Hadoop Performance Study

Team 6 : 20131079 Beomsoo Kim, 20131655 Gheejung Hong, 20141440 Seungwon Lee

CSE48001, Ulsan National Institute of Science and Technology

{ kbs9409, ghdrlwjd12, aldlfkahs }@unist.ac.kr

ABSTRACT

The objective of this experiments is to find several factors that affects the Hadoop performance sensitively. Thus, we studied 3 major factors and lots of other factors. We picked out some of them that have more impact on performance compared to the others. Our Hadoop system is installed on 3 VMs in UNIST server, which each VM has same amount of CPU, memory and disk. In UNIST server, hosts and storages are separated, which means they use network to communicate with each other. Since tuning Hadoop is very different according to the environments, our results may not be applied to other environments.

INTRODUCTION

We want to find what factors affect more about performance but there are so many factors and as we mentioned above, tuning Hadoop can be different according to the environments. However, there definitely exists factors that have a big impact on performance in the most environment case. Those factors are given by mandatory tasks in this project. That is, replication factor, HDFS block size, and CPU/memory configurations.

Replication factor determines the number of copy of data. It has to do with locality. Naturally, we pursue node locality which is the most effective locality. HDFS block size factor literally determines the size of block stored in HDFS. It could affect the number of tasks. CPU and memory configuration is to allocate how much resources we will give for each item(node manager, mapper, reducer, etc..).

In addition to major factors, we also studied lots other factors. Especially, since hosts and storages are separated in our environment, we focused on reducing the number of I/O. Following is the factor we have studied; I/O file buffer size, sort I/O buffer size, the number of sort tasks, input buffer of shuffle stage, input buffer of reduce stage, the number of parallel executions of shuffle stage, the number of workers(slave nodes). Among them, we chose two factors that have the most sensitive effect on performance. That is, I/O file buffer size and shuffle input buffer size are those.

We used three jobs which is provided by sample for benchmark; wordcount, sort, and grep.

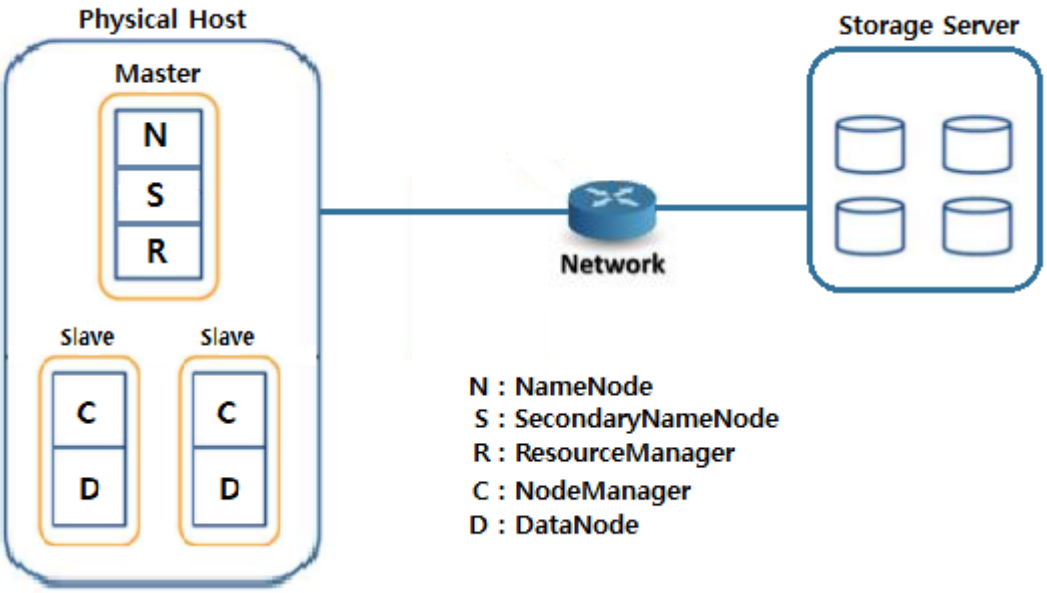
Wordcount receives text inputs and counts how many each word is in the text files. Each mapper of wordcount takes a line as input and breaks it into words. It then emits a key/value pair of the word and each reducer sums the counts for each word and emits a single key/value with the word and sum. Sort literally sorts the inputs. The input files of sort job should be sequence files where the keys and values are BytesWritable. The mapper of sort is the predefined IdentityMapper and the reducer is the predefined IdentityReducer, both of which just pass their inputs directly to the output. Grep searched specific word given by parameter as a regular expression. The mapper of grep counts how many times a matching string occured and the reducer sorts matching strings by their frequency and stores the output in a single output file. All of three jobs are map-intensive tasks, which means that the total amount mapper works is bigger than amount of reducer. However, they have little difference. Among them, sort relatively has more reduce job than the others. Especially, grep does very short time for reduce. Wordcount is between of them.

