Faculty of Sciences of the University of Lisbon

Database Technologies - Report 3

Database Tuning and Optimization

Group 5 - António Rebelo · Duarte Balata · Filipa Serrano · Marco Mercier

taught by
Professor Cátia Pesquita

August 11, 2021

1 Introduction

Performance considerations are a design issue that mustn't be skipped in any database implementation [4, 5].

The first step for this part of the project was to analyze and optimize the queries from the previous stages in order to get rid of unnecessary nested queries and operators. After that, index optimization and database tuning were introduced taking in account not only the proposed complex queries but also the overall usage of a bands/music database.

2 PostgreSQL

2.1 Query 1: Which artists belong/have belonged to the "Indie rock" band which has released the longest albums on average?

Query optimization

The query which was created and used for the other parts of the project was optimized to the following one:

```
SELECT BE.artist_name FROM Belongs BE,
                (SELECT A.band_id AS band_id, AVG(A.time) AS avg_length
                FROM Albums A, Bands BA, Bands_Genres BG, Genres G
                WHERE G.genre = 'Indie rock'
                AND  BG.genre_id = G.genre_id
                AND BA.band_id = BG.band_id
                AND A.band_id = BA.band_id
                GROUP BY A.band_id
                ORDER BY avg_length DESC
                LIMIT 1) AS bandid
WHERE BE.band_id = bandid.band_id;
```

The main differences were the cutback of nested queries from two to only one, the ordering of the equalities starting from the most selective one, the removal of the costly 'DISTINCT' command, and also the replacement of the 'ILIKE' operator for a simple equality. The changes implemented improved the query performance. The difference in performance was very clear in the query plan, where 3 sequential scans turned to only 1. Now the query is making use of an index on the albums table that was created due to a 'UNIQUE' constraint, which means that we can make full use of the database structure and the execution time was reduced to less than half the time. The query plan of the initial non-optimized query can be found in appendix 5. The query plan of the optimized query is discussed in the next part and can be found in figure 1.

Further optimization which eliminated all nested queries was attempted, with the following structure:

```
SELECT BE.artist_name,  AVG(A.time) AS avg_length
                        FROM Albums A, Bands BA, Bands_Genres BG, Genres G, Belongs BE
                           WHERE G.genre = 'Indie rock'
                            AND  BG.genre_id = G.genre_id
                           AND BA.band_id = BG.band_id
                          AND A.band_id = BA.band_id
                        AND BE.band_id = A.band_id
                GROUP BY BE.band_id,BE.artist_name
                ORDER BY avg_length DESC
                FETCH FIRST 1 ROWS WITH TIES;
```

However, this query didn't perform as well as the first optimization presented, among other reasons, because the sorting is more costly.

Query plan without index optimization

With the query optimized, the next step was to apply adequate indexes which could improve the performance of the complex operations. Firstly, it is important to understand the query plan and where it can be improved. The query plan for query 1 without applying any "extra" indexes (there are a few indexes which are already created due to the integrity constraints) is presented in figure 1.

