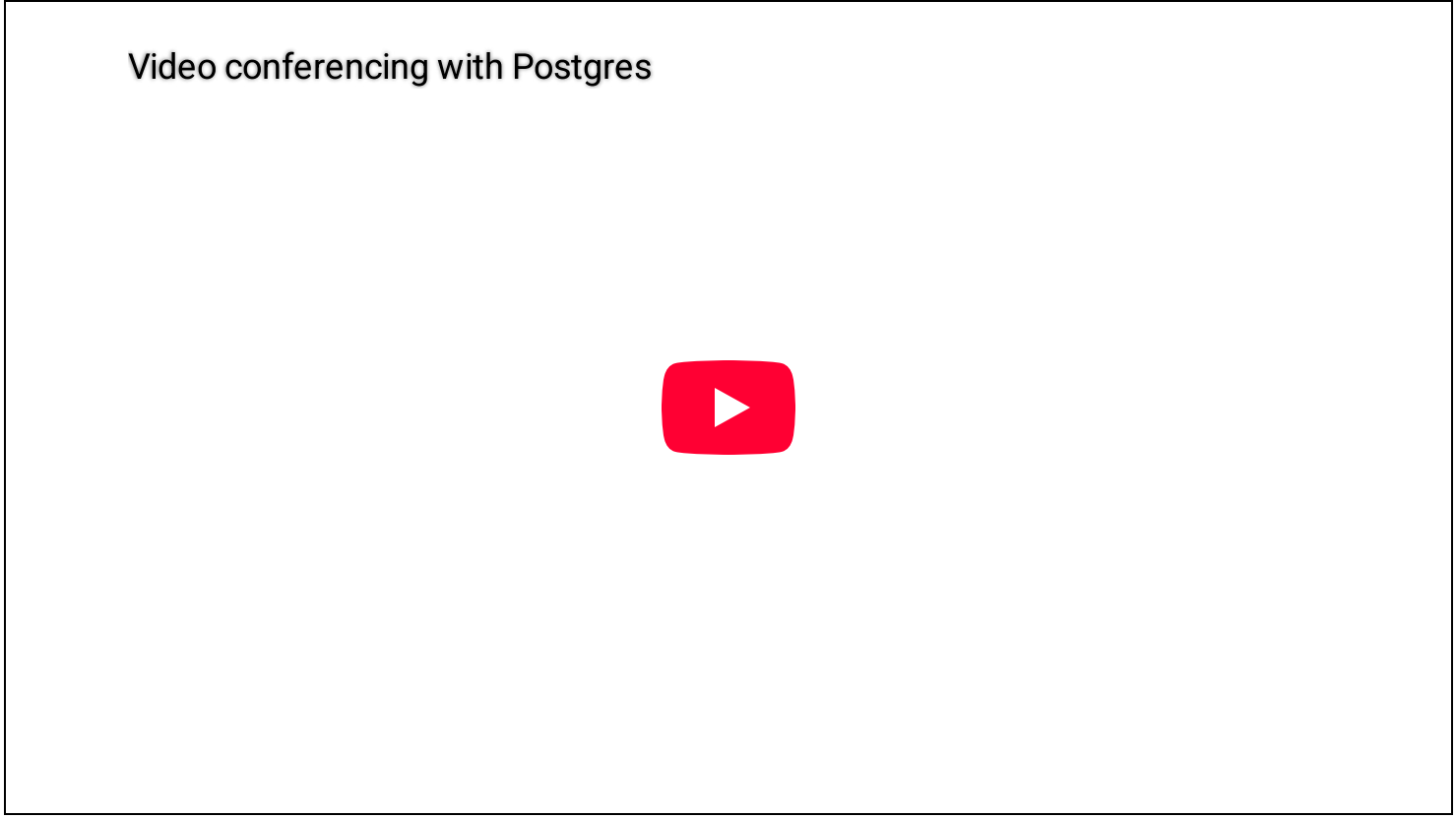
Video Conferencing with Postgres

By Nick Van Wiggeren | February 27, 2026

Yesterday on X, SpacetimeDB tweeted that they had done "the world's first video call over a database" and, in their own way, invited anyone else to give it a try.

Credit to them - it's a cool idea! In short, they built a frontend that captures audio and video from the browser's media APIs, encodes them into compact frames (PCM16LE audio, JPEG video), sends them to a database that acts as a real-time message broker, and streams them back out to the other participant's browser for playback.

Fortunately, the implementation is open sourced (https://github.com/Lethalchip/SpaceChatDB), so I figured I'd see what it looked like for PostgreSQL, the world's most popular open source database, to host the world's second video call over a database.