We did our experiments multiple times at different time for reliability. The result is the average of them.

CLUSTER SETUP

We used Hadoop 2.7.6 for our study. Our hosts and storage servers are separated from each other in our environment. Below is our default setup. At the experiments part, we will mention which property we modified in this default setup.

Hadoop cluster used in our study is composed of 3 VMs, a master VM that runs the ResourceManager, NameNode and SecondaryNameNode, and 2 workers VMs, each of which runs a NodeManager and a DataNode. We visualizes the topology of our cluster setup in Fig 1.



↑ (Fig 1) Cluster Topology

All of three VMs has 3 core, 4GB memory and 20GB for disk. In each worker VM, 3GB of memory and 8 cores are assigned to a NodeManager. For map and reduce tasks, total 1GB is assigned for them and 512MB and 1 core are configured to run each of them. Input buffer percent of Shuffle is assigned to be 0.7. Block size of HDFS is 128MB and it has replication factor 1. Buffer size of I/O is 4096KB.

As we mentioned in introduction, we used three jobs for benchmarking. To run those jobs, we give 5GB files as an input for each test. Especially, we generated random text files for wordcount and grep by using randomtextwriter provided by Hadoop. Also, we generated random files for sort by using randomwriter provided by Hadoop.

Other not mentioned configurations are set to be default value of Hadoop 2.7.6. Below figures are captured of four configuration files.

```
<configuration>
        <property>
                <name>fs.defaultFS</name>
                <value>hdfs://master:9000</value>
        </property>
        <property>
                <name>io.file.buffer.size</name>
                <value>4096</value>
        </property>


</configuration>
```

↑ (Fig 2) core-site.xml

```
<configuration>
    <property>
            <name>dfs.namenode.name.dir</name>
            <value>/home/hadoop/hadoop/data/nameNode</value>
    </property>
    <property>
            <name>dfs.datanode.data.dir</name>
            <value>/home/hadoop/hadoop/data/dataNode</value>
    </property>
    <property>
            <name>dfs.replication</name>
            <value>1</value>
    </property>
    <property>
            <name>dfs.blocksize</name>
            <value>128m</value>
    </property>
</configuration>
```

↑ (Fig 3) hdfs-site.xml

```
<configuration>
    <property>
            <name>yarn.nodemanager.aux-services</name>
            <value>mapreduce_shuffle</value>
    </property>
    <property>
            <name>yarn.resourcemanager.hostname</name>
            <value>master</value>
    </property>
    <property>
            <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
            <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>

    <property>
            <name>yarn.nodemanager.resource.cpu-vcores</name>
            <value>8</value>
    </property>



<!-- RAM Configuratopn -->
        <property>
                <name>yarn.nodemanager.resource.memory-mb</name>
                <value>3072</value>
        </property>
        <property>
                <name>yarn.scheduler.maximum-allocation-mb</name>
                <value>3072</value>
        </property>
        <property>
                <name>yarn.scheduler.minimum-allocation-mb</name>
                <value>256</value>
        </property>
        <property>
                <name>yarn.nodemanager.pmem-check-enabled</name>
                <value>false</value>
        </property>
        <property>
                <name>yarn.nodemanager.vmem-check-enabled</name>
                <value>false</value>
        </property>
</configuration>
```

↑ (Fig 4) yarn-site.xml

```
<configuration>
    <property>
            <name>mapreduce.framework.name</name>
            <value>yarn</value>
    </property>
    <property>
            <name>mapreduce.reduce.shuffle.input.buffer.percent</name>
            <value>0.7</value>
    </property>
    <property>
            <name>mapreduce.reduce.shuffle.parallelcopies</name>
            <value>5</value>
    </property>
    <property>
            <name>mapreduce.reduce.cpu.vcores</name>
            <value>1</value>
    </property>
    <property>
            <name>mapreduce.map.cpu.vcores</name>
            <value>1</value>
    </property>
    <property>
            <name>mapreduce.task.io.sort.mb</name>
            <value>100</value>
    </property>


<!-- RAM Configuration -->

        <property>
                <name>yarn.app.mapreduce.am.resource.mb</name>
                <value>1024</value>
        </property>

        <property>
                <name>mapreduce.map.memory.mb</name>
                <value>512</value>
        </property>

        <property>
                <name>mapreduce.reduce.memory.mb</name>
                <value>512</value>
        </property>


</configuration>
```

↑ (Fig 5) mapred-site.xml

EXPERIMENTS

We will start from 3 mandatory tasks and continue to additional tasks.

