CAP Theorem & KVS

Martijn de Vos

Week 10 - CS-460

Modern Web Workloads

- Web-based applications cause spikes
- Data: large and unstructured
- Random reads and writes; sometimes write-heavy (e.g., finance apps)
- Joins infrequent

Challenges with RDBMS

- Not designed for distributed environments
- Scaling SQL is expensive and inefficient

👉 This shift in workload demands gave rise to NoSQL

EPFL

NoSQL

= Not only SQL

<u>Avoids:</u>

- Strict ACID compliance
- Complex joins and relational schemes

<u>Provides:</u>

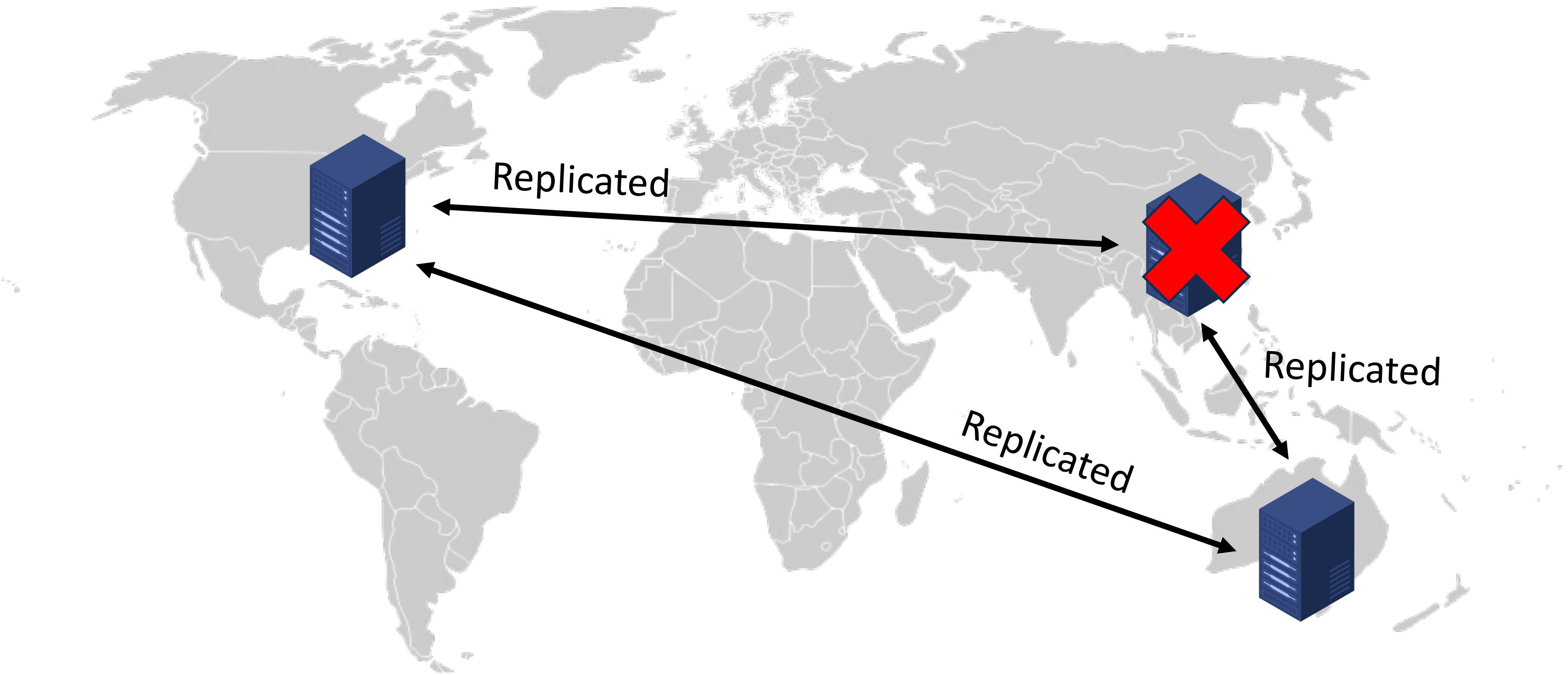
- Scalability
- Easy and frequent changes to DB
- Large data volumes

⚠ No free lunch

Weaker consistency guarantees, limited query expressiveness

Availability

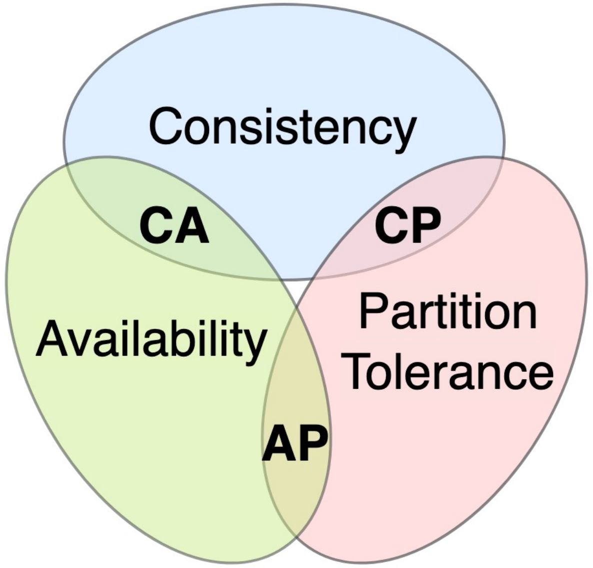
- **Data replication** improves availability in case of failures
  - By storing the same data in more than one site or node



The CAP Theorem*

In a distributed system you can <u>satisfy at most 2 out of 3 guarantees</u>:

1. **Consistency**: every read receives the most recent write or an error
2. **Availability**: every request received by a non-failing node in the system must result in a (timely) response
3. **Partition tolerance**: the system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network



* Proposed by Eric Brewer (Berkeley) in 2000, proved by Gilbert (NUS) and Lynch (MIT) in 2002

Why does Consistency Matter?

Consistency: every read receives the most recent write or an error

Use Case	What you expect (consistency)	What could go wrong (inconsistency)
🏦 Banking app	Transfer €500 via your phone, it instantly shows up on your desktop app too.	Your balance looks updated on your phone but not on your desktop.
✈️ Booking a flight	A seat is shown as unavailable right after someone else books it.	Two users book the **same seat** at once.
🛒 Online shopping	You remove an item from your shopping cart and it's instantly reflected everywhere.	You get charged for the same item because a device has stale cart data.

Why does Availability Matter?

Availability: every request received by a non-failing node in the system must result in a (timely) response

Reliability

Users expect services to work 24/7

- A 500ms delay on Amazon → 20% revenue loss
- If checkout fails, users can abandon their purchase

Speed = Money

Latency kills engagement

- Amazon: every extra 100ms → millions lost
- Google: longer load time → fewer searches → lost revenue

Cognitive Drift

Humans are impatient

- 1s of delay and users mentally move on
- Responsiveness is key to user flow and retention

Why does Partition Tolerance Matter?

Partition tolerance: the system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network

Event	Example	Impact
Internet router outage	Data center ISP failure	Servers are not reachable anymore
Undersea cable cut	SEA-ME-WE 5 cable incident (2024)	Connectivity loss between regions
DNS outage	Dyn DDOS attack (2016)	Users can't resolve hostnames
BGP configuration error	Facebook outage (2021)	Outage of Facebook and subsidiaries

Take-away: partitions actually happen in real-world settings

EPFL

CAP Combinations

CA: Consistency + Availability	AP: Availability + Partition Tolerance	CP: Consistency + Partition Tolerance
✅ Strong consistency	✅ Always available	✅ Strong consistency
✅ Always available	✅ Operational under partitions	✅ Operational under partitions
❌ Fails on partition	❌ May return stale data	❌ May deny some requests
💡 *Cannot exist in practical distributed settings*	💡 *Example:* Cassandra	💡 *Example:* ZooKeeper

CAP in Practice

- 2 out of 3 is somewhat misleading
  - Partition tolerance is **non-negotiable** in real systems, we need it
  - So the real choice is between Consistency and Availability

- Traditional RDBMSs → Consistency, Partition Tolerance
- NoSQL → Availability, Partition Tolerance

💡 Availability prioritizes user experience, consistency prioritizes correctness

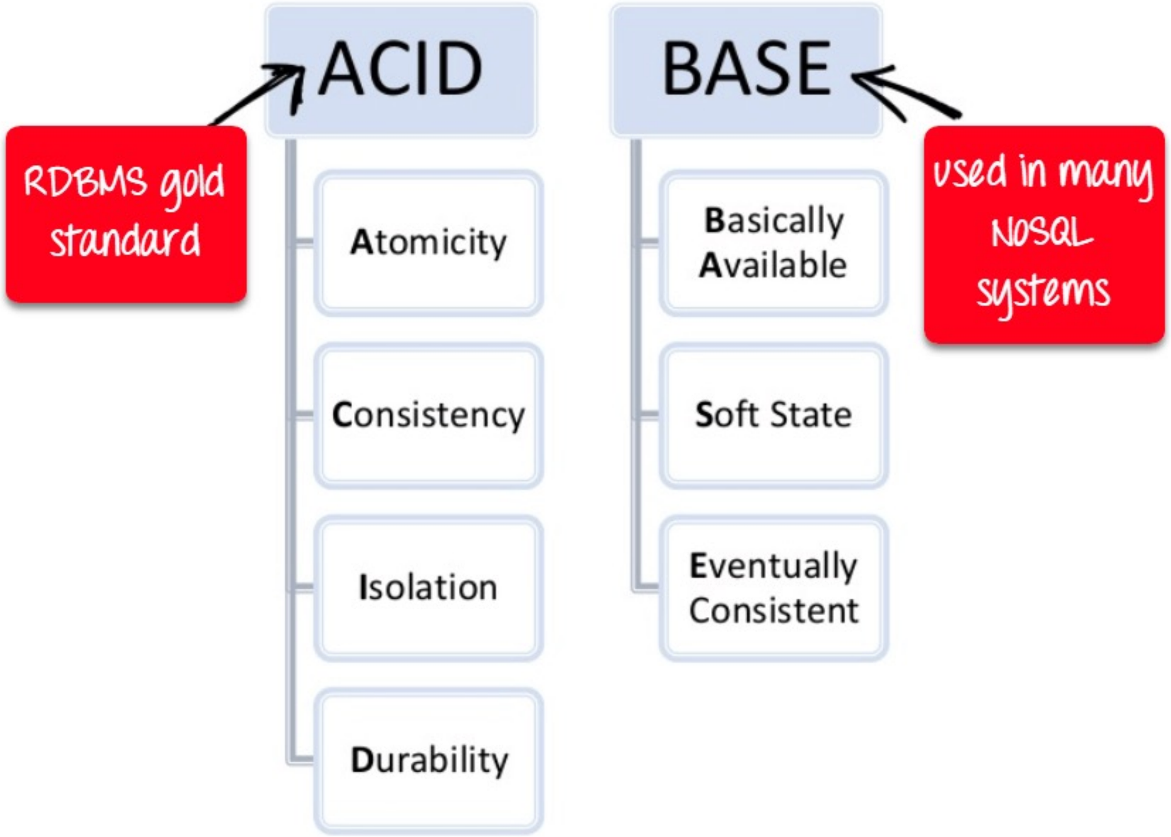
ACID vs. BASE – The Tradeoff in Modern Systems

