Building a Highly-Scalable Highly-Available Transactional Data Store

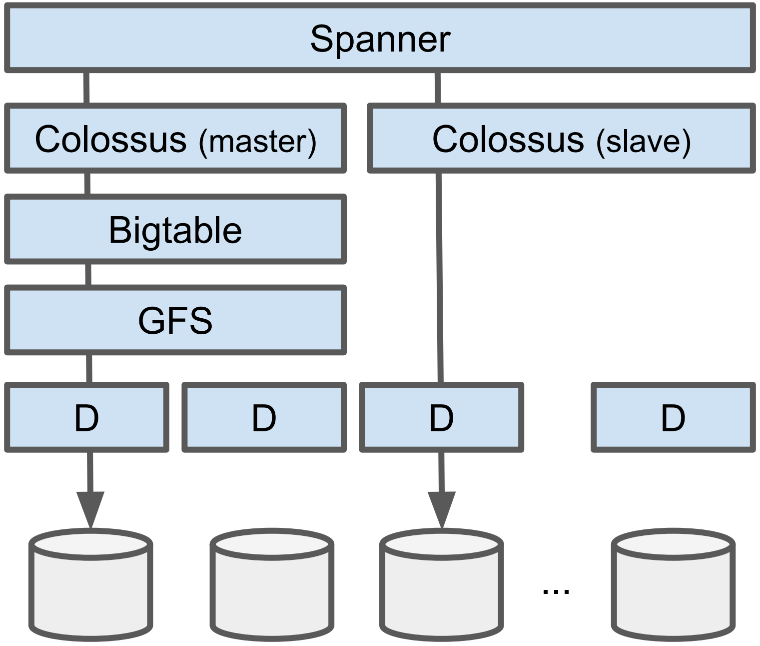
Outline of This Talk

- I hate MySQL and want to build Spanner!
- What will be the shortest path?

Spanner and Other Storage Systems at Google

- Too many software stacks
  with long development history

- Should be able to design in a clearer way



Incremental Steps

- **Step 1.** Build a key-value store on a single node
- **Step 2.** Make it highly-available (e.g., replication)
- **Step 3.** Make it highly-scalable (e.g., sharding)
- **Step 4.** Support ACID transactions
- **Step 5.** Support SQL

Step 1

Build a Key-value Store on a Single Node

KV Store API

- **`Get`**`(k)`
- **`Put`**`(k,v)`
- **`ConditionalPut`**`(k,v,ev)`
- **`Scan`**`(s,e)`
- **`Delete`**`(k)`
- **`DeleteRange`**`(s,e)`

- Keys and values are arbitrary byte arrays

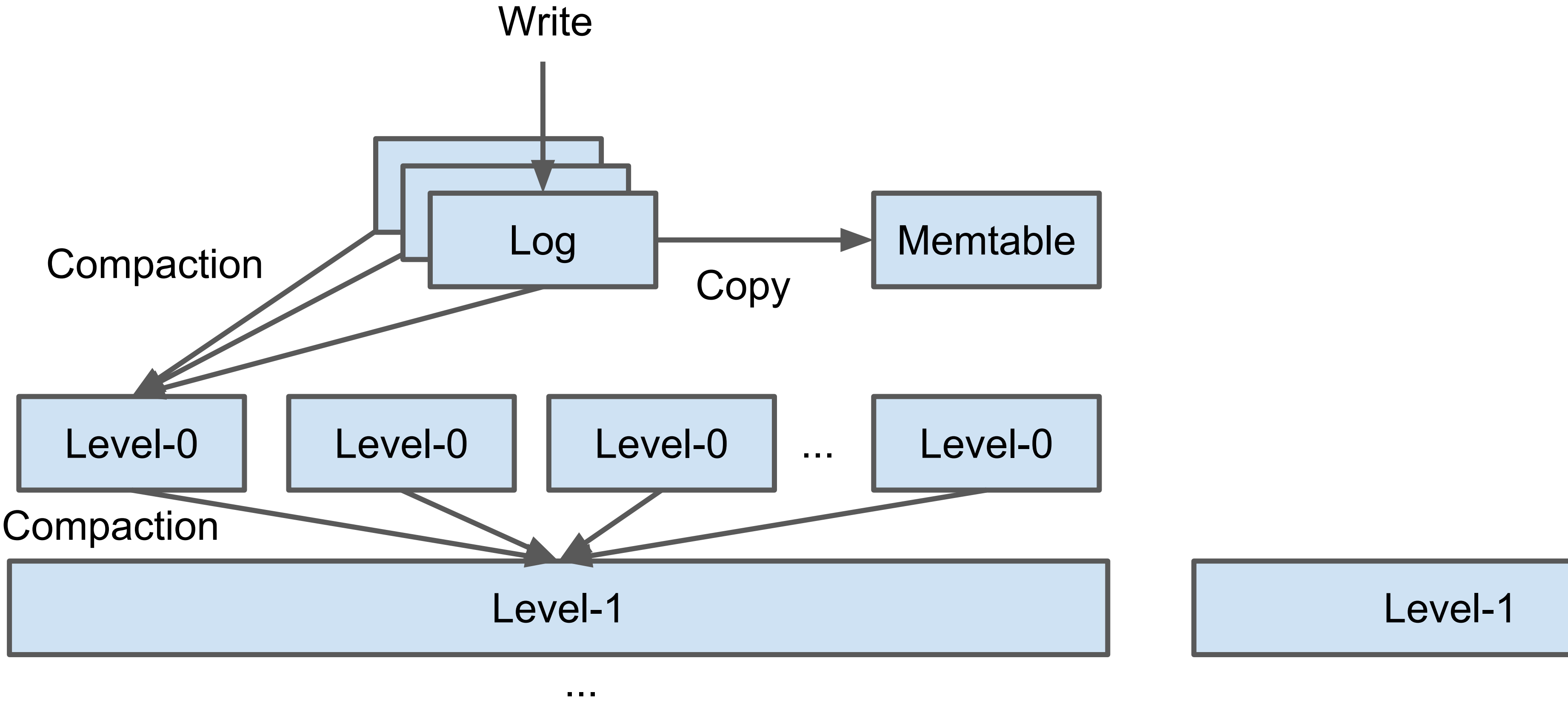
Existing KV Store Implementations

- Log-structured merge tree
  - E.g.) LevelDB, RocksDB
  - Used by Bigtable, HBase, Spanner, MongoDB, SqLite4, InfluxDB, ...
- ...

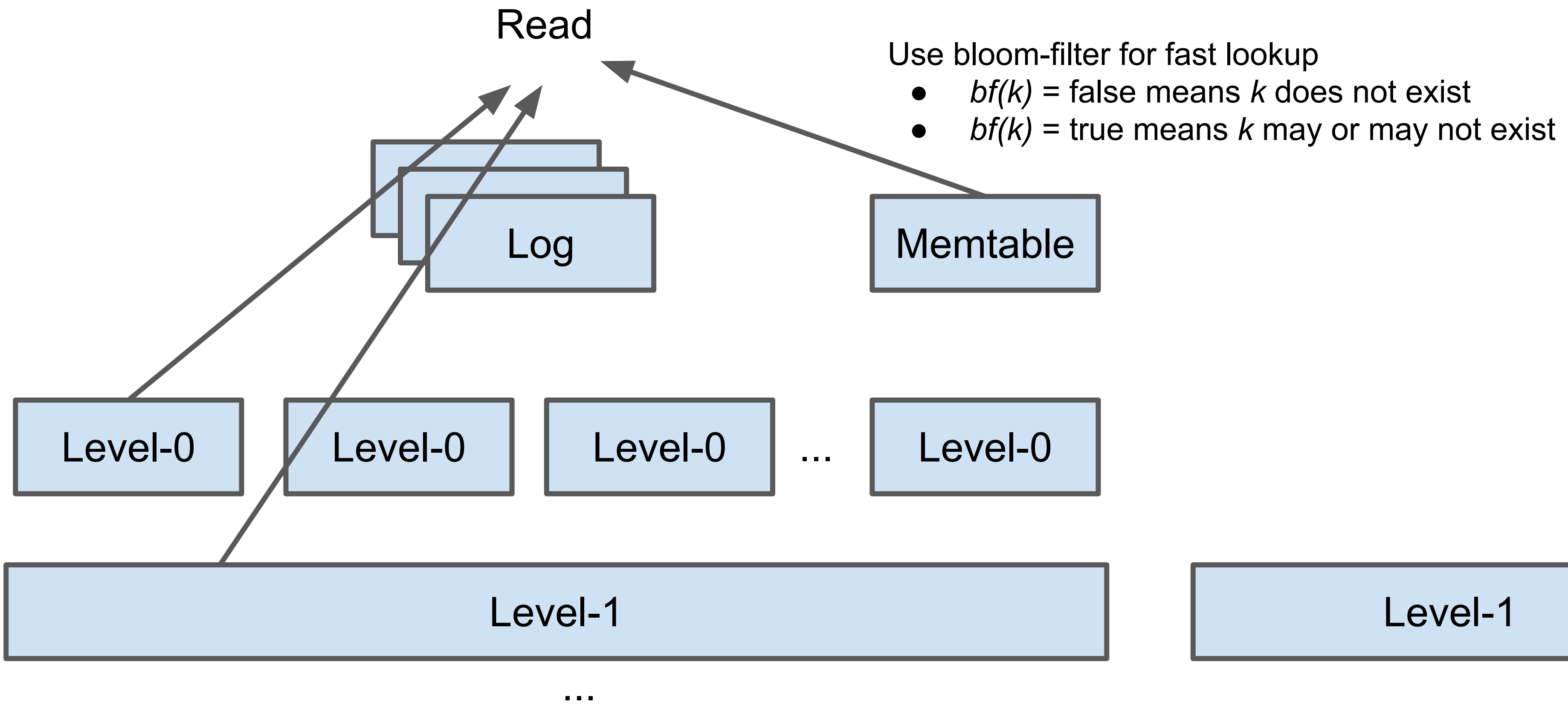
LevelDB Overview

- Log files
  - Store a sequence of recent updates
  - Converted to a sorted table when a file reaches a pre-determined size (e.g., 4MB)
- Sorted tables
  - Store a sequence of entries sorted by keys
  - Organized into a set of levels
- Memtable
  - Copy of the current log file
  - Consulted on every read

LevelDB Put


Write
Log
Memtable
Copy
Compaction
Level-0
Level-0
Level-0
...
Level-0
Compaction
Level-1
Level-1
...

