P2D2 – Praha 2022

MigOps

Asynchronous queries with PostgreSQL

We are going to cover

- The different way to execute queries asynchronously in PostgreSQL
  - Client side
  - Server side
  - Autonomous vs distributed transactions
  - Scheduling

1.

Introducing

Gilles DAROLD

CTO at MigOps Inc

Author of Ora2Pg, pgBadger, ….

MigOps Inc

Company specialized in Support and Migration to PostgreSQL

- ▷ Sponsors the development of Ora2Pg, pgBadger and others tools at https://github.com/MigOpsRepos/ and https://github.com/darold/

Contact : https://www.migops.com/contact-us/

Client Side

Asynchronous queries

Client side

Executing queries asynchronously at application side

- Forks
- Queues
- Libpq

Fork

Main application

- ▷ BEGIN
- ▷ Do some transactional work…
- ▷ Fork a process and continue with main in parallel
  - ○ Child executing asynchronously the query
- ▷ COMMIT/ROLLBACK
- ▷ Wait the end of the child process

The task is executed in another session

Autonomous transaction => no rollback

Results or errors from child process must be read from a table or multi-process communication.

Fork with transaction control

Main application

- BEGIN
- Do some transactional work…
- Fork a process and continue with main in parallel
  - PREPARE TRANSACTION ‘foo’
  - Execute the query in parallel
- Wait for child process
- COMMIT/ROLLBACK PREPARED ‘foo’
- COMMIT/ROLLBACK

The task is executed in another transaction controlled from the main process.

Results or errors from child process must be read from a table or multi-process communication.

Queue Management System

Main application

- BEGIN
- Transactional work
- Register the query/task in a queue (events table)
  - A queue consumer will execute the query in background
- Execute some other works
- COMMIT/ROLLBACK

❑ The task is executed by another application, no need to fork

❑ The event registration can also be done server side using triggers

❑ Autonomous transaction => no rollback

❑ No control when the task will be executed

Queue system with transaction control

Main application

- BEGIN
- Transactional work
- Register the query/task in a queue (events table)
  - A queue consumer will execute the query in a prepared transaction
  - Write an event to forward the status of the task
- Execute some other works
- Wait while the tracking event is not received
- COMMIT/ROLLBACK the prepared transaction
- COMMIT/ROLLBACK

Queuing solutions

- pgq
- que
- RabbitMQ
- Kafka, ...

Principle:

- □ Event table where the tasks to execute are stored
- □ The application register the event to be executed
- □ The events are consumed by the queuing system
- □ FIFO but some handle task priority and chaining
- □ Queuing is generally based on autonomous transaction
- □ Event tracking for distributed transactions

Libpq

- PostgreSQL Client Library for application
  - Provide the API to
    - Connect to a database
    - Execute SQL queries
    - Get results
    - And more

- Most programming languages drivers are wrappers on libpq

- Query execution modes
  - Synchronous
  - Asynchronous
  - Pipelined (>= PG14)

Synchronous command processing

- PQexec
  - Waits for the command to be completed.
  - The application is suspended while it waits for the result.
  - Always collects and buffers the command's entire results.
  - Can return only one PGresult structure
    - with multiple SQL commands, all but the last PGresult are discarded

Libpq, Synchronous example

```
res = PQexec(conn, "SELECT * FROM employees"); /* waits for the query to complete */
if (PQresultStatus(res) != PGRES_TUPLES_OK)
        /* error report … */

/* next, process the rows */
nFields = PQnfields(res);
for (i = 0; i < PQntuples(res); i++) {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
}
PQclear(res);
```

Asynchronous command processing

- PQsendQuery
  - Submits a command to the server without waiting for result.

- PQgetResult
  - Waits for the next result from a prior PQsendQuery.
  - Must be called repeatedly until it returns a null pointer.
  - All results buffered in PGresult struct.
  - For a result with large number of rows
    - Use PQsetSingleRowMode

- PQsendQuery cannot be called again until PQgetResult has returned a null pointer.
- With multiple SQL commands, the results of each commands are available.

Libpq, Asynchronous example

```
res = PQSendQuery(conn, "SELECT * FROM employees"); /* returns immediately without waiting for command completion */
if (PQresultStatus(res) != PGRES_TUPLES_OK)
        /* error report … */

/* next, process the rows */
while(( res = PQgetResult(conn)) != NULL) {
    if (PQresultStatus (res)  == PGRES_TUPLES_OK) {
        nFields = PQnfields(res);
        for (i = 0; i < PQntuples(res); i++) {
            for (j = 0; j < nFields; j++)
                          printf("%-15s", PQgetvalue(res, i, j));
        }
    }
}
```

Libpq, Asynchronous

Calling PQgetResult still cause the client to block until the server completes the SQL command.

