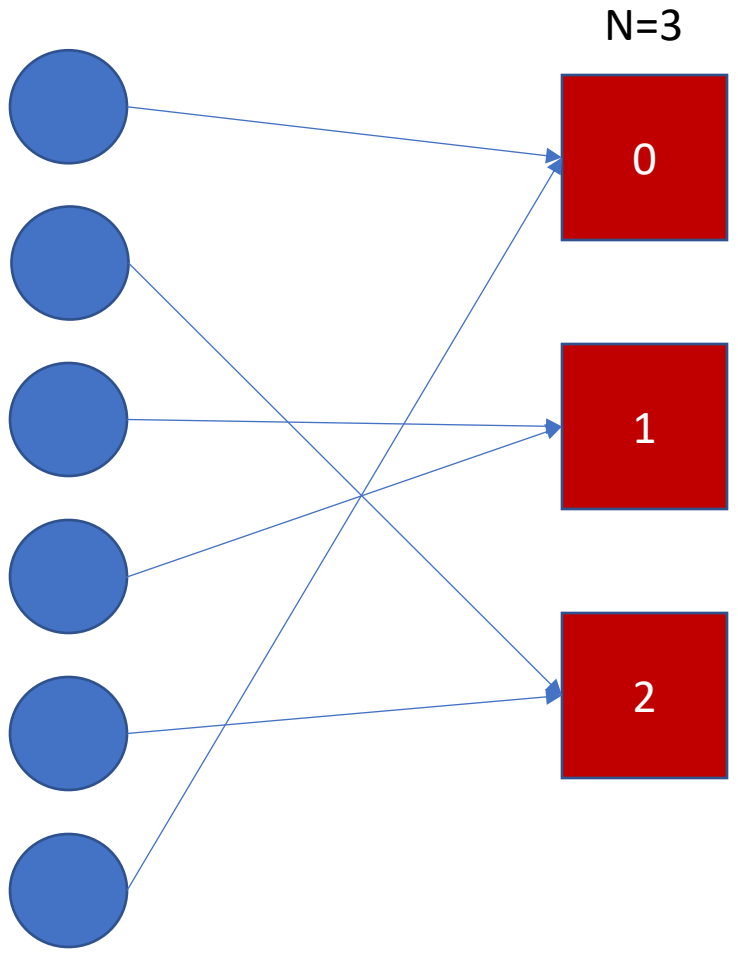
- Common in data flow systems
- Works with composite keys too
- Simple

DISADVANTAGES / POSSIBLE ISSUES

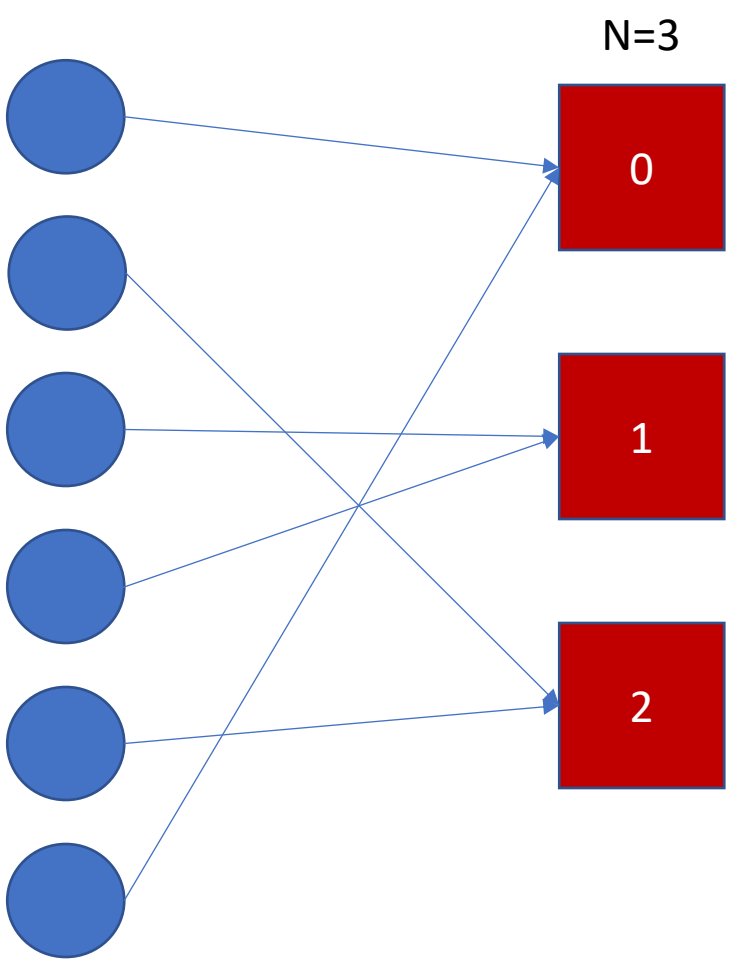
A) Choose hash function carefully!

