Consistency models

IPADS, Shanghai Jiao Tong University

https://www.sjtu.edu.cn

Credits: Rong Chen@IPADS, jinyang Li@NYU

Review: key-value storage (KVS)

Storage abstraction:

- Each data (**V**alue) is opaque to the underlying storage/database
  - The K and V can be arbitrary byte-sequence (e.g., JSON, int, string)
- Indexed by a key (**K**), which itself is also a data
- Stored on disk (tolerate failure & support a large capacity)

Application-level Interface (API)

- Get(K) -> V, Scan(K,N)
- Update(K,V), Insert(K,V), Delete(K,V)

LEVELDB RocksDB redis

Review: naïve KVS

Storage abstraction remapping:

- Key ✉ The file name
    - Assume the key is not so long
- Value ✉ The file content
- So we can store each K,V as a file ⏪

Application-level Interface (API)

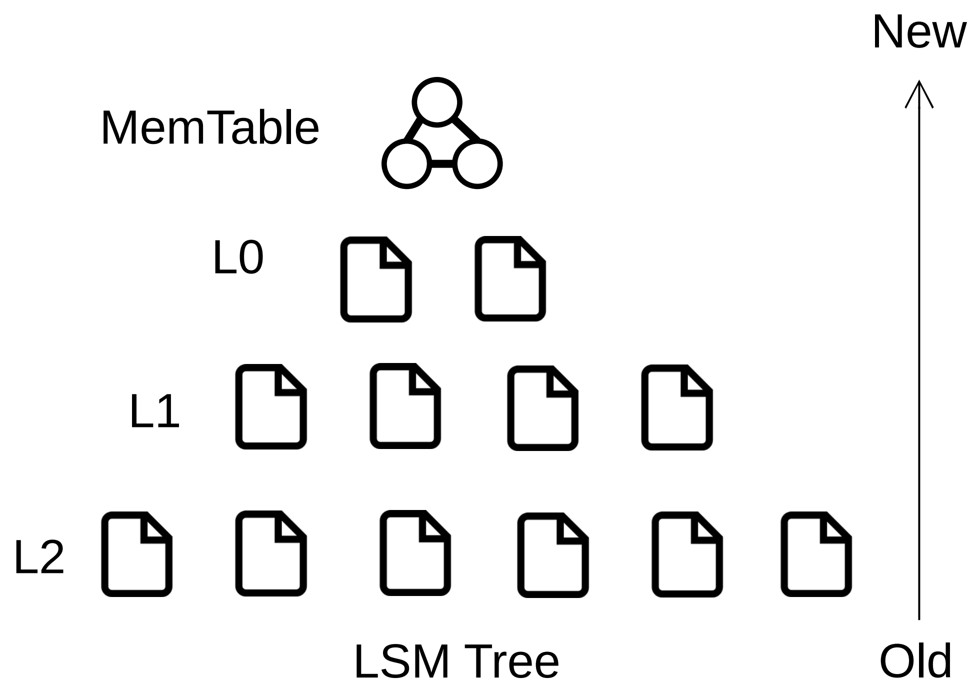
- Get(K) -> V is similar to OPEN(…) + READ(…)
- Update(K,V) -> is similar to OPEN(…) + WRITE()
- Insert(K,V) -> is similar to CREATE(…) + WRITE(…)
- Etc.

LSM Tree organizes SSTables in a hierarchy

The famous Log-structured merge tree

Each layer has the entire KVS data of some time

- Each layer has maximum size
- Except L0, all files in layers are **sorted, and does not have duplicated keys**



Hierarchy can speed up old value lookup

Each layer has only one file that store the key (except L0)

– So we can search only one file per-layer (not per-file search)

Question: how to find the key file?

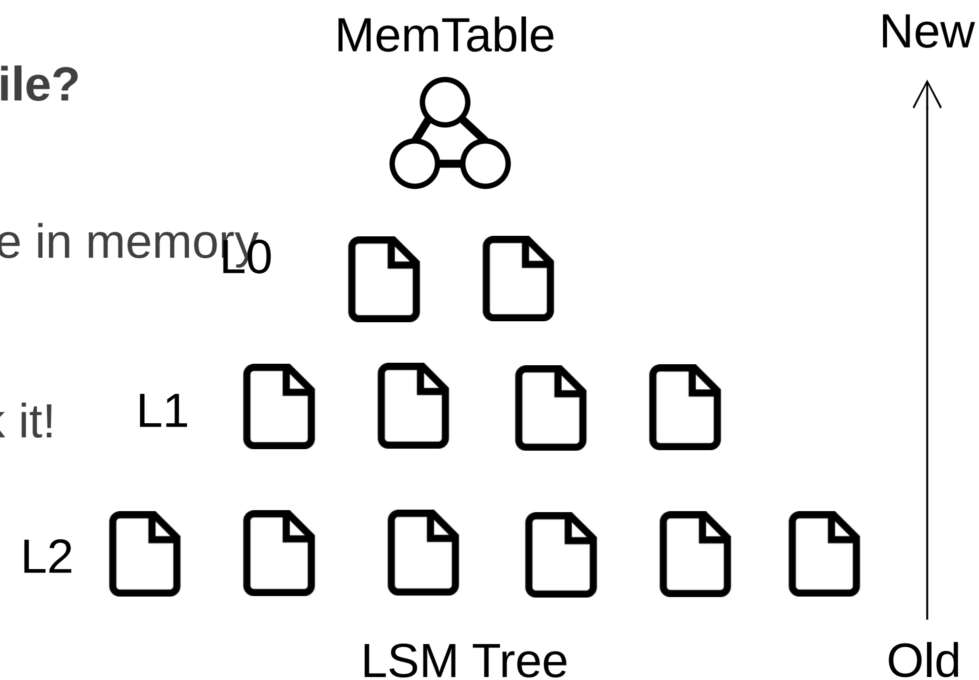
– Many methods exist

– E.g., store a [min, max] per file in memory

Read upon lower layers

– First identify the file, then look it!

– O(1) access per layer

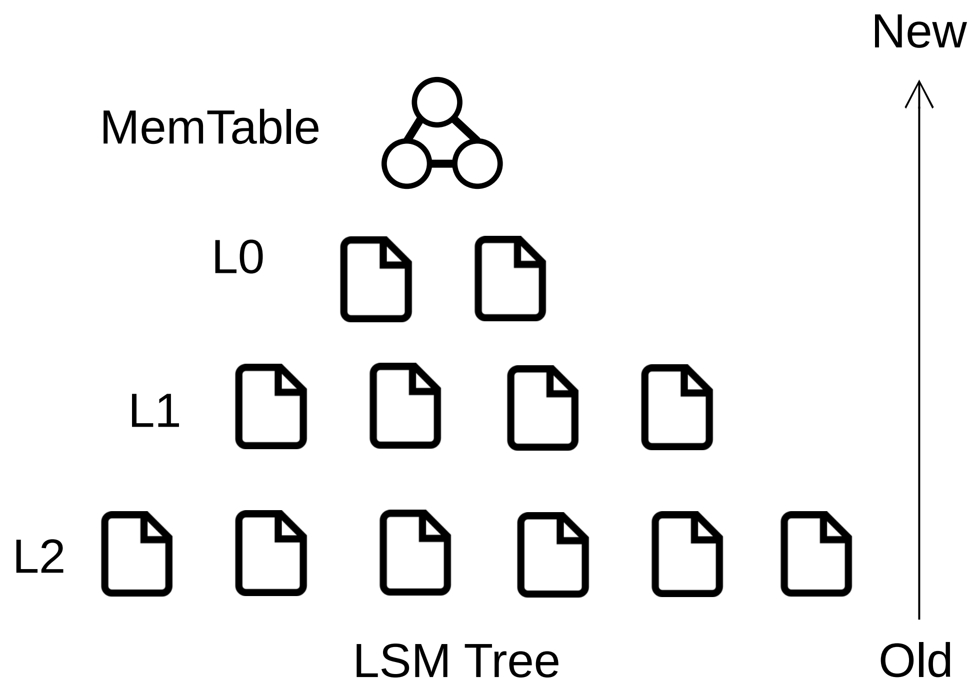


Hierarchy can speed up range query

Store a [min, max] per file

Range query

– Query layer-by-layer
– Then merge the results



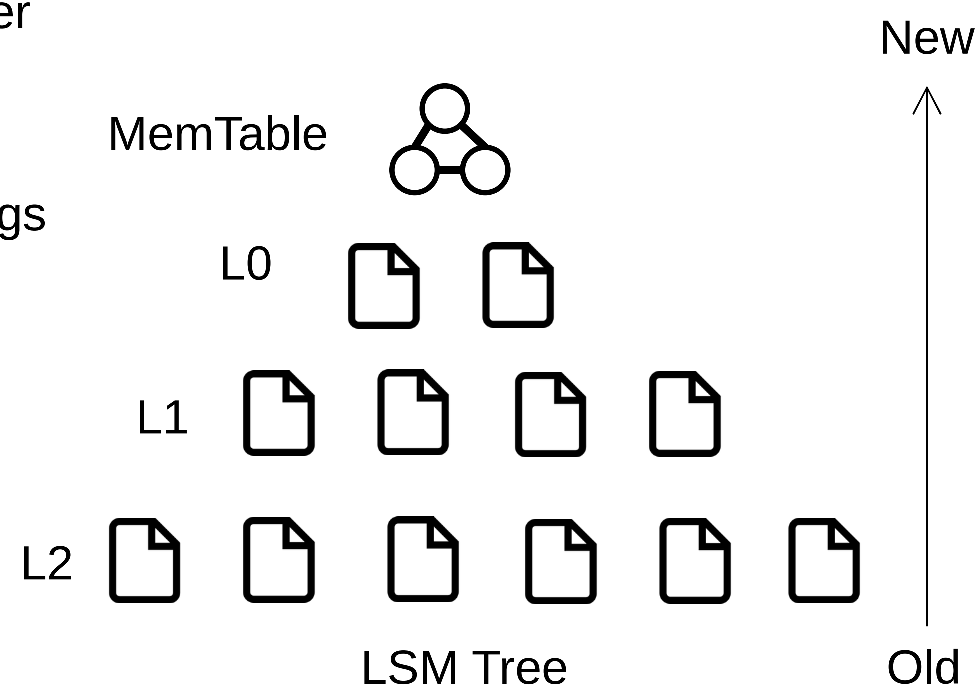
Range Query in LSM Tree

1. Search each layer using **binary search**

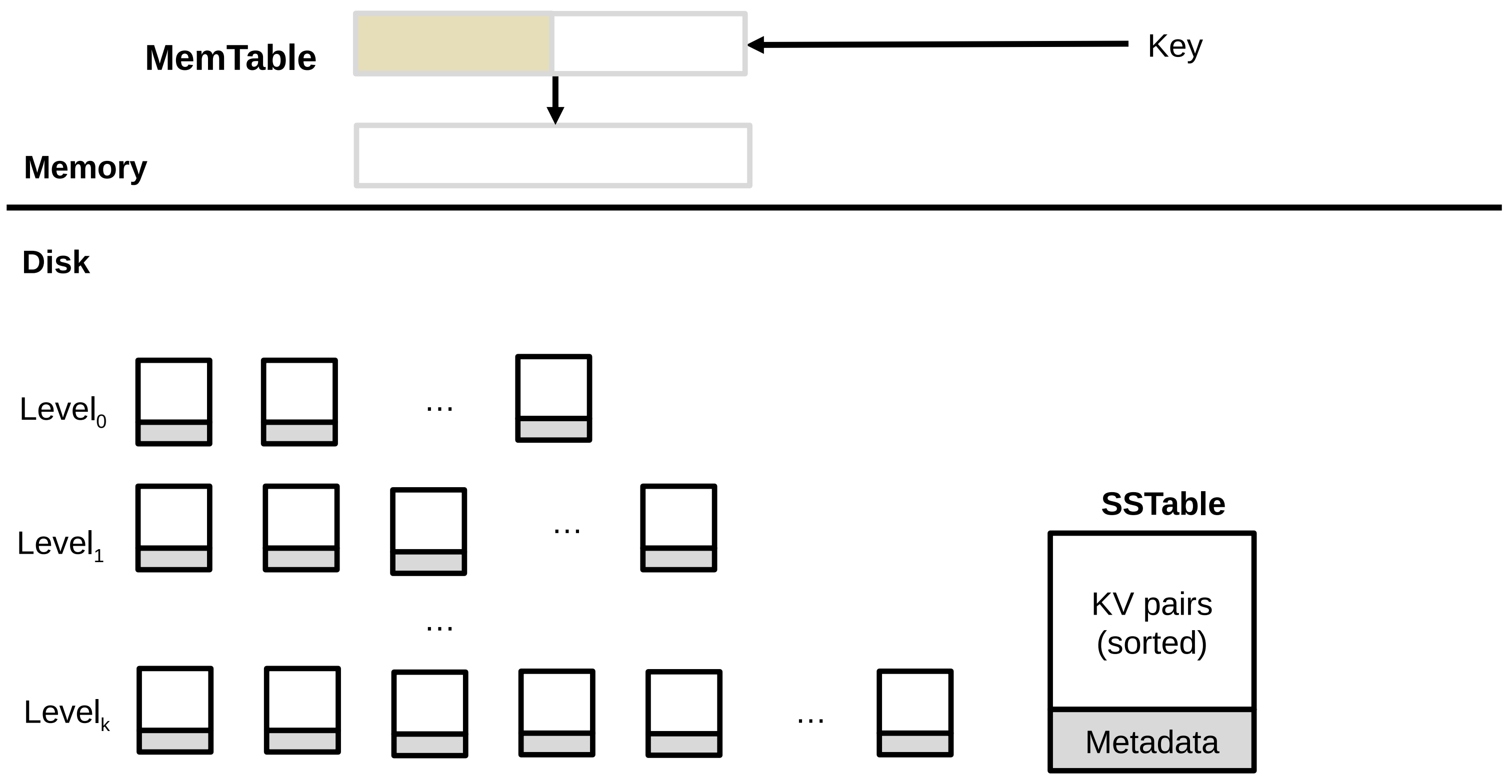
2. **Merge** the results of each layer

Not as good as B+Tree,

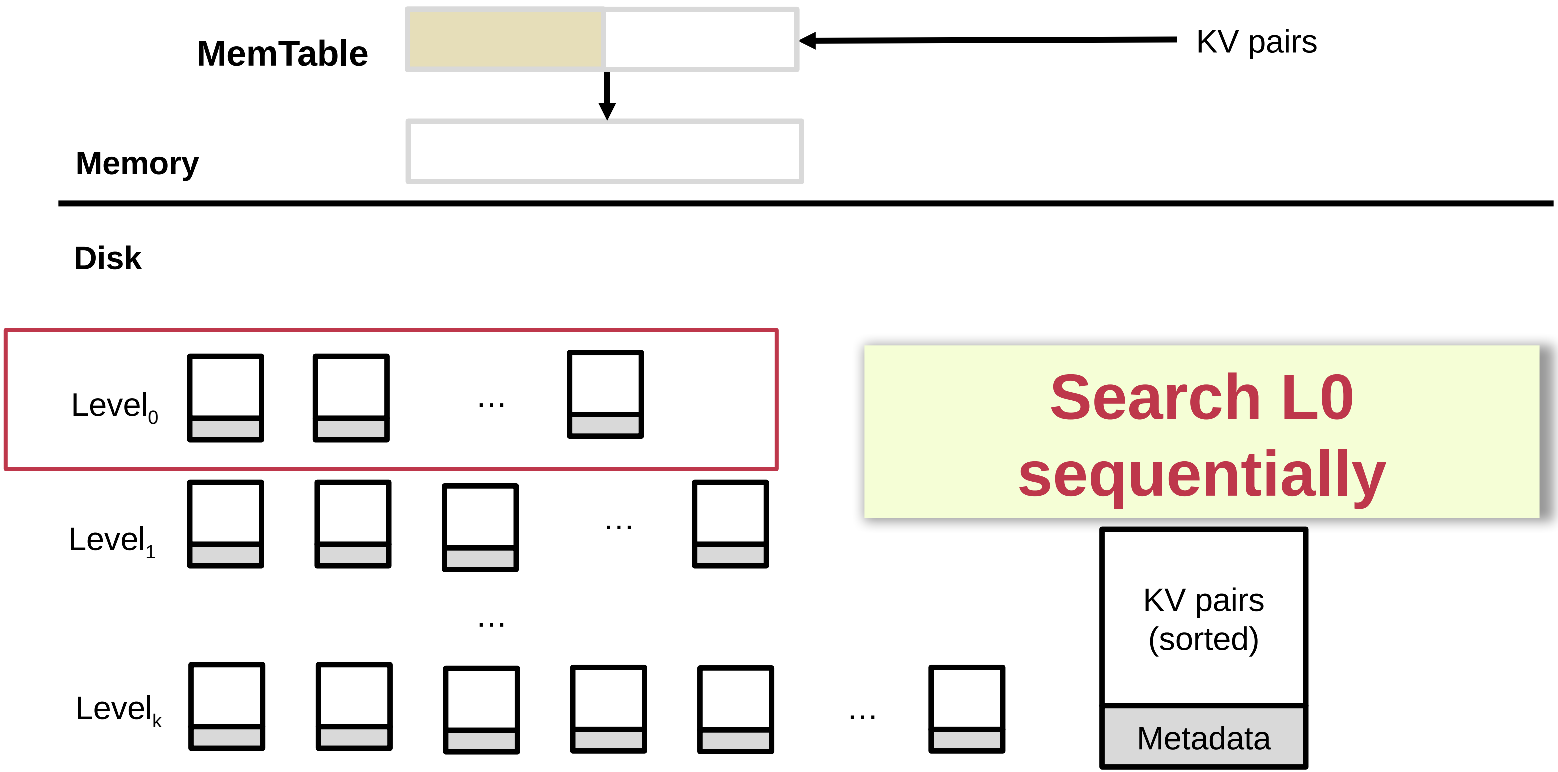
but is much better than hash & logs without hierarchy! !



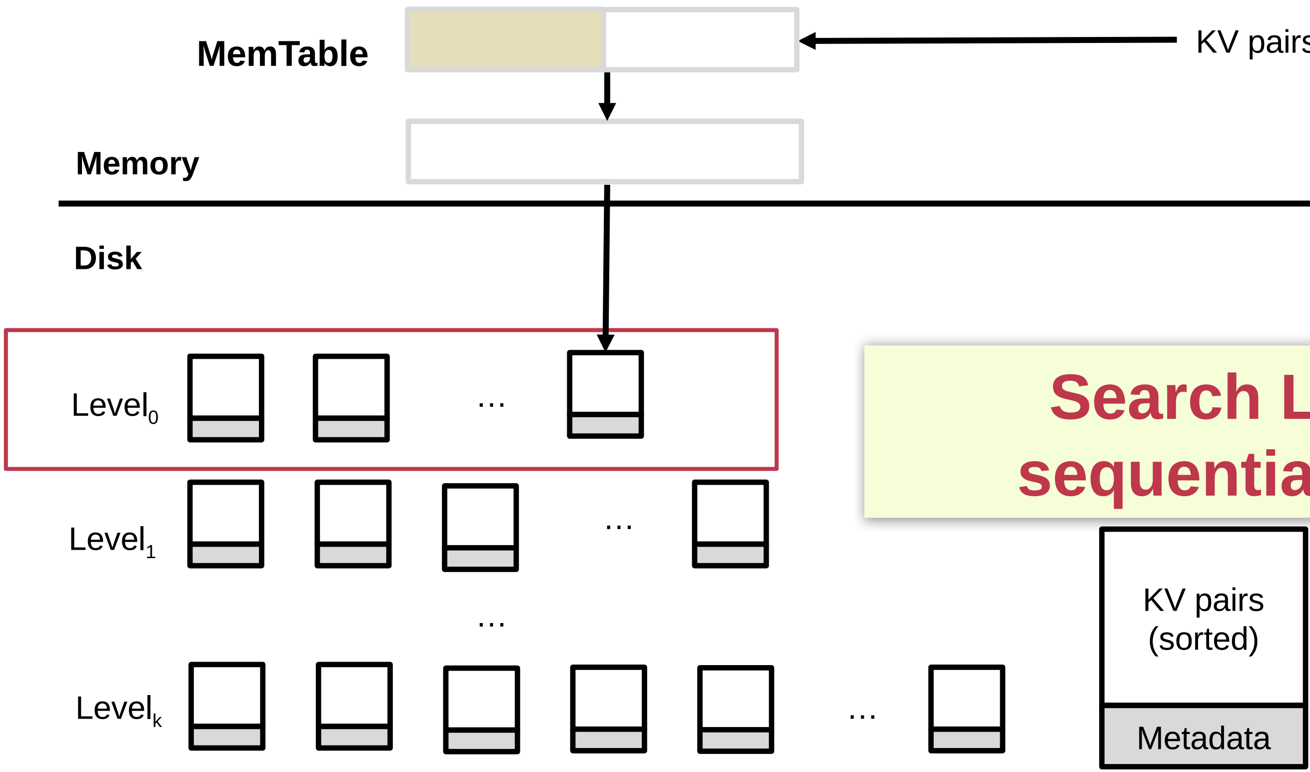
Example: Read in LSM-Tree (Similar to SSTables)



