Project report

Introduction

In this project we have used dataset obtained from Netflix, in this project we have tried to code efficient Map reduce code that will scale up properly on multiple machines. We are a team of 3 people so we have taken at least one major task or sometimes more:

1. **Hbase as index:** In this task we have implemented 3 joins; Hbase join, Reducer side join and Replicated join. We used join to get Average rating of a movie on yearly basis given by users.
   We have also used this result to classify movies in 4 clusters.
   - Ones that are highly watched and rated good by users.
   - Ones that are highly watched and rated not so good by users.
   - Ones that are not watched by many users but rated good by them.
   - Ones that are not watched by many and not rated well by users.

   Similarly we have divided the users into 4 categories:
   - Ones those who watch lot of movies and gives good ratings
   - Ones those who watch a lot of movies and gives not so good ratings, we can call them critics
   - Ones who do not watch a lot of movies but gives good ratings
   - Ones who do not watch a lot of movies and also do not give good ratings.

2. **Kmeans:**
   One of our project topics is k-means clustering which clusters data based on similarity on certain characteristics of the objects. We performed k-means clustering on data received from our HBASE task which populated data in the following format:

Movie name	Year of rating	Average Rating
Gladiator	2000	4.4
Rambo I	1998	3.2
The Silence of the Lambs	2003	2.0
Django	1996	5.3

   The data above indicates the average rating of a movie in that particular year. Our task is to find out movies of a particular year having identical ratings and clustering them together based on our initial clusters. This will give us all movies which have been identically rated in that particular period of time and cluster them together.

   We perform this clustering using a fully distributed map-reduce program and also using Map only NLineInputFormat technique of clustering data. We finally share our analysis of running both programs in terms of performance of these programs which will be shown below.

3. KNN:

Introduction

One of our project topics is KNN Classification algorithm, which we used to classify new cases based on the similarity measures. We performed KNN classification on data received from our HBASE task which populated data in the following format:

Movie name	Year of rating	Average Rating
Gladiator	2000	4.4
Rambo I	1998	3.2
The Silence of the Lambs	2003	2.0
Django	1996	5.3

The data above indicates the average rating of a movie in that particular year. Now based on the above rating, we divided average rating into three classes, i.e., Good, Bad and Average based on specific scale. This is used as to specify the class family for each movie they belong to. On the other hand, we took few test instances to perform the classification on the training data we have obtained from HBase as shown above. This data doesn't have the classes to which they belong and this is predicted by the program. The below is the sample of the test data we have.

Movie name	Year of rating
Fear	2002
Lolita	2003
Man of the house	2005
Born on the fourth of July	1999

We performed the classification by using a fully distributed MapReduce program with sorting done purely on Reduce side and also using MapReduce Secondary Sort technique. We finally shared our analysis of running both programs in terms of performance of these programs which will be shown below. As the given data set has very less features we implemented KNN on the length of the movie name and year of rating features. This means that the program can be run on more features if provided for more accurate results and it also means that feature selection is very important here. Our program can be easily tweaked to consider more features for classification if provided.

Data

We have taken Netflix dataset; this dataset is available on
http://www.lifecrunch.biz/archives/207

Properties of the dataset:
Movie data set contains 2 major tables.

The Data has the following features:

1. Movietitle.txt

Movie information in "movie_titles.txt" is in the following format:
MovieID, YearOfRating, Title

- MovieID does not correspond to actual Netflix movie ids or IMDB movie ids.

- YearOfRating can range from 1890 to 2005 and may correspond to the release of corresponding DVD, not necessarily its theatrical release.
- Title is the Netflix movie title and may not correspond to titles used on other sites. Titles are in English.

Movie_Title.txt

1,2003,Dinosaur Planet
2,2004,Isle of Man TT 2004 Review
3,1997,Character
4,1994,Paula Abdul's Get Up & Dance
5,2004,The Rise and Fall of ECW
6,1997,Sick
7,1992,8 Man
8,2004,What the #$*! Do We Know!?

2. CorrectedData .txt
   This contains information about the movie and its corresponding ratings. The data here is in CSV format. MovieId, UserId, Rating, Date of Rating.
   Corrected Data
   1,1488844,3,2005-09-06
   1,822109,5,2005-05-13
   1,885013,4,2005-10-19
   1,30878,4,2005-12-26
   1,823519,3,2004-05-03
   1,893988,3,2005-11-17
   1,124105,4,2004-08-05
   1,1248029,3,2004-04-22
   1,1842128,4,2004-05-09

[CustomerID,Rating,Date]

· MovieIDs range from 1 to 17770 sequentially.
· CustomerIDs range from 1 to 2649429, with gaps. There are 480189 users.
· Ratings are on a five star scale from 1 to 5.
· Dates have the format YYYY-MM-DD.

The data set is of 2.6 GB's so we did not face any such issue in uploading the dataset to S3. It hardly took few Mins/Hours to upload the data to S3 depending on the network speed. We just made sure we had 2 copies of dataset on AWS so that if by mistake we delete one copy we could easily work with other data set as backup option.

Technical Discussion

Purpose of tasks

1. **Hbase as Index:** The main purpose of this tasks is to find the running time difference between various types of joins so that when anyone has to apply join then we can just look around the data and decide on which join to use. There are 3 types of joins we have tried here, namely:
    a. Reducer side join: In this join we are sending all the data to reducer and then on reducer side all the computations are done.
    b. Replicated join: In this type of join we have distributed a file on all the mapper machines, the mapper machine will have a copy of one of the files to be joined and then mapper only can decide on what data is to be sent forward.
    c. Hbase join: Main purpose of this join was to use 2 programs 1st program will populate Hbase table just like what we did in replicated join, in replicated join we made sure one file i.e. the smallest file was made available to all the mapper function, in this join we made sure the smallest file was stored in a table and the second program named HCompute can read the data from this table and perform join operations.

