Extending PostgreSQL to Google Spanner Architecture

YugabyteDB

Karthik Ranganathan,
Co-founder/CTO

Introduction to YugabyteDB

What is Distributed SQL?

A Revolutionary Database Architecture

SQL & Transactions

Ultra Resilience

Massive Scalability

Geo Distribution

yugabyteDB = distributed SQL database

\+ high performance (low Latency)

\+ cloud native (run on Kubernetes, VMs, bare metal)

\+ open source (Apache 2.0)

Fastest Growing Distributed SQL Project!

Growth in 1 Year

Powers business-critical apps at scale.

Plume	xignite	CIPHERTRACE	narvar
27B+	10B+	3B+	1B+
Ops/Day	Ops/Day	Ops/Day	Ops/Day

We 💛 stars! Give us one:

github.com/YugaByte/yugabyte-db

Join our community:

yugabyte.com/slack

Architecture

SQL Design Goals

- PostgreSQL compatible
  - Re-uses PostgreSQL query layer
  - New changes do not break existing PostgreSQL functionality
- Enable migrating to newer PostgreSQL versions
  - New features are implemented in a modular fashion
  - Integrate with new PostgreSQL features as they are available
  - E.g. Moved from PostgreSQL 10.4 → 11.2 in 2 weeks!
- Cloud native architecture
  - Fully decentralized to enable scaling to 1000s of nodes
  - Tolerate rack/zone and datacenter/region failures automatically
  - Run natively in containers and Kubernetes
  - Zero-downtime rolling software upgrades and machine reconfig

Spanner design + Aurora-like Compatibility

Feature	Amazon Aurora	Google Spanner	yugabyteDB
Horizontal Write Scalability	✗	✓	✓
Fault Tolerance with HA	✓	✓	✓
Globally Consistent Writes	✗	✓	✓
Scalable Consistent Reads	✗	✓	✓
SQL Support	✓	✗	✓

Layered Architecture

YugaByte SQL (YSQL)

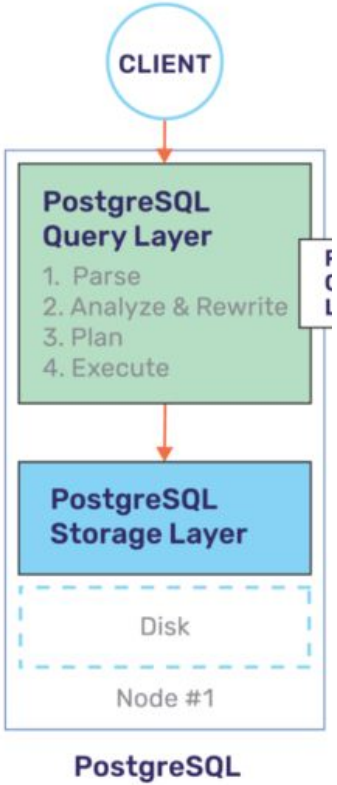
PostgreSQL-Compatible Distributed SQL API

DocDB

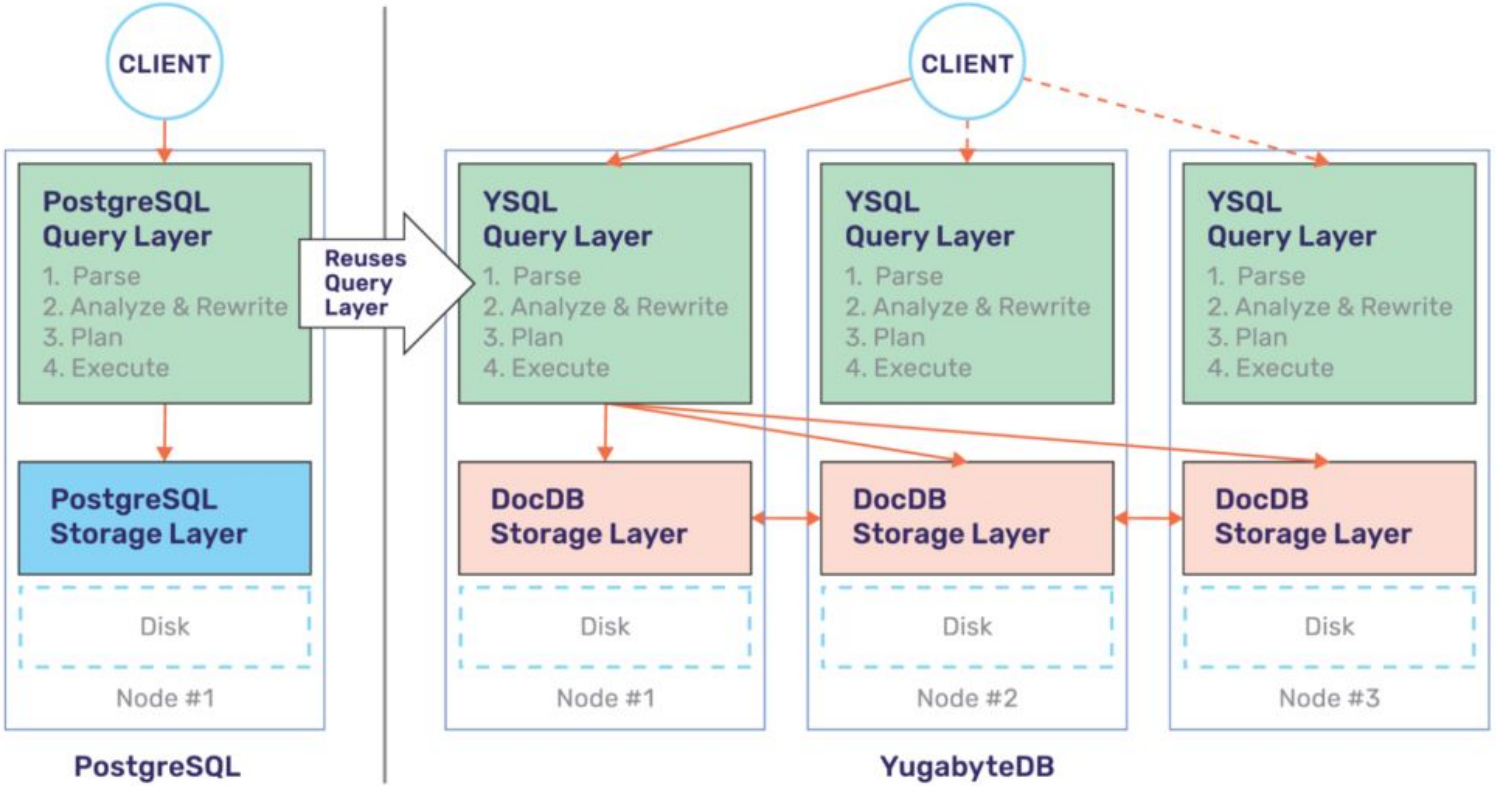
Spanner-Inspired Distributed Document Store

Cloud Neutral: No Specialized Hardware Needed

Single-Node PostgreSQL



Phase #1 - Extend to Distributed SQL



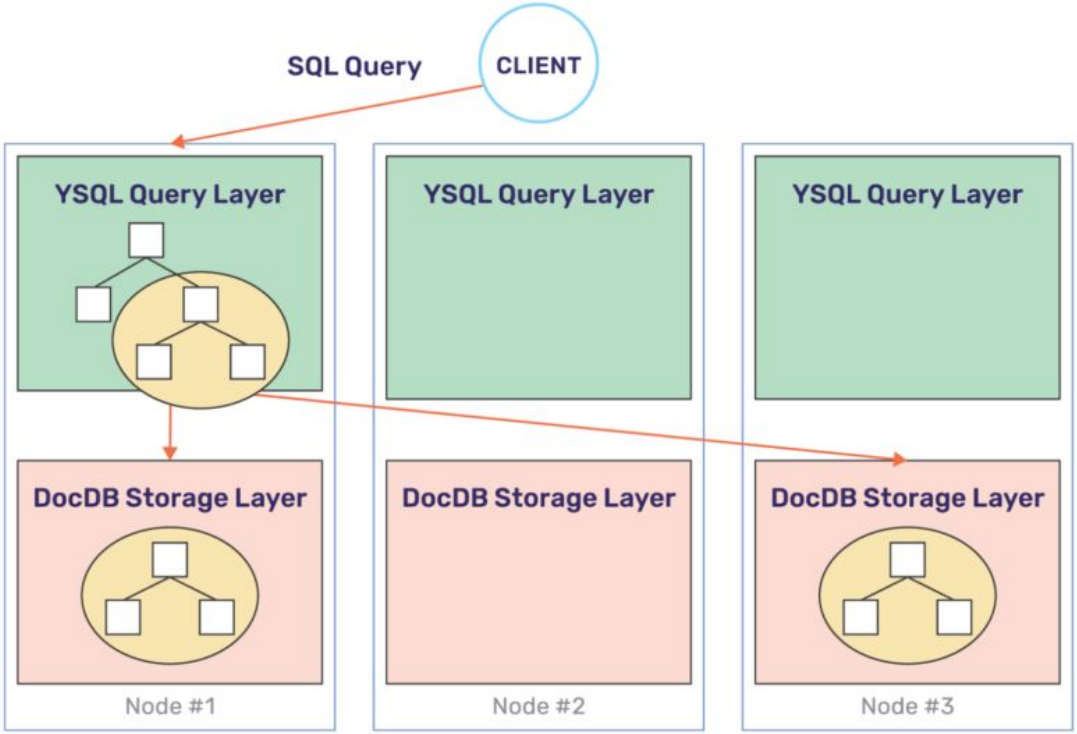
Phase #2: Perform More SQL Pushdowns

1. Build PostgreSQL execution tree

2. Identify subtree to pushdown

3. Identify nodes to send pushdown to

4. Execute subtree using PostgreSQL libraries on DocDB storage layer



Generic pushdown mechanism in YugabyteDB

Phase #3: Enhance Optimizer

- **Table statistics based hints**
  - Piggyback on current PostgreSQL optimizer that uses table statistics
- **Geographic location based hints**
  - Based on “network” cost
  - Factors in network latency between nodes and tablet placement
- **Rewriting query plan for distributed SQL**
  - Extend PostgreSQL “plan nodes” for distributed execution

We'll focus only on phase #1

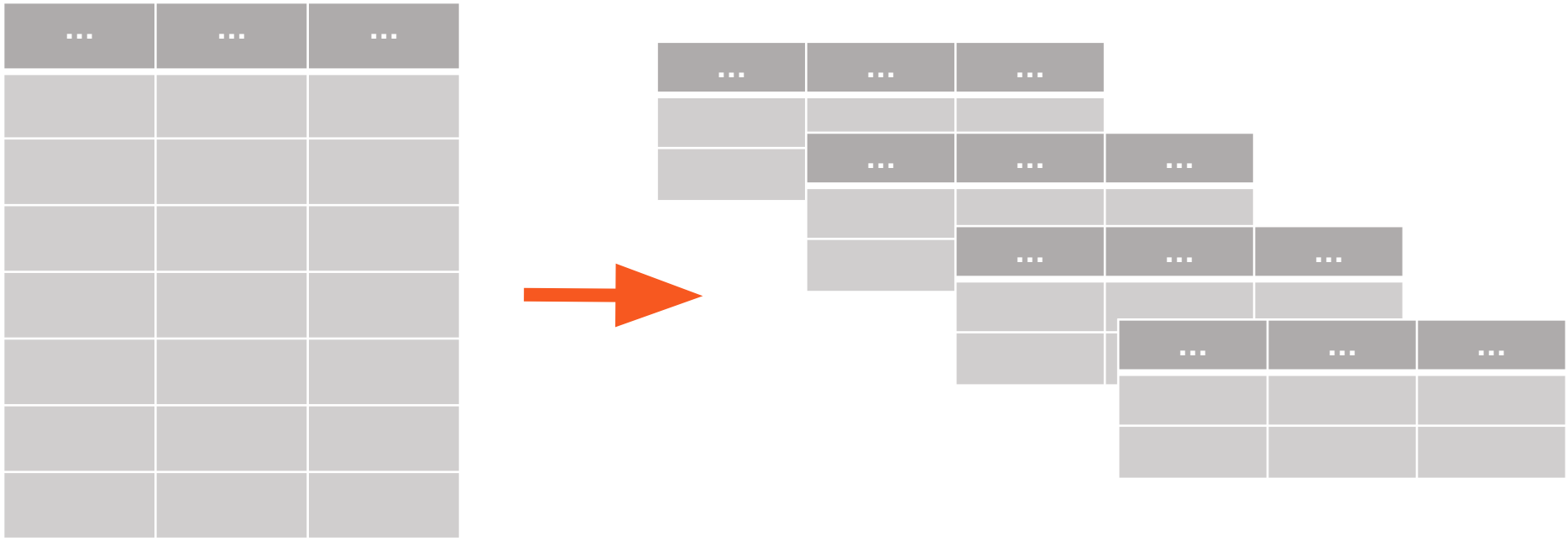
First look at storage layer (DocDB)

Then look at the query layer (YSQL)

DocDB Architecture

DocDB :
Sharding

Every Table is Automatically Sharded

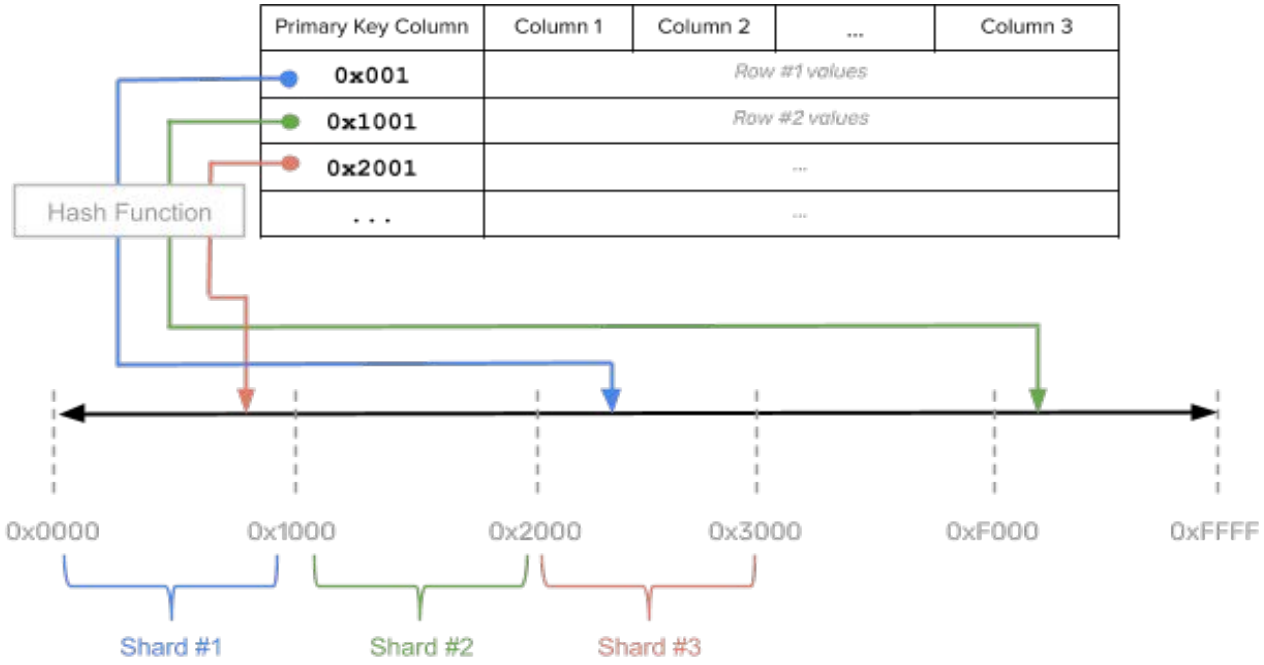


SHARDING = **AUTOMATIC PARTITIONING OF TABLES**

Supports multiple sharding strategies

- Hash Sharding
  - Ideal for massively scalable workloads
  - Distributes data evenly across all the nodes in the cluster
- Range Sharding
  - Efficiently query a range of rows by the primary key values
  - Example: look up all keys between a lower and an upper bound

