Spark Performance Models

Abstract

Spark is a fast and general-purpose cluster computing system that performs in-memory computing, with the goal of outperforming disk-based engines like Hadoop. This technical report describes a detailed set of mathematical performance models for describing the execution of a Spark application. These models describe data and computation cost information at the fine granularity of phases within the map and reduce tasks of an application execution. They 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. It provides programmers with an application programming interface centered on a data structure called the resilient distributed dataset (RDD), a read-only multi-set of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. Spark is 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.

A Spark cluster employs a master-slave architecture where one master node (called Driver) manages a number of slave nodes (called Worker). As Shown in Figure 1, Spark applications run as independent sets of processes on a cluster, coordinated by the *SparkContext* object in users' main program (called the driver program). Specifically, to run on a cluster, the *SparkContext* can connect to several types of cluster managers (either Spark's own standalone cluster manager, Mesos or YARN), which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to *SparkContext*) to the executors. Finally, *SparkContext* sends tasks to the executors to run.

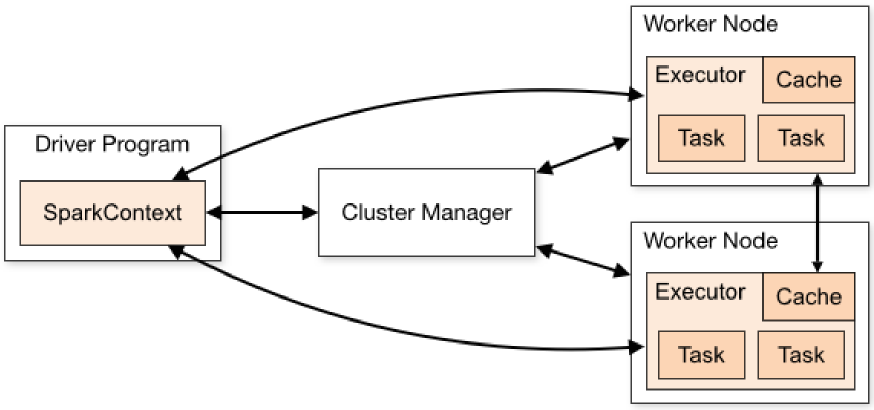

Figure 1 Spark architecture

The main abstraction in Spark is that of a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. There are two type of operations that can be applied to RDD: *transformation*, which applies user-defined function to process the data in a RDD and return a new RDD representing the results; *action*, which returns a value to the driver program after running a computation on the dataset.

RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand.

Figure 2 shows the details for executing a Spark application on the cluster. Spark launches an application by first splitting the input dataset into data splits. Each split is then scheduled to one executor of a *Worker* and is processed by a task. A *Task Scheduler* is responsible for scheduling executions of tasks while taking data locality into account. Each executor has a predefined number of threads for running tasks. If the number of tasks in a stage is more than the number of threads, these tasks will be executed in multiple waves. When compute phases in a *ShuffleMap* stage complete, the output data of each task are partitioned, and the number of partitions equals to the number of tasks in the next stage. Note that Spark will combine or sort the output data for each task if *mapsideCombine* is set to “true” by the transformations leading to shuffle. Finally, the *Result* stage performs actions and send results to the *Driver* program.

