The Pushdown: Building a Foreign Data Wrapper

David E. Wheeler
ClickHouse, PGXN

PGConf.dev
2026-05-21

👋 Hello!

I'm David Wheeler

Principle Engineer, ClickHouse

PostgreSQL user since 2000

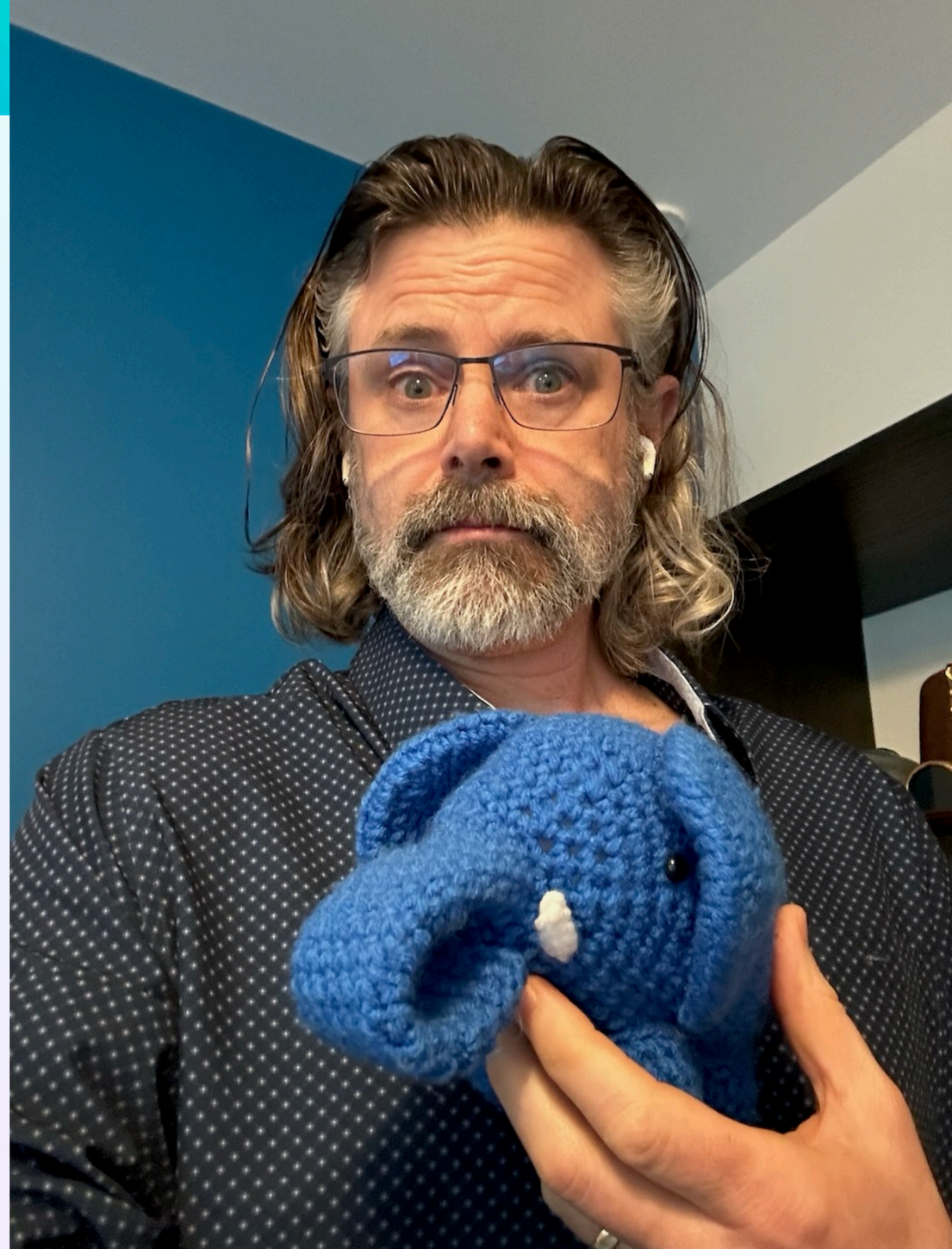
PostgreSQL contributor since 2008

pgTAP: Unit testing for PostgreSQL

Sqitch: Database change management

PGXN: PostgreSQL Extension network

pg_clickhouse: A foreign data wrapper



📖 Context

ClickHouse: OSS analytical database

Postgres: #2 Source for ClickHouse migrations

PeerDB/ClickPipes CDC great for data syncing

Migrating queries more difficult

- Baked into apps or ORMs
- Developed over years

Goal: minimize the need to rewrite queries

Solution: Build a foreign data wrapper

FDW

Foreign Data Wrapper

Postgres SQL/MED

- Allows access to data outside Postgres
- Presents data as foreign tables

FDW communicates with external data source

- Hides connection & data access details
- Data accessed via standard Postgres queries

