CS 4440 A

Emerging Database Technologies

Lecture 22

04/20/26

Annoucement

- Assignment 5 due next Monday (Apr 27)

Rank	Submission Name	Average Accuracy	California Schools Accuracy	Formula 1 Accuracy
1	Teymur Gadzhiev	65.28	71.05	59.52
2	ST	63.84	65.79	61.9
3	DP	63.03	73.68	52.38

- Review lecture this Wednesday during class
- Final format:
  - Take-home, open book and notes, closed Internet
  - Released Thursday, April 30 5:30PM, due Friday, May 1 5:30PM
  - Covering the whole class, but focus on contents after the midterm

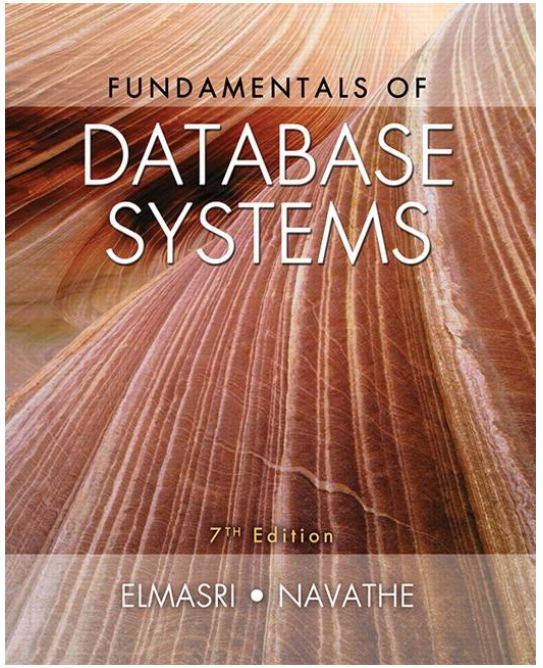
Agenda

1. NoSQL
2. NewSQL
3. Course Summary

Reading Materials

Fundamental of Database Systems (7th Edition)

- Chapter 24: NOSQL Databases and Big Data Storage Systems



Research Papers:

- The LSM Tree Paper
- Spanner Paper

1. NoSQL

NoSQL

NoSQL stands for “Not Only SQL” or “Not Relational”

- relax on consistency requirements to gain efficiency and scalability

Designed to scale simple “OLTP”-style application loads

- to provide good horizontal scalability for simple read/write database operations distributed over many servers

Originally motivated by Web 2.0 applications

- these systems are designed to scale to thousands or millions of users

NoSQL vs RDBMS

We will highlight a few important differences.

Data model

- Relational vs key-value, graph etc.
- Schema flexibility

Consistency guarantee

- Tradeoff between consistency and availability

Key Index structure

- Read vs write optimized

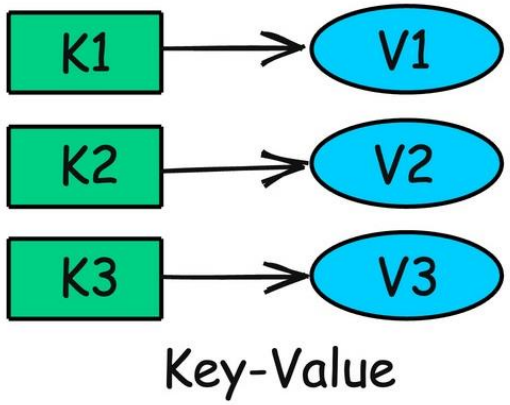
Example NoSQL systems

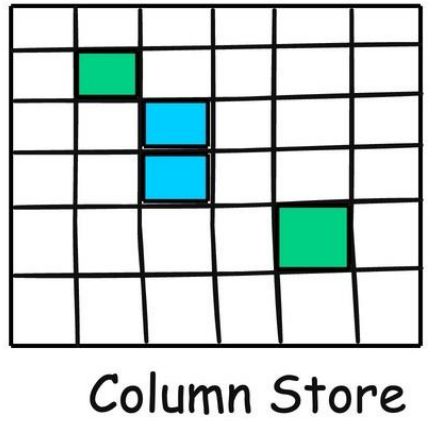
Key-value store:
DynamoDB, Redis

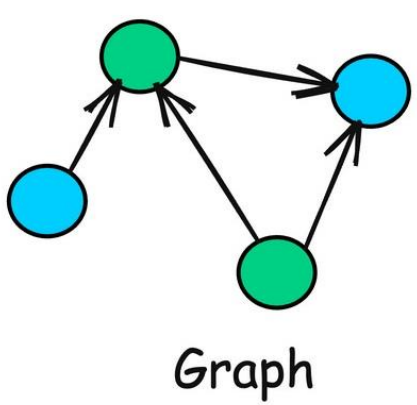
Document store:
MongoDB, CouchDB

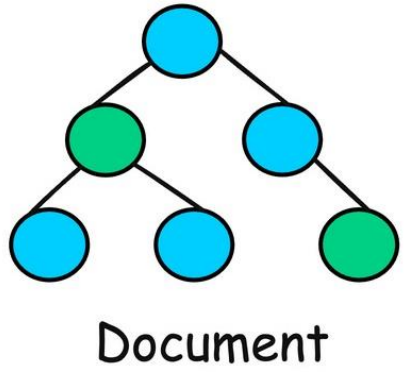
Wide-column stores:
Cassandra, BigTable

Graph database:
Neo4j, AWS Neptune





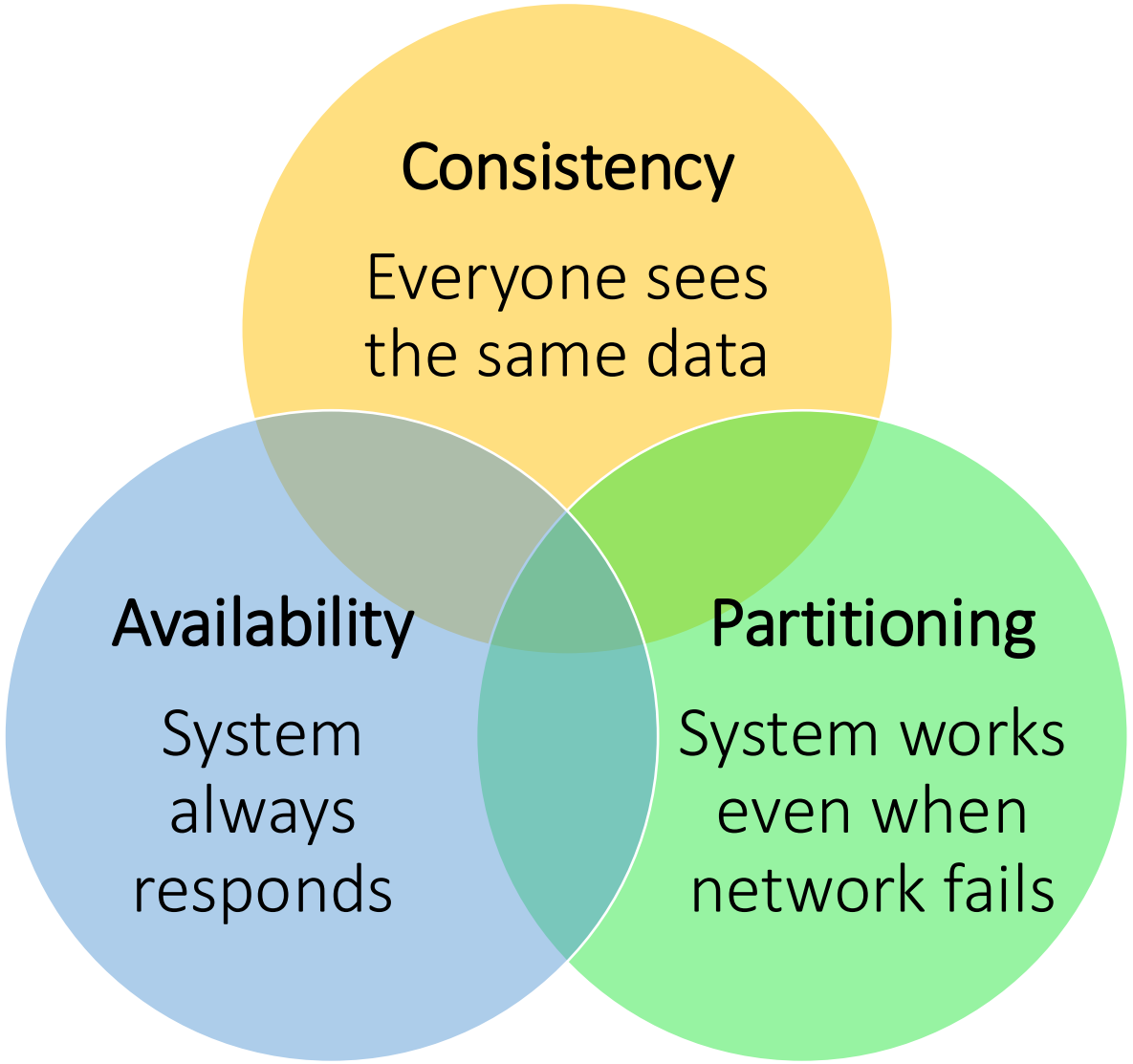




CAP Theorem

Any distributed data store can provide only two of the following three guarantees: **C**onsistency, **A**vailability, and **P**artition-tolerance.

