COPERNICUS MASTER IN DIGITAL EARTH

HPC FOR BIG DATA

4. MapReduce

Frédéric RAIMBAULT, Nicolas COURTY

University of South Brittany, France

IRISA laboratory, OBELIX team

Outlines

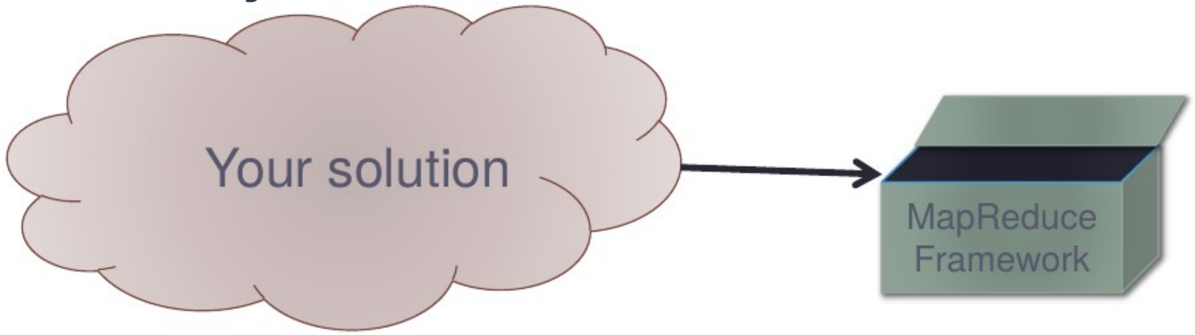
1. MR Programming Model
2. Hadoop MR Runtime Support
3. Mrjob package

MapReduce Programming Model

- **Published in 2004 by two Google scientists**
- **Used for**
  - Processing and generation of huge datasets on clusters
  - Certain kinds of distributed problems
- **Composed of**
  - *map* tasks that process key/value input pairs to generate a set of intermediate key/value pairs
  - *reduce* tasks that process all intermediate values associated with the same intermediate keys
- **Apache Hadoop MapReduce is an open-source Java implementation of the MR programming model**

Pros and Cons

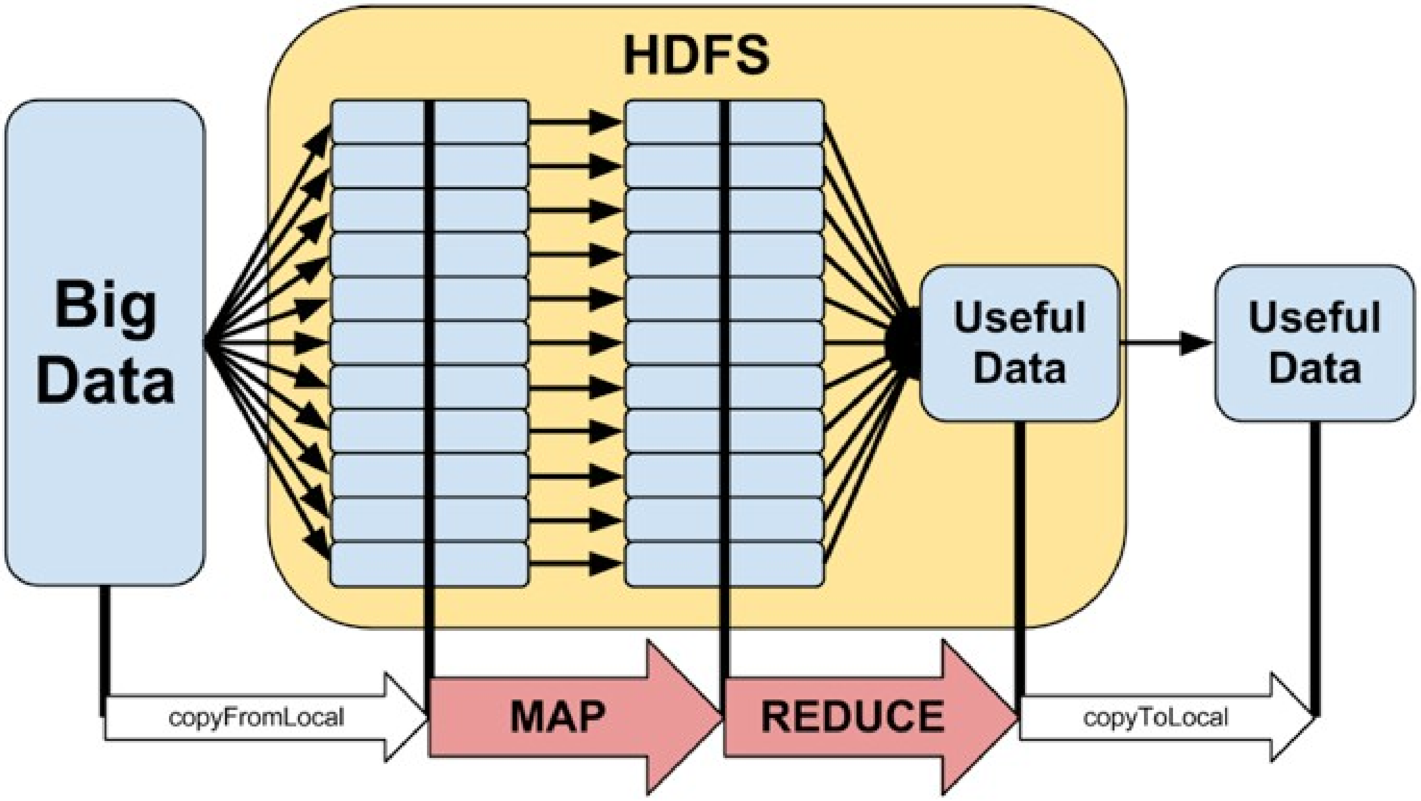
- **As a programming model it constrains you:**



- **In return it provides:**
  - Transparent parallelism
    - Concurrently run many tasks (map and reduce) on big data across a cluster
  - Efficiency
    - process the data where it is stored, based on HDFS blocks location
  - Fault tolerance
    - Thanks to HDFS blocks replication
  - Schema on read
    - Data do not have to have been stored conforming to rigid schema

MapReduce High Level View

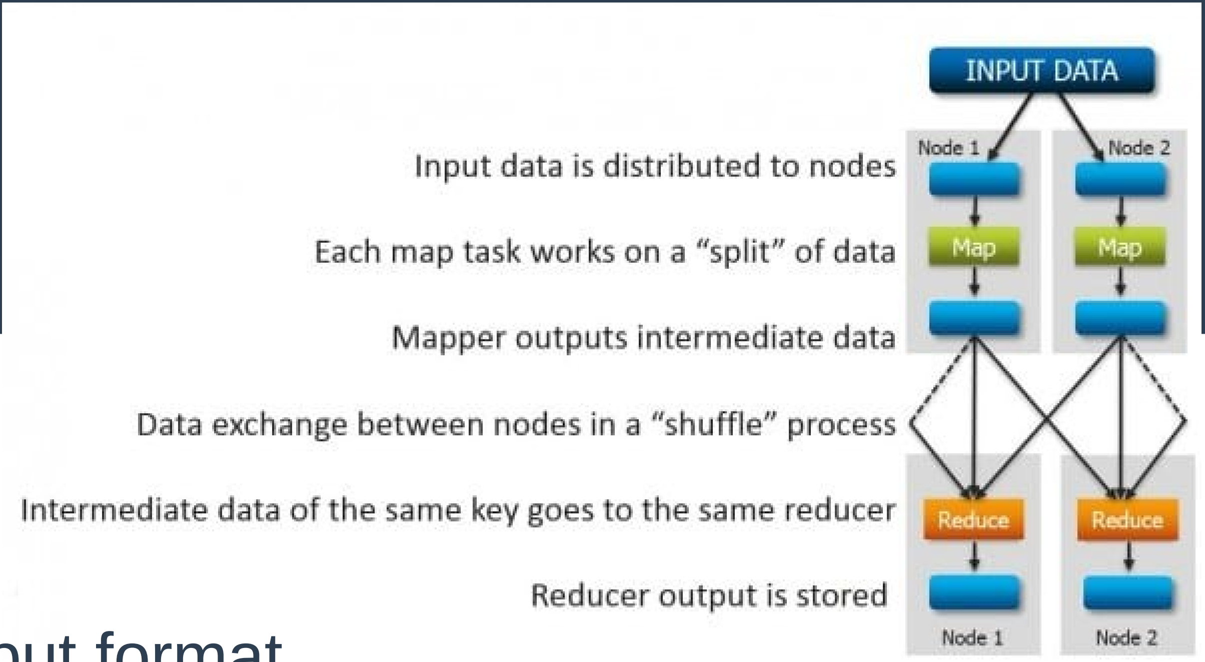
- MR processes data from HDFS and produces data into HDFS
- Work is divided into independent tasks (map and reduce operations) and assigned to node for local processing



Key Concept: Key/Value Pairs

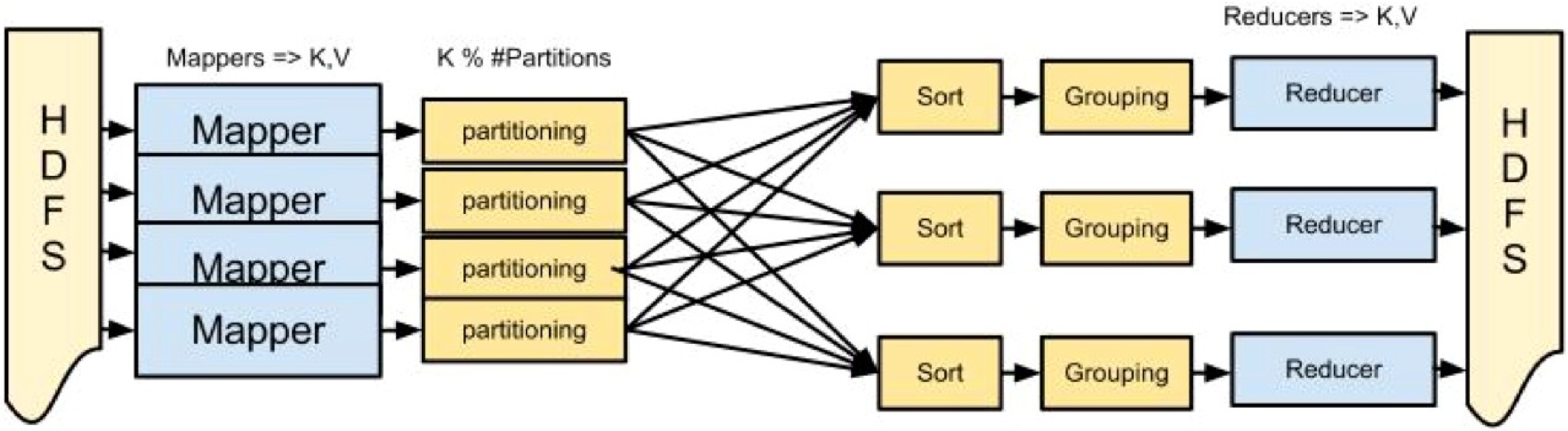
- **HDFS data is stored in such a way that the various values in the data set can be sorted and rearranged across a set of keys**
  - Keys relate to associated values
    - A key could have no associated value
- **Examples:**
  - Address book: name → contact informations
  - Bank account: account number → account details
  - Book index: word → pages on which it occurs
  - Computer file system: filename → data
- **Allows for a powerful programming model widely applicable**
  - MapReduce as a series of key/value transformations

MapReduce Operations


INPUT DATA
Node 1
Node 2
Input data is distributed to nodes
Each map task works on a "split" of data
Map
Map
Mapper outputs intermediate data
Data exchange between nodes in a "shuffle" process
Intermediate data of the same key goes to the same reducer
Reduce
Reduce
Reducer output is stored
Node 1
Node 2

- **Map: $(K_1,V_1) \rightarrow [(K_2,V_2)]$**
  - Reads input data such as a key-value pair according to a chosen input format
  - Writes intermediate data as a list of key-value pairs
- **Reduce: $(K_2,[V_2]) \rightarrow [(K_3,V_3)]$**
  - Processes all intermediate data having the same key
  - Writes output data as a list of key-value pairs according to a chosen output format
- **In-between : Partition & Shuffle & Sort**
  - Based on the key of pairs produced by mappers
  - Managed by the MR runtime

