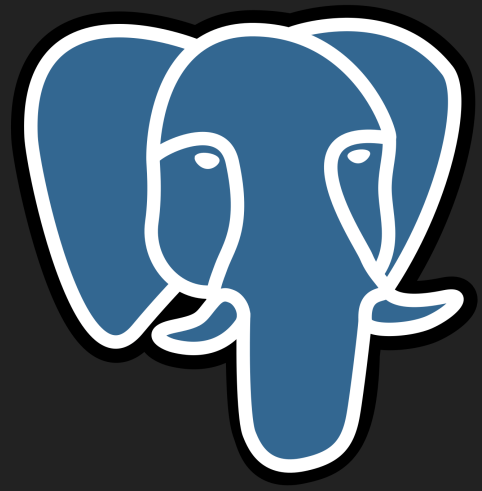
Puny to Powerful PostgreSQL Rails Apps



Andrew Atkinson

`#PgPunyPowerful`

bit.ly/PgPunyPowerful

bit.ly/PgPunyPowerful

Objectives

- Review 5 types of database scaling challenges and solutions
- Consider integrating the tools and techniques presented into your applications

Safe Migrations on Large, Busy Databases ✨

Maximizing Performance Of Database Connections 🔌

Maintaining Fast SQL Queries ⚡

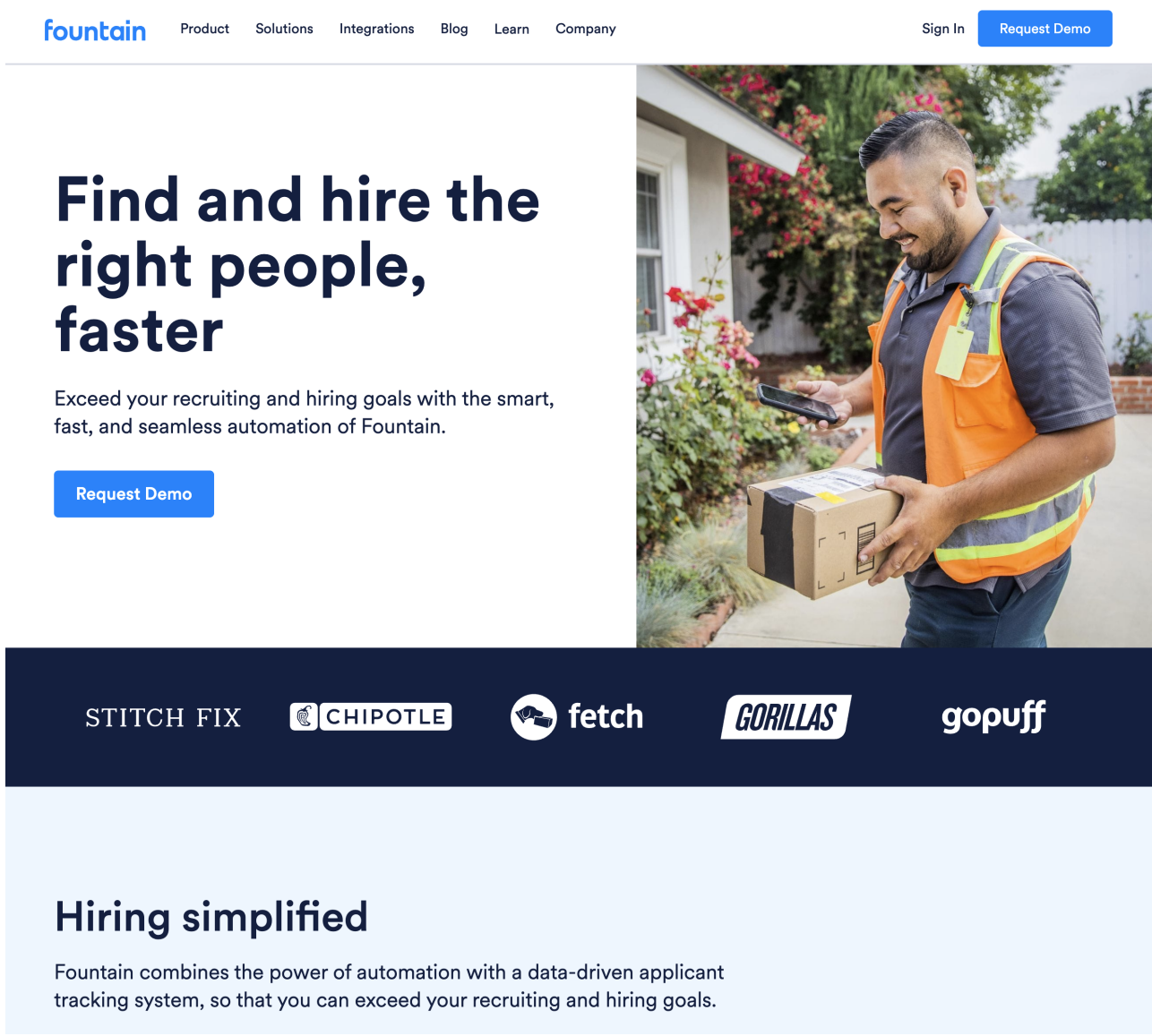
High Impact Database Maintenance 🔨

Implementing Replication and Partitioning 🐑

Fountain ⛲

- High Volume Hiring Platform
- Remote-first
- RailsConf Sponsor 🎉

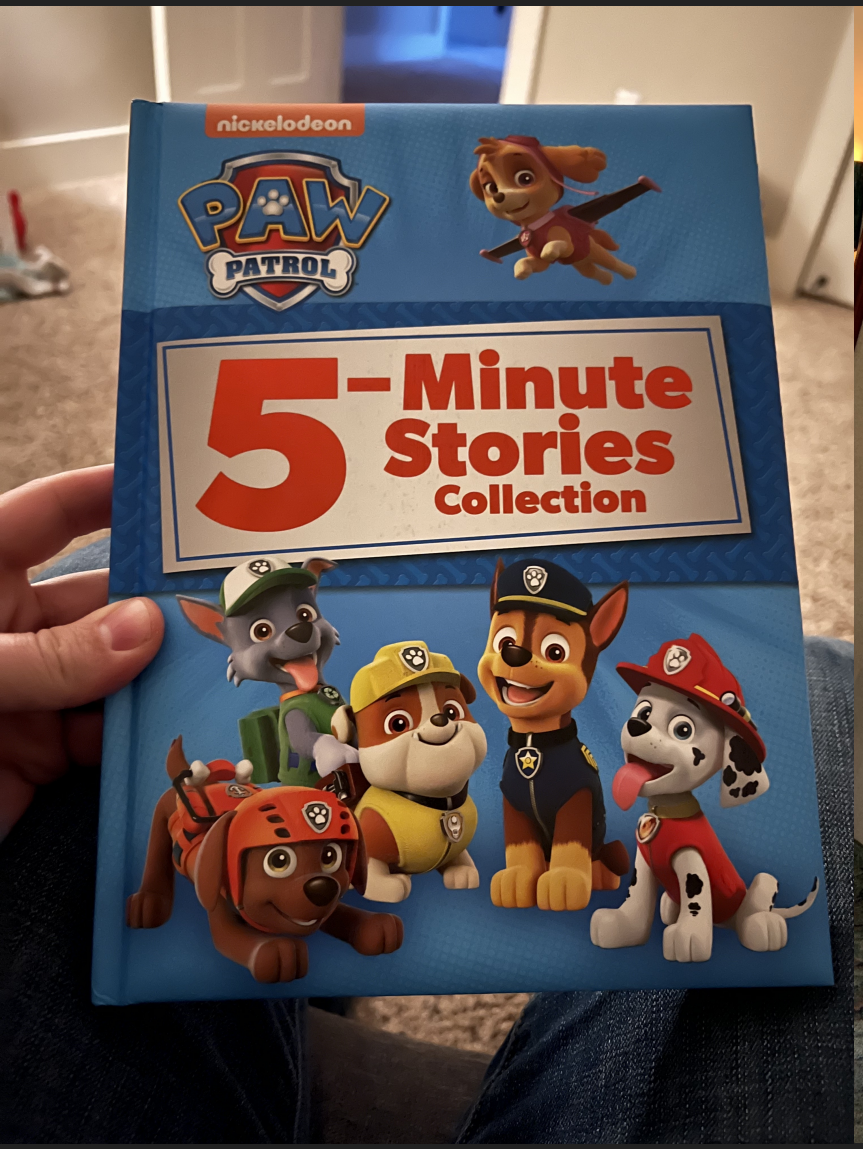
fountain.com/careers



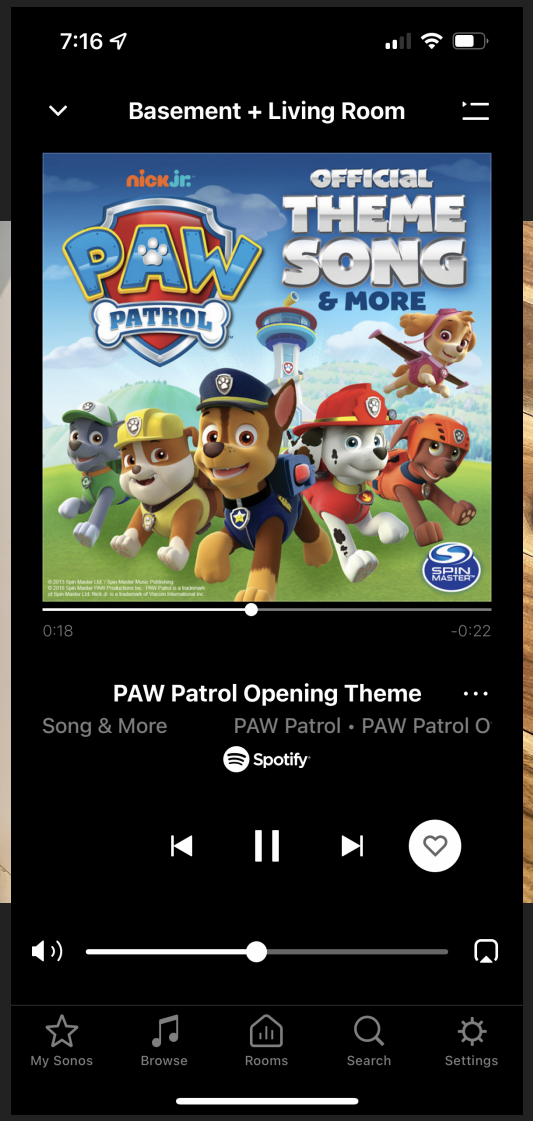
bit.ly/PgPunyPowerful

👋 About Me

- Staff Software Engineer, Fountain
- Work With Rails, PostgreSQL, Elasticsearch
- Dad of 2



