Implementing DDL Deparsing and DDL Replication

Masahiko Sawada

Senior Software Development Engineer @ AWS
PostgreSQL Major Contributor & Committer

DDL Replication : what, why, and history

What Is DDL Replication?

- DDL changes (CREATE / ALTER / DROP) are not replicated today
- Users must run DDL on every node manually
- DDL replication = automatic propagation of schema changes

The Most Common Use Case: Major Version Upgrade

- Build a new-version cluster, replicate from the old one
- Cut over when the new cluster catches up
- Problem: no DDL allowed during the upgrade window
  - “Zero-Downtime Upgrades: PostgreSQL and OS/glibc at Global Scale”[1]
- No new tables, no index changes, no partition maintenance

[1] https://www.postgresql.eu/events/fosdem2026/sessions/session/7370/slides/816/2026-FOSDEM_%20Zero-Downtime%20Upgrades_%20PostgreSQL%20and%20OS_glibc%20at%20Global%20Scale-EXTENDED.pdf

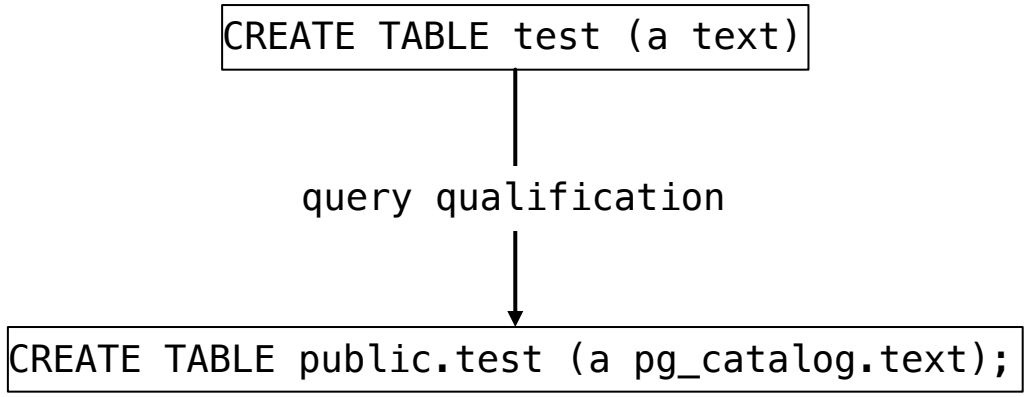
History of Developments

- 2012 - 2015: “deparsing utility commands”[1][2][3]
  - Authors: Álvaro Herrera, Dimitri Fontaine
  - Started with Event Triggers + qualified SQL; later pivoted to JSON form
  - Only fragments (related event trigger changes) were committed
- 2017 - built-in logical replication shipped in PostgreSQL 10
- 2022-2023: “Support logical replication of DDLs”[4]
  - Authors: Zheng Li, Shveta Malik, Zhijie Hou
  - Deparsing approach + logical replication support

[1] https://www.postgresql.org/message-id/m2txrsdzxa.fsf@2ndQuadrant.fr
[2] https://www.postgresql.org/message-id/20131108153322.GU5809%40eldon.alvh.no-ip.org
[3] https://www.postgresql.org/message-id/20150215044814.GL3391%40alvh.no-ip.org
[4] https://www.postgresql.org/message-id/CAAD30U%2BpVmfKwUKy8cbZOnUXyguJ-uBNejwD75Kyo%3DOjdQGJ9g%40mail.gmail.com

Functional Requirements

- All identifiers must be schema-qualified
  - Sending the executed query + search_path is not enough
- Scope - at minimum, table DDLs (CREATE/ALTER/DROP TABLE)
- Support cross-version logical replication
- Existing filters (table, schema) still work
- Plugin-independent
  - Third-party output plugins can use it



Ideas

Approach 1: DDL Deparse

- Parse tree + system catalog lookups -> fully qualified SQL
- Output format: self-documenting JSON blob
- DDL replication integration (proposed approach)
  - Captures DDL events via event triggers
  - Event trigger function deparses and writes JSON to WAL
  - Receiver reconstructs DDL from JSON blob

