CS422: Database Systems Project-2 Report

Hussein Osman
SCIPER N° 336774

May 31, 2022

1 Task 1: Bulk-load Data

This task is straight-forward. We simply load the data into an *RDD* (Resilient Distributed Dataset), split the text file, and finally do the necessary castings according to the task description.

2 Task 2: Average Rating Pipeline

2.1 Task 2.1: Rating aggregation

In this sub-task, we are mainly concerned about two functions `init()` and `getResult()`. In the initializer, we perform a left outer join between the titles and the ratings. We first map the two RDDs to have a common key field so we can perform a join operation. Accordingly, we choose the key to be the movie ID. We also note that we keep the latest of two same user-movie rating pairs based on the timestamp.

In the `getResult()` function, we use the result from the full outer joined RDD, and reduce this RDD by keeping track of the sum and the count of the values corresponding to each key. Finally we map these `(key, (sum, count))` pairs to `(key, average)` pairs, and discard all unnecessary information.

2.2 Task 2.2: Ad-hoc keyword roll-up

In this sub-task, we follow the same strategy as in Task 2.1, but we perform an inner join between the titles and ratings RDDs (after mapping them to have a common key). If the result of the inner join is `null`, then the list of the keywords in the query does not have any return results. Otherwise, we filter the result by selecting only the rows `(movie, rating)` which include the query keywords as a subset of the keywords of the movie. Finally, we roll-up these ratings and return the average value or -1 if there are no qualified tuples.

2.3 Task 2.3: Update Result

In this sub-task, we take into consideration that a new movie-rating pair should shadow its old pair for the same user. The old movie-rating pair should not be used to compute the aggregate. The aggregate value is computed pretty much the same as we did in Task 2.2. In case the incremental update contains a movie-rating paper for a new user, or for a movie that has not been rated before, then we add "`None`" to the old rating field.

3 Task 3: Similarity Search Pipeline

3.1 Task 3.1: Indexing the dataset

We perform an inner join between the movie RDD and the hash values RDD for each movie. The inner join is done based on the list of keywords, resulting in an `RDD(IndexedSeq[Int], List[(Int, String, List[String])`, after applying the necessary map and reduce transformations. The result is bucketed and grouped together.

3.2 Task 3.2: Near-neighbor Lookups

We simply perform a left-outer join between the queries RDD and the buckets RDD, and return the result.

3.3 Task 3.3: Near-neighbor Cache

In this sub-task, we create two RDDs, one for cache hits and the other for cache misses. If the cache is `null`, we mark all the queries as *misses*. Otherwise, we join the queries and the cache, and check if there are *hits*. We return the hits, and then forward the misses to the *LSH* lookup function, and retrieve the results. Finally, we union both results and return them to the driver.

3.4 Task 3.4: Cache Policy

We intercept the cache Lookup to perform an additional preprocessing step by creating a histogram of the hashed values. This histogram is built or updated whenever a query is sent for cache Lookup.

Elsewhere, whenever we are building the cache, we check this histogram for queries (i.e. movie keywords after hashing) that are have been frequently requested (more than 1%). If this is the case, the cache keeps these keywords hash values, and discards all other ones. That's how the cache favors frequently requested queries over less frequent ones to maximize the number of hits per query and minimize the number of misses per query.