pganalyze

How pg_query rewrites, deparses & formats any valid Postgres query

Presented by Keiko Oda & Lukas Fittl

Agenda for today

- What is pg_query?
- Query Rewriting in Practice
- Formatting any valid Postgres Query

What is pg_query?

What is pg_query?

pganalyze

Postgres parser as a standalone library

First launched with Postgres 10, continuously supported new releases since PG 13.

libpg_query

C library for accessing the PostgreSQL parser outside of the server.

This library uses the actual PostgreSQL server source to parse SQL queries and return the internal PostgreSQL parse tree.

```
SELECT * FROM mytable
```

```
SelectStmt
├─ targetList
│  └─ ResTarget
│     └─ ColumnRef
│        └─ A_Star
└─ fromClause
```

```
DELETE FROM abc WHERE id = 1
```

```
DeleteStmt
├─ relation
│  └─ RangeVar
│     └─ relname = abc
└─ whereClause
   └─ A_Expr
```

What is pg_query?

pganalyze

Postgres parser as a standalone library

```c
#include <pg_query.h>
#include <stdio.h>

int main() {
  PgQueryParseResult result;

  result = pg_query_parse("SELECT 1");

  printf("%s\n", result.parse_tree);

  pg_query_free_parse_result(result);

  return 0;
}
```

What is pg_query?

pganalyze

Postgres parser as a standalone library

```
{
    "version": 170007,
    "stmts": [
        {
            "stmt": {
                "SelectStmt": {
                    "targetList": [
                        {
                            "ResTarget": {
                                "val": {
                                    "A_Const": {
                                        "val": {
                                            "Integer": {
                                                "ival": 1
                                            }
                                        },
                                        "location": 7
                                    }
                                },
                                "location": 7
                            }
                        }
                    ],
                    "limitOption": "LIMIT_OPTION_DEFAULT",
                    "op": "SETOP_NONE"
                }
            }
        }
    ]
}
```

What is pg_query?

pganalyze

Or in Ruby:

```
require 'pg_query'
PgQuery.parse("SELECT 1")
```

Or in Rust:

```
use pg_query::parse;
pg_query::parse("SELECT * FROM contacts");
```

Or in Python (via pglast binding):

```
from pglast import parse_sql
root = parse_sql('select 1')
```

Why pg_query?

pganalyze

Applications need to parse SQL with full Postgres compatibility:

At **pganalyze** we needed to parse + fingerprint Postgres queries from pg_stat_statements and the logs. Other projects had other needs:

Postgres Language Server: Autocomplete, syntax checks & more in your editor
SQLlint: Simple SQL linter supporting ANSI and PostgreSQL syntaxes
pgspot: Spot vulnerabilities in postgres SQL scripts
pgDog: Proxy that can rewrite queries/route them to different shards

Fun Fact: The **DuckDB** parser originated from pg_query, before they forked it to expand their syntax.

Protobufs vs JSON

pganalyze

pg_query uses Protobufs to have language-specific structs that can be more easily accessed:

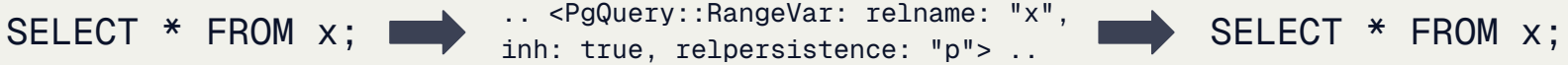
```
q = PgQuery.parse('SELECT * FROM x')
q.tree
#⇒ <PgQuery::ParseResult: version: 180004, stmts: [
  <PgQuery::RawStmt: stmt: <PgQuery::Node: select_stmt:
    <PgQuery::SelectStmt: distinct_clause: [],
      target_list: [
        <PgQuery::Node: res_target: <PgQuery::ResTarget: name: "", indirection: [],
          val: <PgQuery::Node: column_ref: <PgQuery::ColumnRef: fields: [<PgQuery::Node:
a_star: <PgQuery::A_Star: >>], location: 7>>, location: 7>>],
      from_clause: [<PgQuery::Node: range_var: <PgQuery::RangeVar: catalogname: "",
        schemaname: "", relname: "x", inh: true, relpersistence: "p", location: 14>>],
      ...

x.tree.stmts[0].stmt.select_stmt.from_clause
#⇒ [<PgQuery::Node: range_var: <PgQuery::RangeVar: catalogname: "", schemaname: "",
    relname: "x", inh: true, relpersistence: "p", location: 14>>]
```

pg_query has a deparser!

pganalyze

Deparsing means we turn a parse tree (AST) into SQL again

Feature complete, meaning it can turn 100% of the Postgres regression tests back into SQL, including utility statements.