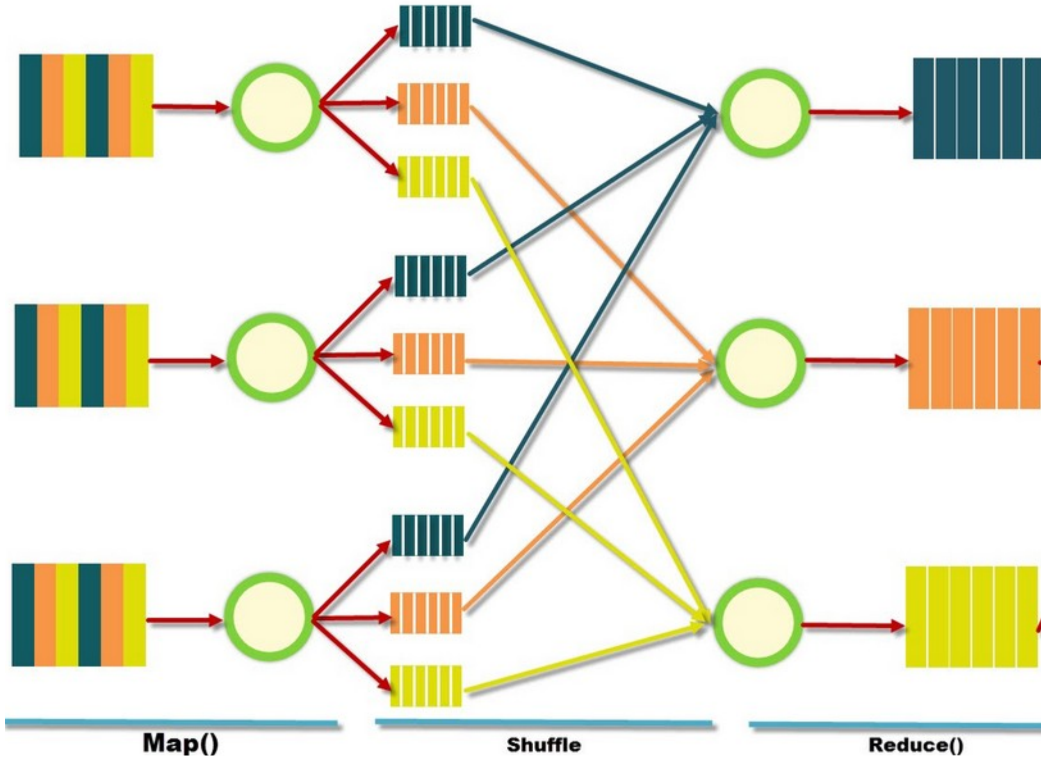
Partition & Shuffle & Sort

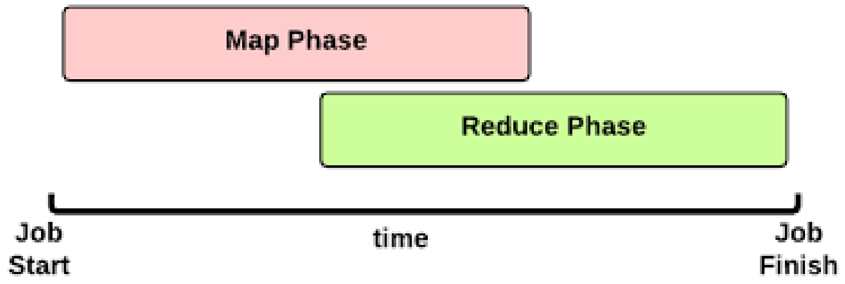


- **Outputs of map tasks are grouped by key and distributed to reducers in the shuffle phase**
- **Reduce tasks receive, sort and process all values having the same key**
- **The MR runtime takes care of the partitioning, the shuffling and the sorting operations**

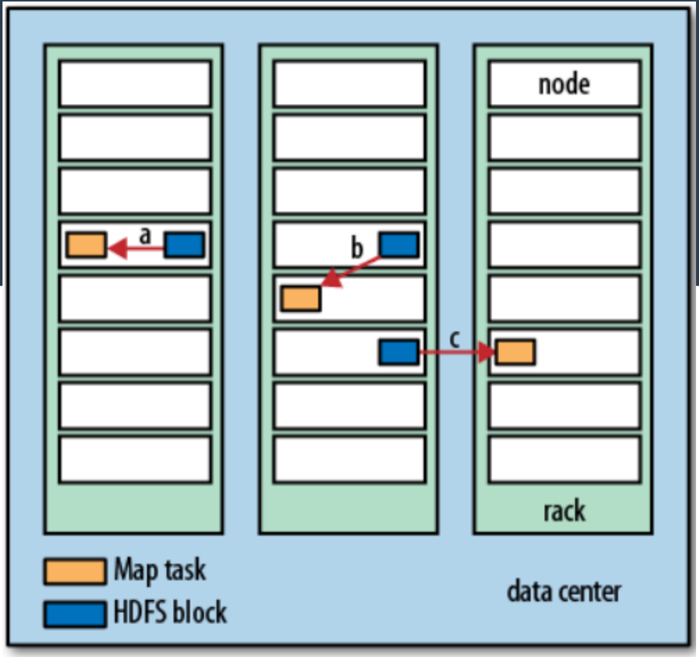
MapReduce Scheduling

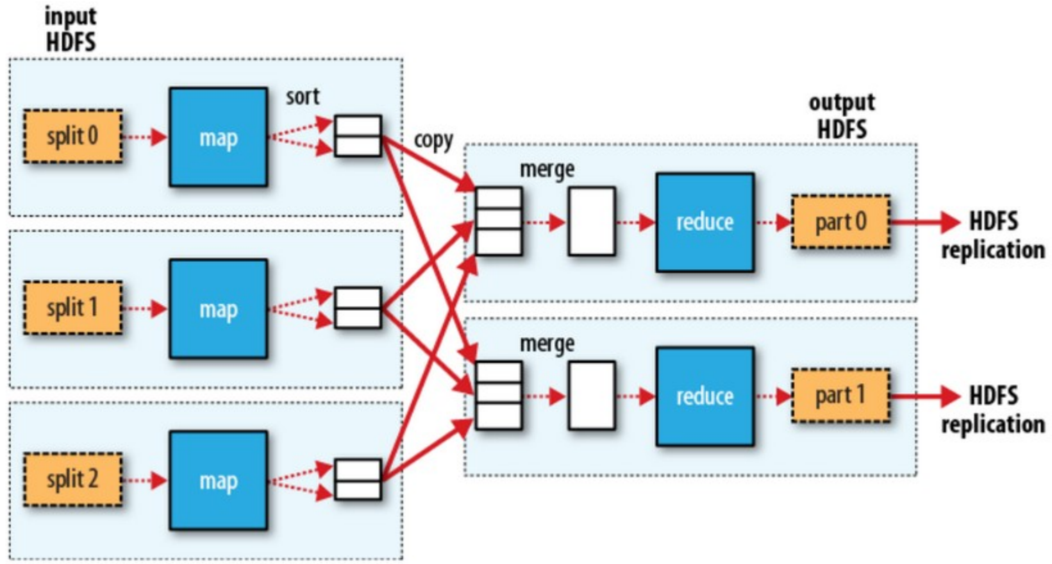
- **Map tasks are run in parallel by mappers**
  - Repeatedly apply the `map()` function to each of its inputs
- **Reduce tasks are run in parallel by reducers**
  - Repeatedly apply the `reduce()` function to each of its inputs
- **Map and Reduce phases are run sequentially**
  - Partitioning, shuffling and sorting occur during the end of map and the beginning of reduce phase

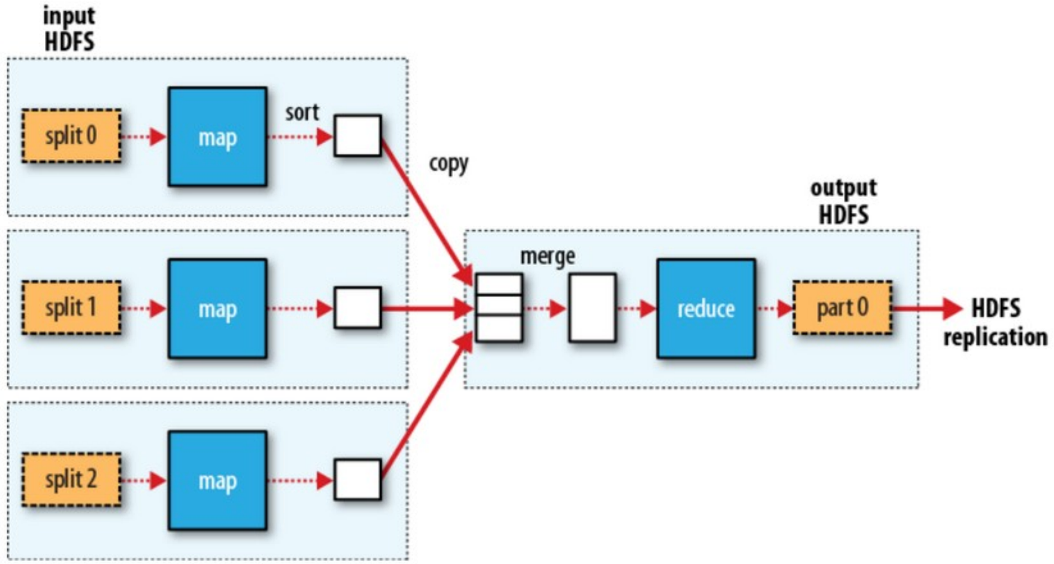


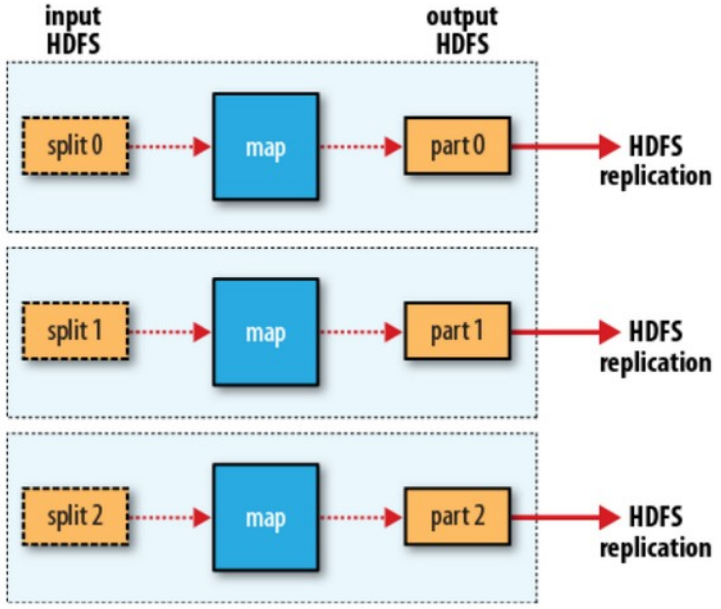


Scaling Out


node
a
b
c
rack
Map task
HDFS block
data center

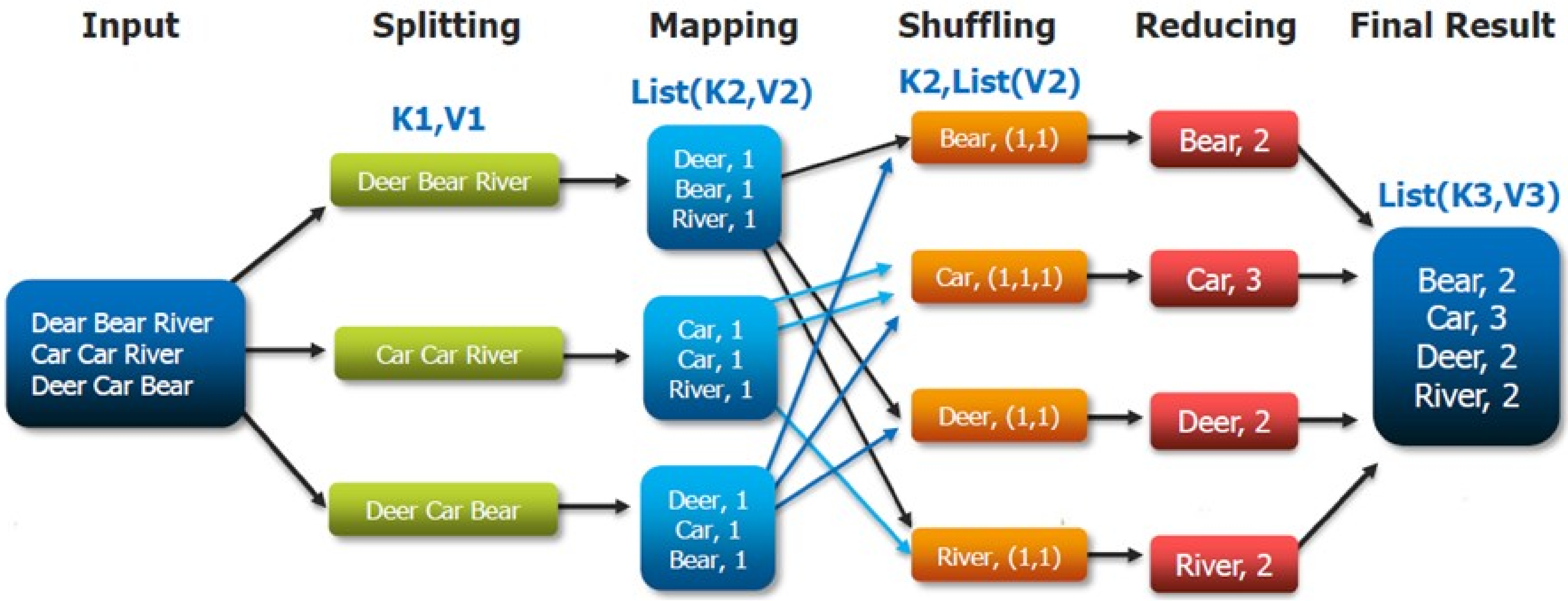

input HDFS
split 0
split 1
split 2
map
sort
copy
merge
reduce
part 0
part 1
output HDFS
HDFS replication


input HDFS
split 0
split 1
split 2
map
sort
copy
merge
reduce
part 0
output HDFS
HDFS replication


input HDFS
output HDFS
split 0
split 1
split 2
map
part 0
part 1
part 2
HDFS replication

