Deeper Understanding of PostgreSQL Execution Plan: At plan time and run time

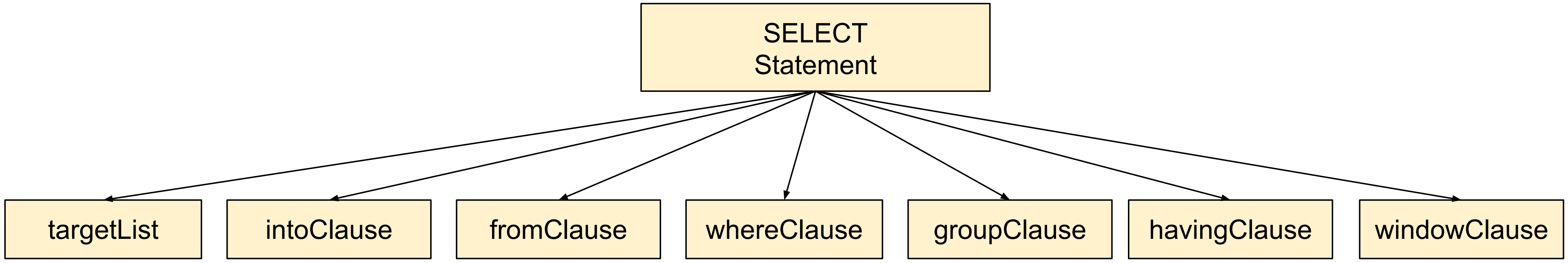
What affects the execution plan of a statement

Jobin Augustine
Senior Support Engineer
PGConf India 2020, Banglore

PERCONA

SQL Execution

- Lex and Parse - flex and Bison
- Analyze - semantic
- Rewrite - Rules
- Plan and Optimize
- Execute



Parse

More than 16400 lines

124 types of statements

```
<node>    stmt schema_stmt
          AlterEventTrigStmt AlterCollationStmt
          AlterDatabaseStmt AlterDatabaseSetStmt AlterDomainStmt AlterEnumStmt
          AlterFdwStmt AlterForeignServerStmt AlterGroupStmt
          AlterObjectDependsStmt AlterObjectSchemaStmt AlterOwnerStmt
          AlterOperatorStmt AlterSeqStmt AlterSystemStmt AlterTableStmt
          AlterTblSpcStmt AlterExtensionStmt AlterExtensionContentsStmt AlterForeignTableStmt
          AlterCompositeTypeStmt AlterUserMappingStmt
          AlterRoleStmt AlterRoleSetStmt AlterPolicyStmt AlterStatsStmt
          AlterDefaultPrivilegesStmt DefACLAction
          AnalyzeStmt CallStmt ClosePortalStmt ClusterStmt CommentStmt
          ConstraintsSetStmt CopyStmt CreateAsStmt CreateCastStmt
          CreateDomainStmt CreateExtensionStmt CreateGroupStmt CreateOpClassStmt
          CreateOpFamilyStmt AlterOpFamilyStmt CreatePLangStmt
          CreateSchemaStmt CreateSeqStmt CreateStmt CreateStatsStmt CreateTableSpaceStmt
          CreateFdwStmt CreateForeignServerStmt CreateForeignTableStmt
          CreateAssertionStmt CreateTransformStmt CreateTrigStmt CreateEventTrigStmt
          CreateUserStmt CreateUserMappingStmt CreateRoleStmt CreatePolicyStmt
          CreatedbStmt DeclareCursorStmt DefineStmt DeleteStmt DiscardStmt DoStmt
          DropOpClassStmt DropOpFamilyStmt DropPLangStmt DropStmt
          DropCastStmt DropRoleStmt
          DropdbStmt DropTableSpaceStmt
          DropTransformStmt
          DropUserMappingStmt ExplainStmt FetchStmt
          GrantStmt GrantRoleStmt ImportForeignSchemaStmt IndexStmt InsertStmt
          ListenStmt LoadStmt LockStmt NotifyStmt ExplainableStmt PreparableStmt
          CreateFunctionStmt AlterFunctionStmt ReindexStmt RemoveAggrStmt
          RemoveFuncStmt RemoveOperStmt RenameStmt RevokeStmt RevokeRoleStmt
          RuleActionStmt RuleActionStmtOrEmpty RuleStmt
          SecLabelStmt SelectStmt TransactionStmt TruncateStmt
          UnlistenStmt UpdateStmt VacuumStmt
          VariableResetStmt VariableSetStmt VariableShowStmt
          ViewStmt CheckPointStmt CreateConversionStmt
          DeallocateStmt PrepareStmt ExecuteStmt
          DropOwnedStmt ReassignOwnedStmt
          AlterTSConfigurationStmt AlterTSDictionaryStmt
          CreateMatViewStmt RefreshMatViewStmt CreateAmStmt
          CreatePublicationStmt AlterPublicationStmt
          CreateSubscriptionStmt AlterSubscriptionStmt DropSubscriptionStmt
```