- You can't have ACID properties and high availability under network partitions
- BASE systems embrace this, trading strict consistency for availability and scalability

💡 ACID is like a strict accountant, BASE is like a bar tab.

BASE Properties

- Basic Availability
  - Possibilities of faults but not a fault of the whole system
- Soft-state
  - Copies of a data item may be inconsistent
- Eventually consistent
  - Copies becomes consistent at some later time if there are no more updates to that data item



[https://www.guru99.com/sql-vs-nosql.html]

Key Takeways

1. Choose the right guarantee for the right task (CP vs. AP)
2. Partition tolerance is non-negotiable in the CAP theorem
3. ACID for RDBMS, BASE for NoSQL systems
4. Different applications might need different consistency guarantees

References

- Theorem first presented as a conjecture by Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC
- Seth Gilbert and Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", *ACM SIGACT News*, Volume 33 Issue 2 (2002), pg. 51–59.
- Eric Brewer, "CAP twelve years later: How the 'rules' have changed", *Computer*, Volume 45, Issue 2 (2012), pg. 23–29.

Key-value stores

Serving Today's Workloads

Performance
- Speed (req/s.)
- Scale out, not up

Reliability
- Avoid single point of failure

Efficiency
- Low total cost of operation
- Fewer system administrators

Scalability
- Need to serve many users

The Key-value Abstraction (1/2)

Key-value is a powerful abstraction powering the modern web

Key	Value
`post_id` (x.com, facebook.com)	Post content, author, timestamp
`item_id` (amazon.com)	Name, price, stock info
`flight_no` (expedia.com)	Route, availability, price
`account_no` (bank.com)	Balance, transactions, owner

The Key-value Abstraction (2/2)

- **A dictionary-like data structure**
  - Supports *insert*, *lookup*, and *delete* by key
  - Example: a local hash table
- But now, **distributed** across many machines
  - Designed to handle web-scale workloads
- Like Distributed Hash tables (DHTs) in P2P systems
- Key-value solutions reuse many techniques from DHTs
  - Consistent hashing, replication, partitioning, …

? How can we effectively locate and retrieve a key in a large, distributed database?

Key-value/NoSQL Data Model

- Core operations: get(key) and put(key, value)
- Storage model: tables, but more flexible
  - Called *column families* (Cassandra), *tables* (Hbase), *collections* (MongoDB)
- Unlike traditional RDBMS tables:
  - May be schema-less: each row can have different columns
  - Does not always support joins or foreign keys

Design of a real key-value store, Cassandra

Released in 2008, after Dynamo (2007) and BigTable (2006)

Cassandra

- A distributed key-value store
- Many companies use Cassandra in their production clusters
  - IBM, Adobe, HP, eBay, Ericsson, Symantec, Twitter, Spotify, Netflix
- Scalable data model: data split across nodes
- CAP: availability and partition tolerance

Objectives

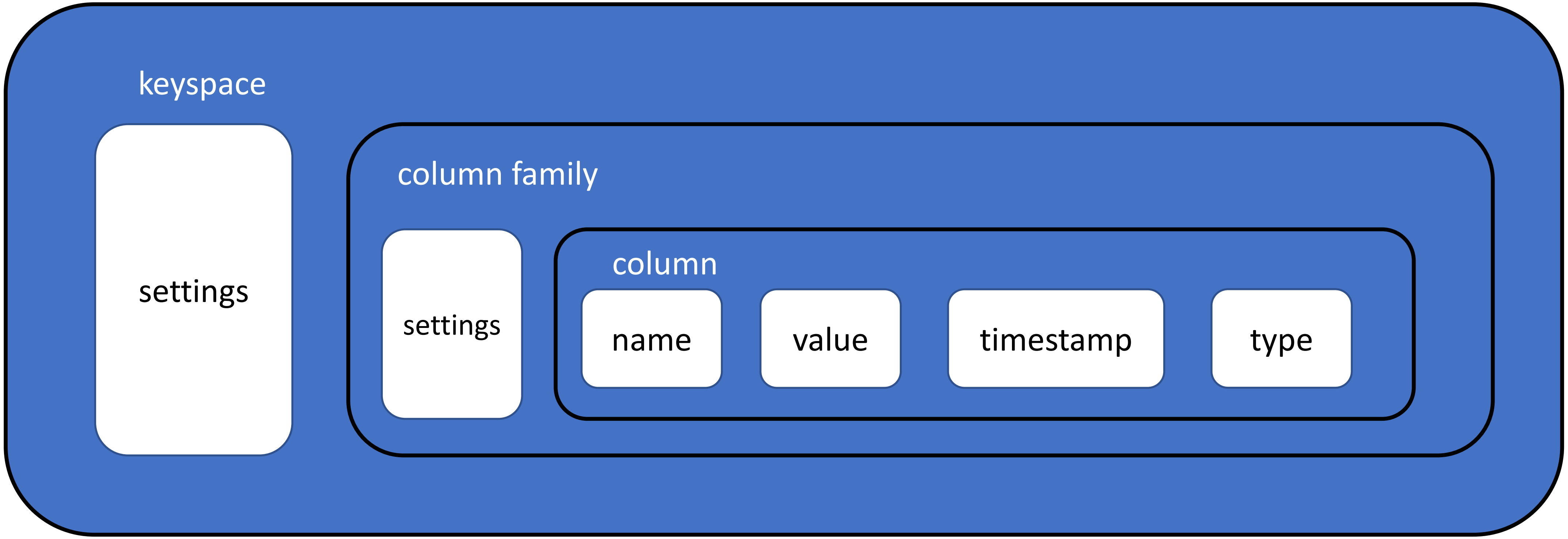
- Distributed storage system
- Targets large amount of unstructured data
- Intended to run in a datacenter (and also across DCs) across many commodity servers
- No single point of failure
- Originally designed at Facebook
- Open-sourced later, today an Apache project (2010)

- *But: does not support joins, limited support for transactions and aggregation*

Data model (1/4)

- Table in Cassandra: distributed map indexed by a key (can be nested)
- **Row**: identified by a Unique Key (Primary key)
- **Keyspace**: A logical container for column families that defines the replication strategy and other configuration options
- **Column Family**: A logical grouping of columns with a shared key, contains Supercolumns or Columns
- **Column**: basic data structures with a name, type, value, timestamp
- **Supercolumn**: stores a map of sub-columns. Columns that are likely to be queried together should be placed in the same column family

Data model (2/4)

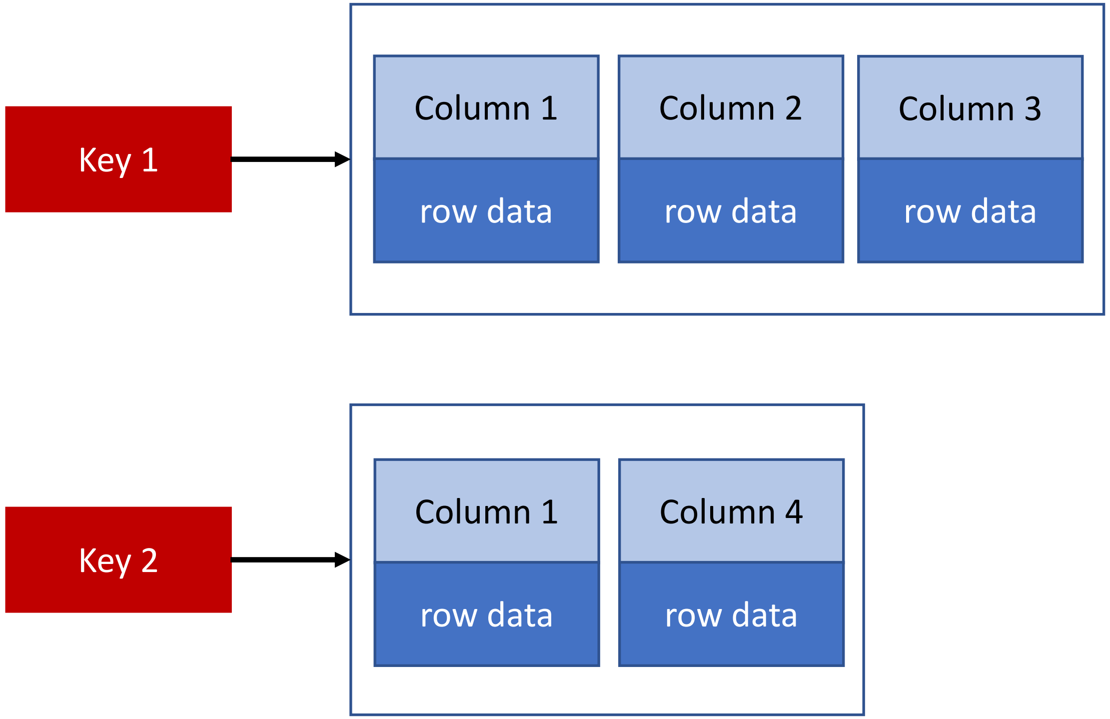


Data model (3/4)

