Assignment 08: Distributed EC2 Sorting

Team Members
Vedant Naik
Dixit Patel
Vaibhav Tyagi
Rohan Joshi
Priyanka Mane

Contents

Approach

- Looking at the input data, using the metadata, we get to know the total size of all the input records.
  We divide this equally between the number of servers. This division is based on the size and not the number of input files.
- Once the divisions are made, each server is notified with the list of files which it is supposed to handle. Servers use sample sort to sort all the data in ascending order.
- The output is stored in part files on the output bucket.

Client:

1. The client starts with the input bucket. It reads all the object keys which contain 'climate' and '.txt.gz'.
2. It sorts the list based on the file sizes and then allocates one file to each server. This is done in method:

   FileSystem.getS3Parts(int numberOfServers)

   These parts are such that each server handles contains roughly equal amount of data.
3. It then issues the command to sort to each server.

Server:

1. Every server knows about the ip of every other server. This information is stored in a HashMap and used for inter-server communication.
2. Each server reads all the files it is supposed to.
   - Data records are uniquely identified as: Filename (on input s3 bucket), offset in that file, Record length, Dry Bulb Temperature Value
   - The Dry Bulb Temperature value is stored to assist in the sorting logic.
   - The class that stores this information is serialized, and the comparator has been overridden to compare only the Dry Bulb Temperature Values (sort value).
3. Once all the records are read, the server sorts the records locally and identifies pivots based on the number of servers running (as a part of Phase I of Distributed Sample Sort).
4. All servers communicate back to the client saying they have finished sorting their local list of DataRecord.
5. Once the client receives a confirmation from all servers, it issues a command telling the servers to send their pivots to the master server (which in our case is always server-0).
6. All the servers communicate with 'server-0' and a list of all pivots is maintained at 'server-0'.
7. At 'server-0', global pivots are calculated. These pivots are sent across to every other server.
8. Now, each server knows the global pivots and partitions its sorted data records using these global pivots. Partitions are made such that every server will later handle one part (example: server-0 will handle part-0, etc.)
9. These partitions are temporarily written to the server's local disk (on their respective EC2 instances). Using 'scp', the files are then transferred to the server which will handle that part.

10. Once, every server has all of its parts from every other server, it will again sort the data records. The sorted data records are then to be written into the output part file. Since the data records only contain file names and offsets, we need to read the actual data record values from the input bucket. This is done before the part files are written.
11. At the end, part files at output bucket contain the sorted outputs.

Discussion of design decisions and challenges

1. DataRecord.java

- Each file contains several data records, in 'csv' format. There are 21 fields in each record. Since we are sorting on only 'Dry Bulb Temperature' field value, we decided to not store the rest of the fields. We hence maintain only 'Dry Bulb Temperature' values for each record.
- Other record values that are required in the output, which help uniquely identify a record are; Date (YEAR_MONTH_DAY), Time and WBAN #. These values are also stored in the DataRecord object.
- Now, this object helps us identify each record uniquely and know the necessary value of that record. So, it is a self-contained data structure that we use throughout the sample sort program.
- This class implements
    - Serializable: if we choose to, this could help us send objects of this class across the network at lower a lower cost than sending a String. We are storing these objects in temp files on the EC2 instances as reading and writing binaries is faster than storing the information in text format.
    - Comparable: Since we want to sort these records, using Collections.sort(), we simply override the compareTo() to return 1, -1 or 0 based on the sortValue (in this case, 'Dry Bulb Temperature').

2. DataFileParser.java

- This is specific to the current input file structure and helps us read records more efficiently. It also makes minor validations.

3. S3FileReader.java

- In order to simplify reading input files from s3 bucket, we implemented S3FileReader which maintains an InputStreamReader for every ObjectKeyFile on the input bucket.

4. FileSystem.java

- All out disk operations specific to EC2 instances and s3 buckets are done through this class. Every FileSystem class is instantiated with the input bucket and output bucket names.
- The FileSystem also maintains AWSCredentials and AmazonS3Client objects. These are required for file operations on EC2 and s3 buckets.
- When EC2 instances are spawned, we maintain the public DNS addresses of each instance in a file called 'publicDnsFile.txt'. This file is copied to each EC2 instance before running Server.java. When Server class is created it contains an object of FileSystem which reads this file and stores the public DNS of every server in a hashmap keyed by serverNumber. This map (serverIPaddrMap) is used to send files across to EC2 instances.
- FileSystem has the following important methods:
    - readInputDataRecordsFromInputBucket – reads input records from the given fileObjectKey name.
    - getS3Parts – divides file names between servers during the initial step in client.
    - writeToEC2 – writes List<DataRecord> to the local file system on the EC2 instance, and if needed, copies the file to other EC2 instances.
    - readMyParts – reads from local file system of EC2
    - writePartsToOutputBucket – writes the final output part file to the output bucket

5. LocalFSSorter.java

- Since the server program has to sort a lot of data records in one go, sometimes the program ran out of heap space.
- To avoid reading and maintaining a large amount of data in the memory, we wrote this class which maintains a cache file which contains a sorted list of data records.
- When a new list of records is to be added, we simply sort it, and then merge it into the cache file.
- Finally, this cache file gives us a sorted list, without having to read all the data records in memory.

Work Allocation

	Vedant	Rohan	Dixit	Vaibhav	Priyanka
Files					
Client.java			X	X	
Server.java			X	X	
DataFileParser.java	X				X
DataRecord.java	X				
FileSystem.java	X				
S3FileReader.java	X				
LocalFSSorter.java					
deploy.sh		X			
sort.sh		X			
start-cluster.sh		X			
stop-cluster.sh		X			
Tasks					
Data design	X	X	X	X	X
Integration	X	X	X	X	X
Java debugging and bug fixes	X	X	X	X	
AWS communication testing	X	X	X	X	X

Every member was initially allocated a module/part of this assignment. We worked on these parts (as seen from the Files section in the table above)

Later, everyone worked together to integrate all parts, and later debug and fix error.

Comparison of 2 and 8 node execution time

Number of nodes	Execution time
2	~26 minutes
8	~7 minutes

Top 10 values in data set

Output is in the following format
WBAN,Date,Time,DryBulbTemp

Top 10 records, as given by our distributed sorter are:

40604,19960915,1550,558.5
26627,19980604,1152,251.6
14847,19960714,1150,245.1
26442,19980818,1250,237.6
40604,19971016,1954,214.0
40504,19971107,0056,212.4
93115,19991213,0855,201.0
14780,19991207,1155,201.0
14611,19991210,1855,201.0
03866,19991215,0955,201.0