Query Processing in PostgreSQL

Amit Langote, EDB

Dec 8 2021

EDB

Agenda

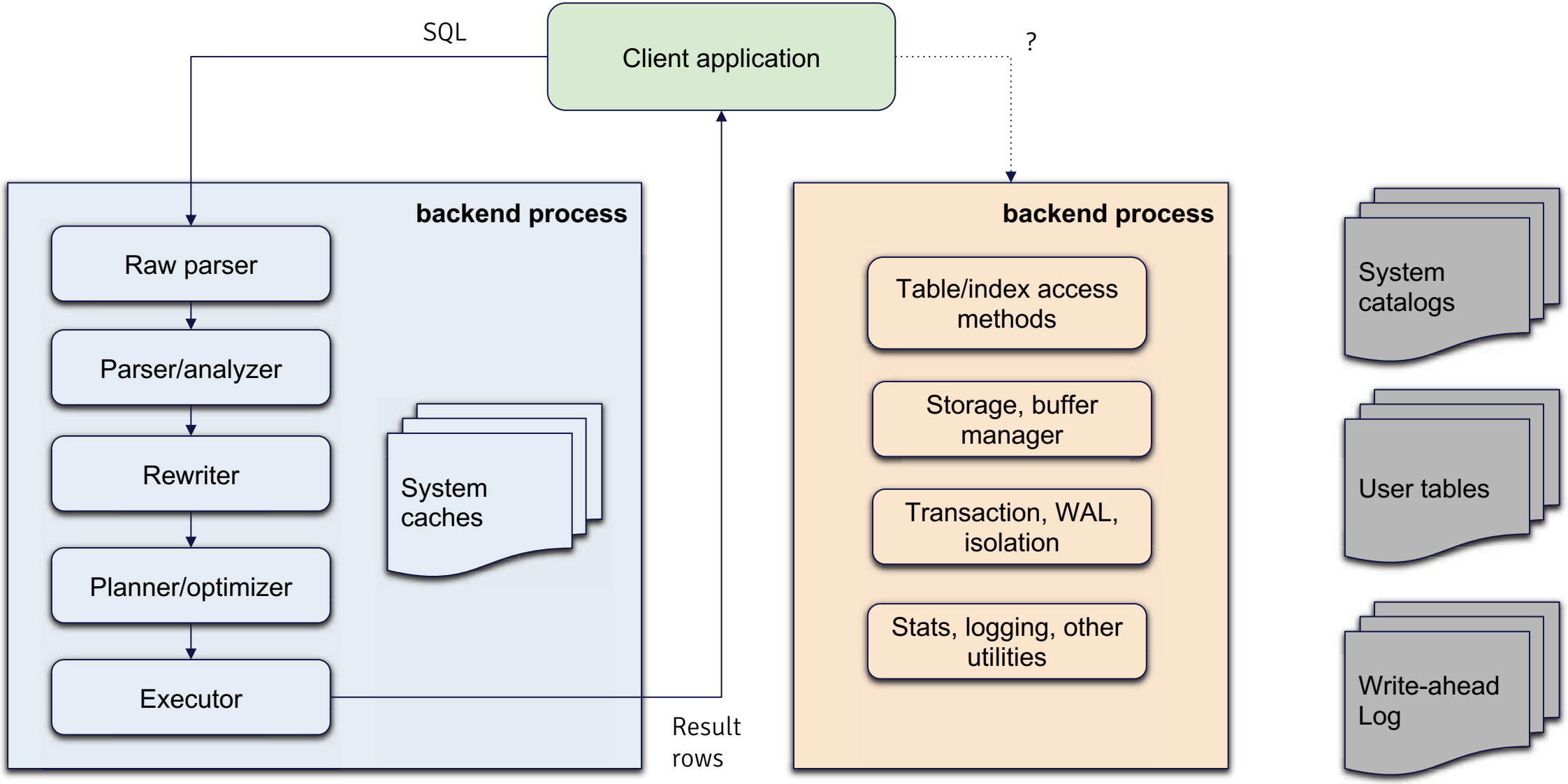
- Overview
- An example query
- Extensibility

Overview

Architecture

- Client-server model with a Postgres-specific wire protocol to exchange formatted messages
  - https://www.postgresql.org/docs/current/protocol.html
- Server accepts **SQL** commands as text strings from an authenticated client and returns rows of data in binary or text format as result

