A New Way to avoid trigger based logging in PostgreSQL

(.. a BillPay use case)

Olakunle Olaniyi
Database Administrator
BillPay GmbH (A Klarna Group Company)
Email: olakunle.olaniyi@klarna.com

Database Logging...

- As far as best practices go, we log events on selected tables caused by inserts/deletes/updates using trigger(s).
- Destination table might or might not have identical structure as the originating table.
- The who,when and what of events are usually of interests.
- Sometimes requirements prompt the use of fairly complicated logic in the trigger functions.

Some considerations …

- Triggers are easy,convenient to setup and maintain and they work incredibly well.
- Triggers actions ultimately determine the amount of overhead.
- There are limitations on the kind of operations they could be used for.

Some more considerations

- Log/audit tables becoming too huge to be stored in the same database as production data.
- We wanted to do more with changes on tables rather than logging and/or auditing.
- Exploiting the same change streams from tables to be used by multiple downstream applications.

Logging using the streaming replication protocol (Logical Decoding)

Logical Decoding; What is it ?

- Extracts all persistent changes to a database's table(s).
- Changes are in commit order
- Provides outputs in an easy to understand format.
- No knowledge of the database's internal state required.
- Extract contents of WAL files into application specific form.
- Available on PostgreSQL v9.4+

What you need

- postgresql.conf
    - wal_level must be at least logical
    - max_replication_slots > 0
    - Max_wal_senders > 0
- pg_hba.conf
    - Should allow replication connection
- An output plugin written in C
- A suitable client to consume changes

Components

- Replication Slot
    - Provides the mechanism to store change streams that can be replayed to a client in the order they were made on the origin server.
- Output Plugins
    - Output plugins transform the data from the WAL’s internal representation into the format the consumer of a replication slot desires
- Consumer clients
    - Any client capable of capturing the transformed changes from replication slot using an output plugin with either
        - SQL decoding interface
        - Streaming replication interface

Replication slots

Physical Replication Slots :

- Used for streaming replication by replicas
- Requires no output plugin for consumption
- All or nothing consumption

Logical Replication Slots:

- Needed for logical decoding
- Output plugin is a must
- Table level consumption
- DDLs are not decoded

Output plugins

- Must be written in C to interface with the postgres backend to utilize various callbacks
- Must be installed on the database server for which replication slots are to be created

What we found out about output plugins…

test_decoding

- Part of the postgres core
- Text output format
- Not particularly useful but may serve as a basis for developing other plugins

Sample output :

```
BEGIN txid

table public.data: INSERT: id[integer]:1 data[text]:'1'

table public.data: INSERT: id[integer]:2 data[text]:'2'

COMMIT txid
```

pglogical_output

- Developed and open sourced by 2ndQuadrant
- A very fast plugin
- Json output format
- Merged into pglogical decoding project - no longer available as a standalone plugin

Sample output :

```
{"action": "B", "has_catalog_changes": "f"}

 {"action": "I", "newtuple": {"ba": null, "js": null, "ts": null, "tx": "textval", "jsb": null, "seq": 1},
"relation": ["public", "demo"]}

 {"action": "C"}
```

wal2json

- Currently still alive - releases, development and bug fixes being applied
- Json output format
- Provides an acceptable decoding speed.
- Very enticing features e.g:
    - including and excluding schemas and tables
    - transaction timestamp, schema-qualified, LSN, data types, and transaction ids
- Supports a wide variety of platforms including Windows

… wal2json

Sample output :

```
{"change": [

{"kind": "delete","schema": "public","table": "xpto","oldkeys": {"keynames": ["id"],"keytypes": [
"int4"],"keyvalues": [1]}}]

}
```

How much data is good enough ?

