Spark Performance Models

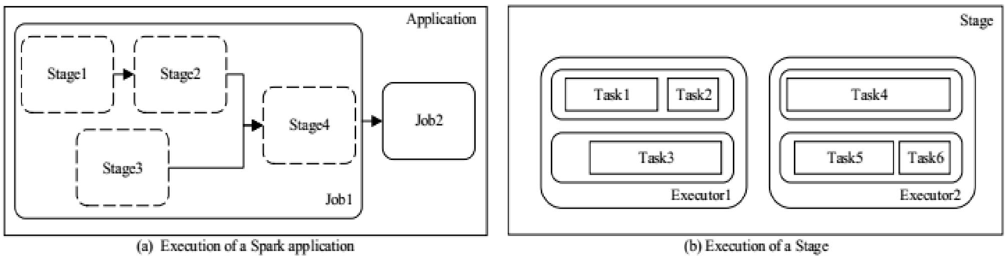
Abstract

Spark 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 Spark Application. The model describe dataflow and cost information at the fine granularity of phases within the tasks in stage of a job execution. The models can be used to estimate the performance of spark applications as well as to find the optimal configuration settings to use when running the applications.

1 Introduction

Spark is a fast and general-purpose cluster computing system. Spark provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. It was developed in response to limitations in the MapReduce cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs. Spark's RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory.

The Spark programming model is RDD. There are some RDD features. Read Only collection of objects spread across a cluster. RDDs are created by input data or built through parallel transformations from other RDD(map, filter, reduceByKey, etc.)Automatically rebuilt on failure using lineage. Controllable persistence (Ram, HDFS, etc.)There are two type of operation can be applied to RDD. Transformation, which applied user-defined function to process the data in RDD and return a new RDD representing the results. Action, which return a value to the driver program after running a computation on the dataset. The job didn’t compute until action is called. And the job execution is divided into many stages by the transformation which lead to shuffle.



(a) Execution of a Spark application

(b) Execution of a Stage

Figure 1: Execution of a Spark Application

A Spark cluster employs a master-slave architecture where one master node(called Driver) manages a number of slave nodes(called Worker). Figure 1 show how a Spark Application is executed on the cluster. Spark launch a job by first splitting the input dataset into data splits. Each data split is then scheduled to one executor of one Worker and is processed by a task. A Task Scheduler is responsible for scheduling the execution of tasks while taking data locality into account. Each executor has a predefined number of task execution threads for running tasks. If the stage will execute more tasks than there are threads, then the tasks will run in multiple waves. When compute phase in one stage is completed, If the stage is a ShuffleMapStage, the output data of each task are partitioned among the next stage's tasks, may be combine and sort, for performing partial aggregation and sort on the map side. Then intermediate data is then shuffled to the executors to run next stage's tasks. If the stage if a ResultStage, which performs action, will send the output data to Driver Program.

As illustrated in Figure2, task execution is divided into five phases. Stage consists of some phases.

1 *Read*: Reading the input split from HDFS and creating the input records.

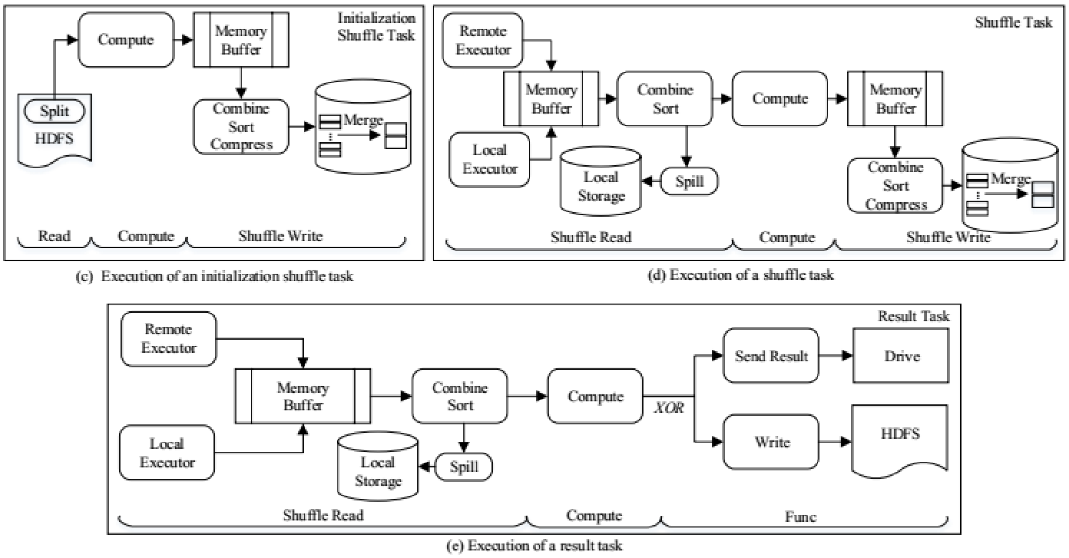
2 *ShuffleRead*: Transferring the intermediate data(last stage ShuffleWrite data) from the executors to a executor and decompressing is needed. Combine or Sort (defined by the transformation lead to shuffle)will occur during this phase, Spill and Merge may appear during the Combine or Sort.

