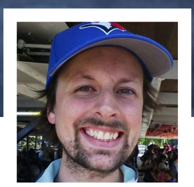


LEIGH HALLIDAY

Efficient Search in Rails with Postgres

Efficient Search in Rails with Postgres

Allowing your users to find the data they are looking for is very important. For example, it can mean the difference between a sale or no sale, in the case of e-commerce applications.

The type of data you have may help determine the best way to find it. Or, imagine another situation, where we are developing a stock market research website where the user can search for a company using a single text input.

In this eBook we show how we speed up a search query from seconds to milliseconds. We walk through using exact matches, similarity matches with trigrams, partial matches with ILIKE, and natural language matches.

ABOUT

They may enter:

- **AAPL:** The stock market symbol for the company.
- **Apple:** The (partial) name of the company.
- **Integrated hardware and software:** When they don't know exactly what they're looking for, and want to find the best match using natural language.

In this ebook, we will cover how to efficiently find the correct data for our user for all three scenarios mentioned above using only Ruby on Rails and Postgres. We will see when you might choose one over the other along with the pros and cons of each approach, aiming to optimize our queries along the way.

```
| Postgres Feature     | Typical Use Case                                     | Can Be Indexed?    | Performance    |
| ---------------------| -----------------------------------------------------| -------------------| ---------------|
| Exact \(=\)          | Exact match or user selecting from predefined list | Yes \(BTREE\)      | Great          |
| LIKE/ILIKE           | Wildcard\-style search for small data               | Sometimes          | Unpredictable |
| Trigram \(pg_trgm\) | Similarity search for names, etc                    | Yes \(GIN, GiST\) | Good           |
| Full Text Search     | Natural language search                              | Yes \(GIN, GiST\) | Good           |
```

What Data Are We Using?

The data used in the following examples consists of 253k records in a table called `companies`. There are about 3k companies listed on the NASDAQ Stock Exchange, 700 from the TSX (Toronto Stock Exchange), and then an additional 250k records consisting of fake data generated using Faker.

Our goal is for all three of the user's searches to return this record:

JSON

```json
{
  "id": 256651,
  "exchange": "NASDAQ",
  "symbol": "AAPL",
  "name": "Apple Inc.",
  "description": "Apple designs a wide variety of consumer electronic devices, including smartphones (iPhone), tablets (iPad), PCs (Mac), smartwatches (Apple Watch), and TV boxes (Apple TV), among others. The iPhone makes up the majority of Apple's total revenue. In addition, Apple offers its customers a variety of services such as Apple Music, iCloud, Apple Care, Apple TV+, Apple Arcade, Apple Card, and Apple Pay, among others. Apple's products run internally developed software and semiconductors, and the firm is well known for its integration of hardware, software and services."
}
```

If you would like to follow along, the source code is available, and by running `rails db:seed` it will generate the 253k records mentioned above.

Exact Matches with Equals

Exact matches are perfect for when the user knows exactly what they are searching for (a stock symbol, an email address or username, a referral code), or when they are choosing from a predefined list of options. This is the most restrictive approach to searching, but it is easy to optimize using a standard B-tree index in Postgres. No additional Ruby gems or Postgres extensions are required for it to work.

Pros:

- It is simple: Works out of the box
- It is performant: A standard B-tree index does wonders

Cons:

- It must be an exact match
- It is case sensitive

The easiest way to find a single matching company record in Rails is by using the `find_by` method.

RUBY

```ruby
Company.find_by(symbol: 'AAPL')
```

The above Ruby code will produce the following SQL:

SQL

```sql
SELECT “companies”.* FROM “companies” WHERE “companies”.”symbol” = ‘AAPL’ LIMIT 1
```

