DATABASES *for* DEVELOPERS

how to scale your services,

save your data,

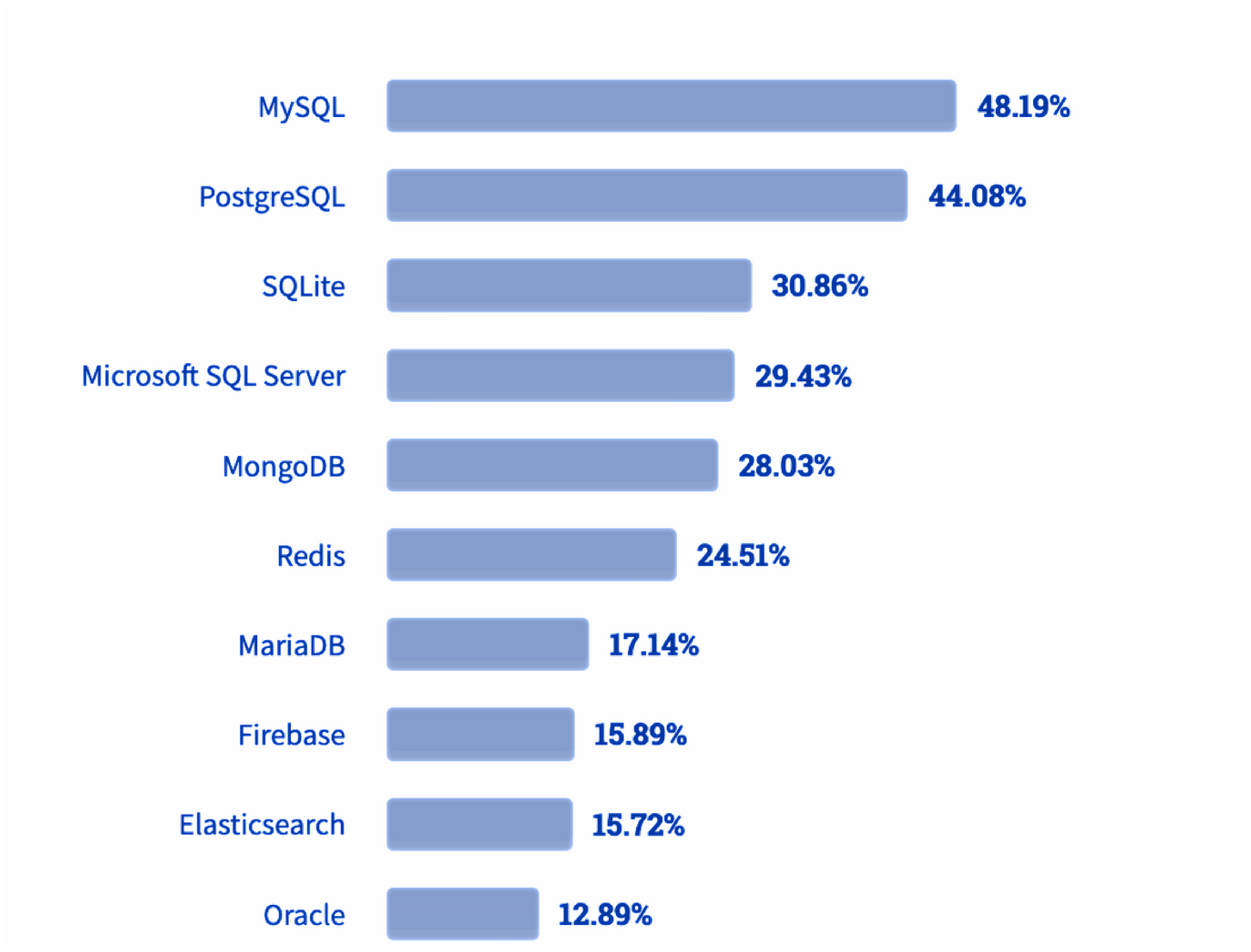
and tame elephants

IF YOU'RE A DEVELOPER,
YOU USE A DATABASE.

YOU MAY LOVE OR HATE IT.

how organizing all your code around your database performance can feel like

It's probably Postgres.



(source: StackOverflow Developer Survey 2021)

This book will teach you how to:

design,

optimize,

and monitor

your application's relational database.

Some examples are Postgres-specific.

Most will work broadly.

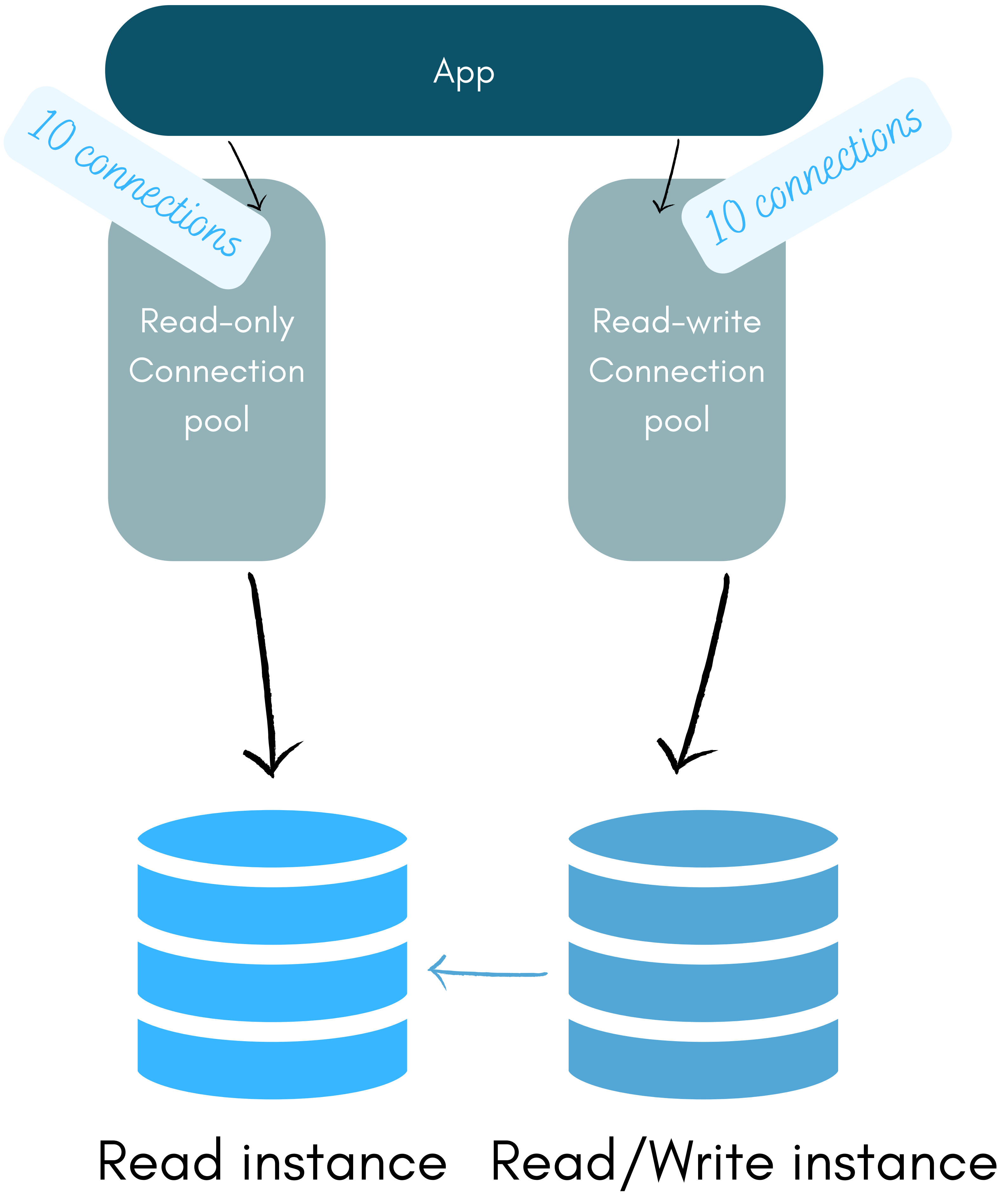
WITH PICTURES

FIRST: WHY DATABASES WORK IN CLUSTERS.

It's not because they're lonely.