2. K-Means:

Main task 1: The purpose of our main task 2 is to cluster the data on two features that is year of rating of a movie and the average rating of that movie for a particular year using a fully distributed map-reduce k-means clustering as taught in class.

Main task 2: The purpose of our main task 2 is to cluster the data on two features that is year of rating of a movie and the average rating of that movie for a particular year using a Map only NLineInputFormat approach.

3. KNN:

Main task 1: The purpose of our third main task is to classify the incoming test data which have two features over the train data on two features that is year of rating of a movie and the length of the movie name using a fully distributed map-reduce with sorting done in reducer side.

Main task 2: The purpose of our second main sub task is to use the advantage of the MapReduce internal sorting functionality instead of sorting manually in Reducer and therefore leading to a possible memory issues. So we implemented Secondary sorting for the KNN to sort over the values which in this case is the distance calculated. Below is the pseudo code for the same.

Main task 3: The purpose of our main task 3 is to perform the prediction on two features that is year of rating of a movie and length of the movie name for a particular year using a Map only NLineInputFormat approach.

Main idea

1. **Hbase as Index:** The main idea behind the join was very simple, we had to join 2 tables then we saw the data and came to a conclusion that we can use the ratings file and the movie title file and come up with a output of all the movies and their average rating per year. We could have done just the average but we required the output of year wise rating for our next few tasks, so we thought of including the same in all the joins.
    - **Reducer Side Join**
      We have 2 files to read the data from, 1st containing movie id and title and the 2nd containing movie id and the rating and rating year. So we had to use multiple inputs class to read the file.
      Mapper1:
      Map(… , r){
      Emit (r.movieId, (r.movieTitle+";"+"F1"))
      }

      Mapper2:
      Map(…, r){
      Emit (r.movieId, (r.Rating+";F2;"+r.RatingYear))
      }

      Reducer:
      > Reducer iterates over all the records coming to it, then stores the sum and count in a hash map keyed with year.
      > Reducer then emits Average as sum / count for each year and also writes title for the movie if the flag is F1.

    - **Hbase join:**
      HPopulate : This is used to add the data to Hbase table.
      File: movie_title.txt
      Table name: DataAll
      Key: MovieId
      Value: MovieTitle + ";" + MovieId

      HCompute: This is used to generate the year wise average of movies.
      Mapper1: Reads the data from Hbase table 'DataAll' populated in the last step
      emits(MovieId, (r.movieTitle+";"+"F1"))
      }

      Mapper2:
      Map(…, r){
      Emit (r.movieId, (r.Rating+";F2;"+r.RatingYear))
      }

      Reducer:

> Reducer iterates over all the records coming to it, then stores the sum and count in a hash map keyed with year.
> Reducer then emits Average as sum / count for each year and also writes title for the movie if the flag is F1.

- **Replicated Join**
  In this approach we have used concept of distributed cache. In distributed cache approach the file provided in the argument is made available to all the machines. We have used this concept to make movie_titles.txt file available to all the machines.
  Pseudo Code:
  1. Make the movie_titles.txt file available to all the machines executing map task.
  2. Store the movie_id as key to hash map and value as movie title
  3. Map: Reads the movie user rating file:
     Map(…,r){
     Emit (Movietitle, (Rating+";"+Rating year))
  4. Reduce:
     Iterate over all the values and store the sum and count in a hash map and emit year wise average of movie title

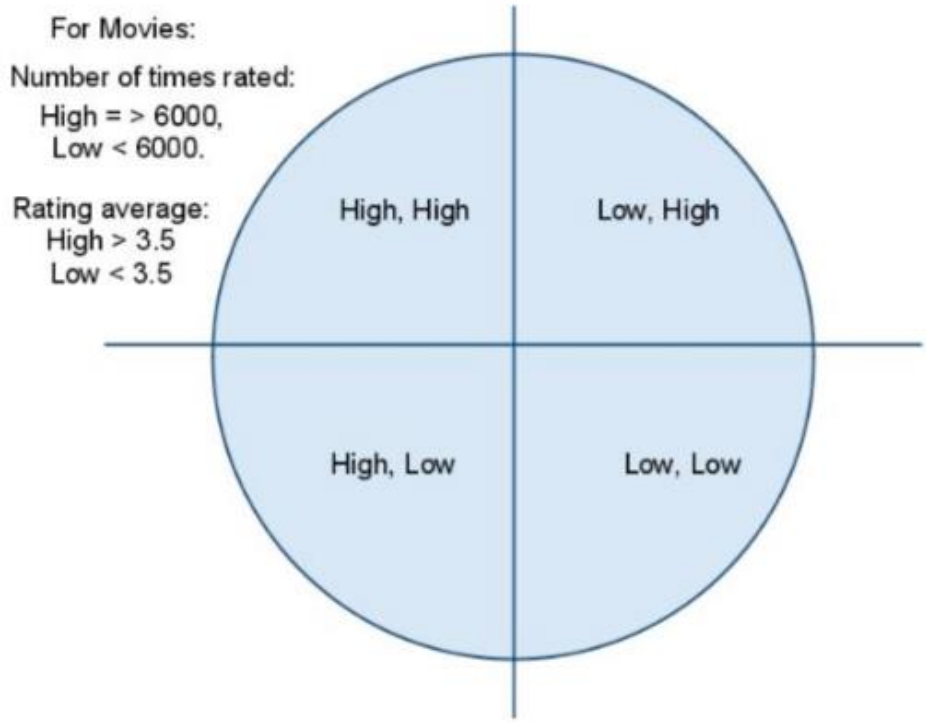
- **Dividing movies into Quadrants:**
  Step1: As our first step we used map/reduce to input the raw dataset and produce a new dataset containing:
  <MovieID firstDateRated, lastDateRated, yearOfProduction, totalNumRatings, avgRating, movieTitle>
  Legend:
  MovieID: Our key in the dataset. A unique Id assigned to each movie present in the dataset.
  firstDateRated: The date on which this movie was first rated by any user.
  lastDateRated: The date on which this movie was last rated by any user.
  yearOfProduction: The year which this movie was produced.
  totalNumRatings: Total number of ratings received by a movie.
  avgRating: The average of all the ratings received by a movie.
  movieTitle: Title of the movie.