3 *Compute*: Executing transformations and action to generate the output data.

4 *ShuffleWrite*: Partitioning , Combine and Sort if transformation define mapsideCombine(Spill and Merge may appear) , the compute output data , and finally write the intermediate data to file in local disk, performing the compression if specified.

5 *Func*: Send the final result to Driver Program.

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



(c) Execution of an initialization shuffle task

(d) Execution of a shuffle task

(e) Execution of a result task

Figure 2:Execution of different type task showing several phases.

2 Preliminaries

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

- Dataflow fields capture information about the amount of the data, both in terms of bytes as well as records(key-value pairs), flowing through the different tasks and phases of a Spark stage 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 Spark stage execution. Table2 lists all the cost fields.

The input required by the models are estimate data statistics fields, estimated cost statistics fields, as well as cluster-wide and application-level parameter settings:

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

- Cost Statistics fields capture statistical information about execution time for a Spark Stage 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 Spark, a set of cluster-wide and application-level configuration parameter settings determine how a given Spark Application will execute on a given cluster. Table 5 lists all relevant configuration parameters.

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 Spark parameters, and t for temporary information not stored in the profile.

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

Abbreviation	Profile Field	Depends on		
	(all field represent information at the level of tasks)	d	r	c
dReadInRecs	Read in records	Y		Y
dReadInBytes	Read in bytes	Y		Y
dDenFileJarSize	Dependency file and jar size	Y		
dComputeOutRecs	Compute out records	Y		Y
dComputeOutBytes	Compute out Bytes	Y		Y
dSpillFileSize	Spill file size	Y		Y
dPerSpillRecs	Number of records in the buffer per spill	Y		Y
dNumSpills	Number of spills	Y		Y
dSpillBuffer	Buffer size per spill	Y		Y
dSpillCombRecs	Number of records total spill	Y		Y
dMergeCombRecs	Number of records after combine	Y		Y
dWriteRecs	Number of records written to local file system	Y		Y
dWriteBytes	Bytes written to local file system	Y		Y
dRemoteReadBytes	Bytes ShuffleRead from remote	Y		Y
dLocalReadBytes	Bytes ShuffleRead from local	Y		Y
dShuffleReadRecs	Number of records ShuffleRead	Y		Y

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

$$Identity\ Function \quad I(x)=\left\{\begin{array}{l}1,\ if\ \ x\,exists\,or\,equal\,true\\ 0,otherwise\end{array}\right\} \tag{1}$$

different *pCompressor* decide different *dsIntermCompressRadio , csIntermComCPUCost, csIntermUncomCPUcost*

different *pSerializer* decide different *dsSerializeRadio* and *csSerdeBytesCPUCost*

Note that Aggregator compress spilled data or not is determined by pIsSpillCompress. While ExternalSorter is determined by pIsShuffleCompress. So we use the *tSpill-* and *tShuffle-* represent this situation.

$$\begin{array}{l}If\ (pIsShuffleCompress == true)\\ \quad tShuffleCompress\,\mathrm{R}\,adio = dsIntermCompressRadio\\ \quad tShuffleComCPU\ \mathrm{Cos}\,t = csIntermComCPU\ \mathrm{Cos}\,t\\ \quad tShuffleUncomCPU\ \mathrm{Cos}\,t = csIntermUncomCPU\ \mathrm{Cos}\,t\end{array}$$

$$\begin{array}{l}If\ (pIsShuffleCompress ==\ false)\\ \quad tShuffleCompressRadio = 1\\ \quad tShuffleComCPU\ \mathrm{Cos}\,t = 0\\ \quad tShuffleUncomCPU\ \mathrm{Cos}\,t = 0\end{array}$$