Example: CREATE TABLE s1.test (a int ,b int)

```
{
  "fmt": "CREATE TABLE %{identity}D (%{table_elements:, }s)",
  "myowner": "masahiko",
  "identity": {
    "objname": "test",
    "schemaname": ”s1”
  },
  "table_elements": [
  {
    "fmt": "%{name}I %{coltype}T STORAGE%{colstorage}s",
    "name": "a",
    "type": "column",
    "coltype": {
      "typmod": "",
      "typarray": false,
      "typename": "int4",
      "schemaname": "pg_catalog"
    },
    "colstorage": "PLAIN”
  },
  {
    "fmt": "%{name}I %{coltype}T STORAGE %{colstorage}s",
    "name": "b",
    "type": "column",
    "coltype": {
      "typmod": "",
      "typarray": false,
      "typename": "int4",
      "schemaname": "pg_catalog"
    },
    “colstorage": "PLAIN”
} ] }
```

Approach 1: Strengths

- Deparsing is useful on its own
- Event Triggers already exists - capture side is light
- Self-documenting JSON - easy to rebuild SQL on the receiver
- Flexible - e.g., rewrite schema names in transit

Approach 1: Concerns

- High maintenance costs
  - every parse-node changes -> update JSON generation code
- Event triggers depend on PUBLICATIONs
  - third-party plugins cannot easily use it

Approach 2: Use pg_get_*_ddl() functions

```
=# select pg_get_database_ddl('postgres'::regdatabase);
                                          pg_get_database_ddl
------------------------------------------------------------------------------------------
 CREATE DATABASE postgres WITH TEMPLATE = template0 ENCODING = 'UTF8' LOCALE_PROVIDER = libc LOCALE = 'C';
 ALTER DATABASE postgres OWNER TO masahiko;
(2 rows)
```

Approach 2: Use pg_get_table_ddl() function

- Write table OID to WAL and use `pg_get_table_ddl()` to reconstruct DDL from catalogs for generating `CREATE TABLE`
  - `DROP TABLE` - usually just an object name; trivial
  - `ALTER TABLE` - use the sub-command parse node, deparse just that
- DDL replication integration
  - No Event Trigger - write WAL inside each DDL command
  - Read catalogs via historical snapshot at decode time