CP: strong consistency
AP: eventual consistency



NoSQL systems generally give up consistency (AP system)

Image source: https://medium.com/nerd-for-tech/understand-cap-theorem-751f0672890e

Problem with ACID

Recall ACID for RDBMS desired properties of transactions

Challenge: maintaining ACID across multiple servers requires coordination

- Coordination = latency + reduced availability

Example: adding an item to shopping cart

- Option 1: System says "I can't guarantee your cart is consistent across all servers, so I'm refusing your request"
- Option 2: System says "I'll accept your cart addition on this datacenter, we'll sync up later"

ACID vs BASE

NOSQL systems typically do not provide ACID: they sacrifice strong consistency for availability/performance

- Basically Available
- Soft state
- Eventually consistent

The systems differ in how much they give up

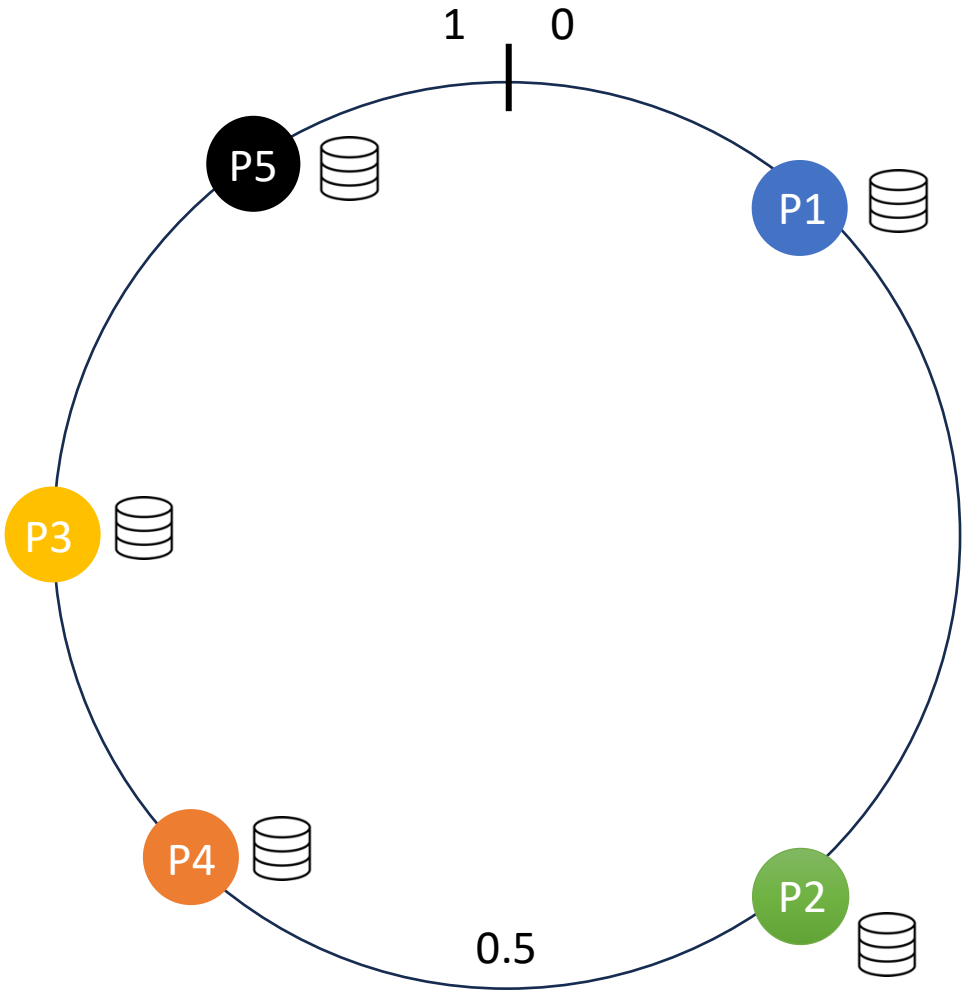
- e.g., most of the systems call themselves “eventually consistent”, meaning that updates are eventually propagated to all nodes
- Many modern NoSQL systems offer tunable consistency - you can choose between strong consistency (more ACID-like) and eventual consistency (BASE) depending on your need

Case Study: Dynamo

- Amazon (2007): hundreds of internal services, millions of customers, peak shopping traffic
  - Internal SLAs were measured at the 99.9th percentile – tail matters!
  - Core requirement: the shopping cart **must always be writable** — availability over consistency
- "Dynamo targets applications that operate with weaker consistency (the "C" in ACID) if this results in high availability."
- Influence: Cassandra (originally at Facebook, directly inspired by Dynamo + BigTable), Riak, DynamoDB

Reading: Dynamo: Amazon's Highly Available Key-value Store

Consistent Hashing



Virtual Nodes: each physical node gets *multiple positions* on the ring

- The number of virtual nodes can be proportional to a node's capacity
- Reduce hot spot
- When a node joins or leaves, its load is spread across many other nodes rather than dumped on one neighbor

Quorum-based Reads and Writes

- Three parameters: **N** (replicas, default=3), **W** (nodes that must ack a write), **R** (nodes consulted on read)
- Rule: R + W > N guarantees you read at least one up-to-date copy → strong consistency
  - Common (N,R,W) configuration is (3,2,2)
  - low values of W and R: increases risks of inconsistency
  - Increasing W improves durability (tolerate more failures without data loss) but reduces availability
- Dynamo lets applications **choose per service** based on their needs

Problem with B+-Trees

Optimized for reads, not for writes (insert, update, delete)

Write amplification:

- ratio of the amount of data written to the storage device versus the amount of data written to the database
- Inserting and deleting a record could require updating multiple pages on disk (many random disk I/Os)

NoSQL systems face extreme *write workloads*:

- High write throughput: social media posts, IoT sensors, logs
- Distributed writes across many nodes
- Append-heavy workloads

Log-structured Merge Tree (LSM Tree)

Proposed by O'Neil, Cheng, and Gawlick in 1996

Uses write-optimized techniques to significantly speed up inserts

Used by many NoSQL systems:

- e.g., Bigtable, Cassandra, Dynamo, HBase, RocksDB, InfluxDB

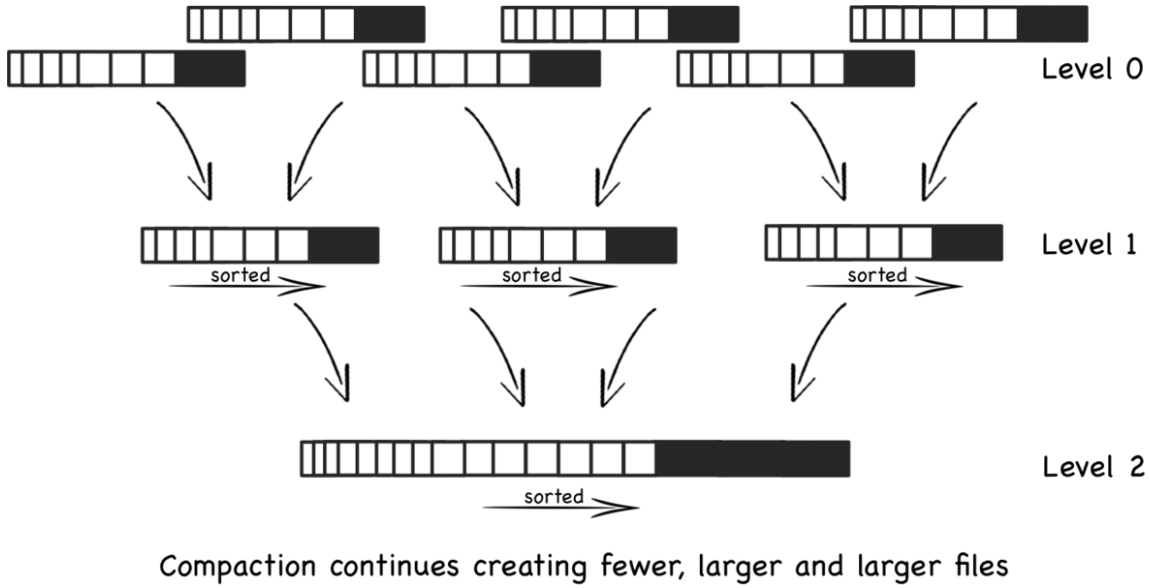


Image source: https://en.wikipedia.org/wiki/Log-structured_merge-tree

LSM Tree: Overview

Log-structured · Merge · Tree

Log-structured

Data is written sequentially in append-only fashion, like writing to a log file

Merge

As data evolves, sequentially written runs of key-value pairs are merged

- Runs of data are indexed for efficient lookup
- Merges happen only after much new data is accumulated

Tree

The hierarchy of key-value pair runs form a tree

- Searches start at the root, progress downwards

LSM Tree: Overview [O'Neil, Cheng, Gawlick '96]

An LSM-tree comprises a hierarchy of levels of increasing size

- All data inserted into in-memory tree (C_0); no I/O cost at this step
- Larger on disk levels ($C_{i>0}$) hold data that does not fit into memory

