CS-422 Database Systems

Project II

Deadline: 1st of June 23:59

The aim of this project is to expose you to a variety of programming paradigms that are featured in modern scale-out data processing frameworks. In this project, you have to implement data processing pipelines over **Apache Spark**.

We provide a skeleton codebase on which you will implement missing functionality. We also provide test cases that will check the functionality of your code. You may find the skeleton code in `https://gitlab.epfl.ch/DIAS/COURSES/CS-422/2022/Project-2-<ID>`.

In what follows, we will go through the functionality that you need to implement.

Overview

You are given a dataset based on IMDB (Internet Movie Database). Your dataset includes:

- A set of titles (e.g. movies, TV series) and their related keywords
- A log of "user rates movie" actions

A video streaming application requires large-scale data processing over the given dataset. The application has a user-facing component (which you are not concerned with) that serves recommendations and interesting statistics to the end-users. Specifically, for each title, it displays:

- All the movies that are "similar" in terms of keywords
- A differentially-private average rating

In the background, the application issues Spark jobs that pre-compute the information that is served. Your task is to write Spark code for the required functionality. You will find more information in the corresponding tasks.

Dataset

You are given two pipe-separated ('|') files that contain the initial data. For each file, you need to build suitable objects that correspond to records. We provide the classes of objects in the skeleton code.

titles.csv: In each row, this file contains the id and name of each title followed by a list of associated keywords. For example, suppose the n_{th} title has k keywords, then the n_{th} line of the file is:

$$id_n|name_n|keyword_{n1}|keyword_{n2}|\dots|keyword_{nk}$$

The number of keywords varies from title to title.

Each row is converted to a $(Int, String, List[String])$ tuple, in which the first element is the title's id, the second element is the title's name and the third element is the title's keywords.

ratings.csv: In each row, this file contains the id of a user, the id of a title, a rating, and a timestamp. For example, the m_{th} line of the file is:

$$id_m|id_m|rating_m|timestamp$$

Each row is converted to a $(Int, Int, Option[Double], Double, Int)$ quadruplet, in which the first element is the user's id, the second element is the title's id, the third element is the old rating from this user for the title ($None$ if no previous rating), the fourth element is the new rating, and the last element is a timestamp for the rating action.

Apart from the initial batch load, additional ratings are appended to the data at runtime. The updates are given as arrays of the corresponding object type.

[Note: you will still have to transform the objects into the appropriate data structure for each task]

Task 1: Bulk-load data

You need to implement the **load()** methods for **TitlesLoader** and **RatingsLoader**, which load data from **titles.csv** and **ratings.csv** respectively. Your code reads the CSV files and convert rows into the corresponding tuples. After converting the data into the appropriate RDD, **load()** also persists the RDDs in memory to avoid reading and parsing the files for every data processing operation.

Task 2: Average rating pipeline

The application needs to display an average rating for each title. This pipeline aggregates ratings in order to compute the average rating for all available titles (and display 0.0 for titles that haven't been rated yet). Furthermore, it enables ad-hoc queries that compute the average rating across titles that are related to specific combinations of keywords. Finally, when a batch of new ratings is provided by the application, the pipeline incrementally maintains existing aggregate ratings. All the functionality is implemented in the **Aggregator** class. In the rest of the Section, we discuss how the pipeline is implemented.

Task 2.1: Rating aggregation

The ratings log includes all the "user (re-)rates movie" actions that the application has registered. The **init()** computes the average rating for each title by aggregating the corresponding actions. Titles with no ratings have a 0.0 rating. For each title in the aggregate result, you need to maintain the related keywords. **init()** persists the resulting RDD in-memory to avoid recomputation for every data processing operation.

To retrieve the result of the aggregation, the applications need to use the **getResult()** method which returns the title name followed by the rating.

[Note 1: Keep in mind that there can be more than one ratings for each user-title pair.]

[Note 2: There are multiple possible implementations for the aggregation and multiple representations for the result.]

