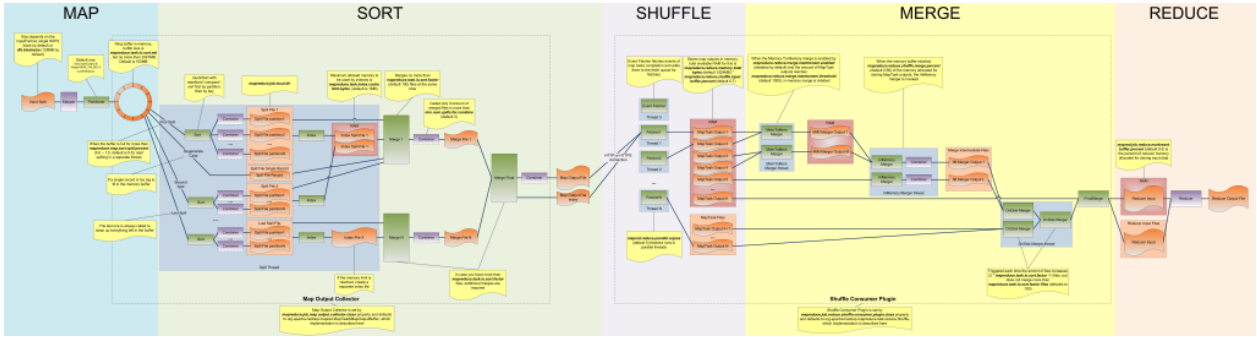
Distributed Systems Architecture

brought to you by Alexey Grishchenko

Hadoop MapReduce Comprehensive Description

Map Reduce is a really popular paradigm in distributed computing at the moment. The first paper describing this principle is the one by Google published in 2004. Nowadays Map Reduce is a term that everyone knows and everyone speaks about, because it was put as one of the foundations to the Hadoop project. For most of the people Map Reduce is an equivalent to "Hadoop" and "Big Data", which is completely wrong. But there are some people that understand the simplest case with WordCount and maybe even building an inverted index using Map Reduce.

But being simple as a concept, it has a kind of complicated implementation in Hadoop. I've tried to find a comprehensive description of it with a good diagram over the internet, but failed. All the diagrams keep repeating "Map – Sort – Combine – Shuffle – Reduce". Of course, it is good to know that the framework works this way, but what about dozens of parameters that are tunable for the framework? What happens if you reduce the buffer size of the Map output or increase it? These diagrams don't offer any help for this. This was the reason for me to build my own diagram and my own description based on the latest source code available in the Hadoop repository.



Hadoop MapReduce Comprehensive Diagram

The overall MapReduce starts with ***InputFormat***. It is the class you specify in the driver application that understands the input and provides you two main interface functions: ***getSplits*** that returns set of input data splits and ***getRecordReader*** that provides you an iterable interface for reading all the records from a single input split. The size of input split depends on the ***InputFormat*** – for text files it is a single block of the HDFS (***dfs.blocksize***), for gzip-compressed files it is the whole file (as gzip archive is not splittable), etc. Each Mapper processes a single input split, which means in most cases it processes ***dfs.blocksize*** data, which equals to 128MB by default.

Your Mapper class function "map" is executed for every key-value pair of the input split, which basically means for each key-value pair returned by the ***RecordReader*** defined by ***InputFormat***. For each of them

you generate some output and write it to the Context object provided as a parameter of “map” function.

The class responsible for collecting the “map” output is specified by ***mapreduce.job.map.output.collector.class*** property and defaults to ***org.apache.hadoop.mapred.MapTask$MapOutputBuffer*** implementation. Later in this step I will describe this specific implementation as no other implementation is shipped with Hadoop at the moment.

