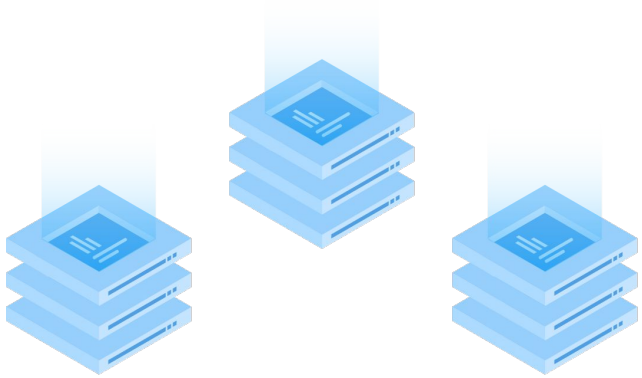
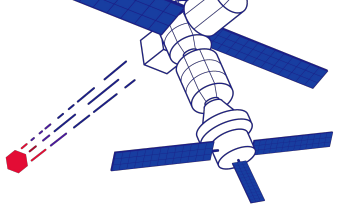
TiKV Internal

How to build a distributed transactional KV engine

PingCAP TiDB



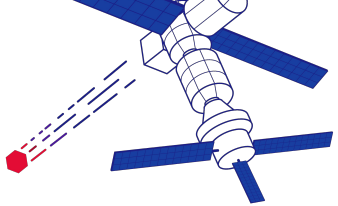
Agenda



1. TiKV aerial view
2. Service layer built with gRPC
3. HA component built with Raft
4. Transaction component built with Percolator

