FROM OBVIOUS TO INGENIOUS

INCREMENTALLY SCALING WEB APPLICATIONS ON POSTGRESQL

THIS IS AN UPDATED AND REFRESHED VERSION V2.0 OF THE ORIGINAL TALK PRESENTED AT SF POSTGRESQL USER GROUP, AND TITLED
"**12-STEP PROGRAM FOR POSTGRESQL-BASED WEB APPLICATIONS PERFORMANCE**"
THE LOCATION OF THIS POPULAR SLIDESHARE IS AT THE FOLLOWING LINK.

@kigster

@kig

@kig

Konstantin Gredeskoul, CTO, Wanelo.com

WANELO

PGConf Silicon Valley 2015

DATA STORE TYPES: **OVERVIEW**

- **Relational Databases:**
  PostgreSQL, MySQL, Oracle, SQL Server have been around for decades. They are flexible, performant, and widely supported.

  Relational DBMSes represent massive **81%** of all data stores surveyed by db-engines.com.

- **Document Stores:**
  MongoDB, CouchDB, Amazon DynamoDB, Couchbase

- **Key Value Stores:**
  Redis, Memcached, Riak, DynamoDB

- **Wide Column Stores:**
  Cassandra, HBase, Accumulo, Hypertable

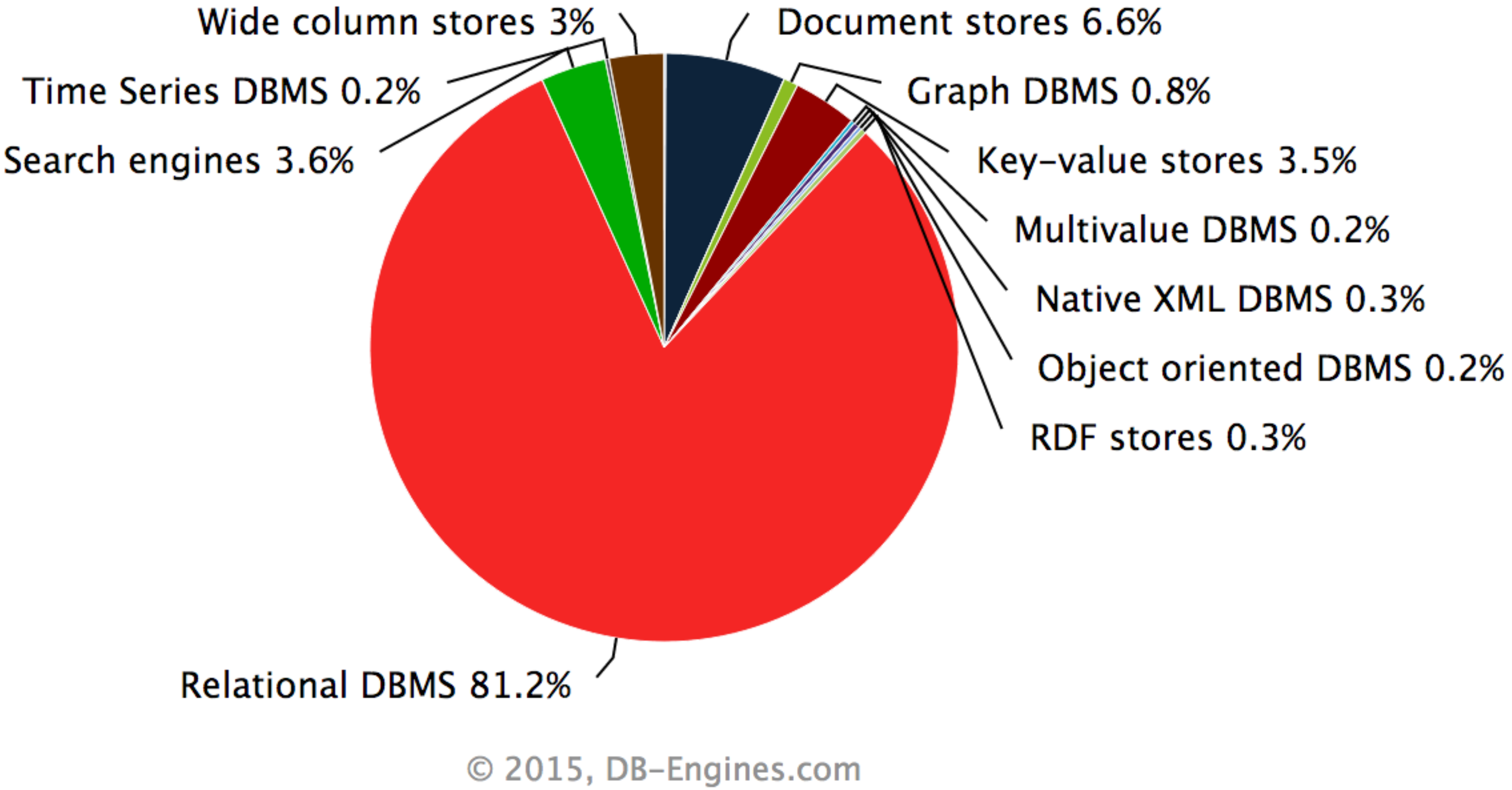
- **Search Engines:**
  Solr, Elasticsearch, Splunk, Sphinx

- **Graph DBMS:**
  Neo4j, Titan, OrientDB

- **Time Series DBMS:**
  RRDtool, InfluxDB, Graphite.

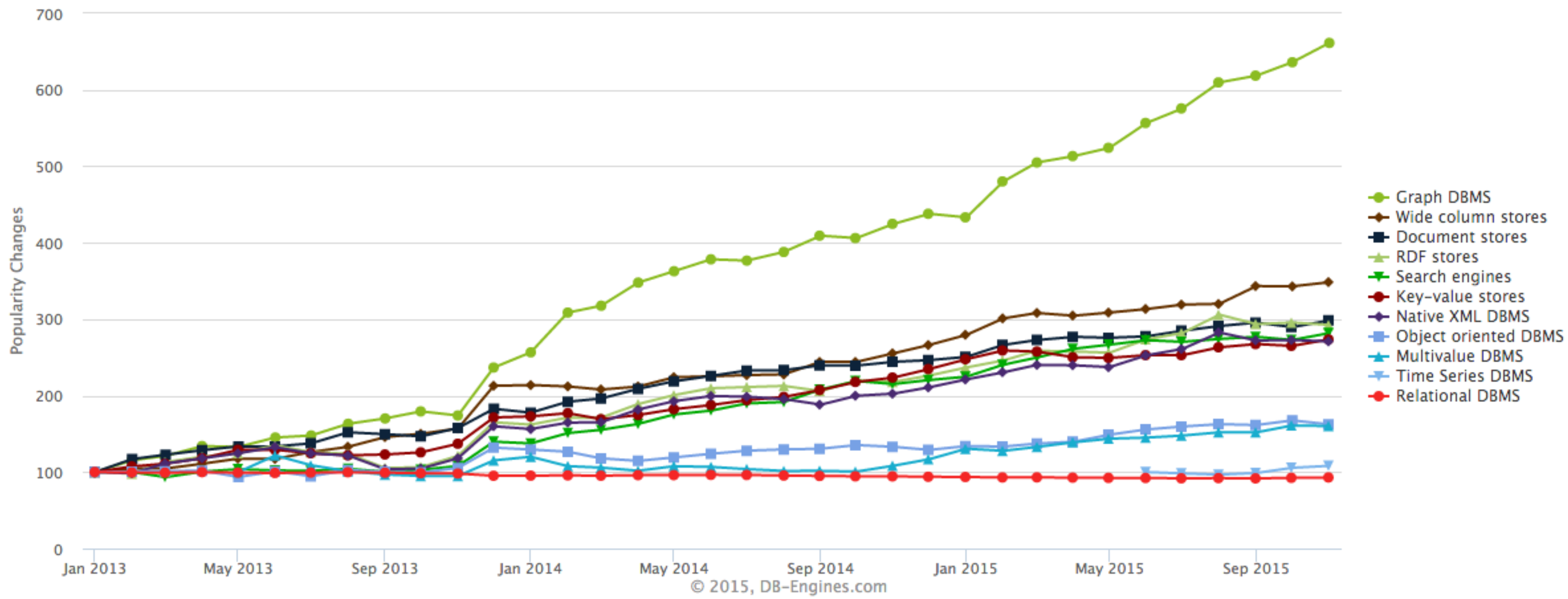
  - Also exist: **RDF stores, Object-Oriented, XMLDB, Content Stores, Navigational DBMS**

DATA STORE TYPES: **MARKET SHARE**



DATA STORE TYPES: **CHANGE OVER TIME**

- Even though Graph DBMSes show the largest increase over time, they account for mere 0.8% of the total market share.



WEB APPS **OF DISTINCTLY DIFFERENT** TYPES

- **OLTP (Online Transaction Processing)**
  These are the typical web applications with **massive number of users**, performing various operations **concurrently**.

  Examples include online stores, social networks, google, etc. – are all OLTP applications. They require huge throughput of small transactions, required to be as fast as possible (otherwise users leave), and achieve the speed by having most of the "live" data cached.

- **Analytics**
  Small number of users (analysts) running very long-running reports across the entire data sets, that are typically much much larger than what would fit into RAM

- **Backend Processing**
  Somewhere in between the two, backend applications are typically processing large amounts of data for either import/export, transformations, synchronization and updates.

  These apps have almost no users (just admins), and push their data store to the limit. But since it does not have real users, speed of operations only affects application's overall throughput.

CASE IN POINT: APP **ASSUMPTIONS**

- In this presentation we'll make an assumption that we are building a massive concurrent multi-user **web application** (using ruby, ruby on rails, and other tools).
- This type of load is typically called "**OLTP**", meaning online transaction processing.
  - In plain English, this means that our database will be getting high throughput of concurrent requests on behalf of each session for each user working with an application at any given time.
  - Sessions may be initiated from the web by users or admins, or from the mobile app by mobile users.
  - Users want their app to be very responsive, and they leave when it isn't. Therefore OLTP applications need to be both fast (performance) and support many users (scalable).
- We are **NOT** going to be addressing the needs of **Analytics** or **Backend Processing Applications**, which have only a few or no users.

1. SCALABILITY AND

- What to choose for data store on a new application?
- Relational data model
- Structured vs Unstructured
- Scalability vs performance
- Understanding latency
- Foundations of web architecture
- First signs of scaling issues:
  - Too many database reads
  - Too many database writes

2. SCALING UP LOW HANGING

1. Caching
2. Fixing slow SQL
   - Optimization example
3. Setting up streaming replication, and doing read/write splitting
4. Upgrading hardware
5. Where not to use PostgreSQL
6. Do not store append-only event data in PostgreSQL

3. SCALING OUT IN THE

7. Tune DB & filesystem
8. Buffer and serialize frequent updates to the same row
9. Optimize schema for scale
10. Vertically shard busy tables
11. Move vertically shared tables behind micro-services
12. Horizontally shard data-store behind micro-service.

Conclusions and final thoughts.

Thanks & contact Info.

PART 1

SCALABILITY IN CONTEXT

PERFORMANCE VS SCALABILITY, LATENCY, CASE STUDY, WEB ARCHITECTURES

STARTING NEW APPLICATION, **WHAT TO USE?**

- If your application starts with a small data-set, then **relational database will give you the most flexibility,** while enabling high productivity software like Rails.
- PostgreSQL is not only a safe choice, but a great choice for new applications due to it's unprecedented **versatility, speed, and reliability.**
- Where it falls short, compared to some of the more specialized storage software, is **massive horizontal scalability.**

STRUCTURED DATA VS UNSTRUCTURED

- Overwhelming majority of common web application data is **structured**. As in, we know pretty well it's properties (columns) in advance – such as **user.firstname**, etc.

- Structured data is very effectively represented by the **relational model** developed in 1969 by E.F. (Ted) Codd.

- Relational model is **mathematically complete**, and in practice excellent for mapping almost any domain, with very few exceptions – in the areas of **time series, directional graphs, and full-text search.**

STRUCTURED DATA VS **UNSTRUCTURED**

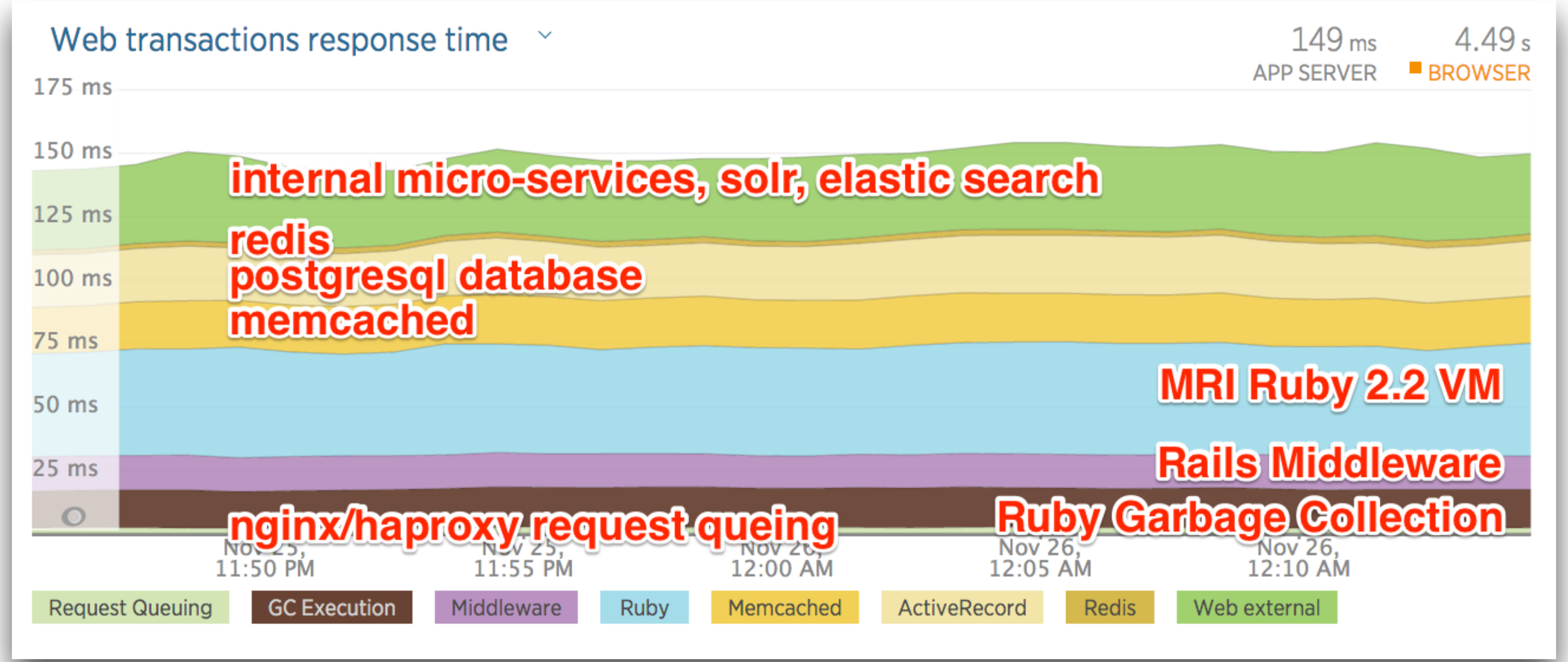
- For the last few years there has been a lot of hype surrounding "**document**" databases, in particular MongoDB.
- MongoDB marketing appears to be set to "kill" (or replace) relational databases. Not only this will very unlikely to occur, but it frames the discussion in a very wrong way: one OR the other, while down the road **it's likely to be both.**
- **PostgreSQL has been continually growing** in the area of non-structured capabilities: it now supports JSON, HSTORE and XML data types natively and very well.

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

PERFORMANCE: REDUCING LATENCY

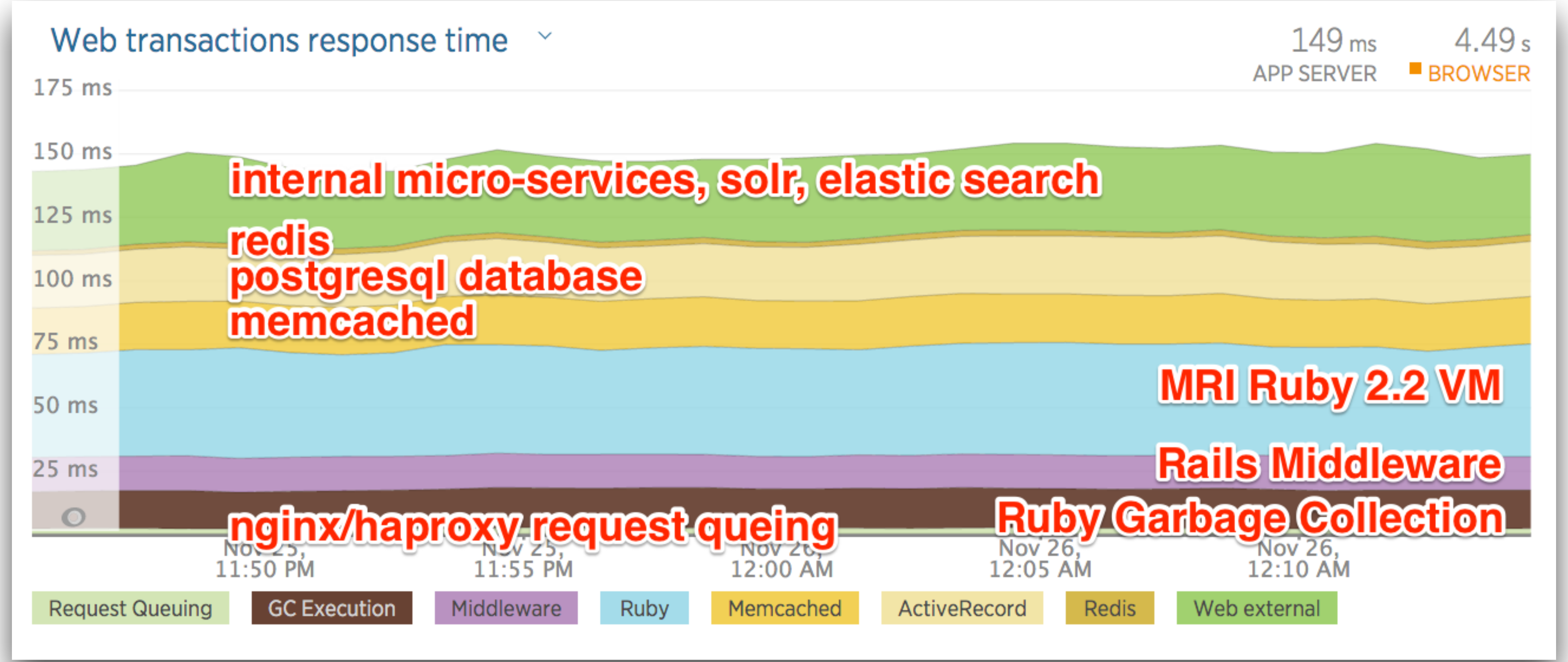
- If your app is high traffic (100K+ RPM) I recommend server latency of **100ms** or lower for web applications
- For fast internal HTTP services, that wrap data-store – **5-10ms** or lower



Graph credits: © NewRelic, Inc.

ZOOM INTO **SERVER LATENCY**

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



Graph credits: © NewRelic, Inc.