Architecture



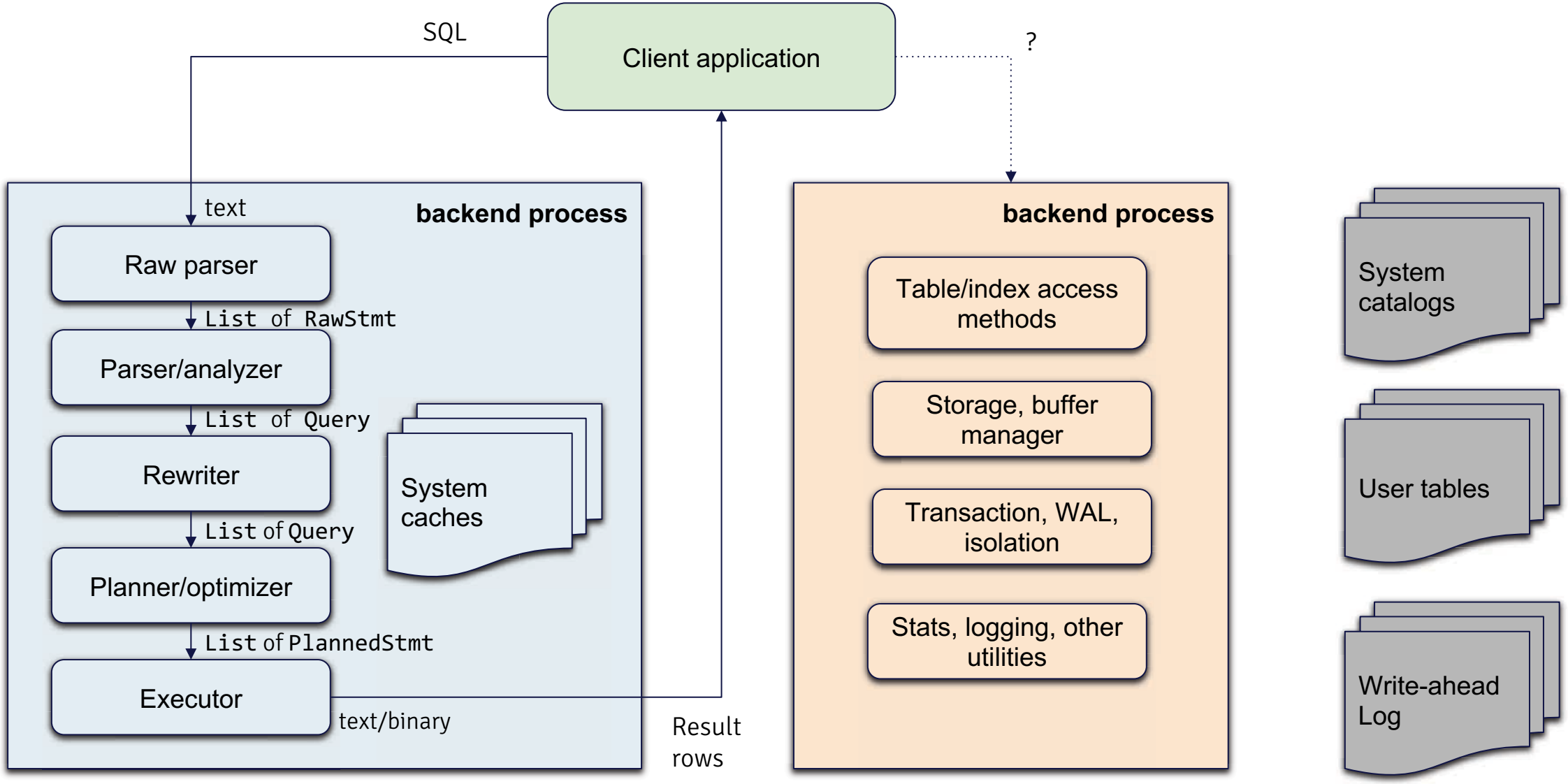
Architecture

- Raw parser
  - Using scanner and parser generated using GNU tools flex, bison, respectively
  - Product: a `List` of `RawStmt`, the raw parse tree
- Parse/analyze
  - Semantic analysis of raw parse tree: mapping object names to OIDs in the catalog, column names to attribute numbers, etc.
  - Product: a `List` of `Query`, the query tree
- Rewrite
  - Expand views, rules
  - Product: a `List` of `Query`, possibly containing multiple query trees

Architecture

- Planner/optimizer
  - Create an optimal plan to execute the queries
  - Product: a `List` of `PlannedStmt`, each containing the plan tree
- Executor
  - Initialize and execute the plan tree
  - Product: result rows delivered to the client over the wire in text/binary format

Architecture



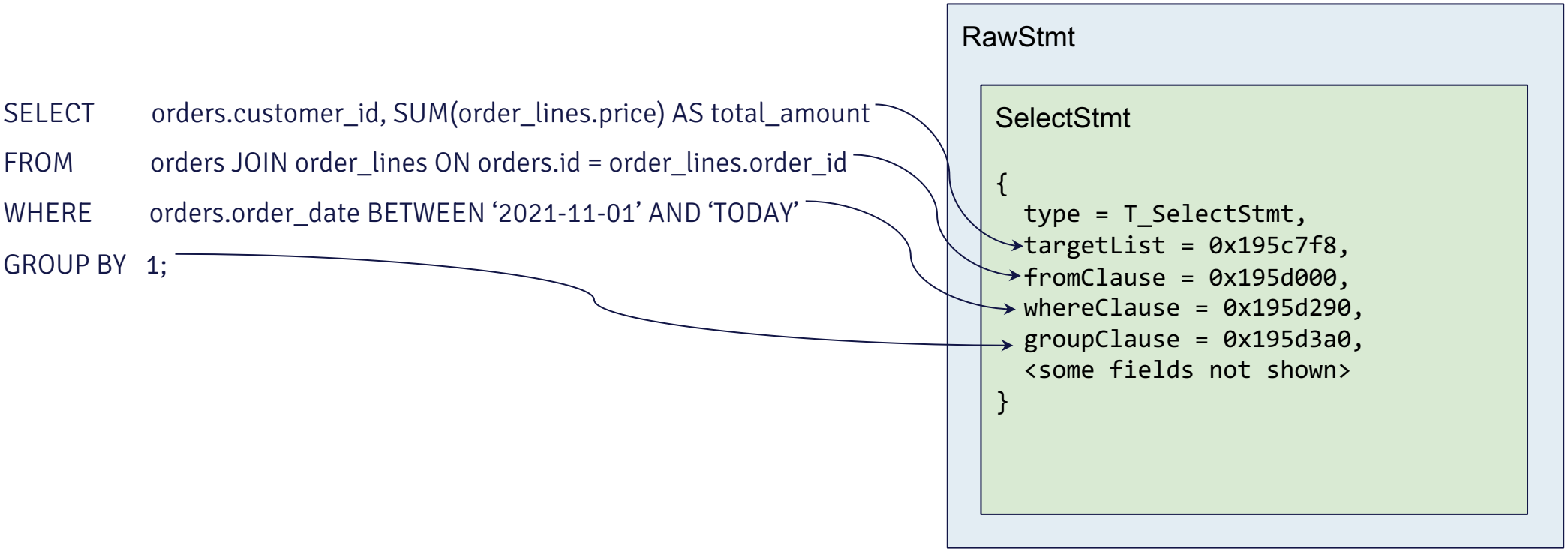
An example query

The query

```
SELECT      orders.customer_id, SUM(order_lines.price) AS total_amount
FROM        orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE       orders.order_date BETWEEN ‘2021-11-01’ AND ‘TODAY’
GROUP BY    1;
```

Raw Parser

- Converts the query string into `RawStmt`, the AST (Abstract Syntax Tree) form.
- No on-disk state is referenced in the process, so no locks are yet taken.



