Database Administration
Lab 06: Indexing.

Andrés Oswaldo Calderón Romero, Ph.D.

September 16, 2025

1 Introduction

Efficient data retrieval is a fundamental aspect of database management, and indexing plays a crucial role in optimizing query performance. In this lab, we will explore the impact of indexing on query execution time by analyzing queries before and after the creation of indexes.

To accomplish this, we will work with a simulated dataset containing user accounts, conversation threads, and posts in a hypothetical forum. We will generate random data to populate the tables and execute a set of predefined queries to measure their performance. Using the `EXPLAIN ANALYZE` command in `PostgreSQL`, we will evaluate query execution times, compare results, and observe how indexing improves efficiency.

By the end of this lab, you will have a practical understanding of how indexing works, how it affects database performance, and how to apply indexing techniques to optimize query execution. This knowledge is essential for effective database administration and performance tuning in real-world applications.

This lab is based on the work presented in this video by Curtis Maloney[1]. We will share the first part here, and then you can continue on your own.

2 Homework

In this first section, we will address the homework proposed in class. Specifically, the task is to *summarize the sections describing the three types of extended statistics, each of which captures a different kind of correlation among column values:*

1. *Functional dependencies*
2. *Number of distinct values (NDV) counts*
3. *Most common values (MCV) lists*

These topics are covered in Chapter 13 of *PostgreSQL for Jobseekers* (2023), by Sonia Valeja and David Gonzales, starting on page 243. You are expected to **handwrite** a concise summary of these points, scan the handwritten work, and append it to the report. Only one summary is required for submission.

3 Data Generation and Performance Evaluation

In this section, we will first generate a dataset to simulate the activity of a forum with accounts, threads, and posts. This synthetic data will provide the foundation for executing a set of predefined queries. Once the data is

[1] Curtis Maloney, 2018. *PostgreSQL Indexing: How, Why, and When.* PyCon Australia. Sydney, August 24–28 2018. Available at https://2018.pycon-au.org/talks/42913-postgresql-indexing-how-why-and-when/

in place, we will measure the performance of these queries both before and after applying indexes. By comparing execution times using the `EXPLAIN ANALYZE` command in PostgreSQL, we will gain practical insights into how indexing strategies improve query efficiency. This integrated approach will allow us to connect the process of data generation directly with its impact on query performance.

3.1 Generating Random Data

We first create a simple schema consisting of three tables to store information about user accounts, conversation threads, and posts made by users in a hypothetical website forum. You must create and connect to a database called `posts`. Once in the `PostgreSQL` prompt, you will create the schema for the tables as follows:

```
CREATE TABLE account (
    account_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    dob DATE
);

CREATE TABLE thread (
    thread_id SERIAL PRIMARY KEY,
    account_id INTEGER NOT NULL REFERENCES account(account_id),
    title TEXT NOT NULL
);

CREATE TABLE post (
    post_id SERIAL PRIMARY KEY,
    thread_id INTEGER NOT NULL REFERENCES thread(thread_id),
    account_id INTEGER NOT NULL REFERENCES account(account_id),
    created TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT NOW(),
    visible BOOLEAN NOT NULL DEFAULT TRUE,
    comment TEXT NOT NULL
);
```

To simulate some data for populating the tables, we will use the `random()` function available in `PostgreSQL`. In addition, we will use a file containing 1,000 frequently used words to generate text fields. You can download the words file from here and place it in a folder you have access to, ensuring that the filename remains unchanged as `words`. Now, execute the following `SQL` code:

```
CREATE TABLE words (word TEXT);
COPY words(word) FROM '/usr/share/dict/words';
```

