Analyzing Big Data from Online Music Subscription Services

2AMD15, Big Data Management - Q3 (2023-2024)

Group 16

Full Name	Student ID
Roëlle Bänffer	1523198
Amir Ali Hashemi	1539531
Juan Luengo	1555383
Luca Mainardi	2014602
Cristóbal Sendín	2025825
Aria Mostajeran	2019558

Eindhoven, July 18, 2024

1 | Question 1: Average user song rating using SparkSQL

In this question, we want to find the total number of all users that provided at least 100 ratings and had an average rating below 2. The SQL query achieving this objective can be represented as following:

```
SELECT COUNT(*) AS num_users
FROM
    (SELECT user_id
    FROM
        (SELECT user_id, AVG(rating) AS avg_rating
        FROM df1
        WHERE rating IS NOT NULL
        GROUP BY user_id
        HAVING COUNT(rating) >= 100 AND AVG(rating) < 2
        ) AS subquery
    ) AS final query
```

Figure 1.1: SQL Query Question 1

For implementing the same functionality in Spark, but without using SparkSQL, we would need to use RDD API and transform the data to process the data.

We would have to first work directly with the RDD obtained from the dataset. Then, as done in the real code, the records where the rating is null should be filtered out since we're only interested in users who have provided ratings. After that, we would transform the dataset into key-value pairs where the key is the user ID and the value is the rating. Now the goal would be to aggregate the ratings for each user, by calculating the sum of the ratings and the number of ratings. This will be useful later to calculate the average rating per user. After this aggregation, we have to filter out users who rated less than 100 songs. For each user, we would calculate the average rating by dividing the sum of ratings by the count of ratings. Finally, we would only keep the users whose average rating is below 2, and count how many there are.

Comparing the efficiency of Spark implementation and SparkSQL implementation, some factors must be taken into consideration. The first thing to take into account is that the SparkSQL query uses SQL's declarative nature, which helps optimize the execution (as it can inherently do things like push down filters and manage aggregation). So, from this point of view, the SQL approach could run faster. On the other hand, the RDD approach gives more control over the data processing steps as it requires manual optimization. This, apart from more control, results in an increased understanding of how data is processed and what optimizations can be done. In summary, in terms of efficiency, SparkSQL gives a performance advantage due to its optimization ability. The RDD approach, while more flexible and controllable, may not achieve the same level of efficiency without careful optimization due to the manual steps required to handle aggregations and filtering.

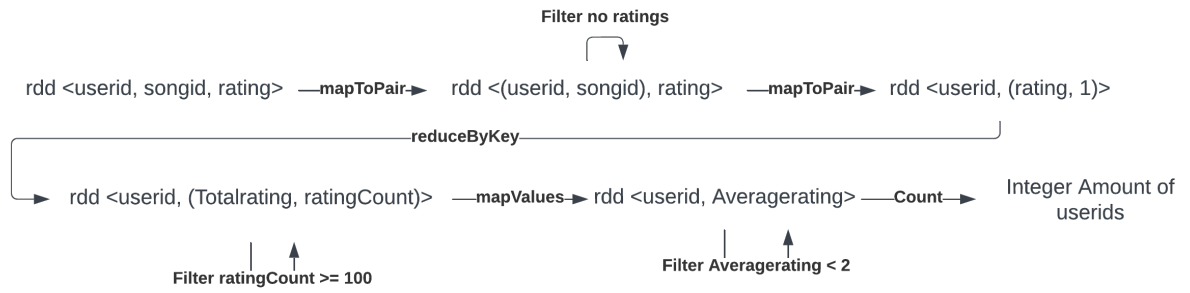

Figure 1.2: Architecture Spark Q1

Metric	Value
Q1	3
Q1 time	20975ms

Table 1.1: Query output

2 | Question 2: Finding the grumpiest user

As did before, we start by transforming the initial dataset into a form easier for analysis. For this, we create a PairRDD of $<(\text{userID}, \text{songID}), \text{rating}>$. This transforms the input RDD into a PairRDD mapping each user-song pair to its rating, helping to handle the cases where a rating is not provided and assigning a default value of -1. After this, entries without a rating are filtered to focus only on user-song pairs that have been explicitly rated. In addition, as indicated in Figure 2.1, we reduce by key to get the total rating and count for each user. This enables the calculation of average ratings by combining individual song ratings into user profiles. Furthermore, this is also done for the next part, in which we calculate the average ratings for users who meet the minimum criteria of having given at least 100 ratings, making sure that the analysis focuses on users with real engagement. Additionally, once the not so active users have been filtered out, the average rating for each user can be calculated by dividing the total sum of ratings by the count of rated songs. This gives as a result a PairRDD mapping each user to their average rating. Finally, the user with the minimum average rating among users with at least 100 is found, effectively finding the "grumpiest" user in the dataset.