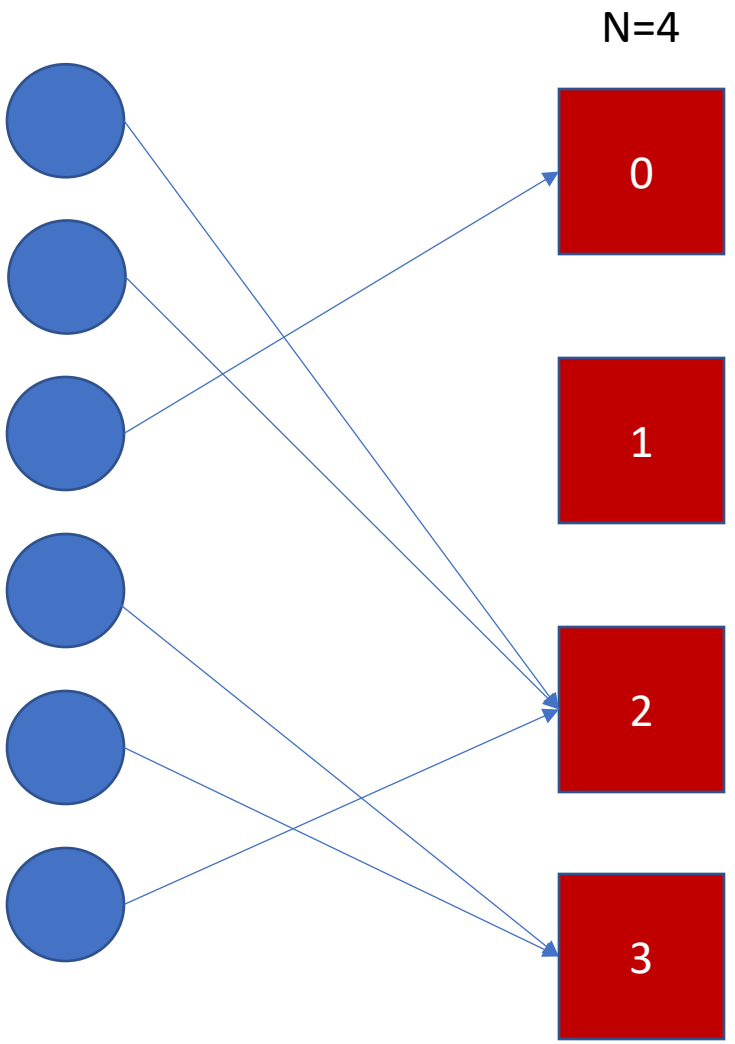
B) Can lead to skew. E.g., partitioning orders by STATE (CA overrepresented, DE underrepresented)

C) A change of N leads to redistribution of all the keys!

