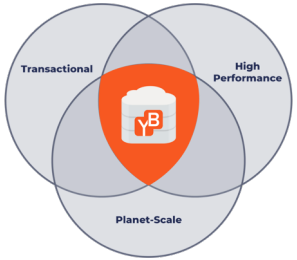


Building Cloud-Scale Apps with YugaByte DB

Karthik Ranganathan
Co-Founder/CTO, YugaByte
Feb 13, 2019

YugaByte

Introduction

Karthik Ranganathan

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

@karthikr

YugaByte DB is a modern NewSQL database

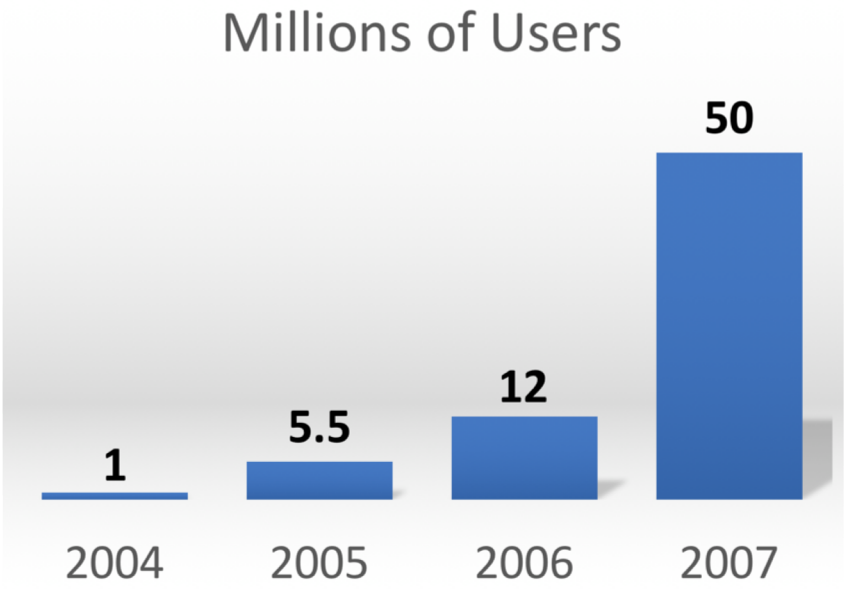
Distributed SQL + NoSQL

High Performance

Cloud-Native

WHAT PROBLEM ARE WE SOLVING?

YugaByte story starts with … Facebook in 2007



Facebook in 2008-2009.....

How to scale to a billion users?

Also: How to survive the week?

What happens at 1 billion users?

Dozens of petabytes

Billions of IOPS

Scale out frequently

Rolling upgrades – zero downtime!

Transformation of Facebook

How we achieved scale

Few datacenters

Geo-distributed DC's

Traditional servers

Containers for applications

Monolithic applications and services

Microservices based design

It's all about developer agility

How did the Tech Leaders simplify this?

