Introducing PostgreSQL SQL Parser

- Use of PostgreSQL Parser in other Applications -

PGCon 2019, Ottawa
2019-05-31

SRA OSS, Inc. Japan
Bo Peng
pengbo@sraoss.co.jp

Who am I?

- Bo Peng
- Chinese National, based in Tokyo
- Organization: SRA OSS
- Experience in using PostgreSQL since 2016
- Current work
  - Open source software technical support
    - Monitoring software
    - Clustering software
  - Construction work
- Pgpool-II developer
  - Chinese documentation translation
  - Committer since Pgpool-II 3.5
  - Release management, bug fix, SQL parser

Outline

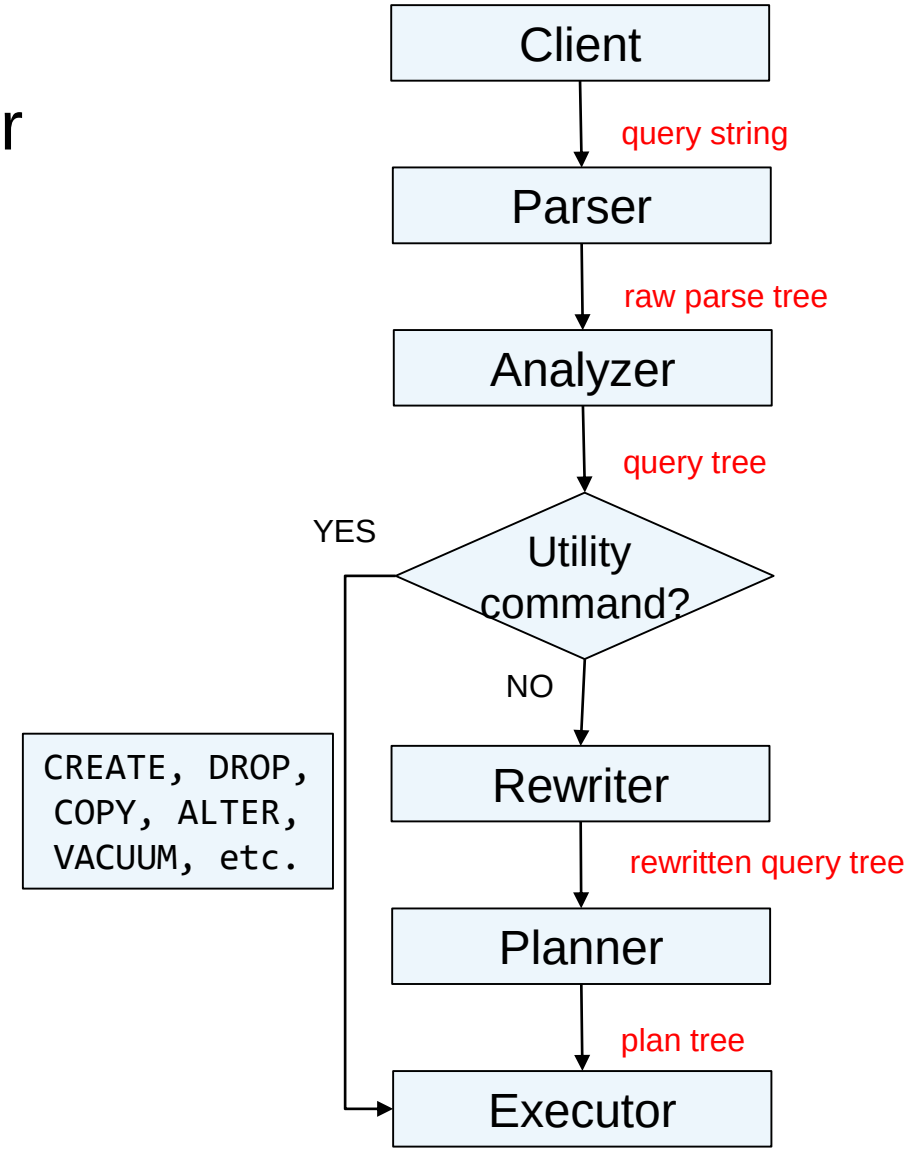
- PostgreSQL Query Processing
  - Parser
  - Analyzer
  - Rewriter
  - Planner
  - Executor
- How to port the raw parser into an application?
  - Give a concrete example
  - Some use cases of raw parser

Part Ⅰ

PostgreSQL Query Processing

Query Processing

- Parser
  - Check the query string for valid syntax
- Analyzer
  - Add detailed info
  - Database lookups
- Rewriter
  - Apply rewrite rules
- Planner
  - Choose the best plan
- Executor