Raw Parser

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

SelectStmt

```
{
  type = T_SelectStmt,
  targetList = 0x195c7f8,
  fromClause = 0x195d000,
  whereClause = 0x195d290,
  groupClause = 0x195d3a0,
  <some fields not shown>
}
```

```
targetList (
   {RESTARGET
   :name <>
   :indirection <>
   :val
     {COLUMNREF
     :fields ("orders" "customer_id")
     :location 7
     }
   :location 7
   }
   {RESTARGET
   :name total_amount
   :indirection <>
   :val
     {FUNCCALL
     :funcname ("sum")
     :args (
       {COLUMNREF
       :fields ("order_lines" "price")
       :location 31
       }
     )
     :agg_order <>
     :agg_filter <>
     :over <>
     :agg_within_group false
     :agg_star false
     :agg_distinct false
     :func_variadic false
     :funcformat 0
     :location 27
     }
   :location 27
   }
)
```

Raw Parser

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id =
order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

SelectStmt

```
{
  type = T_SelectStmt,
  targetList = 0x195c7f8,
  fromClause = 0x195d000,
  whereClause = 0x195d290,
  groupClause = 0x195d3a0,
  <some fields not shown>
}
```

```
fromClause (
        {JOINEXPR
        :jointype 0
        :isNatural false
        :larg
          {RANGEVAR
          :schemaname <>
          :relname orders
          :inh true
          :relpersistence p
          :alias <>
          :location 71
          }
        :rarg
          {RANGEVAR
          :schemaname <>
          :relname order_lines
          :inh true
          :relpersistence p
          :alias <>
          :location 83
          }
        :usingClause <>
        :join_using_alias <>
        :quals
          {AEXPR
          :name ("=")
          :lexpr
            {COLUMNREF
            :fields ("orders" "id")
            :location 99
            }
          :rexpr
            {COLUMNREF
            :fields ("order_lines" "order_id")
            :location 111
            }
          :location 109
          }
        :alias <>
        :rtindex 0
        }
    )
```

Raw Parser

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

SelectStmt

```
{
  type = T_SelectStmt,
  targetList = 0x195c7f8,
  fromClause = 0x195d000,
  whereClause = 0x195d290,
  groupClause = 0x195d3a0,
  <some fields not shown>
}
```

```
whereClause
   {AEXPR BETWEEN
   :name ("BETWEEN")
   :lexpr
     {COLUMNREF
     :fields ("orders" "order_date")
     :location 138
     }
   :rexpr (
     {A_CONST
     :val "¥2021-11-01"
     :location 164
     }
     {A_CONST
     :val "TODAY"
     :location 181
     }
   )
   :location 156
   }
```

```
 :groupClause (
   {A_CONST
   :val 1
   :location 198
   }
 )
```

Parse Analyze

- Converts the `SelectStmt` into `Query,` a generic container for executable statements, containing information about the objects mentioned in the query that is stored in the system catalog
- Locks are taken on the tables

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

Query

```
{
  type = T_Query,
  commandType = CMD_SELECT,
  utilityStmt = 0x0,
  resultRelation = 0,
  rtable = 0x1a29120,
  jointree = 0x1a43d00,
  targetList = 0x1a2ac40,
  groupClause = 0x1a43c80,
  <some fields not shown>
}
```

Parse Analyze

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

Query

```
{
  type = T_Query,
  commandType = CMD_SELECT,
  utilityStmt = 0x0,
  resultRelation = 0,
  rtable = 0x1a29120,
  jointree = 0x1a43d00,
  targetList = 0x1a2ac40,
  groupClause = 0x1a43c80,
  <some fields not shown>
}
```

```
rtable (
      {RANGETBLENTRY
      :alias <>
      :eref
         {ALIAS
         :aliasname orders
         :colnames ("id" "customer_id" "order_date")
         }
      :rtekind 0
      :relid 16384
      }
      {RANGETBLENTRY
      :alias <>
      :eref
         {ALIAS
         :aliasname order_lines
         :colnames ("id" "order_id" "item_id" "price")
         }
      :rtekind 0
      :relid 16389
      }
      {RANGETBLENTRY
      :alias <>
      :eref
         {ALIAS
         :aliasname unnamed_join
         :colnames ("id" "customer_id" "order_date" "id" "order_id"
"item_id" "price")
         }
      :rtekind 2
      :jointype 0
      :joinmergedcols 0
      :joinaliasvars (...)
```

Parse Analyze

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

Query

```
{
  type = T_Query,
  commandType = CMD_SELECT,
  utilityStmt = 0x0,
  resultRelation = 0,
  rtable = 0x1a29120,
  jointree = 0x1a43d00,
  targetList = 0x1a2ac40,
  groupClause = 0x1a43c80,
  <some fields not shown>
}
```

```
targetList (
      {TARGETENTRY
      :expr
         {VAR
         :varno 1
         :varattno 2
         :vartype 23
         }
      :resno 1
      :resname customer_id
      :resjunk false
      }
      {TARGETENTRY
      :expr
         {AGGREF
         :aggfnoid 2110
         :aggtype 700
         :aggcollid 0
         :aggtranstype 0
         :aggargtypes (o 700)
         :args (
            {TARGETENTRY
            :expr
               {VAR
               :varno 2
               :varattno 4
               :vartype 700
               }
            :resno 1
            :resname <>
            :resjunk false
            }
         )
         :aggorder <>
         :aggdistinct <>
         }
      :resno 2
      :resname total_amount
      :resjunk false
      }
   )
```

varno 1: relation “orders”
varattno 2: column 2 of “orders”

varno 2: relation “order_lines”
varattno 4: column 4 of “order_lines”

Parse Analyze

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id =
order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

Query

```
{
  type = T_Query,
  commandType = CMD_SELECT,
  utilityStmt = 0x0,
  resultRelation = 0,
  rtable = 0x1a29120,
  jointree = 0x1a43d00,
  targetList = 0x1a2ac40,
  groupClause = 0x1a43c80,
  <some fields not shown>
}
```

```
jointree
      {FROMEXPR
      :fromlist (
         {JOINEXPR
         :jointype 0
         :isNatural false
         :larg
            {RANGETBLREF
            :rtindex 1
            }
         :rarg
            {RANGETBLREF
            :rtindex 2
            }
         :usingClause <>
         :join_using_alias <>
         :quals
            {OPEXPR
            :opno 96
            :opfuncid 65
            :opresulttype 16
            :args (
               {VAR
               :varno 1
               :varattno 1
               :vartype 23
               }
               {VAR
               :varno 2
               :varattno 2
               :vartype 23
               }
            )
            :location 109
            }
         :alias <>
         :rtindex 3
         }
      )
      :quals
      ...
```

range table relation 1: relation “orders”

range table relation 2: relation “order_lines”

