finmason


# PostgreSQL Logical replication
# Practical use case – Lessons Learned

Aleš Zelený

Prague PostgreSQL Developers Day 2019

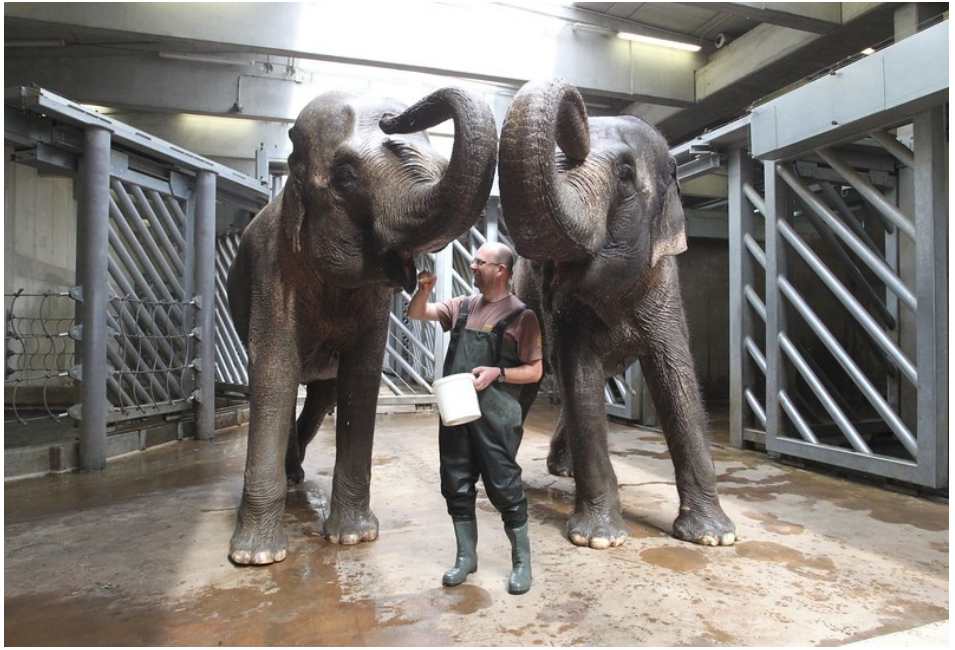
finmason


Who we are
Investment Analytics as a Service Platform
FinMason
Advisors
Compliance
FinTech
Investors
MOBILE TECHNOLOGY
Wealth Management .com
2018 Industry Awards
Winner
•COMPLIANCE•
2018 ADVISORY SOFTWARE SURVEY
VOTED #1 IN
•SATISFACTION•

# Who’s me?

finmason

- InterBase / Firebird app developer, DBA (3 years)
- Oracle DBA (17 years)
- PostgreSQL DBA (8 years)
- Elephants enthusiast ...



# Agenda

finmason

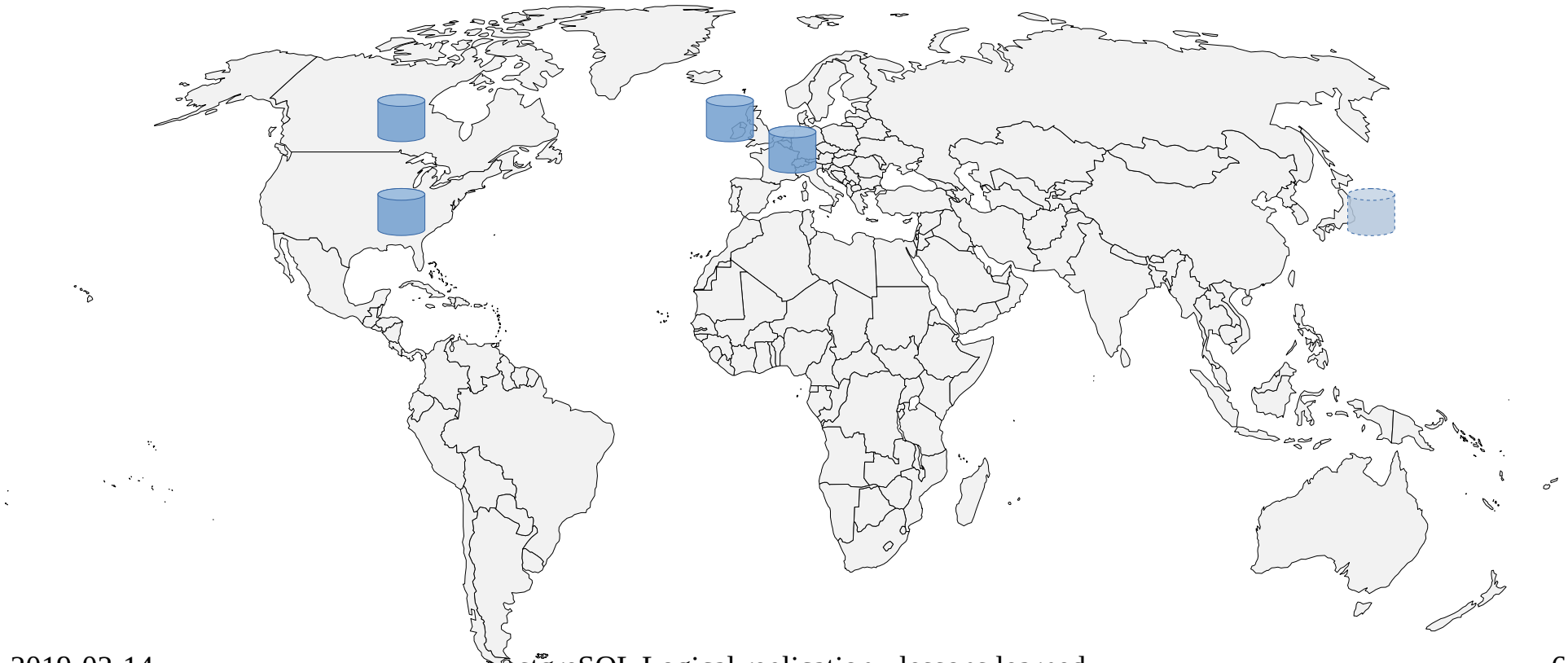
- Distributed solution description
- The boring part
- The adrenaline part (Lessons learned)
- Issues experienced and mitigation
- Conclusion