Parser

- Check the query syntax
  - input: query string
  - output: raw parse tree
- Source code: `src/backend/parser`
- Parser is defined in `scan.l` and `gram.y`:
  - `scan.l`:
    - definition of lexer
    - recognizing identifiers, the SQL key words etc.
    - built using flex
  - `gram.y`:
    - definition of parser
    - built using bison
- No lookups in the system catalogs

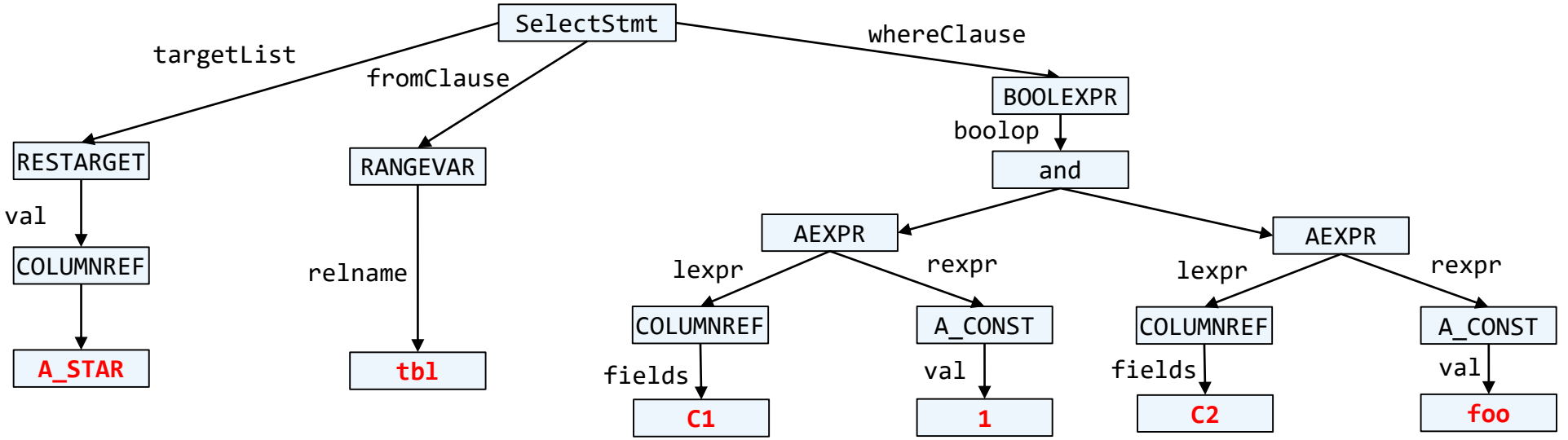
Example: a Simple Select Statement

```
SELECT * FROM tbl WHERE c1 > 1 AND c2 = 'foo';
```

gram.y

```
simple_select:
        | SELECT distinct_clause target_list
        into_clause from_clause where_clause
        group_clause having_clause window_clause
            {
                SelectStmt *n = makeNode(SelectStmt);     /* create SELECT node */
                  n->distinctClause = $2;                 /* DISTINCT clause*/
                  n->targetList = $3;                     /* TARGET list*/
                  n->intoClause = $4;                     /* INTO clause*/
                  n->fromClause = $5;                     /* FROM clause*/
                  n->whereClause = $6;                    /* WHERE clause*/
                  n->groupClause = $7;                    /* GROUP BY clause*/
                  n->havingClause = $8;                   /* HAVING clause*/
                  n->windowClause = $9;                   /* WINDOW clause*/
                $$ = (Node *)n;
```

```
SELECT col1,
       col2
FROM   tbl
WHERE  col1 > 1 AND
       col2 = 'foo';
```



Example: Raw Parse Tree

```
SELECT * FROM tbl WHERE c1 > 1 AND c2 = 'foo';
```

1. Apply patch

```
diff --git a/src/backend/tcop/postgres.c b/src/backend/tcop/postgres.c
index 44a59e1..4e046f6 100644
--- a/src/backend/tcop/postgres.c
+++ b/src/backend/tcop/postgres.c
@@ -1075,6 +1075,9 @@ exec_simple_query(const char *query_string)
                Portal          portal;
                DestReceiver *receiver;
                int16           format;
+
+       if (Debug_print_parse)
+           elog_node_display(LOG, "raw parse tree", parsetree,
Debug_pretty_print);
```

