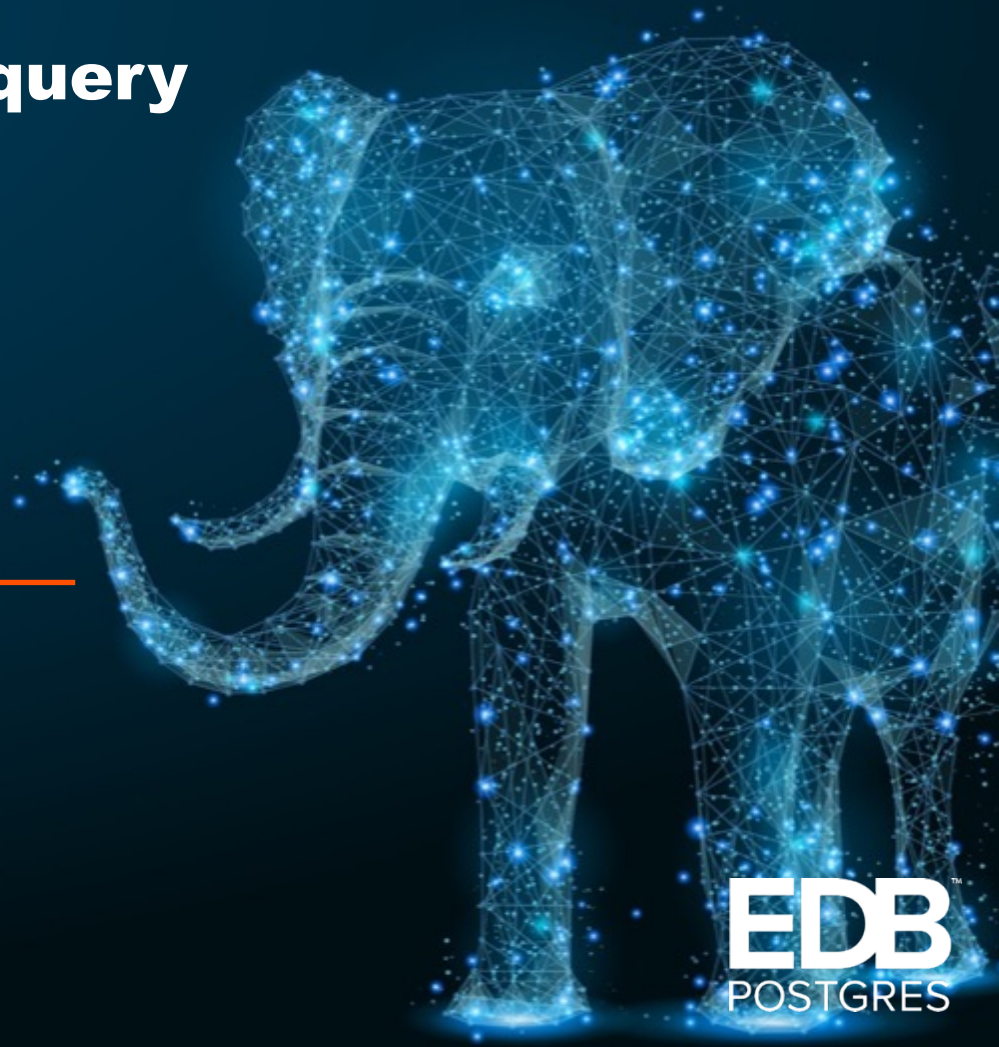
Journey of a SELECT query

27-Feb-2020 @ PGConf India

Vaibhav Dalvi



EDB POSTGRES

Who am I?