“Map” function output first goes to the the ***getPartition*** function of the ***Partitioner*** class. It takes as input the key, the value and the amount of reducers (amount of partition) and returns the number of partition this K-V pair should go to. The default implementation is the following:

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {
public int getPartition(K key, V value, int numReduceTasks) {
return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
}
}
```

Then the output together with the partition number is written to the circular buffer in memory, which size is defined by the parameter ***mapreduce.task.io.sort.mb*** (defaults to 100MB) – it is the total amount of memory allowed for the map output to occupy. If you do not fit into this amount, your data would be spilled to the disk. Be aware that with the change of the default ***dfs.blocksize*** from 64MB to 128MB even the simplest identity mapper will spill to disk, because map output buffer by default is smaller than the input split size.

Spilling is performed in a separate thread. It is started when the circular buffer is filled by ***mapreduce.map.sort.spill.percent*** percent, by default 0.8, which with the default size of the buffer gives 80MB. By default if your map task for a single input split outputs more than 80MB of data, this data would be spilled to local disks. Spilling works in a separate thread allowing mapper to continue functioning and processing input data while spilling happens. Mapper function is stopped only if the mapper processing rate is greater that spilling rate, so that the memory buffer got 100% full – in this case mapper function will be blocked and wait for the spill thread to free up some space.

Spilling thread writes the data to the file on the local drive of the server where the mapper function is called. The directory to write is determined based on the ***mapreduce.job.local.dir*** setting, which contains a list of the directories to be used by the MR jobs on the cluster for temporary data. One directory out of this list is chosen in a round robin fashion. But the data is not just written to the disk. Before this the sorting is performed, and the algorithm used is a QuickSort. The comparator function is defined in the way that it first compares the partition number and only after that the key value, this way the data is sorted by partitions, and within each partition it is sorted by the key.

After the sorting step the ***Combiner*** is invoked. It is used to reduce the amount of data written to the disk. There is a corner case when sorter and combiner are not invoked – when the size of a single record produced by mapper is too big to fit in memory (greater than the output buffer size). In this case the record would be written directly to the disk, without combiner and sorter. Output of the combiner function is written to the disk.

Regardless of the amount of data mapper has outputted, when it is finished the “Spill” method is called, which means the data would be written to the disk at least once. It is performing the same task – sorts and combines the data, then outputs it to the file.

Each file outputted by the spill thread has an index, which contains information about partitions in each of the files – where the partition starts and where it finishes. These indexes are stored in memory, and the amount of memory dedicated to this task is ***mapreduce.task.index.cache.limit.bytes***, which equals to 1MB by default. If it is not enough to store the index in memory, the indexes for all the next spill files created would be written to the disk together with the spill file.

When both mapper and the last spilling is finished, the spill thread is terminated and the merge phase starts. During the merge phase, all the spill files should be grouped together to form a single map output file. By default a single merge process can process up to 100 spill files (the parameter responsible for this is ***mapreduce.task.io.sort.factor***). If the amount of spills are greater than this, subsequent merges would be executed to merge all the spills to a single file.

During the merge, if the amount of files being merged is more or equal to ***min.num.spills.for.combine*** (3 by default), then the combiner would be executed on top of the “merge” result before writing it to the disk.

The result of the MapTask is a single file containing all the output data of the mapper together with the index file describing the partitions start-stop information for the ReduceTask to be able to easily fetch the part related to the reducer it would run.

