CLOUD COMPUTING CONCEPTS

with Indranil Gupta (Indy)

KEY-VALUE STORES NoSQL

Lecture A

WHY KEY-VALUE/NoSQL?

The Key-Value Abstraction

New Generation of Storage

- (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

Very significant in Clouds for efficient Storage and Retrieval

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 (Like Cluster of Servers)

Isn't that just a database?

- Yes, sort of
- Relational Database Management Systems (RDBMSs) have been around for ages
- MySQL is the most popular among them
- Data stored in tables
- Schema-based, i.e., structured tables
- 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 (even simultaneous from multiple clients)
- Sometimes write-heavy (lot more writes than reads unlike RDBMS which is read heavy)
- Foreign keys rarely needed
- Joins infrequent (inspiring us to do away with these to come up with a faster approach)

E.g. Web pages, xml data, text data all of which cannot be divided into columns and fields
Hard to come up with schema that accommodates the data e.g. Web

Needs of Today's Workloads

- Speed
- Avoid Single Point of Failure (SPOF)
- Low TCO (Total cost of operation)
- Fewer system administrators
- Incremental scalability
- Scale out, not up
  - What?

(All important for Clouds)

Scale out, not Scale up

- **Scale up** = grow your cluster capacity by replacing with more powerful machines
  (More CPU, Memory, Disk, Bandwidth...)
  - Traditional approach
  - Not cost-effective, as you're buying above the sweet spot on the price curve
  - And you need to replace machines often

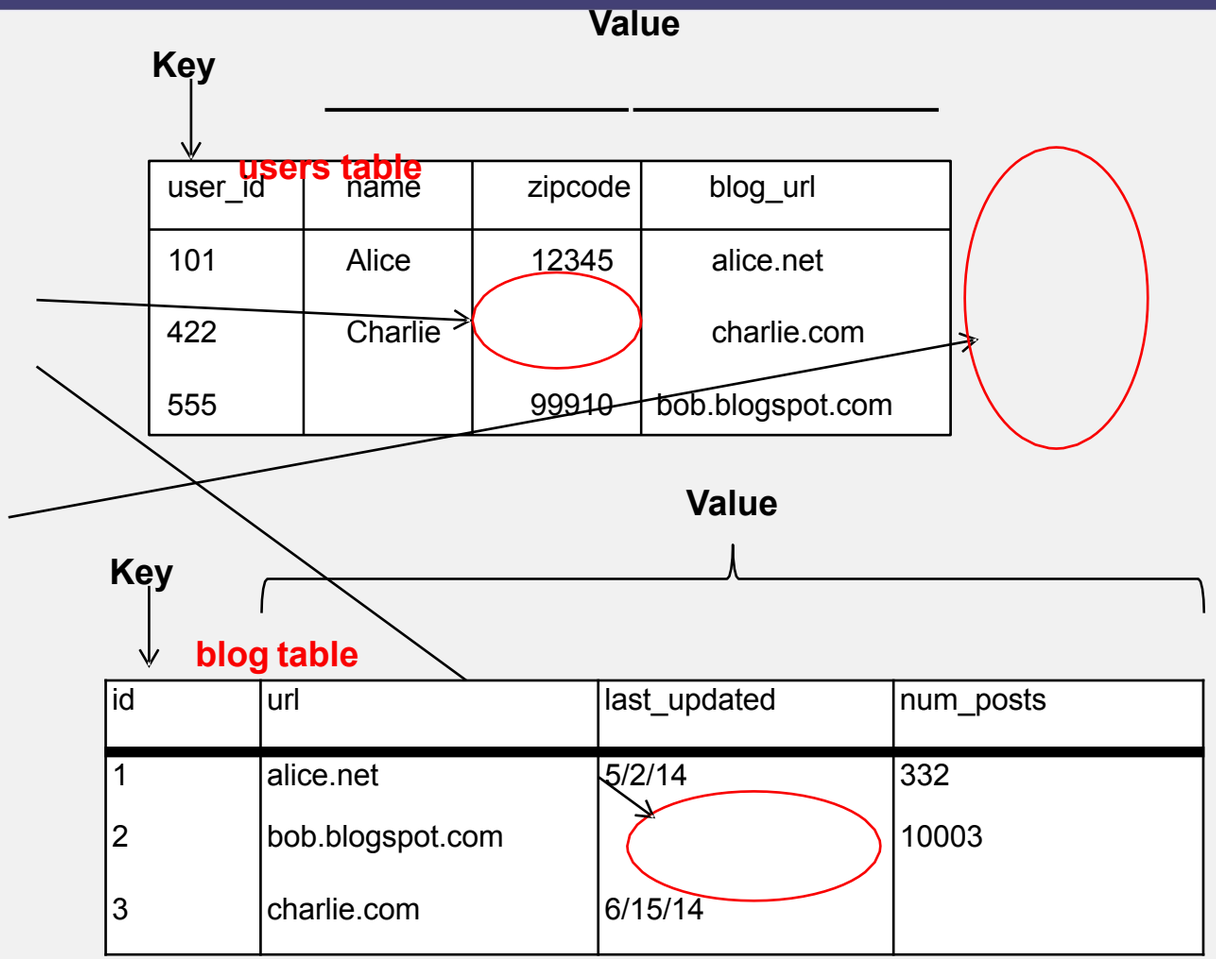
- **Scale out** = incrementally grow your cluster capacity by adding more COTS machines (Components Off the Shelf)
  - Cheaper
  - Over a long duration, phase in a few newer (faster) machines as you phase out a few older machines
  - Used by most companies who run datacenters and clouds today

