Blazingly Fast Message Queue on Postgres with Rust

Adam Hendel <adam@tembo.io>, @adamhendel, github@chuckhend
Founding Engineer, Tembo.io

Agenda

01 Use cases and origin

02 PGMQ is not the first queue...

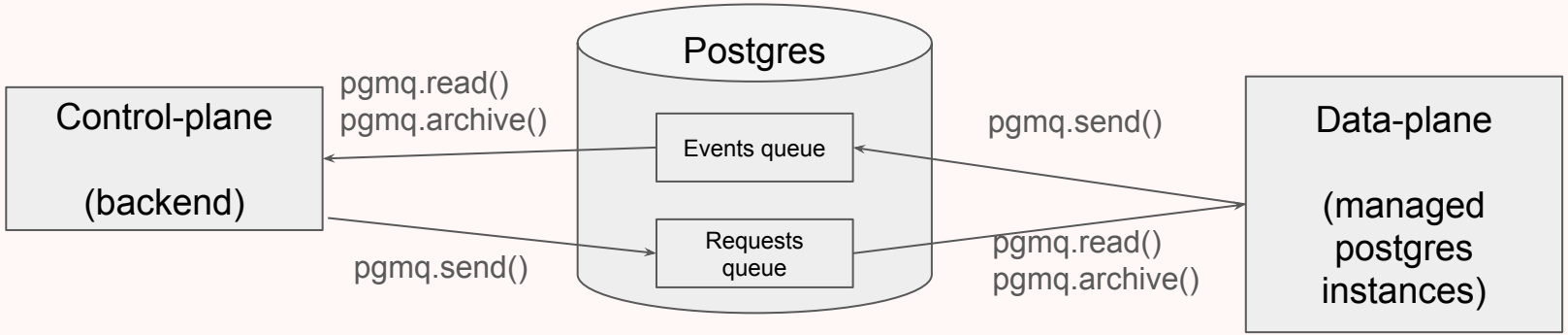
03 API and operations overview

04 Preliminary Benchmarks

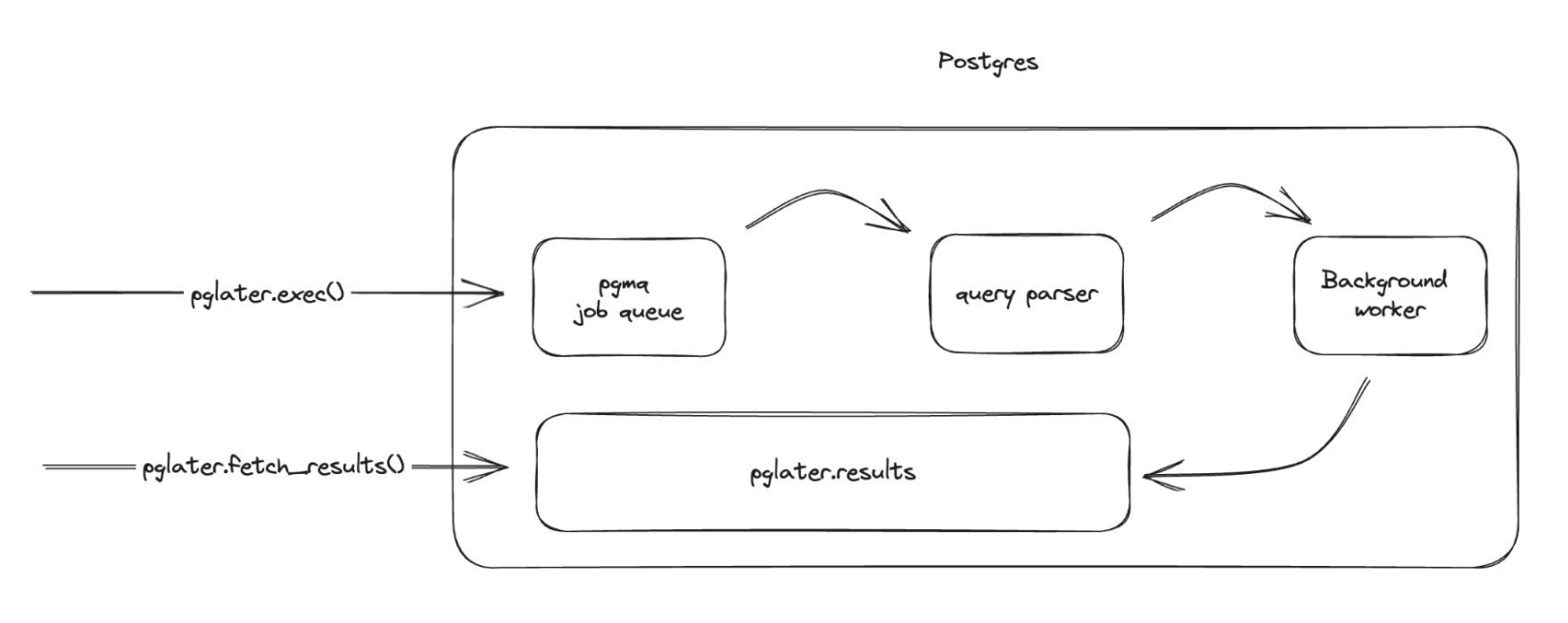
05 Conclusion

Tembo Cloud Platform

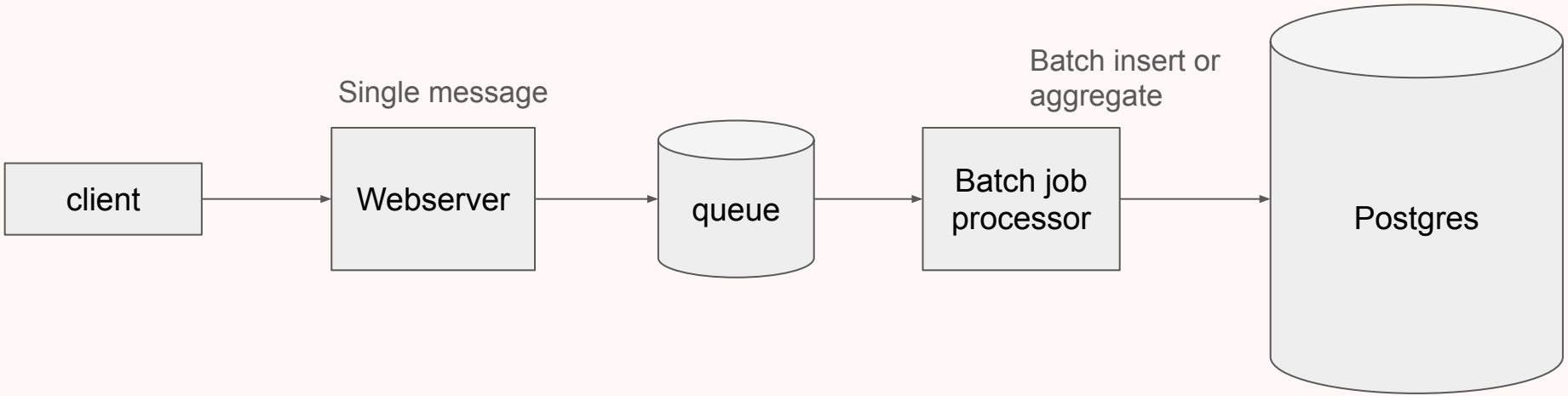
- Control-plane, data-plane, queues in between → lead to development of queue
- Rust producer, rust consumer, lead to a Rust library for queue on Postgres
- Later moved from Rust lib into a Postgres extension using **PGRX**



pgmq runs behind pg_later



Batch processing



PGMQ Features

- **Lightweight**
  - Zero external processes or background worker, just an extension w/ functions
  - Low operational maintenance
- **Exactly-once delivery**, within visibility timeout
- **Simple SQL API**
  - Developer friendly API compatible with any language with a Postgres driver
  - Supports either Delete() or Archive() (retention) of messages
  - Single, batch, and long poll() API
- Build using pgrx, framework for developing Postgres extensions in Rust. Started as a Rust crate then evolved into a Postgres extension.

tembo

PGMQ is not the first...

