A PostgreSQL fork for horizontal scalability: YugabyteDB

Franck Pachot, Developer Advocate

yugabyteDB

@FranckPachot

Franck Pachot

Developer Advocate at Yugabyte

Past:

- 20+ years in databases, dev and ops, consulting
- Oracle ACE Director, AWS Data Hero
- Oracle Certified Master, AWS Database Specialty

fpachot@yugabyte.com
dev.to/FranckPachot
@FranckPachot

PostgreSQL for everything... at scale?

PostgreSQL is one of the most popular Open-Source database

- can run high throughput applications (if well-tuned)
- can run with good availability (failover with sync standbys)

🤔there are many extensions and forks to make it distributed:

- bi-directional replication: BDR/EDB PGD, pgactive, pgEdge
- single writer on distributed storage: Aurora, AlloyDB, Neon
- sharding: PostgresXL, Citus, Aurora Limitless
- distributed SQL: Spanner, CockroachDB, YugabyteDB, YDB

Why do they want to break the monolith?

Why break the monolith?

Cloud can provides multiple data centers, multiple regions

- can run clusters with all nodes active (availability, latency, data governance)

Cloud resources are expensive if not used with elasticity

- want to scale CPU / RAM / IO independently, without downtime

Managed services are responsible for operations

- must operate without downtime (resilience to failures, rolling upgrade online)

On-premises cloud-native (Virtualization, Kubernetes)

- infrastructure can scale horizontally if all pods are equal

PostgreSQL - what is monolithic?

You connect your application (ACID reads and writes) to one node
You read and write from shared buffers (shared memory)
The writes are protected from memory loss by one WAL (sync to disk)
Database files can scale (NAS, SAN, EBS) but are not a bottleneck (written asynchronously)

What is monolithic:
- connection is handled by one stateful process that does everything
- RAM can be shared between processors of only one server
  (not though common network)
- WAL is only one sequential stream because of ACID transactions

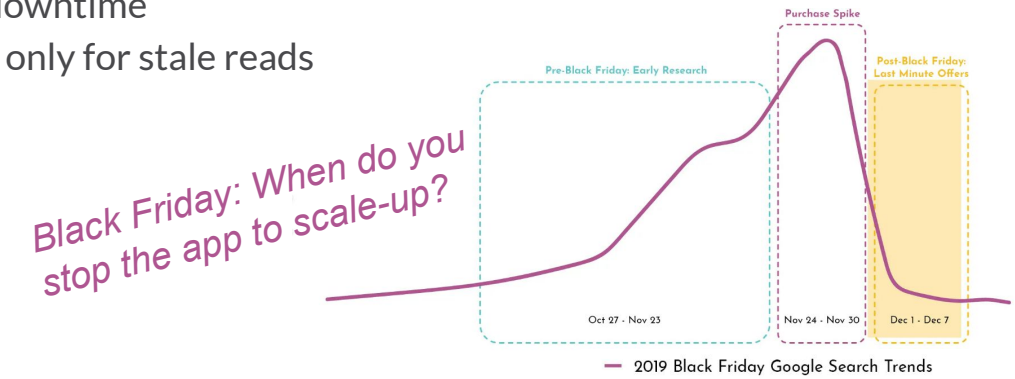
PostgreSQL - scalability / availability

In case of failure of the single read/write node:

- all current sessions receive an error and must re-connect
- failover takes time (split-brain detection, cold cache, re-connections)
  Can be fast but still longer than the application timeout

In case of workload increase in the single read/write node

- scale-up for more CPU/RAM requires downtime
- scale-out to additional servers possible only for stale reads



PostgreSQL - sharding

To scale horizontally, we must split the database

On top of multiple databases:

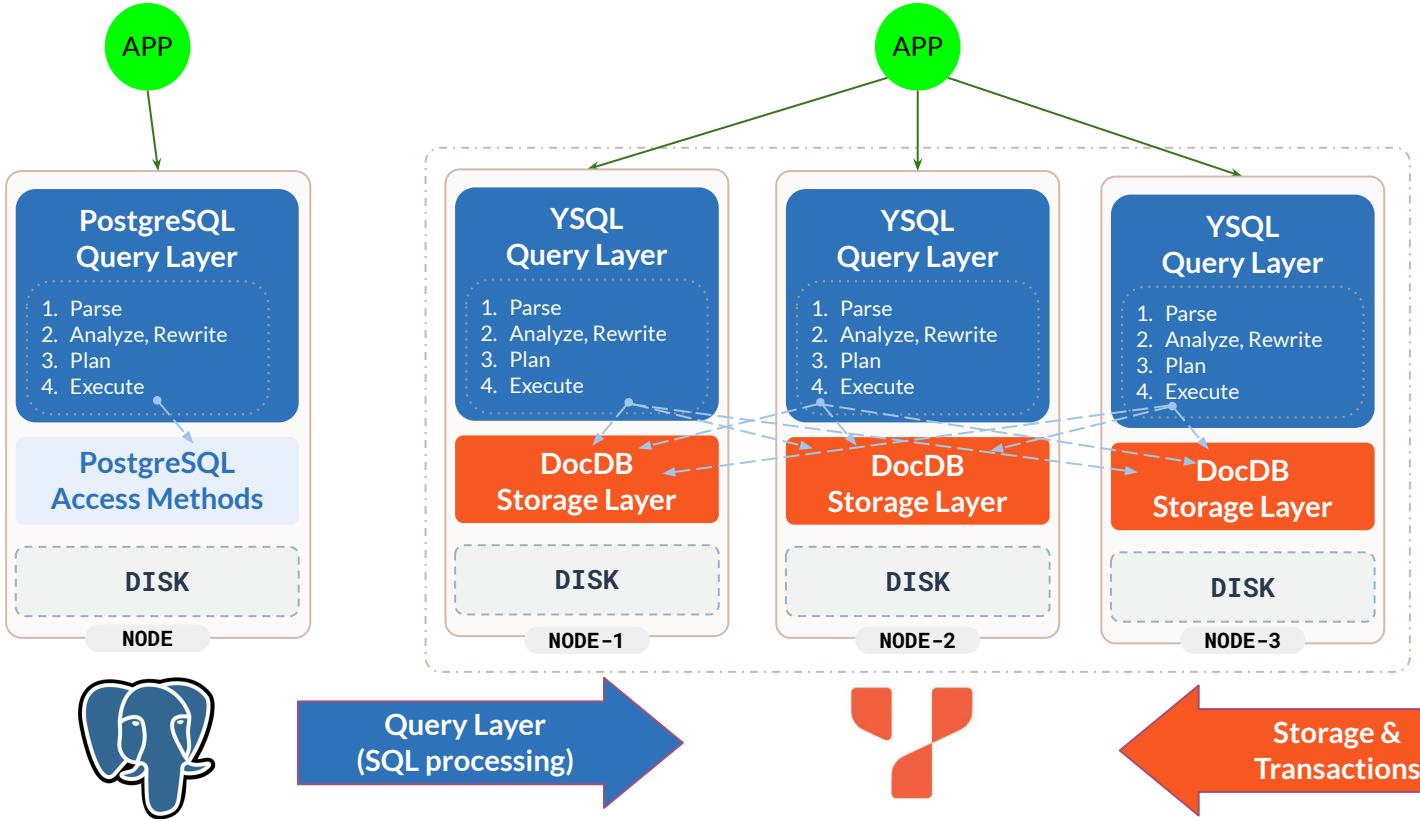
- by the application (more code, more tests)
- by a coordinator (Citus, Aurora limitless)

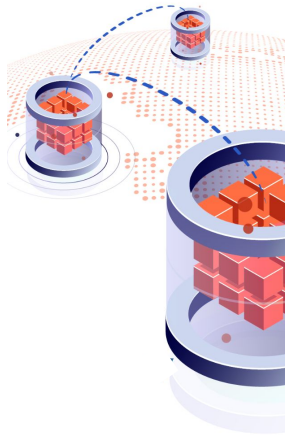
🤔 Local transactions (single-shard) are ACID but not global ones
(Lack of read consistency, global constraints, unique keys, foreign keys)

🤔 Improves scalability and availability but still not fully resilient, and hard to re-shard when scaling-out

Alternative: sharding within the transactional storage
(rows, index entries, transaction intents) 👉 Distributed SQL

Monolithic PostgreSQL ⋈ Distributed YugabyteDB





A fork of PostgreSQL for the query layer

Started with PostgreSQL 10

Currently, based on PostgreSQL 11

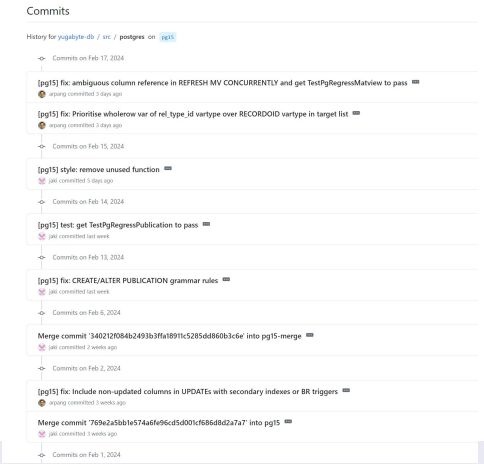
```
psql (16.0, server 11.2-YB-2.20.1.1-b0)
yugabyte=# select * from version();
                                                                                version
-----------------------------------------------------------------------------------------------------------------------------
 PostgreSQL 11.2-YB-2.20.1.1-b0 on aarch64-unknown-linux-gnu, compiled by clang version 16.0.6
```

Work In Progress: merging from PostgreSQL 15

https://github.com/yugabyte/yugabyte-db/tree/pg15/src/postgres

Goal: follow the latest PostgreSQL versions

but with new features controlled by flags (to allow rolling upgrades)



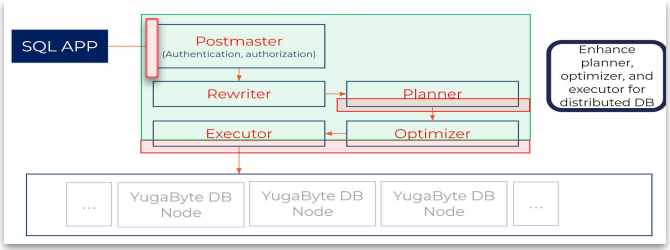
Why not an extension?

PostgreSQL extensibility is great:

- Foreign Data Wrapper
- Table/Index Access Methods
- Some hooks in the code for extensions

But also some limitations:

- expects Heap Tables
- expects tuples with TID, XID
- expects shared buffers (blocks)
- no hooks for WAL, syntax
- no threaded connections



YugabyteDB:

- pushdowns (FDW?), transaction intents (TableAM?)
- lot of batching (ex: Batched Nested Loop)
- additional syntax (ex: hash sharding)
- different cost model, different info in catalog
- threaded connections (connection manager)
- independent of system libraries (GLIBC)

What it looks like for the user?

