

CS 425 / ECE 428

Distributed Systems

Fall 2026

Aishwarya Ganesan & Ram Kesavan

Lecture 11: KV Stores & Consistency Models

Ack: Indranil Gupta

All slides © IG

Announcements

- HW2 due on Sunday 10/4 @11:59 pm US Central
 - Solutions will be posted early next week
- MP2 is being graded this week
- HW1 grades released (see Piazza post)
- In-class midterm Thursday 10/8
 - Course webpage specifies the two rooms
 - **Make sure you're in correct classroom for the midterm**
 - Piazza post with more details
 - Practice mid-term released



Today's Agenda: Finish up KV stores

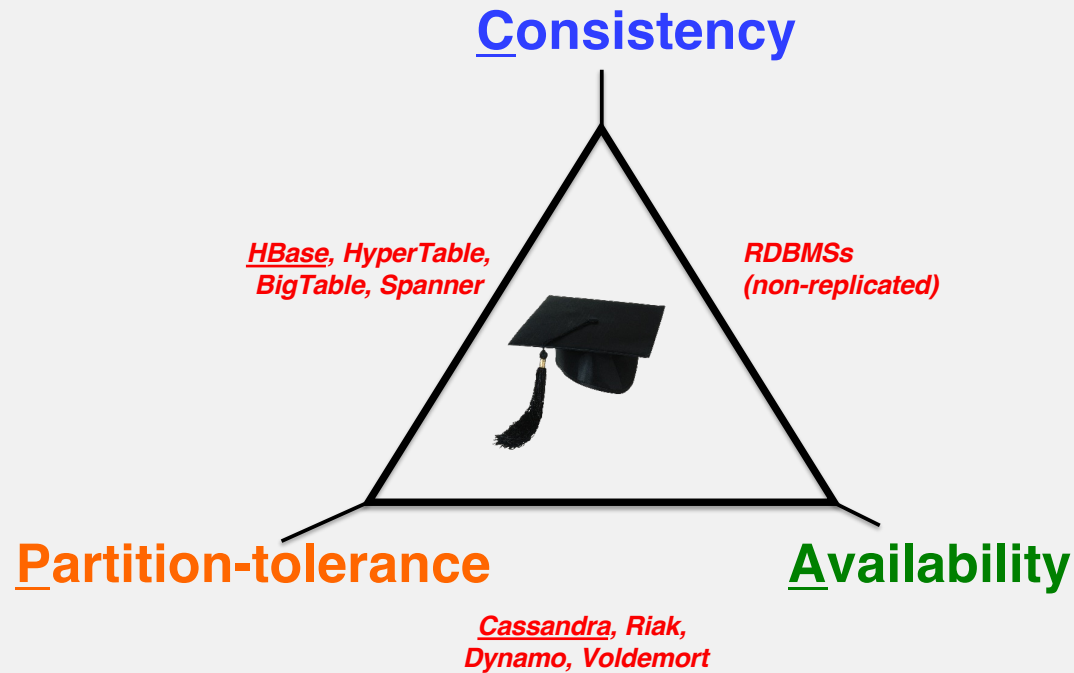
- Consistency Models
 - Consistency vs. Availability tradeoff
 - Cassandra: Availability over Consistency
 - HBase: Consistency over Availability
- Per-replica local storage (LSM-tree)

Start on Time/Ordering

Quick Recap: NoSQL & Cassandra

- Convergence: scale-out & SQL-like ACID
 - Traditional RDBMS: SQL, ACID, enterprise workloads
 - Late '00s & '10s KV-stores: NoSQL, BASE, new workloads
- Cassandra: distributed KV-store
 - Replication: which replicas hold a key
 - Writes & reads; Timestamps
- CAP theorem

