Faculdade de Ciências da Universidade de Lisboa

Indexing and Optimization

Advanced Databases - Report 3

Group 1

Daniela Vieira - 57634

João Raimundo - 57454

João Rato - 57434

Maria Vieira - 53346

Oriented by: Professor Cátia Pesquita

13 December 2021

1. Introduction

On the two past projects the goals were to create queries in both Relational (*Postgresql*) and *NoSQL* databases (*MongoDB*) making them interfere with each other. Then we were meant to control these interferences using locks, and finally we arrived at Project 3.

The first step of this project was to optimize our two queries and the main goal here was to improve the performance of them by having an execution and planning time as low as possible.

We analyzed their performance for these values and observed the query plan, which we will be showing later this report.

To achieve the desired optimization we used indexes, such as *B-tree* and *Clustered*.

We will proceed then to explain our decisions regarding alterations to the data model, both *RDB* and *NoSQL*, and finally we will finish with a discussion of the conclusions we took and the procedures we performed in this project.

1.1 Indexes

Indexes are a common way to enhance database performance and allow the database server to find and retrieve specific rows much faster than without it.

Thus, there are several types of indexes that vary according to the needs of the queries that we want to perform in a database, that can be:

- *Clustered* index that stores and sorts rows of data in a view or table depending on their values. It is used when adjustment of gigantic information is needed in any data set;
- *Non-clustered* index which represents a structure which is isolated from data rows. These types of indexes in SQL servers cover the *non-clustered* key values and each worth pair has a pointer to the data row that comprises vital significance - The main difference between *clustered* and *non-clustered* indexes is that the *non-clustered* index stores the data at one area and indices at another area, while the *clustered* index sorts the data rows in the table based on their key values;
- *Column store* index allows to store away information inside little impressions. The goal of this type of index is putting away and questioning enormous data warehousing truth tables;
- *Filtered* index is used when a column has just a few applicable numbers for questions on the subset of values;
- Hash index which can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the '=' operator;
- *Unique* index confirms and guarantees that the index key doesn't contain any copy esteems and along these lines, allows to examine that each row in the table is exceptional in either way. It's used when we need to have an extraordinary trait of every information;
- B-tree index can handle equality and range queries on data that can be sorted into some ordering. B-trees are used when an indexed column is involved in a comparison using one of these operators: '*=', '<', '<=', '>'* and *'>='*;

On this project we used the *B-Tree* and *Clustered* indexes, to optimize our database querying times. Since the *hash* index works better with '=' operator as we explained above, we decided to not use it, since it didn't fit with our query needs.

2 - *PostgreSQL* Database Tuning and Query Optimization

To optimize the *query 1* conceptual schema, we could apply a denormalization into the table albums, adding a field with the genre name, decreasing the number of joins necessary to be executed.

In both our queries we are only interested in specific decades (90's and 00's), so to improve the performance we could apply a horizontal decomposition replacing the "*release_date*" attribute into new relations.

Although, we didn't proceed to implement this kind of schema optimizations, once our querying times without optimization were already short - we are only querying merely for one row.

2.1 - Query 1

For *query 1* in *PostgreSQL* database, we started to inspect the query plan without indexes integrating the statement "*EXPLAIN ANALYSE SELECT*" at the start of the SELECT query, like presented in *Appendix 1*.