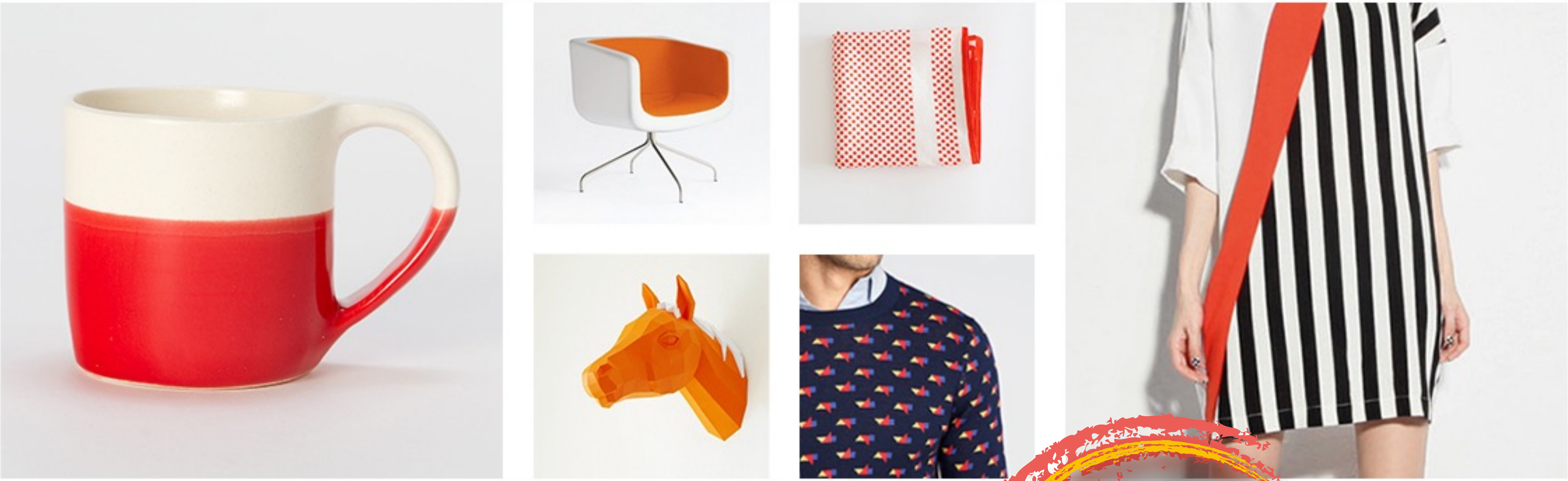
WANELO CASE STUDY

A regular mall has 150 stores, but Wanelo has **550,000** stores which include all the big brands you know, as well as tiny independent boutiques.

Founded in 2010, **Wanelo** ("wah-nee-loh," from Want, Need, Love) is a **mall on your phone.** It helps you find, bookmark ("save") the quirkiest products in the online universe.

In 2013 traffic to Wanelo went from 2,000 requests per minute, to 250,000 in about six months period of a true exponential hyper-growth.

WANELO



550K
STORES

30M
PRODUCTS

3B
SAVES

EARLY **ENGINEERING GOALS**

- Move as fast as possible with product development. We call it "**Aggro-Agile**"™
- Scale the app as needed, but **invest into small and cheap things today, that will save us a lot more time tomorrow**
- Stay ahead of the growth curve by closely monitoring application.
- Keep overall costs low (stay lean, keep app fast)
- Spend $$ where it matters the most: to save precious and expensive developer time
- As a result, we took advantage of a large number of open source tools and paid services, **which allowed us to move fast.**

TALKING ABOUT A "STACK" IS **POINTLESS**

UNLESS YOU HAVE HOURS TO KILL

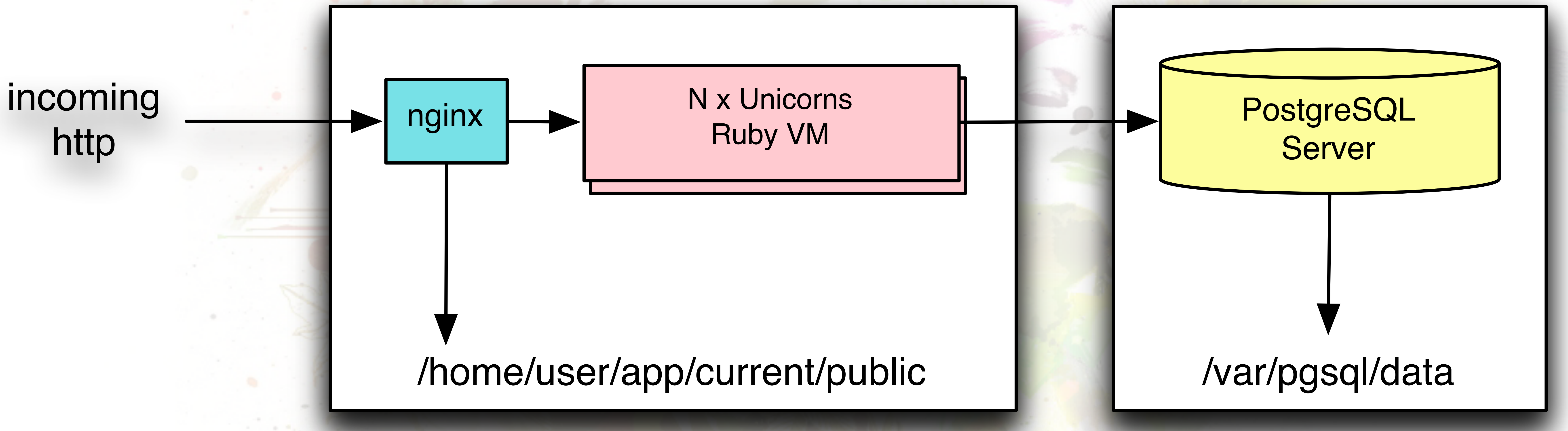
- MRI Ruby, Sinatra, Ruby on Rails, Sidekiq
- PostgreSQL, RabbitMQ, Solr, Redis, Twemproxy, haproxy, pgbouncer, memcached, nginx, ElasticSearch, AWS S3
- Joyent Public Cloud (JPC), Manta Object Store, SmartOS (ZFS, ARC Cache, SMF, Zones, dTrace)
- DNSMadeEasy, Gandi.net, SendGrid, SendWithUs, Fastly
- SiftScience, LeanPlum, Crashalytics, MixPanel, Graphite
- AWS RedShift
- Circonus, NewRelic, statsd, PagerDuty

FOUNDATIONS
OF MODERN WEB ARCHITECTURE

FOUNDATIONAL **TECHNOLOGIES**

- programming language + framework (**RoR**)
- app server (we use **puma**)
- scalable web server in front (we use **nginx**)
- database (we use **postgresql**)
- hosting environment (eg, **AWS, Heroku, etc**)
- deployment tools (**capistrano**)
- server configuration tools (we use **chef**)
- many others, such as monitoring, alerting

LET'S REVIEW – SUPER **SIMPLE** APP



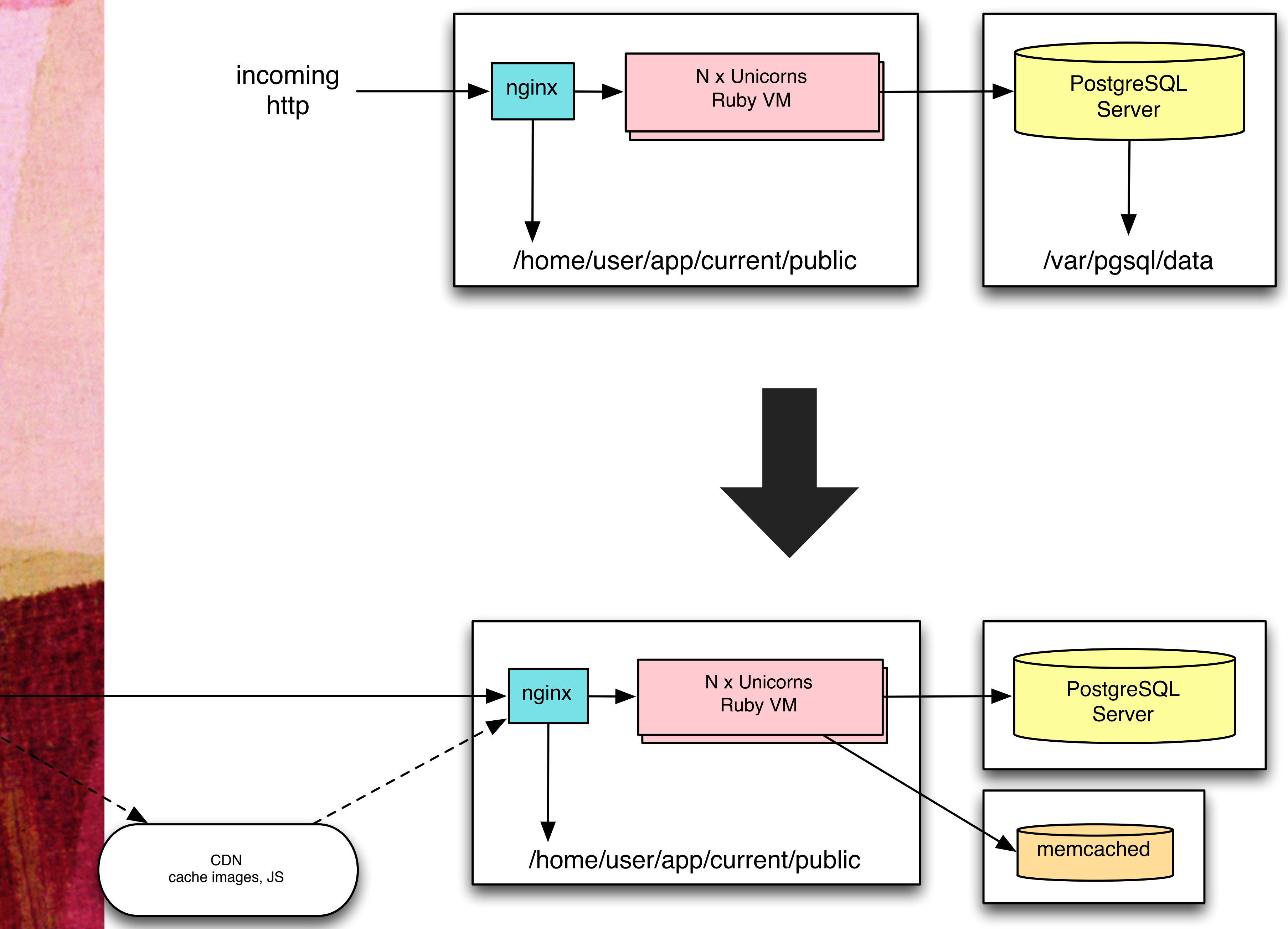
- no redundancy, no caching (yet)
- can only process N concurrent requests
- nginx will serve static assets, deal with slow clients
- web sessions probably in the DB or cookie

AJAXIFY: DO THIS EARLY, HARD TO ADD LATER.

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

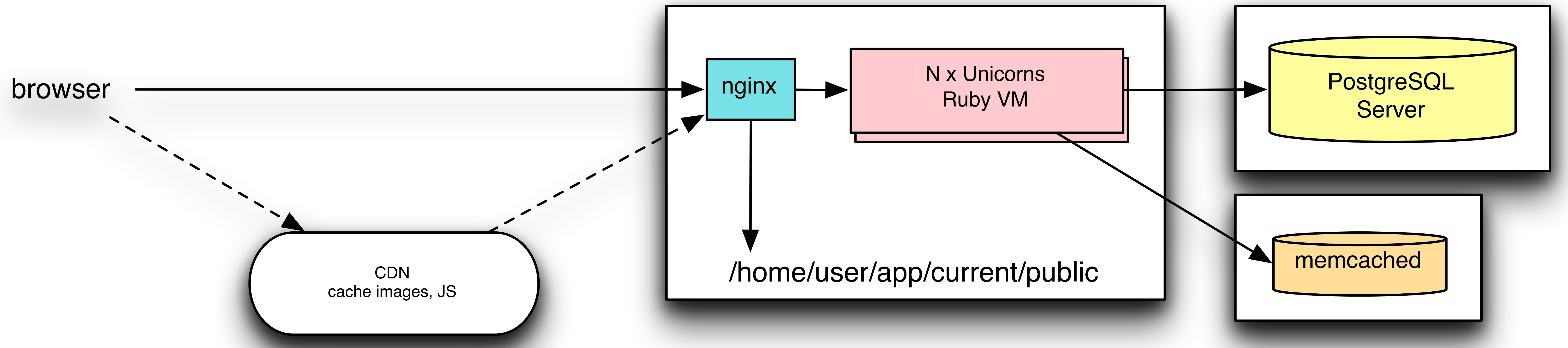
- Page returned unpersonalized, additional AJAX request loads personalization

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 **memcached-based web sessions**. Use sessions sparingly, assume transient nature
  - 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
- Setup **CDN** for asset_host and any user generated content. We use fastly.com

ADD CACHING: **CDN** AND **MEMCACHED**

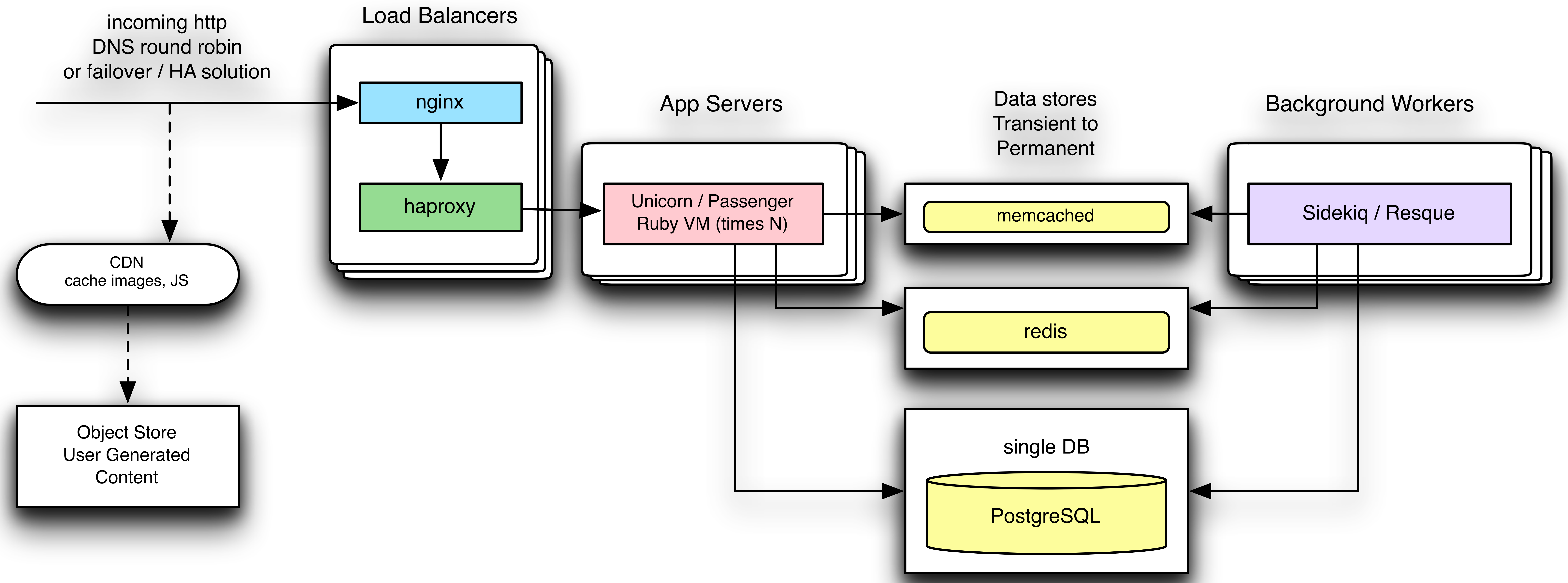


- geo distribute and cache your UGC and CSS/JS
- cache html and serialize objects in
- can increase TTL to alleviate load, if traffic

SIDENOTE: REMOVE **SINGLE POINT OF**

- **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…

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

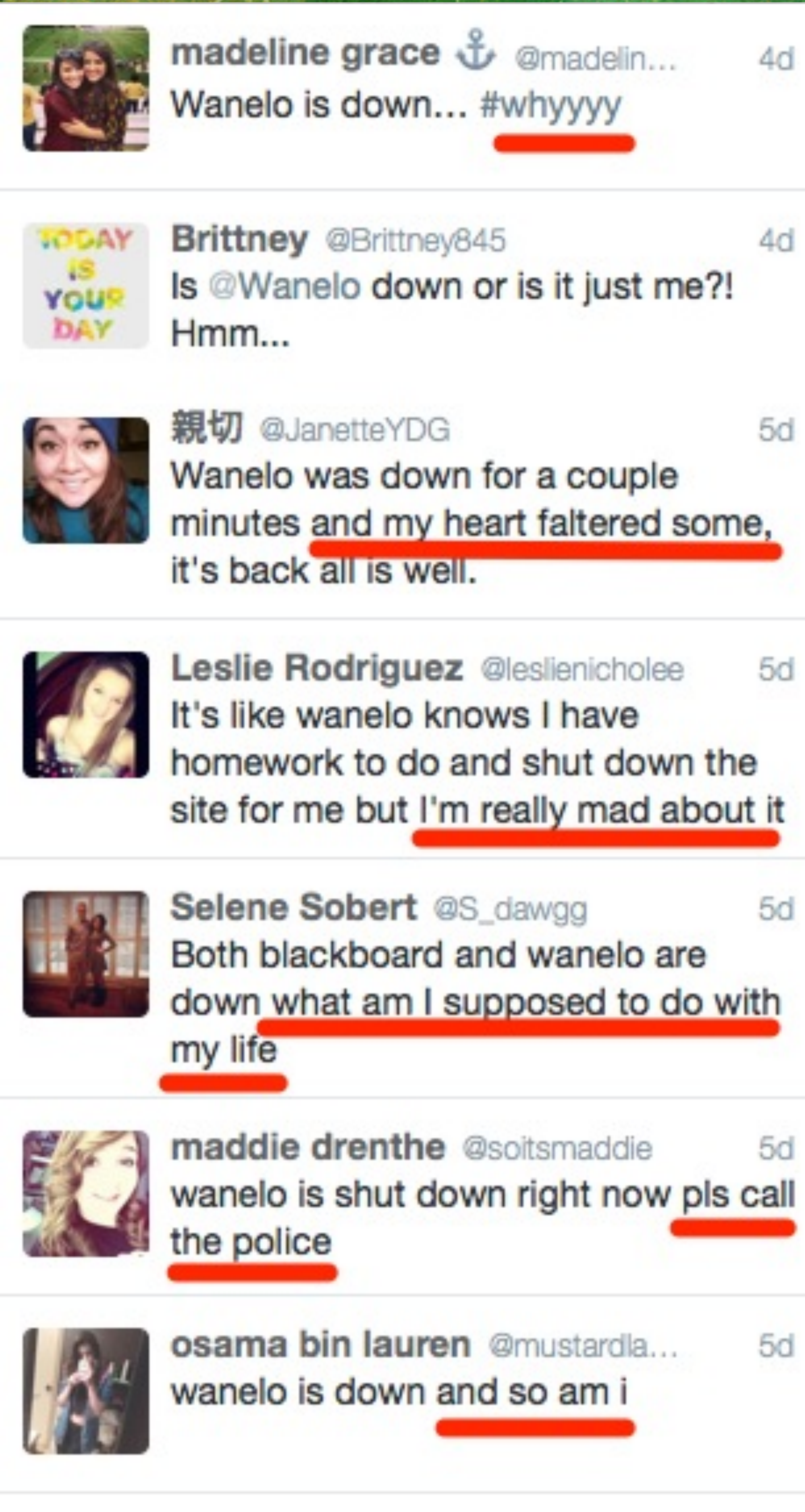
FIRST SIGNS OF **WRITE** SCALABILITY PROBLEMS

- 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***

BOTH SITUATIONS MAY EASILY RESULT IN **DOWNTIME**

Even though we achieved **99.99%** uptime in 2013, in 2014 we had a couple short downtimes caused by an overloaded (by too many read requests) PostgreSQL replica.

OUR USERS NOTICED IN SECONDS…



SCALING UP

1. THE MOST IMPORTANT THINGS FIRST: **CACHING**

CACHING **101**

- Anything that can be cached, should be
- 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.

