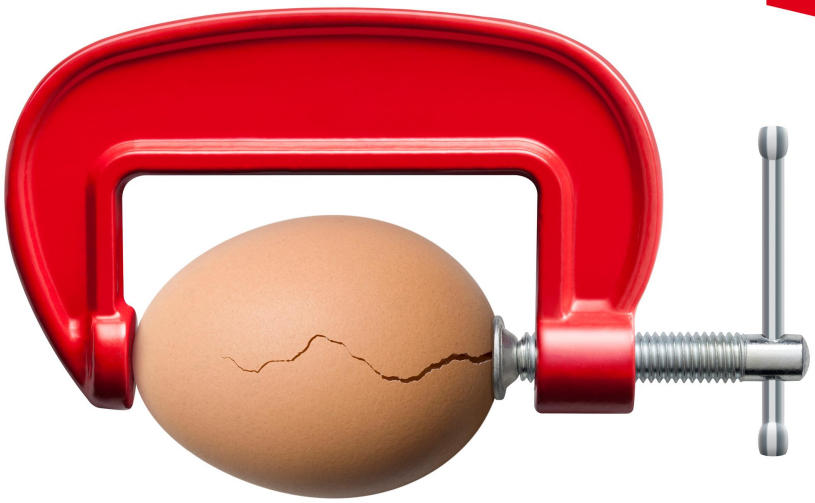
The Pragmatic Programmers

BETA

More SQL Antipatterns



Bill Karwin

edited by
Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Your pager suddenly begins blaring, which means one of the alerts on the database server has been triggered. Checking the infrastructure dashboard, you see that the query traffic has suddenly doubled, spiking to 35,000 queries per second. You didn't know it could go that high on this server.

"Did anyone deploy a code change that could increase the query rate?" you ask quickly in the developers' chatroom.

A few minutes pass.

Finally a curt reply appears from Stas, the most sullen and uncommunicative member of the developer team. "No code change."

"Well, someone changed something," you continue.

"I doubled the number of app instances," Stas writes.

The architecture of Visage has one main database instance, but 15 application servers—now 30 application servers—each running an instance of the same code. The applications handle the web requests in parallel, but they all connect to the same main database. This has worked well enough up until now, but every database has its limits.

"How did that double the rate of queries?" you ask. "I understand it increases the capacity for handling traffic, but it shouldn't cause an immediate doubling of query traffic. I'd expect it to increase gradually."

Stas responds only with an emoji of a man shrugging his shoulders.

You investigate the query logs of the database server, and discover that almost all of the queries are the same, something like the following:

```
SELECT * FROM post
WHERE updated_at > NOW() - INTERVAL '1' MINUTE
  AND notified_at IS NULL
```

This query is accounting for more than 97% of the load on the database—sometimes many instances of the query are running concurrently.

Why is one query being run a thousand times per second by every application instance?

Objective: Notify of Changes to Data

It's fruitless to get Stas to explain, but eventually by digging through git commit logs and long-forgotten software design documents, you find out the purpose of the mysterious query.

When posts are written in an interactive web site like Visage, users want to be notified. But as the number of users increases, sending notifications gets more costly. No one wants to wait for notifications to go out when they are simply saving a post.

So the developers of Visage decided to process notifications asynchronously. That way, the user saving their post immediately sees that it's successful, while the Visage backend application begins sending notifications to other users about the new content.

Of course, the other part of this objective is that notifications are timely. The backend code needs to know as quickly as possible when it's time to notify users, and it needs to scale up to send notifications to all users promptly.

That sets the stage for a number of problems.

Antipattern: Polling for Changes

As you read through the history of the code revisions, you piece together how earlier developers like Stas designed the back-end code to query to check for new posts, then notify users of the new content.

Back-End Code Version 1

The following is how the first version of this code looked.

Polling/anti/backend-polling-v1.py

```
while True:

    with conn:

        with conn.cursor() as cur:

            cur.execute("""
                SELECT post_id FROM post
                WHERE updated_at > NOW() - INTERVAL '1' MINUTE
                  AND notified_at IS NULL
            """)

            posts = cur.fetchall()

            for post in posts:
                post_id = post[0]

                process_notifications(post_id)

                cur.execute("""
                    UPDATE post SET notified_at = NOW()
                    WHERE post_id = %(post_id)s
                """, {'post_id': post_id})
```

It queries for posts for which it hasn't sent notifications (notified_at IS NULL), and for each one, calls a function which you can assume sends notifications to the users who want them. Then it updates notified_at to the current timestamp, so the next time it checks, it'll skip that post.

