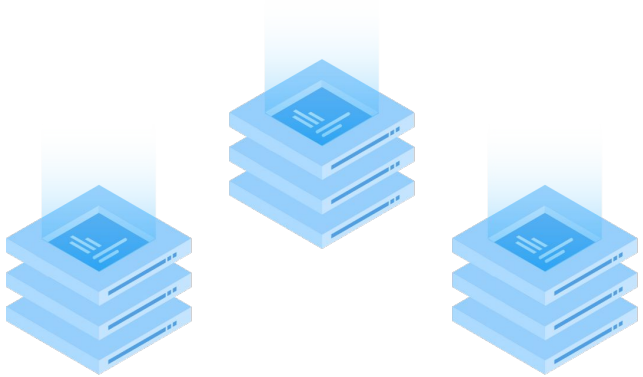
Lesson 7: transactions

Presented by Nick Cameron

PingCAP TiDB



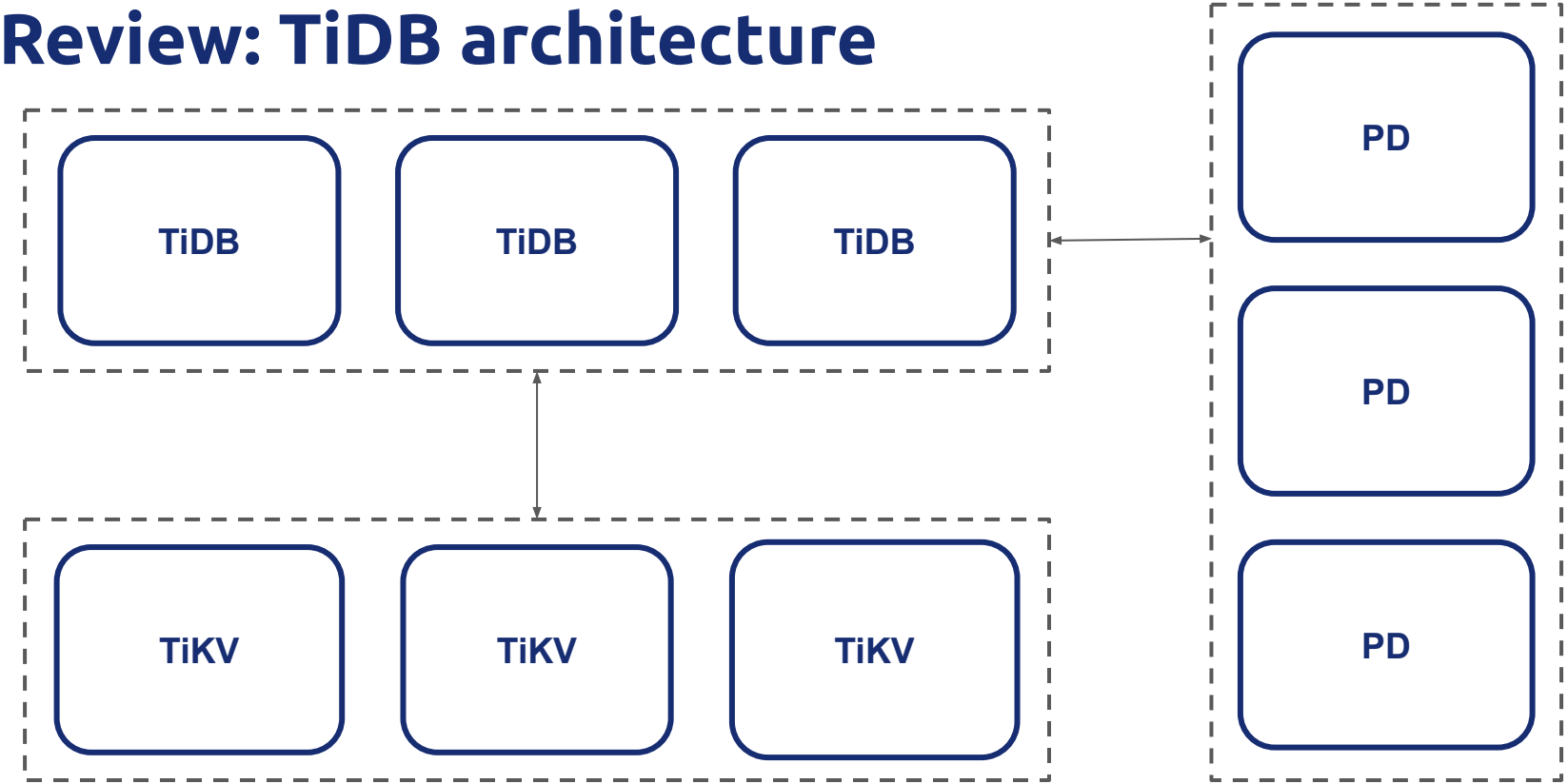
Before we begin

- Context: TiDB Architecure
- Goal : Introduce TiDB transactions
- Outline:
  - Architecture review
  - Consistency
  - Transaction protocol
  - MVCC
  - Performance considerations

