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.

1 Introduction

MapReduce is a relatively young framework—both a programming model and an associated run-time system—for large-scale data processing. *Hadoop* is the most popular open-source implementation of a MapReduce framework that follows the design laid out in the original paper. A combination of features contributes to Hadoop's increasing popularity, including fault tolerance, data-local scheduling, ability to operate in a heterogeneous environment, handling of straggler tasks, as well as a modular and customizable architecture.

The MapReduce programming model consists of a $map(k_1, v_1)$ function and a $reduce(k_2, list(v_2))$ function. Users can implement their own processing logic by specifying a customized *map*() and *reduce*() function written in a general-purpose language like Java or Python. The $map(k_1, v_1)$ function is invoked for every *key-value* pair $\langle k_1, v_1\rangle$ in the input data to output zero or more key-value pairs of the form $\langle k_2, v_2\rangle$ (see Figure 1). The $reduce(k_2, list(v_2))$ function is invoked for every unique key k_2 and corresponding values $list(v_2)$ in the map output. $reduce(k_2, list(v_2))$ outputs zero or more key-value pairs of the form $\langle k_3, v_3\rangle$. The MapReduce programming model also allows other functions such as (i) $partition(k_2)$, for controlling how the map output key-value pairs are partitioned among the reduce tasks, and (ii) $combine(k_2, list(v_2))$, for performing partial aggregation on the map side. The keys k_1, k_2, and k_3 as well as the values v_1, v_2, and v_3 can be of different and arbitrary types.

A Hadoop MapReduce cluster employs a master-slave architecture where one master node (called *JobTracker*) manages a number of slave nodes (called *TaskTrackers*). Figure 1 shows how a MapReduce job is executed on the cluster. Hadoop launches a MapReduce job by first splitting (logically) the input dataset into data *splits*. Each data split is then scheduled to one TaskTracker node and is processed by a map task. A *Task Scheduler* is responsible for scheduling the execution

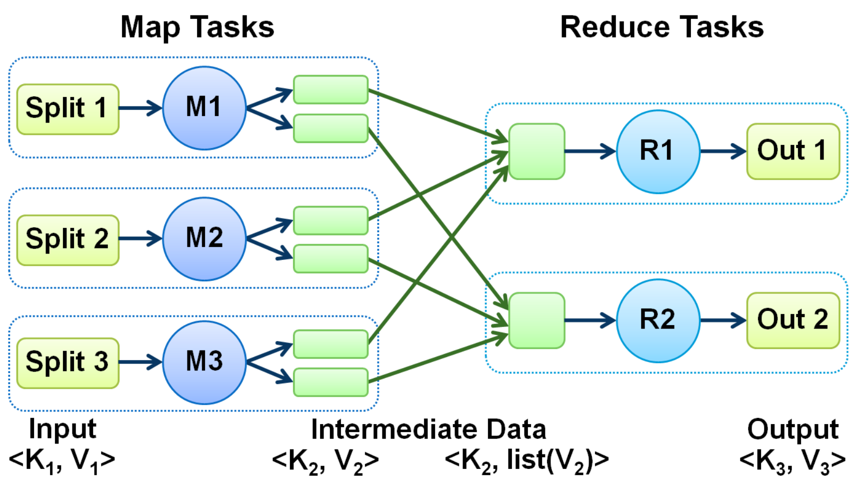

Figure 1: Execution of a MapReduce job.

of map tasks while taking data locality into account. Each TaskTracker has a predefined number of task execution *slots* for running map (reduce) tasks. If the job will execute more map (reduce) tasks than there are slots, then the map (reduce) tasks will run in multiple *waves*. When map tasks complete, the run-time system groups all intermediate key-value pairs using an external sort-merge algorithm. The intermediate data is then *shuffled* (i.e., transferred) to the TaskTrackers scheduled to run the reduce tasks. Finally, the reduce tasks will process the intermediate data to produce the results of the job.

As illustrated in Figure 2, the Map task execution is divided into five phases:

1. *Read*: Reading the input split from HDFS and creating the input key-value pairs (records).
2. *Map*: Executing the user-defined map function to generate the map-output data.
3. *Collect*: Partitioning and collecting the intermediate (map-output) data into a buffer before spilling.
4. *Spill*: Sorting, using the combine function if any, performing compression if specified, and finally writing to local disk to create *file spills*.
5. *Merge*: Merging the file spills into a single map output file. Merging might be performed in multiple rounds.

As illustrated in Figure 3, the Reduce Task is divided into four phases:

1. *Shuffle*: Transferring the intermediate data 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-defined reduce function to produce the final output data.

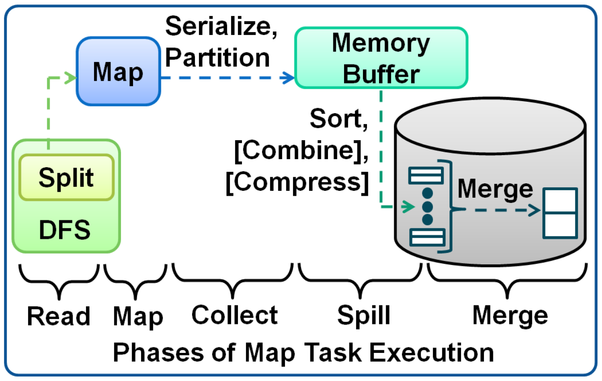

Figure 2: Execution of a map task showing the map-side phases.

