CSP554 BIG DATA TECHNOLOGIES

Module 3b

Map Reduce

More About S3 and HDFS

	S3	HDFS
Data Organization	Object Store • Buckets • Objects	File System • Directories • Nested Directories • Files
Naming	Bucket Name • must be unique across all of AWS Object Key • Unique within the scope of a bucket	Directory Name • Unique within a directory File Names • Unique within a directory
Size	Up to 5TB per object • Use multiple objects in a bucket for data larger than 5TBs • Partition data into its most natural groupings such as one object per State or Sensor, etc.	Essentially unlimited file size • Partitioning into multiple files used to improve query performance
Elasticity (Up-Sizing Storage)	Essentially unlimited	Limited by cluster configuration
Append Data once Written?	No	Yes
Update Data once Written?	No	No

More About S3 and HDFS

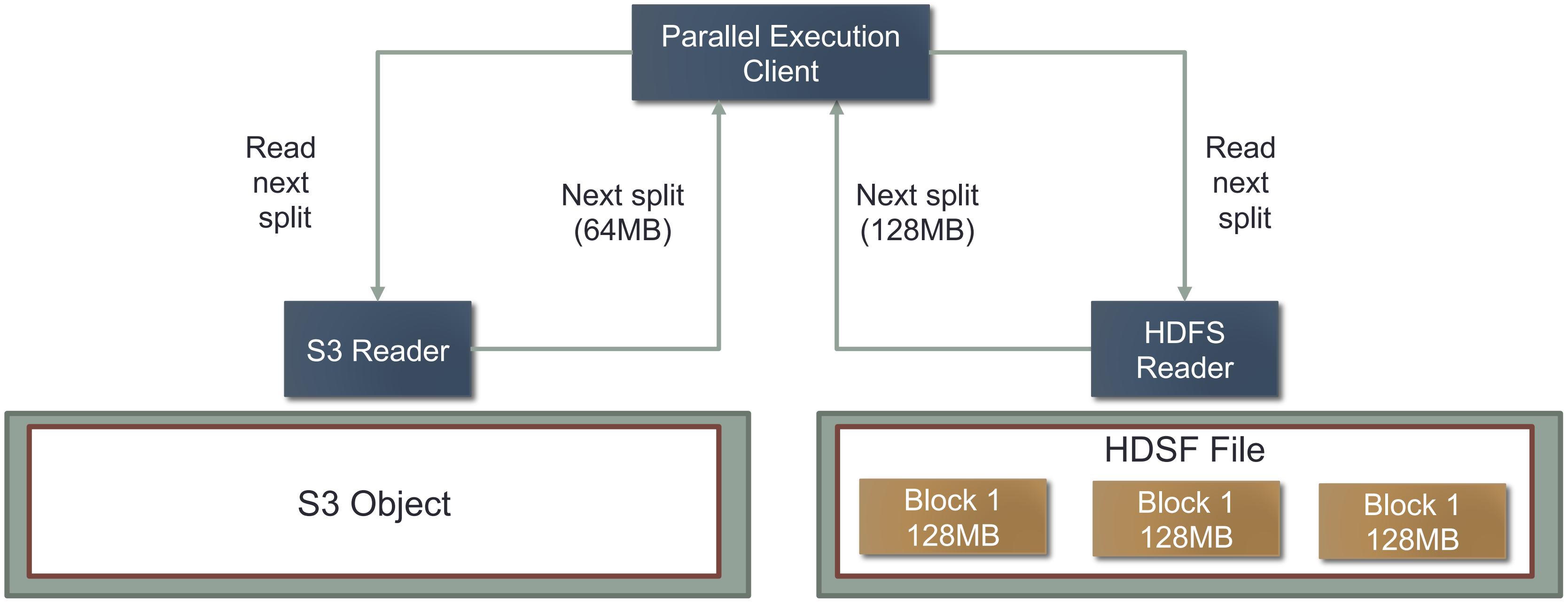
	S3	HDFS
Availability	99.99%	99.9% (estimated)
Durability	99.999999999%	99.9999% (estimated)
Eventual Consistency	Yes (see next slide)	No (strong consistency)
Storage Cost Per TB (AWS)	$20/month	$200/month
Persistent On AWS EMR (Hadoop) Cluster Termination	Yes	No
Performance for Parallel Processing (Hadoop/Spark)	• Much better performance versus its storage cost	• 2x faster for parallel processing than S3
Best Used For	• Persisting large volumes of data being kept long term • Useful in support of the data lake patten (to be discussed later)	• When maximum performance is required • When data is frequently accessed • For transient intermediate results