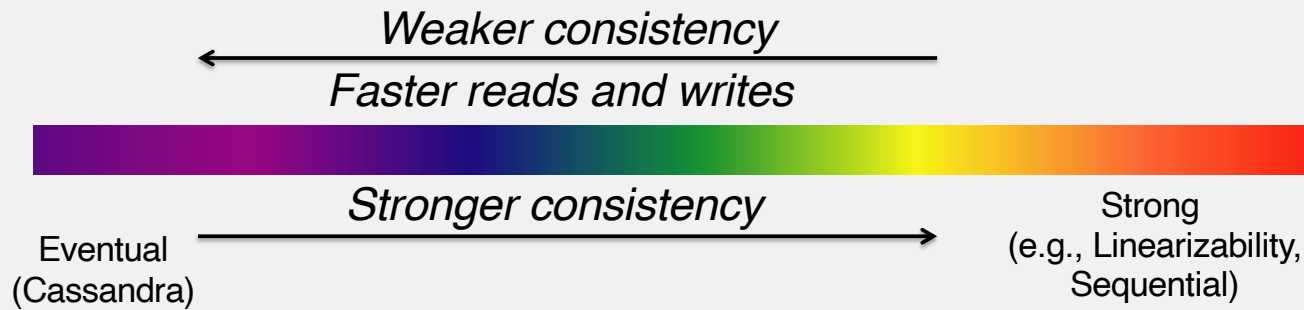
Quick Recap: CAP Tradeoff



A distributed storage system can achieve **at most two of C, A, and P**.

If partition-tolerance is a must-have, the choice is between consistency and availability

Consistency Spectrum



Consistency Models

Contract between distributed system and application

Dictates what results the system can return for operations

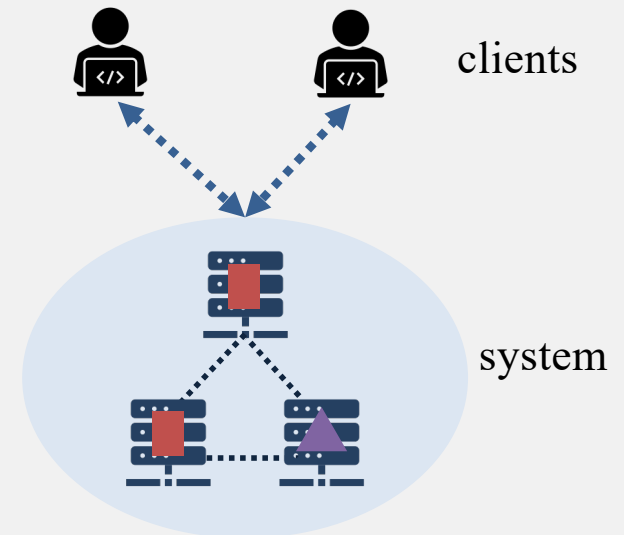
Concurrent reads and writes by clients on replicas

Depending on how operations are performed, guarantees offered by the distributed system differ

Consistency model dictates what is permissible

System developer: design/optimize system to meet contract

Application developer: understand what contract means for application correctness



Spectrum Ends: Eventual Consistency

Cassandra offers **Eventual Consistency**

originally from Amazon's Dynamo and LinkedIn's Voldemort

If writes to a key stop, all replicas of key will converge

Reads might see any previous write

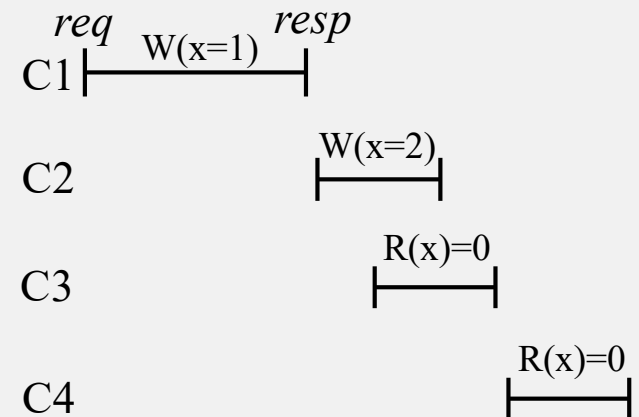
weak guarantee

More **performant** – allows concurrent writes

More **availability** – disconnected operation

Need to resolve conflicts between multiple object versions

Cassandra – latest timestamp wins



Spectrum Ends: Strong Consistency

Linearizability [Herlihy and Wing]