How it works

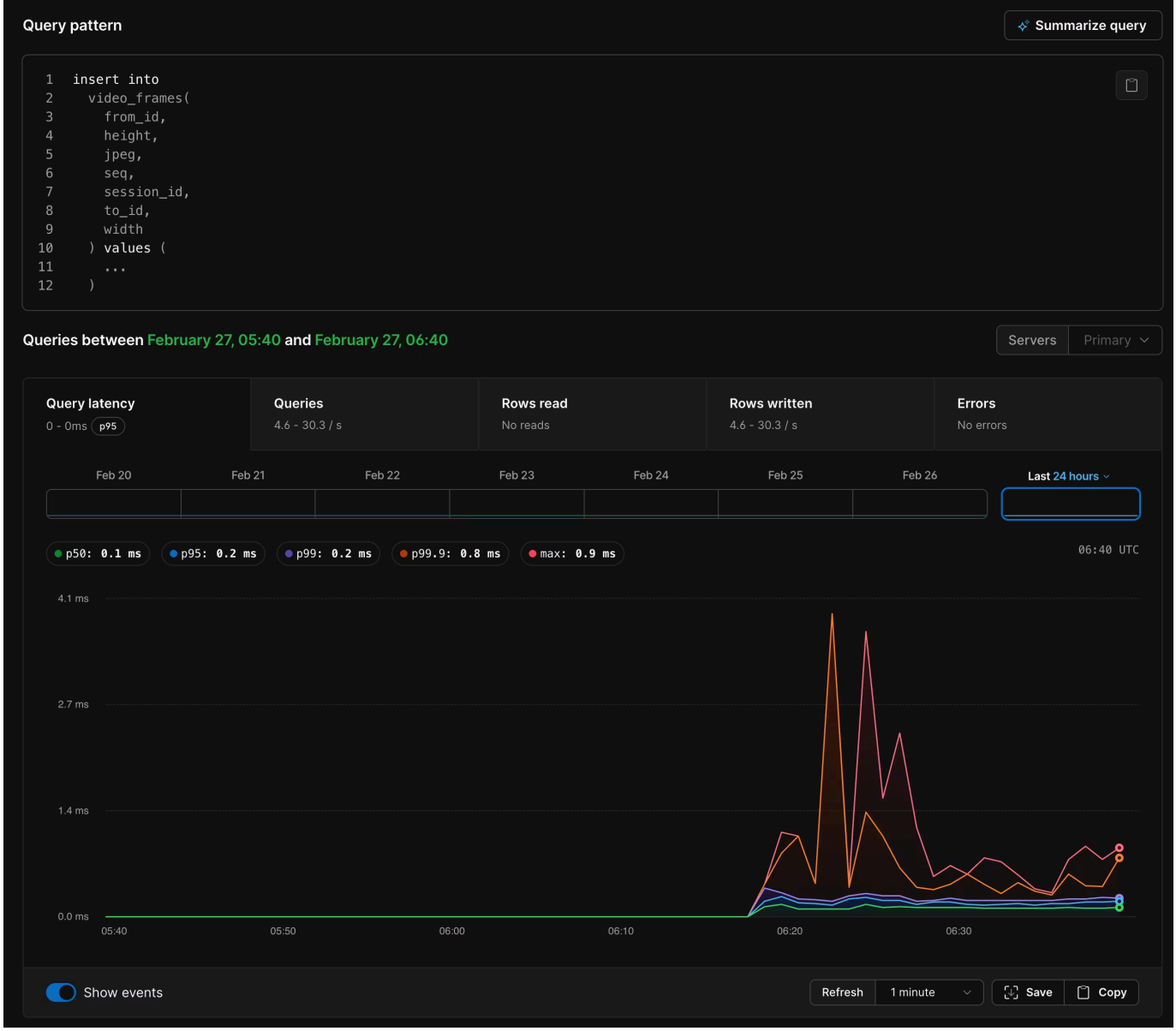
In my implementation, I started with the same SvelteKit frontend, added a small Node.js WebSocket server (`pg-relay`) in the middle, and $5 PlanetScale PostgreSQL as the database.

When you're on a video call:

1. Your browser captures a camera frame, encodes it as a JPEG, and sends it as a binary WebSocket message to `pg-relay`.

2. pg-relay validates that you're in an active call, then runs:

```
INSERT INTO video_frames (session_id, from_id, to_id, seq, width, height, j
VALUES ($1, $2, $3, $4, $5, $6, $7)
```



1. PostgreSQL writes this to the WAL (write-ahead log).

2. pg-relay is also running a logical replication consumer on the same database. It sees the new row appear in the replication stream, checks the to_id column, and forwards the raw JPEG bytes over WebSocket to the recipient.

3. The recipient's browser creates a blob URL from the JPEG and renders it in the browser.

Audio works the same way — PCM samples go into an audio_frames table and come out the other side via replication.

Logical replication?

PostgreSQL's logical replication gives us a reliable and ordered change stream. You get INSERT, UPDATE, and DELETE events for every table in the publication, delivered in commit order. This means we don't have to poll

Postgres with SELECT statements from the table fast enough to render 15fps video.

This means the same mechanism that pushes video frames to call participants also pushes chat messages, user presence changes, and call state transitions. When someone sends a chat message, it gets INSERTed, appears in the replication stream, and gets forwarded to every connected client. When a user disconnects, their row gets DELETEd, and everyone sees them go offline.