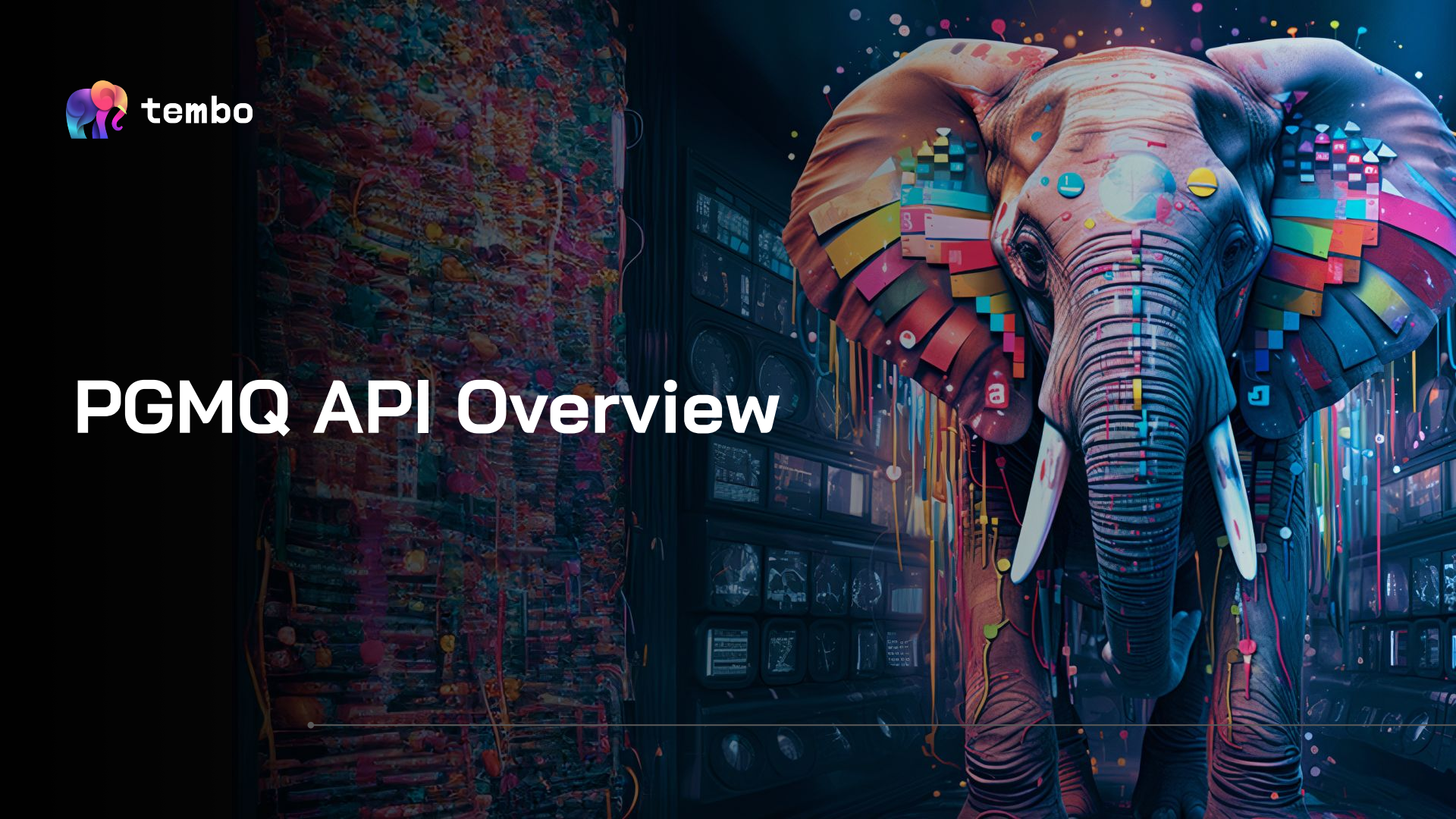
Queues on Postgres

- PGQ - the OG queues on Postgres?
- Postgres Message Queue - (SQL extension)
- River - https://brandur.org/river (Go)
- PgBoss - https://github.com/timgit/pg-boss (Javascript)
- Crunchy - https://www.crunchydata.com/blog/message-queuing-using-native-postgresql
- Dagster - https://dagster.io/blog/skip-kafka-use-postgres-message-queue
- And many more HN articles, projects

PGMQ has lowest complexity, lowest operational maintenance, and accessible to all languages

tembo

PGMQ API Overview



Visibility Timeout (VT) - Inspired by SQS and RSMQ

- Timestamp at which a message can be read by consumers

- Consumer sets VT to a time in the future
- Message unable to be consumed until **now() > VT**
- Message guaranteed to be read “exactly-once” when consumer delete() or archive() that message **before VT** elapses.

- Using a VT means no additional maintenance worker
  - Autovacuum worker handles bloat
  - VT by design is checked on read()
- At-least-once delivery in effect after VT expires

Create a queue

```
select pgmq.create('prague');
```

```
CREATE {maybe_unlogged} TABLE IF NOT EXISTS pgmq.q_{name} (
    msg_id BIGINT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    read_ct INT DEFAULT 0 NOT NULL,
    enqueued_at TIMESTAMP WITH TIME ZONE DEFAULT now() NOT NULL,
    vt TIMESTAMP WITH TIME ZONE NOT NULL,
    message JSONB

