DATABASE PERFORMANCE

FINAL ASSIGNMENT – REPORT

B-Tree Index Performance*

2022/2023

Professor: Maciej Zakrzewicz

- **Dinis Silvestre – ER1443**

I. INTRODUCTION

The goal of this report is to plan and execute my performance experiments, which evaluate the B*-tree index performance to answer several questions. The database system chosen was PostgreSQL and the VM used for the experiment was one made available by the professor.

The experiment's goal is to demonstrate the benefits and drawbacks of the B*-tree index in terms of query response time, index creation time and more. To reach a conclusion, a randomly generated workload with a mix of read and write operations and different query types that exercise the B*-tree index was created. This report contains a detailed explanation of everything I did to test the B*-tree index, and the delivery will include an attached file that is a copy of my terminal (CMD) testing.

The first step is to launch PostgreSQL as taught in class.

```
bash-4.4$ /usr/pgsql-14/bin/pg_ctl start
waiting for server to start.....2023-05-29 00:22:12.678 CEST [36704] LOG:  redirecting log output
 to logging collector process
2023-05-29 00:22:12.678 CEST [36704] HINT:  Future log output will appear in directory "log".
 done
server started
bash-4.4$ psql
psql (14.1)
Type "help" for help.

postgres=#
```

II. METHODOLOGY

Following the introduction, the B*-tree index must be created, and the workload must be run, which executes the workload against the database system. Once everything is tested, performance metrics such as query response time, query throughput, and index performance must be collected, and the results must be analysed using statistical techniques. This report compares the performance of queries with and without the B*-tree index and measures the impact of different index configurations on performance. Every task requested is completed and thoroughly explained:

- **Create a table with multiple columns of varying data types and sizes.**

```
postgres=# CREATE TABLE aa_table (id SERIAL PRIMARY KEY, short_text VARCHAR(50), long_text TEXT,
 number INT, date_column DATE);
CREATE TABLE
postgres=# CREATE TABLE bb_table (id SERIAL PRIMARY KEY, short_text VARCHAR(50), long_text TEXT,
 number INT, date_column DATE);
CREATE TABLE
postgres=# CREATE TABLE cc_table (id SERIAL PRIMARY KEY, short_text VARCHAR(50), long_text TEXT,
 number INT, date_column DATE);
CREATE TABLE
postgres=#
```

Three tables (aa_table, bb_table and cc_table) are being created. The 'id' column represents a primary key column that is automatically populated with a unique value for each row, 'short_text' is a varchar column that can store up to 50 characters of short text, 'long_text' is a text column that can store up to 1 GB of long text, 'number' is an integer column that can store whole numbers, and 'date_column' is a date column that can store dates.

- **Implement a script to generate several synthetic records of random values.**

```
postgres=# INSERT INTO aa_table (short_text, long_text, number, date_column) SELECT CONCAT('Record_
', id), CONCAT('This is long text for record ', id), FLOOR(RANDOM() * 1000000)::INT, DATE '2023-01-
01' + (FLOOR(RANDOM() * 365))::INT FROM generate_series(1, 100) AS id;
INSERT 0 100
postgres=# INSERT INTO bb_table (short_text, long_text, number, date_column) SELECT CONCAT('Record_
', id), CONCAT('This is long text for record ', id), FLOOR(RANDOM() * 1000000)::INT, DATE '2023-01-
01' + (FLOOR(RANDOM() * 365))::INT FROM generate_series(1, 10000) AS id;
INSERT 0 10000
postgres=# INSERT INTO cc_table (short_text, long_text, number, date_column) SELECT CONCAT('Record_
', id), CONCAT('This is long text for record ', id), FLOOR(RANDOM() * 1000000)::INT, DATE '2023-01-
01' + (FLOOR(RANDOM() * 365))::INT FROM generate_series(1, 1000000) AS id;
INSERT 0 1000000
postgres=#
```

I wrote the following script to generate synthetic records: 100 for the aa_table, 10,000 for the bb_table, and 1,000,000 for the cc_table. I used the 'generate_series' function in PostgreSQL to generate a series of numbers, which I then used in the SELECT statement to insert synthetic records into the table. The RANDOM() function generates random values for the number column, while the DATE keyword specifies a date value.