In larger or cloud-based database deployments, you will want a cluster (= several server instances using the same data) to ensure availability, scalability, and in some cases cache optimization.

Each instance will have different access patterns, and may have different resource allocations.



In your application, you may use or see others using different connection pools for read and read/write workloads.

In most applications, the APIs used for writing have significantly different uses than the ones for reading. They may be:
less latency-sensitive;
used by services instead of humans;
limited to specific data;
primarily used at some times of day

When the work is very different, it makes sense to target it for different instances.

Here are some code examples.

Use targetServerType in the connection string:

Java

```
String url = "jdbc:postgresql://host/test?user=user&password=secret&ssl=true&targetServerType=primary";
Connection conn = DriverManager.getConnection(url);
```

Set up two databases in database.yml and use them like this:

Ruby/Rails

```
class ApplicationRecord < ActiveRecord::Base
   self.abstract_class = true
   connects_to database: { writing: :primary, reading: :primary_replica }
end
```

The pg module uses a connection string. Use a different hostname for the write instance:

Node.js

```
var pg = require('pg');
var cs = postgres://user:password@host:5432/db";
var csReader = postgres://user:password@host-ro:5432/db";
```

Unlike the JDBC or Rails example, this won't ensure that the "read-write" connection is to the primary server endpoint. Use a library if you want that behavior.

SECOND: WHY DATABASE CALLS "GET STUCK".

they're "too big"?

No?

When your database connection "gets stuck", what it really means is your application is waiting on it.

have you considered not doing that?

Here's what could be happening:

- you have run out of connections
- you are waiting on a transaction that takes too long to execute
- you are waiting on a locked resource
- the result set you are requesting takes too much memory
- your database query has already returned, but your own server code is taking a really long time to process it

Monitoring will tell you which. At any rate, your application should always time out long database queries.

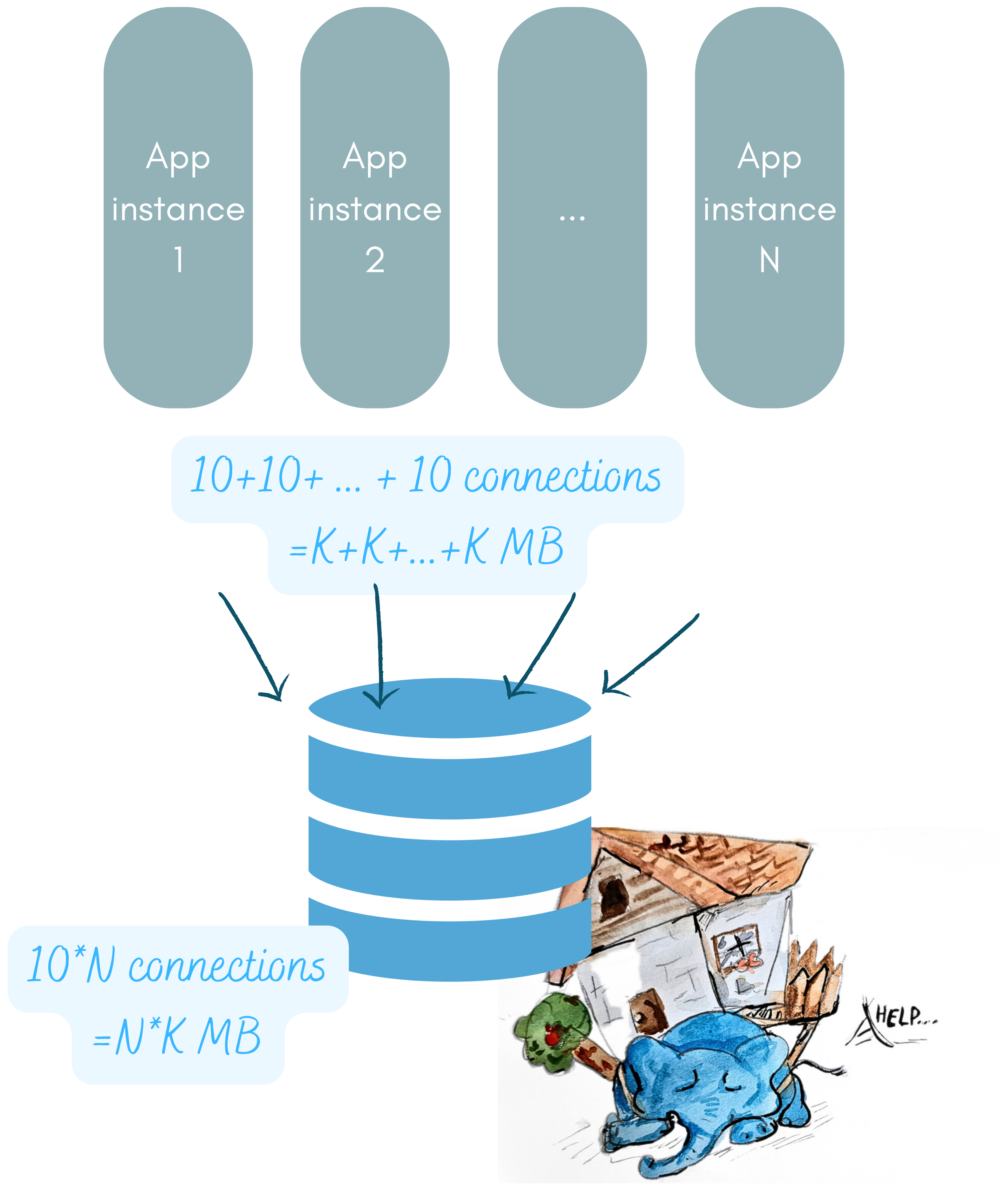
we'll see later how to identify long queries and memory issues!

let's see now how to deal with available connections and database locking...

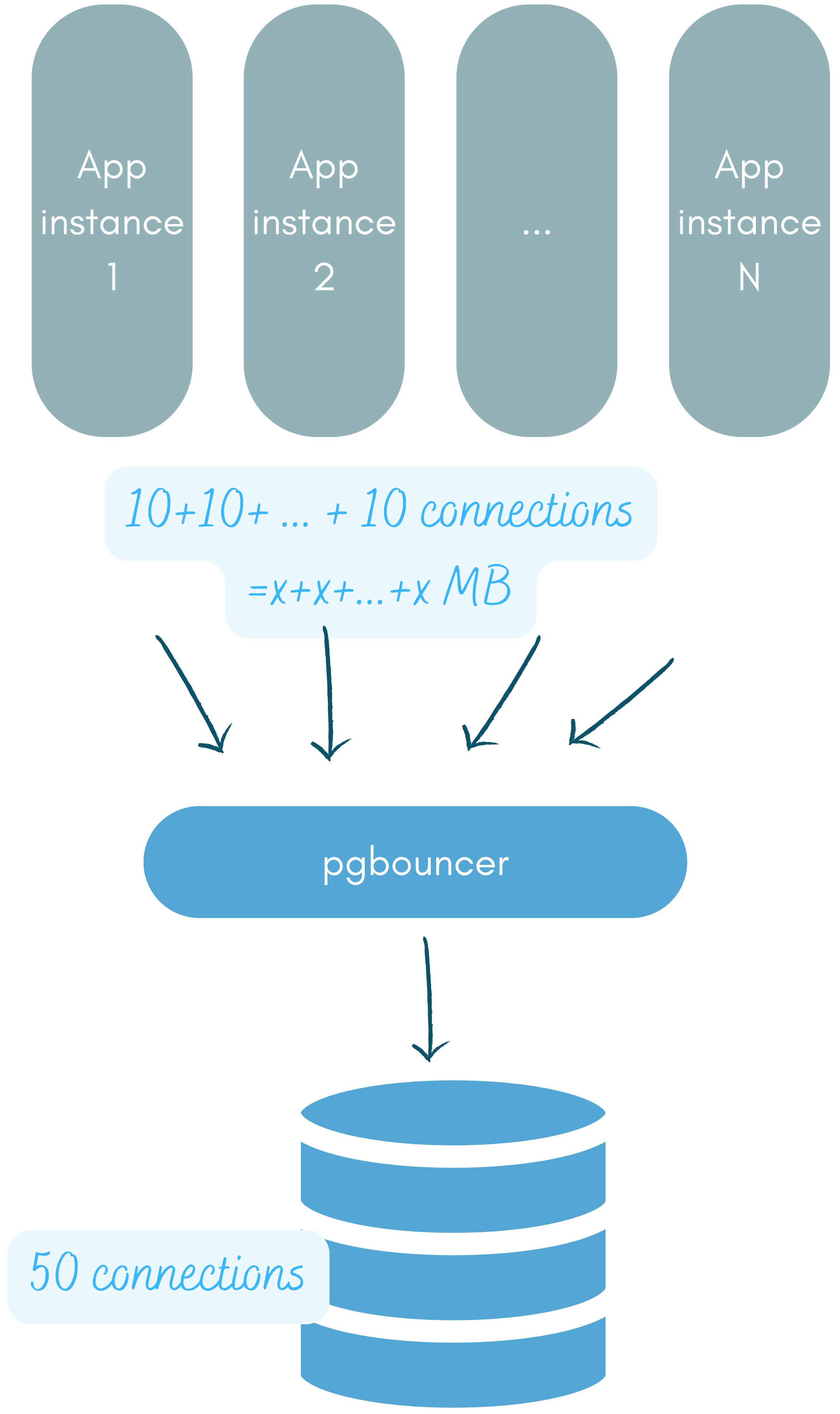
Each of your active database connections has a cost.

