pganalyze

LEIGH HALLIDAY JOSH ALLETTO

Advanced Database Programming with Rails and Postgres

Advanced Database Programming with Rails

PostgreSQL is stable, cost-efficient, and versatile. Paired with its high performance this makes it one of the most popular SQL databases out there.

It can handle large amounts of data and complex operations. In this eBook, we walk through three key skills that will help you become a more rounded Postgres and Ruby on Rails developer.

We will introduce you to the following topics and provide you with realistic, real-world examples:

- Using Subqueries in Rails
- Using Materialized Views in Rails
- Creating Custom Postgres Data Types in Rails

We walk through three key skills you will want to make yourself more familiar with when it comes to working with Postgres and Rails: Using subqueries, materialized views, and creating custom data types in Rails.

ABOUT

Firstly, we will have a look at Active Record and how it lets you access raw SQL. We look at 5 business requests for data, translate them into SQL, then into Rails code to find the requested data.

Then, we show how to utilize both, **views and materialized views**, within Ruby on Rails with a real-world sized dataset of hockey teams and their top scorers.

Lastly, we create **domain types and composite types in PostgreSQL** and show how to hook into the Rails Attributes API to help instantiate them as objects that Ruby can use.

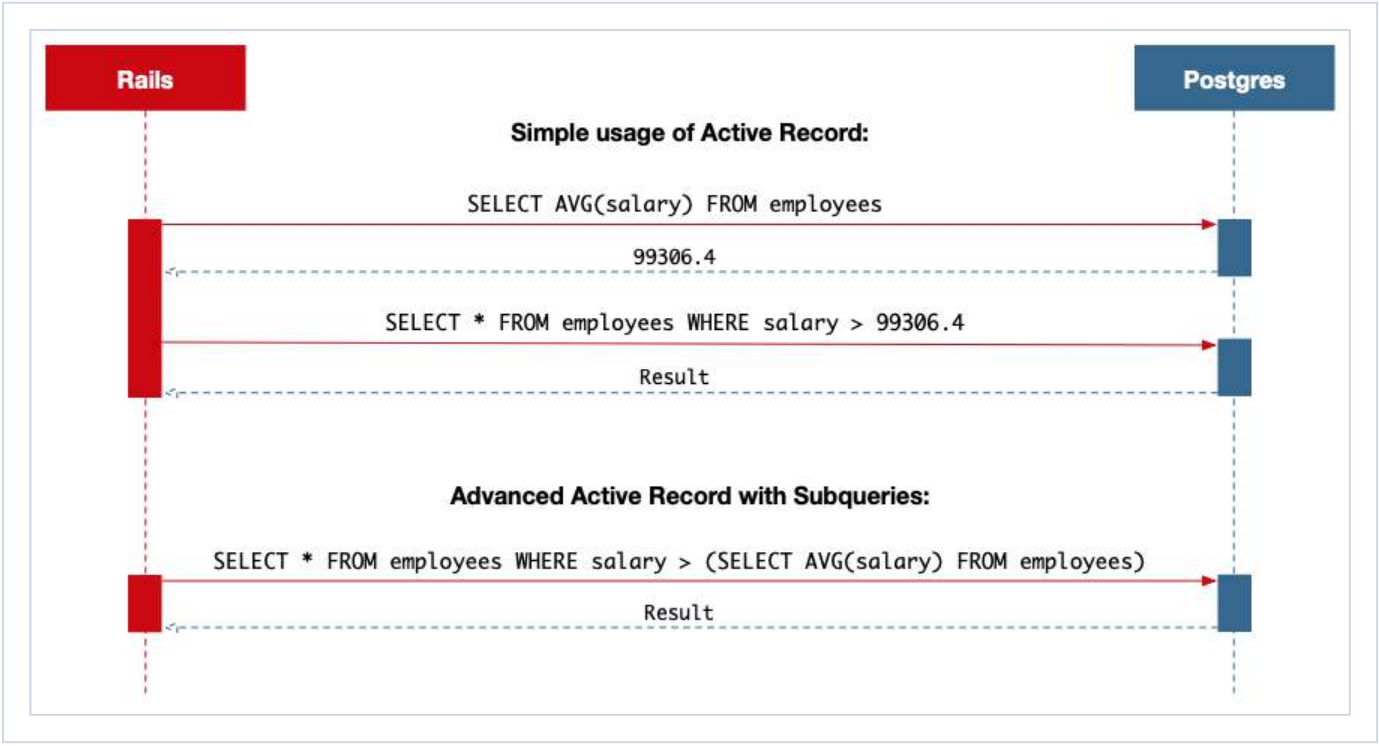
Without further ado, let's jump in!

Using Subqueries in Rails

Active Record provides a great balance between the ability to perform simple queries simply, and also the ability to access the raw SQL sometimes required to get our jobs done. In this eBook, we will see a number of real-life examples of business needs that may arise at our jobs.

They will come in the form of a request for data from someone else at the company, where we will first translate the request into SQL, and then into the Rails code necessary to find those records. We will be covering five different types of subqueries to help us find the requested data.

