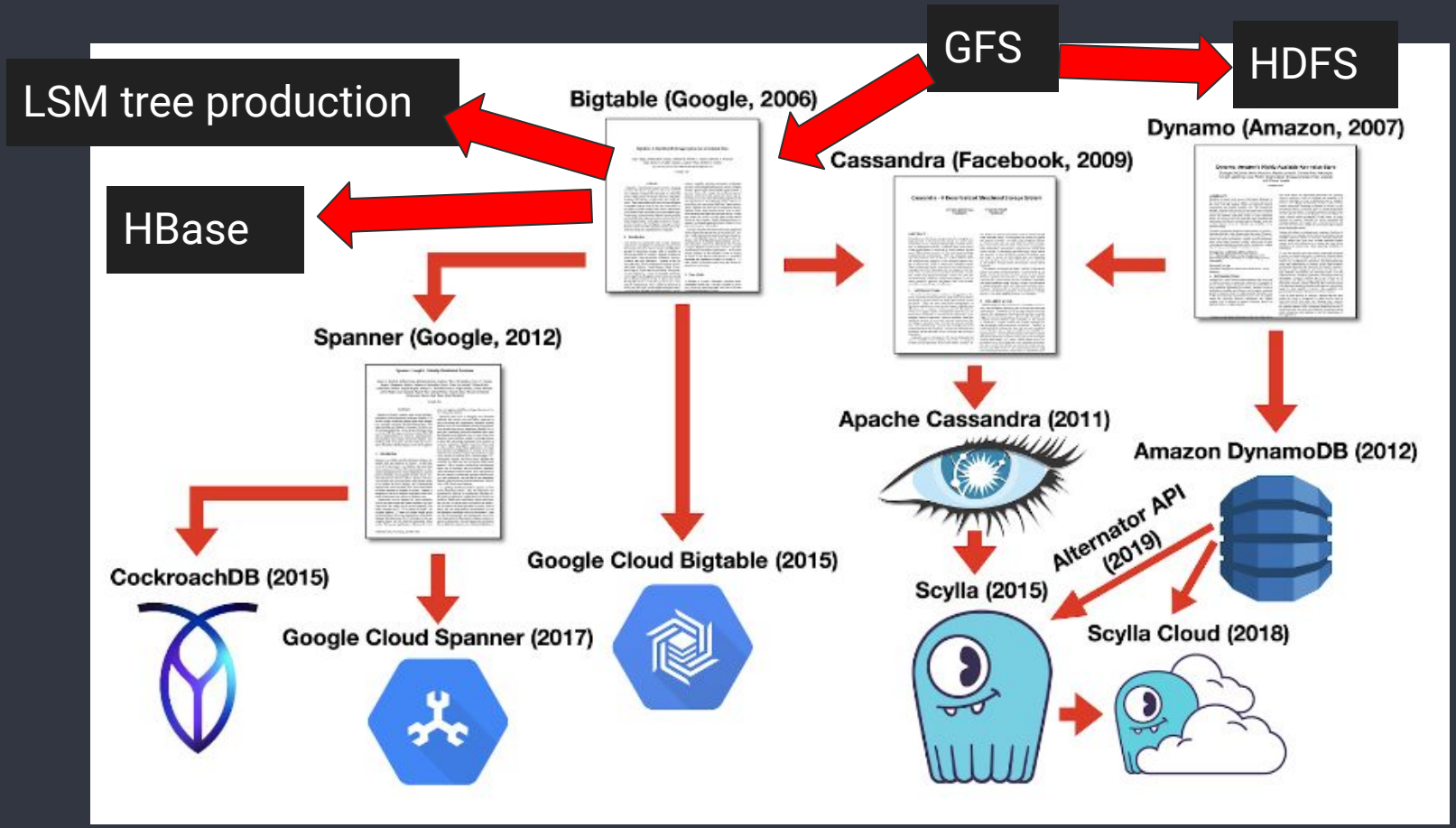
Dynamo, Google Bigtable, GFS, and Borg Case Study

ChiLin Chiou

May 2024 @ NYCU Distributed Systems Course



https://www.scylladb.com/learn/nosql/nosql-database-comparison/

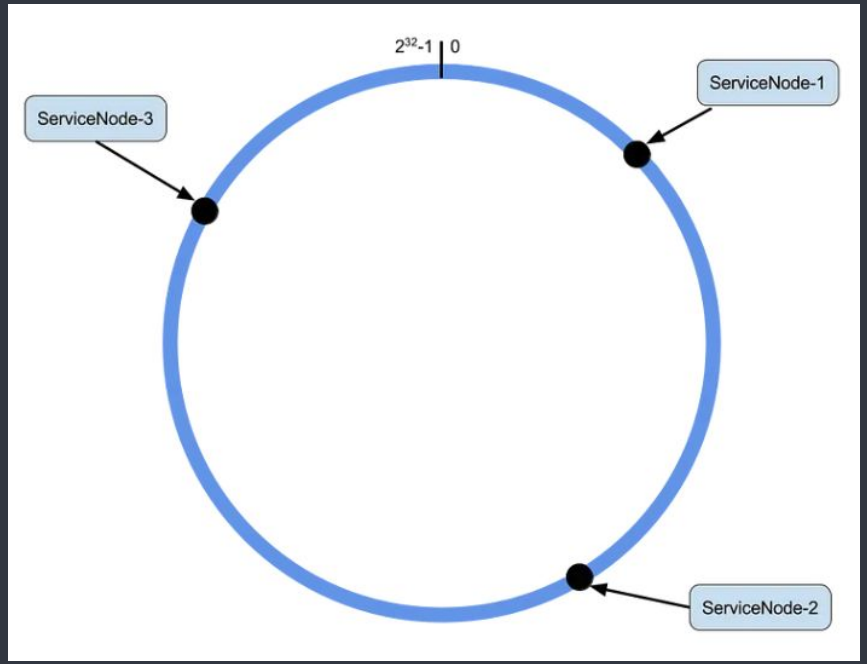
Dynamo: Amazon's Highly Available Key-value Store

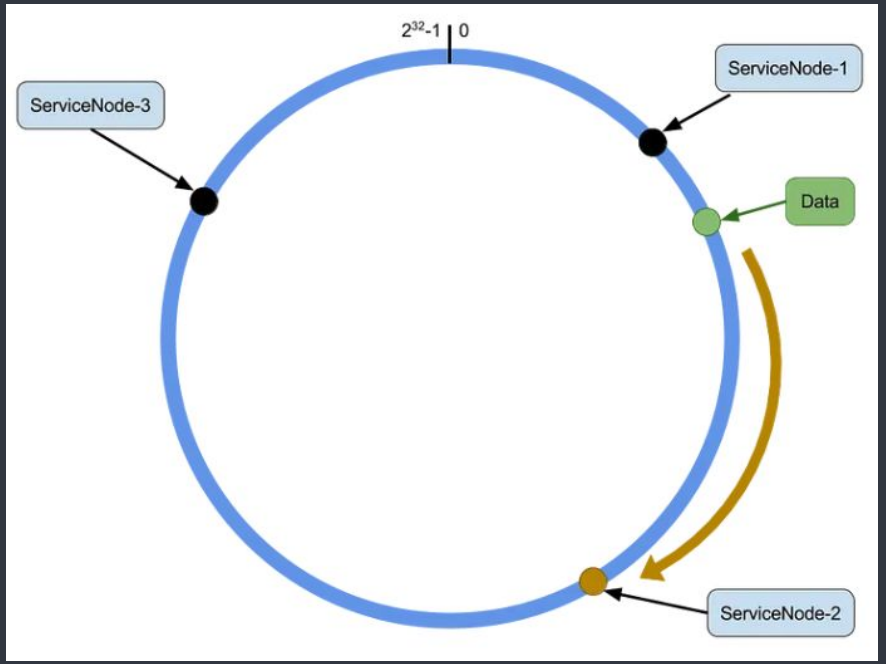
Dynamo

- Problem: Supporting High-Throughput Writes
- Designed to preserve writes as much as possible
- ~~RDBMS~~
  - The available replication technologies are limited and choose consistency over availability
  - Not easy to scale-out databases or use smart partitioning schemes for load balancing.
- K-V server
- AP system
- Eventually consistent
- Leaderless
- Asynchronous replication

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Partitioning - Consistent Hashing

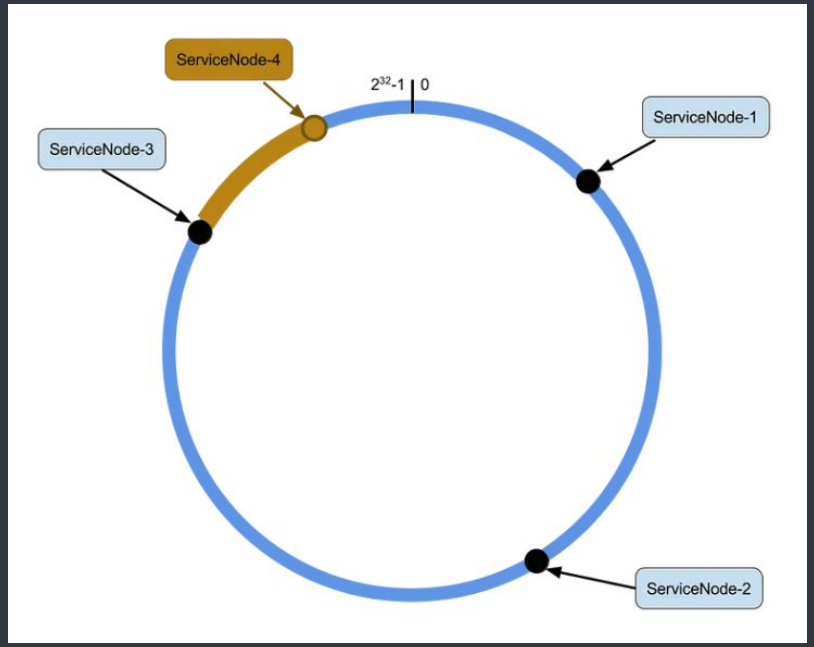




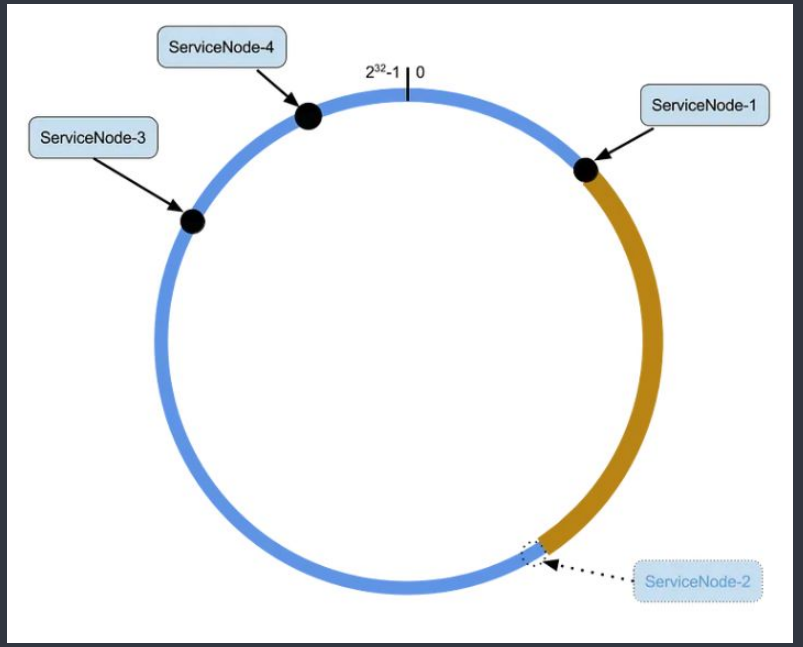
Partitioning - Traditional Hashing

- Expect the hash function to minimize collisions as much as possible
- Hash function: hash(key) mod N
  - Consider exist 10 node in cluster, for key hash(key) = 123456, key on node 6
  - Consider add a node (N=11), key on node 3
  - Consider add a node (N=12), key on node 0
- Two key points
  - Scalable Independence: As the number of nodes increases, the impact on other nodes is minimized as much as possible
  - Load Balancing: Workload across all nodes to prevent any single node from becoming a bottleneck

Partitioning - Consistent Hashing



新增節點的情況

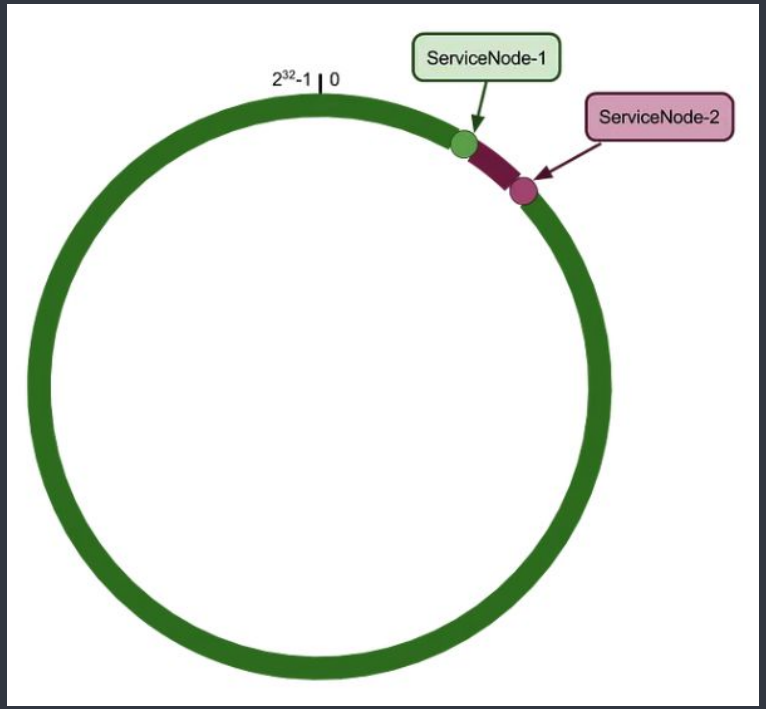


刪除節點的情況

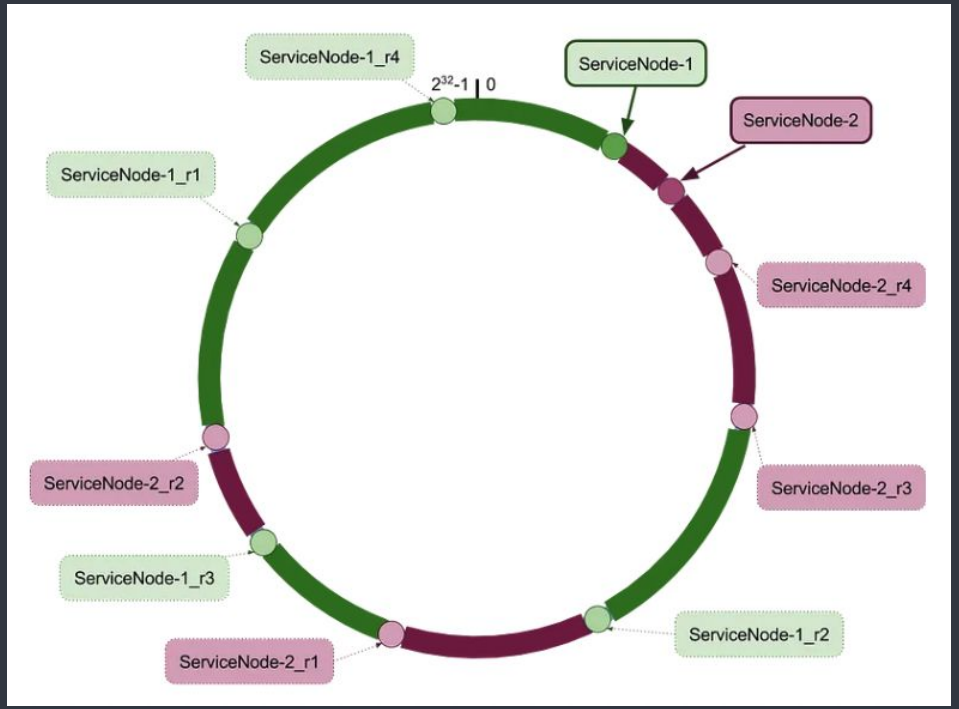
Membership management

- A gossip-based protocol propagates membership changes

Partitioning - Consistent Hashing



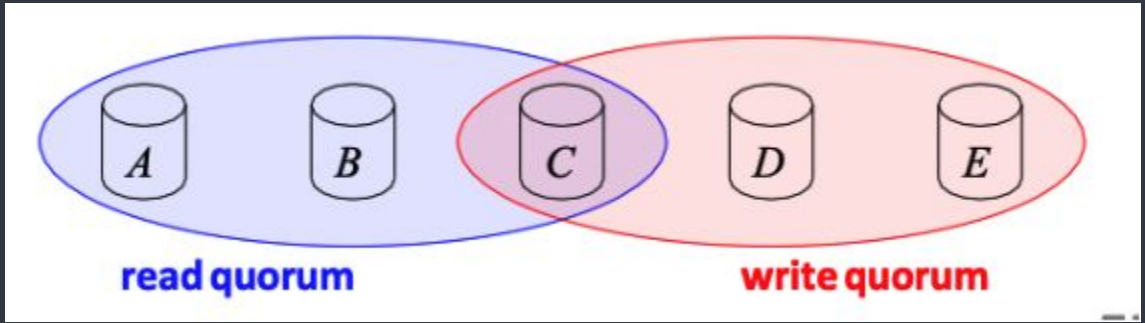
"Skew" of Consistent Hashing



Solved by virtual nodes

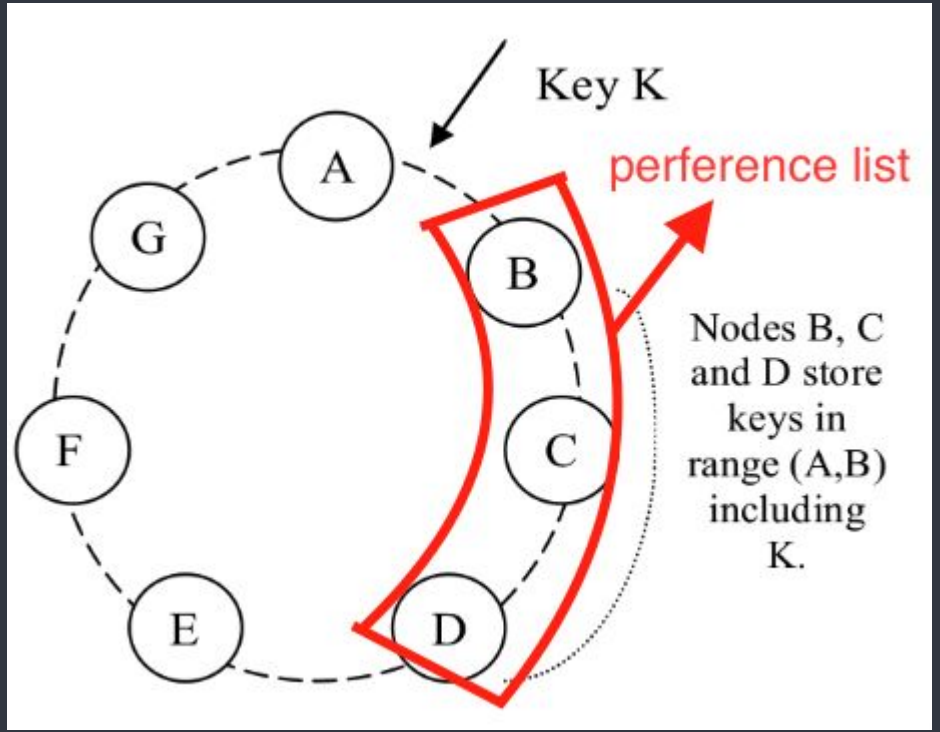
Replication: Strict Quorum System

- R: The minimum number of nodes that must participate in a successful read operation.
- W: The minimum number of nodes that must participate in a successful write operation.
- R + W > N yields a quorum-like system.
- Using Strict Quorum System Sloopy



Replication

- The ***coordinator*** replicates these keys at the N-1 clockwise successor nodes in the ring
- List of nodes that is responsible for storing a particular key is called the ***preference list***.
- Skipping positions in the ring to ensure that the list contains only distinct physical nodes.
- All read and write operations are performed on the first N healthy nodes from the preference list.
    - Not always be the first N nodes encountered while walking the consistent hashing ring if node in preference list is not health



Eventually consistent

- Read repair
- Hinted Handoff
- Anti-entropy

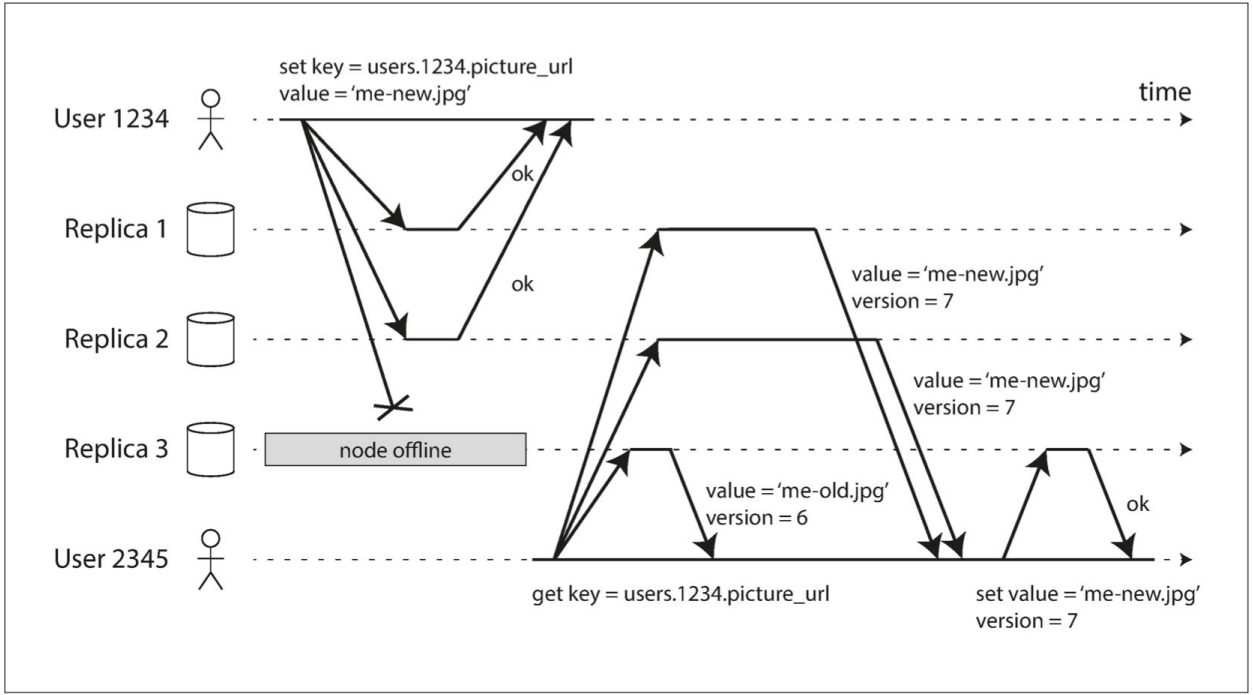
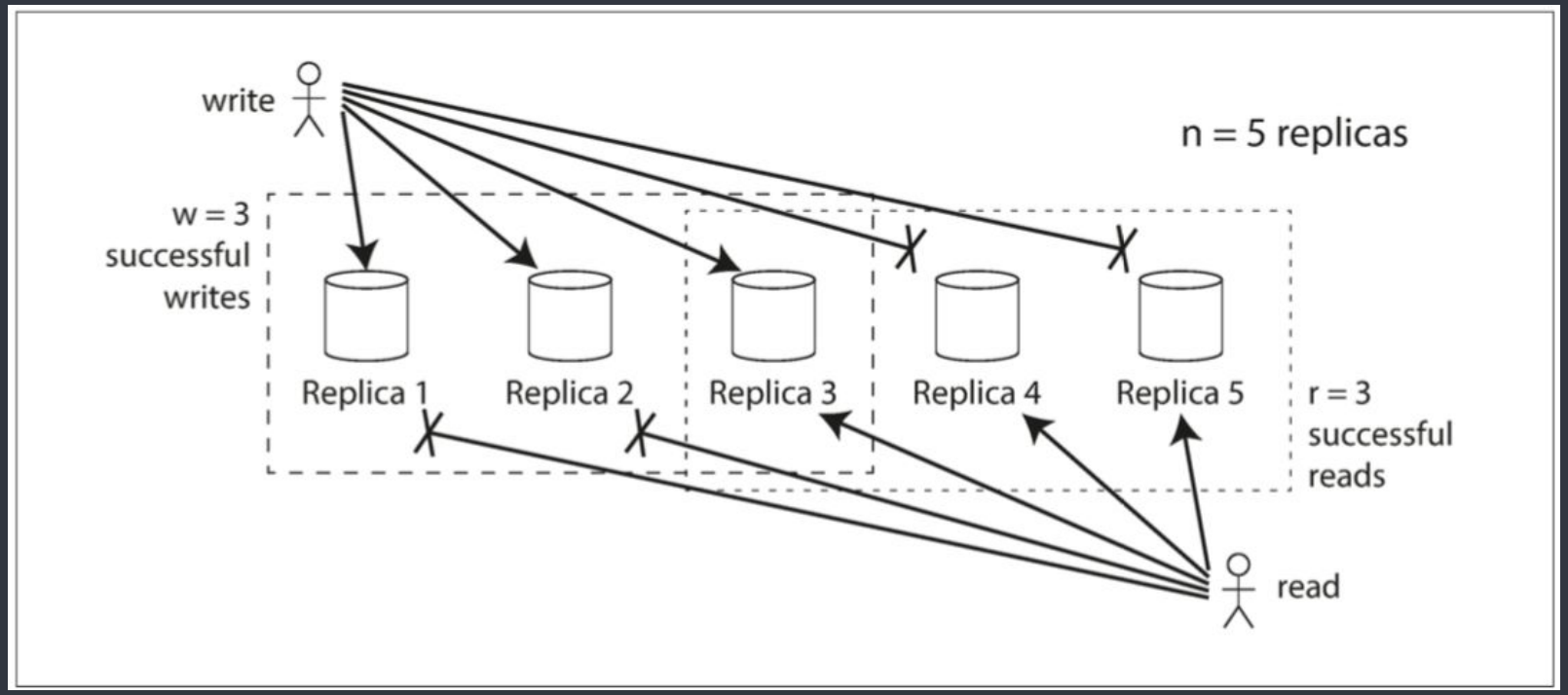


Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

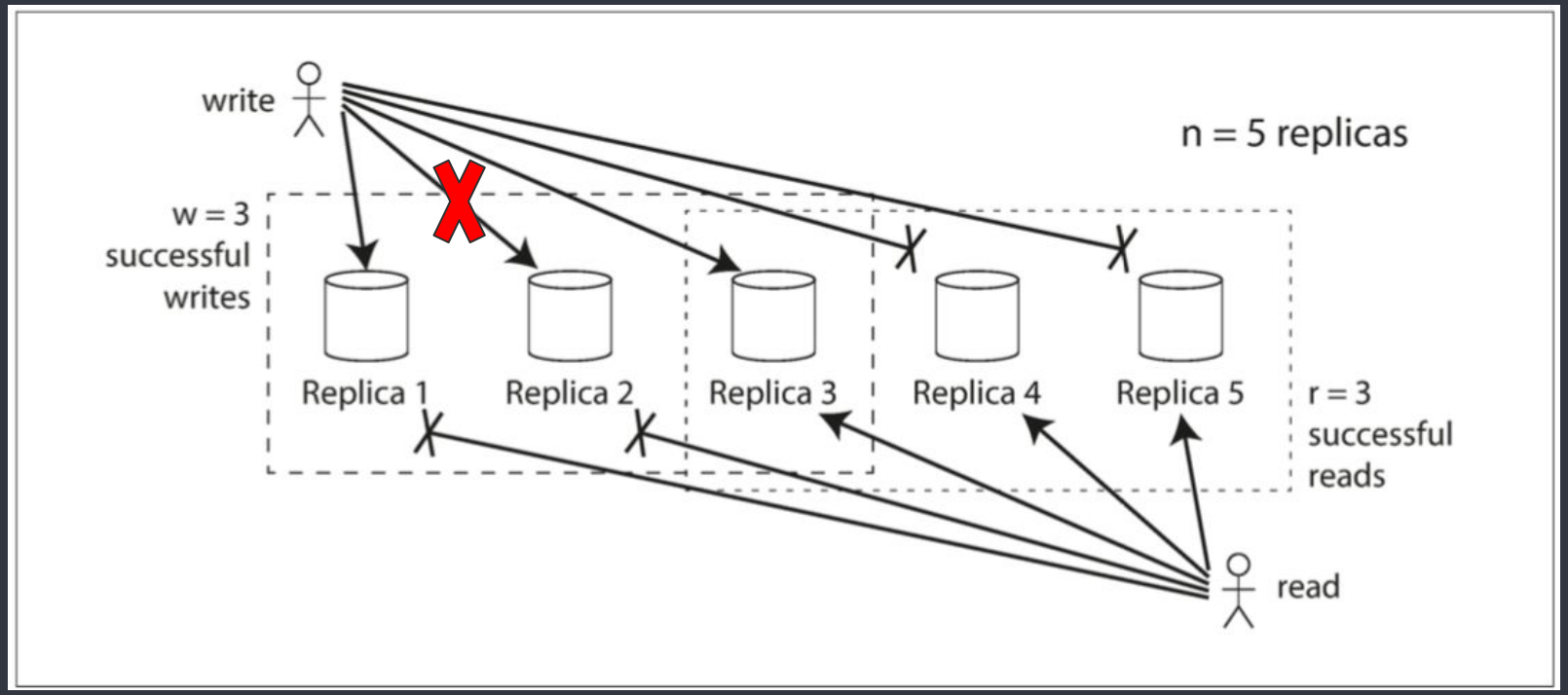
Fault tolerance

- Temporary and Transient Error Handling: Anticipate that nodes will return to normal operation shortly.
  - Hinted handoff
  - Sloppy quorum
- Long-term and Permanent Error Handling: Nodes may never recover.
  - anti-entropy (replica synchronization)
  - Merkle tree

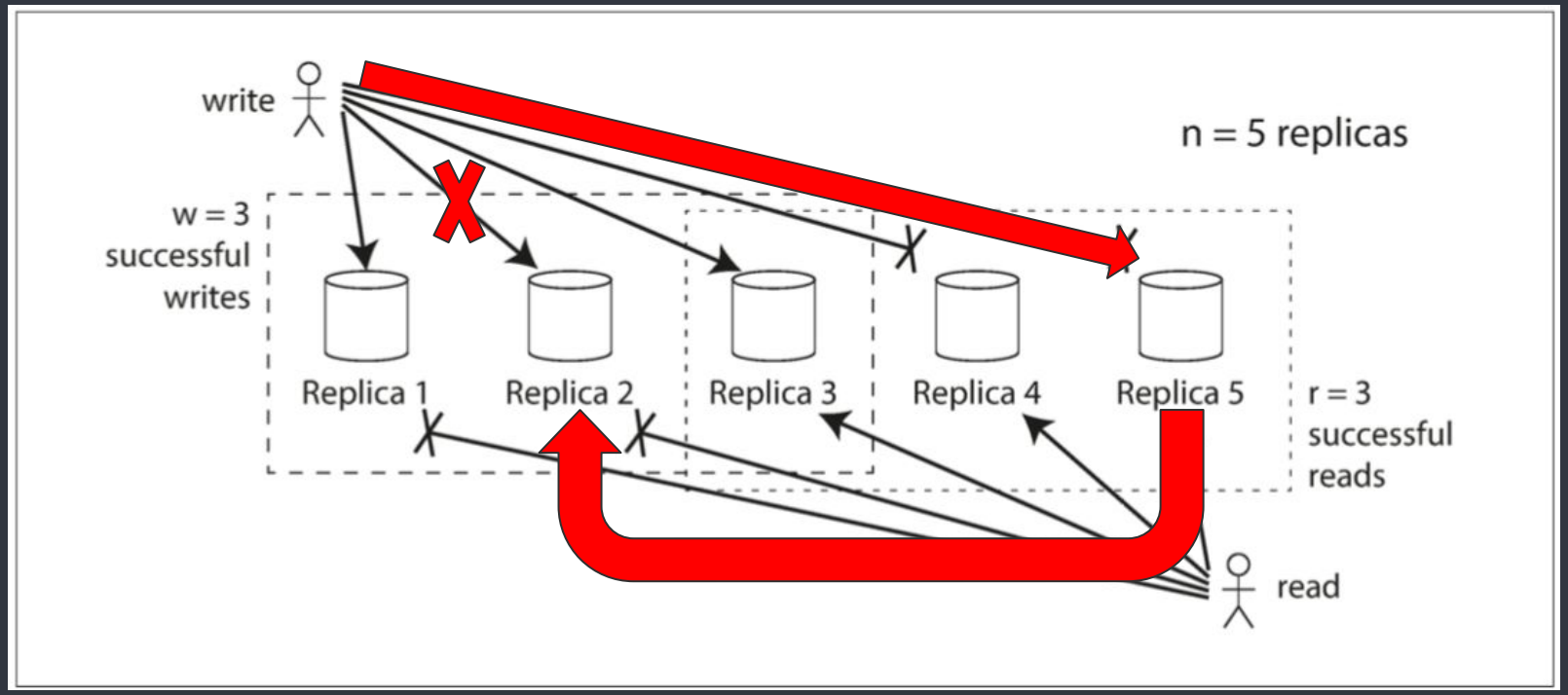
Fault tolerance: Hinted Handoff



Fault tolerance: Hinted Handoff



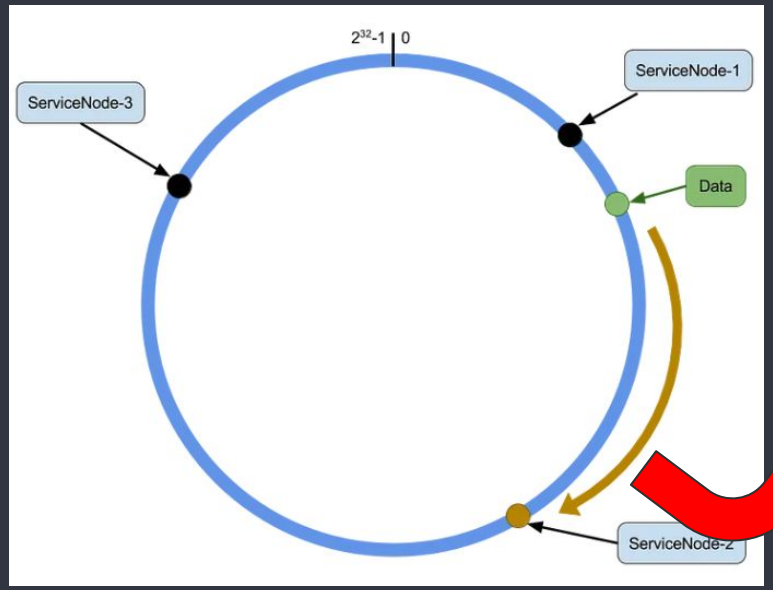
Fault tolerance: Hinted Handoff

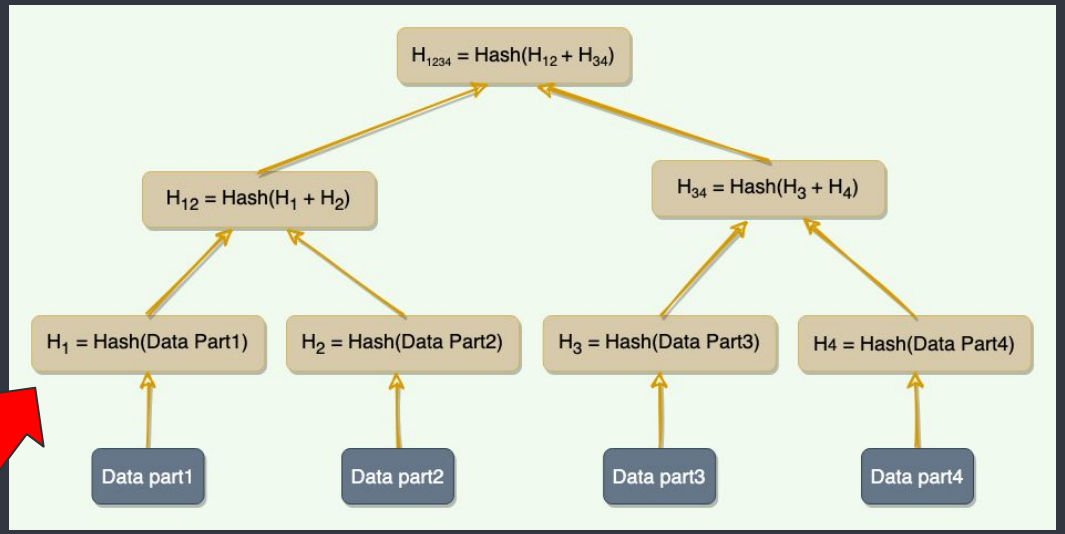


Fault tolerance: Hinted Handoff

- The Temporary Storage Node will have a hint in its metadata that suggests which node was the intended recipient of the replica.
- Nodes that receive hinted replicas will keep them in a separate local database that is scanned periodically.
- Upon detecting that the Temporarily Failed Node has recovered, the Temporary Storage Node will attempt to deliver the replica to A.
- Finally, the Temporary Storage Node may delete the object from its local store

Anti-entropy



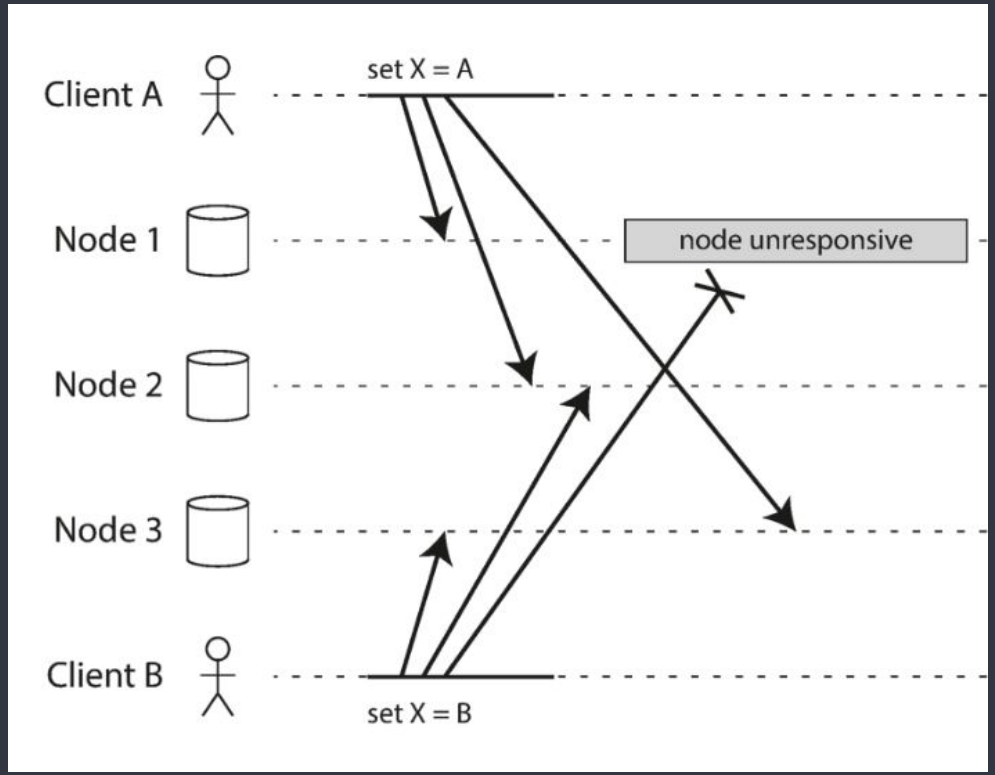


[merkle tree]

Anti-entropy

- Each virtual node maintains a separate Merkle tree for each key range (the set of keys
- Asynchronous replication
- Advantage
  - Each branch of the tree can be checked independently
  - Reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas
- Disadvantage
  - Scheme is that many key ranges change when a node joins or leaves the system thereby requiring the tree(s) to be recalculated

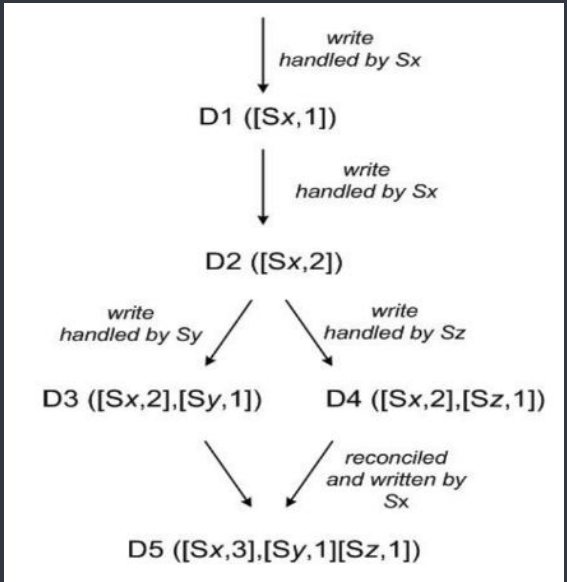
Concurrent write

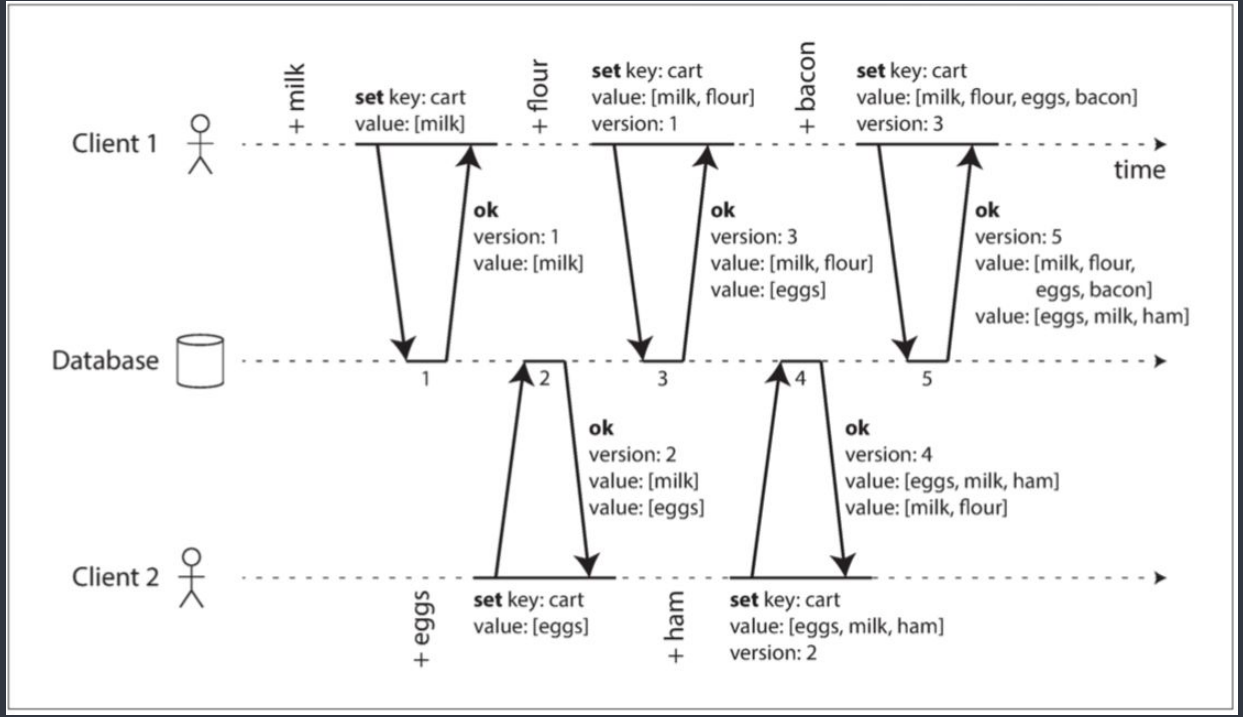


Concurrent write

- LWW (last write win) -> lose data
- In Riak dataset testing https://aphyr.com/posts/285-call-me-maybe-riak
  - *In Riak, last-write-wins resulted in dropping 30-70% of writes, even with the strongest consistency settings (R=W=PR=PW=ALL)*
- In Cassandra https://aphyr.com/posts/294-call-me-maybe-cassandra
  - *28% loss rate*
- Designed to preserve writes as much as possible

Concurrent write



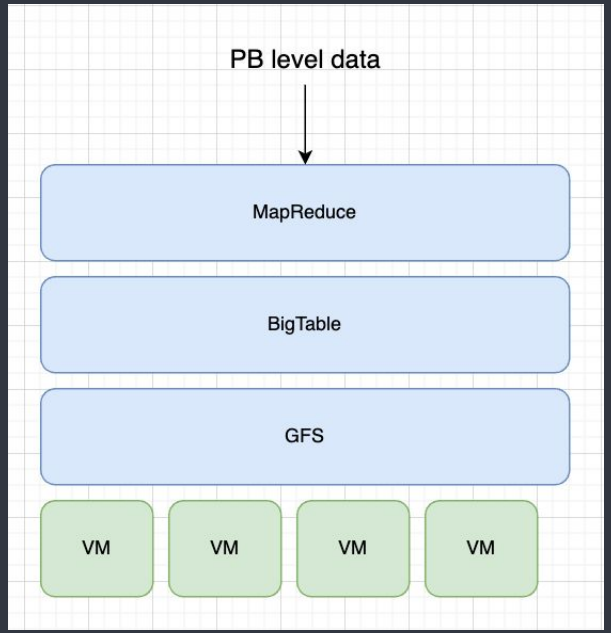


Does not automatically resolve these conflicts.
Returns all conflicting versions to the client application.

Bigtable: A Distributed Storage System for Structured Data & The Google file system (GFS)

Problem

- Managing large-scale structured data across distributed systems with high performance, reliability, and scalability.

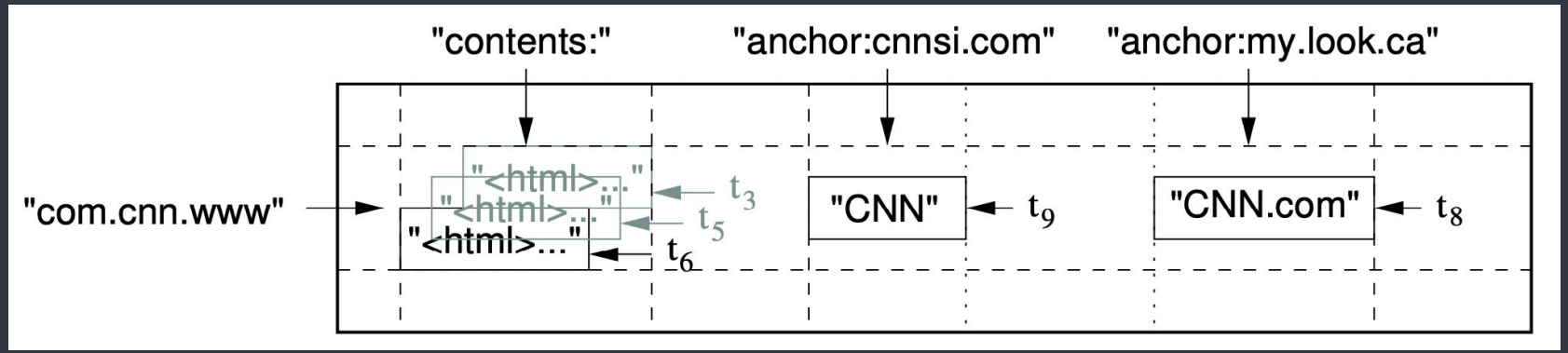


Google Bigtable

- NoSQL
- key-value storage
- Not support like SQL DDL, e.g. Create, ALTER, DROP
  Creating and deleting tables by API
- Supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key
- Does not support general transactions across row keys
  - Reference to Spanner

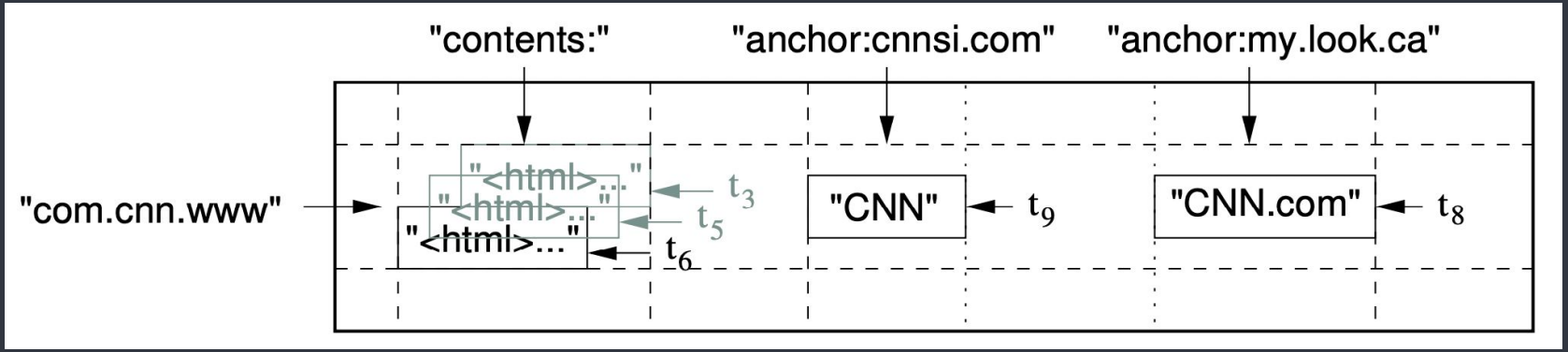
Data Model

- *"A Bigtable is a sparse, distributed, persistent multi- dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes."*
- *(row:string, column:string, time:int64) → string*



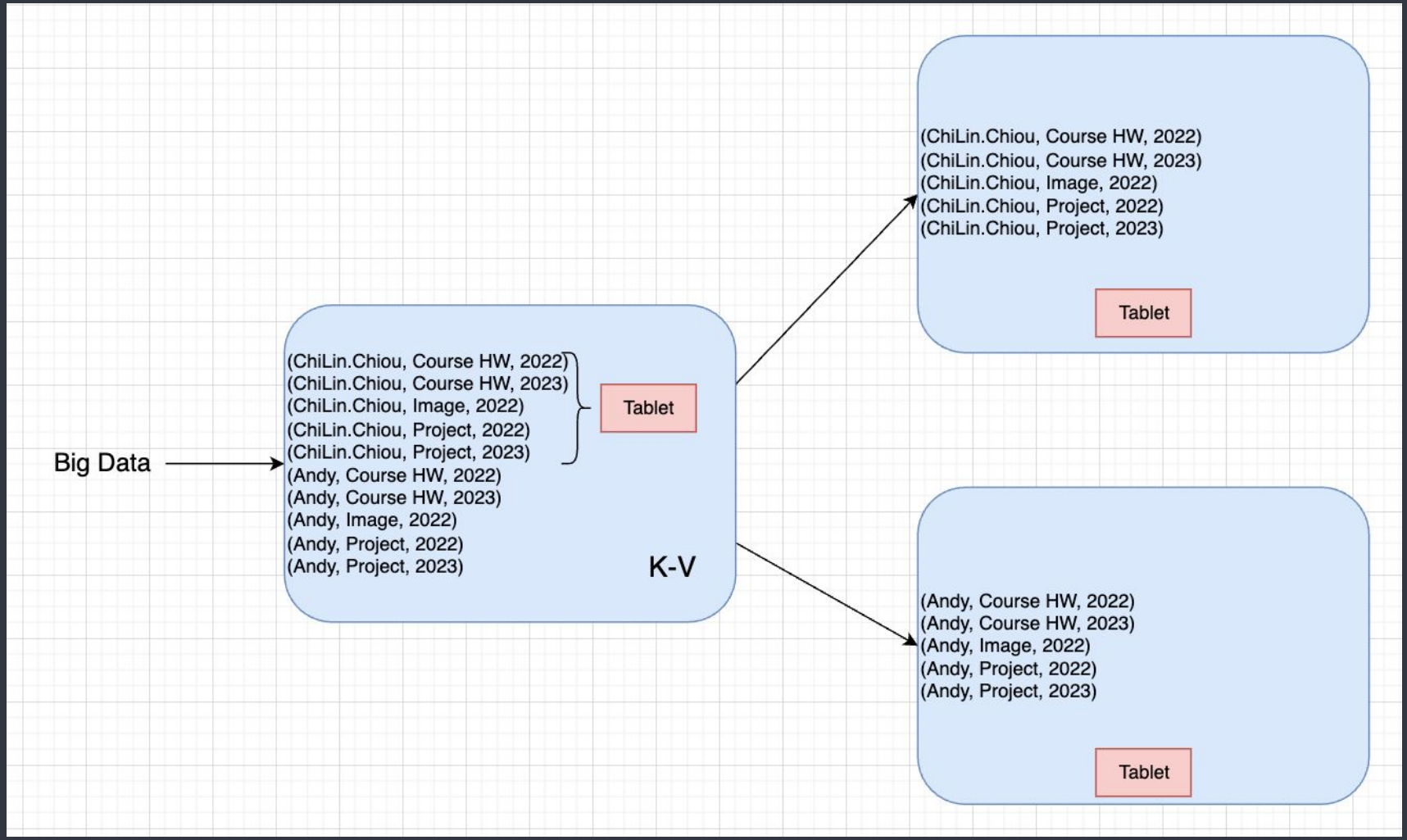
Data Model (cont.)