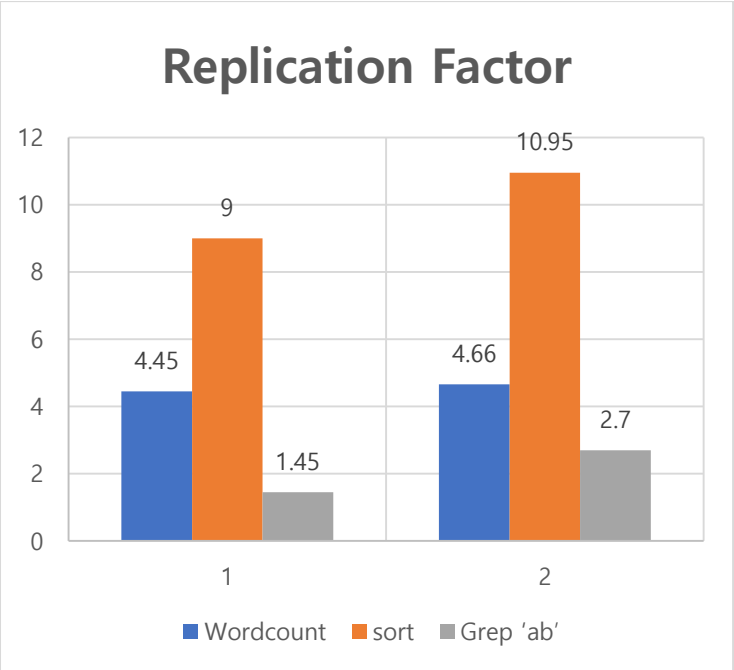
Mandatory Tasks

1) Replication Factor

First is replication factor. Replication factor determines the number of copy of data. For example, if the replication factor is 2, the system will make and store two outputs for same data. Usually, replication factor is closely related to locality. There are 3 main localities, that is, node locality, rack locality and off-rack locality. Node locality is the most preferred scenario as it can reduce the data transfer time. If there are many copies in storage, a node could select data which is the closest from it. However, there is also disadvantage. If replication is large, Hadoop should write that much of data. Since we have only two DataNodes, we can only do two cases, 1 and 2. Graph 1 is our result. The configuration is <dfs.replication> in hdfs-site.xml file.

```
<property>
        <name>dfs.replication</name>
        <value>1</value>
</property>
```

```
<property>
        <name>dfs.replication</name>
        <value>2</value>
</property>
```



↑(Graph 1) Replication Factor

Interestingly, the execution time is slower rather than faster for all three jobs. It would be because overhead caused by writing twice exceed the benefit from locality. Actually, in real world, even in huge Hadoop cluster environment usually set replication factor as no more than 3. Since we only

have two DataNodes, it is natural that overhead is bigger than the benefit.

2) **HDFS Block Size**

Next is HDFS block size. It literally determines the block size of HDFS. If block size becomes large, the total number of tasks will reduce, which means that it can reduces initialization overhead. Thus, generally, it will be faster if the block size gets larger. However, there is trade-off. If block size becomes too large, it damages task scheduling granularity. That is, it will be harder to load balancing and cause map-shuffle phase overlap and job fair problem. See our result in Graph 2. The configuration is <dfs.blocksize> in hdfs-site.xml file.

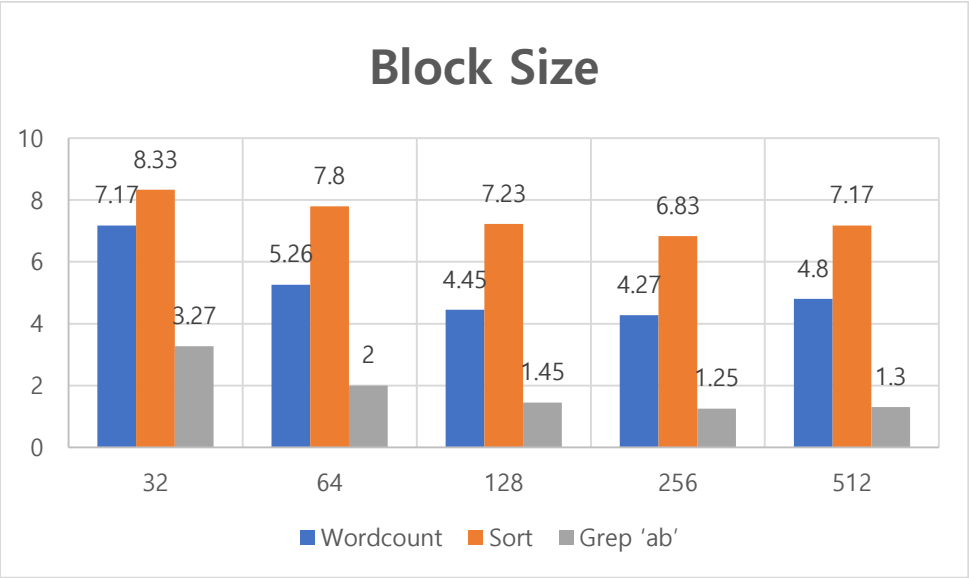
```
<property>
        <name>dfs.blocksize</name>
        <value>32m</value>
</property>
```

```
<property>
        <name>dfs.blocksize</name>
        <value>64m</value>
</property>
```

```
<property>
        <name>dfs.blocksize</name>
        <value>128m</value>
</property>
```

```
<property>
        <name>dfs.blocksize</name>
        <value>256m</value>
</property>
```

```
<property>
        <name>dfs.blocksize</name>
        <value>512m</value>
</property>
```