As a result, most database systems will default to a maximum number of concurrent connections. If your queries take too long, they can pile up until that number is hit.

Two options:
improve your query performance,
or use a connection pooler.



Relevant link: https://www.enterprisedb.com/postgres-tutorials/why-you-should-use-connection-pooling-when-setting-maxconnections-postgres



pgbouncer is an example of a connection pooler.

It does so by acting as a proxy server and reusing one of the connections in its pool to execute any new queries your services send it.

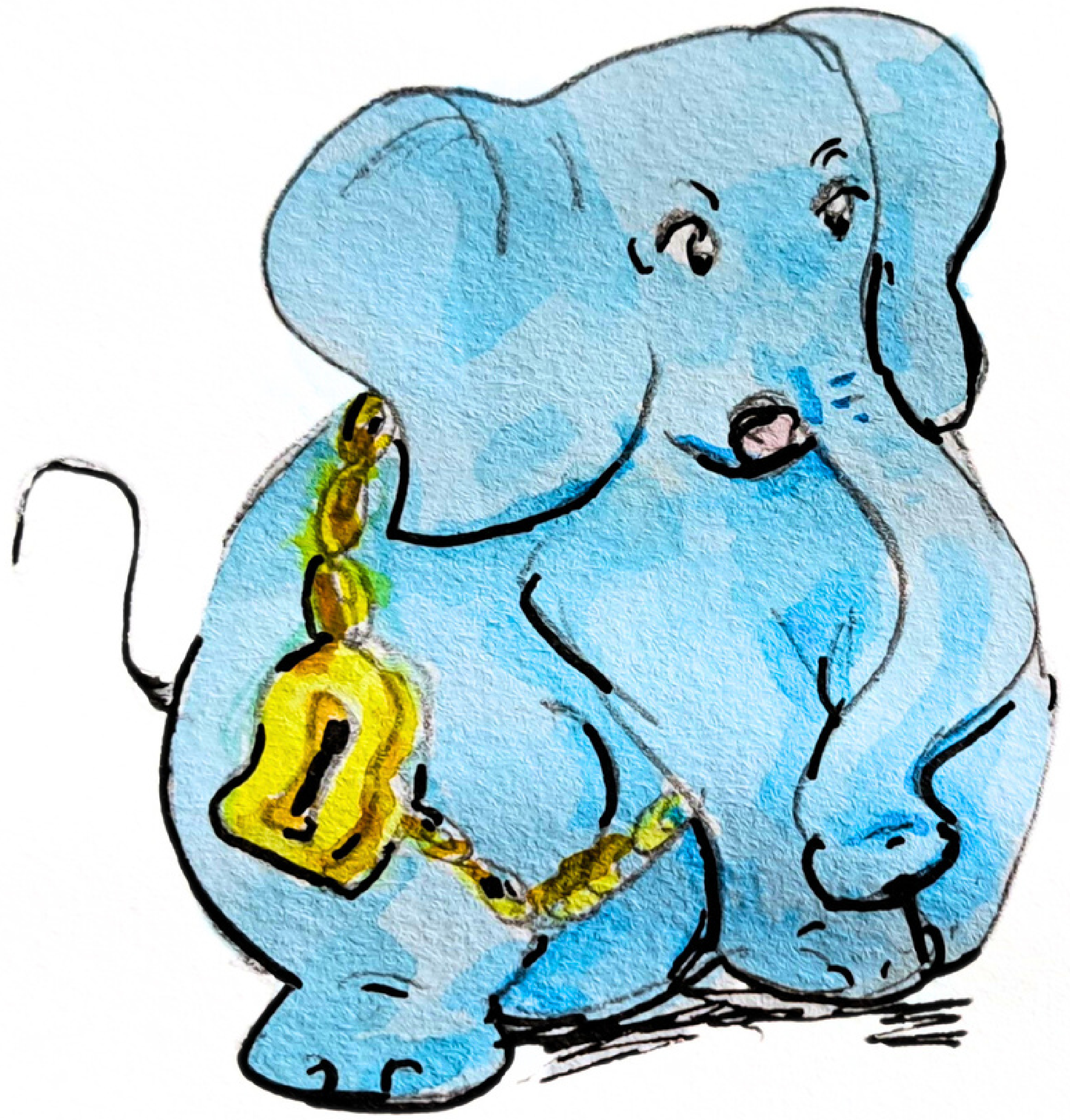
That way, as you add service instances, you do not need each new instance to use up more database connections.

Relevant link: http://www.pgbouncer.org/usage.html

Locking is something that can happen when several processes attempt to update the same table or field.

Database systems will acquire locks during certain operations. Most edit operations only lock the target row(s) (keep in mind a transaction can target many rows).

In rare cases, such as when running some forms of REINDEX or VACUUM, the transaction will lock the whole table. As a result, all other queries may be on hold a long time.



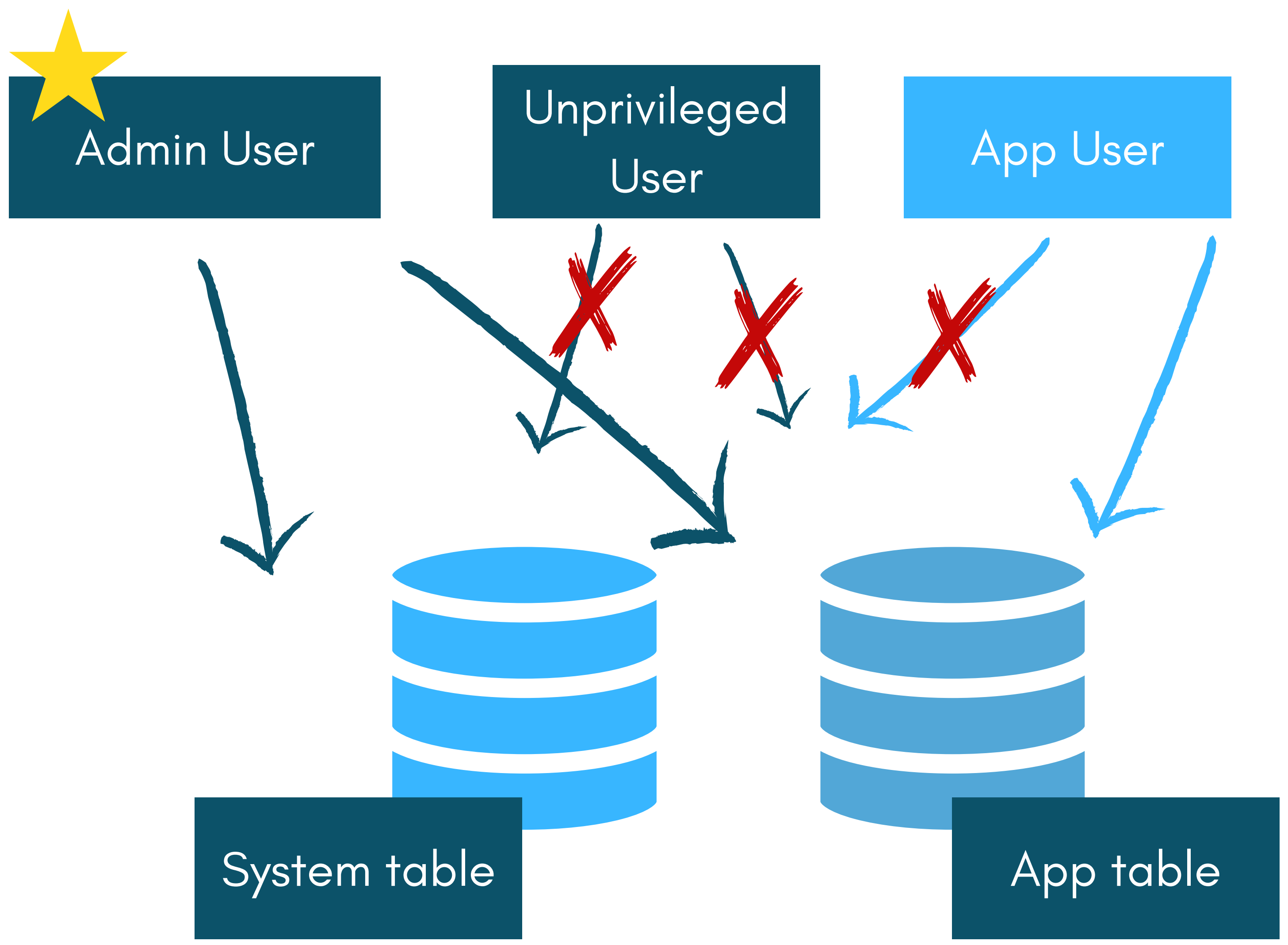
You can monitor which locks are currently active in your database.

