PROJECT 3 REPORT

Project 3 Report

Prepared for: CSCI 493.66: Unix Tools and Big Data Analytics

Prepared by: Zhenghong Dong

May 7, 2014

PROJECT 3 REPORT

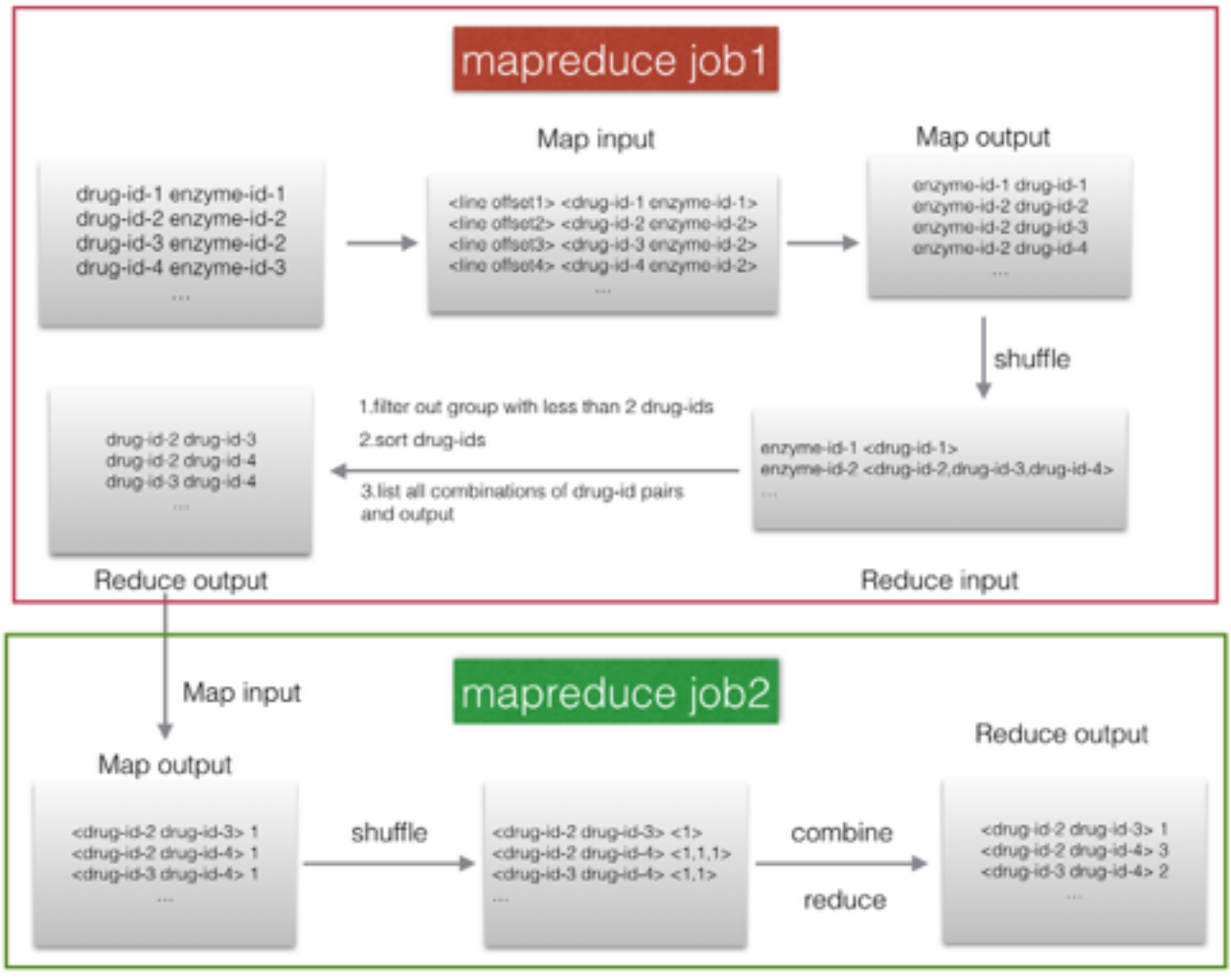
PROJECT REPORT

Objective

The purpose of this project is to develop a mapreduce program that runs on pseudo-distributed hadoop platform, reads the output data from project 2, which are lines of drug ID - enzyme ID pairs, extract the information and output a graph with drug IDs as nodes, their common enzymes as edges and enzyme numbers as weights.