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
  - "Column families" in Cassandra, "Table" in HBase, "Collection" in MongoDB
  - 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



Column-Oriented Storage

NoSQL systems often use column-oriented storage

- RDBMSs store an entire row together (on disk or at a server)
- NoSQL systems typically store a column together (or a group of columns).
  - Entries within a column are indexed and easy to locate, given a key (and vice-versa)

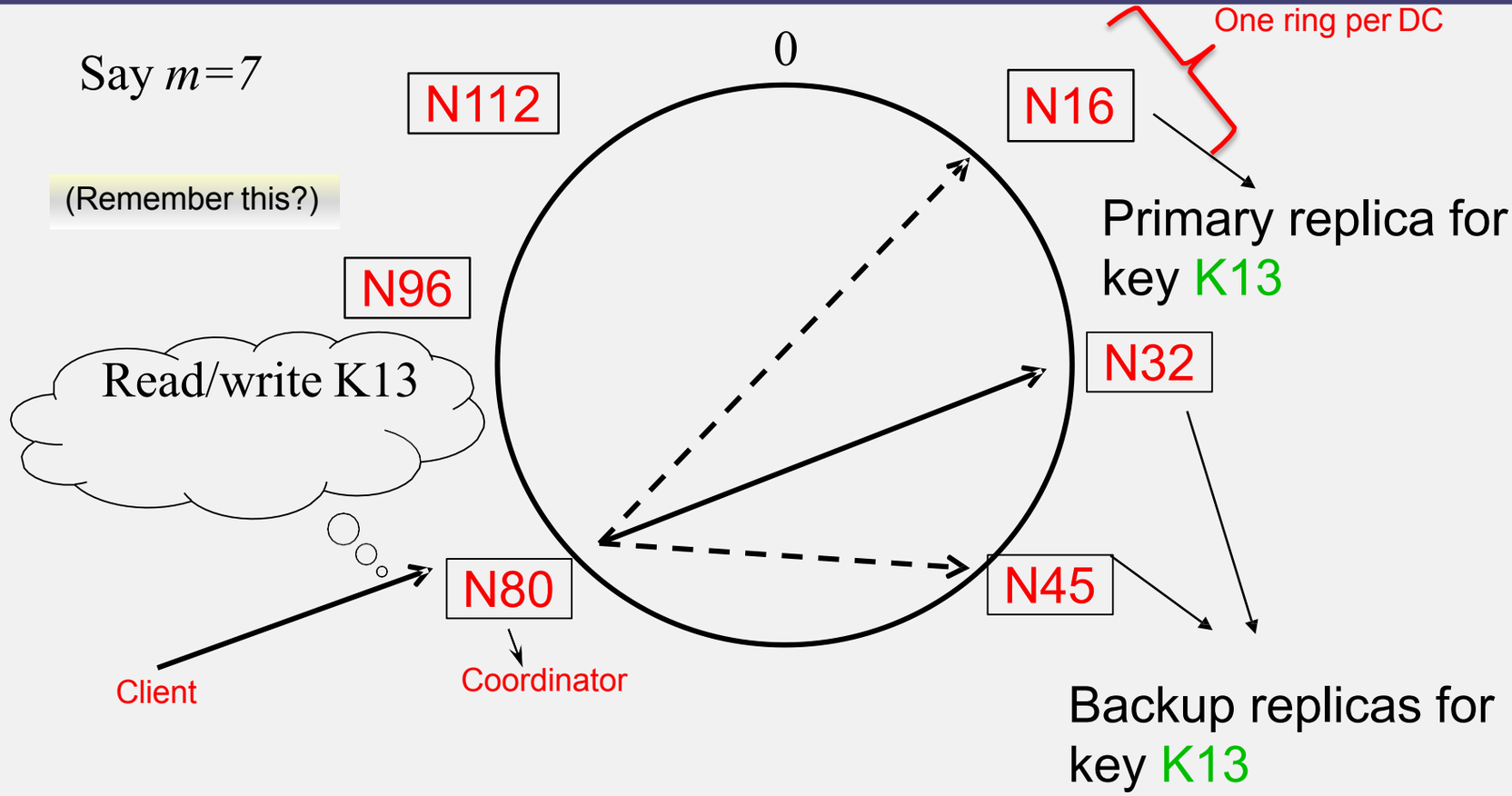
- Why useful?
  - Range searches within a column are fast since you don't need to fetch the entire database
  - E.g., get me all the blog_ids from the blog table that were updated within the past month
    - Search in the the last_updated column, fetch corresponding blog_id column
    - Don't need to fetch the other columns

