Logical Replication – Handling of Large Transactions

Hayato Kuroda kuroda.hayato@fujitsu.com
Hou Zhijie

Self Introduction

- Hayato Kuroda
  - Me.
  - Living in Japan.
  - Working at Fujitsu since 2018.
- Hou Zhijie
  - Co-speaker.
  - Could not be here...

Agenda

- Logical decoding & logical replication.
- Decoding for large transactions in earlier versions.
  - Prior to PG12.
  - Improvements in PG13.
  - Improvements in PG14.
- Parallel apply – Next enhancement in PG16.

Logical decoding & replication

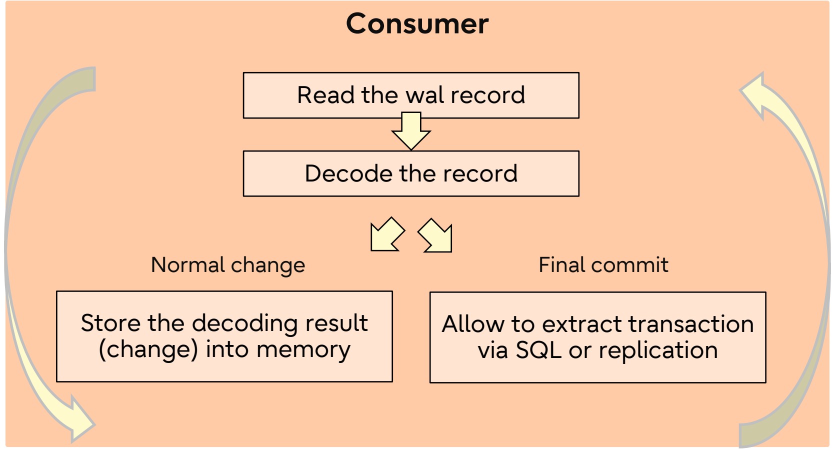
Workflow Changes

Logical decoding

FUJITSU

- An infrastructure that transforms all persistent changes into another format.
  - The specifics of this format are determined by the output plugin.
  - The output can be interpreted without needing detailed knowledge of the database's internal state.
- Implemented based on the Write-Ahead Log (WAL).
- Output plugin modules provide rich callback functions to allow user customization based on their requirements.

Logical decoding | Workflow



Example: consumed by the function

```
postgres=# INSERT INTO tbl VALUES (1);
INSERT 0 1
postgres=# SELECT * FROM pg_logical_slot_get_changes('test', NULL, NULL);
    lsn    | xid |                    data
-----------+-----+-------------------------------------------
 0/1558710 | 749 | BEGIN 749
 0/1558710 | 749 | table public.tbl: INSERT: id[integer]:1
 0/1558780 | 749 | COMMIT 749
(3 rows)
```

- The consumer process reads and decodes the WAL records.
- The decoded results are stored in memory on a per-transaction basis (txn).
- Stored data can be consumed either by calling functions via SQL, or by using the streaming replication protocol.
- If the decoded results are consumed by the streaming replication protocol, they are sent to downstream and cleaned up when the transaction is committed.

Logical replication

FUJITSU

- A method of replicating data objects and their changes, based upon their replication identity.
- Uses a publish and subscribe model.
  - The upstream node is called publisher.
  - The downstream is subscriber.
- Allows fine-grained control over both data replication and security.

- Typical use-cases:
  - Sending incremental changes in a single database to subscribers as they occur.
  - Replicating between different major versions of PostgreSQL.
  - Replicating between PostgreSQL instances on different platforms.
  - …

Logical replication | Usage

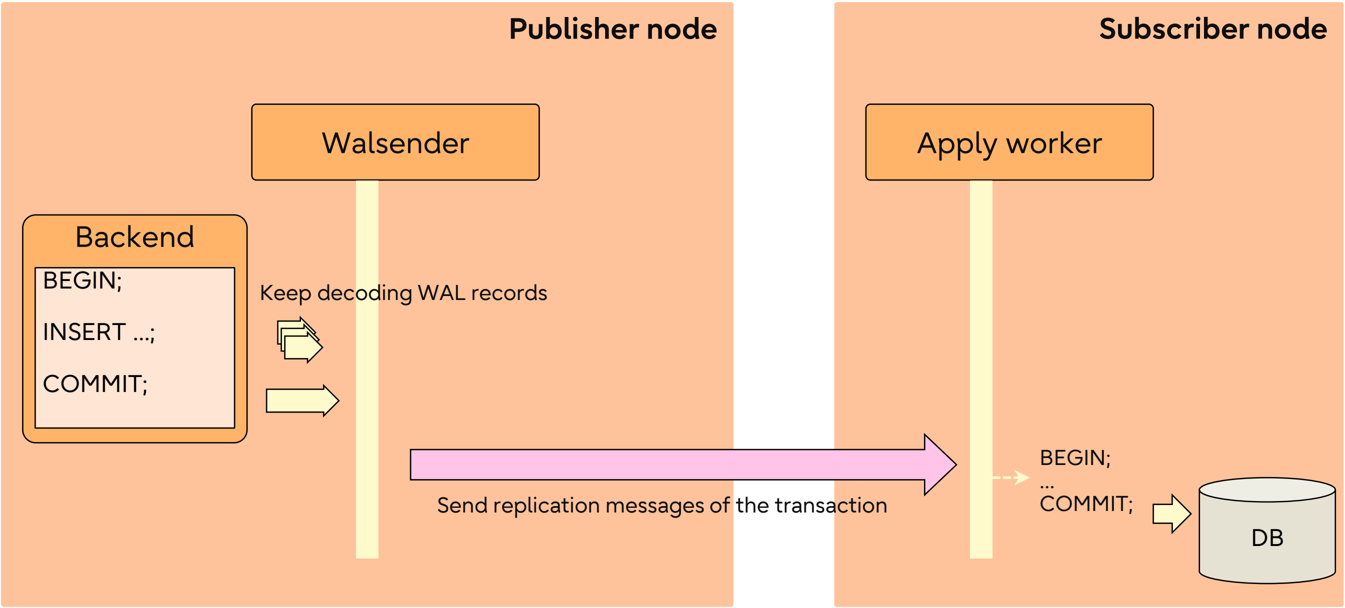
- The publication must be defined on an upstream node.

```
postgres=# CREATE PUBLICATION pub FOR ALL TABLES;
CREATE PUBLICATION
postgres=# SELECT * FROM pg_publication;
  oid   | pubname | pubowner | puballtables | pubinsert | pubupdate | pubdelete | pubtruncate | pubviaroot
-------+---------+----------+--------------+-----------+-----------+-----------+-------------+------------
 16396 | pub     |       10 | t            | t         | t         | t         | t           | f
(1 row)
```

- Then a down stream node subscribes the publication.

```
postgres=# CREATE SUBSCRIPTION sub CONNECTION 'user=postgres dbname=postgres port=5431' PUBLICATION pub;
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
postgres=# SELECT oid, subdbid, subname, subconninfo FROM pg_subscription;
  oid   | subdbid | subname |               subconninfo
-------+---------+---------+-----------------------------------------
 16402 |       5 | sub     | user=postgres dbname=postgres port=5431
(1 row)
```

Logical replication | Workflow



Logical replication | Monitoring

- The progress of logical replication can be checked by reading the *pg_stat_replication* (on publisher) and *pg_stat_subscription* (on subscriber) views.