Blocks Versus Splits

- Files in HDFS are divided into physical chunks call blocks
- Each block is a fixed size and stored as regular files in the Linux file system
- But parallel processing engines and their related tools were designed to be decoupled from any specific file system
- This allows these tools and parallel processing engines to work with both S3 and HDFS
- And it is especially useful as S3 objects have no concept at all of blocks; they are just unified collections of bytes
- So the question is… how is this decoupling is managed, and the answer is through the concept of splits

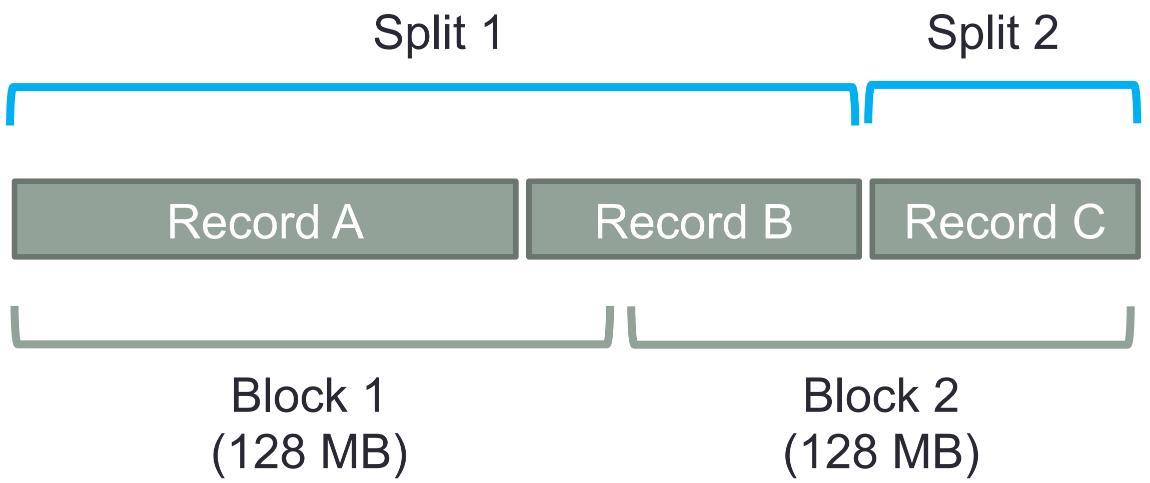
Blocks Versus Splits

- A split is a logical chunk of an underlying HDFS file or S3 object
- A reader object associated with the parallel processing engine accesses data from HDFS or S3 and presents it as splits
- For HDFS the size of a split is by default the size of an HDFS block (128MB)
- For S3 objects the size of a split is by default 64MB



Another Reason Splits Are Important

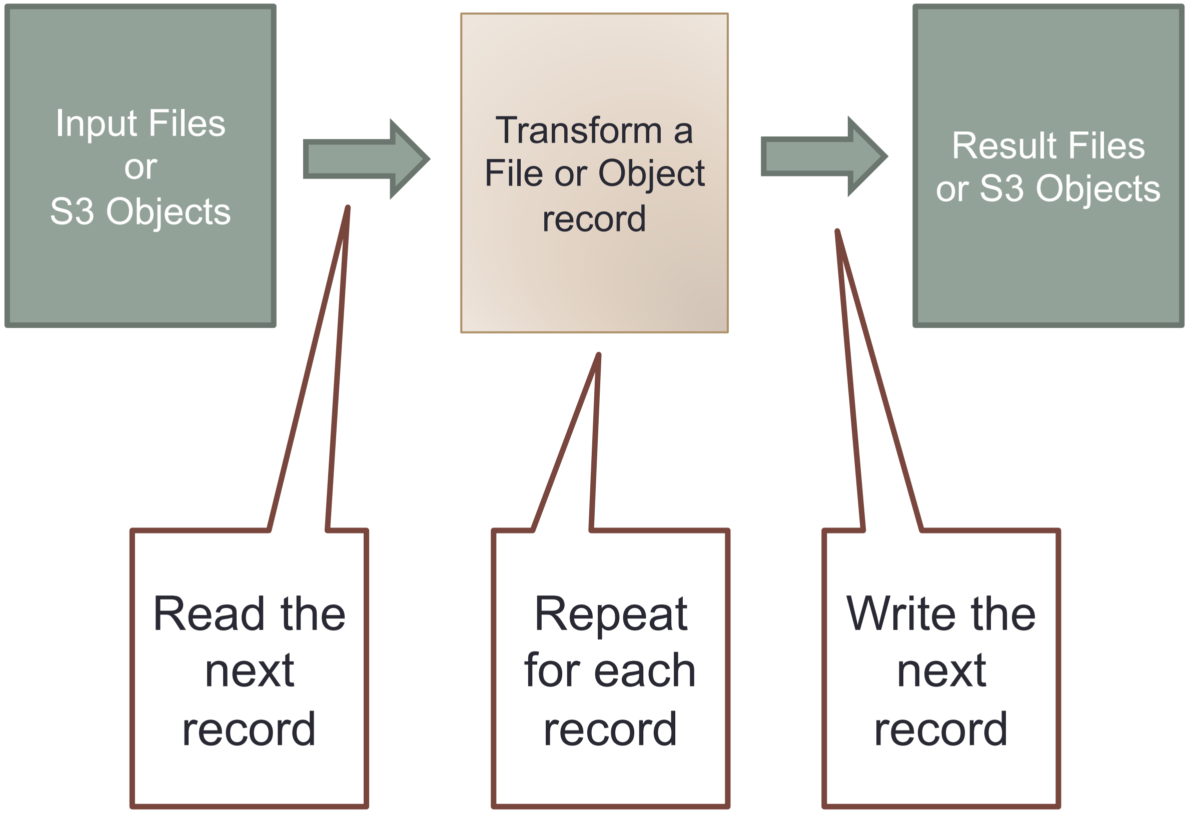
- Records might be stored across two blocks, but are combined into one split
- Splits are logical, they don't store or contain actual data and just refer to the data which is stored in HDFS



