A Study on Query Processing through Node.js implementation of MapReduce paradigm on a multi-node cluster

Final Year Project Report submitted
in the partial fulfillment of the requirements for the degree
of
Bachelor of Computer Science & Engineering

Submitted By

Anurag Chakraborty
Class Roll No.: 001610501079
Registration No.: 135985 of 2016-17

Under the guidance of

Dr. Nandini Mukherjee

Professor, Department of Computer Science & Engineering, Jadavpur University

Department of Computer Science & Engineering Faculty of Engineering & Technology Jadavpur University Kolkata-700032

ACKNOWLEDGEMENT

I owe my heartiest gratitude to my project advisor Prof. Nandini Mukherjee, for her esteemed guidance and encouragement throughout the project work. Without her constant support and motivation the work would not have completed. It was our privilege to do final year project work under her supervision.

I would like to acknowledge with much appreciation, the crucial role of Himadri S. Roy, for providing us with all necessary resources and helping us throughout the project work. Without his contribution, our work couldn't have accomplished.

And last, but not the least, I would also like to thank my project mate, Radib Kar, joint collaboration with whom, helped in bearing the fruits of our experiment.

Anurag Chakraborty

Contribution

In the first semester we built the tool for single-node whereas we extended the idea to multinode in our second semester. As per as my contribution is concerned:

On the single node architecture, Radib wrote the first draft of program logic for map & reduce function for executing queries, using the mapred library and I optimized this standard node.js library and came up with a better mapred module (our own map, reduce and combinator code), which was much more efficient. Finally we ran the entire thing on the weather data using our own mapred module and mapreduce logic to evaluate queries.

Later we extended the architecture to multinode cluster and there Radib handled the client side programing (child nodes) and I handled the server side programing (master node) and Himadri da helped us with the data communication. Also, Himadri da took care of generating the final result via executing on Rasberry Pis clusters.

Contents

1. Abstract

MapReduce has been a profound model for handling data-intensive applications. It is a model with high scalability, simple interface of programming, and capable of processing a high portion of data in the distributive computing environment. This fault-tolerant framework was presented by Google in 2004 to tackle the issue of processing a large amount of data referencing to the internet-based application. The easy availability and accessibility of the MapReduce platforms, such as Hadoop, makes it sufficient for a productive parallelization and execution of data intensive tasks.

Our implementation of MapReduce is mainly focused to a predictive analysis of a health and a weather dataset in Node.js to perform a comparative analysis with the state-of-art MapReduce implementation. We have tested on both multi-node & single node clusters to bring wide analysis and comparative results. We also present the future scope for running the implementation on a single/multi-node cluster of commodity machines using the same Node.js framework.

Keywords: *MapReduce***,** *Node.js***,** *Parallel processing, Distributive computing, Query analysis, Single/multi node cluster*

2. Introduction

Nowadays, with the excessive growth in the information and data, their analysis has become a burdensome challenge. MapReduce being such simple, fault-tolerant framework enables users to process large amount of data. It is a framework for efficient large-scale data processing which is presented by Google in 2004 in order to tackle the issue of processing large amounts of data with reference to the Internet-based applications. Also, it helps us to felicitate the "NoSQL" trend to process unstructured dataset. In our project work, we have done a predictive analysis of a given weather dataset based on some selective query. We used Node.js to implement the MapReduce paradigm and compared performances. The MapReduce can automatically run the applications on a parallel cluster of hardware and in addition, it can process terabytes and petabytes of data more rapidly and efficiently.
Therefore, its popularity has grown swiftly for diverse brands of enterprises in many fields. It provides a highly effective and efficient framework for the parallel execution of the applications, data allocation in distributed database systems, and fault-tolerance network communications. The main objective of MapReduce is to facilitate data parallelization, data distribution and load balancing in a simple library. Though there are popular availability of MapReduce platforms like Hadoop, we have shifted from the state-of-art implementations like Hadoop, Phoenix and Mars etc.

Distributed computing:

The word *distributed* in terms such as "distributed system", "distributed programming", and "distributed algorithm originally referred to computer networks where individual computers were physically distributed within some geographical area. The terms are nowadays used in a much wider sense, even referring to autonomous processes that run on the same physical computer and interact with each other by message passing.

While there is no single definition of a distributed system, the following defining properties are commonly used as:

- There are several autonomous computational entities (*computers* or *nodes*), each of which has its own local memory.

- The entities communicate with each other by message passing.

A distributed system may have a common goal, such as solving a large computational problem; the user then perceives the collection of autonomous processors as a unit. Alternatively, each computer may have its own user with individual needs, and the purpose of the distributed system is to coordinate the use of shared resources or provide communication services to the users.

Other typical properties of distributed systems include the following:

- The system has to tolerate failures in individual computers.
- The structure of the system (network topology, network latency, number of computers) is not known in advance, the system may consist of different kinds of computers and network links, and the system may change during the execution of a distributed program.
- Each computer has only a limited, incomplete view of the system. Each computer may know only one part of the input

MapReduce as a distributive computing

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into *mappers* and *reducers* is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

MapReduce architecture

Google created MapReduce to process large amounts of unstructured or semi-structured data, such as web documents and logs of web page requests, on large shared-nothing clusters of commodity nodes. It produced various kinds of data such as inverted indices or URL access frequencies. The MapReduce has three major parts, including Master, Map function and Reduce function. The Master is responsible for managing the back-end Map and Reduce functions and offering data and procedures to them. A MapReduce application contains a workflow of jobs where each job makes two user-specified functions: Map and Reduce. The Map function is applied to each input record and produces a list of intermediate records. The Reduce function (also called Reducer) is applied to each group of intermediate records with the same key and produces a list of output records. MapReduce program is expected to be done on several computers and nodes when it is performed on Hadoop. Therefore, a master node runs all the necessary services to organize the communication between Mappers and Reducers. An input file (or files) is separated into the same parts called input splits. They pass to the Mappers in which they work parallel together to provide the data contained within each split. As the data is provided by the Mappers, they separate the output; then each Reducer gathers the data partition by each Mapper, merges them, processes them, and produces the output file. An example of this data flow is shown in Fig. 1. The main phases of MapReduce architecture are Mapper, Reducer, and shuffle which are presented below:

Mapper: a Mapper processes input data which are assigned by the master to perform some computation on this input and produce intermediate results in the form of key/value pairs.

Reducer: The Reduce function receives an intermediate key and a set of values of the key. It combines these values together to form a lesser set of values.

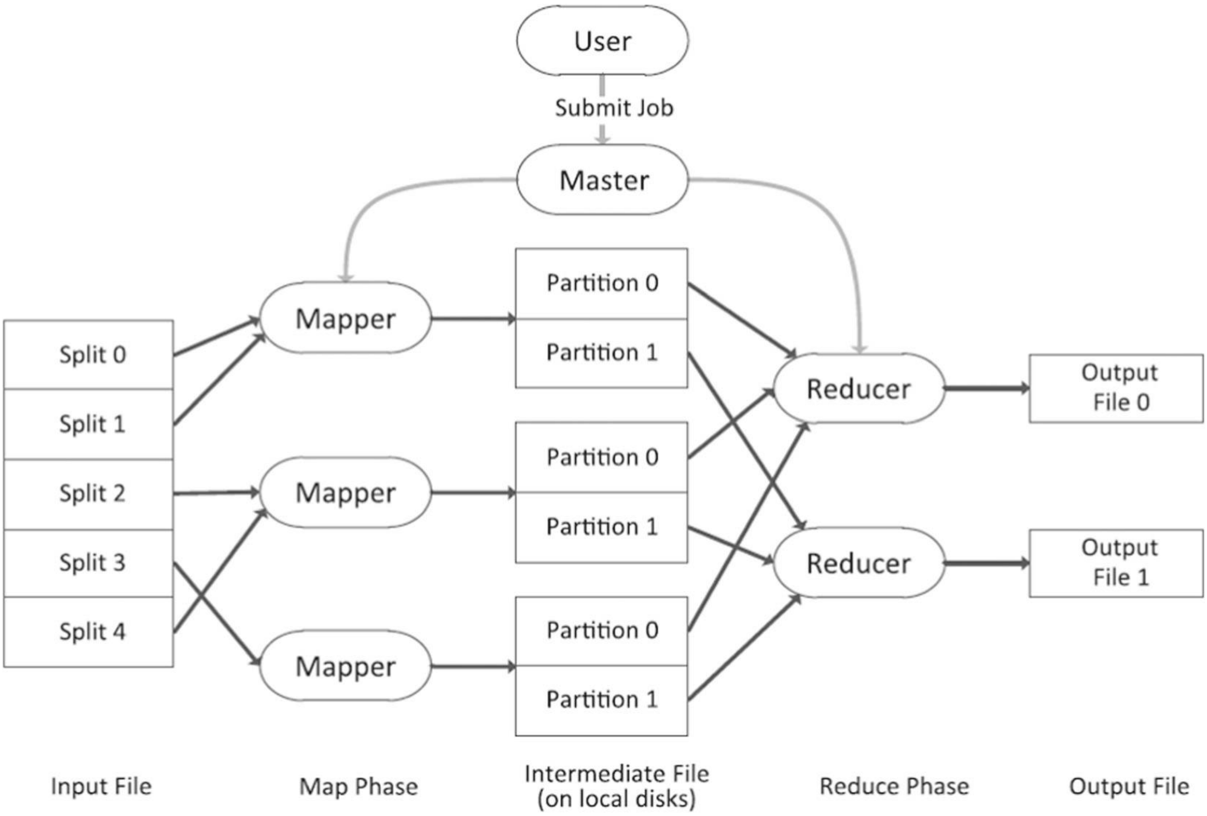


Figure 1: MapReduce Architecture

We have developed a Node.js based MapReduce and performed analysis of the existing weather dataset using selective queries in selective node. Also, we have extended the application to work on multimode cluster and tested on a health data set. In section 2, we have discussed few pros and cons of existing MapReduce implementations, state-of-art methods and section 3-4 gives our detailed present work. Section 5 contains the experiment and performance analysis where section 6 briefs our future work and section 7 to conclude.

3. Related works and State-of-art implementations of MapReduce and Comparisons

1. Google's MapReduce:

Google performs the original MapReduce implementation aims to have large clusters of networked machines. The first version of the MapReduce library was written in February 2003, but it gets some significant changes. Its library automatically handles parallelization and data distribution. Google File System (GFS) as a distributed file system makes a duplicate of data blocks on multiple nodes for enhanced reliability, fault tolerance and is intended to view machine failures as a default rather than an irregularity. MapReduce is highly scalable. Therefore, it can be run on clusters comprising of thousands of low-cost machines.

2. Hadoop:

MapReduce has some other implementations, including Mars, Phoenix, Hadoop and Google's implementation.

Among them, Hadoop has become the most popular one due to its open source feature. The most general implementation of the MapReduce model is the Hadoop framework, which lets applications to run on large clusters. It is applicable for performing the cloud-based large-scale data-parallel applications by providing the reliability and data transfer capabilities. Apache's Hadoop is an open-source implementation of Google's Map/Reduce framework. It enables distributed, data-intensive and parallel applications by analysing a great job into smaller tasks and a massive data set into smaller partitions in a way that each task processes a different partition in parallel. HDFS, the Hadoop Distributed File System, is a distributed file system which is designed (Fig. 2) to hold an immense number of data (terabytes or even petabytes) and provide high throughput access to the information. HDFS consists of a single ***NameNode*** and a master server to manage the file system and provide the access to files by the clients. Furthermore, a file breaks into one or more blocks that they are stored in a set of ***DataNodes***. The ***DataNodes*** are responsible for doing 'read' and 'write' requests from the file system's clients. They also perform block creation, replication, and deletion. High-throughput, fault tolerance and elastic scalability are the features of Hadoop and Google frameworks for MapReduce. Amazon, Facebook, Twitter are considered as leading technology business based on a policy of co-locating and processing data to accelerate the performance.

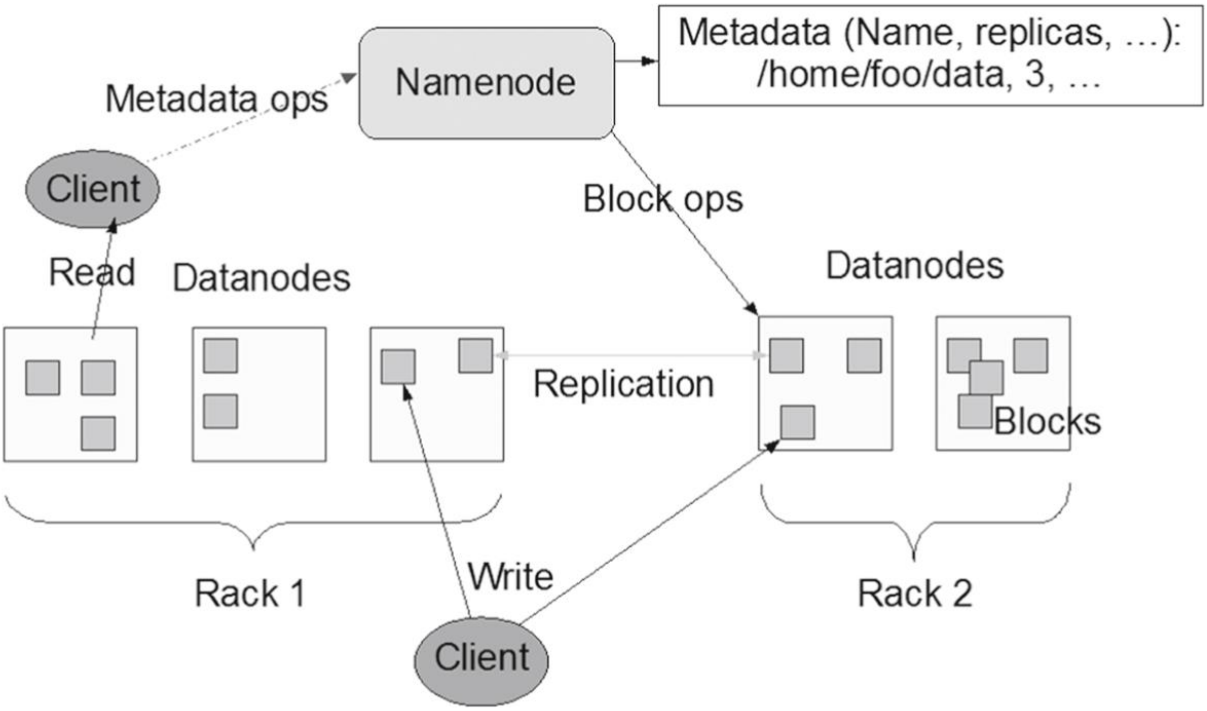


Figure 2: Hadoop architecture

3. Hadoop+

Hadoop+ is a heterogeneous MapReduce framework, which enables GPUs and CPUs for processing the big data and leveraging the heterogeneity model to assist users in selecting the computing resources for different purposes. Figure 3 shows the summary of the Hadoop+ framework. Hadoop provides the Map and Reduce primitives and PMap and PReduce are provided by Hadoop+ to programmers. The PMap and PReduce in Hadoop+ enable programmers to write explicit parallel CUDA/OpenCL functions running on GPUs as plug-ins, as shown by the box of "User-Provided PMap/PReduce Function" in Fig. 3. Meanwhile, users can also use the Map and Reduce functions in Hadoop. In Hadoop+, users can provide Map, PMap or both, and Reduce, PReduce or both. Hadoop+ provides different input parameters for Map and PMap to support explicit parallel Map functions. In particular, key and value are the input of Map, while the input of PMap is a dataset, i.e., a list of (key, value). Meanwhile, Reduce and PReduce have the same input parameters, which are the outputs with the same key from all Map tasks.

