PgQ
Generic high-performance queue
for PostgreSQL
Agenda

- Introduction to queuing
- Problems with standard SQL
- Solution by exporting MVCC info
- PgQ architecture and API
- Use-cases
- Future
Queue properties

- Data is created during ordinary transactions
- But we want to process it later
- After it is processed, its useless

Producer:
change_password
-> password event

User events

Consumer:
mailer
Queue goals

- High-throughput
  - No locking during writing / reading
  - Parallel writes
  - Batched reads
- Low-latency
  - Data available in reasonably short time
- Robust
  - Returns all events
  - Repeatable reads
Implementing a queue with standard SQL
Standard SQL - row-by-row

- Reading process:
  - Select first unprocessed row
  - Update it as in-progress
  - Later update it as done or delete.
- High-throughput – NO
- Low-latency – YES
- Robust - YES
Standard SQL – SELECT with LIMIT

- Reading process:
  - Select several unprocessed rows with LIMIT
  - Later delete all of them.
- High-throughput – YES
- Low-latency – YES
- Robust - NO
Standard SQL – rotated tables

- Reading process:
  - Rename current event table
  - Create new empty event table
  - Read renamed table
- High-throughput – YES
- Low-latency – NO
- Robust - YES
Standard SQL – group by nr / date

- Reading process:
  - Request block of events for reading
  - Read them
  - Tag the block of events as done
- High-throughput – YES
- Low-latency – YES
- Robust - NO
No good way to implement queue with standard SQL
Postgres-specific solution, ideas

- Vadim Mikheev (rserv)
  - We can export internal Postgres visibility info (transaction id / snapshot).

- Jan Wieck (Slony-I)
  - If we have 2 snapshots, we can query events that happened between them.
  - “Agreeable order” - order taken from sequence in AFTER trigger
Postgres-specific solution, PgQ improvements

- Optimized querying that tolerates long transactions
- Optimized rotation, the time when query is ran on both old and new table is minimal (long tx problem)
- 64-bit stable external transaction Ids
- Simple architecture – pull-only readers
- Queue component is generic
Postgres-specific solution, MVCC basics

- Transaction IDs (txid) are assigned sequentially
- Transactions can be open variable amount of time, their operations should be invisible for that time
- Snapshot represents point in time – it divides txids into visible ones and invisible ones
Postgres-specific solution, details

- Event log table:
  - (ev_txid, ev_data)
- Tick table where snapshots are stored
  - (tick_id, tick_snapshot)
- Result:
  - High-performance – YES
  - Low-latency – YES
  - Robust - YES
Postgres-specific solution – Snapshot basics

- Xmin – lowest transaction ID in progress
- Xmax – first unassigned transaction ID
- Xip – list of transaction IDs in progress
- \[
\text{txid\_visible\_in\_snapshot}(\text{txid}, \text{snap}) = \]
  \[
  \text{txid} < \text{snap.xmin} \text{ OR ( txid } < \text{snap.xmax AND txid NOT IN (snap.xip) })
  \]
Postgres-specific solution – Core API

- Current transaction details:
  - `txid_current()`: int8
  - `txid_current_snapshot()`: txid_snapshot

- Snapshot components:
  - `txid_snapshot_xmin(snap)`: int8
  - `txid_snapshot_xmax(snap)`: int8
  - `txid_snapshot_xip(snap)`: SETOF int8

- Visibility check:
  - `txid_visible_in_snapshot(txid, snap)`: bool
Query between snapshots
Query between snapshots – Simple version

- Snapshot 1 – xmin1, xmax2, xip1
- Snapshot 2 – xmin2, xmax2, xip2
- \texttt{SELECT * FROM queue}
  \begin{align*}
  \text{WHERE ev\_txid \ BETWEEN \ xmin1 \ AND \ xmax2} \\
  \text{AND NOT \ is\_visible(ev\_txid, \ snap1)} \\
  \text{AND is\_visible(ev\_txid, \ snap2)}
  \end{align*}
- Index scan between xmin1 and xmax2
Query between snapshots – optimized version