  Pseudo code:
  Map1: Takes user rating files
  Emits(MovieID, ("F1;"+Rating+date))

  Map2: Takes movie_titles.txt
  Emits(MovieID,("F2;"+Year of release + ";"+ Title))

  Reduce: sorts on rating date to get the date first rated, last rated
  Emits(MovieId, (First rating date ,last rating date, Year of production, Total number of rating, Average rating, Title))

Step2: We visualize our dataset derived from first step to be divided into 4 quadrants as shown in fig



1st Quadrant: First quadrant consists of those movies that have received high number of ratings by users and also have been rated very highly. It can be predicted that these movies have a good probability of receiving a high rating by the targeted user.

2nd Quadrant: Second quadrant consists of movies that have received high number of ratings by users but have been rated very low. It can be predicted that these movies have a good probability of receiving a low rating by the targeted user.

3rd Quadrant: Third quadrant consists of movies that have received low number of ratings by users but have been rated very highly. A definitive conclusion about this set of movies and predicted ratings cannot be made.

4th Quadrant: Fourth quadrant consists of movies that have received low number of ratings by users and have been rated very low. A definitive conclusion about this set of movies and predicted ratings cannot be made.

Pseudo Code:
Map: Read the output produced in last step and generates output depending on the criteria as mentioned in the figure above.

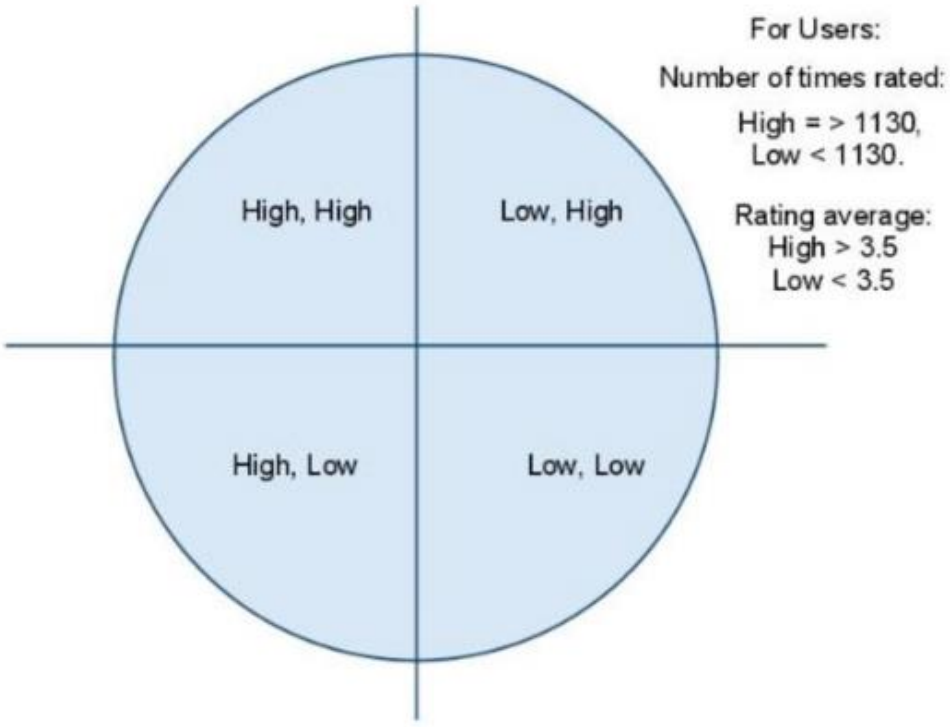
Reduce: Not required

- **Dividing user into quadrants:**

Step 3: As our third step we use map/reduce to input the raw dataset and produce a second dataset containing:
<CustomerID ratingCount, ratingAvg, movieRatingList>
Legend:
CustomerID: Our key in this dataset. A unique Id assigned to each user present in the dataset.
ratingCount: The number of ratings given by this user.
ratingAvg: The average of the ratings given by this user.
movieRatingList: A list of the for every movie that each user rated.

Pseudo code:
Map: Reads movie user rating file. And
Emit(UserId, (Rating ,Movie ID))
Reduce: Emits(UserId, (Total ratings, Average Rating of this user))

Step 4: We visualize our dataset derived from third step to be divided into 4 quadrants as shown in fig:



1st Quadrant: First quadrant consists of those users who have watched high number of movies and also have a high average rating. It can be concluded that the users in this quadrant have a higher credibility when it comes to rating a movie.

