Hadoop Performance Tuning

Table of Contents [hide]

There are many ways to improve the performance of Hadoop jobs. In this post, we will provide a few MapReduce properties that can be used at various mapreduce phases to improve the performance tuning.

There is no one-size-fits-all technique for tuning Hadoop jobs, because of the architecture of Hadoop, achieving balance among resources is often more effective than addressing a single problem.

Depending on the type of job you are running and the amount of data you are moving, the solution might be quite different

We encourage you to experiment with these and to report your results.

Bottlenecks

Hadoop resources can be classified into computation, memory, network bandwidth and input and output (I/O). A job can run slowly if any of these resources perform badly. Below are the common resource bottlenecks in hadoop jobs.

- **CPU** – Key Resource for both Map and Reduce Tasks Computation
- **RAM** – Main Memory available on the slave (node manager) nodes.
- **Network Bandwidth** – When large amounts of data sets are being processed, high network utilization occurs among nodes. This may occur when Reduce tasks pull huge data from Map tasks in the Shuffle phase, and also when the job outputs the final results into HDFS.
- **Storage I/O** – File read write I/O throughput to HDFS. Storage I/O utilization heavily depends on the volume of input, intermediate data, and final output data.

Below are the common issues that may arise in Mapreduce Job Execution flow.Massive I/O Caused by Large Input Data in Map Input Stage.

Problem 1 – Massive I/O Caused by Large Input Data in Map Input Stage

This problem happens most often on jobs with light computation and large volumes of source data. If disk I/O is not fast enough, computation resources will be idle and spend most of the job time waiting for the incoming data. Therefore, performance can be constrained by disk I/O.

We can identify this issue with high values in below job counters.

- **Job counters**: Bytes Read, HDFS_BYTES_READ

Solution 1:Compress Input Data

Compress Input data – Compression of files saves storage space on HDFS and also improves speed of transfer.

We can use any of the below compression techniques on input data sets.

Format	Codec	Extension	Splittable	Hadoop
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec	.deflate	N	Y
Gzip	org.apache.hadoop.io.compress.GzipCodec	.gz	N	Y
Bzip2	org.apache.hadoop.io.compress.BZip2Codec	.bz2	Y	Y
LZO	com.hadoop.compression.lzo.LzopCodec	.lzo	N	Y
LZ4	org.apache.hadoop.io.compress.Lz4Codec	.Lz4	Y	Y
Snappy	org.apache.hadoop.io.compress.SnappyCodec	.Snappy	Y	Y

When we submit a MapReduce job against compressed data in HDFS, Hadoop will determine whether the source file is compressed by checking the file name extension, and if the file name has an appropriate extension, Hadoop will decompress it automatically using the appropriate codec. Therefore, users do not need to explicitly specify a codec in the MapReduce job.

However, if the file name extension does not follow naming conventions, Hadoop will not recognize the format and will not automatically decompress the file. Therefore, to enable self-detection and decompression, we must ensure that the file name extension matches the file name extensions supported by each codec.

Problem 2 – Massive I/O Caused by Spilled Records in Partition and Sort phases

When the map function starts producing output, it is not simply written to disk. Each map task has a circular memory buffer that it writes the output to. The buffer is**100 MB** by default. When the contents of the buffer reaches a certain threshold size, a background thread will start to spill the contents to disk. Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete. Spills are written in round-robin fashion to the directories specified by the **mapred.local.dir** property, in a job-specific sub directory.

To optimize the Map task outputs, we need to ensure that records are spilled (meaning, written to LFS or HDFS file) only once. When records are spilled more than once, the data must be read in and written out multiple

times, causing drains on I/O. If buffer size is too small and it is filled up too quickly, then it will lead to multiple spills. Each extra spill generates a large volume of data in intermediate bytes.

We can identify this issue with high values in below job counters.

- **Job counters**: FILE_BYTES_READ, FILE_BYTES_WRITTEN, Spilled Records

Solution 2:Adjust Spill Records and Sorting Buffer

To reduce the amount of data spilled during the intermediate Map phase, we can adjust the following properties for controlling sorting and spilling behavior.

mapreduce.task.io.sort.factor	10	The number of streams to merge at once while sorting files. This determines the number of open file handles.
mapreduce.task.io.sort.mb	100	The total amount of buffer memory to use while sorting files, in megabytes. By default, gives each merge stream 1MB, which should minimize seeks.
mapreduce.map.sort.spill.percent	0.80	The soft limit in the serialization buffer. Once reached, a thread will begin to spill the contents to disk in the background. Note that collection will not block if this threshold is exceeded while a spill is already in progress, so spills may be larger than this threshold when it is set to less than .5

When Map output is being sorted, **16 bytes of metadata** are **added** immediately **before each key-value pair**. These 16 bytes include 12 bytes for the key-value offset and 4 bytes for the indirect-sort index. Therefore, the total buffer space defined in **io.sort.mb** can be divided into two parts: **metadata buffer** and **key-value buffer**.

