Blog post

Implementing "seen by" functionality with Postgres

2022-07-18 • 33 minute read

Victor
Guest Author

tl;dr: Use HyperLogLog, it's a reasonable approach with great trade-offs and no large architectural liabilities. For a quick & dirty prototype, use `hstore`, which also performs the best with integer IDs.

The year is 2022. You're head DBA at the hot new social site, SupaBook... Your startup is seeing eye-boggling growth because everyone loves fitting their hot-takes in posts restricted to `VARCHAR(256)`.

Why `VARCHAR(256)`? No particular reason, but you don't have time to get hung up on that or ask why -- **you just found out that the priority this quarter is tracking content views across all posts in the app**.

"It sounds pretty simple" a colleague at the meeting remarks -- "just an increment here and an increment there and we'll know which posts are seen the most on our platform". You start to explain why it will be non-trivial, but the meeting ends before you can finish.

Well, it's time to figure out how you're going to do it. There's been a complexity freeze at the company, so you're not allowed to bring in any new technology, but you don't mind that because for v1 you would have picked Postgres anyway. Postgres's open source pedigree, robust suite of features, stable internals, and awesome mascot Slonik make it a strong choice, and it's what you're already running.

(insert record scratch here)

Sure, this scenario isn't real, but it could be - that last part about Postgres definitely is. Let's see how you might solve this problem, as that imaginary DBA.

Experiment setup

We've got the following simple table layout:

In SQL migration form:

hideCopy

```sql
CREATE EXTENSION IF NOT EXISTS uuid-ossp;
CREATE EXTENSION IF NOT EXISTS citext;

-- Create a email domain to represent and cor
CREATE DOMAIN email
AS citext
CHECK ( LENGTH(VALUE) <= 255 AND value ~ '^[a

COMMENT ON DOMAIN email is 'lightly validatec

-- Create the users table
CREATE TABLE users (
    id bigserial PRIMARY KEY GENERATED BY DEF
    uuid uuid NOT NULL DEFAULT uuid_nonmc_v1(

    email email NOT NULL,
    name text,
    about_html text,

    created_at timestamptz NOT NULL DEFAULT N
);

-- Create the posts table
CREATE TABLE posts (
    id bigserial PRIMARY KEY GENERATED BY DEF
    uuid uuid NOT NULL DEFAULT uuid_nonmc_v1(

    title text,
    content text,
    main_image_src text,
    main_link_src text,

    created_by bigint REFERENCES users(id),

    last_hidden_at timestamptz,
    last_updated_at timestamptz,
    created_at timestamptz NOT NULL DEFAULT N
);
```

This basic setup has taken the (imaginary) company quite far -- even though the `posts` table has millions and millions of entries, Postgres chugs along and serves our queries with impressive speed and reliability. Scaling up is the new (and old) scaling out.

How should we do it?

Well we can't pat ourselves for our miraculous and suspiciously simple DB architecture all day, let's move on to the task at hand.

Like any good tinkerer we'll start with the simplest solutions and work our way up in complexity to try and get to something outstanding, testing our numbers as we go.

Try #1: The naive way, a simple counter on every Post

The easiest obvious way to do this is to maintain a counter on every tuple in the `posts` table. It's obvious, and it's almost guaranteed to work -- but maybe not *work well*.

The migration to make it happen isn't too difficult:

hideCopy

```
BEGIN;

ALTER TABLE posts ADD COLUMN seen_by_count;

COMMENT ON COLUMN posts.seen_by_count
  IS 'simple count of users who have seen the

COMMIT;
```

There's one *obvious* glaring issue here -- what if someone sees the same post twice? Every page reload would cause inflated counts in the `seen_by_count` column, not to mention a lot of concurrent database updates (which isn't necessarily Postgres's forte to begin with).

Clearly there's a better way to do things but before that...

Writing a test suite before the CPUs get hot and heavy

How will we know which approach is better without numbers?! Measuring complexity and feeling can only get us so far -- we need to get some numbers that tell us the performance of the solution at the stated tasks -- we need benchmarks.

