PostgreSQL Internals

Overview

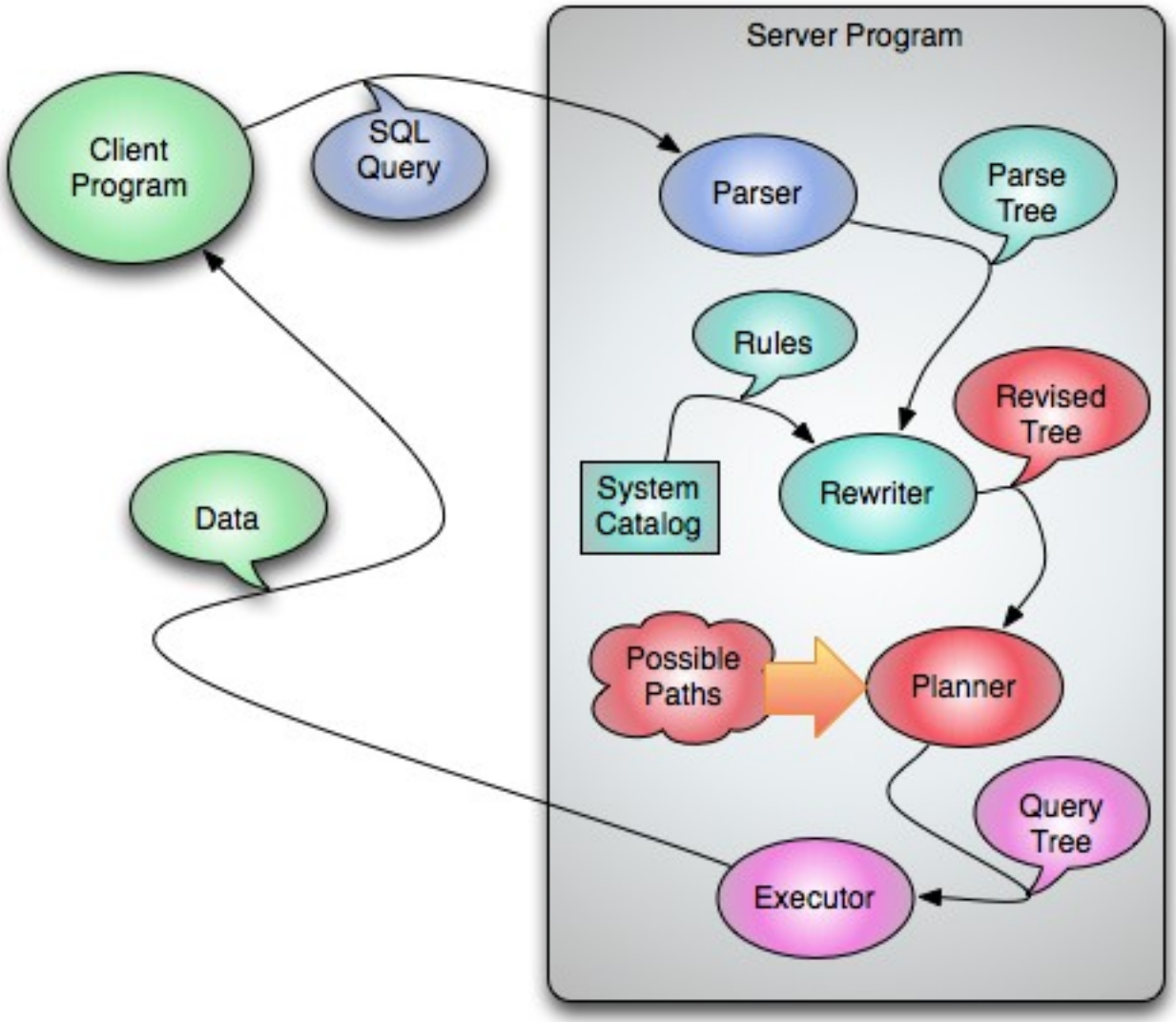
Sheldon E. Strauch

Manager, Database Development
Enova Financial

Path of a Query

- Client program sends SQL query to server
- Server program then:
  - Parses query, producing parse tree
  - Rewrites the tree using rules from the catalog
  - Selects best of many possible execution paths
  - Executes the resulting query tree
  - Returns any data to the client program