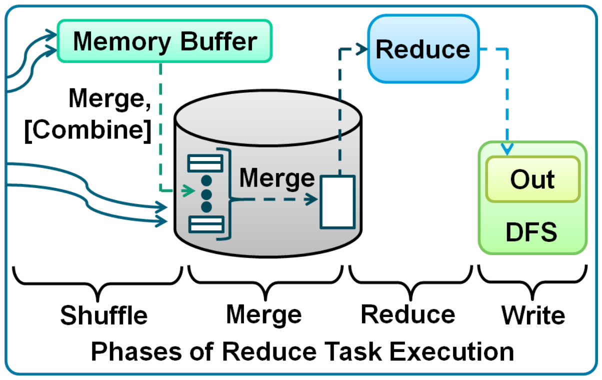

Figure 3: Execution of a reduce task showing the reduce-side phases.

4. *Write*: Compressing, if specified, and writing the final output to HDFS.

We model all task phases in order to accurately model the execution of a MapReduce job. We represent the execution of an arbitrary MapReduce job using a *job profile*, which is a concise statistical summary of MapReduce job execution. A job profile consists of dataflow and cost estimates for a MapReduce job j: dataflow estimates represent information regarding the number of bytes and key-value pairs processed during j's execution, while cost estimates represent resource usage and execution time.

For a map task, we model the Read and Map phases in Section 3, the Collect and Spill phases in Section 4, and the Merge phase in Section 5. For a reduce task, we model the Shuffle phase in Section 6, the Merge phase in Section 7, and the Reduce and Write phases in Section 8.

2 Preliminaries

The performance models calculate the dataflow and cost fields in a profile:

- *Dataflow fields* capture information about the amount of data, both in terms of bytes as well as records (key-value pairs), flowing through the different tasks and phases of a MapReduce job execution. Table 1 lists all the dataflow fields.
- *Cost fields* capture information about execution time at the level of tasks and phases within the tasks for a MapReduce job execution. Table 2 lists all the cost fields.

The inputs required by the models are estimated dataflow statistics fields, estimated cost statistics fields, as well as cluster-wide and job-level configuration parameter settings:

- *Dataflow Statistics fields* capture statistical information about the dataflow that is expected to remain unchanged across different executions of the MapReduce job unless the data distribution in the input dataset changes significantly across these executions. Table 3 lists all the dataflow statistics fields.

Table 1: Dataflow fields in the job profile. d, r, and c denote respectively input data properties, cluster resource properties, and configuration parameter settings.

Abbreviation	Profile Field (All fields, unless otherwise stated, represent information at the level of tasks)	Depends On		
		d	r	c
dNumMappers	Number of map tasks in the job	✓		✓
dNumReducers	Number of reduce tasks in the job			✓
dMapInRecs	Map input records	✓		✓
dMapInBytes	Map input bytes	✓		✓
dMapOutRecs	Map output records	✓		✓
dMapOutBytes	Map output bytes	✓		✓
dNumSpills	Number of spills	✓		✓
dSpillBufferRecs	Number of records in buffer per spill	✓		✓
dSpillBufferSize	Total size of records in buffer per spill	✓		✓
dSpillFileRecs	Number of records in spill file	✓		✓
dSpillFileSize	Size of a spill file	✓		✓
dNumRecsSpilled	Total spilled records	✓		✓
dNumMergePasses	Number of merge rounds	✓		✓
dShuffleSize	Total shuffle size	✓		✓
dReduceInGroups	Reduce input groups (unique keys)	✓		✓
dReduceInRecs	Reduce input records	✓		✓
dReduceInBytes	Reduce input bytes	✓		✓
dReduceOutRecs	Reduce output records	✓		✓
dReduceOutBytes	Reduce output bytes	✓		✓
dCombineInRecs	Combine input records	✓		✓
dCombineOutRecs	Combine output records	✓		✓
dLocalBytesRead	Bytes read from local file system	✓		✓
dLocalBytesWritten	Bytes written to local file system	✓		✓
dHdfsBytesRead	Bytes read from HDFS	✓		✓
dHdfsBytesWritten	Bytes written to HDFS	✓		✓

- *Cost Statistics fields* capture statistical information about execution time for a MapReduce job that is expected to remain unchanged across different executions of the job unless the cluster resources (e.g., CPU, I/O) available per node change. Table 4 lists all the cost statistics fields.

- *Configuration Parameters:* In Hadoop, a set of cluster-wide and job-level configuration parameter settings determine how a given MapReduce job will execute on a given cluster. Table 5 lists all relevant configuration parameters.

The performance models give good accuracy by capturing the subtleties of MapReduce job execution at the fine granularity of phases within map and reduce tasks. The current models were developed for Hadoop, but the overall approach applies to any MapReduce implementation.

To simplify the notation, we use the abbreviations contained in Tables 1, 2, 3, 4, and 5. Note the prefixes in all abbreviations used to distinguish where each abbreviation belongs to: d for dataset fields, c for cost fields, ds for data statistics fields, cs for cost statistics fields, p for Hadoop parameters, and t for temporary information not stored in the profile.

Table 2: Cost fields in the job profile. d, r, and c denote respectively input data properties, cluster resource properties, and configuration parameter settings.