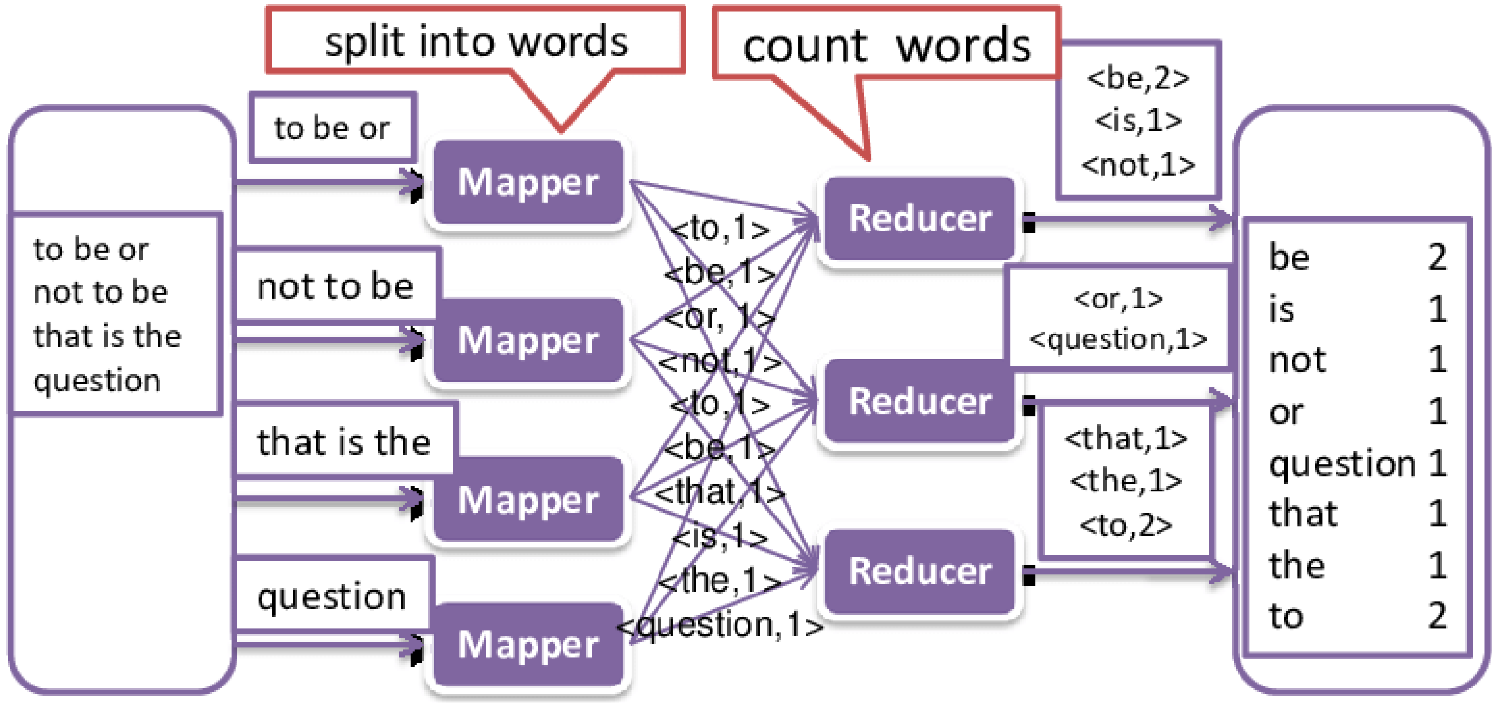
Example: WordCount (1)

- **Counts the occurrence of words in documents**
  - Documents are lines of words
  - (Key,Value) is (offset,line of words)



Example: WordCount (2)

- map: (offset, line) → [word, 1]
- reduce: (word, [1,…,1]) → (word, n)

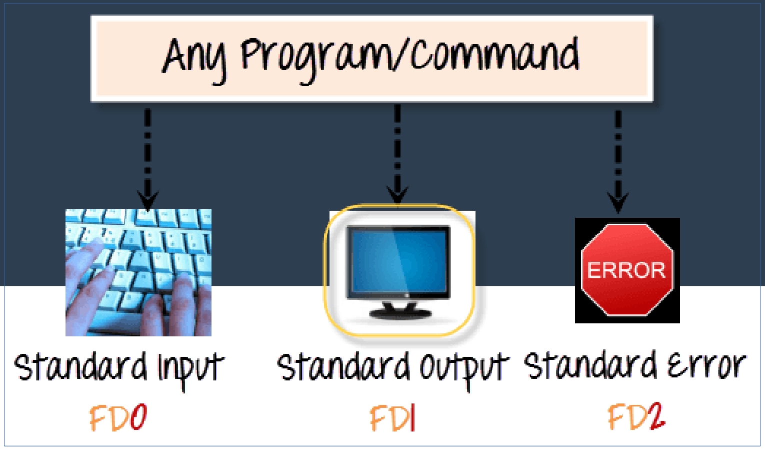


MapReduce takes care of:
splitting input, sorting and distributing intermediate results, merging output.

MR Programming Languages

- **Hadoop is natively written in Java and primarily designed to work with Java code.**
  - The user provides the mapper and the reducer code as Java programs
  - The user chooses or customize an input /output formatter, a key partitioner and a key comparator.
- **Native Java**
  - Maximum flexibility
  - Best performance
  - Most difficult to write
  - Tedious and lengthy
- **Supports others languages via Hadoop Streaming**

Key Concept: Standard Unix Streams



- **Any Unix-based process has**
  - a standard input stream, *stdin*, default to keyboard
  - a standard output stream, *stdout*, default to screen
  - a standard error stream, *stderr*, default to screen
- **Any standard stream can be chosen (redirected) on the command line or programmatically**

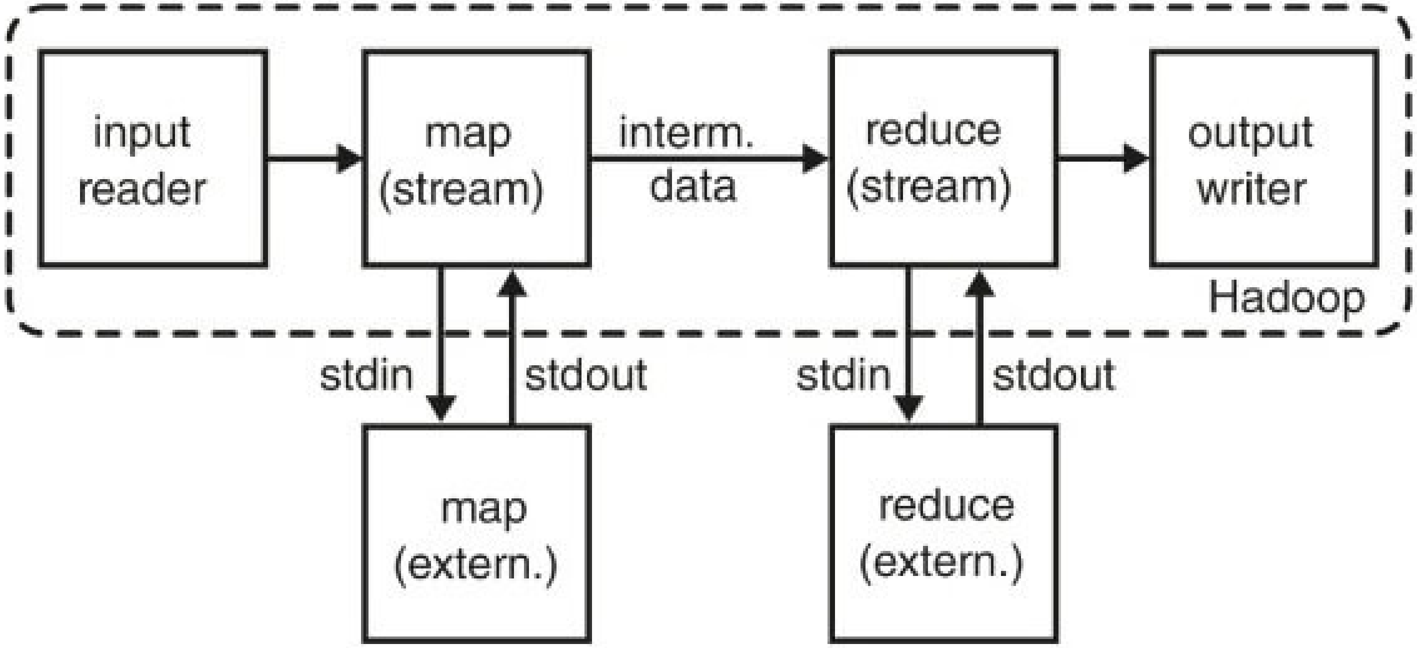
```
ls -a -1 > listing.txt ; wc -l < listing.txt
```

- **Can be piped from one process to another**

```
ls -a -1 | wc -l
```

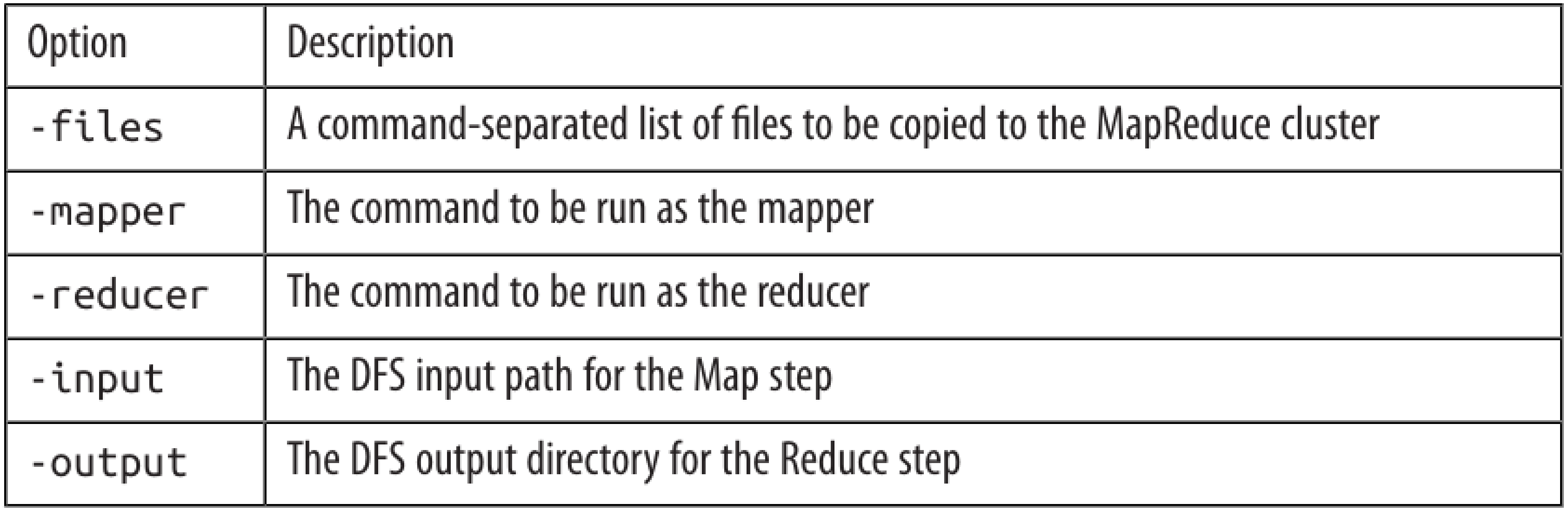
Hadoop Streaming

- **Utility to run MR applications with any executable (e.g. shell utilities, Python programs...) as the mapper or the reducer and Unix stdin/stdout streams as the communication mechanism (no shuffle phase)**



Streaming Command (with YARN)

```
yarn jar <hadoop streaming jar file>
            -input <input data directory>
            -output <output data directory>
            -mapper <mapper program>
            -reducer <reducer program>
```



Option	Description
`-files`	A command-separated list of files to be copied to the MapReduce cluster
`-mapper`	The command to be run as the mapper
`-reducer`	The command to be run as the reducer
`-input`	The DFS input path for the Map step
`-output`	The DFS output directory for the Reduce step

Example in Shell (on dmis)

- Count the number of words in free eBooks of the Gutenberg Project

```
> export PATH=$PATH:$HADOOP_HOME/bin

> export HADOOP_STREAMING_JAR=
    $HADOOP_HOME/share/hadoop/tools/lib/hadoop-
streaming-3.0.0.jar


> yarn jar $HADOOP_STREAMING_JAR
             -input /data/Gutenberg
             -output wc-streaming-output
             -mapper "/bin/sed 's/ /\n/g'"
             -reducer "/usr/bin/uniq -c"
```

Example in Python: mapper

```
#!/usr/bin/env python
import sys
# Read each line from stdin
for line in sys.stdin:
    # Delete leading and ending spaces
    line = line.strip();
    # Get the words in each line
    words = line.split()
    # Generate the count for each word
    for word in words:
        # Write the key-value pair to stdout to be processed by
        # the reducer.
        print (word,'\t','1')
        # The key is anything before the first tab character and the
        # value is anything after the first tab character.
```

Example in Python: reducer (1/2)

```
#!/usr/bin/env python
import sys
prev_word = None
prev_count = 0
# Process each key-value pair from the mapper
for line in sys.stdin:
    line = line.strip()
    # Get the key and value from the current line
    tuple = line.split('\t',1)
    if len(tuple) == 2:
        word = tuple[0]
        value = tuple[1]
    elif len(tuple) == 1:
        word = tuple[0]
        value = 0
    else:
        word = None
        value = 0
```

Example in Python: reducer (2/2)

```
    # Convert the count to an int
    count = int(value)
    # If the current word is the same as the previous word,
    # increment its count, otherwise print the words count
    # to stdout
    if prev_word is None:
        prev_word = word
    if word == prev_word:
        prev_count += count
    else:
        # Write word and its number of occurrences as a key-value
        # pair to stdout
        print(prev_word,'\t',prev_count)
        prev_word = word
        prev_count = count
# Output the count for the last word
if prev_word == word:
    print(prev_word,'\t',prev_count)
```

Example in Python: Test on local machine

- **Get a sample of data:**

```
> scp you@dmis:/share/cluster/cpedago/copernicus/gutenberg/sample.txt .
```

- **Test the mapper: (without Hadoop)**

```
> cat sample.txt | python mapper.py
```

- **Test the reducer: (without Hadoop)**

```
> cat sample.txt | python mapper.py | sort -k1,1 | python reducer.py
```

