Distributed PostgreSQL with YugaByte DB

Karthik Ranganathan
PostgresConf Silicon Valley
Oct 16, 2018
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?
A transactional, planet-scale database for building high-performance cloud services.
NoSQL + SQL

Cloud Native
WHY ANOTHER DB?
Typical Stack Today
Fragile infra with several moving parts

Application Tier (Stateless Microservices)

SQL Master
SQL Slave
redis
mongoDB
cassandra
mongoDB
cassandra

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?

Application Tier (Stateless Microservices)

Elasticache
Aurora
DynamoDB

Still Complex
it’s the same architecture
# System-of-Record DBs for Global Apps

<table>
<thead>
<tr>
<th>Not Portable</th>
<th>Open Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon DynamoDB</td>
<td><strong>cassandra</strong></td>
</tr>
<tr>
<td>Azure Cosmos DB</td>
<td><strong>mongoDB</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Not Portable</th>
<th>Open Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Aurora</td>
<td><strong>PostgreSQL</strong></td>
</tr>
<tr>
<td></td>
<td><strong>MySQL</strong></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Not Portable</th>
<th>Open Source</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Google Cloud Spanner</strong></td>
</tr>
</tbody>
</table>

**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

```
INSERT INTO T (K, V) values ('k', 'v');
```

```
hash_value = hash('k') = 0x1234
```

```
0x0000 0x1000 0x2000 0xE000 0xF000
```

```
tablet 2
```

```
0xFFFF
```
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

Postman
(Authentication, authorization)

Rewriter

Planner/Optimizer
Executor

Create Table Data File
Update System Tables

DISK
DDL Statements in YugaByte DB PostgreSQL

DDL

Postman
(Authentication, authorization)

Rewriter

Planner/Optimizer
Executor

Create sharded, replicated table as data source
Store Table Metadata in YB-Master (in works)

YugaByte master1
YugaByte master2
YugaByte master3
...

© 2018 All rights reserved.
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

- Postman (Authentication, authorization)
- Rewriter
- Planner/Optimizer
- Executor

External Database

Local Table Code Path

WAL Writer → BG Writer

Disk

FDW
DML Queries in YugaByte DB PostgreSQL

- Postman (Authentication, authorization)
- Rewriter
- Planner/Optimizer
- Executor
- Using FDW as a Table Storage API
- YugaByte DB Code Path
- YB Gateway

YugaByte node1 | YugaByte node2 | YugaByte node3 | YugaByte node4 | ...
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
  o Removed RocksDB WAL, re-uses Raft log
  o MVCC at a higher layer
  o Coordinate RocksDB memstore flushing and Raft log garbage collection

• File format changes
  o 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

One of tablet-peers for tablet 1
How Raft Replication Works

YB-Master
Manage shard metadata & coordinate system-wide ops

yb-master1
master-follower

yb-master2
master-follower

yb-master3
master-leader

1. Update record in tablet3

YB-TServer
Host & serve user data

yb-tserver1
tablet1-leader
tablet2-follower
tablet3-follower

yb-tserver2
tablet1-follower
tablet2-leader
tablet3-follower

yb-tserver3
tablet1-follower
tablet2-follower
tablet3-leader

2. Get latest tablet-leader locations and send to client for future caching

3. Redirect to current tablet3-leader

4. Sync update follower replicas via Raft

5. Wait for majority replicas to commit
   Commit to local DocDB
   Ack client after local commit
How Raft Replication Works

**YB-Master**
Manage shard metadata & coordinate system-wide ops

**YB-TServer**
Host & serve user data

1. Update record in tablet3
2. Get latest tablet-leader locations and send to client for future caching
3. Redirect to current tablet3-leader
4. Sync update follower replicas via Raft
5. Wait for majority replicas to commit
   - Commit to local DocDB
   - Ack client after local commit

Sync update follower replicas via Raft
How Raft Replication Works

YB-Master
Manage shard metadata & coordinate system-wide ops

yb-master1
master-follower

yb-master3
master-leader

yb-master2
master-follower

yb-tserver1
tablet1-leader
tablet2-follower
tablet3-follower

yb-tserver2
tablet1-follower
tablet2-leader
tablet3-follower

yb-tserver3
tablet1-follower
tablet2-follower
tablet3-leader

Sync update follower replicas via Raft

1. Update record in tablet3
2. Get latest tablet-leader locations and send to client for future caching
3. Redirect to current tablet3-leader
4. Sync update follower replicas via Raft
5. Wait for majority replicas to commit
   Commit to local DocDB
   Ack client after local commit

YB-TServer
Host & serve user data
How Raft Replication Works

YB-Master
Manage shard metadata & coordinate system-wide ops

YB-TServer
Host & serve user data

yb-master1
master-follower

yb-master2
master-follower

yb-master3
master-leader

yb-tserver1
tablet1-leader
tablet2-follower
tablet3-follower...

yb-tserver2
tablet1-follower
tablet2-leader
tablet3-follower

yb-tserver3
tablet1-follower
tablet2-follower
tablet3-leader...