Abbreviation	Profile Field (All fields represent information at the level of tasks)	Depends On		
		d	r	c
cSetupPhaseTime	Setup phase time in a task	✓	✓	✓
cCleanupPhaseTime	Cleanup phase time in a task	✓	✓	✓
cReadPhaseTime	Read phase time in the map task	✓	✓	✓
cMapPhaseTime	Map phase time in the map task	✓	✓	✓
cCollectPhaseTime	Collect phase time in the map task	✓	✓	✓
cSpillPhaseTime	Spill phase time in the map task	✓	✓	✓
cMergePhaseTime	Merge phase time in map/reduce task	✓	✓	✓
cShufflePhaseTime	Shuffle phase time in the reduce task	✓	✓	✓
cReducePhaseTime	Reduce phase time in the reduce task	✓	✓	✓
cWritePhaseTime	Write phase time in the reduce task	✓	✓	✓

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

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

If ($pUseCombine$ == FALSE)
  $dsCombineSizeSel = 1$
  $dsCombineRecsSel = 1$
  $csCombineCPUCost = 0$

If ($pIsInCompressed$ == FALSE)
  $dsInputCompressRatio = 1$
  $csInUncomprCPUCost = 0$

If ($pIsIntermCompressed$ == FALSE)
  $dsIntermCompressRatio = 1$
  $csIntermUncomCPUCost = 0$
  $csIntermComCPUCost = 0$

If ($pIsOutCompressed$ == FALSE)
  $dsOutCompressRatio = 1$
  $csOutComprCPUCost = 0$

Table 3: Dataflow statistics fields in the job profile. d, r, and c denote respectively input data properties, cluster resource properties, and configuration parameter settings.

Abbreviation	Profile Field (All fields represent information at the level of tasks)	Depends On		
		d	r	c
dsInputPairWidth	Width of input key-value pairs	✓		
dsRecsPerRedGroup	Number of records per reducer's group	✓		
dsMapSizeSel	Map selectivity in terms of size	✓		
dsMapRecsSel	Map selectivity in terms of records	✓		
dsReduceSizeSel	Reduce selectivity in terms of size	✓		
dsReduceRecsSel	Reduce selectivity in terms of records	✓		
dsCombineSizeSel	Combine selectivity in terms of size	✓		✓
dsCombineRecsSel	Combine selectivity in terms of records	✓		✓
dsInputCompressRatio	Input data compression ratio	✓		
dsIntermCompressRatio	Map output compression ratio	✓		✓
dsOutCompressRatio	Output compression ratio	✓		✓
dsStartupMem	Startup memory per task	✓		
dsSetupMem	Setup memory per task	✓		
dsCleanupMem	Cleanup memory per task	✓		
dsMemPerMapRec	Memory per map's record	✓		
dsMemPerRedRec	Memory per reduce's record	✓		

3 Modeling the Read and Map Phases in the Map Task

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

$$dMapInBytes = \frac{pSplitSize}{dsInputCompressRatio} \tag{2}$$

$$dMapInRecs = \frac{dMapInBytes}{dsInputPairWidth} \tag{3}$$

The cost of the Map Read phase is:

$$\begin{aligned} cReadPhaseTime = {} & pSplitSize \times csHdfsReadCost \\ & + pSplitSize \times csInUncomprCPUCost \end{aligned} \tag{4}$$

The cost of the Map phase is:

$$cMapPhaseTime = dMapInRecs \times csMapCPUCost \tag{5}$$

If the MapReduce 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.

$$dMapOutBytes = dMapInBytes \times dsMapSizeSel \tag{6}$$

$$dMapOutRecs = dMapInRecs \times dsMapRecsSel \tag{7}$$

Table 4: Cost statistics fields in the job profile. d, r, and c denote respectively input data properties, cluster resource properties, and configuration parameter settings.

Abbreviation	Profile Field (All fields represent information at the level of tasks)	Depends On		
		d	r	c
csHdfsReadCost	I/O cost for reading from HDFS per byte		✓	
csHdfsWriteCost	I/O cost for writing to HDFS per byte		✓	
csLocalIOReadCost	I/O cost for reading from local disk per byte		✓	
csLocalIOWriteCost	I/O cost for writing to local disk per byte		✓	
csNetworkCost	Cost for network transfer per byte		✓	
csMapCPUCost	CPU cost for executing the Mapper per record		✓	
csReduceCPUCost	CPU cost for executing the Reducer per record		✓	
csCombineCPUCost	CPU cost for executing the Combiner per record		✓	
csPartitionCPUCost	CPU cost for partitioning per record		✓	
csSerdeCPUCost	CPU cost for serializing/deserializing per record		✓	
csSortCPUCost	CPU cost for sorting per record		✓	
csMergeCPUCost	CPU cost for merging per record		✓	
csInUncomprCPUCost	CPU cost for uncompr/ing the input per byte		✓	
csIntermUncomCPUCost	CPU cost for uncompr/ing map output per byte		✓	✓
csIntermComCPUCost	CPU cost for compressing map output per byte		✓	✓
csOutComprCPUCost	CPU cost for compressing the output per byte		✓	✓
csSetupCPUCost	CPU cost of setting up a task		✓	
csCleanupCPUCost	CPU cost of cleaning up a task		✓	

The cost of the Map Write phase is:

$$\begin{aligned}&cWritePhaseTime = \\ &\qquad dMapOutBytes \times csOutComprCPUCost \\ &\qquad + dMapOutBytes \times dsOutCompressRatio \times csHdfsWriteCost \end{aligned} \tag{8}$$

4 Modeling the Collect and Spill Phases in the Map Task

The map function generates output key-value pairs (records) that are placed in the map-side memory buffer of size *pSortMB*. The amount of data output by the map function is calculated as follows:

$$dMapOutBytes = dMapInBytes \times dsMapSizeSel \tag{9}$$

$$dMapOutRecs = dMapInRecs \times dsMapRecsSel \tag{10}$$

$$tMapOutRecWidth = \frac{dMapOutBytes}{dMapOutRecs} \tag{11}$$

The map-side buffer consists of two disjoint parts: the *serialization* part that stores the serialized map-output records, and the *accounting* part that stores 16 bytes of metadata per record. When either of these two parts fills up to the threshold determined by *pSpillPerc*, the spill process begins. The maximum number of records in the serialization buffer before a spill is triggered is:

Table 5: A subset of cluster-wide and job-level Hadoop parameters.

