MapReduce over HDFS

An Introduction

(Spark jobs execute in similar fashion)

Summary from White, T. *Hadoop: The definitive guide*. "O'Reilly Media, Inc.", 2014.

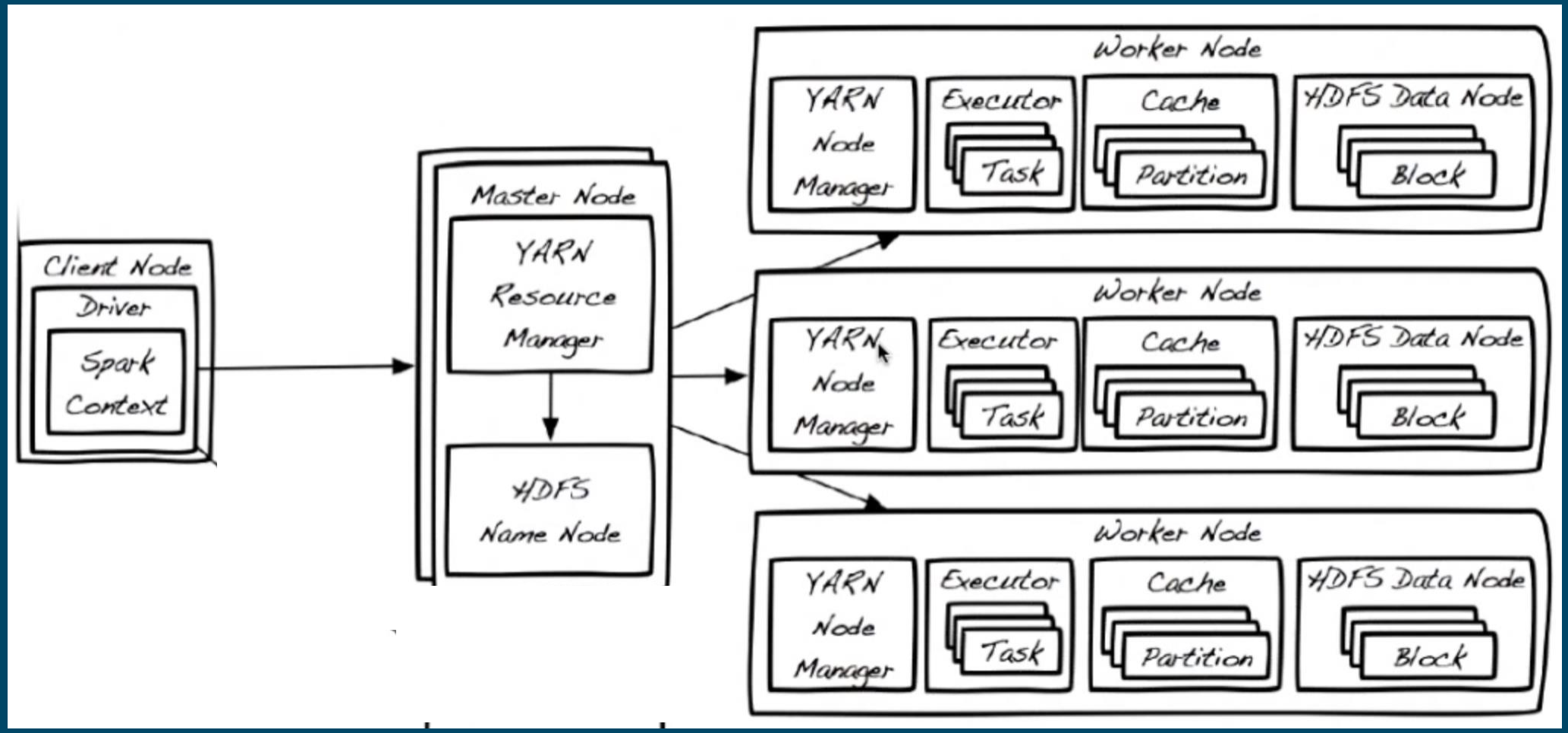
MapReduce execution summary

- Executes Map tasks close to the input data
- Sorts Map output before sending to a Reducer
  - Multiple map outputs are merged within the Reducer
    - input is sorted by key
- Framework performs these tasks efficiently, using a mix of memory and disk buffers

Some review

Let’s assume that we are doing
parallel processing

Hadoop Master/Worker Architecture



How do **tasks execute** over the data network?

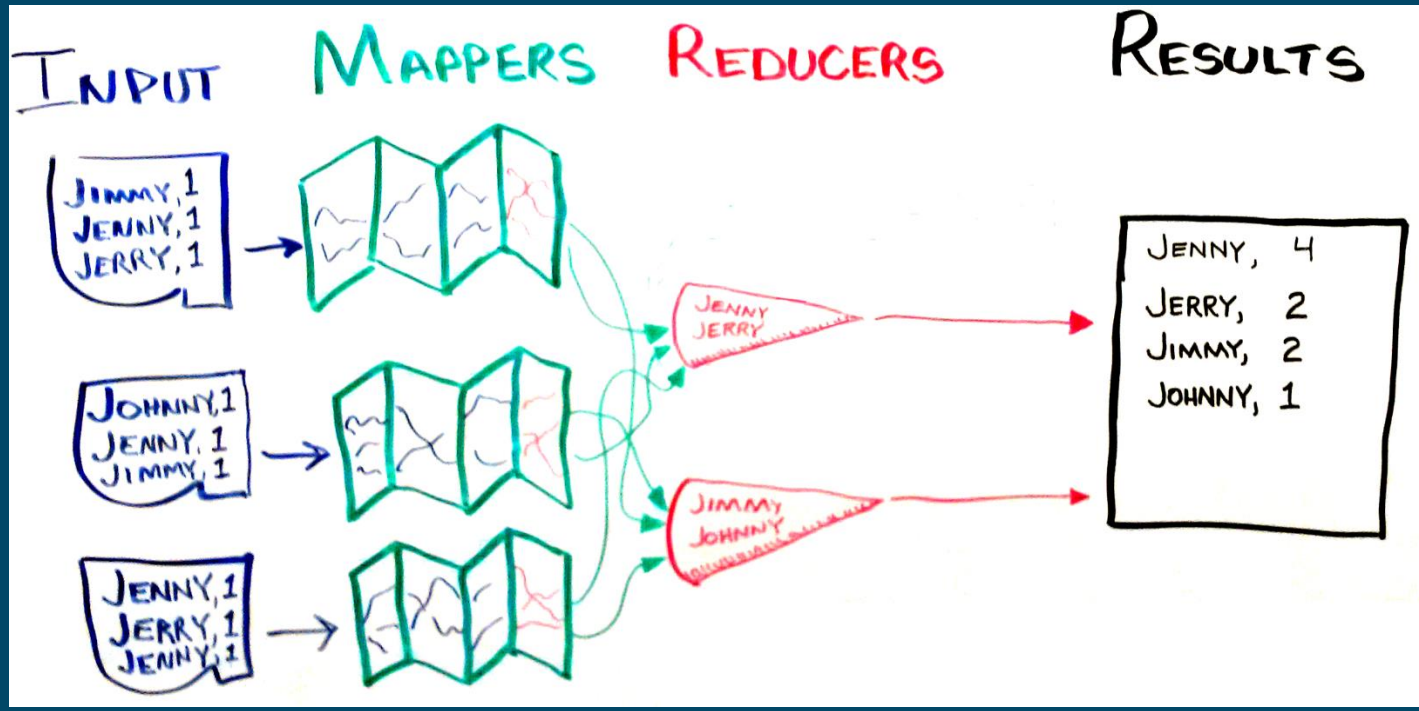
Recall basic terms

- Job
  - A unit of work that processes client input and provides results to the client
- Task
  - Smaller units of work, derived from the job, such as map or reduce (kinds of) tasks. Task processes a partition, typically.
- File system block
  - The smallest unit of bytes tracked by the file system. By default becomes a partition.
  - HDFS block is 128 MB by default. (Windows FS is typically 4K)
- Splits
  - A partition of the larger file
    - e.g., HDFS block
- Splits into input records (MapReduce)
  - Code may need to translate incoming block to records (e.g., read next text line)
  - API InputFormat (sub-class of FileInputFormat) does split record; for text data each line is a split.
  - (Spark does this dependent on file type.)

MapReduce is the original method for executing tasks over HDFS

Map (all data) and then Reduce

- Data input is split into Maps (in parallel)
  - (Key →Value) sets
- The Maps are reduced by (parallel) computation
  - Average(k, v)
- Reducer input data
  - is sorted first



Map Reduce over HDFS

This Yarn/HDFS explanation is also common to Spark

1. Client requests Resource Manager execute a job
2. Resource Manager (YARN)
    1. Obtains file block locations (meta-data from HDFS Name Node)
    2. Computes input splits (API InputFormat)
        1. Depends on kind of file; for text data each line is a split
    3. Requests Data Node Manager start job (with specified resources)
3. Application Manager (in data node)
    1. Starts a map task for each split, and some reduce tasks
        1. Map task placed close to its data (according to meta-data)
    2. Monitors execution, restating tasks as necessary

