RAILS MIGRATIONS

SCALING VS PERFORMANCE
LATENCY — IO VS CPU BURN
UNDERSTANDING POSTGRESQL STATISTICS

OPTIMIZING SCHEMA AND QUERIES

POSTGRESQL DONE RIGHT

By Konstantin Gredeskoul

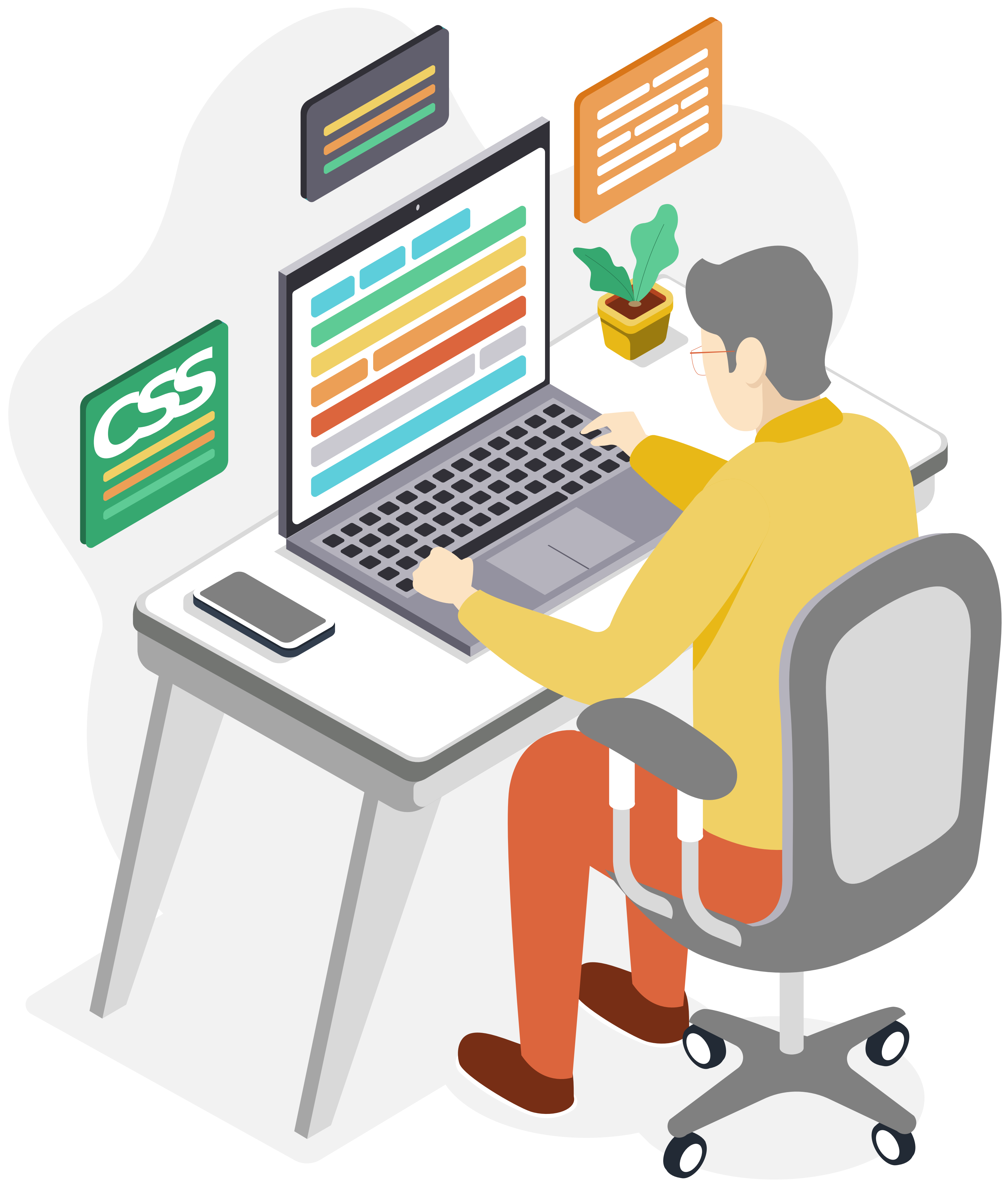
HealthSherpa

PGConf Silicon Valley 2015 · AU RUBY CONF

@kigster

@kig

WHY SHOULD YOU LISTEN TO ME?



- I have been using PostgreSQL since 2004 (version 7.4)
- In 2012 my team scaled wanelo.com to serve 8,000 rack requests per second.
- This equals to about 100K PostgreSQL transactions per second (!!!)
- In 2014 I spoke at SF PG User Group (PUG) about how we scaled Wanelo
Slides: https://bit.ly/scaling-pg

RAILS MIGRATIONS DONE RIGHT

BEST PRACTICES FOR RAILS MIGRATIONS

- Avoid modifying data in schema migrations
- Rails guide says:

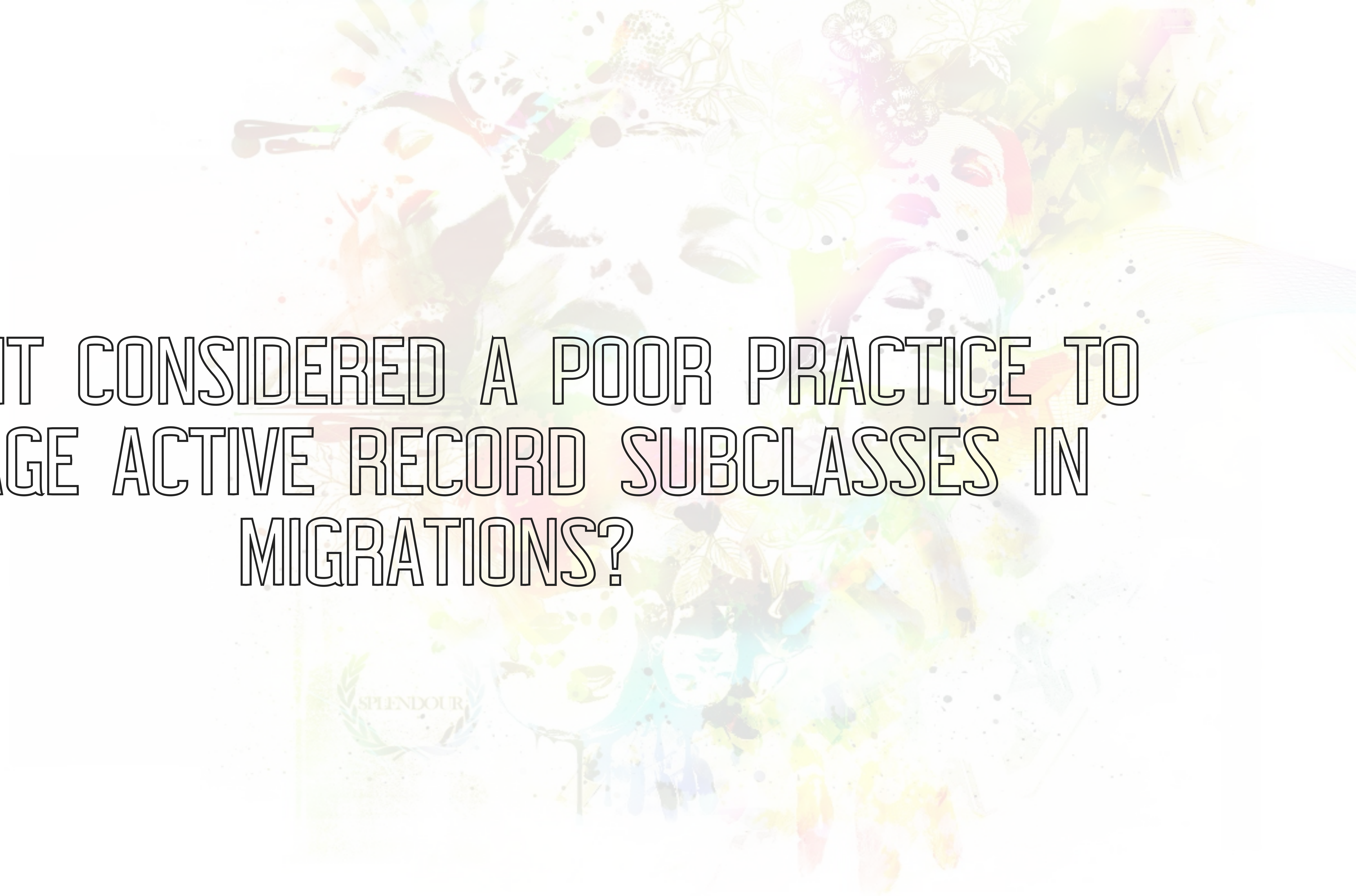
MIGRATIONS ARE A FEATURE OF ACTIVE RECORD THAT ALLOWS YOU TO EVOLVE YOUR DATABASE SCHEMA OVER TIME. RATHER THAN WRITE SCHEMA MODIFICATIONS IN PURE SQL, MIGRATIONS ALLOW YOU TO USE AN EASY RUBY DSL TO DESCRIBE CHANGES TO YOUR TABLES.

- There is no word **"data"** anywhere…
- Nor there is a mention of Rails Models.

BEST PRACTICES FOR RAILS MIGRATIONS (CTD.)

- We have "**data_migrate**" gem specifically for the purposes of data migrations, although for long-running migrations it's better to write rake tasks and execute them interactively.
- Whether using schema or data migrations, avoid using model classes.
- Using things like this is discouraged:

```
User.create!(...)
```



WHY IS IT CONSIDERED A POOR PRACTICE TO LEVERAGE ACTIVE RECORD SUBCLASSES IN MIGRATIONS?

AN EXAMPLE FROM THE REPO

```ruby
# db/migrate/20131217214153_move_agent_state_license_column_to_agent_licenses.rb
class MoveAgentStateLicenseColumnToAgentLicenses < ActiveRecord::Migration
  class Agent < ApplicationRecord
  end

  def up
    Agent.all.each do |agent|
      license = AgentLicense.new(:agent => agent,
                                 :state => agent.state,
                                 :license_number => agent.state_license)

      license.save!
    end

    remove_column :agents, :state_id
    remove_column :agents, :state_license
    Agent.reset_column_information
  end

  def down
    # removed for brevity
  end
end
```

HOUSTON, WE HAVE A PROBLEM

BEST PRACTICES FOR RAILS MIGRATIONS (CTD.)

- Why is this line a problem?

```
license.save!
```

HOUSTON, WE HAVE A PROBLEM

- To answer this we should review how the migrations are used throughout the life of the project.

- One of the fundamental expectations of any rails project is that this command can create and migrate the database to current point:

```
rake db:create && rake db:migrate
```

WHY CAN'T WE USE MODELS IN MIGRATIONS?

- Because migrations should be "frozen in time" and be able to build your schema by applying incremental updates to the database.
- However, model classes are constantly changing. New validations are added.
- Old migration will not know of any such changes, and when attempting to save a model, a recently added validation may cause it to fail.

AS A RESULT, **DB:MIGRATE** RAKE TASK NO LONGER WORKS.

WHERE IS EXCHANGE-COMPARE TODAY?

