DSA5104

Principles of Data Management and Retrieval

Lecture 9: Key-Value Stores

Recap

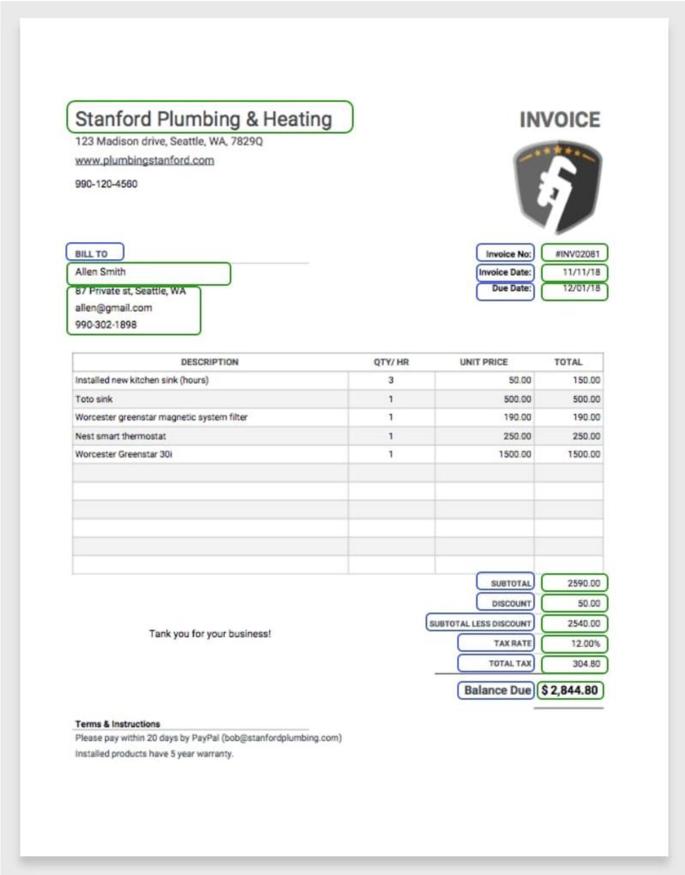
- Complex Data Types
  - Object Orientation
    - Integrate with databases - Object-relational database system
    - Support new types - user defined types/ Table types/multivalued
    - Table/Type Inheritance
    - Reference types
  - Textual Data - unstructured
    - Information retrieval - keyword query
    - IR system vs database systems
    - Relevance ranking for approximate search - Tf-idf / Hyperlinks /Metric
    - Keyword search in relational database
  - Spatial Data
    - Geographic / geometric - representation
    - Applications
    - Spatial query

Key-Value Pair

What is Key-Value Pair?

- A set of two data items – a key and a value
  - The value corresponds to the key, with the key being marked as the *unique identifier*.

Optical Character Recognition (OCR)



Invoice
- Invoice No.
- Date
- Amount
- Tax

key

Value

Survey Forms
- Question
- Answer

https://docsumo.com/blog/key-value-pair-extraction

The Key-Value Abstraction

- Examples of (key, value) pairs in MapReduce
  - (word, count)
  - (year, temperature)
  - (join_key, rest_of_the_tuple)

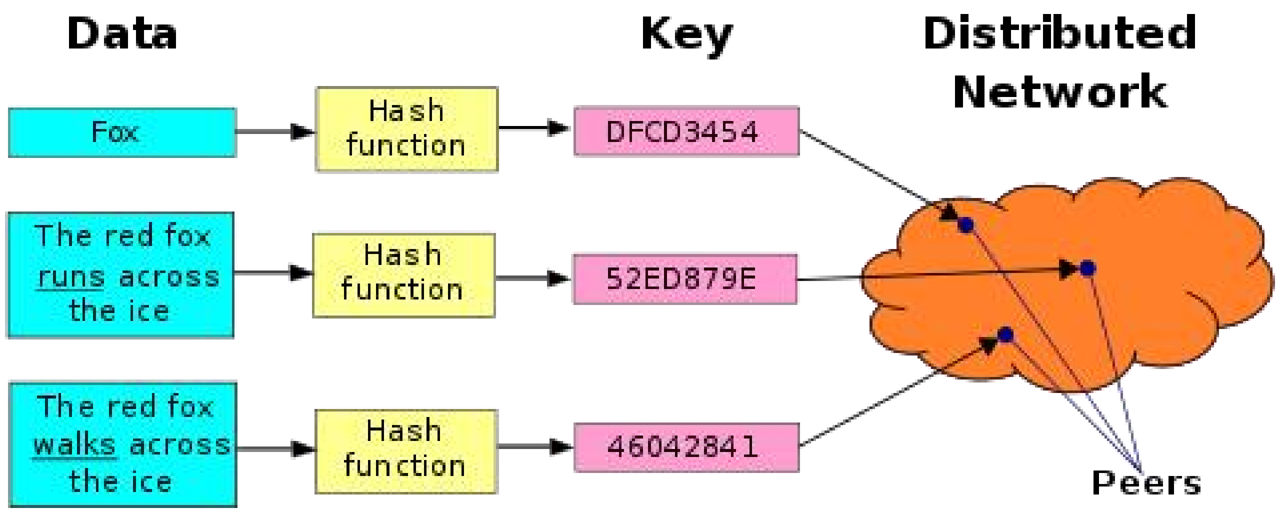
- (Business) Key → Value
- (twitter.com) Tweet id → information about tweet
- (amazon.com) Item number → information about it
- (kayak.com) Flight number → information about flight, e.g., availability
- (web application) Session id → session data including *user profile, messages, themes, recommendations, targeted promotions, and discounts.*
- (E-commerce) Shopping cart id → information about products and orders, discounts, etc.

- Both keys and values can be *anything*, ranging from simple objects to complex compound objects. (Amazon DynamoDB)

The Key-Value Abstraction (Cont.)

- It's a dictionary data structure.
  - Insert, lookup, and delete by key
  - E.g., hash table, binary tree

- But consider the sheer amout of data → maintain on a distributed cluster of servers
- Sound like distributed hash tables (DHT) in P2P systems



DHT is a **distributed system** that provides a lookup service similar to a hash table: key–value pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key.

https://en.wikipedia.org/wiki/Distributed_hash_table

Key-Value Store - Isn't That Just a Database?

- Yes, sort of

- A database management system maintains large amounts of data which can be queried

- E.g., Relational Database Management Systems (RDBMSs)
  - Structured data stored in tables, with well-defined schema
  - Each row (data item) in a table has a primary key that is unique within that table
  - Query data using SQL (Structured Query Language)
  - Supports joins

Relational Database Example

user

<u>userid</u>	name	zipcode	blog_url	blog_id
100	Alice	12345	alice.net	1
200	Bob	56789	bob.blogspot.com	2
333	Charlie	33555	charlie.com	3

Relational Database Example

user

<u>userid</u>	name	zipcode	blog_url	blog_id
100	Alice	12345	alice.net	1
200	Bob	56789	bob.blogspot.com	2
333	Charlie	33555	charlie.com	3

Foreign Key

blog

<u>id</u>	url	last_updated	num_posts
1	alice.net	5/2/22	332
2	bob.blogspot.com	4/2/22	10002
3	charlie.com	6/15/22	11

Relational Database Example

user

userid	name	zipcode	blog_url	blog_id
100	Alice	12345	alice.net	1
200	Bob	56789	bob.blogspot.com	2
333	Charlie	33555	charlie.com	3

Foreign Key

blog

id	url	last_updated	num_posts
1	alice.net	5/2/22	332
2	bob.blogspot.com	4/2/22	10002
3	charlie.com	6/15/22	11

SELECT zipcode
FROM users
WHERE name = "Bob"

SELECT user.zipcode, blog.num_posts
FROM user **JOIN** blog
ON user.blog_id = blog.id

Mismatch with Today’s Workloads

- Data: Large and unstructured → Hard to come up with schemas to fit the data

- Lots of random reads and writes
- Sometimes write-heavy

→ RDBMSs are typically optimized for read-heavy workloads.