bit.ly/PgPunyPowerful

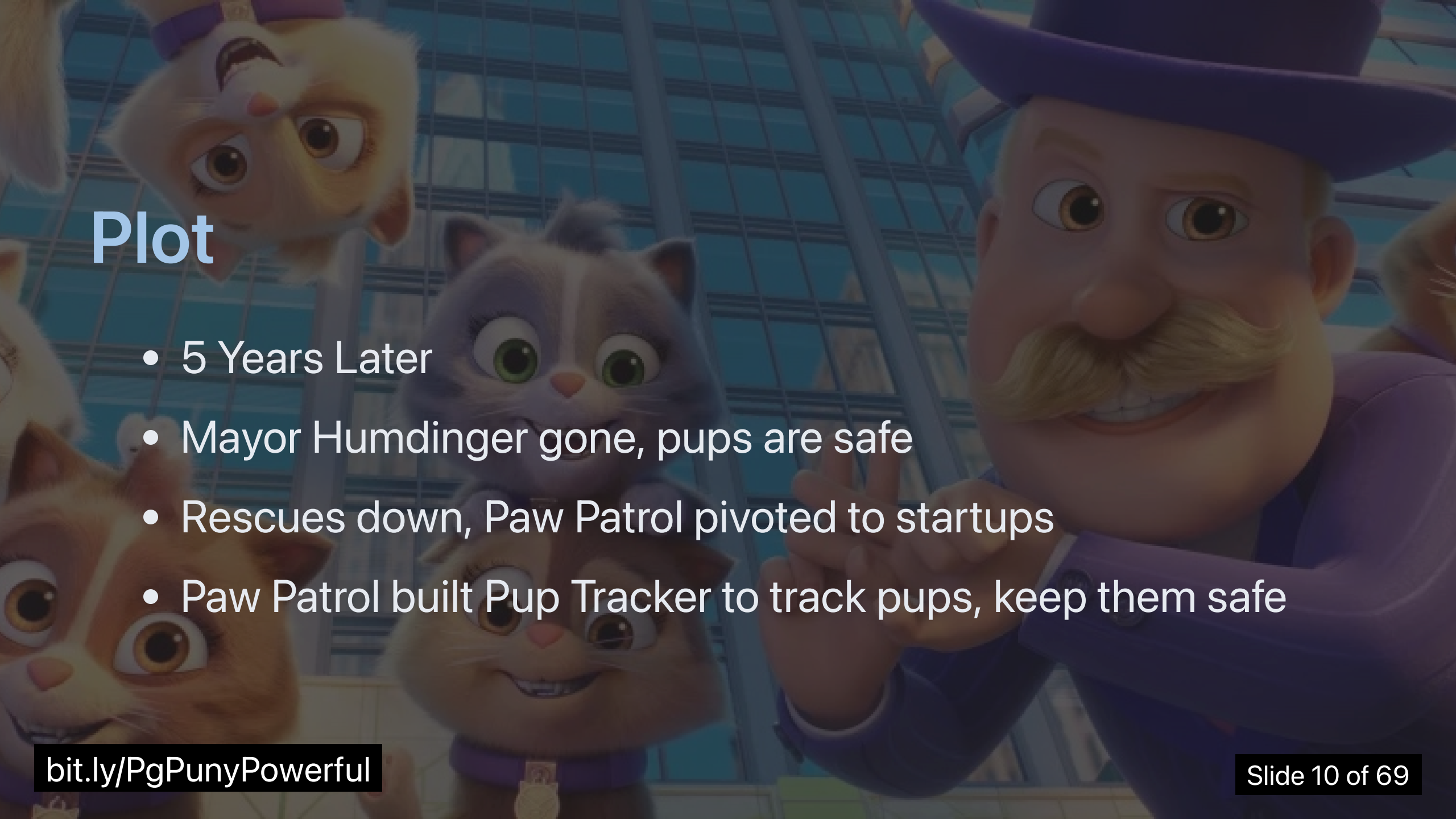


bit.ly/PgPunyPowerful

Paw Patrol 2

The SQL

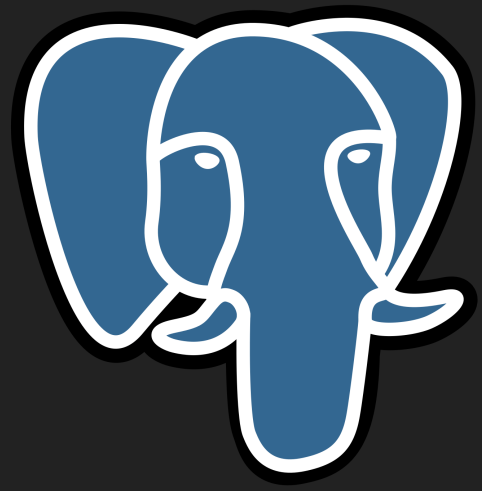
bit.ly/PgPunyPowerful



Plot

- 5 Years Later
- Mayor Humdinger gone, pups are safe
- Rescues down, Paw Patrol pivoted to startups
- Paw Patrol built Pup Tracker to track pups, keep them safe