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

Let's go Inside Cassandra: Key -> Server Mapping

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



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

Key€server mapping is the "Partitioner"

Cassandra places Servers in a virtual ring

Ring considers 2^m points

If m=7, means 128 points. A server is placed at each point

Keys gets stored on servers which are successors of that key value

E.g. Key ID =13, next server point is N=16 so Key gets placed there

Even replicas can be placed accordingly in succeeding servers (but not necessarily always)

Client requests the coordinator which could be one per DC or one per many DCs or created for each query

Different DC: Different Rings Different Servers.

Each server should know about various keys placed in different servers on the ring

Data Placement Strategies

- Replication Strategy: two options:
  1. *SimpleStrategy*
  2. *NetworkTopologyStrategy*

1. SimpleStrategy: uses the Partitioner, of which there are two kinds
   1. *RandomPartitioner*: Chord-like hash partitioning (Like Key ID determines either the successor or predecessor will store the Key)
   2. *ByteOrderedPartitioner*: Assigns ranges of keys to servers.
      - Easier for *range queries* (e.g., get me all twitter users starting with [a-b])

2. NetworkTopologyStrategy: for multi-DC deployments
   - Two replicas per DC
   - Three replicas per DC
   - Per DC
     - First replica placed according to Partitioner
     - Then go clockwise around ring until you hit a different rack

Snitches

- Maps: IPs to racks and DCs. Configured in cassandra.yaml config file

- Some options:
  - SimpleSnitch: Unaware of Topology (Rack-unaware)
  (Useful when you have only one or few VMs; you don't care much then)
  - RackInferring: Assumes topology of network by octet of server's IP address
    - 101.201.301.401 = x.<DC octet>.<rack octet>.<node octet>
  - PropertyFileSnitch: uses a config file
  - EC2Snitch: uses EC2 (Useful as you are aware of the Availability Zone and Region a prior in AWS)
    - EC2 Region = DC
    - Availability zone = rack
- Other snitch options available

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, per-client, or per-query
  - Per-key Coordinator ensures writes for the key are serialized

Request sent by Client to Coordinator to Replica
Being Write heavy need to minimize Reads
Keys important to be always written
Makes the system Fault Tolerant

- Coordinator uses Partitioner to send query to all replica nodes responsible for key

(This coordinator is different from previous one and only serves to coordinate with other DCs)

- When X replicas respond, coordinator returns an acknowledgment to the client
  - X? We'll see later what should be the value of X

Writes (2)

- Always writable Keys: 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).