In Postgres, this information is stored in a special view called pg_locks which stores information about lock objects and which processes hold them.

THIRD:
WHY SOME USERS ARE SUPER, AND SOME ARE NOBODY.

There are several tiers of users in Postgres (and all other databases). When you create your database server, you will have a default, admin user.

Your application should use a different user account, with permissions set up so that it can only see the appropriate tables.



As we've seen with pg_locks, databases often have "system" tables that contain super valuable information about the database itself.

The problem is it's also super valuable to attackers. For instance, it can contain query text for everything that executes on the same database!

As a result, app credentials (which are part of a "more fragile" threat surface) should not see them.

In SQL, GRANTs can apply to specific SQL statements. For instance, a given user may only do SELECTs, or some users can INSERT new data but not modify existing rows.

There are also default privileges, where a new user could be able to read all tables, including those that will be created in the future.

These should be used wisely and audited.

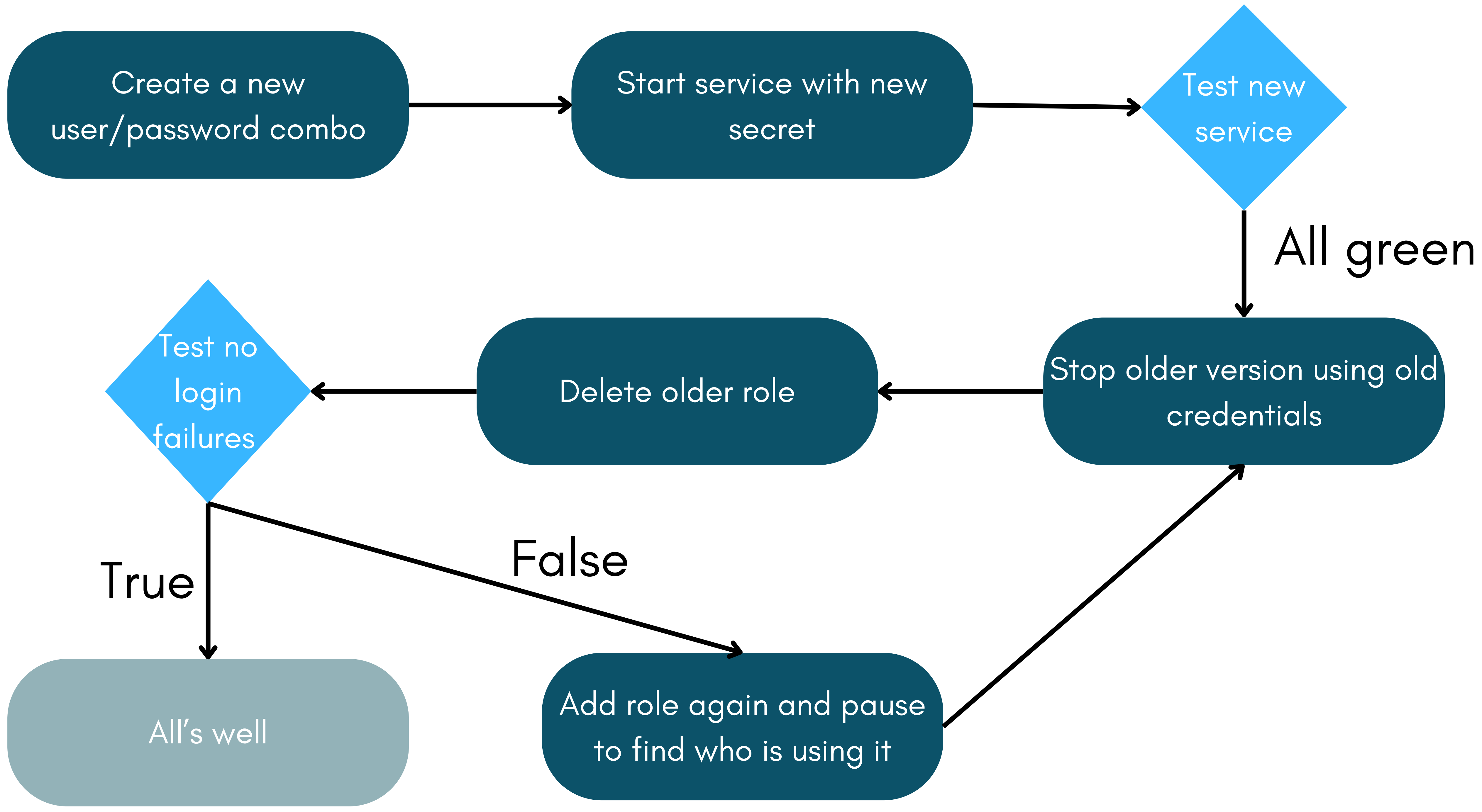
FOURTH:
WHY YOUR DATABASE PASSWORD DOESN'T BELONG IN YOUR CODE.

We now know your application should have its own credentials.

But even if they are unique to this service and well-restricted, they are secrets.

Humans should not use service credentials to access the database, and service credentials need to be managed and rotated by policies implemented in automated systems.

Example: how to automatically rotate secrets in your database and service



Create a new user/password combo
Start service with new secret
Test new service
All green
Stop older version using old credentials
Delete older role
Test no login failures
True
False
All's well
Add role again and pause to find who is using it

specific implementations exist using tools like Vault or SecretsManager (see link)

you can also set up ***temporary*** *database secrets!*

Relevant link: https://docs.aws.amazon.com/secretsmanager/latest/userguide/rotating-secrets.html

FIFTH:

WHY SOME QUERIES ARE HARDER THAN OTHERS.

We have talked about query times before. Some queries will execute a lot slower than others. Here are factors that can contribute:

- complex JOINs (costs CPU; generates large intermediate sets)
- large result sets
- sort operations over large sets; especially, sorting before filtering
- table scans over large tables
- using indexes with out of date statistics
- string operations

You can run the EXPLAIN ANALYZE command to obtain detailed information on query costs.

Let's look at some examples.

