Data-Intensive Distributed Computing CS431/451/631/651

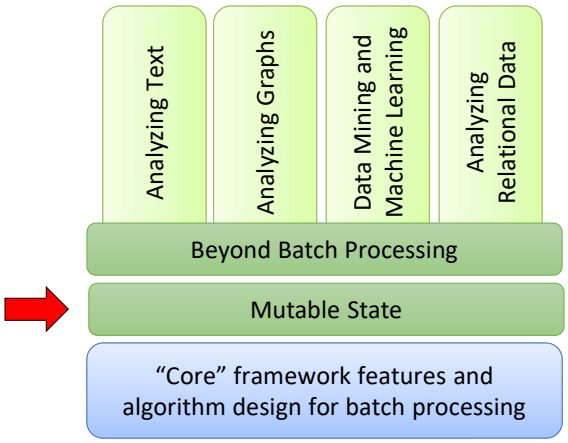
Module 9 – Mutable State

Visual : Change. Because with mutable state we can CHANGE the data. Ohohohohoho

Anyway, this is the last module that's on the final exam! It doesn't have a corresponding assignment so…I'll keep it simple on the exam.

I guess I could leave it out but then what's the point of coming to class this week???

Structure of the Course



When you're out of green bars, simply make more.

Mutable State

Until Now

Sequential Reads

Append-Only Writes

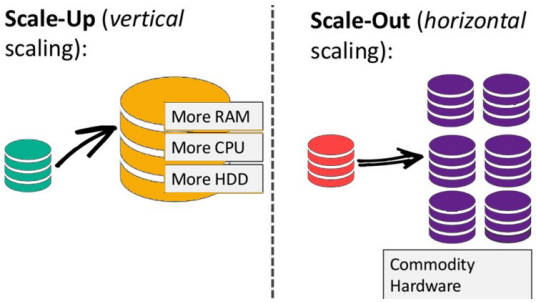
What About

Random Reads

Random Writes

In other words, what if we wanted a distributed database after all?

Why not RDBMS?



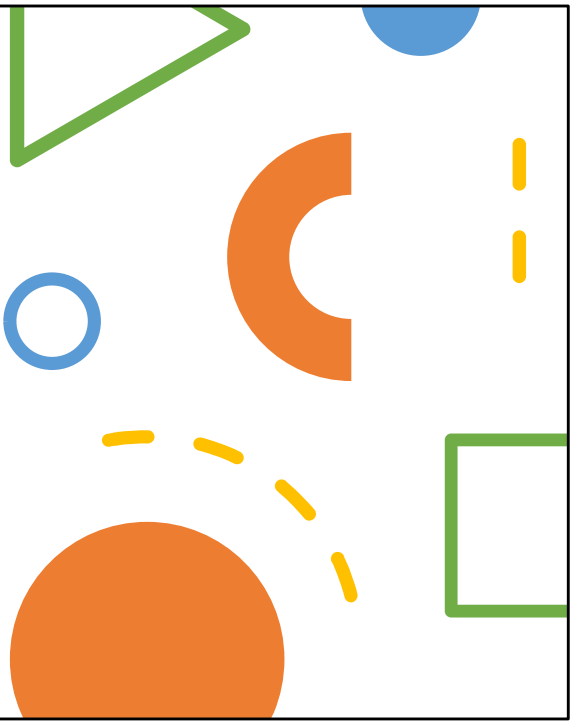
- Does not scale out → expensive
- Does not support semi-structured data

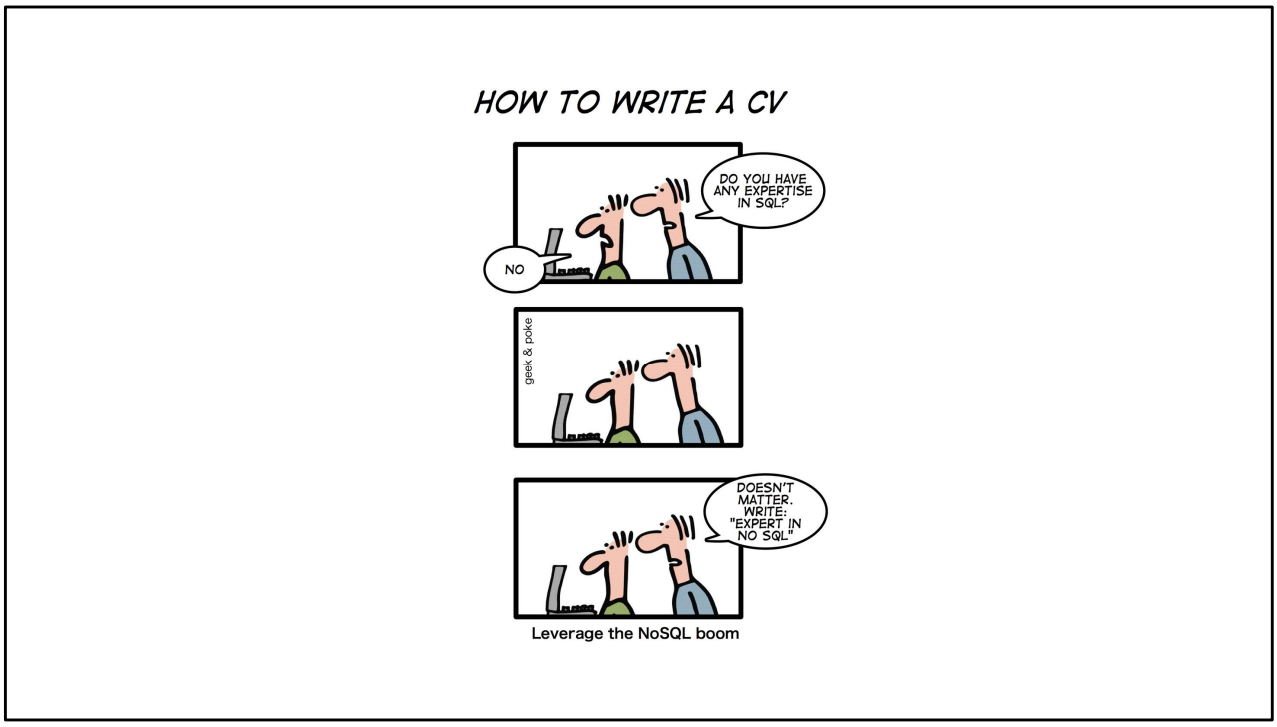
NoSQL

Not only SQL -- Doesn't mean there is no SQL! (There MAY not be)

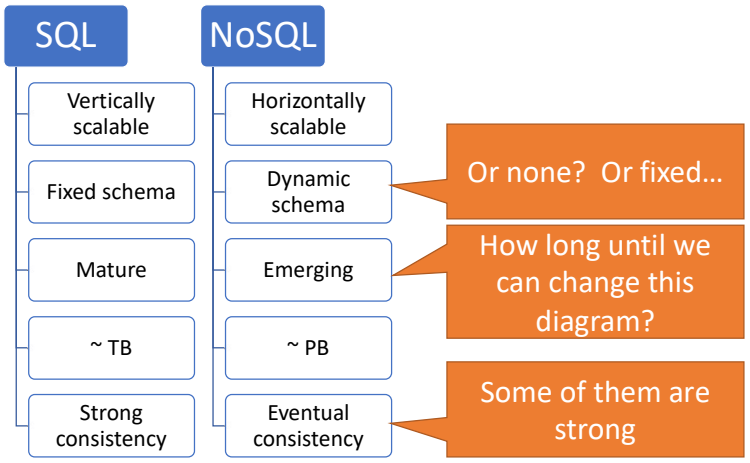
Common Features

- Horizontal Scaling
- Replicated and Distributed Data
- Weaker Concurrency (not ACID)
- Flexible Schemas




HOW TO WRITE A CV
DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"
Leverage the NoSQL boom

SQL vs. NoSQL



"They're more like guidelines than actual rules…"

Hadoop with Hive / SparkSQL is (mostly) NoSQL. Except, cannot update tables. Or, hmmm, can you?

SQL vs. NoSQL

Atomicity
Consistency
Isolation
Durability

ACID: The "Gold Standard" of RDBMS

Basically
Available
Soft State
Eventually consistent

BASE: The "good enough" of NoSQL

Basically Available? Like, yeah, within reason. Without your own reactors you'll never promise 100% availability.
Soft State – State might change even without a "triggering" input. (Because of the next part)
Eventually consistent. An update may result in inconsistent state…but it will eventually become consistent again.
(That's why the state has to be soft)

Types of Database

Relational (OLTP) Database
Analytical (OLAP) Database

SQL

Key-Value Stores
Column Stores
Document-Based
Graph-Based

NoSQL

An Analytical Database is not PER SE relational. The star schema makes a hypercube out of relations, but an OLAP database only needs hypercubes, so it doesn't NEED relations.

NoSQL Databases

Key-Value Stores

Memcached, Redis, Scalaris, etc.

Column Stores

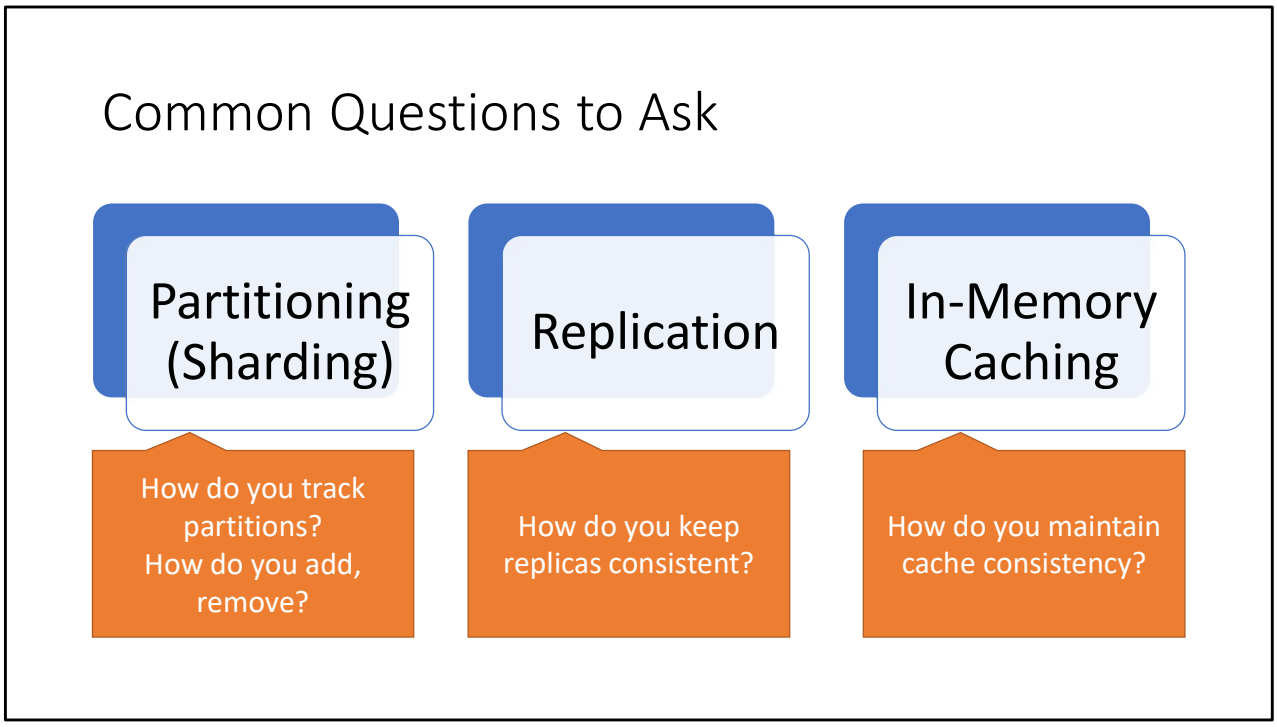
BigTable (Google), HBase (Apache), Cassandra (Apache), Hypertable (RIP 2016)

Document Stores

CouchDB, MongoDB, OrientDB, etc.

Graph Stores

Neo4J, InfoGrid, Flock DB



Oh, a fourth question, actually! What are the keys? What are the values?
The keys are (almost) always strings. What the values are varies by system…here, let's take a look

Fun "Fact" (that might not be true)

I refuse to Google it in case I'm wrong, but I'm PRETTY sure "Sharding" as a term for partitioning dates back to Ultima Online! As the first real MMORPG they could not handle all players on a single server like MUDs could, so they created separate "worlds". The in-game explanation was that a wizard did it (Mondain shattered the world crystal, and now each "shard" has a parallel version of Britannia).

True Fact: In highschool my friend Bruce cut class to play UO. He was a crafter. We joked he was skipping woodshop to make fake cabinets instead of real ones.

What they all have in common

Keys are Strings

Distributed / Clustered

Key-Value Stores

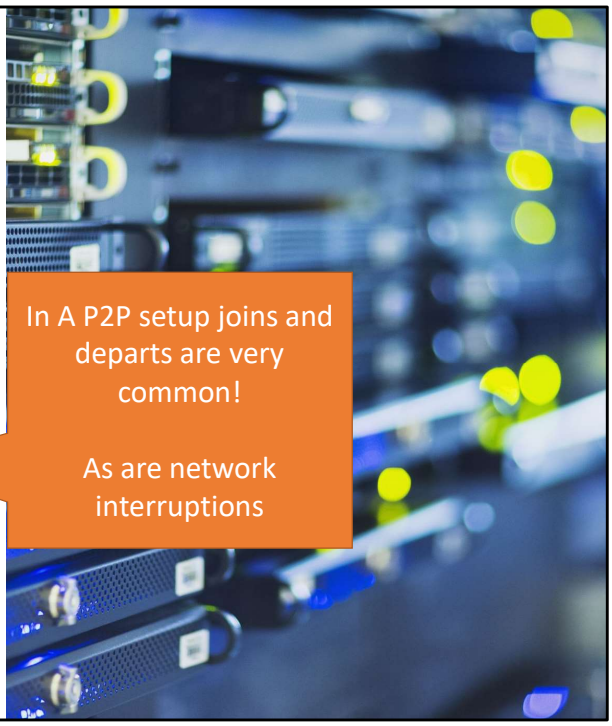
A Map is a Database. Maps keys onto values.