2nd Quadrant: Second quadrant consists of those users who have watched high number of movies but have a low average rating. It can be concluded that the users in this quadrant do not like the movies they have watched, or in other words, they are strict movie critics.

3rd Quadrant: Third quadrant consists of those users who have watched low number of movies but have rated them highly. It can be concluded that the users in this quadrant are occasional movie watchers.

4th Quadrant: Fourth quadrant consists of those users who have watched low number of movies and have rated them low. It can be concluded that the users in this quadrant do not give credible ratings.

Pseudo Code:
Map:
Takes the input generated from step earlier and divides the user into 4 quadrants depending on criteria mentioned in the figure above.

Reduce: Not required

2. KMeans

Main task 1:

Pseudo Code(From professors slides on chapter Data Mining Techniques 1)

CODE

```
Class Mapper {
  Centroids  // Array containing the K cluster centers

  setup()
    Centroids = read centroids from HDFS file

  map( …, record r) {
    closestCenter = Centroids[0]
    minDist = dist(closestCenter, r)

    for i=1 to k-1 {
      if (dist(Centroids[i], r) < minDist))
        closestCenter = Centroids[i]
        minDist = dist(Centroids[i], r)
    }
    emit( closestCenter, r )
  }
}

reduce( center, [r1, r2,…]) {
  for each record r in inputList
    update sum and count for each dimension

  newCentroid = compute the average for each dimension
  write newCentroid to an HDFS file
}
```

Main task 2:

Pseudo Code(From professors slides on chapter Data Mining Techniques 1)

NLineInputFormat

```
// The input file is copied to each worker using
// the distributed file cache
Class Mapper {
  data D

  setup() {
    D = read data from distributed file cache
  }

  map( …, K ) {
    // Kmeans() is a sequential implementation
    // of the K-means algorithm
    clustering = Kmeans( D, K )

    emit( K, clustering )
  }
}
```

Source code file name: KMeans_NLineInputFormat.java

3. KNN

Pseudo Code for Main Task 1:

```
Mapper:
        Create List testData
        Create List trainData

setup(Contrext cntx){
        # Load test data from distributed cache
        newline = read line from distributed cache
        While(newline != null)
                load testData list with newline
}
Map(.., r){
        # Load train data
         Load trainData with new record.
}
Cleanup(){
```

```
        Create List result

        For each row in testData
        Set testMovieLength and testMovieYear

                For each row in trainData
                Set trainMovieLength and trainMovieYear

                Distance = Euclidean distance of testMovieLength and testMovieYear,
                                trainMovieLength and trainMovieYear

                add Distance to the list result with training features appended to it.

                #sort result list based on the distance.
                result.Sort()

                select first k records, where k is specified by the user.
                Key = testMovie+Year;
                Value = trainRow
                Emit(Key, Value)
}
Reducer:

Reduce(){
        Create List finalResults
        For each incoming value
                Load finalResults with value

        # sort the list
        finalResults.sort()

        Iterate through the finalResults list for k times.
                Find maximum class name and store.

        Emit(key, class name)
}
```

Pseudo Code for Main Task 2:

Pseudo Code[4]:

```
Mapper:
        Create List testData
        Create List trainData
```

```
setup(Contrext cntx){
        # Load test data from distributed cache
        newline = read line from distributed cache
        While(newline != null)
                load testData list with newline
}
Map(.., r){
        # Load train data
         Load trainData with new record.
}
Cleanup(){

        Create List result

        For each row in testData
        Set testMovieLength and testMovieYear

                For each row in trainData
                Set trainMovieLength and trainMovieYear

                distance = Euclidean distance of testMovieLength and testMovieYear,
                                trainMovieLength and trainMovieYear

                add distance to the list result with training features appended to it.

                #sort result list based on the distance.
                result.Sort()

                select first k records, where k is specified by the user.

                WritableComparable class = new WritableComparable
                class.setMovie = testMovieName
                class.setDistance = distance

                value = trainRow
                Emit(class, value)
}


SortComparator(){
        return (WritableClass a).compare(WritableClass b)

}

GroupComparator(){
        getMovieA = (WritableClass a )
```

```
        getMovieB  = (WritableClass b )

        return getMovieA.compare(getMovieB)

}

CustomPartitioner(Key k , Value v, number of partitions nop){
        return key.hasCode * 127 / nop
}

Reducer:

Reduce(){

        For k times.
                Find maximum class name and store.

        Emit(key, class name)
}
```

Pseudo Code for Main Task 3:

```
Mapper:
        Create List trainData

setup(Contrext cntx){
        # Load test data from distributed cache
        newline = read line from distributed cache
        While(newline != null)
                load trainData list with newline
}
Map(.., r){
        # Load train data
         Load trainData with new record.

        Create List result

        Set testMovieLength and testMovieYear from input record ‘r’

        For each row in trainData
        Set trainMovieLength and trainMovieYear

        Distance = Euclidean distance of testMovieLength and testMovieYear,
                                trainMovieLength and trainMovieYear

        add Distance to the list result with training features appended to it.
```

#sort result list based on the distance.
result.Sort()

select first k records, where k is specified by the user.