Before we can declare any solution the best, in particular we need a *baseline!*. The simplest possible incorrect solution (simply incrementing a counter on the Post) is probably a reasonable thing to use as a benchmark, so let's take a moment to write our testing suite.

Let's do this the simplest one might imagine:

- Generate a large amount of users
  - Lets model for 1000, 10k, 100K, 1MM, and 10MM users
- Generate an even larger amount of fake posts attributed to those users
  - This is a bit harder -- we need to define a general distribution for our users that's somewhat informed by real life...

    An average/normalized distribution doesn't quite work here -- on sites like twitter 10% of users create 80% of the tweets!
- Generate a *description* of "events" that describe which post was seen by whom, which we can replay.
  - We want the equivalent of an effect system or monadic computation, which is easier than it sounds -- we want to generate an encoding (JSON, probably) of *what to do*, without actually doing it

We'll just do consistent "as fast as we can" execution (more complicated analysis would burst traffic to be ab it closer to real life)

OK, let's roll our hands up and get it done:

Script: User seeding

Here's what that looks like:

```
/**
 * Generate a list of synthetic users to be l
 *
 * @param {object} args
 * @param {number} [args.count] number of use
 * @param {number} [args.aboutHTMLWordCount]
 * @param {string} [args.outputFilePath] outp
 * @returns {any[][]} List of generated synth
 */
export async function generateUsers(args) {
  const count = args.count || DEFAULT_USER_CO
  const aboutHTMLWordCount = args.aboutHTMLWo

  const outputFilePath = args.outputFilePath
  if (!outputFilePath) {
    throw new Error('output file path must be
  }

  for (var id = 0; id < count; id++) {
    const user = {
      id,
      email: `user${id}@example.com`,
      name: `user ${id}`,
      about_html: fastLoremIpsum(aboutHTMLWor
    }

    // Write the entries to disk (returning r
    if (args.outputFilePath) {
      await appendFile(outputFilePath, `${JSO
    }
  }
}
```

Nothing too crazy in there -- we generate a bunch of JSON, and force it out to disk. It's best to avoid trying to keep it in memory so we can handle much larger volumes than we might be able to fit in memory.

If you'd like to see the code, check out `scripts/generate/users.js` in the repo.

Script: Post seeding

Along with users, we need to generate posts that they can view. We'll keep it simple and take an amount of posts to make, generating from 0 to `count` of those.

It's very similar to the user generation code, with the caveat that we can take into account the 80/20 lurker/poster rule. here's what that looks like:

It's a bit long so if you'd like to see the code, check out `scripts/generate/posts.js` in the repo.

Script: action (API call) seeding/generation

This script is a bit tricky -- we need to inject some randomness in the performing of the following actions:

- Record a new view of a post
- Retrieve just the count of a single post
- Retrieve all the users who saw a post

I've chosen to use `autocannon` so I needed to write a request generation script which looks like this:

```
const process = require('process')

const POST_COUNT = process.env.TEST_POST_COUN
  ? parseInt(process.env.TEST_POST_COUNT, 10)
  : undefined
const USER_COUNT = process.env.TEST_USER_COUN
  ? parseInt(process.env.TEST_USER_COUNT, 10)
  : undefined

/**
 * Request setup function for use with autoca
 *
 * @param {Request} request
 * @returns {Request}
 */
```

```
function setupRequest(request) {
  // ENsure we have counts to go off of
  if (!POST_COUNT || !USER_COUNT) {
    throw new Error('Cannot setup request wit
  }

  // Pick a random post to do an operation or
  const postId = Math.floor(Math.random() * F

  // Choose pseudo-randomly whether to regist
  const operationChoice = Math.floor(Math.rar
  if (operationChoice < 1) {
    // 10% of the time, get *all* the users
    request.method = 'GET'
    request.path = `/posts/${postId}/seen-by/
  } else if (operationChoice < 7) {
    // 60% of the time, get the count of seer
    request.method = 'GET'
    request.path = `/posts/${postId}/seen-by/
  } else {
    // 30% of the time, add a new seen-by ent
    const userId = Math.floor(Math.random() *

    // Most of the time we'll be *setting* se
    // And we'll get the count (so we can shc
    request.method = 'POST'
    request.path = `/posts/${postId}/seen-by/
  }

  return request
}

module.exports = setupRequest
```

Nothing too crazy here, and some back of the envelope estimations on how often each operation would normally be called. These numbers could be tweaked more, but we *should* see a difference between approaches even if we messed up massively here.