Transparent to the user

🧺 The Cornucopia

postgres_fw
oracle_fdw
mysql_fdw
nformix_fdw
db2_fdw
firebird_fdw
sqlite_fdw
sybase_fds
tds_fdw
monetdb_fdw

cassandra_fdw
couchdb_fdw
clickhouse_fdw
influxdb_fdw
kafka_fdw
mongo_fdw
neorj_fdw
redis_fdw
sparql_fdw
wdb_fdw

file_fdw
compressedfile_fdw
json_fdw
multicdr_fdw
parquet_fdw
dump_fdw
pg_sheet_fdw
zipfile_fdw
passwd_fdw
rng_fdw

www_fdw
git_fdw
fb_fdw
dataclips_fdw
s3_fdw
telegram_fdw
twitter_fdw
airtable_fdw
firebase_fdw
stripe_fdw

Writing an FDW

PostgreSQL extension

Written in C*

Handles queries for foreign table

Processes query parse tree

- Give planner execution paths
- Executes plan selected by planner
- Returns results to executor

*Maybe Wrappers (Rust) or Multicorn (Python)

```
CREATE FUNCTION my_fdw_handler()
RETURNS fdw_handler
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

CREATE FUNCTION my_fdw_validator(text[], oid)
RETURNS VOID
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;

CREATE FOREIGN DATA WRAPPER my_fdw
HANDLER my_fdw_handler
VALIDATOR my_fdw_validator;
```

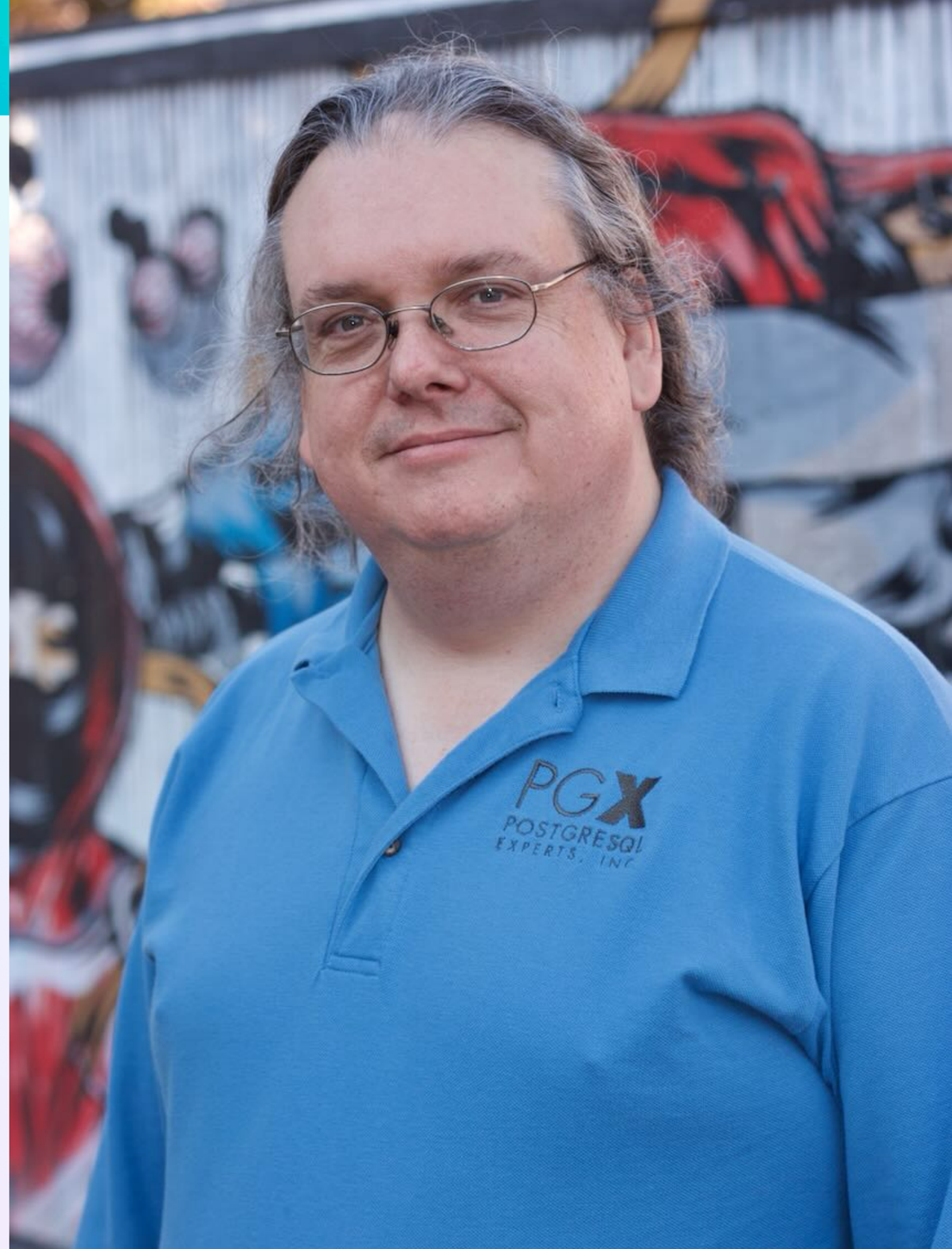
```c
Datum
my_fdw_handler(PG_FUNCTION_ARGS)
{
    FdwRoutine *routine = makeNode(FdwRoutine);

    /* Functions for scanning foreign tables */
    routine->GetForeignRelSize    = myGetForeignRelSize;
    routine->GetForeignPaths      = myGetForeignPaths;
    routine->GetForeignJoinPaths  = myGetForeignJoinPaths;
    routine->GetForeignUpperPaths = myGetForeignUpperPaths;
    routine->GetForeignPlan       = myGetForeignPlan;
    routine->BeginForeignScan     = myBeginForeignScan;
    routine->IterateForeignScan   = myIterateForeignScan;
    routine->ReScanForeignScan    = myReScanForeignScan;
    routine->EndForeignScan       = myEndForeignScan;
    routine->ExplainForeignScan   = myExplainForeignScan;

    /* Functions for foreign table maintenance. */
    routine->AnalyzeForeignTable  = myAnalyzeForeignTable;
    routine->ImportForeignSchema  = myImportForeignSchema;

    PG_RETURN_POINTER(routine);
}
```