- (row:string, column:string, time:int64) → string
  - (com.cnn.www, content, t3)
  - (com.cnn.www, content, t5)
  - (com.cnn.www, content, t6)
  - (com.cnn.www, anchor:cnnsi.com, t9)
  - (com.cnn.www, anchor:my.look.ca, t8)



Data Model (cont.)

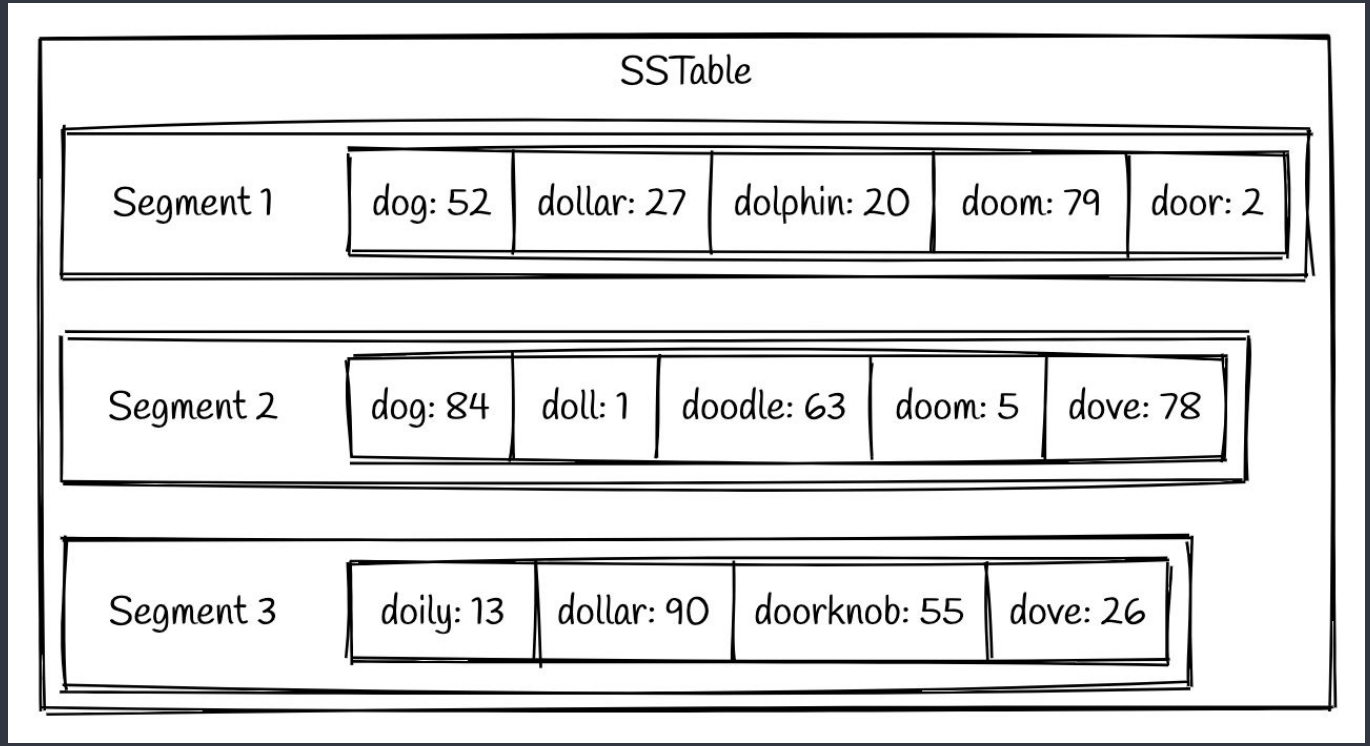
- (row:string, column:string, time:int64) → string
  - (ChiLin.Chiou, Course HW, 2022) -> ...
  - (ChiLin.Chiou, Course HW, 2023) -> ...
  - (ChiLin.Chiou, Image, 2022) -> ...
  - (ChiLin.Chiou, Image, 2023) -> ...
  - (ChiLin.Chiou, Project, 2022) -> ...
  - (ChiLin.Chiou, Project, 2023) -> ...
- Locality


(ChiLin.Chiou, Course HW, 2022)
(ChiLin.Chiou, Course HW, 2023)
(ChiLin.Chiou, Image, 2022)
(ChiLin.Chiou, Project, 2022)
(ChiLin.Chiou, Project, 2023)
Tablet
Big Data
(ChiLin.Chiou, Course HW, 2022)
(ChiLin.Chiou, Course HW, 2023)
(ChiLin.Chiou, Image, 2022)
(ChiLin.Chiou, Project, 2022)
(ChiLin.Chiou, Project, 2023)
Tablet
(Andy, Course HW, 2022)
(Andy, Course HW, 2023)
(Andy, Image, 2022)
(Andy, Project, 2022)
(Andy, Project, 2023)
K-V
(Andy, Course HW, 2022)
(Andy, Course HW, 2023)
(Andy, Image, 2022)
(Andy, Project, 2022)
(Andy, Project, 2023)
Tablet

Dependency

- SSTable
- LSM tree
- GFS
- Chubby
  - One active master at any time
  - Store the bootstrap location of Bigtable data
  - To discover tablet servers and finalize tablet server deaths
  - To store Bigtable schema information
  - To store access control lists

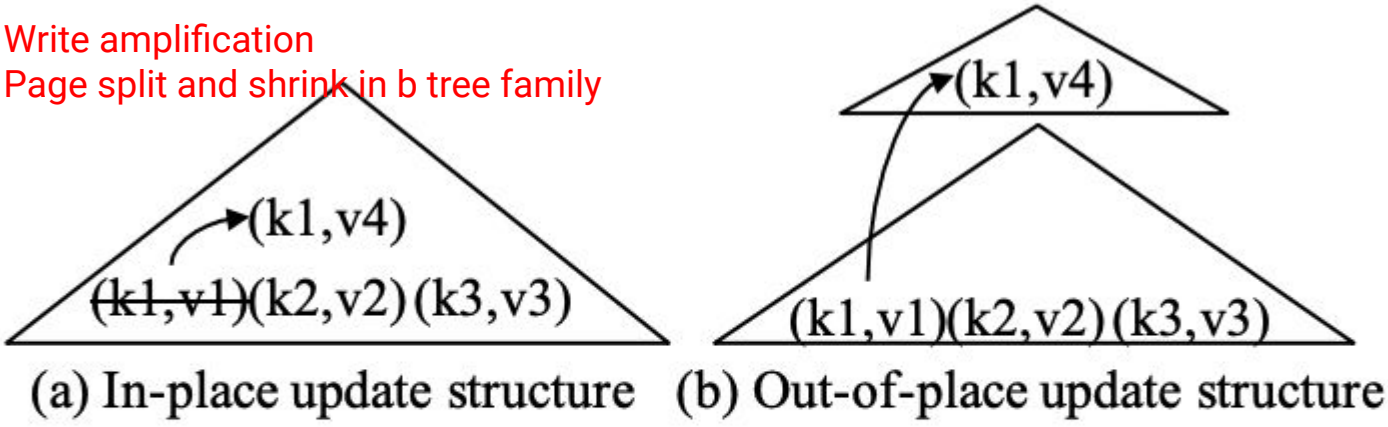
SSTable (Sorted String Table)



reference

LSM Tree (Log Structured-Merge Tree)

Write amplification
Page split and shrink in b tree family

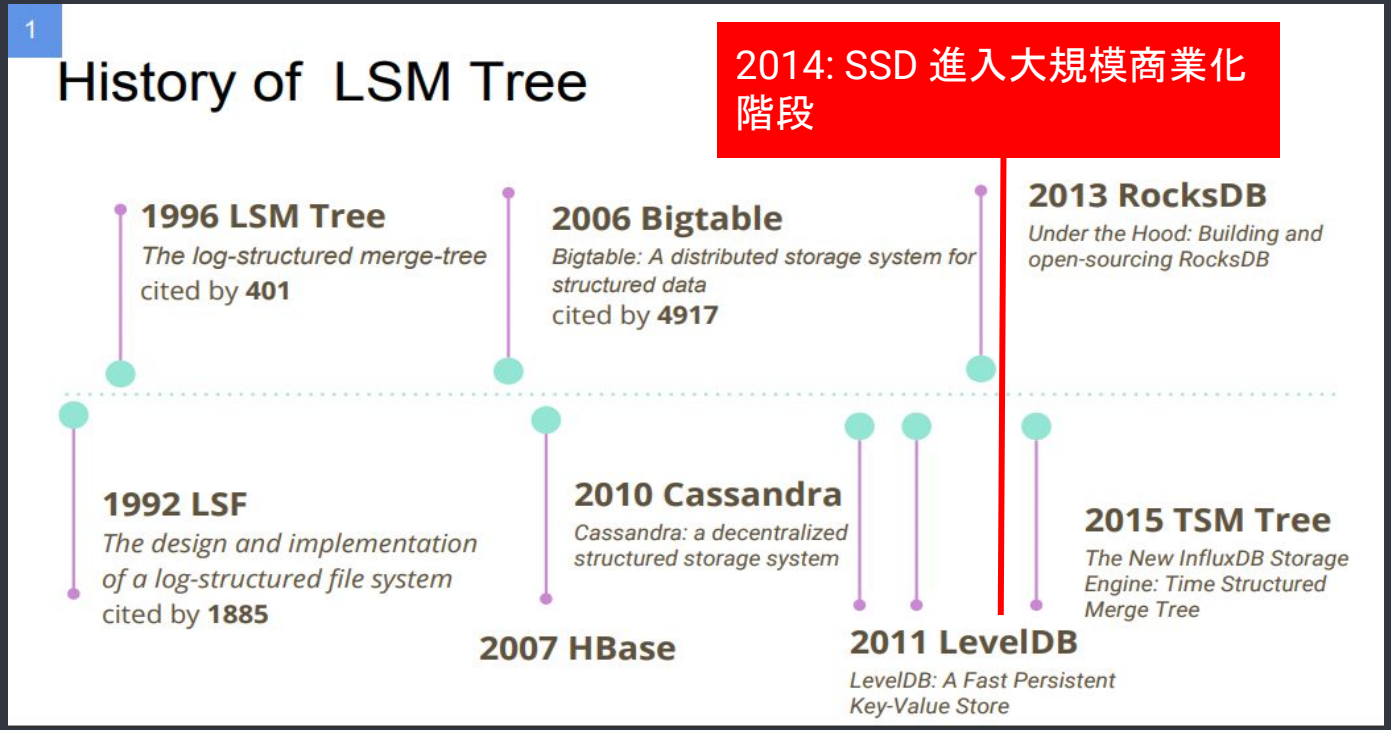


(a) In-place update structure (b) Out-of-place update structure

Fig. 1: Examples of in-place and out-of-place update structures: each entry contains a key (denoted as "k") and a value (denoted as "v")

LSM-based Storage Techniques: A Survey

LSM Tree (Log Structured-Merge Tree)



Pinglei Guo: Log Structured Merge Tree

LSM Tree (cont.)

- Sequential Write
- Append only operation.
- SSD random read >> HDD random read
- SSDs use flash memory, which requires an erase before it can be overwritten. However, erasing can impact the lifespan of the device. This characteristic aligns well with the append-only write operation of an LSM tree.

LSM Tree Architecture

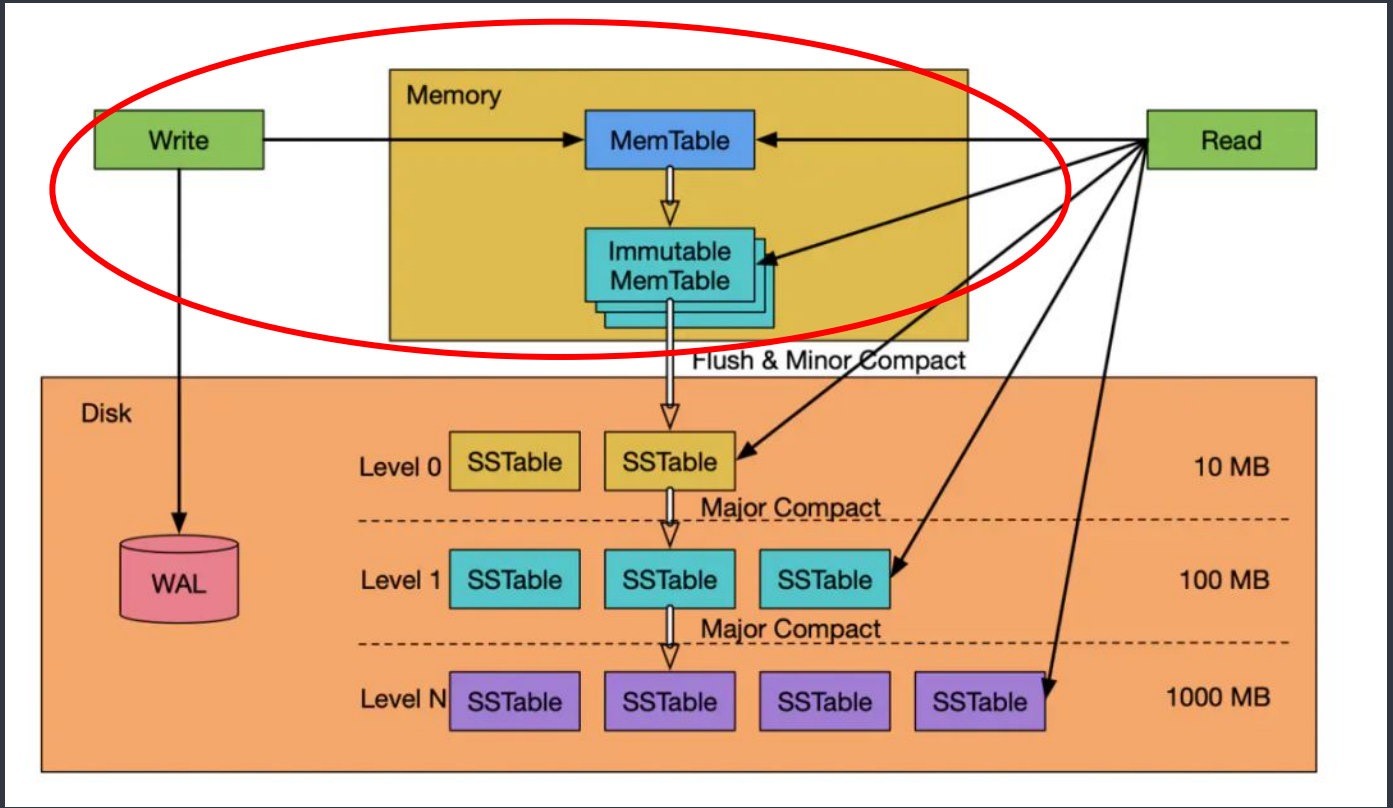


Image reference

LSM Tree Architecture (cont.)

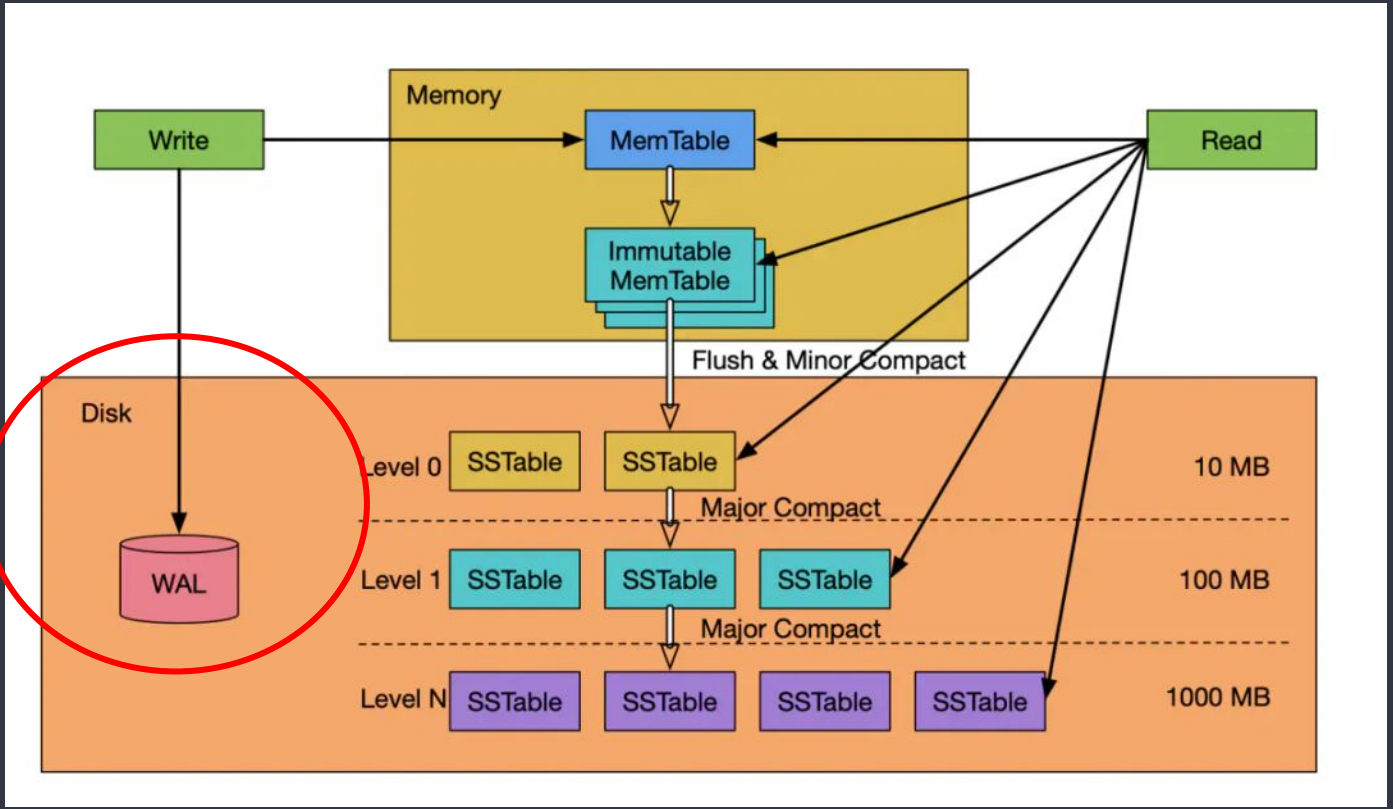


Image reference

LSM Tree Architecture (cont.)

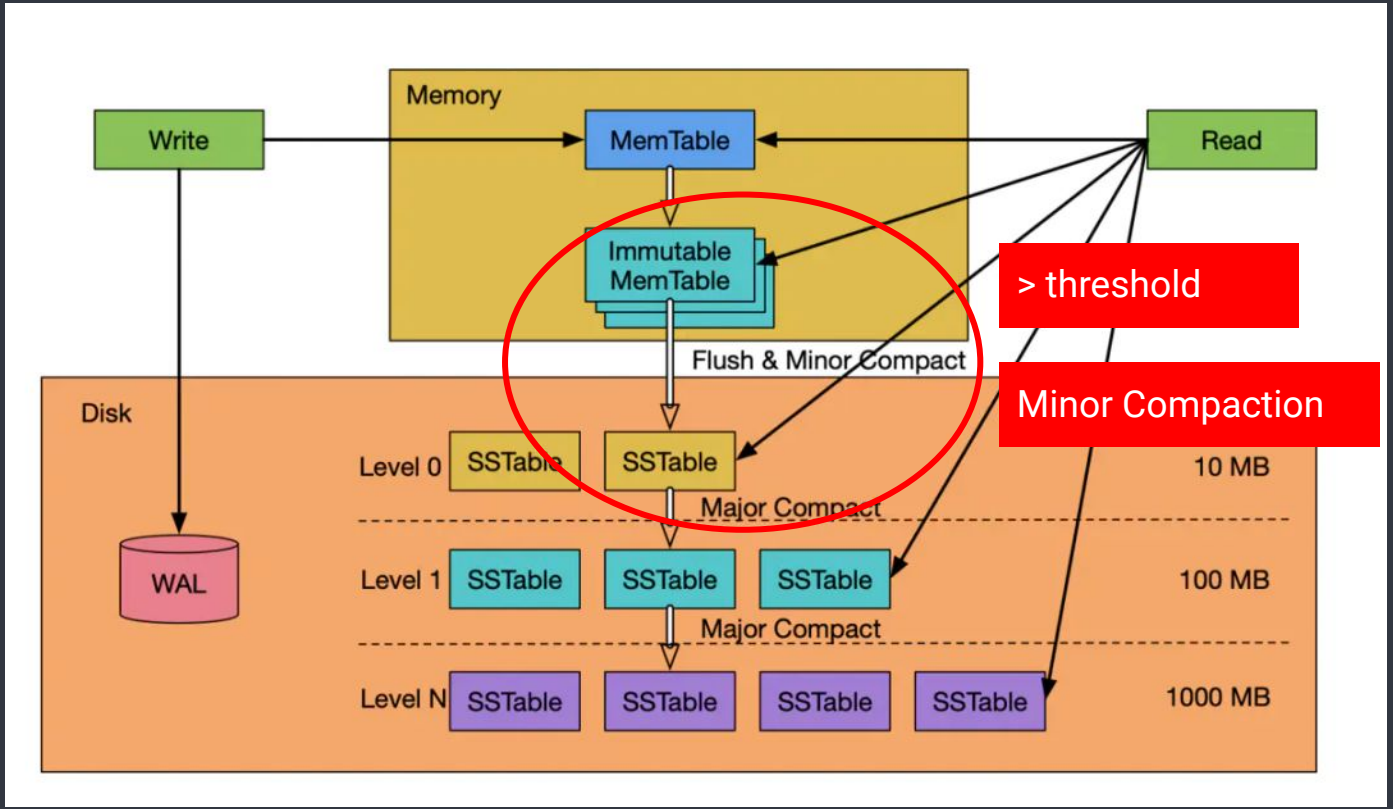


Image reference

LSM Tree Architecture (cont.)

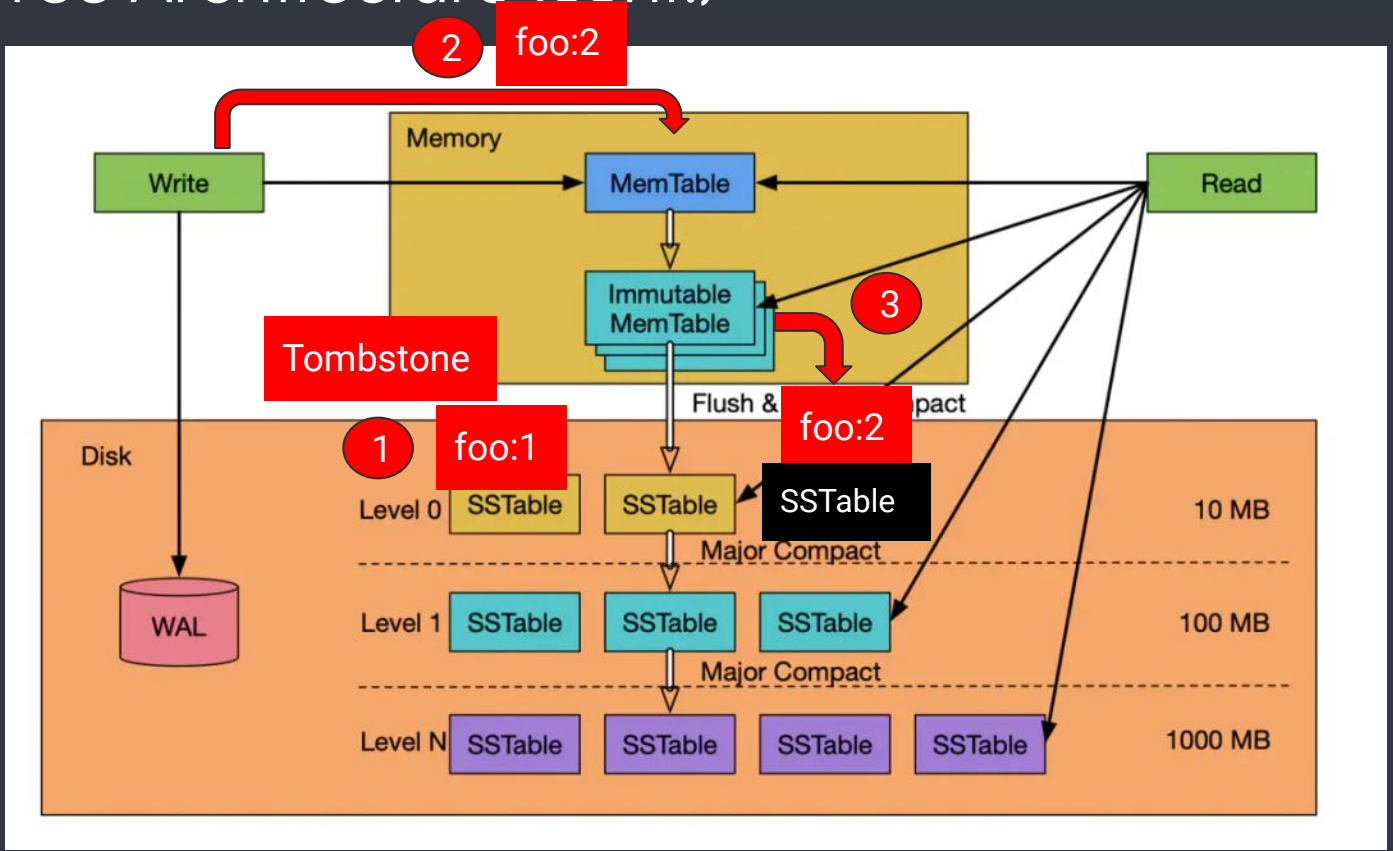


Image reference

LSM Tree Architecture (cont.)