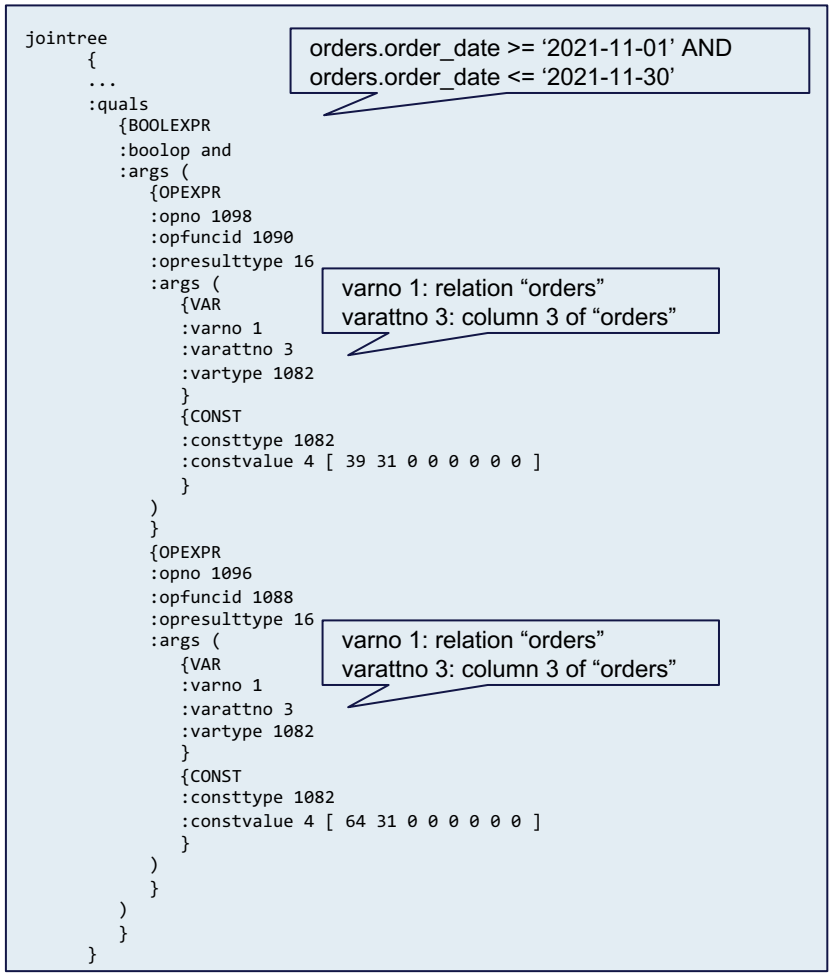
range table relation 3: relation “order” JOIN “order_lines”

Parse Analyze

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

Query

```
{
  type = T_Query,
  commandType = CMD_SELECT,
  utilityStmt = 0x0,
  resultRelation = 0,
  rtable = 0x1a29120,
  jointree = 0x1a43d00,
  targetList = 0x1a2ac40,
  groupClause = 0x1a43c80,
  <some fields not shown>
}
```



Parse Analyze

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

Query

```
{
  type = T_Query,
  commandType = CMD_SELECT,
  utilityStmt = 0x0,
  resultRelation = 0,
  rtable = 0x1a29120,
  jointree = 0x1a43d00,
  targetList = 0x1a2ac40,
  groupClause = 0x1a43c80,
  <some fields not shown>
}
```

```
:groupClause (
   {SORTGROUPCLAUSE
   :tleSortGroupRef 1
   :eqop 96
   :sortop 97
   :nulls_first false
   :hashable true
   }
)
```

tleSortGroupRef 1: group by 1st element of targetlist

Rewrite

- Nothing interesting happens for this query, because there's no view referenced in the query.
- If one of the relations in the query were a view, the rewrite step would add its query to the range table, which the planner then integrates into the main query.

Planner

- Comes up with an optimal plan for the query and puts that into a `PlannedStmt`
- Looks up more information about the objects
  - A table's file size, statistics, partitions, indexes, foreign keys, etc.
- All of the working state is maintained in a `PlannerInfo`

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

PlannerInfo

```
{
  type = T_PlannerInfo,
  parse = 0x195d6e0,
  glob = 0x1a431a0,
  simple_rel_array = 0x0,
  simple_rel_array_size = 0,
  simple_rte_array = 0x0,
  all_baserels = 0x0,
  join_rel_list = 0x0,
  join_rel_hash = 0x0,
  eq_classes = 0x0,
  query_pathkeys = 0x0,
  group_pathkeys = 0x0,
  upper_rels = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  upper_targets = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  processed_tlist = 0x0,
  planner_cxt = 0x195ba40,
  total_table_pages = 0,
  <some fields not shown>
}
```

The Query node

Planner: Pre-processing

- Initial steps, performed after entering the function `subquery_planner()`, involve various simplifications of the query's expressions, like:
  - "pulling up" subqueries into the main query
  - Algebraic simplifications of expressions
    - "col + 0" -> "col"

```
SELECT      orders.customer_id, SUM(order_lines.price) AS total_amount
FROM        orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE       orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY    1;
```

PlannerInfo

Pre-processed Query node

```
{
  type = T_PlannerInfo,
  parse = 0x195d6e0,
  glob = 0x1a431a0,
  simple_rel_array = 0x0,
  simple_rel_array_size = 0,
  simple_rte_array = 0x0,
  all_baserels = 0x0,
  join_rel_list = 0x0,
  join_rel_hash = 0x0,
  eq_classes = 0x0,
  fkey_list = 0x0,
  query_pathkeys = 0x0,
  group_pathkeys = 0x0,
  initial_rels = 0x0,
  upper_rels = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  upper_targets = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  processed_tlist = 0x0,
  planner_cxt = 0x195ba40,
  total_table_pages = 0,
  <some fields not shown>
}
```