Example in Python: Run on AWS EMR

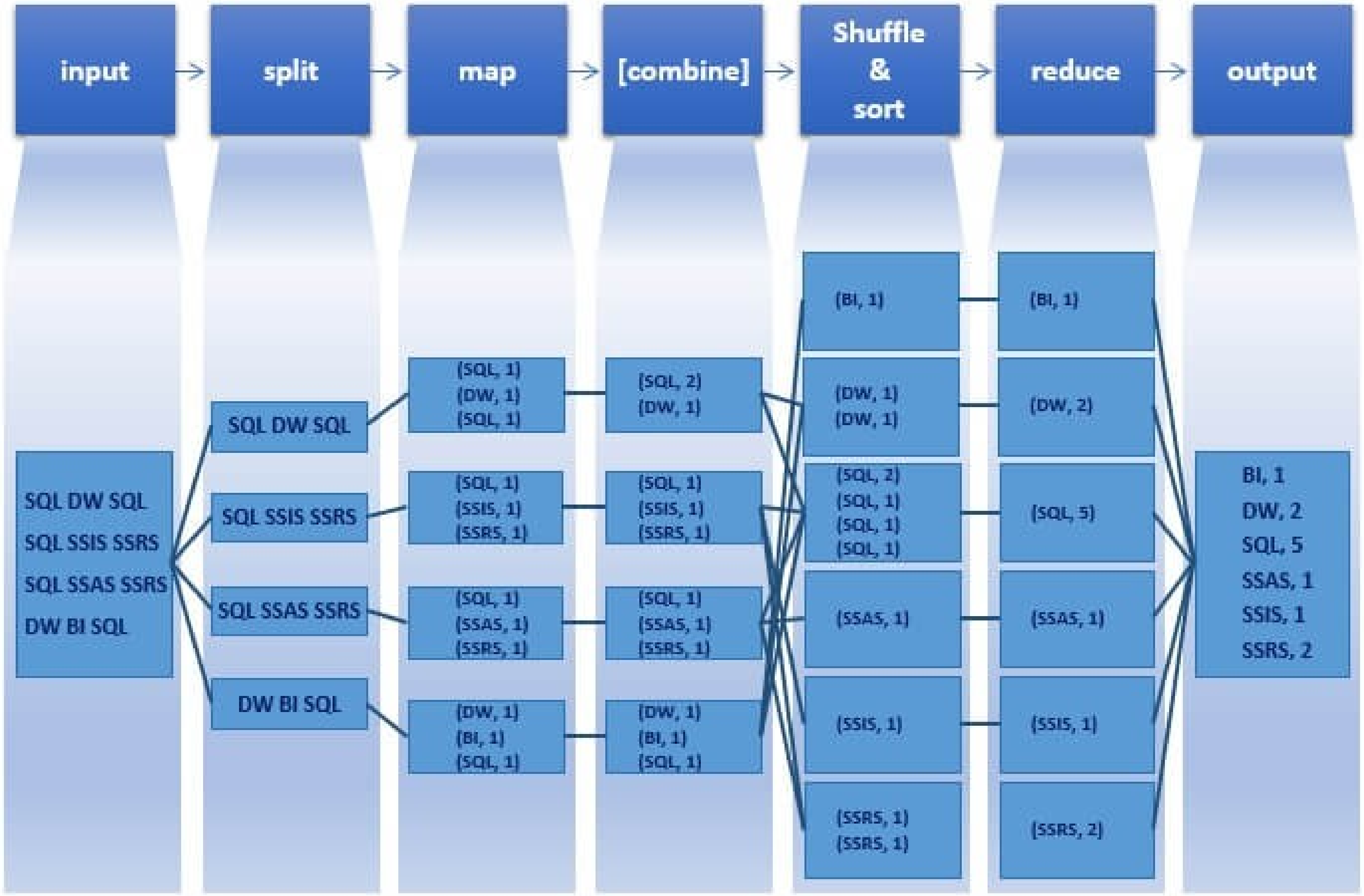
- **Copy Python program on EMR cluster:** `scp`
- **Log on EMR cluster:** ssh
- **Configure env:** `> export HADOOP_STREAMING_JAR=/usr/lib/hadoop/hadoop-streaming.jar`
- **Apply on S3 files**

```
> yarn jar $HADOOP_STREAMING_JAR
     -D mapreduce.job.reduces=10
     -files mapper.py,reducer.py
     -mapper mapper.py
     -input s3://ubs-cde/gutenberg/
     -reducer reducer.py
     -output mapreduce-output-wc
```

MR optimization: Combiner

- **combine: $(K_2,[V_2]) \rightarrow [(K_3,V_3)]$**
  - Optional operation, after map, before shuffling
  - Aggregates locally intermediate data having the same key on the mapper node and outputs it to the reducers
  - Helps to cut down the amount of data transferred from the Mappers to the Reducers, and so the time taken for intermediate data saving and data transfers.
  - Decreases the amount of data that needed to be processed by the reducers, and so improves the overall performance of the reducers.
  - Must be idempotent, i.e.multiple executions of combine operation should not change the result).

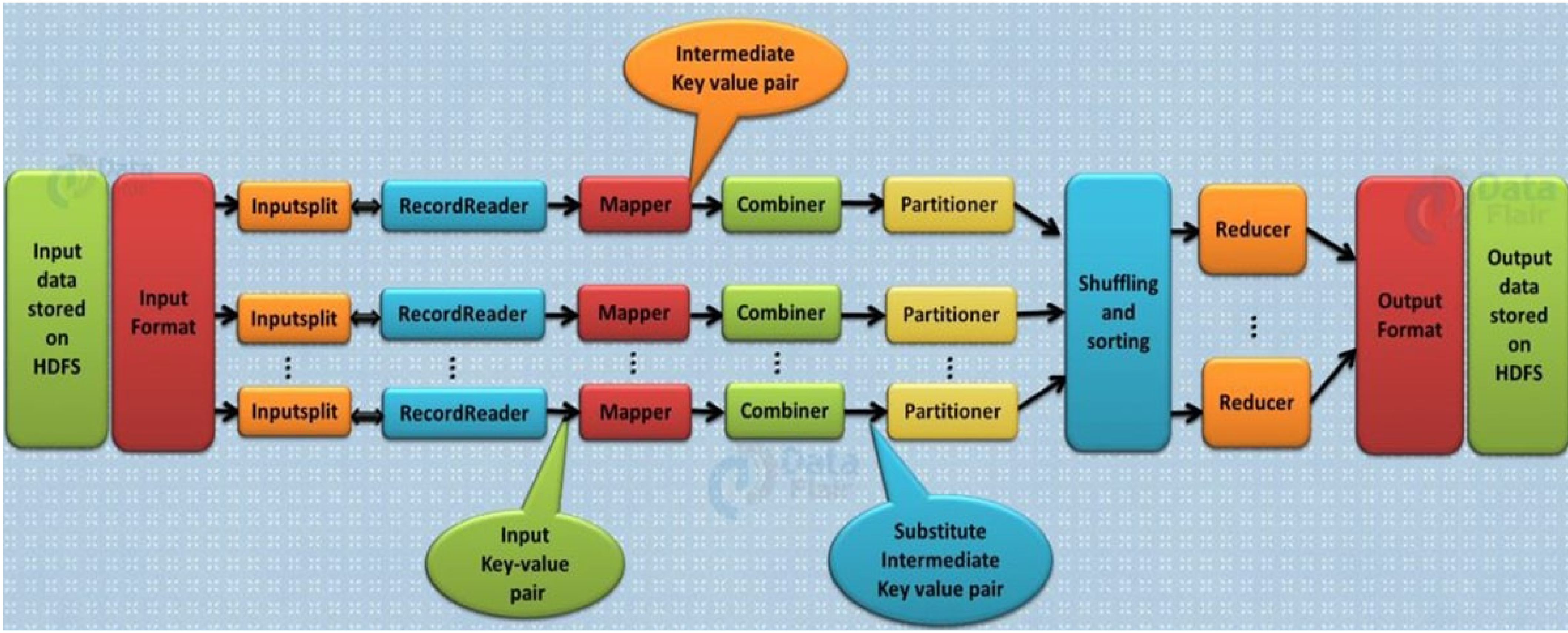
Example: WordCount



Notes on Combiner

- **Combiner input and output key-value pair types should be the same as the Mapper output key-value pair types.**
- **The job's reduce function may be used also as the combiner**
  - In such case the reduce function input and output key-value data types should be the same.
- **In some situations the reduce function can not be used as the combiner**
  - In such case a dedicated reduce function implementation should be written to act as the combiner.
- **Combiner is an optional step of the MapReduce flow.**
  - And not necessarily applied by the runtime system.

Complete MapReduce Workflow



MR Workflow: Reading Phase

- Hadoop MapReduce jobs are divided into a set of map tasks and reduce tasks that run in a distributed fashion on a cluster of computers. Each task works on the small subset of the data it has been assigned so that the load is spread across the cluster.
- The map tasks generally load, parse, transform, and filter data. Each reduce task is responsible for handling a subset of the map task output. Intermediate data is then copied from mapper tasks by the reducer tasks in order to group and aggregate the data.
- The input to a MapReduce job is a set of files in the data store that are spread out over the Hadoop Distributed File System (HDFS). In Hadoop, these files are split with an input format, which defines how to separate a file into input splits. An input split is a byte-oriented view of a chunk of the file to be loaded by a map task.
- The record reader translates an input split generated by input format into records. The purpose of the record reader is to parse the data into records, but not parse the record itself. It passes the data to the mapper in the form of a key/value pair. Usually the key in this context is positional information and the value is the chunk of data that composes a record. Customized record readers may be defined but are outside the scope of this course

MR Workflow: Map Phase

- In the mapper, user-provided code is executed on each key/value pair from the record reader to produce zero or more new key/value output pairs, called the intermediate pairs. The key is what the data will be grouped on and the value is the information pertinent to the analysis in the reducer.
- The combiner, an optional localized reducer, can group data in the map phase. It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in the small scope of that one mapper. In many situations, this significantly reduces the amount of data that has to move over the network.
- The partitioner takes the intermediate key/value pairs from the mapper (or combiner if it is being used) and splits them up into shards, one shard per reducer. By default, the partitioner use the object hashcode of the key to randomly distributes the keyspace evenly over the reducers, but still ensures that keys with the same value in different mappers end up at the same reducer. The default behavior of the partitioner can be customized. The partitioned data is written to the local file system for each map task and waits to be pulled by its respective reducer.

MR Workflow: Reduce and Writing phases

- The reduce task starts with the shuffle and sort step. This step takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running. These individual data pieces are then sorted by key into one larger data list. The purpose of this sort is to group equivalent keys together so that their values can be iterated over easily in the reduce task.
- The reducer takes the grouped data as input and runs a reduce function once per key grouping. The function is passed the key and an iterator over all of the values associated with that key. Once the reduce function is done, it sends zero or more key/value pair to the final step, the output format.
- The output format translates the final key/value pair from the reduce function and writes it out to a file by a record writer. By default, it will separate the key and value with a tab and separate records with a newline character. This can typically be customized to provide richer output formats, but in the end, the data is written out to HDFS, regardless of format. Like the record reader, customizing your own output format is outside of the scope of this course.

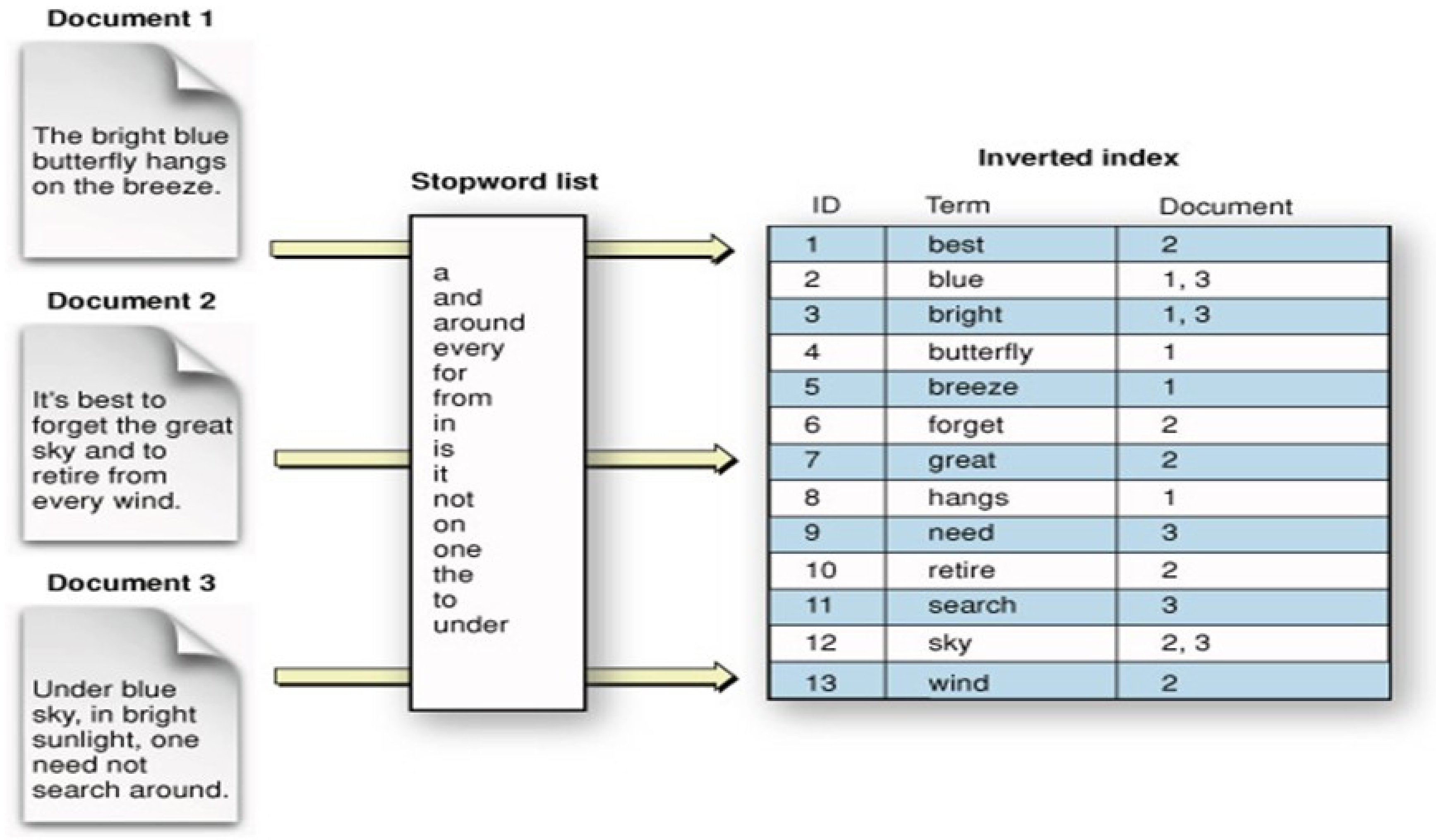
Short Summary

- **MR is a programming model for processing vast amounts of data**
  - A MR job splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner.
  - Both the input and the output of the job are stored in a distributed file system shared by all processing nodes.
  - The MR runtime execution support sorts the outputs of the maps, which are then input to the reduce tasks.
- **MR is also a system for running those programs in a distributed and fault-tolerant way.**
  - Apache Hadoop is one such system, designed primarily to run Java code.
  - The programmer write map(), combine(), and reduce() functions that are submitted to the MR runtime support for execution
    - A mapper takes a single key and value as input, and returns zero or more (key, value) pairs. The pairs from all map outputs of a single step are grouped by key.
    - A combiner takes a key and a subset of the values for that key as input and returns zero or more (key, value) pairs. Combiners are optimizations that run immediately after each mapper and can be used to decrease total data transfer.
    - A reducer takes a key and the complete set of values for that key in the current step, and returns zero or more arbitrary (key, value) pairs as output.
  - The runtime support takes care of scheduling tasks, monitoring them, and re-executing the failed tasks.

