PostgreSQL Functioning and Internals

session by

Nagesh Karmali
Proj Co-ord, Research Fellow
IIT Bombay
email: nags@it.iitb.ac.in
Processing a Query

Establishing connections

**postmaster** - a master process that spawns a new server process called **postgres** on “**process per user**” basis

All postgres processes communicate using **semaphores** and **shared memory** to ensure data integrity throughout concurrent data access
Processing a Query ....contd

- **Parser Stage**

  Two parts:

  a) Lexer (scan.l) and Parser (gram.y) are implemented using well known unix tools lex and yacc

  - `scan.l` is responsible for recognizing identifiers + keywords -> **tokens**

  - `gram.y` consists of a set of grammar rules and actions that are executed whenever a rule is fired -> **parse tree**
Processing a Query ....contd

b) Transformation Process

No semantic is done in the previous process.
The transformation process takes the raw parse tree handed by the parser as input and does the semantic interpretation needed to understand which tables, functions, and operators are referenced by the query and which forms the query tree.

E.g.
- FuncCall node $\rightarrow$ FuncExpr if ordinary
  - $V$
  - $\rightarrow$ Aggref if aggregate
Processing a Query ....contd

The Rule System

Supports a powerful rule system for the specification of views and ambiguous view updates.

The **Query Rewrite Rule** system is totally different from stored procedures and triggers.

It modifies queries (especially the query tree from parser stage) taking rules into consideration and then passes the modified query to the query planner for planning and execution.

So let us have a quick glance at the query tree next --- >
Processing a Query ....contd

The Rule System .... Contd

Query Tree - parts

a) The **command type**
   This is a simple value telling which command (SELECT, INSERT, UPDATE, DELETE) produced the query tree

b) The **range table**
   The range table is a list of relations that are used in the query. In a SELECT statement these are the relations given after the FROM key word.

c) The **result relation**
   This is an index into the range table that identifies the relation where the results of the query go
d) The **target list**

The target list is a list of expressions that define the result of the query.

e) The **qualification**

The qualification is an expression and the result value of this expression is a Boolean that tells whether the operation for the final result row should be executed or not.
f) The join tree

The join tree shows the structure of the JOIN expressions along with restrictions associated with particular JOIN clauses stored as qualification expression attached to those join-tree nodes.

g) The others

For the other parts of the query tree like the ORDER BY clause, the rule system substitutes some entries there while applying rules, but that doesn’t have much to do with the fundamentals of the rule system.
Processing a Query ....contd

**Planner/ Optimizer**

The task of the planner/optimizer is to create an optimal execution plan.

Once the cheapest path is determined, a full-fledged plan tree is built to pass to the executor.

The sequential scan plan is always created, as the possibility is very high.

3 possible join strategies are **nested loop join**, **merge sort join** and **hash join**

plan tree = seq or ind scan of the base rel

+ NL,MJ or HJ nodes as needed

+ sort nodes or agg-func calc. nodes

And additional capability to do selection and projection
Processing a Query ....contd

- **Executor**

  The executor takes the plan handed back by the planner/optimizer and recursively processes it to extract the required set of rows. This is essentially a demand-pull pipeline mechanism.

  The executor mechanism is used to evaluate all four basic SQL query types: SELECT, INSERT, UPDATE, and DELETE
Input:

```
SELECT * FROM tab1, tab2 WHERE tab1.a = tab2.f
```

Output:

Parser example
Views are handled by substitution of subqueries into the parse tree.

For example, if `tab2` were a view then the rewriter would emit something like this:

```
SELECT Query
  Target List
  Join Tree
  Qualification
```

```
<table>
<thead>
<tr>
<th>tab1.a</th>
<th>int4</th>
</tr>
</thead>
<tbody>
<tr>
<td>tab1.b</td>
<td>text</td>
</tr>
<tr>
<td>tab1.c</td>
<td>float8</td>
</tr>
<tr>
<td>tab2.d</td>
<td>text</td>
</tr>
<tr>
<td>tab2.e</td>
<td>text</td>
</tr>
<tr>
<td>tab2.f</td>
<td>int4</td>
</tr>
</tbody>
</table>
```

```
Cross Join
```

```
SELECT Query
  Target List
  Join Tree
  Qualification
```

```
int4 =
```

```
tab1.a
```

```
tab2.f
```

ON INSERT/UPDATE/DELETE rules require more extensive transformations, and may produce multiple queries from a single query.
Query:

```
SELECT SUM(a1)+1 FROM a WHERE a2 < a3
```

Plan tree:

```
Executor top level

Aggregate
  Target List
  Working state(s)
    Running state for SUM(a1)
      SUM result
        +
          1

Sequential Scan
  Table: a
  Target List
    Qualification
      int4 <
        a.a2
        a.a3

Equivalent to SELECT a1 FROM a WHERE a2 < a3
```

Executor example
Query:

```
SELECT DISTINCT a1, b1 FROM a, b WHERE a2 = b2 AND a3 = 42
```

Plan tree:

Executor top level

```
Unique

Sort
Sort Columns: a1, b1

Nestloop Join
Target List: a1, b1
Qualification: a2 = b2

Index Scan
Table: a
Index: a(a3)
Target List: a1, a2
Qualification: a3 = 42

Index Scan
Table: b
Index: b(b2)
Target List: b1, b2
Qualification: b2 = $1

```

Adjacent duplicate tuples removed

Tuples sorted to bring duplicates together

Current a2 value
PostgreSQL Backend Directories

- **bootstrap**
  creates initial template database via initdb

- **main**
  passes control to postmaster or postgres

- **postmaster**
  controls postgres server startup/termination

- **libpq**
  backend libpq library routines
PostgreSQL Backend Directories

- **tcop**
  - traffic cop, dispatches request to proper module

- **parser**
  - converts SQL query to query tree

- **optimizer**
  - creates path and plan

- **optimizer/path**
  - creates path from parser output
PostgreSQL Backend Directories

- optimizer/geqo
  genetic query optimizer

- optimizer/plan
  optimizes path output

- optimizer/prep
  handle special plan cases

- optimizer/util
  optimizer support routines
PostgreSQL Backend Directories

executor
  executes complex node plans from optimizer

commands
  commands that do not require complex handling

catalog
  system catalog manipulation

storage
  manages various storage systems
    storage/buffer       file       ipc      large_object
    lmgr                page      smgr
PostgreSQL Backend Directories

access

various data access methods
access/common  gist  hash  heap  index
nbtree  rtree  transam

nodes
creation/manipulation of nodes and lists

utils
support routines

utils/adt
built-in data type routines
PostgreSQL Backend Directories

- **utils/cache**
  - system/relation/function cache routines

- **utils/error**
  - error reporting routines

- **utils/fmgr**
  - function manager

- **utils/hash**
  - hash routines for internal algorithms
PostgreSQL Backend Directories

utils/init
  various initialization stuff

utils/misc
  miscellaneous stuff

utils/mmgr
  smemory manager (process-local memory)

utils/sort
  sort routines for internal algorithms
PostgreSQL Backend Directories

- **utils/time**
  - transaction time qualification routines
  - include
    - include files
  - lib
    - library support
  - regex
    - regular expression library
  - rewrite
    - rules system
Let us check the different modules/files of PostgreSQL Source Code
System Tables