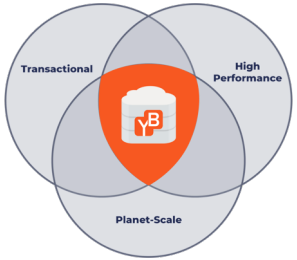


YugaByte DB

Distributed PostgreSQL on Google Spanner Architecture

Karthik Ranganathan
Mihnea Iancu
Mar 21, 2019

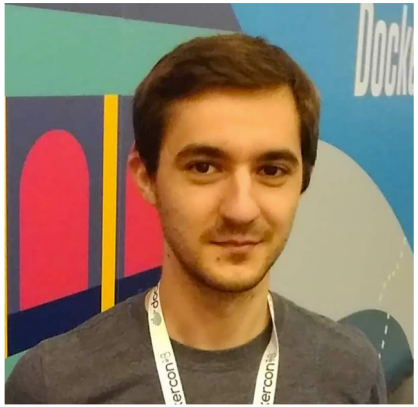
YugaByte

Introduction

Karthik Ranganathan

Co-Founder & CTO, YugaByte
Nutanix Facebook Microsoft
IIT-Madras, University of Texas-Austin

@karthikr



Mihnea Iancu

Software Engineer, YugaByte
Spark SQL YSQL
PhD, Jacobs University Bremen

@mihnea_iancu

YugaByte DB

Distributed SQL DB

PostgreSQL compatible, Elasticity, Fault-Tolerance

Massive Scale

Millions of IOPS, TBs per Node

High Performance

Low Latency Queries

Cloud Native

Multi-Cloud and Kubernetes Ready

YugaByte DB
Built For Microservices

Workload Patterns in Microservices

Internet-Scale OLTP

Optimize for scale, performance

High throughput, low latency

70% of microservice access pattern

Audit trail, stock market data, shopping cart and checkout, messaging, user history, etc.

Cloud-Scale SQL

Scale-out RDBMS

Needs query flexibility

Needs referential integrity and joins

Smaller by volume but critical

CRM and ERP applications, supply chain management, billing services, reporting applications

Distributed SQL

Workload Patterns Fall in a Range

Design Follows a Layered Approach

Query Layer Supports Distributed Postgres

YCQL
SQL-Based Flexible Schema API

YSQL
Globally Distributed Postgres API

DISTRIBUTED, DOCUMENT STORE

Transactional, High Performance, Globally Distributed

RUN ON ANY HARDWARE/IAAS

Core Features of DocDB

YCQL
SQL-Based Flexible Schema API

YSQL
Globally Distributed Postgres API

Self-Healing, Fault-Tolerant

High Perf, Low Latency

ACID Transactions

Auto Sharding & Rebalancing

Global Data Distribution

RUN ON ANY HARDWARE/IAAS

Runs on Bare-metal, VMs, Docker/Kubernetes

YCQL
SQL-Based Flexible Schema API

YSQL
Globally Distributed Postgres API

Self-Healing, Fault-Tolerant

High Throughput, Low Latency

ACID Transactions

Auto Sharding & Rabalancing

Global Data Distribution

DocDB

A Google Spanner-like Distributed, Document Store

Design Goals

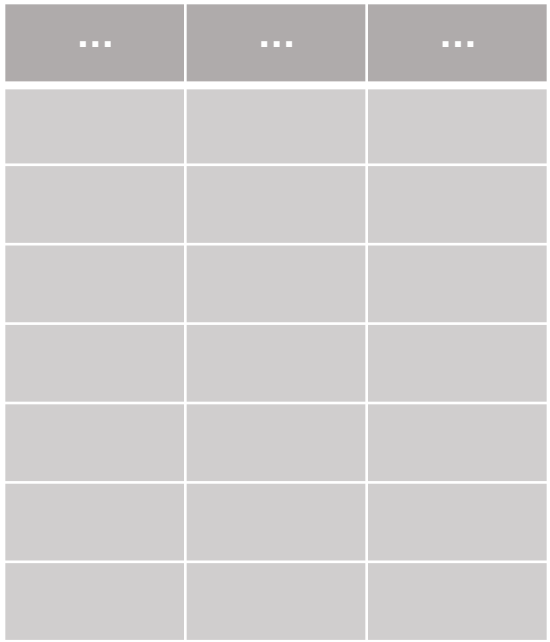
- **CAP Theorem**
  - Consistent
  - Partition Tolerant
  - HA on failures
    (new leader elected in seconds)

- **Transaction Support**
  - Single-row linearizable txns
  - Multi-row txns
    - Serializable
    - Snapshot

- **High Performance**
  - All layers in C++ to ensure high perf
  - Run on large memory machines
  - Optimized for SSDs

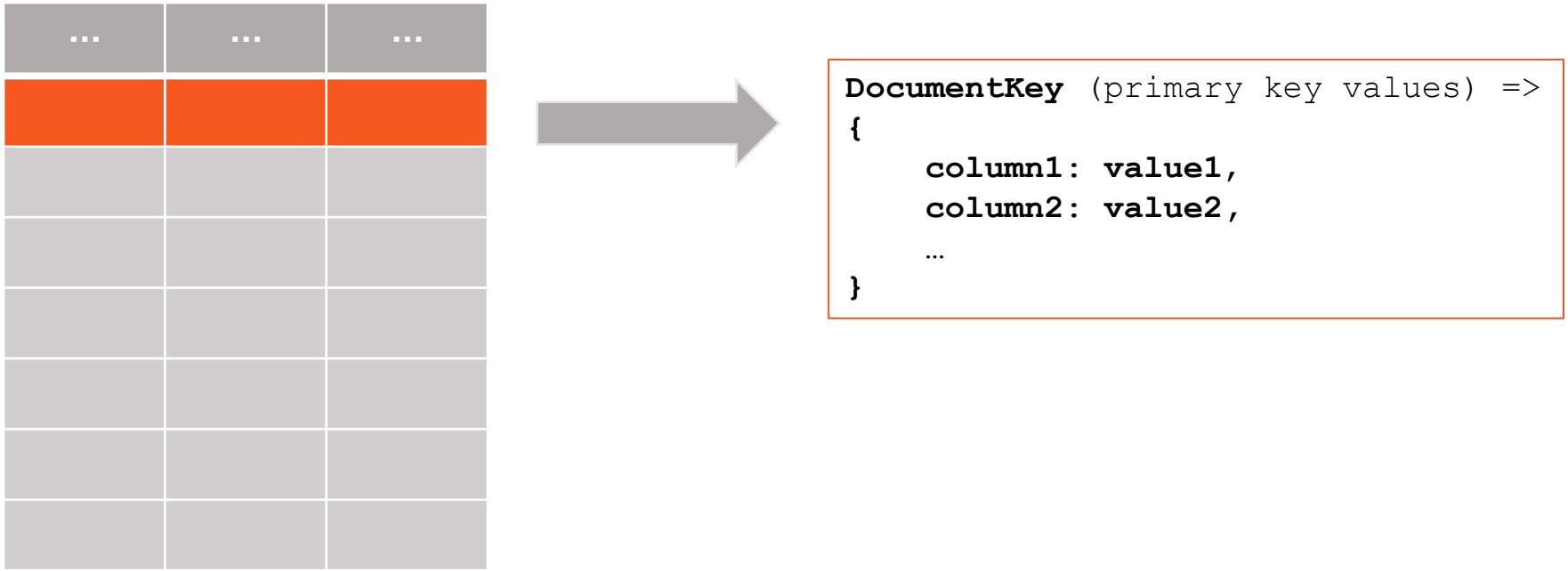
- **Run anywhere**
  - No external dependencies
  - No need for Atomic Clocks
  - Bare metal, VM and Kubernetes

How Does DocDB Work?



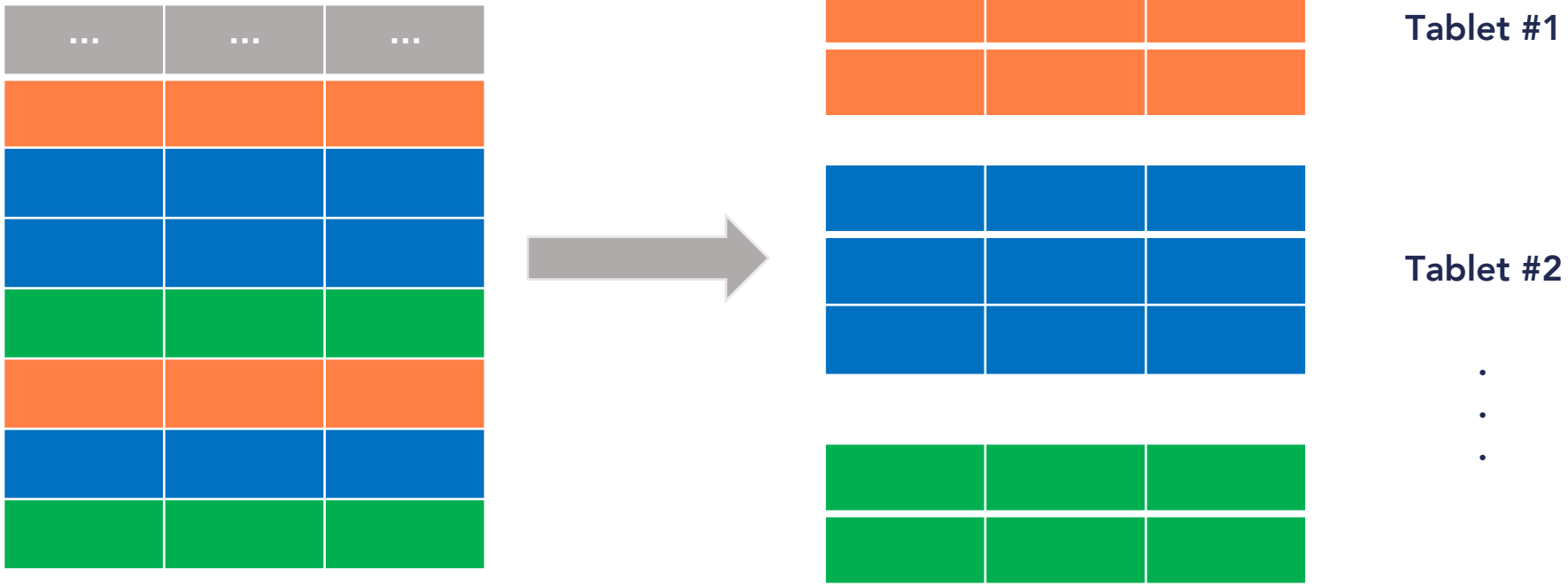
Let's start with this logical view of a table