Some more useful functions:

- PQconsumeInput
  - If input is available from the server, consume it.
- PQisBusy
  - whether you can call PQgetResult without blocking

Libpq, Asynchronous example

```
res = PQSendQuery(conn, "SELECT long_running_query()");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
        /* error report … */
if (PQconsumeInput(conn)) /* search for input */
{
    /* Does calling PGgetResult could be blocking ? */
    While ((PQisBusy(conn) == 1)
    {
            /* In this case do something else and look for next input … */
            PQconsumeInput(conn)
    }
    /* retrieve results */
    res = PqgetResult(conn);
}
```

Libpq, Pipeline mode

Interesting to send multiple queries executed in parallel by the backend, then read results from all queries.

- PQenterPipelineMode
  - Switch the connection to pipeline mode.
- The server executes statements, and returns results, in the order the client sends them.
- The server will begin to execute the commands in the pipeline immediately, not waiting for the end of the pipeline.
- Results are buffered on the server side.
- The server flushes the buffer when a synchronization point is called with PQpipelineSync or a call to PQsendFlushRequest.

Libpq, Pipeline mode example

```
if (!PQenterPipelineMode(conn))   /* error report ... */
/* send a first query */
res = PQSendQuery(conn, "INSERT ... RETURNING id");
/* Instruct the backend that it can start to send the result */
if (PQsendFlushRequest(conn) == 0)  /* error report ... */
/* send a new query */
res = PQSendQuery(conn, "INSERT ... RETURNING id");
/* flush the statements and wait for the results */
if (PQpipelineSync(conn) == 0) /* error report ... */

while ((res = PQgetResult(conn) != NULL) /* retrieve results from first query */
while ((res = PQgetResult(conn) != NULL) /* retrieve results from the second query */

PQexitPipelineMode(conn); /* exit pipeline mode */
```

Libpq, Pipeline mode

Client side since PG14 => but works with old server version

Available in several programming languages:

- Ruby
- Python
- Java
- ...

Server Side

Asynchronous tasks

Server side

Extensions allowing asynchronous execution

- pg_background
- dblink
- ...

pg_background

- pg_background_launch(query) -> pid
  - Launch a background worker to execute the query
  - Loopback connection (same host and same database)
  - Main use: autonomous transaction

- pg_background_detach(pid)
  - Detach the background process from the running session
  - No wait for the user to read the results.

- pg_background_result(pid)
  - Read the result of the command executed by the background process.

pg_background / Synchronous call

```
db=# CREATE EXTENSION pg_background;
CREATE EXTENSION

/* Execute the command in a background process and wait for the result */
db=# SELECT pg_background_result( pg_background_launch('SELECT count(*) FROM employees') ) as (result bigint);
 result
--------
    107

/* Equivalent to the following except that it is executed in another session */
db=# SELECT count(*) from employees;
 count
-------
   107
```

pg_background / Asynchronous call

```
db=# SELECT pg_background_launch('SELECT count(*) FROM employees');
 pg_background_launch
----------------------
            37713
/* Do something else */
db=# SELECT count(*) from employees;
 count
-------
   107
/* Get the result */
db=# SELECT * FROM pg_background_result(37713) as (result bigint);
 result
--------
            107
```

pg_background / No results

- ▷ Fork to execute the command and leave without looking back

```
db=# SELECT pg_background_launch(‘SELECT ');
 pg_background_launch
----------------------
            37791

db=# SELECT * FROM pg_background_detach(37791);
 pg_background_detach
----------------------


db=# SELECT * FROM pg_background_result(37791) as (result bigint);
ERROR:  PID 37791 is not attached to this session
```

dblink

- Execute a command in a remote database
  - Same or different host / database (pg_hba.conf)
  - Autonomous transaction
  - Returns the rows produced by the query

dblink / synchronous call

- dblink(connstr, query [, bool fail_on_error]) -> setof record

```
db=# CREATE EXTENSION dblink;
CREATE EXTENSION

db=# SELECT * FROM dblink('dbname=hr', 'SELECT count(*) FROM employees', true) AS t1(cnt bigint);
 cnt
-----
 107
(1 row)
```

dblink / asynchronous call

- dblink_send_query(connname, query) -> int
  - Execute asynchronously the query on remote connection
  - Returns 1 on success, 0 otherwise

- dblink_get_result(connname [, bool fail_on_error]) -> setof record
  - Collects the results of an asynchronous query
  - Wait when not already completed

- Use dblink_connect(connname, connstr) to open a named connection

dblink / asynchronous call

```
db=# SELECT dblink_connect(‘conn1’, 'dbname=hr');
 dblink_connect
-----------------
 OK
db=# SELECT dblink_send_query('conn1', 'SELECT count(*) FROM huge_table);
 dblink_send_query
--------------------
                    1
[... do some work ...]
db=# SELECT * FROM dblink_get_result('conn1') AS t1(f1 int);
       f1
—-------------
 100000000
```

Scheduling

Asynchronous tasks

Schedulers

- pg_cron
  - The venerable cron-like scheduler for PostgreSQL
- pg_timetable
  - Cron based scheduler with advanced features
- pg_dbms_job
  - Manage scheduled jobs from a job queue
  - Execute immediately jobs asynchronously
- pgAgent, pgBucket,...

