Prof R. Madana Mohana

BIG DATA ANALYTICS

How MapReduce Works

Shuffle and Sort

https://www.youtube.com/c/RASINENIMADANAMOHANA

Shuffle and Sort: *Outline*

The Map Side

The Reduce Side

Shuffle and Sort

Shuffle and Sort

- **MapReduce** makes the guarantee that the input to every **reducer** is sorted by key.
- The process by which the system performs the **sort** and transfers the **map** outputs to the **reducers** as inputs is known as the ***shuffle***.
- The **shuffle** is an area of the codebase where refinements and improvements are continually being made.
- In many ways, the **shuffle** is the heart of **MapReduce** and is where the "**magic**" happens.

The Map Side

The Map Side

- When the **map** function starts producing output, it is not simply written to disk.
- The process is more involved, and takes advantage of buffering writes in memory and doing some presorting for efficiency reasons.
- The following ***Figure*** shows **Shuffle** and **sort** in **MapReduce**.

The Map Side

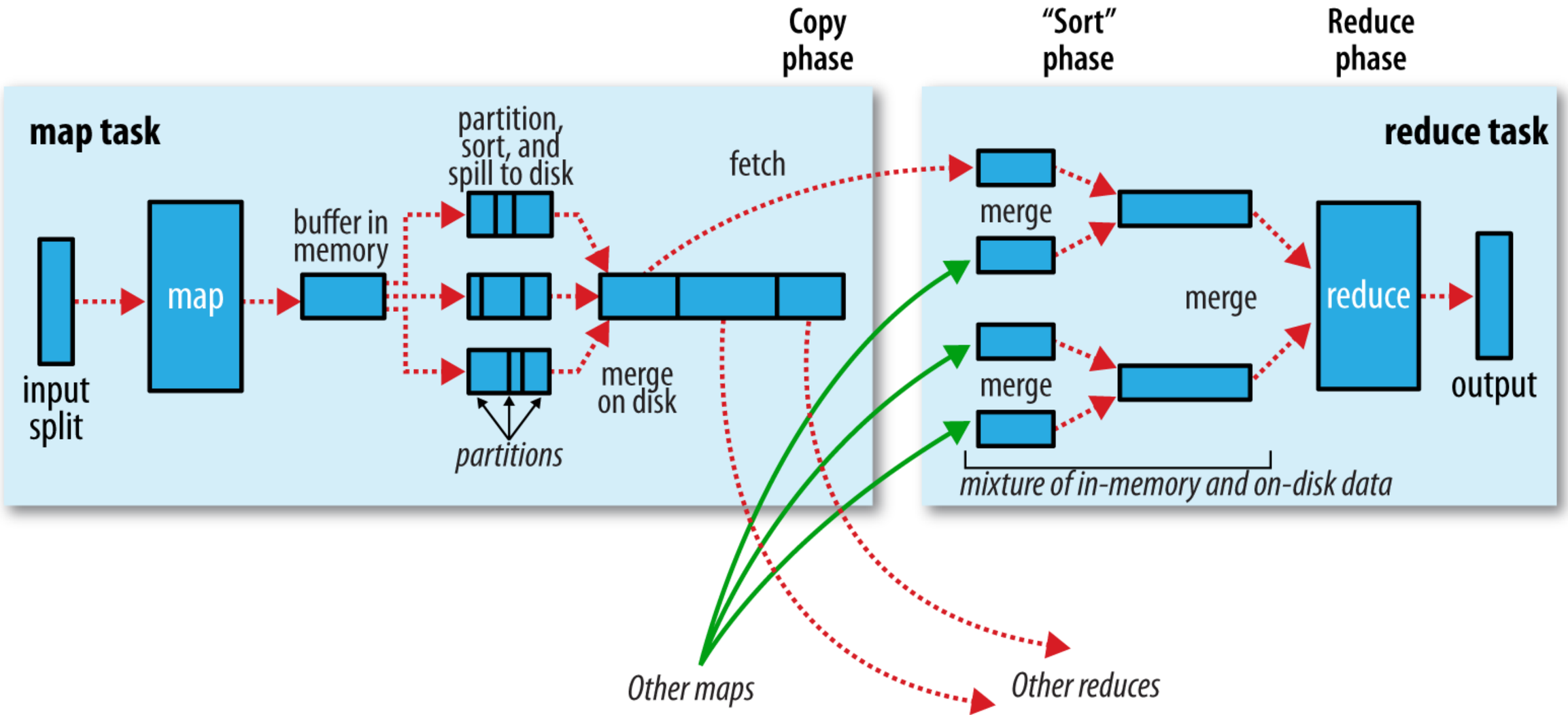


Figure. **Shuffle and sort in MapReduce**

The Map Side

- Each **map** task has a circular memory buffer that it writes the output to.
- The buffer is **100 MB** by default (the size can be tuned by changing the **mapreduce.task.io.sort.mb** property).
- When the contents of the buffer reach a certain threshold size (**mapreduce.map.sort.spill.percent**, which has the default value **0.80**, or **80%**), a background thread will start to ***spill*** the contents to disk.

The Map Side

- **Map** outputs will continue to be written to the buffer while the ***spill*** takes place, but if the buffer fills up during this time, the **map** will block until the ***spill*** is complete.
- ***Spills*** are written in round-robin fashion to the directories specified by the **mapreduce.cluster.local.dir** property, in a job specific subdirectory.

The Map Side

- Before it writes to disk, the thread first divides the data into partitions corresponding to the **reducers** that they will ultimately be sent to.
- Within each partition, the background thread performs an in-memory ***sort*** by key, and if there is a **combiner** function, it is run on the output of the ***sort***.
- Running the **combiner** function makes for a more compact **map** output, so there is less data to write to local disk and to transfer to the **reducer**.

The Map Side

- Each time the memory buffer reaches the ***spill*** threshold, a new spill file is created, so after the **map** task has written its last output record, there could be several spill files.
- Before the task is finished, the spill files are merged into a single partitioned and sorted output file.
- The configuration property **mapreduce.task.io.sort.factor** controls the maximum number of streams to **merge** at once; the default is **10**.

The Map Side

- If there are at least three **spill** files (set by the **mapreduce.map.combine.minspills** property), the **combiner** is run again before the output file is written.