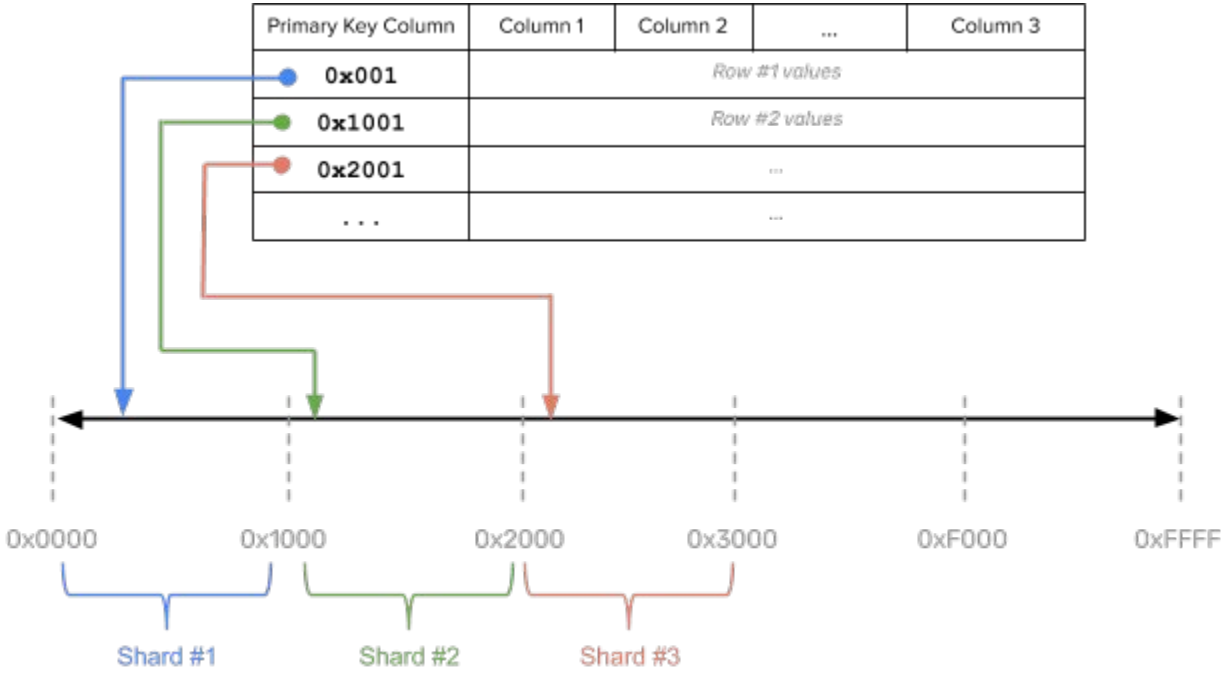
Hash Sharding



Hash Sharding Examples

```
CREATE TABLE customers (
    customer_id bpchar NOT NULL,
    company_name character varying(40) NOT NULL,
    contact_name character varying(30),
    contact_title character varying(30),
    address character varying(60),
    city character varying(15),
    region character varying(15),
    postal_code character varying(10),
    country character varying(15),
    phone character varying(24),
    fax character varying(24),
    PRIMARY KEY (customer_id HASH)
);
```

Range Sharding



Range Sharding Examples

```
CREATE TABLE order_details (
    order_id smallint NOT NULL,
    product_id smallint NOT NULL,
    unit_price real NOT NULL,
    quantity smallint NOT NULL,
    discount real NOT NULL,
    PRIMARY KEY (order_id ASC, product_id),
    FOREIGN KEY (product_id) REFERENCES products,
    FOREIGN KEY (order_id) REFERENCES orders
);
```

COPY

DocDB :
Replication

DocDB uses Raft consensus to replicate data

https://raft.github.io/

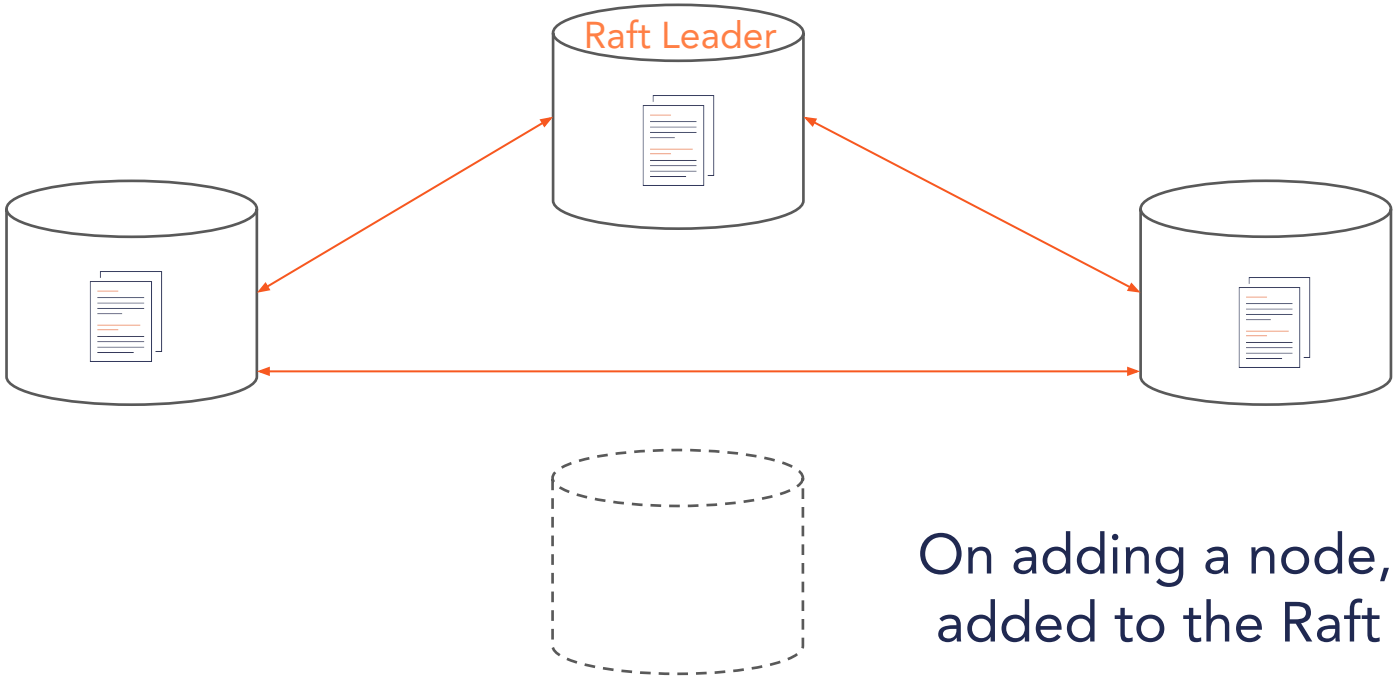
Raft vs Paxos

Easier to understand than Paxos

Dynamic membership changes (eg: change machine types)

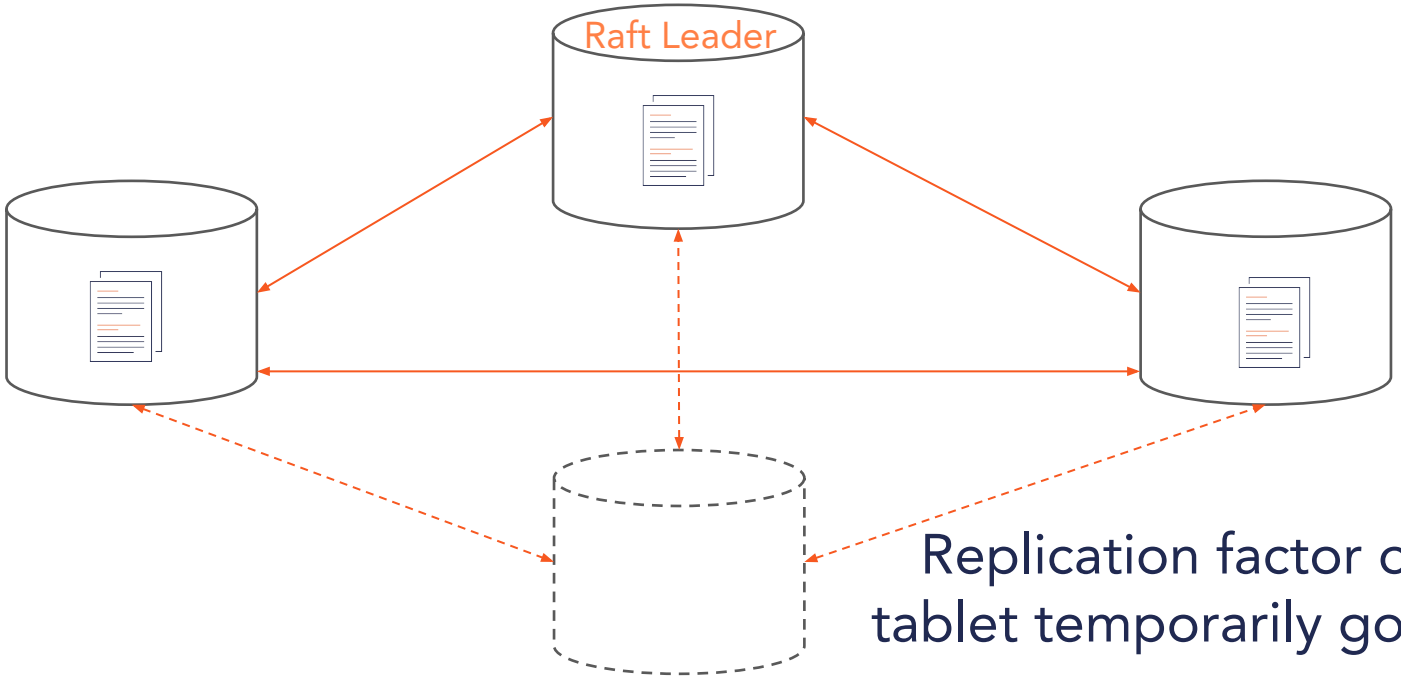
Heidi Howard will talk about “Raft vs Paxos”

Dynamic Membership Changes



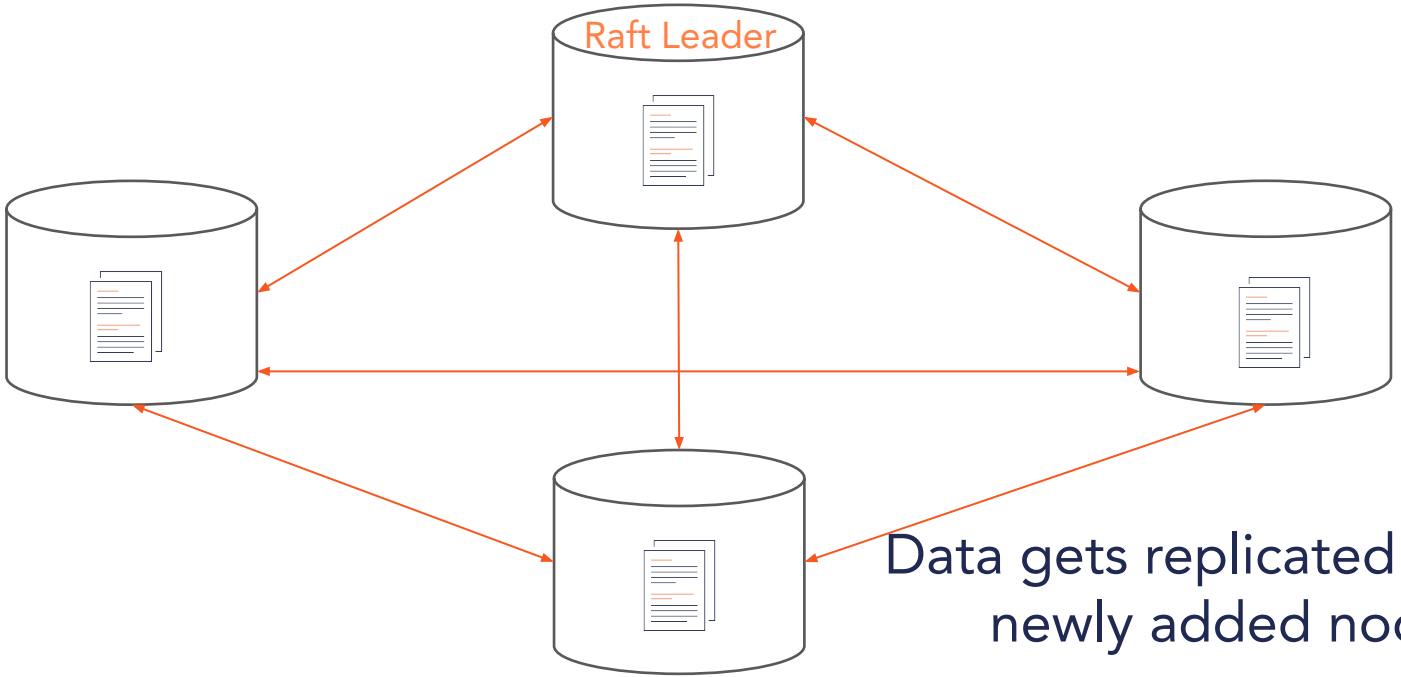
On adding a node, it gets added to the Raft group

Dynamic Membership Changes



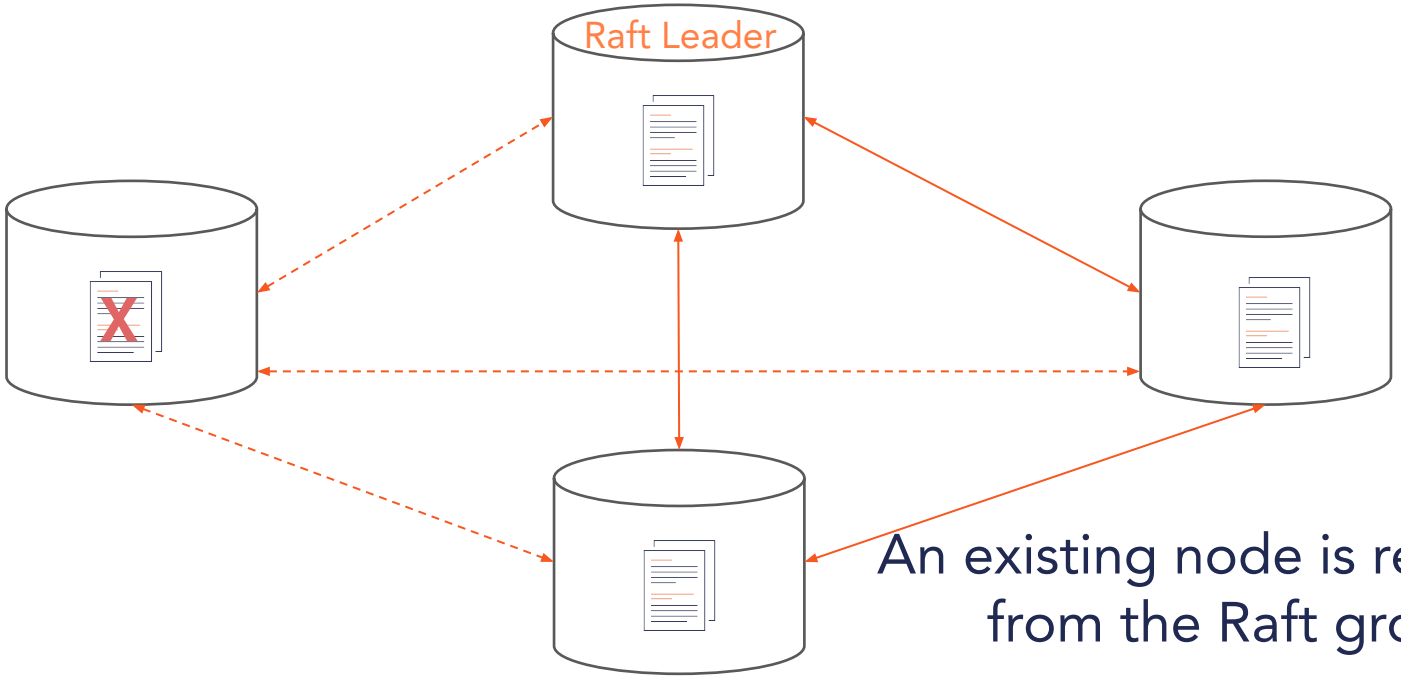
Replication factor of the tablet temporarily goes to 4