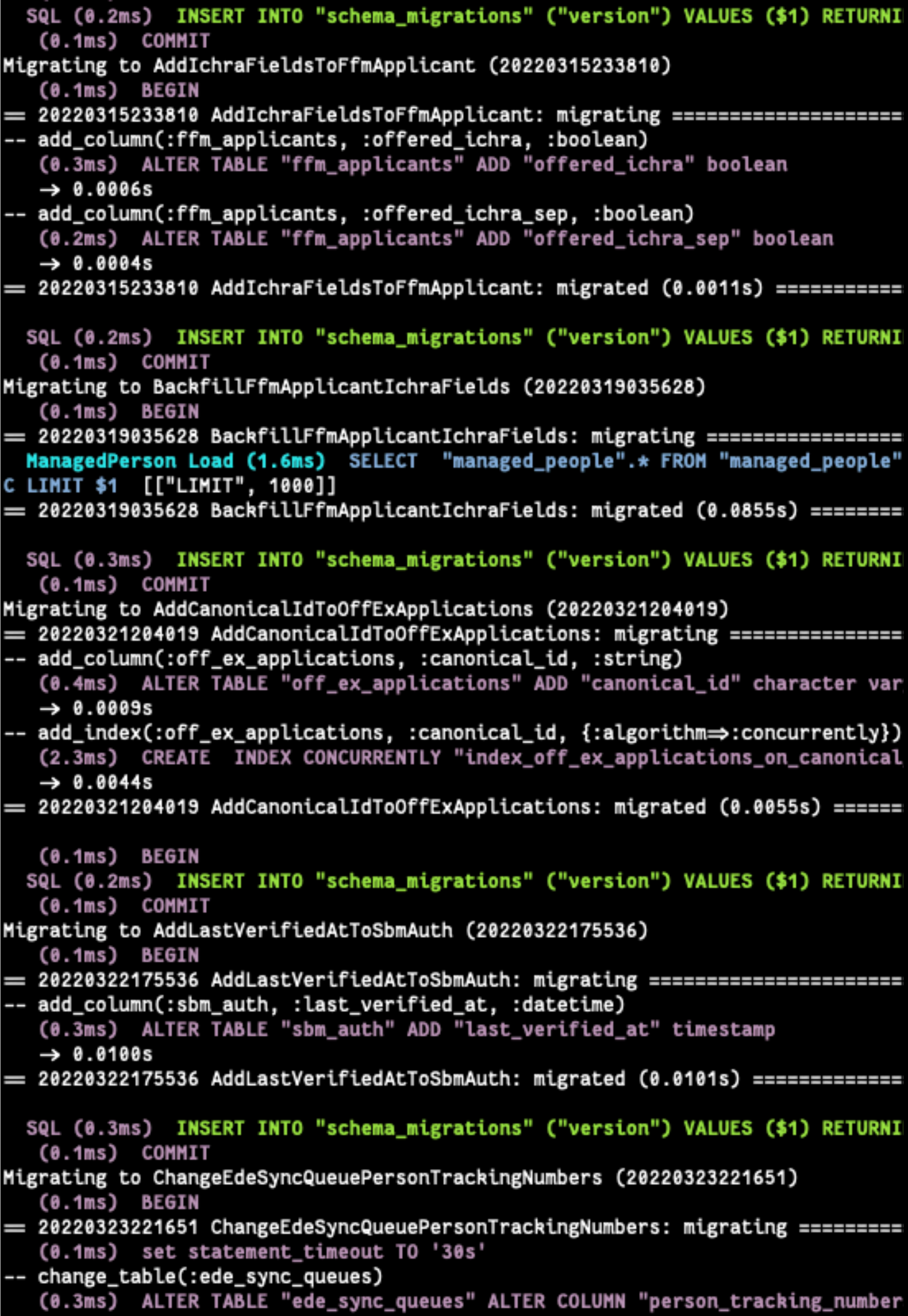
- When I joined the company I was unable to run

  ```
  rake db:migrate
  ```

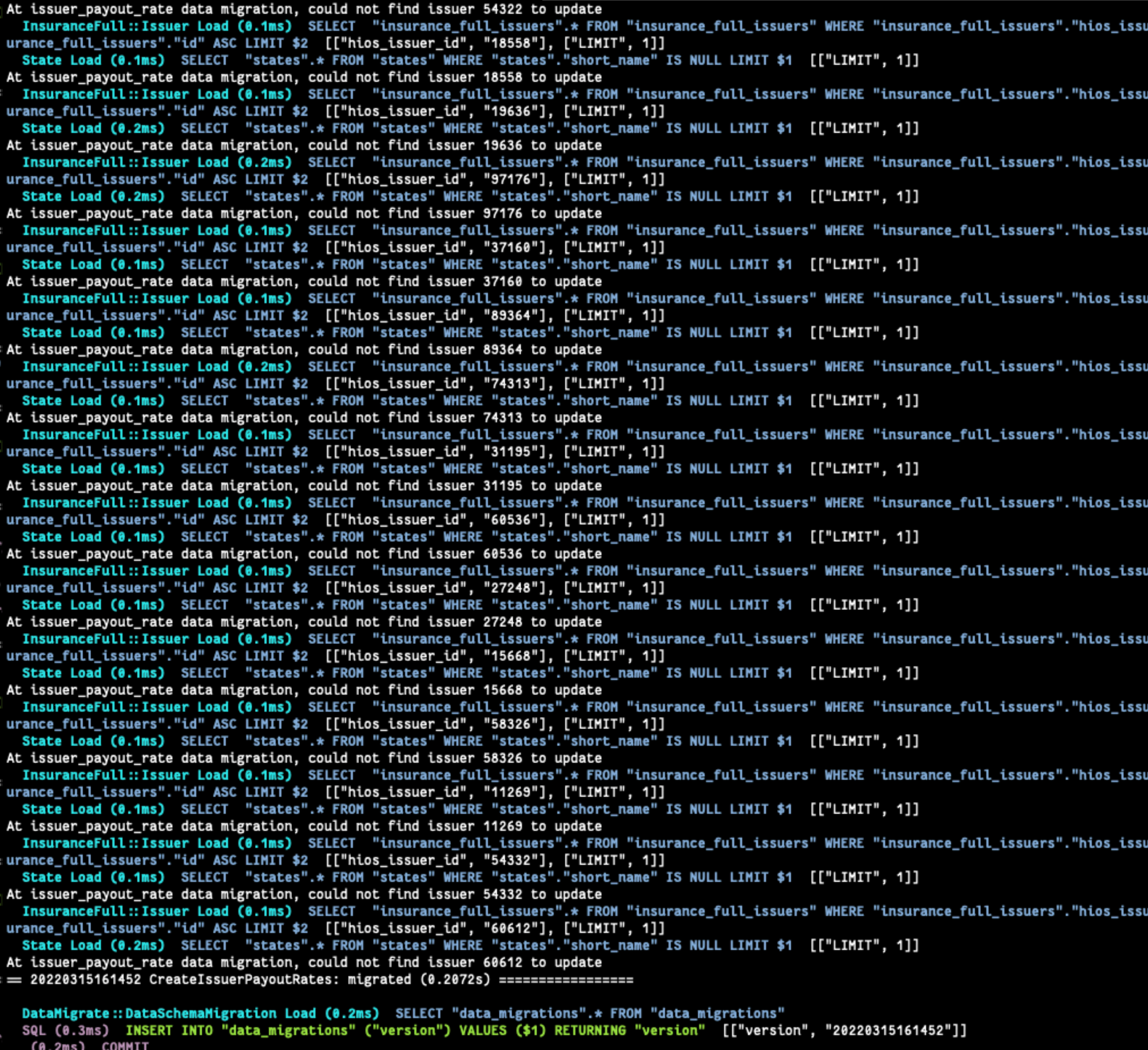
  successfully.

- Today — it works! Likely due to fixes to migrations and Rails 5 upgrade. Yay!

- Proof! (demo)



```
  SQL (0.2ms)  INSERT INTO "schema_migrations" ("version") VALUES ($1) RETURNI
   (0.1ms)  COMMIT
Migrating to AddIchraFieldsToFfmApplicant (20220315233810)
   (0.1ms)  BEGIN
== 20220315233810 AddIchraFieldsToFfmApplicant: migrating ====================
-- add_column(:ffm_applicants, :offered_ichra, :boolean)
   (0.3ms)  ALTER TABLE "ffm_applicants" ADD "offered_ichra" boolean
   -> 0.0006s
-- add_column(:ffm_applicants, :offered_ichra_sep, :boolean)
   (0.2ms)  ALTER TABLE "ffm_applicants" ADD "offered_ichra_sep" boolean
   -> 0.0004s
== 20220315233810 AddIchraFieldsToFfmApplicant: migrated (0.0011s) ===========

  SQL (0.2ms)  INSERT INTO "schema_migrations" ("version") VALUES ($1) RETURNI
   (0.1ms)  COMMIT
Migrating to BackfillFfmApplicantIchraFields (20220319035628)
   (0.1ms)  BEGIN
== 20220319035628 BackfillFfmApplicantIchraFields: migrating =================
  ManagedPerson Load (1.6ms)  SELECT  "managed_people".* FROM "managed_people"
C LIMIT $1  [["LIMIT", 1000]]
== 20220319035628 BackfillFfmApplicantIchraFields: migrated (0.0855s) ========

  SQL (0.3ms)  INSERT INTO "schema_migrations" ("version") VALUES ($1) RETURNI
   (0.1ms)  COMMIT
Migrating to AddCanonicalIdToOffExApplications (20220321204019)
== 20220321204019 AddCanonicalIdToOffExApplications: migrating ===============
-- add_column(:off_ex_applications, :canonical_id, :string)
   (0.4ms)  ALTER TABLE "off_ex_applications" ADD "canonical_id" character var
   -> 0.0009s
-- add_index(:off_ex_applications, :canonical_id, {:algorithm=>:concurrently})
   (2.3ms)  CREATE  INDEX CONCURRENTLY "index_off_ex_applications_on_canonical
   -> 0.0044s
== 20220321204019 AddCanonicalIdToOffExApplications: migrated (0.0055s) ======

   (0.1ms)  BEGIN
  SQL (0.2ms)  INSERT INTO "schema_migrations" ("version") VALUES ($1) RETURNI
   (0.1ms)  COMMIT
Migrating to AddLastVerifiedAtToSbmAuth (20220322175536)
   (0.1ms)  BEGIN
== 20220322175536 AddLastVerifiedAtToSbmAuth: migrating ======================
-- add_column(:sbm_auth, :last_verified_at, :datetime)
   (0.3ms)  ALTER TABLE "sbm_auth" ADD "last_verified_at" timestamp
   -> 0.0100s
== 20220322175536 AddLastVerifiedAtToSbmAuth: migrated (0.0101s) =============

  SQL (0.3ms)  INSERT INTO "schema_migrations" ("version") VALUES ($1) RETURNI
   (0.1ms)  COMMIT
Migrating to ChangeEdeSyncQueuePersonTrackingNumbers (20220323221651)
   (0.1ms)  BEGIN
== 20220323221651 ChangeEdeSyncQueuePersonTrackingNumbers: migrating =========
   (0.1ms)  set statement_timeout TO '30s'
-- change_table(:ede_sync_queues)
   (0.3ms)  ALTER TABLE "ede_sync_queues" ALTER COLUMN "person_tracking_number
```



```
At issuer_payout_rate data migration, could not find issuer 54322 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "18558"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 18558 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "19636"], ["LIMIT", 1]]
  State Load (0.2ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 19636 to update
  InsuranceFull::Issuer Load (0.2ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "97176"], ["LIMIT", 1]]
  State Load (0.2ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 97176 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "37160"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 37160 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "89364"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 89364 to update
  InsuranceFull::Issuer Load (0.2ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "74313"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 74313 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "31195"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 31195 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "60536"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 60536 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "27248"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 27248 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "15668"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 15668 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "58326"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 58326 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "11269"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 11269 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "54332"], ["LIMIT", 1]]
  State Load (0.1ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 54332 to update
  InsuranceFull::Issuer Load (0.1ms)  SELECT  "insurance_full_issuers".* FROM "insurance_full_issuers" WHERE "insurance_full_issuers"."hios_issu
urance_full_issuers"."id" ASC LIMIT $2  [["hios_issuer_id", "60612"], ["LIMIT", 1]]
  State Load (0.2ms)  SELECT  "states".* FROM "states" WHERE "states"."short_name" IS NULL LIMIT $1  [["LIMIT", 1]]
At issuer_payout_rate data migration, could not find issuer 60612 to update
== 20220315161452 CreateIssuerPayoutRates: migrated (0.2072s) ================

  DataMigrate::DataSchemaMigration Load (0.2ms)  SELECT "data_migrations".* FROM "data_migrations"
  SQL (0.3ms)  INSERT INTO "data_migrations" ("version") VALUES ($1) RETURNING "version"  [["version", "20220315161452"]]
   (0.2ms)  COMMIT
```

IT WORKS!

