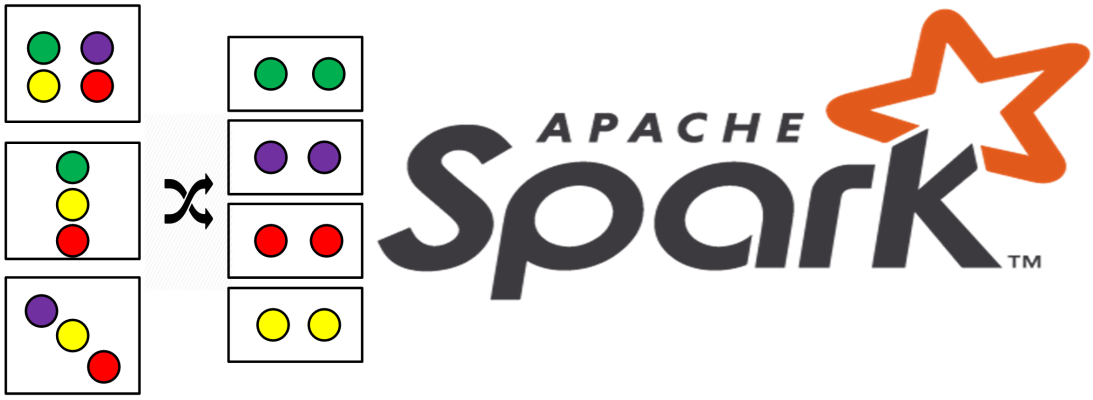


GUIDE TO APACHE SPARK SHUFFLING

Revealing Apache Spark Shuffling Magic

Five Important Aspects of Apache Spark Shuffling to know for building predictable, reliable and efficient Spark Applications

Ajay Gupta Follow
May 22, 2020 · 8 min read ★

1) Data Re-distribution: Data Re-distribution is the primary goal of shuffling operation in Spark. Therefore, Shuffling in a Spark program is executed whenever there is a need to re-distribute an existing distributed data collection represented either by an RDD, Dataframe, or Dataset. The need could be there in order to:

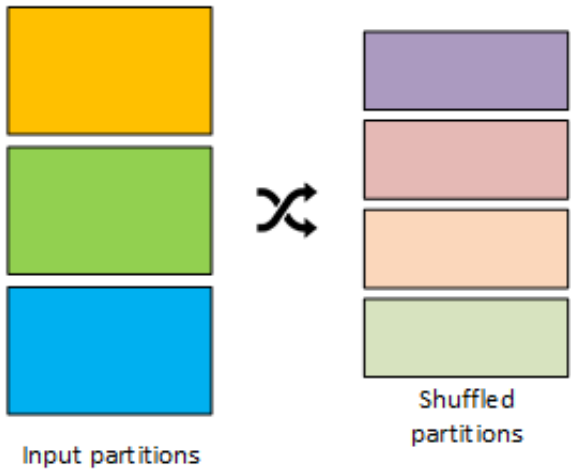
(a) Increase or Decrease the number of data partitions: Since a data partition represents the quantum of data to be processed together by a single Spark Task, there could be situations:

```
 (a) Where existing number of data partitions are not
sufficient enough in order to maximize the usage of
available resources

 (b) Where existing number of data partitions are too
heavy to be computed reliably without memory overruns.

 (c) Where existing number of data partitions are too
high in number such that task scheduling overhead becomes
the bottleneck in the overall processing time.
```

In all of the above situations, redistribution of data is required to either increase or decrease the number of underlying data partitions. The same is achieved by executing shuffling on the existing distributed data collection via commonly available 'repartition' API among RDDs, Datasets, and Dataframes.