Separate C file from the main pg_query code. It could be imported into Postgres if there was an in-core use case for raw parse tree deparsing.

Query Rewriting in Practice

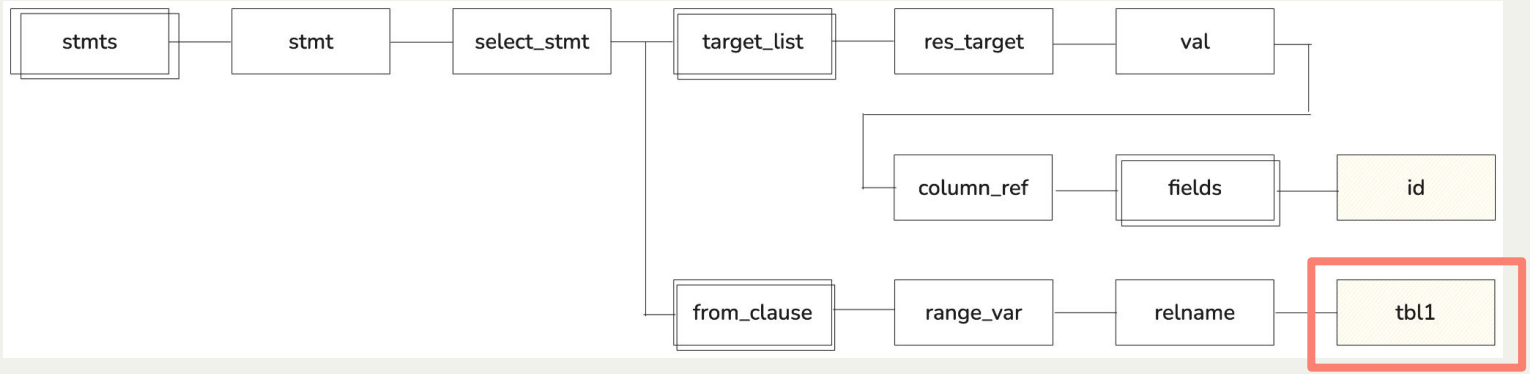
Query rewrite 101: Update Table Name

pganalyze

- **Goal:** Update the table name from "tbl1" to "tbl2"
- Use **"PgQuery.parse"** to turn SQL into a parse tree, then drill down to the field

example uses the Ruby bindings

```ruby
require "pg_query"

parsed_query = PgQuery.parse("SELECT id FROM tbl1")

parsed_query.tree.stmts[0].stmt.select_stmt.from_clause[0].range_var.relname
#⇒ "tbl1"
```