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

$$tMaxSerRecs = \left\lfloor \frac{pSortMB \times 2^{20} \times (1 - pSortRecPerc) \times pSpillPerc}{tMapOutRecWidth} \right\rfloor \tag{12}$$

The maximum number of records in the accounting buffer before a spill is triggered is:

$$tMaxAccRecs = \left\lfloor \frac{pSortMB \times 2^{20} \times pSortRecPerc \times pSpillPerc}{16} \right\rfloor \tag{13}$$

Hence, the number of records in the buffer before a spill is:

$$dSpillBufferRecs = Min\{\ tMaxSerRecs,\ tMaxAccRecs\ ,\ dMapOutRecs\ \} \tag{14}$$

The size of the buffer included in a spill is:

$$dSpillBufferSize = dSpillBufferRecs \times tMapOutRecWidth \tag{15}$$

The overall number of spills is:

$$dNumSpills = \left\lceil \frac{dMapOutRecs}{dSpillBufferRecs} \right\rceil \tag{16}$$

The number of pairs and size of each spill file (i.e., the amount of data that will be written to disk) depend on the width of each record, the possible use of the Combiner, and the possible use of

compression. The Combiner's pair and size selectivities as well as the compression ratio are part of the Dataflow Statistics fields of the job profile. If a Combiner is not used, then the corresponding selectivities are set to 1 by default. If map output compression is disabled, then the compression ratio is set to 1.
Hence, the number of records and size of a spill file are:

$$dSpillFileRecs = dSpillBufferRecs \times dsCombineRecsSel \tag{17}$$

$$\begin{aligned} dSpillFileSize =& dSpillBufferSize \times dsCombineSizeSel \\ & \times dsIntermCompressRatio \end{aligned} \tag{18}$$

The total cost of the Map's Collect and Spill phases are:

$$\begin{aligned} cCollectPhaseTime =& dMapOutRecs \times csPartitionCPUCost \\ & + dMapOutRecs \times csSerdeCPUCost \end{aligned} \tag{19}$$

$$\begin{aligned} cSpillPhaseTime =\ & dNumSpills \times \\ & [\ dSpillBufferRecs \times \log_2(\frac{dSpillBufferRecs}{pNumReducers}) \times csSortCPUCost \\ & + dSpillBufferRecs \times csCombineCPUCost \\ & + dSpillBufferSize \times dsCombineSizeSel \times csIntermComCPUCost \\ & + dSpillFileSize \times csLocalIOWriteCost\ ] \end{aligned} \tag{20}$$

5 Modeling the Merge Phase in the Map Task

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 than one spill file is created. Multiple merge passes might occur, depending on the *pSortFactor* parameter. *pSortFactor* defines the maximum number of spill files that can be merged together to form a new single file. 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 $dNumSpills = 28$ and $pSortFactor = 10$. Then, 2 merge passes will be performed (merging 10 files each) to create 2 new files. This constitutes the first merge round. Then, the 2 new files will be merged together with the 8 original spill files to create the final output file, forming the 2^{nd} and final merge round.

The first merge pass is unique because Hadoop will calculate the optimal number of spill files to merge so that all other merge passes will merge exactly *pSortFactor* files. Notice how, in the example above, the final merge round merged exactly 10 files.

The final merge pass is also 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 round 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 spill files merged during the intermediate merge rounds as well as the total number of merge passes. Since the Reduce task also contains a similar Merge Phase, we define the following three methods to reuse later:

$$calcNumSpillsFirstPass(N, F) = \begin{cases} N & \text{, if } N \leq F \\ F & \text{, if } (N-1)\ MOD\ (F-1) = 0 \\ (N-1)\ MOD\ (F-1) + 1 & \text{, otherwise} \end{cases} \tag{21}$$

$$calcNumSpillsIntermMerge(N, F) = \begin{cases} 0 & \text{, if } N \leq F \\ P + \left\lfloor \frac{N-P}{F} \right\rfloor * F & \text{, if } N \leq F^2 \end{cases} \\ \text{, where } P = calcNumSpillsFirstPass(N, F) \tag{22}$$

$$calcNumSpillsFinalMerge(N, F) = \begin{cases} N & \text{, if } N \leq F \\ 1 + \left\lfloor \frac{N-P}{F} \right\rfloor + (N - S) & \text{, if } N \leq F^2 \end{cases} \\ \text{, where } P = calcNumSpillsFirstPass(N, F) \\ \text{, where } S = calcNumSpillsIntermMerge(N, F) \tag{23}$$

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

$$tNumSpillsFirstPass = calcNumSpillsFirstPass(dNumSpills, pSortFactor) \tag{24}$$

The number of spills read during intermediate merging is:

$$tNumSpillsIntermMerge = calcNumSpillsIntermMerge(dNumSpills, pSortFactor) \tag{25}$$

The total number of merge passes is:

$$dNumMergePasses = \begin{cases} 0 & \text{, if } dNumSpills = 1 \\ 1 & \text{, if } dNumSpills \leq pSortFactor \\ 2 + \left\lfloor \frac{dNumSpills - tNumSpillsFirstPass}{pSortFactor} \right\rfloor & \text{, if } dNumSpills \leq pSortFactor^2 \end{cases} \tag{26}$$

The number of spill files for the final merge round is:

$$tNumSpillsFinalMerge = calcNumSpillsFinalMerge(dNumSpills, pSortFactor) \tag{27}$$