Figure 1 demonstrates our *query 1* plan output where we can perceive that, at the moment, it is performing in '148.778 ms' with '86.578 ms' of planning time.

```
                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=3563.78..3563.80 rows=1 width=504) (actual time=148.644..148.673 rows=1 loops=1)
   ->  Group  (cost=3563.78..3563.93 rows=10 width=504) (actual time=148.638..148.664 rows=1 loops=1)
         Group Key: albums.sales, albums.band_id, albums.release_date, albums.abstract, genres.genre_name
         ->  Sort  (cost=3563.78..3563.81 rows=10 width=504) (actual time=148.631..148.655 rows=1 loops=1)
               Sort Key: albums.sales DESC, albums.band_id, albums.release_date, albums.abstract
               Sort Method: quicksort  Memory: 35kB
               ->  Nested Loop  (cost=415.34..3563.62 rows=10 width=504) (actual time=83.918..148.549 rows=14 loops=1)
                     Join Filter: (albums.band_id = bands.band_id)
                     ->  Hash Join  (cost=415.06..3560.10 rows=11 width=508) (actual time=83.871..148.236 rows=14 loops=1)
                           Hash Cond: (albums.band_id = bands_genre.band_id)
                           ->  Seq Scan on albums  (cost=0.00..3136.76 rows=2179 width=492) (actual time=8.642..81.148 rows=4841 loops=1)
                                 Filter: ((release_date >= '1990-01-01'::date) AND (release_date <= '1999-12-31'::date) AND (length(abstract) > 200))
                                 Rows Removed by Filter: 29247
                           ->  Hash  (cost=414.51..414.51 rows=44 width=16) (actual time=61.220..61.232 rows=62 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 11kB
                                 ->  Hash Join  (cost=10.66..414.51 rows=44 width=16) (actual time=0.270..61.105 rows=62 loops=1)
                                       Hash Cond: (bands_genre.genre_id = genres.genre_id)
                                       ->  Seq Scan on bands_genre  (cost=0.00..341.32 rows=23632 width=8) (actual time=0.060..33.324 rows=23632 loops=1)
                                       ->  Hash  (cost=10.65..10.65 rows=1 width=16) (actual time=0.142..0.146 rows=1 loops=1)
                                             Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                             ->  Seq Scan on genres  (cost=0.00..10.65 rows=1 width=16) (actual time=0.021..0.111 rows=1 loops=1)
                                                   Filter: ((genre_name)::text = 'Math rock'::text)
                                                   Rows Removed by Filter: 531
                     ->  Index Only Scan using bands_pkey on bands  (cost=0.29..0.31 rows=1 width=4) (actual time=0.017..0.017 rows=1 loops=14)
                           Index Cond: (band_id = bands_genre.band_id)
                           Heap Fetches: 0
 Planning Time: 86.578 ms
 Execution Time: 148.778 ms
(28 rows)
```

Figure 1 - Output of the *query 1* execution plan without optimization in *PostgresSQL*.

2.1.1 - Creating Indexes

In *query 1* we have two main motivations to create an index, which are the range selections for the ‘*release_date’* and ‘*abstract’* attributes and sorting these attributes (including the ‘sales’ attribute included). We have seen above that this operations takes some time to be obtained, so to decrease the execution time to create a *B-tree* composite index (‘*date_abstract*’, *Figure 2*) on ‘*release_date*’ and the length of the abstract (‘*length(abstract)*’) to help the sorting of these attributes - *B- tree* indexes support range searches that can be sorted into some ordering efficiently.

Performing the query plan again (*Figure 1*) we conclude that the composite index ‘*date_abstract*’ have optimize our query execution time in 64.466 ms (148.778 ms - 84.312 ms = 64.466 ms) and the planning time in 55.299 ms (86.578ms - 30.279ms = 55.299 ms), applying a ‘*Bitmap Index Scan on date_abstract*’ in ‘*release_date’* and ‘*abstract*’ attributes (*Figure 2*).

Even though it's not fully optimized since the join operations to get the ‘*genre_name*’ attribute was not optimized.

```
CREATE INDEX date_abstract ON albums USING btree (release_date, length(abstract));

--- Output Query 1 Plan with data_abstract index

                                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=3112.42..3112.43 rows=1 width=504) (actual time=83.892..83.926 rows=1 loops=1)
   ->  Group  (cost=3112.42..3112.57 rows=10 width=504) (actual time=83.885..83.916 rows=1 loops=1)
         Group Key: albums.sales, albums.band_id, albums.release_date, albums.abstract, genres.genre_name
         ->  Sort  (cost=3112.42..3112.44 rows=10 width=504) (actual time=83.879..83.908 rows=1 loops=1)
               Sort Key: albums.sales DESC, albums.band_id, albums.release_date, albums.abstract
               Sort Method: quicksort  Memory: 35kB
               ->  Nested Loop  (cost=569.88..3112.25 rows=10 width=504) (actual time=63.279..83.706 rows=14 loops=1)
                     Join Filter: (albums.band_id = bands.band_id)
                     ->  Hash Join  (cost=569.59..3108.73 rows=11 width=508) (actual time=63.207..83.384 rows=14 loops=1)
                           Hash Cond: (albums.band_id = bands_genre.band_id)
                           ->  Bitmap Heap Scan on albums  (cost=154.53..2685.40 rows=2179 width=492) (actual time=1.638..17.599 rows=4841 loops=1)
                                 Recheck Cond: ((release_date >= '1990-01-01'::date) AND (release_date <= '1999-12-31'::date) AND (length(abstract) > 200))
                                 Heap Blocks: exact=474
                                 ->  Bitmap Index Scan on date_abstract  (cost=0.00..153.99 rows=2179 width=0) (actual time=1.496..1.499 rows=4841 loops=1)
                                       Index Cond: ((release_date >= '1990-01-01'::date) AND (release_date <= '1999-12-31'::date) AND (length(abstract) > 200))
                           ->  Hash  (cost=414.51..414.51 rows=44 width=16) (actual time=59.980..59.992 rows=62 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 11kB
                                 ->  Hash Join  (cost=10.66..414.51 rows=44 width=16) (actual time=0.340..59.831 rows=62 loops=1)
                                       Hash Cond: (bands_genre.genre_id = genres.genre_id)
                                       ->  Seq Scan on bands_genre  (cost=0.00..341.32 rows=23632 width=8) (actual time=0.076..30.948 rows=23632 loops=1)
                                       ->  Hash  (cost=10.65..10.65 rows=1 width=16) (actual time=0.177..0.180 rows=1 loops=1)
                                             Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                             ->  Seq Scan on genres  (cost=0.00..10.65 rows=1 width=16) (actual time=0.029..0.156 rows=1 loops=1)
                                                   Filter: ((genre_name)::text = 'Math rock'::text)
                                                   Rows Removed by Filter: 531
                     ->  Index Only Scan using bands_pkey on bands  (cost=0.29..0.31 rows=1 width=4) (actual time=0.018..0.018 rows=1 loops=14)
                           Index Cond: (band_id = bands_genre.band_id)
                           Heap Fetches: 0
 Planning Time: 30.279 ms
 Execution Time: 84.312 ms
(30 rows)
```