Planner: Scan/Join planning

- Actual planning starts after entering the function `grouping_planner()`, which does:
  - `query_planner()` , which creates scan/join `Path`s for the base relations and joins, respectively, covering the `FROM` and `WHERE` clauses. Scan planning considers whether or not use an index. Join planning uses a "dynamic programming" algorithm to incrementally build up the final join relation. It considers nested loop, hash, and merge join algorithm for each join relation at each stage of the algorithm.
  - `RelOptInfo` nodes are set up for relations (base and join) to store catalog info, paths, etc. `EquivalenceClass` and `PathKey` nodes are built for columns and expressions, shared across relations.

```
SELECT      orders.customer_id, SUM(order_lines.price) AS total_amount
FROM        orders JOIN order_lines ON orders.id =
order_lines.order_id
WHERE       orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

PlannerInfo

```
{
  type = T_PlannerInfo,
  parse = 0x195d6e0,
  glob = 0x1a431a0,
  simple_rel_array = 0x1a58fc0,
  simple_rel_array_size = 4,
  simple_rte_array = 0x1a58ff8,
  all_baserels = 0x1a5ab80,
  join_rel_list = 0x1a5cb00,
  join_rel_hash = 0x0,
  join_cur_level = 2,
  eq_classes = 0x1a5a280,
  fkey_list = 0x1a5ab28,
  query_pathkeys = 0x1a5aa50,
  group_pathkeys = 0x1a5aa50,
  initial_rels = 0x1a5c4f8,
  upper_rels = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  upper_targets = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
  processed_tlist = 0x1a439a8,
  planner_cxt = 0x195ba40,
  total_table_pages = 0,
  <some fields not shown>
}
```

Planner: GROUP BY planning

- Actual planning starts after entering the function `grouping_planner()`, which does:
  - Finally back in `grouping_planner()`, create `Paths` for `GROUP BY`, `ORDER BY`, aggregation steps to produce “upper rels”, which have their own `RelOptInfo` nodes. It considers hash or sort based grouping/aggregation paths.

PlannerInfo

```
{
  type = T_PlannerInfo,
  parse = 0x195d6e0,
  glob = 0x1a431a0,
  simple_rel_array = 0x1a58fc0,
  simple_rel_array_size = 4,
  simple_rte_array = 0x1a58ff8,
  all_baserels = 0x1a5ab80,
  join_rel_list = 0x1a5cb00,
  join_rel_hash = 0x0,
  join_cur_level = 2,
  eq_classes = 0x1a5a280,
  fkey_list = 0x1a5ab28,
  query_pathkeys = 0x1a5aa50,
  group_pathkeys = 0x1a5aa50,
  initial_rels = 0x1a5c4f8,
  upper_rels = {0x0, 0x0, 0x1a58178, 0x0, 0x0, 0x0, 0x0, 0x1a587b8},
  upper_targets = {0x0, 0x0, 0x1a57cf8, 0x1a57cf8, 0x1a57cf8,
0x1a57cf8, 0x1a57cf8, 0x1a57cf8},
  processed_tlist = 0x1a439a8,
  planner_cxt = 0x195ba40,
  total_table_pages = 0,
  <some fields not shown>
}
```

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id =
order_lines.order_id
WHERE     orders.order_date BETWEEN ‘2021-11-01’ AND ‘TODAY’
GROUP BY  1;
```

Planner: Path

- A `Path` is a plan-time representation of a plan node that is used to compare alternative implementations to perform a particular execution task, such as scanning a relation or joining two relations
- Planner creates multiple `Paths` for any given relation and selects one to convert into the `Plan`

```
Path
{
  type = T_IndexPath,
  pathtype = T_IndexScan,
  parent = 0x1a44780,
  pathtarget = 0x1a449c0,
  param_info = 0x1a56228,
  parallel_aware = false,
  parallel_safe = true,
  parallel_workers = 0,
  rows = 1,
  startup_cost = 0.1525,
  total_cost = 0.19878378378378381,
  pathkeys = 0x1a55b28
}
```

```
IndexPath
{
  path = {
      <same as shown above>
  },
  indexinfo = 0x195d5c8,
  indexclauses = 0x1a55a50,
  indexorderbys = 0x0,
  indexorderbycols = 0x0,
  indexscandir = ForwardScanDirection,
  indextotalcost = 0.16216216216216217,
  indexselectivity = 0.00049019607843137254
}
```

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN '2021-11-01' AND 'TODAY'
GROUP BY  1;
```

Planner: Plan

- Once the `Paths` for all processing steps have been considered and a “best” path chosen for each step, the best `Path` tree is converted into a `Plan` tree.
  - A `Plan` tree must contain all the information that will be needed when actually executing the plan, while throwing away anything that was only needed during the planning process
- `create_plan()` does this.

```
SELECT    orders.customer_id, SUM(order_lines.price) AS total_amount
FROM      orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE     orders.order_date BETWEEN ‘2021-11-01’ AND ‘TODAY’
GROUP BY  1;
```

Plan

```
{
  type = T_HashJoin,
  startup_cost = 40.724999999999994,
  total_cost = 74.09237499999999,
  plan_rows = 9,
  plan_width = 8,
  parallel_aware = false,
  parallel_safe = true,
  async_capable = false,
  plan_node_id = 0,
  targetlist = 0x1a58918,
  qual = 0x0,
  lefttree = 0x1a58458,
  righttree = 0x1a592a0,
  initPlan = 0x0,
  extParam = 0x0,
  allParam = 0x0
}
```

HashJoin

```
{
  join = {
    plan = {
      <same as shown above>
    },
    jointype = JOIN_INNER,
    inner_unique = true,
    joinqual = 0x0
  },
  hashclauses = 0x1a590e8,
  hashoperators = 0x1a59140,
  hashcollations = 0x1a59198,
  hashkeys = 0x1a591f0
}
```

Planner: PlannedStmt

- The final product of the planning process
  - Contains the `Plan` tree and other global information about the query environment.

PlannedStmt

