Hadoop Performance Models

Herodotos Herodotou
hero@cs.duke.edu

Technical Report, CS-2011-05
Computer Science Department
Duke University

Abstract

Hadoop MapReduce is now a popular choice for performing large-scale data analytics. This technical report describes a detailed set of mathematical performance models for describing the execution of a MapReduce job on Hadoop. The models describe dataflow and cost information at the fine granularity of phases within the map and reduce tasks of a job execution. The models can be used to estimate the performance of MapReduce jobs as well as to find the optimal configuration settings to use when running the jobs.

The execution of a MapReduce job is broken down into map tasks and reduce tasks. Subsequently, map task execution is divided into the phases: *Read* (reading map inputs), *Map* (map function processing), *Collect* (serializing to buffer and partitioning), *Spill* (sorting, combining, compressing, and writing map outputs to local disk), and *Merge* (merging sorted spill files). Reduce task execution is divided into the phases: *Shuffle* (transferring map outputs to reduce tasks, with decompression if needed), *Merge* (merging sorted map outputs), *Reduce* (reduce function processing), and *Write* (writing reduce outputs to the distributed file-system). Each phase represents an important part of the job's overall execution in Hadoop. We have developed performance models for each task phase, which are then combined to form the overall Map-Reduce Job model.

1 Model Parameters

The performance models rely on a set of parameters to estimate the cost of a Map-Reduce job. We separate the parameters into three categories:

1. *Hadoop Parameters*: A set of Hadoop-defined configuration parameters that effect the execution of a job
2. *Profile Statistics*: A set of statistics specifying properties of the input data and the user-defined functions (Map, Reduce, Combine)
3. *Profile Cost Factors*: A set of parameters that define the I/O, CPU, and network cost of a job execution

Variable	Hadoop Parameter	Default Value	Effect
pNumNodes	Number of Nodes		System
pTaskMem	mapred.child.java.opts	-Xmx200m	System
pMaxMapsPerNode	mapred.tasktracker.map.tasks.max	2	System
pMaxRedPerNode	mapred.tasktracker.reduce.tasks.max	2	System
pNumMappers	mapred.map.tasks		Job
pSortMB	io.sort.mb	100 MB	Job
pSpillPerc	io.sort.spill.percent	0.8	Job
pSortRecPerc	io.sort.record.percent	0.05	Job
pSortFactor	io.sort.factor	10	Job
pNumSpillsForComb	min.num.spills.for.combine	3	Job
pNumReducers	mapred.reduce.tasks		Job
pInMemMergeThr	mapred.inmem.merge.threshold	1000	Job
pShuffleInBufPerc	mapred.job.shuffle.input.buffer.percent	0.7	Job
pShuffleMergePerc	mapred.job.shuffle.merge.percent	0.66	Job
pReducerInBufPerc	mapred.job.reduce.input.buffer.percent	0	Job
pUseCombine	mapred.combine.class or mapreduce.combine.class	null	Job
pIsIntermCompressed	mapred.compress.map.output	false	Job
pIsOutCompressed	mapred.output.compress	false	Job
pReduceSlowstart	mapred.reduce.slowstart.completed.maps	0.05	Job
pIsInCompressed	Whether the input is compressed or not		Input
pSplitSize	The size of the input split		Input

Table 1: Variables for Hadoop Parameters

Table 1 defines the variables that are associated with Hadoop parameters.

Table 2 defines the necessary profile statistics specific to a job and the data it is processing.

Variable	Description
sInputPairWidth	The average width of the input K-V pairs
sMapSizeSel	The selectivity of the mapper in terms of size
sMapPairsSel	The selectivity of the mapper in terms of number of K-V pairs
sReduceSizeSel	The selectivity of the reducer in terms of size
sReducePairsSel	The selectivity of the reducer in terms of number of K-V pairs
sCombineSizeSel	The selectivity of the combine function in terms of size
sCombinePairsSel	The selectivity of the combine function in number of K-V pairs
sInputCompressRatio	The ratio of compression for the input data
sIntermCompressRatio	The ratio of compression for the intermediate map output
sOutCompressRatio	The ratio of compression for the final output of the job

Table 2: Variables for Profile Statistics

Table 3 defines system specific parameters needed for calculating I/O, CPU, and network costs. The IO costs and CPU costs related to compression are defined in terms of time per byte. The rest CPU costs are defined in terms of time per K-V pair. The network cost is defined in terms of transferring time per byte.