↑ (Graph 2) HDFS Block Size

As we expected, performance becomes better as block size gets larger. It reduces the initialization overhead so that total execution time also reduces. However, the execution time increases again at some point. That is the trade-off we mentioned. In our environment, 256MB block size shows the best performance. If it is larger than 256MB, the performance gets worse. It's because too small

number of tasks may harm task scheduling granularity.

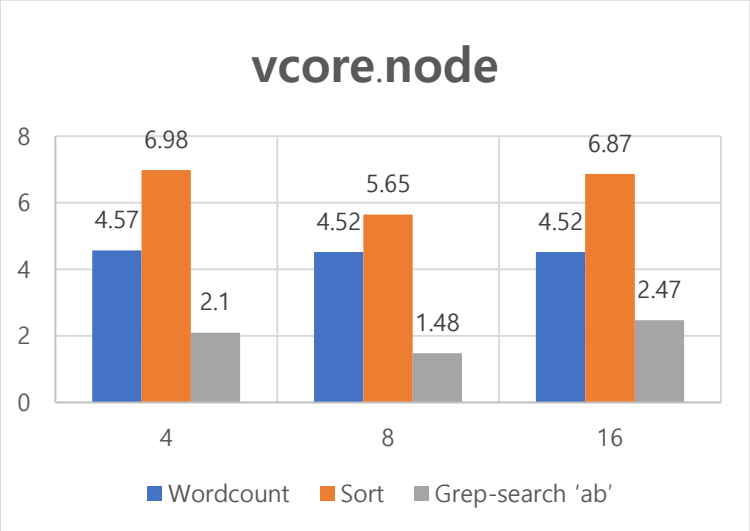
3) CPU and Memory Configuration

Last major factor is CPU/memory configuration. We considered CPU first not the physical core but virtual core. If there are many cores, it will have advantage of executing the large computation. However, we also have to consider synchronization and communication overhead between cores. Considering that trade-off, see the result in Graph 3. We first modified the number of vcores per node. The configuration is <yarn.nodemanager.resource.cpu-vcores> in yarn-site.xml file.

```
<property>
        <name>yarn.nodemanager.resource.cpu-vcores</name>
        <value>4</value>
</property>
```

```
<property>
        <name>yarn.nodemanager.resource.cpu-vcores</name>
        <value>8</value>
</property>
```

```
<property>
        <name>yarn.nodemanager.resource.cpu-vcores</name>
        <value>16</value>
</property>
```



↑(Graph 3) The Number of Vcores Per Node

The execution time gets faster as core becomes larger. When vcores are 8, it shows the best performance. However, when it becomes over than 8 cores, the performance is getting worse. It's because benefit of larger number of cores is smaller than the overhead of synchronization and communication of multiple cores.

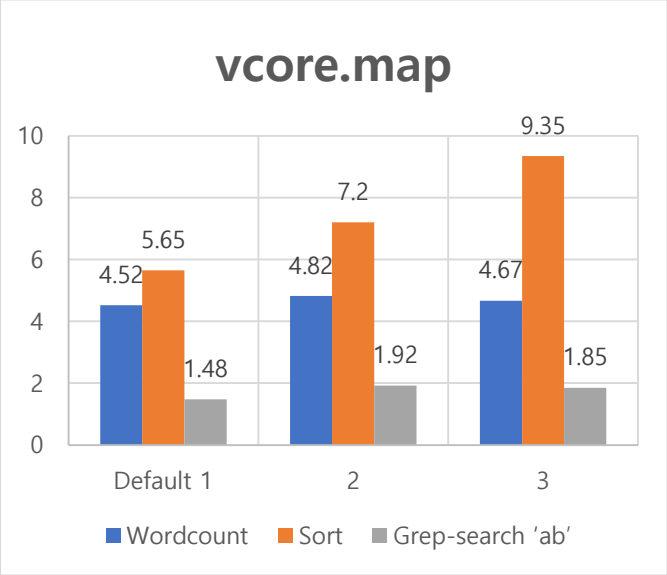
Next, we tried to increase the number of vcores of each mapper and reducer separately. See mapper case first in Graph 4. The configuration is <mapreduce.map.cpu.vcores> in mapred-site.xml

file.

```
<property>
        <name>mapreduce.map.cpu.vcores</name>
        <value>1</value>
</property>
```

```
<property>
        <name>mapreduce.map.cpu.vcores</name>
        <value>2</value>
</property>
```

```
<property>
        <name>mapreduce.map.cpu.vcores</name>
        <value>3</value>
</property>
```