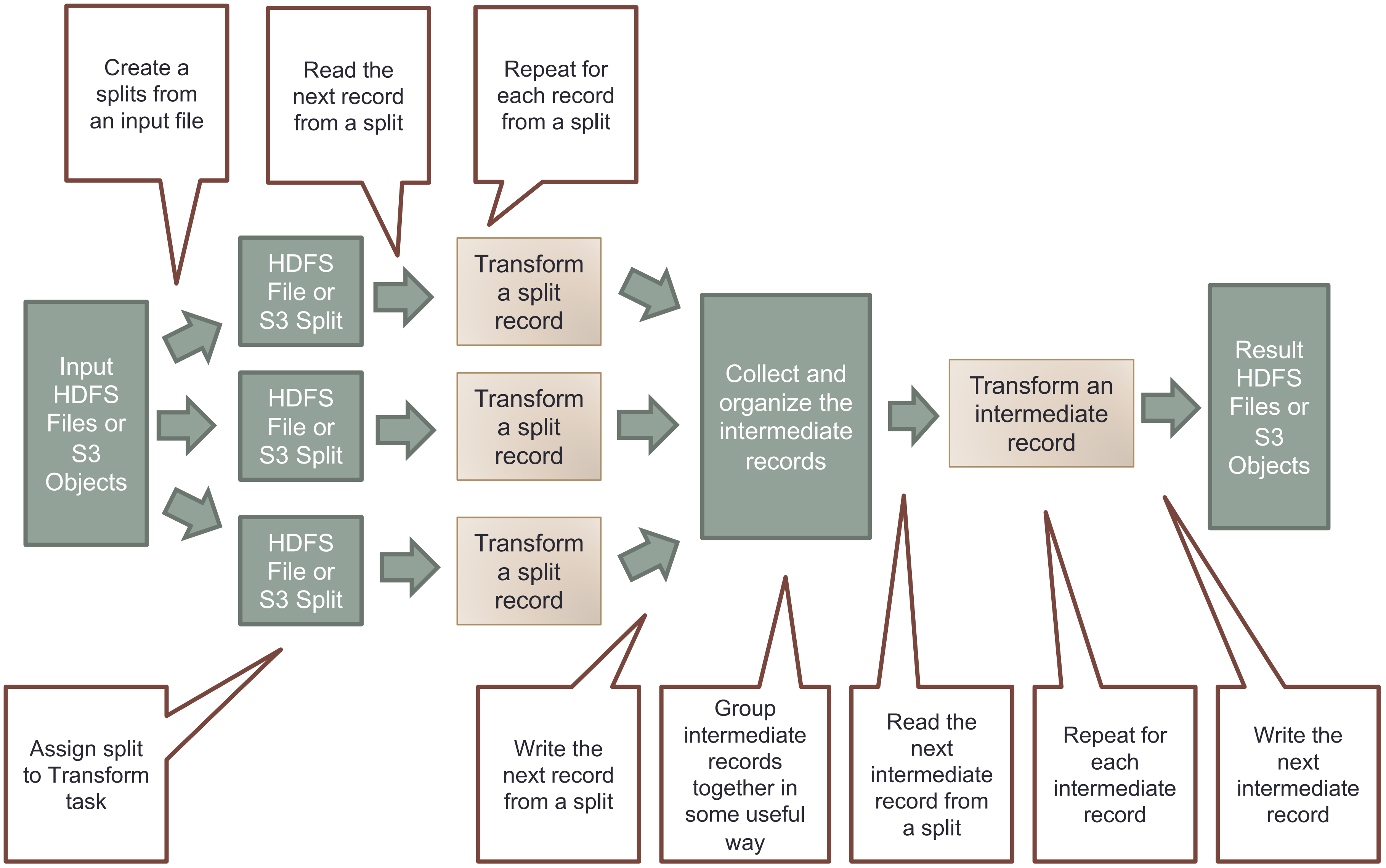
Programming the Batch Data Analysis Process