2. debug_print_parse = on

Raw Parse Tree

```
:stmt
   {SELECT
   :distinctClause <>
   :intoClause <>
   :targetList (
      {RESTARGET
      :name <>
      :indirection <>
      :val
         {COLUMNREF
         :fields (
            {A_STAR
            }
         )
         :location 7
         }
      :location 7
      }
   )
   :fromClause (
      {RANGEVAR
      :schemaname <>
      :relname tbl
      :inh true
      :relpersistence p
      :alias <>
      :location 14
      }
   )
   :whereClause
      {BOOLEXPR
      :boolop and
      :args (
         {AEXPR
         :name (">")
         :lexpr
            {COLUMNREF
            :fields ("c1")
            :location 24
            }
         :rexpr
            {A_CONST
            :val 1
            :location 29
            }
         :location 27
         }
         {AEXPR
         :name ("=")
         :lexpr
            {COLUMNREF
            :fields ("c2")
            :location 35
            }
         :rexpr
            {A_CONST
            :val "foo"
            :location 40
            }
         :location 38
         }
      )
      :location 31
      }
   :groupClause <>
   :havingClause <>
   (snip)
```

Analyzer (1)

- Analyzer
  - input: raw parse tree
  - output: query tree
- System catalog lookups
  - Add detailed info, like:
    - Table OID
    - Column name
    - Type, operator OID
- Query Tree
  - `src/include/nodes/parsenodes.h`

Raw Parse Tree vs. Query Tree

```
SELECT * FROM tbl WHERE c1 > 1 AND c2 = 'foo';
```

raw parse tree

```
:targetList (
   {RESTARGET
   :name <>
   :indirection <>
   :val
      {COLUMNREF
      :fields (
         {A_STAR
         }
      )
      :location 7
      }
   :location 7
   }
)
:fromClause (
   {RANGEVAR
   :schemaname <>
   :relname tbl
   :inh true
   :relpersistence p
   :alias <>
   :location 14
   }
)
```

*

FROM tbl

Analyzer

query tree

```
:rtable (
   {RTE
   :alias <>
   :eref
      {ALIAS
      :aliasname tbl
      :colnames ("c1" "c2")
      }
   :rtekind 0
   :relid 16405
   :relkind r
   :tablesample <>
  ...
}
)
 ...
:targetList (
   {TARGETENTRY
   :expr
      {VAR
      :varno 1
      :varattno 1
      :vartype 23
   ...
}
   :resno 1
   :resname c1
   :ressortgroupref 0
   :resorigtbl 16405
   :resorigcol 1
   :resjunk false
   }
```

table OID

column type

column name

Rewriter

- Rewriter
  - input: query tree
  - output: rewritten query tree
- Apply `RULE` to query tree
- Rewrite `VIEW`
  - Rewrite the client's query to the base table referenced in the view
- Source code
  - `src/backend/rewrite`

Planner

- Planner
  - input: rewritten query tree
  - output: plan tree
- Planner is responsible for creating an optimal execution plan
  1. Create all possible plans
  2. Estimate the cost for each plan
  3. Choose the lowest cost plan and return it to Executor

Executor

- Executor
  - input: plan tree
  - output: result
- Executor executes queries
  - Step through the plan tree
  - Retrieve tuples in the way given by the plan tree

Part Ⅱ

How to Import Parser into an Application.

Why?

- Why import PostgreSQL raw parser for parsing SQL queries in our own applications.
  - Determining if it is a READ query or WRITE query
    - Accomplish load balancing in applications
  - Extracting specific part from query
    - Extract table name or function name
  - Rewriting or modifying parts of the query string

How to parse complex SQL queries?

```
with year_total as (
select c_customer_id customer_id
      ,c_first_name customer_first_name
      ,c_last_name customer_last_name
      ,c_preferred_cust_flag customer_preferred_cust_flag
      ,c_birth_country customer_birth_country
      ,c_login customer_login
      ,c_email_address customer_email_address
      ,d_year dyear
      ,sum(((ss_ext_list_price-ss_ext_wholesale_cost-ss_ext_discount_amt)+ss_ext_sales_price)/2) year_total
      ,'s' sale_type
from customer
    ,store_sales
    ,date_dim
…
)
 select
…
from year_total t_s_firstyear
    ,year_total t_s_secyear
    ,year_total t_c_firstyear
    ,year_total t_c_secyear
    ,year_total t_w_firstyear
    ,year_total t_w_secyear
where t_s_secyear.customer_id = t_s_firstyear.customer_id
  and t_s_firstyear.customer_id = t_c_secyear.customer_id
  and t_s_firstyear.customer_id = t_c_firstyear.customer_id
  and t_s_firstyear.customer_id = t_w_firstyear.customer_id
  and t_s_firstyear.customer_id = t_w_secyear.customer_id
and t_s_secyear.sale_type = 's'
and t_c_secyear.sale_type = 'c'
and t_s_firstyear.year_total > 0
and t_c_firstyear.year_total > 0
and t_w_firstyear.year_total > 0
and case when t_c_firstyear.year_total > 0 then t_c_secyear.year_total / t_c_firstyear.year_total else null end
        > case when t_s_firstyear.year_total > 0 then t_s_secyear.year_total / t_s_firstyear.year_total else null end
and case when t_c_firstyear.year_total > 0 then t_c_secyear.year_total / t_c_firstyear.year_total else null end
        > case when t_w_firstyear.year_total > 0 then t_w_secyear.year_total / t_w_firstyear.year_total else null end
order by t_s_secyear.customer_id
      ,t_s_secyear.customer_first_name
      ,t_s_secyear.customer_last_name
      ,t_s_secyear.customer_birth_country limit 100;
```