For video, the table looks like this:

```
CREATE TABLE video_frames (
    id          BIGSERIAL PRIMARY KEY,
    session_id  UUID      NOT NULL,
    from_id     TEXT      NOT NULL,
    to_id       TEXT      NOT NULL,
    seq         INT       NOT NULL,
    width       SMALLINT  NOT NULL,
    height      SMALLINT  NOT NULL,
    jpeg        BYTEA     NOT NULL,
    inserted_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

There's nothing special about this table. It's just rows with a JPEG in a BYTEA column. This incurs a modest amount of egress, but nothing a database like this can't handle.

The media to PostgreSQL pipeline

On the capture side, the browser grabs camera frames, draws them to an offscreen canvas, and calls `canvas.toBlob()` to get a JPEG. Audio comes from an `AudioWorkletNode` that collects PCM samples, resamples them to 16kHz mono, and encodes them as 16-bit little-endian integers.

Both get packed into binary WebSocket frames with a small JSON header (session ID, sequence number, recipient) and shipped to the relay.

On the playback side, incoming JPEGs get turned into blob URLs and set as the `src` of an `<img>` tag. Audio samples get decoded back to floats and scheduled on a Web Audio `AudioBufferSourceNode` with a small jitter buffer.

The whole thing runs at 640x360 @ 15fps with JPEG quality 0.65. Each frame is roughly 25-40KB, which means it's pushing about 375–600 KB/s of video through Postgres per direction.

Accumulating rows

I didn't want to keep every video frame forever. At 15fps, you'd accumulate about 108,000 rows/hour per active call. So there's a cleanup job that runs every 2 seconds and prunes frames older than 5 seconds:

```
DELETE FROM audio_frames WHERE inserted_at < NOW() - INTERVAL '5 seconds';
DELETE FROM video_frames WHERE inserted_at < NOW() - INTERVAL '5 seconds';
```

This means that for every call, we'd expect to have about 5-7 seconds of frames in the table at any given time, or about 150 rows total. Sure enough:

```
SELECT
  from_id,
  COUNT(*) AS frames_5s,
  ROUND(COUNT(*) / 5.0, 1) AS approx_fps
FROM video_frames
WHERE inserted_at >= NOW() - INTERVAL '5 seconds'
GROUP BY from_id
ORDER BY frames_5s DESC;
                              from_id                               | frames_
--------------------------------------------------------------------+--------
 06cad97a128947a58e8ff754ec1171d4200c4d774b5c43c9b7637f11bba61036 |
 4f984feaee1042939383b0fffb3f1fc172d28aa92b654551a02c618788021995 |
(2 rows)
```

Look at that - our own $5 PostgreSQL is streaming bidirectional 15fps video!

What's really cool about this is that if we wanted, we could keep the frames around. Each JPEG is being durably persisted to PostgreSQL, crash-safe, replicatable and ready for querying later on. My $5 PostgreSQL has the throughput to store hours of video that I can combine later.

I can even pull one of the frames right out of the database and render it in my terminal:

Could we have done this another way?

LISTEN/NOTIFY was my first idea. Postgres has a built-in pub/sub mechanism — NOTIFY on a channel, LISTEN on the other end, messages arrive in real time. We could skip the media tables entirely and just blast JPEG bytes through a notification channel.

The problem is an enforced 8KB payload limit. A video frame at 640x360 is 25-40KB. We'd have to chunk every frame into 4-5 separate notifications, reassemble them on the other end, handle ordering, handle dropped chunks — at which point you've built a worse TCP on top of a notification system. Audio frames would mostly fit under 8KB, so we could do a hybrid approach, but splitting the media pipeline across two different transport mechanisms is the kind of complexity I wasn't interested in.

Unlogged tables go the other direction. Instead of changing how we get data out of Postgres, they change how data goes in. Unlogged tables skip the WAL entirely — no write-ahead logging, no fsync, no crash recovery. Inserts

are faster because Postgres isn't making durability guarantees about video frames.

I didn't like that because logical replication reads from the WAL. If the table doesn't write to the WAL, it doesn't appear in the replication stream. To make this work, we'd have to fall back to polling — `SELECT * FROM video_frames WHERE seq > $1` in a loop. This might have worked fine, maybe better - but something about rendering video from a polling loop of `SELECT *` didn't feel good.

How'd it go?

You be the judge. It exceeded my expectations.

Our $5 PlanetScale PostgreSQL was able to keep up with the insert rate of live video and audio, and browsers are optimized enough that they can take raw JPEG frames and turn them into video pretty convincingly.

The only adjustments I made after I got it working the first time were adding some boundaries to keep audio in sync. Video frames render instantly (we just swap the image), but audio needs to be buffered and scheduled ahead of time to avoid gaps. Getting them to stay in sync required clamping the audio scheduling buffer so it can't drift too far ahead of real time:

```
const now = audioCtx.currentTime;
const clamped = nextPlayTime > now + 0.15 ? now + 0.02 : nextPlayTime;
```

```
    const startAt = Math.max(clamped, now + 0.02);
```

Should you do this?

No! Use WebRTC!

But if you want to understand how logical replication works and want to see how far you can push Postgres as a general-purpose real-time backend, this is a fun way to find out. The entire relay server is about 400 lines of TypeScript.

My fork is at github.com/nickvanw/PgVideoChat. If only Alexander Graham Bell could see us now.