facebook

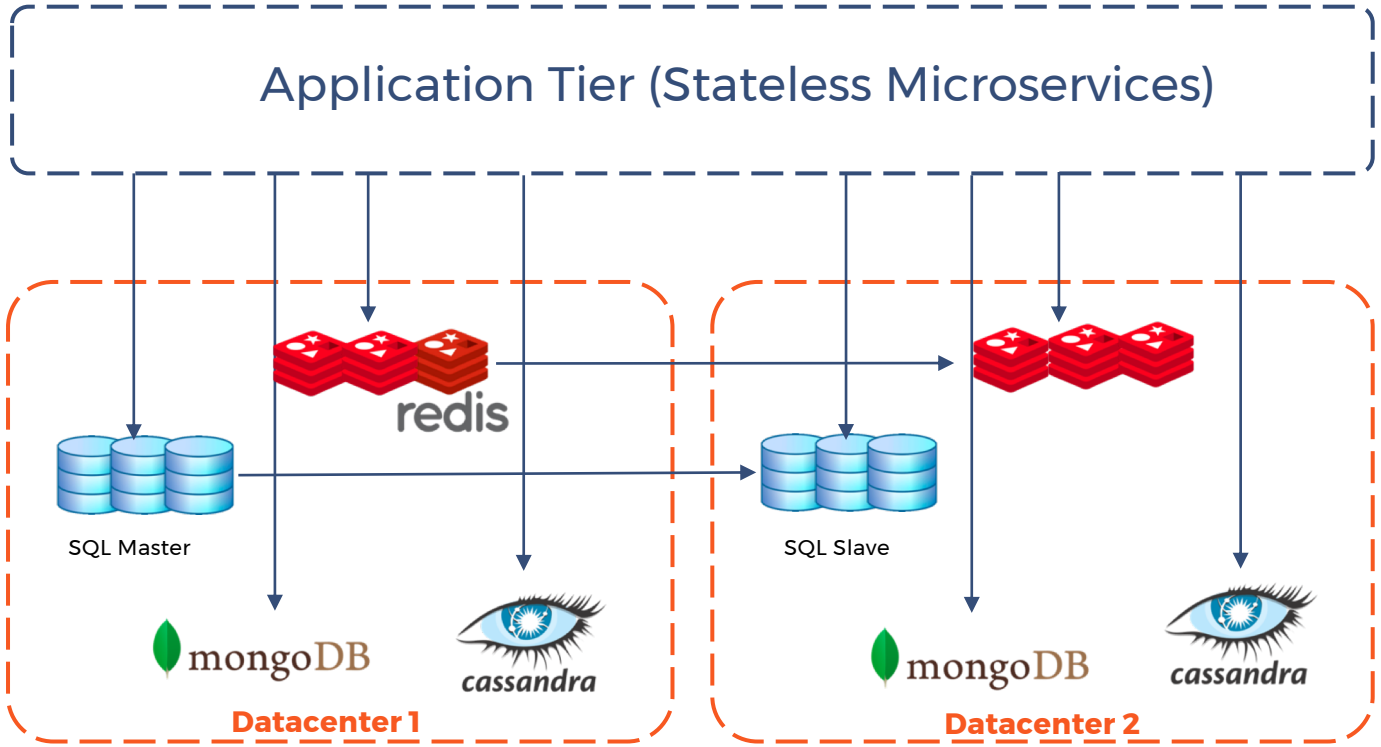
UBER

Pinterest

But there's no general platform for the enterprise

Cloud-Scale App Stack Today is Complex

Fragile infrastructure with many moving parts



Workload patterns in cloud-scale apps

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.

Transactional NoSQL

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

Scale-out SQL

YugaByte DB is Purpose-Built to Kill Complexity

Transactional, High-Performance Database for Building Internet-Scale, Globally-Distributed Apps

NoSQL + SQL

Data Modeling Freedom
(Key Value + Flex Schema + Relational)

Cloud Native

Multi-Cloud & Kubernetes-Ready
(PKS, K8S, AWS, GCP)

HOW DO WE SOLVE THIS PROBLEM?

High Level Architecture

Design is a Layered Approach

YUGABYTE QUERY LAYER

Extensible Query Layer

DISTRIBUTED, DOCUMENT STORE

Transactional, High-Performance, Geo-Distributed

RUN ON ANY HARDWARE/IAAS

High Level Architecture

Design is a Layered Approach

High Level Architecture

Design is a Layered Approach

YEDIS
Transactional NoSQL – Key Value

YCQL
Transactional NoSQL – Flex Schema

YSQL
Globally Distributed SQL

Self-Healing, Fault-Tolerant

High Throughput, Low Latency

ACID Transactions

Auto Sharding & Rebalancing

Global Data Distribution

RUN ON ANY HARDWARE/IAAS

High Level Architecture

Design is a Layered Approach

YEDIS — Transactional NoSQL – Key Value

YCQL — Transactional NoSQL – Flex Schema

YSQL BETA — Globally Distributed SQL

- Self-Healing, Fault-Tolerant
- High Throughput, Low Latency
- ACID Transactions
- Auto Sharding & Rebalancing
- Global Data Distribution

YugaByte Query Layer

Unparalleled data modeling freedom

YEDIS

Transactional NoSQL – Key Value

YCQL

Transactional NoSQL - Flexible Schema

YSQL BETA

Globally Distributed SQL

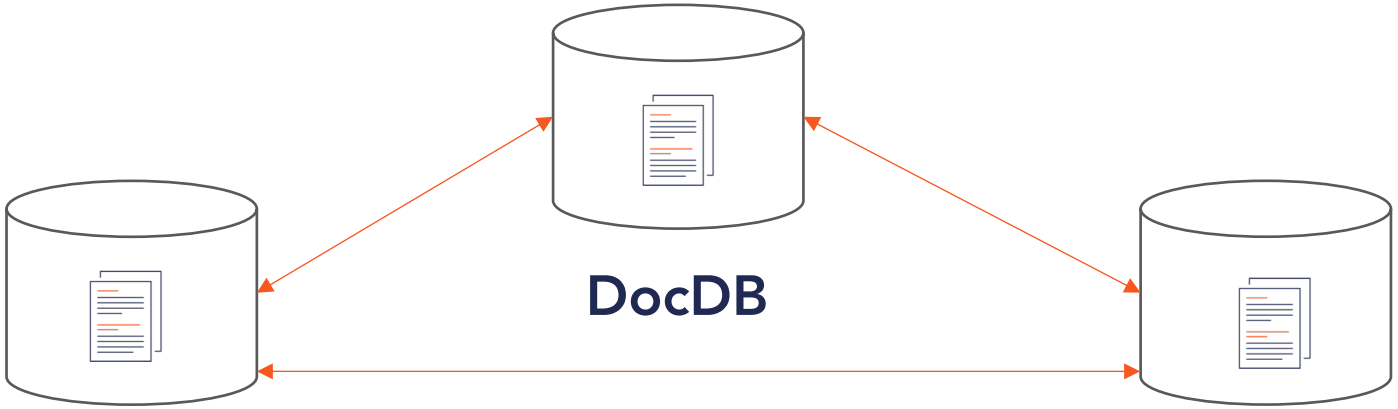
YugaBytE DIctionary Service	YugaByte Cloud Query Language	YugaByte Structured Query Language
Single Key ACID	Multi-Shard ACID	Multi-Shard ACID
Hash, Sorted Sets, Time Series Data Type	Secondary Indexes & Native JSON	JOINS & Data Integrity Constraints
Compatible with Redis commands	**Compatible with SQL-like Cassandra QL**	**Compatible with PostgreSQL**

