Performance Analysis of Apache Hadoop and Spark on AIRS Cloud Using HiBench Benchmarks

Changwen LI
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
118010134@link.cuhk.edu.cn

Minghong XIA
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
118010340@link.cuhk.edu.cn

Hanyang XIA
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
118010339@link.cuhk.edu.cn

Ziqi GAO
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
118010077@link.cuhk.edu.cn

Dongyang WU
School of Data Science
The Chinese University of Hong Kong, Shenzhen
Shenzhen, China
118010324@link.cuhk.edu.cn

***Abstract*—In this paper, we evaluate the performance of AIRS Cloud using the HiBench benchmark suite. Those experiments are conducted on the AIRS Cloud virtual machine built on a physical machine with four core Xeon processors, 400GB disks, and 8GB RAM. Four cloud benchmarks were tested on this machine: Sort, Terasort, PageRank, and Analytical SQL Queries. HiBench benchmarking configuration is introduced in detail. Cloud benchmarking results are analyzed with the speed, throughput, speed up, productivity. We also compare the performance of Spark and Hadoop with sort benchmark and SQL benchmark.**

Keywords—MapReduce, Hadoop, Spark, Cloud

I. Introduction

Cloud computing is getting more and more popular due to the boom of distributed computing, internet technology, systems management and hardware. It is of great significance to do experiments on cloud platform such as AIRS Inspur cloud. Team 5 tried four benches on the cloud, which are Sort, Terasort, Pagerank and SQL. The report content is divided into 3 parts. The first part introduces testing result of SQL and the corresponding analysis. The second part describes the experiments on Page rank. The third part explains some experiment results and findings of Sort and Terasort.

II. Hive SQL and Spark SQL

A. *Environment Setting and Introduction*

Before executing the benchmark, we need to set the authentication for HIVE SQL. We should modify the */home/team5/HibBench3/bin/functions/workload_functions.sh* file. Add *set mapreduce.job.queuename=team5;* to all the scan, aggregation, and join functions. Also, if we want to run Spark, we should also change the USE DEFAULT; to USE team5;. Please look through the appendix to check this modified shell file.

The data used to test: All the data is automatically generated Web data with hyp7erlinks following the Zipfian distribution[1] (the frequency of any word is inversely proportional to its rank in the frequency table). The data generator of HiveBench takes four parameters as inputs: Numbers of maps, number of reducers, pages, and uservisits. The last two parameters will set the number of rows for the two test tables: **Rankings** and **UserVisits**[2].

About the number of mappers and reducers: When we conduct the micro benchmark like wordcount or sort, we can just modify the *conf/hibench.conf* by changing the number of hibench.default.map.parallelism, and hibench,default.shuffle.parallelism. Those two values of the variables will only affect how we generate and load the SQL data. However, those two variables cannot affect the mapper and reducer of the Hive SQL, and the executor of Spark SQL. The number of executors of Spark SQL can be changed in *spark.conf* easily. The following content will introduce how to modify the number of mappers and reducers:

- Reducer:
  In default, every reducer's job is to deal with 1GB data. If the mappers 'output is less than 1GB, then we will have only one reduce job. For example, if the table has 2.4GB data, the number of reducers will be 3. We can also change the number of reducers manually by modifying the workload_functions.sh mentioned at the beginning of this part.
- Mapper:
  The number of mappers depends on the data block's size. Entering the Hive shell, we can get the block's size is 134217728 (128MB). If the data size of the file is 2.4GB, then we will have $\frac{2400}{128} \approx 20$mappers. If we want to change the number of mappers, we can set the MapReduce's split size to adjust the number of mappers.

Please check the appendix if you want to know how to adjust the number of mappers and reducers of Hive SQL.

B. *Benchmark Results – Data Generation*

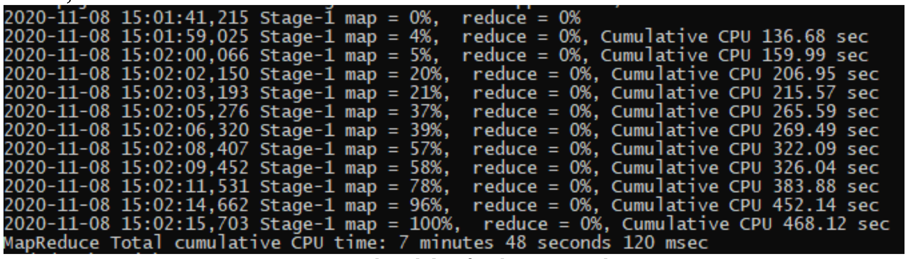
Since Join analytical query is the most completed one among the three benchmarks. Our experiment mainly focuses on the Join instruction. All data are generated and pre-loaded into Hadoop Distributed File System (HDFS) before running the benchmarks by running the *prepare.sh*. We do not consider loading time as a part of the benchmark results. Here is only a table showing the generation and load times of different size's data used in **Join** instruction using eight mappers and four reducers. The time data is got from the *bench.log* of the prepare shell script[shell] using the sum of the two jobs' slots times. The time spent on the task can be calculated by $\frac{Total\ Time\ spent\ by\ all\ maps}{Number\ of\ Mappes}$ or $\frac{Total\ Time\ spent\ by\ all\ reduces}{Number\ of\ Reducers}$. And the data size can be checked in this way recorded in the appendix. From the table, we can see that the time spent on the larger data size's map tasks are faster. This is because the number of mappers is too large for tiny size data. It is time consuming to assign the task. Except the tiny data size, the map time and reduce time is proportional to the data size.

TABLE 1 DATA SIZE AND TIME CONSUMED

Data Size	PAGES	USERVISITS	Maptime(s)	Reducetime(s)
7.3Mb	120	1000	33	8
538Mb	12,000	10,000	24	8
5.2GB	120,000	1,000,000	30	20
51.8GB	1,200,000	10,000,000	78	151
155.5GB	3,600,000	30,000,000	201	255

C. *Benchmark Results – Scan Task*

According to the MapReduce's implementation, scan task does not need reducer to finish the job, but only need mappers to filter the information we need. In this task we will try to improve the number of reducer and check whether the number of reducers will affect the performance of the scan task. Looking through the bench log of the scan task, we can find that the task did not use any reducer just as we expected. However, it is better for us to have one reducer to combine the results from the mapper, otherwise, the output files from the mapper will be considered as the final output files, which is inconvenient in practice. We can set the Hive SQL's configuration to let if combine the files from the mappers using set hive.stats.autogather=false;



```
2020-11-08 15:01:41,215 Stage-1 map = 0%,  reduce = 0%
2020-11-08 15:01:59,025 Stage-1 map = 4%,  reduce = 0%, Cumulative CPU 136.68 sec
2020-11-08 15:02:00,066 Stage-1 map = 5%,  reduce = 0%, Cumulative CPU 159.99 sec
2020-11-08 15:02:02,150 Stage-1 map = 20%,  reduce = 0%, Cumulative CPU 206.95 sec
2020-11-08 15:02:03,193 Stage-1 map = 21%,  reduce = 0%, Cumulative CPU 215.57 sec
2020-11-08 15:02:05,276 Stage-1 map = 37%,  reduce = 0%, Cumulative CPU 265.59 sec
2020-11-08 15:02:06,320 Stage-1 map = 39%,  reduce = 0%, Cumulative CPU 269.49 sec
2020-11-08 15:02:08,407 Stage-1 map = 57%,  reduce = 0%, Cumulative CPU 322.09 sec
2020-11-08 15:02:09,452 Stage-1 map = 58%,  reduce = 0%, Cumulative CPU 326.04 sec
2020-11-08 15:02:11,531 Stage-1 map = 78%,  reduce = 0%, Cumulative CPU 383.88 sec
2020-11-08 15:02:14,662 Stage-1 map = 96%,  reduce = 0%, Cumulative CPU 452.14 sec
2020-11-08 15:02:15,703 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 468.12 sec
MapReduce Total cumulative CPU time: 7 minutes 48 seconds 120 msec
```

Figure 1. bench.log for the Scan Task

D. *Benchmark Results – Aggregation Task*

In this part of experiment, we will compare the speed and the throughput of the Hive SQL, and Spark SQL with different data size.

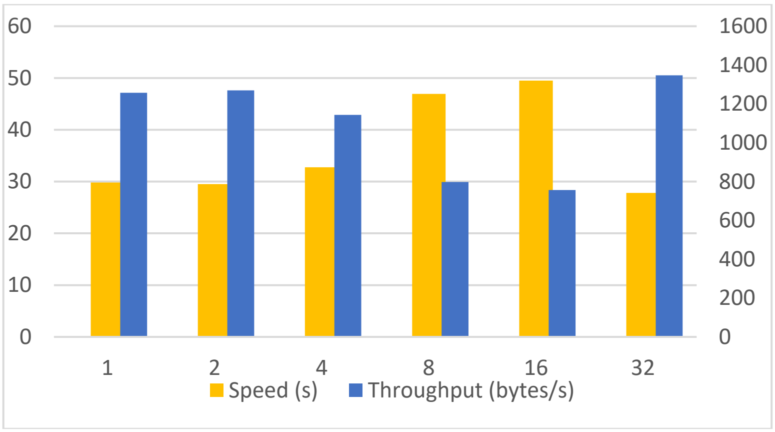


Figure 2 Spark SQL Throughput and Speed Comparison

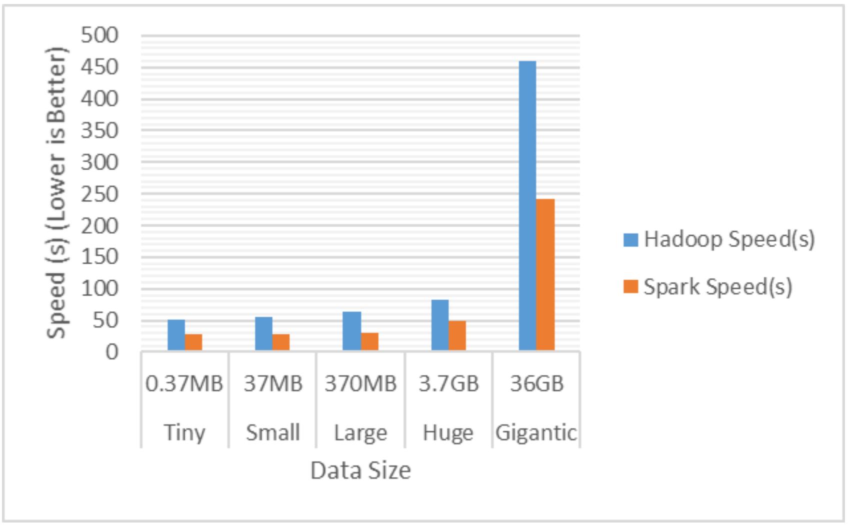


Figure 3. Speed Comparison (Lower is Better)

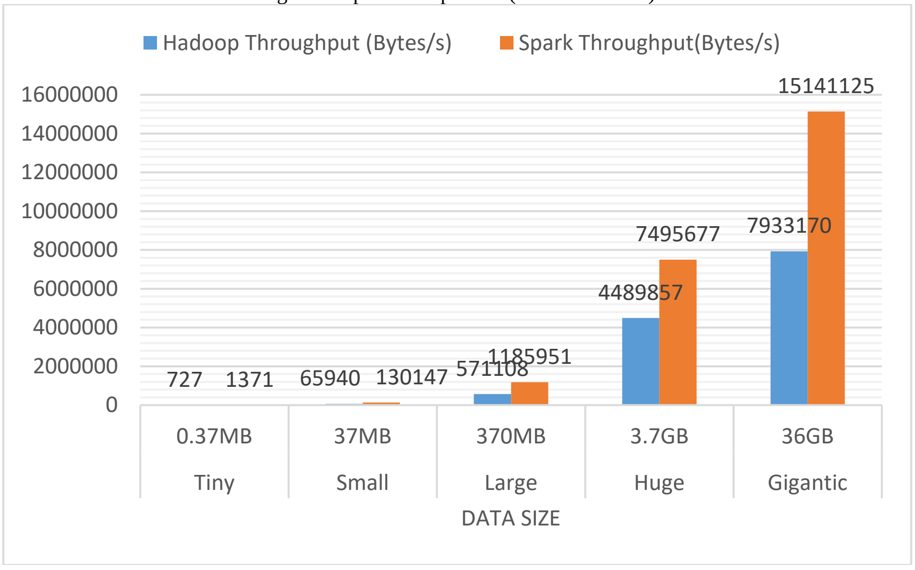


Figure 4. Throughput Comparison (Higher is Better)

In the first part of aggregation task we will try to increase the number of reducer and check whether the number of reducers will affect the performance of the scan task. Looking through the bench log of the aggregation task, we can find that the task used one mapper at the beginning and did not changed the number of mappers. And the number of mappers was doubled each time. According to the task result, we can find out that the speed reaches a higher level when reducer equals to 16 while the throughput is the lowest at the same time. In this experiment, it shows that too many reducers will waste resource and setting proper number of reducers will help increase the speed of Spark SQL.