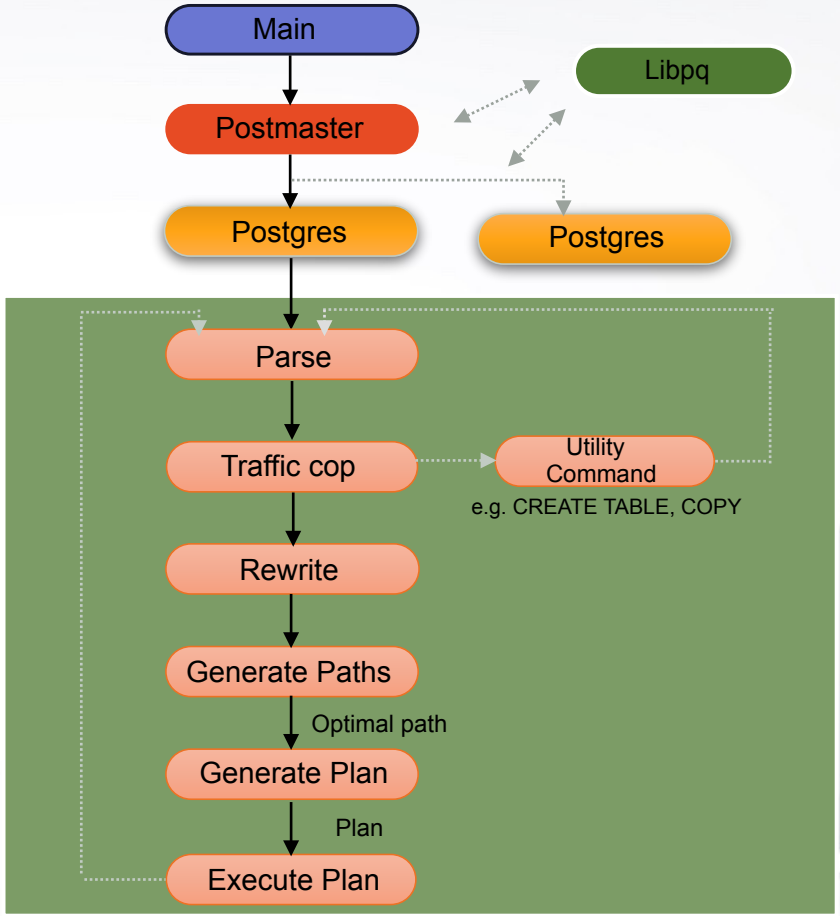
- My name is Vaibhav Dalvi
- I am a Database Developer at **EnterpriseDB**, working from Pune, India office
- Started hacking Postgres few years ago
- Email: vaibhav.dalvi@enterprisedb.com

Agenda:

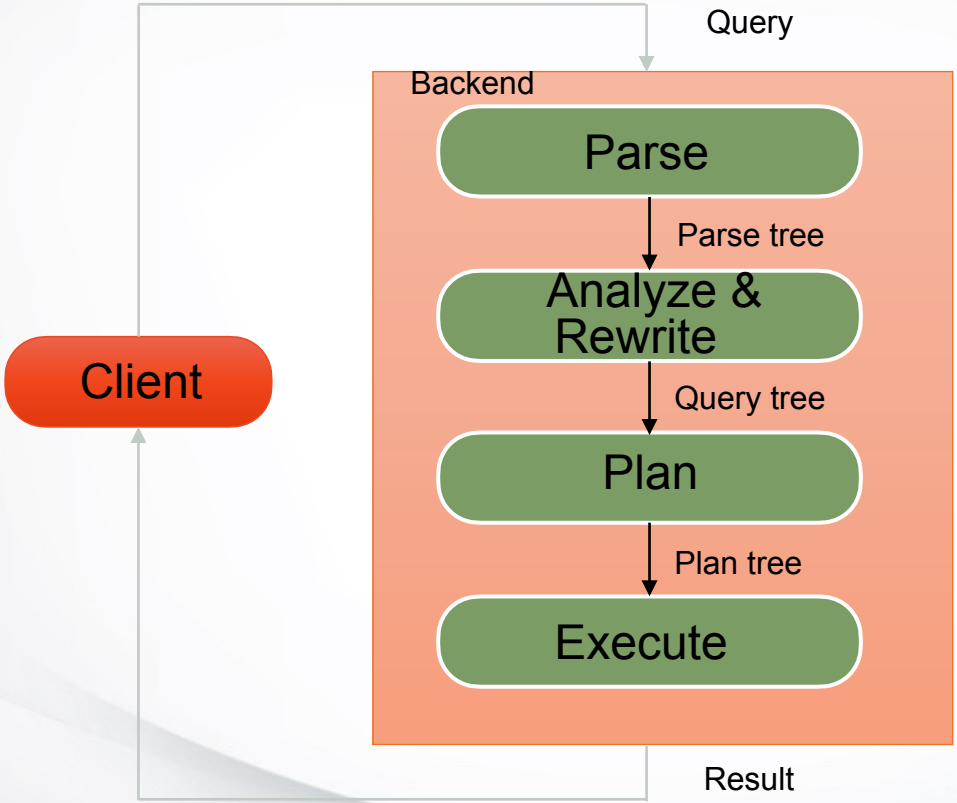
- Brief about backend
- Main query processing
- Entry point for all queries
- Parse
- Analyze and Rewrite
- Plan
- Execution

Backend:

- Initialization
  - Bootstrap
  - Main
  - Postmaster
  - Libpq
- Main query flow
  - Tcop
  - Parser
  - Analyzer & Rewriter
  - Optimizer/Planner
  - Executor



What does postgres do with sql string?