CREATE INDEX IF NOT EXISTS q_{name}_vt_idx ON pgmq.q_{name} (vt ASC);

…create archive table..create archive index…etc.
```

Sends are inserts

```
select pgmq.send(queue_name => 'prague', msg => '{"hello": "world0"}');
```

```
INSERT INTO pgmq.q_{queue_name} (vt, message)
VALUES {vt, message..}
 RETURNING msg_id;
```

Each queue is 1 table

msg_id	read_ct	enqueued_at	vt	message
1	0	2023-12-11 15:12:04.827027-06	2023-12-11 15:12:04.828361-06	{"hello": "world0"}
2	0	2023-12-11 15:12:21.624635-06	2023-12-11 15:12:21.625033-06	{"hello": "world1"}
3	0	2023-12-11 15:12:21.624635-06	2023-12-11 15:12:21.625095-06	{"hello": "world2"}

Reads....

```
select * from pgmq.read(
  queue_name => 'prague',
  vt => 30,
  qty => 1
);
```

```
 msg_id | read_ct |          enqueued_at          |              vt               |       message
--------+---------+-------------------------------+-------------------------------+---------------------
      1 |       1 | 2023-12-11 15:12:04.827027-06 | 2023-12-11 15:25:21.561992-06 | {"hello": "world0"}
(1 row)
```

Reads do all the work

```
WITH cte AS
      (
          SELECT msg_id
          FROM pgmq.q_{queue_name}
          WHERE vt <= clock_timestamp()
          ORDER BY msg_id ASC
          LIMIT {qty}
          FOR UPDATE SKIP LOCKED
      )
UPDATE pgmq.q_{queue_name} t
SET
    vt = clock_timestamp() + interval '{vt} seconds',
    read_ct = read_ct + 1
FROM cte
WHERE t.msg_id=cte.msg_id
RETURNING *;
```

