Distributed Systems

CS425/ECE428

05/01/2020

Today's agenda

- Distributed key-value stores
  - Intro to key-value stores
  - Design requirements and CAP Theorem
  - Case study: Cassandra

- Acknowledgements: Prof. Indy Gupta

Recap

- Cloud provides distributed computing and storage infrastructure as a service.

- Running a distributed job on the cloud cluster can be very complex:
  - Must deal with parallelization, scheduling, fault-tolerance, etc.

- MapReduce is a powerful abstraction to hide this complexity.
  - User programming via easy-to-use API.
  - Distributed computing complexity handled by underlying frameworks and resource managers.

Distributed datastores

- Distributed datastores
  - Service for managing distributed storage.
- Distributed NoSQL key-value stores
  - BigTable by Google
  - HBase open-sourced by Yahoo and used by Hadoop.
  - DynamoDB by Amazon
  - Cassandra by Facebook
  - Voldemort by LinkedIn
  - MongoDB,
  - …
- *Spanner is not a NoSQL datastore. It's more like a distributed relational database.*

The Key-value Abstraction

- (Business) Key → Value
  - (twitter.com) tweet id → information about tweet
  - (amazon.com) item number → information about it
  - (kayak.com) Flight number → information about flight, e.g., availability
  - (yourbank.com) Account number → information about it

The Key-value Abstraction (2)

- It's a dictionary data-structure.
  - Insert, lookup, and delete by key
  - E.g., hash table, binary tree
- But *distributed*.
- Sound familiar?
  - Remember Distributed Hash tables (DHT) in P2P systems (e.g. Chord)?
  - Key-value stores reuse many techniques from DHTs.

Isn't that just a database?

- *Yes, sort of.*
- Relational Database Management Systems (RDBMSs) have been around for ages
  - e.g. MySQL is the most popular among them
- Data stored in structured tables based on a *Schema*
  - Each row (data item) in a table has a primary key that is unique within that table.
- Queried using SQL (Structured Query Language).
  - Supports joins.

Relational Database Example

users table

user_id	name	zipcode	blog_url	blog_id
101	Alice	12345	alice.net	1
422	Charlie	45783	charlie.com	3
555	Bob	99910	bob.blogspot.com	2

Primary keys

Foreign keys

blog table

id	url	last_updated	num_posts
1	alice.net	5/2/14	332
2	bob.blogspot.com	4/2/13	10003
3	charlie.com	6/15/14	7

Example SQL queries

1. SELECT zipcode
   FROM users
   WHERE name = “Bob”

2. SELECT url
   FROM blog
   WHERE id = 3

3. SELECT users.zipcode,
   blog.num_posts
   FROM users JOIN blog
   ON users.blog_url = blog.url

Mismatch with today's workloads

- Data: Large and unstructured
- Lots of random reads and writes
- Sometimes write-heavy
- Foreign keys rarely needed
- Joins infrequent