- Batch data analysis…
  - Input is read from one or more files or S3 objects
  - Output is written to one or more files or S3 objects
- For a small to moderate volume of data
  - Design and code a sequential process on one server
  - Input and output files are read/written using native OS file operations
- For a large volume of data
  - Design and code parallel processes on a cluster of servers/VMs
  - Use a parallel batch execution engine to organize the flow of computation across the cluster
  - Uses a distributed file system or high capacity object store

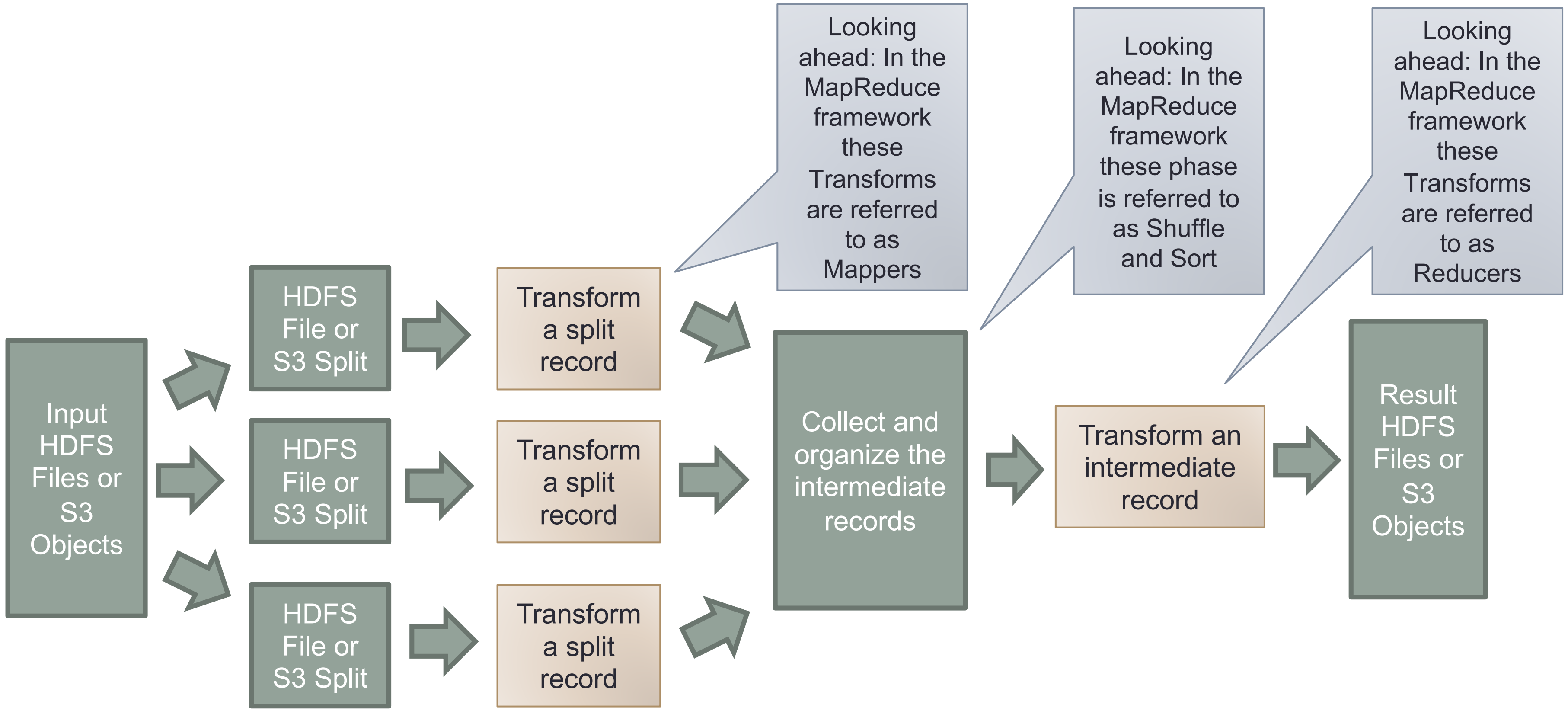
Sequential Batch Data Analysis


Input Files or S3 Objects
Transform a File or Object record
Result Files or S3 Objects
Read the next record
Repeat for each record
Write the next record