Using regular expression to parse complex SQL queries is painful.

How to Port the Raw Parser to an Application - Using Pgpool-II as a concrete example -

Why Pgpool-II needs to parse SQL queries?

- Pgpool-II is a database clustering software.
- It provides the following features:
  - Load Balancing
  - Connection Pooling
  - Automatic failover
  - Replication, etc.
- Pgpool-II has imported PostgreSQL parser
  - To accurately parse the SQLs, using regular expressions to parse complex SQL queries is painful (Load balancing)
    - Determine if it is a READ query or WRITE query
    - Find specific function name from SQL query
    - Find specific patterns from SQL query
  - To rewrite the query (Replication)
    - Rewrite Date/Time functions to timestamp constant for synchronizing databases

How to import raw parser into Pgpool-II?

1. Import the following files to Pgpool-II to create a raw parse tree

src/parser/

```
src/backend/nodes/copyfuncs.c
src/backend/nodes/list.c
src/backend/nodes/makefuncs.c
src/backend/nodes/nodes.c
src/backend/nodes/value.c
src/backend/parser/gram.y
src/backend/parser/parser.c
src/backend/parser/scansup.c
src/backend/parser/scan.c
src/backend/lib/stringinfo.c
src/common/keywords.c
src/backend/utils/mb/wchar.c
```


src/utils/

```
src/backend/utils/mmgr/mcxt.c
src/backend/utils/mmgr/aset.c
src/backend/utils/error/elog.c
src/common/psprintf.c
```

2. Process the raw parse tree in Pgpool-II (See examples)
3. Create the following files to convert a raw parse tree to a query string and send the query to PostgreSQL (new in Pgpool-II)

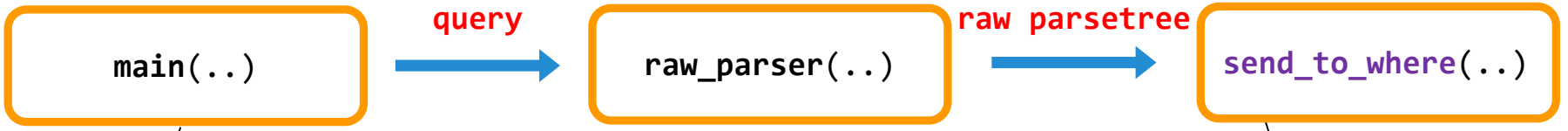
```
outfuncs.c
pool_string.c
```

Generate libsql-parser.a

- Use `libsql-parser.a` to explain the use case of processing raw parse tree in Pgpool-II.
- When compiling Pgpool-II source code, a static library `libsql-parser.a` is generated.
- Create a main program `query_parse.c` and link it to the static library.

```
gcc -o query_parse query_parse.c -L. -lsql-parser
```

