Extracted from:

Rails, Angular, Postgres, and Bootstrap, Second Edition

Powerful, Effective, Efficient, Full-Stack Web Development

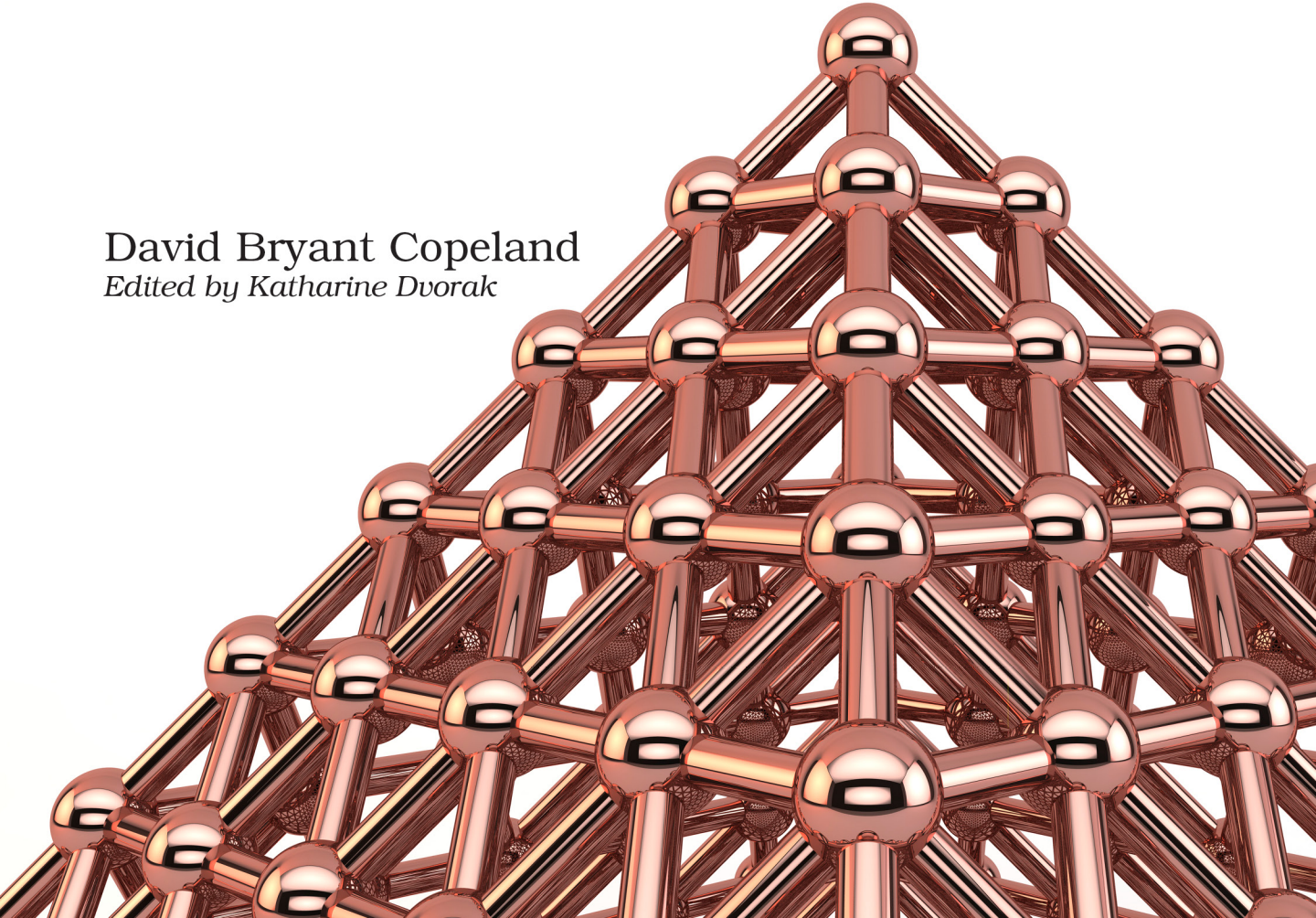
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Rails, Angular, Postgres, and Bootstrap
Second Edition

Powerful, Effective, Efficient, Full-Stack Web Development



David Bryant Copeland
Edited by Katharine Dvorak

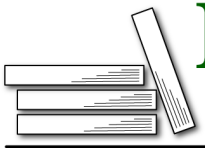
Rails, Angular, Postgres, and Bootstrap, Second Edition

Powerful, Effective, Efficient, Full-Stack Web Development

David Bryant Copeland

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Liz Welch
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-220-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2017

Understanding Query Performance with the Query Plan

If you aren't familiar with database indexes, Wikipedia has a pretty good definition,[6] but in essence, an index is a data structure created inside the database that speeds up query operations. Usually, databases use advanced data structures like B-trees to find the data you're looking for without examining every single row in a table.

If you *are* familiar with indexes, you might only be familiar with the type of indexes that can be created by Active Record's Migrations API. This API provides a "lowest common denominator" approach. The best we can do is create an index on last_name, first_name, and email. Doing so won't actually help us because of the search we are doing. We need to match values that *start* with the search term and ignore case.