PingCAP

Agenda

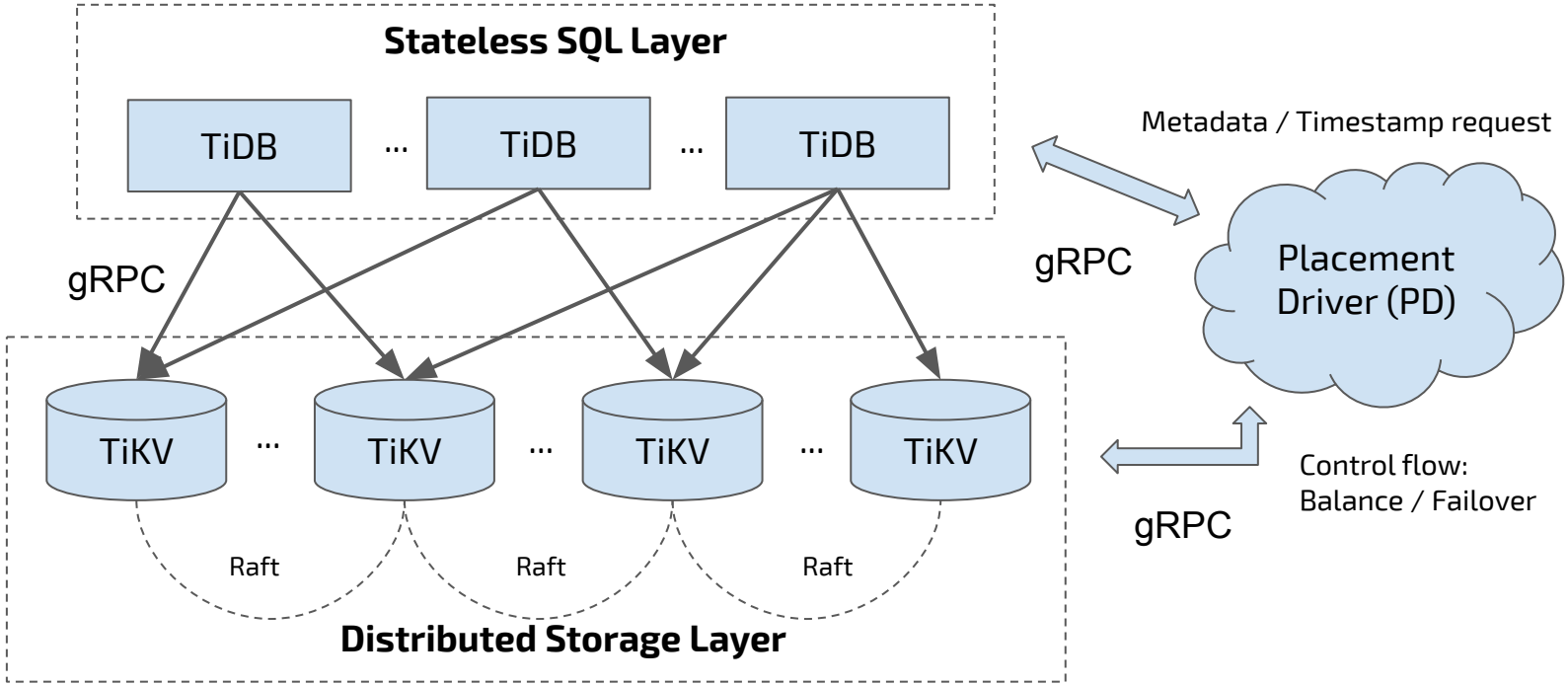


1. **TiKV aerial view**
2. Service layer built with gRPC
3. HA component built with Raft
4. Transaction component built with Percolator

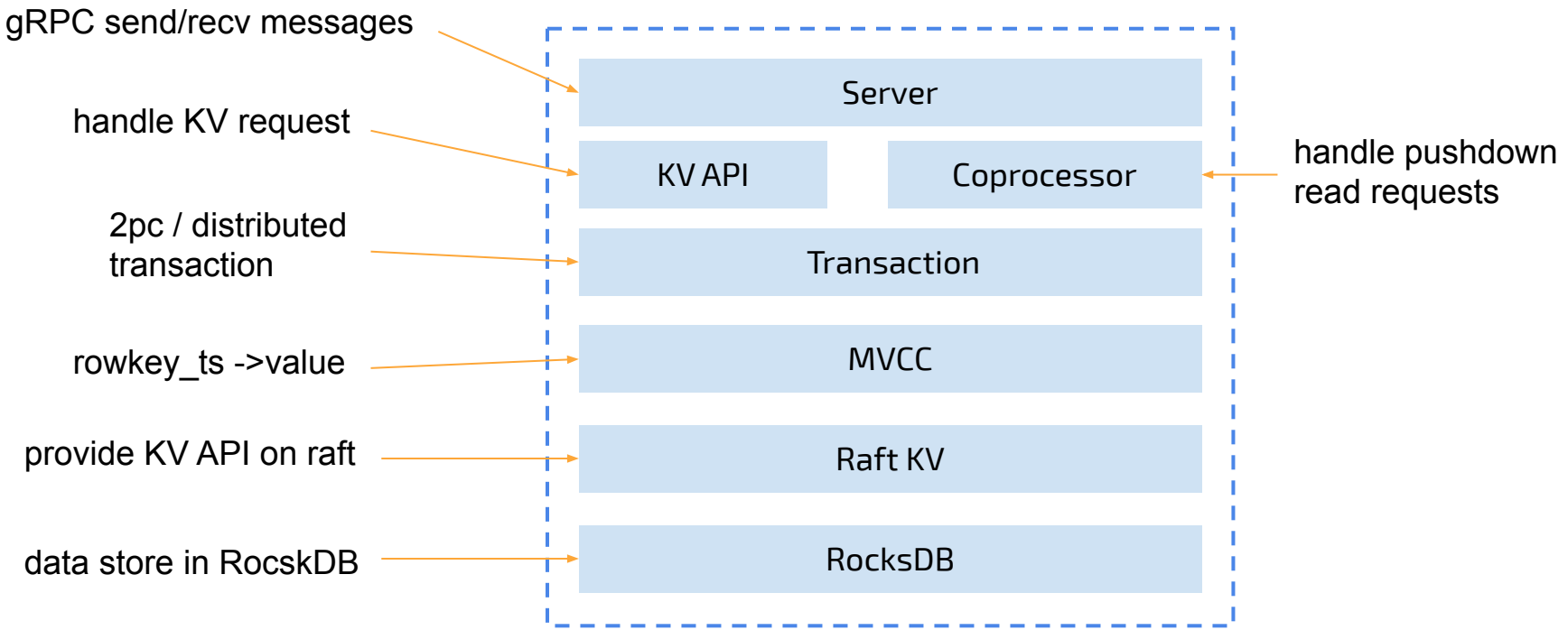
Concepts and Why

1. Why “distributed”
    a. No single point
    b. Easy to scale-out
    c. High availability
2. Why “transactional”
    a. Soul of databases

