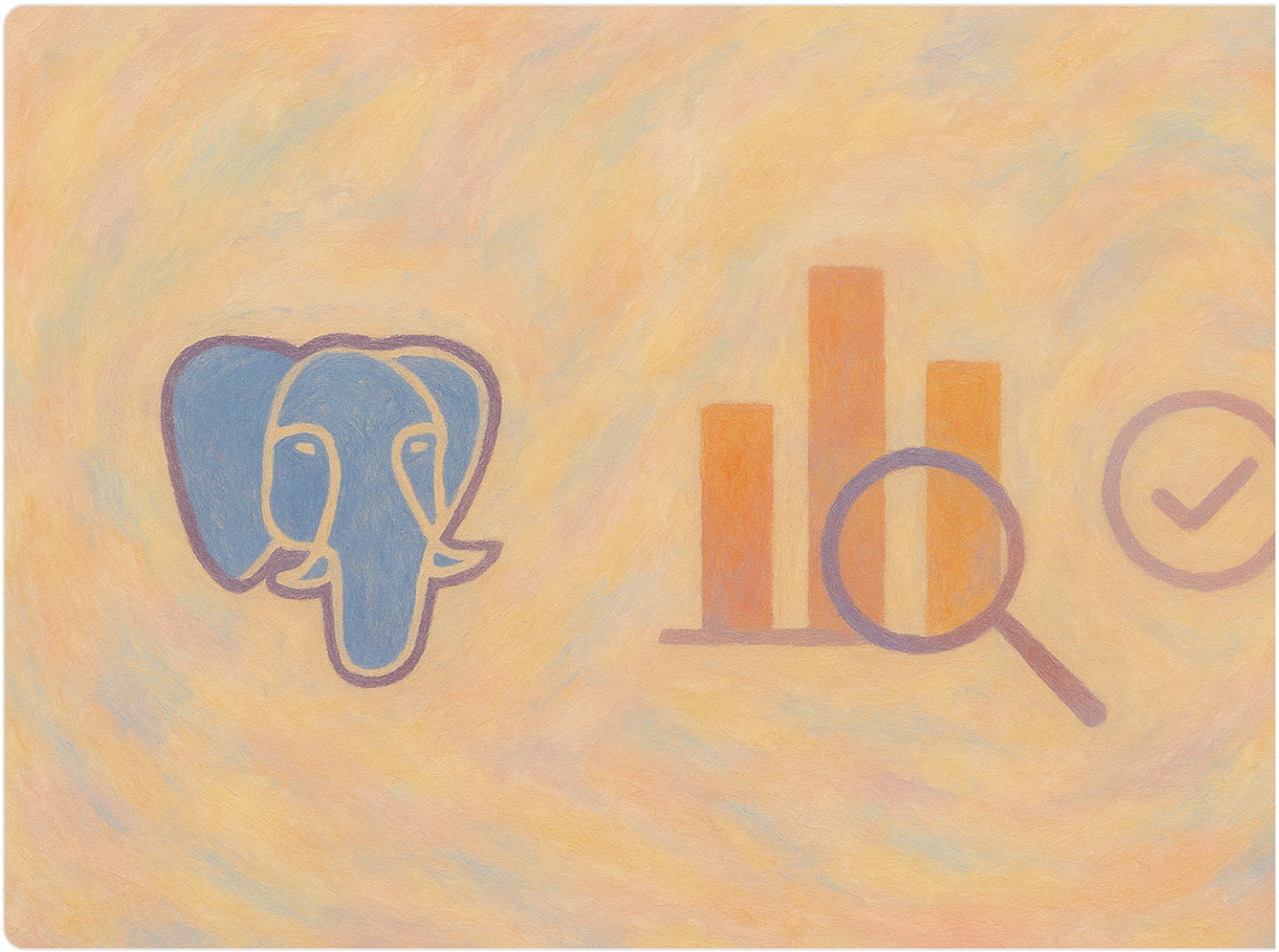
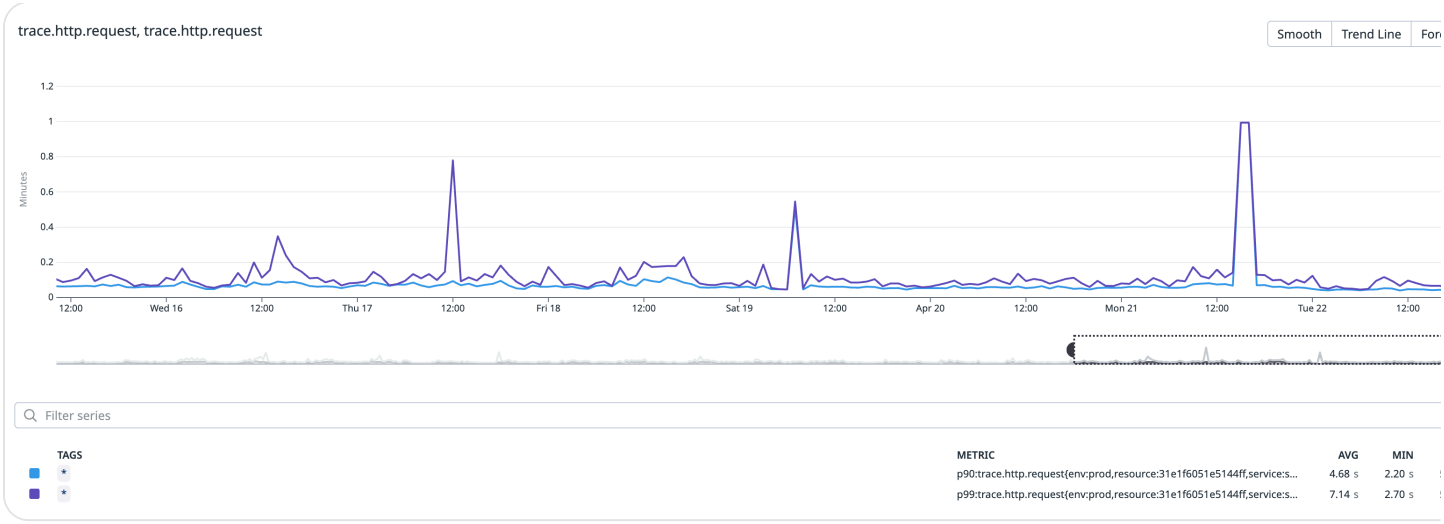
Slow queries, High DB CPU and others