Algorithm design

To obtain the objective, a straight-forward logic is to reverse the drug:enzyme pair into enzyme:drug pair and collect drugs with the same enzyme into a list, from which combinations of drug:drug pairs are generated because apparently these drug:drug pairs share the current enzyme thus can be considered as an edge of drug graph. Drug pairs are then counted and output into graph. It is unfeasible to achieve the goal with only one mapreduce job. Thus a sequential combination of two mapreduce jobs is designed to fulfill the data processing. The first step mapreduce job will output its result into a intermediate directory on the HDFS, whereas the second step mapreduce job will take the folder as input. The flow chart is depicted below and followed by detailed discussion of designing and fulfilling the algorithm.



First step mapreduce job

As depicted in the figure above, the first step mapreduce will read drug:enzyme pairs and output drug:drug pairs that share a common enzyme.

1. Map process

Since the input file consists of lines of text, each line contains a drug-enzyme pair which is divided by space, it is convenient to us TextInputFormat as the input format to parse each line into <offset>:<drug enzyme pair>. The drug-enzyme pair information is then extracted by StringTokenizer object and output as enzyme-drug pair, in which enzyme is the key and drug is the value. Apparently the input and output format should be as common: LongWritable,Text,Text,Text.

```
private final Text drugText = new Text();
private final Text enzymeText = new Text();
private final List<String> drugPair = new ArrayList<String>();
protected void map(LongWritable key, Text value, Context context) {

        //Fetch a line of data and break into fields and store into a list.
        StringTokenizer tokenizer = new StringTokenizer(value.toString());
        while(tokenizer.hasMoreTokens()){
                drugPair.add(tokenizer.nextToken());
        }

        drugText.set(drugPair.get(0));
        enzymeText.set(drugPair.get(1));
        //clear the list for next round usage.
        drugPair.clear();
        context.write(enzymeText, drugText);
}
```

2. Combiner

No combiner process needs to be set up in this mapreduce job.

3. Reduce process

Before the map output result to be read into reducer, there is a default shuffle process run by mapreduce framework. The output then turns into lines of enzyme-iterative drug collection pairs. Since each line contains an unique enzyme and a collection of drugs that are related to the enzyme, it is easy to think that all the drugs in the collection have a connection through the current enzyme. Thus, What the reduce can do is to list all the paired combinations and output as each line a pair. To make sure that two different drugs only make one pair, the collection should be sorted before generating drug pairs. The results will be written into a intermediate directory which will become the input folder for the second mapreduce job. The input and output format should be set to Text, Text, Text, Text.

```
	private List<String> drugList = new ArrayList<String>();
	protected void reduce(Text enzyme, Iterable<Text> drugs, Context context) throws IOException,
InterruptedException{

		for(Text drug:drugs){
			drugList.add(drug.toString());
		}

		//Sort the drug in current list so that the generated drug pairs are in order.
		//Thus,each pair of drugs has only 1 direction in all data.
		Collections.sort(drugList);

		int drugSize =drugList.size();

		//Filter out those enzymes with only 1 drug.
		if(2 <= drugSize){

			//list all possible couple combinations in all drugs related to current enzyme.
			for(int i=0;i<drugSize-1;i++){
				for(int j=i+1;j<drugSize;j++){
					context.write(new Text(drugList.get(i)),new Text(drugList.get(j)));
				}
			}
		}
		//Clear the list for next round usage.
		drugList.clear();
	}
```

Second step mapreduce job

Since all the drug pairs are sorted and the coupled drug pair only has one direction, what the second step mapreduce need to do is just like the common word count example job for every new beginner.

1. Map process

As word count example, this map only needs to read every line as drug pair and add an IntWritable “1” for each pair.

```
	private final static IntWritable One = new IntWritable(1);
	protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{
			context.write(value, One);
	}
```

2. Reducer and combiner

Reducer will receive lines consist of drug pair followed by an iterative collection of IntWritable “1”. What needs to be done in each line is only to iterate through the collection and sum up all the “1”s. Since all the pairs are sorted globally, the final sum will be the number of enzymes that the drug pair in current line shares. We can also set up a combiner and directly utilize our reducer because they are supposed to process same input, output format and logic.

```
        protected void reduce(Text drugPair, Iterable<IntWritable> edges, Context context) throws IOException,
InterruptedException{
                int weight = 0;
                for(IntWritable edge:edges){
                        weight += edge.get();
                }
                context.write(drugPair, new IntWritable(weight));
        }
```

Mapreduce job execution

As this program is designed to run two mapreduce jobs sequentially, and the second job is depending on the first job’s output, a few important settings need to be set.

1. Both jobs need to be configured accordingly.

2. To ensure that second job only runs after the first job correctly finished, the “waitForCompletion” property must be set to “true”.