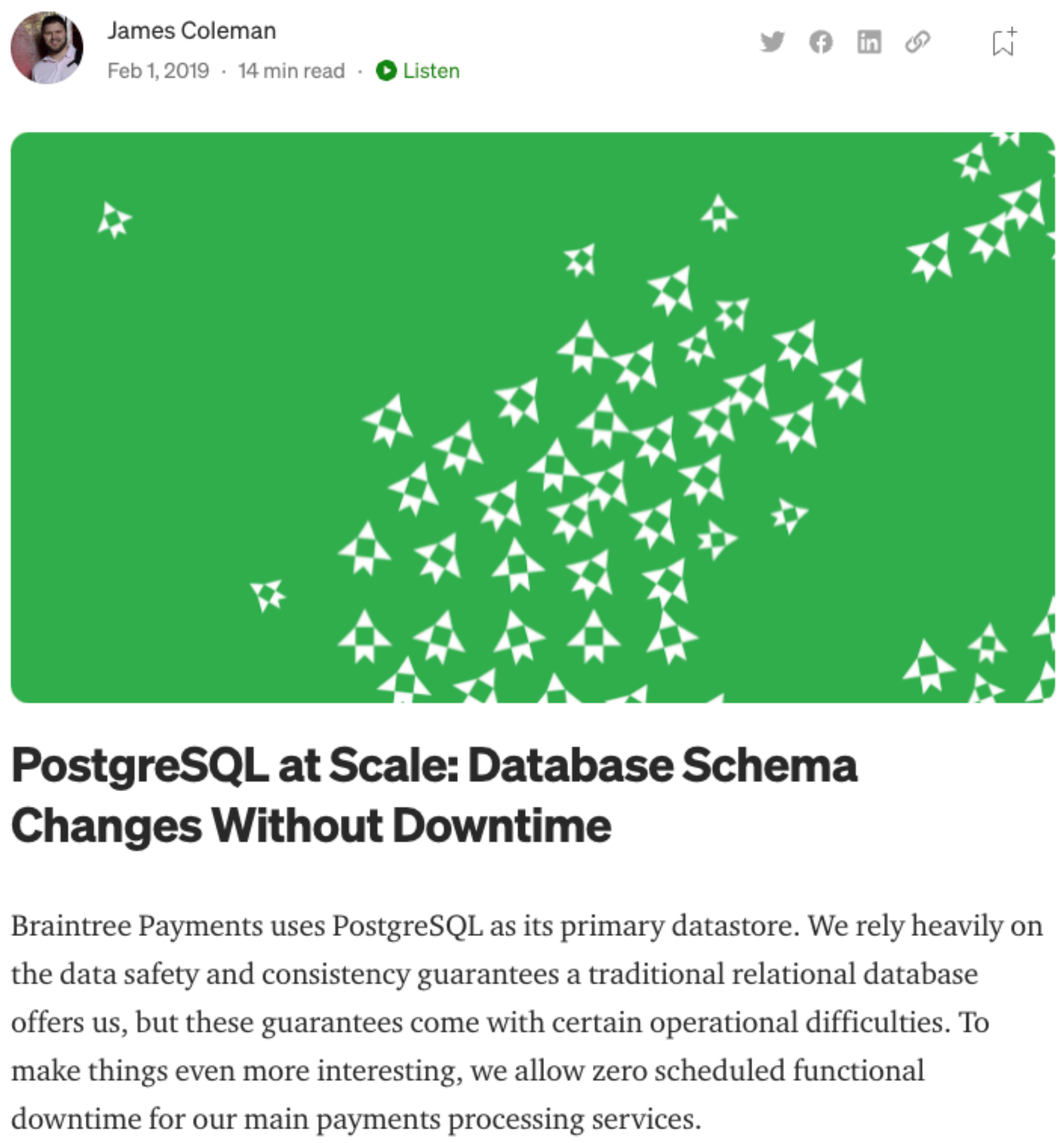
- Today you can run successfully (in about one minute):

```
rake db:create && \
rake db:migrate && \
rake data:migrate
```

- Next we should invest some time into making this work too:

```
rake db:seed
```

HI-AVAILABILITY MIGRATIONS


James Coleman
Feb 1, 2019 · 14 min read · Listen

PostgreSQL at Scale: Database Schema Changes Without Downtime

Braintree Payments uses PostgreSQL as its primary datastore. We rely heavily on the data safety and consistency guarantees a traditional relational database offers us, but these guarantees come with certain operational difficulties. To make things even more interesting, we allow zero scheduled functional downtime for our main payments processing services.

- Read at https://bit.ly/safe-migrations
- Do not do any operations that requires exclusive table lock
  - **ACCESS EXCLUSIVE**: blocks all usage of the locked table.
  - **SHARE ROW EXCLUSIVE**: blocks concurrent DDL against and row modification (allowing reads) in the locked table.
  - **SHARE UPDATE EXCLUSIVE:** blocks concurrent DDL against the locked table.
-

SCALABILITY IN CONTEXT

PERFORMANCE & LATENCY VS SCALABILITY

SCALABILITY: IS THE CAPABILITY OF A SYSTEM, NETWORK, OR PROCESS TO HANDLE A GROWING AMOUNT OF WORK, OR ITS POTENTIAL TO BE ENLARGED IN ORDER TO ACCOMMODATE THAT GROWTH.

PERFORMANCE (INVERSELY PROPORTIONAL TO LATENCY): GENERALLY DESCRIBES THE TIME IT TAKES FOR VARIOUS OPERATIONS TO COMPLETE: I.E. USER INTERFACES TO LOAD, OR BACKGROUND JOBS TO COMPLETE.

QUESTION:

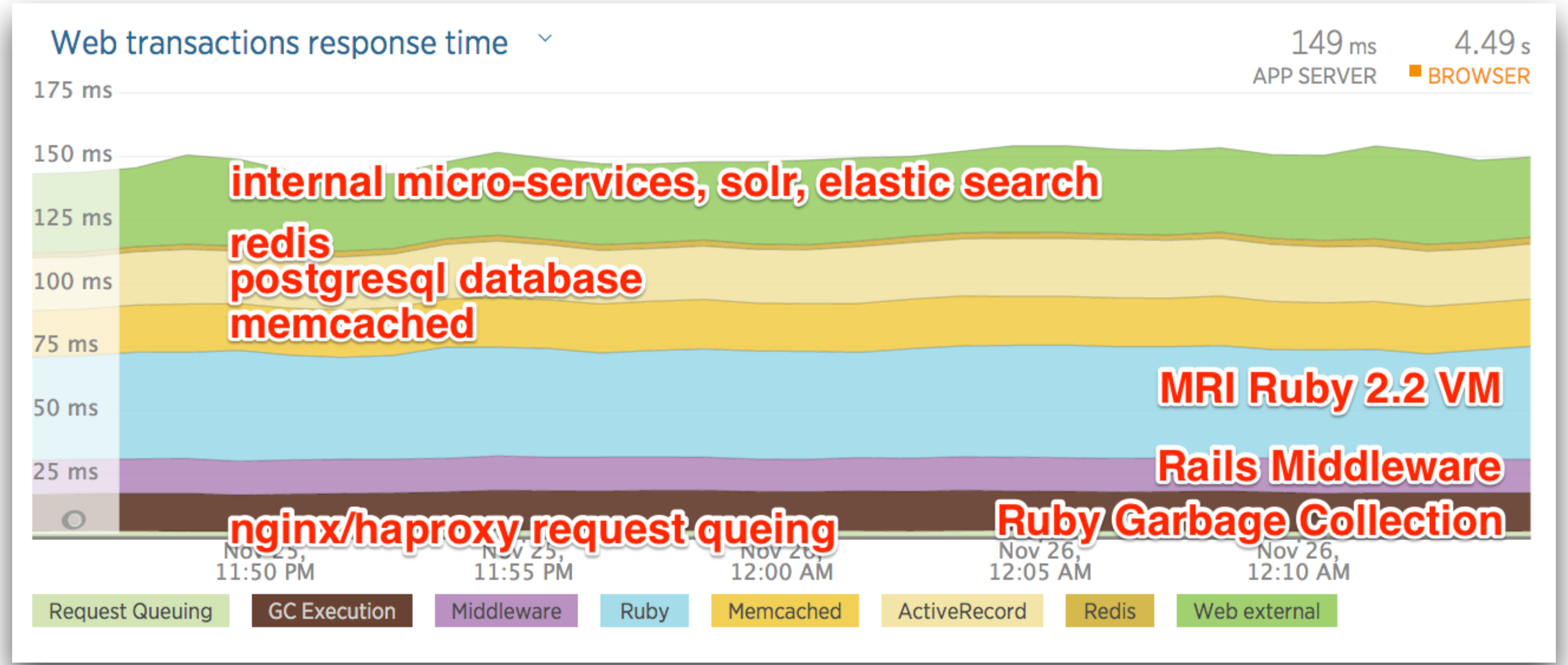
IS IT *NECESSARY* TO INCREASE APPLICATION'S PERFORMANCE (OR DECREASE IT'S LATENCY) IN ORDER TO INCREASE SCALABILITY?

NO.

BUILDING WEB APPS THAT SCALE

PERFORMANCE: REDUCING LATENCY

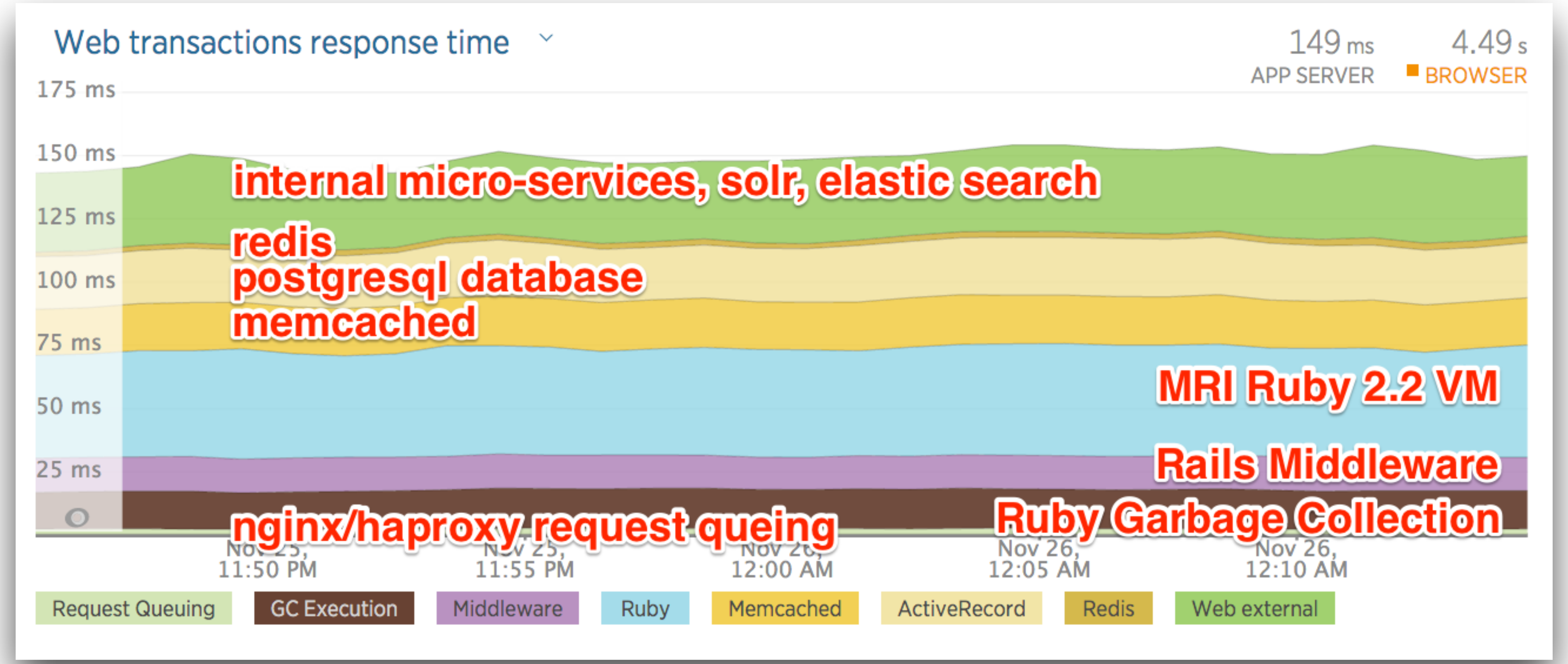
- For high traffic backends (50K+ RPMs) a server latency of < **100ms** or lower is recommended for web applications running rails to contain cloud costs
- For fast internal HTTP services, that wrap data-store – **5-10ms** or lower



Graph credits: © NewRelic, Inc.