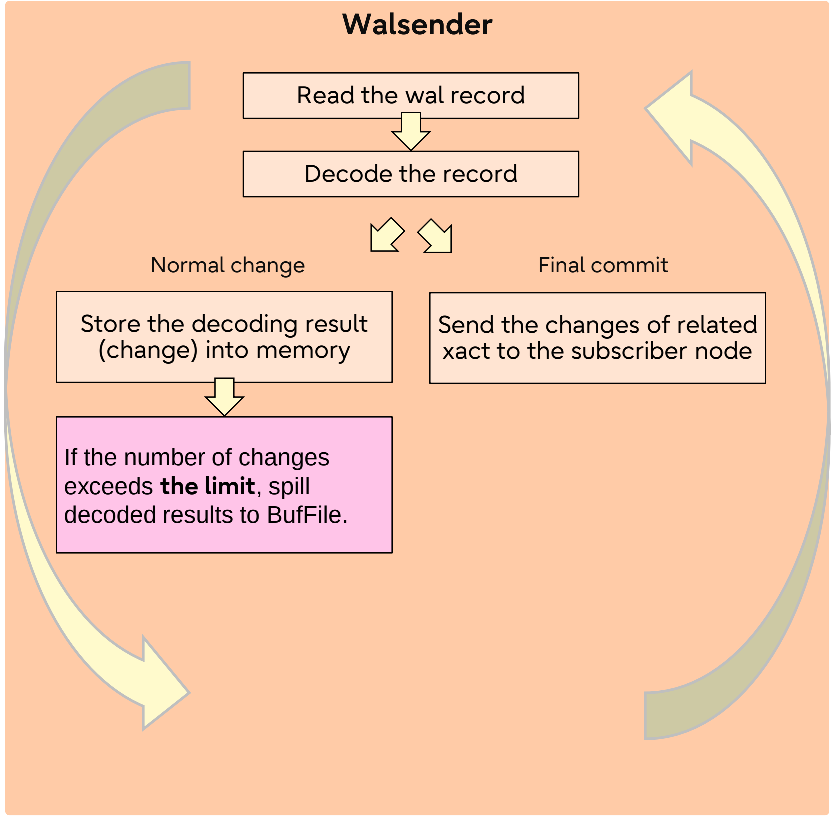
```
publisher=# SELECT pid, application_name, state, sent_lsn, replay_lsn, replay_lag
            FROM pg_stat_replication;
  pid  | application_name |   state   | sent_lsn  | replay_lsn |   replay_lag
-------+------------------+-----------+-----------+------------+-----------------
 26201 | sub              | streaming | 0/37B2070 | 0/37B2070  | 00:00:00.017882
(1 row)
```

```
subscriber=# SELECT subid, subname, pid, received_lsn, last_msg_receipt_time
             FROM pg_stat_subscription;
 subid | subname |  pid  | received_lsn |     last_msg_receipt_time
-------+---------+-------+--------------+-------------------------------
 16388 | sub     | 26198 | 0/9EFED40    | 2023-05-15 08:45:55.720919+00
(1 row)
```

Decoding for large transactions

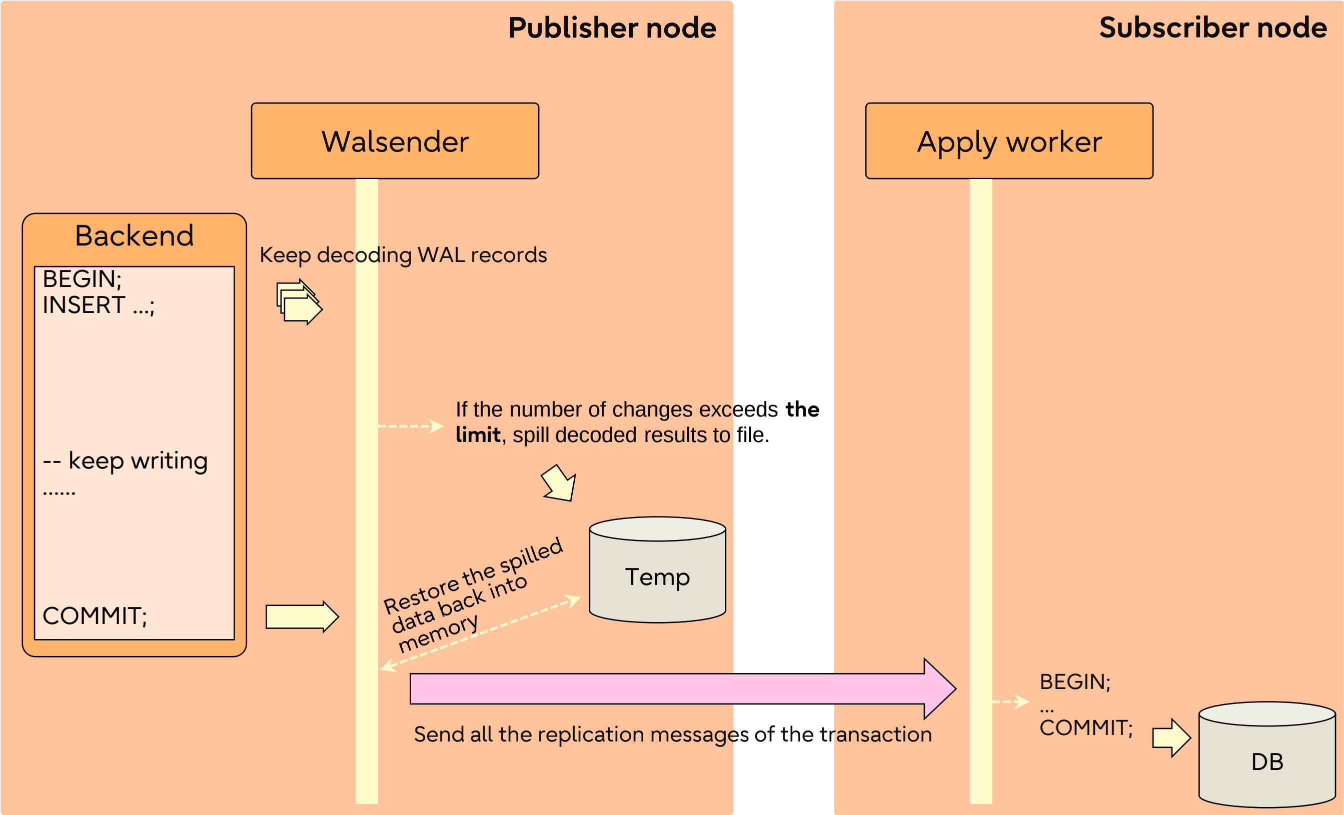
... in earlier versions

Decoding for large transaction (in PG12)



- The decoded results are stored in memory on a per-transaction basis (txn).
- If the number of changes exceeds the limit (Fixed amount: 4096), decoded results are spilled to temporary file.