Variable	Description
cHdfsReadCost	The cost for reading from HDFS
cHdfsWriteCost	The cost for writing to HDFS
cLocalIOCost	The cost for performing I/O from the local disk
cNetworkCost	The network transferring cost
cMapCPUCost	The CPU cost for executing the map function
cReduceCPUCost	The CPU cost for executing the reduce function
cCombineCPUCost	The CPU cost for executing the combine function
cPartitionCPUCost	The CPU cost for partitioning
cSerdeCPUCost	The CPU cost for serialization
cSortCPUCost	The CPU cost for sorting on keys
cMergeCPUCost	The CPU cost for merging
cInUncomprCPUCost	The CPU cost for uncompressing the input data
cIntermUncomprCPUCost	The CPU cost for uncompressing the intermediate data
cIntermComprCPUCost	The CPU cost for compressing the intermediate data
cOutComprCPUCost	The CPU cost for compressing the output data

Table 3: Variables for Profile Cost Factors

Let's define the identity function I as:

$$I(x) = \begin{cases} 1 & \text{if x exists or equals true} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Initializations: In an effort present concise formulas and avoid the use of conditionals as much as possible, we make the following initializations:

If ($pUseCombine ==$ FALSE)
$sCombineSizeSel = 1$
$sCombinePairsSel = 1$
$cCombineCPUCost = 0$

If ($pIsInCompressed ==$ FALSE)
$sInputCompressRatio = 1$
$cInUncomprCPUCost = 0$

If ($pIsIntermCompressed ==$ FALSE)
$sIntermCompressRatio = 1$
$cIntermUncomprCPUCost = 0$
$cIntermComprCPUCost = 0$

If ($pIsOutCompressed ==$ FALSE)
$sOutCompressRatio = 1$
$cOutComprCPUCost = 0$

2 Performance Models for the Map Task Phases

The Map Task execution is divided into five phases:

1. *Read*: Reading the input split and creating the key-value pairs.
2. *Map*: Executing the user-provided map function.
3. *Collect*: Collecting the map output into a buffer and partitioning.
4. *Spill*: Sorting, using the combiner if any, performing compression if asked, and finally spilling to disk, creating *file spills*.
5. *Merge*: Merging the file spills into a single map output file. Merging might be performed in multiple rounds.

2.1 Modeling the Read and Map Phases

During this phase, the input split is read, uncompressed if necessary, the key-value pairs are created, and passed an input to the user-defined map function.

$$inputMapSize = \frac{pSplitSize}{sInputCompressRatio} \tag{2}$$

$$inputMapPairs = \frac{inputMapSize}{sInputPairWidth} \tag{3}$$

The costs of this phase are:

$$IOCost_{Read} = pSplitSize \times cHdfsReadCost$$

$$\begin{aligned} CPUCost_{Read} = {} & pSplitSize \times cInUncomprCPUCost \\ & + inputMapPairs \times cMapCPUCost \end{aligned} \tag{4}$$

If the MR job consists only of mappers (i.e. $pNumReducers = 0$), then the spilling and merging phases will not be executed and the map output will be written directly to HDFS.

$$outMapSize = inputMapSize \times sMapSizeSel \tag{5}$$

$$IOCost_{MapWrite} = outMapSize \times sOutCompressRatio \times cHdfsWriteCost \tag{6}$$

$$CPUCost_{MapWrite} = outMapSize \times cOutComprCPUCost \tag{7}$$

2.2 Modeling the Collect and Spill Phases

The map function generates output key-value (K-V) pairs that are placed in the map-side memory buffer. The formulas regarding the map output are:

$$outMapSize = inputMapSize \times sMapSizeSel \tag{8}$$

$$outMapPairs = inputMapPairs \times sMapPairsSel \tag{9}$$

$$outPairWidth = \frac{outMapSize}{outMapPairs} \tag{10}$$

The memory buffer is split into two parts: the *serialization* part that stores the key-value pairs, and the *accounting* part that stores metadata per pair. When either of these two parts fills up (based on the threshold value *pSpillPerc*), the pairs are partitioned, sorted, and spilled to disk. The maximum number of pairs for the serialization buffer is:

$$maxSerPairs = \left\lfloor \frac{pSortMB \times 2^{20} \times (1 - pSortRecPerc) \times pSpillPerc}{outPairWidth} \right\rfloor \tag{11}$$