Dynamic Membership Changes



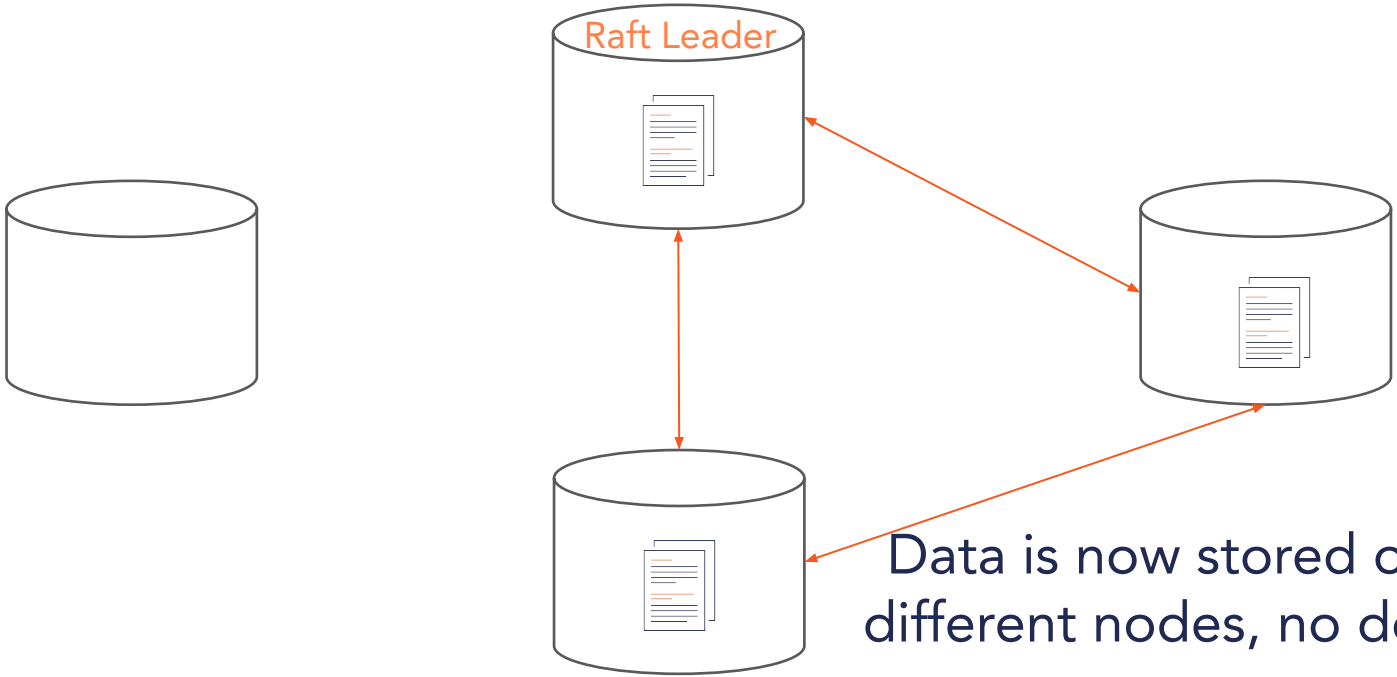
Data gets replicated to the newly added node

Dynamic Membership Changes



An existing node is removed from the Raft group

Dynamic Membership Changes



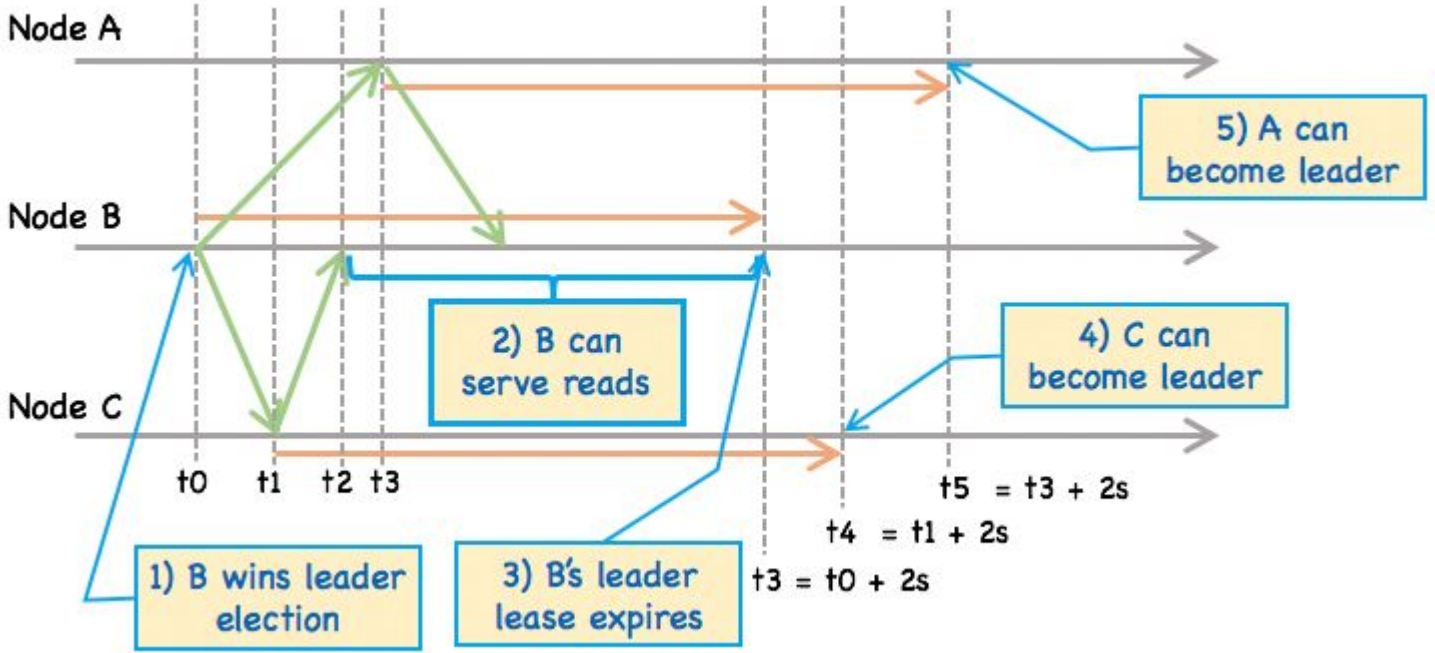
Many other performance enhancements needed!

Multi-region Raft reads are slow!

Each read requires a round-trip to majority

Round trip in multi-region = high latency

Leader Leases to the Rescue!

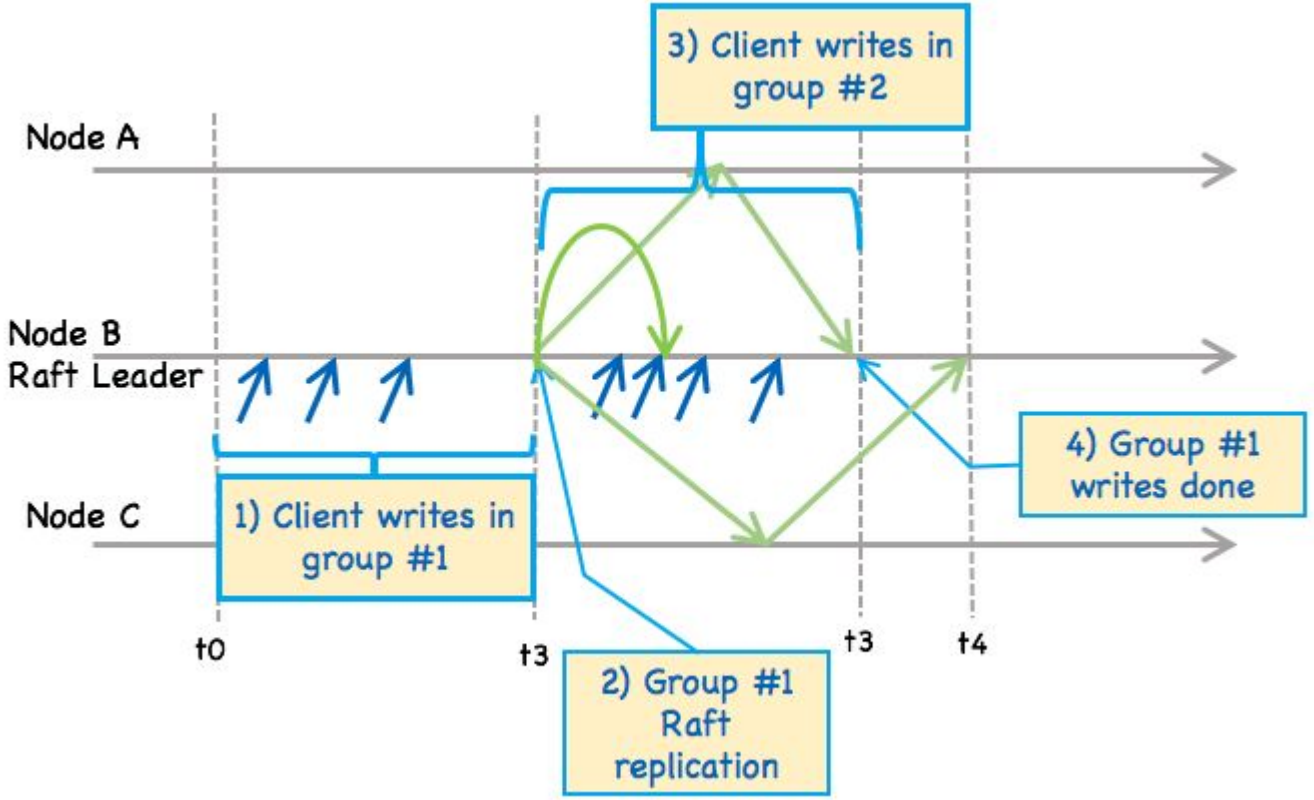


Monotonic Clocks

From Jepsen Testing Report:

Whatever the case, this is a good thing for operators: nobody wants to worry about clock safety unless they have to, and YugaByte DB appears to be mostly robust to clock skew. Keep in mind that we cannot (rigorously) test YugaByte DB's use of `CLOCK_MONOTONIC_RAW` for leader leases, but we suspect skew there is less of an issue than `CLOCK_REALTIME` synchronization.

Group Commits



DocDB :
Storage

Uses a heavily modified RocksDB

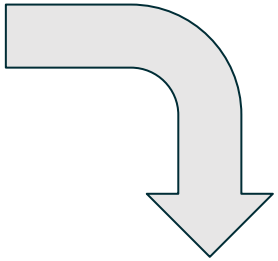
Key to document store

WAL log of RocksDB is not used (Raft log)

MVCC performed at a higher layer

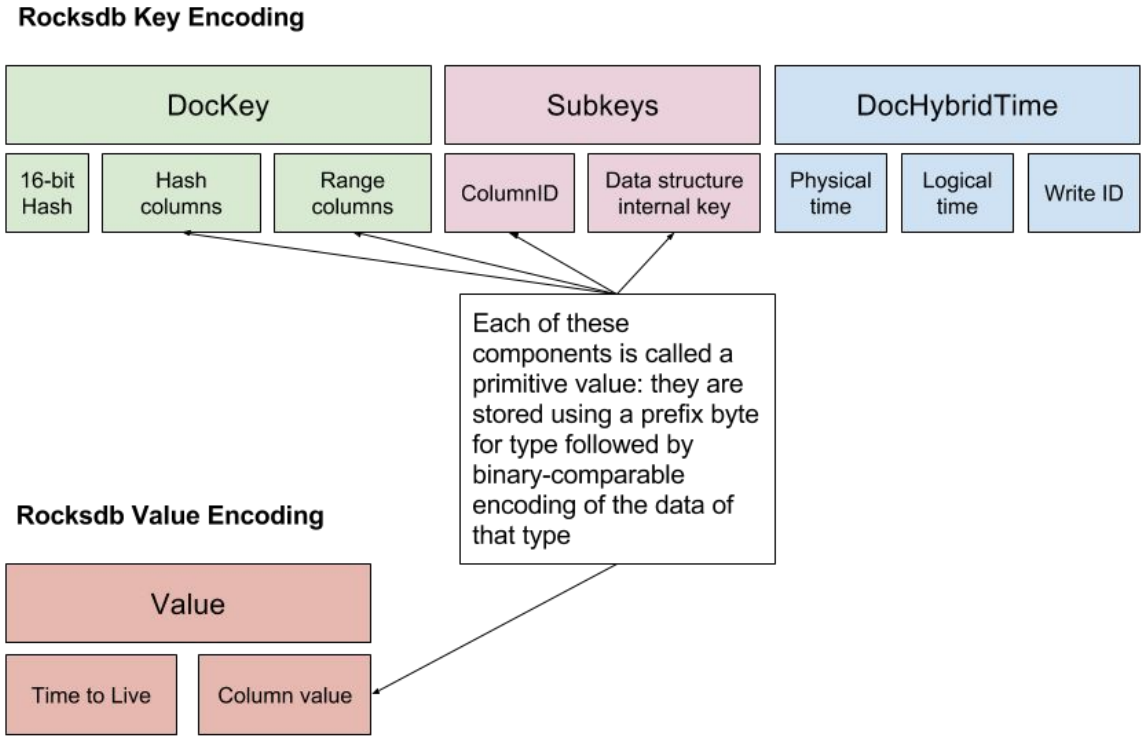
Logical Encoding of a Document

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

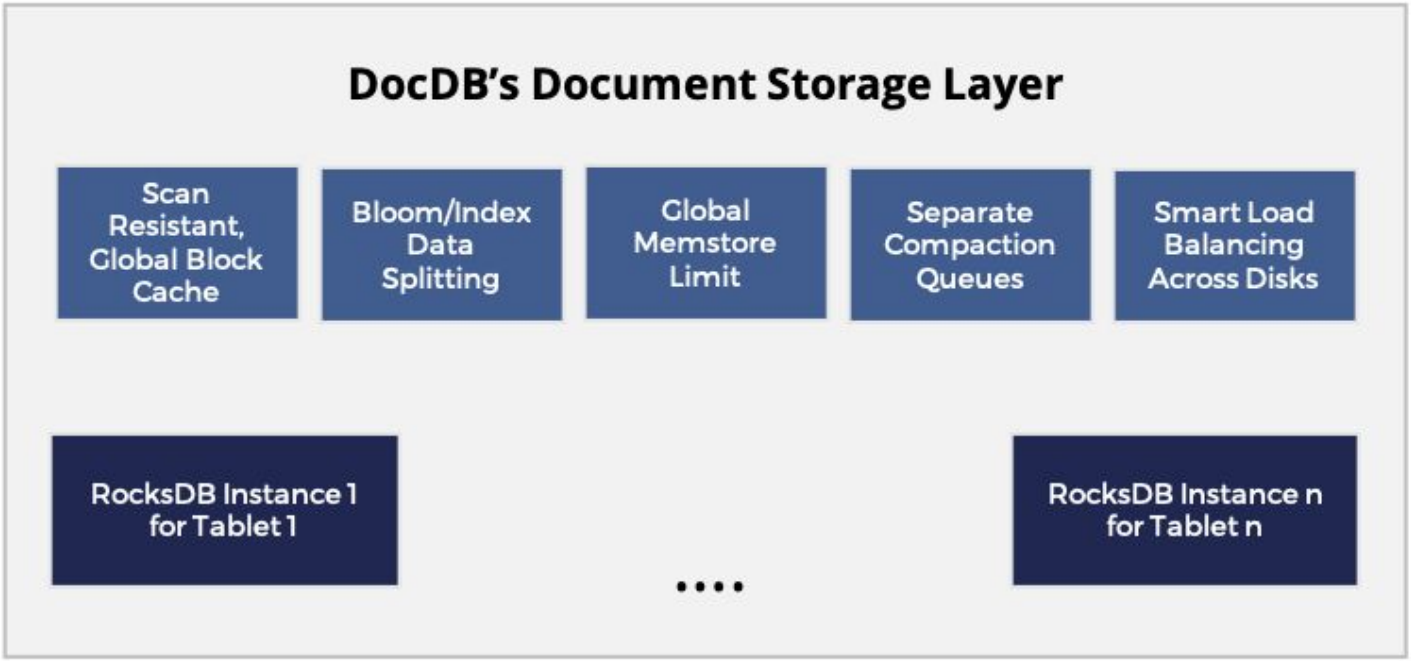


```
DocumentKey1, T10 -> {} // This is an init marker
DocumentKey1, SubKey1, T10 -> {}
DocumentKey1, SubKey1, SubKey2, T10 -> Value1
DocumentKey1, SubKey1, SubKey3, T10 -> Value2
DocumentKey1, SubKey4, T10 -> Value3
```