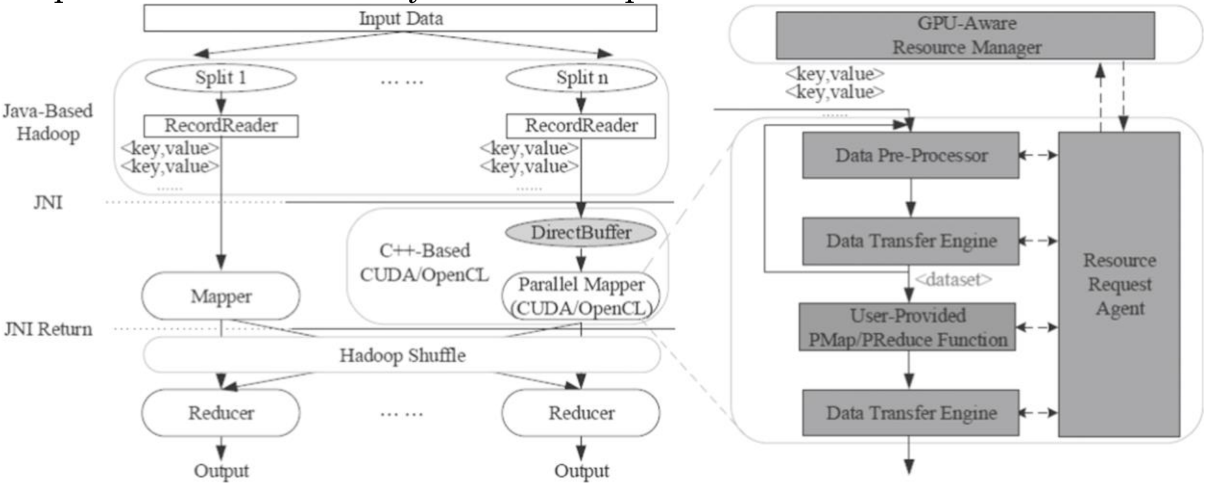


Figure 3:Hadoop+ architecture

4. GridGain

The GridGain1 is another open-source MapReduce implementation. It is an enterprise open-source grid computing made for Java. The GridGain and Hadoop DFS are compatible and it provides a substitute to Hadoop's MapReduce. GridGain offers a distributed, real-time, in-memory and scalable data grid to have a connection between applications and data sources. From the technological point of view, the biggest difference is the primary process of Map tasks assignment to the nodes. GridGain acts as a bridge based on Java in the processing of big data to have high performance and finishes fast in-memory MapReduce implementation. The programmer can process terabytes of data, on 1000s of nodes in less than a second using GridGain. In the MapReduce algorithm, the task is split into subtasks and workers drag the split parts as soon as they have free processor time. In GridGain, the subtasks are pushed to the nodes. This proves to be an advantage since it gives more load balancing capabilities. This benefit depends on situations and the user's needs. As a part of the extra functionality, it introduces some additional complexity that the developer has to plan in advance so that no worker does stay without reason idle. Although GridGain has less popularity, it shows to be better documented and it is interested in beginners.

5. Mars

Mars is developed for Graphics Processing Units (GPUs) using MapReduce framework. It limits the GPUs complication by means of MapReduce connection and automatically takes task partitioning, data distribution, and parallelization on the processors forward. Figure 4 illustrates the workflow of Mars, assuming the data resides on the disk at the beginning. Its planner operates on the CPU and plans tasks to the graphics processing unit. Mars has three stages, including Map, Group, and

Reduce. Before the Map stage, Mars pre-processes the input data from the disk, transforming the input data to key/ value pairs (input records) in the main memory. Following that, it transfers input records from the main memory to the graphics processing unit device memory. In the Map stage, Map Split dispatches input records to GPU threads that the workload for all threads is even. Each thread performs the user-defined MapCount function to calculate a local histogram of the number. Then, the runtime performs a graphics processing unit –based prefix sum on the local histograms to obtain the output size and the write position for each thread. Finally, after the output buffer is allocated to the device memory, each graphics processing unit thread executes the user-defined Map function and outputs the results. There will be no write conflict between concurrent threads when each thread has computed its write position before and it has no contrast with other threads. In the Group stage, the sort-based and hash-based strategies are available for grouping records by key. Yet, some applications need sorting all output records and the hash-based strategy has to perform additional sort within each hash bucket. In the Reduce stage, Reduce Split dispatches each group of records with the identical key to a graphics processing unit thread.

However, it causes a load imbalance between threads, since the number of records of different groups may vary widely. Furthermore, the MapReduce framework of Mars enables the integration of graphics processing unit-accelerated code to distributed environment, like Hadoop, with the least effort.

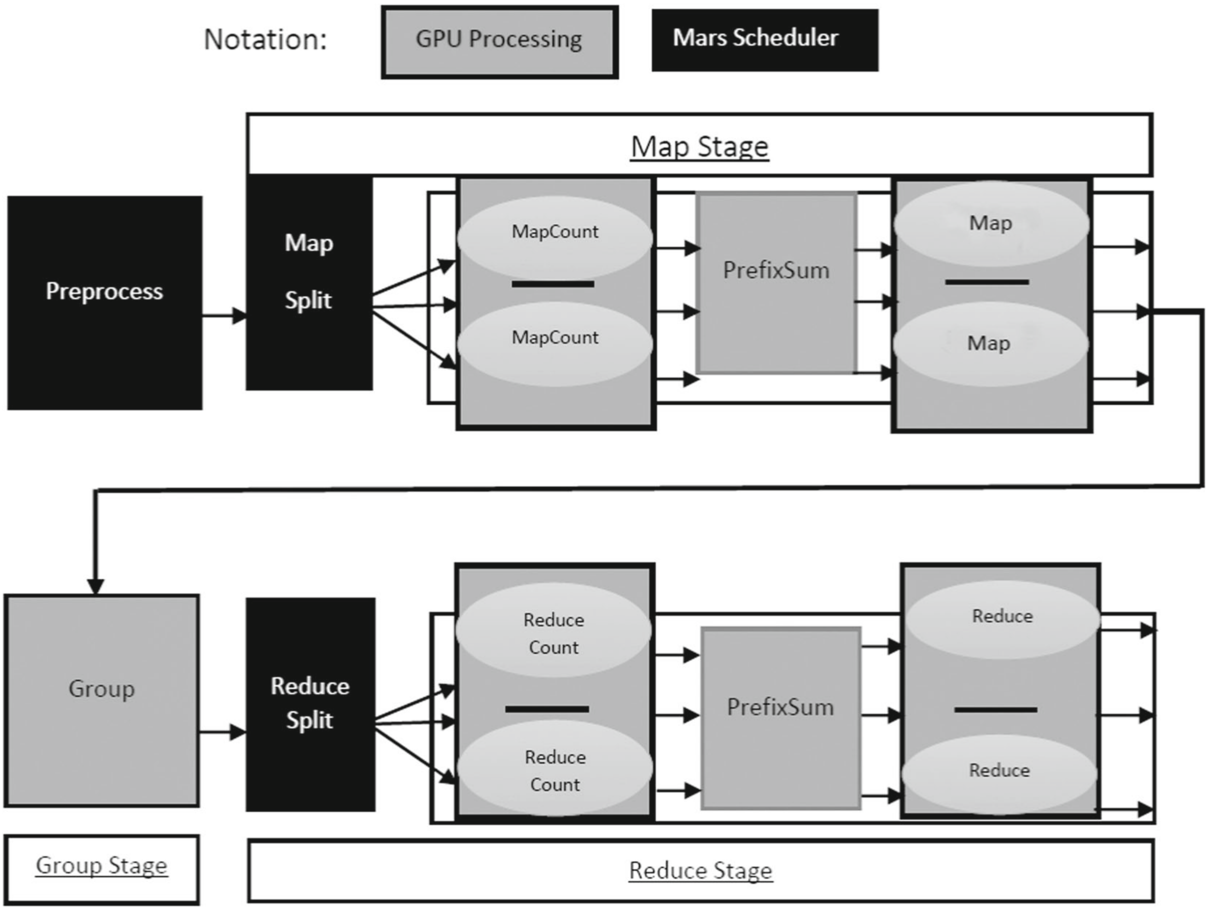


Figure 4: Mars Architecture

6. Phoenix

Phoenix implements MapReduce for shared-memory systems and it aims to support efficient implementation on multiple cores without troubling the programmer with concurrency management. It has a simple API, which has (Application Programming Interface) an efficient runtime to apply parallelization, resource management, and fault recovery. It is used by application programmers to target for multi-core and multiprocessor systems. Parallelism is performed by shared-memory *threads*. First, a user provides the runtime with the Map/Reduce functions for applying on the data. The runtime uses multiple worker threads to execute the computation. In the Map phase (see Fig. 5), the input data are split into *chunks*, and the Map function is invoked on each chunk. This generates intermediate key/value pairs. In Reduce phase, for each unique key, the Reduce function is called with the values for the same key as an argument and reduces them to a single key/value pair. The results of the Reduce tasks are merged and sorted by keys to yield the last output.

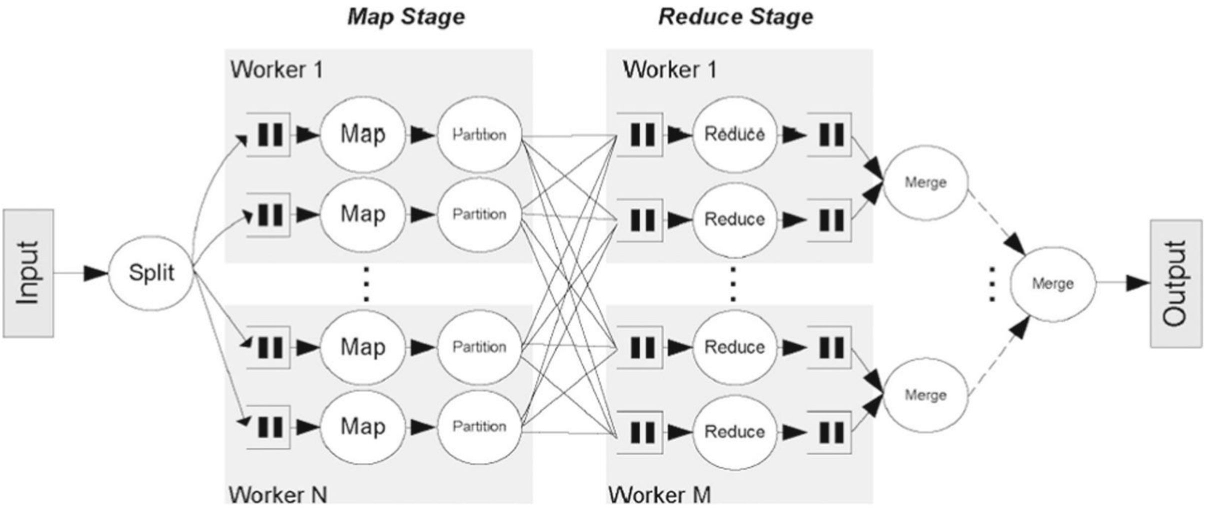


Figure 5: Phoenix Architecture

4. Basic Principle of Implementation

Key Concepts

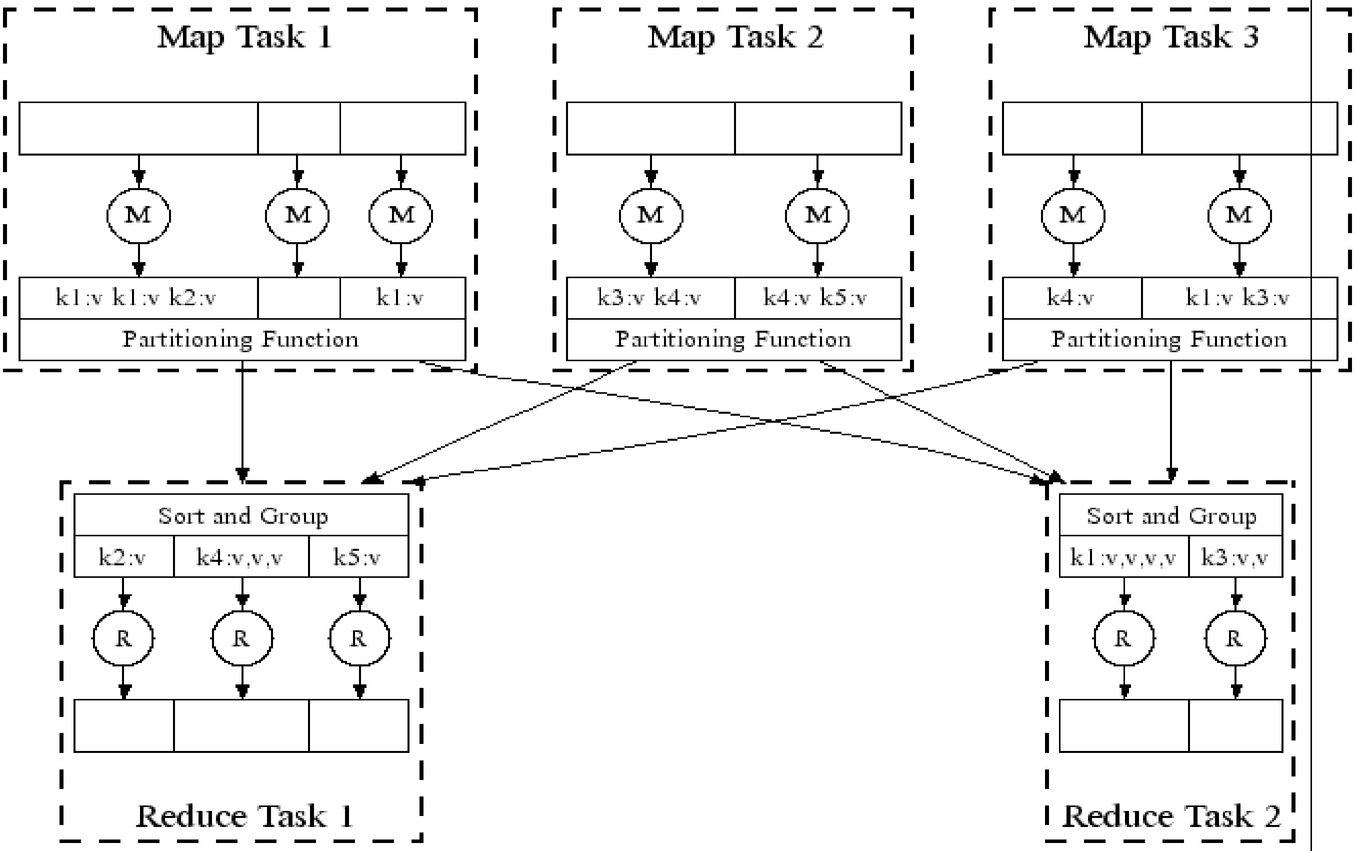
It consists of two main steps, a map and a reduce function. The program takes as input a file with <key, value> pairs.