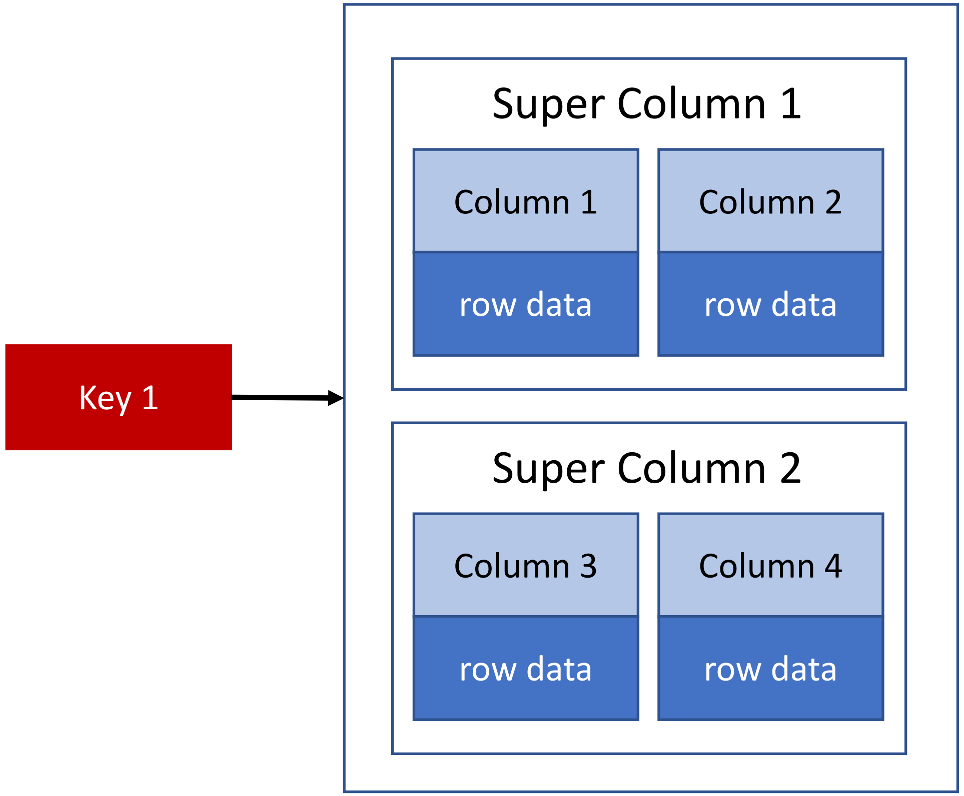
Feature	RDBMS	Cassandra
Organization	Database → table → row	Keyspace → column family → column
Row structure	Fixed schema	Dynamic columns
Column data	Name, type, value	Name, type, value, timestamp
Schema changes	Typically requires downtime	During runtime
Data model	Normalized with JOINs	Denormalized

Data model (4/4)

Simple Column family



Super Column family

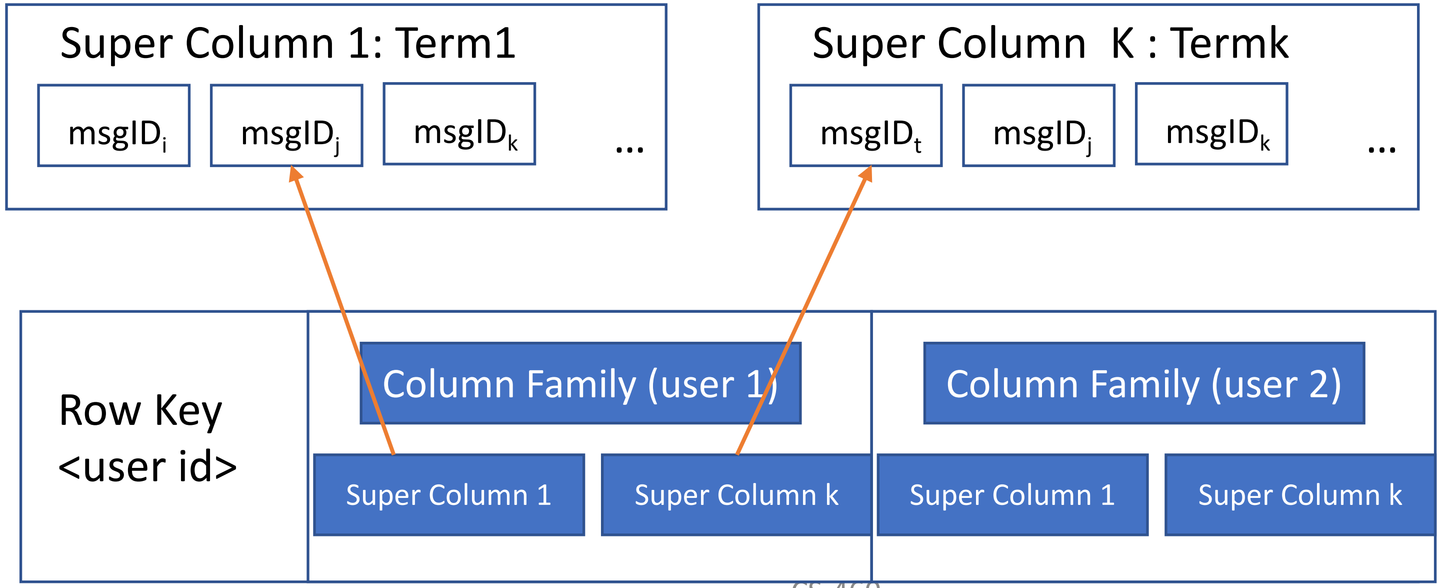


Facebook example

- Facebook maintains a per-user index of all messages exchanged between senders and receivers
- Two kind of search features enabled in 2008
  - **Search by term**
  - **Search by user**: given a user's name, returns all the messages sent/received by that user

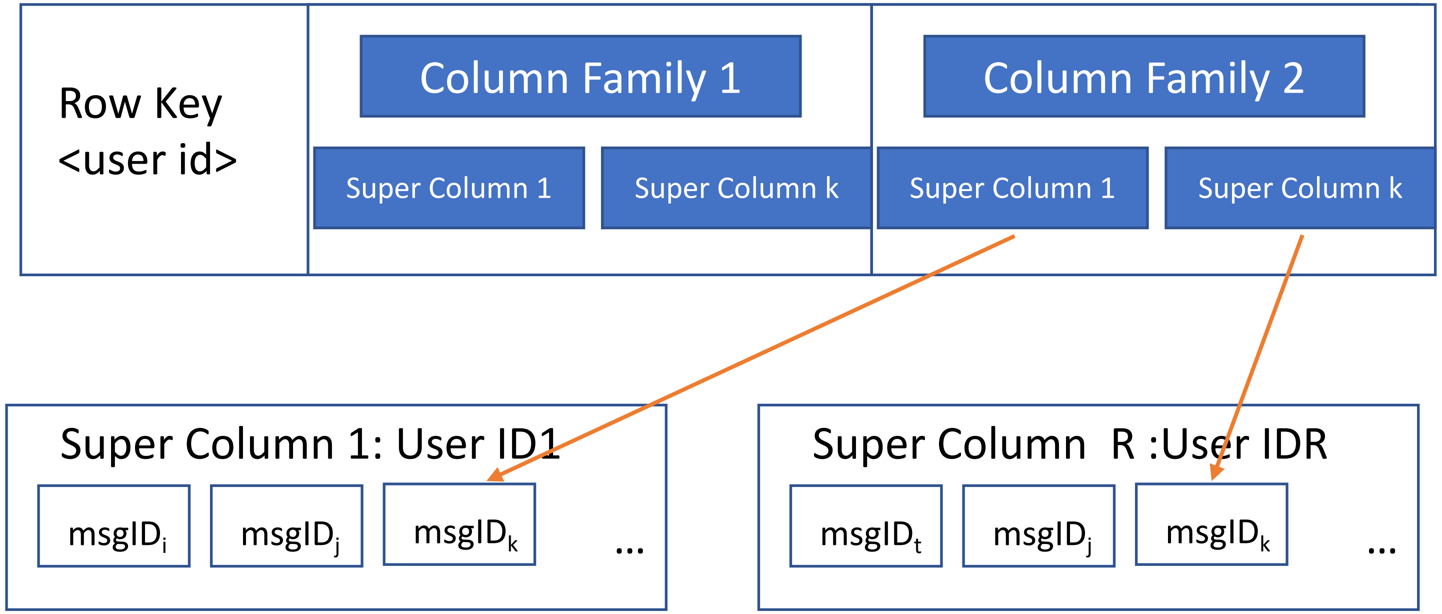
Facebook term search

- Primary key: *UserID*
- Words of messages: super columns
- Columns within the super columns: individual message identifiers (messageId) of the messages that contains the word