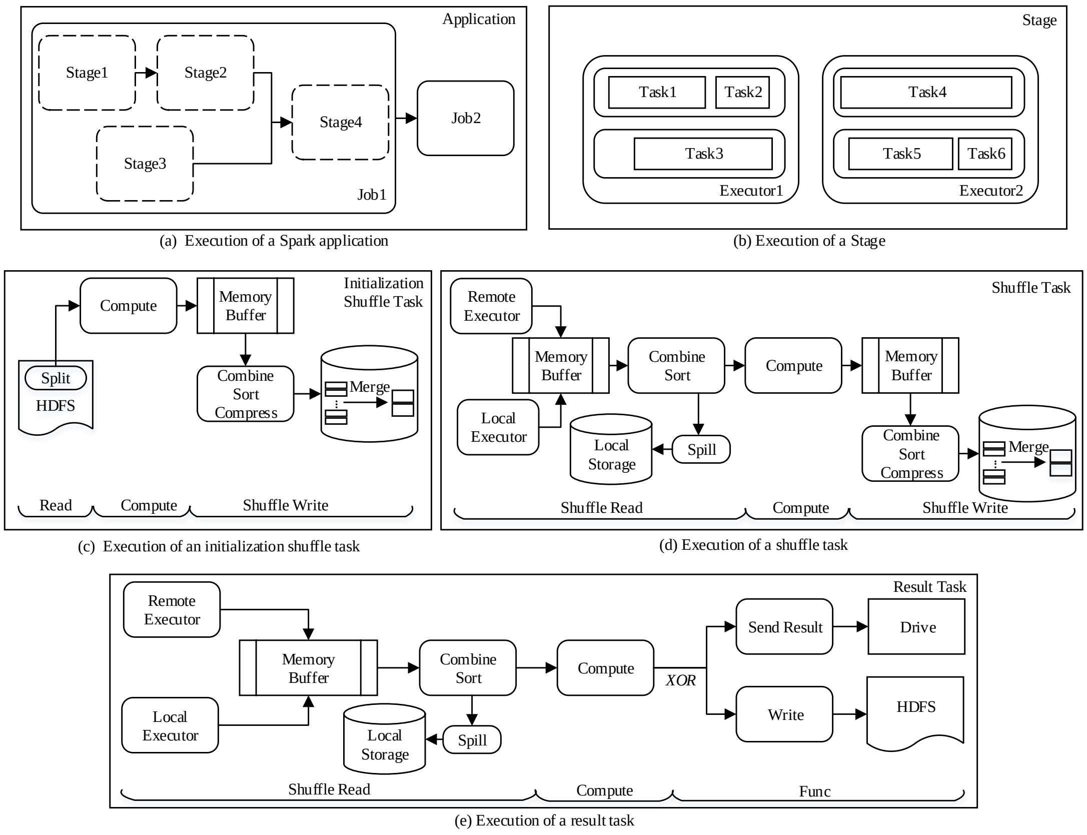

Figure 2: The detailed process for executing a Spark Application

As illustrated in Figure 2(c)-(e), task execution can be divided into five phases:

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

2. *ShuffleRead*: Transferring the intermediate data (generated in the last *ShuffleWrite* stage) from the executors to an executor (decompressing if it is necessary). Combining or sorting (defined by the transformations leading to shuffle operations) operations will occur during this phase, and *Spill* and *Merge* processes may also be involved.

3. *Compute*: Performing transformations and actions to generate the output data.

4. *ShuffleWrite*: Partitioning, combing and sorting operations occurs if transformation has defined *mapsideCombine* (the *Spill* and *Merge* processes may also be involved). Computing the output data and writing the intermediate data to files in the local disk (performing the compression if specified).

5. *Func*: Send the final results to the *Driver* program.

We have modeled all task phases in order to accurately model the execution of a Spark application. 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 data and computation estimates for a stage: data 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.

2. Preliminaries

The performance models calculate the data and computation fields in a profile:

- *Data fields* capture the number of bytes and records (key-value pairs) flowing through the different tasks and phases of a Spark program execution. An example field is the number of records in the buffer per spill. Table 1 lists all the data fields.
- *Computation fields* capture the execution time of tasks and phases of a Spark program execution. An example field is the shuffle read time in a task. Table2 lists all the computation fields.

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

- *Data Statistics fields* capture statistical information about the data, e.g., the average number of the compute phase records output per input record (compute selectivity). Table 3 list all the data statistics fields.
- *Computation Statistics fields* capture statistical information about execution time, e.g., the CPU cost for computing per record. Table 4 lists all the computation 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 1-5. Note the prefixes in all abbreviations used to distinguish where each abbreviation belongs to: *d* for dataset fields, *c* for computation fields, *ds* for data statistics fields, *cs* for computation statistics fields, *p* for Spark parameters, and *t* for temporary information not stored in the profile.