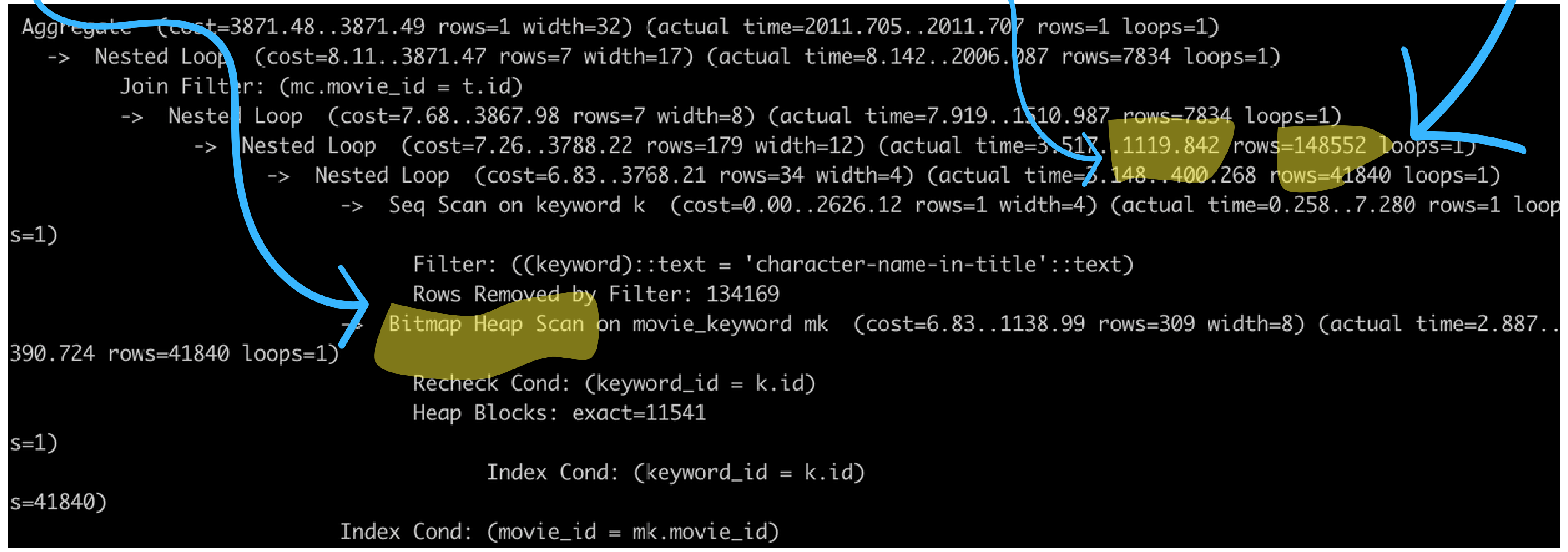
Note that the size of the set (number of rows) has a great effect on the cost.

Seeing the output of ANALYZE can take longer than the "real" query. Try to use it on small reproductions.

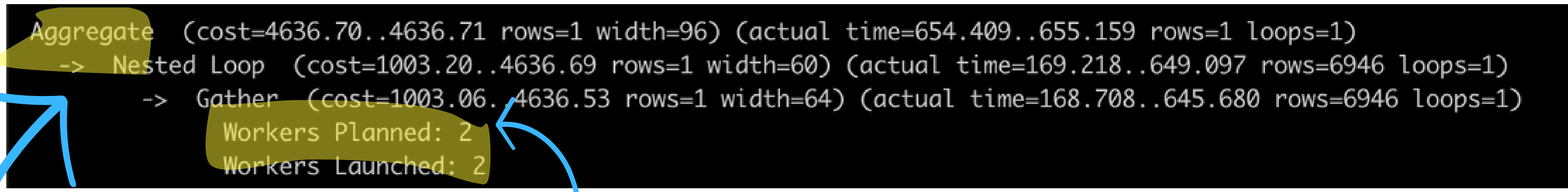
type of scan. "seq scan" indicates iterating through rows without an index. other types of scan will be used depending on index type.

actual time taken by this part of the plan

number of rows This can be larger than the size of the table, when using JOINs



```
Aggregate  (cost=3871.48..3871.49 rows=1 width=32) (actual time=2011.705..2011.707 rows=1 loops=1)
  ->  Nested Loop  (cost=8.11..3871.47 rows=7 width=17) (actual time=8.142..2006.087 rows=7834 loops=1)
        Join Filter: (mc.movie_id = t.id)
        ->  Nested Loop  (cost=7.68..3867.98 rows=7 width=8) (actual time=7.919..1510.987 rows=7834 loops=1)
              ->  Nested Loop  (cost=7.26..3788.22 rows=179 width=12) (actual time=3.517..1119.842 rows=148552 loops=1)
                    ->  Nested Loop  (cost=6.83..3768.21 rows=34 width=4) (actual time=3.148..400.268 rows=41840 loops=1)
                          ->  Seq Scan on keyword k  (cost=0.00..2626.12 rows=1 width=4) (actual time=0.258..7.280 rows=1 loop
s=1)
                                Filter: ((keyword)::text = 'character-name-in-title'::text)
                                Rows Removed by Filter: 134169
                          ->  Bitmap Heap Scan on movie_keyword mk  (cost=6.83..1138.99 rows=309 width=8) (actual time=2.887..
390.724 rows=41840 loops=1)
                                Recheck Cond: (keyword_id = k.id)
                                Heap Blocks: exact=11541
s=1)
                                      Index Cond: (keyword_id = k.id)
s=41840)
                          Index Cond: (movie_id = mk.movie_id)
```



```
Aggregate  (cost=4636.70..4636.71 rows=1 width=96) (actual time=654.409..655.159 rows=1 loops=1)
  ->  Nested Loop  (cost=1003.20..4636.69 rows=1 width=60) (actual time=169.218..649.097 rows=6946 loops=1)
        ->  Gather  (cost=1003.06..4636.53 rows=1 width=64) (actual time=168.708..645.680 rows=6946 loops=1)
              Workers Planned: 2
              Workers Launched: 2
```

number of parallel workers used some queries are parallelizable. A merge step will be added later in the plan

this particular query uses a MIN() so it needs to aggregate the value over the result set

String operations (like using the LIKE operator on a text field) are pretty slow.

If doing queries on text fields, consider either pre-processing the data so that you can use an exact match, or using GIN indexes for text and jsonb fields.

Relevant link: https://pganalyze.com/blog/gin-index

SIXTH: WHY IT'S ALWAYS DAY 2.
(EVEN ON AWS!)

"Day 2 operations" refers to what you do once your system is live in production.

Your database performance, however it currently looks on your laptop, will drastically change when real people use your application.

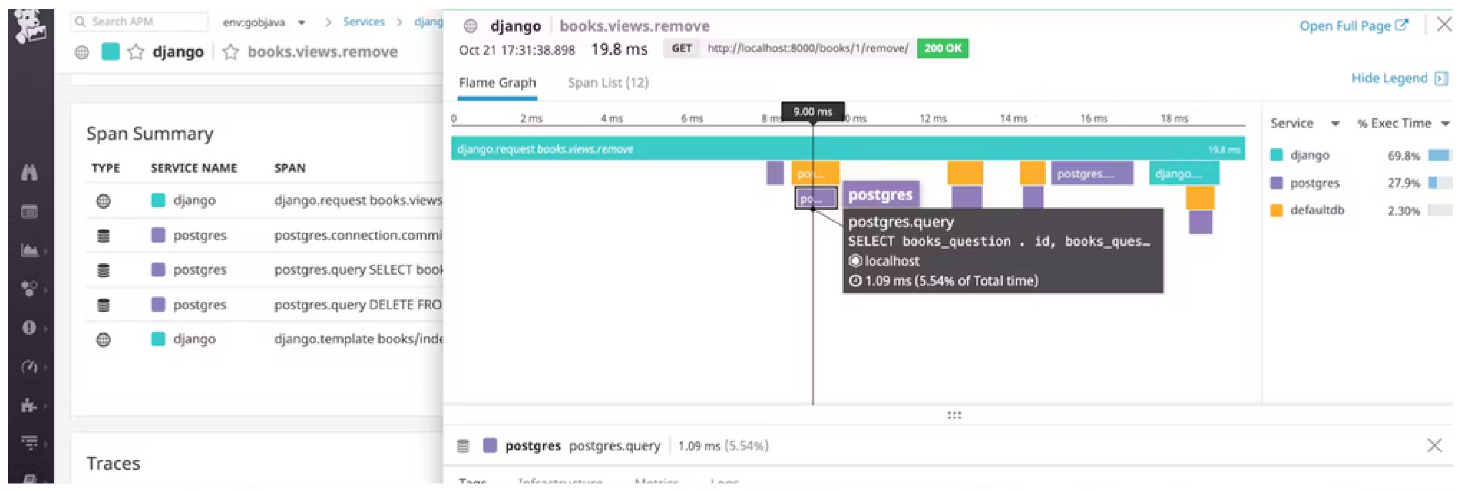
You're beyond advice now. You need data.