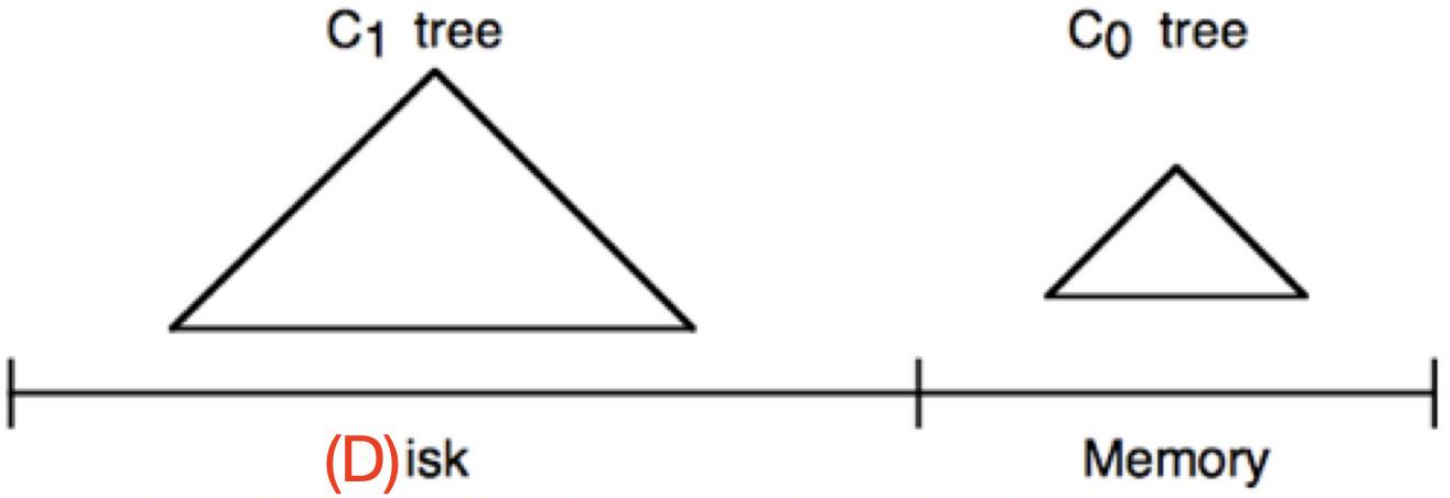


Figure 2.1. Schematic picture of an LSM-tree of two components

LSM Tree: Overview [O'Neil, Cheng, Gawlick '96]

When a level exceeds its size limit, its data is merged and rewritten

- Also called "compaction"
- Higher level is always merged into next lower level (C_i merged with C_{i+1})
- Merging always proceeds top down

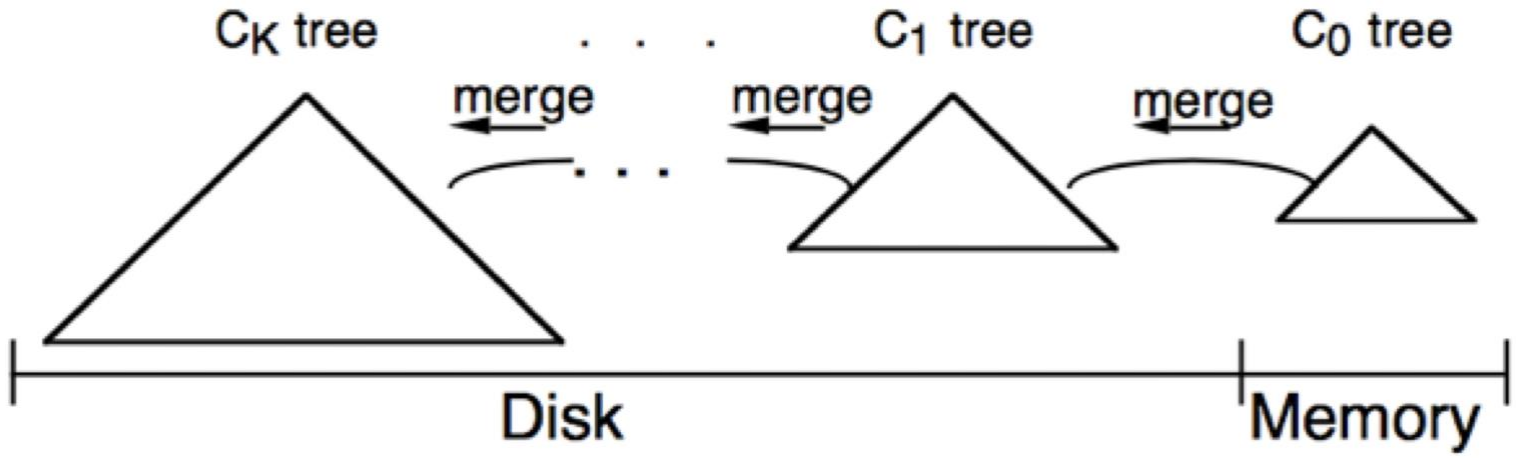


Figure 3.1. An LSM-tree of K+1 components

LSM Tree: Inserts

New data is written to an in-memory buffer (MemTable)

- typically organized as a balanced tree (like a Red-Black tree) to maintain sorted order
- Append-only log

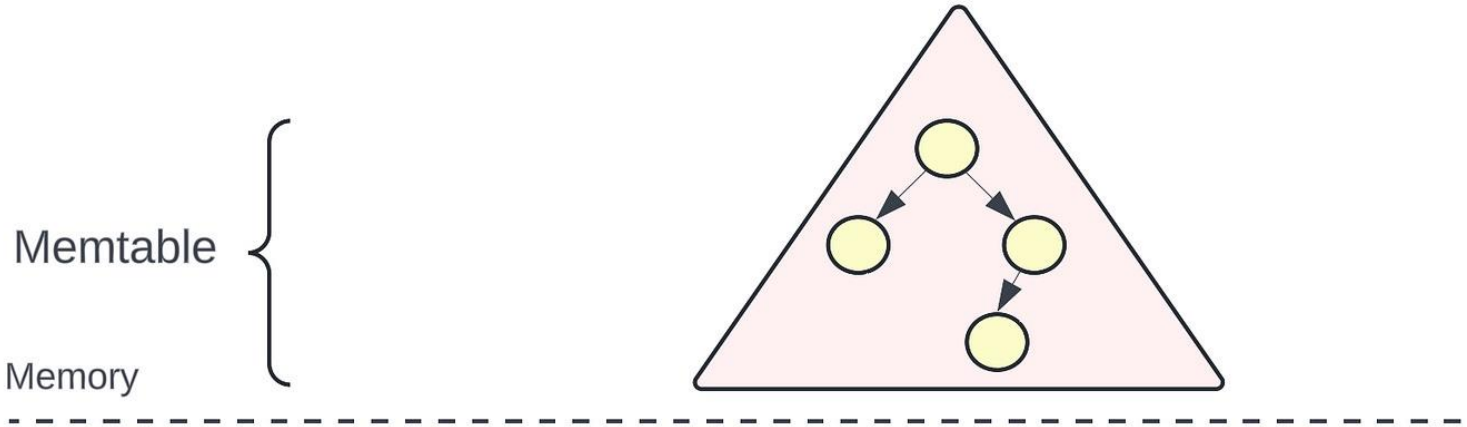


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Inserts

New data is written to an in-memory buffer (MemTable)

- typically organized as a balanced tree (like a Red-Black tree) to maintain sorted order

When the MemTable fills up, it's flushed to disk as a new SSTable file (the “run”) in Level 0