Patterns of MR Programs

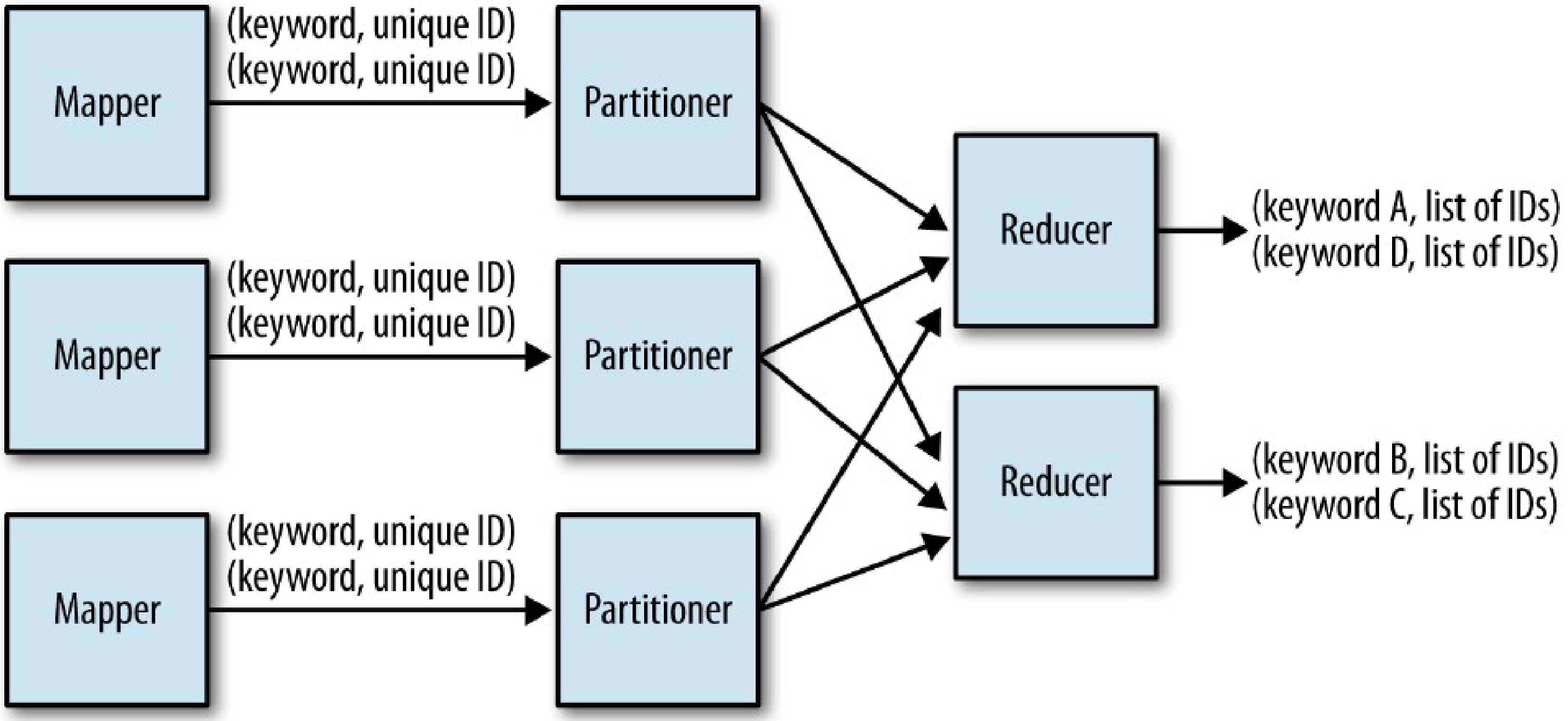
1) Inverted Index
2) Filters
3) Joins

Pattern #1: Inverted Index (1)


Document 1
The bright blue butterfly hangs on the breeze.
Document 2
It's best to forget the great sky and to retire from every wind.
Document 3
Under blue sky, in bright sunlight, one need not search around.
Stopword list
a
and
around
every
for
from
in
is
it
not
on
one
the
to
under
Inverted index

ID	Term	Document
1	best	2
2	blue	1, 3
3	bright	1, 3
4	butterfly	1
5	breeze	1
6	forget	2
7	great	2
8	hangs	1
9	need	3
10	retire	2
11	search	3
12	sky	2, 3
13	wind	2

Pattern #1: Inverted Index (2)

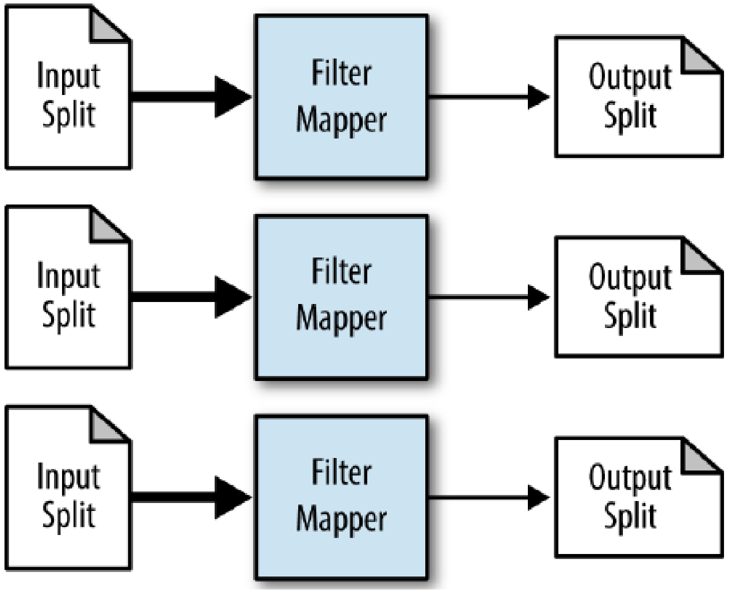


Pattern #2: General Filtering

- **Evaluate each record separately and decides on some condition whether it should stay or go**
- **Mapper:**

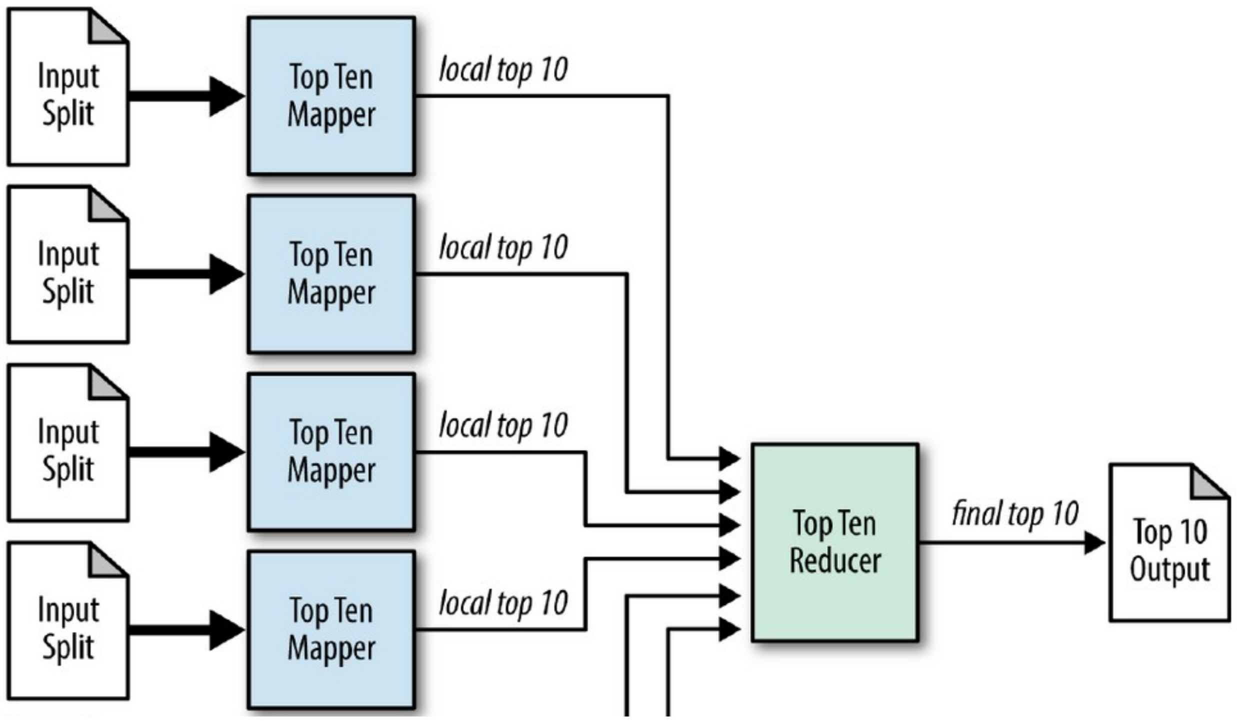
```
map(key, record):
    if we want to keep record then
        emit key,value
```

- **No Reducer**
  - Get one output file per mapper
    - `Filenames part-m-xxxxx`
- **`Examples: grep`**
- **Other kinds of filtering: top10, distinct**



Pattern #2: Top10

- **Retrieve a small number of K records according to a ranking scheme**
  - Need to define a comparison function between two records
- **Unique reducer**
  - To be used for small value of K



Example #2: Distinct

- **Filter out records that look like another records in the data set**
- **Exploits MR ability to group keys together during the shuffle phase**

```
map(key, record):
    emit record,null
```

```
reduce(key, records):
    emit key
```

- **Duplicate records are often located close to another in the data set, so a combiner will deduplicate them just after the map phase**

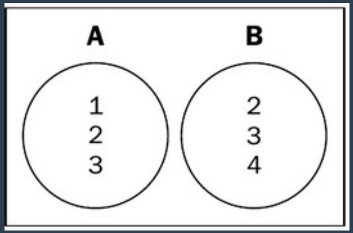
Pattern #3: Joining

- **Having all data in one giant data set is not frequent**
  - Interesting relationships can be discover when starting analyzing multiple data sets together
  - Joins use a lot of network bandwidth
  - Choosing the right type of join is critical
- **Studied in this patterns category:**
  - Reduce Side Join
  - Replicated Join

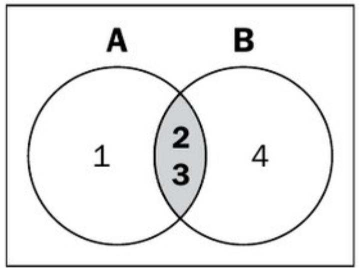
Pattern #3: Join Operator

- **Combines records from data sets based on fields, called the foreign key.**
  - The foreign key is the field in a table that is equal to the column of another table, and it is used as a means to cross-refer between tables (see SQL).
- **Different types of joins can be performed on the datasets: inner, left/right/full outer.**
  - interactive example at https://joins.spathon.com/

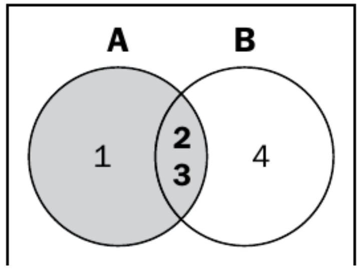
Pattern #3: Inner and Outer Joins

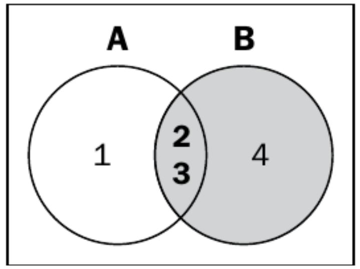


- Inner join: all the matching records from both the datasets are returned

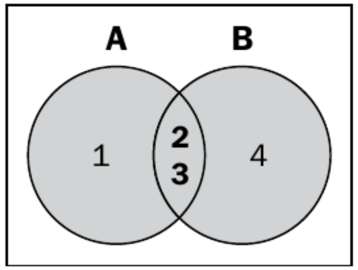


- Left (right) outer join: all the matching records from both the datasets are returned along with the unmatched records from the dataset on the left-hand (right-hand) side.