As discussed earlier, the Combiner might be used during the final merge round. In this case, the size and record Combiner selectivities are:

$$\begin{aligned} \mathit{tUseCombInMerge} = & (\mathit{dNumSpills} > \mathit{1}) \text{ AND } (\mathit{pUseCombine}) \\ & \text{AND } (\mathit{tNumSpillsFinalMerge} \geq \mathit{pNumSpillsForComb}) \end{aligned} \tag{28}$$

$$\mathit{tMergeCombSizeSel} = \begin{cases} dsCombineSizeSel & \text{, if } tUseCombInMerge \\ 1 & \text{, otherwise} \end{cases} \tag{29}$$

$$\mathit{tMergeCombRecsSel} = \begin{cases} dsCombineRecsSel & \text{, if } tUseCombInMerge \\ 1 & \text{, otherwise} \end{cases} \tag{30}$$

The total number of records spilled equals the sum of (i) the records spilled during the Spill phase, (ii) the number of records that participated in the intermediate merge rounds, and (iii) the number of records spilled during the final merge round.

$$\begin{aligned} \mathit{dNumRecsSpilled} = & \mathit{dSpillFileRecs} \times \mathit{dNumSpills} \\ & + \mathit{dSpillFileRecs} \times \mathit{tNumSpillsIntermMerge} \\ & + \mathit{dSpillFileRecs} \times \mathit{dNumSpills} \times \mathit{tMergeCombRecsSel} \end{aligned} \tag{31}$$

The final size and number of records for the final map output data are:

$$\mathit{tIntermDataSize} = \mathit{dNumSpills} \times \mathit{dSpillFileSize} \times \mathit{tMergeCombSizeSel} \tag{32}$$

$$\mathit{tIntermDataRecs} = \mathit{dNumSpills} \times \mathit{dSpillFileRecs} \times \mathit{tMergeCombRecsSel} \tag{33}$$

The total cost of the Merge phase is divided into the cost for performing the intermediate merge rounds and the cost for performing the final merge round.

$$\begin{aligned} \mathit{tIntermMergeTime} = & tNumSpillsIntermMerge \times \\ & [\ \mathit{dSpillFileSize} \times \mathit{csLocalIOReadCost} \\ & \quad + \mathit{dSpillFileSize} \times \mathit{csIntermUncomCPUCost} \\ & \quad + \mathit{dSpillFileRecs} \times \mathit{csMergeCPUCost} \\ & \quad + \frac{\mathit{dSpillFileSize}}{\mathit{dsIntermCompressRatio}} \times \mathit{csIntermComCPUCost} \\ & \quad + \mathit{dSpillFileSize} \times \mathit{csLocalIOWriteCost}\ ] \end{aligned} \tag{34}$$

$$\begin{aligned} \mathit{tFinalMergeTime} = & \mathit{dNumSpills} \times \\ & [\ \mathit{dSpillFileSize} \times \mathit{csLocalIOReadCost} \\ & \quad + \mathit{dSpillFileSize} \times \mathit{csIntermUncomCPUCost} \\ & \quad + \mathit{dSpillFileRecs} \times \mathit{csMergeCPUCost} \\ & \quad + \mathit{dSpillFileRecs} \times \mathit{csCombineCPUCost}\ ] \\ & + \frac{\mathit{tIntermDataSize}}{\mathit{dsIntermCompressRatio}} \times \mathit{csIntermComCPUCost} \\ & + \mathit{tIntermDataSize} \times \mathit{csLocalIOWriteCost} \end{aligned} \tag{35}$$

$$cMergePhaseTime = tIntermMergeTime + tFinalMergeTime \tag{36}$$

6 Modeling the Shuffle Phase in the Reduce Task

In the Shuffle phase, the framework fetches the relevant map output partition from each mapper (called a *map segment*) and copies it to the reducer's node. If the map output is compressed, Hadoop will uncompress it after the transfer as part of the shuffling process. Assuming a uniform distribution of the map output to all reducers, the size and number of records for each map segment that reaches the reduce side are:

$$tSegmentComprSize = \frac{tIntermDataSize}{pNumReducers} \tag{37}$$

$$tSegmentUncomprSize = \frac{tSegmentComprSize}{dsIntermCompressRatio} \tag{38}$$

$$tSegmentRecs = \frac{tIntermDataRecs}{pNumReducers} \tag{39}$$

where *tIntermDataSize* and *tIntermDataRecs* are the size and number of records produced as intermediate output by a single mapper (see Section 5). A more complex model can be used to account for the presence of skew. The data fetched to a single reducer will be:

$$dShuffleSize = pNumMappers * tSegmentComprSize \tag{40}$$

$$dShuffleRecs = pNumMappers * tSegmentRecs \tag{41}$$

The intermediate data is transfered and placed in an in-memory *shuffle buffer* with a size proportional to the parameter *pShuffleInBufPerc*:

$$tShuffleBufferSize = pShuffleInBufPerc \times pTaskMem \tag{42}$$

However, when the segment size is greater than 25% times the *tShuffleBufferSize*, the segment will get copied directly to local disk instead of the in-memory shuffle buffer. We consider these two cases separately.