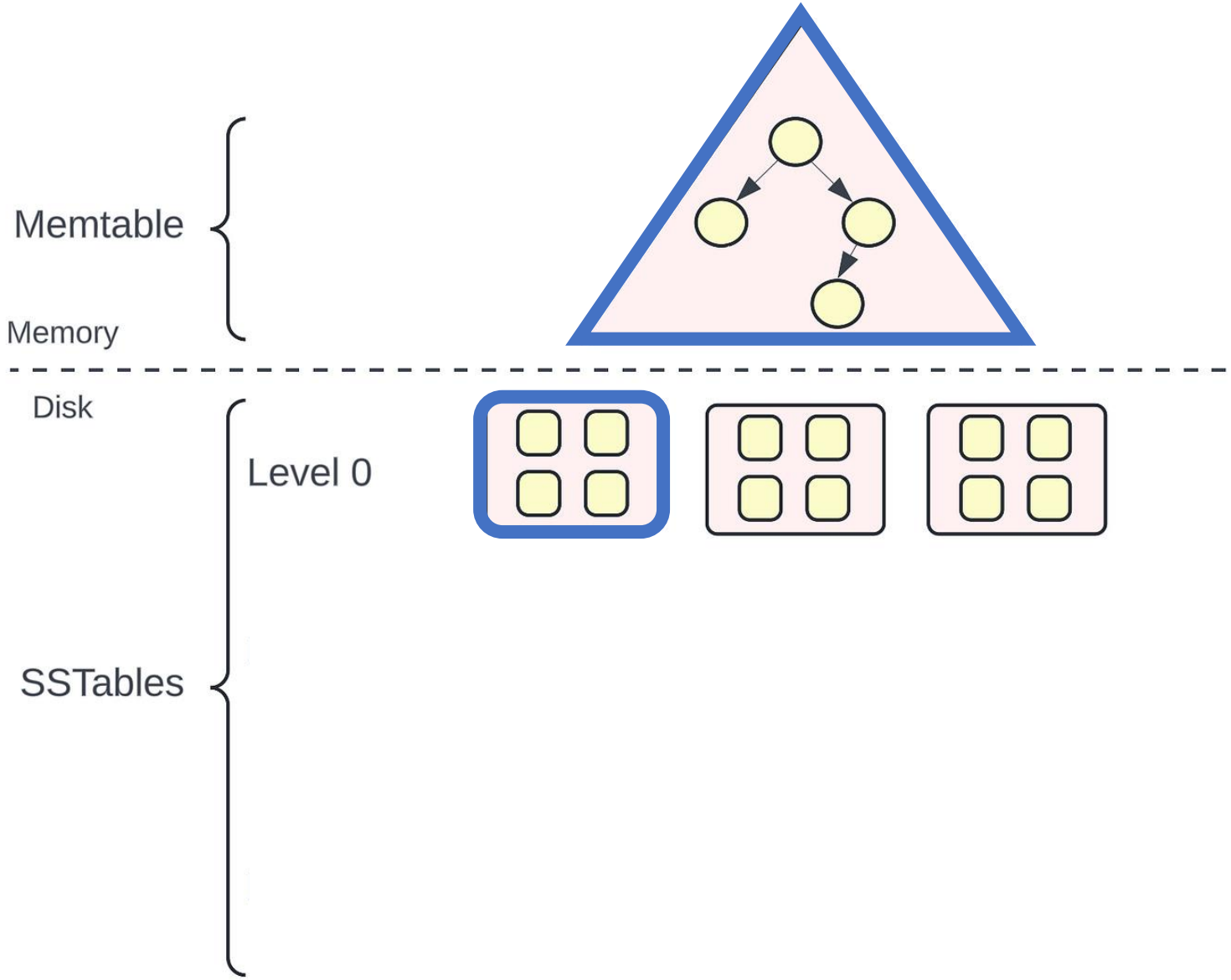


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Inserts

Once too many L0 files exist, compaction merges them into L1

- k-way merge sort
- Input: L0 files + L1 files with overlapping key ranges
- Output: new L1 files with non-overlapping ranges

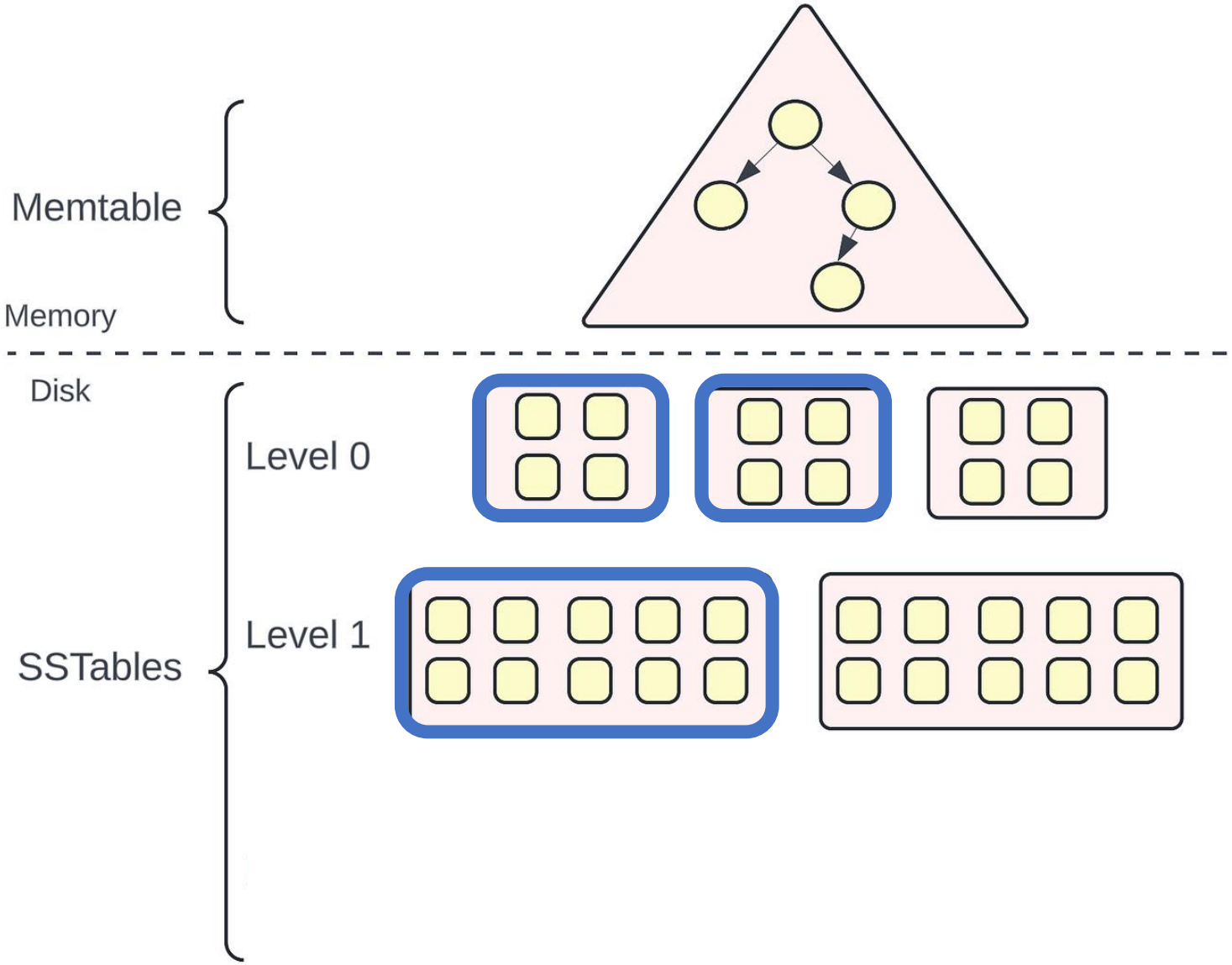


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Inserts

Once too many L0 files exist, compaction merges them into L1

Similarly, when there are too many L1 files, compaction merges them into L2

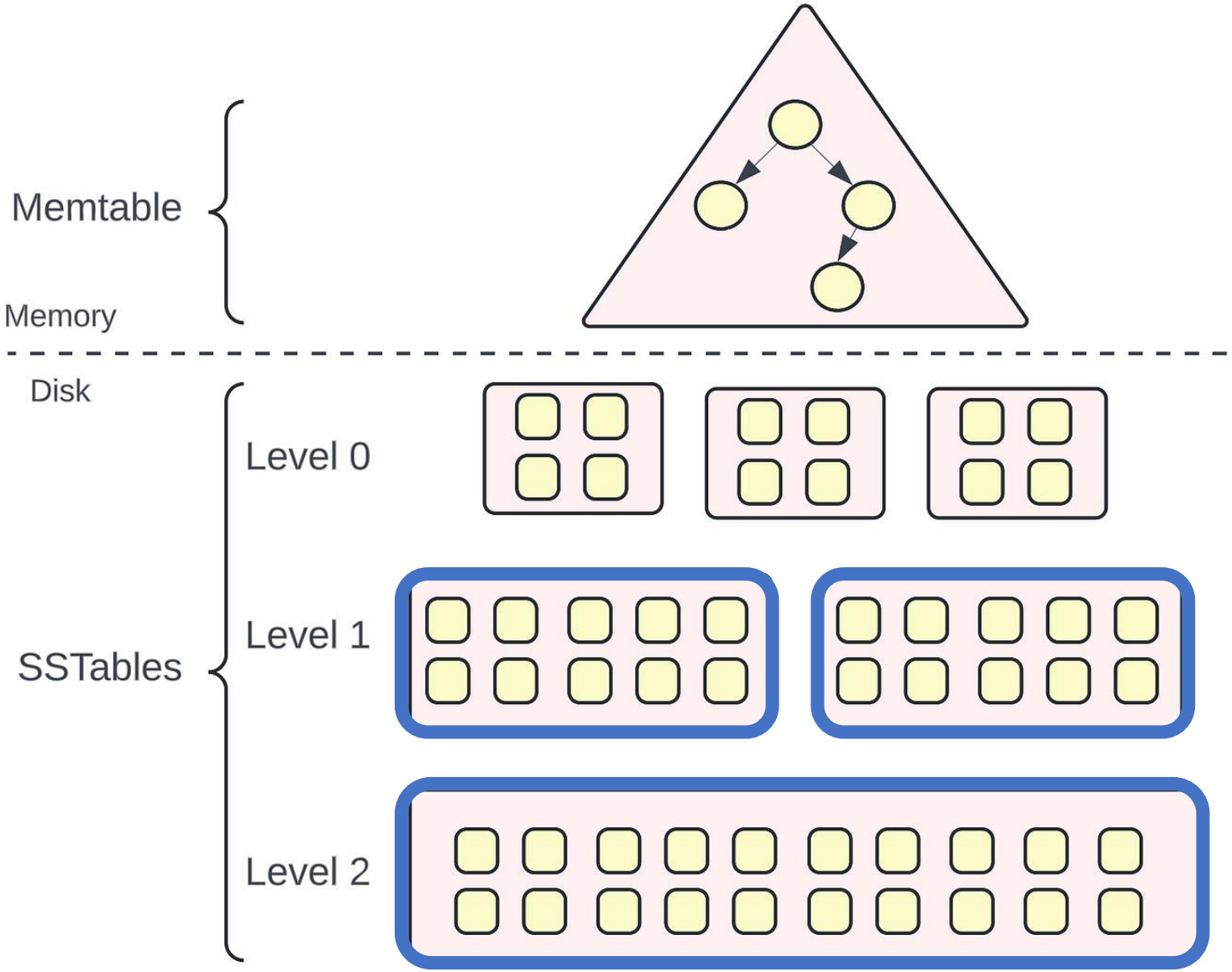


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Inserts

Once too many L0 files exist, compaction merges them into L1

Similarly, when there are too many L1 files, compaction merges them into L2

Different compaction strategies:

- Size tiered compaction strategy (bigger files in deeper levels)
- Leveled compaction strategy (more files per level)

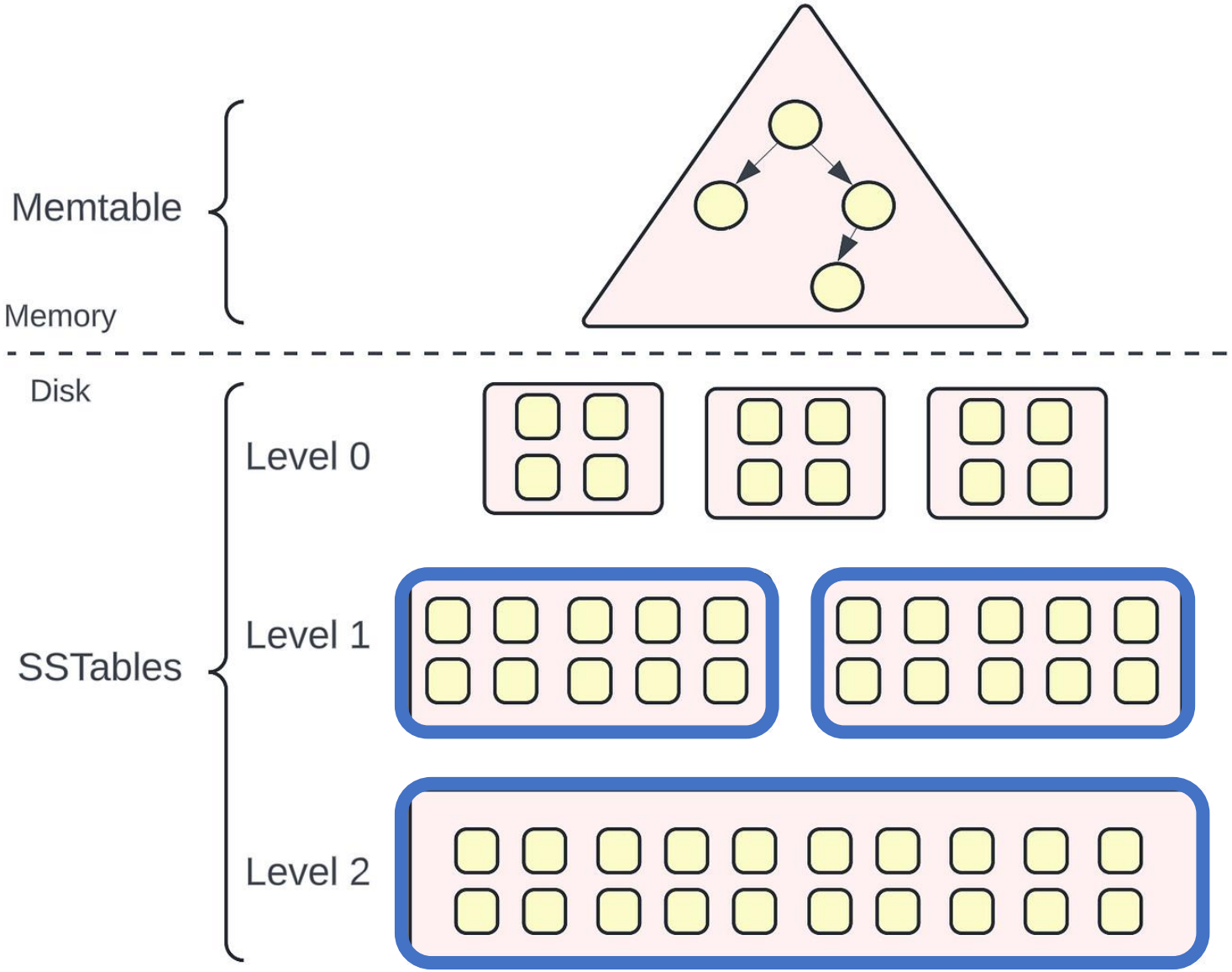


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

B+ Tree vs LSM Tree

	B+ Tree	LSM Tree
Structure	Balanced tree with internal nodes and leaves	Multiple levels, memtable, and SSTables
Reads	Fast lookups and range queries	Potential for slower reads (multiple SSTables may need to be checked)
Writes	Slower, as it can involve rebalancing, splitting and merging nodes	Faster, append-only for the initial writes
Typical Use Cases	Relational databases, read-heavy workloads	NoSQL databases, log-structured systems, write-heavy workloads

2. NewSQL

The rise of NewSQL

Online transaction processing (OLTP)

- Read/write transactions that are short-lived and repetitive
- Touch a small subset of data using indexes

Online analytical processing (OLAP)

- Introduced in the 2000's as Data Warehouses for analyzing large data
- Complex read-only queries (aggregations, multi-way joins)

At some point, OLTP was not fast enough, which led to NoSQL systems

NewSQL: NoSQL performance for OLTP + ACID

- Sacrificing ACID for better performance is no longer worth the effort

Case study: Google Spanner

- Previously, Google used sharded MySQL for their Ads database
- At some point, re-sharding took multiple years
  - Remember: cannot afford to shutdown Ads system, so need to do this carefully
- Could not use existing NoSQL databases (BigTable, Megastore) because they either did not fully support ACID transactions or were too slow
- Took 5 years to develop Spanner, and 5 more years to make it available on Cloud
  - These systems are not easy to implement!

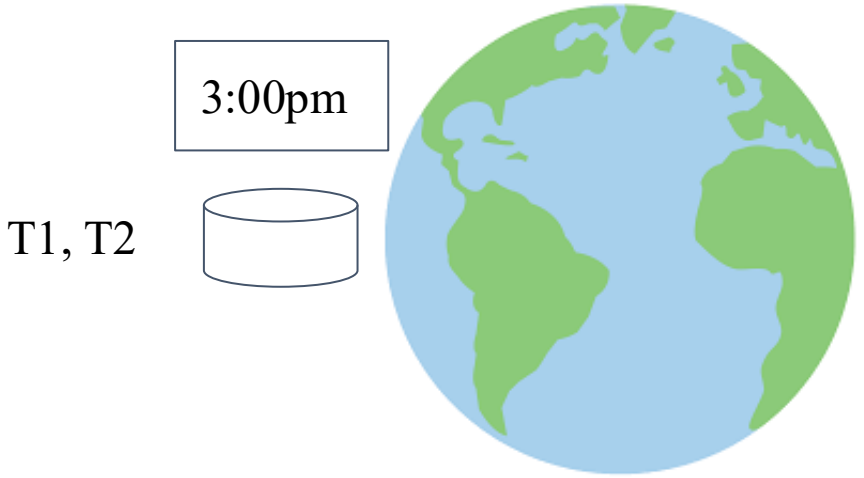
Q: Which properties in the CAP theorem do Spanner provide?

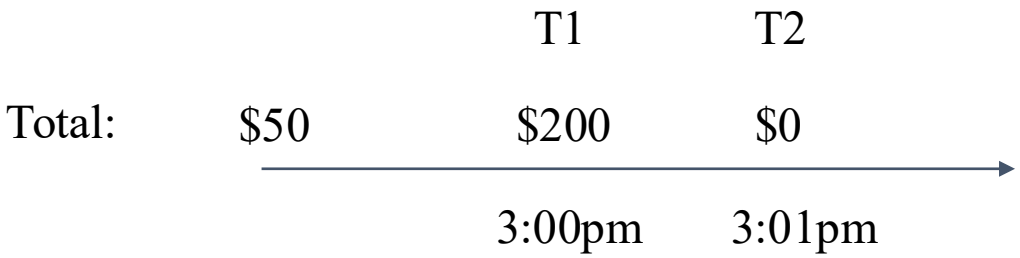
Motivating scenario: banking

- Start with $50 in account (consists of checkings and savings accounts)
- T1: deposit $150 on savings account
- T2: debit $200 from checkings account
- Say client (i.e., you) issues T1 and then T2
- At the end of the day, any negative balance in one account is covered by the other
- Suppose **total** balance must not be negative at any point
  - That is, Spanner must never run T2 and then T1

