MapReduce

Why MapReduce

- After more than 20 years since its introduction, it still plays a crucial role
  - Powerful paradigm to express and implement parallel algorithms that *do* scale
  - At the core of its evolutions : Pig Latin, Hive, Spark, Giraph, Flink.
  - Companies adopts and maintain a considerable amount of MapReduce code
- MapReduce programming is good for delevopping skills in algorithms and programming.

MapReduce

- It is a **paradigm** to design algorithms for large scale data processing

- Several programming languages can be used to implement MapReduce algorithms (Java, Python, C++, etc.)

- Its main runtime support is Hadoop, which was *initially* designed around MapReduce

Main principles

- **Data model**: data collections are represented in terms of collections of key-value pairs (k, v)

- **Paradigm model**: a MapReduce algorithm (or job) consists of two functions Map and Reduce specified by the developer

- Actually, three phases at least during data processing
  - **Map phase**
  - **Shuffle-and-sort phase, pre-defined**
  - **Reduce phase**

Main principles

- Paradigm model: a MapReduce algorithm (or *job*) consists of two functions, Map and Reduce

  - A MapReduce algorithm takes as input a collection of key-value pairs and returns a collection of the same kind

  - The Map *second-order* function takes a user defined input function $\mathbf{m_{udf}}$, then apply it on each input pair (k,v), for which it returns a, possibly empty, list of pairs [(k1,v1), …. , (kn,vn)]

- The Reduce *second-order* function takes a user defined input function $\mathbf{r_{udf}}$ and applies it to pairs of the form (k', [v1', …. , vn']) and for each such pair returns a list of key-value pairs that takes part of the final result.

- How the Reduce input is produced from the Map output?

Shuffle-and-sort

- the collection M of key-value pairs *resulted* by the Map phase is submitted to a *group-by* process, called *shuffle-and-sort,* and aiming at grouping by key the pairs in M, resulting into a collection of pairs, each one having the form

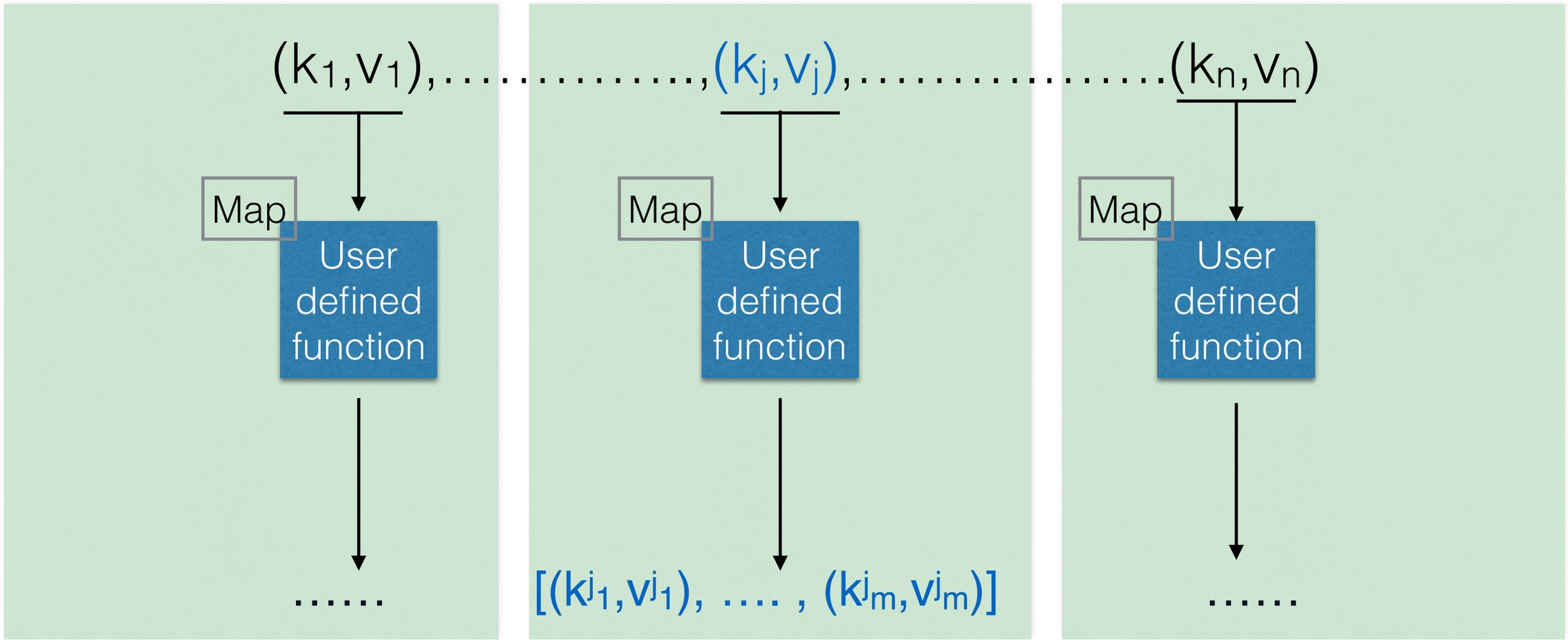
(k, [v1, …. , vn])

- the Reduce phase consists of applying the Reduce user-defined-function $\mathbf{r_{udf}}$ application on each pair of the form (k, [v1, …. , vn]) produced by shuffle-and-sort

- Question: where is parallelism in all the three phases ?

MAP phase

- The Map second-order function is intended to be applied to each input pair (k,v) and for this pair returns a, possibly empty, list of pairs