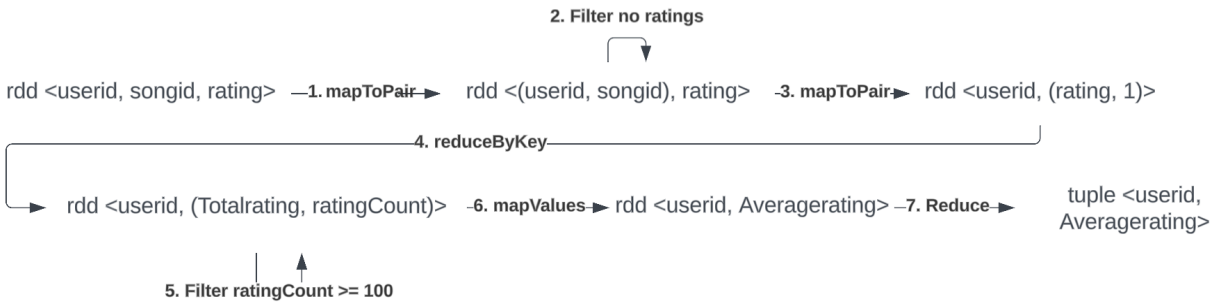

Figure 2.1: Architecture Q2

1. **mapToPair():** Create an RDD format (userid, songid), rating from RDD <String>
2. **filter():** Filter out the users with no ratings
3. **mapToPair():** Create a pair of (userid, (rating, 1))
4. **reduceByKey():** Aggregate values associated with user IDs
5. **filter():** Filter for only the users with a rating count of at least 100
6. **mapValues():** Calculate the average rating for each user
7. **Reduce():** Find the minimum average rating among users with at least 100 ratings

2.1 | Parallelism for performance improvement

The goal is to create parallelism by creating multiple worker treads. We want to utilise all available cores, so the number should not be too small. Too many repartitions leads to more overhead where too many small tasks need to be managed. As we see from the figure 2.2, the best amount of repartitions is equal to the amount of cores on the machine. More repartitions give a longer execution time for Question 2.

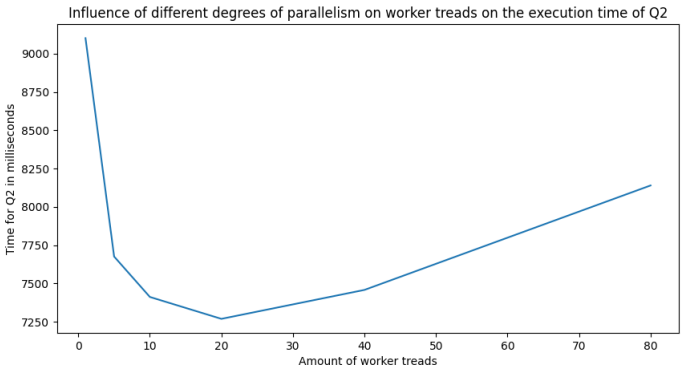

Figure 2.2: Influence of different degrees of parallelism on worker treads on the execution time of Q2

3 | Question 3: Recommending new song using Cosine Similarity

To identify similarities in song ratings for every pair of users, several steps are undertaken. Initially, similar to the approach detailed in Section 2, the initial RRD is transformed into a PairRDD. This transformation results in a structure of the form $<$ userID, (songID, rating) $>$, where any data rows lacking a rating are filtered out. Subsequently, the values within this constructed RDD, denoted as (songID, rating), are mapped into sparse vectors of size 2^{31} using the `mllib` Spark package, corresponding to the maximum song value within the dataset.