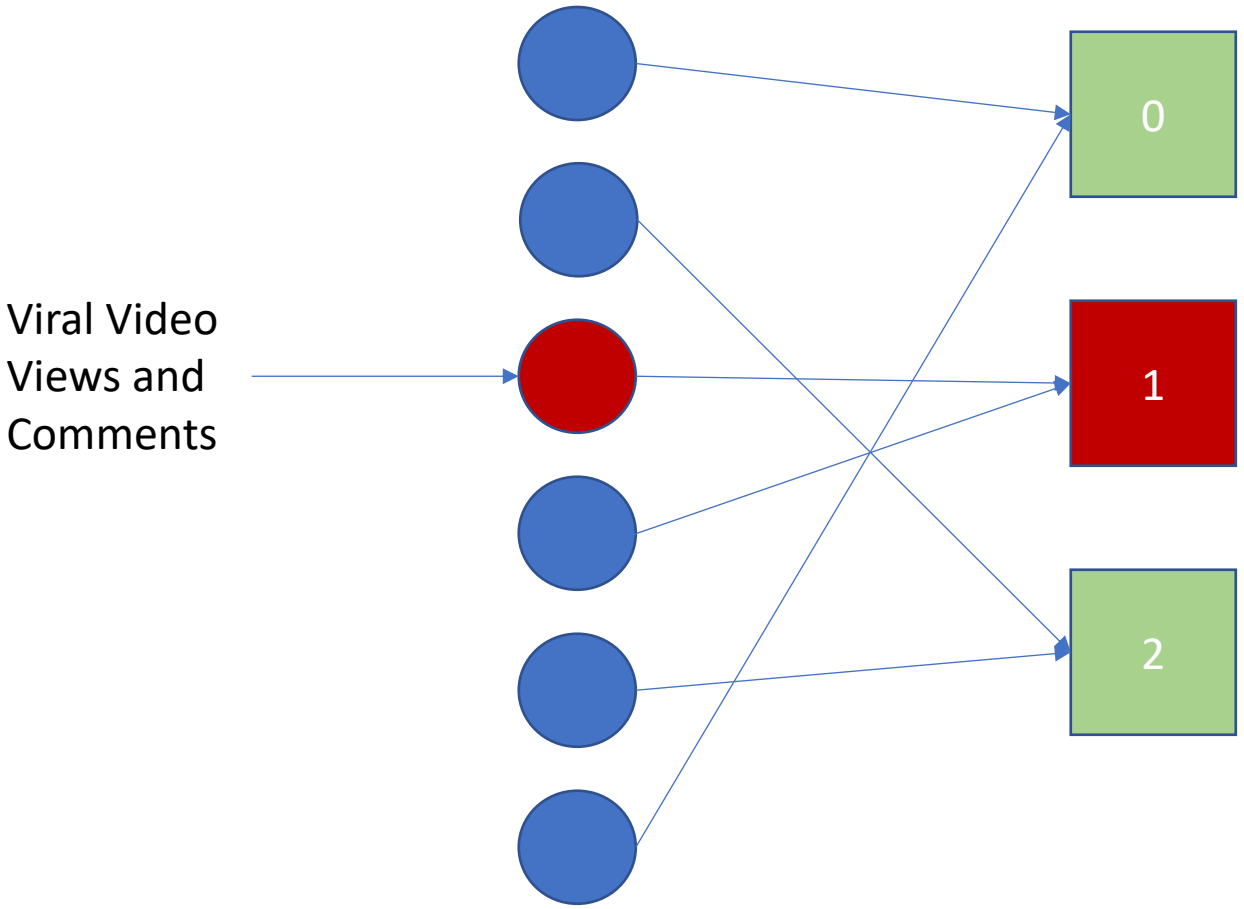


Partitioning by Key: *hash*(key) mod N





Dealing with hotspots with hashed keys



Hash (key) % N

Hash(*XX* + key) % N
or
Hash(Key + *XX*) % N

Secondary Indexes

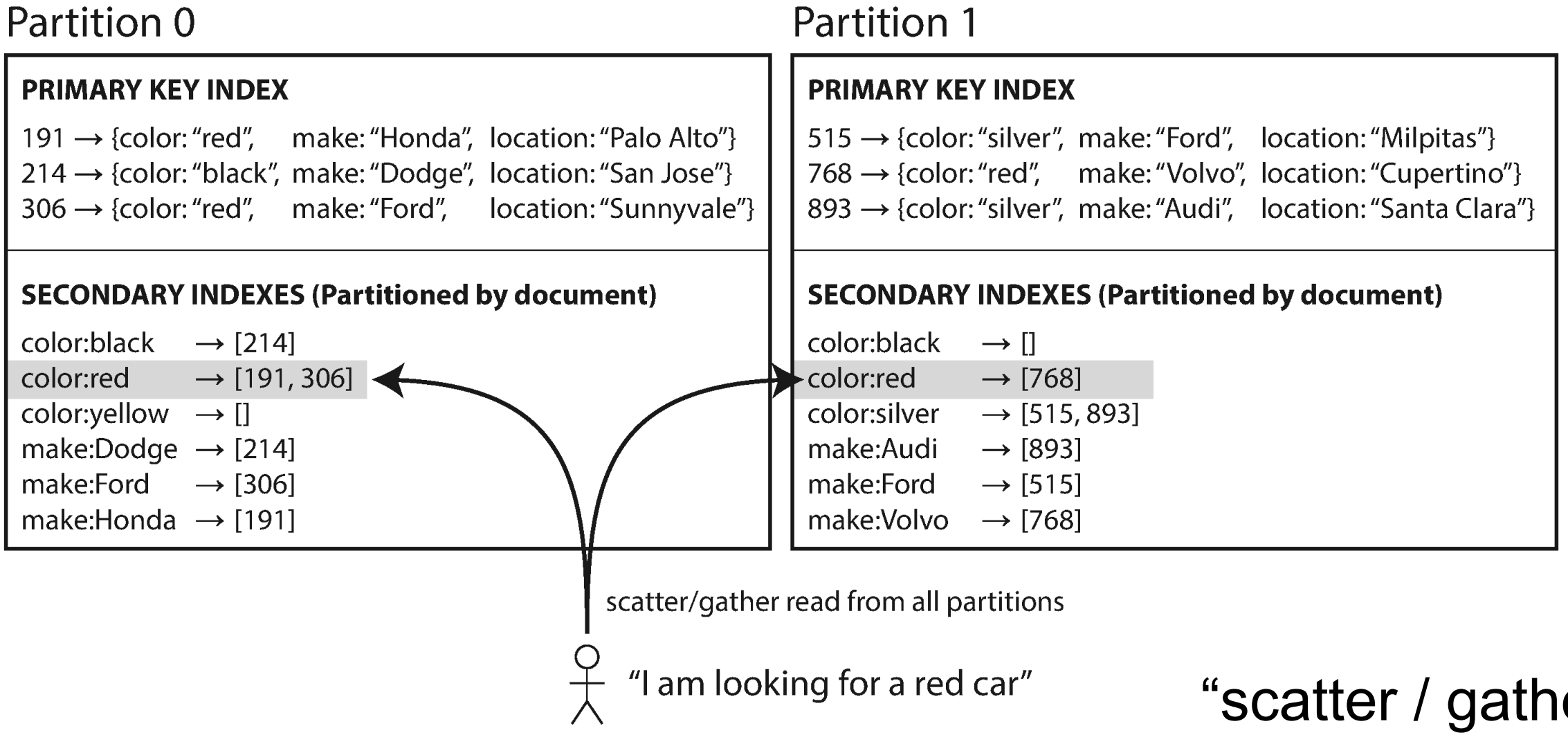
Find Watches by Brand or Style or Size…
Find Restaurants by Location or Rating …
Find Products who description contains some keyword…
Find movies by Genre or Rating or Actor or Director…

Important difference between Key-Value Stores and Document Stores:

Key Value Store: Partitioning data (by Key) is relatively straight-forward since we only try to access data BY KEY