- One ring per datacenter
  - Per-DC coordinator elected to coordinate with other DCs
  - Election done via Zookeeper, which runs a Paxos (consensus) variant

Writes at a replica node

On receiving a write, the replica server

1. Logs it in disk commit log (for its own failure recovery)
2. Make changes to appropriate memtables
    - **Memtable** = In-memory representation of multiple key-value pairs
    - Cache that can be searched by key (If key found read else key-value pair appended in the list)
    - Write-back cache as opposed to write-through

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 (Slow in search as it comprises of both key and the value for different entries)
    - Index file: An SSTable of (key, position in data sstable) pairs (For quicker look at key position)
- And a Bloom filter (for efficient search) – See at your end

Compaction

Problem of large number of SStables in a particular Cassandra Server and a given key is present in many SStables

Data updates accumulate over time and SStables and logs need to be compacted

- The process of compaction merges SSTables. This means the compaction process merges all the SStables and retain only the latest updated value

This is called merging the updates for a key

- Run periodically and locally at each server

Deletes

Delete: don’t delete item right away

- Add a **tombstone** to the log (Marker which says when you encounter this key delete it)
- Eventually, when compaction encounters tombstone it will delete item

Reads

Read: Similar to writes, except

- Coordinator can contact 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? We'll see 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
- A row may be split across multiple SSTables => reads need to touch multiple SSTables => reads slower than writes (but still fast)

Same as Cache Coherence Problem)

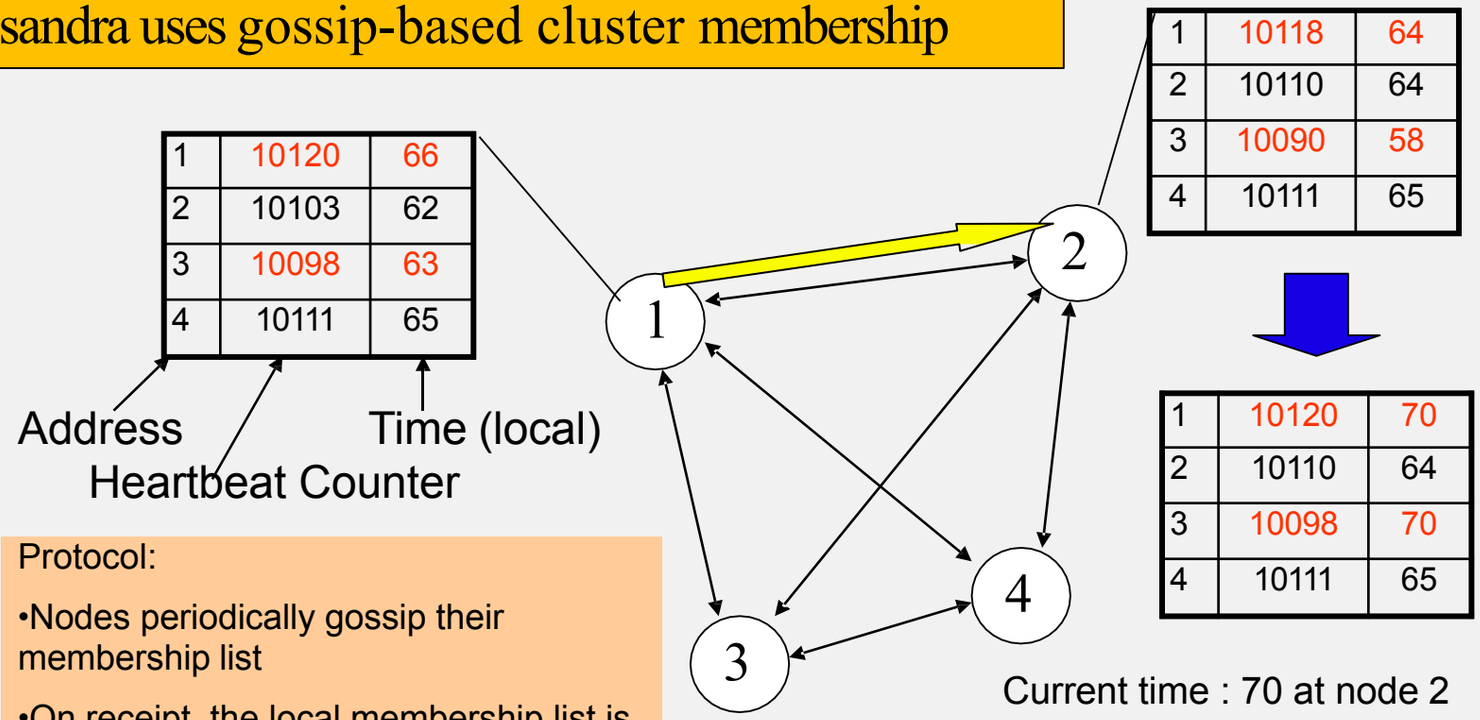
The case with no Compaction or not yet run Compaction

Membership

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

Cluster Membership – Gossip-Style

Cassandra uses gossip-based cluster membership



Protocol:

- 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

Current time : 70 at node 2

(asynchronous clocks)

(Remember this?)