Strong consistency (the **C** in **CAP**)

Intuitively, each operation by a client is visible instantaneously to other clients

instantaneously in real time

All operations are totally ordered in a way that respects real-time order of operations

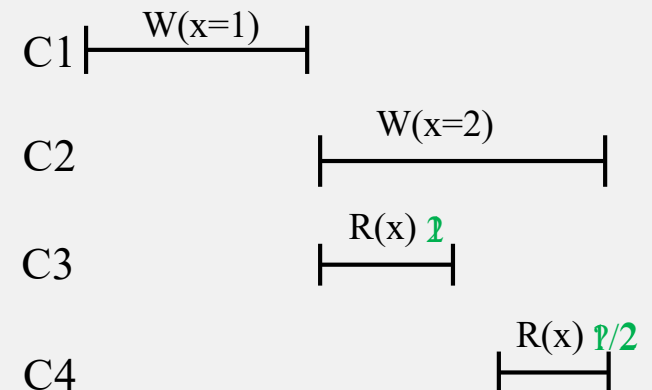
if **A completes before B begins**, A ordered before B

what if A and B are concurrent?

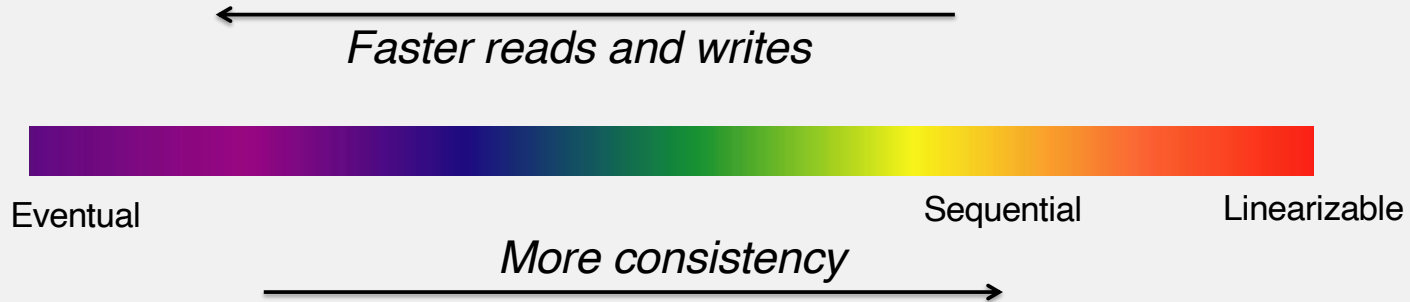
order in a way such that total order is a valid sequential one

Appears to be a single machine (one-copy semantics)

Trades off performance and availability



Consistency Spectrum



Linearizability is easy to program against, but can lead to low performance/availability

Eventual consistency is performant and leads to high availability, but hard to program against

There are a variety of options in between

Causal Consistency

No total order of operations

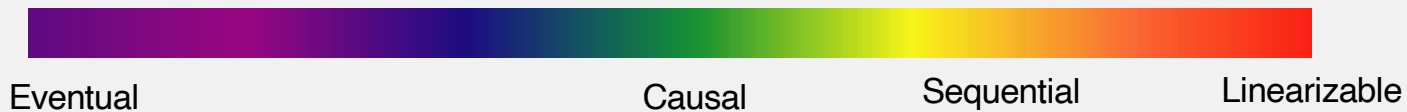
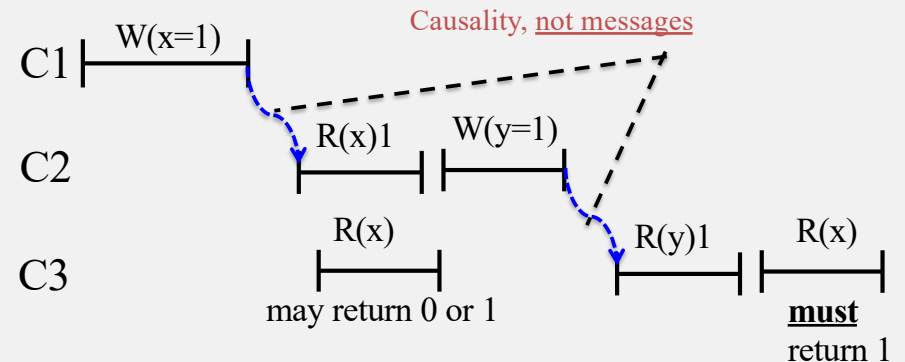
Avoids many inconsistencies that happen with eventual