- **Create B*-tree indexes on table columns and on column expressions.**

```
postgres=# CREATE INDEX id_short_text ON aa_table (short_text);
CREATE INDEX
postgres=# CREATE INDEX id_upper_short_text ON aa_table (upper(short_text));
CREATE INDEX
postgres=#
```

After implementing the script, I created two B*-tree indexes in the aa_table on the 'short_text' and 'upper(short_text)' columns, each one representing:

  - id_short_text: B*-tree index on the short_text column.
  - id_upper_short_text: Creates an index on the uppercase version of the short_text column.

- **Implement a set of SQL queries that use the indexes.**

```
postgres=# SELECT * FROM aa_table WHERE short_text = 'Record_1';
 id | short_text |           long_text            | number | date_column
----+------------+--------------------------------+--------+-------------
  1 | Record_1   | This is long text for record 1 |  60483 | 2023-07-26
(1 row)

postgres=# SELECT * FROM aa_table WHERE upper(short_text) = 'RECORD_1';
 id | short_text |           long_text            | number | date_column
----+------------+--------------------------------+--------+-------------
  1 | Record_1   | This is long text for record 1 |  60483 | 2023-07-26
(1 row)
```

After creating two indexes, I made a simple example of how to perform simple SQL queries to demonstrate that everything is going as planned and that it works by querying an indexed column with a value that exists in the table. If it was, for example, 'SELECT * FROM aa_table WHERE number > 500000;' it represents a query without indexes as the 'number' column is not indexed.

- **Implement a set of INSERT/UPDATE/DELETE commands to modify the table.**

The 'EXPLAIN ANALYZE' prefix is used in each example to obtain the execution plan as well as the actual execution time. Running these commands gives me information about the query, statement planning, and execution times, which allows me to compare the performance of various operations. These commands also assist me in considering the cost and time ranges of each operation. These SQL commands enable me to add, update, and delete records in a specific table while keeping in mind that adding, updating, or deleting records may have an impact on index performance.

The smaller the table is, the less time is required for planning and execution.

- **INSERT command to add a new record.**

```
postgres=# EXPLAIN ANALYZE INSERT INTO aa_table (short_text, long_text, number, date_column) VALUE
S ('New Record', 'This is a new record', 123456, '2022-01-01');
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Insert on aa_table  (cost=0.00..0.01 rows=0 width=0) (actual time=0.817..0.818 rows=0 loops=1)
   ->  Result  (cost=0.00..0.01 rows=1 width=162) (actual time=0.110..0.110 rows=1 loops=1)
 Planning Time: 0.091 ms
 Execution Time: 1.084 ms
(4 rows)
```

Includes the measurement of the execution time and cost range of an INSERT statement.

- **UPDATE command to modify an existing record.**

```
postgres=# EXPLAIN ANALYZE UPDATE aa_table SET long_text = 'Updated text'WHERE short_text = 'Recor
d_1';
                                              QUERY PLAN

-----------------------------------------------------------------------------------------------
-------
 Update on aa_table  (cost=0.00..3.25 rows=0 width=0) (actual time=0.429..0.429 rows=0 loops=1)
   ->  Seq Scan on aa_table  (cost=0.00..3.25 rows=1 width=38) (actual time=0.023..0.043 rows=1 lo
ops=1)
         Filter: ((short_text)::text = 'Record_1'::text)
         Rows Removed by Filter: 100
 Planning Time: 1.086 ms
 Execution Time: 0.590 ms
(6 rows)
```

Includes the measurement of the execution time and cost range of an UPDATE statement.

- **DELETE command to remove a record.**

```
postgres=# EXPLAIN ANALYZE DELETE FROM aa_table WHERE short_text = 'Record_4';
                                              QUERY PLAN

-----------------------------------------------------------------------------------------------
------
 Delete on aa_table  (cost=0.00..3.25 rows=0 width=0) (actual time=0.531..0.532 rows=0 loops=1)
   ->  Seq Scan on aa_table  (cost=0.00..3.25 rows=1 width=6) (actual time=0.104..0.123 rows=1 loo
ps=1)
         Filter: ((short_text)::text = 'Record_4'::text)
         Rows Removed by Filter: 100
 Planning Time: 0.112 ms
 Execution Time: 0.560 ms
(6 rows)
```

Includes the measurement of the execution time and cost range of a DELETE statement.

III. RESULTS AND ANALYSIS

After demonstrating how I experimentally evaluate B*-tree index performance, I must now answer several questions, draw conclusions, summarise my findings, and provide recommendations for improving B*-tree index performance. This section of the report contains graphs, tables, and charts that show my findings.

