CS 6400 A

Database Systems Concepts and Design

Lecture 24

11/23/26

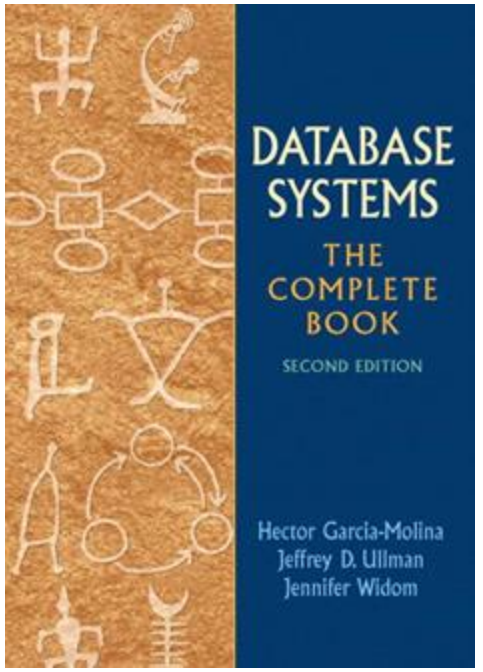
Announcements

- Quiz 3 rescheduled to next Wednesday (Dec 2)
  - Contents covered: lec 17-20
- Quiz 2 grades released
  - Max:30, mean: 19.98, median: 20.25, std: 6.46
- AS3 grades will be released
- Project final report and code requirement released
- No OH this week

Reading Materials

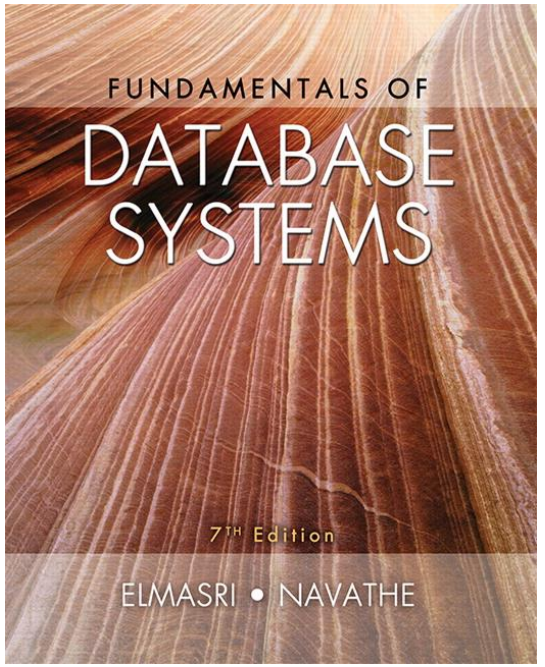
Database Systems: The Complete Book (2nd edition)

- Chapter 20: Parallel and Distributed Databases
- Chapter 24: NOSQL Databases and Big Data Storage Systems



Fundamental of Database Systems (7th Edition)

- Chapter 23: Distributed Database Concepts



Agenda

1. Distributed DBMS
2. NoSQL

1. Distributed DBMS

Design Issues

Databases are hard; distributed DBMSs are even harder…

#1 How to store data in distributed DBMS?

- Partitioning and Replication

#2 Distributed query optimization

- Distributed join algorithms

#3 Distribution transaction

- Two-Phase Commit

#1 Storing Data in a Distributed DBMS

A single relation may be partitioned or fragmented across several sites

- typically at sites where they are most often accessed
- Range, hash, and round-robin partitioning still apply

The data can be replicated as well – when the relation is in high demand or for robustness

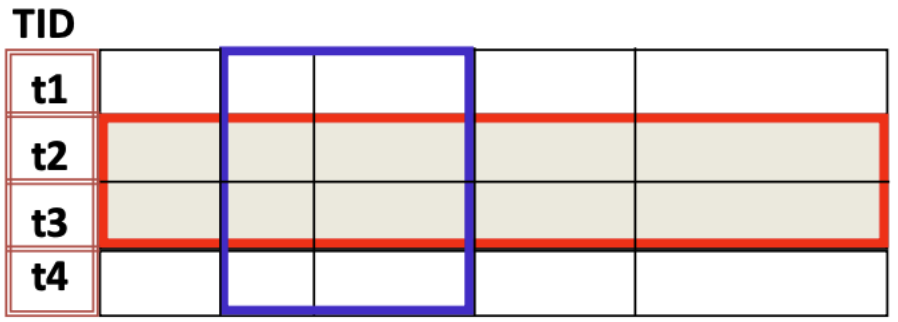
#1 Storing Data in a Distributed DBMS

Horizontal Partitioning (Sharding):

- Usually disjoint
- Can often be identified by a selection query
- To retrieve the full relation, need a union

Vertical Partitioning:

- Identified by projection queries
- Typically unique TIDs added to each tuple
- TIDs replicated in each fragments
- Ensures that we have a Lossless Join



Horizontal Partitioning

What happens if a new node joins?

Example: Hash Partitioning

101	a	XXX	…
102	a	XXY	…
103	b	XYZ	…
104	c	XYX	…
105	d	XZY	…

Hash(a) % 4 = P2

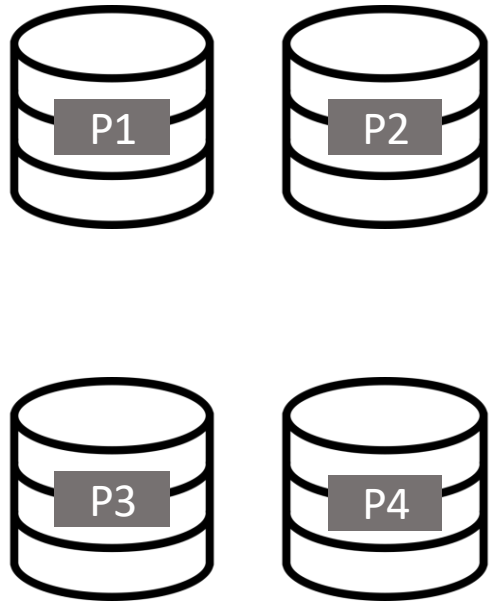
Hash(a) % 4 = P2

Hash(b) % 4 = P4

Hash(c) % 4 = P3

Hash(d) % 4 = P1

Partitions



Horizontal Partitioning

What happens if a new node joins?

Example: Hash Partitioning

101	a	XXX	…
102	a	XXY	…
103	b	XYZ	…
104	c	XYX	…
105	d	XZY	…

Hash(a) % 4 = P2

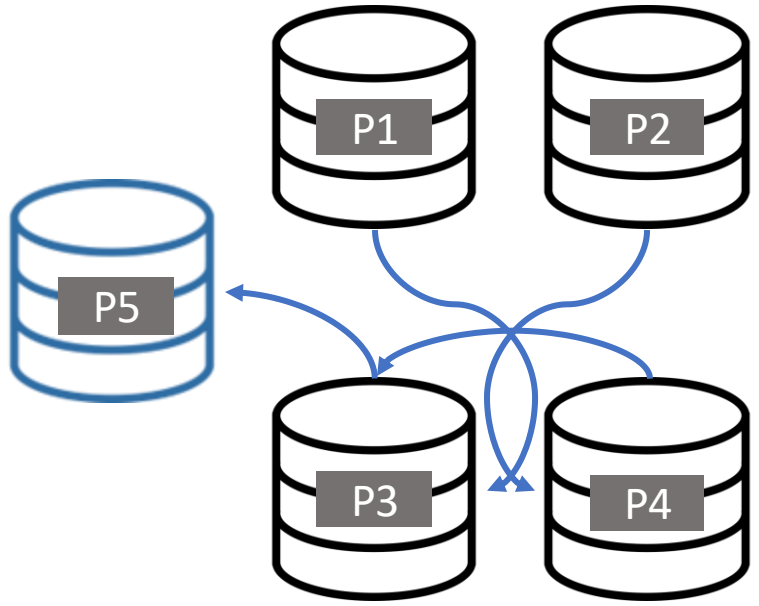
Hash(a) % 4 = P2

Hash(b) % 4 = P4

Hash(c) % 4 = P3

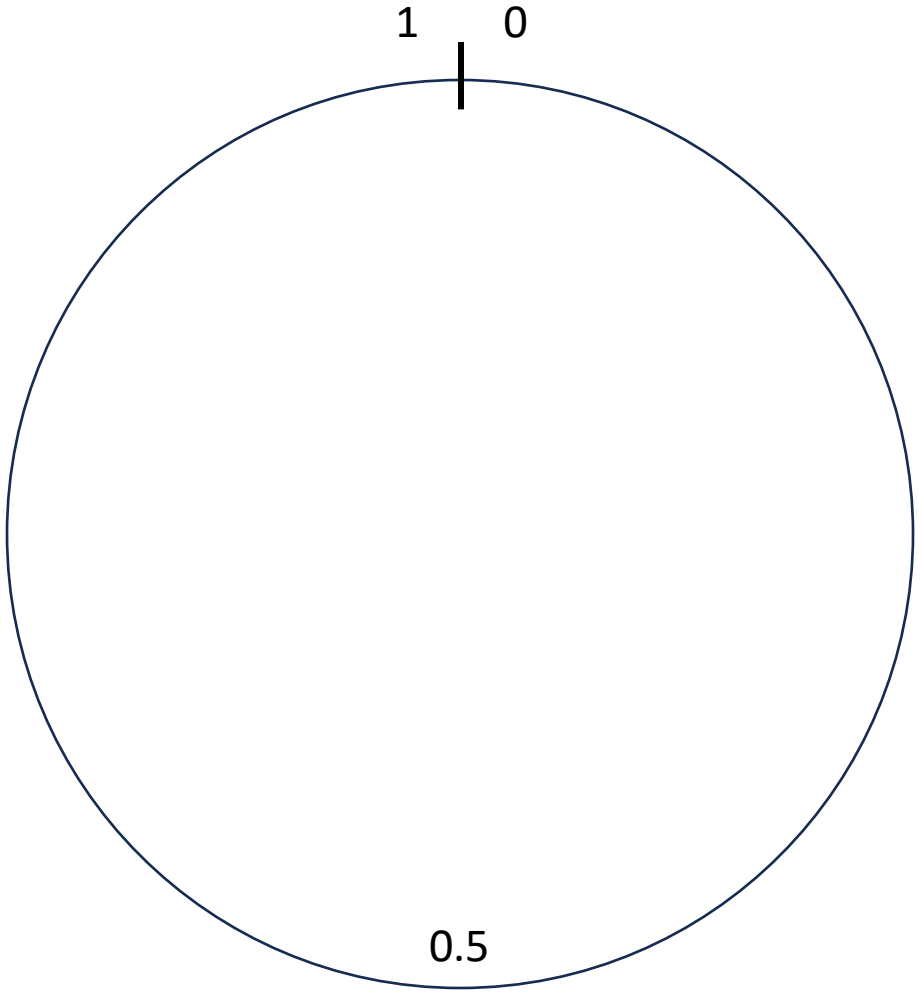
Hash(d) % 4 = P1

Partitions



Consistent Hashing

Key range:
- [0,1]
- mapped to a ring



Consistent Hashing and Random Trees:
Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*

David Karger[1] Eric Lehman[1] Tom Leighton[1,2] Matthew Levine[1] Daniel Lewin[1]
Rina Panigrahy[1]

Abstract

We describe a family of caching protocols for distrib-uted networks that can be used to decrease or eliminate the occurrence of hot spots in the network. Our protocols are particularly designed for use with very large networks such as the Internet, where delays caused by hot spots can be severe, and where it is not feasible for every server to have complete information about the current state of the entire network. The protocols are easy to implement using existing network protocols such as TCP/IP, and require very little overhead. The protocols work with local control, make efficient use of existing resources, and scale gracefully as the network grows.