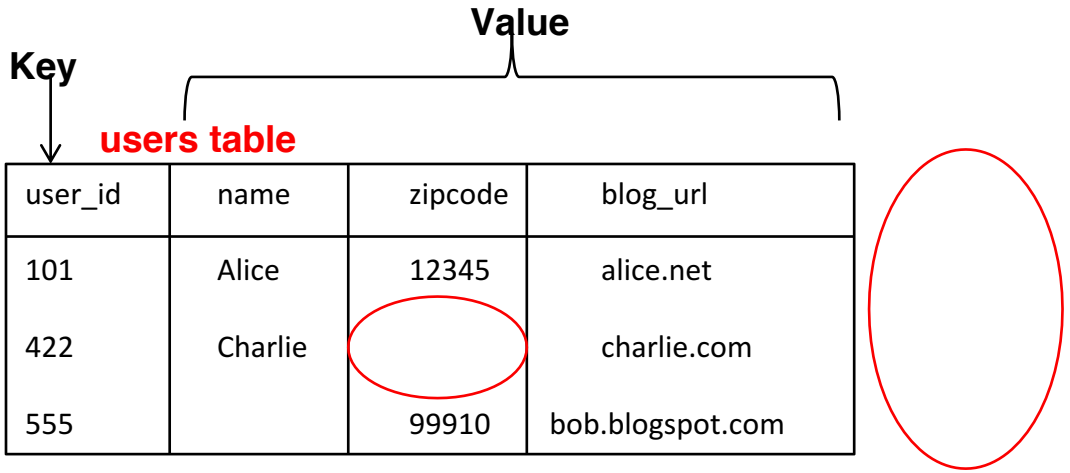
Key-value/NoSQL Data Model

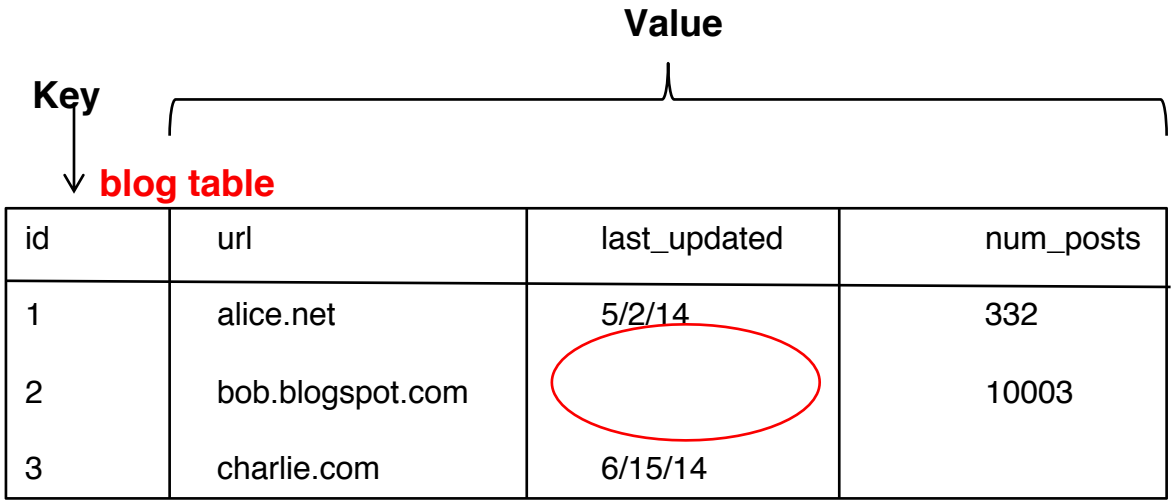
- NoSQL = "Not Only SQL"
- Necessary API operations: get(key) and put(key, value)
  - And some extended operations, e.g., "CQL" in Cassandra key-value store

- Tables
  - Like RDBMS tables, but …
  - May be unstructured: May not have schemas
    - Some columns may be missing from some rows
  - Don't always support joins or have foreign keys
  - Can have index tables, just like RDBMSs

Key-value/NoSQL Data Model

- Unstructured
- No schema imposed
- Columns Missing from some Rows
- No foreign keys, joins may not be supported





How to design a distributed key-value datastore?

Design Requirements

- High performance, low cost, and scalability.
  - Speed (high throughput and low latency for read/write)
  - Low TCO (total cost of operation)
  - Fewer system administrators
  - Incremental scalability
    - Scale out: add more machines.
    - Scale up: upgrade to powerful machines.
    - *Cheaper to scale out than to scale up.*

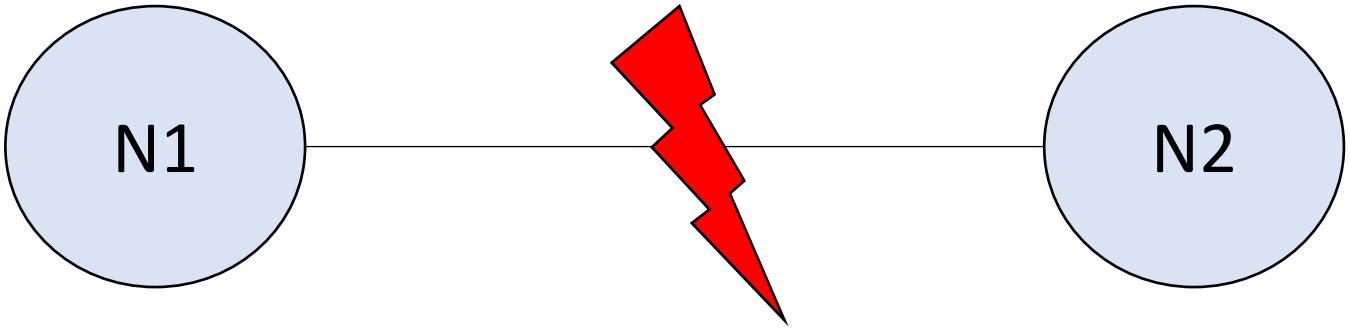
Design Requirements

- High performance, low cost, and scalability.
- Avoid single-point of failure
  - Replication across multiple nodes.
- Consistency: reads return latest written value by any client (all nodes see same data at any time).
  - *Different from the C of ACID properties for transaction semantics!*
- Availability: every request received by a non-failing node in the system must result in a response (quickly).
  - Follows from requirement for high performance.
- Partition-tolerance: the system continues to work in spite of network partitions.

