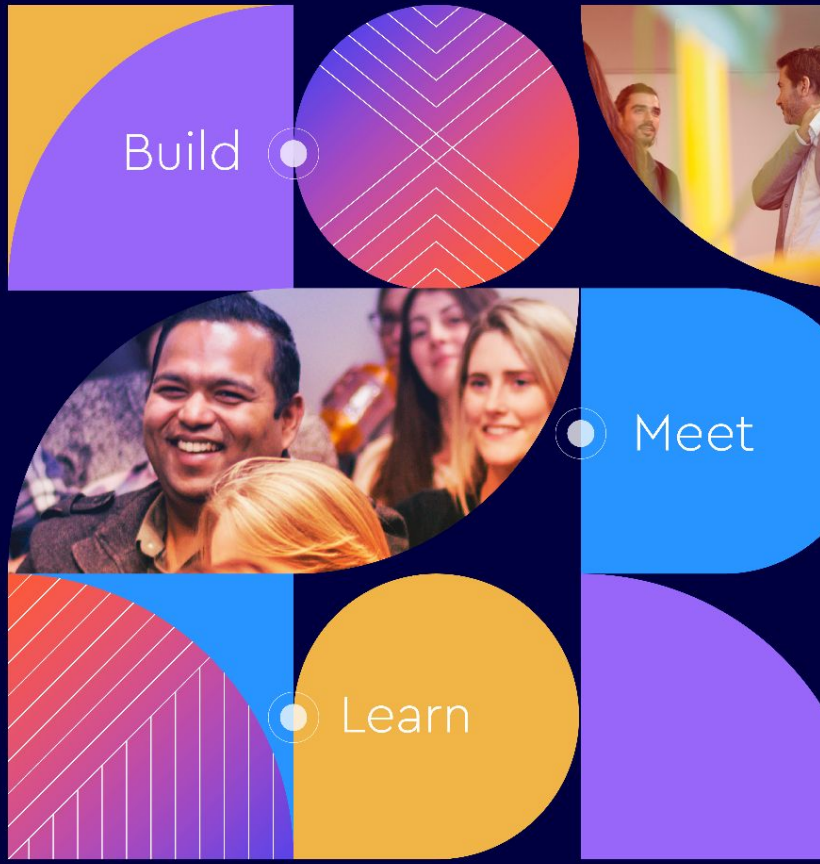


Performance

Accelerate Mission Critical Apps with our Top Tips and Tricks to Optimize Speed

Performance Tuning

Speed costs money, how fast do you want to go?

- The most obvious thing is to throw money at a problem in terms of infrastructure spend, but what we are exploring here is the time invested in optimizing an application/database system.
- There is no single magic bullet that fixes all problems
- The techniques discussed here are usually employed in depth via an iterative process.