Iterate through the result list for k times.
Find maximum class name and store.

key = testMovieName + Year
Emit(key, class name)

}
Cleanup(){

}

Files in the zip folder

Hbase as Index:

Folder name: Join Source

1. Reducer Side Join → ReducerSideJoin.Java
2. Replicated Join → Repl.Java
3. Hbase Join → HPopulate.Java and HCompute.Java
4. Movie Quadrants → Step1: DataVizClass1.Java
   Step2: MovieQuad.Java
5. User Quadrants → Step3: DataVizClass2.Java
   Step4: UserQuad.Java
6. Kmeans → Source Code file name : -kmeans.java
   -KMeans_NLineInputFormat.java

File:

1. Folder Name: K-means Distributed movies1 1 iteration
This folder contains the log files, error file, stdout and controller file for the fully distributed run of one iteration of the k-means with 5 worker machines.

2. Folder Name: K-means Distributed movies1 5 iteration
This folder contains the log files, error file, stdout and controller file of the fully distributed run for 5 iterations of the k-means with 5 worker machines.

3. Folder Name: Kmeans Distributed movies1 5 iterations 10 workers
This folder contains the log files, error file, stdout and controller file of the fully distributed run for 5 iterations of the k-means with 10 worker machines.

4. Folder Name: Kmeans Distributed movies1 5 iterations 15 workers
This folder contains the log files, error file, stdout and controller file of the fully distributed run for 5 iterations of the k-means with 15 worker machines.

5. Folder Name: Kmeans Distributed movies1 5 iterations 19 workers

This folder contains the log files, error file, stdout and controller file of the fully distributed run for 5 iterations of the k-means with 19 worker machines.

Source Code file name : kmeans.java

File:
1. Folder Name: Kmeans Map only movies1 1 iteration
This folder contains the log files, error file, stdout and controller file for the Map only NLineInputFormat for one iteration of the k-means on movies1 which is of size 1GB.

2. Folder Name: Kmeans Map only movies 1 iteration
This folder contains the log files, error file, stdout and controller file for the Map only NLineInputFormat for one iteration of the k-means on movies1 which is of size 2.5MB.

7. KNN :
Source code:
   a. KNN.Java (Manual Sort in Reducer)
   b. KNNSecondarySort.java, KNNWritableClass.java
   c. KNNMapSide.java (NLineInputformat, Map only program)

Files:
Folder Name 'KNN-5C':
This folder contains the log files, error file, stdout and controller file for the fully distributed run of KNN with 5 small machine worker machines on testmovies test data and movie training data.

Folder Name 'KNN-5L':
This folder contains the log files, error file, stdout and controller file for the fully distributed run of KNN with 5 large machine worker machines on testmovies test data and movie training data.

File for 'KNN-SS-5C':
This folder contains the log files, error file, stdout and controller file for the fully distributed run of KNN Secondary Sort with 5 small machine worker machines on testmovies test data and movie training data.

File for 'KNN-SS-5L':
This folder contains the log files, error file, stdout and controller file for the fully distributed run of KNN Secondary Sort with 5 large machine worker machines on testmovies test data and movie training data.

File for 'KNN-MapOnly-5C':

This folder contains the log files, error file, stdout and controller file for the fully distributed run of KNN Map only NLineInputFormat with 5 small machine worker machines on testmovies test data and movie training data.

File for 'KNN-MapOnly-5L':
This folder contains the log files, error file, stdout and controller file for the fully distributed run of KNN Map only NLineInputFormat with 5 large machine worker machines on testmovies test data and movie training data.

Concrete results

1. **Hbase and Index:** From the results we found that:

Join type	5 Large	2 Large
Replicated	3 mins 17 sec	6 mins 32 Sec
Reducer	3 mins 23 sec	6 mins 17 Sec
Hbase	5 mins 18 sec	9 mins 50 Sec

Number of records written:
Replicated: Map output records=100480507
Reducer: Map output records=100498277
Hbase: Map output records=100498277

The number of records written for Replicated join are less because, in replicated join we are joining first in mapper phase itself thus eliminating the unwanted records or the records that have no matching records in the other file. But in case of Hbase and reducer side join we are sending all the records to the reducer and then in reducer we are deciding whether to continue with them. So the number of records in Reducer and Hbase join is more.

2. Kmeans: Fully Distributed k-means

Program	Machine configuration(Master and Workers)	Number of initial cluster centers	Size of Movie data set	Number of iterations	Time Taken	Folder for logs, stdouts, controllers and stderr
MapReduce	1 Master and 5 workers	5	1 GB	1	Start time: 16:01:19 End time: 16:10:21 Time taken:9 min 2 sec	K-means Distributed movies1 1 iteration
MapReduce	1 Master and 5 workers	5	1 GB	5	(avg time) 7min 40 *5 = 38min 20s	K-means Distributed movies1 5 iteration
MapReduce	1 Master and 10 workers	5	1 GB	5	(avg time) 6min 15s *5 = 31min 15s	Kmeans Distributed movies1 5 iterations 10 workers
MapReduce	1 Master and 15 workers	5	1 GB	5	(avg time) 6min 39s * 5 = 33min 15s	Kmeans Distributed movies1 5 iterations 15 workers
MapReduce	1 Master and 19 workers	5	1 GB	5	(avg time) 4min 30s * 5 = 22min 30s	Kmeans Distributed movies1 5 iterations 19 workers

