Databases
Lab 06: A ‘gentle’ Introduction to SQL Indexing.

Andrés Oswaldo Calderón Romero, Ph.D.

September 9, 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.

So let’s get it started

2 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:

```sql
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 (
```

[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/

```sql
    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 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.txt`. Now, execute the following `SQL` code:

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

Note: Be sure to update the full path to the `words` file according to its location on your system.

Now, it's time to generate random data:

```sql
-- 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 Measuring Performance Without and With Indexes

We will now address the following queries and evaluate their response times. We will assume that my account ID is 1.

Q1. View all my posts.

Q2. How many posts have I made?

Q3. View all current posts in a thread.

Q4. How many posts have I made in a thread?

Q5. View all current posts in 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 command to capture the execution time and relevant statistics:

```sql
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` column:

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

Then, let's re-run `EXPLAIN ANALYZE` and examine the updated 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 complete the remaining four queries.

As the deliverable for this section, you will measure the execution time of the four remaining queries (Q2–Q5) both **before** and **after** index creation. Then, you will create a graph to visually compare their performance.

In the video, several types of index implementations are presented –just select **the most effective one** for your analysis and plotting. Use your preferred statistical software (e.g., R, Octave, etc.) to generate the graphs.

Your final plot should include:

- The four queries (Q2 – Q5) on the **X-axis**.
- The execution time on the **Y-axis**.
- Two bars per query: one representing the execution time *before* index creation and the other *after* index creation.

4 Individual Work

You will read a few sections from Chapter 13 – Performance Tuning – of the book "PostgreSQL for Jobseekers" by Sonia Valeja and David Gonzales. You can download versions in English and Spanish. As you can see at the end, there are several types of indexes you can set when creating an index. For example, the following syntax will create an index named *account_id_hash_index* on the column *account_id* from the table `post` using the `HASH` index type:

```sql
CREATE INDEX account_id_hash_index ON post USING HASH (account_id);
```

Table 1 compares the different types of indexes in PostgreSQL. For the next exercise, you will work only with the following index types: B-tree, Hash, and BRIN. To do this, you will pick **just one** of the previous queries and measure the performance of the selected index types. As before, you will present your results by plotting them, where the X-axis represents the types of indexes, and the Y-axis represents the execution time as measured by the `EXPLAIN ANALYZE` command.

Index Type	**Best For**	**Supports Range Queries**	**Supports Full-Text Search**	**Ideal Data Types**
B-Tree (default)	General use, unique keys	Yes	No	Numbers, Text, Dates
Hash	Exact match searches	No	No	Numbers, Text
GIN	Full-text search, JSONB, Arrays	No	Yes	Arrays, JSONB, `tsvector`
GiST	Geospatial, Range queries	Yes	Yes	Geometries, Text
BRIN	Very large tables with ordered data	Yes	No	Large Time-Series, Logs
SP-GiST	Hierarchical & partitioned data	Yes	Yes	Trees, Text
Bloom	Multi-column searches	Yes	No	Multiple Columns

Table 1: Comparison of Index Types in PostgreSQL

We expect you to submit your report through Brightspace™ no later than September 24, 2025, before the lab session on that day.

Happy Hacking! 😎