From the above two charts, we can find that Spark SQL have larger throughput and better speed performance than Hive SQL. Spark SQL can use the hardware resource more efficiently so it has much larger throughput than Hive SQL. One reason for the big gap of the performance is Hive SQL always turns to allocate too many mappers and waste the resources and time as the following table shows. The influence of the number of mappers and reducers will be analyzed in the next part in detail. Also, Spark SQL's RDD and DAG framework give it the leverage to conduct in-memory execution, which will improve the performance a lot.

TABLE 2 DEFAULT NUMBER OF MAPPERS AND REDUCERS

Type	Data Size	Mapper	Reducer
Tiny	0.37MB	2	8
Small	37MB	5	8
Large	370MB	5	8
Huge	3.7GB	14	8
Gigantic	36GB	138	8

E. Benchmark Results – Join Task

The query is recorded in the appendix. In this experiment, we will Hive SQL to conduct the same query from the same size tables (51.8GB).

In this experiment, we will change the number of mappers and reducers to check the execution time and throughput. Then, we will use the collected data to calculate the speed up using Amdahl's law. It is hard for us to control the execution time of parallel tasks to a constant number, as a result, we will not use Gustafson's law to calculate the scaled speedup. Then we will use Amdahl's law to calculate the speed up with the following equation:

$$Speedup(\Lambda) = \frac{T(1)}{T(\Lambda)} \text{ [3]}$$

$T(\Lambda)$ is the execution time which can be got from the bench.log by calculate the difference between beginning time and an ending time. And we set $T(1) = 438\ s$ which comes from the cumulative CPU time in the bench.log when mapper = 2, reducer = 1. The cumulative CPU time does vary between different number of mappers and reducers, which can be explained with that: cumulative CPU time consists of not only the function execution time, but the time of internal data transfer and the data input or output time. Also, this cumulative time shows the sum of the CPU time from all the related processes. We can assume this time as our execution time with the lowest number of nodes. As a result, we think the number of mappers and reducers will not affect the exact execution time for the jobs, and take $T(1) = 438\ s$.
And the efficiency can be calculated with:

$$Efficiency(\Lambda) = \frac{Speedup(\Lambda)}{N(\Lambda)}$$

where $N(\Lambda)$ is considered as the number of ECU instances [4]. Our CPU has four 2.1GHZ cores and we consider it as 7 ECU instances. Thus, we set $N(\Lambda) = 7$ here. Time Consumed in Join Task and the relationship with the nodes is shown as below:

TABLE 3 RELATIONSHIP BETWEEN THE SPEED, THROUGHPUT, AND NUMBER OF MAPPERS AND REDUCERS

Mapper	Reducer	Speed (s)	Throughput(bytes/s)	Speed Up	Efficiency
2	1	270	7191277	1.622222222	0.23174603
2	2	251	7549770	1.74501992	0.24928856
2	4	244	7782571	1.795081967	0.25644028
4	1	188	10106880	2.329787234	0.33282675
4	2	195	9753666	2.246153846	0.32087912
4	4	199	9686854	2.201005025	0.31442929
8	1	145	13056466	3.020689655	0.43152709
8	2	143	12216466	3.062937063	0.43756244
8	4	149	13215136	2.939597315	0.41994247
12	1	141	13183344	3.106382979	0.43543769
12	2	143	13435621	3.062937063	0.43756244
12	4	141	13232044	3.106382979	0.44823769
20	1	160	11871189	2.733132475	0.39107143
20	2	164	12559188	2.670731707	0.38115331
20	4	163	11603092	2.687116564	0.38387379

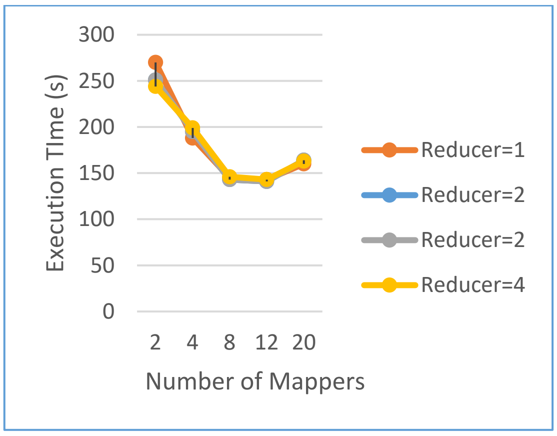


Figure 5. The Execution Time for Different Number of Mappers and Reducer

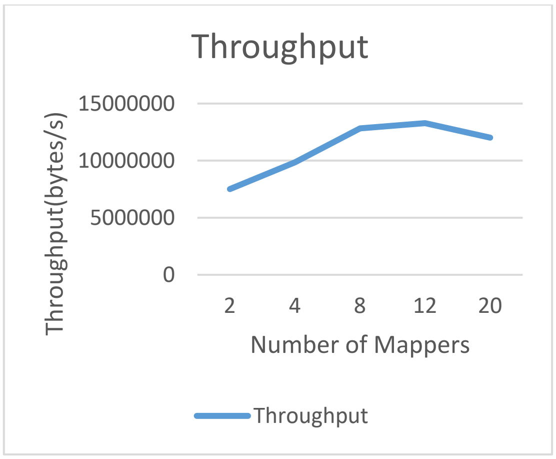


Figure 6. The Throughput for Different Numbers of Mapper

Based on the above table and figures, we have some interesting findings.

a. The number of reducers does not make a difference to the speed. Since the mappers' output files are not that big, it is enough to use only one reducer. Setting too many reducers will only waste the resources. Also, reducers will generate the smaller output files with the same amount to the numbers of reducers.

b. The throughput improves as the number of mappers increases when the mapper number is relatively small. However, when we add more mappers, we will find the throughput will keep almost the same and even drop, while the Speed of execution also drops. Since we only have 4 instances, and add more mappers will reach the threshold of our CPU's throughput, it is reasonable to see the through put reach a limit and even have a little drop. As to the speed, setting too much mappers will waste lots of time and we see the speed reduce when the number of mappers reaches 12.

c. When the Number of mappers is 12, the throughput and the speed up reach the highest. We can take 12 mappers, 1 reducer as the most appropriate setting for this data-size's join task.

As a result, it is not a good idea for a MapReduce task to have as much as mappers and reducers. Adding more mappers and reducers need more time to assign the tasks and generate some troublesome small files which will become a tricky problem if we want to query the data from the results. Usually, the Hive SQL will allocate more mappers than the appropriate number because it will not combine the smaller size files and as a result allocate more mappers to deal with those files. When the data size of the benchmark is as large as 150GB, it will allocate 150 mappers, which is definitely inappropriate.

Some other properties learned from this experiment:

We have mentioned that the CPU in the cloud is nearly 7 ECU. Also, we have 300GB disk space and 8GB memory. According to the price of EC2, the cost of our cloud can be considered as 0.25USD. Then we have $C(\Lambda) = 0.25$ Also since we do not meet hardware problems during our whole experiment, we can take the service availability as $\omega(\Lambda) = 99\%$. We can use that information and the speed data from the above table to calculate the productivity and the scalability:

$$P(\Lambda) = \frac{T(\Lambda) \times \omega(\Lambda)}{C(\Lambda)}$$

$$S(\Lambda, 1) = \frac{P(\Lambda)}{P(1)} = \frac{p(\Lambda) \times \omega(\Lambda) \times C(1)}{p(1) \times \omega(1) \times C(\Lambda)}$$

Take $p(1) = \frac{1}{438}, p(\Lambda) = \frac{1}{142}, C(1) = 0.15\ (2ECU), \omega = 0.99$ we can get the scalability $S(\Lambda, 1) = 1.85$.

F. *Conclusion and Improvement*

There are still some limitations to our experiment. For example, we only use a 4-node cluster with limited disk space to conduct the benchmarks. As a result, we only conduct the data size as big as 150GB. In the practical, we may come across much more complex problems.

Also, we cannot change the number of nodes, which is difficult for us to show the scalability and the speed up of the cluster. In the domain of large-scale structured data processing, we have parallelism DBMS like Presto DB, Big SQL, which are very popular in bigdata field. In the following study, we can compare the query speed of Spark SQL, Hive SQL with those popular parallelism databases to find their performance characteristics.

III. SORT AND TERASORT

A. *Introduction*

The sorting program has been pervasively accepted as an important performance indicator of MapReduce (e.g., it is used in the original MapReduce paper [5]), because sorting is an intrinsic behaviour of the MapReduce framework. For instance, the Hadoop Sort program [6] is often used as a convenient baseline benchmark for Hadoop, and is the primary benchmark for evaluating Hadoop optimizations in Yahoo [7]. The sort algorithm sorts its text input data, which is generated using

RandomTextWriter.Terasort is another widely used algorithm to do big-data sorting. On Hadoop, there are three applications related to Terasort. They are TeraGen, TeraSort and TeraValidate. TeraGen is a MapReduce program to generate the data. TeraSort samples the input data and uses map/reduce to sort the data into a total order. TeraValidate is a map/reduce program that validates the output is sorted [8].

In this paper, the sort algorithm and Terasort algorithm was run and tested on the cloud platform developed by the Inspur Corporation and Shenzhen Institute of Artificial Intelligence and Robotics for Society (AIRS). The sort and Terasort are two benches with similar functionalities. Therefore, this paper writes these two benches together. In the tests, the quantity of mappers and reducers, and the size of dataset will be modified. By modifying these parameters, it is possible to verify the relationship between these parameters and performance. The test bench was tested on both Hadoop and Spark to analyze the difference between Hadoop and Spark. During this process, the performance of the AIRS cloud system can also be measured.

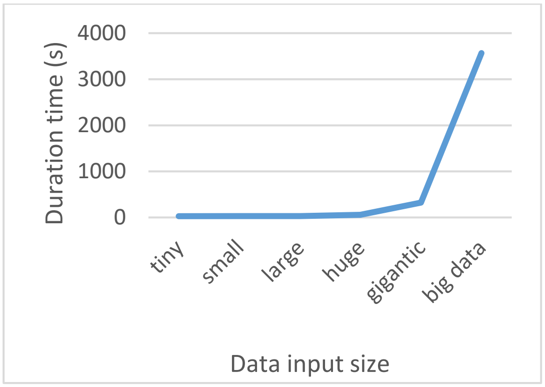
B. *Experimental Results and Analysis*

There are three sets of experiments. The first set of experiments modify the input data size. The second set of experiments modify the number of mappers and reducers. The third set of experiments investigate the difference of running time between Hadoop and Spark. The corresponding performance metrices are calculated and analyzed.

