Sort application in Java, Hadoop, Spark

Veereshwaran Rangasamy Chettiar Ramamoorthi

vrangasa@hawk.iit.edu

All the experiments are performed in the following environment:

OS:

Ubuntu (Amazon trusted AMI ami-6c14310f) and then installed, configured for different needs.

I have customized the above for Hadoop and its AMI is ami-d9f2d1ba [us-west-2] (public)

I have customized the above for Hadoop and its AMI is ami-d8f2d1bb [us-west-2] (public)

Java:

Version 7

To install:

apt-get install default-jdk

Hadoop:

Version 2.7.2

Spark:

Version 1.4.1

1) Virtual Cluster:

Virtual Cluster of varying number of nodes on Amazon using c3.4xlarge is being set up for the sorting application.

2) Shared Memory Sort:

Shared Memory Sort is being implemented in Java using the following algorithm:

```
/*Segregating the records according to the first character*/
for each Record in File:
    read firstCharacter:
        if the firstCharacter doesn't exists in the TreeMap X:
            save the firstCharacter as key, lineNumber as value in the TreeMap X
            create a file (chunk) with lineNumber as name
            write the contents of the current record into the file (chunk)
        else
            fetch the value of the key (firstCharacter) from the TreeMap X
            open the file (chunk) with value(lineNumber) as name
            append the contents of the current record into the file (chunk)

/*TreeMap now contains the key(firstCharacter) in sorted order*/

/*Renaming the file, so that chunk will be in order*/
for each key in the TreeMap X:
    read the value for the key
    rename the file with the value of the key

/*Sorting chunk individually*/
for each chunk:
    open chunk:
        sort the contents of the file
        save it to a new file (sortedChunk)

/*Merging all the sorted chunks*/
for each sortedChunk:
    open sortedChunk:
        read the contents
        write/append the contents it to a file (sortedMergedFile)
```

As shown above, the big file is divided into smaller chunks in ascending order, sorted individually by consuming only the memory needed for the chunk and then sorted chunk is written to the output file. This way, the memory will not be fully consumed by the big size of the file; memory will be consumed only according to the chunk size.

According to the gensort application, it generates only 95 different characters. So there will be only 95 chunk files for any size of the big file. Also, the 95 files are seemed to be of more or less same size files.

The size of each chunk = (Size of Big file) / 95

For e.g., if the size of big file is 100 GB, then the size of each chunk will be 100GB/95=1.05GB.
So an average of 1.05GB memory will be consumed for sorting 100GB file.

Threaded Environment:

The same application is made to run in a multi-threaded manner. In such a multi-threaded environment, each thread will consume memory of size of each chunk.

For e.g., if the size of the big file is 100 GB, then the size of each chunk will be 100GB/95=1.05GB.

So in a two threaded environment, a memory of 2*1.05GB=2.10GB will be consumed. There will be parallel processing of chunks at a time and output will be produced 2x faster.

1 GB sorting:

The sorting of 1 GB data using 1 thread took nearly 37.084 seconds.

So the throughput is 26.965 MB/secs

The sorting of 1 GB data using 2 threads took nearly 19.872 seconds.

So the throughput is 50.32 MB/secs

The sorting of 1 GB data using 4 thread took nearly 11.293 seconds.

So the throughput is 88.55 MB/secs

The sorting of 1 GB data using 8 thread took nearly 1.897 seconds.

So the throughput is 527.14 MB/secs

10 GB sorting:

The sorting of 10 GB data using 1 thread took nearly 328.5 seconds.

So the throughput is 30.44 MB/sec

The sorting of 10 GB data using 2 threads took nearly 193.6 seconds.