↑(Graph 4) The Number of Vcores Per Mapper

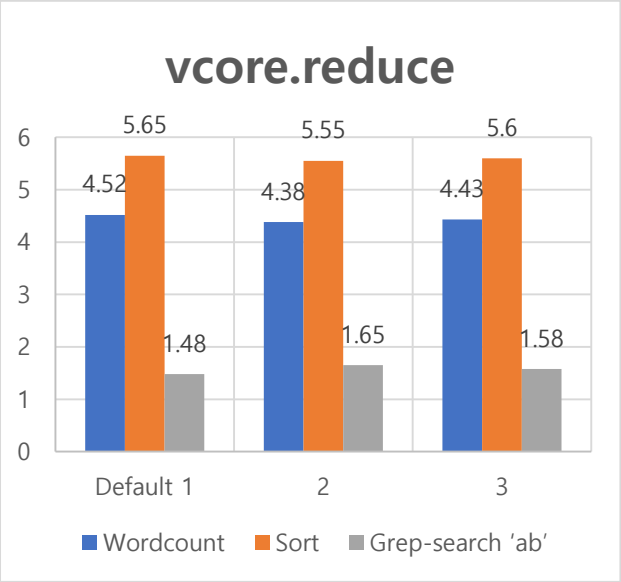
Interestingly, the performance gets worse as the number of vcores of mapper increases. It's because the all of three three jobs are not cpu-intensive job. That is, there are almost no benefits from multiple cores but only the disadvantages of multiple cores' overhead such as synchronization or communication as we mentioned earlier.

Next is the number of vcores of reducer. The configuration is <mapreduce.reduce.cpu.vcores> in mapred-site.xml file.

```
<property>
        <name>mapreduce.reduce.cpu.vcores</name>
        <value>1</value>
</property>
```

```
<property>
        <name>mapreduce.reduce.cpu.vcores</name>
        <value>2</value>
</property>
```

```
<property>
        <name>mapreduce.reduce.cpu.vcores</name>
        <value>3</value>
</property>
```



↑(Graph 5) The Number of Vcores Per Reducer

Unlike the mapper case, the result shows similar performance regardless of the number of vcores. It's because all three jobs are mapper-intensive job as we discussed in Introduction. Thus, it seems that performance change is somewhat feeble compared to the mapper case.

Lastly, we considered memory configuration. Unlike the CPU case it's not virtual but physical memory. The progress is similar with CPU case, start from total memory per node to per mapper and reducer. If memory becomes large, hosts can keep more data than before which means that the number of transferring to buffer reduces and thus the number of disk I/O also reduces. Therefore, the performance will be better. When we modified the total memory per node, we also modified associative other memory configuration such as maximum memory per mapper or reducer. It's because it's important to keep entire memory ratio. The configuration is <yarn.nodemanager.resource.memory-mb> in yarn-site.xml file and <mapreduce.map.memory.mb>, <mapreduce.reduce.memory.mb> in mapred-site.xml file

```
<property>
        <name>yarn.nodemanager.resource.memory-mb</name>
        <value>768</value>
</property>
```

```
<property>
        <name>yarn.nodemanager.resource.memory-mb</name>
        <value>1536</value>
</property>
```

```
<property>
        <name>yarn.nodemanager.resource.memory-mb</name>
        <value>3072</value>
</property>
```

```
<property>
        <name>mapreduce.map.memory.mb</name>
        <value>64</value>
</property>
```

```
<property>
        <name>mapreduce.map.memory.mb</name>
        <value>128</value>
</property>
```

```
<property>
        <name>mapreduce.map.memory.mb</name>
        <value>256</value>
</property>
```

```
<property>
        <name>mapreduce.map.memory.mb</name>
        <value>512</value>
</property>
```

```
<property>
        <name>mapreduce.map.memory.mb</name>
        <value>1024</value>
</property>
```