```
{
  type = T_PlannedStmt,
  commandType = CMD_SELECT,
  queryId = 0,
  hasReturning = false,
  hasModifyingCTE = false,
  canSetTag = true,
  transientPlan = false,
  dependsOnRole = false,
  parallelModeNeeded = false,
  jitFlags = 0,
  planTree = 0x1a59638,
  rtable = 0x1a59868,
  resultRelations = 0x0,
  appendRelations = 0x0,
  subplans = 0x0,
  rewindPlanIDs = 0x0,
  rowMarks = 0x0,
  relationOids = 0x1a598c0,
  invalItems = 0x0,
  paramExecTypes = 0x0,
  utilityStmt = 0x0,
  stmt_location = 0,
  stmt_len = 199
}
```

Planner: EXPLAIN

```
SELECT      orders.customer_id, SUM(order_lines.price) AS total_amount
FROM        orders JOIN order_lines ON orders.id = order_lines.order_id
WHERE       orders.order_date BETWEEN ‘2021-11-01’ AND ‘TODAY’
GROUP BY    1;
```

```
                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=74.24..74.39 rows=9 width=8)
   Output: orders.customer_id, sum(order_lines.price)
   Group Key: orders.customer_id
   ->  Sort  (cost=74.24..74.26 rows=9 width=8)
         Output: orders.customer_id, order_lines.price
         Sort Key: orders.customer_id
         ->  Hash Join  (cost=40.72..74.09 rows=9 width=8)
               Output: orders.customer_id, order_lines.price
               Inner Unique: true
               Hash Cond: (order_lines.order_id = orders.id)
               ->  Seq Scan on public.order_lines  (cost=0.00..28.50 rows=1850 width=8)
                     Output: order_lines.id, order_lines.order_id, order_lines.item_id, order_lines.price
               ->  Hash  (cost=40.60..40.60 rows=10 width=8)
                     Output: orders.customer_id, orders.id
                     ->  Seq Scan on public.orders  (cost=0.00..40.60 rows=10 width=8)
                           Output: orders.customer_id, orders.id
                           Filter: ((orders.order_date >= '2021-11-01'::date) AND (orders.order_date <= '2021-11-29'::date))
(17 rows)
```

Execution

- Recursively processing the `Plan` tree to output result rows
  - Processing follows a demand-pull pipeline mechanism starting at the top.
  - On-disk rows enter through scan nodes at the bottom/leaf.

```
                                                          QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=74.24..74.39 rows=9 width=8)
   Output: orders.customer_id, sum(order_lines.price)
   Group Key: orders.customer_id
   ->  Sort  (cost=74.24..74.26 rows=9 width=8)
         Output: orders.customer_id, order_lines.price
         Sort Key: orders.customer_id
         ->  Hash Join  (cost=40.72..74.09 rows=9 width=8)
               Output: orders.customer_id, order_lines.price
               Inner Unique: true
               Hash Cond: (order_lines.order_id = orders.id)
               ->  Seq Scan on public.order_lines  (cost=0.00..28.50 rows=1850 width=8)
                     Output: order_lines.id, order_lines.order_id, order_lines.item_id, order_lines.price
               ->  Hash  (cost=40.60..40.60 rows=10 width=8)
                     Output: orders.customer_id, orders.id
                     ->  Seq Scan on public.orders  (cost=0.00..40.60 rows=10 width=8)
                           Output: orders.customer_id, orders.id
                           Filter: ((orders.order_date >= '2021-11-01'::date) AND (orders.order_date <= '2021-11-29'::date))
(17 rows)
```

Execution: InitPlan()

- Before the actual execution starts, the `Plan` tree is “walked” to create a `PlanState` node for each `Plan` node in the tree

PlanState

```
{
  type = T_HashJoinState,
  plan = 0x1a54f00,
  state = 0x1a48c40,
  ExecProcNode = 0x7ffbd0 <ExecProcNodeFirst>,
  ExecProcNodeReal = 0x82dc10 <ExecHashJoin>,
  instrument = 0x0,
  worker_instrument = 0x0,
  worker_jit_instrument = 0x0,
  qual = 0x0,
  lefttree = 0x1a499b0,
  righttree = 0x1a49ee8,
  initPlan = 0x0,
  subPlan = 0x0,
  chgParam = 0x0,
  ps_ResultTupleDesc = 0x1a5ac40,
  ps_ResultTupleSlot = 0x1a5ad58,
  ps_ExprContext = 0x1a49918,
  ps_ProjInfo = 0x1a5adf0,
  async_capable = false,
  scandesc = 0x0,
  scanops = 0x0,
  outerops = 0x0,
  innerops = 0x0,
  resultops = 0xe4c458 <TTSOpsVirtual>,
  scanopsset = false,
  outeropsset = false,
  inneropsset = false,
  resultopsset = true
  <some fields not shown>
}
```

HashJoinState

```
{
  js = {
    ps = {
      <same as shown on left>
    },
    jointype = JOIN_INNER,
    single_match = true,
    joinqual = 0x0
  },
  hashclauses = 0x1a7f528,
  hj_OuterHashKeys = 0x1a80738,
  hj_HashOperators = 0x1a56e10,
  hj_Collations = 0x1a56e68,
  hj_HashTable = 0x0,
  hj_CurHashValue = 0,
  hj_CurBucketNo = 0,
  hj_CurSkewBucketNo = -1,
  hj_CurTuple = 0x0,
  hj_OuterTupleSlot = 0x1a7f378,
  hj_HashTupleSlot = 0x1a5a220,
  hj_NullOuterTupleSlot = 0x0,
  hj_NullInnerTupleSlot = 0x0,
  hj_FirstOuterTupleSlot = 0x0,
  hj_JoinState = 1,
  hj_MatchedOuter = false,
  hj_OuterNotEmpty = false