Figure 2 - Script used to create the index ‘*date_abstract*’ and the *query 1* execution plan output using this index in *PostgreSQL*.

Additionally, to compare the *query 1* optimization process between *PostgreSQL* and *MongoDB* we created a *B-tree* composite index for the ‘*sales*’ and ‘*release_date*’ attributes, even though this was not the best index to optimize our *query 1* execution time.

Figure 3 shows the used script to create the composite index *(‘sales_date’*) and the output of the performed query plan when using it. The used of this index have shown worst execution time results (‘184.374 ms’) when compared with the *query 1* execution plan without using an index (‘148.778 ms’). Although the planning time was fastest (‘10.715 ms’) – without using and index took ’30.279 ms’ of planning time.

```
CREATE INDEX sales_date ON albums USING btree (sales, release_date);

                                                                QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=1009.11..2216.25 rows=1 width=506) (actual time=176.687..183.905 rows=1 loops=1)
   ->  Group  (cost=1009.11..13080.51 rows=10 width=506) (actual time=176.681..183.895 rows=1 loops=1)
         Group Key: albums.sales, albums.band_id, albums.release_date, albums.abstract, genres.genre_name
         ->  Nested Loop  (cost=1009.11..13080.39 rows=10 width=506) (actual time=176.677..183.889 rows=1 loops=1)
               Join Filter: (albums.band_id = bands.band_id)
               ->  Nested Loop  (cost=1008.82..13076.87 rows=11 width=510) (actual time=176.618..183.825 rows=1 loops=1)
                     Join Filter: (bands_genre.genre_id = genres.genre_id)
                     Rows Removed by Join Filter: 8146
                     ->  Nested Loop  (cost=1008.82..12979.18 rows=5802 width=502) (actual time=38.756..136.871 rows=8147 loops=1)
                           ->  Gather Merge  (cost=1008.54..11901.25 rows=2189 width=494) (actual time=37.287..59.038 rows=2629 loops=1)
                                 Workers Planned: 1
                                 Workers Launched: 1
                                 ->  Incremental Sort  (cost=8.53..10654.98 rows=1288 width=494) (actual time=1.797..18.702 rows=1365 loops=2)
                                       Sort Key: albums.sales DESC, albums.band_id, albums.release_date, albums.abstract
                                       Presorted Key: albums.sales
                                       Full-sort Groups: 23  Sort Method: quicksort  Average Memory: 69kB  Peak Memory: 69kB
                                       Worker 0:  Full-sort Groups: 62  Sort Method: quicksort  Average Memory: 65kB  Peak Memory: 66kB
                                       ->  Parallel Index Scan Backward using sales_date on albums  (cost=0.29..10597.02 rows=1288 width=494) (actual time=0.168..14.625 rows=1380 loops=2)
                                             Index Cond: ((release_date >= '1990-01-01'::date) AND (release_date <= '1999-12-31'::date))
                                             Filter: (length(abstract) > 200)
                                             Rows Removed by Filter: 474
                           ->  Index Only Scan using bands_genre_pkey on bands_genre  (cost=0.29..0.46 rows=3 width=8) (actual time=0.016..0.020 rows=3 loops=2629)
                                 Index Cond: (band_id = albums.band_id)
                                 Heap Fetches: 0
                     ->  Materialize  (cost=0.00..10.66 rows=1 width=16) (actual time=0.001..0.002 rows=1 loops=8147)
                           ->  Seq Scan on genres  (cost=0.00..10.65 rows=1 width=16) (actual time=0.031..0.128 rows=1 loops=1)
                                 Filter: ((genre_name)::text = 'Math rock'::text)
                                 Rows Removed by Filter: 531
               ->  Index Only Scan using bands_pkey on bands  (cost=0.29..0.31 rows=1 width=4) (actual time=0.053..0.054 rows=1 loops=1)
                     Index Cond: (band_id = bands_genre.band_id)
                     Heap Fetches: 0
 Planning Time: 10.715 ms
 Execution Time: 184.374 ms
(33 rows)
```