The maximum number of pairs for the accounting buffer is:

$$maxAccPairs = \left\lfloor \frac{pSortMB \times 2^{20} \times pSortRecPerc \times pSpillPerc}{16} \right\rfloor \tag{12}$$

Hence, the number of pairs and size of the buffer before a spill will be:

$$spillBufferPairs = Min\{\ maxSerPairs,\ maxAccPairs\ ,\ outMapPairs\ \} \tag{13}$$

$$spillBufferSize = spillBufferPairs \times outPairWidth \tag{14}$$

The overall number of spills will be:

$$numSpills = \left\lceil \frac{outMapPairs}{spillBufferPairs} \right\rceil \tag{15}$$

The number of pairs and size of each spill depends on the width of each K-V pair, the use of the combine function, and the use of intermediate data compression. Note that *sIntermCompressRatio* is set to 1 by default, if intermediate compression is disabled. Note that *sCombinePairsSel* and *sCombinePairsSel* are set to 1 by default, if no combine function is used.

$$spillFilePairs = spillBufferPairs \times sCombinePairsSel \tag{16}$$

$$spillFileSize = spillBufferSize \times sCombineSizeSel \times sIntermCompressRatio \tag{17}$$

The costs of this phase are:

$$IOCost_{Spill} = numSpills \times spillFileSize \times cLocalIOCost \tag{18}$$

$$\begin{aligned} CPUCost_{Spill} = numSpills \times \\ [\ & spillBufferPairs \times cPartitionCPUCost \\ & + spillBufferPairs \times cSerdeCPUCost \\ & + spillBufferPairs \times \log_2(\frac{spillBufferPairs}{pNumReducers}) \times cSortCPUCost \\ & + spillBufferPairs \times cCombineCPUCost \\ & + spillBufferSize \times sCombineSizeSel \times cIntermComprCPUCost\ ] \end{aligned} \tag{19}$$

2.3 Modeling the Merge Phase

The goal of the merge phase is to merge all the spill files into a single output file, which is written to local disk. The merge phase will occur only if more that one spill file is created. Multiple merge passes might occur, depending on the *pSortFactor* parameter. We define a *merge pass* to be the merging of at most *pSortFactor* spill files. We define a *merge round* to be one or more merge passes that merge only spills produced by the spill phase or a previous merge round. For example, suppose $numSpills = 30$ and $pSortFactor = 10$. Then, 3 merge passes will be performed to create 3 new files. This is the first merge round. Then, the 3 new files will be merged together forming the 2nd and final merge round.

The final merge pass is unique in the sense that if the number of spills to be merged is greater than or equal to *pNumSpillsForComb*, the combiner will be used again. Hence, we treat the intermediate merge rounds and the final merge separately. For the intermediate merge passes, we calculate how many times (on average) a single spill will be read.

Note that the remaining section assumes $numSpils \leq pSortFactor^2$. In the opposite case, we must use a simulation-based approach in order to calculate the number of spills merged during the intermediate merge rounds as well as the total number of merge passes.

