Workshop: Deploy a Real-time Polling App with Hasura Cloud and Yugabyte Cloud

Eric Pratt, YugabyteDB

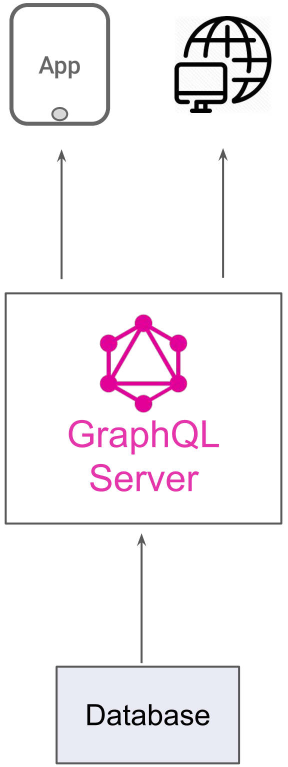
Nikhil Chandrappa, YugabyteDB

Agenda

- Overview of GraphQL Applications using Hasura and YugabyteDB
- Hands on session for Implementing a real-time poll application
    a. Setup the Hasura + YugabyteDB cloud connectivity
    b. Running migrations to setup schema for real-time poll application
    c. Updating node.js application to use Hasura Cloud Instance
    d. Running the real-poll application
- Tuning and performance of GraphQL Queries
- Do's and Don't for GraphQL queries with YugabyteDB

Getting started with GraphQL

- Provides GraphQL abstractions over your Database
    a. Query, mutate data via GraphQL constructs
    b. Build your schema and evolve your domain models
- Evolve your API without versioning
- Provides out of the box pagination and filtering
- Collate disparate Data sources
- Eventing Support

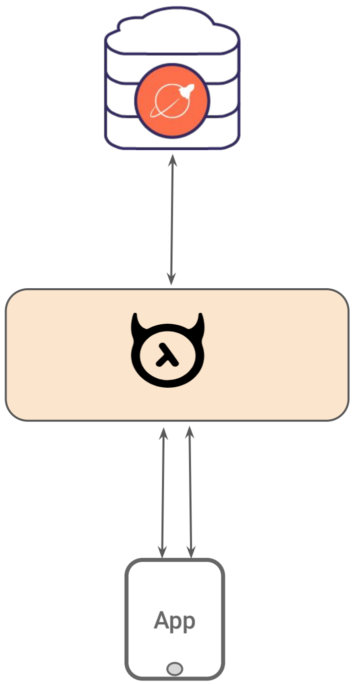


GraphQL Queries for Retrieving Data

<u>Http request</u>

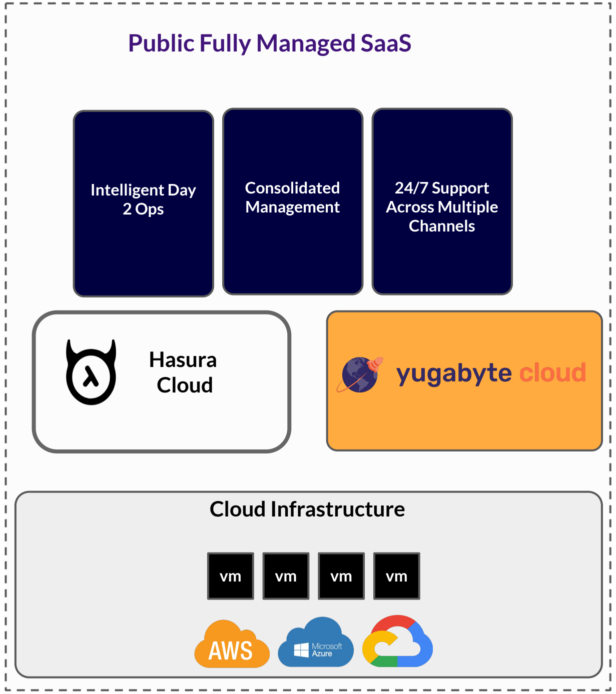
POST /graphql

```
query {
    author {
    name
    articles {
        title
        content
    }
  }
```