Queries: “What value does this key have?”

Keys in the Cluster

Problem: In your cluster of 400 servers, which has the key "X"?

- What if that machine is down?
- What if I add a new machine?
- What if I retire a machine?
  - (Unlike "down", this is permanent)



DHT – Distributed Hash Table

Consistent Hashing (Peer to Peer)

- Each Peer is assigned m-bit GUID by consistent hashing (e.g. SHA1)
- For each key *k*, it's assigned to the **first** node in the ring with a GUID at least *h(k)* – Same hash function as used to determine GUID
  - This is called *successor(k)*
  - Because we're looking for the first GUID *g* >= *k*

What's the benefit?

For K keys and N nodes

Depends on your definition of "high", and on hash code size

With high probability:

- Each node has at most $(1 + \varepsilon)K/N$ keys assigned to it

Therefore:

- When nodes connect/disconnect, only O(K/N) keys need remapping

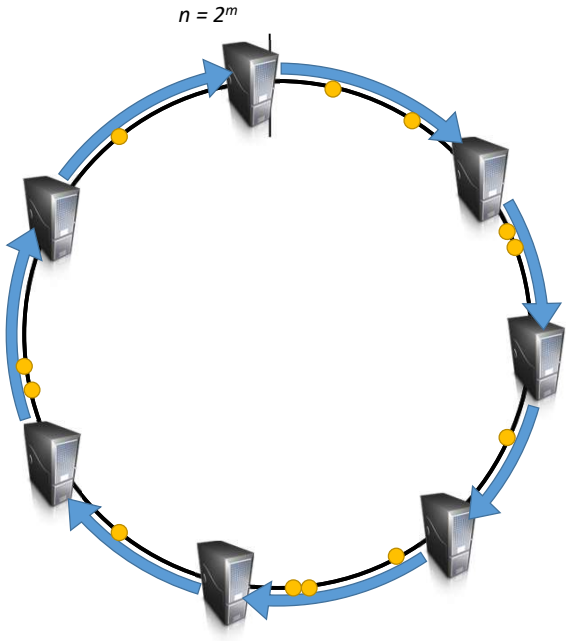
Chord distributed protocol

Stoica et al. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *SIGCOMM.*
And other resources …

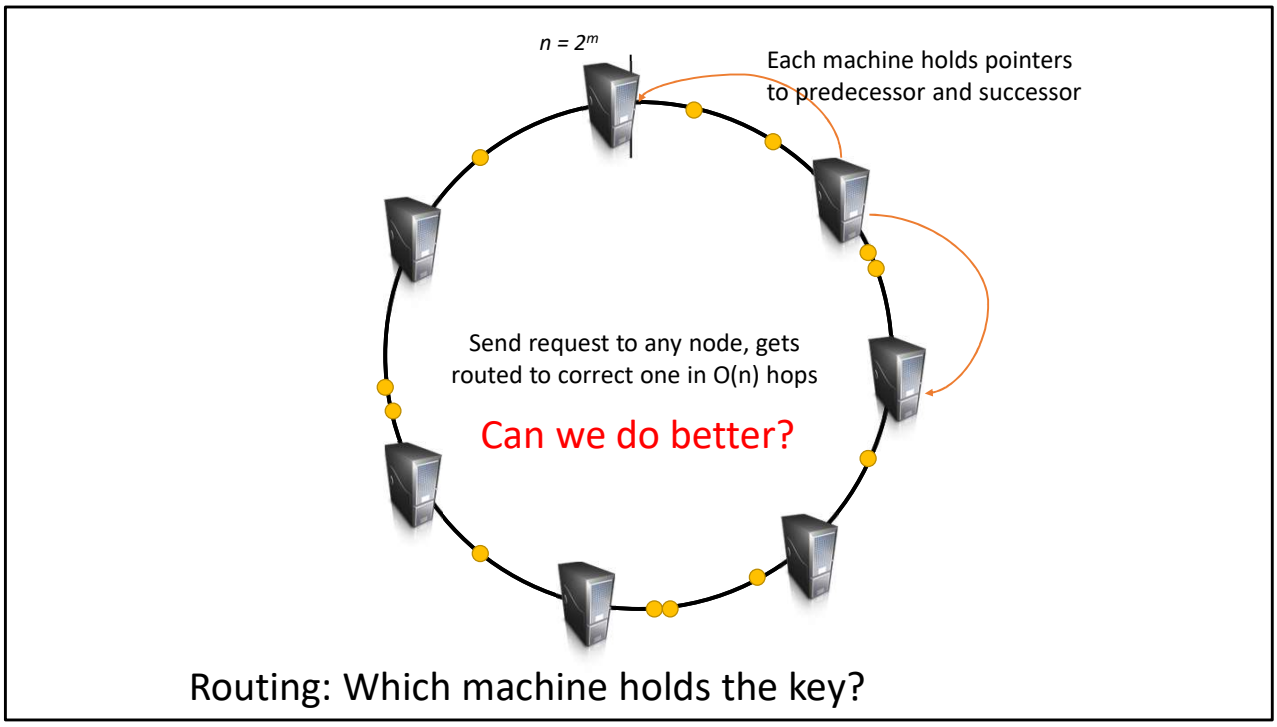
17

First Attempt:

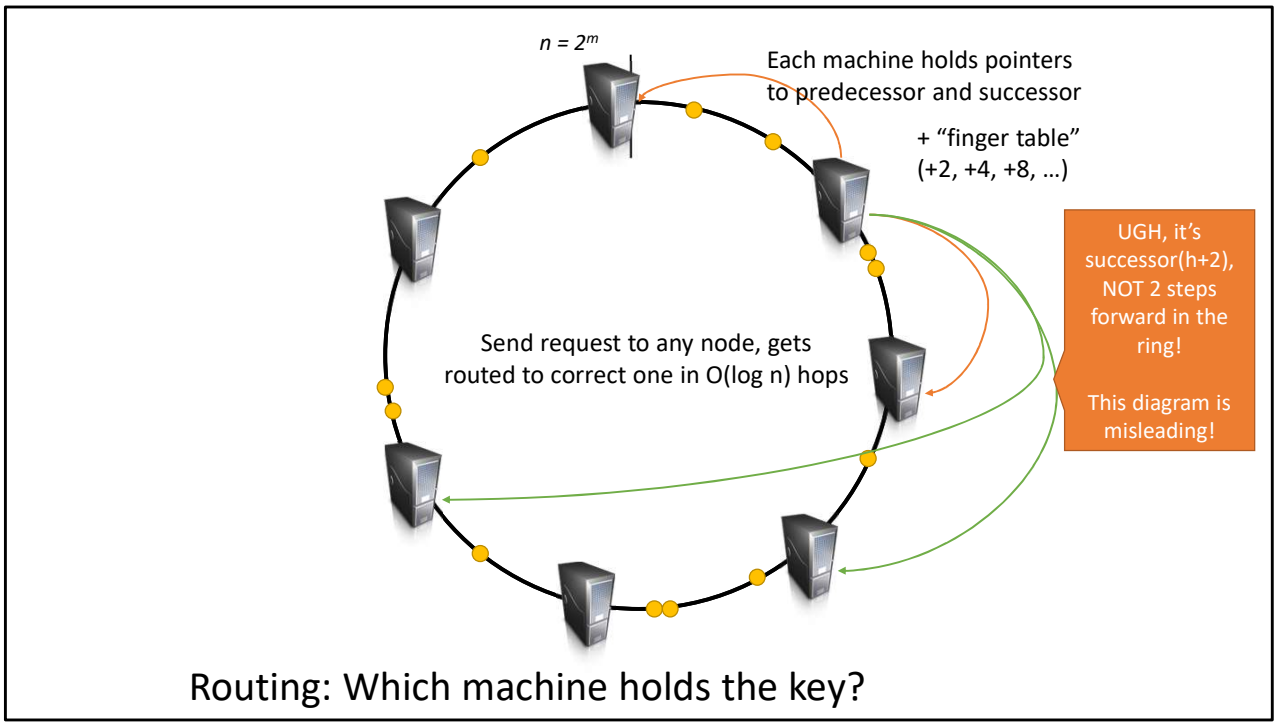
Each node has a pointer to its predecessor and successor



You might want more bits to avoid hash collisions, this is about minimums.



Routing: Which machine holds the key?



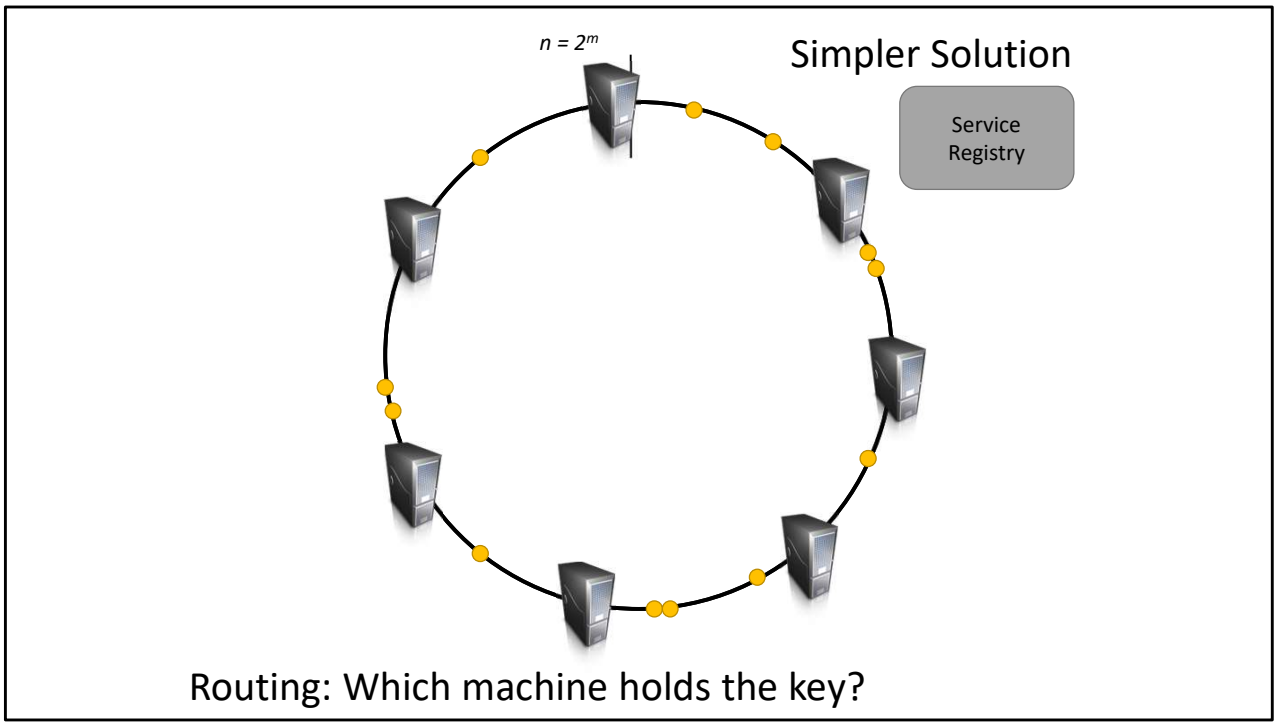
Routing: Which machine holds the key?

About that log(n) to find *successor(k)*

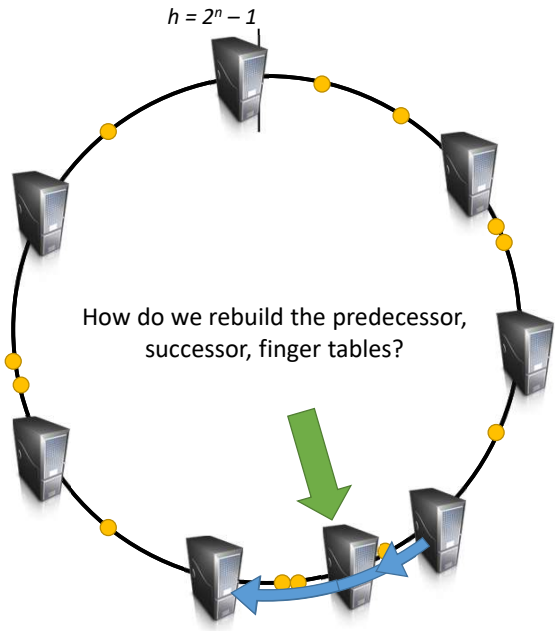
- Jump as far as you can without going over *k*
- Repeat until using the "one step" link – that's pointing to *successor(k)*
- Since working by powers of 2, will follow each "level" at most once
- There are $m = \log_2 n$ levels, thus, O(log n) jumps needed

Complication: Ring is mostly missing!

Solution: Rounding. Finger *i* doesn't need to point exactly 2^i steps away, just as close as you can get. In other words…it points to *successor*(self + 2^i)



DHT is designed for Peer-to-Peer where there is no central repository. If you want a central controller, you can simplify things…
Just ask the registry who is responsible for which key! O(1) messages



New machine joins: What happens?

New Node, Who's This?

Need to maintain two invariants

- Each Node (correctly) knows its successor in the ring
  - And, in fact, the whole finger table
- For every key *k* node *successor(k)* is responsible for k

New Node, Who's This?

