**CS6240 Project Intermediate Report**

**GitHub:** https://github.ccs.neu.edu/orgs/cs6240f18/teams/just-reduce-it

**Team Members**

* Ishan Patel
* Rahul Motan
* Aparna Sharma

**Project Overview**

The main goal of our project is to compare the performance and scalability of using HBase as an index for an equi-join implementation against hash+shuffle (Reduce-side join). For this purpose, we will be using the Youtube data derived from <http://socialcomputing.asu.edu/datasets/YouTube2> which has two files namely : edges.csv and groupEdges.csv. The edges.csv file shows a isFriend relationship between 2 users. For instance : 1,2 depicts that user 1 is friends with user 2. The file groupEdges.csv shows the list of groups a user is a member of. For instance 1, 47 means user 1 is a member of group number 47. Since the data provided was small, we duplicated the data 10 times. Using these data we will be suggesting every user some groups which his friends are a part of but not him. To get this data out of the given data we are using HBase as index to perform equi-join and a hash+shuffle job using both map reduce and spark. The results derived after performing these programs are in the form of 1, 34 which means group 34 has been suggested to user 1 since his friends are a part of this group and not him. For performing hash+shuffle simple mapping, joining and then grouping were performed. But for HBase, we have stored the data of groups in HBase since it was used multiple times during computation. Later we are accessing this data in HBase using map reduce code. This way we do not have to read the entire groupEdges.csv file again and again.

**Input Data**

**Data Source Link:** <http://socialcomputing.asu.edu/datasets/YouTube2> it is one of the datasets available on http://socialcomputing.asu.edu/pages/datasets

**Properties of Data:** This is the data set crawled from YouTube (http://www.youtube.com). YouTube is a video-sharing website on which users can upload, share, and view videos.

This contains the friendship network crawled and group memberships. For easier understanding, all the contents are organized in CSV file format.

Basic statistics

Number of users: 1,138,499

Number of friendship pairs: 2,990,443

Number of groups: 47

**Task 1a: Understanding HBase**

As per our requirement we will be using HBase to store one input relationship of the data in HBase and then use a map reduce job that scans through the other input relation and looks for matches in HBase. So we started by understanding the basics of HBase from the Book “Definitive Guide” by Tom White. After successfully installing HBase we tried to run the basic programs and then started to store the file GroupEdges.csv as our first step. Then we tried to perform the get by rowKey operation on this stored data. We tried it through the HBase shell as well as by writing map reduce code. We also tried to implement a short findByKey program and ran it on AWS cluster. The running of code on AWS code was a challenge and took majority of the time. Initially we faced some basic issues such as deciding upon the configuration such as which server to use. Since we have never worked on hybrid technologies, it was not clear if everything had to be executed on the same cluster or not, what type of cluster should be choosen or how was it possible for us to change the configuration for persistence of HBase table in S3. Other issue which we faced was due to usage of older security group which had all ports open to all traffic which led to DDOS attack on other hosts.

**Task 1b: Dataset Expansion & Reduce Side Join**

Overview

Since the dataset chosen was too small, hence a program to extend the data was required. We accomplished this by emitting an edge to new dummy node for each record read by the mapper. The new dummy node is chosen from the last node in the dataset.

The second task was to join the user-friendship data with the user-relation data to suggest the user an appropriate group based on his friend’s groups.

Pseudo-Code for Dataset Expansion

Map (id1, id2) {

               Emit (id1, id2)

}

Reducer () {

               maxIdSize = someValue

               multiplyData = 10

               reduce (keyID, listOfIds) {

**for** each id in listOfIds **do**

**for** i = 0 to multipyData-1 **do**

                                                            emit (keyID+ maxIdSize\*i, id+ maxIdSize\*i)

}

Pseudo-Code for Reduce Side Join

**Job 1**

Input: extendedEdges.csv  
UserMap(user1,user2) {

Emit (user1, ”<FRIEND>” + user2)

Emit (user2, ”<FRIEND>” + user1)

}

Input: extendedGroupsEdges.csv

GroupMap(userId,groupId) {

            Emit(userId,GROUPgroupId)

}

Output: Intermediate.csv

Reduce () {

            listOfFriends.clear()

listOfSuggestedGroups.clear()