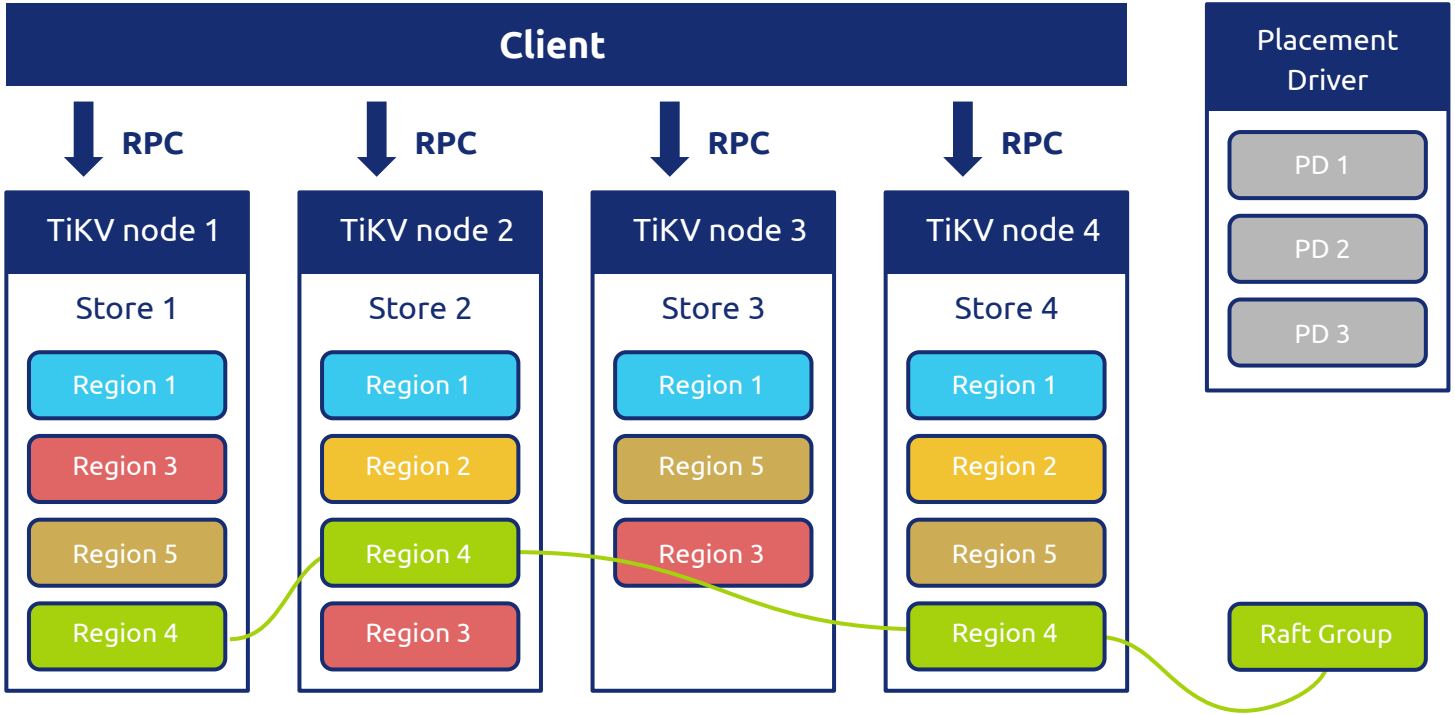
TiKV in TiDB Cluster



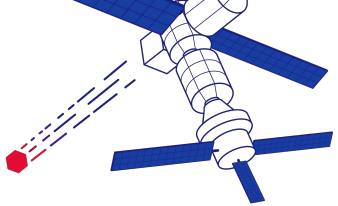
TiKV Layer Structure



Data Orgnization in TiKV



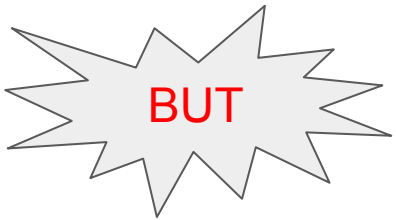
Agenda



1. TiKV aerial view
2. **Service layer built with gRPC**
3. HA component built with Raft
4. Transaction component built with Percolator

Why give up raw TCP

1. Best performance
2. Totally control about protocol
3. Free to choose network I/O model



Not friendly for clients

Why choose gRPC

1. Full features
    a. Unary, client streaming, server streaming and duplex streaming
2. Based on HTTP/2
    a. Easy to debug
    b. Built-in SSL support
3. Performance
    a. gRPC core has a nice network model (or I/O engine)
    b. gRPC coer has a nice API