Each Row is a Document



A row maps to a document, each column to an attribute

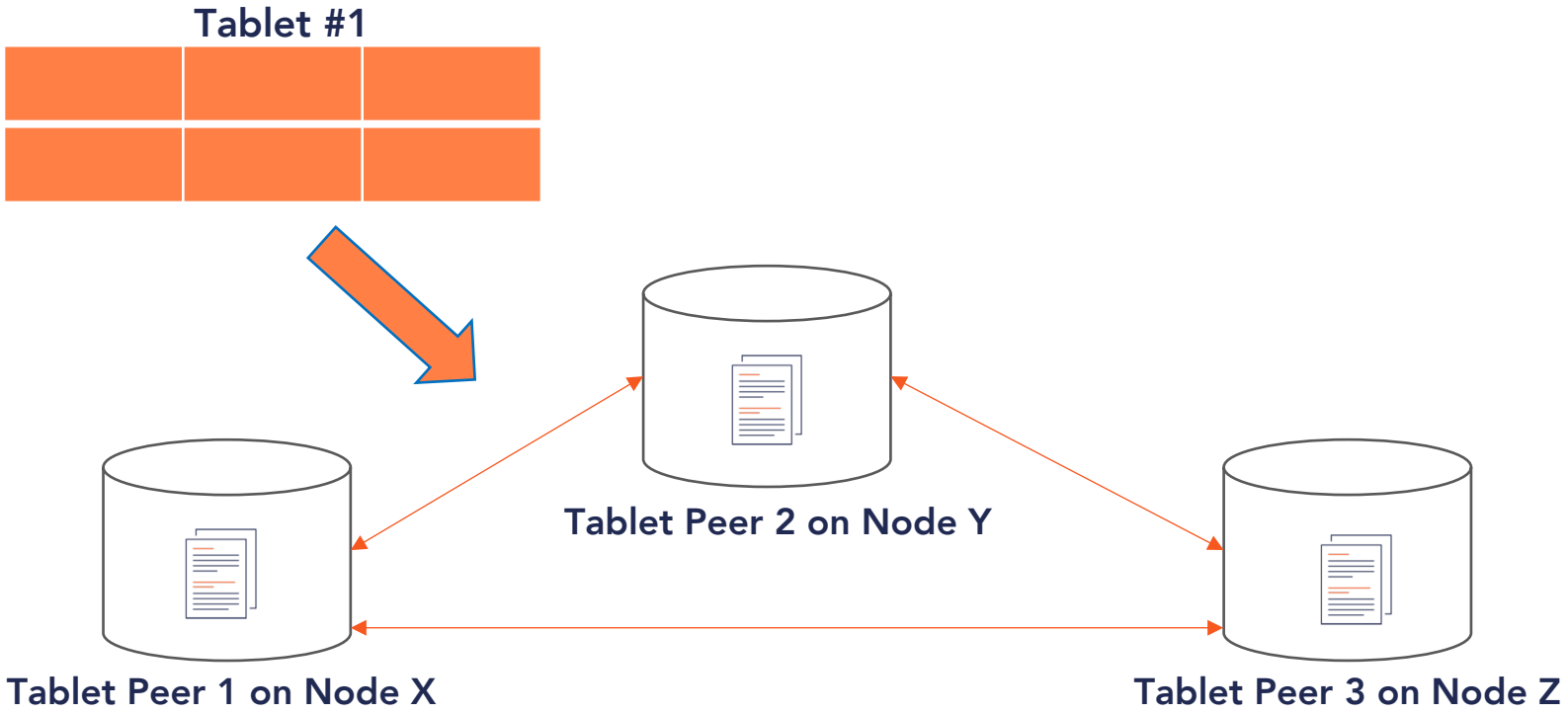
Tables are Sharded into Tablets



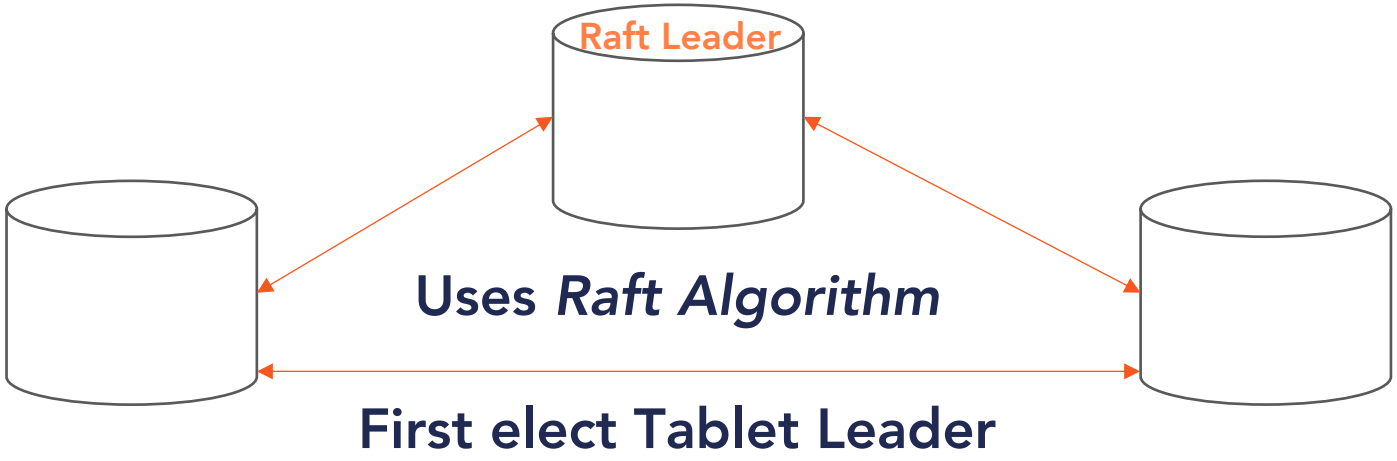
Now partition the table using some strategy

Each partition is a tablet. A row belongs to exactly one tablet.