# Distributed

finmason

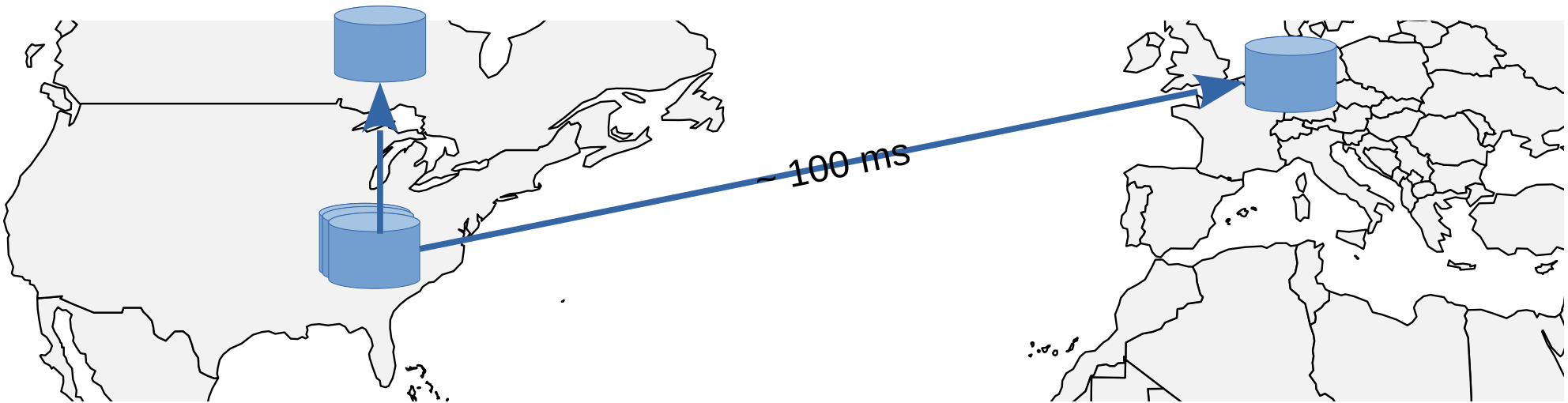
# ...around globe

finmason



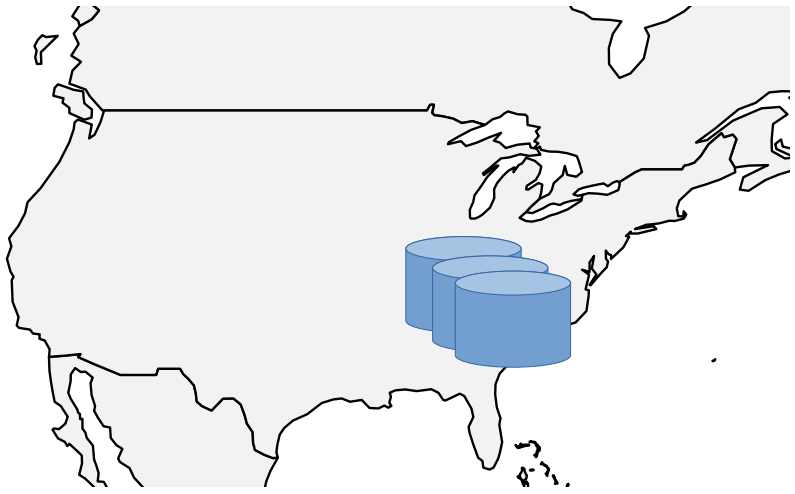
# Asynchronous

finmason



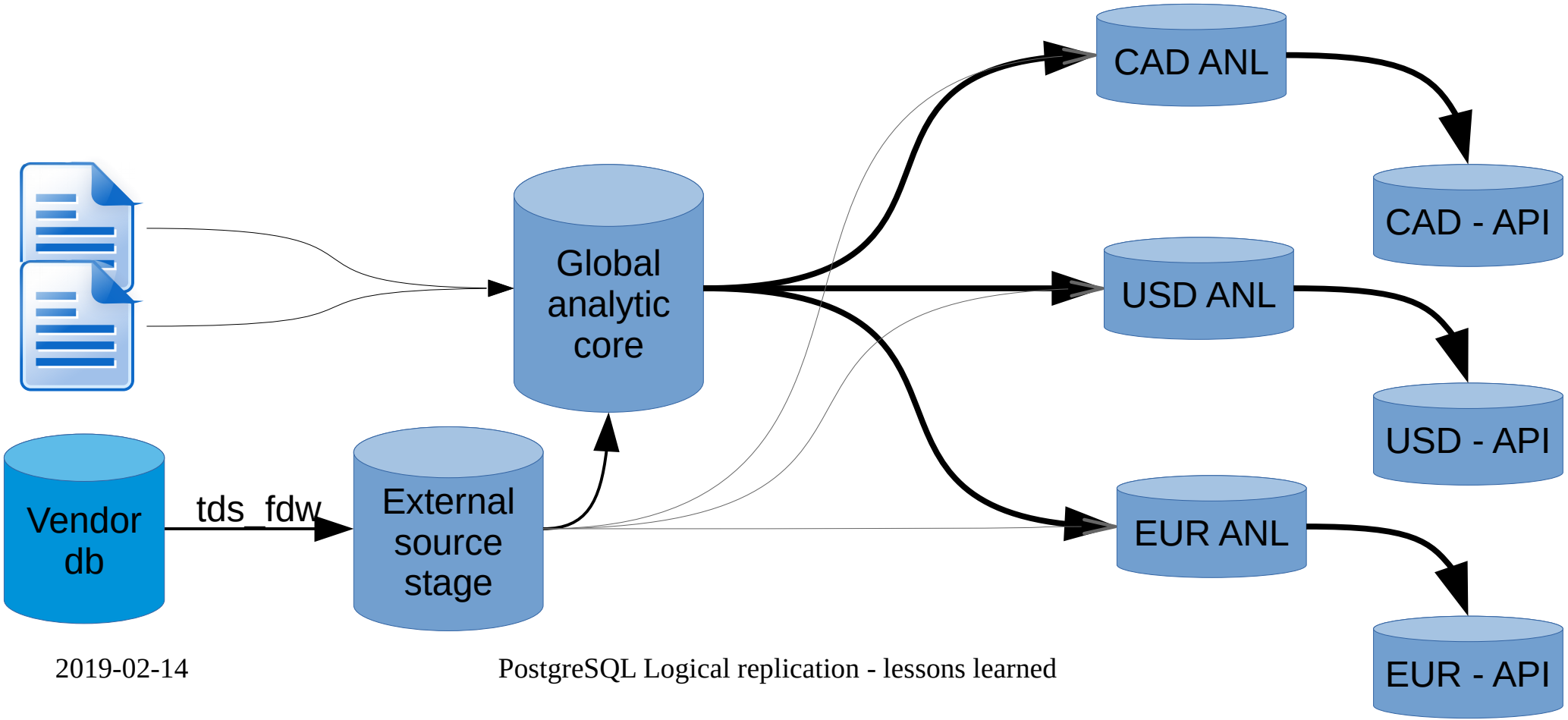
# Replicated also within data center

- Analytic core - publisher
  - API servers - subscribers
- Asynchronous also within data center
  - Batch ETLs or Logical replication
- Logical replication
  - Global data managed at one place
  - Workload distribution
  - Pre–processed data are Geo-distributed