ZOOM INTO SERVER LATENCY

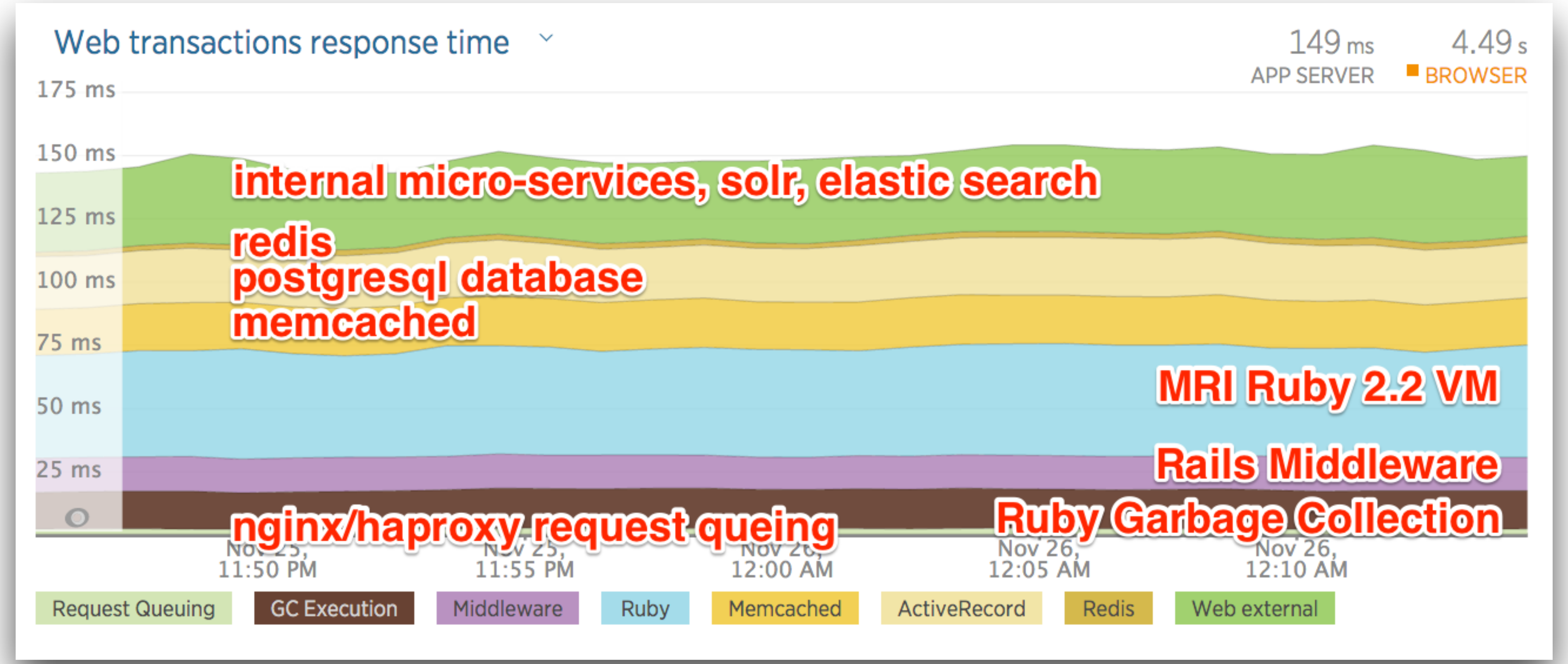
- Not all latency is born equal :)
- Everyone should have seen a similar NewRelic graph by now…
- What colors are "**CPU burn**", versus "**IO Wait**"?



Graph credits: © NewRelic, Inc.

ZOOM INTO SERVER LATENCY

- Internal Microservices, Solr, memcached, redis, database are **waiting on IO**
- RubyVM, Middleware, GC are all **CPU burn**
  - **CPU burn** is easy to scale out by adding more app servers



Graph credits: © NewRelic, Inc.

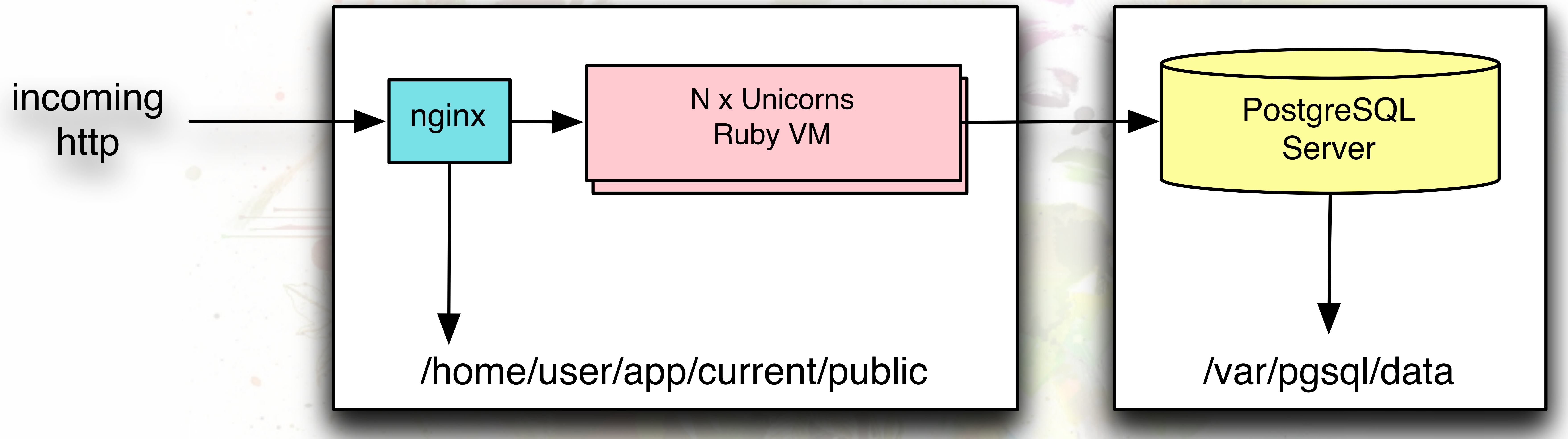
FOUNDATIONS

OF A MODERN WEB ARCHITECTURE

FOUNDATIONAL TECHNOLOGIES

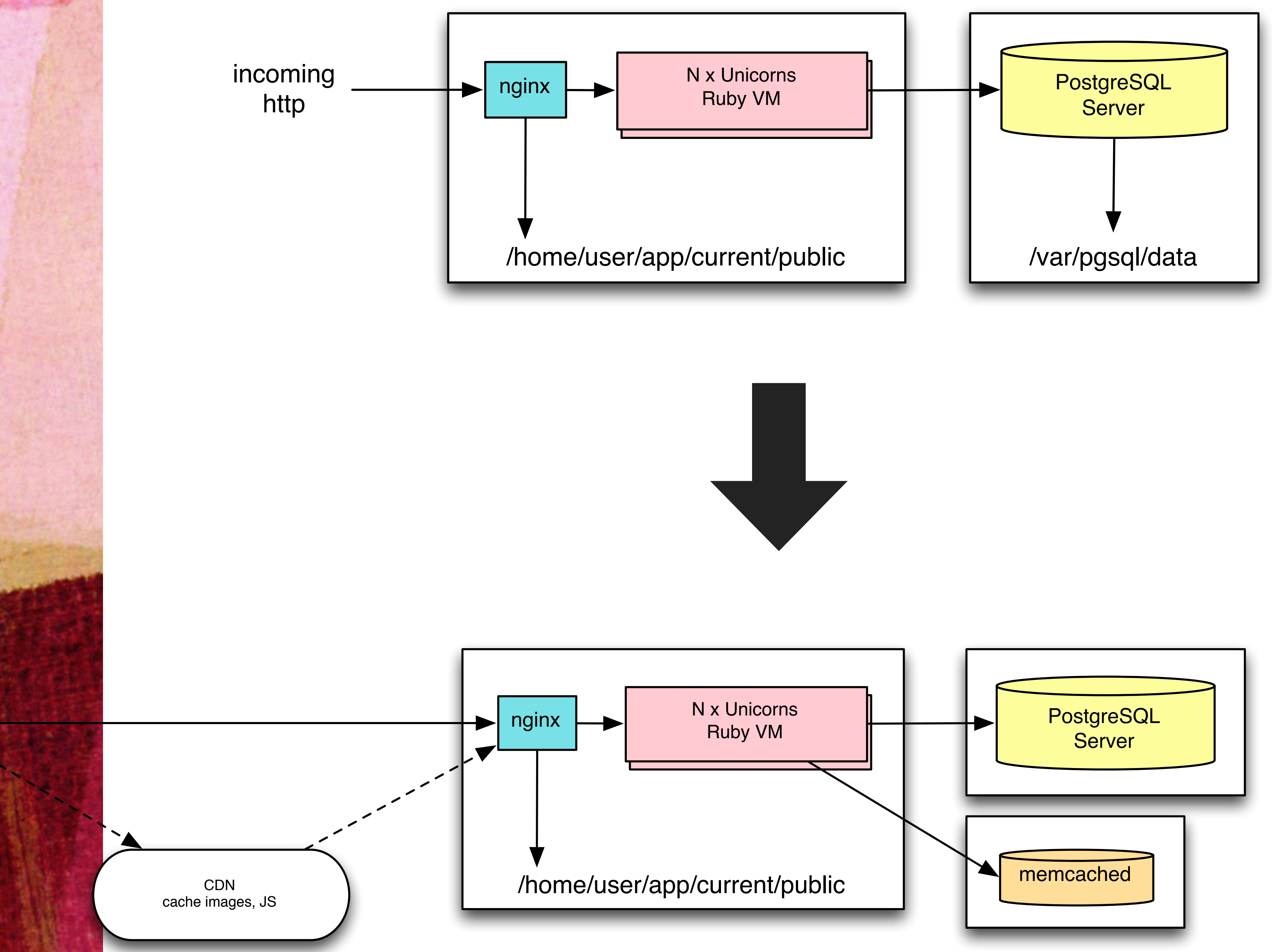
- programming language + framework (**RoR**)
- app server (**puma**)
- load balancer (**haproxy** + **nginx**)
- database (**postgresql**)
- hosting environment (we use **AWS**)
- deployment tools (**heroku / k8s**)
- server configuration tools (**terraform**)
- many others, such as monitoring, alerting

LET'S REVIEW – SUPER SIMPLE APP



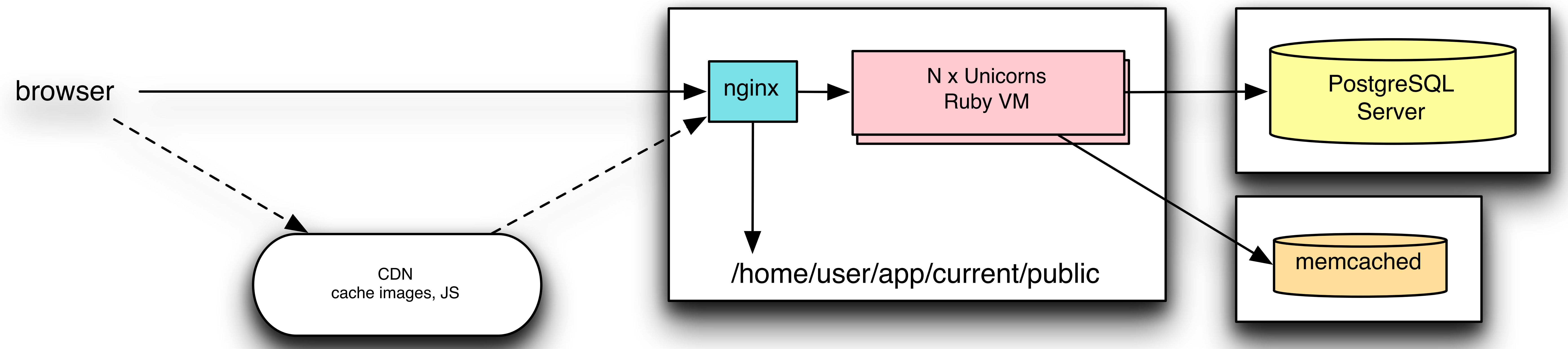
- no redundancy, no caching (yet)
- can only process N concurrent requests
- nginx will serve static assets, deal with slow clients
- web sessions are probably in the cookie

DON'T SHOOT YOURSELF IN THE FOOT! DO THIS.



- Install 2+ **memcached** servers for caching and use **Dalli gem** to connect to it for redundancy
- Switch to using **cookie-based web sessions**. Use tokens for API. In general, use sessions sparingly, assume they are transient and short-lived.
  - **Redis** is also an option for sessions, but it's not as easy to use two redis instances for redundancy, as easily as using memcached with **Dalli**. Plus, **Redis** is single-threaded, single process, while **memcached** is multi-threaded, multi-process.