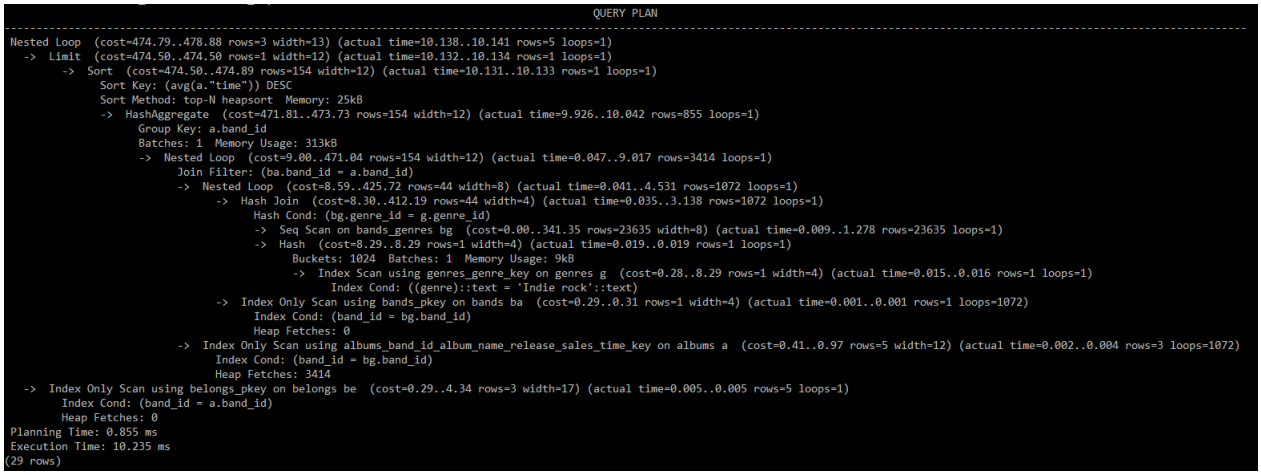

Figure 1: Query plan for rewritten query 1 without index optimization

Index optimization

Most of the operation used numerical ids which already had indexes associated with them, but there were still some possibilities of indexes which had potential to improve the query.

To try to prevent the sequential scan performed on the Bands_Genres table, an index was applied to the field genre_id, which only had a shared index (band_id, genre_id), which was not useful for the join by genre_id. For comparison and testing purposes a hash, btree and a clustered btree index were applied, but the one chosen for the query plan was the hash, which was expected since the query performs an equality and not a range. This index was successful, changing the sequential scan to a "Bitmap Index Scan", and reducing the execution time. The query plan of the optimized query with the index applied can be found in appendix 6. When the hash index was removed, the chosen index was the unclustered btree, this was due to the fact that the clustered index doesn't provide any clear advantage since the database is small enough to fit in memory.

Other indexes were tried but, besides the one mentioned, none of them improved the query plan and execution time. One possible explanation is the fact that the table genres is not very big, so some of the indexes to improve searches on this table might not be very useful.

2.2 Query 2:Which musical genre has the biggest number of albums released on each decade?

Query optimization

The second implemented query originally had 2 nested queries, 1 RANK() with PARTITION and ORDER BY clauses, 3 EXTRACT, and a GROUP BY/ORDER BY clause. It was possible to optimize this query to the following:

```
SELECT * FROM ( SELECT  (EXTRACT (DECADE FROM A.release)) AS decade, G.genre AS genre,
                RANK() OVER (PARTITION BY (EXTRACT (DECADE FROM A.release))
                ORDER BY COUNT(A.album_id) DESC)
                FROM  Albums A, Bands BA, Bands_Genres BG, Genres G
                    WHERE A.band_id = BA.band_id
                    AND BA.band_id = BG.band_id
                    AND BG.genre_id = G.genre_id
                    GROUP BY decade, G.Genre )  AS ranking
WHERE rank = 1;
```

As a result, the query has one less nested query, lost one ORDER BY clause and has just an EXTRACT clause. The query plan of the initial non-optimized query can be found in appendix 7. The query plan of the optimized query is discussed in the next part and can be found in figure 2.

Query plan without index optimization

The query plan for this query is depicted in figure 2. It shows a sequential scan on tables Genres, Bands and Bands_Genres and an external merge sort for the Decade, which means it will get pieces of data into memory, sort it and then finally merge it.