The map function processes a <key, value> pair to generate many intermediate <key', value'> pairs.

An intermediate collector/aggregator function groups all pairs of the same <key'> value together.

Finally, the reduce function merges all intermediate <value'>s associated with the same intermediate <key'>, to produce a single <value">.

The final output consists of <key', value"> pairs.



MapReduce in Parallel

5. Single Node Implementation Code

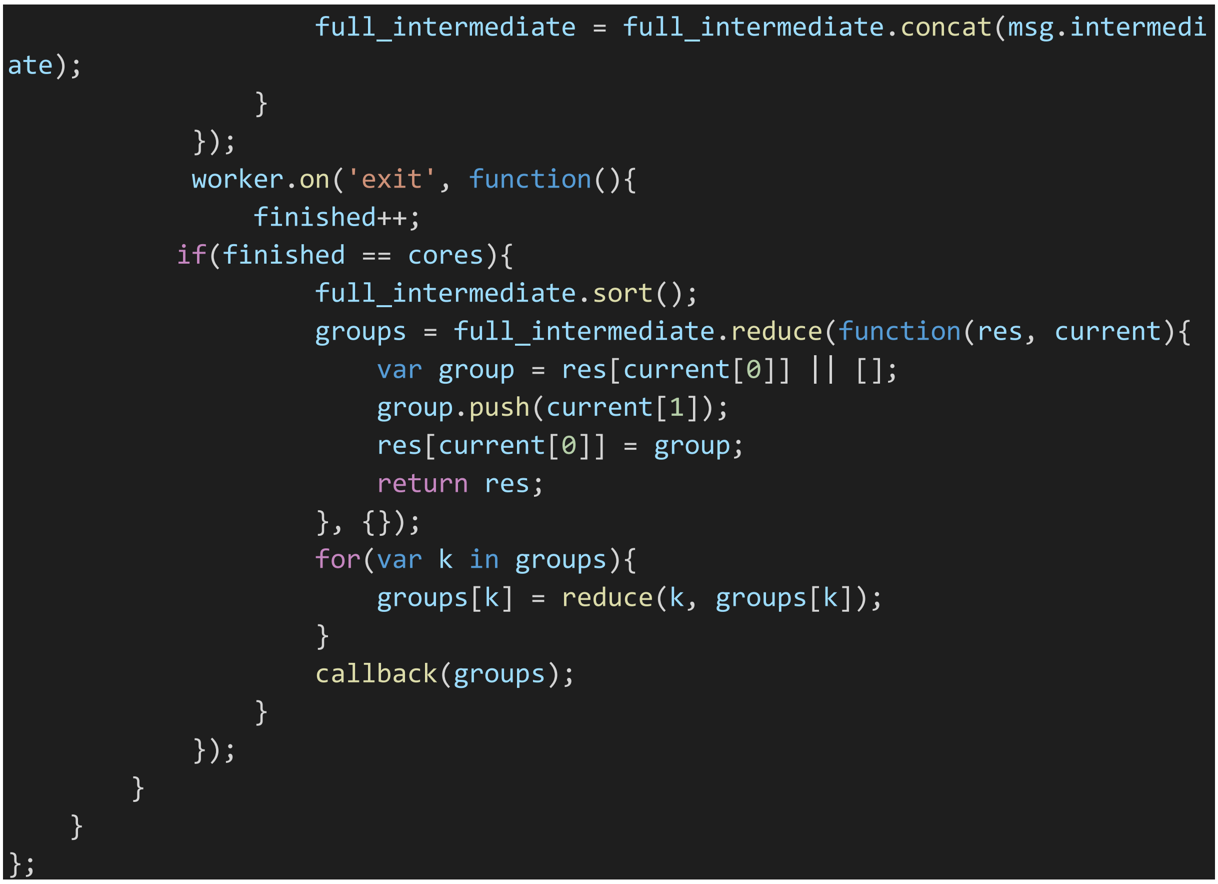
```
var XLSX    = require('xlsx'),
    cluster = require('cluster'),
    cores   = require('os').cpus().length;
```

The two main nodejs modules used in the implementation are *xlsx* and *cluster*.
Xlsx module is used for parsing and writing files in spreadsheet formats.
Cluster allows us to launch multiple instances of a Node.js process, to take advantage of multicore systems, similar to fork() used in C on linux.
It allows easy creation of worker instances, and also provides in-built IPCs to communicate between the master and the worker. The process.send() function is used here to send data from the worker to the master.
The os module has been used to detect the no. of cores of the system, according to which the no. of child process instances will be launched.
The structure of the program has been significantly changed, from the original format of the program which was submitted in the previous semester. The major changes that were brought in, are :-

- The large input file has been divided into smaller pieces.
- The task of reading the input excel file, has been transferred to each worker process that is launched, and each such process is assigned a unique smaller piece of the larger original input file.
- Rest of the logic has stayed the same. Each process takes in as input its assigned file, processes it, into the intermediate key value pairs, and then sends it to the Master process.

This has boosted the overall speed of the program, and reduced the overall execution time of the program, by a large margin.

```
var mapreduce = function(pieces, map, reduce, callback) {
    if(cluster.isMaster) {
        for(var i = 0; i < cores; i++) {
            var worker = cluster.fork(), finished = 0, full_intermediate = [];

            worker.on('message', function(msg){
                if(msg.about == 'mapfinish'){
```



```
                    full_intermediate = full_intermediate.concat(msg.intermedi
ate);
                }
            });
            worker.on('exit', function(){
                finished++;
           if(finished == cores){
                    full_intermediate.sort();
                    groups = full_intermediate.reduce(function(res, current){
                        var group = res[current[0]] || [];
                        group.push(current[1]);
                        res[current[0]] = group;
                        return res;
                    }, {});
                    for(var k in groups){
                        groups[k] = reduce(k, groups[k]);
                    }
                    callback(groups);
                }
            });
        }
    }
};
```

The mapreduce function defined takes a total of 4 arguments –

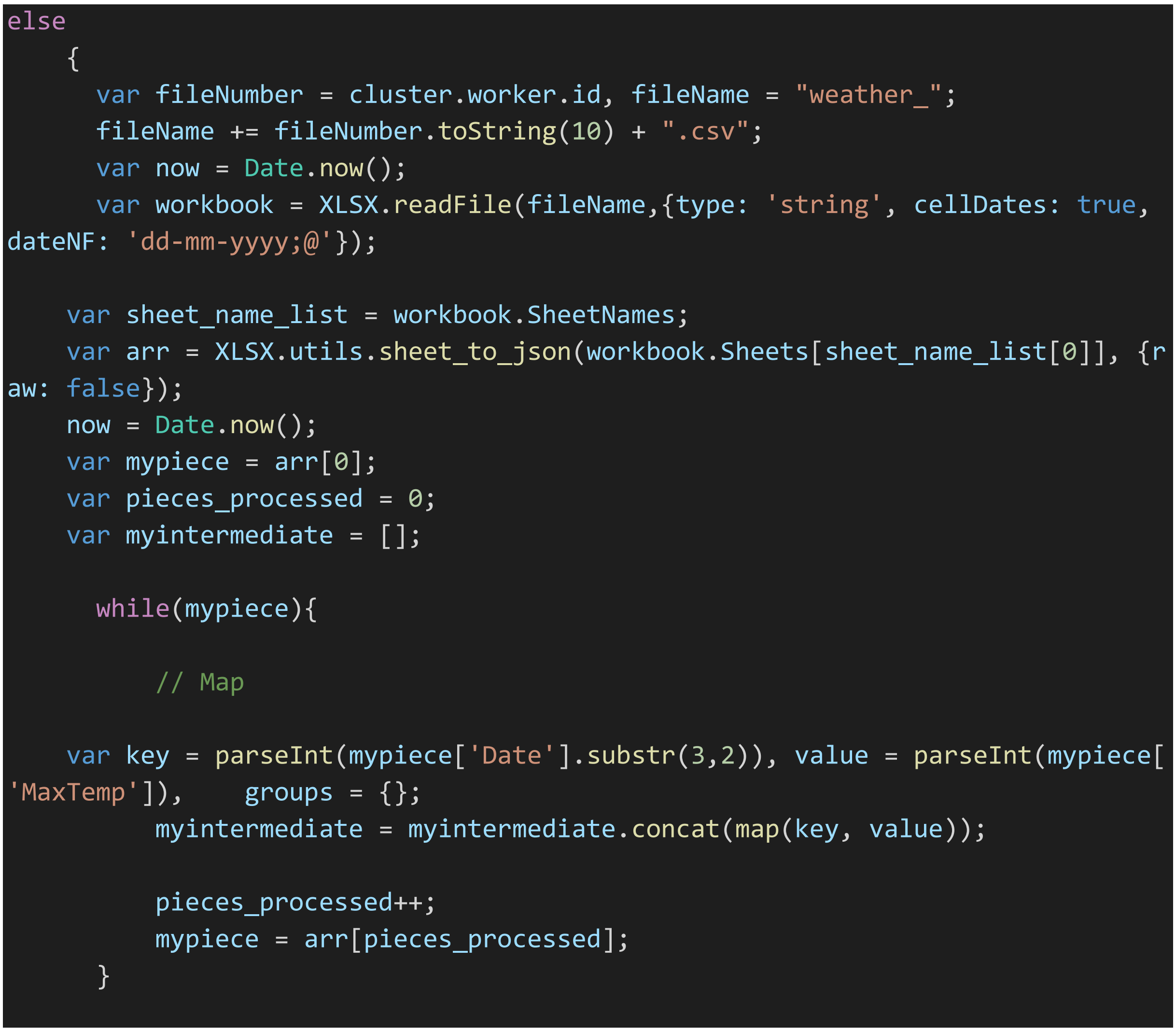
- pieces – The input, which is the array of JSON objects
- map & reduce – The user specified map, reduce functions
- callback – The function to call, after the function execution completes

The cluster.isMaster returns true for the master instance, which performs the collector/aggregator role. From the Master instance, the child instances are launched from a loop, which iterates 'cores' no. of times.
cluster.fork() returns an object of worker type, which is used to receive data from that particular worker.
worker.on('message', callback) receives the message from the worker, and checks what it is about (msg.about). If it is 'mapfinish', then the intermediate results generated by the worker are added to the full_intermediate list.
worker.on('exit', callback) listens for the worker to finish execution. It increments the 'finished' variable, and if it is equal to cores (no. of worker instances), it indicates that the map function is complete. The Master instance then sorts the entire full_intermediate list which contains the intermediate key value list formed.

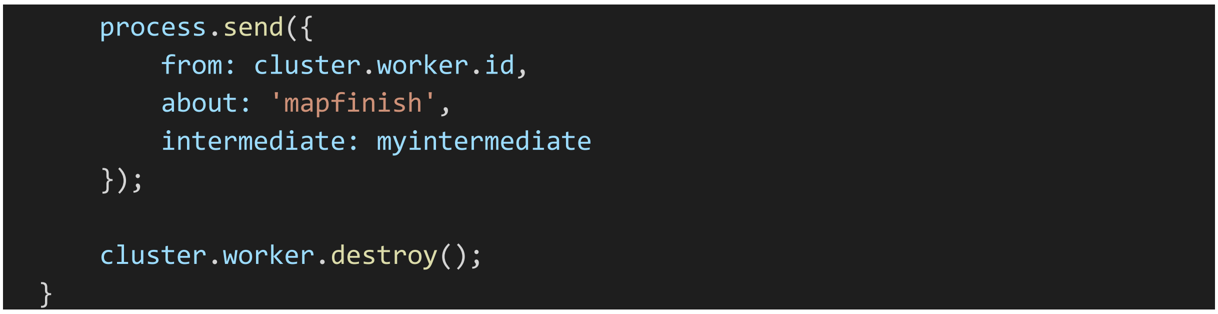
It then calls the reduce function on the list. The res variable is a JSON object, and acts as an accumulator with initial value { }, the current variable is an element of the full_intermediate list in the [key, value] form. The group variable – res[current[0]] is temporarily used to hold the list, and has the 'value' – current[1] appended to it. Then the new res[current[0]] is group. After this the groups object obtained is of the form:

{ key1: [value1, value1', value1", …],
key2: [value2, value2', value2", …],
….
keyN: [valueN, valueN', valueN", …] }

After that, for each key in the groups object, the user reduce function is called. The reduce function returns a single condensed value for each key, which is updated in the groups map object, and passed as argument to the callback function.



```
else
    {
      var fileNumber = cluster.worker.id, fileName = "weather_";
      fileName += fileNumber.toString(10) + ".csv";
      var now = Date.now();
      var workbook = XLSX.readFile(fileName,{type: 'string', cellDates: true,
dateNF: 'dd-mm-yyyy;@'});

    var sheet_name_list = workbook.SheetNames;
    var arr = XLSX.utils.sheet_to_json(workbook.Sheets[sheet_name_list[0]], {r
aw: false});
    now = Date.now();
    var mypiece = arr[0];
    var pieces_processed = 0;
    var myintermediate = [];

      while(mypiece){

          // Map

    var key = parseInt(mypiece['Date'].substr(3,2)), value = parseInt(mypiece[
'MaxTemp']),    groups = {};
          myintermediate = myintermediate.concat(map(key, value));

          pieces_processed++;
          mypiece = arr[pieces_processed];
      }
```



```
      process.send({
          from: cluster.worker.id,
          about: 'mapfinish',
          intermediate: myintermediate
      });

      cluster.worker.destroy();
  }
```

When cluster.isMaster is false, the else part is executed which is basically the worker instance code. Each worker has its own id, which can be obtained by cluster.worker.id and it is 1-based indexing. The main changes from the original implementation of the single node program have been made here. A string of the form "weather_" is first created, and then the file specific to that worker process is opened, by appending the worker id to the string. Since the original file has been broken into equal parts, into the same no. as the no. of cores on the single node workstation, the time spent on reading the files has been reduced. In the previous implementation, there was a constant race condition due to the limited availability of the main input file, because of which the other worker nodes were having to sit idle, before the file could be made available.

XLSX.readfile opens the specified file (weather.xlsx). The options specified while opening the file are, type: 'string' – characters interpreted as JS string (UTF-8), cellDates: true – store dates as type d, JS date object to be parsed as date, and dateNF: 'dd-mm-yyyy;@' – overrides generic date 14 format string.