Let's take a look at why subqueries matter:



In the first case, without subqueries, we are going to the database twice: First to get the average salary, and then again to get the result set. With a subquery, we can avoid the extra roundtrip, getting the result directly with a single query.

Working with Active Record in Rails

Active Record is a little like a walled garden. It protects us as developers (and our users) from the harsh realities of what lies beyond those walls: Differences in SQL between databases (MySQL, Postgres, SQLite), knowing how to properly escape strings to avoid SQL injection attacks, and generally providing an elegant abstraction to interact with our database using the language of our choice, Ruby.

But, SQL is extremely powerful! By understanding the SQL that Active Record is executing, we can open the gate in our walled garden to **reach beyond what you may think is possible to accomplish in Rails**, taking advantage of optimizations and flexibility that may be difficult to achieve otherwise.

What are Subqueries in Rails

In this eBook, we will be learning how to use subqueries in Active Record. Subqueries are what

their name implies: A query within a query. We will look at how to embed subqueries into the `SELECT`, `FROM`, `WHERE`, and `HAVING` clauses of SQL, to meet the demands of our business counterparts who are asking to view data in different and interesting ways.

We'll be playing the role of a developer fielding questions from HR. They are asking for reports about our employees at BCE (Best Company Ever), and we'll do our best to find the data they need using Active Record.

The source code for our examples is available on GitHub.

An Overview of our Data

Our database has 4 tables:

- **roles**: The job roles of our employees (Finance, Engineering, Sales, HR, etc...)
- **employees**: The people that work for BCE
- **performance_reviews**: Performance reviews carried out by an employee's manager, giving them a score between 0 and 100
- **vacations**: Keeping track of when employees have taken vacation

Using https://dbdiagram.io/ we're able to see how these tables relate to each other:

If you are following along, the `rails db:seed` command will generate 1,000 employees, 1,000 vacations, and 10,000 performance reviews.

The Where Subquery

Now that we have our data set and we're ready to go let's help our HR team with their first request:

> *Could you please find us all the employees that make more than the average salary at BCE?*

Here we will use a subquery within the `WHERE` clause to find the employees that match HR's request:

SQL

```sql
SELECT *
FROM employees
WHERE
  employees.salary > (
    SELECT avg(salary)
    FROM employees)
```

My first attempt at replicating the query above looked like this:

SQL

```
Employee.where('salary > :avg', avg: Employee.average(:salary))
```

But **what it produced was two queries**:
One to find the average, and a second to query employees with a salary greater than that number.

Not technically wrong, but **it doesn't line up with the SQL we were going for**. There is also a potential performance impact of two round-trip requests to the database server, along with potential inconsistencies if a new employee making $1B/year is hired between queries one and two. Although this is unlikely in this particular scenario, it's something to consider as a potential risk.

SQL

```
1   -- find the average
2   SELECT AVG("employees"."salary") FROM "employees"
3   -- find the employees
4   SELECT "employees".* FROM "employees" WHERE (salary > 99306.4)
```

What we shouldn't forget about Active Record is that certain methods, such as `average(:salary)`, actually execute the query and return a result, while other methods implement Method Chaining, allowing you to chain multiple Active Record methods together, building up more complex SQL statements prior to their execution.

SQL

```
1   Employee.where('salary > (:avg)', avg: Employee.select('avg(salary)'))
```

This produces the SQL we want, but note that we

had to wrap the placeholder condition `:avg` in brackets, because the database wants subqueries wrapped in brackets as well.

Because the seed data is generated randomly, your results will vary from mine, but I am seeing 487 matching employees, getting a result that looks like this:

BASH

```
1   #<ActiveRecord::Relation [#<Employee id: 4, role_id: 5, name: "Bob Williams", salary:
    127053.0, created_at: "2020-04-26 18:42:53", updated_at: "2020-04-26 18:42:53">,
    #<Employee id: 5, role_id: 4, name: "Bob Florez", salary: 149218.0, created_at: "2020-
    04-26 18:42:53", updated_at: "2020-04-26 18:42:53">, ...]>
```

Where Not Exists

Let's have a look at another request from HR:

> *We would like to encourage employees to have a healthy work-life balance, and were hoping you could provide us with a list of all the employees who have yet to take any vacation time.*

For this case, `NOT EXISTS` is a perfect fit, since it only matches records that **do not** have a match in the subquery. An alternative is to perform a left outer join, only choosing the records with no matches on the right side. This is referred to as an anti-join, where the purpose of the join is to find records that

do not have a matching record.

SQL

```sql
SELECT *
FROM employees
WHERE
  NOT EXISTS (
    SELECT 1
    FROM vacations
    WHERE vacations.employee_id = employees.id)
```

If you're interested in the **LEFT OUTER JOIN** equivalent, it might look like this:

SQL

```sql
SELECT employees.*
FROM
  employees
  LEFT OUTER JOIN vacations ON vacations.employee_id = employees.id
WHERE vacations.id IS NULL
```

The subquery depends on a match between the `employees.id` column and the `vacations.employee_id` column, making it a correlated subquery. Because Rails follows standard naming conventions when querying (the downcased plural form of our model), we can add the above condition into our subquery without too much difficulty.