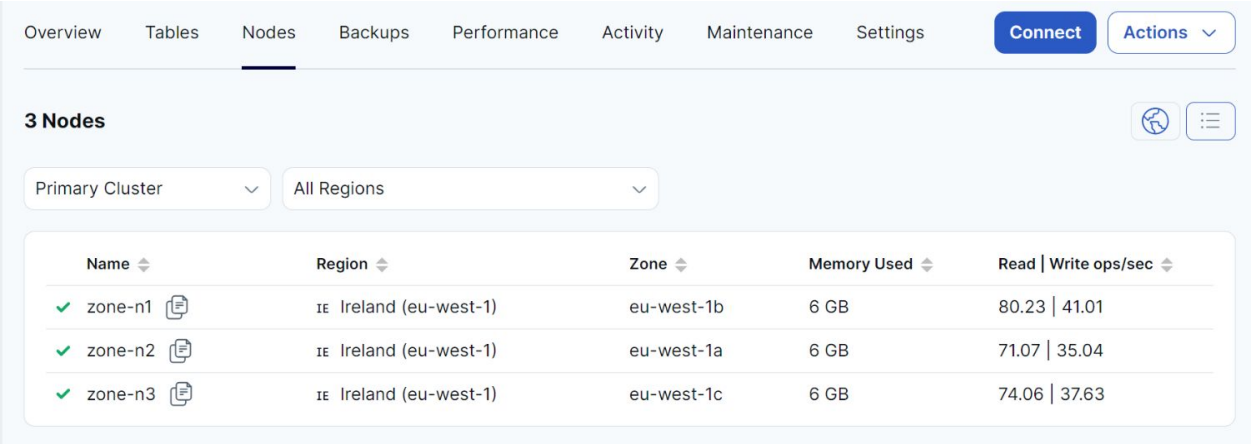
```
create table pgbench_accounts
 (aid int not null,bid int,abalance int,filler char(84))
 with (fillfactor=100);
alter table pgbench_accounts add primary key (aid);
copy pgbench_accounts from stdin;
```

```
yugabyte=# \! pgbench -i -IdtpGf -s100 --no-vac

yugabyte=# \d pgbench_accounts

             Table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 aid      | integer       |           | not null |
 bid      | integer       |           |          |
 abalance | integer       |           |          |
 filler   | character(84) |           |          |
Indexes:
 pgbench_accounts_pkey PRIMARY KEY, lsm (aid HASH)
Foreign-key constraints:
 pgbench_accounts_bid_fkey FOREIGN KEY (bid) REFERENCES pgbench_branches(bid)
Referenced by:
 TABLE pgbench_history CONSTRAINT pgbench_history_aid_fkey FOREIGN KEY (aid) REFERENCES pgbench_accounts(aid)
```

Primary key and Index use `HASH` for hash sharding, or `ASC`/`DESC` for range,
Auto-split on size, or add `SPLIT INTO 8 TABLETS`, `SPLIT AT ( (...),(...) )`

What it looks like for the system?

```
yugabyte=# \! pgbench -n -N -c 10 -T 60
```



Can connect to all nodes, Read and writes are balanced over all nodes

What it looks like for the application?

```
yugabyte=# create index acc_bal
           on pgbench_accounts(abalance ASC)
         -- split at values ( (-4000) , (-1000) , (0) , (1000) , (4000) )
;
```

```
yugabyte=# create extension pgcrypto;
yugabyte=# alter table pgbench_accounts  add column ext_id uuid;
yugabyte=# update pgbench_accounts  set ext_id=gen_random_uuid();
yugabyte=# create unique index acc_ext
           on pgbench_accounts  (ext_id hash) include  (aid);
```

SPLIT AT VALUES is optional (auto-splitting when table grows)

DDL is not (yet) transactional, but does not lock concurrent DML

(optimistic locking for no downtime migrations)

What it looks like for developer?

```
yugabyte=# explain ( analyze, dist, costs off, summary on)
           select from pgbench_accounts
           order by abalance asc fetch first 1000 rows only;
```

```
                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Limit (actual time=0.797..3.102 rows=1000 loops=1)
   ->  Index Only Scan using acc_bal on pgbench_accounts (actual time=0.796..3.035 rows=1000 loops=1)
         Heap Fetches: 0
         Storage Index Read Requests: 2
         Storage Index Read Execution Time: 2.717 ms
 Planning Time: 0.067 ms
 Execution Time: 3.173 ms
 Storage Read Requests: 2
 Storage Read Execution Time: 2.717 ms
 Storage Write Requests: 0
 Catalog Read Requests: 0
 Catalog Write Requests: 0
 Storage Flush Requests: 0
 Storage Execution Time: 2.717 ms
 Peak Memory Usage: 14 kB
(15 rows)
```

True Index Only Scan (no need to vacuum)

Read / Write requests are
- batched
- parallelized

between YSQL (PostgreSQL backend) and DocDB (Distributed Storage and Transaction) Raft leaders in **yb-tserver** nodes

catalog is on yb-master (cluster metadata)