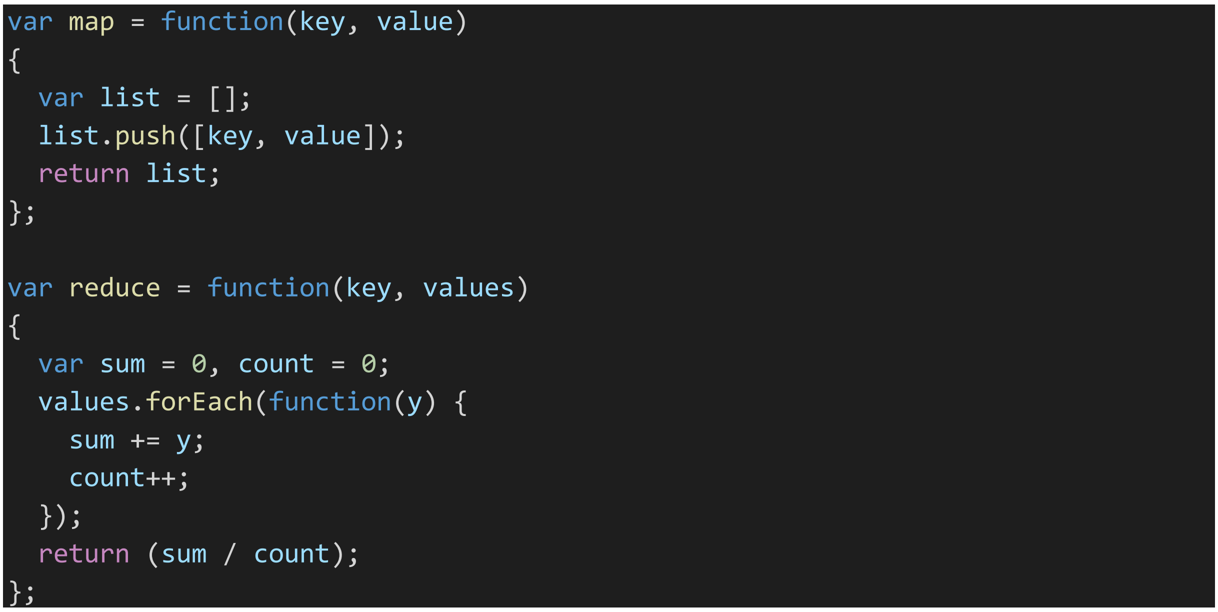
workbook.Sheetnames returns a list of all excelsheets present in the workbook.

XLSX.utils.sheet_to_json converts the records in the excelsheet into JSON objects so that it is easier to handle the cell values.

At first, the mypiece variable is initialized as a member of the pieces list, according to its current worker id. The myintermediate variable will hold the list of the intermediate key, value pairs generated by the user map function. The pieces_processed variable is incremented, and mypiece is updated as a multiple of the cluster.worker.id. If all the values are processed, mypiece will be NULL, and the loop will end.

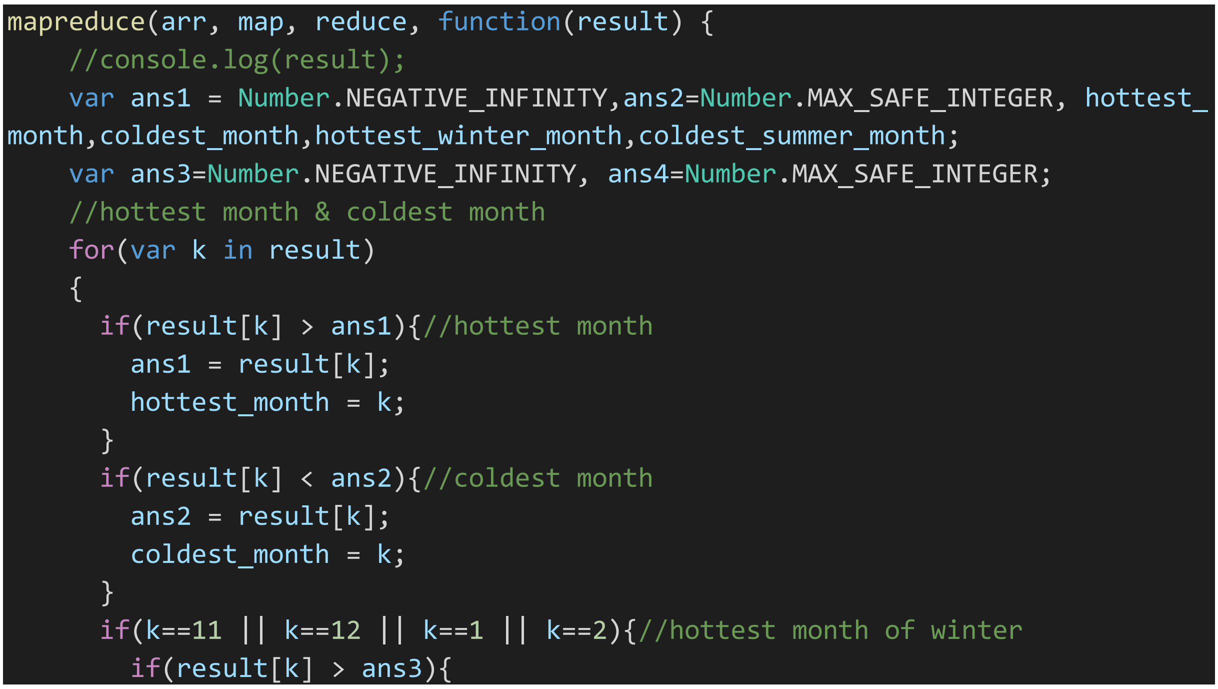
Using process.send(), the worker then sends all the intermediate values to the Master instance, which will collect these values similarly from all other worker instances.

The cluster.worker.destroy() is called to signal the Master that the worker has finished executing, so that it can start operating on all of them, after all the worker instances get destroyed.

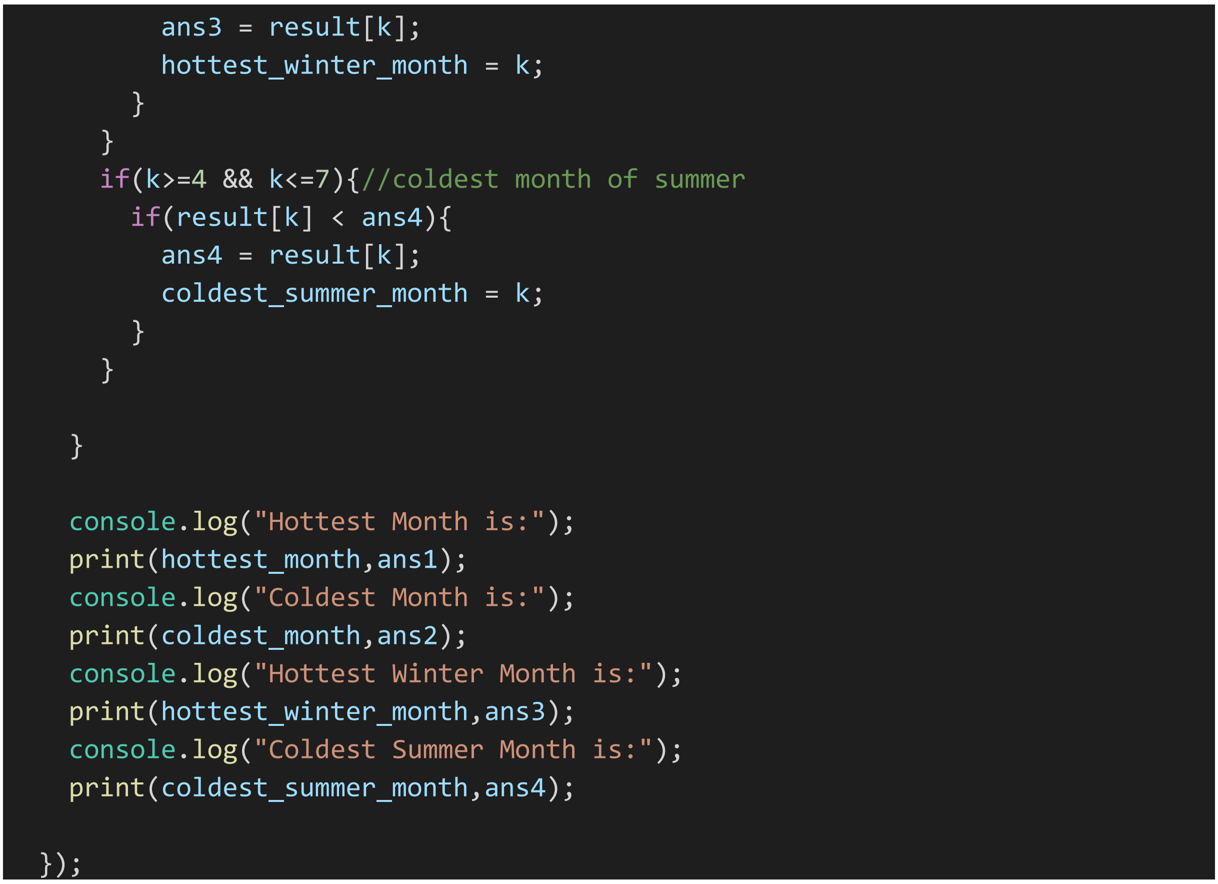


```
var map = function(key, value)
{
  var list = [];
  list.push([key, value]);
  return list;
};

var reduce = function(key, values)
{
  var sum = 0, count = 0;
  values.forEach(function(y) {
    sum += y;
    count++;
  });
  return (sum / count);
};
```

The map and reduce function definition are to be specified by the user, according to the main logic of the program.
The map function here is not required, so is only used as a placeholder to return the arguments in the [key, value] format.
The reduce function receives a list of values for every corresponding key. In this case, it takes the list, sums up all the values, to calculate the average for every key. It returns this value.



```
mapreduce(arr, map, reduce, function(result) {
    //console.log(result);
    var ans1 = Number.NEGATIVE_INFINITY,ans2=Number.MAX_SAFE_INTEGER, hottest_month,coldest_month,hottest_winter_month,coldest_summer_month;
    var ans3=Number.NEGATIVE_INFINITY, ans4=Number.MAX_SAFE_INTEGER;
    //hottest month & coldest month
    for(var k in result)
    {
      if(result[k] > ans1){//hottest month
        ans1 = result[k];
        hottest_month = k;
      }
      if(result[k] < ans2){//coldest month
        ans2 = result[k];
        coldest_month = k;
      }
      if(k==11 || k==12 || k==1 || k==2){//hottest month of winter
        if(result[k] > ans3){
```



```
          ans3 = result[k];
          hottest_winter_month = k;
        }
      }
      if(k>=4 && k<=7){//coldest month of summer
        if(result[k] < ans4){
          ans4 = result[k];
          coldest_summer_month = k;
        }
      }

    }

    console.log("Hottest Month is:");
    print(hottest_month,ans1);
    console.log("Coldest Month is:");
    print(coldest_month,ans2);
    console.log("Hottest Winter Month is:");
    print(hottest_winter_month,ans3);
    console.log("Coldest Summer Month is:");
    print(coldest_summer_month,ans4);

  });
```

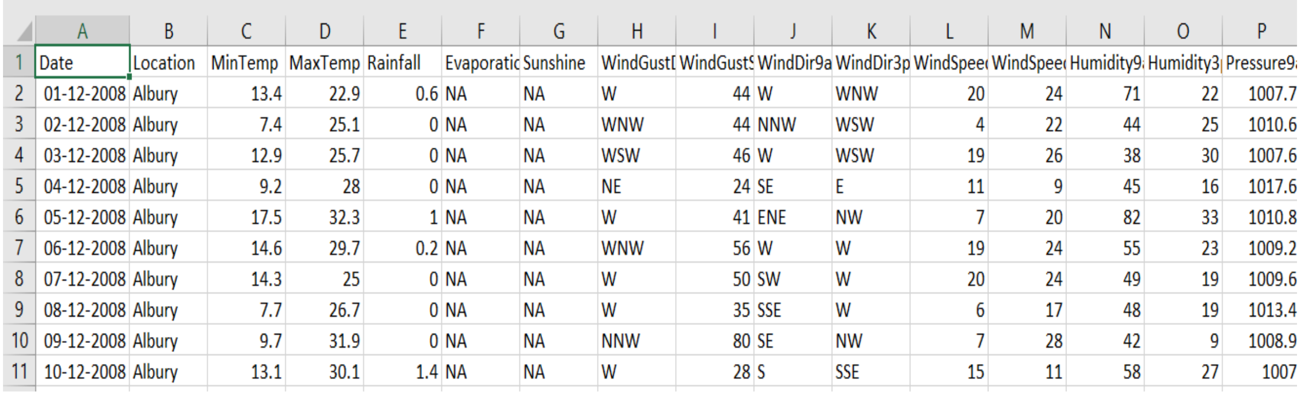
The mapreduce function defined earlier is called here, passing it the array of rows from the excel file in JSON format, the user defined map and reduce functions.
The result is a list of the average temperature of all the months. It is passed as argument to the function(), which finds the hottest, coldest months of summer and winter, respectively.