RUBY

```ruby
Employee.where(
  'NOT EXISTS (:vacations)',
  vacations: Vacation.select('1').where('employees.id = vacations.employee_id')
)
```

Using my seed data, I am seeing **369** employees that have yet to take any vacations.

BASH

```
#<ActiveRecord::Relation [#<Employee id: 2, role_id: 2, name: "Alice Florez", salary: 86920.0, created_at: "2020-04-26 18:42:53", updated_at: "2020-04-26 18:42:53">, #<Employee id: 5, role_id: 4, name: "Bob Florez", salary: 149218.0, created_at: "2020-04-26 18:42:53", updated_at: "2020-04-26 18:42:53">, ...]>
```

The Select Subquery

Here's another request from HR:

> *Could you please provide us with a list of employees, including the average salary of a BCE employee, and how much this employee's salary differs from the average?*

Here's how we could work on this:

SQL

```sql
SELECT
  *,
  (SELECT avg(salary)
    FROM employees) avg_salary,
  salary - (
    SELECT avg(salary)
    FROM employees) above_avg
FROM employees
```

Because the subquery is repeated, we can save ourselves a little bit of hassle by placing the subquery SQL into a variable that we'll embed into the outer query. The `to_sql` method is perfect for this, but it's also fantastic to peak into the SQL that Rails is producing without actually executing the query.

RUBY

```ruby
avg_sql = Employee.select('avg(salary)').to_sql

Employee.select(
  '*',
  "(#{avg_sql}) avg_salary",
  "salary - (#{avg_sql}) avg_difference"
)
```

This query does not limit the results in any way, but instead selects two additional columns (`avg_salary` and `avg_difference`). Looking at the first three results,

I am seeing:

BASH

```
[
  {“id”=>1, “role_id”=>1, “name”=>”Joe Serna”, “salary”=>86340.0, “avg_salary”=>99306.4, “avg_difference”=>-12966.399999999994},
  {“id”=>2, “role_id”=>2, “name”=>”Alice Florez”, “salary”=>86920.0, “avg_salary”=>99306.4, “avg_difference”=>-12386.399999999994},
  {“id”=>3, “role_id”=>3, “name”=>”Amanda Florez”, “salary”=>93600.0, “avg_salary”=>99306.4, “avg_difference”=>-5706.399999999994}
]
```

As with any SQL query, there are often many ways to arrive at the same result. In this example we used subqueries to find the average employee salary, but it may have been better to use window functions instead. They give us the same result, but provide a simpler query which is actually more performant as well. Even on a small dataset of 1000 employees, this query takes approximately 12ms vs 18ms for the subquery equivalent.

SQL

```
SELECT
  *,
  avg(salary) OVER () AS avg_salary,
  salary - avg(salary) OVER () AS avg_salary
FROM
  employees
```

The window function approach is actually easier to write in Rails as well!

RUBY

```ruby
Employee.select(
  '*',
  "avg(salary) OVER () avg_salary",
  "salary - avg(salary) OVER () avg_difference"
)
```

The From Subquery

Here's again from HR:

> *We'd like to know the average performance review score given across all our managers.*

After clarifying with HR, they are looking to take the average score each manager has given, and then take the average of those averages. In other words, the average average. When you are dealing with an **aggregate of aggregates**, it needs to be accomplished in two steps. This can be done using a subquery as the `FROM` clause, essentially giving us a temporary table to then select from, allowing us to find the average of those averages.

SQL

```sql
SELECT avg(avg_score) reviewer_avg
FROM (
  SELECT reviewer_id, avg(score) avg_score
  FROM performance_reviews
  GROUP BY reviewer_id) reviewer_avgs
```

To keep our Ruby code clean, we'll place the subquery into a variable which can then be embedded into the main query.

RUBY

```ruby
from_sql =
  PerformanceReview.select(:reviewer_id, 'avg(score) avg_score').group(
    :reviewer_id
  ).to_sql

PerformanceReview.select('avg(avg_score) reviewer_avg').from(
  "(#{from_sql}) as reviewer_avgs"
).take.reviewer_avg
```

The result of this query is `50.652`. This makes sense given that the seed data used a random value between 1 and 100 (`rand(1..100)`).

The Having Subquery

Let's look at one last request from HR:

> *Certain reviewers are consistently giving low performance review scores. Could you find us a list of all the managers whose average score is 25% below our company average? We need to find out what is happening.*

We will start by joining the `employees` table to the `performance_reviews` table where the employee is the reviewer (a manager), and then take their average

score. Then we will filter out these managers using a `HAVING` clause to only include those whose score increased by 25% is still lower than the company average.