SINGLE-KEY ACCESS ◄---- **DATA MODELING RICHNESS** ----► MULTI-KEY ACCESS

BLAZING FAST (SUB-MS) ◄---- **QUERY PERFORMANCE** ----► FAST (SINGLE-DIGIT MS)

Distributed Document Store

Bringing the Google Spanner architecture to commodity infrastructure



High Throughput, Low Latency

Full ACID Compliance

Global Data Distribution

Self-Healing, Fault-Tolerant

Auto Sharding & Rebalancing

Stores every key as a separate document, irrespective of the data model chosen

Distributed Document Store Features

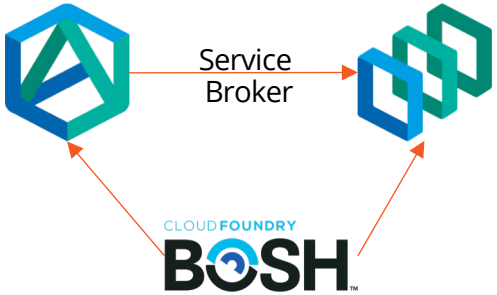
SQL	NoSQL
Strong consistency	Tunable read latency
Secondary indexes	Write optimized for large data sets
ACID transactions	Data expiry with TTL
Expressive query language	Scale out and fault tolerant

Cloud Native Operations

Multi-Cloud, Hybrid Cloud, Pivotal & Kubernetes-Ready

Multi-Cloud & Hybrid Cloud

Pivotal Cloud Foundry



Kubernetes

DESIGN GOALS

Consistency Goals – similar to Google Spanner

CAP

Consistency

Partition Tolerant

HA on failures – new leader elected in seconds

PACELC

No failure:
Low latency

On failure:
Trade off latency for consistency

API Goals – similar to Azure Cosmos DB

- ✓ Multi-model
- ✓ Start with well known APIs
- ✓ Extend to fill functionality gaps
- ✓ Offer uniform semantics across different APIs

Other Design Goals

- ✓ No dependencies on external systems
- ✓ All layers in C++ for high performance
- ✓ Power modern apps with one DB