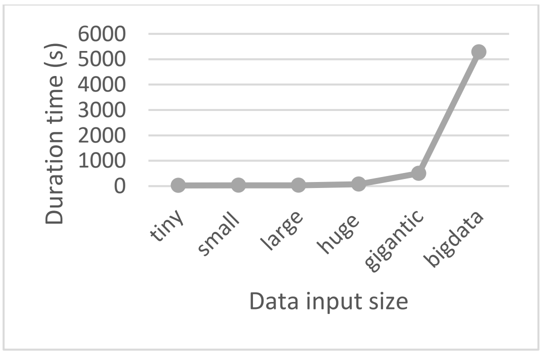
a) Tests on Different Data Sets: In these tests, different input data sets are tested. There are 6 data sets which are denoted as tiny, small, large, huge, gigantic and bigdata. Their corresponding values are $3.2 * 10^6$ bytes, $3.2 * 10^7$ bytes, $3.2 * 10^8$ bytes, $3.2 * 10^9$ bytes, $3.2 * 10^{10}$ bytes and $3.2 * 10^{11}$ bytes. Notice that all these dataset are as 10 times as the large as the former data set. The corresponding duration time, throughput/second, throughput/node and mapping/reducing tasks finished time are recorded. The results are shown in the following graphs or tables.

The relationship between duration time and data input size of sort is shown in graph 1. The relationship between duration time and data input size of Terasort is shown in graph 2. The duration time increases as the size of the data input increases. This phenomenon is intuitively true because the larger input requires more computing time. What should be noticed is that, the derivatives of the graph are changing dynamically. The duration time difference caused by two different large data set is significantly larger than the duration time difference caused by two different small data set. In the graph 1 and 2, it is obvious that the time differnce between input “bigdata” and input “gigantic” is much larger than the time differnce between input “tiny” and input “small”.

There are two reasons for this phenomenon. The first reason is that the available virtual cores for each group is limited, thus the computing power is insufficient when dealing with huge data set. When the dataset gets large, the insufficient situation gets worse, which makes it harder to exploit the algorithm. For instance, when the dataset is large, the memory might not be able to put all dataset inside. Thus, the distributed design cannot be fully applied. The second reason is about the algorithm. Although this algorithm is already a smart-designed algorithm, it is still not a linear algorithm. So it is impossible for the duration time to increase acoording to the dataset linearly. The third reason is that the machine also needs to do I/O operations (such as loading data from hard disks) and I/O operations take a lot of time compared with CPU computations []. More time spent on I/O operations.



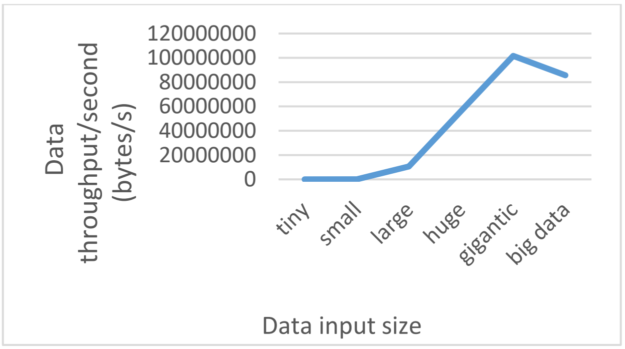
Graph 1: Relationship between duration time and data input size of Sort



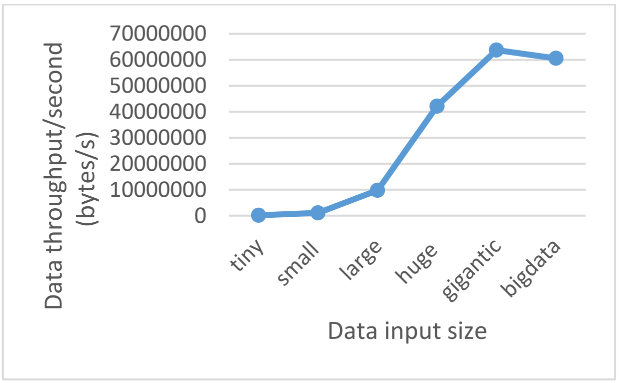
Graph 2: Relationship between duration time and data input size of Terasort

The relationship between data throughput/seconds and data input size is shown in graph 3 and 4 for sort and Terasort, respectively. When the input data size switches from tiny to small, and from small to large, the increase of throughput/second is not very large. When the input data size switches from large to huge, and from huge to gigantic, the increase of throughput/second is very large. A possible reason is that when the data set is small, the program can be finished fastly, thus the throughput/second is small; when the data set is large, the program is executed slowly (can be viewed in duration time), thus the value gets big.

Another interesting result is that the throughput/second gets smaller when the dataset changes from gigantic to bigdata. The possible reason is that the available virtual cores for each group is limited, thus the computing power is insufficient when dealing with huge data set. When the dataset gets large, the insufficient situation gets worse, which makes it harder to exploit the algorithm. For instance, when the dataset is large, the memory might not be able to put all dataset inside. Thus, the distributed design cannot be fully applied.



Graph 3: Relationship betweem data throughput/second and input data size for Sort



Graph 4: Relationship betweem data throughput/second and input data size for Terasort

b) Tests on different number of mappers and reducers: In these tests, different number of mappers and reducers are tested. The corresponding duration time, total map task finishing time and reduce map task finishing task are recorded. In the Terasort test bench, the input data size is large, which is $3.2 * 10^8$ bytes. In the Sort test bench, the input data size is huge, which is $3.2 * 10^9$ bytes. The results are shown in the following tables.

In table 4 and 5, the number of the reducers is settled unchanged. The number of reducer is 4. The number of mappers are configured to be 3, 5, 7, 9, 11. The table 1 represents the data obtained from Sort whose input size is huge. The table 2 represents the data obtained from Terasort whose input size is large. In table 3 and 4, the number of the mappers is settled unchanged. The number of mapper is 7. The number of reducer are configured to be 2, 4, 6, 8, 10. These four table shares similar characteristics so this paper only choose to discuss table 1 and 2.

In table 6 and 7, there is no significant correlation between the number of mappers and the duration time. This phenomenon is counter-intuitive because the computing speed should increase as the number of mappers increase. However, the results from table 1 and 2 does not reflect this instinctive conclusion. The reason for this phenomenon is that the number of mappers configured in the configuration file *hibench.conf* is not the real number of mappers. The Hadoop can adjust the number of mappers to best accommodate the programming situation. Thus, when the data input gets large, the mappers will be increased by Hadoop.

Number of mappers	Duration time (s)
3	55.71
5	63.87
7	72.90
9	66.12
11	70.93

TABLE 4: RELATIONSHIP BETWEEN NUMBER OF MAPPERS AND DURATION TIME OF SORT

Number of mappers	Duration time (s)
3	32.793
5	32.72
7	32.833
9	30.695
11	28.543

TABLE 5: RELATIONSHIP BETWEEN NUMBER OF MAPPERS AND DURATION TIME OF TERASORT

Number of reducers	Duration time (s)
2	87.33
4	66.23
6	73.72
8	66.85
10	63.76

TABLE 6: RELATIONSHIP BETWEEN NUMBER OF REDUCERS AND DURATION TIME OF SORT

Number of reducers	Duration time (s)
2	30.657
4	32.833
6	32.806
8	31.965
10	30.531

TABLE 7: RELATIONSHIP BETWEEN NUMBER OF REDUCERS AND DURATION TIME OF TERASORT

Mapper	Reducer	Speed (s)	Throughput(bytes/s)	Speed Up	Efficiency
1	1	32.777	3106425	0.37	0.37
1	2	31.728	3209283	0.55	0.27
2	1	29.635	3435990	0.64	0.32
2	2	30.651	3322024	0.78	0.39
3	1	32.496	3133289	0.79	0.26
3	2	29.612	3438599	0.99	0.33
4	1	27.728	3672225	1.11	0.2775
4	2	29.502	3451393	1.17	0.29
6	1	30.505	3337987	1.43	0.36
6	2	29.809	3415923	1.62	0.41

TABLE 8 RELATIONSHIP BETWEEN THE SPEED, THROUGHPUT, AND NUMBER OF MAPPERS AND REDUCERS IN DATA OF 100000000 BYTES

Mapper	Reducer	Speed (s)	CPU time (ms)	Speed Up	Efficiency
26	4	45	299560	6.65	1.466
30	4	49	360530	7.35	1.83
40	4	57	404290	7.09	1.77
50	4	65	473940	7.29	1.82

TABLE 9 RELATIONSHIP BETWEEN THE SPEED, CPU TIME, AND NUMBER OF MAPPERS AND REDUCERS IN HUGE DATA

In table 8, the speed up of the program is smaller than 1 first, then increase gradually with the growth of number of mappers. However, the execution time didn’t decrease obviously with the increase of number of mappers. We guess that it may because the dataset is too small, and the setting of the program take up a large rate of the program. Then we increase the dataset in table 9. However, the running time decrease as the number of mappers increase from 26 to 50. This is mainly because that we only have four cores. Although in some time we may be able to utilize up to fifteen cores, it is still quite smaller than the number of mappers.

We find that the mapper we set in hibench.conf will not always be equal to the actual mapper running the program. For example, we run sort in Hadoop for the huge data (3200000000 bytes) and set the mapper to be 11. However, we find that the actual mapper number of 33 in log file. Then we know how it works, the program has the set of minimum number of mappers.

$$minimum\ number\ of\ mappers = floor(\frac{(size\ of\ program)}{1024 * 1024 * 128})$$

If the mapper we set is smaller than the minimum, then the actual number of mapper will be the minimum number of mappers calculated by the program. If the mapper we set is larger than the minimum, then the actual number of mappers will be the number of mappers we set.

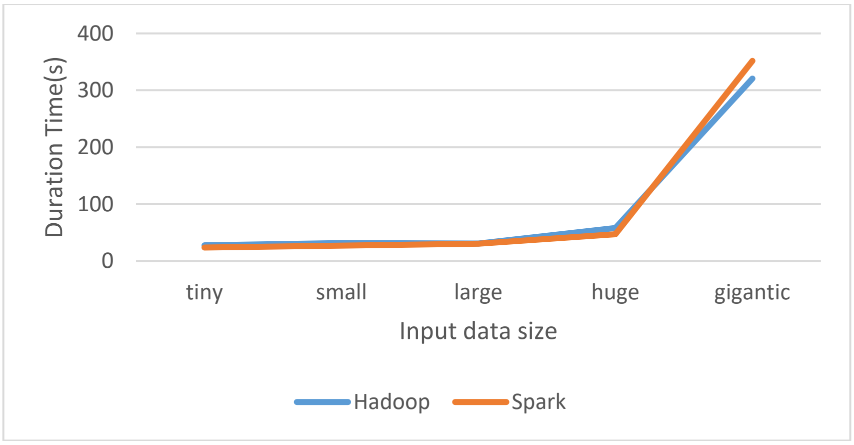
Also, for the speed up and the efficiency of the program.
For the speed up, we could consider the sequential time as the CPU time spent in bench.log. As it is roughly equal to the Total time spent by all map tasks and Total time spent by all reduce tasks. But it is a little smaller the sum of the above two. The total time spent by all map tasks is equal to the sum of all the mappers, it is similar in the case of reducer. Then we need to calculate the actual time of this work. In this case, we roughly choose the running time in the end of bench.log as the parallel time. But it is a little larger than the actual value as it contains some time to set up.

$$Speed\ up \approx \frac{CPU\ time\ spent\ in\ bench.log}{the\ actual\ time\ of\ this\ work\ in\ bench.log}$$

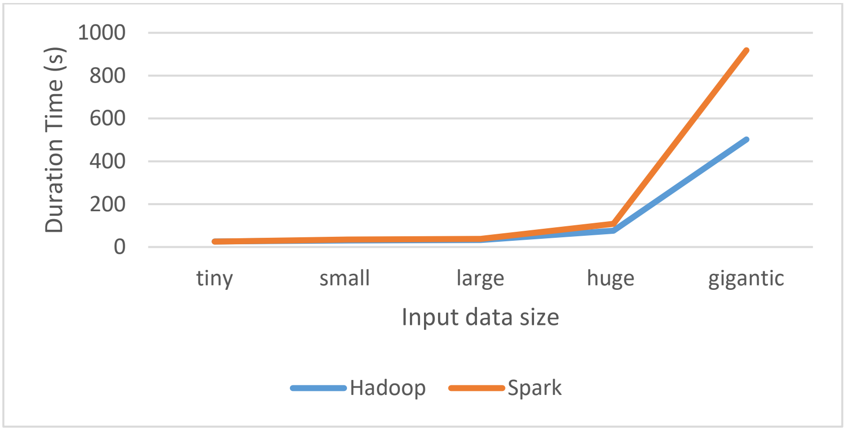
For the efficiency, as the number of nodes in our program is four. If the number of mappers or reducers is larger than four, then efficiency is equal to speed up divided by four. If the number of mapper is one and the number of reducer is one, as only one mapper or reducer can run in one node and the program is sequential, efficiency is equal to speed up divided by one. If the number of mapper is two and the number of reducer is one, then efficiency is equal to speed up divided by two.