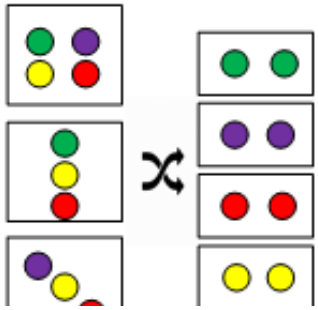
> *You can also read my recently published book, "Guide to Spark Partitioning" which deep dives into all aspects of of Spark*

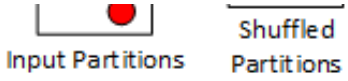
> *Partitioning with multiple examples to explain each of the partitioning aspect in detail: https://www.amazon.com/dp/B08KJCT3XN/*



Shuffling done to increase the input partitions

(b) Perform Aggregation/Join on a data collection(s): In order to perform aggregation/join operation on data collection(s), all data records belonging to aggregation, or a join key should reside in a single data partition. Therefore, if the existing partitioning scheme of the input data collection(s) does not satisfy the condition, then re-distribution in accordance with aggregation/join key becomes mandatory, and therefore shuffling would be executed on the input data collection to achieve the desired re-distribution.





Shuffling done to re-distribute the input partitions on the basis Join/Aggregation Key

2) Partitioner and Number of Shuffle Partitions: Partitioner and number of shuffle partitions are other two important aspects of Shuffling. The number of shuffle partitions specifies the number of output partitions after the shuffle is executed on a data collection, whereas Partitioner decides the target shuffle/output partition number (out of the total number of specified shuffle partitions) for each of the data records. Spark APIs (pertaining to RDD, Dataset or Dataframe) which triggers shuffling provides either of implicit or explicit provisioning of Partitioner and/or number of shuffle partitions.

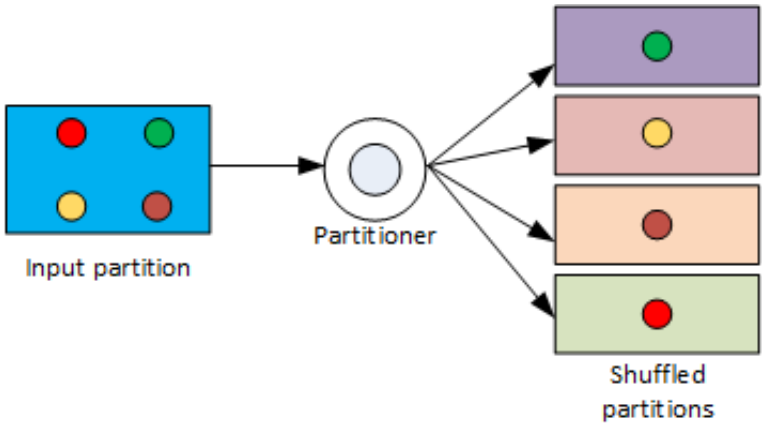


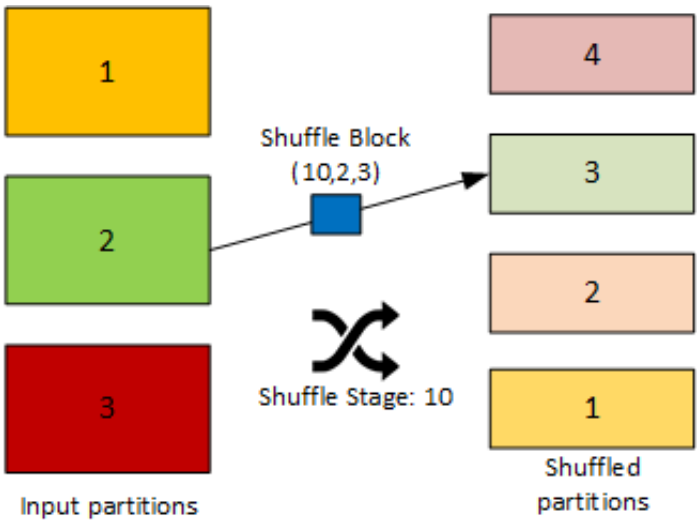
Illustration of a partitioner acting on an input partition to assign each record to one of the shuffled partition

