Описание алгоритма работы Parallel Apply режима применения записей логической репликации.
Данное нововведение было добавлено в 16 версии PostgreSQL.

Введение

Конфигурация: 2 сервера PostgreSQL, один публикатор, второй подписчик, одна публикация на публикаторе и одна соответствующяя ей подписка на подписчике, публикация на несколько одинаковых по структуре таблиц.

В контексте "публикатор-подписчик" (логическая репликация), далее вся речь будет вестись про узел подписчик, т.к. узел публикатор нас не очень интересует.

Процесс "logical replication apply worker for subscription XXX", который существует на узле подписчике при любых видах логической репликации, далее будем называть Leader Apply (LA) или Main Apply worker. А процессы, которые создаются по необходимости и являются нововведением Parallel Apply, будем называть Parallel Apply worker'ами (PA)
Эти новые процессы (PA) являются logical_replication_worker'ами, а также bgworker'ами, т.е. есть 3 параметра, которые влияют на их максимально возможное количество:

- max_parallel_apply_workers_per_subscription (назначение параметра понятно из названия, параметр был добавлен вместе с Parallel Apply)
- max_logical_replication_workers
- max_worker_processes

В директории /src/backend/replication/logical, в файле worker.c лежит код в основном для LA и общий для LA и PA, в файле applyparallelworker.c лежит код для PA и код для взаимодействия LA с PA.

Начало работы

LA worker крутится в цикле и читает сообщения.

worker.c / LogicalRepApplyLoop

```
len = walrcv_receive(LogRepWorkerWalRcvConn, &buf, &fd);
```

Т.к. Parallel Apply это подвид Streaming репликации, первым сообщением будет "STREAM START"

LA переходит в apply_dispatch, а потом обработчик сообщения STREAM START - apply_handle_stream_start

В обработчике вытаскиваем XID транзакции и first_segment.
first_segment - это boolean, указывающий на то, является ли для данной транзакции (XID) стрим первым (т.к. при streaming режиме одна транзакция может поделиться на несколько стримов).

Если это first_segment, т.е., по сути, начало транзакции, переходим в функцию pa_allocate_worker.

worker.c / apply_handle_stream_start

```
if (first_segment) {
        pa_allocate_worker(stream_xid);
}
```

pa_allocate_worker(stream_xid) - функция, выполняющая несколько вещей:

1. Проверяет, что Parallel Apply может быть произведен (вызов pa_can_start). Самый банальный случай, когда он невозможен - подписка не в режиме streaming = parallel.
2. Пытается вытащить из пула PA worker'ов (ParallelApplyWorkerPool) свободного, если такого нет - создать нового.
3. В хэш таблицу (ParallelApplyTxnHash) по соответствующему XID'у вносит информацию о воркере, который был получен на втором шаге. Если хэш таблицы не существует, создает ее и так же вносит информацию.

Далее в apply_handle_stream_start происходит вызов get_transaction_apply_action. Это очень важная функция в алгоритме, которая возвращает значение, определяющее, каким образом нужно применять полученное сообщение. Существует 4 варианта (enum):

- TRANS_LEADER_APPLY - этот вариант можно получить только в нестриминговых транзакциях. Это значит, что мы в LA процессе и мы же эту транзакцию и принимаем либо прямым образом, либо читая из временного файла и применяя.
- TRANS_LEADER_SERIALIZE - мы в LA процессе, сериализуем изменение в файл
- TRANS_LEADER_SEND_TO_PARALLEL - мы в LA процессе, изменение надо отправить на PA worker'а
- TRANS_LEADER_PARTIAL_SERIALIZE - мы в LA процессе, существует PA worker, которому мы до этого отправляли изменения по соответствующему XID, но из-за

таймаута сейчас отправлять невозможно, поэтому записываем его в файл (т.е. сериализуем), потом PA worker будет читать изменения из этого файла.
- TRANS_PARALLEL_APPLY - мы в PA процессе, применяем полученное изменение прямым образом

В нашем случае мы получим TRANS_LEADER_SEND_TO_PARALLEL, т.к. в хэшмапе мы нашли PA worker'a, соответствующего данному XID.

В свою очередь, PA worker так же крутится в цикле и читает сообщения, только уже из shared memory. В этот shared memory ему нужные данные положил LA.

applyparallelworker.c / LogicalParallelApplyLoop

```
shmq_res = shm_mq_receive(mqh, &len, &data, true);
```

PA так же переходит в apply_dispatch, так же читает STREAM START, так же переходит в apply_handle_stream_start, однако у него уже get_transaction_apply_action будет TRANS_PARALLEL_APPLY, и, соответственно, логику он будет выполнять другую.

Середина

В процессе репликации LA так же получает изменения и передает их нужным PA, либо применяет сам, если свободных и / или подходящих PA worker'ов нет.

Допустим, LA получил "INSERT".

worker.c / apply_handle_insert

```
/*
 * Quick return if we are skipping data modification changes or handling
 * streamed transactions.
 */
if (is_skipping_changes() ||
        handle_streamed_transaction(LOGICAL_REP_MSG_INSERT, s))
        return;
```

handle_streamed_transactions содержит некоторый общий код для подобных сообщений. Эта функция вызывается в 6 случаях:

- apply_handle_relation
- apply_handle_type
- apply_handle_insert

- apply_handle_update
- apply_handle_delete
- apply_handle_truncate

В случае TRANS_LEADER_SEND_TO_PARALLEL LA отправит изменения на PA, а в случае TRANS_PARALLEL_APPLY проинкрементируется счетчик parallel_stream_nchanges (кажется, он нужен только для логов) и будет выполнен далее handle_stream_insert, как если бы он выполнялся у обычного LA, т.е. просто применение.

Коммит транзакции

LA получает сообщение "STREAM COMMIT", отсылает его на нужного PA worker'a и блокируется, ожидая, пока соответствующий PA worker завершит свое применение.

worker.c / apply_handle_stream_commit

```
case TRANS_LEADER_SEND_TO_PARALLEL:
        Assert(winfo);

        if (pa_send_data(winfo, s->len, s->data))
        {
                /* Finish processing the streaming transaction. */
                pa_xact_finish(winfo, commit_data.end_lsn);
                break;
        }
```

worker.c / pa_xact_finish

```
/*
 * Wait for that worker to finish. This is necessary to maintain commit
 * order which avoids failures due to transaction dependencies and
 * deadlocks.
 */
pa_wait_for_xact_finish(winfo);
```