Queries used to generate results

4 main queries were executed, and the time it took for the program to run them, was noted down. The queries were,

- The hottest month of all the cities in the dataset
- The coldest month of all the cities in the dataset
- The hottest winter month of all the cities
- The coldest summer month of all the cities

Format of Dataset used (for single node)

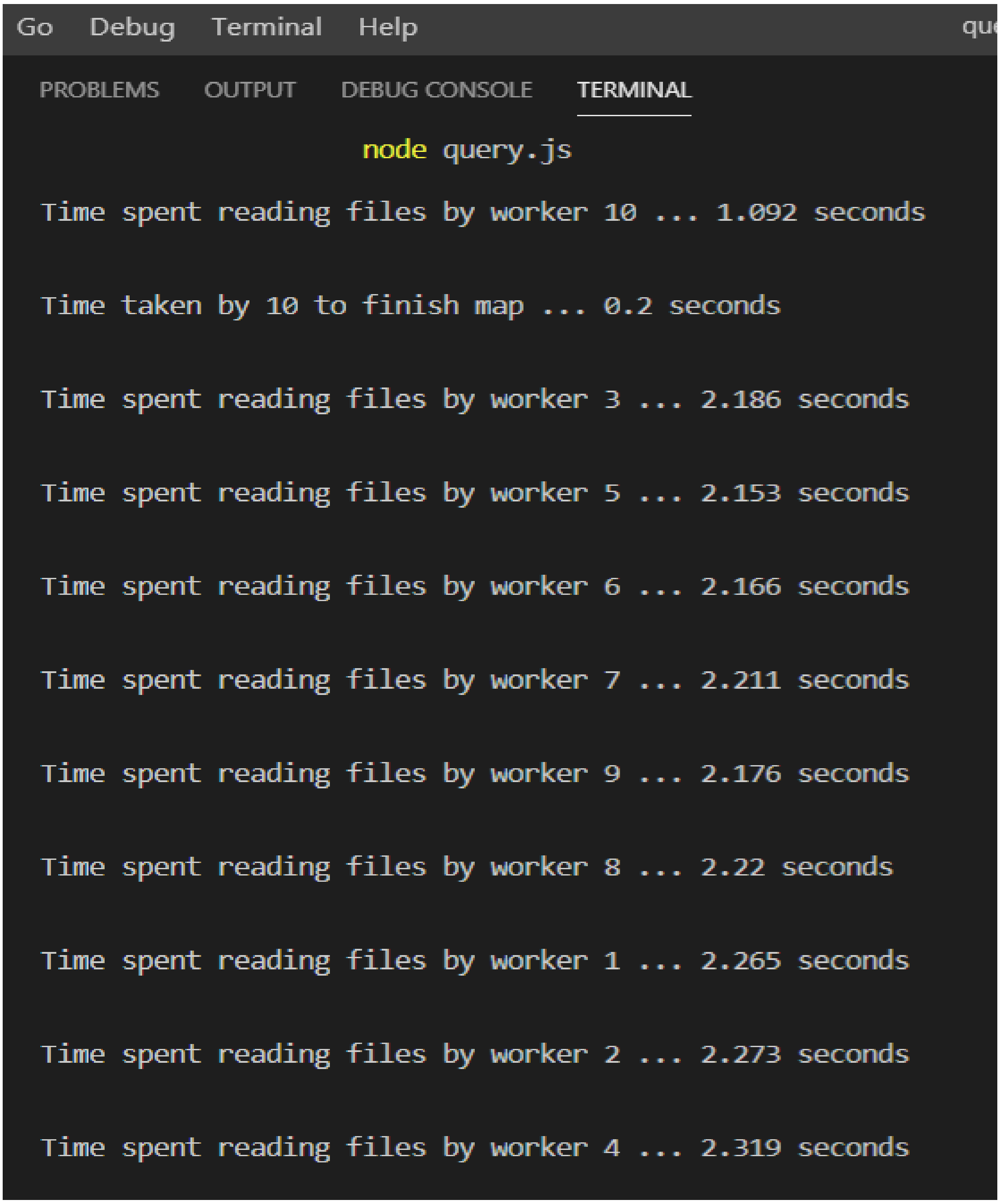


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporatic	Sunshine	WindGustD	WindGustS	WindDir9a	WindDir3p	WindSpee	WindSpee	Humidity9	Humidity3	Pressure9
2	01-12-2008	Albury	13.4	22.9	0.6	NA	NA	W	44	W	WNW	20	24	71	22	1007.7
3	02-12-2008	Albury	7.4	25.1	0	NA	NA	WNW	44	NNW	WSW	4	22	44	25	1010.6
4	03-12-2008	Albury	12.9	25.7	0	NA	NA	WSW	46	W	WSW	19	26	38	30	1007.6
5	04-12-2008	Albury	9.2	28	0	NA	NA	NE	24	SE	E	11	9	45	16	1017.6
6	05-12-2008	Albury	17.5	32.3	1	NA	NA	W	41	ENE	NW	7	20	82	33	1010.8
7	06-12-2008	Albury	14.6	29.7	0.2	NA	NA	WNW	56	W	W	19	24	55	23	1009.2
8	07-12-2008	Albury	14.3	25	0	NA	NA	W	50	SW	W	20	24	49	19	1009.6
9	08-12-2008	Albury	7.7	26.7	0	NA	NA	W	35	SSE	W	6	17	48	19	1013.4
10	09-12-2008	Albury	9.7	31.9	0	NA	NA	NNW	80	SE	NW	7	28	42	9	1008.9
11	10-12-2008	Albury	13.1	30.1	1.4	NA	NA	W	28	S	SSE	15	11	58	27	1007

This is a weather dataset, we accessed from Kaggle (source- dataset)
The columns present in the dataset were,

- Date (dd-mm-yyyy format)
- Location (all Australian cities)
- Minimum Temperature
- Maximum Temperature
- Rainfall
- Evaporation
- Sunshine
- Wind Gust Direction
- Wind Gust Speed
- Wind direction 9 am
- Wind direction 3 pm
- Wind Speed 9 am
- Wind Speed 3 pm
- Humidity 9 am
- Humidity 3 pm
- Pressure 9 am
- Pressure 3 pm
- Cloud 9 am
- Cloud 3 pm
- Temperature 9 am
- Temperature 3 pm
- Rain Today
- Risk_MM (risk variable)
- Rain Tomorrow

Results

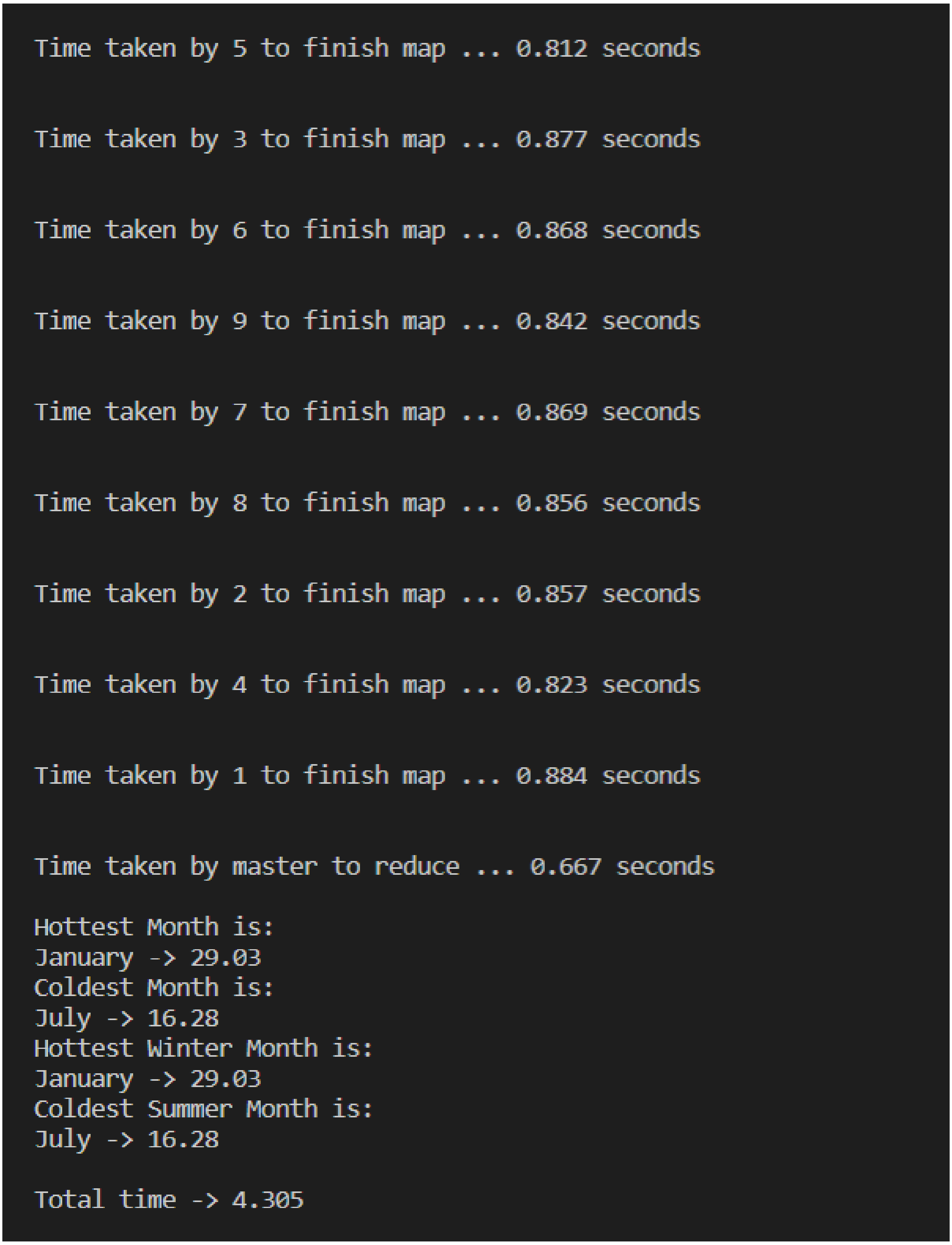


```
node query.js

Time spent reading files by worker 10 ... 1.092 seconds

Time taken by 10 to finish map ... 0.2 seconds

Time spent reading files by worker 3 ... 2.186 seconds

Time spent reading files by worker 5 ... 2.153 seconds

Time spent reading files by worker 6 ... 2.166 seconds

Time spent reading files by worker 7 ... 2.211 seconds

Time spent reading files by worker 9 ... 2.176 seconds

Time spent reading files by worker 8 ... 2.22 seconds

Time spent reading files by worker 1 ... 2.265 seconds

Time spent reading files by worker 2 ... 2.273 seconds

Time spent reading files by worker 4 ... 2.319 seconds
```

The average time spent reading input from the Excel File by each Child Node instance => 2.196 seconds

Except for Instance 10, no other child node is able to finish its map operation as fast, indicating the race condition that is formed due to the child nodes competing for core processor resources.

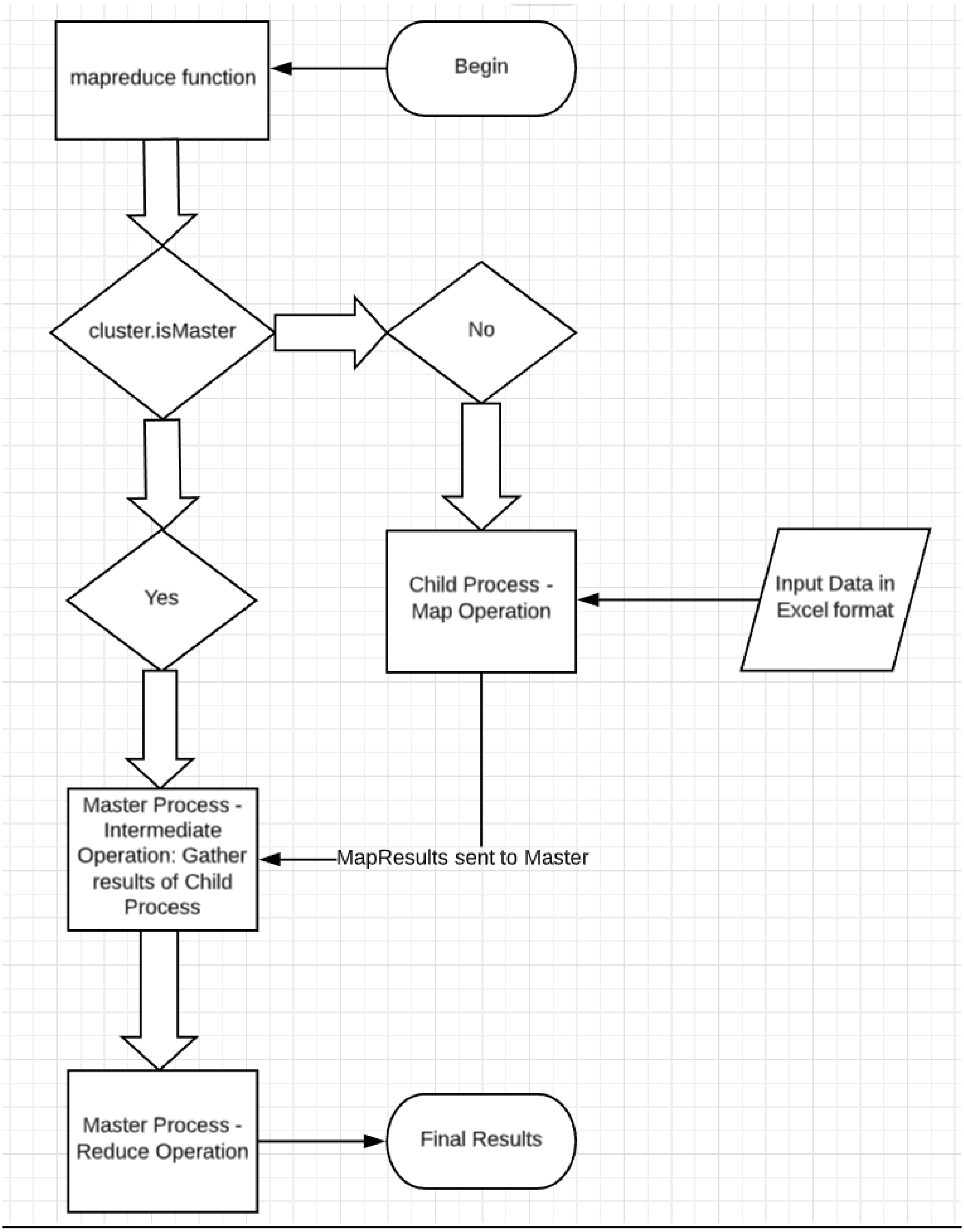
The time taken by the Child Node 10 is also the least among all the instances, only 0.2 seconds.



```
Time taken by 5 to finish map ... 0.812 seconds


Time taken by 3 to finish map ... 0.877 seconds


Time taken by 6 to finish map ... 0.868 seconds


Time taken by 9 to finish map ... 0.842 seconds


Time taken by 7 to finish map ... 0.869 seconds


Time taken by 8 to finish map ... 0.856 seconds


Time taken by 2 to finish map ... 0.857 seconds


Time taken by 4 to finish map ... 0.823 seconds


Time taken by 1 to finish map ... 0.884 seconds


Time taken by master to reduce ... 0.667 seconds

Hottest Month is:
January -> 29.03
Coldest Month is:
July -> 16.28
Hottest Winter Month is:
January -> 29.03
Coldest Summer Month is:
July -> 16.28

Total time -> 4.305
```

Average time spent in Map operation by the Child processes > 0.867 s Time taken for the Master instance to complete the reduce operation, 0.667 seconds, is much lesser than the average time taken by the Child instances to finish their map code logic execution. Total time taken for NodeJS execution of MapReduce > 4.305 s

UML Flow Chart


Begin
mapreduce function
cluster.isMaster
No
Yes
Child Process - Map Operation
Input Data in Excel format
Master Process - Intermediate Operation: Gather results of Child Process
MapResults sent to Master
Master Process - Reduce Operation
Final Results

6. Multi Node Implementation Code

Details of the Experiment

1. Total Single Board Computer (SBC) count: 16
    a. Model 4b: 1
    b. Model 3b+: 13
    c. Model 3b: 2
2. Data Size:
    a. 5.07 MB
    b. 10.18 MB
3. **Sample Query Given (random generated health dataset)**
    a. Find all information of patients (demographic info, food habit etc.) with name "xyz"
    b. Find all information of patients with a certain comorbid condition (eg. high blood pressure, high cholesterol)

Experiment Node structure

1. Master Node: Raspberry pi model 4b (Qty: 1)
2. Client Nodes (Total Qty: 15)
    a. Raspberry Pi 3b+ (Qty: 13)
    b. Raspberry Pi 3b (Qty: 2)

Experiment Steps (configurations used to run the program)

1. Nodes are connected with wireless infrastructure based network. All nodes communicated through a 2.4 Ghz 802.11n Wi-Fi router.(Star topology)
    a. Run **Hadoop**, Choosing the raspberry pi 4b as master node (Highest RAM - 4GB and Highest no. of core processors - 4) and the other nodes as client nodes.
    b. Run **Node JS mapreduce**, Choosing the raspberry pi 4b as master node and the other nodes as client node (same configuration).