New Node needs a finger table

- Naïve – run a query per entry – O(m log n)
- Better – query based on previous entry – $O(\log^2 n)$
- Best – Take successor's table and update only entries that might be wrong – O(log n)

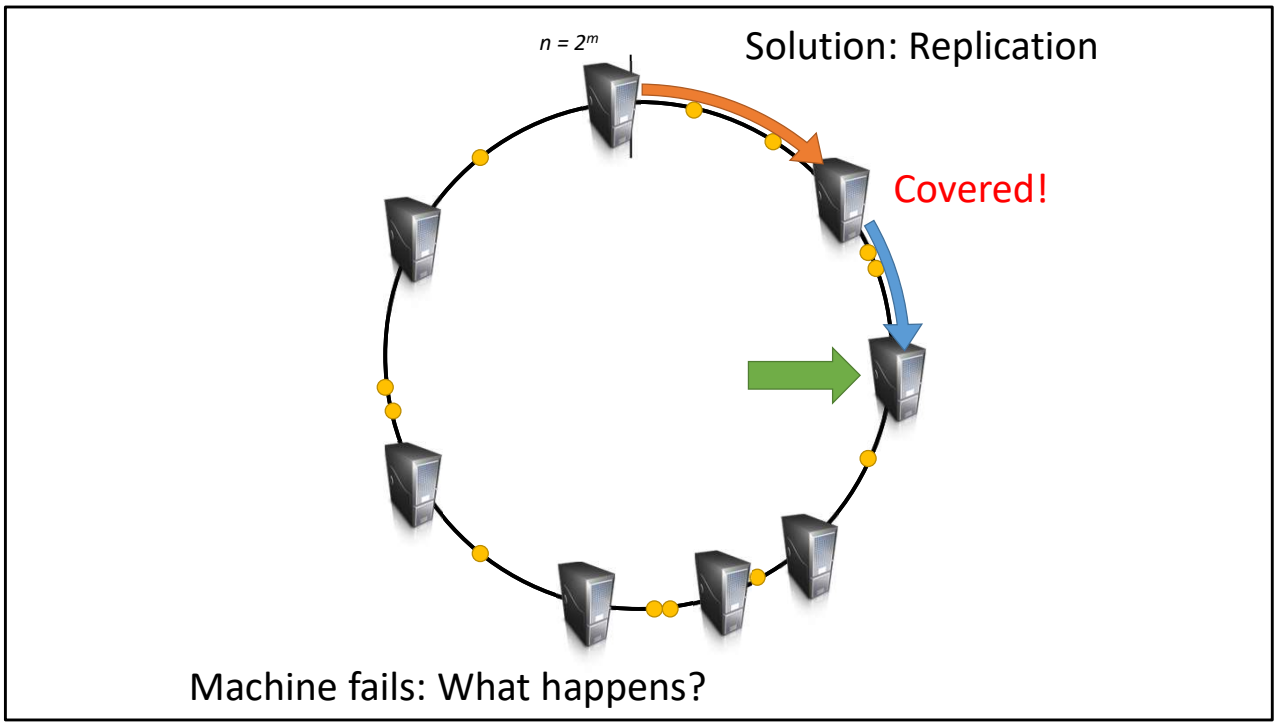
Let's not worry about the details, eh?

What about everyone else?

Nodes that were not contacted by the new one might have out-dated finger tables!

Solution: Stabilization process in the background updates finger tables periodically

Theorem: With high probability, adds (and removes) require $O(\log^2 N)$ messages to re-establish the invariant



How would you solve replication in this sort of pattern???

However…

Most NoSQL systems are not peer-to-peer. Joins and Leaves are not common.

So, none of the following use Chords for their DHT

Memcached

- Simple Distributed Key-Value Store
  - Value is arbitrary bytes. Treated as black box
- Distributed
  - Client hashes keys to pick server
- Disconnected
  - Shared Nothing. Servers do not know about each other
  - (No Replication)
- Not Persistent
  - Cache! “Forgetting is a feature”

Purpose: To cache things
If you want to keep it: Put it somewhere else, too!
Basically, if you’ve got free RAM on your cluster, make a Memcached cluster and use it to cache results. Might boost performance a lot. Might not.

Originally for caching dynamic webpages to avoid hitting the database

Finding your Key in Memcached

- Key x is held on h(x) % NUM_SERVERS

If a server goes down:

- It's content is gone forever. Like tears in the rain

If you add or remove a server from the cluster:

- oops, all cache misses

There is no "reshuffle" – It's only meant as a cache layer. If the number of servers changes, clients look to the wrong server.
It doesn't have the key, so it's fetched from the source. The server that used to hold that key will have its value expire eventually
(Again, it's a cache! The servers don't talk to each other so there's no way to coordinate a reshuffle even if that was desired)

Redis

- Calls itself a “Data-structure Server”
  - Value types include lists, hashes, sets, streams, HLL Counters
  - Extendable
- Distributed
  - Client hashes key to determine server
  - Redundancy / Replication
- Coordinated
  - Servers talk to each other
  - Efficient resharding
- Persistent
  - Stores everything in RAM, but persists to HDD

“red-kiss” without the k . REmote DIctionary Server

Purpose: Also good for caching things. Lets you update values with a wide variety of common data structures / operations
“Eventual” Consistency.

Scales up to 16K nodes!

Adding a new node: reassign buckets from existing servers to new one, trying to even the load
Server dies: buckets reassigned and recreated from replicas
Server getting full: reassign some of its buckets to other servers.

Hash-Tags. Keys with {hash-tags} only hash the tag. (Lets you ensure related keys are all on the same server)

Finding your key in Redis

Key x is held in bucket h(x) % 16384

Cluster coordinates who has what buckets! (With replication)

If a server goes down: Find the replica bucket

If a server gets added/removed: Reassign buckets to maintain balance

Oh dear, it can only scale up to 16384 servers???

(Well no, you can have replicas and load-balance…still, 16384 Redis nodes is extreme overkill)

Redis Stack

Adds a lot of bells and whistles!

Searchable / Queriable JSON documents as values

Secondary Indexing for Redis (or any NoSQL, or any RDBMS)

Those are just two of the main features

Document Stores

AKA “Semi-Structured Data Stores”

- Keys – Strings
- Values – JSON-like documents with optional schema
  - Schema is PER VALUE
  - Each document has its own, each document can be different!
- Allows indexing and queries
- Claimed to be ACID compliant
  - Isn't

Fun story – before it made a lot of news, the default security settings allowed anybody to access the system, and bound to all network interfaces – TL;DR: if you could find a MongoDB install you could probably take it over

Eventually changed the default to only bind to localhost – need to explicitly allow network access if you need it

- Keys – Strings
- Values – JSON Documents
  - Per document schemas (optional)
- Uses MVCC for Eventual Consistency
- Claims “Document-Level ACID Semantics”
  - Still not ACID
- MapReduce evaluation model (Javascript)
  - “Views” – Map-like construction
  - “Query” – Reduce-like aggregation

MVCC – Multi-Value Concurrency Control

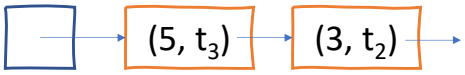
BORING

x | 3

x is 3, when you run x <- 5, now it's 5.

Just lock your table and it'll be consistent! (SNORE)

EXCITING



x is a linked list, sorted by time (descending order)

When you start a query at t_2 it looks backward for a time <= t_2

I made an In-Memory MVCC library for Racket…it was fun, which tells you something about my views on fun, I guess…
https://github.com/djholtby/versioned-box

Read/Write transactions will create a hash table for variables, and write values there!
When trying to commit the transaction: consult all keys in the "update" table. If they all have the same timestamp as they did when read from,
Then the transaction is consistent and can be committed! If not, it must be restarted from the beginning.

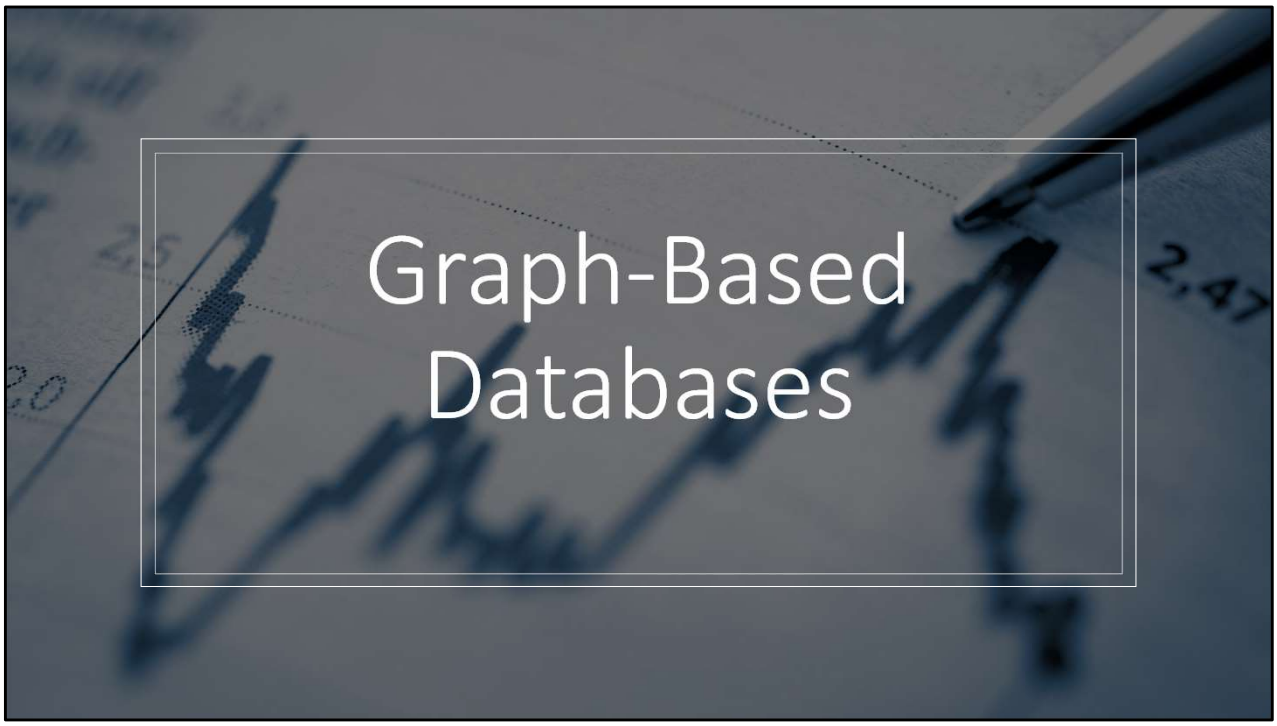
Read-only transactions are consistent (not necessarily consistent with NOW, but consistent with when they were initiated)
Write-only transactions are consistent (because they don't care about prior state)
Need a universal synchronized time though. CouchDB has per-document "revision numbers".

Free bonus: you can "see past versions" like it was a git repo! For free, it's part of how MVCC works! Of course you probably want to garbage collect if you're not PLANNING to have snapshots for every last change. (My library prunes old versions if there are no active transactions older than the timestamp. Easy on Racket, just set the next link to null and the

garbage collector will see the tail as unreachable and collect it)

Most modern databases use MVCC instead of locks, but then hide from the user as an unimportant detail. Perhaps it is?
An MVCC system cannot be distinguished from a lock-based system…if you allow for “maybe this query was instant but the network was slow” ;)

Basically if you start a query transaction, read some tables, and that’s the end, it behaves exactly like your query ran atomically and instantly the moment it was received, even if it took 2 minutes to run and even if the data changed during those 2 minutes. That’s ACID for you.



Silly powerpoint, not that kind of graph! Oh well…

Neo4J

- Stores Graphs
- Everything is a Node, Edge, or Attribute
  - Attributes are applied to nodes and/or edges
- ACID compliant transactions
- Commercial, with open-source “community edition”

I’ve never used a graph-based DB but have always been tempted…
Seems like it would work well for a MUD…rooms are nodes, mobs, PCs etc are attributes that move around the graph…


(Wide) Column Stores

Google BigTable

- Maps 2 strings (row key, column key) and timestamp to an arbitrary byte stream
  - Black Box, doesn't matter what the value is
- Example Use Case:
  - Row Key – URL
  - Col Key – Attribute of a website (content, metadata, etc)
  - Now you've got a way to store snapshots of the web
    - Gives your robots somewhere to put things while they're crawling
- Has Bloom filters at the row and row+col level

If you've forgot, a bloom filter is a probabilistic data structure for testing whether something exists

In the example: row level => "Have we seen this URL?"
Row+col level => "Do we have this attribute for this URL?"

A "no" is always correct. A "yes" can be incorrect

HBase

- HBase is to BigTable as Hadoop is to Google's MapReduce
- Part of the Hadoop Ecosystem, in other words!
  - Backed by HDFS
  - Hadoop Jobs can read from / write to HBase tables

Hi, I'm an Orca, not an elephant! How unusual for something built on Hadoop!

Apache: Hey can I copy your homework?
Google: Sure, just change it a little so it doesn't look obvious
Apache: Sure thing!

MapReduce

GoogleFS

BigTable

What's Google Use It For?