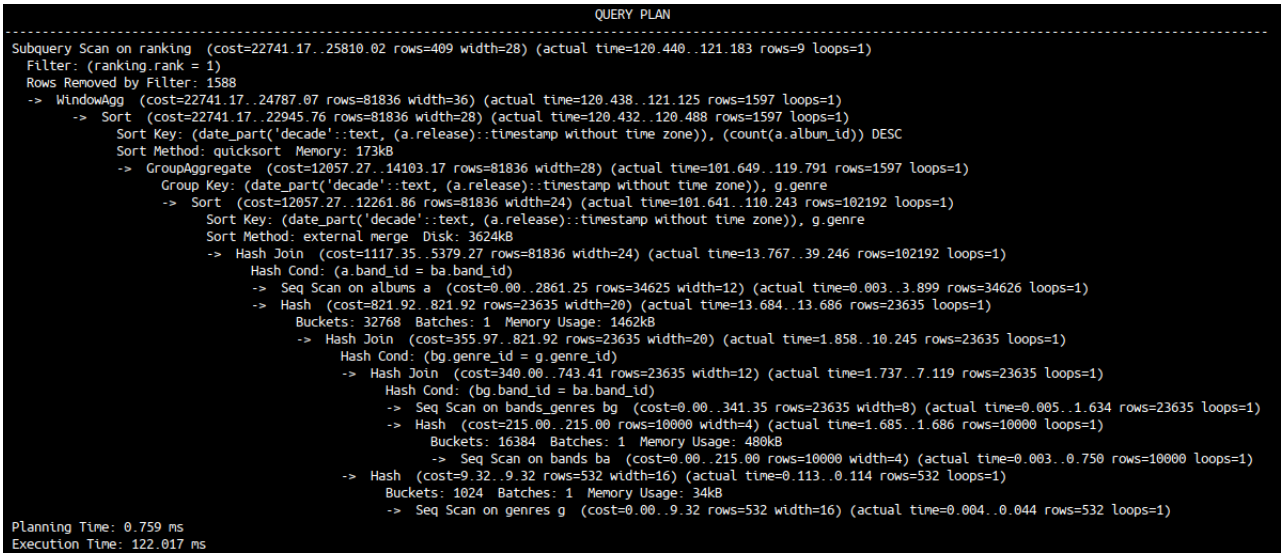

Figure 2: Query plan for rewritten query 2 without index optimization

Index optimization

In order to optimize the query execution time, one would need to overcome the sequential scans and try to make them index only scans. However, since all those tables are used in order to get all the albums of all the genres, there's no index that could help since a sequential scan will always be the best plan. Also, an index on the decades results of the EXTRACT method , using psql's functional index [3], was not possible to implement.

Decade-related topics are a common place in the music industry. Therefore, it could be good to have the albums table clustered on decade, creating a new column that would be computed using the (EXTRACT (DECADE FROM A.release)) on insert time. This solution will be explored in the Database Tuning section further ahead.

2.3 Impact on Performance

Another important point to study when implementing an index in a database is whether it is going to have a large impact on performance. Hash indexes might need to be restructured if one of its buckets fills up. The same can happen if many rows are deleted. These operations can be costly and might decrease performance.

To study this, the insert that was used for the second part of the project will be tested, namely:

```
INSERT INTO Bands_Genres VALUES (band_id, genre_id);
```

A hash index was created for the genre_id field. Many inserts were made to the Bands_Genres table, however **no** noticeable differences in insert time with or without index were observed. Probably the point at where the costly operations mentioned above needed to be done wasn't reached, but this is not too concerning. For this database in particular, the amount of expected writes is low. Hence, regarding the implementation of indexes, the negative impact on write performance will most times be mitigated by the positive impact on read performance.

2.4 Database Tuning

In order to optimize for the execution of the queries multiple potential database schema alterations were considered. Firstly, it was considered having the genre names in the Bands_Genres table in order to decrease the number of table join operations by one. However, when testing this possible optimization, it was found that

the join operation on the table Bands_Genres would be negligible for the overall query performances, since this table is small in size and composed solely of integer identifiers.

Secondly, the addition of band genres directly into the Bands table was considered. Even though this change could enhance the performance of the queries, it was concluded that the downsides of its implementation would exceed its benefits. Not only would it lead to multiple problems regarding the integrity constraints of the data, but also increase very significantly the size of the Bands table, since every band would appear repeated several times, which would take a toll on disk usage.

Finally, the last proposed database optimization was to insert a Decade column into the Albums table in order to increase the performance of the second query, that needs to group the albums by decade. The main concern with the insertion of this column was the toll it could take on insert time, however after testing insert queries on both scenarios, it was possible to conclude that the change on insert time is negligible, especially when taking in consideration its potential benefit in terms of query performance. Consequently, the effects of this change on the performance were tested for the second query and a reduction in execution time to approximately half from the pre-optimized version was verified. The functions used in order to create the new column can be found in appendix 8