DISTRIBUTED SQL IN YUGABYTE DB

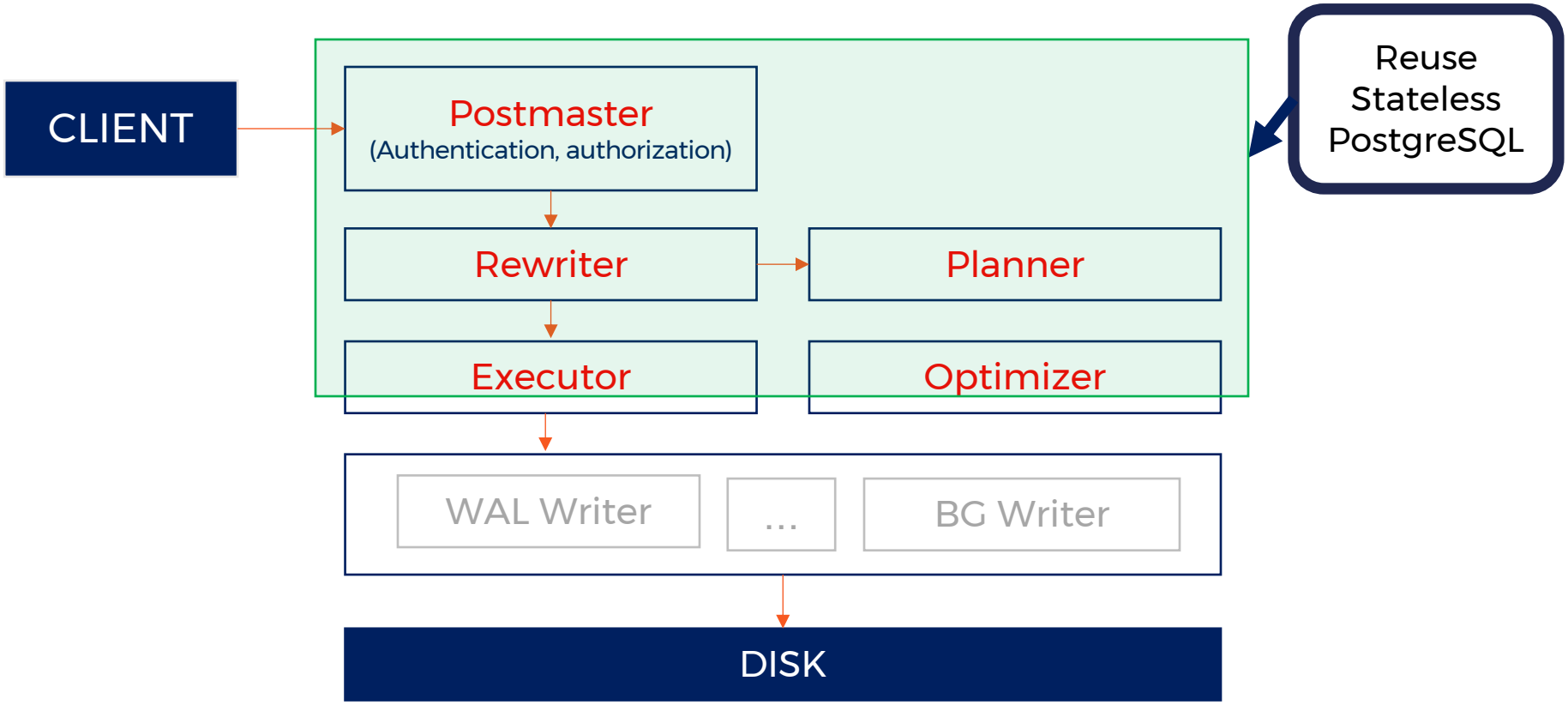
Wire compatible with PostgreSQL

- **PostgreSQL compatible**
  - Re-uses PostgreSQL code base
  - Rebase with newer PostgreSQL versions (e.g. PostgreSQL 10.4 → 11.0)

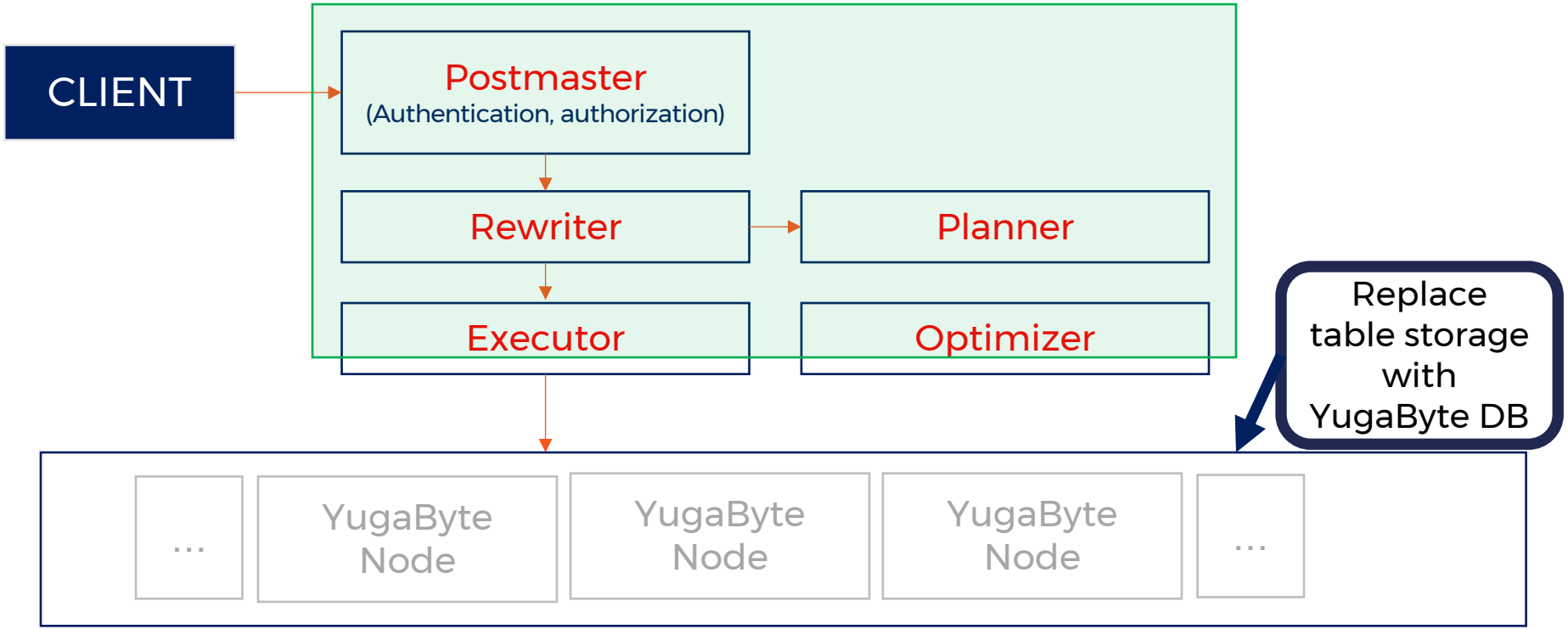
- **Core data engine**
  - Same underlying store as NoSQL
  - Distributed document store - DocDB
  - Database written in C++ for high performance