Figure 3 - Script used to create the index '*sales_date*' and the *query 1* execution plan output using this index in *PostgreSQL*.

2.2 - Query 2

The performed the same process used in *query 1* for the *query 2. Appendix 2* describes the Script used to explain the *query 2 plan and its respective output are presented in Figure 4.* Without using any index *query 2* performed in '105.984 ms' with '1.245 ms' of planning time.

```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=3231.48..3231.48 rows=1 width=16) (actual time=105.643..105.653 rows=1 loops=1)
   ->  Sort  (cost=3231.48..3247.17 rows=6276 width=16) (actual time=105.638..105.644 rows=1 loops=1)
         Sort Key: sales DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=3137.34..3200.10 rows=6276 width=16) (actual time=82.943..94.541 rows=9291 loops=1)
               Group Key: sales, band_id, release_date, running_time
               Batches: 1  Memory Usage: 913kB
               ->  Seq Scan on albums a  (cost=0.00..3051.54 rows=8580 width=16) (actual time=20.549..68.347 rows=9291 loops=1)
                     Filter: ((running_time >= '45'::real) AND (release_date >= '2000-01-01'::date) AND (release_date <= '2010-12-31'::date))
                     Rows Removed by Filter: 24797
 Planning Time: 1.245 ms
 Execution Time: 105.984 ms
(12 rows)
```

Figure 4 - Output of the *query 2* execution plan without optimization in *PostgreSQL*.

2.2.1 - Creating Indexes

For the *query 2* index we chose to create a *B-tree* composite index (*'sales_date_time', Figure 5*) on the attributes '*sales*', '*running_time*' and '*release_date*' attributes. The main reason for this choice was since our queries need to sort these attributes by a distinct order and range select '*running_time*' and '*release_date*' attributes, which are the operations that require more computation time (range selections).

Figure 5 shows that our *query 2* plan starts by using the index '*sales_time_date*' ('*Index Scan Backward using sales_time_date'*) to sort the values of our table. The use of the index drastically decreases the execution time to '1.019 ms' with '13.946 ms' of planning time.

```
--- Create index sales_time_date

CREATE INDEX sales_time_date ON albums USING btree (sales,running_time,release_date);

                                                        QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=2.03..3.82 rows=1 width=16) (actual time=0.848..0.859 rows=1 loops=1)
   ->  Group  (cost=2.03..11258.45 rows=6276 width=16) (actual time=0.842..0.850 rows=1 loops=1)
         Group Key: sales, band_id, release_date, running_time
         ->  Incremental Sort  (cost=2.03..11172.65 rows=8580 width=16) (actual time=0.839..0.844 rows=1 loops=1)
               Sort Key: sales DESC, band_id, release_date, running_time
               Presorted Key: sales
               Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 26kB  Peak Memory: 26kB
               ->  Index Scan Backward using sales_time_date on albums a  (cost=0.29..10862.49 rows=8580 width=16) (actual time=0.087..0.717 rows=36 loops=1)
                     Index Cond: ((running_time >= '45'::real) AND (release_date >= '2000-01-01'::date) AND (release_date <= '2010-12-31'::date))
 Planning Time: 13.946 ms
 Execution Time: 1.019 ms
(11 rows)
```

Figure 5 - Script used to create the index '*sales_time_date*' and the *query 2* plan output using this index in *PostgreSQL*.

2.3 - *Clustered* index

We proceeded to evaluate if the use of a clustered index could optimize the queries performance. Thus, given the fact that both queries rely on range selections for the '*release_date*' attribute, a *B-tree* single field index ('*date*') was created on this attribute. Next, we use this new index to cluster the '*albums*' table by '*release_date' - Figure 6.*

```
-- Cluster albums table by release date, with the index date

CREATE INDEX date ON albums USING btree (release_date);

CLUSTER albums USING date;
ANALYZE albums;
```

Figure 6 - Script used to create the index '*date*' and cluster the '*albums'* table by '*release_date*' attribute.

