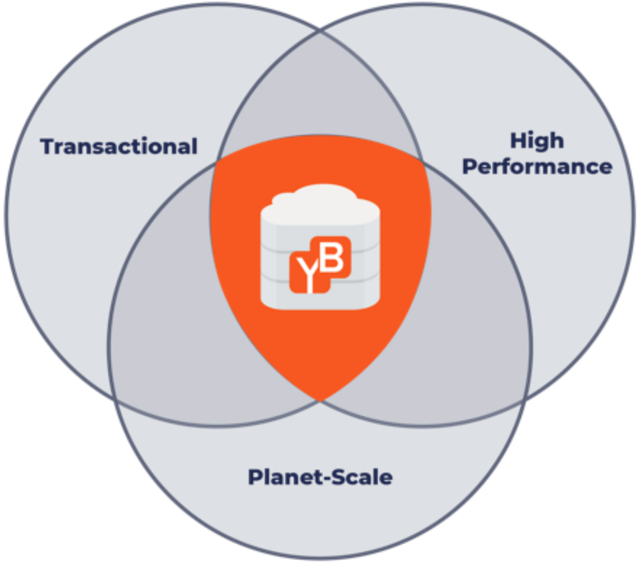


Distributed PostgreSQL with YugaByte DB

Karthik Ranganathan

PostgresConf Silicon Valley
Oct 16, 2018

YugaByte

CHECKOUT THIS REPO:

github.com/YugaByte/yb-sql-workshop

About Us

Founders

Kannan Muthukkaruppan, CEO
Nutanix ♦ Facebook ♦ Oracle
IIT-Madras, University of California-Berkeley

Karthik Ranganathan, CTO
Nutanix ♦ Facebook ♦ Microsoft
IIT-Madras, University of Texas-Austin

Mikhail Bautin, Software Architect
ClearStory Data ♦ Facebook ♦ D.E.Shaw
Nizhny Novgorod State University, Stony Brook

- ✓ Founded Feb 2016
- ✓ Apache HBase committers and early engineers on Apache Cassandra
- ✓ Built Facebook's NoSQL platform powered by Apache HBase
- ✓ Scaled the platform to serve many mission-critical use cases
  - Facebook Messages (Messenger)
  - Operational Data Store (Time series Data)
- ✓ Reassembled the same Facebook team at YugaByte along with engineers from Oracle, Google, Nutanix and LinkedIn

WORKSHOP AGENDA

- What is YugaByte DB? Why Another DB?
- Exercise 1: BI Tools on YugaByte PostgreSQL
- Exercise 2: Distributed PostgreSQL Architecture
- Exercise 3: Sharding and Scale Out in Action
- Exercise 4: Fault Tolerance in Action

WHAT IS YUGABYTE DB?

YugaByte **DB**

A **transactional, planet-scale** database

for building **high-performance** cloud services.

YugaByte DB

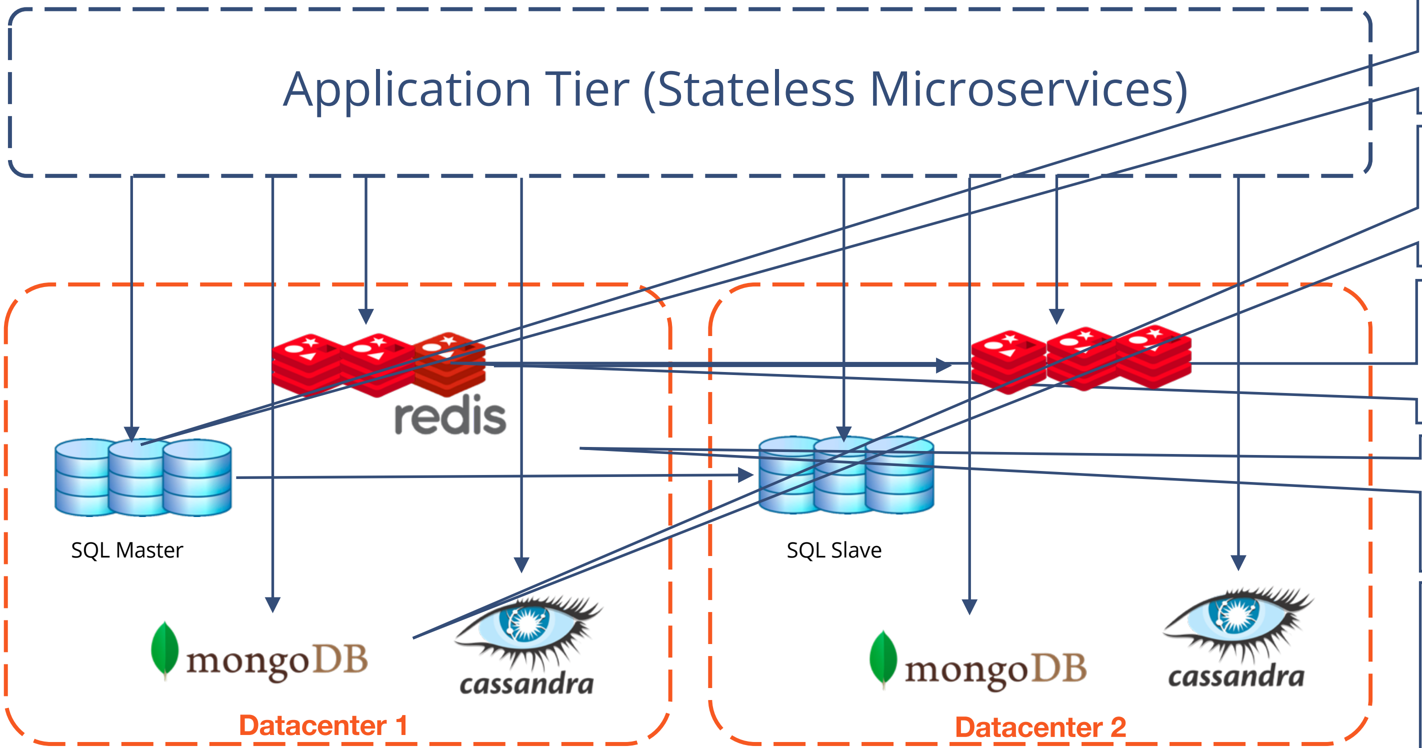
NoSQL + SQL

Cloud Native

WHY ANOTHER DB?

Typical Stack Today

Fragile infra with several moving parts



SQL for OLTP data
Manual sharding
Cost: dev team

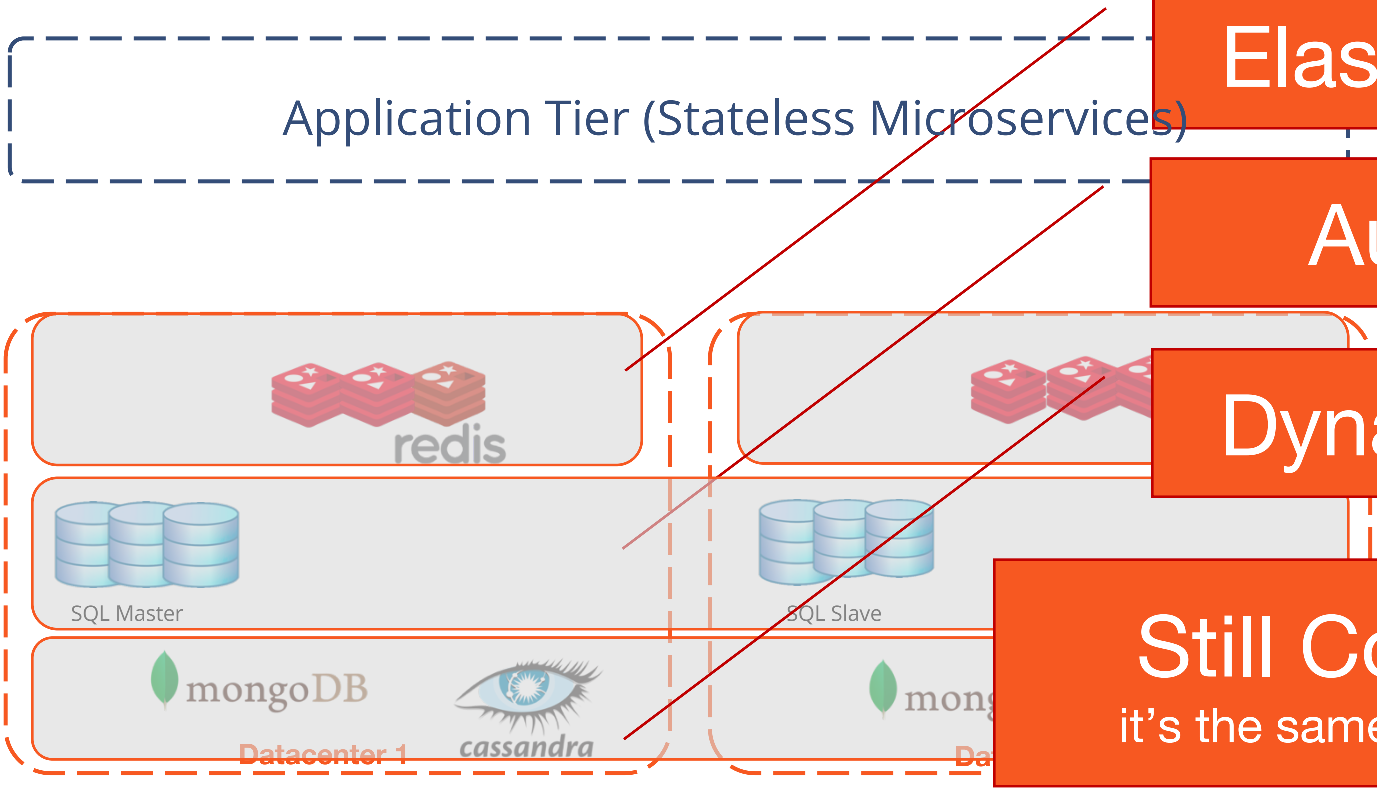
NoSQL for other data
App aware of data silo
Cost: dev team

Cache for low latency
App does caching
Cost: dev team

Manual replication
Manual failover
Cost: ops team

Data inconsistency/loss
Fragile infra
Hours of debugging
Cost: dev + ops team

Does AWS change this?



Elasticache

Aurora

DynamoDB

Still Complex

it's the same architecture

System-of-Record DBs for Global Apps

Not Portable	Amazon DynamoDB, Azure Cosmos DB
Open Source	cassandra, mongoDB

High Performance, **~~Transactional~~**, **Planet-Scale**

Open Source	YB

High Performance, **Transactional**, **Planet-Scale**

Not Portable	Amazon Aurora
Open Source	PostgreSQL, MySQL

High Performance, **Transactional**, **~~Planet-Scale~~**

Not Portable	Google Cloud Spanner

~~High Performance~~, **Transactional**, **Planet-Scale**

Design Principles

TRANSACTIONAL

- Single Shard & Distributed ACID Txns
- Document-Based, Strongly Consistent Storage

HIGH PERFORMANCE

- Low Latency, Tunable Reads
- High Throughput

PLANET-SCALE

- Global Data Distribution
- Auto Sharding & Rebalancing

CLOUD NATIVE

- Built For The Container Era
- Self-Healing, Fault-Tolerant

OPEN SOURCE

- Apache 2.0
- **Popular APIs Extended**
  Apache Cassandra, Redis and PostgreSQL (BETA)