- Gmail
- Webcrawling
- Google Earth
- Google Analytics
- MapReduce (read from and write to)

Data Model

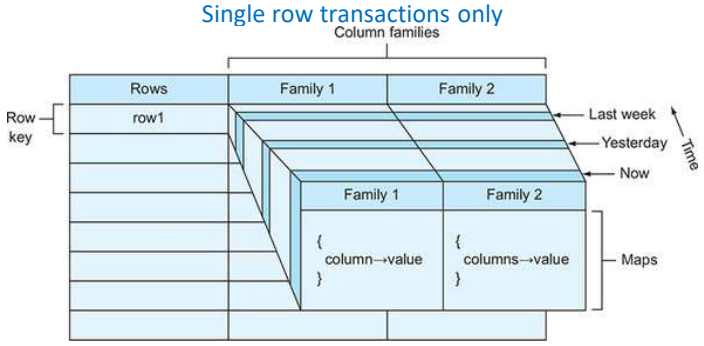
A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map

Map indexed by a row key, column key, and a timestamp

(row:string, column:string, time:int64) → uninterpreted byte array

Supports lookups, inserts, deletes

Single row transactions only



https://livebook.manning.com/book/google-cloud-platform-in-action/chapter-7/40

These next few slides are all about BigTable, but Hbase is basically identical

Rows and Columns

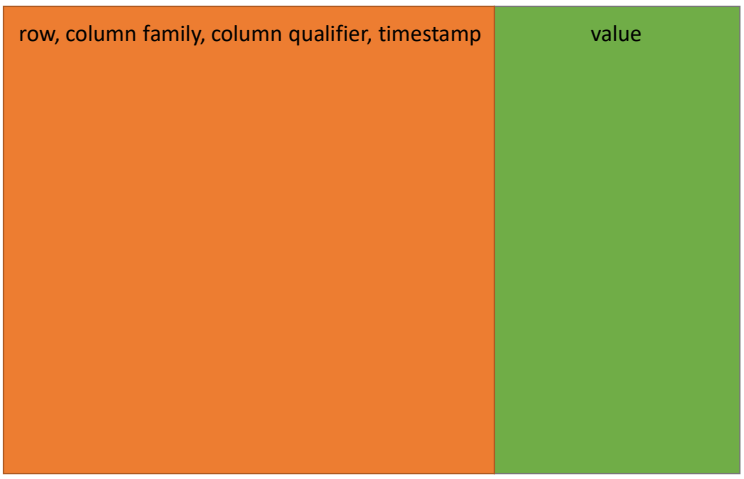
Rows maintained in sorted lexicographic order

- Applications can exploit this property for efficient row scans
- Row ranges dynamically partitioned into tablets

Columns grouped into column families

- Column key = family:qualifier
- Column families provide locality hints
- Unbounded number of columns

Key-Values

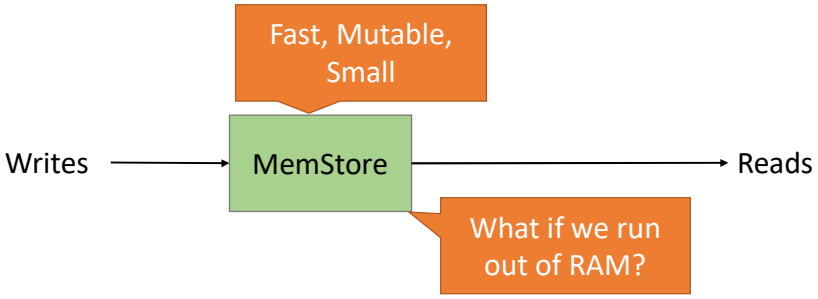



Always has been...
Wait, it's all Key-Value Pairs?
NoSQL

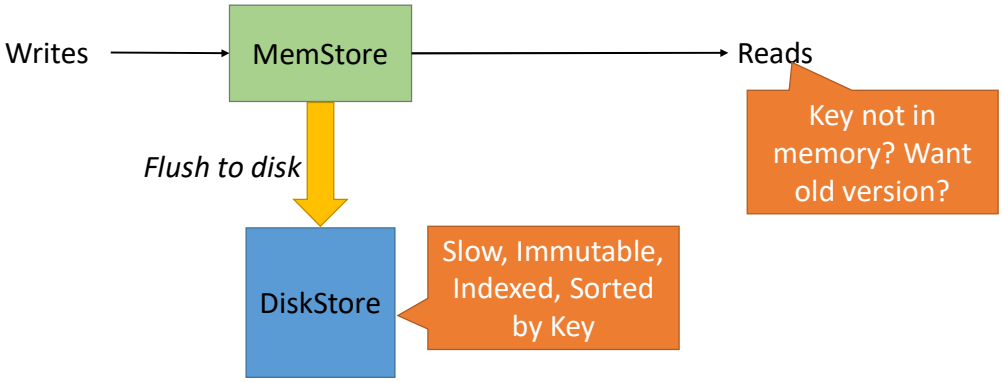
Okay, so how do we build it?

In Memory	On Disk
Mutability Easy	Mutability Hard
Small	Big

Log Structured Merge Trees (LSMT)

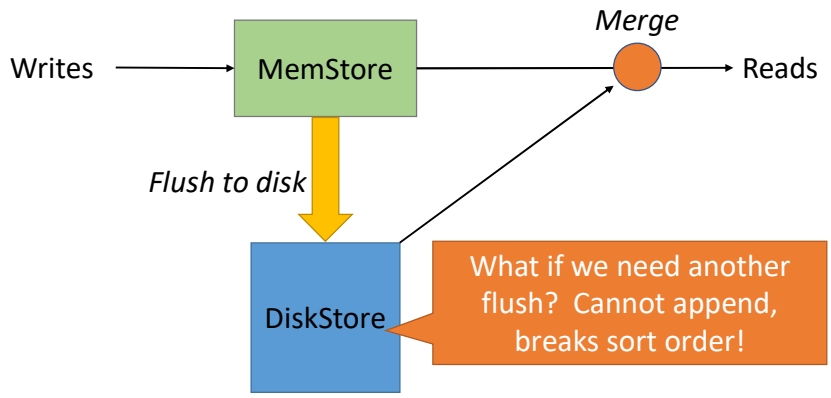


Log Structured Merge Trees (LSMT)

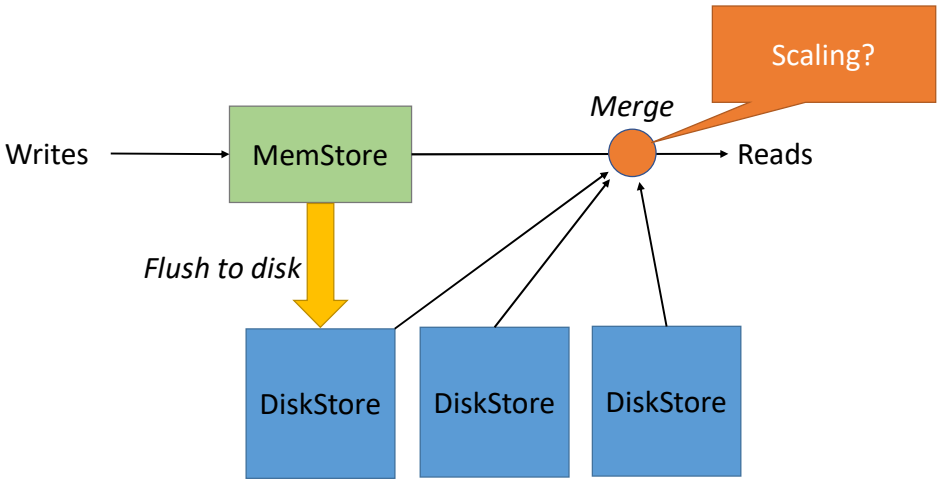


What if there's no new version of a row+col key in memory?
What if there is, but our transaction is looking for an older version?
Reads need to come from both sources.

Log Structured Merge Trees (LSMT)

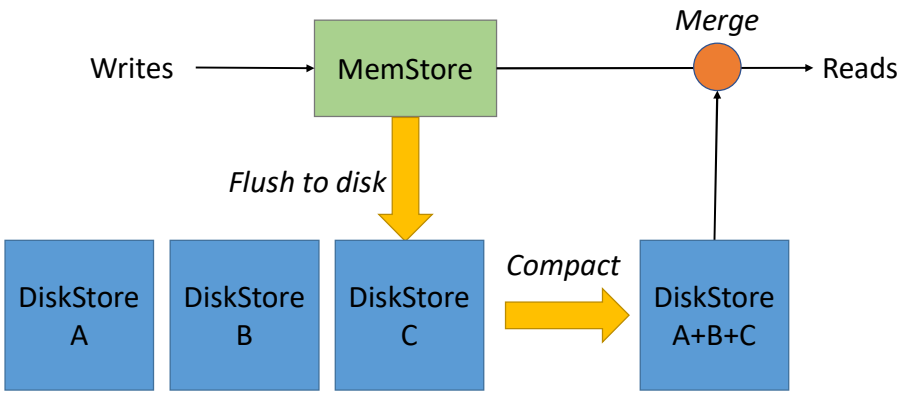


Log Structured Merge Trees (LSMT)



Surely we don't want to have to do a 1000-way merge each time we read???

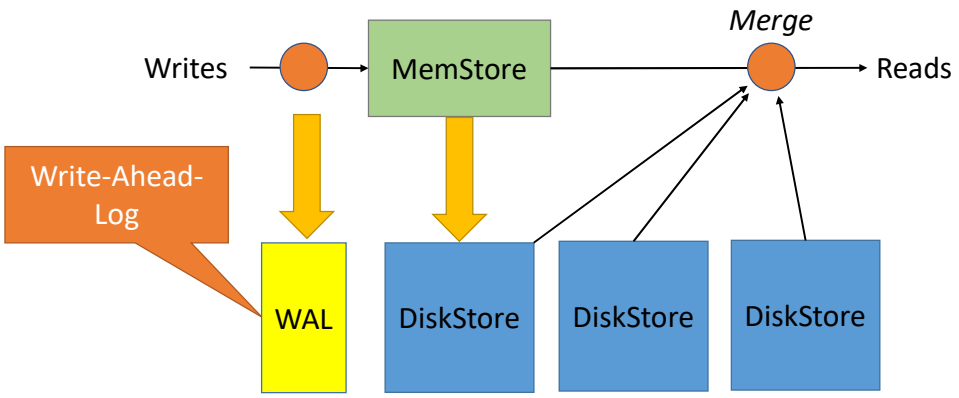
Log Structured Merge Trees (LSMT)



They're sorted, so reads are using "Merge" from Merge-Sort. We can use the same algorithm to merge multiple DiskStores into a single one.
Sound familiar?

(It should, it's what MapReduce does for its intermediate files, too! A lot of the tech is the same)

Log Structured Merge Trees (LSMT)



All writes are first logged, then updates applied in memory. Old writes are periodically spilled to disk.
WAL is a common DB thing.

In event of DiskStore loss – WAL can be replayed to reconstruct lost data

Replication: Forward WAL to other nodes, they can apply the writes to their replica! This is how Postgres replication / snapshotting works

That's one Machine. Cluster?

Building Blocks for Hfile (BigTable):

HDFS	(GFS)
HFile	(SSTable)
Region	(Tablet)
Region Server	(Tablet Server)
Zookeeper	(Chubby)

HFile (SSTable)

The Disk Store from before – Key-Value Pairs, sorted by key.

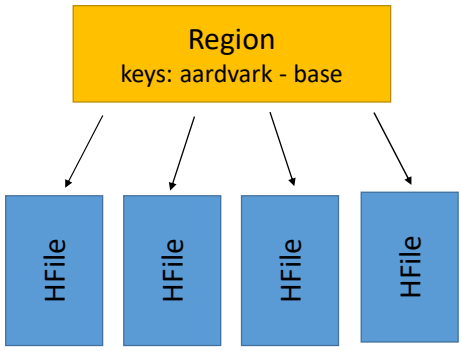
Immutable. (Because it's stored to HDFS / GFS)

Each node has its own.
Replicate for redundancy

Free! (HDFS)

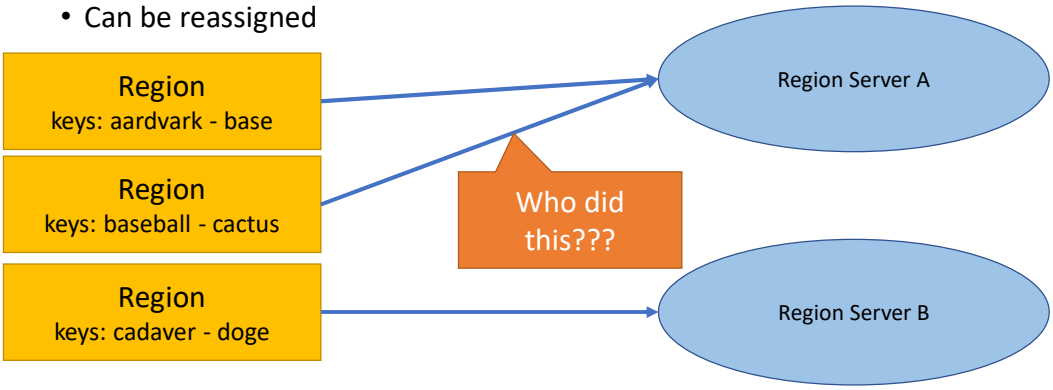
Region (Tablet)

- A Partition of Rows within one Table
- Dynamic!
- One Region : Many HFiles



Region Server (Tablet Server)

- Each Region gets assigned to ONE Region Server at a time
  - Can be reassigned



Not Shown – Each Region Server also acts as the secondary server for other regions. A secondary region server cannot write, only read. (And the reads might contain stale data that hasn't been replicated from the primary server)

HMaster (BigTable Master)

- Assigns Regions to Region Servers
- Adds and removes Region Servers
- Load balances Region assignments
- Garbage Collection
- Handles Schema Changes
- Handles (Some) Structural Changes
  - Table Creation / Deletion
  - Region Merging

What about Region Splits?

Region Splits are initiated by Region servers. When the region they're responsible gets too big, they'll split it into two ranges and ask the HMaster consider rebalancing Region assignments.