Author: Martín San Juan



Context

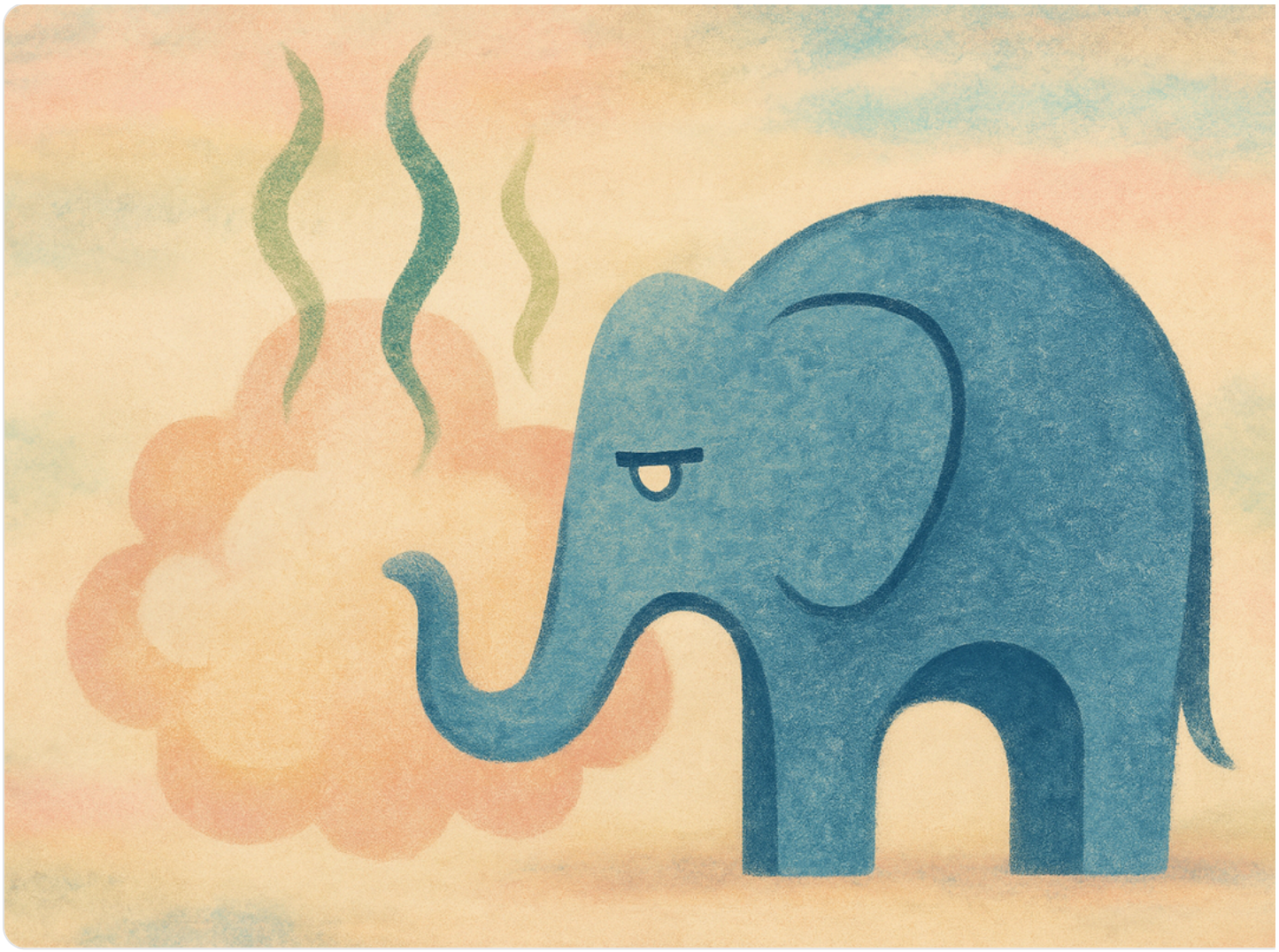
Since always, the queries used on the backoffice were really slow. We improved fixing some missing indexes, reducing the time dramatically (In order of 75%), but isn't good enough given the numbers.