Now let’s describe the ReduceTask. Unlike the mappers, the amount of reducers should be specified by the ***mapreduce.job.reduces*** parameter, which defaults to 1. The overall logic of the shuffle and merge performed before the reducer will start is defined by the class implementing “Shuffle Consumer Plugin” which is defined by ***mapreduce.job.reduce.shuffle.consumer.plugin.class*** property and defaults to ***org.apache.hadoop.mapreduce.task.reduce.Shuffle***. This is the only implementation shipped with Hadoop, so in the later description I would describe only it.

The first thing that the reduce side is doing is starting the “Event Fetcher” thread, which polls the application master for the status of the mappers and listening to the events of mapper execution finish. When it is finished, this information is passed to one of the “Fetcher” threads. The amount of “Fetcher” threads is defined by ***mapred.reduce.parallel.copies*** and defaults to 5, which means that a single reduce task might have 5 threads copying data from the mappers in parallel. The fetch is performed using HTTP or HTTPS protocol, the fetcher is connecting to the related datanode URL.

All the data the “Fetcher” downloads from the Mapper side is stored in memory. The amount of memory allocated for this is equal to ***mapreduce.reduce.shuffle.input.buffer.percent*** out of the whole reducer memory, which is set by ***mapreduce.reduce.memory.totalbytes.*** Having these values default to 0.7 and 1024MB respectively, the total amount of map outputs a single reducer can store in memory by default is 716MB. If this amount of memory is not enough, the fetchers start to save the map outputs to the local disks on the reducer side, in one of the ***mapreduce.job.local.dir*** directories.

Mergers are following the fetchers. But they don’t wait for the whole fetching process to finish, they are working in a separate threads. There are 3 types of mergers implemented in Hadoop, and each one of them

works in a single separate thread:

1. ***InMemory*** merger. Cannot be disabled, triggered when then memory buffer occupied by the MapTask outputs fetched by this task reaches ***reduce.shuffle.merge.percent*** of the total memory allowed to be occupied by this buffer. Executes combiner after the merge. The output is written to the disk. Always invoked at least once;
2. ***MemToMem*** merger. Can be enabled with the setting ***reduce.merge.memtomem.enabled*** (disabled by default). It merges the mapper outputs located in the memory and writes the output back to the memory. It is triggered when the amount of distinct MapTask outputs reaches ***mapreduce.reduce.merge.memtomem.threshold***, equal to 1000 by default;
3. ***OnDisk*** merger. Merges the files located on the disks, triggered when the amount of files increases (2 * ***task.io.sort.factor*** – 1), but does not merge more than ***mapreduce.task.io.sort.factor*** files in a single run.

All of these mergers are followed by the fourth one called ***finalMerge***. This merge is performed in the main thread of the reducer and in a single run groups together all the MapTask outputs left in memory with all the files left on the local disks created by either InMemory or OnDisk mergers. The output data of the final merge is split between the RAM and disks. The amount of RAM allowed to be utilized as a reducer input is ***mapred.job.reduce.markreset.buffer.percent*** of the total reducer heap size, but at the moment it defaults to 0.

After all of this preparation the reducer is started and its output is written directly to the HDFS.

Here is the whole description. Feel free to add your comments if I have missed something.

This entry was posted in Hadoop and tagged architecture, hadoop, mapreduce on December 17, 2014 [https://0x0fff.com/hadoop-mapreduce-comprehensive-description/] .

51 thoughts on “Hadoop MapReduce Comprehensive Description”

Priyank Parihar
April 20, 2015 at 9:18 am

thanks a lot 🙂

sunil
April 29, 2015 at 4:53 am

thank you very much for explanation…but plz put the image in bigger size….its not viewable

0x0FFF Post author
May 5, 2015 at 9:18 am

The big image is available by clicking on the small one, or simply follow this link: https://0x0fff.com/wp-content/uploads/2014/12/MapReduce-v3.png

sunil
April 29, 2015 at 5:02 am

your blog says partioning and filling the circular buffer but some books says map output sends to buffer then partition and spill, which one is correct…plz correct me….

0x0FFF Post author
May 5, 2015 at 9:26 am