HBase – Big Picture

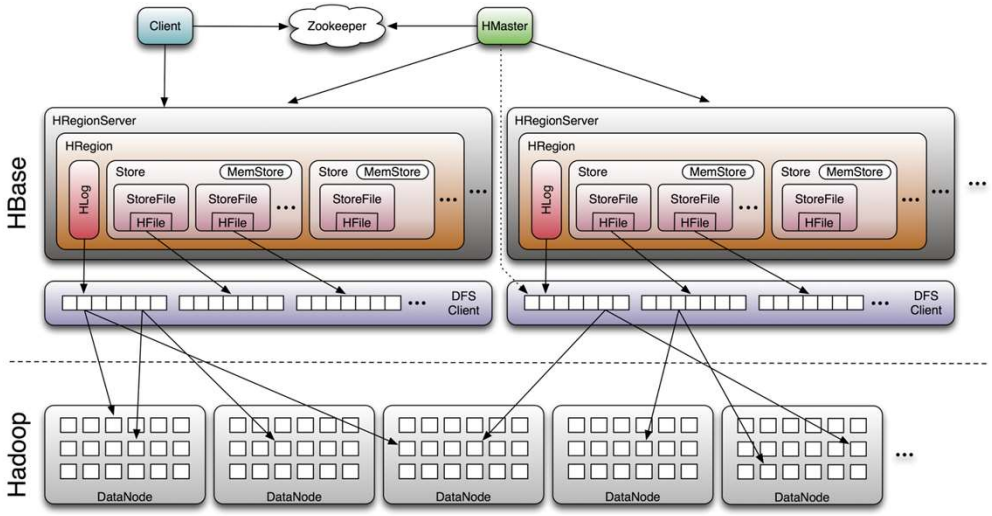


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

The zookeeper cluster is there to stop a bunch of annoying clients from pestering the HMaster all the time


Consistency

Why do we care about consistency

Alice transfers $100 to Carol, and Bob transfers $50 to Carol

- The total amount of money must remain the same
- Bank mad if money is created
- Customers mad if money is destroyed

Bob removes an RTX4090 from his shopping cart because his fire insurance won't cover it

- Clicks another page and it's back again
- Removes again
- Still there
- Gets an error about removing something that doesn't exist

Why do we care about consistency

Doug posts pictures from his vacation on Facebook

- First he changes his gallery settings so his mom can't see the gallery
- Then he posts the picture
- Oh no! His mom can still see his embarrassing photos! Scandalous!

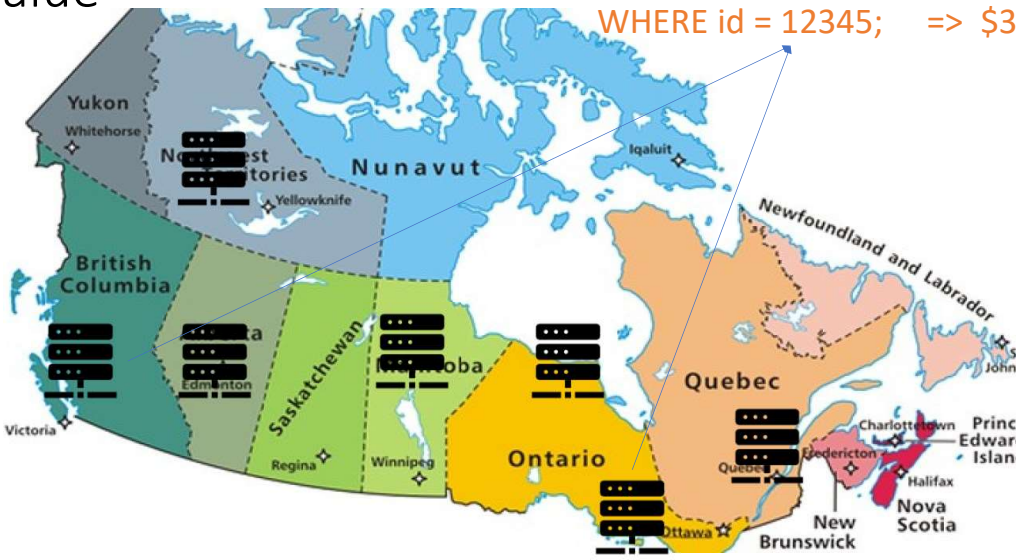
Eve unsubscribes from Piazza emails, then posts in the "say hi" thread

- Eve still gets notifications every time someone replies
- It doesn't stop for several hours! There are almost 1000 students taking CS135 this term!
- Based on a True Story™

We deeply regret this turn of events and after the 225% term do not have a "say hi" post as part of A0

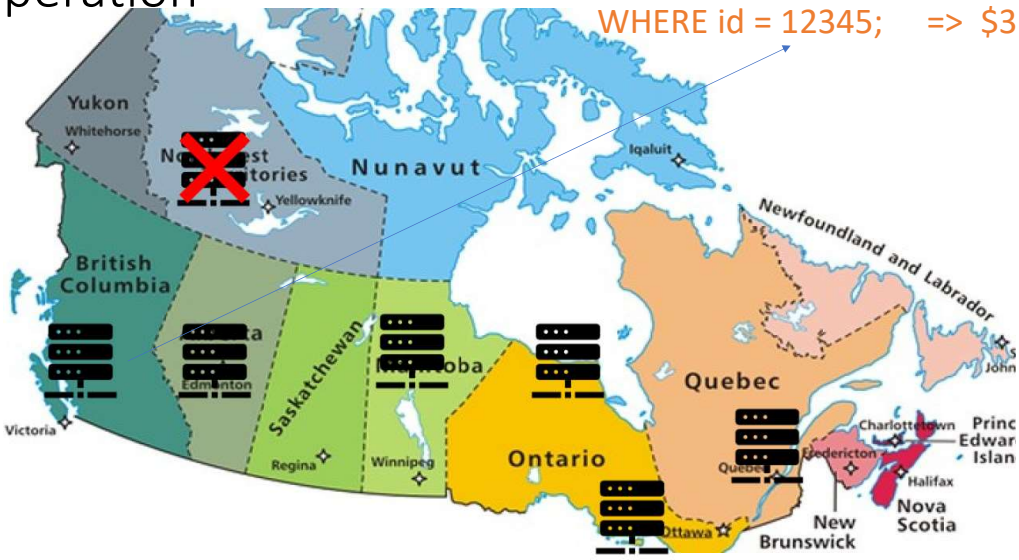
Consistency – All Nodes Return the Same Value

SELECT balance FROM account
WHERE id = 12345; => $3.50

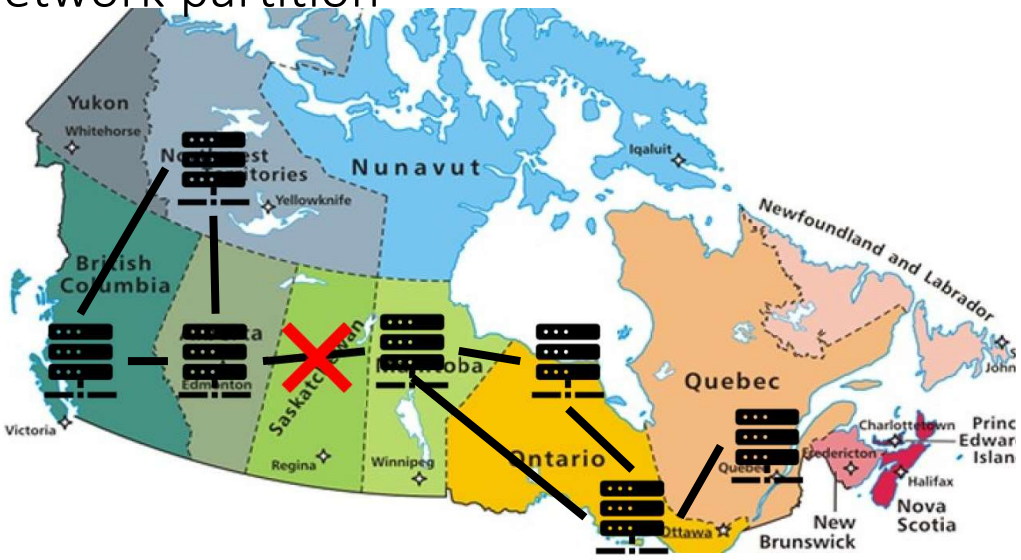


Availability – Node failures do not prevent operation

SELECT balance FROM account
WHERE id = 12345; => $3.50



Partition Tolerance – System operates despite network partition



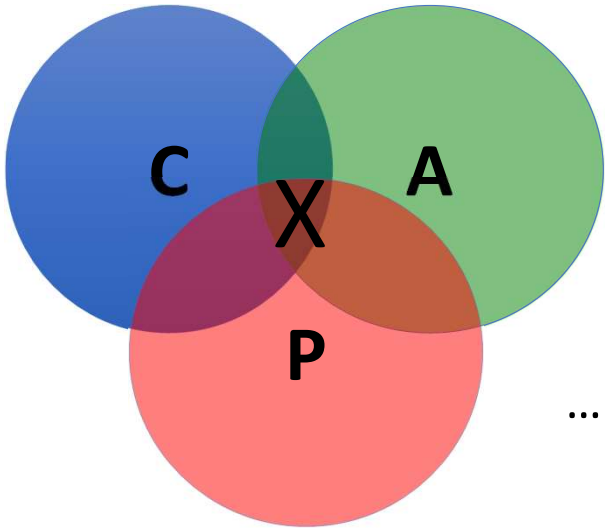
Even if the link goes down, the network still operates in two partitions, and customers in Vancouver and Toronto are both unaffected (other than not being able to connect to each other, if that's relevant)

CAP Theorem

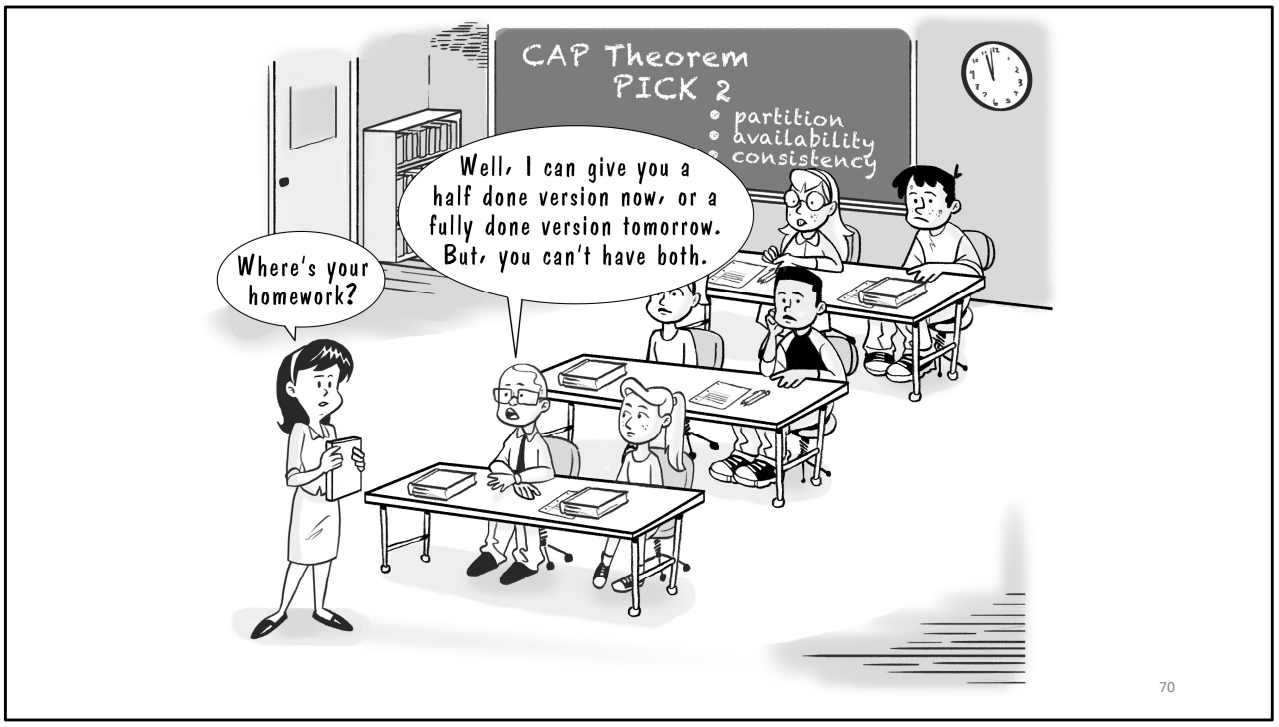
- **C**onsistency
  - All node see the same values at the same time
- **A**vailability
  - If a node fails, the network continues to operate
- **P**artition-Tolerance
  - The system operates despite network partitions

Theorem: Pick 2. **You cannot have all 3**

CAP Theorem



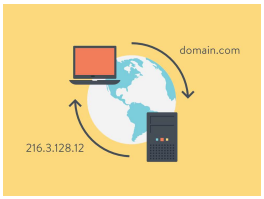
… pick two

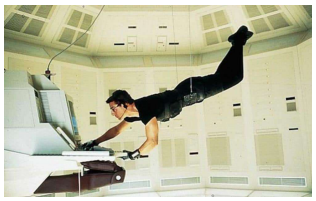

CAP Theorem
PICK 2
partition
availability
consistency
Where's your homework?
Well, I can give you a half done version now, or a fully done version tomorrow. But, you can't have both.
70

CAP Theorem

- This suggests there are three kinds of distributed systems:

- CP: Big Table and Hbase
- AP: DNS
- CA: Impossible in distributed systems





Impossible? You said pick any two!

CAP theorem is misstated…it actually means “when faced with a network partition, do you stay available, or stay consistent?

Impossible?