Formula for io.sort.mb

io.sort.mb = (16 + R) * N / 1,048,576

R – the average length of the key-value pairs (in bytes) and can be calculated by dividing the Map output bytes by the number of Map output records from job counters.

N – calculated by dividing the Map output records by the number of map tasks.

Update the **io.sort.spill.percent** property to 1.0 to make use of complete buffer space.

Problem 3 – Massive Network Traffic Caused by large Map Output

Large output from the Map phase can cause longer I/O and data transfer time, and in worst cases can raise exceptions, if all the I/O throughput channels are saturated or if network bandwidth is exhausted.

We can identify this issue with high values in below job counters.

- **Job counters**: FILE_BYTES_WRITTEN, FILE_BYTES_READ, Combine Input Records
- **Possible exceptions**: java.io.IOException

Solution 3.1: Compress Map Output

If Map Output is very large, it is always recommended to use compression techniques to reduce the size of intermediate data. By default, Map Output is not compressed but we can enable by setting below properties to true.

mapreduce.map.output.compress false

mapreduce.map.output.compress.codec org.apache.hadoop.io.compress.DefaultCodec

Below is the code snippet to enable gzip map output compression in our job:

```
Configuration conf = new Configuration(); conf.setBoolean("mapreduce.map.output.compress", true); conf.setClass("mapreduce.map.output.compress.codec", GzipCodec.class, CompressionCodec.class); Job job = new Job(conf);
```

```
1
2
3
4
5
6
```

```
Configuration conf = new Configuration();
conf.setBoolean("mapreduce.map.output.compress", true);
conf.setClass("mapreduce.map.output.compress.codec", GzipCodec.class,
CompressionCodec.class);
Job job = new Job(conf);
```

Solution 3.2: Implement a Combiner

We can also reduce the network I/O caused by Map Output by implementing Combiner if aggregate operation follows commutative and associative rule.

Problem 4 – Massive Network Traffic Caused by large Reduce Output

Large output from Reducers can cause lot of I/O write operations to HDFS.

We can identify this issue with high values in below job counters.

- **Job counters**: Bytes Written, HDFS_BYTES_WRITTEN
- **Possible exceptions**: java.io.IOException

The above two counters denote the volume of data from Reduce Phase, but these two counters do not include the replication factor. If the replication factor is greater than one, it means that blocks of data will be replicated to different nodes, which requires more I/O for read and write operations, and which also uses network bandwidth.

Solution 4.1: Compress Reducer/Final Output

We can enable compression on Mapreduce job's output by setting below properties to true at site level for all jobs.

mapreduce.output.fileoutputformat.compress	false	Compress?
mapreduce.output.fileoutputformat.compress.type	RECORD	If SequenceFiles, then it Should be one of NONE, RECORD or BLOCK.
mapreduce.output.fileoutputformat.compress.codec	org.apache.hadoop.io.compress.DefaultCodec	

Set the above properties either in Job driver using c**ode snippet like below (Snappy compression with Block Mode)** or in **mapred-site.xml** file.

- **If Output Files are Not Sequence Files**

```
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
```

- **If Output Files are Sequence Files**

```
job.setOutputFormatClass(SequenceFileOutputFormat.class);
SequenceFileOutputFormat.setCompressOutput(job, true);
```

SequenceFileOutputFormat.setOutputCompressorClass(job, SnappyCodec.class);
SequenceFileOutputFormat.setOutputCompressionType(job,
CompressionType.BLOCK);

- **To Make Global Changes to cluster**

```
<property>
<name>mapreduce.output.fileoutputformat.compress</name>
<value>true</value>
</property>
<property>
<name>mapreduce.output.fileoutputformat.compress.codec</name>
<value>SnappyCodec.class</value>
</property>
<property>
<name>mapreduce.output.fileoutputformat.compress.type</name>
<value>BLOCK</value>
</property>
```

Solution 4.2: Adjust Replication Factor

By reducing the replication factor to 1 when more replications are needed we can improve the job performance as , copying data to multiple nodes will be reduced. Set **dfs.replication** property to **1** using **conf** object **in Job Driver program.**

Problem 5 – Insufficient Parallel Tasks

If the number of parallel tasks running concurrently is insufficient to run the job, the job can leave many resources idle. Increasing the number of parallel tasks helps to accelerate the overall job execution by better utilizing resources.