Optimizing an Exact Match Query

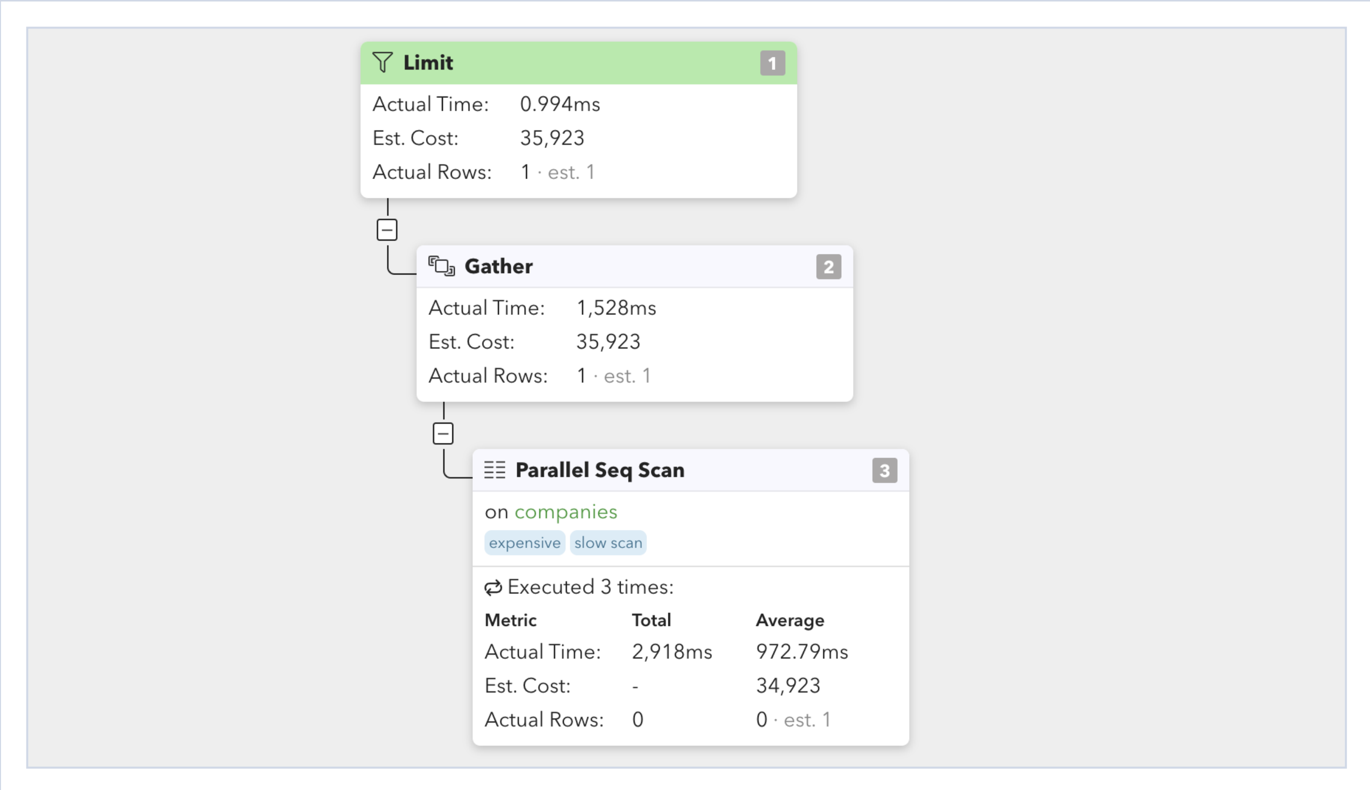
Because we knew that we would be searching for companies by their symbol, we created an index on this column at the same time that the table was created.