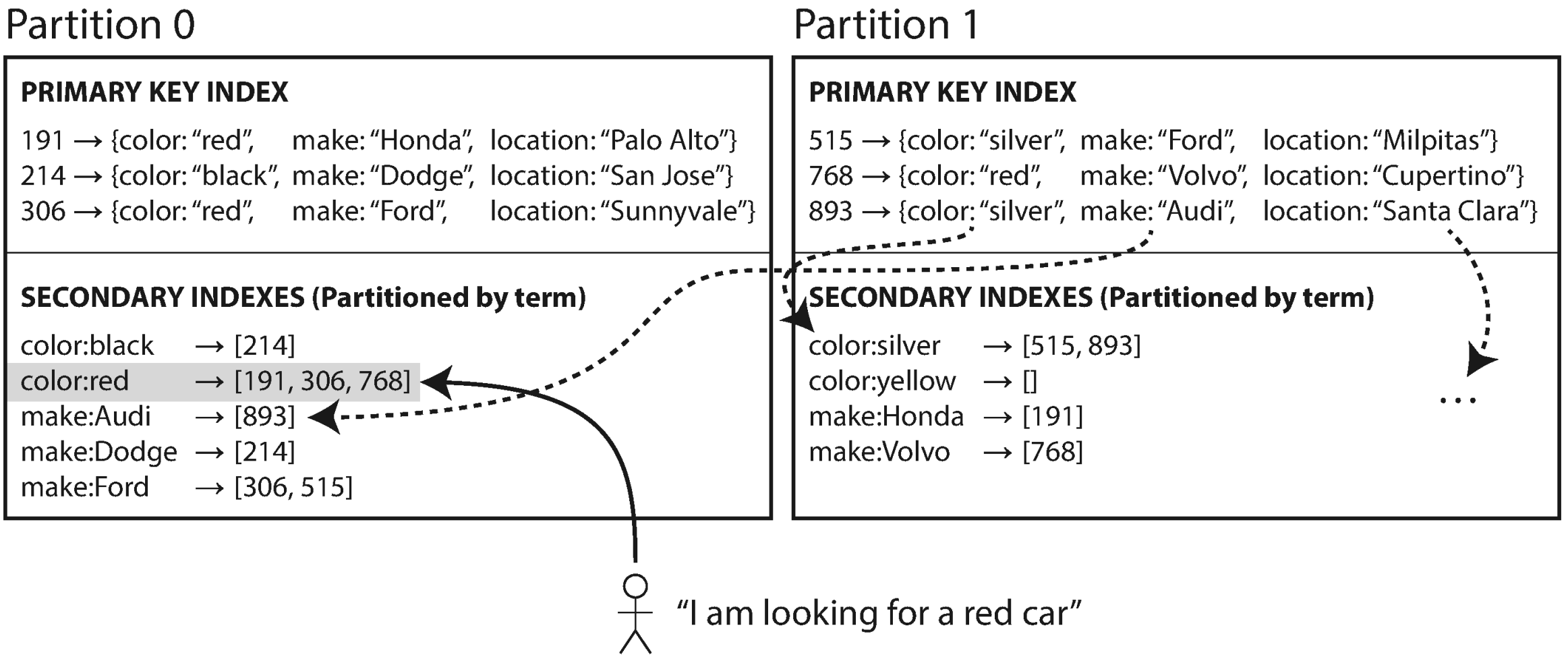
Document Store: We can partition by the document ID to achieve greater scalability, but when we search by some secondary index we are still faced with having to retrieve data from many different partitions.