Joins can scale in Distributed SQL

```
yugabyte=# explain ( analyze, dist, costs off, summary on)
           /*+ Set(yb_bnl_batch_size 1024 )*/
           select count(aid) from pgbench_history
           join pgbench_accounts using(aid) where delta>0;
```

```
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Aggregate (actual time=27.323..27.323 rows=1 loops=1)
   ->  YB Batched Nested Loop Join (actual time=17.209..27.205 rows=1657 loops=1)
         Join Filter: (pgbench_history.aid = pgbench_accounts.aid)
         ->  Seq Scan on pgbench_history (actual time=4.375..4.515 rows=1657 loops=1)
               Remote Filter: (delta > 0)
               Storage Table Read Requests: 1
               Storage Table Read Execution Time: 4.281 ms
         ->  Index Scan using pgbench_accounts_pkey on pgbench_accounts (actual time=10.269..10.565 rows=828 loops=2)
               Index Cond: (aid = ANY (ARRAY[pgbench_history.aid, $1, $2, ..., $1023]))
               Storage Table Read Requests: 1
               Storage Table Read Execution Time: 9.653 ms
 Planning Time: 0.495 ms
 Execution Time: 27.604 ms
 Storage Read Requests: 3
 Storage Read Execution Time: 23.587 ms
```

Joining 10k rows 176ms
👉 50k rows joins second

Shards are LSM-Trees (RocksDB with read optimizations)

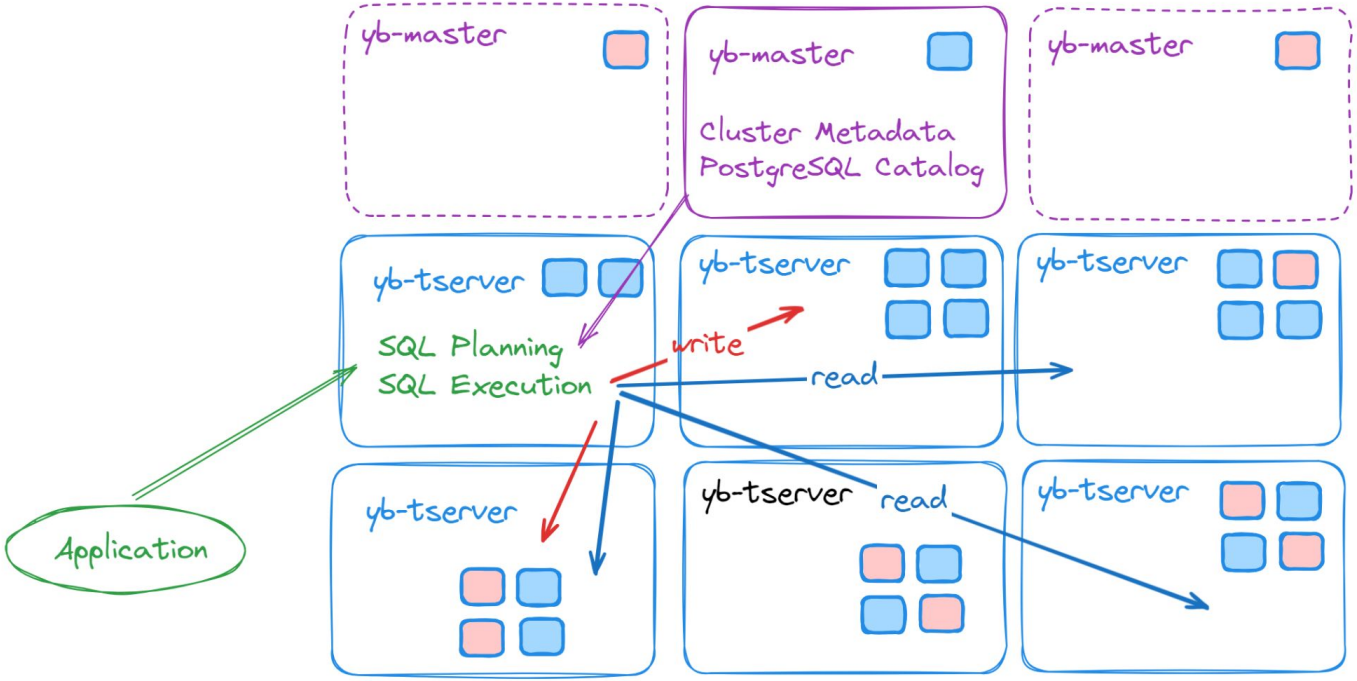
```
yugabyte=# explain ( analyze, dist, debug, costs off, summary off)
           select distinct aid
           from pgbench_accounts
           order by aid
;
```

```
                                              QUERY PLAN
-------------------------------------------------------------------------------------------------------------
Unique (actual time=1.088..73.485 rows=100000 loops=1)
  ->  Index Only Scan using pgbench_account_abal on pgbench_accounts (actual time=1.087..31.300 rows=100000 loops=1)
        Heap Fetches: 0
        Storage Index Read Requests: 98
        Storage Index Read Execution Time: 4.545 ms
        Metric rocksdb_number_db_seek: 98.000
        Metric rocksdb_number_db_next: 100097.000
        Metric rocksdb_number_db_seek_found: 98.000
        Metric rocksdb_number_db_next_found: 100096.000
        Metric rocksdb_iter_bytes_read: 4249346.000
        Metric docdb_keys_found: 100097.000
        Metric ql_read_latency: sum: 38190.000, count: 98.000
```

RocksDB LSM-Tree
- seek() to key (or key prefix)
- next() to read row/column versions

Rpc (network calls)

SQL execution, read/write to Tablets



A fork of PostgreSQL for the query layer

pg15 yugabyte-db / src / postgres / src / backend / access / yb_access /

jaki Merge commit '340212f084b2493b3ffa18911c5285dd860b3c6e' into pg15-merge

This branch is 159 commits ahead of, 868 commits behind `master`.