MapReduce Execution

1. Client (your program) submits job to (Yarn) RM

5a. RM plans resources, starts application in NM

5b. NM starts AM, which manages tasks created in NM's

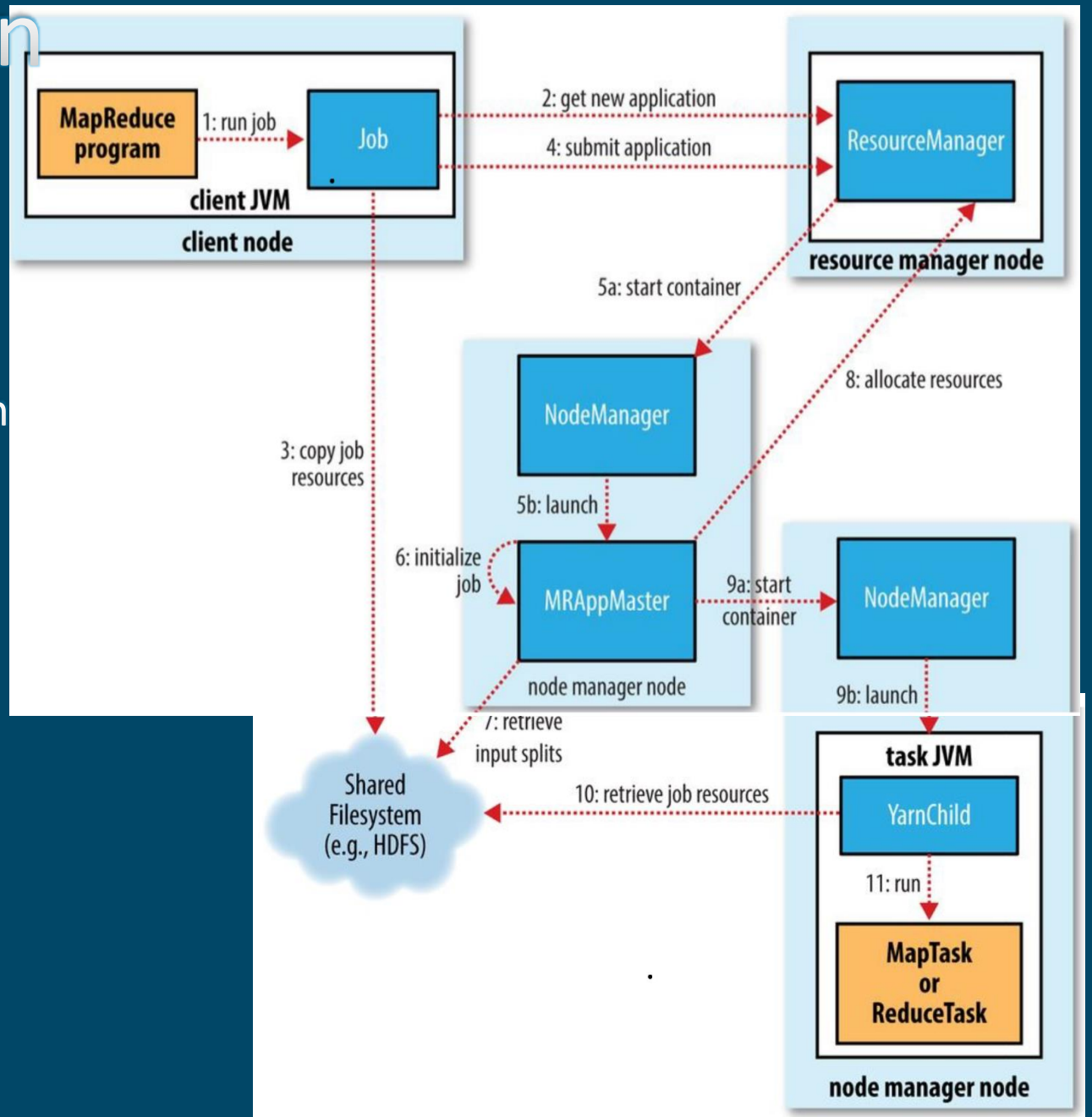
9b. YarnChild preps resources and runs (map or reduce) task

12. Status is sent back to each manager

YarnChild -> AM -> RM -> client

Restarts occur at various levels as necessary

Yarn is this RM, thus the AM and YarnChild are Yarn components



Now, some details…

Input Splits

- Abstract parallel processing unit
- FileInputFormat creates splits from files
- The split size is normally the size of an HDFS block, which is appropriate for most applications;
- however, it is possible to control this value by setting vari

MR Split
is known as
partition in Spark

Property name	Type	Default value	Description
mapred.min.split.size	int	1	The smallest valid size in bytes for a file split.
mapred.max.split.size[a]	long	Long.MAX_VALUE, that is 9223372036854775807	The largest valid size in bytes for a file split.
dfs.block.size	long	64 MB, that is 67108864	The size of a block in HDFS in bytes.

Text Input

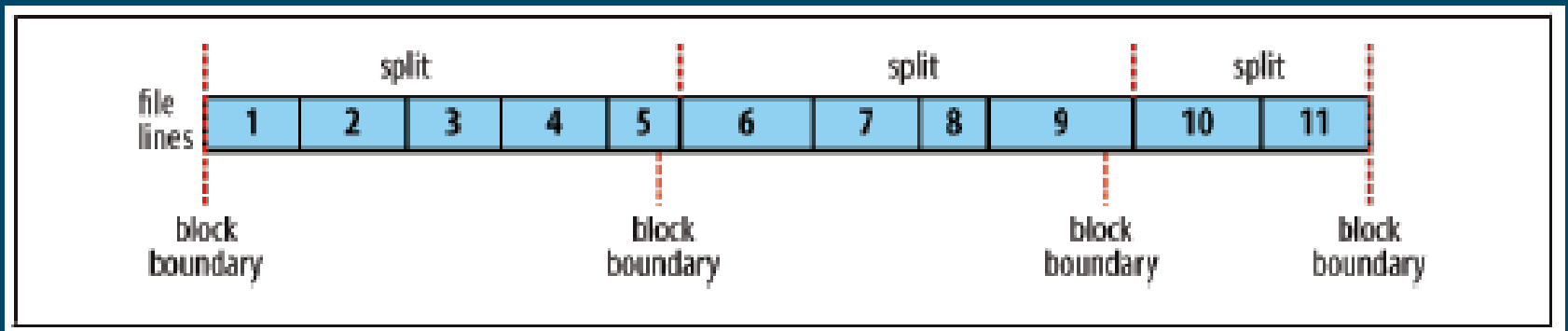


Figure 7-3. Logical records and HDFS blocks for TextInputFormat

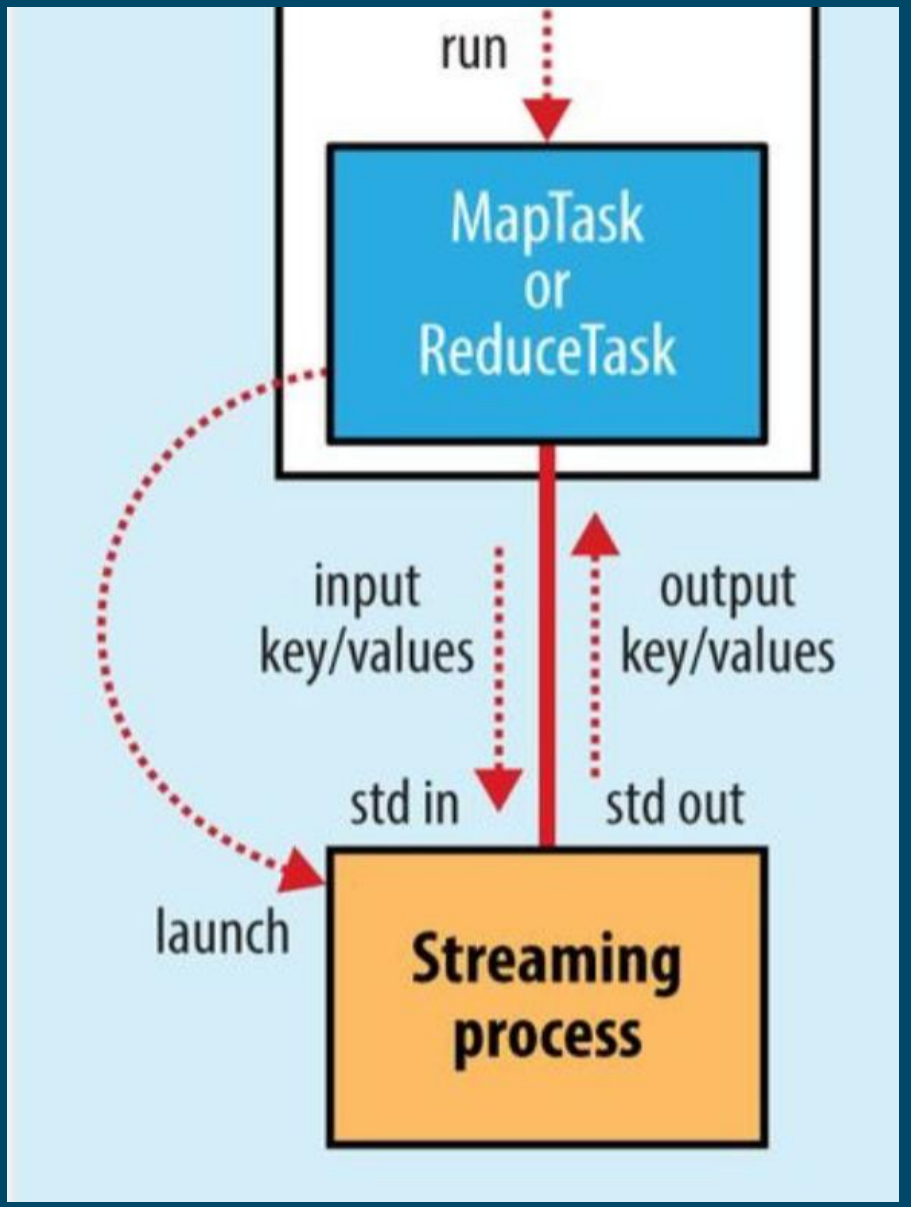
- TextInputFormat is the default InputFormat
- Each record is a line of input
- The key, a LongWritable, is the byte offset within the file of the beginning of the line.
- The value is the contents of the line

Streaming and Uberized tasks..

Explaining uberized tasks will reveal the overhead of the architecture...

Streaming tasks

- For streaming
  - Continually provides records to the MR process
  - Streaming I/O task runs outside of the MR framework
    - E.g., Kafka



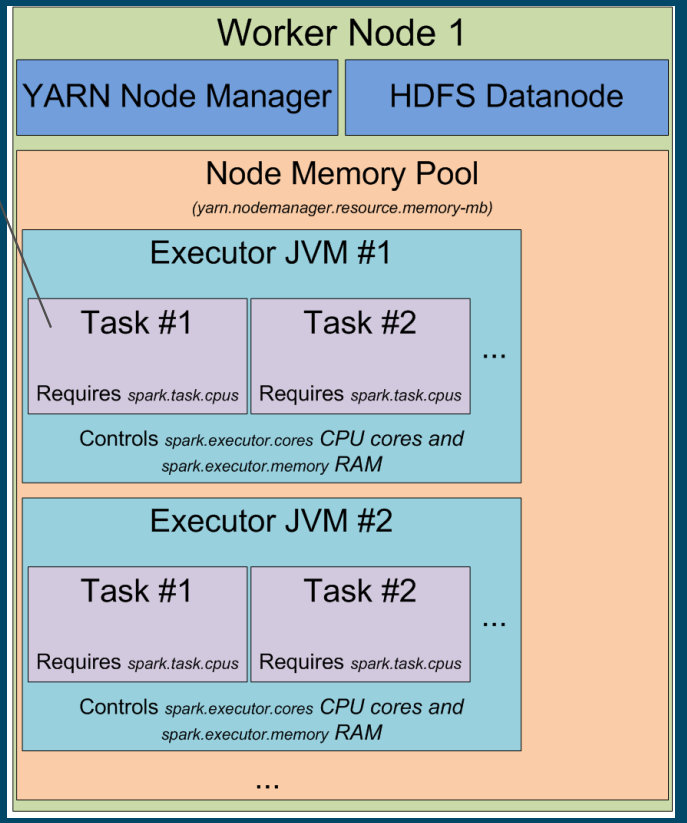
Uberized task

- Two choices for small task
  1. Create the usual task in a JVM (one per split) on each Data Node
     - This occupies the resources of multiple data nodes
  2. Run tasks in sequence on one JVM
     - This occupies the resources of just one data node
     - Likely only when simple map task followed by one reducer

Job tasks execute within a JVM

- The application master decides how to run the tasks of a given job
  - If the job is small, the application master may choose to run the tasks in the same JVM
  - Such *uberized* tasks runs sequentially on one node in one JVM, because that has less overhead than running tasks in new containers over multiple nodes running in parallel
- If the job has less than 10 map tasks, then the tasks are uberized (by default)
- Also, can run duplicate (**speculative**) tasks
  - When task status suggests it's about to fail