EXERCISE #1

BUSINESS INTELLIGENCE

EXERCISE #2

DISTRIBUTED POSTGRES: ARCHITECTURE

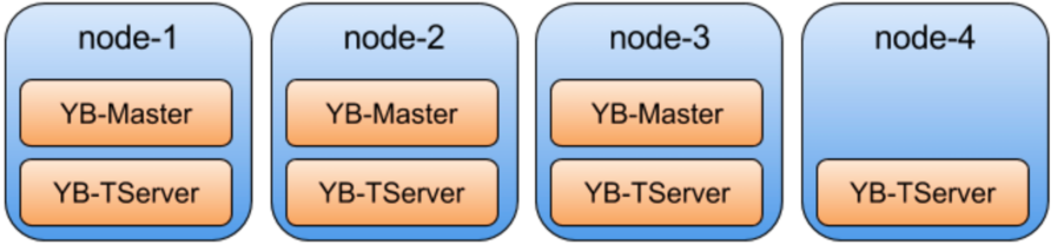
ARCHITECTURE

Overview

YugaByte DB Process Overview

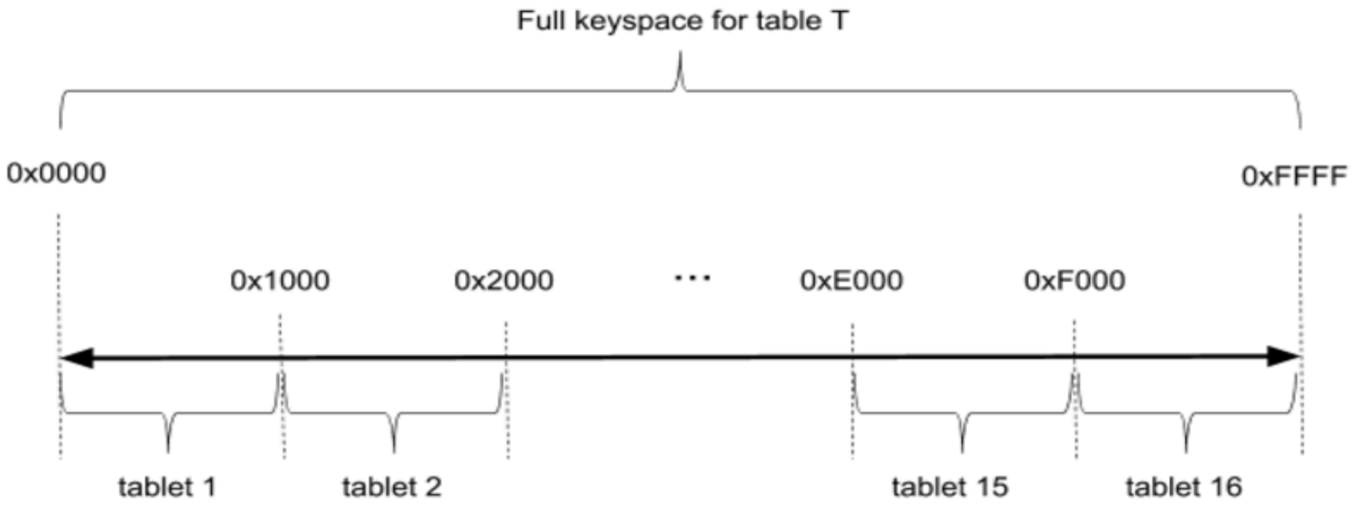
- Universe = cluster of nodes
- Two sets of processes: YB-Master & YB-TServer

- Example universe
  - 4 nodes
  - rf=3

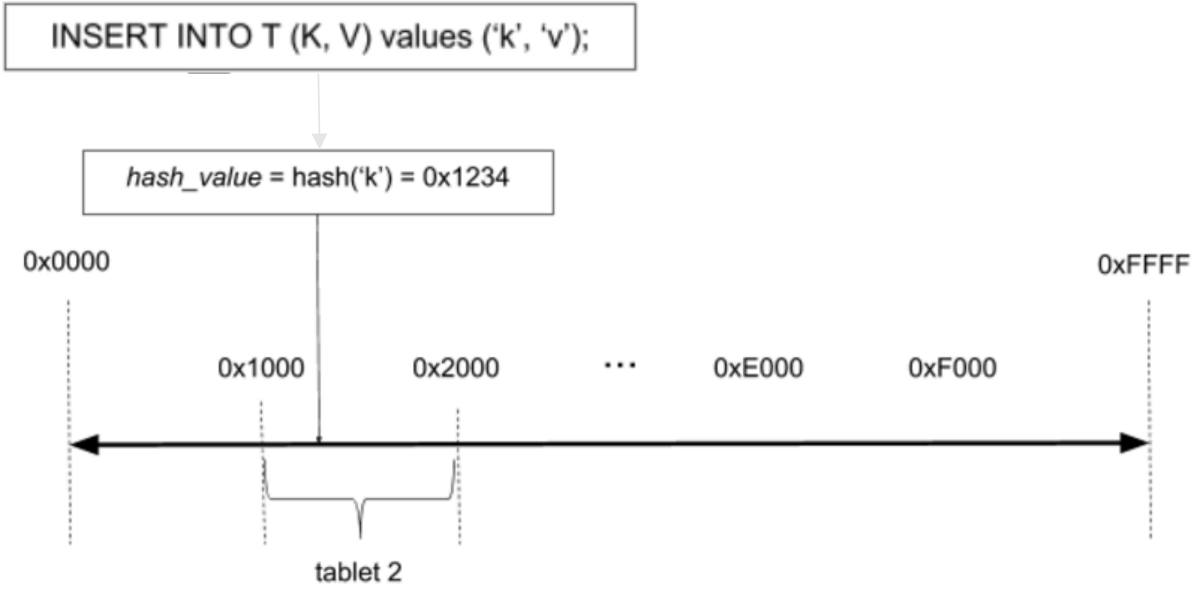


Sharding data

- User table split into tablets

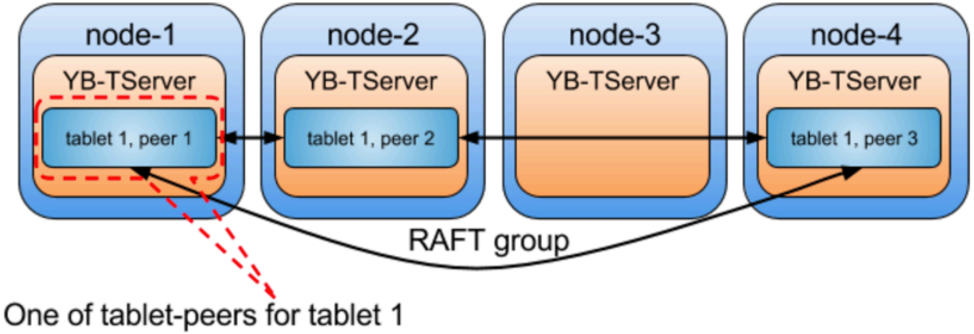


One tablet for every key



Tablets and replication

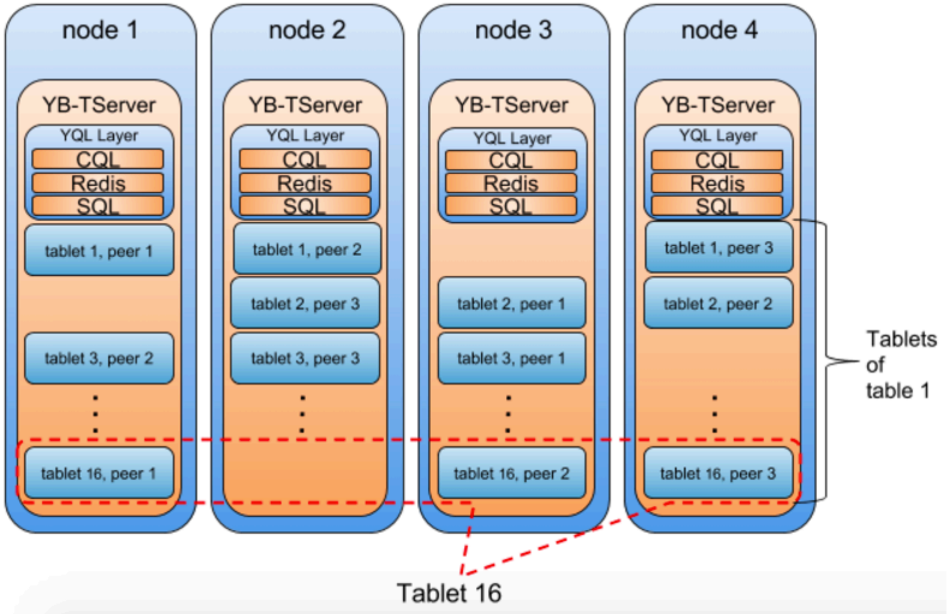
- Tablet = set of tablet-peers in a RAFT group



- Num tablet-peers in tablet = replication factor (*RF*)
  Tolerate 1 failure : RF=3
  Tolerate 2 failures: RF=5

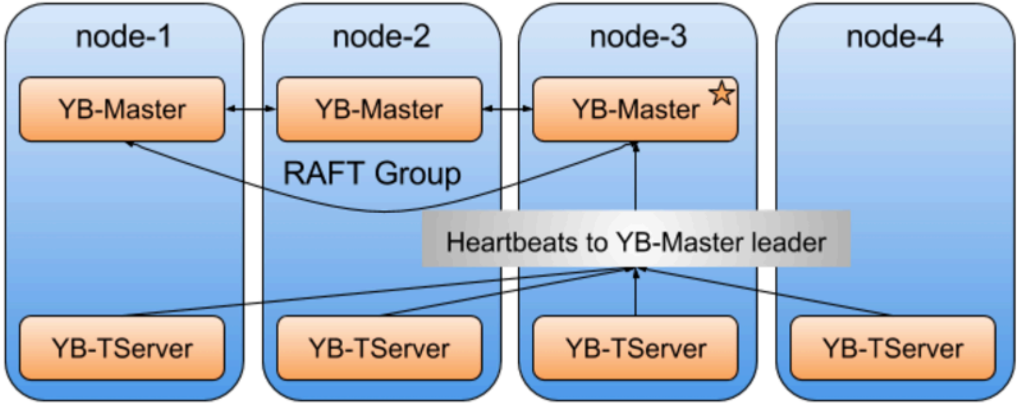
YB-TServer

- Process that does IO
- Hosts tablet for tables
- Hosts transaction manager
- Auto memory sizing
  - Block cache
  - Memstores



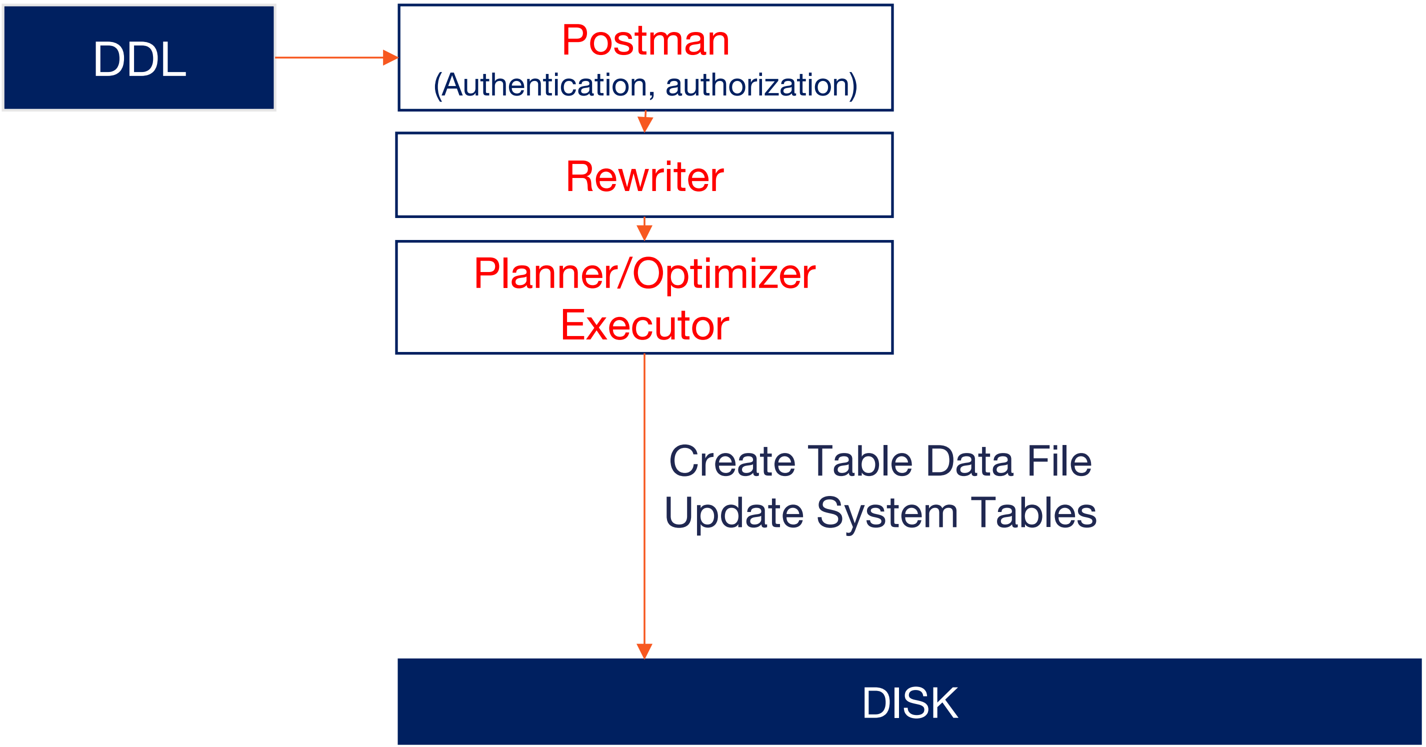
YB-Master

- Not in critical path
- System metadata store
  - Keyspaces, tables, tablets
  - Users/roles, permissions
- Admin operations
  - Create/alter/drop of tables
  - Backups
  - Load balancing (leader and data balancing)
  - Enforces data placement policy