If you'd like to see the code, check out `scripts/setup-request.cjs` in the repo.

Glue it all together

Once we're done we need to glue this all together into one script, with roughly this format:

```
export default async function runBenchmark()
  // Start the server
  // Reset before test
  // Generate & insert users
  // Generate & insert posts
  // Generate actions (API Calls) to run
  // Execute the API calls
  // Write JSON results to tmpdir
  // Stop the server
}
```

If you want to see what the code *actually* ended up looking like, check out `scripts/bench.js` in the repo.

Along with the benchmark, we'll standardize on the following settings:

```
export SEEN_BY_STRATEGY=simple-counter # or:
export TEST_USERS_JSON_PATH=/tmp/supabase-see
export TEST_POSTS_JSON_PATH=/tmp/supabase-see
export TEST_POST_COUNT=1000
export TEST_USER_COUNT=100000
export TEST_DURATION_SECONDS=60

## Use custom postgres image built with hll e
## NOTE: `make db-custom-image` must be run b
#export DB_IMAGE=postgres-14.4-alpine-hll
#export DB_IMAGE_TAG=latest
```

Our first run, on the naive solution

Alright, finally we're ready. Let's see what we get on our naive solution. We expect this to be *pretty fast*, because not only is it *wrong*, but it's just about the simplest thing you could do.

On my local machine, here's our baseline (output from `autocannon`):

```

┌─────────┬──────┬──────┬───────┬──────┬──────
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg
├─────────┼──────┼──────┼───────┼──────┼──────
│ Latency │ 0 ms │ 2 ms │ 6 ms  │ 6 ms │ 2.03
└─────────┴──────┴──────┴───────┴──────┴──────

┌───────────┬─────────┬─────────┬─────────┬───
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 9
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 9
├───────────┼─────────┼─────────┼─────────┼
│ Req/Sec   │ 297     │ 318     │ 389     │ 5
├───────────┼─────────┼─────────┼─────────┼
│ Bytes/Sec │ 54.1 kB │ 57.9 kB │ 70.8 kB │ 9
└───────────┴─────────┴─────────┴─────────┴

Req/Bytes counts sampled once per second.
# of samples: 60

┌────────────┬──────────────┐
│ Percentile │ Latency (ms) │
├────────────┼──────────────┤
│ 0.001      │ 0            │
├────────────┼──────────────┤
│ 0.01       │ 0            │
├────────────┼──────────────┤
│ 0.1        │ 0            │
├────────────┼──────────────┤
│ 1          │ 0            │
├────────────┼──────────────┤
│ 2.5        │ 0            │
├────────────┼──────────────┤
│ 10         │ 0            │
├────────────┼──────────────┤
│ 25         │ 0            │
├────────────┼──────────────┤
│ 50         │ 2            │
├────────────┼──────────────┤
│ 75         │ 3            │
├────────────┼──────────────┤
│ 90         │ 5            │
├────────────┼──────────────┤
│ 97.5       │ 6            │
├────────────┼──────────────┤
│ 99         │ 6            │
├────────────┼──────────────┤
│ 99.9       │ 9            │
├────────────┼──────────────┤
│ 99.99      │ 16           │
├────────────┼──────────────┤
│ 99.999     │ 23           │
└────────────┴──────────────┘

23k requests in 60.02s, 4.28 MB read
```

As you might imagine, pretty darn good latency across all the requests.