Our caching protocols are based on a special kind of hashing that we call *consistent hashing*. Roughly speaking, a consistent hash function is one which changes minimally as the range of the function changes. Through the development of good consistent hash functions, we are able to develop caching protocols which do not require users to have a current or even consistent view of the network. We believe that consistent hash functions may eventually prove to be useful in other applications such as distributed name servers and/or quorum systems.

1 Introduction

In this paper, we describe caching protocols for distributed networks that can be used to decrease or eliminate the occurrences of "hot spots". *Hot spots* occur any time a large number of clients wish to simultaneously access data from a single server. If the site is not provisioned to deal with all of these clients simultaneously, service may be degraded or lost.

Many of us have experienced the hot spot phenomenon in the context of the Web. A Web site can suddenly become extremely popular and receive far more requests in a relatively short time than it was originally configured to handle. In fact, a site may receive so many requests that it becomes "swamped," which typically renders it unusable. Besides making the one site inaccessible, heavy traffic destined to one location can congest the network near it, interfering with traffic at nearby sites.

As use of the Web has increased, so has the occurrence and impact of hot spots. Recent famous examples of hot spots on the Web include the JPL site after the Shoemaker-Levy 9 comet struck Jupiter, an IBM site during the Deep Blue-Kasparov chess tournament, and several political sites on the night of the election. In some of these cases, users were denied access to a site for hours or even days. Other examples include sites identified as "Web-site-of-the-day" and sites that provide new versions of popular software.

Our work was originally motivated by the problem of hot spots on the World Wide Web. We believe the tools we develop may be relevant to many client-server models, because centralized servers on the Internet such as Domain Name servers, Multicast servers, and Content Label servers are also susceptible to hot spots.

1.1 Past Work

Several approaches to overcoming the hot spots have been proposed. Most use some kind of replication strategy to store copies of hot pages throughout the Internet; this spreads the work of serving a hot page across several servers. In one approach, already in wide use, several clients share a *proxy cache*. All user requests are forwarded through the proxy, which tries to keep copies of frequently requested pages. It tries to satisfy requests with a cached copy; failing this, it forwards the request to the home server. The dilemma in this scheme is that there is more benefit if more users share the same cache, but then the cache itself is liable to get swamped.

Malpani et al. [6] work around this problem by making a group of caches function as one. A user's request for a page is directed to an arbitrary cache. If the page is stored there, it is returned to the user. Otherwise, the cache forwards the request to all other caches via a special protocol called "IP Multicast". If the page is cached nowhere, the request is forwarded to the home site of the page. The disadvantage of this technique is that as the number of participating caches grows, even with the use of multicast, the number of messages between caches can become unmanageable. A tool that we develop in this paper, *consistent hashing*, gives a way to implement such a distributed cache without requiring that the caches communicate all the time. We discuss this in Section 4.

Chankhunthod et al. [1] developed the Harvest Cache, a more scalable approach using a *tree* of caches. A user obtains a page by asking a nearby leaf cache. If neither this cache nor its siblings have the page, the request is forwarded to the cache's parent. If a page is stored by no cache in the tree, the request eventually reaches the root and is forwarded to the home site of the page. A cache

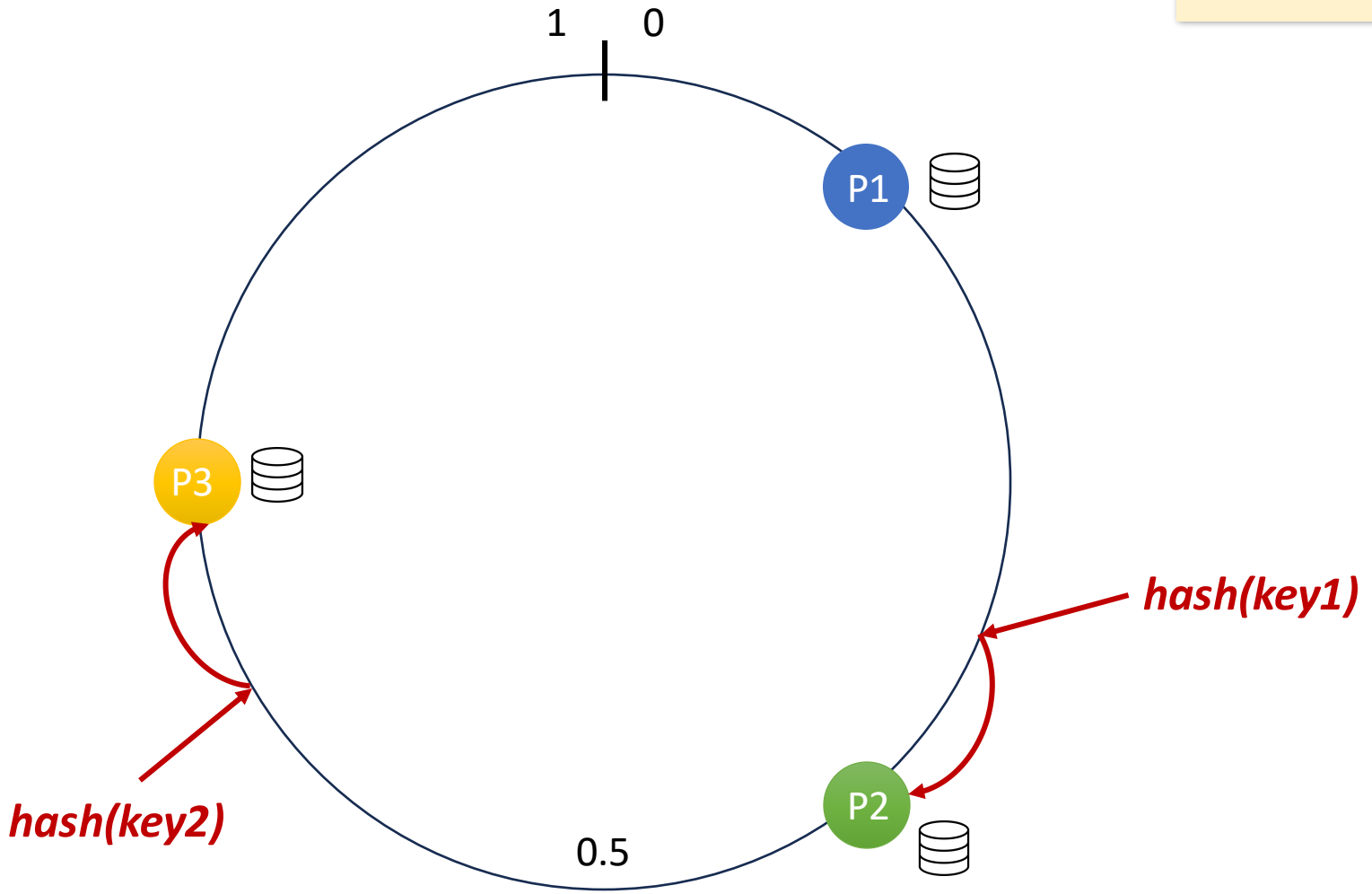
*This research was supported in part by DARPA contracts N00014-95-1-1246 and DABT63-95-C-0009, Army Contract DAAH04-95-1-0607, and NSF contract CCR-9624239

[1]Laboratory for Computer Science, MIT, Cambridge, MA 02139.
email: {karger,e_lehman,danl,ftl,mslevine,danl,rinap}@theory.lcs.mit.edu
A full version of this paper is availble at:
http://theory.lcs.mit.edu/~ {karger,e_lehman,ftl,mslevine,danl,rinap}
[2]Department of Mathematics, MIT, Cambridge, MA 02139