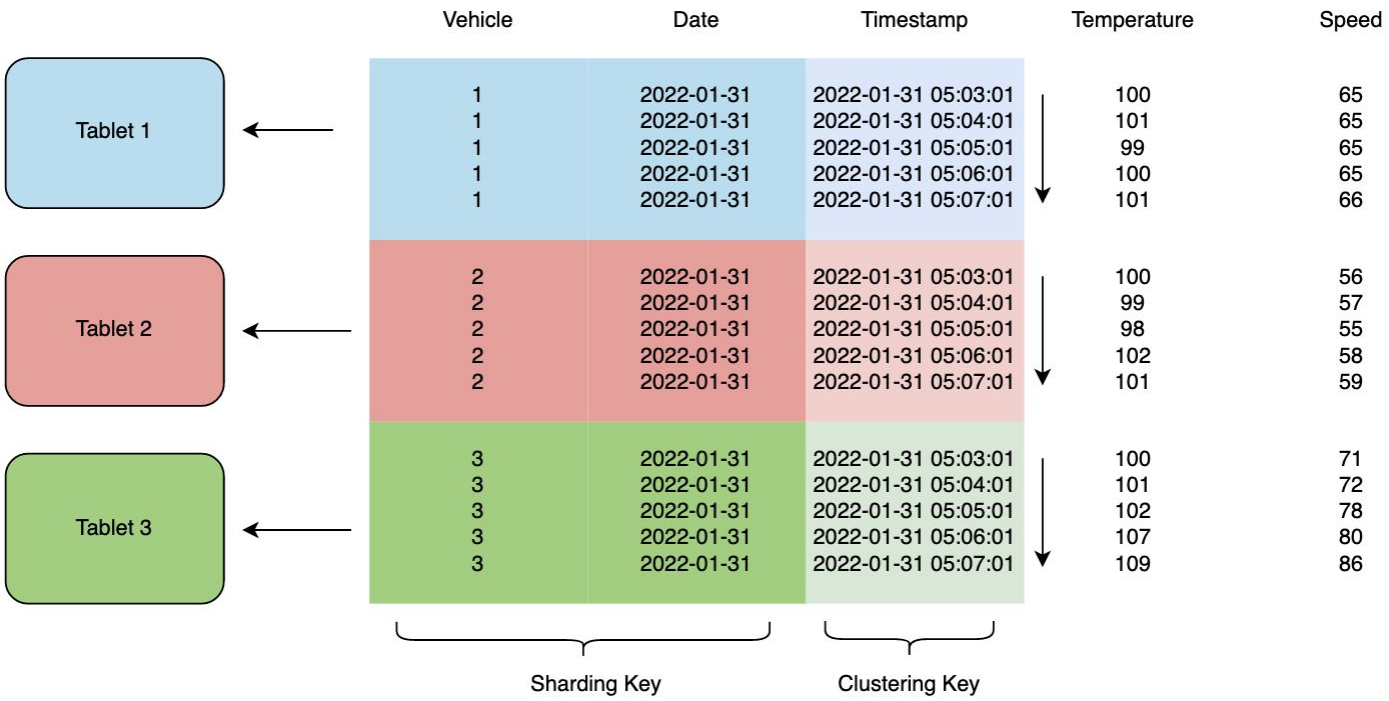
Performance Tuning

Areas of Focus

- Data modeling
- Performance Structures
- Flags and Session Tunables

Data Modeling

Performance: Data Modeling



Performance: Data Modeling

- Prefer defining a primary key for each table
  - YSQL tables are index-organized by the *primary key*
  - Ideally primary-key is the most-commonly used "index"

- Ideally each common query pattern can use an index or the primary key
  - e.g. Index columns should cover as much of the WHERE clause as possible
    - Rule of thumb: columns with stricter restrictions should come earlier in the key
    - e.g. **PRIMARY KEY(b, a)** is better if query pattern is **b = 1 AND a IN (1, ..., n)**

- For both primary key and index keys we support:
  - hash-based splitting (HASH)
    - default: **PRIMARY KEY(a, b) → PRIMARY KEY(a HASH, b ASC)**
    - multi-column: **PRIMARY KEY((a, b) HASH, c ASC)**
  - range-based splitting (ASC/DESC)
    - **PRIMARY KEY(a ASC, b DESC, c ASC)**

Performance: Data Modeling

Choice 1: Hash splitting

- Pros
  - Tables and indexes can be automatically pre-split
    - Ops can scale immediately
    - With a good hash key hot shard problem is rare
  - When a natural hash-key exists offers immediate and future-proof scalability

- Tradeoffs and concerns
  - HASH key needs to be fully specified to compute hash and identify target tablet
    - e.g. h1 = 1 AND h2 >= 2 → means fan-out query if (h1, h2) is the hash key
  - HASH component needs enough unique values to ensure good hash distribution
    - Else only a few tablets could be used, or they could be split unevenly
    - Rule of thumb: count *at least* an order of magnitude larger than tablet count

Performance: Data Modeling

Choice 2: Range Splitting

- Pros
  - More compatible with existing Postgres semantics
  - Can handle inequality or sorting (ORDER BY) conditions

- Tradeoffs and concerns
  - Cannot be automatically pre-split (split bounds depend on actual data)
    - Tablets will dynamically split once they become too large
  - Can be more susceptible to hot shards
    - size and IOPS-based dynamic splitting can mitigate the issue
  - A prefix of the key is needed to identify right tablet (or set of tablets)
    - e.g. r2 >= 2 will be a fanout query if key is (r1 ASC, r2 ASC)

Performance: Data Modeling

Trick: Change the primary key from a synthetic key to a natural key

- Let's say that I had the following table
  - CREATE TABLE users (userid BIGINT NOT NULL PRIMARY KEY, email_address TEXT NOT NULL, name TEXT NOT NULL, ...)
  - CREATE UNIQUE INDEX email_users ON users(email_address);
- For the sake of argument, I will also have a FK on the userid column to multiple tables
- If 90% of my queries use email_address as the key, then I have 2 RPCs for query, one for the index and then another for the attributes in the main table