The first merge pass is also unique because Hadoop will calculate the optimal number of spill files to merge so that all other merge passes will merge exactly *pSortFactor* files.
Since the Reduce task also contains a similar Merge Phase, we define the following three methods to reuse later:

$$calcNumSpillsFirstPass(N, F) = \begin{cases} N & \text{, if } N \leq F \\ F & \text{, if } (N-1)\ MOD\ (F-1) = 0 \\ (N-1)\ MOD\ (F-1) + 1 & \text{, otherwise} \end{cases} \tag{20}$$

$$calcNumSpillsIntermMerge(N, F) = \begin{cases} 0 & \text{, if } N \leq F \\ P + \lfloor \frac{N-P}{F} \rfloor * F & \text{, if } N \leq F^2 \\ & \text{, where } P = calcNumSpillsFirstPass(N, F) \end{cases} \tag{21}$$

$$calcNumSpillsFinalMerge(N,F) = \begin{cases} N & \text{, if } N \leq F \\ 1 + \lfloor \frac{N-P}{F} \rfloor + (N-S) & \text{, if } N \leq F^2 \end{cases}$$
$$\text{, where } P = calcNumSpillsFirstPass(N,F)$$
$$\text{, where } S = calcNumSpillsIntermMerge(N,F) \quad (22)$$

The number of spills read during the first merge pass is:

$$numSpillsFirstPass = calcNumSpillsFirstPass(numSpills, pSortFactor) \quad (23)$$

The number of spills read during the intermediate merging is:

$$numSpillsIntermMerge = calcNumSpillsIntermMerge(numSpills, pSortFactor) \quad (24)$$

The total number of merge passes will be:

$$numMergePasses = \begin{cases} 0 & \text{, if } numSpills = 1 \\ 1 & \text{, if } numSpills \leq pSortFactor \\ 2 + \left\lfloor \frac{numSpills - numSpillsFirstPass}{pSortFactor} \right\rfloor & \text{, if } numSpills \leq pSortFactor^2 \end{cases} \quad (25)$$

The number of spill files for the final merge round is (first pass + intermediate passes + remaining file spills):

$$numSpillsFinalMerge = calcNumSpillsFinalMerge(numSpills, pSortFactor) \quad (26)$$

The total number of records spilled is:

$$numRecSpilled = spillFilePairs \times [numSpills + numSpillsIntermMerge + numSpills \times sCombinePairsSel] \quad (27)$$

The final map output size and number of K-V pairs are:

$$useCombInMerge = (numSpills > 1) \text{ AND } (pUseCombine) \text{ AND } (numSpillsFinalMerge \geq pNumSpillsForComb) \quad (28)$$

$$intermDataSize = numSpills \times spillFileSize \times \begin{cases} sCombineSizeSel & \text{if } useCombInMerge \\ 1 & \text{otherwise} \end{cases} \quad (29)$$

$$intermDataPairs = numSpills \times spillFilePairs \times \begin{cases} sCombinePairsSel & \text{if } useCombInMerge \\ 1 & \text{otherwise} \end{cases} \quad (30)$$

The costs of this phase are:

$$
\begin{aligned}
&IOCost_{Merge} = \\
&\quad 2 \times numSpillsIntermMerge \times spillFileSize \times cLocalIOCost && \text{// interm merges} \\
&\quad + numSpills \times spillFileSize \times cLocalIOCost && \text{// read final merge} \\
&\quad + intermDataSize \times cLocalIOCost && \text{// write final merge}
\end{aligned} \tag{31}
$$

$$
\begin{aligned}
&CPUCost_{Merge} = \\
&\quad numSpillsIntermMerge \times \\
&\qquad [\ spillFileSize \times cIntermUncomprCPUCost \\
&\qquad\ + spillFilePairs \times cMergeCPUCost \\
&\qquad\ + \frac{spillFileSize}{sIntermCompressRatio} \times cIntermComprCPUCost\ ] \\
&\quad + numSpills \times \\
&\qquad [\ spillFileSize \times cIntermUncomprCPUCost \\
&\qquad\ + spillFilePairs \times cMergeCPUCost \\
&\qquad\ + spillFilePairs \times cCombineCPUCost\ ] \\
&\quad + \frac{intermDataSize}{sIntermCompressRatio} \times cIntermComprCPUCost
\end{aligned} \tag{32}
$$