ADD CACHING: CDN AND MEMCACHED

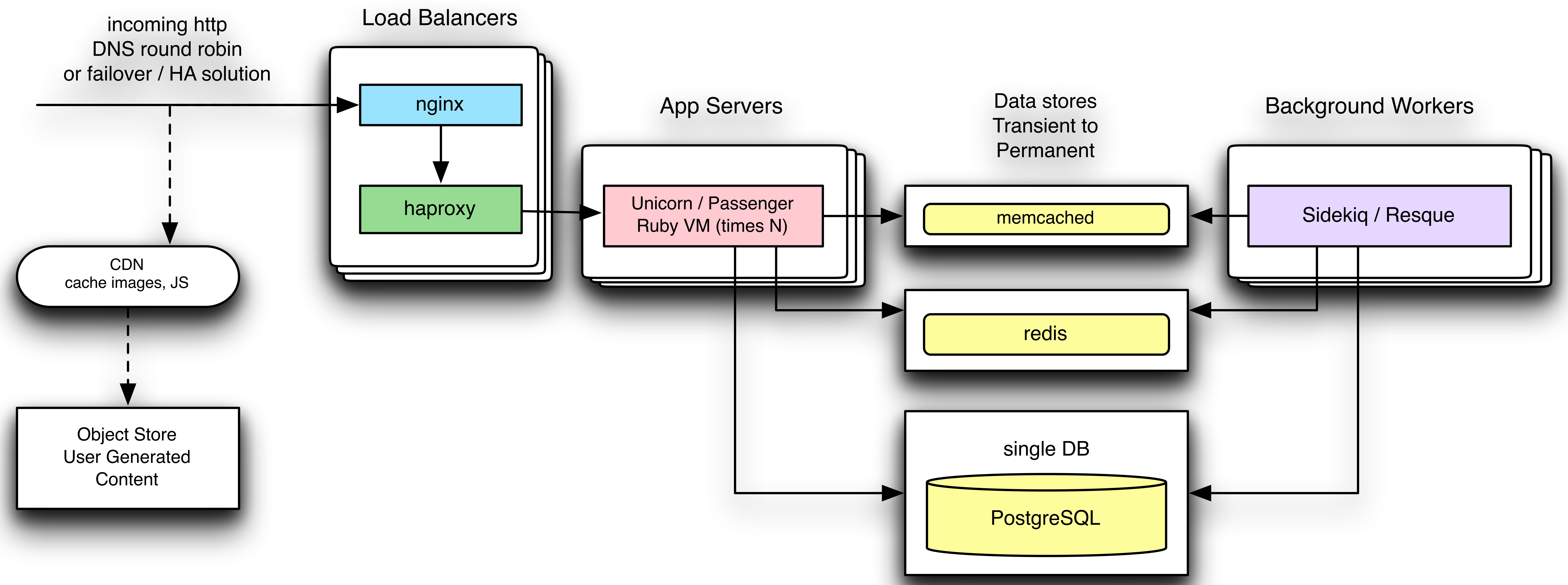


- geo distribute and cache your UGC and CSS/JS assets
- cache html and serialize objects in memcached
- can increase TTL to alleviate load, if traffic spikes

REMOVE SINGLE POINTS OF FAILURE:

- **Multiple load balancers** require DNS round robin and short TTL (dnsmadeeasy.com)
- **Multiple long-running tasks** (such as posting to Facebook or Twitter) require background job processing framework
- **Multiple app servers** require haproxy between nginx and unicorn

- This architecture can horizontally scale our as far the **database** at it's center
- Every other component can be scaled by adding **more of it**, to handle more traffic



TRAFFIC CLIMB IS RELENTLESS

And it keeps climbing, sending our servers into a tailspin…

WHAT ARE THE SYMPTOMS OF UNDER SCALING?

- Intermittent Outages
- Pages and mobile views take forever to load and render
- Saving objects takes forever, sometimes times out
- High level of user-dissatisfaction with the application reliability

FIRST SIGNS OF READ SCALABILITY PROBLEMS

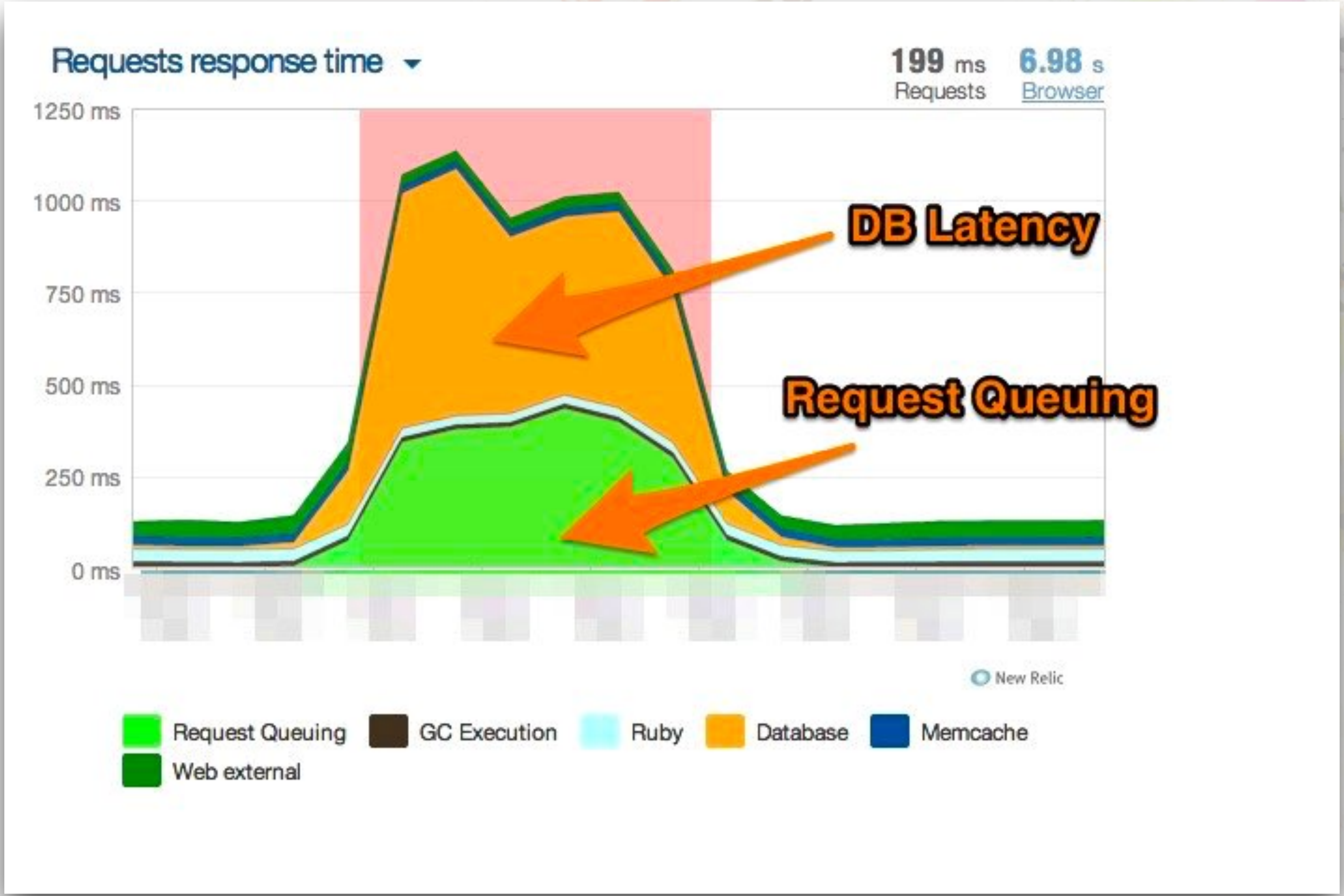
- Pages load slowly or timeout
- Users are getting 503 Service Unavailable
- Database is slammed (very high CPU or read IO)
- Some pages load (cached?), some don't

FIRST SIGNS OF WRITE SCALABILITY PROBLEMS

- Database write **IO is maxed out**, CPU is not
- Updates are waiting on each other, piling up
- Application "**locks up**", timeouts
- Replicas are **not catching up***

* More about replicas and how to observe them "catching up" is further down.

BOTH SITUATIONS MAY EASILY RESULT IN DOWNTIME



SCALING UP

LETS GET THE BASICS RIGHT FIRST:
CACHING 101

CACHING 101

- Anything that can be cached, should be cached
- Cache hit = many database hits avoided
- Hit rate of 17% **still saves DB hits**
- We can cache many types of things…
- Cache is cheap and fast (memcached)

CACHE MANY TYPES OF THINGS

```
git clone https://github.com/wanelo/compositor
git clone https://github.com/wanelo/cache-object
```

- **caches_action** in controllers is very effective
- **fragment** caches of reusable widgets
- we use gem **Compositor** for JSON API.
- We cache serialized object fragments, grab them from memcached using **multi_get** and merge them
- Our gem “**CacheObject**” provides very simple and clever layer within Ruby on Rails framework.

AJAXIFY: DO THIS EARLY, HARD TO ADD LATER.

- Personalization via AJAX, so controller actions can be cached entirely using **caches_action**

- Page returned unpersonalized, additional AJAX request loads personalization

Wanelo - Want, Need, Love
wanelo.com/trending
WANELO
MY FEED
TRENDING
Find products, stores and people
99+
Women
Men
Home
$
$$
$$$
Profile
Find Your Friends
Get the Bookmarklet
Settings

BUT EXPIRING CACHE IS NOT ALWAYS EASY

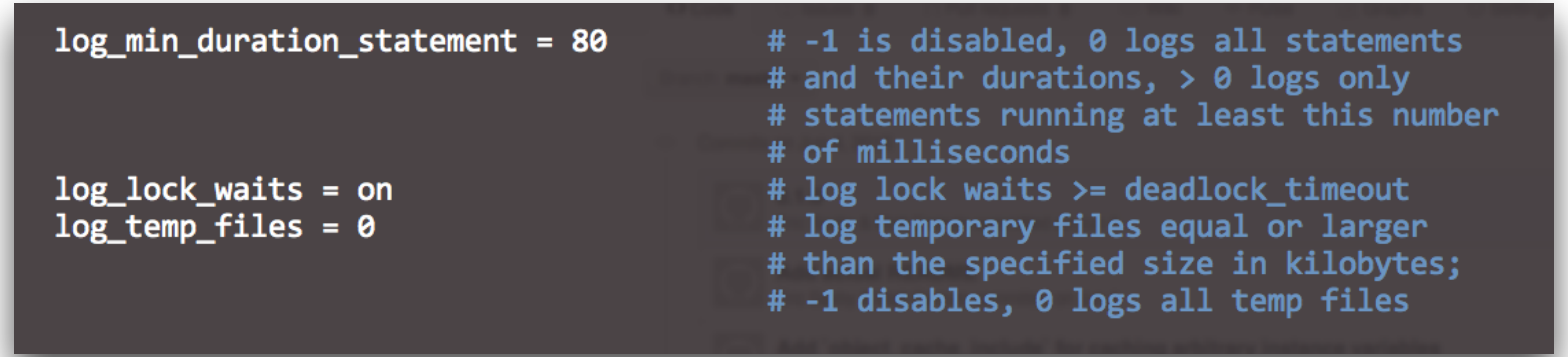
- Easiest way to expire cache is to wait for it to expire (by setting a TTL ahead of time). But that's not always possible (ie. sometimes an action requires wiping the cache, and it's not acceptable to wait)
- **CacheSweepers** in Rails help
- Can and should expiring caches in **background jobs** as it might take time.
- Can cache pages, fragments and JSON using **CDN also!**

SCALING UP

FINDING SLOW SQL

SQL OPTIMIZATION: LOG SLOW QUERIES

- Find **slow SQL** (>100ms) and either remove it, cache the hell out of it, or fix/rewrite the query
- Enable **slow query log** in postgresql.conf (as well as locks, and temp files). These are of the types of things you need to know about.

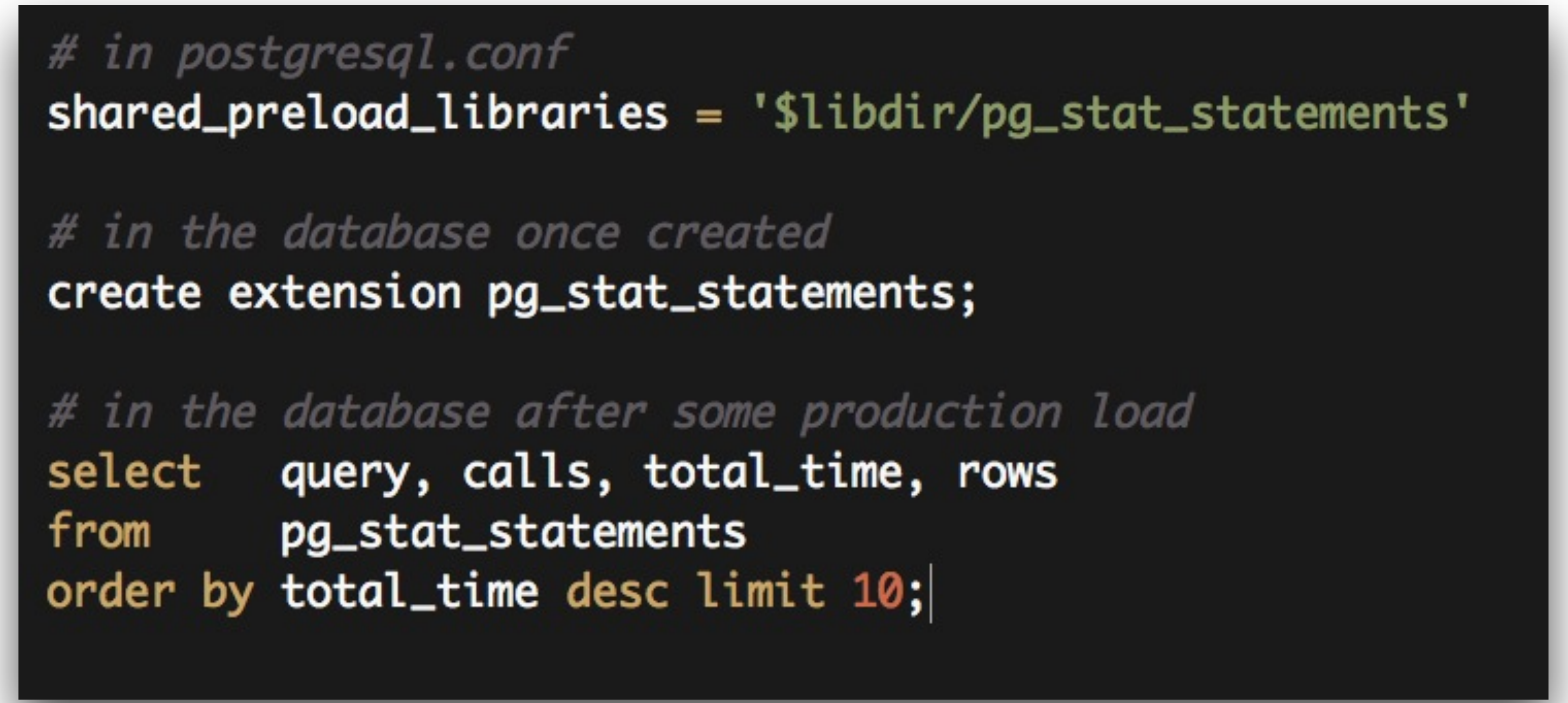


```
log_min_duration_statement = 80         # -1 is disabled, 0 logs all statements
                                        # and their durations, > 0 logs only
                                        # statements running at least this number
                                        # of milliseconds
log_lock_waits = on                     # log lock waits >= deadlock_timeout
log_temp_files = 0                      # log temporary files equal or larger
                                        # than the specified size in kilobytes;
                                        # -1 disables, 0 logs all temp files
```

TRACKING MOST TIME CONSUMING SQL

- The **pg_stat_statements** module provides a means for tracking execution statistics of all SQL statements executed by a server.