Why grpc-rs instead of others

Why not callbacks? The gRPC API is much better than raw epoll.

```
while (1) {
    grpc_event ev = grpc_completion_queue_next(comp_queue);
    if (e.type == GRPC_OP_COMPLETE) {
        struct call_context *cc = e.tag;
        char *method = grpc_slice_to_c_string(cc->details.method);
        if (strcmp(method, "/Debug/unary_example_1") == 0) {
            hanle_unary_example_1(cc);
        }
    }
}

void handle_unary_example_1(struct call_context *cc) {
    if (cc->status == 0) {
        // Handle new registered call.
    } else if (cc->status == 1) {
        // Handle request body, generate a response and then send.
    } else {
        // Sending the response success, clear.
    }
}
```

Why grpc-rs instead of others

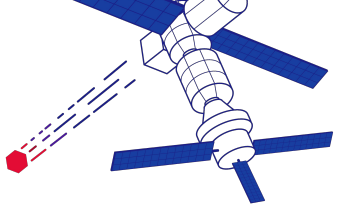
grpc-rs is a Rust bind of gRPC based on futures.

```
impl tikvpb_grpc::Tikv for Service {
    fn coprocessor_stream(
        &mut self,
        ctx: RpcContext<'_>,
        req: Request,
        sink: ServerStreamingSink<Response>,
    ) {
        let stream = self
            .cop
            .parse_and_handle_stream_request(req, Some(ctx.peer()))
            .map(|resp| (resp, WriteFlags::default().buffer_hint(true)))
            .map_err(|e| GrpcError::RpcFailure(RpcStatus::unknown));
        ctx.spawn(sink.send_all(stream));
    }
}
```

Summary about service layer

1. gRPC encapsulates details about network I/O model
2. grpc-rs encapsulates asynchronous code into a synchronous style
3. Completion queue handles thouthands connections in just 4 threads
4. More
    a. Introduce DPDK
    b. Split codec out from service threads

Agenda

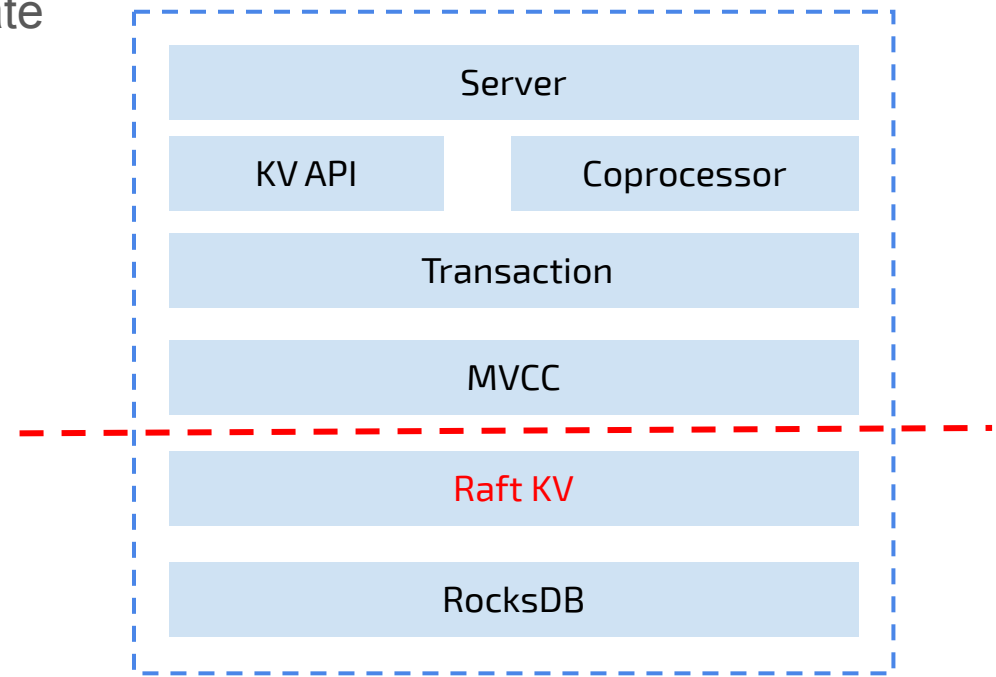


1. TiKV aerial view
2. Service layer built with gRPC
3. **HA component built with Raft**
4. Transaction component built with Percolator