Parallel Batch Data Analysis


Create a splits from an input file
Read the next record from a split
Repeat for each record from a split
Input HDFS Files or S3 Objects
HDFS File or S3 Split
HDFS File or S3 Split
HDFS File or S3 Split
Transform a split record
Transform a split record
Transform a split record
Collect and organize the intermediate records
Transform an intermediate record
Result HDFS Files or S3 Objects
Assign split to Transform task
Write the next record from a split
Group intermediate records together in some useful way
Read the next intermediate record from a split
Repeat for each intermediate record
Write the next intermediate record

Parallel Batch Data Analysis


Input HDFS Files or S3 Objects
HDFS File or S3 Split
HDFS File or S3 Split
HDFS File or S3 Split
Transform a split record
Transform a split record
Transform a split record
Looking ahead: In the MapReduce framework these Transforms are referred to as Mappers
Collect and organize the intermediate records
Looking ahead: In the MapReduce framework these phase is referred to as Shuffle and Sort
Transform an intermediate record
Looking ahead: In the MapReduce framework these Transforms are referred to as Reducers
Result HDFS Files or S3 Objects

Processing Big Data

- The only feasible approach to tackling large data problems today is to divide and conquer, a fundamental concept in computer science
- The basic idea is to partition a large problem into smaller sub-problems.
- To the extent that the sub-problems are independent, they can be tackled in parallel by different workers
  - Threads in a processor core, cores in a multi-core processor
  - Multiple processors in a machine, or many machines in a cluster
- Intermediate results from each individual worker are then combined to yield the final output
- The general principles behind divide-and-conquer algorithms are broadly applicable to a wide range of problems in many different application domains
- However, the details of algorithm implementation are varied and complex

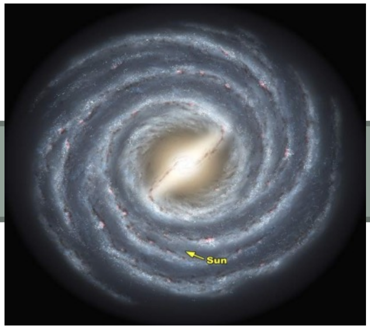
Processing Big Data
Challenges

- How do we break up a large problem into smaller tasks?
  - More specifically, how do we decompose the problem so that the smaller tasks can be executed in parallel?
- How do we assign tasks to workers distributed across a potentially large number of machines?
  - While keeping in mind that some workers are better suited to running some tasks than others, e.g., due to available resources, locality constraints, etc
- How do we ensure that the workers get the data they need?
- How do we coordinate synchronization among the different workers?
- How do we share partial results from one worker that is needed by another?
- How do we accomplish all of the above in the face of software errors and hardware faults?

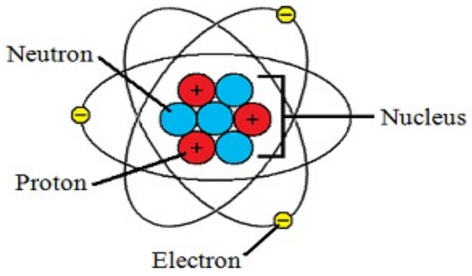
MapReduce
An Execution Framework for Big Data Processing

- A programming model for expressing distributed computations on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers
- Originally developed by Google and built on well-known principles in parallel and distributed processing
- Has since enjoyed widespread adoption via an open-source implementation in Hadoop
- Addresses the challenges of distributed programming by providing an abstraction that isolates the developer from system-level details (e.g., locking of data structures, data starvation issues in the processing pipeline, etc.)
- The programming model specifies simple and well-defined interfaces between a small number of components, and therefore is easy for the programmer to reason about.

MapReduce Concepts



Start with a large volume of data in one arrangement



Break the data down into its smallest parts

Process those small parts in parallel

And group them into a new and informative whole

MapReduce Concepts

- MapReduce can refer to three distinct but related ideas
- First, MapReduce is a programming model, which is the sense discussed above
- Second, MapReduce can refer to the execution framework (i.e., the “runtime”) that coordinates the execution of programs written in this particular style
- Finally, MapReduce can refer to the software implementation of the programming model and the execution framework

MapReduce Concepts

- Large-data processing by definition requires bringing data and code together for computation to occur—no small feat for datasets that are terabytes and perhaps petabytes in size
- MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a scalable, robust, and efficient manner
- Instead of moving large amounts of data around, it is far more efficient, if possible, to move the code to the data
- This is operationally realized by spreading data across the local disks of nodes in a cluster and running processes on nodes that hold the data
- The complex task of managing storage in such a processing environment is handled by the Hadoop Distributed File System that sits underneath MapReduce.

MapReduce Concepts

- However, as anyone who has taken an introductory computer science course knows, abstractions manage complexity by hiding details and presenting well-defined behaviors to users of those abstractions.
- They, inevitably, are imperfect—making certain tasks easier but others more difficult, and sometimes, impossible
- This critique applies to MapReduce: it makes certain large-data problems easier, but suffers from limitations as well.
- This means that MapReduce is not the final word, but rather the first in a new class of programming models that will allow us to more effectively organize computations at a massive scale.