Replication for large transaction (in PG12)



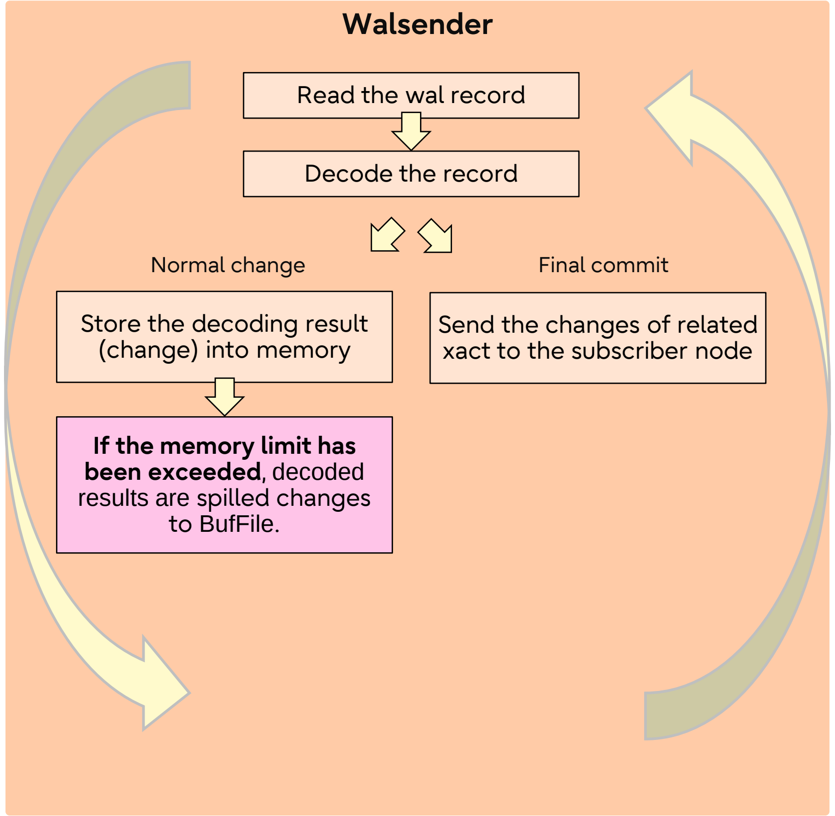
Cons in PG12 and Improvements in PG13

- PG12 cannot precisely control memory size used by walsender.
  - Controlling memory usage is a challenge.
  - If the publisher node has enough memory, it seems to be inefficient.

- In PG13, new GUC *logical_decoding_work_mem* has been introduced
  - Specifies the maximum amount of memory to be used by logical decoding.
  - Default is 64MB, and the minimum is 64KB.

```
logical_decoding_work_mem = 128MB
```

Improvements in PG13



- The decoded results are stored in memory on a per-transaction basis (txn).
- ~~If the number of changes exceeded the limit (Fixed amount: 4096), decoded results are spilled to disk.~~
- When the **memory** limit (GUC “logical_decoding_work_mem”) is exceeded, decoded results are spilled to temporary file.

Cons in PG13 and Improvements in PG14

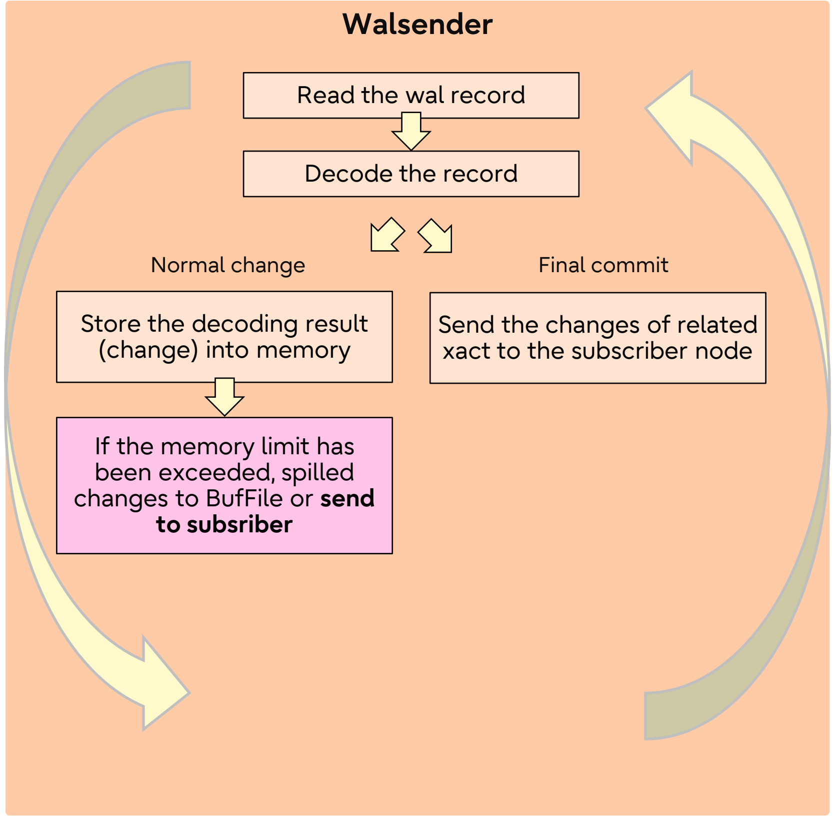
FUJITSU

- Transactions are only sent to the subscriber after they've been committed.
  - Consequently, large transactions may trigger the network congestion,
  - …leads apply lag.

- In PG14, new subscription parameter *streaming* was introduced
  - Specifies whether in-progress transactions can be streamed for this subscription.
  - Default value is false.

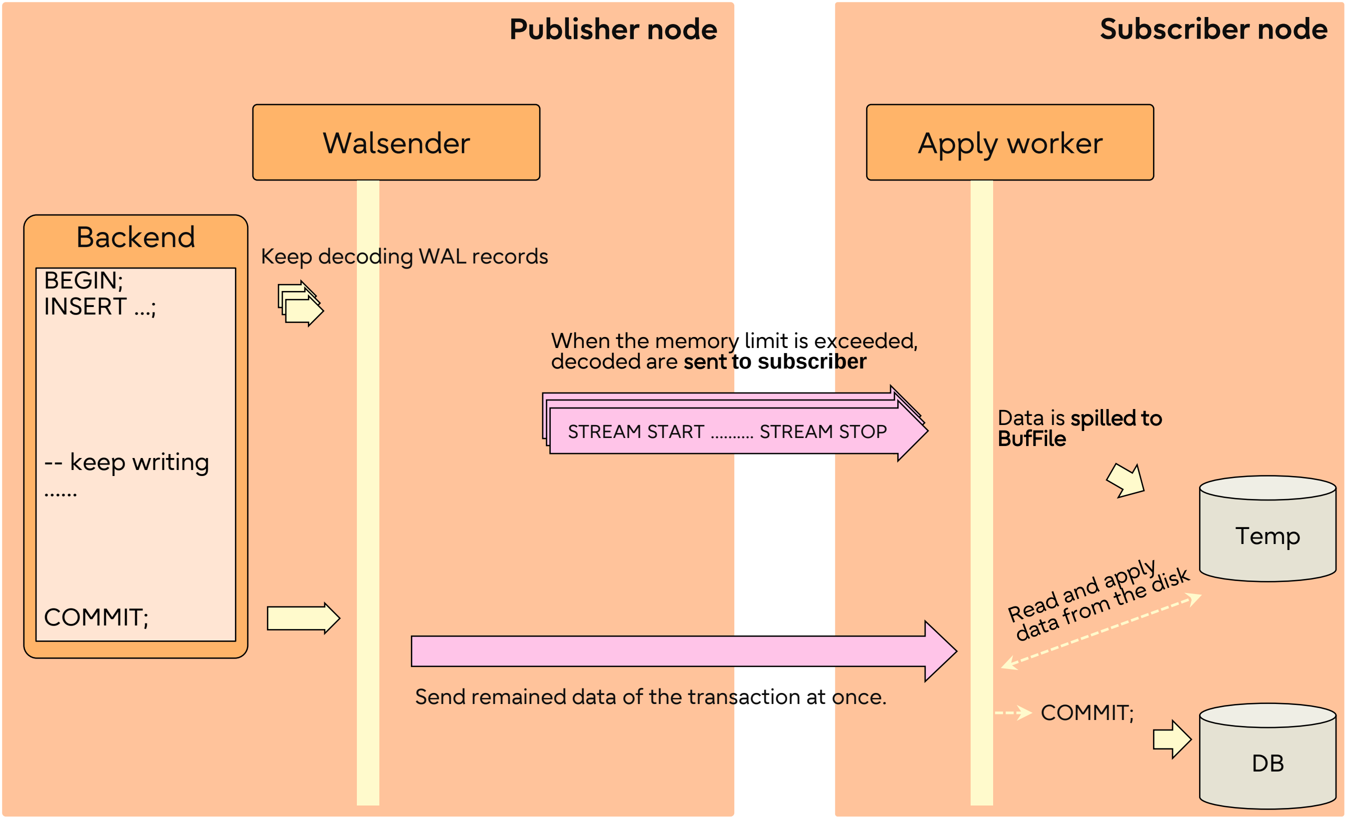
```
postgres=# CREATE SUBSCRIPTION sub CONNECTION 'dbname=postgres'
                 PUBLICATION pub WITH (streaming = TRUE);
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
postgres=# SELECT subname, substream FROM pg_subscription;
 subname | substream
---------+-----------
 sub     | t
(1 row)
```