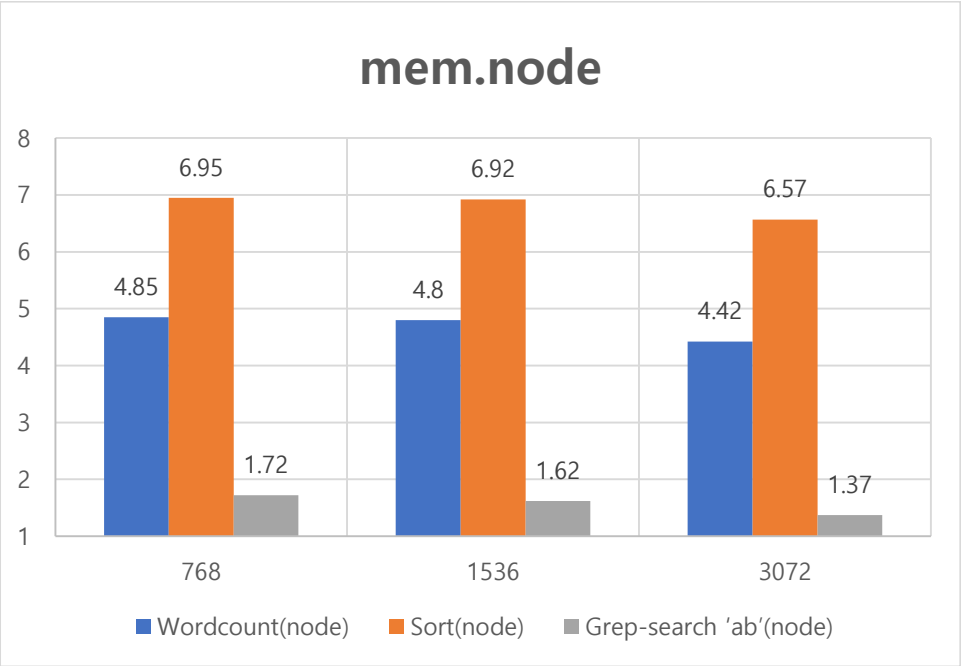
```
<property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>64</value>
</property>
```

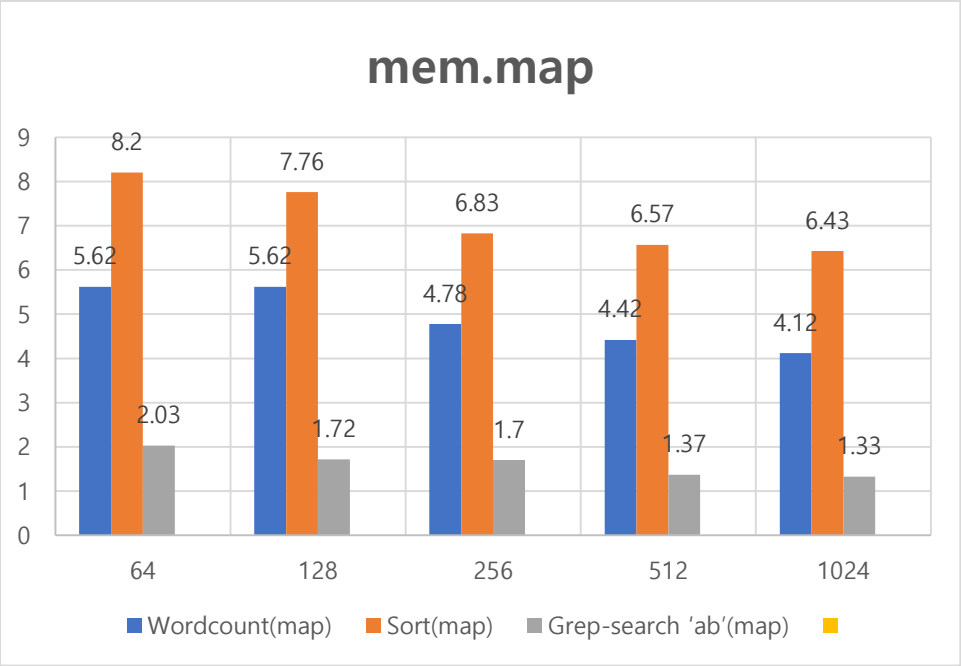
```
<property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>128</value>
</property>
```