Table 1: Data fields in the profile. *d*, *r* and *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***
dReadInRecs	Read in records	√		√
dReadInBytes	Read in bytes	√		√
dDenFileJarSize	Dependency file and jar size	√		
dComputeOutRecs	Compute out records	√		√
dComputeOutBytes	Compute out Bytes	√		√
dSpillFileSize	Spill file size	√		√
dPerSpillRecs	Number of records in the buffer per spill	√		√
dNumSpills	Number of spills	√		√
dSpillBuffer	Buffer size per spill	√		√
dSpillCombRecs	Number of records total spill	√		√
dMergeCombRecs	Number of records after combine	√		√
dWriteRecs	Number of records written to local file system	√		√
dWriteBytes	Bytes written to local file system	√		√
dRemoteReadBytes	Bytes ShuffleRead from remote	√		√
dLocalReadBytes	Bytes ShuffleRead from local	√		√

dShuffleReadRecs	Number of records ShuffleRead	√		√

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:

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

pCompressor decides the value of *dsIntermCompressRatio*, *csIntermComCPUCost* and *csIntermUncomCPUcost*

pSerializer decides the value of *dsSerializeRatio* and *csSerdeBytesCPUCost*

Note that whether an *Aggregator* compresses spilled data or not is determined by *pIsSpillCompress*, and whether or not the compress option is turned on in *ExternalSorter* is determined by *pIsShuffleCompress*. So we use the *tSpill*- and *tShuffle*- to represent these situations.

```
If ( $pIsShuffleCompress == true$ )
    $tShuffleCompressRatio = dsIntermCompressRatio;$
    $tShuffleComCPU\,\mathrm{Cos}t = csIntermComCPU\,\mathrm{Cos}t;$
    $tShuffleUncomCPU\,\mathrm{Cos}t = csIntermUncomCPU\,\mathrm{Cos}t;$

If ( $pIsShuffleCompress == false$ )
    $tShuffleCompressRatio = 1;$
    $tShuffleComCPU\,\mathrm{Cos}t = 0;$
    $tShuffleUncomCPU\,\mathrm{Cos}t = 0;$
```

Table 2: Cost fields in the profile. *d*, *r* and *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	√	√	√
cCleanupPhaseTime	Cleanup phase time in a task	√	√	√
cReadPhaseTime	Read phase time in a task	√	√	√
cShuffleReadPhaseTime	ShuffleRead phase time in a task	√	√	√
cComputePhaseTime	Compute phase time in a task	√	√	√
cShuffleWritePhaseTime	ShuffleWrite phase time in a task	√	√	√
cFuncPhaseTime	Func phase time in a task	√	√	√

```
If ( $pIsSpillCompress == true$ )
    $tSpillCompressRatio = dsIntermCompressRatio;$
    $tSpillComCPU\,\mathrm{Cos}t = csIntermComCPU\,\mathrm{Cos}t;$
    $tSpillUncomCPU\,\mathrm{Cos}t = csIntermUncomCPU\,\mathrm{Cos}t;$

If ( $pIsSpillCompress == false$ )
    $tSpillCompressRatio = 1;$
    $tSpillComCPU\,\mathrm{Cos}t = 0;$
    $tSpillUncomCPU\,\mathrm{Cos}t = 0;$
```

We use postfix *SCW* to represent the time to serialize, compress and write one byte of data to the local disk, postfix *RCS* to represent the time to read, uncompress and deserialize one byte of data into memory.

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

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

$$\begin{aligned} tSpillSCW = {} & csSerdeBytesCPUCost \\ & + dsSerializeRatio \times (tSpillComCPU\,\mathrm{Cos}\,t \\ & + tSpillCompressRatio \times csLocalIOWriteCost) \end{aligned} \tag{4}$$