Improvements in PG14



- The decoded results are stored in memory on a per-transaction basis (txn).
- ~~If the number of changes exceeded the limit (Fixed amount: 4096), decoded results are spilled to disk.~~
- ~~When the **memory** limit (GUC "logical_decoding_work_mem") is exceeded, decoded results are spilled to temporary file.~~
- When the memory limit (GUC "logical_decoding_work_mem") is exceeded, spilled changes to disk **or send to subscriber**.

Improvements in PG14



Improvements in PG14 | Monitoring

- The system view *pg_stat_replication_slots* has been added.

```
postgres=# SELECT slot_name, spill_txns, spill_count,
                  spill_bytes, total_txns, total_bytes
           FROM pg_stat_replication_slots;
 slot_name | spill_txns | spill_count | spill_bytes | total_txns | total_bytes
-----------+------------+-------------+-------------+------------+-------------
 sub       |         87 |         551 |    66398400 |         96 |    67046400
(1 row)
```

```
postgres=# SELECT slot_name, stream_txns, stream_count,
                  stream_bytes, total_txns, total_bytes
           FROM pg_stat_replication_slots;
 slot_name | stream_txns | stream_count | stream_bytes | total_txns | total_bytes
-----------+-------------+--------------+--------------+------------+-------------
 sub       |          96 |          275 |    116812800 |        132 |   126403200
(1 row)
```

Cons in PG14 & PG15

- The performance of replicating large transaction still has room for improvement.
  - Disk IO
    - Changes must be stored on the disk initially before they can be applied.
  - Apply lag
    - Changes can only be applied at the end of the transaction, resulting in a possible slowdown of the transaction.

Parallel Apply

Next enhancement developed in PG16

Overview and Advantage

FUJITSU

- An alternative approach for handling large transactions.
- Could be available in the upcoming release

- If the parallel mode is enabled, the subscriber applies streamed in-progress transactions IMMEDIATELY.
- The subscriber can handle in-progress transactions IN PARALLEL.

- Parallel Apply enables faster and more efficient handling of large transactions.
  - Does not wait for COMMIT message from the publisher.
  - Does not serialize replication messages into files.

- This can be widely used if users allow streaming of intermediate transactions.
  - Batch operation on logical replication system.

Usage

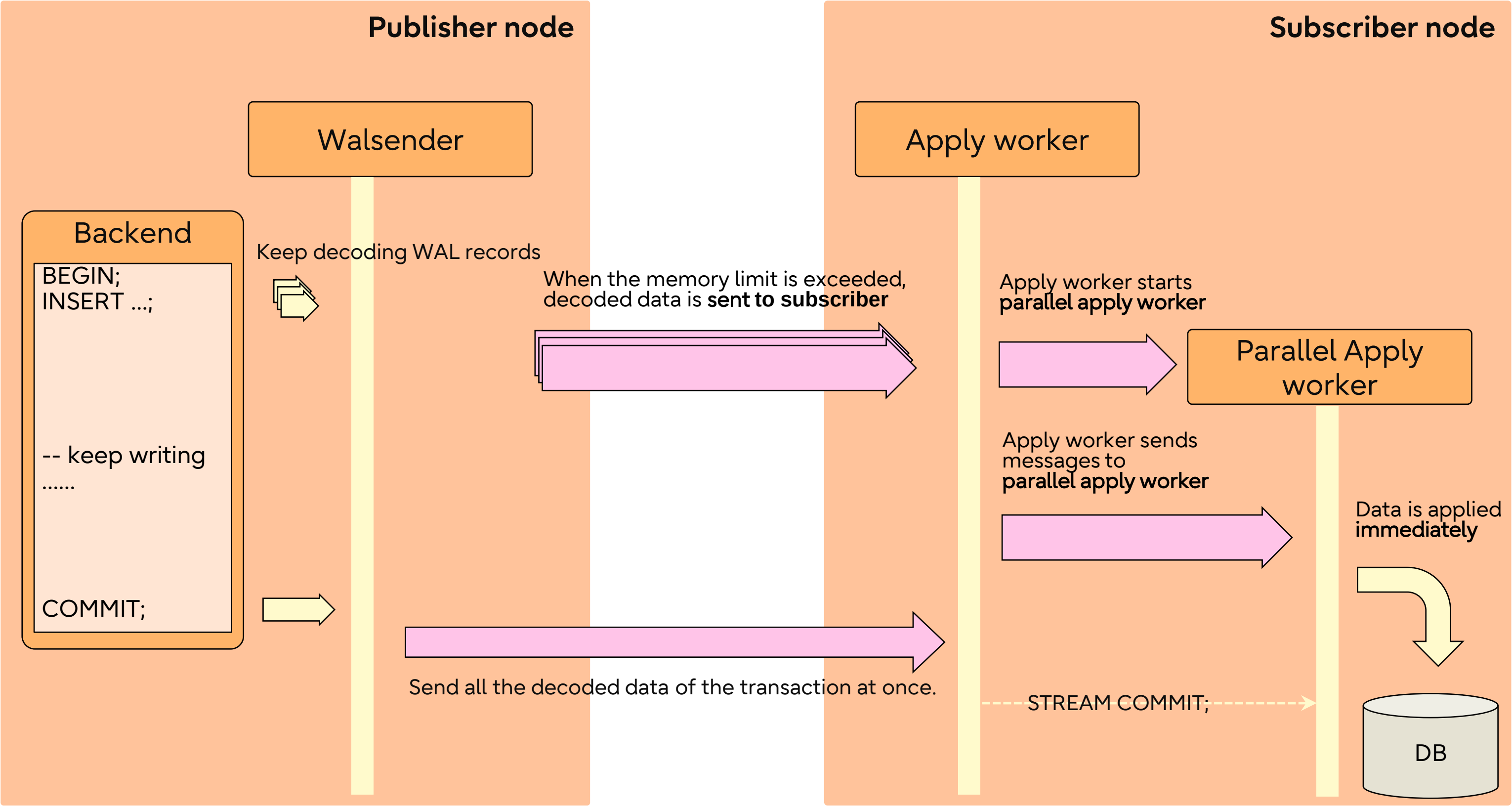
- Users must set the subscription parameter *streaming* to *parallel*.

```
postgres=# CREATE SUBSCRIPTION sub CONNECTION 'dbname=postgres'
                 PUBLICATION pub WITH (streaming = parallel);
NOTICE:  created replication slot "sub" on publisher
CREATE SUBSCRIPTION
postgres=# SELECT subname, substream FROM pg_subscription;
 subname | substream
---------+-----------
 sub     | p
(1 row)
```

