CS-422 Database Systems Project II

Stefan Igescu

June 1, 2022

Task 1: Bulk-load data

In order to implement the **load()** method for both TitlesLoader and RatingsLoader I proceeded in this way:

- First I load the .csv using sc.textFile()
- Second I split each row in the file based on the separator '|'
- Third I select the desired values from each row splitted.

In particular for the RatingsLoad it is needed an extra step in order to obtain the previous rating for each row:

- Tuples are grouped by user id and title id
- Each group is sorted by increasing timestamp
- Each sorted group is traversed. The first tuple has a **Double.NaN** as previous rating. All the others use the rating score of the previous tuple visited as previous rating.

Finally both RDDs are persisted in memory by calling the method **persist()**

Task 2: Average rating pipeline

Task 2.1: Rating aggregation

This task has been implemented by using an internal **state** to the Aggregator class which collects for each title the following informations: title id, title name, sum of all ratings for this title, number of all ratings for this title, list of keywords. The state is built in the **init()** method.

- First I select only the last rating for each title for each user on the ratings RDD. This is done by grouping by user id and title id and then collecting the latest tuple by using the timestamp.
- Second the remained tuples are grouped only by title id. Then I aggregate them by computing: sum of all ratings for each title, number of all ratings for each title.
- In the end I join the result with the title RDD in order to add also title name and the list of keywords.

Finally, when the method **getResult()** is called the state is manipulated through a map in order to compute the average rating (by dividing the sum of ratings by the total count for each title) and in order to retrieve only the desired fields.

Task 2.2: Ad-hoc keyword rollup

I again use the **state** in order to implement this section. I start by filtering it, keeping only the tuples that contain the desired keywords. Then I count how many tuples remained after the filter. If none of them remains then I return -1.0. Otherwise I filter the result again by keeping only those tuples that have a number of ratings different than zero. Again, if none of the tuples remains then I return 0.0, otherwise I compute the average over all the averages with **aggregate**.

Task 2.3: Incremental maintenance

In order to batch-update the internal state, I map each tuple of the state to the updated one. In particular, for each tuple of the state I filter *delta* in order to keep only the new ratings that have the same title id of the title present in tuple. If the number of new ratings is 0 then I return the tuple as-is, since no update is present. Otherwise I iterate over the new ratings remained and:

- If the new rating has a previous rating value, therefore I update only the total sum of ratings in tuple by subtracting the old rating and by adding the new one. The number of ratings remains the same.
- If previous rating is empty, therefore this is the rating of a new user that never rated this title before. Therefore I add the new rating to the total sum, and I update the total number of ratings by adding +1.

In the end I persist the new state and unpersist the old one by calling **persist()**.

Task 3: Similarity-search pipeline

Task 3.1: Indexing the dataset

This subsection is implemented in the method **getBuckets()** inside the **LSHIndex** class. First I compute the signature for each title by extracting the keywords with a map and then by passing them to the **hash()** function. Secondly, I cluster the titles in buckets in the following way:

- For each title I keep only title id and title name with a map
- I zip the resulting tuples with the signature RDD computed previously, this way for each tuple I have: title id, title name, keywords and signature.
- Finally I group the tuples by signature.

Task 3.2: Near-neighbour lookups

In order to implement the processing of batches of near-neighbor queries, two functions have been implemented:

- **lookup()** in the class **NNLookup** which is the most external function and the first to get called. It receives a batch of queries, where each query is expressed as a list of keywords, and it applies the **hash()** function on them obtaining an RDD of pairs of signature and keyword list. Afterwards it passes this result to the second function **lookup()** of the class **LSHIndex**.
- **lookup()** of the class **LSHIndex** receives the hashed queries. First it obtains the bucketization of the titles by calling **getBucket()** described in the previous step. Then it joins the bucket and the hashed queries based on the signature. This way we associate each query with the respective collection of titles based on the signature. Finally, after the join, it selects the relative fields with a map and it returns an RDD containing tuples in the form (signature, query, list of related titles).

Task 3.3: Near-neighbour cache

The implementation of this sections is similar to the previous but for the presence of the cache. Again two functions have been implemented in order to retrieve the near-neighbour titles:

- **lookup()** in the class **NNLookupWithCache** which again is the most external function and the first to get called. In this case the first thing it does is to call **cacheLookup()** in order to check the cache. It obtains then two RDDs: cacheHit with all the neighbours retrieved from the cache, and cacheMiss with the queries that didn't find anything in the cache. Afterwards it retrieves the related tuples for cacheMiss by calling **lshIndex.lookup()** and saving the result in cacheMissResult. Finally in case cacheHit is empty because then it returns only cacheMissResult. Otherwise it combines cacheHit with cacheMissResult and returns it.

- **cacheLookup()** in the class **NNLookupWithCache** which implements how the cache is searched. It first computes the signatures of the queries. Then, if the cache is not defined it simply returns as cacheHit the value *null* and as cacheMiss the whole RDD of signatures. Otherwise it creates cacheMiss by filtering all the signatures that are not present in the cache, and it creates cacheHit by filtering all the signatures that are in the cache and it retrieves the respective values (i.e the near-neighbours).
  In this context the cache is represented as a Map which has as keys the signatures and as values the near-neighbours.

Finally there is also the function **buildExternal()** which force a cache based on the given object that is given. It simply saves the external object in the variable cache.

Task 3.4: Cache policy

In order to implement the bookkeeping the variable **histogram**, that is a simple RDD of tuples (signature, count), is created. **histogram** gets updated each times **cacheLookup()** gets called: after computing the signatures of the queries, they get grouped by signature, and for each signature the respective count is computed, then the new counts are added to the histogram if it is defined (this is implemented with a Full Outer join between the new counts and the previous histogram), otherwise these counts create the new histogram.
The cache creation is implemented in **build()**. First of all the total number of counts is computed from the histogram. Afterwards all the signatures in histogram that occur in more than 1% of the queries are kept and sent to **lshIndex.lookup()** which will retrieve the respective neighbours. Finally the result is collected as Map (with keys corresponding to signatures and values corresponding to neighbours) which is then broadcasted using the function **sc.broadcast()**. In the end the histogram is reset by assigning the value null.