SQL query:

```
SELECT *
FROM student
WHERE name = 'Vaibhav';
```

exec_simple_query:

```
/*
 * exec_simple_query
 *
 * Execute a "simple Query" protocol message.
 */
static void
exec_simple_query(const char *query_string)
{
    .
    .
    parsetree_list = pg_parse_query(query_string);
    .
    .
    querytree_list = pg_analyze_and_rewrite(parsetree, query_string,
                                            NULL, 0, NULL);
    .
    .
    plantree_list = pg_plan_queries(querytree_list,
                                    CURSOR_OPT_PARALLEL_OK, NULL);
    .
    .
    PortalStart(portal, NULL, 0, InvalidSnapshot);
    .
    .
}
```

Parse:

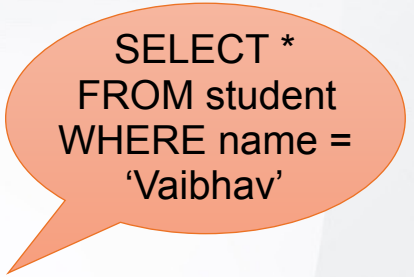
- Lexer

- Parser

pg_parse_query:

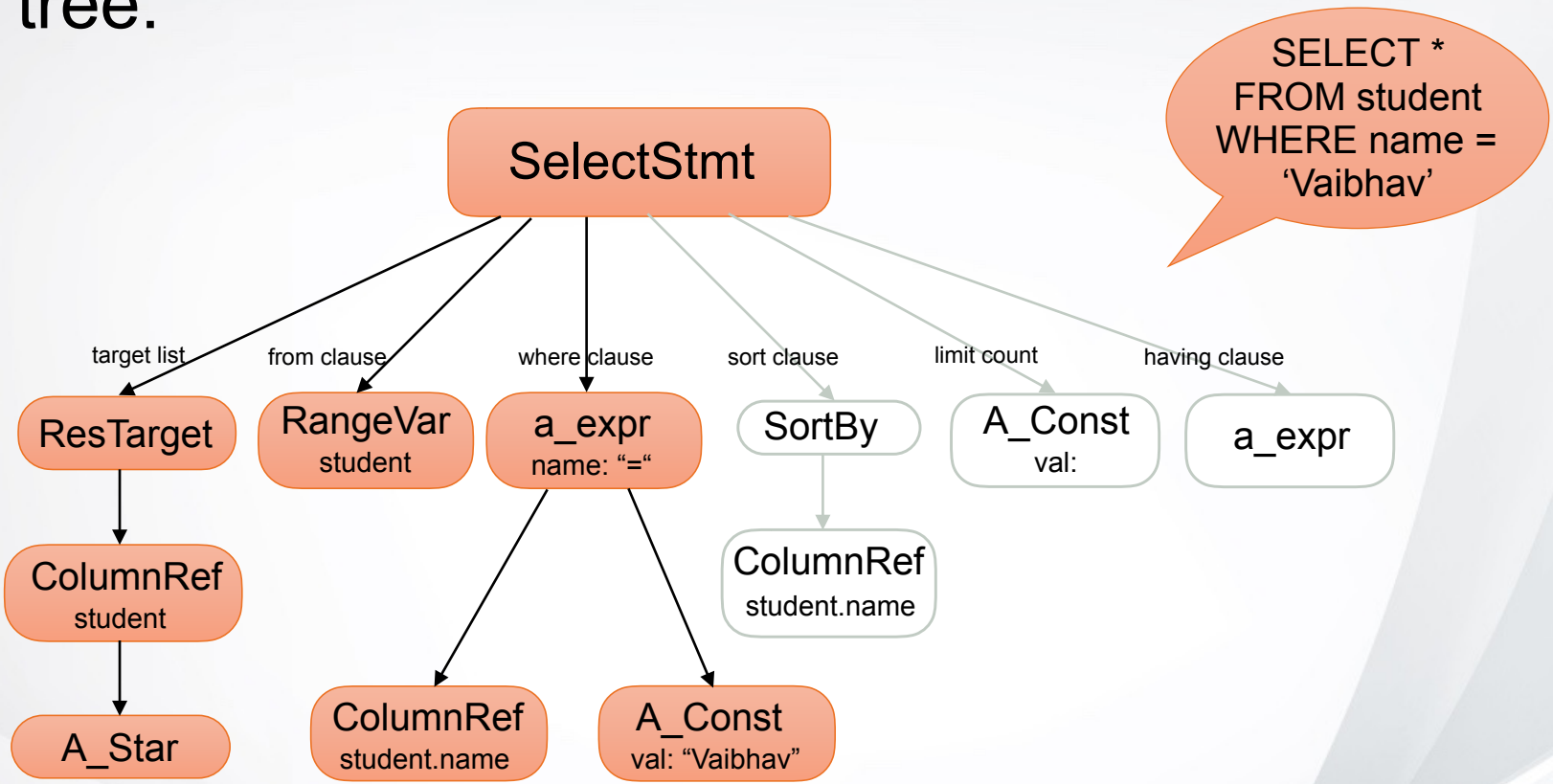
```
/*
 * Do raw parsing (only).
 *
 * A list of parsetrees (RawStmt nodes) is returned, since there might be
 * multiple commands in the given string.
 .
 */
List *
pg_parse_query(const char *query_string)
{
    .
    .
    raw_parsetree_list = raw_parser(query_string);
    return raw_parsetree_list;
}

/*
 * raw_parser
 *      Given a query in string form, do lexical and grammatical analysis.
 .
 */
List *
raw_parser(const char *str)
{
    yyscanner = scanner_init(str, &yyextra.core_yy_extra, &ScanKeywords,
                             ScanKeywordTokens);
    parser_init(&yyextra);
    yyresult = base_yyparse(yyscanner);
    scanner_finish(yyscanner);
    return yyextra.parsetree;
}
```