Encoding of DocDB data

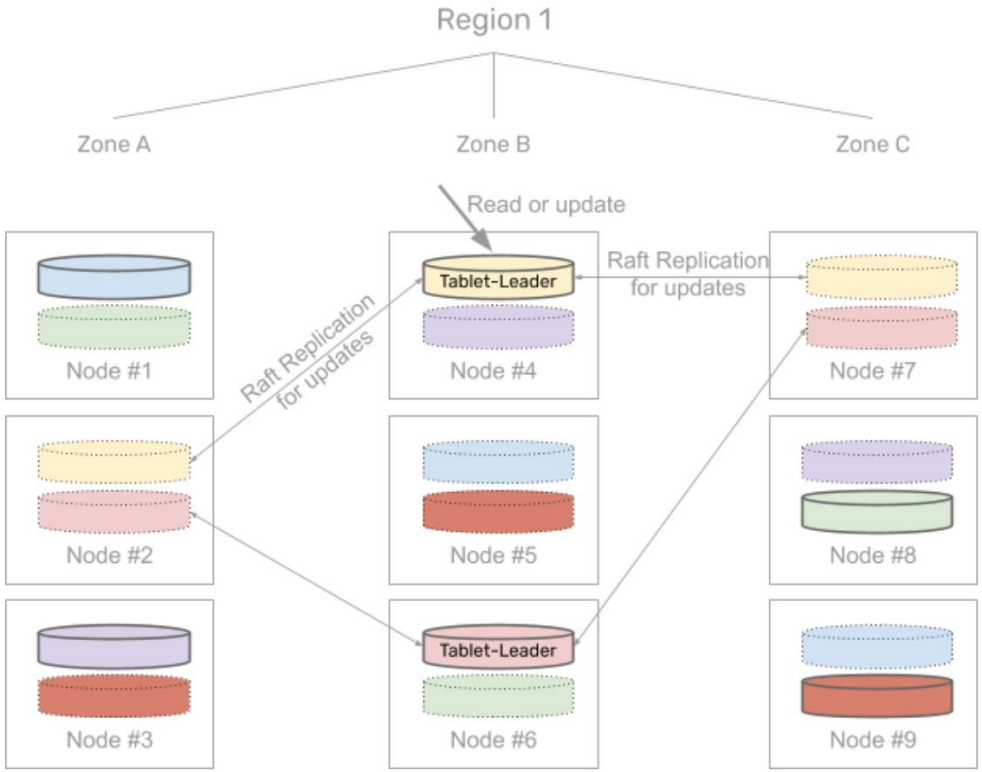


Many Performance Enhancements



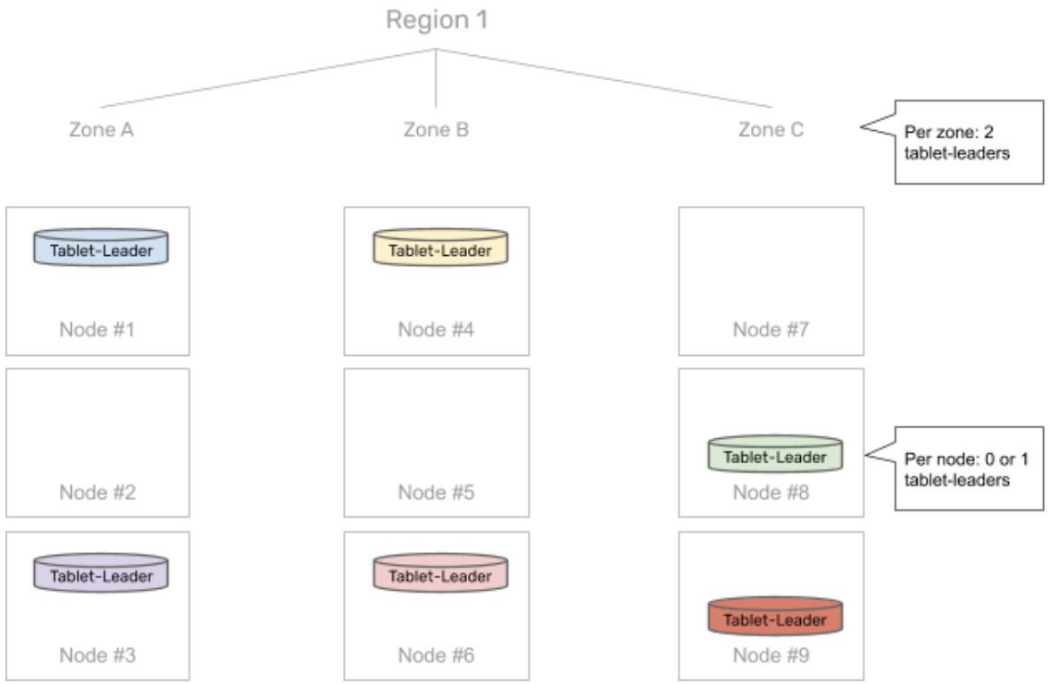
DocDB:
Deployment and Failover

Multi-Zone Deployment



- Multi-region is similar
- 6 tablets in table
- Replication = 3
- 1 replica per zone

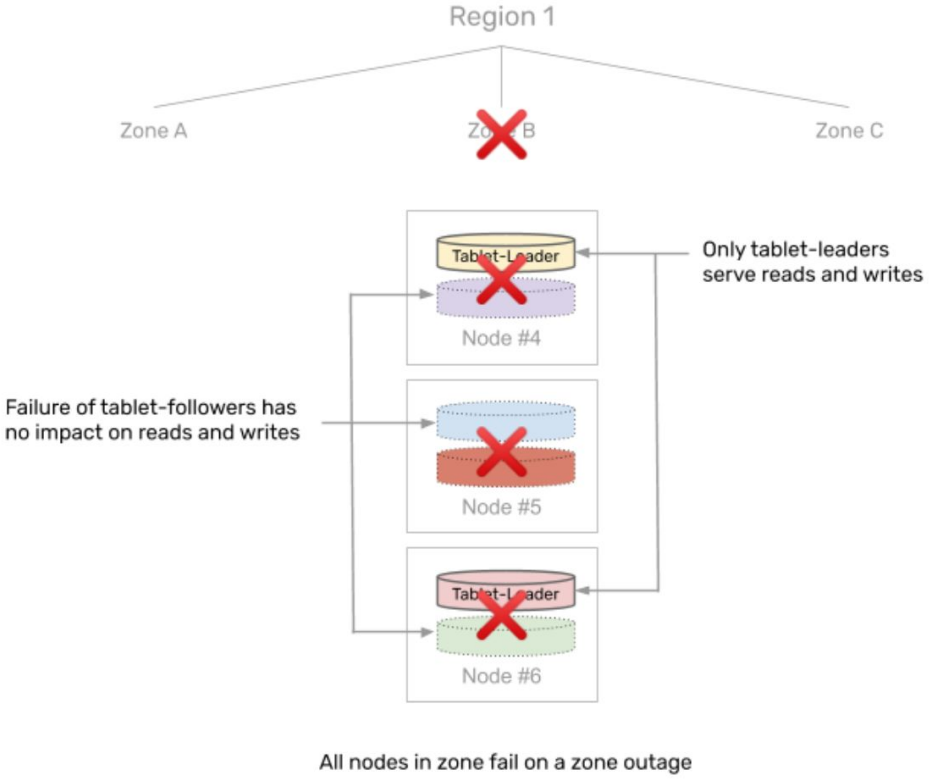
Balancing across zones and regions



Tablet leaders balanced across:

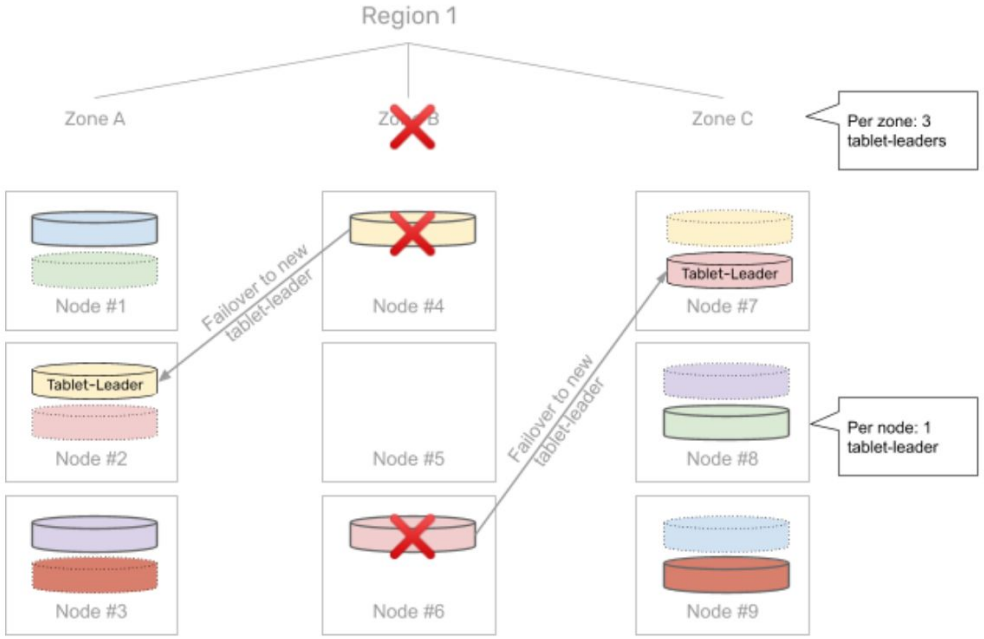
- Zones
- Nodes in a zone
- Per-table balancing

Tolerating Zone Outage



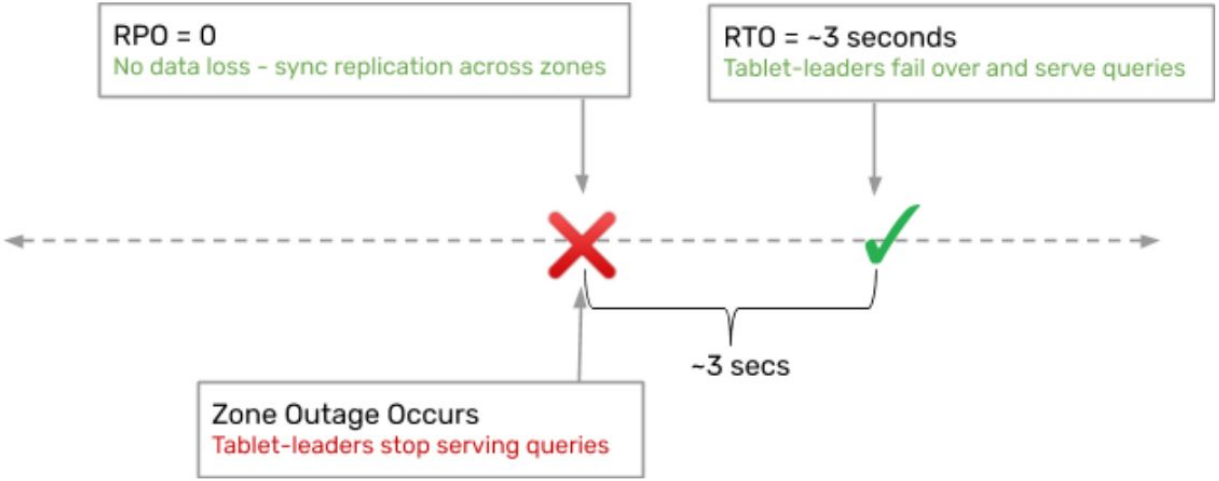
- New tablet leaders re-elected (~3 sec)
- No impact on tablet follower outage
- DB unavailable during re-election window
- Follower reads ok

Automatic rebalancing



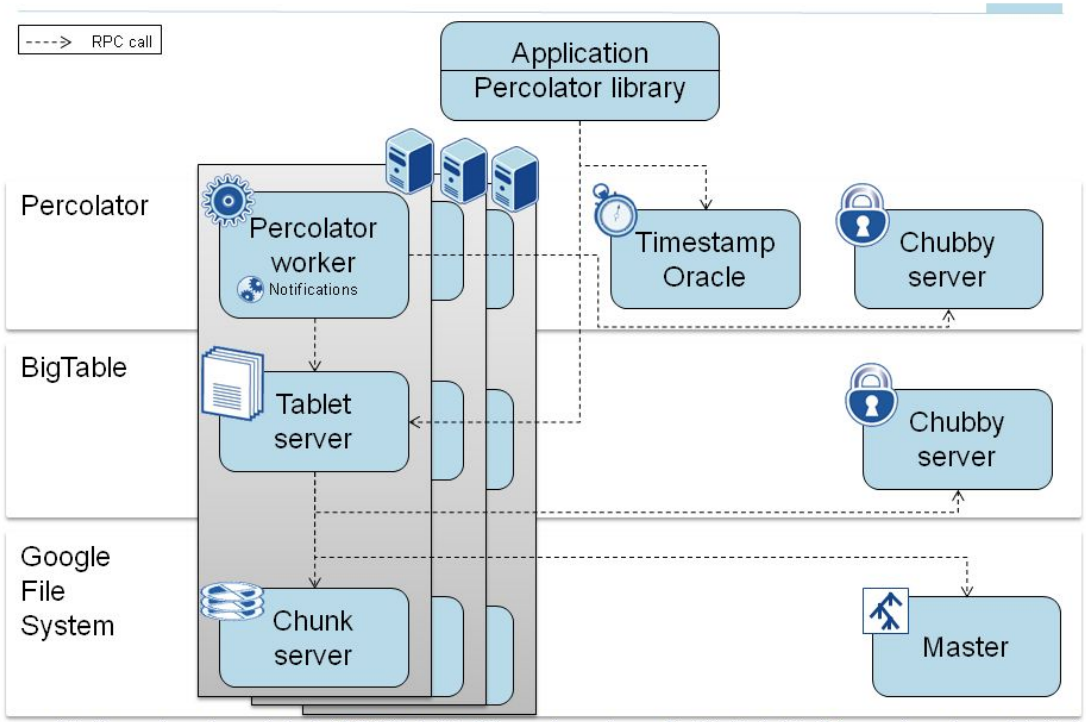
- New leaders evenly rebalanced
- After 15 mins, data is re-replicated (if possible)
- On failed node recovery, automatically catch up

How long to failover in multi-zone setup?



DocDB:
Distributed Transactions

Google Percolator



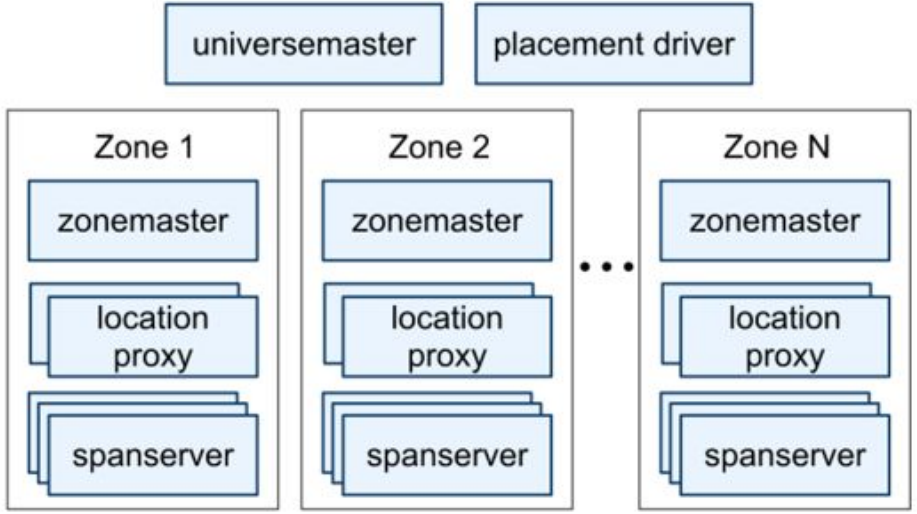
Sources: http://research.google.com/pubs/pub36726.html, http://labs.google.com/papers/bigtable.html, http://labs.google.com/papers/gfs.html