Name

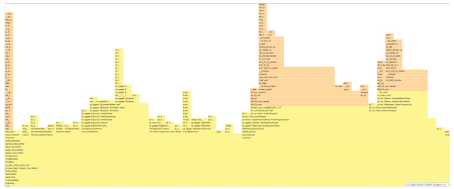
- ..
- Makefile
- yb_lsm.c
- yb_pg_inherits_scan.c
- yb_scan.c

```
1139
1140  /* ------------------------------------------------------------------
1141   *                  heap access method interface
1142   * ------------------------------------------------------------------
1143   */
1144
1145
1146  TableScanDesc
1147  heap_beginscan(Relation relation, Snapshot snapshot,
1148                 int nkeys, ScanKey key,
1149                 ParallelTableScanDesc parallel_scan,
1150                 uint32 flags)
1151  {
1152      HeapScanDesc scan;
1153
1154      /* YB scan methods should only be used for tables that are handled by YugaByte. */
1155      if (IsYBRelation(relation))
1156      {
1157          return ybc_heap_beginscan(relation, snapshot, nkeys, key, flags);
1158      }
1159
1160      /*
```

```
/*
 * Set up scan plan.
 * This function sets up target and bind columns for each type of scans.
 *    SELECT <Target_columns> FROM <Table> WHERE <Binds>
 *
 * 1. SequentialScan(Table) and PrimaryIndexScan(Table): index = 0
 *    - Table can be systable or usertable.
 *    - YugaByte doesn't have a separate PrimaryIndexTable. It's a special case.
 *    - Both target and bind descriptors are specified by the <Table>
 *
```

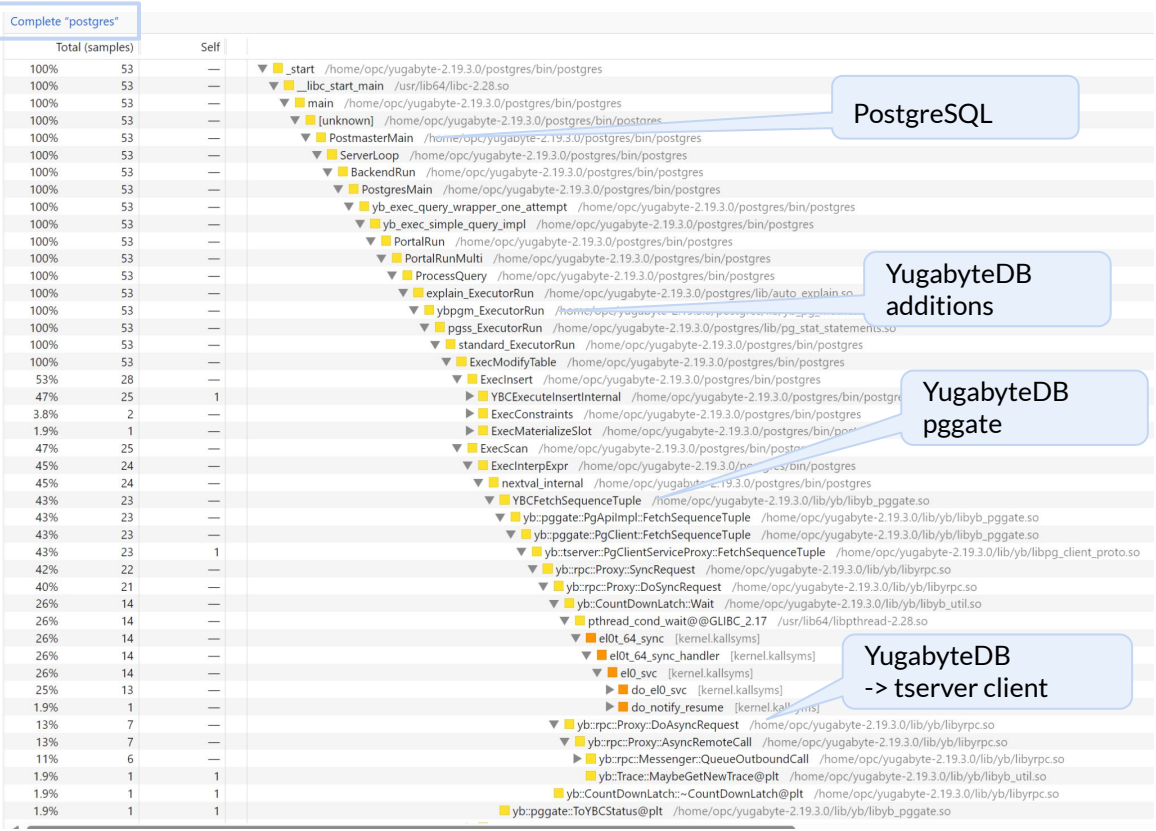
```
 * 3. IndexScan(UserTable, Index)
 *    - Both target and bind descriptors are specifed by the IndexTable.
 *    - For this scan, YugaByte returns an index-tuple, which has a ybctid (ROWID) to be used for
 *      querying data from the UserTable.
 *    - TODO(neil) By batching ybctid and processing it on YugaByte for all index-scans, the target
 *      for index-scan on regular table should also be the table itself (relation).
 *
 * 4. IndexOnlyScan(Table, Index)
 *    - Table can be systable or usertable.
 *    - Both target and bind descriptors are specifed by the IndexTable.
 *    - For this scan, YugaByte ALWAYS return index-tuple, which is expected by Postgres layer.
 */
```