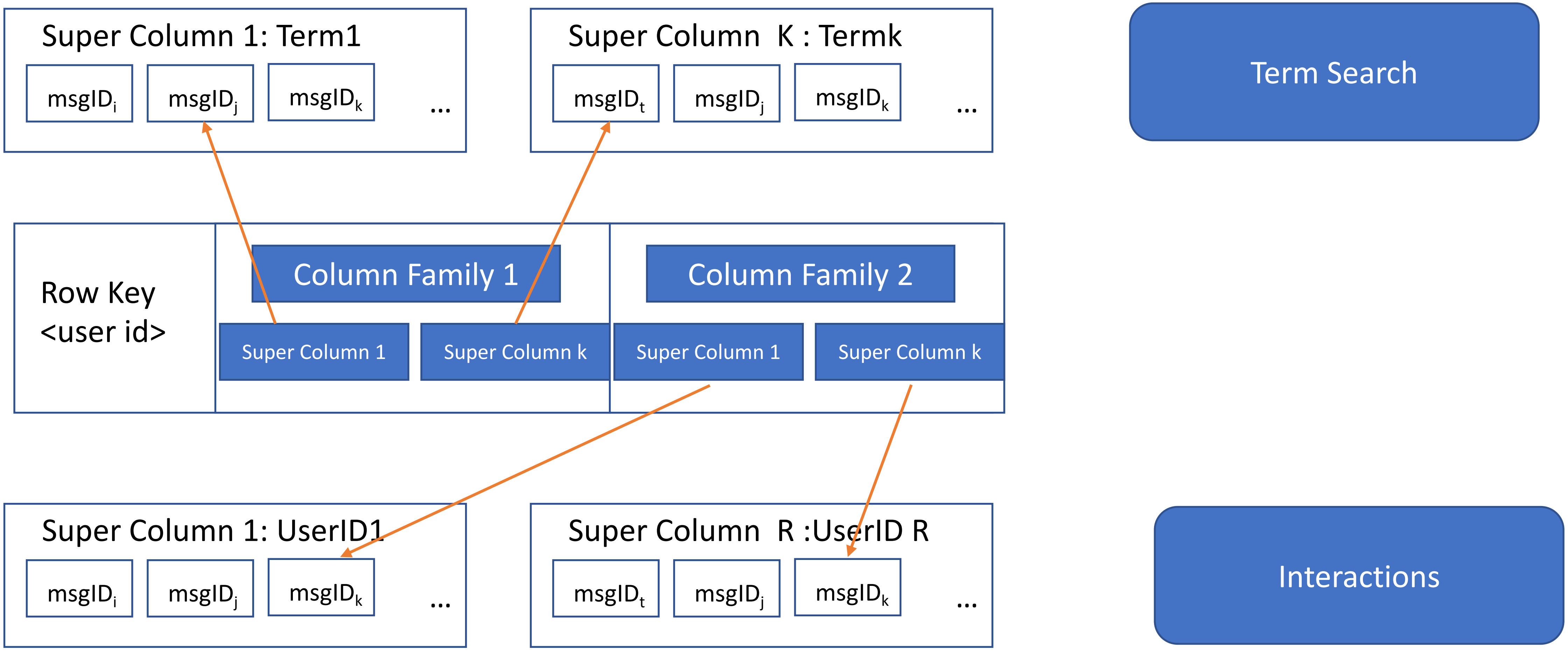


Facebook Inbox search

- Primary key: *UserID*
- Recipients ID's: super columns
- Columns within the super columns: messageId

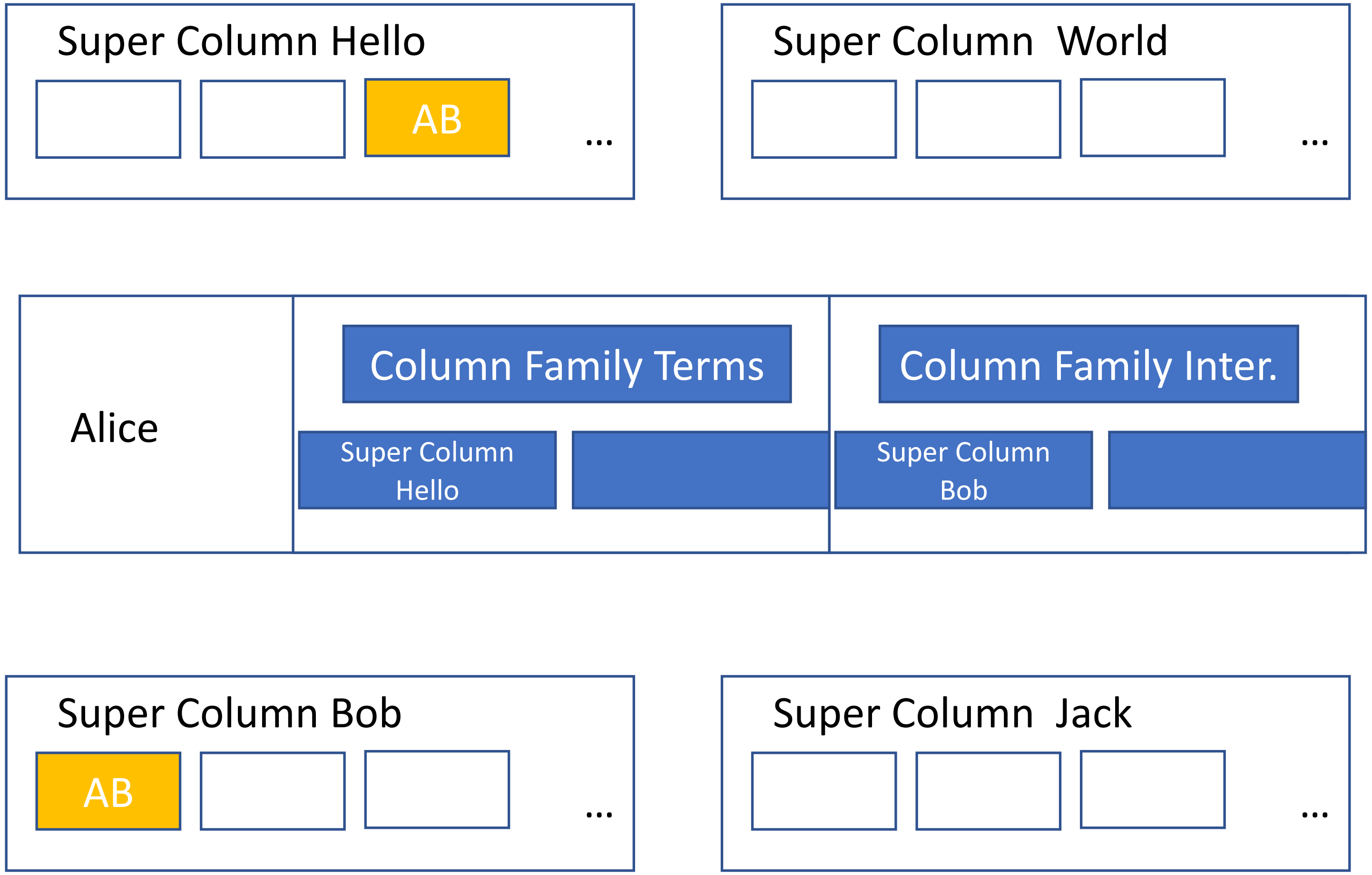


Schema


Super Column 1: Term1
msgID$_i$
msgID$_j$
msgID$_k$
…
Super Column K : Termk
msgID$_t$
msgID$_j$
msgID$_k$
…
Term Search
Row Key
<user id>
Column Family 1
Column Family 2
Super Column 1
Super Column k
Super Column 1
Super Column k
Super Column 1: UserID1
msgID$_i$
msgID$_j$
msgID$_k$
…
Super Column R :UserID R
msgID$_t$
msgID$_j$
msgID$_k$
…
Interactions

Example

Alice sends “Hello” to Bob (msgID: AB)



Cassandra Architecture

- Decentralized, peer-to-peer architecture
- Easy to scale: add/remove nodes
- Read/write requests can go to any replica node
- Reads and write have a configurable consistency level

Cassandra Architecture

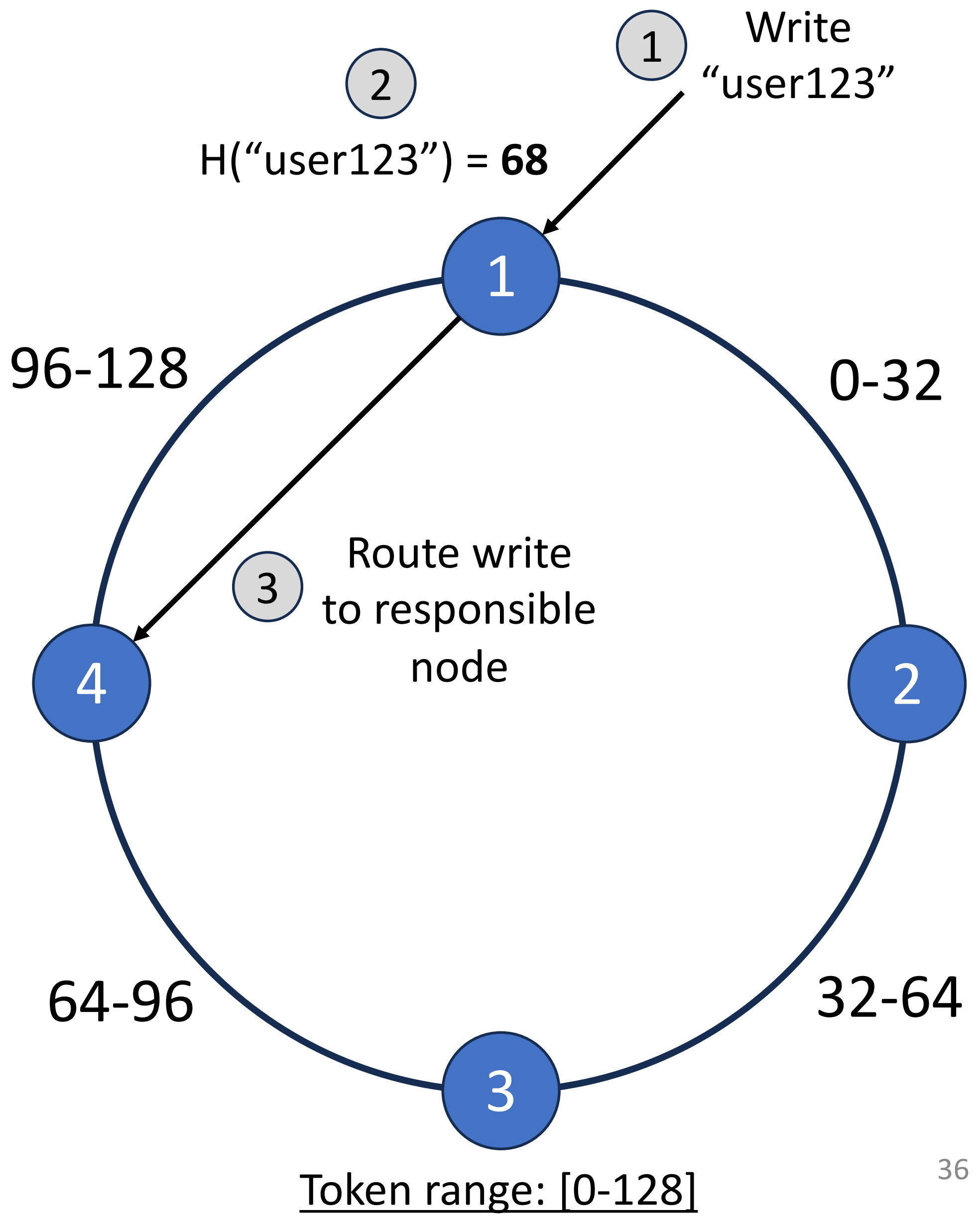
1. Partitioning
2. Load balancing
3. Replication
4. Writes and reads
5. Data structures
6. Membership management
7. Consistency