The P90 and P99, were 4.68s and 7.14s average in the last week with peaks of 60s (timeouts).

It was only affecting backoffice's users, so we considered the improvement enough for the time, and we prioritize different things. Even though we've spotted the opportunity and the potential issue.

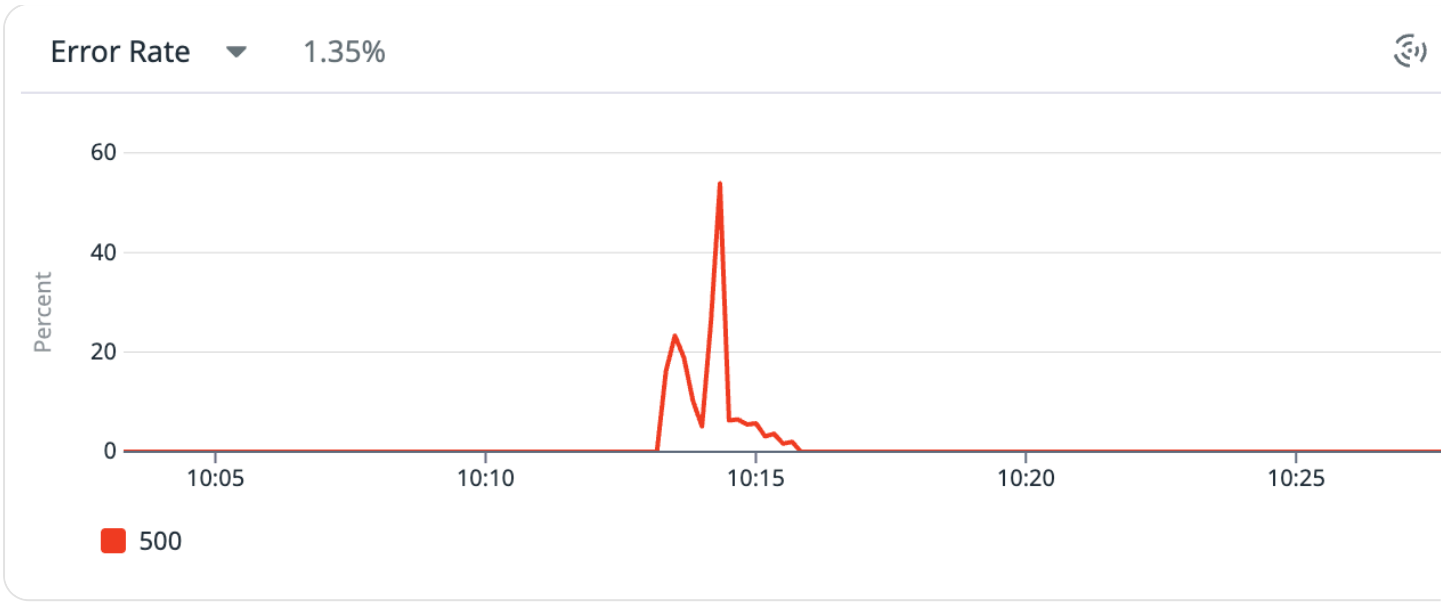
When the smell stinks

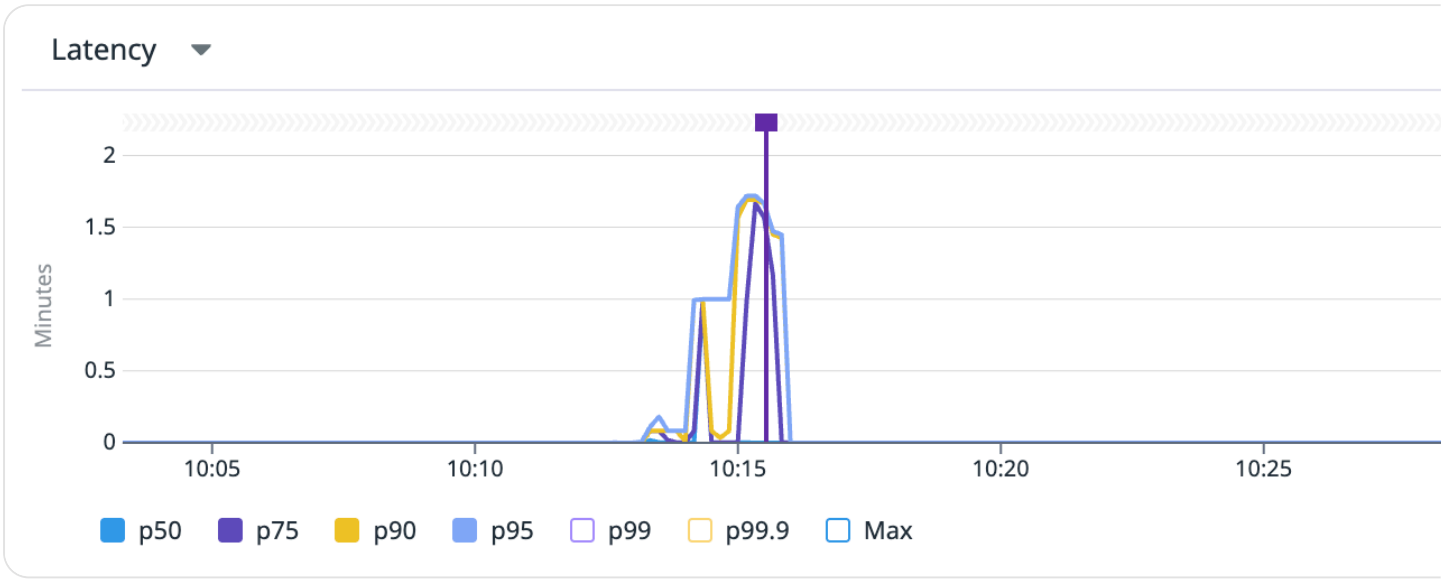