Task 2.2: Ad-hoc keyword rollup

The **getKeywordQueryResult()** implements queries over the aggregated ratings. The parameter of the queries is a set of keywords and the queries compute the average rating among the titles that contain all of the given keywords. The average of averages excludes unrated titles and all titles contribute equally. The result is returned to the driver which then returns it to the end-user. If all titles are unrated the query returns 0.0, whereas if no titles exist for the given keywords the query returns -1.0.

Task 2.3: Incremental maintenance

The application periodically applies batches of append-only updates in the batches in the **updateResult()** method. Updates are provided as arrays of *rating* tuples accordingly. Given the aggregates before the update and the array of updates, the method computes new updated aggregates that include both old and new ratings. You need to persist the resulting RDD in memory to avoid additional recomputation and unpersist the old version.

Hint: Incremental maintenance can use the previous rating element to reduce recomputations.

Task 3: Similarity-search pipeline

The application identifies titles that are "similar" each user's history to make recommendations. In order to retrieve "similar" titles, it batches a large number of near-neighbor queries i.e. queries that return titles with similar keywords. Then, using locality-sensitive hashing (LSH), it computes the near-neighbors, which are pushed to interested users.

In the rest of the Section, we describe what you need to implement concretely. First, we provide a brief introduction of LSH, which contains all the required information for implementing the project. Task 2.1 describes indexing titles to support distributed LSH lookups. Task 2.2 describes the implementation of distributed LSH lookups themselves. Task 2.3 describes a caching mechanism to reduce the cost of lookups. Finally, Task 2.4 describes a simple cache policy which you will implement in the project.

Background: Locality-sensitive Hashing

Often, we need to compute how "similar" two sets of objects are. Intuitively, the set $\{action, thriller, crime\}$ is not similar to $\{comedy, family\}$, whereas it is somewhat similar to $\{action, thriller, superhero\}$. For this reason, several similarity metrics have been proposed. One commonly used metric is Jaccard similarity, which is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

When $J(A, B)$ is close to 1 the sets are similar, and when $J(A, B)$ is close to 0 the sets are not similar. Usually, a threshold t is used as a cut-off point (similar iff $J(A, B) > t$, not-similar otherwise). Given a set A, the near-neighbor problem is defined as finding all sets B such that $J(A, B) > t$ for a given threshold t.

However, computing all similar sets is expensive. Let D be the set of all sets of objects. Then, computing all similar sets requires computing all pairwise Jaccard similarities which are $O(|D|^2)$. Furthermore, for computing each $J(A, B)$, the cost is at least $O(max(|A|, |B|))$. Therefore, exhaustively computing all Jaccard similarities does not scale for large and high-dimensional datasets.

Consequently, researchers have proposed approximate solutions for the near-neighbor problem. A popular algorithm is locality-sensitive hashing (LSH). LSH uses a hash function h that computes a *signature* $h(A)$ for each set A. The hash function is chosen such that:

- if $J(A, B) \leq t$, then $h(A) = h(B)$ with probability at least p_1
- if $J(A, B) \geq (1 + \epsilon)t$, then $h(A) = h(B)$ with probability at most p_2

LSH, thus, uses each set's signature to retrieve all other sets with the same signature in asymptotically lower time. Even though the retrieved sets contain some *false positives* and some *false negatives*, they mostly consist of *true positives* when the hash function is effective.

In the context of this project, you are not concerned with choosing and tuning the hash function; we already provide the hash function to you. Instead, you need to focus on matching the signatures.

The following subtasks guide you through implementing and optimizing an LSH pipeline.