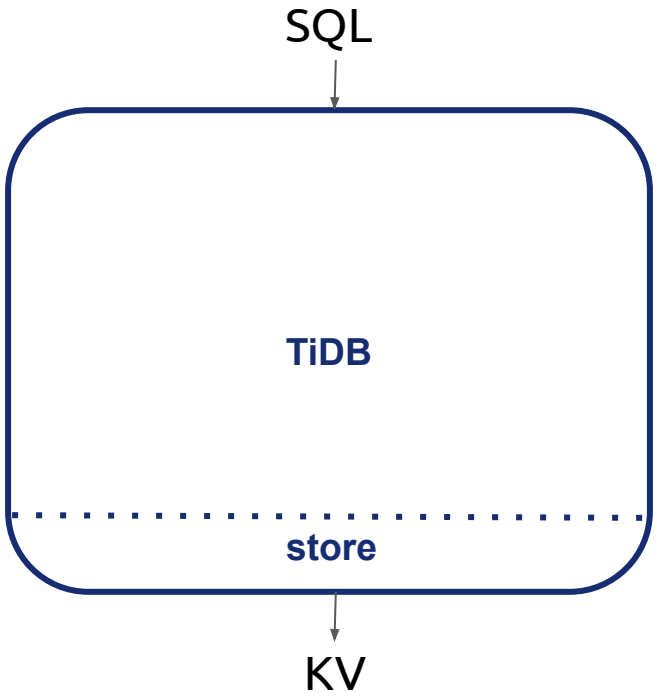
Part I: architecture review

- Refresher on TiDB design
- Subtopics
  - TiDB and TiKV
  - TiKV layers
  - Regions