RUBY

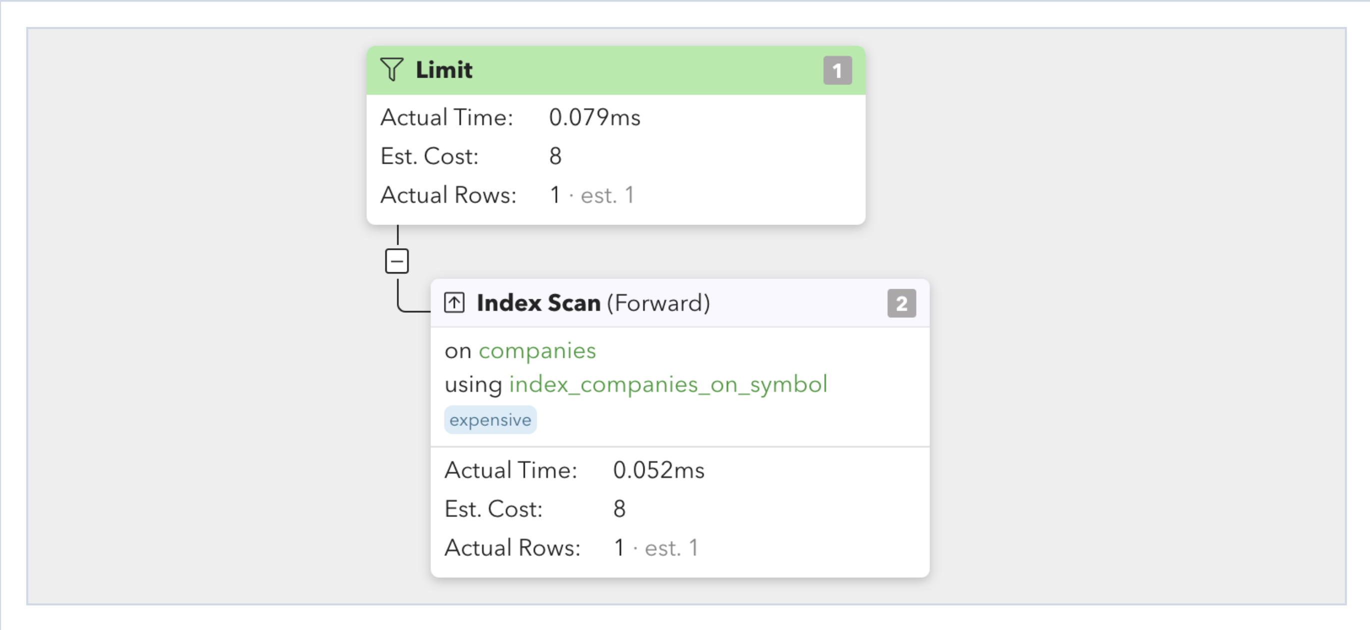
```ruby
class CreateCompanies < ActiveRecord::Migration[6.0]
  def change
    create_table :companies do |t|
      t.string :exchange, null: false
      t.string :symbol, null: false
      t.string :name, null: false
      t.text :description, null: false
      t.timestamps

      t.index :symbol
      t.index %i[exchange symbol], unique: true
    end
  end
end
```

By adding an index, it takes a query that would be close to 1500ms without an index, to just 7ms. That's an improvement of **over 200x!**

Before (Sequential Scan):



After (Index Scan):



Partial / Case Insensitive Matches with ILIKE

Exact matches can be extremely restrictive because they only help in a limited set of search use cases. An alternative is to provide wildcard (partial) searching using LIKE and its case insensitive partner `ILIKE`.

Pros:

- The user doesn't have to worry about the case matching
- The user can enter partial matches
- No additional Ruby gems required

Cons:

- Index usage is unpredictable
- Spelling must be accurate (Applo would not find Apple

To find Apple from our companies table, the following two queries will both do the trick. The first will simply use `ILIKE` to allow for case insensitive searching, while the second query will use the `%` wildcard before and after the term, finding any company that has **apple** in its name.

RUBY

```ruby
Company.where('companies.name ILIKE ?', 'apple inc.').take
Company.where('companies.name ILIKE ?', '%apple%').take
```

SQL

```sql
-- case insensitive
SELECT companies.*
  FROM companies
 WHERE companies.name ILIKE 'apple inc.'
 LIMIT 1;

-- wildcard search
SELECT companies.* FROM companies WHERE companies.name ILIKE '%apple%' LIMIT 1;
```

Using Indexes with LIKE and ILIKE

The pg_trgm extension allows `GIN` and `GiST` indexes on text columns, which can be used to speed up `LIKE` and `ILIKE` searches. That said, even after indexing the `companies.name` column this way, the queries in our example still don't use this index and instead continue doing a full table sequence scan, providing no performance increase. This is due to the structure of the associated trigrams, and the Postgres planner determining a sequential scan is still more effective.

GIN/GIST indexes together with pg_tgrm can sometimes be used for `LIKE` and `ILIKE`, but query performance is unpredictable when user-generated input is presented.