- CAP Theorem is misstated. It's more like:

"If there is a network partition, does your network remain **A**vailable or **C**onsistent? You can only choose one"

CAP Theorem: Impact

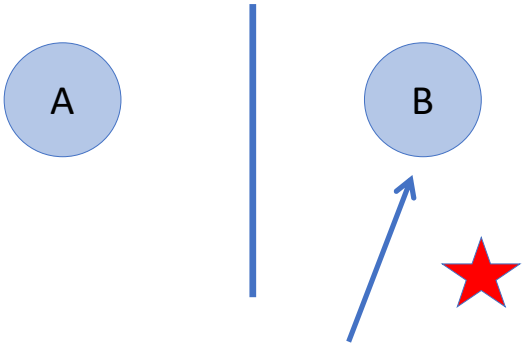
Divides the database community (even today)

SQL
Correctness above all

NoSQL
Availability above all

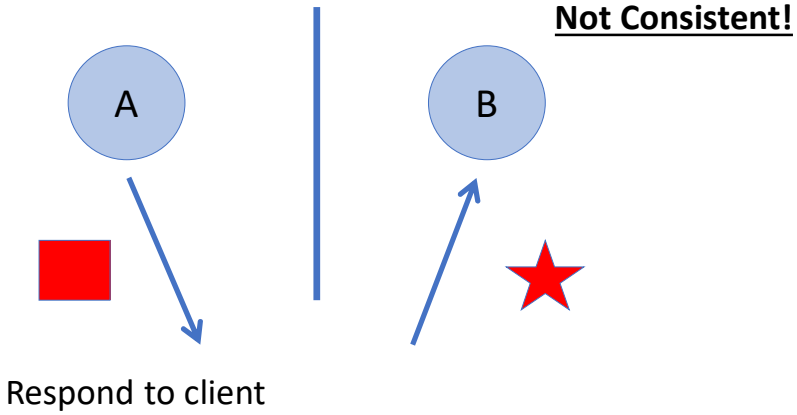
CAP Theorem: Proof

- A simple proof using two nodes:



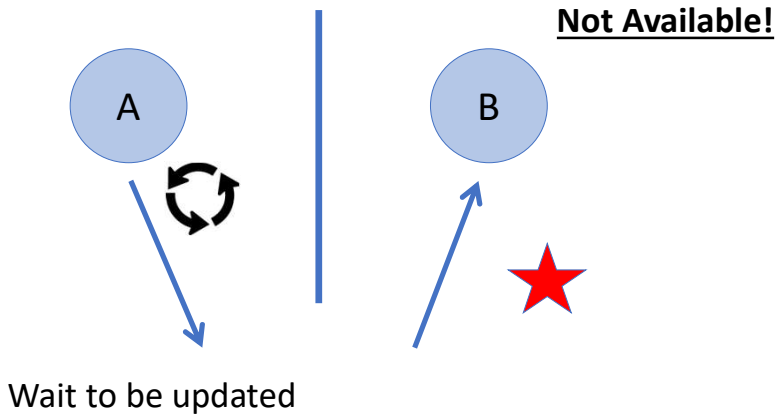
CAP Theorem: Proof

- A simple proof using two nodes:



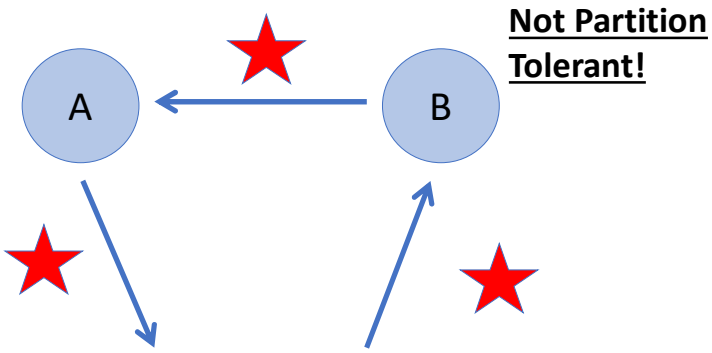
CAP Theorem: Proof

- A simple proof using two nodes:



CAP Theorem: Proof

- A simple proof using two nodes:



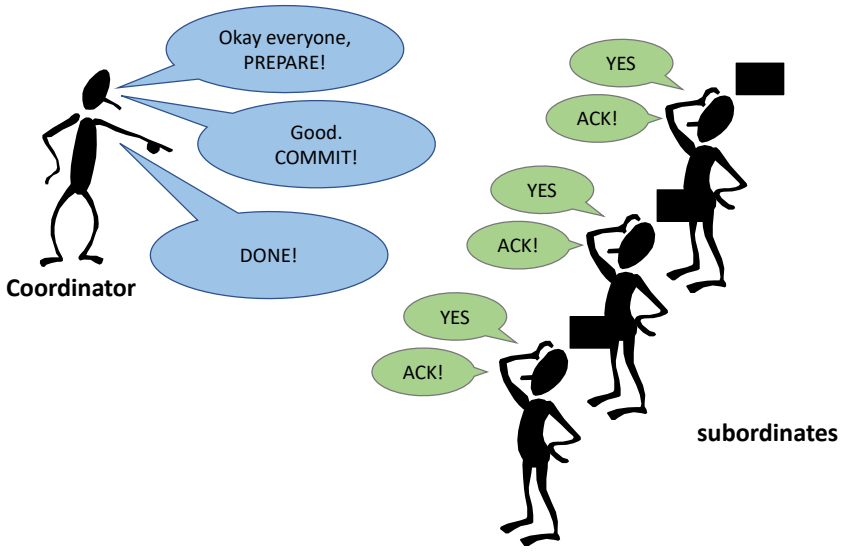
It's easy to be CA if partitions can't happen! But if they do, you can't be both.

Types of Consistency

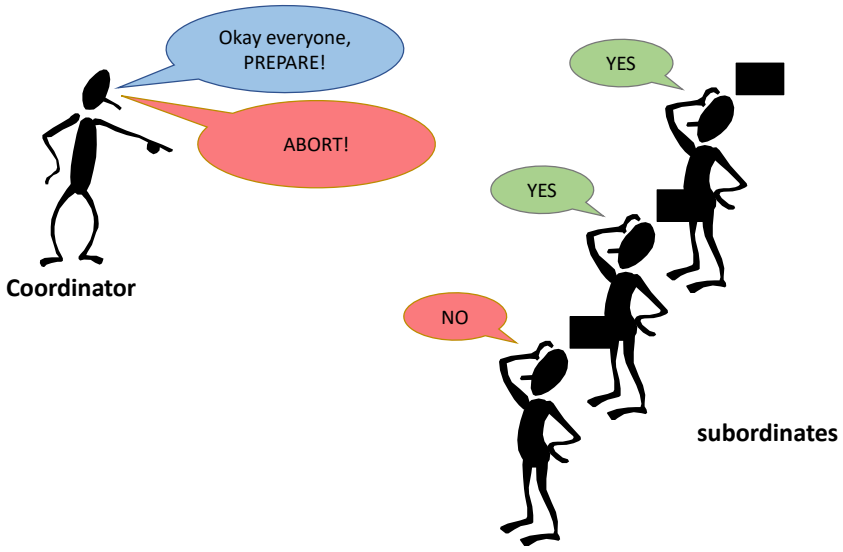
- Strong Consistency
  - After the update completes, **any subsequent access** will return the **same** updated value.

We even have meme consistency!

2 Phase Commit: Sketch



2 Phase Commit: Sketch



2PC: Assumptions and Limitations

Assumptions:

Persistent storage and write-ahead log at every node

WAL is never permanently lost

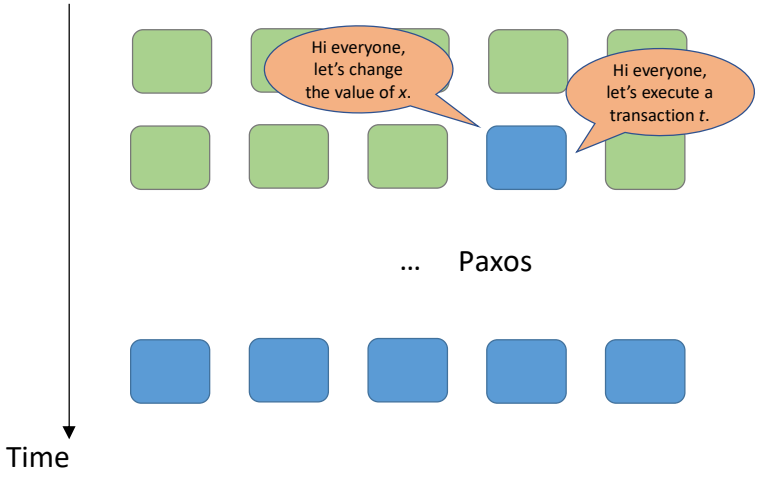
Limitations:

It's blocking and slow

What if the coordinator dies?

Distributed Consensus

More general problem: addresses replication and partitioning



Assumption: Faulty nodes fail arbitrarily, so you should assume worst case behavior, HOWEVER, faulty nodes do not COLLUDE. (If they do, that is Byzantine Agreement, a harder problem than Consensus)

Paxos Algorithm

- 3 Types of Nodes
  - **Proposers** advocate for client requests to change a value
  - **Acceptors** decide whether to accept a change
  - **Learners** remember values
- Acceptors are grouped into Quorums
  - Every Quorum must contain a **majority** of all acceptors
  - No two Quorums can be disjoint

A Machine might be all 3 types of node!

PowerPoint tells me readers won't know what majority means, and I should say "most of the acceptors".

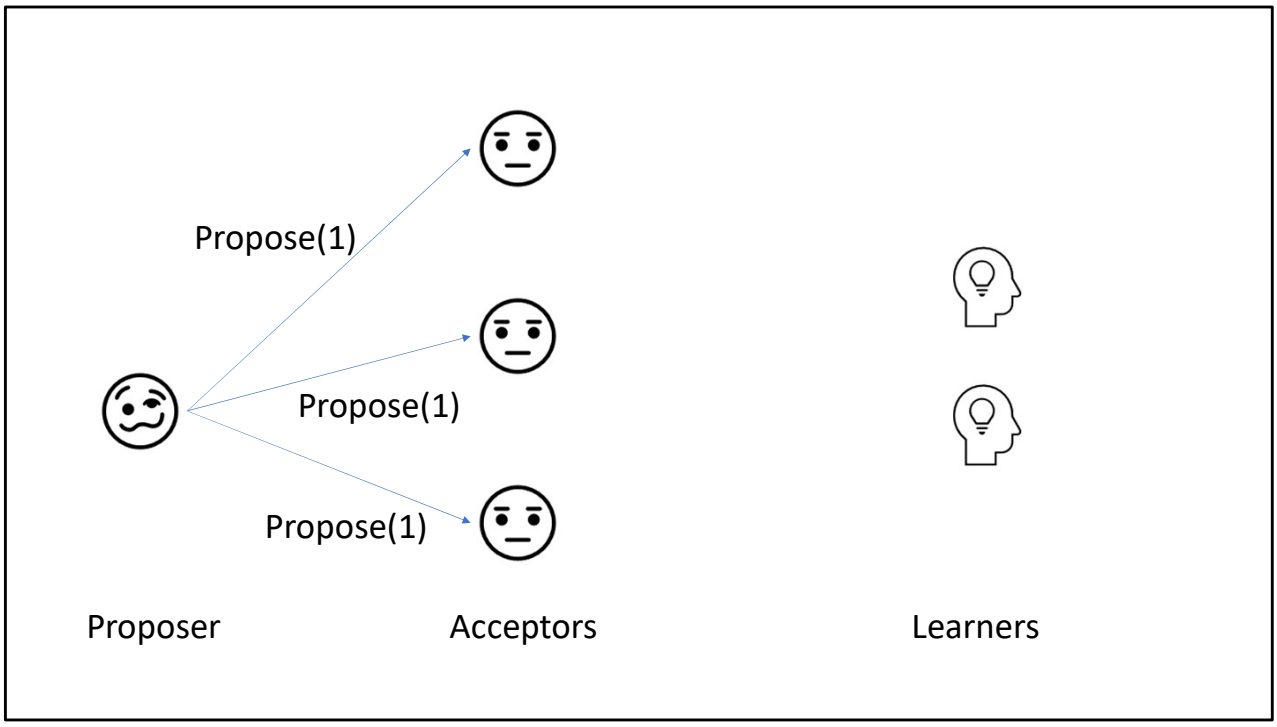
Paxos Protocol

Proposer wants to change the value to *v*

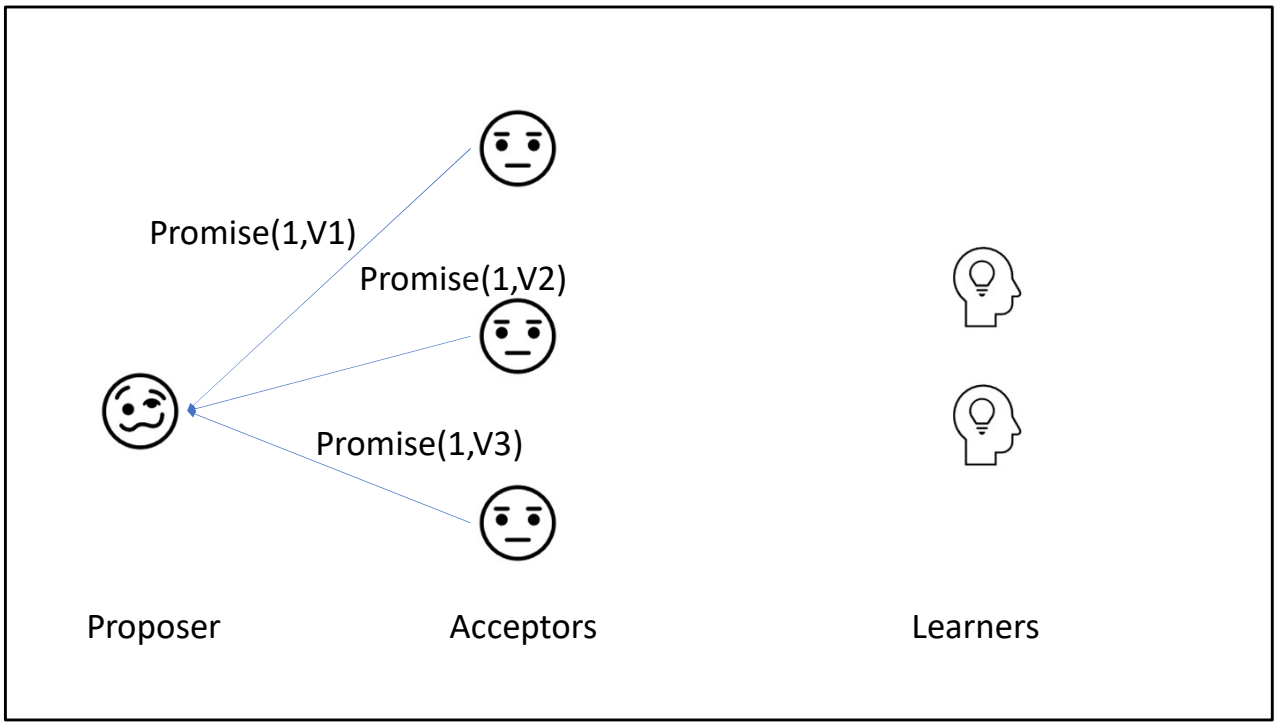
1. Selects proposal number *n* that is greater than all previous proposals
2. Proposes n to a Quorum of nodes (or more / all nodes)
3. Nodes that receive the message either:
   1. **Promise** to reject all future proposals < n
   2. **Reject**, if they've made a similar promise in the past (and n is too small)
4. If a Quorum have sent Promises back, Proposer sends an Accept message
5. Acceptors then accept this value *