Parse - Select

SELECT syntax:

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ] [...] ]

where from_item can be one of:

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
                [ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias ( column_definition [, ...] )
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    [ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS ( column_definition [, ...] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]

and grouping_element can be one of:

    ( )
    expression
    ( expression [, ...] )
    ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
    CUBE ( { expression | ( expression [, ...] ) } [, ...] )
    GROUPING SETS ( grouping_element [, ...] )

and with_query is:
```


PERCONA

Explainable Statements

- SELECT
- INSERT
- UPDATE
- DECLARE CURSOR
- CREATE AS
- CREATE MATERIALIZED VIEW
- REFRESH MATERIALIZED VIEW
- EXECUTE

Traffic Cop

Splits simple and complex query

Takes Raw Parse tree as input

PERCONA

Analyze

Meaning of the query

- **Relations are identified**
- columns, datatype, collation etc are considered
- **Transform** a **Parse tree** into a **Query tree**

```
pg_analyze_and_rewrite(...)
   parse_analyze(...)
   pg_rewrite_query(...)
```

Query Rewrite

- Set of Rules are applied
- transform the query
- between parser and the planner/optimizer
- custom rules stored in pg_rules

```
static Query *transformOptionalSelectInto(ParseState *pstate, Node *parseTree);
static Query *transformDeleteStmt(ParseState *pstate, DeleteStmt *stmt);
static Query *transformInsertStmt(ParseState *pstate, InsertStmt *stmt);
static List *transformInsertRow(ParseState *pstate, List *exprlist,
                                 List *stmtcols,
                                 bool strip_indir
static OnConflictExpr *transformOnConflictClause(ParseState *pstate,

static int      count_rowexpr_columns(ParseState *pstate, Node *expr);
static Query *transformSelectStmt(ParseState *pstate, SelectStmt *stmt);
static Query *transformValuesClause(ParseState *pstate, SelectStmt *stmt);
static Query *transformSetOperationStmt(ParseState *pstate, SelectStmt *stmt);
static Node *transformSetOperationTree(ParseState *pstate, SelectStmt *stmt,
                                                                     bool
static void determineRecursiveColTypes(ParseState *pstate,
                                                                     Node
static Query *transformUpdateStmt(ParseState *pstate, UpdateStmt *stmt);
static List *transformReturningList(ParseState *pstate, List *returningList);
static List *transformUpdateTargetList(ParseState *pstate,
                                                                     List
```

CREATE [OR REPLACE] RULE *name* AS ON *event*
TO *table_name* [WHERE *condition*]
DO [ALSO | INSTEAD] { NOTHING | *command* | (*command* ; *command* ...) }

```
set log_parser_stats=on;
```

A **Query Tree** is the logical Represenation of the query with reference to actual database objects with object id

Rules and Rewrite : Example

Parameter Settings