Where is the Raft component in TiKV

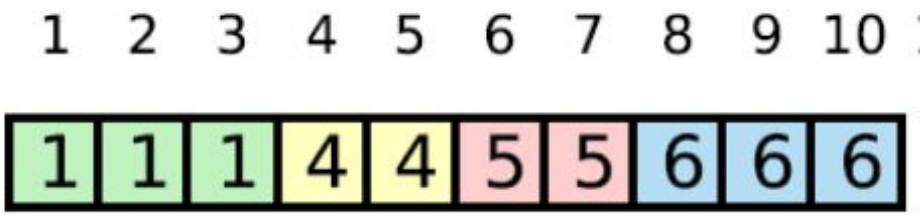
Raft KV module uses raft-rs crate

source path: src/raftstore/store/



Raft basic: Consistency

- Consistency
  - Data is orgnizated as a sorted vector, named as Raft log
  - Elements are attached with terms, and at most 1 leader in each term

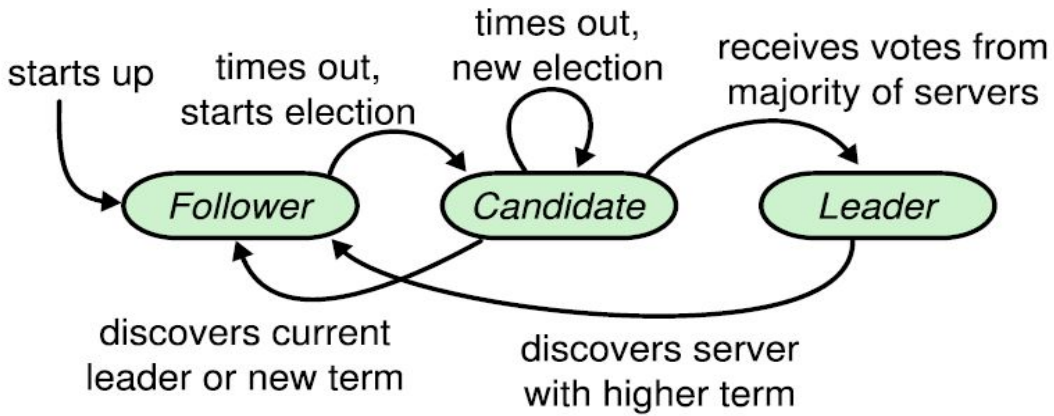


  - Commit an entry needs ACKs from majority replicas
  - New elected leader must contains all committed entries

Raft basic: Election

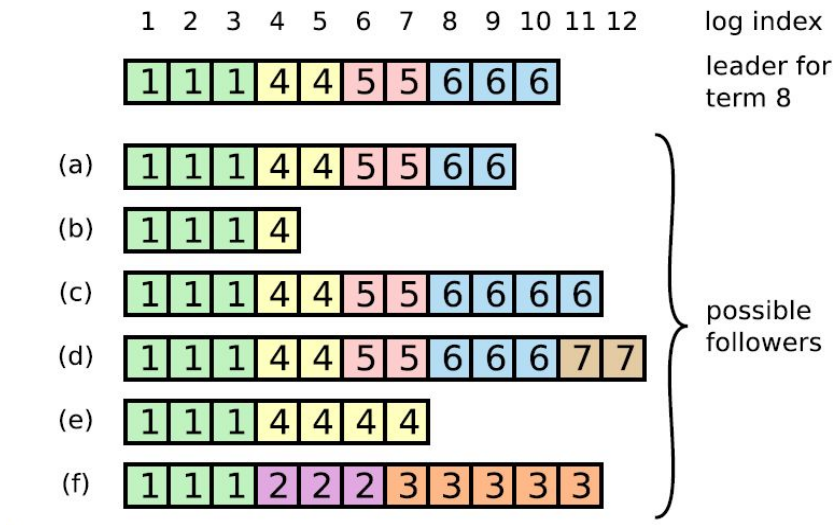
```
follower -> candidate || candidate -> candidate || leader -> candidate:
    self.term += 1

receive(vote_msg[term, index]):
    if self.has_voted:
        return rejected
    else if (term, index) is greater:
        return vote
    else:
        return reject
```



Raft basic: Log Replication

1. Followers can have more logs than leader
2. Entries with higher term will overwrite others
3. Commit an entry needs ACKs from a majority
4. Leader can only commit entries in its term