Postgres allows much more sophisticated indexes to be created. To see how this helps, let's ask Postgres to tell us how our existing query will perform. This can be done by preceding a SQL statement with EXPLAIN ANALYZE. The output is somewhat opaque, but it's useful. We'll walk through it step by step.

```
$ bundle exec rails dbconsole
shine_development> EXPLAIN ANALYZE
                     SELECT *
                     FROM   customers
                     WHERE
                       lower(first_name) like 'pat%' OR
                       lower(last_name)  like 'pat%' OR
                       lower(email)      = 'pat@example.com'
                     ORDER BY
                       email = 'pat@example.com' DESC,
                       last_name ASC ;

                      QUERY PLAN
-------------------------------------------------------------------------------
1 Sort  (cost=13930.19..13943.25 rows=5225 width=79)
       (actual time=618.065..618.103 rows=704 loops=1)
   Sort Key: (((email)::text = 'pat@example.com'::text)) DESC, last_name
   Sort Method: quicksort  Memory: 124kB
2    ->  Seq Scan on customers  (cost=0.00..13607.51 rows=5225 width=79) #
                                (actual time=0.165..612.380 rows=704 loops=1)
3         Filter: ((lower((first_name)::text) ~~ 'pat%'::text) OR
                   (lower((last_name)::text) ~~ 'pat%'::text)  OR
                   (lower((email)::text) = 'pat@example.com'::text))
         Rows Removed by Filter: 349296
 Planning time: 1.223 ms
4 Execution time: 618.258 ms
```

6. http://en.wikipedia.org/wiki/Database_index

This gobbledegook is the *query plan* and is quite informative if you know how to interpret it. There are four parts to it that will help you understand how Postgres will execute our query.

❶ Here, Postgres is telling us that it's sorting the results, which makes sense since we're using an `order by` clause in our query. The details (for example, `cost=15479.51`) are useful for fine-tuning queries, but we're not concerned with that right now. Just take from this that sorting is part of the query.

❷ This is the most important bit of information in *this* query plan. "Seq Scan on customers" means that Postgres has to examine every single row in the table to satisfy the query. This means that the bigger the table is, the more work Postgres has to do to search it. Queries that you run frequently should not require examining every row in the table for this reason.

❸ This shows us how Postgres has interpreted our `where` clause. It's more or less what was in our query, but Postgres has annotated it with the internal data types it's using to interpret the values.

❹ Finally, Postgres estimates the runtime of the query. In this case, it's more than half a second. That's not much time to you or me, but to a database, it's an eternity.

EXPLAIN vs. EXPLAIN ANALYZE

`EXPLAIN ANALYZE` actually runs the query, whereas `EXPLAIN` (without `ANALYZE`) will just show the query plan. This means that repeatedly executing `EXPLAIN ANALYZE` on the same query could produce different timings, because Postgres could cache the query's results. The query plan (everything up to "Execution time" in the output shown earlier) will always be the same. It's not easy to control the caches Postgres uses, but if you vary the search string or ID in your `WHERE` clauses, you can often prevent it from using the cache.

Given all of this, it's clear that our query will perform poorly. It's likely that it performs poorly on our development machine, and will certainly not scale in a real-world scenario.

In most databases, because of the case-insensitive search and the use of `like`, there wouldn't be much we could do. Postgres, however, can create an index that accounts for this way of searching.

Indexing Derived and Partial Values

Postgres allows you to create an index on *transformed* values of a column. This means you can create an index on the lowercased value for each of our

three fields. Further, you can configure the index in a way that allows Postgres to optimize for the "starts with" search you are doing. Here's the basic syntax:

```
CREATE INDEX
  customers_lower_last_name
ON
  customers (lower(last_name) varchar_pattern_ops);
```

If you're familiar with creating indexes, the varchar_pattern_ops might look odd. This is a feature of Postgres called *operator classes*. Specifying an operator class isn't required; however, the default operator class used by Postgres will only optimize the index for an exact match. Because you're using a like in your search, you need to use the nonstandard operator class varchar_pattern_ops. You can read more about operator classes in Postgres's documentation.[7]

Now that you've seen the SQL needed to create these indexes, you need to adapt them to a Rails migration. Previous versions of Rails didn't provide a way to do this, and you'd have to use execute to directly execute SQL, but as of Rails 5, we can pass custom SQL to add_index, making our migration a bit cleaner. Let's create the migration file using Rails's generator.

```
$ bundle exec rails g migration add-lower-indexes-to-customers
    invoke  active_record
    create    db/migrate/20160721030725_add_lower_indexes_to_customers.rb
```

Next, edit the migration to add the indexes. Rails 5 added the ability to create these Postgres-specific indexes using add_index. Previous versions of Rails required using execute and typing the CREATE INDEX SQL directly.