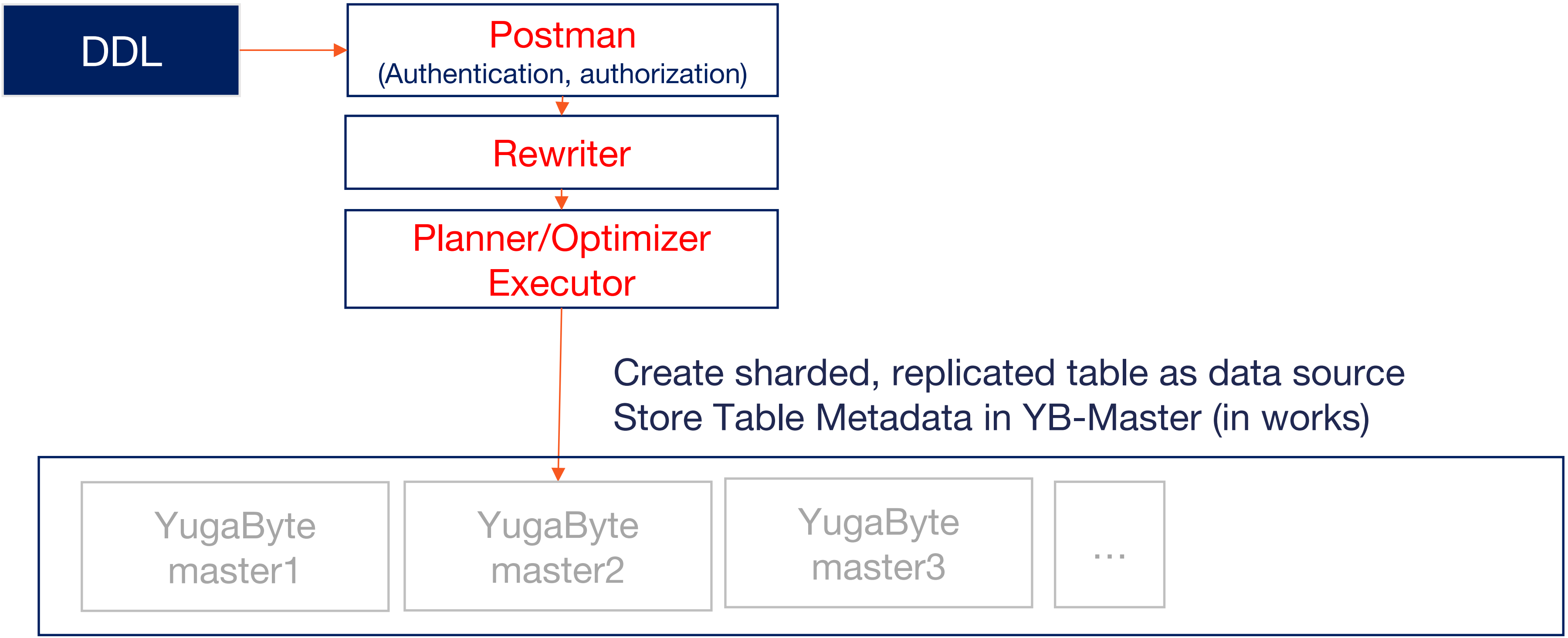


HANDLING DDL STATEMENTS

DDL Statements in PostgreSQL

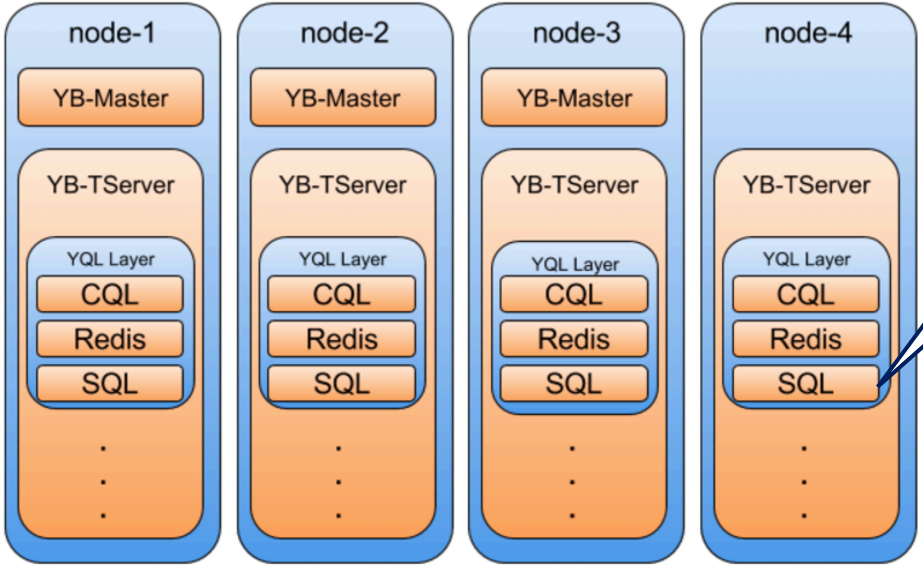


DDL Statements in YugaByte DB PostgreSQL



YugaByte Query Layer (YQL)

- Stateless, runs in each YB-TServer process

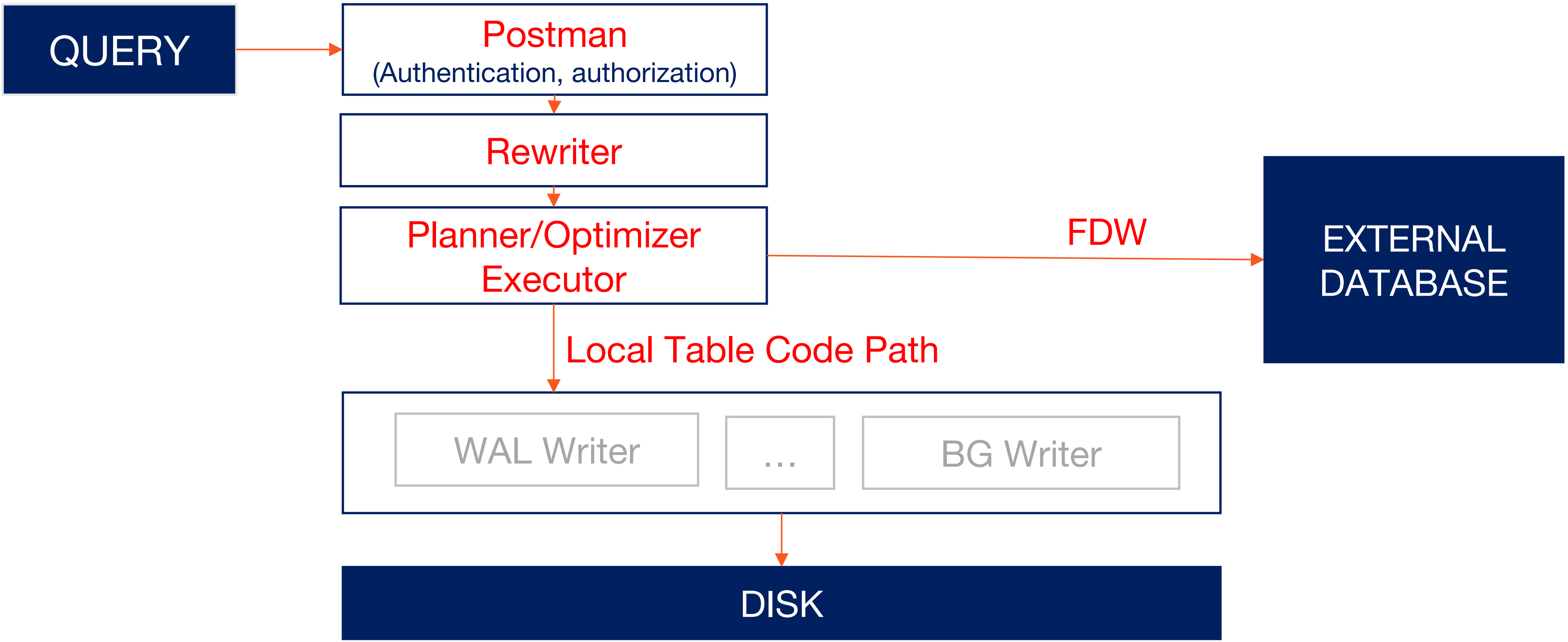


GA Goal: Distributed Stateless PostgreSQL Layer

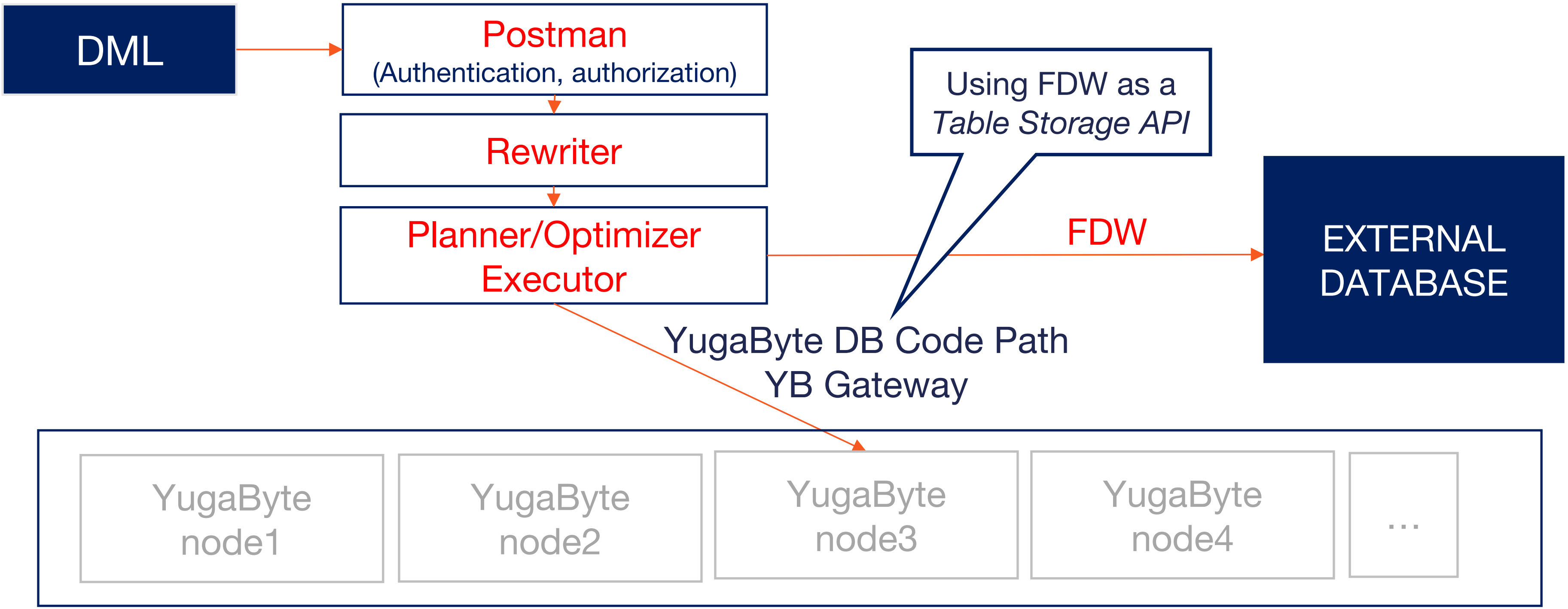
Current Beta uses a single Stateless PostgreSQL Layer

HANDLING DML QUERIES

DDL Queries in PostgreSQL



DML Queries in YugaByte DB PostgreSQL



ARCHITECTURE

Data Persistence

Data Persistence in DocDB

- DocDB is YugaByte DB's LSM storage engine
- Persistent key to document store
- Extends and enhances RocksDB
- Designed to support high data-densities per node

DocDB: Key-to-Document Store

- Document key
  - CQL/SQL/Redis primary key
- Document value
  - a CQL or SQL row
  - Redis data structure
- Fine-grained reads and writes

```
DocumentKey1 = {
        SubKey1 = {
                SubKey2 = Value1
                SubKey3 = Value2
        },
        SubKey4 = Value3
}
```

DocDB Data Format

Example Insert

```
INSERT INTO msgs (user_id, msg_id, msg)
VALUES ('user1', 10, 'msg1');
```

Encoding

```
(hash1, 'user1', 10), liveness_column_id, T1 -> [NULL]
(hash1, 'user1', 10), msg_column_id, T1 -> 'msg1'
```

Some of the RocksDB enhancements

- WAL and MVCC enhancements
  - Removed RocksDB WAL, re-uses Raft log
  - MVCC at a higher layer
  - Coordinate RocksDB memstore flushing and Raft log garbage collection