Back to trying things out

Now that we've got a basic baseline of our tests, let's continue trying out ideas:

Try #2: Storing the users who did the "see"ing, with `hstore`

The next obvious thing (and probably a core requirement if we'd asked around), is knowing *who* viewed each post. Well if we need to know who, then we probably need to store some more information!

Postgres has native support for arrays and a data structure called a `hstore`, so let's try those. It's pretty obvious that having hundreds, thousands, or millions of entries in one of these data structures, inside a tuple isn't the *greatest* idea, but let's try it anyway and let the numbers speak for themselves.

Here's what the migration would look like:

hideCopy

```
BEGIN;

CREATE EXTENSION IF NOT EXISTS hstore;

ALTER TABLE posts ADD COLUMN seen_count_hstor
  NOT NULL DEFAULT ''::hstore;

COMMENT ON COLUMN posts.seen_count_hstore
  IS 'count of users that have seen the post,

COMMIT;
```

`hstore` provides support for both GIST and GIN indices, but after reading the documentation we can conclude that we don't necessarily need those for the current set of functionality.

Caveats

Well as you might have imagined, this is obviously pretty bad and will eventually be hard to scale. If you expect only 0-50 entries in your column `text[]` is perfectly fine, but thousands or millions is another ballgame.

Thinking of how to scale this, a few ideas pop to mind:

- Compress our columns with LZ4 which is newly supported `TOAST` column compression (I first heard about this thanks to Fujitsu's fantastic blog post)
- `PARTITION` our `posts` table

Performance

OK, time to get on with it, let's see how it performs with an `hstore`:

```
┌─────────┬──────┬──────┬───────┬──────┬─────
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg
├─────────┼──────┼──────┼───────┼──────┼─────
│ Latency │ 0 ms │ 2 ms │ 5 ms  │ 6 ms │ 2.15
└─────────┴──────┴──────┴───────┴──────┴─────

┌───────────┬─────────┬─────────┬─────────┬─
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 9
├───────────┼─────────┼─────────┼─────────┼─
│ Req/Sec   │ 287     │ 305     │ 348     │ 5
├───────────┼─────────┼─────────┼─────────┼─
│ Bytes/Sec │ 53.9 kB │ 56.9 kB │ 64.5 kB │ 9
└───────────┴─────────┴─────────┴─────────┴─

Req/Bytes counts sampled once per second.
# of samples: 60

┌────────────┬──────────────┐
│ Percentile │ Latency (ms) │
├────────────┼──────────────┤
│ 0.001      │ 0            │
├────────────┼──────────────┤
│ 0.01       │ 0            │
├────────────┼──────────────┤
│ 0.1        │ 0            │
├────────────┼──────────────┤
│ 1          │ 0            │
├────────────┼──────────────┤
│ 2.5        │ 0            │
├────────────┼──────────────┤
│ 10         │ 0            │
```

```
├─────────────┼───────────────┤
│ 25          │ 1             │
├─────────────┼───────────────┤
│ 50          │ 2             │
├─────────────┼───────────────┤
│ 75          │ 3             │
├─────────────┼───────────────┤
│ 90          │ 5             │
├─────────────┼───────────────┤
│ 97.5        │ 5             │
├─────────────┼───────────────┤
│ 99          │ 6             │
├─────────────┼───────────────┤
│ 99.9        │ 9             │
├─────────────┼───────────────┤
│ 99.99       │ 9             │
├─────────────┼───────────────┤
│ 99.999      │ 16            │
└─────────────┴───────────────┘

22k requests in 60.02s, 4.1 MB read
```

Not too far off! While we didn't try the pathological case(s) of millions of people liking the *same* post to hit breaking point, a slightly more random distribution seems to have done decently -- we actually have *lower* 99.999th percentile latency versus the simple counter.

An average of `2.15ms` versus `2.05ms` with the simpler counter is a ~4% increase in the average latency (though of course, the p99.999 is lower!).