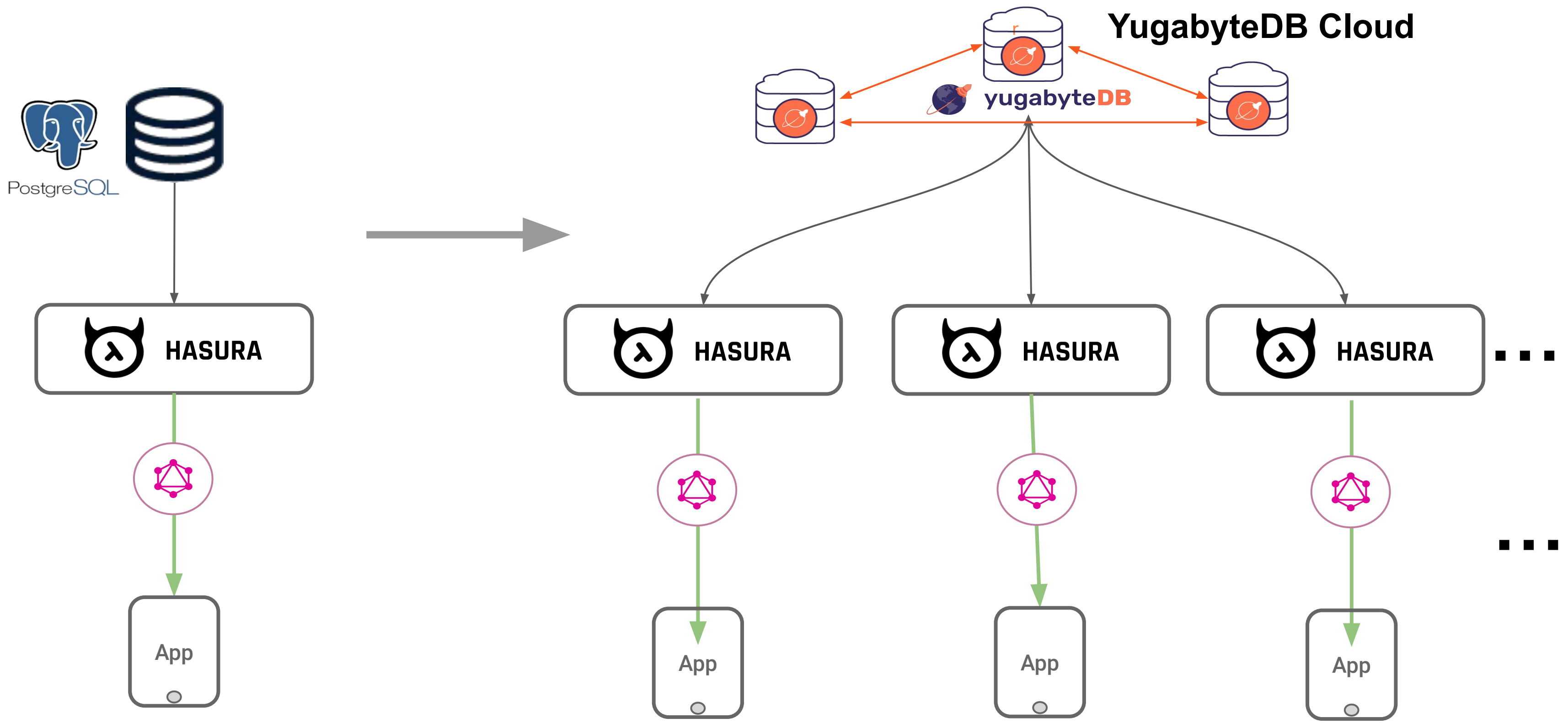
```
{
  "data": {
    "author": [
      {
        "name": "Prof. John",
        "articles": [
          {
            "title": "John's ..",
            "content": "incididunt
ut labore"
          }
        ]
    }}]}
```

Developers embracing cloud



1. Fully managed GraphQL Stack
2. Automated infrastructure operations
3. Removes the need to “install” and “manage” GraphQL engines and backend databases
4. Linear scalability of the backend database at the click of a button

Scaling out GraphQL services with Distributed SQL



Hands-on session

Configuring Hasura Cloud with YugabyteDB Cloud Instance

Hands-on session

Building Real Time poll App using Hasura Cloud and Yugabyte Cloud

Query Tuning - Where to Start?