LevelDB Get

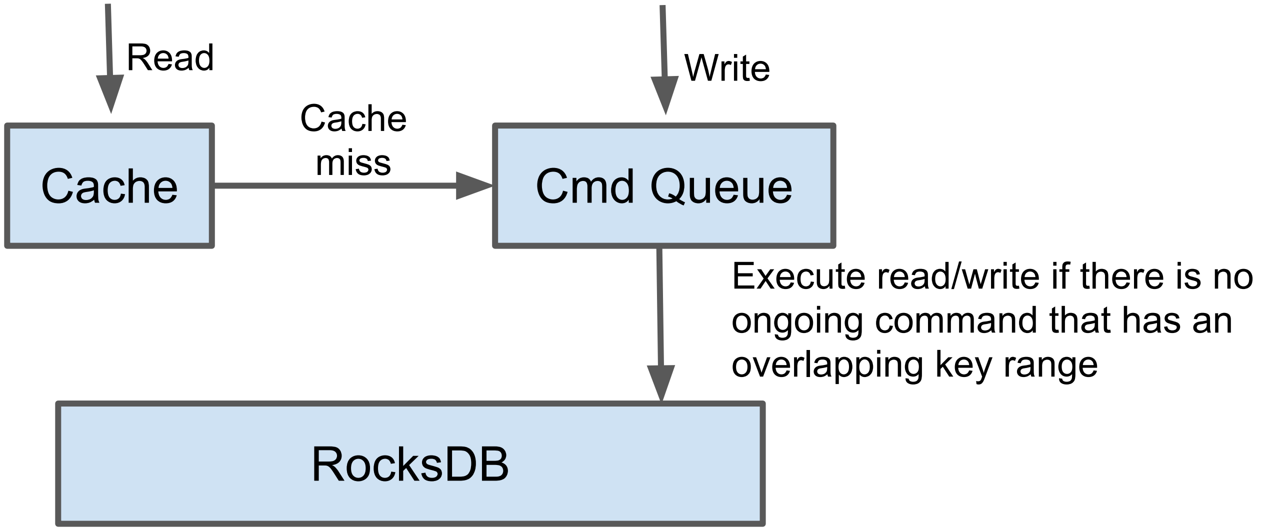


Use bloom-filter for fast lookup

- *bf(k)* = false means *k* does not exist
- *bf(k)* = true means *k* may or may not exist

Our KV Store Architecture

- RocksDB
  - Variant of LevelDB developed by Facebook
- In-memory cache for read
- Command queue to prevent concurrent write-write or read-write



Step 2

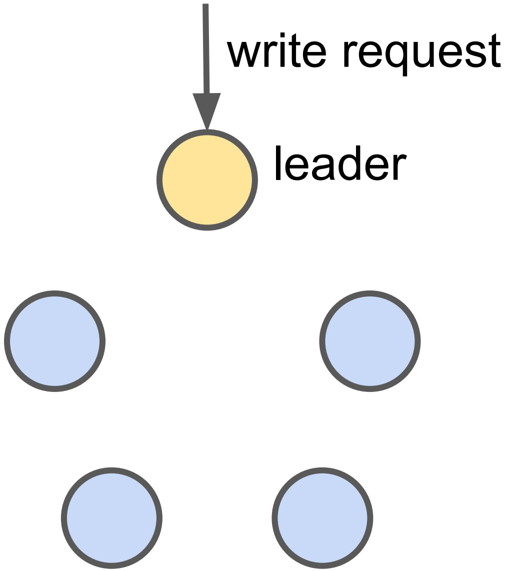
Make it Highly Available

Goal

- Make the database tolerate failures
  - No data loss (= no eventual consistency replication)
  - Tolerance to at least two node failures (one planned + one unplanned)

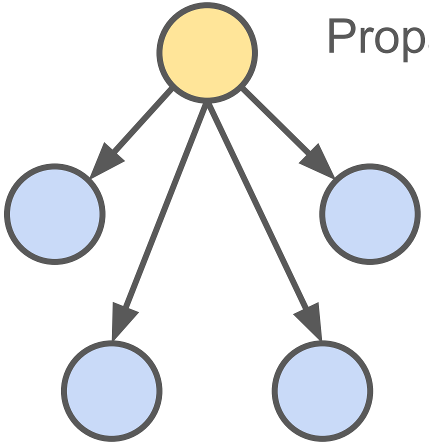
Basic Idea

- Elected leader commits a write after it was written to a majority of nodes



Basic Idea

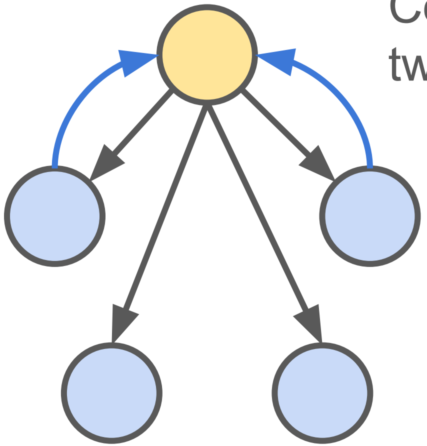
- Elected leader commits a write after it was written to a majority of nodes



Propagate the write to the followers

Basic Idea

- Elected leader commits a write after it was written to a majority of nodes



Commit when at least two nodes responded

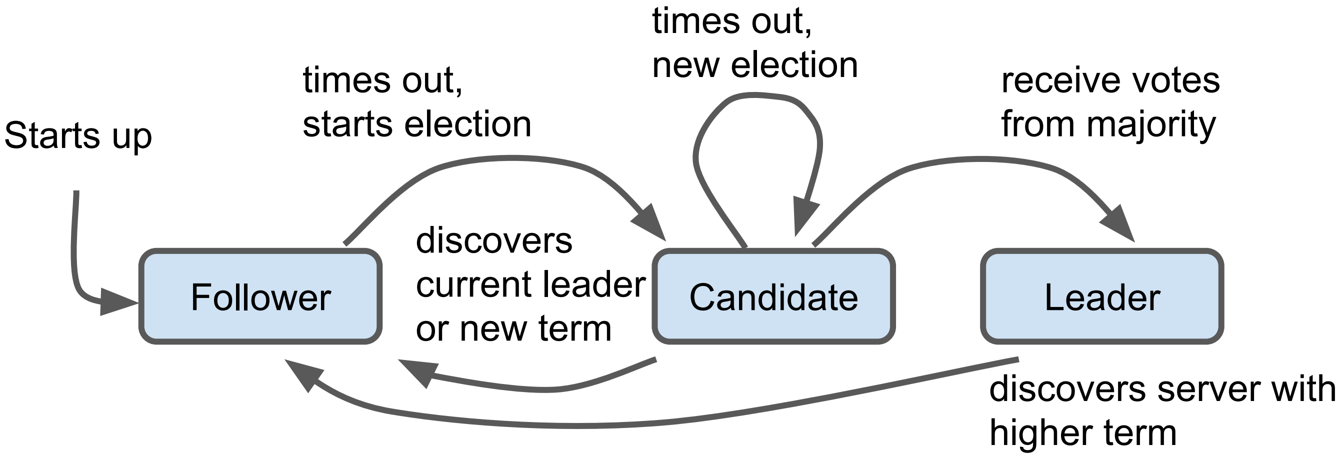
Existing Consensus Algorithms

- Paxos (1998)
  - Heavily used inside Google (but tuned a lot)
  - No major open source implementation
- Zab (2011)
  - Used by ZooKeeper
  - Might not be easy to factor out the consensus logic from ZK
- **Raft (2014)**
  - Simple, easy to understand
  - Reference implementations (e.g., etcd)
- ...

Let's focus on correctness
Performance can be improved later

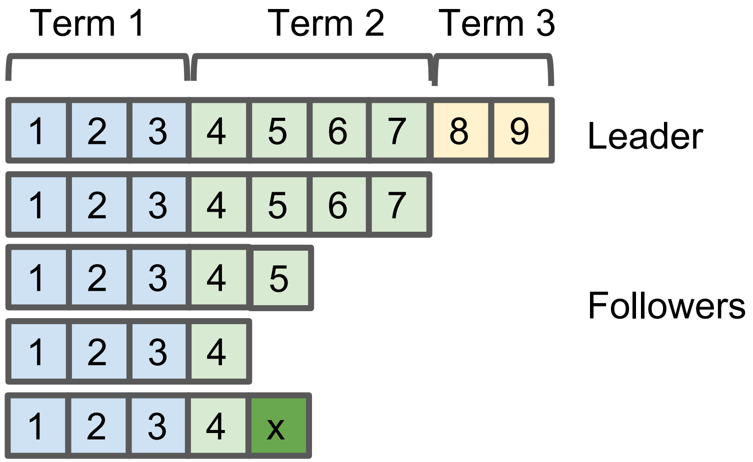
Raft Algorithm Overview

- Leader election
  - A leader is elected in each “term”
  - A candidate wins an election if it receives votes from a majority of the servers
- Log replication
  - Leader appends a proposed command to its log
  - When the log entry has been safely replicated, the leader commits it