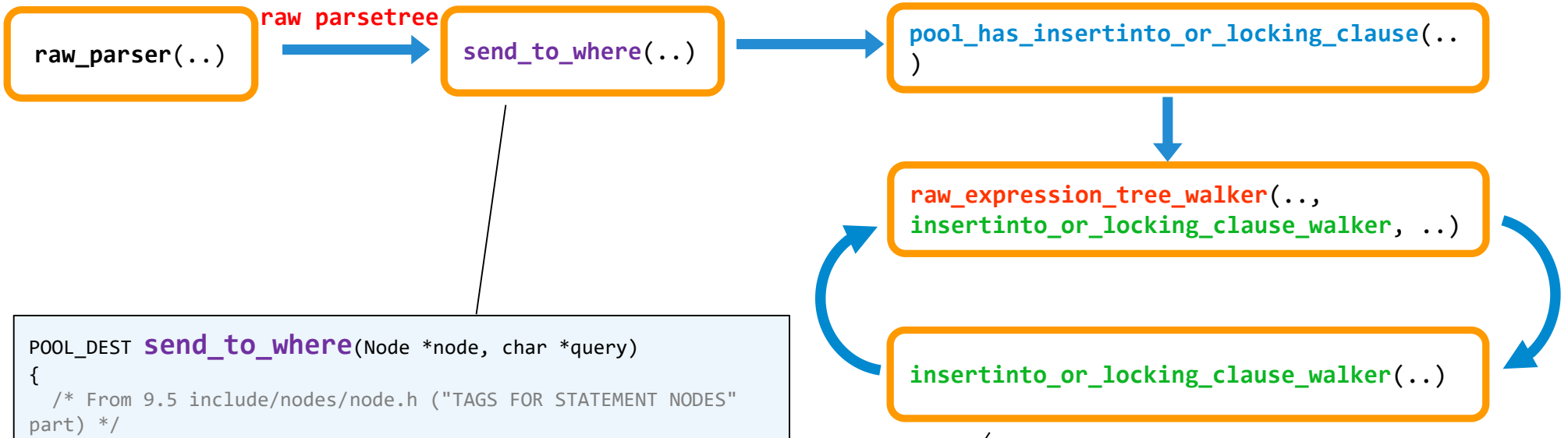
USE CASE (1) : Find a READ or WRITE query




```
# runparser.c
int main(int argc, char **argv)
{
  POOL_DEST   dest;

  parsetree_list = raw_parser(query, &error);
  node = raw_parser2(parsetree_list);

  /* READ or WRITE query */
  dest = send_to_where(node, query);

  if (dest == POOL_EITHER)
        printf("query: ¥"%s¥" is a READ query¥n", query);
  else (dest == POOL_PRIMARY)
        printf("query: ¥"%s¥" is a WRITE query¥n", query);
```

```
POOL_DEST send_to_where(Node *node, char *query)
{
  /* SELECT INTO or SELECT FOR SHARE or UPDATE ? */
  if (IsA(node, SelectStmt))
  {
   ...
  }
  /* COPY */
  else if (IsA(node, CopyStmt))
  {
   ...
  }
  /* Transaction commands */
  else if (IsA(node, TransactionStmt))
  {
   ...
  }
  /* SET */
  else if (IsA(node, VariableSetStmt))
  {
   ...
  }
```

USE CASE (1) : Find a READ or WRITE query – SELECT




```
POOL_DEST send_to_where(Node *node, char *query)
{
  /* From 9.5 include/nodes/node.h ("TAGS FOR STATEMENT NODES"
part) */
  static NodeTag nodemap[] = {
    T_InsertStmt,
    T_UpdateStmt,
    T_SelectStmt,
     . . .
  };

  if (bsearch(&nodeTag(node), nodemap,
      sizeof(nodemap) / sizeof(nodemap[0]),
      sizeof(NodeTag), compare) != NULL)
  {
    if (IsA(node, SelectStmt))
    {
      /* SELECT INTO or SELECT FOR SHARE or UPDATE ? */
      if (pool_has_insertinto_or_locking_clause(node))
        return POOL_PRIMARY;

      return POOL_EITHER;
    }
```

```
/* Walker function to find intoClause or lockingClause */
static bool
insertinto_or_locking_clause_walker(..)
{
  SelectContext ctx;

  if (IsA(node, IntoClause) ||IsA(node, LockingClause))
  {
    ctx.has_insertinto_or_locking_clause = true;
    return false;
  }

  return raw_expression_tree_walker(node,
      insertinto_or_locking_clause_walker,
      &ctx);
}
```

USE CASE (1) : Find a READ or WRITE query - SELECT

Example:

- Check if "**FOR UPDATE**" appears in the **SELECT** query.
- Consider "**SELECT … FOR UPDATE**" as a WRITE query and send it to primary node.

