CS6240 : PROJECT REPORT

TEAM MEMBERS:

1. Manikantha Dronamraju
2. Abhishek Raval

PROJECT OVERVIEW:

We set out to Compute the Page Rank of every Paper in the Bibliographic Network (BibNet) data set using Map-Reduce . Based on our initial analysis we wanted to check that the paper which would have been the most cited would have the highest page rank.

In the second task we planned to use the KNN algorithm on the paper data input such as paperID, conference, year, topics and by using the edit distance algorithm we would be predicting the nearest terms for that paper based on the input dataset we have.

Input Data:

We have use the Bibliographic Network (BibNet) data set available here for our analysis. This dataset contains two real-world graphs. The first graph (that we would be using) is a bibliographic network based on DBLP (2009) and CiteseerX (2011), which has references of citations of papers from year 1990 to 2009. The dataset has more than 2 million nodes and 25 million edges with various attributes such as Paper, Term used in paper, Author, Venue, Conference etc.

The format of the input data is in four formats:

1. Nodes.txt
   [Node ID] \tab **[Node Type]** \tab **[Some String]** \tab **[Some Integer]**
   [Node ID] a unique integer identifier assigned to every node.
   [Node Type] one of the four types: term, paper, conf and author, referring to a term, paper, venue and author respectively.
   [Some String] string representation of the node.
   [Some Integer] ignore this field for a term; year of publication for a paper, number of publications for a venue or author.
   Example:
   0 \tab term \tab systems \tab 0
   75568 \tab paper \tab High-speed three dimensional laser sensor \tab 2000
   1323803 \tab author \tab Phillip S. Yu \tab 550
   1318800 \tab conf \tab IEEE Transactions on Information Theory \tab 7346
2. **Edges.txt**
   [From] \tab **[To]**
   [From] ID of the starting node of the directed edge
   [To] ID of the ending node of the directed edge
   Example:
   1004048 \tab 568
   568 \tab 1004048
   745141 \tab 703031

Primary Task:

Since, the data we had was in four different formats, across two input files the pre-processing part that we thought would be easier was quite complicated to start with. The input files had lines anywhere across the dataset in the format of any of the above-mentioned examples. And a simple HashMap implementation in the setup function did not help us achieve our goal, as it caused Java Heap Space Out of Bound Exceptions,because as m. To complete the task in an efficient format we had to come up with five jobs for each of the following output:

1. PageId->AjacencyLists,
2. PageId->List of Authors,
3. PageId->List of Terms,
4. PageId->Conference.
5. PageId -> PaperName, Year

Each of the above-mentioned Job, implemented Map-Side Join, in such a way that Heap Space wouldn't go out of memory. So, we ran four Jobs individually on the entire dataset to achieve the desired output. With the help of the four-map side jobs performing map-side joins we have four output files each have data in one of the formats as mentioned in the example.

After Computing found Individual tasks, we combined the final output to be in the form of

[PageId | PaperName | AdjList Of Citations |AdjListSize | Number of times this page is cited
| List of Authors(authorId:authorName:totalPublicationsOfAuthor)
| Conference(conferenceId:conferenceName:conferencePublicationCount)
|List of Terms(termId:keyword)]

We had set out to compute Page Rank, but little did we know that pre-processing would be a huge task than being talked about and would take most of our time. Lesson learnt here would be that sometimes pre-processing does take significant amount of the time, so careful planning needs to be done in advanced.

PSEUDO CODE FOR PRE-PROCESSING TASK:

```
Iterators = {AUTHOR, PAPER, TERM, CONF};
iterator=0
//Iterating consecutively and Building Individual DataSet of type(paperId, attribute)
//It will iterate 4 times, for author, paper, term, conf
while(iterator<Iterators.length){
        attribute=Iterators[iterator]
        //Nodes.txt will be input to this Mapper Job1
        IterativeAttributePreprocessor{
          //Node could be either a author, paper, term or conference.
          Map(NodeId, NodeData){
            //so filtering on the basis of attribute from current iteration
            if(NodeData.type == attribute){
              emit(NodeId, NodeData)
            } }  }//Last MR Job to Merge All individual paperId, attributeDatas
```

```
AttributeMergingMapper{
   //emitting by keeping paperId as key
   Map(paperId, attributeData){
      emit(paperId, attributeData)
   }
}
AttributeMergingReduer{
   //Single Reducer function will get all the attributes for unique pageId, thus
   //appending each of them.
   Reduce(paperId,List[paper, term, author,conf]){
         for (final Text val : values) {
            String type = val.toString().split("\t")[0];
            if (val != null) {
               switch (type){
                  case PAPER:
                     paper=val.toString();
                     break;
                  case TERM:
                     term=(val.toString());
                     break;
                  case CONF:
                     venue= val.toString();
                     break;
                  case AUTHOR:
                     author=val.toString();
                     break;
               }
            }
         }
         sb.append(paper+"\t"+author+"\t"+venue+"\t"+term);
         emit(paperId,sb);
      }
}
```

PSEUDO CODE FOR PAGE RANK:

```
1: class Mapper
2: method Map(nid n, node N)
3: p ← N.PageRank/|N.AdjacencyList|
4: Emit(nid n, N) . Pass along graph structure
5: for all nodeid m ∈ N.AdjacencyList do
6: Emit(nid m, p) . Pass PageRank mass to neighbors
```

```
1: class Reducer
2: method Reduce(nid m, [p1, p2, . . .])
3: M ← ∅
```

4: for all p ∈ counts [p1, p2, . . .] do
5: if IsNode(p) then
6: M ← p . Recover graph structure
7: else
8: P'(Dangling page rank) = α (1 /|G|) + (1 − α) (m /|G| + p)
9: p ← p'
10: s ← s + p . Sum incoming PageRank contributions
11: M.PageRank ← s
12: Emit(nid m, node M)

Algorithm and Program Analysis:

The input is pre-processed in the form of Paper and its citation referred as nodes and adjacency list. The initial rank for each node is set to 1/|nodes|, the first column contains our nodes. Other columns are the nodes that the main node has an outbound link to. Mappers receive the value as (node, rank(node), adjacency list). And for each row they emit the following (node, rank(node), sum(page rank of all adjacency list/total nodes in list).