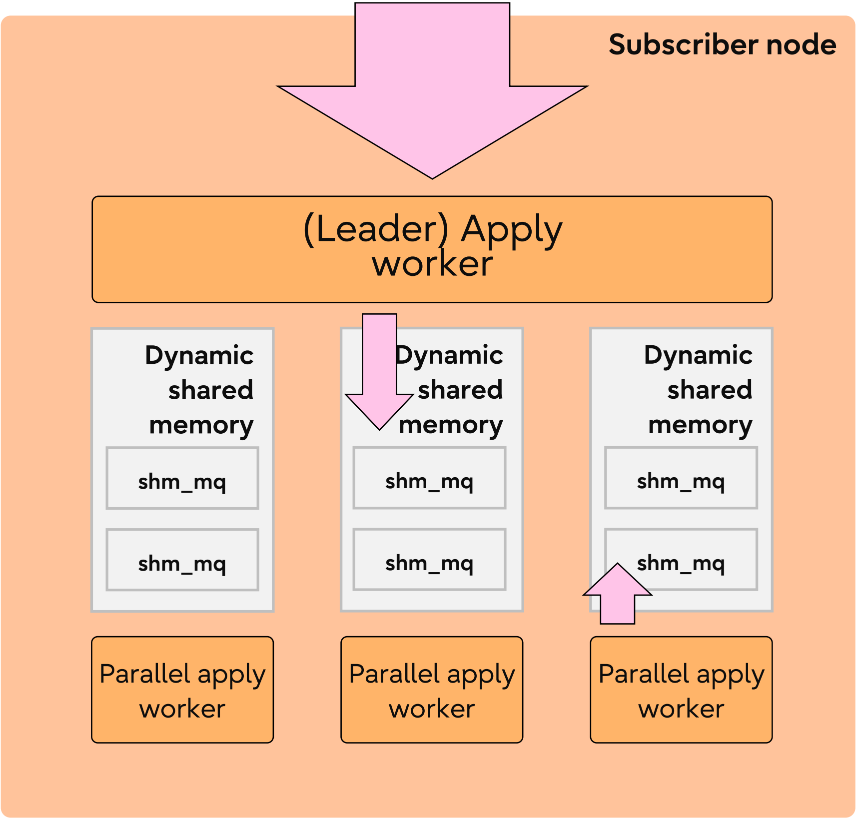
- The parallelism can be tuned by parameter *max_parallel_apply_workers_per_subscription.*

```
max_parallel_apply_workers_per_subscription = 5
```

Architecture

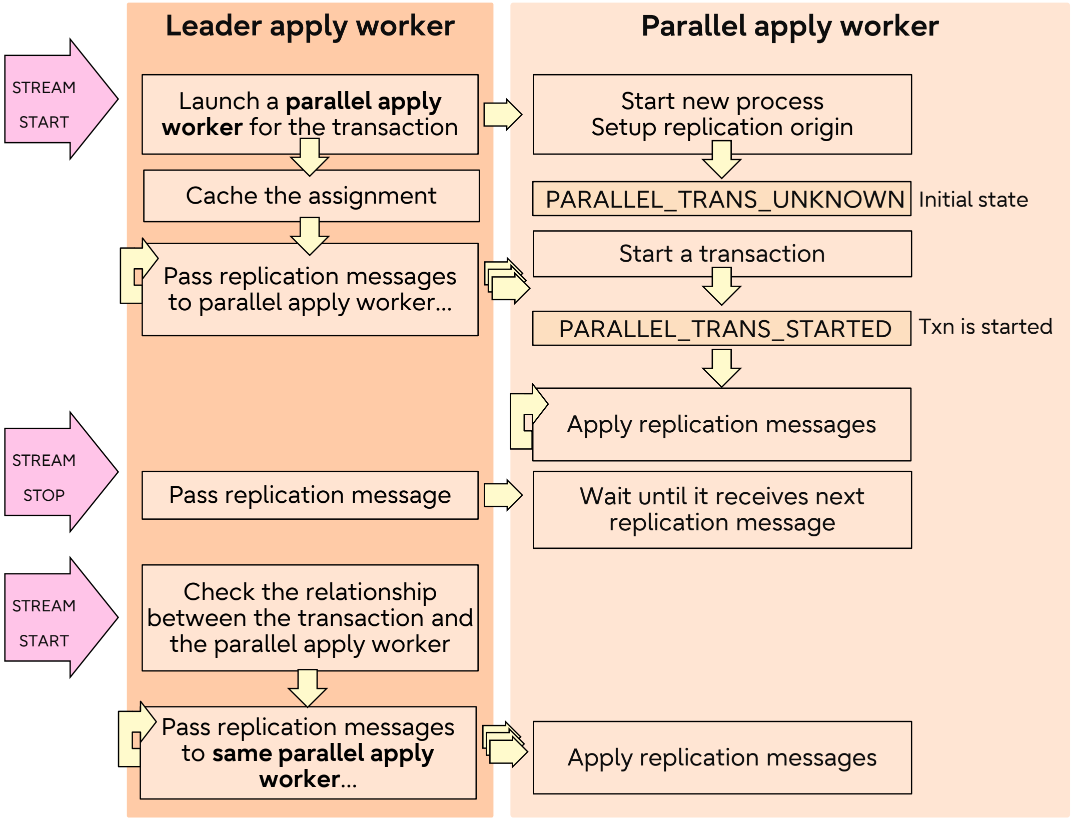


Parallel apply worker



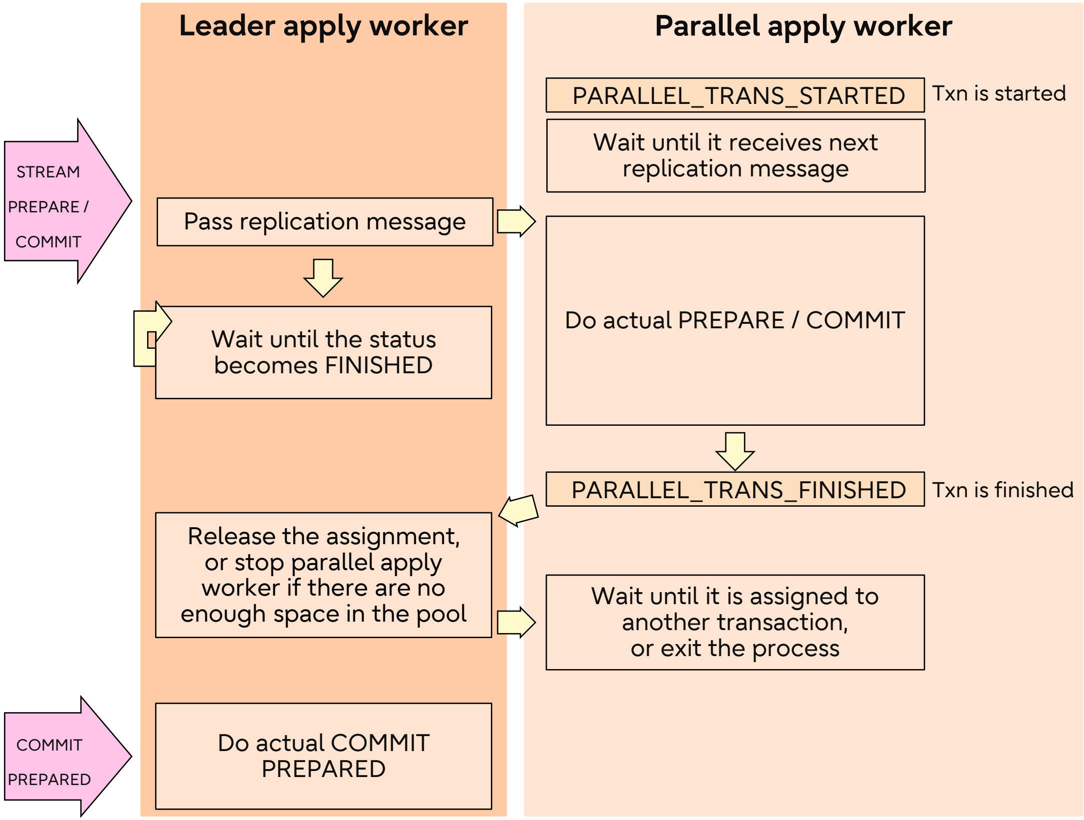
- The parallel apply worker is started when in-progress transactions are streamed.
- Multiple parallel apply workers can run per subscription; the parallelism is based on *max_parallel_apply_workers_per_subscription.*
- Each parallel apply worker is assigned up to one transaction, and the assignment will never be changed during the apply handing.
- The leader apply worker communicates with parallel apply workers through dynamic shared memory and shared message queues.

