Caveats

- A synchronous replica receiving changes via logical decoding will work in the scope of a single database. Since, in contrast to that, synchronous_standby_names currently is server wide, this means this technique will not work properly if more than one database is actively used.
- In synchronous replication setup, a deadlock can happen because logical decoding of transactions can lock catalog tables to access them. Exclusive lock on both system and user catalog tables under the following conditions:
    - Issuing an explicit LOCK on pg_class in a transaction.
    - Perform CLUSTER on pg_class in a transaction.
    - PREPARE TRANSACTION after LOCK command on pg_class and allow logical decoding of two-phase transactions.
    - PREPARE TRANSACTION after CLUSTER command on pg_trigger and allow logical decoding of two-phase transactions. This will lead to deadlock only when the published table has a trigger.
    - Executing TRUNCATE on [user] catalog table in a transaction.

REFERENCE:
https://www.postgresql.org/docs/current/logicaldecoding-synchronous.html
https://www.postgresql.org/docs/current/logicaldecoding.html

Logical Replication Caveat

Note that the described limitations are subject to change with each new major version of PostgreSQL.

Remember to review the release notes for feature updates:

- DDLs not supported
- No Replication Queue Flush (Failover is problematic)
- No Cascaded Replication
- One unique index/constraint/pk per table
- Permissions (remote access by subscriber)
- Primary key must exist
- Sequences
- Triggers
- Truncate command is not propagated
- Unlogged/temporary tables not supported

Conflict Resolution

Conflicts occur when incoming data violates constraints on the subscriber and includes:

- Permissions failures on target tables
- Row-level security enabled on target tables
- Direct writes on the subscriber
- Complex topology between publications and subscriptions
- Other writes to the same set of tables by an application or other subscribers

Examples:

- Adding a constraint, such as a UNIQUE index, on the subscriber that doesn't exist on the publication.
- Inserting a record twice which uses a PRIMARY KEY.
- Recasting a column datatype on the published table and inserting a record inconsistent with the subscribed table's column ex: bigint values cannot be inserted into smallint columns.
- Adding a column on a published table that doesn't exist on the subscriber. *Note adding a new column on the subscriber that doesn't yet exist on the publication is okay.*

In the case of stalled logical replication, there are two possible remedies:

1. Perform the requisite DDL, or DML operation, on the subscriber eliminating the contradiction between publisher and subscriber.
2. Instruct the subscriber to skip over the problem record.

POC

The following example consists of a simple 2 node logically replicating cluster. A conflict is created when an INSERT fails due to a duplicate key violation.

Table "t1" consists of four columns but while pg2.t1 has a unique constraint on column 'a' there is none on pg1.t1.

```
# pg1
                                 Table "public.t1"
  Column  |           Type           | Collation | Nullable |      Default
----------+--------------------------+-----------+----------+--------------------
 id       | uuid                     |           | not null | gen_random_uuid()
 a        | uuid                     |           |          | gen_random_uuid()
 comments | text                     |           |          | 'pg2-openai'::text
 t_stamp  | timestamp with time zone |           | not null | clock_timestamp()
Indexes:
    "t1_pkey" PRIMARY KEY, btree (t_stamp, id)
```

```
# pg2
                              Table "public.t1"
  Column  |           Type           | Collation | Nullable |      Default
----------+--------------------------+-----------+----------+--------------------
 id       | uuid                     |           | not null | gen_random_uuid()
 a        | uuid                     |           |          | gen_random_uuid()
 comments | text                     |           |          | 'pg2-openai'::text
 t_stamp  | timestamp with time zone |           | not null | clock_timestamp()
Indexes:
    "t1_pkey" PRIMARY KEY, btree (t_stamp, id)
    "t1_a_key" UNIQUE CONSTRAINT, btree (a)
```

For the purposes of this POC, both nodes pg1 and pg2 already have these two records in their respective tables:

```
                  id                  |                  a                   |  comments  |   t_stamp
--------------------------------------+--------------------------------------+------------+------------
 c80717d5-0203-4546-a8fe-7749af939e17 | fcfe526d-5237-427c-a317-40d989b66dc4 | pg1-openai | 15:08:53.291944+00
 970f18f7-c95c-4cab-aad6-0ec2faf4a964 | e0bd817d-df04-4f82-a1fa-11aa3f77b6e6 | pg1-openai | 15:08:53.292075+00
```