Case 1: $tSegmentUncomprSize < 0.25 \times tShuffleBufferSize$
The map segments are transfered, uncompressed if needed, and placed into the shuffle buffer. When either (a) the amount of data placed in the shuffle buffer reaches a threshold size determined by the *pShuffleMergePerc* parameter or (b) the number of segments becomes greater than the *pInMemMergeThr* parameter, the segments are merged and spilled to disk creating a new local file (called *shuffle file*). The size threshold to begin merging is:

$$tMergeSizeThr = pShuffleMergePerc \times tShuffleBufferSize \tag{43}$$

The number of map segments merged into a single shuffle file is:

$$tNumSegInShuffleFile = \frac{tMergeSizeThr}{tSegmentUncomprSize} \tag{44}$$

If ($\lceil tNumSegInShuffleFile \rceil \times tSegmentUncomprSize \leq tShuffleBufferSize$)
  $tNumSegInShuffleFile = \lceil tNumSegInShuffleFile \rceil$
else
  $tNumSegInShuffleFile = \lfloor tNumSegInShuffleFile \rfloor$

If ($tNumSegInShuffleFile > pInMemMergeThr$)
  $tNumSegInShuffleFile = pInMemMergeThr$ (45)

If a Combiner is specified, then it is applied during the merging. If compression is enabled, then the (uncompressed) map segments are compressed after merging and before written to disk. Note also that if *numMappers* < *tNumSegInShuffleFile*, then merging will not happen. The size and number of records in a single shuffle file is:

$$tShuffleFileSize = \\ tNumSegInShuffleFile \times tSegmentComprSize \times dsCombineSizeSel \tag{46}$$

$$tShuffleFileRecs = \\ tNumSegInShuffleFile \times tSegmentRecs \times dsCombineRecsSel \tag{47}$$

$$tNumShuffleFiles = \left\lfloor \frac{pNumMappers}{tNumSegInShuffleFile} \right\rfloor \tag{48}$$

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

$$tNumSegmentsInMem = pNumMappers \text{ MOD } tNumSegInShuffleFile \tag{49}$$

Case 2: *tSegmentUncomprSize* $\geq$ *0.25* $\times$ *tShuffleBufferSize*
When a map segment is transfered directly to local disk, it becomes equivalent to a shuffle file. Hence, the corresponding temporary variables introduced in Case 1 above are:

$$tNumSegInShuffleFile = 1 \tag{50}$$

$$tShuffleFileSize = tSegmentComprSize \tag{51}$$

$$tShuffleFileRecs = tSegmentRecs \tag{52}$$

$$tNumShuffleFiles = pNumMappers \tag{53}$$

$$tNumSegmentsInMem = 0 \tag{54}$$

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

$$tNumShuffleMerges = \\ \begin{cases} 0 \text{ , if } tNumShuffleFiles < 2 \times pSortFactor - 1 \\ \left\lfloor \frac{tNumShuffleFiles - 2 \times pSortFactor + 1}{pSortFactor} \right\rfloor + 1 \text{ , otherwise} \end{cases} \tag{55}$$

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

$$tNumMergShufFiles = tNumShuffleMerges \tag{56}$$

$$tMergShufFileSize = pSortFactor \times tShuffleFileSize \tag{57}$$

$$tMergShufFileRecs = pSortFactor \times tShuffleFileRecs \tag{58}$$

$$\begin{aligned} tNumUnmergShufFiles = & tNumShuffleFiles \\ & -(pSortFactor \times tNumShuffleMerges) \end{aligned} \tag{59}$$

$$tUnmergShufFileSize = tShuffleFileSize \tag{60}$$

$$tUnmergShufFileRecs = tShuffleFileRecs \tag{61}$$

The total cost of the Shuffle phase includes cost for the network transfer, cost for any in-memory merging, and cost for any on-disk merging, as described above.

$$\begin{aligned} & tInMemMergeTime = \\ & \quad I(tSegmentUncomprSize < 0.25 \times tShuffleBufferSize) \times \\ & \quad [\ dShuffleSize \times csIntermUncomCPUCost \\ & \quad\ + tNumShuffleFiles \times tShuffleFileRecs \times csMergeCPUCost \\ & \quad\ + tNumShuffleFiles \times tShuffleFileRecs \times csCombineCPUCost \\ & \quad\ + tNumShuffleFiles \times \frac{tShuffleFileSize}{dsIntermCompressRatio} \times csIntermComCPUCost\ ] \\ & \quad + tNumShuffleFiles \times tShuffleFileSize \times csLocalIOWriteCost \end{aligned} \tag{62}$$

$$\begin{aligned} & tOnDiskMergeTime = tNumMergShufFiles \times \\ & \quad [\ tMergShufFileSize \times csLocalIOReadCost \\ & \quad\ + tMergShufFileSize \times csIntermUncomCPUCost \\ & \quad\ + tMergShufFileRecs \times csMergeCPUCost \\ & \quad\ + \frac{tMergShufFileSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\ & \quad\ + tMergShufFileSize \times csLocalIOWriteCost\ ] \end{aligned} \tag{63}$$

$$\begin{aligned} cShufflePhaseTime = & dShuffleSize \times csNetworkCost \\ & + tInMemMergeTime \\ & + tOnDiskMergeTime \end{aligned} \tag{64}$$

7 Modeling the Merge Phase in the Reduce Task

After all map output data has been successful transfered to the Reduce node, the Merge phase[1] begins. During this phase, the map output data is merged into a single stream that is fed to the reduce function for processing. Similar to the Map's Merge phase (see Section 5), the Reduce's Merge phase may occur it multiple rounds. However, instead of creating a single output file during the final merge round, the data is sent directly to the reduce function.

The Shuffle phase may produce (i) a set of merged shuffle files on disk, (ii) a set of unmerged shuffle files on disk, and (iii) a set of map segments in memory. The total number of shuffle files on disk is:

$$tNumShufFilesOnDisk = tNumMergShufFiles + tNumUnmergShufFiles \tag{65}$$

The merging in this phase is done in three steps.