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

Abbreviation	Profile Field (all field represent information at the level of tasks)	Depends on		
		d	r	c
cSetupPhaseTime	Setup phase time in a task	Y	Y	Y
cCleanupPhaseTime	Cleanup phase time in a task	Y	Y	Y
cReadPhaseTime	Read phase time in a task	Y	Y	Y
cShuffleReadPhaseTime	ShuffleRead phase time in a task	Y	Y	Y
cComputePhaseTime	Compute phase time in a task	Y	Y	Y
cShuffleWritePhaseTime	ShuffleWrite phase time in a task	Y	Y	Y
cFuncPhaseTime	Func phase time in a task	Y	Y	Y

$$\begin{aligned}
&If\ (pIsSpillCompress == true)\\
&\quad tSpillCompress\,\mathrm{R}\,adio = dsIntermCompressRadio\\
&\quad tSpillComCPU\,\mathrm{Cos}\,t = csIntermComCPU\,\mathrm{Cos}\,t\\
&\quad tSpillUncomCPU\,\mathrm{Cos}\,t = csIntermUncomCPU\,\mathrm{Cos}\,t
\end{aligned}$$

$$\begin{aligned}
&If\ (pIsSpillCompress == \ false)\\
&\quad tSpillCompressRadio = 1\\
&\quad tSpillComCPU\,\mathrm{Cos}\,t = 0\\
&\quad tSpillUncomCPU\,\mathrm{Cos}\,t = 0
\end{aligned}$$

We use SCW to represent the time one byte to be serialized, compressed and wrote to local disk, RCS to represent the time one byte to be read, uncompressed, deserialized into memory.

$$\begin{aligned}
tShuffleSCW &= csSerdeBytesCPUCost\\
&\quad + dsSerializeRadio \times (tShuffleComCPU\,\mathrm{Cos}\,t\\
&\quad + tShuffleCompressRadio \times csLocalIOWriteCost)
\end{aligned} \tag{2}$$

$$\begin{aligned}
tShuffleRCS &= csLocalIOReadCost + tShuffleUncomCPU\,\mathrm{Cos}\,t\\
&\qquad + \frac{csSerdeBytesCPUCost}{tShuffleCompressRadio}
\end{aligned} \tag{3}$$

$$\begin{aligned}
tSpillSCW &= csSerdeBytesCPUCost\\
&\quad + dsSerializeRadio \times (tSpillComCPU\,\mathrm{Cos}\,t\\
&\quad + tSpillCompress\,\mathrm{R}\,adio \times csLocalIOWriteCost)
\end{aligned} \tag{4}$$

$$\begin{aligned}
tSpillRCS &= csLocalIOReadCost + tSpillUncomCPU\,\mathrm{Cos}\,t\\
&\qquad + \frac{csSerdeBytesCPUCost}{tSpillCompress\,\mathrm{R}\,adio}
\end{aligned} \tag{5}$$

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

Abbreviation	Profile Field (all field represent information at the level of tasks)	Depends on		
		d	r	c
dsReadInPairWidth	Width of input key-value pairs	Y		
dsComputeSizeSel	Compute selectivity in terms of size	Y		
dsComputeRecsSel	Compute selectivity in terms of records	Y		
dsWriteCombSizeSel	Combine selectivity in terms of size in the ShuffleRead phase	Y		Y
dsWriteCombRecsSel	Combine selectivity in terms of records in the ShuffleRead phase	Y		Y
dsWriteMergeSizeSel	Merge selectivity in terms of size in the ShuffleRead phase	Y		
dsWriteMergeRecsSel	Merge selectivity in terms of records in the ShuffleRead phase	Y		
dsReadCombSizeSel	Combine selectivity in terms of size in the ShuffleWrite phase	Y		Y
dsReadCombRecsSel	Combine selectivity in terms of records in the ShuffleWrite phase	Y		Y
dsReadMergeSizeSel	Merge selectivity in terms of size in the ShuffleWrite phase	Y		
dsReadMergeRecsSel	Merge selectivity in terms of records in the ShuffleWrite phase	Y		
dsSerializeRadio	Serialize selectivity in terms of bytes	Y		Y
dsReadSpillPairWidth	Memory per ShuffleRead record	Y		
dsWriteSpillPairWidth	Memory per ShuffleWrite record	Y		
dsIntermCompressRadio	Compress radio	Y		Y