🐘 PGCon 2023

Writing a Foreign Data Wrapper

Christoph Pettus

https://youtu.be/7wuDJxpU7Fo



🐶 Fetching Data

Simple FDW returns all data

Equivalent of a full table scan

Perfect for limited data

- Log files
- Reference tables
- API calls

Not great for large data sets

⬇️ Pushdown

Optimize for remote execution

For capable foreign sources

Keep operations close to data

Rely on remote database optimizations

Minimize network overhead

Pushdown:

- JOINs
- `WHERE`
- `GROUP BY`
- `HAVING`
- `LIMIT/OFFSET`

GetForeignRelSize

Called at start of query planning

Initialize private data structures

Calculate size estimates for foreign table

Categorize local & remote conditions

- Remote conditions reduce bandwidth
- Local conditions increase bandwidth

Determine attributes to retrieve for output, joins, local conditions

Set costs to favor join pushdown over base scans

```
myGetForeignRelSize(
    PlannerInfo * root, RelOptInfo * baserel, Oid foreigntableid
) {
    MyFdwRelationInfo *fpinfo = palloc0(sizeof(MyFdwRelationInfo));
    baserel->fdw_private = (void *) fpinfo;

    fpinfo->rows = baserel->rows;
    fpinfo->startup_cost = 10.0;
    fpinfo->total_cost = 10.0 + baserel->rows * 0.01;

    ListCell   *lc;
    foreach(lc, baserel->baserestrictinfo) {
        RestrictInfo *ri = lfirst_node(RestrictInfo, lc);

        if (is_foreign_expr(ri->clause))
            fpinfo->remote_conds = lappend(fpinfo->remote_conds, ri)
        else
            fpinfo->local_conds = lappend(fpinfo->local_conds, ri);
    }
}
```

GetForeignPaths

Create possible access paths for rel or joinrel

At least one: foreign table scan

Examine sorting expressions (PathKeys)