Easy on single-machine database

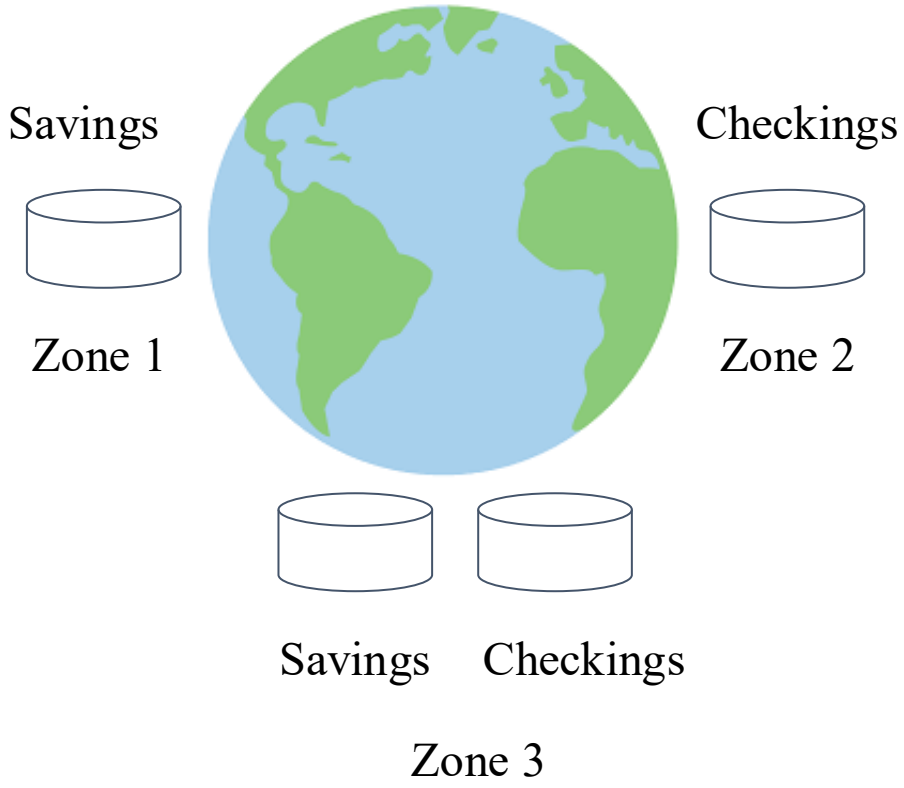
- Give monotonically-increasing timestamps to T1 and then T2
- If another transaction reads the database, use snapshot with most recent timestamp
  - Total balance is never negative





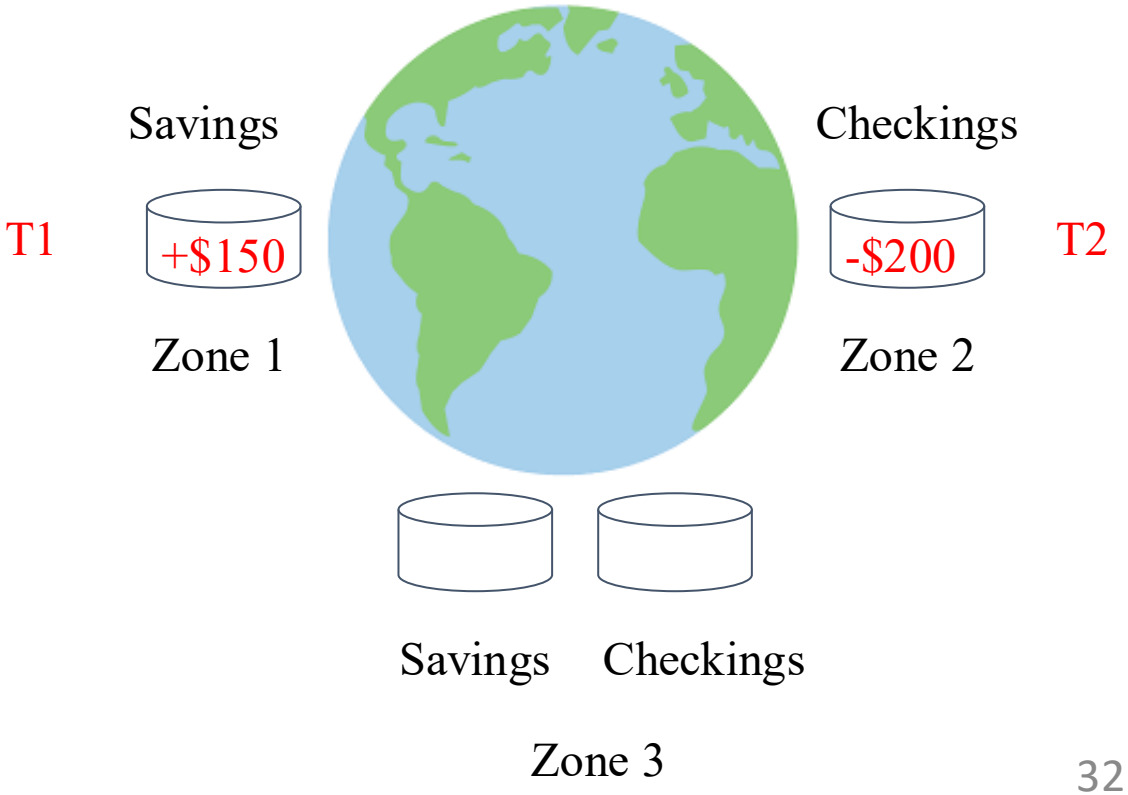
Not easy if database is distributed

- Suppose database is sharded and replicated in three different data centers



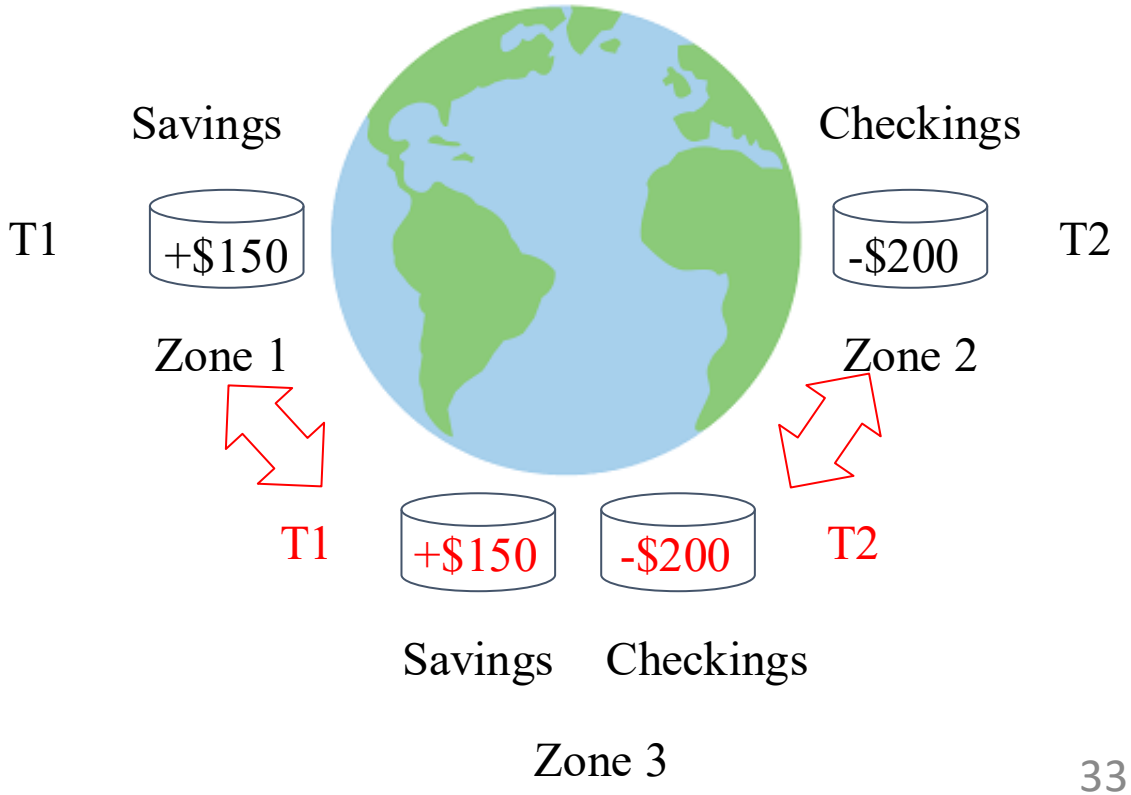
Challenge 1: Consistency

- Need to write on replicas as if there was a single transaction running



Challenge 1: Consistency

- Need to write on replicas as if there was a single transaction running
- Use existing distributed database techniques
  - Use **Paxos algorithm** for synchronizing writes
  - Will not go into details



Spanner: 2PC + Strict 2PL

Ensure isolation and strong consistency

In Spanner, each 2PC member is a Paxos group, which is highly available even if some of its Paxos participants are down.

Data is divided into groups that form the basic unit of placement and replication.