- Foreign keys rarely needed
- Infrequent joins

- Weak distributed availability due to poor horizontal scalability (or scale out)

Needs of Today’s Workloads

- Speed → Don’t keep users wait and answer queries very quickly

- Low TCO (Total cost of operation)
- Fewer system administrators

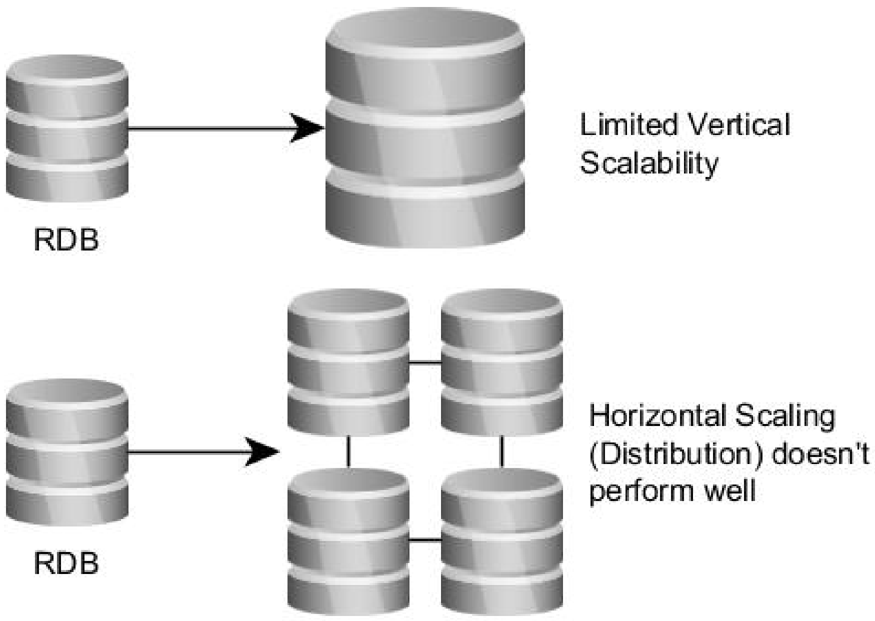
→ Service providers want to minimize the cost

- Incremental scalability → Grow the system based on the workload smoothly
  - Scale out, not up?

Scale Out, Not Scale Up

- Scale up = grow your cluster capacity by replacing with more powerful machines
  - 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 machines (commodity hardware)
  - 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



https://phauer.com/blog/2015/0514-relational-databases-strength-weaknesses-mongodb/RDB-Scalability1.png

Key-Value / NoSQL Data Model

- NoSQL = “Not Only SQL”
  - Support two 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 have a few differences:
    - 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, just like RDBMSs

Key-Value / NoSQL Data Model (Cont.)

- Not structured
- No schema imposed
- Columns missing from some rows
- No foreign keys, joins may not be supported

Key / Value

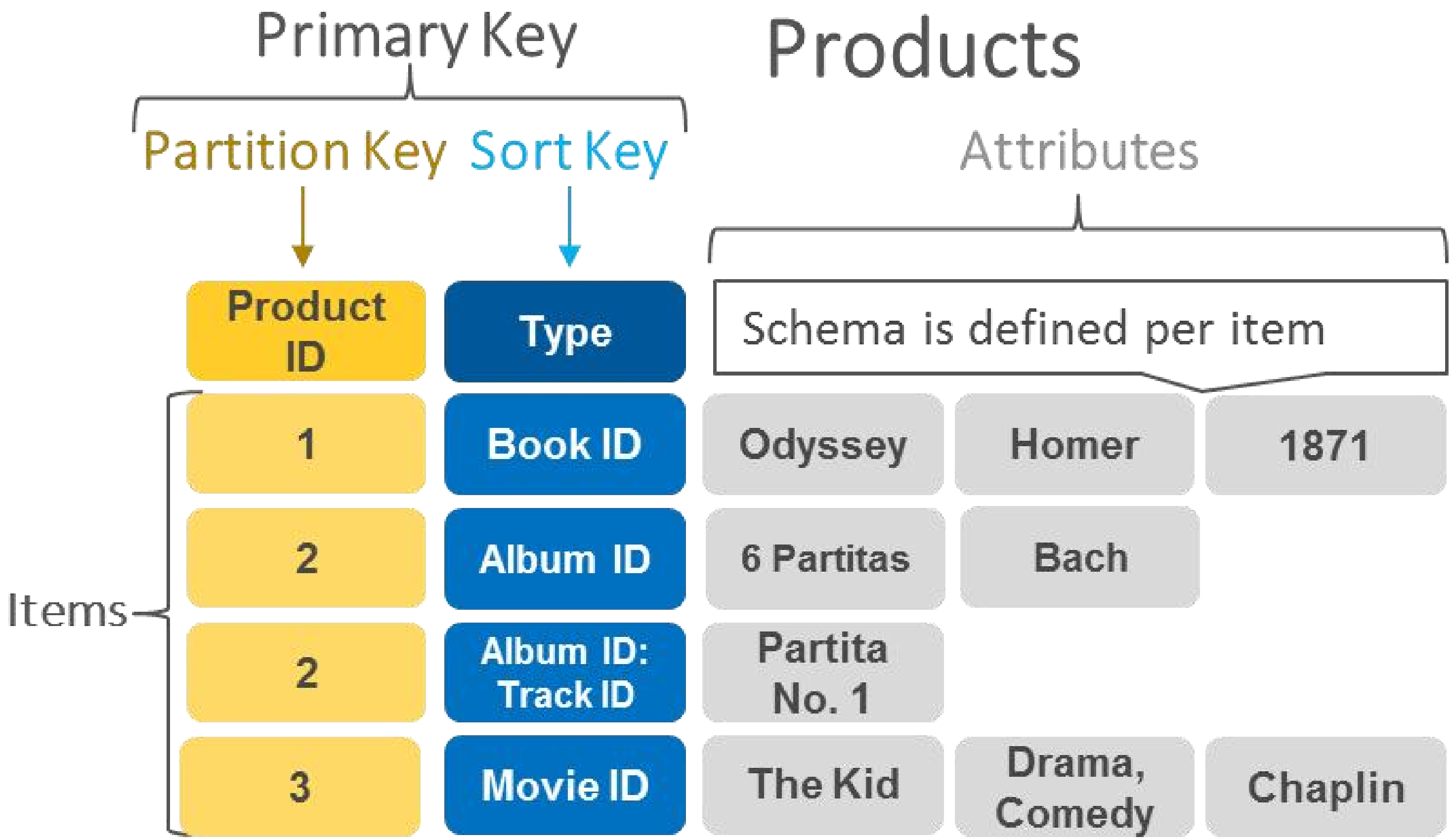
userid	name	zipcode	blog_url	blog_id
100	Alice	12345	alice.net	
200	Bob		bob.blogspot.com	
333		33555	charlie.com	

Key / Value

id	url	last_updated	num_posts	num_likes
1	alice.net	5/2/22	332	999
2	bob.blogspot.com	4/2/22	10002	
3	charlie.com		11	

Key-Value / NoSQL Data Model (Cont.)

- An example of data stored as key-value pairs in Amazon DynamoDB.



Column-Oriented Storage

NoSQL systems often use column-oriented storage

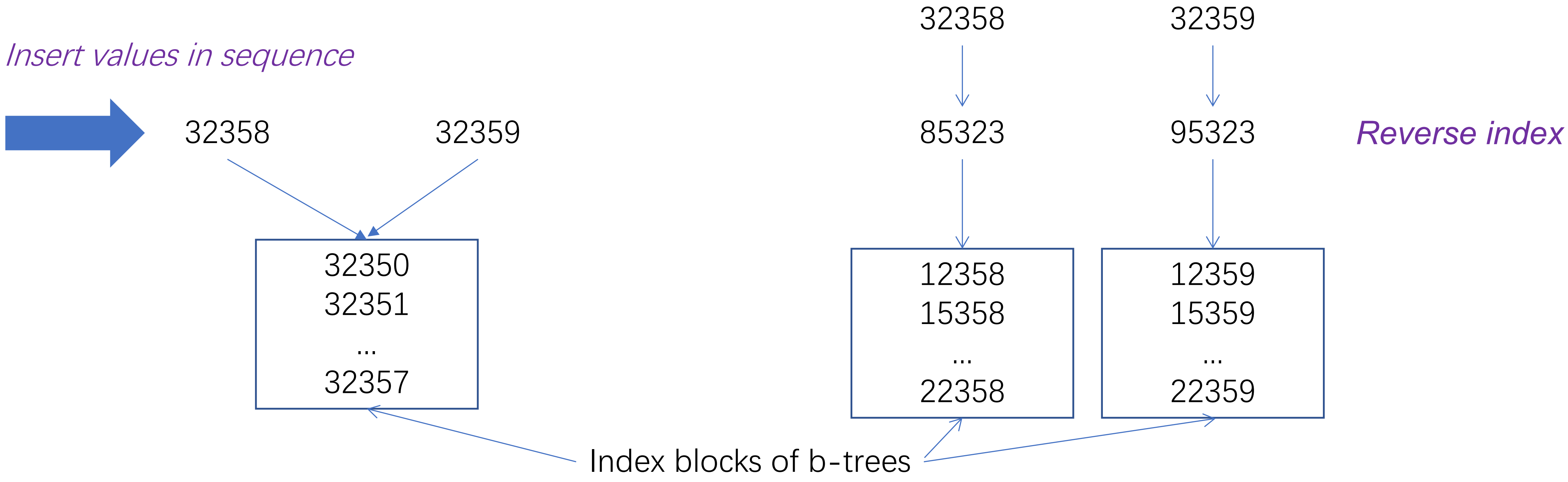
- RDBMSs store an entire row together (on disk or at a server)

- 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

Key-Value Store - Cassandra

Where Did Cassandra Come From?