Suspicion Mechanisms in Cassandra

- Suspicion mechanisms to adaptively set the timeout based on underlying network and failure behavior
- Accrual detector: Failure detector outputs a value (PHI) representing suspicion
- Apps set an appropriate threshold
- PHI calculation for a member

- Inter-arrival times for gossip messages
- PHI(t) =
  - log(CDF or Probability(t_now – t_last))/log 10
- PHI basically determines the detection timeout, but takes into account historical inter-arrival time variations for gossiped heartbeats
- In practice, PHI = 5 => 10-15 sec detection time

This is Per Server Based
Thus PHI is set different for different server with different values due to their different speeds or response times

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
- What's the catch? What did we lose? (Next Discussion)

CAP Theorem

X Specified by the client with every Read or Write indicating the required Number of Replicas

- Proposed by Eric Brewer (Berkeley)
- Subsequently proved by Gilbert and Lynch (NUS and MIT)
- In a distributed system you can satisfy at most 2 out of the 3 guarantees:
  1. **Consistency**: all nodes see same data at any time, or reads return latest written value by any client
  2. **Availability**: the system allows operations (Reading/Writing Keys and Values) all the time, and operations return quickly
  3. **Partition-tolerance**: the system continues to work in spite of network partitions

Why is Availability Important?

- Availability = Reads/writes complete reliably and quickly.
- Measurements have shown that a 500 ms increase in latency for operations at Amazon.com or at Google.com can cause a 20% drop in revenue.
- At Amazon, each added millisecond of latency implies a $6M yearly loss.
- SLAs (Service Level Agreements) written by providers predominantly deal with latencies faced by clients.

Why is Consistency Important?

- Consistency = all nodes see same data at any time, or reads return latest written value by any client.
- When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.
- When thousands of customers are looking to book a flight, all updates from any client (e.g., book a flight) should be accessible by other clients.

Why is Partition-Tolerance Important?

- Partitions can happen across datacenters when the Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working
    (OR May be government censorship)