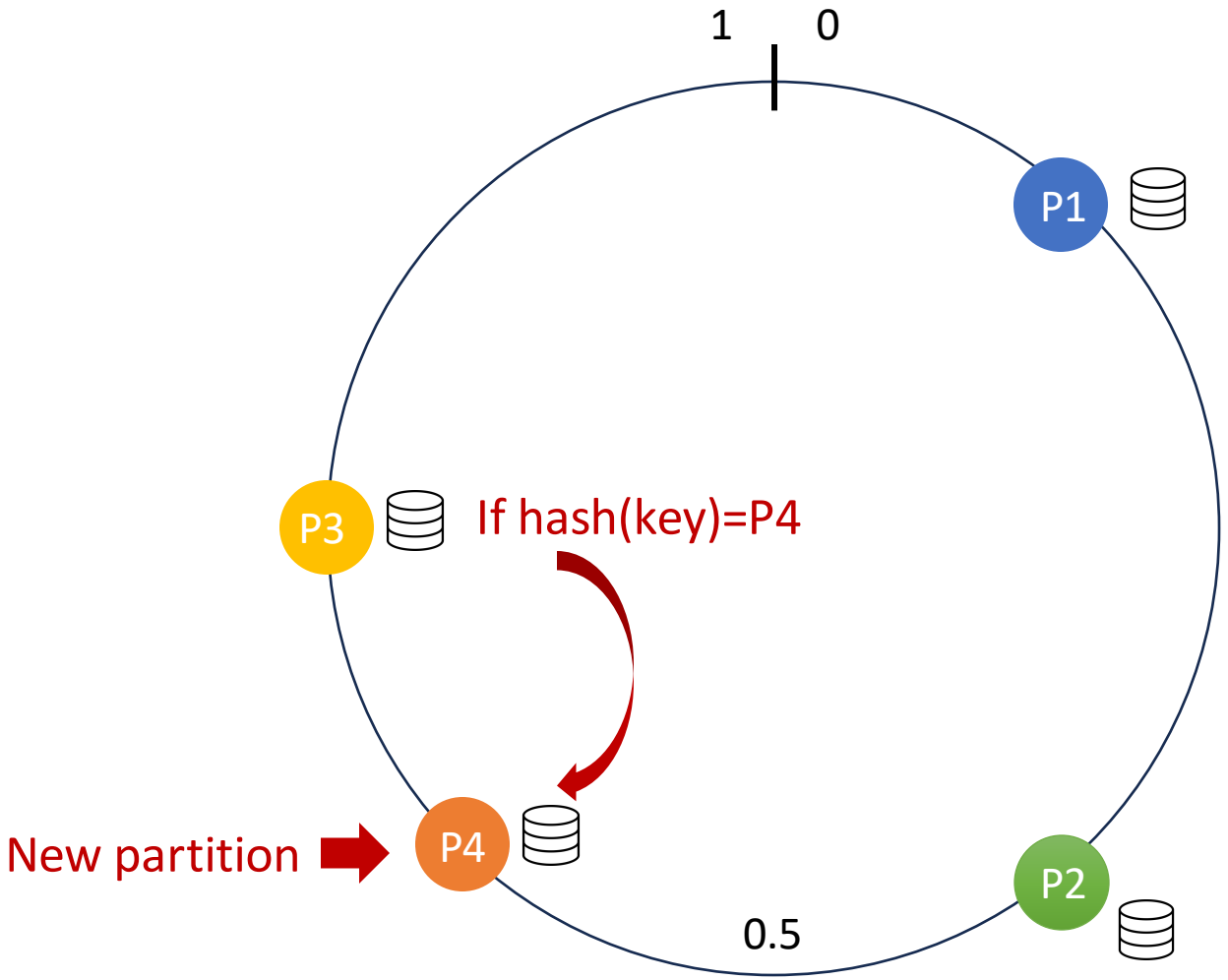
STOC'97

Consistent Hashing

Rule: hash the key onto the ring and assign it to the first node clockwise.



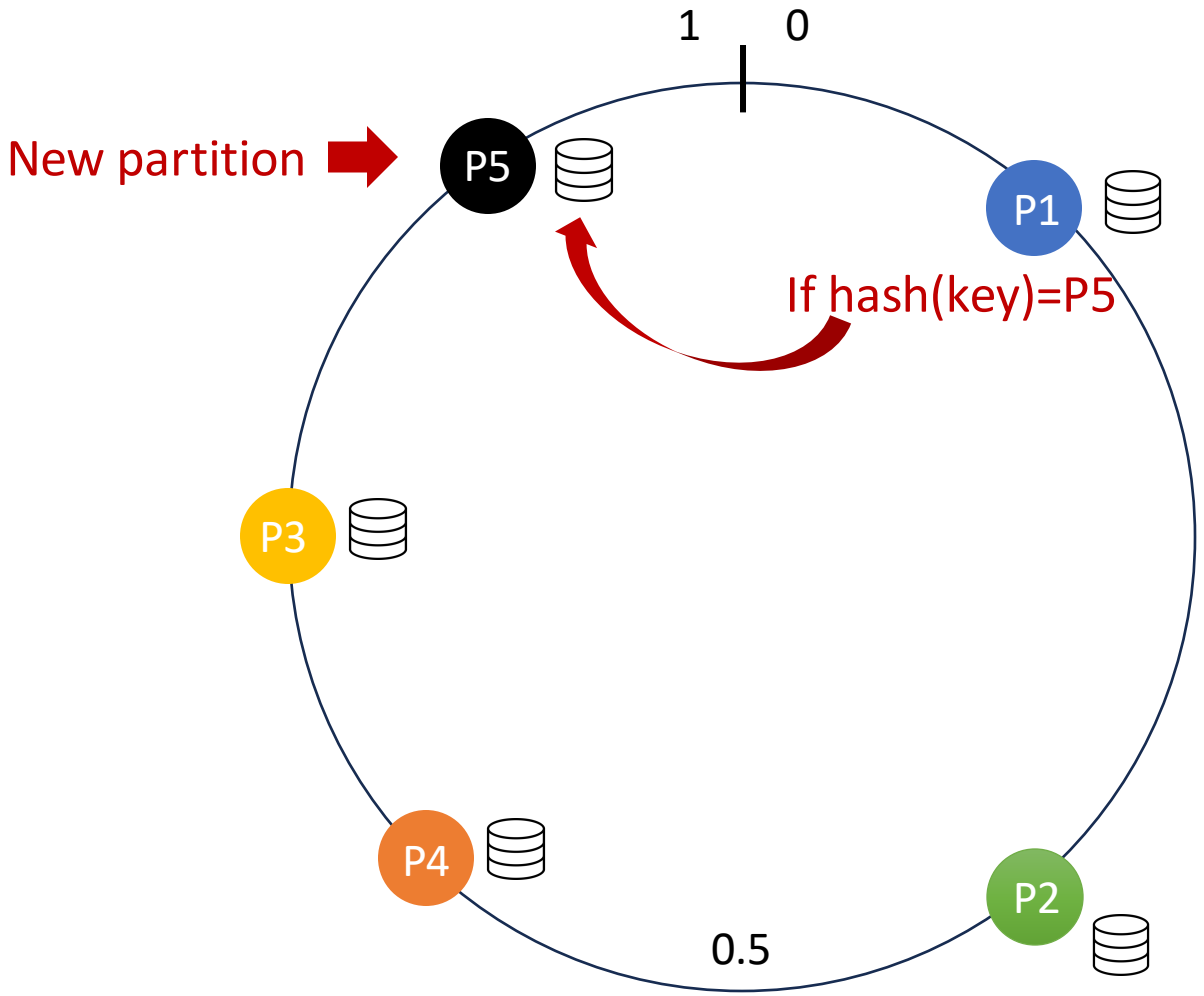
Consistent Hashing



Rule: hash the key onto the ring and assign it to the first node clockwise.

Adding a node only redistributes keys from its clockwise successor.

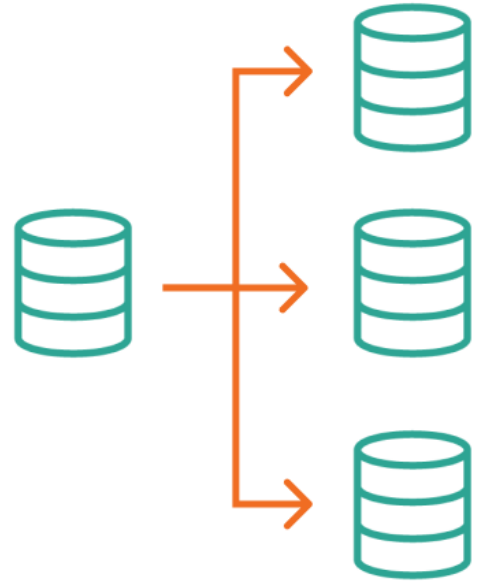
Consistent Hashing



Rule: hash the key onto the ring and assign it to the first node clockwise.

Adding a node only redistributes keys from its clockwise successor.

Replication



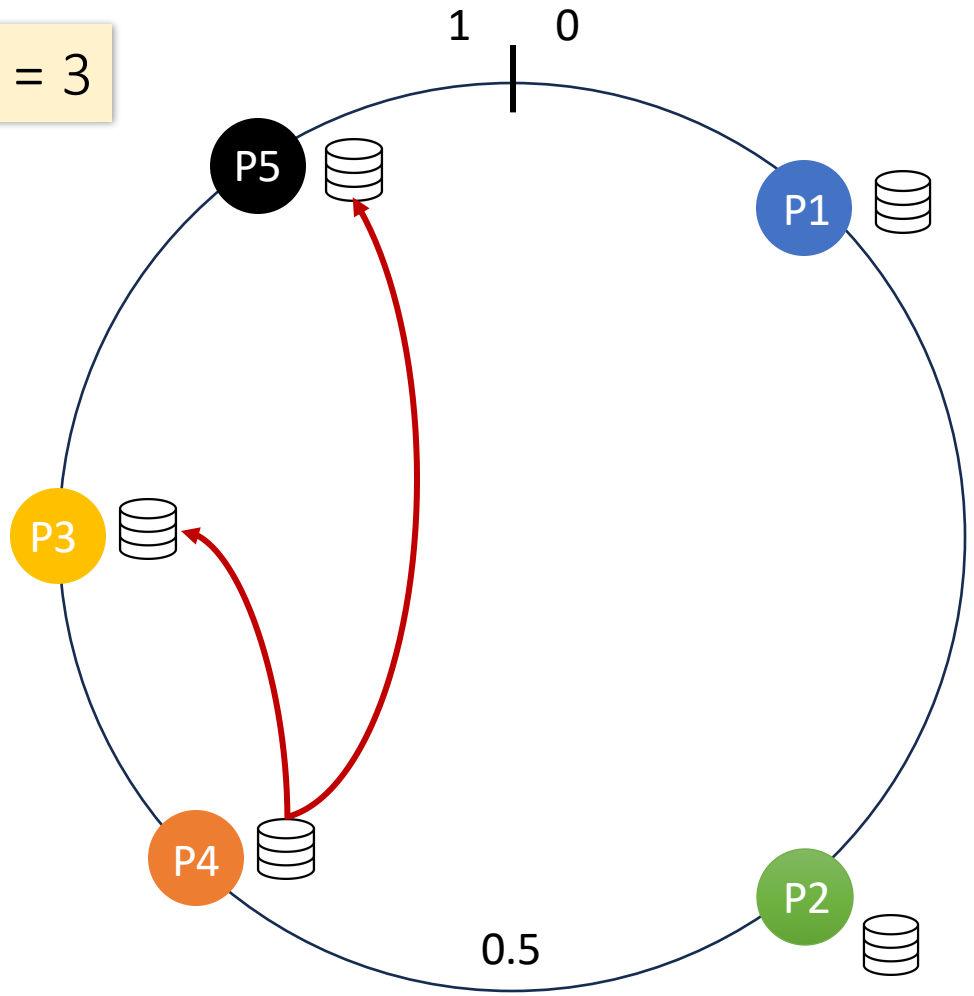
Storing *multiple copies* of a relation

Motivation

- Increased availability: If a site that contains a replica goes down, we can find the same data at other sites
- Faster query evaluation: Queries can execute faster by using a local copy of a relation instead of going to a remote site.