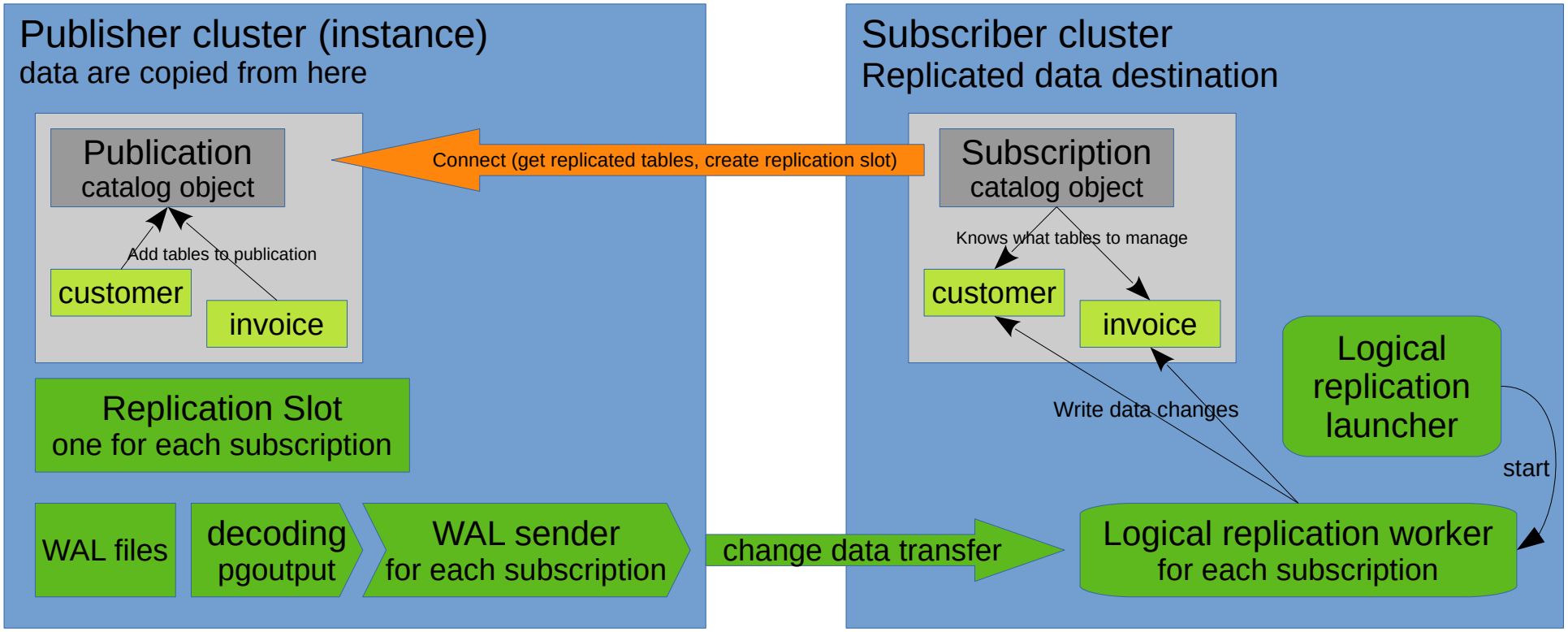


# Table level replication

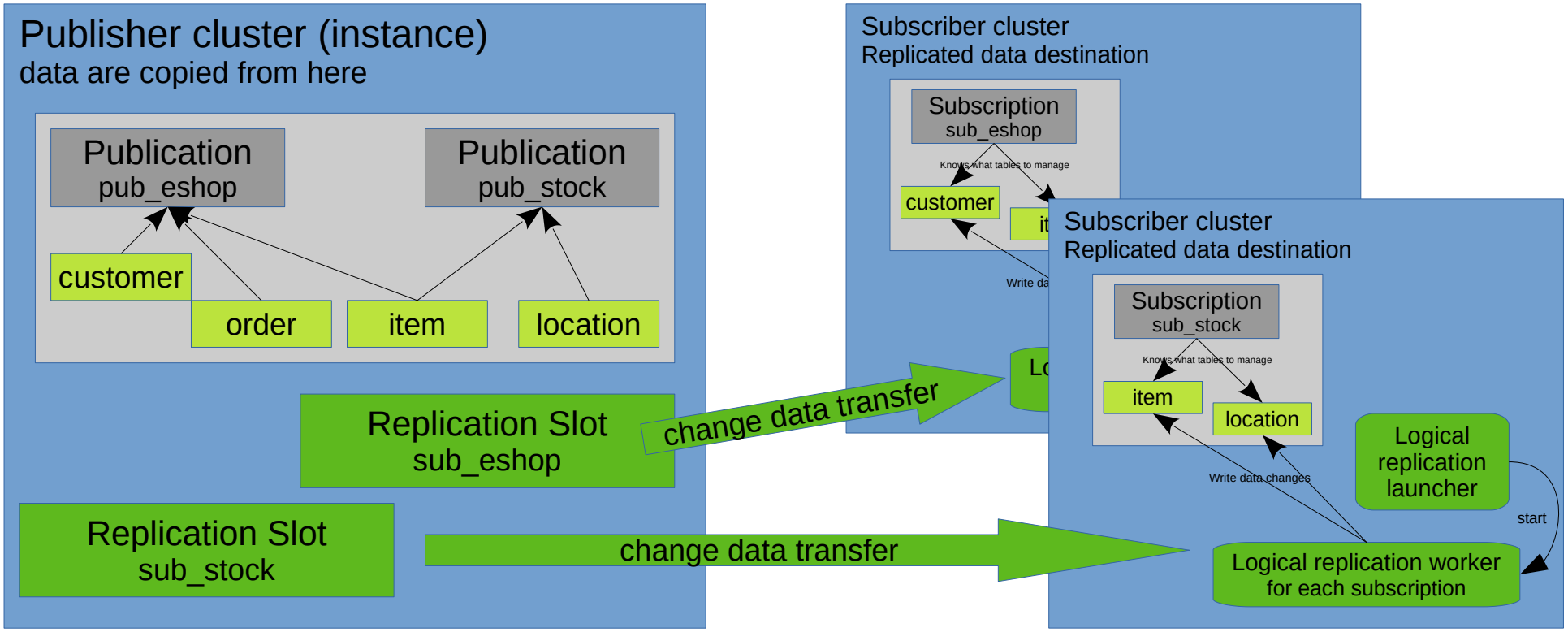
finmason



# Components


Publisher cluster (instance)
data are copied from here
Publication
catalog object
Add tables to publication
customer
invoice
Replication Slot
one for each subscription
WAL files
decoding
pgoutput
WAL sender
for each subscription
Connect (get replicated tables, create replication slot)
change data transfer
Subscriber cluster
Replicated data destination
Subscription
catalog object
Knows what tables to manage
customer
invoice
Write data changes
Logical
replication
launcher
start
Logical replication worker
for each subscription

# Components


Publisher cluster (instance)
data are copied from here
Publication
pub_eshop
Publication
pub_stock
customer
order
item
location
Replication Slot
sub_eshop
Replication Slot
sub_stock
change data transfer
change data transfer
Subscriber cluster
Replicated data destination
Subscription
sub_eshop
Knows what tables to manage
customer
Write da
Subscriber cluster
Replicated data destination
Subscription
sub_stock
Knows what tables to manage
item
location
Write data changes
Logical
replication
launcher
start
Logical replication worker
for each subscription

# Logical replication attributes

finmason

- Transactional (docs) for publications within a singe subscription
  - Commit order is preserved, changes are send to subscribers as they arrive (no batch mode)
  - asynchronous / synchronous
- Target table might have additional colums