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!**

MOBILE API, CDN AND CACHING TRICK

- All API responses that point to other API responses must always use fully qualified URLs (ie. **next_page**, etc)
- Multi-page grids can start to be fetched from: api.example.com
- Second and subsequent pages can be served from api-cdn.example.com
- If CDN is down, small change to configuration and mobile apps are sending all traffic to the source (api.example.com)

SCALING UP

2. FINDING AND OPTIMIZING **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 needed to add or remove the module.

```
# 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:

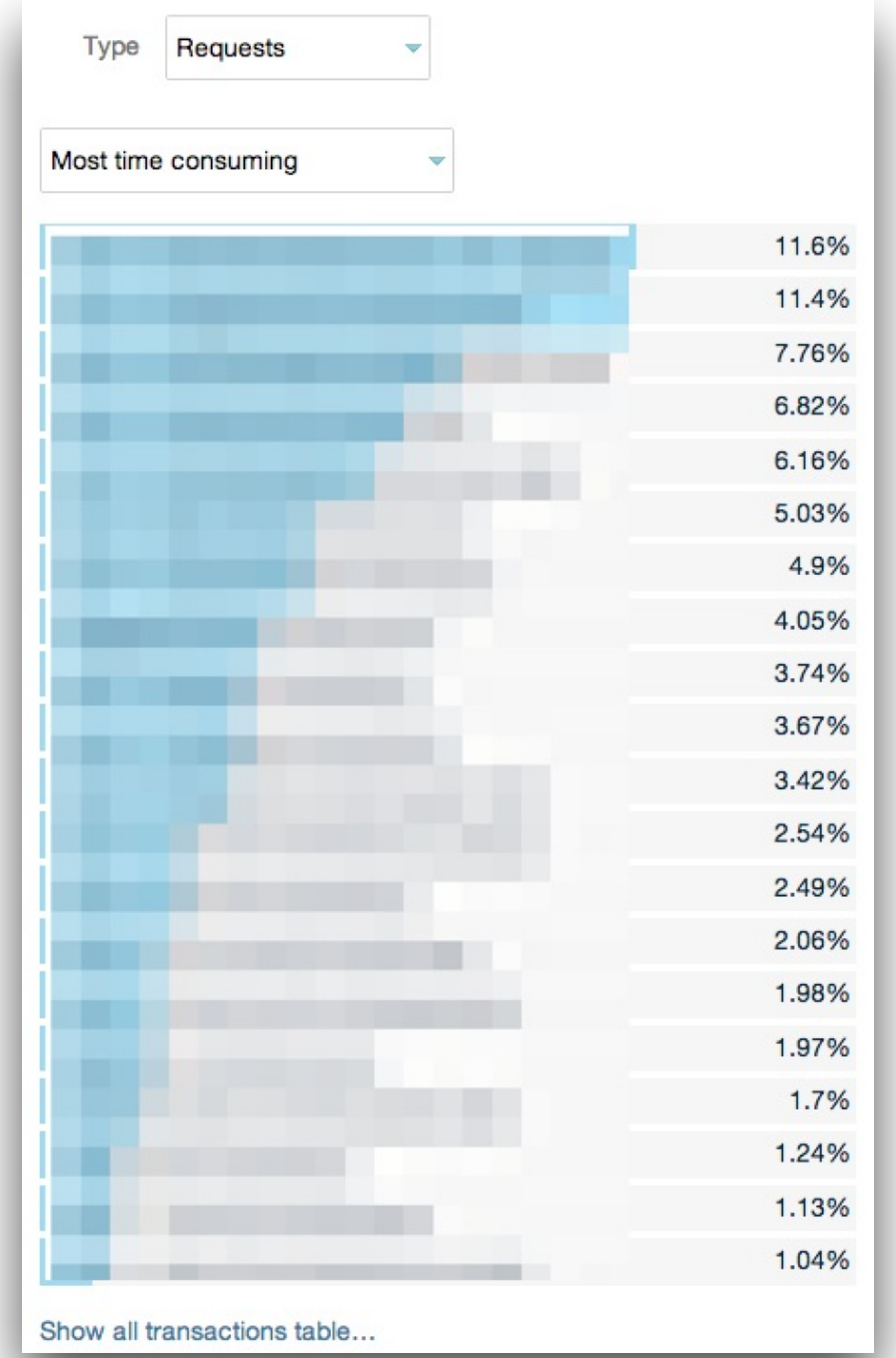
pg_stat_user_tables

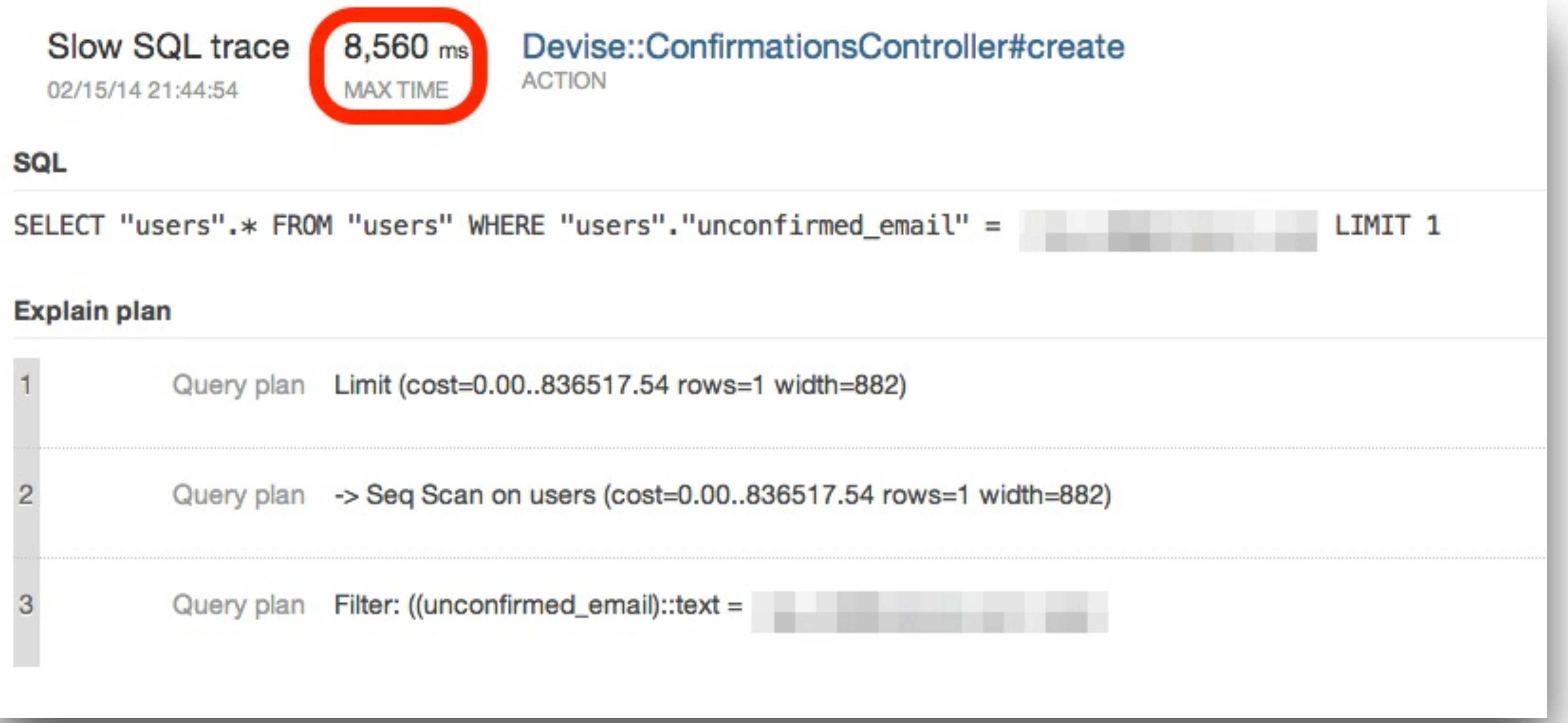
```
(postgres@[local]:5432) [production] > \d pg_stat_user_tables
          View "pg_catalog.pg_stat_user_tables"

       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
```

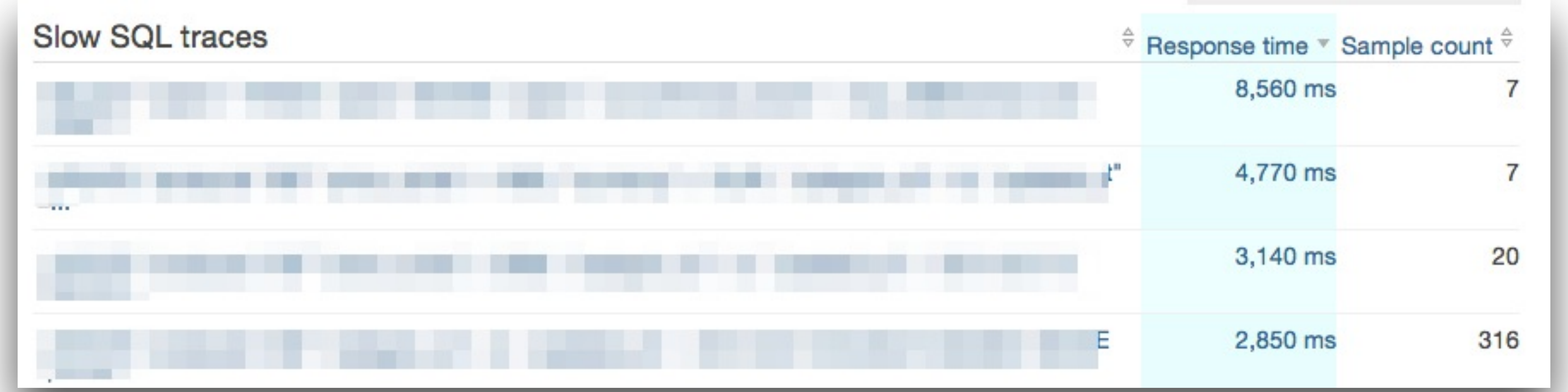
- Run explain plan to understand how DB runs the query using “**explain analyze <query>**”.
- Are there adequate indexes for the query? Is the database using appropriate index? Has the table been recently **analyzed**?
- Can a complex join be simplified into a **subselect**?
- Can this query use an index-only 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.

SQL **OPTIMIZATION**, CTD

Instrumentation software such as NewRelic shows slow queries, with explain plans, and time consuming transactions







FIXING A QUERY: **AN EXAMPLE**

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…

```
> \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)
```

STORIES TABLE…

```
> \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”).

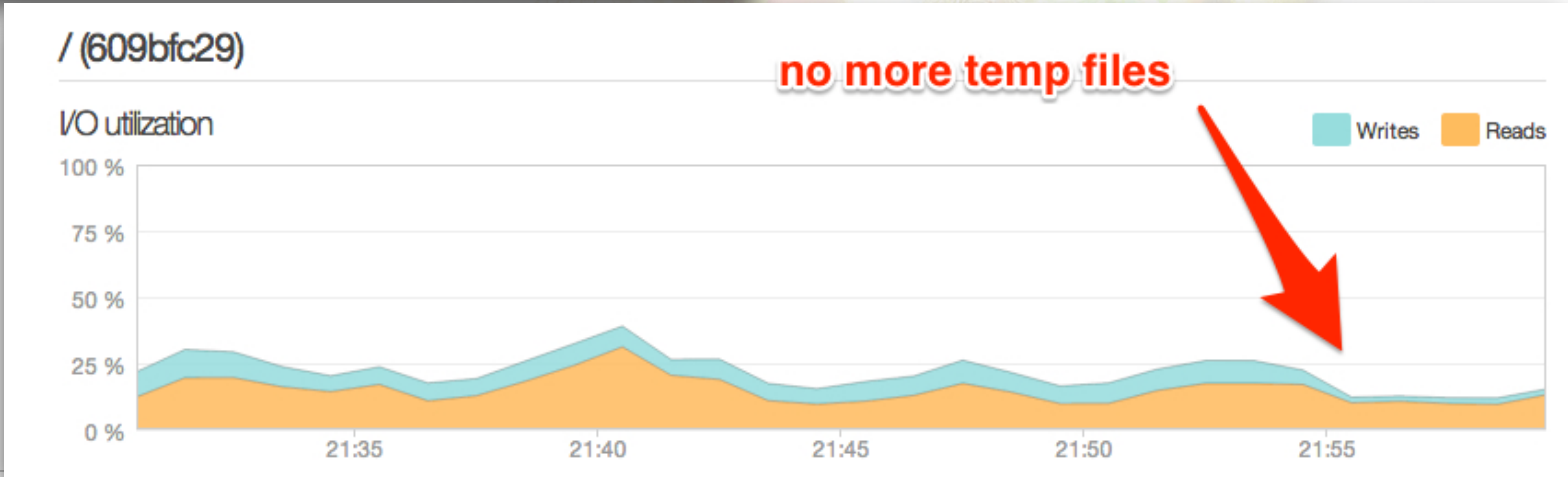
Sequential scan on a large table, in a database used by an OLTP application, is bad, because it “steals” the database cache from many other queries, because OS will now load these pages into the memory.

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:



SCALING UP

3. SETTING UP STREAMING **REPLICATION**

SCALE READS BY **REPLICATION**

```
hot_standby_feedback = 'on'
max_standby_streaming_delay = 30s
hot_standby = 'on'
wal_level = hot_standby
wal_sync_method = fsync
wal_buffers = 32MB
wal_writer_delay = 200ms
full_page_writes = on
```

These settings have been tuned for SmartOS and our application requirements (thanks PGExperts!)

- Version 9.3 and later setting up replication is **very easy**
- **postgresql.conf** (left) both the master & the replica
- So is electing a new master, and **switching replicas to a new timeline.**
- Each PG release seems to be making replication even easier.

REPLICATION 101: **WHERE ARE MY REPLICAS?**

Grab our nagios replication check here: https://github.com/wanelo/nagios-checks

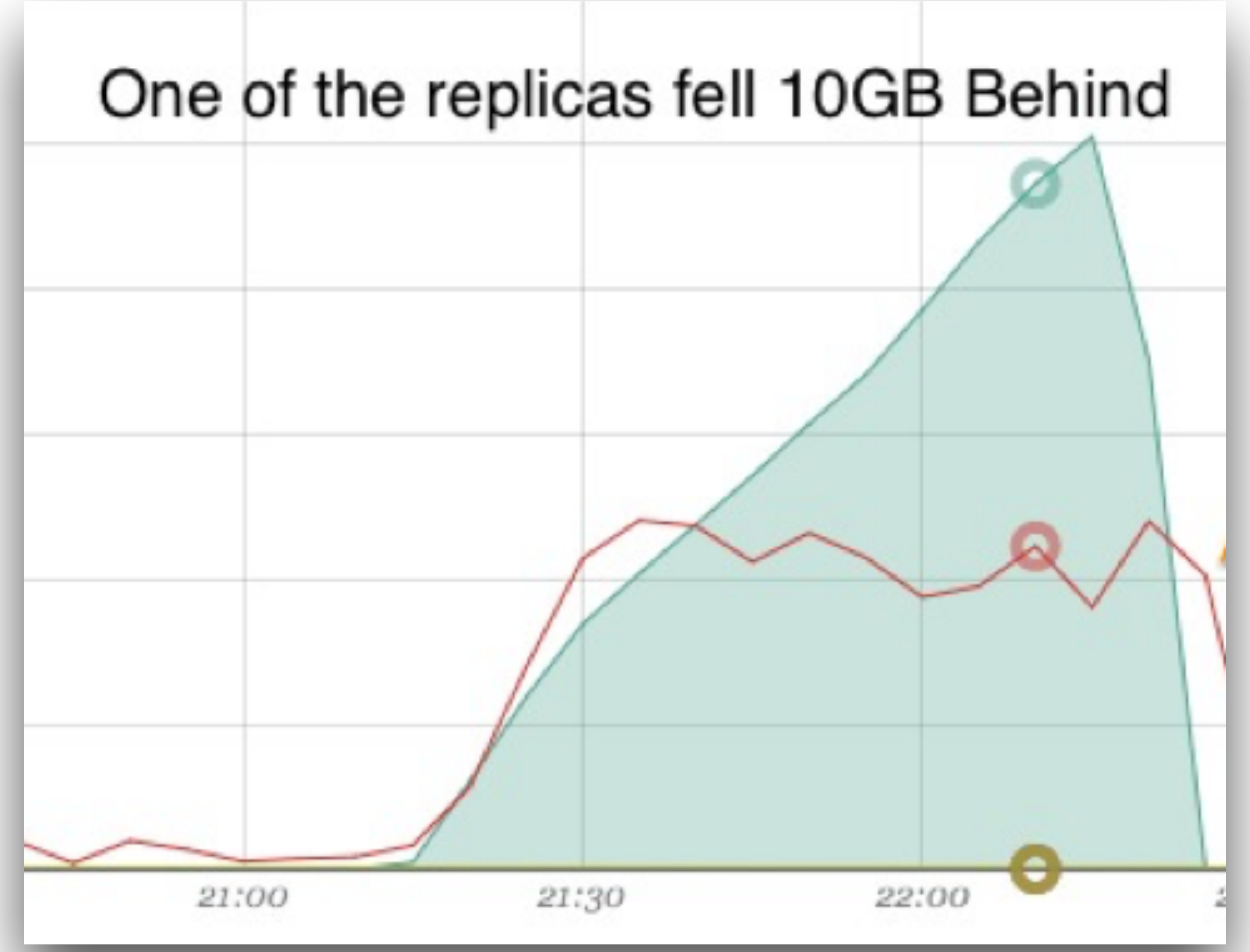
- Once you have at least one streaming replica live, you **must know at all times**, if the replica is falling behind the master.

```sql
-- on the master, location in Bytes
SELECT pg_current_xlog_location();

-- on the replica, location in Bytes
SELECT pg_last_xlog_replay_location();
-- on the replica, delta in terms of time
SELECT NOW() - pg_last_xact_replay_timestamp();
```

Our nagios checks automatically show the difference in MB as well as the time lag:

Service	Status	Last Check	Duration	Attempt	Status Information
PostgreSQL	OK	11-27-2015 04:42:48	1d 10h 1m 28s	1/3	OK - database postgres (0 sec.)
PostgreSQL long queries	OK	11-27-2015 04:42:53	289d 13h 57m 22s	1/3	LONG RUNNING QUERY : db010.prod is A OK
Ruby PGSQL Non-Critical Replication	OK	11-27-2015 04:29:00	14d 2h 8m 57s	1/3	REPLICATION OK : db010.prod replication lag is 0MB : time lag is 00:00:00.092649
Ruby PGSQL Replication	OK	11-27-2015 01:59:14	13d 16h 28m 42s	1/3	REPLICATION OK : db010.prod replication lag is 0MB : time lag is 00:00:00.013031

REPLICATION 102: **USE SSDS EVERYWHERE**

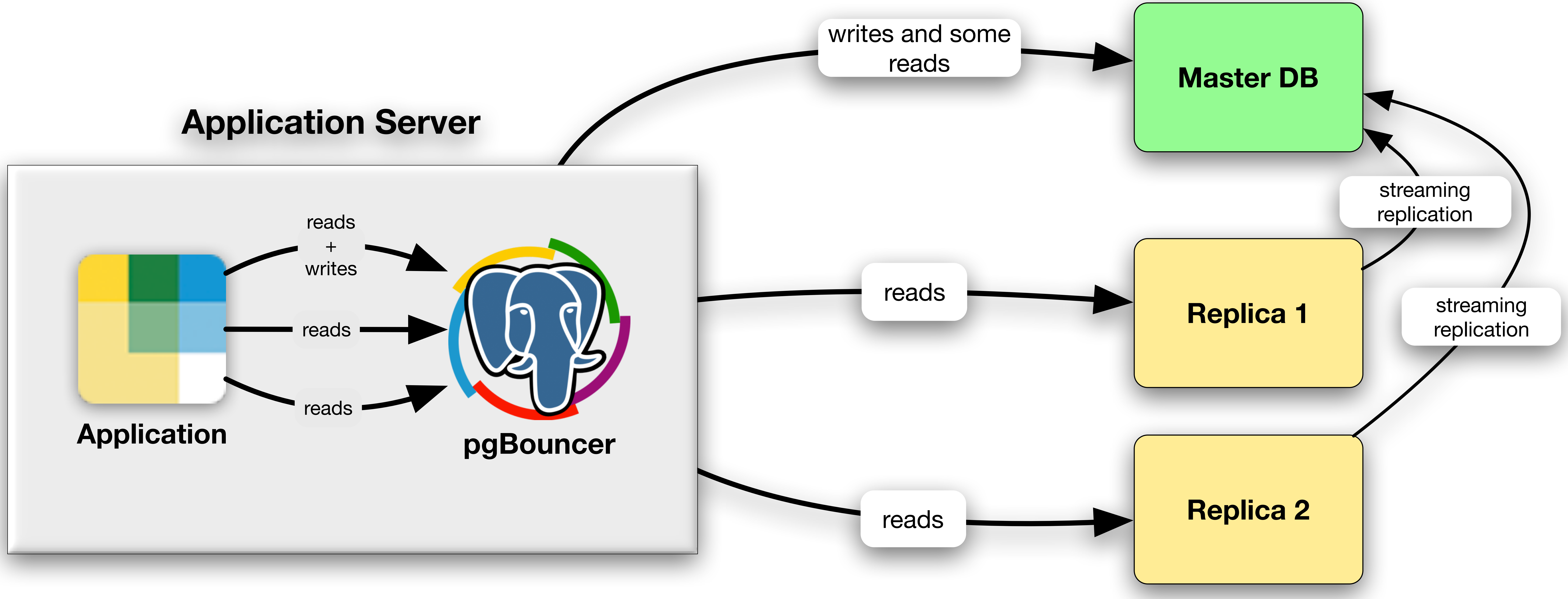


The red line is the site's error rate. Note the correlation.