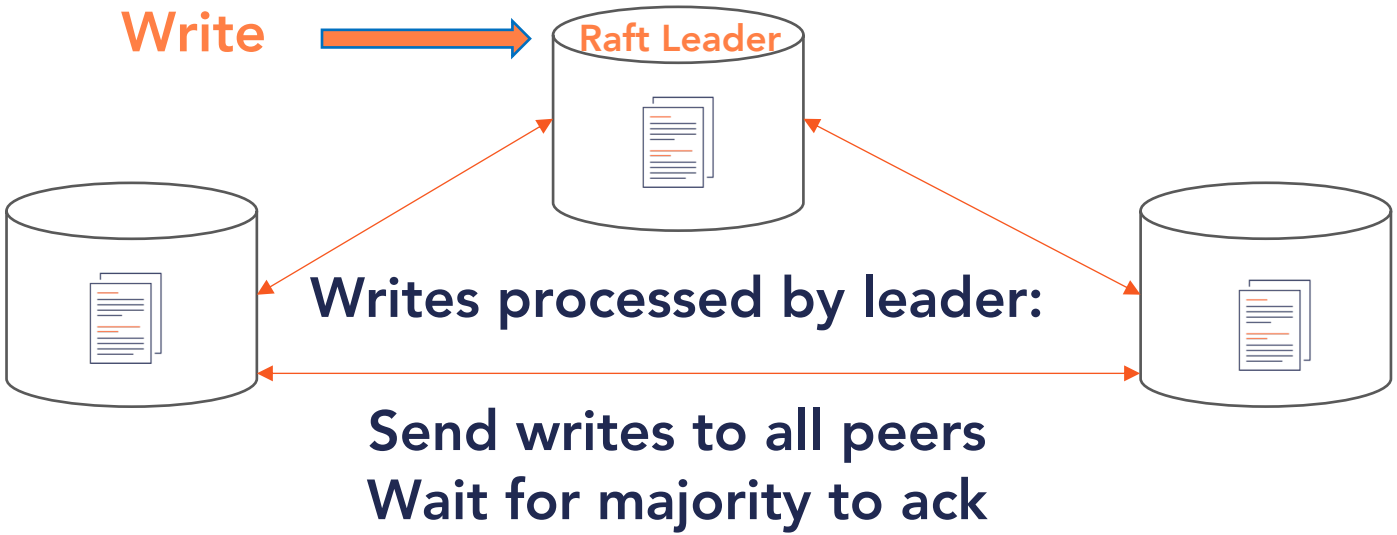
Tablets are Replicated across Nodes



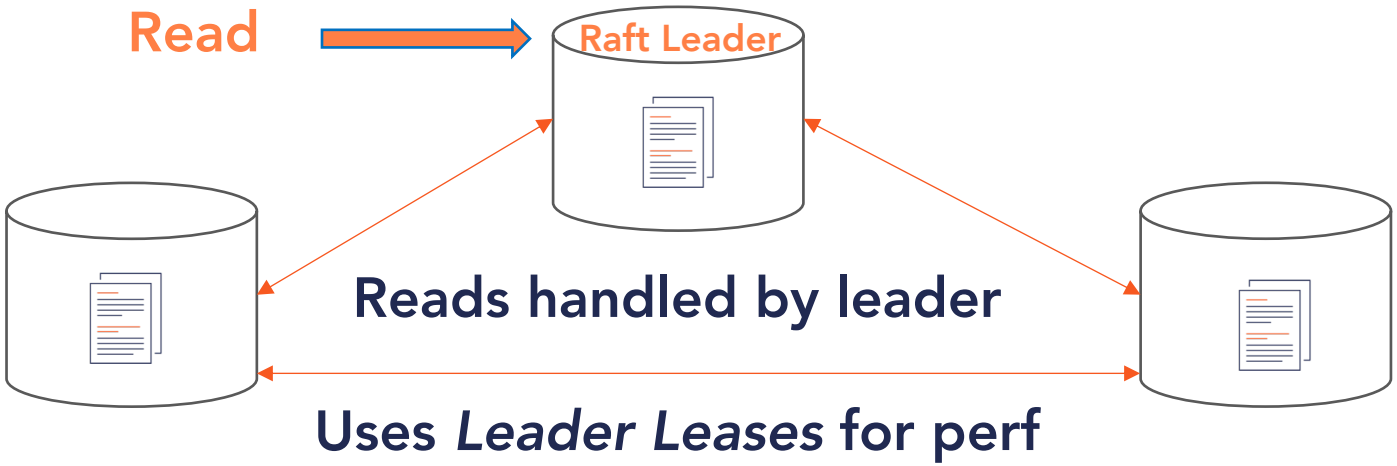
How Replication works



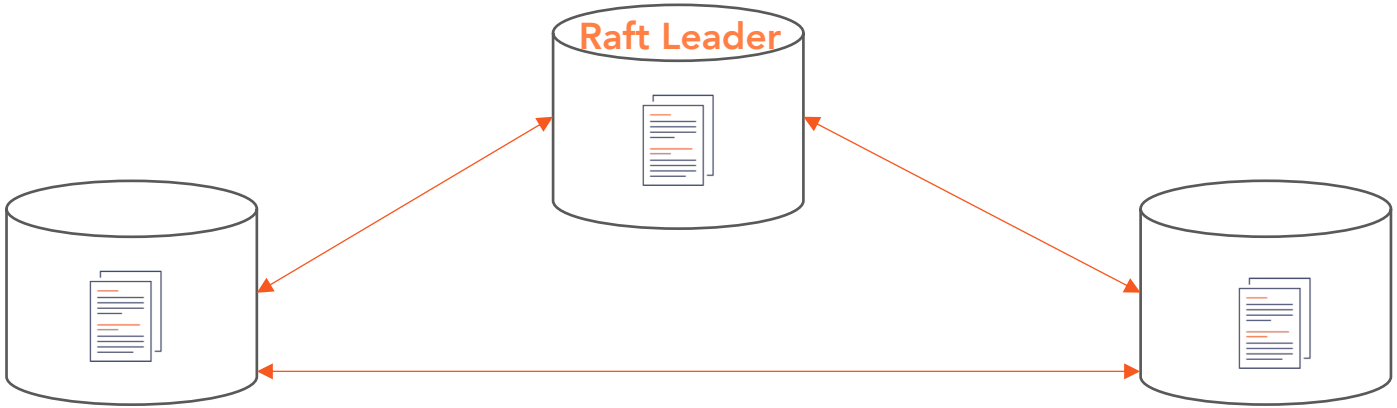
How Replication works



How Replication works



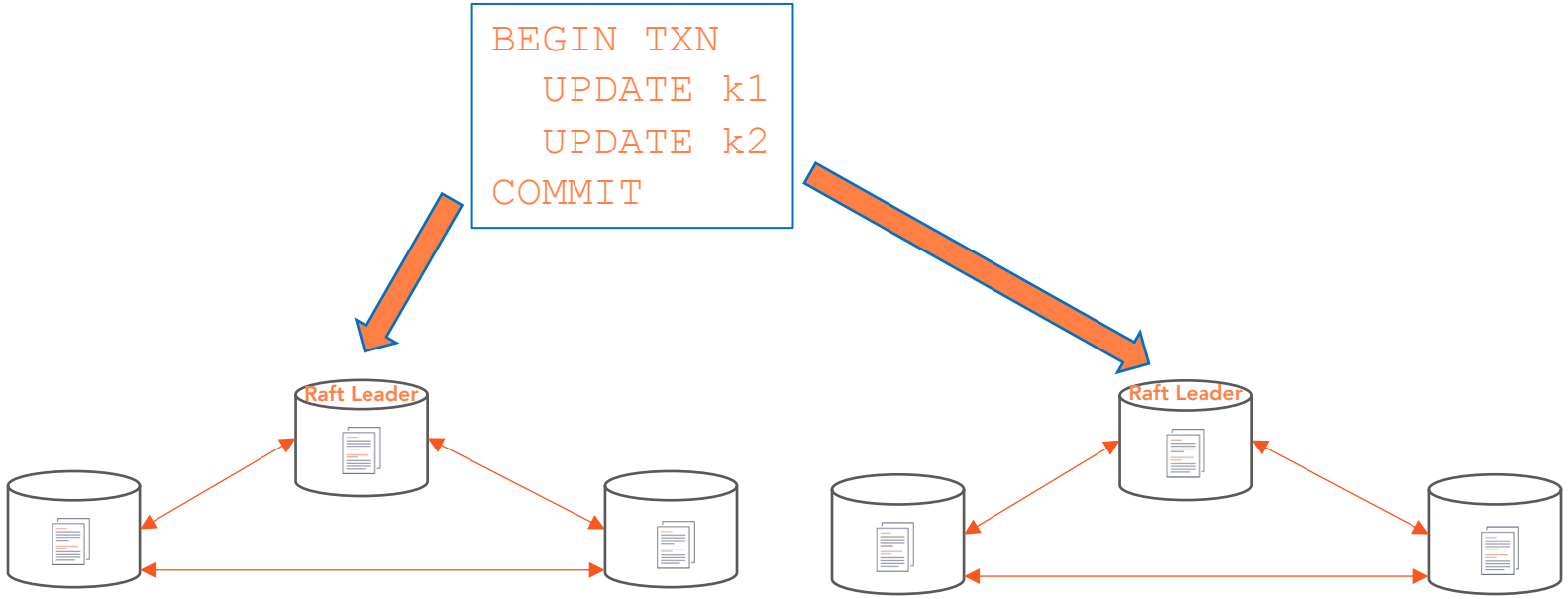
Single-Key Linearizability



This system is now **linearizable**, **HA**, **fault tolerant** with **high-performance**

But no distributed transactions yet!

What do Distributed Transactions need?



Updates should get written at the same physical time

But how will nodes agree on time?

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 or HLC

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

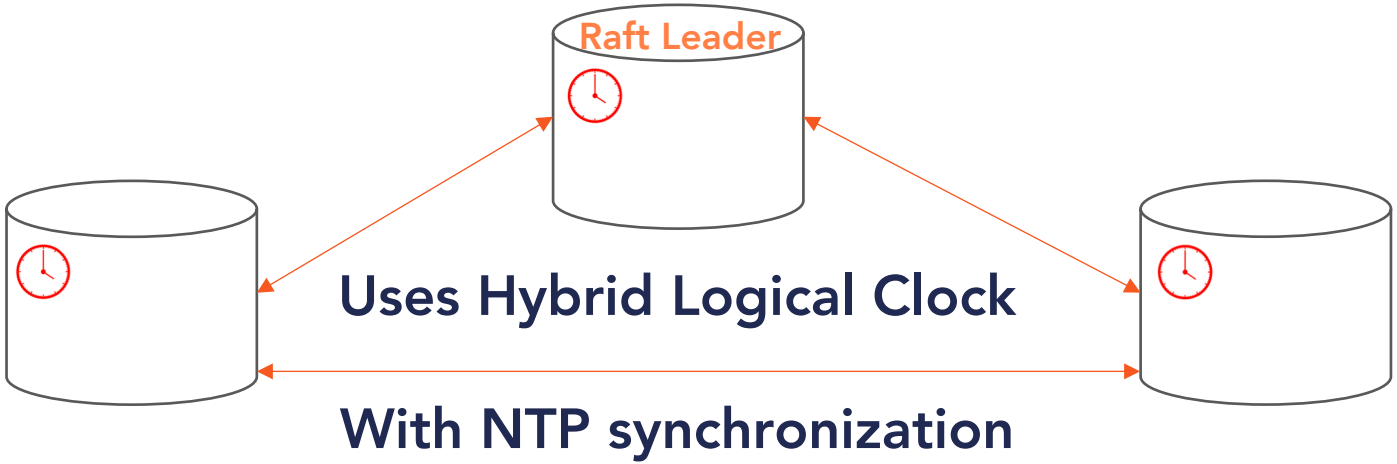
(physical component, logical component)

synchronized using NTP

a monotonic counter

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

No Need For Atomic Clocks



Read more at blog.yugabyte.com

Storage layer details:

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

YSQL
The PostgreSQL Query Layer

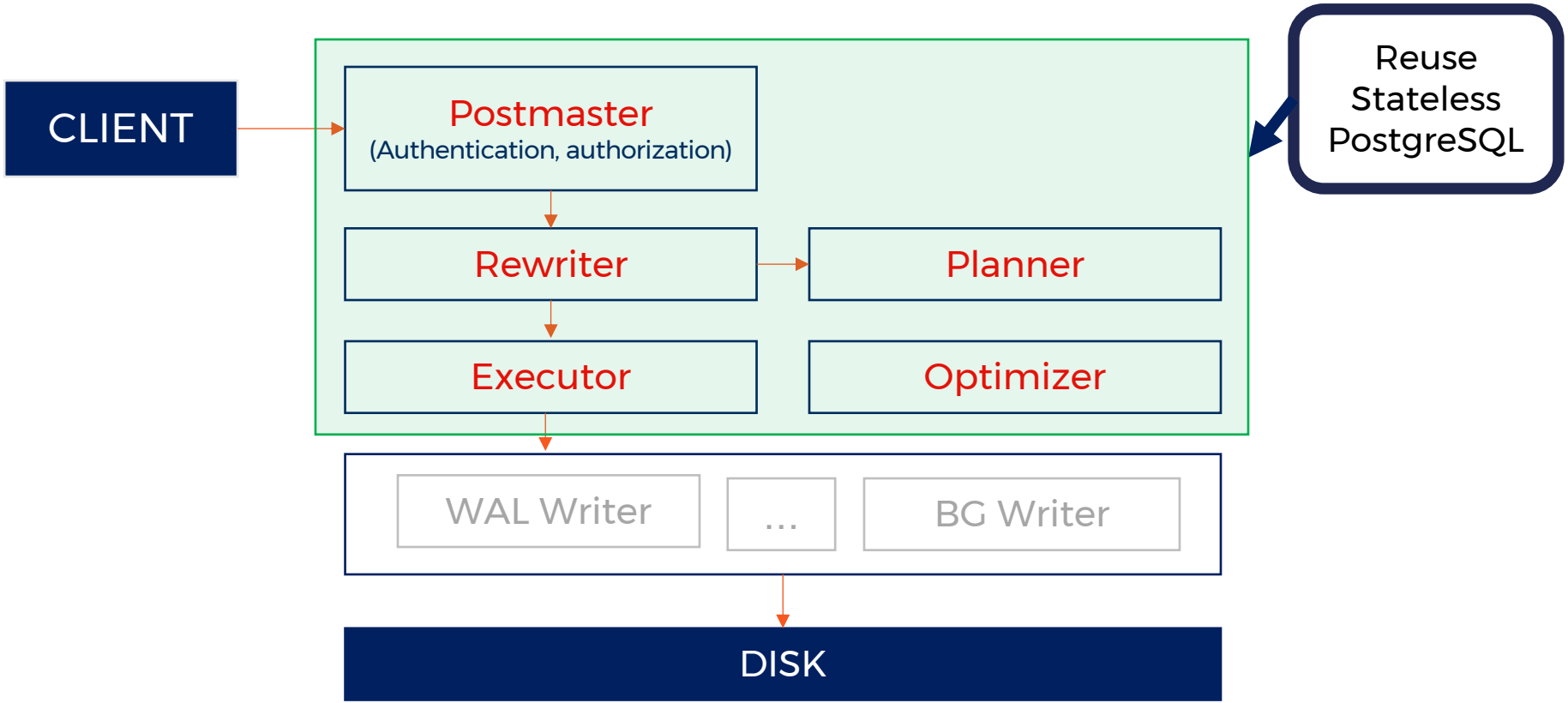
Design Goals

- **PostgreSQL compatible**
  - Re-uses PostgreSQL code base
  - New changes do not break existing PostgreSQL functionality
  - Aim towards building a pluggable distributed storage engine
- **Enable migrating to newer PostgreSQL versions**
  - New features are implemented in a modular fashion
  - Integrate with new PostgreSQL features in an on-going fashion
  - E.g. Moved from PostgreSQL 10.4 → 11.2 in a few weeks!
- **Cloud native design**
  - Designed for running natively in Kubernetes
  - Make drivers cluster aware over time
  - Support multi—zone and geographically replicated deployments

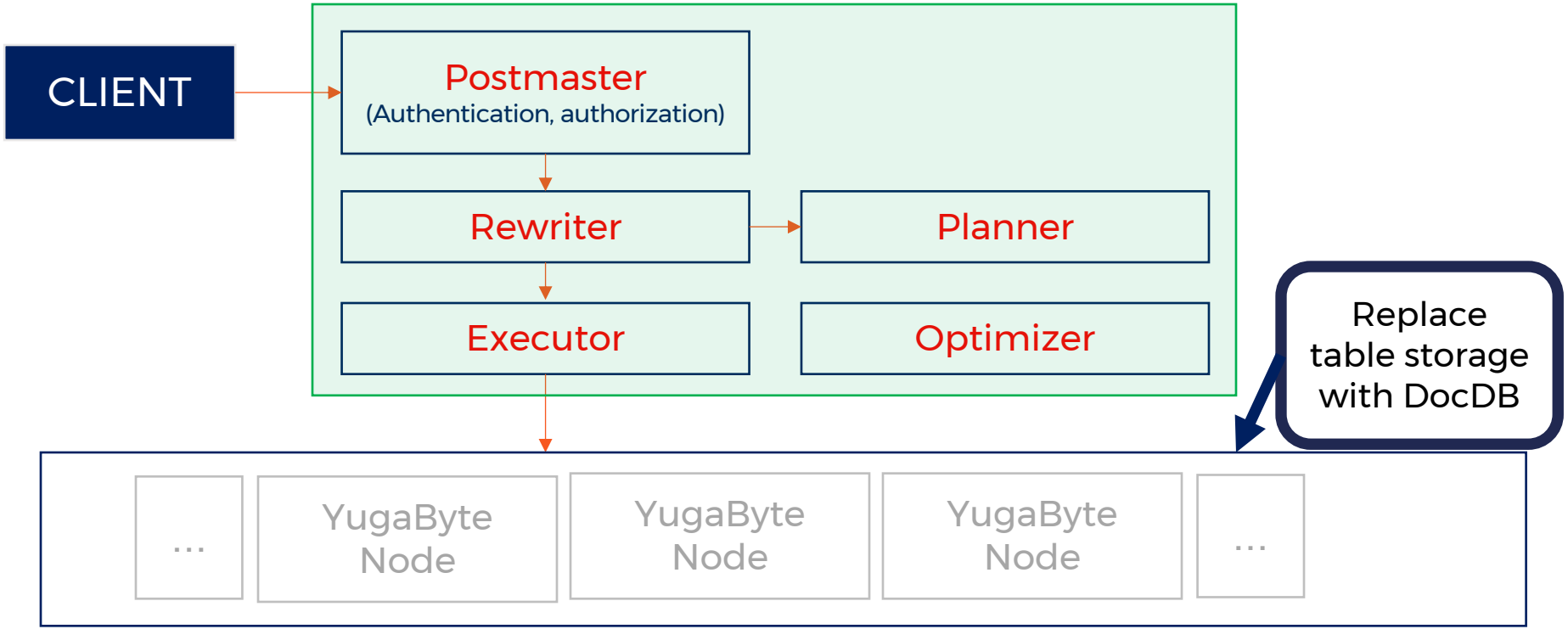
Design Goals - Feature-set Support

- All data types
- Built-in functions and expressions
- Various kinds of joins
- Constraints (primary key, foreign key, unique, not null, check)
- Secondary indexes (incl. multi-column & covering columns)
- Distributed transactions (Serializable and Snapshot Isolation)
- Views
- Stored Procedures
- Triggers