1. Update record in tablet3

2. Get latest tablet-leader locations and send to client for future caching

3. Redirect to current tablet3-leader

4. Sync update follower replicas via Raft

5. Wait for majority replicas to commit
   Commit to local DocDB
   Ack client after local commit
Raft Related Enhancements

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

1. Acquire a lock on x
2. Read current value of x
3. Submit a Raft operation for replication: Insert (k1, v1) at hybrid_time 100
4. Replicate to majority of tablet peers
5. Apply to RocksDB and release lock

Raft Consensus Protocol

INSERT INTO table (k, v) VALUES ('k1', 'v1')

Lock Manager (in memory, on leader only)

DocDB / RocksDB

Tablet follower

Tablet follower

Raft log

k1,v1 @ht=100
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
  o Highest timestamp such that the view as of that timestamp is fixed
  o In the common case it is just before the hybrid time of the next uncommitted record in the tablet
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

YB Tablet Server 1
- YQL
  - Transaction Manager

YB Tablet Server 2
- Txn status tablet (leader)
- Txn status tablet (follower)
  (other tablet servers)
Distributed Transactions – Write Path Step 1: Client request

1. Client's request: set k1=v1, k2=v2

YB Tablet Server 1
- YQL
- Transaction Manager

YB Tablet Server 2
- Txn status tablet (leader)
- Txn status tablet (follower)
- Txn status tablet (follower)
- (other tablet servers)
Distributed Transactions – Write Path Step 2: Create status record

1. Client's request: set k1=v1, k2=v2

   YB Tablet Server 1
   YQL
   Transaction Manager

2. Create status record

   YB Tablet Server 2
   Txn status tablet (leader)

   Txn status tablet (follower)
   (other tablet servers)
Distributed Transactions – Write Path Step 2: Create status record

1. Client's request: set k1=v1, k2=v2

YB Tablet Server 1
- YQL
  - Transaction Manager

YB Tablet Server 2
- Create status record
  - Txn status tablet (follower)
  - Txn status tablet (leader)

YB Tablet Server 3
- Tablet follower
  - Tablet containing k1 (leader)
    - DocDB

YB Tablet Server 4
- Tablet follower
  - Tablet containing k2 (leader)
    - DocDB

© 2018 All rights reserved.
Distributed Transactions – Write Path Step 3: Write provisional records

1. Client's request: set k1=v1, k2=v2

   YB Tablet Server 1
   Transaction Manager
   YQL
   Write provisional records

2. Create status record
   YB Tablet Server 2
   Txn status tablet (follower)
   (other tablet servers)
   Txn status tablet (follower)

3. Write provisional records
   YB Tablet Server 3
   Tablet containing k1 (leader)
   Provisional record: k1 = v1 (txn=txn_id)
   Tablet follower
   Tablet follower
   Tablet follower

   YB Tablet Server 4
   Tablet containing k2 (leader)
   Provisional record: k2 = v2 (txn=txn_id)
   Tablet follower
   Tablet follower
   Tablet follower

© 2018 All rights reserved.
Distributed Transactions – Write Path Step 4: Atomic commit

1. Client's request: set k1=v1, k2=v2

2. Create status record

3. Write provisional records

4. Commit the txn
Distributed Transactions – Write Path Step 5: Respond to client

1. Client's request: set k1=v1, k2=v2

2. Create status record

3. Write provisional records

4. Commit the txn

5. Respond to client

- YB Tablet Server 1
  - YQL
  - Transaction Manager

- YB Tablet Server 2
  - Txn status tablet (leader)

- YB Tablet Server 3
  - Tablet containing k1 (leader)
    - Provisional record: k1 = v1 (txn=txn_id)
  - Tablet follower

- YB Tablet Server 4
  - Tablet containing k2 (leader)
    - Provisional record: k2 = v2 (txn=txn_id)
  - Tablet follower

