Project Part 3

Done by: Group 18

- Pedro Santos 54129
- Rafael Oliveira 52848
- Rodrigo Serra 57414
- Rui Roque 57588

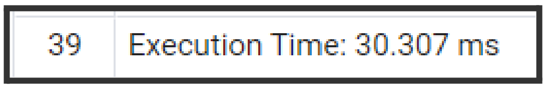
1. Introduction

This final part of the project focuses on the performance of queries to our database, both in SQL and MongoDB. Our two queries were rewritten in order to optimize their running times, and the use of indexing was also part of this approach. In both PostgreSQL and MongoDB only one index was needed. Significant differences were obtained in their running times, when compared to the way they were written before.

2. Relational Query Optimization (PostgreSQL)

2.1 First Query

Before any changes, the first query was run with the command explain analyze to see its execution time, which in this case was 30.307ms.

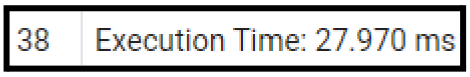


```
39   Execution Time: 30.307 ms
```

Figure 1: Run Time Without Optimization Of Query 1

2.1.1 Query optimization

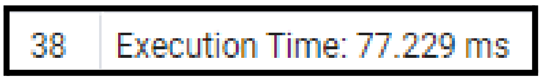
The first step we did to optimize the query running time, is using indexes since it is a way of making the searches that the query does faster since an index stores where each part of the data is in the disk. It can also make the order of the files in data level be ordered according to an index. In this case, looking into our queries we decided the best index to use was the release date, since the query in question had an interval based condition and no other column was feasible. We could have clustered by sales_amount but we went with a different route as we will explain below. We also tried to cluster our index since it was an interval based condition, we thought it would be faster.



```
38   Execution Time: 27.970 ms
```

Figure 2: Time of Query 1 Without Clustering Of Release Date Index

By testing the usage of a clustered index in the query the results are worse performance wise, since the query itself uses a lot of other columns and the clustering of the release date index makes the query planner go for a sequential scan.



```
38   Execution Time: 77.229 ms
```

Figure 3: Time of Query 1 With Clustering Of Release Date Index

In order to optimize the first query which was created and used for the other parts, a new table was created that contains for each existing genre in the last decade the total of sales amount.

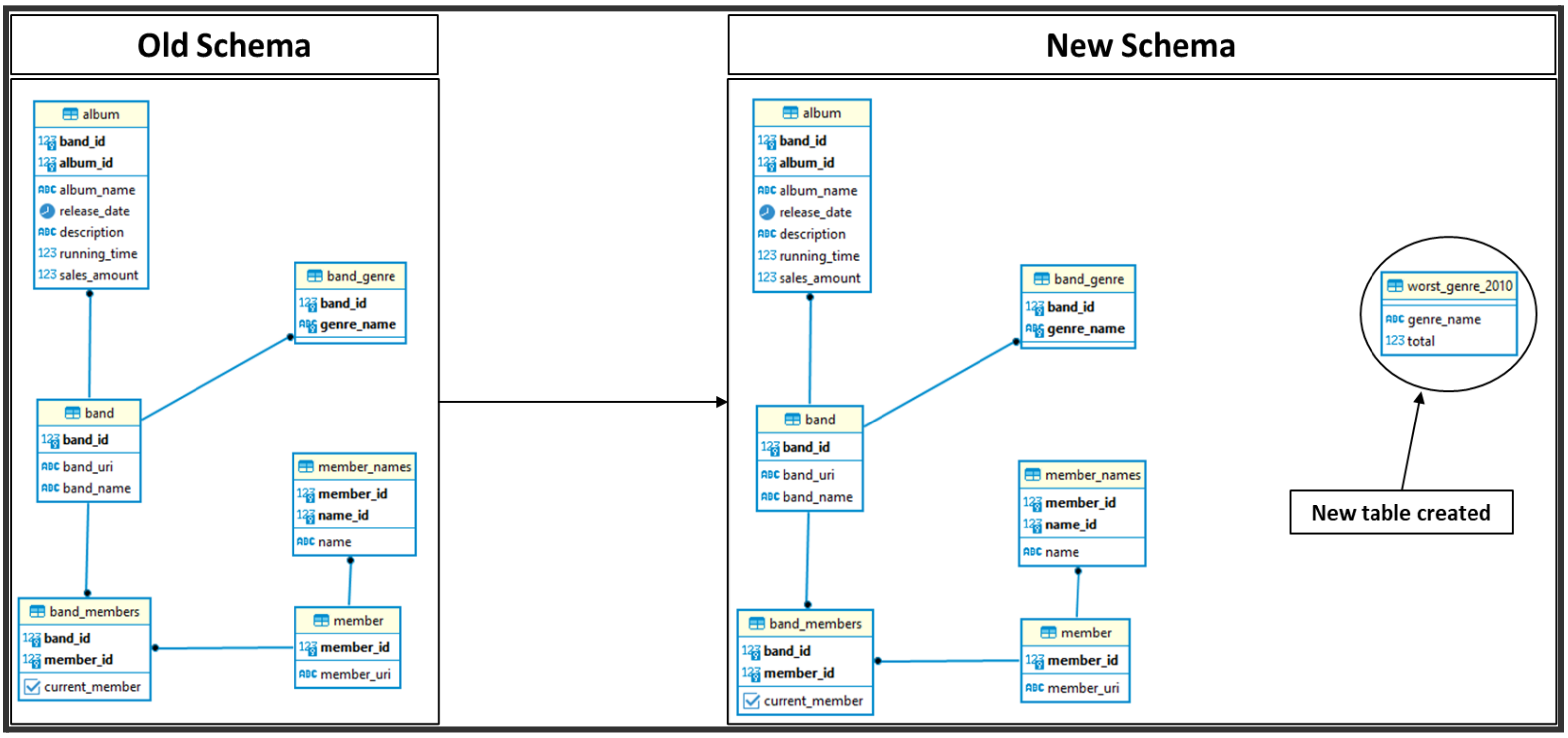
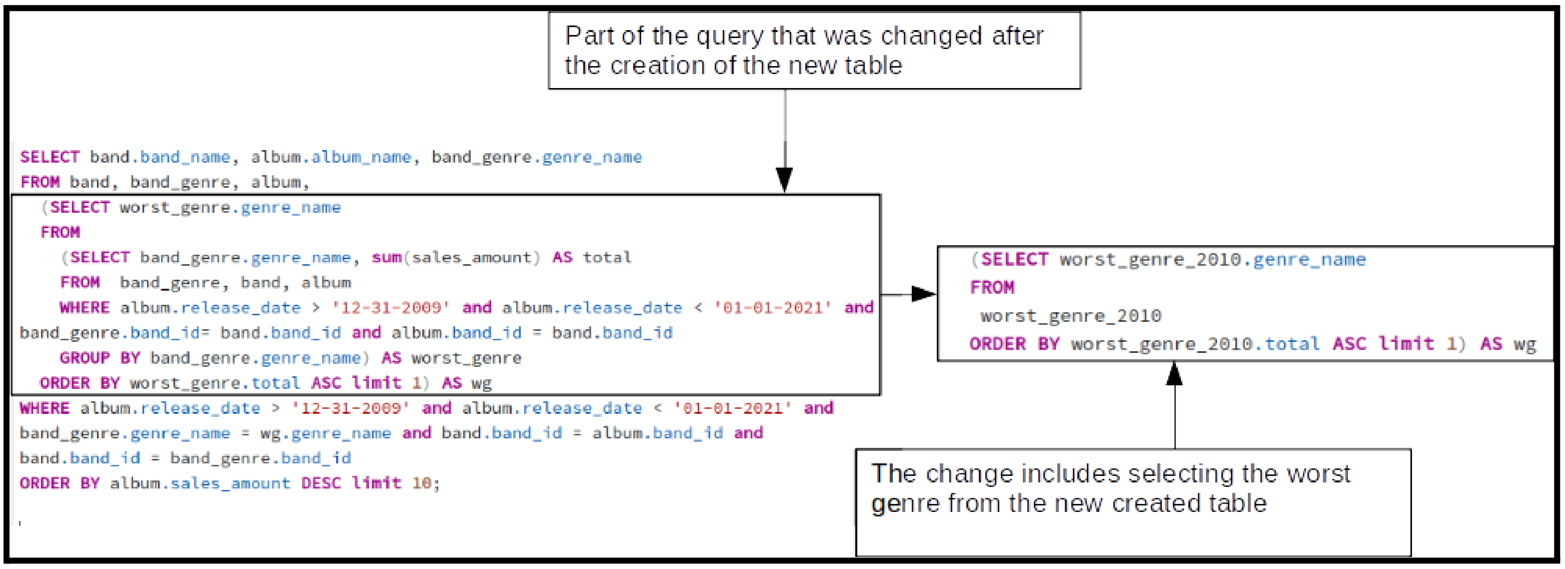


Figure 4: New Table In The New Schema

With this table created, a bigger reduction in execution time was achieved compared to indexing by sales amount, since each time the sales amount of a specific genre was requested the operation did not have to sum up all of the album sales amounts repeatedly, and it was necessary to rewrite the query. It came with the trade off of having to constantly update it if album data is changed, and therefore the worst genre is different.



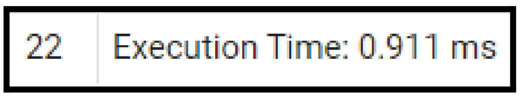
Part of the query that was changed after the creation of the new table

```
SELECT band.band_name, album.album_name, band_genre.genre_name
FROM band, band_genre, album,
  (SELECT worst_genre.genre_name
  FROM
    (SELECT band_genre.genre_name, sum(sales_amount) AS total
    FROM  band_genre, band, album
    WHERE album.release_date > '12-31-2009' and album.release_date < '01-01-2021' and
band_genre.band_id= band.band_id and album.band_id = band.band_id
    GROUP BY band_genre.genre_name) AS worst_genre
  ORDER BY worst_genre.total ASC limit 1) AS wg
WHERE album.release_date > '12-31-2009' and album.release_date < '01-01-2021' and
band_genre.genre_name = wg.genre_name and band.band_id = album.band_id and
band.band_id = band_genre.band_id
ORDER BY album.sales_amount DESC limit 10;
```

```
(SELECT worst_genre_2010.genre_name
FROM
 worst_genre_2010
ORDER BY worst_genre_2010.total ASC limit 1) AS wg
```

The change includes selecting the worst genre from the new created table

Figure 5: Changes To Query 1



```
22    Execution Time: 0.911 ms
```

Figure 6: Run Time With All The Optimizations Made To Query 1

2.2 Second Query

Since the second query used the first one inside of it, it was also heavily influenced by the optimization of the first query explained above, but also has been impacted by the addition of another new table that has all of the top 10 albums of the worst genre previously calculated shown below.