Try #3: An Association table for remembering who liked what

A likely requirement from the original scenario that we've completely ignored is remembering *which* users liked a certain post to. The easiest solution here is an "associative" table like this one:

In SQL:

hideCopy

```
begin;

create table posts_seen_by_users (
  post_id bigint references posts (id),
  user_id bigint references users (id),
  seen_count bigint not null default 0 check
  primary key (post_id, user_id)
);

commit;
```

Caveats

In production, you're going to want to do a few things to make this even remotely reasonable long term:

- `PARTITION` the table (consider using partition-friendly `pg_partman`)
- Move old partitions off to slower/colder storage and maintain snapshots
- Summarize older content that might be seen lots

Consider a partitioning key up front -- post IDs are probably a reasonable thing to use if they're sufficiently randomly distributed

These are good initial stop-gaps, but a realistic setup will have many problems and many more solutions to be discovered.

(It will be a recurring theme but this is a spot where *we probably don't necessarily want to use stock Postgres* but instead want to use tools like Citus Columnar Storage, ZedStore, or an external choice like ClickHouse).

Performance

Alright, enough dilly dally, let's run our test bench against this setup:

```
┌─────────┬──────┬──────┬───────┬──────┬─────
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg
├─────────┼──────┼──────┼───────┼──────┼─────
│ Latency │ 0 ms │ 2 ms │ 8 ms  │ 8 ms │ 2.5
└─────────┴──────┴──────┴───────┴──────┴─────
┌───────────┬─────────┬─────────┬─────────┬──
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 9
├───────────┼─────────┼─────────┼─────────┼──
│ Req/Sec   │ 238     │ 254     │ 321     │ 4
├───────────┼─────────┼─────────┼─────────┼──
│ Bytes/Sec │ 43.4 kB │ 46.3 kB │ 58.5 kB │ 8
└───────────┴─────────┴─────────┴─────────┴──

Req/Bytes counts sampled once per second.
# of samples: 60

┌────────────┬──────────────┐
│ Percentile │ Latency (ms) │
├────────────┼──────────────┤
│ 0.001      │ 0            │
├────────────┼──────────────┤
│ 0.01       │ 0            │
├────────────┼──────────────┤
│ 0.1        │ 0            │
├────────────┼──────────────┤
│ 1          │ 0            │
├────────────┼──────────────┤
│ 2.5        │ 0            │
├────────────┼──────────────┤
│ 10         │ 0            │
├────────────┼──────────────┤
```

```
│ 25          │ 1             │
├─────────────┼───────────────┤
│ 50          │ 2             │
├─────────────┼───────────────┤
│ 75          │ 4             │
├─────────────┼───────────────┤
│ 90          │ 7             │
├─────────────┼───────────────┤
│ 97.5        │ 8             │
├─────────────┼───────────────┤
│ 99          │ 8             │
├─────────────┼───────────────┤
│ 99.9        │ 11            │
├─────────────┼───────────────┤
│ 99.99       │ 25            │
├─────────────┼───────────────┤
│ 99.999      │ 30            │
└─────────────┴───────────────┘


20k requests in 60.02s, 3.57 MB read
```

A little bit more divergence here -- 99.999%ile latency @ 30 which is almost double what it was for simple-hstore.

Average is coming in at `2.50ms` which is 16% slower than simple-hstore and 21% slower than simple-counter.

Try #4: Getting a bit more serious: bringing out the HyperLogLog

We'll just draw the rest of the owl now.

What's HyperLogLog you ask? Well it's just a probabilistic data structure! Don't worry if you've never heard of it before, it's a reasonably advanced concept.

You may have heard of Bloom Filters and they're *somewhat* related but they're not quite a great fit for the problem we're solving since we want to know how many people have seen a particular post. Knowing whether one user has seen a particular post is useful too -- but not quite what we're solving for here (and we'd have to double-check our false positives anyway if we wanted to be absolutely sure).

HyperLogLog provides a probabilistic data structure that is good at counting *distinct* entries, so that means that the count *will not* be exact, but be reasonably close (depending on how we tune). We won't have false positives (like with a bloom filter) -- we'll have a degree of error (i.e. the actual count may be 1000, but the HLL reports 1004).