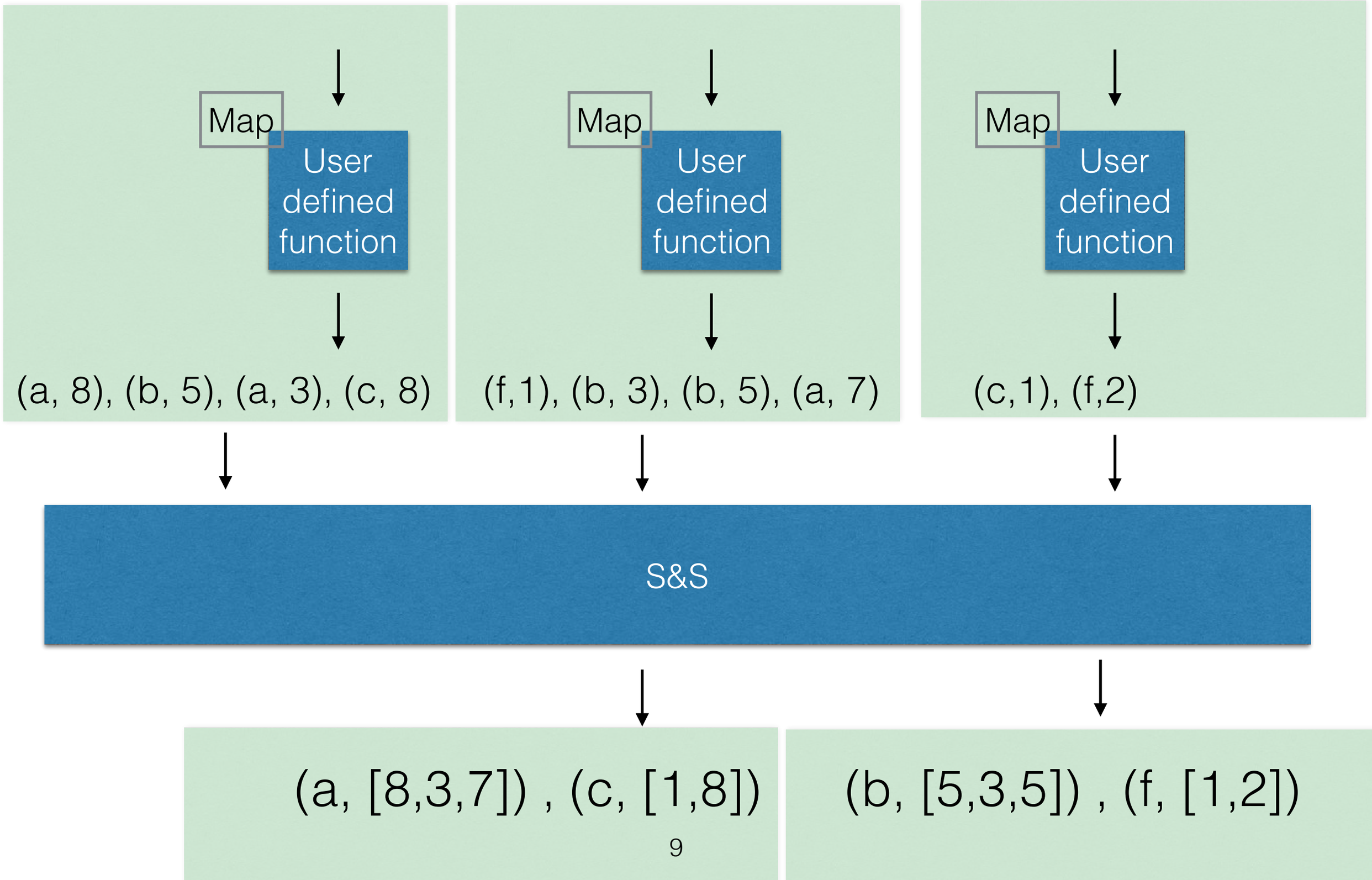


Shuffle and sort

- The shuffle and sort phase **groups** Map outputs on the **key** component producing paris of the form (k’, [v1’, …. , vn’])

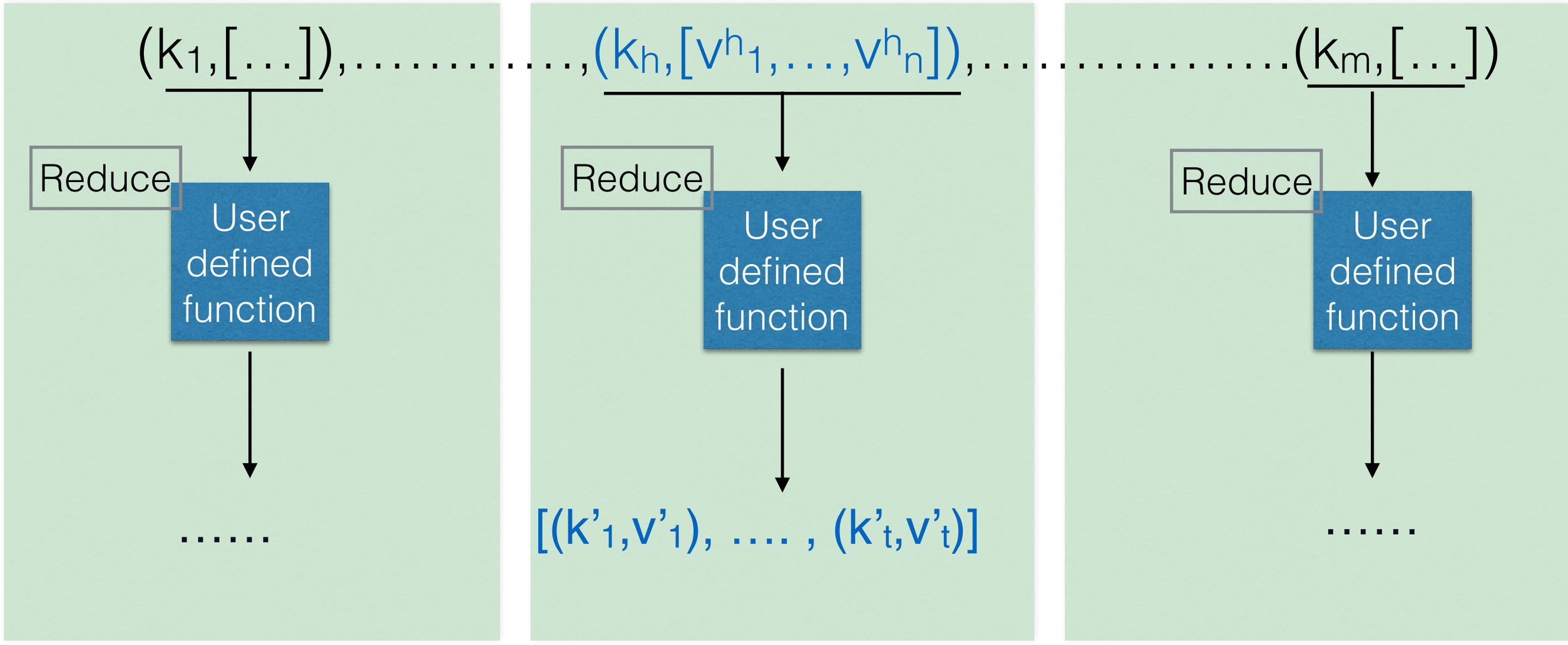
- For instance:

Shuffle and sort



Reduce phase

- applied to each pair (k, [v1, …. , vn]) produced by S&S for which it returns a list of key-value pairs that takes part of the final result



Example: WordCount

- Problem: counting the number of occurrences for each word in a big collection of documents
- Input: a directory containing all the documents
- Pair preparation done by Hadoop: starting from documents, pairs (k,v) where :
  - k is the offset of a text line
  - v is the text line of a document are prepared and passed to the Map phase.

Example: WordCount

- **Map**: takes as input a couple (k,v) returns a pair (w,1) for each word w in v

- **Shuffle&Sort**: groups all of pairs output by Map and produce pairs of the form (w, [1, …,1])

- **Reduce**: takes as input a pair (w, [1, …,1]), sums all the 1's for w obtaining s, and outputs (w,s)

Pseudo code

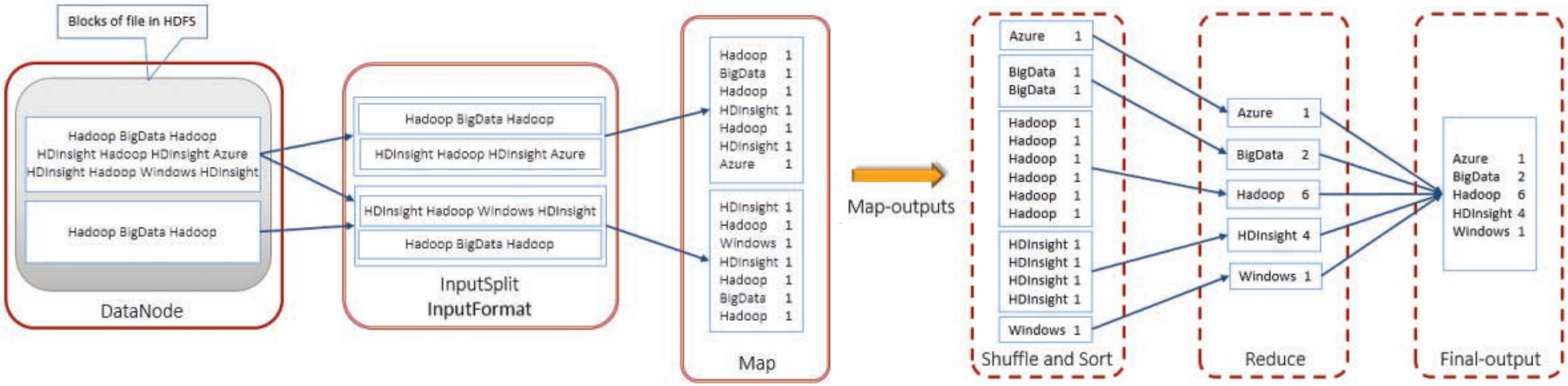
- m(k, v)

```
for each w in v
    emit(w, 1)
```