2.4 Modeling the Overall Map Task

The above models correspond to the execution of a single map task. The overall costs for a single map task are:

$$
IOCost_{Map} = \begin{cases} IOCost_{Read} + IOCost_{MapWrite} & \text{if } pNumReducers = 0 \\ IOCost_{Read} + IOCost_{Spill} + IOCost_{Merge} & \text{if } pNumReducers > 0 \end{cases} \tag{33}
$$

$$
CPUCost_{Map} = \begin{cases} CPUCost_{Read} + CPUCost_{MapWrite} & \text{if } pNumReducers = 0 \\ CPUCost_{Read} + CPUCost_{Spill} + CPUCost_{Merge} & \text{if } pNumReducers > 0 \end{cases} \tag{34}
$$

3 Performance Models for the Reduce Task Phases

The Reduce Task is divided into four phases:

1. *Shuffle*: Copying the map output from the mapper nodes to a reducer's node and decompressing, if needed. Partial merging may also occur during this phase.
2. *Merge*: Merging the sorted fragments from the different mappers to form the input to the reduce function.
3. *Reduce*: Executing the user-provided reduce function.
4. *Write*: Writing the (compressed) output to HDFS.

3.1 Modeling the Shuffle Phase

The following discussion refers to the execution of a single reduce task. In the Shuffle phase, the framework fetches the relevant map output partition from each mapper (called *segment*) and copies it to the reducer's node. If the map output is compressed, it will be uncompressed. For each map segment that reaches the reduce side we have:

$$segmentComprSize = \frac{intermDataSize}{pNumReducers} \tag{35}$$

$$segmentUncomprSize = \frac{segmentComprSize}{sIntermCompressRatio} \tag{36}$$

$$segmentPairs = \frac{intermDataPairs}{pNumReducers} \tag{37}$$

where *intermDataSize* and *intermDataPairs* are the size and number of pairs produced as intermediate output by a single mapper (see Section 2.3).

The data fetched to a single reducer will be:

$$totalShuffleSize = pNumMappers * segmentComprSize \tag{38}$$

$$totalShufflePairs = pNumMappers * segmentPairs \tag{39}$$

As the data is copied to the reducer, they are placed in the shuffle buffer in memory with size:

$$shuffleBufferSize = pShuffleInBufPerc \times pTaskMem \tag{40}$$

When the in-memory buffer reaches a threshold size or the number of segments becomes greater than the *pInMemMergeThr*, the segments are merged and spilled to disk creating a new local file (called *shuffleFile*). The merge size threshold is:

$$mergeSizeThr = pShuffleMergePerc \times shuffleBufferSize \tag{41}$$

However, when the segment size is greater that 25% of the *shuffleBufferSize*, the segment will go straight to disk instead of passing through memory (hence, no in-memory merging will occur).

Case 1: $segmentUncomprSize < 0.25 \times shuffleBufferSize$

$$numSegInShuffleFile = \frac{mergeSizeThr}{segmentUncomprSize} \tag{42}$$

If $(\lceil numSegInShuffleFile \rceil \times segmentUncomprSize \leq shuffleBufferSize)$
  $numSegInShuffleFile = \lceil numSegInShuffleFile \rceil$
else
  $numSegInShuffleFile = \lfloor numSegInShuffleFile \rfloor$

If ($numSegInShuffleFile > pInMemMergeThr$)

$$numSegInShuffleFile = pInMemMergeThr \tag{43}$$

A shuffle file is the merging on *numSegInShuffleFile* segments. If a combine function is specified, then it is applied during this merging. Note that if $numSegInShuffleFile > numMappers$, then merging will not happen.

$$shuffleFileSize = numSegInShuffleFile \times segmentComprSize \times sCombineSizeSel \tag{44}$$

$$shuffleFilePairs = numSegInShuffleFile \times segmentPairs \times sCombinePairsSel \tag{45}$$

$$numShuffleFiles = \left\lfloor \frac{pNumMappers}{numSegInShuffleFile} \right\rfloor \tag{46}$$

At the end of the merging, some segments might remain in memory.