During the shuffling phase, these PairRDDs are reduced by key, yielding the sparse vectors representing the songs rated by each userID. The objective is to compute the cosine similarity for every pair of users. To accomplish this, the Cartesian operator is employed, generating combinations of users in the form $<$ (userID1, sparseVector1), (userID2, sparseVector2) $>$. To avoid duplications, the resulting pairs are filtered to retain only those where userID1 is less than userID2. Following this filtration, another `mapToPair` operation is executed to compute the cosine similarity, resulting in pairs of the form $<$ (userID1, userID2), cosineSimilarity $>$. Pairs with a cosine similarity below 0.95 are filtered out. The subsequent step involves reducing the data by key to determine the userID2 with which userID1 shares the maximum cosine similarity. Additional conditions specified in the assignment question are also checked at this stage.

Finally, the data is mapped to achieve the desired output format of $<$ userID1, userID2, cosineSimilarity $>$, and the results are collected as a list. The entire process is visually depicted in Figure 3.1.

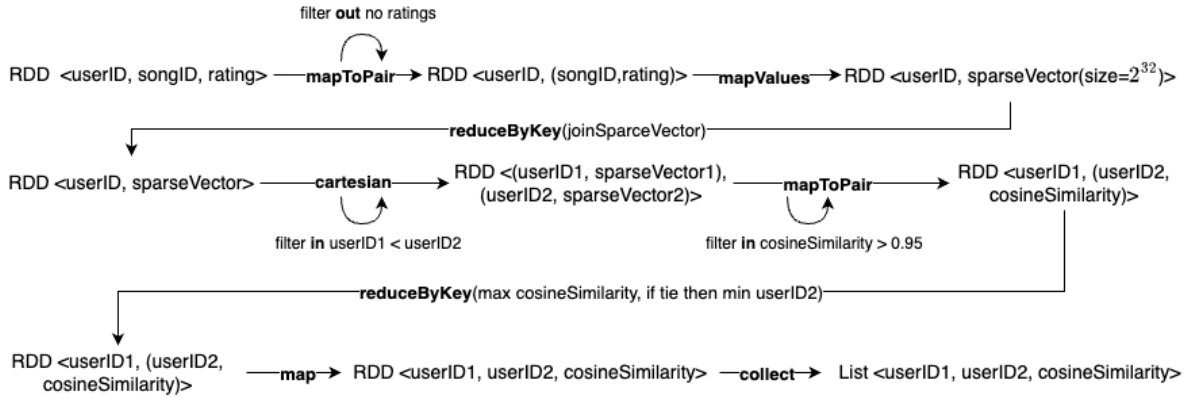

Figure 3.1: Architecture Q3

If two user pairs share the same cosine similarity scores, the tie is resolved by choosing the pair where the second user ID (`userID2`) is the lowest. This makes sure to always get a consistent outcome, preventing the possibility of arbitrary selections in cases with the same scores. Then, the final mapping transforms the data into an easy-to-read and understandable format, joining each unique user pair and their highest cosine similarity as a tuple $<$ `userID1`, `userID2`, `cosineSimilarity` $>$. This is then added to a list using the `collect` action.

4 | Question 4: Total number of distinct songs that users listened to

The Flajolet-Martin (FM) Sketch is a method known for approximating distinct counts through the use of hash functions applied to elements within a universal set, subsequently mapped to a bitmap. In our problem, the bitmap size is set at 32 bits which is determined by the logarithm of the maximum potential cardinality of 2^{32} songs. To construct an estimator using the FM Sketch, each songID undergoes hashing with a series of distinct hash functions over a specified number of repetitions, denoted as r. During each repetition, the corresponding index in an initially empty bitmap is set to 1 for every hash value generated. Given that there are r bitmaps for a single songID, merging these bitmaps involves taking the logical OR operation, indicating which songID has been counted for a given hash. Estimation within this framework involves identifying the mean position of the leftmost zero across these repetitions, represented as d, calculated as the sum of leftmost zeros divided by r. Subsequently, estimation is performed using the formula $1.3 * 2^d$. The determination of the number of repetitions, r, is governed by the expression $r = \frac{1}{\epsilon^2} \cdot \log\left(\frac{1}{\delta}\right)$, where epsilon ($\epsilon$) and delta ($\delta$) are set to 0.1. This calculation helps minimize errors within specific margins, pinpointing the ideal number of hash functions needed to reach the desired level of accuracy. The implementation of this sketch for the given problem is demonstrated in Figure 4.1 and the corresponding pseudo-code is presented in 1.