- r(k, v)

```
c=0
for x in v
    c = c +1
emit(k, c)
```

Example of data flow



Benefits

- Simplicity in parallelism
  - Parallelism is implicit in the model: Map and Reduce functions can be applied in a *shared-nothing* parallelism fashion
  - MapReduce programs can be automatically parallelized over an Hadoop cluster
- Fault tolerance
  - If a slave fails in computing a part of the Map or Reduce phase, the system detects this and re-assign the missing work to another slave in a transparent manner
    - HDFS replication is crucial for this
- Scalability
  - As the data load increases, parallelism level increases by adding slaves to the cluster
- Data locality
  - Hadoop maximize proximity between the slave where data is stored and the slave where processing happens.
  - In the Map phase, most of Mappers are executed on the nodes storing input blocks

We will develop and run MR jobs on Hadoop clusters

- We will use **Python** as programming language.

- But, *only* for today we will focus on algorithmic and programming aspects:
  - So today we will *simulate* on **Linux** a MapReduce job execution

- Assume you have two programs **mapper.py** and **reducer.py** for WordCount (we will see soon them)

- Here is how you can *simulate* MapReduce job execution

```
> cat input.text | ./mapper.py | sort -k1,1 -s | ./reducer.py
```

Based on Linux stdin and stdout

- Both on Hadoop cluster and on Linux simulation

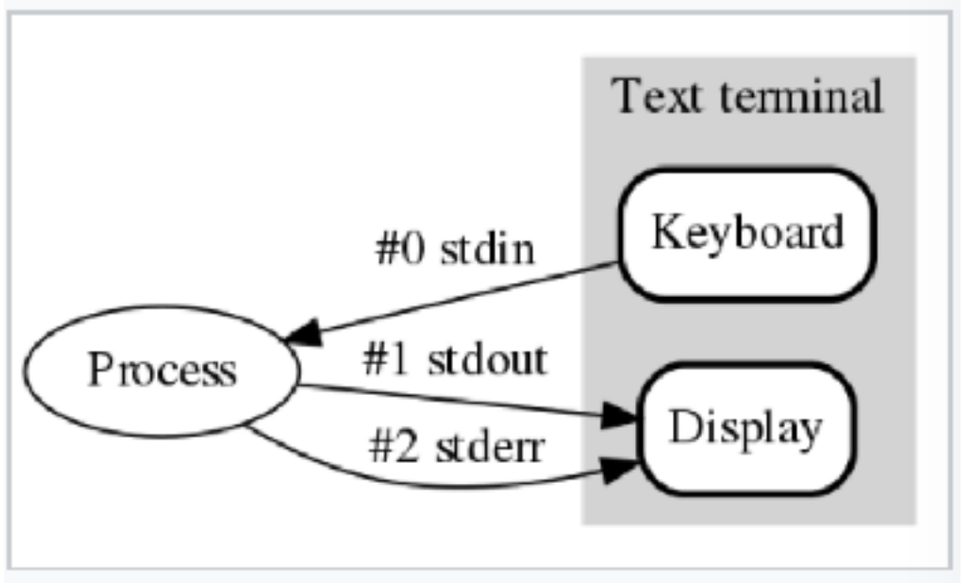
- **stdin** : standard data stream (file) form which a program can read inputs

- **stdout** : standard data stream (file) on which a program can write outputs

- by default data on stdin originates from the *keyboard*

- by default data on stdout are sent to the *display*

- both can be re-directed



Linux pipe(line) |

- Allows for chaining commands together and to compose process execution
- For instance

```
> cat file.txt | grep ‘Hadoop’
```

- In a nutshell
  - the cat command reads all lines of file.txt from **stdin** and write them on **stdout** (by default directed towards the display)
  - The pipe **|** then re-directs the output of **stdout** to the input of **stdin** so that grep can read the file.txt lines produced by cat, and search for the word ‘Hadoop’ in each line
  - grep writes the search results on **stdout**, so you will see them on the screen.

Pseudo code of WordCount

- m(k, v)

```
for each w in v
    emit(w, 1)
```

- r(k, v)

```
c=0
for x in v
    c = c +1
emit(k, c)
```

Word count in Python

- The mapper

```
#!/usr/bin/python

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))
```