Performance tuning vs Query debugging

Performance tuning:

- More often involves OS level debugging
- Looking at statistics for memory, cpu, io
- How do queries effect the statistics above?
- Do we have enough horsepower on the DB side to power our applications?
- Identify hotspots or hot nodes

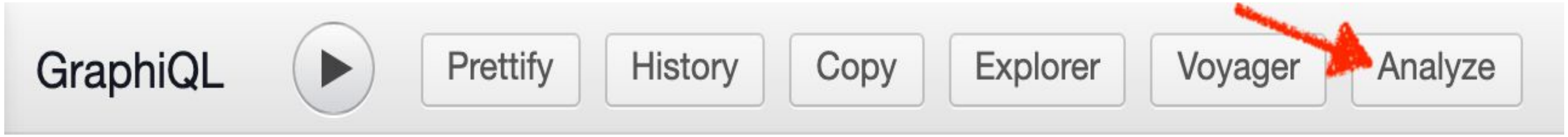
Query debugging and tuning:

- How to identify slow queries in the Database itself
- How to analyze slow queries on Yugabyte platform
- How to analyze slow queries in Hasura
- What to do once we've found a slow query
- Explain vs explain analyze and what it means

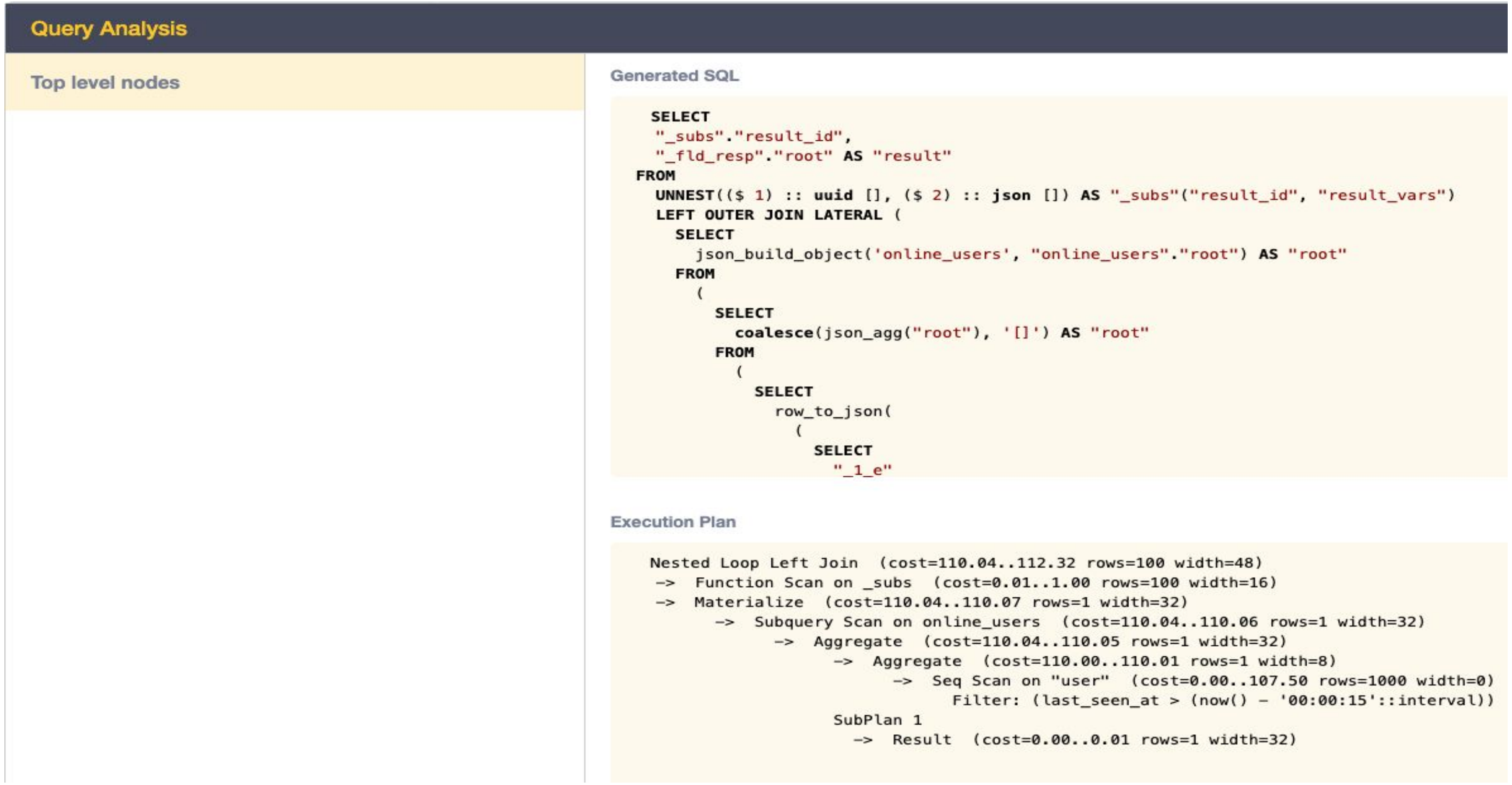
How To Analyze Queries With Hasura

Analyzing queries in Hasura

Hasura has a built-in function that allows you to run the equivalent to an explain analyze within Postgres. You can see it from the Hasura console.



Analyze output with Hasura



Generated SQL

```
  SELECT
   "_subs"."result_id",
   "_fld_resp"."root" AS "result"
 FROM
   UNNEST(($ 1) :: uuid [], ($ 2) :: json []) AS "_subs"("result_id", "result_vars")
   LEFT OUTER JOIN LATERAL (
     SELECT
       json_build_object('online_users', "online_users"."root") AS "root"
     FROM
       (
         SELECT
           coalesce(json_agg("root"), '[]') AS "root"
         FROM
           (
             SELECT
               row_to_json(
                 (
                   SELECT
                     "_1_e"
```

Execution Plan

```
Nested Loop Left Join  (cost=110.04..112.32 rows=100 width=48)
  ->  Function Scan on _subs  (cost=0.01..1.00 rows=100 width=16)
  ->  Materialize  (cost=110.04..110.07 rows=1 width=32)
