Writing a Logical Decoding Plug-In.

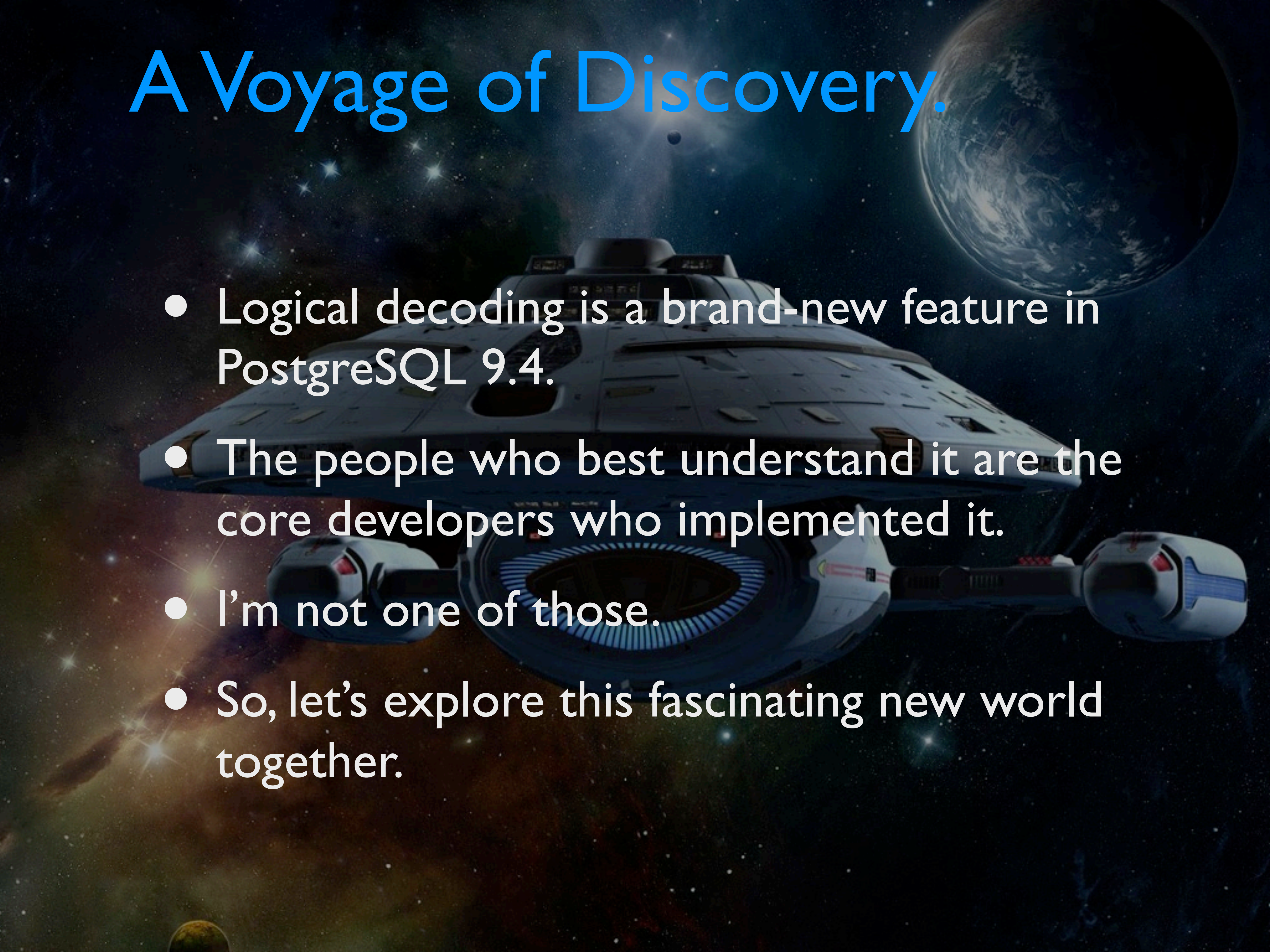
Christophe Pettus
PostgreSQL Experts, Inc.

FOSDEM 2015

Hello!

- We're going to talk about logical decoding in PostgreSQL.
- Christophe Pettus, pleased to meet you.
- PostgreSQL person since 1997.
- Consultant with PostgreSQL Experts, Inc., in sunny San Francisco (more rain, please).