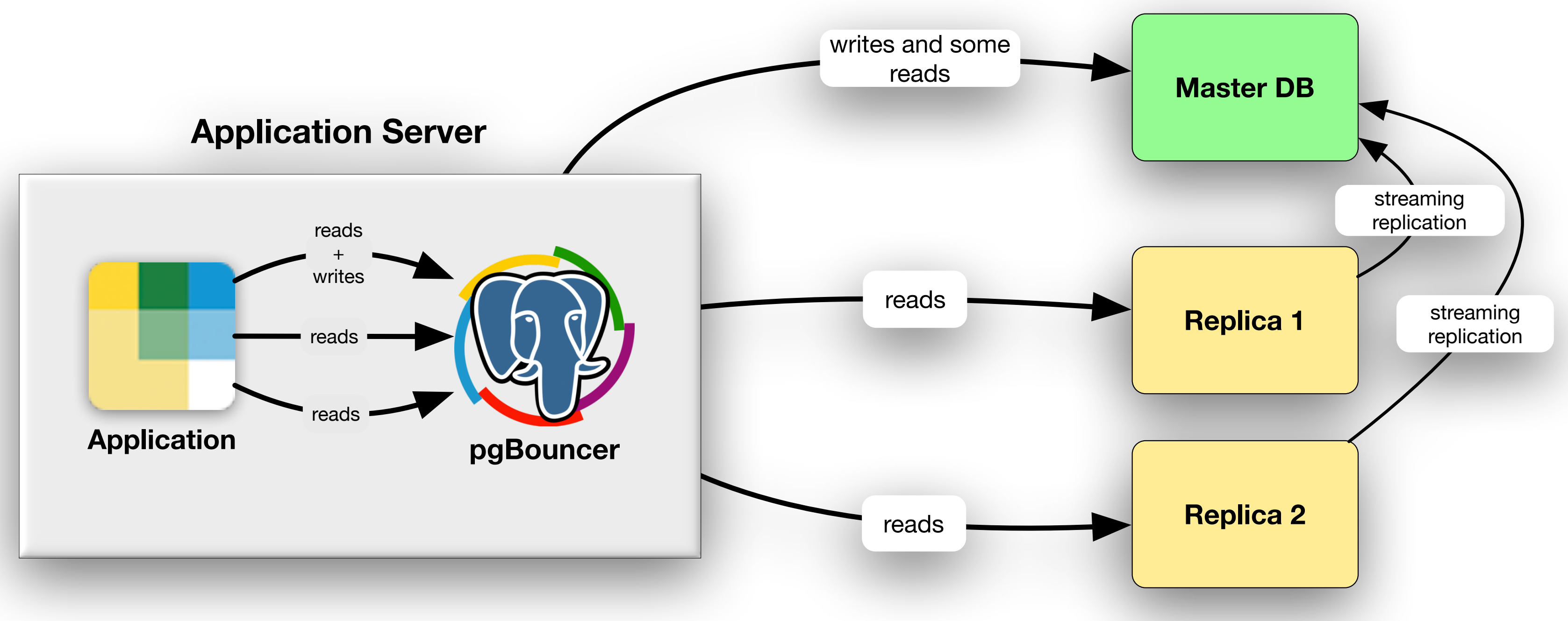
- One question with the replicas: **can they catch up with all the writes coming from the master?**
  - What if the master on SSDs, and replicas aren't? We tried this setup to save $$.
- And we instantly bumped into this problem: **applying WAL logs** to the replicas created very significant disk write load on non-SSD drives
- These replicas were barely able to apply writes from the master without live traffic.
- With traffic, **they would start falling behind, the delta ever increasing.**

HOW TO DISTRIBUTE **READS TO REPLICAS**?

This is a diagram of data flow between the application and the database with it's replicas, using pgBouncer to provide connection pooling from each app server.



BUT HOW DO WE **ROUTE READS TO REPLICAS?**



- We were **hoping** there was a generic solution, homelike like a pgBouncer, that would automatically route SELECT queries to the replicas, while all "write" requests to the master.
- Turns out that it is **nearly impossible to provide a generic tool that does this well.** For instance, how do you deal with a SELECT inside a transaction?
- As a result, most production-ready read/write splitting solutions **are built into the application itself.**
- We started looking for a Ruby solution, and were quickly unimpressed by everything we could find. One of the biggest issues was **thread-safety.** Only one of the libraries we found appeared to be thread safe.

https://github.com/taskrabbit/makara

READ SPLITTING WITH **MAKARA**

Makara is a ruby gem from TaskRabbit that was in production there, but only supported MySQL. We ported the database code from MySQL to PostgreSQL.

- Makara **automatically retries** if replica goes down
- **Load balances** with weights
- Was the **simplest** library to understand, and port to PG
- Was already running in **production**
- Worked in **multi-threaded** environment of Sidekiq Background Framework

```yaml
# database.yml
common: &common
  encoding: unicode
  username: application_user
  password: not-your-real-password
  prepared_statements: false
  host: 10.0.0.10
  pool: 1

  sticky_slave: true
  sticky_master: true

  adapter: makara

  db_adapter: postgresql
  user: application_user
  password: not-your-real-password
  blacklist_duration: 30

production:
  <<: *common

  databases:
    - name: master
      database: application_production_master
      role: master
      weight: 1
    - name: replica_1
      role: slave
      host: 10.0.0.12
      database: application_production_replica_1
      weight: 1
    - name: replica_2
      role: slave
      host: 10.0.0.13
      database: application_production_replica_3
      weight: 1
```

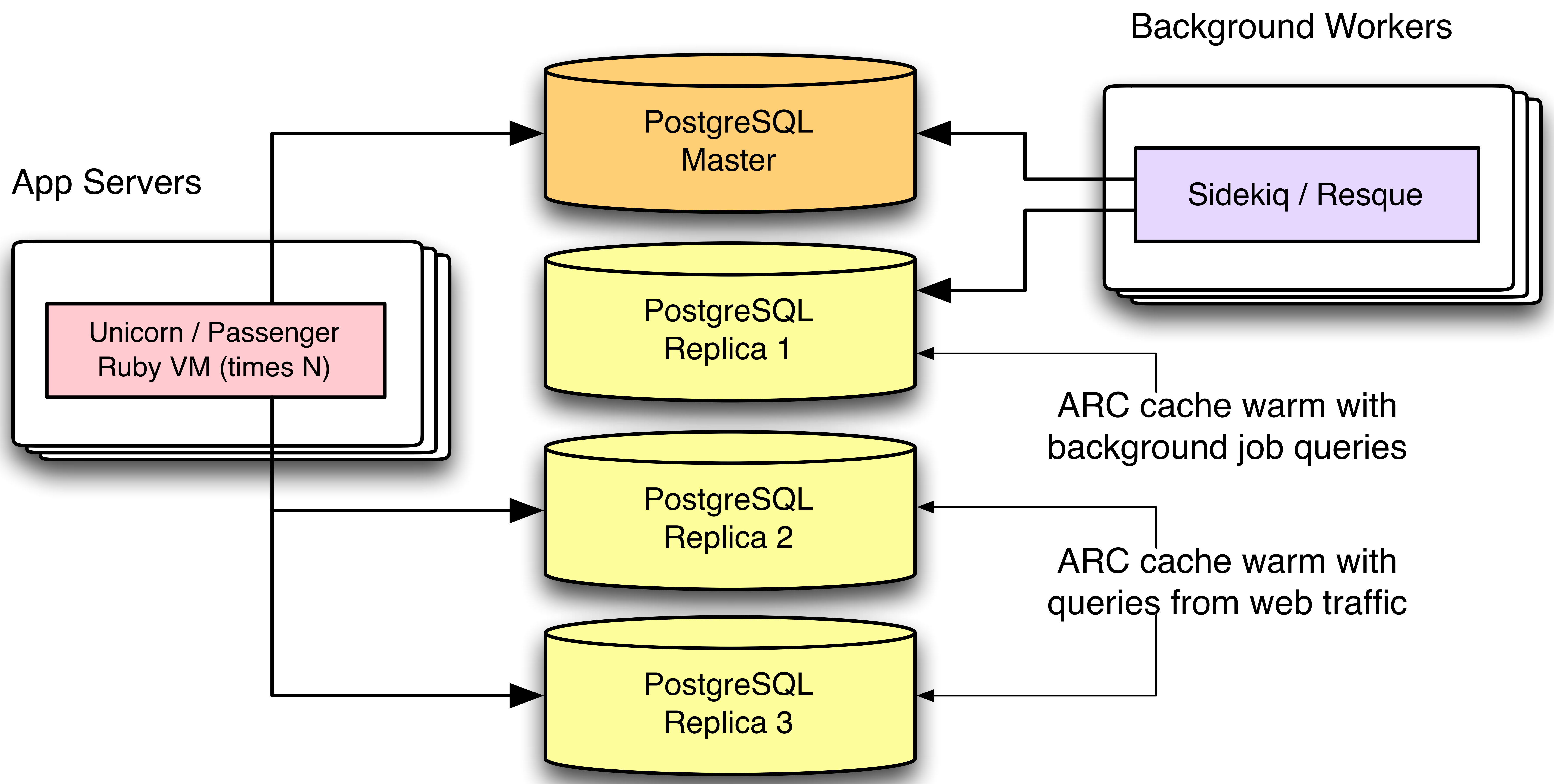
CONSIDERATIONS WHEN USING REPLICATION

- Application must be tuned to support **eventual consistency**. Data may not yet be on replica!
- Must explicitly **force fetch** from the master DB when it's critical (i.e. after a user account's creation)
- We often use below pattern of first trying the fetch, if nothing found retry on master db

```
def require_attribute!(attr, user)
  if user.send(attr).blank?
    Wanelo::DatabaseHelper.force_master!
    user.reload
  end
end
```

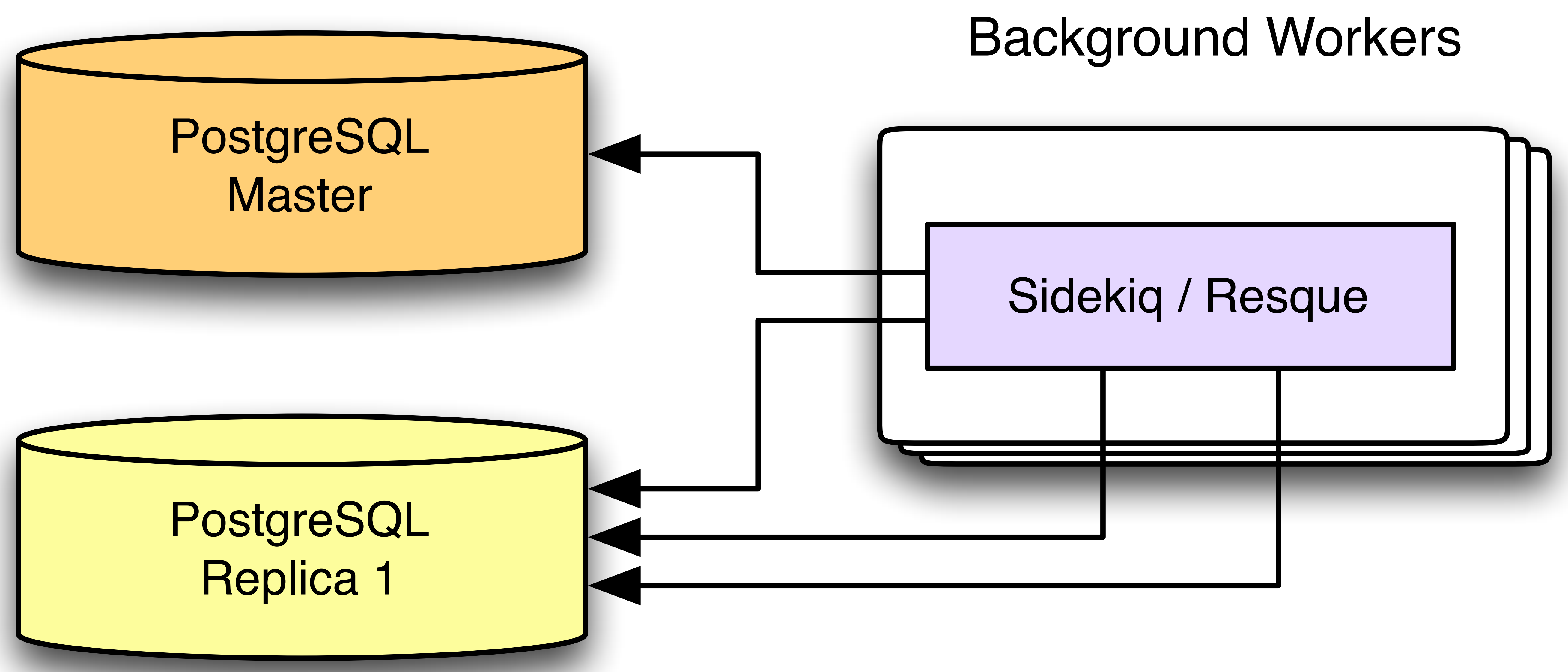
USEFUL TIP: REPLICAS CAN SPECIALIZE

Background Workers can use dedicated replica not shared with the app servers, to optimize hit rate for file system cache (ARC) on both replicas



BIG **HEAVY READS** GO THERE

- Long heavy queries should run by the background jobs against a dedicated replica, to isolate their effect on web traffic
- Each type of load will produce a unique set of data cached by the file system



SCALING UP

4. UPGRADING (VIRTUAL) **HARDWARE**

HARDWARE: **IO** & **RAM**

- Sounds obvious, but better or faster hardware is an obvious choice when scaling out
- Large RAM will be used as file system cache
- On Joyent's SmartOS ARC FS cache is very effective
- **shared_buffers** should be set to 25% of RAM or 12GB, whichever is smaller.
- Using **fast SSD** disk array made an enormous difference
- Joyent's native 16-disk RAID managed by ZFS instead of controller provides excellent performance

SCALING UP

5. NO TOOL EXCELS AT **EVERYTHING**

AND POSTGRESQL IS NO EXCEPTION.

WHEN POSTGRESQL IS **NOT ENOUGH**

Not every type of data is well suited for storing and quickly fetching from a relational DB, even though initially it may be convenient. For example, our initial implementation of the "text search" in PG became too slow when the # of documents reached 1M.

- **Solr** is great for full text search, and deep paginated sorted lists, such as popular, or related products
- **ElasticSearch** is a superset of Solr, but scales wide near infinitum. We ran 0.5Tb ElasticSearch cluster.
- **Redis** is a great data store for transient or semi-persistent data with list, hash or set semantics
- **RabbitMQ** is a fantastic high performance queue, with both point-to-point and pub-sub communications.

SOME **CAVEATS** OF EACH STORE WE TRIED

- **Solr** is easy to replicate to 10-20 replicas, but they take **toll on the master**.
  - Do not serve reads from the master.
  - For high document update rate, set # of documents to commit to a stratospheric value.
- **ElasticSearch** is difficult to manage and configure for high availability. Professional services cost a lot, pricing not startup friendly.
- **Redis** is **not a transactional**, or a txn-**reliable** data store despite what anyone says. Expect all data to go away at some point, and always have a way to rebuild it from the DB if it's critical.
- **RabbitMQ** is great, but remember that queues and messages are not "durable", ie. on disk by default.

REDIS SIDETRACK: LESSONS LEARNED

This in-memory store is very good for certain applications where PostgreSQL isn't.
I like to think of Redis as a in-memory cache with additional hash, set and list semantics. And they totally rock!
When Redis backs up data, or tries to replicate (ugh, that was rough), it forks. Memory reqs double!