CAP Theorem

- **C**onsistency: reads return latest written value by any client (all nodes see same data at any time).
- **A**vailability: every request received by a non-failing node in the system must result in a response (quickly).
- **P**artition-tolerance: the system continues to work in spite of network partitions.
- **In a distributed system you can only guarantee at most 2 out of the above 3 properties.**
  - Proposed by Eric Brewer (UC Berkeley)
  - Subsequently proved by Gilbert and Lynch (NUS and MIT)

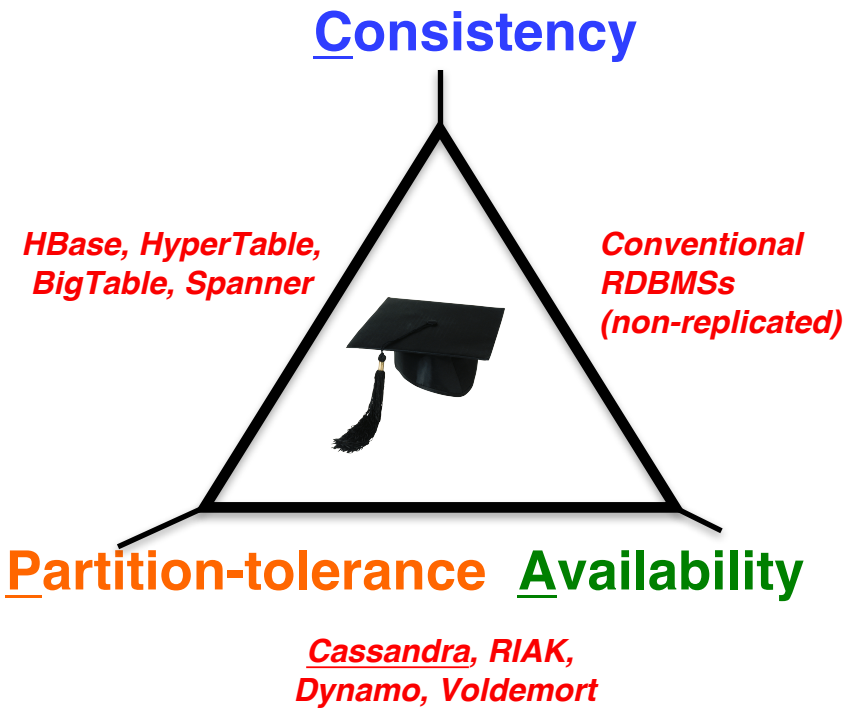
CAP Theorem



- Data replicated across both N1 and N2.
- If network is partitioned, N1 can no longer talk to N2.
- Consistency + availability require N1 and N2 must talk.
  - no partition-tolerance.
- Partition-tolerance + consistency:
  - only respond to requests received at N1 (no availability).
- Partition-tolerance + availability:
  - write at N1 will not be captured by a read at N2 (no consistency).

CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve at most two of C, A, and P.
- When partition-tolerance is important, you have to choose between consistency and availability