“Select” parser action:




```
simple_select:
            SELECT opt_all_clause opt_target_list
            into_clause from_clause where_clause
            group_clause having_clause window_clause
               {
                    SelectStmt *n = makeNode(SelectStmt);
                    n->targetList = $3;
                    n->intoClause = $4;
                    n->fromClause = $5;
                    n->whereClause = $6;
                    n->groupClause = $7;
                    n->havingClause = $8;
                    n->windowClause = $9;
                    $$ = (Node *)n;
               }
```

Parse tree:



Analyze and rewrite:

```
/*
 * Given a raw parsetree (gram.y output), and optionally information about
 * types of parameter symbols ($n), perform parse analysis and rule rewriting.
 *
 * A list of Query nodes is returned, since either the analyzer or the
 * rewriter might expand one query to several.
 *
 * NOTE: for reasons mentioned above, this must be separate from raw parsing.
 */
List *
pg_analyze_and_rewrite(RawStmt *parsetree, const char *query_string,
                       Oid *paramTypes, int numParams,
                       QueryEnvironment *queryEnv)

{
    .
    .
    query = parse_analyze(parsetree, query_string, paramTypes, numParams,
                          queryEnv);
    .
    .
    querytree_list = pg_rewrite_query(query);
    .
    .
}
```

Example of RLS:

- If table student has following data into it,

```
#select * from student;
 id |    name     | class
----+-------------+-------
  1 | Roshan      | A
  1 | Suraj       | B
  2 | Vaibhav     | B
```

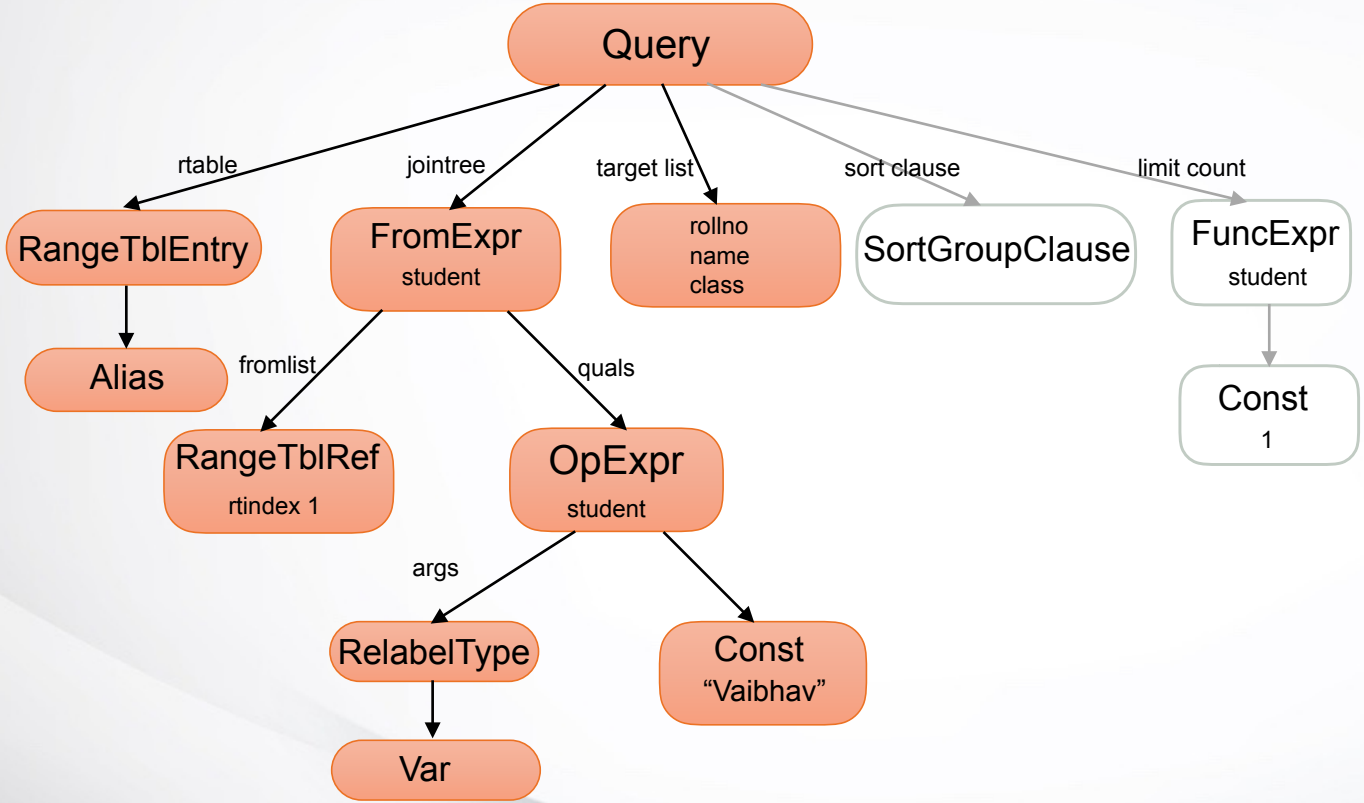
- Policy created on table student,

```
CREATE POLICY p1 ON student FOR ALL USING (class = 'A');
```