Map only NLineInputFormat k-means

Program	Machine configuration(Master and Workers)	Number of initial cluster centers	Size of Movie data set	Number of configurations	Number of iterations	Time Taken	Folder for logs, stdouts, controllers and stderr
Map only NLineInputFormat	1 Master and 1 worker	5	1GB	1	1	Start time: 22:05:11 End time: 22:07:40 Time taken:2 min 29s	Kmeans Map only movies1 1 iteration
Map only NLineInputFormat	1 Master and 1 worker	5	2.5MB	1	1	Start time: 18:51:45 End time: 18:52:41 Time taken:56s	Kmeans Map only movies 1 iteration
Map only NLineInputFormat	1 Master and 1 worker	5	1GB	1	5	Avg time * iterations 7min 40s * 5 Time taken=38 min20s	Kmeans Map only movies 5 iteration
Map only NLineInputFormat	1 Master and 5 worker	5	2.5MB	5	15	Start time: 12:54:18 End time: 12:55:31 Time taken:1min13s	Kmeans Map only movies 15 iterations 5 configs 5 machines
Map only NLineInputFormat	1 Master and 10 worker	5	2.5MB	5	15	Start time: 13:49:28 End time: 13:50:44 Time taken:1min16s	

3. KNN:

KNN – manual sort in reducer:

Program	Machine configuration(Master and Workers)	K value	Size of Movie data set	Time Taken
KNN	1 Master and 5 workers (small machines)	5	Train File: 2.4MB Test File: 210 KB	Start time: 00:21:02 End time: 01:10:37 Time taken: 0:49:35
KNN	1 Master and 5 workers(large machine)	5	Train File: 2.4MB Test File: 210 KB	Start time: 01:05:44 End time: 01:27:25 Time taken: 0:21:41

KNN with Secondary Sort:

Program	Machine configuration(Master and Workers)	K value	Size of Movie data set	Time Taken
KNNSecondarySort	1 Master and 5 workers (small machines)	5	Train File: 2.4MB Test File: 210 KB	Start time: 20:05:17 End time: 20:48:52 Time taken: 0:43:35
KNNSecondarySort	1 Master and 5 workers(large machine)	5	Train File: 2.4MB Test File: 210 KB	Start time: 00:35:47 End time: 00:55:47 Time taken:0:20:00

KNN with Map only NLineInputFormat:

Program	Machine configuration(Master and Workers)	K value	Size of Movie data set	Time Taken
KNNMapOnly	1 Master and 5 workers (small machines)	5	Train File: 2.4MB Test File: 210 KB	Start time: 18:53:28 End time: 19:51:13 Time taken: 0:57:45
KNNMapOnly	1 Master and 5 workers(large machine)	5	Train File: 2.4MB Test File: 210 KB	Start time: 01:49:42 End time: 02:10:29 Time taken: 0:20:47

Log files descriptions

The log files for join tasks is present in Logfile/LogFiles-Join Run 5 Large folder.

Task	Log file name
Data division for Movies	DataViz1.txt
Movie quadrants	MovieQuards.txt
Data Division for Users	DataViz1.txt
User Quadrants	UserQuards.txt
HPopulate	HPopulate.txt
HCompute	HCompute.txt
Reducer Side join	ReducerSide.txt
Replicated Join	ReplicatedJoin.txt
Kmeans	All files given with the folders names as shown in table
KNN	All files given with the folders names as shown in table

Clever solution

K-means Map only NLineInputFormat: In this approach (Main task 2), code does not run that well with big data since map task attempt fails to respond for a long time while calculating convergence of clusters which results in an map task's unresponsive error. The workaround for this is to write centroids that are created after each iteration of k-means clustering into a file at the end of an iteration. This will have map calls communicating with master on a regular basis and thus will not give an unresponsive error.

KNN Secondary Sort: To improve the run time performance of the program and to make use of MapReduce in built sorting functionality, we improved the Main Task 1 KNN algorithm using Secondary Sorting technique. By doing so, we saved extra computation required for sorting the input reducer records, no memory is required and only K instances have to be read in reducer. There is a little programming needed compared to manual value sorting in reducer but the performance was improved significantly.

KNN Map only: This is our third task. As said earlier, this may not well with big data but the advantage of using this method we reduced the programming complexity as there is no need of outer for loop. The reason is, training data is load to list from distributed cache and test instances were sent to mapper as records. So we find prediction for that record over all training data which is one loop and emit the result.

Setup Challenges

In join task two important challenges were faced in Replicated join those were:

1. How to keep the file in distributed cache. How to access it in Map function? We found the solution to this by doing some research and finding few codes for replicated join available on net. We modified the code to suit our needs, the link to the code is provided in the reference.
2. Also giving the path in Replicated join was difficult, we ran multiple jobs, which failed, then we came to know that instead of giving s3:// … we have to provide path like s3n:// …

Challenges faced in Kmeans task:

The major challenge that we faced was with the configuration settings in the fully distributed map-reduce k-means clustering. Here I thought that once I load a file into the distributed cache then I can read the cache file using the local file system object which was not true. The distributed cache is to be read using a file reader object. This took a long time and analysis to find out what was wrong with our program.

In my 2nd task I faced problems with a map task unresponsive error with large data sets. I went with an approach with writing the new centroids created at each iteration into a file and realized that this prevents map tasks from becoming unresponsive for a long time.

Challenges faced in KNN task:

The major challenge that we faces was similar to the one which we faced with Kmeans. The configuration settings, i.e., implementing the distributed cache took quite a long time consuming the smaller chunk of productive time. We have referred many articles and online blogs to fix this issue.

Also, while implementing KNN using secondary sort technique the group comparator used to take time and program became unresponsive because of type mismatch error. This bug in the code is resolved with carefully debug and printing the stages in system console.

