Systems and Database Administration: Lab 4

Indexing

TU Dublin

TU-857/4



Overview

# Part 1. Introduction

## Today’s Lab

In today’s lab we’re going to generate some dummy data in our database to get tables big enough to make good use of indexes. We’re then going to examine the impact of indexes on selecting and inserting data. We’re going to look at how we can use the slow query log to identify long-running queries (these are prime candidates for indexes), and finally, we’re going to see how creating indexes can have some downsides in db maintenance.

## Before we Begin

We’re going to be using a custom script to generate data in our tables. It’s going to be far easier to load this script up in the host machine, so make sure you can connect from your host to your guest. I suggested you use Beekeeper, but you can use whatever IDE you like. To connect from the host to the guest you will need to

1. Boot up your virtual machine
2. Start the SQL server
   * Switch to the postgres user
   * Use the pg\_ctl start command (use the -D flag to specify the data directory)
3. Connect from Beekeeper (or IDE of your choice)
   * Username postgres, password whatever you set (I suggested password)

## Creating the Test Data

Download the indexing\_lab.sql file from brightspace and open it in your SQL IDE. Highlight everything in the script down to (but not including) the INSERT statement for the employees table. Click *run selection* to create the table and function we’ll need to hold and generate the data respectively.

This script contains some pl/pgsql which allows us to write code which runs in the database. We haven’t covered this in detail yet, but I’ve added some comments to explain what’s going on in case you’re interested. This is just a function to generate a random string which we’re going to use to populate our table.

Next up, we’re going to insert the test data. Highlight the INSERT statement on the employees table (ending with **FROM GENERATE\_SERIES(1, 50000);**). Click on *run selection*, it should take a second or two to finish. When it’s done, you’ll get a message saying that the query was executed successfully. The little clock icon down the bottom of the window will tell you how long the query took to execute. Take a screenshot of this value, you’ll upload it to Brightspace as part of the submission.

Graphical user interface, text, application

Description automatically generated

Now that we’ve generated a table full of random data we can run the next SELECT statement to view the top 10 rows.

A screenshot of a computer screen

Description automatically generated with medium confidence

Finally, we’re going to want to run some queries on this table, so we want to give at least one employee a first name and last name we know about. I’ve updated a random employee with the name “JACK ONEILL”; **change this query to match your name**, and use this whenever you are querying for a row.

Finally, we run the **VACUUM ANALYZE** command to tell the optimiser to go take a peek at our new table and start working how it’s going to go about optimising our queries. Postgres *should* look after doing this itself if it decides that a table has changed sufficiently, but it’s better not to leave these things to chance.

# Part 2. Estimating Query Execution Times

The **EXPLAIN** command takes any SQL statement, and tells us how the optimiser intends on running that query. It will also give an estimate of the relative cost of each step of the process. This can be a good way to work out which queries need to be optimised. Each step in the EXPLAIN plan has an associated *cost*. This cost is a somewhat arbitrary measurement including both disk access and CPU cycles required. We don’t really need to be able to map it back to actual times, but it allows us to see how expensive one query is relative to another.

Graphical user interface, text, application

Description automatically generated

The query plan above tells us that the optimiser is going to use an index (employees\_pk) to run this query. We saw this week that indexes are far faster than full table scans. We get two values for cost, separated by a double-dot. The first value, 0.29 in the example above, represents the **startup** cost, or how long until postgres can begin writing rows, the **total cost**, is how long this step will take from when it begins reading data to when it’s finished.

We can see above that we have a startup cost of 0.29 for this query (representing the time it will take to locate the index *etc.*. The total cost of this query will be 8.31 units. How big is this? Add a new select statement to select all columns where last\_name = ‘<Your Last Name>’. What’s the startup-cost/total cost for this query. Take a screenshot of this explain statement and the outputted query plan (as shown above). Why do you see a difference between both?

The **rows** value tells us how many rows the optimiser expects to return, and the **width** tells us how many bytes per row.

## Analysing queries

The EXPLAIN function allows us to get an estimate of the running time of the query but this isn’t guaranteed, and it’s given to us in a strange unit of cost that we can’t easily map back to actual time taken. If we add the ANALYZE keyword before our query then the SQL engine will actually execute the query and report back how long it actually took.