Case Study: Cassandra

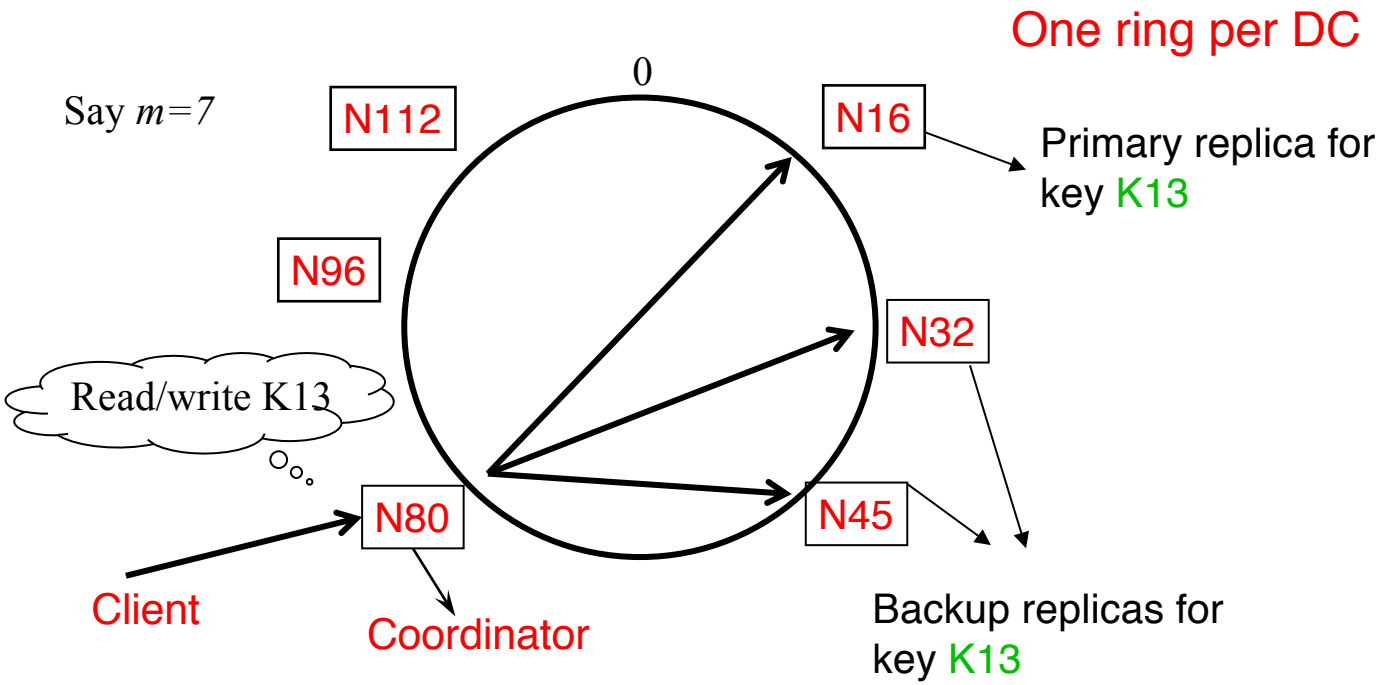
Cassandra

- A distributed key-value store.
- Intended to run in a datacenter (and also across DCs).
- Originally designed at Facebook.
- Open-sourced later, today an Apache project.
- Some of the companies that use Cassandra in their production clusters.
  - IBM, Adobe, HP, eBay, Ericsson, Symantec
  - Twitter, Spotify
  - PBS Kids
  - Netflix: uses Cassandra to keep track of your current position in the video you're watching

Data Partitioning: Key to Server Mapping

- How do you decide which server(s) a key-value resides on?

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



Partitioner

- Component responsible for key to server mapping (hash function).

- Two types:
  - *Chord-like hash partitioning*
    - *Murmer3Partitioner* (default): uses *murmer3* hash function.
    - *RandomPartitioner*: uses MD5 hash function.
  - *ByteOrderedPartitioner*: Assigns ranges of keys to servers.
    - Easier for *range queries* (e.g., get me all twitter users starting with [a-b])

- Determines the primary replica for a key.

Replication Policies

Two options for replication strategy:

1. SimpleStrategy:
    - First replica placed based on the partitioner.
    - Remaining replicas clockwise in relation to the primary replica.
2. NetworkTopologyStrategy: for multi-DC deployments
    - Two or three replicas per DC.
    - Per DC
        - First replica placed according to Partitioner.
        - Then go clockwise around ring until you hit a different rack.

Writes

- Need to be lock-free and fast (no reads or disk seeks).
- Client sends write to one coordinator node in Cassandra cluster.
  - Coordinator may be per-key, or per-client, or per-query.
- Coordinator uses Partitioner to send query to all replica nodes responsible for key.
- When X replicas respond, coordinator returns an acknowledgement to the client
  - X = any one, majority, all….(consistency spectrum)
  - More details later!

