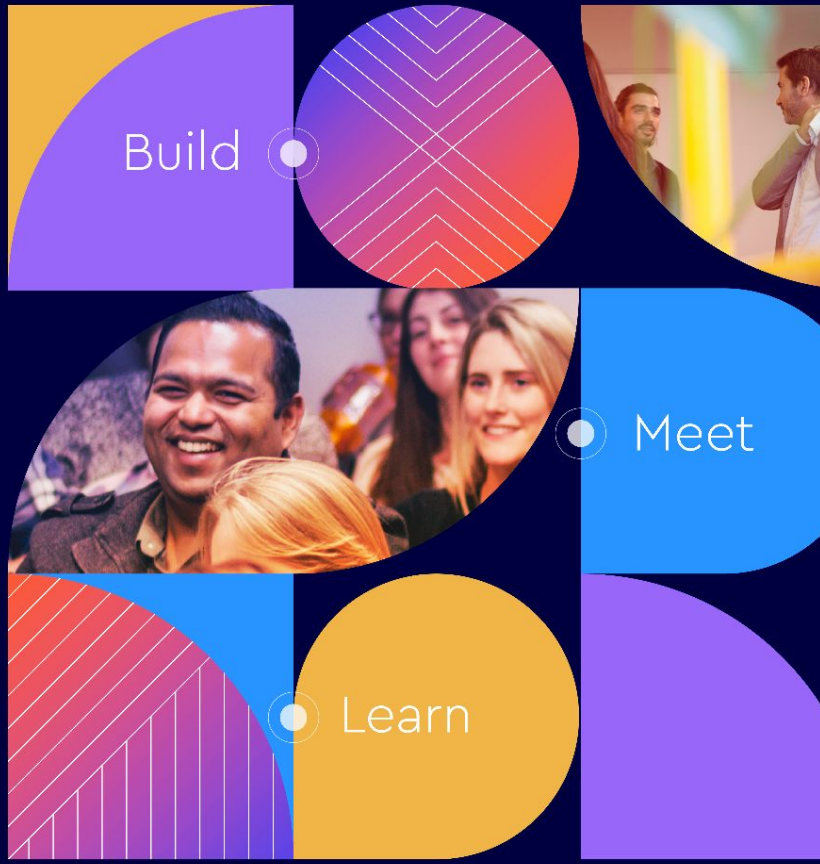


Distributed SQL Tuning

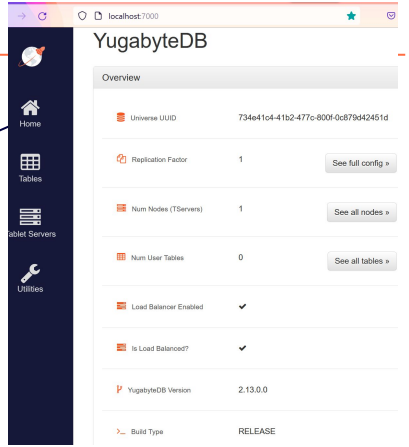
Workshop

Franck Pachot

Developer Advocate

https://www.linkedin.com/in/franckpachot
https://twitter.com/franckpachot
https://dev.to/franckpachot

Quick Start (single-node lab)




```
# Start YugabyteDB (docker)

docker run -d --name yb --rm -p5433:5433 -p7000:7000 \
  yugabytedb/yugabyte:2.14.1.0-b36 \
  yugabyted start --daemon=false

# Connect in container

docker exec -it yb bash -c 'ysqlsh -h $(hostname)'

# or from laptop

psql postgres://localhost:5433

start http://localhost:7000
```

Agenda

Distributed SQL Tuning

- Tables and Indexes (sharding)
- Create Table (Primary Key)
- Instrumentation
- Execution Plans
- Push Down (offload to storage)
- Secondary indexes
- Joins (order and method)
- Hints (pg_hint_plan)
- Geo Distribution
- Follower Reads
- Bulk Load
- SQL Tuning methodology

Distributed SQL Tuning

Tables and Indexes (sharding)

What's different in Distributed SQL?

RPCs

Tables are distributed

reading (and writing) is not a local memory operation

LSM-Tree

No heap tables, no B-Tree

table rows are stored in the primary key index structure

Sharded, Distributed, Replicated

Distributed with sharding (hash or range) into tablets

Replicated into tablet peers

Stored in LSM-Tree logically ordered on:

- the table primary key, or the indexed columns

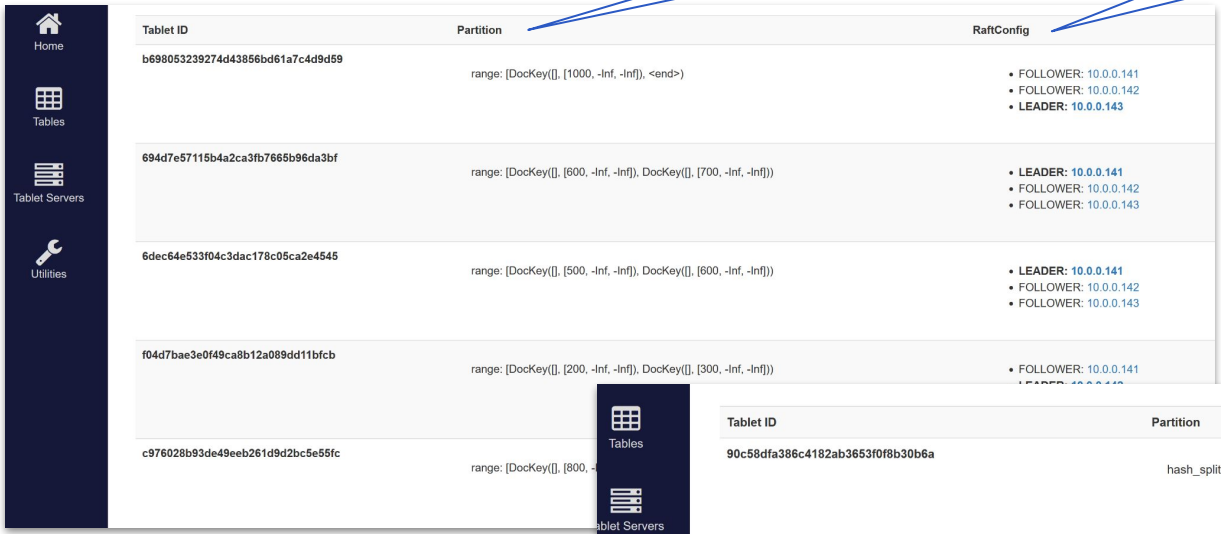
The index includes:

- all columns for the table for the primary key
- some columns for the secondary index (+ the PK value)

Distributed and Replicated

Distributed to tablets

Replicated to tablet peers

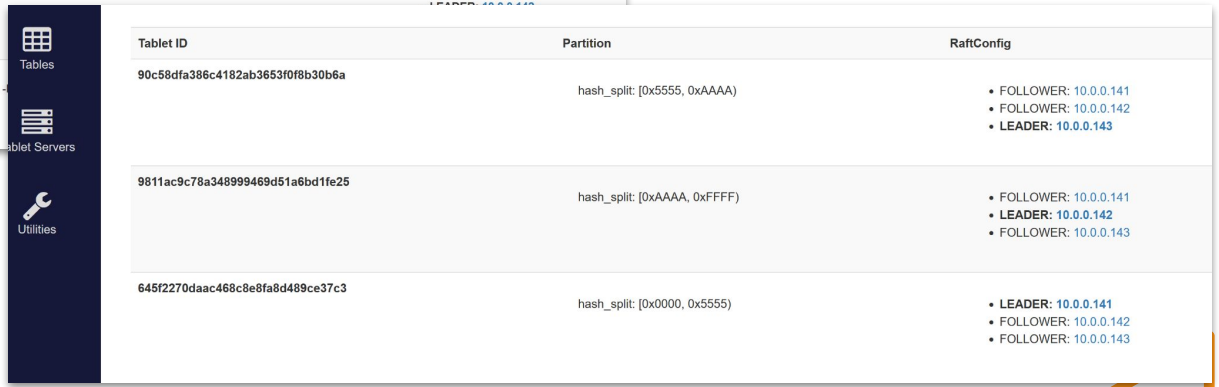


Range Sharding

```
create index ... on ...
( time ASC, ...)
split at values (
(100),(200),(300),(400),
(500),(600),(700),(800),
(900),(1000)
);
```