- Allow selective replication all / insert / update / delete
- Can fire triggers on subscriber – **row level only** (subscriber suns with `session_replication_role` set to `replica` – triggers will not fire by default)
  - `ALTER TABLE name ENABLE REPLICA TRIGGER trigger_name;`

# Logical replication limitations

finmason

- No conflict resolution (docs)
- fully qualified table name match is required
- Base tables only
  - Take care of having identical partitions on publisher and subscriber
- No DDL replication (docs)
- No sequence replication
- TRUNCATE
  - <= PostgreSQL 10 not replicated
  - 11 and above supported
- NO Large Objects

# Yes, It worked (the boring part)

finmason

- GUC: wal_level = **logical**
- Create role for replication, tweak pg_hba.conf
  - `REPLICATION` has to be granted to replication role
- Create a table(s) on source database
  - With primary key – default replica identity
    - serial (or identity) column as surrogate key if desirable
  - Grant select on replicated tables to the role used for replication
    - Necessary for initial data copy (copy_data = ture)
- Create publication object
  - Add table(s) to publication

# Yes, It worked

finmason

- Create “same” table on target database
  - Instead of “BIG/SERIAL” consider use of BIG/INT column
    - Unless you have reasons why to do so (table receives “local” inserts)
- Create subscription
  - Wait a while for data to be copied (default behavior)
  - Superuser privilege required to create subscription
- Enjoy...