Basic workflow



- When the leader apply worker receives the initial segment of an in-progress transaction, it launches a new parallel apply worker.
- The parallel apply worker applies received messages immediately.
- If streaming stops, the assigned parallel apply worker waits until it receives the next set of replication messages.
- When the leader apply worker receives the next chunk, it resumes sending replication messages to the same parallel apply worker.

Commit protocol



- When the leader apply worker receives a PREPARE / COMMIT message, it sends the message to the assigned parallel apply worker and waits for it to finish applying the transaction.
- The parallel apply worker performs the actual PREPARE / COMMIT action and marks the transaction status as FINISHED.
- The leader apply worker removes the relationship between the streamed transaction and the parallel apply worker.
- The COMMIT PREPARED operation is handled by the leader apply worker.

Monitoring

- The presence of parallel apply workers can be checked by reading the *pg_stat_activity* and *pg_stat_subscription* views.

```
postgres=# SELECT datname, pid, leader_pid, state, backend_xid, backend_type
           FROM pg_stat_activity
           WHERE backend_type LIKE 'logical replication parallel worker';
 datname  | pid  | leader_pid | state | backend_xid |            backend_type
----------+------+------------+-------+-------------+-------------------------------------
 postgres | 2169 |       2165 | idle  |             | logical replication parallel worker
(1 row)
```

```
postgres=# SELECT subid, subname, pid, leader_pid, received_lsn FROM pg_stat_subscription;
 subid | subname | pid  | leader_pid | received_lsn
-------+---------+------+------------+--------------
 16390 | sub     | 2169 |       2165 |
 16390 | sub     | 2165 |            | 0/1550108
(2 rows)
```

Failure Handling

- The parallel apply worker exits if it meets an ERROR.
- Before exiting, it puts the error message in the shared message queue and sends a signal to the leader apply worker.
- When the leader apply worker becomes aware of the issue, it pops the message, reports it to the server log, and exits.
- … After this processes follow same procedure as the non-parallel case.

- If users want to skip the transaction, they can check the LSN of the transaction from the log and execute *ALTER SUBSCRIPTION SKIP* command.

Failure Handling

FUJITSU

- Sometimes the finish LSN of the remote transaction cannot be reported on the log.
- The reason is that the streamed in-progress transaction initially lacks a final_lsn, which is assigned at the end of the transaction.

```
[12999] ERROR:  duplicate key value violates unique constraint "tbl_pkey"
[12999] DETAIL:  Key (id)=(1) already exists.
[12999] CONTEXT:  processing remote data for replication origin "pg_16390" during message type "INSERT" for
replication target relation "public.tbl" in transaction 732
[12974] ERROR:  logical replication parallel apply worker exited due to error
[12974] CONTEXT:  processing remote data for replication origin "pg_16390" during message type "INSERT" for
replication target relation "public.tbl" in transaction 732
        logical replication parallel apply worker
```

- In this situation, users must disable parallel mode temporarily and trigger the same conflict again.

```
postgres=# ALTER SUBSCRIPTION sub SET (streaming = on);
ALTER SUBSCRIPTION
postgres=# SELECT subname, substream FROM pg_subscription;
 subname | substream
---------+-----------
 sub     | t
(1 row)
```

Challenges of Implementing Parallel Apply

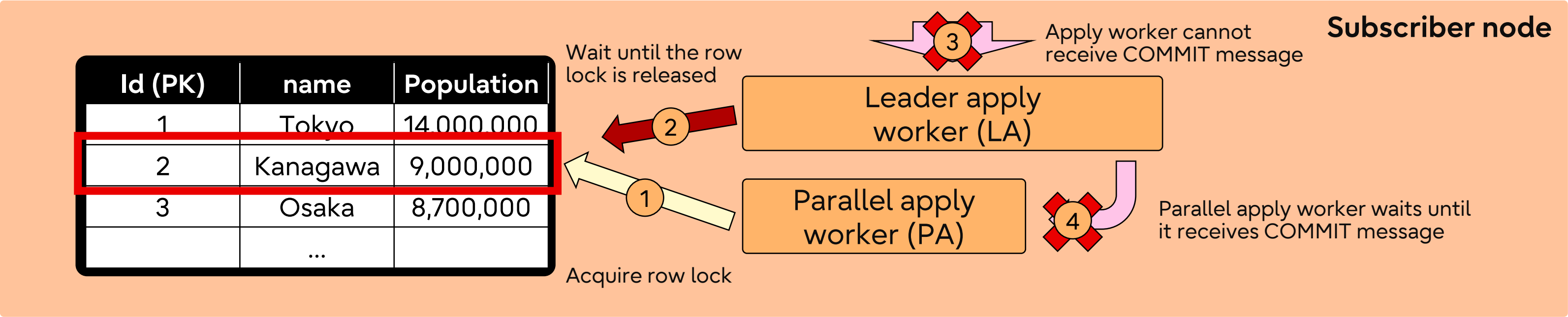
FUJITSU

- Due to the concurrency, there were some additional risks of deadlocks.
- The deadlock might happen if tables that were independent on the publisher side become dependent on the subscriber side.

- Three considerations were found during development, and they were already solved.
  - Consideration #1: Deadlock between the leader apply worker and the parallel apply worker.
  - Consideration #2: Deadlock between the leader apply worker and parallel apply workers.
  - Consideration #3: Deadlock when the shared message queue is full.

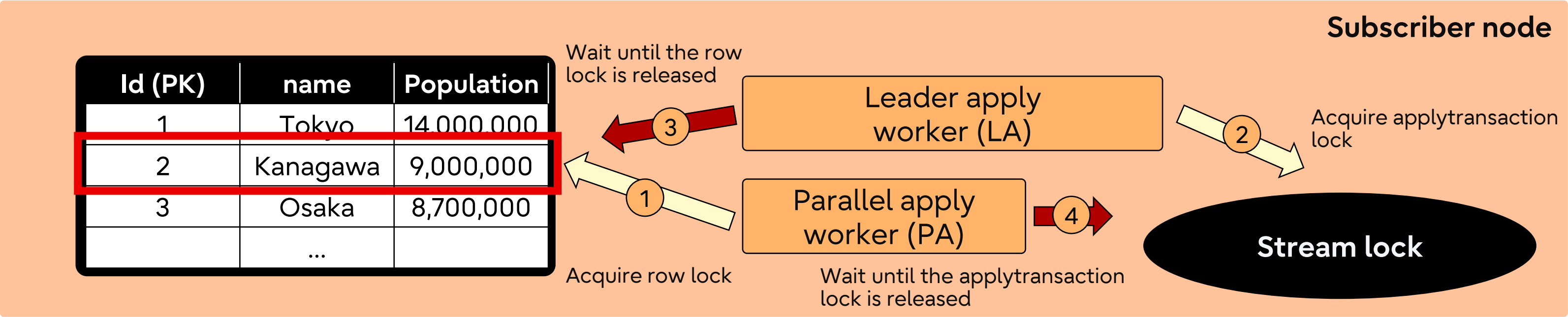
Consideration #1: Deadlock between the leader apply worker and the parallel apply worker

- Assume that two transactions are executing concurrently on subscriber.
- One transaction has been handling by PA, and another one is by LA.
- LA is waiting for PA due to the primary key constraint of the subscribed table, while PA is waiting for LA to send the next stream of changes or a transaction finish command message.
- The PostgreSQL lock manager cannot detect the deadlock because the processes do not form a cycle in the wait-for-graph.