The reducer

- **Important:** if Python is used for MapReduce on Hadoop then
  - pairs (k, [v1,..,vn]) are unfolded to a list of pairs (k, v1),…,(k,vn)
  - In practice each pair (k, vi) is a text line in stdin, where k and vi are separated by the tab character \t
- The Reduce algorithm must identify the groups of lines sharing the same k
- Fortunately lines are ordered/grouped on k by shuffle-and-sort.
- Notice that you do not have this unfolding/folding if you use Java for MapReduce

```
#!/usr/bin/python


import sys

previous_word = None
previous_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 previous_word == word:
        previous_count += count
    else:
        if previous_word:
            # write result to STDOUT
            print('%s\t%s' % (previous_word, previous_count))
        previous_count = count
        previous_word = word

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

Scalability issues

- Ideal scaling characteristics
  - Twice the data, twice the running time
  - Twice the resources, half the time
- Difficult to achieve in practice
  - Synchronisation requires time
  - Communication kills performance (network is slow!)
- Thus…minimise inter-node communication
  - Local aggregation can help: reduce size of Map phase output
  - Use of combiners can help in this direction

Combiner

- It is an additional function you are allowed to define and adopt in MapReduce
- Its input has the shape of that of Reduce and its output has to be compatible with that of Map (*)
- Important: a local shuffle-and-sort is performed locally to prepare couples (k, [v1,]
- Like Reduce it performs aggregation
- Differently from Reduce it is run locally, on slaves executing Map