Performance: Data Modeling

Trick: Change the primary key from a synthetic key to a natural key

- To save this extra RPC without changing the application or the structure of the database:
  - CREATE TABLE users (userid BIGINT NOT NULL, email_address TEXT NOT NULL, name TEXT NOT NULL, ..., PRIMARY KEY (email_address))
  - CREATE UNIQUE INDEX userid_users ON users (userid)

Performance: Data Modeling

Trick: Group less often referenced columns into JSONB structures

- Each column is a separate write into DocDB - So more columns = more writes. This is less true starting in 2.15 with “packed” rows, but still a useful technique.
- Multiple columns can be condensed into the JSONB column and still be referenced and indexed. It also allows for a degree of flexibility during schema evolution since new columns can be added without DDL updates.
- JSONB columns can be included in covering indexes

Performance: Data Modeling

Tip: Use the right number of tablets for each table

- If the table is less than 1 million rows and it is a low velocity table, place it either in it's own tablet or in a colocated tablespace (E.g. CREATE TABLE xxx SPLIT INTO 1 TABLETS)
- If the table is between 1 and 10 million rows and it is a low->medium velocity table, allocate 1 tablet per node.
- If the table is over 10 million rows and/or is high-velocity allocate multiple tablets per node. Scale the number of tablets/node based on the magnitude of how many rows it is.

Performance: Data Modeling

Tip: Things to avoid / be cautious about

- Sequences and serial columns should be avoided. If you must use them for backward compatibility/application integrity, make sure to **ALTER SEQUENCE seq_name CACHE** n, where n should be the number of INSERTS you expect to do per minute.
  - Formerly, sequences were stored on the master tablet in yb-master, but now are stored and cached on the tservers, but still can create hot-spots or delays in fetching new values
  - Use guid/uuid where possible
- If you are placing indexes on DATE/TIMESTAMP columns, do not make them the HASH key. Use RANGE sharding discussed earlier.

Performance: Data Modeling

Tip: Manage Cluster configuration with tablespaces

Leader Affinity

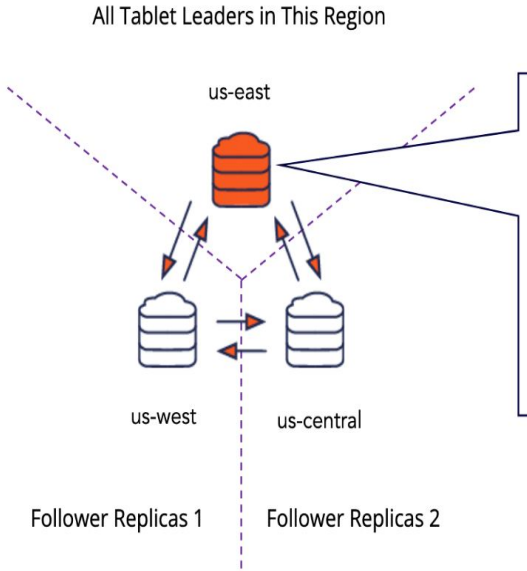
- Set up fine-grained leader priority to a particular region
- Can ensure proxy and data nodes are often or always in the same region
  - Reduces latency and network cost

Geo-partitioning

- Split a logical table into multiple partitions (one per region)
- Each partition can be pinned to its specific region
- Ensure multi-region application can operate with minimal latency

Performance: Data Modeling

Tip: Tablespaces with Leader Affinity



1. Pin all tablet leaders across all tables to a single region, the one where the majority of read requests originate.

2. All reads become blazing fast since no cross-region communication is needed in the read path.

3. Example workload - Loyalty app for a fast food restaurant located in the Eastern US

Performance: Data Modeling

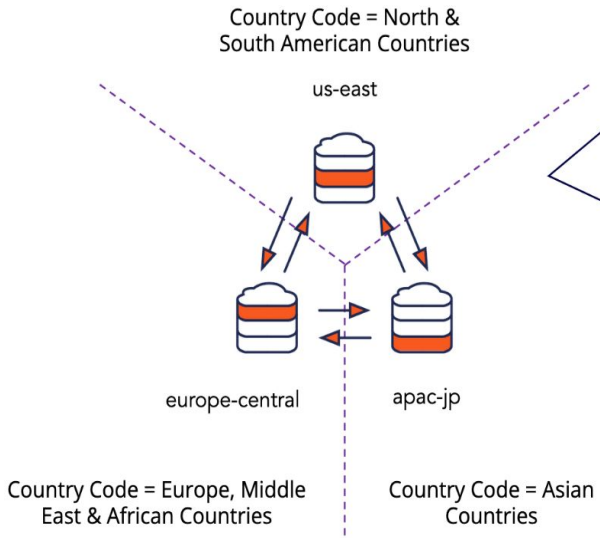
Tip: Manage Cluster configuration with tablespaces (Leader Affinity)

```
CREATE TABLESPACE us_west_2_tablespace WITH (replica_placement='{"num_replicas": 3,
"placement_blocks":
[{"cloud":"aws","region":"us-west-2","zone":"us-west-2a","min_num_replicas":1,"leader_preference":1},
{"cloud":"aws","region":"us-west-1","zone":"us-west-1a","min_num_replicas":1,"leader_preference":2},

{"cloud":"aws","region":"us-east-1","zone":"us-east-1a","min_num_replicas":1}]}');
```