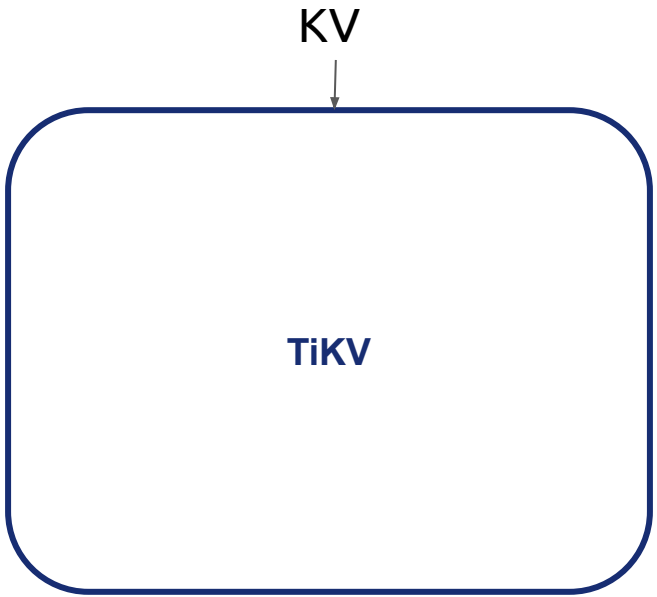
Review: TiDB architecture



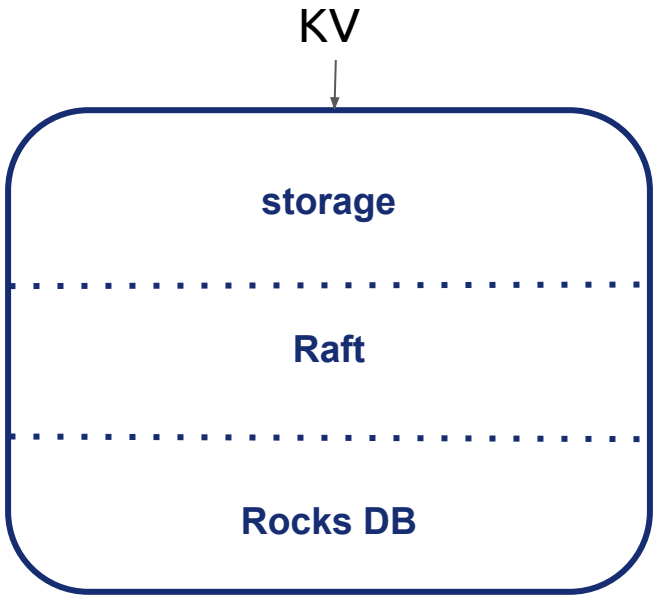
Review: TiDB architecture



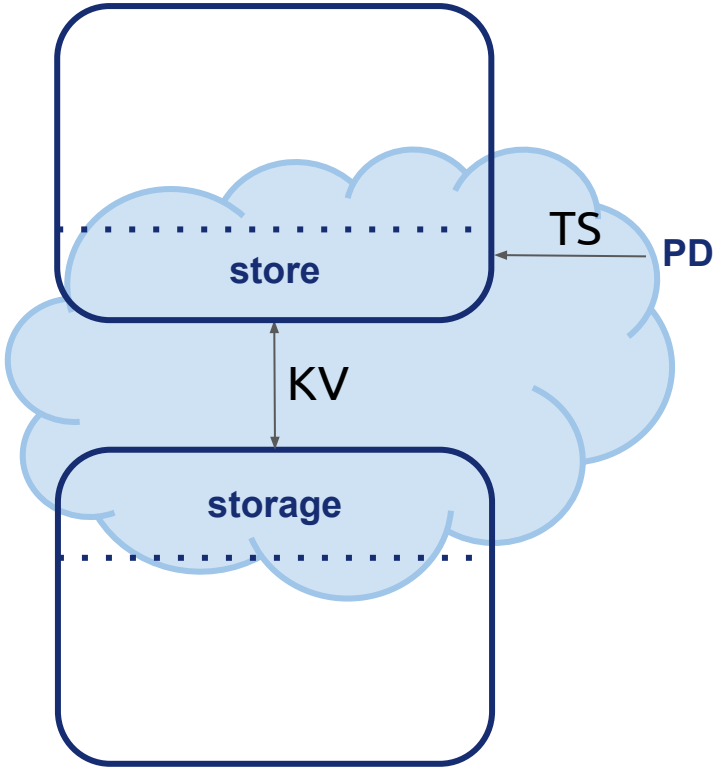
Review: TiDB architecture



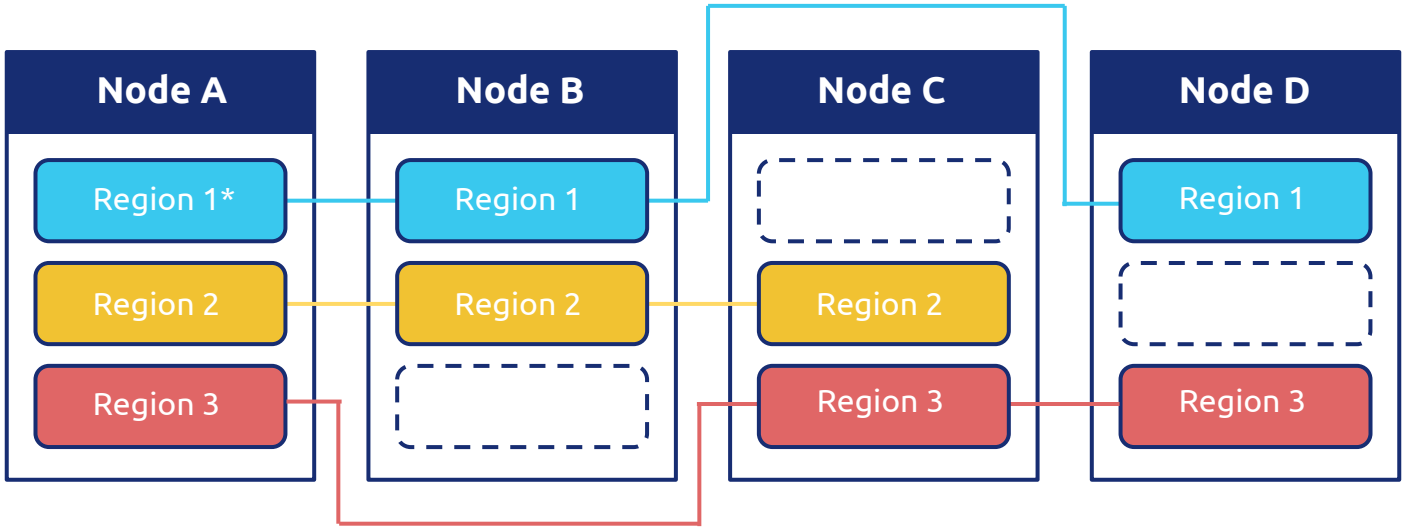
Review: TiDB architecture



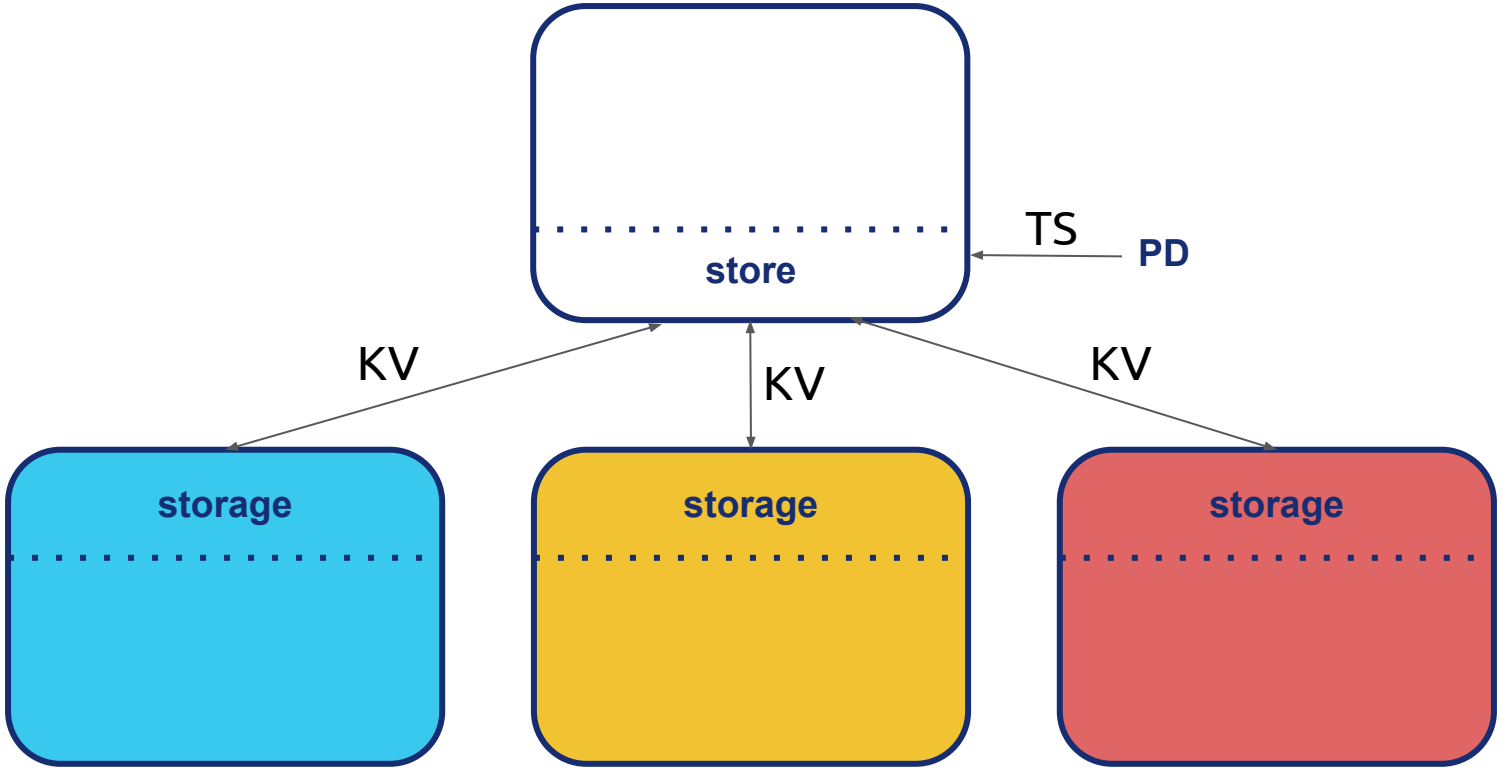
Review: TiDB architecture



Review: regions



Review: regions



Part II: consistency

- TiDBs transactional guarantees
- Subtopics
  - ACID
  - Snapshot isolation

ACID

- Atomicity
- Consistency
- Isolation
- Durability

Snapshot isolation

- Consistent snapshot at a specific time
- Repeatable reads

Snapshot isolation

- > read committed
- Can’t read uncommitted data

Snapshot isolation

- < serializability
- Write skew

Part III: transaction protocol

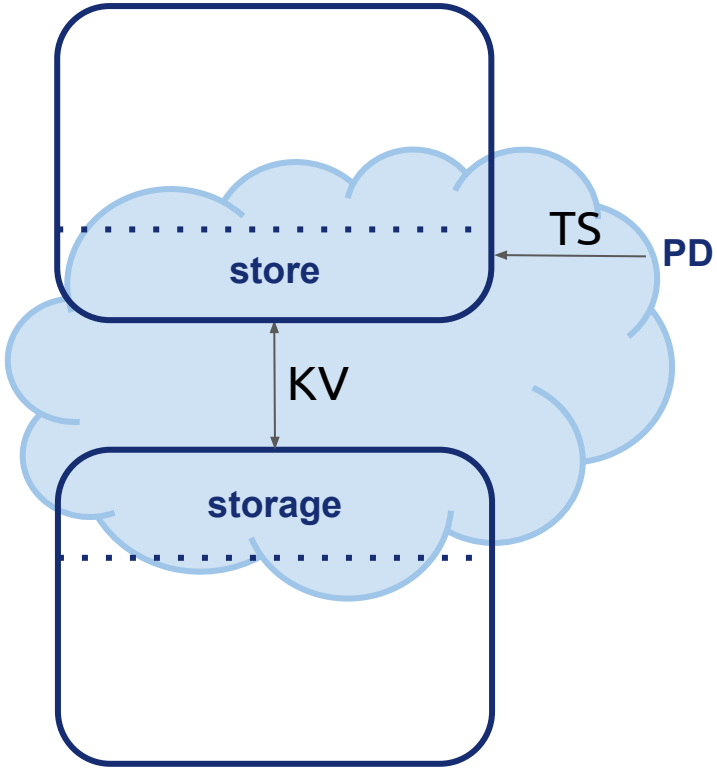
- High-level view of TiDB transactions