💡 Using terms node and replica interchangably

Cassandra: Partioning

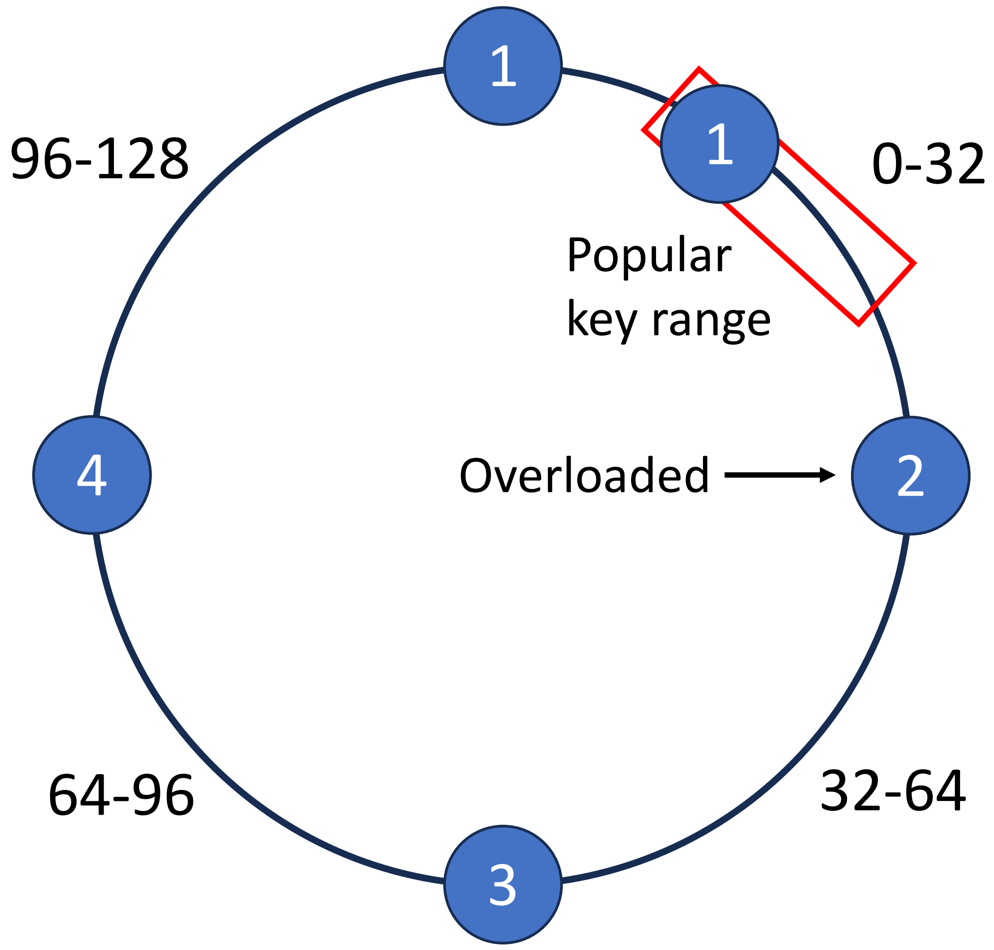
- Nodes are conceptually ordered on a clockwise ring
- Each node is responsible for the region of the ring between itself and its predecessor
- Example of a write without replication (right)

💡 Cassandra uses a ring-based DHT but without finger tables or routing



Cassandra: Load balancing

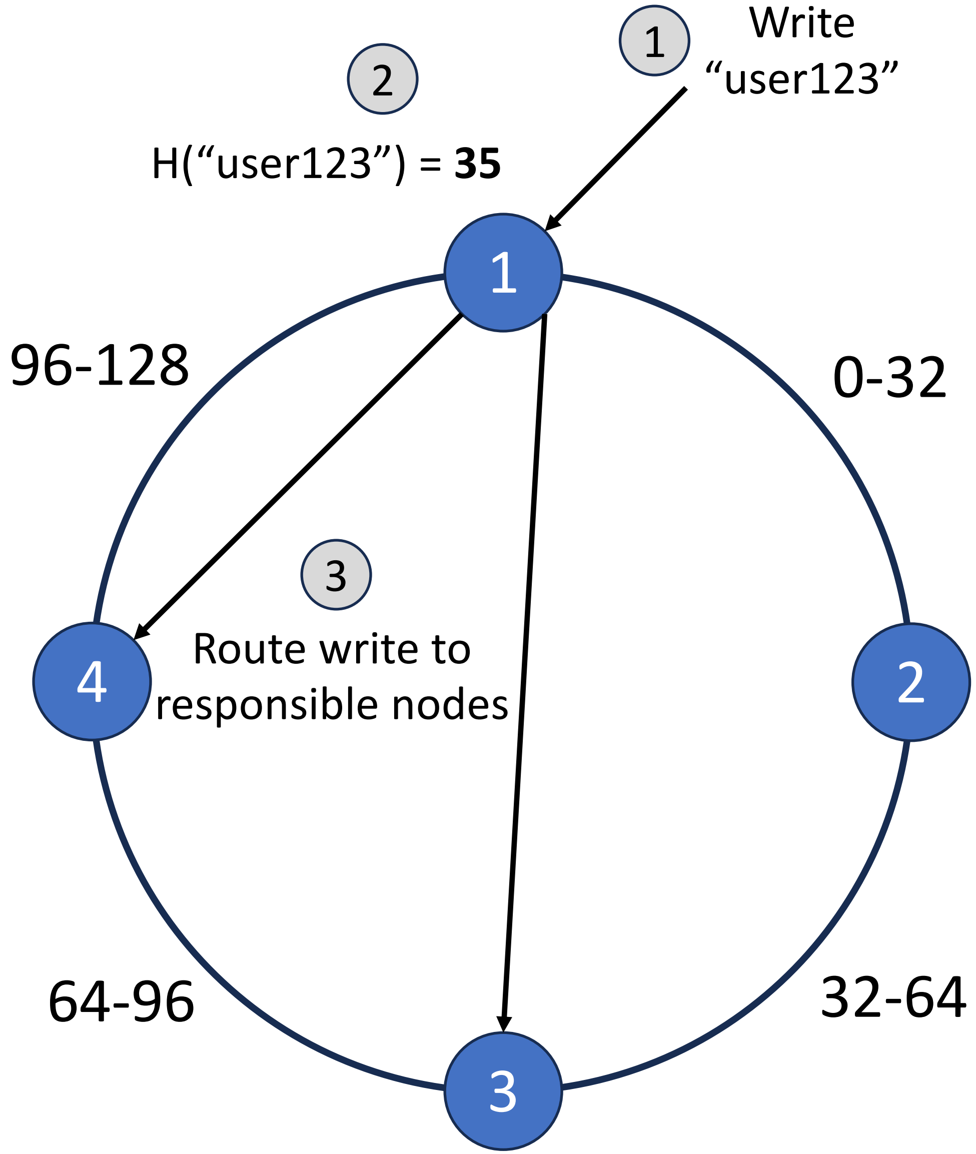
- Random partitioning leads to non-uniform data and load distribution
- Cassandra assumes homogeneous nodes' performance
- How is it addressed
  - Lightly loaded nodes move on the ring to alleviate loaded ones
  - Virtual nodes



Token range: [0-128]

Cassandra: Replication

- Replication factor *N*: determines how many copies of the data exist
- Each data item is replicated at *N* nodes
- Various replication strategies
- Example with *N*=2 (right)



EPFL

Cassandra: Replication Strategies

SimpleStrategy	NetworkTopologyStrategy
Used for single DC and rack	For deployment across different DCs
Easy setup	Tunable replication factor per DC

```
CREATE KEYSPACE cluster1 WITH
replication = {'class’:
'SimpleStrategy',
'replication_factor': 2};
```

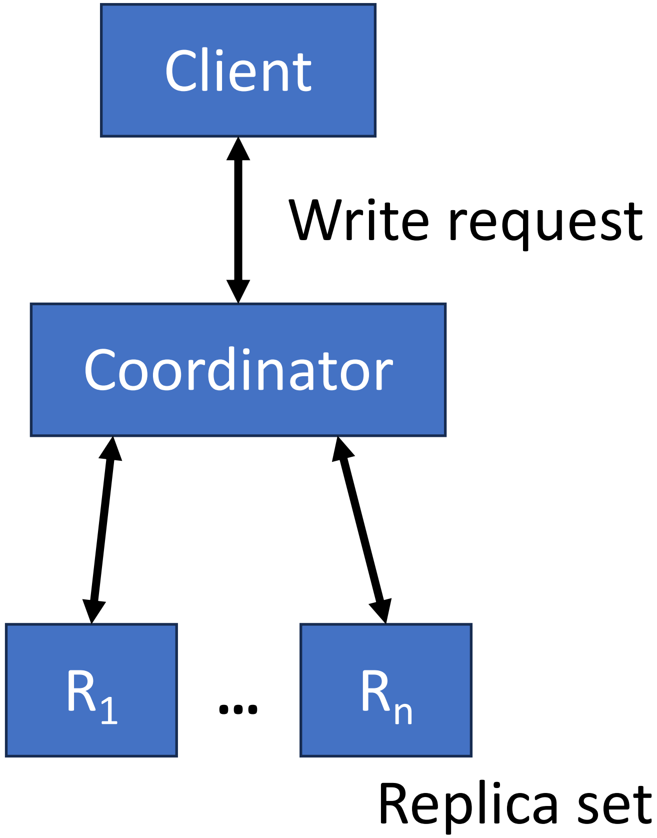
```
CREATE KEYSPACE cluster1 WITH
replication = {'class':
'NetworkTopologyStrategy',
'east': 2, 'west': 3};
```