- **Cloud-native design:**
  - Designed for running natively in Kubernetes
  - Quickly scale-out to meet increased work loads
  - Multi—zone and geographically replicated deployments

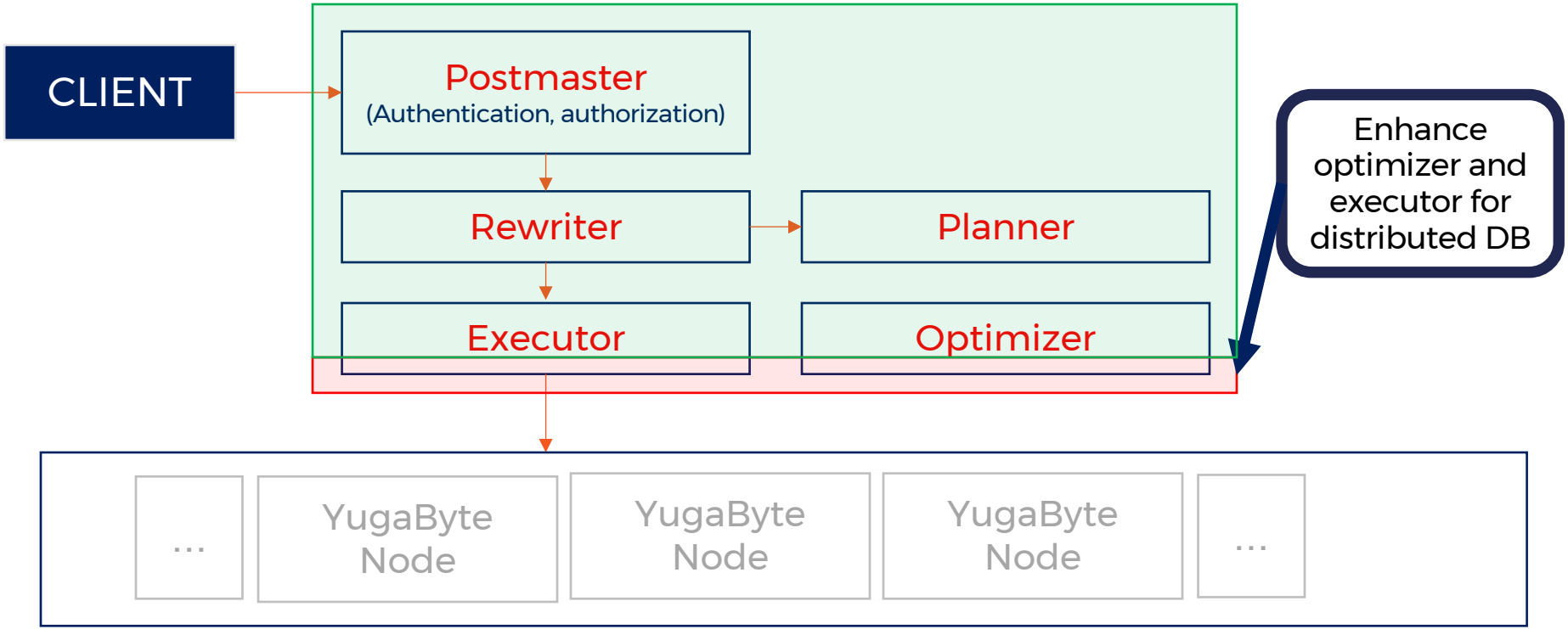
Re-use not Re-write of PostgreSQL



Re-use not Re-write of PostgreSQL



Re-use not Re-write of PostgreSQL



YugaByte PostgreSQL feature-set support

Expect to support most PostgreSQL features

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

NoSQL SUPPORT IN YUGABYTE DB

Wire-compatible with Cassandra and Redis

More Developer Agility:

- Extending Cassandra:
  - Strong consistency
  - Consistent secondary indexes
  - JSON data
  - Distributed transactions
- **10x** data per node
- Superior performance
  - **2.7x** throughput
  - **50%** lower P99 latency
  - Streaming ingest performance without separate SST table load pipeline

Less Operational Complexity:

- Fewer nodes
- Expand in **minutes not days**
- **Less time** maintaining, tuning and managing:
  - No read repairs or anti-entropy
  - No tombstones or deletes reappear
  - No garbage collection pauses
- Reduced RTO and RPO
  - More frequent backups

YugaByte DB: 10x More Data per Node

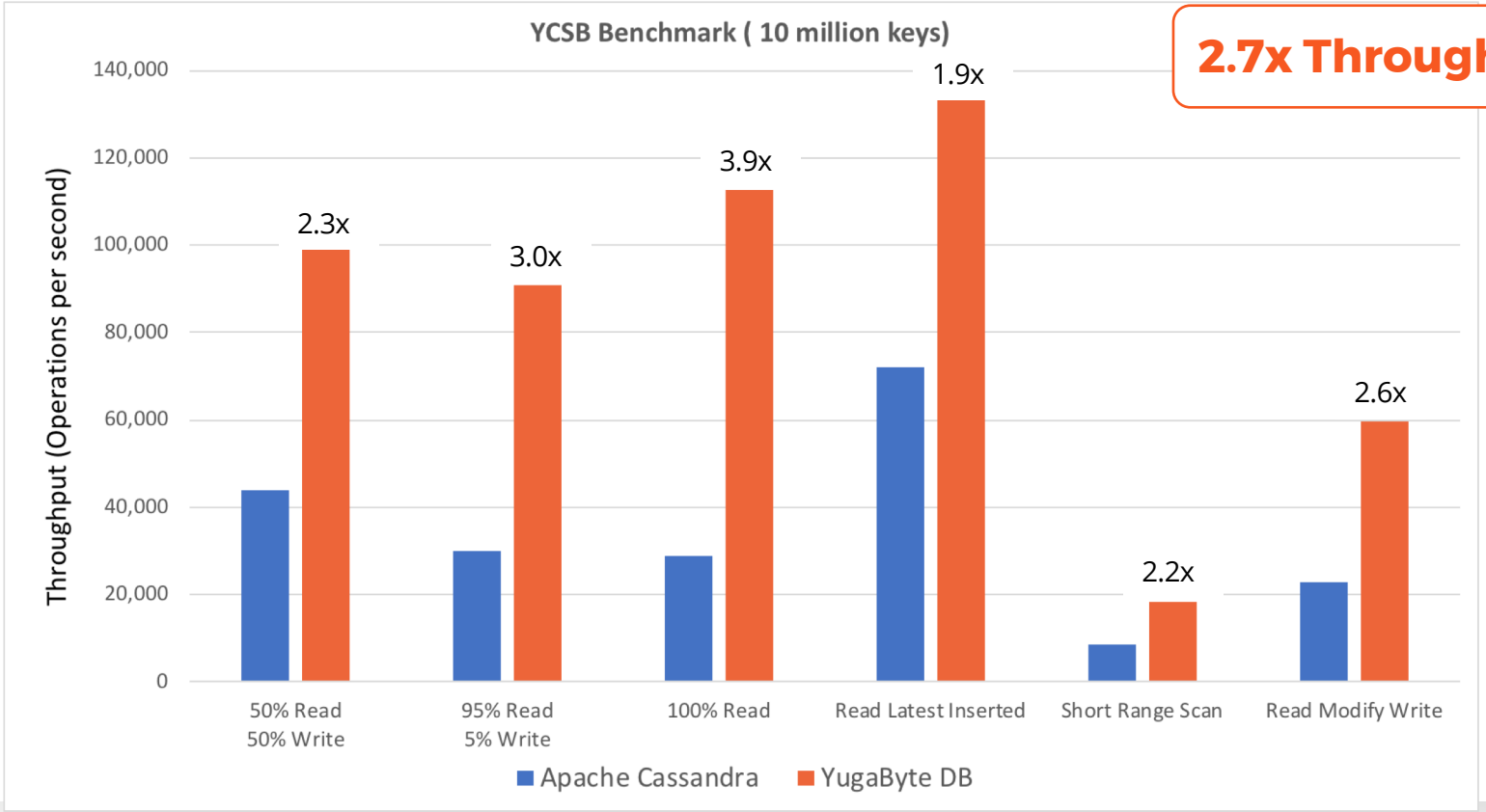
10x Density

Minutes not days

High-density benchmark:

- **26TB over 4 YugaByte DB nodes** compared to **30 nodes** for Cassandra
- **385K** reads/sec (0.25 ms) & 6.5K writes/sec on "Recent Data" Workload
- Expand to 5 nodes (complete in 8 hours), data available in **5 minutes**
- Induced node failure. **Cluster rebalanced in 2.2 hours**