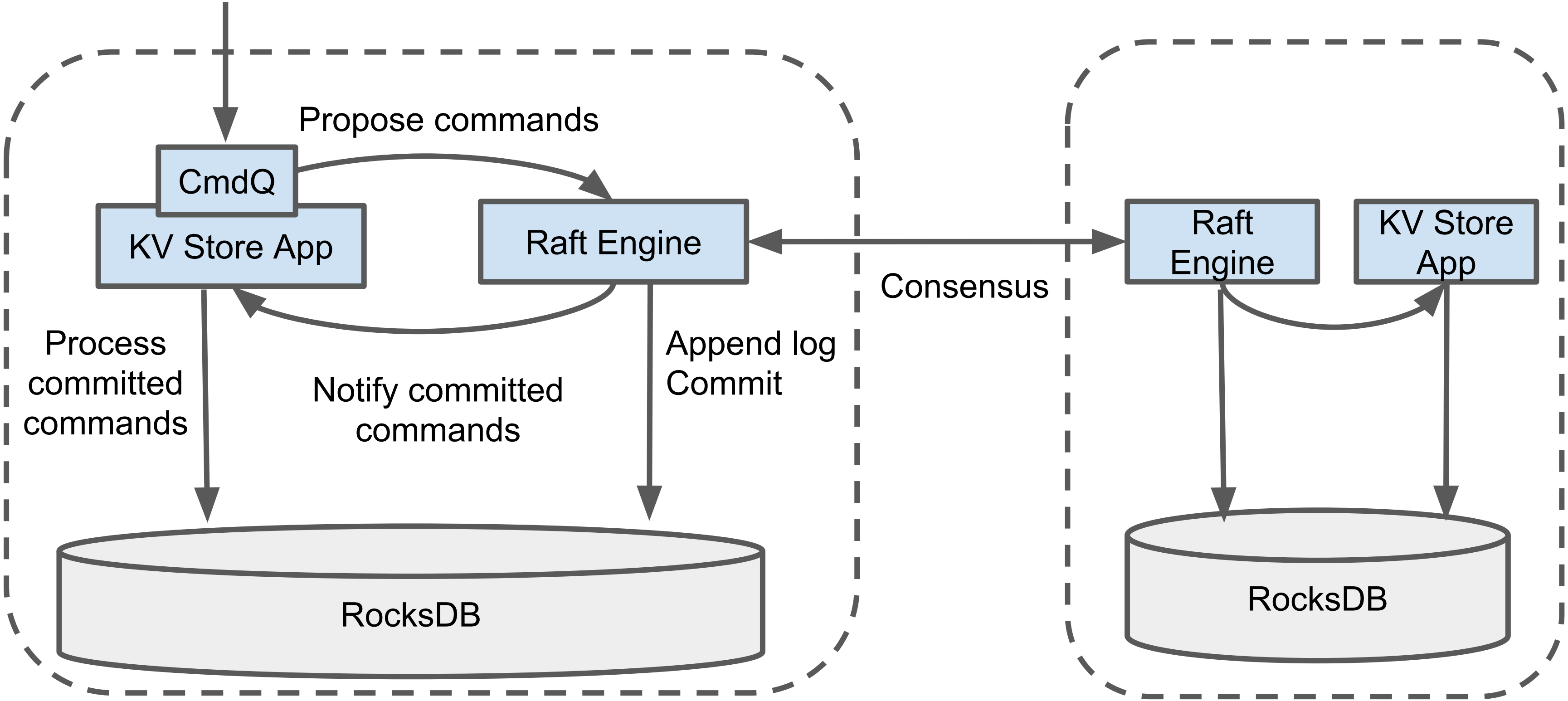


Raft Algorithm Overview

- Leader election
  - A leader is elected in each “term”
  - A candidate wins an election if it receives votes from a majority of the servers
- Log replication
  - Leader appends a proposed command to its log
  - When the log entry has been safely replicated, the leader commits it



KV Store with Raft Replication



Details of Command Execution

- Execution of each command must produce a deterministic result
  - Otherwise the states of replicas will diverge
  - E.g.) Do not use now() or a randomly-generated number
  - E.g.) Do not access in-memory states set only in a leader
- Command execution and update to the applied index must be done in the same batch
  - Raft keeps track of which commands have been applied
  - If the KV store processes the command but crashes before updating the applied index, the same command will be executed again
- All writes are made to RocksDB sequentially

What about Read Commands?

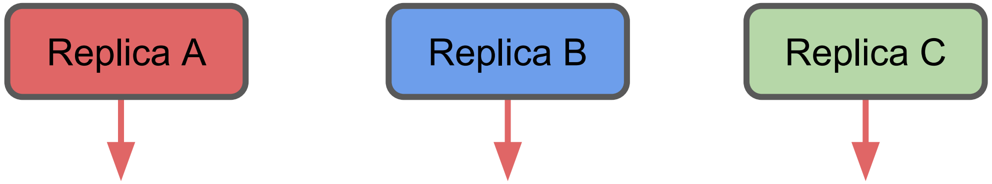
- Process in a random replica
  - Lose consistency guarantee
- Propose commands to Raft
  - Too inefficient
- Process in a Raft leader
  - Doable, but break the API abstraction
- **Implement a leader-election on top of Raft**
  - **Process read (and write) requests in an elected leader**

Leader Election on top of Raft

- New `LeaderLease` command
  - Lease start time
  - Expiration time
  - ID of a replica that wants to acquire a lease
- Every replica runs the same `LeaderLease` command via Raft
- Replica can acquire a lease at time *T* if there is no existing lease that will expire after *T*