```
CREATE VIEW pg_settings AS
    SELECT * FROM pg_show_all_settings() AS A;
```

Views

```
select * from tv;
```

=>

```
select * from (
 select id from t1) tv;
```

- Rewriter replaces the view def.
- Planner optimizes the query by pulling up the inner queries.

Planner

- **SQL query / Query tree can be actually executed in a wide variety of different ways**
- **planner / optimizer create an optimal execution plan**
- **Cost Based**

- **Data structure - Paths.**
  - Paths are cut-down versions of Plans
- **Cheapest path is selected**
- **Full-fledged Plan will be prepared.**

Executor

- Plan nodes are executed **recursively**
- top level node returns the result to client

temp_buffers and work_mem are used

creates temporary files if necessary

Plan Optimizer - Key Decisions

1. **Scan Method**
   - Sequential Scan
   - Index Scan
   - Bitmap Index Scan

2. **Join Method**
   - Nested Loop
   - Hash Joins
   - Merge Joins

3. **Join Order**

   If there are more than 2 relations, planner examines different possible join sequences to find the cheapest one

Planner

- **Plan Tree is Prepared for the cheapest path**
- **Considers Statistics**
  - Estimate of cost of each access path
- **Cost based** **if the number of tables is lessshow than around 12**
- **Generic Query Optimizer (geqo)**

Plan with least cost estimation is taken for execution

PERCONA

Plan Example - Bitmap Index/Heap Scan

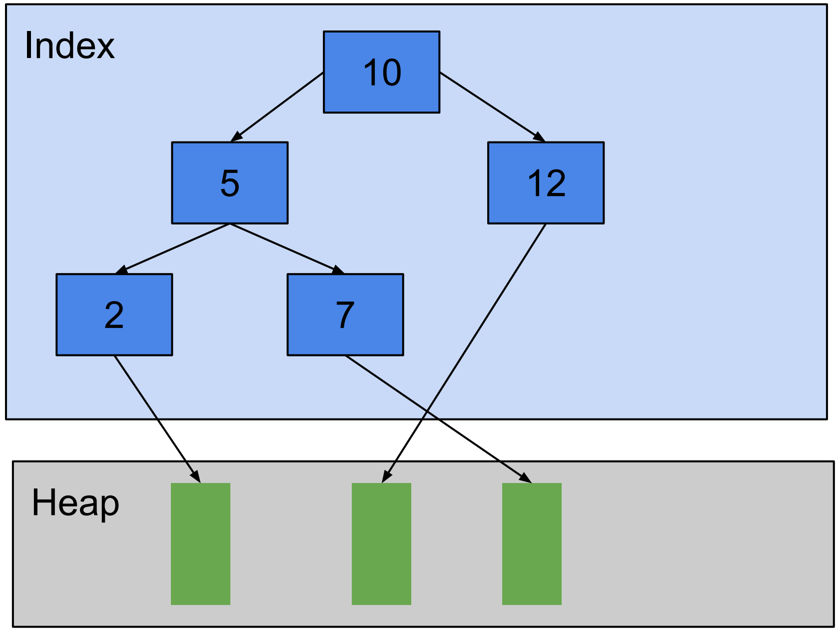
Sub optimal plans?
How to Read a Plan

++

Bitmap scans are good compromise / middle ground between Sequential Scan and Index scan

Index scan

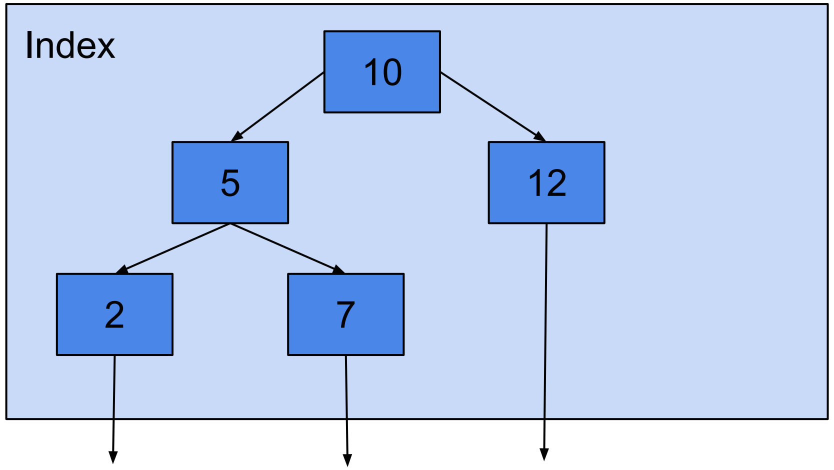
- Fetch one tuple-pointer at a time from the index
- Immediately visits that tuple in the table



PERCONA

Index Only Scan

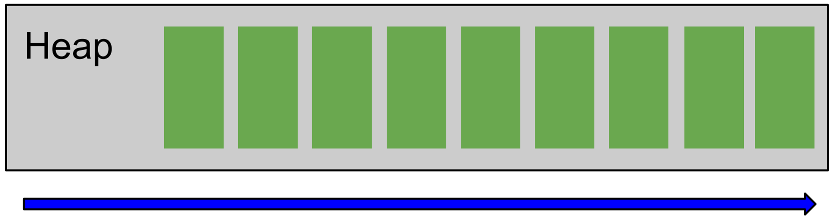
Read Index Tree, But Don't Read Heap Pages



PERCONA

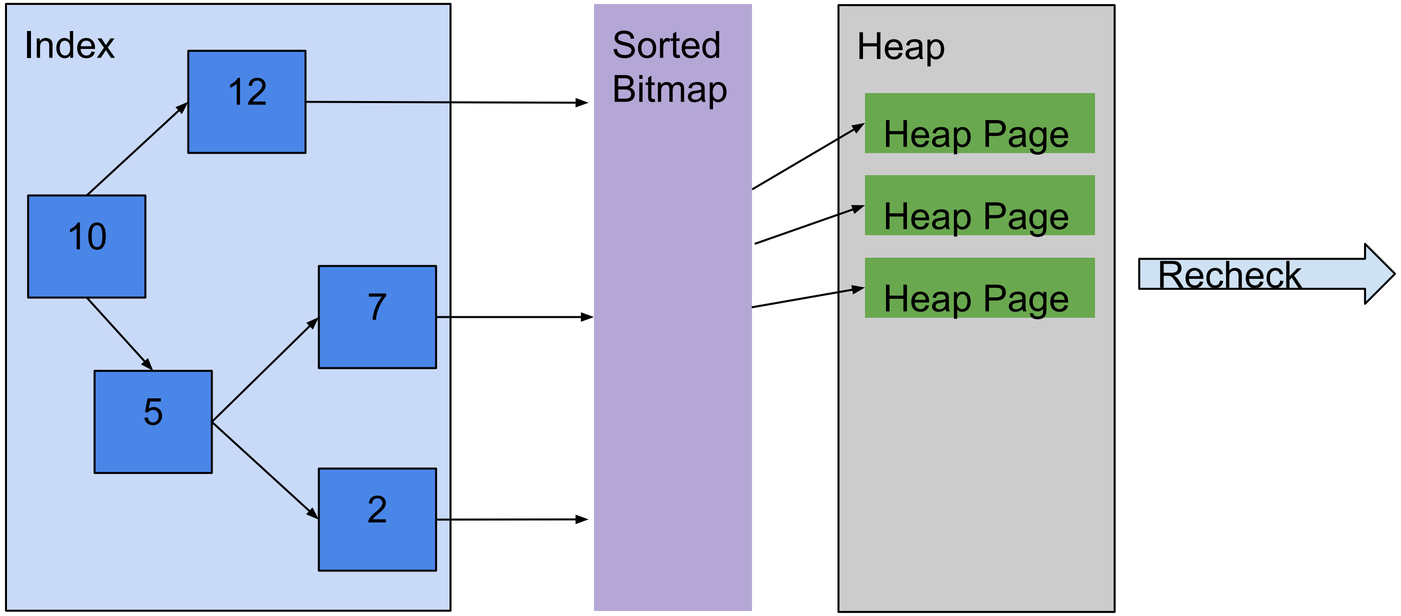
Sequential Scan

Scans the tuples from one end to another end discarding all unmatched rows



Bitmap Index/Heap Scan

1. A bitmap scan fetches all the tuple-pointers from the index in one go
2. Sorts them using an in-memory "bitmap" data structure
3. Then visits the table tuples in physical tuple-location order.