The slow queries are the equivalent of code smells. At the beginning, we have just a few, and that’s okay, but if we don’t clean enough, the smells start to stink, and most of the time, they are harder to work on than before.

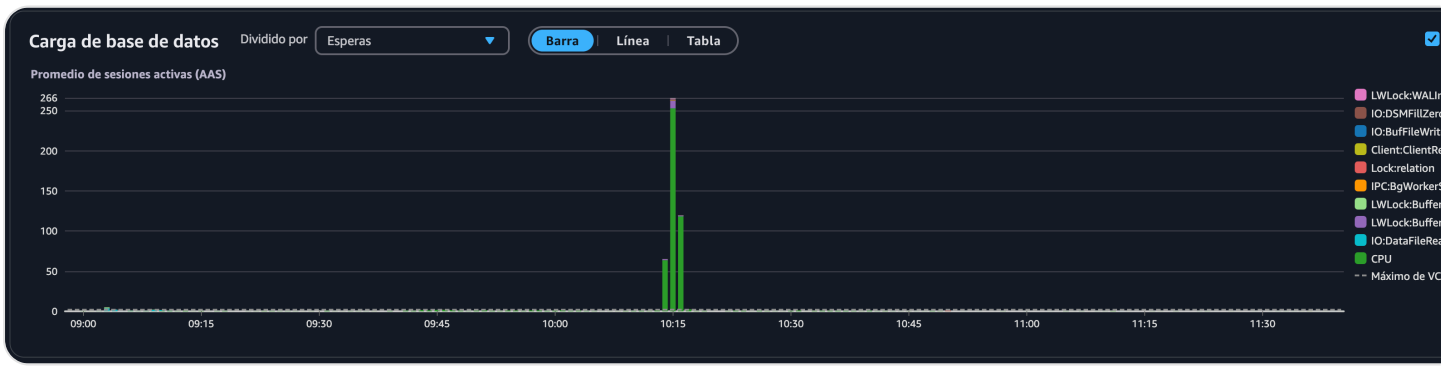
On databases, the problem is not to have more slow queries (which is also a problem), it's just to have it, because even if we do not change anything will start to stink just because of the natural growth of the data. More records, more stink.