With the clustered index implemented we executed the '*EXPLAIN ANALYSE*' procedure for *query 1* and 2 (*Appendix* 1 and 2 respectively), to evaluate the queries performance:

In *Figure 7* we have the output of *query 1* plan, where we can infer that the planning time was reduced in 29.121 ms (30.279 ms - 1.158 ms = 29.121 ms) - due to clustered indexes stores and sorts rows of data in a view or table based on their key values - and the execution time was slightly the same in comparison with the usage of the '*date_abstract*' composite index *(Figure 2).*

```
--- EXPLAIN ANALYSE QUERY 1 WITH the albums table clustered by release date

                                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=1126.69..1126.71 rows=1 width=506) (actual time=80.687..80.716 rows=1 loops=1)
   ->  Group  (cost=1126.69..1127.02 rows=22 width=506) (actual time=80.682..80.708 rows=1 loops=1)
         Group Key: albums.sales, albums.band_id, albums.release_date, albums.abstract, genres.genre_name
         ->  Sort  (cost=1126.69..1126.75 rows=22 width=506) (actual time=80.677..80.701 rows=1 loops=1)
               Sort Key: albums.sales DESC, albums.band_id, albums.release_date, albums.abstract
               Sort Method: quicksort  Memory: 35kB
               ->  Nested Loop  (cost=415.63..1126.20 rows=22 width=506) (actual time=58.888..80.652 rows=14 loops=1)
                     Join Filter: (albums.band_id = bands.band_id)
                     ->  Hash Join  (cost=415.35..1118.20 rows=25 width=510) (actual time=58.871..80.525 rows=14 loops=1)
                           Hash Cond: (albums.band_id = bands_genre.band_id)
                           ->  Index Scan using date on albums  (cost=0.29..684.44 rows=4924 width=494) (actual time=0.047..17.814 rows=4841 loops=1)
                                 Index Cond: ((release_date >= '1990-01-01'::date) AND (release_date <= '1999-12-31'::date))
                                 Filter: (length(abstract) > 200)
                                 Rows Removed by Filter: 1717
                           ->  Hash  (cost=414.51..414.51 rows=44 width=16) (actual time=56.719..56.731 rows=62 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 11kB
                                 ->  Hash Join  (cost=10.66..414.51 rows=44 width=16) (actual time=0.218..56.609 rows=62 loops=1)
                                       Hash Cond: (bands_genre.genre_id = genres.genre_id)
                                       ->  Seq Scan on bands_genre  (cost=0.00..341.32 rows=23632 width=8) (actual time=0.074..28.020 rows=23632 loops=1)
                                       ->  Hash  (cost=10.65..10.65 rows=1 width=16) (actual time=0.086..0.090 rows=1 loops=1)
                                             Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                             ->  Seq Scan on genres  (cost=0.00..10.65 rows=1 width=16) (actual time=0.029..0.075 rows=1 loops=1)
                                                   Filter: ((genre_name)::text = 'Math rock'::text)
                                                   Rows Removed by Filter: 531
                     ->  Index Only Scan using bands_pkey on bands  (cost=0.29..0.31 rows=1 width=4) (actual time=0.003..0.004 rows=1 loops=14)
                           Index Cond: (band_id = bands_genre.band_id)
                           Heap Fetches: 0
 Planning Time: 1.158 ms
 Execution Time: 80.790 ms
(29 rows)
```

Figure 7 - *Output of query 1 execution plan* with the '*albums*' table clustered by '*release_date*' attribute in *PostgreSQL*.

In the output of *query 2* execution plan (*Figure 8*) the planning time was reduced in 6.842 ms (13.946 ms - 7.104 ms = 6.842 ms) and the execution time it was also slightly the same in comparison with the usage of the '*sales_time_date*' composite index (*Figure 5*).

```
--- EXPLAIN ANALYSE QUERY 2 WITH the albums table clustered by release date

                                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=2.03..3.83 rows=1 width=16) (actual time=0.918..0.929 rows=1 loops=1)
   ->  Group  (cost=2.03..11257.76 rows=6270 width=16) (actual time=0.912..0.920 rows=1 loops=1)
         Group Key: sales, band_id, release_date, running_time
         ->  Incremental Sort  (cost=2.03..11172.02 rows=8574 width=16) (actual time=0.909..0.914 rows=1 loops=1)
               Sort Key: sales DESC, band_id, release_date, running_time
               Presorted Key: sales
               Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 26kB  Peak Memory: 26kB
               ->  Index Scan Backward using sales_time_date on albums a  (cost=0.29..10862.08 rows=8574 width=16) (actual time=0.090..0.776 rows=36 loops=1)
                     Index Cond: ((running_time >= '45'::real) AND (release_date >= '2000-01-01'::date) AND (release_date <= '2010-12-31'::date))
 Planning Time: 7.104 ms
 Execution Time: 1.089 ms
(11 rows)
```

Figure 8 - *Output of query 2 plan* with the '*albums*' table clustered by '*release_date*' attribute.