>1 task / executor = uberized
Otherwise, 1 task per executor



Example of
How many tasks can be run in parallel

- A cluster with 12 nodes, 64GB of RAM each and 32 CPU cores each (16 physical cores with hyper threading).
- On each node you could start 2 executors having 12 cores with 26GB of RAM each
  - RAM and core per executor are YARN configuration settings
  - leave some RAM and cores for system processes,
    - YARN NM and DataNode
    - Underlying Linux OS
- In total, this cluster configuration would handle 12 machines * 2 executors per machine * 12 cores per executor / 1 core for each task = 288 task slots

MapReduce vs Spark Tasks

- MapReduce
  - Each task runs inside a JVM, sequentially
  - A new job causes a new JVM to be created for the tasks that run within
    - Slow to create new JVM (but can uberize small tasks)
- Spark
  - Each task runs as a thread inside a JVM
  - New tasks are created by forking new threads in an existing JVM
    - Forking within a JVM is faster than a new JVM

An uberized task is not that important, but it points to the issue that **Hadoop tasks have overhead**, which the framework tries to eliminate

```
00:45:43,041 - Parsing command: select 'A' from dual where 1=1
00:45:44,184 - Starting command: select 'A' from dual where 1=1
00:45:45,232 - Connecting to ResourceManager (by client)
00:45:48,459 - Submitted application
00:45:52,148 - Created MRAppMaster
00:45:55,742 - Connecting to ResourceManager (by AM)
00:45:58,184 - ContainerLauncher - CONTAINER_REMOTE_LAUNCH
00:45:58,246 - Transitioned from ASSIGNED to RUNNING
00:46:01,195 - JVM given task
00:46:04,181 - Progress of TaskAttempt is : 0.0
00:46:04,595 - Progress of TaskAttempt is : 1.0
00:46:04,677 - Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.85 sec
00:46:06,820 - Ended Job
Time taken: 23.8 seconds, Fetched: 1 row(s)
```

- 75% Overhead!
  - 1 second to parse the query
  - 9 seconds to submitting the query and launching ApplicationMaster (00:45:43 – 00:45:52)
  - 6 seconds to initialize and launch the container for Map task (00:45:52 – 00:45:58)
  - 3 seconds to initialize JVM (00:45:58 – 00:46:01)
  - 6 seconds for actual MapReduce and cleanup (00:46:01 – 00:46:07)
    - 6/23.8 = 25%

Hadoop has overhead, but a Hadoop cluster has resources. Use them!

If a task is on a **non-responding node, then start another version of the task**,
that is run a **speculative task**

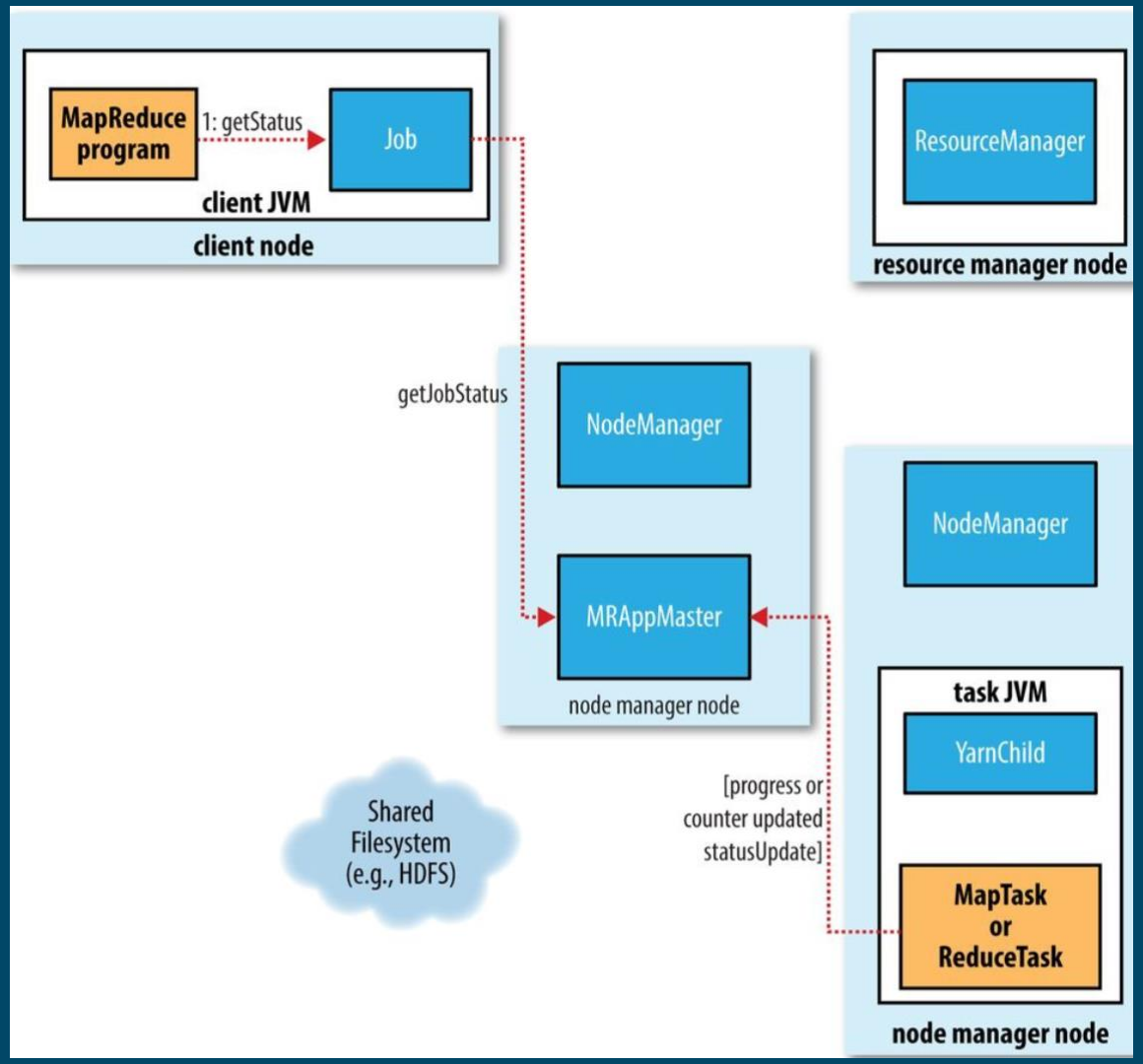
How is the job (its tasks) monitored?

Status is sent back to each **manager**

- YarnChild -> A**M** -> R**M** -> client

Failure of

- Task is noticed and restarted by AM or NM
- AM or NM is noticed by RM (and restarted)
- RM addressed on high availability (HA) mode
  - a pair of RM's watch each other



What happens when we only have one Data Node?

The tasks can be run in sequence using one JVM,
or a network can be simulated

Single node cluster : By default, Hadoop is configured to run in a non-distributed or standalone mode, as a single Java process. There are no daemons running and everything runs in a single JVM instance. HDFS is not used.

Pseudo-distributed or multi-node cluster: The Hadoop daemons run on a local machine, thus simulating a cluster on a small scale. Different Hadoop daemons run in different JVM instances, but on a single machine. HDFS is used instead of local FS

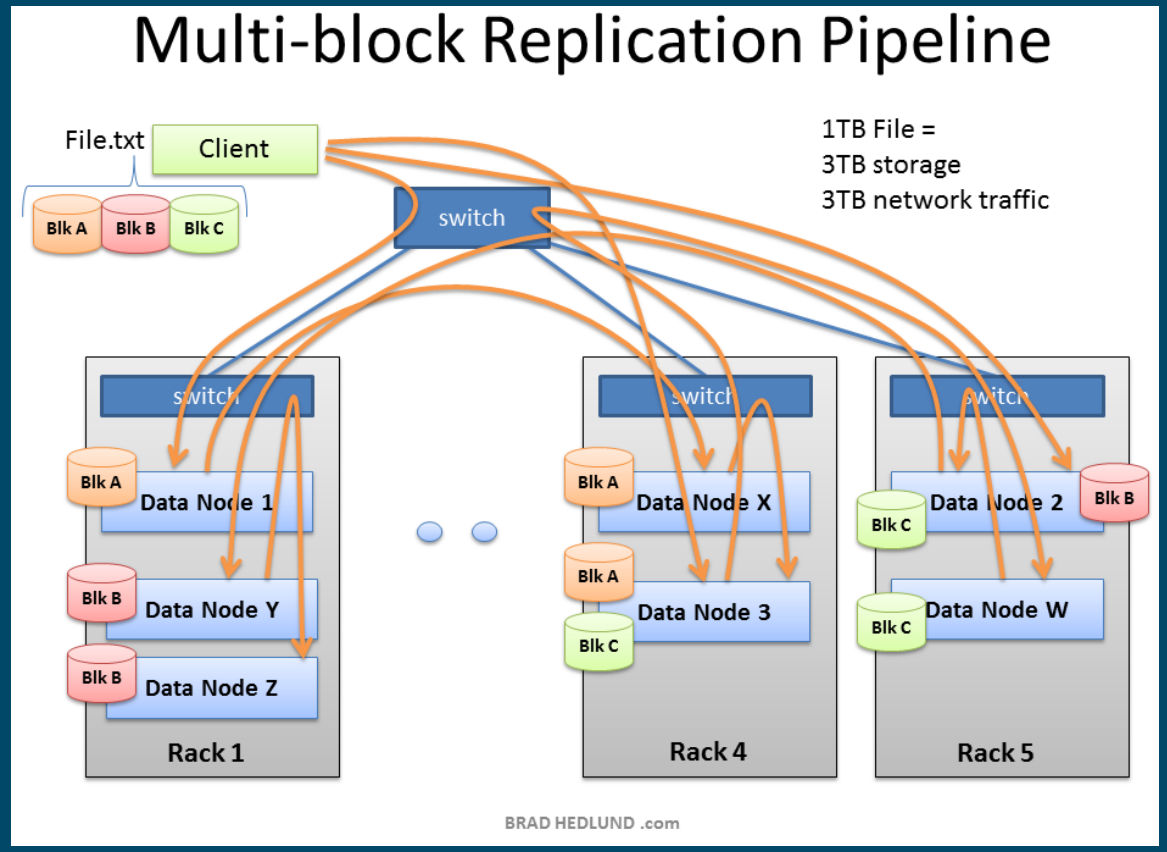
In MapReduce, data is split and then computed, but

…

How do results from parallel tasks get combined?