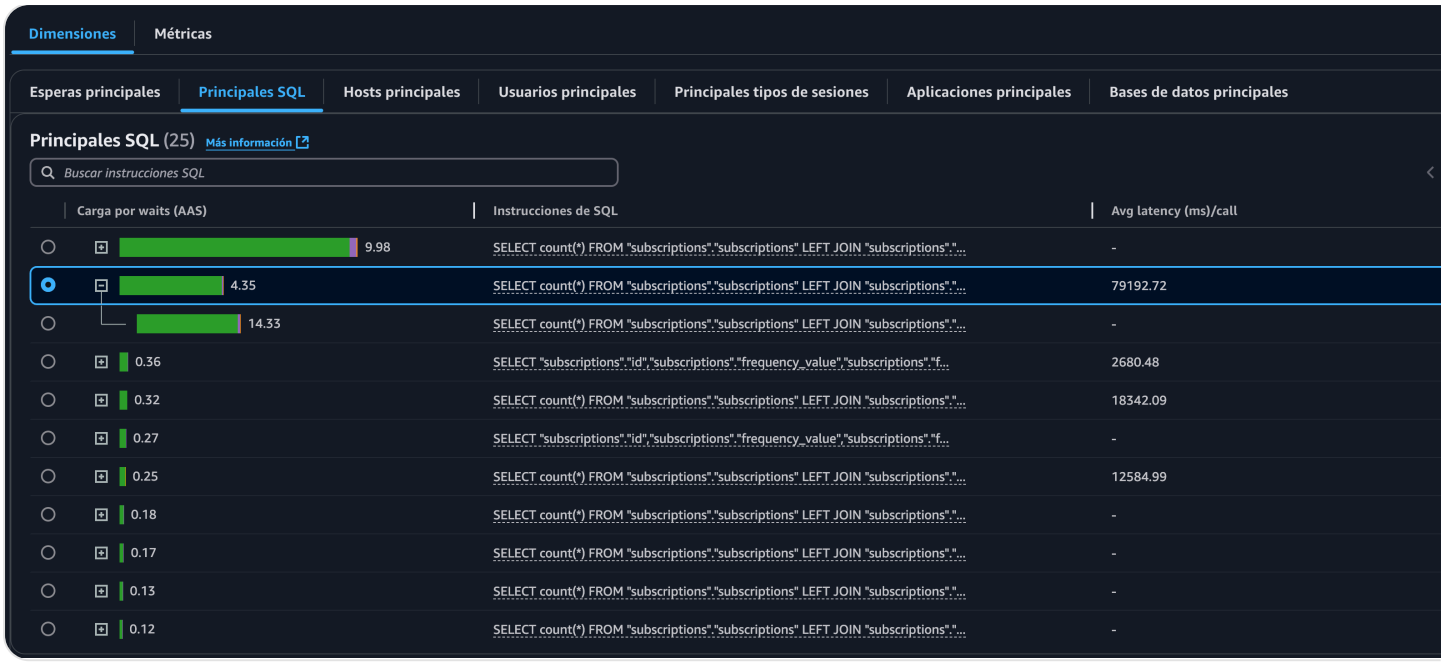
So, the stink struck like a spike of errors and latency on the API.



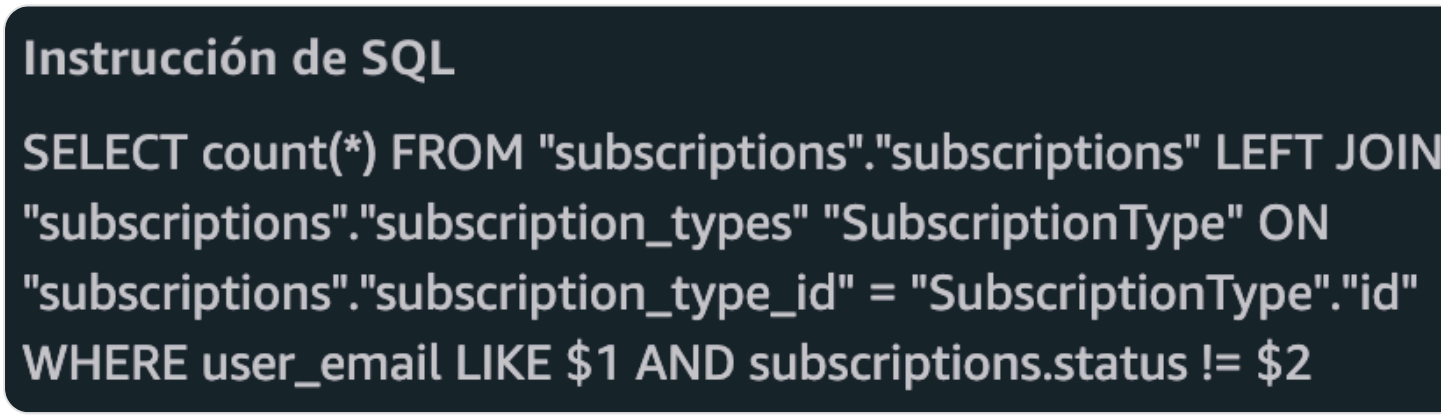


This was related to a spike in CPU usage of the database, caused by some innocent searches in the back office triggering a slow query.