- Goal: pre-aggregate Map output pairs in order to lower number of pairs sent (trough the network) to shuffle-and-sort
- Attention: it is up to Hadoop to decide whether a Mapper node runs a Combiner
  - so some of the Map nodes run the combiner, some do not; this is why we need (*) above
  - we will see in which cases the Combiner is triggered.

Pseudo code with Combine

- m(k, v)
  for each w in v
    emit(w, 1)

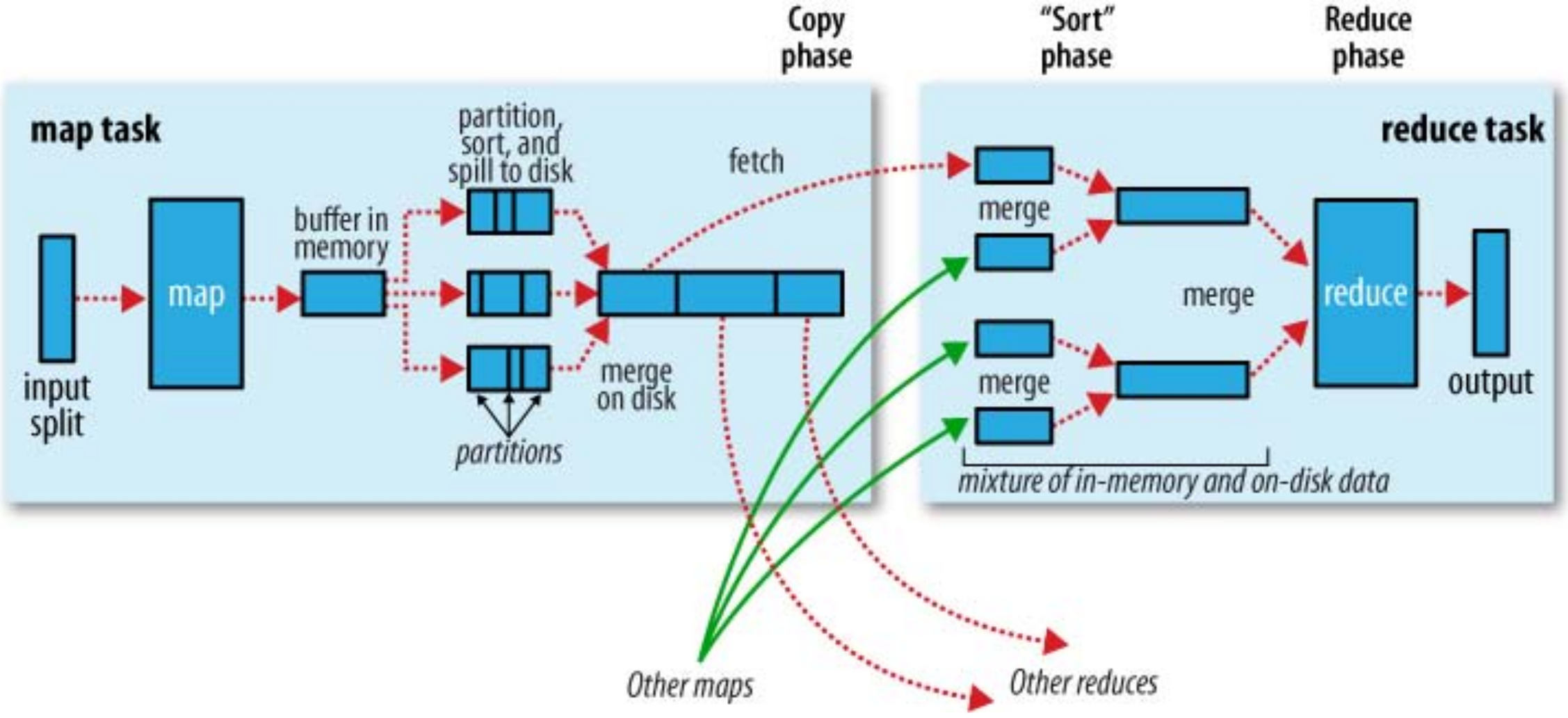
- c(k,v)
  c=0
  for x in v
    c = c +1
  emit(k, c)

- r(k, v)
  c=0
  for x in v
    c = c +x
  emit(k, c)

Note that Combine is isomorphic to Reduce.
This is because the sum operation performed by WordCount is *associative*.

We will see cases next where things are more complex.

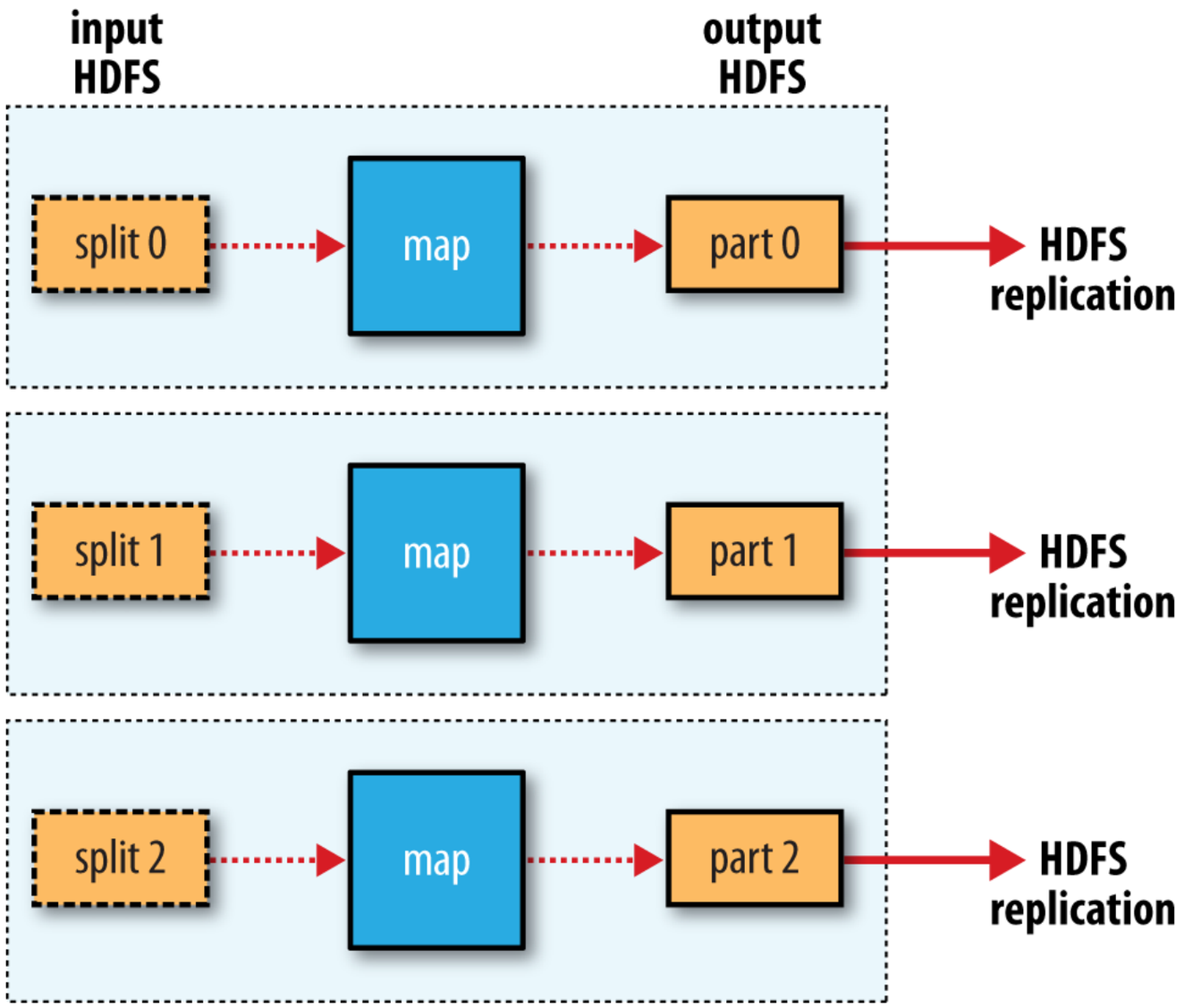
A deeper look at shuffle&sort



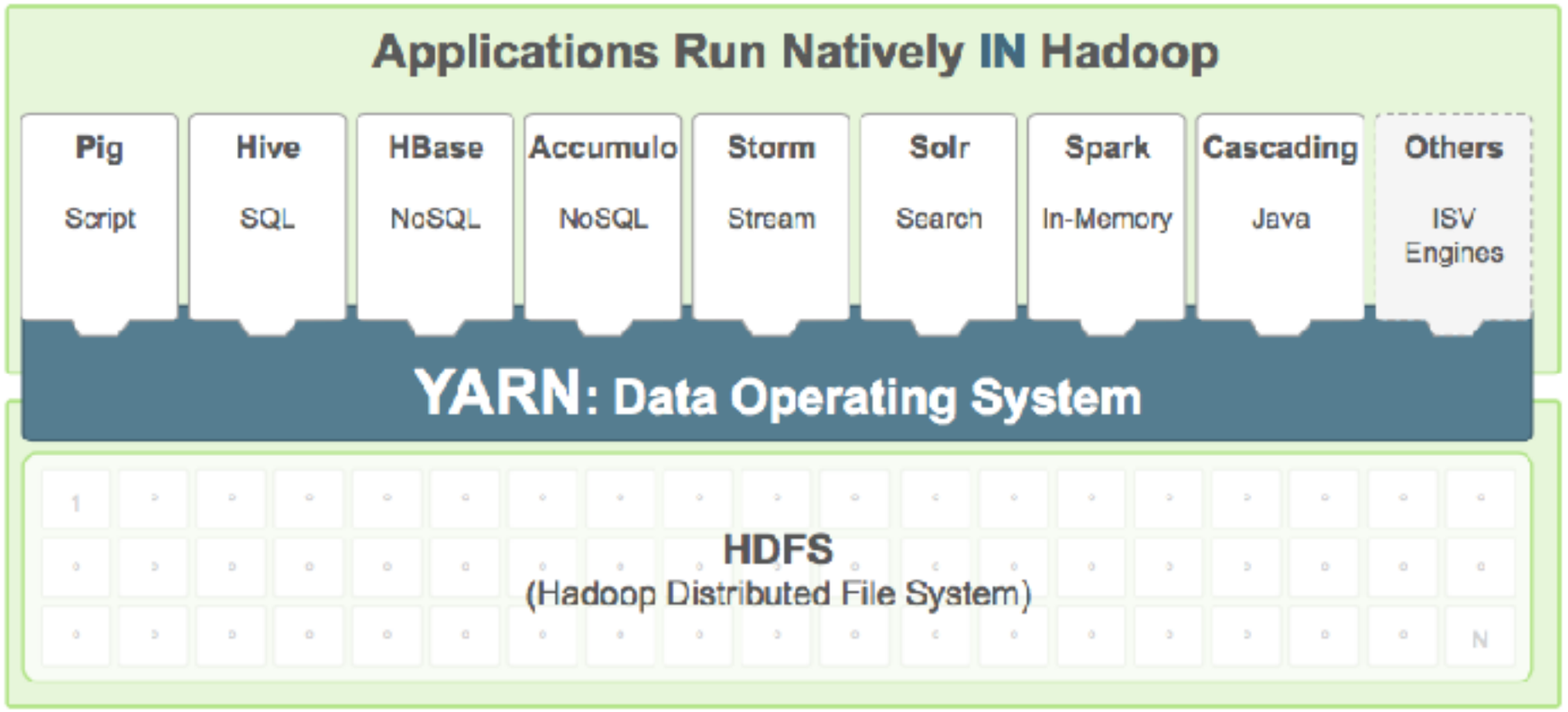
By default, a **spill** file is created each time the buffer memory is 80% full

Combine is triggered if at least three spill files are created.

Dataflow with no reduce



YARN



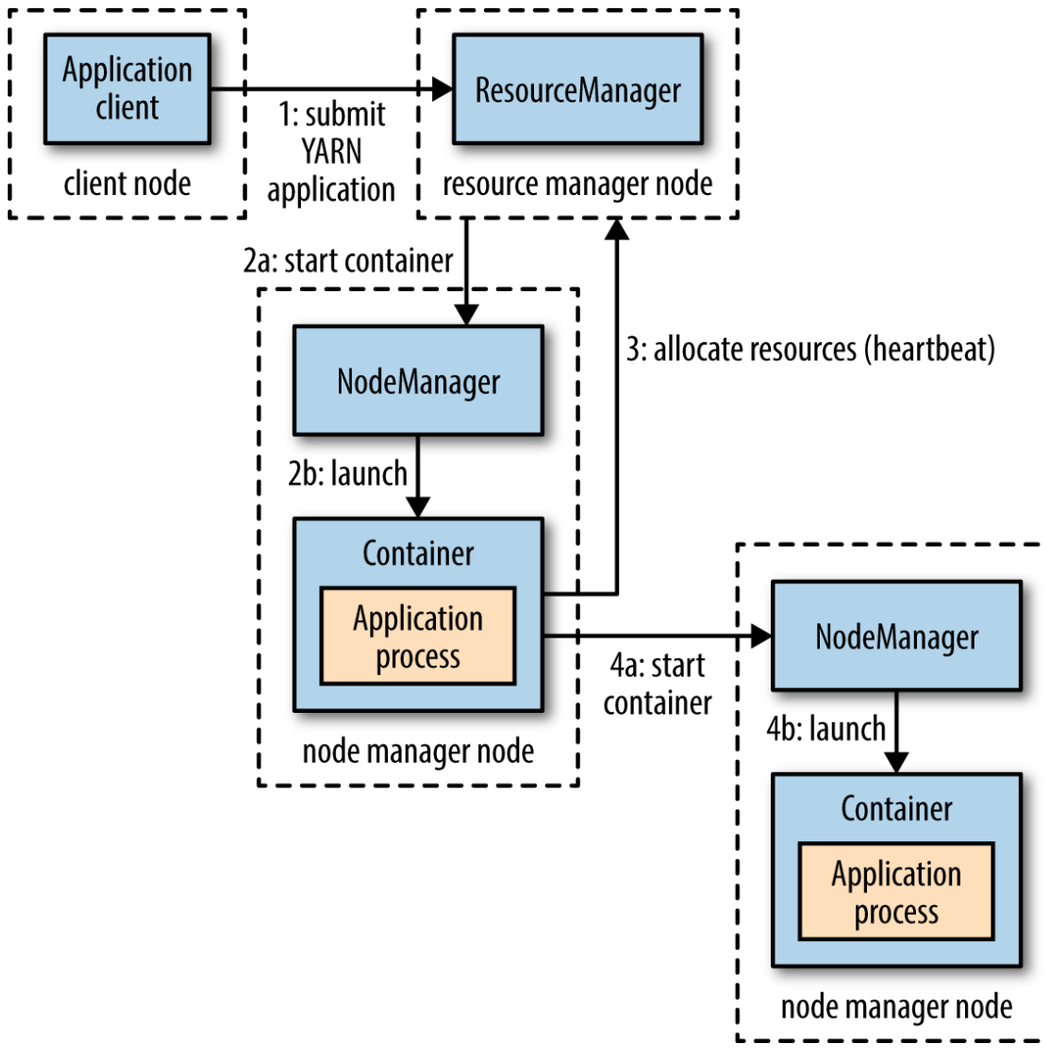
References : *Hadoop: The Definitive Guide* - Tom White.

Apache Hadoop Yarn - Arun C.Murty, Vinod Kumar Vavilapalli, et al.

Big Data Analytics with Microsoft HDInsight - Manpreet Singh, Ashrad Ali.

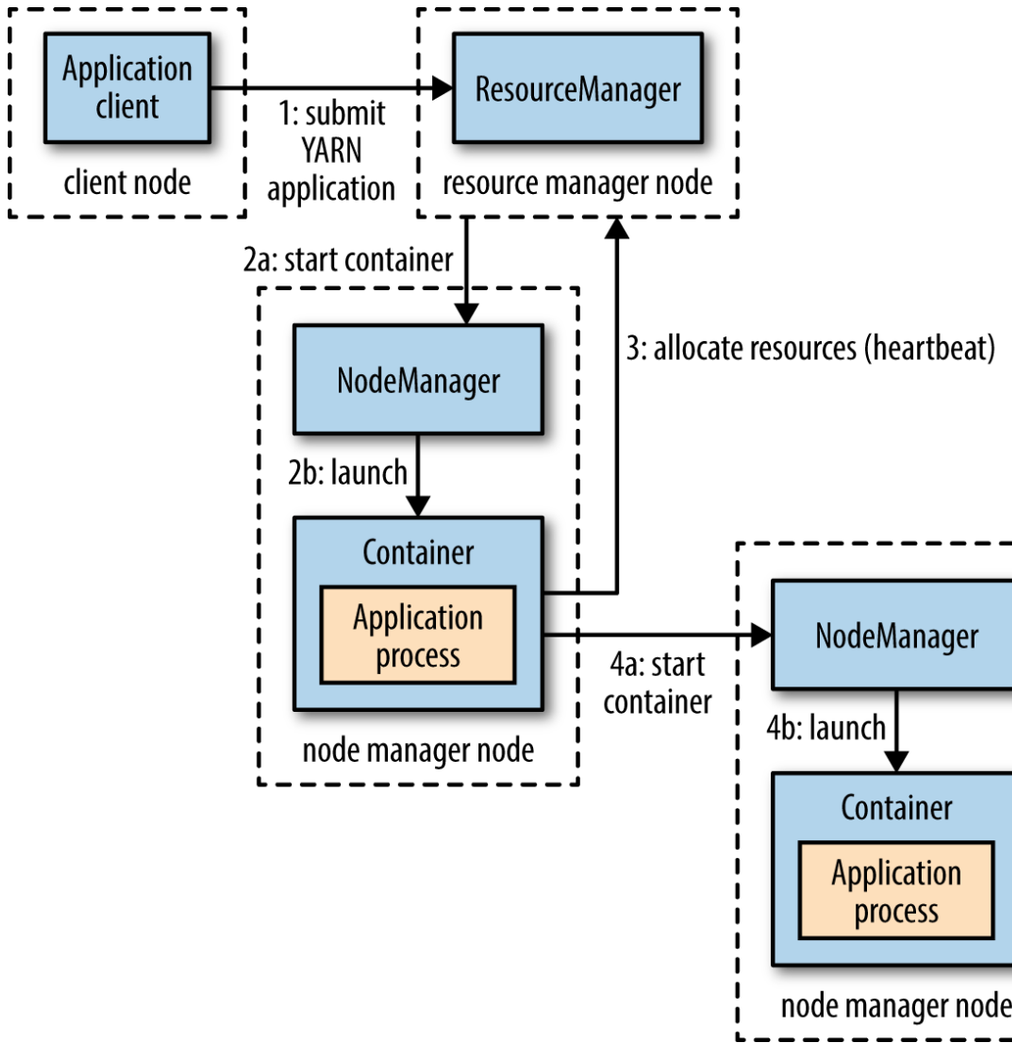
YARN

- YARN is a general purpose **data operating systems**
- It accepts requests of task executions on the cluster and allocate resources for them
- For instance YARN can accept requests for executing MR jobs and MPI (Message Passing Interface) programs on the same cluster
- The set of resources for a task on a given node is called ***container*** and it includes given amounts of memory space and CPU power (cores)