To add a `GIN` trigram index to the name column, we will first need to enable the `pg_trgm` extension. Note the pg_trgm extension ships with a standard Postgres installation and is available with major cloud providers, making it safe to rely on it:

RUBY

```ruby
class EnableTrigramExtension < ActiveRecord::Migration[6.0]
  def change
    enable_extension :pg_trgm
  end
end
```

With the extension enabled, we can add an index to our `name` column:

RUBY

```ruby
class AddTrigramIndexCompaniesName < ActiveRecord::Migration[6.0]
  disable_ddl_transaction!

  def change
    add_index :companies,
              :name,
              opclass: :gin_trgm_ops,
              using: :gin,
              algorithm: :concurrently,
              name: 'index_companies_on_name_trgm'
  end
end
```

On the plus side, this is the **very same** extension and index we will be using in the following section on similarity matches, so all is not lost!

Similarity Matches with Trigrams

You searched **"Applo"**... did you mean **"Apple"**?

Trigrams allow us to find matches based on similarity. This means that even if the user misspells a company name, as long as their spelling is similar to the real one, we'll be able to find the correct match.

Pros:

- Misspellings are no problemmatching
- Searches are case insensitive
- Searches can be indexed

Cons:

- Does not replace the need for natural language search

What is a Trigram?

Trigrams involve breaking a string into groups of three consecutive letters. For the word `Apple`, it would be broken into **three** groups: app, ppl, ple.

This process of breaking a piece of text into smaller groups allows you to compare the groups of one word to the groups of another word. Knowing how many groups are shared between the two words allows you to make a comparison between them, based on how similar their groups are.

Let's see an example of this functionality in SQL:

SQL

```
SELECT
  show_trgm(‘Apple’), -- {“  a”,” ap”,app,”le “,ple,ppl}
  show_trgm(‘Applo’), -- {“  a”,” ap”,app,”lo “,plo,ppl}
  similarity(‘Apple’, ‘Apple’), -- 1
  similarity(‘Applo’, ‘Apple’), -- 0.5
  similarity(‘Opple’, ‘Apple’),  -- 0.33
  ‘Applo’ % ‘Apple’ -- TRUE
```

The `show_trgm` function takes a string and returns us a list of trigrams. You'll notice something interesting here, that there are more than three results... there are actually six! Two spaces were added to the beginning of the string, and a single space was added to the end.

The first reason is that it allows similarity calculations on words with less than three characters, such as `Hi`.

Secondly, it ensures the first and last characters are not overly de-emphasized for comparisons. If we used only strict triplets, the first and last letters in longer words would each occur in only a single group: with padding they occur in three (for the first letter) and two (for the last). The last letter is less important for matching, which means that `Apple` and `Applo` are more similar than `Apple` and `Opple`, even though they are both off by a single character.

The `similarity` function provides us a percentage of shared trigrams between zero and one. Zero meaning that they aren't similar at all, one meaning they are a perfect match.

Lastly, we have the `%` operator, which gives you a boolean of whether two strings are similar. By default, Postgres uses the number 0.3 when making this decision, but you can always update this setting.

Using Trigrams in Rails

The `pg_trgm` extension must be enabled to use Trigram functionality. We have already seen this when adding an index to improve `ILIKE` search, but if you skipped forward to this section:

RUBY

```ruby
class EnableTrigramExtension < ActiveRecord::Migration[6.0]
  def change
    enable_extension :pg_trgm
  end
end
```

Typically we not only care whether a match is considered similar or not, but also **how** similar it is. We want more similar results before less similar ones. To accomplish this, let's create a `scope` that will not only filter results using the `%` (similarity) operator, but will also sort them by how similar they are, from most to least (descending order).

RUBY

```ruby
class Company < ApplicationRecord
  scope :name_similar,
        lambda { |name|
          quoted_name = ActiveRecord::Base.connection.quote_string(name)
          where('companies.name % :name', name: name).order(
            Arel.sql("similarity(companies.name, '#{quoted_name}') DESC")
          )
        }
end

Company.name_similar('Applo').first
```

The SQL produced with the above Ruby code looks like:

SQL

```sql
  SELECT companies.*
    FROM companies
   WHERE companies.name % 'Applo'
ORDER BY similarity(companies.name, 'Applo') DESC
   LIMIT 1;
```

Optimizing Trigram Search

Trigram (similarity) search can be optimized by adding either a GIN or GiST index to your column, using a special `opclass` of `gin_trgm_ops`. We will be working with GIN indexes because Postgres recommends them when working with text:

> GIN indexes are the preferred text search index type. As inverted indexes, they contain an index entry for each word (lexeme), with a compressed list of matching locations. Multi-word searches can find the first match, then use the index to remove rows that are lacking additional words. GIN indexes store only the words (lexemes) of tsvector values, and not their weight labels. Thus a table row recheck is needed when using a query that involves weights.

