Understanding logical decoding and replication

Postgres Open, Chicago, 2014/09/19

Michael Paquier, Member of Technical Staff – PostgreSQL
About your lecturer

- **Michael Paquier**
  - Working on Postgres for Vmware
  - Community hacker and blogger
  - Based in Tokyo

- **Contacts:**
  - Twitter: @michaelpq
  - Email: mpaquier@vmware.com, michael@otacoo.com
Agenda

- Introduction to logical decoding
- Output decoders...
- ... And logical receivers
- And then...
Introduction to logical decoding
Before coming to it…

- **WAL = Write-ahead Log**
  - Internal journal of Postgres to maintain data integrity
  - Used for recovery, archives, etc.
  - LSN = Log Sequence Number, or WAL record position

- **WAL sender**
  - Process on root node sending WAL stream
  - On master or standby (cadcading)

- **WAL receiver**
  - Process on standby node receiving WAL stream
  - On standby

- **Replication protocol, set of commands to control replication**
  - Used internally for replication, externally as well with replication connections
  - [http://www.postgresql.org/docs/devel/static/protocol-replication.html](http://www.postgresql.org/docs/devel/static/protocol-replication.html)
What is logical decoding?

- Newly introduced in 9.4 (release Q4 of 2014)
- Plugin infrastructure
  - Customizable
  - Extensible
  - Adaptable
  - No need to modify core code
- Use cases
  - Replication solutions (Slony…)
  - Auditing
  - Online upgrade
- Result of hundreds of emails
- Introduction of many new features and principles…
Concept of logical decoding

- Decode WAL to get DML changes (INSERT, DELETE and UPDATE)
- Shape changes as desired and stream them
- Get changes and apply them on a remote source
Replication slots (1)

- Store WAL as long as changes are not consumed
- Can be used by one single WAL sender at the same time
- Careful: space consumption for pg_xlog partition if used
- System view pg_replication_slots

Physical slots

- System-wide, conflict resolution with oldestXmin in feedback message
- For recovery: primary_slot_name in recovery.conf
- Creation:
  - SELECT pg_create_physical_replication_slot('slot_name')
  - Replication protocol: CREATE_REPLICATION_SLOT foo PHYSICAL
- Configuration: max_replication_slots > 0
- Drop:
  - SELECT pg_drop_replication_slot('slot_name')
  - Replication protocol DROP_REPLICATION_SLOT foo
Replication slots (2)

- **Logical**
  - Attached to a database
  - Need a decoder plugin to reshape changes when requested
  - Cannot be used for recovery
  - Creation
    - `pg_create_logical_replication_slot('slot_name', 'plugin_name')`
    - `CREATE_REPLICATION_SLOT foo LOGICAL plugin`
  - Configuration: `max_replication_slots > 0 + wal_level = logical`

```
=# SELECT * FROM
   pg_create_physical_replication_slot('physical_slot');
 slot_name    | xlog_position
--------------+---------------
 physical_slot | null          (1 row)

=# SELECT * FROM
   pg_create_logical_replication_slot('logical_slot', 'test_decoding');
 slot_name    | xlog_position
--------------+---------------
 logical_slot | 0/5000100     (1 row)
```
Exported snapshots – Obtain it

- Can be used to retrieve consistent image of database

- Export with replication connection
  - In result of CREATE_REPLICATION_SLOT
  - Available for duration of replication connection

- Export with vanilla connection
  - SELECT pg_export_snapshot();
  - Available for duration of transaction calling function, not connection!

- No snapshot available with pg_create_logical_replication_slot()

- Maintain connection/transaction for duration as long as necessary
Exported snapshots – Dump consistent data

- Single transaction
- Use with SET TRANSACTION SNAPSHOT
- Limitations
  - Need to be tightly linked with application
  - pg_dump offers no real solutions in 9.4

1) Export Snapshot

```bash
$ psql "replication=database dbname=postgres"
=# CREATE_REPLICATION_SLOT logical_slot
   LOGICAL test_decoding;
-[ RECORD 1 ]-----------------
   slot_name | logical_slot
   consistent_point | 0/5000E58
   snapshot_name | 000003F0-1
   output_plugin | test_decoding
```

2) Fetch data

```bash
$ psql postgres
=# BEGIN ISOLATION LEVEL
   REPEATABLE READ;
BEGIN
=# SET TRANSACTION
   SNAPSHOT '000003F0-1';
SET
=# [stuff]
=# COMMIT;
```
Output decoder

- Decodes WAL from logical replication slot
- Plugin to be added on server side
- Used in WAL sender if changes streamed with replication protocol
- Output can be queried with SQL functions
- 1 change per tuple modified
  - Good for OLTP and short transactions
  - Less for warehouse, bulk writes…
- Postgres ships one: test_decoding
- Can use custom options improving output granularity
- Documentation:
REPLICA IDENTITY