# Yes, It worked

finmason

- Create “same” table on target database
  - Instead “BIG/SERIAL” use BIG/INT column
    - Unless you have reasons why to do so (table receives “local” inserts)
- Create subscription
  - Wait a while for data to be copied (default behavior)
  - Superuser privilege required
- Enjoy...
- ***Configure replication state and lag monitoring***

# Replication lag monitoring

finmason

- SQL queries like for physical streaming replication
  - Publisher database
    - psql hint: `\watch 30`

```
SELECT pid, usename, application_name, state
    , pg_current_wal_lsn() AS current_lsn
    , sent_lsn
    , pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), sent_lsn)) AS sent_diff
    , write_lsn
    , pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), write_lsn)) AS write_diff
    , replay_lsn
    , pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn)) AS replay_diff
    , write_lag, flush_lag, replay_lag
FROM pg_stat_replication
ORDER BY application_name, pid;
```

# Lag monitoring is not enough

finmason

- An issue, like network outage might result in not running *wal sender process*, therefore this needs to be monitored
- `active_pid` column of `pg_replication_slots` has to be **not null**
  - Can be joined to `pg_stat_replication` column `pid`
- Instance log files on publisher and subscriber side has to be monitored as well

# Instance parameters - planning

Publisher

- `wal_level = logical`
- `max_worker_processes`
  - parallel workers, extension workers, subscriptions (if any)...
- `max_wal_senders`
  - Logical replication, streaming standby, streaming backup (pg_basebackup)
- `max_replication_slots`
  - At least max_wal_senders
- `wal_sender_timeout`
  - Network, workload peaks

Subscriber

- `max_worker_processes`
  - Considerations like on publisher
- `max_logical_replication_workers`
  - # subscriptions + initial sync
- `max_sync_workers_per_subscription`
  - Initial sync parallel processes
- `wal_receiver_timeout`
  - Network, workload peaks

Manual pages

# When it does not work… adrenalin raise

finmason

- Logical replication as of our experience is reliable solution able to handle large workload (trn/s or huge trns in terms of wal size)
- Despite some testing we had some issues resulting in stopped replication process
- None of the issues was caused by a bug, know root causes
  - reached known documented replication limitations
  - improper configuration
  - user code issue (because of know limitation)

# Conflict resolution – known limitation

Subscriber instance log file

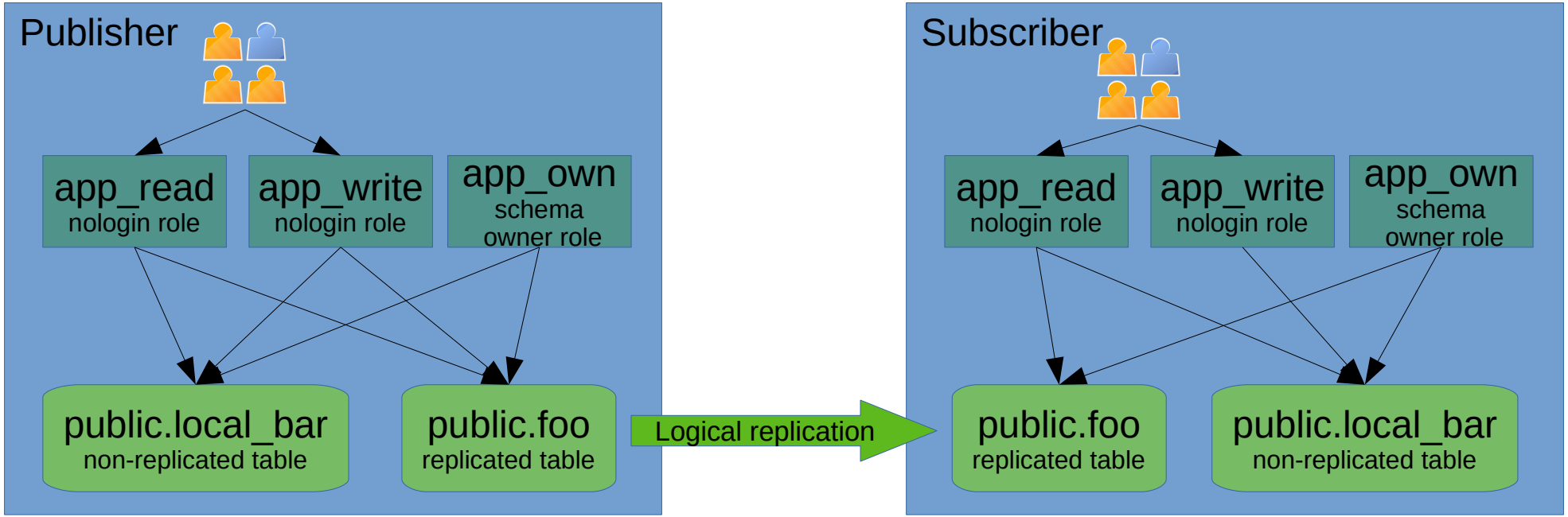
```
[23728] LOG:  logical replication apply worker for subscription
"sub_test" has started
[23728] ERROR:  duplicate key value violates unique constraint "foo_pkey"
[23728] DETAIL:  Key (foo_id)=(1) already exists.
[11289] LOG:  worker process: logical replication worker for subscription
16397 (PID 23728) exited with exit code 1
```