- YARN keeps track of allocated resources in order to schedule container allocation for new requests
- Containers can be demanded all in advance like for MR jobs and Spark tasks, or at run time

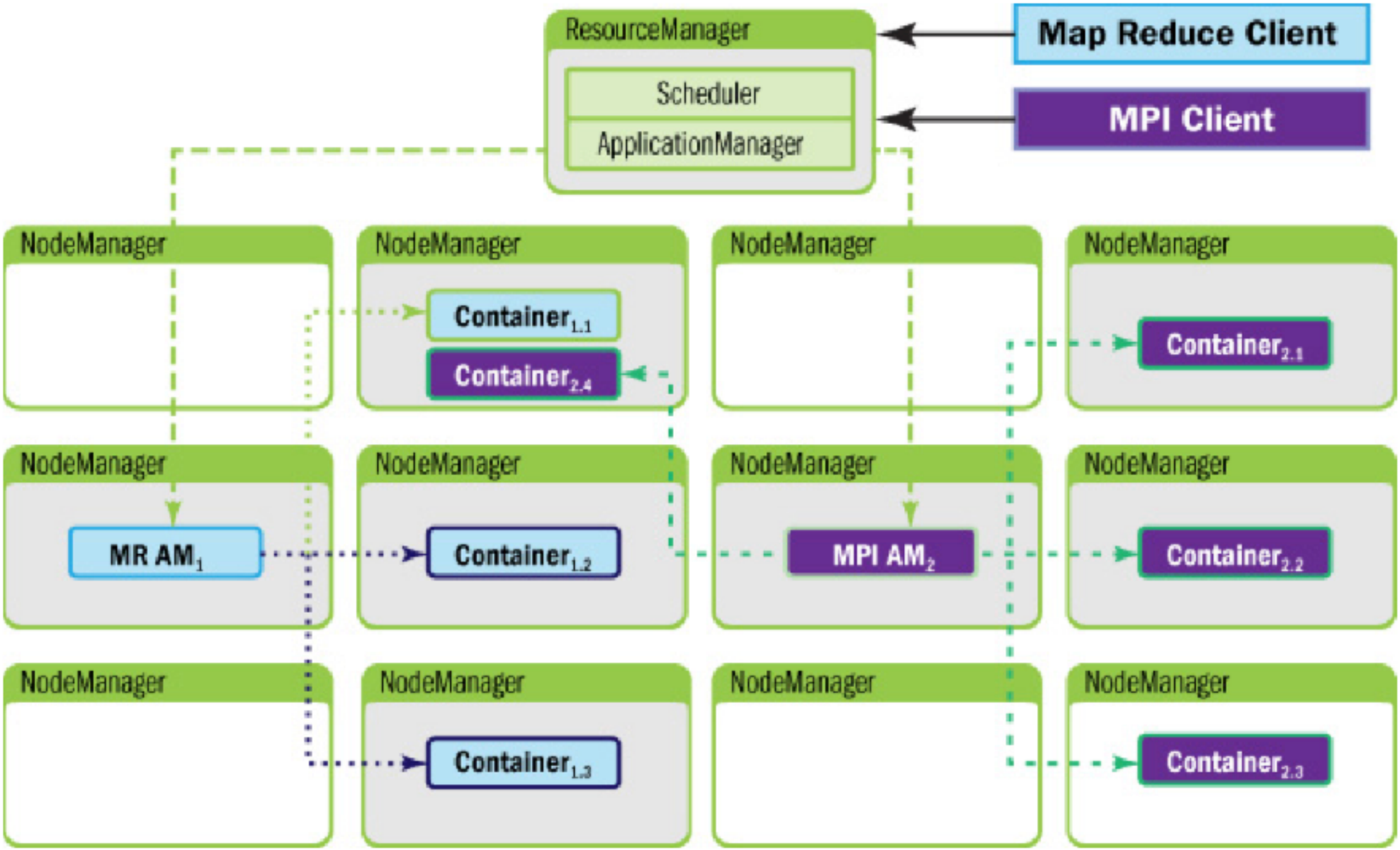


YARN

- Step 1: a client contact the **resource manager** (*running on the master node*)
- Step 2.a: the resource manager then finds a **node manager** (*running on a slave*) that can launch and manage the running operations of the application; this is the **application manager**.
- Step 2.b: the node manager allocates the container indicated by the resource manager. The application runs within the container
- Step 3: eventually, the application manager can request new containers to the resource manager
- Step 4: a parallel container is then started after the acknowledge of the resource manager. Started container informs the application manager about their status upon request.
- Containers can be demanded all in advance or at run time (step 4)
- At the end of the process, the application manager informs the resource manager, which will kill the allocated containers.



YARN as a data OS



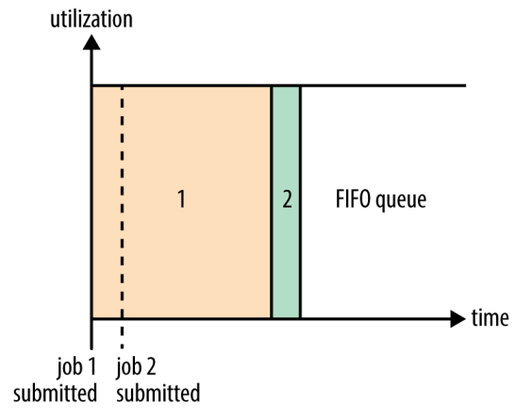
YARN scheduling policies

The kind of scheduling policy has to be set in the yarn-default.xml configuration file

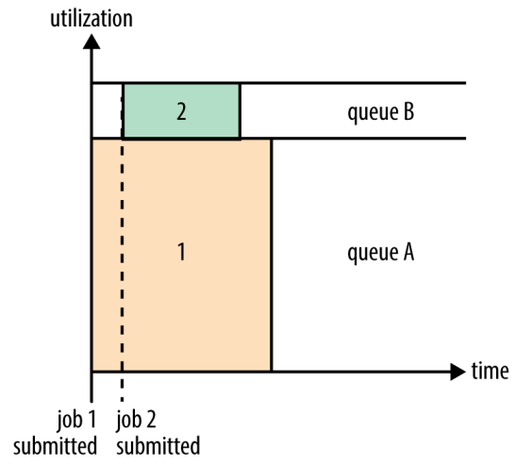
- FIFO, for clusters with ‘small’ workloads
- Capacity Scheduler (introduced by Yahoo! for **large clusters**)
    - the administrator configures several queues with a predetermined fraction of the total resources (this ensures minimal resources for each queue)
    - each queue is FIFO, and has strict Access Control Policies to determine what user can submit to what queue