- Same query after applying policy,

```
#explain(costs off) select * from student;
          QUERY PLAN
------------------------------
 Seq Scan on student
   Filter: (class = 'A'::text)
```

Query tree:



SELECT *
FROM student
WHERE name =
‘Vaibhav’

Plan:

- Paths
- Cost
- Access paths
- Best path
- Plan tree

pg_plan_queries

```
/* Generate plans for a list of already-rewritten queries.
 *
 * For normal optimizable statements, invoke the planner.  For utility
 * statements, just make a wrapper PlannedStmt node.
 *
 * The result is a list of PlannedStmt nodes.
 */
List *
pg_plan_queries(List *querytrees, int cursorOptions, ParamListInfo boundParams)
{
    .
    .
    stmt = pg_plan_query(query, cursorOptions, boundParams);
    .
    .
}
/*
 * Generate a plan for a single already-rewritten query.
 * This is a thin wrapper around planner() and takes the same parameters.
 */
PlannedStmt *
pg_plan_query(Query *querytree, int cursorOptions, ParamListInfo boundParams)
{
    .
    .
    plan = planner(querytree, cursorOptions, boundParams);
    .
    .
}
```

Create plan tree:

- The planner performs following steps:
  1. Preprocessing
  2. Finding cheapest access path by estimating the costs of all possible access paths
  3. Creating plan tree from the cheapest path
- *PlannerInfo* structure

Preprocessing:

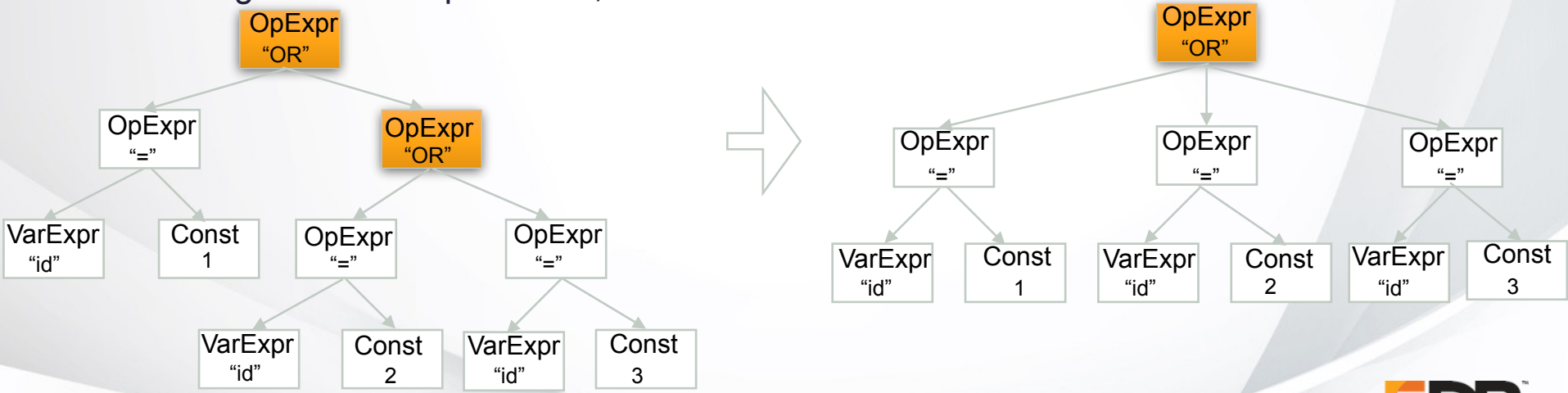
- Simplifying target lists, limit clauses and so on

  For example, ‘2 + 3’ is rewritten to 5 by the *eval_const_expressions()* function defined in *clauses.c*

- Normalizing Boolean expressions,

  For example, ‘NOT(NOT b)’ is rewritten to ‘b’