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

Table 3: Data statistics fields in the profile. *d*, *r* and *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	√		
dsComputeSizeSel	Compute selectivity in terms of size	√		
dsComputeRecsSel	Compute selectivity in terms of records	√		
dsWriteCombSizeSel	Combine selectivity in terms of size in the ShuffleRead phase	√		√
dsWriteCombRecsSel	Combine selectivity in terms of records in the ShuffleRead phase	√		√
dsWriteMergeSizeSel	Merge selectivity in terms of size in the ShuffleRead phase	√		
dsWriteMergeRecsSel	Merge selectivity in terms of records in the ShuffleRead phase	√		
dsReadCombSizeSel	Combine selectivity in terms of size in the ShuffleWrite phase	√		√
dsReadCombRecsSel	Combine selectivity in terms of records in the ShuffleWrite phase	√		√
dsReadMergeSizeSel	Merge selectivity in terms of size in the ShuffleWrite phase	√		
dsReadMergeRecsSel	Merge selectivity in terms of records in the ShuffleWrite phase	√		
dsSerializeRatio	Serialize selectivity in terms of bytes	√		√
dsReadSpillPairWidth	Memory per ShuffleRead record	√		
dsWriteSpillPairWidth	Memory per ShuffleWrite record	√		
dsIntermCompressRatio	Compress ratio	√		√

Table 4: computation statistics fields in the profile. *d*, *r* and *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		√	

csHdfsWriteCost	I/O cost for writing to HDFS per byte		√	
csLocalIOReadCost	I/O cost for reading local disk per byte		√	
csLocalIOWriteCost	I/O cost for writing to local disk per byte		√	
csNetworkCost	Cost for network transfers per byte		√	
csComputeCPUCost	CPU cost for computing per record		√	
csSpillCPUCost	CPU cost for spilling per record		√	
csSpillCombCPUCost	CPU cost for combining per record		√	
csMergeCombCPUCost	CPU cost for merging per record		√	
csPartitionCPUCost	CPU cost for partitioning per record		√	
csSortCPUCost	CPU cost for sorting per record		√	
csIntermComCPUCost	CPU cost for compressing per bytes		√	√
csIntermUncomCPUCost	CPU cost for uncompressing per bytes		√	√
csSerializeCPUCost	CPU cost for serializing /deserializing per bytes		√	√
csSetupCPUCost	CPU cost of setting up a task		√	
csCleanupCPUCost	CPU cost of cleaning up a task		√	

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	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	

3. Modeling the *Read* Phase

During this phase, the input splits are read first, and then the input records are created and passed to the user-defined transformations.

$$dReadInBytes = pSplitSize \tag{6}$$

$$dReadInRecs = \frac{dReadInBytes}{dsReadInPairWidth} \tag{7}$$

$$tComputeInBytes = dReadInBytes \tag{8}$$

$$tComputeInRecs = dReadInRecs \tag{9}$$

The cost of the *read* phase is:

$$cReadPhaseTime = csHdfsReadCost \times dReadInBytes \tag{10}$$

4. Modeling the *ShuffleRead* Phase

We use *tIntermDataSize* and *tIntermDataRecs* to represent the number and size of records generated in the last *ShuffleWrite* phase.

In the *ShuffleRead* phase, each task fetches the relevant *ShuffleWrite* partition from an executor. If the *ShuffleWrite* partition has been compressed, Spark will uncompress it first. Assuming a uniform distribution of each *ShuffleWrite* output to all executors, the size and number of records one task to *ShuffleRead* is:

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

$$tEachSRRecs = \frac{tIntermDataRecs}{pCurrentTaskNum} \tag{12}$$

Then, the size of data reading from local and remote are:

$$dLocalReadBytes = \frac{tEachSRSize}{pNumExec} \tag{13}$$

$$dRemoteReadBytes = \frac{tEachSRSize \times (pNumExec - 1)}{pNumExec} \tag{14}$$

After fetching the data, a task will first uncompress (if the data have been compressed) and deserialize it. The total cost of fetching, uncompressing and deserializing can be represented as:

$$\begin{aligned} tShuffleFetchTime = {} & dLocalReadBytes \times csLocalIOReadCost \\ & + dRemoteReadBytes \times csNetworkCost \\ & + tEachSRSize \times (tShuffleUncomCPUCost \\ & + \frac{csSerializeCPUCost}{tShuffleCompressRadio} \end{aligned} \tag{15}$$

Note that the transformation leading to shuffle operations decides whether to combine or sort in the *ShuffleRead* phase.

When blocks are transferred, uncompressed (if needed) and deserialized, Spark may use *Aggregator* to combine the records or *ExternalSorter* to sort records. In the case of *Aggregator*, records are put into a hashmap sequentially. If the key of a record already exists in the hashmap, then Spark will combine them. In the case of *ExternalSorter*, records are put into a buffer directly.