3 Modeling the Read Phases

During this phase, the input split is read and the input records are created and passed as input to the user-defined transformation.

$$d\,\mathrm{Re}\,adInBytes = pSplitSize \tag{6}$$

$$d\,\mathrm{Re}\,adIn\,\mathrm{Re}\,cs = \frac{d\,\mathrm{Re}\,adInBytes}{ds\,\mathrm{Re}\,adInPairWidth} \tag{7}$$

$$tComputeInBytes = d\,\mathrm{Re}\,adInBytes \tag{8}$$

$$tComputeIn\,\mathrm{Re}\,cs = d\,\mathrm{Re}\,adIn\,\mathrm{Re}\,cs \tag{9}$$

The cost of the Read Phase is:

$$c\,\mathrm{Re}\,adPhaseTime = csHdfsReadCost \times d\,\mathrm{Re}\,adInBytes \tag{10}$$

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

Abbreviation	Profile Field (all field represent information at the level of tasks)	Depends on		
		d	r	c
csHdfsReadCost	I/O cost for reading from HDFS per byte		Y	
csHdfsWriteCost	I/O cost for writing to HDFS per byte		Y	
csLocalIOReadCost	I/O cost for reading local disk per byte		Y	
csLocalIOWriteCost	I/O cost for writing to local disk per byte		Y	
csNetworkCost	Cost for network transfers per byte		Y	
csComputeCPUCost	CPU cost for computing per record		Y	
csSpillCPUCost	CPU cost for spilling per record		Y	
csSpillCombCPUCost	CPU cost for combining per record		Y	
csMergeCombCPUCost	CPU cost for merging per record		Y	
csPartitionCPUCost	CPU cost for partitioning per record		Y	
csSortCPUCost	CPU cost for sorting per record		Y	
csIntermComCPUCost	CPU cost for compressing per bytes		Y	Y
csIntermUncomCPUCost	CPU cost for uncompressing per bytes		Y	Y
csSerializeCPUCost	CPU cost for serializing /deserializing per bytes		Y	Y
csSetupCPUCost	CPU cost of setting up a task		Y	
csCleanupCPUCost	CPU cost of cleaning up a task		Y	

4 Modeling the ShuffleRead Phases

We use *tIntermDataSize* and *tIntermDataRecs* represent the number and size of records which last stage ShuffleWrite

In the ShuffleRead phases, each task fetches the relevant ShuffleWrite partition from each executor. If the ShuffleWrite partition is compressed, Spark will uncompress it after the transfer as part of ShuffleRead process. Assuming a uniform distribution of the ShuffleWrite output to all executors, the size and number of records one task to ShuffleRead is

$$tEachSRSize = \frac{tIntermDataSize}{pCurrentTaskNum} \tag{11}$$

$$tEachSR\,\mathrm{Re}\,cs = \frac{tIntermData\,\mathrm{Re}\,cs}{pCurrentTaskNum} \tag{12}$$

then, the size of data read from local and remote are

$$dLocal\,\mathrm{Re}\,adBytes = \frac{tEachSRSize}{pNumExec} \tag{13}$$

$$d\,\mathrm{Re}\,mote\,\mathrm{Re}\,adBytes = \frac{tEachSRSize \times (pNumExec - 1)}{pNumExec} \tag{14}$$

When the task fetched the data, task will uncompress(if compressed) and deserialize it. The total cost of fetching, uncompressing, deserializing data can be represent as

$$\begin{aligned} tShuffleFetchTime = {} & dLocal\,\mathrm{Re}\,adBytes \times csLocalIO\,\mathrm{Re}\,ad\,\mathrm{Cos}\,t \\ & + d\,\mathrm{Re}\,mote\,\mathrm{Re}\,adBytes \times csNetwork\,\mathrm{Cos}\,t \\ & + tEachSRSize \times (tShuffleUncomCPU\,\mathrm{Cos}\,t \\ & + \frac{csSerializeCPU\,\mathrm{Cos}\,t}{tShuffleCompress\,\mathrm{R}\,adio}) \end{aligned} \tag{15}$$

Then the transformation which lead to shuffle decide whether Combine or Sort in ShuffleRead.