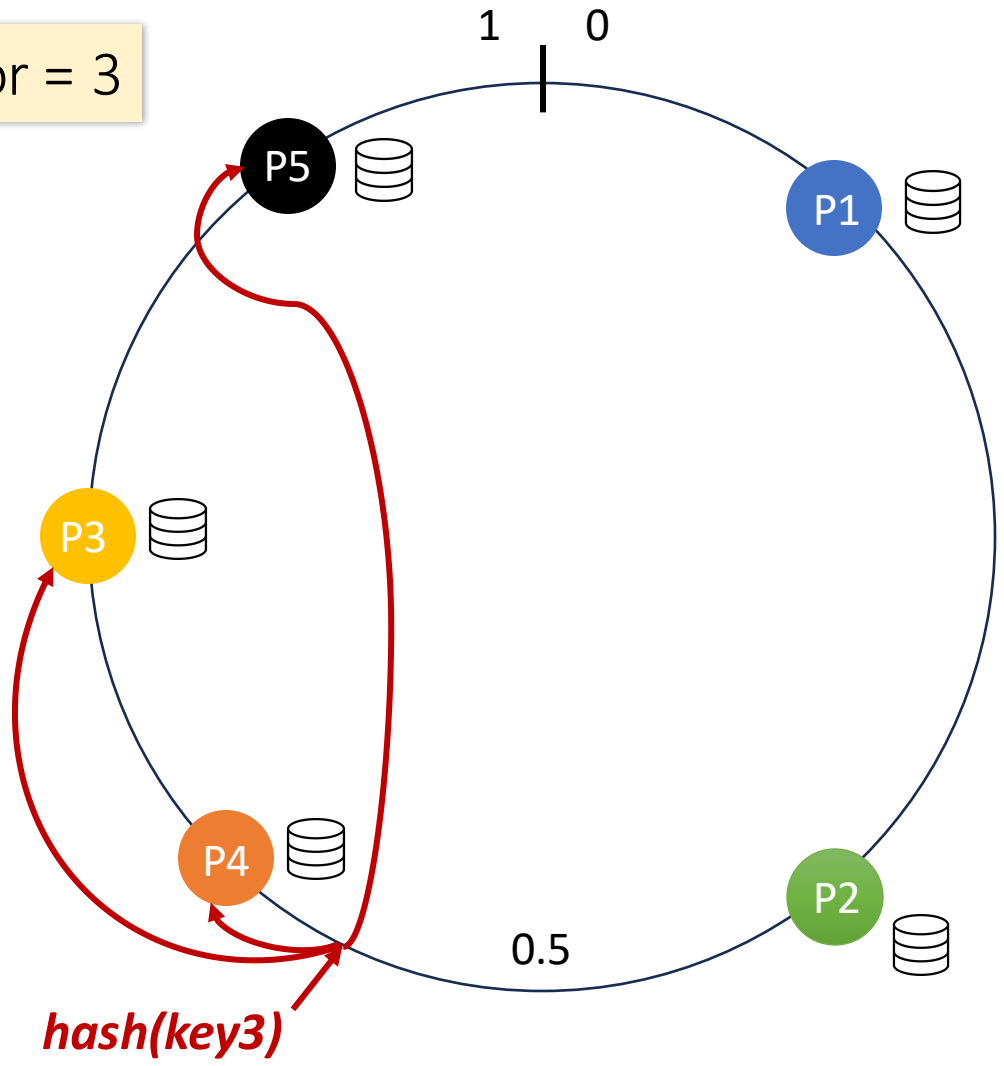
Consistent Hashing

Replication factor = 3



Consistent Hashing

Replication factor = 3



Amazon DynamoDB

cassandra

riak

Updating Distributed Data

Synchronous Replication

- Wait for the required replicas to acknowledge the write before reporting success.
- **Tradeoff:** Stronger guarantees, but higher write latency and potentially lower availability.

Asynchronous Replication

- Acknowledge the write before all replicas have been updated.
- **Tradeoff:** Lower write latency, but replicas may temporarily have stale data.

#2 Distributed Query Optimization

Same goal as centralized query optimization: find an efficient execution plan.

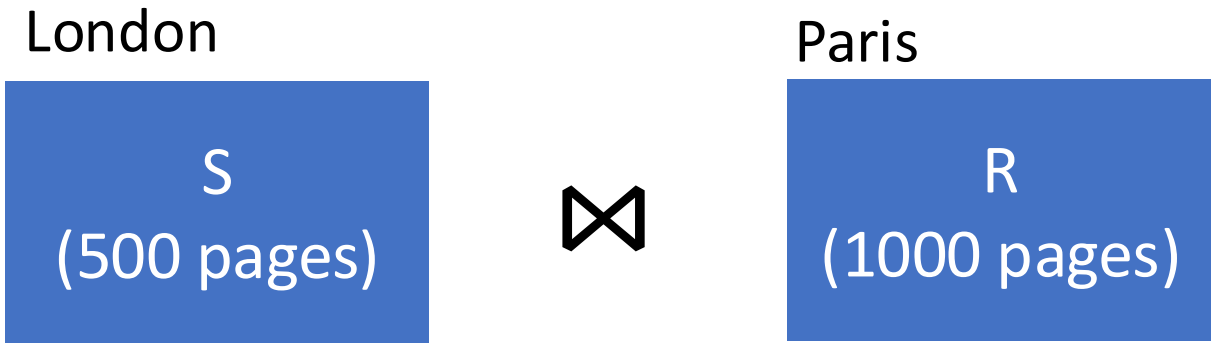
Additional considerations in a distributed DBMS:

- **Communication cost:** Minimize data transferred between machines.
- **Data locality:** Execute operations close to where data is stored.
- **Load balancing:** Distribute work across machines to avoid bottlenecks.

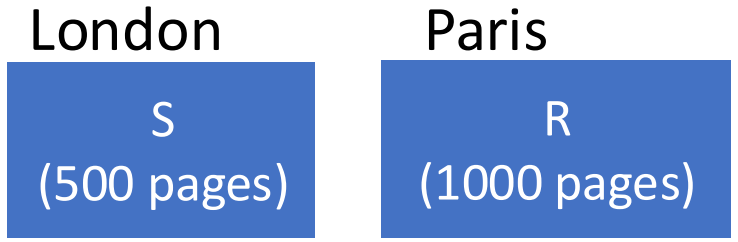
Joins in Distributed DBMS

Can be very expensive if relations are stored at different sites!

Goal: Ship R to London, then do join with S at London.



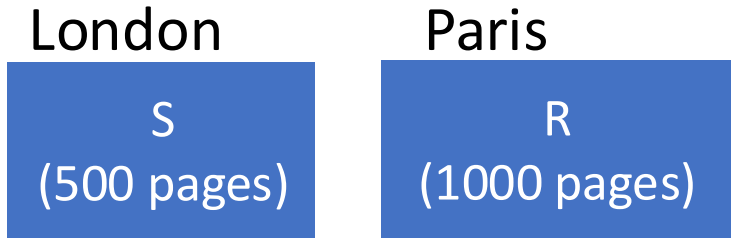
Semijoin (⋉)



1. At London, project S onto join columns and ship this to Paris
2. At Paris, join S-projection with R
   - Result is the reduction of R w.r.t. S (only these tuples are needed)
3. Ship reduction of R to back to London
4. At London, join S with reduction of R

Tradeoff the cost of computing and shipping projection for cost of shipping full R relation

Bloomjoin



1. At London, compute a bit-vector of some size k:
   - Hash column values into range 0 to k-1
   - If some tuple hashes to p, set bit p to 1 (p from 0 to k-1)
   - Ship bit-vector to Paris
2. At Paris, hash each tuple of R similarly
   - discard tuples that hash to 0 in S's bit-vector
   - Result is called reduction of R w.r.t S
3. Ship "bit-vector-reduced" R to London
4. At London, join S with reduced R

#3 Distributed Transaction

A given transaction is submitted at one site, but it can access data at other sites.

- When a transaction is submitted at some site, the transaction manager at that site breaks it up into a collection of *sub-transactions* that can be executed at different sites.

Task: Transfer $100 from Account A to Account B

Machine 1: Deduct $100 from A
Machine 2: Add $100 to B

What happens if Machine 2 crashes after Machine 1 completes its update?

Distributed Transaction

A given transaction is submitted at one site, but it can access data at other sites.

- When a transaction is submitted at some site, the transaction manager at that site breaks it up into a collection of *sub-transactions* that can be executed at different sites.

New questions:

- Distributed CC: How can locks for objects stored across several sites be managed?
- Distributed Atomic Commit: How to ensure transaction atomicity when data is distributed across sites?

Distributed Atomic Commit

New issues:

- New kinds of failure, e.g., network/remote site failures
- If “sub-transactions” of a transaction execute at different sites, all or none must commit

Need a commit protocol to achieve this

- Goal: Ensures atomicity (all-or-nothing commits) for transactions spanning multiple sites in a distributed database.
- Most widely used: Two Phase Commit (2PC)

This is not to be confused with 2PL!