We have to take this into account on the UI side but and maybe retrieve the full count if anyone ever *really* needs to know/view individual users that have seen the content, so we can fall back to our association table there.

Given that every second there are about 6000 tweets on Twitter(!), this is probably one of the only solutions that could actually work at massive scale with the limitations we've placed on ourselves.

Here's what that looks like in SQL:

hideCopy

```sql
BEGIN;

CREATE EXTENSION IF NOT EXISTS hll;

ALTER TABLE posts ADD COLUMN seen_count_hll h
  NOT NULL DEFAULT hll_empty();

COMMENT ON COLUMN posts.seen_count_hll
  IS 'HyperLogLog storing user IDs';

COMMIT;
```

Here we need the `citus/postgresql-hll` extension, which is generously made (truly) open source by citusdata.

NOTE that we still have access to the association table -- and while we still insert rows into it, we can *drop* the primary key index, and simply update our HLL (and leave ourselves a note on when we last updated it).

Caveats

There's not much to add to this solution, as the heavy lifting is mostly done by `postgresql-hll`, but there's one big caveat:

This approach *will* need a custom Postgres image for this, since `hll` is not an official `contrib` module

There are also a few optimizations that are easy to imagine:

Batching inserts to the association table (storing them in some other medium in the meantime -- local disk, redis, etc)

Writing our association table entries in a completely different storage medium altogether (like object storage) and use Foreign Data Wrappers and `pg_cron` and delay or put off processing all together

Performance

The most complicated solution by far, let's see how it fares:

```
┌─────────┬──────┬──────┬───────┬──────┬─────
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg
├─────────┼──────┼──────┼───────┼──────┼─────
│ Latency │ 0 ms │ 2 ms │ 6 ms  │ 6 ms │ 2.28
└─────────┴──────┴──────┴───────┴──────┴─────
┌───────────┬─────────┬─────────┬─────────┬──
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 9
├───────────┼─────────┼─────────┼─────────┼──
│ Req/Sec   │ 272     │ 285     │ 351     │ 4
├───────────┼─────────┼─────────┼─────────┼──
│ Bytes/Sec │ 49.5 kB │ 51.9 kB │ 63.9 kB │ 8
└───────────┴─────────┴─────────┴─────────┴──

Req/Bytes counts sampled once per second.
# of samples: 60

┌────────────┬──────────────┐
│ Percentile │ Latency (ms) │
├────────────┼──────────────┤
│ 0.001      │ 0            │
├────────────┼──────────────┤
│ 0.01       │ 0            │
├────────────┼──────────────┤
│ 0.1        │ 0            │
├────────────┼──────────────┤
│ 1          │ 0            │
├────────────┼──────────────┤
```

```
│ 2.5         │ 0             │
├─────────────┼───────────────┤
│ 10          │ 0             │
├─────────────┼───────────────┤
│ 25          │ 1             │
├─────────────┼───────────────┤
│ 50          │ 2             │
├─────────────┼───────────────┤
│ 75          │ 4             │
├─────────────┼───────────────┤
│ 90          │ 6             │
├─────────────┼───────────────┤
│ 97.5        │ 6             │
├─────────────┼───────────────┤
│ 99          │ 6             │
├─────────────┼───────────────┤
│ 99.9        │ 9             │
├─────────────┼───────────────┤
│ 99.99       │ 28            │
├─────────────┼───────────────┤
│ 99.999      │ 59            │
└─────────────┴───────────────┘

21k requests in 60.03s, 3.86 MB read
```

Another somewhat nuanced degradation in performance -- while the 99.99%ile latency was nearly 2x higher, the average latency was actually *lower* than the assoc-table approach @ `2.28ms`.

The average latency on the HLL approach is 11% worse than simple-counter, 6% worse than simple-hstore, and *faster* than assoc-table alone, which is an improvement.

