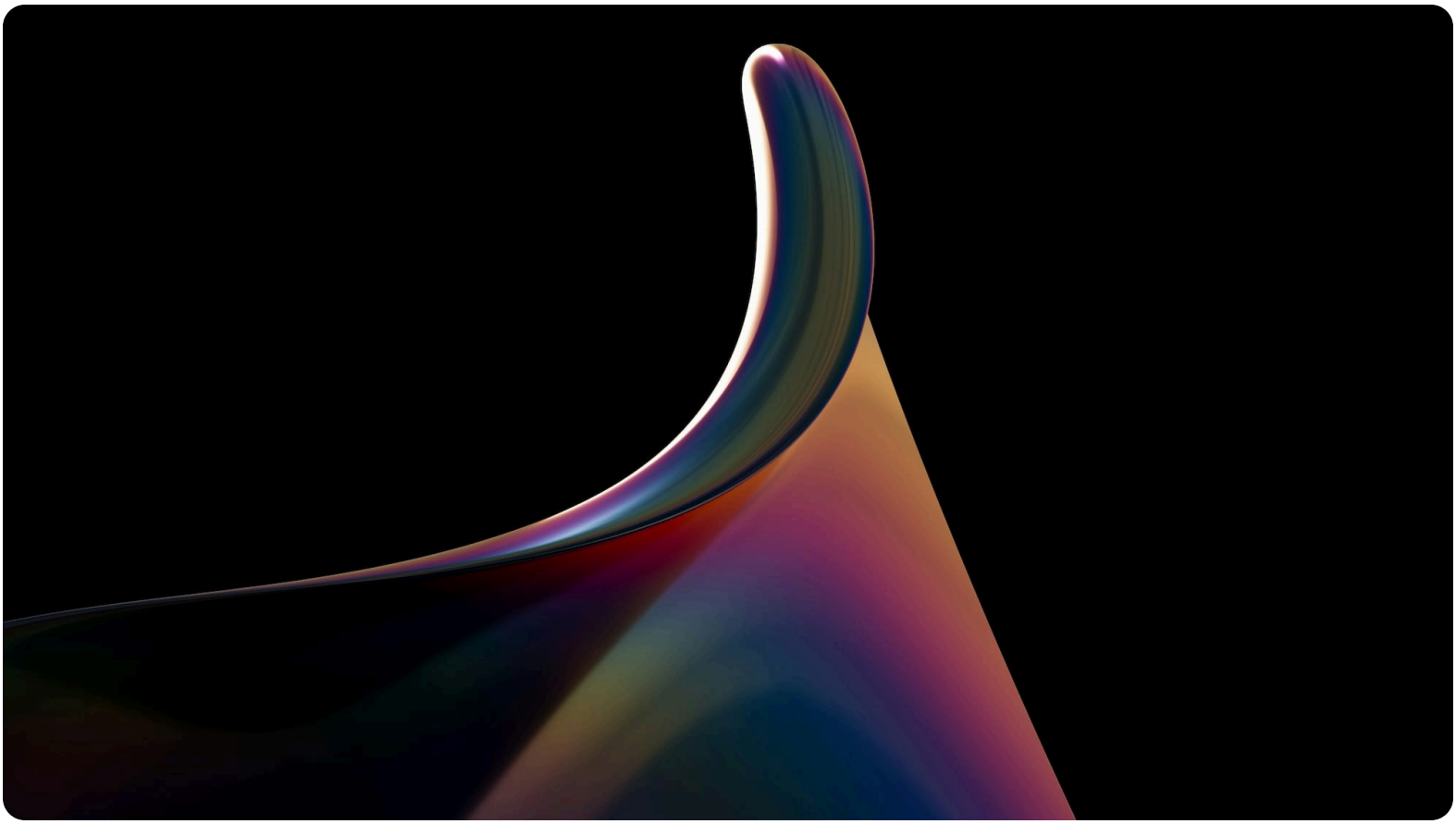
← Back to posts

Request-reply in Postgres

Anthony Accomazzo
@accomazzo

• Oct 23, 2024 • 7 min read



Who doesn't like to go a little overboard with Postgres? Extensions like pgsql-http push what we think is possible (or permissible) to do in the database.

I wondered the other day if you could build a request-reply mechanism using plain Postgres components. Turns out, you can! Whether or not you should is left as a decision for the reader.

Along the way, I'll pull in a bunch of Postgres tools we rarely get a chance to use like unlogged tables, advisory locks, and procedures:

Pub-sub vs request vs request-reply

Postgres has support for pub-sub with its `LISTEN/NOTIFY` feature. But pub-sub is limited. The sender doesn't know if any processes are around to handle its notification. And the sender can't receive a reply.