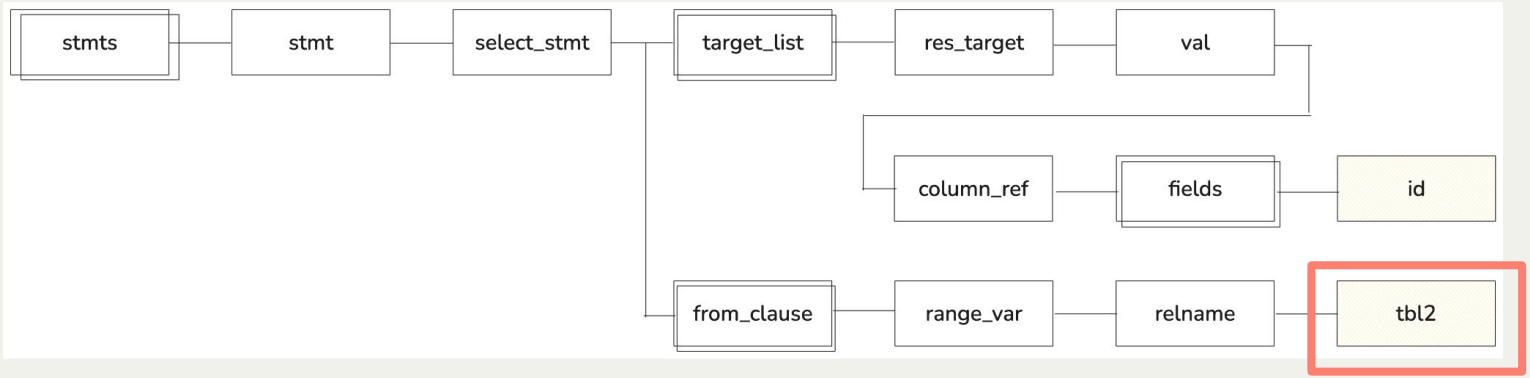


Query rewrite 101: Update Table Name

pganalyze

- Assign a new value to the field, then call **"#deparse"** to get SQL back
- The full pattern: **parse → modify → deparse**

```
parsed_query #⟹ input: "SELECT id FROM tbl1"
parsed_query.tree.stmts[0].stmt.select_stmt.from_clause[0].range_var.relname = "tbl2"
parsed_query.deparse
#⟹ "SELECT id FROM tbl2"
```



Rewrite OR to UNION to enable index scans across JOINs

pganalyze

- OR on columns from different tables → JOIN runs first, then filter

```
SELECT o.id, o.total, c.email
FROM orders o JOIN customers c
  ON c.id = o.customer_id
WHERE
  o.flagged IS TRUE
  OR
  c.is_blocked IS TRUE
# orders: 1M rows
(~1k rows with flagged: true)
# customers: 100k rows
(~500 rows with is_blocked: true)
```

Plan

```
1 Hash Join
    Hash Cond: (o.customer_id = c.id)
    Inner Unique: true
    Join Filter: (o.flagged OR c.is_blocked)
    Join Type: Inner
    Rows Removed by Join Filter: 994294
2   ├ Seq Scan on orders
3   └ Hash
4     └ Seq Scan on customers
```

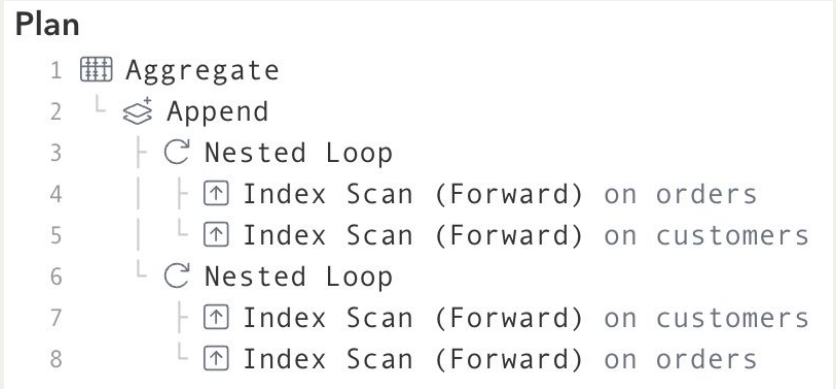
— Rewrite OR to UNION in PostgreSQL queries by Laurenz Albe

Rewrite OR to UNION to enable index scans across JOINs

pganalyze

- By rewriting OR to UNION, orders and customers can each use its own index

```sql
SELECT o.id, o.total, c.email
FROM orders o JOIN customers c
  ON c.id = o.customer_id
WHERE o.flagged IS TRUE

UNION

SELECT o.id, o.total, c.email
FROM orders o JOIN customers c
  ON c.id = o.customer_id
WHERE c.is_blocked IS TRUE
```