If the size of hashmap or buffer size gets larger than the predefined threshold and the value of *pIsSpill* is set to “true”, then the records in hashmap or buffer will be sorted, serialized, compressed (if compressed), and finally spilled to disk. Spark repeats this process until all records are put into the hashmap or buffer. This process is often referenced as a *Spill* phase.

If the “sort” option is set, we have:

$$\left\{\begin{array}{l} dsReadCombRecsSel = 1 \\ dsReadCombSizeSel = 1 \\ dsReadMergeRecsSel = 1 \\ dsReadMergeSizeSel = 1 \\ csSpillCombCPUCost = 0 \\ csMergeCombCPUCost = 0 \end{array}\right\}$$

In order to represent the following expressions uniformly, let:

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

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

$$tIntermCompressRatio = \left\{\begin{array}{ll} tShuffleCompressRatio & if\ sort \\ tSpillCompressRatio & if\ combine \end{array}\right\} \tag{18}$$

Assuming each task can get equal size of memory, then the size of hashmap or buffer will be:

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

The sizes of spill records and files in each spill phase are:

$$dSpillFileSize = dSpillBufferSize \times dsSerializeRatio \times tIntermCompressRatio \tag{20}$$

$$dPerSpillRecs = \frac{dSpillBufferSize}{dsReadSpillPairWidth} \tag{21}$$

After combining in a spill phase, the number of records equals to:

$$dSpillCombRecs = tEachSRRecs \times dsReadCombRecsSel \tag{22}$$

Then the times of spill operations and the number of records located in memory are:

$$dNumSpills = \left\{\begin{array}{ll} 0 & pIsSpill == true \\ \left\lfloor \dfrac{dSpillCombRecs}{dPerSpillRecs} \right\rfloor & pIsSpill == \ flase \end{array}\right\} \tag{23}$$

$$tInMemRecs = dSpillCombRecs \text{ - } dPerSpillRecs \times dNumSpills \tag{24}$$

The cost of spill phase can be represented as:

$$\begin{aligned} tSpillPhaseTime = {} & tEachSRRecs \times csSpillCombCPUCost \\ & + dNumSpills \times dPerSpillRecs \\ & \times log_2 dPerSpillRecs \times csSortCPUCost \\ & + tInMemRecs \times log_2 tInMemRecs \times csSortCPUCost \\ & + dNumSpills \times dSpillBufferSize \times tSCW \end{aligned} \tag{25}$$

Until now, we have records in a hashmap or a buffer and may have spilled records in disk. For records in the hashmap or buffer, Spark generates a memory iterator to traverse them; for the records in the disk, Spark generates a file iterator to traverse them. Since all of these records have been sorted before, all we need to do every time is take the first record through iterator and judge whether it is marked as “combine”. If so, Spark combines these records with the same key value.

This process is often referenced as a *Merge* phase.

After combining in a *Merge* phase, the size and number of records equal to:

$$tComputeInRecs = dSpillCombRecs \times dsReadMergeRecsSel \tag{26}$$

$$tComputeInBytes = dSpillCombRecs \times dsReadSpillPairWidth \times dsReadMergeSizeSel \tag{27}$$

The cost of *Merge* phase can be represented as:

$$\begin{aligned} tMergePhaseTime = {} & dNumSpills \times dSpillFileSize \times tRCS \\ & + dSpillCombRecs \times log_2(dNumSpills + 1) \times csSortCPUCost \\ & + I(dNumSpills! = 0) \times dSpillCombRecs \times csMergeCombCPUCost \end{aligned} \tag{28}$$

In summary, the total cost of *ShuffleRead* phase is:

$$cShuffleReadPhaseTime = tShuffleFetchTime + tSpillPhaseTime + tMergePhaseTime \tag{29}$$

5. Modeling the *Compute* Phase

During this phase, the user-defined transformations (except transformations leading to shuffle operations) and actions will be applied to the data generated in the *ShuffleRead* phase.