- All are interesting for planned tasks
- Short planned date to emulate asynchronous execution
  - Schedulers are not done for that unlike Queue system
  - Except pg_dbms_job

pg_cron

- Simple cron-based job scheduler for PostgreSQL
  - https://github.com/citusdata/pg_cron
  - PostgreSQL extension written in C
  - Background worker started/stopped with PostgreSQL
    - shared_preload_libraries = 'pg_cron'
  - Automatically starts when a standby server is promoted
  - Scheduler granularity: minute

pg_cron, example

```
/* Delete old data on Saturday at 3:30am (GMT) */
SELECT cron.schedule('30 3 * * 6',
                                   $$DELETE FROM events WHERE event_time < now() - interval '1 week'$$);
 schedule
----------
      42
/* Run a function asap and remove it */
SELECT cron.schedule('run-vacuum', '* * * * *', 'CALL my_proc()');
 schedule
----------
      43

SELECT cron.unschedule('run-vacuum'); /* remove the task */
```

pg_timetable

- Advanced cron-based job scheduler for PostgreSQL
    - https://github.com/cybertec-postgresql/pg_timetable
    - Standalone process, written in GO
    - Some useful advanced feature:
        - Chained tasks,
        - Executes SQL, built-in or executable command
        - Database driven configuration
        - Parameters can be passed to tasks
        - Scheduler granularity: minute
        - Etc

- No immediate asynchronous task execution.

pg_timetable, example

```
-- Run public.my_func() at 00:05 every day in August:
SELECT timetable.add_job('execute-func', '5 0 * 8 *', 'SELECT public.my_func()');

-- Run a function asap and remove it
SELECT timetable.add_job(
                        job_name =>’run-vacuum',
                        job_schedule => '* * * * *',
                        job_command => ‘CALL my_proc()',
                        job_self_destruct => TRUE);
```

pg_dbms_job

- Schedules and manages jobs in a job queue
    - https://github.com/MigOpsRepos/pg_dbms_job
    - Standalone process, written in Perl
    - Scheduler based on a Queue system
        - Immediate asynchronous query execution
        - Executes SQL statements, PLPGSQL procedures or code
        - Database driven
        - Scheduler granularity: second

pg_dbms_job

- A job definition consist on:
  - a code to execute,
  - the next date of execution
    - NULL/CURRENT_TIMESTAMP for immediate execution
  - and how often the job is to be run.
    - NULL for a single execution
- A job runs a SQL command, plpgsql code or an existing stored procedure.
- Job_queue_interval:
  - poll interval of the jobs queue. Default 5 seconds.
- Job_queue_processes:
  - Maximum number of job processed at the same time. Default 1000.

pg_dbms_job, immediate execution

- Job submitted without execution date
- Stored in a queue (FIFO) table dbms_job.all_async_jobs
- Jobs in that queue at start of the scheduler are executed immediately

```
SELECT dbms_job.submit(
        -- what to execute immediately
         'BEGIN
                CALL proc1();
        END;'
    ) INTO jobid;
```

pg_dbms_job, really immediate?

- Job_queue_interval:
  - poll interval of the jobs queue. Default 5 seconds.

- Hard to trust an immediate execution with such polling interval!

  - dbms_job.submit() use NOTIFY to instruct the daemon pg_dbms_job that a new job has been registered.
  - LISTEN is called every 100ms by pg_dbms_job.
  - pg_dbms_job look at job definitions every “job_queue_interval” seconds if no notification have been received.

pg_dbms_job, delayed execution

- Job submitted with an execution date
- And if necessary an interval for a repeated execution
- Example of a job that must be executed next coming hour and after that, every 2 hours.

```
SELECT dbms_job.submit(
          'BEGIN CALL my_stored_procedure(); END;',
         date_trunc( 'hour', now() ) + '1 hour'::interval, /* to be executed next starting hour */
         date_trunc( 'second', now() ) + '2 hours'::interval /* every 2 hours */
   ) INTO jobid;
```

Thanks !

Email : gilles@darold.net

Any questions?