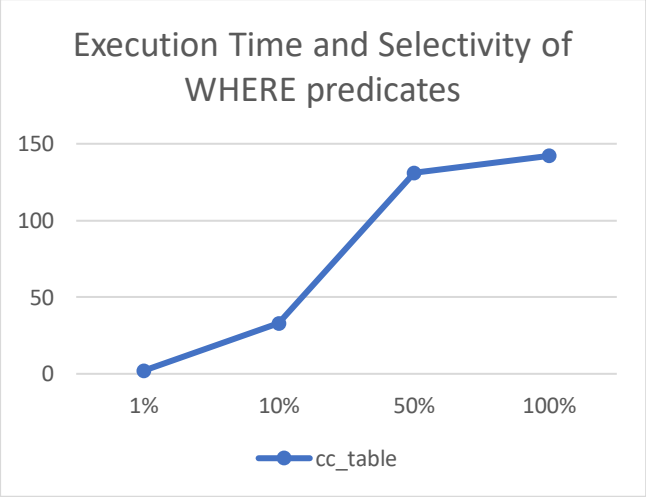
- **How much can indexes reduce query execution times** (vs. sequential scan) **and how accurate are query optimizer cost estimates** (vs. real execution time) for index-based query execution plans in relation to:
  - Table size (number of records)

Table Size	*Column Status*	Estimated Cost	Execution Time (milliseconds)
100 records	*Indexed*	0.00…3.25	0.025
	Non-Indexed	0.00…3.25	0.104
10.000 records	*Indexed*	0.00…238.00	0.933
	Non-Indexed	0.00…238.00	2.247
1.000.000 records	*Indexed*	0.00…23863.00	124.692
	Non-Indexed	0.00…23863.00	154.214

I compare the execution times of the same query across the three tables to assess the impact of table size on cost estimation accuracy and execution time ('EXPLAIN ANALYSE SELECT * FROM cc_table WHERE number > 500000;'). I run the query without an indexed column first, then add a B*-tree index to the 'number' column and compare the results. Indexed columns take less time to execute than non-indexed columns, regardless of table size.

  - Selectivity of WHERE predicates (percentage of records retrieved)

Table Size	*Selectivity*	Estimated Cost	Execution Time (milliseconds)
1.000.000 Records	*1%*	0.42…387.57	2.092
	10%	0.42…4047.24	32.907
	50%	0.42…19887.73	131.096
	100%	0.00…23863.00	142.114



I altered the queries by changing the WHERE predicates to retrieve different percentages of records.

'EXPLAIN ANALYZE SELECT * FROM cc_table WHERE id <= 500000', for example, returned 50% of the table's records. The table has a million records.

The conclusion is that, for large tables, the higher the selectivity, the longer the execution time, and the higher the estimated cost.

- Indexed column data type/size, column expression

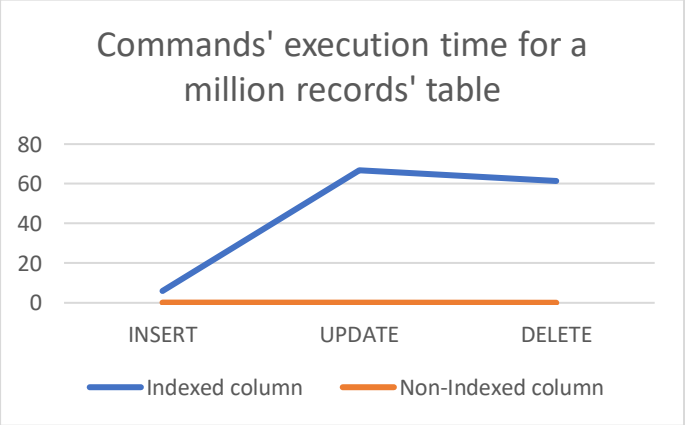
Table Size	*Data Type/Size*	Estimated Costs	Execution Time (milliseconds)
1.000.000 Records	*TEXT*	0.42…19887.73	118.468
	INTEGER	0.42…19887.73	70.209

I compare a WHERE predicate for an integer (WHERE id = 4) and a text (WHERE id = "4") to evaluate the impact of indexed column data type/size and column expressions on cost estimation accuracy and execution time. In comparison to text and VARCHAR, for example, the integer has the lowest estimated costs and execution time.