One layer up from the pub-sub pattern is the **request** pattern. In a request, the sender should raise if there are no handlers. This gives the sending process firmer guarantees that its request will be handled. You can also specify that only one handler should receive the request.

Finally, in the **request-reply** pattern, the sender blocks until it receives a response from the handler. This gives the sender the most guarantees, as it knows for sure whether or not the request was successfully handled. And, the sender can receive a response payload, which it might use to continue its operation.

Why doesn't Postgres have request or request-reply?

Before building these things in Postgres, it's worth asking why they don't come packaged with the database. The reasons are reasonable 😄

Pub-sub is a great fit for a database system. Publishing adds very little overhead to transactions, as it's a non-blocking broadcast.

Requests seem like they'd be a nice improvement to pub-sub. After all, what's the point of publishing if nothing is around to hear it?

The issue with requests is that you're coupling transactions to the lifecycle of an external service. If that service is down, your requests will fail, and so your transactions will grind to a halt.

So, the correct design pattern here is to treat `NOTIFY` as an *optimization*, not as critical communication. If your service really needs to know about database changes, it should have some other primary way of detecting them. It can poll the database, for example. Then, you can layer `NOTIFY` on top to reduce polling frequency *and* learn about changes instantly – a great optimization.

Request-reply is where things really get out of hand. If this is happening inside of a transaction, you're now blocking a process – a precious resource in Postgres – on a reply. What if the service takes 30 seconds to respond?

When might you use request/request-reply?

There are two situations where these patterns in Postgres aren't so crazy:

Service-to-service communication in a radically simple setup

Maybe you want to keep your stack as simple as possible. You don't want or need to add more infrastructure to do service-to-service communication.

With this approach, your app instances just need to connect to Postgres and they have everything they need to do their jobs and coordinate.

You're building a tool with a Postgres interface

Think of a tool like Supabase or the aforementioned pgsql-http, where there's a lot of application logic happening *inside* Postgres. The more logic that lives in Postgres instead of your application, the more you'll be tempted to reach for power features like this.

The code

I built out a proof-of-concept, which you can checkout on GitHub.

How it works

The sender

As you'll see, using a table is key to (1) overcoming limitations with payload size and (2) providing a channel for the requester to receive a reply.

The `request_reply` table:

```
create type request_reply_state as enum ('sending', 'pr

create unlogged table request_reply (
  id serial primary key,
  channel text not null,
  request text not null,
  response text,
  state request_reply_state not null default 'sending'
);
```

`channel` is the channel to use in `LISTEN/NOTIFY`. `request` is the payload of the request. `response` will be used for the payload of the response.

Finally, a great use case for unlogged tables! Unlogged tables in Postgres are far more efficient to write to, as they don't write to the WAL. But they're not crash-safe, which limits their use cases. For temporary data like our request-reply mechanism, they're a great fit.

After inserting into the `request_reply` table, you can emit a `NOTIFY` message to get a handler to respond. The `NOTIFY` will broadcast on the `channel` specified. The body of the `NOTIFY` will be the `id` for the `request_reply` entry.

You might be tempted to insert into `request_reply` , emit the `NOTIFY` message, then await the response all in the same query/function call. However, you need to commit the row first so that it is visible to other sessions. After you commit, Postgres will send your `NOTIFY` to the handlers. *Then* you can block, awaiting the response.

> Later, in "See it in action", I describe how you can use a procedure to turn the operation into a one-liner.

So, you can use two function calls, `request()` and then `await_reply()`.

Before inserting and broadcasting, Postgres can check if anyone is listening. If not, Postgres should raise:

```
create or replace function request(p_channel text, p_re
returns int as $$
declare
  v_id int;
begin
  -- Check if anyone is listening on the channel
  if not exists (select 1 from pg_stat_activity where w
    raise exception 'No listeners on channel `%`', p_ch
  end if;

  -- Insert the request and get the ID
  insert into request_reply (channel, request) values (

  -- Notify listeners
  -- Postgres sends after the commit completes
  perform pg_notify(p_channel, v_id::text);

  return v_id;
end;
$$ language plpgsql;
```

Then, the sender needs to block and await the reply. To do so, you can poll the table `request_reply` while waiting, checking if the `state` has transitioned to `replied`. An optimization to that is to use advisory locks:

1. Poll the table until the handler has picked up the message (`request_reply.state != 'sending'`).
2. Then, move from polling to trying to acquire an advisory lock. As you'll see, when the handler picks up the message, it will acquire a lock.