- Subtopics
  - Percolator
  - Timestamps
  - Two-phase commit
  - Ensuring reliability

Transaction protocol

- Based on Percolator (Google)
- Collaborative

Transaction protocol

- Based on Percolator (Google)
- Collaborative



Timestamps

- **`uint64`** from PD
- combination of wall-clock time and a counter
- *unique, monotonic and linear*

Timestamps

- startTS: taken when transaction begins
  - transaction sees a snapshot of database at startTS
- commitTS: taken after all prewrites succeed, before sending commit message
  - writes in a transaction are externally visible after commitTS

Two-phase commit

- Phase 1: prewrite
  - Get startTS from PD
  - Transaction is built in TiDB
  - Prewrite messages sent to TiKV nodes
  - TiKV acquires locks and writes data
  - Returns success or failure to TiDB

Two-phase commit

- Phase 2: commit
  - TiDB collects responses to prewrite messages
  - If **all** prewrites succeed, then get commitTS from PD, send commit messages
  - TiKV commits the earlier prewrites
  - If commit succeeds, then return success to the client

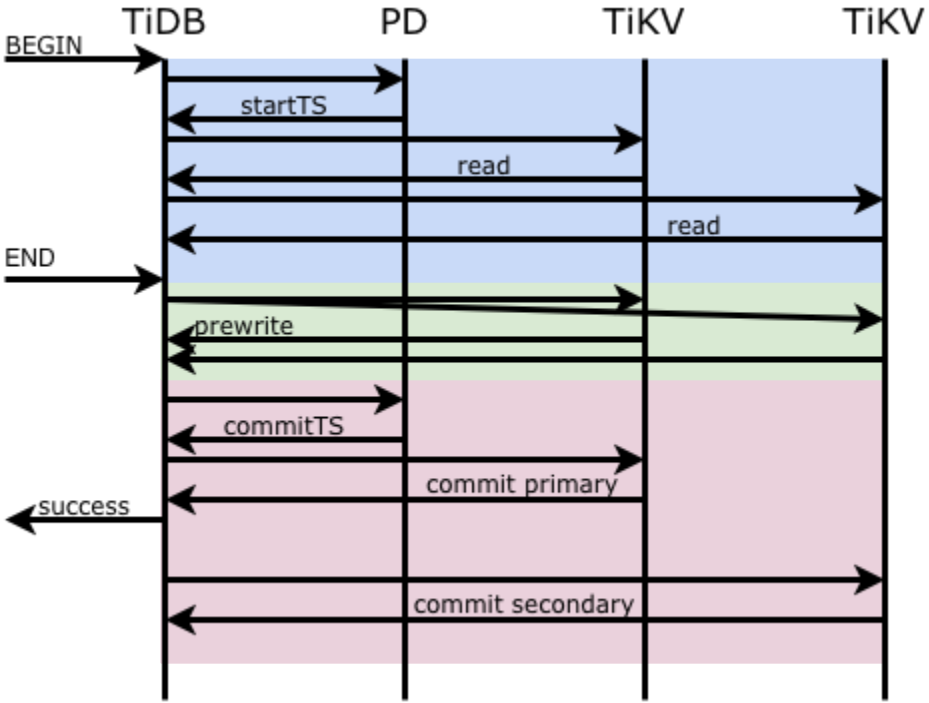
Two-phase commit

- Phase 2: commit
    - If **all** prewrites succeed, then send commit messages
        - If any prewrites fail, then *rollback*
        - Send rollback message to each TiKV node
        - Prewrites are removed

Two-phase commit

- Phase 2: commit
  - If **all** prewrites succeed, then send commit messages
    - One key is selected as *primary*
    - Other locks refer to the primary lock
    - Only the primary lock must be committed before transaction is considered committed
    - Other locks can be committed asynchronously