Operations must respect partial order based on
information flow or causality

Operations from same client are causally related

If client C1 does a write and a read at client C2 sees it,
then these ops are causally related

Transitive



Session-based Consistency Models

[Terry et al. 1994]



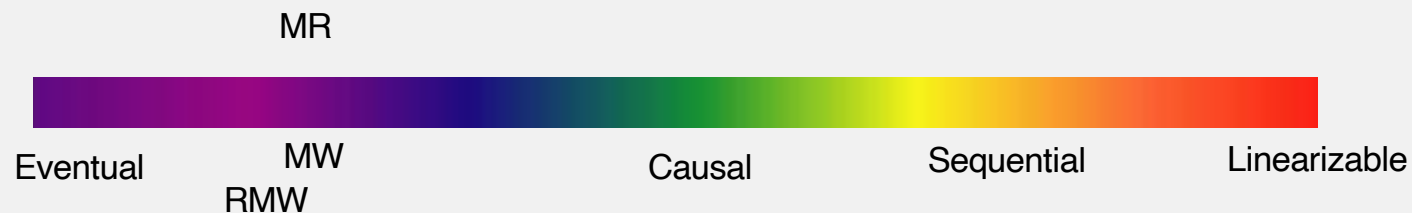
Does causal imply MR?

Monotonic reads (MR) – intuitively, reads cannot go back in time

if a client issues read R_1 and then R_2 , R_2 must at least observe the state as of R_1

Monotonic writes (MW) – if a client issues write W_1 and then W_2 , clients must see W_1 before W_2

Read my writes (RMW) – if a client issues W_1 and then R_1 , R_1 must see the effects of W_1



Newer Consistency Models

CRDTs (Commutative Replicated Data Types): Data structures for which commutated writes give same result [INRIA, France]

- E.g., value == int, and only op allowed is +1

- Guaranteed to eventually converge

- Effectively, servers don't need to worry about consistency and conflict resolution

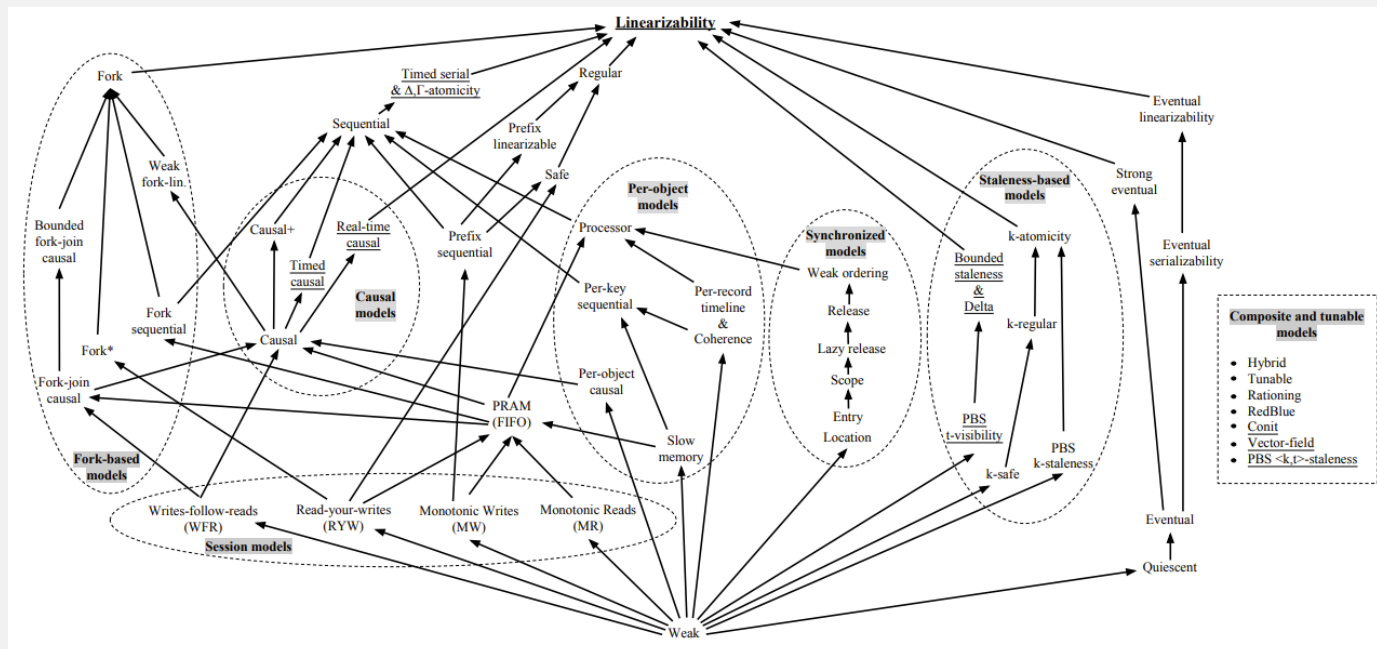
Red-blue Consistency: Rewrite client transactions to separate ops into red ops vs. blue ops [MPI-SWS Germany and others]