- We use Redis for ActivityFeed by **precomputing each user's feed at write time**. But we can regenerate it if the data is lost from Redis
- We use **twemproxy** in front of Redis which provides automatic **horizontal sharding** and connection pooling.
- We run **clusters of 256 redis shards** across many virtual zones; sharded redis instances use many cores, instead of just one (as a single instance would)
- Small redis shards can easily fork to backup the data, as the data is small.
- We squeezed more performance of Redis by packaging **multiple**

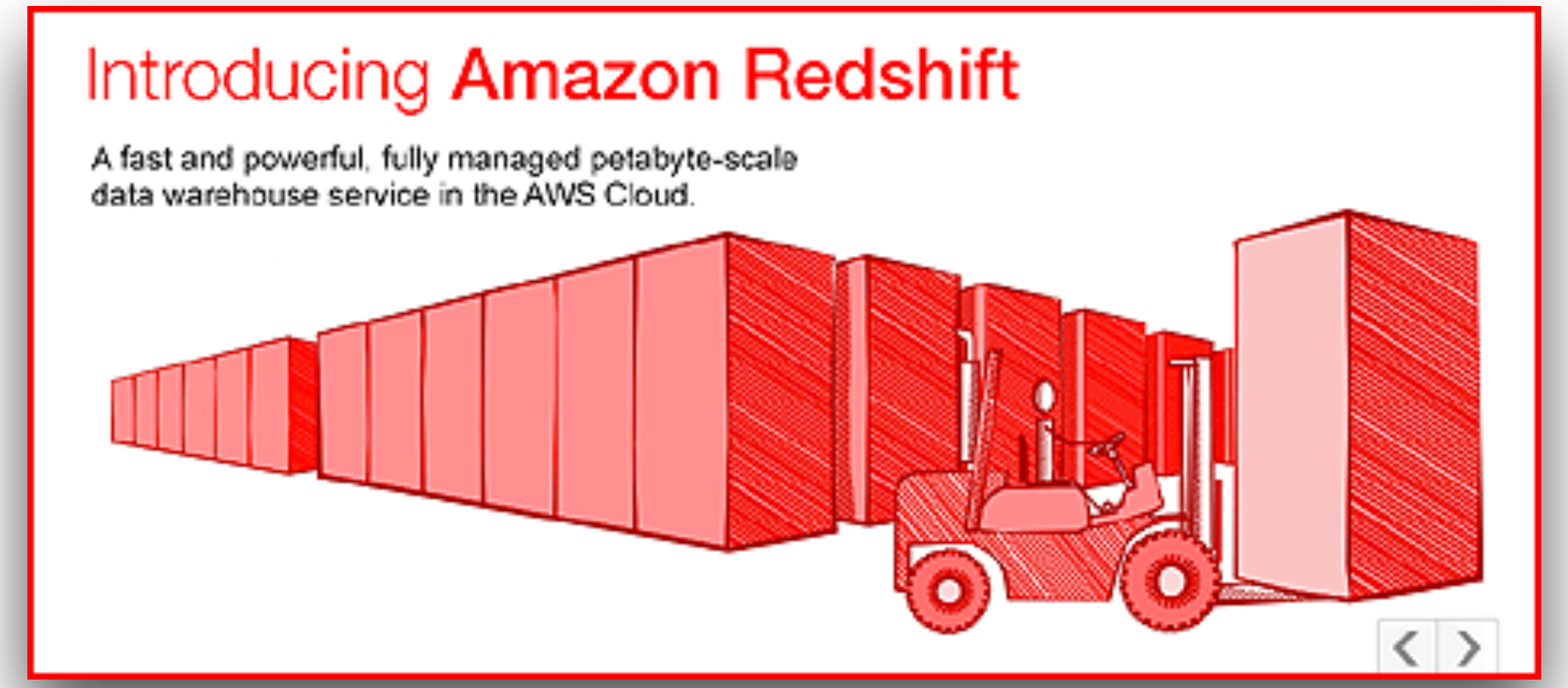
SCALING UP

6. MOVE **EVENT**-LIKE TABLES OUT OF POSTGRESQL

EVENT LOGS, & **APPEND-ONLY** TABLES

- Many analysts and business stakeholders like to **collect an ever-growing list of metrics**, i.e.
  - User/business events such as “registered”, “ordered”
  - System events, such as “database went offline”
  - Click-stream events, that follow nginx access_log file
  - State changes history on key models, like an Order
- These append-only tables often **start in the database**, but quickly prove to be a nuisance
- They generate **very high write IO**, often overwhelming the underlying hardware

MOVE **EVENT TABLES** OUT OF DATABASE



- We were appending all user events into a single table **user_events**
- The application was generating **millions of rows per day**!
- After realizing that there was no reason this data needed to stay in PostgreSQL we moved it out using a clever solution, that combined:
  - Event dispatch system using ruby gem **Ventable**
  - Event recording using **rsyslog**
  - Data analysis using a combination of AWS **Redshift**, and Joyent's **Manta.**
  - **Manta is an object store with native compute facility, that supports concurrent** analysis of thousands of objects in parallel. It provides map/reduce facility, and bash tools like awk and grep for filtering and mapping.
  - We eventually migrated most of the analytics to **RedShift**, in order to return to regular SQL for analytics.

DETAILED BLOG POST ABOUT THIS MIGRATION

http://wanelo.ly/event-collection

DOING THE HARD STUFF, BUT INCREMENTALLY AND **METHODICALLY**

SCALING OUT: TUNING

7. TUNE POSTGRESQL & FILESYSTEM

THIS HAPPENED TO US

- Problem: zones (virtual hosts) with "write problems" appeared to be **writing 16 times more data to disk**, compared to what virtual file system reports
  - **vfsstat** says 8Mb/sec write volume
  - **iostat** says 128Mb/sec is actually written to disk

So what's going on?

TUNING FILESYSTEM

- Turns out default ZFS **block size** is **128Kb**, and PostgreSQL **page size** is **8Kb**.
- Every small write that touched a page, had to **write 128Kb** of a ZFS block to the disk
- This may be good for huge sequential writes, but not for random access, lots of tiny writes

TUNING **ZFS** & PGSQL

- Solution: Joyent changed ZFS block size for our zone, **iostat** write volume dropped to 8Mb/sec
- We also added **commit_delay**

```
fsync = on
synchronous_commit = off
wal_sync_method = fsync
full_page_writes = on
wal_buffers = 32MB
wal_writer_delay = 200ms
commit_delay = 100
commit_siblings = 5
```

THIS KNOWLEDGE SHOULD BE **PART OF**

- Many of these settings are the default in our open-source **Chef cookbook** for installing PostgreSQL from sources

https://github.com/wanelo-chef/postgres

- It installs PG in eg **/opt/local/postgresql-9.5.0**
- It configures it's data in **/var/pgsql/data95**
- It allows *seamless* and *safe* upgrades of minor or major versions of PostgreSQL, never overwriting binaries

ONLINE RESOURCES ON **PG TUNING**

- Josh Berkus's "5 steps to PostgreSQL Performance" on SlideShare is fantastic

http://www.slideshare.net/PGExperts/five-steps-perform2013

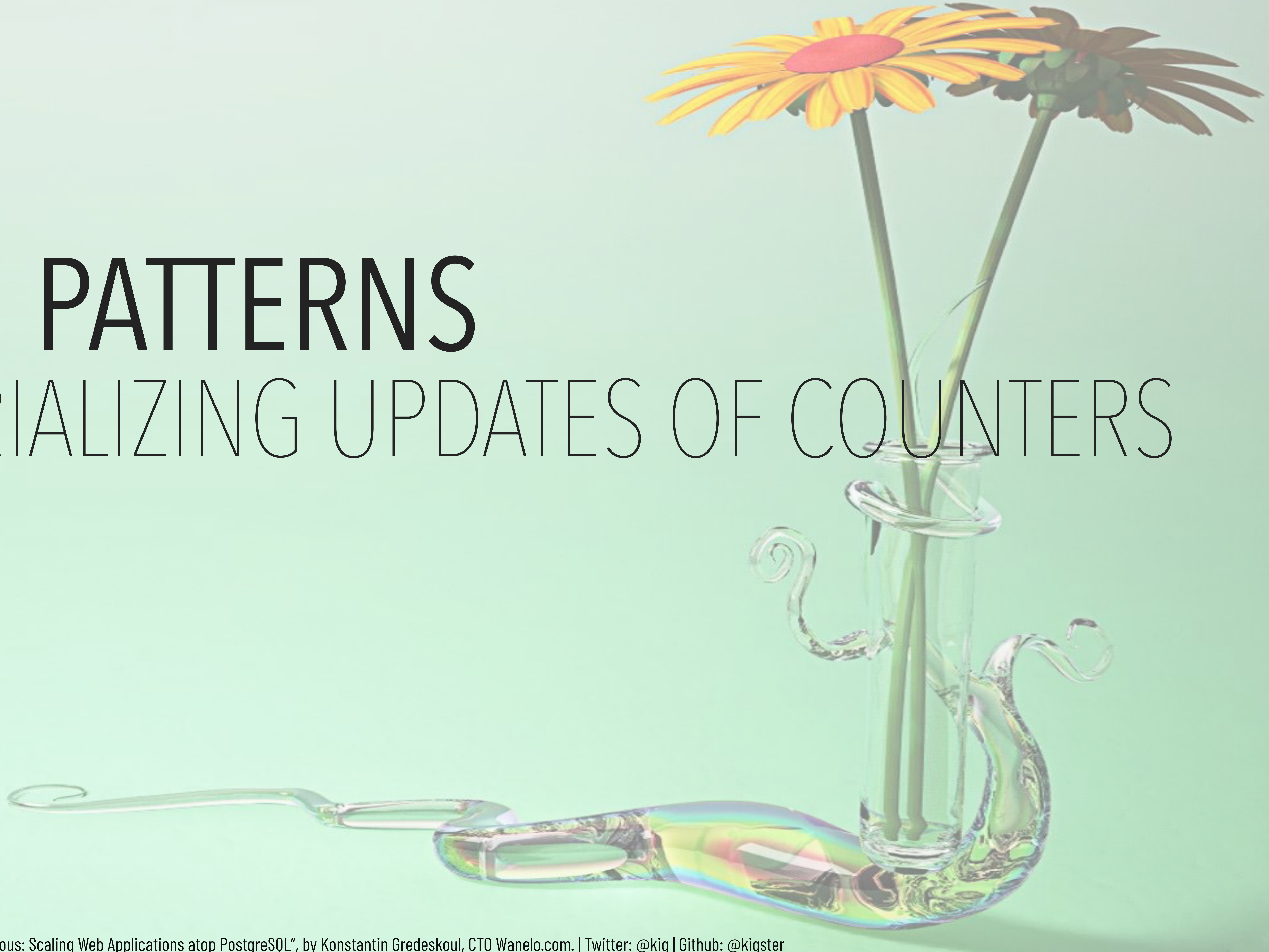
- PostgreSQL wiki pages on performance tuning are excellent

http://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server

http://wiki.postgresql.org/wiki/Performance_Optimization

SCALING OUT: PATTERNS

8. BUFFERING, SERIALIZING UPDATES OF COUNTERS



REDUCE **WRITE IO** AND **LOCK CONTENTION**

- Problem: **products.saves_count** is incremented every time someone saves a product (by 1)
- At 100s of inserts/sec, that's a lot of updates
- Worse: 100s of concurrent requests trying to obtain a **row level lock** on the same popular product

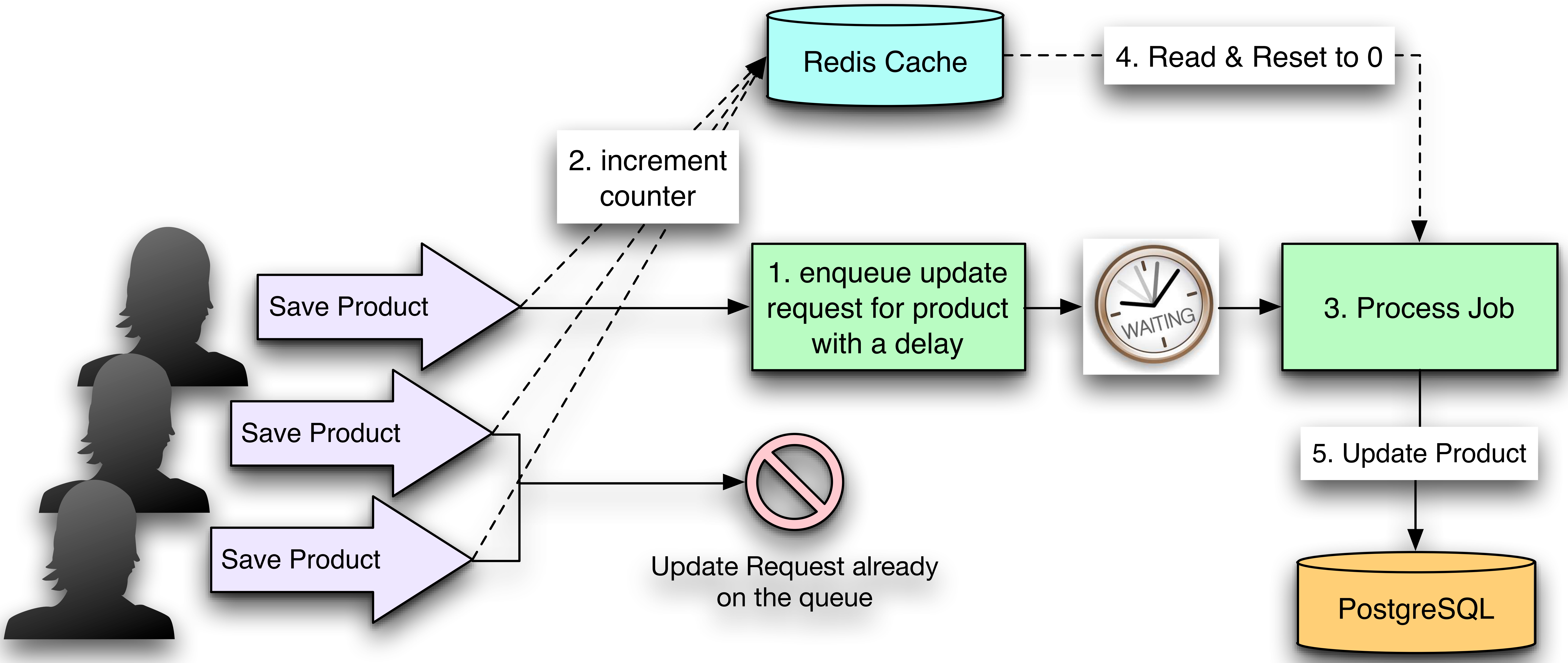
BUFFERING AND **SERIALIZING**

- Sidekiq background job framework has two inter-related features:
  - **scheduling in the future** (say 10 minutes ahead)
  - **UniqueJob** extension

- We increment a counter in redis, and enqueue a job that says "update product in 10 minutes"

- Once every 10 minutes popular products are updated by adding a value stored in Redis to the database value, and resetting Redis value to 0

BUFFERING **IN PICTURES**

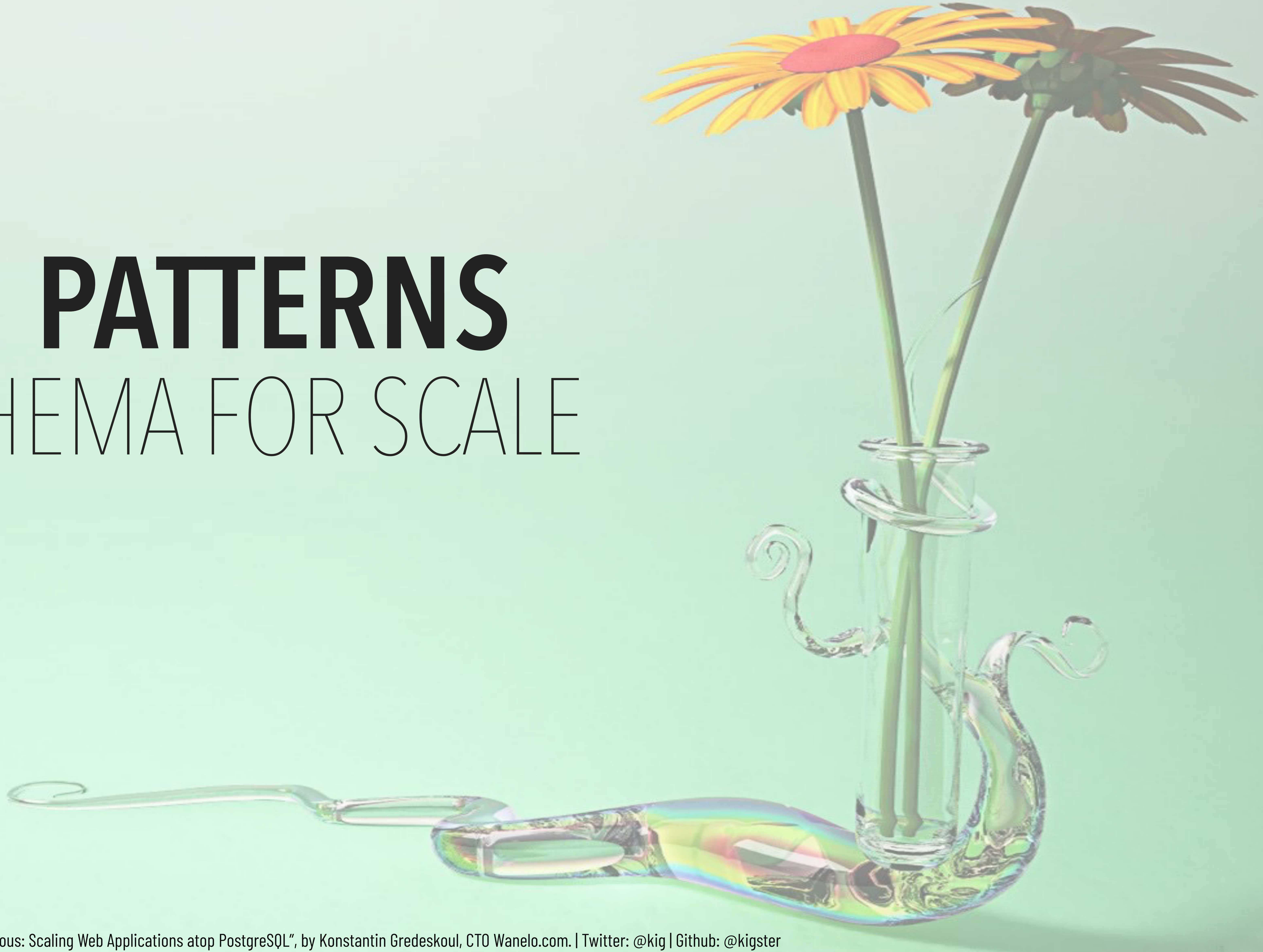


BUFFERING **CONCLUSIONS**

- If we show objects from the database, they might be sometimes behind on the counter. **It might be okay if the alternative is to be down.**
- If not, to achieve read consistency, we can display the count as database value plus the redis-cached value at **read time**

SCALING OUT: PATTERNS

9. OPTIMIZING SCHEMA FOR SCALE



MVCC DOES **COPY ON WRITE**

Problem: heavy writes on the master db, due to the fact that PostgreSQL rewrites each row for most updates.