Two-phase commit



Two-phase commit

- Reliability
  - Lost or duplicated messages
  - Lost TiKV node
  - Lost TiDB node

Two-phase commit

- Abandoned locks
  - Find transaction state from primary lock
  - Commit or prewrite (i.e., resolve) the lock

Part IV: MVCC

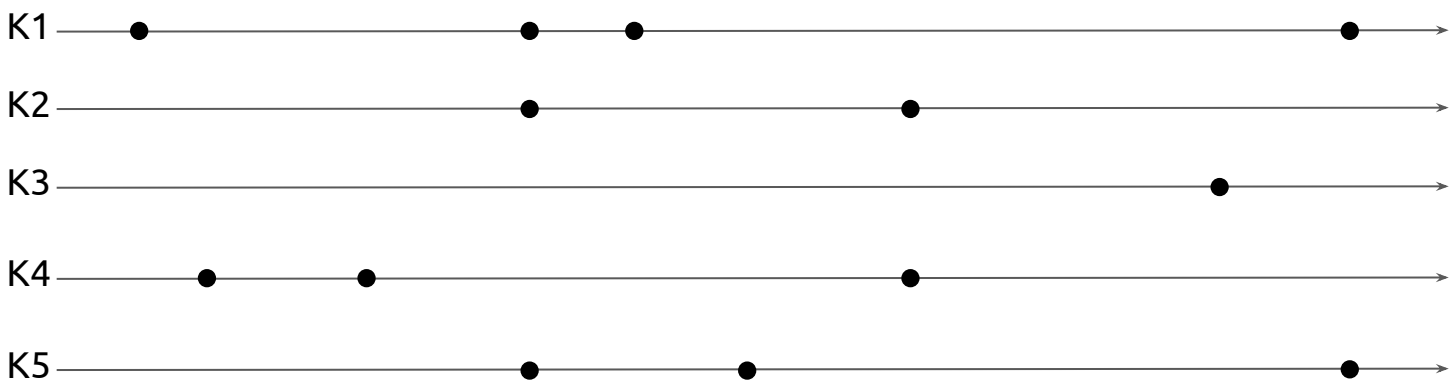
- Multi-version concurrency control
- Subtopics
  - Multi-versions
  - Representation
  - Action implementation

MVCC: multi-versions

- Don’t only store the ‘current’ value, but entire history of values
- Store history for each key, with a value for each commitTS

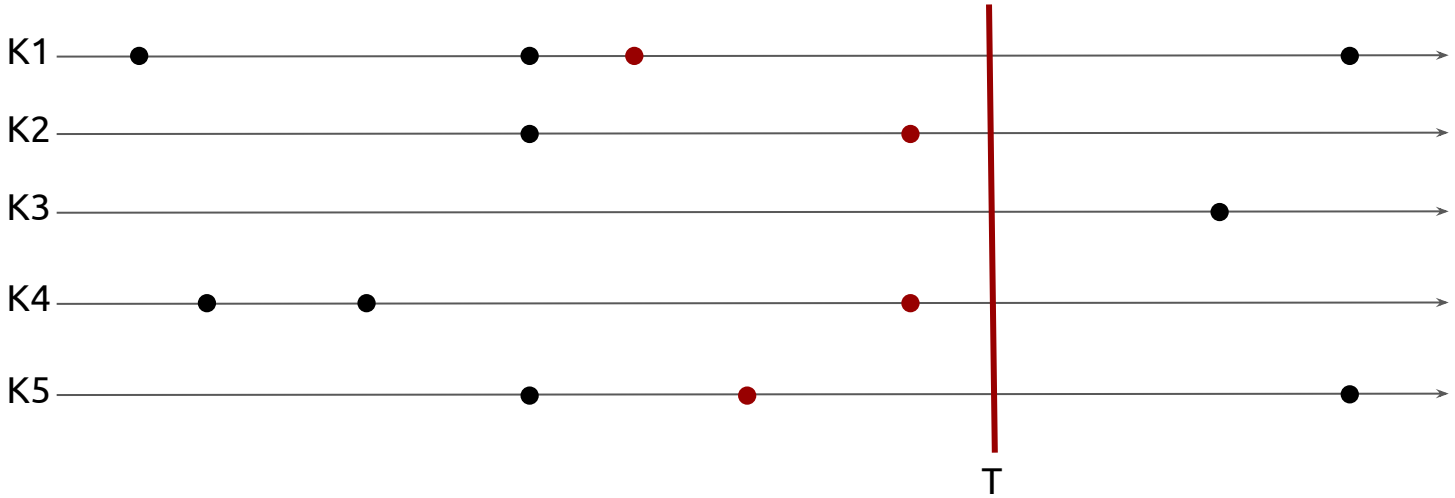
MVCC: multi-versions

- Reading a snapshot: reads most recent version prior to the snapshot time



MVCC: multi-versions

- Reading a snapshot: reads most recent version prior to the snapshot time



MVCC: representation

- Underlying storage is just keys and values
- Need to encode the MVCC concepts (versions, locks, etc.)