Your use case largely determines the replica identity you choose

- DEFAULT
    - Only new records + old value(s) of changing pkey columns
    - This is what you want; mostly
- USING INDEX
    - Unique Indexes
    - Same as Default
- FULL
    - No key needed
    - New + old records always written to WAL
- NOTHING

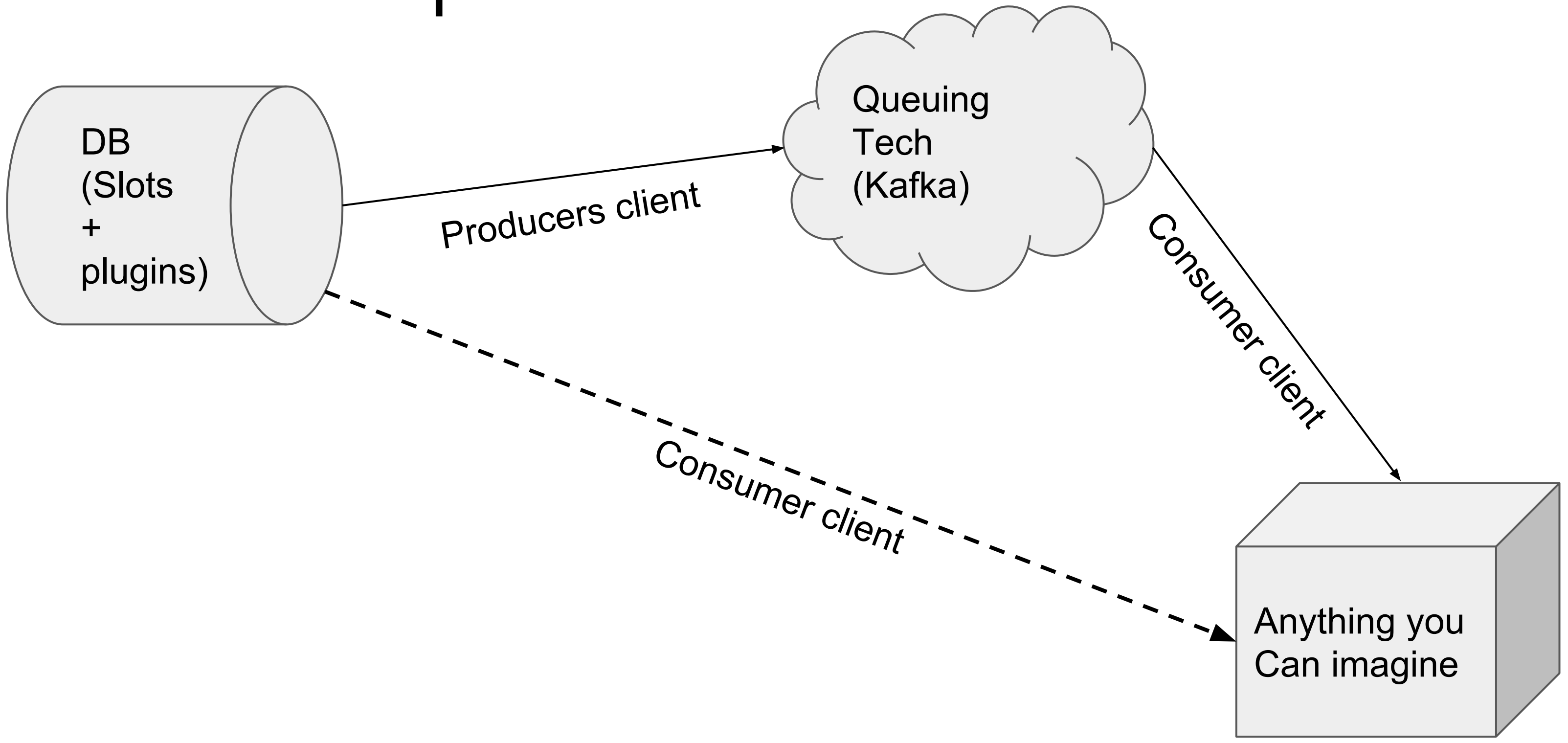
Clients

- Our client of choice -> Python
  - Psycopg2
    - connection_factory=**psycopg2.extras.LogicalReplicationConnection**
    - For bytea data types, set **decode=False** for the start_replication call

Others :

- Java
  - Postgresql java jdbc driver supports logical replication
- C
  - libpq is all you need

An oversimplified Architecture



Logging Tables

```
SELECT pg_create_logical_replication_slot('test_slot','wal2json') ;

BEGIN ;
CREATE TABLE test_table
(
    t_id bigserial primary key,
    t_country text
);

INSERT INTO test_table(t_id,t_country)  VALUES(3,'Italy') ;

UPDATE  test_table SET t_country = 'Netherlands' where  t_id  = 3 ;

COMMIT;
```

How do we queue in reality ?