- Flattening AND/OR expressions,



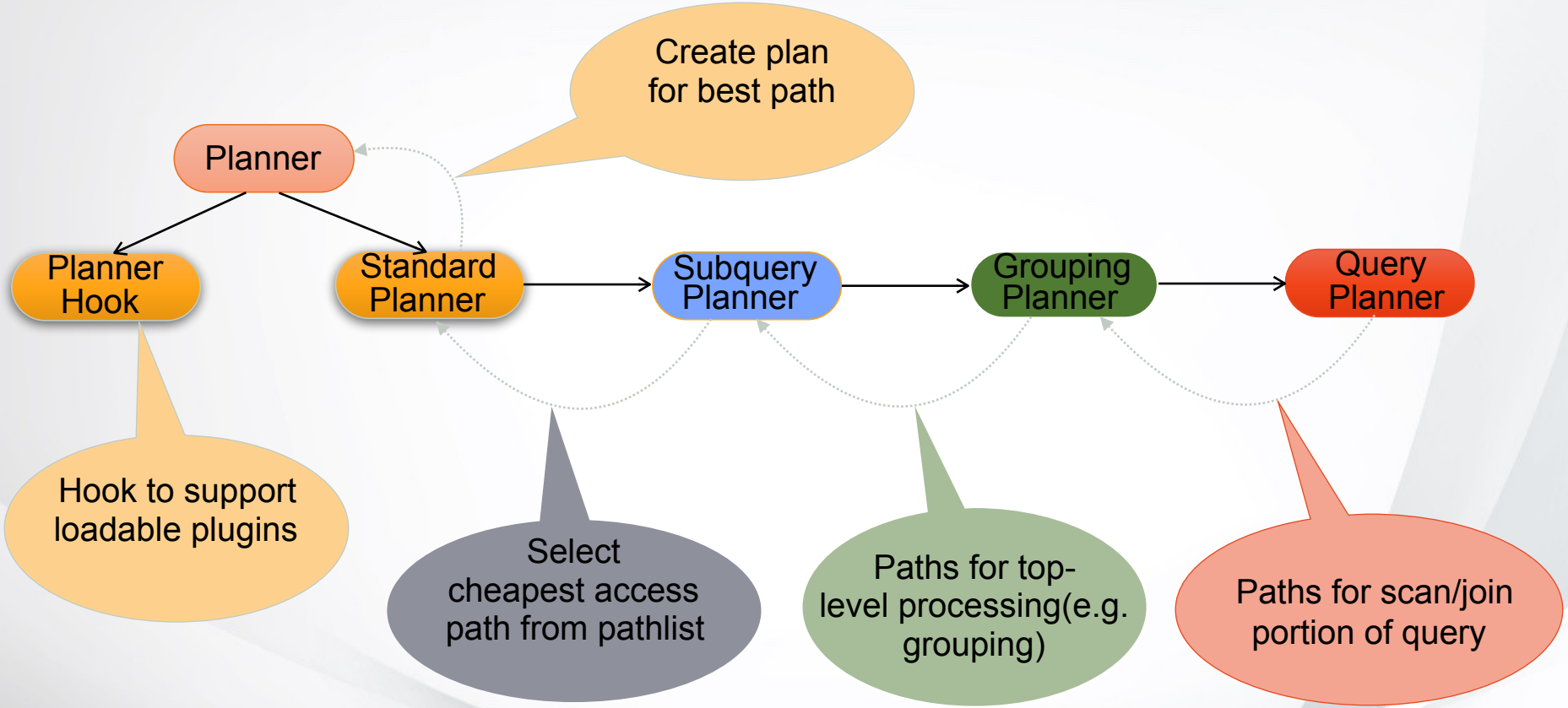
Finding cheapest access path:

1. Create *RelOptInfo* structure
2. Paths for the scan/join portion of this Query

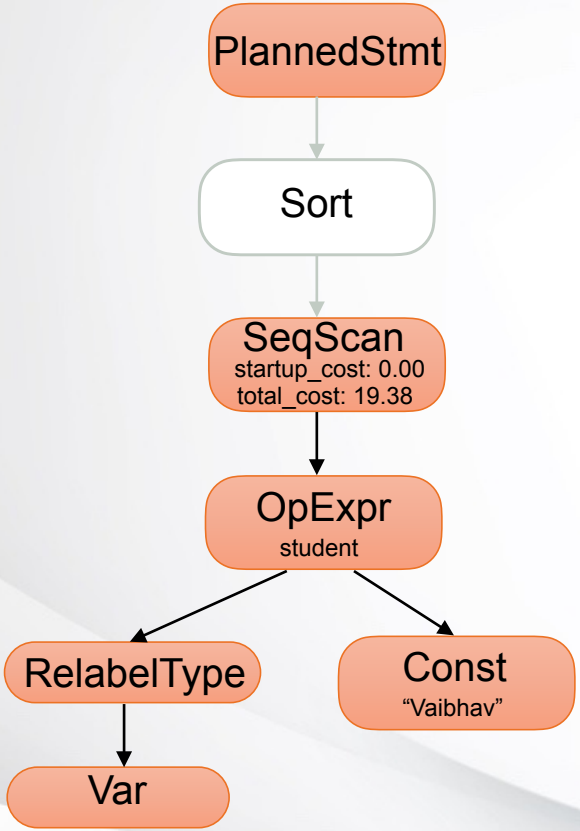
    This processing is as follows:

    1. sequential scan
    2. index access paths
    3. bitmap scan paths
3. Create and add path for top-level processing
4. Cheapest access path

Flow of planning:



Plan tree:

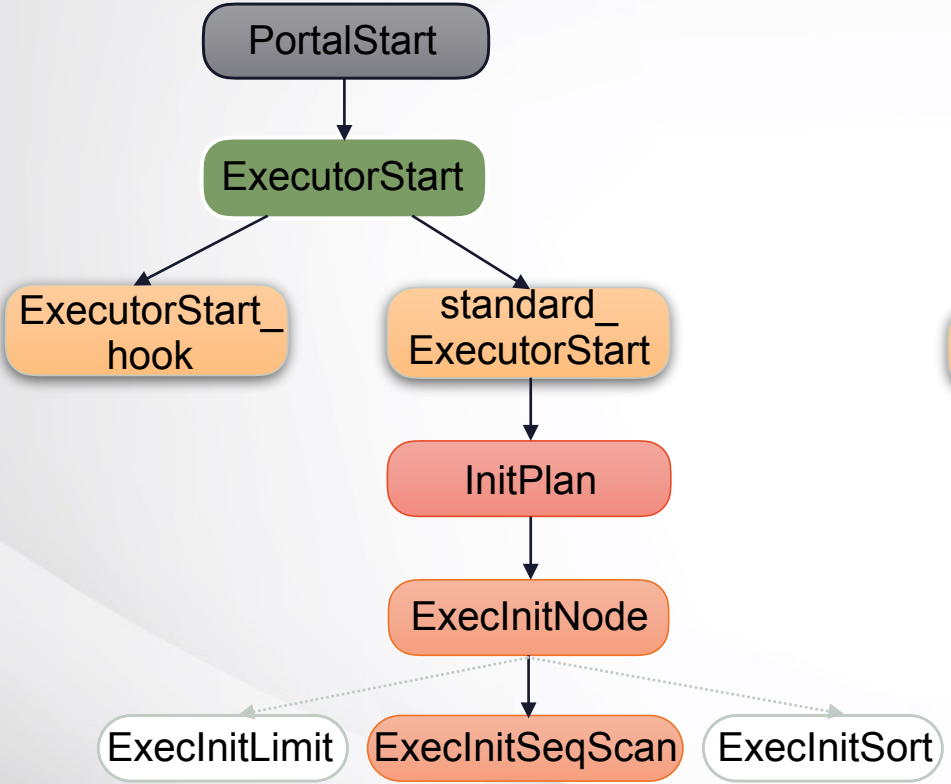


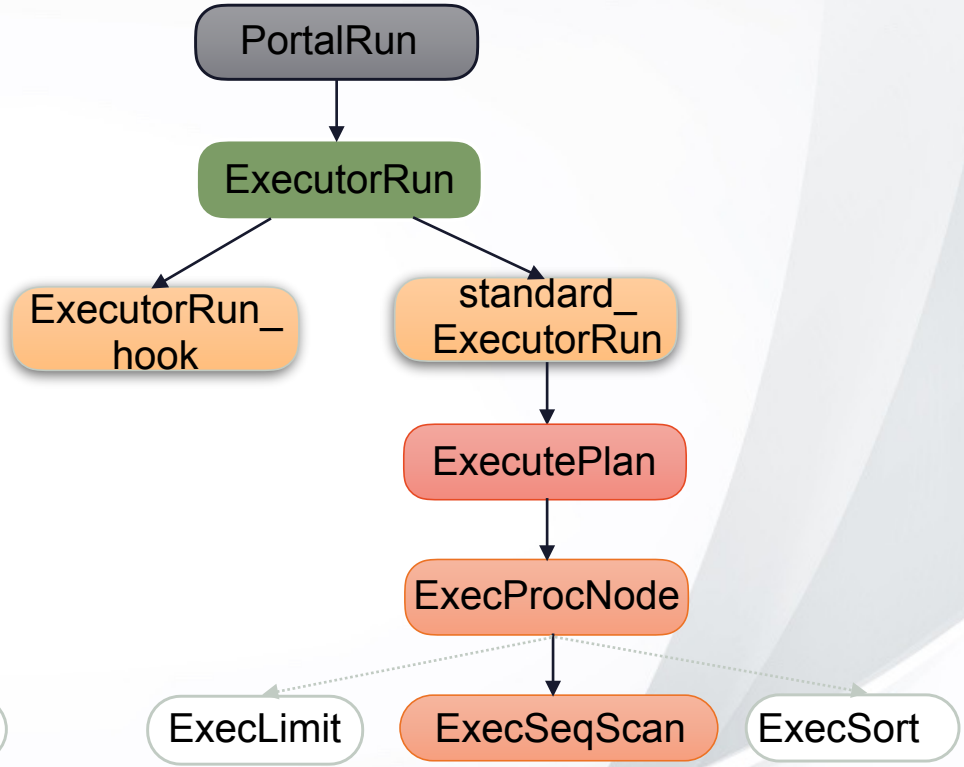

```
#explain(verbose) select * from student where name = 'Vaibhav';
                              QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on s1.student  (cost=0.00..19.38 rows=4 width=80)
   Output: id, name, class
   Filter: (student.name = 'Vaibhav'::bpchar)
```