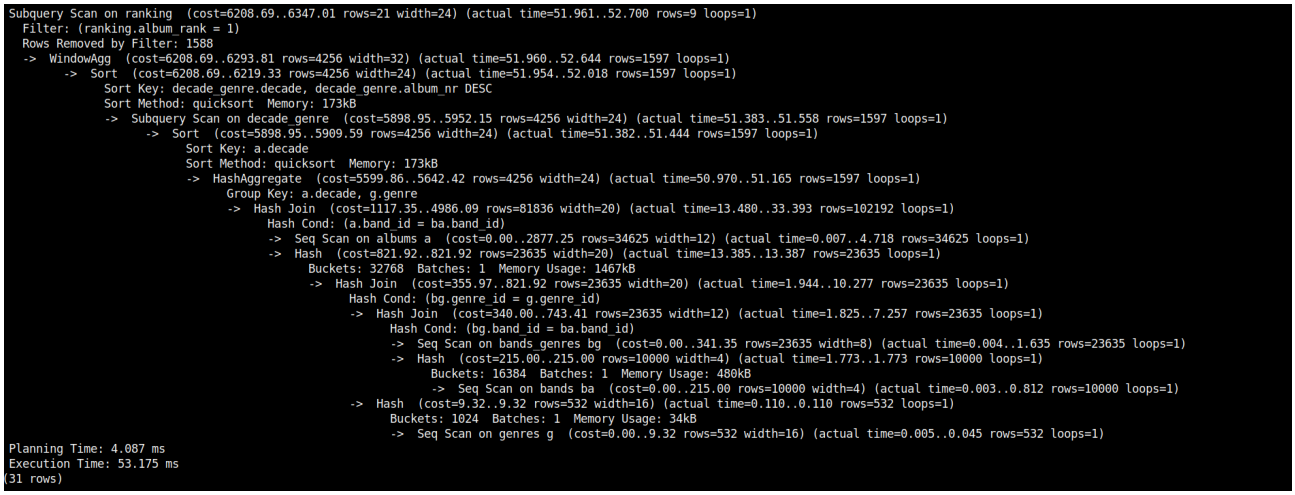

Figure 3: Query plan for rewritten query 2 with decade column.

The second query was rewritten taking in account this newly created columns is in appendix 9 .We then inserted a btree index on the newly generated decade column. Since the second query needs to perform reads on all rows, adding an index to this column wouldn't affect its performance, however if the necessity arrives to extract information regarding a single decade or group of decades it would greatly optimize the execution schema. In order to test this, the second query was rewritten in order to retrieve solely the top genre for 1950's decade. The execution of this query was found to be more than 10 times faster when using a btree type index on the decade column of the Albums table, proving the utility of this index for possible future operations involving the decade attribute.

3 MongoDB

3.1 Query 1: Which artists belong/have belonged to the "Indie rock" band which has released the longest albums on average?

This complex operation for MongoDB consisted on the following:

```
db.Bands.aggregate([
        {"$match": {"genres":"Indie rock"}},
        {"$project": {
                "_id":0,
                "AvgTime": { "$avg": "$albums.time"},
                "name":1 ,
                "artists.names":1 }},
```

```
        {"$sort":{"AvgTime":-1 }},
        {"$limit":1} ])
```

A way to rewrite the query to optimize it was not found.

Query plan without index optimization

Some of the most relevant statistics which were analyzed for the execution of this query are the following:

```
"executionStats" : {"executionSuccess" : true,
                    "executionTimeMillis" : 14,
                    "totalKeysExamined" : 0,
                    "totalDocsExamined" : 10000,
                    "executionStages" : {
                        "stage" : "COLLSCAN",
                        "nReturned" : 1072,
```

The query planner decides that the winning plan is a full collection scan (COLLSCAN) because of the match phase. This happens because genres is an array inside the document and, since we want all the documents that match genre = "Indie rock", it will scan all documents and return only 1072.

Index optimization

In order to improve this query's performance, it was suggested that a multi-key index [1] on genres could be applied, so it is possible to scan the genre of interest (in this case 'Indie rock') directly. The index was applied using the command:

```
db.Bands.createIndex( { "genres": 1 } )
```

The execution statistics after applying this index were improved, by examining 1072 keys and only the 1072 documents containing them- corresponding to the 'Indie rock' genre, instead of 10000 documents.

3.2 Query 2:Which musical genre has the biggest number of albums released on each decade?

The original query adds an year field extracted from the albums' release date followed by a bucket phase to get the decades, which creates a new genres array that needs to be unwinded in order to operate on it.
It was possible to optimize the query by getting the decade from a substring of the first 3 characters of the release date, avoiding the bucket and subsequent unwinding phases, as follows:

```
db.Bands.aggregate([
        {"$unwind":"$genres"},
        {"$unwind":"$albums"},
        {"$addFields":{"Decade":{ "$substr" :  ["$albums.release", 0 , 3]}}},
        {"$group":{"_id":{"Decade":"$Decade", "Genre":"$genres" },"count":{"$sum":1}}},
        {"$sort":{"_id.Decade":1,"count":-1}},
        {"$group":{"_id":"$_id.Decade",
                "Top genre":{"$first":"$_id.Genre"},
                "Albums":{"$first":"$count"} }}  ,
        {"$sort":{"_id":1}} ])
```