A Voyage of Discovery.



- Logical decoding is a brand-new feature in PostgreSQL 9.4.
- The people who best understand it are the core developers who implemented it.
- I'm not one of those.
- So, let's explore this fascinating new world together.

The Problem.

- Something changes on one database server.
- We want that change to appear on another database server.
- Seems pretty straight-forward, yes?

Why do we want this?

- A server to fail over to if the first one dies.
- Pushing transactional information to a data analysis system.
- Distributing centrally-generated information to peripheral systems.
- Multi-master scaling, one could dream.

So, how can we do this?

- Our options circa 2014 were:
  - WAL shipping.
  - Streaming replication.
  - Trigger-based replication.

WAL shipping.

- The only in-core solution before 9.0.
- Secondary database servers read WAL files generated by a primary.
- Applying those WAL files, it stays in sync with the primary.
- Great! Problem solved!

Uh, no, not really.

- Secondary can do nothing (not even queries) except read WAL segments.
- Each secondary can only read from a single primary.
- No selectivity: The entire database cluster is replicated.
- Pretty much only good for failover.

PGX
POSTGRESQL
EXPERTS, INC.

Other WAL shipping issues.

- Only as good as the last WAL file sent over.
- WAL file management is a pain in the neck.
  - … especially for multiple secondaries.
- No synchronous replication.
  - You can lose committed transactions.

Streaming Replication to the rescue!

- Secondary connects directly to the primary.
- WAL information is streamed over as it is generated.
- Secondary (can) stay very close to the primary.
- Synchronous replication possible if you don’t mind the throughput penalty.

Problem solved!

- Uh, no, sorry.
- Secondaries can take reads, but not writes.
- It’s still all-or-nothing.
- Long disconnections can require that they be re-initialized.

Fine. How about slony?

- … or Bucardo, or Londiste, or…
- Installs triggers on tables to track changes.
- Triggers fire on data changes, add deltas to queues.
- Daemons drain the queues, distribute the changes to secondary machines.

Sounds promising!

- Changes operate on a logical (INSERT, DELETE, UPDATE) level, not at the WAL level.
- Can replicate a subset of the cluster: just some database, just some tables.
- No (theoretical) limit to replication topology.

Problem solved!

- Well, sorta.
- Triggers are not free.
- One more moving part.
- Schema changes don't (currently) fire triggers, so have to be applied "by hand."
- Not in core.

Aaaand…

- … notoriously fiddly to set up and keep running.
- … each have their own quirks and limitations.
- … not general-purpose frameworks for other possible tasks, like auditing.

What would be great would be…

- … if we could get a stream like the streaming replication stream…
- … but on the logical level, rather than WAL pages.
- … and then we could do whatever we want with it.

Behold: Logical Decoding.

- A framework in PostgreSQL, not a specific tool.
- Decodes the WAL stream back into INSERT / UPDATE / DELETE-level statements.
  - Not the exact statements, but ones corresponding to the changes done.

New feature, new concepts.

- Logical decoding introduces some new concepts.
  - Slots.
  - Output plug-ins.

The World Before Slots.

- Pre-9.4, replication was driven by the secondary.
- The secondary connected to the primary.
- The secondary told the primary where it needed the stream to start.
- The primary started streaming, or told the secondary that it was out of luck.

Enter Slots.

- Brand new 9.4 feature.
- A named structure in the primary server.
- Optional for WAL-based (physical) streaming replication.
- Required for logical streaming replication.
- Can be created either in advance, or by the secondary on connection.

Physical Replication Slots.

- In essence, a persistent record of WAL position.
- Once activated, prevents WAL removal on the primary if the secondary hasn't received it.
- More accurate WAL cleanup.
- A whole new way to run out of disk space.

Logical Replication Slots.

- A "pipe" that receives a continuous stream of logical changes.
- The "end" of the pipe is an output plug-in.
- The output plug-in takes the logical stream, and does whatever it wants to it.
- The output of the plug-in (not the stream itself!) is sent to the client.

Output plug-ins…

- … are bits of C code that respond to function calls.
    - The logical replication stream is that series of function calls.
- Loaded into PostgreSQL as shared libraries.
- Not inherently complex! Mostly just a lot of C-level push-ups to deal with.