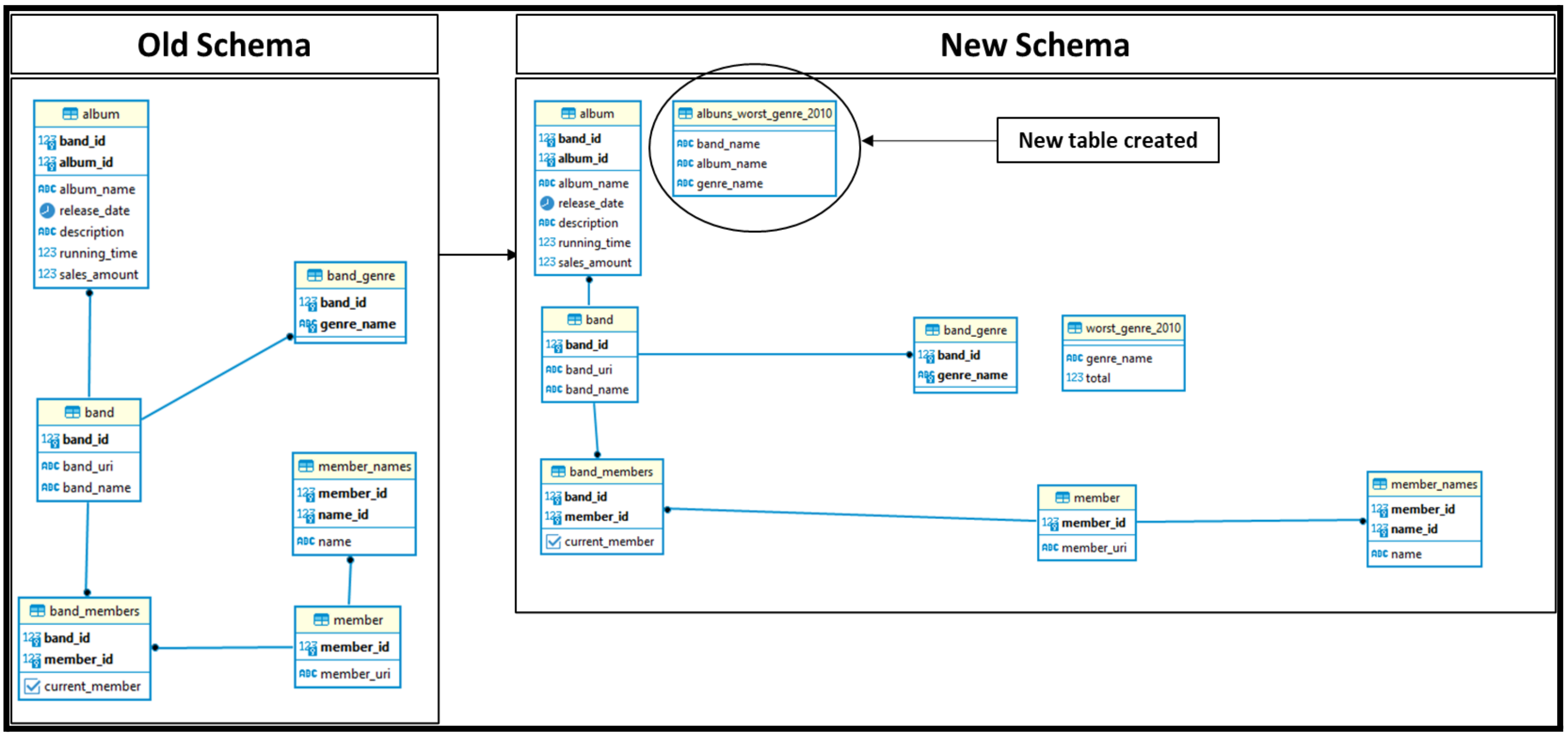
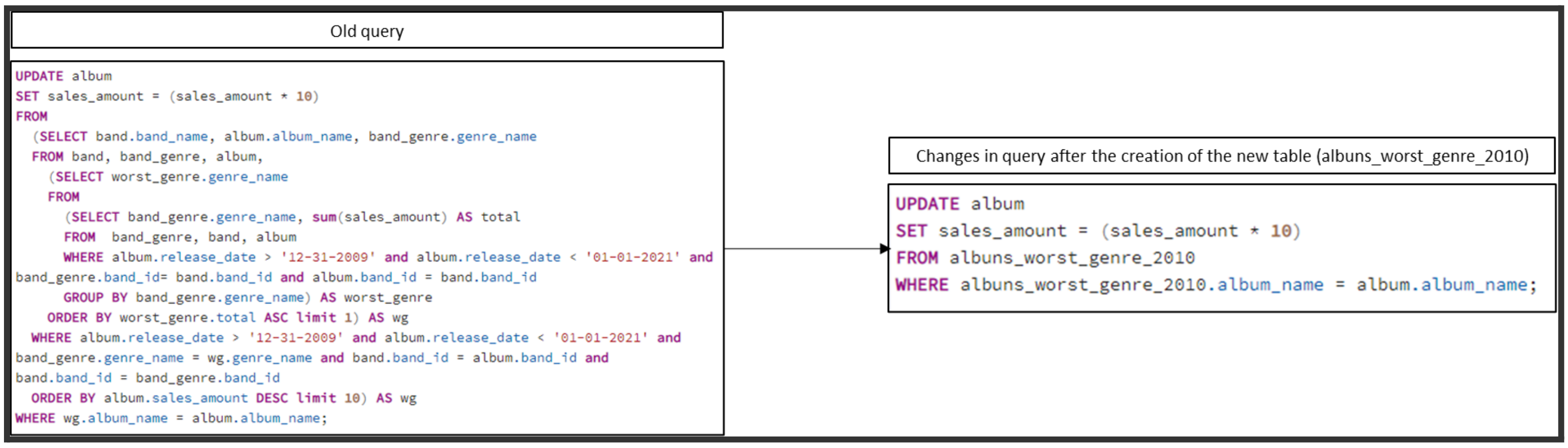


Figure 7: Second New Table In The New Schema

That change also permitted that the query itself could be severely simplified (shown below), making it easier for the query planner to find the best plan. It came with the trade off of having to constantly update it if album data is changed and therefore the worst genre is different, meaning that the albums of the worst genre could change as well.



Old query

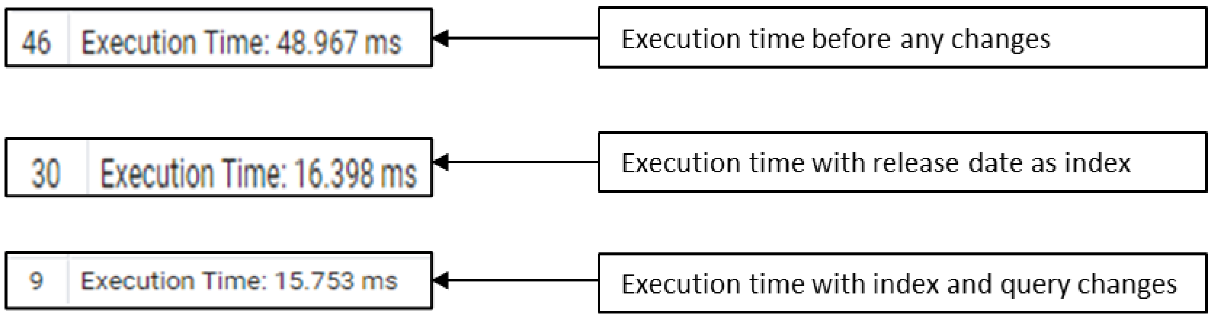
```
UPDATE album
SET sales_amount = (sales_amount * 10)
FROM
  (SELECT band.band_name, album.album_name, band_genre.genre_name
  FROM band, band_genre, album,
    (SELECT worst_genre.genre_name
    FROM
      (SELECT band_genre.genre_name, sum(sales_amount) AS total
      FROM  band_genre, band, album
      WHERE album.release_date > '12-31-2009' and album.release_date < '01-01-2021' and
band_genre.band_id= band.band_id and album.band_id = band.band_id
      GROUP BY band_genre.genre_name) AS worst_genre
    ORDER BY worst_genre.total ASC limit 1) AS wg
  WHERE album.release_date > '12-31-2009' and album.release_date < '01-01-2021' and
band_genre.genre_name = wg.genre_name and band.band_id = album.band_id and
band.band_id = band_genre.band_id
  ORDER BY album.sales_amount DESC limit 10) AS wg
WHERE wg.album_name = album.album_name;
```

Changes in query after the creation of the new table (albuns_worst_genre_2010)

```
UPDATE album
SET sales_amount = (sales_amount * 10)
FROM albuns_worst_genre_2010
WHERE albuns_worst_genre_2010.album_name = album.album_name;
```

Figure 8: Changes To Query 2

The results of each step are shown below in terms of time to run. Here there was no clustering attempt, since the places where a clustered index could be beneficial, have been shown by query 1 to not be the case.



```
46  Execution Time: 48.967 ms
```
Execution time before any changes

```
30  Execution Time: 16.398 ms
```
Execution time with release date as index

```
9  Execution Time: 15.753 ms
```
Execution time with index and query changes

Figure 9: Execution Times For Query 2

2.4 Conclusion

The database used had a SSD drive and therefore the time gains by usage of the index were not as noticeable compared to if our database had a HDD, since the HDD has a moving arm, physical disks and several rings inside those disks, compared to the chips used in the SSD and therefore position of the data and knowing where it is more significant. With that said we could still see a big improvement especially in the first query where that improvement was of an order of magnitude.