3 - *MongoDB* Query Optimization

3.1 - Query 1

The querying optimization procedure behind the non-relational database *MongoDB* is similar to the relational database *PostgreSQL.* In *query 1* we started to analyse the execution plan.

Thereby, to obtain this detailed plan we should add in the '*db.albums.find()*' query function the statement '*.explain("executionStats")*' (*Appendix 3*), although we can´t get the desired output - the output describes that were examined '0' documents (*Appendix 4*). This anomaly may happen due to permission issues between the *MongoDB Compass* GUI and the *MongoDB* Shell (*mongosh*). To overcome this problem, we used the '*Explain Plan*' tab in the *MongoDB Compass,* which retrieved the set of outputs presented in *Figure 9.*

Figure 9 - Outputs of *query 1* execution plan without the usage of an index in *MongoDB Compass*.

Inferring the set of outputs described in *Figure 9*, we establish that the *query 1* has two stages in its execution plan. Starting by sorting in descending order the '*sales*' attribute. Then proceeds to do the collection scan ('*COLLSCAN*') considering the query constraints, where were examined '34088' documents and returned '1' document in '27 ms' of execution time.

Comparing with *PostgreSQL* results, when not using an index, it took less 121.778 ms (148.778 ms – 27 ms = 121.778 ms) of execution time.

3.1.1 - Creating indexes

Once again, we had problems with the MongoDB shell when trying to create the indexes, returning an error saying that it couldn't create the index due to not having permission. We overcame this anomaly creating the indexes using the tab '*Indexes*' in *MongoDB Compass*. However, we detailed below how the index should be created using a shell.

We evaluated various combinations of composite indexes and single indexes, considering the number of documents examined and returned in a certain execution time. MongoDB uses as default the *B-tree* index. Our considerations conclude that the index that best optimized the *query 1* was a composite index in the '*sales*' and '*release_date*' attributes (*'sales_date')* - *'db.albums.createIndex({"sales":-1},{"release_date":1},name: "sales_date")'*.

Thus, we inferred the outputs of *query 1* execution plan when using the '*sales_date*' composite index. The returned set of outputs were described in *Figure 10. Query 1* plan has two main stages which benefits with the use of the applied index, - (*"stage": "IXSCAN"), scanning for index keys* - starting by limiting (*"stage": "LIMIT"*) the number of documents who will be passed to the next stage (filtering) - without the usage of an index the *query 1* examined '34088'

documents and now, with the index implement, only examines ‘1926’. Then in the next stage, ‘*FETCH*’, it retrieved ‘1’ document. The implemented composite index ‘*sales_date*’ allowed to reduce the execution time of the *query 1* in 22 ms (27 ms - 5 ms = 22 ms).

Comparing with the results in *PostgreSQL*, when using the same composite index, it took less 179.369 ms of execution time (184.374 ms – 5 ms = 179.369 ms).

Figure 10 - Outputs of *quey 1* execution plan using the composite index *‘sales_date’* in *MongoDB Compass*.

3.2 - Query 2

For the *query 2* we followed the same steps described in the *query 1* optimization section.

We started by performing (*Appendix 5*) and analyzing the *query 2* execution plan outputs (*Figure 11*) without using an index. Like *query 1* the *query 2* also has two stages in its execution plan. Starting by sorting in descending order the ‘*sales*’ attribute. Then proceeds to do the collection scan (‘*COLLSCAN*’) considering the query constraints, where were examined ‘34088’ documents and returned ‘1’ document in ‘28 ms’ of execution time. Which compared with the obtained results in *PostgresSQL* was 77.984 ms fastest (105.984 ms – 28 ms = 77.984 ms).