3. Block until the handler releases the lock.

Again, this is an optimization that ensures the polling period is brief. Advisory locks release instantly, so the sender will be able to immediately see the reply when it's ready.

Here's `await_reply()`, which accepts the `id` of the `request_reply` entry:

```
create or replace function await_reply(v_id int)
returns text as $$
declare
  v_response text;
begin
  -- Wait for the response
  loop
    -- Check if the state has changed from 'sending'
    if exists (select 1 from request_reply where id = v
      -- Try to acquire the advisory lock
      if pg_try_advisory_lock(v_id) then
        -- Lock acquired, fetch the response and delete
        delete from request_reply where id = v_id retur
        -- Release the lock
        perform pg_advisory_unlock(v_id);
        return v_response;
      end if;
    end if;
    -- Wait a bit before trying again
    perform pg_sleep(0.1);
  end loop;
end;
$$ language plpgsql;
```

When the handler releases the lock, the sender can continue. The sender runs a `delete` query to remove the message from `request_reply` and retrieve the payload.

The handler

In your application code, you can register a listener for the `NOTIFY` broadcast. When it receives a message, it:

1. Opens a transaction.
2. Runs an `update ... returning` to retrieve the message, setting the state to `processing`. It simultaneously acquires an advisory lock.
3. The handler can process the request.
4. The handler runs a final `update`, setting the `response`.
5. The handler releases the advisory lock.

Here's what the first update query looks like:

```
begin;
with available_message as (
  select id, request
  from request_reply
  where id = $1 and state = 'sending'
  order by id
  for update skip locked
  limit 1
)
update request_reply r
set state = 'processing'
from available_message am
  where r.id = am.id
  returning r.request, pg_try_advisory_lock(r.id) as lo
```

The `available_message` CTE selects the message `for update skip locked`. This prevents other listeners from grabbing and processing the message.

The `update` query acquires the advisory lock, which will block the sender until the response is ready.

After the handler processes the request, it can run the final `update` and set the `response`:

```
with updated as (
  update request_reply
  set state = 'replied', response = $2
  where id = $1
  returning id
)
select pg_advisory_unlock(id) from updated;
```

When the update is complete, you can unlock the advisory lock, which unblocks the sender so it can retrieve the response and return.

See it in action

I built a fun demo demonstrating a request-reply that generates vector embeddings for a table in the database. You could run a background job with `pg_cron` that populates missing embeddings.

To make `request()/await_reply()` work well for queries running directly in Postgres, you can use a *procedure*. A procedure lets you orchestrate function calls. Importantly, you can commit in the middle of a procedure. So, you can:

1. Call `request()`.
2. `commit`, which makes the new entry in `request_reply` available to all senders.
3. `await_reply()`.
4. Handle the reply.

First, create a table of `dune_quotes` and insert some data into it:

```
create table if not exists dune_quotes (
    id serial primary key,
    quote text not null,
    embedding vector(1536)
);

insert into dune_quotes (quote) values
    ('i must not fear. fear is the mind-killer. fear is
    ('he who controls the spice controls the universe.'
    ('the mystery of life isn''t a problem to solve, bu
```

Next, create a procedure that will generate embeddings for `dune_quotes` that have a `null` `embedding`:

```
create or replace procedure process_dune_quote_embeddin
declare
    quote_record record;
    request_id int;
    embedding_json json;
    embedding_array float[];
begin
    -- select quotes without embeddings
    for quote_record in select id, quote from dune_quot
        -- request embedding
        request_id := request('embeddings', quote_recor
        commit;

        -- wait for and retrieve the embedding
        embedding_json := await_reply(request_id)::json

        -- parse the json response and convert to array
        -- assuming the embedding is directly an array
        select array(select elem::float
                     from json_array_elements_text(embe
        into embedding_array;

        -- update the quote with the new embedding
        update dune_quotes
        set embedding = embedding_array::vector(1536)
        where id = quote_record.id;

        -- commit after each update to make it visible
        commit;
    end loop;
end;
$$ language plpgsql;
```

The bones of this proof of concept are available on GitHub.

The neat part about this example is that we can do all this without needing an extension like pgsql-http. We can use native Postgres components and move all other logic over to our application.

Sequin sends Postgres changes to your applications and services. It's designed to never miss an insert, update, or delete and provide exactly-once processing of all changes.

Try Sequin now to add async triggers to your existing Postgres tables. Or, Sequin can add streaming mechanics to Postgres to do the work of SQS / Kafka without the operational overhead.

© Sequin Labs, Inc. 2024