Execute:

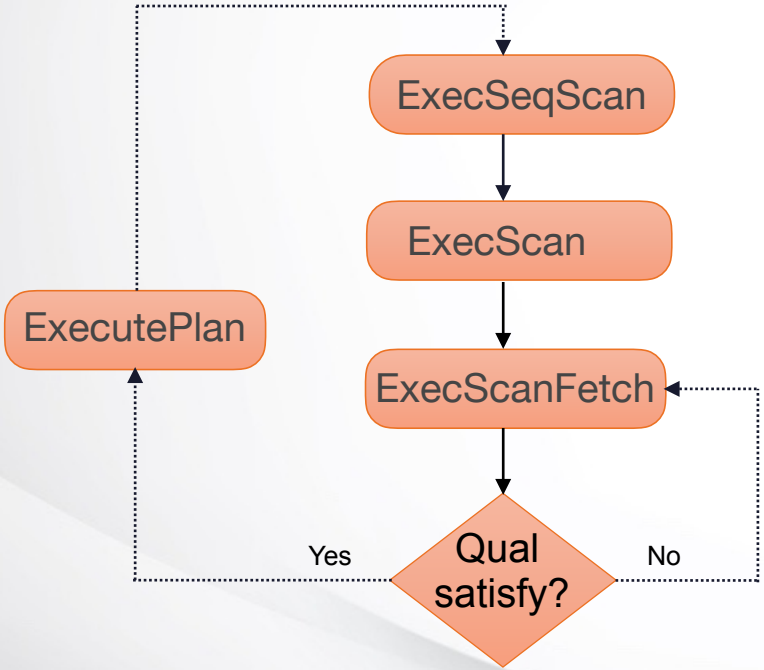
```
/*
 * exec_simple_query
 *
 * Execute a "simple Query" protocol message.
 */
static void
exec_simple_query(const char *query_string)
{
    .
    .
    portal = CreatePortal("", true, true);
    .
    .
    PortalDefineQuery(portal, NULL, query_string, commandTag, plantree_list, NULL);
    .
    .
    PortalStart(portal, NULL, 0, InvalidSnapshot);
    .
    .
    (void) PortalRun(portal, FETCH_ALL, true, true, receiver, receiver, completionTag);
    .
    .
    PortalDrop(portal, false);
    .
    .
}
```

Executor Flow





ExecSeqScan node



SELECT *
FROM student
WHERE name = 'Vaibhav'

Roshan
Karan
Vaibhav
Sahil
Rohan

Explain Analyze

```
#explain analyze select * from student where name = 'Vaibhav';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on student  (cost=0.00..19.38 rows=4 width=80) (actual time=0.021..0.022 rows=1 loops=1)
   Filter: (name = 'Vaibhav'::bpchar)
   Rows Removed by Filter: 2
 Planning Time: 0.100 ms
 Execution Time: 0.050 ms
```

References:

- PostgreSQL 12 official documentation
- PostgreSQL Internals Through Pictures by Bruce Momjian
- The Internals of PostgreSQL by Hironobu SUZUKI

QUESTIONS & DISCUSSION

EDB POSTGRES

THANK YOU

info@enterprisedb.com
www.enterprisedb.com

EDB POSTGRES

ARE YOU AWESOME?

WE'RE HIRING

Connect with us at EDB Booth

Department	Position Title	Location
Product Development	Database Internals	Pune/Bangalore
Product Development	Java/Kafka/JDBC	Pune
Product Development	Devops - Build & Release	Pune
Product Development	Scrum Master	Pune
Product Development	Technical Architect - Java	Pune
Technical Services	Oracle/Postgres DBA's	Pune
Professional Services	Sales Engineer - Oracle/Postgres	Remote
IT	Drupal	Pune
Sales	AE - Sales	Remote
Customer Success Technical	Customer Success Specialist	Remote