- Cassandra originated at Facebook in 2007 to solve its inbox search problem
- Facebook had to deal with huge volumes of data in the form of message copies, *reverse indices* of messages, and many random reads and many simultaneous random writes.



Where Did Cassandra Come From?

- Cassandra originated at Facebook in 2007 to solve its inbox search problem
- Facebook had to deal with huge volumes of data in the form of message copies, *reverse indices* of messages, and many random reads and many simultaneous random writes.
- The code was open-sourced in July 2008 and now a Apache project.

Cassandra - A Decentralized Structured Storage System

Avinash Lakshman
Facebook

Prashant Malik
Facebook

Cassandra

Lakshman, Avinash, and Prashant Malik. "Cassandra: a decentralized structured storage system." ACM SIGOPS Operating Systems Review 44.2 (2010): 35-40.

Cassandra

- "*Apache Cassandra* is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, column-oriented database that bases its distribution design on Amazon's *Dynamo* and its data model on Google's *Bigtable*"

  - "Cassandra: The Definitive Guide," O'Reilly Media, 2010, p.14

- A distributed key-value store
- Intended to run in a datacenter (and also across multiple DCs)

- 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

Key → Server Mapping

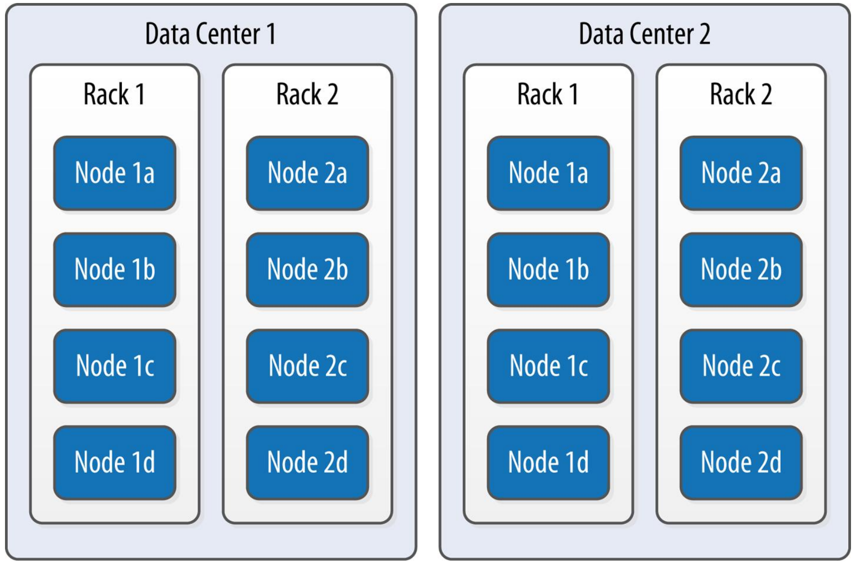
- How Cassandra distributes data across these nodes

→ How to decide which server(s) a key-value resides on?

Data Centers and Racks

- Cassandra provides two levels of grouping to describe the topology of a cluster
  - Data center (DC) and rack
- A rack is a logical set of nodes in close proximity to each other → on physical machines in a single rack of equipment.
- A data center is a logical set of racks → located in the same building and connected by reliable network.

Topology of a sample cluster with data centers, racks, and nodes



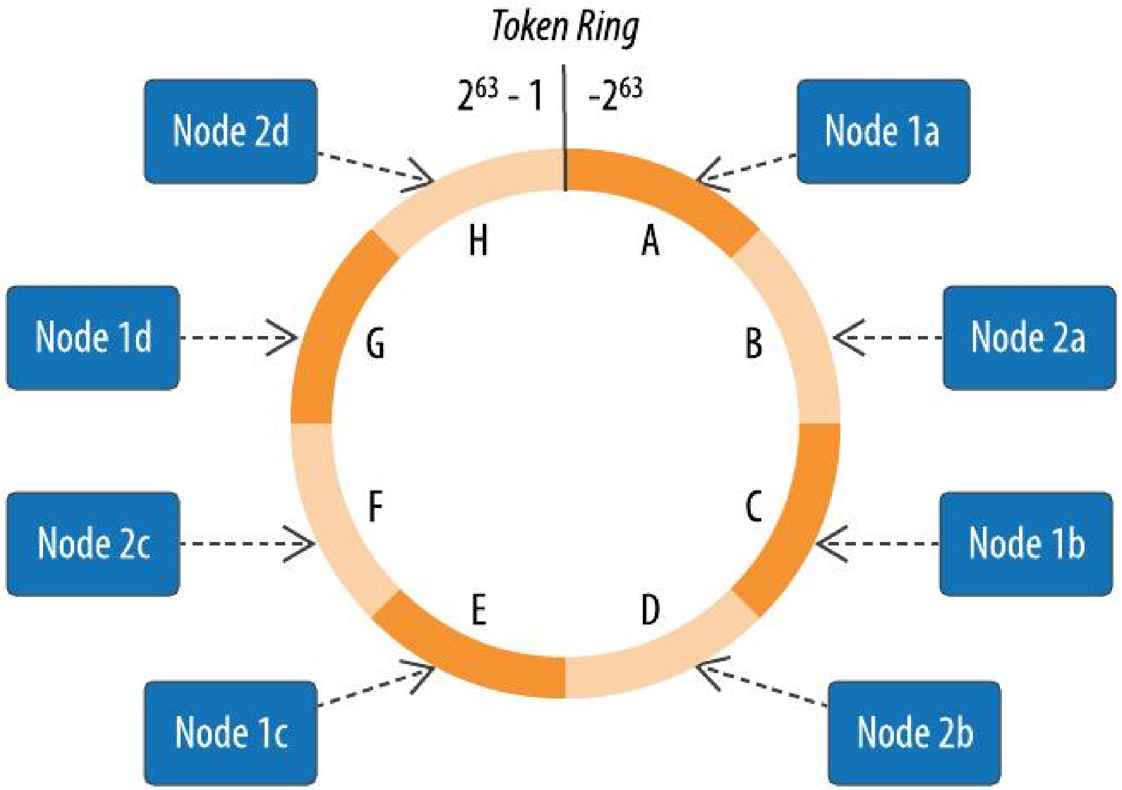
Cassandra: The Definitive Guide, (Revised) Third Edition, 3rd Edition, O'Reilly

https://media.fs.com/images/community/upload/kindEditor/202103/20/data-center-server-rack-wiki-1616212663-TCalmdBNU2.jpg

Rings & Tokens

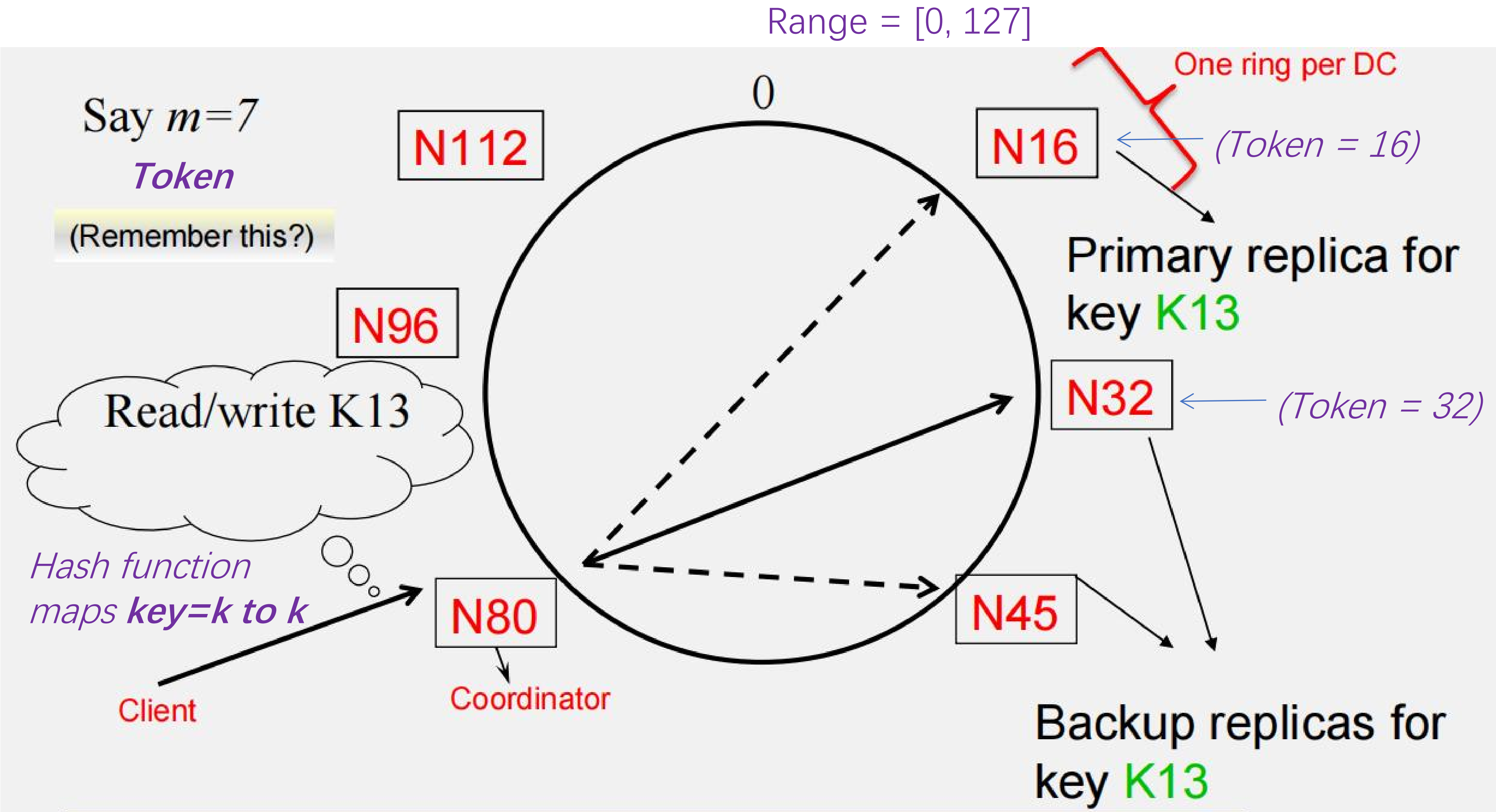
- Cassandra represents the data managed by a cluster as a ***ring***.
- Each node in the ring is assigned one or more ranges of data described by a ***token***, which determines its position in the ring.
- A node claims ownership of the range of values less than or equal to each token and greater than the last token of the previous node, known as a token range.
- Data is assigned to nodes by using a hash function to calculate a token for the partition key, which is compared to the token values for the various nodes.