Example: Read in LSM-Tree

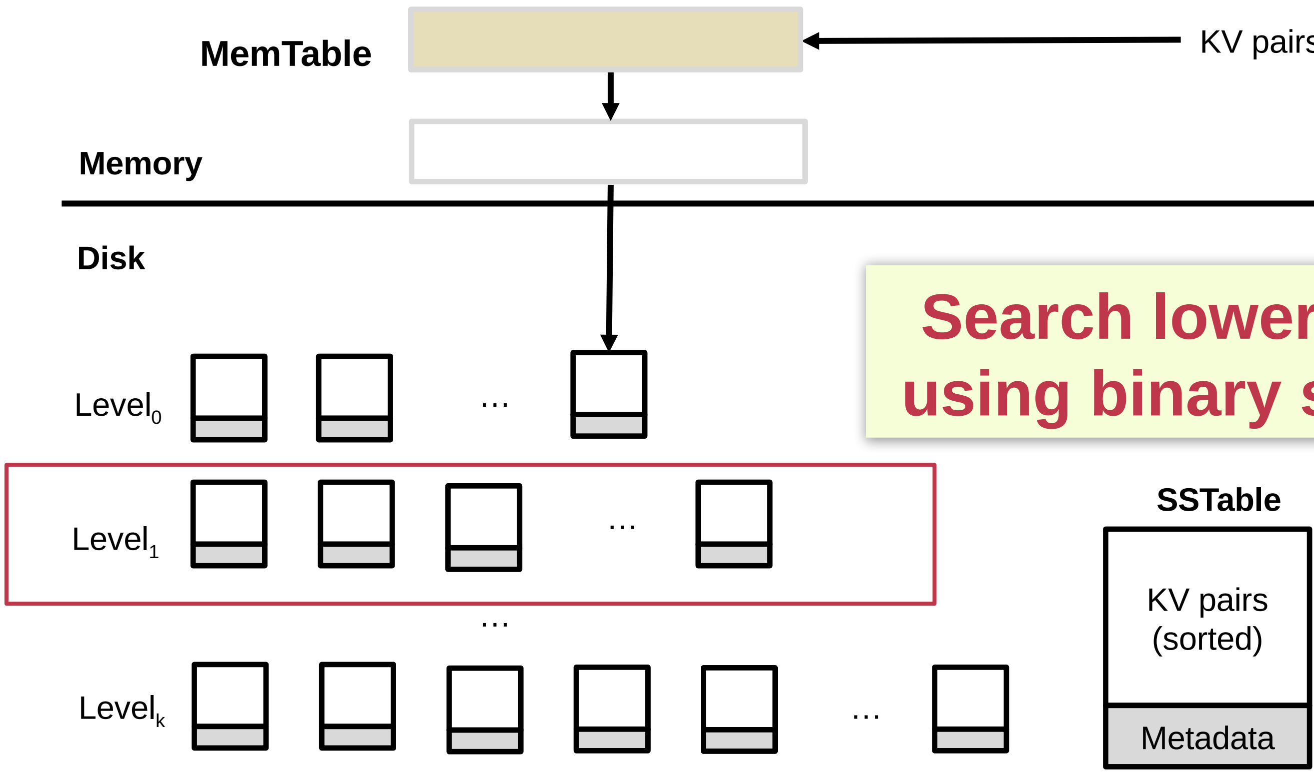


Example: Read in LSM-Tree



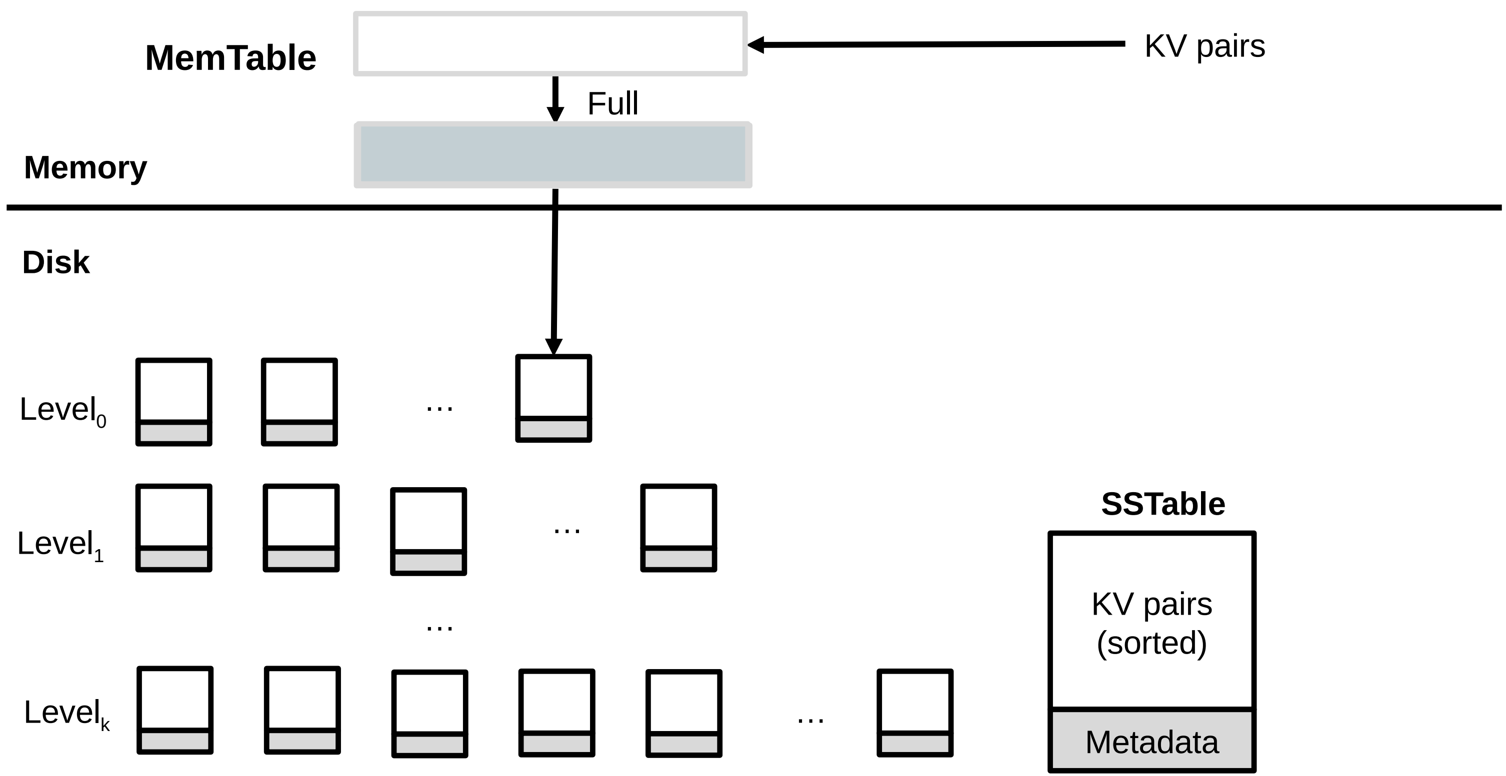
Search L0 sequentially

Example: Read in LSM-Tree

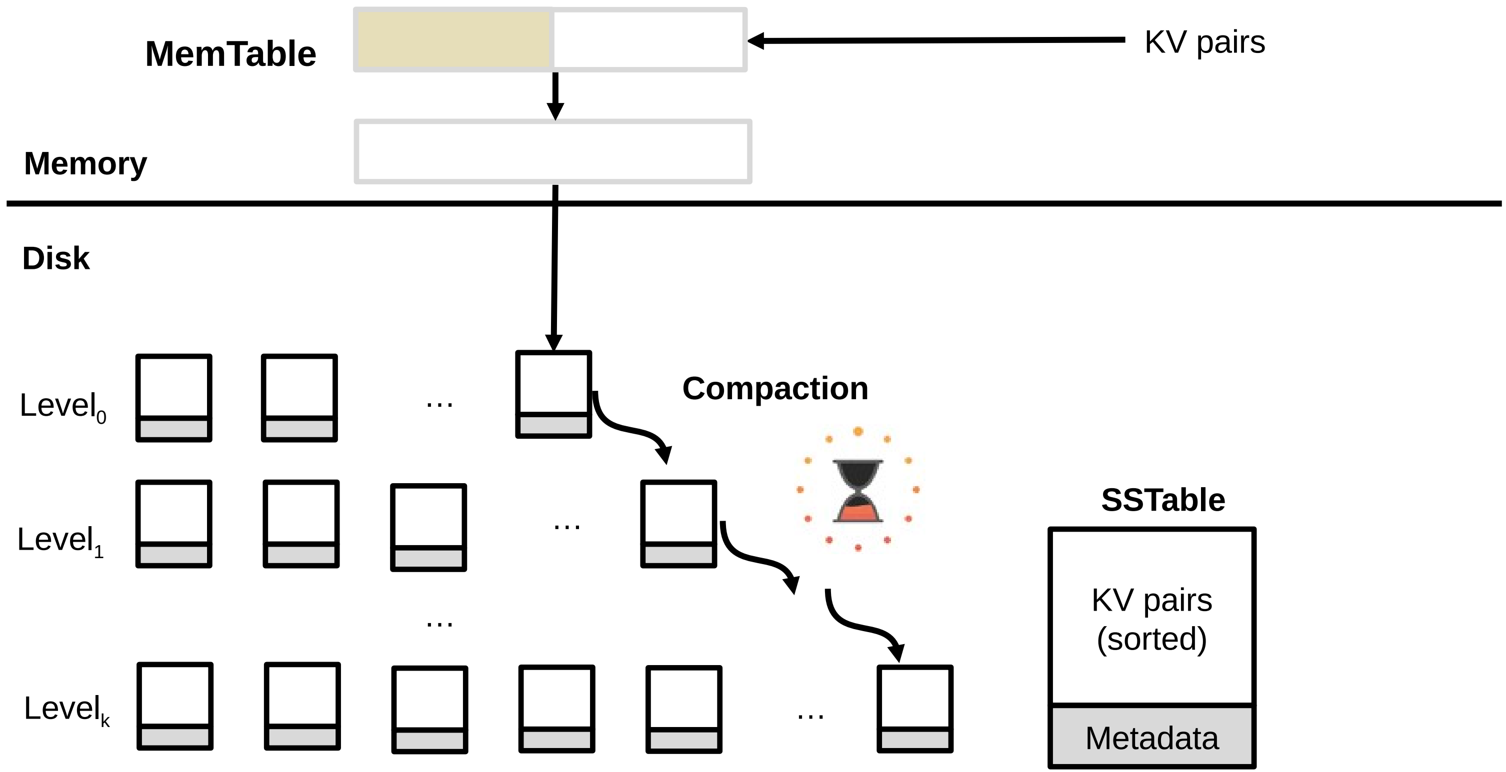


Search lower layer using binary search

Example: insert/update in LSM-Tree



Example: insert/update in LSM-Tree



What about crash?

MemTable is an in-memory data structure

- So it is vulnerable to machine failure

Goal: a successful insertion will store the data durably

Solution:

- Keep a separate log file the MemTable (may not be sorted)
- Before inserting to the MemTable, adding the KV to the log first (also a sequential write); reply if the log is successful
- If the machine crashed, reboot it, and reconstruct the MemTable from the log

LSM Tree Summary

Good when

- Massive dataset
- Rapid updates/insertions
- Fast single-point lookup for recently updated data

Widely adopted in modern single-node key-value stores

LEVELDB

Google Cloud Bigtable

RocksDB

cassandra

LSM Tree Summary

Good when

- Massive dataset
- Rapid updates/insertions
- Fast single-point lookup for recently updated data

Compared with B-Tree

- **Pros**: good write performance due to sequential writes
- **Cons**: additional compaction process, possible slow range queries, write stall caused by the compaction, slow lookup for non-existent key

LSM Tree Summary

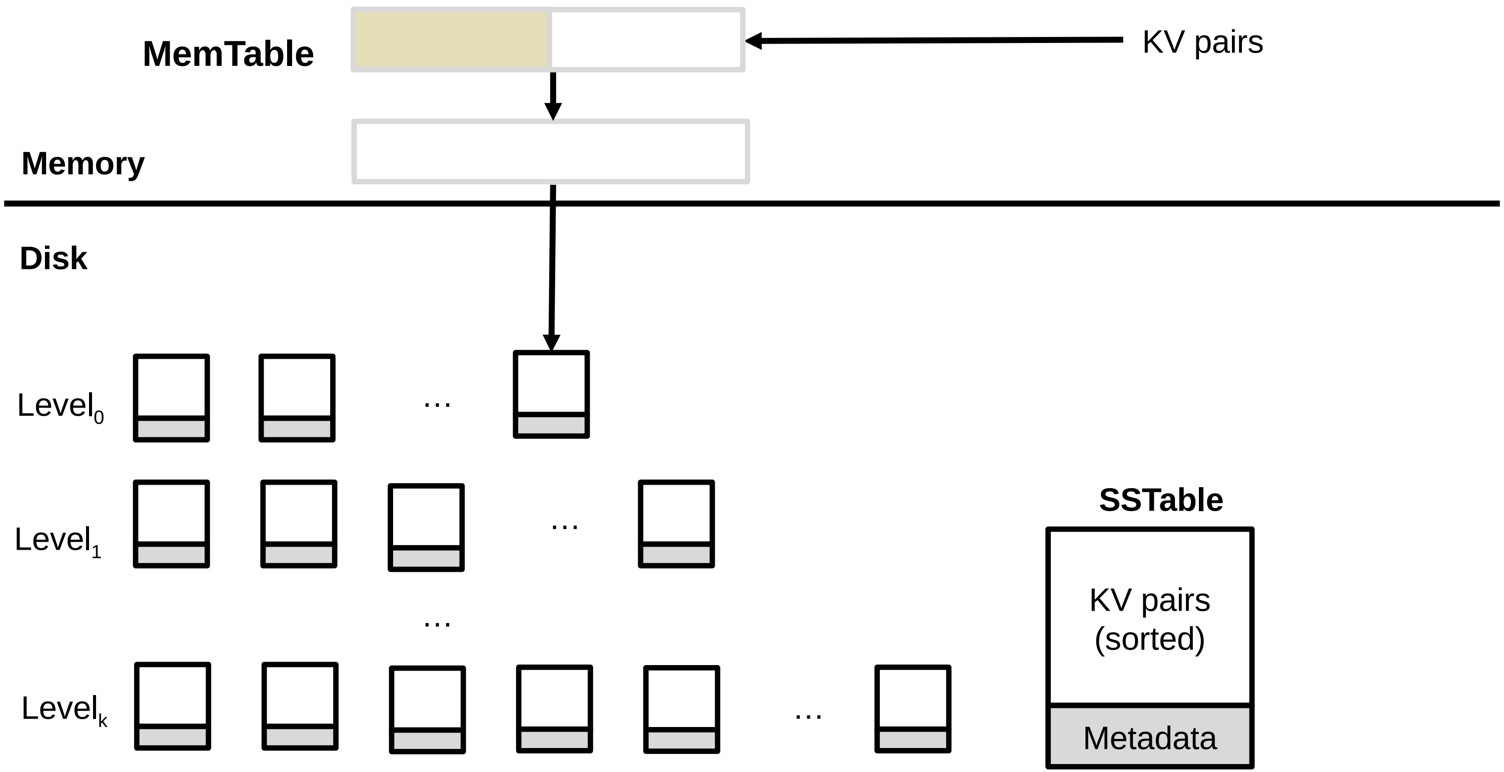
Good when

- Massive dataset
- Rapid updates/insertions
- Fast single-point lookup for recently updated data

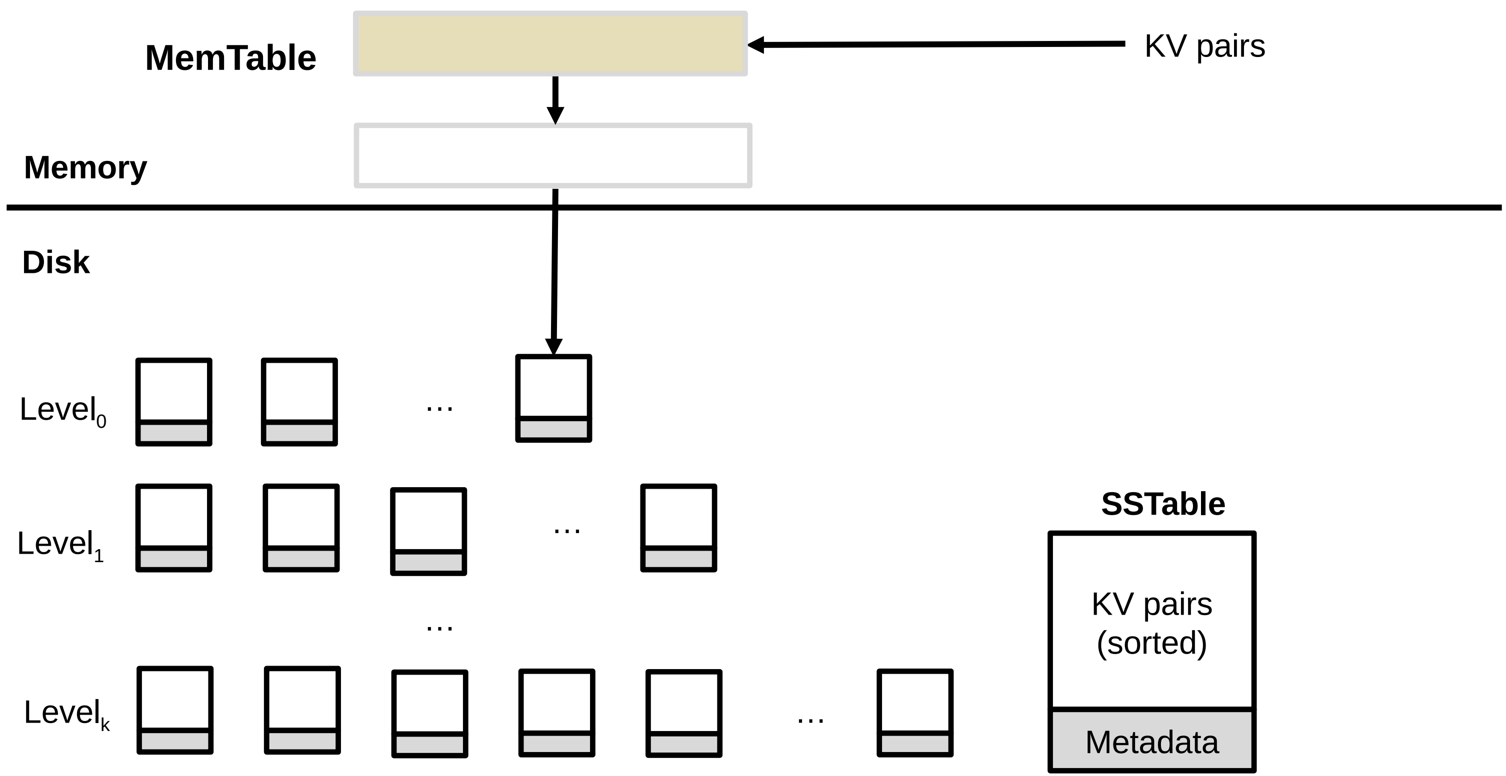
Compared with B-Tree

- **Pros**: good write performance due to sequential writes
- **Cons**: additional compaction process, possible slow range queries, **write stall** caused by the compaction, slow lookup for non-existent key

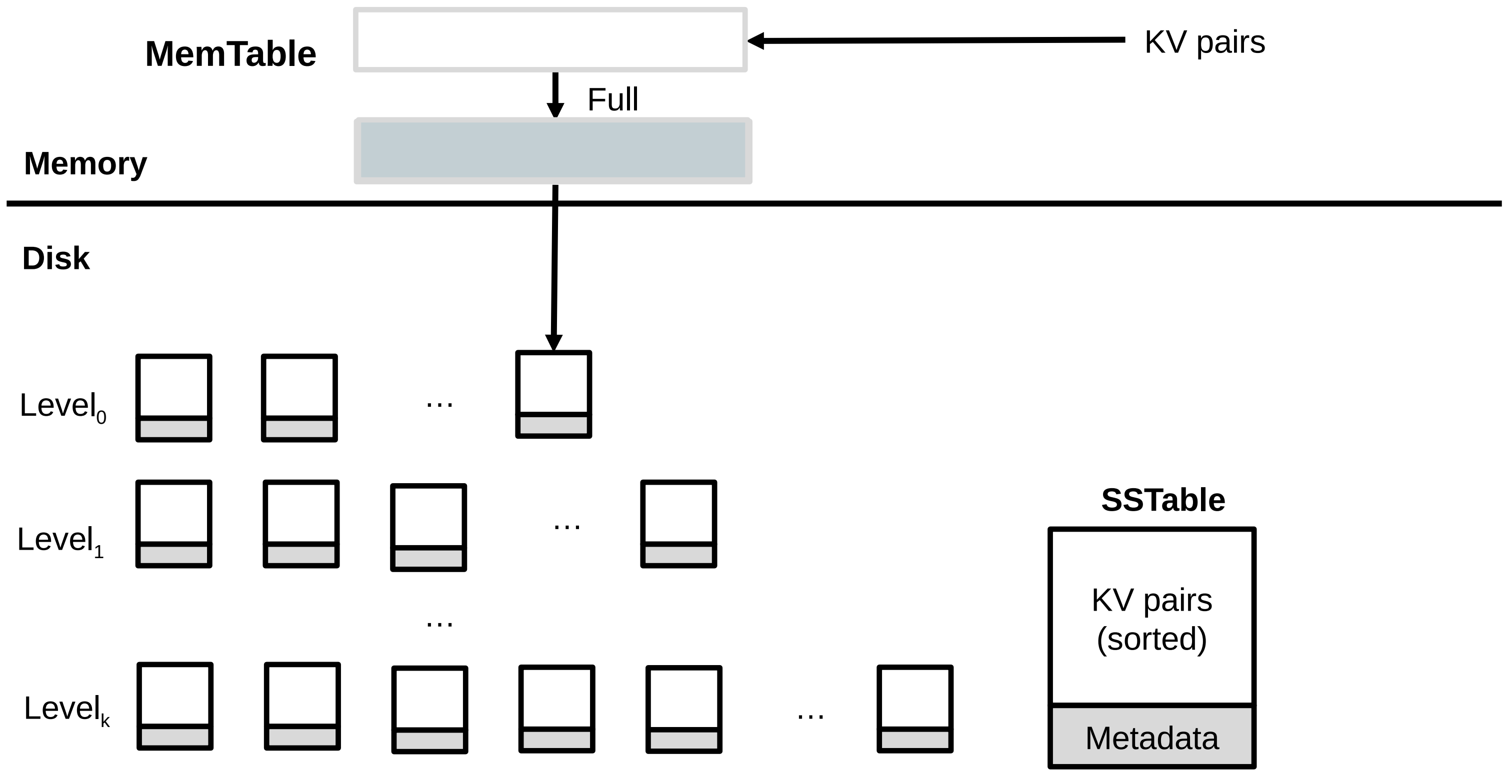
Write stall caused by compaction in LSM Tree