Scaling Pup Tracker 🐶



- Pup Tracker built with Rails and PostgreSQL
- Used around the world
- 5 pups share their challenges and solutions

bit.ly/PgPunyPowerful

Liberty ✨

Newest Paw Patrol member, eager to make an impact

Database Migrations

Migrations on Large, Busy Tables ✨

As tables grew to millions of rows, making structure changes could take a long time and cause application errors

Check Constraints: Potentially Unsafe

```
class AddCheckConstraint < ActiveRecord::Migration[7.0]
  def change
    add_check_constraint :pups, "birth_date > '1990-01-01'",
    name: "birth_date_check"
  end
end
```

> Adding a check constraint blocks reads and writes in Postgres
>
> Source: Strong Migrations

Check Constraint: Safe Variation

> Add the check constraint without validating existing rows

Constraint added as `NOT VALID`

```
class AddCheckConstraint < ActiveRecord::Migration[7.0]
  def change
    add_check_constraint :pups, "birth_date > '1990-01-01'",
    name: "birth_date_check",
    validate: false
  end
end
```

Validate Separately

```
class ValidateCheckConstraint < ActiveRecord::Migration[7.0]
  def change
    validate_check_constraint :pups, name: "birth_date_check"
  end
end
```

Less disruptive, but consider running off-peak or in downtime.

> Validation acquires only a SHARE UPDATE EXCLUSIVE lock

Source: Understanding Check Constraints

Data Backfill: Potentially Unsafe

```
class AddCityIdToLocations < ActiveRecord::Migration[7.0]
  def change
    add_column :locations, :city_id, :integer
    Location.update_all(city_id: 1)
  end
end
```

> backfilling in the same transaction that alters a table keeps the table locked for the duration of the backfill.

Data Backfill: Safe Variation

> There are three keys to backfilling safely: batching, throttling, and running it outside a transaction.

Source: Strong Migrations

Safe Variation. Migration 1: Add Column.

```ruby
class AddLocationsCityId < ActiveRecord::Migration[7.0]
  def up
    add_column :locations, :city_id, :integer

    # Application sets `city_id` values for new rows
  end
end
```

Safe Variation. Second Migration: Backfill

```ruby
class BackfillCityId < ActiveRecord::Migration[7.0]
  disable_ddl_transaction!

  def up
    Location.unscoped.in_batches do |relation|
      relation.update_all(city_id: 1)
      sleep(0.01) # throttle
    end
  end
end
```

Migrations Good Practices ✨

- Studied potentially unsafe migrations from Strong Migrations
- Used safe variations
- Modified new rows and modified existing rows as separate operations

Marshall 🔌

Emergency responder,
extinguishes literal fires,
extinguishes database fires

Database Connections

bit.ly/PgPunyPowerful

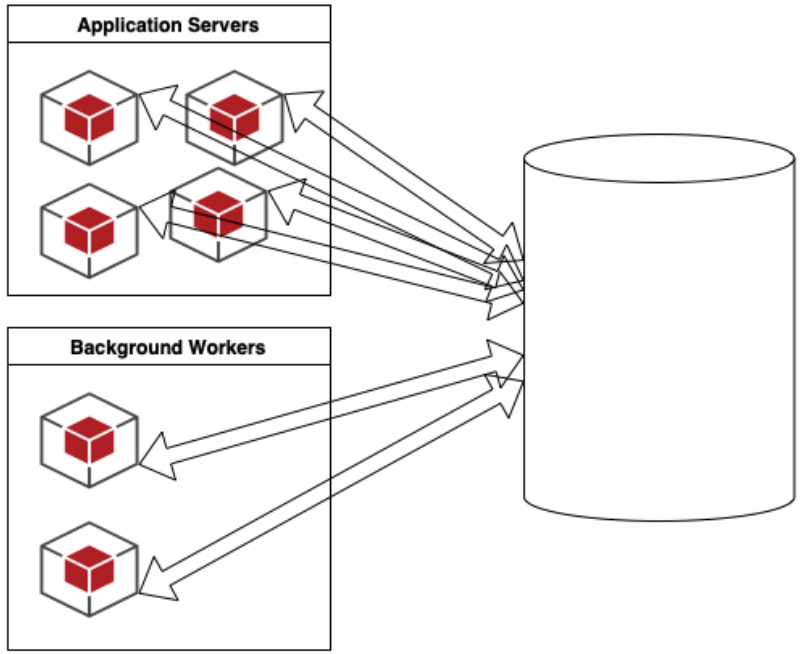
Database Connections 🔌

- Marshall learned database connections are finite and may be used up, blocking queries and causing downtime
- Marshall learned that app servers like Puma use connections
- Background workers like Sidekiq use connections

Active Record Connection Pool

Can set limits

> As demand for connections increases, it will create them until it reaches the connection pool limit.