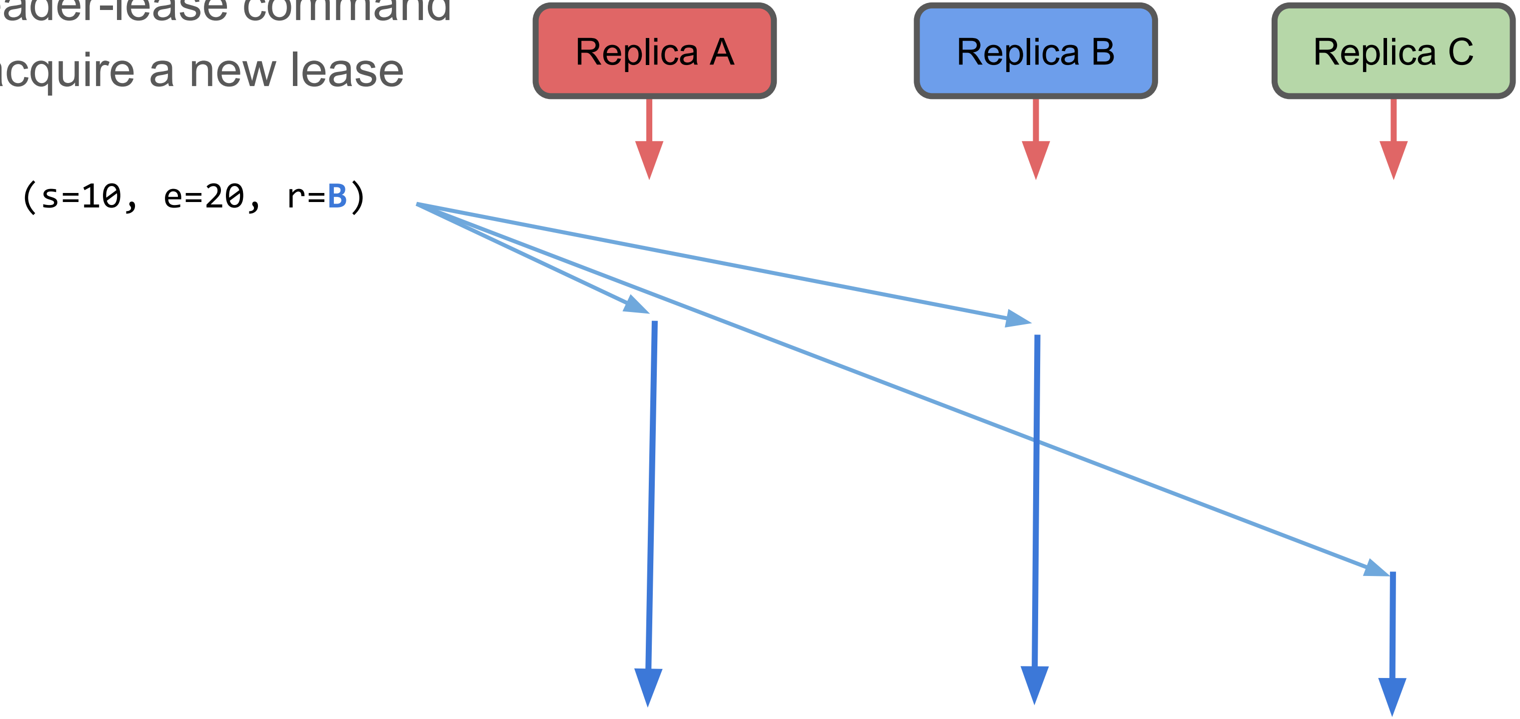
Leader Election Process

1) Replica A’s lease expires



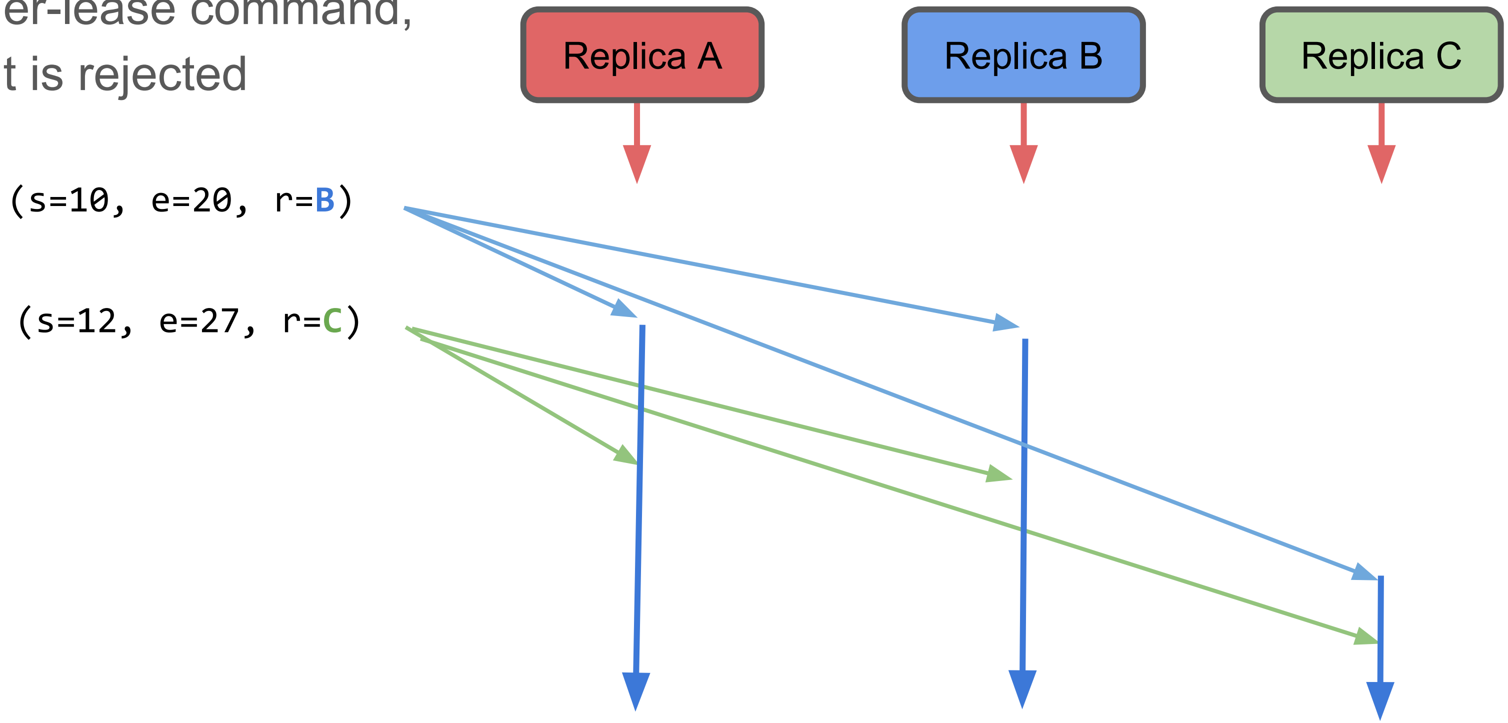
Leader Election Process

2) Replica B proposes a leader-lease command to acquire a new lease



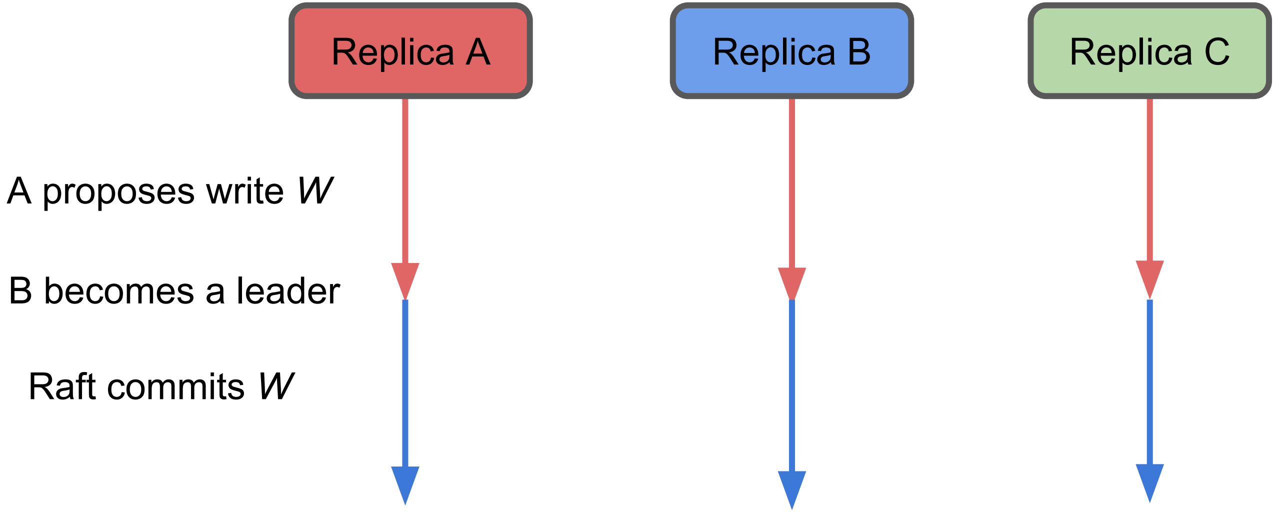
Leader Election Process

3) Replica C proposes a leader-lease command, but it is rejected



Some Complex Scenarios

- What if a lease expires after a write is proposed but before it is committed?
    - Replica ignores a committed write from replica *X* if *X* is no longer a leader
- Keep some buffer between lease exchange to work around clock skew



Skipped Topics

- Membership change
- Snapshot
- Failure detection
- Bootstrap
- ....

Short Break: CAP Theorem

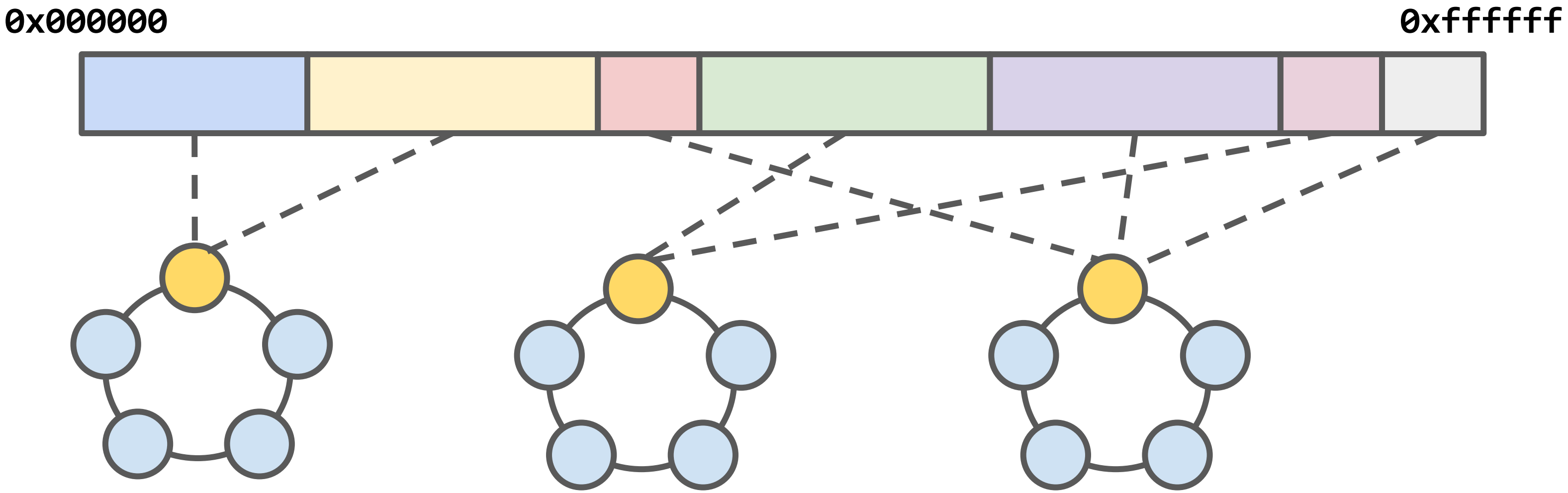
- Trade-off between consistency, availability, and partition tolerance
- Impossibility of guaranteeing both **safety** and **liveness** in an **unreliable distributed system**
  - Safety: nothing bad happens
  - Liveness: eventually something good happens
  - Unreliable: partition, crash failures, message loss, malicious attacks, Byzantine failures, ...
- Implication to practical distributed systems
  - Best-effort availability
  - Your system won’t make progress when every machine is constantly failing
  - But, it’s safe to assume that such a catastrophic failure rarely happens in practice

Step 3

Make it Highly Scalable

Sharding the KV Store

- Split the key space into multiple buckets
- Distribute buckets over multiple nodes
- Create a Raft group per bucket



Challenges

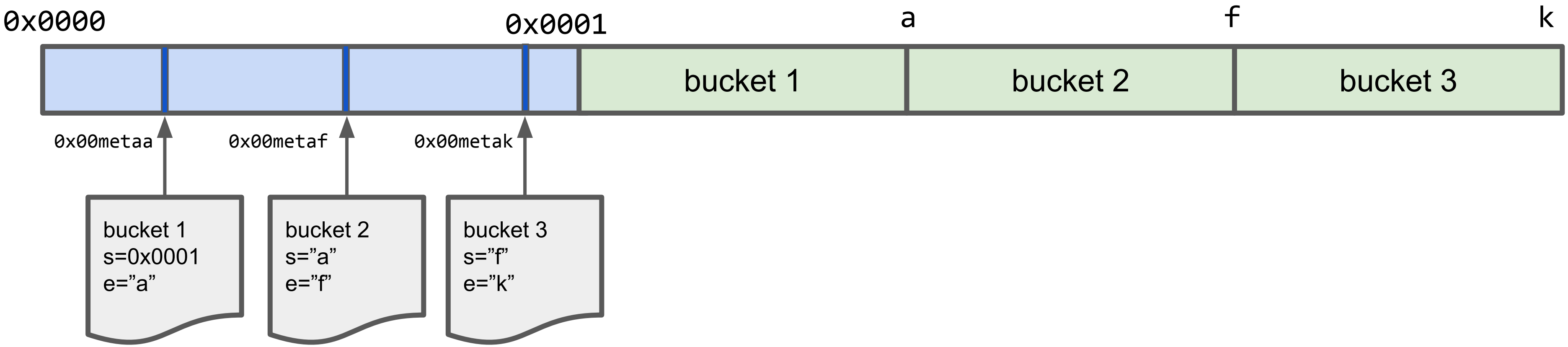
- Key/bucket lookup
  - Which bucket has key K?
  - Which node has bucket B?
- Scan requests touching multiple buckets
  - How can we atomically process such scan requests?