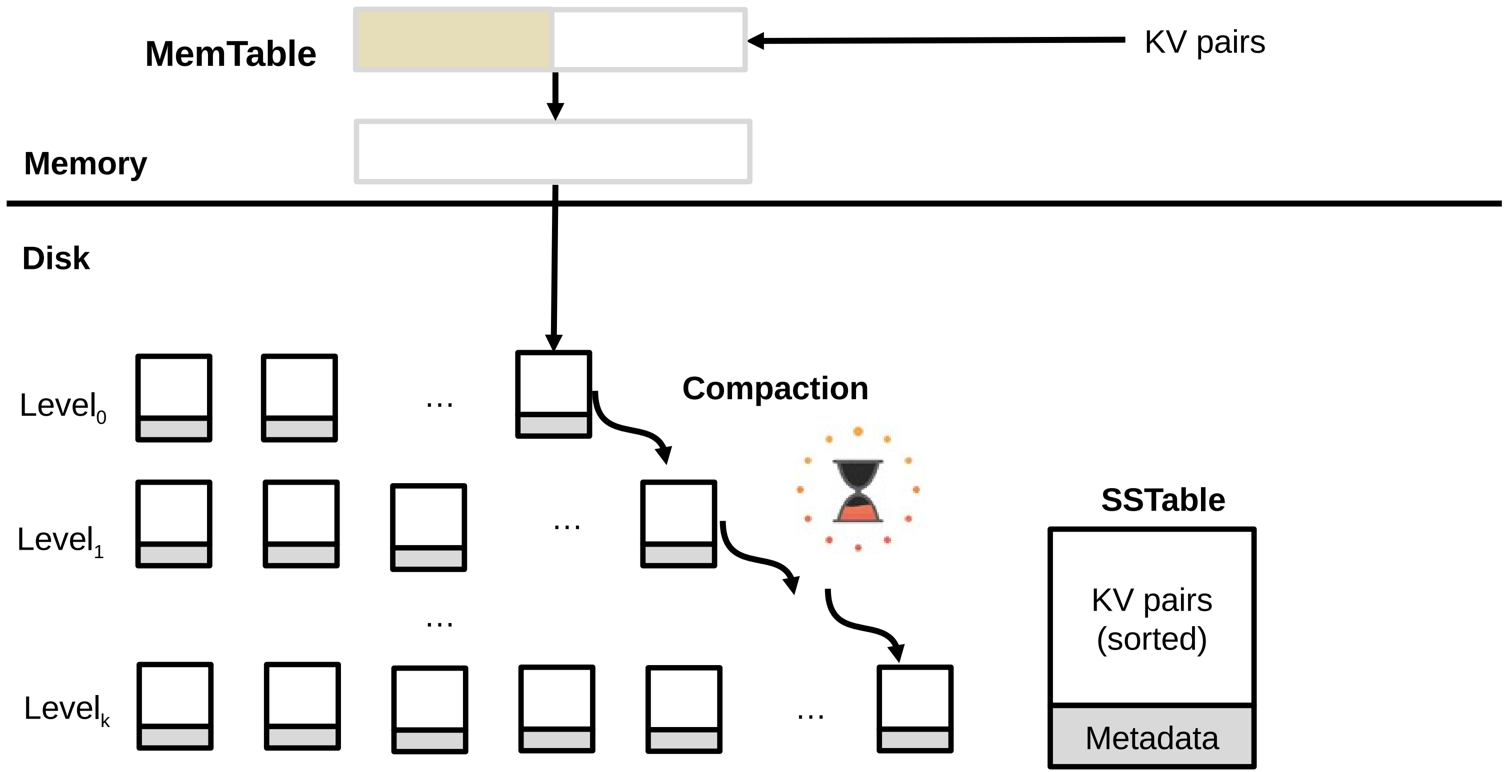
Write stall caused by compaction in LSM Tree



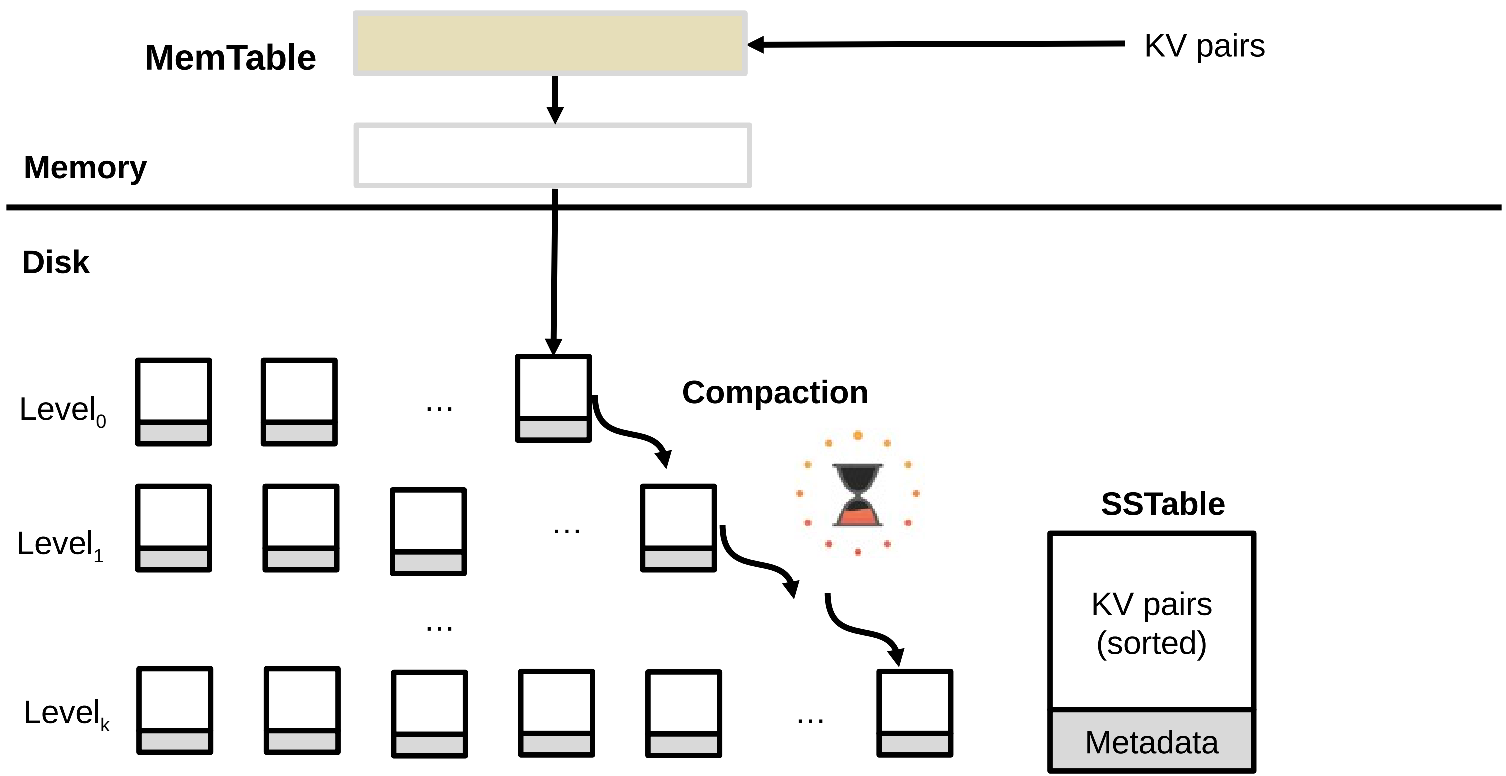
Write stall caused by compaction in LSM Tree



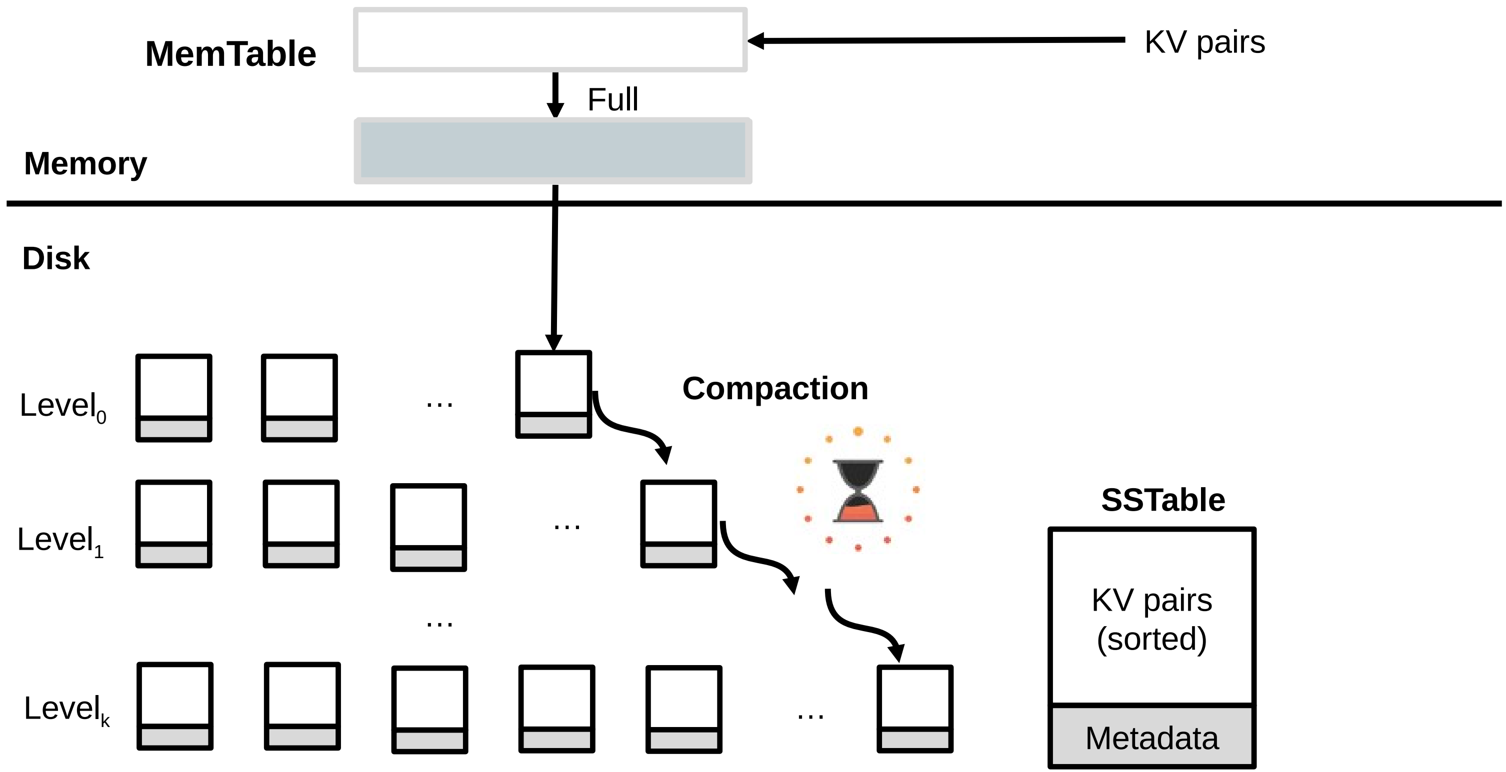
Write stall caused by compaction in LSM Tree



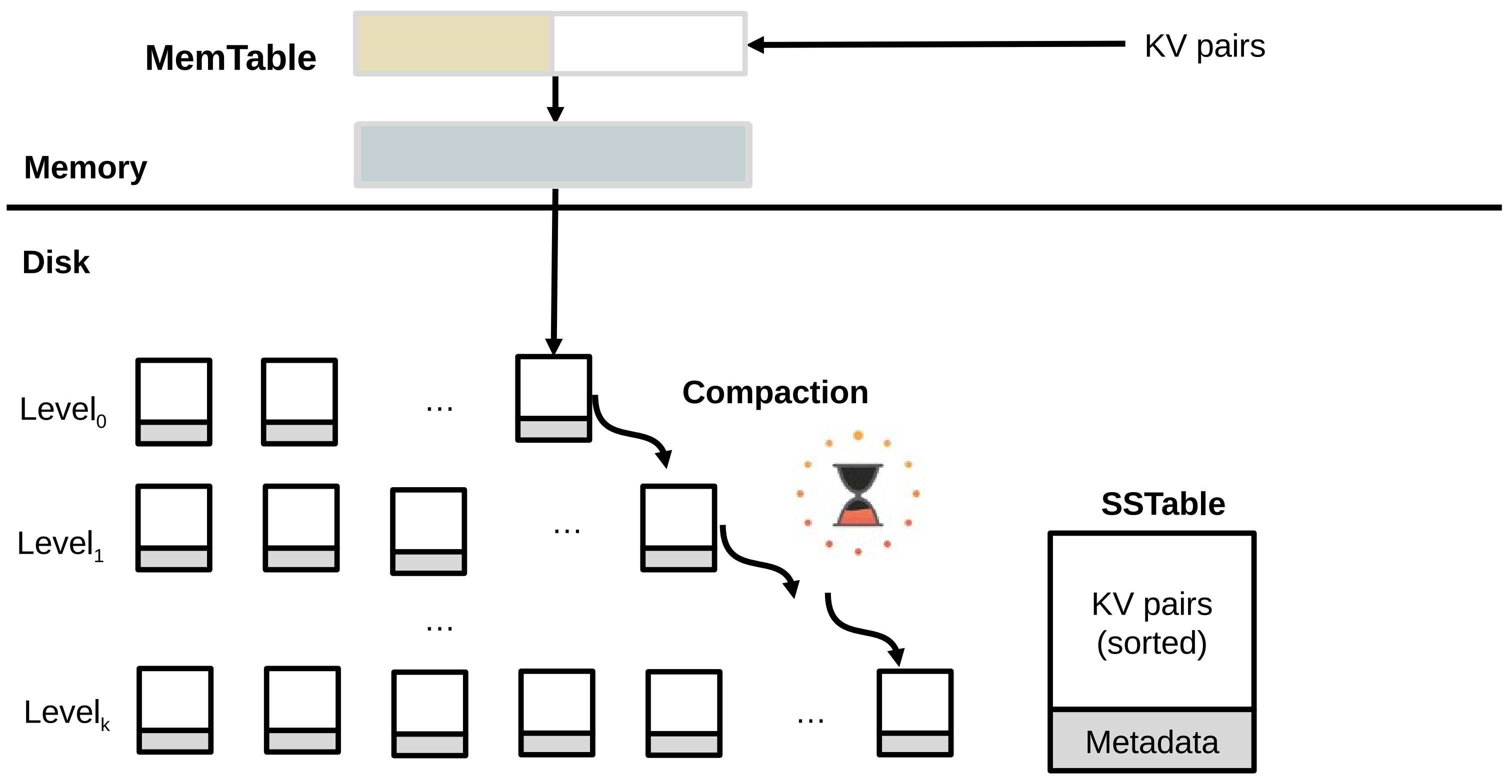
Write stall caused by compaction in LSM Tree



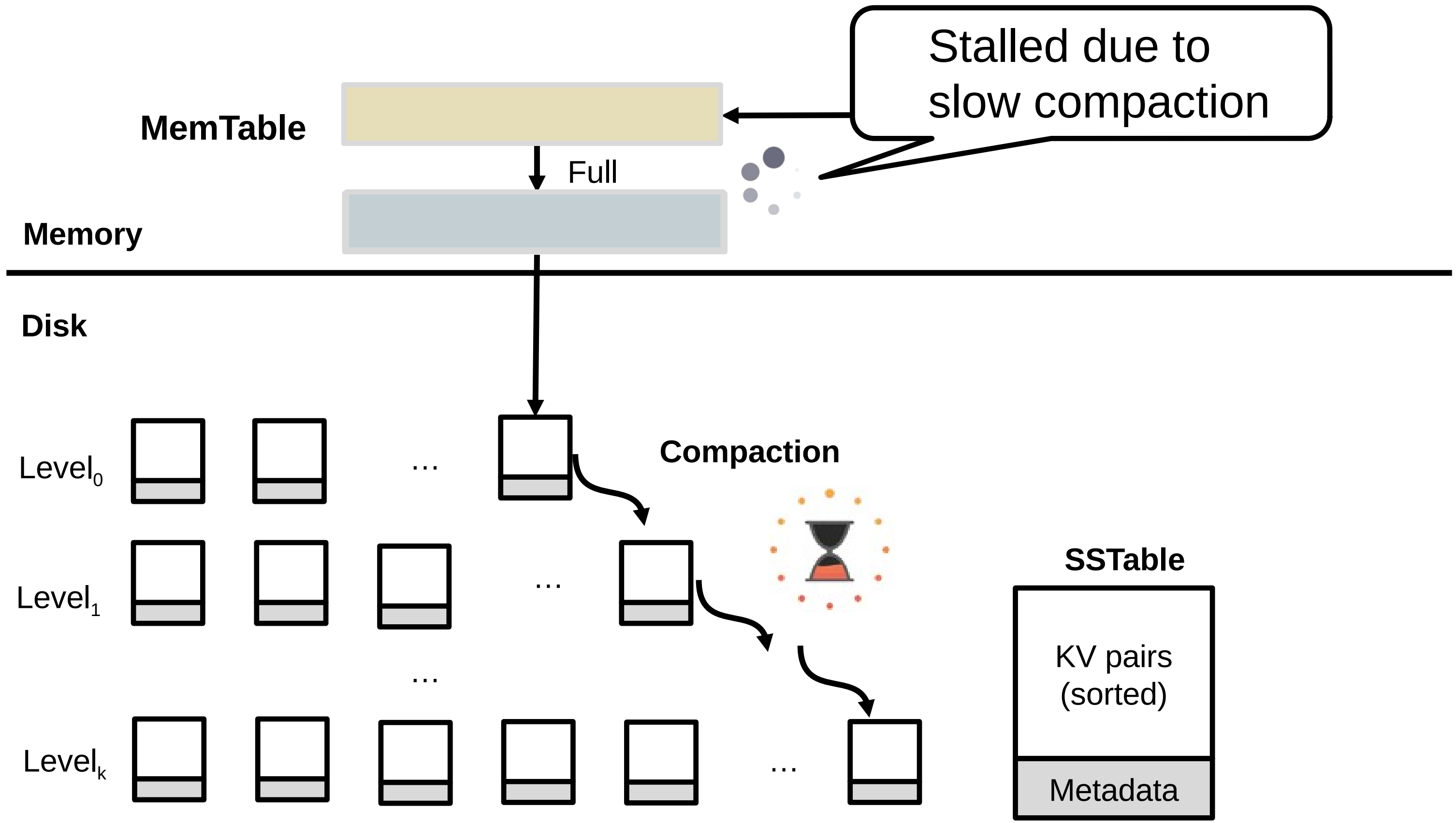
Write stall caused by compaction in LSM Tree



Write stall caused by compaction in LSM Tree



Write stall caused by compaction in LSM Tree

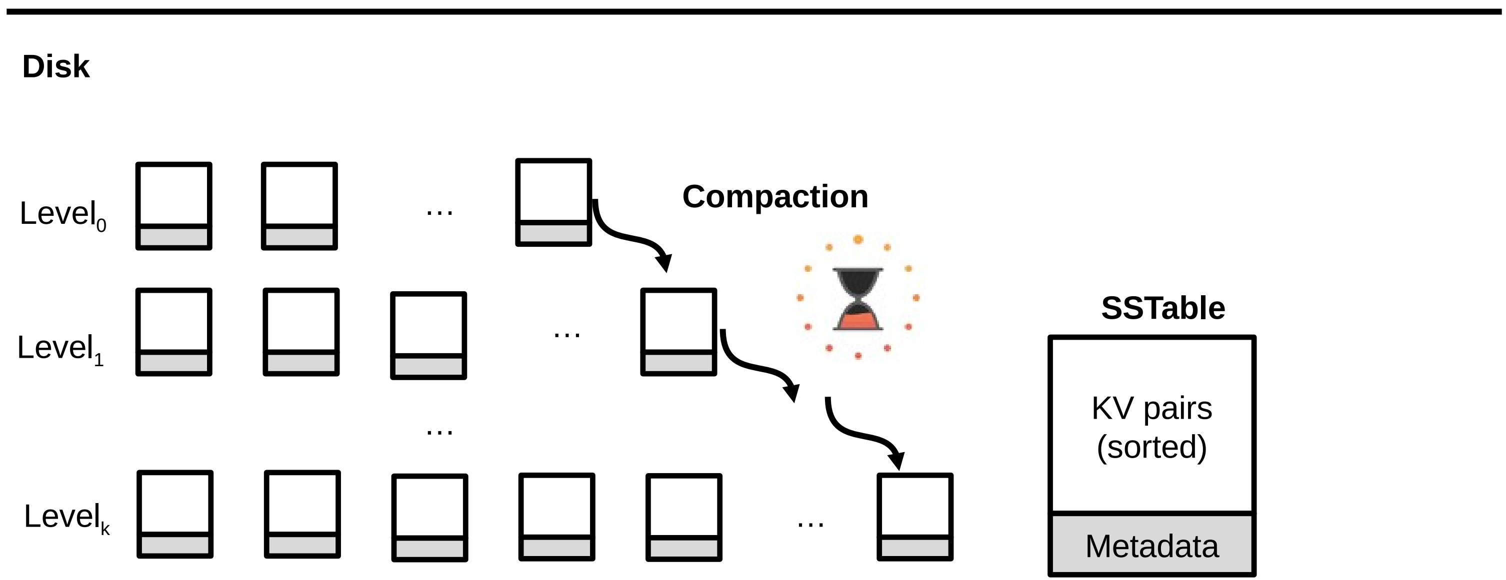


How to avoid write stall?

In principle, hard to prevent

Can only alleviate

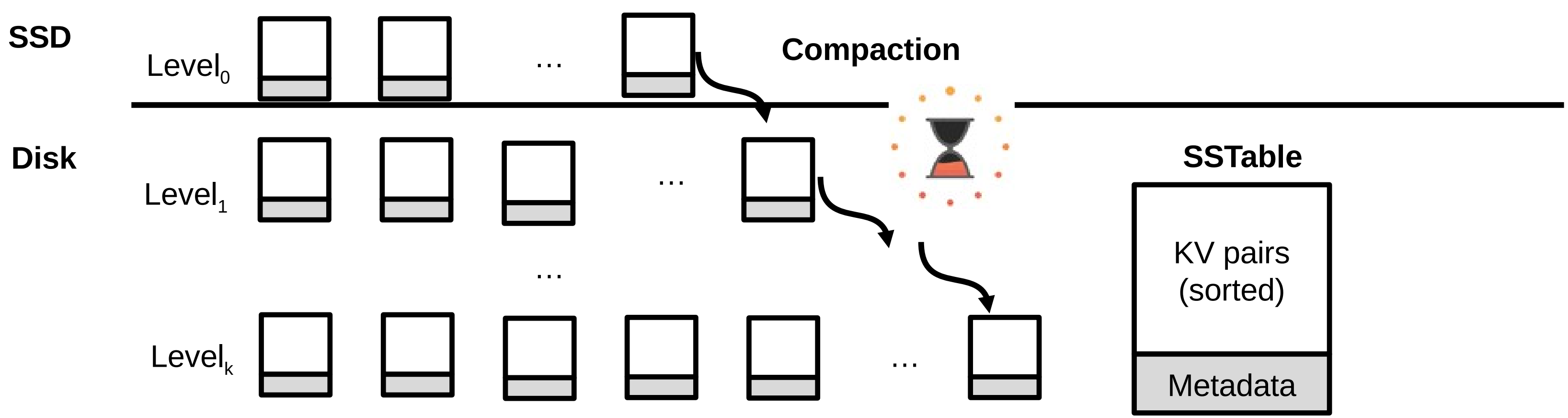
– E.g., speed up compaction & merge process with advanced hardware



In principle, hard to prevent

Can only alleviate

- E.g., speed up compaction & merge process with **advanced hardware**
- **Observation**: SSD is much faster than disk on storage
  - Using it to store up-layer SSTables



LSM Tree Summary

Good when

- Massive dataset
- Rapid updates/insertions
- Fast lookups

Compared with B-Tree

- **Pros**: good write performance due to sequential writes
- **Cons**: additional compaction process, possible slow range queries, write stall caused by the compaction, **slow lookup for non-existent key**

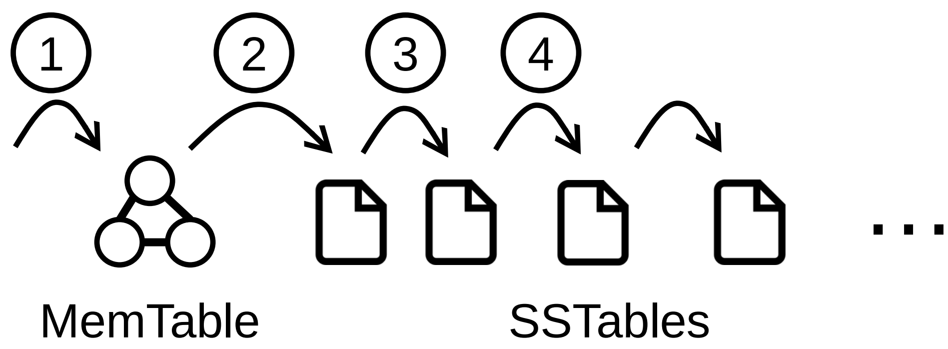
Slow lookup for non-existent key

Recall: how LSM Tree lookup keys

1. Checks the MemTable
2. If misses, checks the latest SSTable
3. If still misses, checks the next older SSTable
4. …

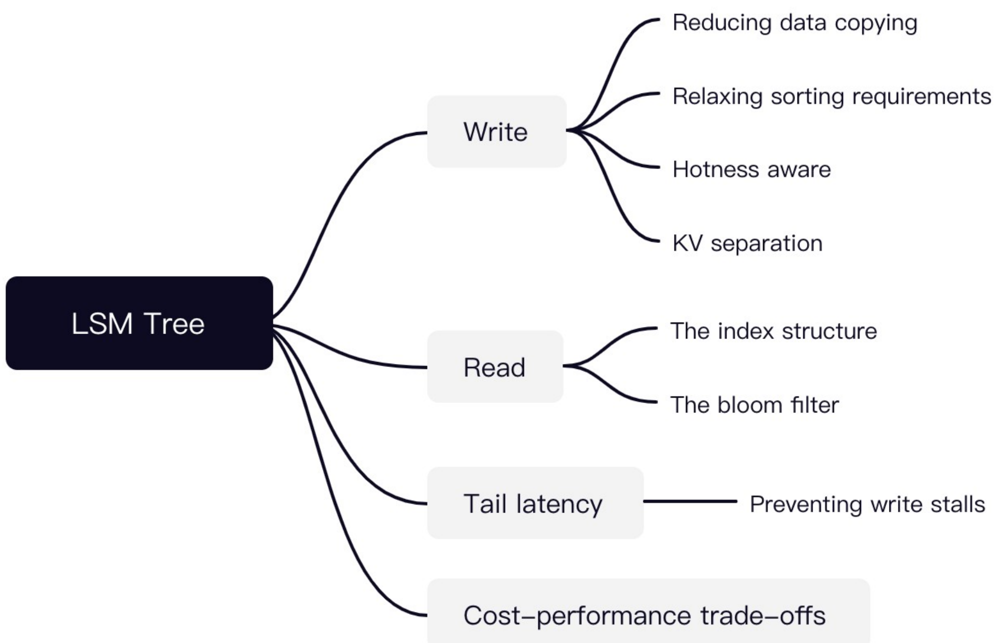
Question: what if the key non-exist?