Partitioning Secondary Indexes by Document

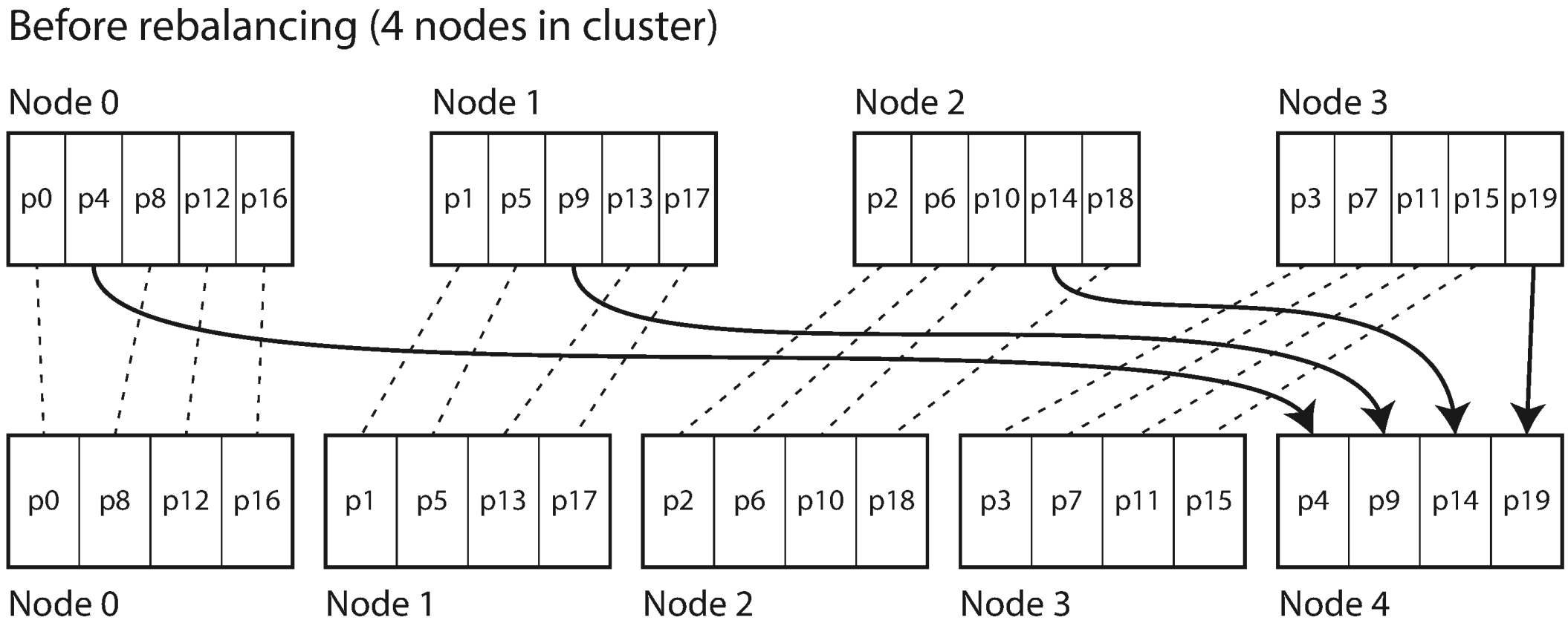


"scatter / gather"