> *Spark provides two widely used implementations of Partitioner, viz., Hash and Range partitioner. Hash Partitioner decides the output partition based on hash code computed for key object specified for the data record, while Range Partitioner decides the*

> *output partition based on the comparison of key value against the range of key values estimated for each of the shuffled partition. Most of the Spark RDD/Dataframe/Dataset APIs requiring shuffling implicitly provision the Hash partitioner for the shuffling operation. There are very few Dataset/Dataframe APIs which provisions for the Range partitioner for the shuffling operation. Also, one can define their own custom partitioner and use the same for shuffling in limited RDD APIs. However, there is no such provision of custom partitioner in any of the Dataframe/Dataset APIs.*
>
> *Provision of number of shuffle partitions varies between RDD and Dataset/Dataframe APIs. In case of RDD, number of shuffle partitions are either implicitly assumed to be same as before shuffling, or number of partitions has to be explicitly provided in the APIs as an argument. In case of Dataset/Dataframe, a key configurable property 'spark.sql.shuffle.partitions' decides the number of shuffle partitions for most of the APIs requiring shuffling. The default value for this property is set to 200. However, in few other Dataframe/Dataset APIs requiring shuffling, user can explicitly mention the number of shuffle partitions as an argument.*

3) Shuffle Block: A shuffle block uniquely identifies a block of data which belongs to a single shuffled partition and is produced from executing shuffle write operation (by ShuffleMap task) on a single input partition during a shuffle write stage in a Spark application.

> *The unique identifier (corresponding to a shuffle block) is represented as a tuple of ShuffleId, MapId and ReduceId. Here, ShuffleId uniquely identifies each shuffle write/read stage in a Spark application, MapId uniquely identifies each of the input partition (of the data collection to be shuffled) and ReduceId uniquely identifies each of the shuffled partition.*



A shuffle block is hosted in a disk file on cluster nodes, and is either serviced by the Block manager of an executor, or via external shuffle service. All shuffle blocks of a shuffle stage are tracked by MapOutPutTracker hosted in the driver.

> *If the status of a Shuffle block is absent against a shuffle stage tracked by MapOutPutTracker, then it leads to 'MetadataFetchFailedException' in the reducer task corresponding to ReduceId in Shuffle block. Also, failure in fetching the shuffle block from the designated Block manager leads to 'FetchFailedException' in the corresponding reducer task.*

4) Shuffle Read/Write: A shuffle operation introduces a pair of stage in a Spark application. Shuffle write happens in one of the stage while Shuffle read happens in subsequent stage. Further, Shuffle write operation is executed independently for each of the input partition which needs to be shuffled, and similarly, Shuffle read operation is executed independently for each of the shuffled partition.

Shuffle write operation (from Spark 1.6 and onward) is executed mostly using either 'SortShuffleWriter' or 'UnsafeShuffleWriter'. The former is used for RDDs where data records are stored as JAVA objects, while the later one is used in Dataframes/Datasets where data records are stored in tungusten format. Both shuffle writers produces a index file and a data file corresponding to each of the input partition to be shuffled. Index file contains locations inside data file for each of the shuffled partition while data file contains actual shuffled data records ordered by shuffled partitions.