Will lookup all the files!



LSM Tree is a hot research topic today

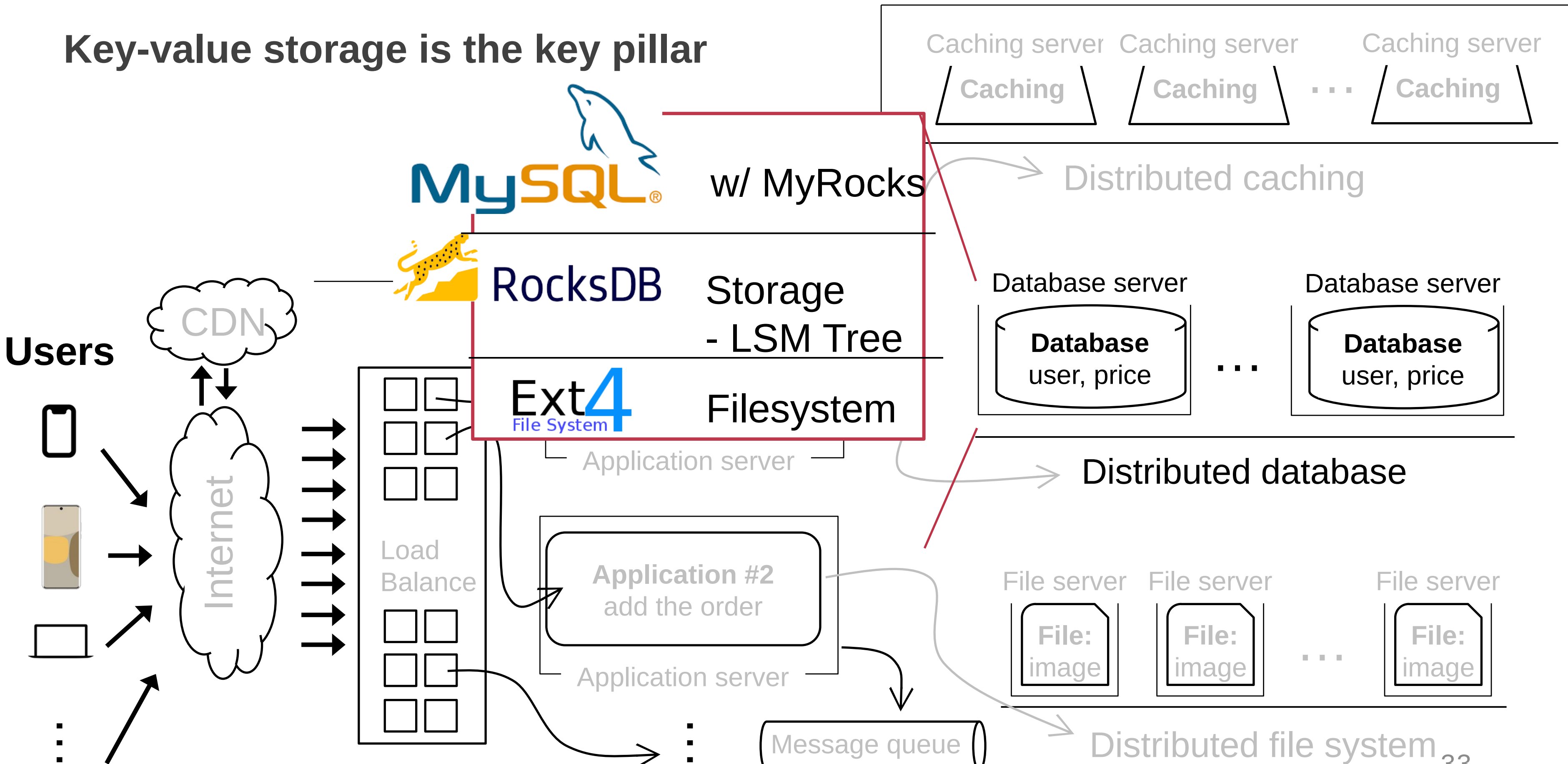
Many possible directions



Key-value storage is a key component in large-scale website

Review: large-scale websites

Key-value storage is the key pillar



Distributed key-value storage

Distributed key-value storage

Make KV store distributed (see later lectures)

- RPC + key-value storage = distributed key-value storage!
    - See the next lecture
- We can also shard the data across multiple nodes
    - i.e., high scalability

Key challenge:

- How to find the data?
    - E.g., consistent hashing

Other problems:

- Fault tolerance (see later lectures)
- Availability, replication & consistency (see later lectures)

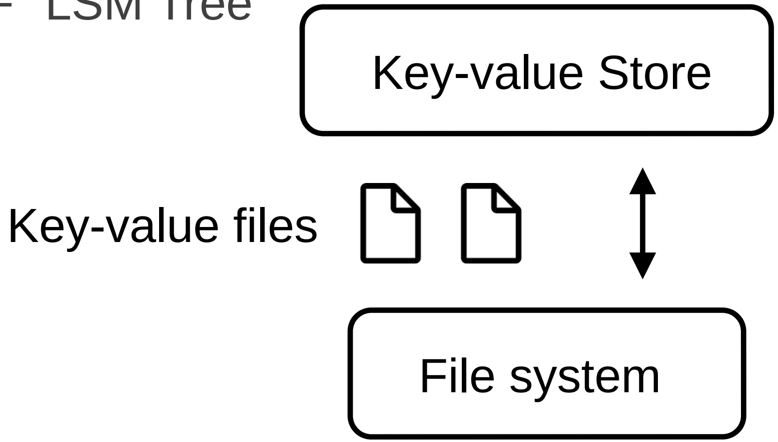
Summary of this lecture

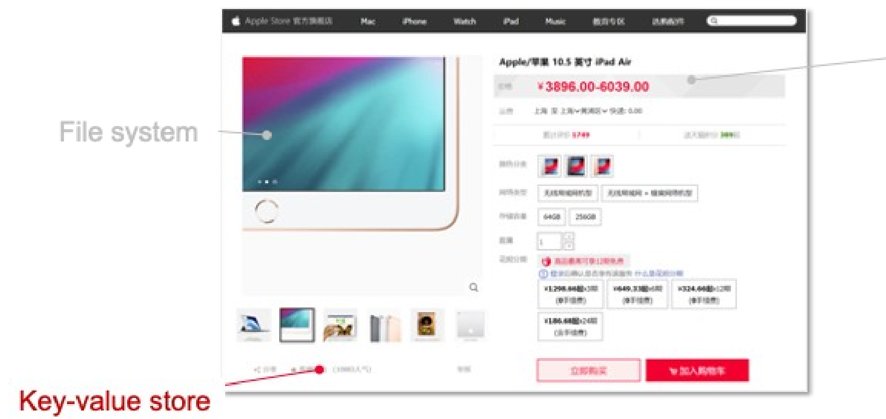
Key-value store is an important component in computer systems

- Typically built upon a filesystem to simplify disk hardware management

We show how KVS is evolved from log-structured file to the LSM tree

- Log-structured file
- Indexing
- LSM Tree





Comparison

Disk	Memory	Sorted	Range Q	Get	Put
Log for key+value	No	No	No	Extremely Slow	Fast
Log for key+value	Hash table (key-index)	No	No	Medium	Fast
Hash table key+value	Cache for Hash table	No	No	Medium	Medium
B-Tree key+value	Cache for B-Tree	Yes	Yes	Slow	Slow
SSTables key+value	MemTable key+value	Partial	Yes	Slow-Medium	Fast+

How many disk operations for each operations?

Assumption (throughout this lecture):

- The request key is selected randomly
- There is no page cache, i.e., all the requests are served from the disk
- For simplification, we directly use the key as the inode id

Get(K)✉ V

- OPEN(…) + READ(…)
- 1 random Disk read + 1 random Disk read + 1 random Disk read

Update(K,V)

- OPEN(…) + WRITE(…)
- 1 random Disk read + 1 random Disk read + 1 random Disk write

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Google

Source: Jeaf dean

Numbers everyone should know (depends on the hardware)

Random read/write ~= seek + read/write

Sequential read/write = read/write

Typically, random read/write is orders of magnitude slower than sequential

- Especially for small-sized values, e.g., 4KB

How to get the real estimation? Can do a simple profile (e.g., FIO)

Our testbed

- 4KB sequential read: READ: aggrb=**208,636KB**/s
- 4KB random read: READ: aggrb=**4,409KB**/s

Note that for disk, the minimal read/write unit is the block size

Performance estimation of the naïve KVS

Get(K)✉ V

- 1 random Disk read + 1 random Disk read + 1 random Disk read

Update(K,V)

- 1 random Disk read + 1 random Disk read + 1 random Disk write

Assumption

- Each key and value are both 16B

Suppose random read perf ~= random write perf, block sz = 512B

- Get = Update ~= 2.9K reqs/s
- 4,409KB / (512B * 3)

Review: A naïve Key-Value Storage (KVS) w/ log file

Suppose we store all the KV **in a single file** in the following format:

Key Value

```
123456 , '{"name":"London","attractions":["Big Ben","London
Eye"]}'
42 , '{"name":"San Francisco","attractions":["Golden Gate
Bridge"]}'
...
```

Update:

– One sequential WRITE

Get

– One random READ (with in-memory index)

Performance estimation of the naïve KVS + Log file

Assumption (can be adjusted given different workload patterns)

- (K,V)s are padded to 512B (smaller size makes it slower)

Update

- 407 K reqs/s // disk sequential write bandwidth
- 208,636KB / 512B

Get

- 8.8 K reqs/s (in the optimal case, i.e., all index are stored in the memory)
- 4,409KB / 512B

What if no in-memory index is given?

- Too slow to be estimated …

Review: B+Tree to store the (K,V)s

A B-tree is a tree-like data structure

- Each node is **fixed-sized**, can store multiple keys, and keys are **sorted**
- Support **efficient range** operations (e.g., Scan)
- Optimized for **large** read/write blocks of data

Many variants exist, a standard choice is B+Tree

- All the leaf nodes of the B-tree must be at the same level
    - Simper to link leave nodes to support range queries

Performance estimation of B+Tree-based KVS

From a high-level:

– Get ~= update ~= random disk access

But, the tree height depends on the setup of the B+Tree

– E.g., how many (K,V)s are stored, the size of each (K,V), the configuration of each tree node

What are the common (K,V) sizes?

– 16B keys are common [1], and small values (e.g., 64B) dominates

What are the common node size?

– Match the disk block size (e.g., 512B)

[1] Workload Analysis of a Large-Scale Key-Value Store, from Facebook

Performance estimation of B+Tree-based KVS

Setup

- Keys are 16B, node size 512B, 1 million key-value pairs
- This means up to 32 keys per node

Tree height ~= 7 (6.8) in the worst case [1]

- $1+ \log_{32}((n+1)/2)$

So Get = Update = 1.1K reqs/s

- Even slower than our naïve KVS
- Though it supports ordered accesses and range queries

[1] Workload Analysis of a Large-Scale Key-Value Store

Review: LSM Trees

Store (K,V)s in SSTables

– Segmented log file + sorted (K,V)s

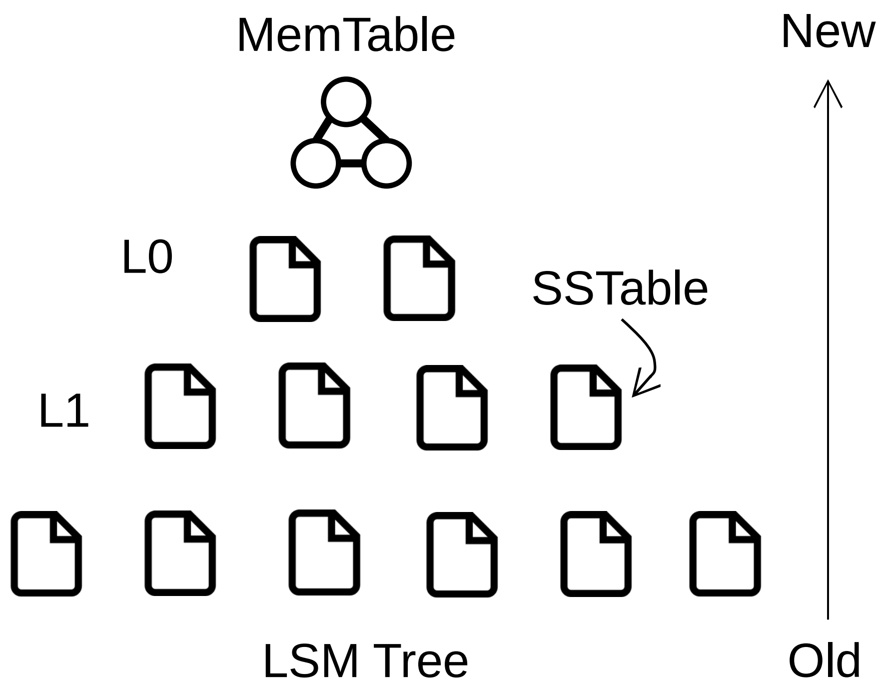
MemTable to simplify building SSTable

– Also as a cache to absorb recent updates for sequential disk writes

SSTable's are organized as layers

– To accelerate old key lookups & range queries

– Each layer has an (in-memory) sparse index to accelerate the lookup (O(1) lookup per layer)



Performance estimation of the LSM (Much harder)

Update (not considering compaction & merge)

- MemTable not full: DRAM write bandwidth (>> disk sequential write)
- MemTable full: disk sequential write

Not slower than the naïve log file

Get is even harder. We can only measure #disk accesses for different Get scenarios (Trade READ for WRITE)

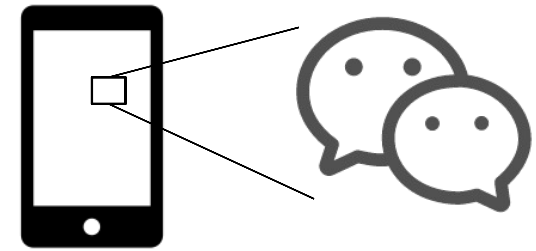
- Optimal case: 0 (in the MemTable)
- Worst case: number of layers random disk accesses

The number of levels can vary but is often between 4 and 7 in practice.

- Typically the number is small, because each SSTable is large (>> than block size)

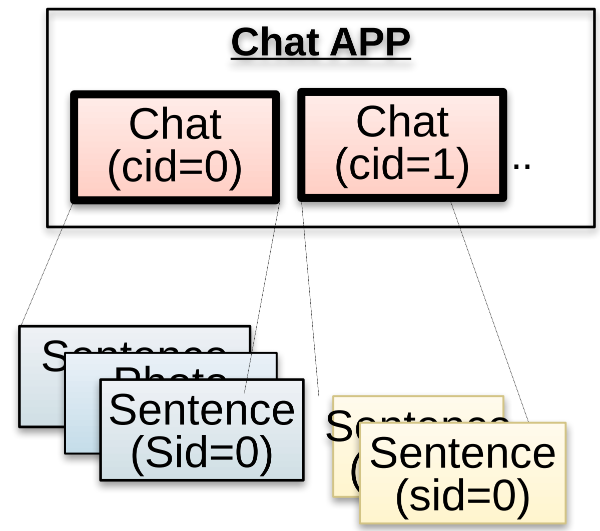
Building applications on KVS

Motivating application: Chat APP



Abstracted data type

– Chat: a chat to someone (or a group)
– Sentence: the detailed sentence from a conversion
– Operation: add a sentence to a chat



Typical system to support the APP: key-value storage

– E.g., each sentence can be uniquely identified by <sid, "some sentence”>
– Each chat can be uniquely identified by <cid, [a list of sids]>

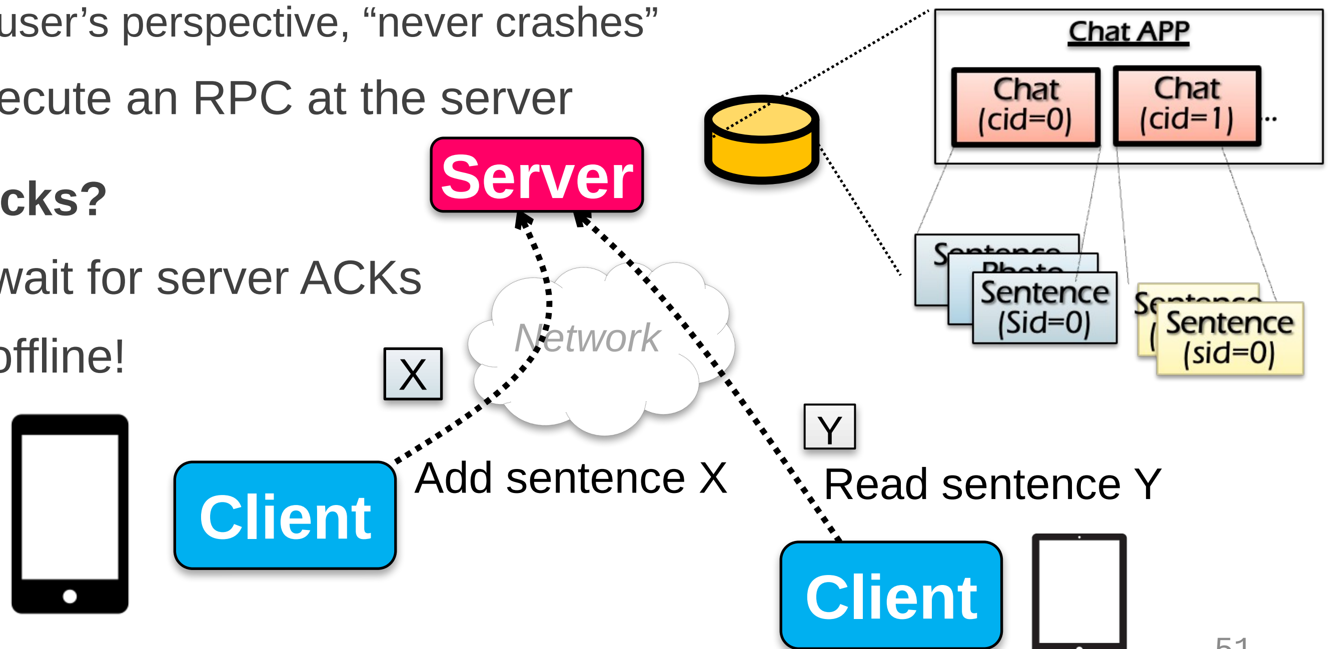
How is the key-value storage (KVS) deployed?

Approach #1: store the KVS at a centralized server

- The server resides in a datacenter (e.g., 贵州)
- The server can be made highly reliable (see later lectures)
  - e.g., on the user's perspective, "never crashes"
- All **operations**: execute an RPC at the server

What are the drawbacks?

- Inefficiency! Must wait for server ACKs
- Cannot work with offline!



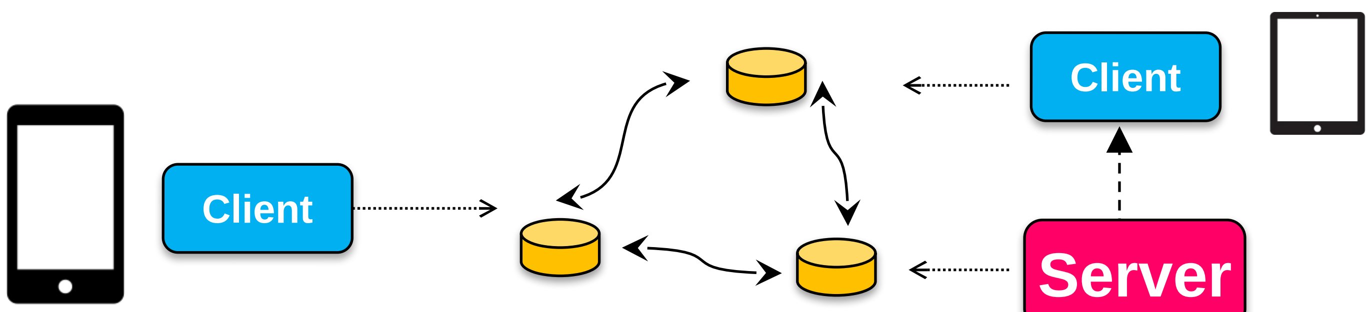
How is the key-value storage (KVS) deployed?