- Change information verbosity of old rows being updated or deleted

- Different modes
  - DEFAULT, use PRIMARY KEY if any
  - USING INDEX index_name
    - Unique, not partial, no expression, no NOT NULL columns
    - Same as DEFAULT with PRIMARY KEY
  - ALL, old values of all columns
  - NOTHING
    - No values recorded
    - Same as DEFAULT without PRIMARY KEY

- SQL level
  - CREATE TABLE sets it to DEFAULT
  - ALTER TABLE to change it
Logical change receiver

- **Runs on client side**
  - Anything able to connect to Postgres node with replication protocol
  - In short, something able to fetch changes and process them
  - Can use options of decoder for custom output

- **SQL interface**
  - Textual format
    - `pg.logical_slot.get_changes` to consume
    - `pg.logical_slot.peek_changes` to look at
  - Binary format
    - `pg.logical_slot.get_binary_changes` to consume
    - `pg.logical_slot.peek_binary_changes` to look at

- **Replication connection => mainly COPY protocol**
Logical decoding and replication

- **Replication connection**
  - Extended “replication” with mode “database” in 9.4
  - Need application_name for pg_stat_replication, dbname
  - Example:
    host=$IP replication=database dbname=my_db application_name=my_app

- **Queries**
  - IDENTIFY_SYSTEM (to get current LSN write position, timeline, system ID or connected database)
  - CREATE_REPLICATION_SLOT
  - DROP_REPLICATION_SLOT
  - START_REPLICATION SLOT slot_name LOGICAL [start_pos | 0/0]

- **Position 0/0**
  - oldest LSN position available in slot.
  - Not InvalidXLogRecPtr…
Logical decoding and replication (2)

- Use application_name in connection string
- standby_synchronous_names on master for synchronous receiver
- Feedback to master!
  - To release WAL files on a slot
  - flush_position, write_position useful (depends on synchronous_commit)
  - Message format
    - ‘r’ for message type
    - 8 bytes for write position (XLogRecPtr)
    - 8 bytes for flush position (XLogRecPtr)
    - 8 bytes for applied/replay position (XLogRecPtr)
    - 8 bytes for timestamp
    - 1 byte to request reply from server
Output decoders...
Basics

- **Set of callback functions for events:**
  - Startup (Initialization when opening slot)
  - Shutdown
  - BEGIN
  - COMMIT
  - Tuple change triggered by INSERT, UPDATE, DELETE

- **Example with decoder generating raw queries**

- **Available as decoder_raw here (PostgreSQL license):**
  - git clone [https://github.com/michaelpq/pg_plugins](https://github.com/michaelpq/pg_plugins)
  - cd pg_plugins/decoder_raw
Loading callbacks

- Loaded by _PG_output_plugin_init
  - Similar to _PG_init, but for decoder context
- Startup and shutdown can be NULL
- Begin, commit and change mandatory

```c
def void _PG_output_plugin_init(OutputPluginCallbacks *cb) {
    cb->startup_cb = decoder_raw_startup;
    cb->begin_cb = decoder_raw_begin_txn;
    cb->change_cb = decoder_raw_change;
    cb->commit_cb = decoder_raw_committxn;
    cb->shutdown_cb = decoder_raw_shutdown;
}
```
Callback - Initialization

- Initialize context and options
- Use ctx->output_plugin_private for parameters
- Output format: OUTPUT_PLUGIN_[BINARY|TEXTUAL]_OUTPUT

```c
static void
decoder_raw_startup(LogicalDecodingContext *ctx,
                     OutputPluginOptions *opt,
                     bool is_init)
{
    ListCell  *option;
    DecoderRawData *data;
    data = palloc(sizeof(TestDecodingData));
    data->context = AllocSetContextCreate(ctx->context,
                                           "Raw decoder context", ...);

    /* Options */
    foreach(option, ctx->output_plugin_options)
    {
        DefElem  *elem = lfirst(option);
        [...blah...]
    }
}
```
Callbacks - Shutdown

- Called each time replication connection ends...
- Or decoder context not needed
- Removal of initialization things

```c
static void
decoder_raw_shutdown(LogicalDecodingContext *ctx)
{
    DecoderRawData *data = ctx->output_plugin_private;

    /* cleanup our own resources via memory context reset */
    MemoryContextDelete(data->context);
}
```
.callbacks - begin

- Called each time decoding is done for a single record
- Somewhat similar to BEGIN transaction
- ReorderBufferTXN with information of transaction (txid, etc.)
- StringInfo of ctx->out
- OutputPluginPrepareWrite to prepare the field
- OutputPluginWrite to write change

```c
static void decoder_raw_begin_txn(LogicalDecodingContext *ctx,
                                   ReorderBufferTXN *txn)
{
    OutputPluginPrepareWrite(ctx, true);
    appendStringInfoString(ctx->out, "BEGIN;");
    OutputPluginWrite(ctx, true);
}
```
Callbacks - COMMIT