4. Recheck for filtering condition



PERCONA

Bitmap Scan

++

The bitmap scan improves locality of reference to the table

--

bookkeeping overhead to manage the "bitmap" data structure
data is no longer retrieved in index order

PERCONA

ANALYZE

- AUTOVACUUM
- pg_statistic and pg_stats
- default_statistics_target

PERCONA

Statistics

- Autovacuum worker
- Asynchronous

++

Better Execution Plan

Cost based Plan

--

Statistics collection overhead

More data to analyze for arriving at execution plan

Nested loop join

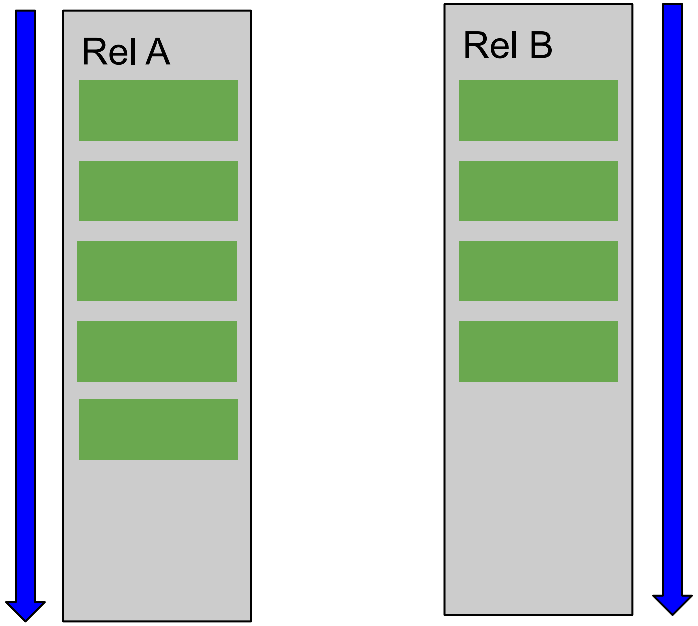
- Right relation is scanned once for every row found in the left relation
- Good strategy if index scan is possible on Right relation

Hash Joins

- Equi Joins
- In memory
- Generally the fastest

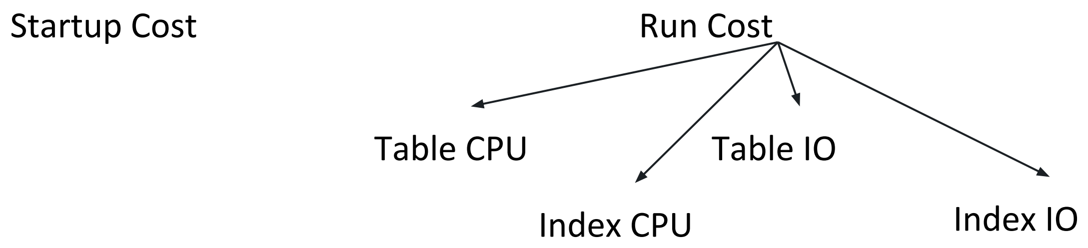
Merge Joins

- Relations are Sorted on the join attributes
- Equi-Join
- If data to join is too big to fit in memory



Cost

- **Its a reference number for comparison. Assuming that cost of sequentially accessing a page is 1.**
- **seq_page_cost, random_page_cost, cpu_tuple_cost,cpu_index_tuple_cost**