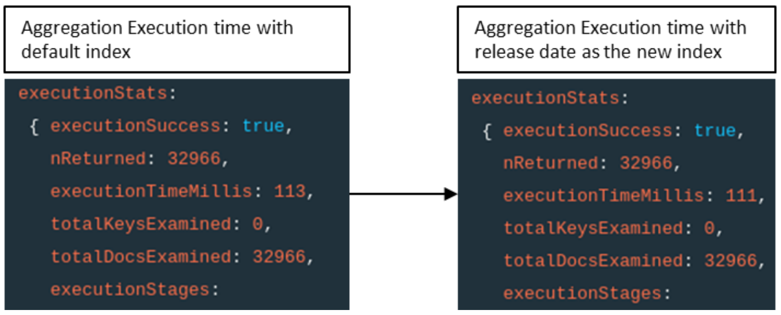
3. Mongodb Optimization

No alterations were made to the data model in this part, since there were no better ways of doing this more efficiently, after some ideas were tried. However, we used the release date as an index for the same reasons as in the Postgres implementation and it yielded some good results, as shown below.

3.1 Aggregation

Before any changes, the aggregation was run with the command .explain("executionstats") to see its execution time, which in this case was 113 ms.

After the use of the index it was noticed that there was a 2 millisecond decrease in run time, since the query planner used the index, that was faster to get the 2010 albums instead of a full sequential scan.


Aggregation Execution time with default index

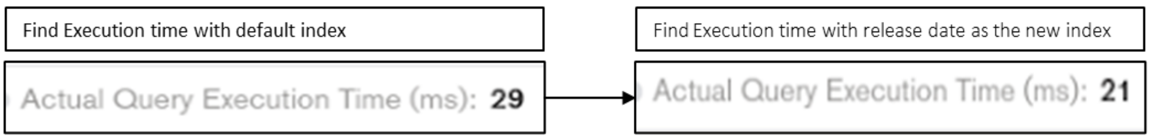
```
executionStats:
 { executionSuccess: true,
   nReturned: 32966,
   executionTimeMillis: 113,
   totalKeysExamined: 0,
   totalDocsExamined: 32966,
   executionStages:
```
Aggregation Execution time with release date as the new index

```
executionStats:
 { executionSuccess: true,
   nReturned: 32966,
   executionTimeMillis: 111,
   totalKeysExamined: 0,
   totalDocsExamined: 32966,
   executionStages:
```

Figure 10: Aggregation Execution Times With And Without New Index

3.2 First Query

After the use of the index it was noticed that there was a 8 millisecond decrease in run time per each find, since the query planner used the index, that was faster to get the 2010 albums instead of a full sequential scan. Since this find expression is only run for the worst genre in our original query, it will only run once. With that said the whole gain in the first query will be the 2 milliseconds of the aggregation plus the 8 milliseconds gained in the sole find that is run, making the total gain of the query 10 milliseconds.


Find Execution time with default index

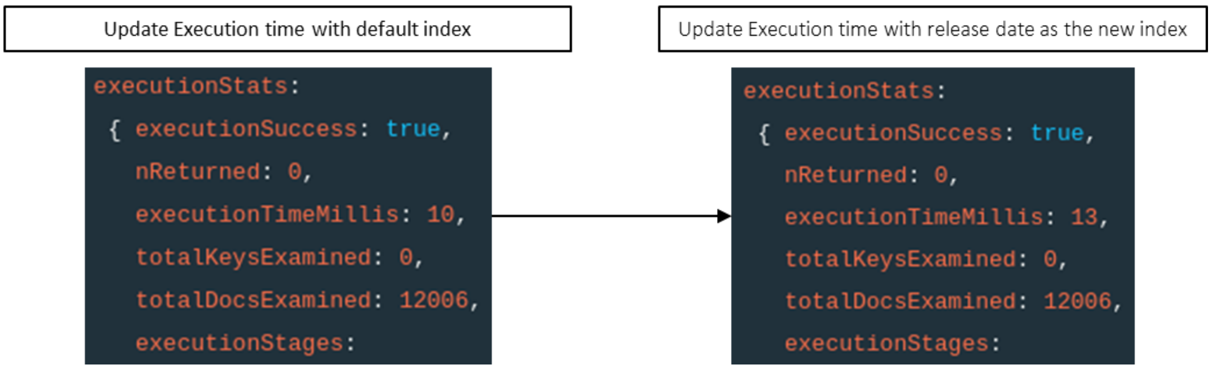
Actual Query Execution Time (ms): **29**

Find Execution time with release date as the new index

Actual Query Execution Time (ms): **21**

Figure 11: Find Expression Execution Times With and Without New Index

3.3 Second Query


Update Execution time with default index

```
executionStats:
 { executionSuccess: true,
   nReturned: 0,
   executionTimeMillis: 10,
   totalKeysExamined: 0,
   totalDocsExamined: 12006,
   executionStages:
```

Update Execution time with release date as the new index

```
executionStats:
 { executionSuccess: true,
   nReturned: 0,
   executionTimeMillis: 13,
   totalKeysExamined: 0,
   totalDocsExamined: 12006,
   executionStages:
```

Figure 12: Update Expression Execution Times of The Update Statement With and Without Index Usage

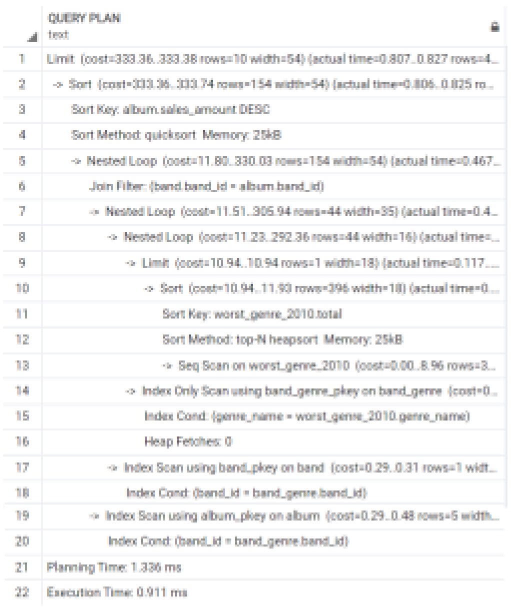
After the implementation of the index it was noticed that the query planner never used our index, since the update part of the query never uses the release date. This update expression is run for the 10 best albums of the worst genre calculated before (if there are that many). With the previous in mind the update expression could run 1 to 10 times, depending on the number of albums existent within that genre in that decade. With that said, the second query will consist of the gain of the 2 milliseconds gained in the aggregation. So the gain that could be generated on the second query will be of 2 milliseconds. The changes seen in the picture above are only related to latency/cpu usage, therefore not counting for the calculations made for the gain/loss of time by usage of the index.

3.4 Conclusion

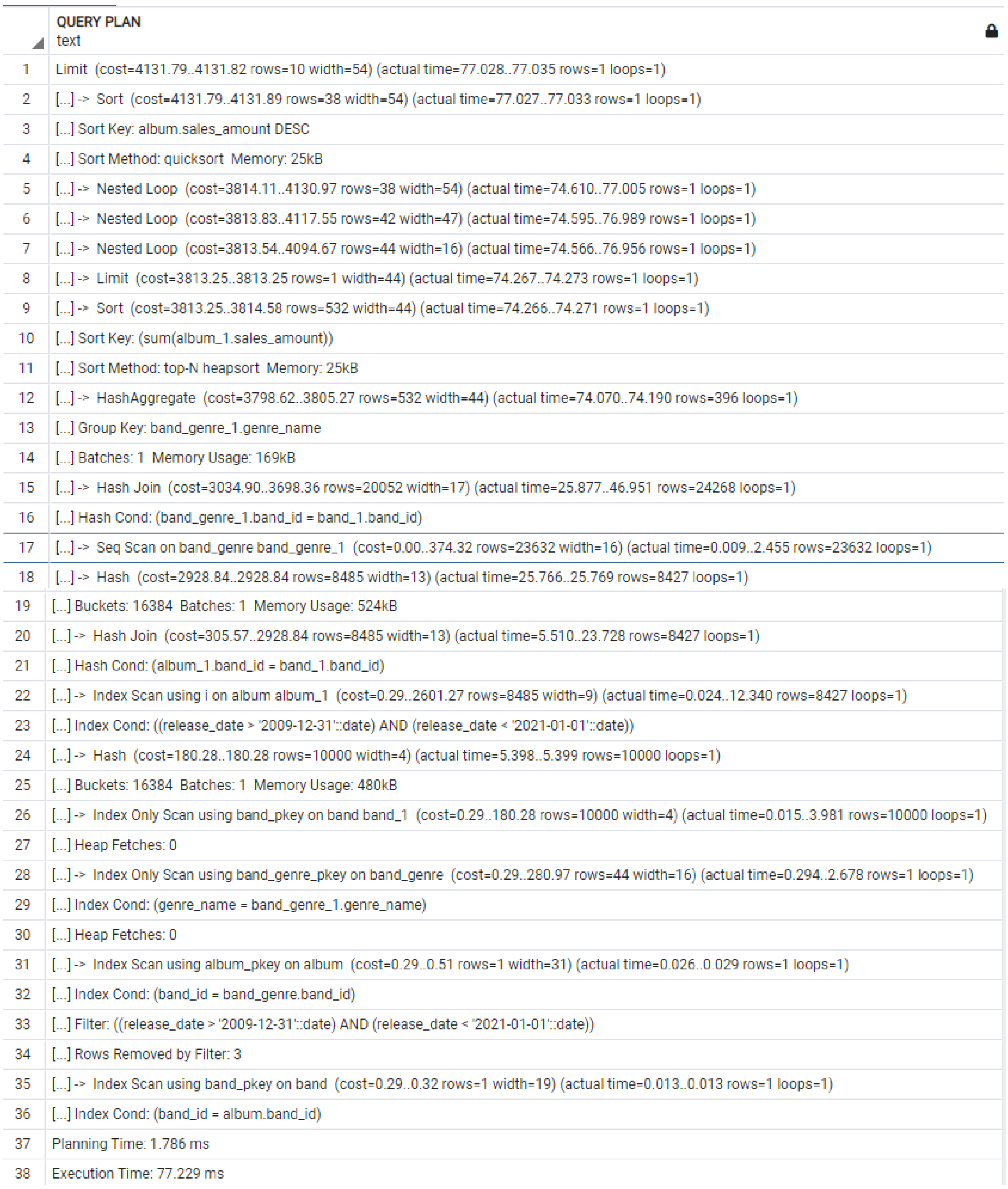
There is a gain on the first query find expression of 8 milliseconds and on the aggregation part of both queries of 2 milliseconds.With that said it makes sense then to use the release date index to better performance of the query. The database used had a SSD drive as the Postgres implementation, therefore the time gains by usage of the index were not as noticeable compared to if our database had a HDD, since the HDD has a moving arm, physical disks and several rings inside those disks, compared to the chips used in the SSD and therefore position of the data and knowing where it is more significant.