Map tasks pass their results to Reduce tasks, in a particular manner

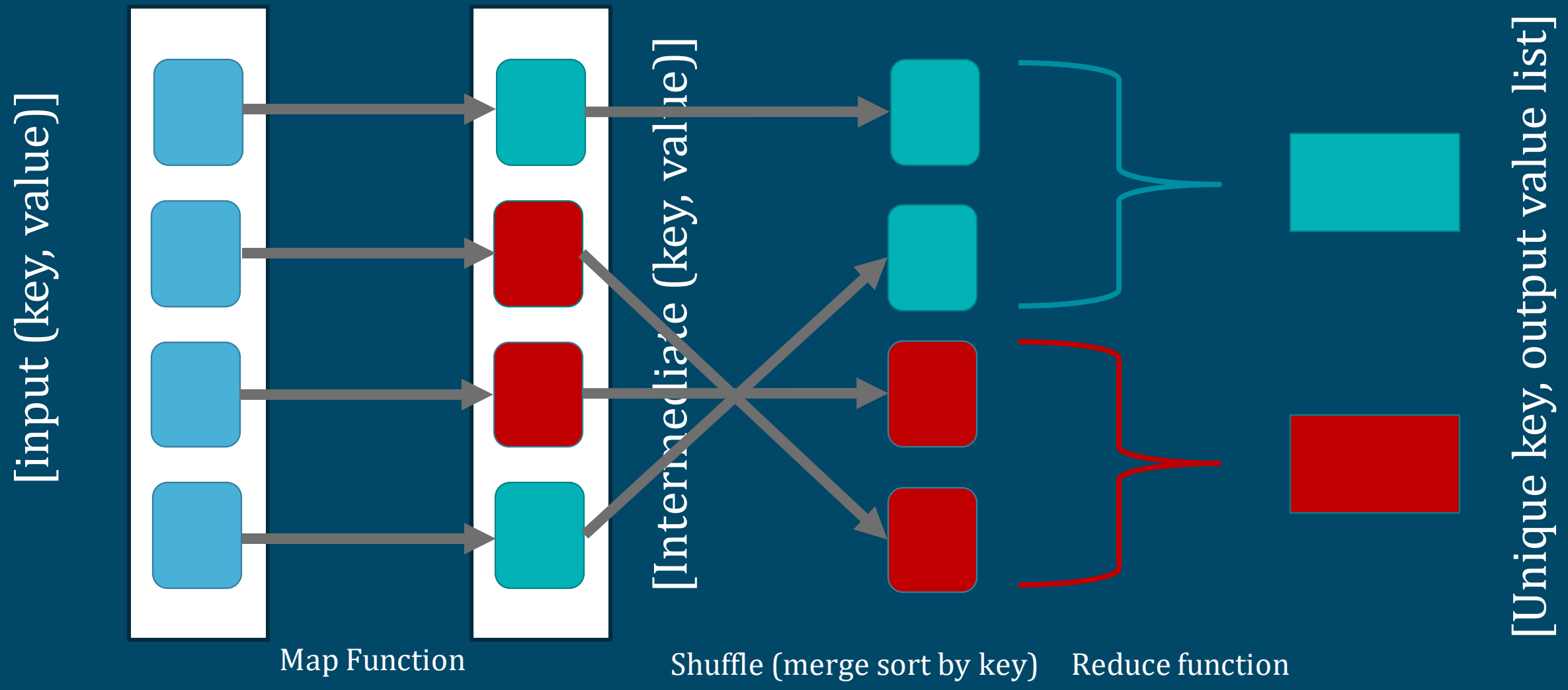
- Transfers through switch (of rack) are slower
- Blocks A & B can be combined within rack
- Blocks B & C could be combined within rack
- If possible, within rack is desired, but otherwise data will be transferred off-rack
- Using location data from Named Node an execution plan for locality is created



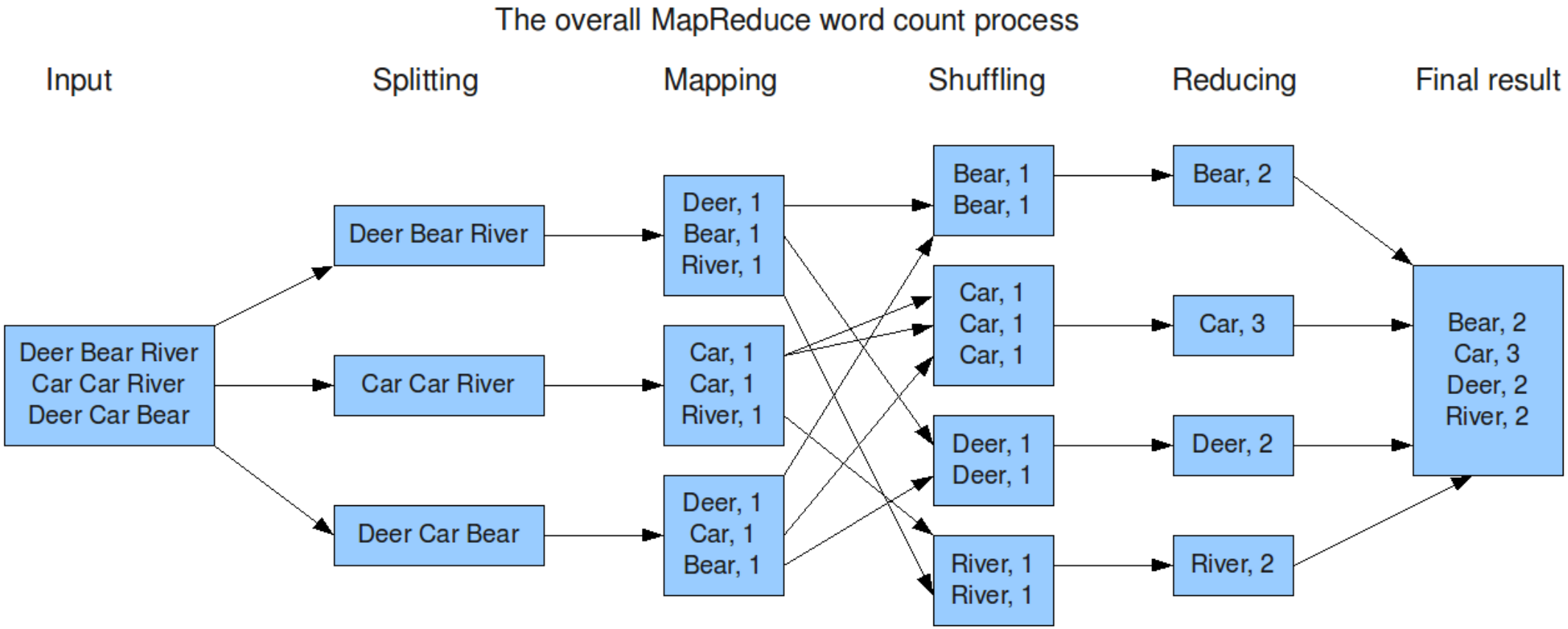
Consider the classic example of
Word Count