MVCC: representation

- Column families (CFs)
  - default (values)
  - write (committed metadata)
  - lock
- Essentially three key-value stores + atomicity

MVCC: representation

- Default CF
  - key, startTS -> value
  - written in prewrite

MVCC: representation

- Write CF
  - key, commitTS -> writeType, startTS
  - written on commit or rollback (using startTS)
  - writeType: put, delete, …
  - commitTS in descending order

MVCC: representation

- Lock CF
  - key -> lock
  - Note: only one lock per key (no TS)
  - lock:
    - lockType
    - primaryKey
    - startTS
    - TTL
    - ...

MVCC: actions

- Prewrite
  - Check write CF for a non-rollback entry more recent than startTS (write conflict)
  - Check and write lock
  - Write value in default CF

MVCC: actions

- Commit
  - Check Lock
  - Write `key, commitTS -> type, startTS` to write CF
  - Delete lock

MVCC: actions

- Rollback
  - Delete `key, startTS` from default CF
  - Write `key, startTS -> rollback, startTS` to write CF
  - Delete lock

MVCC: actions

- Read (key, ts)
  - scan write CF for key,*
  - keep scanning untill we find a non-rollback entry where commitTS < ts
  - write entry gives startTS, use `key, startTS` in default CF to find value

Part V: performance considerations

- How we can optimise the transaction system
- Subtopics
  - Constrained resources
  - Optimisations

Performance considerations

- Reads, uncontested writes are common
- Contested writes are rarer
- Recovery actions are very rare

Performance: resources

- Network I/O (PD, TiDB, TiKV)
- Durable writes to the Raft store (network + disk I/O)

- CPU
- RAM

- Locks and latches

Performance: pessimistic transactions

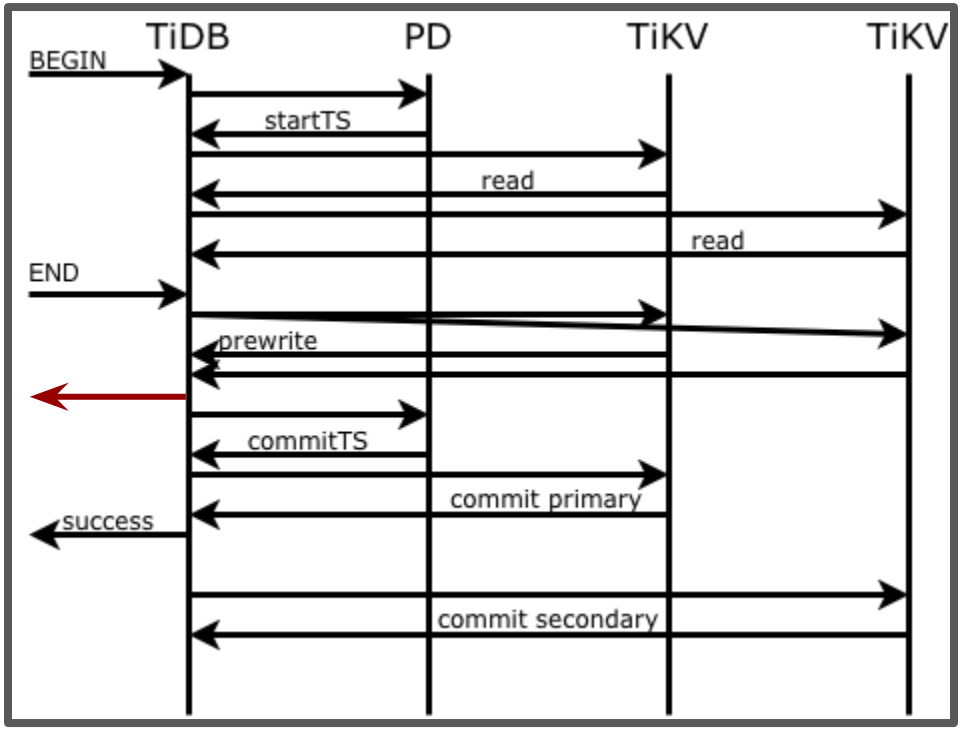
- TiDB can lock keys during the transaction
- Prewrite to these keys should always succeed
- Allows partial retries

- But … deadlock

Performance: async commit

- If commit is guaranteed to succeed, why do we have to wait for it before returning success?