Approach 2: Strengths

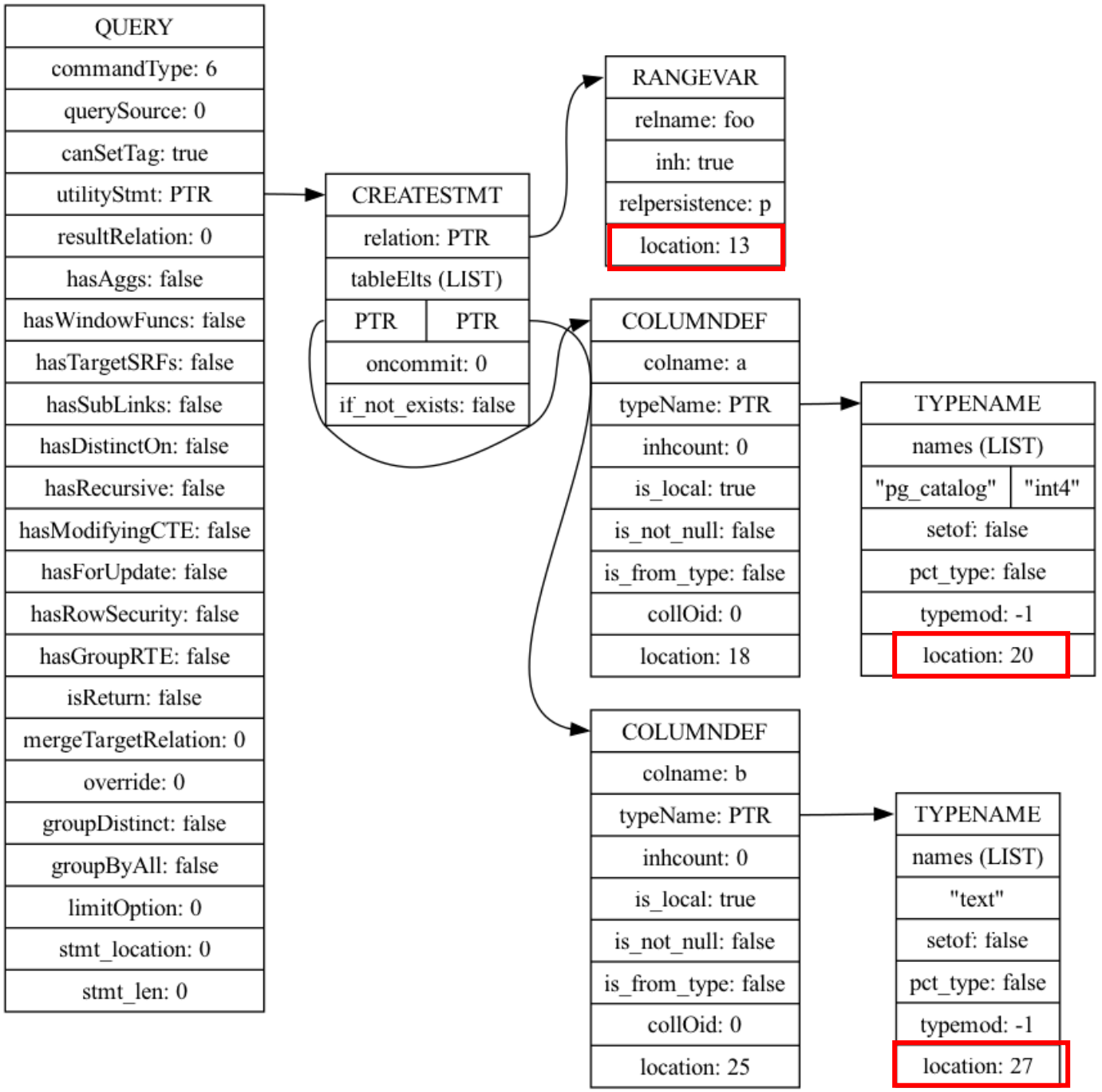
- `pg_get_*_ddl()` is also useful as a SQL function
- All plugins can use it when `wal_level` is ‘logical’, not just `pgoutput`