- File format changes
  - Sharded (multi-level) indexes and Bloom filters

- Splitting data blocks & metadata into separate files for tiering support

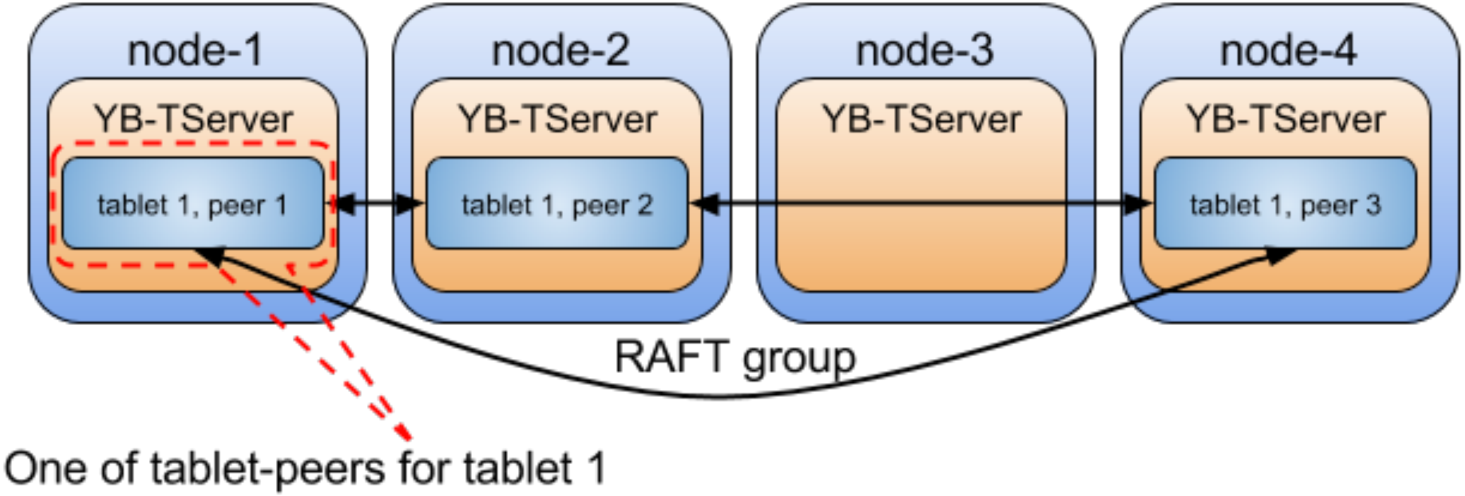
- Separate queues for large and small compactions

More Enhancements to RocksDB

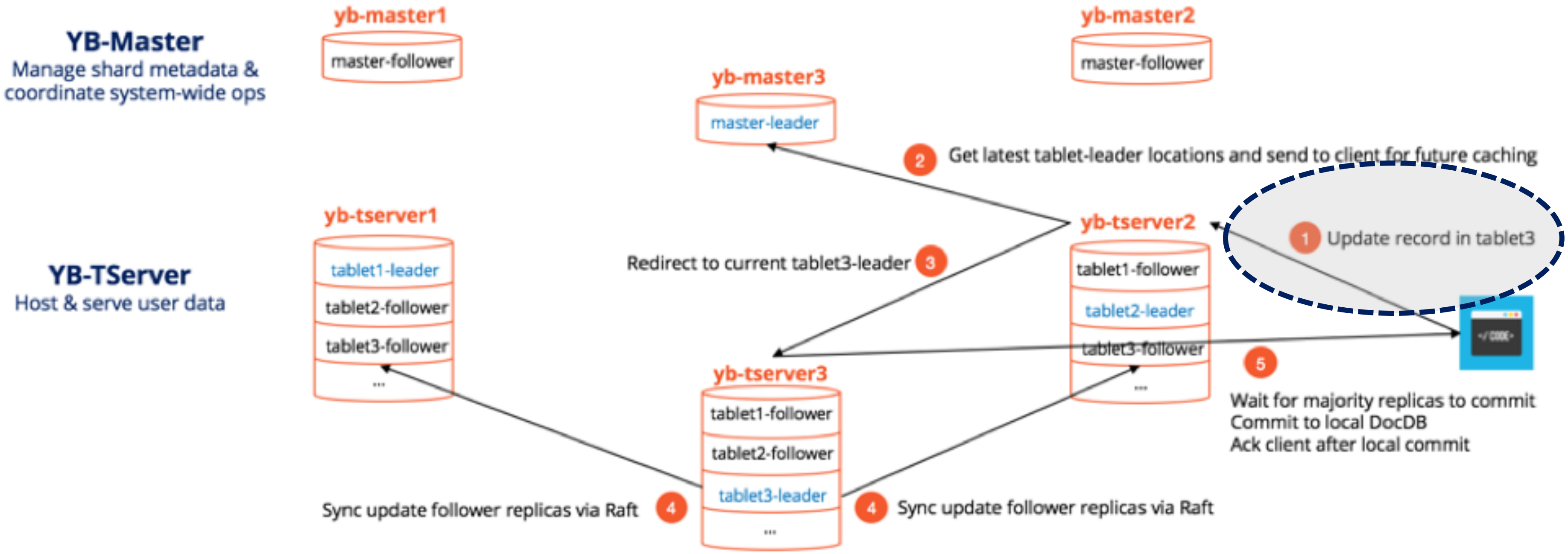
- Data model aware Bloom filters
- Per-SSTable key range metadata to optimize range queries
- Server-global block caches & memstore limits
- Scan-resistant block cache (single-touch and multi-touch)