Step 1: Some map segments are marked for eviction from memory in order to satisfy a memory constraint enforced by the *pReducerInBufPerc* parameter, which specifies the amount of memory allowed to be occupied by the map segments before the reduce function begins.

$$tMaxSegmentBufferSize = pReducerInBufPerc \times pTaskMem \tag{66}$$

The amount of memory currently occupied by map segments is:

$$tCurrSegmentBufferSize = tNumSegmentsInMem \times tSegmentUncomprSize \tag{67}$$

Hence, the number of map segments to evict from, and retain in, memory are:

$$\begin{aligned}
&\text{If } (tCurrSegmentBufferSize > tMaxSegmentBufferSize)\\
&\quad tNumSegmentsEvicted = \left\lceil \frac{tCurrSegmentBufferSize - tMaxSegmentBufferSize}{tSegmentUncomprSize} \right\rceil\\
&\text{else}\\
&\quad tNumSegmentsEvicted = 0
\end{aligned} \tag{68}$$

$$tNumSegmentsRemainMem = tNumSegmentsInMem - tNumSegmentsEvicted \tag{69}$$

If the number of existing shuffle files on disk is less than *pSortFactor*, then the map segments marked for eviction will be merged into a single shuffle file on disk. Otherwise, the map segments marked for eviction are left to be merged with the shuffle files on disk during Step 2 (i.e., Step 1 does not happen).

[1]The Merge phase in the Reduce task is also called "Sort phase" in the literature, even though no sorting occurs.

$$\begin{aligned}
&\text{If } (\mathit{tNumShufFilesOnDisk} < \mathit{pSortFactor})\\
&\quad \mathit{tNumShufFilesFromMem} = \mathit{1}\\
&\quad \mathit{tShufFilesFromMemSize} = \mathit{tNumSegmentsEvicted} \times \mathit{tSegmentComprSize}\\
&\quad \mathit{tShufFilesFromMemRecs} = \mathit{tNumSegmentsEvicted} \times \mathit{tSegmentRecs}\\
&\quad \mathit{tStep1MergingSize} = \mathit{tShufFilesFromMemSize}\\
&\quad \mathit{tStep1MergingRecs} = \mathit{tShufFilesFromMemRecs}\\
&\text{else}\\
&\quad \mathit{tNumShufFilesFromMem} = \mathit{tNumSegmentsEvicted}\\
&\quad \mathit{tShufFilesFromMemSize} = \mathit{tSegmentComprSize}\\
&\quad \mathit{tShufFilesFromMemRecs} = \mathit{tSegmentRecs}\\
&\quad \mathit{tStep1MergingSize} = \mathit{0}\\
&\quad \mathit{tStep1MergingRecs} = \mathit{0}
\end{aligned} \tag{70}$$

The total cost of Step 1 (which could be zero) is:

$$\begin{aligned}
\mathit{cStep1Time} =& \mathit{tStep1MergingRecs} \times \mathit{csMergeCPUCost}\\
&+\frac{\mathit{tStep1MergingSize}}{\mathit{dsIntermCompressRatio}} \times \mathit{csIntermComCPUCost}\\
&+\mathit{tStep1MergingSize} \times \mathit{csLocalIOWriteCost}
\end{aligned} \tag{71}$$

Step 2: Any shuffle files that reside on disk will go through a merging phase in multiple merge rounds (similar to the process in Section 5). This step will happen only if there exists at least one shuffle file on disk. The total number of files to merge during Step 2 is:

$$\begin{aligned}
&\mathit{tFilesToMergeStep2} =\\
&\qquad \mathit{tNumShufFilesOnDisk} + \mathit{tNumShufFilesFromMem}
\end{aligned} \tag{72}$$

The number of intermediate reads (and writes) are:

$$\begin{aligned}
&\mathit{tIntermMergeReads2} =\\
&\qquad \mathit{calcNumSpillsIntermMerge}(\mathit{tFilesToMergeStep2}, \mathit{pSortFactor})
\end{aligned} \tag{73}$$

The main difference from Section 5 is that the merged files in this case have different sizes. We account for the different sizes by attributing merging costs proportionally. Hence, the total size and number of records involved in the merging process during Step 2 are:

$$\begin{aligned}
\mathit{tStep2MergingSize} =& \frac{\mathit{tIntermMergeReads2}}{\mathit{tFilesToMergeStep2}} \times\\
&[\ \mathit{tNumMergShufFiles} \times \mathit{tMergShufFileSize}\\
&\ +\mathit{tNumUnmergShufFiles} \times \mathit{tUnmergShufFileSize}\\
&\ +\mathit{tNumShufFilesFromMem} \times \mathit{tShufFilesFromMemSize}]
\end{aligned} \tag{74}$$

$$\begin{aligned}
tStep2MergingRecs = & \frac{tIntermMergeReads2}{tFilesToMergeStep2} \times \\
& [\ tNumMergShufFiles \times tMergShufFileRecs \\
& \ +tNumUnmergShufFiles \times tUnmergShufFileRecs \\
& \ +tNumShufFilesFromMem \times tShufFilesFromMemRecs] \qquad (75)
\end{aligned}$$

The total cost of Step 2 (which could also be zero) is:

$$\begin{aligned}
cStep2Time = & tStep2MergingSize \times csLocalIOReadCost \\
& +tStep2MergingSize \times cIntermUnomprCPUCost \\
& +tStep2MergingRecs \times csMergeCPUCost \\
& +\frac{tStep2MergingSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\
& +tStep2MergingSize \times csLocalIOWriteCost \qquad (76)
\end{aligned}$$