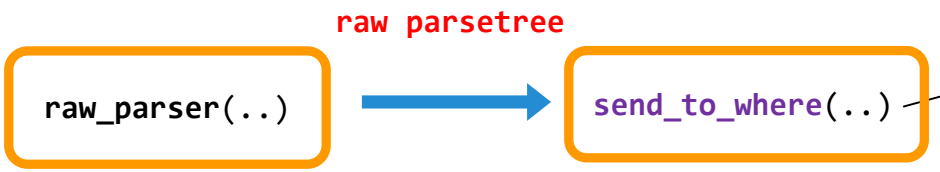
```
$ ./query_parse "SELECT * FROM tbl FOR UPDATE"

query: " SELECT * FROM tbl FOR UPDATE " is a WRITE query
```

- Consider "**SELECT …** " without " **FOR UPDATE** " query as a READ query and send it to the load balance node.

```
$ ./query_parse "SELECT * FROM tbl"

query: " SELECT * FROM tbl " is a READ query
```

USE CASE (1) : Find a READ or WRITE query - COPY

```
COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
```




```
POOL_DEST send_to_where(Node *node, char *query)
{
 (snip)
    /* COPY */
    if (IsA(node, CopyStmt))
    {
      if (((CopyStmt *) node)->is_from)
        return POOL_PRIMARY;
      else
      {
        if (((CopyStmt *) node)->query == NULL)
          return POOL_EITHER;
        else
          return (IsA(((CopyStmt *) node)->query,
SelectStmt)) ?
                  POOL_EITHER : POOL_PRIMARY;
        }
      }
  (snip)
```

- Check if a **COPY** query is a **READ** query or a **WRITE** query.
- If a **COPY** query copies the results of a WRITE query, then it is a **WRITE** query.

```
$ ./query_parse "COPY (UPDATE tbl SET i = i + 1 RETURNING *) TO STDOUT;"
query: "COPY (UPDATE test SET i=i+1 RETURNING *) TO STDOUT;" is a WRITE query
```

- If a **COPY** query copies the results of a SELECT query , then it is a **READ** query

```
$ ./query_parse "COPY (SELECT * FROM tbl) TO STDOUT;"
query: " COPY (SELECT * FROM test) TO STDOUT;" is a READ query
```

USE CASE (1) : Find a READ or WRITE query- SET

SET

- Send "***SET … TO…*** " to both of primary node and load balance node
- However, the following "***SET TRANSACTION***" should be sent to the *primary* only.
  - *SET transaction_read_only TO off*
  - *SET transaction_isolation TO 'serializable'*
  - *SET default_transaction_isolation TO 'serializable'*
  - *SET TRANSACTION ISOLATION LEVEL SERIALIZABLE*
  - *SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE*
  - *SET TRANSACTION READ WRITE*
  - *SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE*


```
POOL_DEST send_to_where(Node *node, char *query)
{
 (snip)
  /* COPY */
  if (IsA(node, VariableSetStmt) )
  {
    if (((VariableSetStmt *) node)->kind == VAR_SET_VALUE &&
         !strcmp(((VariableSetStmt *) node)->name,
         "transaction_read_only"))
    {
     (snip)
      switch (v->val.type)
      {
        case T_String:
          if (!strcasecmp(v->val.val.str, "off") ||
              !strcasecmp(v->val.val.str, "f") ||
              !strcasecmp(v->val.val.str, "false"))
              ret = POOL_PRIMARY;
     (snip)
    }
    else if (((VariableSetStmt *) node)->kind == VAR_SET_VALUE &&
              (!strcmp(((VariableSetStmt *) node)->name,
               "transaction_isolation") ||
               !strcmp(((VariableSetStmt *) node)->name,
               "default_transaction_isolation")))
    {
      switch (v->val.type)
      {
        case T_String:
          if (!strcasecmp(v->val.val.str, "serializable"))
              return POOL_PRIMARY;
     (snip)
```

USE CASE (1) : Find a READ or WRITE query- SET

Example:

```
# runparser.c
int main(int argc, char **argv)
{
  (snip)
  /* READ or WRITE query */
  dest = send_to_where(node, query);

  if (dest == POOL_PRIMARY)
    printf("send ¥"%s¥" to primary node, query);
  else if (dest == POOL_BOTH)
    printf("send ¥"%s¥" to primary node and load blance node, query);
  (snip)
```

- SET parameters

```
$ ./query_parse "set client_encoding to 'utf8';"
send "SET client_encoding TO 'utf8'" to primary node and load balance node
```