Some exceptions exist: i.e. non-indexed integer column, a counter, timestamp or other simple non-indexed type

- But we often index these so we can sort by them
- So rewriting **user** means rewriting the entire row
- Solution: move frequently updated columns away

TOO MANY WRITES: THIS IS HOW IT BEGINS

```
dbname> select * from pg_stat_user_tables where relname = 'users';
-[ RECORD 1 ]-------+------------------------------
relid               | 192077
schemaname          | public
relname             | users
seq_scan            | 941091708
seq_tup_read        | 751033336502
idx_scan            | 701052608541
idx_tup_fetch       | 65975559248
n_tup_ins           | 65127168
n_tup_upd           | 1133654522
n_tup_del           | 0
n_tup_hot_upd       | 6523872047
n_live_tup          | 7416758956
n_dead_tup          | 3573
```

```ruby

after_filter do |controller|
  controller.current_user.update_attribute(:last_logged_in_at, Time.now)
end
```

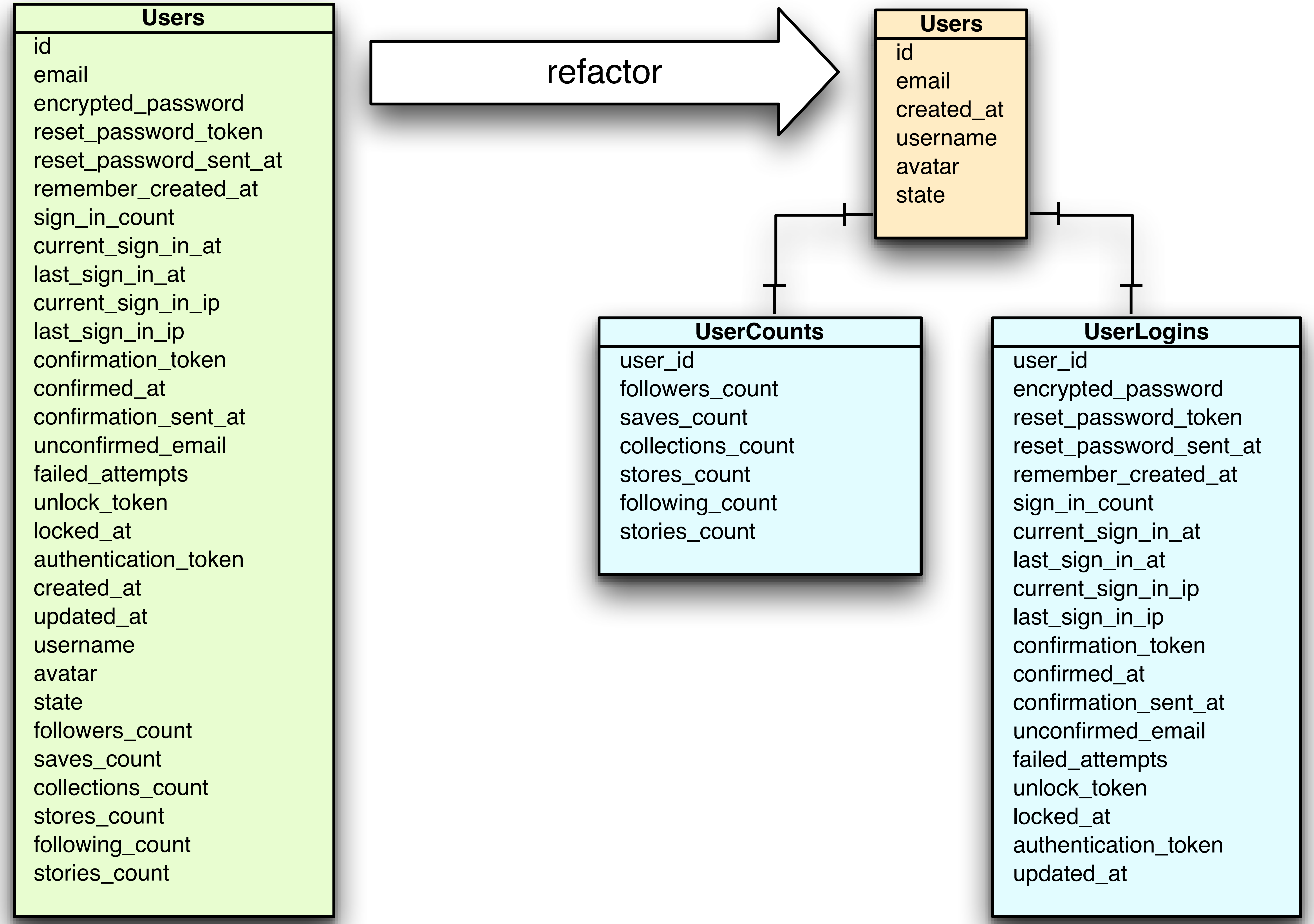
- We notice how much writes we are doing on the database machine, and become curious.
- Something must not be right. What is it?
- Quick check with **pg_stat_user_tables** reveals that our **users** table is doing a huge number of updates, many of them are not "hot" updates
- Subsequent research reveals the **following line is at fault:** we update the entire user row for each

SCHEMA **TRICKS**

Users
id
email
encrypted_password
reset_password_token
reset_password_sent_at
remember_created_at
sign_in_count
current_sign_in_at
last_sign_in_at
current_sign_in_ip
last_sign_in_ip
confirmation_token
confirmed_at
confirmation_sent_at
unconfirmed_email
failed_attempts
unlock_token
locked_at
authentication_token
created_at
updated_at
username
avatar
state
followers_count
saves_count
collections_count
stores_count
following_count
stories_count

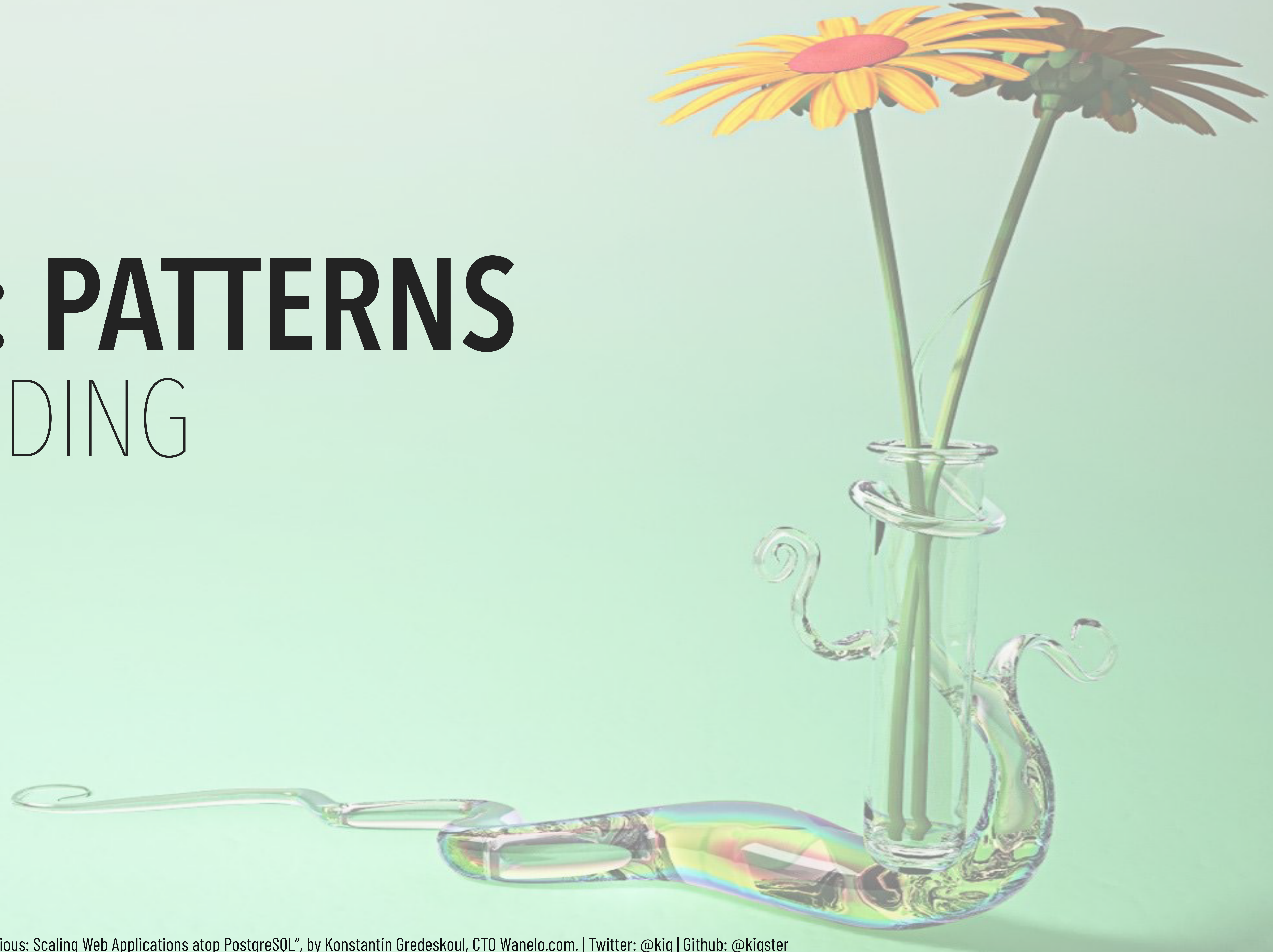
- **Split wide tables** that get a lot of updates into two more more 1-1 tables, to reduce the impact of an update
- Much less vacuuming required when smaller tables are frequently updated, **especially if this allows the updates to remain "hot".**

VERTICAL **TABLE SPLIT**



SCALING OUT: PATTERNS

10. VERTICAL SHARDING



VERTICAL SHARDING – WHAT IS IT?

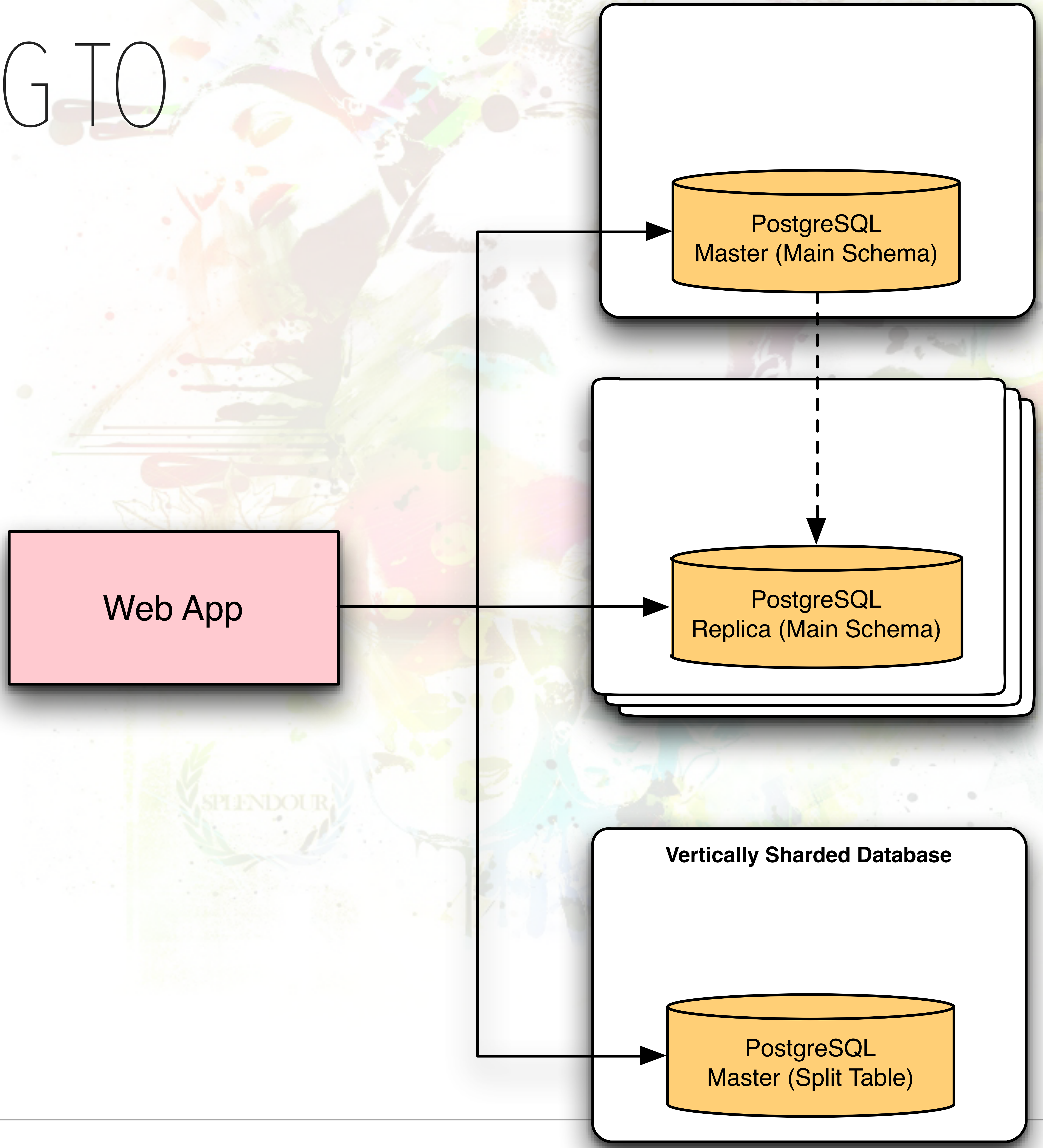
- Heavy tables with too many writes, can be moved into their own separate database
- For us it was **saves**: now @ **3B+ rows**
  - At hundreds of inserts per second, and 4 indexes, we were feeling the pain.
  - “Save” is like a “Like” on Instagram, or “Pin” on Pinterest.
- It turns out moving a single table (in Rails) out is a not a huge effort: **it took our team 3 days**

VERTICAL SHARDING – **HOW**?

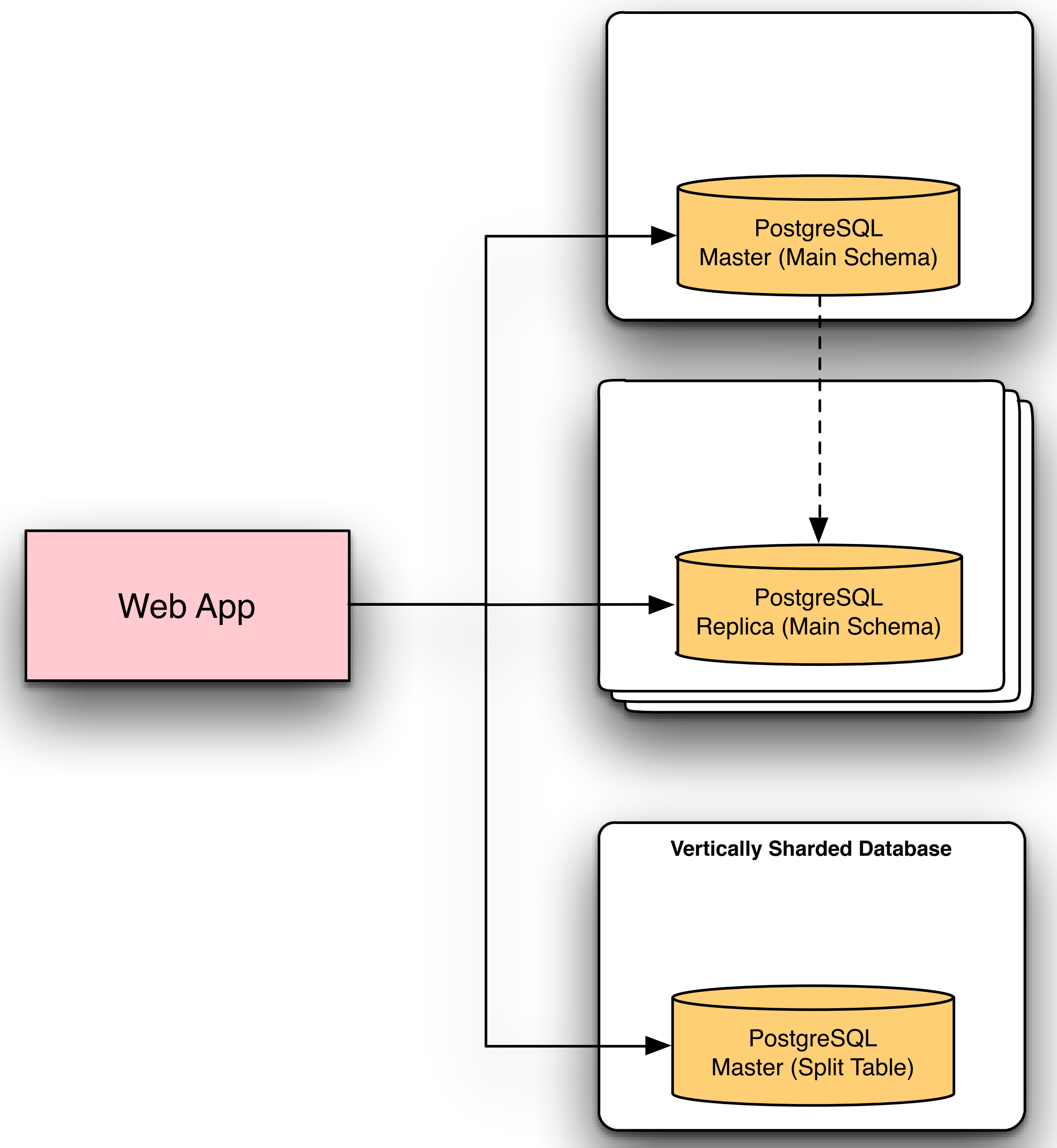
http://wanelo.ly/vertical-sharding

- Update code to point to both old and new databases (new – only for the shared model)
- Implement any dynamic Rails association methods as real methods
  - ie. **save.products** becomes a method on Save model, lookup up Products by IDs
- Update development and test setup with two primary databases and fix all the tests

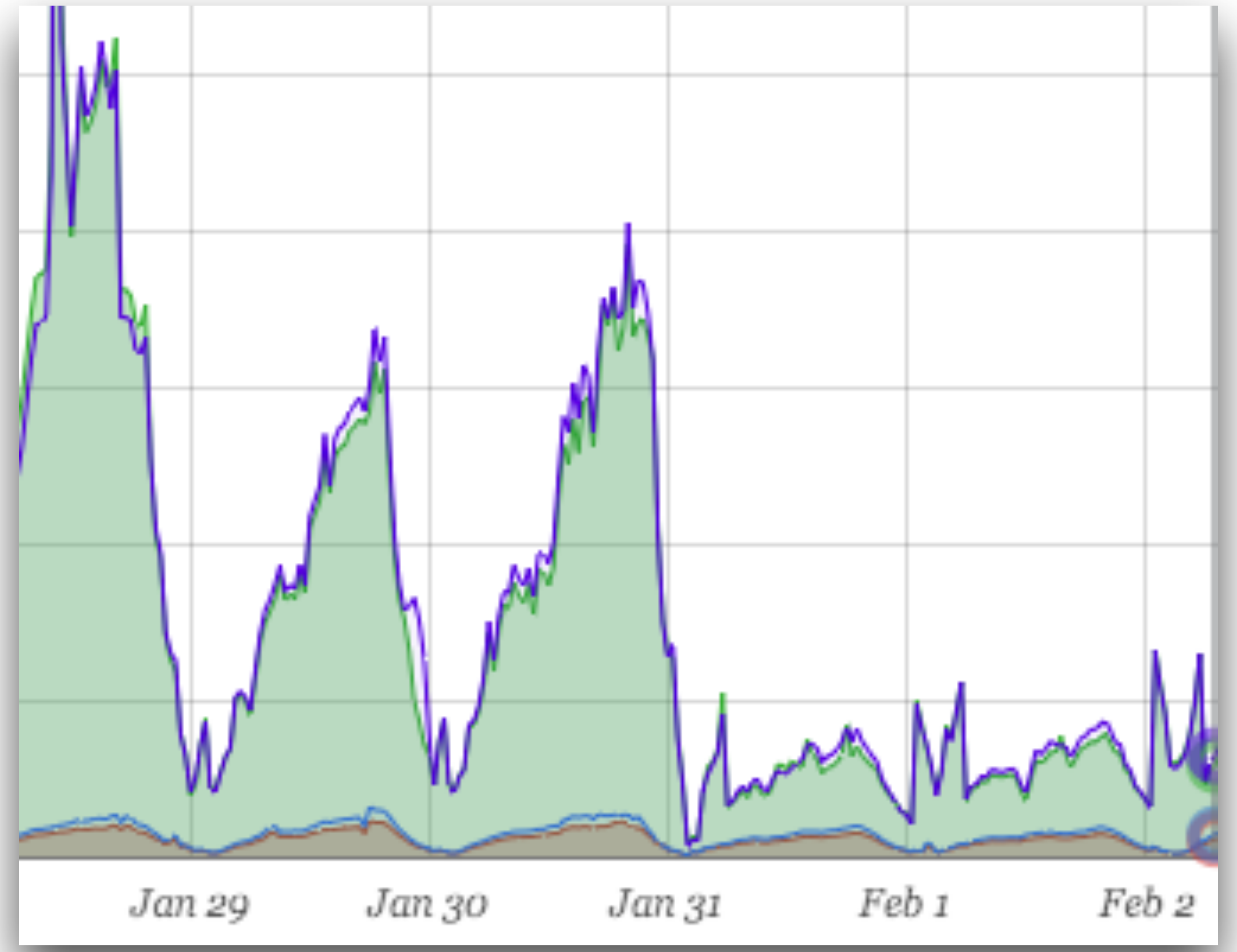
APPLICATION TALKING TO
TWO DATABASES



VERTICAL SHARDING – **DEPLOYING**



- Drop in write IO on the main DB after splitting off the **high IO** table into a dedicated compute node