YSQL (Query Layer)

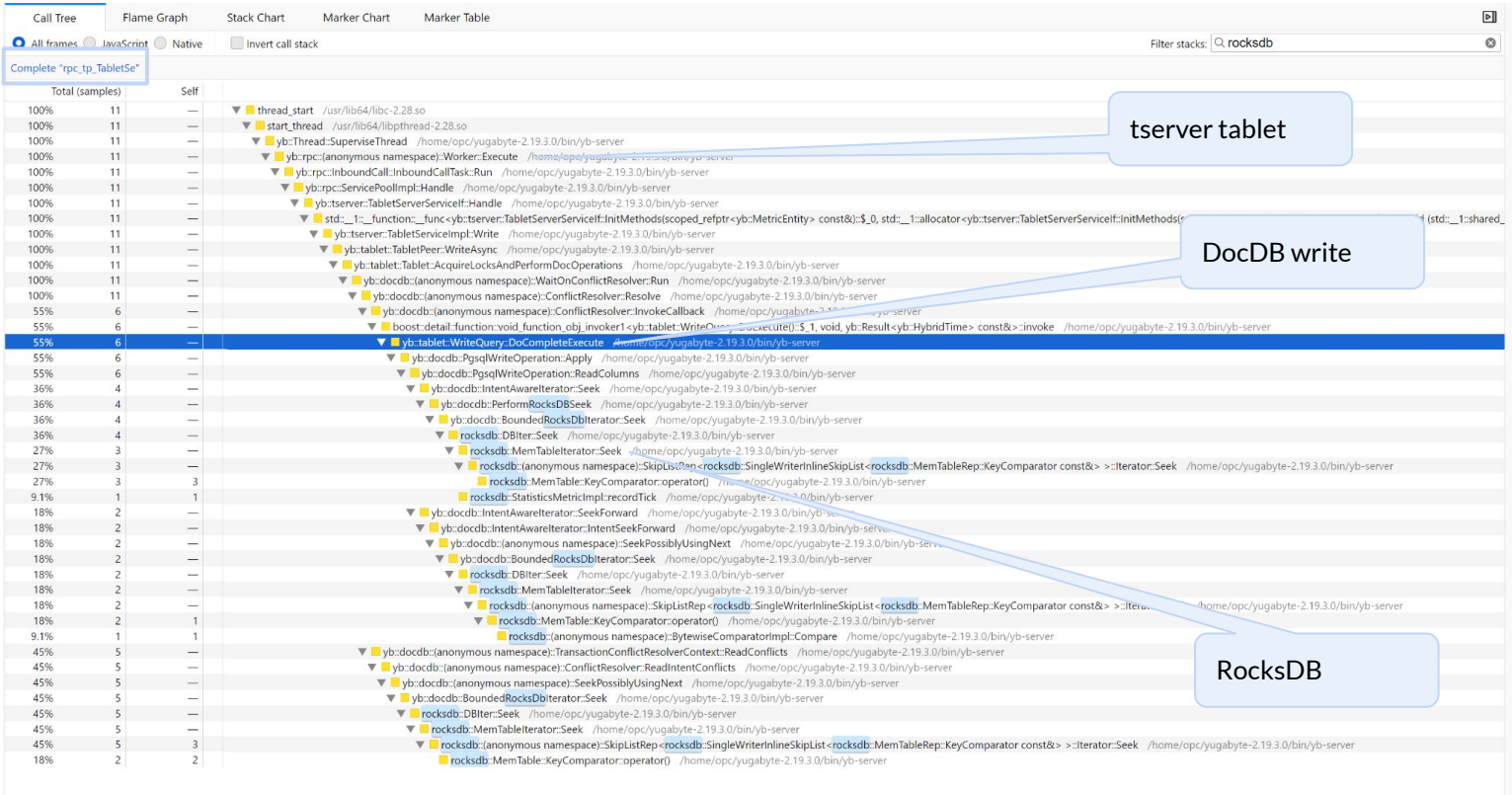


https://share.firefox.dev/3I5HS3e

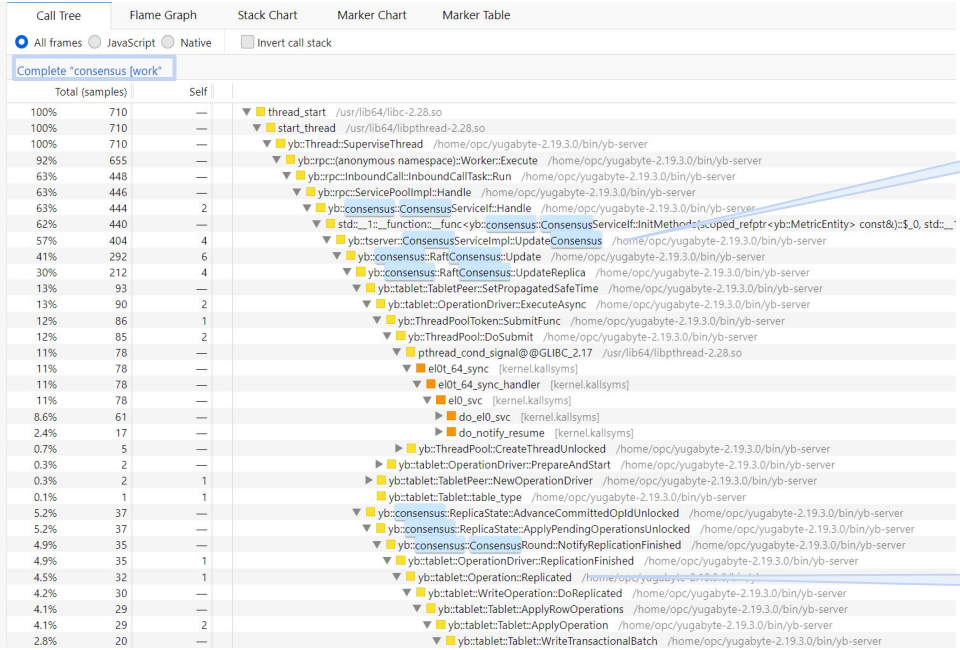
```
create table demo (
 k bigserial primary key
, v int default 0);
insert into demo
 select from generate_series
   (1,1000000);
\watch 0.001
```