SQL

```sql
SELECT
  employees.*,
  avg(score) avg_score,
  (SELECT avg(score)
    FROM performance_reviews) company_avg
FROM
  employees
  INNER JOIN performance_reviews
    ON performance_reviews.reviewer_id = employees.id
GROUP BY employees.id
HAVING
  avg(score) < 0.75 *
    (SELECT avg(score)
    FROM performance_reviews)
```

You'll notice that we actually included **two** subqueries in the above SQL. Because the SQL was saved to a variable (`avg_sql`), we were able to reuse this both within the `SELECT` portion of the query, and also within the `HAVING` clause.

RUBY

```ruby
avg_sql = PerformanceReview.select('avg(score)').to_sql

Employee.joins(:employee_reviews).select(
  'employees.*',
  'avg(score) avg_score',
  "(#{avg_sql}) company_avg"
).group('employees.id').having("avg(score) < 0.75 * (#{avg_sql})")
```

The result of this query gives me **103** employees, and the first three of them look like:

BASH

```
1  [
2    {“id”=>173, “role_id”=>1, “name”=>”Bob Williams”, “salary”=>109206.0, “avg_
   score”=>23.75, “company_avg”=>50.04},
3    {“id”=>390, “role_id”=>5, “name”=>”Bob Serna”, “salary”=>127559.0, “avg_score”=>26.0,
   “company_avg”=>50.04},
4    {“id”=>802, “role_id”=>4, “name”=>”Alice Halliday”, “salary”=>94956.0, “avg_
   score”=>35.88, “company_avg”=>50.04}
5  ]
```

Conclusion

In this section of the eBook, we were able to see a number of (somewhat) real-life examples of real business needs translating first into SQL, and then into the Rails code necessary to find those records. A backend developer’s career will consist in most likely hundreds of similar requests!

Active Record gives us the ability to perform simple queries simply, but also lets us access the raw SQL which is sometimes required to get our jobs done. Subqueries are a perfect example of that, and we saw how to create subqueries in Rails and Active Record in the `SELECT`, `FROM`, `WHERE`, and `HAVING` clauses of an SQL statement. As we have seen in the examples above, with the expressiveness of Active Record, one doesn’t have to resort to writing completely in SQL to use a subquery.

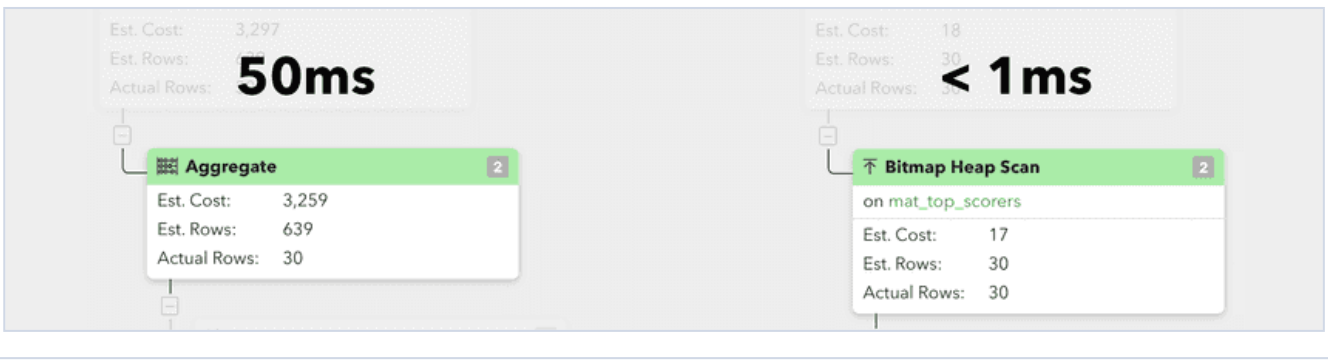
Materialized Views in Ruby on Rails

It's every developer's nightmare: SQL queries that get large and unwieldy. This can happen fairly quickly with the addition of multiple joins, a subquery and some complicated filtering logic. I have personally seen queries grow to nearly one hundred lines long in both the financial services and health industries.

Luckily Postgres provides two ways to encapsulate large queries: Views and Materialized Views. In this section of the eBook, we will cover in detail how to utilize both views and materialized views within **Ruby on Rails**, and we can even take a look at creating and modifying them with database migrations.

Our example will be a real-world sized dataset of hockey teams and their top scorers. If you'd like to follow along, the source code can be found here.

We'll also talk a bit about the performance benefits that a **Materialized View** can bring to your app:



What is a view?

A view allows us to query against the result of another query, providing a powerful way of abstracting away a complex query full of joins, conditions, groupings, and any other clause that can be added to an SQL query. Looking at the query below, it isn't overly complex, but it **does** include 3 joins, grouping by a number of fields to aggregate the numbers of goals scored for a player each season.

It takes approximately 450ms to execute on my computer. I am using seed data that generates 31 teams, each playing 200 games in a season, scoring 20 goals per game... a little unrealistic, but I wanted the dataset used to be substantial!