Query plan without index optimization

Some of the most relevant statistics which were analyzed for the execution of this query are the following:

```
"executionStats" : {"executionSuccess" : true,
                    "nReturned" : 10000,
                    "executionTimeMillis" : 451,
                    "totalKeysExamined" : 0,
                    "totalDocsExamined" : 10000,
                    "executionStages" : {
                        "stage" : "COLLSCAN",
                        "nReturned" : 10000,
                        "docsExamined" : 10000}
```

The query planner considers the winning plan to be a full collection scan and it makes sense, since all the documents were examined and retrieved. There seems to be no other way to optimize this query since it uses all the documents in the collection and then group them by decade and sort them by number of released albums per genre.

Index optimization

Several indexes were tried but the query planner always chooses to do a full collection scan. The solution, since decade-related queries are, as stated before, a common place in the music industry, would be to add a decade field per album, computed on insert time. This solution is explored in the subsequent sections.

3.3 Impact on Performance

Even though in this case the index in question is multikey, the same results as in the SQL case were obtained. No difference in write or delete performance were noticed. For the same reasons stated before, not reaching the point where the impact can be noted is not very concerning and is only an indication of the robustness of the indexing implementation strategies used in these databases.

3.4 Database Tuning

In similarity to what was done on PostgreSQL, a decade attribute was also added to each Album object on the MongoDB's Bands documents.The python code [2] to implement this change is located is annexed to this report.

Again, similarly to what was verified on PostgreSQL, the benefits resulting from the reduction in the execution time of our second query, far exceeded the almost unnoticeable increased computational cost of data insertions. The rewritten query with this decade field is in appendix 10.

4 Discussion and Conclusion

First it should be noted that what provided the largest increase on performance was rewriting the queries. In SQL the rewriting not only optimizes the operations themselves, which is huge, but in some cases it also allows for the usage of indexes created by Primary or Unique constraints. In other words, rewriting the queries allows the user to take full advantage of the structure of the database. In MongoDB even though the same doesn't hold true regarding the indexes, the performance gain that comes from the optimization of the operations can also be very large.

The SQL schema was already in Boyce Codd Normal Form, very decomposed and with several indexes provided by the integrity constraints. This meant that there were few indexes to implement that would improve the queries' performance. Even so, an index on Bands_Genre.genre_id improved the performance of the first query.

Concerning the NoSQl data model, there were no indexes, except for the default document's one. The only addition that provided a performance improvement was a multikey index on genres array.

Besides index implementation, query restructuring both in SQL and MongoDB, namely unnesting, filtering by selectivity criteria and removal of unnecessary steps made the queries a lot faster.

Taking in account the complexity of the second query and the domain of the database, introducing a decade field in the albums relation/array proved to be a great solution, not only improving query performance by a large factor, but also making querying based on this relevant attribute in the future a lot easier. It should also be noted, that overall, the changes introduced in the databases did not impact write performance on a noticeable level. Even if this increase was noticeable, the optimization on reads performance would always be preferred, since in a database of this domain reads are expected more frequently than writes.

All of the optimization tests and queries were performed on SSD (there were no availiable HDDs). This means that all the performance gains introduced by query optimization, indexes and schema evolution would have a greater impact on an HDD database.

Regarding the entirety of the project in all its three parts, the objectives were achieved: since implementing the databases to *fully* optimize it, including ACID property studies, concurrency control and overall performance tuning.

References

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

[2] MongoDB. *PyMongo 3.11.2 Documentation*. URL: https://pymongo.readthedocs.io/en/stable/.

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

[4] Raghu Ramakrishnan and Johannes Gehrke - 3^{rd}*edition*. *Database Management Systems*. McGraw-Hill, 2003. ISBN: 0072465638.