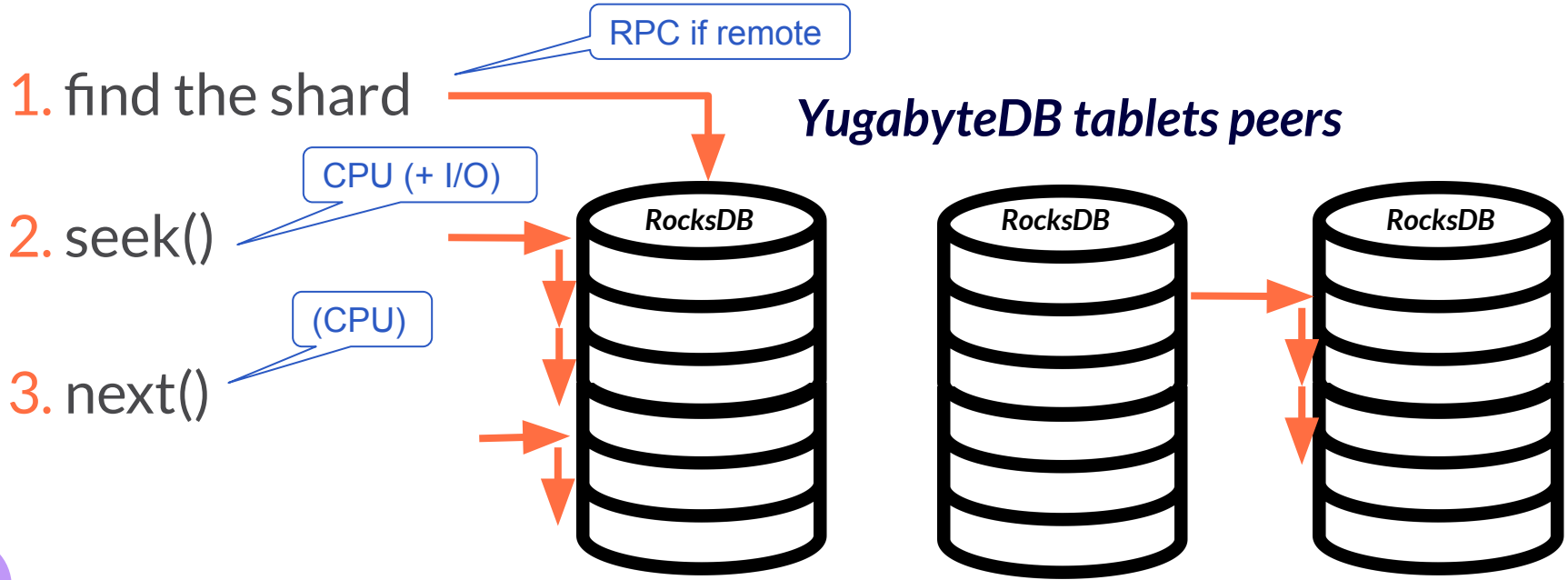
Hash Sharding

```
create table
( id HASH primary key,
...)
split into 3 tablets;
```



Tables and Indexes access

Point or Range query (WHERE clause, index value)



Distributed SQL Tuning

CREATE TABLE

The importance of Primary Key

The primary key columns (datatype, order, sharding) is

- not easy to change (because physically organized on PK)
- critical for the main access pattern (WHERE and JOIN)
- fundamental for the distribution of data / processing

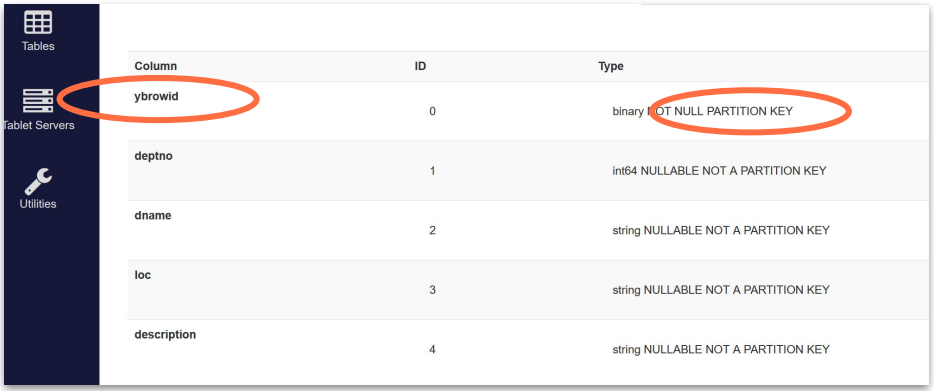
💡 Define the PRIMARY KEY with the CREATE TABLE

Without a declared PK

YugabyteDB adds one but not very useful as it is hidden

```
CREATE TABLE dept (
  deptno bigint NOT NULL,
  dname text,
  loc text,
  description text
);

\d dept
```

```
yugabyte=# \d dept
                Table "public.dept"
   Column    |  Type  | Collation | Nullable | Default
-------------+--------+-----------+----------+---------
 deptno      | bigint |           | not null |
 dname       | text   |           |          |
 loc         | text   |           |          |
 description | text   |           |          |

yugabyte=#
```



ALTER TABLE ... ADD PRIMARY KEY

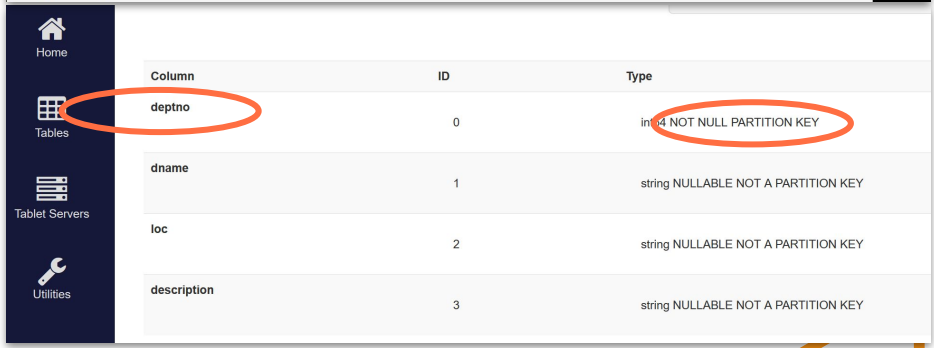
Supported for compatibility but not efficient

```
insert into dept(deptno)
 select
 generate_series(1,100000);
```

```
alter table dept
 add primary key(deptno);
```

```
\d dept
```

```
yugabyte=# \d dept
                Table "public.dept"
   Column    |  Type  | Collation | Nullable | Default
-------------+--------+-----------+----------+---------
 deptno      | bigint |           | not null |
 dname       | text   |           |          |
 loc         | text   |           |          |
 description | text   |           |          |
Indexes:
    "dept_pkey" PRIMARY KEY, lsm (deptno HASH)

yugabyte=#
```



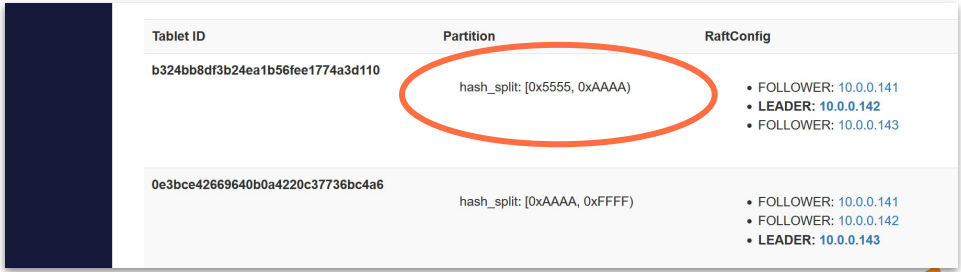
You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE

Default sharding: HASH on first column

```
DROP TABLE IF EXISTS dept;

CREATE TABLE dept (
  deptno bigint NOT NULL,
  dname text,
  loc text,
  description text,
  CONSTRAINT pk_dept
   PRIMARY KEY (deptno)
);

\d dept
```