SQL

```sql
SELECT
  players.name AS player_name,
  players.id AS player_id,
  players.position AS player_position,
  matches.season AS season,
  teams.name AS team_name,
  teams.id AS team_id,
  count(goals.id) AS goal_count
FROM goals
  INNER JOIN players ON (goals.player_id = players.id)
  INNER JOIN matches ON (goals.match_id = matches.id)
  INNER JOIN teams ON (goals.team_id = teams.id)
GROUP BY players.id, teams.id, matches.season
```

A view allows us to take the final result of this query, and query **against** that as if it were any other table. You can see why views can come in handy in many different scenarios. They allow for the succinct abstraction of a complicated query, and allow us to re-use this logic in a simple to understand way.

Now, we could make a new view by running `CREATE VIEW` in Postgres. But, as we all know, one-off schema changes are hard to keep track of. Instead, let's try something thats closer to how Rails does things. How does that look like? First things first, we'll create a view using Scenic. Scenic gives us the ability to define migrations that create, update, or drop views, just as you're used to doing with regular tables in Rails.

BASH

```
1   rails g scenic:view top_scorers
```

This will generate two files. The first is named `db/views/top_scorers_v01.sql`, and in it we will paste the SQL for the underlying query (from above). The second is `db/migrate/[date]_create_top_scorers.rb`, and this is where the migration will live to migrate/rollback the creation of our view:

RUBY

```ruby
class CreateTopScorers < ActiveRecord::Migration[6.0]
  def change
    create_view :top_scorers
  end
end
```

With the view in place, we can now query against it. This query takes approximately 50ms to execute.

SQL

```sql
SELECT *
FROM top_scorers
WHERE
  team_name = 'Toronto Maple Leafs'
ORDER BY goal_count DESC
```

By creating a model in Rails, we can interact with it much like we would be able to with a typical model which is backed by a table. First things first, let's define the model, letting Rails know it is **read-only**. Whilst some views can be updated, this view contains a top-level `GROUP BY` clause and thus can't be updated.

RUBY

```ruby
# app/models/top_scorer.rb
class TopScorer < ApplicationRecord
  def readonly?
    true
  end
end
```

Now we can perform the same SQL query using the `TopScorer` model:

```ruby
TopScorer.where(team_name: 'Toronto Maple Leafs').order(goal_count: :desc)
```

What makes a view materialized?

A regular view still performs the underlying query which defined it. **It will only be as efficient as its underlying query is**. This means, if the larger query discussed above takes 450ms to execute, executing `SELECT * FROM top_scorers` will also take 450ms.

Materialized views take regular views to the next level, though they aren't without their drawbacks. The difference is that they save the result of the original query to a cached/temporary table. When you query a materialized view, you aren't querying the source data, rather the cached result.

This can provide serious performance benefits, especially considering you can index materialized views. But, when the underlying data from the source tables is updated, the materialized view becomes out of date, serving up an older cached version of the data. We can resolve this by

refreshing the materialized view, which we'll get to in a bit.

Creating a materialized view

Just like we saw with our regular view, materialized views begin the same way, by executing a command to generate a new view migration: `rails g scenic:view mat_top_scorers`. This produces two files, the first of which contains the SQL to produce the underlying view of the data. The difference is in the migration, passing in `materialized: true` to the `create_view` method. Also **notice that we are able to add indexes to the materialized view**.

RUBY

```ruby
class CreateMatTopScorers < ActiveRecord::Migration[6.0]
  def change
    create_view :mat_top_scorers, materialized: true

    add_index :mat_top_scorers, :player_name
    add_index :mat_top_scorers, :player_id
    add_index :mat_top_scorers, :team_name
    add_index :mat_top_scorers, :team_id
    add_index :mat_top_scorers, :season
  end
end
```

Utilizing a materialized view

Like a regular view, we are able to define an ActiveRecord model that can query it. Also notice that we can define relationships which point to other ActiveRecord models. If you didn't know, you might not even realize it is pointing to a materialized view, except for the `readonly?` method which was defined.

RUBY

```ruby
class MatTopScorer < ApplicationRecord
  belongs_to :player
  belongs_to :team
  belongs_to :match

  def self.top_scorer_for_season(season)
    where(season: season).order(goal_count: :desc).first
  end

  def readonly?
    true
  end
end
```

Let's take our new materialized view for a spin! Running the query `select * from mat_top_scorers`, which took 450ms as a view, takes 5ms as a materialized view, **90x faster**! The ruby code below, which took 50ms as a view, takes under 1ms to execute!

RUBY

```ruby
MatTopScorer.where(team_name: 'Toronto Maple Leafs').order(goal_count: :desc)
```

For a side-by-side comparison, this performs the same query on both views:

RUBY

```
irb(main):001:0> TopScorer.where(team_name: 'Toronto Maple Leafs').count
   (60.2ms)  SELECT COUNT(*) FROM "top_scorers" WHERE "top_scorers"."team_name" = $1
[["team_name", "Toronto Maple Leafs"]]
=> 30

irb(main):002:0> MatTopScorer.where(team_name: 'Toronto Maple Leafs').count
   (1.3ms)  SELECT COUNT(*) FROM "mat_top_scorers" WHERE "mat_top_scorers"."team_name"
= $1  [["team_name", "Toronto Maple Leafs"]]
=> 30
```

Refreshing a materialized view

As mentioned previously, materialized views cache the underlying query's result to a temporary table. This is what gives us the speed improvements and the ability to add indexes. The downside is that we have to control when the cache is refreshed. Modifying the `MatTopScorer` model, let's add a `refresh` method that can be called any time the data is to be refreshed. You will need to figure out how often it makes sense to update the data for your specific use-case, depending on how often the data is changing and how quickly those changes need to be reflected to the end user.