However, this arouse some problems. If we choose very small data size, like the case of table 8. Here we calculate the minimum number of mappers = 100000000 bytes/(1024*1024*128) = 0.74 <1, we could set the mapper equal to 1. However as the size of the program is too small, with the increase of the number of mappers, the time to set up the program is very significant of the program. So the program didn’t run faster as we expect. Then we enlarge the size of program. In table 9, the minimum number of mappers = 3200000000 bytes/(1024*1024*128)= 23.6. Then we choose the mapper number of 26,30,40,50. However, as the number of cores is limited, varying from four to fifteen. The increase of mapper didn’t bring the expected result to us. In conclusion, we cannot get the exact number of cores, and the time of setting up or preparing for the mapper is too long for small size input. These two make the following table didn’t have the expected result.

c) Tests on Hadoop and Spark: Both Hadoop and Spark have implemented the same Sort and Terasort experiment. Both of them have used data with tiny, small, large, huge, gigantic size. The duration time of them have been posted in graph 5 and 6. It is counter-intuitive to see that the running time of Spark is even larger than Hadoop when they are using the same size of data. The major reason is that Hadoop is a hard disk-based platform, while Spark is a memory-based platform. The data structure Resiliennt Distributed Datasets (RDD) used in Spark is utilized in memory. Spark requires big memory, when the memory is not big enough, the needed data will be stored in hard disk. This leads to high latency and causes the slowing down of Spark in the AIRS cloud platform.



Graph 5: Execution time difference between Hadoop and Spark for Sort



Graph 6: Execution time difference between Hadoop and Spark for Terasort

C. *Conclusion*

The Sort and Terasort are tested on the cloud. The data size, number of reducers and mappers are modified. Both Sort and Terasort are executed. The analysis are shown in the upper part. There are several key findings. First, spark behaves worse than Hadoop in these two test benches. The reason is that the Hadoop runs in memory and the provided memory is not sufficient. Another finding is that Hadoop can distribute the mappers according to the computation need. The real number of mappers and reducers can be calculated. The method and speed-up analysis are presented in the second part of the test result part.

IV. PageRank

A. Introduction

PageRank, named after Lawrence Edward Page, is an algorithm to rank web pages in searching engine results. In such an era with abundant information no matter valuable or invaluable, PageRank is also dealing with a huge amount of data. Therefore, it is important to find if parallel algorithm of PageRank could accelerate its calculation significantly. In this experiment, the workload benchmarks PageRank algorithm is implemented in Hadoop examples. The data source is generated from Web data whose hyperlinks follow the Zipfian distribution.

B. Experimental Results

The experimental result is too large to attach. It could be downloaded at https://github.com/WilliamDYW/CSC4160Proj1Exp.Result.

C. Result Analysis

According to Amdahl's law:

$$S = \frac{1}{1 - P + \frac{P}{N}}$$

in this experiment, the speedup could be measured by operating time given equal size of input data, that is:

$$\frac{T_{accelerated}}{T_{original}} = \frac{1}{S} = 1 - P + \frac{P}{N}$$

Let 1-P = a, P = b, the equation becomes:

$$\frac{T_{accelerated,N\ Core}}{T_{original}} = a + \frac{b}{N}$$

It could be assumed that given the same task and size of input, the quantity of *a* and *b* must be constant. Therefore, the following method could be used for calculation of *a* and *b*:

$$\frac{NT_{accelerated,N\ Core}}{T_{original}} = aN + b$$

$$\frac{(N+1)T_{accelerated,N+1\ Core}}{T_{original}} = a(N+1) + b$$

As a result, *a* could be calculated by the following equation:

$$a = \frac{(N+1)T_{accelerated,N+1\ Core} - NT_{accelerated,N\ Core}}{T_{original}}$$

b could be calculated by the following equation:

$$\frac{b}{N} = \frac{T_{accelerated,N\ Core}}{T_{original}} - a$$

As a result, the nonparallel work time could be measured as $aT_{original}$, and parallel work time should be $T_{accelerated,N\ Core} - aT_{original}$

Such method will be implemented for average calculation:

For numbers x_1-x_6, their average difference $\overline{diff} = \frac{x_6 + x_5 + x_4 - x_3 - x_2 - x_1}{9}$

In order to avoid significant error, the following modifications will be implemented:

1) As data of “tiny” and “small” sizes are too small to avoid significant influence of unexpected fluctuations, while data of “bigdata” size requires a huge amount of computing time, the analysis is mainly focused on data of “large” and “huge” sizes.

2) Most of experiment results showed that when the cores become large enough (e.g. 7 cores), the operation time of hadoop will unexpectedly increase. In order to keep the accuracy of proportion of parallel work, such results will not be calculated into average difference.

TABLE I. 7 MAPPERS HANDLING HUGE SIZE OF DATA

Mapper Number	Reducer Number	Data Size	Operation Time	Single-Core Time	Average Difference	Time of Parallel Work	Time Comparison (Single core = 1)	Speedup (Single core = 1)
7	1	Huge	3983.774	3983.774	597.15317	3386.621	1	1
	2		2427.879	4855.758		1830.726	0.609441951	1.640845
	3		1926.202	5778.606		1329.049	0.483511866	2.068202
	4		1445.722	5782.888		848.5688	0.362902614	2.755560
	5		1300.138	6500.69		702.9848	0.326358373	3.064116
	6		1185.558	7113.348		588.4048	0.297596701	3.360252
	7		1167.134	8169.938		569.9808	0.292971941	3.413296
	8		1470.648	11765.184		873.4948	0.369159495	2.708856

The former 7 data point of time of parallel work could be fit into a power function:

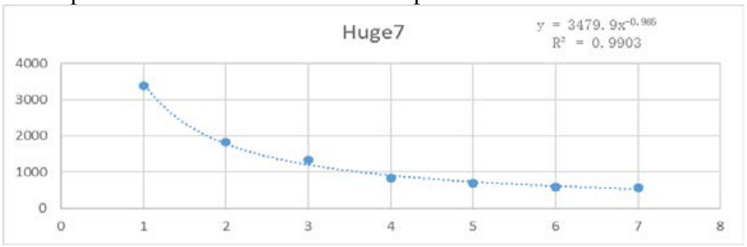


Fig. 1. Varying parallel work time of different numbers of reducers under 7 mappers handling huge size of data

TABLE II. 5 MAPPERS HANDLING HUGE SIZE OF DATA

Mapper Number	Reducer Number	Data Size	Operation Time	Single-Core Time	Average Difference	Time of Parallel Work	Time Comparison (Single core = 1)	Speedup (Single core = 1)
5	1	Huge	3048.11	3048.11	837.474222	2210.636	1	1
	2		2214.363	4428.726		1376.889	0.726470829	1.376518
	3		1927.418	5782.254		1089.944	0.632332166	1.581447
	4		1428.908	5715.632		591.4338	0.468784919	2.133174
	5		1420.492	7102.46		583.0178	0.466023864	2.145813
	6		1329.711	7978.266		492.2368	0.436241146	2.292310
	7		1934.359	13540.513		1096.885	0.634609315	1.575773
	8		1509.161	12073.288		671.6868	0.495113693	2.019738

The former 6 data point of time of parallel work could be fit into a power function:

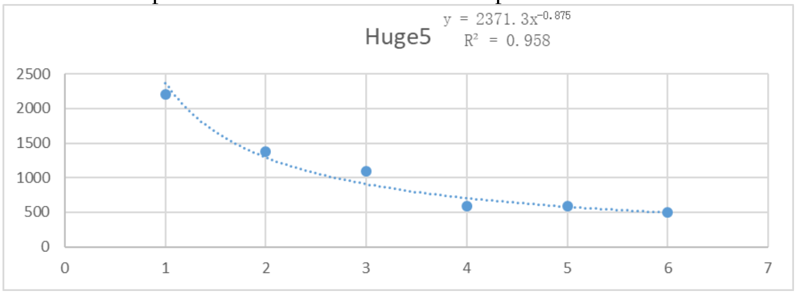


Fig. 2. Varying parallel work time of different numbers of reducers under 5 mappers handling huge size of data

TABLE III. 8 MAPPERS HANDLING HUGE SIZE OF DATA

Mapper Number	Reducer Number	Data Size	Operation Time	Single-Core Time	Average Difference	Time of Parallel Work	Time Comparison (Single core = 1)	Speedup (Single core = 1)
8	1	Huge	3267.595	3267.595	768.414	2499.181	1	1
	2		2526.91	5053.82		1758.496	0.773324111	1.293119
	3		1912.843	5738.529		1144.429	0.585397823	1.708240
	4		1520.891	6083.564		752.477	0.465446605	2.148474
	5		1369.667	6848.335		601.253	0.419166696	2.385686
	6		1425.815	8554.89		657.401	0.436349976	2.291738
	7		1607.644	11253.508		839.23	0.491996101	2.032536
	8		1722.751	13782.008		954.337	0.527222927	1.896731

The former 5 data point of time of parallel work could be fit into a power function:

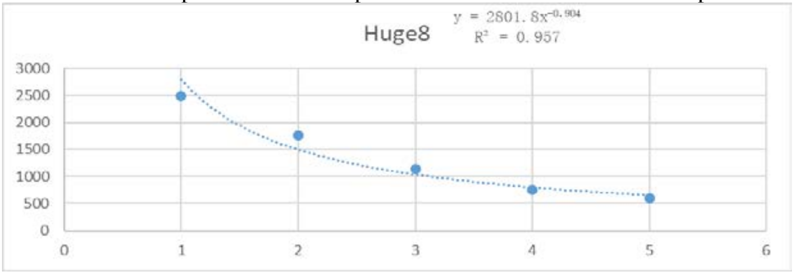


Fig.4. Varying parallel work time of different numbers of reducers under 8 mappers handling huge size of data

TABLE IIII. 5 MAPPERS HANDLING LARGE SIZE OF DATA

Mapper Number	Reducer Number	Data Size	Operation Time	Single-Core Time	Average Difference	Time of Parallel Work	Time Comparison (Single core = 1)	Speedup (Single core = 1)
5	1	Large	589.645	589.645	242.993	346.6518	1	1
	2		346.315	692.63		103.3218	0.587327969	1.702626
	3		328.005	984.015		85.01183	0.556275386	1.797671
	4		322.606	1290.424		79.61283	0.547119029	1.827756
	5		289.962	1449.81		46.96883	0.491756905	2.033525

	6		304.469	1826.814		61.47583	0.516359844	1.936634
	7		297.709	2083.963		54.71583	0.504895318	1.980609
	8		309.058	2472.464		66.06483	0.524142493	1.907878

The former 5 data point of time of parallel work could be fit into a power function:

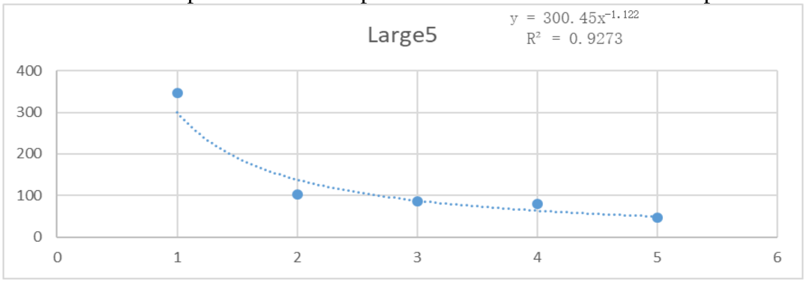


Fig.4. Varying parallel work time of different numbers of reducers under 5 mappers handling large size of data