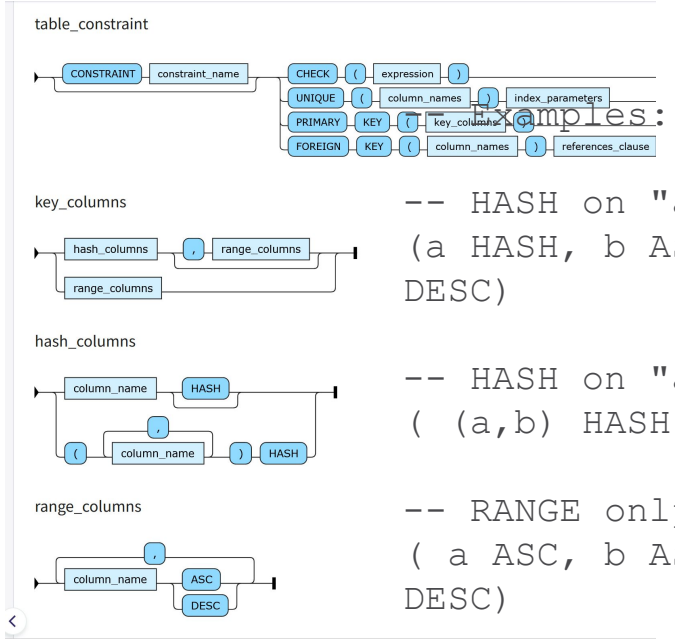
```
yugabyte=# \d dept
                Table "public.dept"
   Column    |  Type  | Collation | Nullable | Default
-------------+--------+-----------+----------+---------
 deptno      | bigint |           | not null |
 dname       | text   |           |          |
 loc         | text   |           |          |
 description | text   |           |          |
Indexes:
    "pk_dept" PRIMARY KEY, lsm (deptno HASH)
```



Better be explicit on sharding key

https://docs.yugabyte.com/preview/api/ysql/the-sql-language/statements/ddl_create_table

```
DROP TABLE IF EXISTS dept;

CREATE TABLE dept (
  deptno bigint NOT NULL,
  dname text,
  loc text,
  description text,
  CONSTRAINT pk_dept
   PRIMARY KEY ( (deptno) HASH)
);

\d dept
```



Index Access with HASH sharding

```
explain analyze select * from dept where deptno=10;
                  QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using pk_dept on dept (cost=0.00..4.11)
   Index Cond: (deptno = 10)
```

```
explain analyze select * from dept where deptno in (10,20,30,40);
                  QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using pk_dept on dept (cost=0.00..4.12)
   Index Cond: (deptno = ANY ('{10,20,30,40}'::bigint[]))
```

```
explain analyze select * from dept where deptno between 10 and 40;
                  QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on dept  (cost=0.00..105.00)
   Filter: ((deptno >= 10) AND (deptno <= 40))
```

You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)

Distributed SQL Tuning

Instrumentation

Slow queries

PostgreSQL shows

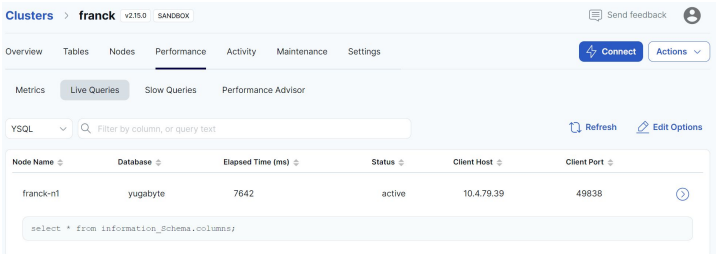
- current queries is pg_stat_activity
- all queries in pg_stat_statements

```
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           |           |          |
```

Those are per-node in YugabyteDB

They are consolidated in the GUI:

- YugabyteDB Managed (cloud DBaaS)
- YugabyteDB Anywhere (on-prem DBaaS)



Execution Plans and Tablet Server work

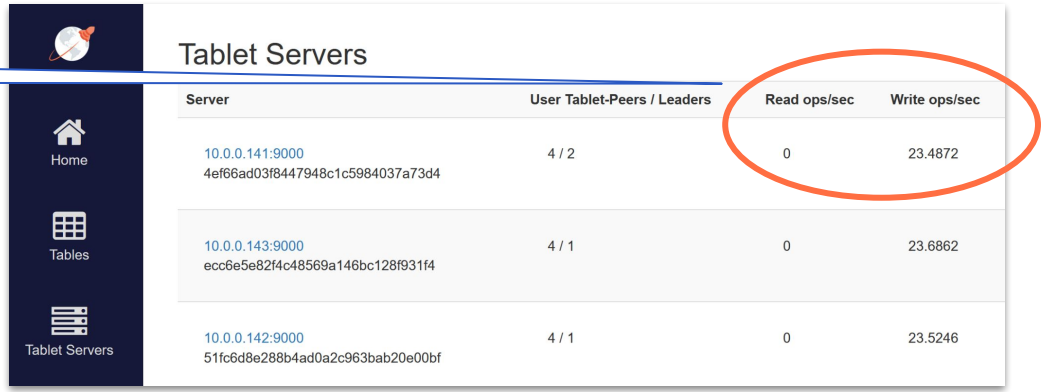
YugabyteDB has two layers:

- YSQL parses, optimizes and executes the execution plan
- it reads/writes to the tablet servers (DocDB)

```
\timing on
insert into dept(deptno)select generate_series(1,1000000);
```

response time from the client)

DocDB operations



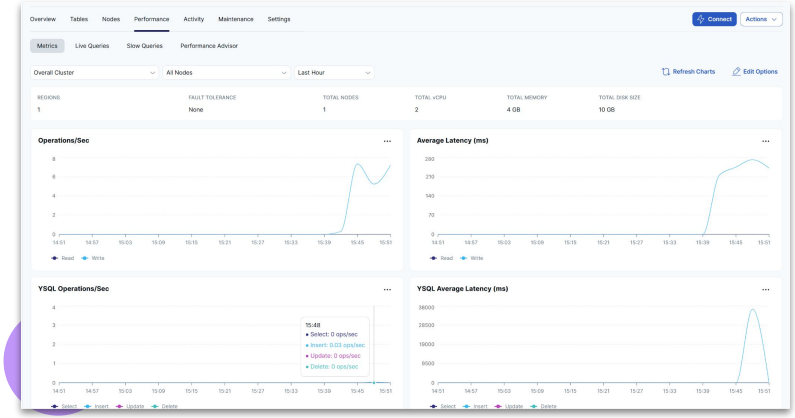
Example:

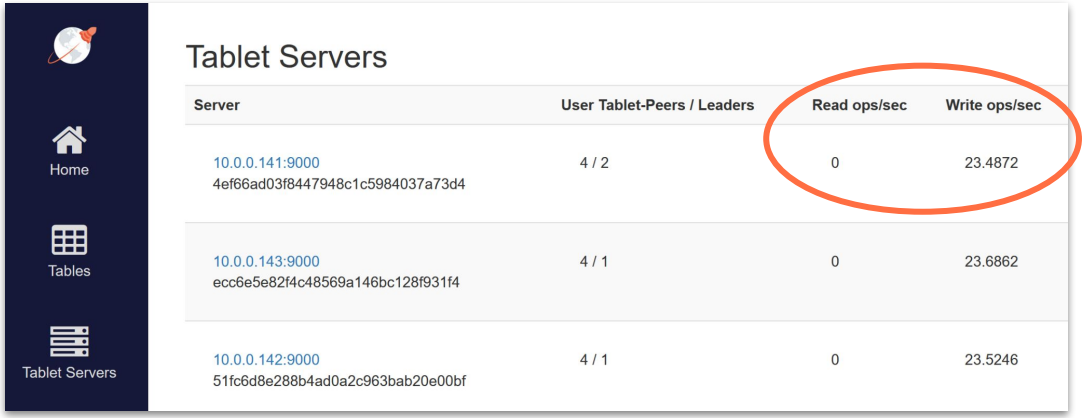
1 million rows in 14s
70k rows per seconds
but batched into 22 write ops/s

Read / Write ops/s

Used to get an idea of RPCs (YSQL -> DocDB)

- gives an idea how data is distributed (hotspots)
- doesn't account for replication (only the leader)
- doesn't account for intra DocDB operations