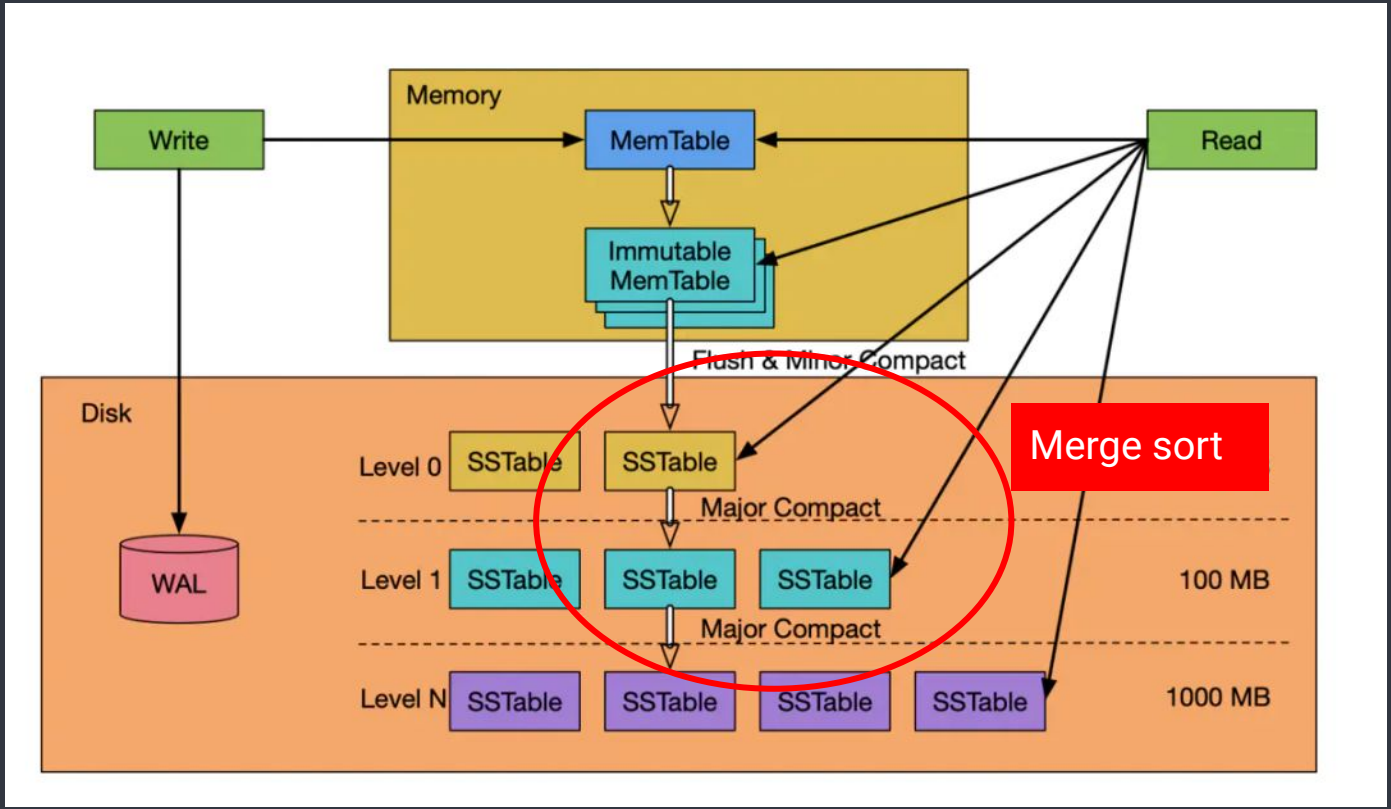


Image reference

LSM Tree Architecture (cont.)

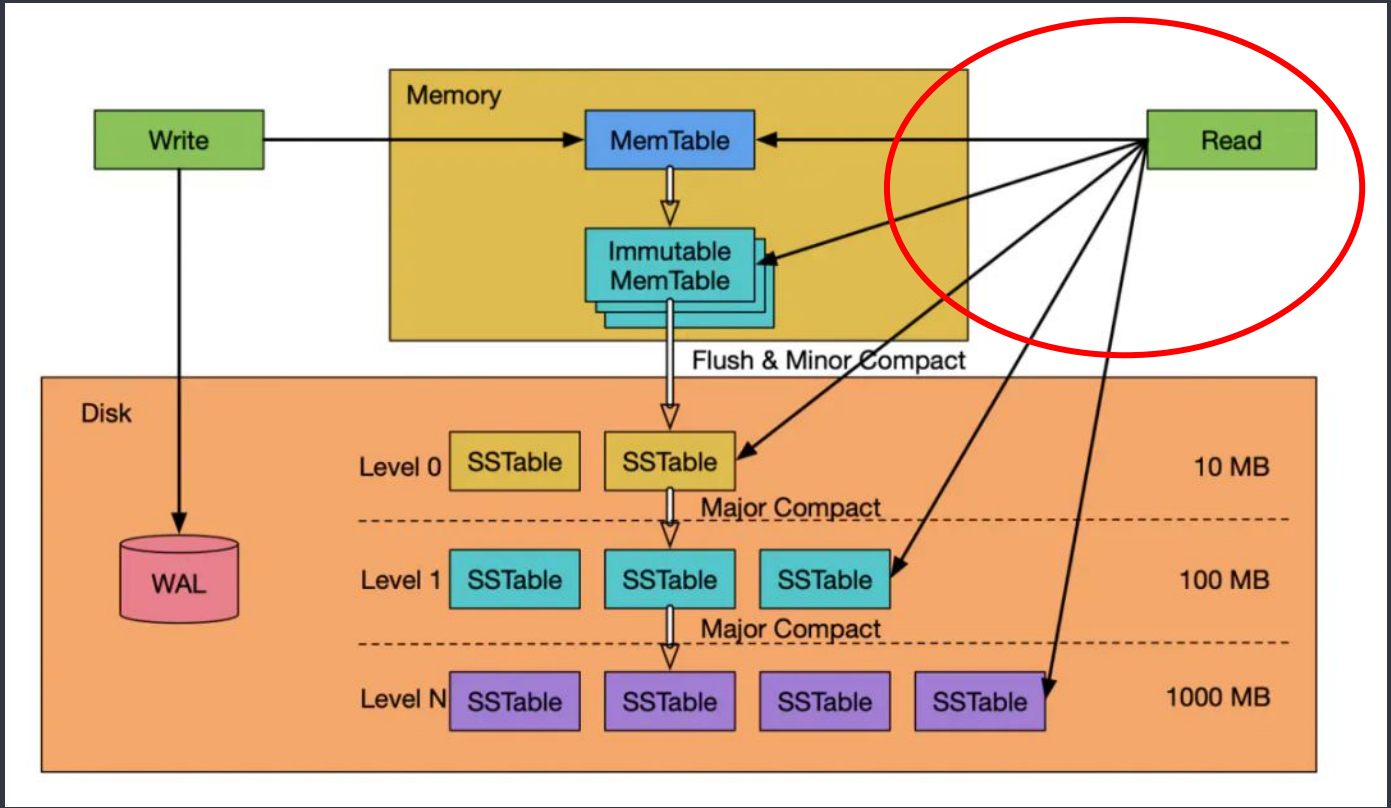
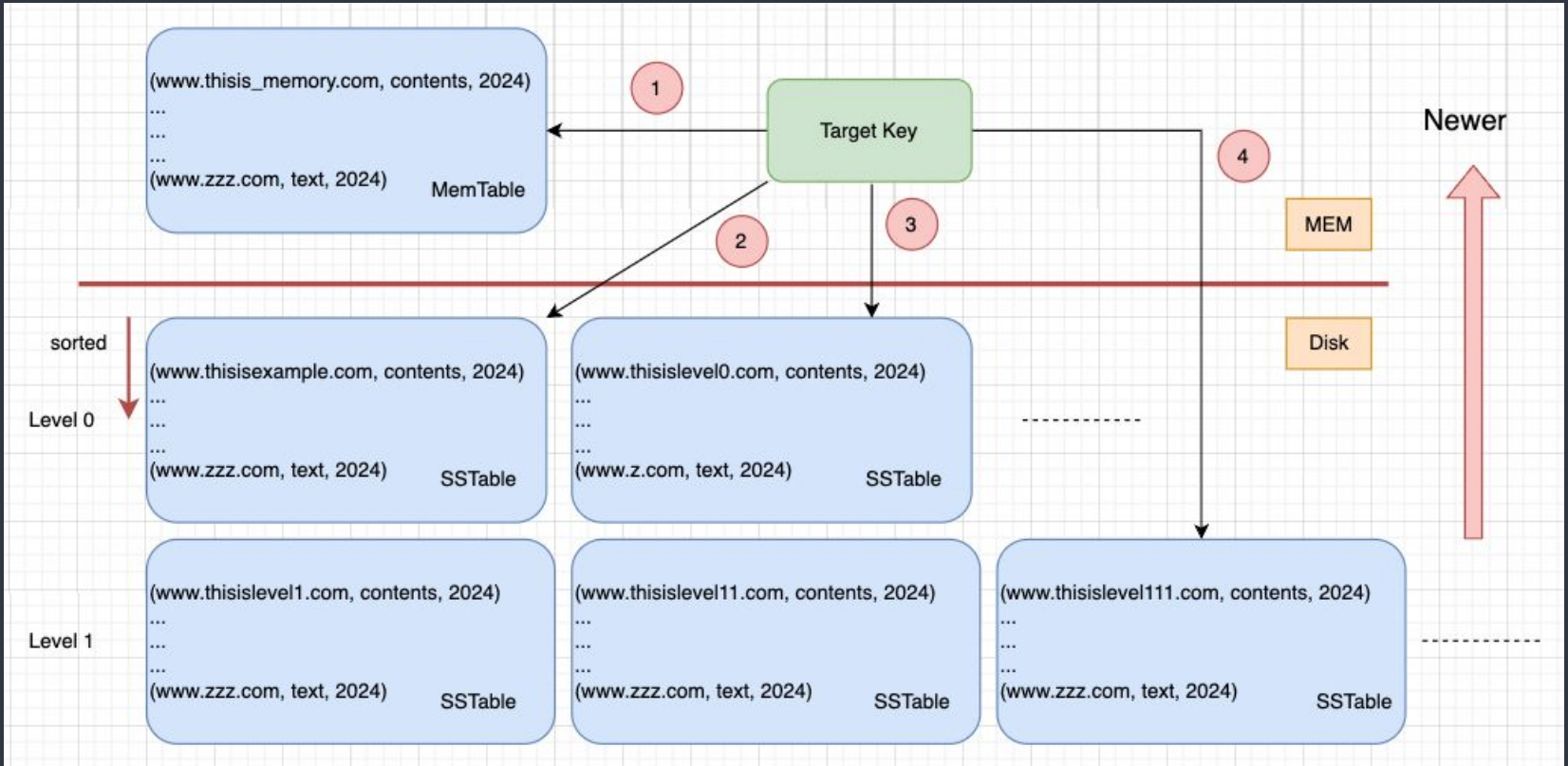


Image reference

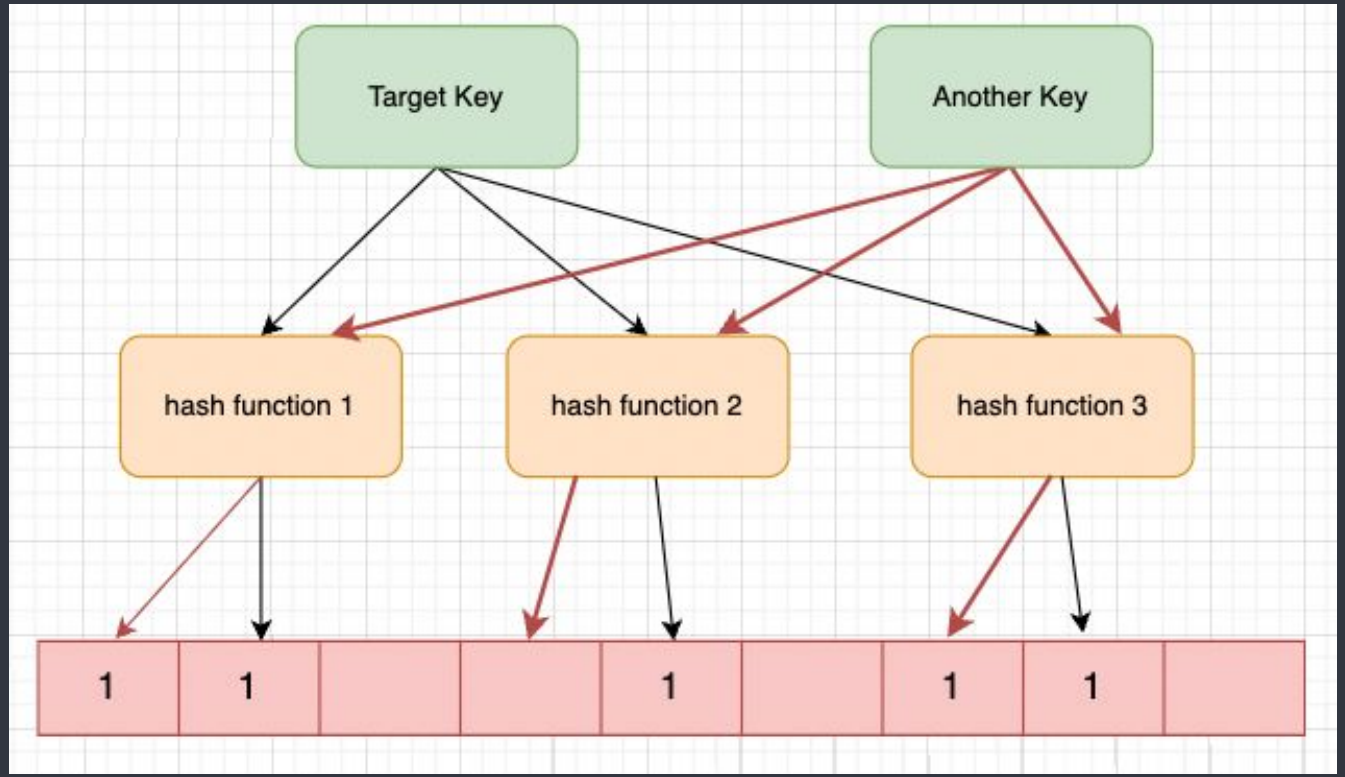
Read operation

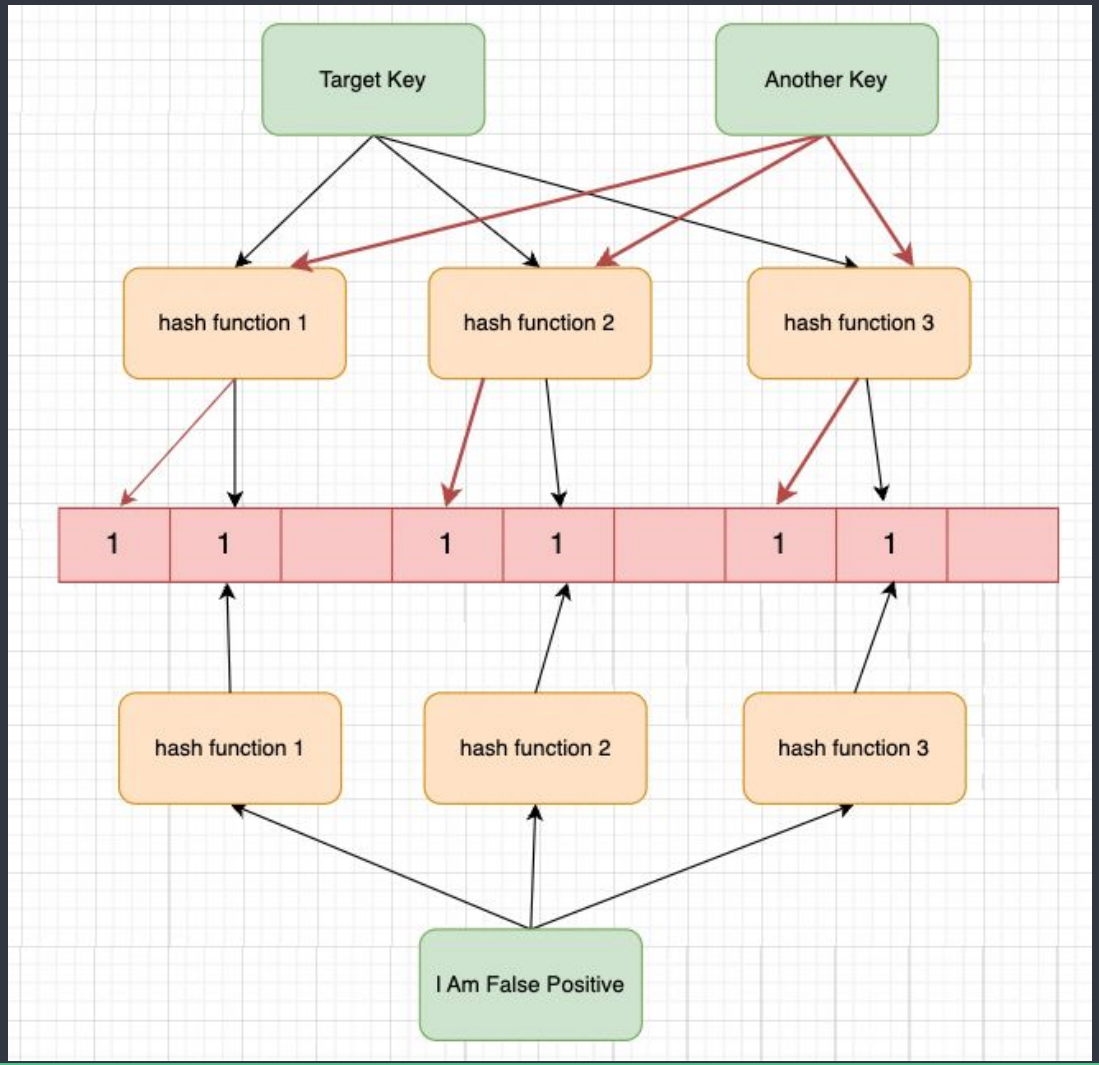
The SSTable internally is sorted, while SSTables among each other are unsorted.
Search from memory to disk, from high level to low level.



Read optimization: bloom filter

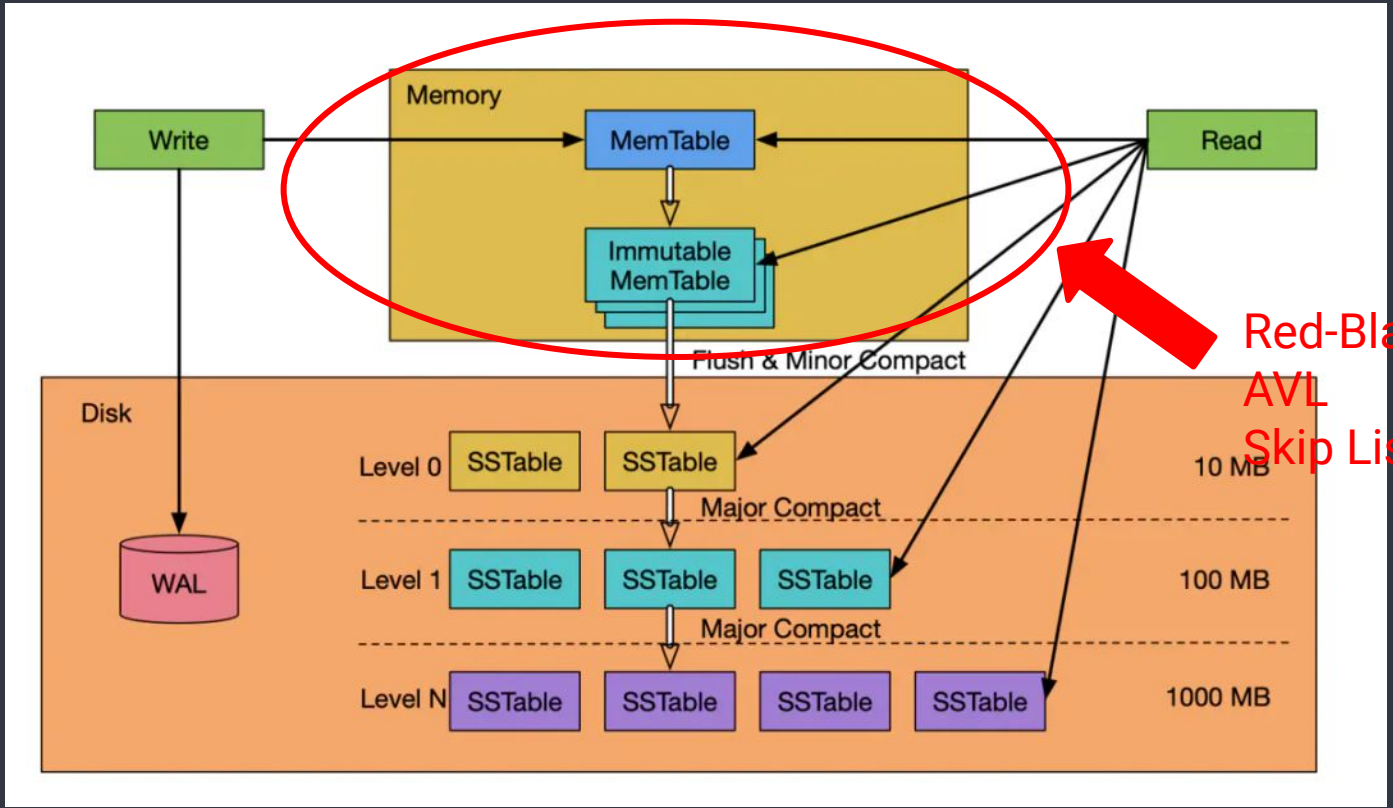
May cause false positive
Performance base on hash function




Target Key
Another Key
hash function 1
hash function 2
hash function 3
1
1
1
1
1
1
hash function 1
hash function 2
hash function 3
I Am False Positive

SSTable (Sorted String Table)

How and when to sort ?!



Red-Black Tree
AVL
Skip List

Image reference

https://github.com/google/leveldb/blob/main/db/skiplist.h

https://github.com/facebook/rocksdb/blob/main/memtable/inlineskiplist.h

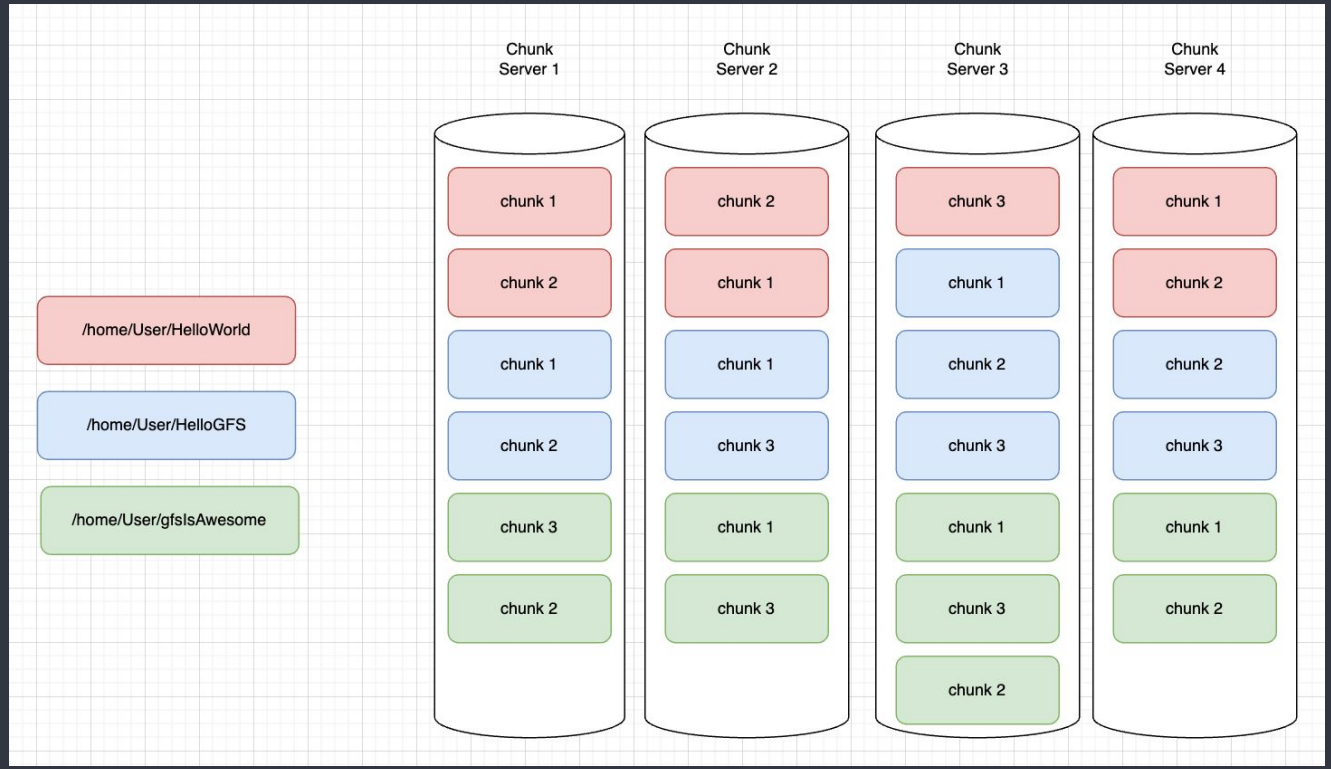
LSM Tree component

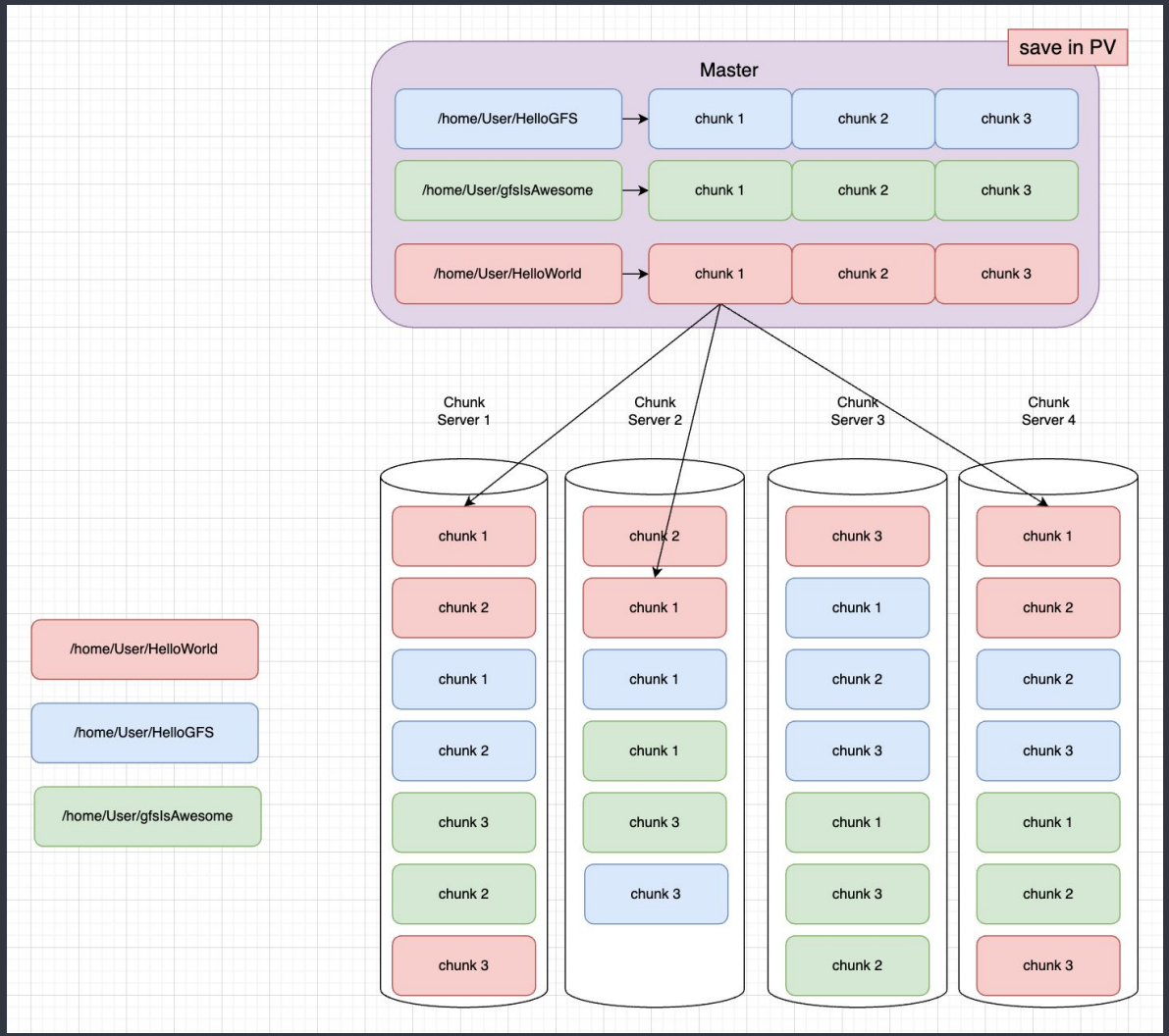
- MemTable
  - Do not defined data structure, e.g. Red-Black tree, Skip List
  - Save in memory
  - WAL (write-ahead logging)
- Immutable MemTable
  - Read-only MemTable.
  - Process copy immutable MemTable to disk (aka. *SSTABLE*), and release memory.
- SSTable
  - First described in BigTable paper.
  - Provides a persistent, ordered(sorted) immutable map from keys to values.