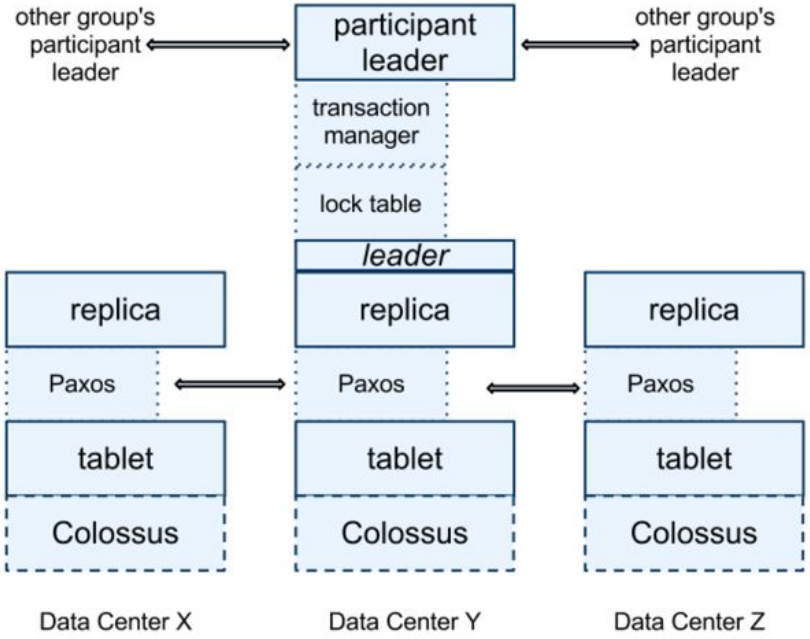
Percolator = Single Timestamp Oracle

Not scalable

Does not work for multi-region deployments

Google Spanner





Spanner = Distributed Time Synchronization

Scalable

Low-latency, multi-region deployments

2-phase commit

We picked Google Spanner like design for distributed transactions in YugabyteDB

YugabyteDB distributed transactions:

Based on 2-phase commit

Uses hybrid logical clocks

Fully Decentralized Architecture

- **No single point of failure or bottleneck**
  - Any node can act as a Transaction Manager
- **Transaction status table distributed across multiple nodes**
  - Tracks state of active transactions
- **Transactions have 3 states**
  - Pending
  - Committed
  - Aborted
- **Reads served only for Committed Transactions**
  - Clients never see inconsistent data

Isolation Levels

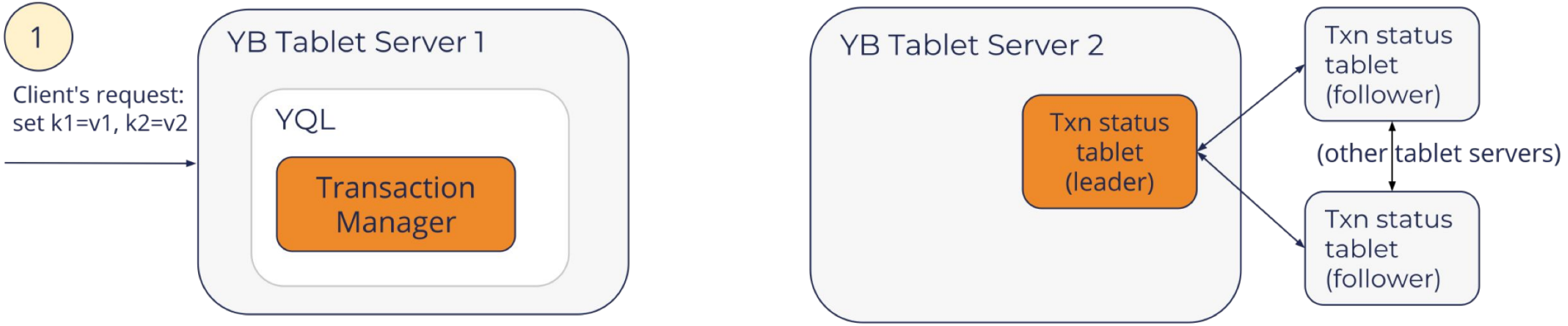
- **Serializable Isolation**
  - Read-write conflicts get auto-detected
  - Both reads and writes in read-write txns need provisional records
  - Maps to SERIALIZABLE in PostgreSQL

- **Snapshot Isolation**
  - Write-write conflicts get auto-detected
  - Only writes in read-write txns need provisional records
  - Maps to REPEATABLE READ, READ COMMITTED & READ UNCOMMITTED in PostgreSQL

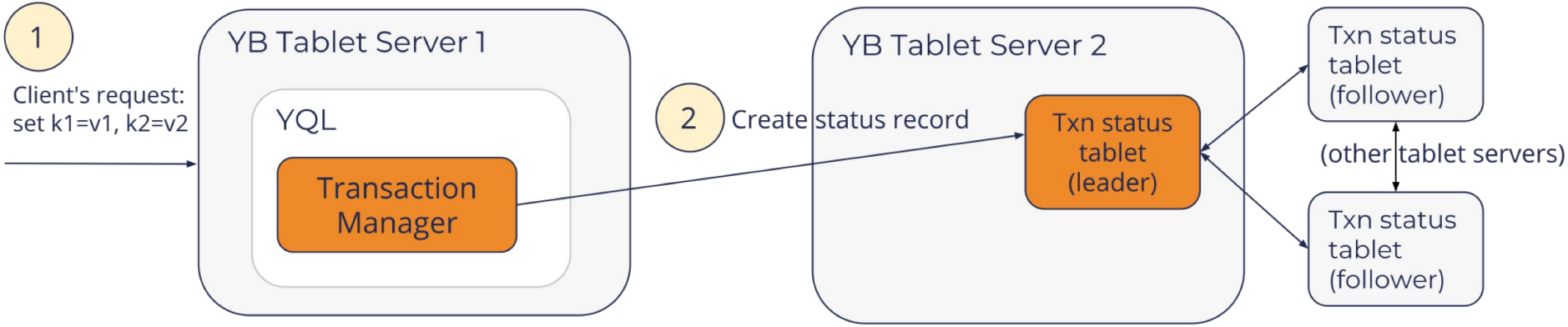
- **Read-only Transactions**
  - Lock free