Approach #2: store the KVS at a centralized server + at each device

- Default implementation of many ChatAPP, e.g., WeChat
- Question: how to do the updates? We need to sync with other devices!

Naïve solution

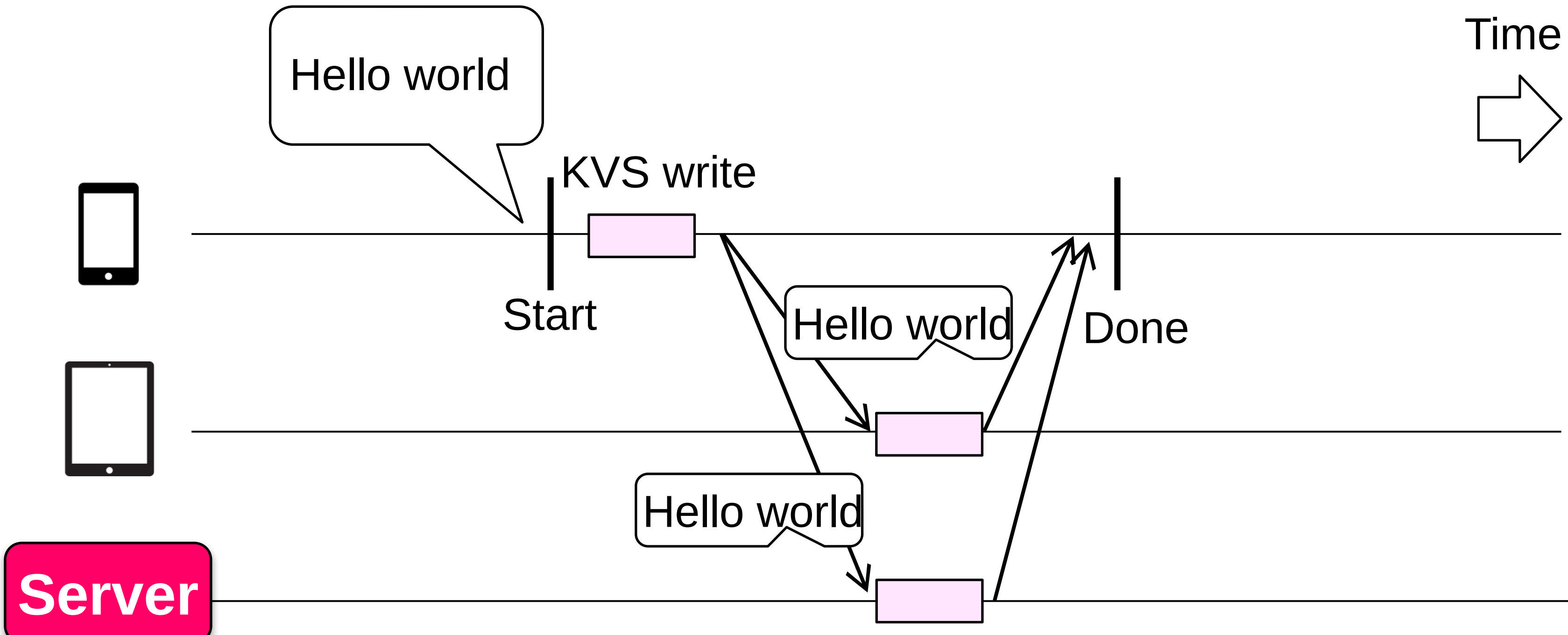
- Read: return the latest copy on the local KVS
- Write: update the local KVS, sync with other KVS, then return to client



Naïve solution: wait sync for each updates

Read: return the latest copy on the local KVS

Write: update the local KVS, sync with other KVS, then return to client



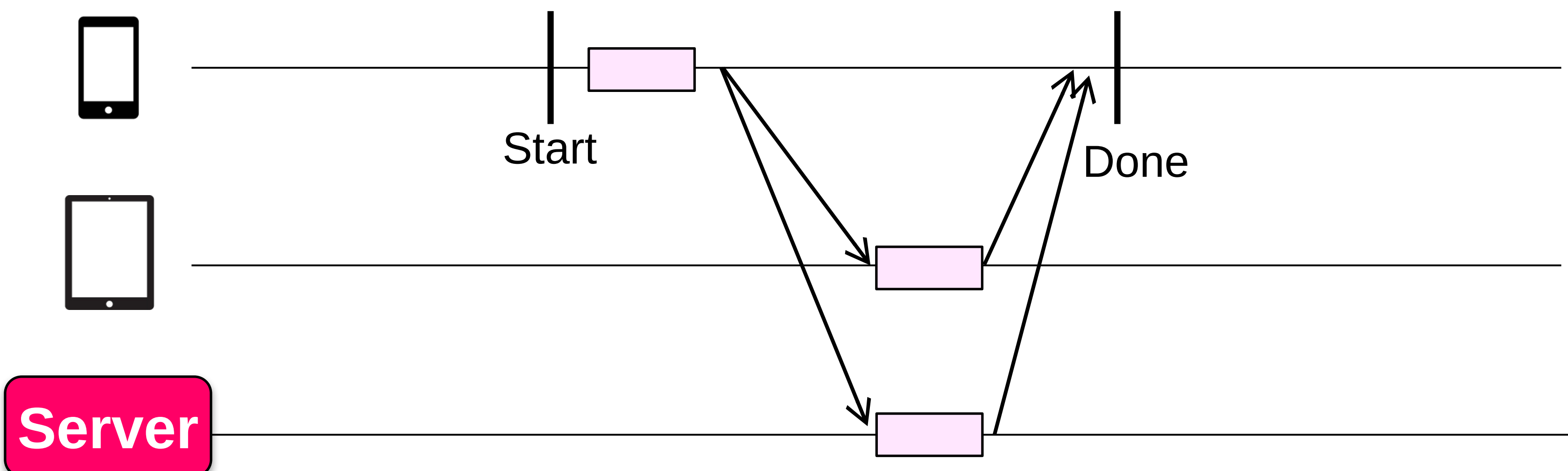
Naïve solution: wait sync for each updates

Problem#1: inefficiency

– Each write must wait for the sync to be done, which may be lengthy

– Numbers every programmer is suggested to know

  - RTT among devices: **100 – 400ms**

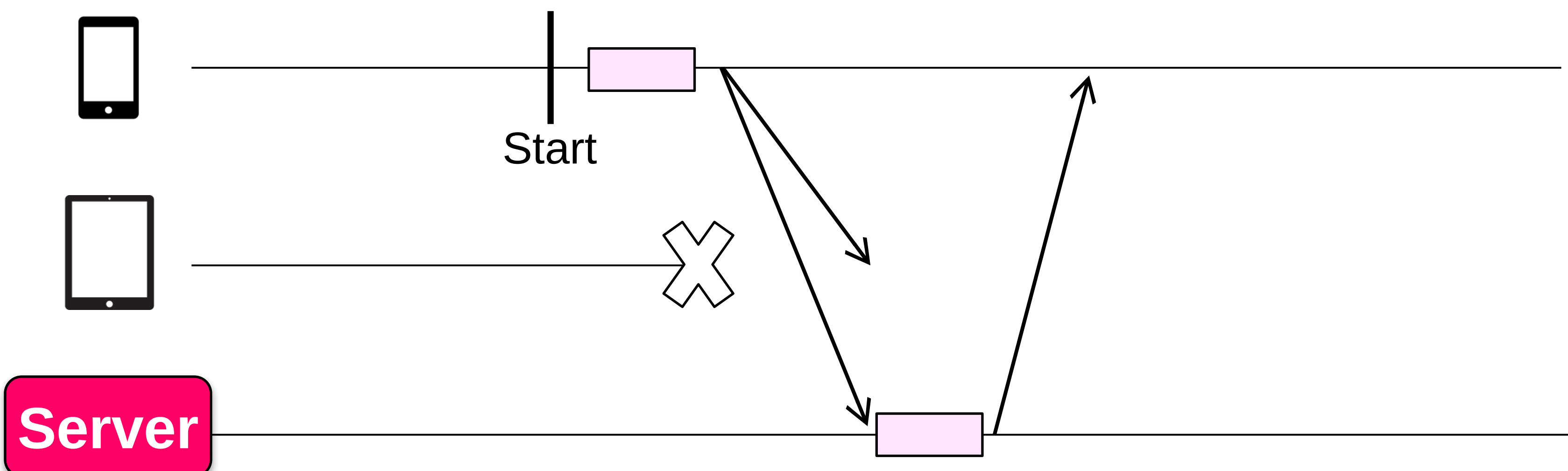


Naïve solution: wait sync for each updates

Problem#2: cannot tolerate network connectivity

- Periodic connectivity to net and other nodes (e.g., lost WIFI)
- Common under the setup of chat APP (e.g., I have entered the subway 🔇)

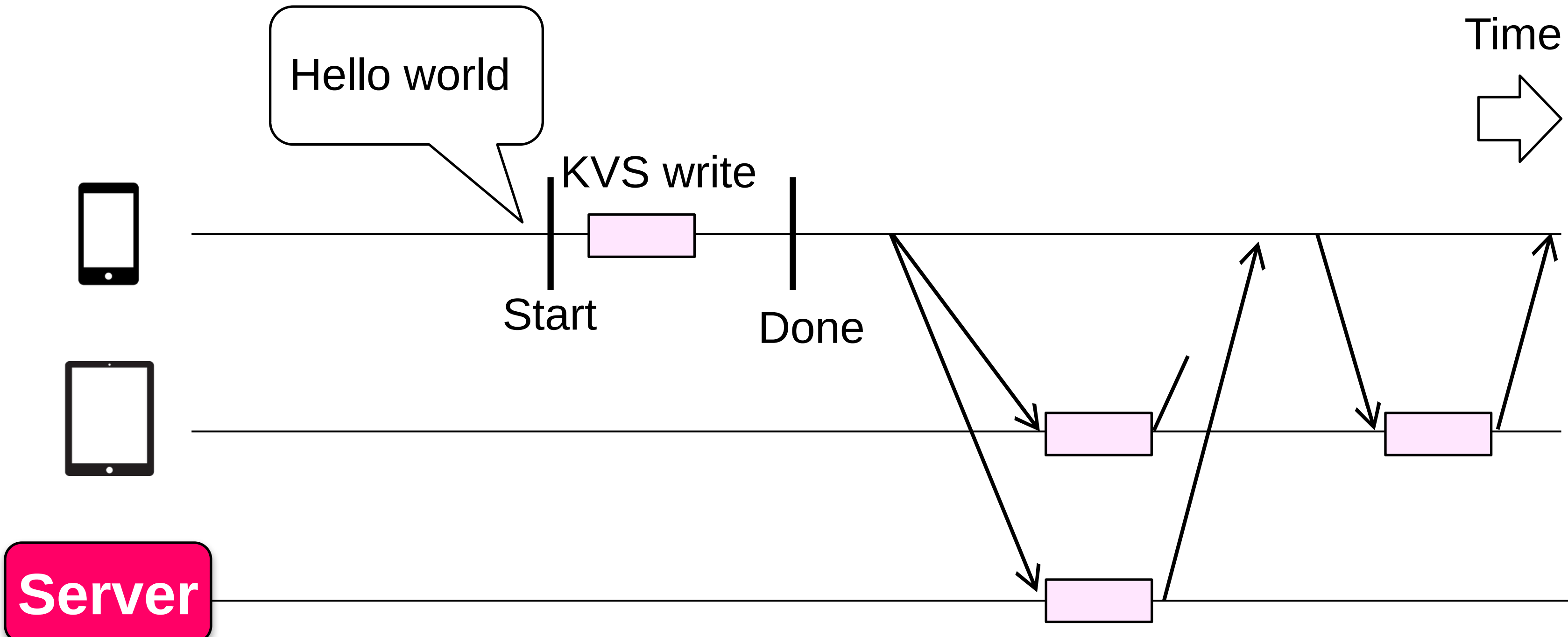
Under network disconnection, the sender will be blocked



Naïve solution++: sync but not wait

Read: return the latest copy on the local KVS

Write: update the local KVS, sync with the others in background & return



Question: what can go wrong?

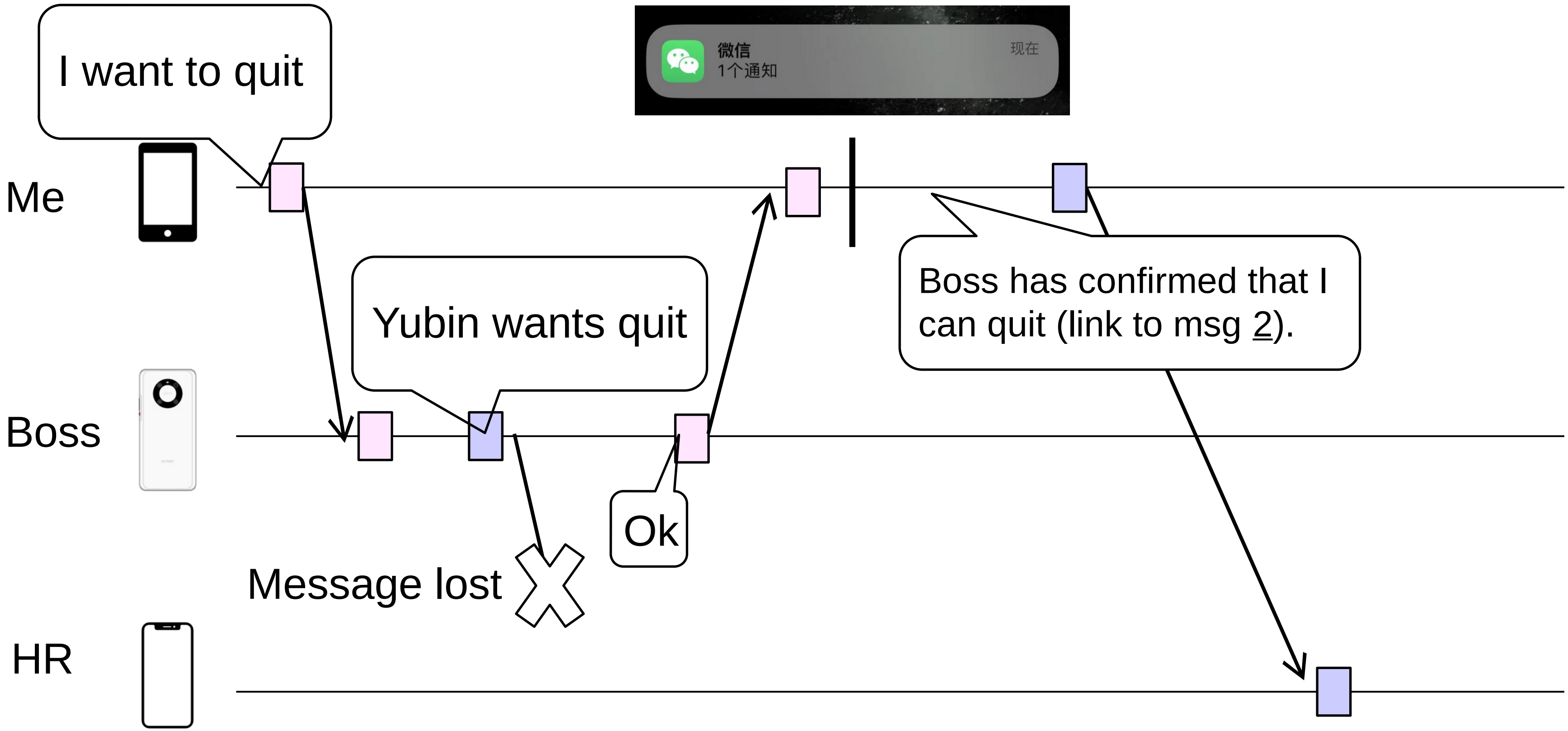
Considering the following scenario:

1. Me -> Boss: boss, I want to quit
2. Boss -> HR: Yubin (me) wants to quit
3. Boss -> Me: OK
4. Me -> HR: Boss has confirmed that I can quit (link to msg 2).

The HR's job:

- If sees "Boss has confirmed that I can quit" from Yubin, check whether "Yubin wants to quit" has been sent by the boss

Question: what can go wrong?



Question: what can go wrong? (Abstracted version)

Example scenario

1. Me -> Boss: boss, I want to quit
2. Boss -> HR: Yubin wants to quit **(x)**
3. Boss -> Me: OK
4. Me -> HR: Boss has confirmed that I can quit (link to msg x). **(y)**

The HR's job:

– If sees "Boss has confirmed that I can quit. " (**y's update**), check whether "Yubin wants to quit" has been sent (**x must have been updated**)

Question: what can go wrong? (Abstracted version)

We have two data, X and Y (initialized as 0)

- Process #1: Put (X,1), Put (Y,1)
- Process #2: If sees Y = 1, Print (X) // must be 1

Unexpected behavior

- Process #2 sees Y = 1, But not X = 1

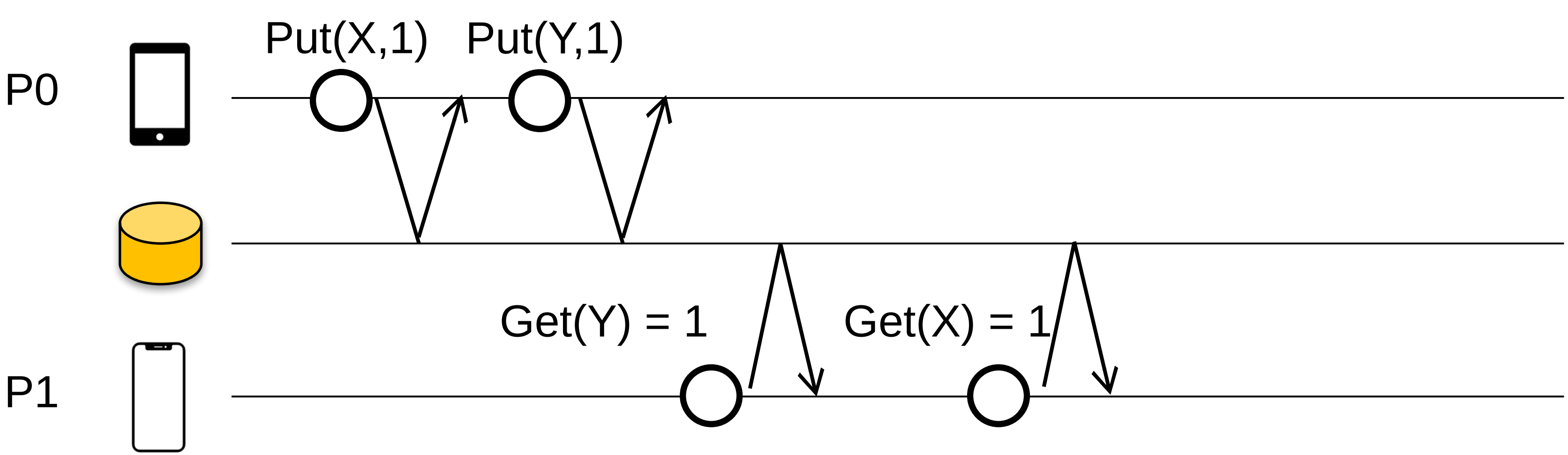
Question: what can go wrong? (Abstracted version)

Unexpected behavior

– Process sees Y = 1, But not X = 1

Questions

– Can the unexpected behavior happen in approach #1 (a single centralized KVS and RPC for all the operation implementation)?



Question: what can go wrong? (Abstracted version)

Unexpected behavior

– Process sees Y = 1, But not X = 1

Questions

– Can the unexpected behavior happen in approach #1 (a single centralized KVS)?

– Can the unexpected behavior happen in approach #2’s naïve (sync for each update)?

Naïve++ is efficient, but have unexpected behavior

An trade-off. The unexpected behavior is usually called inconsistency

How can we write correct distributed programs?

– The developer must cope with inconsistency issues!

How to cope with inconsistency?

– The system must provide **a consistency model** when operating the distributed data

What is consistency model?

> **Consistency model** defines **rules** for the apparent order and visibility of updates, and it is a continuum with **tradeoffs**.
>
> \- Todd Lipcon

Single object consistency is also called "coherence"

Examples

- Local shared virtual memory — W(y) 1 — R(y) (should be 1)

Time

- Database — X <- X + 1 ; Y <- Y - 1 — Assert X+Y unchanged

Consistency across multiple objects
e.g., all-or-nothing + before-or-after

Consistency Challenges

No Right or Wrong consistency models

- Tradeoffs between **ease** of programmability & **performance**

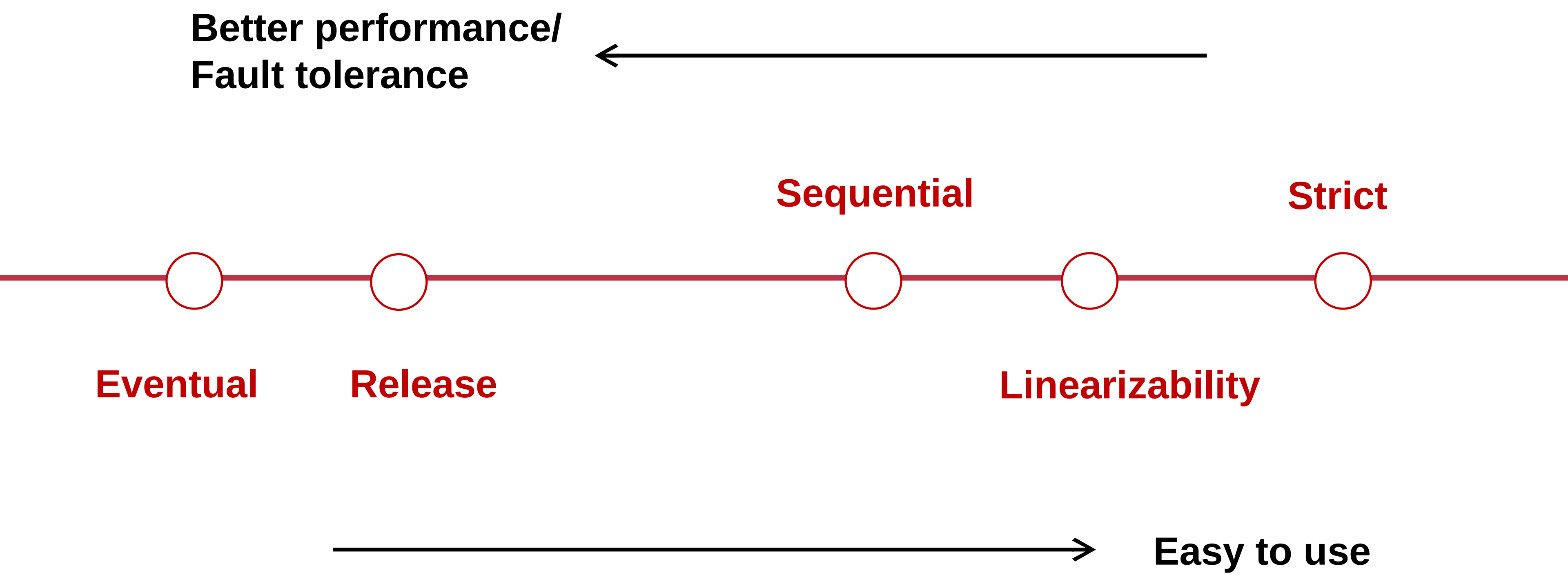
Why programmability?