DocDB (Storage and Transactions)



DocDB (Raft replication)



Update Raft Log

Commit write

Replication, Compaction, Auto-Splitting

Tablets are split on range of ASC/DESC primary key or range of HASH.

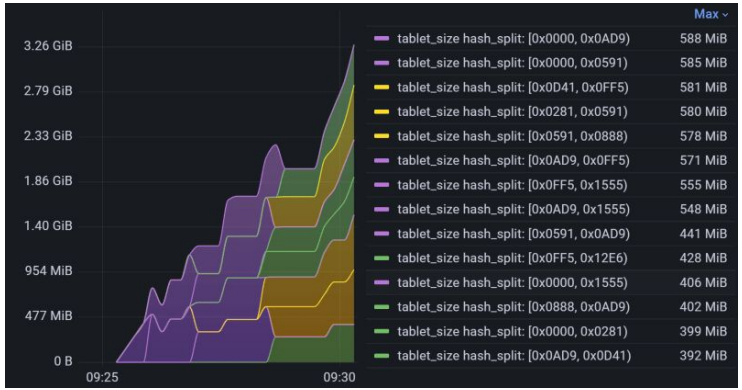
💡 Automatically split when growing

Each tablet has IntentsDB (provisional records, committed or not) and RegularDB (committed only) LSM-Trees (RocksDB).

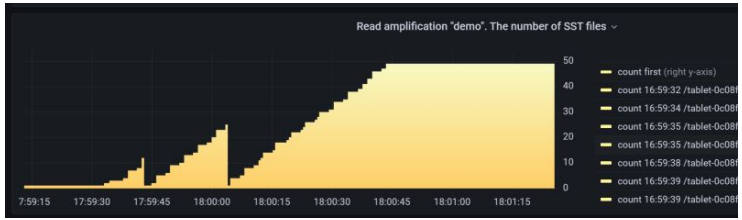
💡 Tablets are Raft group (read/write to leaders)

Each RocksDB has one writable MemTable, flushed to immutable SST Files + Level 0 universal compaction