Conclusions

About Hbase as Index:

From the above table about the time it can be concluded that Replicated join is fastest, this is fastest because we are doing the joining in Map task itself and sending the output to reduce task for further processing. But the main issue is the storing the file in distributed cache. If the file to the stored on local machine is too large it might give java heap space error since we are creating a Hash Map from file and the map can easily go out of memory, in such cases we can use Hbase to store the data in table and in map/reduce phase we can fetch from it. Since the Hbase is sorted on key so it works like a Binary search and finds the key quickly.

Also choosing which data / file to be stored on distributed cache is important consideration, we would want to keep the file which requires multiple access each time to be on cache so that we can fetch data easily from hash.

Considering the time difference between same tasks on different number of machines it can be seen that, as the number of machines increased from 2 to 5 the time got reduced drastically in all the 3 joins. It nearly got halved.

Load balancing: There is load balancing in this task since the key is movieId(in Hbase and reducer side join) or title (in replicated join), the data is emitted based on key and all the give movie records will end up on one reducer call. The problem can arise when there is very less data for a given movie and since only that record will be processed by that reducer and hence it will be working less in comparison to reducer call with lot of movie data.

Memory consumption: The memory consumption will be very less as only task of memory is to store all the ratings and their count per year. There is a hash map on reducer side which stores movie rating sum based on key as year and one more hash map to store count. So basically the number of entries in hash map is equal to number of years data present. This cannot be bottleneck since the number of year's data is not too high in this case.

Network traffic: The network traffic is bit too high since all the data is given out by mapper to reducer, mapper gives out id, ratings and year details to reducer so there is lot of data transfer from mapper to reducer. In replicated join we have tried to reduce the number of records emitted since the join is performed in Map function itself. Hence it sill not give out the id values title whose other matching half is not found.

Running time is discussed above.

2. KMeans:

Main task 1: **Performance of fully distributed k-means clustering:**

Input: Movie data file
Output: Centroids
Distributed Cache: Contains the new centroids after each iteration

Load Balancing: There is load balancing in this approach as the input file is divided across different machines and the k nearest neighbor movies and it's rated year are emitted as key with its class name as values. These values are collected by the reduce phase and rank of the neighbor class names is calculated. Hence the load is distributed well across the mappers and reducers but the number of reducers that are being populated is equal to the number of centroids which are present. Hence if centroids < reducers then few reducers may remain idle but if centroids > reducers then the load will be well balanced without any machines going idle.

Memory Consumption: The memory consumption in mappers is not there since anything that is close to the centroid is emitted to the reducer. The reducer only keeps a count of the sum of years and sum of stars and the total number of records in the cluster to find out the new centroid for the next iteration of kmeans. The only other memory consumption is the centroids which are stored in the distributed cache.

Network Traffic: The network is well occupied based on the number of centroids that are present. So if the number of machines allocated for the job is more than the number of centroids then map workers will be occupied but the number of reduce calls would be at max equal to the number of centroids. This will do fine as long as we have enough number of centroids. Also the data moves from mappers to reducers based on keys which are centroids in our case.

Local CPU cost: The local CPU cost should not be too much. Only in cases where the centroids are less than the worker machines then the reducers may remain idle because of this but the mappers will be consumed. Also the mappers will be distributing their work so cost of each CPU will reduce due to input data being sent to different mappers. The reducers may get a little loaded if the centroids are not chosen wisely as one reducer can have too much data due to a huge cluster.

Running Time: In the fully distributed map reduce, the data is read twice but the number of machines reading this data is more. So reading this data is not the major problem but this data is written once by map to be collected by reducer later which results in expensive FILE I/O.

Main task 2: **Performance of map only NLineInputFormat KNN clustering:**

Input: Centroid file

Output: Centroids
Distributed Cache: Movie data set

Load Balancing: Load is balanced across the map calls in a way that multiple configurations of k-means can run simultaneously but these map calls will be working on their respective k-means just like local machines would perform k-means clustering on a single configuration. Also a huge distributed cache file of the data is shared among all the worker(mapper) machines.

Memory Consumption: Mapper machines store the totals of year and ratings with the respective number of elements in each cluster. Iterations are done on a repetitive basis till convergence of centroids within the map and new centroids are found. Also there is a huge Distributed cache file of the movie set data which is shared among all the mappers.

Network Traffic: The Network traffic is as follows:

- The movie data file is loaded into the distributed cache which is made available for all mapper machines.
- Each map call performs the kmeans clustering for one configuration. Here input is just a line for each mapper and just a line as output.

Local CPU cost: Each machine performs the entire clustering which means that machines will be computing a lot since movie data is not distributed across mappers but the whole data is sent to each of the mapper machines.
Flaw: Mapper machines often have map tasks which go unresponsive which results in error.

Running Time: Running time for the program is good since data is read only once on each of the data except that a large amount of the data set is kept in the Distributed cache which is made available to a lot of mappers which can be expensive when distributed cache runs out of memory on large data set.

Analysis Task: Fully Distributed map-reduce k-means clustering v/s Map only NLineInputFormat k-means clustering

Fully distributed k-means	Map only k-means
Made for big data so that work can be distributed	Made for not so big data as a single machine is needed to perform the whole operation
Scalable based on number of centroids	Scalable based on number of configurations
Complex in terms of map-reduce	Same code can be implemented in map phase as it is in a sequential k-means program.
Currently analysis of above programs show it has been beaten for performance by map-only k-means because of the amount of network traffic that takes place here but when the amount of data becomes very	This works faster because of lesser network traffic across machines but when data increases the cost on a single local machine is very high since it is performing the whole k-means clustering within one local

large fully distributed k-means is the approach to be taken. This is because our data is distributed across mappers and reducers here which reduces the load on a single local machine.	machine. This leads to unresponsiveness on many occasions when involving big data.
Memory consumption is low here as the data is distributed and distributed cache only holds centroids which is not too much of data	Memory consumption is high here since the whole data set is stored in the cache which might run out of memory if data is very large which can result in expensive I/O operations
For a single configuration of k-means clustering load is balanced across the machines(mappers and reducers).	For a single configuration of k-means load is all on a single machine and not distributed for a single configuration but for multiple configurations.

