ILLINOIS INSTITUTE OF TECHNOLOGY

CLOUD COMPUTING
CS 553

Programming Asssignment2 Part1 Report

13th April, 2018

Contents

1. Introduction

As the sheer amount of data available in the world kept increasing by the day, we realized we needed a way to categorize, filter and sort it in a reliable and organized fashion. Another thing to note is that we also noticed the increasing need for large-scale storage, i.e., we'd reached a point where we just couldn't fit the entire dataset into memory. Thus a need to devise an algorithm to store data onto high capacity disks and read from them whenever necessary. This form the basis for **External Sorting.** To study more about big data processing as a part of cloud computing, we shall be implementing things in this programming assignment:

- External Merge Sort

In this assignment we are calculating the time taken for each sort to execute (excluding any extra time for the input to be loaded) and compare the results. In order to utilize all the available, the resource efficiently we are using multithreading. This assignment implementation of 2, 4 and 8 threads is done and corresponding processing time is calculated. The datasets used are of size **2 GB and 20** GB we shall sort the data using external merge sort, which breaks it down into multiple chunks, sorts them separately by reading their respective portion(files) from disk and combines them into a single sorted output file

2. External Merge Sort

2.1 Description

Simply put, external sorts are used when the data to be sorted does not fit into the main memory of the computer. It has to be written to an external disk, broken down into chunks and loaded into memory whenever the data is needed. To attain maximum efficiency, the I/O operations as well as the sort mechanism itself need to be parallelized. The time taken by the external merge to completely and successfully sort the data is recorded and stored to be compared with the results of the Hadoop experiment.

2.2 Implementation

For implementation external sort, some data is passed to determine few of the parameter required for efficient performance and resource utilization. Argument 1 determine the file to be used for sorting, Argument 2 determine the chunks size in terms of records to be loaded in the RAM for sorting. Argument 3 determine the number of thread working concurrently in this chunk to get the sorted output.

```
String sortFileSize=args[0];
int noOfThreads=Integer.parseInt(args[1]); //This represent the Number of thread to process each chunks
long maxRecordCount=Long.parseLong(args[2]); //This represents the maximum records of each chunk. Size of chunk=maxRecordCount*10
String filePath=null;
```

By default, the file is sorted by using 1 Gb of the RAM size and passing 4 thread.

The Algorithm implementation is determined as follows:

1. Determine the number of chunks to sort the file

```
long size = 100;
File file = new File(filePath);
long fileSize=file.length();
long recordsCount =  (fileSize/size);
System.out.println("Sorting of file size :"+sortFileSize);
logger.info("Sorting of file size :"+sortFileSize);
System.out.println("Total number of records(lines) in the file:"+recordsCount);
logger.info("Total number of records(lines) in the file:"+recordsCount);
//long maxRecordCount = 2500000 ;
long numberofChunks = recordsCount/maxRecordCount;
```

2. Determine the number of files required to sort. (This is calculated by number of chunks * number of threads)

```
System.out.println("Number of files = numberofChunks *  number of threads per each chunk");
logger.info("Number of files = numberofChunks *  number of threads per each chunk");
long numberofFiles = (numberofChunks * noOfThreads);
```

3. For each chunk , read the corresponding file and sort it and write to a temporary file. This is done by sorting each chunk parallelly.

The Logic implemented for sorting is as follows:

The actual “data-2GB” and “data-20GB” consists of records on each line. Each records corresponds to 100 bytes. Each records consists of key(unique and duplicate), acscii value in hexadecimal of the representing actual record position in the file. The aim was to sort the gensort input by the keys. We have overridden the comparator method for sorting the records. The First 10 byte of each records is the keys of the records and we have used it for sorting. Refer the below screenshot for reference.

```
public void run(){
    try {
            ArrayList<String> data = new ArrayList();
            String line = null;
            LineNumberReader reader;
            File inFile = new File(inputFile);
            long fileLength = (inFile.length())/100;
            reader = new LineNumberReader(new FileReader(inputFile));
            for(long i=0;i<fileLength;i++){
                line = reader.readLine();
                while(line == null && i < fileLength){
                    line = reader.readLine();
                }
                data.add(line);
            }
            //System.out.println("Array size is " + data.size());
            reader.close();
            File f = new File(inputFile);
            f.delete();
            Collections.sort(data, new Comparator<String>() {
             public int compare(String str1, String str2){
                if(str1 == null || str2 == null){
                    return 0;
                }
                 String substr1 = str1.substring(0,10);
                 String substr2 = str2.substring(0,10);
                 return substr1.compareTo(substr2);
             }
            });
            File tmpfile = new File(outputFile);
            FileWriter fileWriter = new FileWriter(tmpfile);
```