- Query must be done in 2 parts – range scan and list of explicit ids
- SELECT * FROM queue
  WHERE (ev_txid IN (xip1) OR (ev_txid BETWEEN xmax1 AND xmax2))
  AND NOT is_visible(ev_txid, snap1)
  AND is_visible(ev_txid, snap2)
Query between snapshots – more optimizations

- More optimizations
  - Pick txids that were actually committed
  - Decrease explicit list by accumulating nearby ones into range scan
- Final notes:
  - The values must be substituted literally into final query, Postgres is not able to plan parametrized query.
  - PgQ itself uses UNION ALL instead OR. But OR seems to work at least on 8,3.
Query between snapshots – helper function

- All complexity can be put into helper function
  - \[
  \text{SELECT range_start, range_end, explicit_list}
  \text{FROM txid_query_helper(snap1, snap2)};
  \]
- This results in query:
  - \[
  \text{SELECT * FROM queue}
  \text{WHERE ev_txid IN (explicit_list) OR}
  \text{( ev_txid BETWEEN range_start AND range_end}
  \text{AND NOT is_visible(ev_txid, snap1)}
  \text{AND is_visible(ev_txid, snap2) )}
  \]
There is PgQ.

Take a deep breath.
**PgQ architecture**

- **Ticker** (pgqadm.py -d config.ini ticker)
  - Inserts ticks – per-queue snapshots
  - Vacuum tables
  - Rotates tables
  - Re-inserts retry events
- **Event Producers**
  - pgq.insert_event()
  - pgq.sqltriga() / pgq.logutriga()
- **Event Consumers**
  - Need to register
  - Poll for batches
PgQ event structure

- CREATE TABLE pgq.event (  
  ev_id int8 NOT NULL,  
  ev_txid int8 NOT NULL DEFAULT txid_current(),  
  ev_time timestamptz NOT NULL DEFAULT now(),  
  -- rest are user fields --  
  ev_type text, -- what to expect from ev_data  
  ev_data text, -- main data, urlenc, xml, json  
  ev_extra1 text, -- metadata  
  ev_extra2 text, -- metadata  
  ev_extra3 text, -- metadata  
  ev_extra4 text -- metadata  
);  
CREATE INDEX txid_idx ON pgq.event (ev_txid);
**PgQ ticker**

- Reads event id sequence for each queue.
- If new events have appeared, then inserts tick if:
  - Configurable amount of events have appeared
    \[\text{ticker\_max\_count} \ (500)\]
  - Configurable amount of time has passed from last tick
    \[\text{ticker\_max\_lag} \ (3 \ sec)\]
- If no events in the queue, creates tick if some time has passed.
  - \[\text{ticker\_idle\_period} \ (60 \ sec)\]
- Configuring from command line:
  - `pgqadm.py ticker.ini config my_queue`  
    \[\text{ticker\_max\_count}=100\]
PgQ API: event insertion

• Single event insertion:
  • `pgq.insert_event(queue, ev_type, ev_data): int8`

• Bulk insertion, in single transaction:
  • `pgq.current_event_table(queue): text`

• Inserting with triggers:
  • `pgq.sqltriga(queue, ...) - partial SQL format`
  • `pgq.logutriga(queue, ...) - urlencoded format`
PgQ API: insert complex event with pure SQL

- CREATE TABLE queue.some_event (col1, col2);
  CREATE TRIGGER some_trg
      BEFORE INSERT ON queue.some_event
      FOR EACH ROW EXECUTE PROCEDURE
          pgq.logutriga('dstqueue', 'SKIP');

- Plain insert works:
  - INSERT INTO queue.some_event(col1, col2)
    VALUES ('value1', 'value2');

- Type safety, default values, sequences, constraints!
- Several tables can insert into same queue.
PgQ API: reading events

- Registering
  - `pgq.register_consumer(queue, consumer)`
  - `pgq.unregister_consumer(queue, consumer)`
- Reading
  - `pgq.next_batch(queue, consumer): int8`
  - `pgq.get_batch_events(batch_id): SETOF record`
  - `pgq.finish_batch(batch_id)`
Remote event tracking