    - configuration can be changed dynamically, when feedback from the Node managers is available, more starved queues can have more resources
    - work best when there is knowledge about the kind of workload, this allows for setting minimal resources for queues
- Fair Scheduler
    - each application is bound to a queue
    - YARN containers are given with priority to queues consuming less resources
    - within a queue the application having the fewest resources is assigned the asked container
    - this can imply resource reduction for running jobs
    - attention: in the presence of a single application, that application can ask for the entire cluster.

YARN scheduling policies

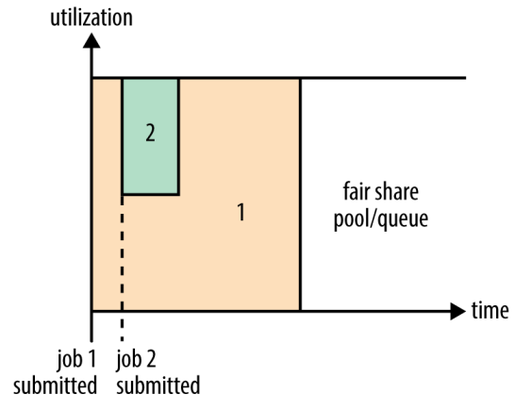
i. FIFO Scheduler



ii. Capacity Scheduler

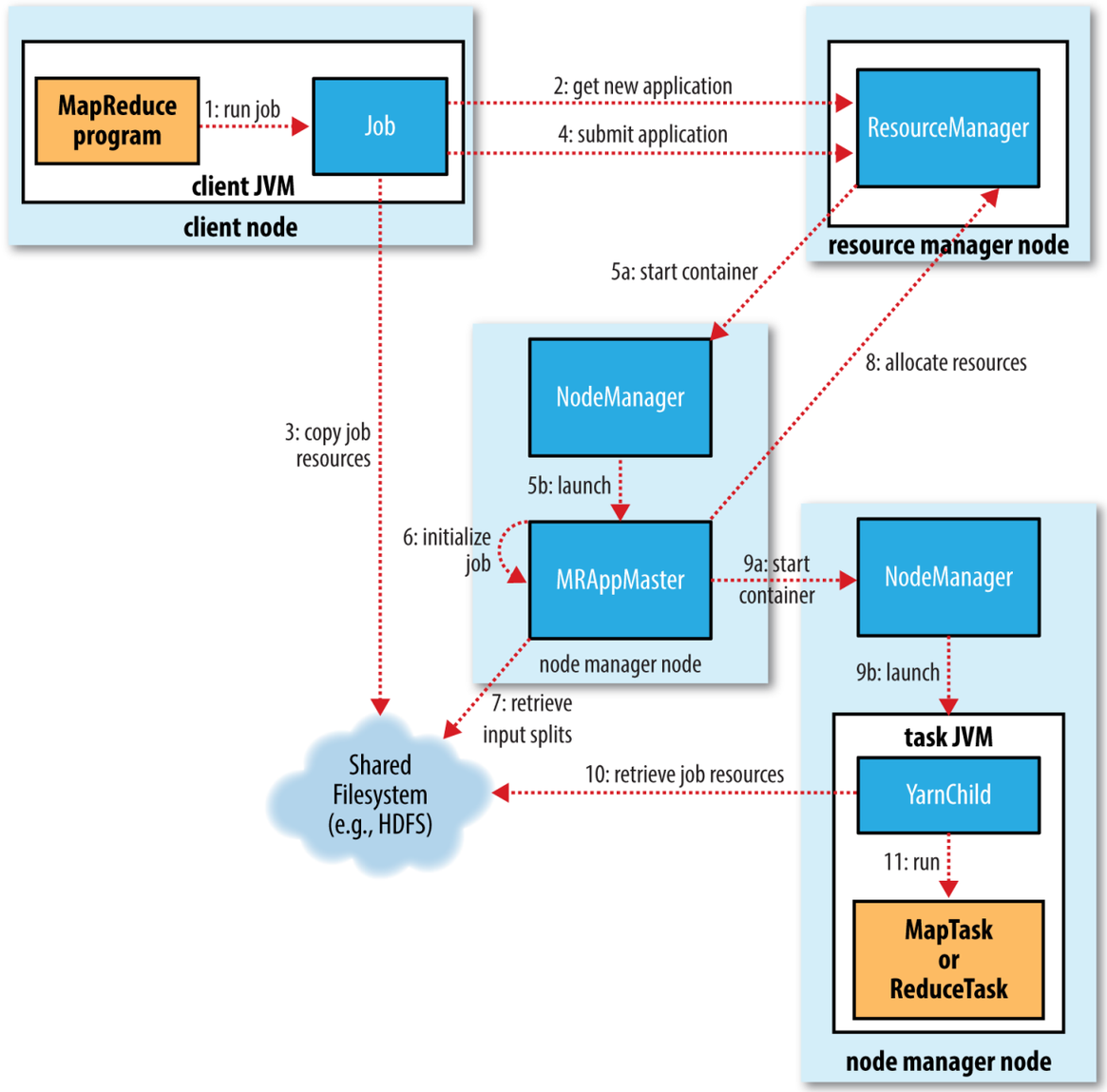


iii. Fair Scheduler



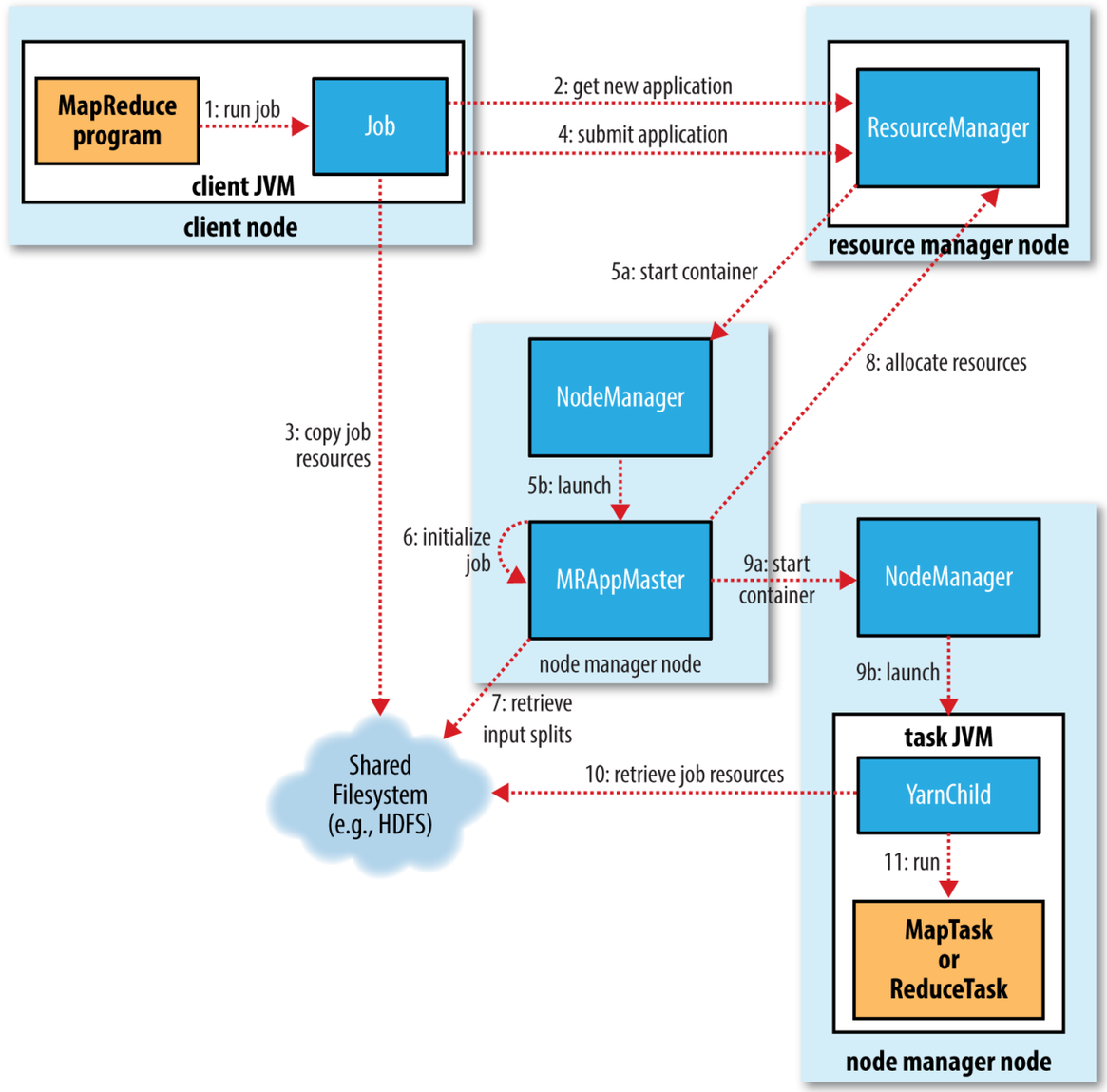
MapReduce on YARN

- As already said, MapReduce job execution relies on YARN
- In a first phase (steps 1-3) the client node asks the RM for a job ID and then sends all the information needed to start the JOB (….)
- The RM then starts a container (steps 5-6) for the the MR Application manager, which initialise the jobs and decide whether to run the job locally (**uber execution**, input size < size of a block) or on the cluster in parallel.
- In case of distributed execution, the Application Manager (MA) asks for containers for Map and Reduce tasks (a container for each task)
- Only Map tasks container are allocated keeping into account locality when possible
- The number of running Map tasks depends on the number of input splits; which in turns depends on configuration; it can be set by the user