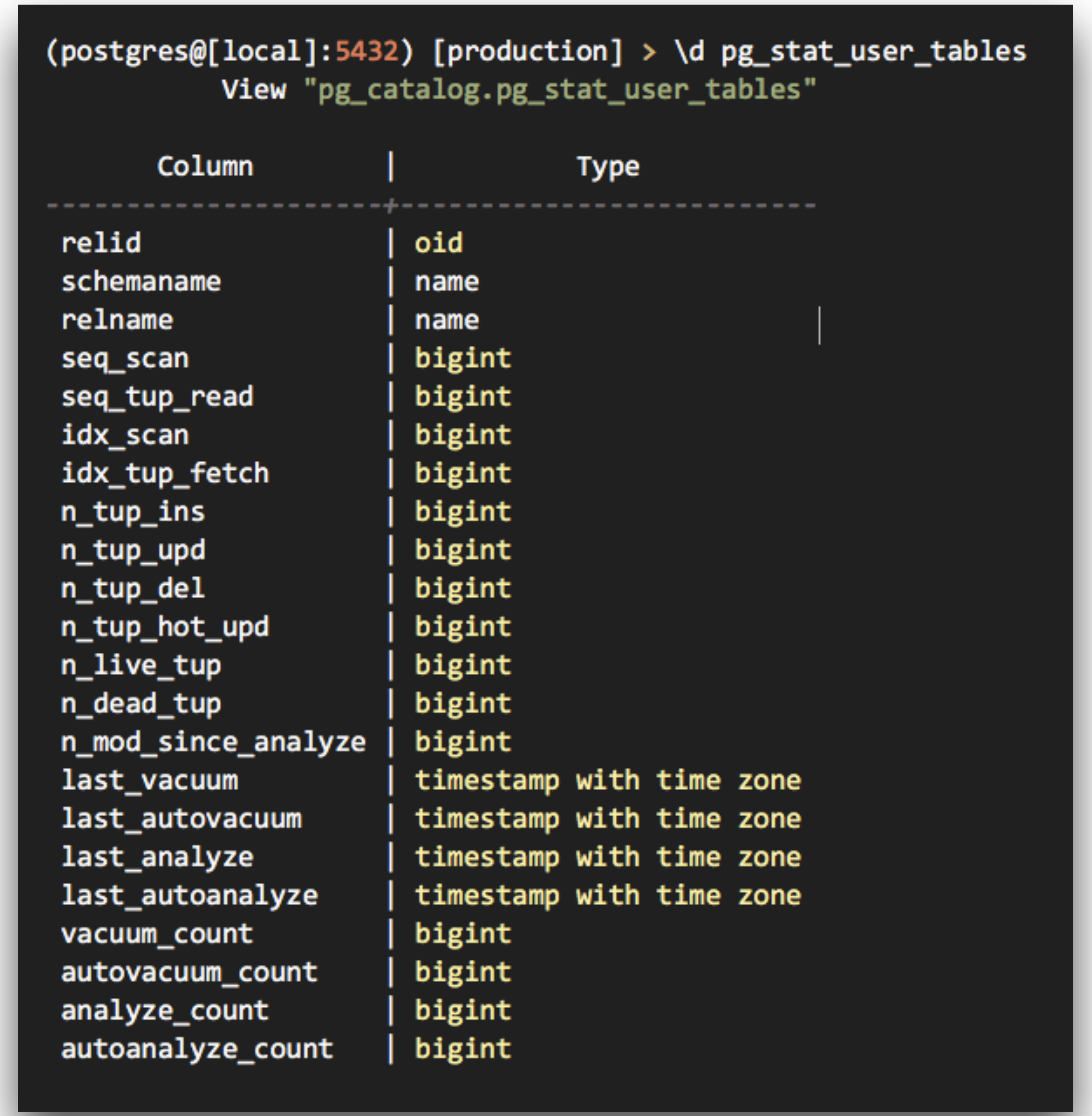
- The module must be loaded by adding **pg_stat_statements** to **shared_preload_libraries** in **postgresql.conf**, because it requires additional shared memory. This means that a server restart is



```
# in postgresql.conf
shared_preload_libraries = '$libdir/pg_stat_statements'

# in the database once created
create extension pg_stat_statements;

# in the database after some production load
select   query, calls, total_time, rows
from     pg_stat_statements
order by total_time desc limit 10;
```

FIXING SLOW QUERY:

`pg_stat_user_tables`



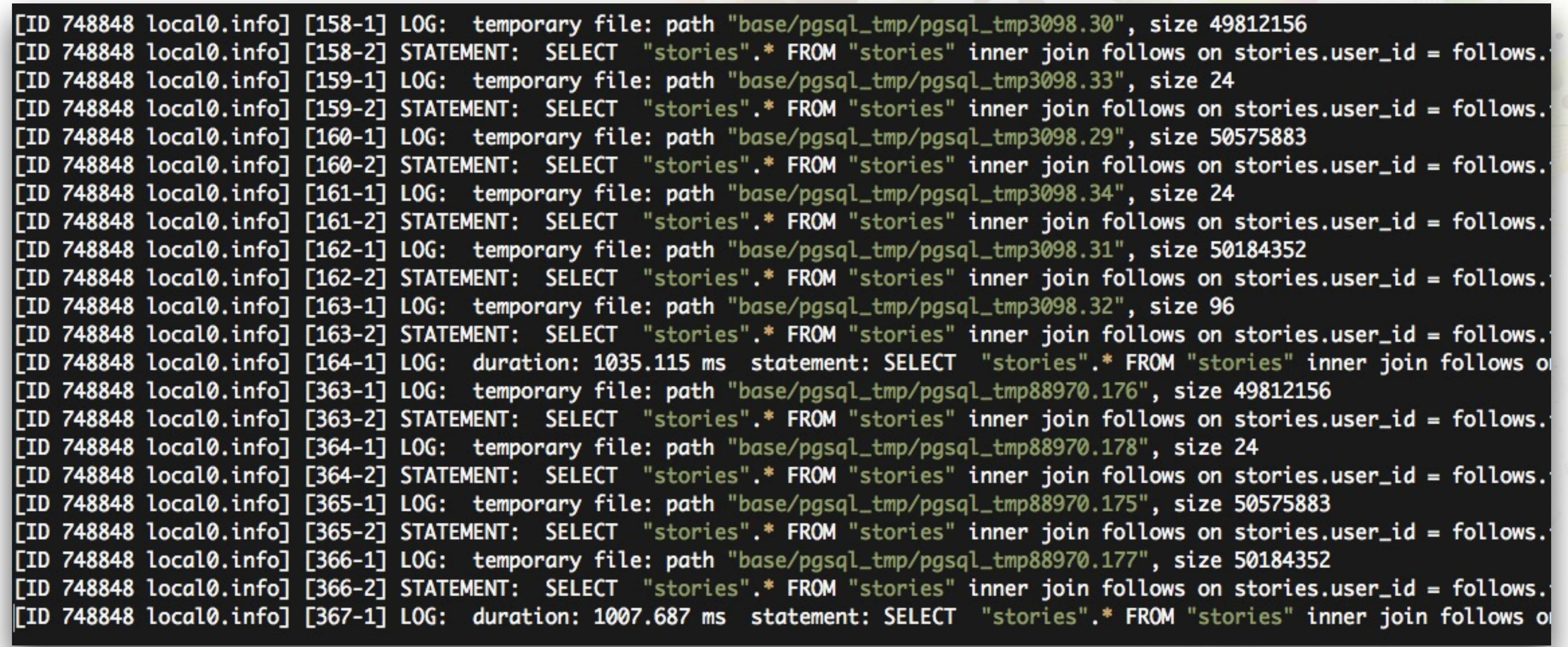
```
(postgres@[local]:5432) [production] > \d pg_stat_user_tables
           View "pg_catalog.pg_stat_user_tables"

       Column        |           Type
---------------------+--------------------------
 relid               | oid
 schemaname          | name
 relname             | name
 seq_scan            | bigint
 seq_tup_read        | bigint
 idx_scan            | bigint
 idx_tup_fetch       | bigint
 n_tup_ins           | bigint
 n_tup_upd           | bigint
 n_tup_del           | bigint
 n_tup_hot_upd       | bigint
 n_live_tup          | bigint
 n_dead_tup          | bigint
 n_mod_since_analyze | bigint
 last_vacuum         | timestamp with time zone
 last_autovacuum     | timestamp with time zone
 last_analyze        | timestamp with time zone
 last_autoanalyze    | timestamp with time zone
 vacuum_count        | bigint
 autovacuum_count    | bigint
 analyze_count       | bigint
 autoanalyze_count   | bigint
```

- Run explain plan to understand how DB runs the query using "**`explain analyze <query>`**".
- Are there adequate indexes for the query? Is the database using appropriate index? Has the table been recently **analyzed**?
- Can a complex join be simplified into a **subselect**?
- Can this query use an index-only scan?
- Can a column being sorted on be added to the index?
- What can we learn from watching the data in the two tables **`pg_stat_user_tables`** and **`pg_stat_user_indexes`**?
  - We could discover that the application is doing many sequential scans, has several unused indexes, that take up space and slow down "inserts" and much more.

AN EXAMPLE

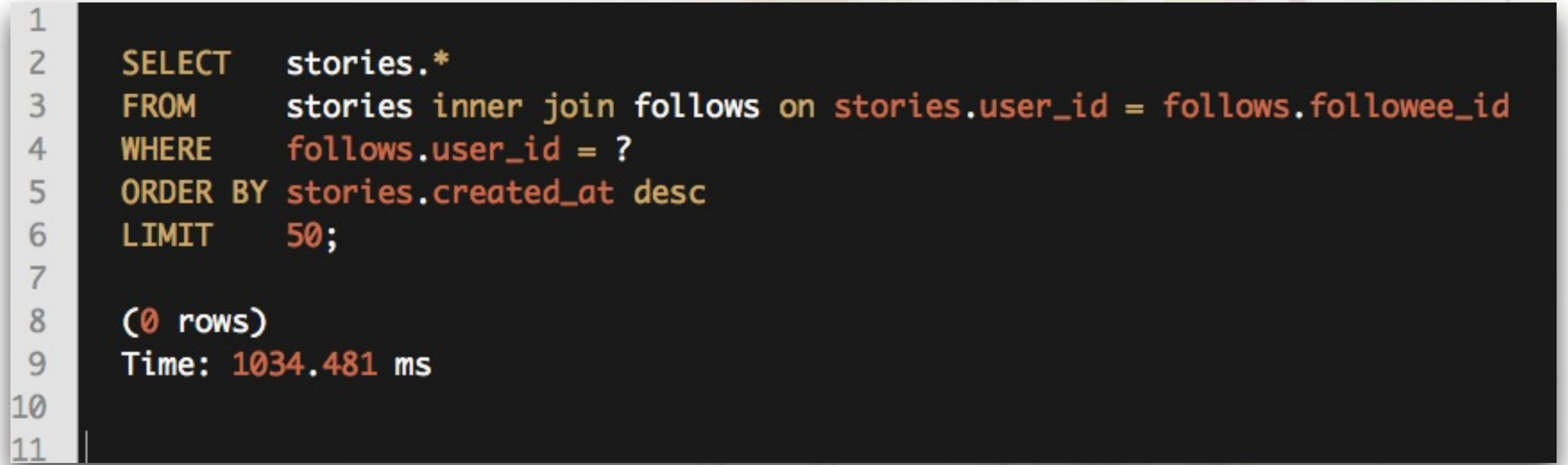
FIXING SLOW SQL

ONE DAY, I NOTICED LOTS OF TEMP FILES created in the postgres.log



```
[ID 748848 local0.info] [158-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp3098.30", size 49812156
[ID 748848 local0.info] [158-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [159-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp3098.33", size 24
[ID 748848 local0.info] [159-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [160-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp3098.29", size 50575883
[ID 748848 local0.info] [160-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [161-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp3098.34", size 24
[ID 748848 local0.info] [161-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [162-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp3098.31", size 50184352
[ID 748848 local0.info] [162-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [163-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp3098.32", size 96
[ID 748848 local0.info] [163-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [164-1] LOG:  duration: 1035.115 ms  statement: SELECT  "stories".* FROM "stories" inner join follows o
[ID 748848 local0.info] [363-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp88970.176", size 49812156
[ID 748848 local0.info] [363-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [364-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp88970.178", size 24
[ID 748848 local0.info] [364-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [365-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp88970.175", size 50575883
[ID 748848 local0.info] [365-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [366-1] LOG:  temporary file: path "base/pgsql_tmp/pgsql_tmp88970.177", size 50184352
[ID 748848 local0.info] [366-2] STATEMENT:  SELECT  "stories".* FROM "stories" inner join follows on stories.user_id = follows.
[ID 748848 local0.info] [367-1] LOG:  duration: 1007.687 ms  statement: SELECT  "stories".* FROM "stories" inner join follows o
```

LET'S RUN THIS QUERY...