ARCHITECTURE
Data Replication

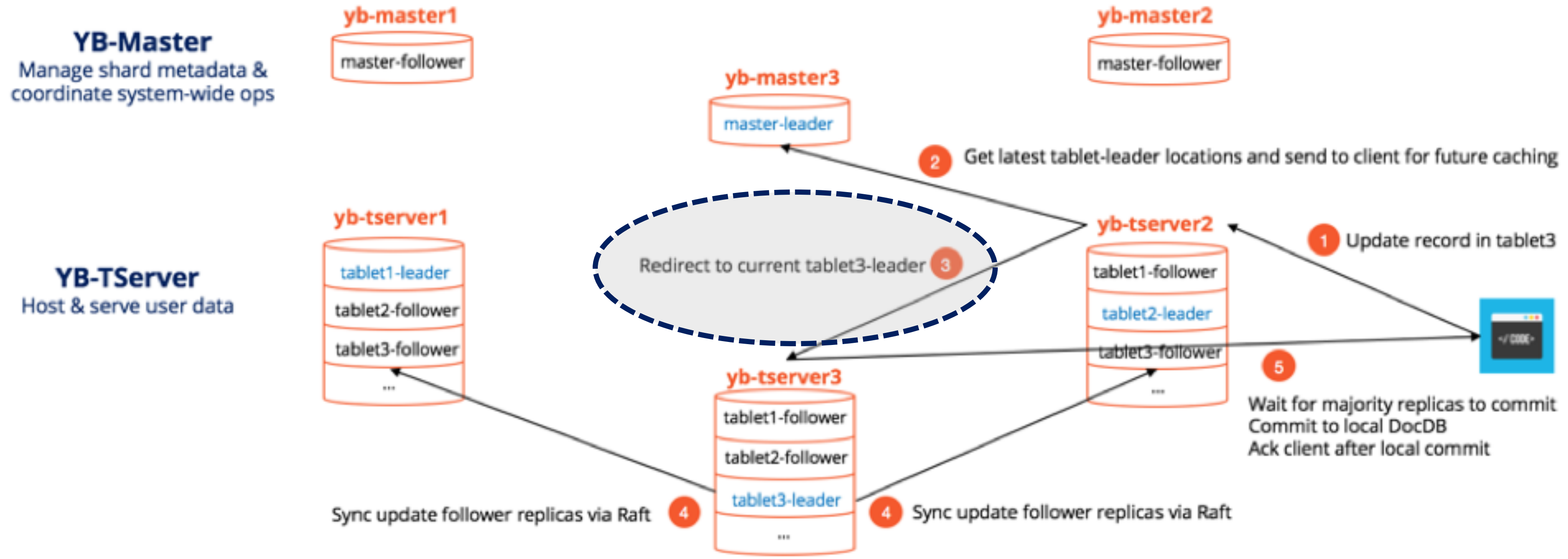
Raft Replication for Consistency



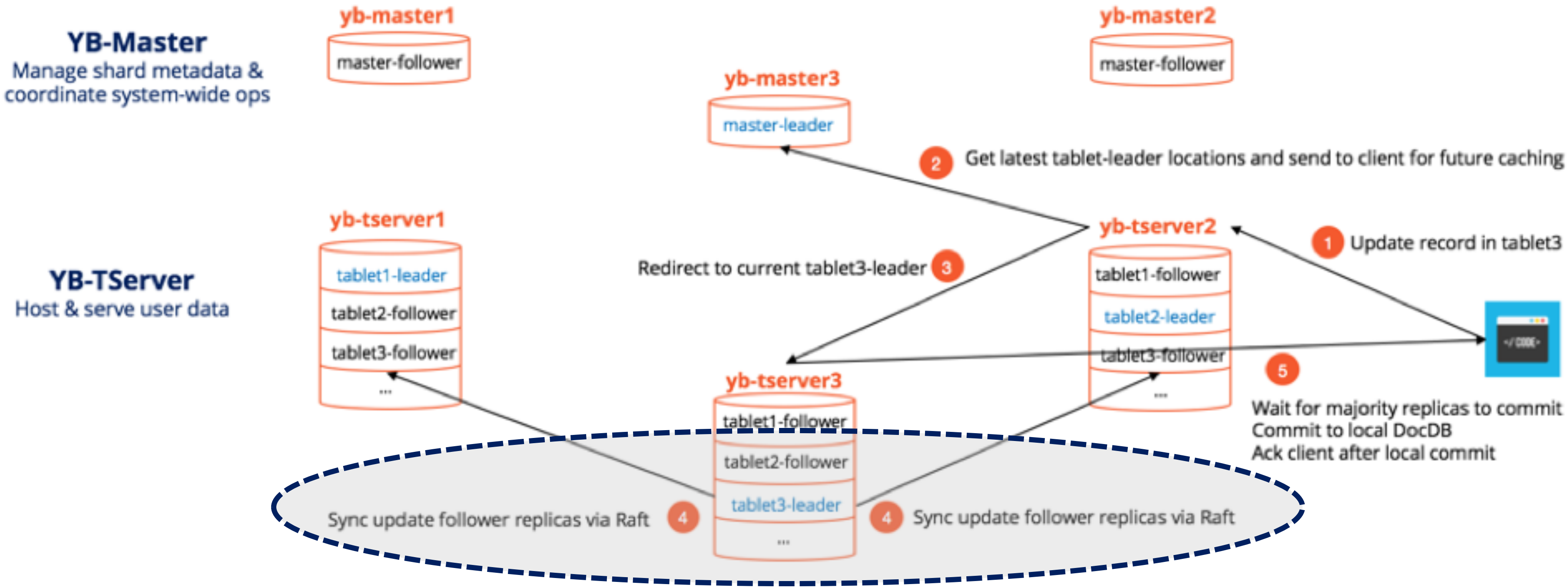
How Raft Replication Works



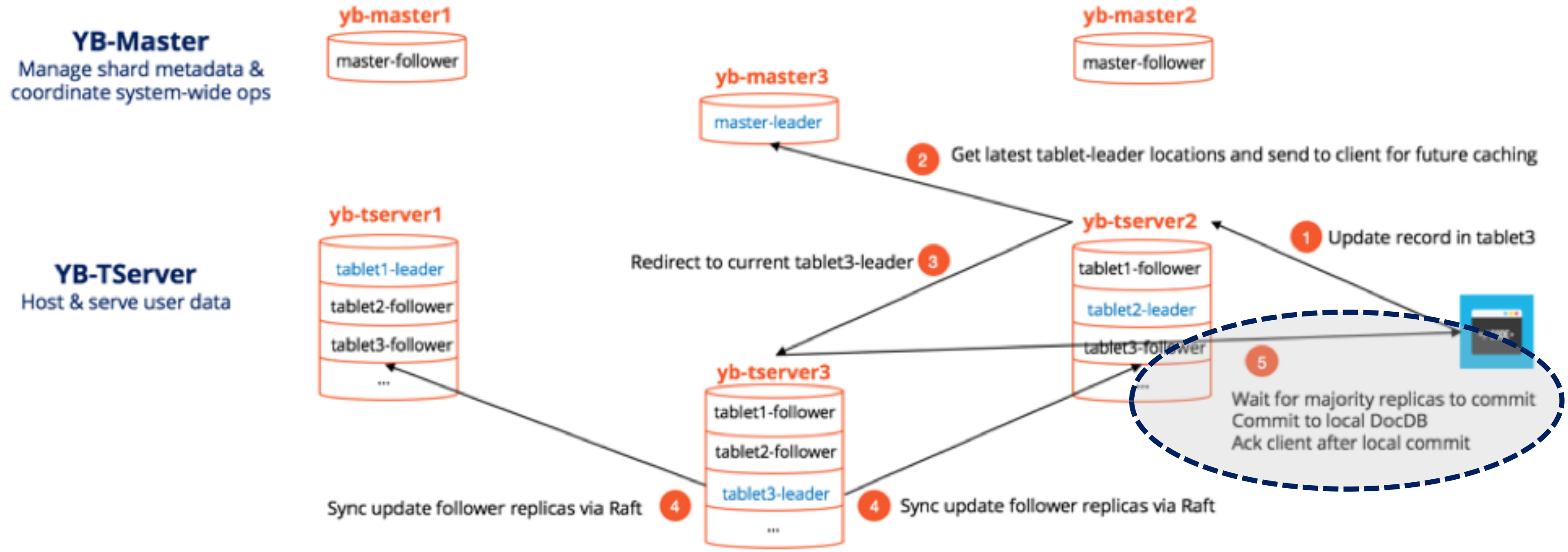
How Raft Replication Works



How Raft Replication Works



How Raft Replication Works

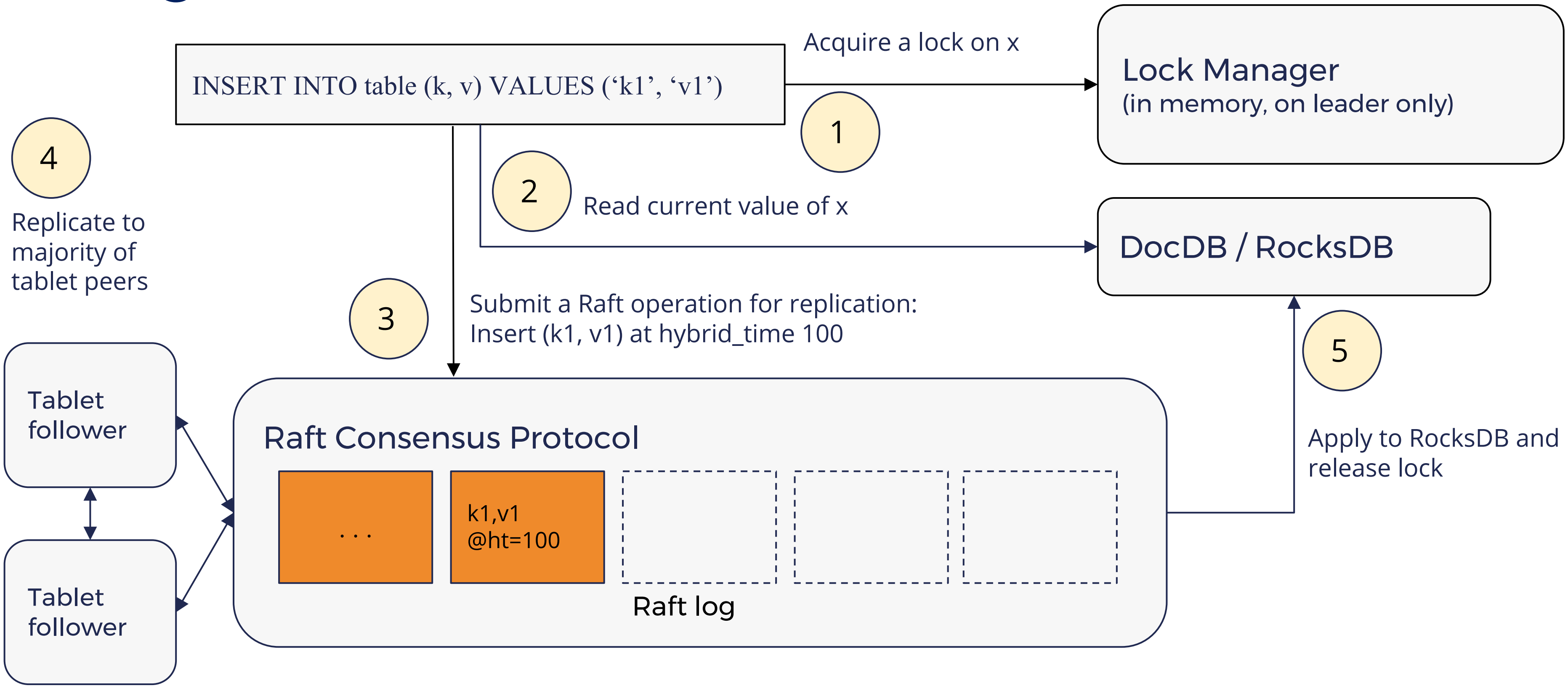


Raft Related Enhancements

- Leader Leases
- Multiple Raft groups (1 per tablet)
- Leader Balancing
- Group Commits
- Observer Nodes / Read Replicas

ARCHITECTURE
Transactions

Single Shard Transactions



MVCC for Lockless Reads

- Achieved through HybridTime (HT)
  - Monotonically increasing timestamp
- Allows reads at a particular HT without locking
- Multiple versions may exist temporarily
  - Reclaim older values during compactions

Single Shard Transactions

- Each tablet maintains a "safe time" for reads
  - Highest timestamp such that the view as of that timestamp is fixed
  - In the common case it is just before the hybrid time of the next uncommitted record in the tablet

Committed (in Raft)				Uncommitted		
k1, ht1	k2, ht2	k3, ht3	|	k1, ht4	k2, ht5	k3, ht6

Distributed Transactions

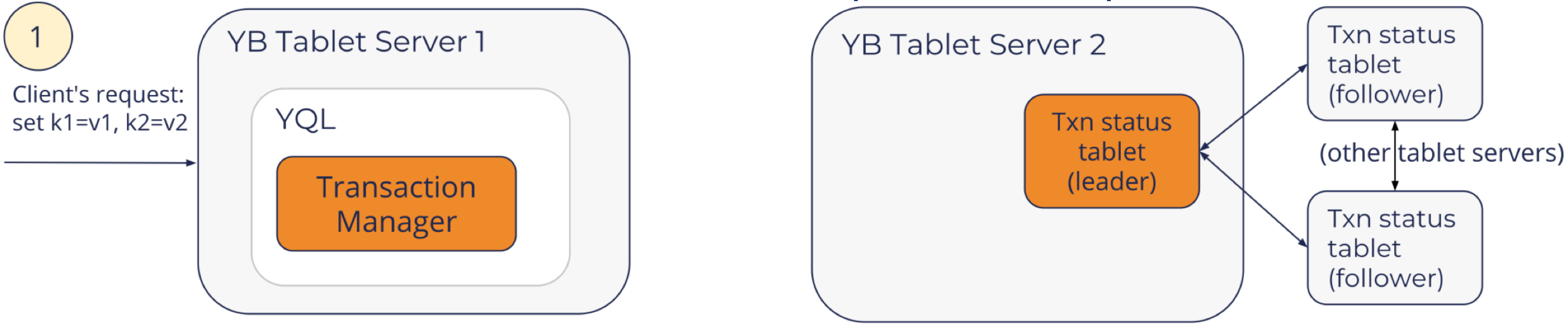
- Fully decentralized architecture
- Every tablet server can act as a Transaction Manager
- A distributed Transaction Status table

  Tracks state of active transactions
- Transactions can have 3 states:

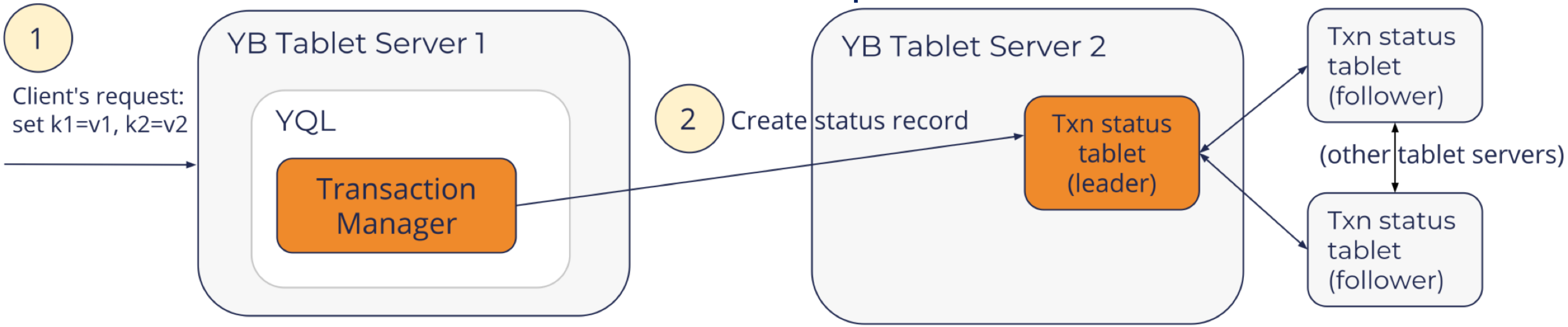
  pending, committed, aborted

Distributed Transactions – Write Path

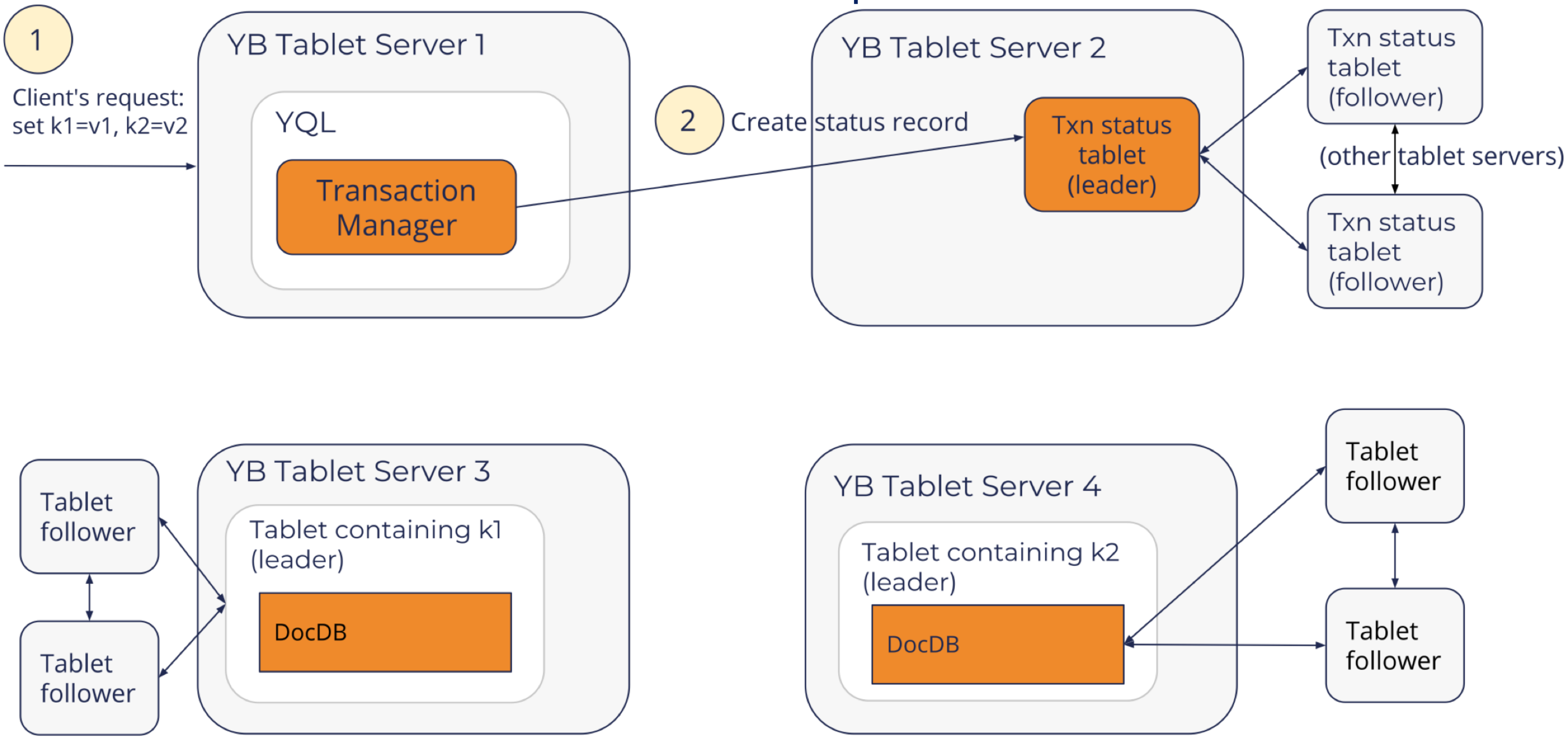
Distributed Transactions – Write Path Step 1: Client request