Active Record Connection Pool and Sidekiq Concurrency 🔌

```
# config/database.yml
production:
  pool: 5
```

```
# config/sidekiq.yml
production:
  :concurrency: 5
```

Resource Consumption 🔌

- Each PostgreSQL client connection forks a new process and uses some of the available memory.
- Database server memory is finite
- Besides client connections, memory is also used by background processes and idle connections

Scaling Options

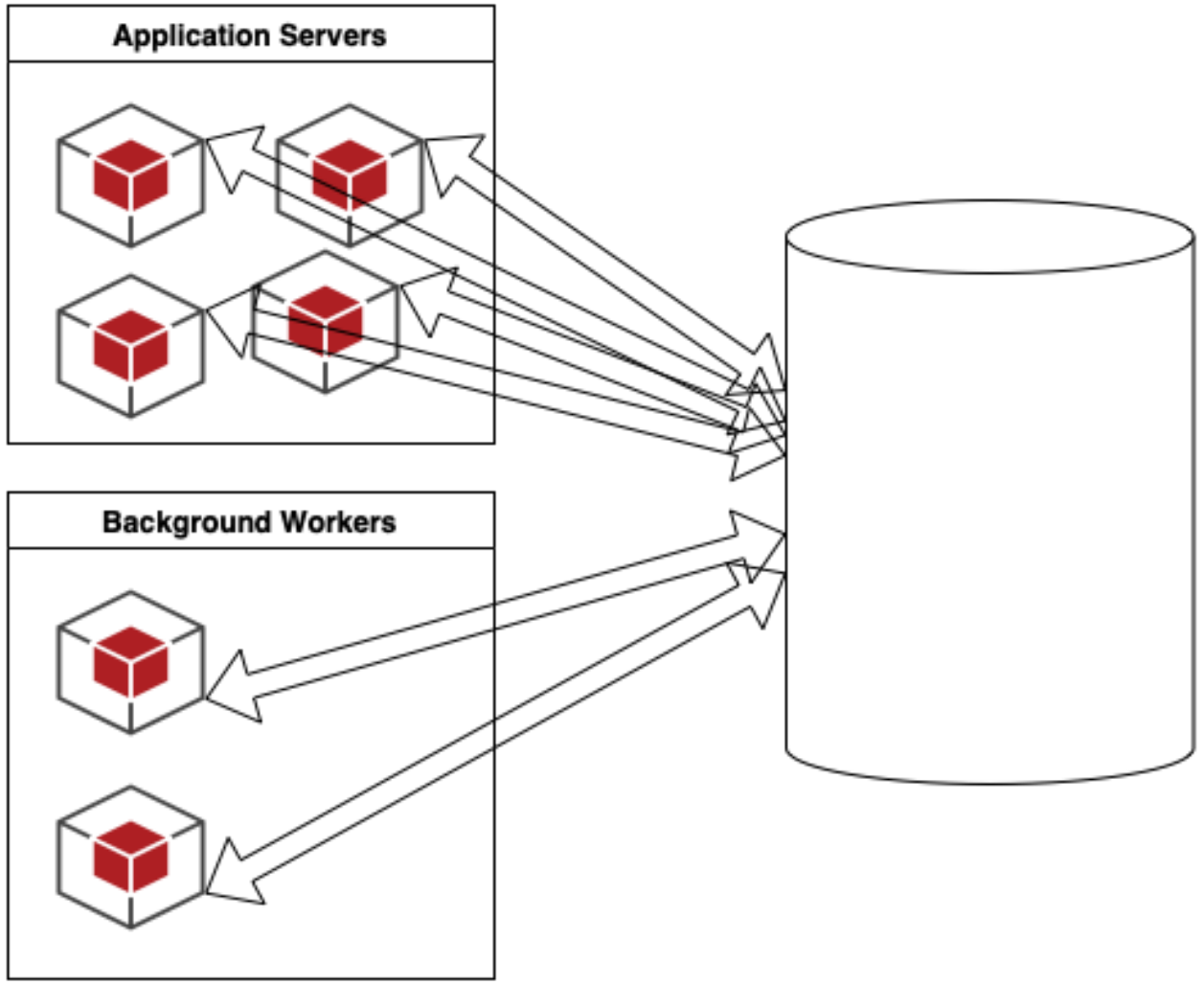
Marshall needed more memory or more efficient usage of limited connections