- Blue ops can be executed (commutated) in any order across DCs

- Red ops need to be executed in the same order at each DC

- Composite model: blue ops are eventually consistent and red ops are strongly consistent

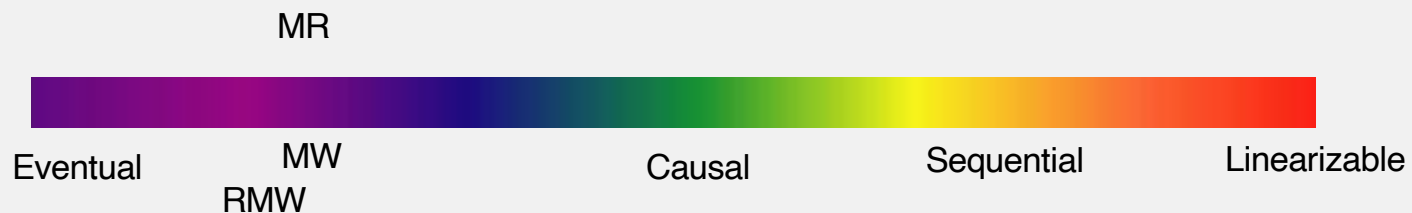
The Consistency Spectrum



[1] Viotti and Vukolic 2016

Which Consistency Model should you use?

- Use the lowest consistency (to the left) consistency model that is “correct” for your application
 - Gets you fastest availability



HBase

- Google's BigTable was first “blob-based” storage system
 - Yahoo! Open-sourced it → HBase
 - Still a major Apache project today
- Facebook used HBase internally for a while
- API functions
 - Get/Put(row)
 - Scan(row range, filter) – range queries
 - MultiPut
- Unlike Cassandra, HBase prefers consistency (over availability)
- Watch Video in course webpage

Local Storage at Each Replica

- Different ways of storing data: Big topic of file systems (423/523/598)
- Storage write performance: random vs sequential
- Historically, cloud-scale file systems have evolved into:
 - Append-only files, immutable once they're "closed"
 - Files are deleted eventually
- But workloads insert/modify/delete keys randomly across the key space
- Need data structure that can absorb random writes to keys
 - Log structured merge tree (LSM-tree)

Log Structured Merge-tree (LSM-tree)

On receiving a write

1. Record it on a write-ahead-log, aka WAL (for failure recovery)
2. Update **Memtable**
 - In-memory data structure to store many key-value pairs
 - Easily searched by key
 - Write-back cache (as opposed to write-through)
3. Eventually written to persistent storage
 - After which the log can be discarded
 - As new files with a specific format (e.g. SSTables)

Used by many distributed systems: Cassandra, Hbase, Spanner, etc.