They also send the most recently Accepted proposal, if any

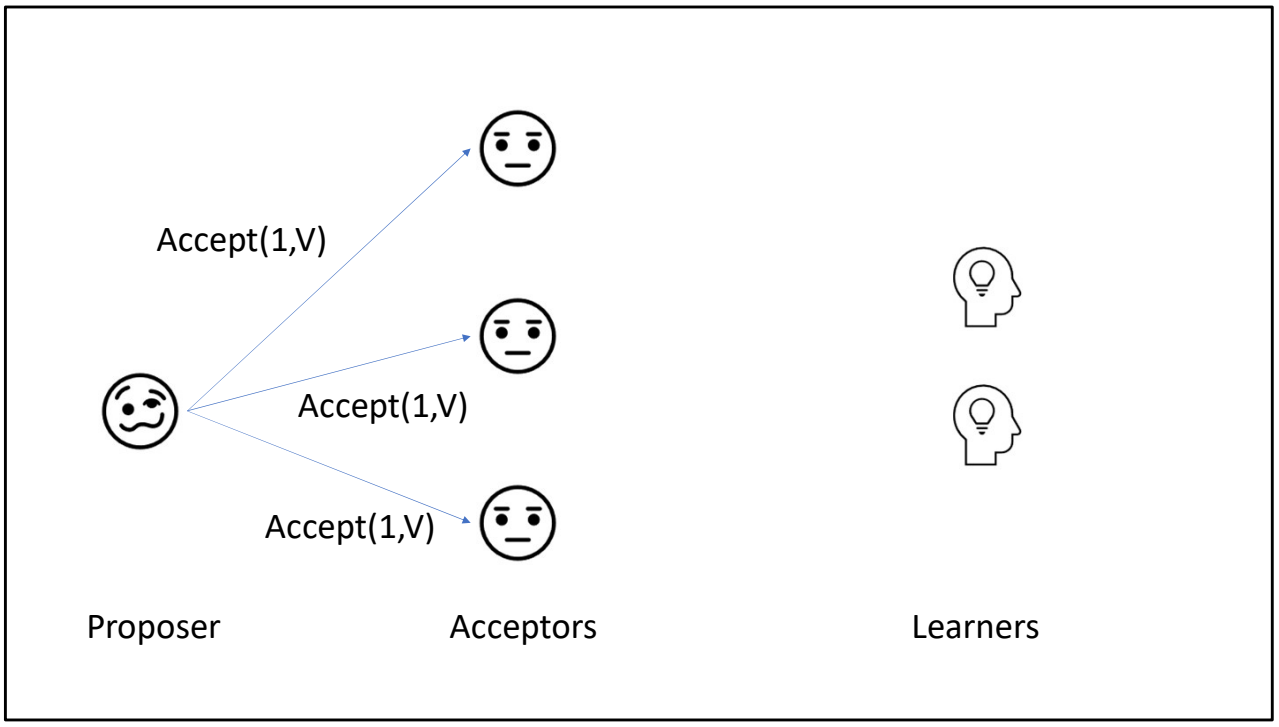
- - if they haven't made another promise in the meantime
- Note -



With 3 accepters, ANY 2 accepters will form a Quorum.

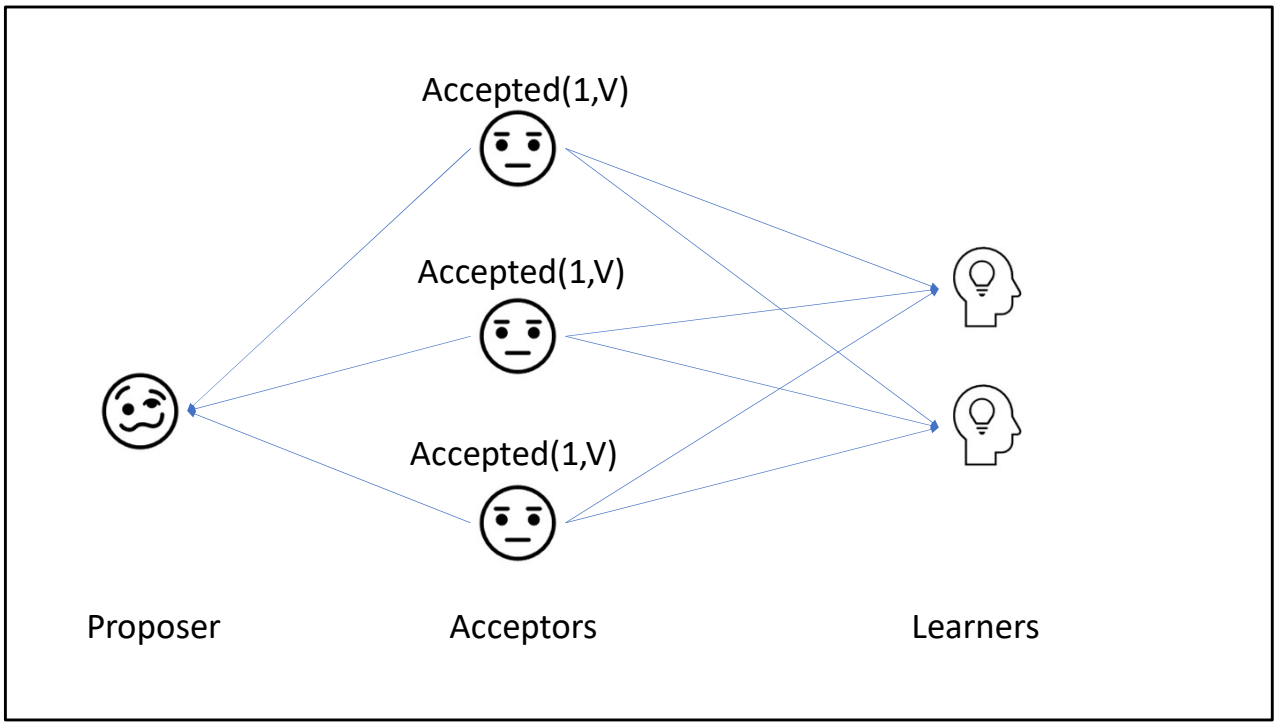


The promise values V1, V2, V3, are a pair of the form (m, w) where m is ID (unique number) of the most recent proposal that this accepter accepted, and w is the value that the acceptor accepted. If the accepter has never accepted a proposal, then this is an empty response.



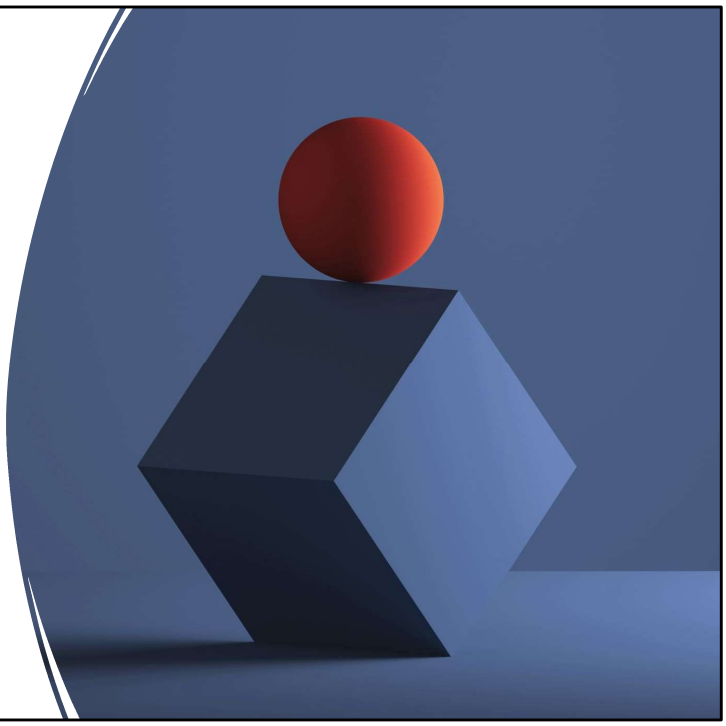
After receiving a Quorum of Promises, Proposer tells them to accept value V.
This V will be either:

- The most recent value sent by an Acceptor as part of their Promise
- The value the Proposer was advocating for (ONLY if all Acceptors sent back “None”)



Acceptors now accept value V. They notify Proposer that they have accepted it, and forward the values to the Learners.
Learners Learn this value, if they get a Quorum of Accepted messages.

That’s the simplified version



Want More?

Take a distributed systems course!

Oh wait…isn't this one of those?

It's not…it's a distributed computing course! We talk a LITTLE about how the systems are engineered, but not how to build them in detail. Sorry.

Types of Consistency

- Strong Consistency
  - After an update operation, **all subsequent access** will return the **same value**
- Weak Consistency
  - After an update operation, **some access** will return the new value, and **others** the old value
- **Eventual Consistency**
  - Special form of Weak Consistency
  - After an update at time t_0 all access after some time $t_1 > t_0$ will return the same value
    - If there were no other updates to that value after the first one

Eventual Consistency In Real Systems

Consider an ATM

- You'd want Strong Consistency, right?
- An ATM that doesn't work makes customers unhappy (Availability > Consistency)
- ATM will work even if partitioned
  - BUT will limit your transaction size to $200
- If you didn't have that amount after all, overdraft fees!

Of course if you deposit into a partitioned ATM and then to buy groceries, then you get overdraft fees because your deposit isn't in the system yet, and won't be until the ATM's network connection is restored.

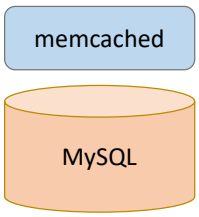
Eventual Consistency In Real Systems

- Bob says to Alice “You haven’t liked my Facebook post!”
- Alice doesn’t see any new posts
- Bob sees it and is confused
- Later that day, it shows up in Alice’s feed
  - She still doesn’t like it. Bob needs to seek validation from within, not from pretend Social Media points

Why is Facebook Eventually Consistent?

- Did the network get partitioned?
- Probably not
- It's a huge distributed network of datacenters, is why
  - If would be unworkable to have strong consistency **AND** low latency
  - If it took 1-2 minutes for a post to stop spinning, people would give up

Facebook Architecture



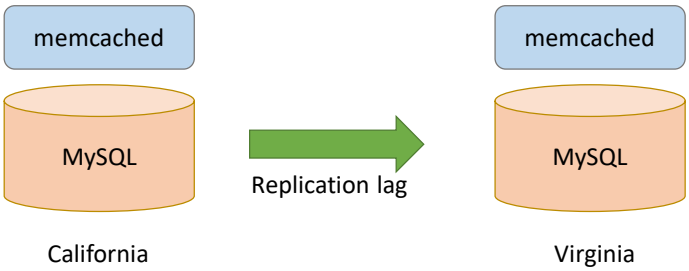
Read path:
Look in memcached
Look in MySQL
Populate in memcached

Write path:
Write in MySQL
Remove in memcached

Subsequent read:
Look in MySQL
Populate in memcached

Source: www.facebook.com/note.php?note_id=23844338919

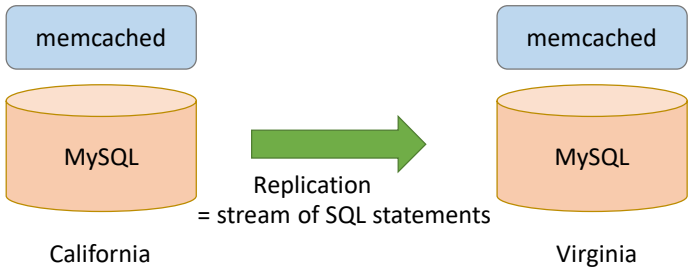
Facebook Architecture: Multi-DC



1. User updates first name from “Jason” to “Monkey”.
2. Write “Monkey” in master DB in CA, delete memcached entry in CA and VA.
3. Someone goes to profile in Virginia, read VA replica DB, get “Jason”.
4. Update VA memcache with first name as “Jason”.
5. Replication catches up. “Jason” stuck in memcached until another write!

Source: www.facebook.com/note.php?note_id=23844338919

Facebook Architecture: Multi-DC



Solution: Piggyback on replication stream, tweak SQL

```
REPLACE INTO profile (`first_name`) VALUES ('Monkey')
WHERE `user_id`='jsobel' MEMCACHE_DIRTY 'jsobel:first_name'
```

Source: www.facebook.com/note.php?note_id=23844338919

What if there are no partitions?