By default, a token is a 64-bit integer ID used to identify each partition.



Example ring arrangement of nodes in a data center.

Rings & Tokens (Cont.)

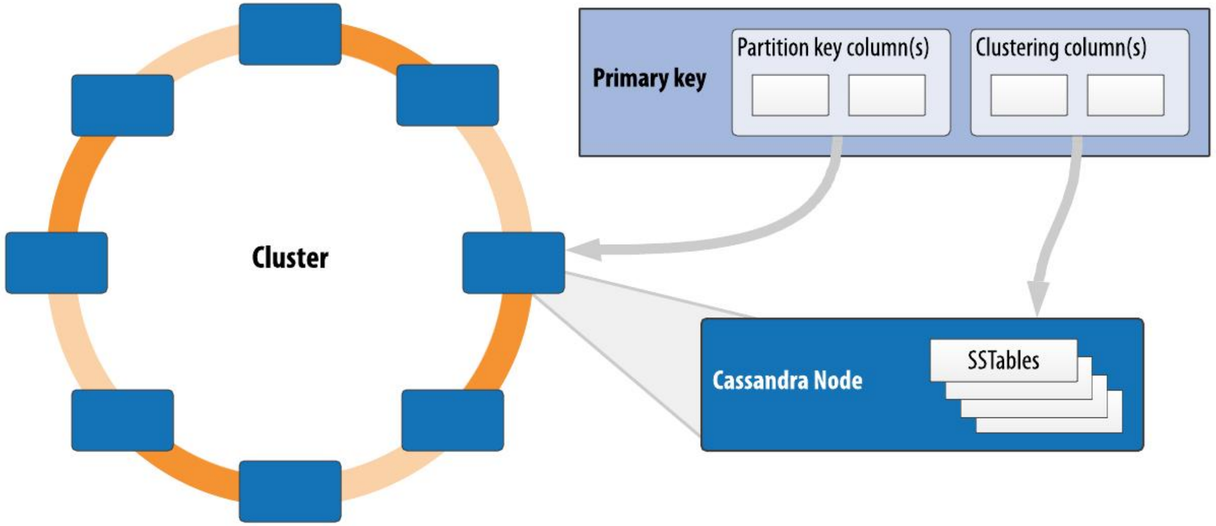


Cassandra uses a ring-based DHT but without finger tables or routing

Key→server mapping is the "Partitioner"

Partitions & Partitioners

- Cassandra organizes rows in partitions.
- Each row has a partition key that is used to identify the partition to which it belongs.
- A partitioner is a hash function for computing the token of a partition key.
- Each row of data is distributed within the ring according to the value of the partition key token.



Replication Strategies

Replication Strategy:

- SimpleStrategy
- NetworkTopologyStrategy

- SimpleStrategy (uses the partitioner) - place replicas at consecutive nodes around the ring, starting with the node indicated by the partitioner.
  1. RandomPartitioner: essentially keys are hashed to a point in the ring
  2. ByteOrderedPartitioner: assigns ranges of keys (e.g., timestamps) to servers
     - Easier for range queries (e.g., get me all twitter users starting with [a-b])

- NetworkTopologyStrategy (for multi-DC deployments) - allows you to specify a different replication factor for each DC
  - E.g., two replicas per DC / three replicas per DC
  - Per DC
    - First replica placed according to Partitioner
    - Attempts to choose replicas within a datacenter from different racks

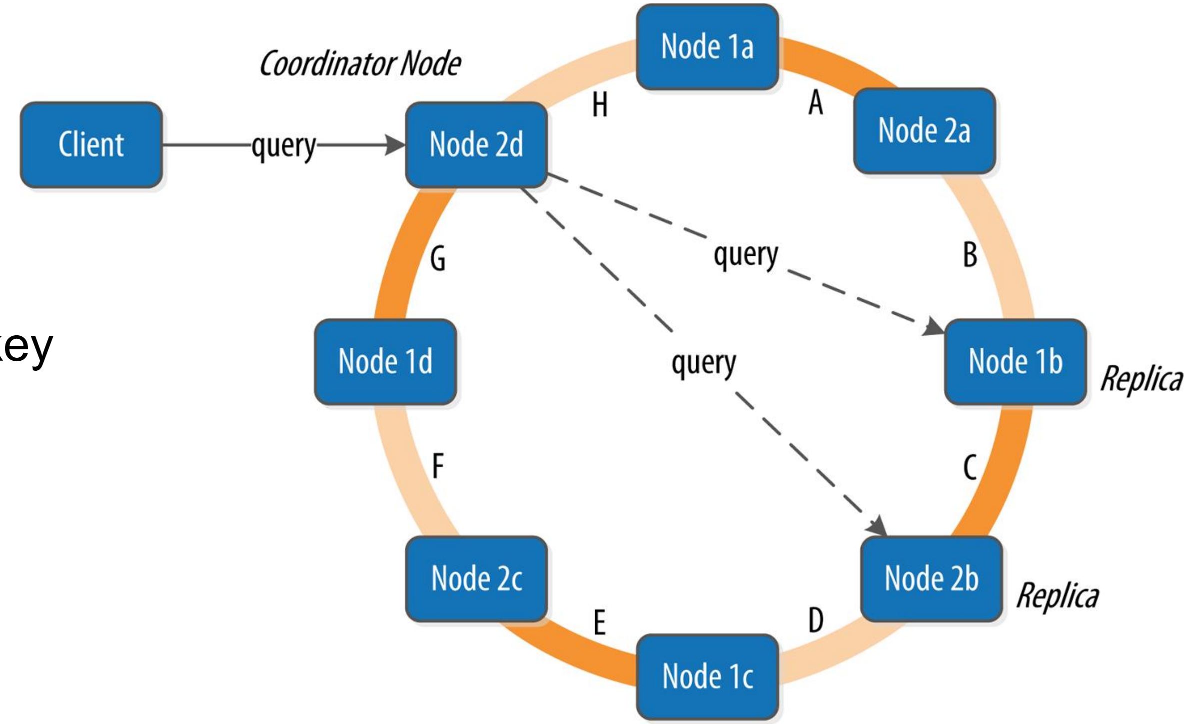
Snitches

- Maps: IPs to racks and DCs. Configured in *cassandra.yaml* config file
  - To provide information about your network topology → Cassandra can efficiently route (read/write) requests.
  - E.g., determine relative host proximity for each node in a cluster

- Options:
  - SimpleSnitch: topology unaware (Rack-unaware)
  - RackInferring: Assumes topology of network by octet (8-bit byte) of server's IP address
    - 101.201.301.401 = x.<DC octet>.<rack octet>.<node octet>
  - PropertyFileSnitch: uses a config file (accurate mapping of IP addresses to racks and DCs)
  - Cassandra provides snitches for different network topologies and cloud environments, including Amazon EC2, Google Cloud, and Apache Cloudstack.
    - EC2Snitch: uses EC2
      - EC2 Region = DC
      - Availability zone = rack

Write

- Need to be lock-free and fast → deadling with write heavy workloads
- 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
- Coordinator uses Partitioner to send query to all replica nodes responsible for key
- When X replicas respond, coordinator returns an acknowledgement to the client
  - What is the value of X? We'll see later.