Indexes are most useful for large tables with high selectivity queries involving simple column comparisons. The execution time difference is significant. The difference in execution time between an indexed and non-indexed column grows with the number of records.

- How much do **indexes degrade the performance of INSERT/UPDATE/DELETE commands?**

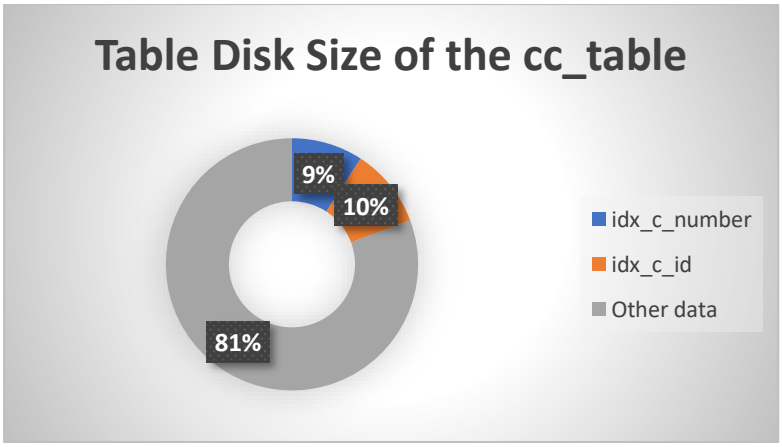
Table Size	Command	Column Status	Execution Time (milliseconds)
1.000.000 Records	*INSERT*	Indexed	5.970
		Non-Indexed	0.126
	UPDATE	Indexed	66.798
		Non-Indexed	0.068
	DELETE	Indexed	61.474
		Non-Indexed	0.043



Indexes can degrade the performance of INSERT/UPDATE/DELETE commands, particularly in large tables with many indexes. This is since each index must be updated whenever a new row is inserted, or an existing row is updated or deleted. The precise performance impact is determined by several factors, including the number of indexes, index size, and others.

- How much **disk space do indexes consume** in relation to:
  - Table size (number of records)

Table Size	Table Disk Space	*Index Name*	Index Disk Space
100 Records	128 kB	*idx_a_number*	16 kB
		Idx_a_id	16 kB
10.000 Records	1664 kB	*idx_b_number*	240 kB
		Idx_b_id	240 kB
1.000.000 Records	207 MB	*idx_c_number*	19 MB
		Idx_c_id	21 MB



Indexes take up a lot of disk space, especially for large tables. The difference in disk space between tables is significant.

The size of the indexes is generally proportional to the size of the table and the number of indexed columns, so the total size of the indexes should not exceed 30-40% of the total size of the table.

'SELECT pg_size_pretty(pg_total_relation_size('idx_c_id')) AS index_size;' gives the total size of the index, including the index's main data structure and associated metadata.

- Indexed column data type/size, column expression

Table Size	*Data Type*	Index Disk Space (MB)
1.000.000 Records	*INTEGER*	21
	TEXT	30

Between two 'SELECT pg_size_pretty(pg_total_relation_size('cc_table'));' - to get the table size, I created two indexes: one for an integer-only column and another for a text-only column. After that, I subtracted the after and before values to get the final value. When creating an index, the column that only accepts integers takes up the least amount of disk space.

- How much **time does it take to build a new index** in relation to:
  - Table size (number of records)

Table Size	Index Creation Time (milliseconds)
100 Records	16.053
10.000 Records	23.605
1.000.000 Records	823.882

The larger the table, the longer it takes to build the index. This is since creating the index necessitates scanning the entire table and creating index entries for each row.

- Indexed column data type/size, column expression

Table Size	*Data Type*	Index Creation Time (milliseconds)
1.000.000 Records	*INTEGER*	5569.692
	TEXT	11451.918

The time required to build the index is affected by the data type and size of the indexed column. Indexing a large text column, for example, may take longer than indexing a small integer column.

IV. CONCLUSION

The experiment examined the impact of indexes on database performance optimisation. According to my research, indexes can significantly reduce query execution times, particularly for large tables and selective queries. While indexes have a minor impact on the performance of data modification operations and take up more disk space than table data, the benefits of faster data retrieval outweigh the minor slowdown. The results presented have implications for database performance optimisation, emphasising the importance of strategic index selection and design.