```
void
cost_seqscan(Path *path,... )
{
    Cost       startup_cost = 0;
    Cost       cpu_run_cost;
    Cost       disk_run_cost;
    double     spc_seq_page_cost;
    QualCost   qpqual_cost;
    Cost       cpu_per_tuple;
```

- **Tune at Per Session / User**
- **Tune at Per tablespace / Storage**

```
create tablespace tbs1 LOCATION '/var/lib/pgsql/11/tbs1' WITH
(random_page_cost=1.1, seq_page_cost=1, effective_io_concurrency=2);
```

Stastistics - Selectivity

- **Most Common Value (MCV) and Histogram**
- **histogram bounds**

Extended Statistics

- **Functional dependencies**

```
CREATE STATISTICS stts1 (dependencies) ON  city_id,state_id FROM
```

- **N-Distinct**

```
CREATE STATISTICS stts2 (dependencies, ndistinct) ON  city_id,state_id FROM participant;
```

- **Multi-Variate MCV List**
  - Not Just Column level, But Row level too.
  - More expensive in building the statistics, storage and planning time

```
CREATE STATISTICS stts (mcv) ON col1, col2 FROM tab1
```

Parallel execution

- **Great Advantage for OLAP load**
- **Takes more server resources**
- **Test and Verify each query**

parallel_setup_cost -> Decides the plan

max_parallel_workers_per_gather -> Number of works in plan

Table level setting : `SET ( parallel_workers = N)`

max_parallel_workers -> Decides execution

JIT

- Turning some form of interpreted program evaluation into a native program
- Replace arbitrary expression to compiled functions

  `Example : WHERE a.col = 3`
- Expression evaluation and tuple deforming
- In-lining new data types, functions, operators and other database objects

Good for

- Query containing complex expressions
- Query processing large volume of data
- Complex query

Typically OLAP workload and those queries which takes longer duration.

JIT

```
SELECT COMPANY_ID,TRADE_TS::DATE DT,
       SUM(SHARES) TOT_SHARES,
       SUM(SHARES* RATE) TOT_INVEST,
       MIN(SHARES* RATE) MIN_TRASACTION,
       SUM(SHARES* RATE * 0.002) BROKERAGE,
       SUM(SHARES* RATE * 0.002 + SHARES* RATE * 0.002*0.15) BROKERAGE_PLUS_SERVICE_TAX,
       SUM(SHARES* RATE * 0.002 + SHARES* RATE * 0.002*0.15 + SHARES* RATE*0.0001) FULL_SPEND,
       SUM(SHARES* RATE * 0.002*0.15 + SHARES* RATE*0.0001) EXPENSES_ONLY,
       AVG(SHARES* RATE * 0.002 + SHARES* RATE * 0.002*0.15 + SHARES* RATE*0.0001) AVG_SPEND
FROM TRADING
GROUP BY COMPANY_ID,TRADE_TS::DATE
ORDER BY COMPANY_ID,DT;
```

With Parallel

```
 JIT:
   Functions: 34
   Options: Inlining true, Optimization true,
Expressions true, Deforming true
   Timing: Generation 14.923 ms, Inlining 179.005 ms,
Optimization 328.287 ms, Emission 194.164 ms, Total
716.379 ms
 Execution Time: 227053.360 ms
```

- Parallel execution has negative impact
- Difference visible if IO is not bottleneck

Without Parallel

```
 JIT:
   Functions: 9
   Options: Inlining true, Optimization true,
Expressions true, Deforming true
   Timing: Generation 5.742 ms, Inlining 34.036 ms,
Optimization 76.847 ms, Emission 46.334 ms, Total
162.958 ms
 Execution Time: 211243.728 ms
```

