Open in app

Christoph Bussler
Aug 19 · 6 min read · Listen

Save

Implementing Queues in PostgreSQL (Part 1: Design and Measurement)

And some initial performance measurements

Out of curiosity (ha!) I implemented a queue in PostgreSQL with readers (enqueue) and writers (dequeue). I did some basic performance measurement on my laptop with pgbench to get a ballpark idea of how many queue elements can be enqueued and dequeued.

Overview: queue

A queue is a data structure that writers and readers access independently. A queue stores many queue elements, and the term “queue” implies that there is an order to the queue elements. A writer enqueues queue elements into the queue, while a reader dequeues messages from the queue in the desired order. The order can be based on time, identifier or any other property.

Note that if there is more than one client accessing the queue concurrently independent of each other, each client retrieves queue elements independently and from its perspective selects always the oldest queue element first (when time is used as criteria), however, since other clients accessing it concurrently each client only observes a subset of the queue elements.

A strict FIFO queue behavior on the reader side would require coordination. A single reader implements a FIFO queue by default. If more than one reader accesses a queue, the reader have to coordinate amongst themselves, or the queue coordinates reader access.

The queue itself is implemented as a single table with three columns:

- `element_identifier` as a UUID
- `time_inserted` as a current database timestamp
- `payload` as a JSON object of type `JSON`. `JSONB` was not used as the payload will not be accessed by any operation.

The enqueue operation inserting queue elements is implemented as a SQL insert statement. Each insert statement inserts a single queue element.

The dequeue operation is implemented as a SQL delete statement. It finds the oldest queue element based on `time_inserted`, deletes it and returns its column values. This ensures that fetched queue elements are removed from the queue. Each dequeue operation removes a single queue element.

Implementation: table and queries

The following shows the implementation.

Table schema:

```
CREATE TABLE IF NOT EXISTS queue_schema.queue
(
    element_identifier UUID PRIMARY KEY,
    time_inserted      TIMESTAMP,
    payload            JSON
);
```

Index (dequeue is based on `time_inserted`):

```
CREATE INDEX time_inserted_idx
    ON queue_schema.queue (time_inserted ASC);
```

```
INSERT INTO queue_schema.queue (element_identifier, time_inserted,
payload)
VALUES (gen_random_uuid(), current_timestamp, '{
  "type": "performance test",
  "topic": "fifo queue read and write, no domain logic involved"
}');
```

Dequeue operation (delete based on select for update skip locked):

```
DELETE
FROM queue_schema.queue pse
WHERE pse.element_identifier =
      (SELECT pse_inner.element_identifier
       FROM queue_schema.queue pse_inner
       ORDER BY pse_inner.time_inserted ASC
           FOR UPDATE SKIP LOCKED
       LIMIT 1)
RETURNING pse.element_identifier, pse.time_inserted, pse.payload;
```

The use of `select for update skip locked` ensures that concurrent clients can access the table concurrently and do not block each other on existing locks.

The clause `limit 1` ensures that only one queue element is dequeued at a time for each execution.

Machine and PostgreSQL database

The following pgbench runs are executed on the following machine:

```
OS Name Microsoft Windows 11 Pro

Version 10.0.22000 Build 22000

Processor Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz, 1992 Mhz, 4
Core(s), 8 Logical Processor(s)
```

The database is a standard installation without configuration changes:

```
select version()

PostgreSQL 13.4, compiled by Visual C++ build 1914, 64-bit
```

Execution: pgbench — operations in isolation

In order to have a baseline, the following shows isolated executions of a single enqueue as well as a single dequeue operation. The observations are based on a run of 60 seconds with 15 clients.