LSM animation – see additional slides

Writes at a replica node

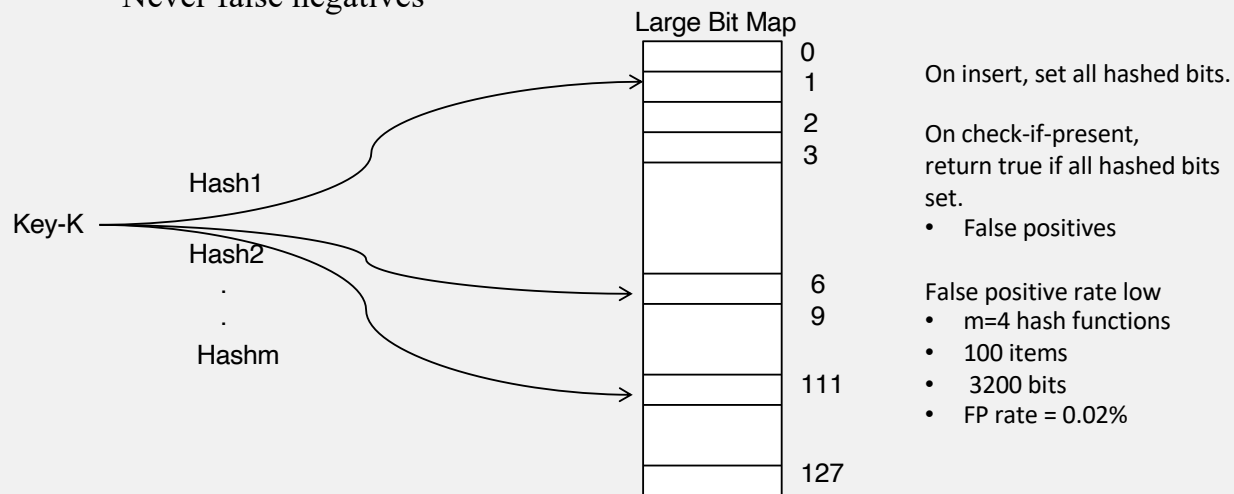
What's written to storage?

Whenever Memtable is full or old → flush to disk

- Data File: An **SSTable** (Sorted String Table) – list of key-value pairs, sorted by key
 - Other formats too
- *Each data file is immutable (once created, they don't change)*
- Includes
 - Indexes: key → location in data file
 - 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



Compaction

Data file (SSTables) accumulate over time:

- Consumes storage
- Many of the values of keys are stale; newer values have been written since
 - Another level of complexity: <key, value, timestamp>
 - System may want to retain some timestamped values
- Compaction merges data files: fewer values per key
 - Run periodically and locally at each server

Reads at a Replica



What if the false positive rate of Bloom filter is too high?

- Read consults Memtables first
- And then data files: from L0, L1, ...until Ln
 - Check if SSTable key range intersects
 - If yes, check Bloom filter
 - If no, repeat with the next level SSTable
 - If yes, search Index to get position in SSTable
 - If no, repeat with the next level SSTable (false +ve)
 - If yes, read actual KV pair from SSTable
- Read may specify a timestamp:
 - Youngest value of <key> before <timestamp>

Deletes

Delete: don't delete item right away

- Add a **tombstone** to the log and Memtable
 - <Key:Tombstone:timestamp>
- Eventually, when compaction merges values and encounters the tombstone it can delete the key