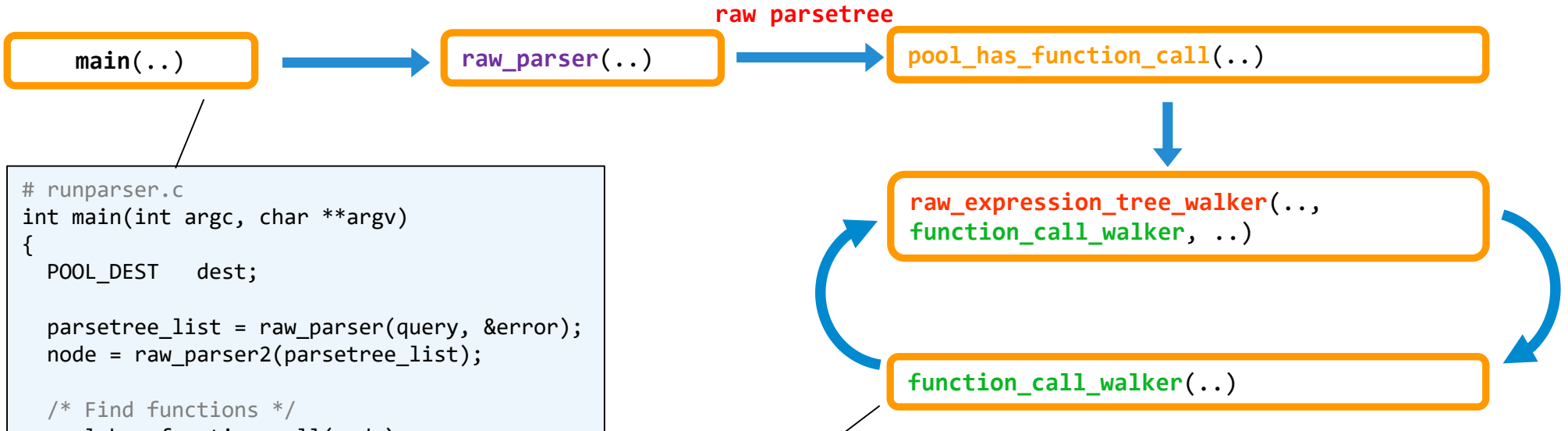
- SET transaction isolation level repeatable read

```
$ ./query_parse "set transaction isolation level repeatable read;"
send "SET TRANSACTION  ISOLATION LEVEL repeatable read" to primary node and load balance
node
```

- SET transaction isolation level serializable

```
$ ./query_parse "set transaction isolation level serializable;"
send "SET TRANSACTION  ISOLATION LEVEL serializable" to primary node
```

USE CASE (2) : Find functions




```
# runparser.c
int main(int argc, char **argv)
{
  POOL_DEST   dest;

  parsetree_list = raw_parser(query, &error);
  node = raw_parser2(parsetree_list);

  /* Find functions */
  pool_has_function_call(node);
```

```
/* Walker function to find intoClause or lockingClause */
static bool
function_call_walker(Node *node, void *context)
{
  SelectContext *ctx = (SelectContext *) context;

  if (IsA(node, FuncCall))
  {
    Call    *fcall = (FuncCall *) node;
    char    *fname;
    int     length = list_length(fcall->funcname);

    if (length > 0)
    {
      if (length == 1)     /* no schema qualification? */
        fname = strVal(linitial(fcall->funcname));
      else                 /* with schema qualification */
        fname = strVal(lsecond(fcall->funcname));
    }
    printf("function call walker, function name: ¥"%s¥"", fname)
   (snip)
  }
```

USE CASE (2) : Find functions

Example:

```
# test.sql
WITH customer_total_return AS
(
         SELECT   sr_customer_sk AS ctr_customer_sk ,
                  sr_store_sk    AS ctr_store_sk ,
                  Sum(sr_fee)    AS ctr_total_return
         FROM     store_returns ,
                  date_dim
         WHERE    sr_returned_date_sk = d_date_sk
         AND      d_year =2000
         GROUP BY sr_customer_sk ,
                  sr_store_sk)
SELECT   c_customer_id
FROM     customer_total_return ctr1 ,
         store ,
         customer
WHERE    ctr1.ctr_total_return >
         (
                SELECT Avg(ctr_total_return)*1.2
                FROM   customer_total_return ctr2
                WHERE  ctr1.ctr_store_sk = ctr2.ctr_store_sk)
AND      s_store_sk = ctr1.ctr_store_sk
AND      s_state = 'TN'
AND      ctr1.ctr_customer_sk = c_customer_sk
ORDER BY c_customer_id LIMIT 100;
```

- find functions and print function names

```
$ ./query_parse test.sql
function call walker, function name: avg
function call walker, function name: sum
```

Why rewrite Date/Time Functions?