Looking up Keys and Buckets

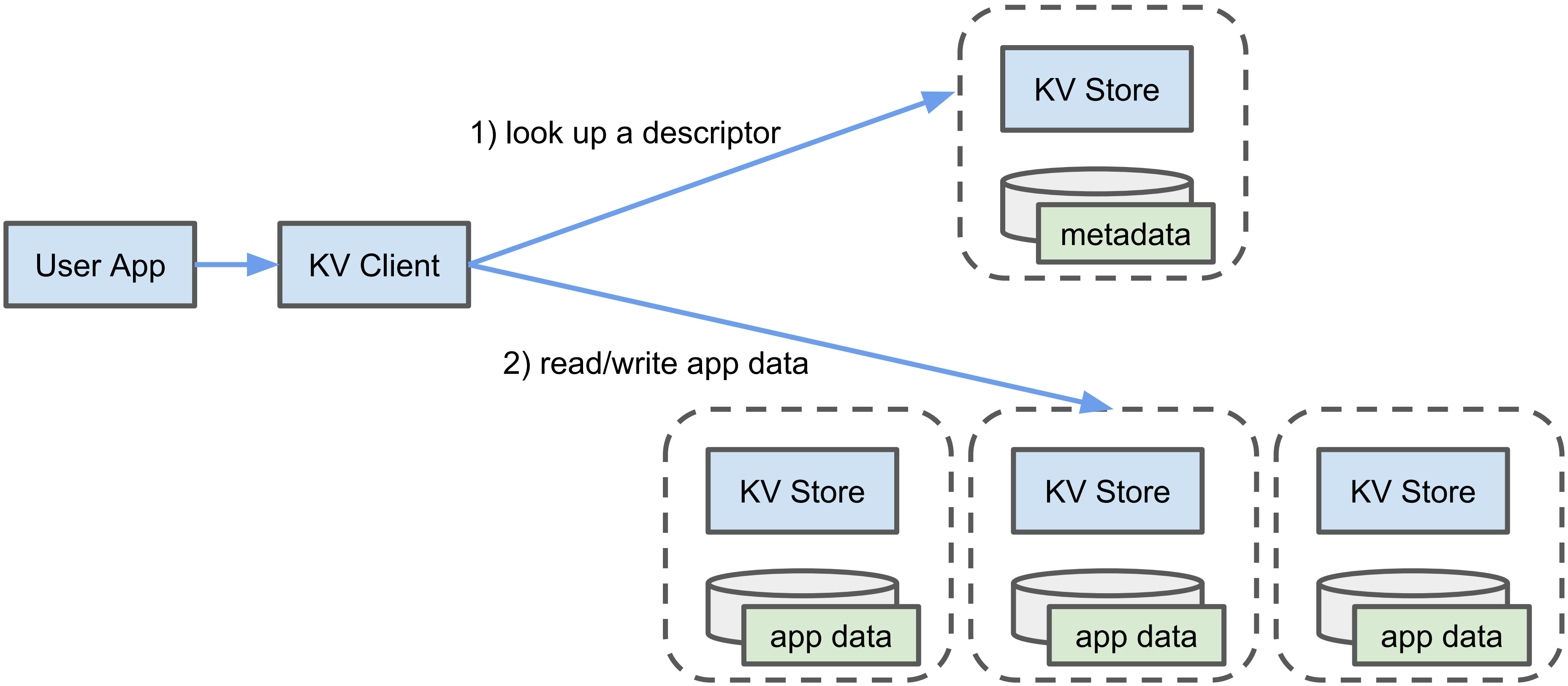
- Centralized master managing all bucket metadata
  - Need to special code/operation for the master
  - Won’t scale
- P2P bucket metadata distribution
  - Each node periodically gossips info with random other nodes
  - Hard to predict the reliable behavior
- **Special bucket for holding bucket metadata**
  - **Read/update bucket metadata like other normal keys/values**

Metadata Bucket and Bucket Descriptors

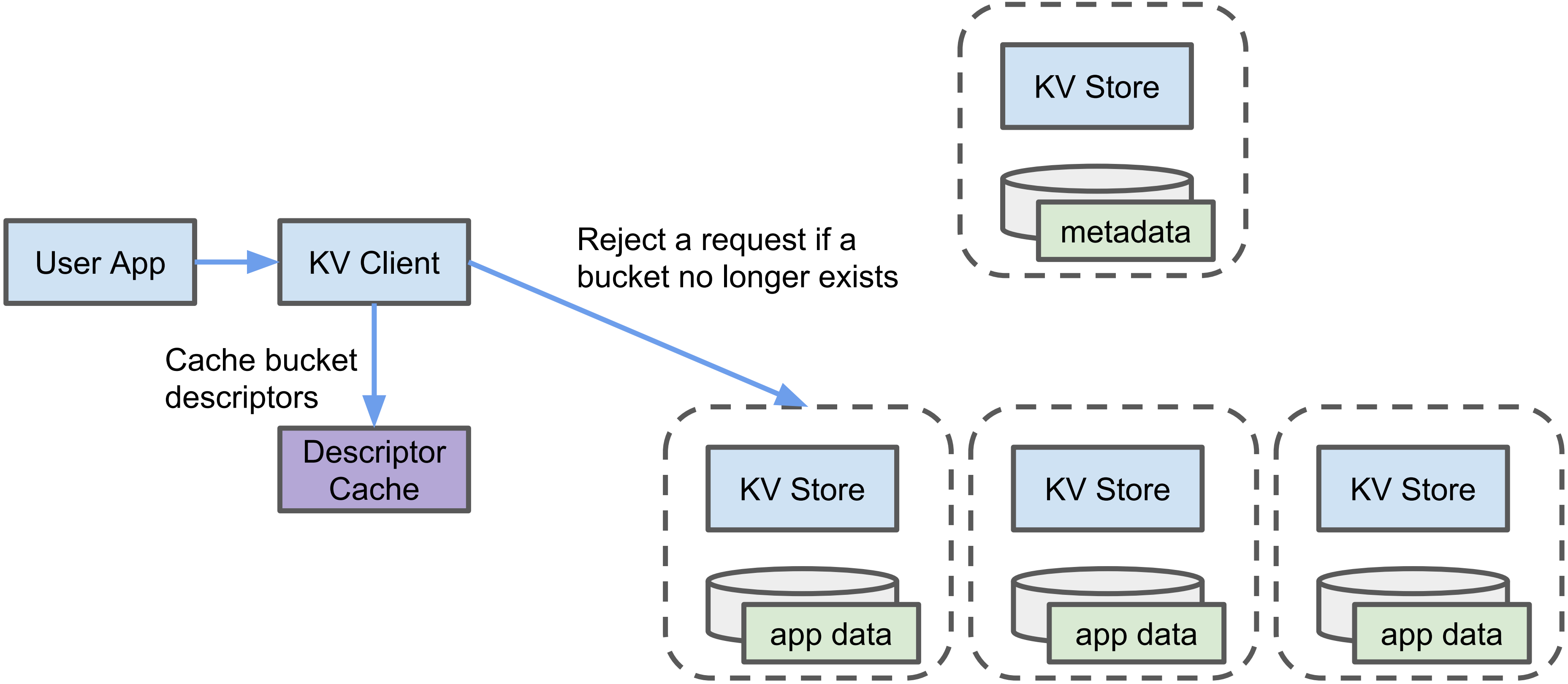
- Reserve **`[0x0000, 0x0001)`** for system data
- Store a bucket descriptor at key **`0x00meta<bucket_end_key>`**
- Look up a descriptor of a bucket containing `K` by scanning the metadata bucket
  - Scan from `K.Next()` until descriptor data is found