        ->  Subquery Scan on online_users  (cost=110.04..110.06 rows=1 width=32)
              ->  Aggregate  (cost=110.04..110.05 rows=1 width=32)
                    ->  Aggregate  (cost=110.00..110.01 rows=1 width=8)
                          ->  Seq Scan on "user"  (cost=0.00..107.50 rows=1000 width=0)
                                Filter: (last_seen_at > (now() - '00:00:15'::interval))
                    SubPlan 1
                      ->  Result  (cost=0.00..0.01 rows=1 width=32)
```

How to Identify Slow Queries in YugabyteDB

Finding slow queries in YB

As YugabyteDB is Postgres compatible we can use features built-in to Postgres to dig into problem queries. One of the best options is pg_stat_statements.

```
yugabyte=# \d pg_stat_statements
                View "pg_catalog.pg_stat_statements"
       Column        |       Type       | Collation | Nullable | Default
---------------------+------------------+-----------+----------+---------
 userid              | oid              |           |          |
 dbid                | oid              |           |          |
 queryid             | bigint           |           |          |
 query               | text             |           |          |
 calls               | bigint           |           |          |
 total_time          | double precision |           |          |
 min_time            | double precision |           |          |
 max_time            | double precision |           |          |
 mean_time           | double precision |           |          |
 stddev_time         | double precision |           |          |
 rows                | bigint           |           |          |
```