2. Nodes are connected with wireless infrastructure less network (BATMAN-ADV). All Nodes communicated with each other by BATMAN-ADV ADHOC network (2.4 Ghz /802.11 n/ channel 1). (Mesh topology) – The same order is followed, first Hadoop is run, then the NodeJS mapreduce.

3. All nodes are connected with wireless infrastructure less network (BATMAN-ADV). All Nodes communicated with each other by BATMAN-ADV ADHOC network (5.22 Ghz /802.11 ac/ channel 44). Similar order of program execution (Hadoop, NodeJS MapReduce).

Program Code

File Transfer (Client)

```
const app = express()

app.use(fileUpload());


app.post('/upload', function(req, res) {
    console.log("Start UPLOAD");
    var startTime = getCurrentTime('start');

    var  file_name = req.files.file_name;

    console.log(req.body);

    console.log( file_name.name +" :: "+startTime[0]);

    file_name.mv('blocks/'+ file_name.name, function(err) {
    if (err)
        {
            console.log("error in file uploading");
      return res.status(500).send(err);
        }
        var content = [file_name.name];
        content = JSON.stringify(content);

        fs.appendFile("client_meta/"+req.body.client_meta, content, function (
err) {
        });
        var endTime = getCurrentTime('end');

        var execTime = (endTime[1]- startTime[1])/1000;

        console.log(file_name.name + " ::"+endTime[0]);

        console.log("Time taken: "+execTime);

        console.log("====================================");
    return res.status(200).send('File uploaded!');
  });
});

app.post('/rfupload', function(req, res) {
    console.log("Start RF UPLOAD");
    var startTime = getCurrentTime('start');

    var  file_name = req.files.file_name;

    console.log(req.body);

    console.log( file_name.name +" :: "+startTime[0]);

    file_name.mv('blocks/'+ file_name.name, function(err) {
    if (err)
        {
            console.log("error in file uploading");
      return res.status(500).send(err);
        }
        var content = [file_name.name];
        content = JSON.stringify(content);

        fs.appendFile("client_meta/"+req.body.client_meta, content, function (
err) {
        });
        var endTime = getCurrentTime('end');

        var execTime = (endTime[1]- startTime[1])/1000;

        console.log(file_name.name + " ::"+endTime[0]);

        console.log("Time taken: "+execTime);

        console.log("====================================");
    return res.status(200).send('File uploaded!');
  });
});


app.listen(3002, () => console.log('Example app listening on port 3002!'))

function getCurrentTime(type){
var currentdate = new Date();
var datetime = type+" : " + currentdate.getDate() + "/"
                + (currentdate.getMonth()+1)  + "/"
                + currentdate.getFullYear() + " @ "
                + currentdate.getHours() + ":"
                + currentdate.getMinutes() + ":"
                + currentdate.getSeconds();
var inMiliSec = currentdate.getTime();
```

```
return [datetime, inMiliSec];

}
```

The Express Server library package has been used primarily for communication between the server and client nodes. First, the express application listening at port 3002 is launched. The post request method is used for transmitting the input file contents. The file name is fetched from the request (req) object, and stored in the file_name variable. Using the fileupload function in the Express library, the .mv() function of the “filename” object (file_name) is invoked, the data of the file (in string format) is converted into JSON objects, using the stringify method, and then attached to the response (res) object. The time taken to upload the file contents is measured parallelly, and is initiated as soon as the post method starts executing. There are two instances of the post method here, one is the “upload” path handler, and another is the “rfupload” path handler. The “upload” path is used for normal input file transmission, and the “rfupload” path is used for retransmission of input files already transmitted. The retransmission is done, for faster availability of data in the client nodes, during the map reduce query execution.

Client_Response.js

```
app.use('/assets', express.static('assets'));
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');

app.get('/', function (req, res) {
   var ipAddr = ip.address()
   res.render( __dirname + "/" + "index.html",{ip:ipAddr} );
   console.log("opening index file at: "+ipAddr);

})


//client machine
app.get('/node_response_status', function (req, res){
    console.log("request received ...");
    var response = {
        ipAddr : ip.address(),
        status : 1
    }
    res.send(response);
})
```

```
//
var server = app.listen(3001, function () {
   var host = server.address()
   var port = server.address().port
   console.log(ip.address());
   console.log("Client1 starts on port no 3001");

})
```

This program only checks if the client node is responding or not. Using the get method, from the “node_response_status” path, the ip address of the specific node to be verified for response, is stored in the get method response body. A separate variable called status is added to the response object, and it is made to store 1, to indicate the client node is up. The root path “/” is separately used to render the index html file. The render() function of the response (res) object is separately used for this, also through a get method body. This index html file records all the interaction between the client and server nodejs programs. So, only when a particular client is found to be responding to the node_response requests, is the index file launched for writing the query results.

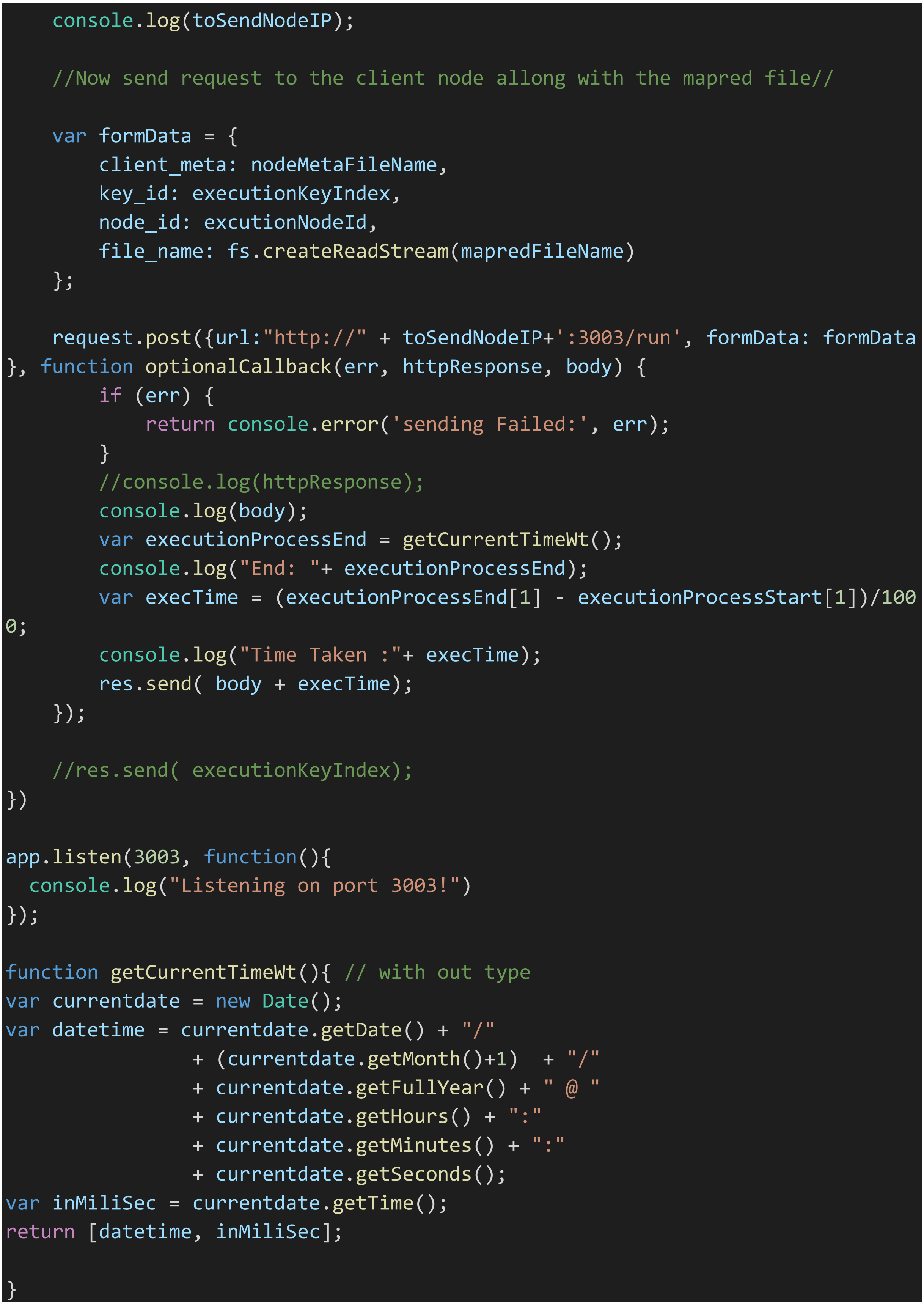
Server_Response.js

```
app.use('/public', express.static('public'));
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');

app.get('/', function (req, res) {
   var ipAddr = ip.address()
   res.render( __dirname + "/public/" + "index.html",{ip:ipAddr} );
   console.log("opening index file at: "+ipAddr);
})
app.get('/get_response_from_client', function (req, res){
    //console.log("call for client response");
    //console.log(req.query);
    var temp = req.query;
    var host= temp.host;
    var ip = temp.ip;
    const url = "http://"+ip +":3001/node_response_status";
    request.get(url, (error, response, body) => {
        if(!error)
        {
            let json = JSON.parse(body);
            json.ipAddr= ip;
            json.host =host;
            //console.log(json);
            res.send(json);
        }
        else
        {
            error.status = 0;
            error.ipAddr=ip;
            error.host =host;
            //console.log(error);
            res.send(error);
        }
    });
})
var server = app.listen(3001, function () {
   var host = server.address()
   var port = server.address().port
   console.log(ip.address());
   console.log("Master starts on port no 3001");
})
```

This program is primarily responsible for interacting with the client side response.js program. It receives the response from a client, and then transmits the ipaddress and host name through a json object, stored in the response (res) object, to whichever other client / server program that requested the info. The index.html is also rendered separately here, via the root path “/” get request method.

MapReduce_Server.js

```
const app = express();

app.use('/public', express.static('public'));
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');



app.get('/', function (req, res) {
    var executionProcessStart = getCurrentTimeWt();
    console.log("Start: "+ executionProcessStart);
//URL SHOULD BE : http://localhost:3003/?key=kolkata&file=50.txt&mapred_file=C
ovidCount.js

    var executionKey = req.query.key;
    var executionKeyIndex;

    var excutionNodeId;
    var nodeMetaFileName;

    var mapredFileName = req.query.mapred_file;

    var nodesDetails = [];
    var fileName = req.query.file;
    var file = fileName.split(".");
    file= file[0];
    //console.log(file);
    var masterFileName = "master_meta/master_meta_"+file+".json";
    var rawMetaData = fs.readFileSync(masterFileName);
    var metaData = JSON.parse(rawMetaData);
    //console.log(metaData);
    for(var key in metaData['keys'])
    {
        if(metaData['keys'][key].toLowerCase() == executionKey){
            executionKeyIndex = key;
        }
    }
    for(var key in metaData['distribution'])
    {
        if(metaData['distribution'][key]['key_id'] == executionKeyIndex)
        {
            excutionNodeId = metaData['distribution'][key]['node_id'];
            nodeMetaFileName = metaData['distribution'][key]['client_meta'];
        }
    }
    console.log(executionKeyIndex);
    console.log(excutionNodeId);
    console.log(nodeMetaFileName);
    // read nodes.txt file and make an array of nodes ip and host name
    var nodesDetailsTemp = fs.readFileSync('public/nodes.txt').toString().spli
t("\n");
    for(var line in nodesDetailsTemp)
    {
        if(nodesDetailsTemp[line])
        {
            var temp = nodesDetailsTemp[line].split(" ");
            var temp1 ={};
            temp1['ip'] = temp[0];
            temp1['host'] = temp[1];

            nodesDetails.push(temp1);
        }
    }
    //console.log(nodesDetails);

    var toSendNodeIP = nodesDetails[excutionNodeId-1]['ip'];
```

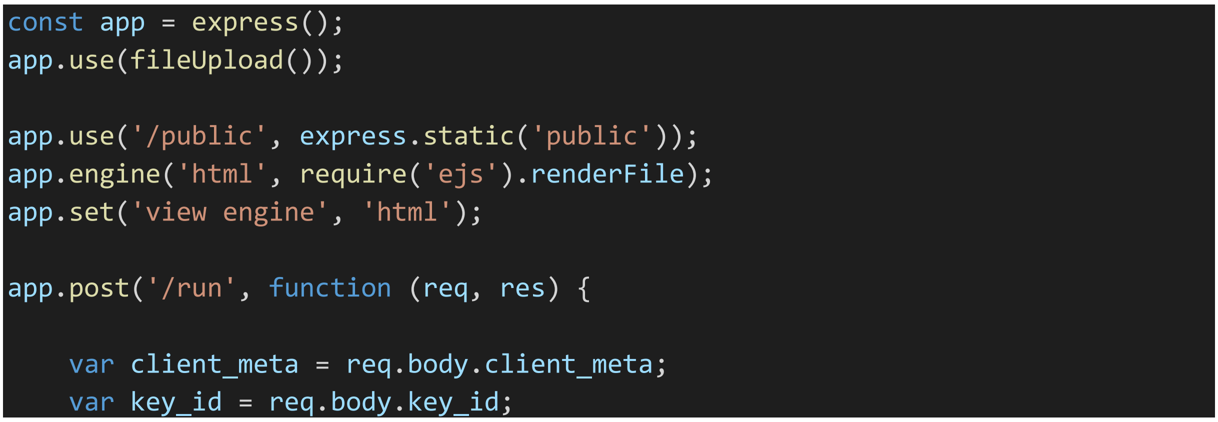


```
    console.log(toSendNodeIP);

    //Now send request to the client node allong with the mapred file//

    var formData = {
        client_meta: nodeMetaFileName,
        key_id: executionKeyIndex,
        node_id: excutionNodeId,
        file_name: fs.createReadStream(mapredFileName)
    };

    request.post({url:"http://" + toSendNodeIP+':3003/run', formData: formData
}, function optionalCallback(err, httpResponse, body) {
        if (err) {
            return console.error('sending Failed:', err);
        }
        //console.log(httpResponse);
        console.log(body);
        var executionProcessEnd = getCurrentTimeWt();
        console.log("End: "+ executionProcessEnd);
        var execTime = (executionProcessEnd[1] - executionProcessStart[1])/100
0;
        console.log("Time Taken :"+ execTime);
        res.send( body + execTime);
    });

    //res.send( executionKeyIndex);
})

app.listen(3003, function(){
  console.log("Listening on port 3003!")
});

function getCurrentTimeWt(){ // with out type
var currentdate = new Date();
var datetime = currentdate.getDate() + "/"
                + (currentdate.getMonth()+1)  + "/"
                + currentdate.getFullYear() + " @ "
                + currentdate.getHours() + ":"
                + currentdate.getMinutes() + ":"
                + currentdate.getSeconds();
var inMiliSec = currentdate.getTime();
return [datetime, inMiliSec];

}
```