Resolution:

- Delete or update conflicting row(s) **on subscriber**
- Skip changes by using `pg_replication_origin_advance()`
  - **not recommended –** data will be inconsistent and changes to other rows/tables might be lost

# Conflict mitigation

- Do not trust your users not doing things you tell them they should not do… configure privileges to avoid it



# Conflict (again)

finmason

- Permissions are set, how it is possible?
  - Schema upgrade (flyway, Sqitch) runs using app_own role and a patch created new lookup tables with initial *data pre-loaded…*
- Mitigation
  - Do a *Code Review* for ***every*** patch
  - Choose good naming convention for tables
  - Use dedicates schema for replicated tables
    - Both can make code review easier to spot potential conflict

# Inappropriate objects configuration

finmason

```
postgres=# update foo set baz=-11 where bar=11;
ERROR:  cannot update table "foo" because it does not have a replica identity and publishes updates
HINT:  To enable updating the table, set REPLICA IDENTITY using ALTER TABLE.
```

- Replicated table on **publisher** missing replica identity
  - Default replica identity: Primary key
- Initial copy proceeded
  - Subsequent changes coming *later* fails on publisher fails
    - Replication still works, only the update statement on publisher DB fails
- Mitigation – carefully plan continues integration tests
  - Apply DDL changes in CI pipeline
  - Test processing including **real data changes** CI pipelines

# Missing replica identity on subscriber

- Replicated table has PK or other replica identity defined
- Initial copy / synchronization proceeded
- Subsequent changes on publisher proceed
- Subscriber fails to apply changes, replication is stopped




```
ERROR:  logical replication target relation "public.foo" has neither REPLICA IDENTITY index
nor PRIMARY KEY and published relation does not have REPLICA IDENTITY FULL
```

- Mitigation
  - Continuous testing including real data changes as already mentioned

# Don't Panic, see what might happen...

finmason

Replica identity

```
create table foo (foo_id serial PRIMARY KEY, product_id int not null,
serial_no text not null, constraint foo_key UNIQUE (product_id,
serial_no));
insert into foo(product_id, serial_no) select i, 'A'||i::text from
generate_series(1,5) gs(i);
alter publication pub_test add table foo;
```

```
table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      4 |          4 | A4
      5 |          5 | A5
(5 rows)
```

Publisher

```
create table foo (foo_id serial not null, product_id int not null, serial_no text not null, constraint foo_key
UNIQUE (product_id, serial_no));
postgres=# \d foo;
                                Table "public.foo"
   Column   |  Type   | Collation | Nullable |                Default
------------+---------+-----------+----------+---------------------------------------
 foo_id     | integer |           | not null | nextval('foo_foo_id_seq'::regclass)
 product_id | integer |           | not null |
 serial_no  | text    |           | not null |
Indexes:
    "foo_key" UNIQUE CONSTRAINT, btree (product_id, serial_no)
```

Subscriber

# Suddenly subscriber did not receiving changes...

```
alter subscription sub_test refresh publication with (COPY_DATA = true);
table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      4 |          4 | A4
      5 |          5 | A5
(5 rows)
```

Initial data copy was OK.

PK – default replica Identity
exists on publisher, so the update proceeded.

```
update foo set serial_no = 'B5' where foo_id = 5;
```

```
table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      4 |          4 | A4
      5 |          5 | B5
(5 rows)
```

```
table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      4 |          4 | A4
      5 |          5 | A5
(5 rows)
```

But the update was not applied on subscriber

# Panic action in place...

```
alter table foo replica identity using index foo_key;
```

```
LOG:  logical replication apply worker for subscription "sub_test" has started
ERROR:  logical replication target relation "public.foo" has neither REPLICA IDENTITY
index nor PRIMARY KEY and published relation does not have REPLICA IDENTITY FULL
```

```
postgres=# \d foo;
                                Table "public.foo"
   Column   |  Type   | Collation | Nullable |                Default
------------+---------+-----------+----------+---------------------------------------
 foo_id     | integer |           | not null | nextval('foo_foo_id_seq'
 product_id | integer |           | not null |
 serial_no  | text    |           | not null |
Indexes:
    "foo_pkey" PRIMARY KEY, btree (foo_id)
    "foo_key" UNIQUE CONSTRAINT, btree (product_id, serial_no) REPLICA IDENTITY
Publications:
    "pub_test"
```

WAL already received changes using PK as default Replica Identity,

therefore new replica identity using UQ constraint defined on publisher applies only to subsequent changes and did not resolve current issue...

```
postgres=# \d foo;
                                Table "public.foo"
   Column   |  Type   | Collation | Nullable |                Default
------------+---------+-----------+----------+---------------------------------------
 foo_id     | integer |           | not null | nextval('foo_foo_id_seq'::regclass)
 product_id | integer |           | not null |
 serial_no  | text    |           | not null |
Indexes:
    "foo_key" UNIQUE CONSTRAINT, btree (product_id, serial_no)
```

# Add publisher replica identity on subscriber

```
alter table foo add constraint foo_pkey PRIMARY KEY (foo_id);
```

```
postgres=# table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      4 |          4 | A4
      5 |          5 | B5
(5 rows)
```

Restoring replica identity which exists during changes written to WAL resolved the issue

Success!, Yes, but… not necessarily end of story.

# The road to hell...

finmason

# ..is paved with good intentions

finmason

Unique constraint satisfy business natural primary key role, we can drop existing primary key to **save** some **disk space**!