You can take a look at the code here: https://github.com/apache/hadoop/blob/branch-2.7.0/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapred/MapTask.java, classes OldOutputCollector and NewOutputCollector. Both of them are calling partitioner.getPartition on insertion the data to the buffer, so “partitioning” itself happens immediately before the data is inserted into the circular buffer

Amit
July 5, 2015 at 9:46 am

Hi I am getting confused with other articles with regards to sequence of processing . i am non java user but wanted to understand the real exact sequence for map reduce .
Can you please help ?

Thanks

Amit
July 5, 2015 at 10:04 am

Also please look into the white paper located at “https://hadoop-toolkit.googlecode.com/…/White%20paper-HadoopPerfo”

0x0FFF Post author
July 6, 2015 at 8:27 am

My description does not conflict with the description given in the article you are referencing to. They are simplifying and omitting some steps for you to get a better understanding of what is happening there on the high level, while my article covers more in-depth steps happening in the MR process. Also be aware that the article you are referencing to is from 2009 year, while my article is from late 2014 – many changes has happened to the code and logic

---

Abhay Godbole
May 5, 2015 at 4:58 pm

Hi Alexey ,
Excellent article, first time I have read such a detail explanation after Tom White. Thanks for sharing it.
I have one doubt about the split. When file is stored on HDFS, it gets split into blocks as per the dfs.blocksize. My question is when mapreduce job takes it for processing, the InputFormat’s RecordReader splits it again as per the record boundaries or use the same HDFS splits?

Please get back

Regards
Abhay

0x0FFF Post author
May 5, 2015 at 6:19 pm

Thank you! I've written this article because I wasn't satisfied with the description given by Tom White plus I didn't find a good answer for my questions on stackoverflow.

Regarding your question: data in HDFS is stored in blocks split by binary boundaries, each block is exactly 128MB (by default). When the data is processed, you need to process logical records, not the binary blocks. The way to split the data into logical records is determined by the input format – for TextInputFormat one logical records is a string ending by newline character. This way InputFormat is responsible for splitting the data into input splits, each input split containing even number of logical records. For TextInputFormat it is implemented this way: whole first block of the file is the first input split + the first characters of the second block ending by the first newline character there. Second input split is the block of data starting from the first newline character in the second HDFS block and finishing with the first newline character in the third HDFS block (or EOF). And so on. For sequence files it is a bit different, they have special sync-markers for the InputFormat to be able to effectively split the data into logical records

---

Bhagyashree B
August 6, 2015 at 11:21 am

Excellent Article!!
Well Written.

lokesh
August 12, 2015 at 9:28 am

Thank you very much sir,excellent article

Ankit
September 2, 2015 at 10:54 pm

Genius …_/_it took me 3 hours to understand it completely…..what have you referred to go such in depth.?

0x0FFF Post author
September 3, 2015 at 6:51 am

Mostly the source code itself. I didn't find enough information in guides/descriptions/books, so source code was the only source of truth

---

mike
October 13, 2015 at 11:46 pm

does each reducer get all the data? i.e if you have 5 reducers, each 5 reducers gets all the data by all the maps? that seems unclear in diagram. I thought each partition data goes to its corresponding reducer and not all.

0x0FFF Post author
October 14, 2015 at 3:12 pm

No, you are wrong. Each reducer reads only part of each mapper's output related to its partition, which in general might be empty. On the picture I drew the mapper output as 2 files: map output file and index file. Index file shows the reducer how to locate the part of map output file related to its partition

---

Shuxin Lin
October 25, 2015 at 3:11 pm

Really good explanation! it is said that the reducer is always the same as the combiner. but I cant get the point until I read this article about the sort before combiner. Thank you very much!

sudheer
November 5, 2015 at 12:04 pm

Great article.....hoping to see more articles like this on Spark.

Dario
December 3, 2015 at 4:15 pm

Hello Alexey,
great article. After reading TW I was wondering how reducers would know which parts of the mapper output file to read from...indexes, of course!

I have one more doubt: if I would read a mapper output file (which never passed through a combiner function) how would its insides look like?