The number and size of records outputted by the *compute* phase are:

$$dComputeOutRecs = tComputeInRecs \times dsComputeRecsSel \tag{30}$$

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

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

So the cost of the *compute* phase can be calculated as:

$$cComputePhaseTime = tComputeInRecs \times csComputeCPUCost \tag{33}$$

6. Modeling the *ShuffleWrite* Phase

The goal of the *ShuffleWrite* phase is to partition the data outputted in *compute* phase for tasks in the next phase (one data partition for one task). It may also include some *combine* and *sort* operations that depend on transformation.

The processes in *ShuffleWrite* phase are very complicated: 1) there are two kinds of shuffle managers, namely *hash manager* and *sort manager*, that behave differently in this phase; 2) transformations leading to shuffle operations define whether *mapsideCombine* or not, which may cause different processing logic; 3) When we choose the options of “sort manager” and “*mapsideCombine*”, the different configurations of *pBypassThreshold* may also lead to difference processing logic; and 4) the parameter *pIsSpill* determines whether the intermediate data are written to the disk or not when the size of these data is larger than the size of the available memory in the *ShuffleWrite* phase.

Spark doesn’t perform combine operations if the *mapsideCombine* is set to “false”, so we have:

$$\left\{\begin{array}{l} dsWriteCombRecsSel = 1 \\ dsWriteCombSizeSel = 1 \\ dsWriteMergeRecsSel = 1 \\ dsWriteMergeSizeSel = 1 \\ csSpillCombCPUCost = 0 \\ csMergeCombCPUCost = 0 \end{array}\right\}$$

Case 1: the value of *pShuffleManager* is set to “sort” (sort manager)

In this case, *ExternalSorter* is used to process the data just like that of in the *ShuffleRead* phase. Specifically, in the *Spill* phase, if *mapsideCombine* is set to “true”, hashmap is chosen by the *ExternalSorter* to process records. If the size of hashmap gets bigger than the set value, it will spill (if pIsSpill has been set to “true”). If *mapsideCombine* is set to “false”, buffer is used by *ExternalSorter* to process records. The only difference between them is that the “hashmap” process will perform combine operations, and “buffer” process will not.

Note that when condition *pNextTaskNum <= pBypassThreshold* (the number of tasks in the next stage is less than the value of *pBypassThreshold*) holds, the output records in compute phase are directly partitioned and written to the corresponding files (without processing). We use following expression to indicate this situation:

$$tBypassMergeSort = !mapSideCombine \,\&\&pNextTaskNum <= pBypassThreshold \tag{34}$$

The size of spill file and the size of spill records in each spill, which are equal to that of in *ShuffleRead* phase, are:

$$dSpillFileSize = dSpillBufferSize \times dsSerializeRatio \times tShuffleCompressRatio \tag{35}$$

$$dPerSpillRecs = \frac{dSpillBufferSize}{dsWriteSpillPairWidth} \tag{36}$$

After combining in the spill phase, the number of records is

$$dSpillCombRecs = dComputeOutRecs \times dsWriteCombRecsSel \tag{37}$$

The spill times and the number of records located in memory are the same as that of in *ShuffleRead* phase, so the cost of spill phase can be represented as

$$\begin{aligned} tSpillPhaseTime = dComputeOutRecs \times (&csPartitionCPUCost \\ &+ csSpillCombCPUCost) \\ &+ I(!tByPassMergeSort) \times \\ &(dNumSpills\,? \qquad \times \quad {}_2 \qquad s \times csSortCPUCost \\ &+ tInMemRecs \times log_2 tInMemRecs \times csSortCPUCost \\ &+ dNumSpills \times dSpillBufferSize \times tShuffleSCW) \\ &+ I(tByPassMergeSort) \times \\ &dComputeOutRecs \times dsWriteSpillPairWidth \times tShuffleSCW \end{aligned} \tag{38}$$

Until now, we have records in a hashmap or a buffer and may have spilled records in disk. For records in the hashmap or buffer, Spark generates a memory iterator to traverse them; for the records in the disk, Spark generates a file iterator to traverse them. Since all of these records have been sorted before, all we need to do every time is take the first record through iterator and judge whether it is marked as “combine”. If so, Spark combines these records with the same key value. Finally, all intermediate results (including spilled data and in-memory data) are written to a local file. This process is often referenced as a *Merge* phase. Note that in this case, if the *tBypassMergeSort* is set to “true”, Spark simply reads all of the spilled files and merge them into a single file.