- Called each time decoding is finished for a single record
- Similar to COMMIT transaction, and previous BEGIN...
- commit_lsn = WAL position of this commit

```c
static void
decoder_raw_commit_txn(LogicalDecodingContext *ctx,
                        ReorderBufferTXN *txn,
                        XLogRecPtr commit_lsn)
{
    OutputPluginPrepareWrite(ctx, true);
    appendStringInfoString(ctx->out, "COMMIT;");
    OutputPluginWrite(ctx, true);
}
```
Callbacks – DML changes

- Called each time for each tuple changed
- Depending on query and REPLICA IDENTITY, old and new tuple data change
- For decoder_raw
  - WHERE clause of UPDATE and DELETE depends on REPLICA IDENTITY
  - Use relation->rd_rel->relreplident and relation->rd_repidindex!

```c
static void decoder_raw_change(LogicalDecodingContext *ctx, ReorderBufferTXN *txn, Relation relation, ReorderBufferChange *change)
{
    old = MemoryContextSwitchTo(data->context);
    switch (change->action)
    {
        case REORDER_BUFFER_CHANGE_INSERT:
        case REORDER_BUFFER_CHANGE_UPDATE:
        case REORDER_BUFFER_CHANGE_DELETE:
            break;
    }
}
```
So now..

- Hack your own decoders! Or contribute back.
- Use test_decoding in contrib/ as a base
  - Options present as a model
  - Able to manage field values correctly
  - Reuse and abuse of it
- Demonstration with SQL interface
- Remember:
  - 1 change per tuple
  - N tuples changed => more or less N output entries for single record + 2 (BEGIN + COMMIT)
... And logical receivers
With SQL interface

- SQL interface
- Primitive, maybe fine for simple cases
- Advantage
  - Light
  - Do SQL operations on output, leverage decoder effort to receiver
  - Replication slot changes automatically consumed and incremented
- Disadvantage
  - Lack of flexibility: IDENTIFY_SYSTEM, no flush and written position control
  - No replication async or even sync

```bash
#!/bin/bash
psql -c "SELECT pg_create_logical_replication_slot('slot', 'decoder_raw')"
while :
do
  psql -At -c "SELECT * FROM pg_logical_slot_get_changes('slot', NULL, 1)"
sleep 1
done
```
With replication protocol (1) – Open connection

- Open replication connection
- Use PGRES_COPY_BOTH to check result validity
- Possible to pass options

```c
/* Start logical replication at specified position */
appendPQExpBuffer(query, "START_REPLICATION SLOT \"slot\" LOGICAL 0/0 ");
res = PQexec(conn, query->data);
if (PQresultStatus(res) != PGRES_COPY_BOTH)
{
    PQclear(res);
    proc_exit(1);
}
PQclear(res);
[...continue...]
```
With replication protocol (2) – Fetch changes

- **PQgetCopyData as central piece**
  - Status 0 = no data. Wait for more and continue process
  - Status -1 = End of stream. -2 = Failure when reading stream

```
PQgetCopyData(conn, &copybuf, 1);
```

**Keepalive message**
- 1 byte for ‘k’
- 8 bytes for WAL end position
- 8 bytes for send timestamp

**Record message:**
- 1 byte for ‘w’
- 8 bytes for WAL start
- 8 bytes for WAL end
- 8 bytes for send time
- Rest is data generated
With replication protocol (3)

- **Look at pg_recvlogical in core!**
  - Create, drop slots, fetch changes as-is
  - [http://www.postgresql.org/docs/devel/static/app-pgrecvlogical.html](http://www.postgresql.org/docs/devel/static/app-pgrecvlogical.html)
  - SendFeedback() is really, really important to avoid WAL file bloat

- **Demonstration with receiver_raw**
  - Fetch raw queries from decoder_raw
  - Apply them on local database
  - Need some pre-process:
    - Dump of remote schema
    - Correct REPLICA IDENTITY targets depending on application relations
  - Code
    - receiver_raw in this repo => [https://github.com/michaelpq/pg_plugins](https://github.com/michaelpq/pg_plugins)
    - PostgreSQL license
And then…

### Cool use cases
- Online upgrade (doable with 9.4 but tightly linked with application)
- Auditing
- Replication solutions: synchronous replication out-of-the-box!

### Limitations
- Need advanced hacking skills
- Consistent dumps of replication slots by pg_dump
- No DDL yet, but event triggers perhaps showing up
- In-core online upgrade solution not there yet
  - Drastic reduction of downtime
  - Need some pg_upgrade --online
Thanks!
Questions?