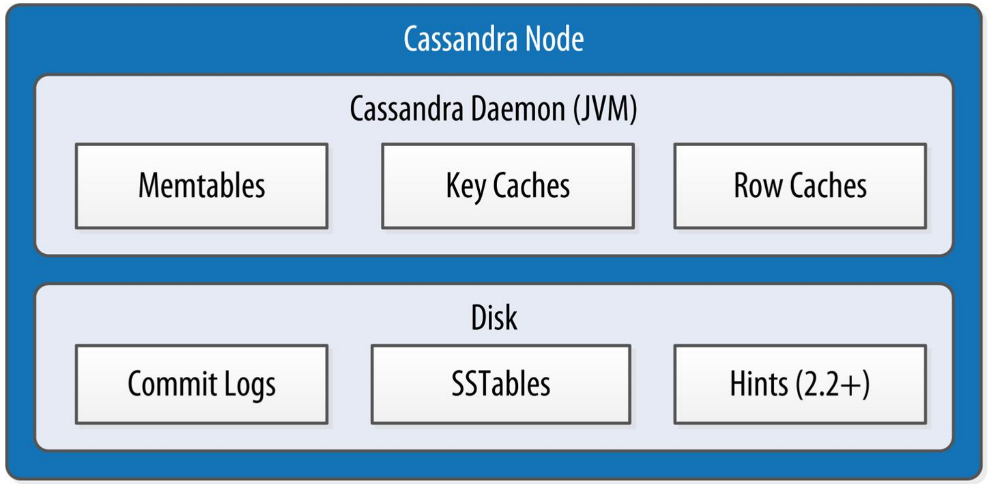


Clients, coordinator nodes, and replicas

Write (Cont.)

- 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 buffers writes (for up to a few hours).
- One ring per DC
  - Per-DC coordinator elected to coordinate with other DCs
  - Election done via Zookeeper

Write At a Replica Node



On receiving a write

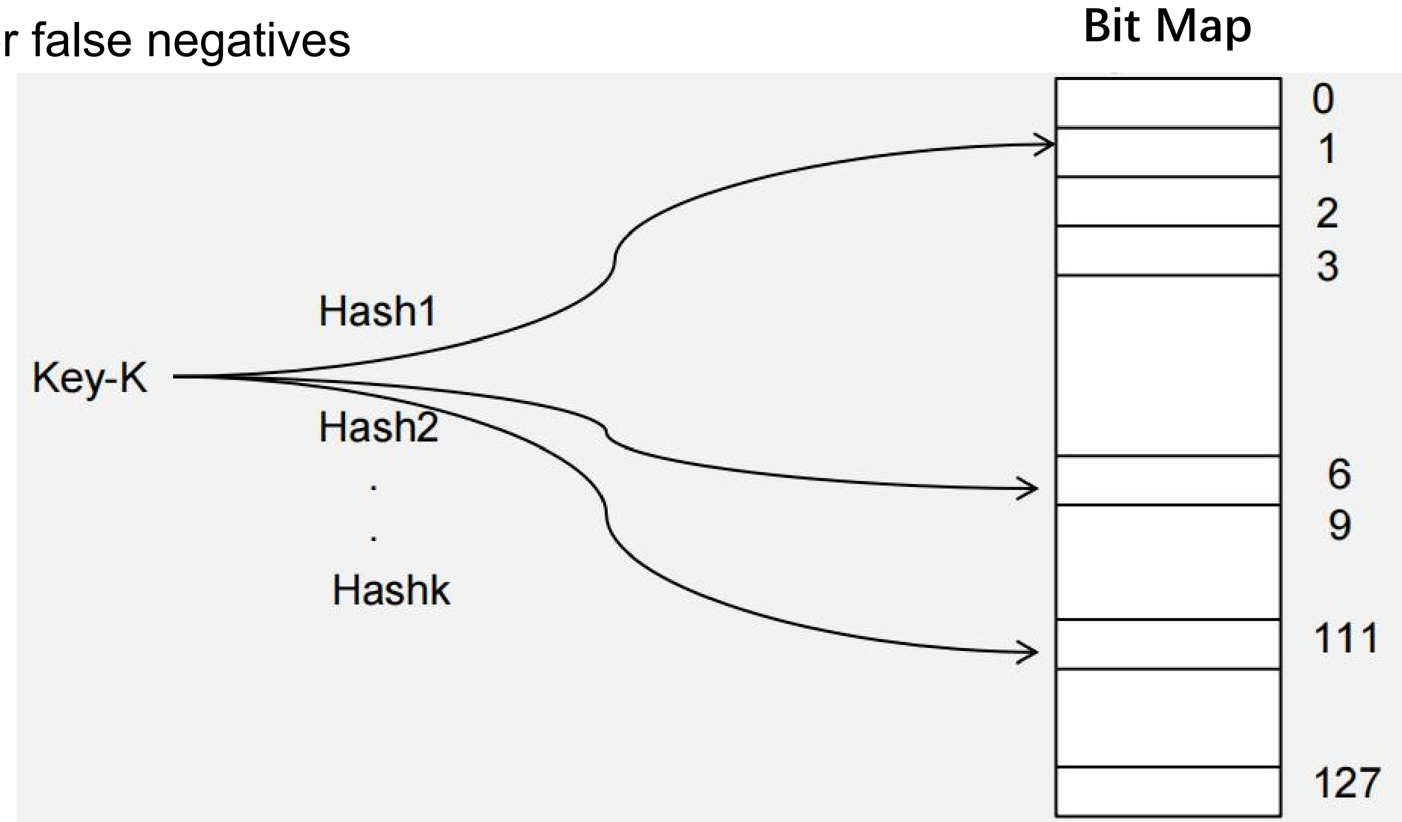
1. Log it in disk commit log (for failure recovery)

2. Make changes to appropriate memtables
   - Memtable = In-memory representation of multiple key-value pairs
   - Cache that can be searched by key
     - Update key’s value in memtable or append key-value pair to memtable
   - This is cal 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
   - Add a Bloom filter (for efficient search) – next slide

Bloom Filter

- Used to boost the performance of reads.
- Compact way of representing a set of items
- Checking for existence in set is cheap (very fast)
- 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 (FPs)

False positive rate low

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

Compaction

- Data updates accumulate over time and SStables and logs need to be compacted
  - A given key might be present in multiple SStables → Not efficient for e.g., a “read” request
  - The process of compaction merges SSTables, i.e., by merging updates for a key (keep the latest update)
  - Run periodically and locally at each server

Delete

- Delete: don’t delete a key-value pair right away
  - Add a tombstone to the log - a “soft delete”
    - A tombstone is a marker that is kept to indicate data that has been deleted.
  - Eventually, when compaction encounters tombstone, it will delete the item
  - To prevent deleted data from being reintroduced
    - A failed node when recovers may ‘resurrect’ data

Read

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
  - What is the value of 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)

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

- Gossiper - responsible for managing gossip for the local node.

- Gossip is used for failure detection

- Here is how the gossiper works:
  1. Once per second, the gossiper will choose a random node in the cluster and initialize a gossip session with it.
  2. The gossip initiator sends its chosen friend a *GossipDigestSyn* message.
  3. When the friend receives this message, it returns a *GossipDigestAck* message.
  4. When the initiator receives the ack message from the friend, it sends the friend a *GossipDigestAck2* message to complete the round of gossip.

Suspicion Mechanisms in Cassandra

- Cassandra uses suspicion mechanisms to increase the accuracy of failure detections.
- Suspicion mechanisms to adaptively set the timeout based on underlying network and failure behavior

- Cassandra - Phi Accrual Failure Detector
- Accrual detector: Failure detector outputs a value (PHI) representing suspicion
- Set an appropriate threshold
  - Adjusts the sensitivity of the failure detector.
- PHI calculation for a member
  - PHI basically determines the detection timeout, but takes into account historical inter-arrival time variations for gossiped heartbeats
  - PHI is designed to be adaptive in the face of volatile network conditions
- In practice, PHI = 5 => 10-15 sec detection time

Cassandra vs. RDBMS

- Use MySQL as an example of RDBMS

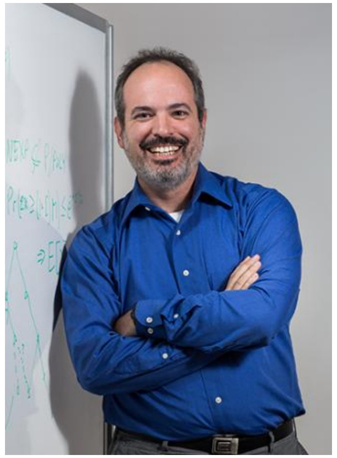
On > 50 GB data

- MySQL
  - Write - 300 ms avg
  - Read - 350 ms avg
- Cassandra
  - Write - 0.12 ms avg
  - Read - 15 ms avg
- Orders of magnitude faster

- What’s the catch? What did we lose?