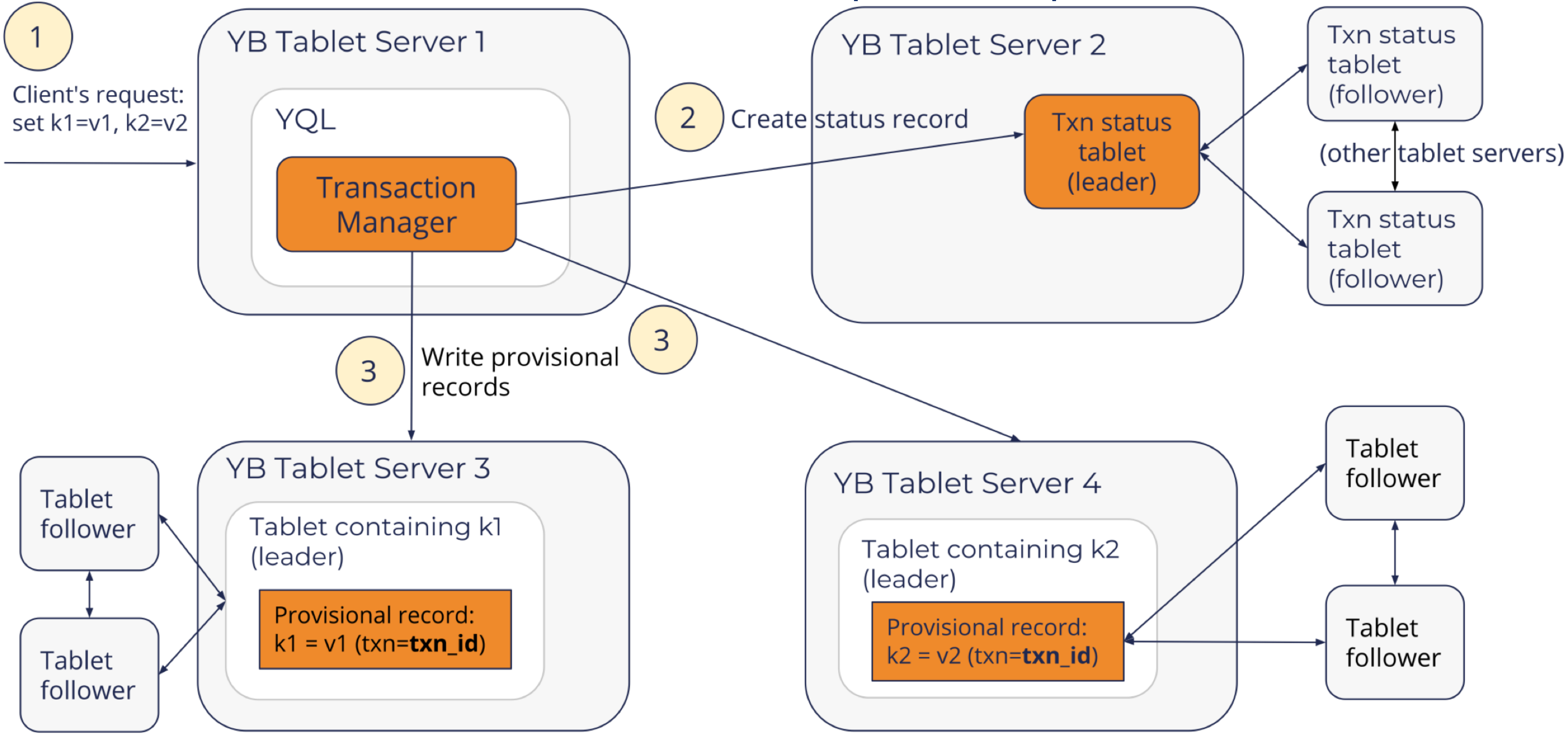
Distributed Transactions – Write Path Step 2: Create status record



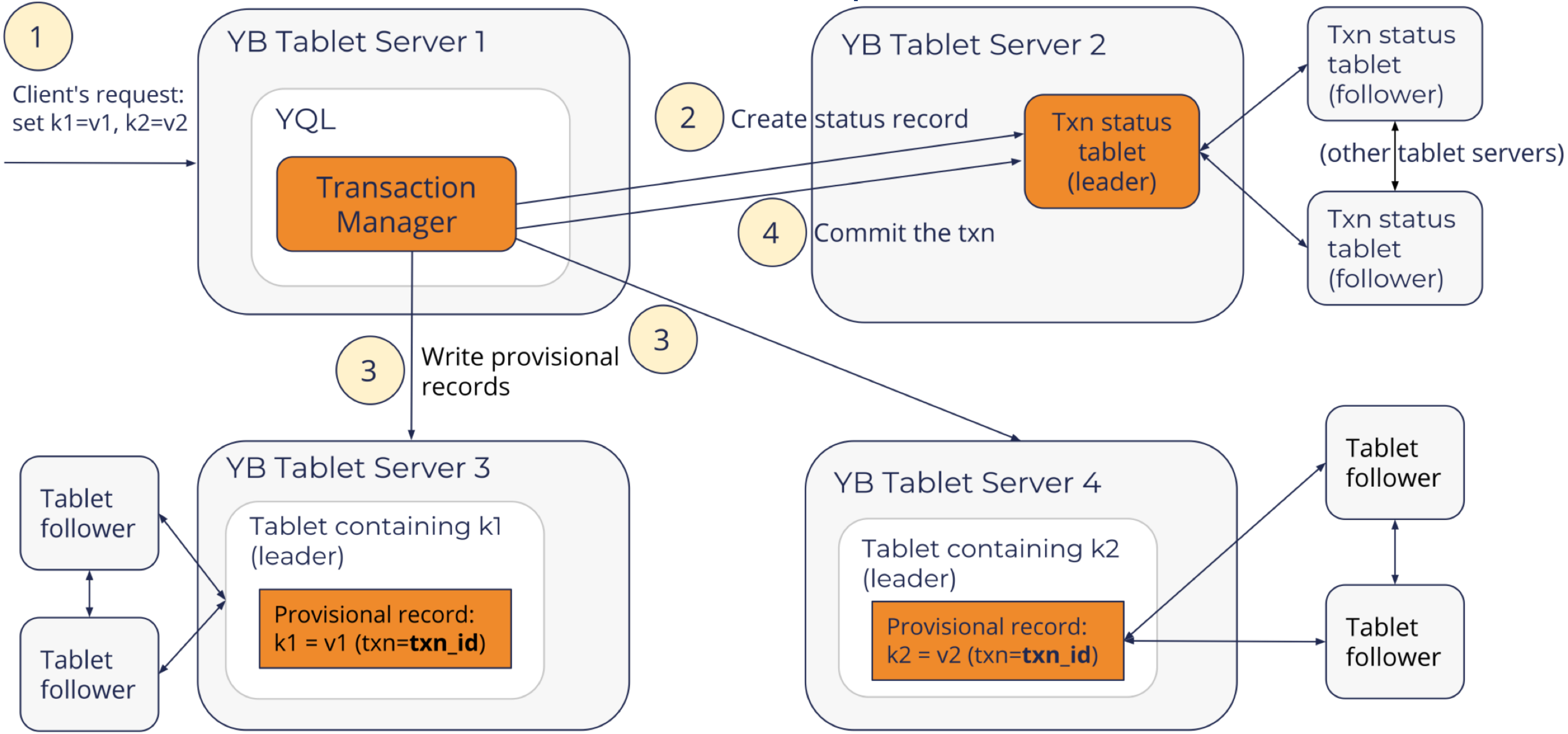
Distributed Transactions – Write Path Step 2: Create status record



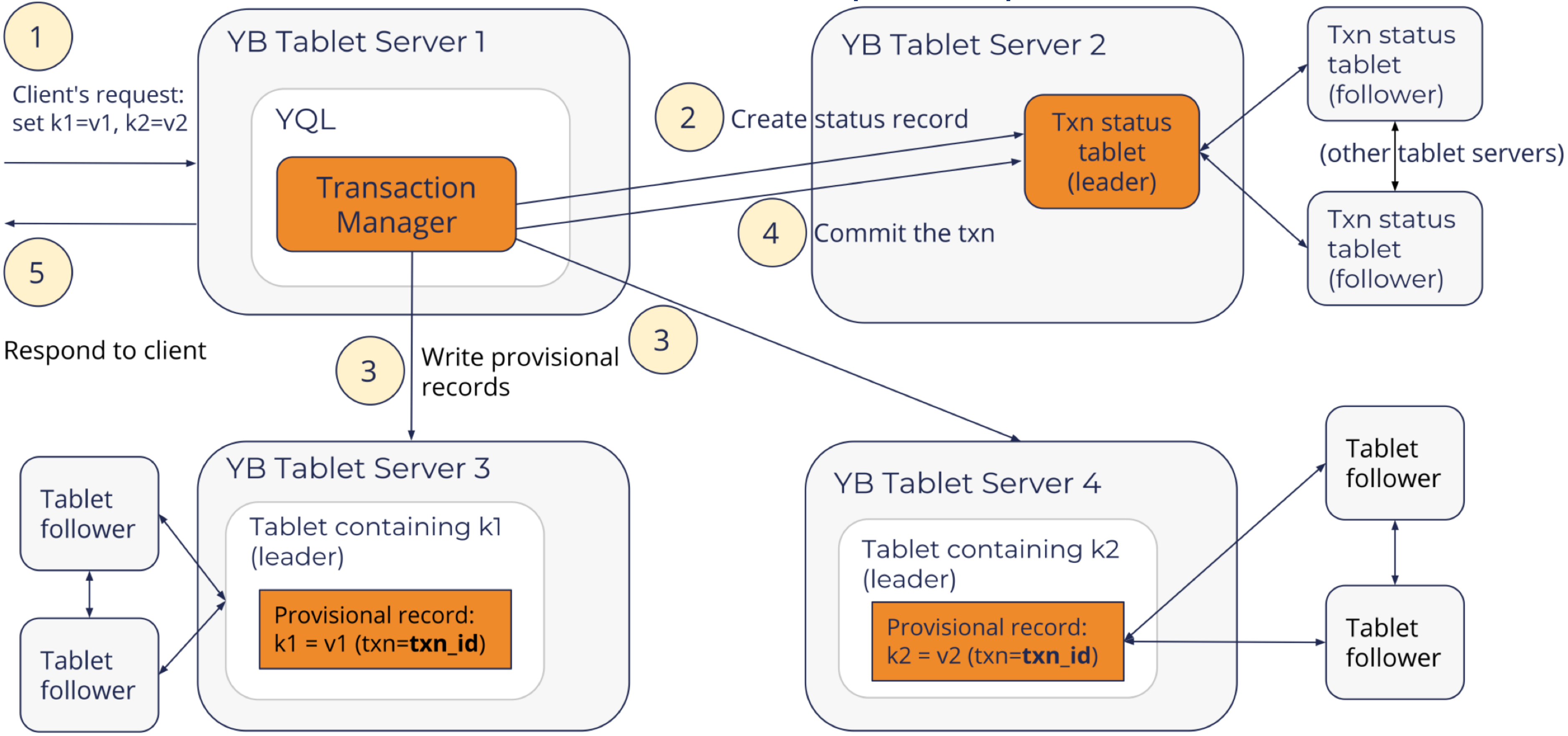
Distributed Transactions – Write Path Step 3: Write provisional records



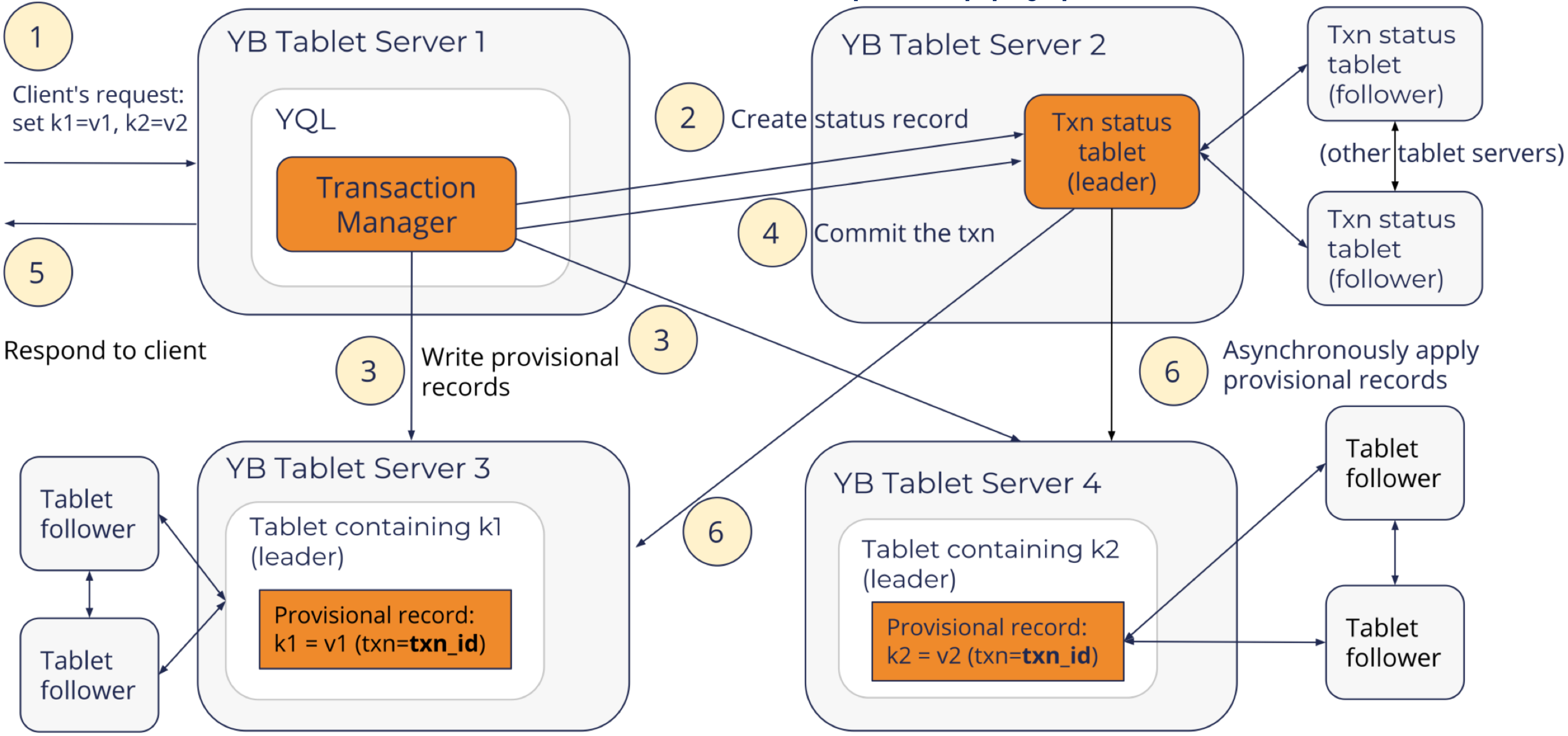
Distributed Transactions – Write Path Step 4: Atomic commit