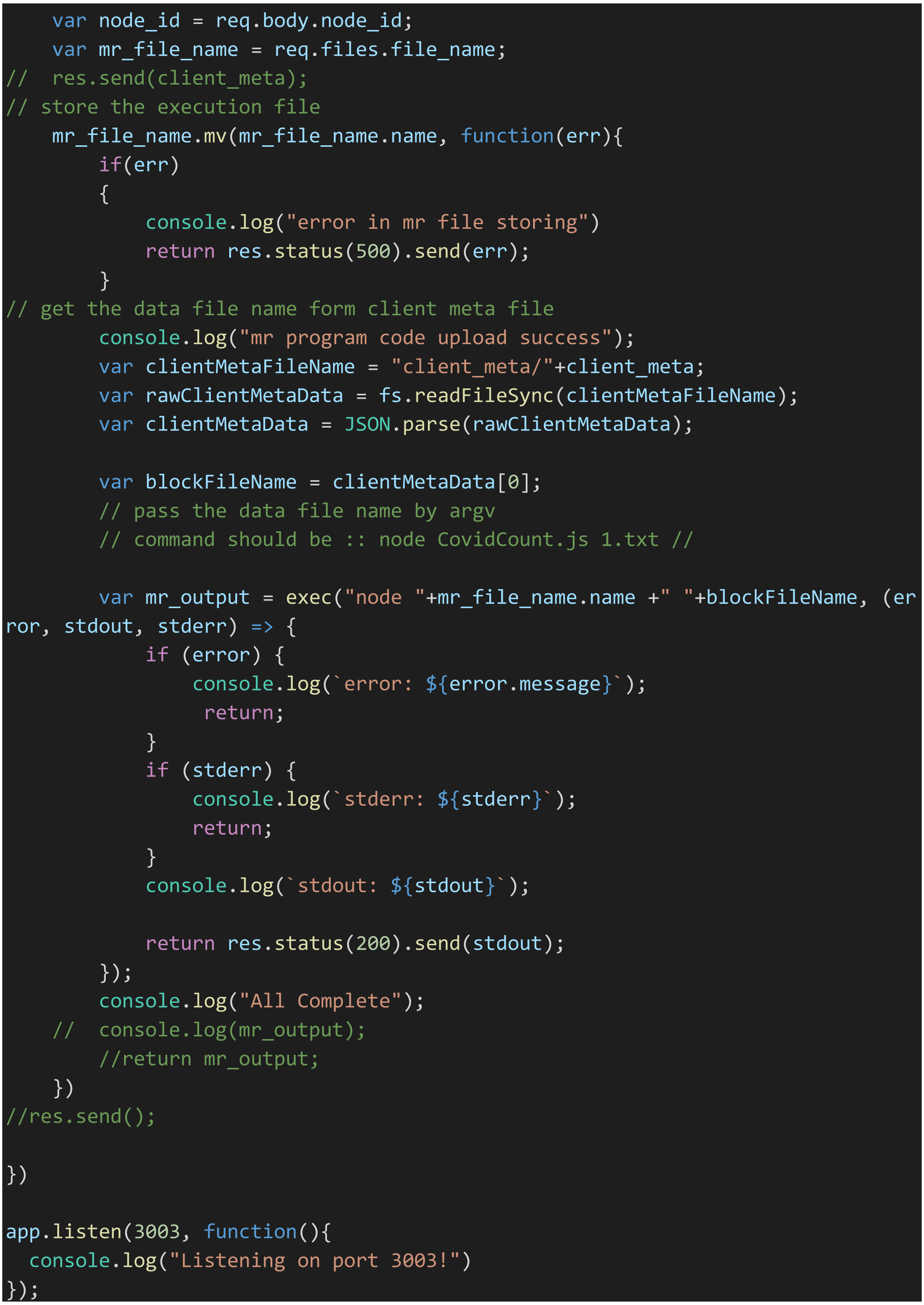
This is the map reduce main program implementation on the server node. It works in the following steps :-

- It initially reads the master node meta data file. It stores the relevant keys and the nodes in which they have been distributed, after the map process completion by the client nodes.
- All the keys are matched with the search execution key, received in the request object (req), if any one of them matches, it is set as the index for the input search key.
- Next, the distribution list is searched, for the node in which the key has been stored. This is kept recorded in the distribution meta data file. When the distribution index file's right index matches with the search execution key's index, recorded in the previous loop, the node's meta data file name is noted down (which is stored in the master node index distribution file).
- The ip address and host id of the client nodes, are then read from the "nodes.txt" file, and stored in as a key value JSON object in the nodedetails variable.
- The post request is finally sent to the particular execution node, depending on the 'ip' key. The execution key and distribution index, gathered from the previous iterations, are inserted into a JSON object, and then sent as a request parameter to the appropriate client node.
- The 4 main parameters being sent through the JSON object in the request object are, the metadata file on the client node, the key required for the execution of the query, the node ID, and the map reduce intermediate results file.
- The response of the post request made is timed simultaneously, for performance measurement.

MapReduce_Client.js



```
const app = express();
app.use(fileUpload());

app.use('/public', express.static('public'));
app.engine('html', require('ejs').renderFile);
app.set('view engine', 'html');

app.post('/run', function (req, res) {

    var client_meta = req.body.client_meta;
    var key_id = req.body.key_id;
```

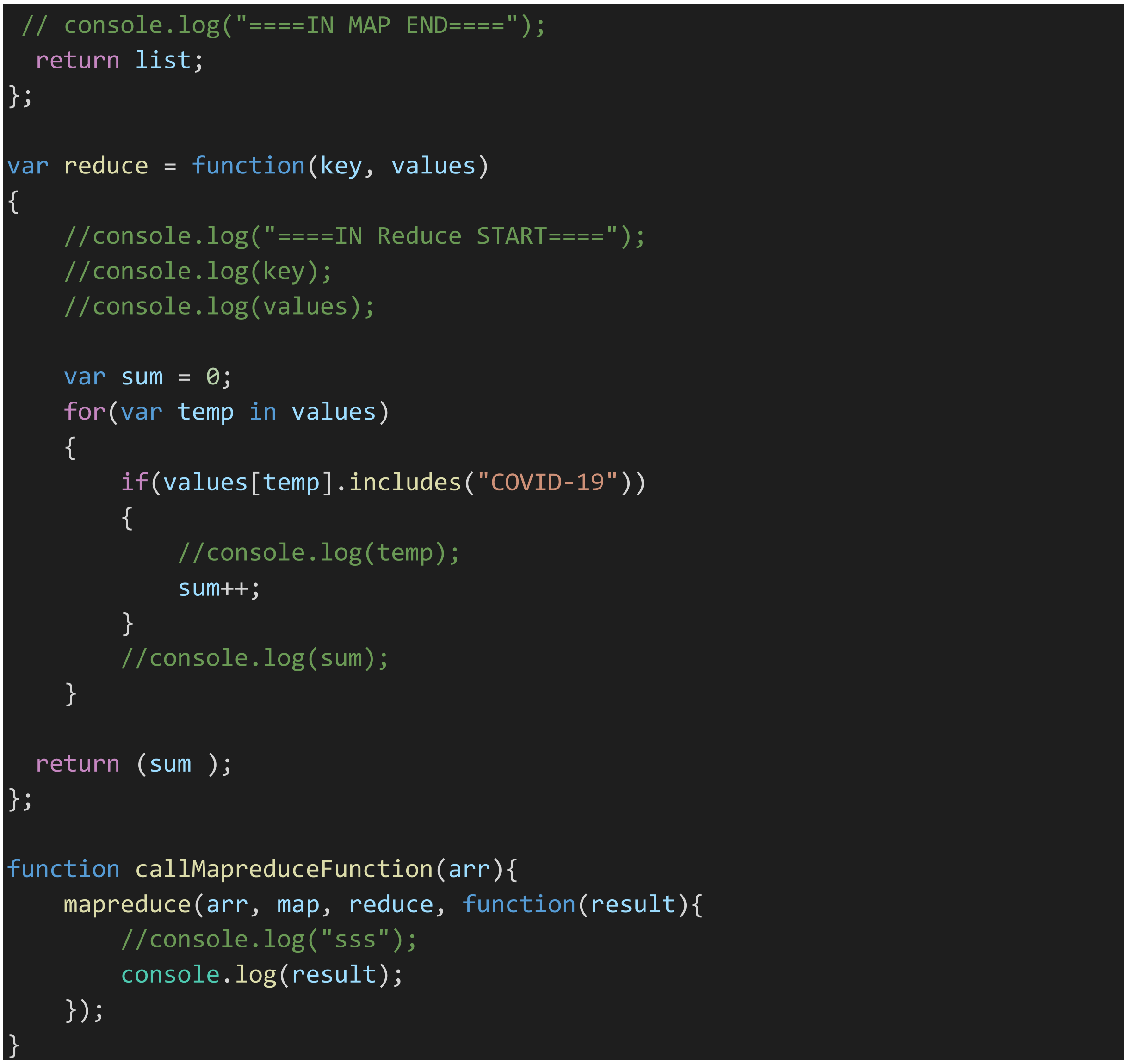


```
    var node_id = req.body.node_id;
    var mr_file_name = req.files.file_name;
//  res.send(client_meta);
// store the execution file
    mr_file_name.mv(mr_file_name.name, function(err){
        if(err)
        {
            console.log("error in mr file storing")
            return res.status(500).send(err);
        }
// get the data file name form client meta file
        console.log("mr program code upload success");
        var clientMetaFileName = "client_meta/"+client_meta;
        var rawClientMetaData = fs.readFileSync(clientMetaFileName);
        var clientMetaData = JSON.parse(rawClientMetaData);

        var blockFileName = clientMetaData[0];
        // pass the data file name by argv
        // command should be :: node CovidCount.js 1.txt //

        var mr_output = exec("node "+mr_file_name.name +" "+blockFileName, (error, stdout, stderr) => {
            if (error) {
                console.log(`error: ${error.message}`);
                 return;
            }
            if (stderr) {
                console.log(`stderr: ${stderr}`);
                return;
            }
            console.log(`stdout: ${stdout}`);

            return res.status(200).send(stdout);
        });
        console.log("All Complete");
    //  console.log(mr_output);
        //return mr_output;
    })
//res.send();

})

app.listen(3003, function(){
  console.log("Listening on port 3003!")
});
```

The mapreduce program on the client node, receives the post request from the map reduce program on the server node, and unpacks all the relevant information from the body of the request object. The file contents of the metadata file on the client node, is then changed, to add the response of the get request, sent to the actual mapreduce code, within the client node. All the request parameters, along with the right block of data, determined from the client metadata file, are sent to the query.js program, and the results are appended to the mapreduce file (mr_filename).

Query.js

```
var mapreduce = require('mapred')();
var fs = require('fs');
var es = require('event-stream');

var arr = [];
var lineNr = 0;


var dataFileName = process.argv[2];
var s = fs.createReadStream("blocks/" + dataFileName)
.pipe(es.split())
.pipe(es.mapSync(function(line){
        s.pause();
        lineNr += 1; // current line no.
        if(lineNr > 1)
        {
            var lineArr = line.split('\t');
            var temp =[lineArr[1], lineArr[9]];
            arr.push(temp);
        }

        s.resume();
    })
    .on('error', function(err){
        console.log('Error while reading the main file.', err);
    })
    .on('end', function(){
        callMapreduceFunction(arr);
    })
);

var map = function(key, value)
{
    //console.log("====IN MAP START====");
    //console.log(key);
    //console.log(value);
    var list = [];
    list.push([key, value]);
```

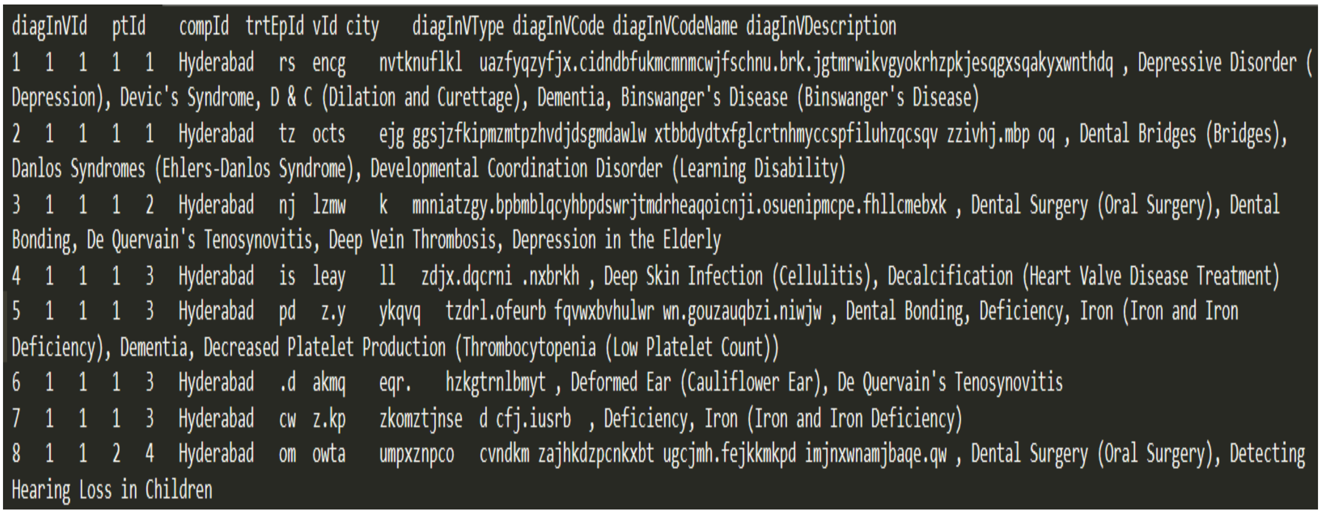


```
 // console.log("====IN MAP END====");
  return list;
};

var reduce = function(key, values)
{
    //console.log("====IN Reduce START====");
    //console.log(key);
    //console.log(values);

    var sum = 0;
    for(var temp in values)
    {
        if(values[temp].includes("COVID-19"))
        {
            //console.log(temp);
            sum++;
        }
        //console.log(sum);
    }

  return (sum );
};

function callMapreduceFunction(arr){
    mapreduce(arr, map, reduce, function(result){
        //console.log("sss");
        console.log(result);
    });
}
```

This part of the multi node implementation is similar to the single node map reduce program. The main logic of the code has been condensed into the mapred js library, and only the map and reduce functions are present here, so that the user dependent logic can be changed around, according to the query.

Before the mapreducefunction is launched, the file reader stream is used to create a stream of the data from “dataFilename” file in the “/blocks/” directory. The filename is fetched as an argument, from the request object, sent by the client mapreduce program. The reader file stream is created using the readerFileStream method, and sending the stream through the pipe() method, to control and stop / start the stream as required. Each line is taken up first, split based on the ‘/t’ symbol, the individual words, pushed into the array, and the stream resumed.