$$numSegmentsInMem = pNumMappers \text{ MOD } numSegInShuffleFile \tag{47}$$

Case 2: $segmentUncomprSize \geq 0.25 \times shuffleBufferSize$

$$numSegInShuffleFile = 1 \tag{48}$$

$$shuffleFileSize = segmentComprSize \tag{49}$$

$$shuffleFilePairs = segmentPairs \tag{50}$$

$$numShuffleFiles = pNumMappers \tag{51}$$

$$numSegmentsInMem = 0 \tag{52}$$

Either case will create a set of shuffle files on disk. When the number of shuffle files on disk increases above a certain threshold ($2 \times pSortFactor - 1$), a new merge thread is triggered and *pSortFactor* shuffle files are merged into a new larger sorted one. The Combiner is not used during this disk merging. The total number of such merges are:

$$numShuffleMerges = \begin{cases} 0 & \text{, if } numShuffleFiles < 2 \times pSortFactor - 1 \\ \left\lfloor \frac{numShuffleFiles - 2 \times pSortFactor + 1}{pSortFactor} \right\rfloor + 1 & \text{, otherwise} \end{cases} \tag{53}$$

At the end of the Shuffle phase, a set of merged and unmerged shuffle files will exist on disk.

$$numMergShufFiles = numShuffleMerges \tag{54}$$

$$mergShufFileSize = pSortFactor \times shuffleFileSize \tag{55}$$

$$mergShufFilePairs = pSortFactor \times shuffleFilePairs \tag{56}$$

$$numUnmergShufFiles = numShuffleFiles - pSortFactor \times numShuffleMerges \tag{57}$$

$$unmergShufFileSize = shuffleFileSize \tag{58}$$

$$unmergShufFilePairs = shuffleFilePairs \tag{59}$$

The cost of the Shuffling phase is:

$$\begin{aligned} IOCost_{Shuffle} &= numShuffleFiles \times shuffleFileSize \times cLocalIOCost \\ &\quad + numMergShufFiles \times mergShufFileSize \times 2 \times cLocalIOCost \end{aligned} \tag{60}$$

$$\begin{aligned} &CPUCost_{Shuffle} = \\ &\quad [\ totalShuffleSize \times cIntermUncomprCPUCost \\ &\qquad + numShuffleFiles \times shuffleFilePairs \times cMergeCPUCost \\ &\qquad + numShuffleFiles \times shuffleFilePairs \times cCombineCPUCost \\ &\qquad + numShuffleFiles \times \frac{shuffleFileSize}{sIntermCompressRatio} \times cIntermComprCPUCost \\ &\quad] \times I(segmentUncomprSize < 0.25 \times shuffleBufferSize) \\ &\quad + numMergShufFiles \times mergShufFileSize \times cIntermUncomprCPUCost \\ &\quad + numMergShufFiles \times mergShufFilePairs \times cMergeCPUCost \\ &\quad + numMergShufFiles \times \frac{mergShufFileSize}{sIntermCompressRatio} \times cIntermComprCPUCost \end{aligned} \tag{61}$$

3.2 Modeling the Merge Phase

After all the map outputs have been successful copied in memory and/or on disk, the sorting/merging phase begins. This phase will merge all data into a single stream that is fed to the reducer. Similar to the Map Merge phase (see Section 2.3), this phase may occur it multiple rounds, but during the final merging, instead of creating a single output file, it will send the data directly to the reducer.

The shuffle phase produced a set of merged and unmerged shuffle files on disk, and perhaps a set of segments in memory. The merging is done in three steps.

Step 1: Some segments might be evicted from memory and merged into a single shuffle file to satisfy the memory constraint enforced by *pReducerInBufPerc*. (This parameter specifies the amount of memory allowed to be occupied by segments before the reducer begins.)

$$maxSegmentBuffer = pReducerInBufPerc \times pTaskMem \tag{62}$$

$$currSegmentBuffer = numSegmentsInMem \times segmentUncomprSize \tag{63}$$

If $(currSegmentBuffer > maxSegmentBuffer)$

$$numSegmentsEvicted = \left\lceil \frac{currSegmentBuffer - maxSegmentBuffer}{segmentUncomprSize} \right\rceil$$

else

$$numSegmentsEvicted = 0 \tag{64}$$

$$numSegmentsRemainMem = numSegmentsInMem - numSegmentsEvicted \tag{65}$$

The above merging will only occur if the number of existing shuffle files on disk are less than the *pSortFactor*. If not, then the shuffle files would have to be merged, and the in-memory segments that are supposed to be evicted are left to be merge with the shuffle files on disk.