Oh, the other places we could go

One of the great things about Postgres is it's expansive ecosystem -- while Postgres may (and frankly *should not*) beat the perfect specialist tool for your use case, it often does an outstanding job in the general case.

Let's look into some more experiments that could be run -- maybe one day in the future we'll get some numbers behind these (community contributions are welcome!).

Incremental view maintenance powered by `pg_ivm`

If you haven't heard about `pg_ivm` it's an extension for handling Incremental View Maintenance -- updating `VIEW`s when underlying tables change.

IVM is a hotly requested feature whenever views (particularly materialized views) are mentioned, so there has been much fanfare to it's release.

There are a couple advantages we could gain by using `pg_ivm`:

- Ability to time constrain calculations (newer posts which are more likely to be seen can exist in instant-access views)
- We could theoretically remove the complicated nature of the HLL all together by using `COUNT` with IVM

`pg_ivm` is quite new and cutting edge but looks to be a great solution -- it's worth giving a shot someday.

Doing graph computations with AGE

As is usually the case in academia and practice, we can make our problem drastically easier by simply changing the data structures we use to model our problem!

One such reconfiguration would be storing the information as a graph:

As you might imagine, finding the number of "seen-by" relations would simply be counting the number of edges out of one of the nodes!

Well, the Postgres ecosystem has us covered here too! AGE is an extension that allows you to perform graph related queries in Postgres.

We won't pursue it in this post but it would be a great way to model this problem as well -- thanks to the extensibility of Postgres, this data could live right next to our normal relational data as well.

So what's the best way to do it?

OK, so what's the answer at the end of the day? What's the best way to get to that useful v1? Here are the numbers:

In tabular form:

Approach	Avg (ms)	99%ile (ms)	99.999%ile (ms)
simple-counter	2.03	6	23
simple-hstore	2.15	6	16
assoc-table	2.5	8	30
hll	2.16	7	27

If we go strictly with the data, the best way *looks* to be the `hstore`-powered solution, but I think the HLL is probably the right choice.

The HLL results were quite variable -- some runs were faster than others, so I've taken the best of 3 runs.

Even though the data says `hstore`, knowing that posts will be seen by more and more people over time, I *might* choose the HLL solution for an actual implementation. It's far less likely to pose a bloated row problem, and it has the absolute correctness (and later recall) of the assoc-table solution, while performing better over all (as you can imagine, no need to `COUNT` rows).

Another benefit of the HLL solution is that PostgreSQL tablespaces allow us to put the association table on a different, slower storage mechanism, and keep our `posts` table fast. Arguably in a real system we might have the HLL in something like `redis` but for a v1, it looks like Postgres does quite well!

Wrap-up

I hope you enjoyed this look down the trunk hole, and you've got an idea of how to implement solutions to surprisingly complex problems like this one with Postgres.

As usual, Postgres has the tools to solve the problem *reasonably* well (if not completely) before you reach out for more complicated/standalone solutions.

See any problems with the code, solutions that haven't been tried? -- reach out, or open an issue!

More Postgres resources

- Partial data dumps using Postgres Row Level Security
- Postgres Views
- Postgres Auditing in 150 lines of SQL
- Cracking PostgreSQL Interview Questions
- What are PostgreSQL Templates?
- Realtime Postgres RLS on Supabase

Share this article

Last post

Supabase Flutter SDK 1.0 Developer Preview
2 August 2022

Next post

Revamped Auth Helpers for Supabase (with SvelteKit support)
13 July 2022

Related articles

Supabase Beta May 2023

Supabase Vecs: a vector client for Postgres

Flutter Hackathon Winners

ChatGPT plugins now support Postgres & Supabase

Building ChatGPT Plugins with Supabase Edge Runtime

View all posts

Build in a weekend, scale to millions

Start your project

Open Source

SupaSquad

DevTo

RSS

Company

Terms of Service

Privacy Policy

Acceptable Use Policy

Support Policy

Service Level Agreement

Humans.txt

Lawyers.txt

Security.txt