The 💩

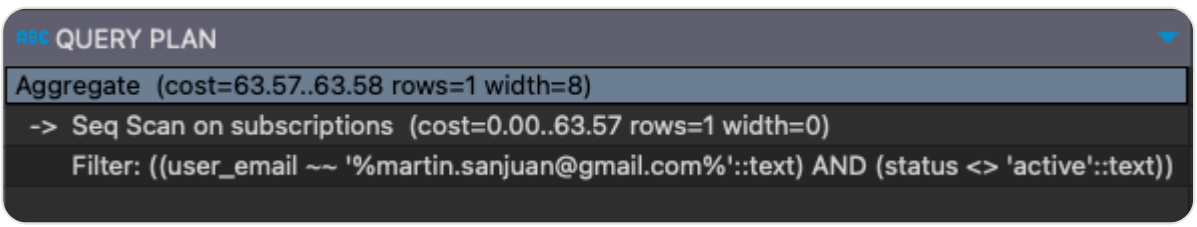


The problem and THE problem

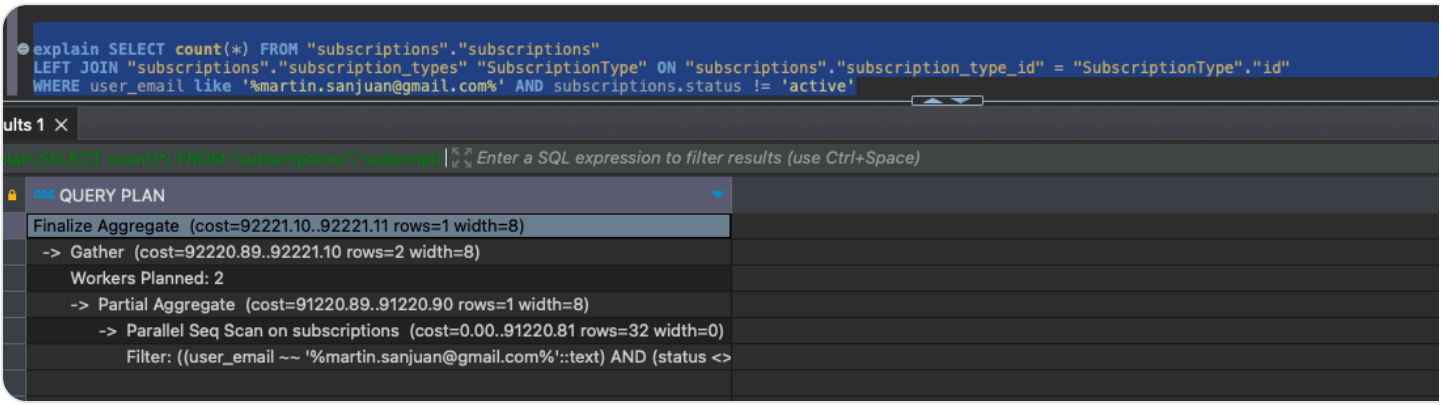
I want to split the problem in two parts, "**The problem**" and "**THE problem**"

The problem (and solution)

The problem in this query is kind of tricky if you are not familiar with some kind of indexes and functions. Here we are filtering by `user_email` and `status`, and both have indexes, so the query should be pretty straightforward... but... here the plan.



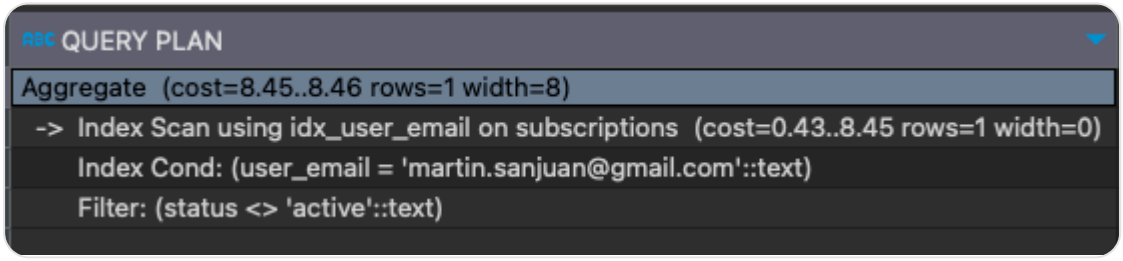
We are not using the index, and we are performing a Sequential Scan, which means we are walking through every row on the database, and the cost will grow bigger with the data. Here a productive example of today



The difference, 1.137 rows in develop vs 1.235.469 in production.

The reason about why we are not using the index it's because using LIKE `%value%` because uses a **B-tree index** which works fine with **prefix-based lookups.**

This is the plan of almost the same query but using `user_email = 'value'` instead of `user_email like '%value%'` where the index applies.



The query is **approximately 10,920 times more expensive** than the second one.