}
```

Execution: ExecutePlan()

- Recursively calls `ExecProcNode()` on the `PlanState` nodes contained in the tree
  - Result rows are bubbled up and the top node's result row is returned as the result of the query

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
GroupAggregate  (cost=74.24..74.39 rows=9 width=8)
  Output: orders.customer_id, sum(order_lines.price)
  Group Key: orders.customer_id
  ->  Sort  (cost=74.24..74.26 rows=9 width=8)
        Output: orders.customer_id, order_lines.price
        Sort Key: orders.customer_id
        ->  Hash Join  (cost=40.72..74.09 rows=9 width=8)
              Output: orders.customer_id, order_lines.price
              Inner Unique: true
              Hash Cond: (order_lines.order_id = orders.id)
              ->  Seq Scan on public.order_lines  (cost=0.00..28.50 rows=1850 width=8)
                    Output: order_lines.id, order_lines.order_id, order_lines.item_id, order_lines.price
              ->  Hash  (cost=40.60..40.60 rows=10 width=8)
                    Output: orders.customer_id, orders.id
                    ->  Seq Scan on public.orders  (cost=0.00..40.60 rows=10 width=8)
                          Output: orders.customer_id, orders.id
                          Filter: ((orders.order_date >= '2021-11-01'::date) AND (orders.order_date <= '2021-11-29'::date))
(17 rows)
```

Execution: Returning Result Rows

- Before `ExecutePlan()` is called, a message describing the result row format is sent to the client, which consists of:
  - Message type (Letter ‘T’ for Tuple Descriptor)
  - Number of attributes as a 16-bit integer
  - For each attribute:
    - Attribute name (as null terminated string)
    - Table OID as 32-bit integer
    - Column number as 16-bit integer,
    - Type information as 3 integers (32-bit type OID, 16-bit type length, 32-bit type modifier)
    - Output format descriptor as 16-bit integer
- For each result row, `ExecutePlan()` sends a message describing the result row format to the client, which consists of:
  - Message type (Letter ‘D’ for Data Row)
  - Number of attributes as a 16-bit integer
  - For each attribute:
    - If null, a 32-bit integer value -1
    - If non-null, the value in the client-requested format
      - By calling the attribute type’s “output” function if the client requested text format
      - By calling the attribute type’s “send” function if the client requested binary format

Extensibility

Foreign Data Wrappers

- Extend Postgres to access non-Postgres data sources as (“foreign”) tables
  - Other relational or non-relational databases, CSV files, Hadoop, Twitter timeline, etc.
- The planner API for handling queries mentioning foreign tables

```
void GetForeignRelSize(PlannerInfo *root, RelOptInfo *baserel, Oid foreigntableid);
void GetForeignPaths(PlannerInfo *root, RelOptInfo *baserel, Oid foreigntableid);
ForeignScan *GetForeignPlan(PlannerInfo *root, RelOptInfo *baserel, Oid foreigntableid,
                            ForeignPath *best_path, List *tlist, List *scan_clauses,
                            Plan *outer_plan);
struct ForeignScan
{
    Scan        scan;
    CmdType     operation;
    Index       resultRelation;
    Oid         fs_server;
    List       *fdw_exprs;
    List       *fdw_private;
    List       *fdw_scan_tlist;
    List       *fdw_recheck_quals;
    Bitmapset  *fs_relids;
    bool        fsSystemCol;
};
```

Foreign Data Wrappers

- The executor API:

```
void BeginForeignScan(ForeignScanState *node, int eflags);

TupleTableSlot *IterateForeignScan(ForeignScanState *node);

void ReScanForeignScan(ForeignScanState *node);

void EndForeignScan(ForeignScanState *node);

struct ForeignScanState

{

    ScanState   ss;

    ExprState  *fdw_recheck_quals;

    Size        pscan_len;

    ResultRelInfo *resultRelInfo;

    struct FdwRoutine *fdwroutine;

    void       *fdw_state;

};
```

- Other APIs for DML queries and advanced stuff like joins, aggregation
  - Join, aggregation APIs allow “push-down” of those operations to the remote side if supported

Custom Scan Providers

- Extend Postgres to make scans/joins to use algorithms not present in the core executor
  - For example, use GPU acceleration for join/aggregate computation
- The planner API consists of the following “hook” functions to insert a scan or join `CustomPath` that the custom scan module must provide:

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root, RelOptInfo *rel, Index rti, RangeTblEntry *rte);
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root, RelOptInfo *joinrel,
                                             RelOptInfo *outerrel, RelOptInfo *innerrel,
                                             JoinType jointype, JoinPathExtraData *extra);

typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List     *custom_paths;
    List     *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

Custom Scan Providers

- The planner API continued: The following function must be provided to convert a `CustomPath` into the executable `Plan` form:

```
Plan *(*PlanCustomPath) (PlannerInfo *root, RelOptInfo *rel, CustomPath *best_path,
                         List *tlist, List *clauses, List *custom_plans);
typedef struct CustomScan
{
    Scan      scan;
    uint32    flags;
    List     *custom_plans;
    List     *custom_exprs;
    List     *custom_private;
    List     *custom_scan_tlist;
    Bitmapset *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;
```

Custom Scan Providers

- The executor API: a function to initialize execution state of a `CustomScan` in `CustomScanState` and a bunch of other support functions that allow the executor to fetch rows using the custom node

```
Node *(*CreateCustomScanState) (CustomScan *cscan);

typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;

void (*BeginCustomScan) (CustomScanState *node, EState *estate, int eflags);
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
void (*EndCustomScan) (CustomScanState *node);
void (*ExplainCustomScan) (CustomScanState *node, List *ancestors, ExplainState *es);
```