derived from the real-world application indexing all web pages

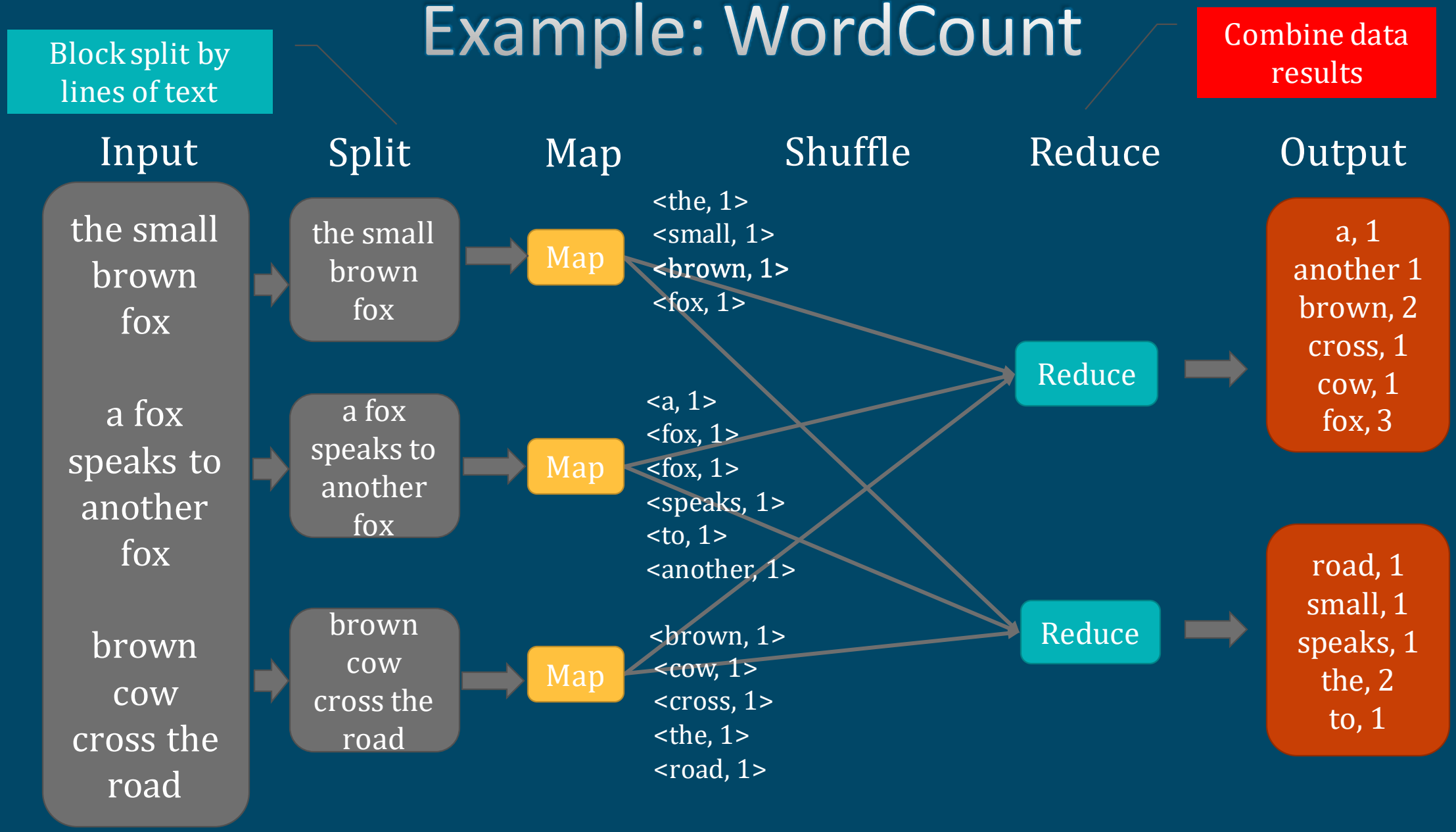
Programming Model



MapReduce: Word Count

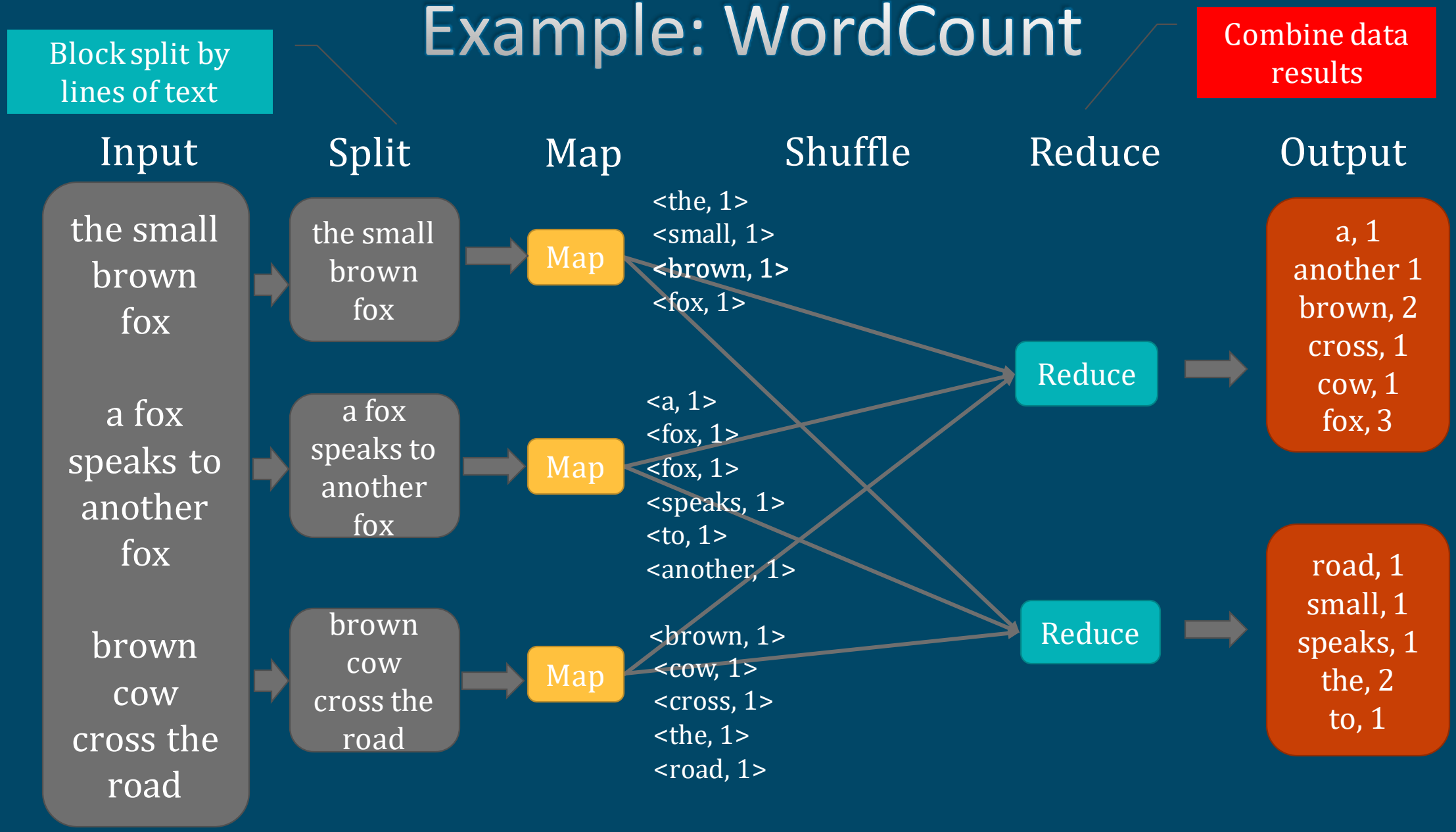


Example: WordCount

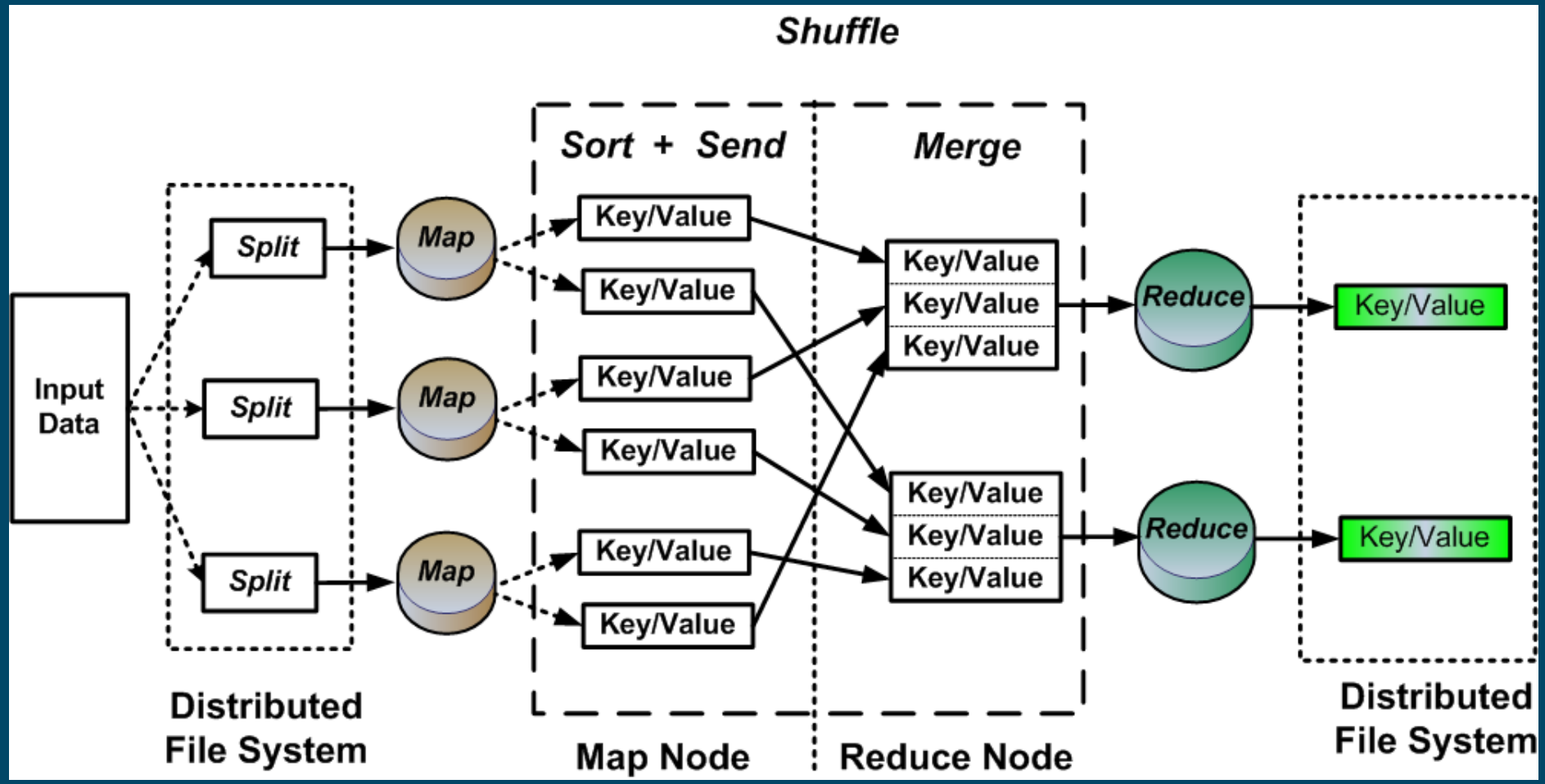


Each Map task can run on its own Data Node, thus
the first block
is processed on Node A
and the second block is processed on Node B

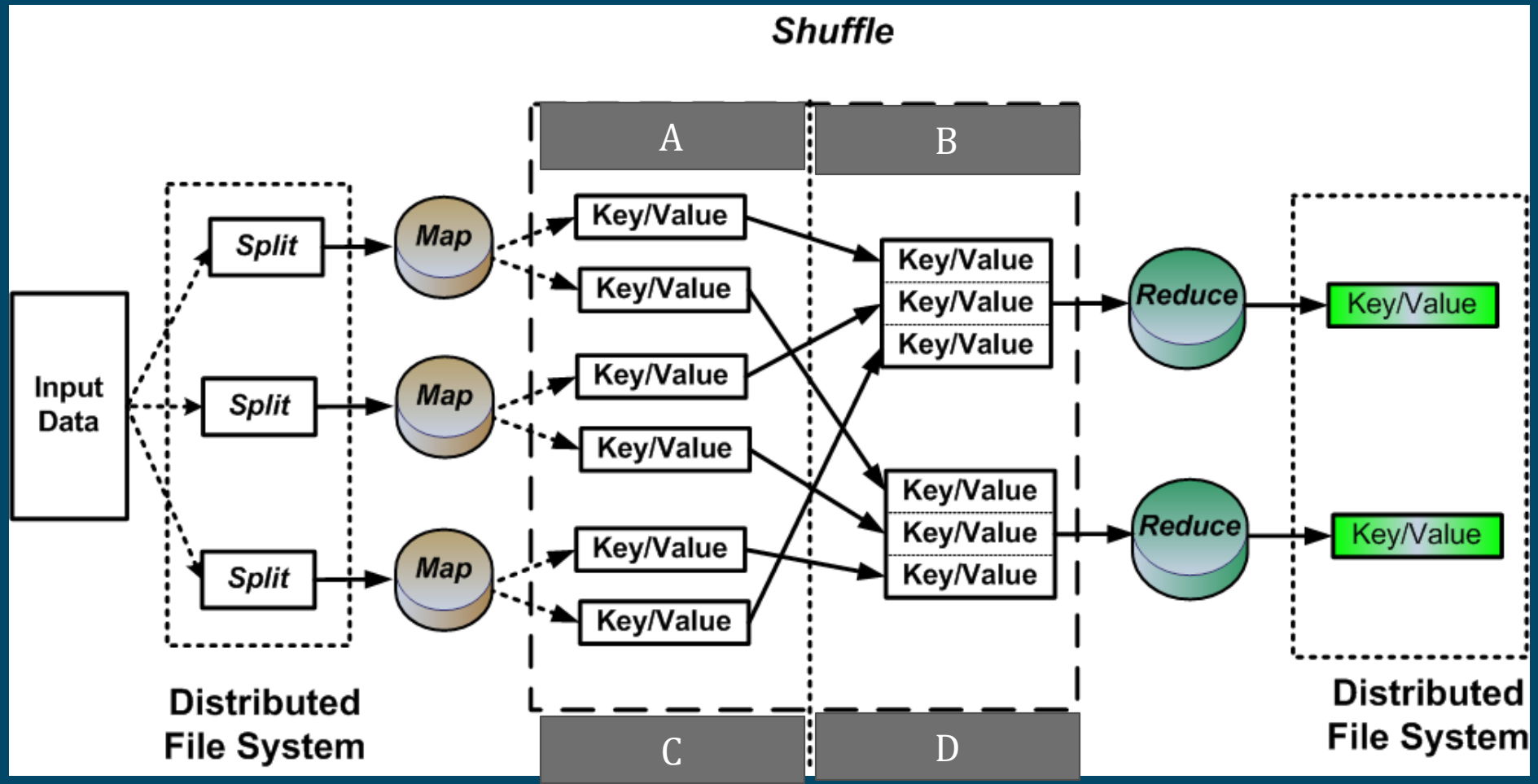
Example: WordCount



Map Reduce provides unique value through Shuffle & Sort



Map Reduce provides unique value through Shuffle & Sort



When the map task(s) are complete on a node then the framework **sorts the resulting values before sending** them onto the reduce task