- Recall that **combiners** may be run repeatedly over the input without affecting the final result.
- If there are only one or two **spills**, the potential reduction in **map** output size is not worth the overhead in invoking the **combine**r, so it is not run again for this **map** output.

The Map Side

- It is often a good idea to compress the map output as it is written to disk, because doing so makes it faster to write to disk, saves disk space, and reduces the amount of data to transfer to the **reducer**.
- By default, the output is not compressed, but it is easy to enable this by setting **mapreduce.map.output.compress** to **true**.
- The compression library to use is specified by **mapreduce.map.output.compress.codec**

The Map Side

- The output file's partitions are made available to the **reducers** over **HTTP**.
- The maximum number of worker threads used to serve the file partitions is controlled by the **mapreduce.shuffle.max.threads** property; this setting is per node manager, not per **map** task.
- The default of **0** sets the maximum number of threads to twice the number of processors on the machine.

The Reduce Side

The Reduce Side

- The *map output file* is sitting on the local disk of the machine that ran the **map task** (note that although *map outputs* always get written to local disk, *reduce outputs* may not be), but now it is needed by the machine that is about to run the **reduce task** for the partition.

The Reduce Side

- Moreover, the **reduce task** needs the *map output* for its particular partition from several *map tasks* across the cluster.
- This is known as the ***copy phase*** of the **reduce task**.
- The **reduce task** has a small number of copier threads so that it can fetch *map outputs* in parallel.
- The default is **five** threads, but this number can be changed by setting the **mapreduce.reduce.shuffle.parallelcopies** property.

The Reduce Side

- *Map outputs* are copied to the **reduce task JVM**'s memory if they are small enough (the buffer's size is controlled by **mapreduce.reduce.shuffle.input.buffer.percent**, which specifies the proportion of the heap to use for this purpose); otherwise, they are copied to disk.

The Reduce Side

- When the in-memory buffer reaches a threshold size (controlled by **mapreduce.reduce.shuffle.merge.percent**) or reaches a threshold number of *map outputs* (**mapreduce.reduce.merge.inmem.threshold**), it is merged and spilled to disk.
- If a **combiner** is specified, it will be run during the merge to reduce the amount of data written to disk.

The Reduce Side

- As the copies accumulate on disk, a background thread **merges** them into larger, sorted files. This saves some time *merging* later on.
- Note that any *map outputs* that were compressed (by the map task) have to be decompressed in memory in order to perform a **merge** on them.