- Unexpected behavior usually needs to be fixed by the developers ⏪

Consistency is hard in (distributed) systems

- Data replication (& caching)
- Concurrency (multi-core & multi-server)
- Failures (e.g., machine or network)

Spectrum of Consistency Models



Note that many other models exists

What is the desired model for the developers/users?

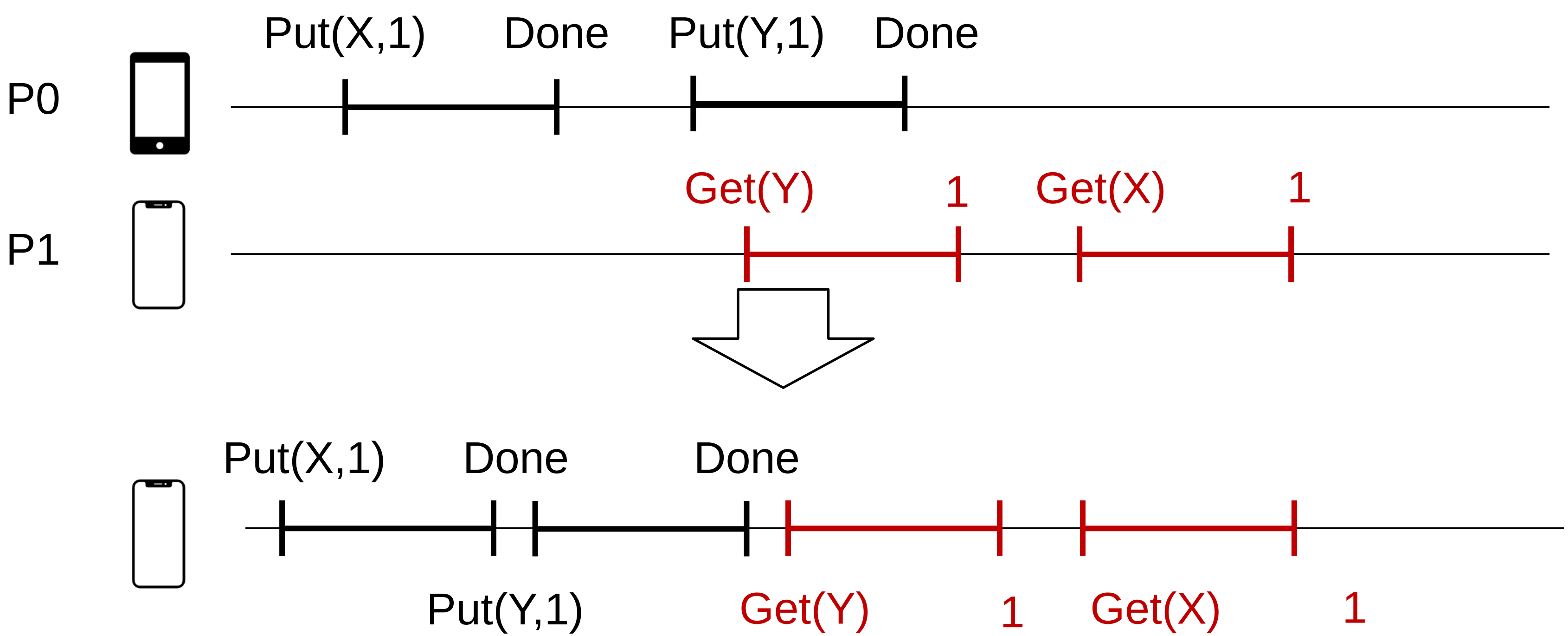
It's easy for users to reason about correctness assuming

- Everything has only one-copy
- The overall behavior is equivalent to **some serial behavior**

Example: equivalence to some serial execution

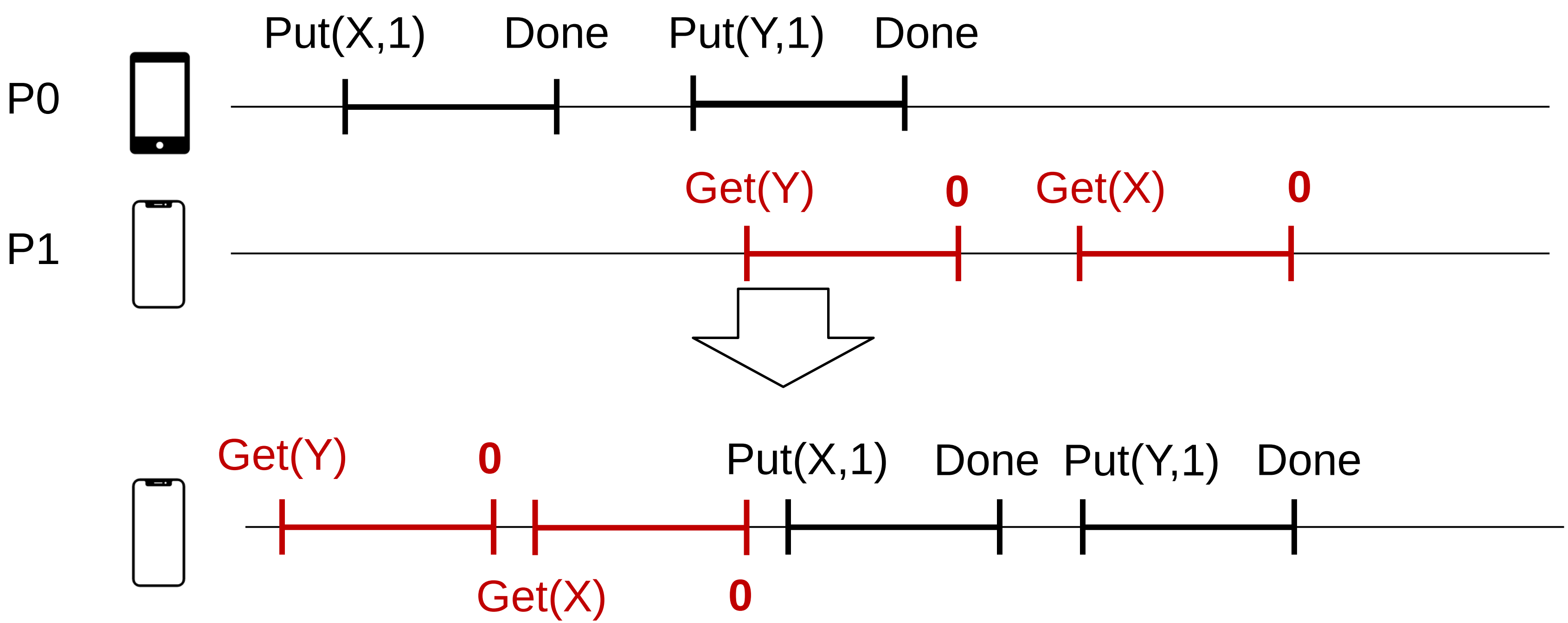
Though being concurrent, we can map it to some serial order

- E.g., one device, execute the chat one by one (as a **atomic** unit)



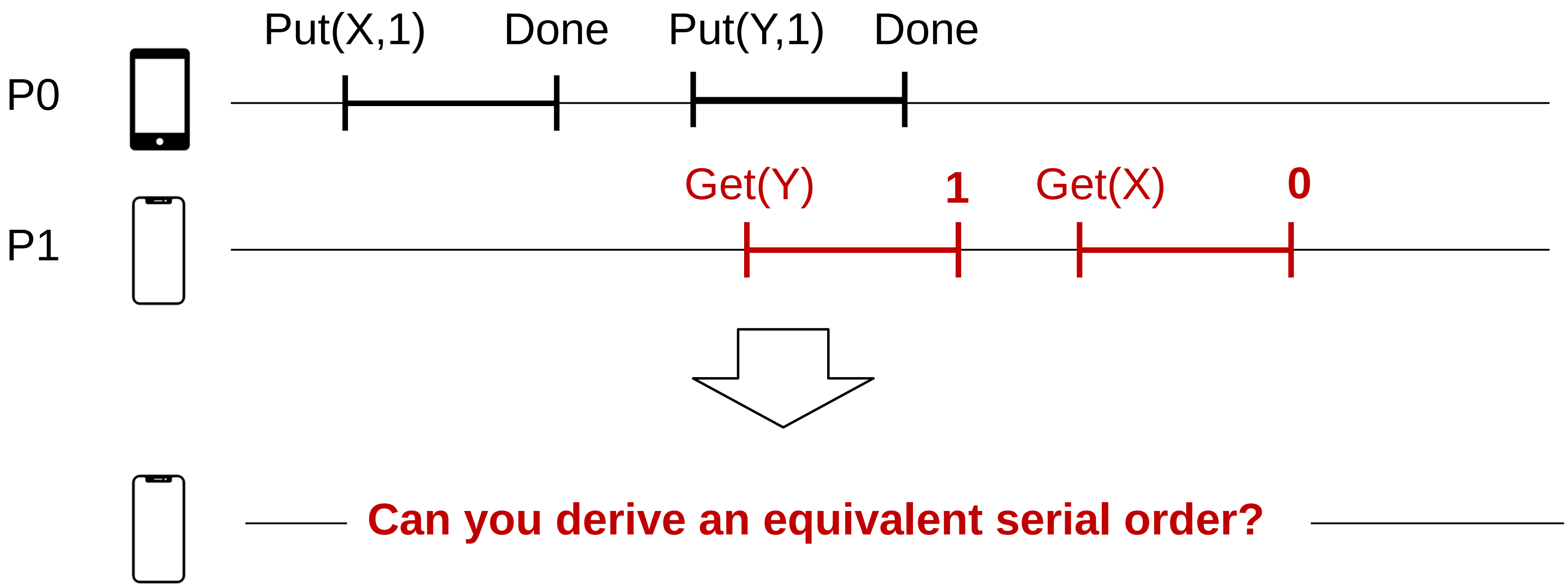
Can this concurrent exe. equivalent to some serial exe.?

Though being concurrent, it is easier to understand if we map it to some serial execution



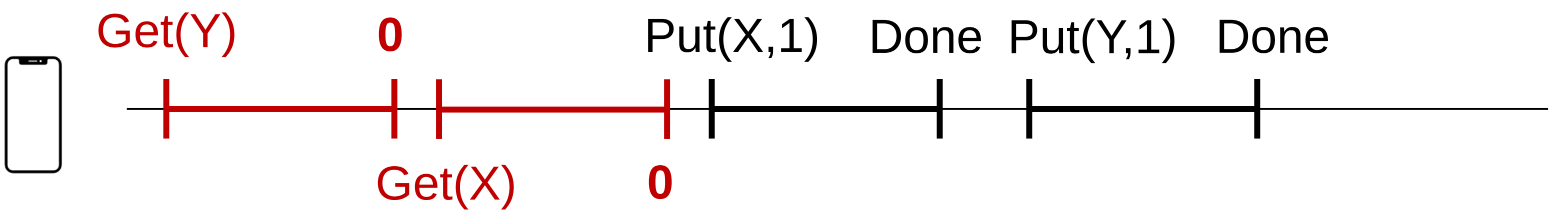
Can this concurrent exe. equivalent to some serial exe.?

Though being concurrent, it is easier to understand if we map it to some serial execution

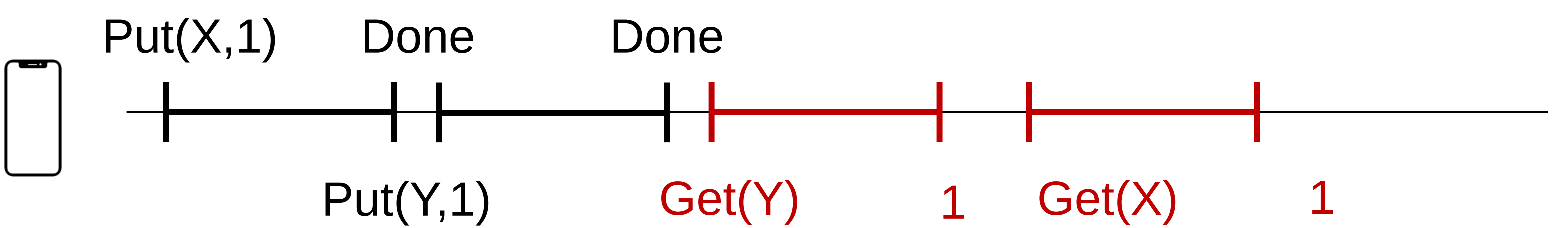


Proof: we cannot find an equivalent serial execution

Case #1.

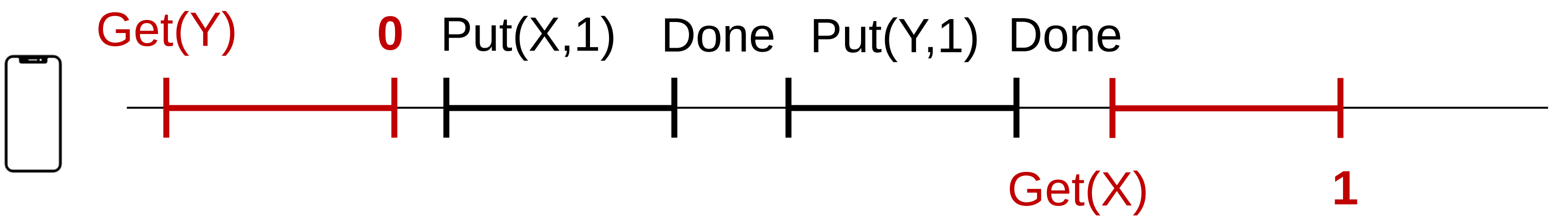


Case #2.

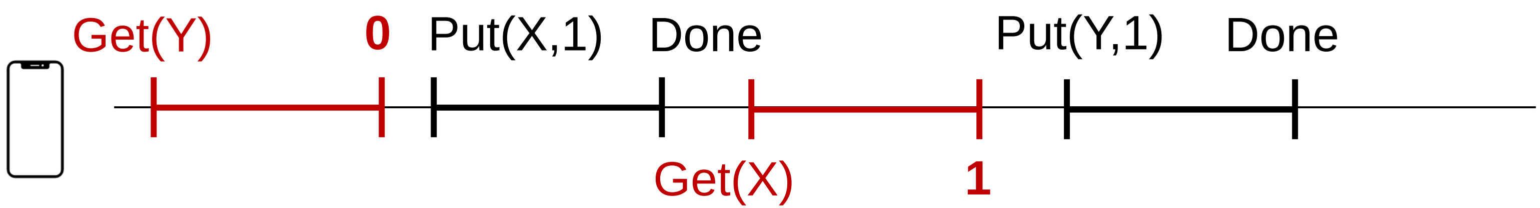


Proof: we cannot find an equivalent serial execution

Case #3.

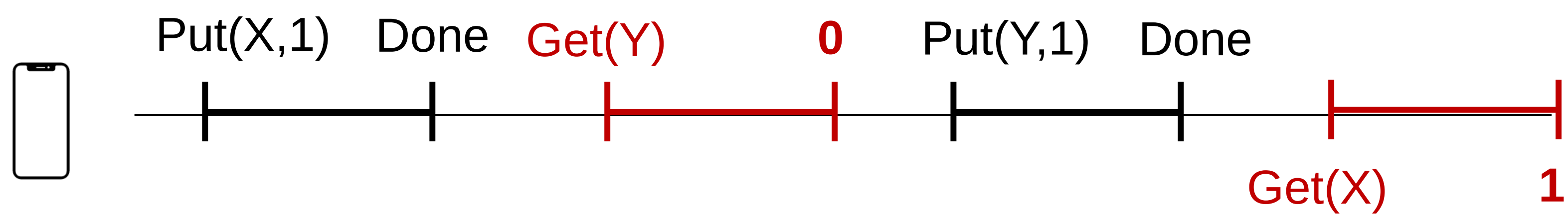


Case #4.

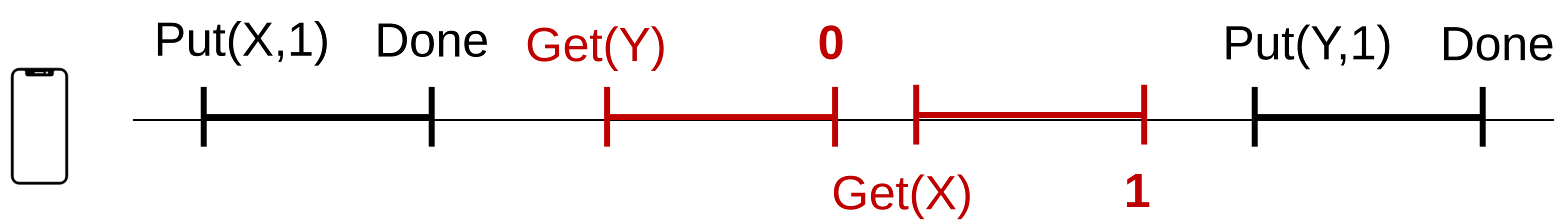


Proof: we cannot find an equivalent serial execution

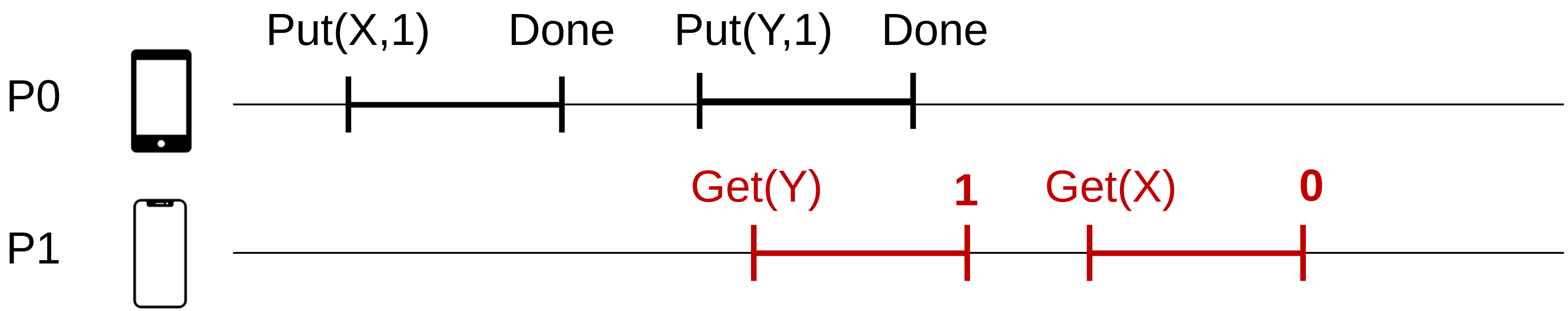
Case #5.



Case #6.



Key problem: order mismatches



Update order observed by P0:

– X first, then Y

Update order observed by P1:

– Y first, then X

In a serial order, only one can happen

Recall: What is the right model?

It's easy for users to reason about correctness assuming

- Everything has only one-copy
- The overall behavior is equivalent to **some serial behavior**

Typically, a convenient consistency model is defined by

- Every data has only “one copy” (logically)
- The concurrent read/write behavior is equivalent to some serial order

Question: which serial order to give?

- So many (possible) serial behaviors

How to define the equivalent serial order?

#1. Global issuing order (strict consistency)

#2. Per-process issuing/completion order (sequential consistency)

#3. Global "completion-to-issuing" order (linearizability)

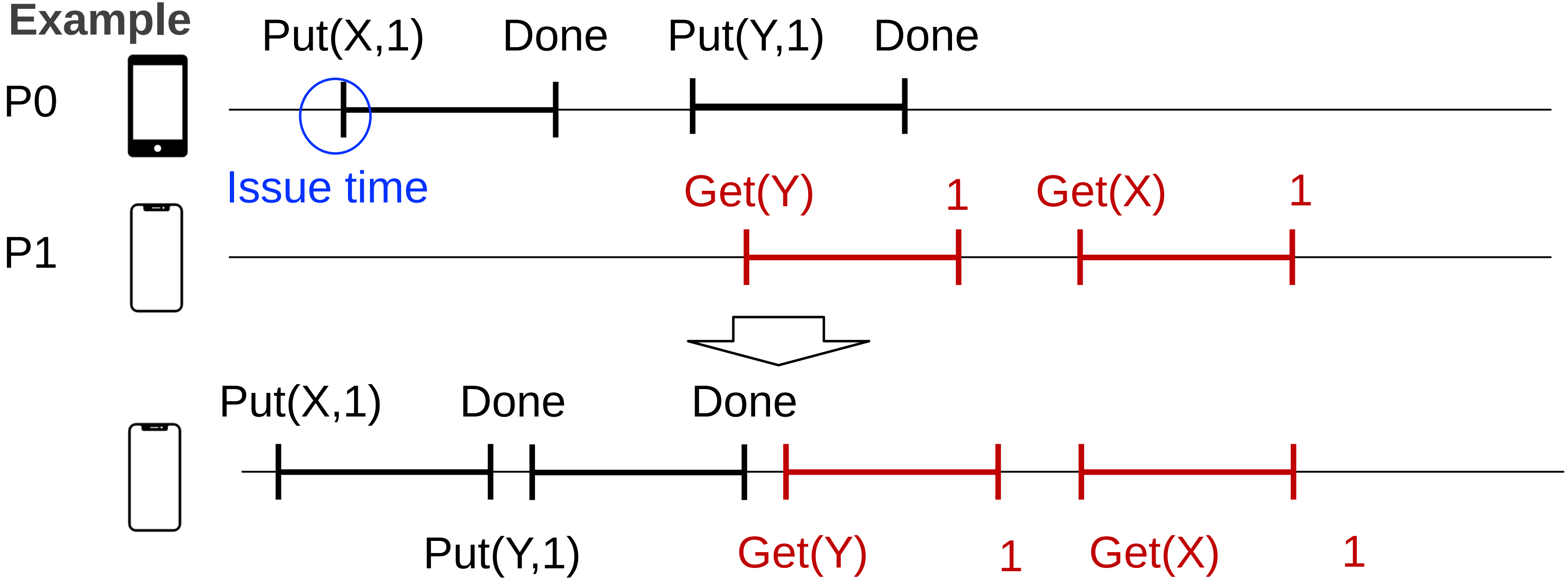
Try #1. Use strict consistency

Strict consistency: global issuing order

– All the concurrent execution is equivalent to a serial execution

– The order of each op matches the global wall clock time

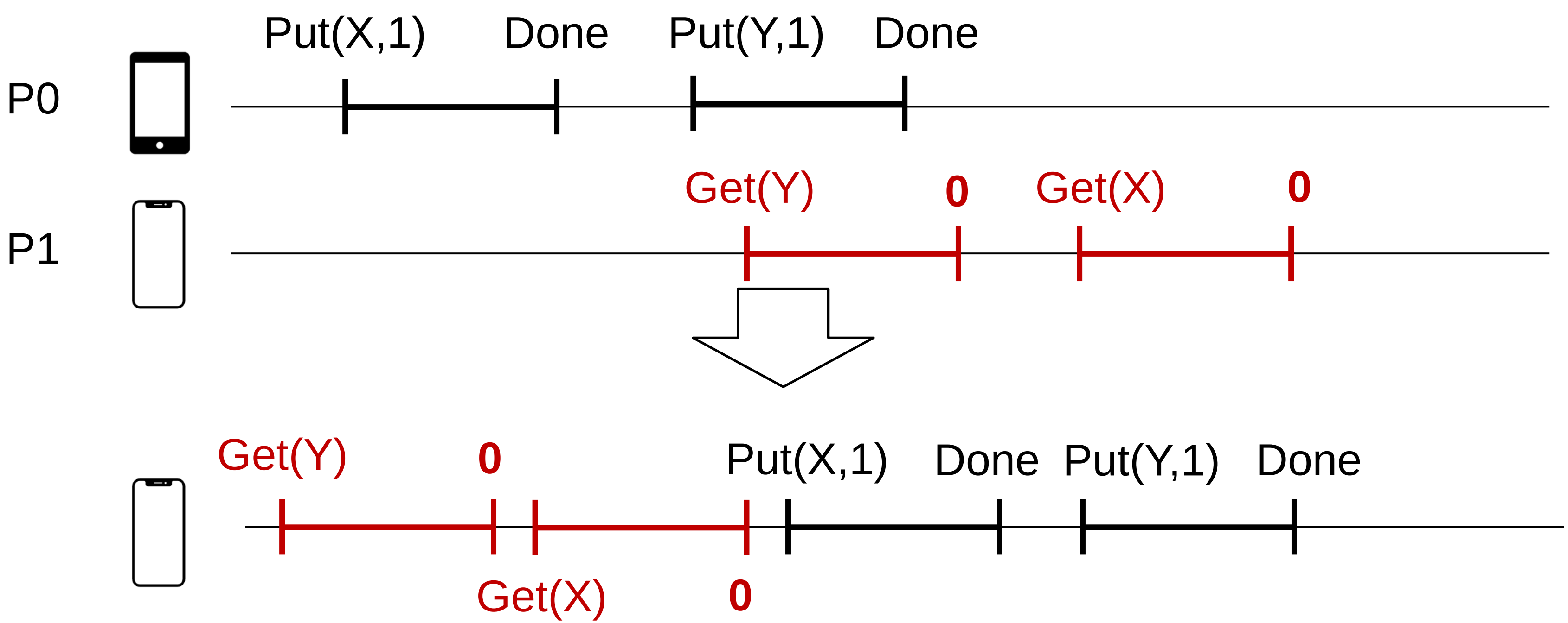
Example



Try #1. Use strict consistency

Is the following serial order matches global issuing time?

