CS422 - Database Systems: Project 2 Report

Andrej Milicevic

May 12, 2022

Task 1 - Bulk-load data

This task consisted of simply reading the files containing movie titles and ratings of those movies and converting each line into the appropriate data type. I read the lines directly into an RDD, which is persisted at the end of the function. Both `load()` functions in `RatingsLoader` and `TitlesLoader` are basically the same.

Task 2 - Average rating pipeline

The average rating for each title was calculated in this task, with extra functionalities such as calculating the average rating of movies related to specific keywords and incremental maintenance of the aggregate ratings.

Subtask 2.1 - Rating aggregation

The `init()` method contains the aggregation of ratings for each movie title, where the average rating for titles which have not been rated yet is 0.0. The way I implemented this is I first found the ids of films which have not been rated yet and put them in the `zeroRatings` RDD, and then appended it to the RDD which contains the rest of the aggregated values. The final result is stored in the `state` RDD, which is persisted at the end of the `init()` method. The following code shows this implementation:

```
def init(
           ratings : RDD[(Int, Int, Option[Double], Double, Int)],
           title : RDD[(Int, String, List[String])]
         ) : Unit = {

  val ratingsId = ratings.map(el => el._2)
  val titleId = title.map(el => el._1)
  val zeroRatingsTmp = titleId.subtract(ratingsId)
  val zeroRatings =  zeroRatingsTmp.map(el => (el, (0.0, 0)))

  val aggregated = new PairRDDFunctions(ratings.groupBy(el => el._2))
     .aggregateByKey((0.0, 0))((a, b) => {
       var sum = 0.0;
       b.foreach(x => sum = sum + x._4)
       (a._1 + sum, a._2 + b.size)
    }, (c, d) => (c._1 + d._1, c._2 + d._2)).union(zeroRatings)

  state = aggregated.join(title.keyBy(el => el._1))
         .map(el => (el._1, el._2._2._2, el._2._1, el._2._2._3))
  state.persist()
}
```

I did not keep the average in `state`, but rather the sum and number of all ratings that exist for each title. The average is calculated in the `getResult()` method before returning the required RDD. The following code is for the `getResult()` method:

```
def getResult() : RDD[(String, Double)] = {

  state.map(el => {
    if (el._3._2 == 0) {
      (el._2, 0.0)
    }else {
      (el._2, el._3._1 / el._3._2)
    }
  })
}
```

Subtask 2.2 - Ad-hoc keyword rollup

In this part, the `getKeywordQueryResult(keywords)` method is implemented, which calculates the average rating of movies that contain all the keywords from the method parameter. The method returns -1.0 in case there are no movies matching the given keywords. Firstly, all movies from the previously persisted RDD are filtered based on the keywords, and then the average is computed in a similar way to Task 2.1, where the sum and number of ratings are calculated, and the average is then returned.

Subtask 2.3 - Incremental maintenance

The `updateResult(newRatings)` method implements the updating of calculated averages in the case we get new ratings. In order to avoid recomputation, all this method does is compute the sum (`deltaSum`) that needs to be added (or subtracted in case it is negative) and the number of new ratings (`deltaCount`) that occured (by new, we only consider the user-title pairs that did not exist yet). After computing that, the `deltaSum` and `deltaCount` are added to the previous sum and count for each title, and the new RDD is persisted (the old one is unpersisted at the beginning). The code that implements this part:

```
def updateResult(delta_ : Array[(Int, Int, Option[Double], Double, Int)])
                 : Unit = {

  state.unpersist()
  val delta = delta_.groupBy(el => el._2)
  state = state.map(el => {
    var deltaSum = 0.0
    var deltaCount = 0
    if (delta.contains(el._1)) {
      delta(el._1).foreach(r => {
        if (r._3.nonEmpty) {
          deltaSum = deltaSum + r._4 - r._3.get
        }else {
          deltaSum = deltaSum + r._4
          deltaCount = deltaCount + 1
        }
      })
    }
    (el._1, el._2, (el._3._1 + deltaSum, el._3._2 + deltaCount), el._4)
  })
  state.persist()
}
```

Task 3 - Similarity-search pipeline

This task sees the implementation of searching and retrieving similar movie titles, based on the keywords provided.