RUBY

```ruby
class MatTopScorer < ApplicationRecord
  belongs_to :player
  belongs_to :team
  belongs_to :match

  def self.refresh
    Scenic.database.refresh_materialized_view(table_name, concurrently: false, cascade: false)
  end

  def self.top_scorer_for_season(season)
    where(season: season).order(goal_count: :desc).first
  end

  def readonly?
    true
  end
end
```

To schedule the refresh, I like to use the whenever gem. Let's call a rake task to refresh the materialized view every hour:

RUBY

```ruby
# config/schedule.rb
every 1.hour do
  rake "refreshers:mat_top_scorers"
end
```

The rake task is simple, only calling the `refresh` method defined on the `MatTopScorer` model.

RUBY

```ruby
# lib/tasks/refreshers.rake
namespace :refreshers do
  desc “Refresh materialized view for top scorers”
  task mat_top_scorers: :environment do
    MatTopScorer.refresh
  end
end
```

Whenviews vs. materialized views?

Views focus on abstracting away complexity and encouraging reuse. Views allow you to interact with the **result** of a query as if it were a table itself, but they do not provide a performance benefit, as the underlying query is still executed, perfect for sharing logic but still having real-time access to the source data.

Materialized Views are related to views, but go a step further. You get all the abstraction and reuse of a view, but the underlying data is cached, providing serious performance benefits. Materialized views are especially useful for - for example - reporting dashboards because they can be indexed to allow for performant filtering.

If the purpose of the view is to provide a cleaner interface to complicated joins and query logic, and

performance isn't too much of an issue, by all means stick with a regular view. Views have the advantage of always being **real-time**, since they simply reference the real underlying data rather than a cached copy of it.

If your purpose is to provide a cleaner interface in addition to performance improvements, and you can live with the data being not quite real-time, then creating it as a materialized view can provide some great benefits.

Migrating views

It's easy to migrate views in Scenic. Views are versioned by default in Scenic and generating a view with the same name will create a v2, providing two files, just like it did the first time we generated a view (earlier in this eBook). Likewise, Scenic also provides a way to drop a view.

Testing with materialized views

Views and materialized views aren't particularly challenging to test, but it does require remembering that both types of views don't contain any original data in and of themselves, they are either a live

view of an underlying query, or a cached view of an underlying query, as in the case of materialized views.

Let's see how we would populate and then test our `MatTopScorer` model in RSpec and factory_bot.

After creating some test data using factory_bot, we'll call a method which is supposed to return the top scorer for a given season. It returns `nil`, and that is expected. The underlying data exists, but because materialized views must be refreshed, something we haven't done yet, there is no data to be found.

After calling `MatTopScorer.refresh`, we're now able to retrieve the expected result.

RUBY

```ruby
RSpec.describe MatTopScorer, type: :model do
  describe "#top_scorer_for_season" do
    it "finds top scorer" do
      # create some data using factory_bot helper methods
      match = create(:match)
      player = create(:player)
      goal = create(:goal, match: match, player: player)

      # without any data in materialized view
      expect(MatTopScorer.top_scorer_for_season(match.season)).to eq(nil)

      MatTopScorer.refresh

      # with data in materialized view
      top_scorer = MatTopScorer.top_scorer_for_season(match.season)
      expect(top_scorer).to be_present
      expect(top_scorer.player).to eq(player)
      expect(top_scorer.goal_count).to eq(1)
    end
  end
end
```

Conclusion

With the help of Scenic, using views and materialized views feels right at home in Rails. Truthfully, I haven't used views as much as I have used materialized views. In particular, I've found materialized views incredibly useful when building searchable reporting dashboards.

The ability to group and summarize data by geographic region, category, grouped by date, in combination with adding the correct indexes has provided an efficient way to report on large amounts of data without relying on external reporting systems or causing excessive load on the production database.

Creating Custom Postgres Data Types in Rails

Postgres ships with the most widely used common data types, like integers and text, built in, but it's also flexible enough to allow you to define your own data types if your project demands it.

Say you're saving price data and you want to ensure that it's never negative. You might create a `not_negative_int` type that you could then use to define columns on multiple tables. Or maybe you have data that makes more sense grouped together, like GPS coordinates. Postgres allows you to **create a type to hold that data together in one column** rather than spread it across multiple columns.

In Rails, all attributes pass through the attributes API when they're entered by the user or read from the database. Rails 5 introduced the Attributes API, allowing you to define your own attribute types and use them in your application.

In this section of the eBook, you'll learn how to **work with two of the most common custom types available in PostgreSQL**. You'll also see how to incorporate them into your Rails application using the Attributes API.