To reproduce some types of issues that only occur with production scale, you can consider load testing your database.

This is relatively safe as you can run it on a local copy. Make sure to use a set of test queries that is representative of your real-world use.

pgbench can be used for load testing.

Relevant link: https://www.postgresql.org/docs/current/pgbench.html



django books.views.remove
Oct 21 17:31:38.898 19.8 ms GET http://localhost:8000/books/1/remove/ 200 OK

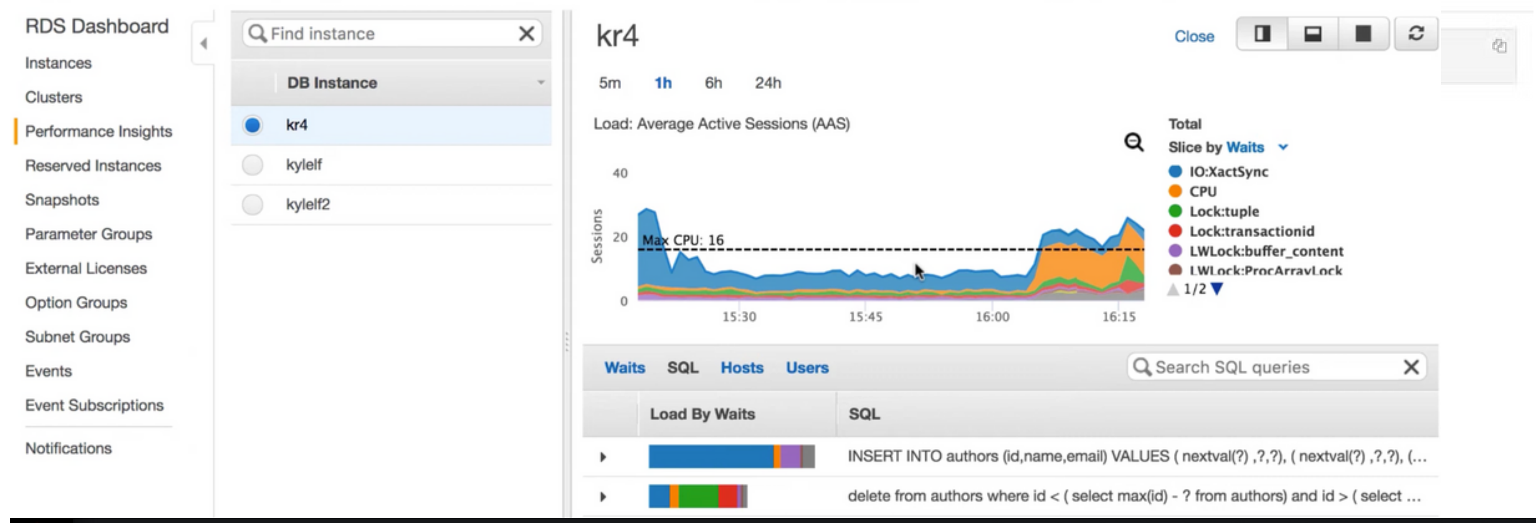
Span Summary

TYPE	SERVICE NAME	SPAN
	django	django.request books.views
	postgres	postgres.connection.commi
	postgres	postgres.query SELECT book
	postgres	postgres.query DELETE FRO
	django	django.template books/inde

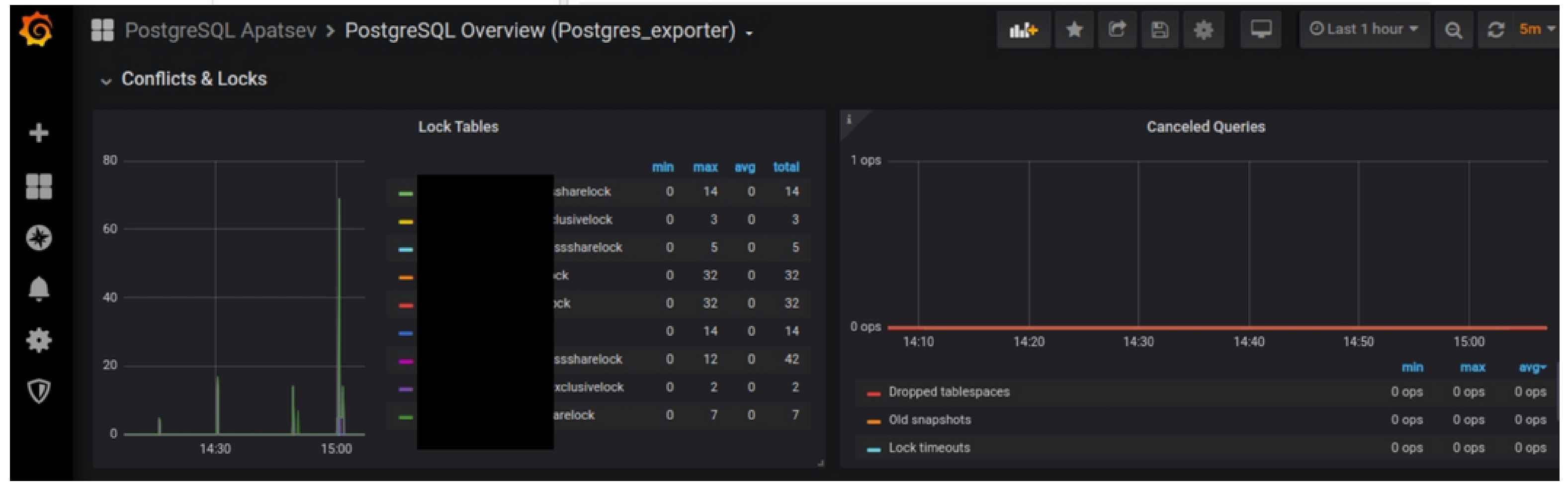
postgres.query
SELECT books_question . id, books_ques…
localhost
1.09 ms (5.54% of Total time)

Service	% Exec Time
django	69.8%
postgres	27.9%
defaultdb	2.30%

postgres postgres.query 1.09 ms (5.54%)



Load By Waits	SQL
	INSERT INTO authors (id,name,email) VALUES (nextval(?) ,?,?), (nextval(?) ,?,?), (…
	delete from authors where id < (select max(id) - ? from authors) and id > (select …


PostgreSQL Apatsev > PostgreSQL Overview (Postgres_exporter)
Last 1 hour
5m
Conflicts & Locks
Lock Tables
Canceled Queries

	min	max	avg	total
[redacted] sharelock	0	14	0	14
[redacted] lusivelock	0	3	0	3
[redacted] ssharelock	0	5	0	5
[redacted] ck	0	32	0	32
[redacted] ck	0	32	0	32
[redacted]	0	14	0	14
[redacted] ssharelock	0	12	0	42
[redacted] xclusivelock	0	2	0	2
[redacted] arelock	0	7	0	7

	min	max	avg
Dropped tablespaces	0 ops	0 ops	0 ops
Old snapshots	0 ops	0 ops	0 ops
Lock timeouts	0 ops	0 ops	0 ops

Several types of database monitoring are useful.

You want to know which queries are slower, and what pattern they occur in.