Path of a Query



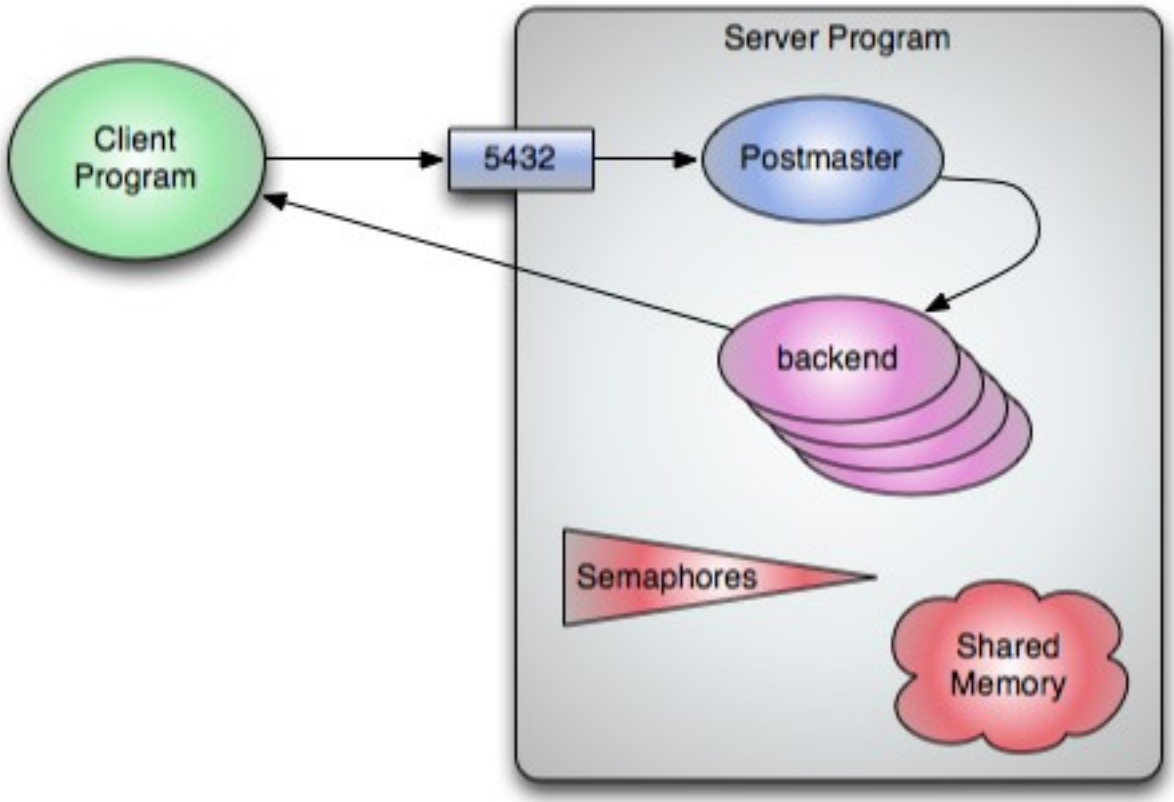
Client Programs

- The classic command line: psql
- Any program understanding the protocol:
  - C-language `libpq` library implements it
  - Java JDBC driver has distinct implementation
- Graphical interface tool: pgadmin3
- Third party:
  - SQL Manager
  - HTSQL
  - Etc.

Establishing Connections

- One server process per client process
- One *postmaster* process to rule them all...
- Postmaster spawns *backend* processes
- Individual server processes:
  - Concurrently access data
  - Must ensure data integrity
  - Communicate using
    - Semaphores
    - Shared memory

Establishing Connections



Communication and Processing

- Queries transmitted as plain-text SQL
- No parsing done in client-side *frontend*
- Server-side *backend*:
  - Creates an *execution plan*
  - Retrieves *tuples* (rows) of data
  - Returns data on established connection

Initial Parse Stage

- Uses fixed rules to validate SQL syntax
- Lexical analyzer `scan.l` written for flex (lex)
  - Recognizes identifiers
  - Fore each, generates a token
- Grammar `gram.y` written for bison (yacc)
  - Instantiates grammar rules
  - Uses code written in C to build parse tree
- Returns parse tree ready for transformation

Transform

- Performs semantic transformation
- Builds up *query tree* from *parse tree*
- Uses system *catalogs* to look up
  - Tables
  - Functions
  - Operators
- Catalog lookups **must** be performed within a transaction

Transform

- Transaction control commands
  - `BEGIN`, `COMMIT`, `ROLLBACK`
  - Are complete on parsing
  - Require no further analysis
  - Allow initiating transaction for catalog lookup
- Output similar to parse tree
  - Substitutes precise info for `FuncCall` nodes
  - Adds data type information to tree

Query Rewrite Rule System

- Major improvement over Berkley rule system
- Interposed between parser and planner
- Converts view nodes into collections of underlying table nodes
- Uses user defined rewrite rules to do so
- Rule: a query tree with additional information
- Output representable as a SQL statement

Query Tree

- Internal representation of a SQL statement
- Each component part stored separately
- Can be logged with configuration parameters:
  - `debug_print_parse`
  - `debug_print_rewritten`
  - `debug_print_plan`
- Rules stored as query trees in `pg_rewrite`

Query Tree Parts

- Command type
- Range table
- Result relation
- Target list
- Qualification
- Join tree

The Command Type

- Simple:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
- Some command types don't use certain query tree parts