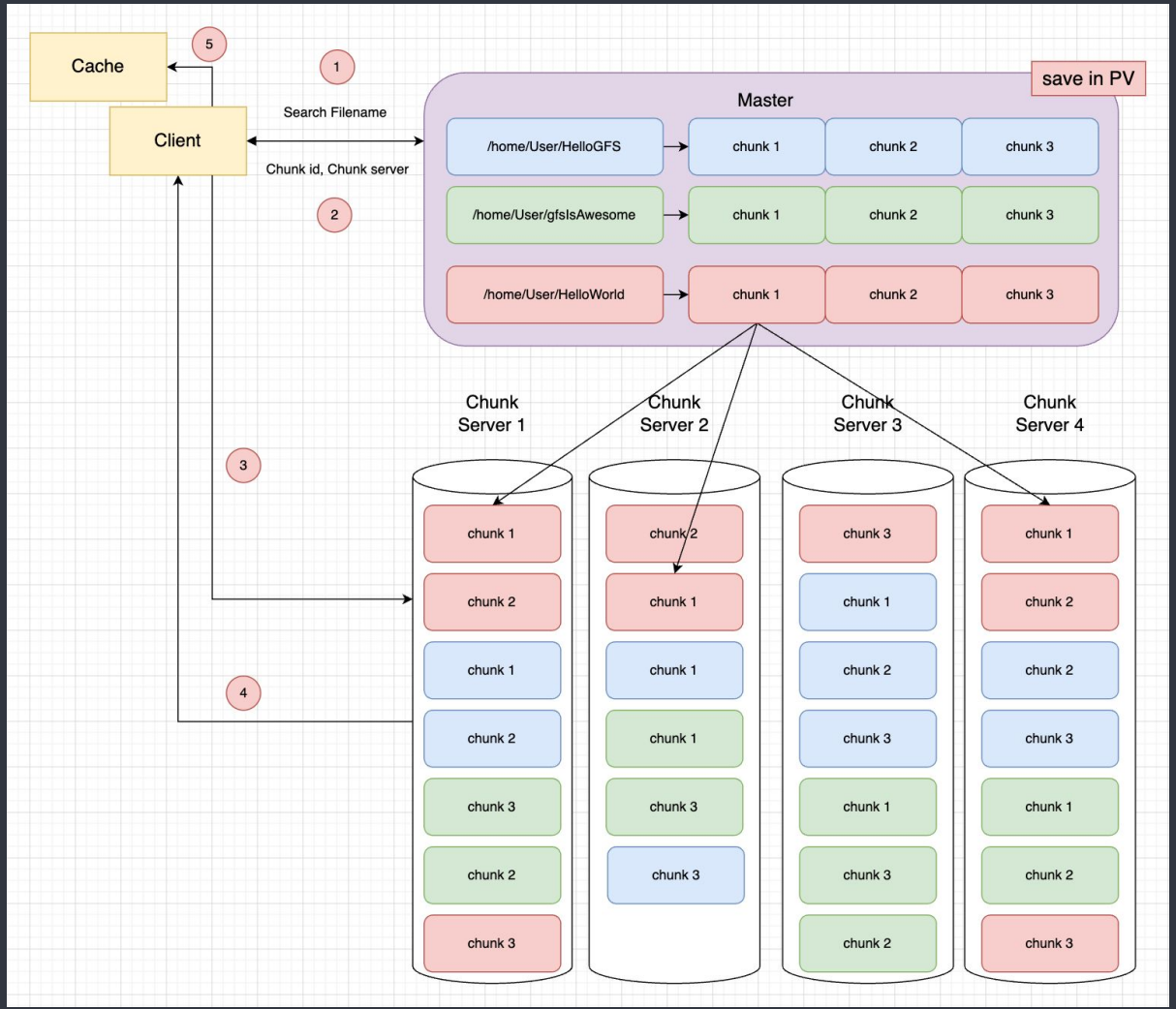
LSM Tree compaction

- Minor compaction
  - Shrinks the memory usage
  - Reduces the amount of data that has to be read from the commit log during recovery if this server dies
- Major compaction
  - Remove space of outdated data. (Tombstone)
  - Improve search efficiency

Google file system (GFS)




save in PV
Master
/home/User/HelloGFS
chunk 1
chunk 2
chunk 3
/home/User/gfsIsAwesome
chunk 1
chunk 2
chunk 3
/home/User/HelloWorld
chunk 1
chunk 2
chunk 3
Chunk Server 1
Chunk Server 2
Chunk Server 3
Chunk Server 4
chunk 1
chunk 2
chunk 1
chunk 2
chunk 1
chunk 3
chunk 2
chunk 3
chunk 3
chunk 1
chunk 1
chunk 1
chunk 2
chunk 2
chunk 3
chunk 3
chunk 1
chunk 1
chunk 2
chunk 3
chunk 1
chunk 2
chunk 3
chunk 3
chunk 2
chunk 3
/home/User/HelloWorld
/home/User/HelloGFS
/home/User/gfsIsAwesome


Cache
5
1
Search Filename
Client
Chunk id, Chunk server
2
save in PV
Master
/home/User/HelloGFS
chunk 1
chunk 2
chunk 3
/home/User/gfsIsAwesome
chunk 1
chunk 2
chunk 3
/home/User/HelloWorld
chunk 1
chunk 2
chunk 3
Chunk Server 1
Chunk Server 2
Chunk Server 3
Chunk Server 4
3
4
chunk 1
chunk 2
chunk 1
chunk 2
chunk 3
chunk 2
chunk 3
chunk 2
chunk 1
chunk 1
chunk 1
chunk 1
chunk 3
chunk 3
chunk 3
chunk 1
chunk 2
chunk 2
chunk 3
chunk 1
chunk 2
chunk 3

Write control flow

- Master Slave Replication
  - Primary -> Replica A
  - Primary -> Replica B
  - Primary become bottleneck
- Detail
  - 1, 2. Ask primary and replica chunk server from master.
  - 3. Write to nearest replica, Each chunkserver will store the data in an internal LRU buffer
  - 4. Client request to primary start write to disk.
  - 5. Primary request to replica.
  - 6. Replica writing done.
  - 7. Primary response to client if all successfully; otherwise, restart from 3.
- Synchronize Replication

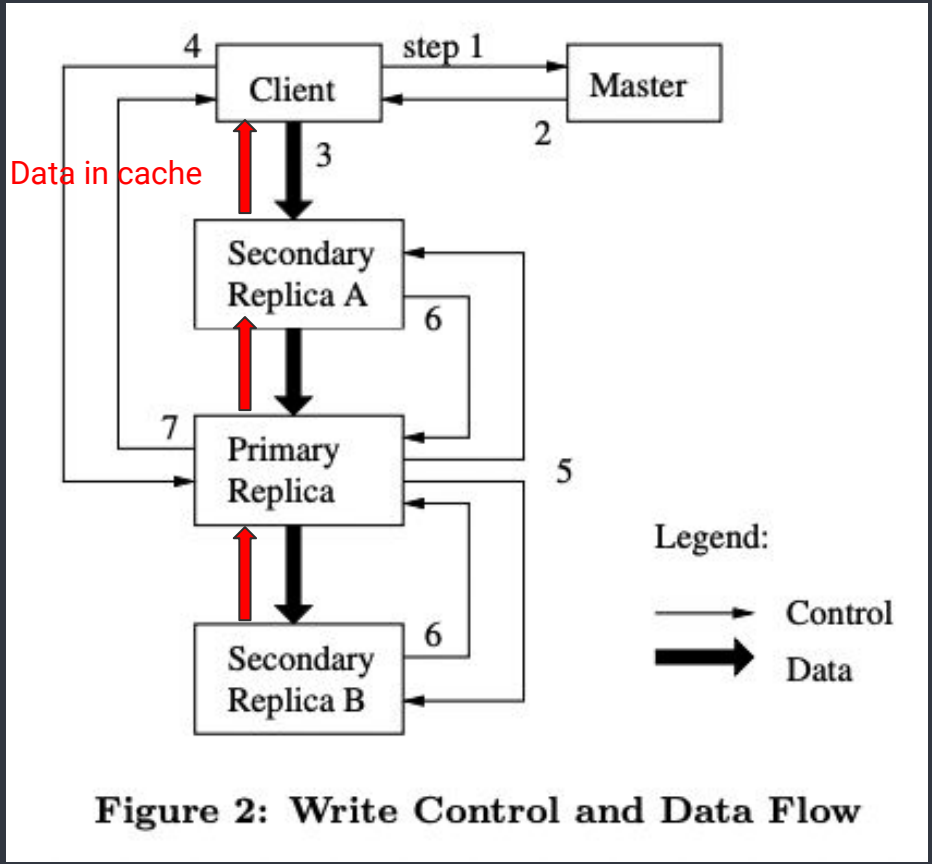
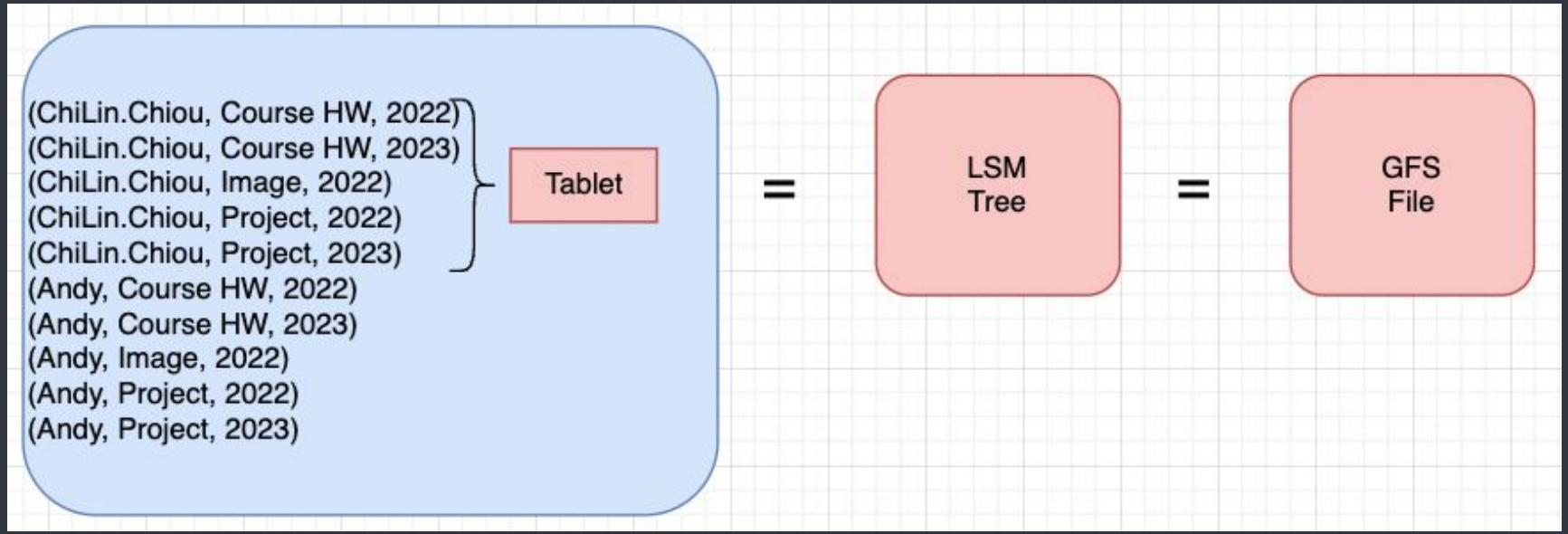
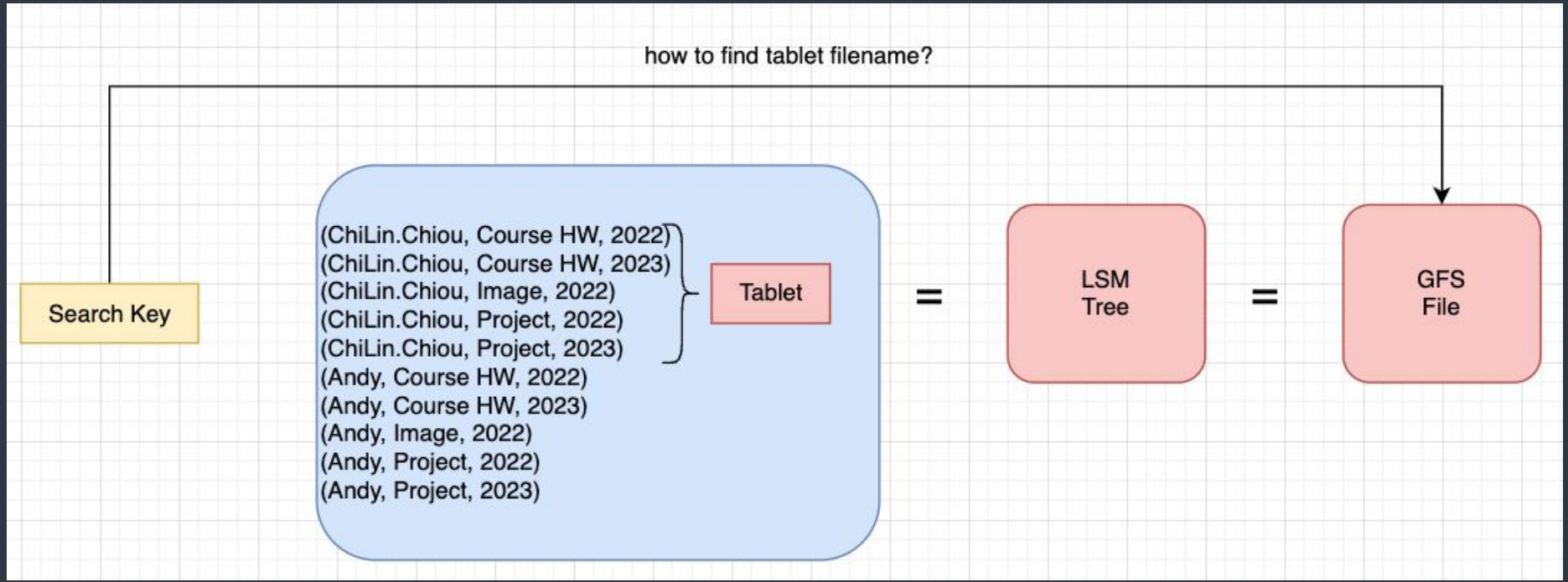


Figure 2: Write Control and Data Flow

GFS + LSM Tree + Tablet



GFS + LSM Tree + Tablet (cont.)



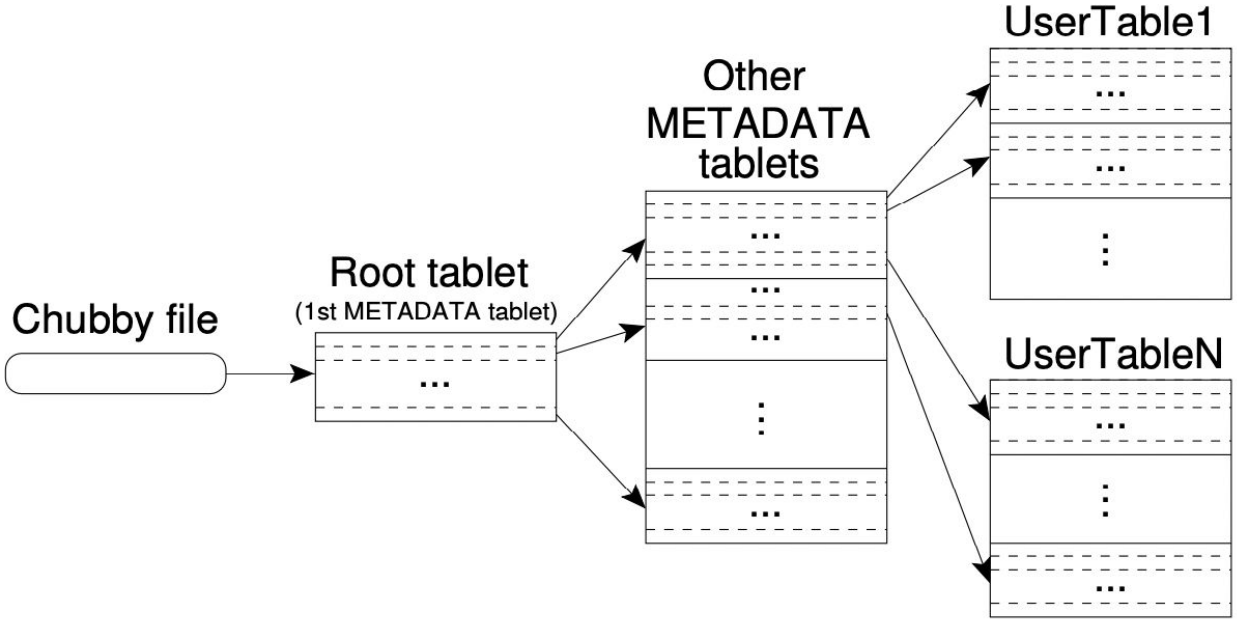
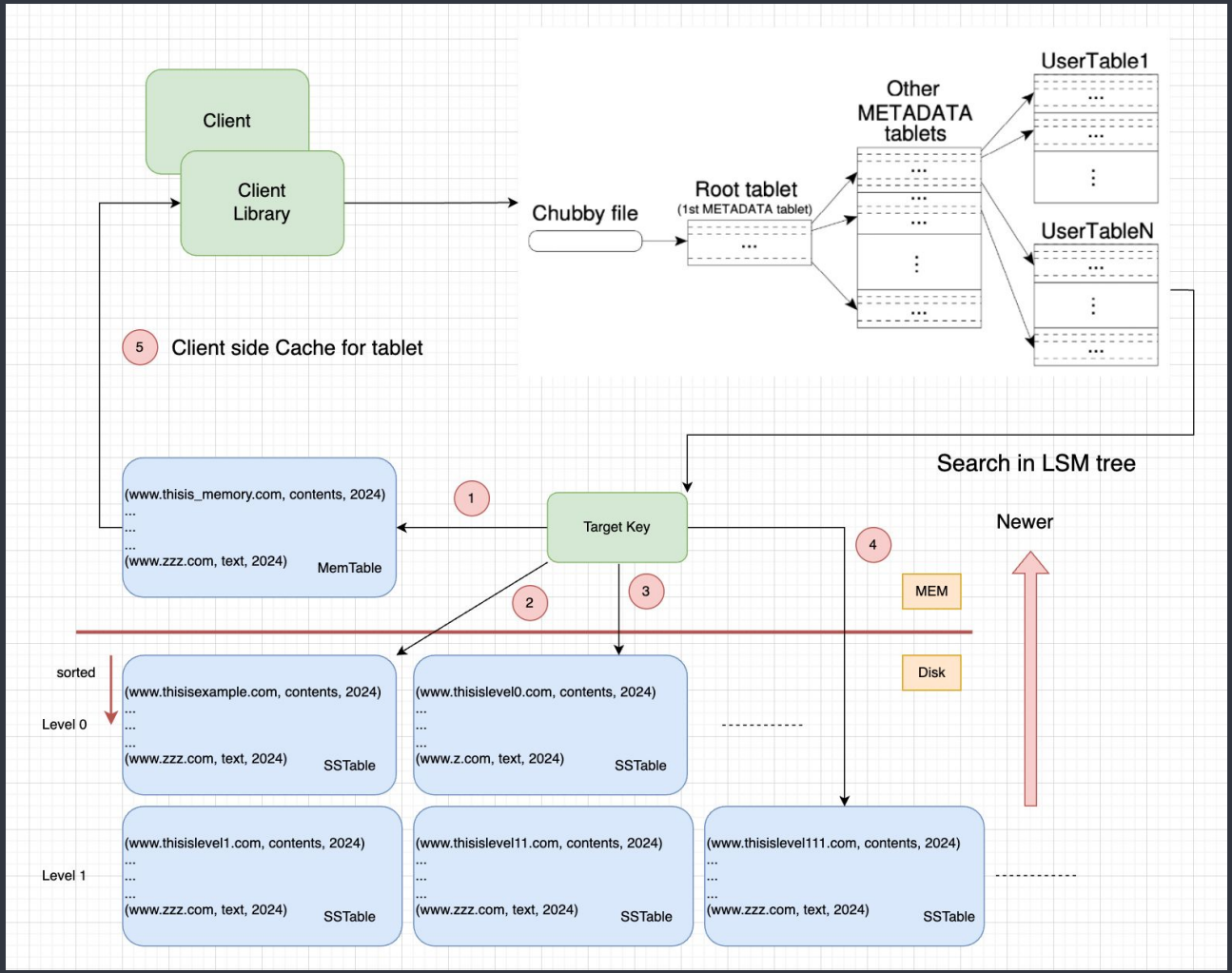


Figure 4: Tablet location hierarchy.

TOTAL: 2048 PB

Three-level hierarchy

- Root tablet
  - Root tablet stored in Chubby.
  - Contains the location of all tablets in a special *METADATA* table.
  - Never split, ensure that the tablet location hierarchy has no more than three levels.
- Metadata
  - Stores the location of a user table.
  - Record tablet is assigned or unassigned to tablet server.
- User table
  - Store multiple user tablet.