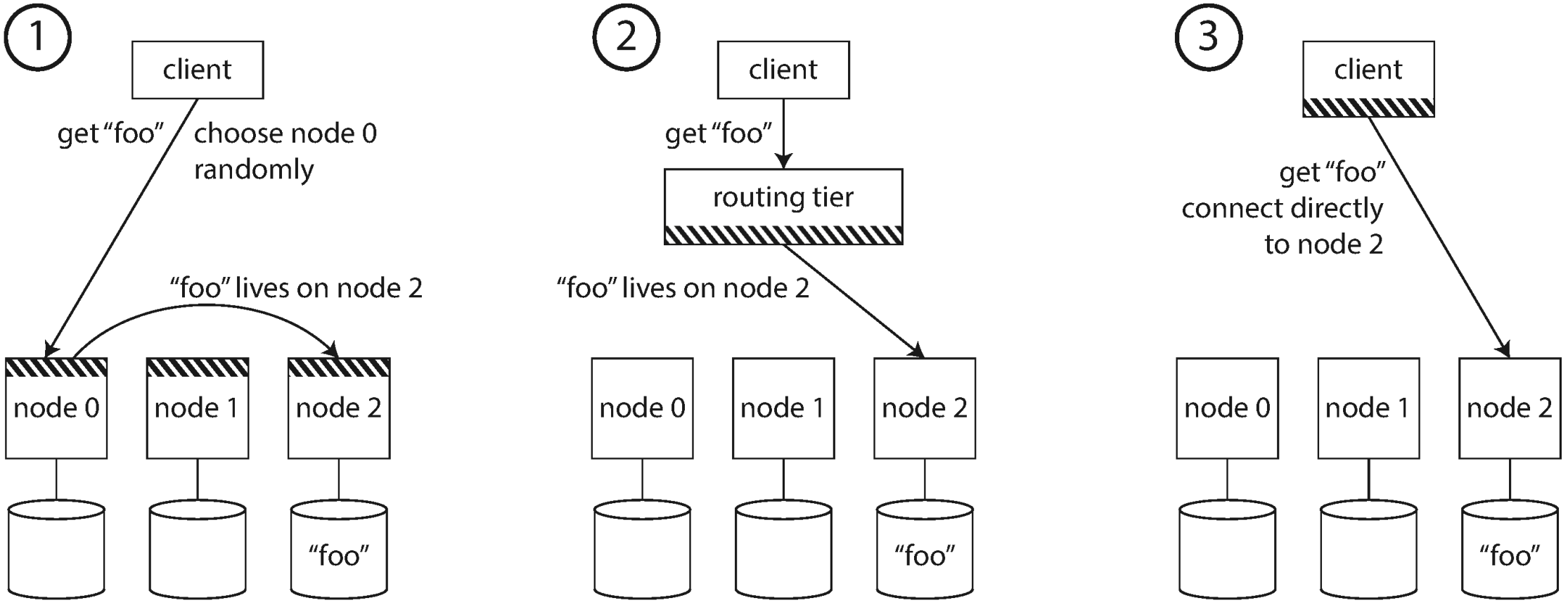
Partitioning Secondary Indexes by Term



Rebalancing Nodes with Fixed (Virtual) Partitions

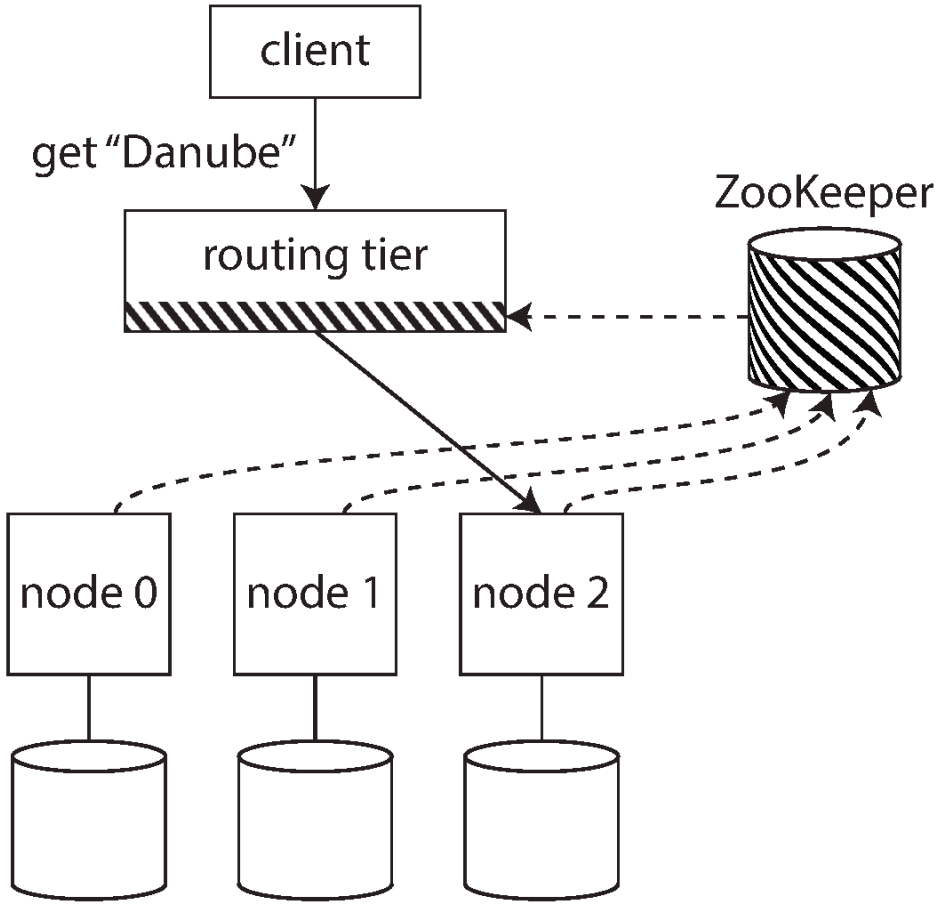


Routing Requests



= the knowledge of which partition is assigned to which node

ZooKeeper: Managing Cluster Metadata



Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

 = the knowledge of which partition is assigned to which node