Why is there a CTE? this…

Deletes are simple

```
select pgmq.delete(
        queue_name => 'prague',
        msg_id => 1
);
```

```
DELETE FROM pgmq.q_{queue_name}
WHERE msg_id = {msg_id}
RETURNING msg_id;
```

Archive is a delete + insert

```
select pgmq.archive(
     queue_name => 'prague',
     msg_id => 2
);
```

```
WITH archived AS (
  DELETE FROM pgmq.q_{queue_name}
  WHERE msg_id = ANY(msg_id)
  RETURNING msg_id, vt, read_ct, enqueued_at, message
)
INSERT INTO pgmq.q_{queue_name} (msg_id, vt, read_ct,
enqueued_at, message)
SELECT msg_id, vt, read_ct, enqueued_at, message
FROM archived
RETURNING msg_id;
```

PGMQ API

pgmq.pop() – read and delete, at-most-once delivery

pgmq.set_vt() – change the VT of an existing message

pgmq.purge_queue() – delete all the messages on a queue

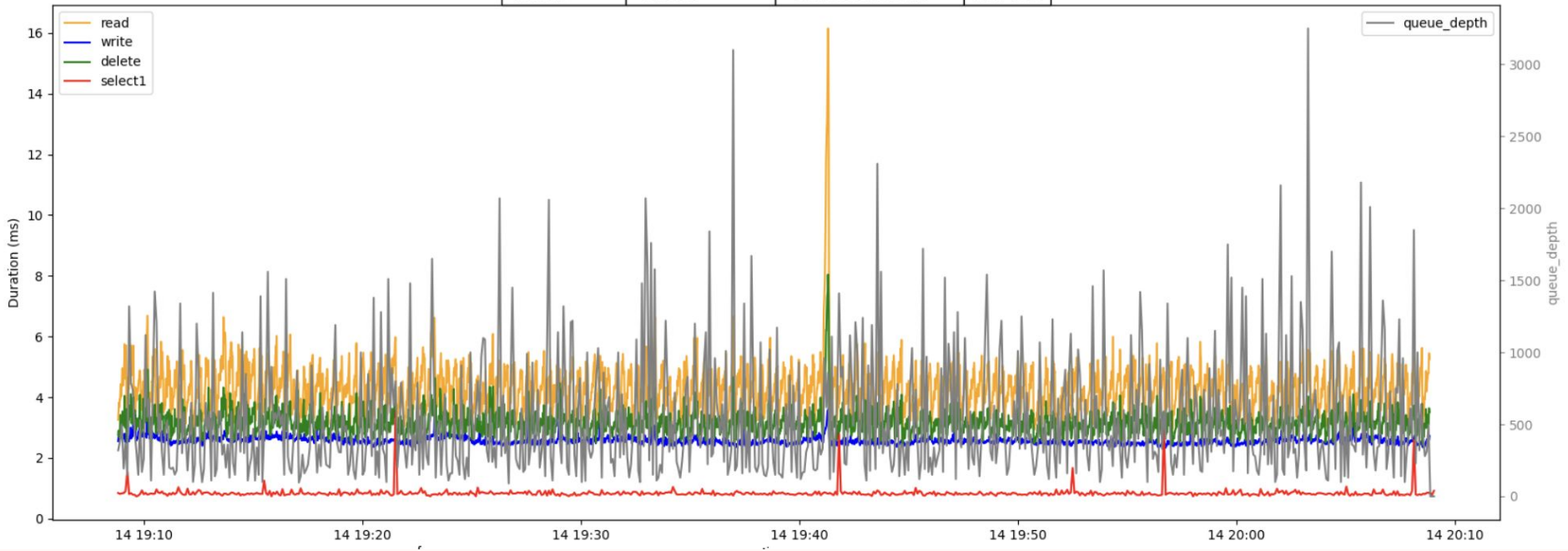
more…

See docs for complete API
https://tembo-io.github.io/pgmq/api/sql/functions/


tembo
Early Benchmarks
(more to come)