- Async operation allows coordinating work between several database.
- Occasionally data itself allows tracking:
  - eg. Delete order.
- If not then explicit tracking is needed.
- pgq_ext module.
- Tracking can happen in multiple databases.
Tracking events

- Per-event overhead
- Need to avoid accumulating
- pgq_ext solution
  - `pgq_ext.is_event_done(consumer, batch_id, ev_id)`
  - `pgq_ext.set_event_done(consumer, batch_id, ev_id)`
- If batch changes, deletes old events
- Eg. email sender, plproxy.
Tracking batches

- Minimal per-event overhead
- Requires that all batch is processed in one TX
  - `pgq_ext.is_batch_done(consumer, batch_id)`
  - `pgq_ext.set_batch_done(consumer, batch_id)`
- Eg. replication, most of the Skytools partitioning script.
Use-case: row counter for count(*) speedup

```
import pgq

class RowCounter(pgq.Consumer):
    def process_batch(self, db, batch_id, ev_list):
        tbl = self.cf.get('table_name'); delta = 0
        for ev in ev_list:
            if   ev.type == 'I' and ev.extra1 == tbl: delta += 1
            elif ev.type == 'D' and ev.extra1 == tbl: delta -= 1
            ev.tag_done()

        q = 'select update_stats(%s, %s)
        db.cursor().execute(q, [tbl, delta])

        RowCounter('row_counter', 'db', sys.argv[1:]).start()

[row_counter]
```

```
db = ...
pqq_queue_name = ...
table_name = ...
job_name = ...
logfile = ...
pidfile = ...
```
import pgq

class QueueMover(pgq.RemoteConsumer):
    def process_remote_batch(self, db, batch_id, ev_list, dst_db):
        # prepare data
        rows = []
        for ev in ev_list:
            rows.append([ev.type, ev.data, ev.time])
            ev.tag_done()

        # insert data
        fields = ['ev_type', 'ev_data', 'ev_time']
        curs = dst_db.cursor()
        dst_queue = self.cf.get('dst_queue_name')
        pgq.bulk_insert_events(curs, rows, fields, dst_queue)

script = QueueMover('queue_mover', 'src_db', 'dst_db', sys.argv[1:])
script.start()
Use-case: email sender

- Non-transactional, so need to track event-by-event
- Needs to commit at each event
Use-case: replication (Londiste)

- Per-batch tracking on remote side
- COPY as a parallel consumer
  - Register, then start COPY
  - If COPY finishes, applies events from queue for that table
  - Then gives it over to main consumer
- Example session:

  $ ed replic.ini; ed ticker.ini
  $ londiste.py replic.ini provider install
  $ londiste.py replic.ini subscriber install
  $ pgqadm.py -d ticker.ini ticker
  $ londiste.py -d replic.ini replay
  $ londiste.py replic.ini provider add table1 table2 ...
  $ londiste.py replic.ini subscriber add table1 table2 ...
Future: cascaded queues

- The goal is to have exact copy of queue in several nodes so reader can freely switch between them.
- Exact means tick_id + events. For simplicity the txids and snapshots are not carried over.
- To allow consumers to randomly switch between nodes, the global horizon is kept. Each node has main worker that sends its lowest tick_id to provider. Worker on master node send global lowest tick_id to queue, where each worker can see it.
- Such design allows workers to care only about 2 node.
- Fancy stuff: merging of plproxy partitions.
Questions?
### PgQ queue info table

```sql
create table pgq.queue (  
    queue_id serial,  
    queue_name text not null,  
    queue_ntables integer not null default 3,  
    queue_cur_table integer not null default 0,  
    queue_rotation_period interval not null default '2 hours',  
    queue_ticker_max_count integer not null default 500,  
    queue_ticker_max_lag interval not null default '3 seconds',  
    queue_ticker_idle_period interval not null default '1 minute',  
    queue_data_pfx text not null,  
    queue_event_seq text not null,  
    queue_tick_seq text not null,  
);
```