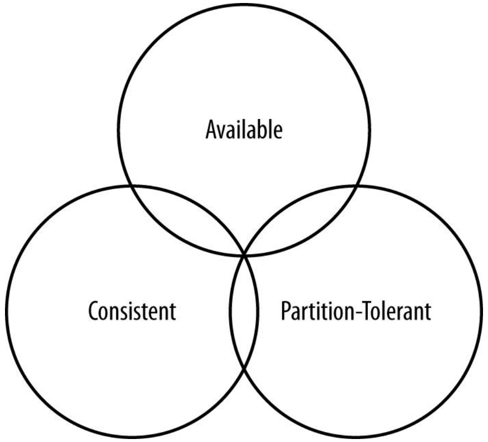
The CAP Theorem

CAP Theorem



Seth Gilbert

- Proposed by Eric Brewer, 2000 (Berkeley)
- Subsequently proved by *Gilbert* and Lynch, 2002 (*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 all the time, and operations return quickly
  3. **Partition-tolerance**: the system continues to work in spite of network partitions



"The CAP theorem encourages engineers to make awful decisions." – Stonebraker

Why is Availability Important?

- Availability = Reads/writes complete reliably and quickly.
- Amazon find that 100ms of latency cost them 1% of sales and then Google discovered that a half second increase in search latency dropped traffic by 20% (2020)
- 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
- 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 (devices).
- 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
  - Domain Name System (DNS) not working
- 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

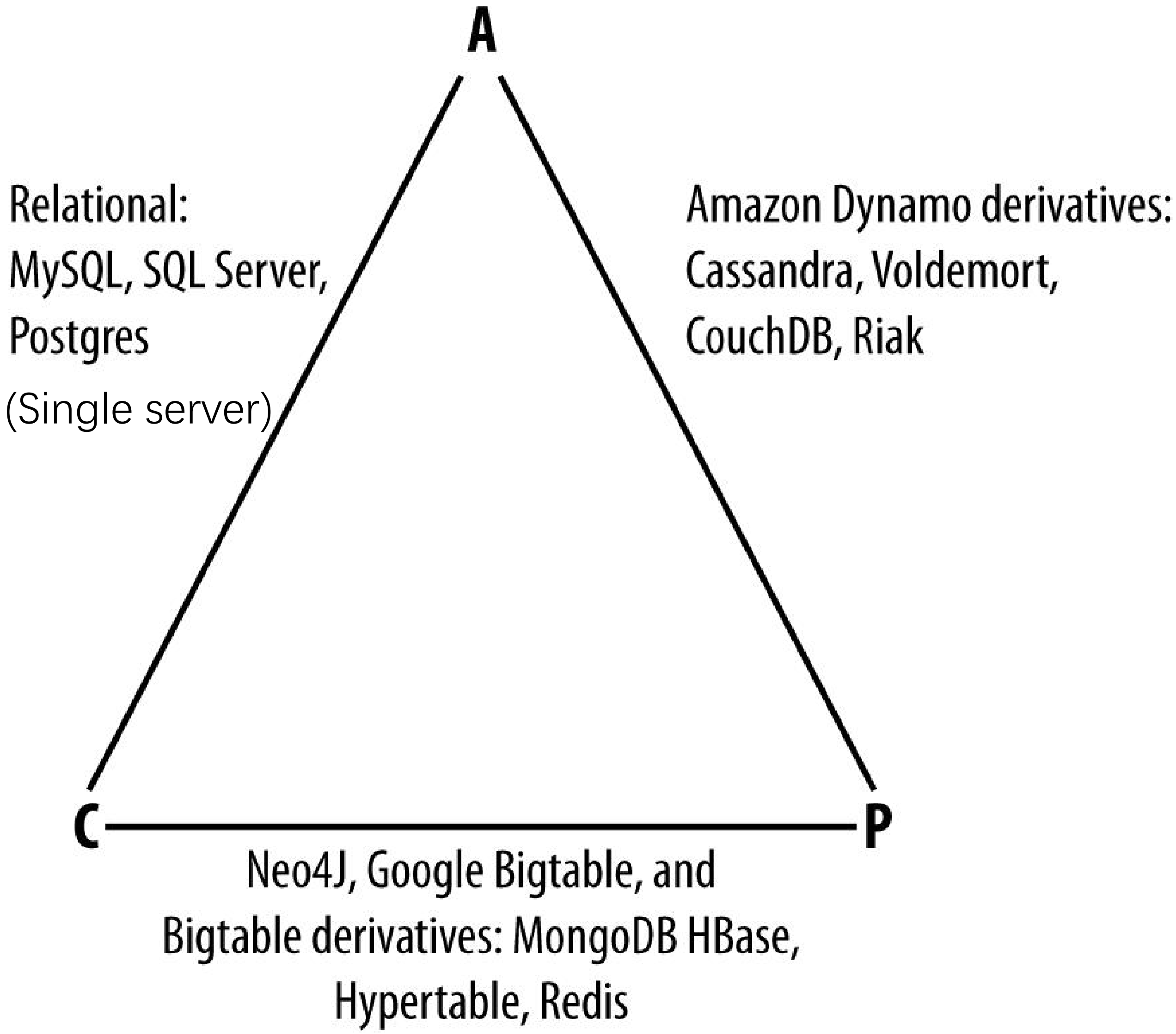
- Since partition-tolerance is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability

- Cassandra
  - Eventual (weak) consistency, availability, partition-tolerance

- Traditional RDBMSs
  - Strong consistency over availability under a partition

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



Where different databases appear on the CAP continuum

Eventual Consistency

- If all writes stop (to a key), then all its values (replicas) will converge eventually.
- 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 the front wave of latest writes.
- 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
  - Prefers availability over consistency

Back to Cassandra - Mystery of X

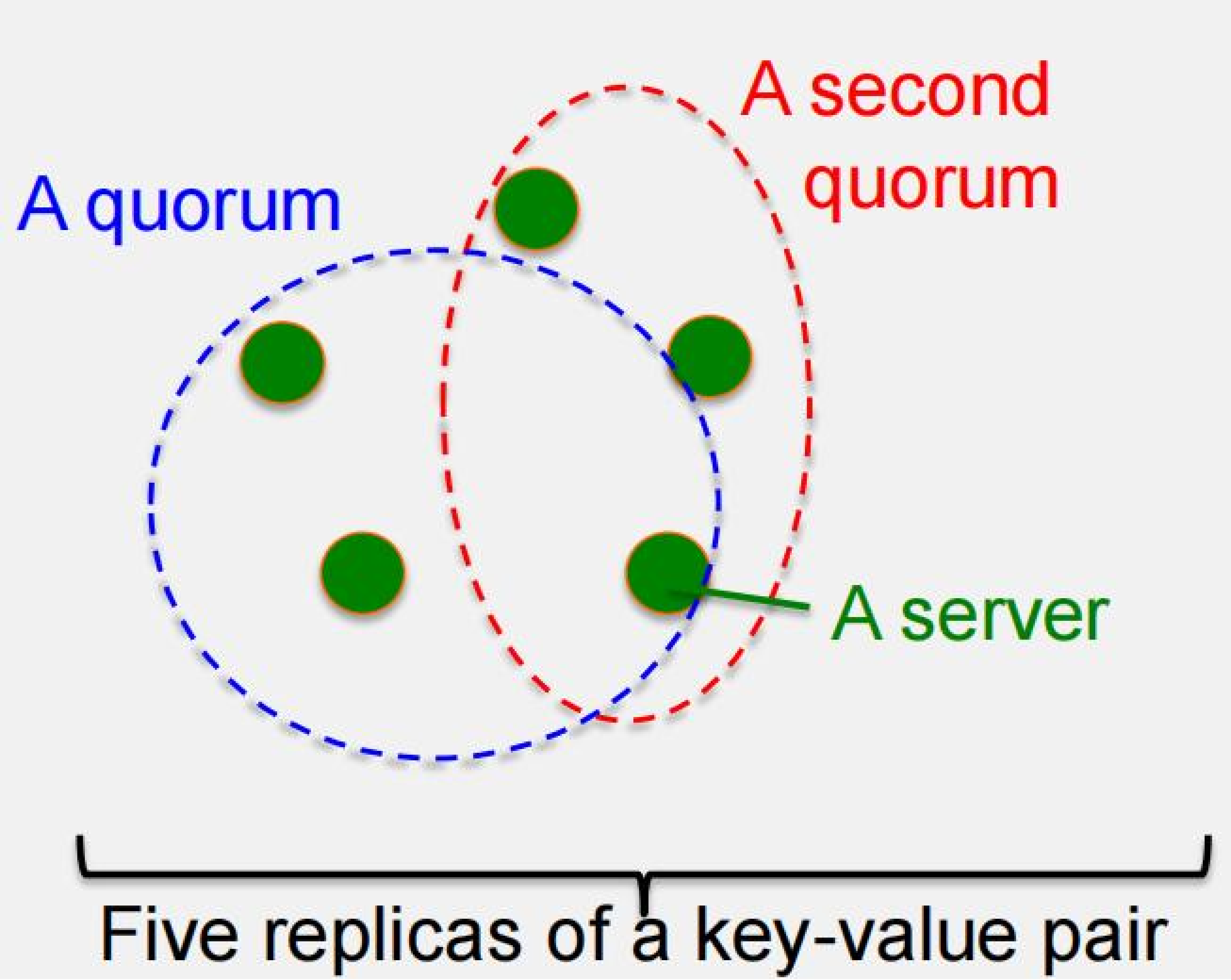
- Value of X - Consistency levels in Cassandra