👉 SimpleStrategy: random partitioner or byte-ordered (ideal for range queries)

EPFL

Cassandra: Writes (1/2)

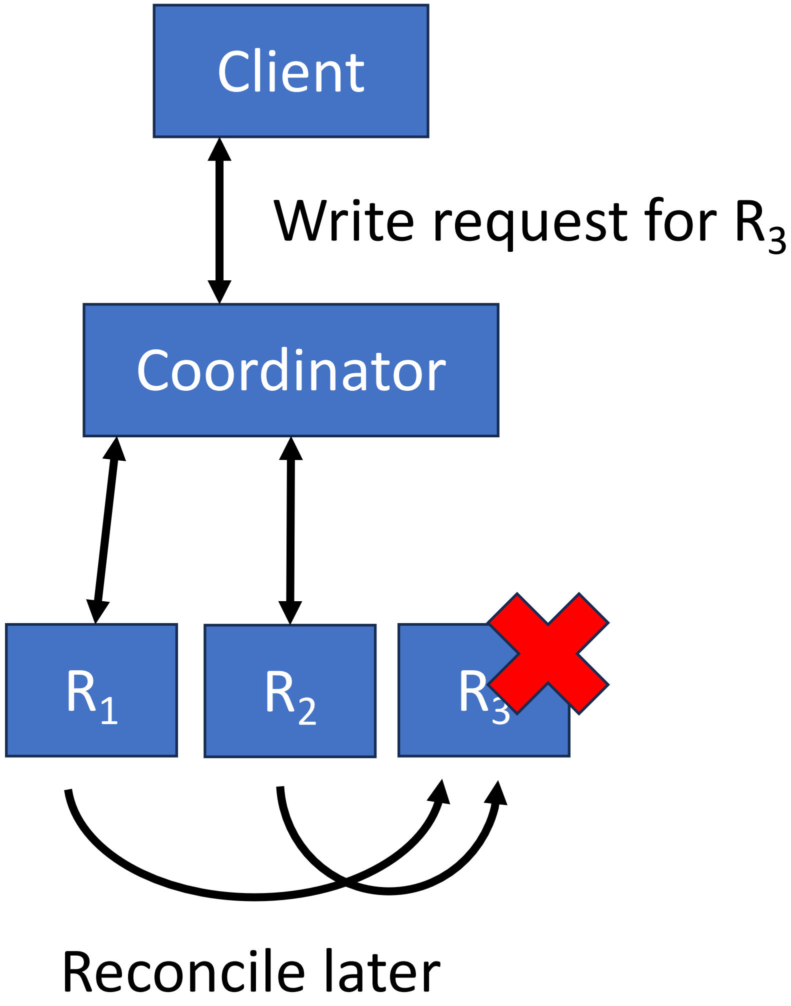
- **Coordinator**: acts as a proxy between clients and replicas
- Writes need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one coordinator node in a Cassandra cluster
  - Coordinator may be per-key, or per-client, or per-query
  - Per-key coordinator ensures writes for that key are serialized
- When *X* replicas respond, coordinator returns an acknowledgement to the client



Cassandra: Writes (2/2)

- Always writable: Hinted Handoff mechanism
  - If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until the down replica comes back up.
  - When all replicas are down, the coordinator (front end) buffers writes (for up to a few hours).

📦 Real-world analogy: accepting parcels of neighbors who are not at home

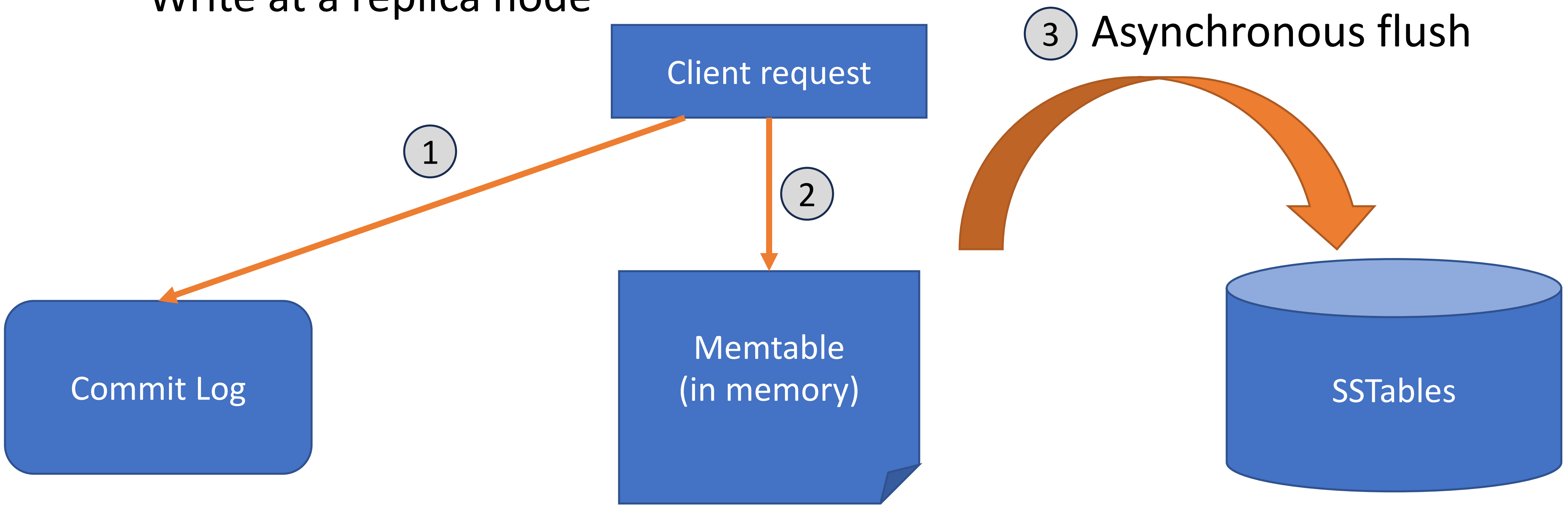


Cassandra: lightweight transactions

- Ensures sequential transaction execution
- Implemented using Paxos consensus
- At the cost of performance

Cassandra: Data structures

Write at a replica node



Transactional log, used for recovery in case of failures

- Write-back cache of data partitions that **can be searched** by key.
- In-memory representation of multiple key-value pairs
- Append-only data structure (fast)

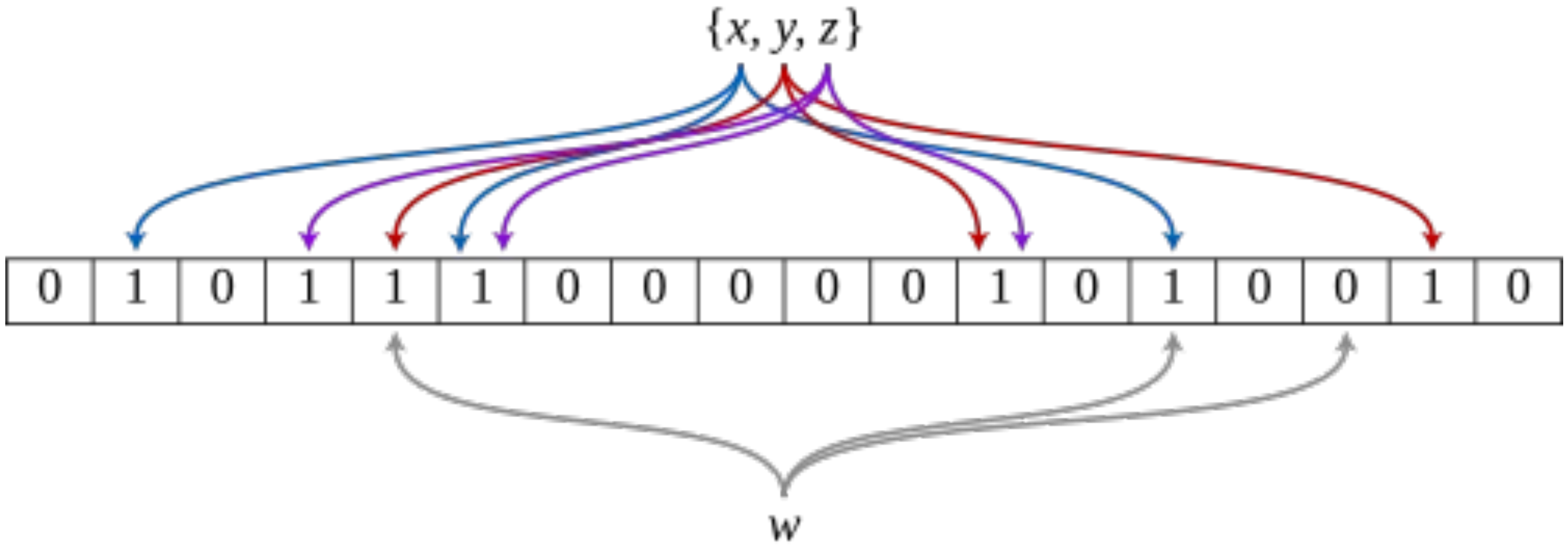
Sorted String Tables (disk):
- Persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings
- Uses Bloom filters