💡 Compacted to reduce space and read amplification

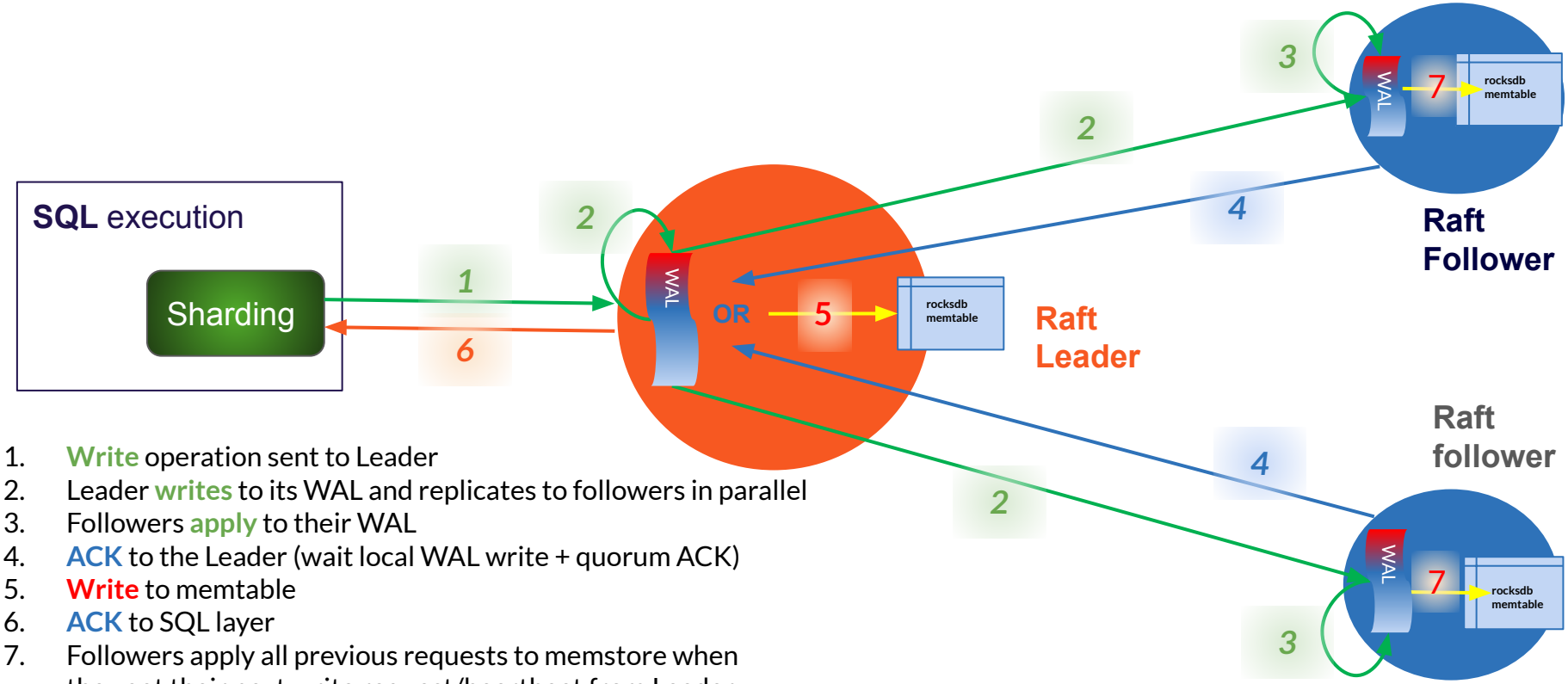


https://dev.to/yugabyte/testing-lsm-tree-merge-for-size-amplification-in-yugabytedb-2kh9



https://dev.to/yugabyte/yugabytedb-auto-sharding-2ahc

Raft replication

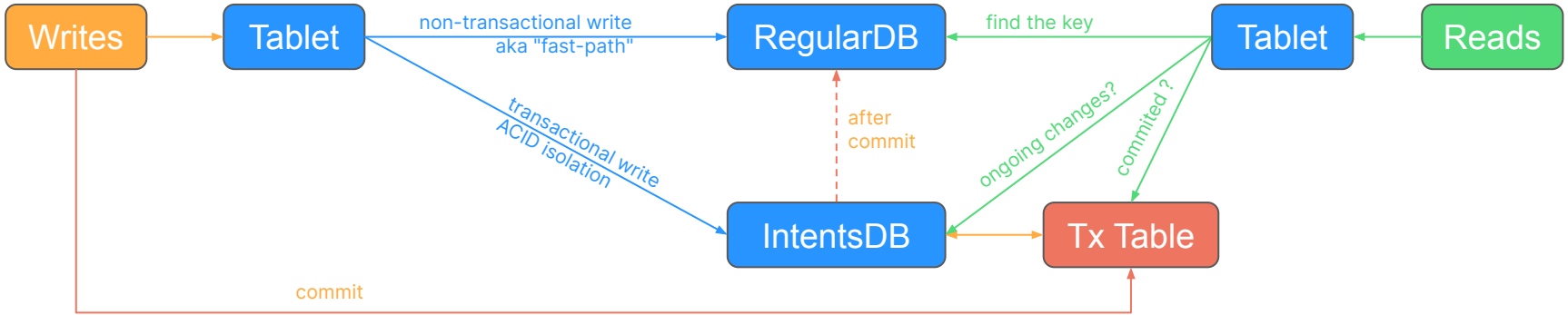


1. **Write** operation sent to Leader
2. Leader **writes** to its WAL and replicates to followers in parallel
3. Followers **apply** to their WAL
4. **ACK** to the Leader (wait local WAL write + quorum ACK)
5. **Write** to memtable
6. **ACK** to SQL layer
7. Followers apply all previous requests to memstore when they get their next write request/heartbeat from Leader

Tablets and data distribution

The key/value read and writes are per tablet (after sharding):

- Table rows and Index entries are distributed to tablets (by hash or range)
- Tablets are replicated to tablet peers (Raft groups)
- tablet peers have an IntentsDB to support transactional writes
- tablet peers store their committed versions in RegularDB
- compaction removes unnecessary intermediate versions (MVCC)



Advantage of a new storage layer

Beyond horizontal scalability - no vacuum problems 😎

- No bloat: MVCC version are stored per key, removed by RocksDB compaction
- Real Index Only Scans: indexes entries are versioned like tables
- Fast primary key access: index organized tables
- No transaction ID wraparound: clusterwide Hybrid Logical Clock ttimestamp
- transparent compression and encryption

Other advantages 😎

- Connection Manager: a threaded resident connection pool
- pg_hint_plan installed by default
- no downtime upgrades (rolling upgrades)
- rolling restart (online parameter change, key rotation, operating system patching

Reason for Distributed SQL

- scale-out for elasticity and resilience
- rolling upgrades, geo-distribution

Reason for PostgreSQL compatibility

- no need to learn a new DB, many SQL features

Reason for a fork

- the best compatibility with advanced features

Major difference with PostgreSQL:

- think more about the primary key (hash/range sharding)
- leverage new access patterns
  (primary index, skip scan, index only scan on secondary index)
- understand throughput vs. latency

When it can be an alternative to PostgreSQL:

- 24/7 system of records (OLTP)
- cloud native (Compute instances, kubernetes pods,... all active)
- multitenant (no hardware limitation to scale)
- geo-distribution (data residency, latency)

Not for datawarehouse (only some pushdowns for analytics)

E-mail:
fpachot@yugabyte.com

Blogs:
dev.to/FranckPachot
blog.yugabyte.com/author/fpachot

Twitter:
@FranckPachot

Youtube:
youtube.pachot.net

Twitch:
www.twitch.tv/franckpachot

LinkedIn:
www.linkedin.com/in/franckpachot

yugabyteDB