Anex

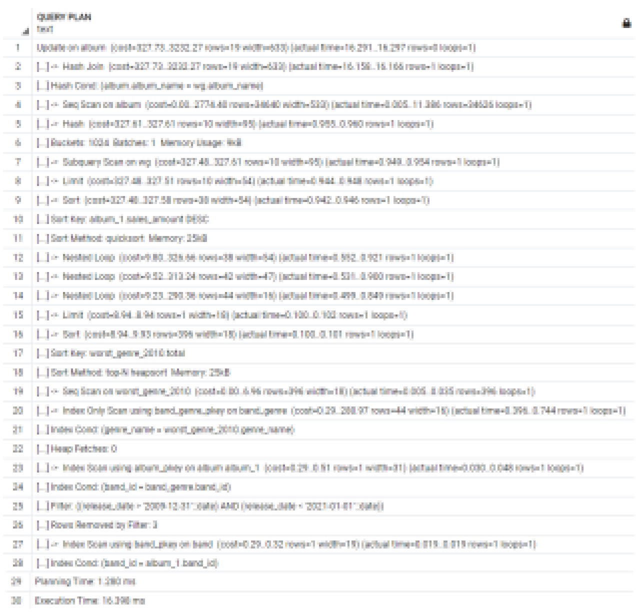
Query plans
SQL



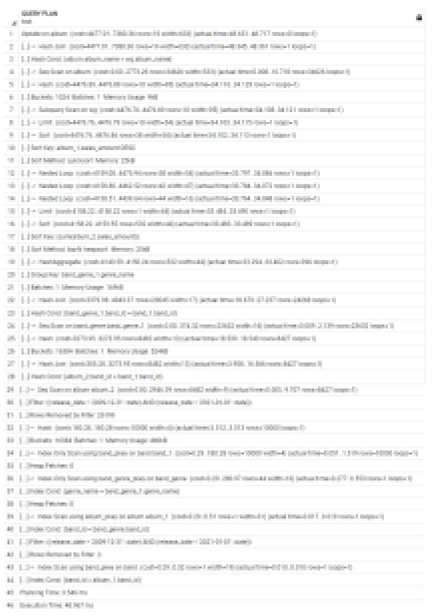
```
QUERY PLAN
text
Limit  (cost=333.36..333.38 rows=10 width=54) (actual time=0.807..0.827 rows=4…
  -> Sort  (cost=333.36..333.74 rows=154 width=54) (actual time=0.806..0.825 ro…
        Sort Key: album.sales_amount DESC
        Sort Method: quicksort  Memory: 25kB
        -> Nested Loop  (cost=11.80..330.03 rows=154 width=54) (actual time=0.467…
              Join Filter: (band.band_id = album.band_id)
              -> Nested Loop  (cost=11.51..305.94 rows=44 width=35) (actual time=0.4…
                    -> Nested Loop  (cost=11.23..292.36 rows=44 width=16) (actual time=…
                          -> Limit  (cost=10.94..10.94 rows=1 width=18) (actual time=0.117.…
                                -> Sort  (cost=10.94..11.93 rows=396 width=18) (actual time=0…
                                      Sort Key: worst_genre_2010.total
                                      Sort Method: top-N heapsort  Memory: 25kB
                                      -> Seq Scan on worst_genre_2010  (cost=0.00..8.96 rows=3…
                          -> Index Only Scan using band_genre_pkey on band_genre  (cost=0…
                                Index Cond: (genre_name = worst_genre_2010.genre_name)
                                Heap Fetches: 0
                    -> Index Scan using band_pkey on band  (cost=0.29..0.31 rows=1 widt…
                          Index Cond: (band_id = band_genre.band_id)
              -> Index Scan using album_pkey on album  (cost=0.29..0.48 rows=5 width…
                    Index Cond: (band_id = band_genre.band_id)
Planning Time: 1.336 ms
Execution Time: 0.911 ms
```



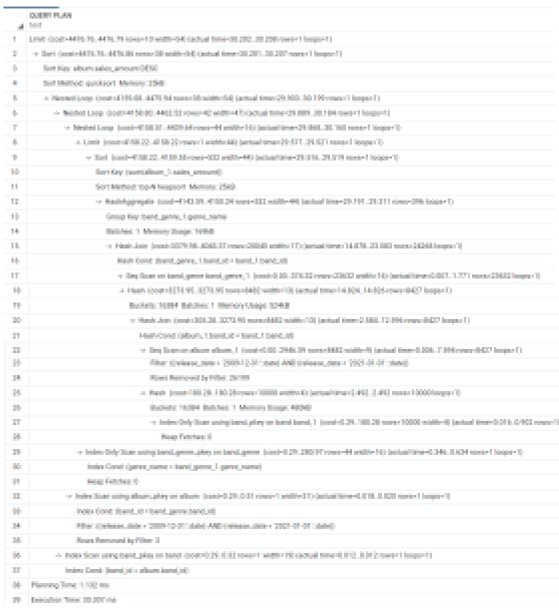
```
QUERY PLAN
text
1   Limit  (cost=4131.79..4131.82 rows=10 width=54) (actual time=77.028..77.035 rows=1 loops=1)
2   [...] -> Sort  (cost=4131.79..4131.89 rows=38 width=54) (actual time=77.027..77.033 rows=1 loops=1)
3   [...] Sort Key: album.sales_amount DESC
4   [...] Sort Method: quicksort  Memory: 25kB
5   [...] -> Nested Loop  (cost=3814.11..4130.97 rows=38 width=54) (actual time=74.610..77.005 rows=1 loops=1)
6   [...] -> Nested Loop  (cost=3813.83..4117.55 rows=42 width=47) (actual time=74.595..76.989 rows=1 loops=1)
7   [...] -> Nested Loop  (cost=3813.54..4094.67 rows=44 width=16) (actual time=74.566..76.956 rows=1 loops=1)
8   [...] -> Limit  (cost=3813.25..3813.25 rows=1 width=44) (actual time=74.267..74.273 rows=1 loops=1)
9   [...] -> Sort  (cost=3813.25..3814.58 rows=532 width=44) (actual time=74.266..74.271 rows=1 loops=1)
10  [...] Sort Key: (sum(album_1.sales_amount))
11  [...] Sort Method: top-N heapsort  Memory: 25kB
12  [...] -> HashAggregate  (cost=3798.62..3805.27 rows=532 width=44) (actual time=74.070..74.190 rows=396 loops=1)
13  [...] Group Key: band_genre_1.genre_name
14  [...] Batches: 1  Memory Usage: 169kB
15  [...] -> Hash Join  (cost=3034.90..3698.36 rows=20052 width=17) (actual time=25.877..46.951 rows=24268 loops=1)
16  [...] Hash Cond: (band_genre_1.band_id = band_1.band_id)
17  [...] -> Seq Scan on band_genre band_genre_1  (cost=0.00..374.32 rows=23632 width=16) (actual time=0.009..2.455 rows=23632 loops=1)
18  [...] -> Hash  (cost=2928.84..2928.84 rows=8485 width=13) (actual time=25.766..25.769 rows=8427 loops=1)
19  [...] Buckets: 16384  Batches: 1  Memory Usage: 524kB
20  [...] -> Hash Join  (cost=305.57..2928.84 rows=8485 width=13) (actual time=5.510..23.728 rows=8427 loops=1)
21  [...] Hash Cond: (album_1.band_id = band_1.band_id)
22  [...] -> Index Scan using i on album album_1  (cost=0.29..2601.27 rows=8485 width=9) (actual time=0.024..12.340 rows=8427 loops=1)
23  [...] Index Cond: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
24  [...] -> Hash  (cost=180.28..180.28 rows=10000 width=4) (actual time=5.398..5.399 rows=10000 loops=1)
25  [...] Buckets: 16384  Batches: 1  Memory Usage: 480kB
26  [...] -> Index Only Scan using band_pkey on band band_1  (cost=0.29..180.28 rows=10000 width=4) (actual time=0.015..3.981 rows=10000 loops=1)
27  [...] Heap Fetches: 0
28  [...] -> Index Only Scan using band_genre_pkey on band_genre  (cost=0.29..280.97 rows=44 width=16) (actual time=0.294..2.678 rows=1 loops=1)
29  [...] Index Cond: (genre_name = band_genre_1.genre_name)
30  [...] Heap Fetches: 0
31  [...] -> Index Scan using album_pkey on album  (cost=0.29..0.51 rows=1 width=31) (actual time=0.026..0.029 rows=1 loops=1)
32  [...] Index Cond: (band_id = band_genre.band_id)
33  [...] Filter: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
34  [...] Rows Removed by Filter: 3
35  [...] -> Index Scan using band_pkey on band  (cost=0.29..0.32 rows=1 width=19) (actual time=0.013..0.013 rows=1 loops=1)
36  [...] Index Cond: (band_id = album.band_id)
37  Planning Time: 1.786 ms
38  Execution Time: 77.229 ms
```