TABLE V. 3 Mappers Handling Large Size of Data

Mapper Number	Reducer Number	Data Size	Operation Time	Single-Core Time	Average Difference	Time of Parallel Work	Time Comparison (Single core = 1)	Speedup (Single core = 1)
	1		508.749	508.749		346.6518	1	1
	2		429.312	858.624		103.3218	0.84385817	1.185033
	3		344.477	1033.431		85.01183	0.677105999	1.476874
3	4	Large	329.803	1319.212	242.993	79.61283	0.648262699	1.542585
	5		310.237	1551.185		46.96883	0.609803656	1.639872
	6		299.338	1796.028		61.47583	0.588380518	1.699580
	7		285.05	1995.35		54.71583	0.560295942	1.784771
	8		380.108	3040.864		66.06483	0.747142501	1.338433

The former 7 data point of time of parallel work could be fit into a power function:

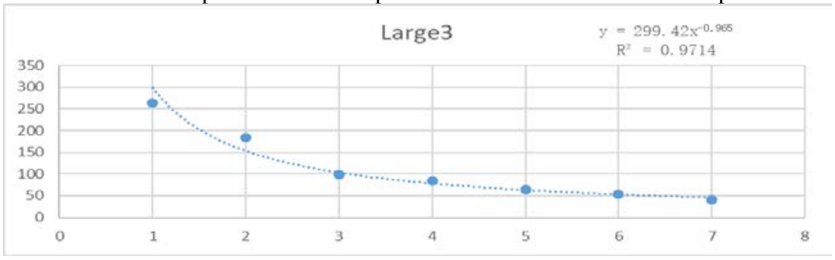


Fig.5. Varying parallel work time of different numbers of reducers under 5 mappers handling large size of data

TABLE VI. Relationship between Input Data Size and Throughput

Input data size	Throughput	Number of Mapper	Number of Reducer
10712	192		
1813330	10694		
259901664	930509	4	4
2993339197	2339407		
19933279609	2377193		
33884243335	2677541		

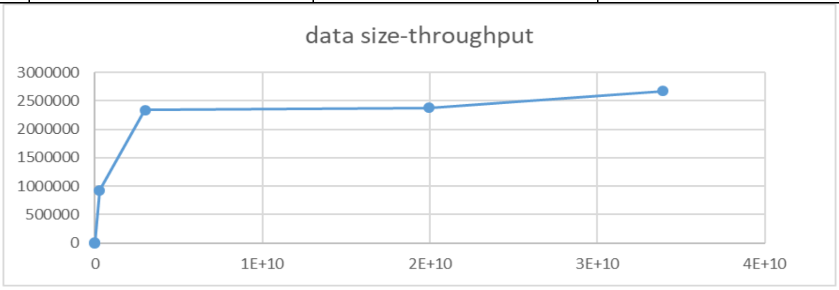


Fig.6. Relationship between input data size and throughput

TABLE VII. Throughput of Huge Size of Data

1	2	3	4	5	6	7	8	Mapper/

								Reducer
1955234	1780301	1919848	2067164	1983468	1696082	2035421	1737597	8
2192374	1937048	1883171	2505108	1547475	2024470	2564734	1862009	7
1458922	2087555	1451570	2408199	2251145	1776603	2524877	2099464	6
1628799	2260543	1582337	2112600	2107278	2112339	2302362	2185529	5
1597157	1846252	1715660	1836280	2094867	1733531	2070514	1968219	4
1277906	1511516	1359660	1828531	1553047	1457197	1554036	1564920	3
1337788	1418691	1256809	1563904	1351798	1238108	1232923	1184627	2
579949	1006250	876112	1048987	982042	881079	751395	916101	1

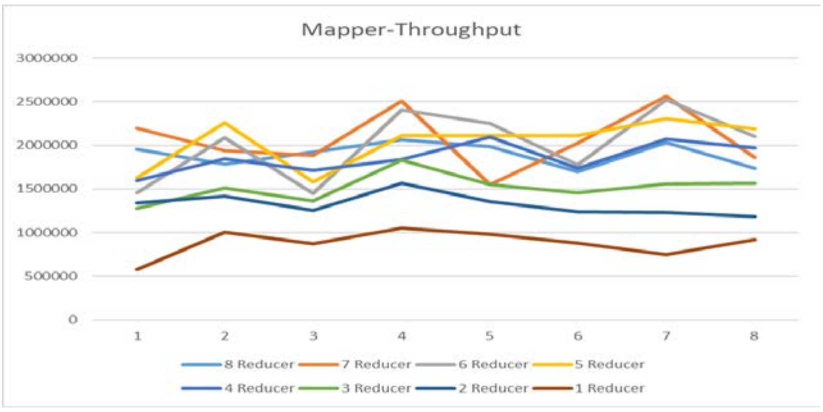


Fig.7. Throughput of huge size of data

D. Discussion

1) Most of experiment results showed that when the core number become large enough (e.g. 7 cores), the operation time of hadoop will unexpectedly increase. Besides fluctuation of data due to multitask processing, the regularity of such phenomena might have some other reasons. Generally, there are only 4 cores in the cloud computer. As a result, when the core required is more than 4, the computer might cooperate with other computers to finish given work. However, it may take extra time for extra cores. When the core number is large enough, the extra time might not be offset. This may be a reason for the increasing operation time.

2) When number of mappers change, the throughput does not change significantly. This may be due to the speed of mapper is far faster than reducer; As a result, a decreased mapper speed could hardly affect the total operation time.

3) When input data size rises, the throughput will rise rapidly at first; as the size becomes large enough, the throughput will become stable. This may be due to the cloud elasticity. At first the computer could adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible; however, when the size becomes too large, the resource required may reach the maximum limit of computer. Therefore, the throughput would become stable.

E. Conclusion

In ideal conditions (i.e. enough parallel cores), the speedup of cloud computer fits the Amdahl’s law; meanwhile, the elasticity could be found in varied input data size. However, the actual condition might be constrained by various conditions, such as insufficient cores and computing power. As a result, the practical performance could usually fit Amdahl’s law, but hardly reach theoretical performance.

V. Conclusion

In our experiments, we use lots of different kinds of experiments to test the performance of AIRS. We also use the knowledge of the cloud performing to analyze our results. At the same time, we have a deeper understanding of how the MapReduce system works with HDFS and Yarn.

There are still some results that are out of our expectation. For example, we expect the Spark's performance may be better in some experiments, but the result is Hadoop do better than Spark. We also have some speed up that is less than 1 when we conduct the MapReduce algorithm. Although we try our best to make some reasonable explanation, we still need to do further study. We should monitor our disk and memory usage to prove our ideas and learn more about the AIRS cloud properties to do further and more concise results.

VI. APPENDIX

1. Introduction to Hive SQL

In this section, we give an overview of a Bigdata SQL system Hive SQL.

Recently the trade press has been filled with news of the revolution of "cluster computing." With the rise of interest in clusters computing, some tools for programming have come proliferation. One of the most and best known such tools is MapReduce (MR). MR is attractive because it provides a model through which users can express relatively distributed programs [9]. Hadoop has been the big shining star in the MapReduce framework. And spark improves the Hadoop's framework and using RDDs and DAG to achieve better performance and has more function as a streaming program. The Apache Hive is introduced as a data warehousing solution that can run on top of Hadoop, Spark, or Tez as its execution engine to achieve the goal of dealing with big data.

The compilation and execution process of the Hive SQL instructions:

a. Parse the code and finish the syntax, semantic analysis. Then generate an abstract syntax tree.
b. Traverse the syntax tree and get the basic query unit – Query Block.
c. Traverse the Query Block and translate it to the operator tree.
d. Traverse the operator tree and generate a MapReduce task.
e. Send the job to the Hadoop's job tracker (or YARN) to execute the result.
f. Hadoop sends the results back to the HIVE space, and the user can check the result.

Figure 1

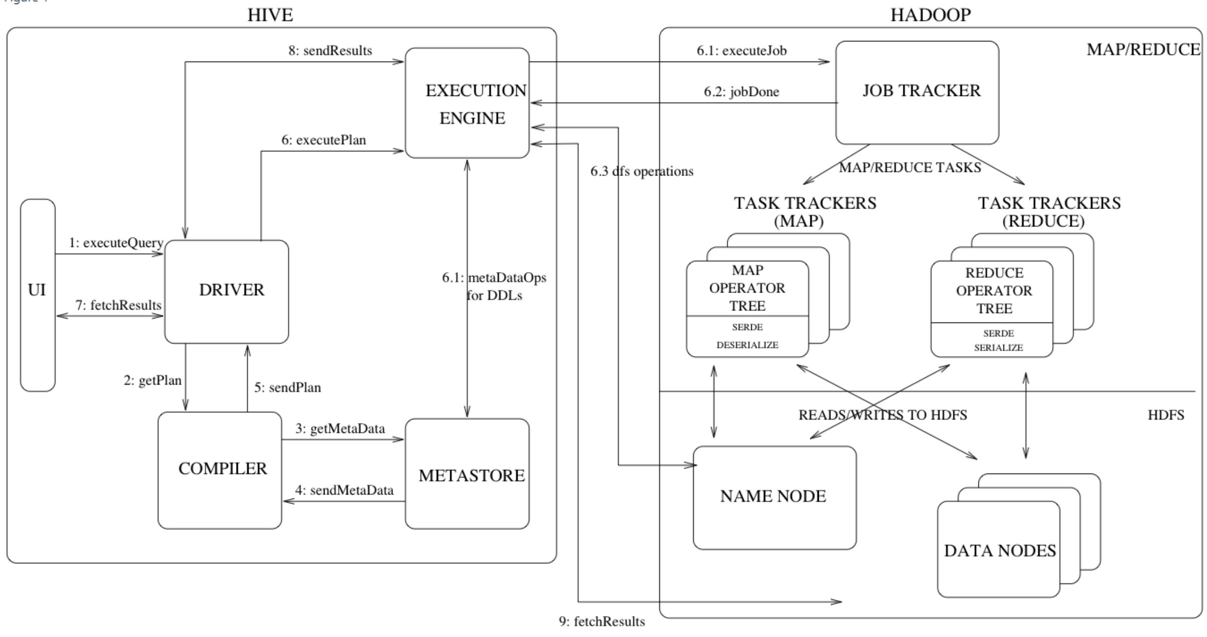


Figure II.1 Major of Hive and Its Interactions with Hadoop[10]

In this experiment, we will focus on the Hive SQL and the Spark SQL to test their performance with different parameters. Every Hive SQL instruction can be turned into a MapReduce task.

Mapper: Basically, a mapper is a filter that filters and organizes data in sorted order. For example, if we execute this instruction: $SELECT\ col1, col2\ FROM\ exampleTable\ LIMIT\ 5l;$ The mapper will filter the col1 and col2 out from the table.

Reducer: Reducer summarizes operation data across the rows. For example, get the sum of a column.