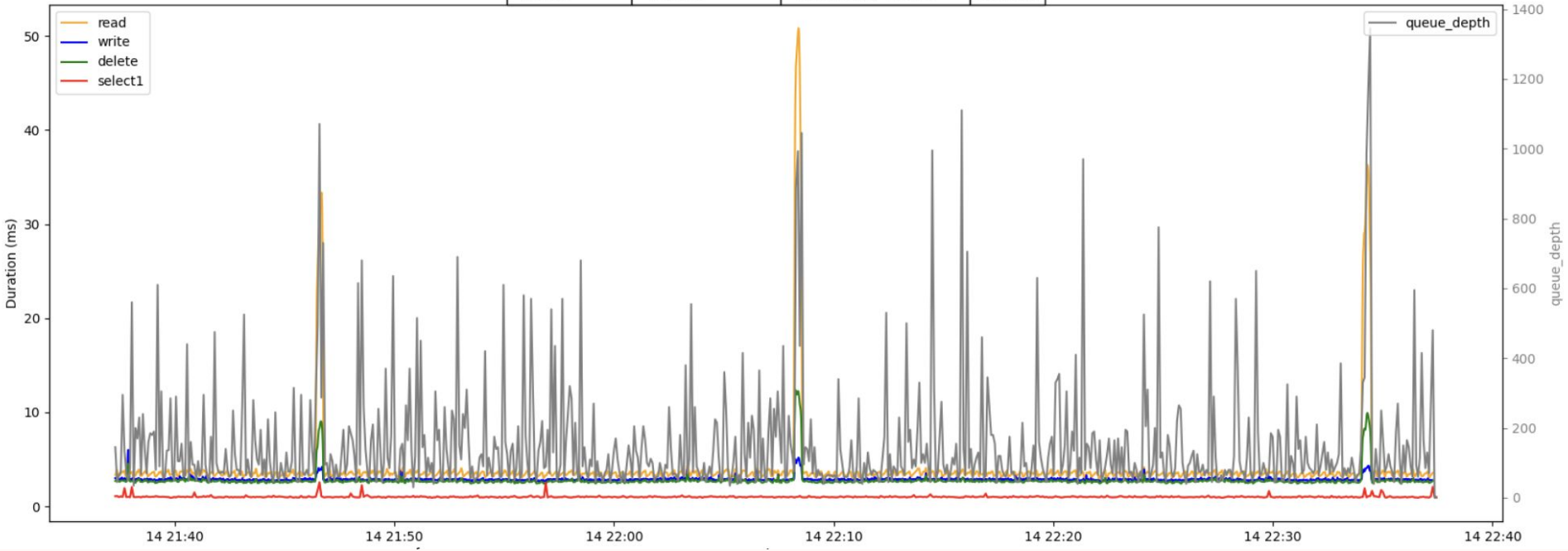
1 hour. 16 vCPU, 32GB RAM, (22 byte message) 5 producers, 40 consumers – batch size 10 Non-partitioned queue

Operation	Duration (s)	Total Messages	msg/s
delete	3,600	69,960,270.0	19,433
read	3,600	69,960,270.0	19,433
write	3,600	69,960,270.0	19,433