Add paths where sorting can be pushed down

- Avoid local sorting & allow efficient local merge joins
- Set lower costs to encourage pushdown
- Add foreign scan or join scan paths for each

```
static void myGetForeignPaths(
    PlannerInfo * root, RelOptInfo * baserel, Oid foreigntableid
) {
   MyFdwRelationInfo *fpinfo = baserel->fdw_private;

    /* Start with a full scan of the base relation. */
    add_path(baserel, (Path *) create_foreignscan_path(
        root, baserel, NULL, fpinfo->rows, 0,
        fpinfo->startup_cost, fpinfo->total_cost,
        NULL, NULL, NULL, NIL, NIL
    ));

    /* Add paths where sorting can be pushed down. */
    add_paths_with_pathkeys_for_rel(root, baserel);
}
```

```
static void add_paths_with_pathkeys_for_rel(
    PlannerInfo * root, RelOptInfo * rel
) {
    ListCell   *lc;

    /* Create one path for each set of pathkeys we found above. */
    foreach(lc, get_useful_pathkeys_for_relation(root, rel)) {

        if (IS_SIMPLE_REL(rel)) {
            add_path(rel, (Path *) create_foreignscan_path(
                root, rel, NULL, 1000, 0, 1.0, 0.5,
                lfirst(lc), NULL, NULL, NIL, NIL
            ));
        } else {
            add_path(rel, (Path *)create_foreign_join_path(root, rel,
                root, rel, NULL, 1000, 0, 1.0, 0.5,
                lfirst(lc), NULL, NULL, NIL, NIL
            ));
        }
    }
}
```

```
static List * get_useful_pathkeys_for_relation(
    PlannerInfo * root, RelOptInfo * rel
) {
    MyFdwRelationInfo *fpinfo = (MyFdwRelationInfo *) rel->fdw_private;
    fpinfo->qp_is_pushdown_safe = false;

    if (!root->query_pathkeys)
        return NIL;

    ListCell   *lc;
    foreach(lc, root->query_pathkeys) {
        PathKey    *pathkey = (PathKey *) lfirst(lc);
        EquivalenceClass *pathkey_ec = pathkey->pk_eclass;
        if (!is_foreign_expr(get_expr(pathkey_ec)))
            return NIL;
    }

    fpinfo->qp_is_pushdown_safe = true;
    return list_make1(list_copy(root->query_pathkeys));
}
```

Only if all shippable

🤝 GetForeignJoinPaths

Create possible access paths for a JOIN

Determine whether JOIN can be pushed down

- JOIN type
- Prefer remote conditions
- No local OUTER JOIN conditions
- Local INNER JOIN conditions okay

Add paths for JOIN pushdown

- Avoid remote full table scans
- Set lower costs to encourage pushdown
- Add foreign scan or join scan paths for each

Examine sorting expressions (PathKeys)

Add paths for sorting pushdown

```
static void myGetForeignJoinPaths(
    PlannerInfo * root, RelOptInfo * joinrel, RelOptInfo * outerrel,
    RelOptInfo * innerrel, JoinType jointype, JoinPathExtraData * extra
) {
    MyFdwRelationInfo *fpinfo = palloc0(sizeof(MyFdwRelationInfo));
    joinrel->fdw_private = fpinfo;

    if ( foreign_join_ok(root, joinrel, jointype, outerrel, innerrel, extra))
        return;

    fpinfo->pushdown_safe = true;
    fpinfo->width = 32;
    fpinfo->startup_cost = 1.0;
    fpinfo->total_cost = 0.1;

    joinrel->rows = 1000;
    joinrel->reltarget->width = fpinfo->width;

    add_path(joinrel, (Path *) create_foreign_join_path(
        joinrel, NULL, fpinfo->width, 0, fpinfo->startup_cost,
        fpinfo->total_cost NIL, NULL, NULL, NIL, NIL
    ));

    add_paths_with_pathkeys_for_rel(root, joinrel);
}
```

Same as before

```
static bool foreign_join_ok(
    PlannerInfo * root, RelOptInfo * joinrel, JoinType jointype,
    RelOptInfo * outerrel, RelOptInfo * innerrel, JoinPathExtraData * extra
) {
    if (jointype != JOIN_INNER && jointype != JOIN_LEFT &&
        jointype != JOIN_RIGHT)
        return false;

    MyFdwRelationInfo *fpinfo = joinrel->fdw_private;
    MyFdwRelationInfo *fpinfo_o = outerrel->fdw_private;
    MyFdwRelationInfo *fpinfo_i = innerrel->fdw_private;

    if (!can_pushdown_relations(fpinfo_o, fpinfo_i))
        return false;
```

Both pushdown_safe & no local conditions

```
joinclauses = NIL;
foreach(lc, extra->restrictlist) {
    RestrictInfo *rinfo = lfirst_node(RestrictInfo, lc);
    bool is_remote_clause = is_foreign_expr(root, joinrel, rinfo->clause);

    if (is_pushable_outer_join(jointype, rinfo, joinrel->relids)) {
        if (!is_remote_clause)
            return false;
        joinclauses = lappend(joinclauses, rinfo);
    } else {
        if (is_remote_clause)
            fpinfo->remote_conds = lappend(fpinfo->remote_conds, rinfo);
        else
            fpinfo->local_conds = lappend(fpinfo->local_conds, rinfo);
    }
}

if (!can_pushdown_target_list(root, fpinfo))
    return false;

/* Save the join clauses for later use. */
fpinfo->joinclauses = joinclauses;
```