More info with tablet statistics

EXPLAIN ANALYZE will expose more metrics
in future YugabyteDB releases
The statistics endpoint can show RocksDB seek() and next()

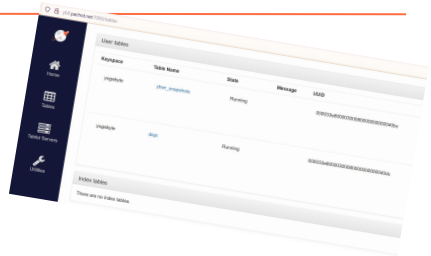
```
yugabyte=# insert into dept(deptno)select generate_series(1,1000);
INSERT 0 1000
yugabyte=# execute snap_table;
 rocksdb_seek | rocksdb_next | rocksdb_insert | dbname / relname / tserver / tabletid / leader
--------------+--------------+----------------+-----------------------------------------------
         1625 |              |            325 | yugabyte dept 10.0.0.141 10265287379c... L
              |              |            344 | yugabyte dept 10.0.0.141 6eb4335b17c0...
              |              |            331 | yugabyte dept 10.0.0.141 f93feacc533b...
              |              |            325 | yugabyte dept 10.0.0.142 10265287379c...
              |              |            344 | yugabyte dept 10.0.0.142 6eb4335b17c0...
         1655 |              |            331 | yugabyte dept 10.0.0.142 f93feacc533b... L
              |              |            325 | yugabyte dept 10.0.0.143 10265287379c...
         1720 |              |            344 | yugabyte dept 10.0.0.143 6eb4335b17c0... L
              |              |            331 | yugabyte dept 10.0.0.143 f93feacc533b...
(9 rows)
```

read from leader (for duplicate key detection)

Inserts into leader and followers

My lab (for my demos - you don't need this)

YugabyteDB managed (cloud) doesn't show the internals
YugabyteDB OSS exposes more stats on port 7000



```
docker run -d -p5433:5433 -p7000:7000 \
 yugabytedb/yugabyte:latest yugabyted start --daemon=false
psql
```

My script to show seek() and next():

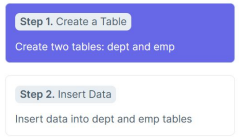
```
\! curl -s
https://raw.githubusercontent.com/FranckPachot/ybdemo/main/
docker/yb-lab/client/ybwr.sql | grep -v '\watch' > ybwr.sql
\i ybwr.sql
execute snap_table;
```

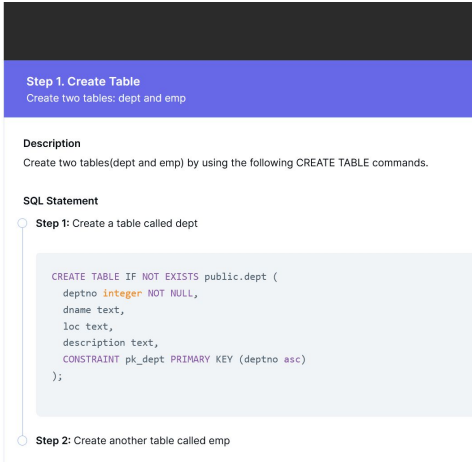
Distributed SQL Tuning

Execution Plans

Let's insert some data before looking at the execution plans

This is the cloud (free tier) tutorial tables






```
drop table if exists dept cascade;

CREATE TABLE IF NOT EXISTS dept (
  deptno integer NOT NULL,
  dname text,
  loc text,
  description text,
  CONSTRAINT pk_dept PRIMARY KEY (deptno asc)
);

CREATE TABLE IF NOT EXISTS emp (
  empno integer generated by default as identity (start with 10000) NOT NULL,
  ename text NOT NULL,
  job text,
  mgr integer,
  hiredate date,
  sal integer,
  comm integer,
  deptno integer NOT NULL,
  email text,
  other_info jsonb,
  CONSTRAINT pk_emp PRIMARY KEY (empno hash),
  CONSTRAINT emp_email_uk UNIQUE (email),
  CONSTRAINT fk_deptno FOREIGN KEY (deptno) REFERENCES dept(deptno),
  CONSTRAINT fk_mgr FOREIGN KEY (mgr) REFERENCES emp(empno),
  CONSTRAINT emp_email_check CHECK ((email ~ '^[a-zA-Z0-9.!#$%&''*+/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-
);

INSERT INTO dept (deptno,  dname,       loc, description)
            VALUES       (10,        'ACCOUNTING', 'NEW YORK','preparation of financial statements, maintenance of general ledger, payment of bills, preparation of customer bills, payroll, and more.'),
            (20,         'RESEARCH',   'DALLAS','responsible for preparing the substance of a research report or security recommendation.'),
            (30,         'SALES',     'CHICAGO','division of a business that is responsible for selling products or services'),
            (40,         'OPERATIONS', 'BOSTON','administration of business practices to create the highest level of efficiency possible within an organization');

INSERT INTO emp (empno, ename,          job,         mgr,   hiredate,          sal, comm, deptno, email, other_info)
            VALUES   (7369, 'SMITH',  'CLERK',      7902, '1980-12-17',  800, NULL,   20,'SMITH@acme.com', '{"skills":["accounting"]}'),
            (7499, 'ALLEN',  'SALESMAN',  7698, '1981-02-20', 1600,  300,   30,'ALLEN@acme.com', null),
            (7521, 'WARD',   'SALESMAN',  7698, '1981-02-22', 1250,  500,   30,'WARD@compuserve.com', null),
            (7566, 'JONES',  'MANAGER',   7839, '1981-04-02', 2975, NULL,   20,'JONES@gmail.com', null),
            (7654, 'MARTIN', 'SALESMAN',  7698, '1981-09-28', 1250, 1400,   30,'MARTIN@acme.com', null),
            (7698, 'BLAKE',  'MANAGER',   7839, '1981-05-01', 2850, NULL,   30,'BLAKE@hotmail.com', null),
            (7782, 'CLARK',  'MANAGER',   7839, '1981-06-09', 2450, NULL,   10,'CLARK@acme.com', '{"skills":["C","C++","SQL"]}'),
            (7788, 'SCOTT',  'ANALYST',   7566, '1982-12-09', 3000, NULL,   20,'SCOTT@acme.com', '{"cat":"tiger"}'),
            (7839, 'KING',   'PRESIDENT', NULL, '1981-11-17', 5000, NULL,   10,'KING@aol.com', null),
            (7844, 'TURNER', 'SALESMAN',  7698, '1981-09-08', 1500,    0,   30,'TURNER@acme.com', null),
            (7876, 'ADAMS',  'CLERK',  7788, '1983-01-12', 1100, NULL,   20,'ADAMS@acme.org', null),
            (7900, 'JAMES',  'CLERK',  7698, '1981-12-03',  950, NULL,   30,'JAMES@acme.org', null),
            (7902, 'FORD',   'ANALYST',   7566, '1981-12-03', 3000, NULL,   20,'FORD@acme.com', '{"skills":["SQL","CQL"]}'),
            (7934, 'MILLER', 'CLERK',  7782, '1982-01-23', 1300, NULL,   10,'MILLER@acme.com', null);
```

EXPLAIN is the most important tuning tool for SQL

EXPLAIN

shows the execution plan with estimations (no execution)