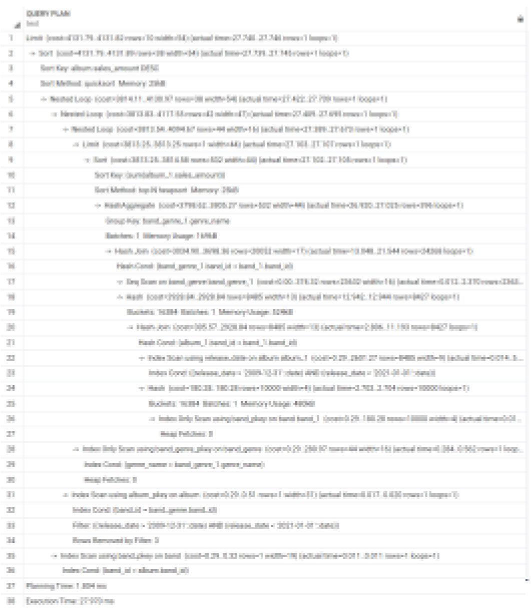
```
QUERY PLAN
text
Update on album  (cost=327.73..3232.27 rows=19 width=533) (actual time=16.291..16.297 rows=0 loops=1)
[...] -> Hash Join  (cost=327.73..3232.27 rows=19 width=533) (actual time=16.158..16.166 rows=1 loops=1)
[...] Hash Cond: (album.album_name = wg.album_name)
[...] -> Seq Scan on album  (cost=0.00..2774.40 rows=34640 width=533) (actual time=0.005..11.386 rows=34626 loops=1)
[...] -> Hash  (cost=327.61..327.61 rows=10 width=95) (actual time=0.955..0.960 rows=1 loops=1)
[...] Buckets: 1024  Batches: 1  Memory Usage: 9kB
[...] -> Subquery Scan on wg  (cost=327.48..327.61 rows=10 width=95) (actual time=0.949..0.954 rows=1 loops=1)
[...] -> Limit  (cost=327.48..327.51 rows=10 width=54) (actual time=0.944..0.948 rows=1 loops=1)
[...] -> Sort  (cost=327.48..327.58 rows=38 width=54) (actual time=0.942..0.946 rows=1 loops=1)
[...] Sort Key: album_1.sales_amount DESC
[...] Sort Method: quicksort  Memory: 25kB
[...] -> Nested Loop  (cost=9.80..326.66 rows=38 width=54) (actual time=0.552..0.921 rows=1 loops=1)
[...] -> Nested Loop  (cost=9.52..313.24 rows=42 width=47) (actual time=0.531..0.900 rows=1 loops=1)
[...] -> Nested Loop  (cost=9.23..290.36 rows=44 width=16) (actual time=0.499..0.849 rows=1 loops=1)
[...] -> Limit  (cost=8.94..8.94 rows=1 width=18) (actual time=0.100..0.102 rows=1 loops=1)
[...] -> Sort  (cost=8.94..9.93 rows=396 width=18) (actual time=0.100..0.101 rows=1 loops=1)
[...] Sort Key: worst_genre_2010.total
[...] Sort Method: top-N heapsort  Memory: 25kB
[...] -> Seq Scan on worst_genre_2010  (cost=0.00..6.96 rows=396 width=18) (actual time=0.005..0.035 rows=396 loops=1)
[...] -> Index Only Scan using band_genre_pkey on band_genre  (cost=0.29..288.97 rows=44 width=16) (actual time=0.396..0.744 rows=1 loops=1)
[...] Index Cond: (genre_name = worst_genre_2010.genre_name)
[...] Heap Fetches: 0
[...] -> Index Scan using album_pkey on album album_1  (cost=0.29..0.51 rows=1 width=31) (actual time=0.030..0.048 rows=1 loops=1)
[...] Index Cond: (band_id = band_genre.band_id)
[...] Filter: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
[...] Rows Removed by Filter: 3
[...] -> Index Scan using band_pkey on band  (cost=0.29..0.32 rows=1 width=19) (actual time=0.019..0.019 rows=1 loops=1)
[...] Index Cond: (band_id = album_1.band_id)
Planning Time: 1.280 ms
Execution Time: 16.398 ms
```



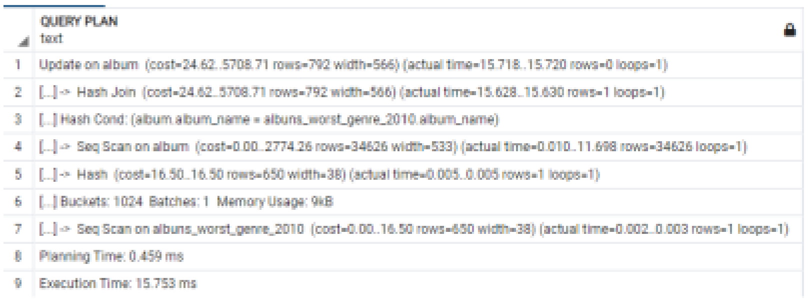
```
QUERY PLAN
text
Update on album (cost=4477.01..7380.36 rows=19 width=533) (actual time=48.651..48.717 rows=0 loops=1)
[...] -> Hash Join (cost=4477.01..7380.36 rows=19 width=533) (actual time=48.545..48.551 rows=1 loops=1)
[...] Hash Cond: (album.album_name = wg.album_name)
[...] -> Seq Scan on album (cost=0.00..2773.26 rows=34626 width=533) (actual time=0.008..10.716 rows=34626 loops=1)
[...] -> Hash (cost=4476.89..4476.89 rows=10 width=95) (actual time=34.114..34.123 rows=1 loops=1)
[...] Buckets: 1024 Batches: 1 Memory Usage: 9kB
[...] -> Subquery Scan on wg (cost=4476.76..4476.89 rows=10 width=95) (actual time=34.108..34.121 rows=1 loops=1)
[...] -> Limit (cost=4476.76..4476.79 rows=10 width=54) (actual time=34.103..34.115 rows=1 loops=1)
[...] -> Sort (cost=4476.76..4476.86 rows=38 width=54) (actual time=34.102..34.112 rows=1 loops=1)
[...] Sort Key: album_1.sales_amount DESC
[...] Sort Method: quicksort Memory: 25kB
[...] -> Nested Loop (cost=4159.06..4475.94 rows=38 width=54) (actual time=33.797..34.086 rows=1 loops=1)
[...] -> Nested Loop (cost=4158.80..4462.52 rows=42 width=47) (actual time=33.784..34.073 rows=1 loops=1)
[...] -> Nested Loop (cost=4158.51..4439.64 rows=44 width=16) (actual time=33.764..34.048 rows=1 loops=1)
[...] -> Limit (cost=4158.22..4158.22 rows=1 width=44) (actual time=33.484..33.490 rows=1 loops=1)
[...] -> Sort (cost=4158.22..4159.55 rows=532 width=44) (actual time=33.483..33.489 rows=1 loops=1)
[...] Sort Key: (sum(album_2.sales_amount))
[...] Sort Method: top-N heapsort Memory: 25kB
[...] -> HashAggregate (cost=4143.59..4150.24 rows=532 width=44) (actual time=33.264..33.402 rows=396 loops=1)
[...] Group Key: band_genre_1.genre_name
[...] Batches: 1 Memory Usage: 169kB
[...] -> Hash Join (cost=3379.98..4043.37 rows=20045 width=17) (actual time=18.675..27.237 rows=24268 loops=1)
[...] Hash Cond: (band_genre_1.band_id = band_1.band_id)
[...] -> Seq Scan on band_genre band_genre_1 (cost=0.00..376.32 rows=23632 width=16) (actual time=0.009..2.139 rows=23632 loops=1)
[...] -> Hash (cost=3273.95..3273.95 rows=8482 width=13) (actual time=18.530..18.540 rows=8427 loops=1)
[...] Buckets: 16384 Batches: 1 Memory Usage: 524kB
[...] -> Hash Join (cost=303.28..3273.95 rows=8482 width=13) (actual time=2.958..16.360 rows=8427 loops=1)
[...] Hash Cond: (album_2.band_id = band_1.band_id)
[...] -> Seq Scan on album album_2 (cost=0.00..2946.39 rows=8482 width=9) (actual time=0.005..9.707 rows=8427 loops=1)
[...] Filter: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
[...] Rows Removed by Filter: 26199
[...] -> Hash (cost=180.28..180.28 rows=10000 width=4) (actual time=3.312..3.313 rows=10000 loops=1)
[...] Buckets: 16384 Batches: 1 Memory Usage: 480kB
[...] -> Index Only Scan using band_pkey on band band_1 (cost=0.29..180.28 rows=10000 width=4) (actual time=0.031..1.519 rows=10000 loops=1)
[...] Heap Fetches: 0
[...] -> Index Only Scan using band_genre_pkey on band_genre (cost=0.29..288.97 rows=44 width=16) (actual time=0.277..0.550 rows=1 loops=1)
[...] Index Cond: (genre_name = band_genre_1.genre_name)
[...] Heap Fetches: 0
[...] -> Index Scan using album_pkey on album album_1 (cost=0.29..0.51 rows=1 width=31) (actual time=0.017..0.018 rows=1 loops=1)
[...] Index Cond: (band_id = band_genre.band_id)
[...] Filter: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
[...] Rows Removed by Filter: 3
[...] -> Index Scan using band_pkey on band (cost=0.29..0.32 rows=1 width=19) (actual time=0.010..0.010 rows=1 loops=1)
[...] Index Cond: (band_id = album_1.band_id)
Planning Time: 3.546 ms
Execution Time: 48.987 ms
```



```
QUERY PLAN
text
Limit (cost=4476.76..4476.79 rows=10 width=54) (actual time=30.202..30.208 rows=1 loops=1)
  -> Sort (cost=4476.76..4476.86 rows=38 width=54) (actual time=30.201..30.207 rows=1 loops=1)
       Sort Key: album.sales_amount DESC
       Sort Method: quicksort  Memory: 25kB
       -> Nested Loop (cost=4159.06..4475.94 rows=38 width=54) (actual time=29.900..30.190 rows=1 loops=1)
            -> Nested Loop (cost=4158.80..4462.52 rows=42 width=47) (actual time=29.889..30.184 rows=1 loops=1)
                 -> Nested Loop (cost=4158.51..4439.64 rows=44 width=16) (actual time=29.868..30.160 rows=1 loops=1)
                      -> Limit (cost=4158.22..4158.22 rows=1 width=44) (actual time=29.517..29.521 rows=1 loops=1)
                           -> Sort (cost=4158.22..4159.55 rows=532 width=44) (actual time=29.516..29.519 rows=1 loops=1)
                                Sort Key: (sum(album_1.sales_amount))
                                Sort Method: top-N heapsort  Memory: 25kB
                                -> HashAggregate (cost=4143.59..4150.24 rows=532 width=44) (actual time=29.191..29.311 rows=396 loops=1)
                                     Group Key: band_genre_1.genre_name
                                     Batches: 1  Memory Usage: 169kB
                                     -> Hash Join (cost=3379.98..4043.37 rows=20045 width=17) (actual time=14.878..23.083 rows=24268 loops=1)
                                          Hash Cond: (band_genre_1.band_id = band_1.band_id)
                                          -> Seq Scan on band_genre band_genre_1 (cost=0.00..376.32 rows=23632 width=16) (actual time=0.001..1.771 rows=23632 loops=1)
                                          -> Hash (cost=3273.95..3273.95 rows=8482 width=13) (actual time=14.826..14.828 rows=8427 loops=1)
                                               Buckets: 16384  Batches: 1  Memory Usage: 524kB
                                               -> Hash Join (cost=303.28..3273.95 rows=8482 width=13) (actual time=2.686..12.896 rows=8427 loops=1)
                                                    Hash Cond: (album_1.band_id = band_1.band_id)
                                                    -> Seq Scan on album album_1 (cost=0.00..2946.39 rows=8482 width=9) (actual time=0.006..7.896 rows=8427 loops=1)
                                                         Filter: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
                                                         Rows Removed by Filter: 26199
                                                    -> Hash (cost=180.28..180.28 rows=10000 width=4) (actual time=2.492..2.492 rows=10000 loops=1)
                                                         Buckets: 16384  Batches: 1  Memory Usage: 480kB
                                                         -> Index Only Scan using band_pkey on band band_1 (cost=0.29..180.28 rows=10000 width=4) (actual time=0.016..0.903 rows=10000 loops=1)
                                                              Heap Fetches: 0
                      -> Index Only Scan using band_genre_pkey on band_genre (cost=0.29..288.97 rows=44 width=16) (actual time=0.346..0.634 rows=1 loops=1)
                           Index Cond: (genre_name = band_genre_1.genre_name)
                           Heap Fetches: 0
                 -> Index Scan using album_pkey on album (cost=0.29..0.51 rows=1 width=31) (actual time=0.018..0.020 rows=1 loops=1)
                      Index Cond: (band_id = band_genre.band_id)
                      Filter: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
                      Rows Removed by Filter: 3
            -> Index Scan using band_pkey on band (cost=0.29..0.32 rows=1 width=19) (actual time=0.012..0.012 rows=1 loops=1)
                 Index Cond: (band_id = album.band_id)
Planning Time: 1.102 ms
Execution Time: 30.207 ms
```