Distributed Transactions – Write Path Step 5: Respond to client



Distributed Transactions – Write Path Step 6: Apply provisional records



Isolation Levels

- Currently Snapshot Isolation is supported
  - Write-write conflicts detected when writing provisional records
- Serializable isolation (roadmap)
  - Reads in RW txns also need provisional records
- Read-only transactions are always lock-free

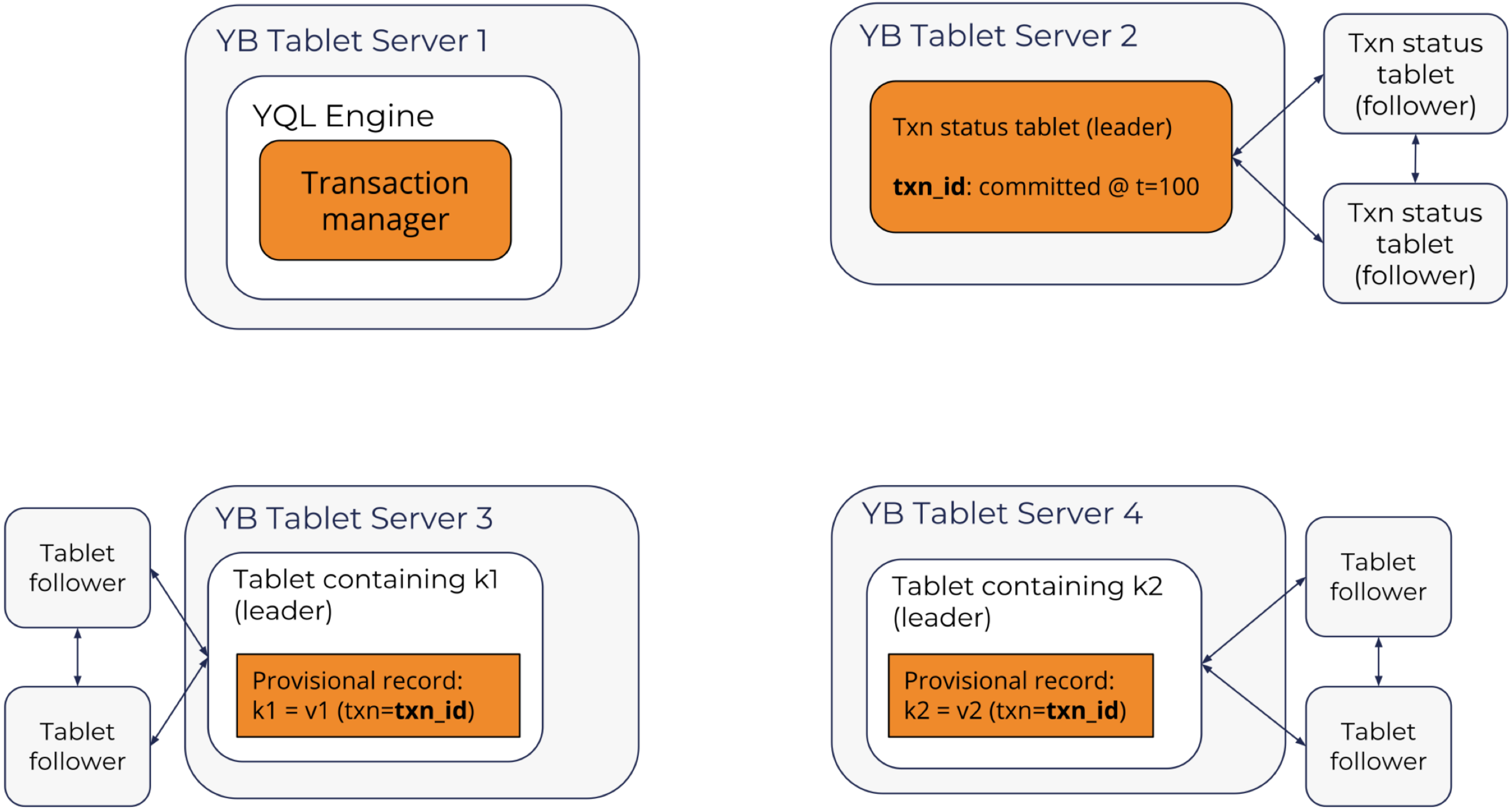
Clock Skew and Read Restarts

- Need to ensure the read timestamp is high enough
  - Committed records the client might have seen must be visible

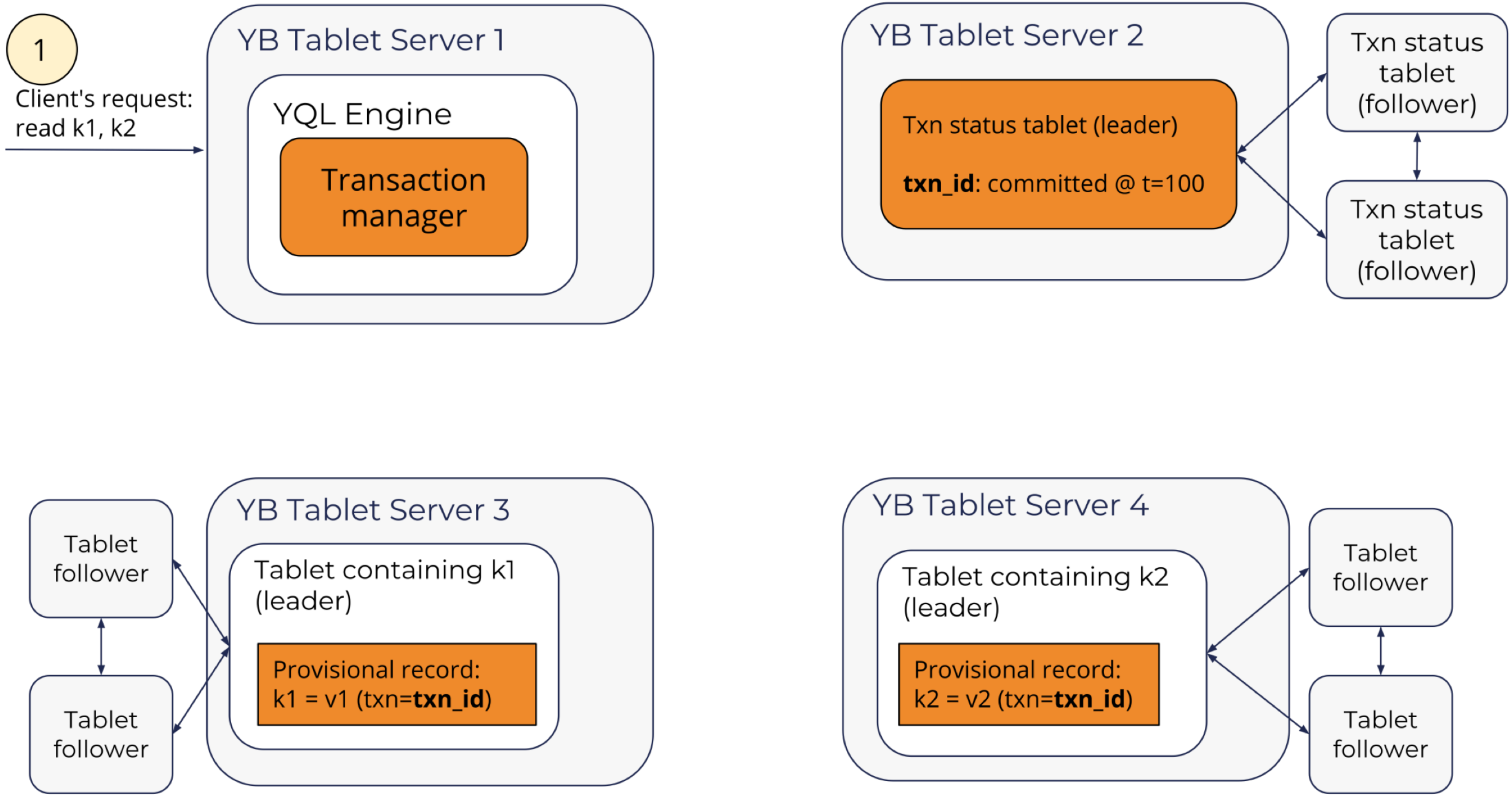
- Optimistically use current Hybrid Time, re-read if necessary
  - Reads are restarted if a record with a higher timestamp that the client could have seen is encountered
  - Read restart happens at most once per tablet
  - Relying on bounded clock skew (NTP, AWS Time Sync)