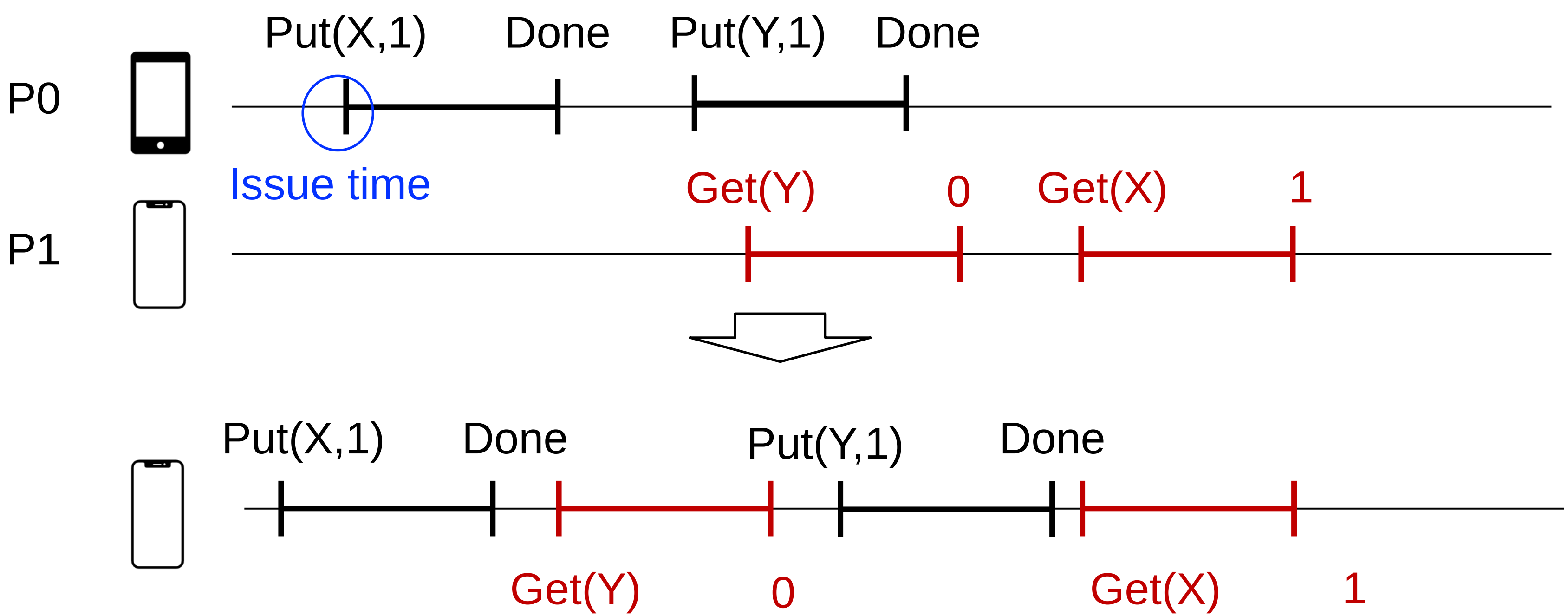
– No. Put (X,1) must be executed first.



Try #1. Use strict consistency

Is the following serial order matches global issuing time?

– No. Put(Y) must be executed before the Get(Y)



Strict consistency: Pros & Cons

Pros

– The strongest consistency model for single value operations

– Fallback to a single thread, executing each operations one by one

Cons

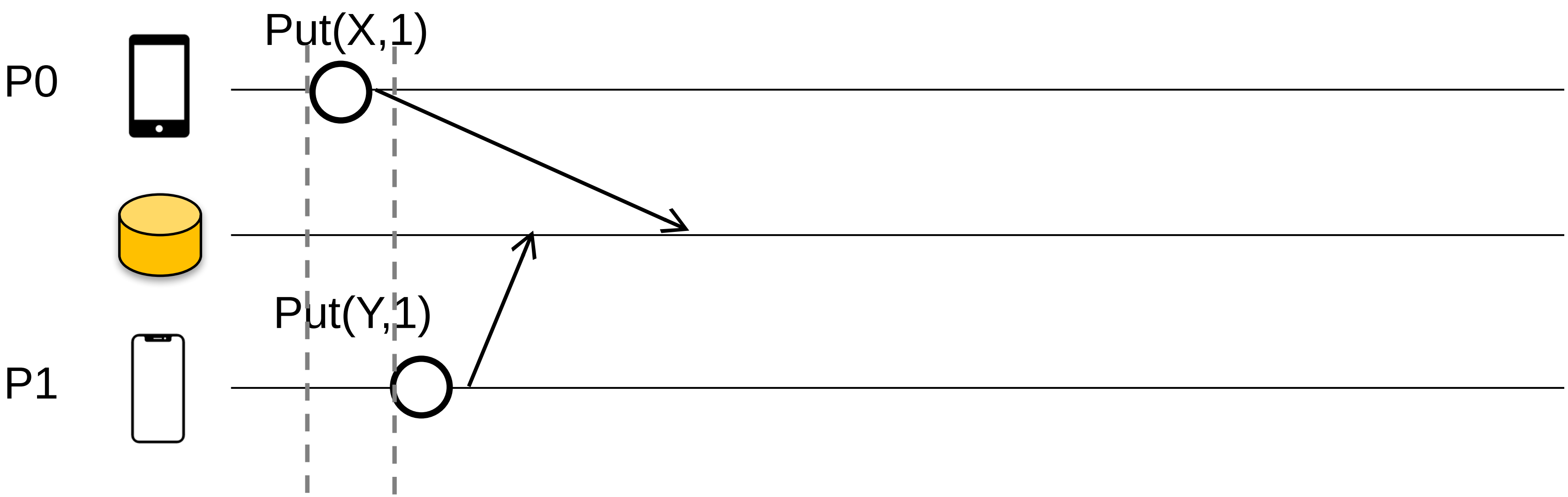
– Nearly impractical to implement (e.g., no global wall clock time)

Implementing strict consistency: challenge

Assuming we implement strict consistency using a centralized KVS

– The simplest setup we can get in a distributed setting

Question: how the KVS can determine to process Put(Y,1) or not?



How to define the equivalent serial order?

#1. Global issuing order (strict consistency)

#2. Per-process issuing/completion order (sequential consistency)

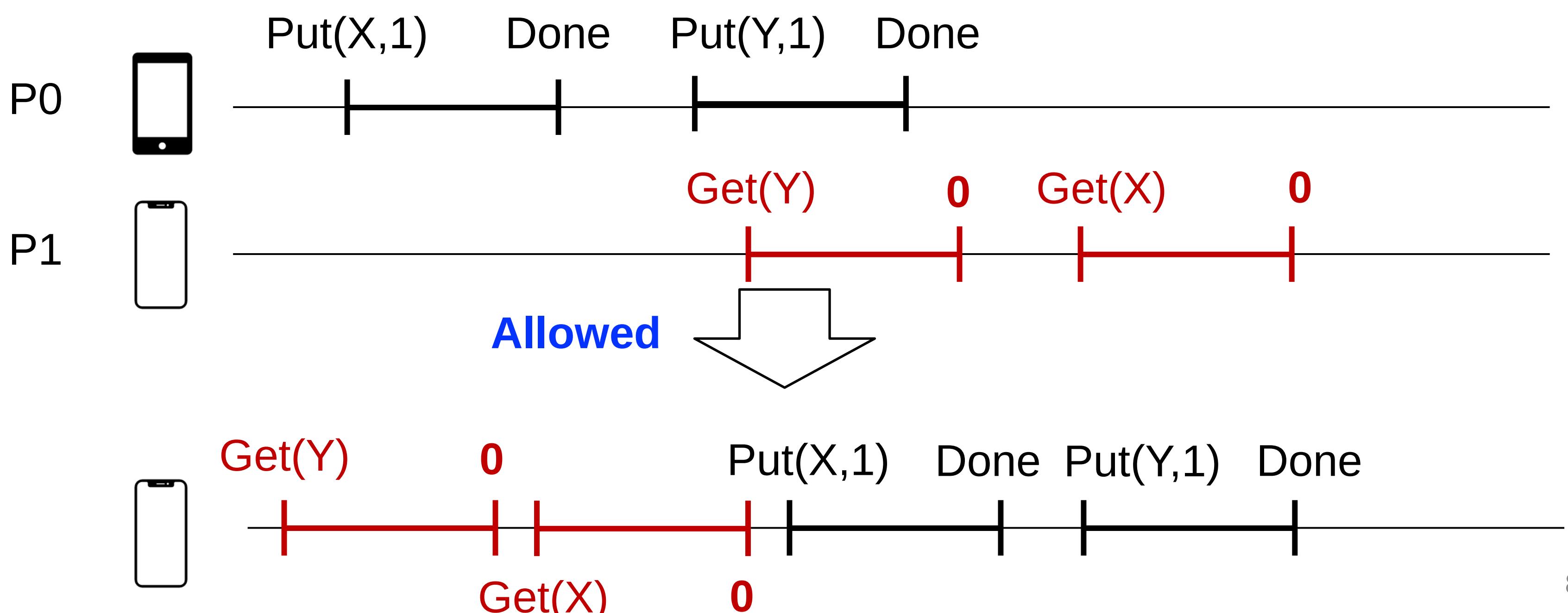
#3. Global "completion-to-issuing" order (linearizability)

Also convenient, but are practical to implement

Try #2. Use sequential consistency

#2. Per-process issuing/completion order (sequential consistency)

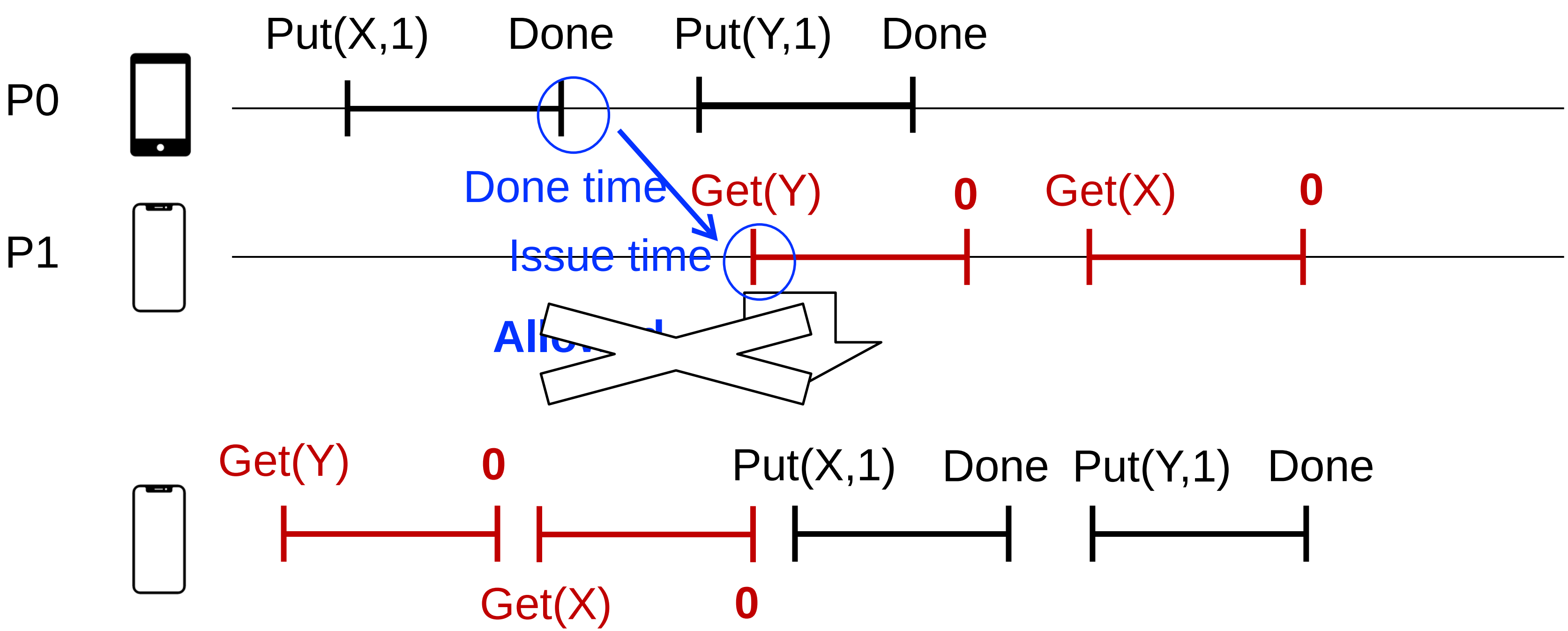
- All the concurrent execution is equivalent to a serial execution
- The order of each op matches per-process issuing/completion order



Try #3. Use linearizability

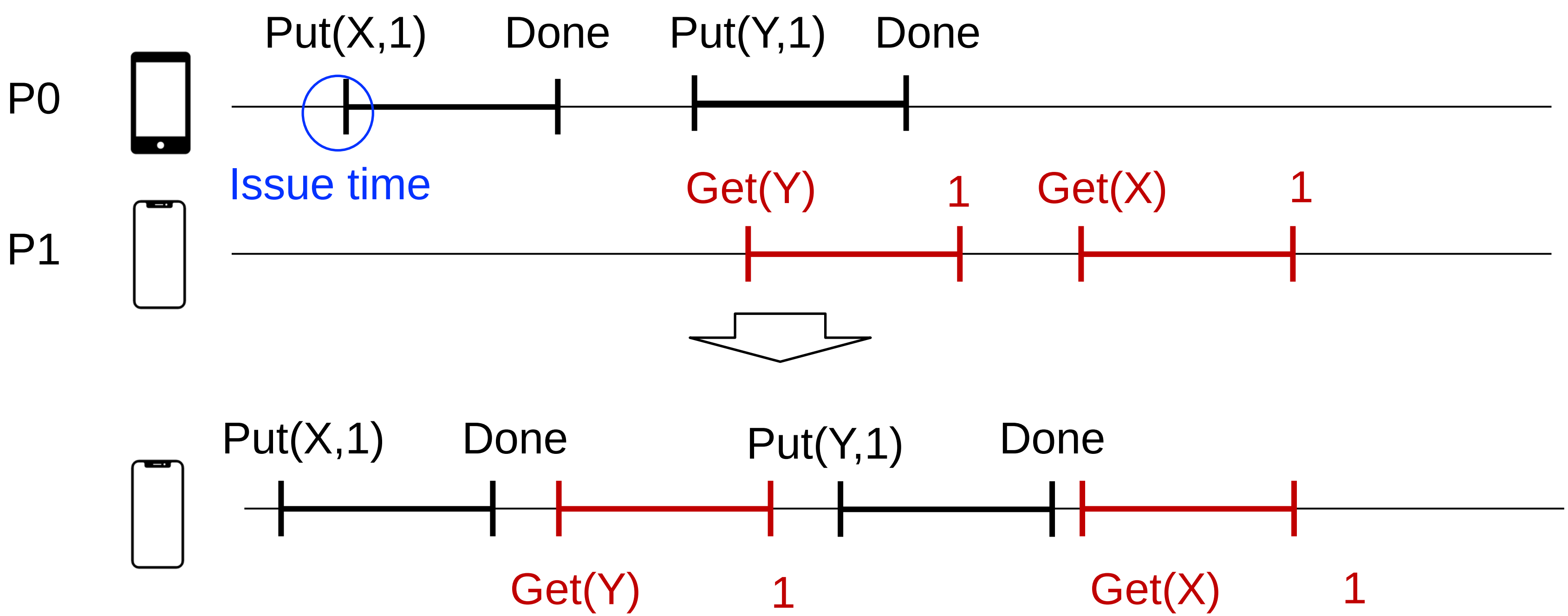
#3. Linearizability "completion-to-issuing" order

- All the concurrent execution is equivalent to a serial execution
- The order of each op matches "completion-to-issuing"



Question: what are the differences between 1 & 3?

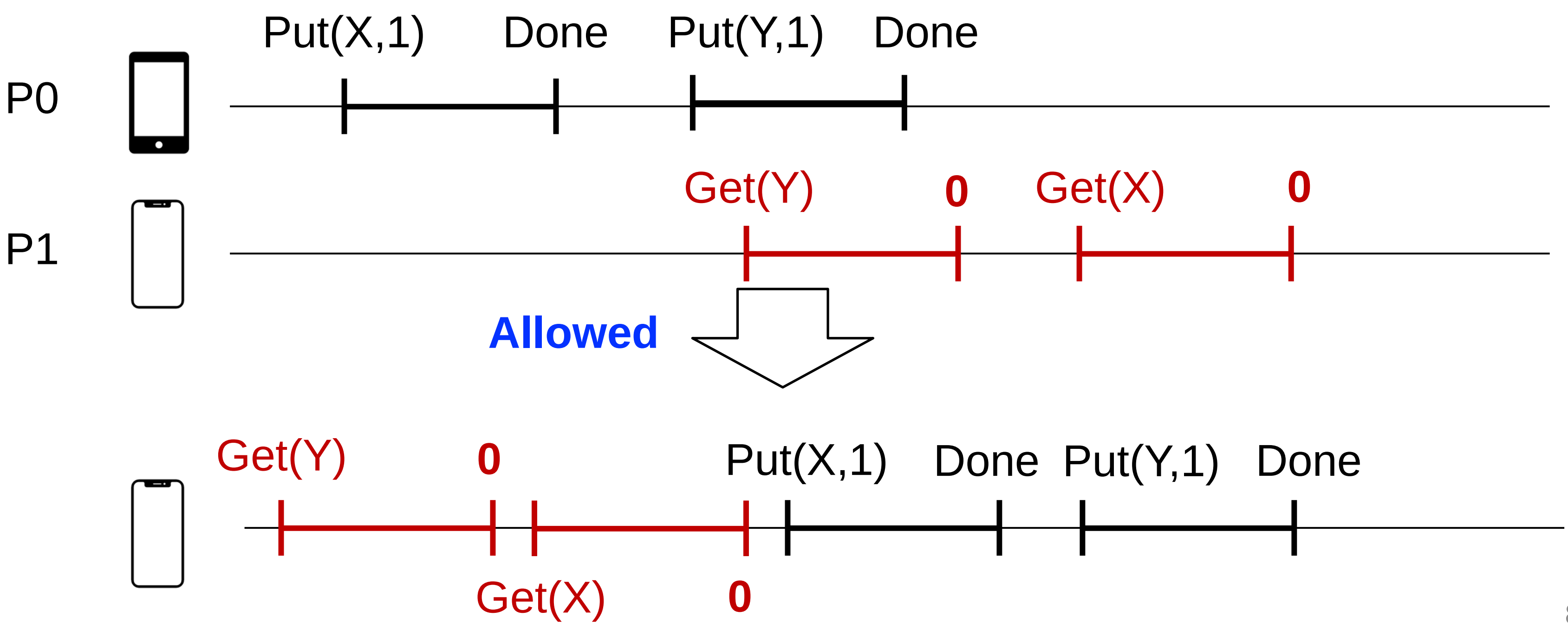
Recall: the following scenario can happen in 3 but not 1



2 or 3, which to choose?

In practice, 3 is in favor of 2

– E.g., P0 finishes X. But later P1 cannot sees its effect!



Implementing linearizability of KVS

Warmup property of linearizability: The local property [1]

If each object's op is linearizable, then overall system is linearizable

- Our implementation only needs to focus on a single object !

(Very) Simplified & (very) informal proof (By contradiction)

- Suppose we have two ops on x,y, e1 & e2
- If non-linearizable, then we must have e1 < e2 & e2 < e1
- This is impossible:
  - e1 < e2 means real_time(e1_ok) < real_time(e2_start)
  - real_time(e2_start) < real_time(e2_ok)
  - e2 < e1 means real_time(e2_ok) < real_time(e1_start)
  - Since real_time(e1_start) < real_time(e1_ok), contradiction happens
- The concrete proof needs to reason on multiple ops & objects (w/ graph)

[1] Linearizability: A Correctness Condition for Concurrent Objects， TPLS'90

Why locality is important to implement linearzability?

We only need to ensure operations on a single object is linearzable!