Distributed Transactions - Write Path

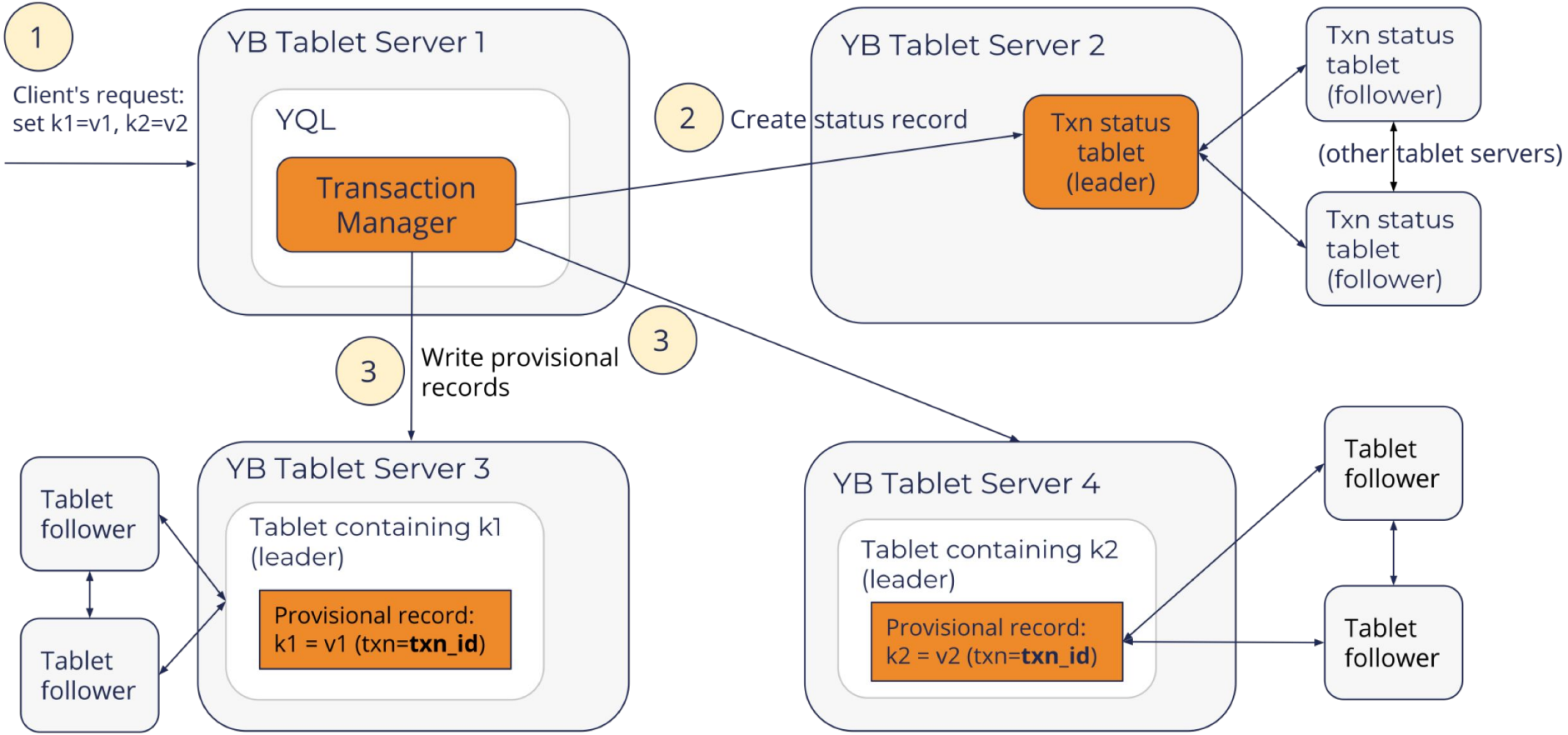
Distributed Transactions - Write Path



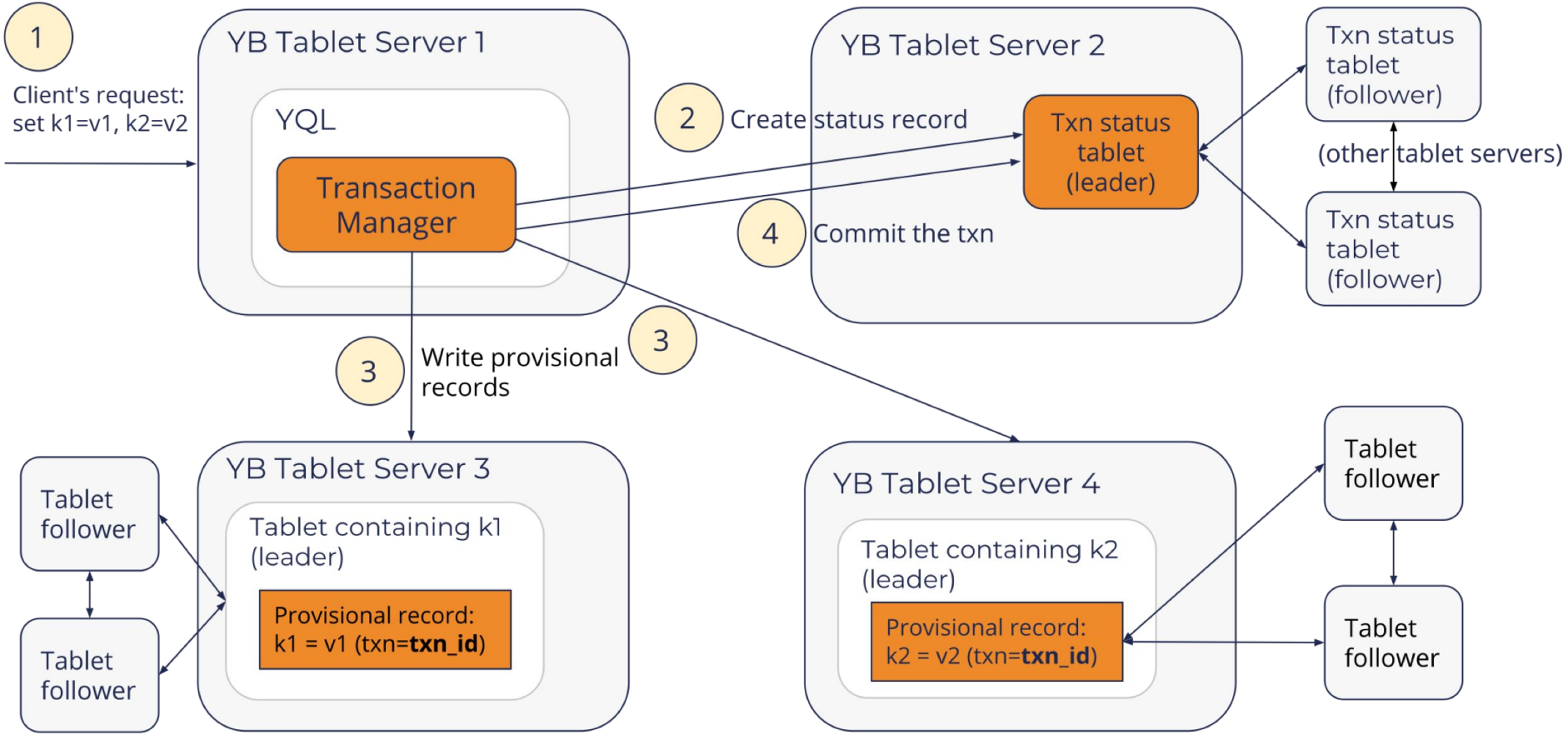
Distributed Transactions - Write Path



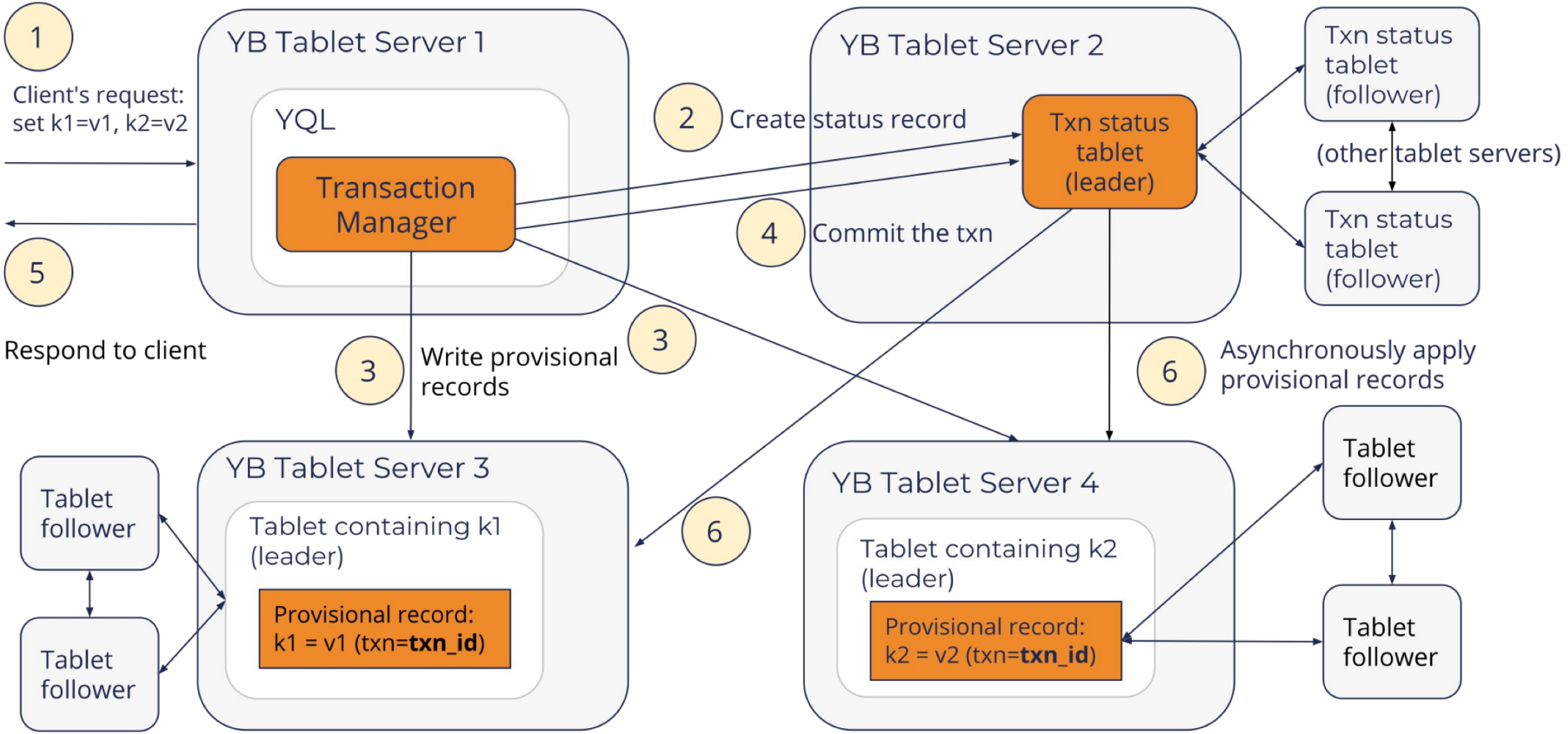
Distributed Transactions - Write Path



Distributed Transactions - Write Path



Distributed Transactions - Write Path



DocDB:
Software Defined Atomic Clocks

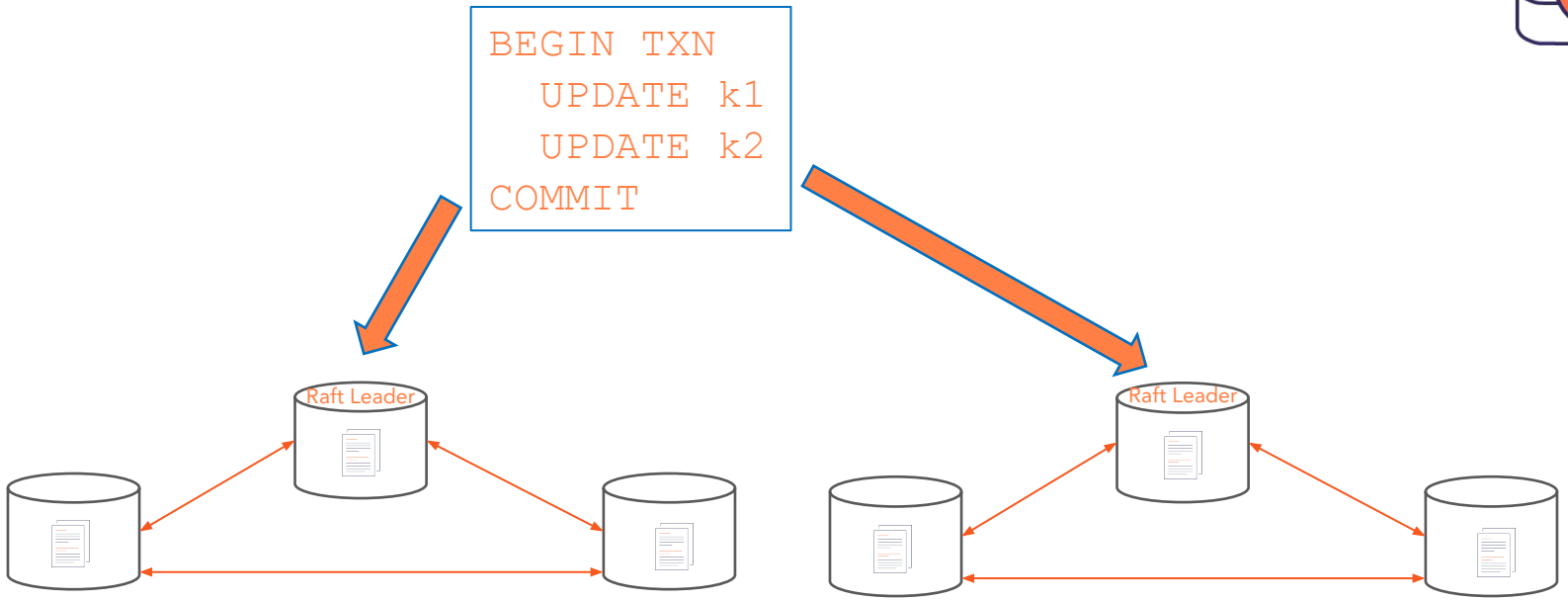
Distributed (aka Multi-Shard) Transactions

```
BEGIN TXN
  UPDATE k1
  UPDATE k2
COMMIT
```

k1 and k2 may belong to **different shards**

Belong to **different Raft groups** on completely **different nodes**

What do Distributed Transactions need?



Updates should get written at the same time

But how will nodes agree on time?

Atomic Clocks

You would need an Atomic Clock or two lying around

Atomic Clocks:
highly available, globally synchronized clocks, tight error bounds

Jeez! I'm fresh out of those.

Most of my physical clocks are never synchronized

Use a Physical Clock

You would need an Atomic Clock or two lying around

Atomic Clocks are highly available,
globally synchronized clocks with tight error bounds

Jeez! I'm fresh out of those.

Most of my physical clocks are never synchronized

Hybrid Logical Clock (HLC)

Combine coarsely-synchronized physical clocks with Lamport Clocks to track causal relationships

(physical component, logical component)

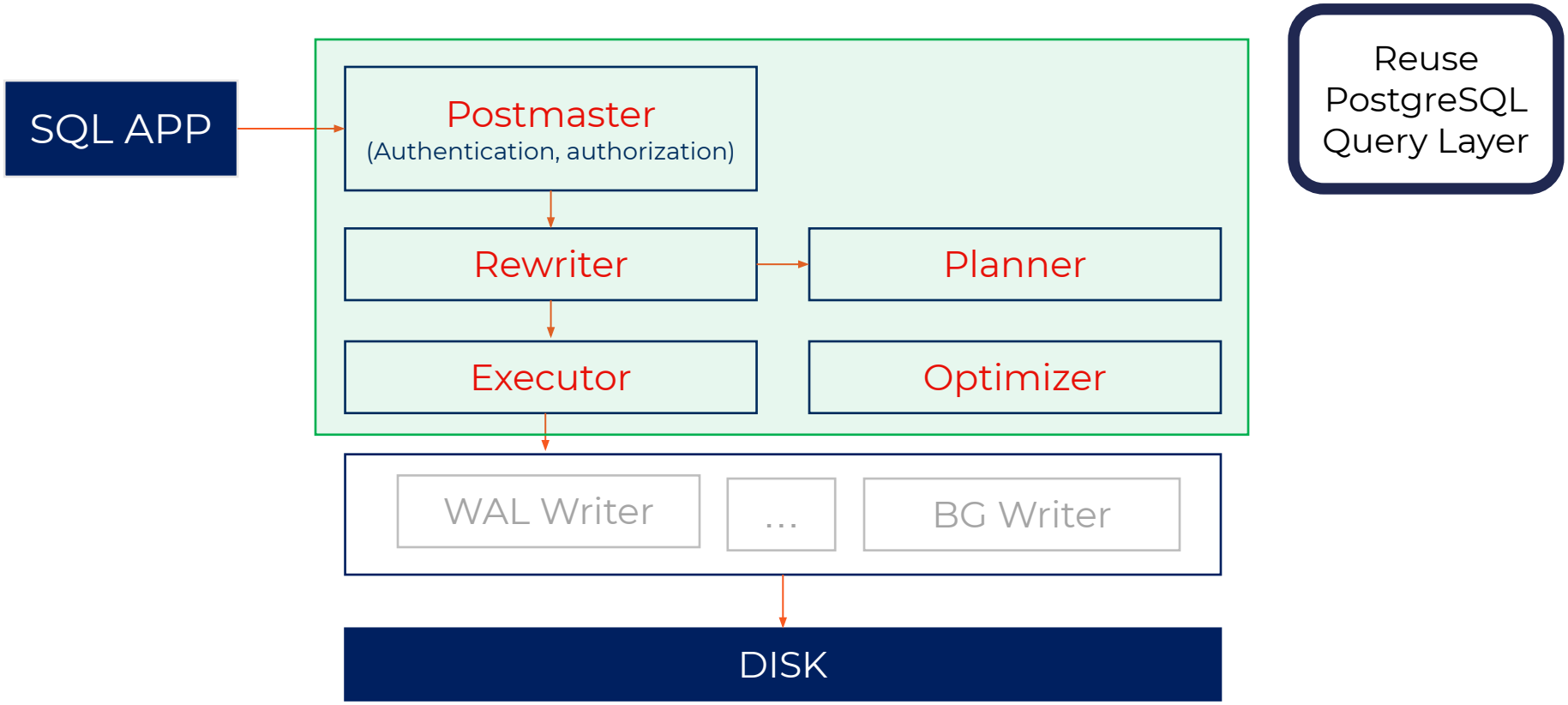
synchronized using NTP

a monotonic counter

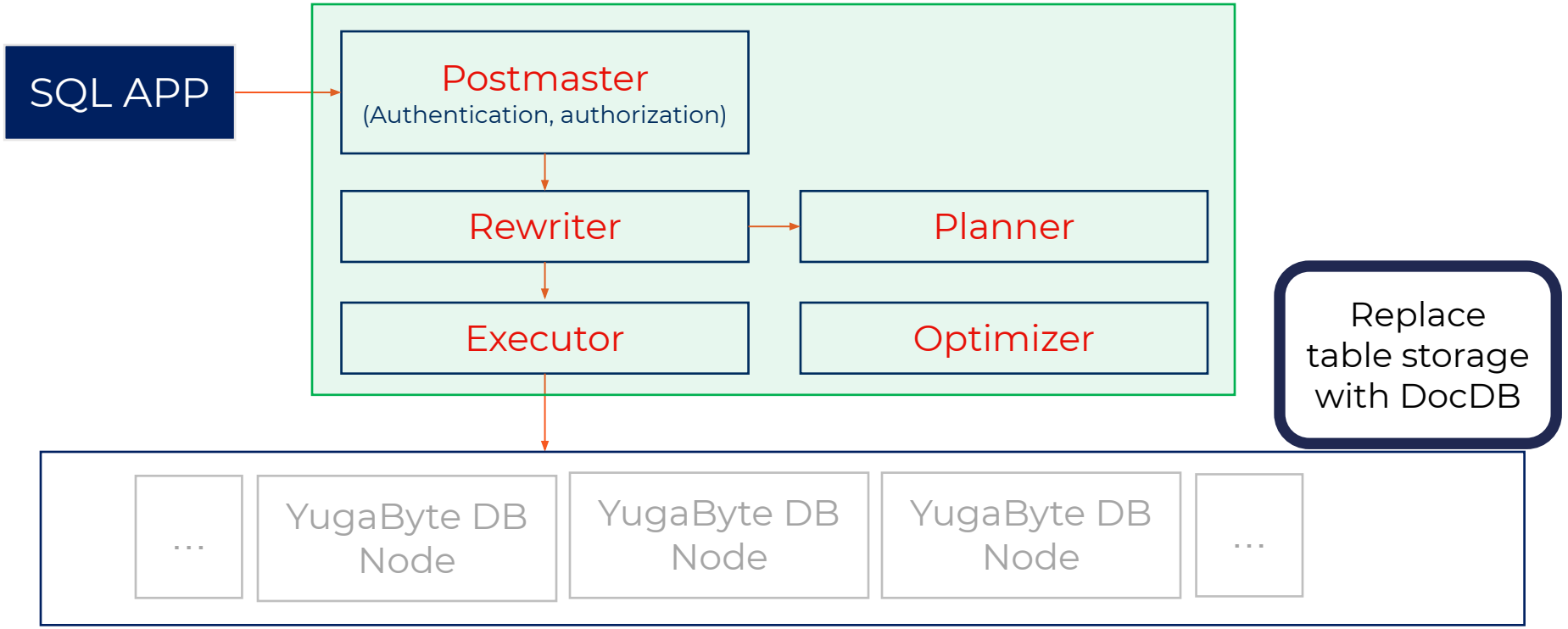
Nodes update HLC on each Raft exchange for things like heartbeats, leader election and data replication