The framework guarantees **the input to every reducer is sorted by key**

- Map

```
for line in sys.stdin:
    for word in line.strip().split():
        print "%st%d" % (word, 1)
```

This is a Python conceptual example
Note: MR only runs in Java

Block/partition input, which is split to line records

- Reduce

```
current_word = None
current_count = 1

for line in sys.stdin:
    word, count = line.strip().split('t')
    if current_word:
        if word == current_word:
            current_count += int(count)
        else:
            print "%st%d" % (current_word, current_count)
            current_count = 1

    current_word = word

if current_count > 1:
    print "%st%d" % (current_word, current_count)
```

Don't need a table lookup; Input words are sorted

We know that we now have moved onto the next Word

Real Map Task (Java)

```
#!/usr/bin/env python
"""mapper.py"""

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

Reduce Task (Java)

```
current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Reduce function is called exactly once per key (non-streaming)

This reduce function (for MR) has the key as an argument

- It's called once per key
- This simplifie

```
public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

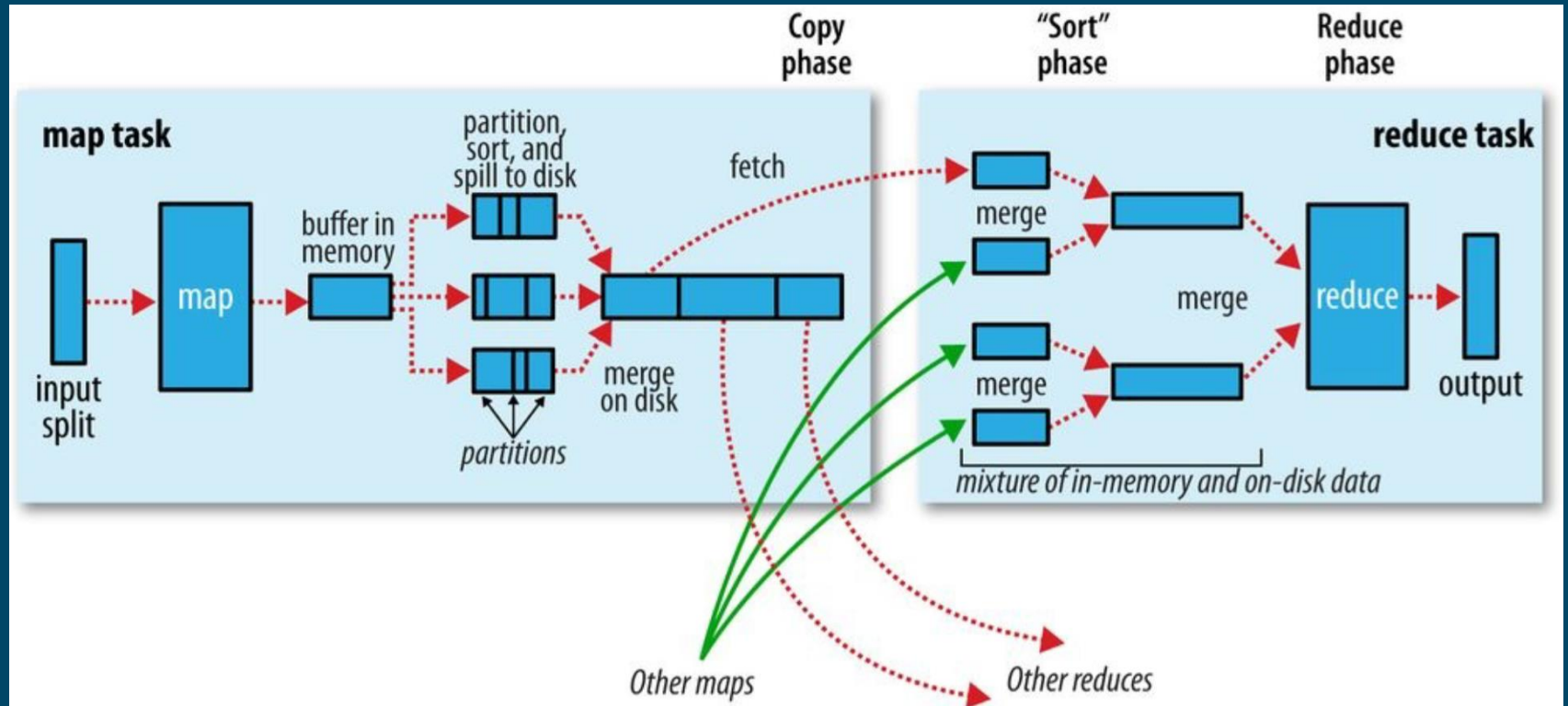
All the values for the input key

After a map task processes the data
and before the reduce task receives the map output
the framework does
Shuffle and Sort
on both the map and reduce nodes

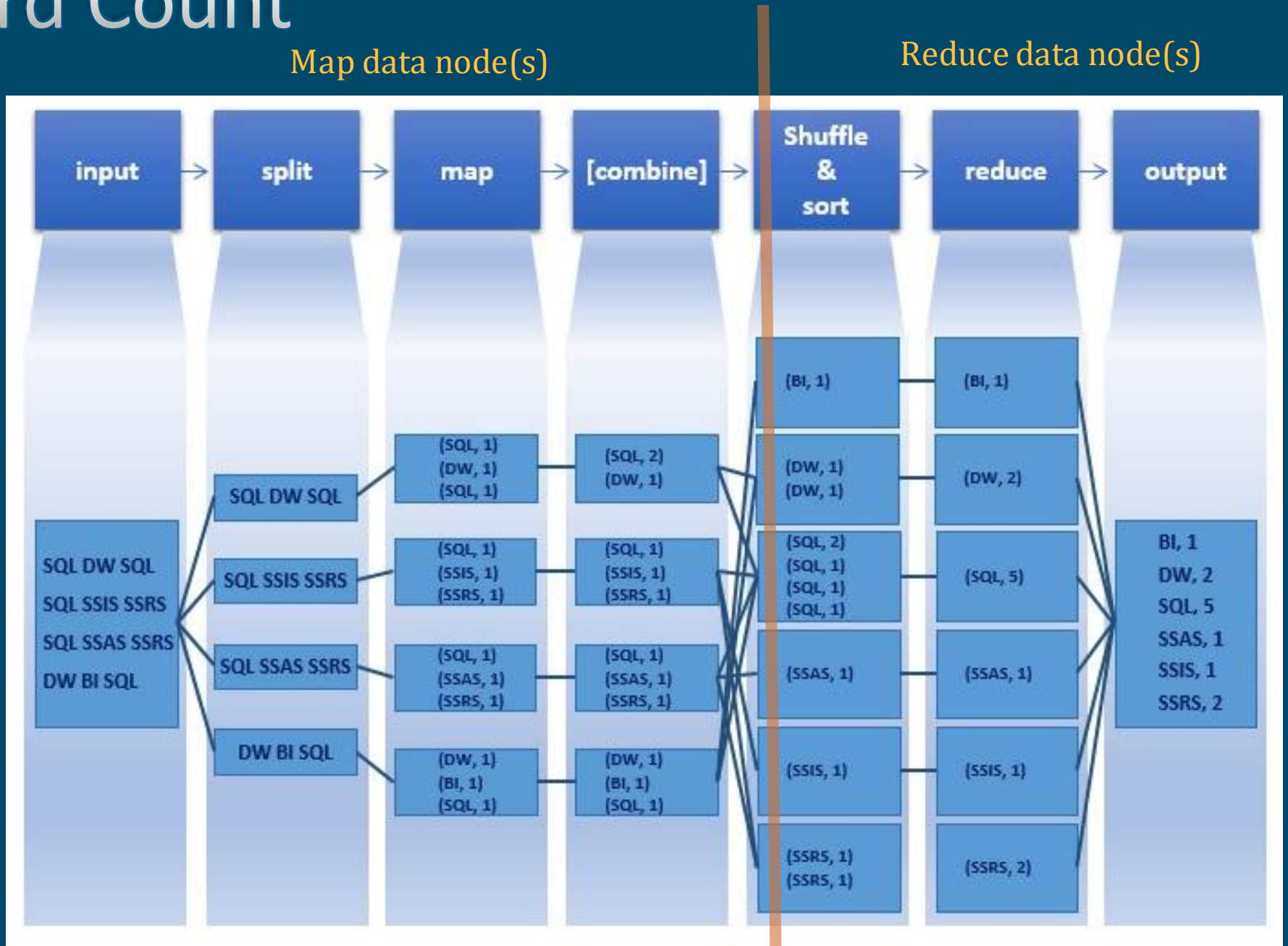
Shuffle is done by framework

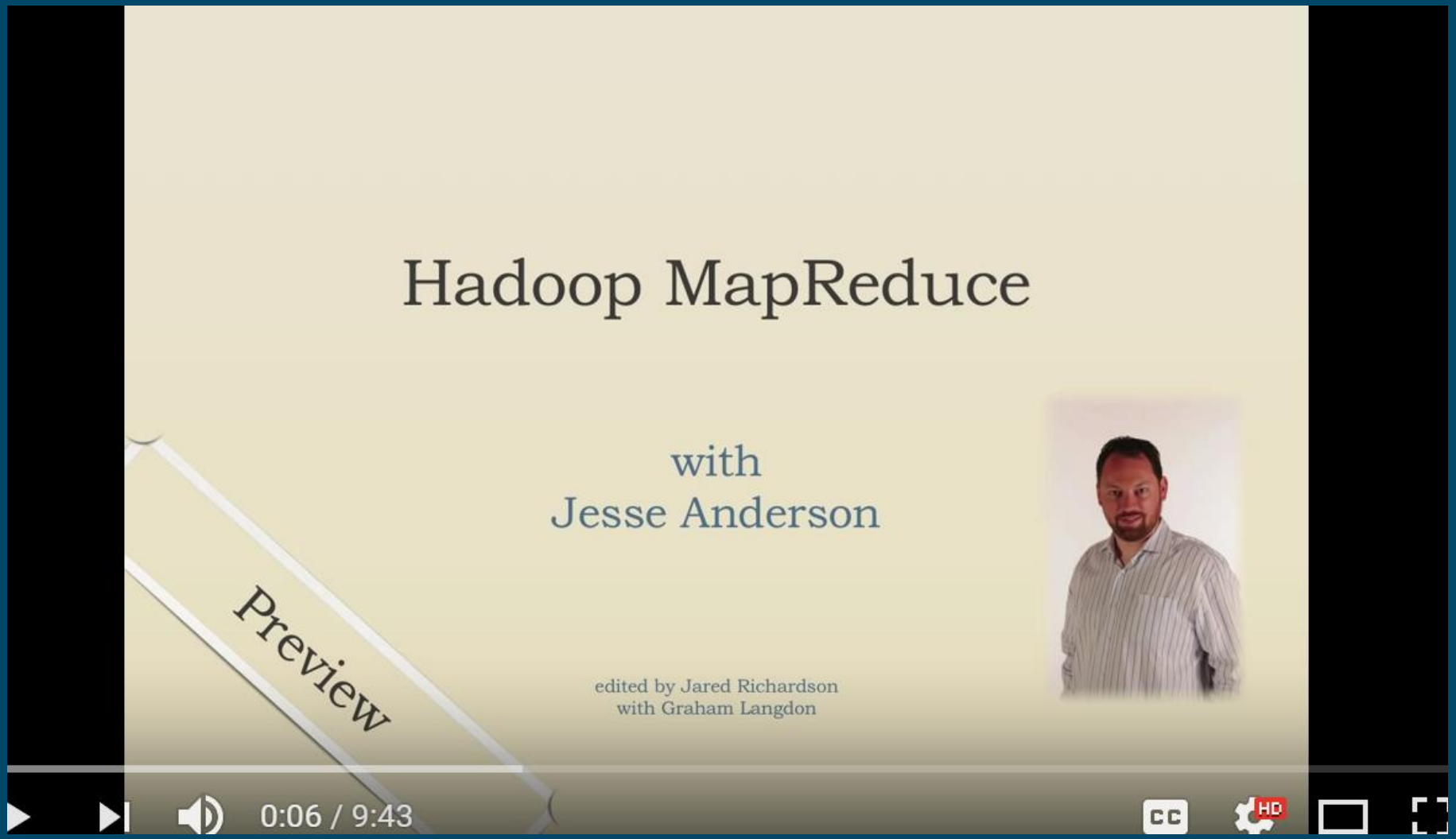
Shuffle: the process of sorting map output & transfer to reducer

1. Sorting occurs on Map node
2. AM directs copy of sorted Map results to Reduce nodes
3. **Reduce node merges the sorted inputs** prior to Map task



Word Count


Map data node(s)
Reduce data node(s)
input
split
map
[combine]
Shuffle & sort
reduce
output
SQL DW SQL
SQL SSIS SSRS
SQL SSAS SSRS
DW BI SQL
SQL DW SQL
SQL SSIS SSRS
SQL SSAS SSRS
DW BI SQL
(SQL, 1) (DW, 1) (SQL, 1)
(SQL, 1) (SSIS, 1) (SSRS, 1)
(SQL, 1) (SSAS, 1) (SSRS, 1)
(DW, 1) (BI, 1) (SQL, 1)
(SQL, 2) (DW, 1)
(SQL, 1) (SSIS, 1) (SSRS, 1)
(SQL, 1) (SSAS, 1) (SSRS, 1)
(DW, 1) (BI, 1) (SQL, 1)