- Tradeoff between **Consistency** and **Latency**:
- Caused by the **possibility of failure** in distributed systems
  - High availability -> replicate data -> consistency problem
- Basic idea:
  - Availability and latency are arguably **the same thing**: unavailable -> extreme high latency
  - Achieving different levels of consistency/availability takes different amount of time

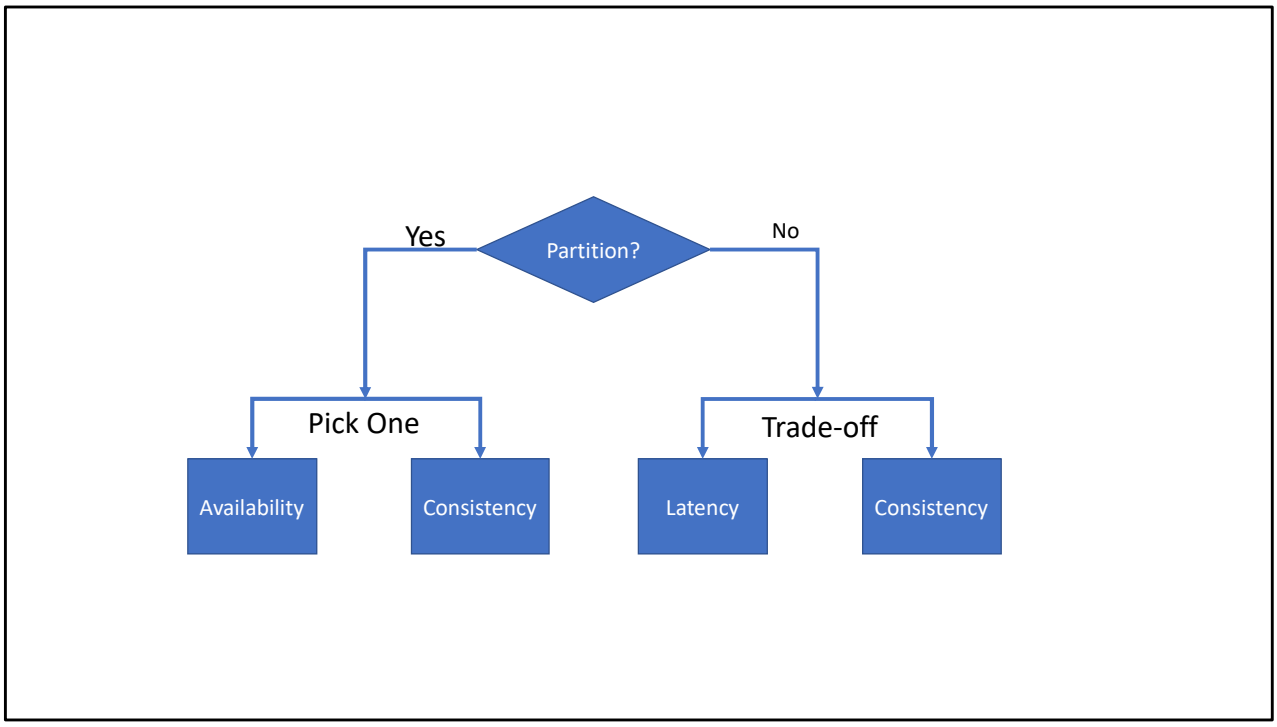
CAP -> PACELC

- A more complete description of the space of potential tradeoffs for distributed system:
  - If there is a **partition (P)**, how does the system trade off **availability and consistency (A and C)**; **else (E)**, when the system is running normally in the absence of partitions, how does the system trade off **latency (L) and consistency (C)**?

Abadi, Daniel J. "Consistency tradeoffs in modern distributed database system design." Computer-IEEE Computer Magazine 45.2 (2012): 37.

99

Really rolls of the tongue


Yes
Partition?
No
Pick One
Trade-off
Availability
Consistency
Latency
Consistency

Examples

- **PA/EL Systems:** Give up both Cs for availability and lower latency
  - Dynamo, Cassandra, Riak
- **PC/EC Systems:** Refuse to give up consistency and pay the cost of availability and latency
  - BigTable, Hbase, VoltDB/H-Store, any ACID DB
- **PA/EC Systems:** Give up consistency when a partition happens and keep consistency in normal operations
  - MongoDB
- **PC/EL System:** Keep consistency if a partition occurs but gives up consistency for latency in normal operations
  - Azure Cosmos*

What? So Cosmos is consistent when partitioned, but not otherwise?
Sort of. It lets you tune your parameters, so you CAN make it PC/EL if you want to. Or PC/EC
If there's a partition, one partition is unavailable, so it's NOT PA. If you elect for EL it's not really PC either though, since it has only eventual consistency.

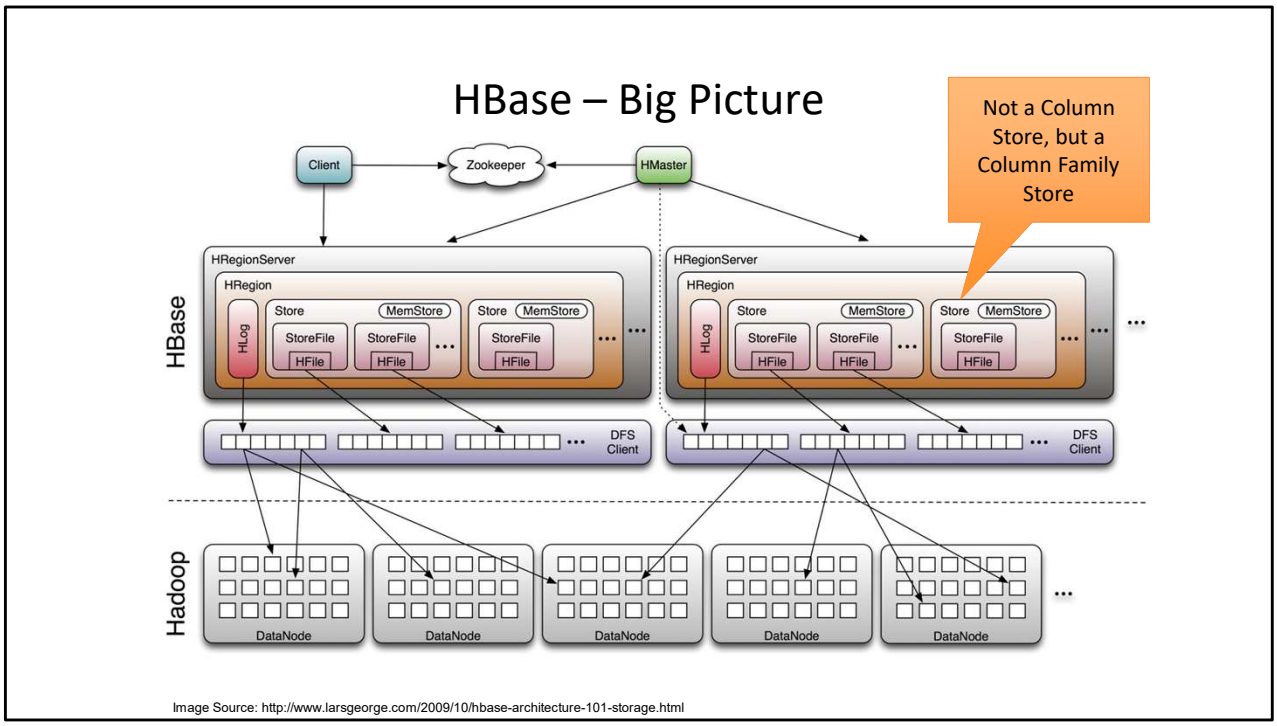


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

Back to this slide again, now that we've talked about types of systems. HBase is PC/EC. Seems like not? Let's dig in

UPDATE: Someone in class asked about "StoreFile" vs "HFile" – these are synonyms. StoreFile is the java object, Hfile is the actual file on HDFS.
The distinction is made in some diagrams as old versions of HBase used the MapFile format, where the StoreFile will be put into two files: the MapFile and the IndexFile. Newer versions use an HFile, which contains both the key-value pairs and the index.

Also important here: Instead of each column being in its own store entirely, they're grouped by family.
If columns within a family are normally read/updated together, you get all the benefits of a column store, with fewer penalties.
(Of course if you only need one column within a family, you're reading unnecessary data still)

HBase is PC/EC? So it has poor latency?

No. HBase is low latency, has non-blocking reads, and strong consistency!

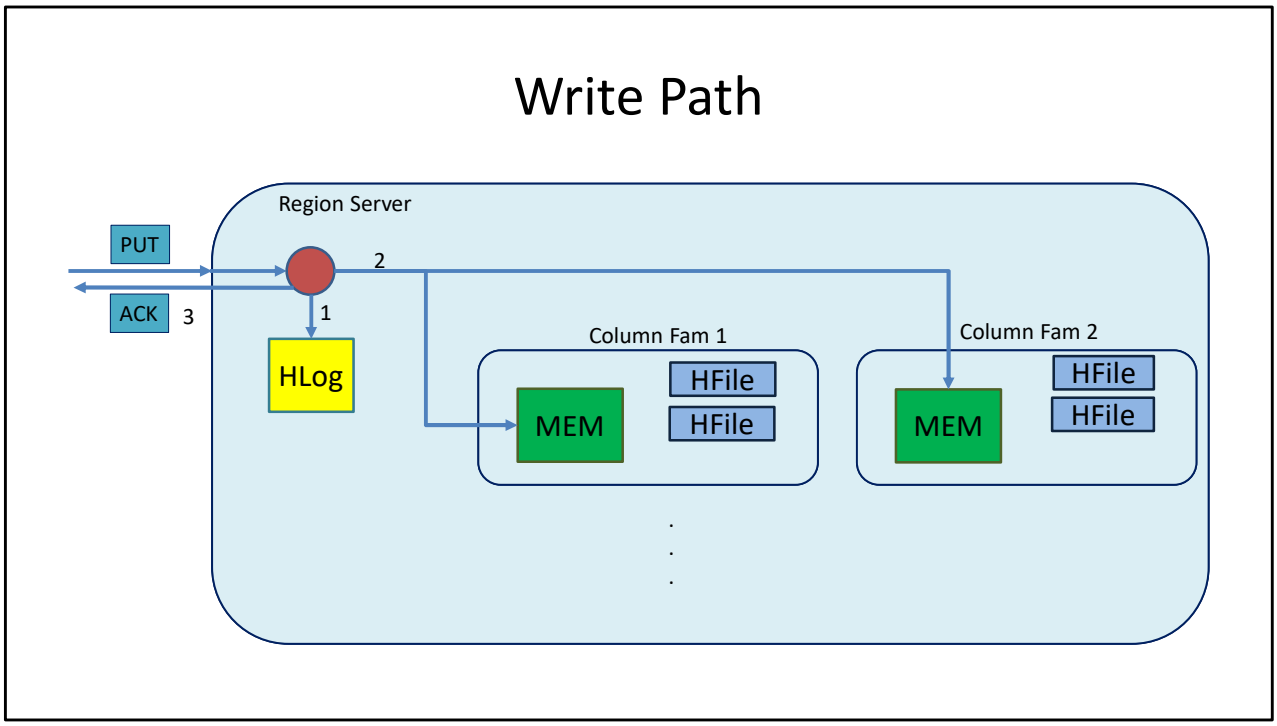
How can this be? Think about the design!

How HBase does it

Transactions are only allowed to update a single row

- With an appropriate row key that's often not much of a limit
- Only ONE Region Server oversees a given Region
  - It uses MVCC for ACID semantics
  - All row queries see results consistent with the moment the transaction begins
  - All row updates happen in isolation
- You can think of each row being its own database!

Write Path



1. Region Server writes transaction into WAL (Called the HLog) HDFS replicates this across the cluster, though not instantly
2. Region Server writes transaction into MemStore for each affected Column Family
3. Region informs client that transaction has been committed
4. As needed, MemStore is spilled to an additional Hfile
    1. To keep the number of files low, HFiles may be merged (compacted)

What's in an HFile

Sorted key-value pairs (+ an index header for fast lookups)

Key?

(row, family, qualifier, timestamp)

The timestamp is used for MVCC

The Region is partitioned by column family, so not needed to actually store this

Strong Consistency!

It wouldn't have values that old

djholtby	hair	style	Mar-03-1999	Short, Spiked
djholtby	hair	colour	Mar-03-1999	Brown, frosted tips
djholtby	beard	style	Mar-03-1999	goatee
djholtby	beard	colour	Mar-03-1999	brown
djholtby	hair	style	Nov-25-2022	balding
djholtby	beard	colour	Nov-25-2022	Brown & Gray

What's in a MemStore?

The content is the same

It's stored in a skip list, sorted by the 3-tuple key

Skip lists are cool, end of discussion

Read Path

When read transaction starts, makes note of time

- Retrieve all requested columns with a timestamp <= read start

That's all there is to it! In a typical case, at least…

Read Path, Replicated Edition

What happens if the row you want belongs to a Region where the Region Server is not responding?

1. You can read immediately from the secondary Region Server for that Region
    1. It might not have replayed a relevant WAL entry yet – stale data
2. You can also wait until all WAL entries from your start time have been replayed
    1. High Latency

In other words at the client's discretion HBase is either EC or EL. The system is both. It lets the client pick their favoured tradeoff

Failure Recovery

When a Region Server comes back online, it has only lost the MemStore

- Using the HFiles it knows when the last flush happened
- Using the HLog, it can replay all writes after the flush, recreating the MemStore exactly as it was

Merges

Two types of HFile Merges Happen

1. Fast Merge – Merges two smaller HFiles together. Frequent
2. Full Merge – Merges ALL HFiles for a given Column Family
   1. Also garbage collects – any timestamps earlier than the oldest query will never be read and can be pruned
   2. Any "delete" markers that do not have an earlier "write" can be removed

Why are these needed to begin with?

"Delete" markers are needed because in MVCC you can't simply delete a value from the table. A query might be running and it needs to see the value as it appeared at the start of the query transaction (consistency / isolation).

Also, HDFS doesn't let you delete lines from a file anyway, so it's not physically possible to delete from one of the HFiles.
It WOULD be possible to delete from the MemStore, but that's still not done because of the MVCC issue.

During compaction, only the most recent entry from before the oldest still-active query can be pruned. If a delete marker is the "oldest" surviving entry then it's unnecessary and can also be pruned.