Two-phase Commit (2PC)

Distributed transaction protocol that ensures all participants in a distributed system either commit or abort a transaction atomically

Site at which transaction originates is coordinator

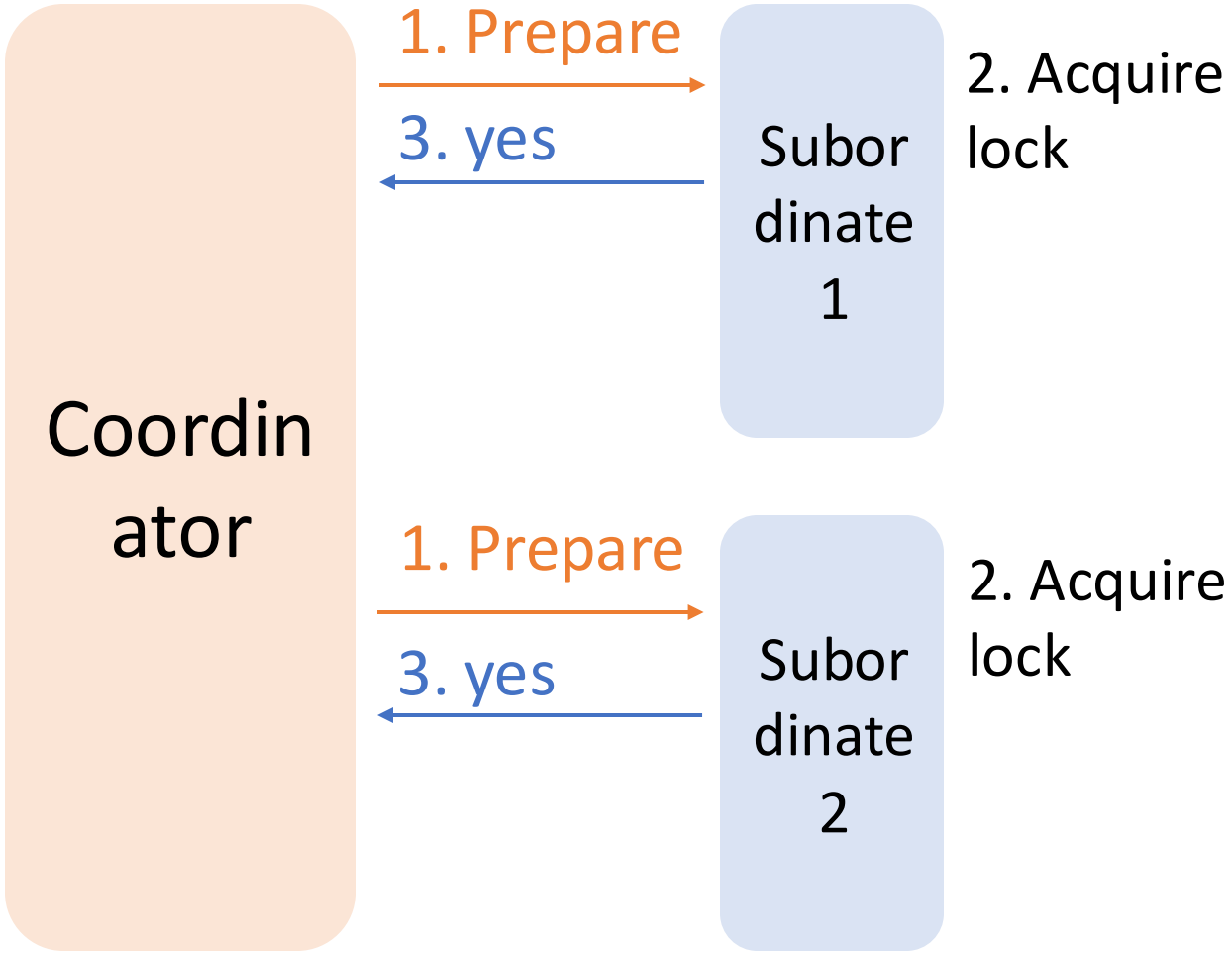
Other sites at which it executes are subordinates

- w.r.t. coordination of this transaction

A log is maintained at each site – as in a centralized DBMS – commit protocol actions are additionally logged.

Two-phase Commit (2PC)

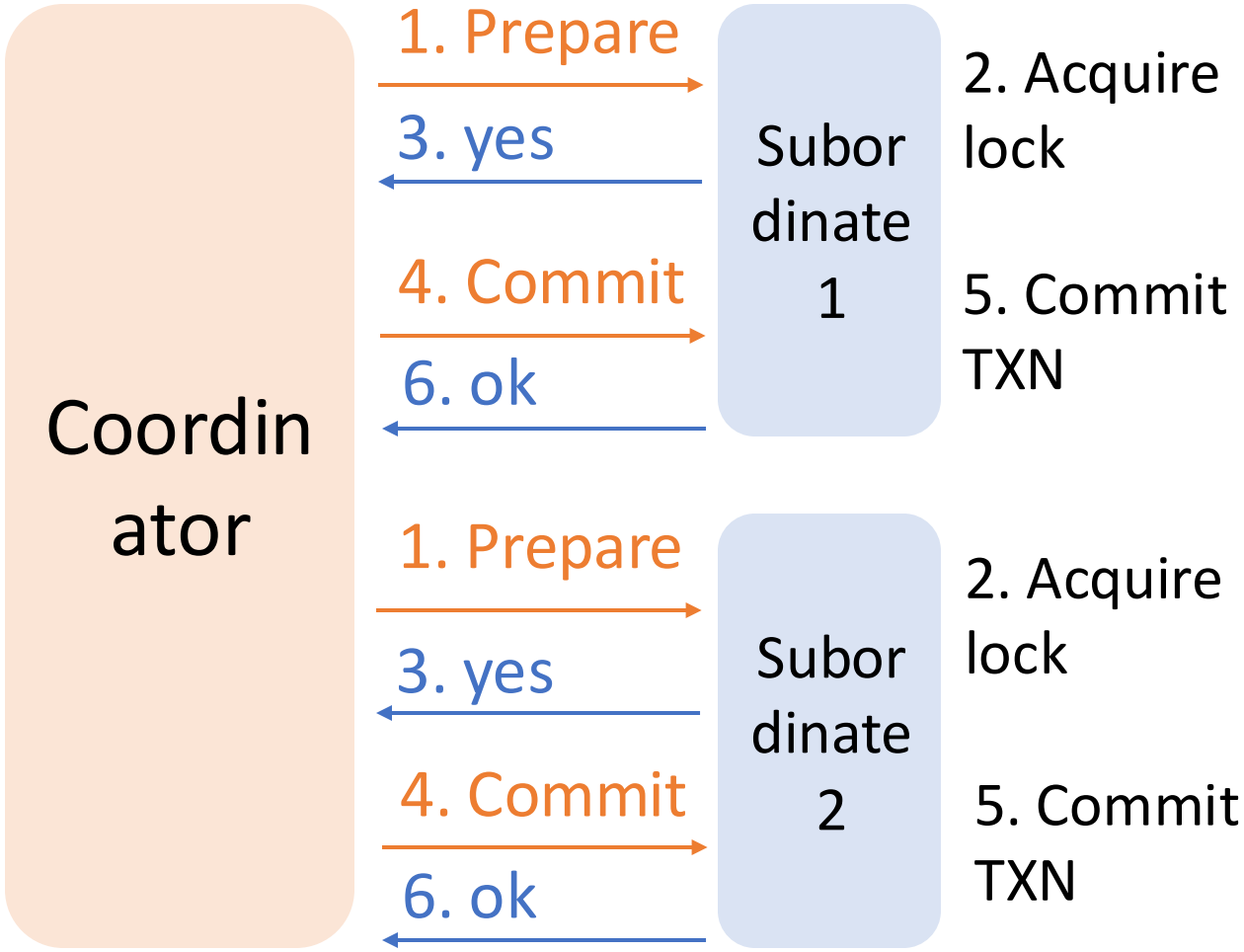
Prepare Phase: asking other nodes whether they can commit the proposed transaction.



Two-phase Commit (2PC)

Prepare Phase: asking other nodes whether they can commit the proposed transaction.

Commit Phase: commanding other nodes to either commit or abort the proposed transaction



2PC comments

Two rounds of communication

- voting => termination
- Both initiated by the coordinator

Any site (coordinator or subordinate) can unilaterally decide to abort a transaction

- but consensus needed to commit

Every message reflects a decision by the sender

- to ensure that this decision survives failures, it is first recorded in the local log and is force-written to disk

Weakness of 2PC

2PC is often called a **“blocking”** atomic commit protocol (or **“anti-availability”** protocol) because all nodes/members must be up for it to work.

- In particular, if a coordinator dies, all nodes would have to wait to hear the final decision (commit or abort)
- Participant nodes cannot simply time out and abort the transaction because it promised to follow the final decision from the coordinator.

2. NoSQL

NoSQL

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

- Many early systems 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.