YugaByte DB: Improved Apache Cassandra Performance



SIMPLIFYING CLOUD APPS

Distributed ACID Transactions

Multi-Row/Multi-Shard Operations At Any Scale

YCQL

```
cqlsh> CREATE TABLE banking.accounts (
  account_name varchar,
  account_type varchar,
  balance float,
  PRIMARY KEY ((account_name), account_type)
) with transactions = { 'enabled' : true };
```

```
BEGIN TRANSACTION
  UPDATE banking.accounts SET balance = balance - 200 WHERE account_name='John' AND account_type='checking';
  UPDATE banking.accounts SET balance = balance + 200 WHERE account_name='Smith' AND account_type='checking';
END TRANSACTION;
```

Secondary Indexes

Consistent & Low Latency

YCQL

```
cqlsh> CREATE TABLE store.orders (
  customer_id int,
  order_date timestamp,
  amount double,
  PRIMARY KEY ((customer_id), order_date)
) with transactions = { 'enabled' : true };
```

```
cqlsh> create index orders_by_date on store.orders (order_date, customer_id) covering (amount);
```

```
INSERT INTO store.orders (customer_id, order_date, amount) VALUES (1, '2018-04-02', 100.30);
INSERT INTO store.orders (customer_id, order_date, amount) VALUES (2, '2018-04-02', 50.45);
INSERT INTO store.orders (customer_id, order_date, amount) VALUES (1, '2018-04-06', 20.25);
INSERT INTO store.orders (customer_id, order_date, amount) VALUES (3, '2018-04-06', 200.80);
```

```
cqlsh> select sum(amount) from store.orders where customer_id = 1;

 sum(amount)
-------------
      120.55
(1 rows)
```

```
cqlsh> select sum(amount) from store.orders where order_date = '2018-04-02';

 sum(amount)
-------------
      150.75
(1 rows)
```

Native JSON Data Type

Modeling document & flexible schema use-cases

YCQL

```
cqlsh> CREATE TABLE store.books ( id int PRIMARY KEY, details jsonb );
```

```
cqlsh> SELECT * FROM store.books;
```

```
 id | details
----+-----------------------------------------------------------------------------------------------------------------------
  5 | {"author":"Stephen Hawking","genre":"science","name":"A Brief History of Time","year":1988}
  1 |                                    {"author":"William Shakespear","name":"Macbeth","year":1623}
  4 |          {"author":"Charles Dickens","genre":"novel","name":"Great Expectations","year":1950}
  2 |                                     {"author":"William Shakespear","name":"Hamlet","year":1603}
  3 |                {"author":"Charles Dickens","genre":"novel","name":"Oliver Twist","year":1838}

(5 rows)
```

Auto Data Expiry with TTL

Database tracks and expires older data

YCQL YEDIS

```
cqlsh:example> INSERT INTO employees(department_id, employee_id, name) VALUES (2, 2, 'Jack') USING TTL 10;

cqlsh:example> SELECT * FROM employees;

 department_id | employee_id | name
---------------+-------------+------
             2 |           1 |  Joe
             2 |           2 | Jack
             1 |           1 | John
             1 |           2 | Jane

cqlsh:example> SELECT * FROM employees; -- 11 seconds after the insert.

 department_id | employee_id | name
---------------+-------------+------
             2 |           1 |  Joe
             1 |           1 | John
             1 |           2 | Jane
```

Write a key with a 10 second expiry

```
127.0.0.1:6379> SET key "I expire in 10 seconds" EX 10
OK
```

Query the key right away

```
127.0.0.1:6379> GET key
"I expire in 10 seconds"
```

Query the key after 10 seconds

```
127.0.0.1:6379> GET key
(nil)
```

Native TimeSeries Data Type

Fine grained control on expiry of each record

YEDIS

Insert time-value data

```
> TSADD cpu_usage 201708110501 “80%” 201708110502 “60%”
“OK”
```

Delete time-value pairs

```
> TSREM cpu_usage 201708110501
“OK”
> TSGET cpu_usage 201708110501
(nil)
```

Query data in time windows

```
> TSRANGEBYTIME cpu_usage 201708110501 201708110503
1) 201708110501
2) “80%”
3) 201708110502
4) “60%”
```

Fine-grained expiry of each time-value pair

```
// This entry would expire in 3600 seconds (1 hour)
> TSADD cpu_usage 201708110504 “40%” EXPIRE_IN 3600
“OK”
// This entry would expire at the unix timestamp 1513642307
> TSADD cpu_usage 201708110505 “30%” EXPIRE_AT 1513642307
“OK”
```