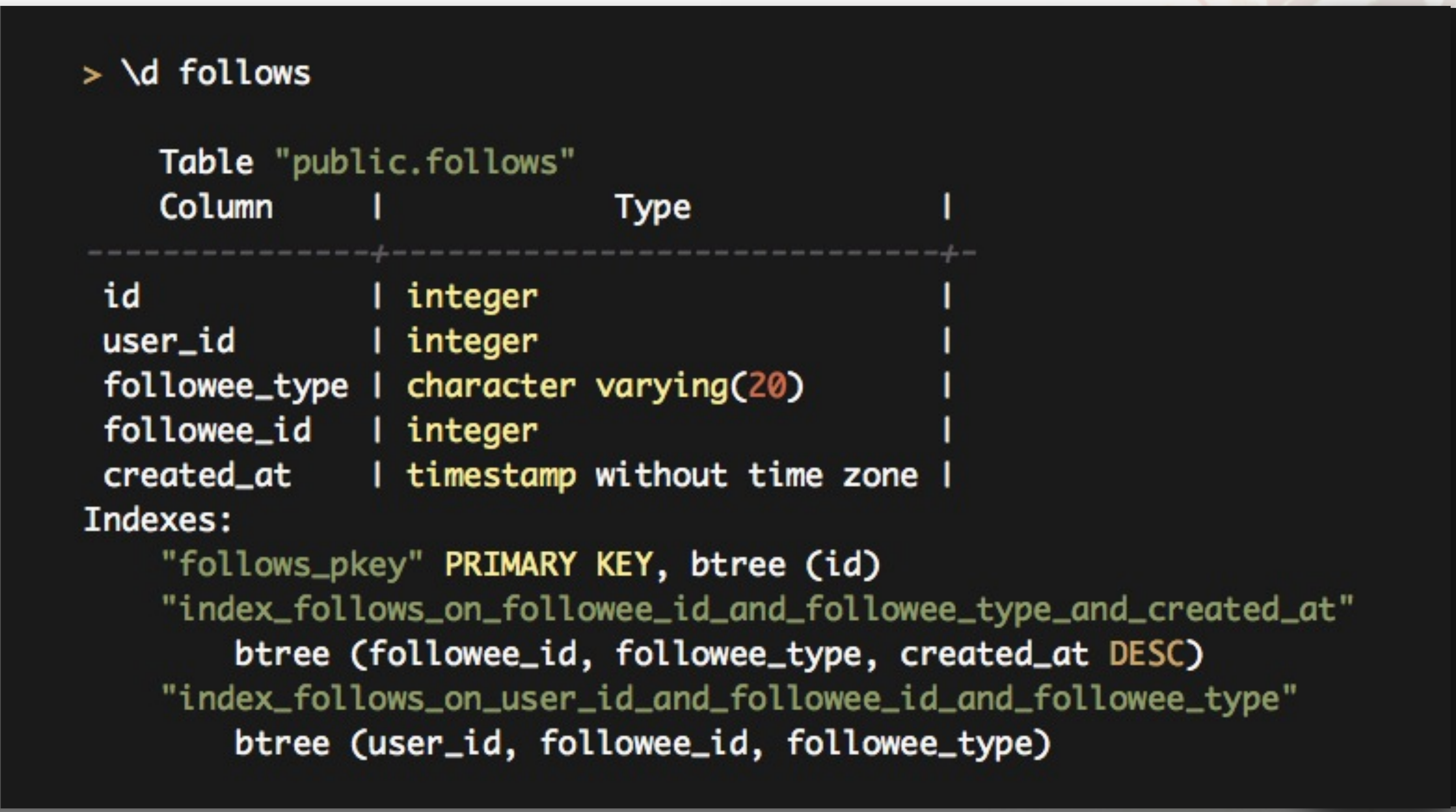


```
SELECT   stories.*
FROM     stories inner join follows on stories.user_id = follows.followee_id
WHERE    follows.user_id = ?
ORDER BY stories.created_at desc
LIMIT    50;

(0 rows)
Time: 1034.481 ms
```

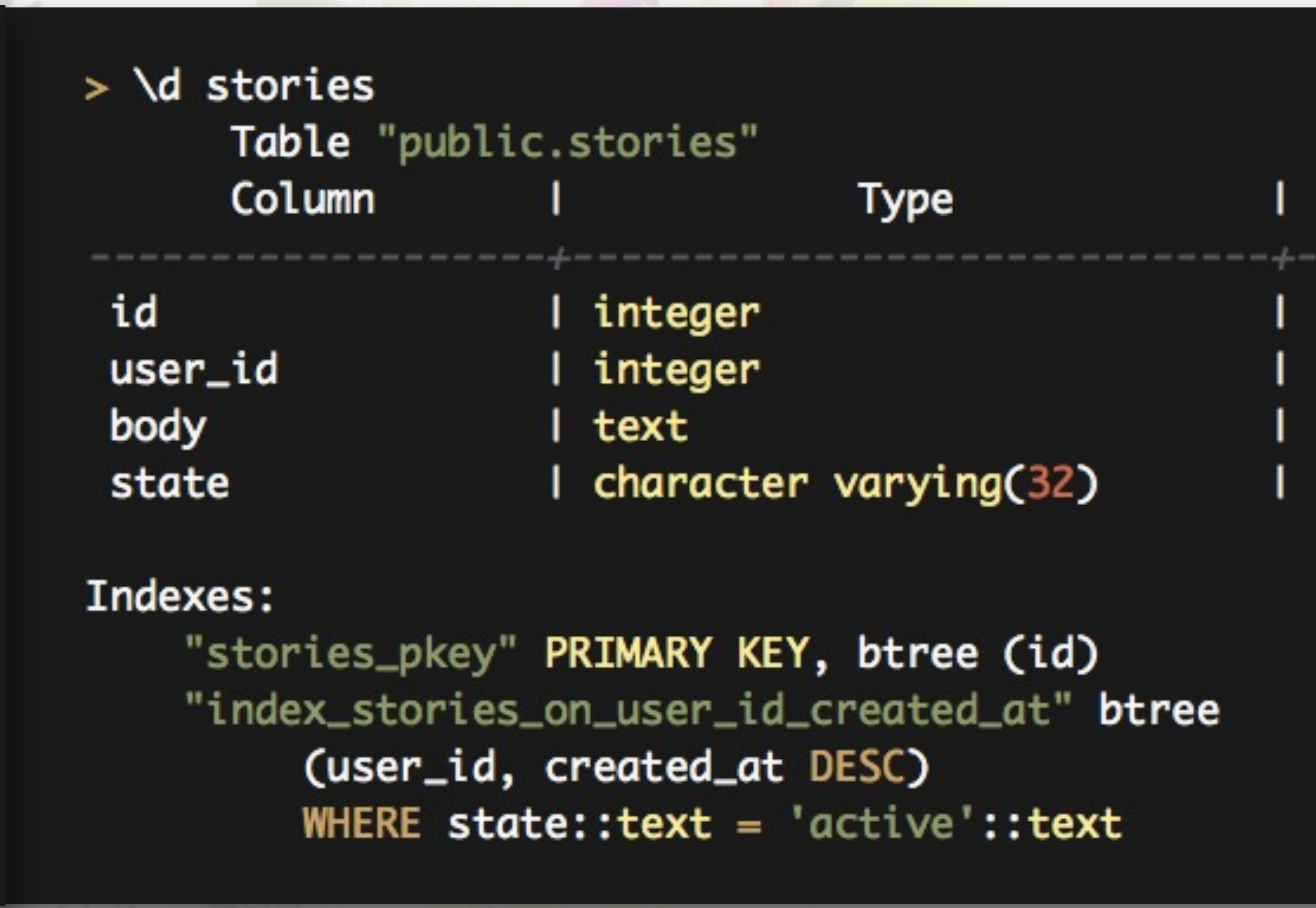
This join takes a whole second to return :(