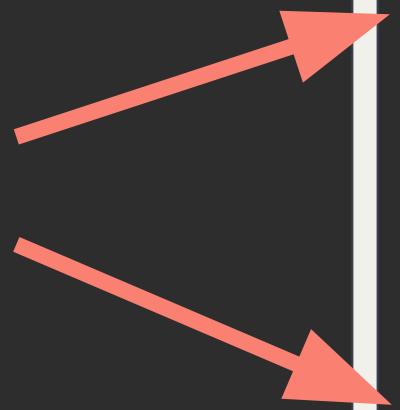
Rewrite OR to UNION: Plan Comparison

pganalyze

Plan A Using OR - Parameter Set 1	Plan B Using UNION - Parameter Set 1	Plan A Runtime	Plan B Runtime
-> Hash Join		177.76ms	
-> Seq Scan on orders		51.28ms	
-> Hash		32.84ms	
-> Seq Scan on customers		16.24ms	
	-> Aggregate		5.07ms
	-> Append		0.64ms
	-> Nested Loop		1.05ms
	-> Index Scan[4] on orders		5.17ms
	-> Index Scan[2] on custome…		8.36ms
	-> Nested Loop		1.22ms
	-> Index Scan[1] on custome…		0.74ms
	-> Index Scan[3] on orders		12.93ms
		278.34ms	35.74ms

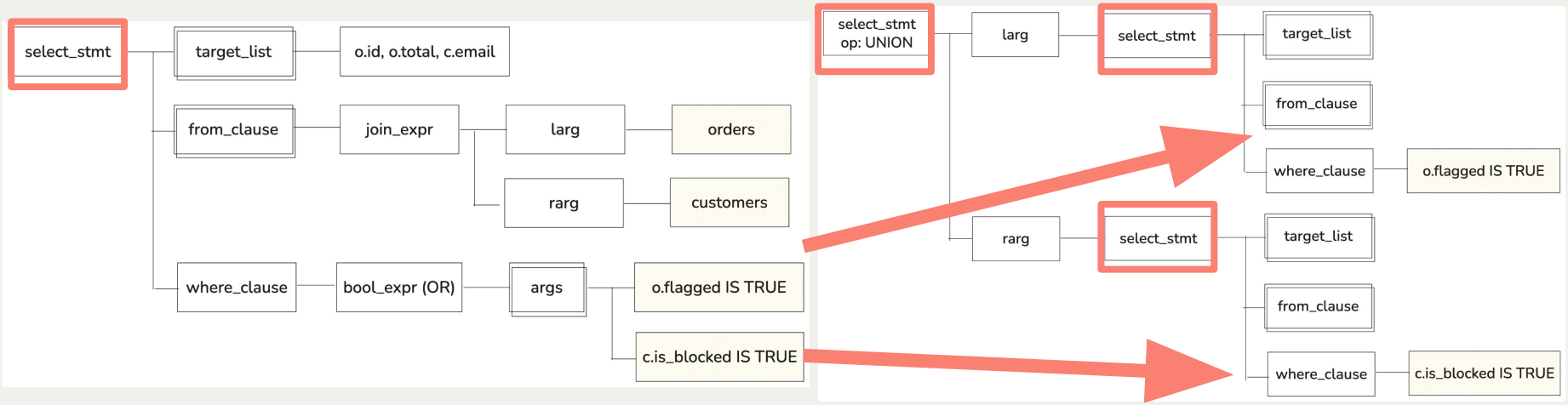
Rewrite OR to UNION: Query Comparison

```sql
SELECT o.id, o.total, c.email
FROM orders o JOIN customers c
  ON c.id = o.customer_id
WHERE
  o.flagged IS TRUE
  OR
  c.is_blocked IS TRUE
```



```sql
SELECT o.id, o.total, c.email
FROM orders o JOIN customers c
  ON c.id = o.customer_id
WHERE o.flagged IS TRUE

UNION

SELECT o.id, o.total, c.email
FROM orders o JOIN customers c
  ON c.id = o.customer_id
WHERE c.is_blocked IS TRUE
```