Writes: Hinted Handoff

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

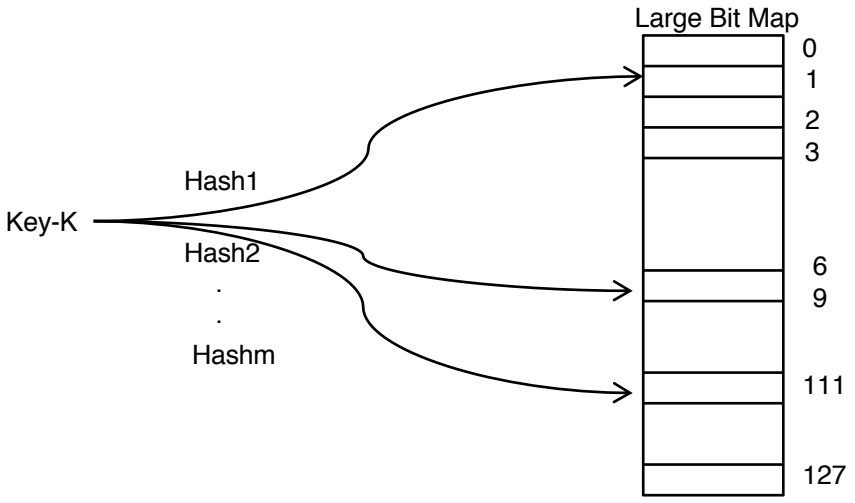
Writes at a replica node

On receiving a write

1. Log it in disk commit log (for failure recovery)
2. Make changes to appropriate memtables
    - **Memtable** = In-memory representation of multiple key-value pairs
    - Cache that can be searched by key
    - Write-back cache as opposed to write-through
3. Later, when memtable is full or old, flush to disk
    - Data File: An **SSTable** (Sorted String Table) – list of key-value pairs, sorted by key
    - Index file: An SSTable of (key, position in data sstable) pairs
    - And a Bloom filter (for efficient search) – next slide.

Bloom Filter

- Compact way of representing a set of items.
- Checking for existence in set is cheap.
- Some probability of false positives: an item not in set may check true as being in set.
- Never false negatives.



On insert, set all hashed bits.

On check-if-present,
return true if all hashed bits set.

- False positives

False positive rate low

- m=4 hash functions
- 100 items
- 3200 bits
- FP rate = 0.02%

Compaction

- Data updates accumulate over time and over multiple SSTables.
- Need to be compacted.
- The process of compaction merges SSTables, i.e., by merging updates for a key.
- Run periodically and locally at each server.

Deletes

Delete: don't delete item right away

- Write a **tombstone** for the key.
- Eventually, when compaction encounters tombstone it will delete item

Reads

- Coordinator contacts X replicas (e.g., in same rack)
  - Coordinator sends read to replicas that have responded quickest in past.
  - When X replicas respond, coordinator returns the latest-timestamped value from among those X.
  - X = based on consistency spectrum (more later).
- Coordinator also fetches value from other 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 => reads need to touch multiple SSTables => reads slower than writes (but still fast).

Cross-DC coordination

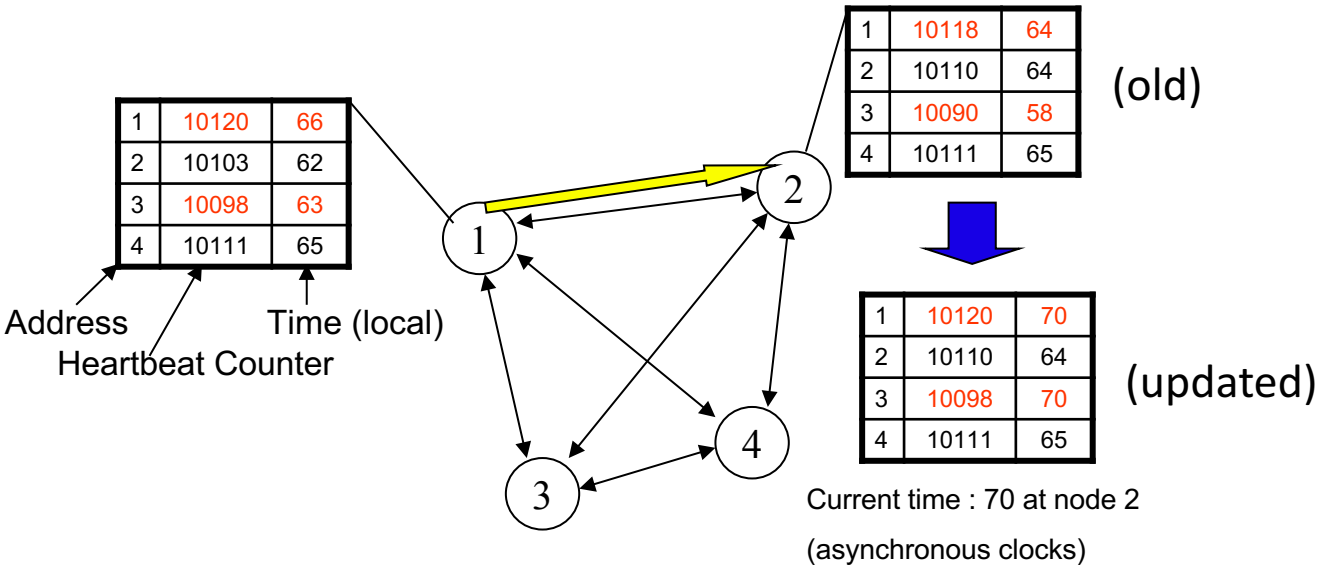
- Replicas may span multiple datacenters.
- Per-DC coordinator elected to coordinate with other DCs.
- Election done via Zookeeper which runs a Bully algorithm variant.