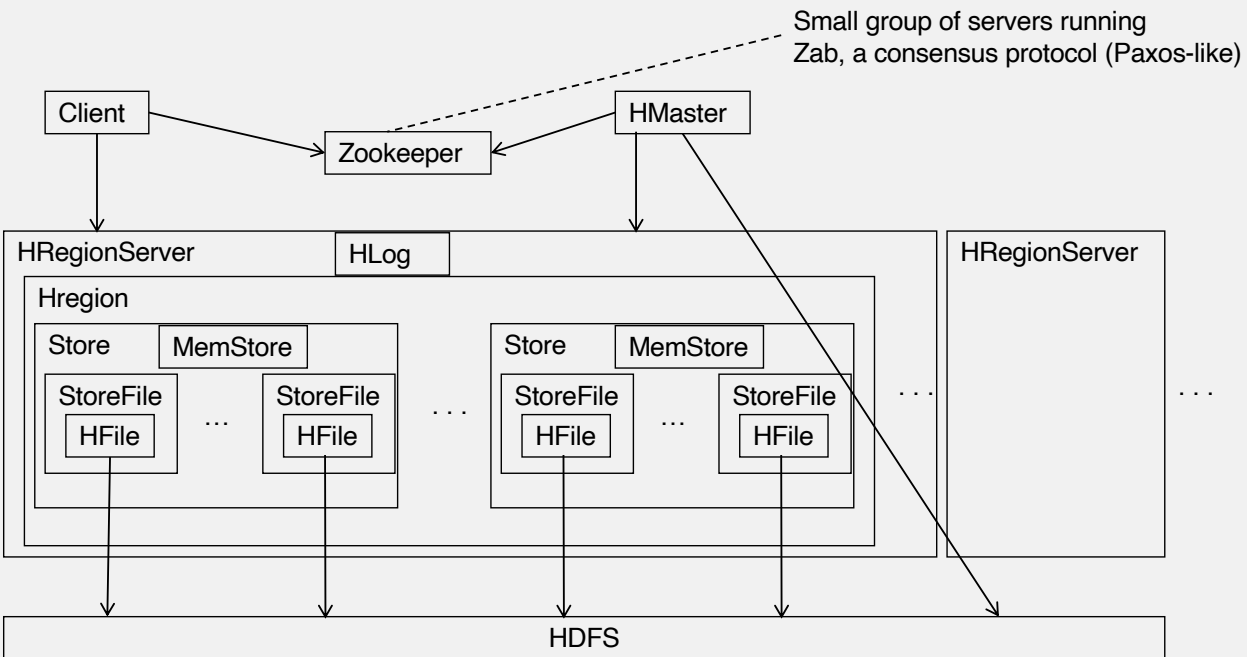
CS 425 / ECE 428
Distributed Systems
Fall 2026

HBase

HBase

- Google's BigTable was first “blob-based” storage system
- Yahoo! Open-sourced it → HBase
- Major Apache project today
- Facebook uses HBase internally
- API functions
 - Get/Put(row)
 - Scan(row range, filter) – range queries
 - MultiPut
- Unlike Cassandra, HBase prefers consistency (over availability)
- (Version of 2014)