#1 Data model
- Relational vs key-value, graph etc.
- Schema flexibility

#2 Consistency guarantee
- Tradeoff between consistency and availability

#3 Key Index structure
- Read vs write optimized

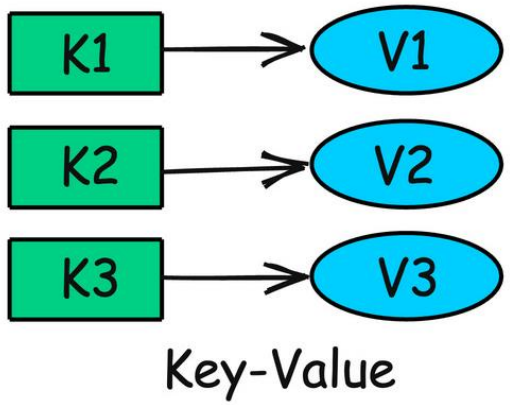
Example NoSQL systems

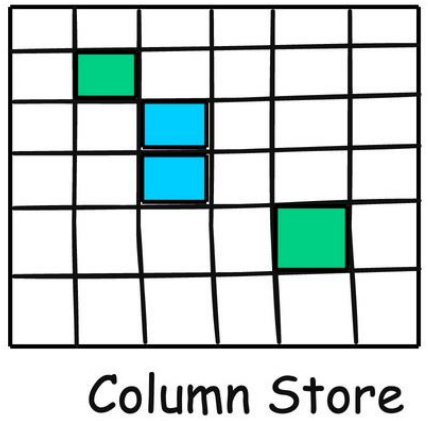
Key-value store:
DynamoDB, Redis

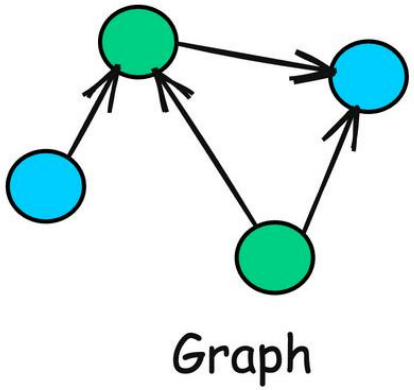
Document store:
MongoDB, CouchDB

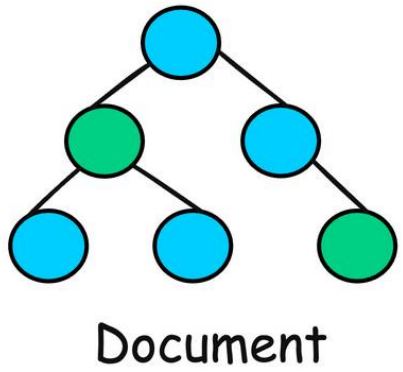
Wide-column stores:
Cassandra, BigTable

Graph database:
Neo4j, AWS Neptune









The Cost of Coordination

Recall ACID for RDBMS desired properties of transactions

Challenge: maintaining ACID across multiple servers requires coordination

- Coordination = latency + reduced availability

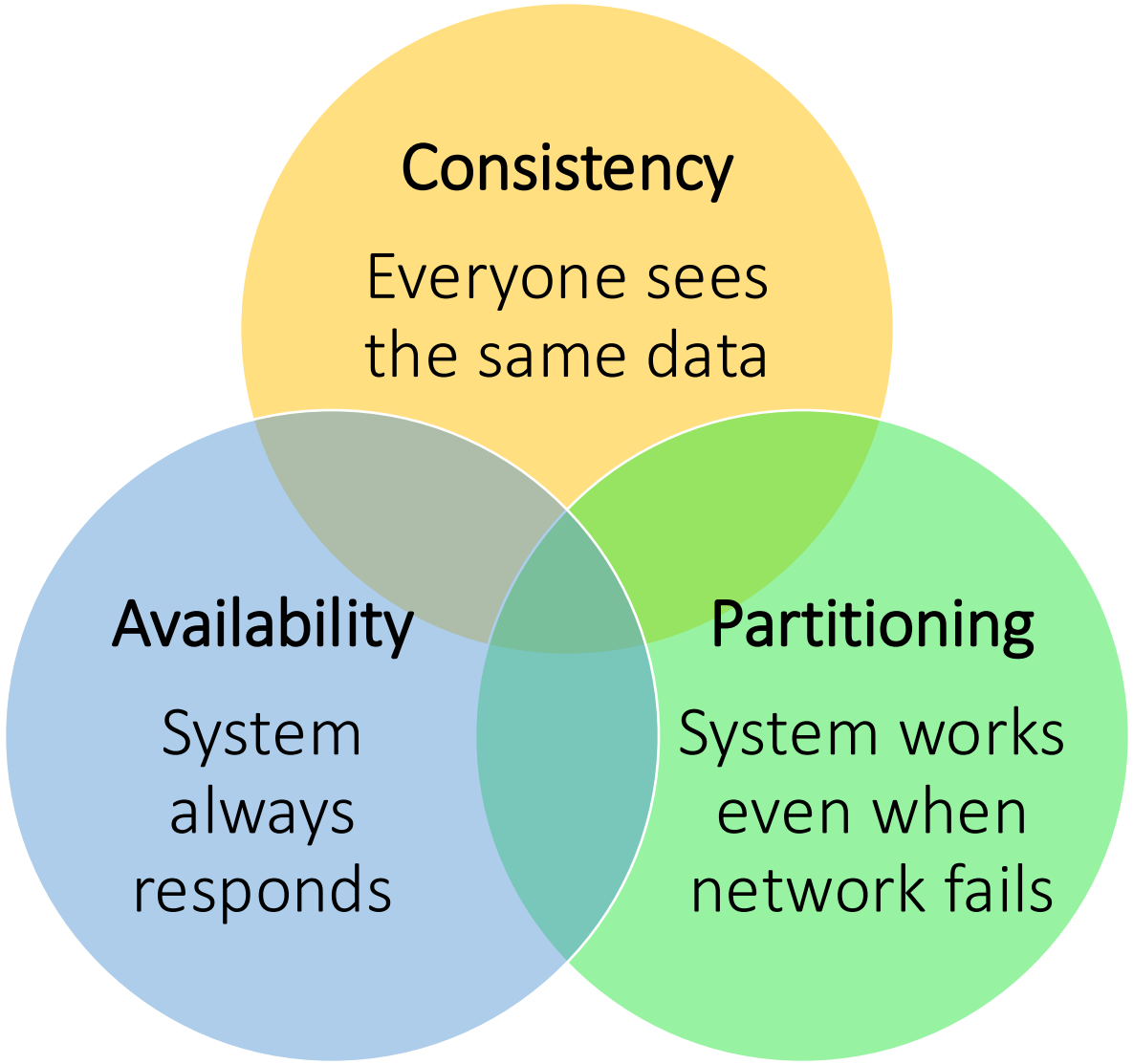
Example: adding an item to shopping cart

- 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"

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)

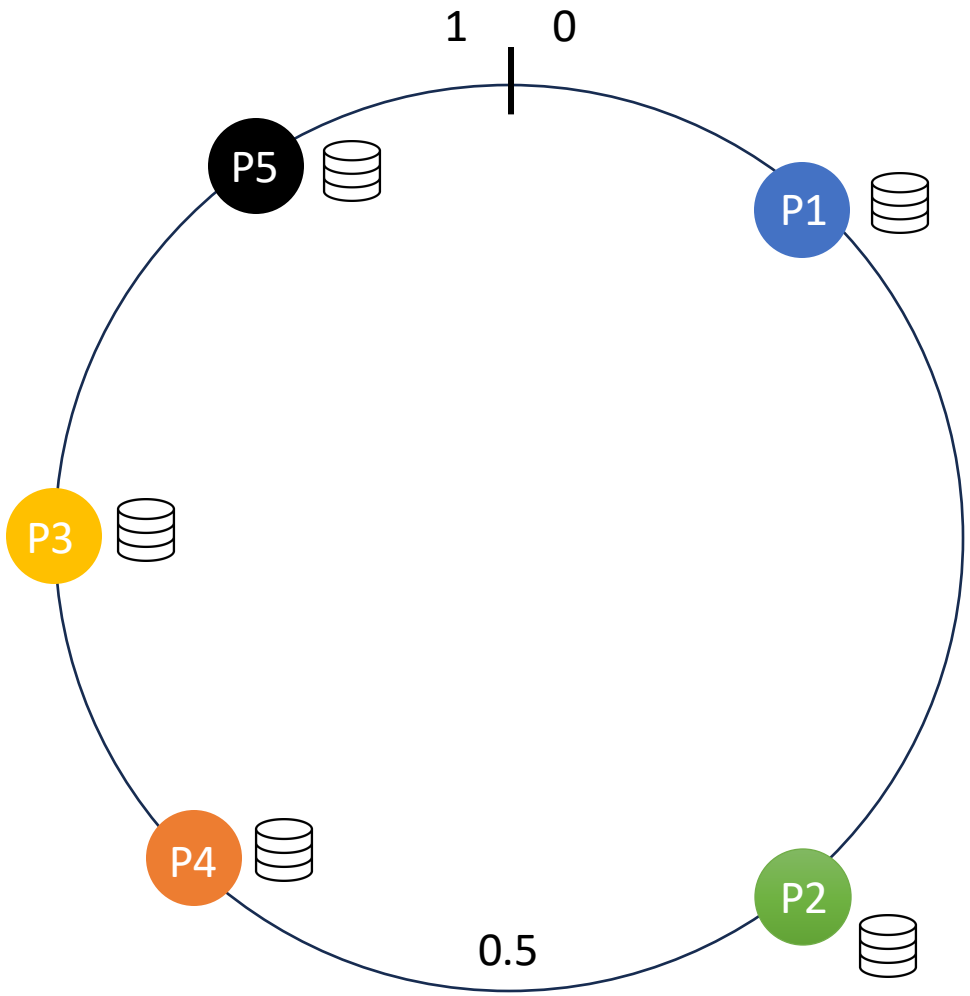
Image source: https://medium.com/nerd-for-tech/understand-cap-theorem-751f0672890e

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
- Better load balancing
- 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)

- R + W > N guarantees you read at least one up-to-date copy
  - 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