- Partitions can also occur within a datacenter, e.g., a rack switch outage

- Still desire system to continue functioning normally under this scenario

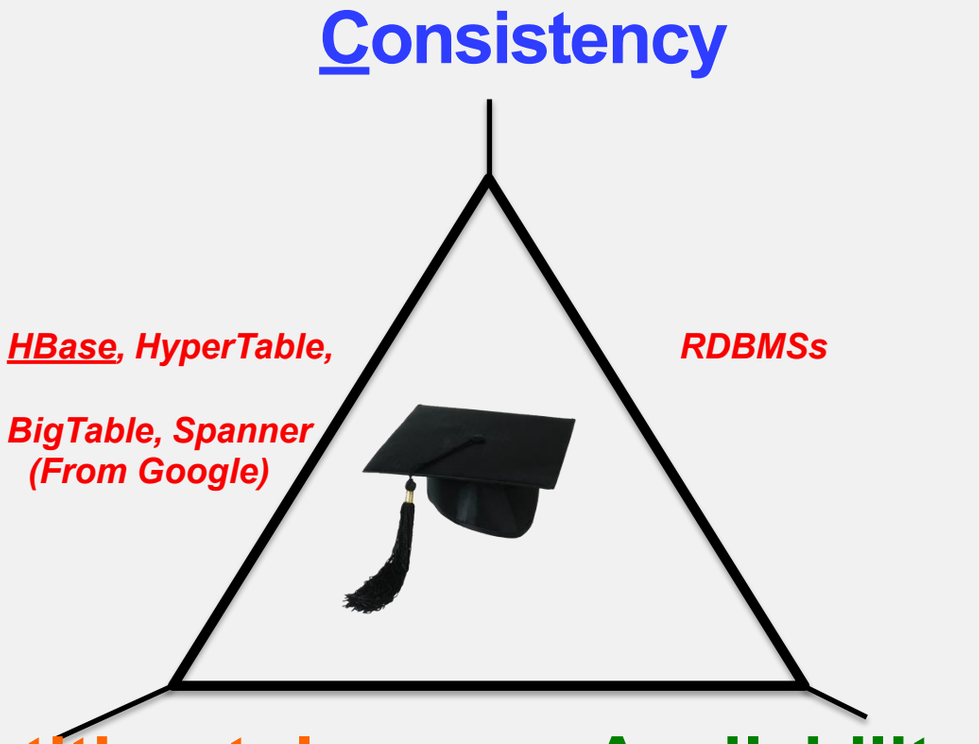
CAP Theorem Fallout

- Since partition-tolerance is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability
- Cassandra
  - Eventual (weak) consistency, availability, partition-tolerance
- Traditional RDBMSs
  - Strong consistency over availability under a partition

Remember, replicas becoming consistent after compaction. Only the latest copy retained

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



Eventual Consistency

- If all writes stop (to a key), then all its values (replicas) will converge eventuallyusing Cassandra.
- If writes continue, then system always tries to keep converging.
  - Moving "wave" of updated values lagging behind the latest values sent by clients, but always trying to catch up.
- 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.

RDBMS vs. Key-value stores

- While RDBMS provide ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Key-value stores like Cassandra provide BASE
  - Basically Available Soft-state Eventual consistency (Soft State means in Memory states like SStables)
  - Prefers availability over consistency

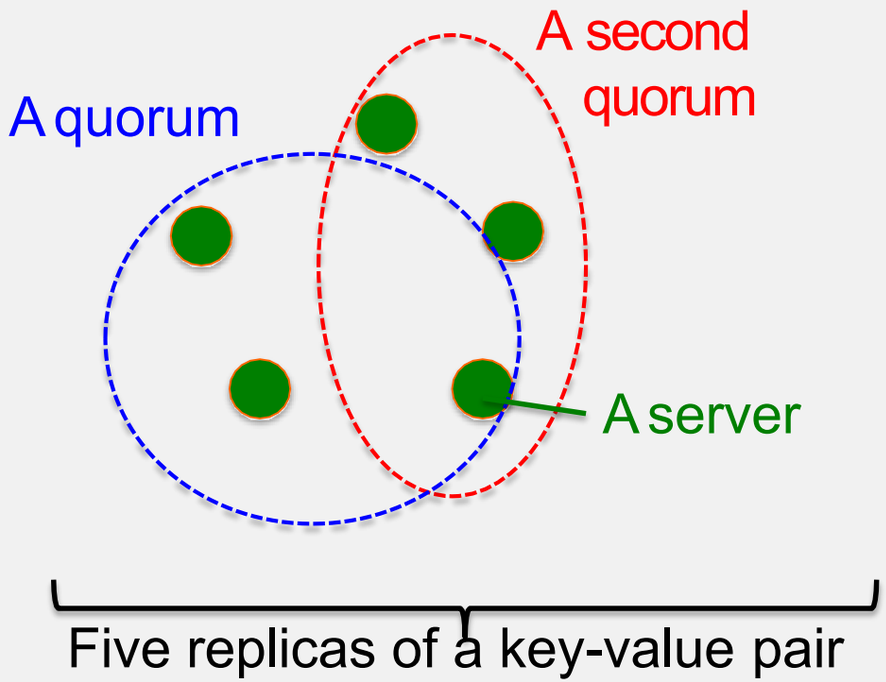
Back to Cassandra: Mystery of X

- Cassandra has consistency levels
- Client is allowed to choose a consistency level for each operation be it Read or Write
  - ANY: any server can store the Write (may not be replica)
    - Fastest: coordinator caches write and replies quickly to client (Even if all the replicas are down)
  - ALL: all replicas need to acknowledge the coordinator for the Write which only afterwards acknowledge the client
    - 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 = majority
  - > 50%
- 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 being common to both
- Quorums faster than ALL, but still ensure strong consistency



For a set of five replicas we can have two Quorums, Blue and Red but corresponding to the same Key-Value Set

Quorums good as they do not require the coordinator to wait for all the replicas to reply/return an ACK

Quorums in Detail

- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use 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 (update) if needed.

Quorums in Detail (Contd.)