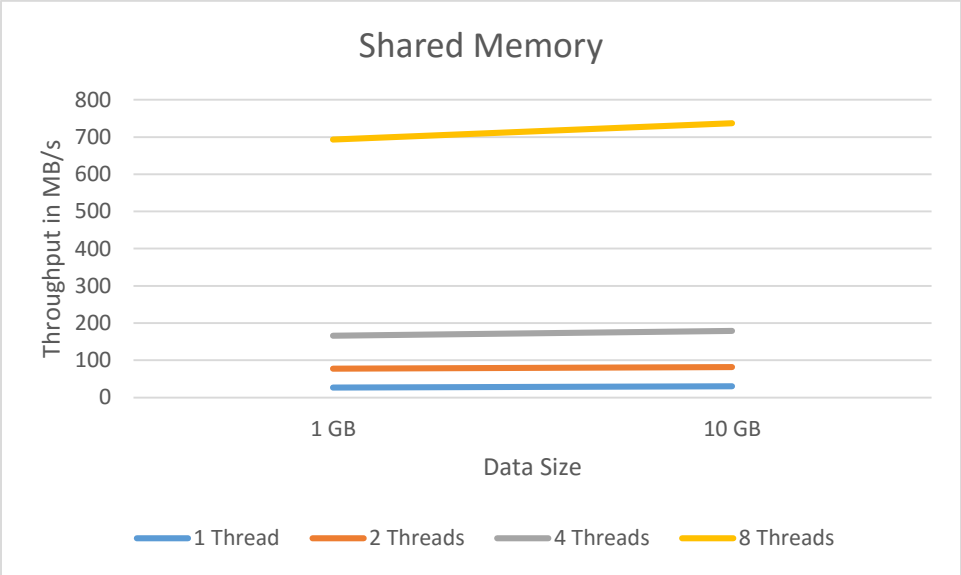
So the throughput is 193.6 MB/sec

The sorting of 10 GB data using 4 thread took nearly 103.5 seconds.

So the throughput is 96.61 MB/sec

The sorting of 10 GB data using 8 thread took nearly 17.92 seconds.

So the throughput is 558.03 MB/sec



3) Hadoop Sort:

Mappers:

Maps are the individual tasks which transform input records into intermediate records.

The framework calls the following methods in the order:

- setup(org.apache.hadoop.mapreduce.Mapper.Context)
- map (Object, Object, Context) for each key/value pair in the InputSplit
- cleanup(Context)

All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to a Reducer to determine the final output.

In Mapper, mapping all the first 10 bytes(key) of the record as the key and the rest bytes of the record as value.

Reducers:

Reducer has 3 primary phases:

- Shuffle
- Sort
- Reduce

As the instance type is c3.4xlarge, there are 16 vCPUs available. Among those 16 vCPUs, let's take only 14 vCPUs for our Hadoop sorting experiment.

We can define 7 mappers and 7 reducers for performing the sort.

In the latest Hadoop version (2.7.2), the configuration to define mappers are deprecated.

But there is an option to define the number of input splits, which in-turn defines the number of mapper tasks. So in order to define the number of input splits, we have to define the following property in mapred-site.xml

To define the number of reducers, we have to define the following property in mapred-site.xml

Config files

1) etc/hadoop/master

Master nodes hostnames are defined here

2) etc/hadoop/slaves

All slave nodes hostnames are defined here

3) etc/hadoop/core-site.xml

The Hostname and port address of the master node is defined here to denote the path of the Distributed File System.

4) etc/hadoop/hdfs-site.xml

In this file, replication factor, DFS's namenode path, DFS's datanode path are defined here.

5) etc/hadoop/mapred-site.xml

In this file, the name of the mapreduce framework (say yarn), mapreduce job tracker,etc are defined here.

Also the number of reducers can be defined here.

Defining the number of mappers is deprecated in Hadoop 2.7.2. But there is an option to work like the defining the number of mappers. This option is nothing but defining the input split size.

So by defining the input split size, the number of mappers will be called according to the data size.

6) etc/hadoop/yarn-site.xml

yarn-site.xml can be used to define resource manager address, resource tracker address, mapreduce shuffle class

Single node to multi configuration

Single node and multi node configurations are more or less same, but defining the slaves and masters make hadoop to work in a multi node environment.

And edit the yarn-site.xml as follows

<name>yarn.resourcemanager.resource-tracker.address</name>

<value>MASTER-IP:8025</value>

</property>

<property>

<name>yarn.resourcemanager.scheduler.address</name>

<value> MASTER-IP:8035</value>

</property>

<property>

<name>yarn.resourcemanager.address</name>

<value> MASTER-IP:8050</value>

</property>

Generating data, loading into HDFS, performing sorting experiment, getting back the result

In order to perform sorting of 10 GB data, we have to first create a 10 GB dataset using gensort with the following command

gensort –a 10000000 <filename>

This generates the 10 GB data and then we have put this data into HDFS using the following command

hadoop fs -mkdir /input_dir

hadoop fs -put /home/ubuntu/raid0/input /input_dir

hadoop fs -ls /input_dir

After this, we can run the program using the following command:

hadoop jar sorte.jar Sorte /input_dir/input output_dir

And save the output from hdfs to local using the following command:

hadoop fs -get output_dir /home/ubuntu/raid0

The experiment results are as follows:

10 GB sorting:

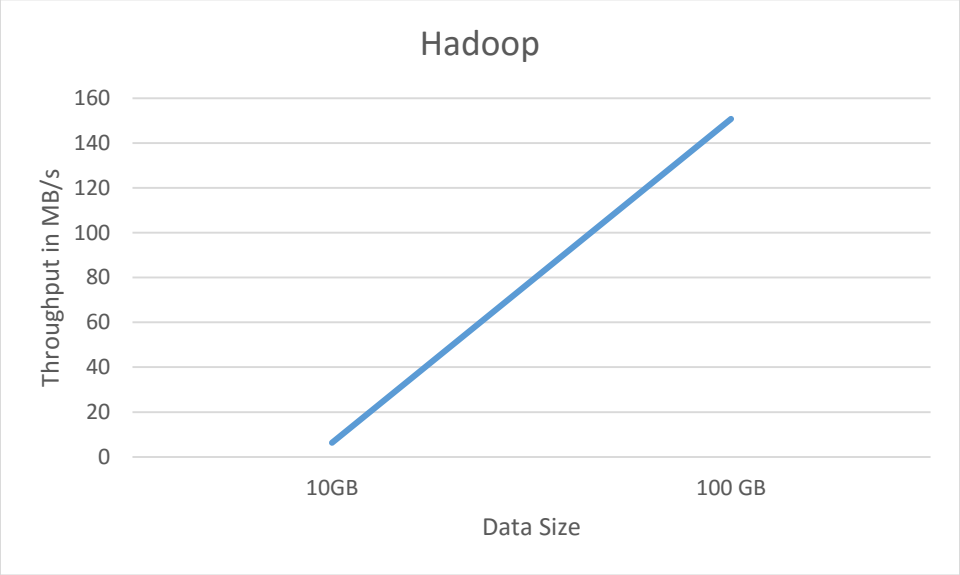
The sorting of 10 GB data on a single took nearly 1584 seconds.

So the throughput is 6.3131 MB/sec

100 GB sorting:

The sorting of 10 GB data on 16 slave nodes and 1 master node took nearly 663 seconds.

So the throughput is 150.82 MB/sec



4) Spark Sort:

Spark typically works on top of HDFS, but it has the capacity of moving blocks of data from memory of one node to memory of other node, which makes the data processing a way speeder.

Spark can take input from various sources such as HDFS, Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets.

From HDFS, spark create Resilient Distributed Datasets and performs transformation such as map filter, etc and then it can be further processed and distributed among memories of various nodes.

So, we can generate data using gensort and save them in HDFS as we have seen earlier.

Then we have to build the spark source code, generate jar file as veeresh-spark-terasort-dep.jar and then run the program as follows:

*spark-submit --class com.veeresh.terasort.spark.terasort.TeraSort spark-terasort-master/target/veeresh-spark-terasort-dep.jar hdfs://<hdfs URL> /input_dir/input hdfs://<hdfs URL>/output_dir/output*And

HDFS url is the URL which we defined in core-site.xml with name fs.default.name

save the output from hdfs to local using the following command:

hadoop fs -get output_dir /home/ubuntu/raid0

The experiment results are as follows:

10 GB sorting:

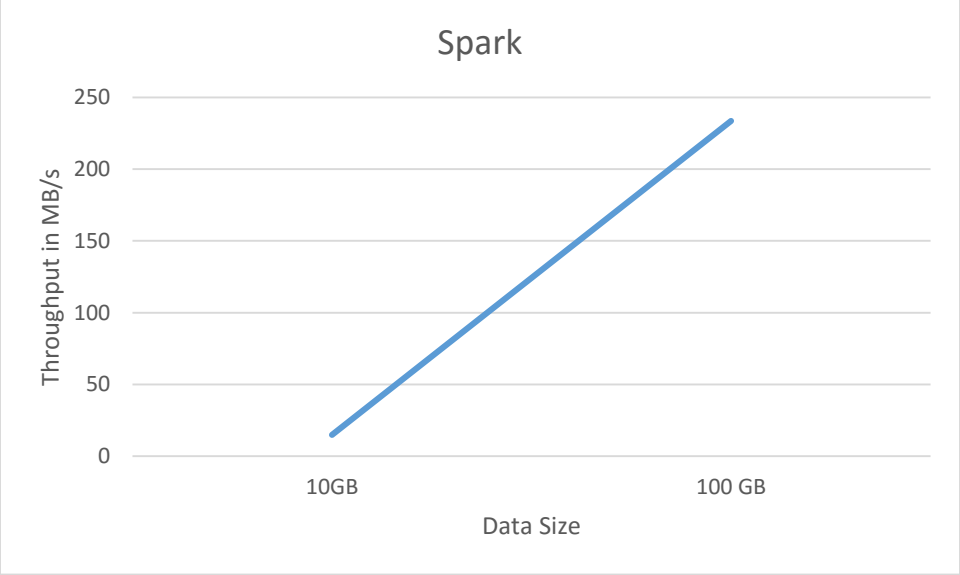
The sorting of 10 GB data took nearly 672 seconds.