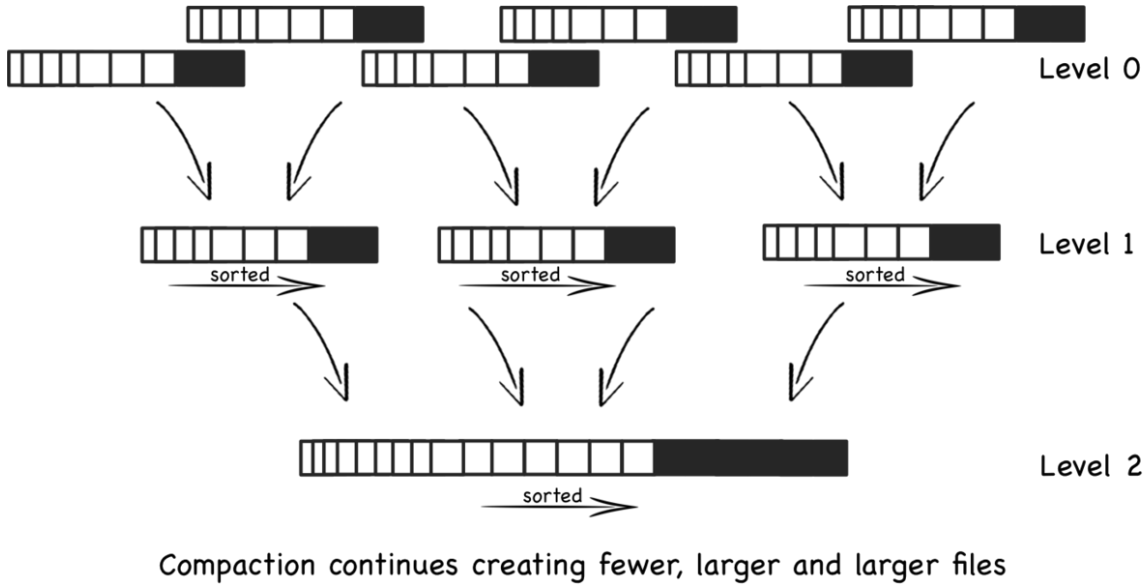


Image source: https://en.wikipedia.org/wiki/Log-structured_merge-tree

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

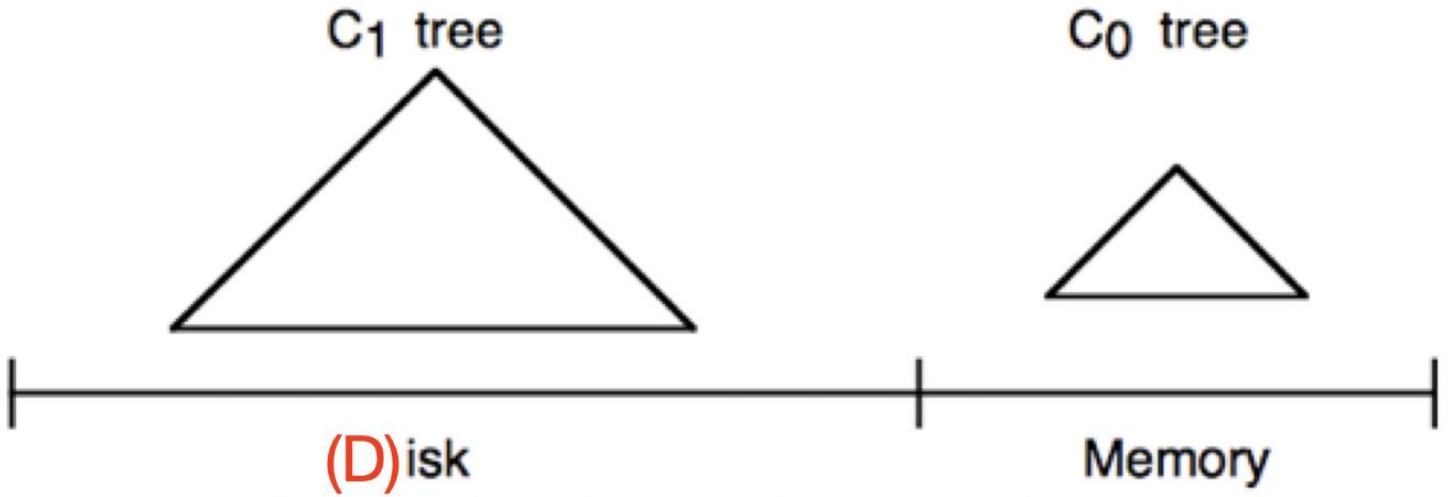


Figure 2.1. Schematic picture of an LSM-tree of two components

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

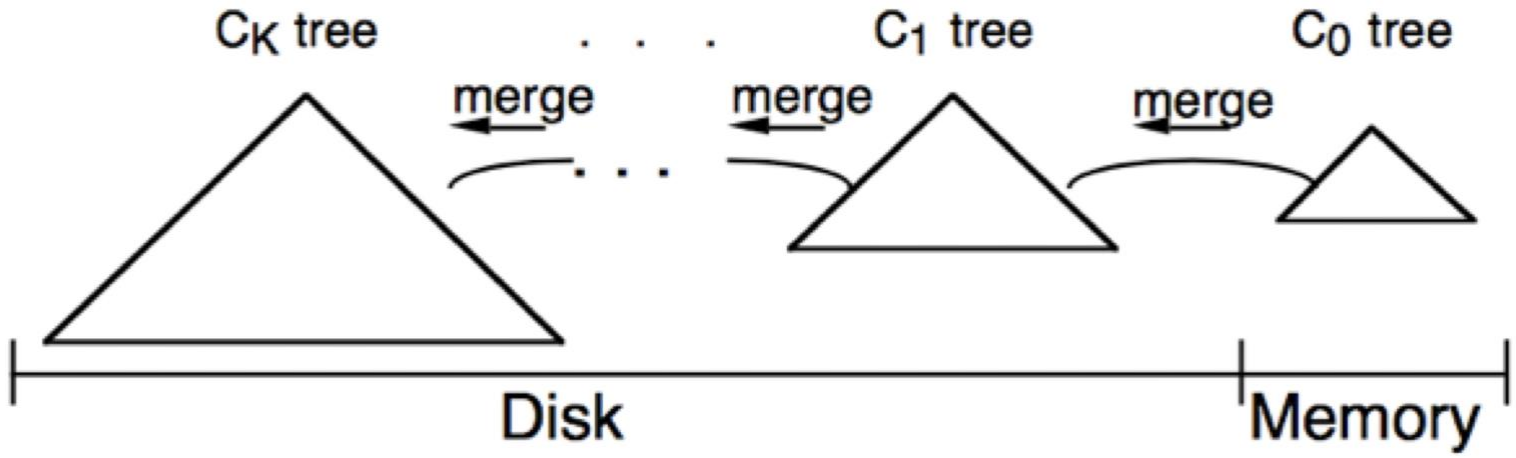


Figure 3.1. An LSM-tree of K+1 components

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

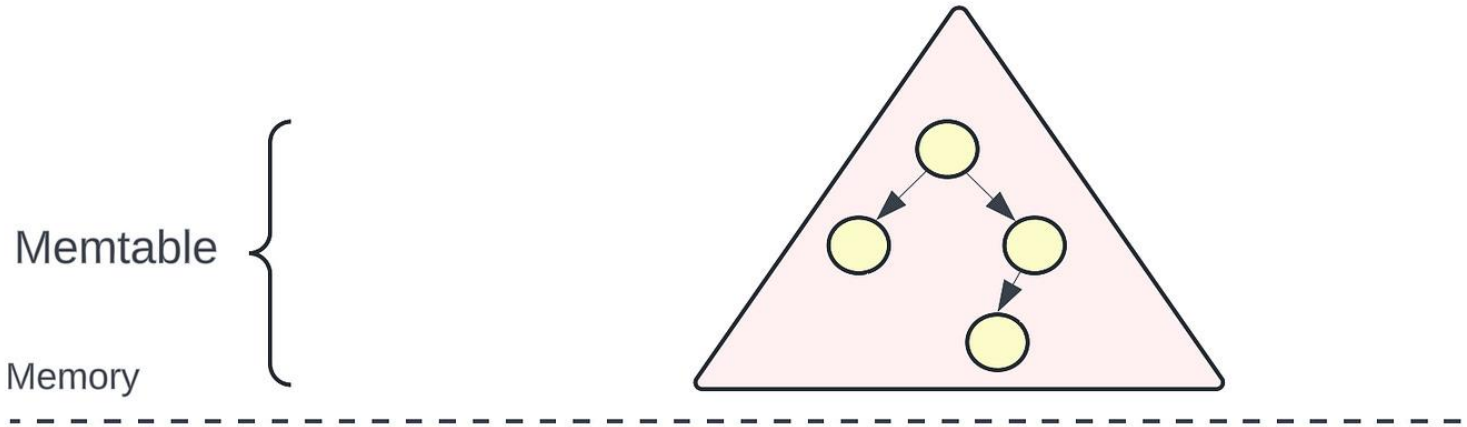


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

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