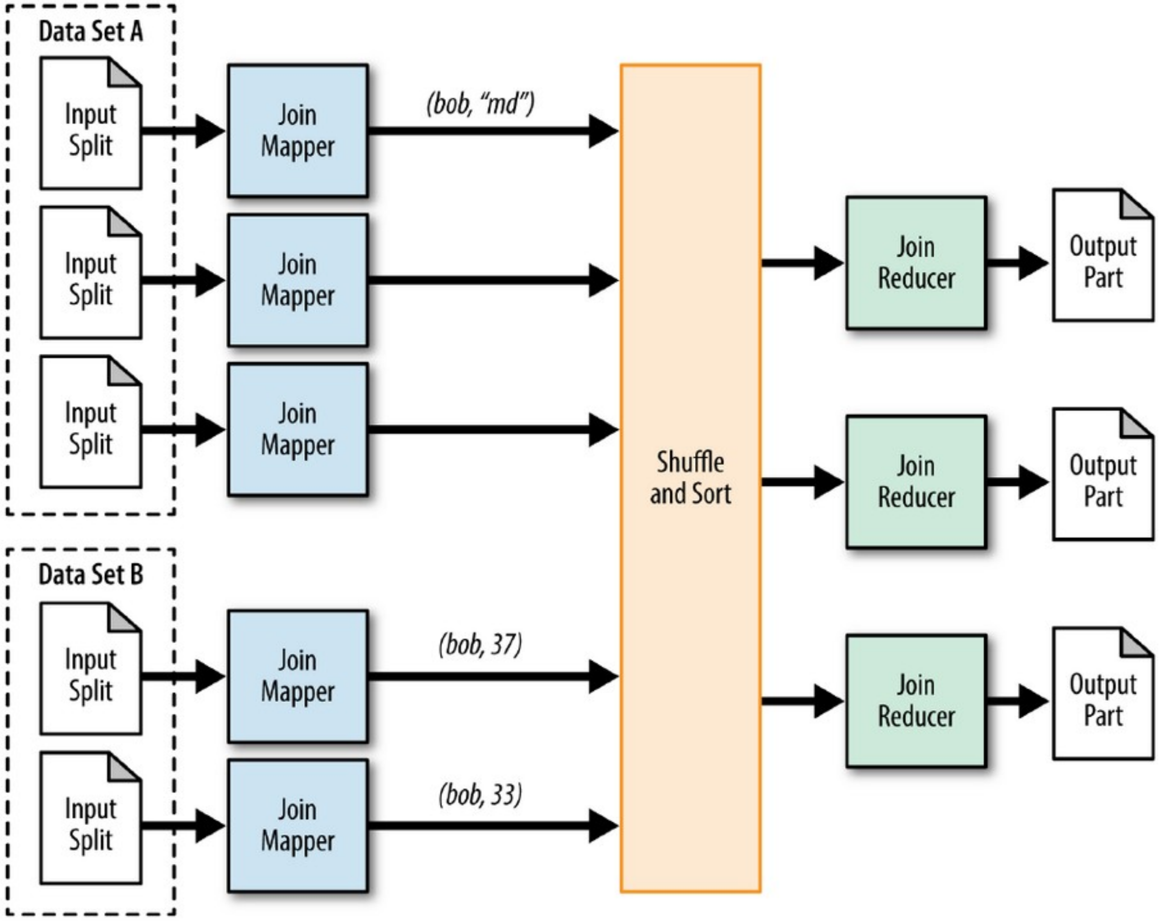


Full outer join: all the matching records from both datasets are returned along with the unmatched records from both tables.



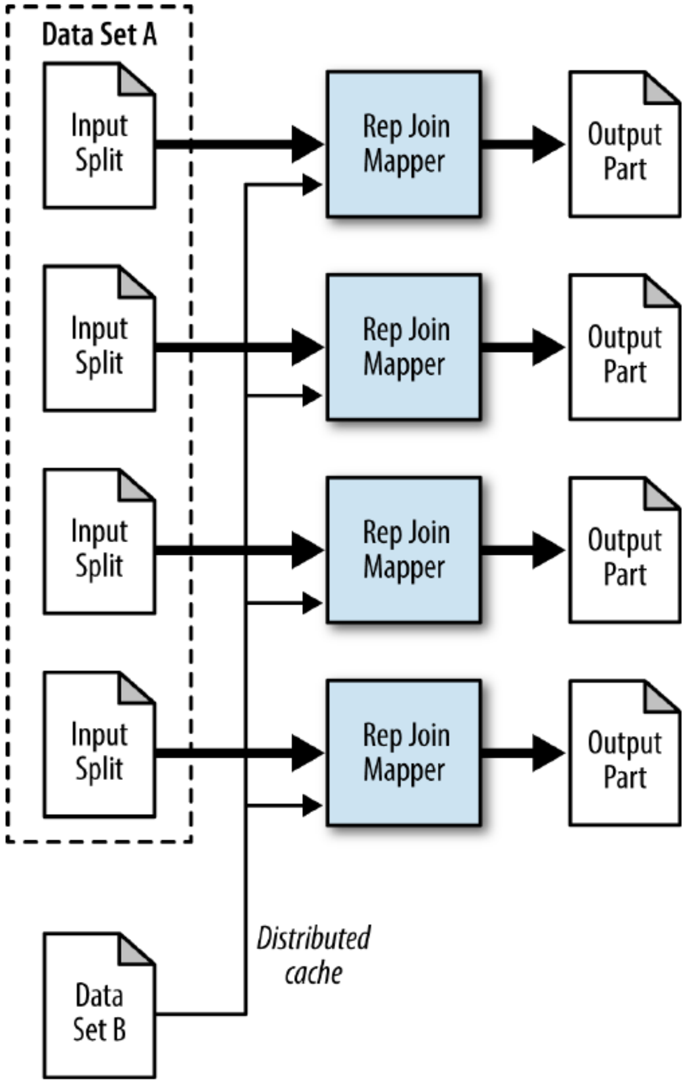
Pattern #3: Reduce Side Join

- **Can be used to execute any type of joins, with any number of data sets**
- **Structure:**
  1) Flag the data sets by a group identifier on map outputs
  2) Performs the desired join by collecting the values of each input group into temporary lists.



Pattern #3: Map Side Join

- **Inner or left outer join operation between one large and many small data sets**
  - The large data set is the “left” data set
  - The small sets are stored into a fast memory
- **The mapper is responsible for reading all files from the distributed cache during the setup phase and storing them into in-memory lookup tables.**
- **Join is performed on the map-side**
  - one of the fastest type of join executed.

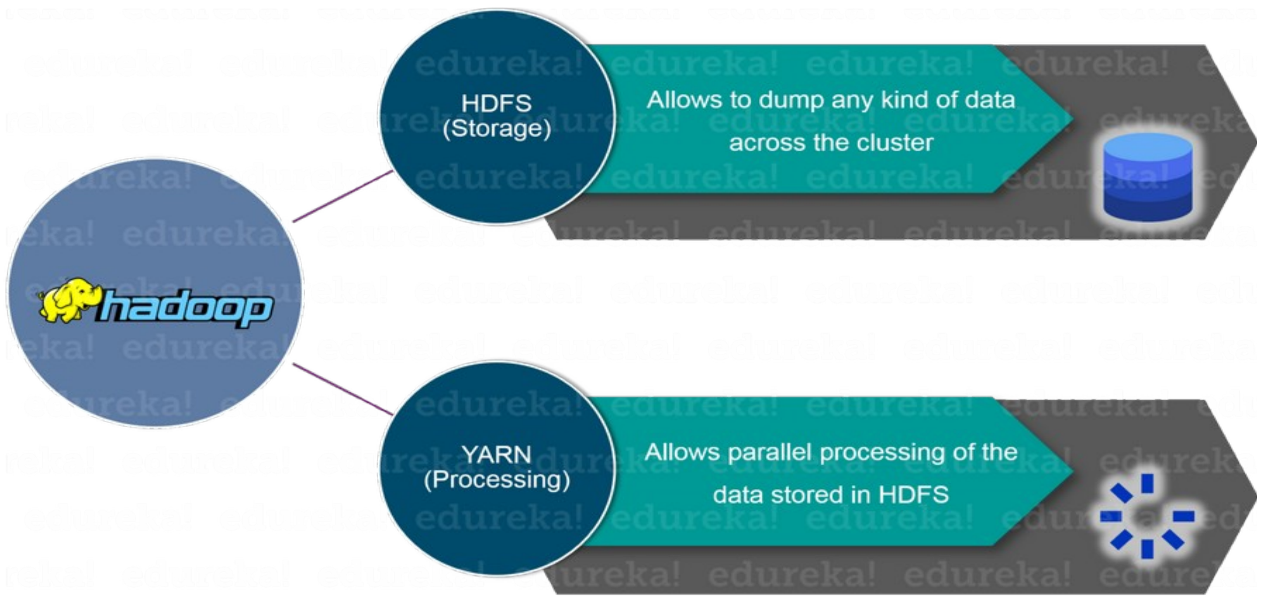


Outlines

1. MR Programming Model
2. Hadoop MR Runtime Support
3. Mrjob package

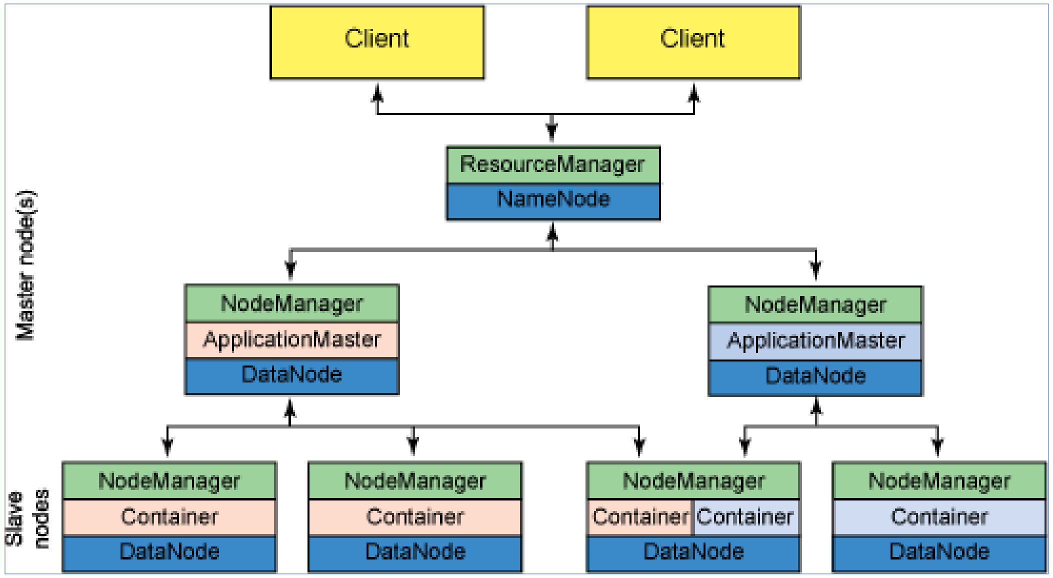
Hadoop MapReduce Runtime Support

- **Hadoop MR applications need a distributed file-system service and a parallel processing service running on a cluster**
  - HDFS
  - YARN


hadoop
HDFS (Storage)
Allows to dump any kind of data across the cluster
YARN (Processing)
Allows parallel processing of the data stored in HDFS

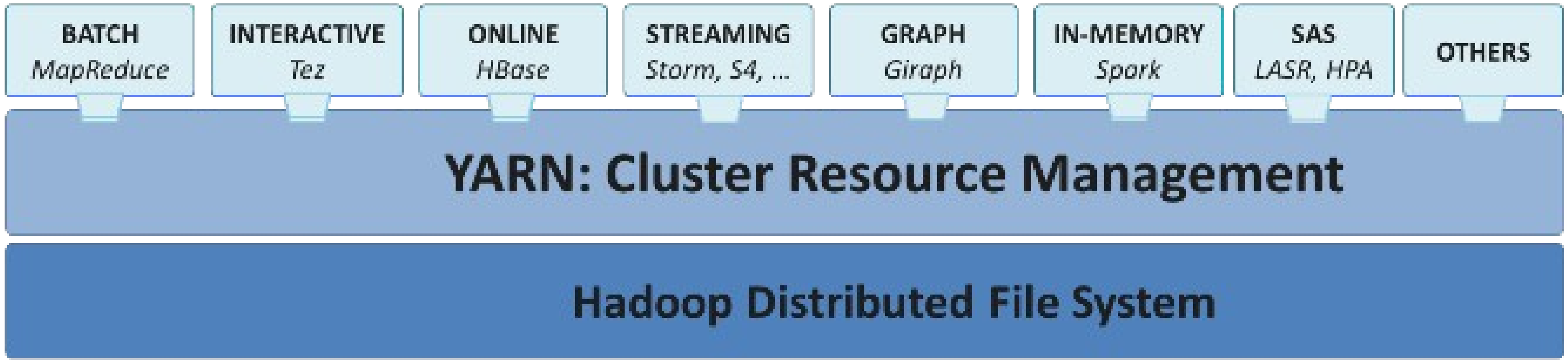
YARN & HDFS Nodes

- **Recall: HDFS is a distributed service implemented using the master/slave parallel programming model**
  - NameNode
  - DataNodes
- **Same for YARN:**
  - ResourceManager (RM)
  - NodeManagers (NM)
- **Same for Application using YARN (YARN Clients)**
  - ApplicationMaster (AM)
  - Slaves running on Container manager by NMs