Multi-Raft in TiKV:

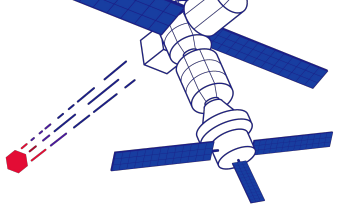
raft-rs example

```
let node = raft::RawNode::new();
loop {
    while let Some(raft_msg) = node.receiver.try_recv() {
        // Handle all received raft messages.
        node.step(raft_msg);
    }
    if time_to_tick { // Tick the raft node.
        node.tick();
        time_to_tick = now();
    }
    let ready = node.ready();
    if ready.has_snapshot() { // Apply snapshot.
    } else if !ready.entries.is_empty() { // Persist new appended raft logs.
    }
    if !ready.committed_entries.is_empty() {
        // Send commmitted entries to apply threads or apply them here.
    }
    send_raft_messages(&ready.messages); // Send out raft messages.
    node.advance(ready); // Finish a ready.
}
```

Summary about Raft component

1. Basic Raft: concepts and algorithm
2. More to learn in TiKV
    a. region split and merge
    b. thread model about `raftstore`
3. More to implement
    a. configuration change on received instead of applied
    b. flexible Raft
    c. follower replication

Agenda



1. TiKV aerial view
2. Service layer built with gRPC
3. HA component built with Raft
4. **Transaction component built with Percolator**

PingCAP

Transaction Model

- Inspired by Google Percolator
- 3 column families
  - **cf:lock**: An uncommitted transaction is writing this cell; contains the location/pointer of primary lock
  - **cf: write**: it stores the commit timestamp of the data
  - **cf: data**: Stores the data itself

PingCAP

Transaction Model

Bob wants to transfer 7$ to Joe

Key	Bal: Data	Bal: Lock	Bal: Write
Bob	6: 5: $10	6: 5:	6: data @ 5 5:
Joe	6: 5: $2	6: 5:	6: data @ 5 5:

PingCAP

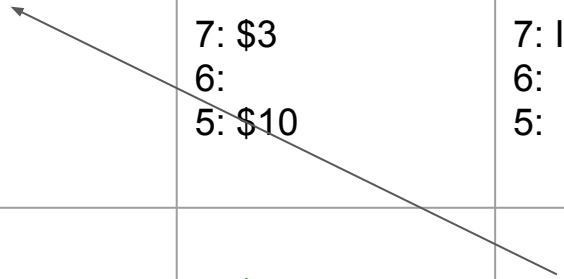
Transaction Model (prewrite)

Key	Bal: Data	Bal: Lock	Bal: Write
Bob	7: $3 6: 5: $10	7: I am Primary 6: 5:	7: 6: data @ 5 5:
Joe	6: 5: $2	6: 5:	6: data @ 5 5:

PingCAP

Transaction Model (prewrite)

Key	Bal: Data	Bal: Lock	Bal: Write
Bob	7: $3 6: 5: $10	7: I am Primary 6: 5:	7: 6: data @ 5 5:
Joe	7: $9 6: 5: $2	7:Primary@Bob.bal 6: 5:	7: 6: data @ 5 5:



PingCAP

Transaction Model (commit)

Key	Bal: Data	Bal: Lock	Bal: Write
Bob	8: 7: $3 6: 5: $10	8: 7: ~~I am Primary~~ 6: 5:	8: data @ 7 7: 6: data @ 5 5:
Joe	8: 7: $9 6: 5: $2	8: 7:Primary@Bob 6: 5:	8: 7: 6: data @ 5 5:

PingCAP

Transaction Model (read)

Key	Bal: Data	Bal: Lock	Bal: Write
Bob	8: 7: $3 6: 5: $10	8: 7: ~~I am Primary~~ f ② 6: 5:	8: data @ 7 ③ 7: 6: data @ 5 5:
Joe	8: 7: $9 6: 5: $2	8: 7:Primary@Bob ① 6: 5:	8: data @ 7 ④ 7: 6: data @ 5 5:

Summary about transaction component

1. Think more
    a. snapshot isolation & write skew
    b. pessimistic lock
    c. put short value in cf write
2. More to implement
    a. optimize to not fetch commit ts
    b. 1PC for single region/store transactions

PingCAP

Thank You!

Any Questions ?

关注 PingCAP 官方微信

了解更多技术干货