YSQL Architecture

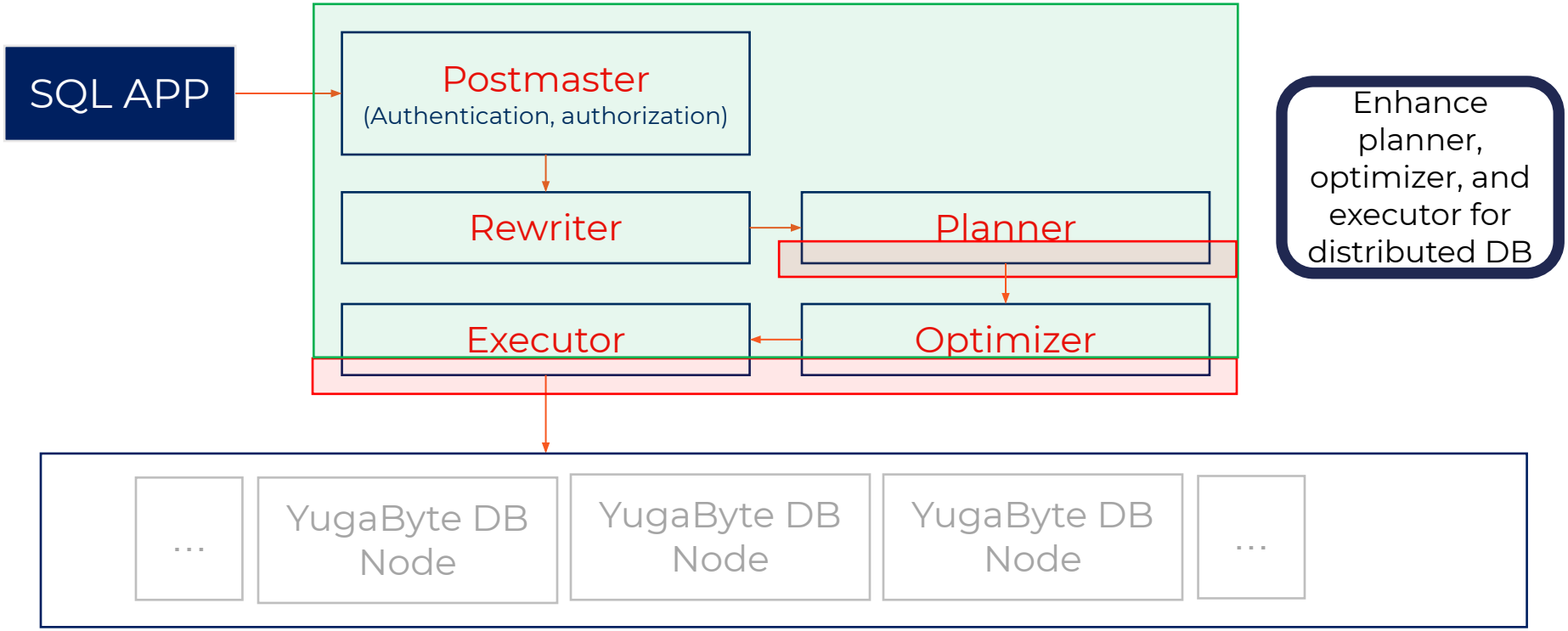
Existing PostgreSQL Architecture



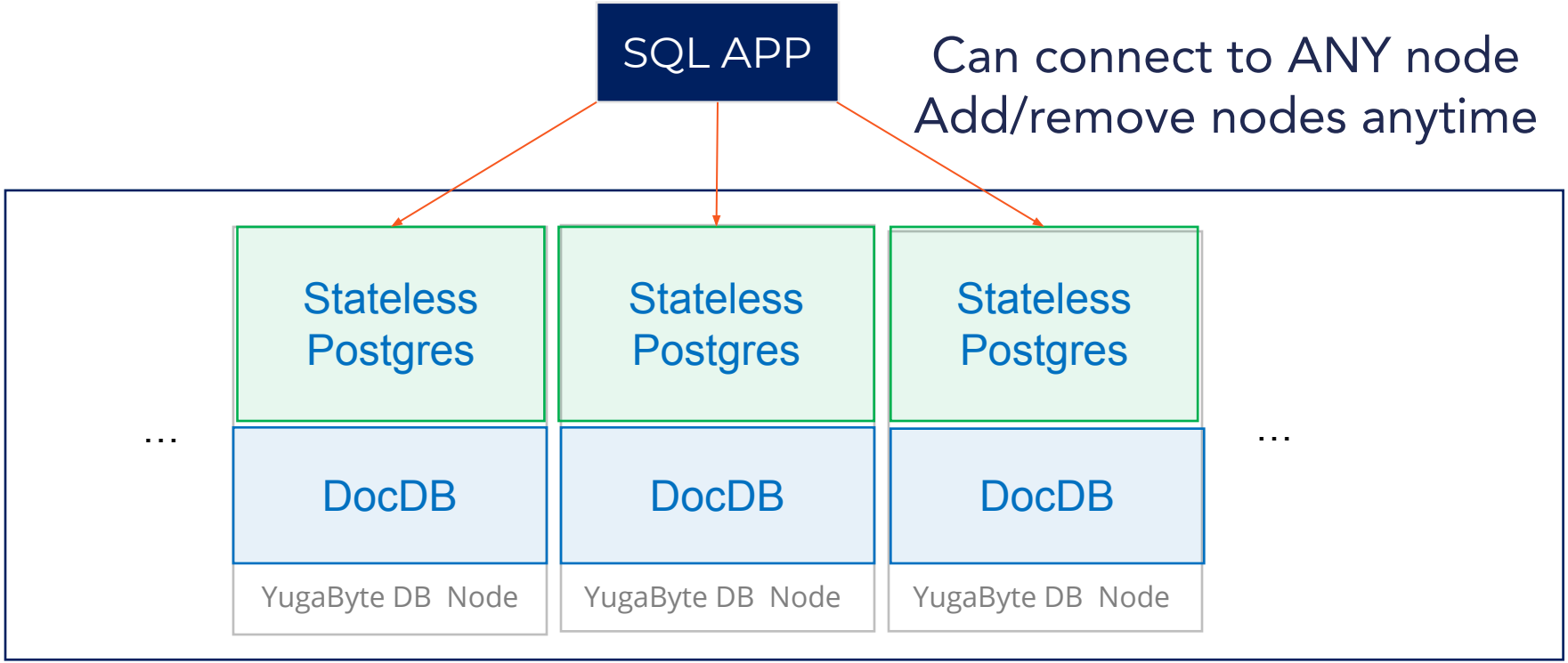
DocDB as Storage Engine



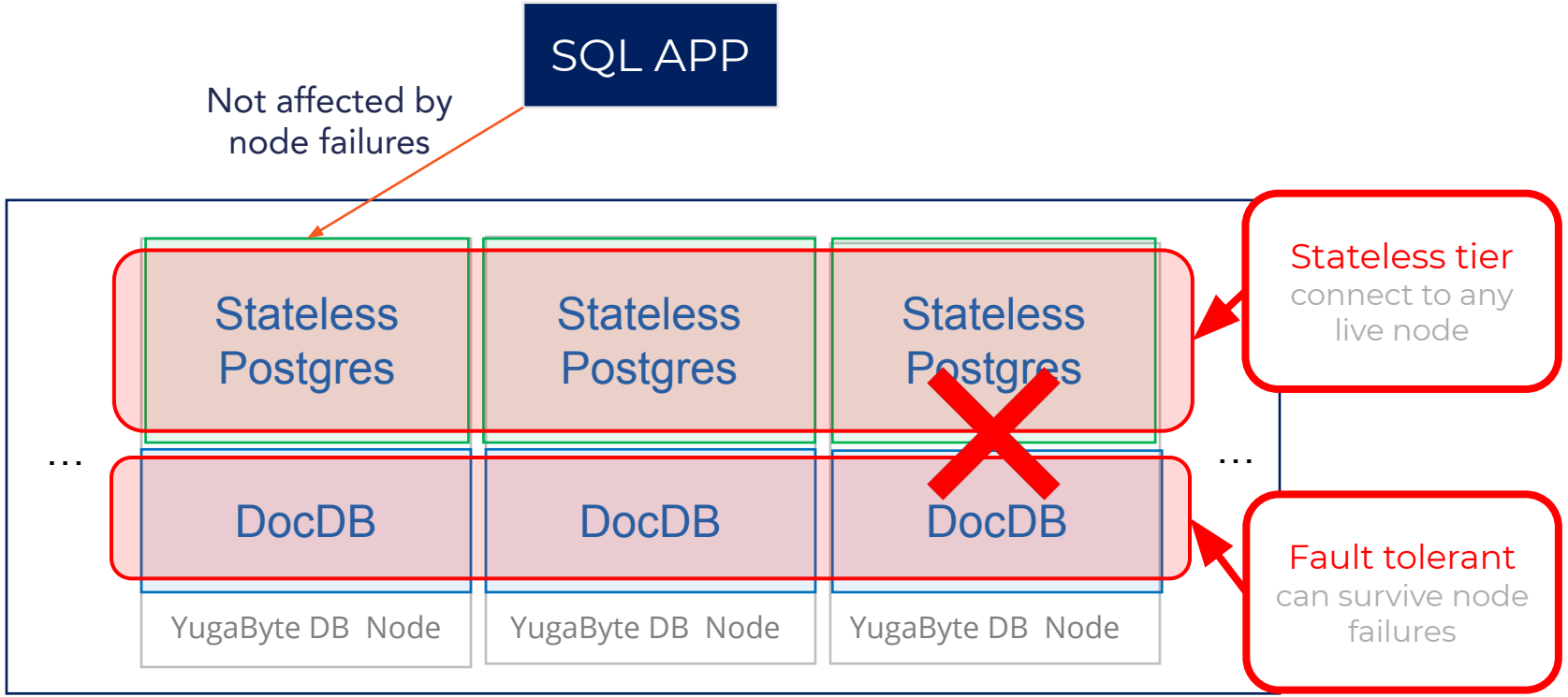
Make PostgreSQL Run on Distributed Store



All Nodes are Identical



Self-Healing Against Failures

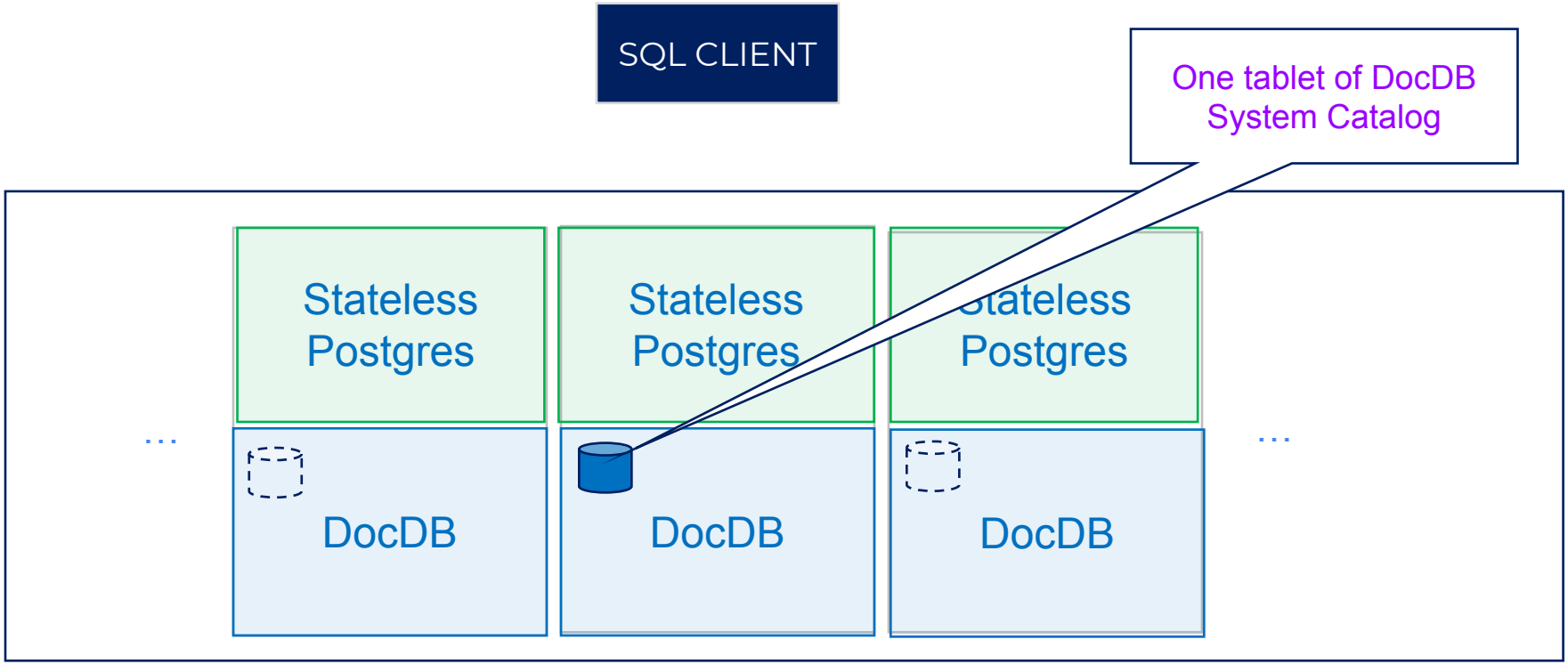


YSQL:
Create Table

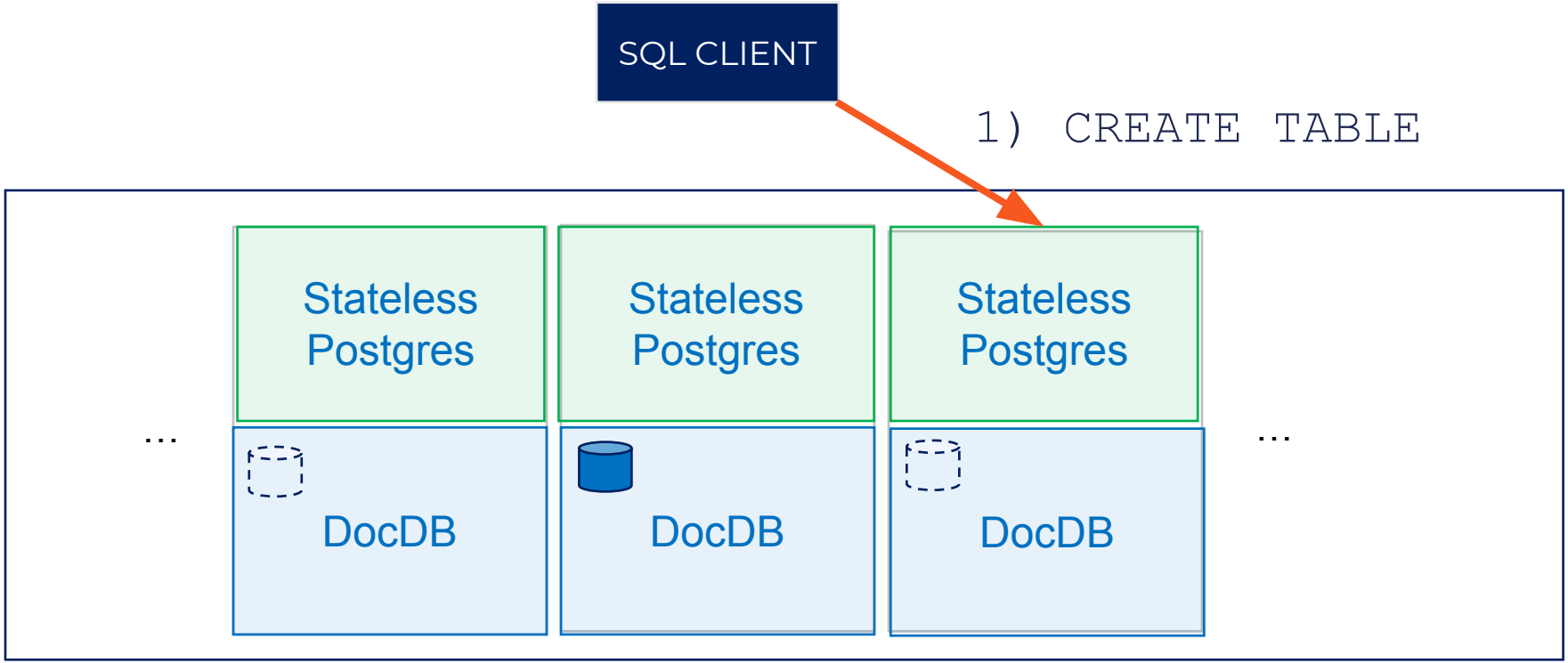
YSQL Tables

- **Tables**
  - Each table maps to one DocDB table
  - Each DocDB table is sharded into multiple tablets
- **System tables**
  - PostgreSQL system catalog tables map to special DocDB tables
  - All such special DocDB tables use a single tablet
- **(Internal) DocDB tables**
  - Have same key → document format
  - Schema enforcement using the table schema metadata

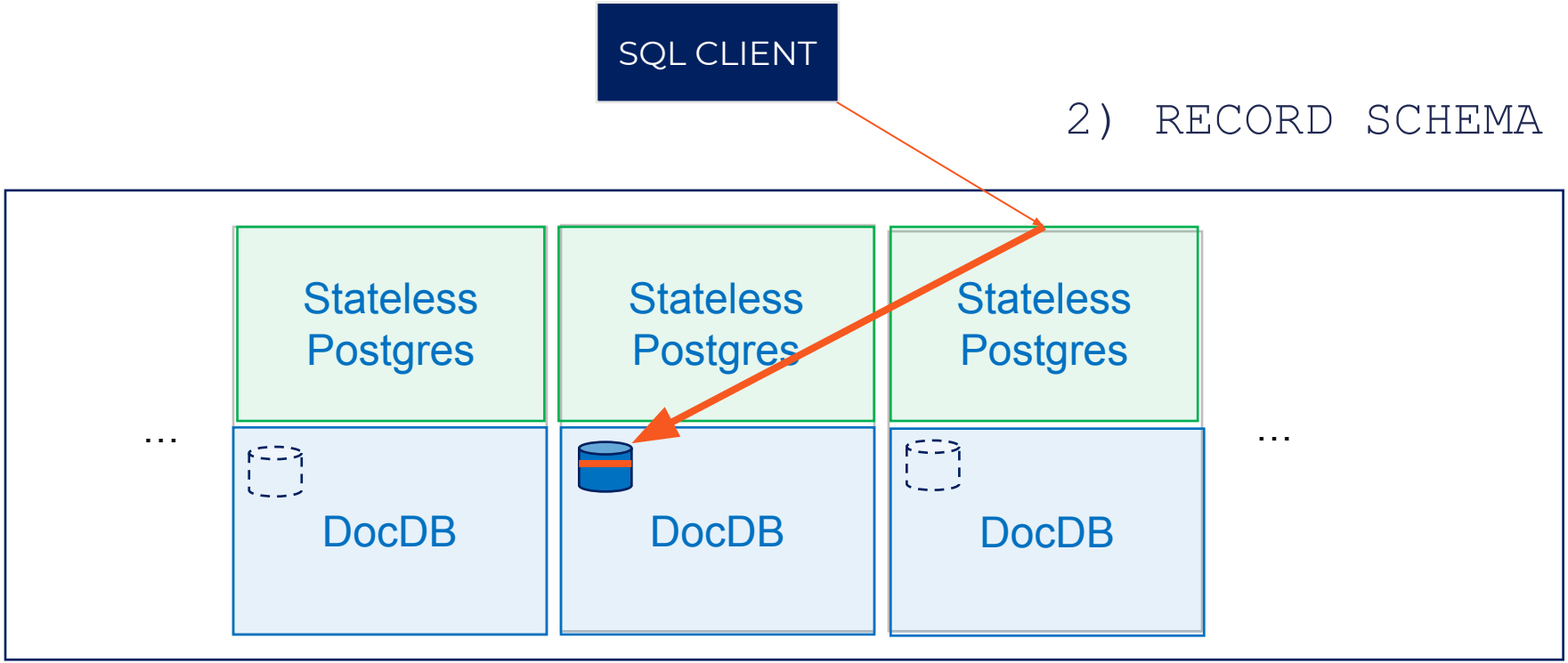
System Catalog Tables are Special Tables