FOLLOWS TABLE…

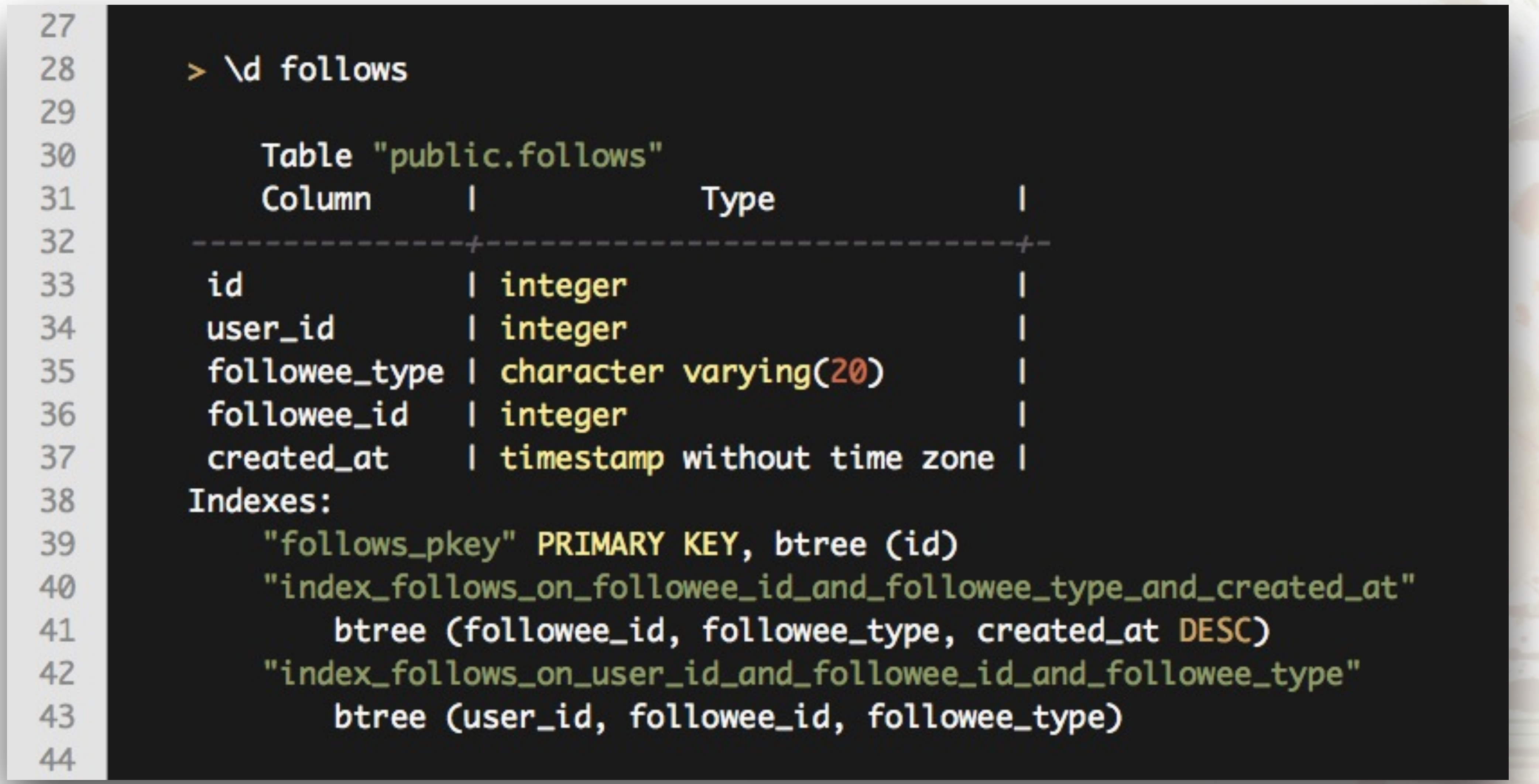
STORIES TABLE…



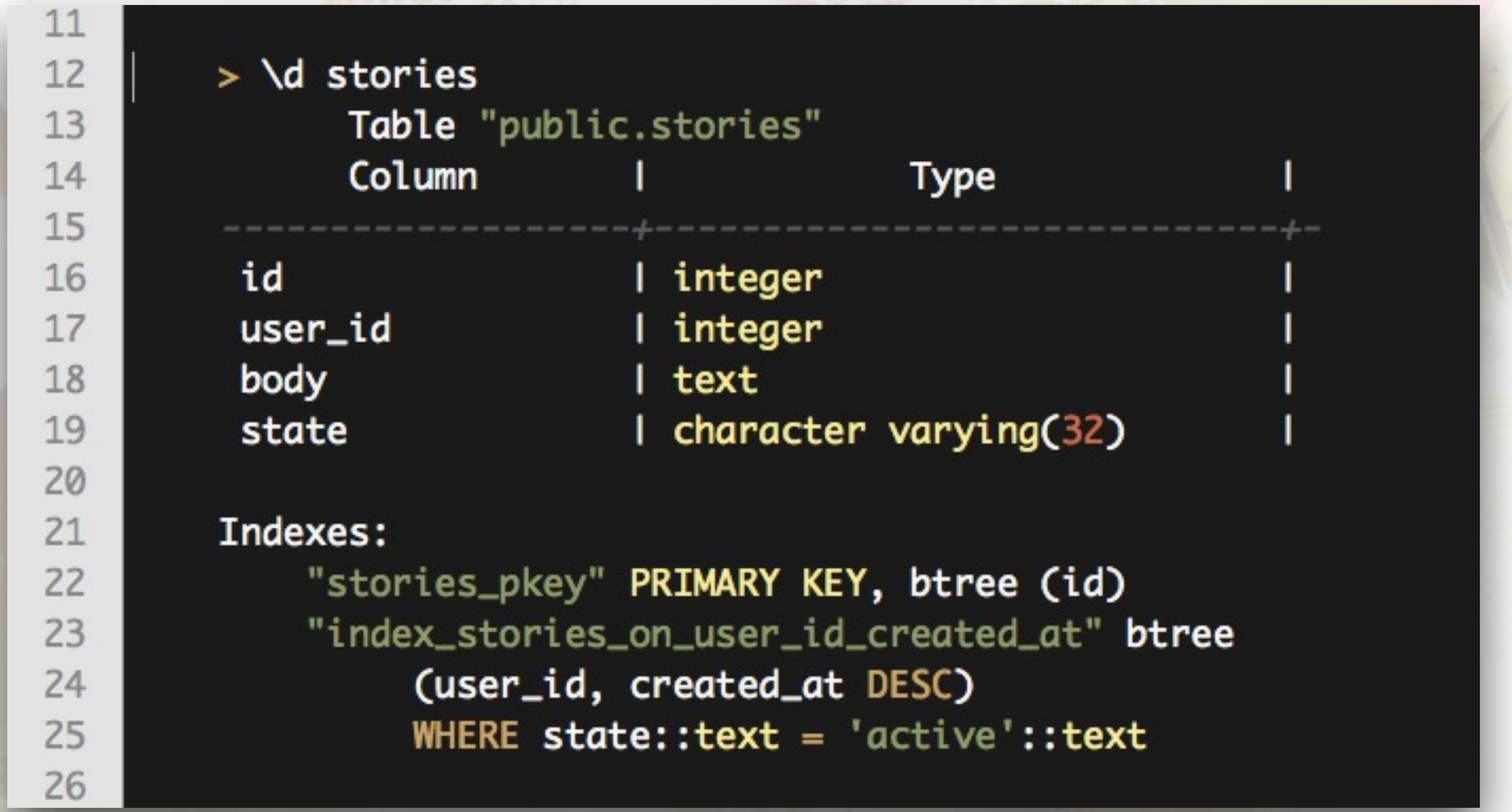
```
> \d follows

    Table "public.follows"
    Column     |            Type             |
---------------+-----------------------------+--
 id            | integer                     |
 user_id       | integer                     |
 followee_type | character varying(20)       |
 followee_id   | integer                     |
 created_at    | timestamp without time zone |
Indexes:
    "follows_pkey" PRIMARY KEY, btree (id)
    "index_follows_on_followee_id_and_followee_type_and_created_at"
        btree (followee_id, followee_type, created_at DESC)
    "index_follows_on_user_id_and_followee_id_and_followee_type"
        btree (user_id, followee_id, followee_type)
```



```
> \d stories
      Table "public.stories"
      Column      |          Type          |
------------------+------------------------+-
 id               | integer                |
 user_id          | integer                |
 body             | text                   |
 state            | character varying(32)  |

Indexes:
    "stories_pkey" PRIMARY KEY, btree (id)
    "index_stories_on_user_id_created_at" btree
        (user_id, created_at DESC)
        WHERE state::text = 'active'::text
```



```
> \d follows

   Table "public.follows"
   Column     |            Type             |
--------------+-----------------------------+-
 id           | integer                     |
 user_id      | integer                     |
 followee_type | character varying(20)      |
 followee_id  | integer                     |
 created_at   | timestamp without time zone |
Indexes:
    "follows_pkey" PRIMARY KEY, btree (id)
    "index_follows_on_followee_id_and_followee_type_and_created_at"
        btree (followee_id, followee_type, created_at DESC)
    "index_follows_on_user_id_and_followee_id_and_followee_type"
        btree (user_id, followee_id, followee_type)
```



```
> \d stories
       Table "public.stories"
       Column        |              Type              |
---------------------+--------------------------------+-
 id                  | integer                        |
 user_id             | integer                        |
 body                | text                           |
 state               | character varying(32)          |

Indexes:
    "stories_pkey" PRIMARY KEY, btree (id)
    "index_stories_on_user_id_created_at" btree
         (user_id, created_at DESC)
         WHERE state::text = 'active'::text
```

So our index is partial, only on state = 'active'

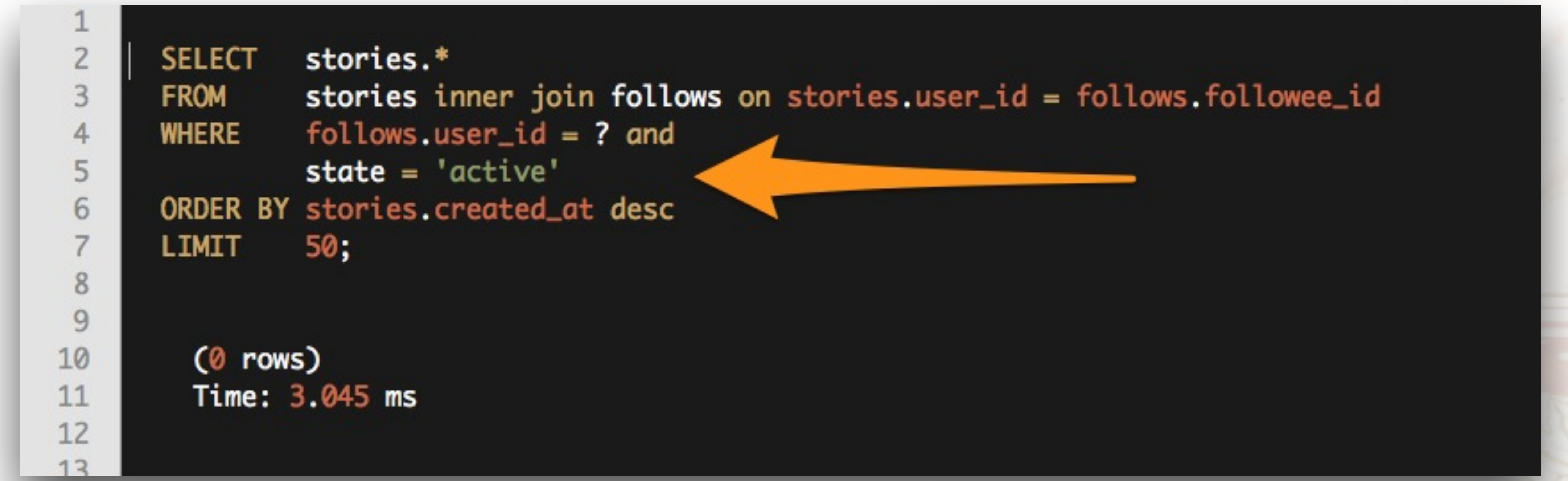
But the **state column** isn't used in the query at all! Perhaps it's a bug?

Regardless of whether this was intentional, the join results is a full table scan (called "sequential scan").

Sequential scan on a large table, in a database used by an OLTP application, is bad, because it "steals" the database cache from many other queries, because OS will now load these pages into the memory.

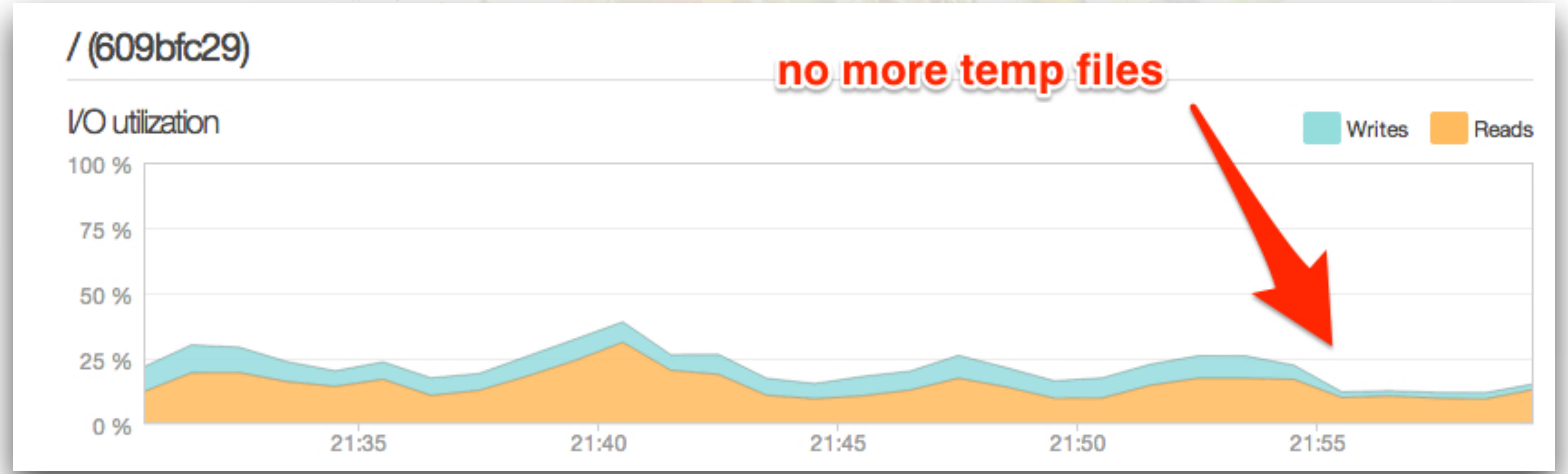
FIXING IT: LETS ADD STATE = "ACTIVE"

It was meant to be there anyway :)



```
SELECT   stories.*
FROM     stories inner join follows on stories.user_id = follows.followee_id
WHERE    follows.user_id = ? and
         state = 'active'
ORDER BY stories.created_at desc
LIMIT    50;


  (0 rows)
  Time: 3.045 ms
```

Now query takes 3ms instead of 1000ms, and the IO on the server drops significantly according to this NewRelic graph:



THERE IS A LOT MORE...

Part II of this presentation talks about:

- ➤ Setting up and using read replicas
- ➤ Choosing adequate hardware and tuning it
- ➤ Moving high-throughput tables out of the database, eg event collection
- ➤ Tuning PostgreSQL & File System
- ➤ Buffering updates to popular rows such as updating rails counters
- ➤ Schema tricks for high scale
- ➤ Vertical sharding
- ➤ Horizontal sharding & microservices

Thanks!

Blog:

- https://kig.re

Music:

- https://soundcloud.com/leftctrl
- https://soundcloud.com/polygroovers

twitter.com/kig
github.com/kigster
linkedin.com/in/kigster
slideshare.net/kigster