Open in app

Christoph Bussler

Sep 3 · 4 min read · Listen

Save

Implementing Queues in PostgreSQL (Part 2: AlloyDB for PostgreSQL)

Some performance numbers when deploying into AlloyDB for PostgreSQL

Recently Google Cloud made AlloyDB for PostgreSQL available for preview. This is a great opportunity to run the queue example from Part 1 in AlloyDB for PostgreSQL to get performance numbers from a brand-new production system.

AlloyDB for PostgreSQL configuration

I decided to use the largest deployment option available in AlloyDB for PostgreSQL currently and setup an AlloyDB for PostgreSQL cluster as follows:

- Cluster specification (no read pools):

```
Version: PostgreSQL 14 compatible
Type: Highly available
```

- Primary instance specification

```
High availability: Highly available (multi-zone)
Machine type: 64 vCPU, 512 GB
```

```
version
-----------------------------------------------------------------
PostgreSQL 14.4 on x86_64-pc-linux-gnu, compiled by Debian clang
version 12.0.1, 64-bit
(1 row)
```

Driver VM specification

The VM that runs the pgbench execution is specified as follows (the largest I was allowed to create — there are larger once available):

```
Machine type: n2-highcpu-8
CPU platform: Intel Cascade Lake
Architecture: x86/64
```

Database, schema and tables

The setup is precisely that of Part 1 without any changes.

Execution: pgbench — operations in isolation

The performance numbers for running the enqueue and the dequeue operation on AlloyDB for PostgreSQL in isolation are:

- Enqueue in isolation (no concurrent dequeue operations):

```
pgbench -n -c 39 -r -T 60 -h 10.0.0.7 -U queuedev -f writer.sql
queue_database
Password:
transaction type: writer.sql
scaling factor: 1
query mode: simple
number of clients: 39
number of threads: 1duration: 60 s
number of transactions actually processed: 1877369
latency average = 1.246 ms
tps = 31288.793094 (including connections establishing)
```

With 39 clients it is possible to enqueue 31K messages per second into the queue, each being a separate transaction.

- Dequeue in isolation (no concurrent enqueue operations):

```
pgbench -n -c 44 -r -T 60 -h 10.0.0.7 -U queuedev -f reader.sql
queue_database
Password:
transaction type: reader.sql
scaling factor: 1
query mode: simple
number of clients: 44
number of threads: 1
duration: 60 s
number of transactions actually processed: 1109711
latency average = 2.379 ms
tps = 18494.124467 (including connections establishing)
tps = 18499.540774 (excluding connections establishing)
statement latencies in milliseconds:
  2.318 DELETE
```

With 44 clients it is possible to dequeue 18K messages per second. Remember, the dequeue is reading the oldest element and delete it, so the dequeue needs to retrieve the message as well as delete it.

Note: the particular number of clients, 39 vs. 44, I determined by executing different pgbench runs and select the most performing.

Execution: pgbench — concurrent operations

As before, when running enqueue and dequeue operations concurrently, the throughput reduces. The following are the performance numbers for a duration of 300 seconds, first the result for the enqueue operations, followed by the result for the dequeue operations.

- Enqueue operations (concurrent with dequeue operations):

```
query mode: simple
number of clients: 39
number of threads: 1
duration: 300 s
number of transactions actually processed: 7716530
latency average = 1.516 ms
tps = 25720.640230 (including connections establishing)
tps = 25722.116233 (excluding connections establishing)
statement latencies in milliseconds:
 1.485 INSERT INTO queue_schema.queue (element_identifier,
time_inserted, payload)
```

- Dequeue operations (concurrent with enqueue operations):

```
pgbench -n -c 50 -r -T 300 -h 10.0.0.7 -U queuedev -f reader.sql
queue_database
Password:
transaction type: reader.sql
scaling factor: 1
query mode: simple
number of clients: 50
number of threads: 1
duration: 300 s
number of transactions actually processed: 4550282
latency average = 3.297 ms
tps = 15167.414547 (including connections establishing)
tps = 15168.353280 (excluding connections establishing)
statement latencies in milliseconds:
 3.270 DELETE
```

In production this situation would not be a desirable one as the dequeue operations cannot keep up to establish a stable queue (one that does not keep growing)— unless of course the amount of messages arriving varies over time so that the dequeue operations can keep up.

Execution — summary

Here the performance numbers in form of a summary:

- Dequeue: 18494 tps (44 clients)

Concurrent operations with 15 clients each

- Enqueue: 25720 tps (39 clients)
- Dequeue: 15167 tps (50 clients)

I want to point out that the above are very rough numbers; no tuning of the database took place, and I did not have access to a larger driver VM for running the pgbench tests. Several driver VMs (instead of just one) might further increase the throughput. Again, the goal was to get a rough ballpark idea, not to establish a fine-tuned benchmark.

Summary

As expected, a production system like AlloyDB for PostgreSQL is significantly more performant than my laptop.

As outlined in Part 1, a relational database, here PostgreSQL, might very well be suitable to support queueing functionality with the benefit of backup/restore, high availability, disaster recovery, etc., aside from the very nice ability to transactionally connect queuing operations like enqueue and dequeue with queue element processing domain logic.

As always, please ping me with feedback or input.