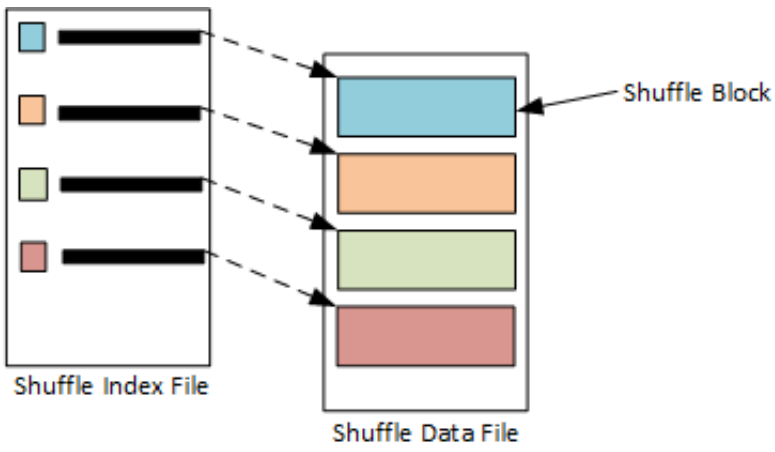
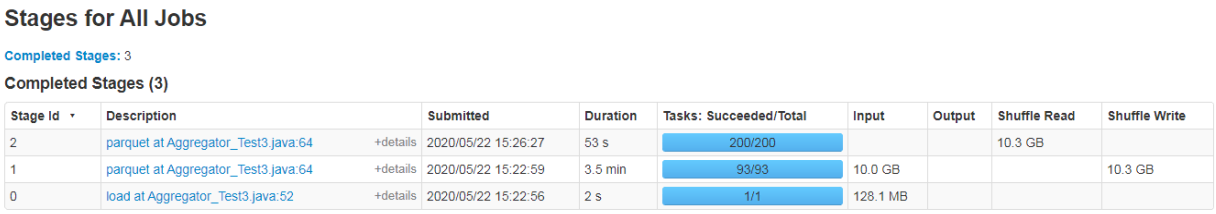


Illustration of Shuffle Index and Data File, Also shown is a Shuffle block contained in the data file.

Shuffle read operation is executed using 'BlockStoreShuffleReader' which first queries for all the relevant shuffle blocks and their locations. This is then followed by pulling/fetching of those blocks from respective locations using block manager module. Finally, a sorted iterator on shuffled data records derived from fetched shuffled blocks is returned for further use.

Metrics is available for both, number of data records and the total bytes written to disk (in shuffle data file) during a shuffle write operation (happening on an input partition). Similarly, metrics is available for number of shuffled data records which are fetched along with total shuffled bytes being fetched during the shuffle read operation (happening on each of the shuffled partition. Individual shuffle metrics of all partitions are then combined to get the shuffle read/write metrics of a shuffle read/write stage.



Stages for All Jobs

Completed Stages: 3

Completed Stages (3)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	parquet at Aggregator_Test3.java:64	+details	2020/05/22 15:26:27	53 s	200/200			10.3 GB	
1	parquet at Aggregator_Test3.java:64	+details	2020/05/22 15:22:59	3.5 min	93/93	10.0 GB			10.3 GB
0	load at Aggregator_Test3.java:52	+details	2020/05/22 15:22:56	2 s	1/1	128.1 MB			

Illustration from Spark Job View showing a pair of Shuffle read and write stages along with corresponding metrics

> *With all these shuffle read/write metrics at hand, one can be aware of data skew happening across partitions during an intermediate stages of a Spark application. Also, since shuffle operation generally involves remote fetches of shuffle blocks over*

> *network, the same could incur considerable additional latency in the data processing pipeline for large amounts of shuffled data. Therefore, a user, with these metrics at hand, can potentially redesign the data processing pipeline in the Spark application in order to target for reduced amounts of shuffled data or completely avoid the shuffle.*

5) Shuffle Spill: During shuffle write operation, before writing to a final index and data file, a buffer is used to store the data records (while iterating over the input partition) in order to sort the records on the basis of targeted shuffled partitions. However, if the memory limits of the aforesaid buffer is breached, the contents are first sorted and then spilled to disk in a temporary shuffle file. This process is called as shuffle spilling. If the breach happens multiple times, multiple spill files could be created during the iteration process. After the iteration process is over, these spilled files are again read and merged to produce the final shuffle index and data file.

A similar buffer shall be used during shuffle read operation, when the data records in shuffle blocks being fetched are required to be sorted on the basis of key values in data records. However, here also, the shuffle read buffer could breach the designated memory limits leading to sorting and disk spilling of the buffer contents. After all the shuffle blocks are fetched, all spilled files are again read and merged to generate the final iterator of data records for further use.