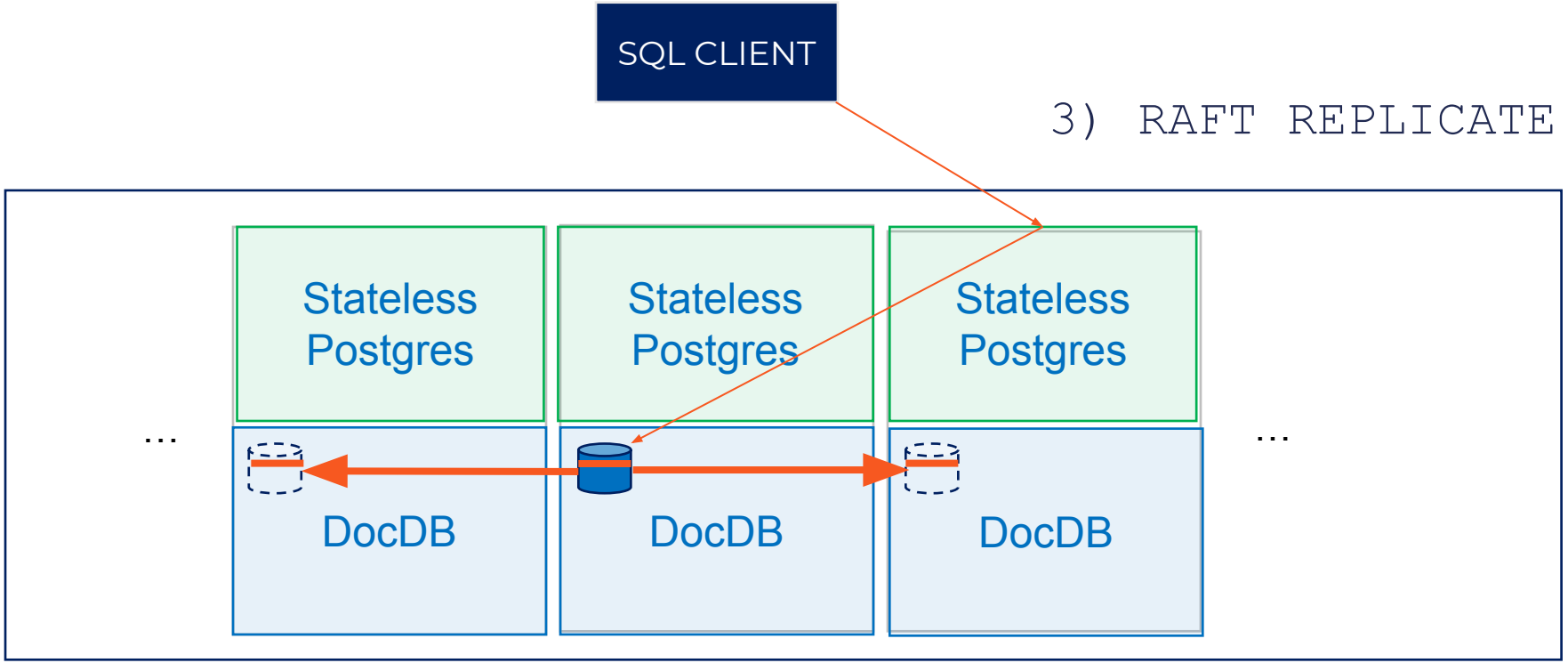
Create a Table



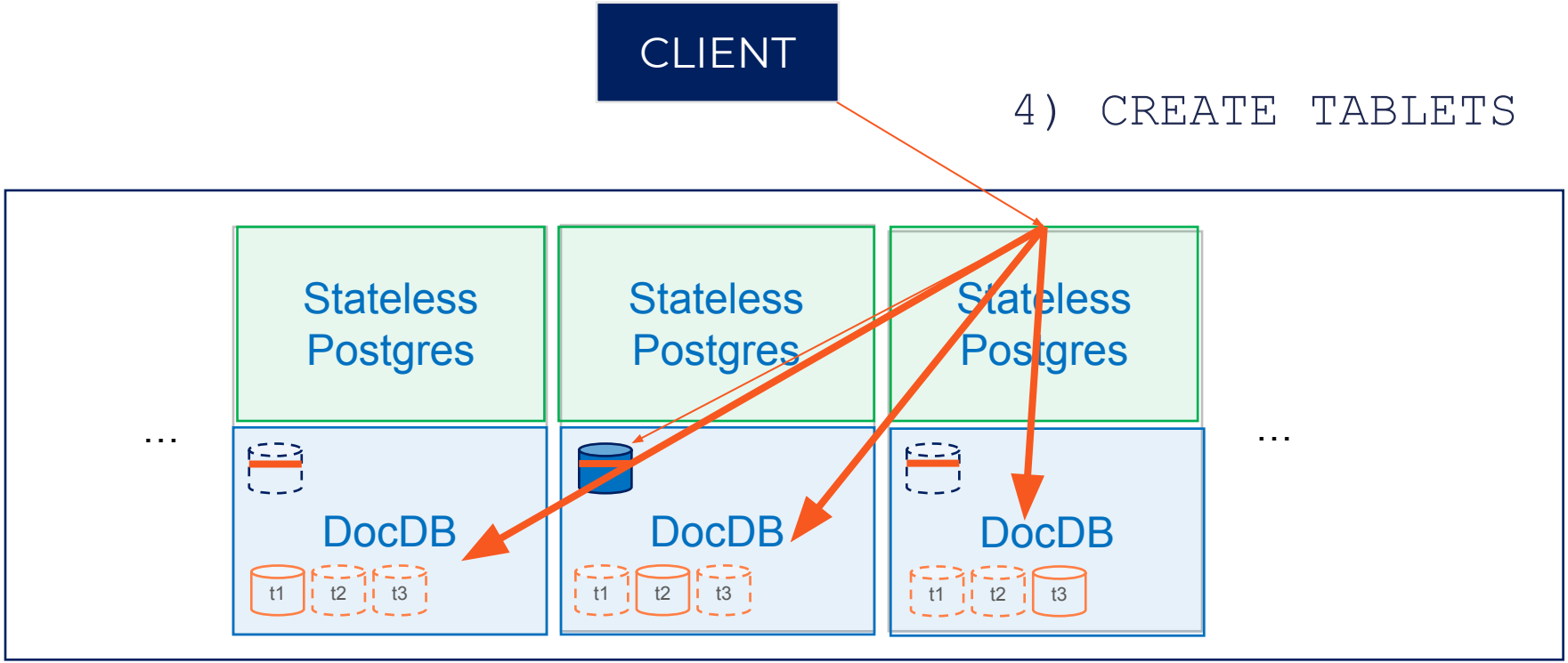
Create a Table



Create a Table



Create a Table



YSQL:
Insert Data

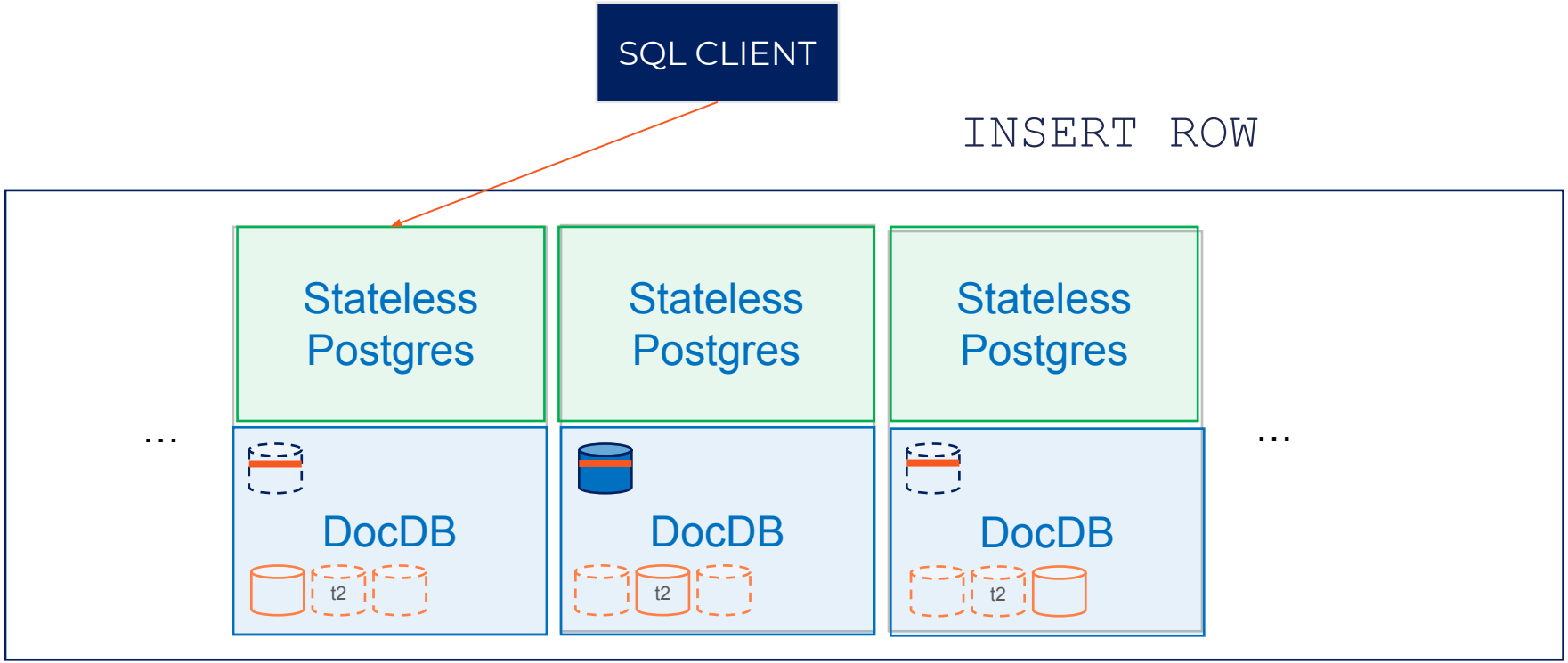
Insert Data into Tables

- **Primary keys**
  - The primary key column(s) map to a single document key
  - Each row maps to one document in DocDB
  - Tables without primary key use an internal ID (logically a row-id)

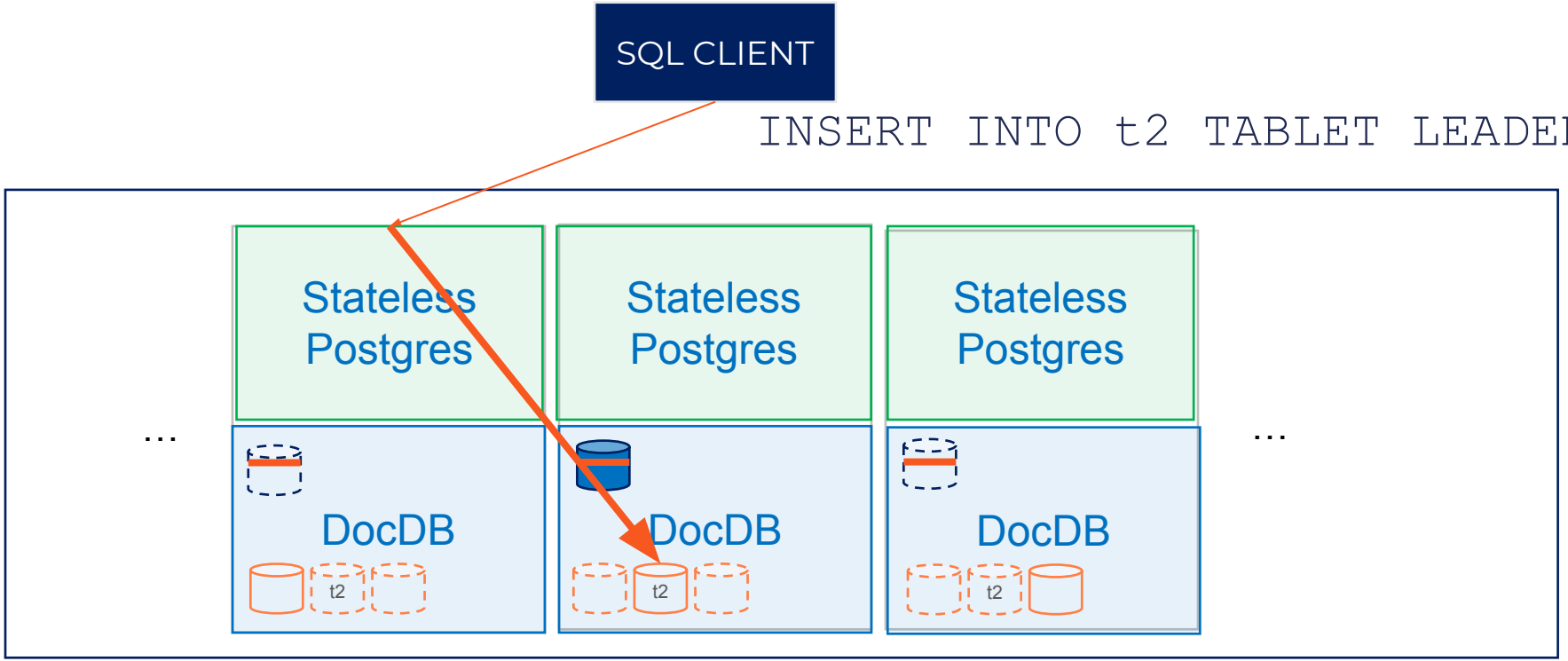
- **Secondary indexes**
  - Each index maps to a separate distributed DocDB table
  - DML implemented using **DocDB distributed transactions**
  - E.g: insert into table with one index will perform the following:

```
BEGIN DOCDB DISTRIBUTED TRANSACTION
    insert into index values (…)
    insert into table values (…)
COMMIT
```

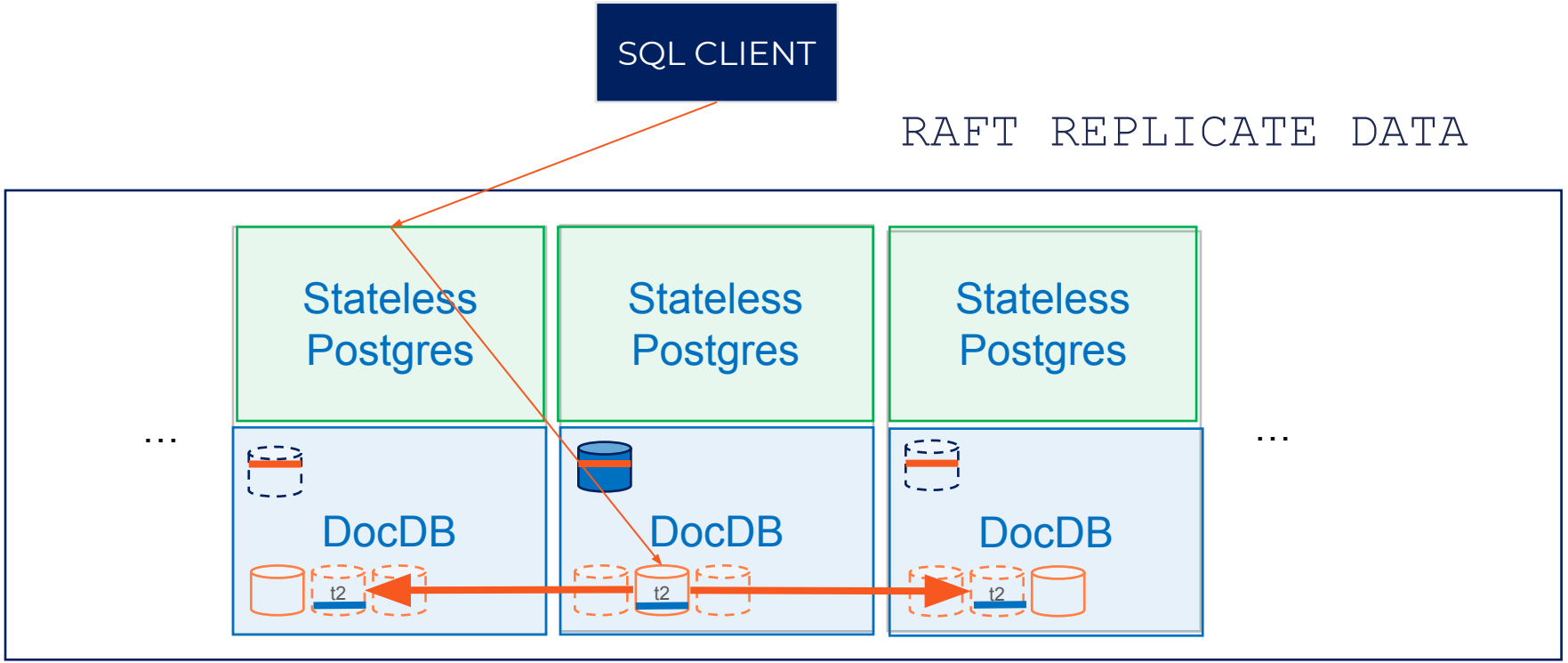
Insert Data



Insert Data



Insert Data



Conclusion

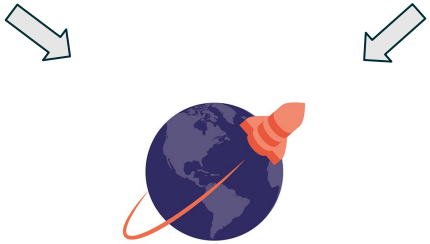
Most Advanced Open Source Distributed SQL

Query Layer

World's Most Advanced
Open Source SQL Engine

Storage Layer

World's Most Advanced
Distributed OLTP Architecture



yugabyteDB

Read more at blog.yugabyte.com

Storage Layer

blog.yugabyte.com/distributed-postgresql-on-a-google-spanner-architecture-storage-layer

Query Layer

blog.yugabyte.com/distributed-postgresql-on-a-google-spanner-architecture-query-layer

Questions?

Download
download.yugabyte.com

Join Slack Discussions
yugabyte.com/slack

Star on GitHub
github.com/YugaByte/yugabyte-db

Thanks!

Summary