Performance: Data Modeling

Tip: Tablespaces with Geo-partitioning



1. Single table to store global customer data with each customer row also storing their country of residence.

2. Pin customer rows to the nearest geo/region they reside in.

3. End result is reads from that region never experience cross-region latency (because of strong consistency w/o quorum).

4. Example workload - user profile of a global web conferencing app

Performance Structures

Performance: Performance Structures

Tip: Use covering indexes to reduce the number of fetches to the base table

- INCLUDE syntax can be used to add regular (non-key) columns to a secondary index
- If an index contains all selected columns it can skip the table lookup
  - Can reduce the number of RPCs by around half for a typical index read
- Tradeoff: An index needs to be updated when either the indexed or included columns are modified

```
CREATE TABLE t2(k int PRIMARY KEY, v1 int, v2 int, v3 int);
CREATE INDEX ON t2(v1) INCLUDE (v3);

EXPLAIN SELECT v3 FROM t2 WHERE v1 = 10;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using t2_v1_v3_idx on t2  (cost=0.00..5.12 rows=10 width=4)
 Index Cond: (v1 = 10)
 -- Go to the index to identify the row and get the value of v3
```

Performance: Performance Structures

Trick: Use partial indexes to reduce the number of writes and size of indexes

- WHERE syntax can be used to only index a subset of the rows
  - Query planner will know to only use the index if the WHERE clause matches
- Reduce index size and overhead
- Useful when index-based search always sets a particular filter
  - e.g. Avoid indexing `null` values if we always search for non-null values

```
CREATE TABLE t2(k int PRIMARY KEY, v1 int, v2 int, v3 int);
CREATE INDEX ON t2(v1) INCLUDE (v3) WHERE v1 IS NOT NULL;

INSERT INTO t2 VALUES (1, null, 2, 3); -- skip write to the partial index

EXPLAIN SELECT v3 FROM t2 WHERE v1 = 10;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Only Scan using t2_v1_v3_idx on t2  (cost=0.00..4.90 rows=10 width=4)
   Index Cond: (v1 = 10)
-- Condition implies target v1 is not null, so partial index is applicable
```

Performance: Performance Structures

Trick: Use duplicate indexes in zonal/regional tablespaces to force local reads

- We can create duplicate indexes in different geographies to ensure local reads that are strongly consistent.
- Useful for small reference data type tables.
- There is a marginal uplift in write latency for each additional index
- Index-only tables are limited to 32 columns total (PG limitation)

Flags and Session Tunables

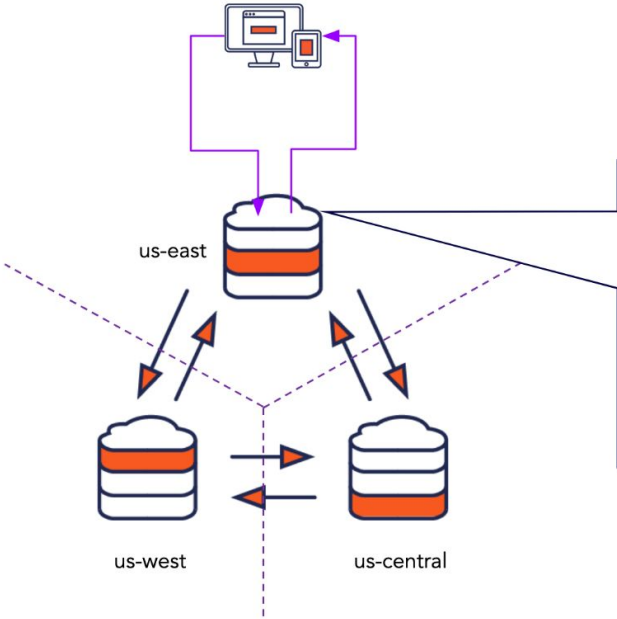
Performance: Flags and Session Tunables

Tip: Use yb_enable_expression_pushdown

- Processing query conditions using index (WHERE clause)
  - DocDB scanning the minimal subset of rows based on the predicate
  - In-memory filter (remote or local) of additional non-index columns
  - Pushes expressions down to DocDB on remote nodes to avoid filter when coming back to the local node
- Enabled at the session level
  - SET yb_enable_expression_pushdown to on;
- Can also be enabled universe-wide by using Gflag *ysql_pg_conf_csv*

Performance: Flags and Session Tunables

Follower Reads



1. Follower replicas always have timeline-consistent data that is slightly stale compared to the data at the leader.

2. Reading from follower replicas directly avoids any consultation with the leader or other follower in different regions.