SCALING OUT: PATTERNS

11. WRAPPING VERTICALLY SHARDED DATA WITH **MICRO SERVICES**

SPLITTING OFF MICRO-**SERVICES**

- Vertical Sharding is a great **precursor** to a micro-services architecture

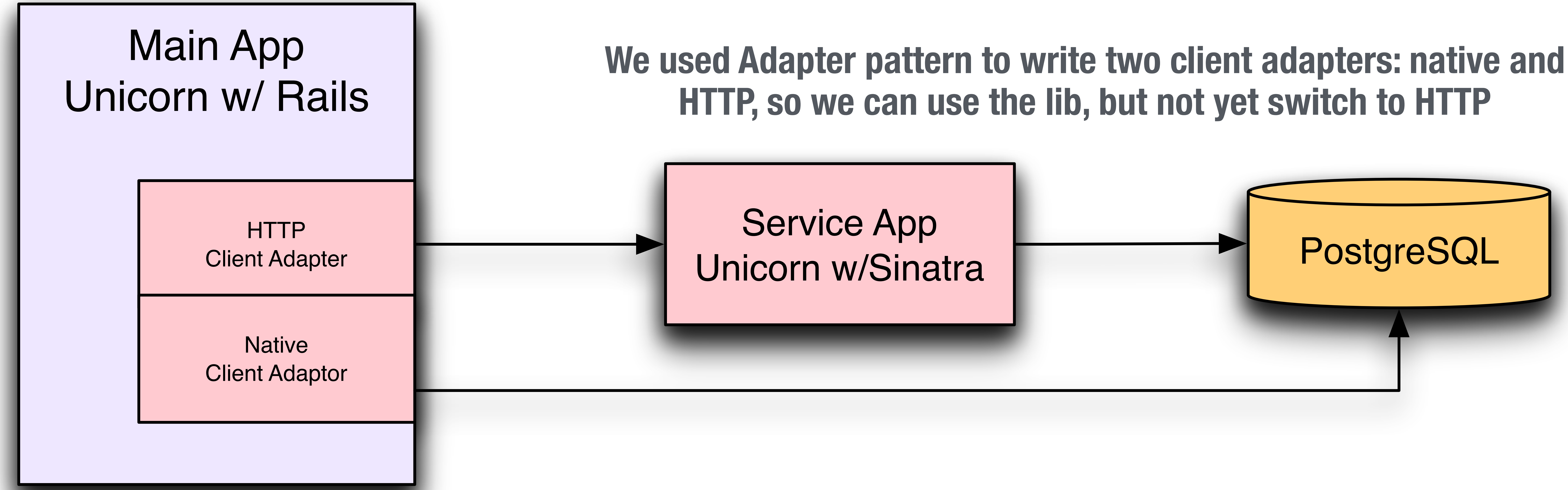
- We already have **Saves** in another database, let's migrate it to a light-weight HTTP service

  - **New service**: Sinatra, client and server libs, updated tests & development, CI, deployment without changing db schema

  - **2-3 weeks a pair of engineers** level of effort

ADAPTER DESIGN PATTERN TO THE RESCUE

We used Adapter pattern to write two client adapters: native and HTTP, so we can use the lib, but not yet switch to HTTP

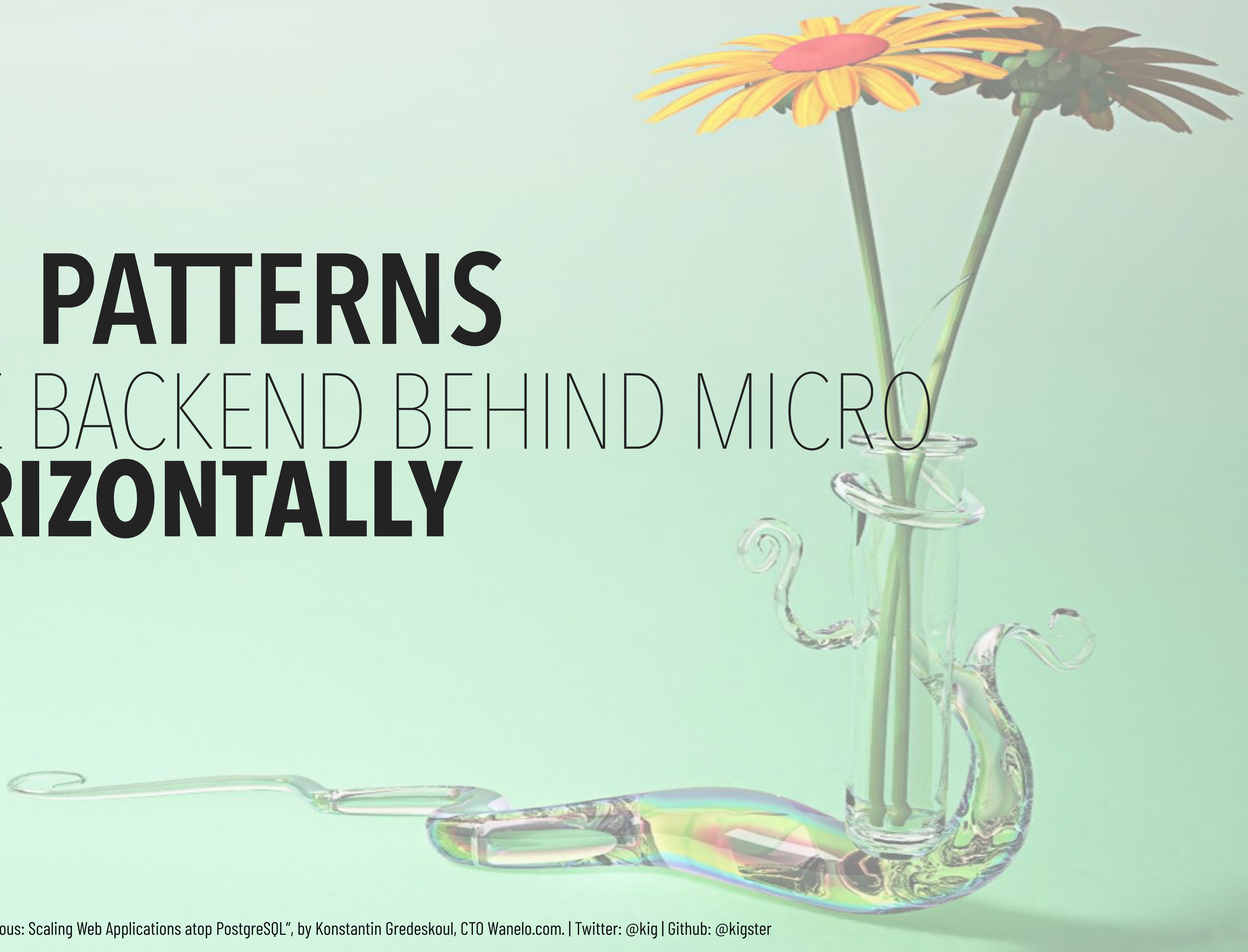


SERVICES **CONCLUSIONS**

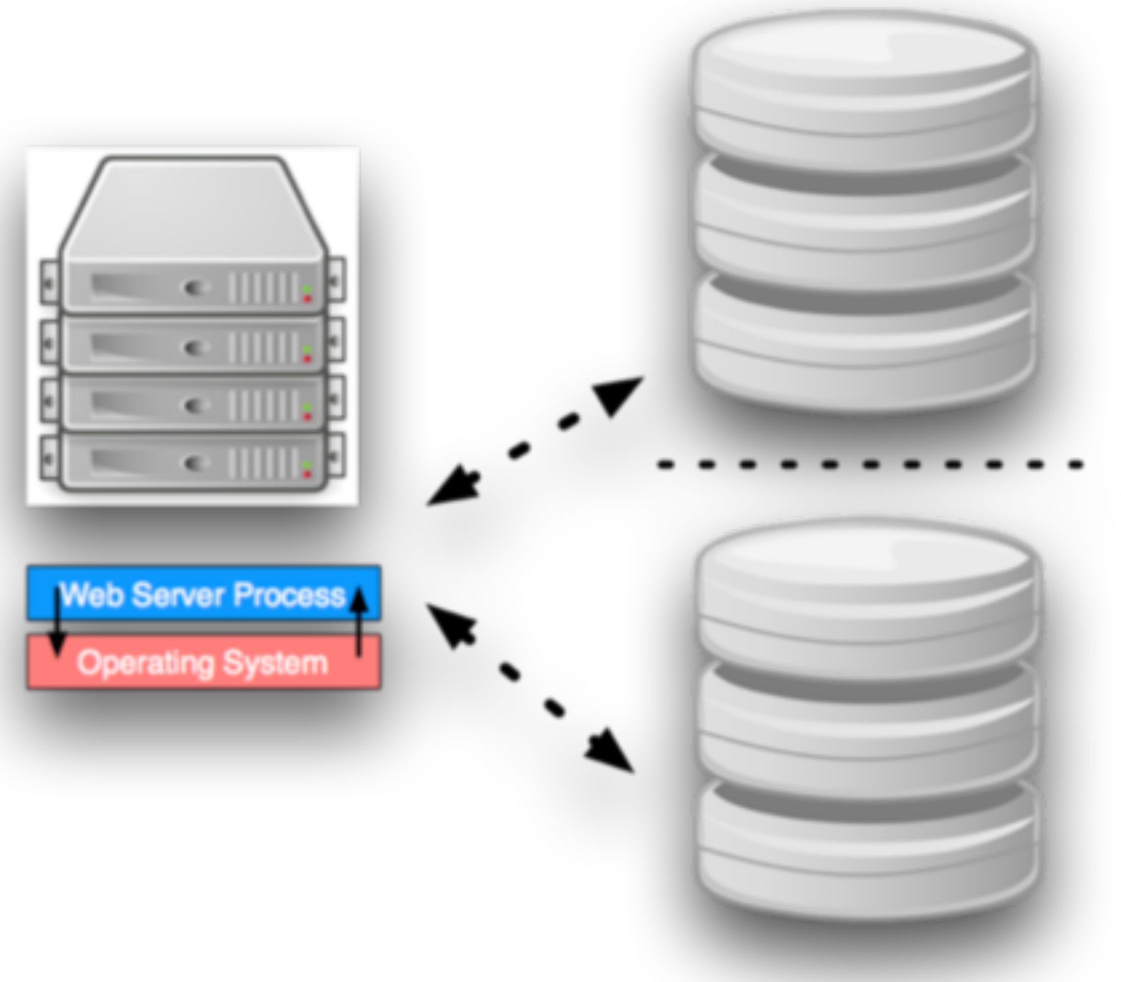
- Now we can independently scale service backend, in particular reads by **using replicas**
- This prepares us for the next inevitable step: **horizontal sharding**
- At a cost of added request latency, lots of extra code, extra runtime infrastructure, and **2 weeks of work**
- Do this only if you absolutely **have to:** it adds complexity, more moving parts, etc. This is not to be taken lightly!

SCALING OUT: PATTERNS

12. SHARDING THE BACKEND BEHIND MICRO SERVICES **HORIZONTALLY**



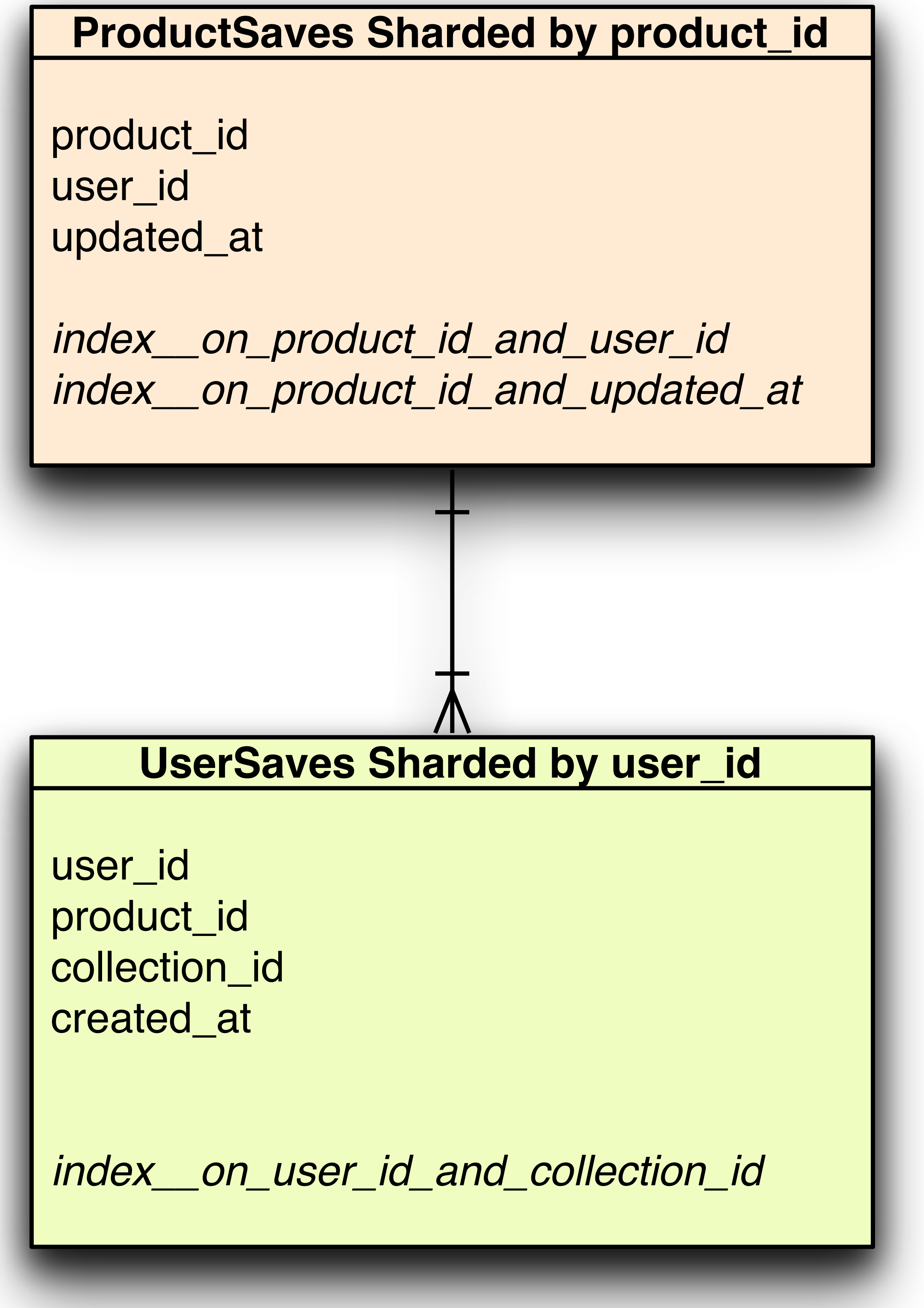
HORIZONTAL SHARDING CONCEPTS



- We wanted to stick with PostgreSQL for critical data such as saves, and avoid learning a new tool.
- Really liked Instagram's approach with **schemas**
- Built our own **schema-based sharding** in ruby, on top of Sequel gem, and open sourced it
- It supports mapping of physical to logical shards, and connection pooling

SCHEMA DESIGN FOR HORIZONTAL SHARDING

- We needed two lookups, by **user_id** and by **product_id** hence we needed two tables, independently sharded
- Since saves is a join table between user, product, collection, we did not need unique ID generated
- Composite base62 encoded ID: **fpua-1BrV-1kKEt**

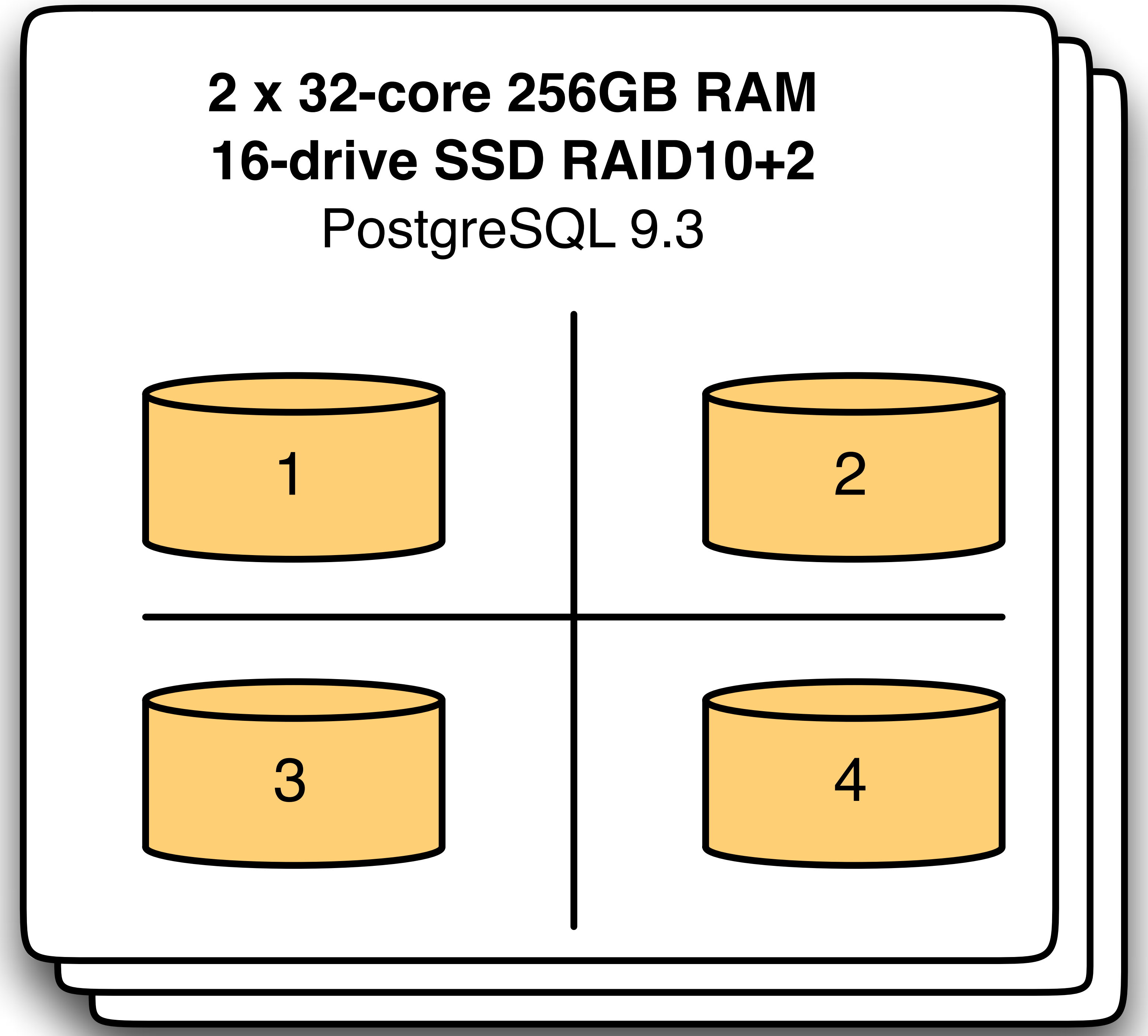


https://github.com/wanelo/sequel-schema-sharding

SPREADING YOUR SHARDS :)

Use our ruby library to do the this:
https://github.com/wanelo/sequel-schema-sharding