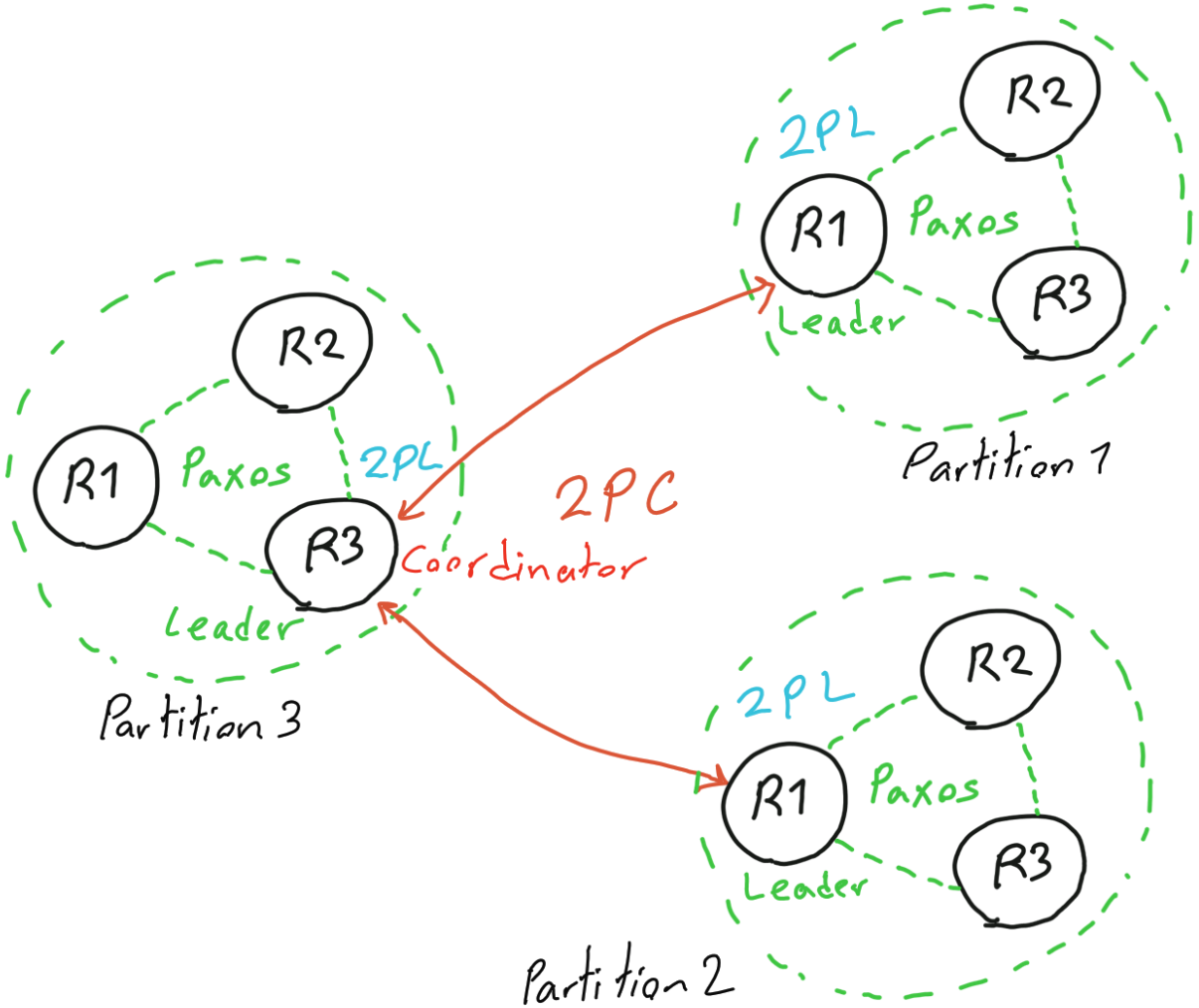
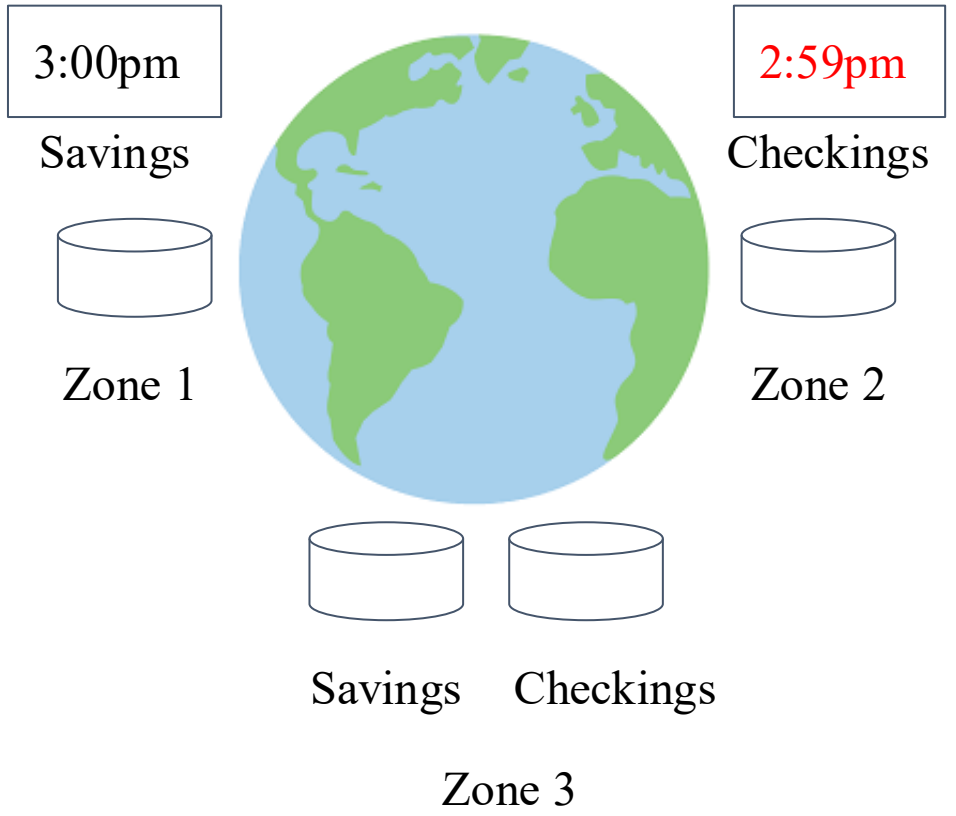
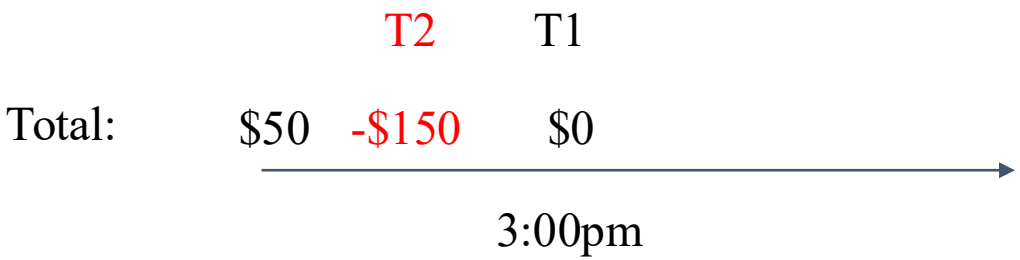


image source: https://www.mydistributed.systems/2020/08/google-spanner.html

Challenge 2: Clock Uncertainty

- If clock on the right is slower, then T2 may have a smaller timestamp than T1
- A transaction that reads after T2 sees a negative total balance!





Desired Property: External Consistency

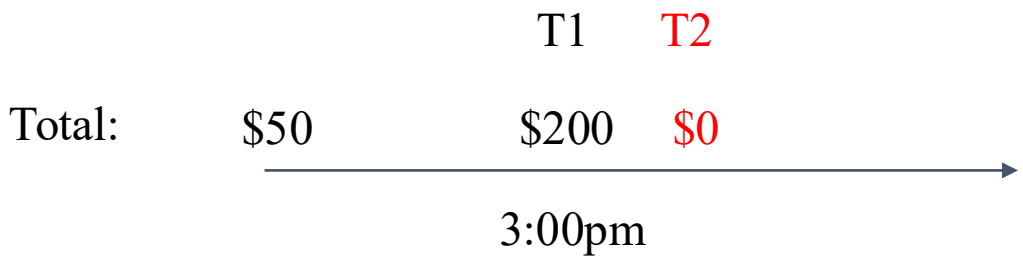
- **External consistency is the guarantee that the serialization order matches real-time order.** If T1 finishes before T2 begins (as observed by any external observer), then T1's timestamp < T2's timestamp, guaranteed.

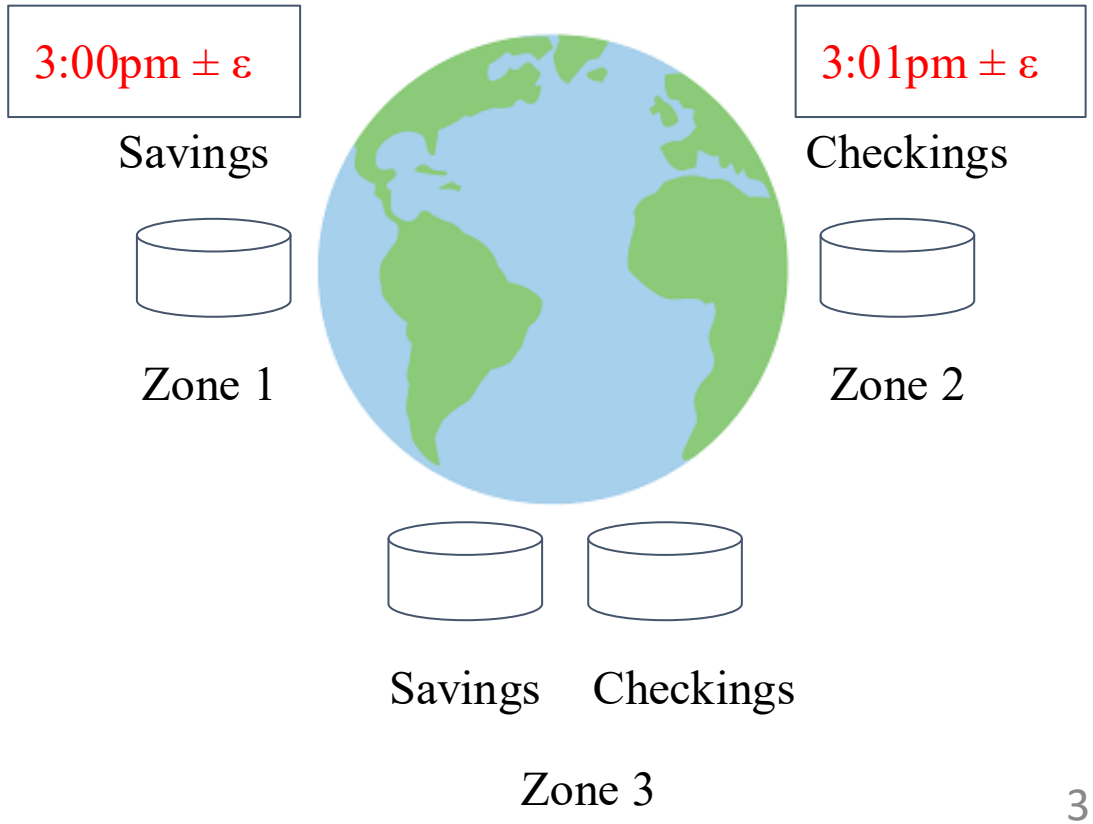
In Spanner, commit order (= timestamp order) respects global wall-time order