The reducers receive the values from page rank and use the formula to aggregate the values and calculate the new page rank. New page rank and old page rank values are used to calculate the convergence factor.

Instead of using a separate job to calculate the dangling mass, we are using global counters in the reduce phase that get updated and the mass is equally distributed among all other nodes.

PSEUDO CODE FOR K Nearest Neighbour:

Predicts the Keywords for a given paper.

```
//For each record in test dataset, runs a new MR job
For(t in testDataSet){
class Mapper{
Setup(){
//this will be values computed by processing string from test record.
setValues for k, tPaperId, tPaperName, tPaperYer, tAuthor, tConference;
TreeMap<Double, String> KnnMap = new TreeMap<Double, String>();
}


//match test records with final output records and compute distance metrics and emit top k
//neighbours;
map(pageId,attributes){
setValues for paperName, year,adjList,authors,conference,keywords
//Used Edit Distance for computing distance between two strings
//also, bonus points if there is any direct match for words from testpaper
adjListMetric = computeSummedAdjListDist(tPaperId, adjList)
normalisedMetric = computeAllEditDistance(paperName,tPaperName)
authorMetric = computeAllEditDistance(tAuthor,authors)
conferenceMetric = computeAllEditDistance(tConference,conference)
```

```
keywordMetric = computeAllEditDistance(paperName,keywords)
totalDist = summation of all metrics
KnnMap.put(totalDist, keywords);//if knnMap.size>k, remove lastkey
 }

cleanUp(){
foreach a in knnMap
distanceAndModel.set(a.key, a.value)
emit(null,distanceAndModel)
}
}//End Mappper

class Reducer{
reduce(null,Iterable< distanceAndModel >){
TreeMap<Double, String> KnnMap = new TreeMap<Double, String>();
//Populate another TreeMap with the distance and model information extracted from the
// DoubleString objects and trim it to size K as before.
//determines which of the K values (models) in the TreeMap occurs most frequently
// by means of constructing an intermediate ArrayList and HashMap
Map<String, Integer> freqMap = new HashMap<String, Integer>();
//Build HashMap,which determines string, and it's frequency
mostCommonModel=mostFrequentKey
context.write(null,mostCommonModel)
}
}//End Reducer

}//End of all the iterations
```

Algorithm and Program Analysis:

The pre-processor we ran, generated enriched dataset to make predictions, as we had paperId, paperName, authors, conference and keywords. The most challenging task was to design a computation model which would predict perfectly in our case. Although we were not able to predict accurately, our prediction generated at least 1-2 keywords related to the paper.
Mappers receive each test-cases iteratively, So there will be a new job for each test cases. For each test case, we'll compare it with entire dataset and compute metrics. And we'll put distance and keywords into map. We'll finally emit top k values which are closest to the given page.

There will be a single reducer task, where top k from all the mappers will be accumulated and the keyword String with most frequency will be our winner and predicted keyword for test dataset.

Instead of using a single reducer task, we can optimize it to run in parallel. Also, if we had more time, we could have figured out a way to compute predictions for all the test cases of test dataset all at once.

I ran 15test-cases for 5 machines and 11 machines and following was the time taken:

Experiments

Speedup:

Number of Test Cases	Machine	Time To Execute
15	**6**	**11 minutes**
15	**11**	**13 minutes**

Scalability:

- We were not able to run out DataSet on AWS given the time-constraints, but we experimented a lot on various types of Join, and compared it's efficiency. And decided to go with Map Side Join as it worked perfectly in out case.

Result Sample

[958747
paper 873359 677861 595158 216743 1223752 1223694
611554 445796 608961 185080 266972
author 1374085,Daniel Wu,12 1323869,Divyakant Agrawal,273 1323906,Amr El Abbadi,243
conf 1319766,Distributed and Parallel Databases,300
term 39,framework,0 49239,stratosphere,0 656,mobility,0 6467,extensibility,0]

[958748
paper 1244414 1244463 1244541 235511 449421 688516
688847 688442 688515 689771 688439 689196
877901 457662 490262 384146 169024 383139
733882 732773 734119 934834 734201 948644
315546 708143
author 1330408,Paul Watson,54 1336744,Jim Smith,361324319,Norman W. Paton,154
1329898,Rizos Sakellariou,56 1351099,Anastasios Gounaris,21 1329278,Alvaro A. A. Fernandes,60
conf 1319766,Distributed and Parallel Databases,300
term 31,adaptive,0 79,processing,0 108,environments,0 237,allocation,0
165,query,0 258,heterogeneous,0 418,autonomous,0 1567,workload,0]

[958749
paper 382282 687554 621796 688476 733200 756441
author 1326032,Kun-Lung Wu,91 1323934,Ming-Syan Chen,232 1323803,Philip S. Yu,550
conf 1319766,Distributed and Parallel Databases,300
term 211,energy,0 461,cache,0 28,mobile,0 6934,invalidation,0 19,efficient,0]

Google Drive Link to Outputs:
https://drive.google.com/open?id=1TFgoZm1oGRmaGJ5rEeTyRtPBi04JHK9v