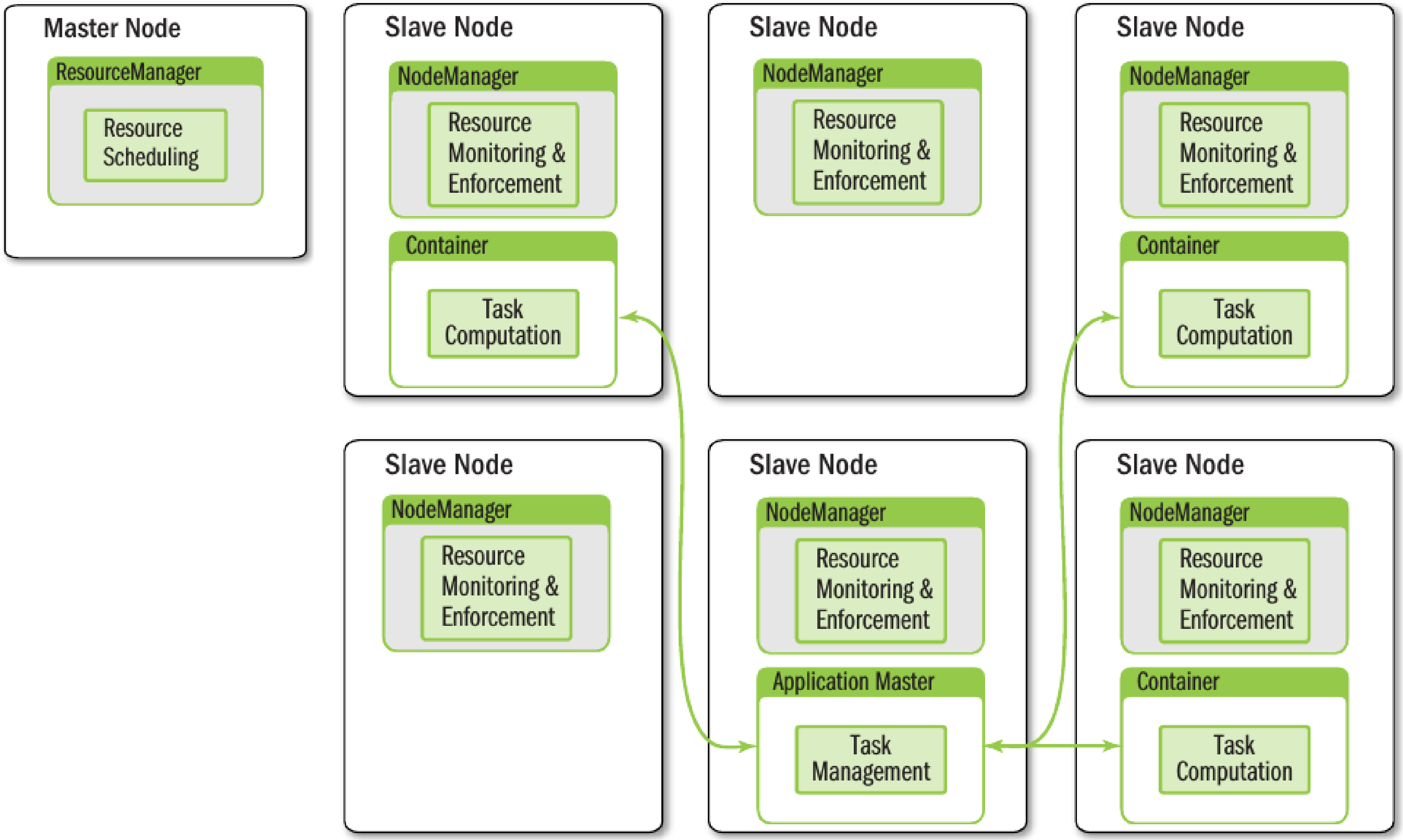


What is YARN ?

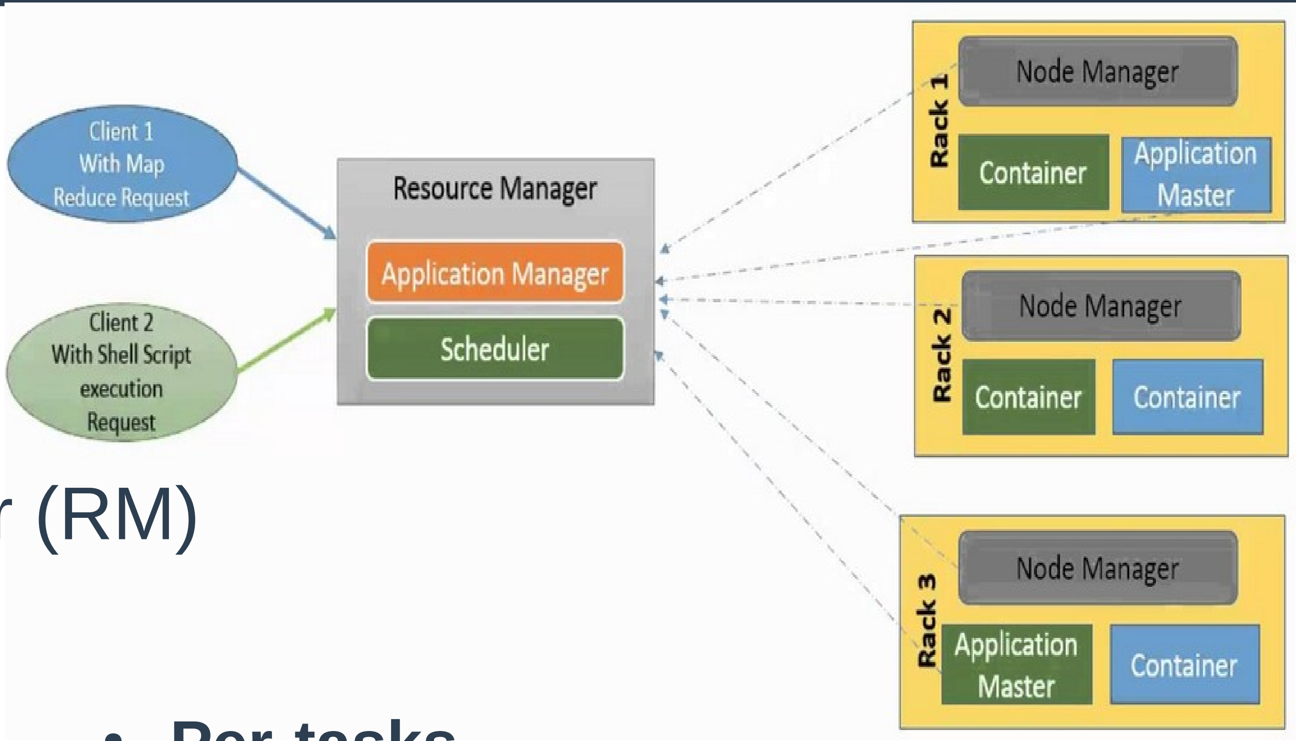
- **A generic resource manager in a typical cluster**
  - Conceived and architected at Yahoo! (2008)
  - Introduced with Hadoop 2.x, aka MRv2 (2013)
- **YARN allows different data processing engines to run and process data stored in HDFS.**



YARN Architectural Overview



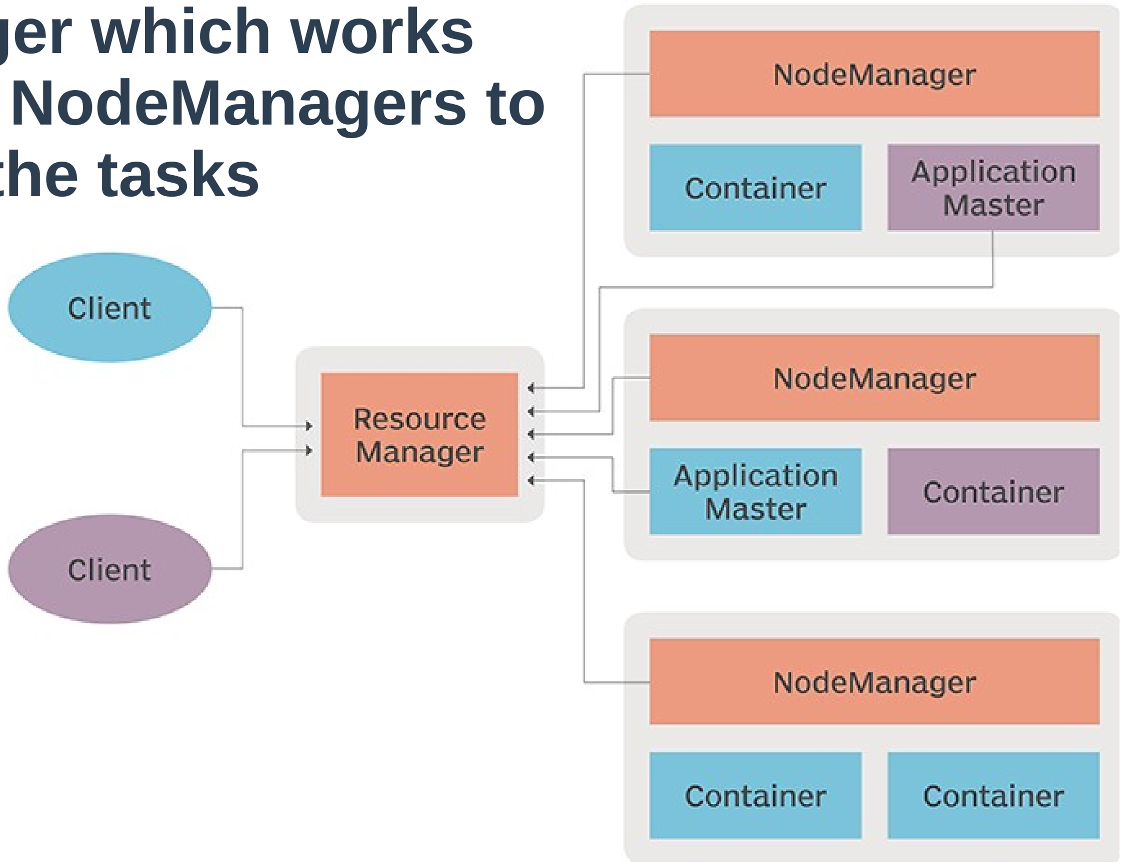
YARN Components



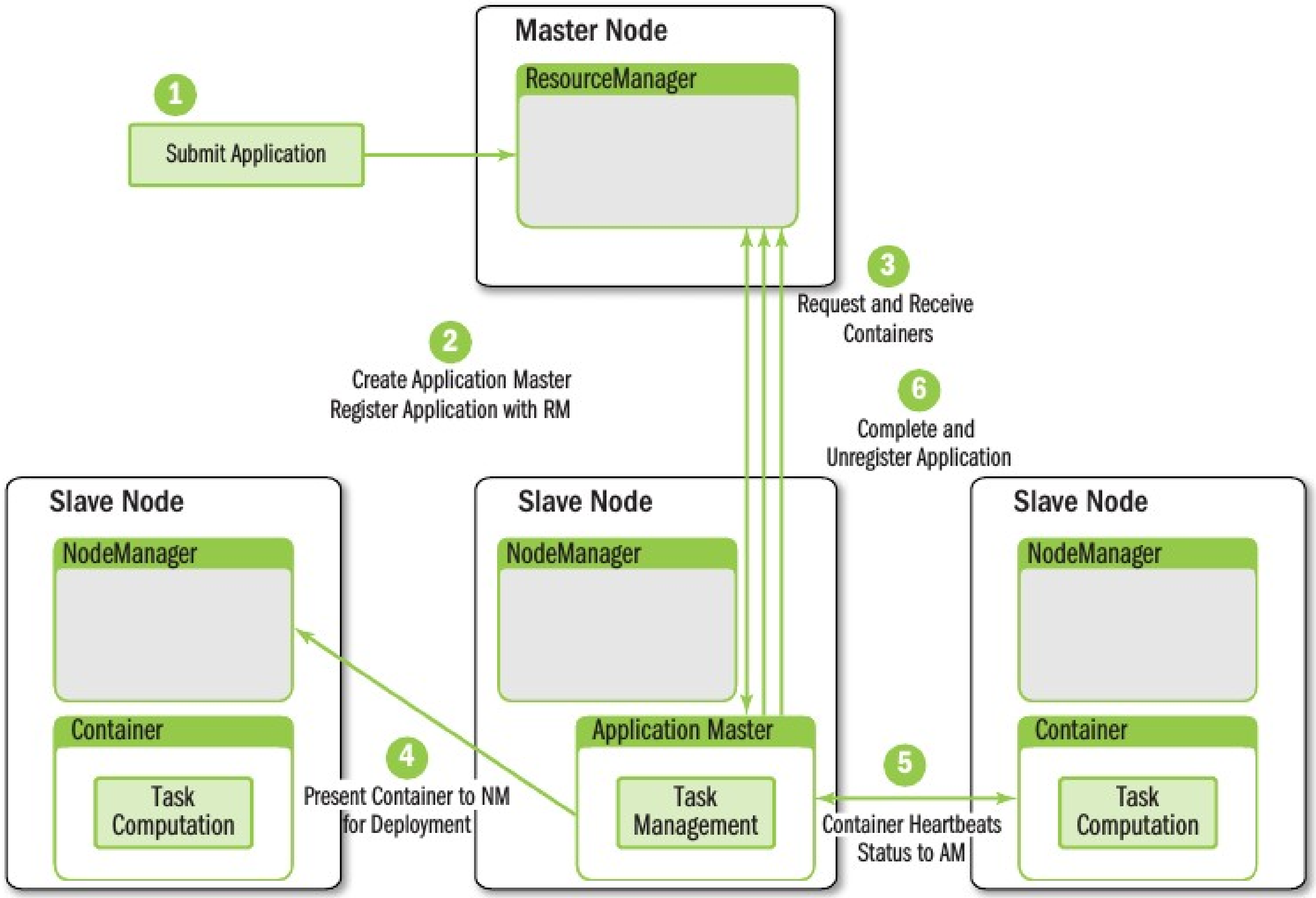
- **Per-cluster**
  - Resource Manager (RM)
- **Per-node**
  - Node Manager (NM)
- **Per-application**
  - Application Master (AM)
- **Per-tasks**
  - Containers