Custom Data Types in Postgres

There are two custom data types you'll learn about in this section:

Domain types: These allow you to put certain restrictions on a data type that can be reused later.

Composite types: These let you group data together to form a new type.

First, let's take a look at how to create a domain type. Say you want to ensure a username doesn't contain a `!`:

SQL

```
1   CREATE DOMAIN string_without_bang as VARCHAR NOT NULL CHECK (value !~ '!');
```

After that, you can use this domain type when you create our users table:

SQL

```
1   CREATE TABLE users (
2    id serial primary key,
3    user_name string_without_bang
4   );
```

Let's try creating a user with a username that contains an exclamation point. You'll see an error message:

SQL

```
1   INSERT INTO users(user_name) VALUES ('coolguy!!');
2   -- ERROR:  value for domain string_without_bang violates check constraint "string_
    without_bang_check"
```

You can even use a domain in the definition of another domain:

SQL

```
1   CREATE DOMAIN email_with_check AS string_without_bang NOT NULL CHECK (value ~ '@');
2
3   CREATE TABLE email_addresses (
4     user_id integer,
5     email email_with_check
6   );
7
8   INSERT INTO email_addresses(email) VALUES ('frank!@gmail.com');
9   -- ERROR:  value for domain email_with_check violates check constraint "string_
    without_bang_check"
10
11  INSERT INTO email_addresses(email) VALUES ('joshgmail.com');
12  -- ERROR:  value for domain email_with_check violates check constraint "email_with_
    check_check"
```

Composite Types

Composite types allow you to group different pieces of data together into one column. They're useful for information that has more meaning when grouped together, like RGB color values or the dimensions of a package.

Let's start by creating a dimensions type:

SQL

```
CREATE TYPE dimensions as (
  depth integer,
  width integer,
  height integer
);
```

Next, let's create a table using this new type. Try also using the domain type you created previously:

SQL

```
CREATE TABLE orders (
  product string_without_bang,
  dims dimensions
);
```

Add some data and take a look at the output when you query the table:

SQL

```
INSERT INTO orders(product, dims) VALUES('widget', (50,88,101));

SELECT * FROM orders;

 product |    dims
---------+------------
 widget  | (50,88,101)
```

You'll see that all the data related to the dimensions of the package is **saved together in the dims**

column. But don't worry, you'll still be able to access the individual ints.

SQL

```
1   SELECT (dims).width FROM orders;
2
3    width
4   -------
5     88
```

Custom Types in Rails

In order to use our custom types in Rails, **you'll have to do two things**:

- Create the migration that sets the types up for us in the database.
- Tell Rails how to handle your new type so you can easily work with it in Ruby.

Currently, Rails doesn't offer any built-in solution for creating types in migrations, so you'll have to run some raw SQL. The code below runs exactly what you ran above to create the type directly in PostgreSQL, then immediately uses the types to build the orders table:

RUBY

```ruby
class CreateOrders < ActiveRecord::Migration[6.1]
  def up
    execute <<~SQL
      CREATE TYPE dimensions as (
        depth integer,
        width integer,
        height integer
      );
      CREATE DOMAIN string_without_bang as VARCHAR NOT NULL CHECK (value !~ '!');
    SQL
    create_table :orders do |t|
      t.column :product, :string_without_bang
      t.column :dims, :dimensions
    end
  end

  def down
    drop_table :orders
    execute "DROP TYPE dimensions"
    execute "DROP DOMAIN string_without_bang"
  end
end
```

You'll need to use up and down methods here since you're running raw SQL that Rails won't be able to easily undo on its own if you want to do a rollback.

Run the migrations, and you'll see output that looks similar to this:

BASH

```
== 20210211230550 Orders: migrating ===========================================
-- execute("CREATE TYPE dimensions as (\n  depth integer,\n  width integer,\n  height integer\n);\nCREATE DOMAIN string_without_bang as VARCHAR NOT NULL CHECK (value !~ '!');\n")
   -> 0.0012s
-- create_table(:Orders)
   -> 0.0058s
== 20210211230550 Orders: migrated (0.0071s) ==================================

unknown OID 25279: failed to recognize type of 'dims'. It will be treated as String.
```

Notice that the migration succeeded, but **Rails does not know what to do with the composite type**, so it will treat it as a string. If you check the database directly, you'll see that the type for dims column is what you expect:

BASH

```
=# \d orders
 Column  |        Type         |
---------+---------------------+
 id      | bigint              |
 product | string_without_bang |
 dims    | dimensions          |
```

Right now, if you create a new product, you'll need to enter the dims data as a properly formatted string like this:

RUBY

```
2.6.3 :001 > o = Order.create product: 'hat', dims: '(1,2,3)'
2.6.3 :002 > o.dims
 => "(1,2,3)"
2.6.3 :003 >
```

This setup doesn't allow you to update the individual elements without having to completely override the entire string. What's needed here is a dimensions class that has methods that know how to deal with this new data type. Luckily, **Rails has a solution for this.**

Using the Active Record Attributes API to Register new Custom Types in Rails

You can use the Active Record Attributes API to register the new type and control what it looks like when leaving and entering the database.