Client
Client Library
Chubby file
Root tablet
(1st METADATA tablet)
Other METADATA tablets
UserTable1
UserTableN
5
Client side Cache for tablet
Search in LSM tree
Newer
(www.thisis_memory.com, contents, 2024)
(www.zzz.com, text, 2024)
MemTable
1
Target Key
2
3
4
MEM
Disk
sorted
Level 0
(www.thisisexample.com, contents, 2024)
(www.zzz.com, text, 2024)
SSTable
(www.thisislevel0.com, contents, 2024)
(www.z.com, text, 2024)
SSTable
Level 1
(www.thisislevel1.com, contents, 2024)
(www.zzz.com, text, 2024)
SSTable
(www.thisislevel11.com, contents, 2024)
(www.zzz.com, text, 2024)
SSTable
(www.thisislevel111.com, contents, 2024)
(www.zzz.com, text, 2024)
SSTable

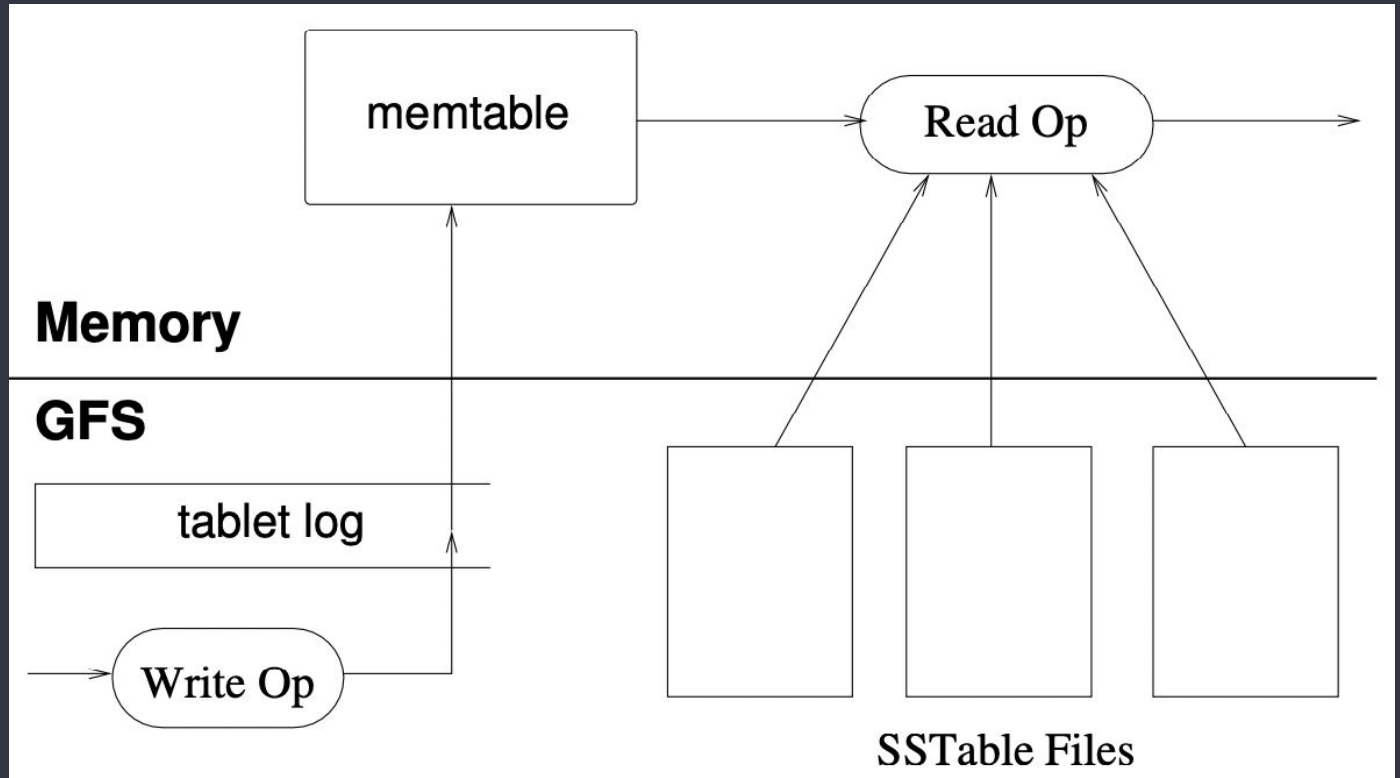
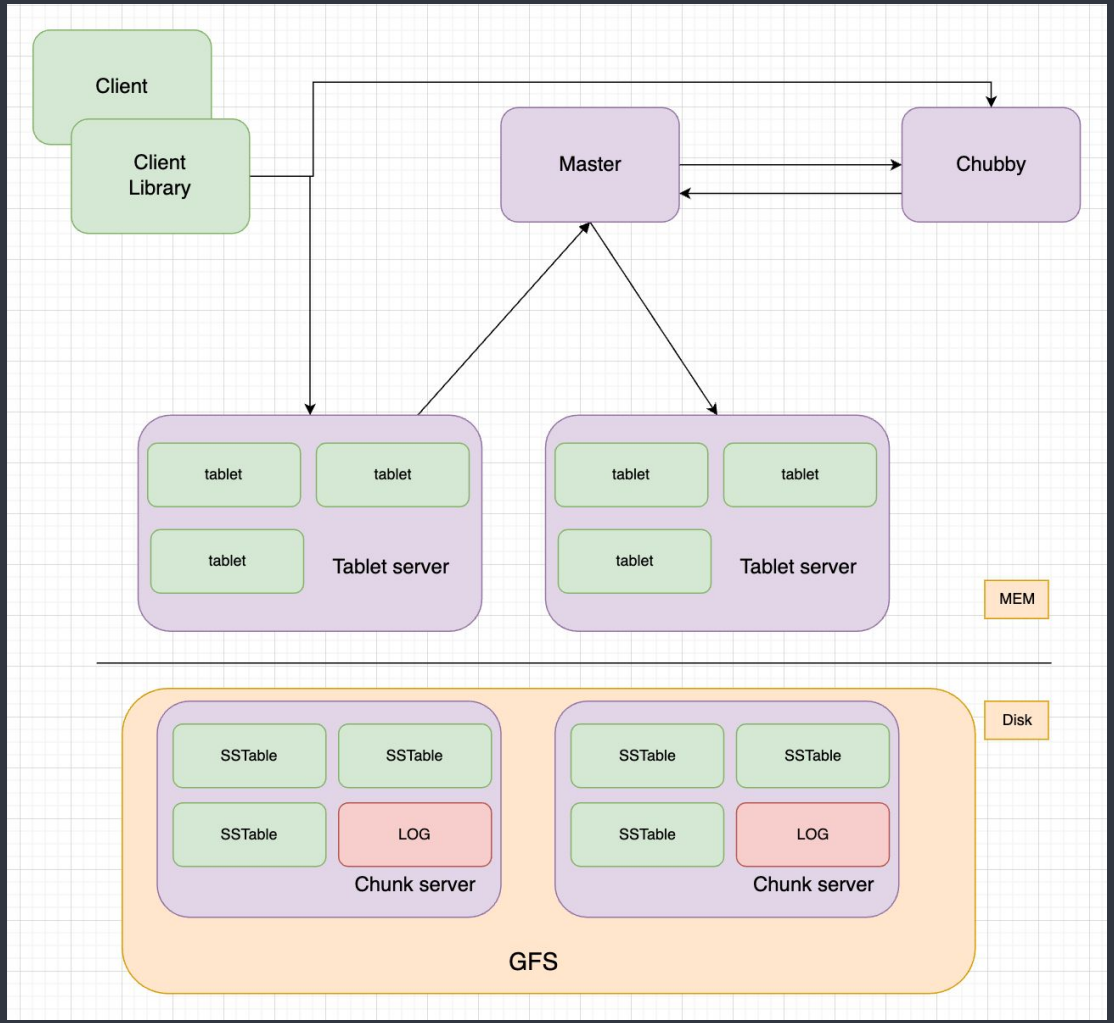
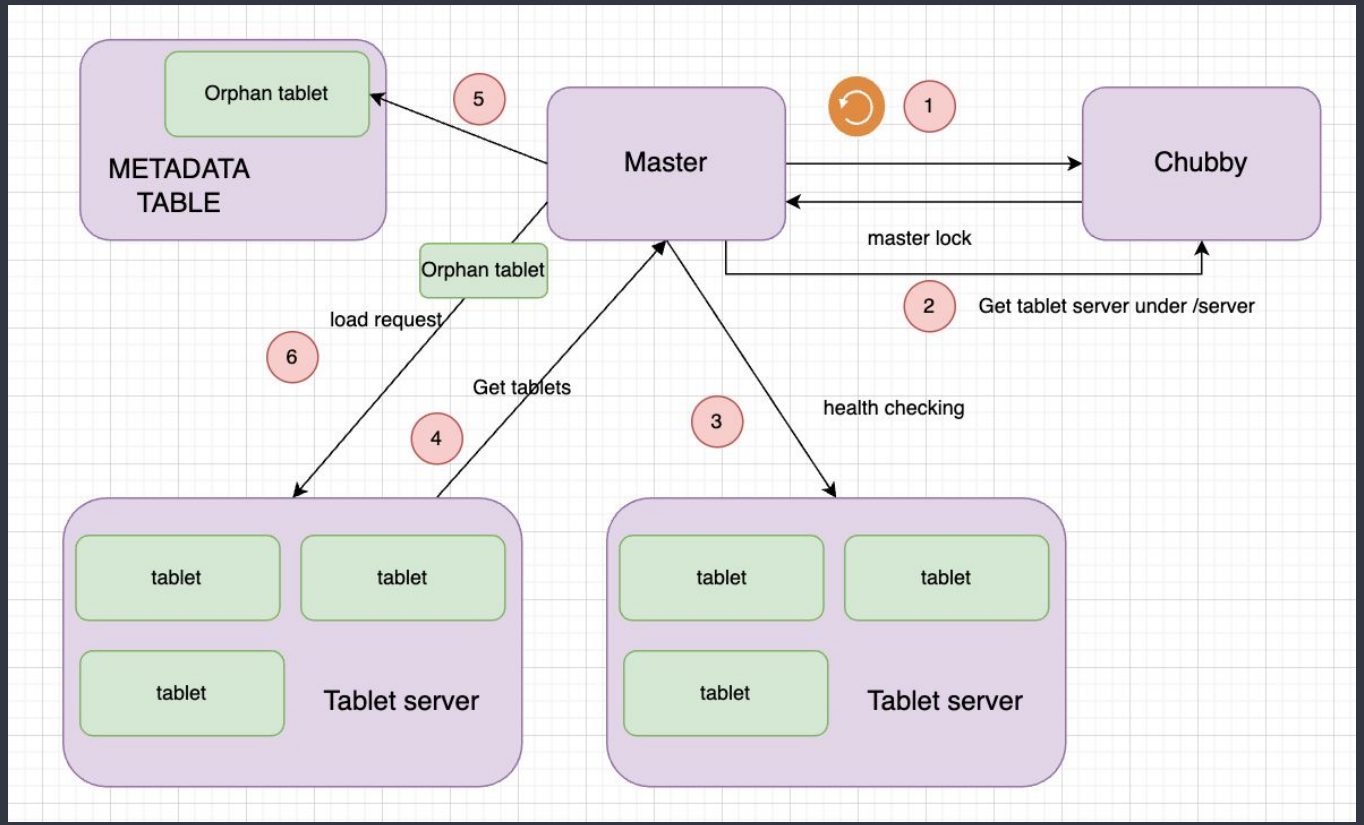


Figure 5: Tablet Representation


Client
Client Library
Master
Chubby
tablet
tablet
tablet
Tablet server
tablet
tablet
tablet
Tablet server
MEM
Disk
SSTable
SSTable
SSTable
LOG
Chunk server
SSTable
SSTable
SSTable
LOG
Chunk server
GFS

Fault Tolerance

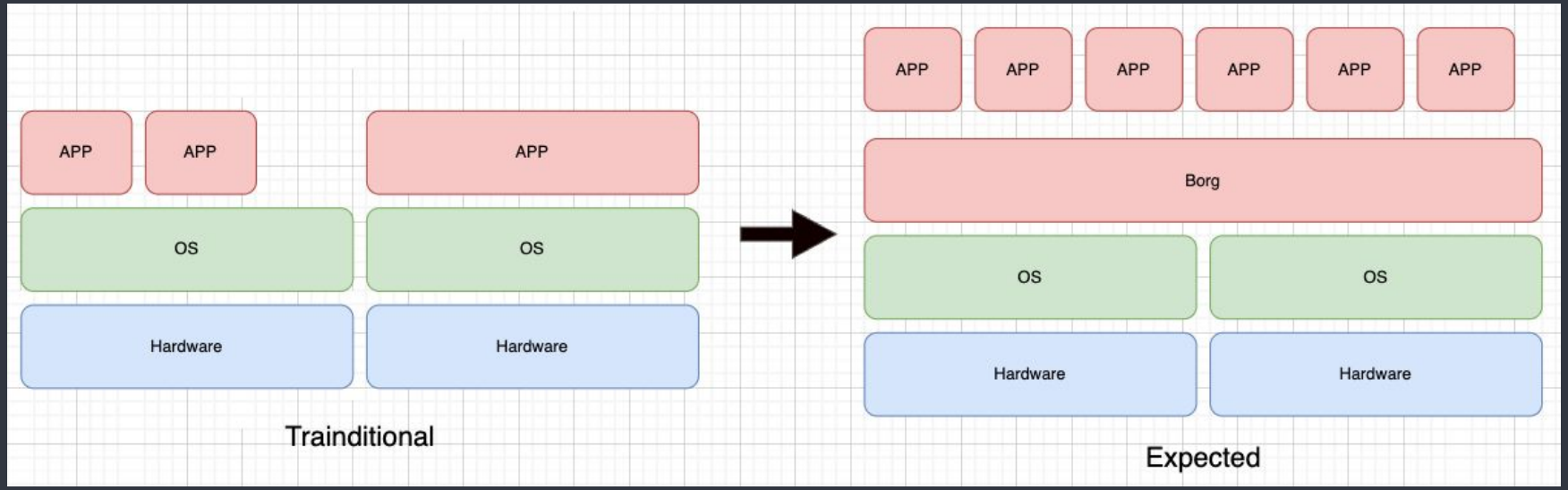


Bigtable

- BigTable = Client * 1 + Master * 1 + Tablet server * N
- Master
  - Assigning tablets to tablet servers
  - Detecting the addition and expiration of tablet servers
  - Balancing tablet-server load
  - Garbage collection of files in GFS
  - Client data does not move through the master
- Tablet server
  - 10 ~ 1000 tablets per tablet server
  - Each tablet is assigned to one tablet server at a time
  - Handles read and write requests to the tablets
  - Splits tablets that have grown too large

Large-scale cluster management at Google with Borg

Problem


APP
APP
APP
OS
OS
Hardware
Hardware
Trainditional
APP
APP
APP
APP
APP
APP
Borg
OS
OS
Hardware
Hardware
Expected

Objectives

1. Run workloads across tens of thousands of machines effectively.
2. Very high utilization of resource.
3. Hides the details of resource management and failure handling.
4. High reliability, availability and scalability.

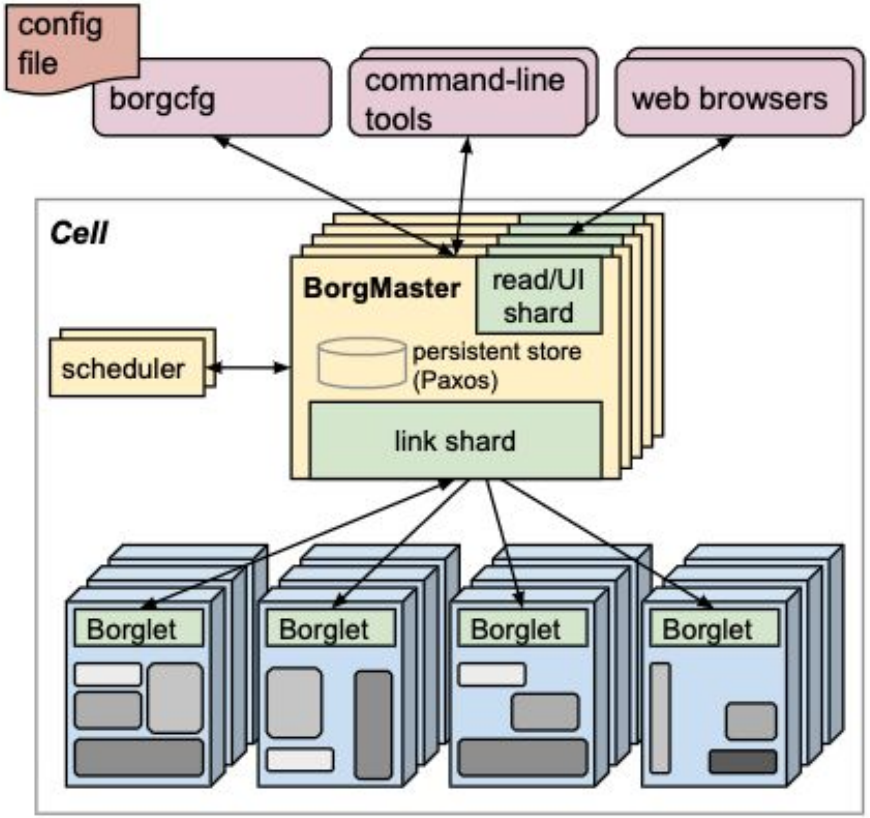
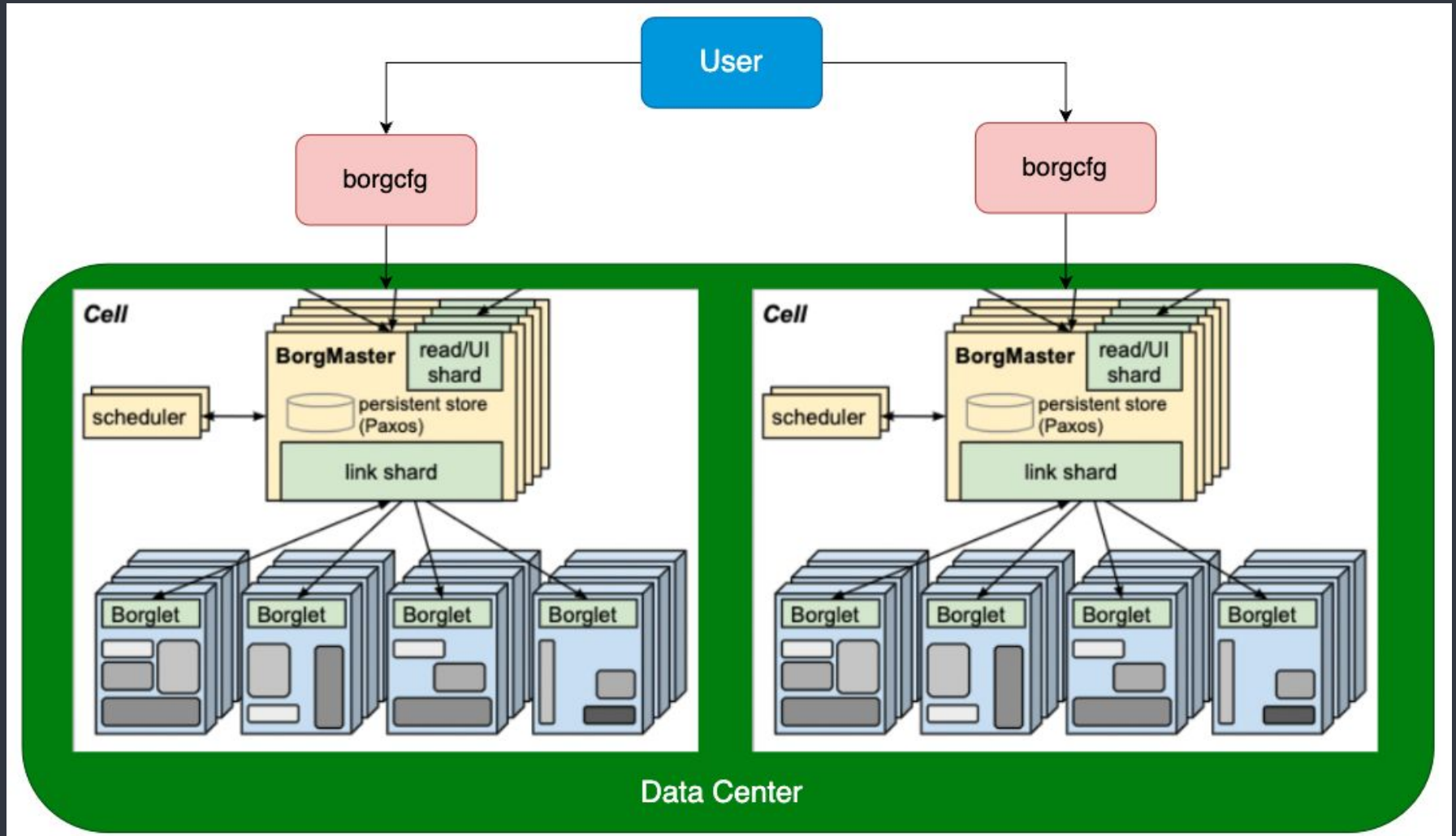
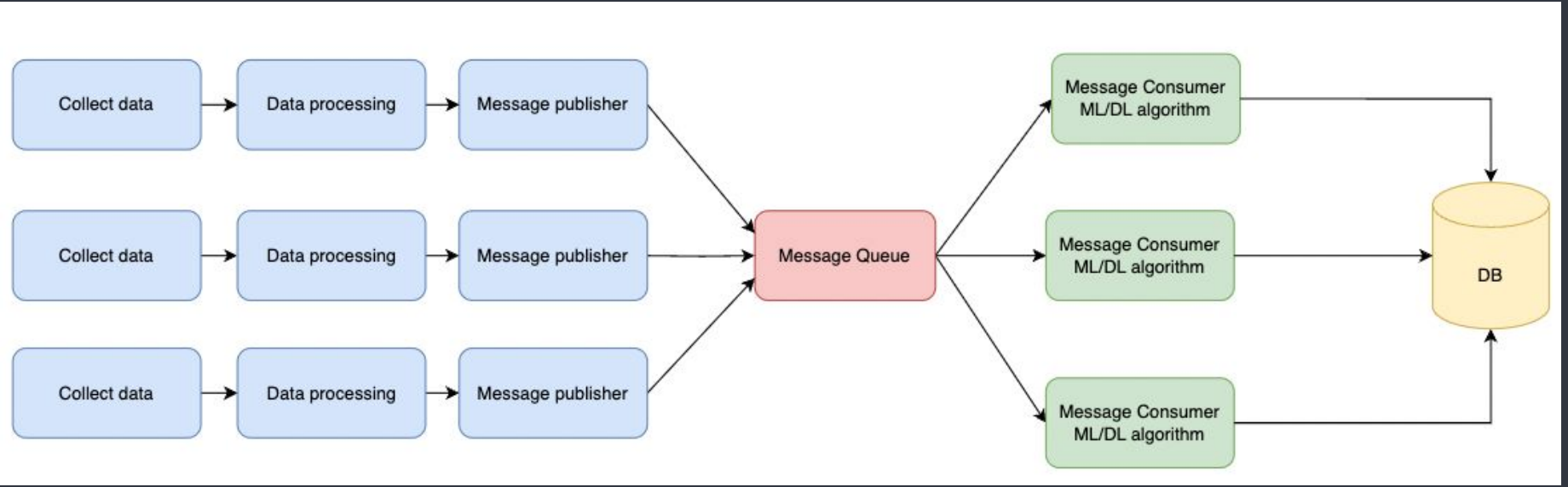
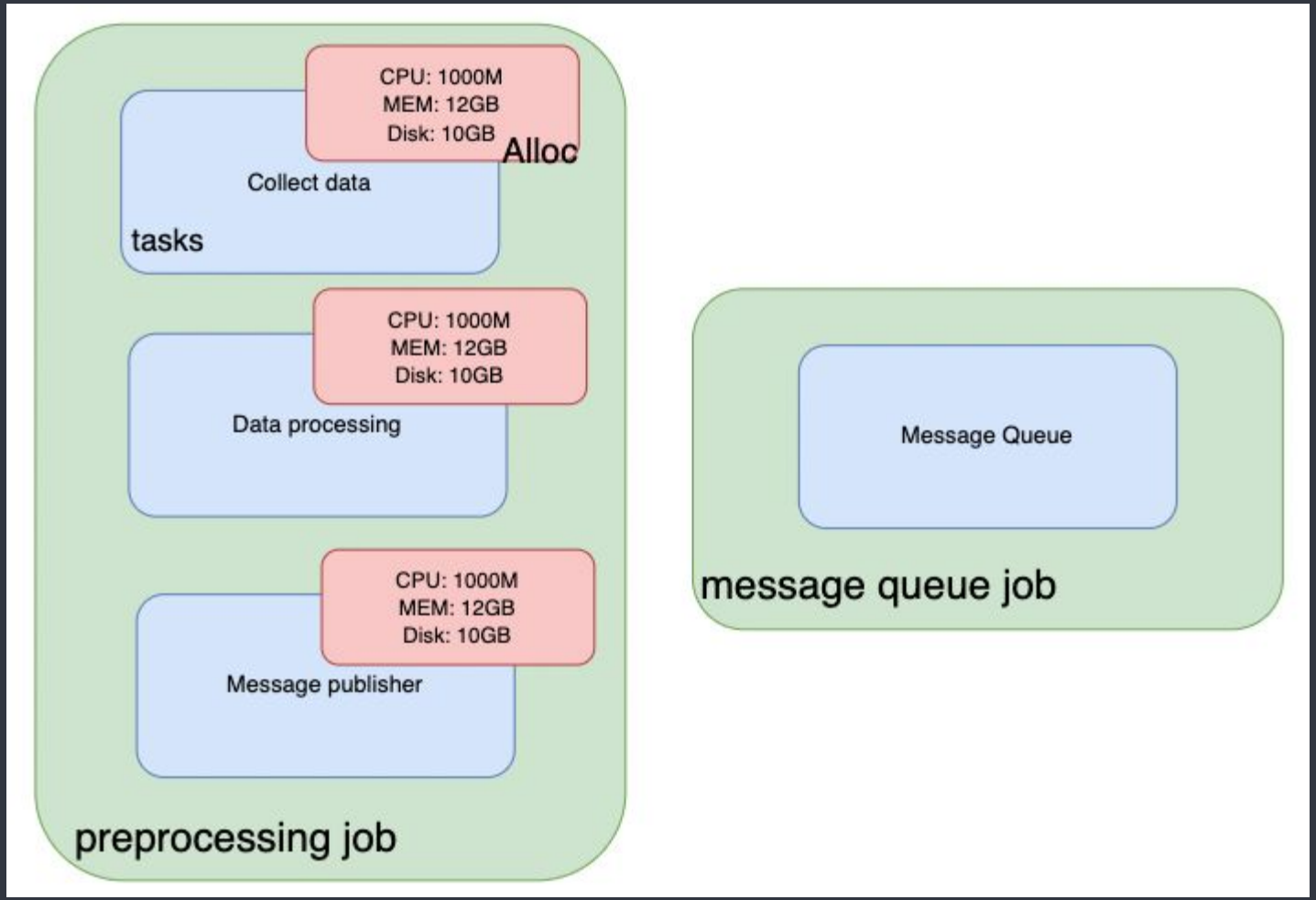


Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*


User
borgcfg
borgcfg
Cell
BorgMaster
read/UI shard
scheduler
persistent store (Paxos)
link shard
Borglet
Borglet
Borglet
Borglet
Cell
BorgMaster
read/UI shard
scheduler
persistent store (Paxos)
link shard
Borglet
Borglet
Borglet
Borglet
Data Center

Example


Collect data
Data processing
Message publisher
Collect data
Data processing
Message publisher
Collect data
Data processing
Message publisher
Message Queue
Message Consumer
ML/DL algorithm
Message Consumer
ML/DL algorithm
Message Consumer
ML/DL algorithm
DB


CPU: 1000M
MEM: 12GB
Disk: 10GB
Alloc
Collect data
tasks
CPU: 1000M
MEM: 12GB
Disk: 10GB
Data processing
CPU: 1000M
MEM: 12GB
Disk: 10GB
Message publisher
preprocessing job
Message Queue
message queue job

Borg Terminology

- Cell (Cluster)
  - A set of physical server.
- Task
  - A set of processes running in container.
- Job
  - A collection of tasks.
  - Production job (long-running jobs) and non-production job (batch jobs).
- Allocs
  - Resource reservation on a machine used by tasks.