MapReduce Concepts

- Viewed from a slightly different angle, MapReduce codifies a generic “recipe” for processing large datasets that consists of two stages
  - Map—First a user-specified computation is applied over all input records in a dataset. These operations occur in parallel and yield intermediate output
  - Reduce—Next, the output is then aggregated by another user-specified computation
- The programmer defines these two types of computations, and the execution framework coordinates the actual processing
- Such a two-stage processing structure may appear to be very restrictive, many interesting algorithms can be expressed quite concisely…
- Especially if one decomposes complex algorithms into a sequence of MapReduce jobs.

Developing Some MapReduce Intuition

- Most explanations of how the MapReduce framework supports the development of parallel programming algorithms start with…
  - Some mathematical representation
  - Or a diagram so complex one cannot develop any sense about what is happening
- We are going to take something of a different approach and start by developing our intuition about how the Map Reduce framework functions
- And then, once we have some of the fundamentals understood move on to more detail

Developing Some MapReduce Intuition

- Let's think through how we might take advantage of the following Hadoop capabilities we have encountered to process large volumes of data
- We have a cluster of computing nodes each having some amount of CPUs, RAM and storage
- We have a distributed file systems that takes a large file, decomposes it into blocks, and then stores them across the nodes of our cluster
- And we have a means to launch applications comprised of multiple tasks and distribute them across the cluster

Developing Some MapReduce Intuition

- So our first thought might be: let’s run an application with a task on each node where some block of a file I want to process is located
- That is a good start; we can execute searches and sorts or filtering operations on each block in parallel

Developing Some MapReduce Intuition

Challenges

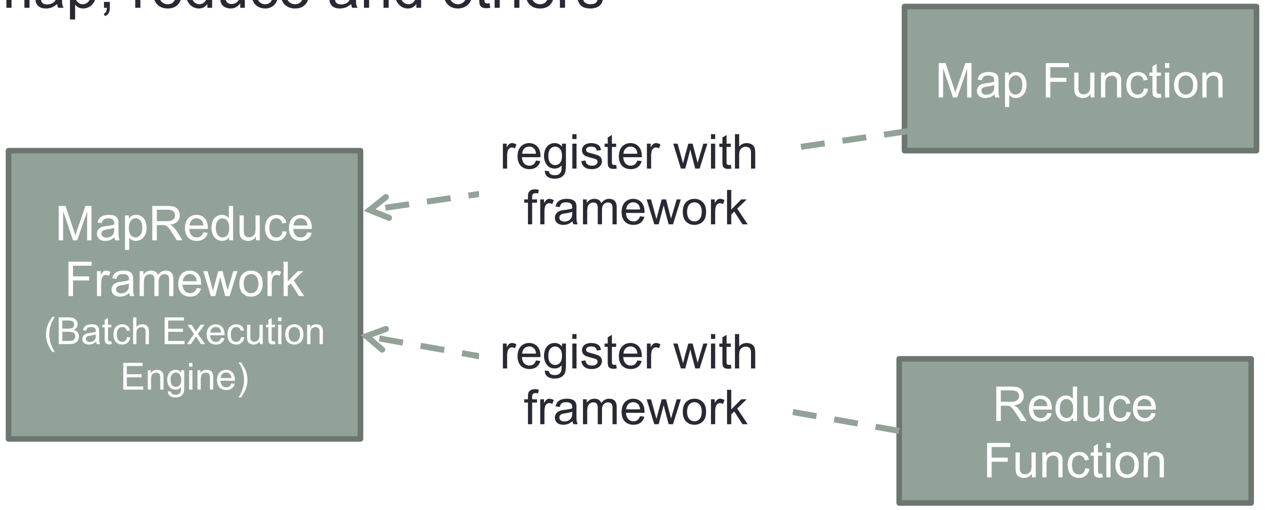
- We need to make sure that our application is fault tolerant
  - That is tolerant of node, software and network failures that might happen in a distributed environment
- We also need to somehow merge and finish processing together the output of the tasks that have executed on each node
  - That is support data sharing and synchronization across multiple application tasks
- Our file was decomposed into blocks without regard to record boundaries
  - But what we really want to do is get each record from a block and process it
  - But because of the naïve way the file is stored in blocks some records may cross block boundaries
- All these details and more are complicated, and I want things simple
  - I want to use a framework that I can customize by just a small amount to do what I want
  - This framework should work well with YARN and other aspects of the Hadoop environment including security.