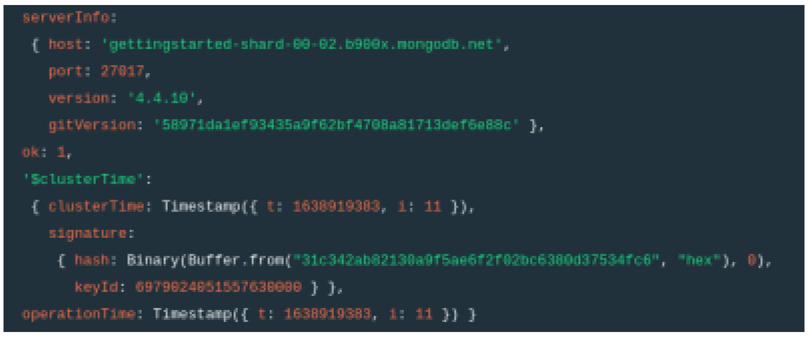


```
QUERY PLAN
text
Limit (cost=4131.79..4131.82 rows=10 width=54) (actual time=27.740..27.746 rows=1 loops=1)
  -> Sort (cost=4131.79..4131.89 rows=38 width=54) (actual time=27.739..27.745 rows=1 loops=1)
       Sort Key: album.sales_amount DESC
       Sort Method: quicksort  Memory: 25kB
       -> Nested Loop (cost=3814.11..4130.97 rows=38 width=54) (actual time=27.422..27.709 rows=1 loops=1)
            -> Nested Loop (cost=3813.83..4117.55 rows=42 width=47) (actual time=27.409..27.695 rows=1 loops=1)
                 -> Nested Loop (cost=3813.54..4094.67 rows=44 width=16) (actual time=27.389..27.673 rows=1 loops=1)
                      -> Limit (cost=3813.25..3813.25 rows=1 width=44) (actual time=27.103..27.107 rows=1 loops=1)
                           -> Sort (cost=3813.25..3814.58 rows=532 width=44) (actual time=27.102..27.105 rows=1 loops=1)
                                Sort Key: (sum(album_1.sales_amount))
                                Sort Method: top-N heapsort  Memory: 25kB
                                -> HashAggregate (cost=3798.62..3805.27 rows=532 width=44) (actual time=26.930..27.025 rows=396 loops=1)
                                     Group Key: band_genre_1.genre_name
                                     Batches: 1  Memory Usage: 169kB
                                     -> Hash Join (cost=3034.90..3698.36 rows=20052 width=17) (actual time=13.048..21.544 rows=24268 loops=1)
                                          Hash Cond: (band_genre_1.band_id = band_1.band_id)
                                          -> Seq Scan on band_genre band_genre_1 (cost=0.00..376.32 rows=23632 width=16) (actual time=0.013..2.370 rows=2363...
                                          -> Hash (cost=2928.84..2928.84 rows=8485 width=13) (actual time=12.942..12.944 rows=8427 loops=1)
                                               Buckets: 16384  Batches: 1  Memory Usage: 524kB
                                               -> Hash Join (cost=305.57..2928.84 rows=8485 width=13) (actual time=2.806..11.193 rows=8427 loops=1)
                                                    Hash Cond: (album_1.band_id = band_1.band_id)
                                                    -> Index Scan using release_date on album album_1 (cost=0.29..2601.27 rows=8485 width=9) (actual time=0.014..3...
                                                         Index Cond: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
                                                    -> Hash (cost=180.28..180.28 rows=10000 width=4) (actual time=2.703..2.704 rows=10000 loops=1)
                                                         Buckets: 16384  Batches: 1  Memory Usage: 480kB
                                                         -> Index Only Scan using band_pkey on band band_1 (cost=0.29..180.28 rows=10000 width=4) (actual time=0.01...
                                                              Heap Fetches: 0
                      -> Index Only Scan using band_genre_pkey on band_genre (cost=0.29..288.97 rows=44 width=16) (actual time=0.284..0.562 rows=1 loop...
                           Index Cond: (genre_name = band_genre_1.genre_name)
                           Heap Fetches: 0
                 -> Index Scan using album_pkey on album (cost=0.29..0.51 rows=1 width=31) (actual time=0.017..0.020 rows=1 loops=1)
                      Index Cond: (band_id = band_genre.band_id)
                      Filter: ((release_date > '2009-12-31'::date) AND (release_date < '2021-01-01'::date))
                      Rows Removed by Filter: 3
            -> Index Scan using band_pkey on band (cost=0.29..0.32 rows=1 width=19) (actual time=0.011..0.011 rows=1 loops=1)
                 Index Cond: (band_id = album.band_id)
Planning Time: 1.804 ms
Execution Time: 27.970 ms
```

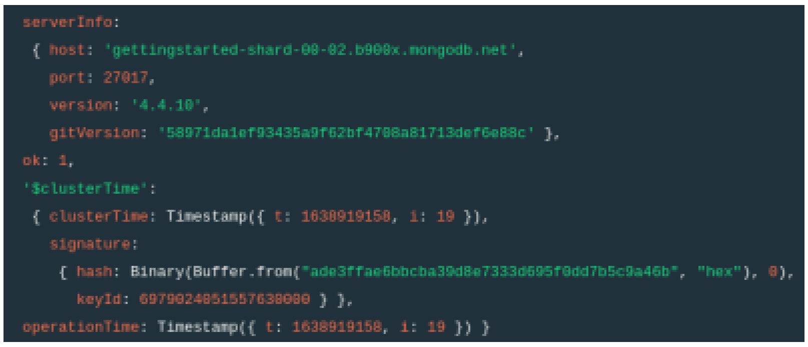


```
QUERY PLAN
text
Update on album  (cost=24.62..5708.71 rows=792 width=566) (actual time=15.718..15.720 rows=0 loops=1)
[...] -> Hash Join  (cost=24.62..5708.71 rows=792 width=566) (actual time=15.628..15.630 rows=1 loops=1)
[...] Hash Cond: (album.album_name = albums_worst_genre_2010.album_name)
[...] -> Seq Scan on album  (cost=0.00..2774.26 rows=34626 width=533) (actual time=0.010..11.698 rows=34626 loops=1)
[...] -> Hash  (cost=16.50..16.50 rows=650 width=38) (actual time=0.005..0.005 rows=1 loops=1)
[...] Buckets: 1024  Batches: 1  Memory Usage: 9kB
[...] -> Seq Scan on albums_worst_genre_2010  (cost=0.00..16.50 rows=650 width=38) (actual time=0.002..0.003 rows=1 loops=1)
Planning Time: 0.459 ms
Execution Time: 15.753 ms
```

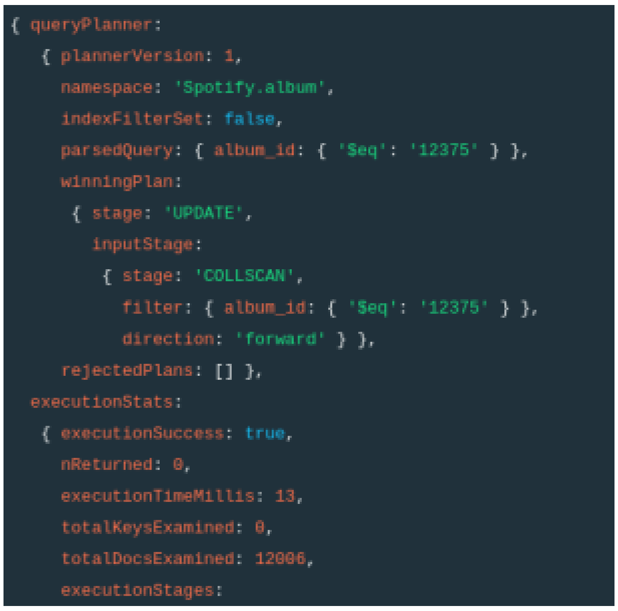
MongoDB



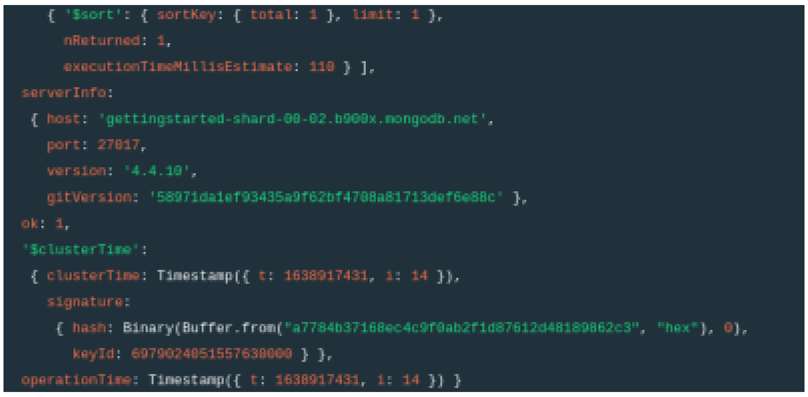
```
serverInfo:
 { host: 'gettingstarted-shard-00-02.b900x.mongodb.net',
   port: 27017,
   version: '4.4.10',
   gitVersion: '58971da1ef93435a9f62bf4708a81713def6e88c' },
ok: 1,
'$clusterTime':
 { clusterTime: Timestamp({ t: 1638919383, i: 11 }),
   signature:
    { hash: Binary(Buffer.from("31c342ab82130a9f5ae6f2f02bc6380d37534fc6", "hex"), 0),
      keyId: 6979024051557630000 } },
operationTime: Timestamp({ t: 1638919383, i: 11 }) }
```