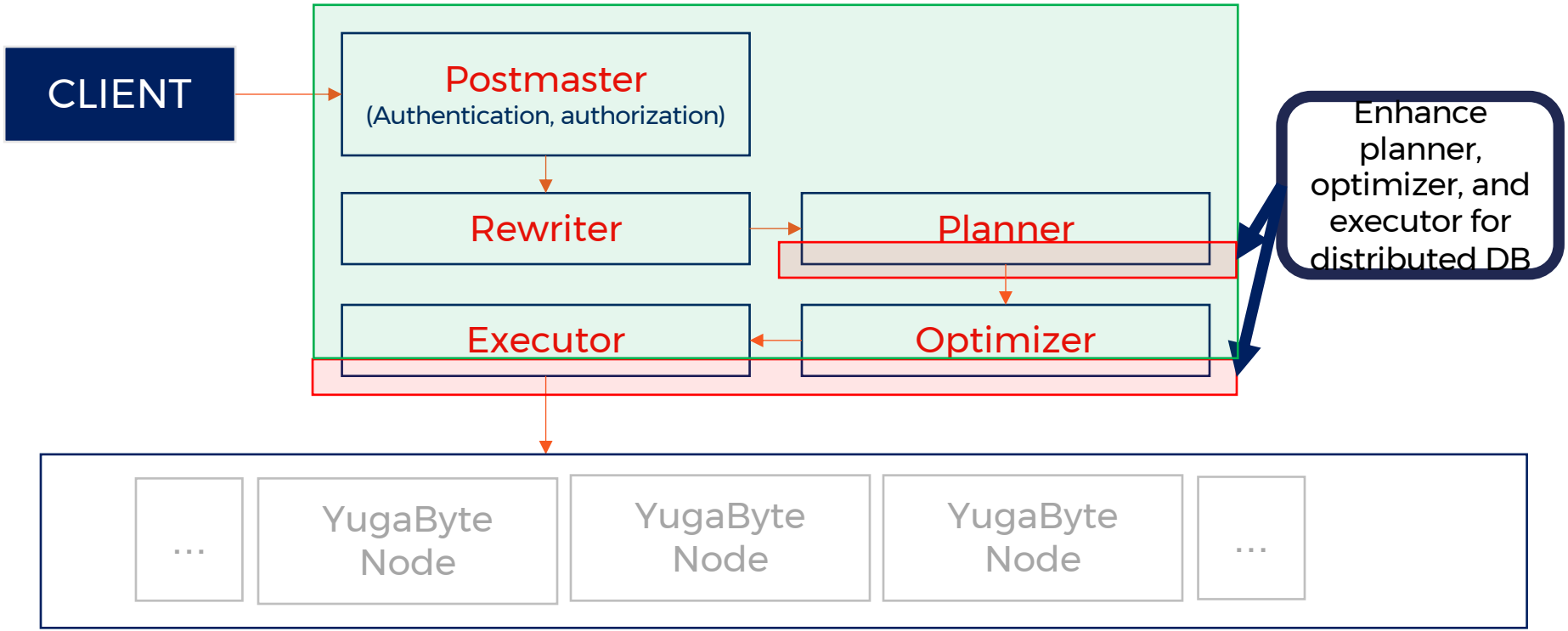
Existing PostgreSQL Architecture



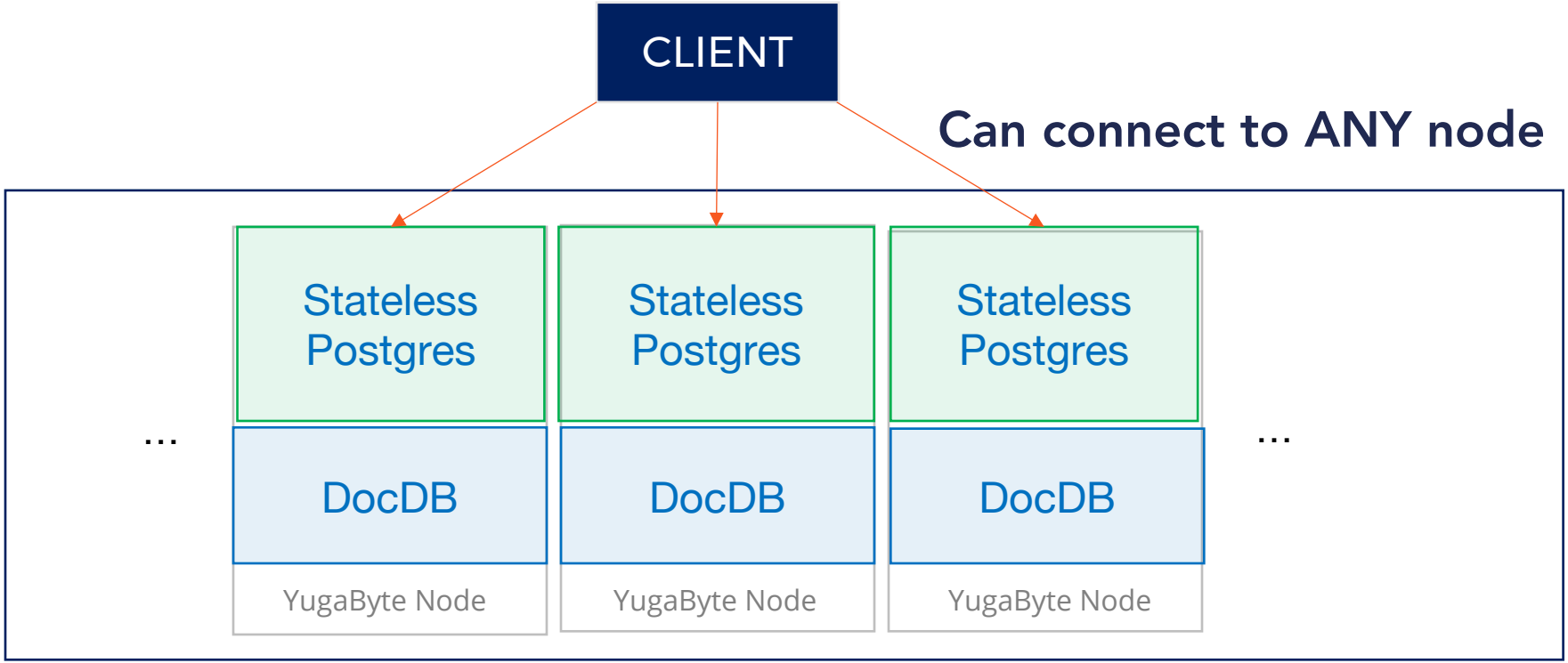
DocDB as Storage Engine



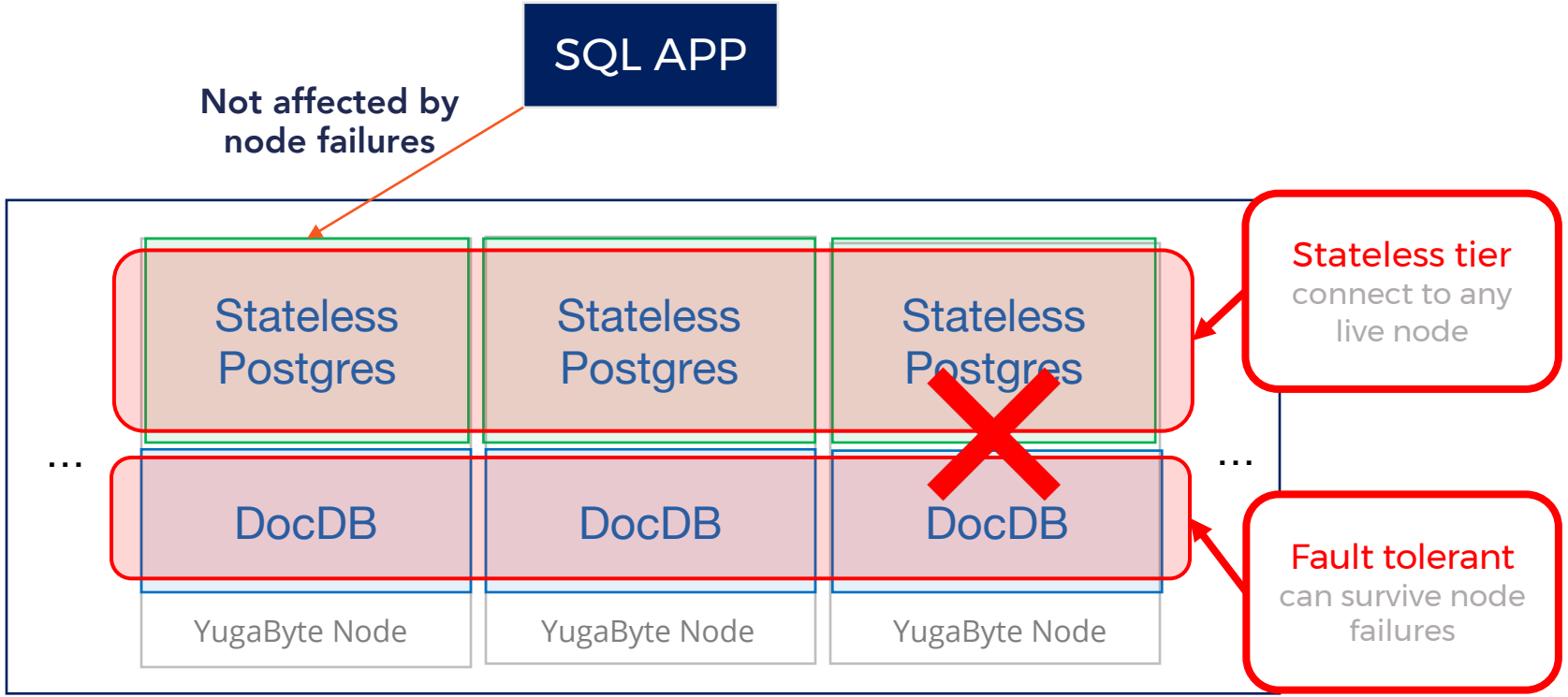
Make PostgreSQL Stateless



All Nodes are Identical



All Nodes are Identical



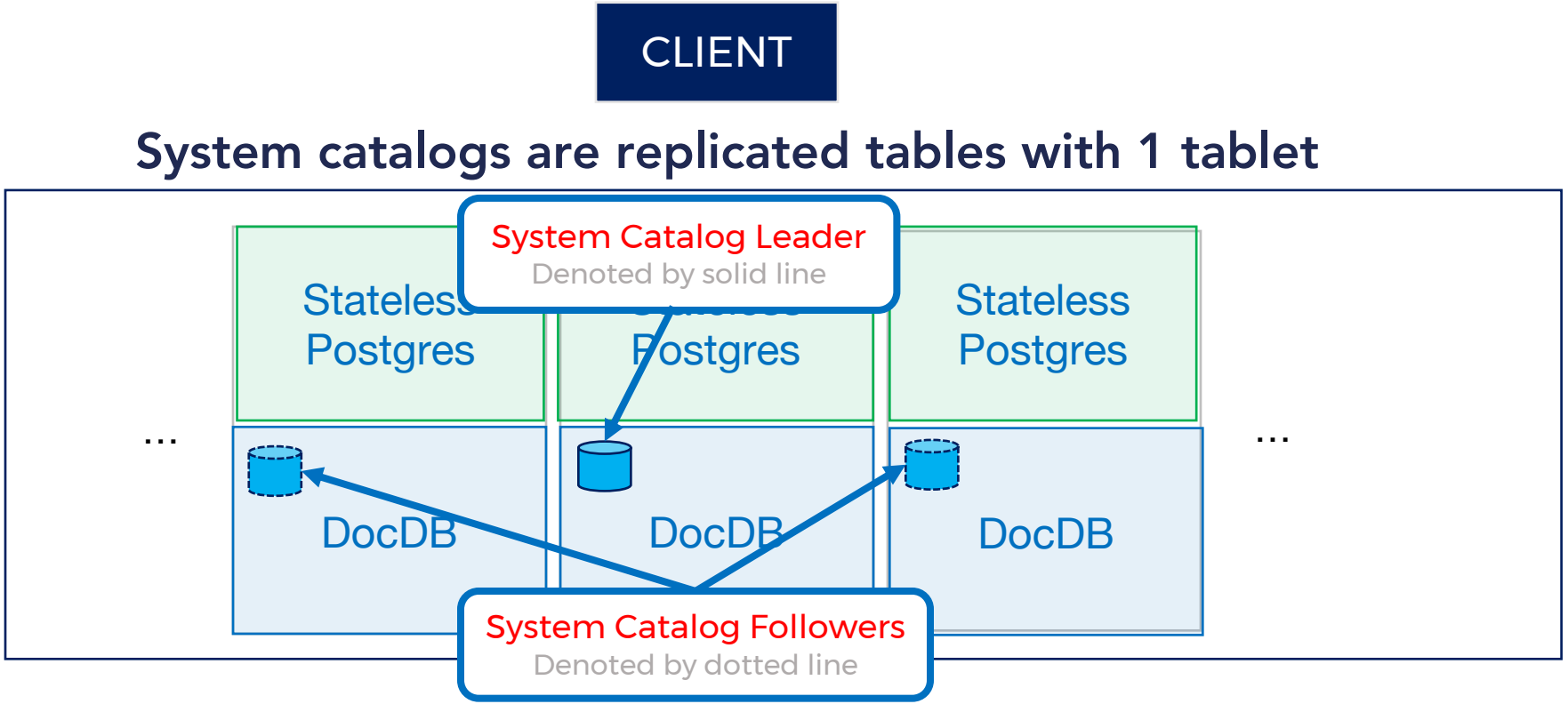
YSQL
Using distributed PostgreSQL

Creating YSQL Tables

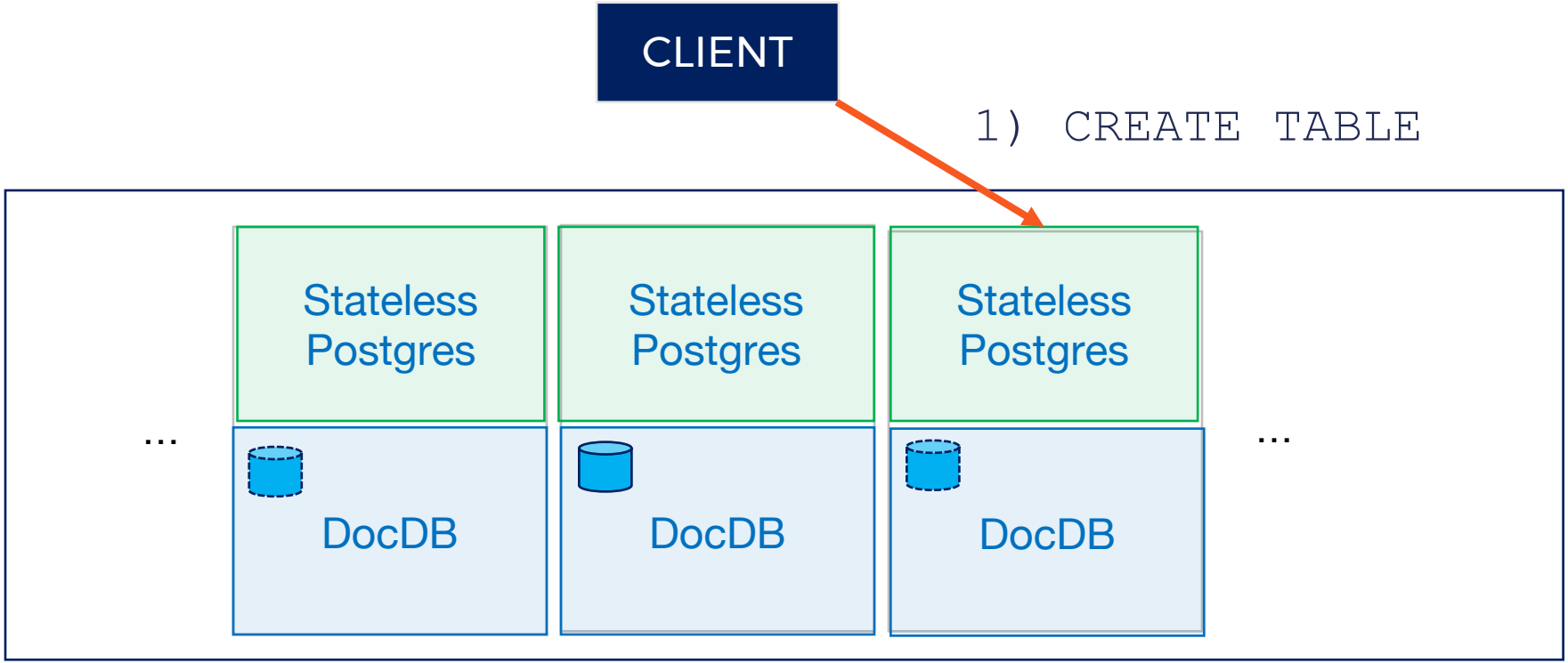
- **YSQL Tables**
  - User tables map to one DocDB table
  - Each index maps to a separate DocDB table
  - PostgreSQL system catalogs map to special DocDB tables
    - Used for schema enforcement
    - Handle views, foreign tables, stored procedures, etc.

- **YSQL Rows**
  - Each row maps to one document in DocDB: `key` → `document`
  - The primary key column(s) map to the document key
  - Tables without primary key use an internal ID (logically a *row-id*)