HiBench gives three kinds of instructions in the SQL benchmark.

a. Scan: Those instructions will select data from the generated input's **uservisits** table, and insert the data to the output's **uservisits_copy** table.
b. Aggregation: This benchmark will group up the *sourceIPs* and calculate the sum of the *adRevenue* for all the *adRevenue* with the same *sourceIP*. Actually, the implementation of GROUP BY (Aggregation) is very similar to the wordcount. Here is a dataflow graph:

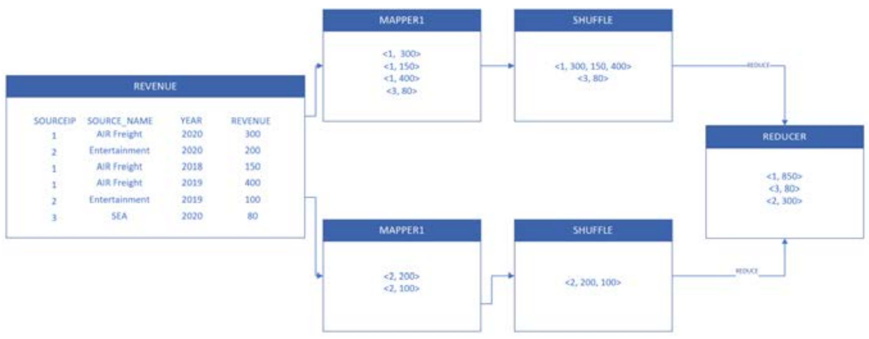


Figure II.2 Dataflow of a Aggregation Program Through MapReduce

c. Join: This benchmark follows from the Hive Aggregation, and it is more complicated. This benchmark has two tables: **uservisits** and **rankings.** It will compute both the average and sum for each group by joining two different tables and generate a new table **rankings_uservisits_join**[11]. Join instruction also needs both mapper and reducer.records the instruction of Hive SQL used in Join benchmark. Here is a dataflow graph for a more straightforward example of join:

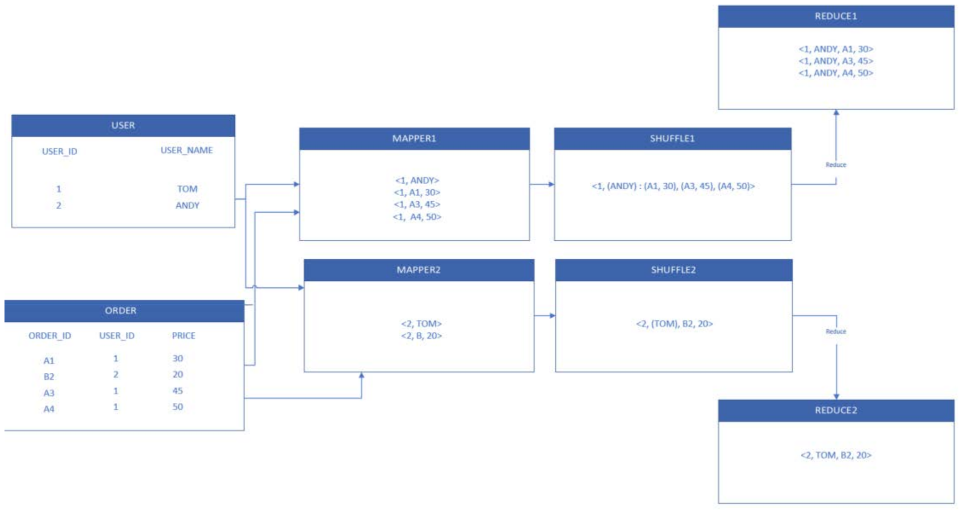


Figure II.3 Dataflow of a Join Program Through MapReduce

2. Spark SQL

In order to decrease the workload, we use Spark SQL, which is Apache Spark's module for working with structured data. It has following advantages:

Integrated, which means that Spark SQL let the users query structured data inside Spark programs using SQL.
Uniform Data Access, the Spark SQL connect to any data source the same way. SQL provides a common way to access a variety of data sources, and we use Hive in our project.

Hive Integration, Spark SQL can run SQL or HiveQL queries on existing warehouses. Spark SQL supports the HiveQL syntax as well as Hive SerDes and UDFs, allowing you to access existing Hive warehouses.

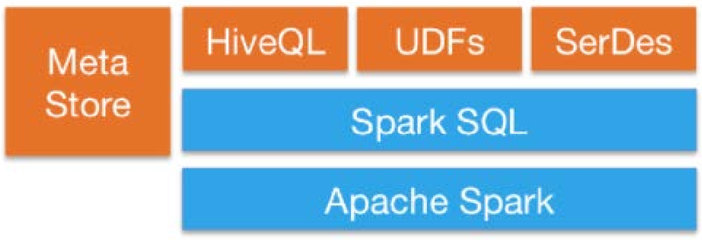


Figure II.4 Spark SQL Structure[12]

3. HIVE SQL Scan Instruction

DROP TABLE IF EXISTS uservisits;
CREATE EXTERNAL TABLE uservisits (sourceIP STRING,destURL STRING,visitDate STRING,adRevenue DOUBLE,userAgent STRING,countryCode STRING,languageCode STRING,searchWord STRING,duration INT) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde' STORED AS SEQUENCEFILE LOCATION 'hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Scan/Input/uservisits';
DROP TABLE IF EXISTS uservisits_copy;
CREATE EXTERNAL TABLE uservisits_copy (sourceIP STRING,destURL STRING,visitDate STRING,adRevenue DOUBLE,userAgent STRING,countryCode STRING,languageCode STRING,searchWord STRING,duration INT) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde' STORED AS SEQUENCEFILE LOCATION 'hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Scan/Output/uservisits_copy';
INSERT OVERWRITE TABLE uservisits_copy SELECT * FROM uservisits;

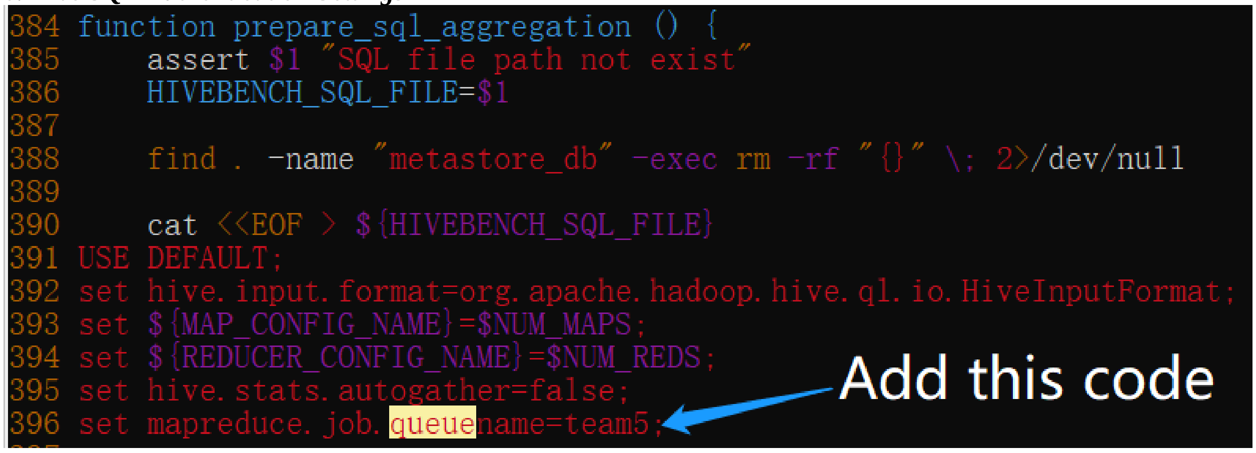
4. HIVE SQL Aggregation Instruction

DROP TABLE IF EXISTS uservisits;
CREATE EXTERNAL TABLE uservisits (sourceIP STRING,destURL STRING,visitDate STRING,adRevenue DOUBLE,userAgent STRING,countryCode STRING,languageCode STRING,searchWord STRING,duration INT) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde' STORED AS SEQUENCEFILE LOCATION 'hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Aggregation/Input/uservisits';
DROP TABLE IF EXISTS uservisits_aggre;
CREATE EXTERNAL TABLE uservisits_aggre (sourceIP STRING, sumAdRevenue DOUBLE) STORED AS SEQUENCEFILE LOCATION 'hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Aggregation/Output/uservisits_aggre';
INSERT OVERWRITE TABLE uservisits_aggre SELECT sourceIP, SUM(adRevenue) FROM uservisits GROUP BY sourceIP;

5. HIVE SQL Join Instruction

DROP TABLE IF EXISTS rankings;
CREATE EXTERNAL TABLE rankings (pageURL STRING, pageRank INT, avgDuration INT) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde' STORED AS SEQUENCEFILE LOCATION 'hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Join/Input/rankings';
DROP TABLE IF EXISTS uservisits_copy;
CREATE EXTERNAL TABLE uservisits_copy (sourceIP STRING,destURL STRING,visitDate STRING,adRevenue DOUBLE,userAgent STRING,countryCode STRING,languageCode STRING,searchWord STRING,duration INT) ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde' STORED AS SEQUENCEFILE LOCATION 'hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Join/Input/uservisits';
DROP TABLE IF EXISTS rankings_uservisits_join;
CREATE EXTERNAL TABLE rankings_uservisits_join (sourceIP STRING, avgPageRank DOUBLE, totalRevenue DOUBLE) STORED AS SEQUENCEFILE LOCATION 'hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Join/Output/rankings_uservisits_join';
INSERT OVERWRITE TABLE rankings_uservisits_join SELECT sourceIP, avg(pageRank), sum(adRevenue) as totalRevenue FROM rankings R JOIN (SELECT sourceIP, destURL, adRevenue FROM uservisits_copy UV WHERE (datediff(UV.visitDate, '1999-01-01')>=0 AND datediff(UV.visitDate, '2000-01-01')<=0)) NUV ON (R.pageURL = NUV.destURL) group by sourceIP order by totalRevenue DESC;

6. Hive SQL Authentication Settings



```
function prepare_sql_aggregation () {
    assert $1 "SQL file path not exist"
    HIVEBENCH_SQL_FILE=$1

    find . -name "metastore_db" -exec rm -rf "{}" \; 2>/dev/null

    cat <<EOF > ${HIVEBENCH_SQL_FILE}
USE DEFAULT;
set hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;
set ${MAP_CONFIG_NAME}=$NUM_MAPS;
set ${REDUCER_CONFIG_NAME}=$NUM_REDS;
set hive.stats.autogather=false;
set mapreduce.job.queuename=team5;
```

7. Hive SQL Setting of Mapper and Reducer

[reducer] set mapred.reduce.tasks = 15;
[mapper] = 2
set mapred.max.split.size=10000000000;

set mapred.min.split.size.per.node=10000000000;
set mapred.min.split.size.per.rack=10000000000;
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
[mapper] = 4
set mapred.max.split.size=700000000;
set mapred.min.split.size.per.node=700000000;
set mapred.min.split.size.per.rack=700000000;
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
[mapper] = 8
set mapred.max.split.size=250000000;
set mapred.min.split.size.per.node=250000000;
set mapred.min.split.size.per.rack=250000000;
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat;
[mapper] = 20 Do not need to change.
Those three instructions set each mapper will only deal with 100MB size of task.
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat; (Combine the small size file to a larger file that is close to 100MB)

8. How to check the datasize of a SQL benchmark

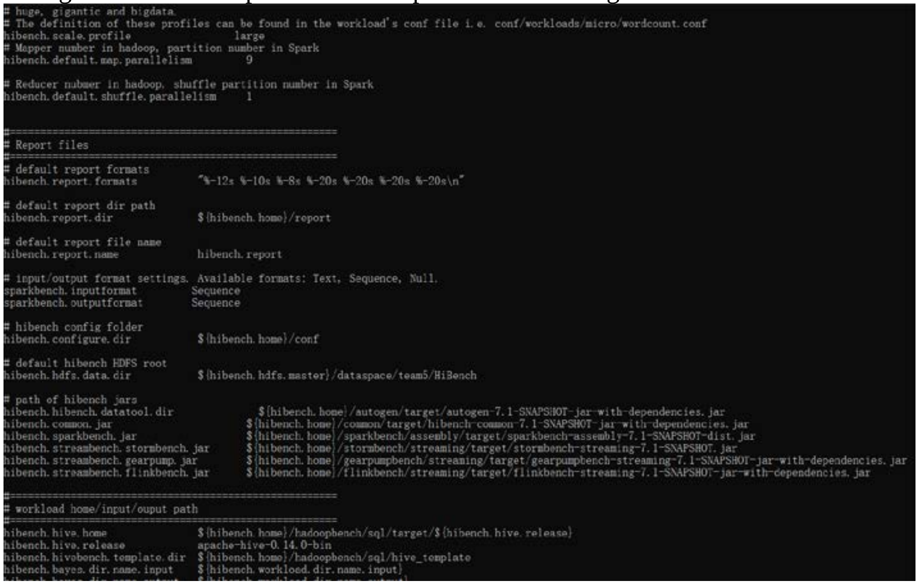
```
hdfs dfs -du -s -h hdfs://cuhkcluster/dataspace/team5/Hibench/Join
hdfs dfs -du -s -h hdfs://cuhkcluster/dataspace/team5/Hibench/Scan
hdfs dfs -du -s -h hdfs://cuhkcluster/dataspace/team5/Hibench/Aggregation
```