Table 5: A subset of cluster-wide and application-level Spark parameters.

Abbreviation	Profile Field (all field represent information at the level of tasks)	Default Value
pExecMem	spark. executor. memory	512m
pShuffleManager	spark. shuffle. manager	sort
pShuffleMemFrac	spark. shuffle. memoryFraction	0. 2
pShuffleSafeFrac	spark. shuffle. safetyFraction	0. 8
pBypassThreshold	spark. shuffle. sort. bypassMergeThreshold	200
pIsSpill	spark. shuffle. spill	true
pIsShuffleCompress	spark. shuffle. compress	true
pIsSpillCompress	spark. shuffle. spill. compress	true
pSerializer	spark. serializer	org. apache. spark. serializer. JavaSerializer
pExecCores	spark. executor. cores	1 in YARN mode, all the available cores on the worker in standalone mode.
pTaskCpus	spark. task. cpus	1
pCompressor	spark. io. compression. codec	snappy
pSplitSize	The size of the input split	
pNumExec	The number of executor	
pCurrentTaskNum	The number of task number this stage	Determined by program and parameter
pNextTaskNum	The number of task number next stage	

When one block is transfered, uncompressed if needed and deserialized, spark will use Aggregator to combine the records or ExternalSorter to sort records. In the Aggregator, records are put into a map one by one, if key already exists, combine them. While in the ExternalSorter, records are put into buffer and no combine. When the map or buffer size get bigger than the setted size and pIsSpill is true, the records in map or buffer will sort, serialize, compress(if compress) and spill to disk. Spark will repeat this process until all records are put into the map or buffer. We can reference this process as Spill.

Note that if Sort , in profile:

$$\left\{ \begin{array}{l} ds\mathrm{Re}\,adComb\,\mathrm{Re}\,csSel = 1 \\ ds\mathrm{Re}\,adCombSizeSel = 1 \\ ds\mathrm{Re}\,adMerge\,\mathrm{Re}\,csSel = 1 \\ ds\mathrm{Re}\,adMergeSizeSel = 1 \\ csSpillCombCPU\,\mathrm{Cos}\,t = 0 \\ csMergeCombCPU\,\mathrm{Cos}\,t = 0 \end{array} \right\}$$

$$tSCW = \left\{ \begin{array}{ll} tShuffleSCW & if\ \ sort \\ tSpillSCW & if\ \ combine \end{array} \right\} \tag{16}$$

$$tRCS = \left\{ \begin{array}{ll} tShuffleRCS & if\ \ sort \\ tSpillRCS & if\ \ combine \end{array} \right\} \tag{17}$$

$$tIntermCompress\,\mathrm{R}\,adio=\begin{Bmatrix}tShuffleCompressRadio & if\ sort\\ tSpillCompressRadio & if\ combine\end{Bmatrix} \tag{18}$$

So we can use same formula to express Combine or Sort.
Assuming each task can get equal memory , then the map or buffer size will be

$$dSpillBufferSize=\frac{pExecMem\times pShuffleMemFrac\times pShuffleSafeFrac\times pTaskCpus}{pExecCores} \tag{19}$$

Then the spill records and file size in each spill are

$$dSpillFileSize=dSpillBufferSize\times dsSerializeRadio\times tIntermCompressRadio \tag{20}$$

$$dPerSpill\,\mathrm{Re}\,cs=\frac{dSpillBufferSize}{ds\,\mathrm{Re}\,adSpillPairWidth} \tag{21}$$

After combine in the spill, the number of records is

$$dSpillComb\,\mathrm{Re}\,cs=tEachSR\,\mathrm{Re}\,cs\times ds\,\mathrm{Re}\,adComb\,\mathrm{Re}\,csSel \tag{22}$$