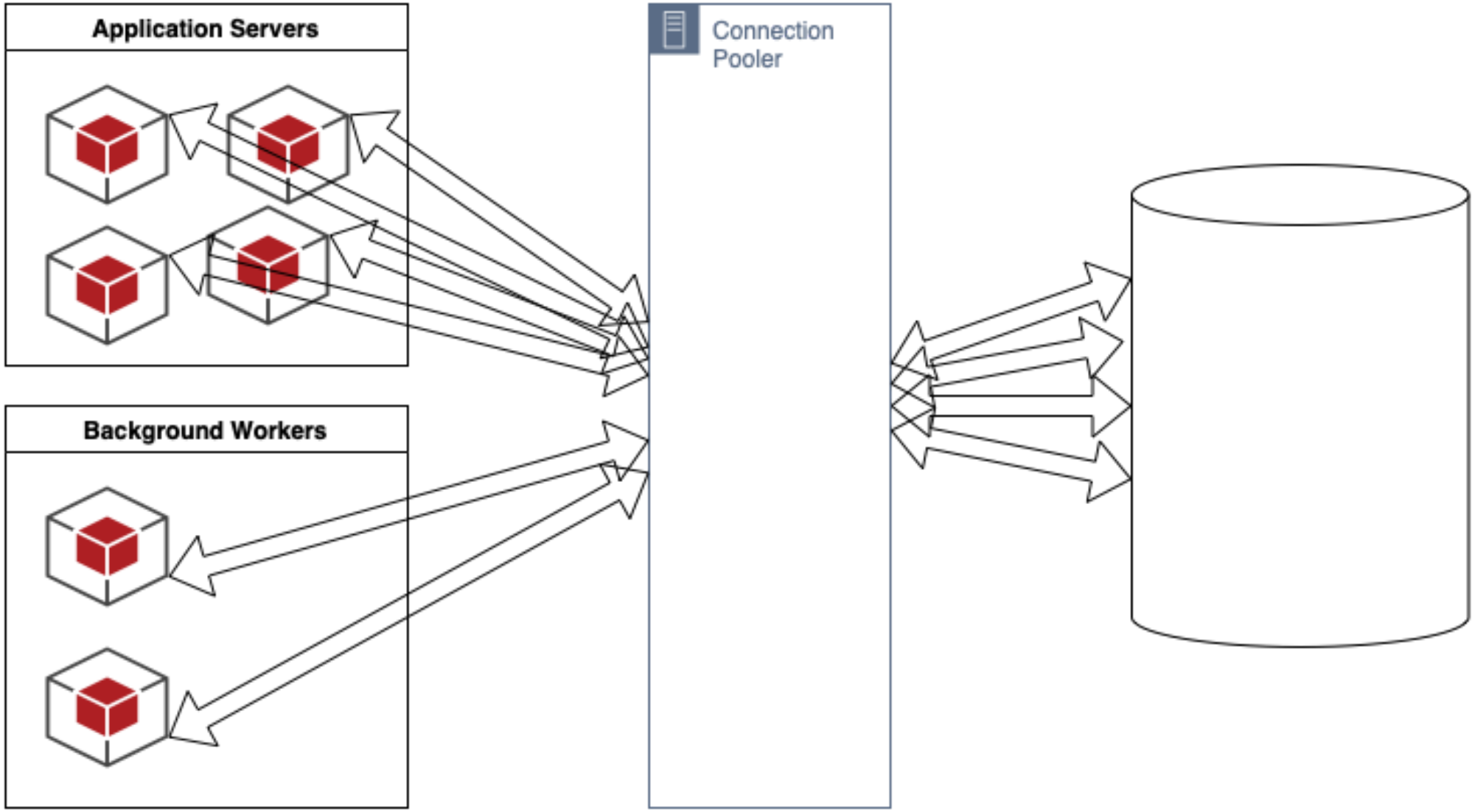
Option #1: Scale database server vertically, gain memory

Option #2: Use existing available connections more efficiently

Connection Poolers 🔌

- Connection Poolers are additional software, dedicated to efficient connection usage
- A Connection Pooler does not replace the Active Record Connection Pool
- A Pooler eliminates overhead in establishing new connections
- Poolers offer additional features like pooling modes


Application Servers
Background Workers


Application Servers
Background Workers
Connection
Pooler

Marshall Implemented Connection Pooling

Marshall installed and configured `PgBouncer` for Pup Tracker

```
[pgbouncer]
 listen_port = 6432
 listen_addr = localhost
 auth_type = md5
 auth_file = /usr/local/etc/userlist.txt
 logfile = pgbouncer.log
 pidfile = pgbouncer.pid
 admin_users = app_user
```

Connection Pooling improved performance without the need to scale vertically

Chase ⚡

Emergency responder,
first on the scene

Fast SQL Queries

Chase Investigates Slow Queries ⚡

Chase starts by looking at the query plan for a slow query

Example:

```
select * from locations where id = 1;
```

Chase uses `explain` and `explain (analyze)` to explore the query plan and execute the query.

Query Plan ⚡ (no indexes)

```
select * from locations where id = 1;
```

```
# explain (analyze) select * from locations where id =1;
                QUERY PLAN
-------------------------------------------------------------------------------
 Gather  (cost=1000.00..126613.85 rows=1 width=28)
  (actual time=0.474..332.856 rows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on locations (cost=0.00..125613.75 rows=1 width=28)
      (actual time=215.987..326.496 rows=0 loops=3)
         Filter: (id = 1)
         Rows Removed by Filter: 3333333
 Planning Time: 0.228 ms
 Execution Time: 332.898 ms
```

Query Plan ⚡ (with primary key constraint/index)

```
alter table locations add primary key (id);
\d locations
Indexes:
    "locations_pkey" PRIMARY KEY, btree (id)
```

```
# explain (analyze) select * from locations where id =1;
     QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using locations_pkey on locations
  (cost=0.43..8.45 rows=1 width=28) (actual time=0.070..0.072 rows=1 loops=1)
   Index Cond: (id = 1)
 Planning Time: 0.675 ms
 Execution Time: 0.111 ms
```

Chase Collects Query Stats ⚡

- Which queries are the most impactful? (most resource intensive)
- Need data. Enabled `pg_stat_statements`
- Query normalization (removing parameters):