$$numFilesOnDisk = numMergShufFiles + numUnmergShufFiles \tag{66}$$

If $(numFilesOnDisk < pSortFactor)$

$$numFilesFromMem = 1$$
$$filesFromMemSize = numSegmentsEvicted \times segmentComprSize$$
$$filesFromMemPairs = numSegmentsEvicted \times segmentPairs$$
$$step1MergingSize = filesFromMemSize$$
$$step1MergingPairs = filesFromMemPairs$$

else

$$numFilesFromMem = numSegmentsEvicted$$
$$filesFromMemSize = segmentComprSize$$
$$filesFromMemPairs = segmentPairs$$
$$step1MergingSize = 0$$
$$step1MergingPairs = 0 \tag{67}$$

$$filesToMergeStep2 = numFilesOnDisk + numFilesFromMem \tag{68}$$

Step 2: Any files on disk will go through a merging phase in multiple rounds (similar to the process in Section 2.3. This step will happen only if $numFilesOnDisk > 0$ (which implies $filesToMergeStep2 > 0$). The number of intermediate reads (and writes) are:

$$intermMergeReads = calcNumSpillsIntermMerge(filesToMergeStep2, pSortFactor) \tag{69}$$

The main difference from Section 2.3 is that the merged files have different sizes. We account for this by attributing merging costs proportionally.

$$\begin{aligned} step2MergingSize = & \frac{intermMergeReads}{filesToMergeStep2} \times \\ & [\ numMergShufFiles \times mergShufFileSize \\ & + numUnmergShufFiles \times unmergShufFileSize \\ & + numFilesFromMem \times filesFromMemSize] \end{aligned} \tag{70}$$

$$\begin{aligned} step2MergingPairs = & \frac{intermMergeReads}{filesToMergeStep2} \times \\ & [\ numMergShufFiles \times mergShufFilePairs \\ & + numUnmergShufFiles \times unmergShufFilePairs \\ & + numFilesFromMem \times filesFromMemPairs] \end{aligned} \tag{71}$$

$$filesRemainFromStep2 = calcNumSpillsFinalMerge(filesToMergeStep2, pSortFactor) \tag{72}$$

Step 3: All files on disk and in memory will go through merging.

$$filesToMergeStep3 = filesRemainFromStep2 + numSegmentsRemainMem \tag{73}$$

The process is identical to step 2 above.

$$intermMergeReads = calcNumSpillsIntermMerge(filesToMergeStep3, pSortFactor) \tag{74}$$

$$step3MergingSize = \frac{intermMergeReads}{filesToMergeStep3} \times totalShuffleSize \tag{75}$$

$$step3MergingPairs = \frac{intermMergeReads}{filesToMergeStep3} \times totalShufflePairs \tag{76}$$

$$filesRemainFromStep3 = calcNumSpillsFinalMerge(filesToMergeStep3, pSortFactor) \tag{77}$$

$$totalMergingSize = step1MergingSize + step2MergingSize + step3MergingSize \tag{78}$$

The cost of the Sorting phase is:

$$IOCost_{Sort} = totalMergingSize \times cLocalIOCost \tag{79}$$

$$\begin{aligned} CPUCost_{Sort} = & \\ & totalMergingSize \times cMergeCPUCost \\ & [\frac{totalMergingSize}{sIntermCompressRatio}] \times cIntermComprCPUCost \\ & [step2MergingSize + step3MergingSize] \times cIntermUnomprCPUCost \end{aligned} \tag{80}$$

3.3 Modeling the Reduce and Write Phases

Finally, the user-provided reduce function will be executed and the output will be written to HDFS.

$$\begin{aligned} inReduceSize = & \frac{numShuffleFiles \times shuffleFileSize}{sIntermCompressRatio} \\ & + \frac{numSegmentsInMem \times segmentComprSize}{sIntermCompressRatio} \end{aligned} \tag{81}$$

$$\begin{aligned} inReducePairs = & numShuffleFiles \times shuffleFilePairs \\ & + numSegmentsInMem \times segmentComprPairs \end{aligned} \tag{82}$$

$$outReduceSize = inReduceSize \times sReduceSizeSel \tag{83}$$

$$outReducePairs = inReducePairs \times sReducePairsSel \tag{84}$$

The input to the reduce function resides in memory and/or in the shuffle files produced by the Shuffling and Sorting phases.