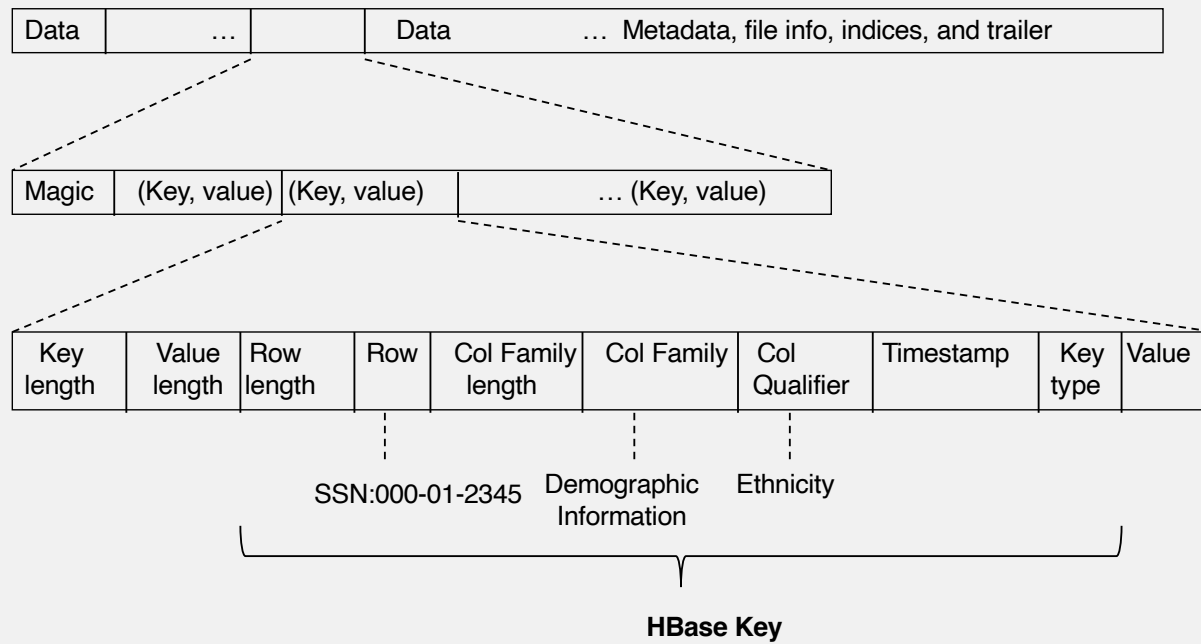
HBase Architecture



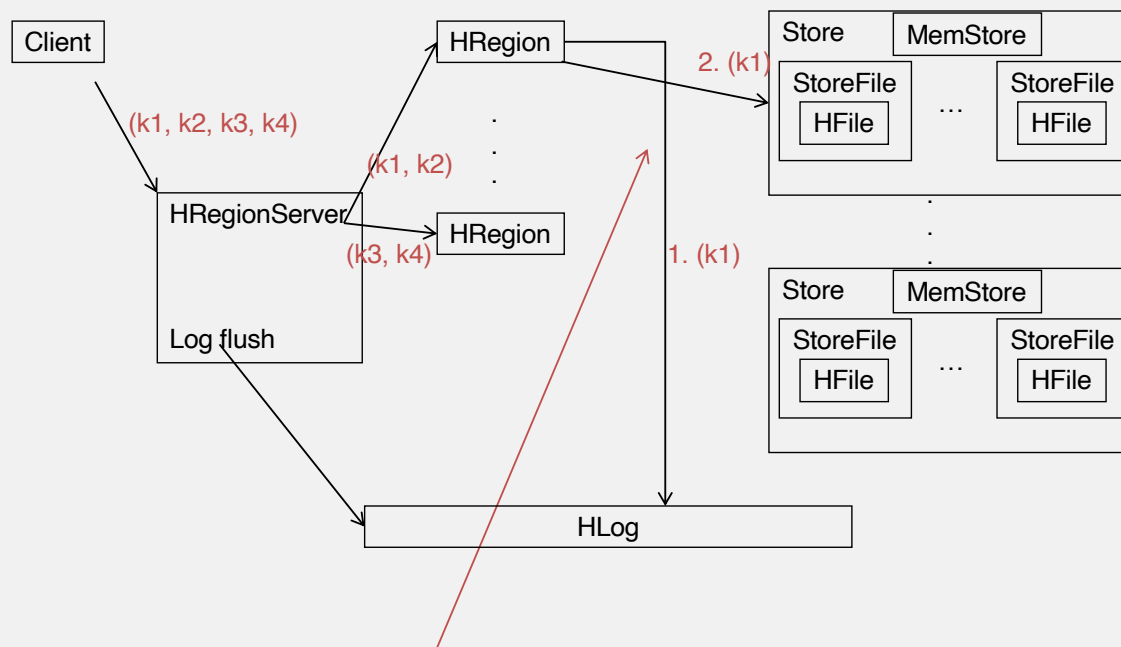
HBase Storage hierarchy

- 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

HFile



Strong Consistency: HBase Write-Ahead Log



Write to HLog before writing to MemStore
Helps recover from failure by replaying Hlog.

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 “Leader” (~~“Master”~~) cluster
- Other “Follower” (~~“Slave”~~) clusters replicate the same tables
- Leader cluster synchronously sends HLogs over to follower 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>*