- Enqueue in isolation (no concurrent dequeue operations):

```
pgbench -n -c 15 -r -T 60 -h 127.0.0.1 -U queuedev -f writer.sql
queue_database
Password:
transaction type: writer.sql
scaling factor: 1
query mode: simple
number of clients: 15
number of threads: 1
duration: 60 s
number of transactions actually processed: 1144519
latency average = 0.786 ms
tps = 19074.572599 (including connections establishing)
tps = 19086.568392 (excluding connections establishing)
statement latencies in milliseconds:
         0.608  INSERT INTO queue_schema.queue (element_identifier,
time_inserted, payload)
```

- Dequeue in isolation (no concurrent enqueue operations):

```
pgbench -n -c 15 -r -T 60 -h 127.0.0.1 -U queuedev -f reader.sql
queue_database
```

```
number of threads: 1
duration: 60 s
number of transactions actually processed: 288852
latency average = 3.117 ms
tps = 4812.968979 (including connections establishing)
tps = 4816.042062 (excluding connections establishing)
statement latencies in milliseconds:
         3.051  DELETE
```

The dequeue operation is slower since it has to select the oldest queue element, retrieve and return its values, and delete the queue element. This means that several dequeue operations will have to be run concurrently in order to maintain a steady state ideally decreasing the number of queue elements in the queue over time so that no backlog of queue elements builds up.

Execution: pgbench — concurrent operations

Preliminaries

When setting up concurrent operations with the above pgbench specifications it turns out that the dequeue operations cannot keep up with the enqueue operations. In a production system this would require additional fine tuning in order to get into a stable state where the queue length might be oscillating, but never constantly increase over time.

If there are more enqueue than dequeue operations, one pragmatic approach could be to throttle the enqueue and create pressure so that the queue clients writing into the queue are slowed down. A temporary increase of queue elements is fine to observe a spike but a continuously increasing number of queue elements would not be a good situation.

If the queue reaches an empty state, one could consider a back-off on dequeue if the dequeue operation encounters an empty queue reducing unnecessary access. For example, on an empty queue, wait for some amount of time before executing the next dequeue.

A further improvement could be to implement a variable set of dequeue operations that

Depending on the domain logic, batch enqueues or batch dequeues might be an option so that the execution of one of the operations processes several queue elements in a single transaction.

Observations

The following observations were run longer than those before. The enqueue ran for 300 seconds, and the dequeue ran for 300 seconds as well (I started the enqueue first, then the dequeue operations). Starting tests manually is a crude approach, no question, but it gives me a basic baseline upon which to improve on.

- Enqueue operations (concurrent with dequeue operations):

```
pgbench -n -c 15 -r -T 300 -h 127.0.0.1 -U queuedev -f writer.sql
queue_database
Password:
transaction type: writer.sql
scaling factor: 1
query mode: simple
number of clients: 15
number of threads: 1
duration: 300 s
number of transactions actually processed: 1990767
latency average = 2.261 ms
tps = 6634.770044 (including connections establishing)
tps = 6635.616311 (excluding connections establishing)
statement latencies in milliseconds:
         2.148  INSERT INTO queue_schema.queue (element_identifier,
time_inserted, payload)
```

- Dequeue operations (concurrent with enqueue operations):

```
pgbench -n -c 15 -r -T 300 -h 127.0.0.1 -U queuedev -f reader.sql
queue_database
Password:
transaction type: reader.sql
scaling factor: 1
```

```
latency average = 5.457 ms
tps = 2748.761692 (including connections establishing)
tps = 2749.536880 (excluding connections establishing)
statement latencies in milliseconds:
         5.395  DELETE
```

In this scenario, the queue would be ever increasing and lead to a situation where at some point the system is running out of space. It would have to be dealt (as discussed earlier) with so that the queue length will decrease over time.

Execution — summary

Isolated operations with 15 clients each

- Enqueue: 19074 tps
- Dequeue: 4812 tps

Concurrent operations with 15 clients each

- Enqueue: 6634 tps
- Dequeue: 2748 tps

Summary

This is a very rough ballpark evaluation on the possible throughput of a queue in PostgreSQL based on pgbench. When implementing enqueue and dequeue in client code, several improvements are possible that might increase the the throughput beyond what is shown here.

Of course, the above numbers are based on my laptop and not a PostgreSQL installation on a production server. It'd be interesting to see how different the throughput numbers would be on a production machine.

The benefit of queues in the database are that the queue operations can be transactionally

Furthermore, queues in the database can be consistently backed up with any domain logic state in the same database, making recovery or failover consistent.

As always, please ping me with feedback or input.