3. Create an intermediate directory which is the output folder of the first job as well as the input source for the second job.

```
                String[] otherArgs =new GenericOptionsParser(getConf(),args).getRemainingArgs();
                if(otherArgs.length != 2){
                        System.err.println("Usage: DrugBankStat <in> <out>");
                        System.exit(2);
                }

                Path inputDir = new Path(otherArgs[0]);
                Path intermediateDir = new Path("hdfs://hd:9000/drugbank/intermediateDir");
                Path outputDir = new Path(otherArgs[1]);

                //Set up first mapreduce job.
                Job FirstStepJob = Job.getInstance(getConf(),"DrugBank Statistics step 1");
                FirstStepJob.setJarByClass(getClass());

                TextInputFormat.addInputPath(FirstStepJob,inputDir);
```

```
        TextOutputFormat.setOutputPath(FirstStepJob,intermediateDir);

        FirstStepJob.setInputFormatClass(TextInputFormat.class);
        FirstStepJob.setMapOutputKeyClass(Text.class);
        FirstStepJob.setMapOutputValueClass(Text.class);
        FirstStepJob.setMapperClass(DrugBankStepOneMapper.class);
        FirstStepJob.setReducerClass(DrugBankStepOneReducer.class);

        FirstStepJob.setOutputFormatClass(TextOutputFormat.class);
        FirstStepJob.setOutputKeyClass(Text.class);
        FirstStepJob.setOutputValueClass(Text.class);
        FirstStepJob.waitForCompletion(true);//Make sure second job only runs after the first job finished./


        //Set up second mapreduce job.
        Job SecondStepJob = Job.getInstance(getConf(),"DrugBank Statistics step 2");
        SecondStepJob.setJarByClass(getClass());

        TextInputFormat.addInputPath(SecondStepJob,intermediateDir);
        TextOutputFormat.setOutputPath(SecondStepJob,outputDir);

        SecondStepJob.setInputFormatClass(TextInputFormat.class);
        SecondStepJob.setMapOutputKeyClass(Text.class);
        SecondStepJob.setMapOutputValueClass(IntWritable.class);
        SecondStepJob.setMapperClass(DrugBankStepTwoMapper.class);
        SecondStepJob.setCombinerClass(DrugBankStepTwoReducer.class);
        SecondStepJob.setReducerClass(DrugBankStepTwoReducer.class);

        SecondStepJob.setOutputFormatClass(TextOutputFormat.class);
        SecondStepJob.setOutputKeyClass(Text.class);
        SecondStepJob.setOutputValueClass(IntWritable.class);
        return SecondStepJob.waitForCompletion(true)?0:1;
    }
```

Output statistics

Total node number is: 1001.

Total egde number is: 212526.

First mapreduce job statistics:

```
File System Counters
        FILE: Number of bytes read=68190
        FILE: Number of bytes written=513728
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=122148
        HDFS: Number of bytes written=5637952
        HDFS: Number of read operations=15
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=4
Map-Reduce Framework
        Map input records=3393
        Map output records=3393
        Map output bytes=61074
        Map output materialized bytes=67866
        Input split bytes=113
        Combine input records=0
        Combine output records=0
        Reduce input groups=225
        Reduce shuffle bytes=0
        Reduce input records=3393
        Reduce output records=352372
        Spilled Records=6786
        Shuffled Maps =0
        Failed Shuffles=0
        Merged Map outputs=0
        GC time elapsed (ms)=58
        CPU time spent (ms)=0
        Physical memory (bytes) snapshot=0
        Virtual memory (bytes) snapshot=0
        Total committed heap usage (bytes)=242360320
File Input Format Counters
        Bytes Read=61074
File Output Format Counters
        Bytes Written=5637952
```

Second mapreduce job statistics:

```
File System Counters
        FILE: Number of bytes read=4811968
        FILE: Number of bytes written=10241226
```

```
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=11398052
                HDFS: Number of bytes written=15101444
                HDFS: Number of read operations=43
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=16
        Map-Reduce Framework
                Map input records=352372
                Map output records=352372
                Map output bytes=7047440
                Map output materialized bytes=4675578
                Input split bytes=117
                Combine input records=352372
                Combine output records=212526
                Reduce input groups=212526
                Reduce shuffle bytes=0
                Reduce input records=212526
                Reduce output records=212526
                Spilled Records=425052
                Shuffled Maps =0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=26
                CPU time spent (ms)=0
                Physical memory (bytes) snapshot=0
                Virtual memory (bytes) snapshot=0
                Total committed heap usage (bytes)=336732160
        File Input Format Counters
                Bytes Read=5637952
        File Output Format Counters
                Bytes Written=3825540
```

Source code sent within separate file.