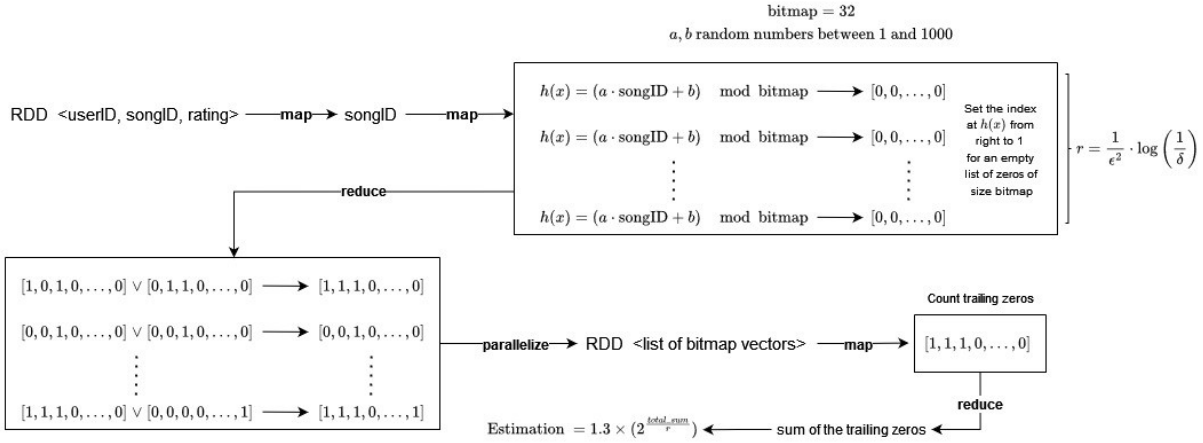

Figure 4.1: System architecture for Q4

Algorithm 1 EstimateDistinctFM(RDD, {h1(), . . . , hr ()}, bitmap_size)

Input: RDD of <songID>, randomizing hash functions $h_i(i = 1, ..., r)$, bitmap_size
Output: Estimate R of the number of distinct songIDs in the RDD

```
Begin
    bitmap_size = log(2^32)
    δ, ϵ = 0.1
    r = (1/ϵ²) · log(1/δ)
    h_i(i = 1, ..., r) = (ax + b)% bitmap_size for random a, b
    for i := 1 to r do
        bitVector_i[] := [0, ..., 0]                          ▷ of size bitmap_size
    for each songID ∈ S do
        for i := 1 to r do bitVector[h_i(songID)] := 1         ▷ from right to left
    for i := 1 to r do
        for m := bitmap_size down to 0                        ▷ counting trailing zeros from right to left
            if bitVector_i[m] = 0 then leftmostZero := m
        sum := sum + leftmostZero
    end for
    R := 1.3 × 2^(sum/r)
    return (R)
end
```

Epsilon (ϵ)	Delta (δ)	Number of hash functions (r)	Computation Time	Song Estimation
0.05	0.05	1199	108859 ms	82129
0.1	0.1	231	21787 ms	140867
0.1	0.3	121	12300 ms	83852
0.3	0.1	26	3745 ms	166838
0.3	0.3	14	2837 ms	447155

Table 4.1: Performance table for varying Delta and Epsilon

We conducted several experiments with varying delta and epsilon values to measure the performance and the outcomes of the estimated songs. The table 4.1 summarizes these findings. What is expected to happen is that, as we increase the delta and epsilon values, the computation time should decrease since the number of hashes also decreases, resulting in faster computations. However, as the number of hashes decreases, accuracy gets compromised, as the song estimations do increase significantly, showing over-estimations. It is important to note that these values are not exactly reproducible, as there is an inherent randomness associated with the hash functions.

5 | Question 5: Triplets of users that have cumulatively rated at most 8000 distinct songs