Performance: async commit



Performance: async commit

- We *can* return early
- But …
  - Commit status is shared between locks
  - Recovery is more complex and slower
  - We need to keep track of all secondary keys -> increased memory use

Performance: implementation optimisations

- Pipelining
  - pessimistic transactions
  - if key is locked, write must succeed
  - don’t wait for Raft

Performance: implementation optimisations

- Scheduling and prioritisation
  - TiKV does a lot of waiting
  - Run many tasks in parallel
  - Optimise scheduling

Performance: implementation optimisations

- Batching
  - Batch reads
  - Batch multiple commands
  - Batch Raft actions

Performance: implementation optimisations

- Small values
  - Keep small values in the lock or write CF
  - Saves an indirection to lookup value in default CF

Performance: implementation optimisations

- Collapse rollbacks
  - When reading, need to skip rollback writes
  - Collapsing rollbacks together means one skip instead of n skips
  - But … some rollbacks must be preserved

Homework

Issue 8650

Score : 2000

Description :

TiKV uses MVCC to implement transactions. To check constraints, MvccTxn must perform several reads, we should be able to improve their performance.

Link: issues/8650

Issue 8651

Score : 4000

Description :

To detect deadlocks, TiKV has a central deadlock detector for the whole cluster. The Raft leader of the first region is the leader of the deadlock detector. However, the leadership is affected by region scheduling which may affect the performance of detecting deadlock. We can implement the leader election through PD rather than Raft and hopefully avoid these issues.

Link: issues/8651

Issue 8652

Score : 5000

Description :

The work pattern of lock CF in TiKV is insert one when prewrites and then remove it when commits and they are usually in the same memtable. However, RocksDB flushes all entries in the memtable to the disk which is wasteful for lock CF.

Link: issues/8652

Issue 8653

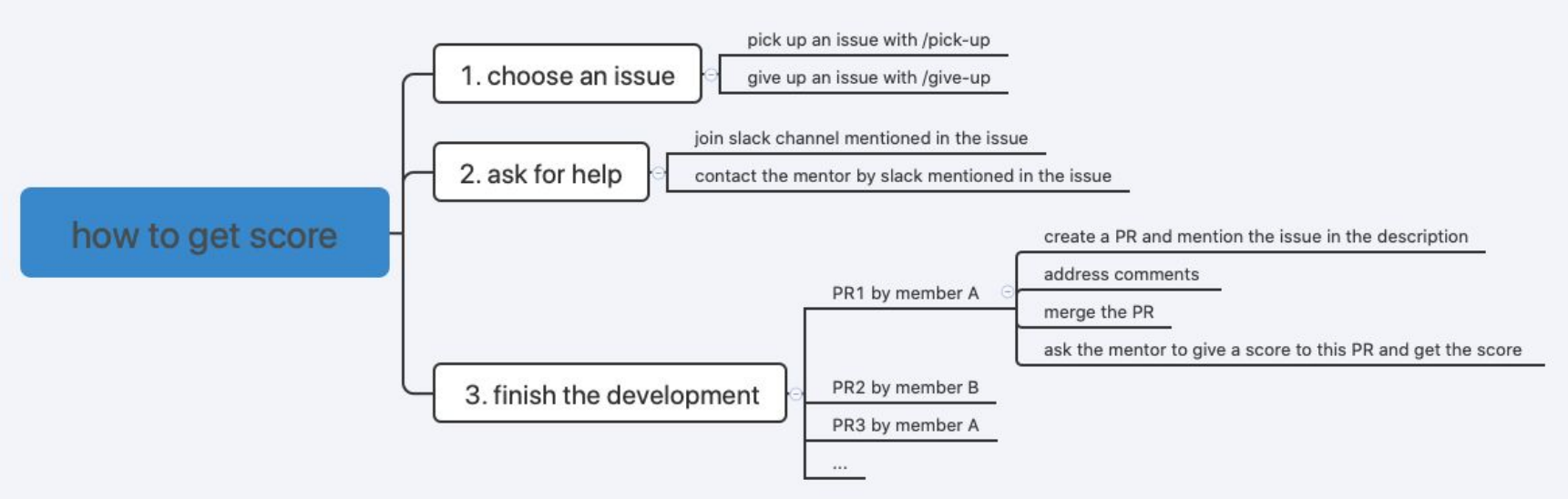
Score : 7000

Description :

TiKV writes locks to a memory lock table during prewrite to guarantee the correctness of async commit, and read operations need to check locks in it. The performance of the memory lock table is important, and we use crossbeam-skiplist as the memory lock table now. The task for you is to find the optimal data structures for it which should have great performance and low memory overhead.

Link: issues/8653

How to get score



Q&A

If you have any questions, please scan the QR code and join our wechat group. We also welcome suggestions and feedback from you.

Learn more

More courses:

- Scan the QR code
- Visit our website: http://university.pingcap.com

Thank you!