Start by creating a dimensions class that takes in a string in the initialize method. Data will come in from the database as a string with parentheses `"(1,2,3)"`, so you'll need to parse it and then set some instance variables. Notice the code also includes a `to_s` method that returns the data back to the string with parentheses that the database will understand.

RUBY

```ruby
class Dimension
  attr_accessor :depth, :width, :height

  def initialize(values)
    dims = values ? sanitize_string(values) : [0,0,0]
    @depth = dims[0]
    @width = dims[1]
    @height = dims[2]
  end

  def sanitize_string(values)
    values.delete("()").split(',').map(&:to_i)
  end

  def to_s
    "(#{depth},#{width},#{height})"
  end
end
```

This class will act as a **wrapper** for the dimension type and will make it easier to work with, but you still need to tell Rails how to handle it when saving to the database and instantiating your order objects. Just like Rails knows how to take a Ruby string type or a Ruby int type and pass it off to PostgreSQL in a way it can save and understand, you need to tell Rails how to handle your new dimension type. You can do that by creating a `DimensionType` that inherits from `ActiveRecord::Type::Value` and setting up a few methods.

RUBY

```ruby
class DimensionType < ActiveRecord::Type::Value
  def cast(value)
    Dimension.new(value)
  end

  def serialize(value)
    value.to_s
  end

  def changed_in_place?(raw_old_value, new_value)
    raw_old_value != serialize(new_value)
  end
end
```

The `#cast` method gets called by Active Record when setting an attribute in the model. You can use your new dimension class for this.

The `#serialize` method converts your dimension object to a type that PostgreSQL can understand. This is why you set up your `to_s` method in your `Dimension` class.

Finally, `#changed_in_place?` takes care of comparing the raw value in the database with your new value. This is what gets called whenever Active Record tries to decide if it needs to make an update to the database. `raw_old_value` will always be a string because it's read directly from the database. `new_value`, in this case, will be an instance of `Dimension`, so it needs to be converted to a string in order to make the comparison.

The last piece of the puzzle will be to tell the order model to use the new `DimensionType` for the `dims` attribute.

RUBY

```ruby
class Order < ApplicationRecord
  attribute :dims, DimensionType.new
end
```

Let's open a Rails console and test out the new type:

RUBY

```
2.6.3 :001 > o = Order.new
 => #<Order id: nil, product: nil, dims: #<Dimension:0x00007fda47295e40 @depth="0", @width="0", @height="0">>
2.6.3 :002 > o.product = 'a wig'
 => "a wig"
2.6.3 :003 > o.dims.width = 9
 => 9
2.6.3 :004 > o.dims.depth = 4
 => 4
2.6.3 :005 > o.dims.height = 1
 => 1
2.6.3 :006 > o.save
D, [2021-02-22T09:55:41.588716 #79057] DEBUG -- :    (0.2ms)  BEGIN
D, [2021-02-22T09:55:41.607391 #79057] DEBUG -- :   Order Create (0.8ms)  INSERT INTO "orders" ("product", "dims") VALUES ($1, $2) RETURNING "id"  [["product", "a wig"], ["dims", "(4,9,1)"]]
D, [2021-02-22T09:55:41.610457 #79057] DEBUG -- :    (1.2ms)  COMMIT
 => true
2.6.3 :007 >
```

See if you can follow the same process to set up the `string_without_bang`!

Conclusion

In this last section, we showed how to create two different unique data types in PostgreSQL.

The first, a **domain type**, allows you to create checks on your data and reuse those checks on multiple columns. The second, the **composite type**, lets you group data in a meaningful way for storage in a single column. Finally, we learned how to hook into the Rails Attributes API to help instantiate a new type as an object that Ruby knows how to use.

Try pganalyze for free

Have performance issues with your database, and looking for a way to improve indexes and query plans?

pganalyze provides deep insights for your database, with detailed performance analysis, and automated EXPLAIN plans, to quickly understand what is slow with your database. pganalyze integrates directly with major cloud providers, as well as self-managed Postgres installations.

Get started easily with a free 14-day trial, or learn more about our Enterprise product.

If you want, you can also request a personal demo.

"Our overall usage of Postgres is growing, as is the amount of data we're storing and the number of users that interact with our products. pganalyze is essential to making our Postgres databases run faster, and makes sure end-users have the best experience possible."

Robin Fernandes, Software Development Manager
Atlassian

Interested in learning more about Postgres? Check out our Resources Library.

pganalyze resources library

Efficient Search in Rails with Postgres eBook

Best Practices for Optimizing Postgres Query Performance

PGANALYZE

About pganalyze.

DBAs and developers use pganalyze to identify the root cause of performance issues, optimize queries and to get alerts about critical issues.

Our rich feature set lets you optimize your database performance, discover root causes for critical issues, get alerted about problems before they become big, gives you answers and lets you plan ahead.

Hundreds of companies monitor their production PostgreSQL databases with pganalyze.

Be one of them.
Sign up for a free trial today!