Cassandra: Memtables flushes

- Background thread keeps checking the size of all memtables
- When a new Memtable is created, the previous one marked for flushing
  - Node’s global memory threshold have been reached
  - Commit log is full
- Another thread flushes all the marked Memtables
- Commit log segments of the flushed Memtable are marked for recycling
- A Bloom filter and index are created

Bloom Filters

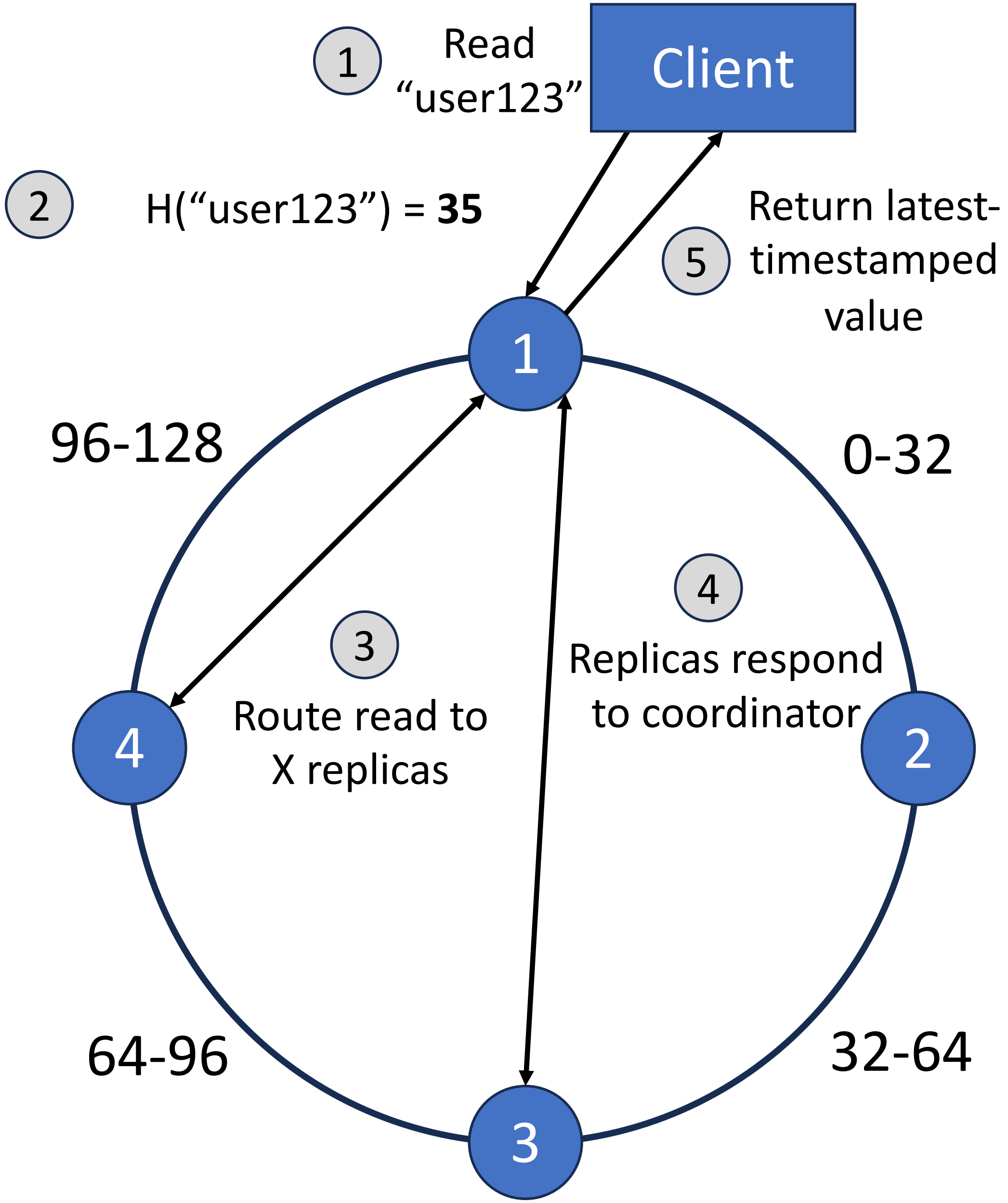
- Compact way of representing a set of items
- Checking for existence (membership) in set is cheap
- Probability of **false positives**: an item not in set may return true as being in set
- Never false negatives



Example FP rate:
- m=4 hash functions
- 100 items in filter
- 3200 bits
- **FP rate = 0.02%**

Cassandra: Reads

- Coordinator can contact *X* replicas
- Checks consistency in the background, initiating a **read repair** if any two values are different
  - This mechanism seeks to eventually bring all replicas up to date
- At a replica: read looks at Memtables first, and then SSTables
  - A row may be split across multiple SSTables



Cassandra: Membership Management (1/2)

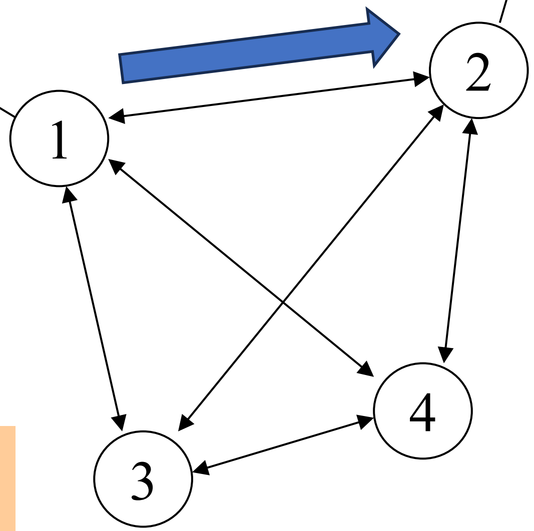
- Any server in the cluster could be the coordinator
- So every server needs to maintain a list of all the other servers that are currently in the cluster: full membership
- Membership needs to be updated automatically as servers join, leave, and fail
- Membership Protocol
  - Efficient anti-entropy gossip-based protocol
  - P2P protocol to discover and share location and state information about other nodes in a Cassandra cluster

Cassandra: Membership Management (2/2)

Cassandra uses gossip-based cluster membership

1	10120	66
2	10103	62
3	10098	63
4	10111	65

Address — Heartbeat Counter — Time (local)



1	10118	64
2	10110	64
3	10090	58
4	10111	65

1	10120	70
2	10110	64
3	10098	63
4	10111	65

Current time: 70 at node 2

(asynchronous clocks)

<u>Protocol:</u>

- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated, as shown
- If any heartbeat older than Δ_{fail}, node is marked as failed

Cassandra: Consistency

- Cassandra has tunable **consistency levels**
- Client chooses a consistency level for each read/write operation

Level	Behavior	Remarks
ANY	Contact any node	Fast; low consistency
ALL	Contact all replicas	Slow; strong consistency
ONE	Contact at least one replica	Faster than all
QUORUM	Contact quorum across replicas in DCs	
LOCAL_QUORUM	Wait for quorum in first DC client contacts	Faster than QUORUM

Quorum-based protocols

In Cassandra, the coordinator must contact a **quorum** of replicas to read or write data

Let:
N = # of replicas
R = # of nodes in read quorum
W = # of nodes in write quorum

Constraints (for strong consistency):

R + W > N

✅ Ensures **most recent write is always read**

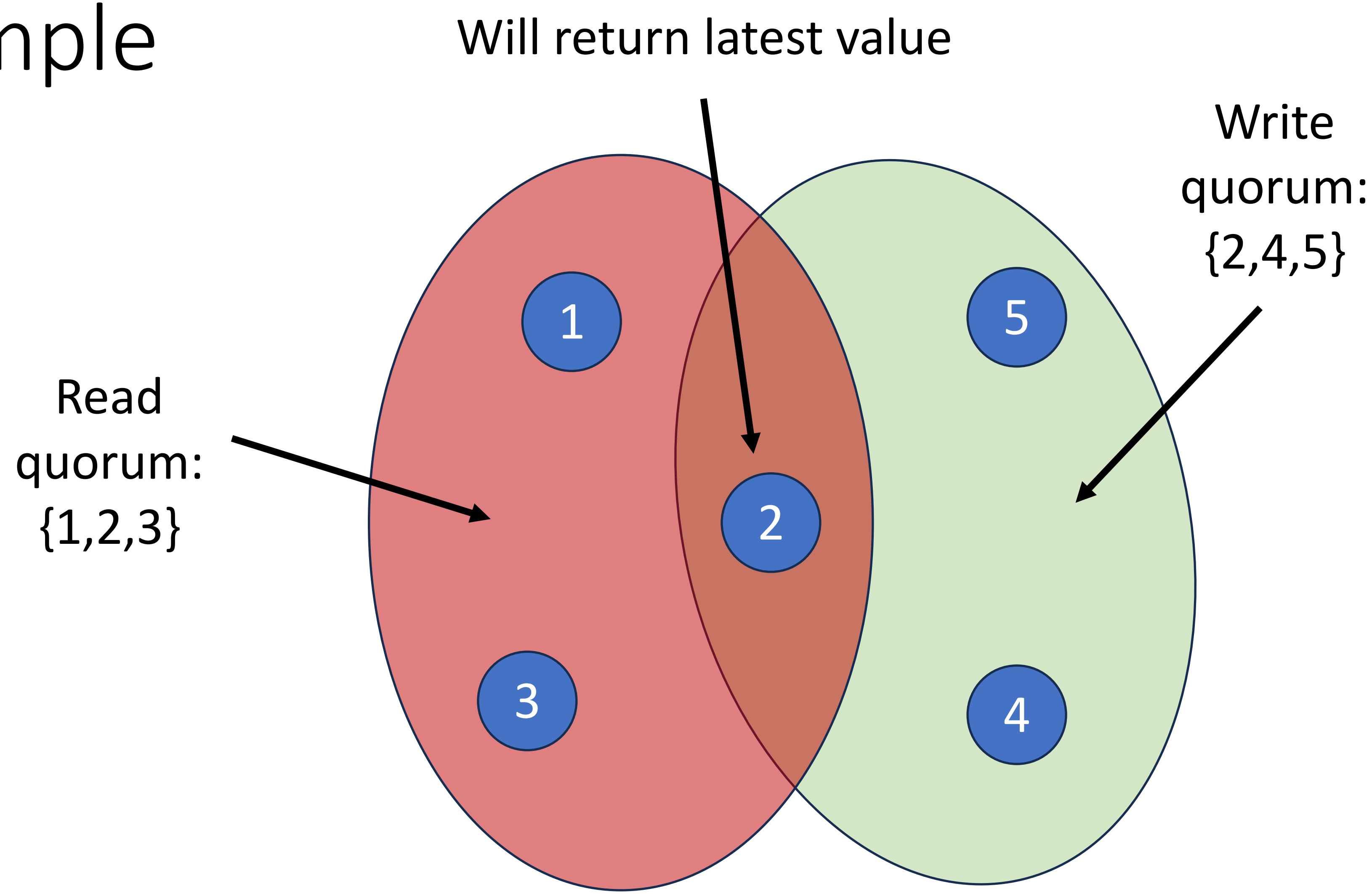
Quorum = Getting agreement from a committee – you don't need everyone, just a majority