We have implemneted CountDownLatch instead of join to perform the fork operatiom. CountDownLatch acts like a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. CountDownLatch intialze count is set to the number of threads for each chunk being processed. For each file succesfully sorted in the chunk the CountDownLatch is decreased ie. Thread has completed its exc. The awaits method will block until the current count reach to zero ie. All the files in the chunks are completely excecuted by the threads.

```
for(int i=0;i<numberofChunks;i++){

    final CountDownLatch executionCompleted = new CountDownLatch(noOfThreads);
    //System.out.println("Sorting threads started");
    sortMultiThread threadSort[] = new sortMultiThread[noOfThreads];
    String tmpSortFilePath = "";
    for(int j=0;j<noOfThreads;j++){
        System.out.println("Sorting of files initialized!");
        logger.info("Sorting of files initialized!");
        if(tempIndex<10)
            tmpSortFilePath = "tmpFile0"+tempIndex;
        else
            tmpSortFilePath = "tmpFile" + tempIndex;
        threadSort[j] = new sortMultiThread(tmpSortFilePath,tmpSortFilePath+"sort",executionCompleted);
        threadSort[j].start();
        tempIndex = tempIndex + 1;
    }
```

4. Read all the sorted chunks file and perform K way merge to merge the sorted file. Each item(records) from the sorted run and pick the minimum and store it an output file. This is implemented by using Priority Queue as shown below.

```
public static void performKWayMergeSort(List<File> sortedFiles, File outputFile, int numberofFiles) {
        PriorityQueue<DataBuffer> priorityQueue = new PriorityQueue<DataBuffer>((numberofFiles + 1),
            new Comparator<DataBuffer>() {
                public int compare(DataBuffer data1, DataBuffer data2) {
                    String sub1 = data1.getBufferString();
                    String sub2 = data2.getBufferString();
                    if (sub1 == null || sub2 == null) {
                        return 0;
                    } else {
                        sub1 = sub1.substring(0,10);
                        sub2 = sub2.substring(0,10);
                        return sub1.compareTo(sub2);
                    }

                }
            });

    try {
        for (File eachFile : sortedFiles) {
            DataBuffer fileBuffer = new DataBuffer(eachFile);
            priorityQueue.add(fileBuffer);
        }

        FileWriter fileWriter = new FileWriter(outputFile);
        while (priorityQueue.size() > 0) {
            DataBuffer fileBuffer = priorityQueue.poll();
```

3. Throughput

The number of I/O throughput is calculated by the number of reads and write operation performed to get the sorted output.

4. Results

The below mentioned Shared Memory Results are obtained as per mylocal Ubuntu system configuration due to unavailability of cluster. The Linux Sort result are obtained from the cluster, when the cluster was up and running. Since Linux sort consists of executing a single sort command these results were obtained first

Experiment	**Shared Memory (1VM 2GB) 4 Thread**	**Linux Sort (1VM 2GB)**	**Shared Memory (1VM 20GB) 1 thread**	**Linux Sort (1VM 20GB)**
Compute Time (sec)	192	27.78	7369	448.12
Data Read (GB)	6	2	60	20
Data Write (GB)	6	2	60	20
I/O Throughput (MB/sec)	20.83	147.44	5.42	91.40

Experiment	**Shared Memory (1VM 2GB) 4 Thread**	**Shared Memory (1VM 2GB) 8 Thread**
Compute Time (sec)	255	157
Data Read (GB)	6	6
Data Write (GB)	6	6
I/O Throughput (MB/sec)	15.68	25.47

5. Analysis

For 2 GB input: From the result table we can conclude that execution time decreases as the number of threads operating on each chunk increases. Similarly, throughput increases as the number of threads increases. From the above analysis we can conclude that MySort works efficiently and produce correct results

For 20GB input: Since the data input is increasing the execution time and throughput will also increase. But due to unavailability of cluster and limitation of my Ubuntu system , I was could able to test this data using 1 thread and results are described in the results table.