No PlaceholderVar except at top of JOIN tree

```
switch (jointype) {
case JOIN_INNER:
    fpinfo->remote_conds = list_concat(
        fpinfo->remote_conds, list_copy(fpinfo_i->remote_conds)
    );
    fpinfo->remote_conds = list_concat(
        fpinfo->remote_conds, list_copy(fpinfo_o->remote_conds)
    );
    break;

case JOIN_LEFT:
    fpinfo->joinclauses = list_concat(
        fpinfo->joinclauses, list_copy(fpinfo_i->remote_conds)
    );
    fpinfo->remote_conds = list_concat(
        fpinfo->remote_conds, list_copy(fpinfo_o->remote_conds)
    );
    break;
```

```
    case JOIN_RIGHT:
        fpinfo->joinclauses = list_concat(
            fpinfo->joinclauses, list_copy(fpinfo_o->remote_conds)
        );
        fpinfo->remote_conds = list_concat(
            fpinfo->remote_conds, list_copy(fpinfo_i->remote_conds)
        );
        break;
    default:
        /* Should not happen, checked above */
        elog(ERROR, "unsupported join type %d", jointype);
    }

    return true;
}
```

🆙 GetForeignUpperPaths

Create possible access paths for a Upper relation processing

- aggregation, grouping, window functions, sorting

Determine type of upper relation

Aggregation & Grouping

- Can `GROUP BY` expressions push down?
- Can Aggregate function push down?
- Can args to function push down?

Window Functions:

- Can window function push down?
- Can args to function push down?

`ORDER BY`

- Can expression push down?
- Maybe already okay for rels & joinrels

Final processing (`LIMIT`, `OFFSET`)

- Can expressions push down?

```
static void myGetForeignUpperPaths(
    PlannerInfo * root, UpperRelationKind stage,
    RelOptInfo * input_rel, RelOptInfo * output_rel, void *extra
) {
    /* Skip if cannot push down inner relation.
    if (!input_rel->fdw_private ||
        !((MyFdwRelationInfo *) input_rel->fdw_private)->pushdown_safe)
        return;

    /* Skip if already seen outer relation. */
    if (output_rel->fdw_private)
        return;

    /* Skip if unsupported upper relation stage. */
    if (   stage != UPPERREL_GROUP_AGG
        && stage != UPPERREL_ORDERED
        && stage != UPPERREL_FINAL
    )
        return;
```

```
    MyFdwRelationInfo *fpinfo = palloc0(sizeof(MyFdwRelationInfo));
    fpinfo->pushdown_safe = false;
    fpinfo->stage = stage;
    output_rel->fdw_private = fpinfo;

    switch (stage) {
    case UPPERREL_GROUP_AGG:
        return add_foreign_grouping_paths(root, input_rel, output_rel, extra);
    case UPPERREL_ORDERED:
        return add_foreign_ordered_paths(root, input_rel, output_rel);
    case UPPERREL_FINAL:
        return add_foreign_final_paths(root, input_rel, output_rel, extra);
    default:
        elog(ERROR, "unexpected upper relation: %d", (int) stage);
    }
}
```

```
static void add_foreign_grouping_paths(
    PlannerInfo * root, RelOptInfo * input_rel,
    RelOptInfo * grouped_rel, GroupPathExtraData * extra
) {
    Query       *parse = root->parse;
    MyFdwRelationInfo *ifpinfo = input_rel->fdw_private;
    MyFdwRelationInfo *fpinfo = grouped_rel->fdw_private;

    /* Nothing to be done if no grouping or aggregation required. */
    if (!parse->groupClause && !parse->groupingSets
        && !parse->hasAggs && !root->hasHavingQual
    )
        return;

    if (!foreign_grouping_ok(root, grouped_rel, extra->havingQual))
        return;

    /* Create and add foreign path to the grouping relation. */
    add_path(grouped_rel, (Path *) create_foreign_upper_path(
        root, grouped_rel, grouped_rel->reltarget,
        1000, 0, 1.0, 0.2, NIL, NULL, NIL, NIL
    ));
}
```

```
static bool foreign_grouping_ok(
    PlannerInfo * root, RelOptInfo * grouped_rel, Node * havingQual
) {
    Query       *query = root->parse;
    MyFdwRelationInfo *fpinfo = grouped_rel->fdw_private;
    PathTarget *grouping_target = grouped_rel->reltarget;

    /* We currently don't support pushing Grouping Sets. */
    if (query->groupingSets)
        return false;

    /* Get the fpinfo of the underlying scan relation. */
    MyFdwRelationInfo *ofpinfo = fpinfo->outerrel->fdw_private;
    if (ofpinfo->local_conds)
        return false;
```

```
ListCell *lc,
List *tlist = NIL;
foreach(lc, grouping_target->exprs) {
    Expr        *expr = (Expr *) lfirst(lc);
    Index        sgref = get_pathtarget_sortgroupref(grouping_target, i);

   if (sgref && get_sortgroupref_clause_noerr(sgref, query->groupClause)) {
        if (!is_foreign_expr(expr))
            return false;
        tlist = lappend(tlist, makeTargetEntry(expr, list_length(tlist) + 1, NULL, false));
    }
    else {
       if (is_foreign_expr(expr))
           tlist = add_to_flat_tlist(tlist, list_make1(expr));
       else {
           List *aggvars = pull_var_clause((Node *) expr, PVC_INCLUDE_AGGREGATES);

           if (!is_foreign_expr((Expr *) aggvars))
               return false;

           tlist = append_aggvars(tlist, aggvars);
        }
    }
}
```

```
    if (havingQual)
    {
        ListCell   *having_lc;
        foreach(having_lc, (List *) havingQual) {
            Expr       *expr = (Expr *) lfirst(having_lc);
            RestrictInfo *rinfo = make_restrictinfo(
                root, expr, true, false, false, false,
                root->qual_security_level, grouped_rel->relids, NULL, NULL
            );

            if (is_foreign_expr(expr))
                fpinfo->remote_conds = lappend(fpinfo->remote_conds, rinfo);
            else
                fpinfo->local_conds = lappend(fpinfo->local_conds, rinfo);
        }
    }

    /* Safe to pushdown */
    fpinfo->pushdown_safe = true;
    return true;
}
```