```
set jit=off;
```

Prepared Statements - Save Tax

Reason to have Prepare a Statement:
Complex, time and resource consuming steps involved in SQL Processing

- Lex and Parse
- Analysis
- Rewrite
- Plan
- Execute

Planning time: 150.562 ms

Execution time: 5.663 ms

PERCONA

Need for Plan cache

```
postgres=# EXPLAIN ANALYZE SELECT count(*) FROM COMPANY WHERE COMPANY_TYPE=1;
                                            QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Aggregate  (cost=103.06..103.07 rows=1 width=8) (actual time=2.354..2.354 rows=1 loops=1)
   ->  Seq Scan on company  (cost=0.00..90.56 rows=5000 width=0) (actual time=0.386..2.014 rows=5000 loops=1)
         Filter: (company_type = 1)
         Rows Removed by Filter: 5
 Planning Time: 7.813 ms
 Execution Time: 3.836 ms
(6 rows)
```

PREPARABLE Statements

- SELECT
- INSERT
- UPDATE
- DELETE

Prepare it in Advance

```
PREPARE preplan(int) AS SELECT  count(*) FROM  company WHERE   company_type = $1
```

```
EXPLAIN ANALYZE execute preplan(1);
```

```
EXPLAIN ANALYZE execute preplan(2);
```

Replaning happen

- ~~Lex and Parse~~
- ~~Analysis~~
- ~~Rewrite~~
- Plan
- Execute

Repated Execution - Generic Plan

```
PREPARE preplan(int) AS SELECT  count(*) FROM  company WHERE   company_type = $1
```

```
EXPLAIN ANALYZE execute preplan(1);
```

```
EXPLAIN ANALYZE execute preplan(2);
```


PERCONA

Generic Plan is selcted

- ~~Lex and Parse~~
- ~~Analysis~~
- ~~Rewrite~~
- ~~Plan~~
- Execute

Repeated Execution which ends up in same execution plan

“It occurs only after **five or more executions produce plans whose estimated cost average (including planning overhead) is more expensive** than the **generic plan** cost estimate”

Repated Execution

```
PREPARE preplan(int) AS SELECT  count(*) FROM  company WHERE   company_type = $1
```

```
EXPLAIN ANALYZE execute preplan(2);
```

```
EXPLAIN ANALYZE execute preplan(1);
```

What's happening?

“Using EXECUTE values which are rare in columns with many duplicates can generate **custom plans** that are **so much cheaper than the generic plan, even after adding planning overhead, that the generic plan might never be used**”

PERCONA

PG 12 - plan_cache_mode

```
SET plan_cache_mode=force_custom_plan;
```

```
SET plan_cache_mode=force_generic_plan;
```

Functions

```
CREATE OR REPLACE FUNCTION funcplan(int) RETURNS bigint AS $$
 begin
   RETURN (SELECT  count(*) FROM  company WHERE   company_type = $1);
 end;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION funcplan(int) RETURNS bigint AS $$
     SELECT  count(*) FROM  company WHERE   company_type = $1;
$$ LANGUAGE SQL;
```

```
SELECT funcplan(1);
```

```
SELECT funcplan(2);
```

Functions

- The PL/pgSQL interpreter parses the function's source text and produces an internal binary instruction tree the first time the function is called (within each session).
- Individual SQL expressions and SQL commands used in the function are not translated immediately.
-

Recommended References

Reference
https://www.tutorialdba.com/p/postgresql-query-flow.html
Effect of width on scan
http://www.interdb.jp/pg/pgsql03.html
Function cache
https://www.postgresql.org/docs/current/plpgsql-implementation.html#PLPGSQL-PLAN-CACHING
https://www.endpoint.com/blog/2008/12/11/why-is-my-function-slow

PERCONA

Champions of Unbiased
Open Source Database Solutions