Quorums: example

Let:
N = 5
R = 3
W = 3

✅ Strong consistency

💡 Trade-off consistency and availability



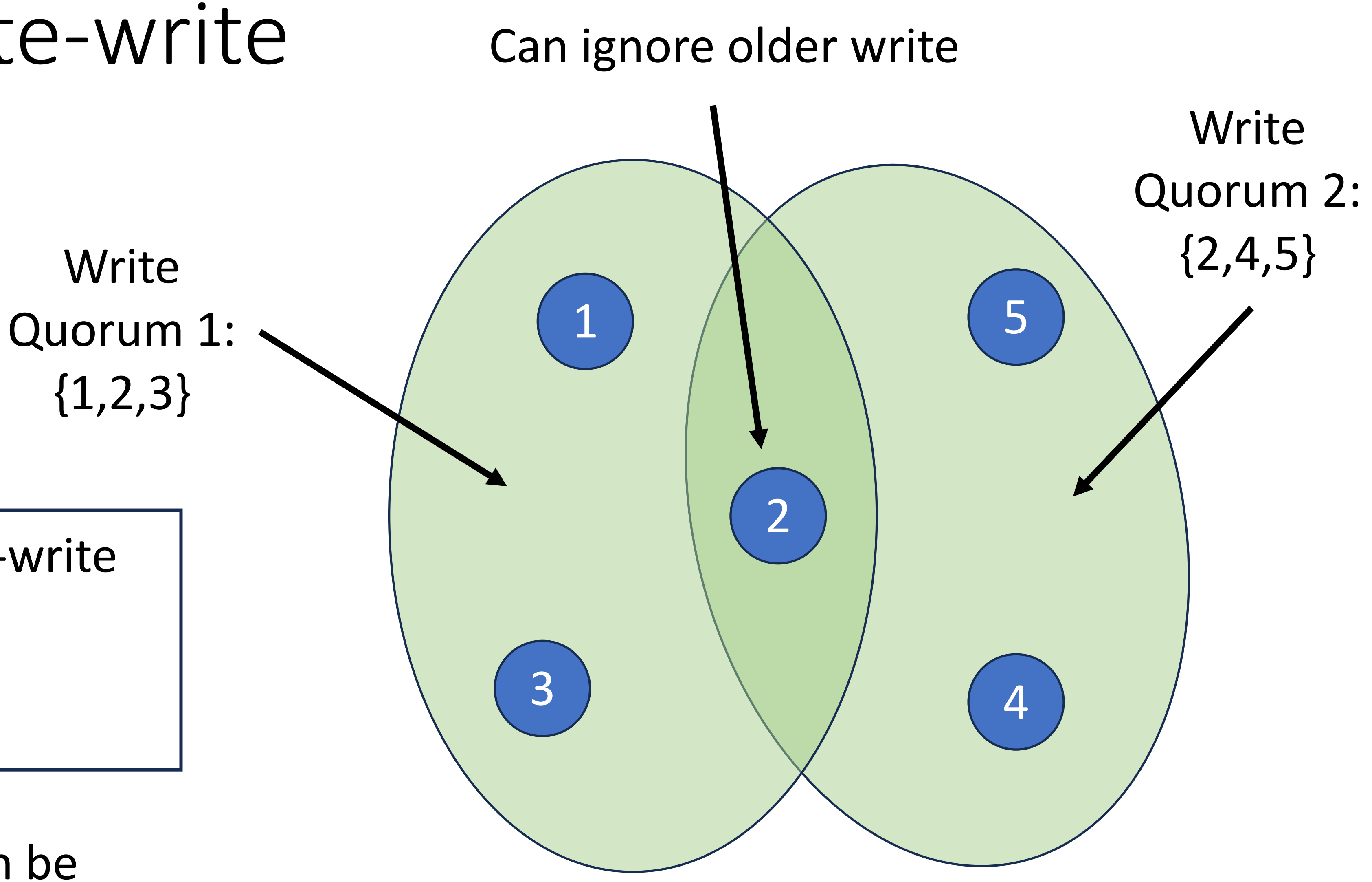
Quorums: write-write conflicts

Let:
N = 5
W = 3

Constraints (to detect write-write conflicts):

W > N / 2

✅ Write-write conflicts can be detected and resolved



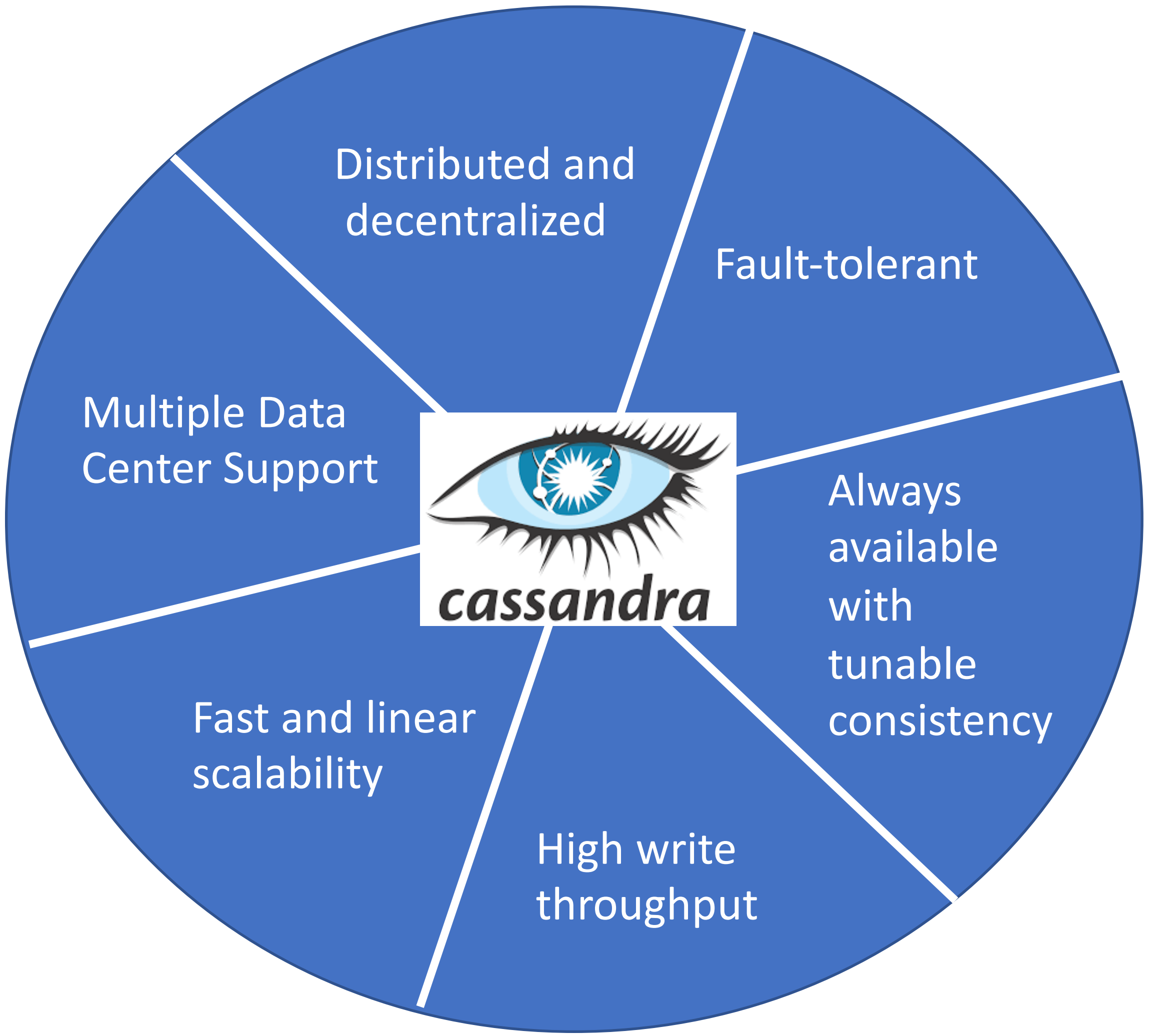
Quorum Trade-offs

- In Cassandra, values of R and W are configurable per query
- No need for strong consistency sometimes → eventual consistency

Goal	Choose:	Why?
Consistency	High R and W	Ensures quorum overlap
Write availability	Lower W	Less nodes need to acknowledge a write
Low read latency	Lower R	Faster reply collection by coordinator

EPFL

Key features of Cassandra



- NoSQL appropriate datastructures for many Big Data applications
- Distributed key-value stored widely used in production
- Uses many algorithms from P2P systems and distributed computing

Key Takeaways

1. Designing distributed systems is all about trade-offs
2. Designing for scale requires rethinking consistency
3. Key-value abstractions power modern web applications

References

- *Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels: Dynamo: amazon's highly available key-value store. SOSP 2007*
- *Avinash Lakshman, Prashant Malik: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. 44(2): 35-40 (2010)*