GIN INDEXES

In Rails, the type of index and opclass can be set using options on the `add_index` method. Note that we also included `disable_ddl_transaction!` at the top of this migration, and set the algorithm to `concurrently` so that this index will be added concurrently. This is useful (or crucial) when you have large amounts of rows and want to avoid locking the table while the index is being applied.

RUBY

```ruby
class AddTrigramIndexCompaniesName < ActiveRecord::Migration[6.0]
  disable_ddl_transaction!

  def change
    add_index :companies,
              :name,
              opclass: :gin_trgm_ops,
              using: :gin,
              algorithm: :concurrently,
              name: 'index_companies_on_name_trgm'
  end
end
```

Natural Language Matches with Full Text Search

Full Text Search allows a user to search through large documents by simply describing in English words what they want. In our example, searching „integrated hardware and software“ allows us to find Apple based on their description, even though those words don't show up in exactly the same form or order.

Pros:

- Users can use natural language
- Easily search through large documents
- Searches are very performant

Cons:

- Can be difficult to configure

Minimal Full Text Search in Rails Setup

We will be using a gem called pg_search to make Postgres' Full Text Search easier to integrate into our Rails application.

After installing the `pg_search` gem, it's extremely easy to add Full Text Search to your Rails model:

RUBY

```ruby
class Company < ApplicationRecord
  include PgSearch::Model
  pg_search_scope :search, against: :description
end
```

Then we can call `search` and search through the companies' description column using a natural language search.

RUBY

```ruby
Company.search('integrated hardware and software').first
```

Unfortunately, there are a **number of issues** with the simple setup above. It is slow (15 seconds), it isn't aware that we are searching in the English language, and what if we wanted to search on both the `name` and the `description` columns, giving precedence to the `name`? We will learn how to solve all of these issues below.

Interested in learning more about searching in Rails with Postgres? Check out our blog posts!

Similarity in Postgres and Rails using Trigrams (click to read article)

Full Text Search in Milliseconds with Rails (click to read)

PGANALYZE

The Foundations of Full Text Search

By taking time to understand the underlying concepts of Postgres' Full Text Search, we can better solve the issues that have arisen in our simple search configuration.

SQL

```sql
SELECT
  to_tsvector('english', 'the firm is well known for its integration of hardware,
software and services');
-- 'firm':2 'hardwar':10 'integr':8 'known':5 'servic':13 'softwar':11 'well':4
```

In the above SQL we have some text (taken from the Apple company description); often referred to as a `document` when talking about Full Text Search. A document must be parsed and converted into a special data type called a `tsvector`, which we did using the function `to_tsvector`.

The `tsvector` data type is comprised of lexemes. Lexemes are normalized key words which were contained in the document that will be used when searching through it. In this case we used the English language dictionary to normalize the words, breaking them down to their root. This means that

„integration“ became „integr“, „services“ became „servic“, with very common words being removed completely, to avoid false positives.

SQL

```
SELECT to_tsvector(
        ‘english’,
        ‘the firm is well known for its integration of hardware, software and services’
       )
       @@ to_tsquery(‘english’, ‘integrated & hardware & and & software’), -- TRUE
       to_tsquery(‘english’, ‘integrated & hardware & and & software’); -- ‘integr’ & ‘hardwar’ & ‘softwar’
```

The `@@` operator allows us to check if a query (data type `tsquery`) exists within a document (data type `tsvector`). Much like `tsvector`, `tsquery` is also normalized prior to searching the document for matches. Although pg_search handles this for us, search terms passed to `to_tsquery` must be separated by `&`.

SQL

```
SELECT ts_rank(
        to_tsvector(
            ‘english’,
            ‘the firm is well known for its integration of hardware, software and services’
        ),
        to_tsquery(‘english’, ‘integrated & hardware & and & software’)
       );
-- 0.2669
```

The `ts_rank` function takes a `tsvector` and a `tsquery`, returning a number that can be used when sorting the matching records, allowing us to sort the results from highest to lowest ranking.