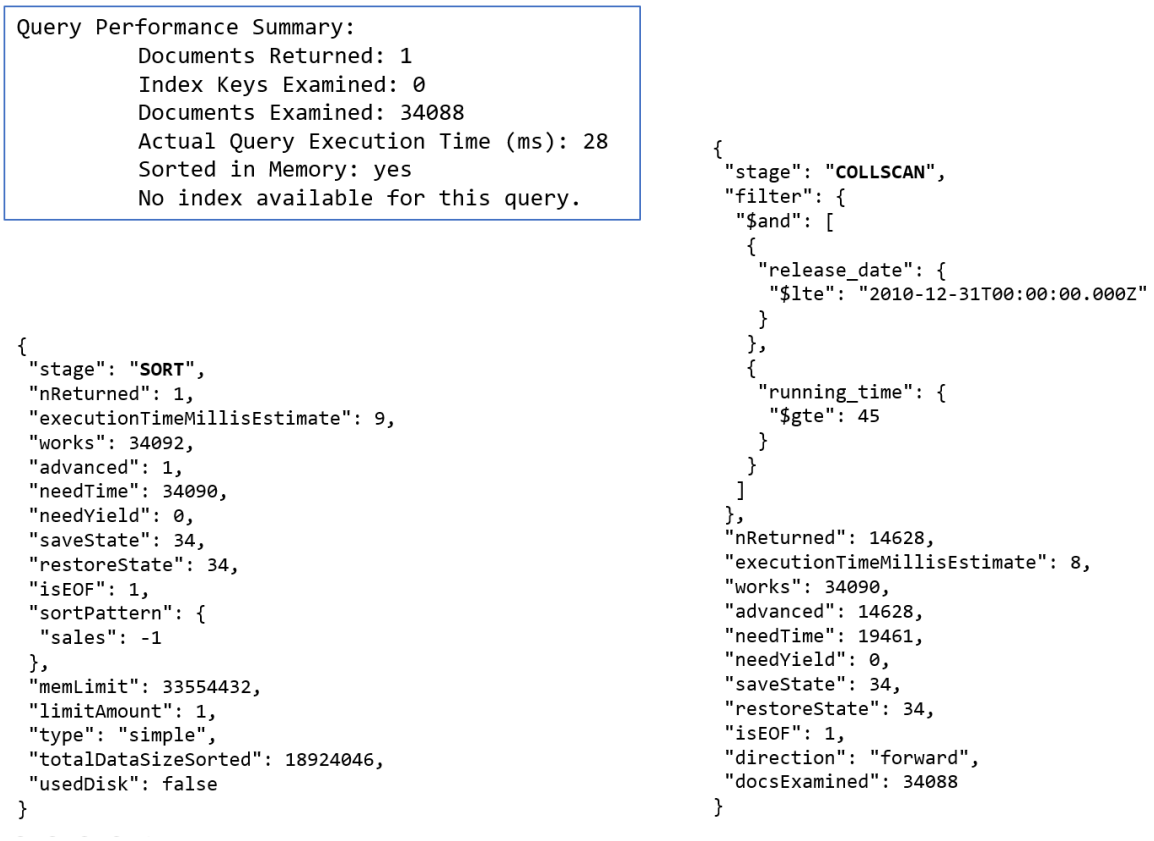

Figure 11 - Outputs of *quey 2* execution plan without the usage of an index in *MongoDB Compass*.

3.2.1 - Creating indexes

To create the index to optimize the *query 2*, once again, we followed the same steps described in the *query* 1 index creation section (*3.1.1*).

The index which was shown to best optimize the *query 2* was a composite index for the "*sales*", "*release_date*" and "*sales_date_time"* attributes (*'sales_date_time')* – *'db.albums.createIndex({"sales":-1},{"release_date":1},{"running_time":1}, name: "sales_date_time") '.*

Inferring the outputs of the *query 2* execution plan when using the *'sales_date_time'* index (*Figure 12*), it has two main stages '*LIMIT*' and '*FETCH*' as *query 1*. The query examined and returned only one document with '1 ms' of execution time. This means that the index managed to reduce 27 ms (28 ms - 1 ms = 27 ms) in the querying execution time.

Compared with *PosgreSQL* obtained results, that uses the same composite index, *NoSQL* performed the *query 2* slightly with the same execution time.

Figure 12 - Outputs of *quey 2* execution plan using the composite index *'sales_date'* in *MongoDB Compass*.

4. Discussion

By running each query on both *NoSQL* and *RDB* databases we reached the conclusion that *MongoDB* is relatively faster than a *PostgreSQL*, the main reason for this is the fact that we structured our *NoSQL* so the need to pass through many aggregates was nonexistent, unlike our *SQL* schema where it is necessary to do three table joins to obtain the desired result ('*genre_name'* attribute) for the query 1. *RDB* had a maximum execution time of 84.3 ms, while *NoSQL* had a maximum of 27 ms to execute our queries.

On a similar note, the *RDB* could have some alterations, for example denormalization of our *SQL* schema to obtain a similar behavior as our *NoSQL* schema, as we mentioned before. Also, the addition of a new relation with the release decade of an album, instead of using the regular *release_date* attribute, while on the *NoSQL* we saw no need for alterations on the model nor any relevant alterations that could be done, which adds another point in favor of the *NoSQL*.