1) Values would be grouped per key in a long array:
key_A, [value_1, value_2, ..., value_n]
key_B, [value_1, value_2, ..., value_n]
...
key_N, [value_1, value_2, ..., value_n]

2) Values would be ordered per key as simple key/value pairs (until they get to the merge phase on the reducer):
key_A, value_1
key_A, value_2
...
key_A, value_n
key_B, value_1
key_B, value_2
...
key_B, value_n
...
key_N, value_1
key_N, value_2
...
key_N, value_n
?

Thank you
Dario

0x0FFF Post author
December 8, 2015 at 8:41 am

Here is the code responsible for writing map output: https://github.com/apache/hadoop/blob/branch-2.7.1/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapred/IFile.java

As you see from this code, it writes to the file key-value pairs in a following way:
“key length in bytes”, “value length in bytes”, “key”, “value”
“key length in bytes”, “value length in bytes”, “key”, “value”
…

Pingback: Hadoop on Remote Storage | Distributed Systems Architecture

Hrishikesh Tiwary
December 31, 2015 at 7:00 pm

Great Explanation, but i have a question here.. i understand HashPartioner is called by default but does it happen for Combiner also ? does MapReduce call Combiner also by default even if we dont provide ? Combiner has been used at mapper stage when it writes to file and then in Shuffle and Reducer stage. kindly clear this.

0x0FFF Post author
January 4, 2016 at 10:31 am

No, combiner is optional and by default nothing is called. Combiner is a separate piece of code that should be provided by you, and only in special cases (like WordCount) you can reuse Reducer class for it

---

tao
January 8, 2016 at 11:32 pm

I think the property “mapred.job.reduce.input.buffer.percent” decide the amount of RAM allowed to be utilized as a reducer input. Other than “mapred.job.reduce.markreset.buffer.percent”: The percentage of memory -relative to the maximum heap size- to be used for caching values when using the mark-reset functionality.

0x0FFF Post author
January 11, 2016 at 10:01 am

Parameter “mapred.job.reduce.input.buffer.percent” is used only in finalMerge step (https://github.com/apache/hadoop/blob/branch-2.7.2/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/task/reduce/MergeManagerImpl.java#L690), that merges results of memtomem and inmemory mergers. It is the percentage of the memory allowed to remain in memory when the final merge starts, and if the size of the segments in memory is greater than this allowed value they would be spilled to disk (https://github.com/apache/hadoop/blob/branch-2.7.2/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapreduce/task/reduce/MergeManagerImpl.java#L619)

markreset parameter is completely different. It is utilized only when you use MarkableIterator to read the values on the reducer side (http://blog.ring.idv.tw/comment.ser?i=373), and is not directly related to the MapReduce implementation. Here is the code for it (https://github.com/apache/hadoop/blob/branch-2.7.2/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-core/src/main/java/org/apache/hadoop/mapred/BackupStore.java), just a different kind of iterator supporting re-iteration, here is some more details on how it works: http://stackoverflow.com/questions/21464720/how-can-i-use-markableiterator-in-hadoop (but don’t mention the details of the question, it is supported now)

john77eipe
January 25, 2016 at 11:13 am

It would be great if you could add examples using a sample data in between your explanations.

Btw, great article and exploration. Examples would have helped more.

0x0FFF Post author
January 26, 2016 at 9:13 am

Thank you for the feedback. Generally speaking, this article is not meant to be an introduction to MapReduce. You can find tons of introductions with examples over the internet. This is a comprehensive description – when you already know how it works and how MapReduce processes the data, to get a deeper insight into what’s happening there

---

manyam
January 26, 2016 at 5:50 pm

Good article and better than the books i read

Ghebrekristos
February 16, 2016 at 6:16 pm

This really is comprehensive and explanatory diagram. Thanks for the explanation.

hunhun
March 9, 2016 at 7:21 am

I have some puzzles.
The code
bufferRemaining = Math.min(distanceTo(bufindex, kvbidx) – 2 * METASIZE,
softLimit – bUsed) – METASIZE;
What is meaning “2*METASIZE”? Why “bufferRemaining” equal that?

Thanks

0x0FFF Post author
March 10, 2016 at 8:17 am

I’m not the one who has written this code (because I’d create a discriptive diagram before writing this), but I suspect that the buffer itself stores only METASIZE bytes of data, or NMETA integer entries (4 integers in current implementation):