Then the spill times and records last in memory are

$$dNumSpills=\begin{Bmatrix}0 & pIsSpill==true\\ \left\lfloor\dfrac{dSpillComb\,\mathrm{Re}\,cs}{dPerSpill\,\mathrm{Re}\,cs}\right\rfloor & pIsSpill==\ flase\end{Bmatrix} \tag{23}$$

$$tInMem\,\mathrm{Re}\,cs=dSpillComb\,\mathrm{Re}\,cs-dPerSpill\,\mathrm{Re}\,cs\times dNumSpills \tag{24}$$

The cost of spill process can be represent as

$$\begin{aligned}tSpillPhaseTime=\ &tEachSR\,\mathrm{Re}\,cs\times csSpillCombCPU\,\mathrm{Cos}\,t\\ &+dNumSpills\times dPerSpill\,\mathrm{Re}\,cs\\ &\times\log_2 dPerSpill\,\mathrm{Re}\,cs\times csSortCPU\,\mathrm{Cos}\,t\\ &+tInMem\,\mathrm{Re}\,cs\times\log_2 tInMem\,\mathrm{Re}\,cs\times csSortCPU\,\mathrm{Cos}\,t\\ &+dNumSpills\times dSpillBufferSize\times tSCW\end{aligned} \tag{25}$$

Then, we have records in memory map or buffer and may have spilled records in disk. Spark will sort the file iterator and map or buffer iterator by comparing the first record in each, combine the records if there has spilled files(if combine), who have same key while reading. We can reference this process as Merge.

After combine in Merge, the size and number of records are

$$tComputeIn\,\mathrm{Re}\,cs=dSpillComb\,\mathrm{Re}\,cs\times ds\,\mathrm{Re}\,adMerge\,\mathrm{Re}\,csSel \tag{26}$$

$$\begin{aligned}tComputeInBytes=\ &dSpillComb\,\mathrm{Re}\,cs\times ds\,\mathrm{Re}\,adSpillPairWidth\\ &\times ds\,\mathrm{Re}\,adMergeSizeSel\end{aligned} \tag{27}$$

The cost of Merge process can be represent as

$$\begin{aligned} tMergePhaseTime &= dNumSpills \times dSpillFileSize \times tRCS \\ &+ dSpillComb\,\mathrm{Re}\,cs \times \log_2(dNumSpills + 1) \times csSortCPU\,\mathrm{Cos}\,t \\ &+ I(dNumSpills\,! = 0) \times dSpillComb\,\mathrm{Re}\,cs \times csMergeCombCPU\,\mathrm{Cos}\,t \end{aligned} \tag{28}$$

The total cost of ShuffleRead Phase Time is:

$$\begin{aligned} cShuffle\,\mathrm{Re}\,adPhaseTime &= tShuffleFetchTime + tSpillPhaseTime \\ &+ tMergePhaseTime \end{aligned} \tag{29}$$

5 Modeling the Compute Phases

During this phase, the user-defined transformation(except transformation lead to shuffle)and action will be applied to the data from read of ShuffleRead.
The number and size of records output by the compute phase are

$$dComputeOut\,\mathrm{Re}\,cs = tComputeIn\,\mathrm{Re}\,cs \times dsCompute\,\mathrm{Re}\,csSel \tag{30}$$

$$dComputeOutSize = tComputeInBytes \times dsComputeSizeSel \tag{31}$$

$$tComputeOutPairWidth = \frac{dComputeOutSize}{dComputeOut\,\mathrm{Re}\,cs} \tag{32}$$

The cost of the Compute Phase is:

$$cComputePhaseTime = tComputeIn\,\mathrm{Re}\,cs \times csComputeCPU\,\mathrm{Cos}\,t \tag{33}$$

6 Modeling the ShuffleWrite Phases

The goal of the ShuffleWrite phase is to partition the compute output data for next stage tasks, one partition to one task. It may also include some combine and sort operation depends on transformation.

ShuffleWrite is the most complex phase in execution. There are two shuffle manager , hash and sort to choose, each shuffle manager has different behavior. And the transformation which lead to shuffle define mapsideCombine or not lead to different process path. When use sort manager and no mapsideCombine, pBypassThreshold also cause difference. And pIsSpill determine whether to spill intermediate data to disk.