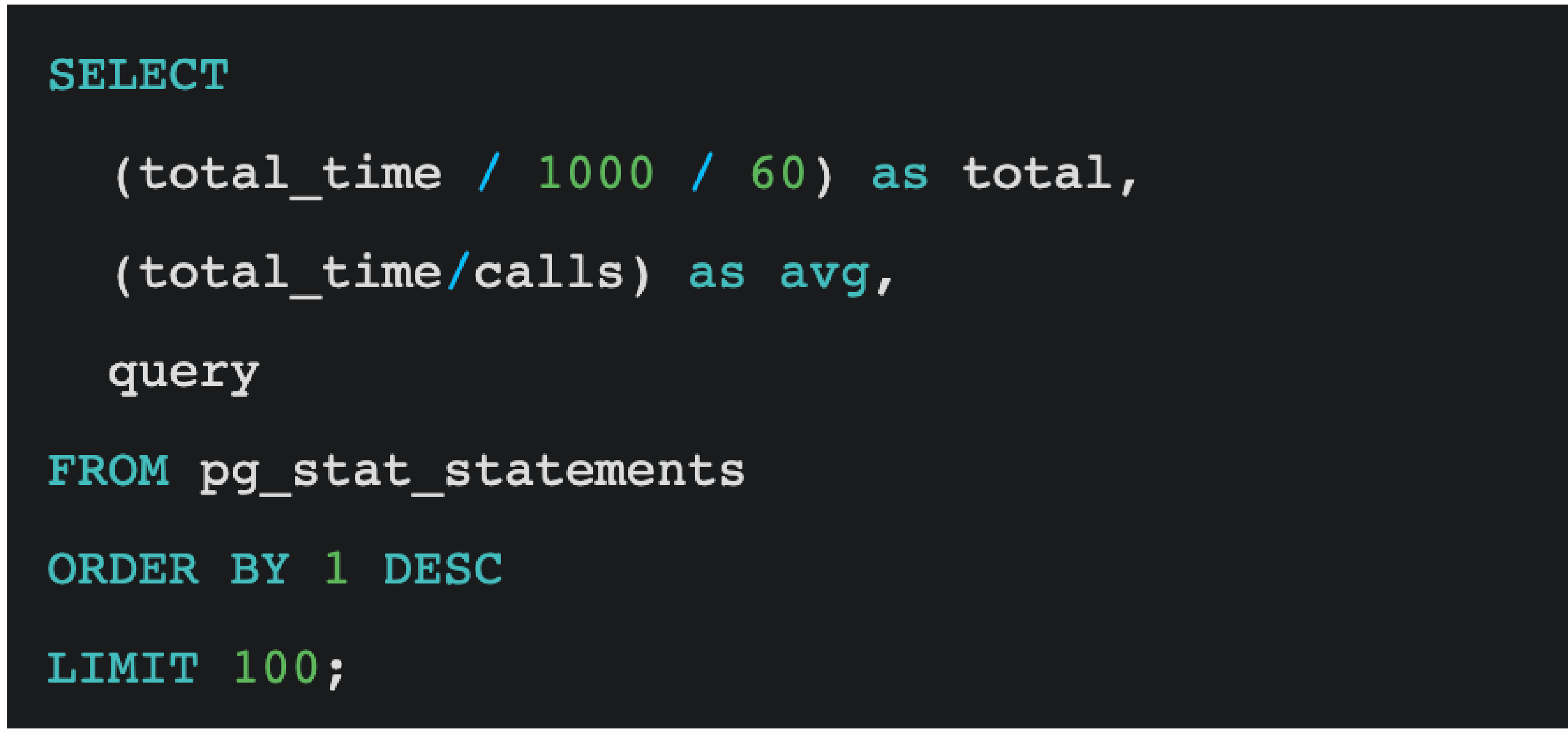
You want to know what is your average CPU/memory usage, and when it changes.

And you want some tracing of long-held locks, long transactions, and all errors.

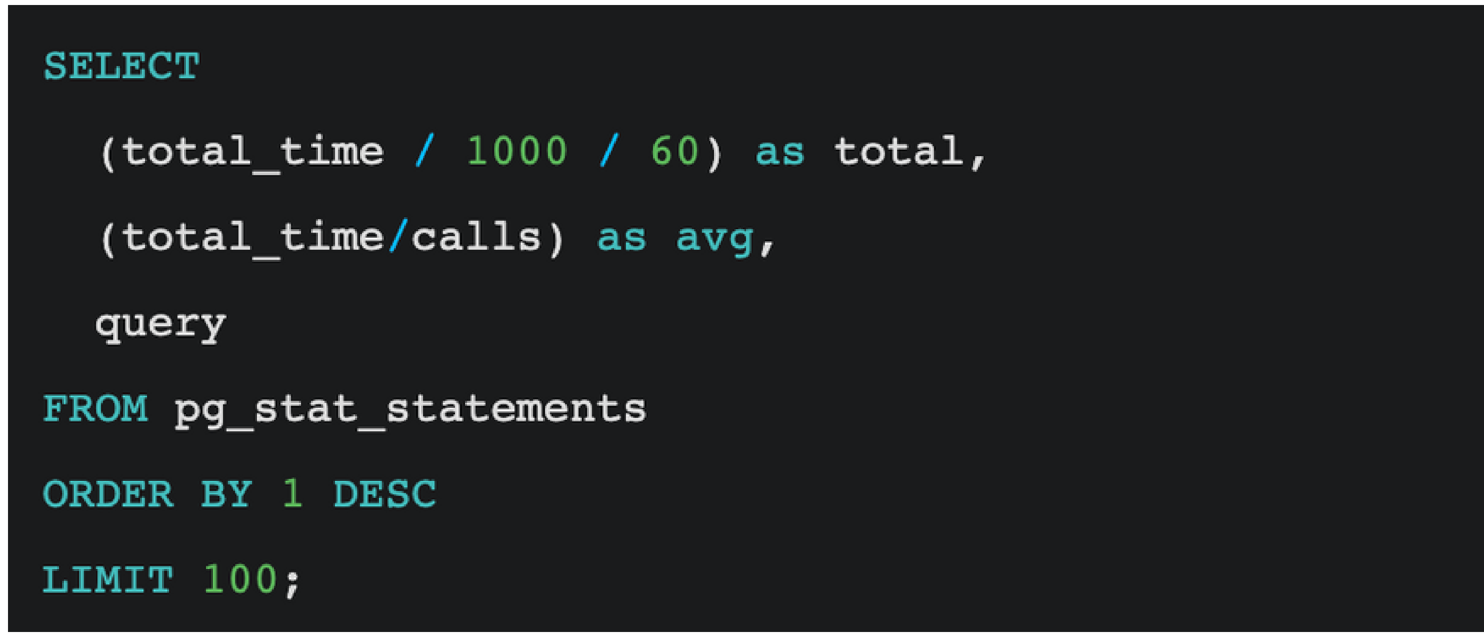
Relevant links:
https://grafana.com/grafana/dashboards/12273-postgresql-overview-postgres-exporter/
https://www.datadoghq.com/blog/collect-postgresql-data-with-datadog/#tracing-postgresql-queries-with-apm
https://youtu.be/4462hcfkApM

For cloud databases, slow queries (and query performance generally) can be found in both RDS performance insights and the Datadog APM view.

For Postgres in all deployments, pg_stat_statements and the pg_activity tool also provide that information.



```
SELECT
  (total_time / 1000 / 60) as total,
  (total_time/calls) as avg,
  query
FROM pg_stat_statements
ORDER BY 1 DESC
LIMIT 100;
```



```
SELECT
  (total_time / 1000 / 60) as total,
  (total_time/calls) as avg,
  query
FROM pg_stat_statements
ORDER BY 1 DESC
LIMIT 100;
```