- Pgpool-II native replication mode
  - Not rely on PostgreSQL streaming replication
  - Pgpool-II replicates queries to synchronize backend nodes
  - Local time difference of each node or query execution time difference could cause **inconsistency**
- Date/Time Functions
  - `CURRENT_DATE`
  - `CURRENT_TIME`
  - `CURRENT_TIMESTAMP`
  - `CURRENT_TIME (precision)`
  - `CURRENT_TIMESTAMP (precision)`
  - `LOCALTIME`
  - `LOCALTIMESTAMP`
  - `LOCALTIME (precision)`
  - `LOCALTIMESTAMP (precision)`

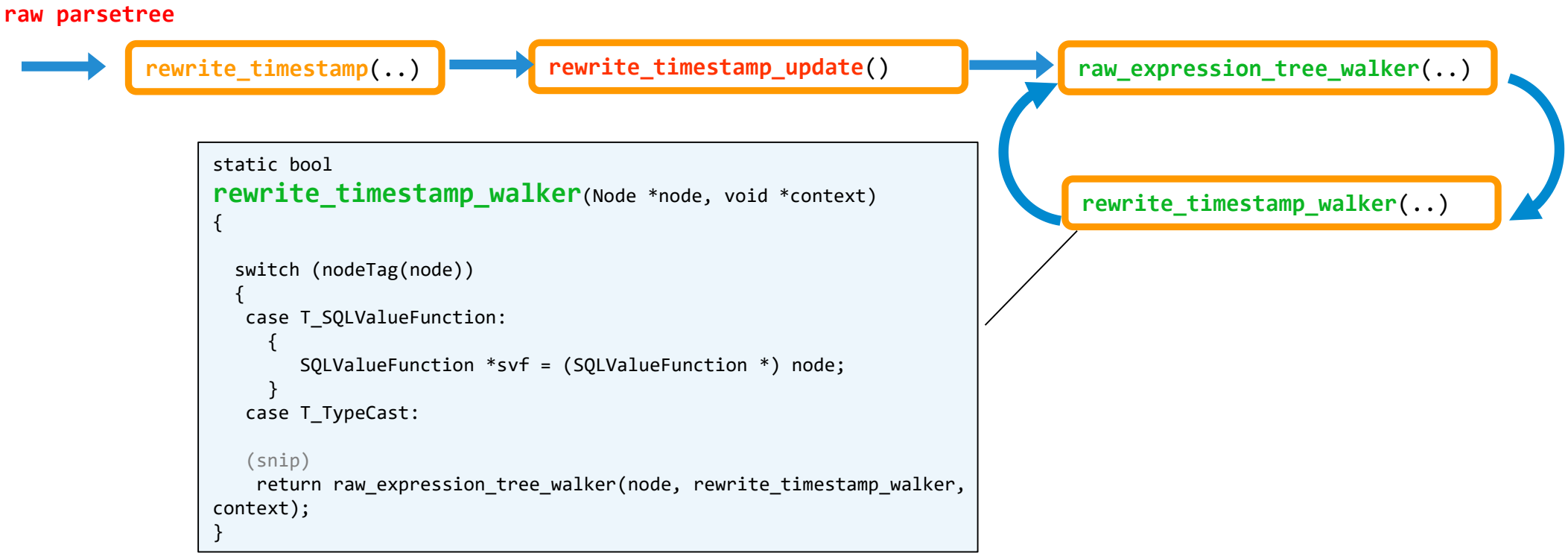
SRA OSS,INC.

Rewrite Date/Time Functions

1. Convert Date/Time functions such as `CURRENT_DATE, CURRENT_TIME, LOCALTIMESTAMP` etc. to "`'now'::text::timestamptz`" format.
2. Rewrite "`'now'::text::timestamptz`" to timestamp constant.

Example:

`CURRENT_TIMESTAMP -> '2019-04-10 13:59:59.123456+09'::text::timestamptz`




```
static bool
rewrite_timestamp_walker(Node *node, void *context)
{

  switch (nodeTag(node))
  {
   case T_SQLValueFunction:
     {
        SQLValueFunction *svf = (SQLValueFunction *) node;
     }
   case T_TypeCast:

   (snip)
    return raw_expression_tree_walker(node, rewrite_timestamp_walker,
context);
}
```

USE CASE (3) : Rewrite Date/Time Functions

Example:

- Check if a Date/Time function exists
  - If a Date/Time function exists

```
$ ./query_parse "INSERT INTO tbl VALUES(1, now(), 2, CURRENT_DATE);"
INSERT INTO "t3" VALUES (1,"pg_catalog"."timestamptz"('2019-04-22
23:48:20.0601+09'::text),2,'2019-04-22 23:48:20.0601+09'::text::date)
```

Summary

- PostgreSQL query processing
  - Parser
  - Analyzer
  - Rewriter
  - Planner
  - Executer
- Use PostgreSQL parser in applications to achieve functions, like:
  - Load balancing
  - Retrieve specific part from query
  - Rewrite query
- Parser processing library
  - https://github.com/pengbo0328/query_parse

Thank you!