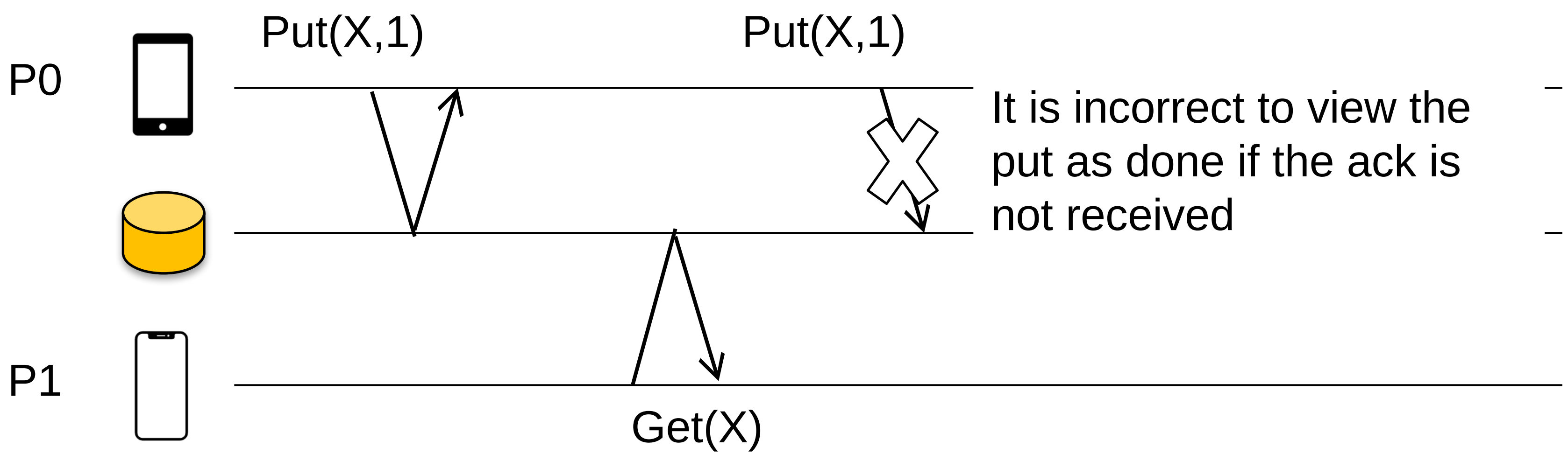
The simplest approach: centralized KVS

Basic model

– There is only one centralized KVS in the system

Put: Send an RPC to the KVS, wait for it to be done

Get: Send an RPC to the KVS to read the result

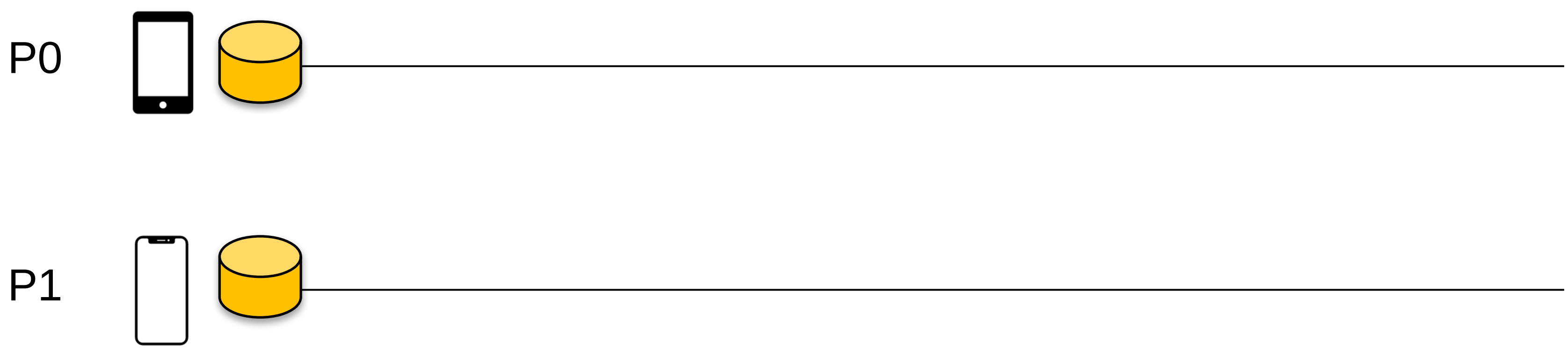


Approaches for replicated KVS: replicated KVS

Model (More suitable for the chat app)

– Each device has a replicated KVS on its local machine

– Question: how can the Get/Put being implemented?



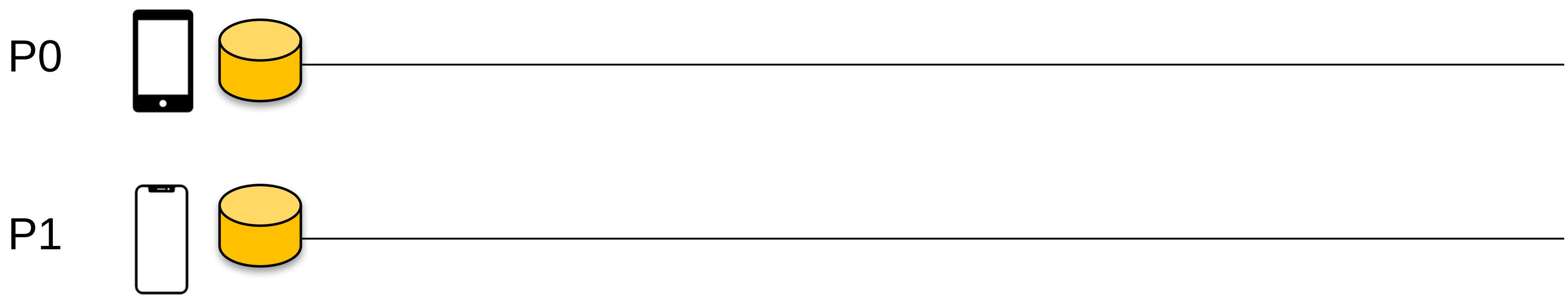
Approaches for replicated KVS (more realistic)

Put

- How many servers need the client send the updates to?
- Must a client wait till all servers sent have processed the update?

Read

- Which server will the read sent to?
- Can a server always return its current value?

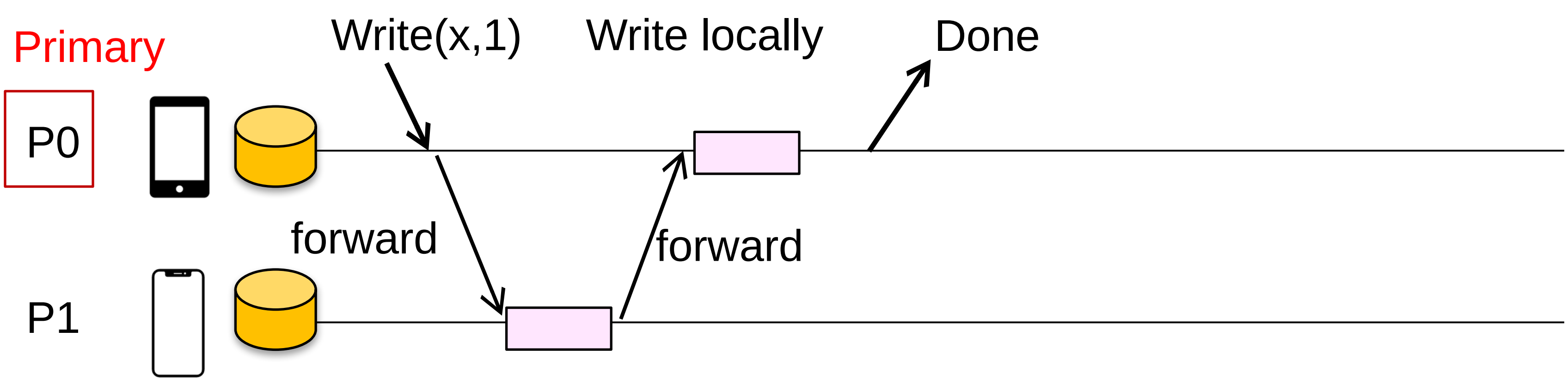


Approach #1. Primary-backup model

For each object, Clients send all reads/writes at a designated machine

For writes

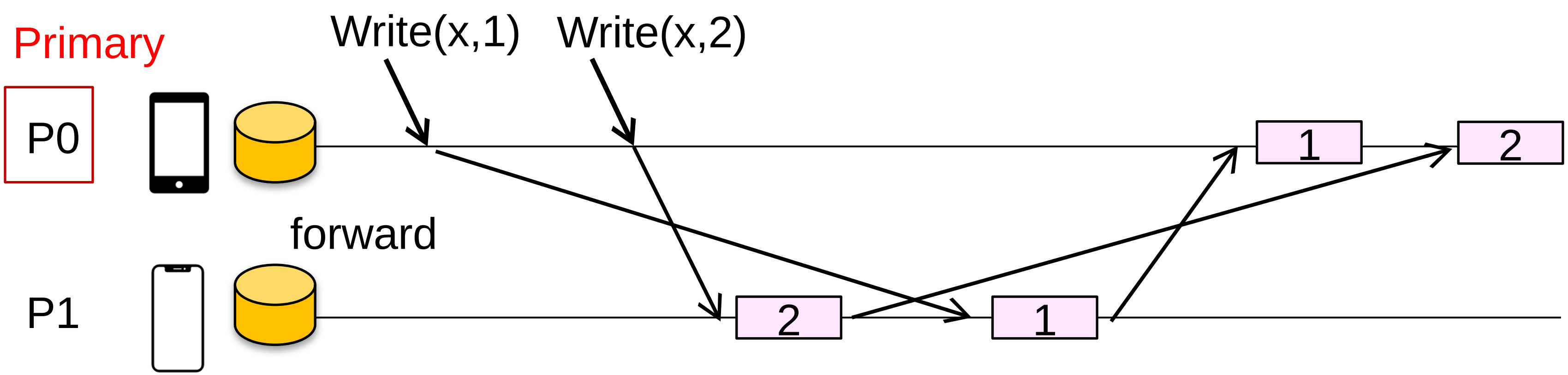
1. Primary forwards writes to all the replicas
2. M0 executes writes locally (in order)
3. Respond OK



What does the 「in order」 mean?

Suppose we have two writes send to the primary

- Due to network problem, the message may reorder
- If the reorder happens, it is possible that two replicas apply the same writes in a different order!



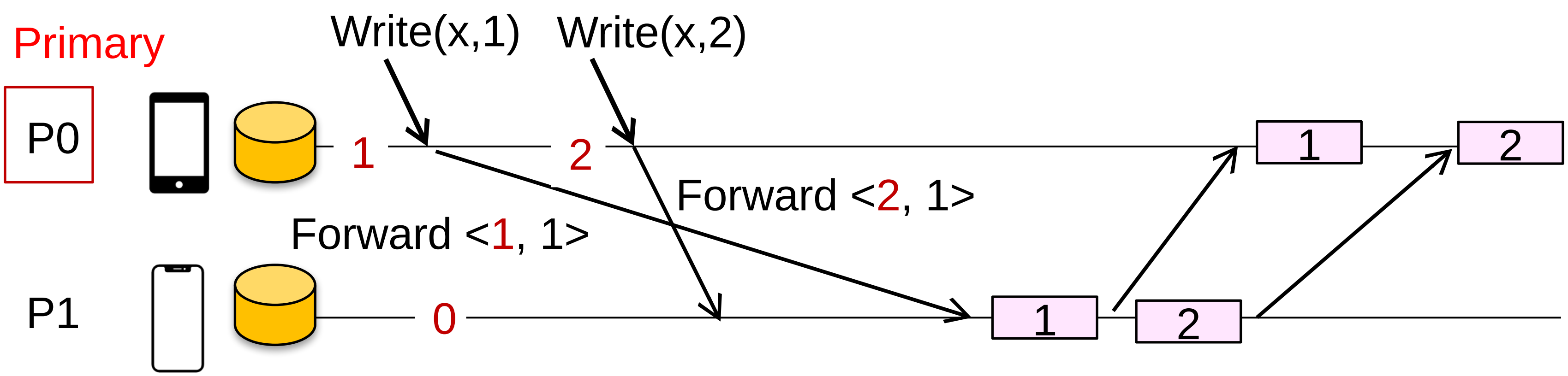
In order updates: Primary must use some seq number

Seq number: orders of update

- Possible implementation: a global counter
- Incremented upon receive a write request

All replica apply writes in the order of seq number

- Delay writes if the previous write has not been finished



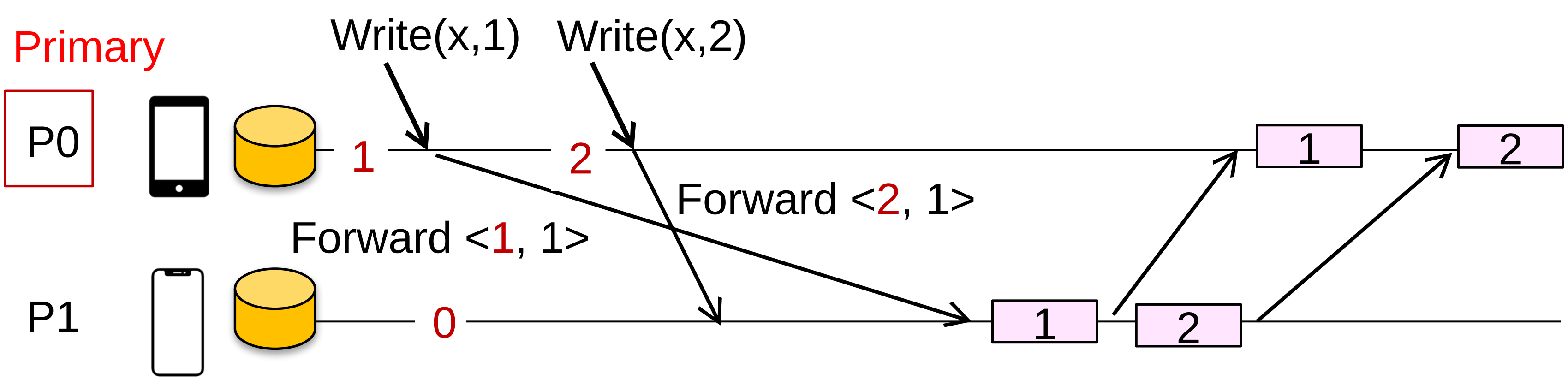
Question: where to implement the in-order semantic?

At the network layer

- May rely on the transport (e.g., RPC or TCP/IP) layer to implement

But we can also implement it at the application layer

- E.g., the primary stores a global counter
- Not so hard, and is more flexible

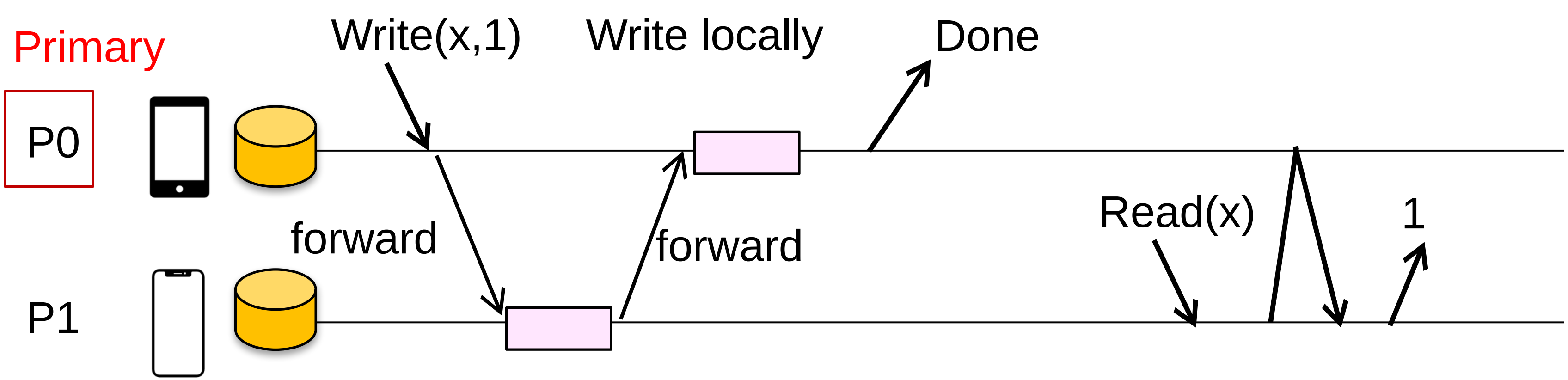


Approach #1. Primary-backup model

For writes

1. Primary forwards writes to all the replicas
2. M0 executes writes locally (in order)
3. Respond OK

Read: return the local copy of the data of the primary

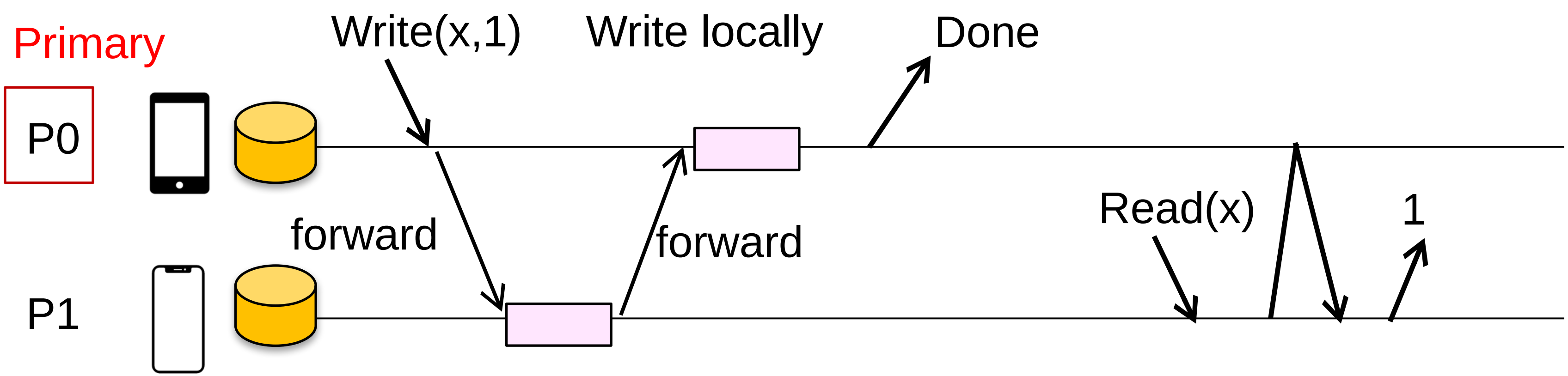


Drawback of the primary-backup model

Performance issues of reads and writes

- Read: extra RTT for contacting the primary
- Writes: extra RTTs for contacting the primary + backups
- Scalability issue: the primary may become the bottleneck!

~~Reliability issue: what if some primary of backups crash?~~ (Not today)

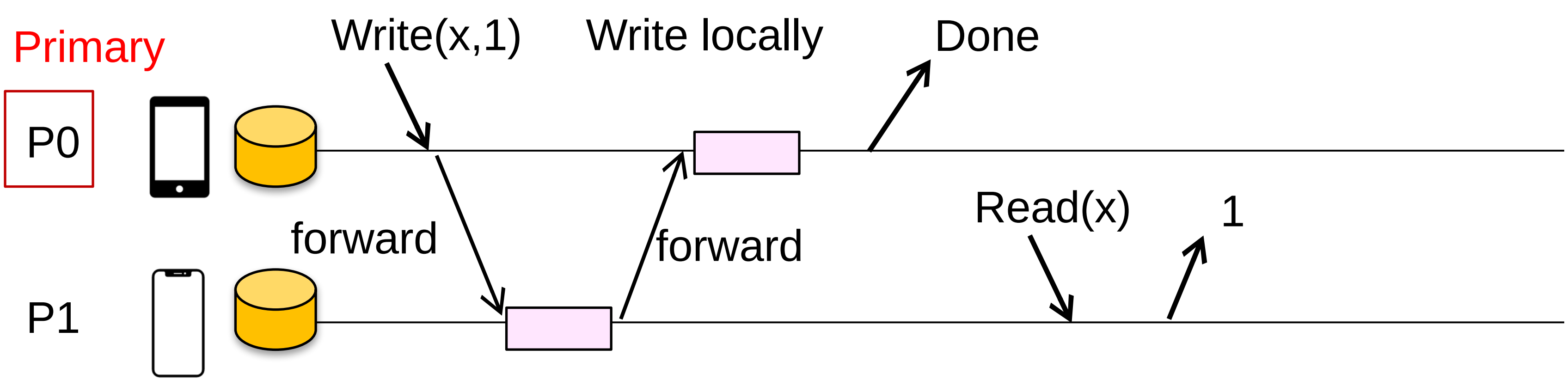


Approach #2. Primary-backup model + Relaxed reads

For writes

1. Primary forwards writes to all the replicas
2. M0 executes writes locally (in order)
3. Respond OK

Read: return the local copy of the data on any replica

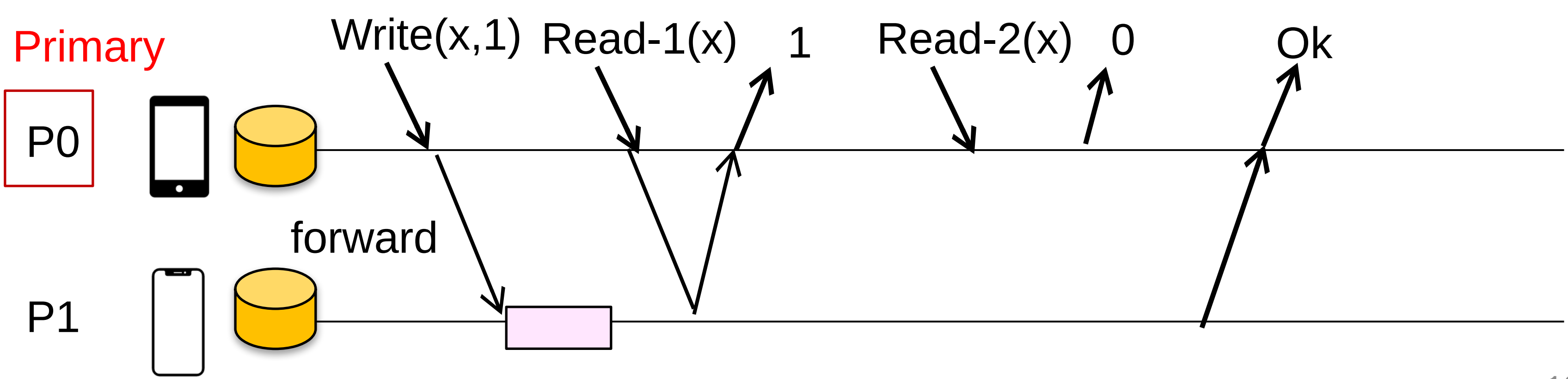


Question: Is approach #2 linearizable?

Read-2 < Read-1

Write < Read-1

Read-1 < Read-2 (Read-1's completion is before Read-2)



Summary

It is challenging to distributed object distributed

– Consistency issue

It is also challenging to define the consistency model

– Different trade-offs

Correct consistency model is defined via serial execution

– Strict

– Sequential

– Linearizable

Implementation trades performance (and reliability) for correctness