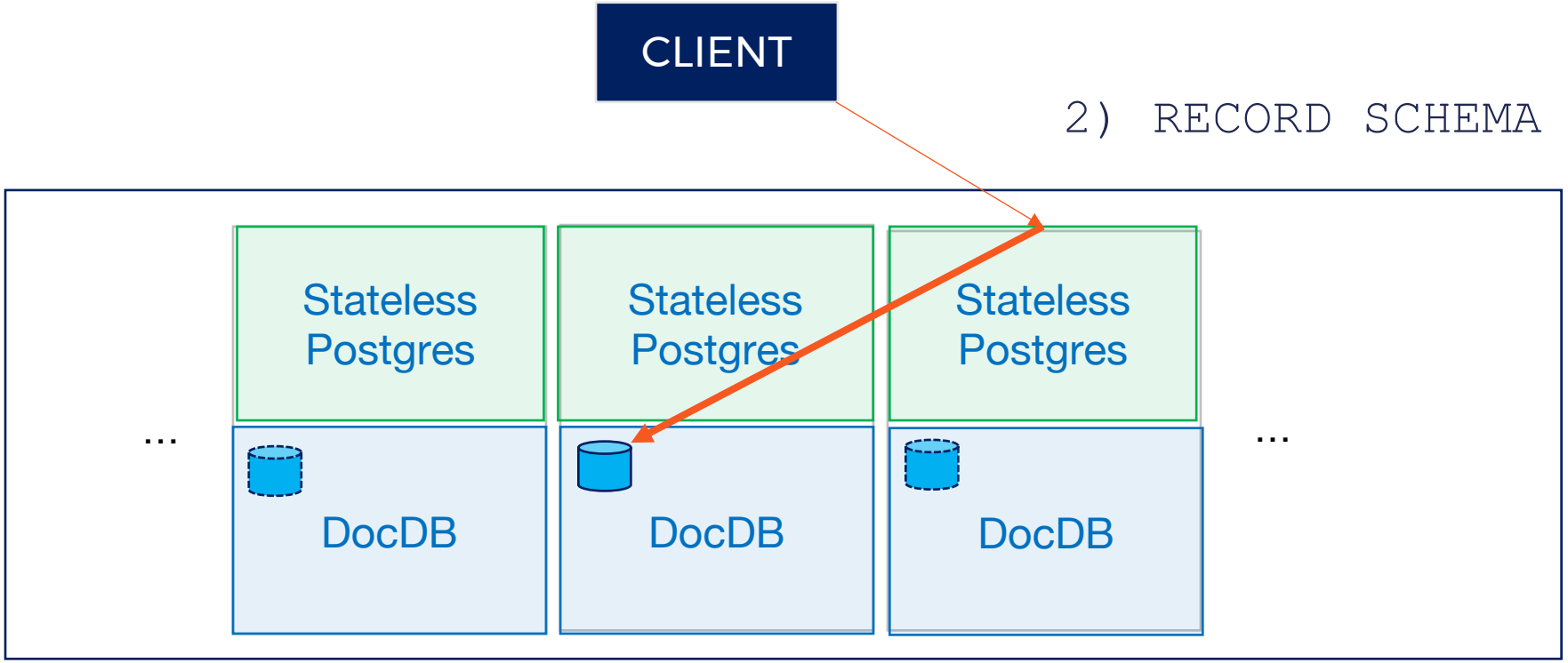
System Catalogs are Special Tables



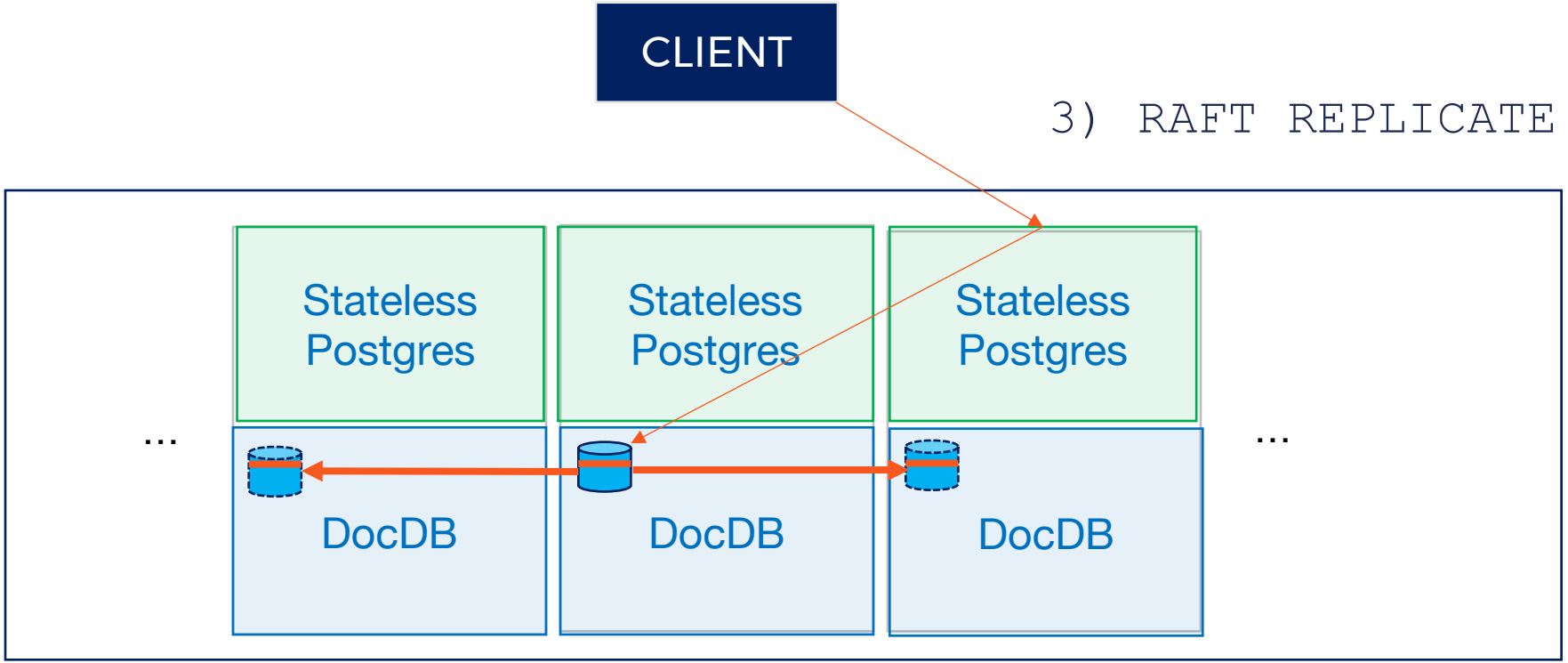
Creating a Table



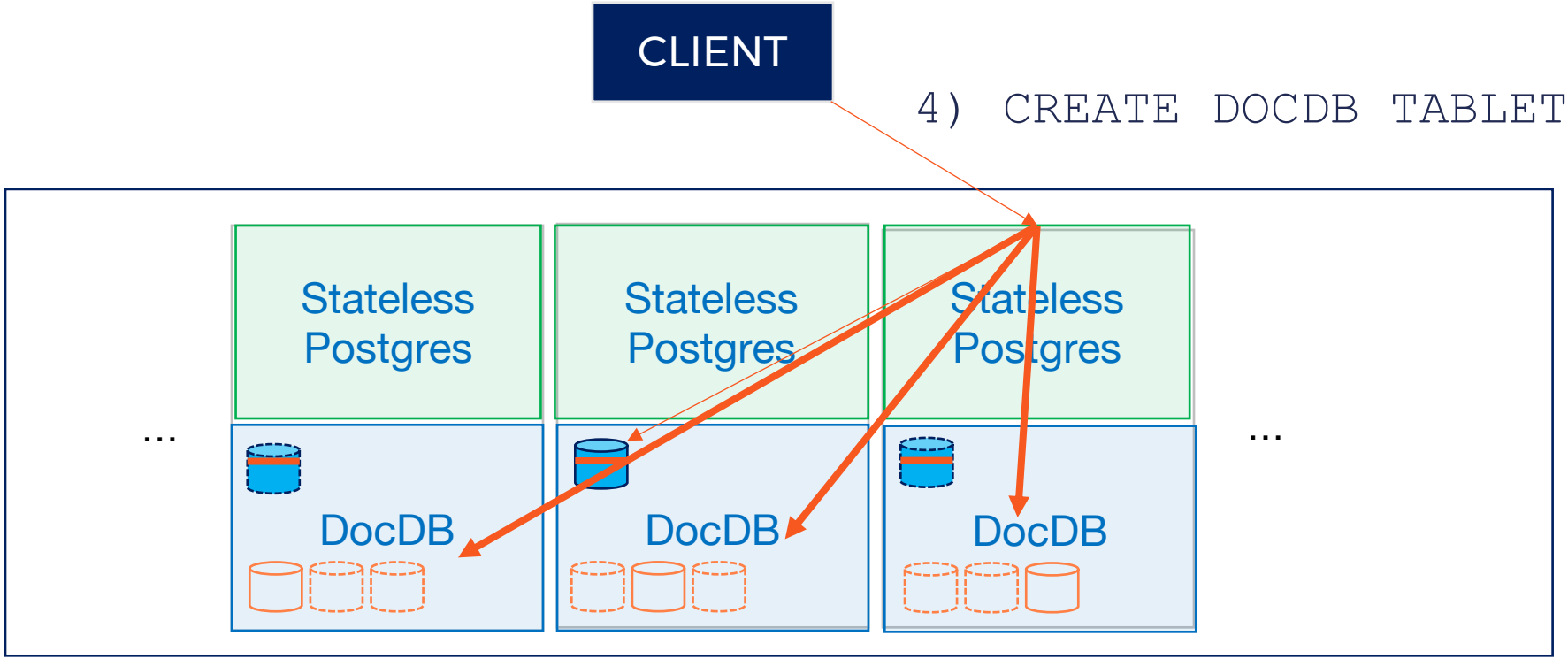
Creating a Table



Creating a Table



Creating a Table



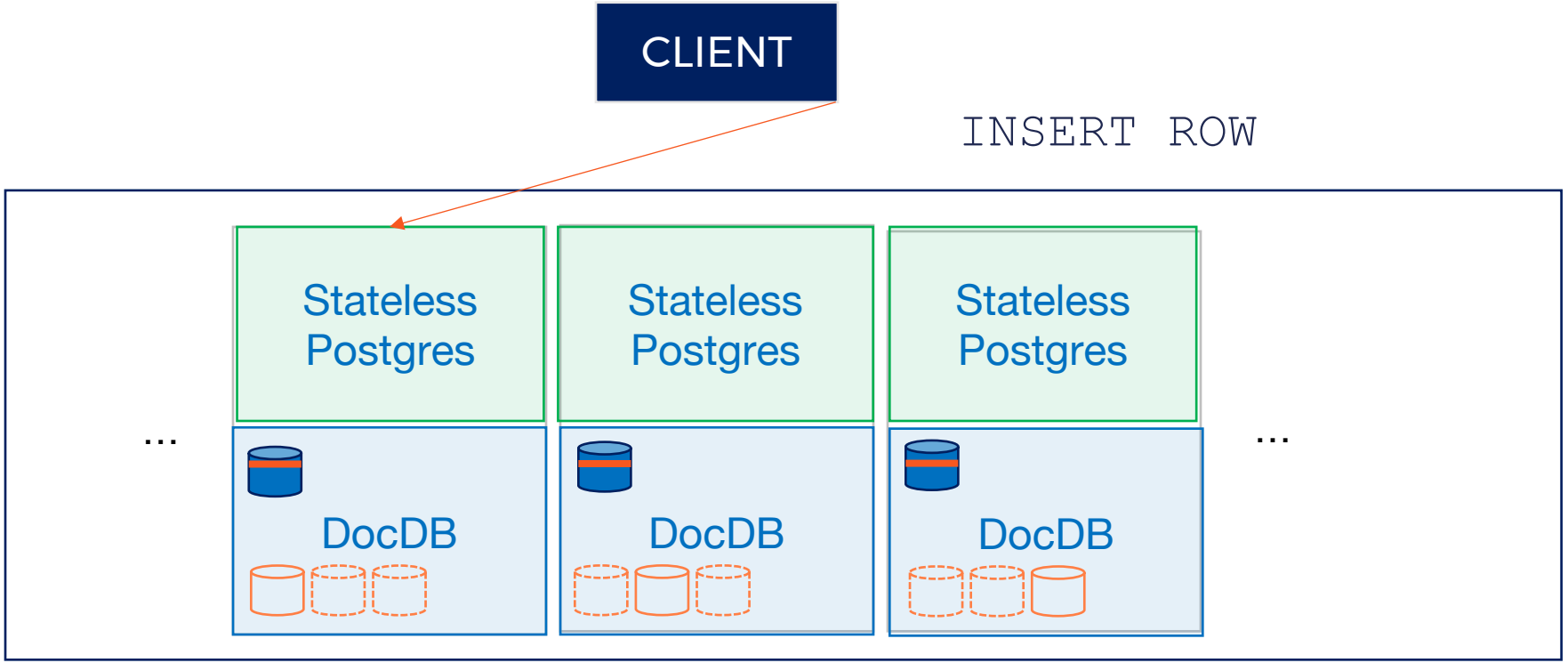
Using YSQL Tables

- **Single-row Operations**
  - Reads and writes handled by DocDB tablet leader
  - YSQL query layer is aware of clustering and partitioning
  - Will route queries to the right node (tablet leader).

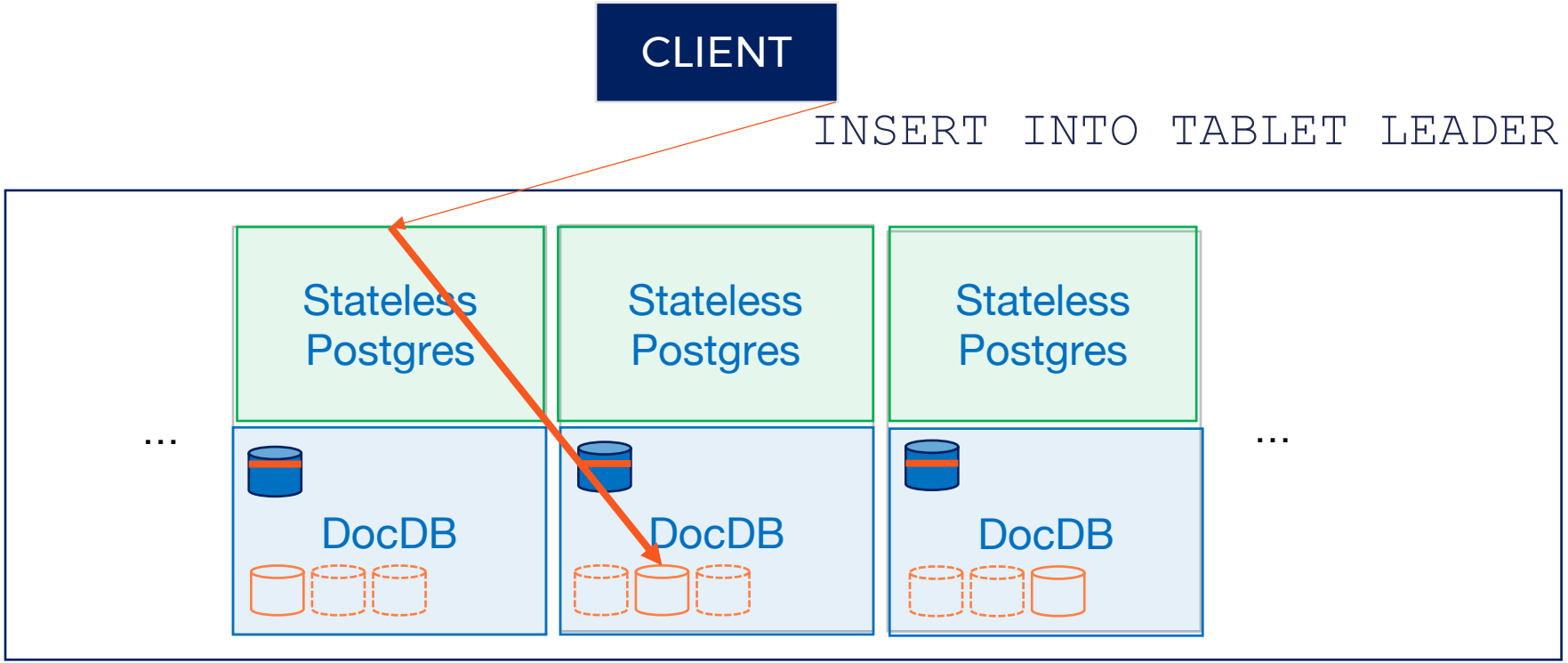
- **Multi-row Operations**
  - Implemented using DocDB distributed transactions
  - E.g. `insert into table` with one `index` will perform the following:

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

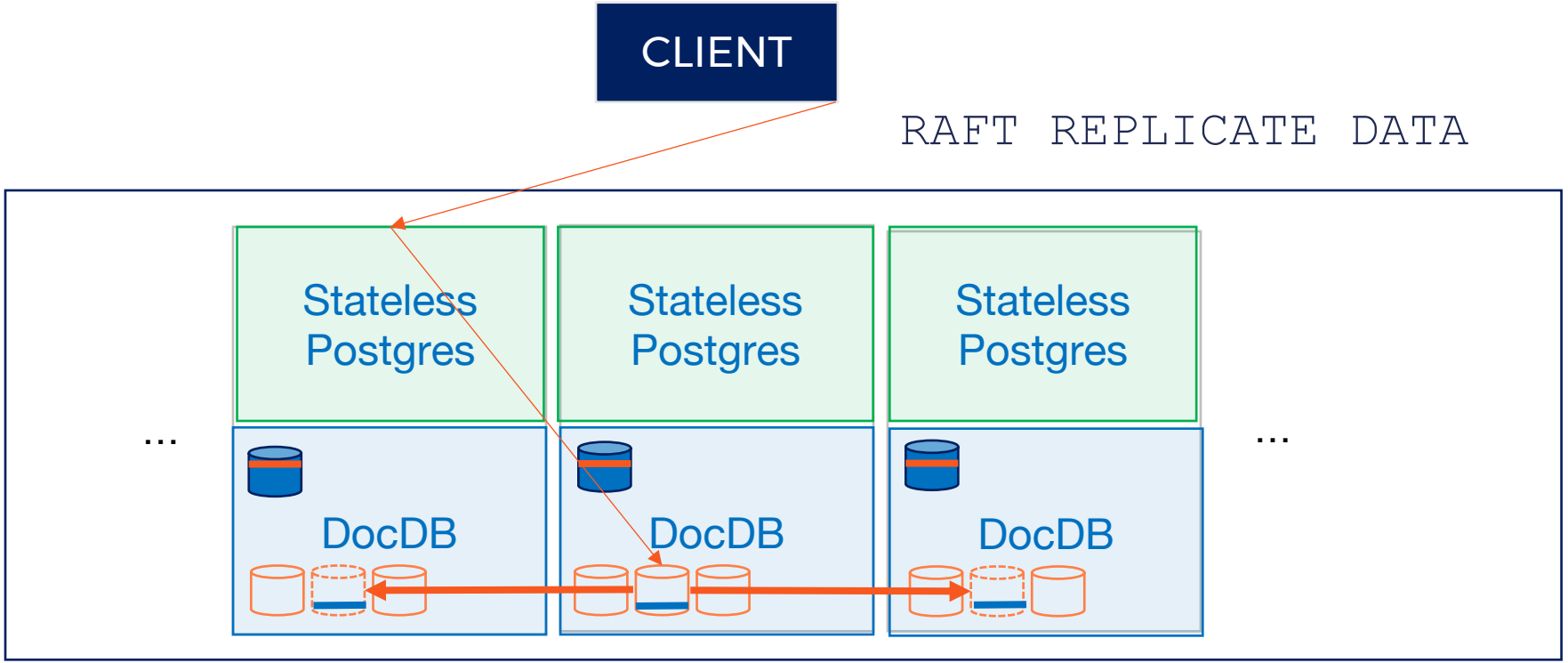
INSERTING DATA



INSERTING DATA



INSERTING DATA



Read more at blog.yugabyte.com

Storage layer details:

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

Query layer details:

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

PostgreSQL Meets Spanner!

- **Leverage PostgreSQL features**
  - Built-in expressions and functions
  - Joins, Aggregations, Views
  - Stored Procedures, Triggers
  - Extensions like Foreign Data Wrappers (FDW)

- **Leverage Spanner-like DocDB features**
  - Linear Scalability
  - Fault Tolerance with high availability
  - Run natively in Kubernetes
  - Zero Downtime SQL database
    - Alter schema
    - Rolling software upgrades
    - Change machine types

DEMO

Try it yourself!

Questions?

Try it at docs.yugabyte.com/quick-start

Check us out on GitHub
https://github.com/YugaByte/yugabyte-db