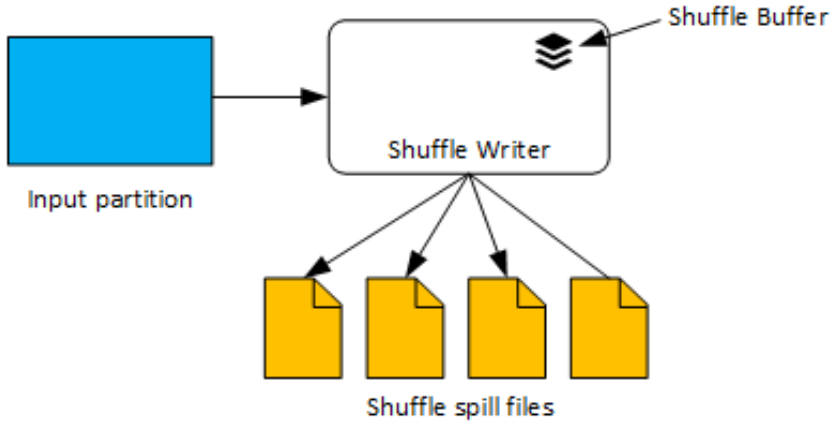


Illustration of generation of shuffle spill files during by Shuffle writer

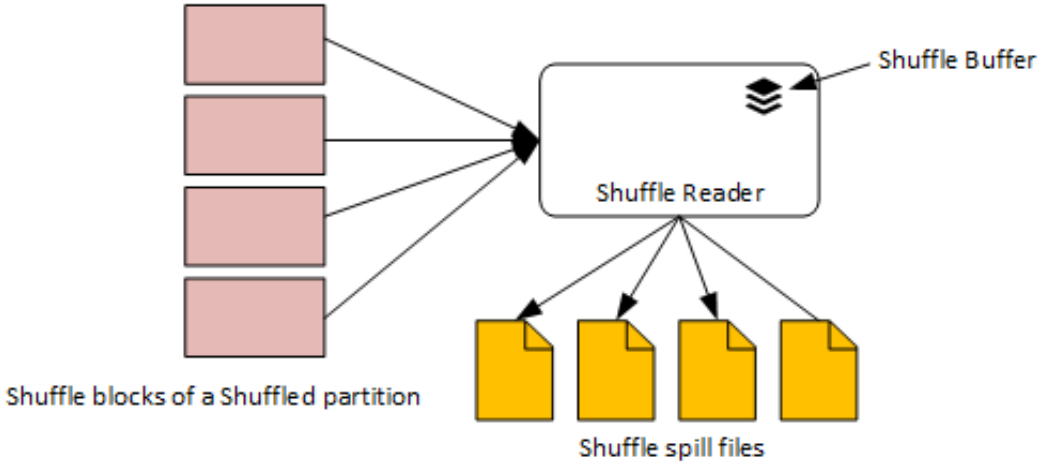


Illustration of generation of shuffle spill files during by Shuffle reader

> *Disk spilling of shuffle data although provides safeguard against memory overruns, but at the same time, introduces considerable latency in the overall data processing pipeline of a Spark Job. This latency is due to the fact that spills introduces additions disk read/write cycles along with ser/deser cycles (in case where data records are JAVA objects) and optional comp/decomp cycles. Amount of shuffle spill (in bytes) is available as a metric against each shuffle read or write stage. This spilling information could help a lot in tuning a Spark Job.*

Summary: Shuffle, being the most prevalent operation in Spark data processing pipelines, it is very important to understand the above critical aspects related to it. The understanding would definitely help one in building reliable, robust, and efficient Spark applications. Last and not the least, the understanding would surely help in quick troubleshooting of commonly reported shuffling problems/errors during Spark Job execution.

In case of further queries about shuffle, or for any feedback, do write in the comments section.

Also, Get a copy of my recently published book on Spark Partitioning: https://www.amazon.com/dp/B08KJCT3XN/

Data Science | Programming | Big Data | Technology | Software Development