9. An example of the bench.log for Join SQL prepare.sh

```
Job Counters
        Launched map tasks=8
        Launched reduce tasks=4
        Other local map tasks=8
        Total time spent by all maps in occupied slots (ms)=235593
        Total time spent by all reduces in occupied slots (ms)=66330
        Total time spent by all map tasks (ms)=78531
        Total time spent by all reduce tasks (ms)=11055
        Total vcore-milliseconds taken by all map tasks=78531
        Total vcore-milliseconds taken by all reduce tasks=11055
        Total megabyte-milliseconds taken by all map tasks=241247232
        Total megabyte-milliseconds taken by all reduce tasks=67921920
```

10. Job configuration of Sort

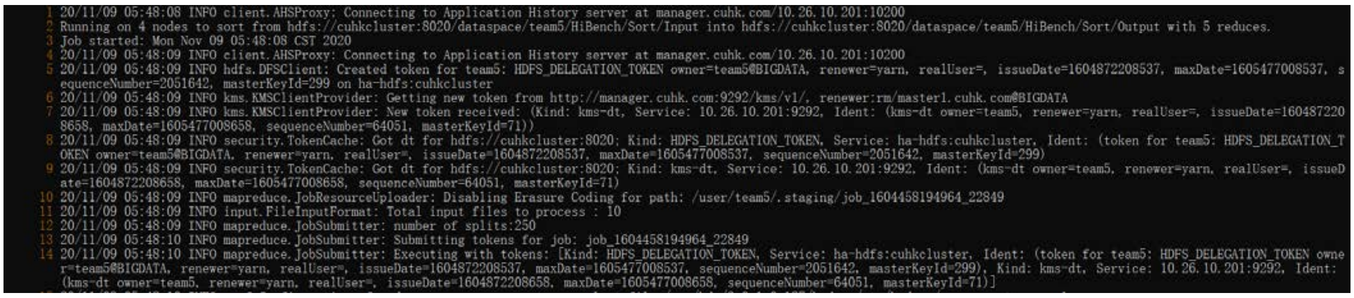
In this project, we modify the file of hibench.conf to control the size of data and the number of mapper and reducer. It offer the configuration to both Spark and Hadoop. We should change the data dir into our team name at first to complete the configuration.



```
# huge, gigantic and bigdata.
# The definition of these profiles can be found in the workload's conf file i.e. conf/workloads/micro/wordcount.conf
hibench.scale.profile                 large
# Mapper number in hadoop, partition number in Spark
hibench.default.map.parallelism         9

# Reducer nubmer in hadoop, shuffle partition number in Spark
hibench.default.shuffle.parallelism     1


#======================================================
# Report files
#======================================================
# default report formats
hibench.report.formats          "%-12s %-10s %-8s %-20s %-20s %-20s %-20s\n"

# default report dir path
hibench.report.dir              ${hibench.home}/report

# default report file name
hibench.report.name             hibench.report

# input/output format settings. Available formats: Text, Sequence, Null.
sparkbench.inputformat         Sequence
sparkbench.outputformat        Sequence

# hibench config folder
hibench.configure.dir           ${hibench.home}/conf

# default hibench HDFS root
hibench.hdfs.data.dir           ${hibench.hdfs.master}/dataspace/team5/HiBench

# path of hibench jars
hibench.hibench.datatool.dir              ${hibench.home}/autogen/target/autogen-7.1-SNAPSHOT-jar-with-dependencies.jar
hibench.common.jar                      ${hibench.home}/common/target/hibench-common-7.1-SNAPSHOT-jar-with-dependencies.jar
hibench.sparkbench.jar                  ${hibench.home}/sparkbench/assembly/target/sparkbench-assembly-7.1-SNAPSHOT-dist.jar
hibench.streambench.stormbench.jar      ${hibench.home}/stormbench/streaming/target/stormbench-streaming-7.1-SNAPSHOT.jar
hibench.streambench.gearpump.jar        ${hibench.home}/gearpumpbench/streaming/target/gearpumpbench-streaming-7.1-SNAPSHOT-jar-with-dependencies.jar
hibench.streambench.flinkbench.jar      ${hibench.home}/flinkbench/streaming/target/flinkbench-streaming-7.1-SNAPSHOT-jar-with-dependencies.jar

#======================================================
# workload home/input/ouput path
#======================================================
hibench.hive.home               ${hibench.home}/hadoopbench/sql/target/${hibench.hive.release}
hibench.hive.release            apache-hive-0.14.0-bin
hibench.hivebench.template.dir  ${hibench.home}/hadoopbench/sql/hive_template
hibench.bayes.dir.name.input    ${hibench.workload.dir.name.input}
```

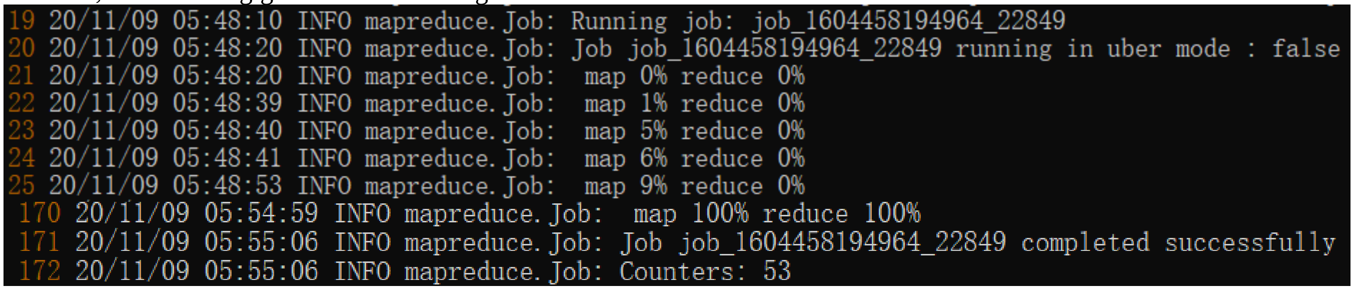
11. Results Analysis of Sort

The understanding of bench.log: after finish the run.sh procedure, we can see the report in the corresponding bench.log file. For example, after running the prepare.sh and run.sh of sort of Hadoop. We could find the corresponding bench.log under report/sort/Hadoop. Then we could use vi bench.log to view.



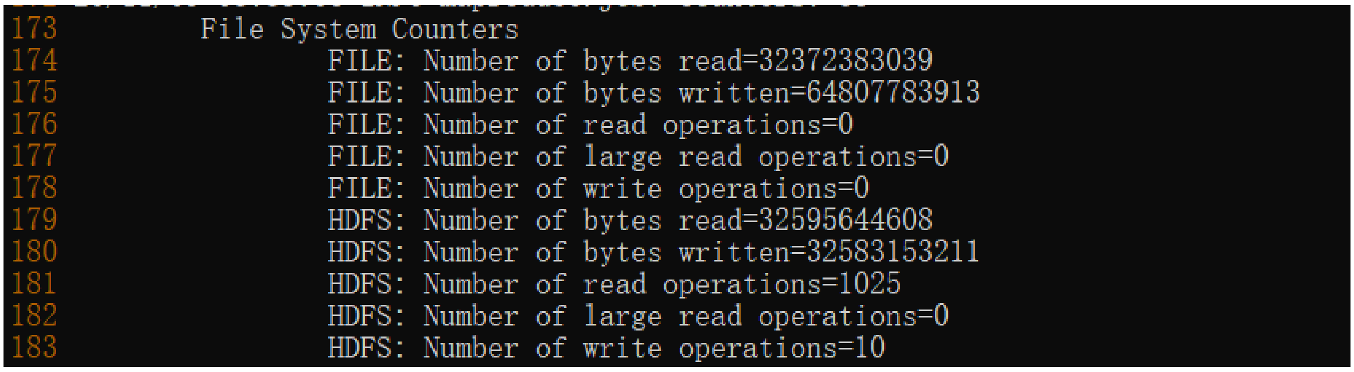
```
20/11/09 05:48:08 INFO client.AHSProxy: Connecting to Application History server at manager.cuhk.com/10.26.10.201:10200
Running on 4 nodes to sort from hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Sort/Input into hdfs://cuhkcluster:8020/dataspace/team5/HiBench/Sort/Output with 5 reduces.
Job started: Mon Nov 09 05:48:08 CST 2020
20/11/09 05:48:09 INFO client.AHSProxy: Connecting to Application History server at manager.cuhk.com/10.26.10.201:10200
20/11/09 05:48:09 INFO hdfs.DFSClient: Created token for team5: HDFS_DELEGATION_TOKEN owner=team5@BIGDATA, renewer=yarn, realUser=, issueDate=1604872208537, maxDate=1605477008537, s
equenceNumber=2051642, masterKeyId=299 on ha-hdfs:cuhkcluster
20/11/09 05:48:09 INFO kms.KMSClientProvider: Getting new token from http://manager.cuhk.com:9292/kms/v1/, renewer:rm/master1.cuhk.com@BIGDATA
20/11/09 05:48:09 INFO kms.KMSClientProvider: New token received: (Kind: kms-dt, Service: 10.26.10.201:9292, Ident: (kms-dt owner=team5, renewer=yarn, realUser=, issueDate=160487220
8658, maxDate=1605477008658, sequenceNumber=64051, masterKeyId=71))
20/11/09 05:48:09 INFO security.TokenCache: Got dt for hdfs://cuhkcluster:8020; Kind: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:cuhkcluster, Ident: (token for team5: HDFS_DELEGATION_T
OKEN owner=team5@BIGDATA, renewer=yarn, realUser=, issueDate=1604872208537, maxDate=1605477008537, sequenceNumber=2051642, masterKeyId=299)
20/11/09 05:48:09 INFO security.TokenCache: Got dt for hdfs://cuhkcluster:8020; Kind: kms-dt, Service: 10.26.10.201:9292, Ident: (kms-dt owner=team5, renewer=yarn, realUser=, issueD
ate=1604872208658, maxDate=1605477008658, sequenceNumber=64051, masterKeyId=71)
20/11/09 05:48:09 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /user/team5/.staging/job_1604458194964_22849
20/11/09 05:48:09 INFO input.FileInputFormat: Total input files to process : 10
20/11/09 05:48:09 INFO mapreduce.JobSubmitter: number of splits:250
20/11/09 05:48:10 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1604458194964_22849
20/11/09 05:48:10 INFO mapreduce.JobSubmitter: Executing with tokens: [Kind: HDFS_DELEGATION_TOKEN, Service: ha-hdfs:cuhkcluster, Ident: (token for team5: HDFS_DELEGATION_TOKEN owne
r=team5@BIGDATA, renewer=yarn, realUser=, issueDate=1604872208537, maxDate=1605477008537, sequenceNumber=2051642, masterKeyId=299), Kind: kms-dt, Service: 10.26.10.201:9292, Ident:
(kms-dt owner=team5, renewer=yarn, realUser=, issueDate=1604872208658, maxDate=1605477008658, sequenceNumber=64051, masterKeyId=71)]
```

In the first part, we could see the number of nodes for the job, the number of reducers, the start time of job. Then we will submit the job and get the job tokens. The job will be split into splits. For the tiny size of file, the number of splits is 10, however, it is 250 for gigantic and 30 for huge.