- Writes come in two flavors
  - Client specifies W ($\leq$ N)
  - W = write consistency level.
  - Client writes new value to W replicas and returns. Two flavors:
    - •Coordinator blocks until quorum is reached i.e. all the W replicas have acknowledged
    - •Asynchronous: Just write and return. Later coordinator ensures that at-least W replicas have acknowledged.

Quorums in Detail (Contd.)

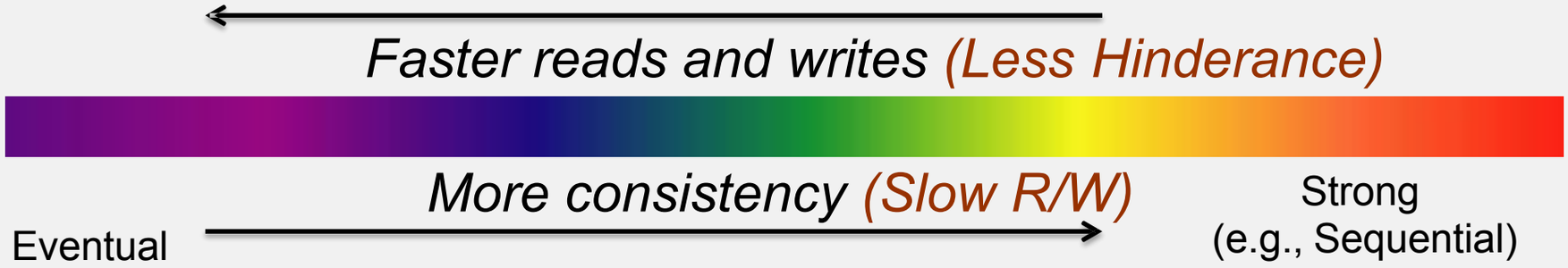
- R = read replica count, W = write replica count
- Two necessary conditions for ensuring strong consistency:
  1. W+R > N
  2. W > N/2 (Same Quorum reason of having intersection)
- Select values based on application
  - (W=1, R=1): very few writes and reads (assuming later updation catches up)
  - (W=N, R=1): great for read-heavy workloads
  - (W=N/2+1, R=N/2+1): great for write-heavy workloads
  - (W=1, R=N): great for write-heavy workloads with mostly one client writing per key

Cassandra Consistency Levels (Contd.)

At the Data Centre (DC) Level

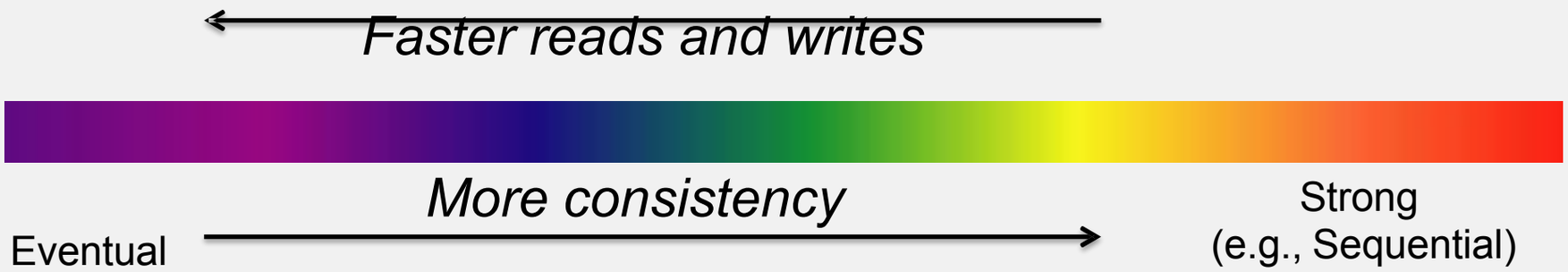
- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator may cache write and reply quickly to client
  - ALL: all replicas
    - Slowest, but ensures strong consistency
  - ONE: at least one replica
    - Faster 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

Consistency Spectrum



Consistency Spectrum

- 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



Summary

- Traditional databases (RDBMSs) work with strong consistency and offer ACID
- Modern workloads don't need such strong guarantees but do need fast response times (availability)
- Unfortunately, CAP theorem
- Key-value/NoSQL systems offer BASE
  - Eventual consistency, and a variety of other consistency models striving towards strong consistency

Thanks!!!