Task 3.1: Indexing the dataset

To identify signature matches efficiently, data needs to be organized accordingly. The **LSHIndex** class restructures the title RDD so that subsequent lookups can find matches with the same signature efficiently. This requires the following steps, which are implemented in the constructor:

1. **Hashing:** In the first step, you need to use the provided hash function to compute the signature for each title. The signature is used to label the title thereupon.
2. **Bucketization:** In the second step, you need to cluster titles with the same signature together. Suppose your RDD consist of the following five annotated titles:

```
(555, "LotR", ["ring"], 5)
(556, "Star Wars", ["space"], 3)
(557, "The Hobbit", ["ring"], 5)
(558, "Inception", ["dream"], 1)
(559, "Star Trek", ["space"], 3)
```

Then, bucketization should result in the following data structure:

```
(1, [(558, "Inception", ["dream"])])
(3, [(559, "Star Trek", ["space"]), (556, "Star Wars", ["space"])])
(5, [(557, "The Hobbit", ["ring"]), (555, "LotR", ["ring"])])
```

3. **Partitioning (Optional):** The result of bucketization is distributed. Therefore, each lookup is directed to one specific cluster of titles. To improve efficiency and reduce data movement at runtime, you need to partition the data structure by signature. Moreover, you need to cache the result of the partitioning for subsequent uses.

Task 3.2: Near-neighbor lookups

At runtime, the application submits batches of near-neighbor queries for processing. Each query is expressed as a list of keywords. Computing the near-neighbors requires the following steps:

1. **Hashing:** You need to use the provided hash function to compute the signature for each query. The signature is used to label the query thereupon.
2. **Lookup:** The signature is used to retrieve the set of titles with the same signature as the query. This will result in a shuffle for the given queries.

You need to implement the **lookup()** method in **LSHIndex** which performs signature matching against the constructed buckets. Also, you need to implement **lookup()** in class **NNLookup** which handles hashing and also uses **LSHIndex.lookup()** for matching.

Task 3.3: Near-neighbor cache

Each batch of queries requires shuffling the queries which is expensive. For this reason, a cache is added to the pipeline. The cache is broadcasted/replicated across Spark workers and includes the near-neighbors from frequently requested signatures. Then, each worker can immediately compute the results of its local queries that hit the cache. Distributed lookups occur only for queries that result in cache misses. The results are merged and returned to the applications.

You need to implement the logic for exploiting the cache in **cacheLookup()** method in class **NNLookupWithCache**. Specifically, you need to use the cache to separate queries that result in a cache hit and queries that result in a cache miss. Then, for queries that hit the cache, you need to retrieve and return the near-neighbors. If there is no cache, the RDD of hits needs to be *null*.

Moreover, you also need to implement the **lookup()** in **NNLookupWithCache**, which combines **cacheLookup()** and the **lookup()** method of **LSHIndex** to compute the full result for the queries.

Finally, for testing purposes, you need to implement the **buildExternal()** method of **NNLookupWithCache** which receives a broadcast object (you can use it as it is or transform it to your preferred format) that should be used as a cache for testing purposes.

Task 3.4: Cache policy

The cache needs to be populated and distributed to the workers. In this subtask, you need to implement two components:

1. **Bookkeeping:** You need to measure how frequently each signature occurs among queries. You can assume that the number of distinct signatures is low and that a histogram can fit in memory. Your code needs to intercept the signatures of the queries in **cacheLookup()**, and update a histogram for occurrences of signatures.
2. **Cache creation:** The cache creation algorithm computes signatures that occur in more that 1% ($> 1\%$) of the queries. Then, the buckets for these frequent signatures are retrieved, merged into one data structure and broadcasted to workers. Subsequently, the histogram data structure is reset. The algorithm should be implemented in the **build()** method of **NNLookupWithCache**.

Infrastructure

You can fully implement and test your implementation locally, on your own computer. However, we will also set up a 10-node cluster on IC Cluster for you to run your Spark programs (we will give you access and usage details during the second week of your project). We highly recommend trying to run your program in the cluster as well because you will get the chance to experiment with HDFS and distributed Spark.

Deliverables

We will grade your last commit on your GitLab repository. Your implementation must be on the **master** branch. **You do not submit anything on moodle**. Your repository must contain a README file, **README.pdf**.

Grading: Keep in mind that we will test your code automatically. Do not change the interfaces of the provided methods.