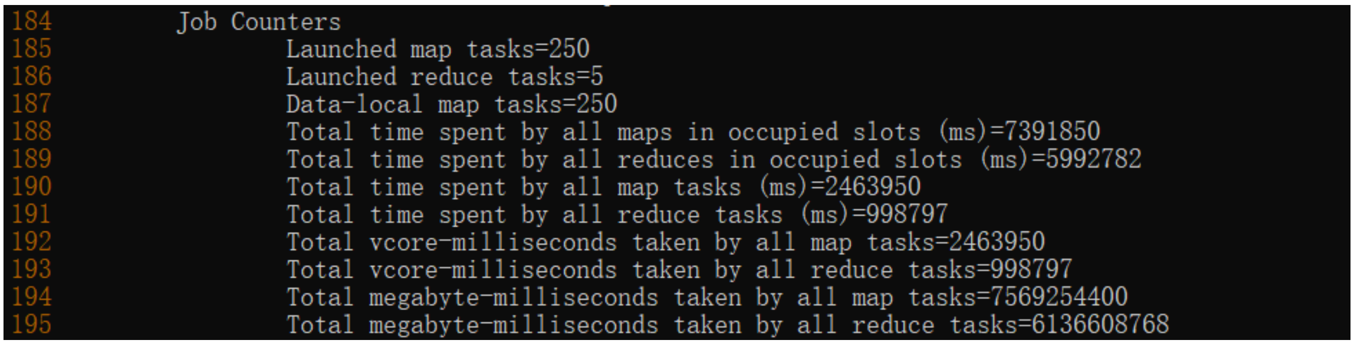
```
20/11/09 05:48:10 INFO mapreduce.Job: Running job: job_1604458194964_22849
20/11/09 05:48:20 INFO mapreduce.Job: Job job_1604458194964_22849 running in uber mode : false
20/11/09 05:48:20 INFO mapreduce.Job:  map 0% reduce 0%
20/11/09 05:48:39 INFO mapreduce.Job:  map 1% reduce 0%
20/11/09 05:48:40 INFO mapreduce.Job:  map 5% reduce 0%
20/11/09 05:48:41 INFO mapreduce.Job:  map 6% reduce 0%
20/11/09 05:48:53 INFO mapreduce.Job:  map 9% reduce 0%
20/11/09 05:54:59 INFO mapreduce.Job:  map 100% reduce 100%
20/11/09 05:55:06 INFO mapreduce.Job: Job job_1604458194964_22849 completed successfully
20/11/09 05:55:06 INFO mapreduce.Job: Counters: 53
```

Then the mapper and reducer will work, and we could see their rate of progress.



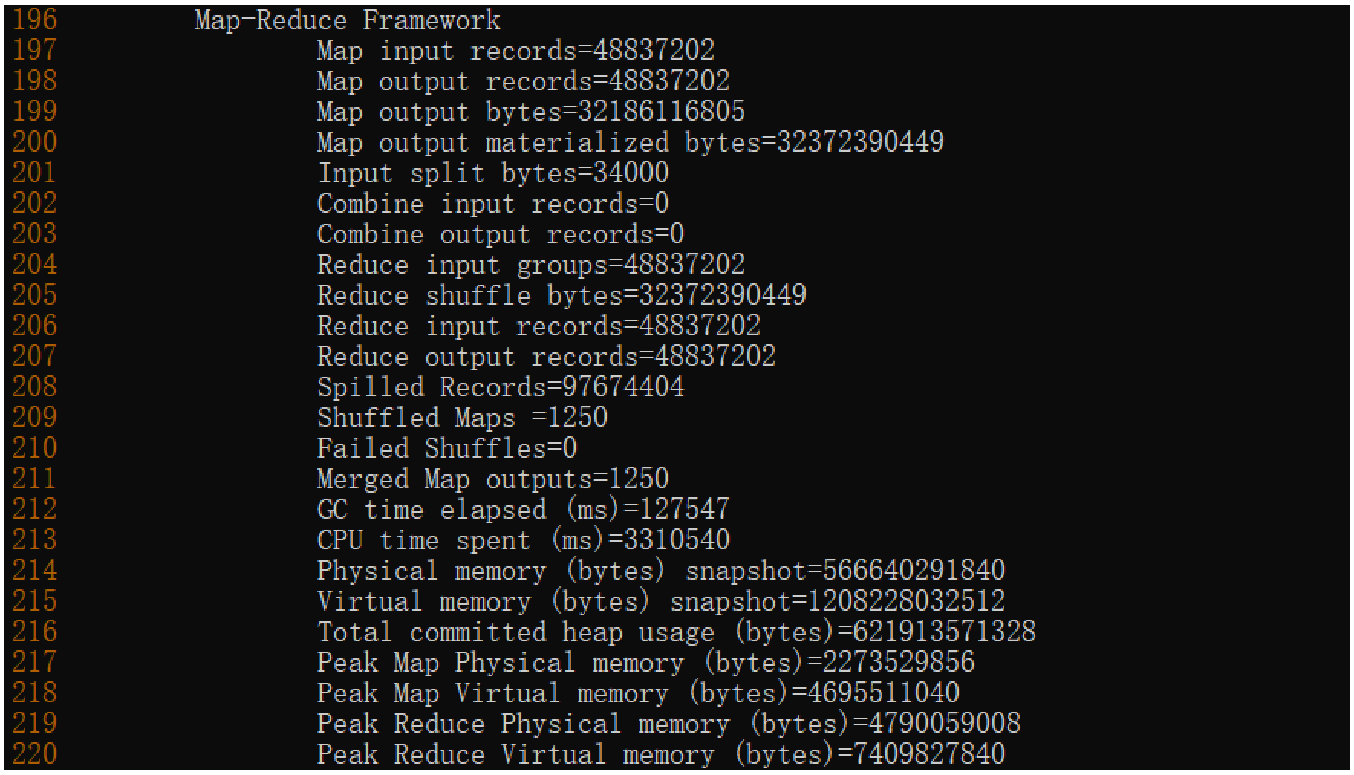
```
        File System Counters
                FILE: Number of bytes read=32372383039
                FILE: Number of bytes written=64807783913
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=32595644608
                HDFS: Number of bytes written=32583153211
                HDFS: Number of read operations=1025
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=10
```

In the File System Counters, it prints out the result of the counters of File System. The number of bytes for read and written in File and HDFS can be seen.



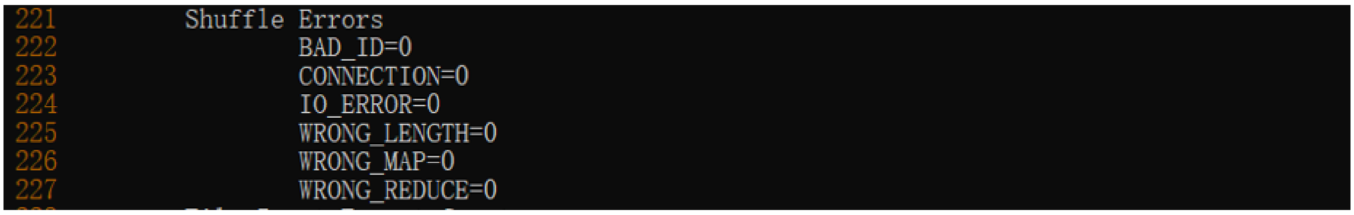
```
        Job Counters
                Launched map tasks=250
                Launched reduce tasks=5
                Data-local map tasks=250
                Total time spent by all maps in occupied slots (ms)=7391850
                Total time spent by all reduces in occupied slots (ms)=5992782
                Total time spent by all map tasks (ms)=2463950
                Total time spent by all reduce tasks (ms)=998797
                Total vcore-milliseconds taken by all map tasks=2463950
                Total vcore-milliseconds taken by all reduce tasks=998797
                Total megabyte-milliseconds taken by all map tasks=7569254400
                Total megabyte-milliseconds taken by all reduce tasks=6136608768
```

In the job counter, the running time of maps tasks and reduce tasks will be counted.



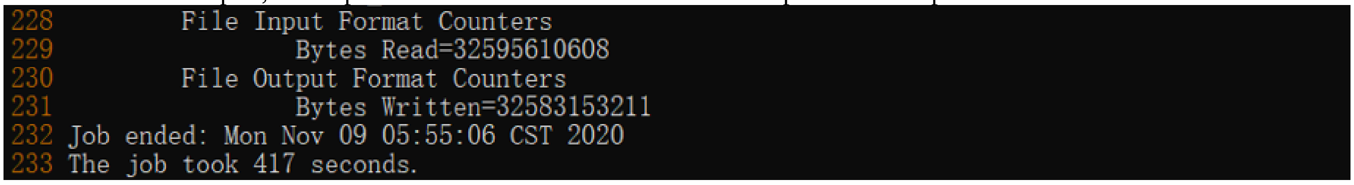
```
        Map-Reduce Framework
                Map input records=48837202
                Map output records=48837202
                Map output bytes=32186116805
                Map output materialized bytes=32372390449
                Input split bytes=34000
                Combine input records=0
                Combine output records=0
                Reduce input groups=48837202
                Reduce shuffle bytes=32372390449
                Reduce input records=48837202
                Reduce output records=48837202
                Spilled Records=97674404
                Shuffled Maps =1250
                Failed Shuffles=0
                Merged Map outputs=1250
                GC time elapsed (ms)=127547
                CPU time spent (ms)=3310540
                Physical memory (bytes) snapshot=566640291840
                Virtual memory (bytes) snapshot=1208228032512
                Total committed heap usage (bytes)=621913571328
                Peak Map Physical memory (bytes)=2273529856
                Peak Map Virtual memory (bytes)=4695511040
                Peak Reduce Physical memory (bytes)=4790059008
                Peak Reduce Virtual memory (bytes)=7409827840
```

In the Map-Reduce Framework, the data flow going through mapper and reducer will be counted. Also, the memory peak physical and virtual memory taken by mapper and reducer can be seen. We could adjust the number of mapper and reducer due to this number.



```
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
```

In the Shuffle Error part, it will print the number of error occurred in the process of map and reduce.



```
        File Input Format Counters
                Bytes Read=32595610608
        File Output Format Counters
                Bytes Written=32583153211
Job ended: Mon Nov 09 05:55:06 CST 2020
The job took 417 seconds.
```

In the last part, it prints the running and end time of job and the end time of job and the size for read and write in bytes.

VII. References

[1] Ivanov, T., Niemann, R., Izberovic, S., Rosselli, M., Tolle, K., & Zicari, R. V. (2015). Performance Evaluation of Enterprise Big Data Platforms with HiBench. 2015 IEEE Trustcom/BigDataSE/ISPA.

[2] Performance Monitoring, Testing and Optimizing Hadoop-MapReduce Job using Hadoop Counters http://hadoopmania.blogspot.com/2015/10/performance-monitoring-testing-and.html

[3] Samadi, Y., Zbakh, M., & Tadonki, C. (2016, May). Comparative study between Hadoop and Spark based on Hibench benchmarks. In *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)* (pp. 267-275). IEEE.

[4] Hwang, K., & Chen, M. (2017). Big-data analytics for cloud, IoT and cognitive computing. John Wiley & Sons.

[5] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[6] K. Elissa, “Title of paper if known,” unpublished.

[7] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[8] White .T. Hadoop The Definitve Mode https://learning.oreilly.com/library/view/hadoop-the-definitive/97814919/

[9] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., & Stonebraker, M. (2009, June). A comparison of approaches to large-scale data analysis. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (pp. 165-178).Hwang, K., & Chen, M. (2017). Big-data analytics for cloud, IoT and cognitive computing. John Wiley & Sons.

[10] Apache Hive Tutorial – Design https://cwiki.apache.org/confluence/display/Hive/Design

[11] Huang S, Huang J, Liu Y, et al. Hibench: A representative and comprehensive hadoop benchmark suite[C]//Proc. ICDE Workshops. 2010: 41-51.What is Spark https://databricks.com/spark/about

[12] Apache Spark – Introduction https://www.tutorialspoint.com/apache_spark/apache_spark_introduction.htm