Developing Some MapReduce Intuition

- It is by this sort of thinking we arrive at the Hadoop MapReduce execution engine
- It is a framework which addresses the above challenges and more
- It does so in a way that makes it comparatively simple to write parallel processing algorithms
- But, as we shall see, as with all engineered systems we have to accept some tradeoffs to take advantage of this framework

Developing Some MapReduce Intuition
A Little About MapReduce as a Framework

- Most frameworks have a core of services and offer hooks for you to add custom code
- These hooks are generally member functions that must accept specified input types and return specified output types
- Framework users code this functions and register them with the framework
- The MapReduce framework supports hooks for functions such as map, reduce and others



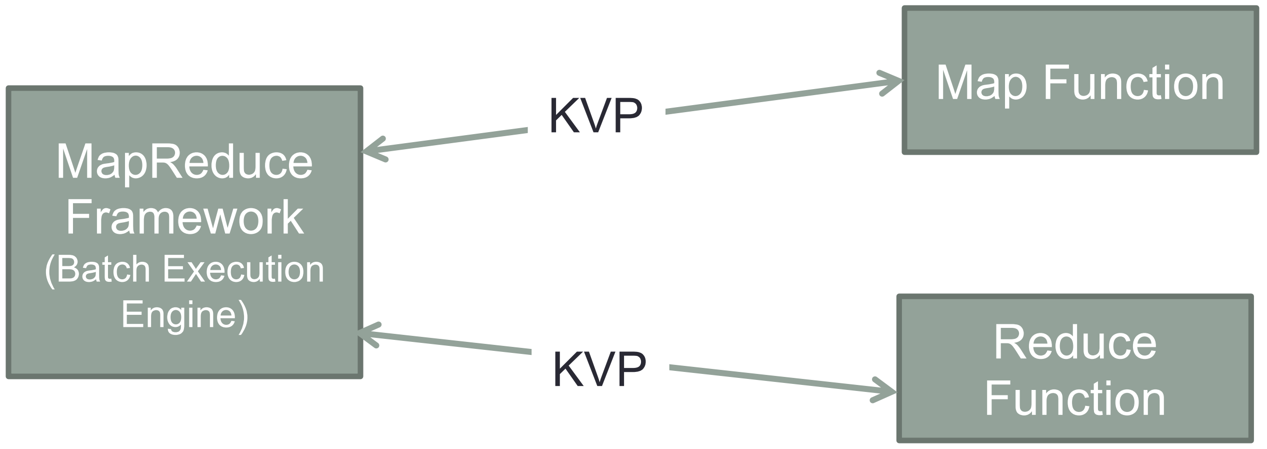
Developing Some MapReduce Intuition
A Little About MapReduce as a Framework

- Most frameworks provide data to user member functions and otherwise manage it
- And this data may be of any type useful to the framework user
- But this means the framework must have a means to hold and pass data of unknown types
- The way most frameworks handle this is to place user data types is a “wrapper”

Developing Some MapReduce Intuition

A Little About MapReduce as a Framework

- So the framework uses the wrapper means to handle and move data
- To hold a wide range of user data types, this wrapper must be very generic
- The MapReduce framework wraps data in key value pairs
- The pairs are used to pass user data types between the hook functions and the framework
- Part of the design of MapReduce algorithms involves imposing this key value structure on arbitrary datasets



Developing Some MapReduce Intuition

A Little About MapReduce as a Framework

- A set of two data items: a **key**, which is a unique identifier for some item of data, and the **value**, which is either the data that is identified or a pointer to the location of that data.
- The key and value can be either simple or complex types. For example, here are simple key value pairs
  - <“hello”, 5>
  - <”hello”, “there”>
- Key value pairs can also be complex
  - <”hello”, list(1, 2, 3, 4, 5)> or equivalently <”hello”, {1, 2, 3, 4, 5}>
  - <struct(“a”, 5), “hello”>

Developing Some MapReduce Intuition

A Little About MapReduce as a Framework

- Key-value pairs form the basic data structure in MapReduce.
- Keys and values may be primitives such as integers, floating point values, strings, and raw bytes, or they may be arbitrarily complex structures (lists, tuples, associative arrays, etc.).
- Part of the design of MapReduce algorithms involves imposing the key-value structure on arbitrary datasets

Developing Some MapReduce Intuition
A Little About MapReduce as a Framework

- For a collection of web pages, keys may be URLs and values may be the actual HTML content.
- For a graph, keys may represent node ids and values may contain the adjacency lists of those nodes
- In some algorithms, input keys are not particularly meaningful and are simply ignored during processing,
- While in other cases input keys are used to uniquely identify a datum (such as a record id)