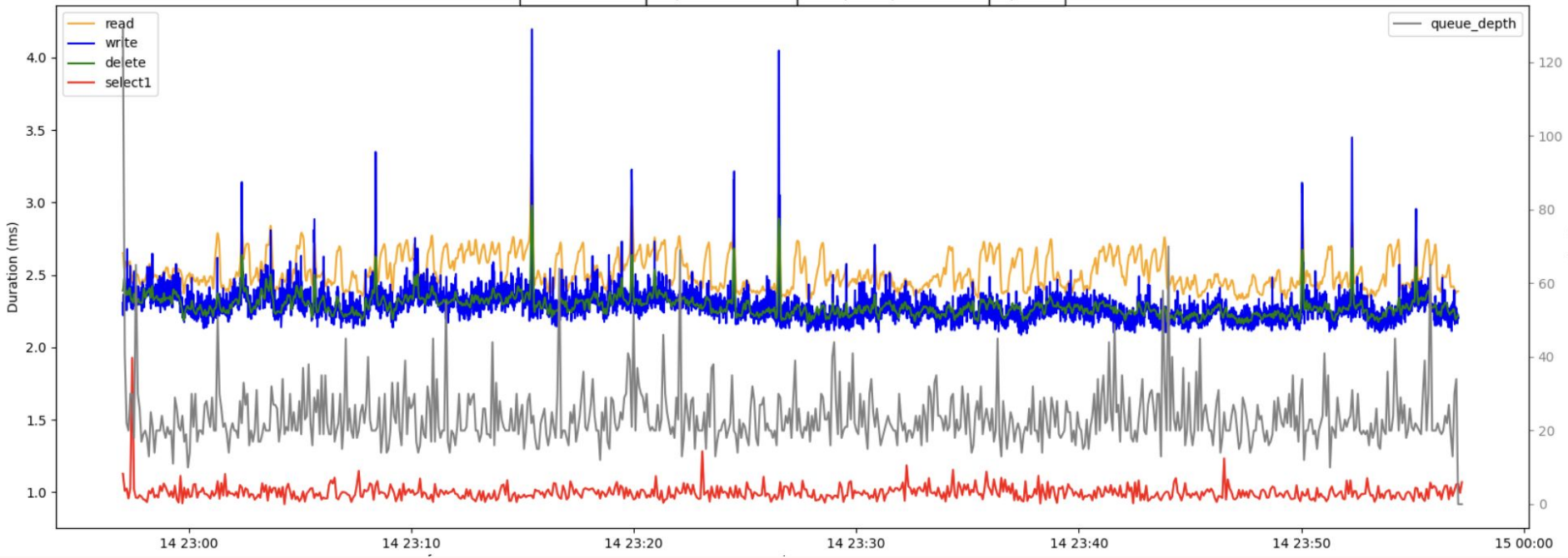
1 hour. 16 vCPU, 32GB RAM, (1KB message) 5 producers, 40 consumers – batch size 10
Non-partitioned queue

Operation	Duration (s)	Total Messages	msg/s
delete	3,600	31,192,410.0	8,664
read	3,600	31,192,410.0	8,664
write	3,600	31,192,410.0	8,664