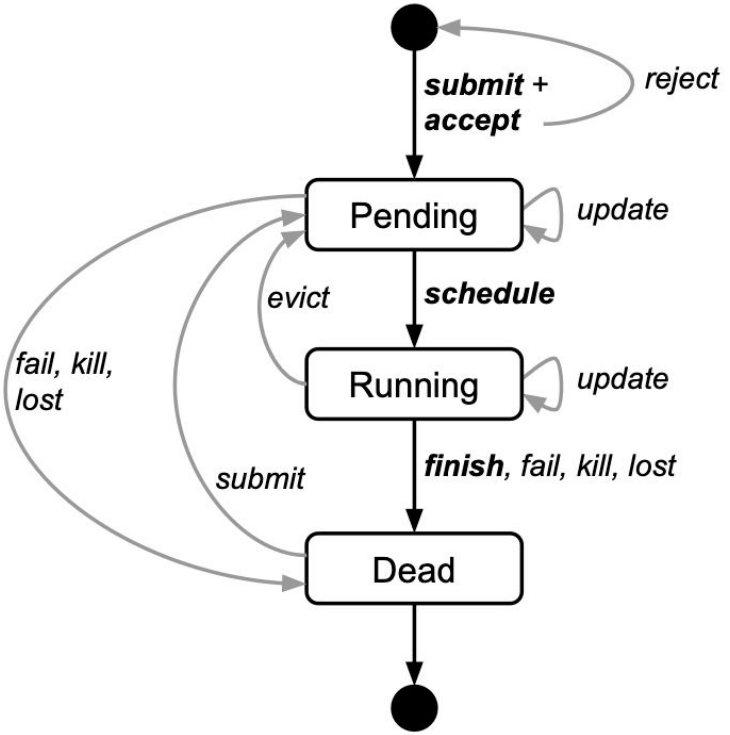
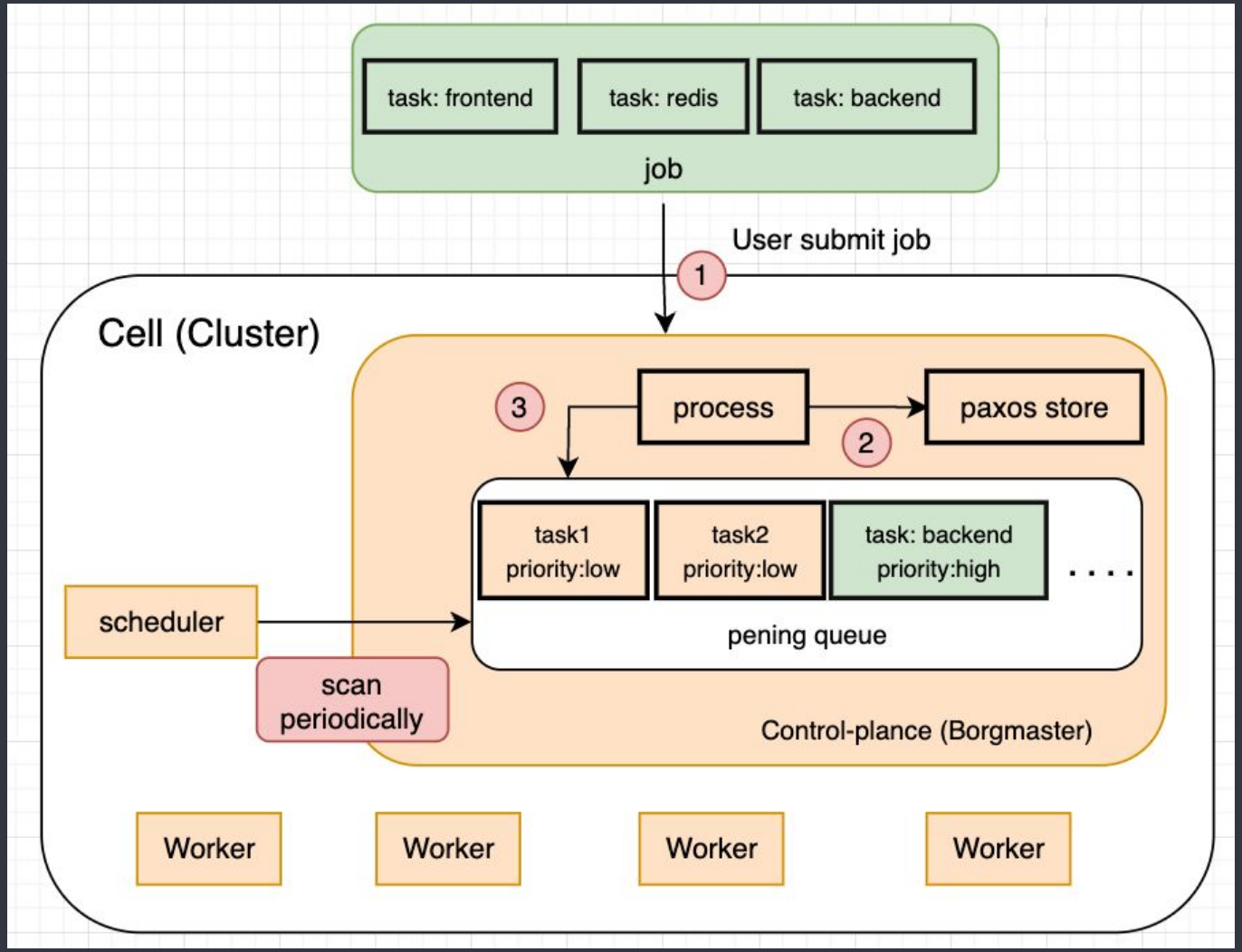


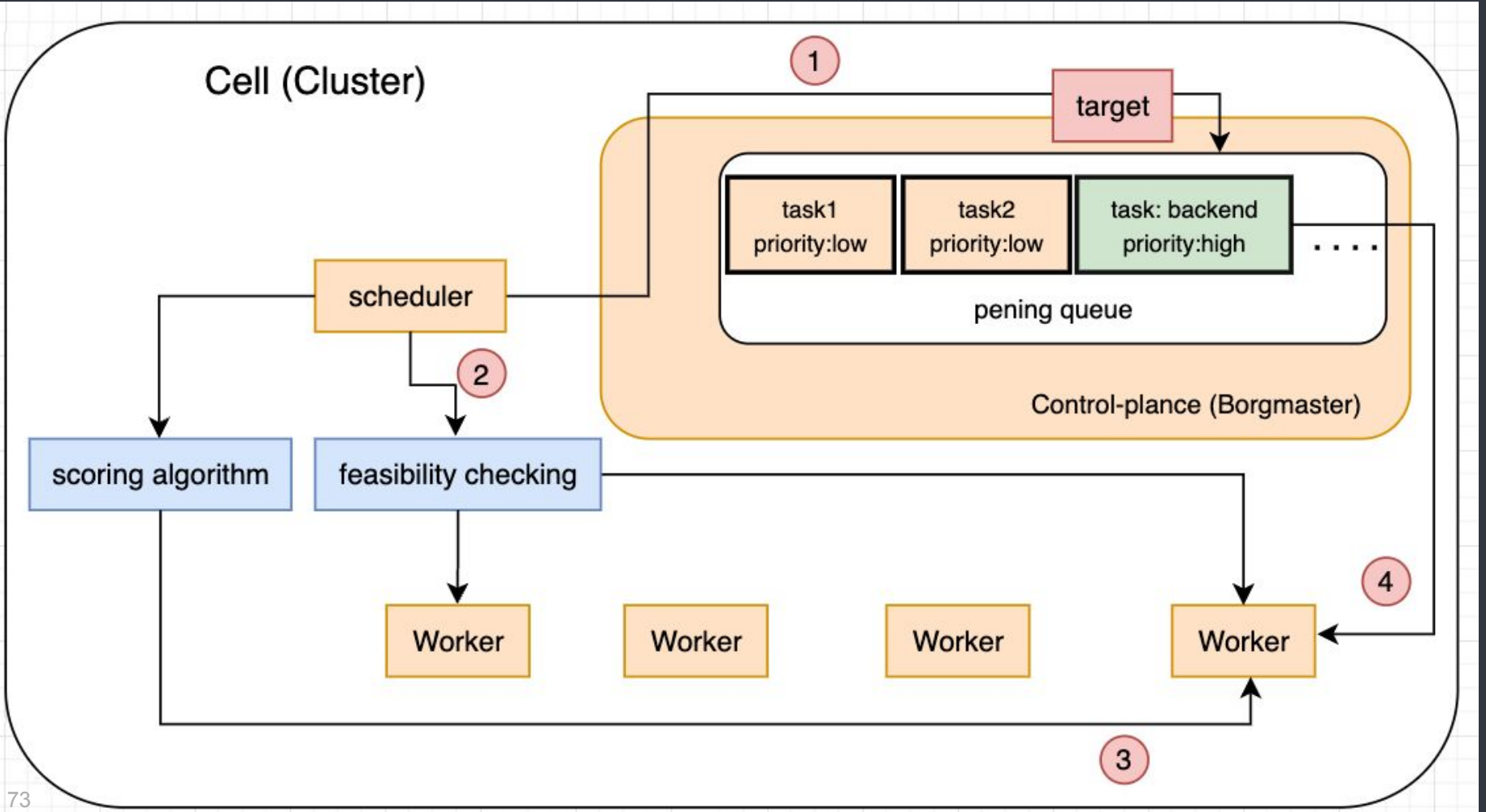
Figure 2: The state diagram for both jobs and tasks. *Users can trigger* submit, kill, *and* update *transitions.*

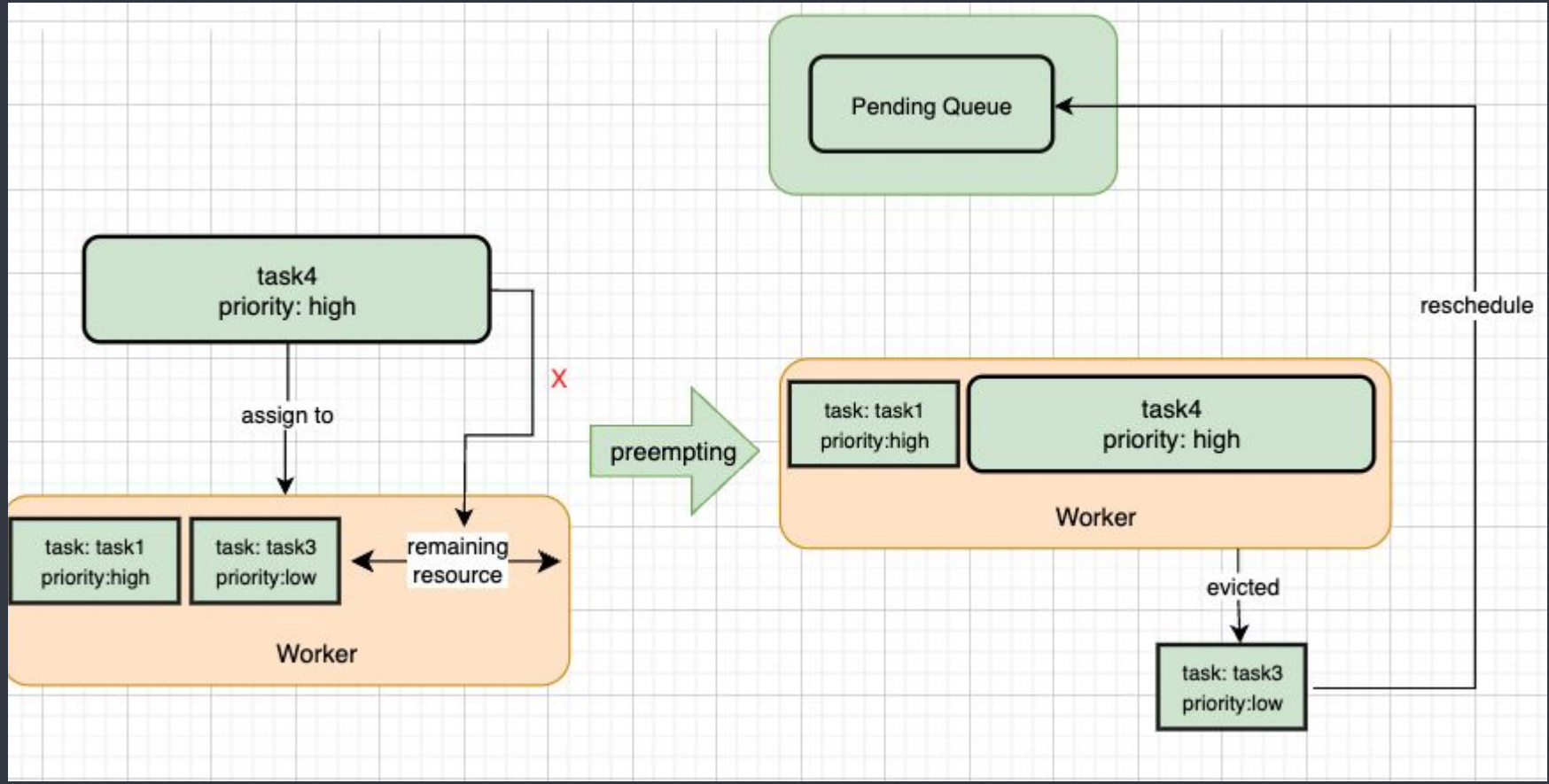
User view

```
job hello_world = {
   runtime = { cell = 'ic' }                // Cell (cluster) to run in
   binary = '.../hello_world_webserver'     // Program to run
   args = { port = '%port%' }               // Command line parameters
   requirements = {      // Resource requirements (optional)
     ram = 100M
     disk = 100M
     cpu = 0.1
   }
   replicas = 10000      // Number of tasks
}
```

Google Cloud Platform


task: frontend
task: redis
task: backend
job
User submit job
1
Cell (Cluster)
3
process
paxos store
2
task1
priority:low
task2
priority:low
task: backend
priority:high
. . . .
pening queue
scheduler
scan
periodically
Control-plance (Borgmaster)
Worker
Worker
Worker
Worker

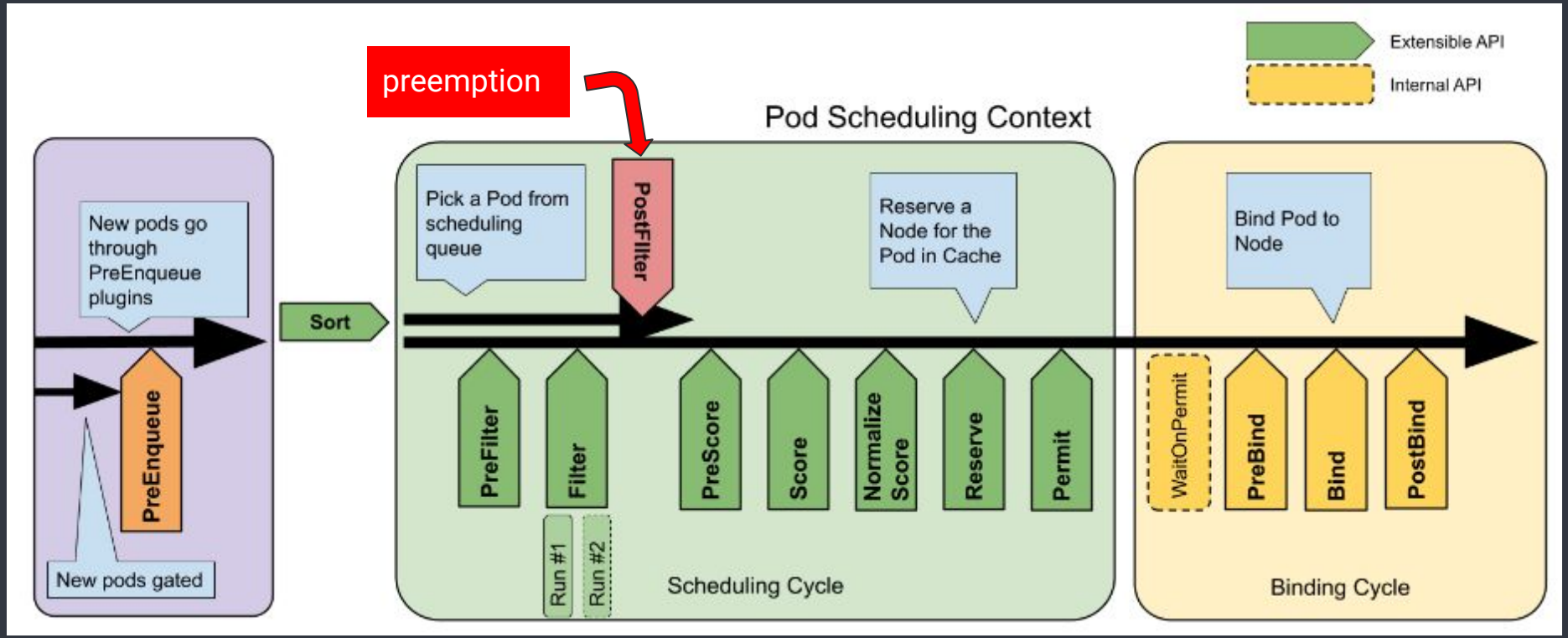

Cell (Cluster)
1
target
task1
priority:low
task2
priority:low
task: backend
priority:high
. . . .
pening queue
scheduler
2
Control-plance (Borgmaster)
scoring algorithm
feasibility checking
Worker
Worker
Worker
Worker
4
3


Pending Queue
task4
priority: high
assign to
X
task: task1
priority:high
task: task3
priority:low
remaining
resource
Worker
preempting
task: task1
priority:high
task4
priority: high
Worker
evicted
task: task3
priority:low
reschedule

Scheduling

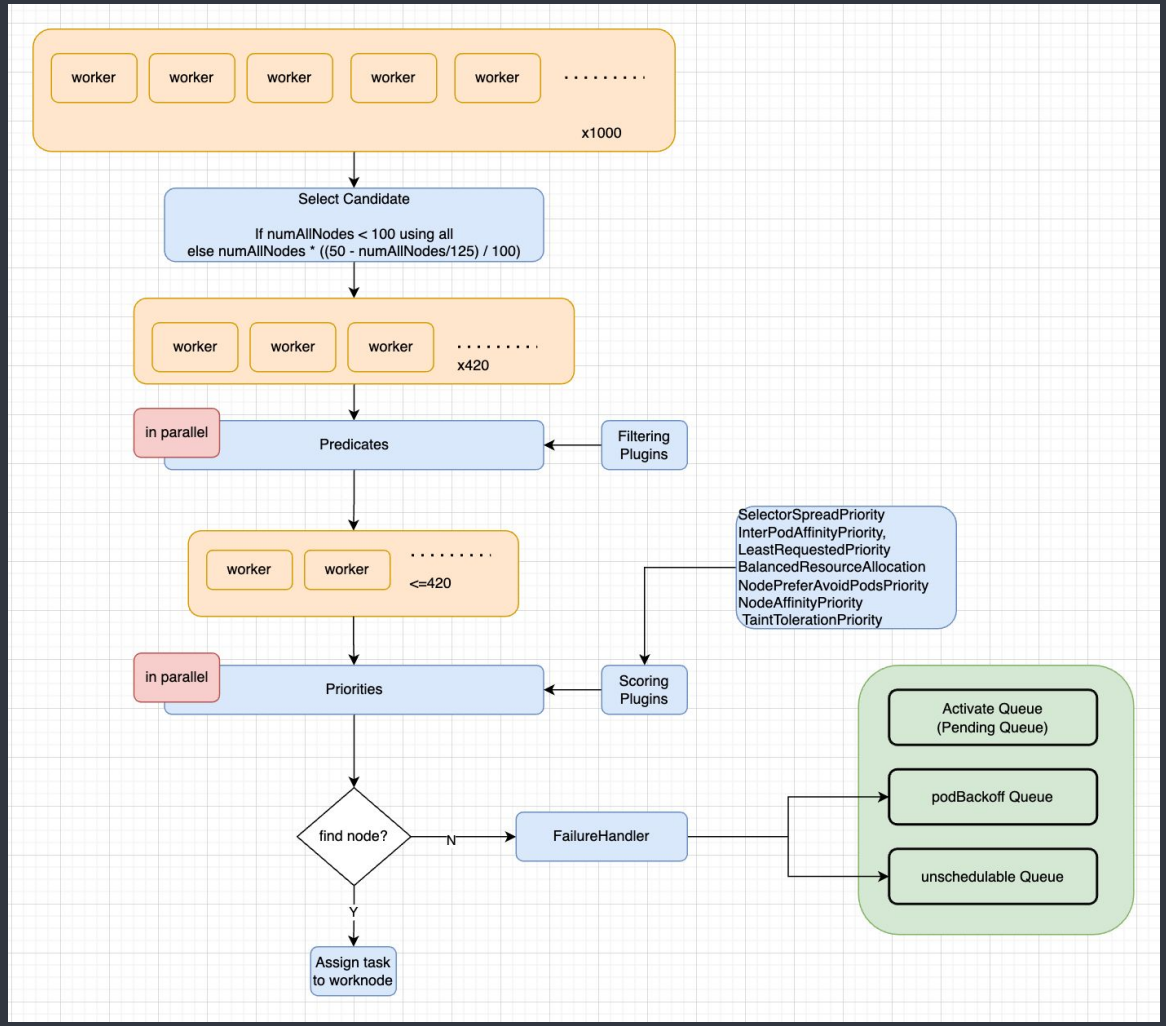
1. User submit job.
2. Borgmaster record job to Paxos store.
3. Borgmaster adds the job's tasks to the pending queue.
4. Scheduler scan pending queue periodically.
5. Select high priority task from queue, using scheduling algorithm to match machine.
6. Predicates: Using feasible checking to check which machine has enough resource.
7. Priorities: Using Score algorithm to select machine from 6.

After v1.19:

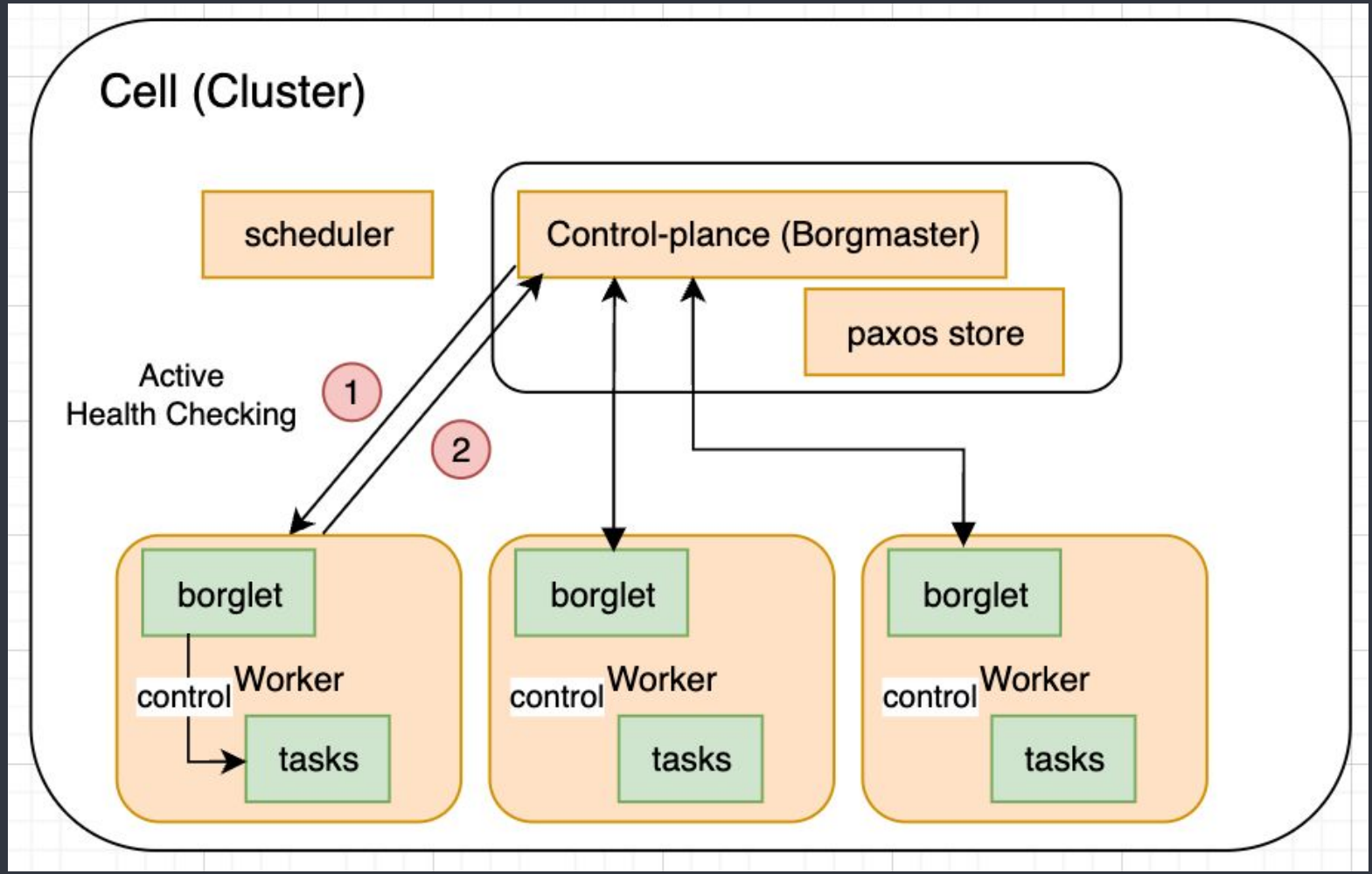


https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

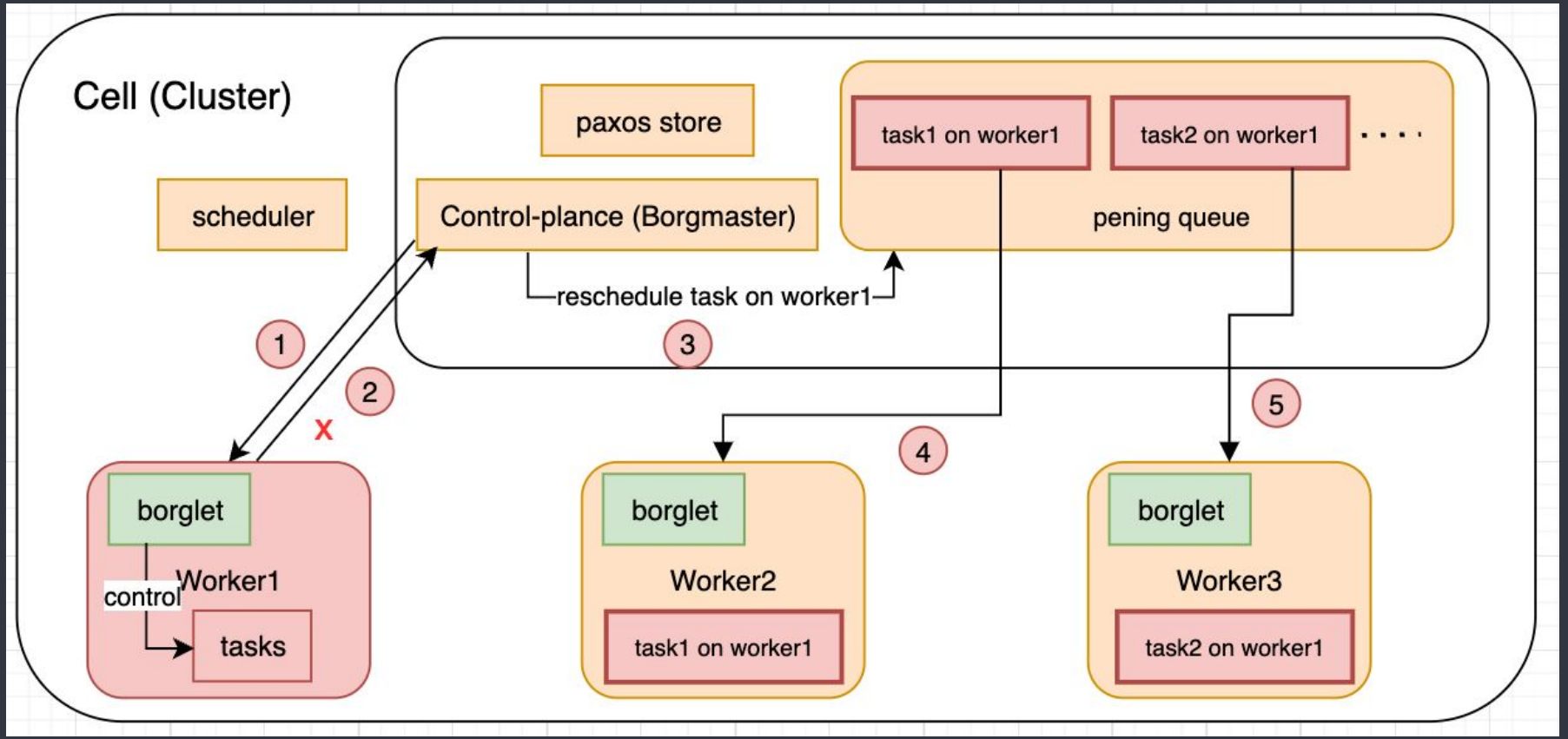
Due to time constraints, we won't discuss certain details today, such as how preemption work and skip source code review.