The Range Table

- List of *relations* (tables, views) used in query
- For `SELECT`, is basically the `FROM` clause
- Every table entry identifies:
  - A table or view – E.G: customer_sources
  - Its reference name – E.G: cs
- Internally, entries are referenced by number rather than name
- Merging in rule trees can create duplicate references, internal reference by number resolves ambiguity

The Result Relation

- Index into the range table identifying the relation where the results of the query go
- `SELECT` queries don't have a result relation
- Disregard `SELECT INTO` which is a special case of a `CREATE TABLE` followed by an `INSERT INTO`
- For `INSERT`, `UPDATE` and `DELETE`, the table or view that will be changed

The Target List

- List of expressions defining the query result
- For `SELECT` these build the final output
- For `DELETE` a normal target list isn't needed
- For `INSERT` these describe the rows going into the result relation
- For `UPDATE` These describe the new rows replacing the old

The Target List

- Every target list item contains an expression which can be:
  - A constant value
  - A pointer to a range table relation column
  - A parameter
  - An expression tree made of:
    - Function calls
    - Constants
    - Variables
    - Operators

The Qualification

- Is an expression similar to that contained in target list entries
- Boolean result determines if the operation to achieve the final result row is to be executed
- Corresponds to the `WHERE` clause

The Join Tree

- Corresponds to the `FROM` and `WHERE` clauses
- A simple `SELECT` has a list of relations
- If `JOIN` order matters, this also shows the structure of the `JOIN` expressions
- Restrictions on `JOIN` clauses (`ON` or `USING`) present as qualification expressions attached to those join-tree nodes
- The top-level `WHERE` expression is stored as a qualification attached to the top-level join-tree item

Views as Rules

- In postgreSQL, all views are implemented using the rule system
- The representation of a view in the catalog system is identical to that of a table
- To the parser, views and tables are identical
- Views are implemented as `ON SELECT...DO INSTEAD SELECT...` rules where the `INSTEAD` is a `SELECT` on the tables that the view is built on top of

The Planner

- There are many ways to execute a particular query tree
- For example, a SELECT of a column on a table where that column is indexed provides two possible plans:
  - A sequential table scan, fetching every tuple and evaluating the column data
  - An index scan and subsequent tuple fetch

The Planner

- For a small table, a sequential scan is faster because we need load and access only one data page, rather than loading and accessing an index page in order to determine we need to load and access another data page.
- for a large table, not so much...
- The planner examines all the possible execution paths, choosing the most efficient

The Planner

- Sometimes, examining all possible execution paths is computationally or memory prohibitive
- In this case, postgreSQL switches to a genetic query optimizer in order to return a reasonable plan in a reasonable amount of time
- The genetic query optimizer has been a source of considerable debate in the postgreSQL community

The Path

- Query planner data structure is called a *path*
- A path is a stripped-down version of a *plan*
- The plan to be executed is constructed after the cheapest path has been chosen

Relation Scanning Paths

- The first step is to determine paths for scanning each individual relation in the query
- Available indexes indicate possible paths
- A sequential table scan path is always created
- Create an index path if a restriction attribute has an index using an available operator
- Create an Index path for indexes that have a sort ordering that can match the query's `ORDER BY` clause

Join Paths

- The next step is to determine paths for joining relation pairs in the query
- Nested loop join: The right relation is scanned once for each tuple found in the left relation
- Merge join: Each relation is sorted on join attributes before the join starts
- Hash join:
  - The right relation is scanned and loaded into a hash table with join attributes as the hash keys
  - The left relation is scanned, appropriate values of every tuple found are used as hash keys to locate matching tuples

Choosing Paths

- If a query involves more than two relations, the final result must be built up by a tree of join steps, each with two inputs
- Join pairs that have a corresponding `JOIN` in the `WHERE` qualification get preference
- Join pairs with no `JOIN` clause are considered only when there is no other choice

Over the Threshold

- Unless the genetic query optimizer's relation count threshold `geqo_threshold` is exceeded, the planner generates all paths for every join pair, then chooses the cheapest set
- When the `geqo_threshold` is exceeded, heuristics are used to determine which join sequences to consider, otherwise the process is the same

The Chosen Plan

- Base relation index and sequential scans
- Any nested-loop, merge, or hash join nodes
- Any auxiliary steps needed:
  - Sort nodes
  - Aggregate-function calculation nodes

Loose Ends

- Most plan nodes have the ability to:
  - Select: Discard excluded tuples
  - Project: Compute derived column values
- The planner assigns:
  - Selection conditions
  - Output expression computation

  To the most appropriate plan nodes

Executor

- Recursively process plan tree extracting tuples
- One call to a plan node returns one tuple
- Complex queries have numerous plan nodes
- Each node applies selections and projections
- SELECT: Returns data to client
- INSERT: ModifyTable node inserts tuples
- UPDATE: ModifyTable uses tuple ID as target
- DELETE: ModifyTable uses tuple ID to delete