Consideration #1: Deadlock between the leader apply worker and the parallel apply worker

- A new session-level lock (stream lock) is introduced.
- The Lock is acquired using the subscription ID and the related transaction ID.
- The LA acquires the lock before sending STREAM STOP, and releases it after sending STREAM START/ABORT/PREPARE/COMMIT.
- The PA acquires the lock after processing STREAM STOP, and releases it immediately
- The wait-for-graph becomes cyclic.



Performance improvements: Elapsed time

- Performance testing is done with following steps:

1. Construct a synchronous logical replication system.
2. Insert tuples via *"psql –c …"*.
3. Measure the execution time of the command.

- Both publisher and subscriber were located on the same server.

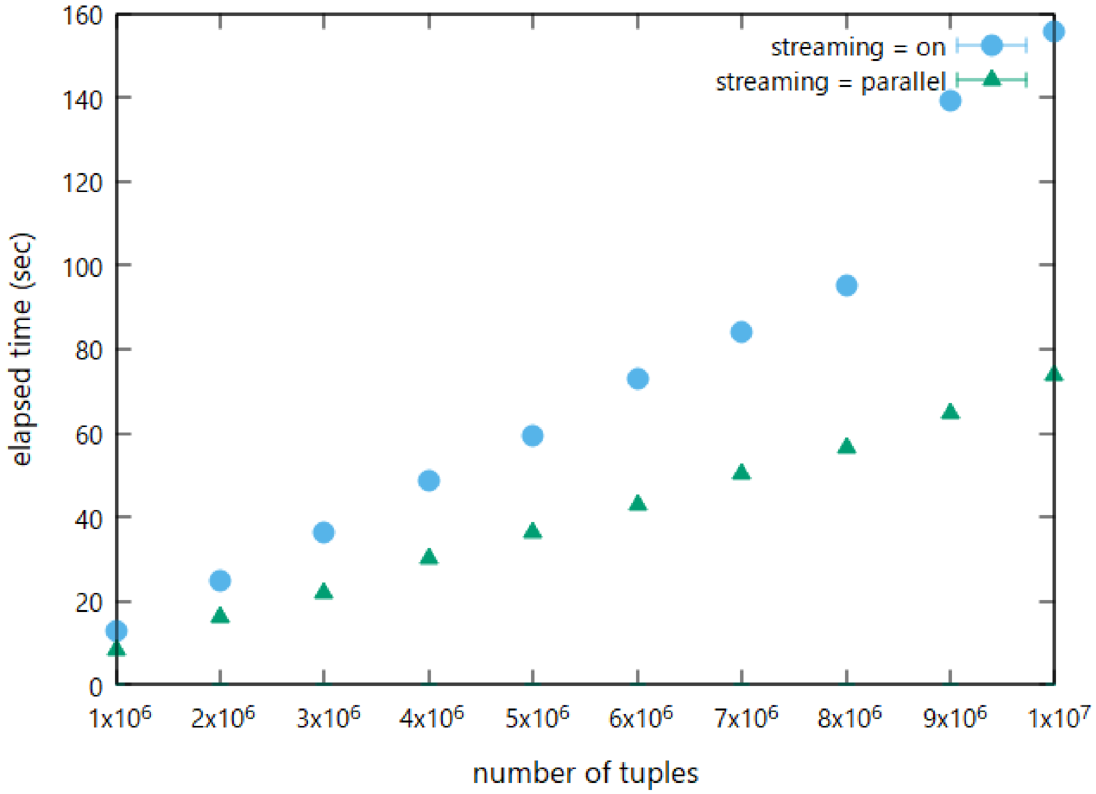
```
shared_buffers = 100GB
Checkpoint_timeout = 30min
max_wal_size = 20GB
min_wal_size = 10GB
autovacuum = off

CREATE TABLE large_test (
      id INTEGER PRIMARY KEY,
      num1 BIGINT,
      num2 DOUBLE PRESICION,
      num3 DOUBLE PRESICION
);
```

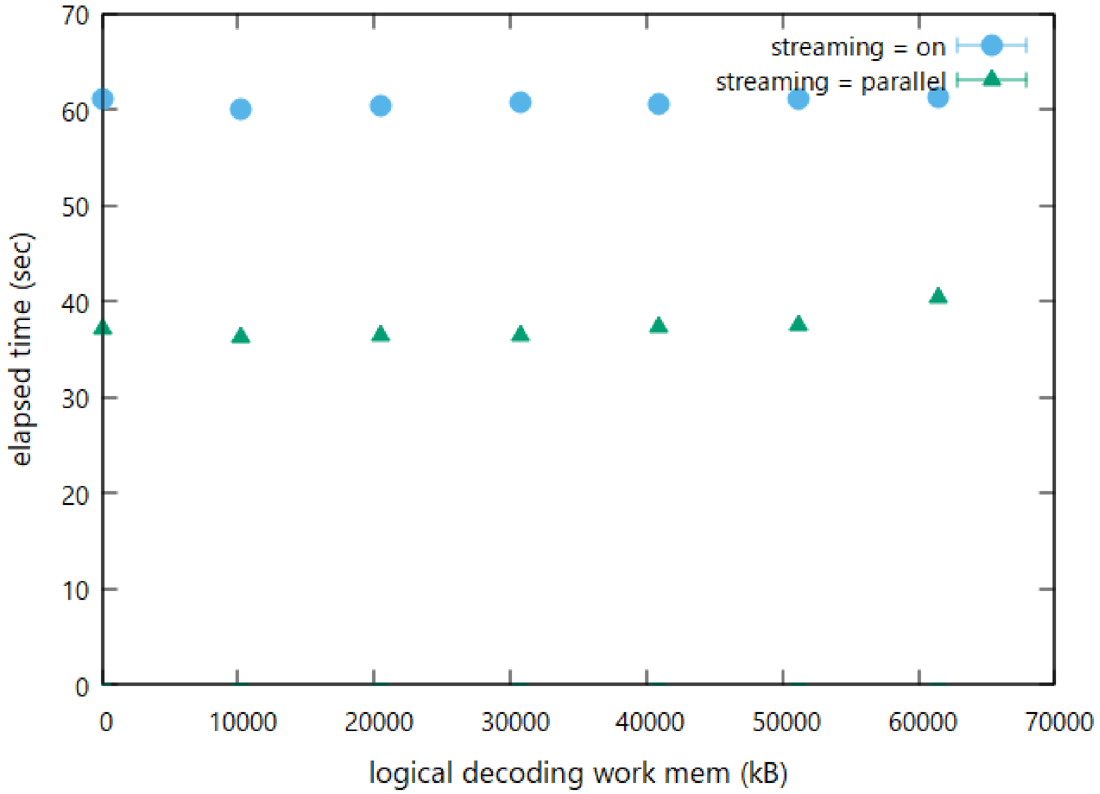
Executed SQL:

```
¥timing
INSERT INTO large_test (id, num1, num2, num3)
       SELECT i, round(random()*10), random(), random()*142
       FROM generate_series(1, 5000000) s(i);
```

Result | # of tuples vs. time



Result | decoding buffer size vs. time



Future development

FUJITSU

- We aim to extend parallel apply to normal cases.
- The basic idea is **to launch parallel apply workers whenever the subscriber side receives new transactions.**
- Some mechanism can be re-used.
- Latency improvements may not be as significant as in the current case.