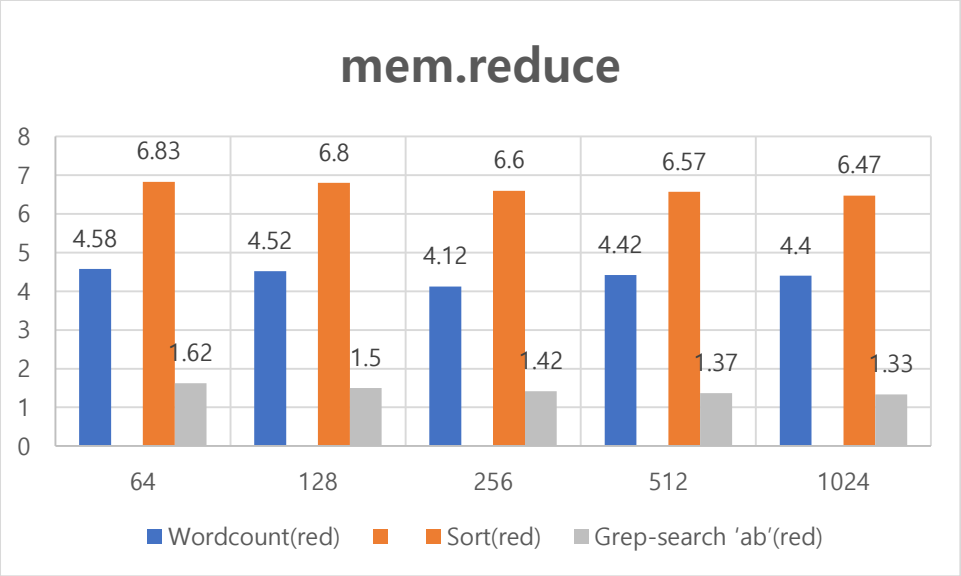
```
<property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>256</value>
</property>
```

```
<property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>512</value>
</property>
```

```
<property>
        <name>mapreduce.reduce.memory.mb</name>
        <value>1024</value>
</property>
```







↑(Graph 6) Memory Configuration

Obviously, all three cases show better performance when they are assigned more memory. They are available to keep more data in own memory, which reduces the number of disk I/O ultimately. However, reduce case does not show big performance improvement than the other two cases. The reason is same with CPU case. Since all three jobs are mapper-intensive job, changing reducer's configuration is hard to expect a big improvement in performance.

Additional Tasks

Motivation : Since the environment Hadoop cluster we studied has separated hosts and storage servers, it's obvious that all disk I/O is equivalent to network I/O which is very expensive operation. Thus, we thought that factor related with I/O will have more effect on performance so we focused on reducing the number of disk I/O to get sensitive improvement of performance.

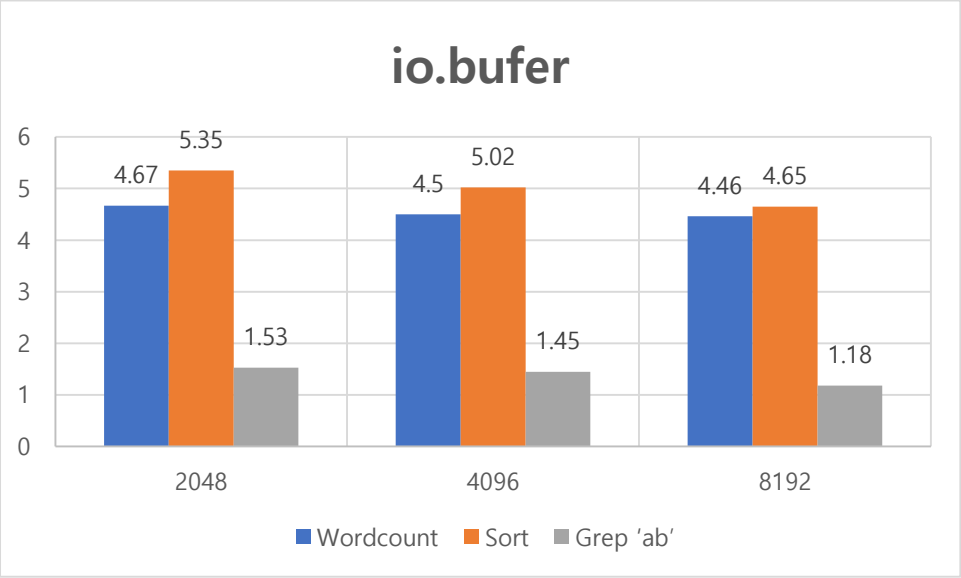
1) I/O File Buffer Size

I/O file buffer size determines the buffer size of read/write operation about HDFS. In our environment, accessing the HDFS storage costs network I/O. If buffer size is large, it can read or write larger data at once, which results in reducing the number of I/O. The configuration <io.file.buffer.size> in core-site.xml file.

```
<property>
        <name>io.file.buffer.size</name>
        <value>2048</value>
</property>
```

```
<property>
        <name>io.file.buffer.size</name>
        <value>4096</value>
</property>
```

```
<property>
        <name>io.file.buffer.size</name>
        <value>8192</value>
</property>
```



↑ (Graph 7) I/O Buffer Size

As we expected, performance is better when buffer size is larger. It really reduces the number of network I/O which results in reducing overall execution time.