To solve the problem without changing the query itself, because that can lead to some functionality problems and a bigger development effort, we decided to use `pg_trgm` extension.

pg_trgm it's a **PostgreSQL extension** that enables `fast fuzzy text search and similarity matching` using **trigrams**.

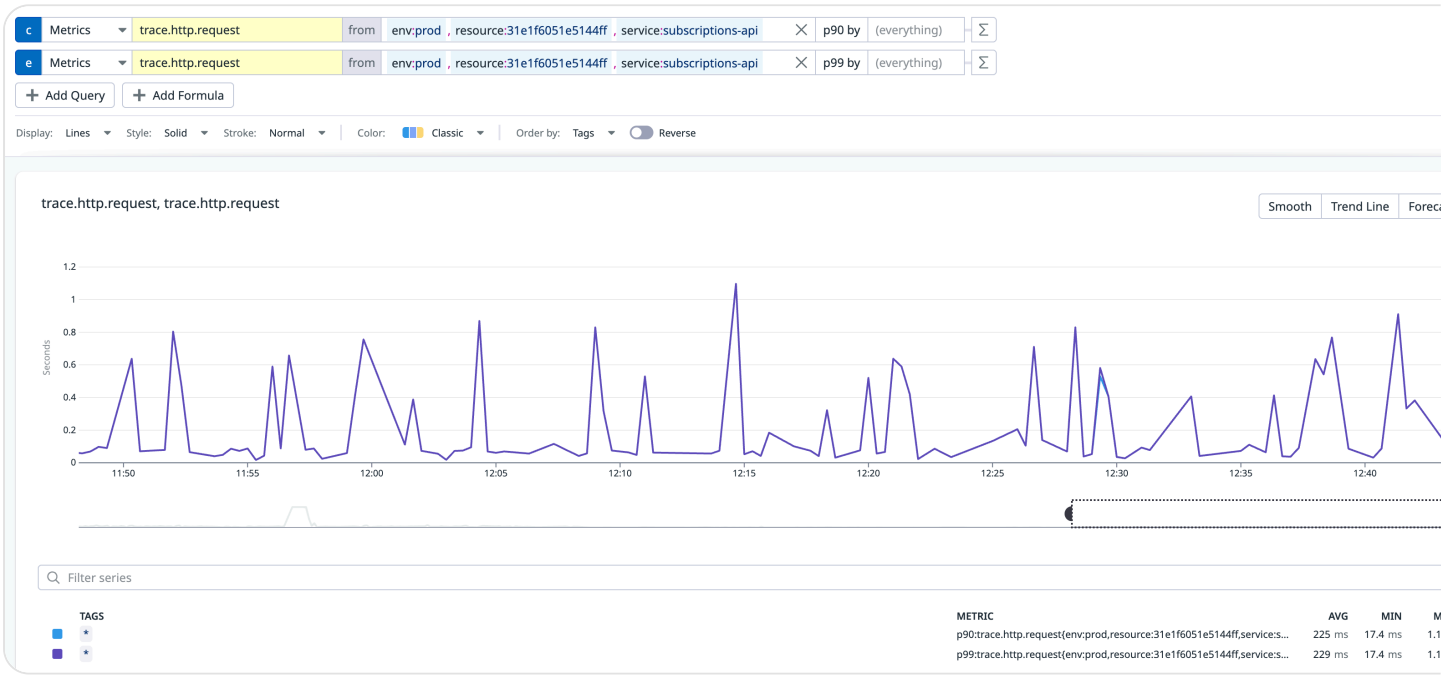
By installing this extension and creating the right GIN index, we reduce the cost of those queries like this:

```
QUERY PLAN
Aggregate  (cost=1400.67..1400.68 rows=1 width=8)
  ->  Bitmap Heap Scan on subscriptions  (cost=916.95..1400.48 rows=76 width=0)
        Recheck Cond: (user_email ~~ '%martin.sanjuan@gmail.com%'::text)
        Filter: (status <> 'active'::text)
        ->  Bitmap Index Scan on idx_user_email_trgm  (cost=0.00..916.93 rows=124 width=0)
              Index Cond: (user_email ~~ '%martin.sanjuan@gmail.com%'::text)
```

`Which means a 98.48% reduction of the query cost.`

And also represents a impact on the response time of the endpoints using this query moving us from the P90 and P99 of 4.68s and 7.14s mentioned before to a P90 and P99 of 225ms and 229ms after the change.

`Which means a reduction of 95.19% and 96.79% respectively.`



Now we fix the problem, we can focus on what it was **THE problem**.

THE Problem

Many years ago I read **The Wrong Abstraction by Sandi Metz,** and It stuck with me. It changed the way I write code and my thoughts about DRY, which were a must for me back in time.

Today, I was able to prove, one more time, the truth about the phrase.