So, we consider the shuffle manager set to hash or sort separately.
Note that when !mapsideCombine, in profile:

$$\begin{cases} dsWriteComb\,\mathrm{Re}\,csSel = 1 \\ dsWriteCombSizeSel = 1 \\ dsWriteMerge\,\mathrm{Re}\,csSel = 1 \\ dsWriteMergeSizeSel = 1 \\ csSpillCombCPU\ \mathrm{Cos}\,t = 0 \\ csMergeCombCPU\ \mathrm{Cos}\,t = 0 \end{cases}$$

Case 1:pShuffleManager is sort
ExternalSorter is used to process the data, which is also used in the ShuffleRead if sort.
As in ShuffleRead, we divide ShuffleWrite to Spill and Merge.

In the Spill phase, if mapsideCombine is true. ExternalSorter will use map to combine compute out records, if map size get bigger than setted size, it will spill(if pIsSpill). if mapsideCombine is false, ExternalSorter will use buffer. Other thing is same as map, except combine. But when *pNextTaskNum <= pBypassThreshold,* compute out records are just be partitioned and write to correspond files.
We reference *pNextTaskNum <= pBypassThreshold* as

$$tBypassMergeSort = !mapSideCombine \,\&\&\, pNextTaskNum <= pBypassThreshold \quad (34)$$

The setted size is equal as ShuffleRead, then the spill records and file size in each spill are

$$dSpillFileSize = dSpillBufferSize \times dsSerializeRadio \times tShuffleCompressRadio \quad (35)$$

$$dPerSpill\,\mathrm{Re}\,cs = \frac{dSpillBufferSize}{dsWriteSpillPairWidth} \quad (36)$$

After combine in the spill, the number of records is

$$dSpillComb\,\mathrm{Re}\,cs = dComputeOut\,\mathrm{Re}\,cs \times dsWriteComb\,\mathrm{Re}\,csSel \quad (37)$$

The spill times and records last in memory formula are same as in ShuffleRead, the cost of spill process can be represent as

$$\begin{aligned} tSpillPhaseTime = {} & dComputeOut\,\mathrm{Re}\,cs \times (csPartitionCPU\ \mathrm{Cos}\,t \\ & + csSpillComb\,\mathrm{C}\,PU\ \mathrm{Cos}\,t) \\ & + I(!tByPassMergeSort) \times \\ & (dNumSpills \times dPerSpill\,\mathrm{Re}\,cs \times \log_2 dPerSpill\,\mathrm{Re}\,cs \times csSortCPU\ \mathrm{Cos}\,t \\ & + tInMem\,\mathrm{Re}\,cs \times \log_2 tInMem\,\mathrm{Re}\,cs \times csSortCPU\ \mathrm{Cos}\,t \\ & + dNumSpills \times dSpillBufferSize \times tShuffleSCW) \\ & + I(tByPassMergeSort) \times \\ & dComputeOut\,\mathrm{Re}\,cs \times dsWriteSpillPairWidth \times tShuffleSCW \end{aligned} \quad (38)$$

Then, we have records in memory map or buffer and may have spilled records in disk. Spark will sort the file iterator and map or buffer iterator by comparing the first record in each, combine the records (if combine)who have same key while reading, if there has spilled files. Last, write final result to one File on local disk. We can reference this process as Merge. One special case, if tBypassMergeSort is true, it will just read the spilled files and write the content to one file.

Then the cost of merge process can be represent as

$$\begin{aligned} tMergePhaseTime = I(!tByPassMergeSort) \times \{ & dNumSpills \times \\ & dSpillFileSize \times tShuffleRCS \\ & + dSpillComb\mathrm{Re}cs \times \\ & [\log_2(dNumSpills + 1) \times csSortCPU\,\mathrm{Cos}\,t \\ & + I(dNumSpills != 0) \times csMergeCombCPU\,\mathrm{Cos}\,t \\ & + dsWriteSpillPairWidth \times dsWriteMergeSizeSel \times tShuffleSCW]\} \\ & + I(tByPassMergeSort) \times [dComputeOut\mathrm{Re}cs \times \\ & dsWriteSpillPairWidth \times (csLocalIO\mathrm{Re}ad + csLocalIOWrite)] \end{aligned} \tag{39}$$