worker
worker
worker
worker
worker
x1000
Select Candidate
If numAllNodes < 100 using all
else numAllNodes * ((50 - numAllNodes/125) / 100)
worker
worker
worker
x420
in parallel
Predicates
Filtering Plugins
worker
worker
<=420
SelectorSpreadPriority
InterPodAffinityPriority,
LeastRequestedPriority
BalancedResourceAllocation
NodePreferAvoidPodsPriority
NodeAffinityPriority
TaintTolerationPriority
in parallel
Priorities
Scoring Plugins
find node?
N
FailureHandler
Activate Queue
(Pending Queue)
podBackoff Queue
unschedulable Queue
Y
Assign task
to worknode

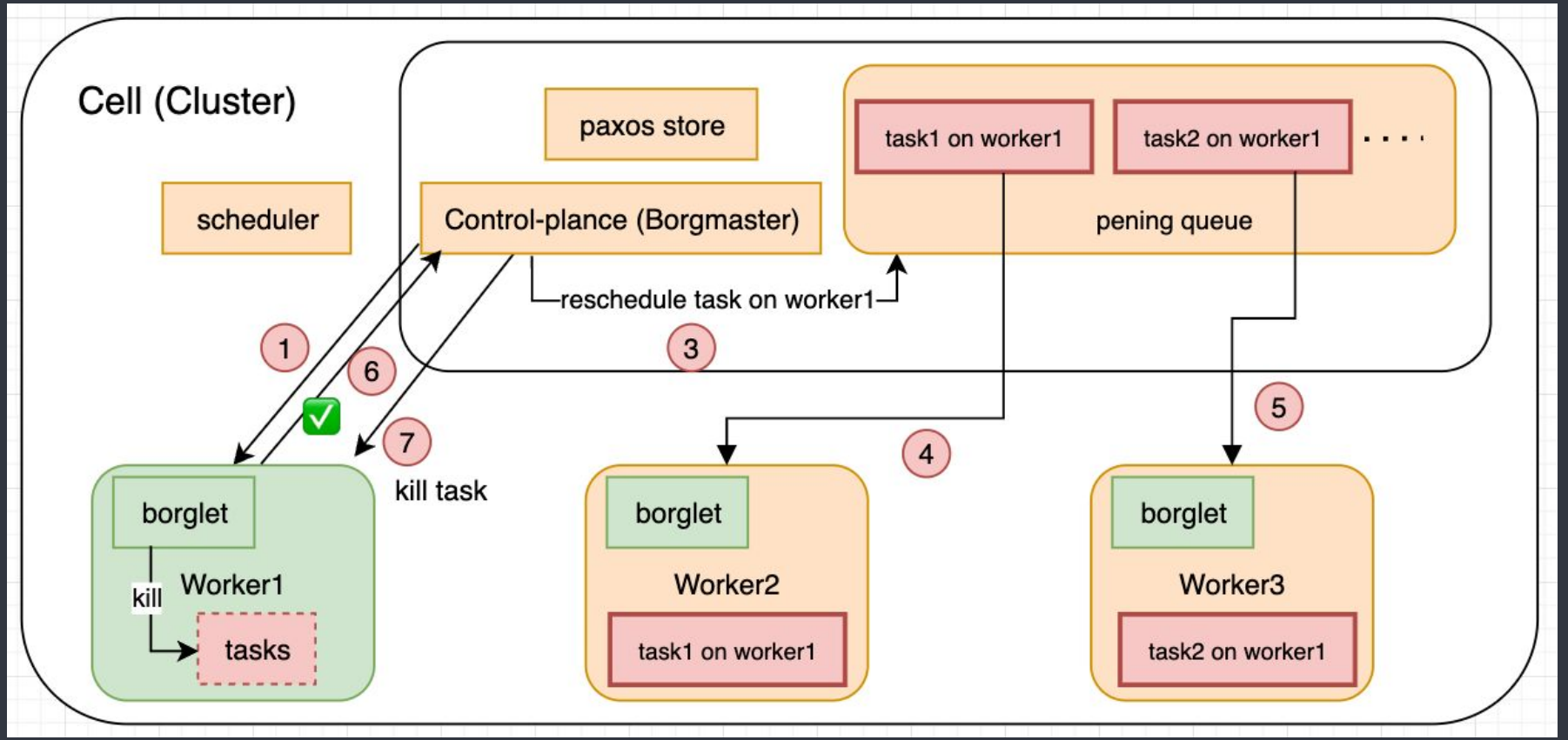
Borglet


Cell (Cluster)
scheduler
Control-plance (Borgmaster)
paxos store
Active
Health Checking
1
2
borglet
control
Worker
tasks
borglet
control
Worker
tasks
borglet
control
Worker
tasks

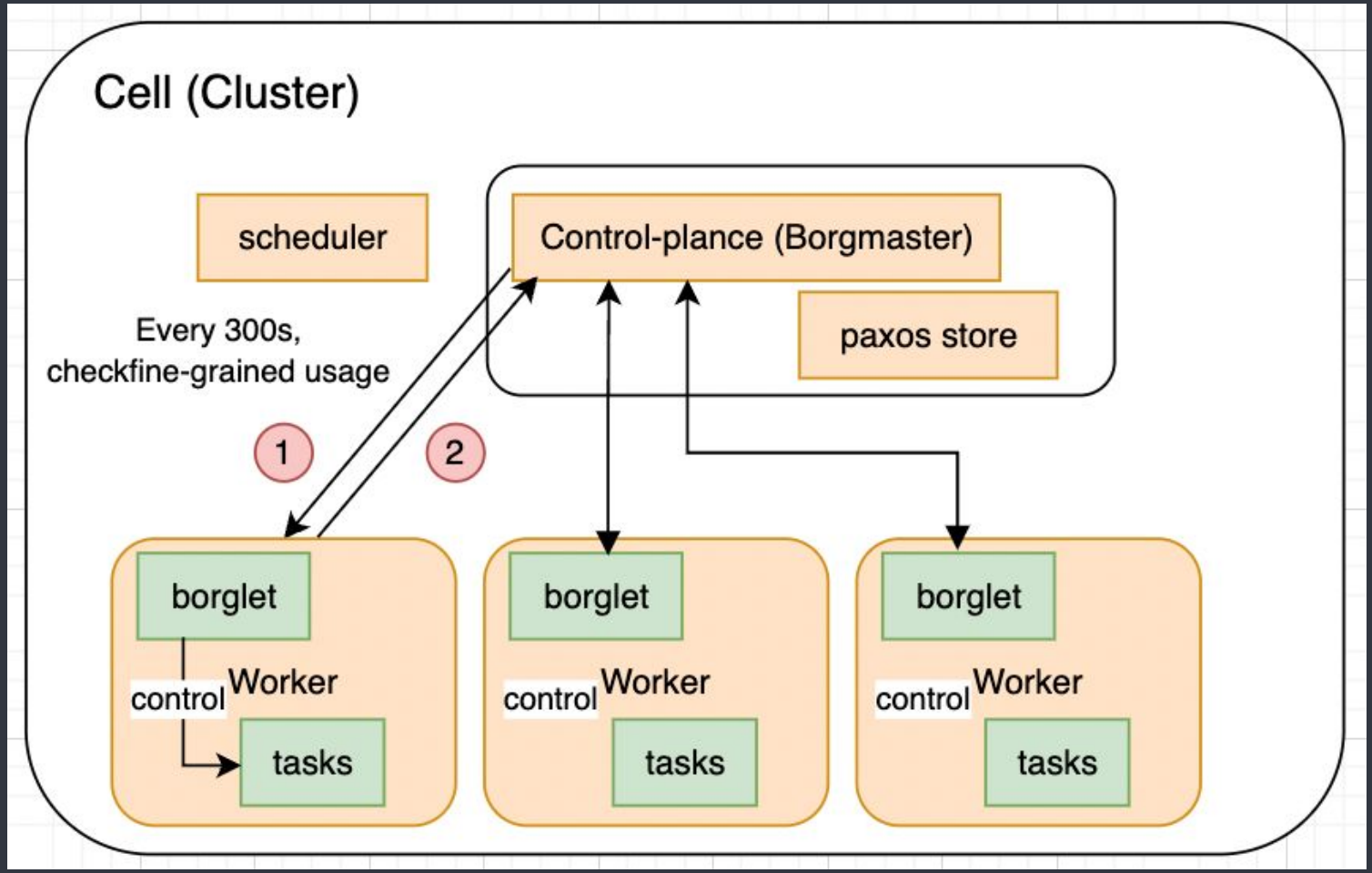
Failure detection and tolerance



Failure detection and tolerance



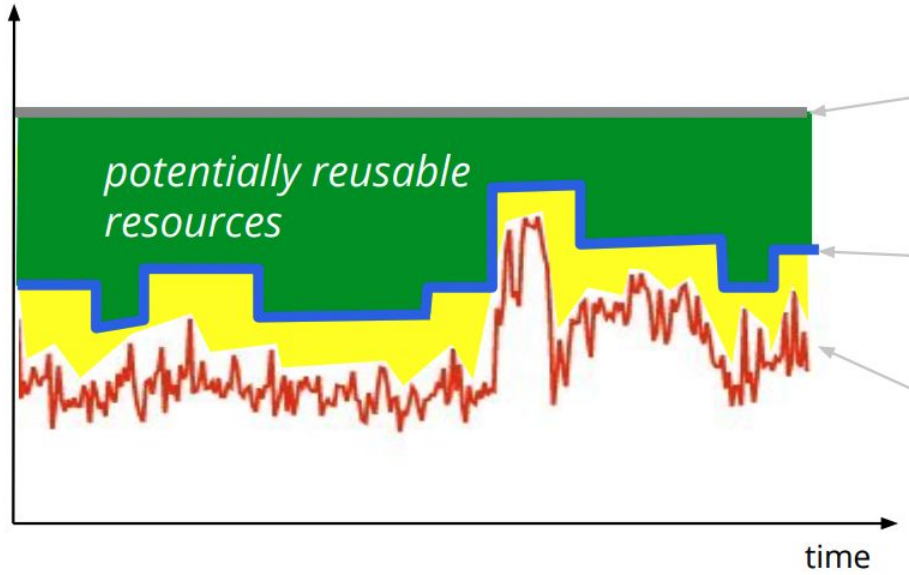
Borglet



Resource reclamation for non-production job (batch jobs).

Efficiency

Resource reclamation



limit: amount of resource requested

reservation: estimate of future usage

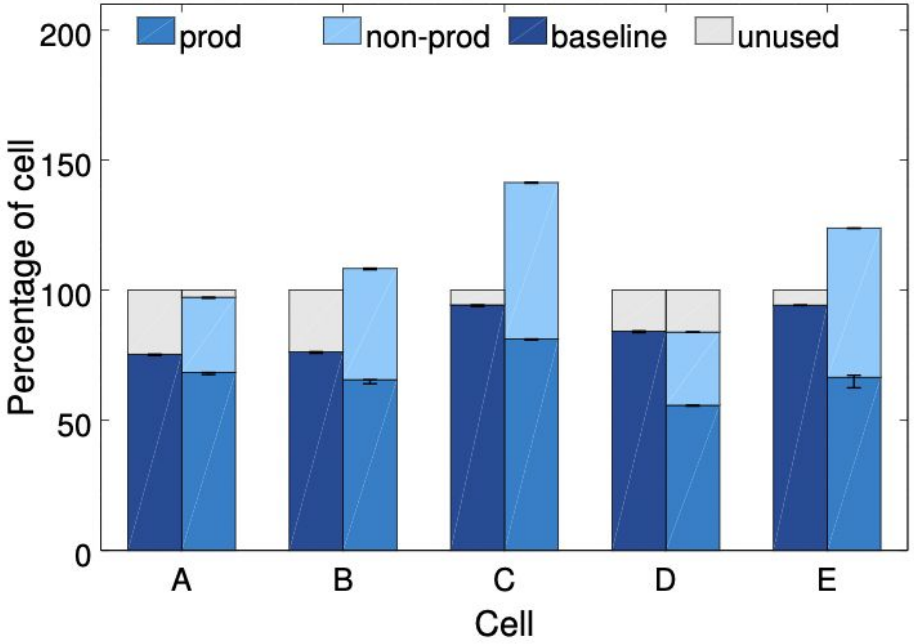
usage: actual resource consumption

Cluster Management at Google with Borg • John Wilkes • GOTO 2016

Borglet

1. Start, stop task of worker node.
2. Response for Borgmaster health checking prevent DOS.
3. Fine-grained usage (resource- consumption)

Utilization



(a) *The left column for each cell shows the original size and the combined workload; the right one shows the segregated case.*

Segregating prod and non-prod work would need 20–30% more machines in the median cell to run workload.

Utilization

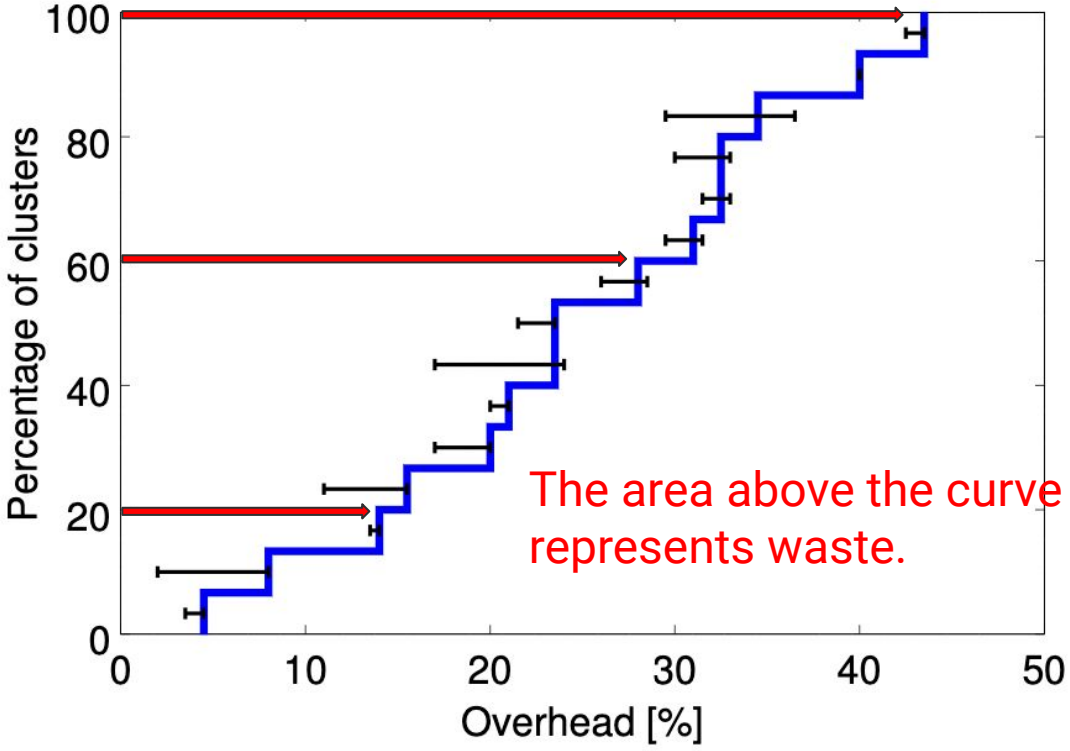
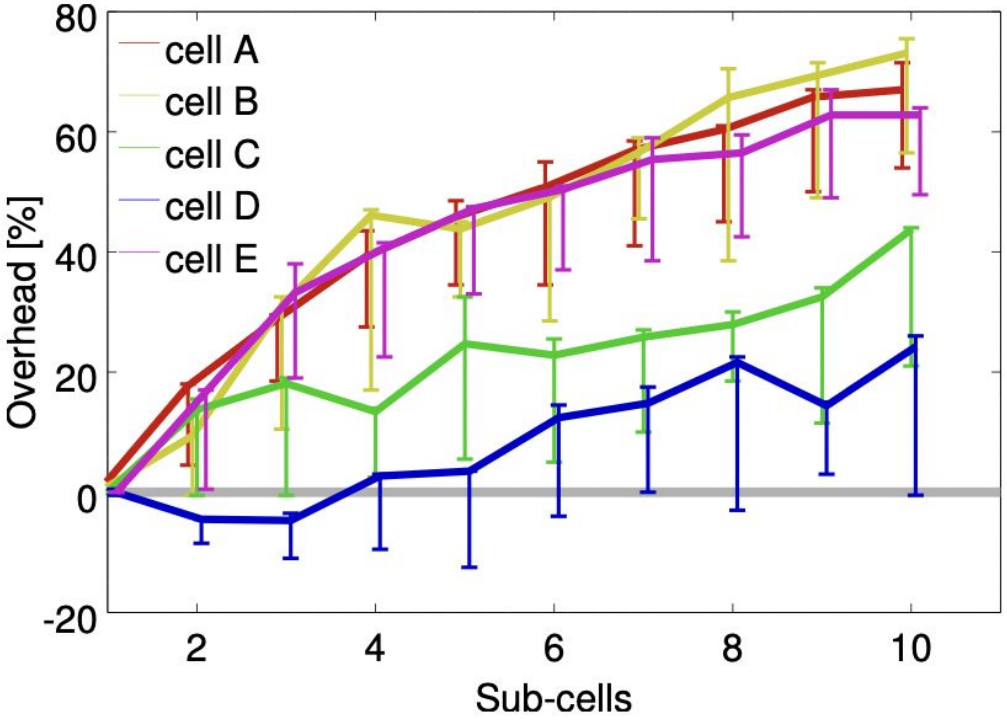


Figure 10: Resource reclamation is quite effective. *A CDF of the additional machines that would be needed if we disabled it for 15 representative cells.*

Utilization



(a) *Additional machines that would be needed as a function of the number of smaller cells for five different original cells.*

Large cells, both to allow large computations to be run, and to decrease resource fragmentation. Using smaller cells would require significantly more machines.

Reference

- DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007). Dynamo: Amazon's Highly Available Key-value Store. In SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (pp. 205-220).
- Ghemawat, S., Gobioff, H., & Leung, S. T. (2003). The Google file system. Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03.
- Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2006). Bigtable: A distributed storage system for structured data. OSDI'06: Seventh Symposium on Operating System Design and Implementation.
- Verma, A., Lantz, R., Pedrosa, L., Ranganathan, K., & Wilkes, J. (2015). Large-scale cluster management at Google with Borg.
- Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade.

Scheduling: Source code tracing (v1.30.0)

```
138    // schedulingCycle tries to schedule a single Pod.
139    func (sched *Scheduler) schedulingCycle(
140            ctx context.Context,
141            state *framework.CycleState,
142            fwk framework.Framework,
143            podInfo *framework.QueuedPodInfo,
144            start time.Time,
145            podsToActivate *framework.PodsToActivate,
146    ) (ScheduleResult, *framework.QueuedPodInfo, *framework.Status) {
147            logger := klog.FromContext(ctx)
148            pod := podInfo.Pod
149            scheduleResult, err := sched.SchedulePod(ctx, fwk, state, pod)
```

Entrypoint
Choose pod to schedule

kubernetes/pkg/scheduler/schedule_one.go

Predicates: Source code tracing (v1.30.0)

```
func (sched *Scheduler) schedulePod(...) {

        if err := sched.Cache.UpdateSnapshot(klog.FromContext(ctx), sched.nodeInfoSnapshot); …

        feasibleNodes, diagnosis, err := sched.findNodesThatFitPod(ctx, fwk, state, pod)

        if len(feasibleNodes) == 0 {
                return result, &framework.FitError{ …

        // When only one node after predicate, just use it.
        priorityList, err := prioritizeNodes(ctx, sched.Extenders, fwk, state, pod, feasibleNodes)

        host, _, err := selectHost(priorityList, numberOfHighestScoredNodesToReport)

        trace.Step("Prioritizing done")
}
```


kubernetes/pkg/scheduler/schedule_one.go L390

Predicates: Source code tracing (v1.30.0)

```
// Filters the nodes to find the ones that fit the pod based on the framework
// filter plugins and filter extenders.
func (sched *Scheduler) findNodesThatFitPod(...) {
	allNodes, err := sched.nodeInfoSnapshot.NodeInfos().List()

	// Run "prefilter" plugins.
	preRes, s := fwk.RunPreFilterPlugins(ctx, state, pod)

	feasibleNodes, err := sched.findNodesThatPassFilters(ctx, fwk, state, pod, &diagnosis, nodes)

	feasibleNodesAfterExtender, err := findNodesThatPassExtenders(ctx, sched.Extenders, pod, feasibleNodes,
diagnosis.NodeToStatusMap)

	return feasibleNodesAfterExtender, diagnosis, nil
}
```


kubernetes/pkg/scheduler/schedule_one.go L442

Predicates: Source code tracing (v1.30.0)

```
$ tree -Ld 1 pkg/scheduler/framework/plugins
pkg/scheduler/framework/plugins
├── defaultbinder
├── defaultpreemption
├── dynamicresources
├── feature
├── helper
├── imagelocality
├── interpodaffinity
├── names
├── nodeaffinity
├── nodename
├── nodeports
├── noderesources
├── nodeunschedulable
├── nodevolumelimits
├── podtopologyspread
.
.
.
```

default plugins

kubernetes/pkg/scheduler/framework/plugins

Predicates: Source code tracing (v1.30.0)

```
// findNodesThatPassFilters finds the nodes that fit the filter plugins.
func (sched *Scheduler) findNodesThatPassFilters(...) {
        numNodesToFind := sched.numFeasibleNodesToFind(fwk.PercentageOfNodesToScore(), int32(numAllNodes))

        errCh := parallelize.NewErrorChannel()
        checkNode := func(i int) {
                nodeInfo := nodes[(sched.nextStartNodeIndex+i)%numAllNodes]
                status := fwk.RunFilterPluginsWithNominatedPods(ctx, state, pod, nodeInfo)
                if status.Code() == framework.Error { … }
                if status.IsSuccess() { … }
                …
        }

        fwk.Parallelizer().Until(ctx, numAllNodes, checkNode, metrics.Filter)
        feasibleNodes = feasibleNodes[:feasibleNodesLen]

        return feasibleNodes, nil
}
```

1. Select part of nodes
2. Filtering in parallel

kubernetes/pkg/scheduler/schedule_one.go L576

Predicates: Source code tracing (v1.30.0)

```
// Filters the nodes to find the ones that fit the pod based on the framework
// filter plugins and filter extenders.
func (sched *Scheduler) numFeasibleNodesToFind(...) {
        if numAllNodes < minFeasibleNodesToFind {     return numAllNodes }
        if percentage == 0 {
                percentage = int32(50) - numAllNodes/125
                if percentage < minFeasibleNodesPercentageToFind {
                        percentage = minFeasibleNodesPercentageToFind
                }
        }
        numNodes = numAllNodes * percentage / 100
        if numNodes < minFeasibleNodesToFind {
                return minFeasibleNodesToFind
        }

        return numNodes
}
```

If numAllNodes < 100 using all
else numAllNodes * ((50 - numAllNodes/125) / 100)

kubernetes/pkg/scheduler/schedule_one.go L661

Priorities: Source code tracing (v1.30.0)

```
func prioritizeNodes(...)  {
        // checking env …

        // Run PreScore plugins.
        preScoreStatus := fwk.RunPreScorePlugins(ctx, state, pod, nodes)

        // Run the Score plugins.
        nodesScores, scoreStatus := fwk.RunScorePlugins(ctx, state, pod, nodes)

        if len(extenders) != 0 && nodes != nil {
                // allNodeExtendersScores has all extenders scores for all nodes.
                // It is keyed with node name.
                …
        }

        return nodesScores, nil
}
```

Run prescore, score and extender if user specify
Prescore unrelated to score, load data for scoring

kubernetes/pkg/scheduler/schedule_one.go L748

Priorities: Source code tracing (v1.30.0)