A conflict is created by inserting these two records on host "pg1". Both records exist on "pg1.t1" but not on "pg2.t1":

```
# pg1
insert into t1(a,comments) values ('fcfe526d-5237-427c-a317-40d989b66dc4','broken');
insert into t1(a,comments) values (default, default, 'it works', default);
```

```
# pg1
db01=# select * from t1 order by a;
                  id                  |                  a                   |  comments  |          t_stamp
--------------------------------------+--------------------------------------+------------+----------------------
 d2923618-14e5-4301-b910-f09ce60dc9ae | dfc21e99-b7e4-4ff1-a2b7-e1300d828b23 | pg1-openai | 15:14:13.027779+00
 970f18f7-c95c-4cab-aad6-0ec2faf4a964 | e0bd817d-df04-4f82-a1fa-11aa3f77b6e6 | pg1-openai | 15:08:53.292075+00
 c80717d5-0203-4546-a8fe-7749af939e17 | fcfe526d-5237-427c-a317-40d989b66dc4 | pg1-openai | 15:08:53.291944+00
 667ead60-6060-4793-99aa-c990020a594f | fcfe526d-5237-427c-a317-40d989b66dc4 | broken     | 15:13:38.784325+00
 db5b9102-3ae8-4e0e-aef1-94bec8142173 | 820bed1c-eda8-4938-b2c4-d2c9493a27f3 | it works   | 15:34:04.847141+00
```

```
# pg2
db01=# select * from t1 order by a;
                  id                  |                  a                   |  comments  |          t_stamp
--------------------------------------+--------------------------------------+------------+--------------------------
 970f18f7-c95c-4cab-aad6-0ec2faf4a964 | e0bd817d-df04-4f82-a1fa-11aa3f77b6e6 | pg1-openai | 15:08:53.292075+00
 c80717d5-0203-4546-a8fe-7749af939e17 | fcfe526d-5237-427c-a317-40d989b66dc4 | pg1-openai | 15:08:53.291944+00
```

```
# pg2: log
2024-07-18 15:16:43.963 UTC [284] LOG:  background worker "logical replication worker" (PID 466) exited with exit code 1
2024-07-18 15:16:48.956 UTC [468] LOG:  logical replication apply worker for subscription "pg1" has started
2024-07-18 15:16:48.967 UTC [468] ERROR:  duplicate key value violates unique constraint "t1_a_key"
2024-07-18 15:16:48.967 UTC [468] DETAIL:  Key (a)=(fcfe526d-5237-427c-a317-40d989b66dc4) already exists.
2024-07-18 15:16:48.967 UTC [468] CONTEXT:  processing remote data for replication origin "pg_16481" during message type
"INSERT" for replication target relation "public.t1" in transaction 927, finished at 0/1E656E0
2024-07-18 15:16:48.968 UTC [284] LOG:  background worker "logical replication worker" (PID 468) exited with exit code 1
```

Resolving The Conflict

Resolving the conflict is a two step process:

1. Host pg2: identify the LSN of concern which is found in the host's log file
2. Host pg1: advance the problem slot's LSN and skip past the troublesome record

```
# pg1: resolve the conflict by incrementing the LSN obtained from pg2 logs
#       tip: you can use a hex calculator ...
#               1E656E0+1 = 1E656E1
#
db01=# select * from pg_replication_slot_advance('pg2','0/1E656E1');
 slot_name |  end_lsn
-----------+-----------
 pg2       | 0/1E656E1
```

Validation

```
# pg2: duplicate record is skipped
db01=# select * from t1 order by a;
                  id                  |                  a                   |  comments  |           t_stamp
--------------------------------------+--------------------------------------+------------+---------------------------
 db5b9102-3ae8-4e0e-aef1-94bec8142173 | 820bed1c-eda8-4938-b2c4-d2c9493a27f3 | it works   | 15:34:04.847141+00
 d2923618-14e5-4301-b910-f09ce60dc9ae | dfc21e99-b7e4-4ff1-a2b7-e1300d828b23 | pg1-openai | 15:14:13.027779+00
 970f18f7-c95c-4cab-aad6-0ec2faf4a964 | e0bd817d-df04-4f82-a1fa-11aa3f77b6e6 | pg1-openai | 15:08:53.292075+00
 c80717d5-0203-4546-a8fe-7749af939e17 | fcfe526d-5237-427c-a317-40d989b66dc4 | pg1-openai | 15:08:53.291944+00
```

Replication Latency

Monitoring

These are among the more relevant views that can be used to monitor the state of replication.

```
-- standard views
select * from pg_stat_replication;
select * from pg_replication_slots;
select * from pg_get_replication_slots();
```

Although not directly related to measuring latency between PUBLICATIONs and SUBSCRIPTIONs, these examples demonstrate streaming latency.

Example 1:

```
select
    case
        when pg_last_wal_receive_lsn() = pg_last_wal_replay_lsn()
        then 0
    else extract (EPOCH FROM now() - pg_last_xact_replay_timestamp())
    end as log_delay;
```

Example 2:

```
select
    slot_name,
    confirmed_flush_lsn,
    pg_current_wal_lsn(),
    pg_walfile_name(pg_current_wal_lsn()),
    (pg_current_wal_lsn() - confirmed_flush_lsn) as lsn_distance
from pg_replication_slots;
```