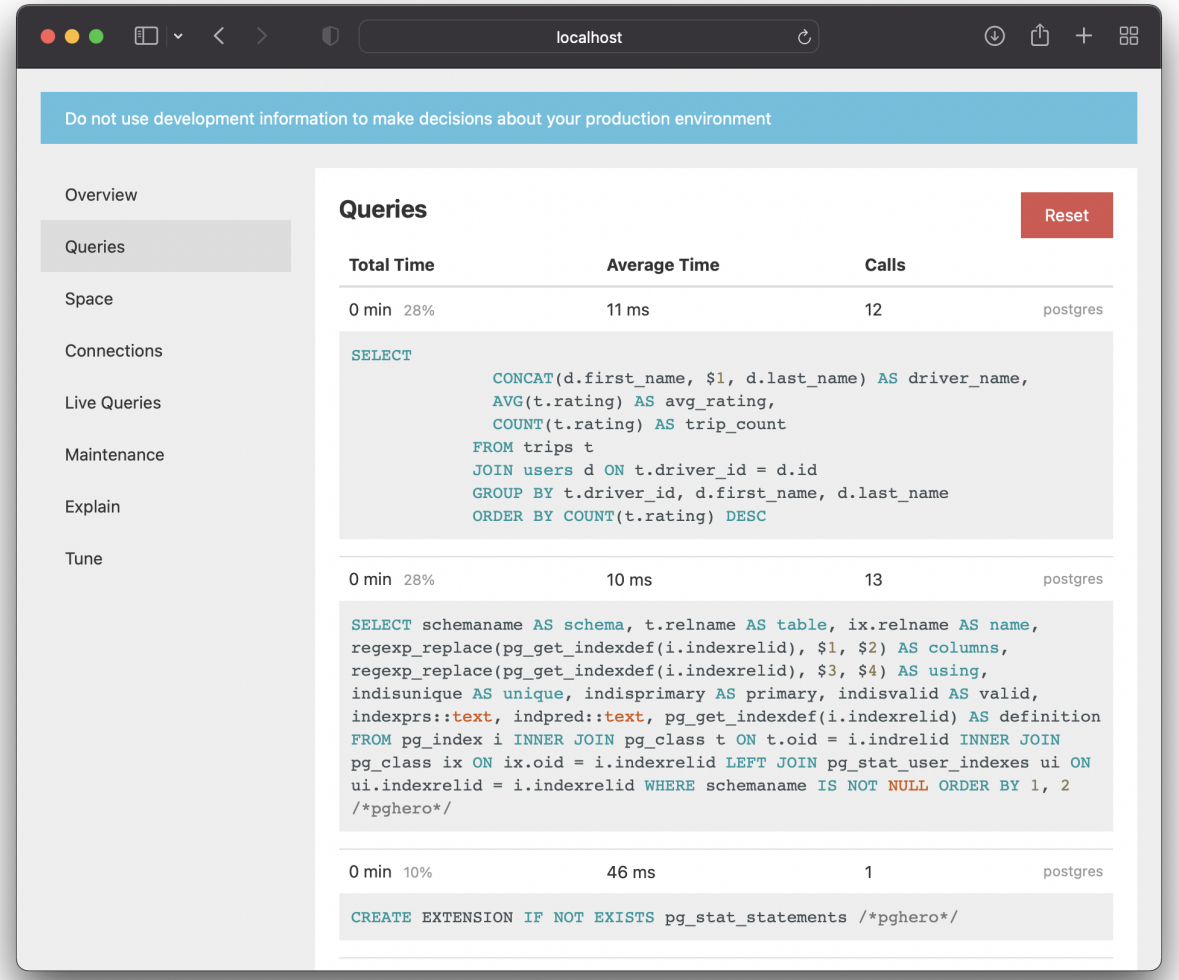
```
select * from locations where id = ?;
```

- Total Calls, Average Time

PgHero ⚡

> A performance dashboard for Postgres
> Source: PgHero

- Slow Queries
- Unused and Invalid Indexes
- High Connections



bit.ly/PgPunyPowerful

Query Investigation Process ⚡

Get query plan via `EXPLAIN`, make database object changes

OR

- Using Marginalia annotations (in Rails 7 🎉), find source location

  `/*controller:locations,action:index*/`

- Make query more selective (add scopes) or rewrite query

Improving Query Performance ⚡

- Ensured `ANALYZE` ran for table or ran manually (psql)
- Added missing indexes (replacing `sequential scan` with `index scan`)
- Rebuilt indexes that were heavily used and had high bloat (psql)
- Used performance analysis tools like pgMustard and pganalyze for insights

bit.ly/PgPunyPowerful

Rubble 🔨

Likes construction and maintenance

Database Maintenance

Database Maintenance 🔨

- `vacuum` ("garbage collect") and Autovacuum
- Updated stats via `analyze` (important for queries)
- Rebuilding Bloated Indexes
- Removing Unused Indexes

Manually Rebuild Indexes 🔨

- Rubble used `REINDEX CONCURRENTLY` on `PG >= 12`
- pg_repack `PG <= 11` is another option

Example

```
# reindex index concurrently locations_rating_idx;
REINDEX
Time: 50.108 ms
```

Automating Maintenance ⏰

- Rubble used pg_cron extension
- pg_cron is cron for database tasks (`PG >= 10`)
- Use the `cron.schedule()` function

```
--- Vacuum every day at 10:00am (GMT) / 3 AM PT
SELECT cron.schedule('0 10 * * *', 'VACUUM locations');
```

Database Maintenance Recap 🔨

- Made sure `vacuum` and `analyze` ran regularly
- Removed unused indexes
- Automated some maintenance tasks

Skye

Aerial response, takes in big picture.