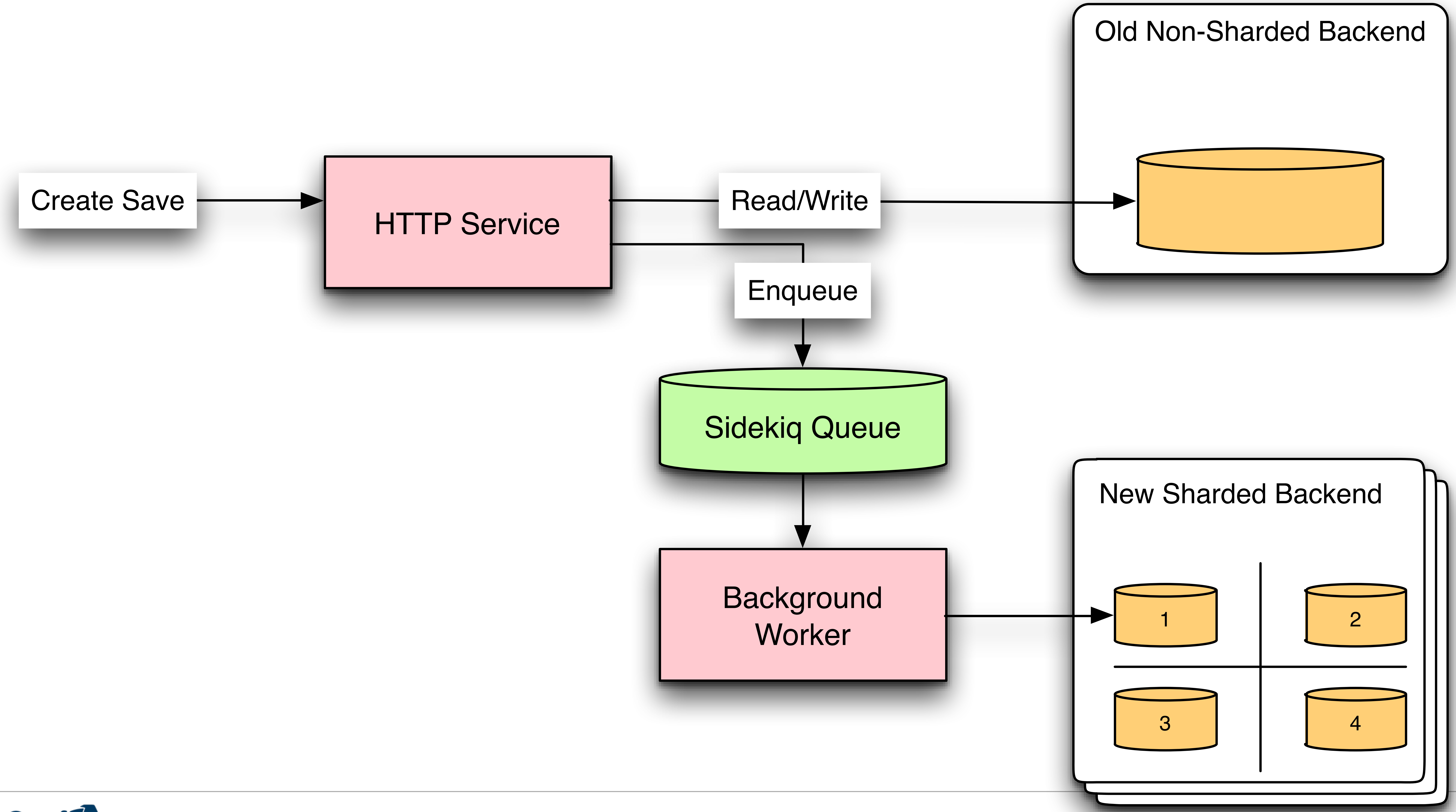
- We split saves into **8192** logical shards, distributed across 8 PostgreSQL databases
- Running on **8** virtual zones spanning **2** physical SSD servers, **4** per compute node
- Each database has **1024** schemas (twice, because we sharded saves into two tables)



QUESTION:

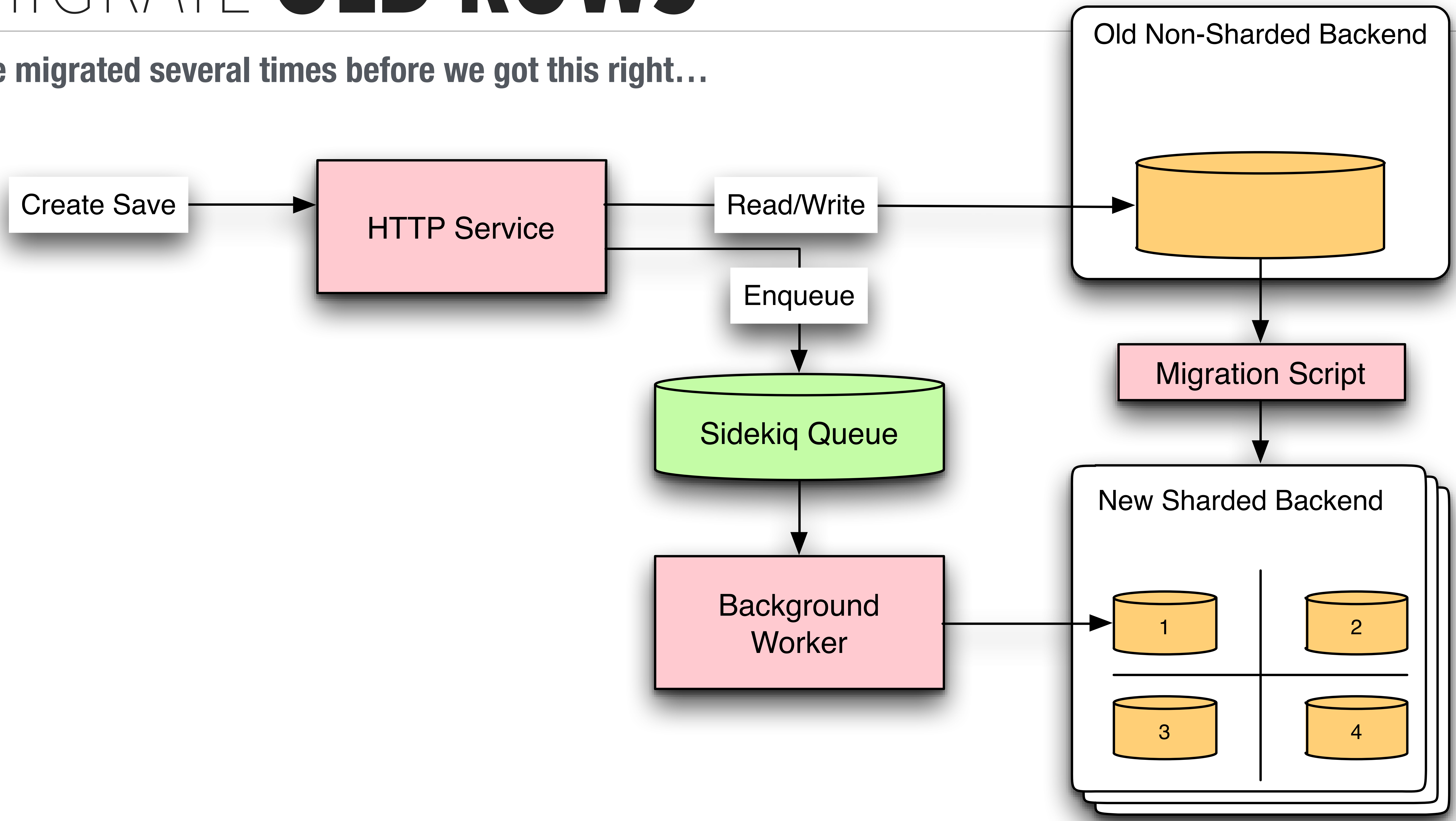
HOW CAN WE MIGRATE THE DATA FROM THE OLD BACKEND TO THE NEW HORIZONTALLY SHARDED ONE, BUT **WITHOUT ANY DOWNTIME?**

YES! **NEW RECORDS** GO TO BOTH

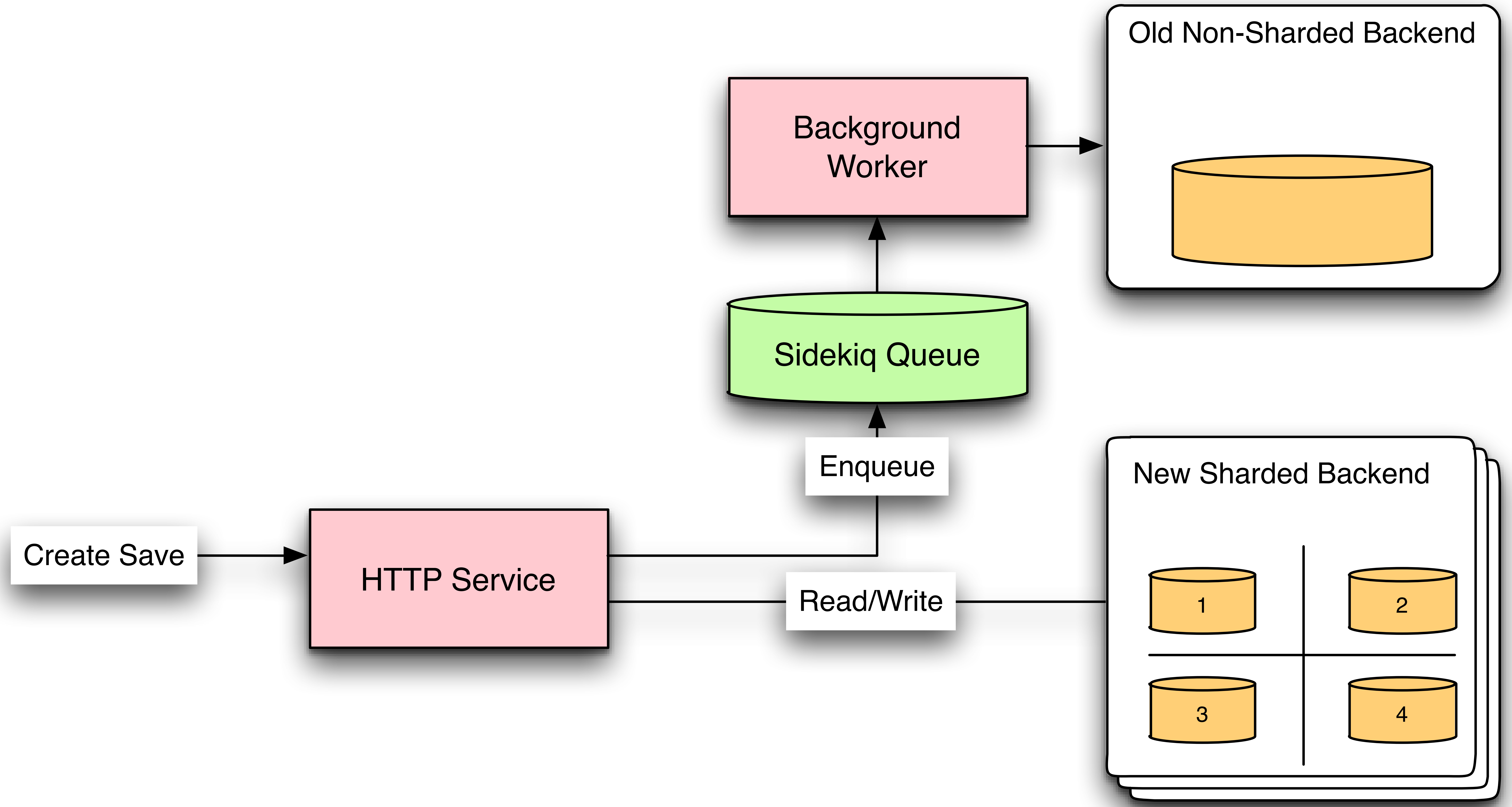


MIGRATE **OLD ROWS**

We migrated several times before we got this right…



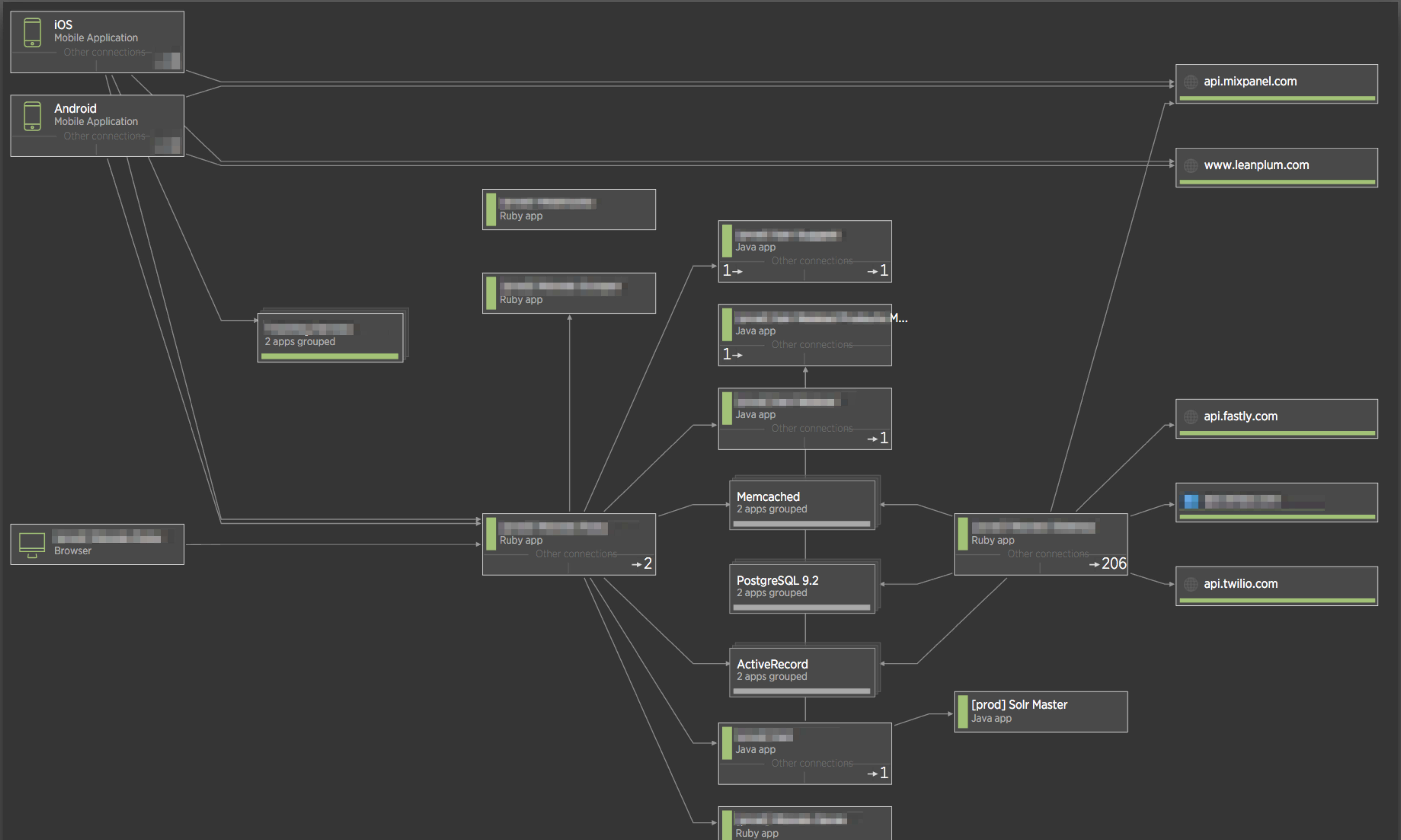
SWAP OLD AND NEW BACKENDS



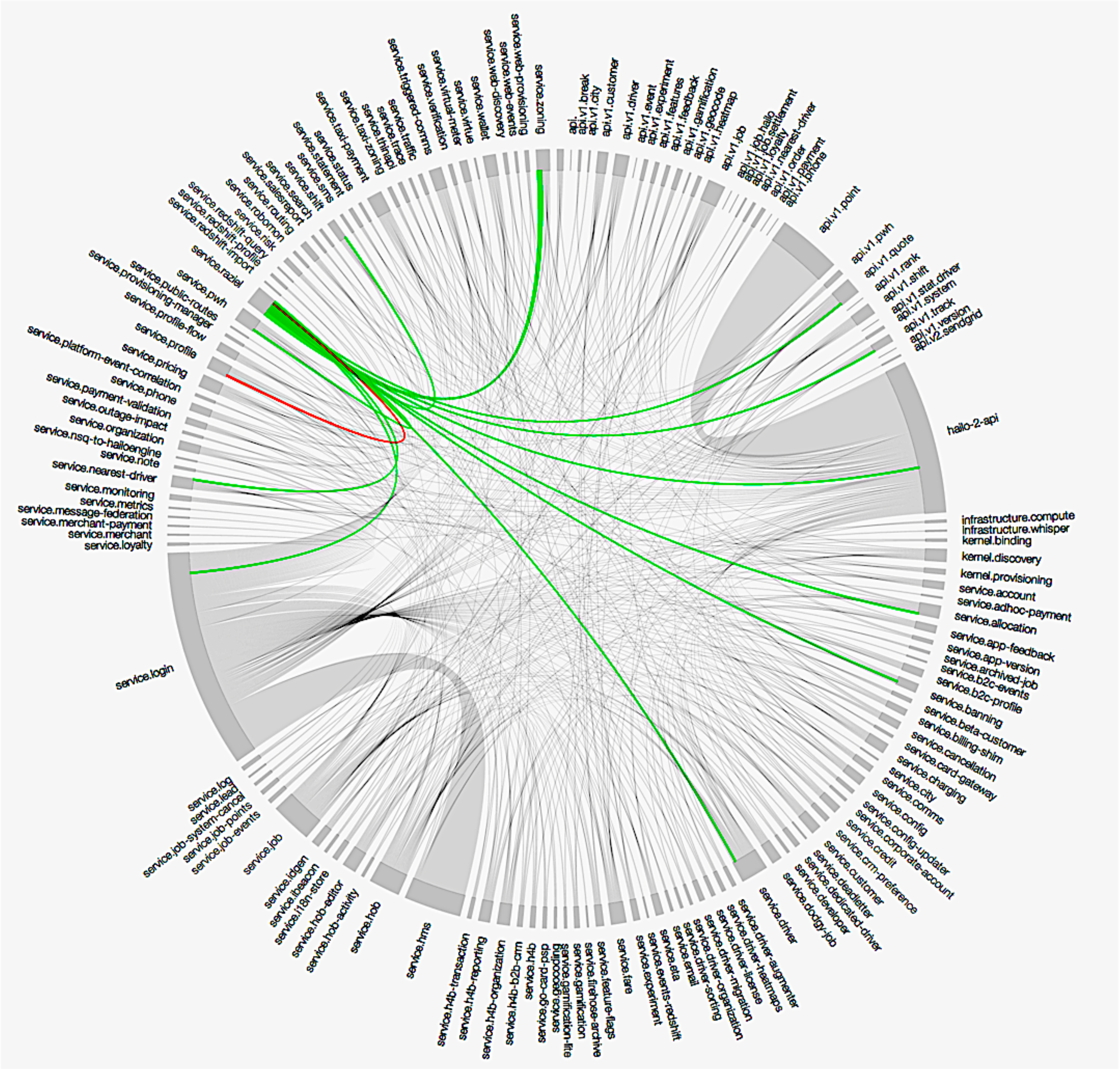
HORIZONTAL SHARDING **CONCLUSIONS**

- This is the final destination of any scalable architecture: just **add more boxes**
- Pretty sure we can now support 1,000 - 100,000 inserts/second by scaling out wide
- This effort took **2 months of 2 engineers**, including the migration, and we managed to do it with zero downtime.
- You can arrive there incrementally, like we did, without too much added cost. But don't start with this on a new application!

MICRO SERVICES **ARCHITECTURE:** NEW RELIC MAP



THOUGHTS ON **MICRO-SERVICES**



https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-3/

- The new micro-services infrastructure complexity **does not come for free**
- It requires new **code, new automation, testing, deployment, monitoring, graphing, maintenance and upgrades,** and comes with it's own unique set of bugs and problems.
- But the advantages, in this case, by far supersede the cost, particularly with **billion+ sized data sets, and/or large teams:**
  - Autonomy by ownership: a dedicated team for each service (aka Twitter model)
  - Each service can be scaled up independently.

CONCLUDING THOUGHTS

- Hopefully you can see that it is possible to **scale application to millions of users methodically, and incrementally.**

- Patterns presented here can be readily copied, and implemented on any application that's running slow, or having difficulty supporting a growing user-base. **Congrats, these are great problems to have!**

ACKNOWLEDGEMENTS

This is an early Wanelo team watching "Mean Girls" as part of cultural education.

Finally, our learnings and discovery of these solutions would not have been possible without:

- Obsessive monitoring and debugging, made possible by SmartOS, PostgreSQL and such tools as: **dTrace, htop, vfsstat, iostat, prstat, nagios, statsd, graphite**
- Excellent performance insight products from **NewRelic and Circonus**
- Help from the wonderful folks at **PGExperts and Joyent**
- And relentless professionalism, zeal and ingenuity of the **Wanelo's Engineering Team.**

Thanks!

github.com/wanelo
github.com/wanelo-chef

Wanelo's Technical Blog
building.wanelo.com

Personal Technical Blog:
kig.re

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