Approach 2: Concerns

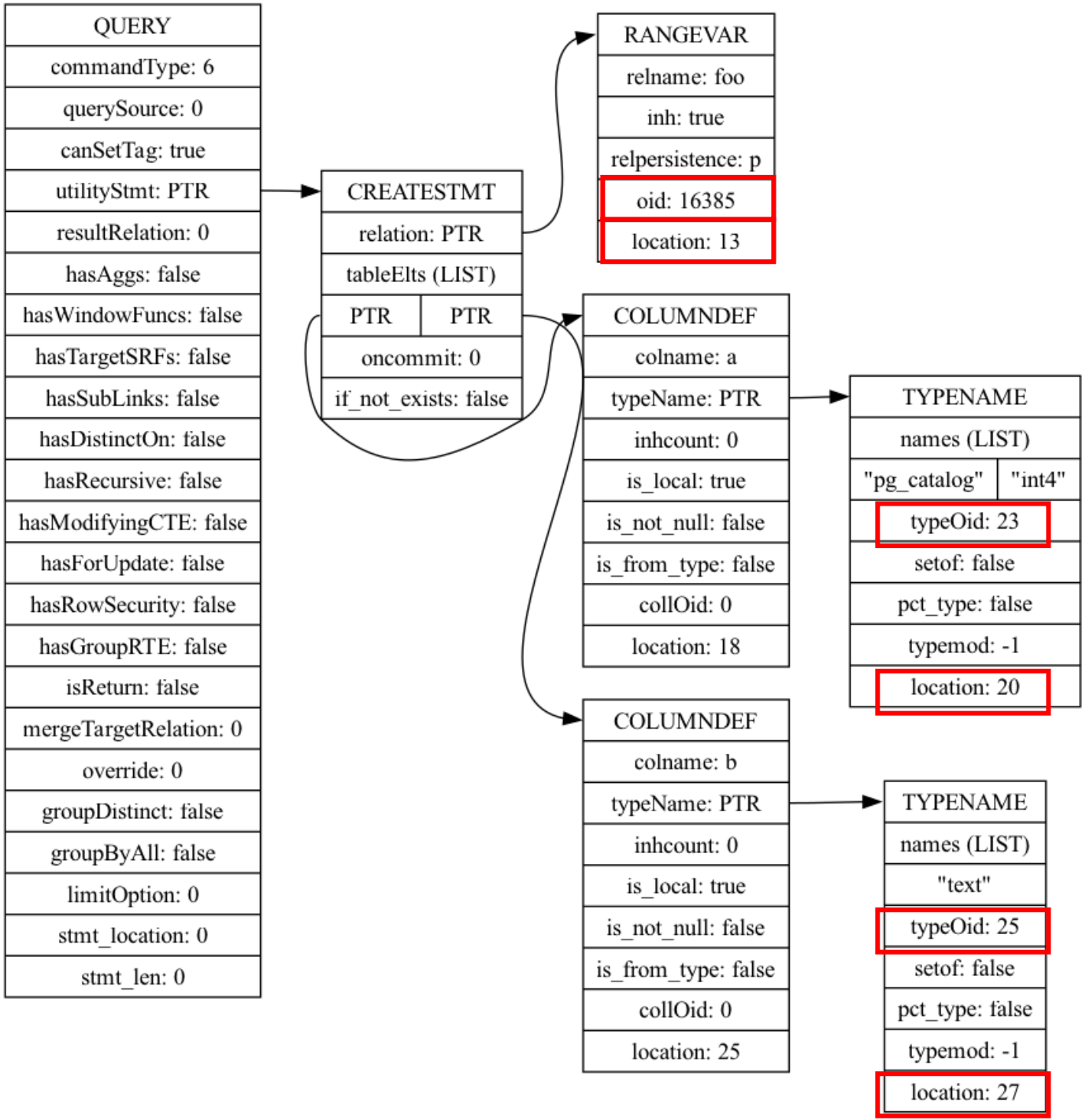
- Generated DDL may not be equal to the original DDL
  - Same effect, but possibly different statements
  - e.g., `CREATE TABLE foo (id SERIAL PRIMARY KEY, …)` becomes:
    - `CREATE TABLE foo`
    - `CREATE SEQUENCE foo_id_seq`
    - `ALTER TABLE ADD CONSTRAINT etc.`
- No JSON-style flexibility

Approach 3: In-place Query Qualification

- Walk a parse tree and collect the location and OID of objects
- Qualify the objects based on the locations and OIDs