EXPLAIN analyze

executes the statements and show some execution statistics

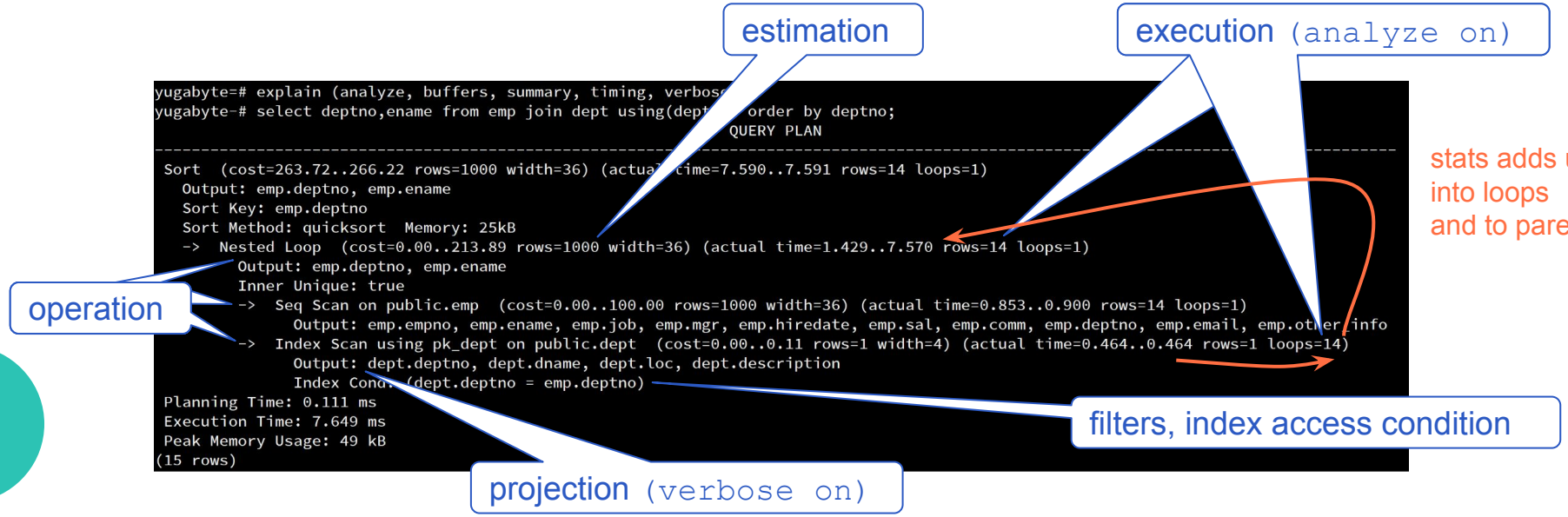
EXPLAIN (costs off, analyze, buffers)

controls what is displayed

https://docs.yugabyte.com/preview/explore/query-1-performance/explain-analyze

EXPLAIN information

```
explain
select * from emp join dept using(deptno);

explain (analyze, buffers, summary, timing, verbose)
select * from emp join dept using(deptno);
```



You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)
- EXPLAIN ANALYZE is your way to understand why a query is slow

Distributed SQL Tuning

Push Down (offload to storage)

Aggregate pushdown

```
select count(*) from emp;
 count
-------
    14
(1 row)
```

```
explain (costs off, analyze)
select count(*) from emp;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Finalize Aggregate (actual time=0.713..0.713 rows=1 loops=1)
   ->  Seq Scan on emp (actual time=0.706..0.708 rows=3 loops=1)
         Partial Aggregate: true
 Planning Time: 0.046 ms
 Execution Time: 0.760 ms
 Peak Memory Usage: 24 kB
(6 rows)
```

partial counts are summed

this is the number of partial count (3 tablets here)

partial count on each tserver

Expression pushdown

```
set yb_enable_expression_pushdown=off;
explain analyze select * from emp where upper(ename) like 'KING';
                                  QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on emp (actual time=1.005..2.132 rows=1 loops=1)
   Filter: (upper(ename) ~~ 'KING'::text)
   Rows Removed by Filter: 13
(3 rows)
```

those rows are sent from DocDB to YSQL to be discarded there

```
set yb_enable_expression_pushdown=on;
explain analyze select * from emp where upper(ename) like 'KING';
                                  QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on emp (actual time=1.261..1.263 rows=1 loops=1)
   Remote Filter: (upper(ename) ~~ 'KING'::text)
(2 rows)
```

filtered on tserver while they are read

You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)
- EXPLAIN ANALYZE is your way to understand why a query is slow
- Watch for `Rows Removed by Filter` with high amount of rows
- Always enable `yb_enable_expression_pushdown`

Distributed SQL Tuning

Secondary indexes

Secondary Indexes

YugabyteDB tables are stored in the Primary Key LSM-Tree

👉fast access for point (and range if range sharding) queries

SQL databases allow queries for multiple use cases

👉global secondary indexes are maintained
(with strong consistency != NoSQL)

YugabyteDB provides lot of options to shape the index on required columns, rows, order, ... (🙏 to PostgreSQL)

Index Scan and Index Only Scan

Index Scan on Primary Key reads only the table

Index Scan on Secondary Index gets the Primary Key value from the index, then reads from the table

✈️ Index Only Scan finds all required columns in the Secondary Index index with no hop to the table

💡 look at `Output` from `explain (verbose)`

Index Only Scan

```
explain (analyze, verbose) select * from emp where email = 'SCOTT@acme.com';
                              QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using emp_email_uk on public.emp (actual time=1.736..1.739 rows=1 loops=1)
   Output: empno, ename, job, mgr, hiredate, sal, comm, deptno, email, other_info
   Index Cond: (emp.email = 'SCOTT@acme.com'::text)
```

```
yugabyte=# execute snap_table;
 rocksdb_seek | rocksdb_next | rocksdb_insert |   dbname / relname / tserver / tabletid / leader
--------------+--------------+----------------+-----------------------------------------------
            1 |            1 |                | yugabyte emp 10.0.0.143 c6474899ef31... L
            1 |            1 |                | yugabyte emp_email_uk 10.0.0.143 7eb5f77ad7e2... L
```

```
explain (analyze, verbose) select email from emp where email = 'SCOTT@acme.com';
                              QUERY PLAN
---------------------------------------------------------------------------
 Index Only Scan using emp_email_uk on public.emp(actual time=1.064..1.066 rows=1 loops=1)
   Output: email
   Index Cond: (emp.email = 'SCOTT@acme.com'::text)
```

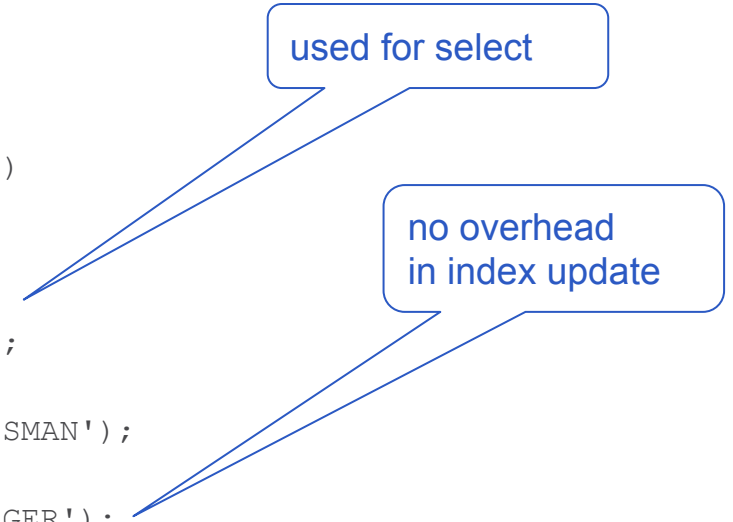
```
yugabyte=# execute snap_table;
 rocksdb_seek | rocksdb_next | rocksdb_insert |   dbname / relname / tserver / tabletid / leader
--------------+--------------+----------------+-----------------------------------------------
            1 |            1 |                | yugabyte emp_email_uk 10.0.0.143 7eb5f77ad7e2... L
```

Partial Indexes

You can add a WHERE clause to an index

- saves storage space
- saves DML overhead

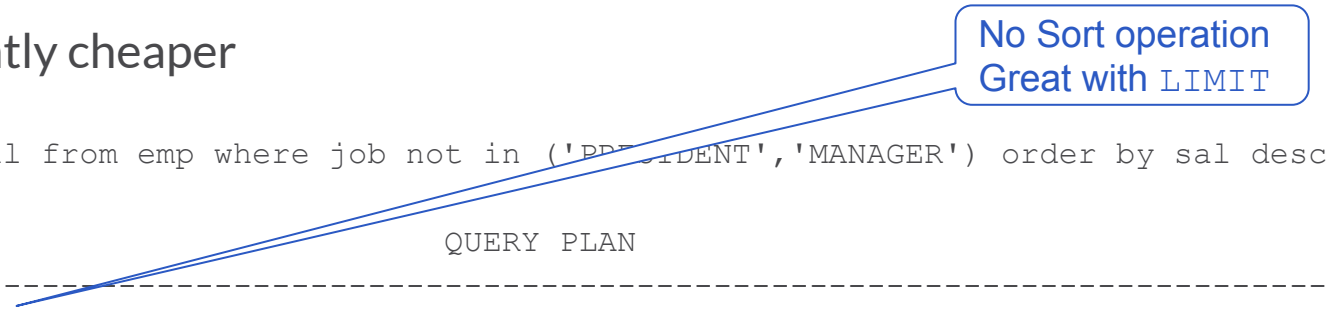
```
create index emp_sal_nonmgr on emp (sal desc)
 where job not in ('PRESIDENT','MANAGER');

explain (analyze, verbose) select ename,sal
from emp  where job = 'ANALYST' and sal>2000;

update emp set sal=sal+0 where job in ('SALESMAN');

update emp set sal=sal+0 where job in ('MANAGER');
```