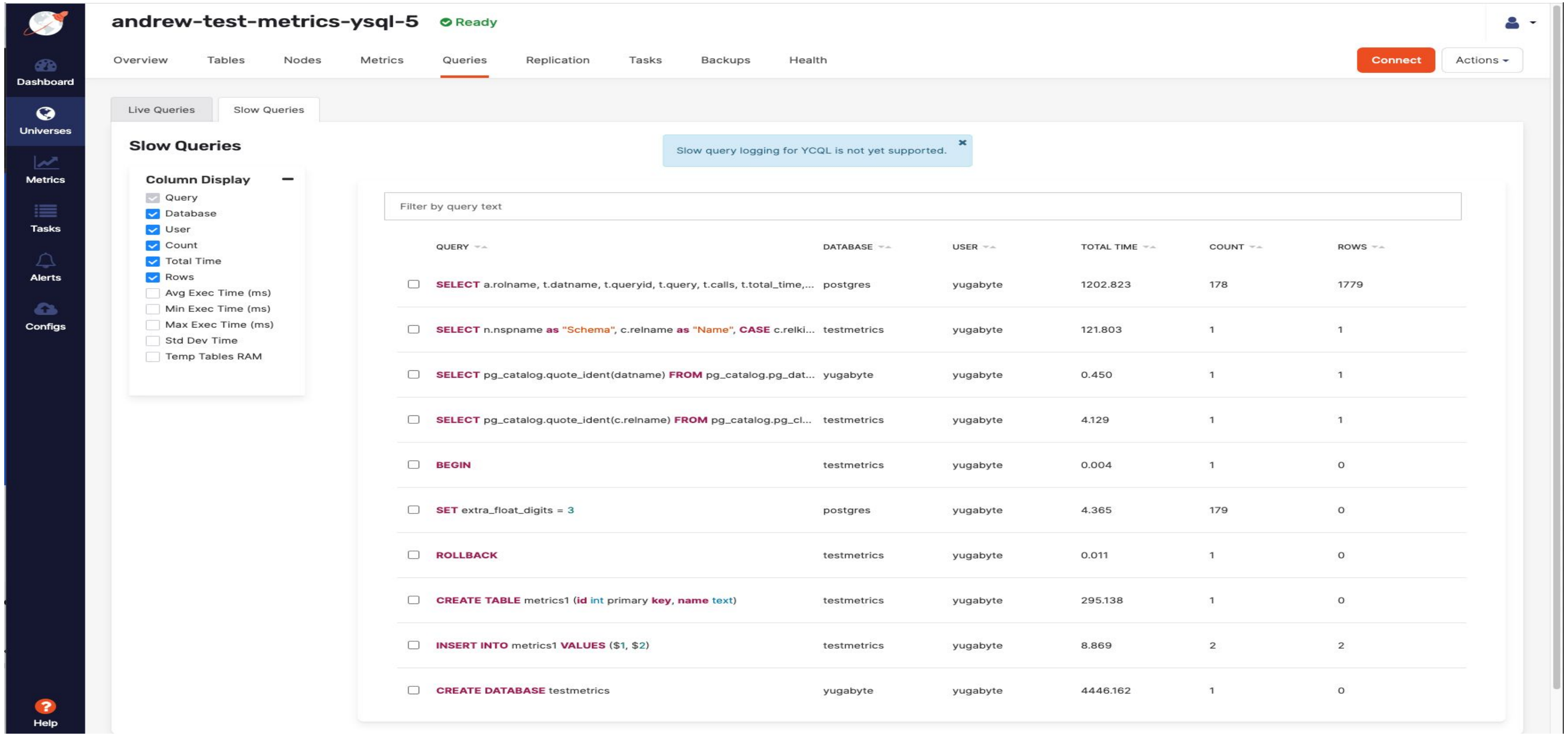
Queries to run against pg_stat_statement

I find the following queries to be the most helpful:

- SELECT pg_stat_statements_reset();
  - This allows us to clear out pg_stat_statements

- select dbid,query,calls,total_time,min_time,max_time,mean_time,rows from pg_stat_statements order by total_time desc limit 10;

Identifying a slow query in YugabyteDB Platform



andrew-test-metrics-ysql-5 Ready

Slow Queries

Slow query logging for YCQL is not yet supported.