Be sure to update the full path to the words file according to its location on your system. Now, it's time to generate some random data:

```
-- Creating 100 dummy accounts...
INSERT INTO account (name, dob)
    SELECT
        substring('AEIOU', (random() * 4)::int + 1, 1) ||
        substring('bcdfghjklmnpqrstvwxyzbcdfghjklmnpqrstvwxyz', (random() * 21 * 2 + 1)::int, 2) ||
        substring('aeiou', (random() * 4 + 1)::int, 1) ||
```

```
        substring('bcdfghjklmnpqrstvwxyzbcdfghjklmnpqrstvwxyz', (random() * 21 * 2 + 1)::int, 2) ||
        substring('aeiou', (random() * 4 + 1)::int, 1),
        NOW() + ('1 days'::interval * random() * 365)
    FROM
        generate_series(1, 100);

-- Creating 1000 random threads...
INSERT INTO thread (account_id, title)
    SELECT
        random() * 99 + 1,
        (
            SELECT
                initcap(string_agg(word, ' '))
            FROM
                (TABLE words ORDER BY random() + n LIMIT 5) AS words (word)
        )
    FROM
        generate_series(1, 1000) AS s(n);

-- Let's work with 100K random posts...
INSERT INTO post (thread_id, account_id, created, visible, comment)
    SELECT
        random() * 999 + 1,
        random() * 99 + 1,
        NOW() - ('1 days'::interval * random() * 1000),
        CASE WHEN random() > 0.1 THEN TRUE ELSE FALSE END,
        (
            SELECT string_agg(word, ' ') FROM (TABLE words ORDER BY random() * n LIMIT 20) AS words(word)
        )
    FROM
        generate_series(1, 100000) AS s(n);
```

That will be enough. We are done here.

3.2 Measuring Performance Without and With Indexes

We will now address the following queries and evaluate their response times.

Q1 See all my posts

Q2 How many posts have I made?

Q3 See all current posts for a Thread

Q4 How many posts have I made to a Thread?

Q5 See all current posts for a Thread for this month, in order.

The purpose of this tutorial is to evaluate query execution performance before and after creating an index. For instance, for Q1, we will measure the current performance using only the default indexing (i.e., the indexes automatically created for primary keys).

Now, let's execute the query using the `EXPLAIN ANALYZE` clause to capture the execution time and relevant statistics:

```
EXPLAIN ANALYZE
SELECT
    *
FROM
    post
WHERE
    account_id = 1;
```

We should obtain the following response. Pay special attention to the `Execution Time` in the last line of the output:

```
posts=# EXPLAIN ANALYZE
SELECT
        *
FROM
        post
WHERE
        account_id = 1;
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Seq Scan on post  (cost=0.00..3395.00 rows=493 width=138) (actual time=0.034..10.786 rows=474 loops=1)
   Filter: (account_id = 1)
   Rows Removed by Filter: 99526
 Planning Time: 0.513 ms
 Execution Time: 10.857 ms
(5 rows)
```

Now, we will create an index on the `account_id` field:

```
CREATE INDEX ON post (account_id);
```

Then, let's re-run `EXPLAIN ANALYZE` and examine the new results:

```
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on post  (cost=8.11..1182.31 rows=493 width=138) (actual time=0.106..0.641 rows=474 loops=1)
   Recheck Cond: (account_id = 1)
   Heap Blocks: exact=424
   ->  Bitmap Index Scan on post_account_id_idx  (cost=0.00..7.99 rows=493 width=0) (actual time=0.062..0.062 rows=474 loops=1)
         Index Cond: (account_id = 1)
 Planning Time: 0.255 ms
 Execution Time: 0.694 ms
(7 rows)
```

Not bad at all! From here, you can continue watching the video and completing the remaining four queries. Be sure to track the execution time both before and after index creation, and then plot a graph to visually compare the performance of the queries with the best index implementation.

Use your preferred statistical software (e.g., R, Octave, Python, Scilab, Gnuplot, Grace, Veusz, etc.) to generate the graph.

4 Individual Work

As before, we will explore a similar approach but in a different environment. For this week's individual work, you will choose another DBMS (different to PostgreSQL) and investigate how it implements indexing, as well as the difference in performance before and after its implementation for the same five queries. Use the same dataset from this tutorial to gain a comprehensive understanding.

We expect you to submit your report by **September 29, 2025** at **2:00pm** via Brightspace™.

Happy Hacking! 😎