HelperTask:
Creating dummy data set movies1 by increasing the size of the original data we got from HBASE task to check performance of the k-means in both approaches.
Code available in Movies.java

KNN:
Main task 1:

Input: Movie data file
Output: all test data with class prediction
Distributed Cache: Contains the test data set

Performance of KNN reducer sort:
Load Balancing: There is load balancing in this approach as the input file is divided across different machines and the test movie name and year is emitted as key with its class. These values are collected by the reduce phase and then sorted to get top K classes to find the ranking. Hence the load is distributed well across the mappers and reducers but the number of reducers that are being populated is equal to the K value specified. Hence if K value is less than the number of reducers then few reducers may remain idle but if K is greater than reducers then the load will be well balanced without any machines going idle.

Memory Consumption: The memory consumption in mappers as we capture all the train data in the collection or in other terms memory but not all data will be sent to the reducer as the mapper emits k value per test instance. The reducer again saves all the incoming records in the memory to sort based on the values and emit the highly ranked class from first K records in memory. The other memory consumption is the test data which are stored in the distributed cache.

Network Traffic: The network is well occupied based on the K value. So if the number of machines allocated for the job is more than the K value then map workers will be occupied but

the number of reduce calls would be at max equal to the K value. Also the data moves from mappers to reducers based on keys which are test records in our case.

Local CPU cost: The local CPU cost should not be too much. Only in cases where the K value is less than the worker machines then the reducers may remain idle because of this but the mappers will be consumed. Also the mappers will be distributing their work so cost of each CPU will reduce due to input data being sent to different mappers.

Running Time: In the fully distributed map reduce, the data is read once and only k records were emitted per map per test record. So there is less data traffic from mapper to reducer and the run time improves with it.

Main task 2:

Input: Movie data file
Output: all test data with class prediction
Distributed Cache: Contains the test data set

Performance of KNN Secondary Sort:

Load Balancing: There is load balancing in this approach as the input file is divided across different machines and the test movie name and year is emitted as key with its class. The values are just read in reducer for k times and rest is omitted. Hence the load is distributed well across the mappers and reducers but the number of reducers that are being populated is equal to the K value specified. Hence if K value is less than the number of reducers then few reducers may remain idle but if K is greater than reducers then the load will be well balanced without any machines going idle.

Memory Consumption: The memory consumption in mappers as we capture all the train data in the collection or in other terms memory but not all data will be sent to the reducer as the mapper emits k value per test instance. The reducer then just have to read the first K records and then the highly ranked class from first K records is emitted. This is the advantage over the Main Task 1 as we use extra memory in reduce phase too. The other memory consumption is the test data which are stored in the distributed cache.

Network Traffic: The network is well occupied based on the K value. So if the number of machines allocated for the job is more than the K value then map workers will be occupied but the number of reduce calls would be at max equal to the K value. Also the data moves from mappers to reducers based on keys which are test records in our case.

Local CPU cost: The local CPU cost should not be too much. Only in cases where the K value is less than the worker machines then the reducers may remain idle because of this but the mappers will be consumed. Also the mappers will be distributing their work so cost of each CPU will reduce due to input data being sent to different mappers.

Running Time: In the fully distributed map reduce, the data is read once and only k records were emitted per map per test record. So there is less data traffic from mapper to reducer and then the reducer only reads first K records thus improving performance compared to Main task 1.

Main task 3:

Input: Movie data file
Output: all test data with class prediction
Distributed Cache: Contains the train data set

Performance of map only NLineInputFormat KNN classification:

Load Balancing: Load is balanced across the map calls. The map calls will be working on their respective input test data just like local machines would perform classification on a single configuration. Also a huge distributed cache file of the data is shared among all the worker (mapper) machines.

Memory Consumption: Mapper machines, there is a huge Distributed cache file of the movie set data which is shared among all the mappers. Apart from that, for each input test record the classification is done and emitted.

Network Traffic: The Network traffic is as follows:

- The train movie data file is loaded into the distributed cache which is made available for all mapper machines.
- Each map call performs the KNN classification for one configuration. Here input is just a line for each mapper and just a line as output.

Local CPU cost: Each machine performs the entire classification which means that machines will be computing a lot since train movie data is not distributed across mappers but the whole data is sent to each of the mapper machines.

Running Time: Running time for the program is good since data is read only once on each of the data except that a large amount of the data set is kept in the Distributed cache which is made available to a lot of mappers which can be expensive when distributed cache runs out of memory on large data set.

References

1. The user and movie Quadrants division idea is taken from http://hackedexistence.com/project-netflix.html.
2. The replicated join idea is taken from: https://gist.github.com/airawat/6587341.
3. Professor slides for k-means clustering.
4. KNN Classification paradigm. http://ijssst.info/Vol-15/No-3/data/3857a513.pdf

5. https://code.google.com/p/joycrawler/source/browse/trunk/org/niubility/learning/knn/KNNMapper.java?r=254