# Resolved? Test it!

finmason

Actually to spot the issue, it is not necessary to drop the primary key constraint, troubles were already “ordered” during initial panic actions…

```
update foo set serial_no = 'B4' where foo_id = 4;
```

```
postgres=# \d foo;
                              Table "public.foo"
   Column   |  Type   | Collation | Nullable |               Default
------------+---------+-----------+----------+-------------------------------------
 foo_id     | integer |           | not null | nextval('foo_foo_id_seq'::regclass)
 product_id | integer |           | not null |
 serial_no  | text    |           | not null |
Indexes:
    "foo_key" UNIQUE CONSTRAINT, btree (product_id, serial_no) REPLICA IDENTITY
Publications:
    "pub_test"
```

Replica identity is defined on publisher, update proceed.

```
table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      5 |          5 | B5
      4 |          4 | B4
(5 rows)
```

# Issue, again...

finmason

```
postgres=# table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      4 |          4 | A4
      5 |          5 | B5
(5 rows)
```

Well… no it does not work

```
LOG:  logical replication apply worker for subscription "sub_test" has started
ERROR:  publisher did not send replica identity column expected by the logical
replication target relation "public.foo"
LOG:  background worker "logical replication worker" (PID 6037) exited with exit code 1
```

```
postgres=# \d foo;
                          Table "public.foo"
   Column   |  Type   | Collation | Nullable |               Default
------------+---------+-----------+----------+-------------------------------------
 foo_id     | integer |           | not null | nextval('foo_foo_id_seq'::regclass)
 product_id | integer |           | not null |
 serial_no  | text    |           | not null |
Indexes:
    "foo_pkey" PRIMARY KEY, btree (foo_id)
    "foo_key" UNIQUE CONSTRAINT, btree (product_id, serial_no)
```

Subscriber did not know about replica identity change done on publisher, So it expects PK as replica identity in decoded wal changes it received

# We have read the manual by now...

```
alter subscription sub_test refresh publication with ( copy_data = false );
```

```
postgres=# table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      4 |          4 | A4
      5 |          5 | B5
(5 rows)
```

Subscriber knows about
replica identity change done on publisher,
but...

```
LOG:  logical replication apply worker for subscription "sub_test" has started
ERROR:  publisher did not send replica identity column expected by the logical replication target relation
"public.foo"
LOG:  background worker "logical replication worker" (PID 6526) exited with exit code 1
```

# Non default replica identity on both sides

Replica identity has to be same on publisher and subscriber, so when using not default replica identity, it has to be defined identically on both sides.

```
alter table foo replica identity using index foo_key;
```

```
postgres=# table foo;
 foo_id | product_id | serial_no
--------+------------+-----------
      1 |          1 | A1
      2 |          2 | A2
      3 |          3 | A3
      5 |          5 | B5
      4 |          4 | B4
(5 rows)
```

# Conclusion – expect human errors

finmason

- Publisher replica identity missing
  - on publisher, PostgreSQL protects you – updates/deletes will fail
    - add primary key on publisher
    - add same PK on all subscribers, refresh subscription on all subscribers
- Subscriber side replica identity issues
  - Keep calm, **resist temptation to define new replica identity**
    (see previous point, there has to be one on publisher, otherwise it will reject changes)
  - Identify replica identity on publisher (most probably PK) and create same on all subscribers – adding missing PK, replication will resume

# Coding issue...

finmason

- There are auditing triggers examples available
- "Premature optimization is the root of all evil", Donald Knuth, Computing Surveys, Vol 6, No 4, **December 1974**

```
CREATE OR REPLACE FUNCTION audit.if_modified_func() RETURNS TRIGGER AS $body$
DECLARE
    v_old_data TEXT;
    v_new_data TEXT;
BEGIN
…
EXCEPTION
    WHEN data_exception THEN
        RAISE WARNING '[AUDIT.IF_MODIFIED_FUNC] - UDF ERROR [DATA EXCEPTION] - SQLSTATE: %, SQLERRM: %',SQLSTATE,SQLERRM;
        RETURN NULL;
    WHEN unique_violation THEN
        RAISE WARNING '[AUDIT.IF_MODIFIED_FUNC] - UDF ERROR [UNIQUE] - SQLSTATE: %, SQLERRM: %',SQLSTATE,SQLERRM;
        RETURN NULL;
    WHEN OTHERS THEN
        RAISE WARNING '[AUDIT.IF_MODIFIED_FUNC] - UDF ERROR [OTHER] - SQLSTATE: %, SQLERRM: %',SQLSTATE,SQLERRM;
        RETURN NULL;
END;
$body$
LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = pg_catalog, audit;
```

# Do not clone templates blindly...

finmason

- Should an audited action succeeded with warning if there was an issue with creating audit record itself?
  - Not a topic for this talk...
- We had several triggers inspired by the auditing trigger in code base more than year, exceptions handling within trigger functions provides application logic/context structured error messages / custom exceptions, it worked well with no issues.
- More than 6 months replication works like a charm (yes, we already learned about conflicts, replica identity… easy stuff)

# But the dormant disaster was in place...

finmason

- Signs of production disaster
  - Server become terribly slow
  - Monitoring warns us – swap is being used
- Curious, swap was not an issue before
- Postgres was using the memory
- Restart was not a solution, memory and swap was exhausted within few minutes (16GB RAM, 30GB swap)
- Lost weekend

# Some observations

- LSNs on the replication stats view were not advancing on subscriptions
- Timeouts occurs, while network was constantly check and not considered as issue root cause

```
open("pg_replslot/sub_usd/xid-6040508-lsn-429-57000000.snap", O_WRONLY|O_CREAT|O_APPEND, 0600) = 15
write(15, "\334\0\0\0\0\0\0H\3\234W)\4\0\0\0\0\0\0\0\0\0\177\6\0\0%@\0\0"..., 220) = 220
close(15)                                  = 0
```