Once the notifications have been sent and the posts updated, it's time to check again, so the loop iterates to the top. This is called *polling* the database.

Now you see why the query you observed runs so frequently. Suppose the query itself is well optimized and takes only 10 milliseconds. The time for the loop code is negligible. So even if no one is writing new posts in Visage, the loop checking for new posts will iterate as much as 100 times per second. It can't slow down, because when someone eventually does write a new post, the requirement is to send out notifications with as little delay as possible.

Back-End Code Version 2

The next revision of the code is almost identical, except it adds a line to the end of the loop to sleep for 60 seconds, to delay before the next query runs. This limited the load caused by the queries checking for unprocessed posts.

Polling/anti/backend-polling-v2.py

```
time.sleep(60)
```

But the next day, the line was changed to a low duration:

Polling/anti/backend-polling-v2.py

```
time.sleep(1)
```

And then increased again:

Polling/anti/backend-polling-v2.py

```
time.sleep(10)
```

And finally changed to a line that reads the sleep duration value from an environment variable. Subsequently the duration could have been changed any number of times without changing code.

Polling/anti/backend-polling-v2.py

```
if "VISAGE_CHECK_DELAY" in os.environ:
    time.sleep(int(os.environ["VISAGE_CHECK_DELAY"]))
```

What's going on? You imagine there must have been numerous discussions about the best value for the sleep. Querying too frequently caused high load on the database. Querying too infrequently caused users to be delayed in getting their notifications. No value could solve both problems.

Back-End Code Version 3

There was a final change to the code. As the popularity of Visage kept increasing, more posts were written and more users wanted to be notified. A single function looping over posts and sending notifications serially couldn't keep up. So the code was refactored so that multiple back-end threads could work in parallel.

Polling/anti/backend-polling-v3.py

```
while True:

    with conn:

        with conn.cursor() as cur:

            cur.execute("""
                SELECT post_id FROM post
                WHERE updated_at > NOW() - INTERVAL '1' MINUTE
                  AND notified_at IS NULL
                FOR NO KEY UPDATE
                SKIP LOCKED
                LIMIT 1
            """)

            post = cur.fetchone()

            if post is not None:
                post_id = post[0]

                process_notifications(post_id)

                cur.execute("""
                    UPDATE post SET notified_at = NOW()
                    WHERE post_id = %(post_id)s
                """, {'post_id': post_id})
```

The query in the preceding code looks different in the following ways:

- The FOR NO KEY UPDATE clause means it locks rows that it examines. The NO KEY part is a specific feature of PostgreSQL that allows insertion of child rows that reference the locked row. For example, another session inserting a new row in comment would have to wait if it references a row in post locked with FOR UPDATE, but it would not wait if the row in post were locked with FOR NO KEY UPDATE.
- The SKIP LOCKED clause means if the query tries to examine a row and can't immediately lock it because it's already locked by another instance, the query will skip the row and move on to examine the next row, until it finds a row that isn't locked. This avoids any delay from a lock-wait.
- The LIMIT 1 clause means the query stops examining rows after it successfully reads and locks one row.

This modified query allows multiple instances to run concurrently. Which instance reserves a given row doesn't matter; what matters is that no row is read by more than one instance. Together this design is effectively like a *queue*, in that each row is processed once by one of the back-end instances. After sending notifications for the locked row, the code commits the transaction, and committing automatically releases the lock.

If new posts are being added faster than the back-end instances can keep up with, then more instances can be started. They're simply Python scripts, and any number of them can be run on a given application server. You find that the current application server starts ten instances of the new post checker script.

Again, this explains why you saw the query run so frequently, and why you sometimes saw many of the same query running concurrently. As more and more back-end instances are added to handle the traffic, they add more database sessions running the same query. Each server is configured to run ten such instances, and Stas deployed 15 additional application servers. Now there's a total of 300 back-end instances over 30 servers, each running a loop to check repeatedly for new posts. The default delay in the loop is zero, so every one of the instances is looping as fast as it can.

Even if the query is well optimized, it's no longer a mystery why it accounts for 97% of the query traffic.

You still have the problem of how to reduce that load, because the high traffic at this level is consuming so much processing power on the database server that it's degrading performance for all queries—including the remaining 3% of queries which serve the other functionality for Visage.

Shortly you'll see how you can make notifications happen in a timely way, without using polling queries that overload the database.