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

5 Appendix: Query and query plan for query 1 in PostgreSQL before query rewriting

```sql
SELECT DISTINCT BE.artist_name
FROM Belongs BE, (SELECT B_avg.band_id  FROM
(SELECT A.band_id AS band_id, AVG(A.time) AS avg_length
        FROM Albums A
        GROUP BY A.band_id
        HAVING A.band_id IN (SELECT BA.band_id
           FROM Bands BA, Bands_Genres BG, Genres G
           WHERE BA.band_id = BG.band_id
             AND BG.genre_id = G.genre_id
             AND G.genre ILIKE 'Indie rock')) AS B_avg
        ORDER BY B_avg.avg_length DESC
        LIMIT 1)  AS bandid
WHERE BE.band_id = bandid.band_id;
```

```
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 Unique  (cost=3567.19..3567.20 rows=3 width=13) (actual time=21.207..21.212 rows=5 loops=1)
   ->  Sort  (cost=3567.19..3567.19 rows=3 width=13) (actual time=21.206..21.209 rows=5 loops=1)
         Sort Key: be.artist_name
         Sort Method: quicksort  Memory: 25kB
         ->  Nested Loop  (cost=3563.07..3567.16 rows=3 width=13) (actual time=21.191..21.195 rows=5 loops=1)
               ->  Limit  (cost=3562.78..3562.78 rows=1 width=12) (actual time=21.181..21.183 rows=1 loops=1)
                     ->  Sort  (cost=3562.78..3571.07 rows=3316 width=12) (actual time=21.180..21.182 rows=1 loops=1)
                           Sort Key: (avg(a."time")) DESC
                           Sort Method: top-N heapsort  Memory: 25kB
                           ->  HashAggregate  (cost=3413.56..3513.04 rows=3316 width=12) (actual time=19.961..21.101 rows=855 loops=1)
                                 Group Key: a.band_id
                                 Filter: (hashed SubPlan 1)
                                 Batches: 1  Memory Usage: 1489kB
                                 Rows Removed by Filter: 6018
                                 ->  Seq Scan on albums a  (cost=0.00..2812.25 rows=34625 width=12) (actual time=0.009..4.468 rows=34625 loops=1)
                                 SubPlan 1
                                   ->  Nested Loop  (cost=10.95..428.08 rows=44 width=4) (actual time=0.928..5.302 rows=1072 loops=1)
                                         ->  Hash Join  (cost=10.66..414.54 rows=44 width=4) (actual time=0.897..3.953 rows=1072 loops=1)
                                               Hash Cond: (bg.genre_id = g.genre_id)
                                               ->  Seq Scan on bands_genres bg  (cost=0.00..341.35 rows=23635 width=8) (actual time=0.017..1.214 rows=23635 loops=1)
                                               ->  Hash  (cost=10.65..10.65 rows=1 width=4) (actual time=0.866..0.867 rows=1 loops=1)
                                                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                                     ->  Seq Scan on genres g  (cost=0.00..10.65 rows=1 width=4) (actual time=0.069..0.861 rows=1 loops=1)
                                                           Filter: ((genre)::text ~~* 'Indie rock'::text)
                                                           Rows Removed by Filter: 531
                                         ->  Index Only Scan using bands_pkey on bands ba  (cost=0.29..0.31 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=1072)
                                               Index Cond: (band_id = bg.band_id)
                                               Heap Fetches: 0
               ->  Index Only Scan using belongs_pkey on belongs be  (cost=0.29..4.34 rows=3 width=17) (actual time=0.009..0.010 rows=5 loops=1)
                     Index Cond: (band_id = a.band_id)
                     Heap Fetches: 0
 Planning Time: 0.912 ms
 Execution Time: 21.503 ms
(33 rows)
```

Figure 4: Query plan for query 1 before query rewrite.