Membership

- Any server in cluster could be the leader.
- So every server needs to maintain a list of all the other servers that are currently in the cluster.
- List needs to be updated automatically as servers join, leave, and fail.

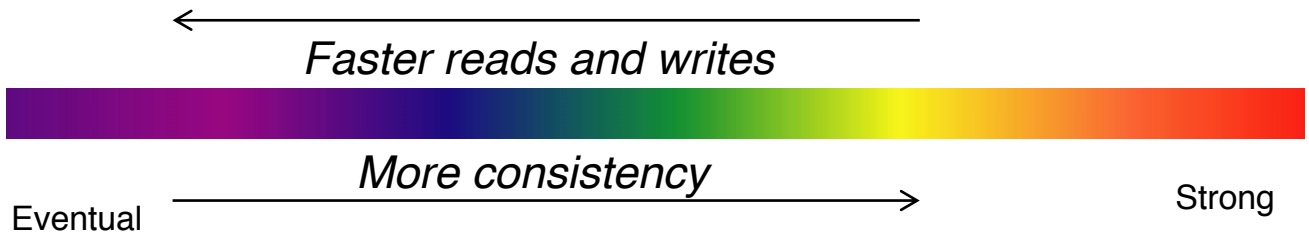
Cluster Membership

Cassandra uses gossip-based cluster membership



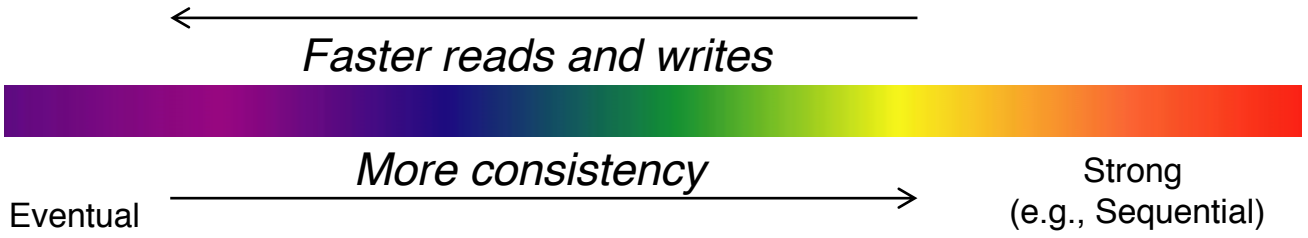
- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated, as shown
- If any heartbeat older than Tfail, node is marked as failed

Consistency Spectrum



Eventual Consistency

- Cassandra offers Eventual Consistency
  - If writes to a key stop, all replicas of key will converge.
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems



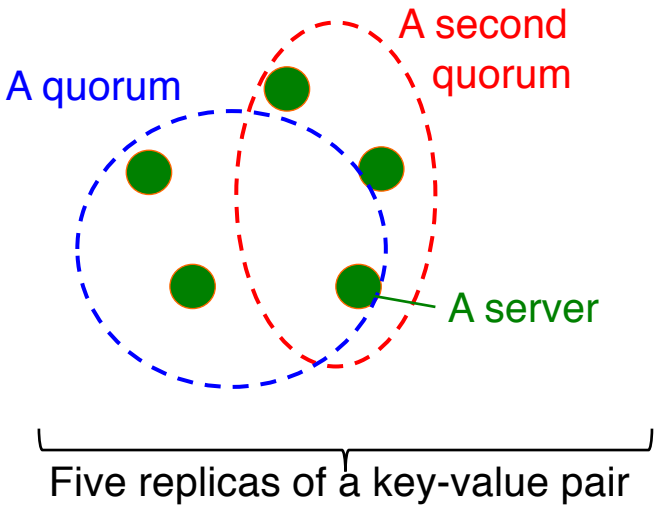
Consistency levels: value of X

- Cassandra has consistency levels.
- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator caches write and replies quickly to client
  - ALL: all replicas
    - Ensures strong consistency, but slowest
  - ONE: at least one replica
    - Faster than ALL, but cannot tolerate a failure
  - QUORUM: quorum across all replicas in all datacenters (DCs)

Quorums?

In a nutshell:

- Quorum = (typically) majority
- Any two quorums intersect
  - Client 1 does a write in red quorum
  - Then client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums faster than ALL, but still ensure strong consistency
- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.



Read Quorums

- Reads
  - Client specifies value of R (≤ N = total number of replicas of that key).
  - R = read consistency level.
  - Coordinator waits for R replicas to respond before sending result to client.
  - In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed.

Write Quorums

- Client specifies W (≤ N)
- W = write consistency level.
- Client writes new value to W replicas and returns.
- Two flavors:
  - Coordinator blocks until quorum is reached (default).
  - Asynchronous: Just write and return.
    - Source of inconsistency.

Quorums in Detail (Contd.)

- R = read replica count, W = write replica count
- Necessary conditions for consistency:
  1. W+R > N
     - Write and read intersect at a replica. Read returns latest write.
  2. W > N/2
     - Two conflicting writes on a data item don't occur at the same time.
- Select values based on application
  - (W=N, R=1):
    - great for read-heavy workloads
  - (W=1, R=N):
    - great for write-heavy workloads with no conflicting writes.
  - (W=N/2+1, R=N/2+1):
    - great for write-heavy workloads with potential for write conflicts.
  - (W=1, R=1):
    - very few writes and reads / high availability requirement.

Cassandra Consistency Levels

- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator may cache write and reply quickly to client
  - ALL: all replicas
    - Slowest, but ensures strong consistency
  - ONE: at least one replica
    - Faster than ALL, and ensures durability without failures
  - QUORUM: quorum across all replicas in all datacenters (DCs)
    - Global consistency, but still fast
  - LOCAL_QUORUM: quorum in coordinator's DC
    - Faster: only waits for quorum in first DC client contacts
  - EACH_QUORUM: quorum in every DC
    - Lets each DC do its own quorum: supports hierarchical replies

Eventual Consistency

- Sources of inconsistency:
  - Quorum condition not satisfied R + W < N.
    - R and W are chosen as such.
    - when write returns before W replicas respond.
      - Sloppy quorum: when value stored elsewhere if intended replica is down, and later moved to the replica when it is up again.
  - When local quorum is chosen instead of global quorum.
- Hinted-handoff and read repair help in achieving *eventual consistency*.
  - If all writes stop (to a key), then all its values (replicas) will converge eventually.
  - May still return stale values to clients (e.g., if many back-to-back writes).
  - But works well when there a few periods of low writes – system converges quickly.

Cassandra Vs. RDBMS

- MySQL is one of the most popular (and has been for a while)
- On > 50 GB data
- MySQL
  - Writes 300 ms avg
  - Reads 350 ms avg
- Cassandra
  - Writes 0.12 ms avg
  - Reads 15 ms avg
- Orders of magnitude faster.

Other similar NoSQL stores

- Amazon's DynamoDB
  - Cassandra's data partitioning, replication, and eventual consistency strategies inspired from Dynamo.
  - Uses sloppy quorum as the default mechanism for eventual consistency with availability.
  - Uses vector clocks to capture causality between different versions of an object.
  - Dynamo: Amazon's Highly Available Key-value Store, SOSP'2007.
- LinkedIn's Voldemort
  - Inspired from DynamoDB.
- …..

Summary

- CAP theorem: cannot only achieve 2 out of 3 among consistency, availability, and partition-tolerance.

- Partition-tolerance is required in distributed datastores.
  - Choose between consistency and availability.

- Many modern distributed NoSQL key-value stores (e.g. Cassandra) choose availability, providing only eventual consistency.