When are changes decoded (part 1)?

- The output plug-in is only called when there is a consumer for the changes.
- Either a consumer is connected via to a replication slot, or one of the `pg_logical_slot_get_changes`() family is called.

When are changes decoded (part 2)?

- Decodes only happen when a transaction has been flushed to disk.
  - even if synchronous_commit = off
- Always in transaction commit order.
- Each transaction is decoded before moving on to another one.
  - No “interleaved” transactions.

What can an output plug-in write?

- Pretty much anything it wants.
- By default, it is assumed to write a `bytea` stream.
- If it writes text in the current server encoding, it can declare that.
- It's up to the consumer to deal with whatever the output plug-in generates.

Creating a slot.

```
xof=# select
pg_create_logical_replication_slot('test_slot',
'test_decoding');
pg_create_logical_replication_slot
------------------------------------
 (test_slot,0/32009880)
(1 row)
```

Once a slot is created…

- … no WAL records are cleaned up until they are no longer required.
- This means that if you create a slot but no client ever connects…
- … no WAL records are *ever* cleaned up.

LET ME SAY THAT AGAIN.

- If you create a replication slot but no consumer connects…
- WAL segments will be kept **FOREVER**.
- And you **WILL RUN OF OUT DISK SPACE**.
- So **DON'T DO THAT**.

Flow of Execution.

- Consumer calls slot asking for output.
- PostgreSQL determines last WAL position for that slot.
- Decodes the WAL and calls the output plug-in repeatedly, collecting output from it.
- Transmits that output to the consumer.
- Lather, rinse, repeat.

What data is sent?

- Only completed transactions that have been flushed to disk are sent to the output plug-in.
- No partial transactions.
- No rolled-back transactions.
- No transactions that haven't yet been flushed.

Savepoints?

- Only the final transaction state is streamed, so…
- All committed/rolled-back savepoints are “smoothed out” in the data stream.

Example: We have this table.

```
xof=# \d t
                         Table "public.t"
 Column |  Type   |                    Modifiers
--------+---------
+----------------------------------------
 pk     | integer | not null default
nextval('t_pk_seq'::regclass)
 z      | text    |
Indexes:
    "t_pkey" PRIMARY KEY, btree (pk)
```

So, we do an INSERT.

```
xof=# INSERT INTO t(z) VALUES('foo');
INSERT 0 1
```

And we look at the output.

```
xof=# SELECT * FROM
pg_logical_slot_get_changes('test_slot', NULL, NULL,
'include-xids', '0');
  location  | xid  |                       data
------------+------
+-----------------------------------------
 0/320499F0 | 4983 | BEGIN
 0/320499F0 | 4983 | table public.t: INSERT: pk[integer]:
1 z[text]:'foo'
 0/32049B38 | 4983 | COMMIT
(3 rows)
```

What you have to write.

- `_PG_output_plugin_init`
- `pg_decode_startup`
- `pg_decode_shutdown`
- `pg_decode_begin_txn`
- `pg_decode_commit_txn`
- `pg_decode_change`

test_decoding

- Sample logical decoding plugin in contrib/.
- Gives a lot of useful boilerplate on how to write a plugin.
- Follow along if you want!
- Use it as a template; don’t bother starting with an empty .c file.

_PG_output_plugin_init

- This function must have this particular name.
- Used to supply the addresses of the other callback functions to the framework.
- The other functions can have whatever names you want.
- You have to specify all of them.

pg_decode_startup

- Called when the plugin is "started."
- A plugin is started when a slot is created or a consumer connects.
- The same plugin is used multiple times for multiple slots.
- You'll get called for each consumer connection.

pg_decode_startup parameters.

- LogicalDecodingContext: Includes a place for your stuff. Never store state anywhere else!
- OutputPluginOptions: The options specified with this particular stream.
- is_init: True on slot creation; false when a new consumer connects to the slot.

pg_decode_startup timing.