Searching a Single Column

Combining these concepts together, we can find companies with a matching query string:

SQL

```sql
   SELECT id,
          exchange,
          symbol,
          name,
          ts_rank(
             to_tsvector('english', description),
             to_tsquery('english', 'integrated & hardware & and & software')
          ) AS rank
     FROM companies
    WHERE to_tsvector('english', description)
          @@ to_tsquery('english', 'integrated & hardware & and & software')
 ORDER BY rank DESC
    LIMIT 2;
```

This will return Apple as the number one rank, followed by a company called Vecima Networks Inc. Interestingly, if the language `english` isn't specified (defaulting to a dictionary called `simple`), these companies will come back in the opposite order.

JSON

```
[
  {
    "id": 9,
    "exchange": "NASDAQ",
    "symbol": "AAPL",
    "name": "Apple Inc.",
    "rank": 0.29986802
  },
  {
    "id": 3640,
    "exchange": "TSX",
    "symbol": "VCM",
    "name": "Vecima Networks Inc.",
    "rank": 0.2642635
  }
]
```

Searching Multiple Columns

Up until this point we have only seen a single column being searched at a time, but what if we want our Full Text Search to look at both the `name` **AND** the `description` columns? And what if we wanted to give precedence to the `name` column?

Data types of `tsvector` can be concatenated with the `||` operator, and precedence (weight) can be given to a certain column using the `setweight` function. Valid weighting options are: A, B, C or D.

SQL

```sql
  SELECT id,
         exchange,
         symbol,
         name,
         ts_rank(
             setweight(to_tsvector(‘english’, name), ‘A’)
             || setweight(to_tsvector(‘english’, description), ‘B’),
             to_tsquery(‘english’, ‘integrated & hardware & and & software’)
         ) AS rank
    FROM companies
   WHERE setweight(to_tsvector(‘english’, name), ‘A’)
         || setweight(to_tsvector(‘english’, description), ‘B’)
             @@ to_tsquery(‘english’, ‘integrated & hardware & and & software’)
ORDER BY rank DESC
   LIMIT 2;
```

Configuring pg_search

With our knowledge of `tsvector` concatenation, using `setweight` to specify column precedence, and setting `english` as the language of our choice, let’s configure `pg_search` to build this query for us.

RUBY

```ruby
class Company < ApplicationRecord
  include PgSearch::Model

  pg_search_scope :search,
                  against: { name: ‘A’, description: ‘B’ },
                  using: { tsearch: { dictionary: ‘english’ } }
end
```

Which can be queried with:

RUBY

```ruby
Company.search('integrated hardware and software').first
```

Optimizing Full Text Search

We still have a major issue on our hands! The query takes **15 seconds** to complete, **much** too long for it to be used in a production system. Why does it take so long?

It is slow for two reasons:

- The tsvectors are being calculated for the entirety of our data on every query
- No index is being used

As we are venturing into the territory of more custom Postgres functionality, not easily supported by the Rails schema file in Ruby, we'll want to switch the schema format from `:ruby` to `:sql`. This line can be added to the application.rb file:

Which can be queried with:

RUBY

```ruby
config.active_record.schema_format = :sql
```

To solve the first problem, we can create a migration to add a `generated` and `stored` column that will store the resulting `tsvector` value. Note that generated columns are only supported with Postgres 12 and newer, but there are alternatives for older Postgres versions. This generated column actually occupies space in our table and gets written on each INSERT/UPDATE. This also means that when we add a generated column to a table, it will require a rewrite of the table to actually set the values for all existing rows. This may take some time for a large table and block other operations on the database.

RUBY

```ruby
class AddSearchableToCompanies < ActiveRecord::Migration[6.0]
  def up
    execute <<-SQL
      ALTER TABLE companies
      ADD COLUMN searchable tsvector GENERATED ALWAYS AS (
        setweight(to_tsvector('english', coalesce(name, '')), 'A') ||
        setweight(to_tsvector('english', coalesce(description,'')), 'B')
      ) STORED;
    SQL
  end

  def down
    remove_column :companies, :searchable
  end
end
```

Now let's add an index to this new `searchable` column of type `GIN`, doing it concurrently to avoid table locking:

RUBY

```ruby
class AddIndexToCompaniesS                         ::Migration[6.0]
  disable_ddl_transaction!

  def change
    add_index :companies, :searchable, using: :gin, algorithm: :concurrently
  end
end
```

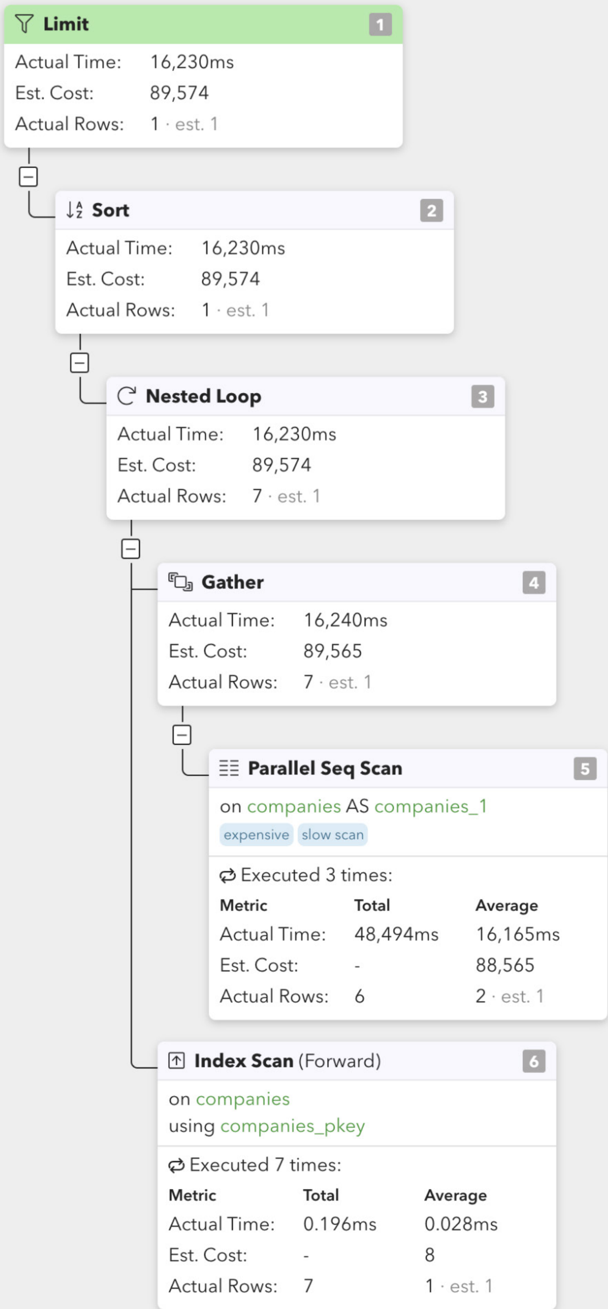
With our generated, stored column that is GIN indexed in place, we have to adjust the pg_search configuration to take advantage of the tsvector column:

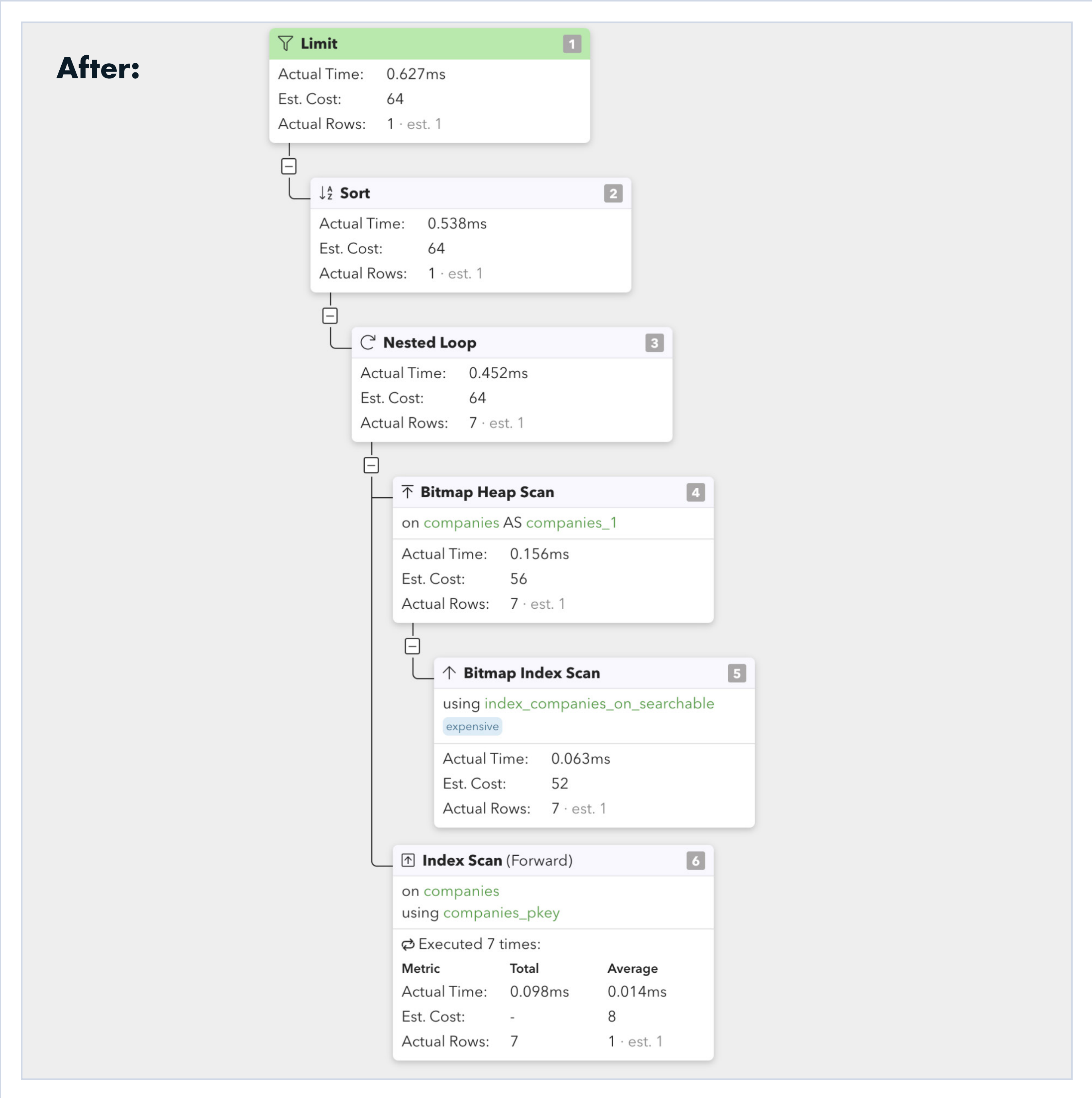
RUBY

```ruby
class Company < ApplicationRecord
  include PgSearch::Model

  pg_search_scope :search,
                  against: { name: 'A', description: 'B' },
                  using: {
                    tsearch: {
                      dictionary: 'english', tsvector_column: 'searchable'
                    }
                  }
end
```

To finish, let's look at the results. When we compare the EXPLAIN plans, we can see that we've gone from **16 seconds** to under **1 ms**, approximately **16000 times faster**. Talk about a speed improvement!

Before:




After:
Limit 1
Actual Time: 0.627ms
Est. Cost: 64
Actual Rows: 1 · est. 1
Sort 2
Actual Time: 0.538ms
Est. Cost: 64
Actual Rows: 1 · est. 1
Nested Loop 3
Actual Time: 0.452ms
Est. Cost: 64
Actual Rows: 7 · est. 1
Bitmap Heap Scan 4
on companies AS companies_1
Actual Time: 0.156ms
Est. Cost: 56
Actual Rows: 7 · est. 1
Bitmap Index Scan 5
using index_companies_on_searchable
expensive
Actual Time: 0.063ms
Est. Cost: 52
Actual Rows: 7 · est. 1
Index Scan (Forward) 6
on companies
using companies_pkey
Executed 7 times:
Metric Total Average
Actual Time: 0.098ms 0.014ms
Est. Cost: - 8
Actual Rows: 7 1 · est. 1

Conclusion

With Postgres and Rails, we have the tools to get search right. Starting with exact matches, going to similarity searching using Trigrams, and ending up with Full Text Search allowing for natural language querying. For each of these options we have ways to optimize our queries by adding the appropriate indexes, giving us accurate and performant search results.

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

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!