Custom Scan Providers

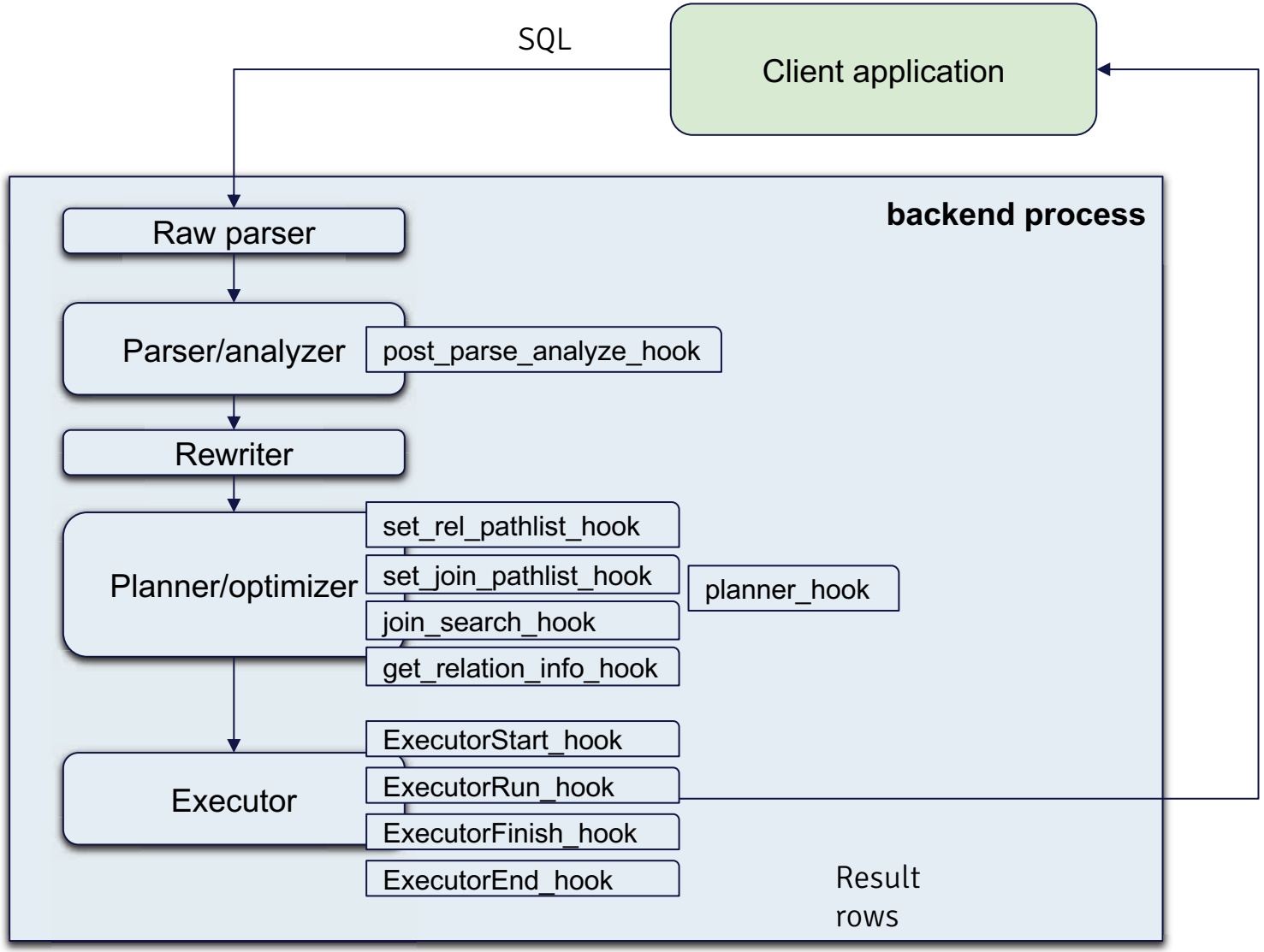
- An example plan containing custom nodes as implemented by PGStrom, a custom scan provider, taken verbatim from https://heterodb.github.io/pg-strom/operations/

```
                                QUERY PLAN
---------------------------------------------------------------------------
 GroupAggregate  (cost=1239991.03..1239995.15 rows=27 width=20)
   Group Key: t0.cat
   ->  Sort  (cost=1239991.03..1239991.50 rows=189 width=44)
         Sort Key: t0.cat
         ->  Custom Scan (GpuPreAgg)  (cost=1239980.10..1239983.88 rows=189 width=44)
               Reduction: Local
               GPU Projection: cat, pgstrom.nrows(), pgstrom.nrows((ax IS NOT NULL)), pgstrom.psum(ax)
               ->  Custom Scan (GpuJoin)  (cost=50776.43..1199522.96 rows=33332245 width=12)
                     GPU Projection: t0.cat, t1.ax
                     Depth 1: GpuHashJoin  (nrows 33332245...33332245)
                              HashKeys: t0.aid
                              JoinQuals: (t0.aid = t1.aid)
                              KDS-Hash (size: 10.39MB)
                     ->  Custom Scan (GpuScan) on t0  (cost=12634.49..1187710.85 rows=33332245 width=8)
                           GPU Projection: cat, aid
                           GPU Filter: (aid < bid)
                     ->  Seq Scan on t1  (cost=0.00..1972.85 rows=103785 width=12)
```

Hooks

- A hook: an interface provided by the core engine to allow user-written C code being called to augment the core functionality
- Postgres has 26 hook points in total as of v14

Hooks



Hooks: examples

- **pg_stat_statements**, which provides a means for tracking planning and execution statistics of all SQL statements executed by a server
  - To do that, it implements the following hooks:
    - `planner_hook`: to measure and store the planning time duration for a given query
    - `ExecutorStart_hook`: to start “instrumentation” for a given query
    - `ExecutorRun_` / `Finish_hook`: to track query “nesting level” of a given query
    - `ExecutorEnd_hook`: to finish “instrumentation” for a given query

Hooks: examples

- **Citus**, which transforms Postgres into a distributed database
  - To do that, it implements the following hooks:
    - `planner_hook`: to plan queries by taking into account that data is distributed across a cluster of Postgres servers
    - `set_rel_pathlist_hook`: to collect information about a table for distributed planning
    - `set_join_pathlist_hook`: to collect information about a join for distributed planning
    - `ExecutorStart_hook`: to set a global flag to allow writes even on hot standby servers
    - `ExecutorRun_hook`: to fix up subplans in a distributed plan before main execution
  - Actually, Citus also seems to rely on `CustomPath`, `CustomScan` constructs to implement distributed planning and execution.

Summary

- Postgres supports processing SQL queries over relational data.
- An SQL query enters the server as a text string, gets parsed, analyzed, planned, and converted into an optimal executable plan, whose execution produces the result rows that are returned to the client.
- The default query processing behavior can be augmented using a number of extension APIs and hook points.

Thank you

References

- A Tour of PostgreSQL Internals (Tom Lane): https://www.postgresql.org/files/developer/tour.pdf
- Bruce Momjian's presentations: https://momjian.us/main/presentations/
- PostgreSQL source code: https://doxygen.postgresql.org/