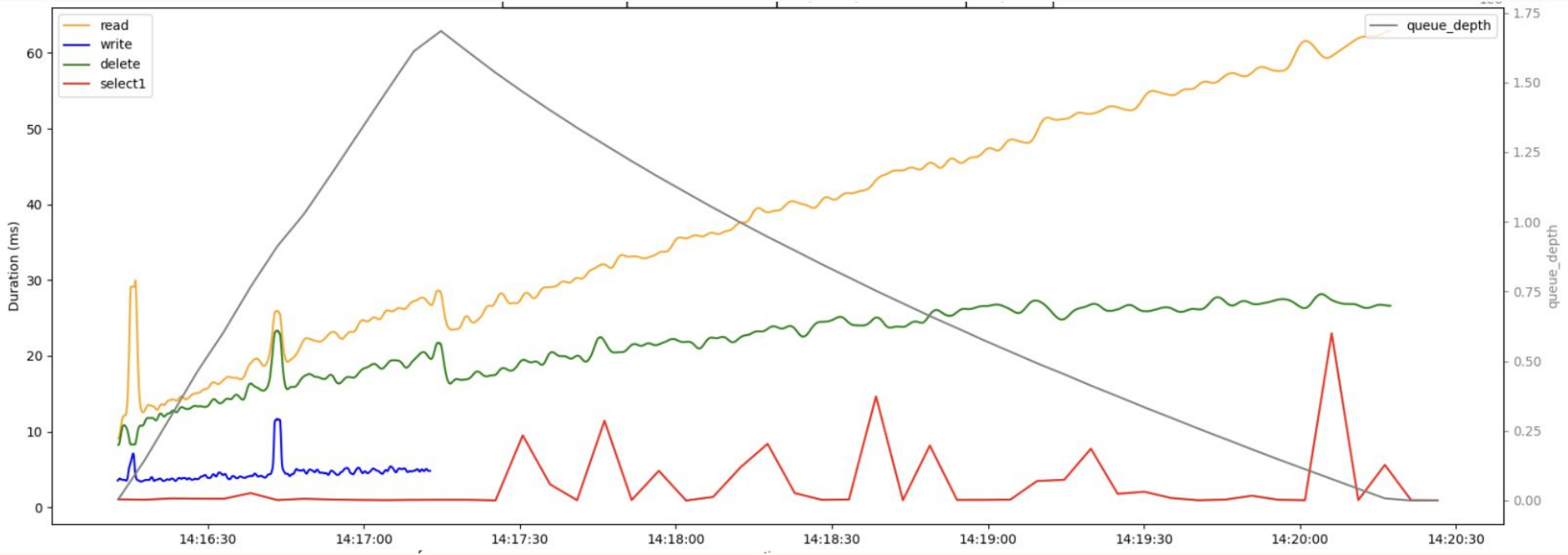


1 hour. 16 vCPU, 32GB RAM, (22 byte message) 10 producers x batch (1), 30 consumers batch(10) Non-partitioned queue

Operation	Duration (s)	Total Messages	msg/s
delete	3,599	15,775,182.0	4,382
read	3,599	15,775,182.0	4,382
write	3,600	15,775,182.0	4,381



Autovacuum disabled (yikes)



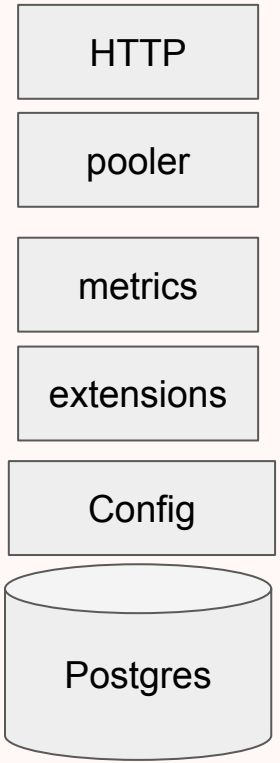
Conclusion

- These are early benchmarks. Stay tuned for more benches including partition queues, resource consumption, etc.

- Vacuum is critical
- Low enough latency, high enough throughput for most workloads
- Keep message sizes small
- Tune batch size to the use case
- Compute isolation for high throughput workloads

Recommended MQ Stack

- HTTP interface w/ authentication – PostgREST
- Connection pooler - pgbouncer
- Metrics/Alert - queue length, message age, total messages
- Extensions – pgmq, pg_partman
- Postgresql.conf - mostly shared buffers, autovacuum
- Open Source Postgres, dedicated to MQ workload



https://github.com/tembo-io/tembo/blob/main/tembo-operator/src/stacks/templates/message_queue.yaml

Next...

Looking for feedback and contributors!

- Benchmarking - partitioned queues, LARGE message sizes, tuning, etc.
- Bridge - connect pgmq to external queues (PG, SQS, RabbitMQ, Kafka, etc)
- alert/notify - consumers receive messages without continuously polling
- Serialization options - MessagePack, Protobuf, Avro...

Community Contributors

https://github.com/tembo-io/pgmq/graphs/contributors

- Felipe Stival - https://github.com/v0idpwn
- Craig Pastro - https://github.com/craigpastro
- Dorian Hoxha https://github.com/ddorian

Demo?


tembo
Thank you!
Give us a star!
https://github.com/tembo-io/pgmq
Questions?
Email me at adam@tembo.io
Tweet at @adamhendel