👣 Paths

So far just costs & paths

- GetForeignRelSize
- GetForeignPaths
- GetForeignJoinPaths
- GetForeignUpperPaths

All to identify pushdown

And inform the planner

Common pattern...

```
if (!can_pushdown_feature(feature))
    return NULL;

foreach(expr, expressions) {
    if (!is_foreign_expr(expr))
        return NULL;
}

return paths_for(rel, cost(rel));
```

🛳️ is_foreign_expr()

Takes a node argument

Examines node type

Determines shippability

Recurses into child nodes

Squirrels away private info

Returns true or false

```c
static bool is_foreign_expr(Node * node) {
    switch (nodeTag(node)) {
    case T_DistinctExpr:
        OpExpr      *oe = (OpExpr *) node;
        return is_foreign_expr((Node *) oe->args);
    case T_Var:
        /* .... */
        return true;
    case T_Const:
        /* .... */
        return true;
    case T_FuncExpr:
        /* .... */
        return true;
   default:
        return false;
    }
}
```

Nodes

21 in original clickhouse_fdw

35 in pg_clickhouse today

29 in postgres_fdw

Postgres 18 defines 479!

Not all relevant

Still a looooong way to go

🚦 GetForeignPlan

Called at the end of planning

Sort `WHERE` and `JOIN` expressions into local and remote

Build list of columns to fetch from remote

Generate (“deparse”) the SQL statement from plan nodes

Create foreign scan node

```
static ForeignScan * myGetForeignPlan(
    PlannerInfo * root, RelOptInfo * foreignrel,
    Oid foreigntableid, ForeignPath * best_path,
    List * tlist, List * scan_clauses, Plan * outer_plan
) {
    MyFdwRelationInfo *fpinfo = foreignrel->fdw_private;
    List       *remote_exprs = extract_actual_clauses(fpinfo->remote_conds, false);
    List       *local_exprs = extract_actual_clauses(fpinfo->local_conds, false);

    StringInfoData sql;
    initStringInfo(&sql);
    my_deparse_select_stmt_for_rel(
        &sql, root, foreignrel, tlist, remote_exprs, best_path->path.pathkeys
    );

    List *fdw_private = list_make3(
        makeString(sql.data), tlist, makeInteger(fpinfo->fetch_size)
    );

    return make_foreignscan(
        tlist, local_exprs, 0, NULL, fdw_private, tlist, NULL, outer_plan
    );
}
```

```
void my_deparse_select_stmt_for_rel(
    StringInfo buf, PlannerInfo * root, RelOptInfo * rel,
    List * tlist, List * remote_conds, List * pathkeys,
    bool has_final_sort, bool has_limit, bool is_subquery,
    List * *retrieved_attrs, List * *params_list
) {
    deparse_expr_cxt context;
    CHFdwRelationInfo *fpinfo = rel->fdw_private;
    List       *quals;
    /* Fill portions of context common to upper, join and base relation. */
    context.buf = buf;
    context.root = root;
    context.foreignrel = rel;
    context.scanrel = IS_UPPER_REL(rel) ? fpinfo->outerrel : rel;
    context.params_list = params_list;
    context.func = NULL;
    context.interval_op = false;
    context.array_as_tuple = false;
    context.no_sort_parens = false;

    /* Construct SELECT clause */
    deparseSelectSql(tlist, is_subquery, retrieved_attrs, &context);
```

```
/*
 * For upper relations, WHERE built from remote conditions of scan relation;
 * otherwise the supplied list of remote conditions directly.
 */
if (IS_UPPER_REL(rel)) {
    CHFdwRelationInfo *ofpinfo = fpinfo->outerrel->fdw_private;
    quals = ofpinfo->remote_conds;
} else
    quals = remote_conds;