```
kvmeta.put(kvindex + PARTITION, partition);
kvmeta.put(kvindex + KEYSTART, keystart);
kvmeta.put(kvindex + VALSTART, valstart);
kvmeta.put(kvindex + VALLEN, distanceTo(valstart, valend));
```

So each metadata entry is METASIZE bytes long. bufferRemaining is the number of bytes left in buffer before soft limit is reached, i.e. before the background spilling process is started. In this calculation, *“distanceTo(bufindex, kvbidx) – 2 * METASIZE”* means the total capacity of the buffer without 2 records, see the comment *“The write should block only if there is insufficient space to complete the current write, write the metadata for this record, and write the metadata for the next record”* – i.e. we always want to keep at least 2 slots available. The second option under “min” is “softLimit – bUsed”, and this is the calculation of the number of bytes before soft limit (i.e. capacity triggering spilling) is reached. In extreme case with empty

buffer and *io.sort.spill.percent* set to 1.0 (or 100%) this value might equal to the total capacity of the buffer, so the first parameter would be smaller than the second.

Sudhakar
March 14, 2016 at 6:15 pm

Wow. Awesome explanation. We had a long running reducer and was trying to figure out why by looking at the log output. Log output is not really intuitive (obviously) but your explanation will surely simplify this. Awesome job again. And kudos to your documentation and diagrammatic skills as well. Way to go!! You should write a book in the future.

Sudhakar
March 14, 2016 at 11:35 pm

Is the below statement correct? Should it not be a single REDUCER and not mapper?

“Having these values default to 0.7 and 1024MB respectively, the total amount of map outputs a single mapper can store in memory by default is 716MB.”

0x0FFF Post author
March 22, 2016 at 2:05 pm

Thank you for your comment, you are right, I have fixed it in the article

---

Pingback: Some problems and solutions when deploying and running Hadoop-2.7.2 – Robin On Linux

Geoffrey Cheng
May 16, 2016 at 7:55 pm

thank you.

Deepak
May 26, 2016 at 6:51 am

Your article cleared all my doubts.. excellent job .. Thank you for writing such an article.

Zx Yuan
May 30, 2016 at 8:56 am

According to the description of your picture, actually, a Reducer gets its parts of Mapper output at the beginning stage of SHUFFLE. Is that right?

0x0FFF Post author
May 30, 2016 at 9:06 am

There is no clear definition of boundaries between sort, shuffle and merge. Some even call the whole “sort-shuffle-merge” process a “shuffle”, which is not very precise but correct in general. I prefer to think that “shuffle” starts after the “map” tasks generate their indexed output files, because “shuffle” really means shuffling data across the cluster, while no shuffling is performed before the “map” output file is generated

---

sampat kumar
June 22, 2016 at 7:27 am

Awesomely!! explained

gooner87
July 2, 2016 at 7:40 pm

Is number of spills equal to number of map tasks? i.e. circular buffer memory is allocated per map task? Also, in the flow diagram Combiner is after Sort. Isn’t it combiner followed by sort?

0x0FFF Post author
July 2, 2016 at 10:05 pm

No. Each map task has one or more spills. One happens when there is enough space in circular buffer to fit the mapper output.
Circular buffer is allocated per map task
Combiner is effectively doing “reduce” task, i.e. combining different values for the same key. Its input should already be split by key (i.e. >) in order to operate, this is why it consumes sorted data

---

gooner87
July 2, 2016 at 8:09 pm

If we have 1000 spill files then we will have 10 Merge files in the first phase if sort.factor is 100 and in the second phase we will have merge final after merging 10 merge files from the first phase. Am I correct? Also, Since we have merged from 100 spill files there can be entries for the same key hence we are doing combiner after first phase of merging. Am I correct?
Thanks in advance!!!!!!!

0x0FFF Post author
July 2, 2016 at 10:08 pm

Yes and yes. You might have combiner called multiple times on your input, and even partially combined input, this is why combiner function is associative and commutative

---

gooner87
July 3, 2016 at 3:10 pm

Thanks a lot for both the replies……. I might come up with more questions 🙂

Aashish Soni
July 13, 2016 at 9:51 am

Great blog ! I Have question's
1. What is the execution flow of MapReduce when no Combiner and default Hash Partitioner used?
Map-> Partition-> Sort->Shuffle->Sort->reduce
Correct me if I am wrong
2. What is the execution flow of MapReduce when Costum Combiner and custom partitioner is used?
Map-> Partition->Combiner-> Sort->Shuffle->Sort->Reducer
Correct me if I am wrong
3. In case of Identity reducer Shuffle,Sort phase will execute or not ? and happen in case of Identity Mapper?
4. combiner run on the every mapper output right? and where is execution of partition happen in map or reduce?

0x0FFF Post author
July 13, 2016 at 2:38 pm

1. map->partition->sort->{merge}->shuffle->merge->reduce
2. map->partition->sort->combine->{merge->[combine->]}shuffle->{merge->[combine->]}->reduce
3. Shuffle and sort would execute in case of identity mapper, identity reducer and both at the same time. They won't execute only in case of map-only jobs
4. Combiner might run many times on top of map output or not run at all – it depends. Partitioning happens on the mapper side

---

aashish soni
July 14, 2016 at 1:34 pm

Thank you very much for your reply ..

I have some doubts

1. As per your comment merging is happens between sort and shuffle phase(why did you put merge in bracket). As per my understanding I know, merging happens after the shuffle part and also sorting would perform there. Sorting would perform after merging or not?(Is merging and sorting same?)

2. Can you please explain to me this , I am unable to understand.

Can you please guide me .

Thanks

0x0FFF Post author
July 14, 2016 at 1:59 pm

Merging happens before shuffle if mapper produces more than one output file. Merging on the reducer side happens every time.
Sorting is ordering elements in array, while merging is joining N sorted arrays together in a single sorted array

---

aashish soni
July 14, 2016 at 2:05 pm

Thanks !

Could you please to me in more detail ,the execution flow when Costum Combiner and custom partitioner is used?

map->partition->sort->combine->{merge->[combine->]}shuffle->{merge->[combine->]}->reduce

---

0x0FFF Post author
July 14, 2016 at 2:17 pm

I described it both in text and on the picture. Feel free to refer to the source code as well. If you have a more specific question – feel free to ask

---