Index Order

Range indexes can be read Forward or Backward

Forward is slightly cheaper




```
explain select sal from emp where job not in ('PRESIDENT','MANAGER') order by sal desc;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using emp_sal_salesman on emp (actual time=1.015..1.022 rows=10 loops=1)

explain select sal from emp where job not in ('PRESIDENT','MANAGER') order by sal asc;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan Backward using emp_sal_salesman on emp (actual time=1.322..1.329 rows=10 ...
```

Covering Index

You add output columns to get an Index Only Scan

with the INCLUDE keyword if they are not part of the key

```
create unique index emp_uk_email on emp (email asc) include (ename, hiredate);

explain select email, ename, hiredate from emp where email like 'BLAKE@hotmail.%';

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Only Scan using emp_uk_email on public.emp  (cost=0.00..4.    rows=1 width=68)
   Output: email, ename, hiredate
   Index Cond: ((emp.email >= 'BLAKE@hotmail.'::text)
                AND (emp.email < 'BLAKE@hotmail/'::text))
   Filter: (emp.email ~~ 'BLAKE@hotmail.%'::text)
```

Not sorted, not part of the key

Shape your indexes

In short:

- Primary Key is the main access
- Secondary indexes are shaped for other use-cases
  - key columns (Hash/Range) to avoid large scans
  - range order to avoid sorts (ASC / DESC)
  - more columns to avoid table access (INCLUDE)
- Indexes slow the inserts / delete / update indexed columns

You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)
- EXPLAIN ANALYZE is your way to understand why a query is slow
- Watch for `Rows Removed by Filter` with high amount of rows
- Always enable `yb_enable_expression_pushdown`
- Covering indexes are important in Distributed SQL (to avoid RPCs)
- Don't `SELECT *` if you don't need all the columns
- Use Partial Index is you don't need the index for all the rows

Distributed SQL Tuning

Joins

Join methods (chosen by the query planner or via pg_hint_plan)

Nested Loop

- good for few rows from outer table (<=10)
- and fast access to inner table (index)

Hash Join

- good for many rows joined to small table (lookup)

Sort Merge join

- when rows are ordered, avoids the hashing

The join order is the most important (Outer ⋈ Inner)

The optimal join order

Starts with a table that fetches quickly a small result

(driving small table or selective index, ideally sorted for the `order by`)

Joins to tables that eliminate more rows for cheap

(predicates on lookup tables, Hash Join to small or selective Index)

Joins to tables which adds more columns (master)

(lookup tables without predicates, Nested Loop if few outer rows, Hash Join if not)

Joins to tables which add rows (detail)

(detail table, Nested Loop with Index Only Scan)

Join order with some guidance

SQL is declarative, write it as you want

the query planner determines the order... but

You can guide the join order with `Leading()` hint

You may want to follow your `FROM` clause order

```
set local join_collapse_limit 1;
```

local: the scope is the transaction

You should remember

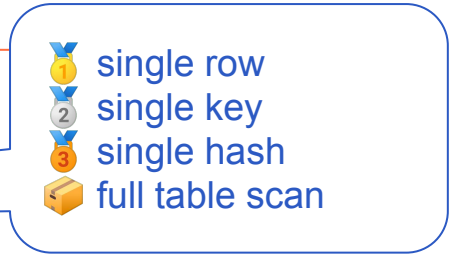
- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)
- EXPLAIN ANALYZE is your way to understand why a query is slow
- Watch for `Rows Removed by Filter` with high amount of rows
- Always enable `yb_enable_expression_pushdown`
- Covering indexes are important in Distributed SQL (to avoid RPCs)
- Don't `SELECT *` if you don't need all the columns
- Use Partial Index is you don't need the index for all the rows
- Check the join order in the execution plan, provide the right indexes to filter early

Distributed SQL Tuning

Hints

Cost Based Optimizer

In current version (2.15) YugabyteDB uses a few constant for cost estimation, similar to rule-based.



Work in progress for future versions: ANALYZE and selectivity estimation (`yb_enable_optimizer_statistics`)

💡 pg_hint_plan (installed by default) for complex queries

Join order and method - example 1

```
/*+ leading( ( (emp mgr) dept ) ) HashJoin(emp mgr dept) NestLoop(emp mgr) */
explain (costs off) select * from dept join emp using(deptno)
left join emp mgr on emp.mgr=mgr.empno;

                    QUERY PLAN
-------------------------------------------------
 Hash Join
   Hash Cond: (emp.deptno = dept.deptno)
   ->  Nested Loop Left Join
         ->  Seq Scan on emp
         ->  Index Scan using pk_emp on emp mgr
               Index Cond: (emp.mgr = empno)
   ->  Hash
         ->  Seq Scan on dept
```

HashJoin (... dept)

NestLoop (emp mgr)

Leading (emp mgr)

Leading ((...) dept)

built table

Join order and method - example 2

```
/*+ leading( ( dept (emp mgr) ) ) HashJoin(emp mgr dept) NestLoop(emp mgr) */
explain (costs off) select * from dept join emp using(deptno)
left join emp mgr on emp.mgr=mgr.empno;

                    QUERY PLAN
-----------------------------------------------
 Hash Join
   Hash Cond: (dept.deptno = emp.deptno)
   ->  Seq Scan on dept
   ->  Hash
         ->  Nested Loop Left Join
               ->  Seq Scan on emp
               ->  Index Scan using pk_emp on emp mgr
                     Index Cond: (emp.mgr = empno)
```

HashJoin (... dept)

Leading (dept (...))

NestLoop (emp mgr)

Leading (emp mgr)

built table

You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)
- EXPLAIN ANALYZE is your way to understand why a query is slow
- Watch for `Rows Removed by Filter` with high amount of rows
- Always enable `yb_enable_expression_pushdown`
- Covering indexes are important in Distributed SQL (to avoid RPCs)
- Don't `SELECT *` if you don't need all the columns
- Use Partial Index is you don't need the index for all the rows
- Check the join order in the execution plan, provide the right indexes to filter early
- For complex query, pg_hint_plan may be used to define the execution plan

Distributed SQL Tuning

Geo Distribution

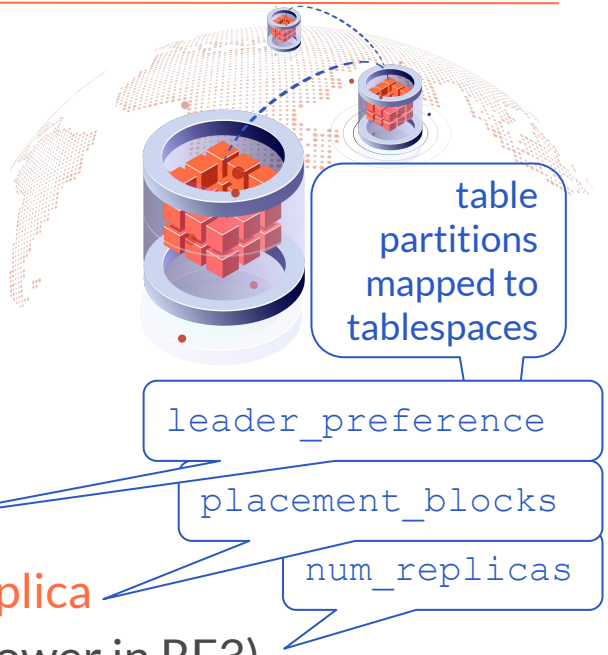
Basics of geo-distribution

You can't change the speed of light

but you can control the distance of your replicas

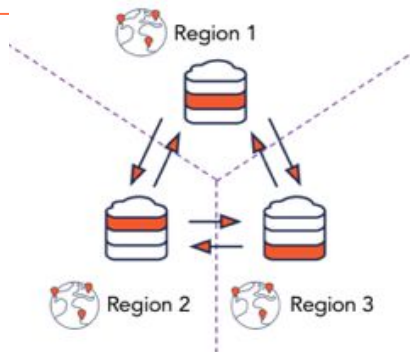
The latency you wait on:

- strongly consistent reads: only the Raft Leader
- timeline consistent read (bounded staleness): Any replica
- writes: the Quorum (Raft Leader and the nearest follower in RF3)



Placement of tables / indexes / partitions

By default, tablets are distributed on all cluster.
When geo-distributed, add placement info:




```
create tablespace rf3 with ( replica_placement= $$ {
    "num_replicas": 3, "placement_blocks": [
{ "zone": "us-east-1", "region": "us-east-1"   , "cloud": "aws" ,
, "min_num_replicas": 1                                          },
{ "zone": "us-west-1a"   "region": "us-west-1" , "cloud": "aws" ,
, "min_num_replicas": 1                                          },
{ "zone": "us-west-2a"   , "region": "us-west-2" , "cloud": "aws" ,
  "min_num_replicas": 1  , "leader_preference": 1                 }
    ] } $$) ;
```

Duplicate Covering Indexes

From mostly-static tables read from everywhere (like reference tables),

create a Covering Index in each region (tablespace)

- reads are local (Index Only Scan)
- writes are broadcasted (in sync)

```
create index i_eu on t (key) include (all columns) tablespace eu;
create index i_us on t (key) include (all columns) tablespace us;
create index i_ap on t (key) include (all columns) tablespace ap;
```

You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)
- EXPLAIN ANALYZE is your way to understand why a query is slow
- Watch for `Rows Removed by Filter` with high amount of rows
- Always enable `yb_enable_expression_pushdown`
- Covering indexes are important in Distributed SQL (to avoid RPCs)
- Don't `SELECT *` if you don't need all the columns
- Use Partial Index is you don't need the index for all the rows
- Check the join order in the execution plan, provide the right indexes to filter early
- For complex query, pg_hint_plan may be used to define the execution plan
- In geo-distributed cluster, table / index (or partition) placement can reduce latency
- Create duplicate covering indexes for reference tables

Distributed SQL Tuning

Follower Reads

Reporting doesn't need to read from the Raft Leader

If your transaction is only reading and can ignore changes in the latest 30 seconds:

```
set default_transaction_read_only = on;
set yb_read_from_followers=on;
set yb_follower_read_staleness_ms= 30000;
```

Can read from a Follower (sync - writes wait on quorum)
Can read from a Read Replica (async - writes don't wait)

You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)
- EXPLAIN ANALYZE is your way to understand why a query is slow
- Watch for `Rows Removed by Filter` with high amount of rows
- Always enable `yb_enable_expression_pushdown`
- Covering indexes are important in Distributed SQL (to avoid RPCs)
- Don't `SELECT *` if you don't need all the columns
- Use Partial Index is you don't need the index for all the rows
- Check the join order in the execution plan, provide the right indexes to filter early
- For complex query, pg_hint_plan may be used to define the execution plan
- In geo-distributed cluster, table / index (or partition) placement can reduce latency
- Create duplicate covering indexes for reference tables
- Identify read-only use-case like reporting and place followers / read replicas

Distributed SQL Tuning

Bulk Load

Faster bulk load

Use packed columns format

💡 `--ysql_enable_packed_row=true`

Create Secondary Indexes and Foreign Key later

single-shard transactions are faster

No need to check for duplicates:

💡 `set yb_enable_upsert_mode=on`

No need for ACID:

💡 `set yb_disable_transactional_writes=on`

bypasses the possibility to rollback (Atomicity) and consistent queries (Isolation)

You should remember

- Define the primary key in the CREATE TABLE rather than ALTER TABLE
- HASH sharding is good to distribute when columns are used in point queries (!= range)
- EXPLAIN ANALYZE is your way to understand why a query is slow
- Watch for `Rows Removed by Filter` with high amount of rows
- Always enable `yb_enable_expression_pushdown`
- Covering indexes are important in Distributed SQL (to avoid RPCs)
- Don't `SELECT *` if you don't need all the columns
- Use Partial Index is you don't need the index for all the rows
- Check the join order in the execution plan, provide the right indexes to filter early
- For complex query, pg_hint_plan may be used to define the execution plan
- In geo-distributed cluster, table / index (or partition) placement can reduce latency
- Create duplicate covering indexes for reference tables
- Identify read-only use-case like reporting and place followers / read replicas
- some features are not enabled by default to allow rolling upgrades, but are recommended

Distributed SQL Tuning

SQL Tuning methodology

Explain plan in real life

When learning, build small queries

In real life, execution plans may be like this:

```
yugabyte=>
yugabyte=>
yugabyte=> explain analyze SELECT column_name, table_name, is_generated, is_identity, identity_generation FROM information_schema.columns where table_schema = 'public';
                                                                                     QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=69.36..120.09 rows=56 width=160) (actual time=3784.825..3852.013 rows=92 loops=1)
   ->  Hash Right Join  (cost=69.36..102.37 rows=56 width=133) (actual time=3784.107..3784.186 rows=92 loops=1)
         Hash Cond: ((dep.refobjid = c.oid) AND (dep.refobjsubid = a.attnum))
         ->  Nested Loop  (cost=0.00..32.25 rows=100 width=8) (actual time=17.219..17.219 rows=0 loops=1)
               ->  Index Scan using pg_depend_reference_index on pg_depend dep  (cost=0.00..17.25 rows=100 width=12) (actual time=17.218..17.218 rows=0 loops=1)
                     Index Cond: (refclassid = '1259'::oid)
                     Filter: ((classid = '1259'::oid) AND (deptype = 'i'::"char"))
                     Rows Removed by Filter: 876
               ->  Index Scan using pg_sequence_seqrelid_index on pg_sequence seq  (cost=0.00..0.15 rows=1 width=4) (never executed)
                     Index Cond: (seqrelid = dep.objid)
         ->  Hash  (cost=68.52..68.52 rows=56 width=139) (actual time=3766.877..3766.877 rows=92 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 24kB
               ->  Nested Loop  (cost=0.00..68.52 rows=56 width=139) (actual time=3553.979..3766.712 rows=92 loops=1)
                     ->  Nested Loop  (cost=0.00..62.14 rows=56 width=143) (actual time=3553.493..3726.749 rows=92 loops=1)
                           ->  Nested Loop Left Join  (cost=0.00..54.63 rows=56 width=147) (actual time=45.984..2929.645 rows=1752 loops=1)
                                 Join Filter: (t.typtype = 'd'::"char")
                                 ->  Nested Loop  (cost=0.00..41.15 rows=56 width=152) (actual time=45.545..1887.008 rows=1752 loops=1)
                                       ->  Nested Loop  (cost=0.00..32.75 rows=56 width=147) (actual time=44.995..1058.333 rows=1752 loops=1)
                                             ->  Index Scan using pg_attribute_relid_attnum_index on pg_attribute a  (cost=0.00..15.25 rows=100 width=79) (actual time=30.992..38.581 rows=2087 loops=1)
                                                   Index Cond: (attnum > 0)
                                                   Filter: (NOT attisdropped)
                                             ->  Index Scan using pg_class_oid_index on pg_class c  (cost=0.00..0.17 rows=1 width=76) (actual time=0.474..0.474 rows=1 loops=2087)
                                                   Index Cond: (oid = a.attrelid)
                                                   Filter: ((relkind = ANY ('{r,v,f,p}'::"char"[])) AND (pg_has_role(relowner, 'USAGE'::text) OR has_column_privilege(oid, a.attnum, 'SELECT, INSERT, UPDATE, REFERENCES'::text)))
                                                   Rows Removed by Filter: 0
                                       ->  Index Scan using pg_type_oid_index on pg_type t  (cost=0.00..0.15 rows=1 width=13) (actual time=0.459..0.459 rows=1 loops=1752)
                                             Index Cond: (oid = a.atttypid)
                                 ->  Nested Loop  (cost=0.00..0.23 rows=1 width=4) (actual time=0.593..0.593 rows=0 loops=1752)
                                       ->  Index Scan using pg_type_oid_index on pg_type bt  (cost=0.00..0.11 rows=1 width=8) (actual time=0.423..0.424 rows=0 loops=1752)
                                             Index Cond: (t.typbasetype = oid)
                                       ->  Index Scan using pg_namespace_oid_index on pg_namespace nbt  (cost=0.00..0.11 rows=1 width=4) (actual time=0.427..0.427 rows=1 loops=627)
                                             Index Cond: (oid = bt.typnamespace)
                           ->  Index Scan using pg_namespace_oid_index on pg_namespace nc  (cost=0.00..0.13 rows=1 width=4) (actual time=0.443..0.443 rows=0 loops=1752)
                                 Index Cond: (oid = c.relnamespace)
                                 Filter: ((NOT pg_is_other_temp_schema(oid)) AND (((nspname)::information_schema.sql_identifier)::text = 'public'::text))
                                 Rows Removed by Filter: 1
                     ->  Index Scan using pg_namespace_oid_index on pg_namespace nt  (cost=0.00..0.11 rows=1 width=4) (actual time=0.423..0.423 rows=1 loops=92)
                           Index Cond: (oid = t.typnamespace)
   ->  Nested Loop  (cost=0.00..0.28 rows=1 width=4) (actual time=0.734..0.734 rows=0 loops=92)
         ->  Index Scan using pg_collation_oid_index on pg_collation co  (cost=0.00..0.15 rows=1 width=72) (actual time=0.429..0.430 rows=1 loops=92)
               Index Cond: (a.attcollation = oid)
         ->  Index Scan using pg_namespace_oid_index on pg_namespace nco  (cost=0.00..0.12 rows=1 width=68) (actual time=0.446..0.446 rows=0 loops=59)
               Index Cond: (oid = co.collnamespace)
               Filter: ((nspname <> 'pg_catalog'::name) OR (co.collname <> 'default'::name))
               Rows Removed by Filter: 1
 Planning Time: 1.142 ms
 Execution Time: 3852.179 ms
 Peak Memory Usage: 297 kB
(48 rows)

yugabyte=>
yugabyte=>
```