/* Construct FROM and WHERE clauses */
deparseFromExpr(quals, &context);

if (IS_UPPER_REL(rel)) {
    /* Append GROUP BY clause */
    appendGroupByClause(tlist, &context);

    /* Append HAVING clause */
    if (remote_conds) {
        appendStringInfoString(buf, " HAVING ");
        appendConditions(remote_conds, &context);
    }
}
```

```
    /* Add ORDER BY clause if we found any useful pathkeys */
    if (pathkeys)
        appendOrderByClause(pathkeys, has_final_sort, &context);

    /* Add LIMIT clause if necessary */
    if (has_limit)
        appendLimitClause(&context);
}


static void appendLimitClause(deparse_expr_cxt * context) {
    PlannerInfo *root = context->root;
    StringInfo  buf = context->buf;

    if (root->parse->limitCount) {
        appendStringInfoString(buf, " LIMIT ");
        deparseExpr((Expr *) root->parse->limitCount, context);
    }
    if (root->parse->limitOffset) {
        appendStringInfoString(buf, " OFFSET ");
        deparseExpr((Expr *) root->parse->limitOffset, context);
    }
}
```

Complement to is_foreign_expr()

```
static void deparseExpr(Expr * node, deparse_expr_cxt * context) {
    if (node == NULL)
        return;

    switch (nodeTag(node)) {
    case T_DistinctExpr:
        deparseDistinctExpr((DistinctExpr *) node, context);
        break;
    case T_Var:
        deparseVar((Var *) node, context);
        break;
    case T_Const:
        deparseConst((Const *) node, context, 0);
        break;
    case T_FuncExpr:
        deparseFuncExpr((FuncExpr *) node, context);
        break;
    default:
        elog(ERROR, "unsupported expression type for deparse: %d",
                (int) nodeTag(node));
        break;
    }
}
```

```
static void deparseDistinctExpr(
    DistinctExpr * node, deparse_expr_cxt * context
) {
    StringInfo  buf = context->buf;
    Assert(list_length(node->args) == 2);

    appendStringInfoChar(buf, '(');
    deparseExpr(linitial(node->args), context);
    appendStringInfoString(buf, " IS DISTINCT FROM ");
    deparseExpr(lsecond(node->args), context);
    appendStringInfoChar(buf, ')');
}
```

Foreign Query Syntax

 Wrapper Unwrapped

No FDW pushdown is “complete”

postgres_fdw the leading example

- No window function pushdown
- No UNION query pushdown
- etc.

👍🏼 Full Scan Always Correct

But defaults work great

- As long as at least full scan path
- Postgres fetches data it needs
- Evaluates locally

Slower, but correct

🤝 FDW Negotiation

Even base FDW works

Improving pushdown is a negotiation

- Between Postgres syntax
- And foreign syntax

Add more mappings over time

Improve pushdown over time

 The Process

Customer moved data to ClickHouse

Pointed Postgres queries at foreign tables

A few much slower than before

Needed regex & array functions, `CURRENT_DATE` pushdown

```
EXPLAIN (VERBOSE, COSTS OFF)
SELECT * FROM t1 WHERE c < CURRENT_DATE;

                          QUERY PLAN
---------------------------------------------------------
 Foreign Scan on public.t1
   Output: a, b, c
   Filter: (t1.c < CURRENT_DATE)
   Remote SQL: SELECT a, b, c FROM functions_test.t1
```

📅 Example: CURRENT_DATE

Inherited internal function pushdown

Good for identical local & foreign functions (ex `abs()`)

Did not push down:

  CURRENT_DATE, CURRENT_TIMESTAMP, CURRENT_USER

Reason: not functions, but “SQL value Functions”