**for each** t in values **do**

            tokens = t.split(",")

**if** tokens[0] is FRIEND **then**

                         listOfFriends.add(tokens[1])

**else if** tokens[0] is GROUP **then**

                         listOfSuggestedGroups.add(tokens[1])

**If** listOfFriends is not empty **AND** listOfSuggestedGroups is not empty **then**

**for** each friend in listOfFriends **do**

**for** each suggestedGroup in listOfSuggestedGroups **do**

                                    emit (friend, suggestedGroup)

}

**Job 2**

Input: Intermediate.csv

Map (friend, suggestedGroup) {

            emit (friend, “<SUGGESTEDGROUP>” + suggestedGroup)

}

Input: extendedGroupsEdges.csv

GroupMap (userId,groupId) {

            emit (userId,” <GROUP>” + groupId)

}

Output: output.csv

Reduce (user, listofGroups) {

**for** each group in listofGroups **do**

**If** group.startsWith(“GROUP”) **then**

                                    existingGroup.add(group)

**else if** group. startsWith (“SUGGESTEDGROUP”) **then**

                                    newSuggestedGroup.add(suggested)

**For** each group in newSuggestedGroup **do**

**If** group not in existingGroup **then**

                        emit (user, group)

}

Speedup

|  |  |
| --- | --- |
| Cluster Size | Time Elapsed |
| 5 | 170 |
| 10 | 113 |

Speedup = 170/113 = 1.504

This speedup seems satisfactory as it’s quite near to the ideal case which is 2.

Result

The output is in the format (userId, groupId). The group id is the suggested group that the user can choose to join.

Sample output: <https://drive.google.com/open?id=1fOOfPaiM7hAYn-gNDd9-HGQIRmbKVnSU>

1000010,35

1000010,21

**Task 2: Performing Join Using HBase**

**Overview**

For this task, we first implemented a MapReduce task to create a table for the user groups. The data is inserted in the following format.

*<rowKey:userId> column=<ColumnFamily:”data”>:<qualifier:”groups”, <timestamp>, <value>*

Sample Data:

8285981 column=data:groups, timestamp=1544400657616, value=7969529-7969512-7969500

The user friendship dataset which is edges.csv represents a bidirectional edge. So, in the MR job where we perform the join with the group data, we emit a reversed edge too. For instance, a mapper gets the following record:

Map:

*1,2 -> emit (1,2) and emit (2,1)*

Reduce:

(1, [2, 4, 6])

*GroupData in HBase:*

*2: 2-50-145*

*4: 2-35-50*

*6: 1-5*

*//suggested groups for user 1*

*Emit (1, “1-2-5-35-50-145”)*

**Pseudo Code for Inserting Data in HBase**

Using the file generated after running data-duplication job on groups\_edges.csv, I used this pseudo code to insert records into HBase.

Mapper {

Map (User user, Group grp) {

Emit (user, grp)

}

}

Reducer {

Reduce (User key, Group [] groups) {

// convert key to bytes

//convert groups to bytes

row <- new Row(key)

row.add (“group”,value=groups)

}

}

**Pseudo Code for Joining Data**

MR Job to join user-friendship data with group data to find group suggestions for the user.

Mapper {

map (User u1, User u2) {

emit (u1, u2) // emit the edges to show friendship between users

emit (u2, u1) // emit the edges to show friendship between users

}

}

Reducer {

Table table;

Setup () {

*//get table name from job configuration*

String tableName = configuration.get(“tableName”)

*//get hbase configuration and set master node properties*

*// create table and set column family and column name*

Table = connection.get(tableName)

}

Reducer (User key, User [] users) {

listOfGroupsForThisUser = getListOfGroupsFromHBase(key)

listOfSuggestedGroups = new List()

for each user in users do {

groups = table.get(user);

for each group in groups do

*// add only if it already doesn’t have it*

listOfSuggestedGroups.add(grp)

}

emit (user, listOfSuggestedGroups)

}

}

**Algorithm and Program Analysis**

In this task, we observed that HBase is better when used with an extremely large dataset because HBase can provide great scalability and greater through-put for random lookups thus its better to use it with a large dataset for further optimization.