Tune with method

ong-term solution

g_stat_statements)

an

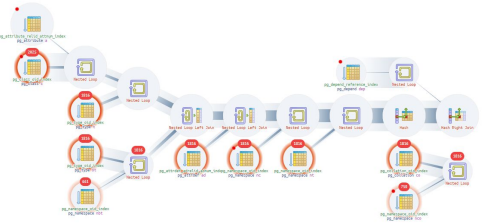
- Be sure to have the right indexes
- Understand read and write latency
- detect hotspots from the GUI or other tools (yb_stats)

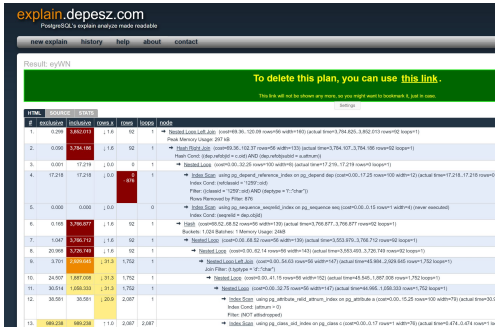
Drill down to the root cause

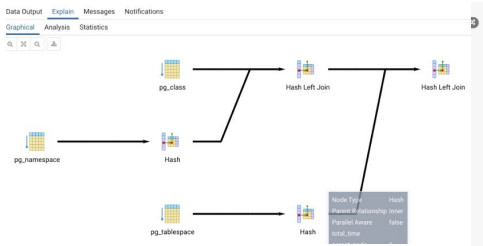
```
explain analyze select *
from information_schema.columns
```

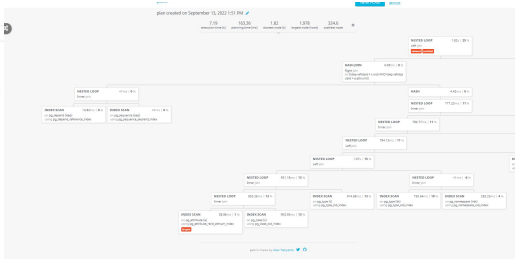
you can use:

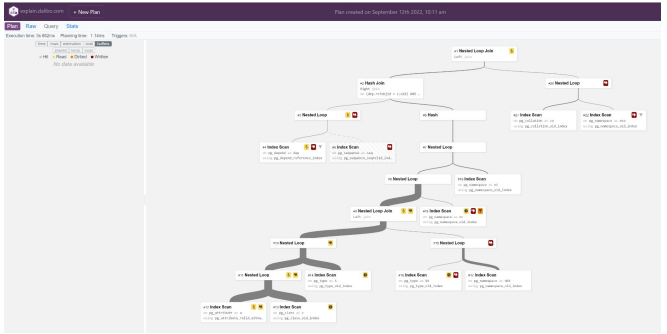
https://explain.depesz.com
http://tatiyants.com/pev
https://explain.dalibo.com
https://explain-postgresql.com





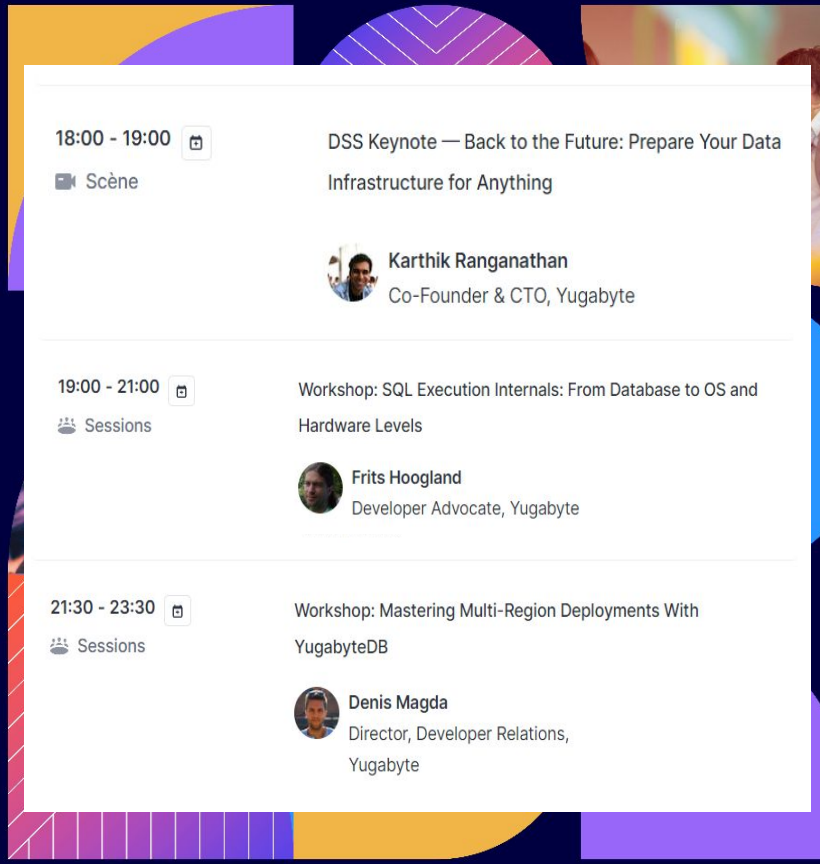






Do you remember...?

1. Define the primary key in the CREATE TABLE rather than ALTER TABLE
2. HASH sharding is good to distribute when columns are used in point queries (!= range)
3. EXPLAIN ANALYZE is your way to understand why a query is slow
4. Watch for `Rows Removed by Filter` with high amount of rows
5. Always enable `yb_enable_expression_pushdown`
6. Covering indexes are important in Distributed SQL (to avoid RPCs)
7. Don't `SELECT *` if you don't need all the columns
8. Use Partial Index is you don't need the index for all the rows
9. Check the join order in the execution plan, provide the right indexes to filter early
10. For complex query, pg_hint_plan may be used to define the execution plan
11. In geo-distributed cluster, table / index (or partition) placement can reduce latency
12. Create duplicate covering indexes for reference tables
13. Identify read-only use-case like reporting and place followers / read replicas
14. some features are not enabled by default to allow rolling upgrades, but are recommended



Thank You

Join us on Slack: yugabyte.com/slack

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