Subtask 3.1 - Indexing the dataset

This subtask organizes the data according to the keys obtained from hashing the title keywords. This is done in the constructor of the `LSHIndex` class, and I implemented it by first hashing the keywords from all titles in order to get the corresponding signatures. After that, I transformed the data into the appropriate type of RDD, partitioned it by signature and cached it in order to improve efficiency. The `getBuckets()` method just returns the cache. The code for this part is the following:

```
// distinct() is needed because there is a chance to have same films
// appear twice due to the way I keyed and joined the elements
val hashed :
RDD[(List[String], (IndexedSeq[Int], (Int, String, List[String])))] =
  hash(data.map(el => el._3))
    .keyBy(el => el._2)
    .map(el => (el._1, el._2._1))
    .join(data.keyBy(el => el._3))
    .distinct()
// cache for the partitioning by signature
val cache : RDD[(IndexedSeq[Int], List[(Int, String, List[String])])] =
  hashed.map(el => (el._2._1, el._2._2))
    .groupBy(el => el._1)
    .map(el => (el._1, el._2.map(x => x._2).toList))
    .partitionBy(new HashPartitioner(hashed.getNumPartitions)).cache()
```

Subtask 3.2 - Near-neighbour lookups

After constructing the appropriate data structure, we can implement the near-neighbour lookup. This is done in the `lookup(queries)` methods of the `LSHIndex` and `NNLookup` classes. The method in `NNLookup` hashes the queries using the `hash()` method from `LSHIndex` and calls the `lookup(queries)` method from `LSHIndex`. I implemented this second lookup by joining the received queries with the data from `getBuckets()` and transforming it into the required data type.

Subtask 3.3 - Near-neighbour cache

This part is similar to the previous one, only this time there is a cache which can be checked in order to return results faster. This is implemented in the NNLookupWithCache class. I implemented the cacheLookup(queries) method by hashing the received queries, and then dividing the result into two RDDs, hit and missed, which represent the queries that hit and miss the cache, respectively. This I obtained by filtering the hashed queries based on whether they were in the cache or not. Before doing this, however, I check whether there is a cache, and also update the histogram of signatures which is needed for the next task. I implemented the lookup(queries) method of this class by calling cacheLookup(queries), using the lookup from LSHIndex for the queries that missed the cache, and then returning the union of the two lookups. Also, the buildExternal(Broadcast[Map[]]) method is implemented by just putting the broadcasted map in the variable cache. The code for the cacheLookup(queries) method is the following:

```
def cacheLookup(queries: RDD[List[String]])
  : (RDD[(List[String], List[(Int, String, List[String])])],
     RDD[(IndexedSeq[Int], List[String])]) = {

  val sc = queries.sparkContext
  val hashed = lshIndex.hash(queries)

  if (cache == null){
    return (null, hashed)
  }
  // Updating the histogram
  count = count + hashed.count()
  hashed.foreach(el => {
    if (!histogram.contains(el._1)) {
     histogram += ((el._1, 0))
    }
    histogram(el._1) = histogram(el._1) + 1
  })

  val hashedList = hashed.collect().toList
  val cacheMap = cache.value

  val hit = sc.makeRDD(hashedList
              .filter(el => cacheMap.contains(el._1))
              .map(el => (el._2, cacheMap(el._1))))

  val missed = sc.parallelize(hashedList
                                          .filter(el => !cacheMap.contains(el._1)))

  (hit, missed)
}
```

Subtask 3.4 - Cache policy

The way I implemented the `build()` method for this task is by creating the histogram as a `Map[IndexedSeq[Int], Int]`, which contains the number of occurences in queries for each signature. Based on that number and the number of all queries (variable `count`), I check which queries appear more than 1% of the time, get the corresponding data from the `getBuckets()`, merge the data, and broadcast it to all the workers. The code for this task is the following:

```
var histogram : mutable.Map[IndexedSeq[Int], Int]
                  = mutable.Map[IndexedSeq[Int], Int]()
var count : Long = 0

def build(sc : SparkContext) = {

  val frequent = histogram.filter(el => el._2 / count > 0.01)
                          .keys.toList
  val data = lshIndex.getBuckets()
  cache = sc.broadcast(data.filter(el => frequent.contains(el._1))
                           .collect().toMap)
  histogram.clear()
  count = 0
}
```