The cost of *Merge* process can be represented as:

$$\begin{aligned} tMergePhaseTime = {} & I(!tByPassMergeSort) \times \{dNumSpills\, ? \\ & dSpillFileSize \times tShuffleRCS \\ & + dSpillCombRecs \times \\ & [log_2(dNumSpills + 1) \times csSortCPUCost \\ & + I(dNumSpills! = 0) \times csMergeCombCPUCost \\ & + dsWriteSpillPairWidth \times dsWriteMergeSizeSel \times tShuffleSCW]\} \\ & + I(tByPassMergeSort) \times [dComputeOutRecs \times \\ & dsWriteSpillPairWidth \times (csLocalIORead + csLocalIOWrite)] \end{aligned} \tag{39}$$

***Case 2*:** the value of *pShuffleManager* is set to “hash” (hash manager)

Hash manager uses *Aggregator* to process data. If *mapsideCombine* is set to “true”, hash manager behaves similarly just like sort manager, except that the timing of partitioning is different. Specifically, hash manager partitions the final data before writing them to the disk, while sort manager does partitioning operations when the data generated in the compute phase are fed in. Another difference between hash manager and sort manger is that hash manager doesn’t merge data files into one single file.

The spill time is represented as:

$$\begin{aligned} tSpillPhaseTime = {} & dComputeOutRecs \times csSpillCombCPUCost \\ & + dNumSpills \times dPerSpillRecs \times \\ & log_2 dPerSpillRecs \times csSortCPUCost \\ & + tInMemRecs \times log_2 tInMemRecs \times csSortCPUCost \\ & + dNumSpills \times dSpillBufferSize \times tSpillSCW \end{aligned} \tag{40}$$

The merge time can be represented as:

$$\begin{aligned} tMergePhaseTime = {} & dNumSpills \times dSpillFileSize \times tSpillRCS \\ & + dSpillCombRecs \times \\ & [log_2(dNumSpills + 1) \times csSortCPUCost \\ & + I(dNumSpills! = 0) \times csMergeCombCPUCost \\ & + dsWriteMergeRecsSel \times csPartitionCPUCost \\ & + dsWritePairWidth \times dsWriteMergeSizeSel \times tShuffleSCW] \end{aligned} \tag{41}$$

If *mapsideCombine* is set to “false”, things are different: records generated in the compute phase are partitioned and then written to corresponding files.

In summary, the cost of *ShuffleWrite* phase can be represented as:

$$\begin{aligned}
&If(pShuffleManager == hash \,\&\&!mapSideCombine)\\
&\quad cShuffleWritePhaseTime = dComputeOutRecs \times (csPartitionCPUCost\\
&\qquad\qquad + tComputeOutPairWidth \times tShuffleSCW)\\
&else\\
&\quad cShuffleWritePhaseTime = tSpillPhaseTime + tMergePhaseTime
\end{aligned} \tag{42}$$

The size and number of records written to disk are:

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

$$\begin{aligned}
tIntermDataRecs = {} & pCurrentTaskNum \times dComputeOutRecs\, ?\\
& dsWriteCombRecsSel \times dsWriteMergeRecsSel
\end{aligned} \tag{44}$$

7. Modeling the *Func* Phase

During this phase, the results of the Spark application will be sent to *Driver* program. This phase is only included in the *ResultStage*.

The cost of *Func* phase is:

$$cFuncPhaseTime = dComputeOutSize \times csNetworkCost \tag{45}$$

8. Modeling the overall Spark Application Execution

In FIFO scheduling mode, A Spark application consists of several jobs 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:

$$\begin{aligned}
taskTime = {} & cReadPhaseTime \text{ or } cShuffleReadPhaseTime\\
& + cComputePhaseTime\\
& + cShuffleWritePhaseTime \text{ or } cFuncPhaseTime
\end{aligned} \tag{46}$$

After computing 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*.

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