Multiple Databases

Replication and Partitioning

Write and Read Workloads

Reads: `select`

Writes: `insert`, `update`, `delete`

- Pup Tracker had 10x more reads than writes
- All writes and reads were on a single database

Implementing Replication for Read Separation 🐑

- Skye set up a read replica to be used for the read workload
- Skye implemented PostgreSQL streaming replication (`PG >= 9.0`) (outside Rails)
- Skye used Rails' Multiple Databases (`>= 6.0`) to create a primary and replica configuration (inside Rails)

Database Configuration

`config/database.yml`

```yaml
production:
  primary:
    database: pup_tracker_production
    adapter: postgresql # obviously!
  replica:
    database: pup_tracker_production
    replica: true
```

Rails will not run migrations against replica

Application Record

`app/models/application_record.rb`

Set the `reading` role

```ruby
class ApplicationRecord < ActiveRecord::Base
  self.abstract_class = true

  connects_to database: {
      writing: :primary,
      reading: :replica
  }
end
```

Manual Query Relocation 🐏

```ruby
ActiveRecord::Base.connected_to(role: :reading) do

  # do a read query here!!
  # e.g. Pup.find_by(name: "Skye").locations.limit(5)

end
```

Consider replication lag

Automatic Query Relocation 🐑

Rails (`>= 6.0`) Automatic role switching

> For a GET or HEAD request the application will read from the replica unless there was a recent write.

Pup Tracker Data Growth 🪣

- Pup Tracker only queries most recent month of `locations` data, but the database has 100s of millions of rows and years of locations
- Query performance has worsened
- Vacuum takes forever to run

Skye Researches Partitioning 🪣

- Skye implemented PostgreSQL Table Partitioning (`PG >= 10`)
- Skye selected `Range` partitioning, `created_at` column partition key, monthly partitions

Partitioning Benefits 🪣

- Partitions that no longer reachable by application can be removed
- Greatly reduces quantity of data needed in database
- Smaller indexes, lower cost of table scans, faster maintenance

Implementation Challenges 🪣

- Migration to partitioned tables requires either some downtime or some dual writing (triggers)
- Need to test queries on partitioned table before rollout
- More operational complexity to manage unneeded and upcoming partitions

Skye used pgslice and pg_cron 🪣

- pgslice generates SQL to help migrate to a partitioned table
- Partition management can be scheduled using pg_cron

Some steps 🪣

```
CREATE TABLE "locations_intermediate" (LIKE "locations" ...)
PARTITION BY RANGE ("created_at");
```

```
CREATE TABLE "locations_202206"
PARTITION OF "locations_intermediate"
FOR VALUES FROM ('2022-06-01') TO ('2022-07-01');
```

And more. `pgslice` lists all the commands needed!

Partition Detachment 🪣

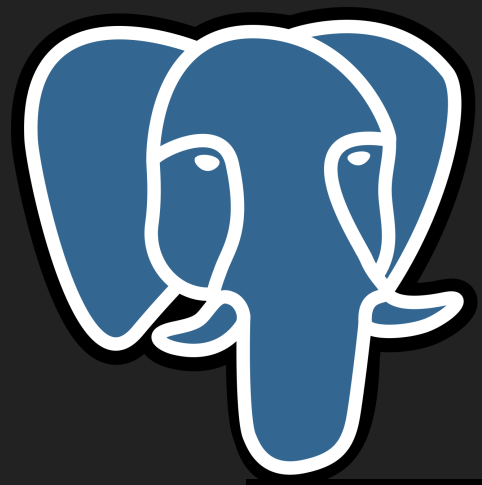
Partitions can be detached from parent table

```
ALTER TABLE locations
DETACH PARTITION locations_20220501 CONCURRENTLY;
```

A detached partition becomes a regular table that can be archived or deleted.

Summaries

Database Scaling Challenges and Solutions



Use Safe Migrations ✨

- Liberty used safe migrations suggested by Strong Migrations
- Liberty separated database changes into two steps, modifying new rows and modifying existing rows

Use Connection Pooling 🔌

- Marshall learned database connections are limited resources
- Marshall implemented a Connection Pooler to improve performance and scalability for Pup Tracker

Find and Fix Slow Queries ⚡

- Chase set up PgHero as a performance dashboard
- Chase identified and analyzed the most impactful slow queries
- Chase added indexes to improve queries, or sometimes rewrote them

Database Maintenance 🔨

- Rubble monitored background vacuum and `analyze` operations, sometimes performing them manually
- Rubble performed manual index rebuilds, sometimes scheduling them with `pg_cron`

Replication and Partitioning

- Skye created and configured replication between multiple databases
- Skye used the read replica to separate write and read workloads
- Skye implemented range partitioning to cycle out old data and improve performance

A Special Thank You To Reviewers 🐦

@be_haki	@jmcharnes	@ryanbooz
@davidrowley_pg	@andrewkane	@andygauge
@dwfrank	@lukasfittl	@kevin_j_m
@crash_tech		

👋 Thanks!

#PgPunyPowerful

bit.ly/PgPunyPowerful

- Change new rows and existing rows separately
- Use a connection pooler like PgBouncer
- Fix the most resource intensive slow queries
- Rebuild bloated indexes
- Writes on primary, reads on replica
- Partition very large tables