Comparing this task with the reduce-side join executed in the previous task, we observed that this program can achieve the same results with 2 less map tasks and 1 less reduce task.

**Speedup**

The code ran for 55 mins on AWS cluster but we could not get the correct results. But we were successfully able to execute the code on local system. The code ran for almost 2 hours on local system including populating the HBase with the group data.

**Result**

The group id is the suggested group that the user can choose to join.

The result is in the following format:

<userId>, <suggestedGroups>

10000501,9108031

1000261,20

1000261,21

**Task 3: Performing Hash+shuffle in spark**

We have performed hash+shuffle to suggest users their groups which their friends are a part of but not them. The pseudo code for the same has been given below:

val userTextFile = sc.textFile(args(0))

val friends = userTextFile.map(line => line.split(",")) // split line by comma

.map(record => (record(0), ("friend-"+record(1)))) // emit 1 for userid being followed

val reverseFriends = userTextFile.map(line => line.split(",")) // split line by comma

.map(record => (record(1), ("friend-"+record(0)))) // emit 1 for userid being followed

val friendList = friends.union(reverseFriends)

//friendList.foreach(println)

val groupTextFile = sc.textFile(args(1))

val userGroupList = groupTextFile.map(line => line.split(","))

.map(record => (record(0), ("group-"+record(1)))).cache()

//userGroupList.foreach(println)

val data = friendList.union(userGroupList)

.groupByKey()

.mapValues(\_.toList)

//data.foreach(println)

val suggestedList = data.flatMap{case(user, list) => getSuggestionsList(user,list)}

//val suggestedList = sc.parallelize(suggestedListMap.toSeq)

val intermediate = suggestedList.union(userGroupList).groupByKey().mapValues(\_.toList.distinct)

//intermediate.foreach(println)

val finalResult = intermediate.flatMap{case (user, list) => removeAlreadyExistingGroups(user,list)}

finalResult.foreach(println)

finalResult.saveAsTextFile(args(2))

}

def getSuggestionsList(user:String, list:List[String]): List[(String, String)] = {

var result: List[(String, String)] = List()

var groups : List[String] = List()

var friends : List[String] = List()

for(item <- list)

{

val tokens = item.split("-")

if (tokens(0).equals("friend"))

{

friends = friends :+ tokens(1)

} else if (tokens(0).equals("group"))

{

groups = groups:+ tokens(1)

}

}

for (friend <- friends){

for (group <- groups){

result = result :+ (friend,"suggestion-"+group)

}

}

return result.distinct

}

def removeAlreadyExistingGroups (user:String, list:List[String]): List[(String, String)] = {

var result: List[(String, String)] = List()

var existingGroups : List[String] = List()

var suggestionGroups : List[String] = List()

var finalSuggestions : List[String] = List()

for(item <- list)

{

val tokens = item.split("-")

if (tokens(0).equals("group"))

{

existingGroups = existingGroups :+ tokens(1)

} else if (tokens(0).equals("suggestion"))

{

suggestionGroups = suggestionGroups:+ tokens(1)

}

}

for (group <- suggestionGroups.distinct){

if (!existingGroups.contains(group) && !finalSuggestions.contains(group)){

finalSuggestions = finalSuggestions :+ group

}

}

for (group <- finalSuggestions.distinct){

result = result :+ (user,group)

}

return result

}

Speedup

The speedup was 1 since running the job on 5 node machine and 10 node machine took the same time i.e 16 mins. Hence it was not efficient.

Result

The output is in the format (userId, groupId). The group id is the suggested group that the user can choose to join.

Sample output: <https://drive.google.com/open?id=1fOOfPaiM7hAYn-gNDd9-HGQIRmbKVnSU>

1000010,35

1000010,21

Which means the user 1000010 has been suggested to join group 35

**Task 4: Analysis of performance**

**Conclusion**

Time to execute MR job on 5 worker Machine: 2 min 50 sec

Time to execute MR job on 10 worker Machine: 1 min 7 sec

Time to execute Spark job on 5 worker Machine: 16 min

Time to execute Spark job on 10 worker Machine: 16 min

Time to execute HBase+MapReduce job on 5 worker Machine: Not Available

Time to execute HBase+MapReduce job on 10 worker Machine: 50 min, 12 sec