(BI, 1)
(DW, 1) (DW, 1)
(SQL, 2) (SQL, 1) (SQL, 1) (SQL, 1)
(SSAS, 1)
(SSIS, 1)
(SSRS, 1) (SSRS, 1)
(BI, 1)
(DW, 2)
(SQL, 5)
(SSAS, 1)
(SSIS, 1)
(SSRS, 2)
BI, 1
DW, 2
SQL, 5
SSAS, 1
SSIS, 1
SSRS, 2



Key is: 2:20 – 7:30

The MR Framework does Shuffle & Sort work

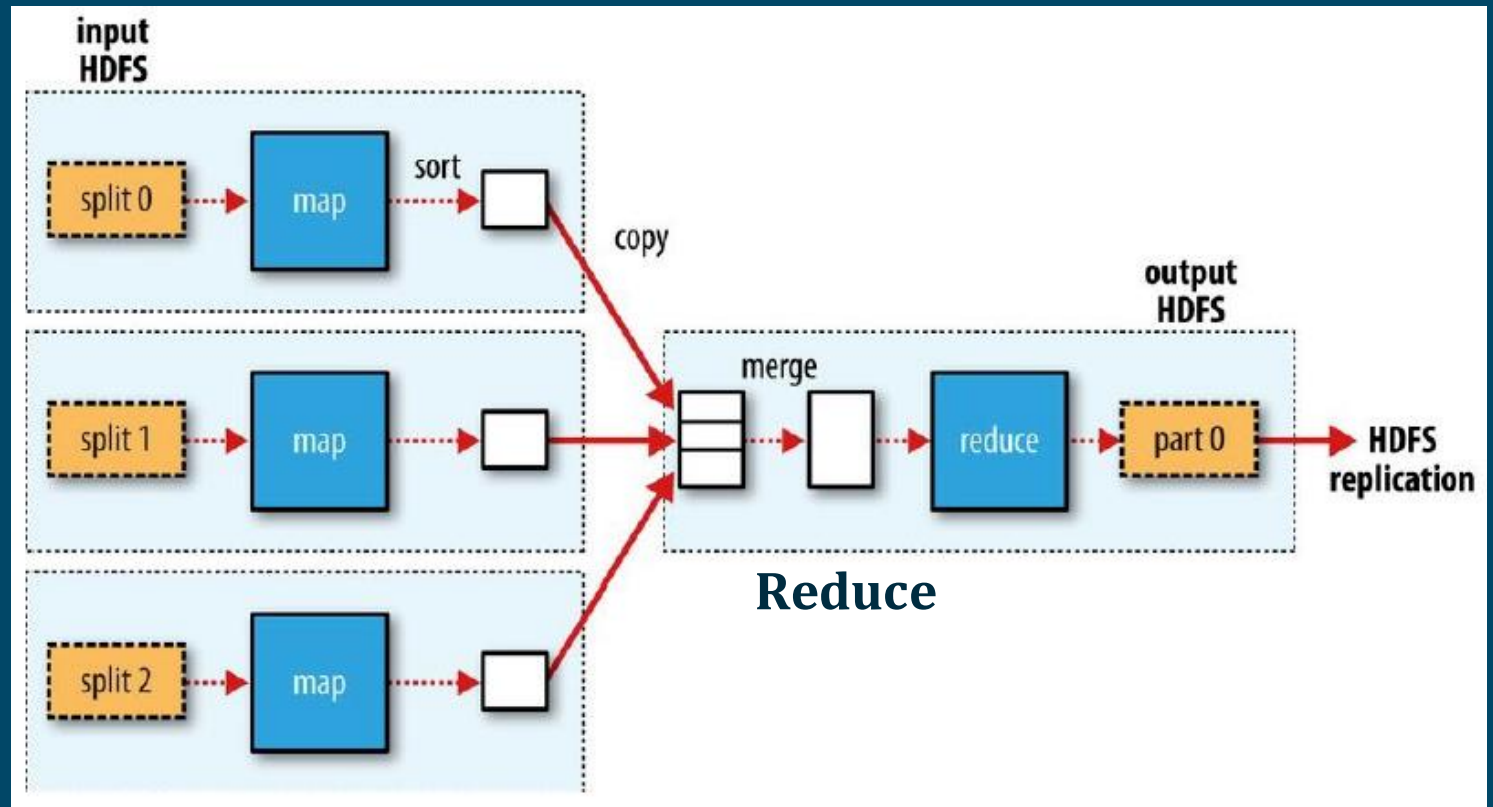
- The framework cleverly manages data
  - Map node output is sorted and “spills” overflow memory buffer to file
  - Reduce node input is merged / sorted in memory (unless needs to “spill” to file)
- Data is copied from Map to Reducer as available
  - Reducer asks AM where to get data
  - AM notifies reducer when data copy is complete
  - As data arrives in Reducer (from multiple nodes), framework merges & sorts it

Map Reduce and its shuffle and sort are one form of distributed computation

Spark includes this and others

MR with data flow within and between nodes

- Solid arrows show data flow across nodes
  - Intermediate results stored to local disk (not HDFS)
    - Tasks restarted in case of failure (task or data)



By default, Hadoop has only **one reducer**

To see results, look in your output directory

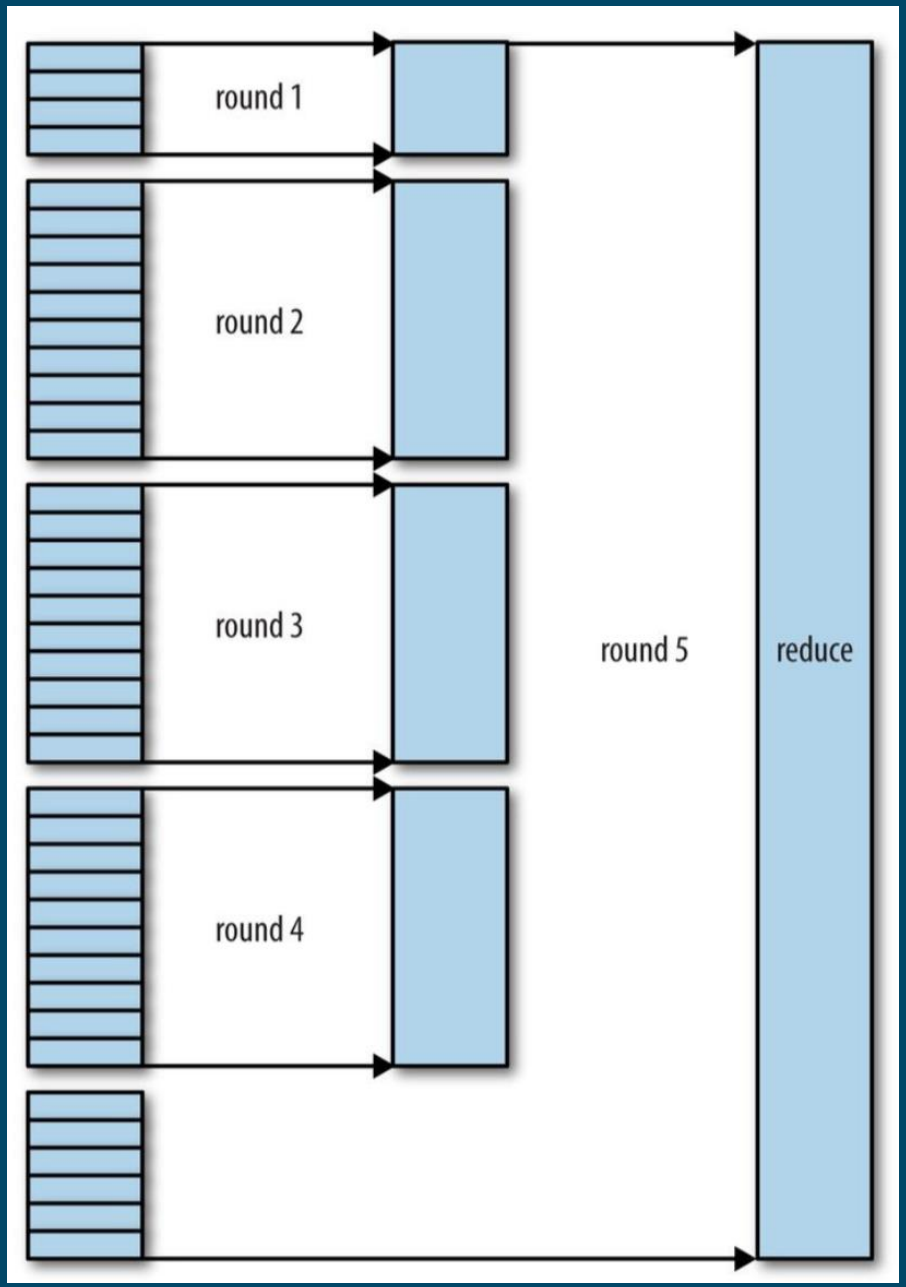
```
hadoop@hadoop-laptop:~$ hadoop fs -cat /Become/Customer_Claims_out/part-r-00000
Customer_4      9999
Customer_4      548
Customer_4      268
hadoop@hadoop-laptop:~$ hadoop fs -cat /Become/Customer_Claims_out/part-r-00001
Customer_1      2500
Customer_1      1200
hadoop@hadoop-laptop:~$ hadoop fs -cat /Become/Customer_Claims_out/part-r-00002
Customer_2      2500
Customer_2      1000
hadoop@hadoop-laptop:~$ hadoop fs -cat /Become/Customer_Claims_out/part-r-00003
Customer_3      1005
Customer_3      650
```

part-….

- The number of output files is equal to the number of reducers if your map/reduce job contains at least one reducer
- For a map-only job, the number of output files is equal to the number of mappers (number of input blocks)

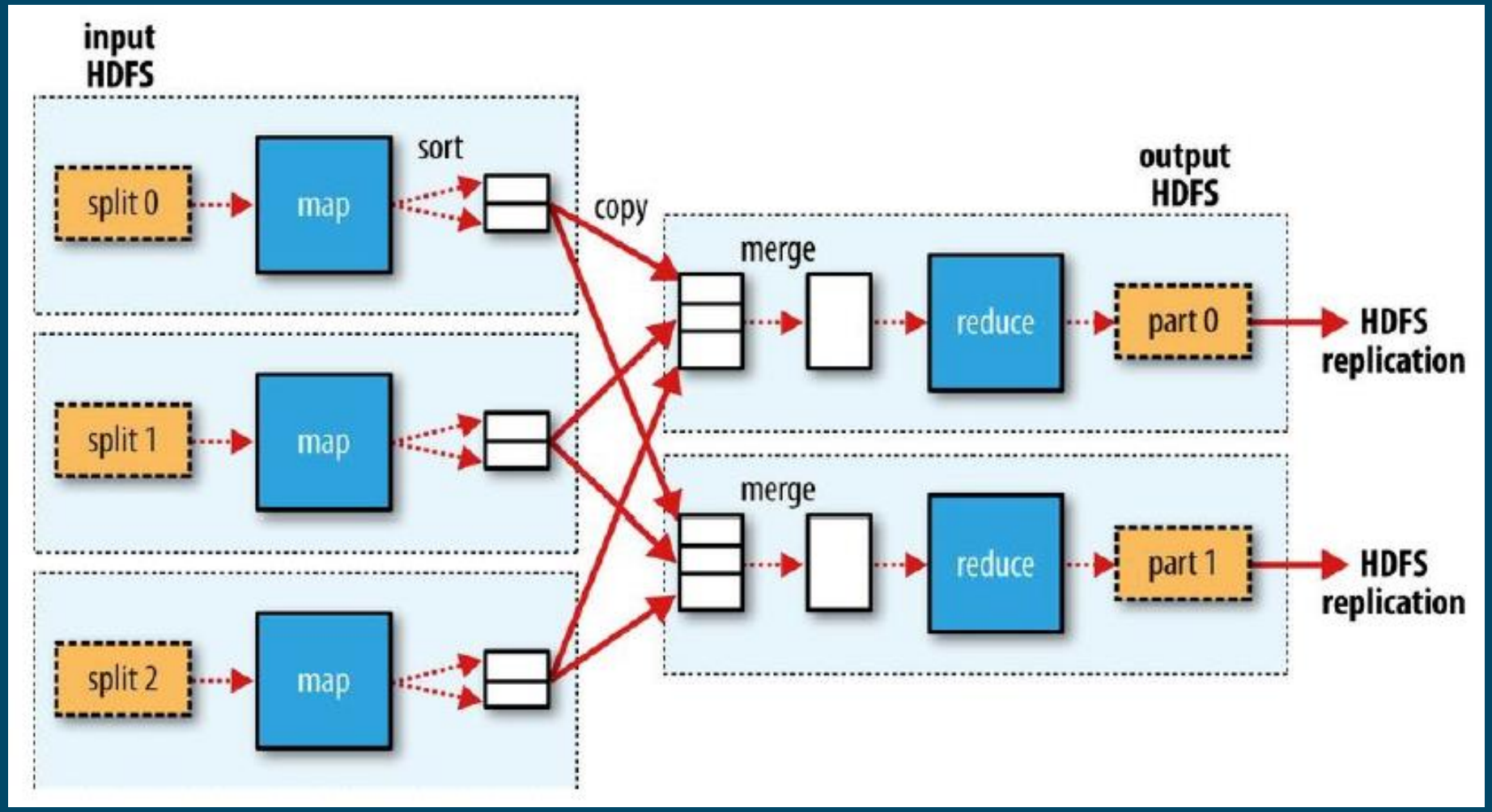
Reduce node merges data

- Map output continuously streams to Reduce as input
- Reduce continuously merges incoming map results
- Incoming (sorted) map outputs (files) are merged in subsets
  - The figure shows 5 subsets for 50 map output files



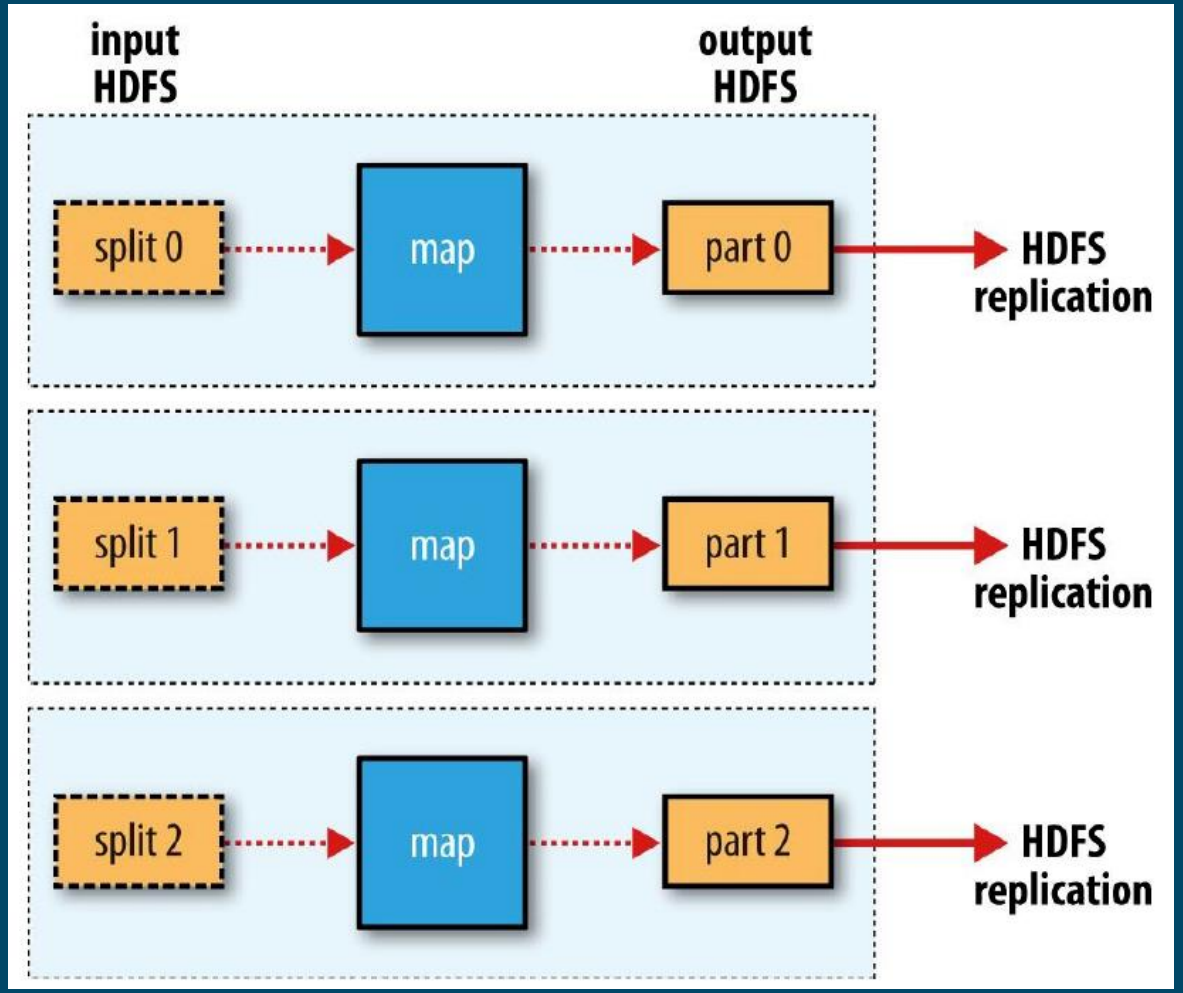