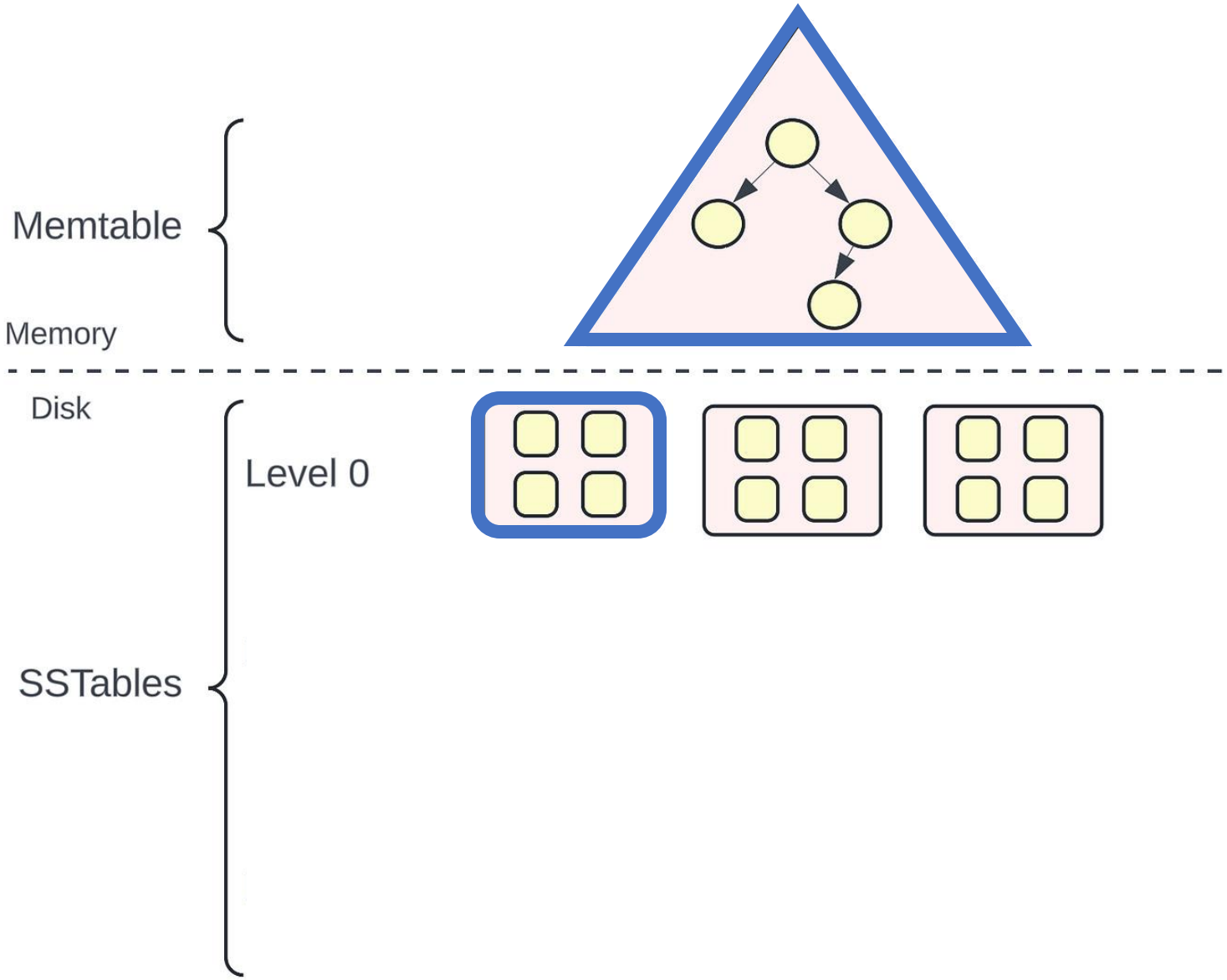


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

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

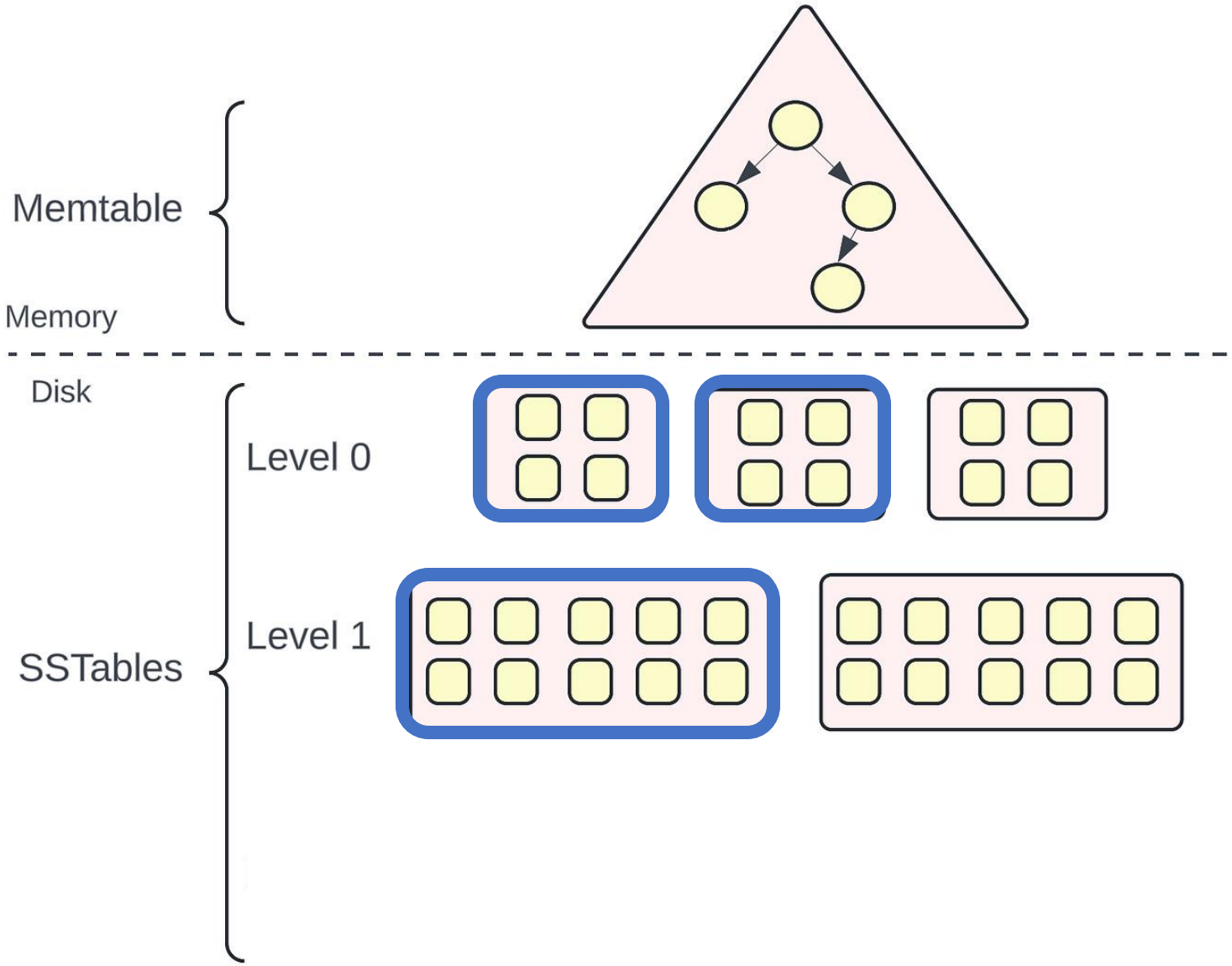


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

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

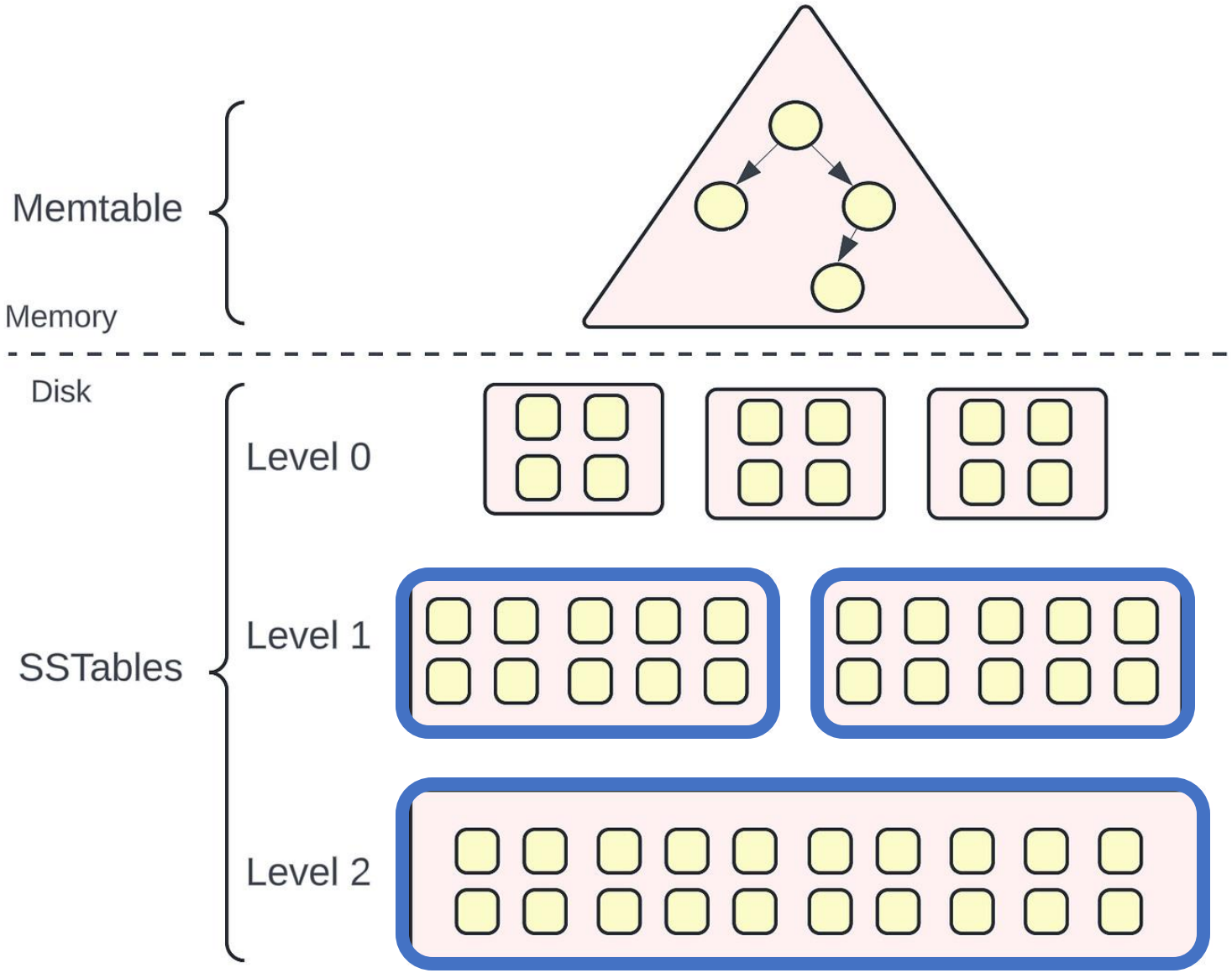


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

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)

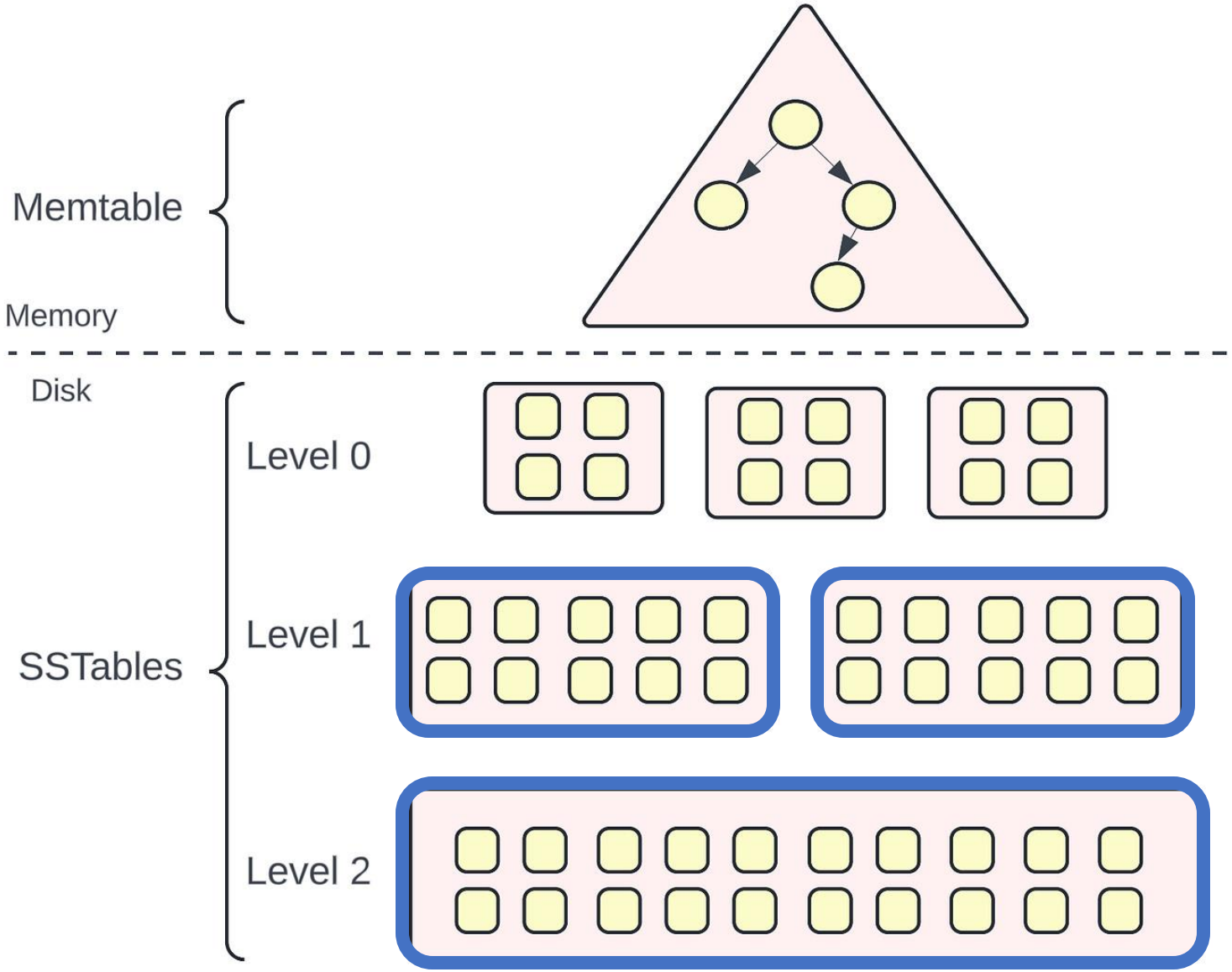


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

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