- Called each time a consumer connects.
- Each `pg_logical_slot_get_changes` counts as a “connection.”
- Options are specified on the get_changes calls, not at slot creation time.
    - So, each call could have different options.

pg_decode_shutdown

- Called when the framework is done streaming changes to the plugin.
- Either at the end of a get_changes call, or when the consumer disconnects.
- Release everything you've allocated; no telling when you might be called again.

pg_decode_begin_txn

- Called when a transaction begins.
- Called even for single-statement transactions.
- Note that empty transactions are both possible and (at the moment) quite common.

pg_decode_commit_txn

- Called on commit.
- Note that the plug-in is never called for rolled-back transactions.

pg_decode_change

- The fun part!
- Called once per tuple, per operation.
- Currently: INSERT, UPDATE, DELETE.
- Corresponds to the logical change, not to the actual SQL statement executed.

pg_decode_change parameters

- LogicalDecodingContext: A way to get your private data.
- ReorderBufferTXN: Info about the open transaction.
- Relation: The relation the tuple belongs to.
- ReorderBufferChange: The change itself.

ReorderBufferChange* change

- change->action: specifies if it is an INSERT, UPDATE, DELETE.
- change->data.tp.newtuple has the new tuple data for INSERT and UPDATE.
- change->data.tp.oldtuple has the old tuple data for DELETE.

Caveats…

- … always be prepared for data.tp.newtuple and data.tp.oldtuple to be NULL.
- newtuple is the whole tuple, regardless of what has changed, except unchanged TOASTed data.

What do we get on an UPDATE?

```
xof=# SELECT * FROM
pg_logical_slot_get_changes('test_slot', NULL, NULL,
'include-xids', '1');
  location   | xid  |
data
------------+------
+---------------------------------------------------------
-------------------------------
 0/3204A090 | 4986 | BEGIN 4986
 0/3204A090 | 4986 | table public.t: UPDATE: old-key:
pk[integer]:1 new-tuple: pk[integer]:7 z[text]:'bar'
 0/3204A1E0 | 4986 | COMMIT 4986
(3 rows)
```

REPLICA IDENTITY

- New ALTER TABLE option in 9.4.
- Controls what data is presented to the plug-in on an UPDATE or DELETE.
- DEFAULT is primary key values, if they changed.
- FULL, NOTHING, USING INDEX.

tuples.

- You are getting pointers to standard PostgreSQL tuple structures.
- Can only be decoded using the Relation's TupleDesc structure.
- See tuple_to_stringinfo in test_decoding.c for an example of how to iterate through the tuple structure.

Writing.

- Once you have something to say, how do you say it?
- Two output functions:
  - `OutputPluginPrepareWrite`
  - `OutputPluginWrite`

OutputPluginPrepareWrite

- Called before doing any output in any callback function.
- Parameters:
  - ctx: The context.
  - last_write: true if the subsequent write is the last one in this callback invocation.

Writing.

- ctx->out is a StringInfo; just append to that.
- You can use the standard PostgreSQL StringInfo functions.
- You can append to it multiple times after calling OutputPluginPrepareWrite.
- When done…

OutputPluginWrite

- Called to indicate that output can be sent to the consumer.
- Two parameters:
  - ctx: Our friend, the context.
  - last_write: If true, done with writing this callback cycle. Must match the value you passed in OutputPluginPrepareWrite.

Output structuring.

- Output is transmitted to the consumer as OutputPluginWrite is called.
- It is tagged with the WAL position and xid it relates to.
- The decoded output is passed along as an opaque byte string, and the consumer is responsible for understanding it.

Restrictions.

- A plug-in cannot create an xid.
- Cannot modify any table.
- Can only read system catalogs (created with init_db) or (new feature!) user catalog tables.
  - user_catalog_table = true

pg_recvlogical

- Utility to connect to and receive the streaming output of a logical replication slot.
- Streams the output to a file or stdout.
- Doesn’t process it; just stores it.
- Very handy for debugging; just tail the output!

Now, the bad news.

- Brand new feature: Expect some lumps and bumps.
- Schema changes are not passed to logical decoding plugins (as of 9.4).
- Plugins link directly into PostgreSQL, and can bring down the whole server.
- Slots can cause disk space exhaustion.

What can we do?

- Build slony-like replication engines that don't require triggers.
  - Partial replication, filtered changes, multi-master replication…
- Audit trails that don't require local tables (which can be compromised).
- Anything else you can think of!

Now, go crazy.

Thank you!

Questions?

- thebuild.com — personal blog.
- pgexperts.com — company website.
- Twitter @Xof
- christophe.pettus@pgexperts.com

PGX
POSTGRESQL
EXPERTS, INC.