Incorrect configurations may degrade the MR job performances some times. For example, if our data node machine has 16 CPU cores but we configured only 8 mappers on each machine, then remaining 8 cores will be idle because no work load will be assigned to them; As a result, only a limited portion of I/O throughput and network bandwidth will be actually utilized, because there are no requests coming from the other CPU cores.

Try to use all the available Map and Reduce task slots on the cluster across all the current running jobs. From YARN Web UI we can verify below counters to identify this issue.

- **Task Summary List**: Num Tasks, Running, Map Task Capacity, Reduce Task Capacity

Observe the cluster's total available Map and Reduce slots and Job's currently assigned no of Map and Reduce tasks to identify incorrect configuration.

Solution 5: Adjust Number of Map Tasks & Reduce Tasks & Memory

Both over allocation and Under allocation of Map Tasks & Reduce Tasks will degrade the performance. So we need to find out the optimized values by trial and error methods to keep the cluster resource utilization in balanced.

mapreduce.tasktracker.map.tasks.maximum	2	The maximum number of map tasks that will be run simultaneously by a task tracker.
mapreduce.tasktracker.reduce.tasks.maximum	2	The maximum number of reduce tasks that will be run simultaneously by a task tracker.

mapreduce.map.memory.mb	1024	The amount of memory to request from the scheduler for each map task.
mapreduce.map.cpu.vcores	1	The number of virtual cores to request from the scheduler for each map task.
mapreduce.reduce.memory.mb	1024	The amount of memory to request from the scheduler for each reduce task.
mapreduce.reduce.cpu.vcores	1	The number of virtual cores to request from the scheduler for each reduce task.
yarn.app.mapreduce.am.resource.mb	1536	The amount of memory the MR AppMaster needs.
yarn.app.mapreduce.am.resource.cpu-vcores	1	The number of virtual CPU cores the MR AppMaster needs.

The above are the default values in mapred-default.xml file and these can be overridden in **mapred-site.xml** to better utilize node managers resources completely.

We can also adjust the Memory for tasks with property **mapred.child.java.opts = -Xmx2048M** in**mapred-site.xml**

If we have 16 CPU cores and 32 GB RAM on Node Managers, then we can tune these properties upto 8 Map Tasks and 4 Reduce Tasks with memory 2048 MB allocated to each task at the maximum and leaving 4 cpu cores in buffer for other tasks/operations running on same Node Manager.

We can also **set these properties at Job level on Configuration Object**.

Below are some additional **Reduce Side Tuning Properties**

mapreduce.reduce.shuffle.parallelcopies	5	The default number of parallel transfers run by reduce during the copy(shuffle) phase.
mapreduce.shuffle.max.threads	0	Max allowed threads for serving shuffle connections. Set to zero to indicate the default of 2 times the number of available processors (as reported by Runtime.availableProcessors()). Netty is used to serve requests, so a thread is not needed for each connection.
mapreduce.shuffle.transferTo.allowed		This option can enable/disable using nio transferTo method in the shuffle phase. NIO transferTo does not perform well on windows in the shuffle phase. Thus, with this configuration property it is possible to disable it, in which case custom transfer method will be used. Recommended value is false when running Hadoop on Windows. For Linux, it is recommended to set it to true. If nothing is set then the default value is false for Windows, and true for Linux.
mapreduce.shuffle.transfer.buffer.size	131072	This property is used only if mapreduce.shuffle.transferTo.allowed is set to false. In that case, this property defines the size of the buffer used in the buffer copy code for the shuffle phase. The size of this buffer determines the size of the IO requests.
mapreduce.reduce.markreset.buffer.percent	0.0	The percentage of memory -relative to the maximum heap size- to be used for caching values when using the mark-reset functionality.
mapreduce.map.speculative	true	If true, then multiple instances of some map tasks may be executed in parallel.

mapreduce.reduce.speculative	true	If true, then multiple instances of some reduce tasks may be executed in parallel.
mapreduce.job.speculative.speculative-cap-running-tasks	0.1	The max percent (0-1) of running tasks that can be speculatively re-executed at any time.
mapreduce.job.speculative.speculative-cap-total-tasks	0.01	The max percent (0-1) of all tasks that can be speculatively re-executed at any time.
mapreduce.job.speculative.minimum-allowed-tasks	10	The minimum allowed tasks that can be speculatively re-executed at any time.
mapreduce.job.speculative.retry-after-no-speculate	1000	The waiting time(ms) to do next round of speculation if there is no task speculated in this round.
mapreduce.job.speculative.retry-after-speculate	15000	The waiting time(ms) to do next round of speculation if there are tasks speculated in this round.