6 Appendix: Query plan for query 1 optimized and with indexes applied

```
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=158.33..162.43 rows=3 width=13) (actual time=5.801..5.803 rows=5 loops=1)
   ->  Limit  (cost=158.05..158.05 rows=1 width=12) (actual time=5.795..5.796 rows=1 loops=1)
         ->  Sort  (cost=158.05..158.43 rows=154 width=12) (actual time=5.794..5.795 rows=1 loops=1)
               Sort Key: (avg(a."time")) DESC
               Sort Method: top-N heapsort  Memory: 25kB
               ->  HashAggregate  (cost=155.35..157.28 rows=154 width=12) (actual time=5.599..5.701 rows=855 loops=1)
                     Group Key: a.band_id
                     Batches: 1  Memory Usage: 313kB
                     ->  Nested Loop  (cost=5.32..154.58 rows=154 width=12) (actual time=0.084..4.861 rows=3414 loops=1)
                           Join Filter: (ba.band_id = a.band_id)
                           ->  Nested Loop  (cost=4.90..109.27 rows=44 width=8) (actual time=0.077..1.517 rows=1072 loops=1)
                                 ->  Nested Loop  (cost=4.62..95.73 rows=44 width=4) (actual time=0.071..0.384 rows=1072 loops=1)
                                       ->  Index Scan using genres_genre_key on genres g  (cost=0.28..8.29 rows=1 width=4) (actual time=0.014..0.015 rows=1 loops=1)
                                             Index Cond: ((genre)::text = 'Indie rock'::text)
                                       ->  Bitmap Heap Scan on bands_genres bg  (cost=4.34..87.00 rows=44 width=8) (actual time=0.052..0.303 rows=1072 loops=1)
                                             Recheck Cond: (genre_id = g.genre_id)
                                             Heap Blocks: exact=105
                                             ->  Bitmap Index Scan on bands_genres_genre_id_hash  (cost=0.00..4.33 rows=44 width=0) (actual time=0.039..0.039 rows=1072 loops=1)
                                                   Index Cond: (genre_id = g.genre_id)
                                 ->  Index Only Scan using bands_pkey on bands ba  (cost=0.29..0.31 rows=1 width=4) (actual time=0.001..0.001 rows=1 loops=1072)
                                       Index Cond: (band_id = bg.band_id)
                                       Heap Fetches: 0
                           ->  Index Only Scan using albums_band_id_album_name_release_sales_time_key on albums a  (cost=0.41..0.97 rows=5 width=12) (actual time=0.002..0.003 rows=3 loops=1072)
                                 Index Cond: (band_id = bg.band_id)
                                 Heap Fetches: 3414
   ->  Index Only Scan using belongs_pkey on belongs be  (cost=0.29..4.34 rows=3 width=17) (actual time=0.005..0.005 rows=5 loops=1)
         Index Cond: (band_id = a.band_id)
         Heap Fetches: 0
 Planning Time: 0.857 ms
 Execution Time: 5.893 ms
(30 rows)
```

Figure 5: Query plan for query 1 after query optimization and indexes

7 Appendix: Query and query plan for query 2 in PostgreSQL before query rewriting

```sql
SELECT Ranking.decade AS "Decade", Ranking.genre AS "Genre", Ranking.album_nr AS "Album_nr"
FROM (SELECT Decade_Genre.decade AS decade, Decade_Genre.genre AS genre,
        RANK() OVER (PARTITION BY Decade_Genre.decade
        ORDER BY Decade_Genre.album_nr DESC)
        album_rank, Decade_Genre.album_nr as album_nr
        FROM (SELECT (EXTRACT (DECADE FROM A.release)) AS decade,
        G.genre AS genre, COUNT(A.album_id) AS album_nr
                FROM Albums A, Bands BA, Bands_Genres BG, Genres G
                WHERE A.band_id = BA.band_id
                AND BA.band_id = BG.band_id
                AND BG.genre_id = G.genre_id
                GROUP BY (EXTRACT (DECADE FROM A.release)), G.genre
                ORDER BY (EXTRACT (DECADE FROM A.release)) ) AS Decade_Genre)  AS Ranking
WHERE Ranking.album_rank = 1;
```

```
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 Subquery Scan on ranking  (cost=23559.53..26219.20 rows=409 width=28) (actual time=372.491..374.362 rows=9 loops=1)
   Filter: (ranking.album_rank = 1)
   Rows Removed by Filter: 1588
   ->  WindowAgg  (cost=23559.53..25196.25 rows=81836 width=36) (actual time=372.489..374.229 rows=1597 loops=1)
         ->  Sort  (cost=23559.53..23764.12 rows=81836 width=28) (actual time=372.478..372.624 rows=1597 loops=1)
               Sort Key: decade_genre.decade, decade_genre.album_nr DESC
               Sort Method: quicksort  Memory: 173kB
               ->  Subquery Scan on decade_genre  (cost=12057.27..14921.53 rows=81836 width=28) (actual time=308.241..371.159 rows=1597 loops=1)
                     ->  GroupAggregate  (cost=12057.27..14103.17 rows=81836 width=28) (actual time=308.240..370.936 rows=1597 loops=1)
                           Group Key: (date_part('decade'::text, (a.release)::timestamp without time zone)), g.genre
                           ->  Sort  (cost=12057.27..12261.86 rows=81836 width=24) (actual time=308.228..335.195 rows=102192 loops=1)
                                 Sort Key: (date_part('decade'::text, (a.release)::timestamp without time zone)), g.genre
                                 Sort Method: external merge  Disk: 3624kB
                                 ->  Hash Join  (cost=1117.35..5379.27 rows=81836 width=24) (actual time=53.960..125.557 rows=102192 loops=1)
                                       Hash Cond: (a.band_id = ba.band_id)
                                       ->  Seq Scan on albums a  (cost=0.00..2861.25 rows=34625 width=12) (actual time=0.004..13.912 rows=34626 loops=1)
                                       ->  Hash  (cost=821.92..821.92 rows=23635 width=20) (actual time=53.798..53.802 rows=23635 loops=1)
                                             Buckets: 32768  Batches: 1  Memory Usage: 1462kB
                                             ->  Hash Join  (cost=355.97..821.92 rows=23635 width=20) (actual time=16.240..44.960 rows=23635 loops=1)
                                                   Hash Cond: (bg.genre_id = g.genre_id)
                                                   ->  Hash Join  (cost=340.00..743.41 rows=23635 width=12) (actual time=15.985..37.641 rows=23635 loops=1)
                                                         Hash Cond: (bg.band_id = ba.band_id)
                                                         ->  Seq Scan on bands_genres bg  (cost=0.00..341.35 rows=23635 width=8) (actual time=0.258..13.239 rows=23635 loops=1)
                                                         ->  Hash  (cost=215.00..215.00 rows=10000 width=4) (actual time=15.649..15.649 rows=10000 loops=1)
                                                               Buckets: 16384  Batches: 1  Memory Usage: 480kB
                                                               ->  Seq Scan on bands ba  (cost=0.00..215.00 rows=10000 width=4) (actual time=0.011..13.211 rows=10000 loops=1)
                                                   ->  Hash  (cost=9.32..9.32 rows=532 width=16) (actual time=0.235..0.236 rows=532 loops=1)
                                                         Buckets: 1024  Batches: 1  Memory Usage: 34kB
                                                         ->  Seq Scan on genres g  (cost=0.00..9.32 rows=532 width=16) (actual time=0.008..0.096 rows=532 loops=1)
 Planning Time: 2.592 ms
 Execution Time: 376.950 ms
(31 rows)
```