<u>Format of Dataset used (for multinode)</u>



```
diagInVId   ptId    compId  trtEpId vId city    diagInVType diagInVCode diagInVCodeName diagInVDescription
1   1   1   1   1   Hyderabad   rs  encg    nvtknuflkl  uazfyqzyfjx.cidndbfukmcmnmcwjfschnu.brk.jgtmrwikvgyokrhzpkjesqgxsqakyxwnthdq , Depressive Disorder (
Depression), Devic's Syndrome, D & C (Dilation and Curettage), Dementia, Binswanger's Disease (Binswanger's Disease)
2   1   1   1   1   Hyderabad   tz  octs    ejg ggsjzfkipmzmtpzhvdjdsgmdawlw xtbbdydtxfglcrtnhmyccspfiluhzqcsqv zzivhj.mbp oq , Dental Bridges (Bridges),
Danlos Syndromes (Ehlers-Danlos Syndrome), Developmental Coordination Disorder (Learning Disability)
3   1   1   1   2   Hyderabad   nj  lzmw    k   mnniatzgy.bpbmblqcyhbpdswrjtmdrheaqoicnji.osuenipmcpe.fhllcmebxk , Dental Surgery (Oral Surgery), Dental
Bonding, De Quervain's Tenosynovitis, Deep Vein Thrombosis, Depression in the Elderly
4   1   1   1   3   Hyderabad   is  leay    ll   zdjx.dqcrni .nxbrkh , Deep Skin Infection (Cellulitis), Decalcification (Heart Valve Disease Treatment)
5   1   1   1   3   Hyderabad   pd   z.y    ykqvq   tzdrl.ofeurb fqvwxbvhulwr wn.gouzauqbzi.niwjw , Dental Bonding, Deficiency, Iron (Iron and Iron
Deficiency), Dementia, Decreased Platelet Production (Thrombocytopenia (Low Platelet Count))
6   1   1   1   3   Hyderabad   .d  akmq    eqr.    hzkgtrnlbmyt , Deformed Ear (Cauliflower Ear), De Quervain's Tenosynovitis
7   1   1   1   3   Hyderabad   cw  z.kp    zkomztjnse  d cfj.iusrb  , Deficiency, Iron (Iron and Iron Deficiency)
8   1   1   2   4   Hyderabad   om  owta    umpxznpco   cvndkm zajhkdzpcnkxbt ugcjmh.fejkkmkpd imjnxwnamjbaqe.qw , Dental Surgery (Oral Surgery), Detecting
Hearing Loss in Children
```

This was a health dataset. Most of the data used here, was generated randomly (except for city and diagInVDescription). The columns present here are as follows :-

- diagInVId (Diagnosis Virtual Id)
- ptId (Patient Id)
- compId (Computer Id)
- trrtEpId (Treatment Electronic Portal Id)
- vId (Virtual Id)
- city
- diagInVType (Diagnosis Virtual Type)
- diagInVCode (Diagnosis Virtual Code)
- diagInVCodeName (Diagnosis VirtualCode Name)
- diagInVDescription (Diagnosis Virtual Description)

The final column (diagInVDescription) is the one primarily used for result generation for the queries. It is a list of all the conditions, the patient is diagnosed with.
So for example, for the query, Find the distribution city wise, of patients suffering from xyz disease, the diagInVDescription list will be divided into an array, and then all patients and their cities will be grouped, according to the xyz disease.

Results

1. All node connected with **Wi-Fi Router**
   a. Query execution time in **Hadoop cluster** (**Time in Sec**)

File Size ode Count	~~5MB	~~10mb
1	40.02	41.21
2	45.00	43.09
3	46.32	44.39
4	59.57	49.23
5	63.33	58.46
6	62.52	61.01
7	72.14	65.49
8	71.23	69.89
9	71.59	72.07
10	74.27	73.52
11	76.01	75.33
12	76.53	78.34
13	77.00	78.68
14	75.23	80.24
15	76.04	81.06
16	75.13	82.25
17	77.54	82.89

   b. Query execution time in **Node JS cluster** (**Time in minutes**)

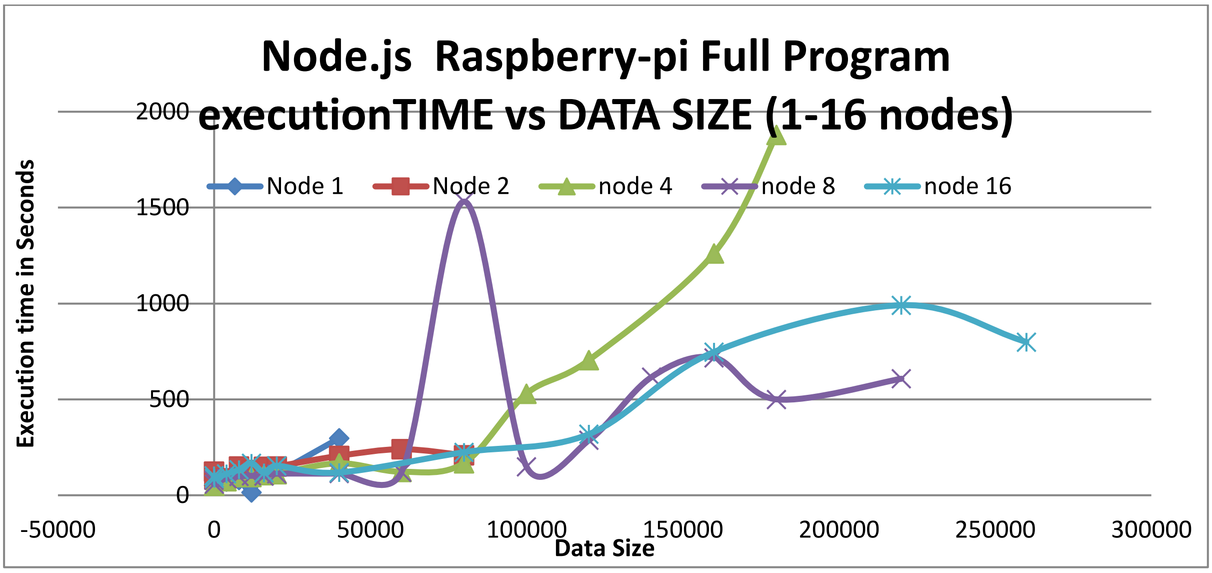
File Size ode Count	~~5MB	~~10mb
1	1.56	2.36
2	2.36	4.08
3	4.09	7.69
4	6.7	8.58
5	7.23	9.012
6	9.10	10.53
7	12.25	11.27
8	13.53	12.24
9	14.21	15.01
10	16.34	19.46
11	17.49	19.53
12	18.37	19.49
13	18.58	20.19
14	18.49	23.64
15	19.4	24.03
16	19.51	24.36
17	20.21	25.21

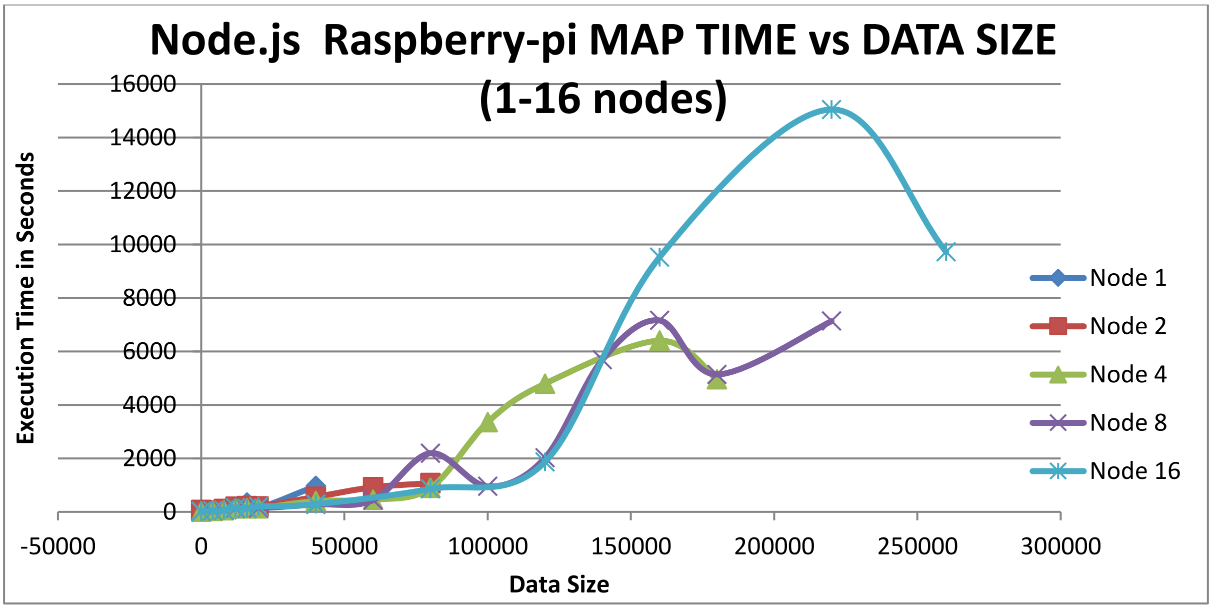
2. All node connected with **ADHOC NETWORK (BATMAN- ADV) 2.4 Ghz (802.11 n) channel 1**
    a. Query execution time in **Hadoop cluster** (**Time in Sec**)

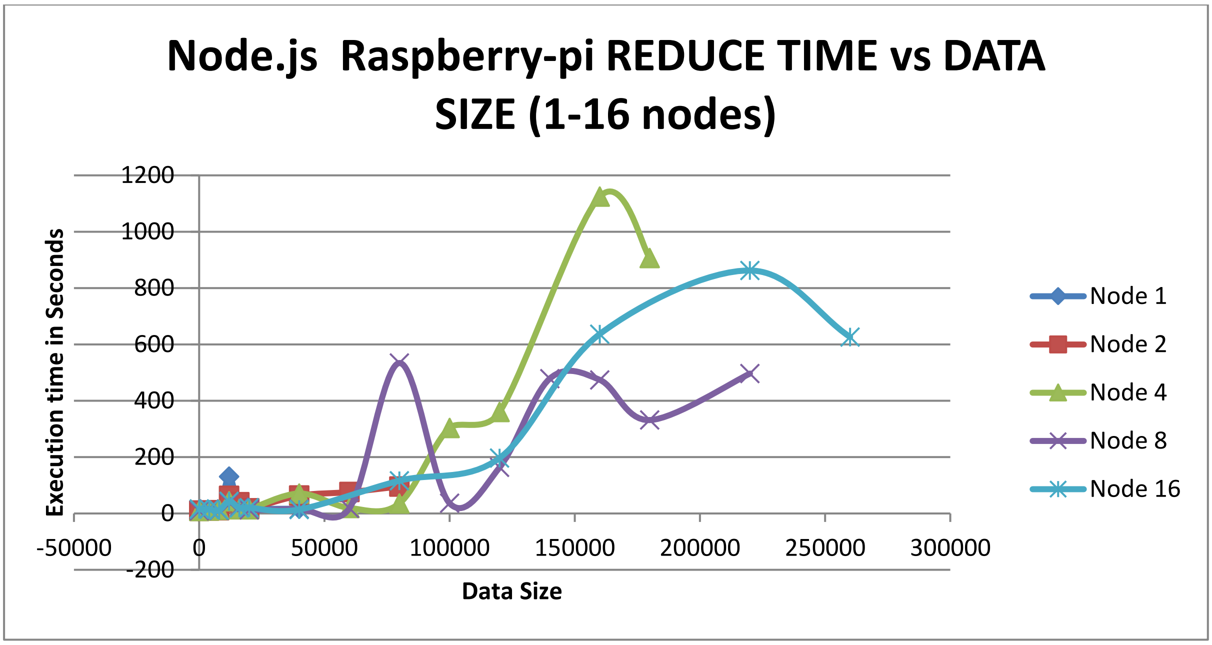
File Size ode Count	~~5MB	~~10mb
1	36.28	36.954
2	36.54	39.21
3	37.39	35.97
4	38.02	38.25
5	38.73	40.50
6	40.31	40.21
7	44.28	39.97
8	45.57	46.29
9	46.10	46.85
10	46.87	47.35
11	48.23	51.29
12	49.63	50.25
13	51.09	52.87
14	51.68	52.29
15	49.27	54.13
16	52.14	53.99
17	56.49	56.28

    b. Query execution time in **Node JS cluster** (**Time in minutes**)

File Size ode Count	~~5MB	~~10mb
1	0.58	1.28
2	2.49	3.25
3	4.26	4.25
4	4.37	8.21
5	6.53	9.47
6	6.59	12.52
7	7.43	11.20
8	9.56	13.12
9	9.57	13.43
10	10.24	13.26
11	10.49	14.27
12	11.00	16.29
13	13.45	16.21
14	13.58	19.26
15	15.24	18.51
16	16.21	18.20
17	16.87	18.29


Node.js Raspberry-pi Full Program executionTIME vs DATA SIZE (1-16 nodes)
Execution time in Seconds
2000
1500
1000
500
0
-50000
0
50000
100000
150000
200000
250000
300000
Data Size
Node 1
Node 2
node 4
node 8
node 16


Node.js Raspberry-pi MAP TIME vs DATA SIZE (1-16 nodes)
Execution Time in Seconds
16000
14000
12000
10000
8000
6000
4000
2000
0
-50000
0
50000
100000
150000
200000
250000
300000
Data Size
Node 1
Node 2
Node 4
Node 8
Node 16



7. Future scope:

This project leaves us with a future scope to further optimize the adhoc MapReduce architecture used for the data transfer over the network. Also there is a future scope to distribute the data analytically which can reduce down query resolution time.

8. Conclusion:

So, the initial goal of the project was to implement the MapReduce paradigm using the Node.js framework on a multi node infrastructure, and to validate the performance through a health dataset predictive analysis (search query execution). The performance of this multi node implementation has been satisfactory. For datasets of size 5-10 mb for each work node, the NodeJS implementation has given better results compared to the Hadoop program, under the same master-worker node set up.

9. References:

- Jeffrey Dean and Sanjay Ghemawat - MapReduce: Simplified Data Processing on Large Clusters
- Bingsheng He, Wenbin Fang, Naga K. Govindaraju - Mars: A MapReduce Framework on Graphics Processors
- Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler - The Hadoop Distributed File System
- Shankar Ganesh Manikandan, Siddarth Ravi -Big data analysis using Apache Hadoop
- In-Memory Data Grid White Paper GridGain Systems, 2013
- Himadri Sekhar Ray, Swastik Mukherjee, Nandini Mukherjee - Performance Enhancement in Big Data handling
- Rotsnarani Sethy, Mrutyunjaya Panda - Big Data Analysis using Hadoop: A Survey
- Urmila R. Pol - Big Data Analysis Using Hadoop Mapreduce
- Nimesh Chhetri - A Comparative Analysis of Node.js (Server-Side JavaScript)