In *PostgreSQL,* the results when clustering the '*albums*' table using the single index showed slightly the same execution times but fastest planning times, which could indicate that a single index (*'date'*) was a best solution to optimize our queries in *RDB*.

In conclusion, for our queries and the level of optimization as it is, a *NoSQL* approach would work better to obtain the results of our queries faster.

5. References

[1] MongoDB. *MongoDB Documentation.* URL: https://docs.mongodb.com/

[2] PostgreSQL. *Documentation.* URL: https://postgresql.org/docs/

[3] Raghu Ramakrishnan and Johannes Gehrke - *3ºedition*. *Database Management Systems*. McGraw-Hill, 2003. ISBN: 0072465638.

[4] M. Sadalage P. J. Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence.* Pearson Education, 2013. ISBN: 0072465638.

6 - Appendices

```
EXPLAIN ANALYZE SELECT albums.band_id, albums.release_date, albums.sales, genres.genre_name, albums.abstract
FROM (((albums
INNER JOIN bands ON albums.band_id = bands.band_id)
INNER JOIN bands_genre ON bands.band_id = bands_genre.band_id)
INNER JOIN genres ON bands_genre.genre_id = genres.genre_id)
        WHERE genres.genre_name = 'Math rock'
        AND albums.release_date >= '1990/01/01'
        AND albums.release_date <= '1999/12/31'
        AND LENGTH(albums.abstract) > 200
        GROUP BY albums.band_id, albums.release_date, albums.abstract, albums.sales, genres.genre_name
        ORDER BY albums.sales
        DESC
        LIMIT 1;
```

Appendix 1 - *Query 1* execution plan in *PostgreSQL.*

```
EXPLAIN ANALYSE SELECT A.band_id, A.sales, A.release_date, A.running_time
FROM albums AS A
    WHERE A.running_time >= '45'
    AND A.release_date >= '2000/01/01'
    AND A.release_date <= '2010/12/31'
    GROUP BY  A.band_id,A.sales, A.release_date, A.running_time
    ORDER BY A.sales
    DESC
    LIMIT 1;
```

Appendix 2 - *Query 2* execution plan in *PostgreSQL.*

```
db.albums.find({
            $and :
                                [
                                {genres: "Math rock"},
                                 {release_date: {$gte: ISODate("1990-01-01T00:00:00.000+00:00")}},
                                 {release_date: {$lte: ISODate("1999-12-31T00:00:00.000+00:00")}},
                                 {abstract: /^[\s\S]{200,}$/ }
                                 ]
                        }).sort({sales: -1}).limit(1).explain("executionStats")
```

Appendix 3 - *Query 1* execution plan in *MongoDB.*

```
{ queryPlanner:
   { plannerVersion: 1,
     namespace: 'test.albums',
     indexFilterSet: false,
     parsedQuery:
      { '$and':
         [ { genres: { '$eq': 'Math rock' } },
           { release_date: { '$lte': 1999-12-31T00:00:00.000Z } },
           { release_date: { '$gte': 1990-01-01T00:00:00.000Z } },
           { abstract: BSONRegExp("^[\\s\\S]{200,}$", "") } ] },
     winningPlan: { stage: 'EOF' },
     rejectedPlans: [] },
  executionStats:
   { executionSuccess: true,
     nReturned: 0,
     executionTimeMillis: 0,
     totalKeysExamined: 0,
     totalDocsExamined: 0,
     executionStages:
      { stage: 'EOF',
        nReturned: 0,
        executionTimeMillisEstimate: 0,
        works: 1,
        advanced: 0,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1 } },
  serverInfo:
   { host: 'bandscluster-shard-00-02.1eit9.mongodb.net',
     port: 27017,
     version: '4.4.10',
     gitVersion: '58971da1ef93435a9f62bf4708a81713def6e88c' },
  ok: 1,
  '$clusterTime':
   { clusterTime: Timestamp({ t: 1638893065, i: 1 }),
     signature:
      { hash: Binary(Buffer.from("b3d25691b040bfacdab4258e5d2263d5d2ab514b", "hex"), 0),
        keyId: 6986443594676568000 } },
  operationTime: Timestamp({ t: 1638893065, i: 1 }) }
```

Appendix 4 - Output of *Query 1* execution plan in *MongoDB.*

```
db.albums.find({
                $and :
                                [
                                 {running_time: {$gte: 45}},
                                 {release_date: {$gte: ISODate("2000-01-01T00:00:00.000+00:00")}},
                                 {release_date: {$lte: ISODate("2010-12-31T00:00:00.000+00:00")}}
                                ]
                        }).sort({sales: -1}).limit(1).explain("executionStats")
```

Appendix 5 - *Query 2* execution plan in *MongoDB.*