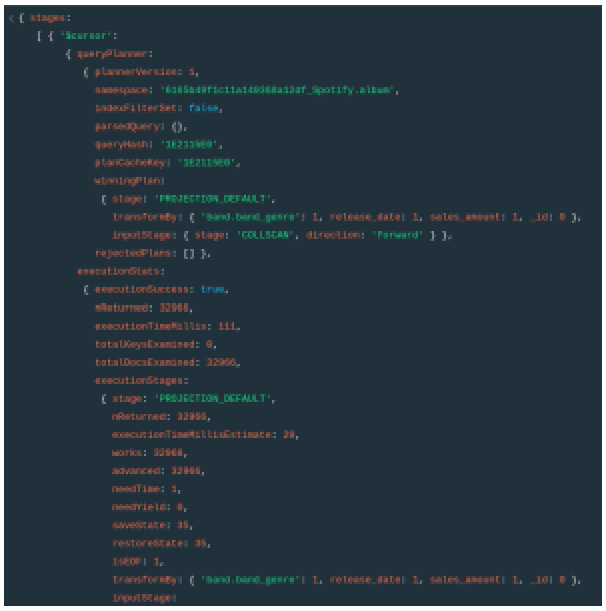
```
serverInfo:
 { host: 'gettingstarted-shard-00-02.b900x.mongodb.net',
   port: 27017,
   version: '4.4.10',
   gitVersion: '58971da1ef93435a9f62bf4708a81713def6e88c' },
ok: 1,
'$clusterTime':
 { clusterTime: Timestamp({ t: 1638919158, i: 19 }),
   signature:
    { hash: Binary(Buffer.from("ade3ffae6bbcba39d8e7333d695f0dd7b5c9a46b", "hex"), 0),
      keyId: 6979024051557630000 } },
operationTime: Timestamp({ t: 1638919158, i: 19 }) }
```



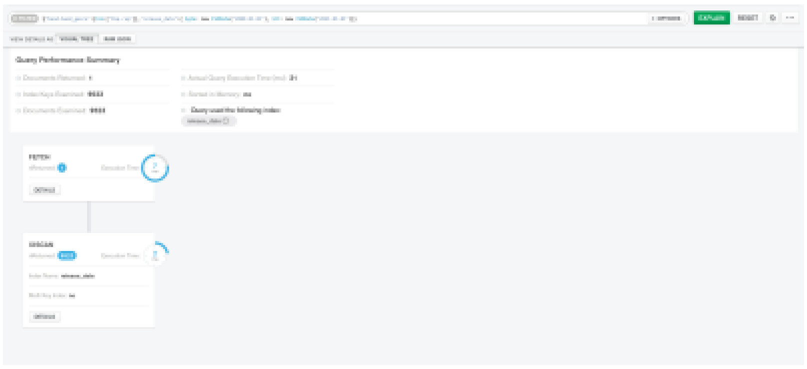
```
{ queryPlanner:
   { plannerVersion: 1,
     namespace: 'Spotify.album',
     indexFilterSet: false,
     parsedQuery: { album_id: { '$eq': '12375' } },
     winningPlan:
      { stage: 'UPDATE',
        inputStage:
         { stage: 'COLLSCAN',
           filter: { album_id: { '$eq': '12375' } },
           direction: 'forward' } },
     rejectedPlans: [] },
  executionStats:
   { executionSuccess: true,
     nReturned: 0,
     executionTimeMillis: 13,
     totalKeysExamined: 0,
     totalDocsExamined: 12006,
     executionStages:
```



```
    { '$sort': { sortKey: { total: 1 }, limit: 1 },
      nReturned: 1,
      executionTimeMillisEstimate: 110 } ],
serverInfo:
 { host: 'gettingstarted-shard-00-02.b900x.mongodb.net',
   port: 27017,
   version: '4.4.10',
   gitVersion: '58971da1ef93435a9f62bf4708a81713def6e88c' },
ok: 1,
'$clusterTime':
 { clusterTime: Timestamp({ t: 1638917431, i: 14 }),
   signature:
    { hash: Binary(Buffer.from("a7784b37168ec4c9f0ab2f1d87612d48189862c3", "hex"), 0),
      keyId: 6979024051557630000 } },
operationTime: Timestamp({ t: 1638917431, i: 14 }) }
```



```
< { stages:
     [ { '$cursor':
          { queryPlanner:
             { plannerVersion: 1,
               namespace: '6165b49f1c11a140368a12df_Spotify.album',
               indexFilterSet: false,
               parsedQuery: {},
               queryHash: '1E2115E0',
               planCacheKey: '1E2115E0',
               winningPlan:
                { stage: 'PROJECTION_DEFAULT',
                  transformBy: { 'band.band_genre': 1, release_date: 1, sales_amount: 1, _id: 0 },
                  inputStage: { stage: 'COLLSCAN', direction: 'forward' } },
               rejectedPlans: [] },
            executionStats:
             { executionSuccess: true,
               nReturned: 32966,
               executionTimeMillis: 111,
               totalKeysExamined: 0,
               totalDocsExamined: 32966,
               executionStages:
                { stage: 'PROJECTION_DEFAULT',
                  nReturned: 32966,
                  executionTimeMillisEstimate: 28,
                  works: 32968,
                  advanced: 32966,
                  needTime: 1,
                  needYield: 0,
                  saveState: 35,
                  restoreState: 35,
                  isEOF: 1,
                  transformBy: { 'band.band_genre': 1, release_date: 1, sales_amount: 1, _id: 0 },
                  inputStage:
```



EXPLAIN RESET

Query Performance Summary

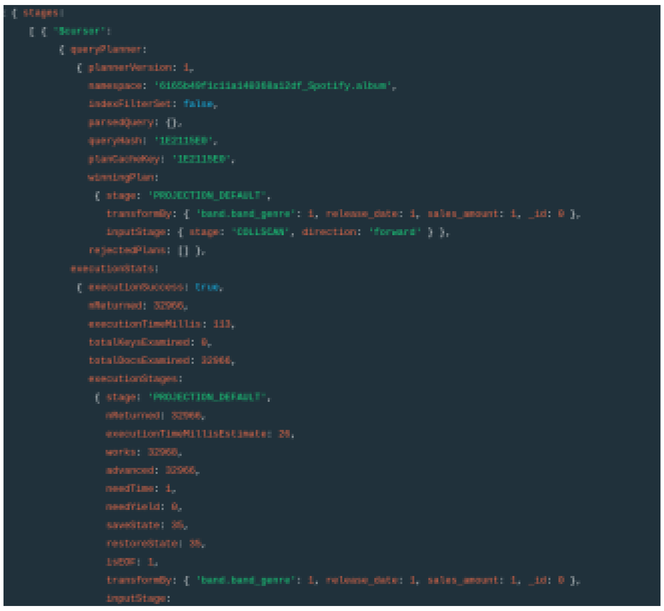
[illegible]