So the throughput is 14.88 MB/sec

100 GB sorting:

The sorting of 10 GB data took nearly 428 seconds.

So the throughput is 233.64 MB/sec



Conclusions

From the results obtained from all the above experiments, it is observed that Shared Memory Sort runs faster than other two frameworks. This can be true, but with this setup, we cannot perform experiments in a distributed environment. Hadoop and Spark helps in performing operation in a distributed environment which in turn results in little slower performance for a smaller dataset.

When the dataset grows bigger, Spark and Hadoop will result in a very high performance.

Among Hadoop and Spark in these experiments, Spark runs really faster than Hadoop as Spark processes and make us of intermediary data by placing them in memory.

At 1 node scale, Hadoop seems to perform better than Spark. But when we go to 16 nodes, we can see a slight throughput raise with Spark when compared to Hadoop.

When we scale upto 100 nodes, there will be a big difference in the throughput achieved with Spark which can be even more than 10x to Hadoop's throughput.

When we scale upto 100 nodes, there will be a big difference in the throughput achieved with Spark which can be even 100x to Hadoop's throughput.

CloudSort benchmark is used to assess the efficacy of cloud environments for IO-intensive workloads. It makes the question of what is the minimum cost for sorting a fixed number of records in a public cloud. The reason for going public cloud is to get the effectiveness of cloud platform and Total Cost Ownership of external sort. The sort to be adapted is external sort, because it requires more IO intensive operations.

There are various advantages of going public cloud is

- Availability
- Affordability
- Auditability

The rules for this benchmark are

- Use only public cloud
- Pricing and Total Cost should be calculated on prorated basis for software, hardware, everything which comes under public cloud
- Storage should be done only in virtual arrays such as EBS, S3, etc
- Need not include the provisioning cost
- Reporting the total time taken

Scripts to run the experiment

For all these experiments, the type of AWS EC2 instance used is c3.4xlarge for all single node conditions; for master: slave conditions, c3.8xlarge is used as master and c3.4xlarge are used as slaves.

So in order to run to run all these experiments, I have made scripts, which will perform the following operations

1) Create a security group with the parameters needed for this experiment and fetch the created security group ID

2) Create a key to access the instances

3) Place spot-instance request with the following properties:

Instance Type (say c3.4xlarge)

AMI for the instance (say ami-948d67f4)

Key Name (say c3-large-fresh)

Security Group ID (say sg-b22947d5)

Availability Zone (say us-west-b)

Bid value (say $0.04)

Number of instances (say 16)

This request will return a list of values, among which the SpotInstanceRequests[*].Status.Code will return the

approval or denial of the request with the instances id(s) of the instance(s).

4) With the instance ID, that is fetch, we can fetch the following details

public IP address

public DNS

private IP address

private DNS

5) The above details can be used for

updating the hosts file,

masters and slaves of hadoop,

config files for hadoop,

slaves of spark,

config files for hadoop, etc

6) With all the updated files, we then distribute and update the files to all the nodes accordingly.

7) And then some actions to be performed on each node(such as RAID0, ssh-keygen, generating data and storing it in HDFS, etc) can be done as well.

8) Finally, when all the nodes are updated with required files, we are able to start required services such as start-dfs.sh, start-yarn.sh, $SPARK_HOME/sbin/start-all.sh, etc through the script written.

9) And then we can execute the programs through the scripts as well.

Requirements:

1) An account in AWS and the following details

    AWS Access Key ID

    AWS Secret Access Key

    Default region name

2) Install AWS Command Line Interface:

    $apt-get install aws-cli

3) Configure AWS CLI

    $aws configure