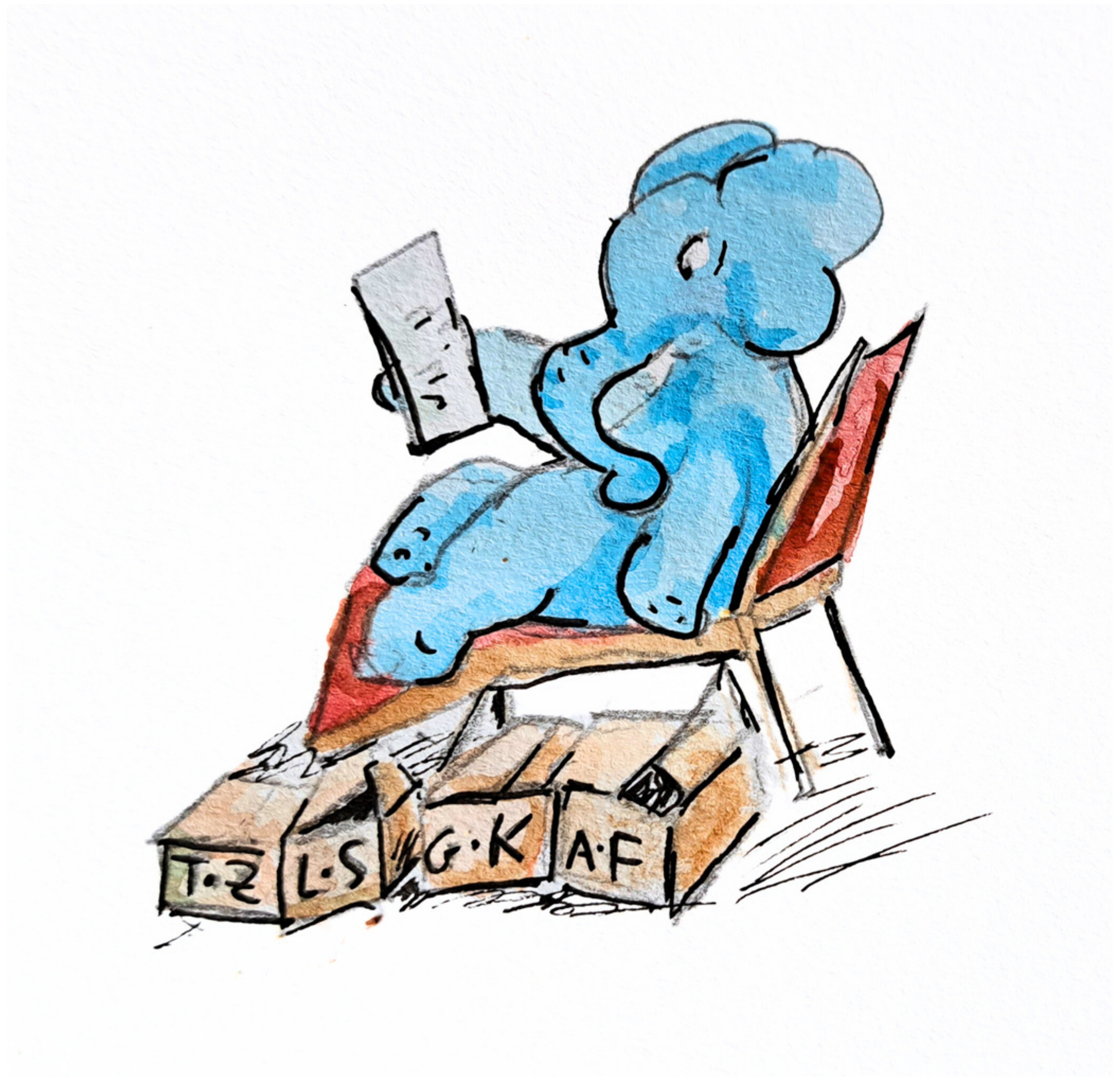
Relevant link: https://www.citusdata.com/blog/2019/02/08/the-most-useful-postgres-extension-pg-stat-statements/

SEVENTH: WHY YOUR DATABASE ISN'T "TOO BIG". UNTIL IT IS.

Partitioning allows you to split a table into physical partitions based on a given column value.

This makes it easier to query and drop individual partitions, as well as gives smaller physical blocks to work with when querying that field.

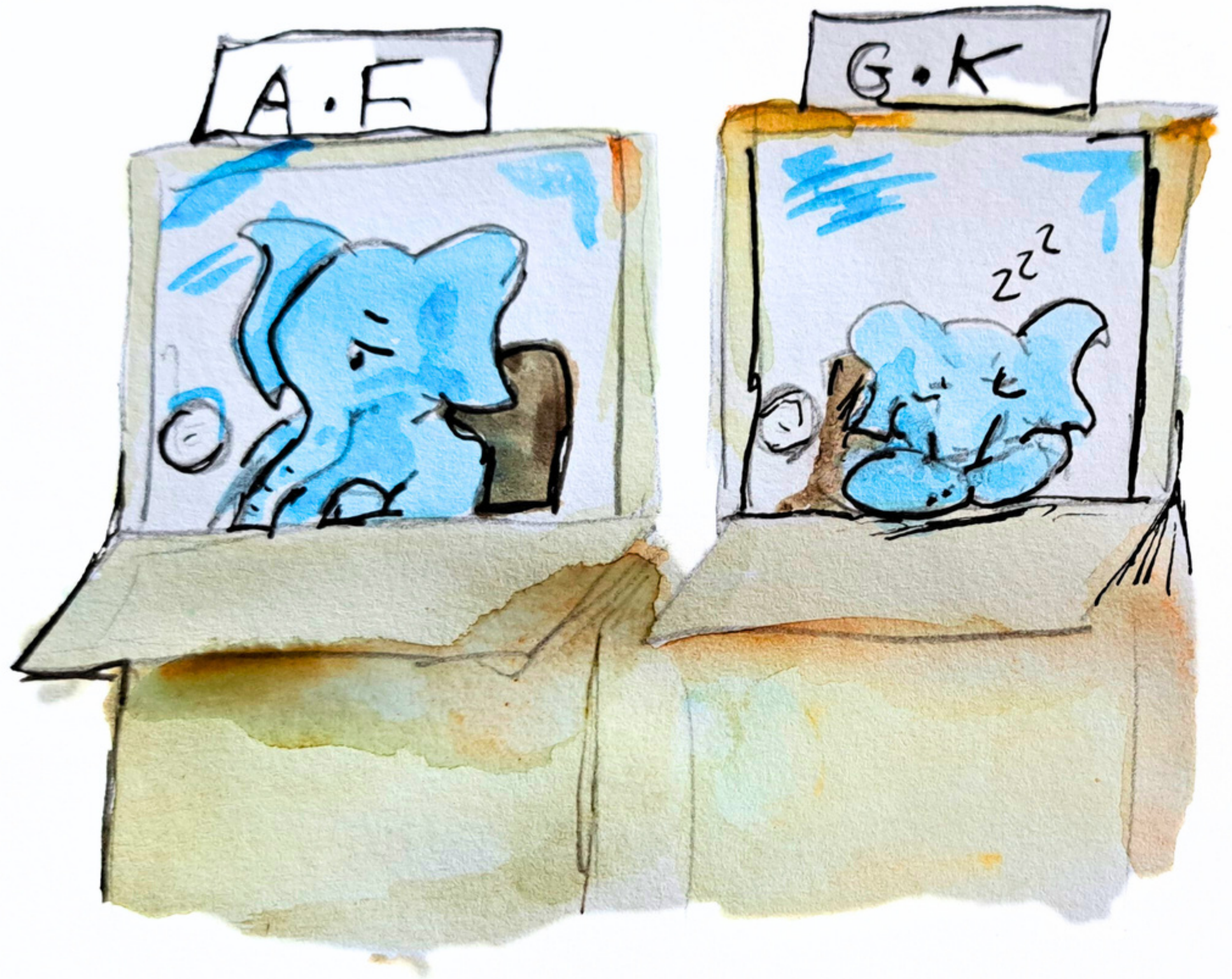
Partitions can be based on a range of values or on specific individual values. They're a good match for time values or for partitioning out most active rows.



Sharding refers to splitting up the data in your database into several instances according to factors such as:

the geography it needs to live in,
the customers it is serving,
the frequency it is updated,
or its data retention policy.

It often leads to smaller data (and therefore lower cost, better perfomance).
It is also not easy.



EIGHTH: WHY YOU SHOULD USE THE INDEX, LUKE UNLESS YOU DON'T ACTUALLY USE IT.

"Just add an index" is common advice...

But it has a catch.
First, indexes take up a lot of space.
Each index grows the size of your table,
as well as the time it takes to insert fields
(and VACUUM them).

Second, not all query plans will use your
new index. As a result, monitoring is
important to prune less important
indexes.

Postgres provides a view called
pg_stat_all_indexes
(it is part of the family of super-useful
pg_stats views).

It contains statistics on how often your indexes are being hit, and how many rows are returned each time.

It can be combined with EXPLAIN output to analyze how many of your queries use a given index, and what's their resulting performance.

NEXT: ASK YOUR OWN QUESTIONS.

Learning is not just in books.

If you want to learn how databases work, a really good start is asking deep questions about how your database works. For instance:

What is the slowest page load in your application? Which SQL queries does it translate to? How long do they take?

What happens when your database server reboots if your service had transactions pending?

You want to replace your existing table with a new one with a different schema. What code do you need to change? How long will it take to execute?

In this way...

it is just the beginning.

The best way to prevent having big problems in production...

is to seek out small problems before they grow up.

And then you'll need an elephant's memory...

to keep track of everything that went wrong, and the root cause of it.

USEFUL LINKS *for* CURIOUS PEOPLE

https://wiki.postgresql.org/wiki/Main_Page

https://sqlfordevs.com/

https://www.databass.dev/

https://microservices.io/patterns/data/database-per-service.html