Rewrite OR to UNION: Parse Tree Comparison



Rewrite OR to UNION to enable index scans across JOINs

pganalyze

```ruby
# 1. Walk the tree to find the SelectStmt to rewrite, and remember where it lives
target_select = nil
target_loc = nil
parsed_query.walk! do |_, _, node, loc| # walk! visits every node in the tree
  next unless is_target_select?(node)
  target_select = node
  target_loc = loc
end
# 2. Walk the subtree to find the OR node
or_node = nil
parsed_query.walk_subtree!(target_select) do |node| # visits every node in the subtree
  or_node = node if node.is_a?(PgQuery::BoolExpr) && node.boolop == :OR_EXPR
end
```

Rewrite OR to UNION to enable index scans across JOINs

pganalyze

```
# 3. Turn each OR branch into its own SelectStmt
left_select = Google::Protobuf.deep_copy(target_select)
left_select.where_clause = or_node.args[0]
right_select = Google::Protobuf.deep_copy(target_select)
right_select.where_clause = or_node.args[1]

# 4. Combine the branches with UNION and write it back to the recorded location
union = PgQuery::SelectStmt.new(larg: left_select, rarg: right_select, op: :SETOP_UNION)
parsed_query.find_tree_location(parsed_query.tree, target_loc) do |parent_node, _, _|
  parent_node.select_stmt = union
end
```

Rewrite OR to UNION to enable index scans across JOINs

pganalyze

```
# 5. Deparse

parsed_query.deparse

#⇒ "SELECT o.id, o.total, c.email FROM orders o JOIN customers c ON c.id = o.customer_id
WHERE o.flagged IS TRUE UNION SELECT o.id, o.total, c.email FROM orders o JOIN customers
c ON c.id = o.customer_id WHERE c.is_blocked IS TRUE"
```

Formatting any valid Postgres Query

Query Formatter

- **Problem:** Rewrite diff is messy and unclear what changed
- **Solution:** Teach deparser to format before/after queries

```
SELECT databases.id, s.reset_value::bigint FROM databases JOIN
postgres_settings s USING (server_id) WHERE databases.id =
ANY($1::bigint[]) AND s.name = $2 AND s.removed_at IS NULL
```

```
WITH __postgres_settings AS MATERIALIZED (SELECT s.reset_value,
s.server_id FROM postgres_settings s WHERE s.name = $2) SELECT
databases.id, s.reset_value::bigint FROM databases JOIN
__postgres_settings s USING (server_id) WHERE databases.id =
ANY($1::bigint[]) AND s.removed_at IS NULL
```

```
SELECT databases.id, s.reset_value::bigint
FROM
  databases
  JOIN postgres_settings s USING (server_id)
WHERE
  databases.id = ANY($1::bigint[])
  AND s.name = $2
  AND s.removed_at IS NULL
```

```
WITH __postgres_settings AS MATERIALIZED (
  SELECT s.reset_value, s.server_id
  FROM postgres_settings s
  WHERE s.name = $2
)
SELECT databases.id, s.reset_value::bigint
FROM
  databases
  JOIN __postgres_settings s USING (server_id)
WHERE
  databases.id = ANY($1::bigint[])

  AND s.removed_at IS NULL
```

Query Formatter

- Pass options to deparse method

```
opts = PgQuery::DeparseOpts.new(
  pretty_print: true,              # default: false
  indent_size: 2,                  # default: 4
  max_line_length: 80,             # default: 80
  trailing_newline: true,          # default: false
  commas_start_of_line: false      # default: false
)
parsed_query.deparse(opts: opts)
```

— Pretty Print Options

Query Formatter

pganalyze

- Comments can be preserved

```
query_text = "/* my comment */\nSELECT /* my inline comment */1"
parsed_query = PgQuery.parse(query_text)
opts = PgQuery::DeparseOpts.new(…)
parsed_query.deparse(opts: opts)
#⇒ "SELECT 1\n"
opts.comments = PgQuery.deparse_comments_for_query(query_text)
parsed_query.deparse(opts: opts)
#⇒ "/* my comment */\nSELECT /* my inline comment */1\n"
```

— Pretty Print Options

pganalyze Query Formatter

- Available on https://pganalyze.com/format

Thank you!

Learn more at github.com/pganalyze/libpg_query

Contact us: lukas@pganalyze.com, keiko@pganalyze.com