Case 2:pShuffleManager==hash
Hash manager use Aggregator to process data. If mapsideCombine, it nearly have no difference between Sort manager. Just the time to partition is different. Hash manager partition the final data before write to the disk while Sort manager partition when the compute out data come in. Contrary to sort manager, Hash manager don't merge these files into one single file.
So compute the spill time as

$$\begin{aligned} tSpillPhaseTime = dComputeOut\mathrm{Re}cs \times csSpillComb\mathrm{C}PU\,\mathrm{Cos}\,t \\ & + dNumSpills \times dPerSpill\mathrm{Re}cs \times \\ & \log_2 dPerSpill\mathrm{Re}cs \times csSortCPU\,\mathrm{Cos}\,t \\ & + tInMem\mathrm{Re}cs \times \log_2 tInMem\mathrm{Re}cs \times csSortCPU\,\mathrm{Cos}\,t \\ & + dNumSpills \times dSpillBufferSize \times tSpillSCW \end{aligned} \tag{40}$$

Compute the merge time as

$$\begin{aligned} tMergePhaseTime = dNumSpills \times dSpillFileSize \times tSpillRCS \\ & + dSpillComb\mathrm{Re}cs \times \\ & [\log_2(dNumSpills + 1) \times csSortCPU\,\mathrm{Cos}\,t \\ & + I(dNumSpills != 0) \times csMergeCombCPU\,\mathrm{Cos}\,t \\ & + dsWriteMerge\mathrm{Re}csSel \times csPartitionCPU\,\mathrm{Cos}\,t \\ & + dsWritePairWidth \times dsWriteMergeSizeSel \times tShuffleSCW] \end{aligned} \tag{41}$$

If !mapsideCombine, things become different . Compute out records are just be partitioned and write to correspond files . so we take this case separately. So, the cost of ShuffleWrite phase time is

$$\begin{aligned} & If\,(pShuffleManager == hash \,\&\&!mapSideCombine) \\ & \quad cShuffleWritePhaseTime = dComputeOut\mathrm{Re}\,\mathrm{cs} \times (csPartitionCPU\,\mathrm{Cos}\,t \\ & \qquad\qquad + tComputeOutPairWidth \times tShuffleSCW) \\ & else \\ & \quad cShuffleWritePhaseTime = tSpillPhaseTime + tMergePhaseTime \end{aligned} \tag{42}$$

The number and size of records from all task to write to disk are

$$\begin{aligned} tIntermDataSize = {} & pCurrentTaskNum \times dComputeOutSize \times \\ & dsWriteCombSizeSel \times dsWriteMergeSizeSel \times \\ & dsSerializeRadio \times dsShuffleCompressRadio \end{aligned} \tag{43}$$

$$\begin{aligned} tIntermData\mathrm{Re}cs = {} & pCurrentTaskNum \times dComputeOut\mathrm{Re}cs \times \\ & dsWriteComb\mathrm{Re}csSel \times dsWriteMerge\mathrm{Re}csSel \end{aligned} \tag{44}$$

7 Modeling the Func Phase

During this phase, the result of this job will be sent to Driver Program. It only occurs in ResultStage.
Then the cost of Func phase time is

$$cFuncPhaseTime = dComputeOutSize \times csNetwork\mathrm{Cos}t \tag{45}$$

8 Modeling the overall Spark Application Execution

In FIFO scheduling mode, A Spark Application consists of several job run in line. And each job consists of several stages executing in parallel or in line according to DAG. The tasks in one stage executing in parallel and in waves.

One simple way for estimating the cost for each task is to sum up the cost fields estimated in Section 3-7. The overall cost for a single task is:

$$\begin{aligned} taskTime = {} & c\mathrm{Re}adPhaseTime\ or\ cShuffle\mathrm{Re}adPhaseTime \\ & + cComputePhaseTime \\ & + cShuffleWritePhaseTime\ or\ cFuncPhaseTime \end{aligned} \tag{46}$$

After compute all *taskTime* of stages in one Job.Then we will use Task Scheduler Simulator to schedule and simulate the execution of individual tasks on a virtual cluster. According to the DAG, the Task Scheduler Simulator can determine when tasks of one stage whose parent stage is finished to run and which stages can run in parallel. Then we get one job execution time as *jobTime.*

The overall application cost is simply the sum of *jobTime*.

$$applicationTime = \sum_{i=1}^{n} jobTime_i \tag{47}$$