Column Display
- [✓] Query
- [✓] Database
- [✓] User
- [✓] Count
- [✓] Total Time
- [✓] Rows
- [ ] Avg Exec Time (ms)
- [ ] Min Exec Time (ms)
- [ ] Max Exec Time (ms)
- [ ] Std Dev Time
- [ ] Temp Tables RAM

Filter by query text

QUERY	DATABASE	USER	TOTAL TIME	COUNT	ROWS
SELECT a.rolname, t.datname, t.queryid, t.query, t.calls, t.total_time,...	postgres	yugabyte	1202.823	178	1779
SELECT n.nspname as "Schema", c.relname as "Name", CASE c.relki...	testmetrics	yugabyte	121.803	1	1
SELECT pg_catalog.quote_ident(datname) FROM pg_catalog.pg_dat...	yugabyte	yugabyte	0.450	1	1
SELECT pg_catalog.quote_ident(c.relname) FROM pg_catalog.pg_cl...	testmetrics	yugabyte	4.129	1	1
BEGIN	testmetrics	yugabyte	0.004	1	0
SET extra_float_digits = 3	postgres	yugabyte	4.365	179	0
ROLLBACK	testmetrics	yugabyte	0.011	1	0
CREATE TABLE metrics1 (id int primary key, name text)	testmetrics	yugabyte	295.138	1	0
INSERT INTO metrics1 VALUES ($1, $2)	testmetrics	yugabyte	8.869	2	2
CREATE DATABASE testmetrics	yugabyte	yugabyte	4446.162	1	0

You’ve Identified a Slow Query, Now What?

Running an explain plan

Let’s have a look at an explain plan:

```
EXPLAIN SELECT * FROM foo;

                      QUERY PLAN
---------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

Here is a basic example taken from the postgres docs on a sample table with a single integer column and 1000 rows.

We can see that this query will do a sequential scan on table foo for all rows.

Understanding explain plan cont.

Let’s take a look at the following explain analyze:

```
$ explain analyze select * from t limit 100;
                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.00..9.33 rows=100 width=608) (actual time=0.008..0.152 rows=100 loops=1)
   ->  Seq Scan on t  (cost=0.00..93333.86 rows=999986 width=608) (actual time=0.007..0.133
rows=100 loops=1)
 Total runtime: 0.181 ms
(3 rows)
```

Similar to explain, explain analyze will show you where exactly the time is spent. In this output we can see that the actual time is recorded in the output.

What about something more complex?

Let’s consider the following tables:

```
create table breweries (
brew_id int,
name varchar(100),
city varchar(100),
state varchar(3),
primary key (brew_id)
);
```

```
create table beers (
name varchar(100),
beer_id int,
abv real,
ibu int,
brewery_id int,
style varchar(100),
ounces real,
primary key (beer_id),
constraint fk_beer_breweries
  foreign key (brewery_id)
  references breweries(brew_id)
);
```

Let’s query those tables and see what it looks like

```
explain
select be.name,
       br.name
from   beers be, breweries br
where  be.brewery_id = br.brew_id;
```

```
                                         QUERY PLAN
----------------------------------------------------------------------
 Nested Loop  (cost=0.00..213.89 rows=1000 width=436)
   ->  Seq Scan on beers be  (cost=0.00..100.00 rows=1000 width=222)
   ->  Index Scan using breweries_pkey on breweries br  (cost=0.00..0.11 rows=1 width=222)
         Index Cond: (brew_id = be.brewery_id)
```

Thank You

Join us on Slack: yugabyte.com/slack

Star us on Github:
https://github.com/yugabyte/spring-data-yugabytedb