Step 3: All files on disk and in memory will be merged together. The process is identical to step 2 above. The total number of files to merge during Step 3 is:

$$\begin{aligned}
tFilesToMergeStep3 = & tNumSegmentsRemainMem \\
& +calcNumSpillsFinalMerge(tFilesToMergeStep2, pSortFactor) \qquad (77)
\end{aligned}$$

The number of intermediate reads (and writes) are:

$$\begin{aligned}
& tIntermMergeReads3 = \\
& \quad calcNumSpillsIntermMerge(tFilesToMergeStep3, pSortFactor) \qquad (78)
\end{aligned}$$

Hence, the total size and number of records involved in the merging process during Step 3 are:

$$tStep3MergingSize = \frac{tIntermMergeReads3}{tFilesToMergeStep3} \times dShuffleSize \qquad (79)$$

$$tStep3MergingRecs = \frac{tIntermMergeReads3}{tFilesToMergeStep3} \times dShuffleRecs \qquad (80)$$

The total cost of Step 3 (which could also be zero) is:

$$\begin{aligned}
cStep3Time = & tStep3MergingSize \times csLocalIOReadCost \\
& +tStep3MergingSize \times cIntermUnomprCPUCost \\
& +tStep3MergingRecs \times csMergeCPUCost \\
& +\frac{tStep3MergingSize}{dsIntermCompressRatio} \times csIntermComCPUCost \\
& +tStep3MergingSize \times csLocalIOWriteCost \qquad (81)
\end{aligned}$$

The total cost of the Merge phase is:

$$cMergePhaseTime = cStep1Time + cStep2Time + cStep3Time \qquad (82)$$

8 Modeling the Reduce and Write Phases in the Reduce Task

Finally, the user-defined reduce function will processed the merged intermediate data to produce the final output that will be written to HDFS. The size and number of records processed by the reduce function is:

$$\begin{aligned} \mathit{dReduceInBytes} =& \frac{\mathit{tNumShuffleFiles} \times \mathit{tShuffleFileSize}}{\mathit{dsIntermCompressRatio}} \\ &+ \frac{\mathit{tNumSegmentsInMem} \times \mathit{tSegmentComprSize}}{\mathit{dsIntermCompressRatio}} \end{aligned} \tag{83}$$

$$\begin{aligned} \mathit{dReduceInRecs} =& \mathit{tNumShuffleFiles} \times \mathit{tShuffleFileRecs} \\ &+ \mathit{tNumSegmentsInMem} \times \mathit{tSegmentRecs} \end{aligned} \tag{84}$$

The size and number of records produce by the reduce function is:

$$\mathit{dReduceOutBytes} = \mathit{dReduceInBytes} \times \mathit{dsReduceSizeSel} \tag{85}$$

$$\mathit{dReduceOutRecs} = \mathit{dReduceInRecs} \times \mathit{dsReduceRecsSel} \tag{86}$$

The input data to the reduce function may reside in both memory and disk, as produced by the Shuffle and Merge phases.

$$\begin{aligned} \mathit{tInRedFromDiskSize} =& \mathit{tNumMergShufFiles} \times \mathit{tMergShufFileSize} \\ &+ \mathit{tNumUnmergShufFiles} \times \mathit{tUnmergShufFileSize} \\ &+ \mathit{tNumShufFilesFromMem} \times \mathit{tShufFilesFromMemSize} \end{aligned} \tag{87}$$

The total cost of the Reduce phase is:

$$\begin{aligned} \mathit{cReducePhaseTime} =& \mathit{tInRedFromDiskSize} \times \mathit{csLocalIOReadCost} \\ &+ \mathit{tInRedFromDiskSize} \times \mathit{cIntermUncompCPUCost} \\ &+ \mathit{dReduceInRecs} \times \mathit{csReduceCPUCost} \end{aligned} \tag{88}$$

The total cost of the Write phase is:

$$\begin{aligned} &\mathit{cWritePhaseTime} = \\ &\qquad \mathit{dReduceOutBytes} \times \mathit{csOutComprCPUCost} \\ &\qquad + \mathit{dReduceOutBytes} \times \mathit{dsOutCompressRatio} \times \mathit{csHdfsWriteCost} \end{aligned} \tag{89}$$

8.1 Modeling the Overall MapReduce Job Execution

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

Simulation involves scheduling and simulating the execution of individual tasks on a virtual cluster. One simple way for estimating the cost for each task is to sum up the cost fields estimated in Sections 3–8. The overall cost for a single map task is:

$$totalMapTime = \begin{cases} cReadPhaseTime + cMapPhaseTime + cWritePhaseTime & \text{, if } pNumReducers = 0 \\ cReadPhaseTime + cMapPhaseTime + cCollectPhaseTime & \\ +cSpillPhaseTime + cMergePhaseTime & \text{, if } pNumReducers > 0 \end{cases} \tag{90}$$

The overall cost for a single reduce task is:

$$\begin{aligned} totalReduceTime = & cShufflePhaseTime + cMergePhaseTime \\ & + cReducePhaseTime + cSpillPhaseTime \end{aligned} \tag{91}$$

The second approach for calculating the total job execution time involves using the following analytical estimates:

$$totalMapsTime = \frac{pNumMappers \times totalMapTime}{pNumNodes \times pMaxMapsPerNode} \tag{92}$$

$$totalReducesTime = \frac{pNumReducers \times totalReduceTime}{pNumNodes \times pMaxRedPerNode} \tag{93}$$

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

$$totalJobTime = \begin{cases} totalMapsTime & \text{, if } pNumReducers = 0 \\ totalMapsTime + totalReducesTime & \text{, if } pNumReducers > 0 \end{cases} \tag{94}$$