3. Example workload - product reviews in a retail app.

Performance: Flags and Session Tunables

Tip: Use yb_read_from_followers

- Allows rows to be read from the local fault domain even if it is not the leader
- Must be used in conjunction with read_only transactions
  - Hint, session level, BEGIN, application method annotation (Hibernate/Spring)
  - If not in read_only transaction, reads will always go to the leader
- Staleness of data can be controlled by yb_follower_read_staleness_ms
- Enabled at the session level
  - SET yb_enable_expression_pushdown to on;
- Can also be enabled universe-wide by using Gflag *ysql_pg_conf_csv*

Performance: Flags and Session Tunables

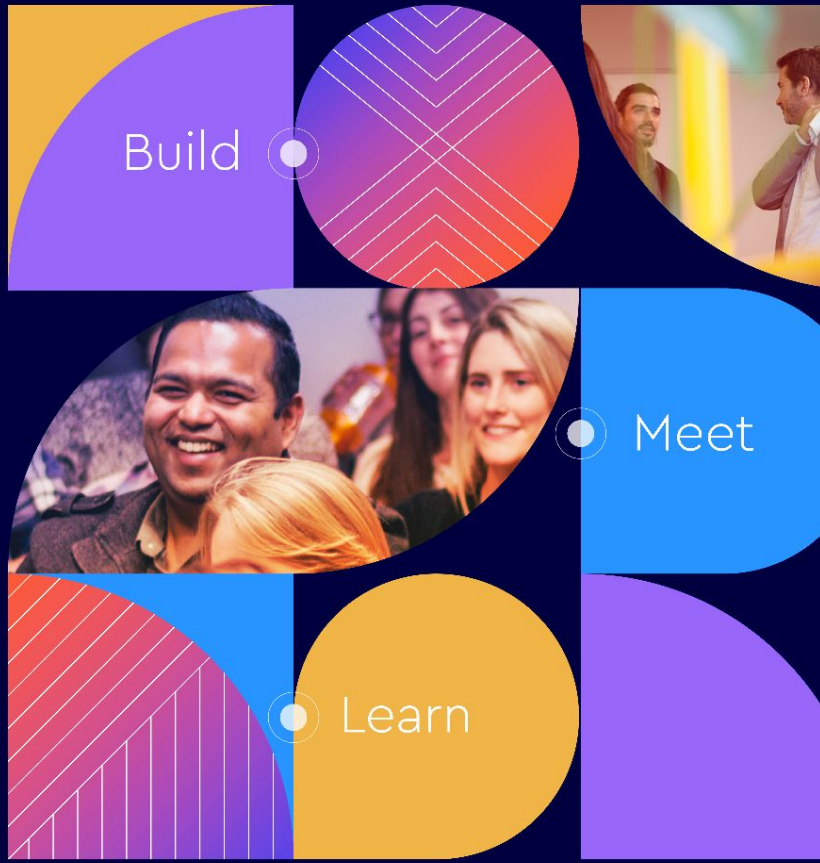
Tip: Enable GFlag ysql_enable_packed_row = true

- Available in 2.15+
- Traditionally, rows have been stored as (k1,c1), (k1,c2),...,(k1,cn) with the associated hybrid times
- New format is (k1, c1, c2, … cn)
- Reduces space amplification as well as reduces latency on read/write operations
- Individual fields are still updated, and if all non-key columns are updated in an UPDATE statement, it is written as a packed-row.
- Compaction re-merges UPDATEs done to non-key columns

Summary

- Performance tuning is an iterative process and an art
- Requires investment of time and willingness to think in terms of the DistributedSQL model
- Future enhancements may obviate some of these techniques
- These are the most common tricks that have yielded high returns, so this is not an exhaustive list

Closing Announcement



Thank You

Join us on Slack: yugabyte.com/slack

Star us on Github:
github.com/yugabyte/yugabyte-db