Different node type

```
static bool is_foreign_expr(Node * node) {
    switch (nodeTag(node)) {
    case T_DistinctExpr:
        OpExpr      *oe = (OpExpr *) node;
        return is_foreign_expr((Node *) oe->args);
    case T_Var:
        /* .... */
        return true;
    case T_Const:
        /* .... */
        return true;
    case T_FuncExpr:
        /* .... */
        return true;
    case T_SQLValueFunction:
        return false;
    default:
        return false;
    }
}
```

```
static void deparseExpr(Expr * node, deparse_expr_cxt * context) {
    switch (nodeTag(node)) {
     case T_DistinctExpr:
         deparseDistinctExpr((DistinctExpr *) node, context);
         break;
     case T_Var:
         deparseVar((Var *) node, context);
         break;
     case T_Const:
         deparseConst((Const *) node, context, 0);
         break;
     case T_FuncExpr:
         deparseFuncExpr((FuncExpr *) node, context);
         break;
     case T_SQLValueFunction:
         deparseSQLValueFunction((SQLValueFunction *) node, context);
         break;
     default:
         elog(ERROR, "unsupported expression type for deparse: %d",
              (int) nodeTag(node));
         break;
     }
}
```

```c
#define QUOTED_TZ ch_quote_literal(pg_get_timezone_name(session_timezone))

static void deparseSQLValueFunction(
    SQLValueFunction * svf, deparse_expr_cxt * context
) {
    StringInfo  buf = context->buf;
    LOCAL_FCINFO(fcinfo, 0);
    Datum       datum;

    switch (svf->op) {
    case SVFOP_CURRENT_DATE:
        appendStringInfo(buf, "toDate(now(%s))", QUOTED_TZ);
        break;
    case SVFOP_CURRENT_TIME:
    case SVFOP_LOCALTIME:
        appendStringInfo(buf, "toTime64(now64(6, %s), 6)", QUOTED_TZ);
        break;
    case SVFOP_CURRENT_TIME_N:
    case SVFOP_LOCALTIME_N:
        appendStringInfo(buf, "toTime64(now64(%1$d, %2$s), %1$d)", svf->typmod, QUOTED_TZ);
        break;
    case SVFOP_CURRENT_TIMESTAMP:
    case SVFOP_LOCALTIMESTAMP:
        appendStringInfo(buf, "now64(6, %s)", QUOTED_TZ);
        break;
    case SVFOP_CURRENT_TIMESTAMP_N:
    case SVFOP_LOCALTIMESTAMP_N:
        appendStringInfo(buf, "now64(%d, %s)", svf->typmod, QUOTED_TZ);
        break;
```

```
        case SVFOP_CURRENT_ROLE:
        case SVFOP_CURRENT_USER:
        case SVFOP_USER:
           datum = current_user(fcinfo);
            if (fcinfo->isnull)
                appendStringInfoString(buf, "NULL");
            else
                appendStringInfoString(buf, ch_quote_literal(DatumGetCString(datum)));
            break;
        case SVFOP_SESSION_USER:
           datum = session_user(fcinfo);
            if (fcinfo->isnull)
                appendStringInfoString(buf, "NULL");
            else
                appendStringInfoString(buf, ch_quote_literal(DatumGetCString(datum)));
            break;
        case SVFOP_CURRENT_CATALOG:
           datum = current_database(fcinfo);
            appendStringInfoString(buf, ch_quote_literal(DatumGetCString(datum)));
            break;
        case SVFOP_CURRENT_SCHEMA:
           datum = current_schema(fcinfo);
            if (fcinfo->isnull)
                appendStringInfoString(buf, "NULL");
            else
                appendStringInfoString(buf, ch_quote_literal(DatumGetCString(datum)));
            break;
        default:
            elog(ERROR, "unknown SQL Value function: %i", svf->op);
        }
    }
```

```
EXPLAIN (VERBOSE, COSTS OFF)
SELECT * FROM t1 WHERE c < CURRENT_DATE;

                              QUERY PLAN
-----------------------------------------------------------------------
 Foreign Scan on public.t1
   Output: a, b, c
   Remote SQL: SELECT a, b, c FROM functions_test.t1
                WHERE ((c < toDate(now('Asia/Tokyo'))))
(3 rows)
```

4️⃣ Four Categories

Of pushdown work

1. Mapping Postgres features to foreign equivalents

   ex: Function, JOIN Pushdown
2. Adding Postgres objects for compatibility

   ex: RE2 (full support), `toUInt*()` (no-op placeholders)
3. Adding Postgres features for new mappings

   ex: UNION Pushdown
4. Incompatibilities remain local

   ex: 3-arg `array_sort()`, `range_agg()`

Add features to ClickHouse?

Thank You

The Pushdown: Building a Foreign Data Wrapper

David E. Wheeler
ClickHouse, PGXN

PGConf.dev
2026–05–21