- Client is allowed to choose a consistency level for each operation (read/write)
  - **ANY**: any server (may not be replica) can store that particular write, return immediately to the client.
    - 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 of replies across all replicas in all datacenters (DCs)
    - Next slide

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 the latest write
- Quorums faster than ALL, but still ensure strong consistency



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 if needed

Quorums in Detail (Cont.)

- Writes come in two flavors
  - 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.
    - Asynchronous: Just write and return.

Quorums in Detail (Cont.)

- R = read replica count, W = write replica count

- Two necessary conditions:
  1. W+R > N
  2. W > N/2

- Select values based on application
  - (W=1, R=1): very few writes and reads
  - (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

- 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)
    - 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

E.g, Consider 3 DCs, each with 3 replicas of a given key

HBase

HBase

- Google’s BigTable was first “blob-based” storage system (paper published in 2006)
- Yahoo! Open-sourced it → HBase
- Major Apache project today
- A variety of companies use HBase including Facebook
- API functions
  - Get/Put by key
  - Scan(row range, filter) – range queries
  - MultiPut
- Unlike Cassandra, HBase prefers consistency (over availability) under partitions

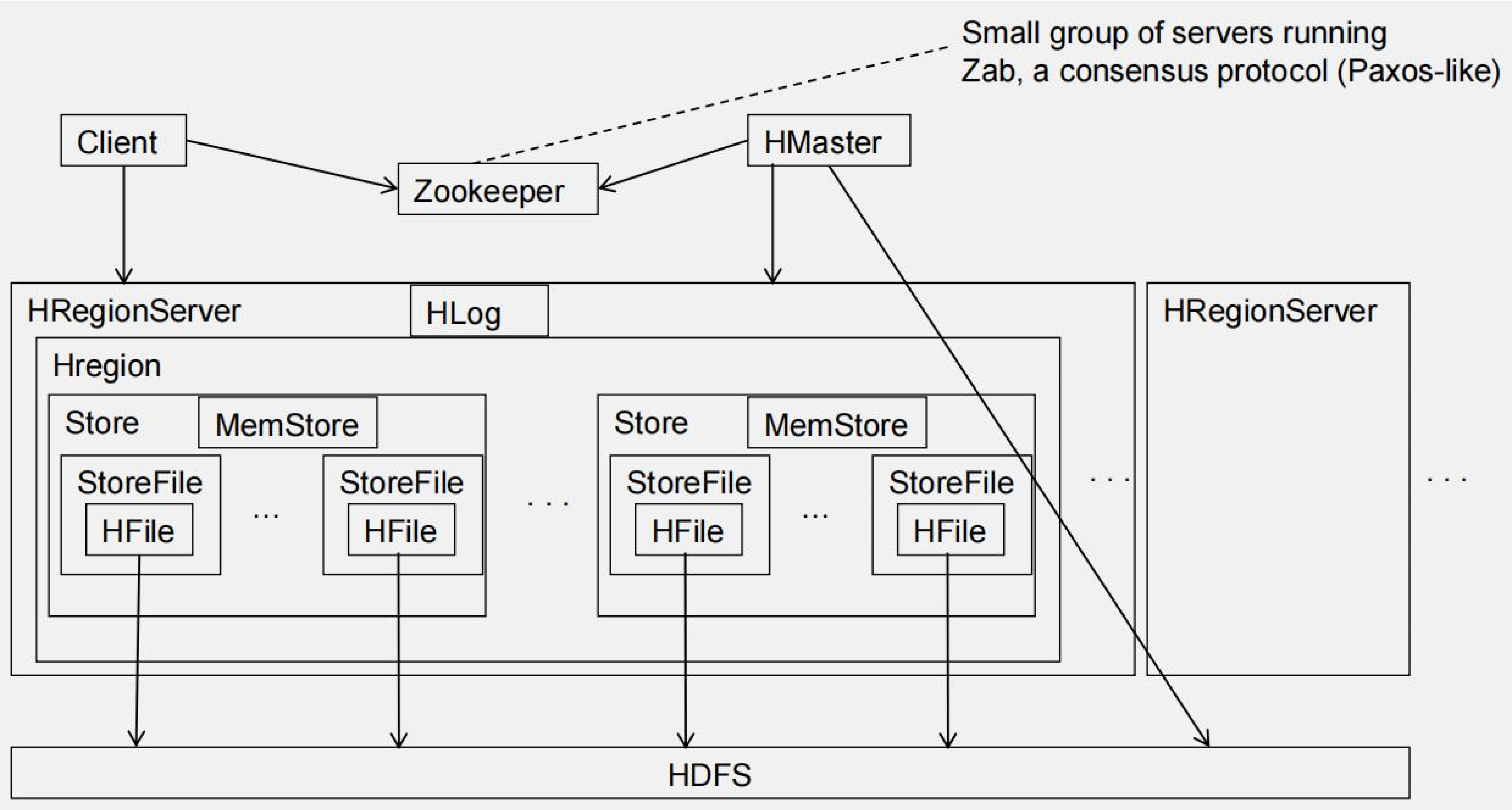
Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