MR with multiple reduce tasks

- Solid arrows show data flow across nodes



MR with no reduce tasks

- Solid arrows show data flow across nodes



Notice that data (as files) are copied from the map
to the reduce task,

which may occur among racks,
that can be slow

How can network traffic (data copies) be reduced,
so as to speed up processing?

Programmer tip: data compression

- Like using Zip to compress files for email
- Use Hadoop's compression API to compressed map output
  - makes it faster to
    - write to disk,
    - saves disk space,
    - and reduces the amount of data to transfer to the reducer.
  - `mapreduce.map.output.compress = true`

Summary of Map Reduce

MapReduce

- A method for distributing computation across multiple nodes
  - Data begins partitioned in the underling HDFS
  - The Map produces sets of data, according to a specified key
    - Data -> (Key, Value)
  - The Maps are sent through the distributed network,
  - where a key and its values are processed (Reduced) by a node, resulting in a value
- Programs declare Maps and Reductions
  - The system takes care of the rest
- MapReduce simplifies distribution data processing

Important to remember

- Executes Map tasks close to the input data
- Sorts Map output before sending to a Reducer
  - Multiple map outputs are merged within the Reducer
    - input is sorted by key
- Framework performs these tasks efficiently, using a mix of memory and disk buffers

Designing MapReduce algorithms

- Key decision: What should be done by map, and what by reduce?
  - map can do something to each individual key-value pair, but it can't look at other key-value pairs
    - Example: Filtering out key-value pairs we don't need
  - map can emit more than one intermediate key-value pair for each incoming key-value pair
    - Example: Incoming data is text, map produces (word,1) for each word in input
  - reduce can aggregate data; it can look at multiple values, as long as map has mapped them to the same (intermediate) key
    - Example: Count the number of words, add up the total cost, ...
- Need to get the intermediate format right!
  - If reduce needs to look at several values together, map must emit them using the same key!