YARN Architecture

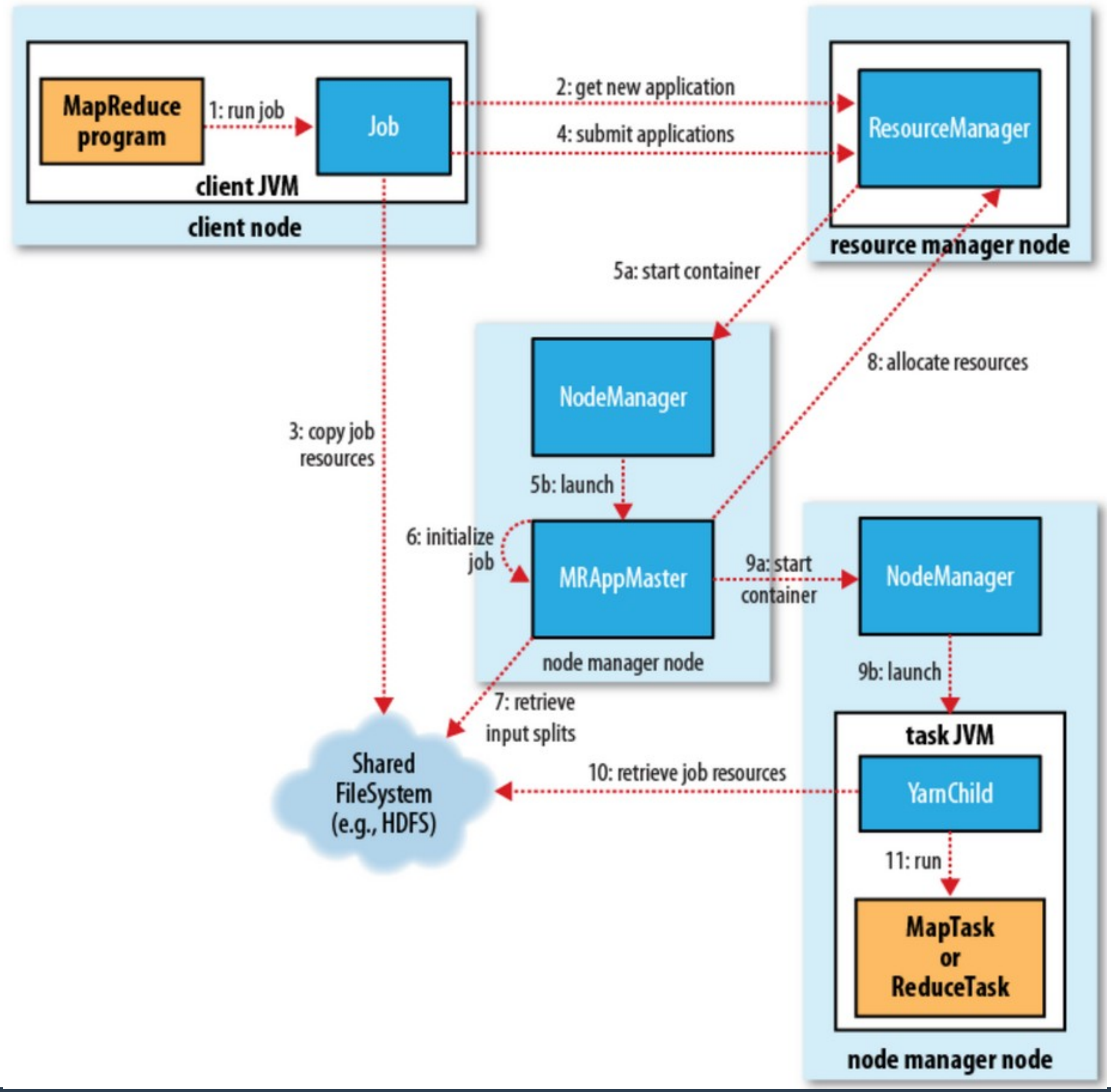
- Application clients submit processing tasks to YARN’s RessourceManager which works with ApplicationMaster and NodeManagers to schedule, run and monitor the tasks
- Several applications may run concurrently on the cluster, depending on the available resources (container)



Generic Application Data Flow



MR Application Data Flow

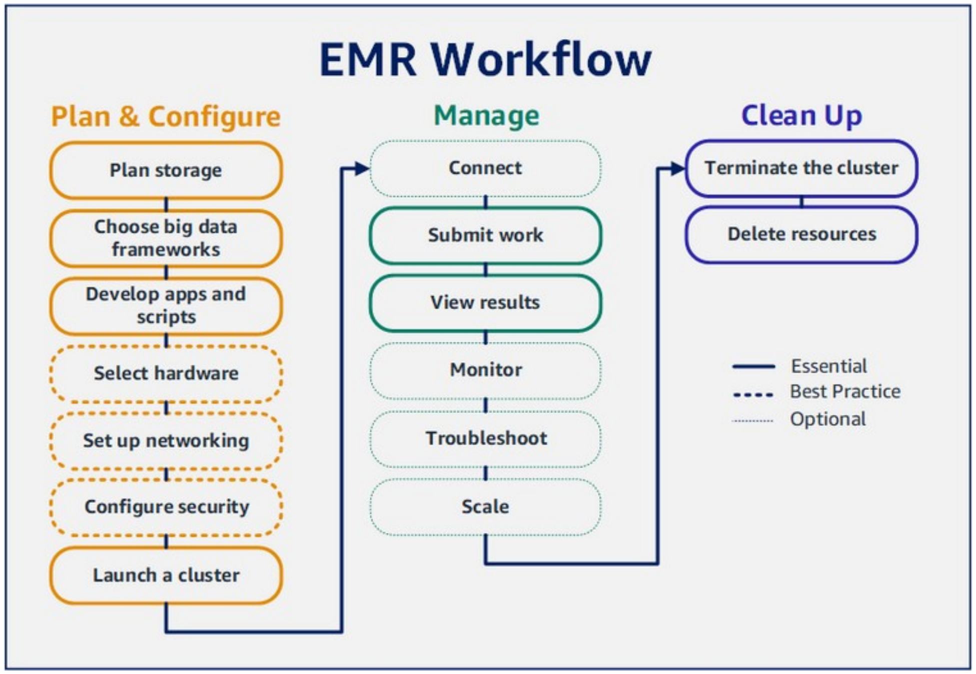


Managing YARN Jobs

- **Launching:**
```
yarn jar $HADOOP_STREAMING_JAR
    -input <input data directory>
    -output <output data directory>
    -mapper <mapper program>
    -reducer <reducer program>
```
- **Monitoring:**
```
yarn application –list
yarn application –status <app id>
```
- **Logs viewing:**
```
yarn logs -log_files=stderr <app id>
yarn logs -log_files=stdout <app id>
```
- **Killing:**
```
yarn application –kill <app id>
```

Amazon Elastic MapReduce

- **A platform managing virtual clusters dedicated to big data analytics**
- **EMR clusters**
  - Composed of nodes (EC2 instances)
    - Master node + Core nodes
  - Resource management by YARN
- **Storage**
  - Intermediate results in HDFS
  - Input and output data in S3
- **Data processing frameworks**
  - Hadoop MapReduce
  - Apache Spark
  - Hive, HBase, ...



Outlines

1. MR Programming Model
2. Hadoop MR Runtime Support
3. Mrjob package

Mrjob Package

- **Python MapReduce library**
  - Created by Yelp, Inc.
- **Wraps Hadoop Streaming**
  - Allows MR apps to be written in a more Pythonic manner
  - Enables multi-step MR jobs
- **Can be tested locally, run on a Hadoop cluster or run in the cloud (e.g. Amazon EMR, Google Dataproc)**
- **(also easily run Spark jobs written in Python on a Hadoop cluster)**

WordCount (V1)

```
from mrjob.job import MRJob

class MRWordCount(MRJob):

    def mapper(self, _, line):
        for word in line.split():
            yield(word, 1)

    def reducer(self, word, counts):
        yield(word, sum(counts))

if __name__ == '__main__':
    MRWordCount.run()
```

- **A job is a class that inherits from MRJob**
- **This mapper ignores the input key (, _,)**
- **The last two lines are mandatory**

- **Takes advantage of the concept of Python generators, see https://stackoverflow.com/questions/231767/what-does-the-yield-keyword-do or https://www.pythoncentral.io/python-generators-and-yield-keyword/**

Running a mrjob Different Ways

- **In a single Python process (for debugging purpose)**

  > python word_count.py **-r inline** words.txt
  > python word_count.py Gutenberg/*.txt

- **In multiple subprocesses using the local filesystem**

  > python word_count.py **-r local** words.txt

- **On a Hadoop cluster**

  > python word_count.py **-r hadoop** hdfs:///data/Gutenberg/*.txt

- **On Amazon EMR**

  > python word_count.py **-r emr** s3://ubs-cde/gutenberg/*.txt

  – Assumes you have an Amazon account and credentials
    - you will need to run you MR jobs from the master node of the EMR cluster we have created for you, so with "**-r hadoop**"

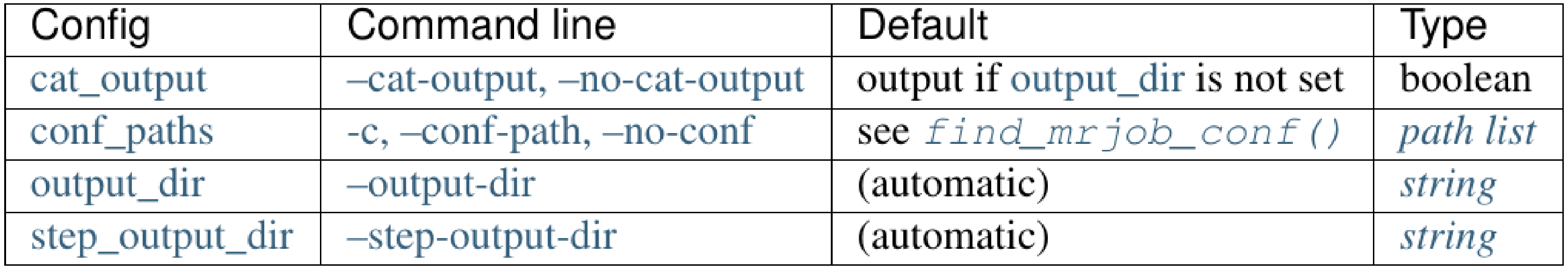
WordCount (V2)

```
from mrjob.job import MRJob
import re

WORD_RE = re.compile(r"[\w']+")

class MRWordCount(MRJob):

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield(word.lower(), 1)

    def combiner(self, word, counts):
        yield(word, sum(counts))

    def reducer(self, word, counts):
        yield(word, sum(counts))

if __name__ == '__main__':
    MRWordCount.run()
```

- **Use Regular Expression to extract words without ponctuations, numbers…**
- **Lower the words**
- **Add a combiner (local reducer)**

Mrjob Configuration (1/2)

- **Hadoop streaming args -D <key>=<value>, e.g.:**
  `-D mapreduce.job.reduces=10` **`(Default: 1 reducer)`**
- **Command-line switches:**



Config	Command line	Default	Type
cat_output	–cat-output, –no-cat-output	output if output_dir is not set	boolean
conf_paths	-c, –conf-path, –no-conf	see `find_mrjob_conf()`	*path list*
output_dir	–output-dir	(automatic)	*string*
step_output_dir	–step-output-dir	(automatic)	*string*

- **Example:**

```
> python mrjob_word_count_v2.py
           -r hadoop
           --output-dir mrjob-output-wc
            -D mapreduce.job.reduces=10
          s3://ubs-cde/gutenberg
```

Mrjob Configuration (2/2)

- **MRJob.JOBCONF attribute (dictionary)**
- **Example:**

```
class MRWordCount(MRJob):

    JOBCONF={
       'mapreduce.job.reduces':'10',
       }

    def mapper(self, _, line):
        for word in WORD_RE.findall(line):
            yield(word.lower(), 1)

    def reducer(self, word, counts):
        yield(word, sum(counts))

    combiner = reducer
```

Multi-steps Jobs

- **Most of the time, you'll need more than one step in your job.**
  - To define multiple steps, override the function `MRJob.steps()` to return a list of `MRStep`.
- **Example: find the most commonly used word**
  - Step 1: same as `MRWordCount` class
    - Produce all the `(word,count)` pairs
  - Step 2:
    - Mapper:
      - Each takes part of the `(word,count)` pairs
      - And produces one pair having the partial maximum counter
    - Reducer: selects the pair having the maximum counter

Multi-steps: Example (1/2)

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re
WORD_RE = re.compile(r"[\w']+")
class MRMostUsedWord(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.reducer_count_words,
                   reducer=self.reducer_count_words
                   jobconf={'mapreduce.job.reduces':'18'}),
            MRStep(mapper_init=self.mapper_find_max_init,
                   mapper=self.mapper_find_max,
                   mapper_final=self.mapper_find_max_final,
                   reducer=self.reducer_find_max
                   jobconf={'mapreduce.job.reduces':'1'}),
      ]
```

Multi-steps: Example (2/2)

```
def mapper_find_max_init(self):
   self.top_word,self.top_count='',0

def mapper_find_max(self, word, count):
   if count>self.top_count:
     self.top_word,self.top_count= word,count

def mapper_find_max_final(self):
   yield (None,(self.top_word, self.top_count))

def reducer_find_max(self, _, word_count_pairs):
   max_word,max_count='',0
   for word_count in word_count_pairs:
      if word_count[1]>max_count:
        max_count_word,max_count= word_count[0],word_count[1]
   yield (max_word,max_count)
```

Counters

- **Hadoop tracks counters aggregated over a step**
  - Automatically printed when a MR job finishes
  - Built-in counters
    - Gather statistics about MR jobs
      - MR task counters (number of records read/written...)
      - Filesystem counters (bytes read/written…)
      - MR job counters (# launched map tasks, failed map tasks,...)
    - Way to measure the progress or the number of operations within jobs
  - User-defined counters
    - Defined programmatically

Custom Counters

- **A user-defined counter has a group (string), a name (string) and an integer value**
- **Example: count the number of lines and words in wc**

```
def mapper(self, _, line):
   self.increment_counter('lines','total',1)
   for word in WORD_RE.findall(line):
     self.increment_counter('words','total',1)
     yield(word.lower(), 1)

def reducer(self, word, counts):
  self.increment_counter('words','unique',1)
  yield(word, sum(counts))
```