Figure 6: Query plan for query 2 before query rewrite.

8 Appendix: Code for the creation of the Decade column in PostgreSQL

To create the column based on the current data:

```
ALTER TABLE Albums add column decade INTEGER;
UPDATE Albums
SET decade = EXTRACT(DECADE from Albums.release)
WHERE decade IS NULL;
```

To automatically calculate decade on insert time:

```
CREATE OR REPLACE FUNCTION insert_album_new( band_id_ albums.band_id%TYPE,
album_name_ Albums.album_name%TYPE,
release_ Albums.release%TYPE, sales_ Albums.sales%TYPE, time_ Albums.time%TYPE,
abstract_ Albums.abstract%TYPE)
        RETURNS void AS $$
        BEGIN
        INSERT INTO Albums (band_id, album_name, release, sales, time, abstract, decade)
        VALUES (band_id_, album_name_, release_, sales_, time_, abstract_,
        (EXTRACT (DECADE FROM release_)));
        END;
$$ LANGUAGE plpgsql;
```

9 Appendix: Code for query 2 after schema evolution in PostgreSQL

```
SELECT  * FROM (
    SELECT  A.decade AS decade, G.genre AS genre, COUNT(A.album_id) AS album_nr  ,
        RANK() OVER (PARTITION BY A.decade ORDER BY COUNT(A.album_id) DESC) rank
        FROM  Albums A, Bands BA, Bands_Genres BG, Genres G
                WHERE A.band_id = BA.band_id
                AND BA.band_id = BG.band_id
                AND BG.genre_id = G.genre_id
                GROUP BY decade, G.Genre )  AS ranking
WHERE rank = 1;
```

10 Appendix: Code for query 2 after schema evolution in MongoDB

```
db.Bands2.aggregate([
        {"$unwind":"$genres"},
        {"$unwind":"$albums"},
        {"$group":{
                "_id":{"Decade":"$albums.decade", "Genre":"$genres" },
                "count":{"$sum":1}   }  } ,
        {"$sort":{"_id.Decade":1,"count":-1}},
        {"$group":{
                "_id":"$_id.Decade",
                "Top genre":{"$first":"$_id.Genre"},
                "Albums":{"$first":"$count"} }}  ,
        {"$sort":{"_id":1}} ]).pretty()
```