- Only affects multi-row reads of frequently updated records

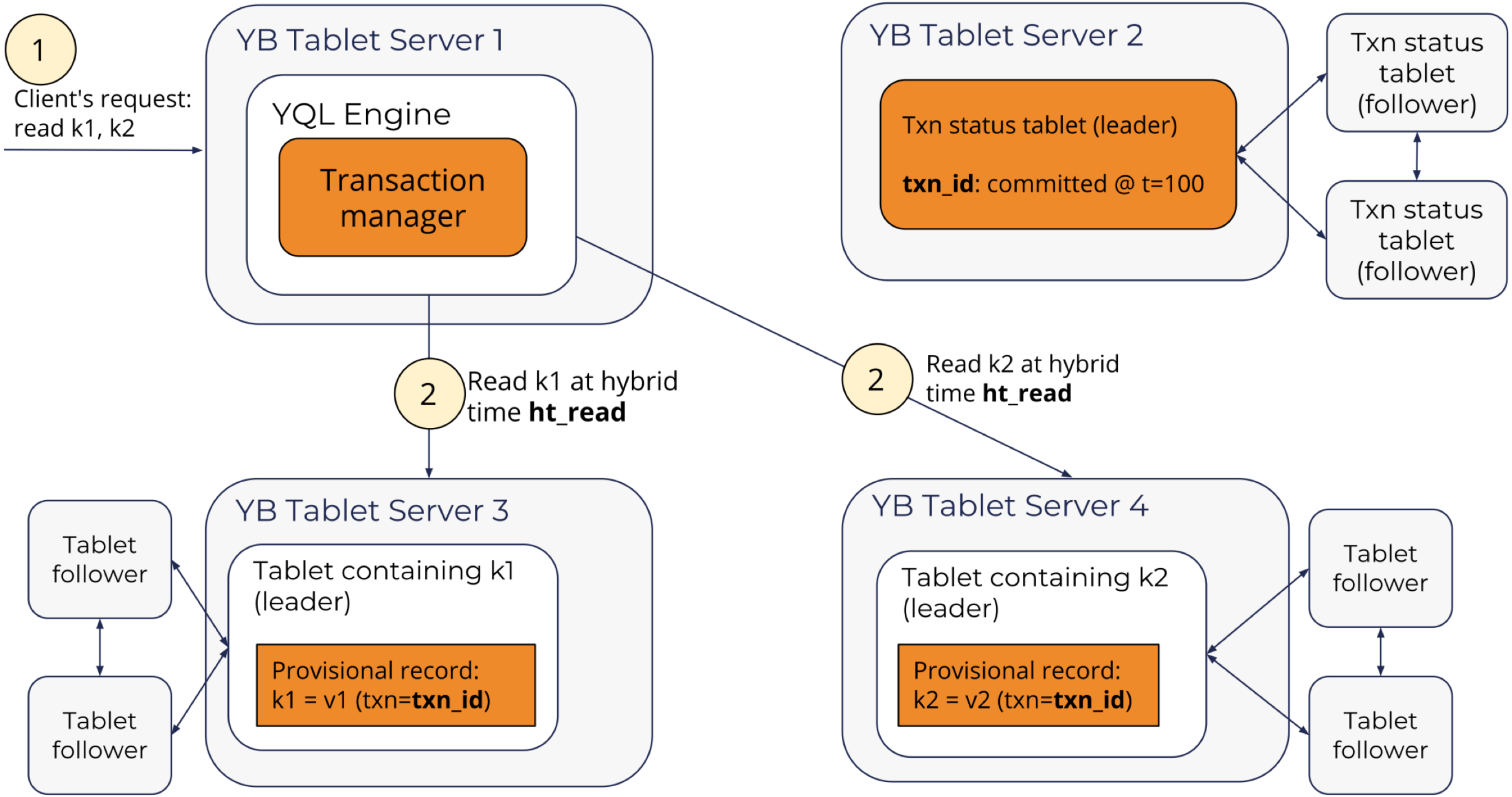
Distributed Transactions – Read Path



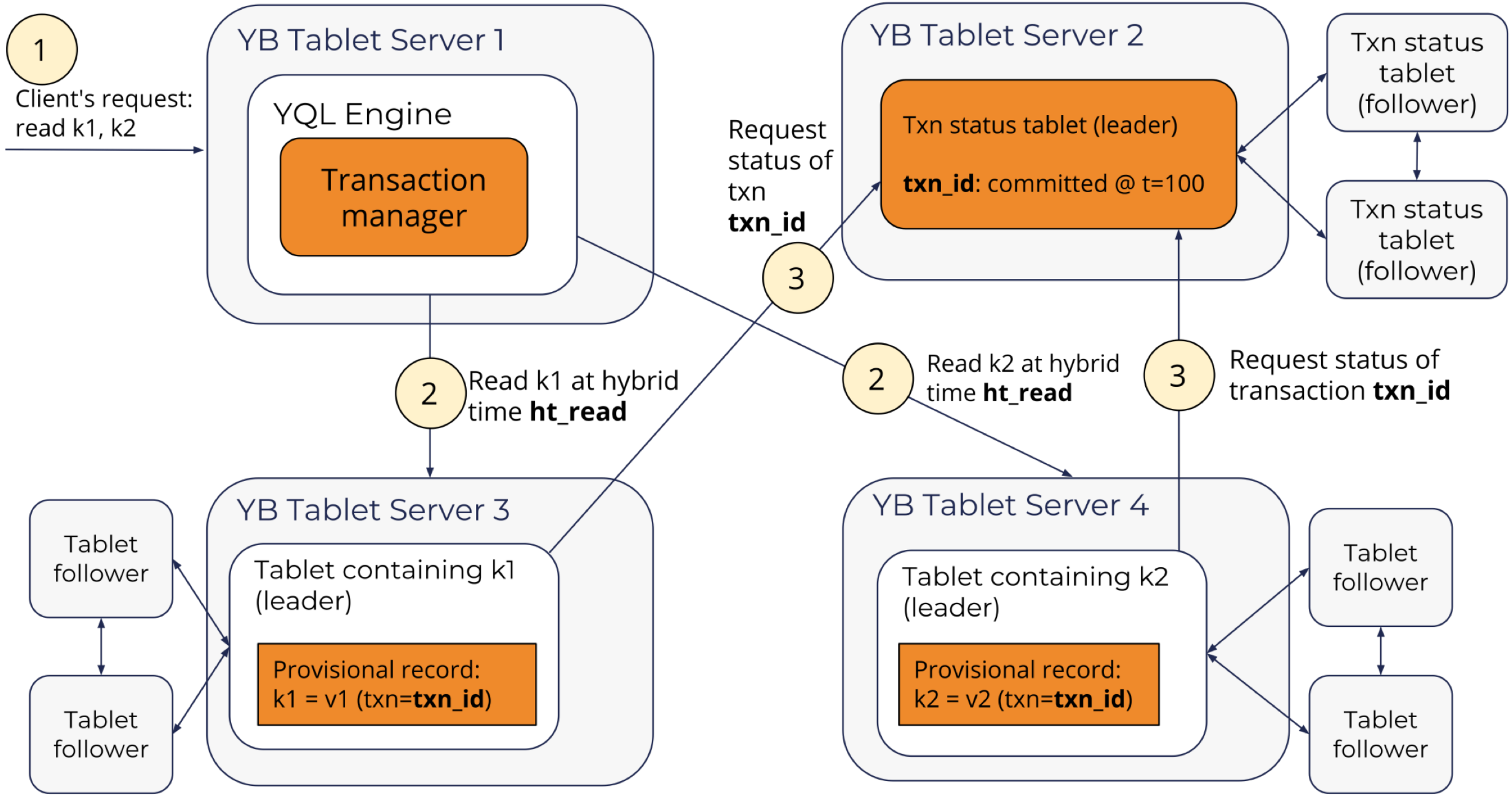
Distributed Transactions – Read Path Step 1: Client request; pick ht_read



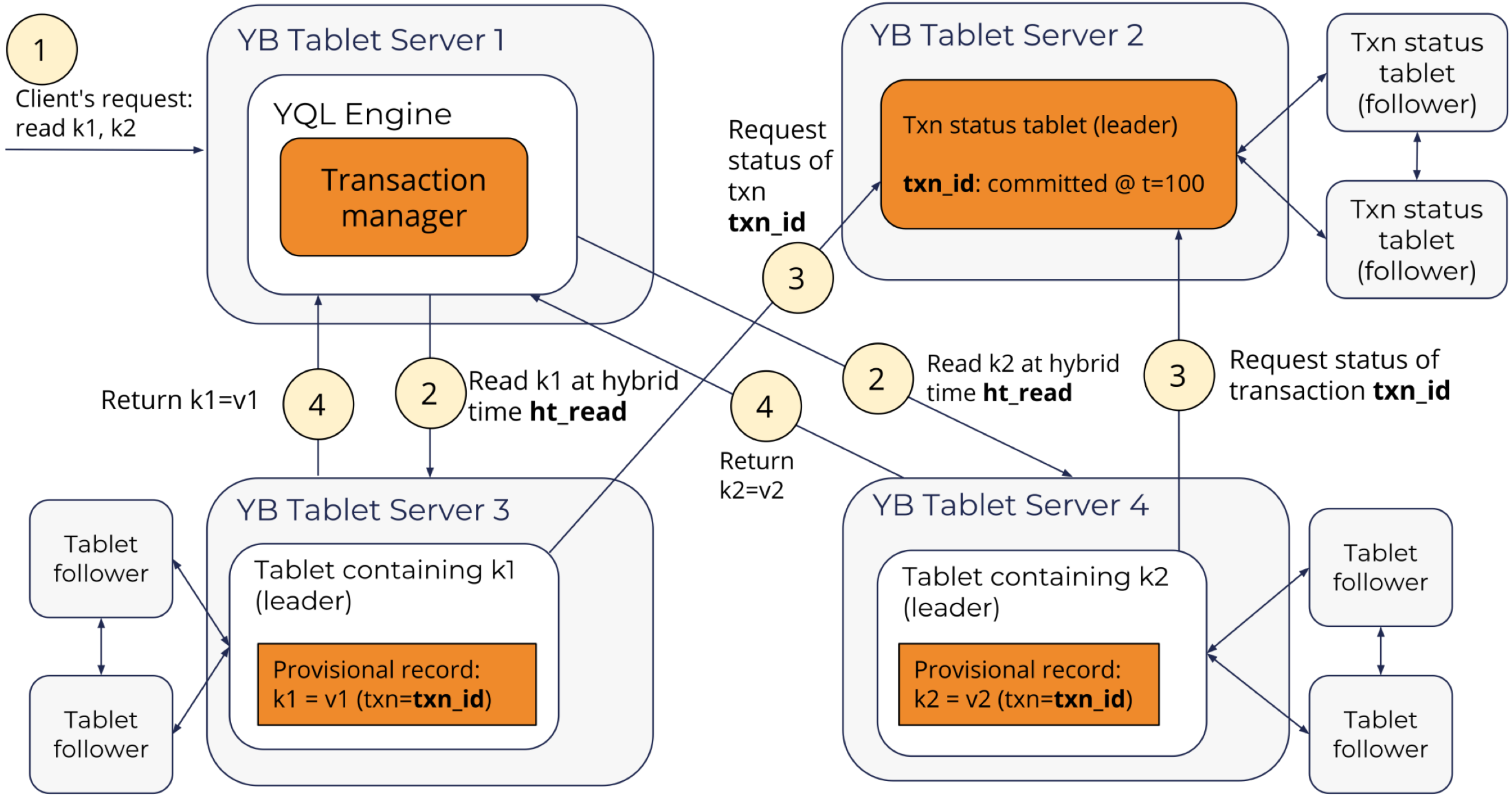
Distributed Transactions – Read Path Step 2: Read from tablet servers



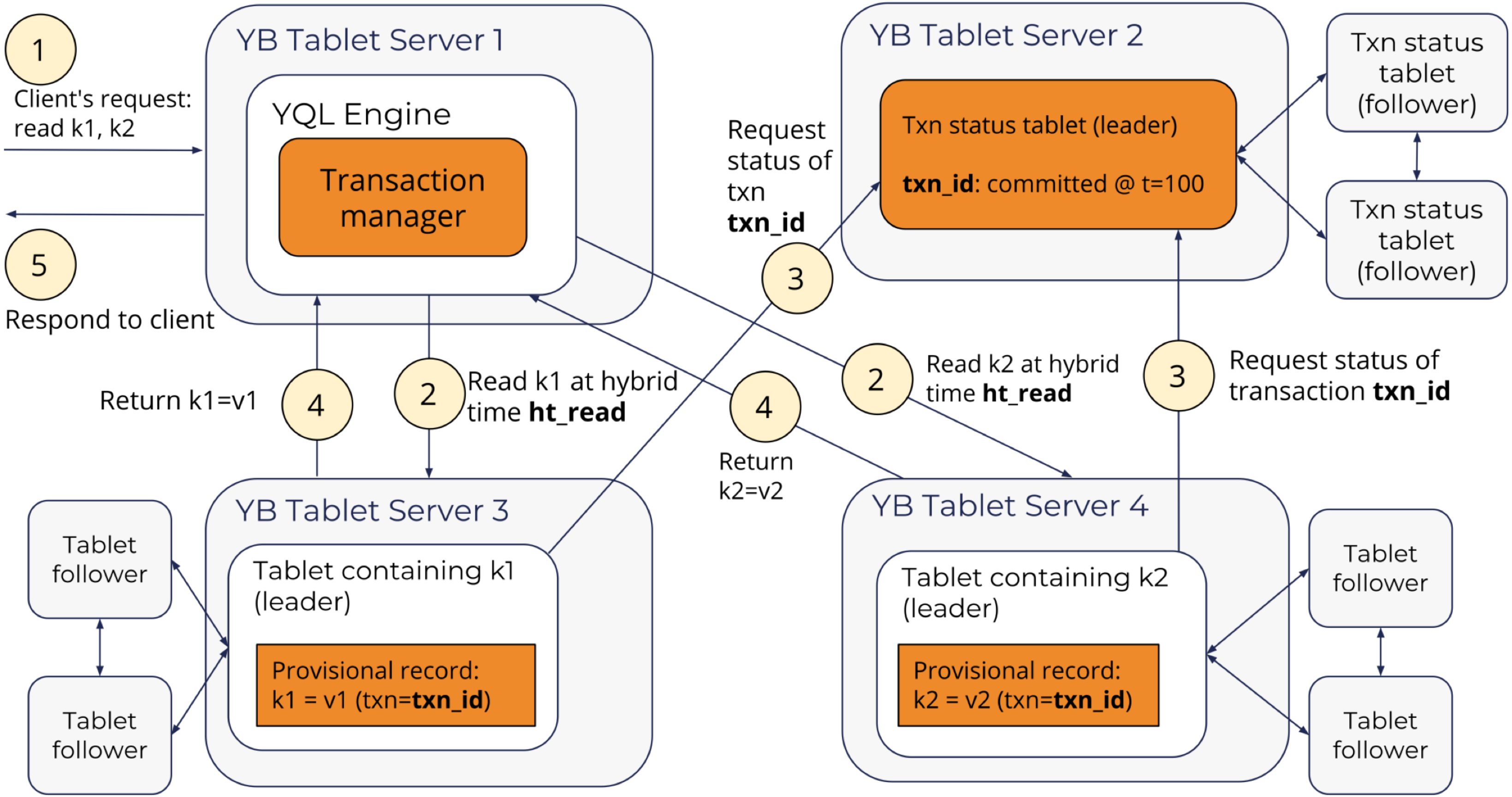
Distributed Transactions – Read Path Step 3: Resolve txn status



Distributed Transactions – Read Path Step 4: Respond to YQL Engine



Distributed Transactions – Read Path Step 5: Respond to client



Distributed Transactions – Conflicts & Retries

- Every transaction is assigned a random priority

- In a conflict, the higher-priority transaction wins
  - The restarted transaction gets a new random priority
  - Probability of success quickly increases with retries

- Restarting a transaction is the same as starting a new one

- A read-write transaction can be subject to read-restart

EXERCISE #3 and #4

SHARDING AND SCALE OUT
FAULT TOLERANCE

Questions?

Try it at

docs.yugabyte.com/latest/quick-start