WAL senders were writing dome files

Replication workers on subscriptions were waiting

```
recvfrom(7, 0x55f56a3783c3, 5, 0, NULL, NULL) = -1 EAGAIN (Resource temporarily unavailable)
epoll_create1(EPOLL_CLOEXEC)               = 10
epoll_ctl(10, EPOLL_CTL_ADD, 12, {EPOLLIN|EPOLLERR|EPOLLHUP, {u32=1781282352, u64=94512536630832}}) = 0
epoll_ctl(10, EPOLL_CTL_ADD, 9, {EPOLLIN|EPOLLERR|EPOLLHUP, {u32=1781282376, u64=94512536630856}}) = 0
epoll_ctl(10, EPOLL_CTL_ADD, 7, {EPOLLIN|EPOLLERR|EPOLLHUP, {u32=1781282400, u64=94512536630880}}) = 0
epoll_wait(10, [], 1, 1000)                = 0
```

# Operation restore, iteration one

finmason

- We have created clones of publisher and one subscriber instance for analyses
- Disabled PostgreSQL service start on publisher, activated per process memory monitoring
- **wal sender** processed were identified as the unusual memory consumers
- Subscriptions were dropped (I takes several hours, we'll see later why), re-created with `copy_data = false` and gory weekend of manual data synchronization (thanks to postgres_fdw, we were able to do so, using application time stamp columns)

# Never more

finmason

# Analyses

- On cloned machines while extending swap to 50GB and updating configuration parameters
  - `wal_sender_timeout` (from default 60 to 600 sec)
  - `wal_receiver_timeout` (from default 60 to 600 sec)
- Timeout messages (see bellow) disappeared
  - Several hours later, replication was streaming again (state catch-up => streaming)

Publisher instance

```
LOG:  terminating walsender process due to replication timeout
```

```
ERROR:  could not receive data from WAL stream: SSL connection has been closed unexpectedly
LOG:  worker process: logical replication worker for subscription 37932 (PID 8657) exited with exit
code 1
```

Subscriber instance

# Root cause one (kind of)

finmason

- Under some kind of workload, wal sender did not communicate with wal receiver within default timeout of 60 seconds
  - So it seems timeouts are there to cover network and some other issues
- Logical replication can be weird and allocate lots of memory, so extend SWAP and place it on NVMe, it is expensive, but might help to survive form such issues
  - Is the memory allocation a bug? => future investigation needed, test-case needed

# Few months later...

# Monitoring alert – growing replication lag

finmason

- Replication was obviously broken, again, but we were prepared – 300GB NVMe swap and 60GB of RAM
- System was slow, but much faster than with swap on HDD
  - We were able to connect there
- Each of our 3 wal sender processes allocated 75GB of memory (75 GB, Vss, 17GB Rss)
- No timeouts, only the replication lag growth over time 250GB...

# strace pattern for wal sender

finmason

- Write to files  like “`pg_replslot/sub_usd/xid-6040508-lsn-429-57000000.snap`”
- Read all of them afterwards (and sometimes send some data over network to wal receivers)
- Delete all of them afterwards
- Each of the phases above lasts 3-4 hours
- pg_replslot/<slot_name> contain about 17 000 000 files, usually small ones
- Ext4, nor ZFS can’t handle such amount of files within single directory while keeping reasonable performance (`find -type f | wc -l` takes 10+ minutes)

# Replication resumes

finmason

- Finally, after long night on lunchtime next day, replication changes its state to streaming
- VSS was not released, Rss shrinks to 3,5GB or RAM
- Bonus! We have learned new logical replication feature
  - **disable subscription**
    - Stop wal sender / receiver processes as expected
  - Resume starts them (Vss memory was released, swap empty)
  - All data changes during “disable time” were queued in replication slot, therefore disable/enable does not mean lost changes (in the end, it makes sense, but I haven’t found it in docs)

# It is not a bug, known limitation

finmason

- We have also collected much more data, perf records (all time was spent on ext4 functions)...
- So we have tried to find similar issues on the web using better keywords
- Nice presentation pgconfasia 2017, we were aware before, recommended reading
- This amazing logical replicationinternals article pointed us right way – `reorderbuffer.c`
- This post in pgsql-hackers makes us sure, what to look for – **subtransactions**

# subtransactions

finmason

- Because of the monitoring, we knew when the issue occurs
  - Standard operation did not caused this issue yet
    - We have checked processed data volumes – as usual
  - Logging long statements is useful – we have found very simple delete from table without predicates, lasting 20 minutes, issued by a personal account
- Delete was used instead of truncate, to make sure that delete trigger on that table can do it’s archiving job
- The table was not replicated (not in any publications)

# subtransactions cont.

- Other deletes similar amounts of rows were not issue in past on similar tables, except, they did not have the archiving trigger
- Manual pages (again!): "*Also, a block containing an EXCEPTION clause effectively forms a subtransaction that can be rolled back without affecting the outer transaction.*"
- That is the true root cause – **implicit subtransactions** – used just for more convenient error reporting
  - By the way, there is also performance impact due to subtransaction handling

# Conclusion

finmason

- We were able to create reproducible test case
- As soon as we removed exception handling from row level triggers, the test case proceeded without any issue
- We were testing transaction on 20E6 rows in single transaction, it takes some time to parse this data, during that time some replication lag raises, streaming state was not lost, works well
- We have also tried “large” transaction in terms of WAL size (delete from table 6GB – long data in text columns) without any issue
- Currently we consider **logical replication as stable well working**

# So Long, and Thanks for All the Fish

finmason

Questions

Thanks for your time