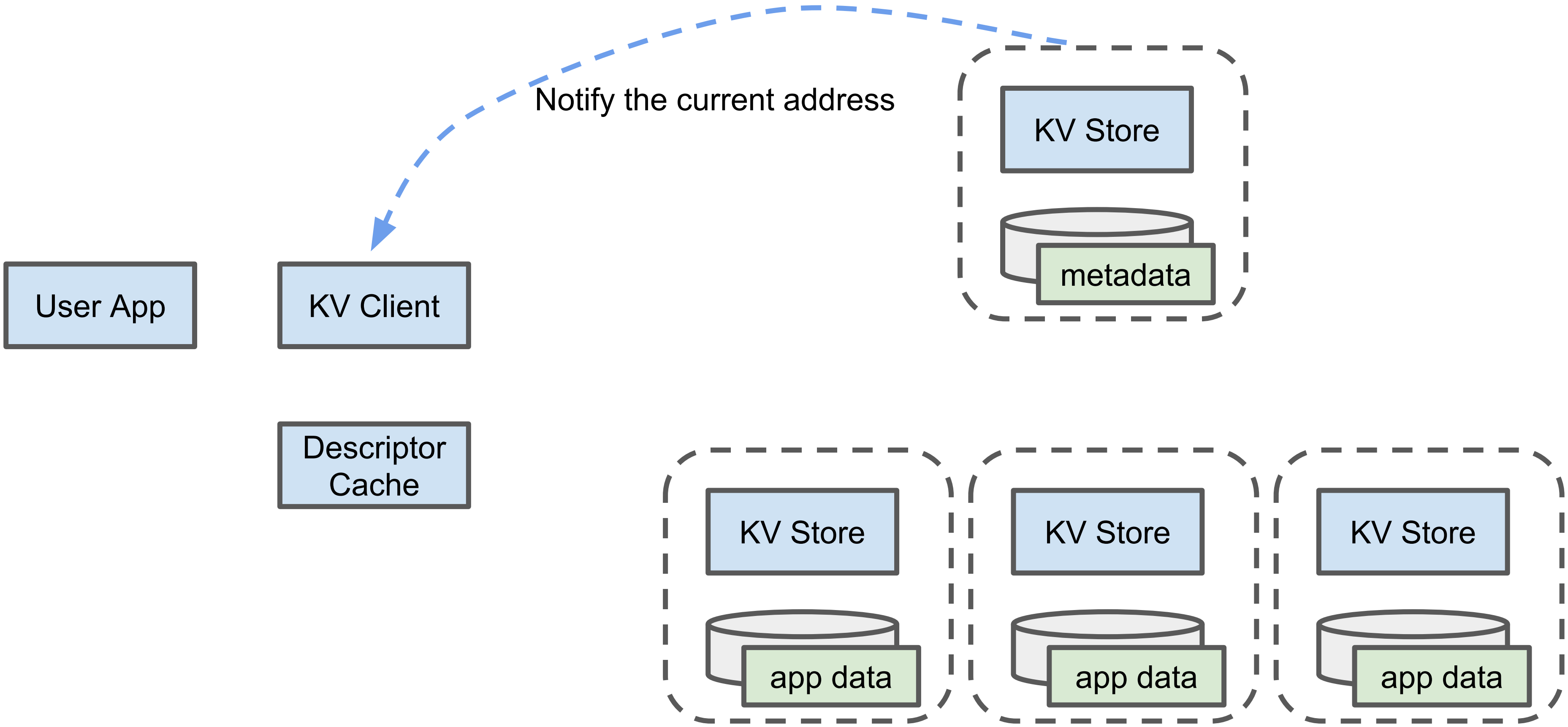
Command Execution Flow



Command Execution Flow



Command Execution Flow

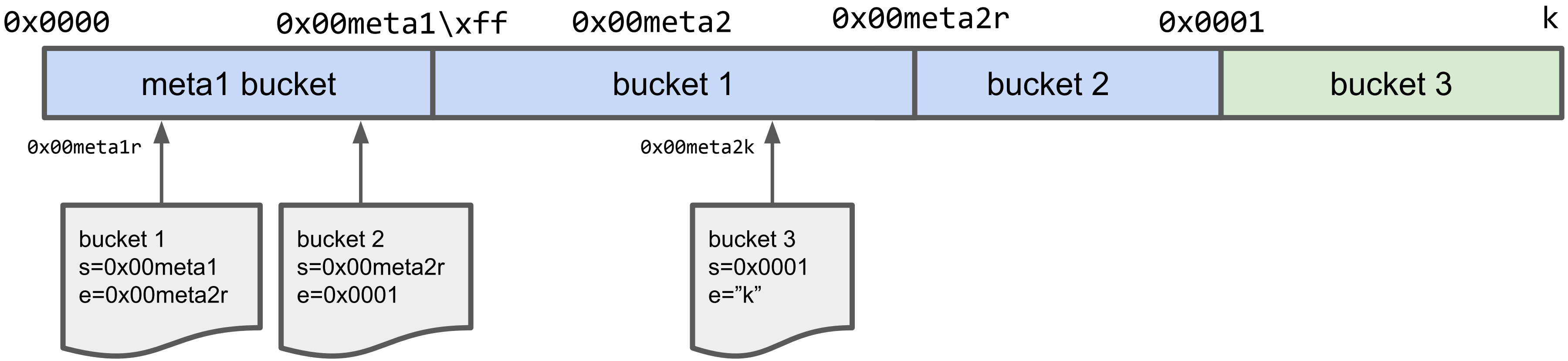


Limitation of the Single Metadata Bucket

- How much metadata do we need to support 1PB of logical data?
  - Size of each bucket: 64MB (= 2^26)
  - Number of buckets: 1PB / 64MB (= 2^(50 - 26) = 2^24)
  - Size of each bucket descriptor: 256B (= 2^8)
  - Total bucket descriptor size: 2^(24 +8) = **4GB**
- Not bad, but we can do better

2-Level Metadata Lookup

- Two metadata space: meta1 and meta2
  - `[meta1, meta1\xff)` stores descriptors of the meta2 space
  - `[meta2, meta2\xff)` stores descriptors of user data
  - Meta2 space can be split into multiple buckets
- First look up meta1 descriptor and then meta2

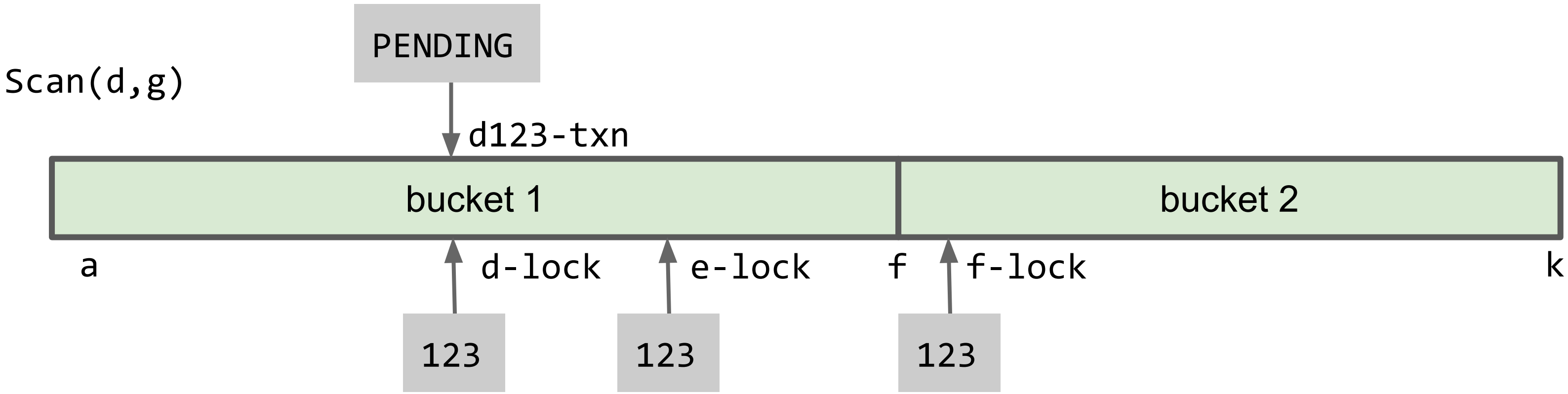


Scan Requests Across Multiple Buckets

- Two phase commit
  - 1) Acquire a lock from every accessed key
  - 2) Process read/write
  - 3) Release the locks
- Wound-wait deadlock prevention
  - When conflict happens, a lower-priority txn will be restarted (and blocked) by a higher-priority txn

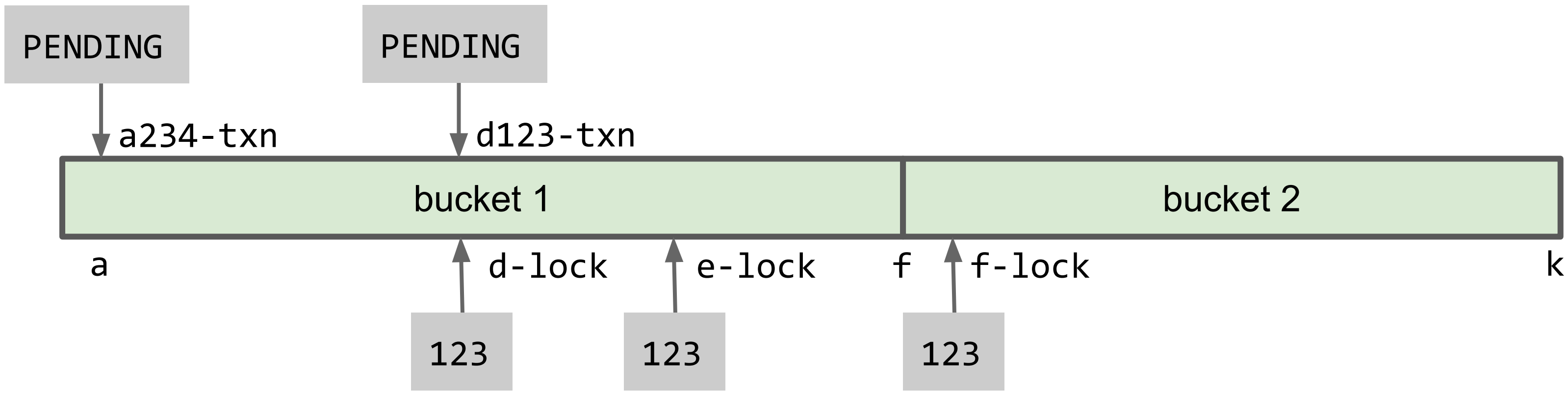
Implementing Two-Phase Commit on KV Store

- Txn record:
  - Key: txn ID (e.g., <first_accessed_key> + <random_number> + “-txn”)
  - Value: state (`PENDING`, `COMMITTED`, `ABORTED`)
- Lock record:
  - Key: <accessed_key> + “-lock”
  - Value: txn ID