The Reduce Side

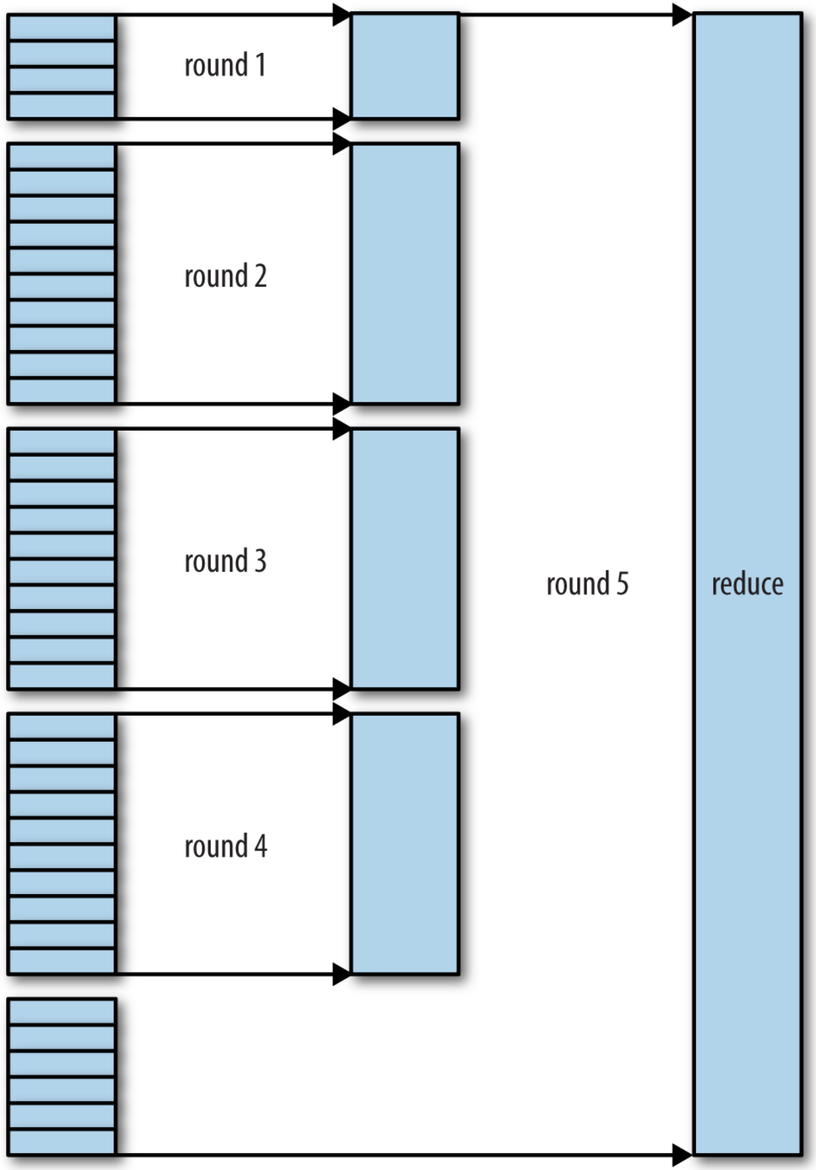
- When all the *map outputs* have been copied, the **reduce task** moves into the **sort phase** (which should properly be called the **merge phase**, as the sorting was carried out on the *map side*), which merges the *map outputs*, maintaining their sort ordering. This is done in rounds.
- For example, if there were **50** *map outputs* and the ***merge factor*** was **10** (the default, controlled by the **mapreduce.task.io.sort.factor** property, just like in the *map's merg*e), there would be **five** rounds.
- Each round would merge **10** files into **1**, so at the end there would be **5** intermediate files.

The Reduce Side

- Rather than have a final round that *merges* these **five** files into a single sorted file, the *merge* saves a trip to disk by directly feeding the **reduce** function in what is the last phase: the **reduce phase**.
- This *final merge* can come from a mixture of in-memory and on-disk segments.

The Reduce Side

Ex. *Efficiently merging 40 file segments with a merge factor of 10*



The Reduce Side

- During the **reduce phase**, the *reduce* function is invoked for each key in the sorted output.
- The output of this phase is written directly to the output ***filesystem***, typically **HDFS**.
- In the case of **HDFS**, because the node manager is also running a datanode, the first block replica will be written to the local disk.