$$\begin{aligned} inRedSizeDiskSize = & numMergShufFiles \times mergShufFileSize \\ & + numUnmergShufFiles \times unmergShufFileSize \\ & + numFilesFromMem \times filesFromMemSize \end{aligned} \tag{85}$$

The cost of the Write phase is:

$$\begin{aligned} IOCost_{Write} = & inRedSizeDiskSize \times cLocalIOCost \\ & + outReduceSize \times sOutCompressRatio \times cHdfsWriteCost \end{aligned} \tag{86}$$

$$\begin{aligned} CPUCost_{Write} = & inReducePairs \times cReduceCPUCost \\ & + inRedSizeDiskSize \times cIntermUncompCPUCost \\ & + outReduceSize \times cOutComprCPUCost \end{aligned} \tag{87}$$

3.4 Modeling the Overall Reduce Task

The above models correspond to the execution of a single reduce task. The overall costs for a single reduce task, excluding network transfers, are:

$$IOCost_{Reduce} = IOCost_{Shuffle} + IOCost_{Sort} + IOCost_{Write} \tag{88}$$

$$CPUCost_{Reduce} = CPUCost_{Shuffle} + CPUCost_{Sort} + CPUCost_{Write} \tag{89}$$

4 Performance Models for the Network Transfer

During the shuffle phase, all the data produced by the map tasks is copied over to the nodes running the reduce tasks (except for the data that is local). The overall data transferred in the network is:

$$netTransferSize = finalOutMapSize \times pNumMappers \times \frac{pNumNodes - 1}{pNumNodes} \tag{90}$$

where *finalOutMapSize* is the size of data produced by a single map tasks.

The overall cost for transferring data over the network is:

$$NETCost_{Job} = netTransferSize \times networkCost \tag{91}$$

5 Performance Models for the Map-Reduce Job

The MapReduce job consists of several map and reduce tasks executing in parallel and in waves. There are two primary ways to estimating the total costs of the job: (i) simulate the task execution using a *Task Scheduler Simulator*, and (ii) calculate the expected total costs analytically.

Simulation involves scheduling and simulating the execution of individual tasks on a virtual Cluster. The cost for each task is calculated using the proposed performance models.

The second approach involves using the following analytical costs:

$$IOCost_{AllMaps} = \frac{pNumMappers \times IOCost_{Map}}{pNumNodes \times pMaxMapsPerNode} \tag{92}$$

$$CPUCost_{AllMaps} = \frac{pNumMappers \times CPUCost_{Map}}{pNumNodes \times pMaxMapsPerNode} \tag{93}$$

$$IOCost_{AllReducers} = \frac{pNumReducers \times IOCost_{Reduce}}{pNumNodes \times pMaxRedPerNode} \tag{94}$$

$$CPUCost_{AllReducers} = \frac{pNumReducers \times CPUCost_{Reduce}}{pNumNodes \times pMaxRedPerNode} \tag{95}$$

The overall job cost is simply the sum of the costs from all the map and the reduce tasks.

$$IOCost_{Job} = \begin{cases} IOCost_{AllMaps} & \text{if } pNumReducers = 0 \\ IOCost_{AllMaps} + IOCost_{AllReducers} & \text{if } pNumReducers > 0 \end{cases} \tag{96}$$

$$CPUCost_{Job} = \begin{cases} CPUCost_{AllMaps} & \text{if } pNumReducers = 0 \\ CPUCost_{AllMaps} + CPUCost_{AllReducers} & \text{if } pNumReducers > 0 \end{cases} \quad (97)$$

With appropriate system parameters that allow for equal comparisons among the I/O, CPU, and network costs, the overall cost is:

$$Cost_{Job} = IOCost_{Job} + CPUCost_{Job} + NETCost_{Job} \quad (98)$$