We begin by creating a PairRDD which maps each user-song pair to its rating, as we did in Q1 and Q2. This is achieved by parsing the input dataset and mapping each entry to a tuple of the form $< (\text{userID}, \text{songID}), \text{rating} >$. We filter out the entries without ratings, considering only those with valid ratings, and remove the ratings values, which is not needed for this question. Thus, we obtain a PairRDD of $< \text{userID}, \text{songID} >$. Next, we transform the PairRDD to represent rated songs using a BitSet (containing a single song). This will allow us to efficiently compute the distinct count of songs for each user. We then reduce by key to merge the song sets for each user, through a logical OR operation on the BitSets. In this way we create a PairRDD of $< \text{userID}, \text{BitSet} >$, in which each user is associated with a BitSet which represents the list of all the songs he has reviewed. We filter out users whose cumulative number of distinct songs exceeds the limit of 8000, as they cannot belong to triplets with less than 8000 distinct songs, as required. To count the number of distinct songs, simply count the number of cells in the BitSet whose value is 1. We calculate the Cartesian product to generate all possible user pairs (keeping only the pairs where userID1 < userID2, as required) and map the resulting RDD into a new PairRDD of $< (\text{userID1}, \text{UserID2}), \text{BitSet} >$, where the BitSet represents the list of distinct songs voted by the two users, also obtained in this case with a bitwise OR between the users' BitSets. Now, we filter out pairs where the cumulative distinct song count exceeds 8000, as we did before. Similarly, we compute the Cartesian product between user pairs and valid users, to generate all possible user triplets along with their merged song sets. Again, we filter out triplets where the cumulative distinct song count exceeds 8000. Finally, we map the result to an RDD of $< \text{userID1}, \text{userID1}, \text{userID1}, \text{totalRatings} >$, which contains three user IDs representing a valid triplet, along with the distinct count of songs for that triplet.

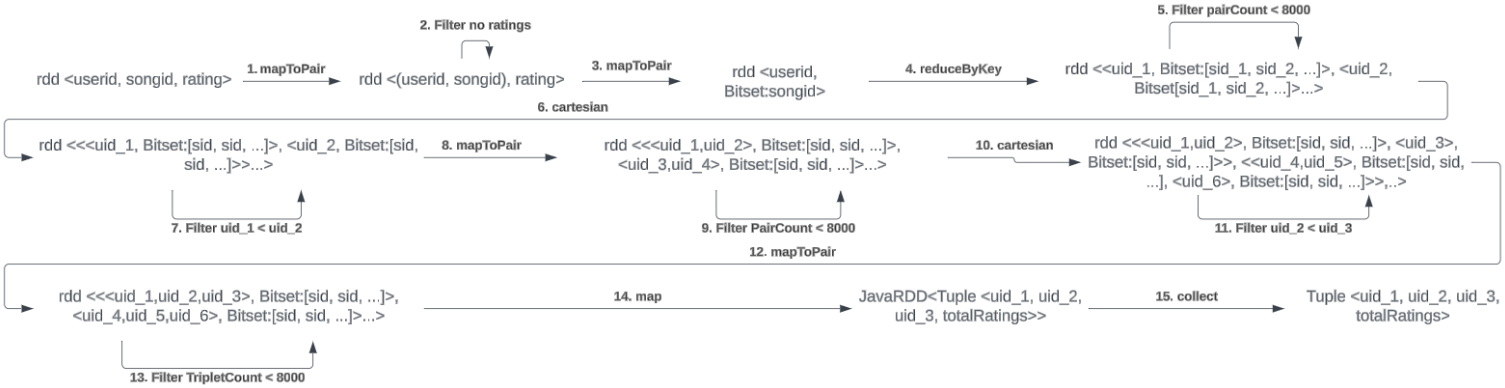

Figure 5.1: Architecture Q5

1. **mapToPair():** Create a PairRDD of the format (userID, songID) → rating from the input RDD of strings. <String>
2. **filter():** Filter out the users with no ratings.
3. **mapToPair():** Transform each rated entry into a pair of the form (userID, BitSet(songIDs)).
4. **reduceByKey():** Aggregate song sets associated with each user, merging them into a single BitSet.
5. **filter():** Retain only those users whose distinct song count is less than 8000.
6. **cartesian():** Compute the Cartesian product of valid user-song pairs, generating all possible pairs of users along with their sets of songs.
7. **filter():** Retain only user pairs (userid1, userid2) where userid1 <userid2
8. **mapToPair():** Create a Tuple of user pairs along with their merged sets of songs.
9. **filter():** Remove pairs where the cumulative distinct song count exceeds 8000.
10. **cartesian():** Compute the Cartesian product of valid user pairs with valid user-song pairs, generating all possible triplets of users along with their merged sets of songs.
11. **filter():** Retain only user triplets (userid1, userid2, userid3) where userid2 <userid3
12. **mapToPair():** Create a Tuple of user triplets along with their merged sets of songs.
13. **filter():** Use filter() to remove triplets where the cumulative distinct song count exceeds 8000.
14. **map():** Extract the distinct song count for each user triplet, resulting in an RDD of the format (userID1, userID2, userID3, distinctSongsCount).
15. **collect():** Materialize the results into a list of tuples.