> **Duplication is far cheaper than the wrong abstraction**

The query problem relies in this two functions and the assumptions (or abstractions) we did here.

```
func buildQueryString(subQueryKeys []string, subsQueryItems []interface{}, filters map[string]string) map[string][]inte

    var qry map[string][]interface{}

    if len(subQueryKeys) <= 0 {
        return qry
    }

    qryKeys := fmt.Sprintf("%s LIKE ?", strings.Join(subQueryKeys, " LIKE ? AND "))

    if _, found := filters["status"]; !found {
        qryKeys += " AND subscriptions.status != ?"
        subsQueryItems = append(subsQueryItems, "cancelled_due_order_fail")
    }

    qry = map[string][]interface{}{qryKeys: subsQueryItems}

    return qry
}
```

```go
func getQueryFilters(filters map[string]string) ([]string, []interface{}) {

    subsQueryItems := []interface{}{}
    subQueryKeys := []string{}
    for key, val := range filters {
        switch key {
        case "user_email":
            userEmail := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "user_email")
            subsQueryItems = append(subsQueryItems, "%"+userEmail+"%")
        case "subscription_type_name":
            subTypeName := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "name")
            subsQueryItems = append(subsQueryItems, "%"+subTypeName+"%")
        case "cancellation_reason":
            cancelReason := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "cancellation_reason")
            subsQueryItems = append(subsQueryItems, "%"+cancelReason+"%")
        case "order_id":
            orderID := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "triggering_order_id")
            subsQueryItems = append(subsQueryItems, "%"+orderID+"%")
        case "store":
            store := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "subscriptions.store")
            subsQueryItems = append(subsQueryItems, "%"+store+"%")
        case "vendor":
            vendor := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "subscriptions.vendor")
            subsQueryItems = append(subsQueryItems, "%"+vendor+"%")
        case "status":
            status := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "subscriptions.status")
            subsQueryItems = append(subsQueryItems, "%"+status+"%")
        case "price":
            price := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "subscriptions.price::text")
            subsQueryItems = append(subsQueryItems, "%"+price+"%")
        case "created_at":
            createdAt := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "subscriptions.created_at::text")
            subsQueryItems = append(subsQueryItems, "%"+createdAt+"%")
        case "updated_at":
            updatedAt := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "subscriptions.updated_at::text")
            subsQueryItems = append(subsQueryItems, "%"+updatedAt+"%")
        case "id":
            id := strings.ToLower(val)
            subQueryKeys = append(subQueryKeys, "subscriptions.id::text")
            subsQueryItems = append(subsQueryItems, "%"+id+"%")
        }
    }
    return subQueryKeys, subsQueryItems
}
```

Aside from the improvement opportunites with the big case and how we are mapping this, the REAL problem relies on the misconception that all the parameters should behave the same and we consider them all just as a string.

Our abstraction was, all the inputs are strings, so lets group all of them and build the query to accommodate on that.

But to get an abstraction we forgot a lot of things.

For example:

- All the parameters are strings, but it makes sense to look for partial data?
  - Is it useful to look for `mint` instead of `shapermint` ?
  - Do we want to provide that feature?
- It makes sense to compare a date using a LIKE
- Users will look require to `get all the emails starting with ma` or they will try to look for the exact match?

> We saw abstractions instead of "features" or use cases. It's like an premature optimization but for DRY, and wired for that.

In this scenario, the filters, are not just strings and can have their own business

- We don't want to use `LIKE %store%` because this filter will be a dropdown, so instead of that we want to use `= 'store'` , same for vendor.
- `triggering_orded_id` is a UUID, looking for something different than an exact match is quite usless.
- `created_at` and `updated_at` are dates, and we don't want to look by time, so using `updated_at::text LIKE '2024-01-01'` will return the same than `updated_at >= 2024-01-01 and updated_at < 2024-01-02` but using the index.

> Are all those strings really the same, or we are making a `Wrong Abstraction`?

Some Lessons Learned

- Not all the 💩 have the same value, and not always we can see the impact before it stinks. `Keep your house clean.`
- Making the wrong abstractions can hurt us more than some duplicated code, and you will not notice that until it's too late. Not everything needs to be DRY, but you can't also duplicate everything. `Keep the balance` ⚖️

- *More technical*: Try to avoid queries with functions like `LOWER`, `::date`, or unnecessary `LIKE` that force a sequential scan and can harm your queries and database metrics.

Appendix

Using `pg_trgm`

1. **Create the extension**

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
```

2. **Check it**

```
SELECT * FROM pg_extension WHERE extname = 'pg_trgm';
```

3. **Create the index**

```
CREATE INDEX idx_user_email_trgm ON subscriptions.subscriptions USING gin (user_email gin_trgm_ops);
```

4. **Collect the earnings** 💰