- The number of Reduce tasks can be set by the user, and depends from configuration. By default, only one reduce task is run.



MapReduce on YARN

- Task execution is made within a dedicated Java Virtual Machine.
- If task succeeds it commits and inform the AM
- According to configuration, the same task can be run in parallel in order to augment robustness and efficiency (**speculative task execution**)
- Only one of speculative tasks commits, the fastest one, the other ones abort.
- The job succeeds if all the to be executed tasks succeed.
- Several statistics about execution are made available by Counters, displayed at running time.

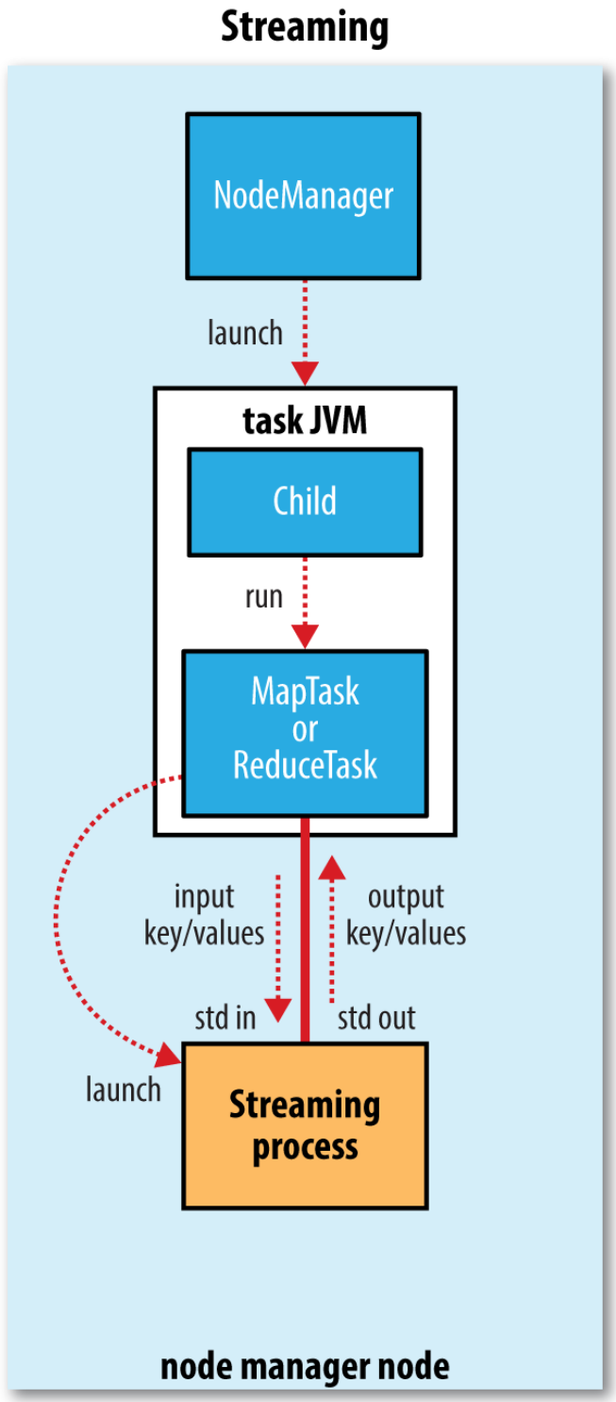


Hadoop streaming

- Although Hadoop is a Java-based system it can support execution of MapReduce jobs written in other languages.

- This is particularly important for us, as we are going to implement MapReduce jobs in Python.

- To this end we will rely on Hadoop streaming.



- In a nutshell, the task JVM runs all the auxiliary operations (split and record reading) output writing, etc.

- The Map/Reduce algorithms are executed on the node-manager and can read records (pairs) on the Linux/Unix **standard input stream (stdin)** and write records **(pairs)** on the **standard output stream (stdout).**

- For instance, the Map task running on the JVM reads records and put them on the stdin stream so that the Map program can read and process them.

- The Map program emits pairs by performing simple print operations, that put pairs on the stdout stream, that are in turn read by the Map task and sent to the Shuffle&Sort phase.

- So streaming implies an overhead, that is negligible for complex (time consuming) tasks