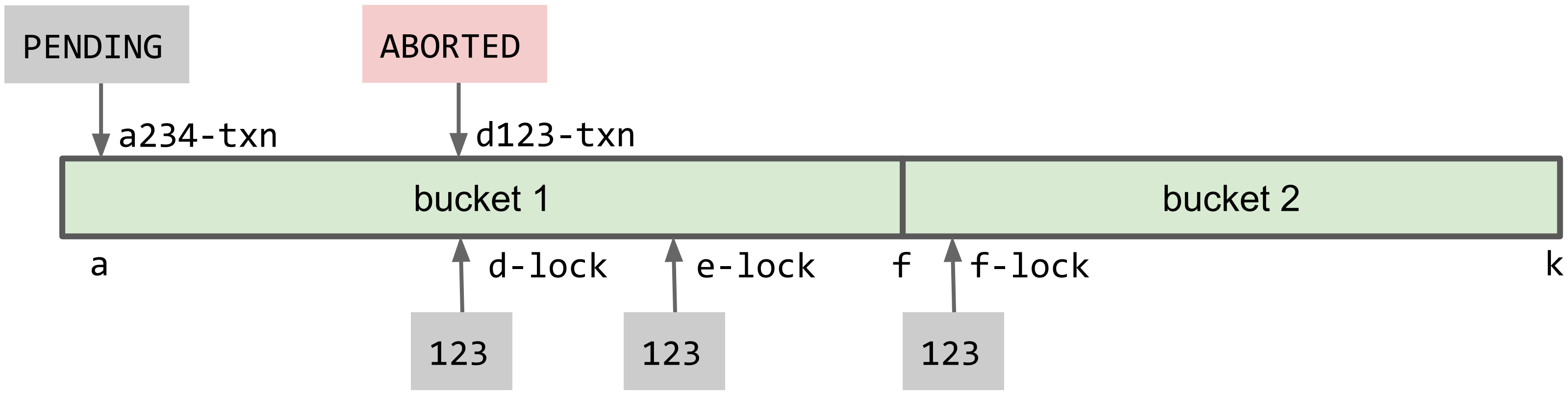
Resolving Conflicts - Case 1 -

- Higher-priority txn *T2* force-restarts lower-priority txn *T1*
  - 1) *T2* updates *T1*' txn record to `ABORTED`
  - 2) *T2* override the lock records with its txn ID
  - 3) *T1* restarts itself when seeing the `ABORTED` txn state



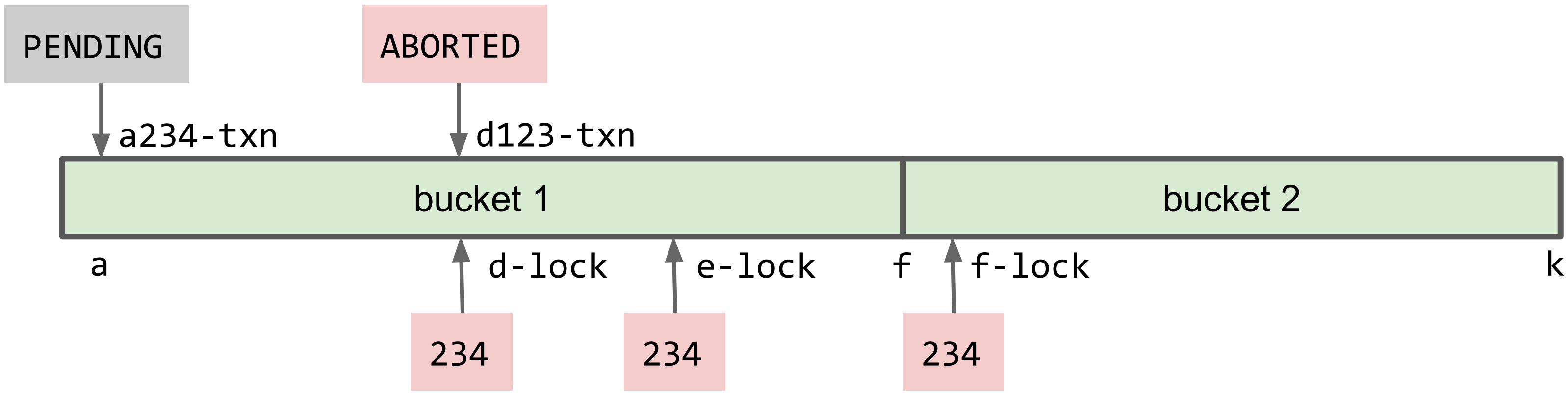
Resolving Conflicts - Case 1 -

- Higher-priority txn *T2* force-restarts lower-priority txn *T1*
  - 1) *T2* updates *T1*' txn record to `ABORTED`
  - 2) *T2* override the lock records with its txn ID
  - 3) *T1* restarts itself when seeing the `ABORTED` txn state



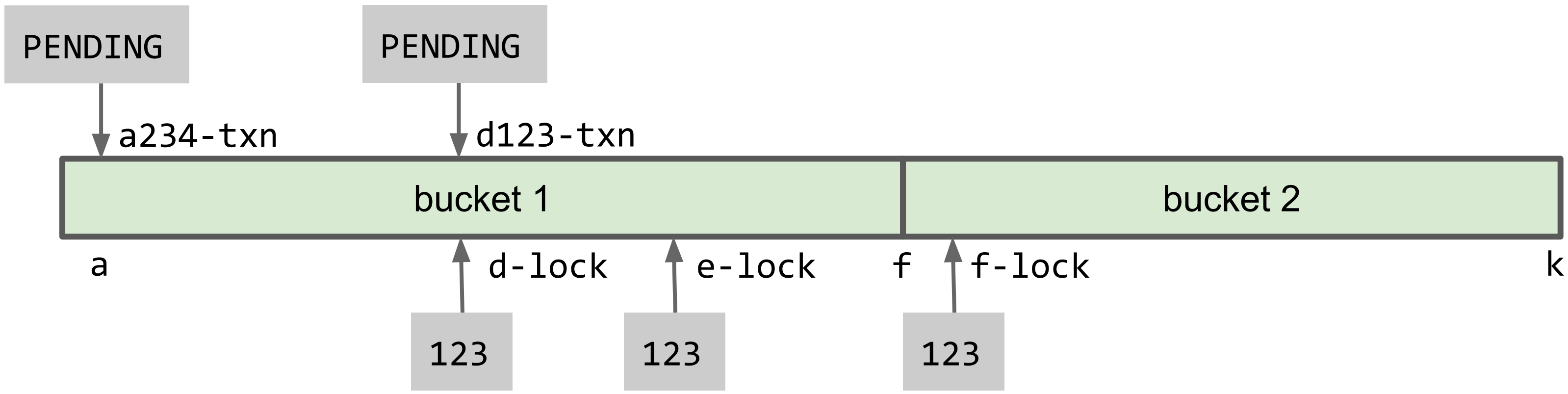
Resolving Conflicts - Case 1 -

- Higher-priority txn *T2* force-restarts lower-priority txn *T1*
  - 1) *T2* updates *T1*' txn record to `ABORTED`
  - 2) *T2* override the lock records with its txn ID
  - 3) *T1* restarts itself when seeing the `ABORTED` txn state



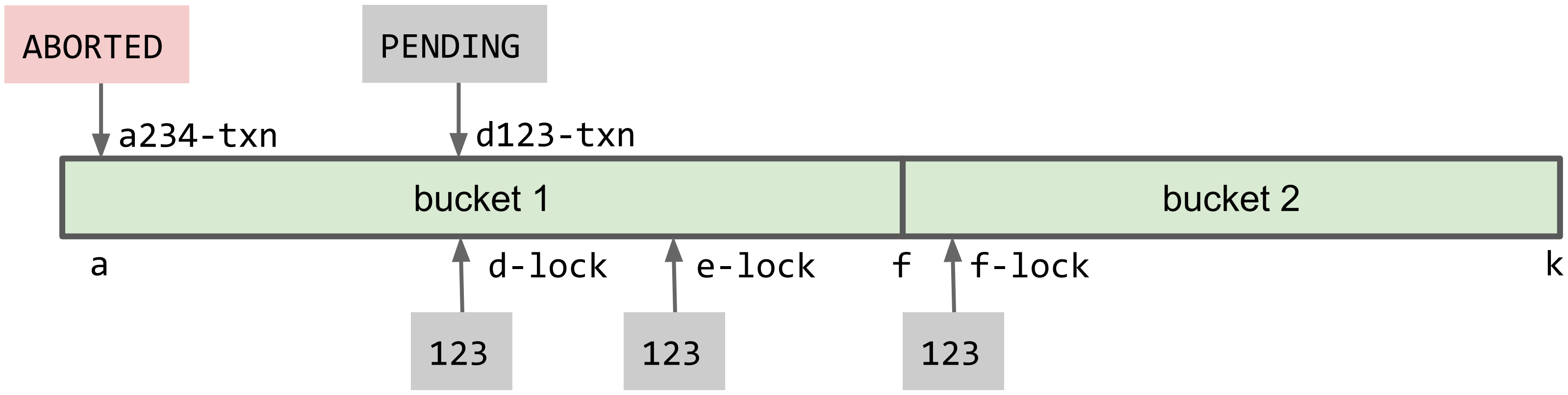
Resolving Conflicts - Case 2 -

- Lower-priority txn *T2* restarts itself when a lock record created by higher priority txn *T1* is found
  - 1) *T2* attempts to update the txn state of *T1* to `ABORTED`
  - 2) *T2* finds *T1*'s priority is higher than itself and restarts.



Resolving Conflicts - Case 2 -

- Lower-priority txn *T2* restarts itself when a lock record created by higher priority txn *T1* is found
  - 1) *T2* attempts to update the txn state of *T1* to `ABORTED`
  - 2) *T2* finds *T1*'s priority is higher than itself and restarts.