Summary

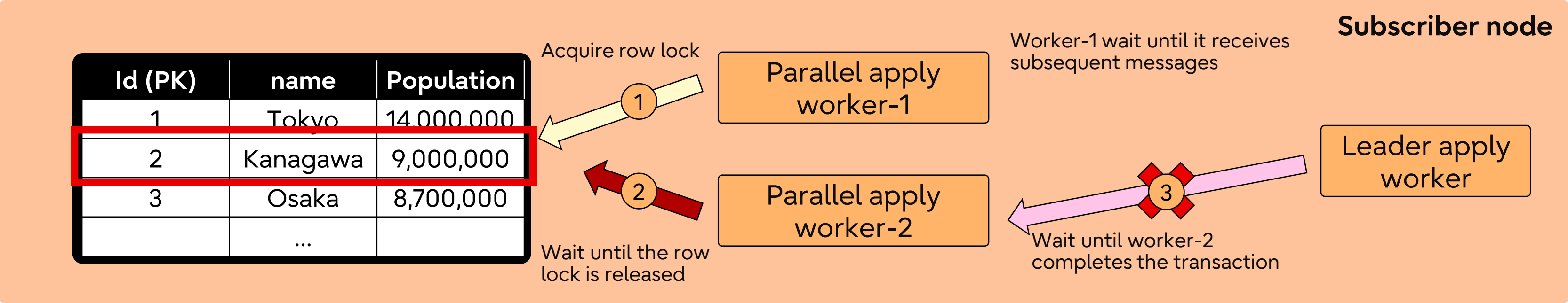
- Logical replication is a powerful feature that continues to evolve.
- One issue with logical replication has been handling large transactions.
- Initially, the publisher node occupied considerable memory.
- Since PostgreSQL 13, this has become controllable.
- Since PostgreSQL 14, the publisher could stream in-progress transactions.
- Since PostgreSQL 16, such transactions can be applied more quickly.
- Our next goal is to extend parallel apply to normal cases.

- If you have any questions and suggestions, please contact me:
- kuroda.hayato@fujitsu.com

Thank you

Issue #2: Deadlock between the leader and parallel apply workers

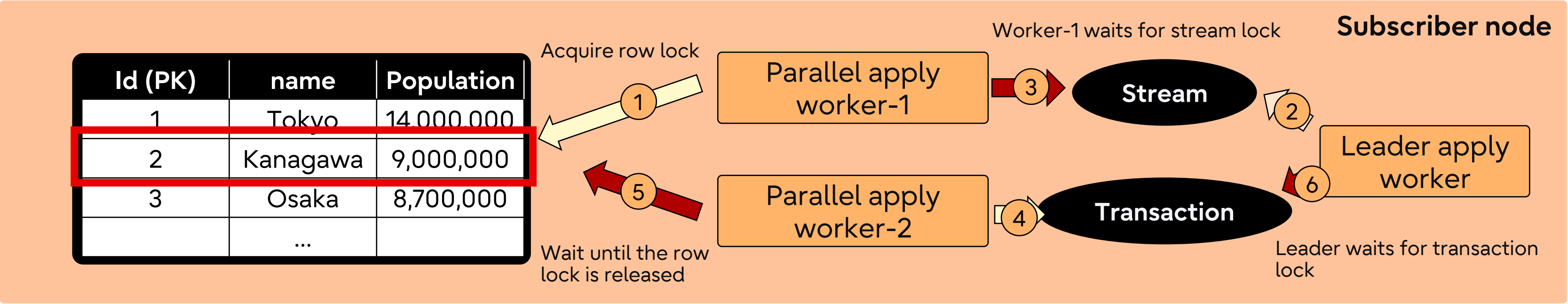
- Consider that the TX-1 and TX-2 are executed by two parallel apply workers (PA-1, PA-2)
- PA-2 is waiting for PA-1 to complete its transaction while PA-1 is waiting for subsequent input from LA.
- Also, LA is waiting for PA-2 to complete its transaction in order to preserve the commit order.



Id (PK)	name	Population
1	Tokyo	14,000,000
2	Kanagawa	9,000,000
3	Osaka	8,700,000
	...	

Issue #2: Deadlock between the leader and parallel apply workers

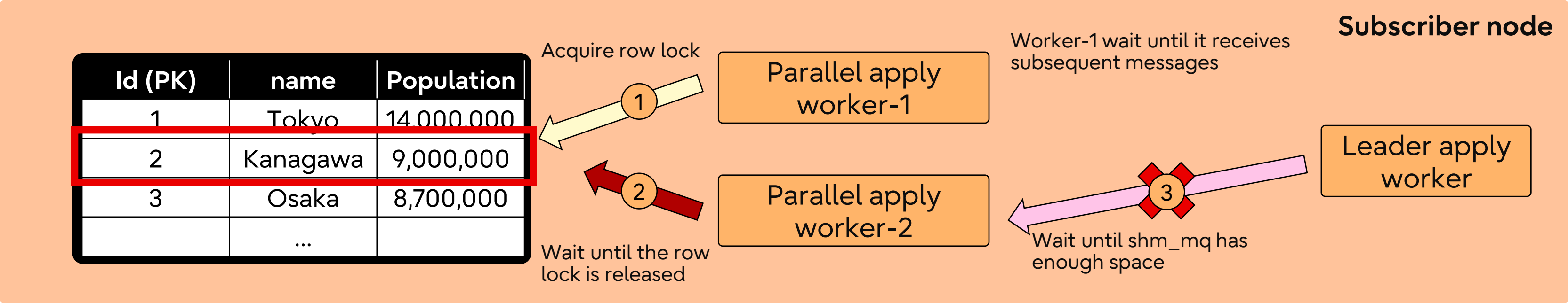
- To resolve this, another session-level lock (transaction lock) is introduced
- The Lock is identified by the subid and the transaction id
- The LA acquire the transaction lock at the end of transactions, and release immediately
- PAs acquire the transaction lock before applying the first message of the transaction, and release at the end of it



Issue#3: Deadlock when the shared message queue is full

FUJITSU

- Consider that the TX-1 and TX-2 are executed by two parallel apply workers (PA-1, PA-2)
- PA-2 is waiting for PA-1 to complete its transaction while PA-1 is waiting for subsequent input from LA.
- If the shared message queue between PA-2 and LA becomes full, LA waits until the queue has enough space, but PA-2 cannot consume messages



Id (PK)	name	Population
1	Tokyo	14,000,000
2	Kanagawa	9,000,000
3	Osaka	8,700,000
	…	

Issue#3: Deadlock when the shared message queue is full

- To resolve this, the wait for enqueuing has a timeout
- If the timeout exceeds, the LA serialize all the pending messages to a file and start to wait committing
- When PA-2 detects the file, apply spooled changes
- In this example, we can regard the case same as issues#2