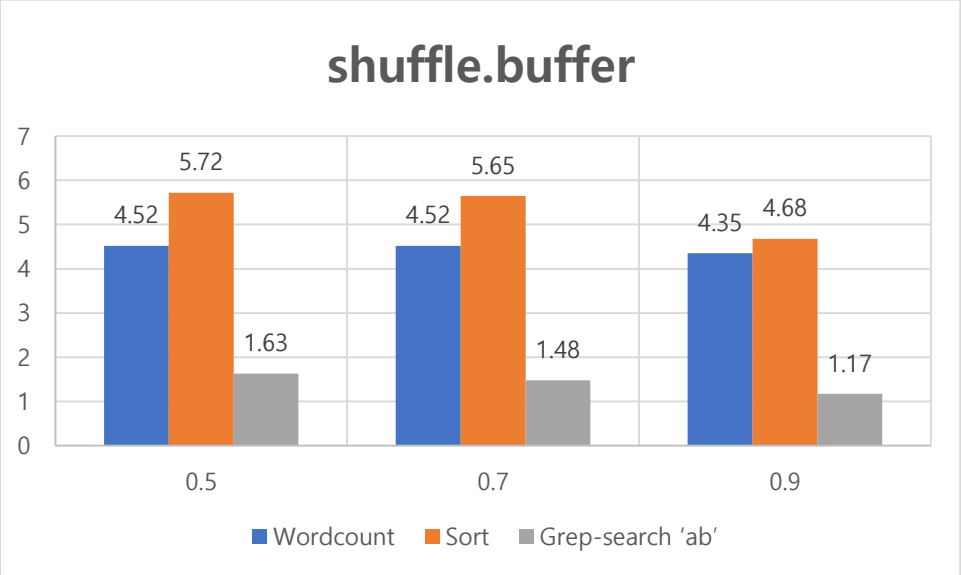
2) Shuffle Input Buffer Size

Buffer size of shuffle input determines the buffer size that store the data from map outputs to shuffle inputs. In the configuration file, it represented as ratio. For example, if the ratio is 0.7, data in buffer will be transferred to disk when buffer is filled by 70%. Thus, if we set this value to large, the number of disk I/O will reduce. The configuration is <mapreduce.reduce.shuffle.input.buffer.percent> in marped-site.xml file.

```
<property>
        <name>mapreduce.reduce.shuffle.input.buffer.percent</name>
        <value>0.5</value>
</property>
```

```
<property>
        <name>mapreduce.reduce.shuffle.input.buffer.percent</name>
        <value>0.7</value>
</property>
```

```
<property>
        <name>mapreduce.reduce.shuffle.input.buffer.percent</name>
        <value>0.9</value>
</property>
```



↑(Graph 8) Shuffe Input Buffer Size

As we expected, the execution time reduced when shuffle input buffer size is large. The reason is same with I/O file buffer size. It reduces the number of disk I/O which is expensive operation.

REMARKS

Above results are the small part of our experiments. We tried a lot of configurations but showed only for small part of configurations that had a big effect on performance. We want to introduce

about them that is not shown this report. Sort stage of Mapreduce is one of them. We tried to modify the buffer or the number of it, which are <Mapreduce.task.io.sort.mb>, <Mapreduce.task.io.sort.factor> and <Mapreduce.map.sort.spill.percent>. We also tried about the buffer of reducer and parallel copies of shuffle stages, which are <Mapreduce.reduce.input.buffer.percent> and <Mapreduce.reduce.shuffle.parallel.copies>. To save disk, we even tried to compress the size of intermediate data and output data by modifying <Mapreduce.output.fileoutputformat.compress>. <Mapred.child.java.opts> is also one of configuration not shown in the report. Even if these properties did not contribute on performance on our Hadoop cluster, they may can be major factors in other environments, so we recommend to try these factors too.

Throughout the experiments, there were some difficulties in carrying out study. The server was more unstable than expected and could not be connected or disconnected during the experiment. Also, the value varied depending on the time of experiment. To overcome this problem, we had to experiment several times on different dates.

There were also resource limitations that could be utilized. The number of nodes was small, and the replication factor was tested in only two cases. The capacity of the memory and the disk in the experiment is small, which is a limitation in various attempts.

In addition, minor parts such as IP address need to use public address instead of internal address, or use randomtextwriter instead of randomwriter, these became unexpected factors that prolonged the experiment.

CONCLUSION

By experimenting with several factors to improve Hadoop performance, it was able to see how each factor affected performance.

For example, there were efficient factors in the default state like the map.cpu.vcore factor, and there were factors that could improve the performance by changing the value like the buffer.size factor. In addition to these factors, some of the factors that have been tried to change have little value change, and some have difficulty in analyzing the tendency.

However, in order to improve the performance of Hadoop, various factors need to be changed in various ways. If do that, it will be able to see more of the trends that have not seen in this experiment and find better performance. And since the experiment will have to be consider a very large number of cases, it will be helpful to refer to the independent tendency of the factor as in this experiment.

Contribution : Beomsoo Kim – 40%, Gheejung Hong - 30%, Seungwon Lee – 30%