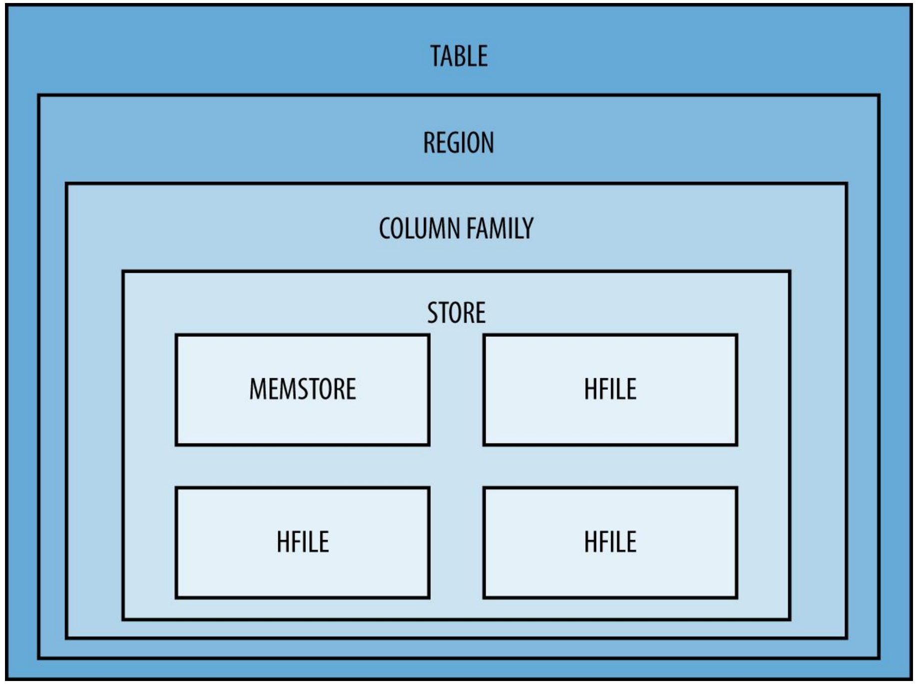
HBase Architecture



HBase Storage Hierarchy

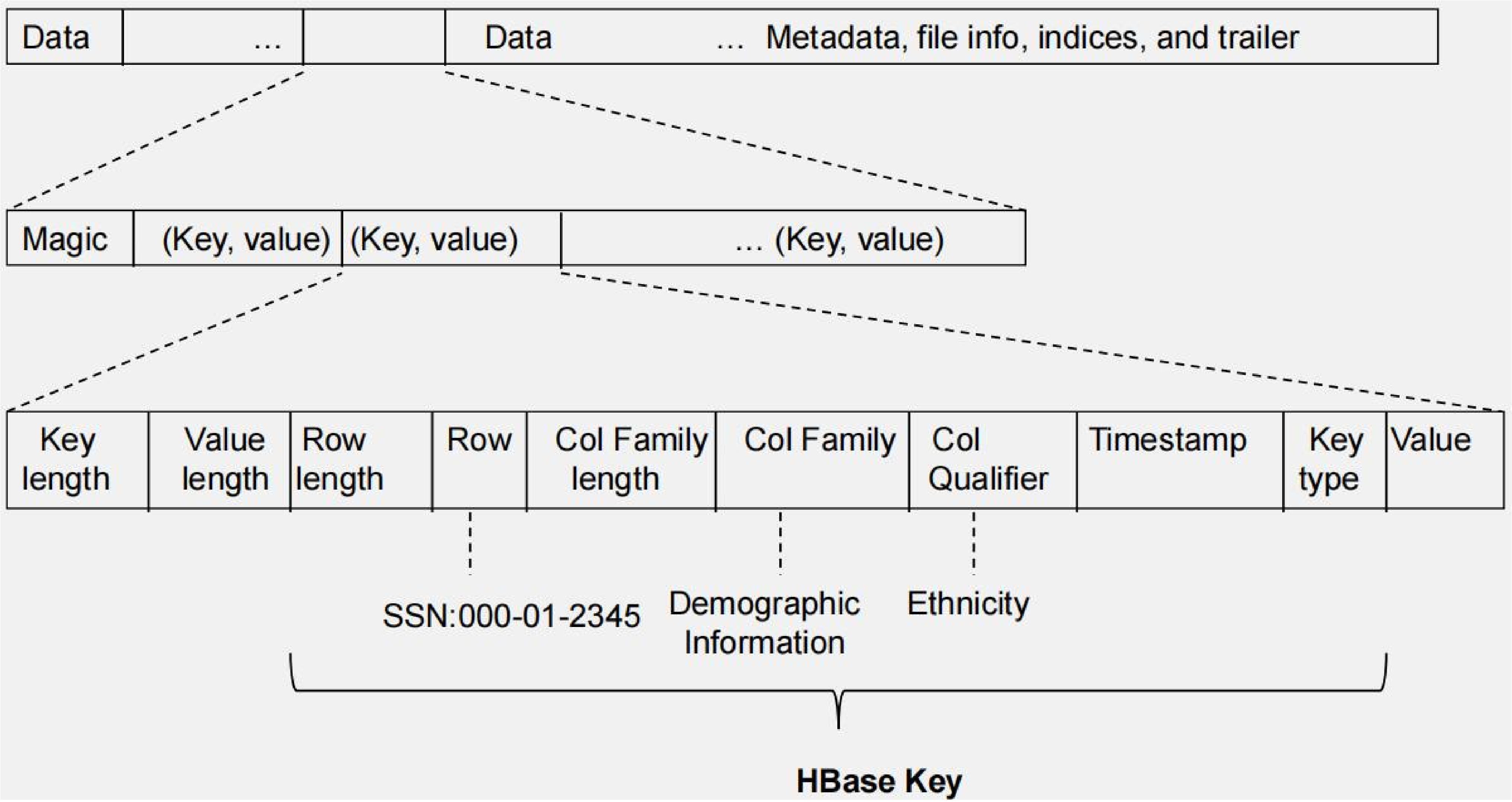
- HBase Table
  - Split it into multiple regions: replicated across servers
    - ColumnFamily = subset of columns with similar query patterns
    - One Store per combination of ColumnFamily + region
      - Memstore for each store: in-memory updates to store; flushed to disk when full
        - StoreFiles for each store for each region: where the data lives
          - HFile

- HFile
  - SSTable from Google's BigTable

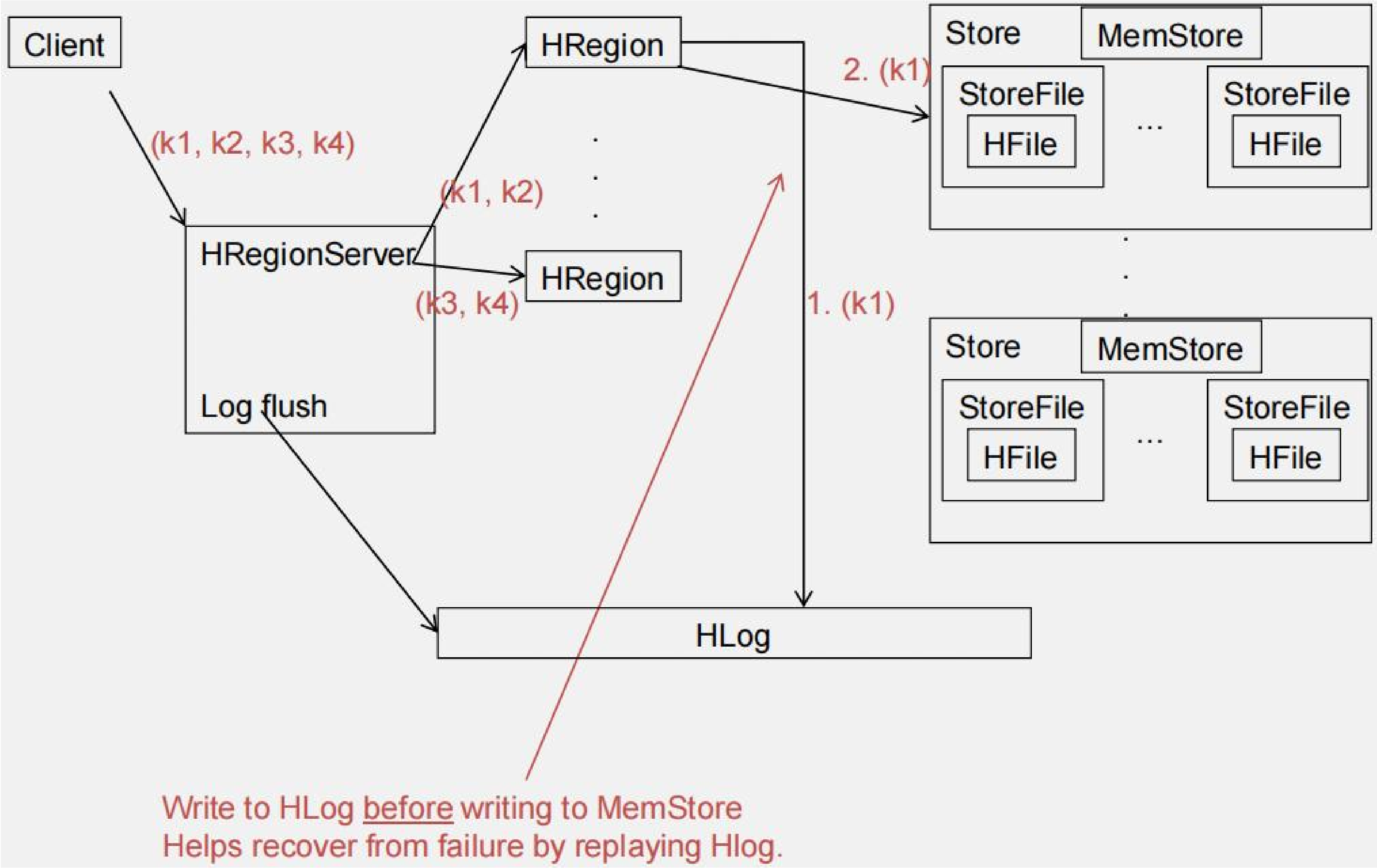


Storage layers

HFile


Data
...
Data
... Metadata, file info, indices, and trailer
Magic
(Key, value)
(Key, value)
... (Key, value)
Key length
Value length
Row length
Row
Col Family length
Col Family
Col Qualifier
Timestamp
Key type
Value
SSN:000-01-2345
Demographic Information
Ethnicity
HBase Key

Strong Consistency - HBase Write-Ahead Log



Log Replay

- After recovery from failure, or upon bootup (HRegionServer/HMaster)
  - Replay any stale logs (use timestamps to find out where the database is w.r.t. the logs)
  - Replay: add edits to the MemStore

Cross-Datacenter Replication

- Single “Master” cluster
- Other “Slave” clusters replicate the same tables
- Master cluster synchronously sends HLogs over to slave clusters

- Coordination among clusters is via Zookeeper
- Zookeeper can be used like a file system to store control information
  1. /hbase/replication/state
  2. /hbase/replication/peers/<peer cluster number>
  3. /hbase/replication/rs/<hlog>

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)

- CAP theorem
- Key-value/NoSQL systems offer BASE
  - Eventual consistency, and a variety of other consistency models striving towards strong consistency

- We discussed design of
  - Cassandra
  - HBase