FETCH

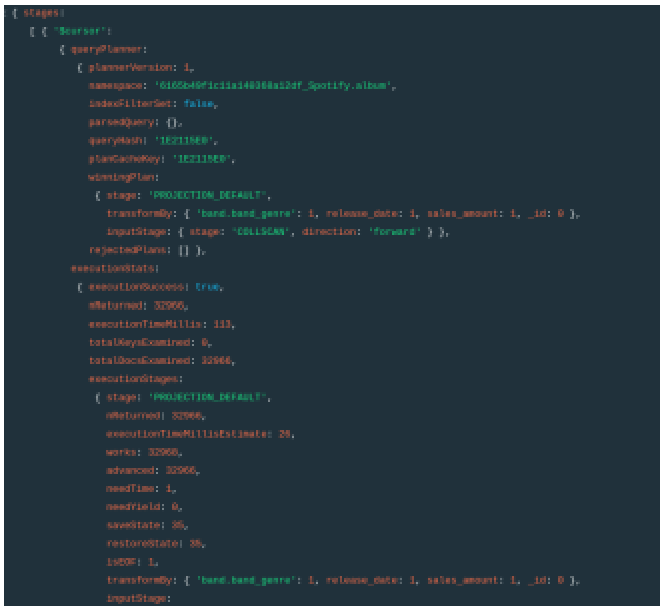
DETAILS

IXSCAN

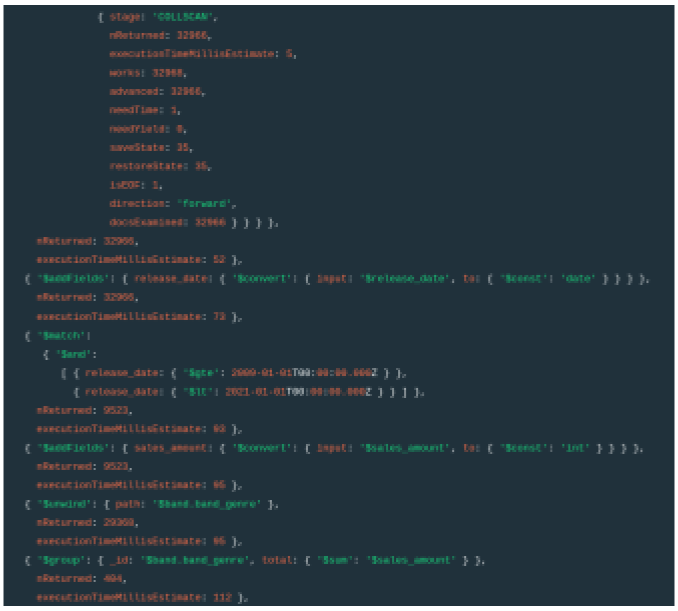
DETAILS



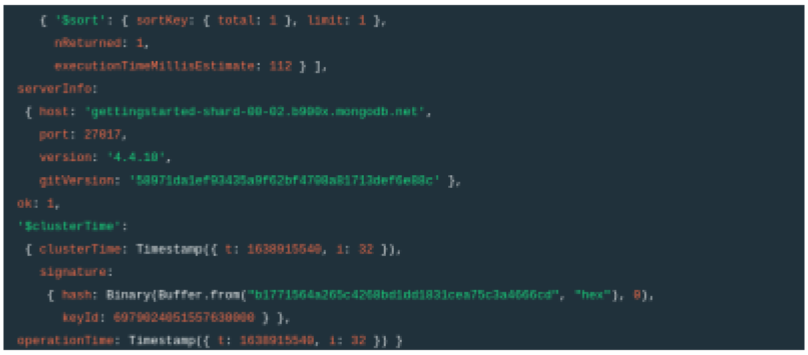
```
: { stages:
     [ { '$cursor':
          { queryPlanner:
             { plannerVersion: 1,
               namespace: '6165b49f1c11a140368a12df_Spotify.album',
               indexFilterSet: false,
               parsedQuery: {},
               queryHash: '1E2115E0',
               planCacheKey: '1E2115E0',
               winningPlan:
                { stage: 'PROJECTION_DEFAULT',
                  transformBy: { 'band.band_genre': 1, release_date: 1, sales_amount: 1, _id: 0 },
                  inputStage: { stage: 'COLLSCAN', direction: 'forward' } },
               rejectedPlans: [] },
            executionStats:
             { executionSuccess: true,
               nReturned: 32966,
               executionTimeMillis: 113,
               totalKeysExamined: 0,
               totalDocsExamined: 32966,
               executionStages:
                { stage: 'PROJECTION_DEFAULT',
                  nReturned: 32966,
                  executionTimeMillisEstimate: 26,
                  works: 32968,
                  advanced: 32966,
                  needTime: 1,
                  needYield: 0,
                  saveState: 35,
                  restoreState: 35,
                  isEOF: 1,
                  transformBy: { 'band.band_genre': 1, release_date: 1, sales_amount: 1, _id: 0 },
                  inputStage:
```



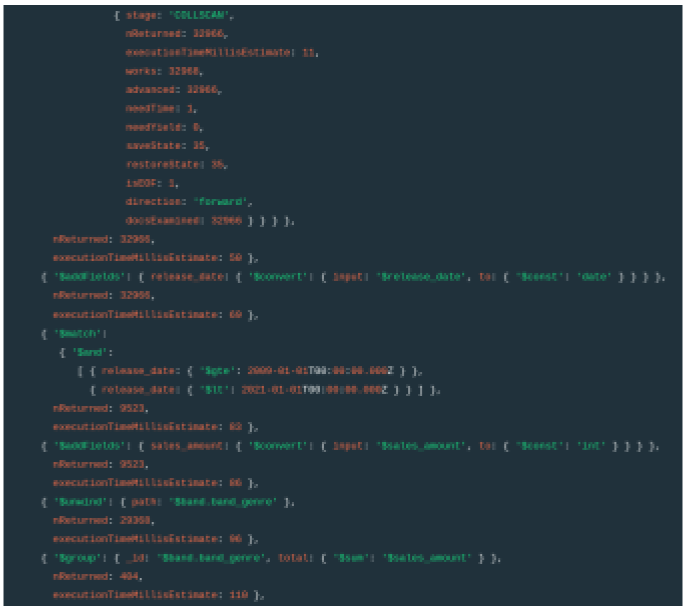
```
: { stages:
   [ { '$cursor':
        { queryPlanner:
           { plannerVersion: 1,
             namespace: '6165b49f1c11a140368a12df_Spotify.album',
             indexFilterSet: false,
             parsedQuery: {},
             queryHash: '1E2115E0',
             planCacheKey: '1E2115E0',
             winningPlan:
              { stage: 'PROJECTION_DEFAULT',
                transformBy: { 'band.band_genre': 1, release_date: 1, sales_amount: 1, _id: 0 },
                inputStage: { stage: 'COLLSCAN', direction: 'forward' } },
             rejectedPlans: [] },
          executionStats:
           { executionSuccess: true,
             nReturned: 32966,
             executionTimeMillis: 113,
             totalKeysExamined: 0,
             totalDocsExamined: 32966,
             executionStages:
              { stage: 'PROJECTION_DEFAULT',
                nReturned: 32966,
                executionTimeMillisEstimate: 28,
                works: 32968,
                advanced: 32966,
                needTime: 1,
                needYield: 0,
                saveState: 35,
                restoreState: 35,
                isEOF: 1,
                transformBy: { 'band.band_genre': 1, release_date: 1, sales_amount: 1, _id: 0 },
                inputStage:
```



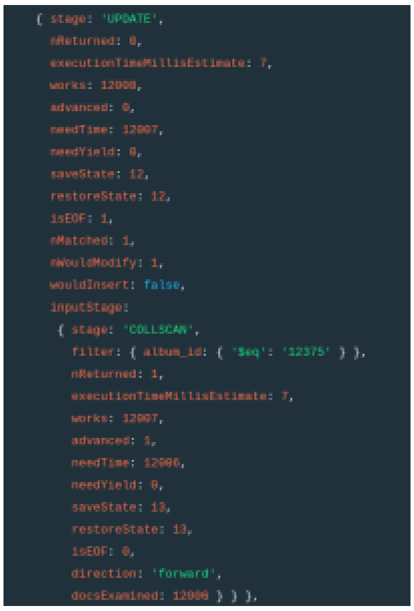
```
                 { stage: 'COLLSCAN',
                   nReturned: 32966,
                   executionTimeMillisEstimate: 5,
                   works: 32968,
                   advanced: 32966,
                   needTime: 1,
                   needYield: 0,
                   saveState: 35,
                   restoreState: 35,
                   isEOF: 1,
                   direction: 'forward',
                   docsExamined: 32966 } } } },
     nReturned: 32966,
     executionTimeMillisEstimate: 52 },
   { '$addFields': { release_date: { '$convert': { input: '$release_date', to: { '$const': 'date' } } } },
     nReturned: 32966,
     executionTimeMillisEstimate: 73 },
   { '$match':
      { '$and':
         [ { release_date: { '$gte': 2000-01-01T00:00:00.000Z } },
           { release_date: { '$lt': 2021-01-01T00:00:00.000Z } } ] },
     nReturned: 9523,
     executionTimeMillisEstimate: 63 },
   { '$addFields': { sales_amount: { '$convert': { input: '$sales_amount', to: { '$const': 'int' } } } },
     nReturned: 9523,
     executionTimeMillisEstimate: 66 },
   { '$unwind': { path: '$band.band_genre' },
     nReturned: 29369,
     executionTimeMillisEstimate: 66 },
   { '$group': { _id: '$band.band_genre', total: { '$sum': '$sales_amount' } },
     nReturned: 494,
     executionTimeMillisEstimate: 112 },
```



```
    { '$sort': { sortKey: { total: 1 }, limit: 1 },
      nReturned: 1,
      executionTimeMillisEstimate: 112 } ],
serverInfo:
 { host: 'gettingstarted-shard-00-02.b900x.mongodb.net',
   port: 27017,
   version: '4.4.10',
   gitVersion: '58971da1ef93435a9f62bf4708a81713def6e88c' },
ok: 1,
'$clusterTime':
 { clusterTime: Timestamp({ t: 1638915540, i: 32 }),
   signature:
    { hash: Binary(Buffer.from("b1771564a265c4268bd1dd1831cea75c3a4666cd", "hex"), 0),
      keyId: 6979024051557630000 } },
operationTime: Timestamp({ t: 1638915540, i: 32 }) }
```



```
                  { stage: 'COLLSCAN',
                    nReturned: 32966,
                    executionTimeMillisEstimate: 11,
                    works: 32968,
                    advanced: 32966,
                    needTime: 1,
                    needYield: 0,
                    saveState: 35,
                    restoreState: 35,
                    isEOF: 1,
                    direction: 'forward',
                    docsExamined: 32966 } } } },
      nReturned: 32966,
      executionTimeMillisEstimate: 50 },
    { '$addFields': { release_date: { '$convert': { input: '$release_date', to: { '$const': 'date' } } } },
      nReturned: 32966,
      executionTimeMillisEstimate: 60 },
    { '$match':
       { '$and':
          [ { release_date: { '$gte': 2000-01-01T00:00:00.000Z } },
            { release_date: { '$lt': 2021-01-01T00:00:00.000Z } } ] },
      nReturned: 9523,
      executionTimeMillisEstimate: 60 },
    { '$addFields': { sales_amount: { '$convert': { input: '$sales_amount', to: { '$const': 'int' } } } },
      nReturned: 9523,
      executionTimeMillisEstimate: 60 },
    { '$unwind': { path: '$band.band_genre' },
      nReturned: 29360,
      executionTimeMillisEstimate: 90 },
    { '$group': { _id: '$band.band_genre', total: { '$sum': '$sales_amount' } },
      nReturned: 404,
      executionTimeMillisEstimate: 110 },
```



```
    { stage: 'UPDATE',
      nReturned: 0,
      executionTimeMillisEstimate: 7,
      works: 12008,
      advanced: 0,
      needTime: 12007,
      needYield: 0,
      saveState: 12,
      restoreState: 12,
      isEOF: 1,
      nMatched: 1,
      nWouldModify: 1,
      wouldInsert: false,
      inputStage:
       { stage: 'COLLSCAN',
         filter: { album_id: { '$eq': '12375' } },
         nReturned: 1,
         executionTimeMillisEstimate: 7,
         works: 12007,
         advanced: 1,
         needTime: 12006,
         needYield: 0,
         saveState: 13,
         restoreState: 13,
         isEOF: 0,
         direction: 'forward',
         docsExamined: 12006 } } },
```