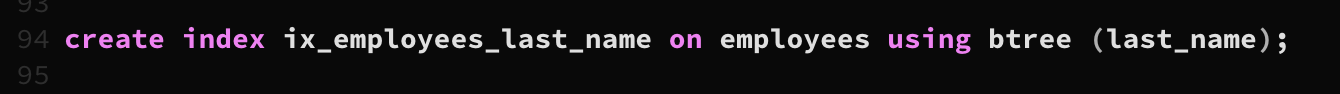
A screenshot of a computer

Description automatically generated with medium confidence

Notice we have two extra rows in our output now, planning time and execution time. We can see that our cost of 8.31 mapped back to about 0.5 milliseconds. Analysing queries is useful to get an actual running time but beware that it actually executes the query so you may find yourself waiting quite some time for the results to come back. Run EXPLAIN ANALYZE on your query by last\_name and compare the results. Take a screenshot like the one above for your submission.

## Creating an Index

We’re going to speed up queries on the last\_name column. Create a suitably named index on the table to index the last\_name column



After you create the index, run VACUUM ANANLYZE again to make sure the optimiser takes notice and re-run your explain analyze query. In your word document, state how many ms you shaved off the query using the index.

## BTree vs Hash Indexes

Run a select query and find out what date-of-birth has been assigned to the employee bearing your name.

Find the cost of a query to select **only the date of birth** in all rows from the table which share a date of birth with this employee.

Use EXPLAIN ANALYZE to find the actual cost of this query in milliseconds.

Create a new index on the date\_of\_birth column using **hash** and re-analyze the query.

Re-write the query to find all employees older than this employee.

Using the explain command, determine whether or not this index is used in your query (is it using a Sequential Scan or an index scan)?

Create another index on the date\_of\_birth column, this time using the btree algorithm.

Does the query now use the index? Have you noticed a speed-up? Add a screenshot of the EXPLAIN ANALYZE query plan to your word document.

### Does the Optimizer Always Use Indexes

You might find if you try to select all rows that one or both of the above queries will ignore the index entirely. The optimiser will only use the index if it feels like it’s going to speed up the query. Indexes are great, but if we’re selecting lots of rows anyway, then the extra overhead involved in finding the entry first in the index and then going and pulling from the table is often not worth the hassle. By restricting our query to only return the date\_of\_birth we’re ensuring that all of the data we need can be retrieved from the index and making it much more likely that the index will be used.

# Part 3 – Slow Running Queries

In this section we’re going to use the postgres logger to log slow running queries. Before we go any further, make sure we’ve disabled pgaudit so it doesn’t drown out our slow query logging. Run the following command in your SQL editor to set the pgaudit.log variable accordingly

set pgaudit.log = ‘none’;

Similarly, we want to disable regular postgres logging. Find the log\_statement configuration parameter in postgresql.conf and change it to ‘none’.

Finally, set the minimum duration before a statement gets logged as 5 seconds (5000 ms) by altering the **log\_min\_duration\_statement** parameter in your postgrresql.conf file.

Open up your VM and restart your database by typing the **pg\_ctl restart** command into the terminal. Don’t forget to use the -D flag to specify the data directory for your database.

## Simulating a Slow Running Query

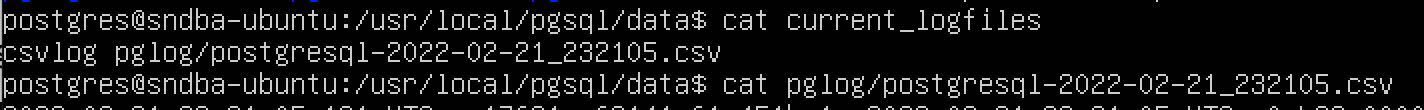
We’re going to simulate a slow-running query using the pg\_sleep function.

Graphical user interface

Description automatically generated with medium confidence

Run this query and wait 10 seconds while it executes. When it’s finished we want to check whether it’s been logged.

Hop onto your VM and check the contents of the data directory using ls. You should see a file called current\_logfiles. This file contains the name of the latest log file. Use the **cat** command to read out the contents of this file. You should see something like the below



Use the cat command on the current logfile to cat out the latest log, the last row should be your slow running query. Make sure it’s showing the query duration. Take a screenshot.

# Wrapping Up – The Downside of Indexes

Copy the SQL commands for creating each of the indexes and paste them just underneath your CREATE TABLE statement near the top of your script.

Drop the original table (make sure you use CASCADE to also drop the associated indexes). Re-create the table and add the indexes.

Next, re-run the insert statements to populate the table. Check how long it took to create. The larger a table the more of an impact each index will have on the time taken to update and insert data on that table. Use indexes where they help, but only when needed!

# Submission

For your submission include the following screenshots (directly, not as a zip).

1. The initial insert statement
2. The output of EXPLAIN
3. The output of EXPLAIN ANALYZE
4. The output of **cat** showing the *slow running quey*