- Tablet follower
  - Txn status tablet (follower)
  - Txn status tablet (follower)

© 2018 All rights reserved.
Distributed Transactions – Write Path Step 6: Apply provisional records

1. Client's request: set k1=v1, k2=v2

2. Create status record

3. Write provisional records

4. Commit the txn

5. Respond to client

6. Asynchronously apply provisional records

YB Tablet Server 1
- YQL
- Transaction Manager

YB Tablet Server 2
- Txn status tablet (leader)

YB Tablet Server 3
- Tablet containing k1 (leader)
  - Provisional record: k1 = v1 (txn=txn_id)
- Tablet follower

YB Tablet Server 4
- Tablet containing k2 (leader)
  - Provisional record: k2 = v2 (txn=txn_id)
- Tablet follower

Txn status tablet (follower)
(other tablet servers)
Isolation Levels

• Currently Snapshot Isolation is supported
  o Write-write conflicts detected when writing provisional records

• Serializable isolation (roadmap)
  o 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
  o Committed records the client might have seen must be visible

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

• Only affects multi-row reads of frequently updated records
Distributed Transactions – Read Path

YB Tablet Server 1

YQL Engine

Transaction manager

YB Tablet Server 2

Txn status tablet (leader)

txn_id: committed @ t=100

Txn status tablet (follower)

YB Tablet Server 3

Tablet containing k1 (leader)

Provisional record: k1 = v1 (txn=txn_id)

Tablet follower

Tablet follower

YB Tablet Server 4

Tablet containing k2 (leader)

Provisional record: k2 = v2 (txn=txn_id)

Tablet follower

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

1. Client’s request: read k1, k2

YB Tablet Server 1
- YQL Engine
- Transaction manager

YB Tablet Server 2
- Txn status tablet (leader)
- txn_id: committed @ t=100

YB Tablet Server 3
- Tablet containing k1 (leader)
- Provisional record: k1 = v1 (txn=txn_id)

YB Tablet Server 4
- Tablet containing k2 (leader)
- Provisional record: k2 = v2 (txn=txn_id)
Distributed Transactions – Read Path Step 2: Read from tablet servers

1. Client's request: read k1, k2

2. YB Tablet Server 1
   - YQL Engine
   - Transaction manager
   - Read k1 at hybrid time \textit{ht\_read}

2. YB Tablet Server 2
   - Txn status tablet (leader)
   - \texttt{txn\_id}: committed @ t=100
   - Txn status tablet (follower)

YB Tablet Server 3
- Tablet containing k1 (leader)
  - Provisional record: k1 = v1 (txn=\texttt{txn\_id})

YB Tablet Server 4
- Tablet containing k2 (leader)
  - Provisional record: k2 = v2 (txn=\texttt{txn\_id})

YB Tablet Server 2
- Tablet follower
- Tablet follower

YB Tablet Server 1
- Tablet follower
- Tablet follower
Distributed Transactions – Read Path Step 3: Resolve txn status

1. Client's request: read k1, k2

2. Read k1 at hybrid time \(ht_{\text{read}}\)
   - YB Tablet Server 1: YQL Engine
   - Tablet containing k1 (leader)
     - Provisional record: \(k1 = v1 (txn=\text{txn}\_id)\)

3. Request status of txn \text{txn}\_id
   - YB Tablet Server 2: Txn status tablet (leader)
     - \text{txn}\_id: committed @ t=100
   - YB Tablet Server 3: Tablet containing k1 (leader)
   - YB Tablet Server 4: Tablet containing k2 (leader)
     - Provisional record: \(k2 = v2 (txn=\text{txn}\_id)\)
   - Tablet follower
   - Tablet follower
   - Tablet follower

4. Read k2 at hybrid time \(ht_{\text{read}}\)
   - Tablet follower
Distributed Transactions – Read Path Step 5: Respond to client

1. Client's request: read k1, k2

2. Read k1 at hybrid time \(ht_{\text{read}}\)
   - Return \(k1 = v1\)

3. Request status of transaction \(txn_id\)
   - Txn status tablet (leader)
     - \(txn_id\): committed @ \(t=100\)
   - Txn status tablet (follower)

4. Return \(k2 = v2\)

5. Respond to client
Distributed Transactions – Conflicts & Retries

• Every transaction is assigned a random priority

• In a conflict, the higher-priority transaction wins
  o The restarted transaction gets a new random priority
  o 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