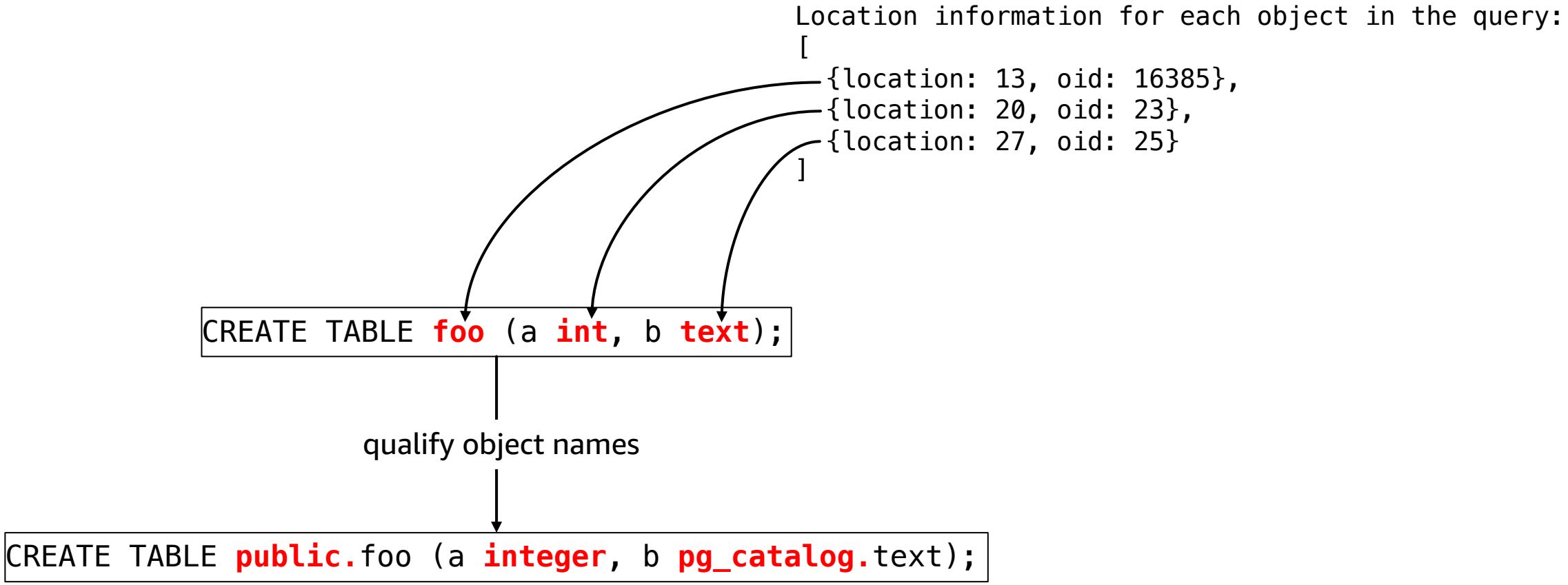



```
CREATE TABLE foo (a int, b text);
```




```
CREATE TABLE foo (a int, b text);
```

Approach 3: In-place Query Qualification



Approach 3: Auto-generation

```
typedef struct CreateStmt
{
   :
   RangeVar   *relation pg_node_attr(query_qualify_node);
   List       *tableElts pg_node_attr(query_qualify_node);
   :
} AlterTableStmt;


typedef struct ColumnDef
{
   :
   TypeName   *typeName pg_node_attr(query_qualify_node);
   :
} ColumnDef;

typedef struct TypeName
{
   :
   Oid typeOid pg_node_attr(query_qualify_oid);
   :
} TypeName
```

gen_node_support.pl

```
static void
_qualifyCreateStmt(QualifyState *state, Node *node)
{
    CreateStmt *expr = (CreateStmt *) node;

    COLLECT_NODE(expr->relation);
    COLLECT_NODE(expr->cmds);
}

static void
_qualifyColumnDef(QualifyState *state, Node *node)
{
    CreateStmt *expr = (CreateStmt *) node;

    COLLECT_NODE(expr->typeName);
    :
}

static void
_qualifyTypeName(QualifyState *state, Node *node)
{
    CreateStmt *expr = (CreateStmt *) node;

    COLLECT_OID(expr, exprt->typeOid);
}
```

```
:
    case T_AlterTableStmt:
        _qualifyAlterTableStmt(state, node);
        break;
    case T_ColumnDeft:
        _qualifyColumnDef(state, node);
        break;
    case T_TypeName:
        _qualifyTypeName(state, node);
        break;
:
```

Approach 3: Strengths

- Auto-generation support
- Preserves original DDL structure
- Can be used for table/schema mapping

Approach 3: Concerns

- Resolved OID should be back to parse tree nodes
  - Possibly need to change DDL execution codes much

Summary

- The hard part is how to/where generate a qualified SQL
- Each approach makes a different trade-off
- Discussion thread: ”Support logical replication of DDLs, take2” [1]

[1] https://www.postgresql.org/message-id/CAD21AoCzT3sytVbimRNdjRF%3DN3R-8ddaWKW95EzsdderXqcm4g%40mail.gmail.com

Thank you!

Masahiko Sawada
sawadamm@amazon.com
sawada.mshk@gmail.com