Sample output from kafka topic :

```
{"action": "B", "xid": 50089281, "commit_time": "2018-06-22 14:25:02.68791+01", "startlsn": "6F/BE3CE6F8"}

{}

{"action": "I", "newtuple": {"t_id": 3, "t_country":'Italy'}, "relation": ["public", "test_table"]}

{"action": "U", "newtuple": {"t_id": 3, "t_country":'Netherlands'}"relation": ["public", "test_table"]}

{"endlsn": 479933032184, "action": "C", "xid": 50089281, "commit_time": "2018-06-22 14:25:02.68791+01", "startlsn": "6F/BE3CE6F8"}
```

Output enrichment

- For logging and/or auditing tables default replica identity is barely enough.
- To capture old and new records full replca identity must be enabled for interested tables
    - ALTER TABLE test_table REPLICA IDENTITY FULL
    - UPDATE test_table SET t_country = ‘Holland’ where t_id = 3 ;

More output

```
{"action": "B", "xid": 50089281, "commit_time": "2018-06-22 14:35:02.68791+01", "startlsn": "6F/BE3CE6F9"}

{"action": "U", "newtuple": {"t_id": 3, "t_country":‘Netherlands’},"oldtuple": {"t_id": 3, "t_country":’Holland’}, "relation": ["public", "test_table"]}

{"endlsn": 479933032184, "action": "C", "xid": 50089281, "commit_time": "2018-06-22 14:35:02.68791+01", "startlsn": "6F/BE3CE6F9"}
```

… in DB2

```
CREATE TABLE test_table_log
(
    tl_id bigserial,
    tl_t_id bigint,
    tl_t_country_new text,
    tl_t_country_old text,
    tl_action char(1)
);
```

Consume changes with our python client;

```
row = to_json(stream_payload)
new_row = row.get(‘newtuple’)
old_row = row.get(‘oldtuple’)
tl_t_id = new_row[‘t_id’]
tl_t_country_new = new_row[‘t_country’]
tl_t_country_old = old_row[‘t_country’]
tl_action = row[‘action’]
```

Incrementally Loading DWH

- A default Replica Identity on all tables of interest is adequate
- Create a replication slot
- Export a snapshot
    - Select pg_export_snapshopt()
- Use pg_dump or some other method to get the data up to the current snapshot
- Consume changes from the slot to the DWH stage tables
    - PostgreSQL upsert comes in handy here
    - To prevent unwanted increase in WAL size, changes are queued in kafka
    - Aside preventing increase in WAL size, queuing is of interest since the same changes is used by different downstream applications

Aggregation

- Default Replica Identity may or may not be enough depending on aggregation the use case
- Data is exported just like for a DWH system
- Sometimes a streaming database for aggregation
    - Depending on the use case, a custom consumer which does the aggregation might be desirable
    - Aggregation are done on different levels

Database major upgrades?

- Default Replica Identity works fine in this case
- Create a replication slot
- Perform an initial export of the database
  - A replica database could be used in this case
- Consume the changes from the replication slot
- Perform a one-to-one mapping of the source database to destination database

Problems we faced … so far

- Slow output plugin leads to significant lag
- Huge transactions slow down slot consumption, increase memory requirement and might render downstream application unusable in the meantime.
- Duplicate entries if progress is not properly handled
- A new and exciting way to send the production database into panic once the WAL directory is full

Monitoring

- Use pg_replication_slots
- pg_stat_replication
- Replication origin functions

Useful tweaks that helped

- Set the absolute minimum replica identity on a need to need basis
- For huge databases, maintain a minimum number replication slots as needed
- For huge DDLs/Transactions (e.g adding a column with default value) and a table with replica identity FULL:
  - Assuming you are brave enough to run a ddl with default value in a single command in production:
    - Set replica identity to Nothing
    - Run DDL
    - Reset replica identity back to old Value
- Never use the SQL interface in production , except for testing purposes ; it’s a recipe for disaster

Useful links

- wal2json : https://github.com/eulerto/wal2json
- Pglogical_ouput: https://github.com/postgrespro/postgres_cluster/tree/master/contrib/pglogical_output
- Our python client ? … open source soonish ….

Klarna.

???