4_postgres-index/40-add-indexes/shine/db/migrate/20160721030725_add_lower_indexes_to_customers.rb

```
class AddLowerIndexesToCustomers < ActiveRecord::Migration[5.0]
  def change
    add_index :customers, "lower(last_name) varchar_pattern_ops"
    add_index :customers, "lower(first_name) varchar_pattern_ops"
    add_index :customers, "lower(email)"
  end
end
```

Note that we aren't using the operator class on the email index since we'll always be doing an exact match. Sticking with the default operator class is recommended if we don't have a reason not to. Next, let's run this migration (it may take several seconds due to the volume of data being indexed).

```
$ bundle exec rails db:migrate
== 20160721030725 AddLowerIndexesToCustomers: migrating ===========
-- add_index(:customers, "lower(last_name) varchar_pattern_ops")
```

7. http://www.postgresql.org/docs/9.5/static/indexes-opclass.html

```
   -> 0.5506s
-- add_index(:customers, "lower(first_name) varchar_pattern_ops")
   -> 0.4963s
-- add_index(:customers, "lower(email)")
   -> 7.1292s
== 20160721030725 AddLowerIndexesToCustomers: migrated (8.1763s) ==
```

Before you try the app, let's run the EXPLAIN ANALYZE again and see what it says. Note the highlighted lines.

```
$ bundle exec rails dbconsole
shine_development> EXPLAIN ANALYZE
                     SELECT *
                     FROM   customers
                     WHERE
                       lower(first_name) like 'pat%' OR
                       lower(last_name)  like 'pat%' OR
                       lower(email)      = 'pat@example.com'
                     ORDER BY
                       email = 'pat@example.com' DESC,
                       last_name ASC
                     ;
                        QUERY PLAN
-----------------------------------------------------------------------------
Sort  (cost=5666.10..5679.16 rows=5224 width=79)
      (actual time=14.467..14.537 rows=704 loops=1)
   Sort Key: (((email)::text = 'pat@example.com'::text)) DESC, last_name
   Sort Method: quicksort  Memory: 124kB
➤    ->  Bitmap Heap Scan on customers
         (cost=145.31..5343.49 rows=5224 width=79)
         (actual time=0.387..8.650 rows=704 loops=1)
       Recheck Cond: ((lower((first_name)::text) ~~ 'pat%'::text) OR
                      (lower((last_name)::text) ~~ 'pat%'::text)  OR
                      (lower((email)::text) = 'pat@example.com'::text))
         Filter: ((lower((first_name)::text) ~~ 'pat%'::text) OR
                  (lower((last_name)::text) ~~ 'pat%'::text)  OR
                  (lower((email)::text) = 'pat@example.com'::text))
         Heap Blocks: exact=655
         ->  BitmapOr  (cost=145.31..145.31 rows=5250 width=0)
                       (actual time=0.263..0.263 rows=0 loops=1)
            -> Bitmap Index Scan on
                index_customers_on_lower_first_name_varchar_pattern_ops
                  (cost=0.00..41.92 rows=1750 width=0)
                  (actual time=0.209..0.209 rows=704 loops=1)
➤                 Index Cond: (
➤                   (lower((first_name)::text) ~>=~ 'pat'::text) AND
➤                   (lower((first_name)::text) ~<~ 'pau'::text))
            -> Bitmap Index Scan on
                index_customers_on_lower_last_name_varchar_pattern_ops
                  (cost=0.00..41.92 rows=1750 width=0)
                  (actual time=0.007..0.007 rows=0 loops=1)
```

```
                    Index Cond: (
                      (lower((last_name)::text) ~>=~ 'pat'::text) AND
                      (lower((last_name)::text) ~<~ 'pau'::text))
               ->  Bitmap Index Scan on index_customers_on_lower_email
                     (cost=0.00..57.55 rows=1750 width=0)
                     (actual time=0.046..0.046 rows=0 loops=1)
                      Index Cond: (
                        lower((email)::text) = 'pat@example.com'::text)
 Planning time: 0.193 ms
 Execution time: 14.732 ms
```

This time, there is *more* gobbledegook, but if you look closely, `Seq Scan on customers` is gone, and you can see a lot of detail around our `where` clause. The highlighted lines indicate *index scans*, in contrast to the Seq Scan you saw before. And the index scan is using our index and thus *not* examining each row in the table to find the correct results. You can see that it's doing three lookups, one for each field, using our indexes, and then `or`-ing the results together.

Setting aside the details of how Postgres does this, you can see that the results are about 40 times faster—the query should complete in under 15 milliseconds!

If you try our search in Shine now, the results come back almost instantly. We've improved the performance of our search by more than a factor of 40, all with just a few lines of SQL in a migration. *And* you didn't have to change a line of code in the Rails application. If you were using a less powerful database, you'd need to set up new infrastructure for making this search fast, and that could have a significant cost to development, maintenance, and production support.

This sort of index is just the tip of the iceberg—Postgres has many advanced features.

With our search performing better, let's take a final pass at the user interface. Bootstrap's default table styling made it a snap to create a reasonable user interface in no time. This then enabled us to focus on the Rails application's behavior and performance. If you stopped now and shipped what you have, you'd be shipping a feature you could be proud of. But, because you haven't spent *that* much time on this feature, let's see if there's any way to make the UI better for our users.