Skipped Topics

- Dynamic bucket split/merge
- Bootstrap
- Discovery
- Multiraft
- Recovery from client crashes
- Idempotence guarantee
- …

```
split(bucket, key) {
  txn(session -> {
      newBucket = new Bucket();
      newBucket.startKey = bucket.StartKEy
      newBucket.endKey = key;
      session.put(meta(newBucket), newBucket);

      bucket.endKey = key;
      session.put(meta(bucket), bucket);

      session.commitWithTrigger(splitTrigger);
  });
}

splitTrigger() {
   // update internal bucket map
}
```

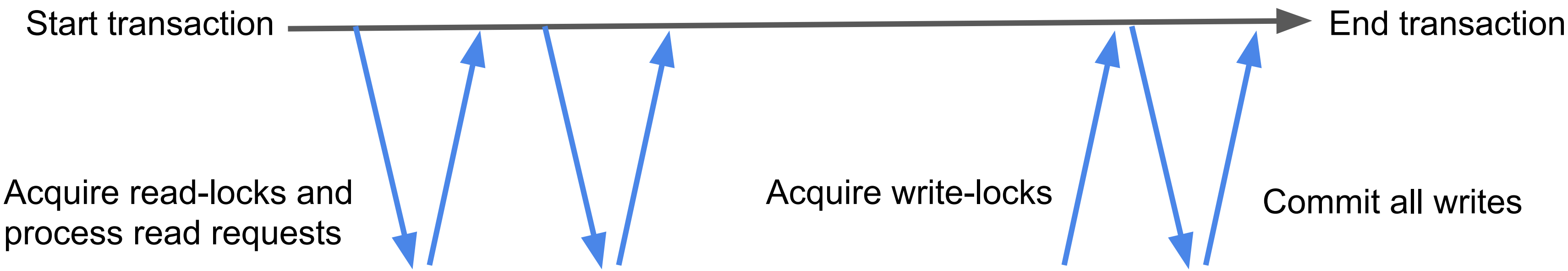
Step 4
Support ACID Transactions

Goal

- Support read-write transactions
- Support read-only transactions
- Avoid acquiring a pessimistic lock whenever possible

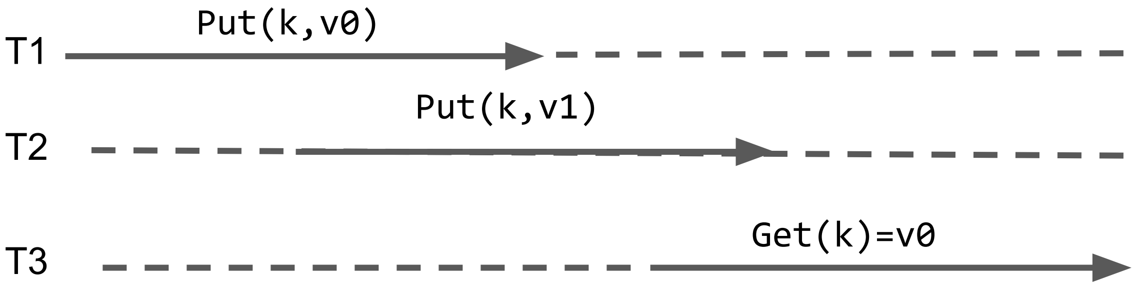
Read-Write Transactions

- Two phase commit with write buffering
  - 1) Process read requests with read-locks
  - 2) Commit with buffered write requests
- Serializable isolation level



Read-Only Transactions

- Snapshot isolation
  - `Get()` with timestamp *t* reads the latest value written before *t*
  - Reduced isolation level approach between `READ COMMITTED` and `REPEATABLE READ`
- Need Multi Version Concurrency Control (MVCC)
  - Assign a timestamp to each key-value pair
- Need to serve read requests at *t* only if no future write will happen before *t*



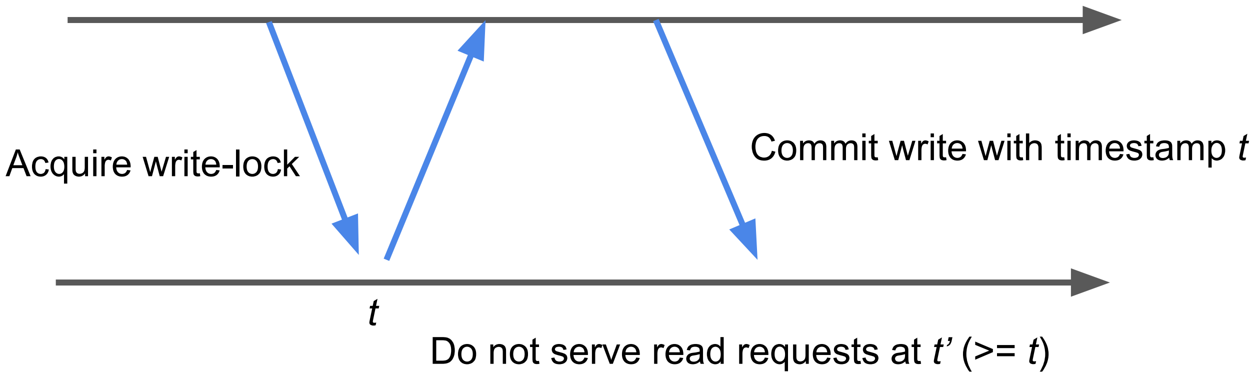
Data Versioning in KV Store

Key	Value
<key>	Metadata
<key>_<timestamp_n>	Value of version N
<key>_<timestamp_n-1>	Value of version N-1
...	...
<key>_<timestamp_0>	Value of version 0

- Txn ID if a write lock is held (i.e., write intent)
- Timestamp of most recent version
- ...

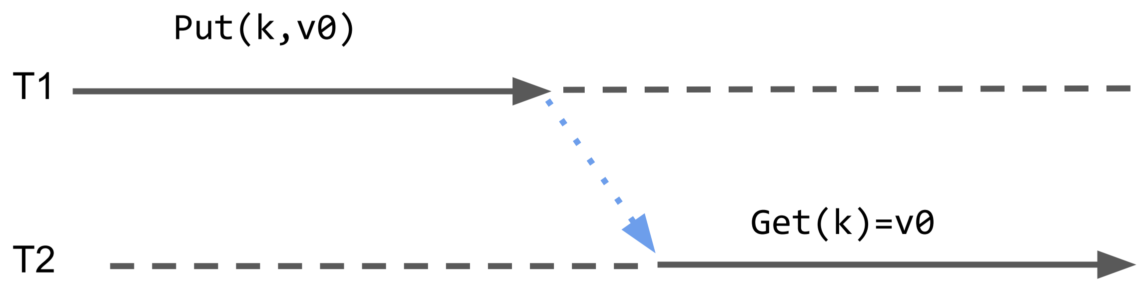
Choosing a Commit Timestamp

- Exchange timestamp when acquiring write-lock
- Set the commit time to max(write-lock acquisition timestamp)
- Serve read request for snapshot at *t* only when *t* is less than the write lock acquisition timestamp



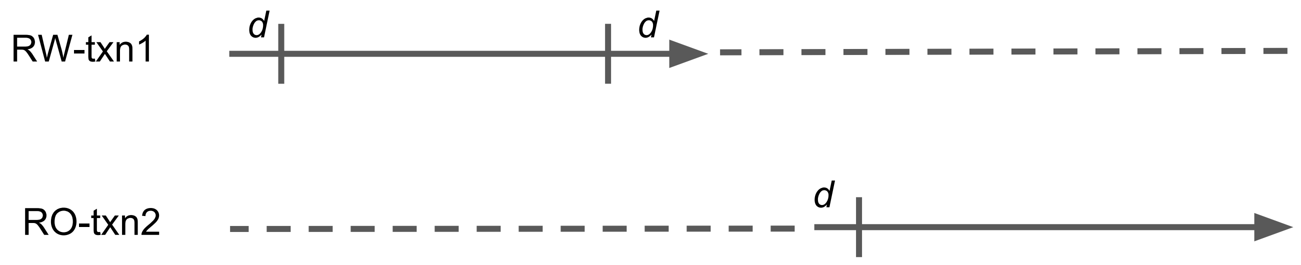
Guaranteeing Linearizability

- Definition:
  - Serializable + serialization order satisfying external consistency
  - If *T1* commits before *T2* starts, *T1*'s commit timestamp is smaller than *T2*'s
- Timestamps must be chosen carefully to achieve linearizability
  - Commit timestamps for RW transactions
  - Snapshot timestamp for RO transactions



Choosing Timestamps with Linearizability Guarantee

- Let the max clock skew be *d*
- RW transaction
  - Use (current time + *d*) as the lower bound of the commit timestamp
  - Release write-locks after (commit timestamp + *d*)
- RO transaction
  - Set the snapshot timestamp to (current time + *d*)



Skipped Topics

- Unbounded clock skew
  - Spanner relies on an atomic clock and GPS receivers to bound the clock skew
- Optimization for single-bucket read/write
  - No need to have two phase commits
- Other transaction models
  - E.g.) No write buffering
  - E.g.) Serializable Snapshot Isolation (read and write at a given timestamp)
- Garbage collection of MVCC data

Step 5
Support SQL

Encoding Tables in KV Store

```
CREATE TABLE merchants (
  id       INT PRIMARY KEY,
  token    STRING
  fee_rate FLOAT
)
INSERT INTO merchants VALUES (10, “MR1XAW”, 2.75)
```

Key	Value
/merchants/10/token	MR1XAW
/merchants/10/fee_rate	2.75

Summary

Summary of This Talk

- Design of a highly-scalable highly-available transactional data store
    - Log-structured merge tree (e.g., LevelDB, RocksDB)
    - Raft consensus algorithm
    - Bucket lookup and two-phase commit
    - Linearizability, snapshot isolation, MVCC
    - ...
- Many undiscussed issues
    - Performance
    - Failure detection and bucket relocation
    - Dynamic bucket rebalancing
    - Clock skew with no hardware support
    - Scheme change, secondary index
    - ...

References

- Level DB
  - https://github.com/google/leveldb
- RocksDB
  - http://rocksdb.org/
- Raft
  - Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm”
  - Diego will come to Square on Nov 3rd
- Spanner
  - Jeff Dean, Sanjay Ghemawat, et al. “Spanner: Google’s Globally-Distributed Database”
  - Dahlia Malkhi, Jean-Philippe Martin. “Spanner’s Concurrency Control”
- Cockroach DB
  - https://github.com/cockroachdb/cockroach/blob/master/docs/design.md
  - http://www.cockroachlabs.com/blog/