Spark Integration for AI/ML

YCQL

Realtime analytics on top of transactional data without ETL

1

```
// Setup the local spark master, with the desired parallelism.
SparkConf conf = new SparkConf().setAppName("yb.wordcount")
                                .setMaster("local[1]")       // num Spark threads
                                .set("spark.cassandra.connection.host", hostname);

// Create the Java Spark context object.
JavaSparkContext sc = new JavaSparkContext(conf);

// Create the Cassandra connector to Spark.
CassandraConnector connector = CassandraConnector.apply(conf);

// Create a Cassandra session, and initialize the keyspace.
Session session = connector.openSession();
```

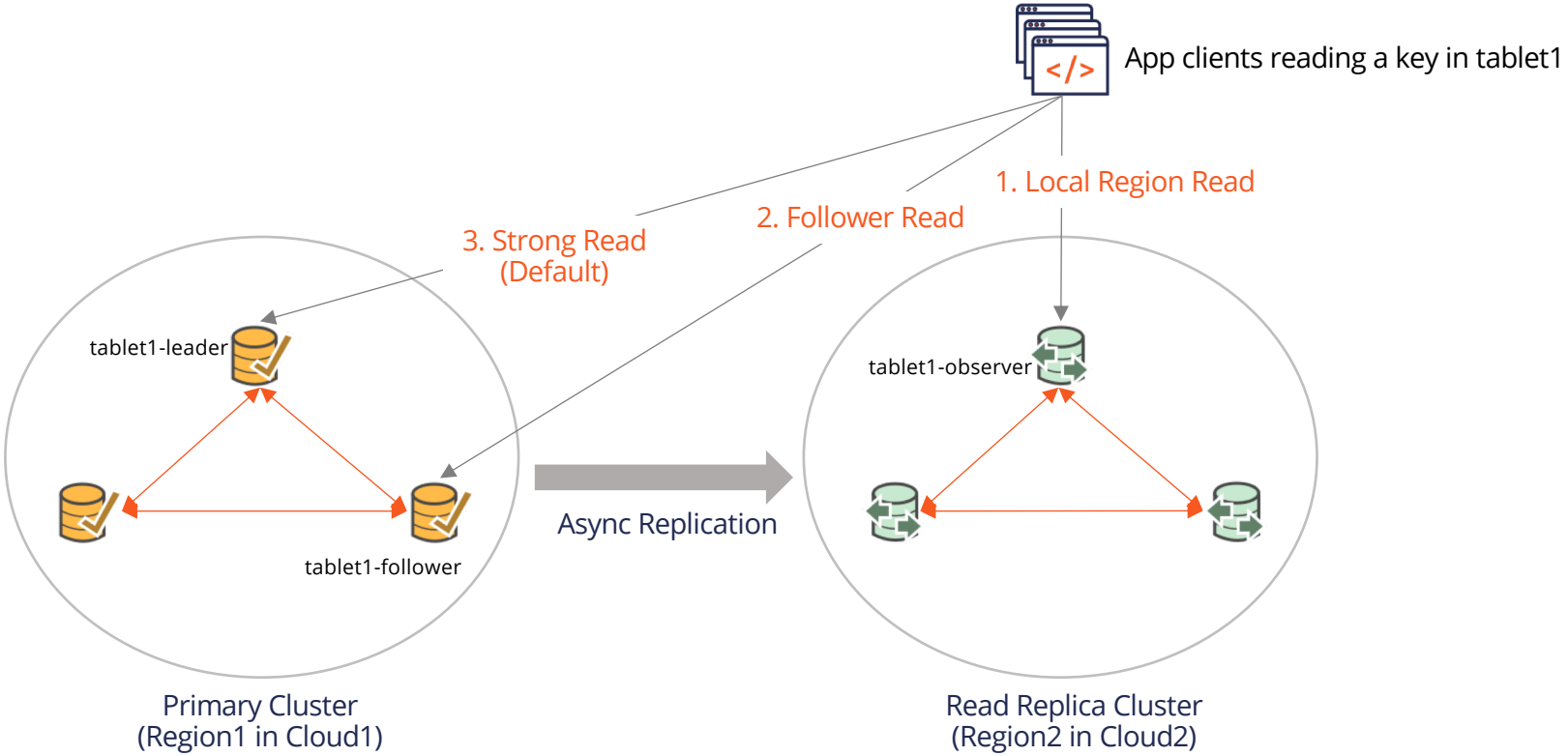
2

```
// Read rows from table and convert them to an RDD.
JavaRDD<String> rows = javaFunctions(sc).cassandraTable(keyspace, inputTable)
                                        .select("line")
                                        .map(row -> row.getString("line"));
```

3

```
// Save the output to the CQL table.
javaFunctions(counts).writerBuilder(keyspace, outputTable, mapTupleToRow(String.class, Integer.class))
                     .withColumnSelector(someColumns("word", "count"))
                     .saveToCassandra();
```

Tunable Read Latency



Questions?

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

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