- Same as a traditional database using strict 2PL
- System behaves as if all (conflicting) transactions were executed sequentially in one machine

Solution: TrueTime

- Global time with bounded uncertainty
- Guarantees that if T1 commits before T2 starts, then ts(T1) < ts(T2)
- Spanner "waits out" any uncertainty





True Time

Idea: There is a global “true” time t

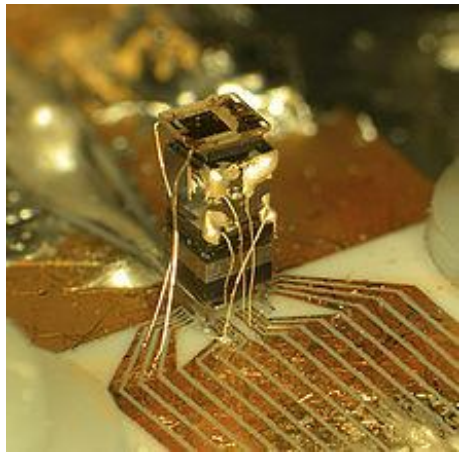
TT.now() = $t \in$ [earliest, latest]

- TT.now().earliest: definitely in the past
- TT.now().latest: definitely in the future

Slide Source: https://www.infoq.com/presentations/spanner-distributed-google/

TrueTime implementation

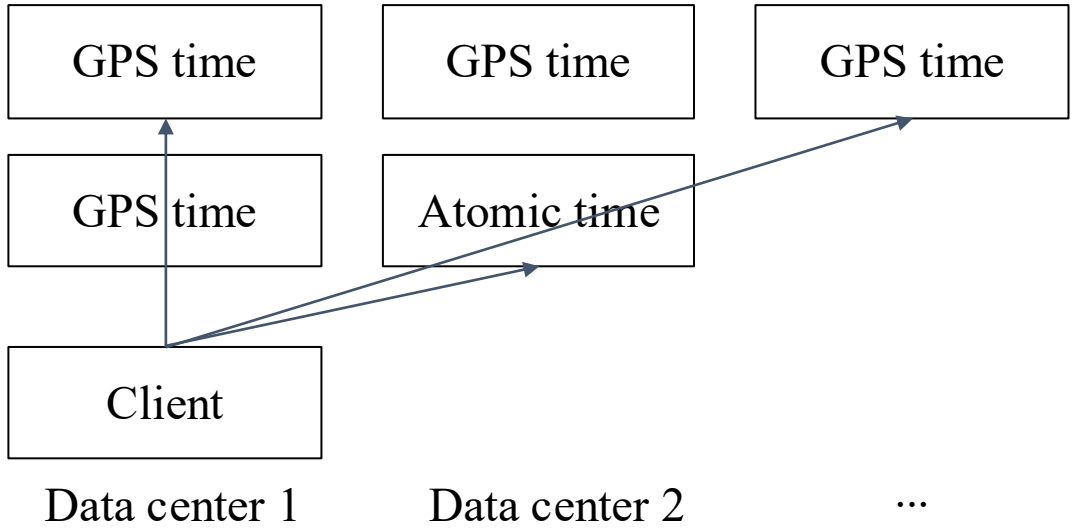
- Use time master machines that have GPS or atomic clocks
  - GPS is precise, but may have connection problems
  - Atomic clocks do not have connections, but may drift
  - The two types complement each other and are not expensive



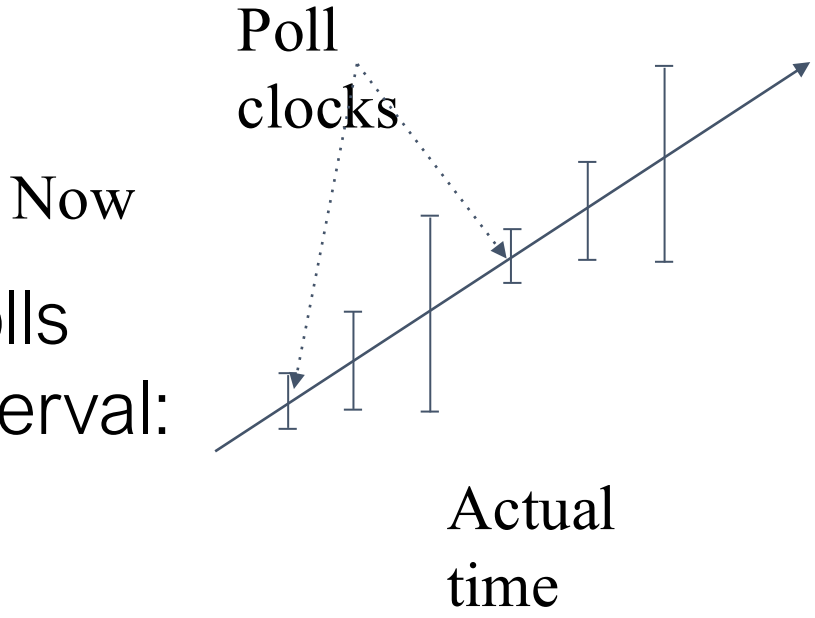
TrueTime implementation

Step 1: periodically poll [earliest, latest] of selected GPS and atomic clock times

- Initially, [earliest, latest] = now ± ε



TrueTime implementation



Step 2: Account for local clock drift between polls

- Right after a poll, TrueTime returns a tight interval: [now - ε, now + ε]
- If X seconds passed,
  - now += X seconds
  - ε += X * 200μs (200μs per second is an upper bound of clock drift)
- Smaller ε = shorter commit wait = better performance, so tight clocks matter for both correctness and speed

3. Course Summary

Course Summary

We learned…

1. How to design a database

1. Intro

2. Relational Algebra

3-4. Design Theory

5-8. TXNs

10-14. Storage, Index

15-17. QO

18-22. Beyond RDBMS

Course Summary

We learned…

1. How to design a database
2. How to handle concurrent user requests and crashes/aborts

1. Intro

2. Relational Algebra

3-4. Design Theory

5-8. TXNs

10-14. Storage, Index

15-17. QO

18-22. Beyond RDBMS

Course Summary

We learned…

1. How to design a database
2. How to handle concurrent user requests and crashes/aborts
3. **How records are stored and indexed**

1. Intro

2. Relational Algebra

3-4. Design Theory

5-8. TXNs

10-14. Storage, Index

15-17. QO

18-22. Beyond RDBMS

Course Summary

We learned…

1. How to design a database
2. How to handle concurrent user requests and crashes/aborts
3. How records are stored and indexed
4. **How to optimize the performance of a database**

1. Intro

2. Relational Algebra

3-4. Design Theory

5-8. TXNs

10-14. Storage, Index

15-17. QO

18-22. Beyond RDBMS

Course Summary

We learned…

1. How to design a database
2. How to handle concurrent user requests and crashes/aborts
3. How records are stored and indexed
4. How to optimize the performance of a database
5. How RDBMS relates to OLAP, distributed/parallel processing framework, Parallel and Distributed DBMS, NoSQL etc

1. Intro

2. Relational Algebra

3-4. Design Theory

5-8. TXNs

10-14. Storage, Index

15-17. QO

18-22. Beyond RDBMS

Relational databases => Data-intensive systems

Most important computer applications must manage, update and query datasets

- Bank, store, search app…

Data quality, quantity & timeliness becoming even more important with AI

- Machine learning = algorithms that generalize from data

Relational databases => Data-intensive systems

Relational databases are the most popular type of data-intensive system (MySQL, Oracle, etc)

Many other systems facing similar concerns: key-value stores, streaming systems, ML frameworks, your custom app?

Reliability in the face of crashes, bugs, bad user input, etc
Concurrency: access by multiple users
Performance: throughput, latency, etc
Access interface from many, changing apps
Security and data privacy (not covered in this course)

Beyond this class

Classes:

- CS 4420/6422: Database System Implementation
- CS 4423/6423: Advanced Database System Implementation
- CS 6220: Big Data Systems and Analytics
- CS 8803-LRV: Large Scale & Real-Time Visual Analysis
- CS 8803-DML: Data-Centric Machine Learning

DB research at GT:

- Data Systems and Analytics Group