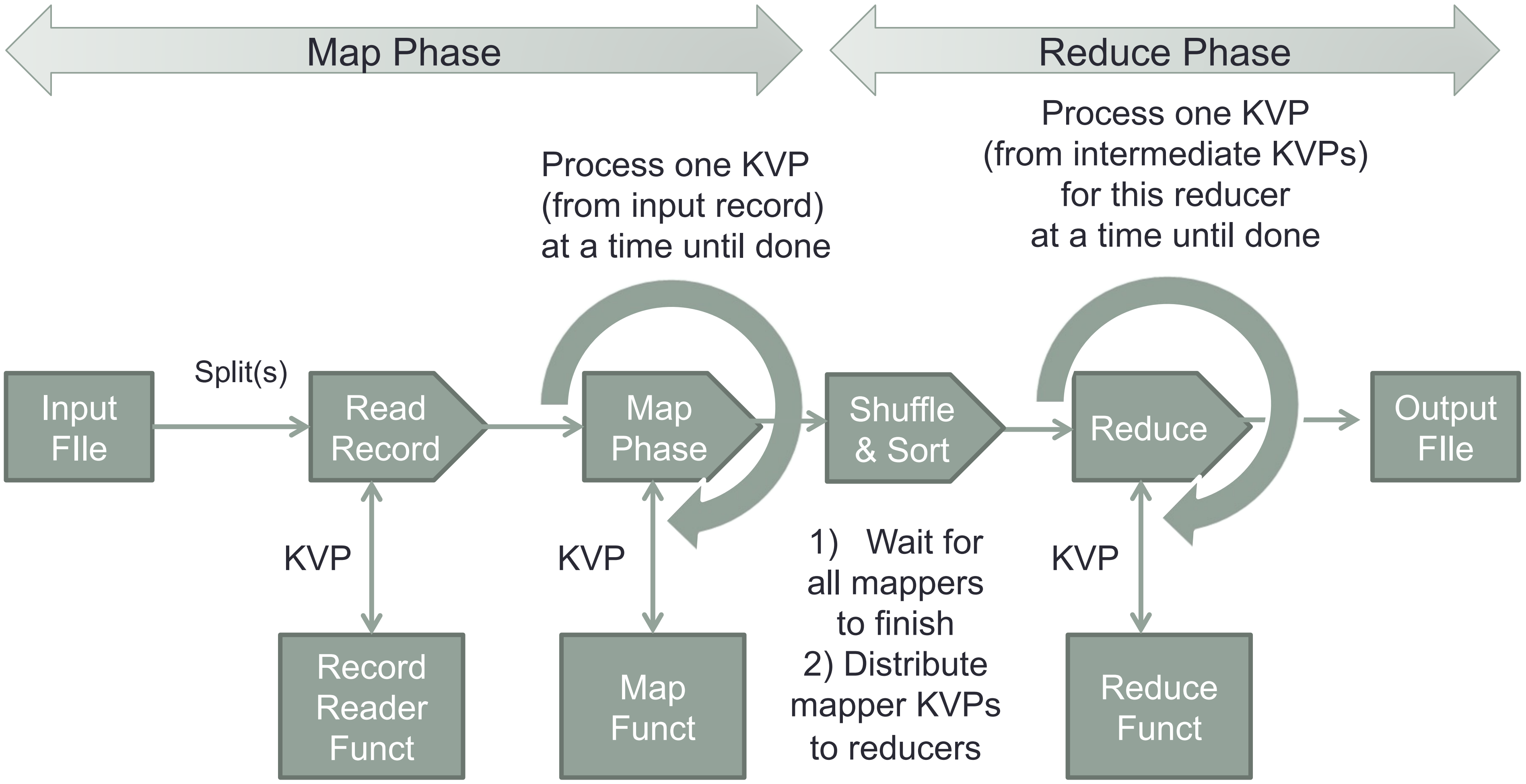
MapReduce Framework
Workflow

- MapReduce splits the input data into independent splits often aligned with HDFS file blocks
- Each of these splits is processed by map functions in a completely parallel manner
- But what if some of the output from a map function should be grouped for further processing with the output of some other map function
- Recall each map functions executes on a different node with a separate file system
- So each map function also indicates which subset of the data it outputs should be processed as a unique group

MapReduce Framework
Workflow

- The MapReduce framework collects the map output from each node for a group
- The data of the same group from all map nodes is then assembled together then presented to a reduce function
- The reduce function processes data from multiple nodes which should, as a final step, be processed together
- The output of each reduce function is saved to an HDFS file (or S3 object)

MapReduce Framework
Conceptual Landscape


Map Phase
Reduce Phase
Process one KVP (from input record) at a time until done
Process one KVP (from intermediate KVPs) for this reducer at a time until done
Input FIle
Split(s)
Read Record
Map Phase
Shuffle & Sort
Reduce
Output FIle
KVP
KVP
KVP
Record Reader Funct
Map Funct
Reduce Funct
1) Wait for all mappers to finish
2) Distribute mapper KVPs to reducers

MapReduce Framework

Operational View

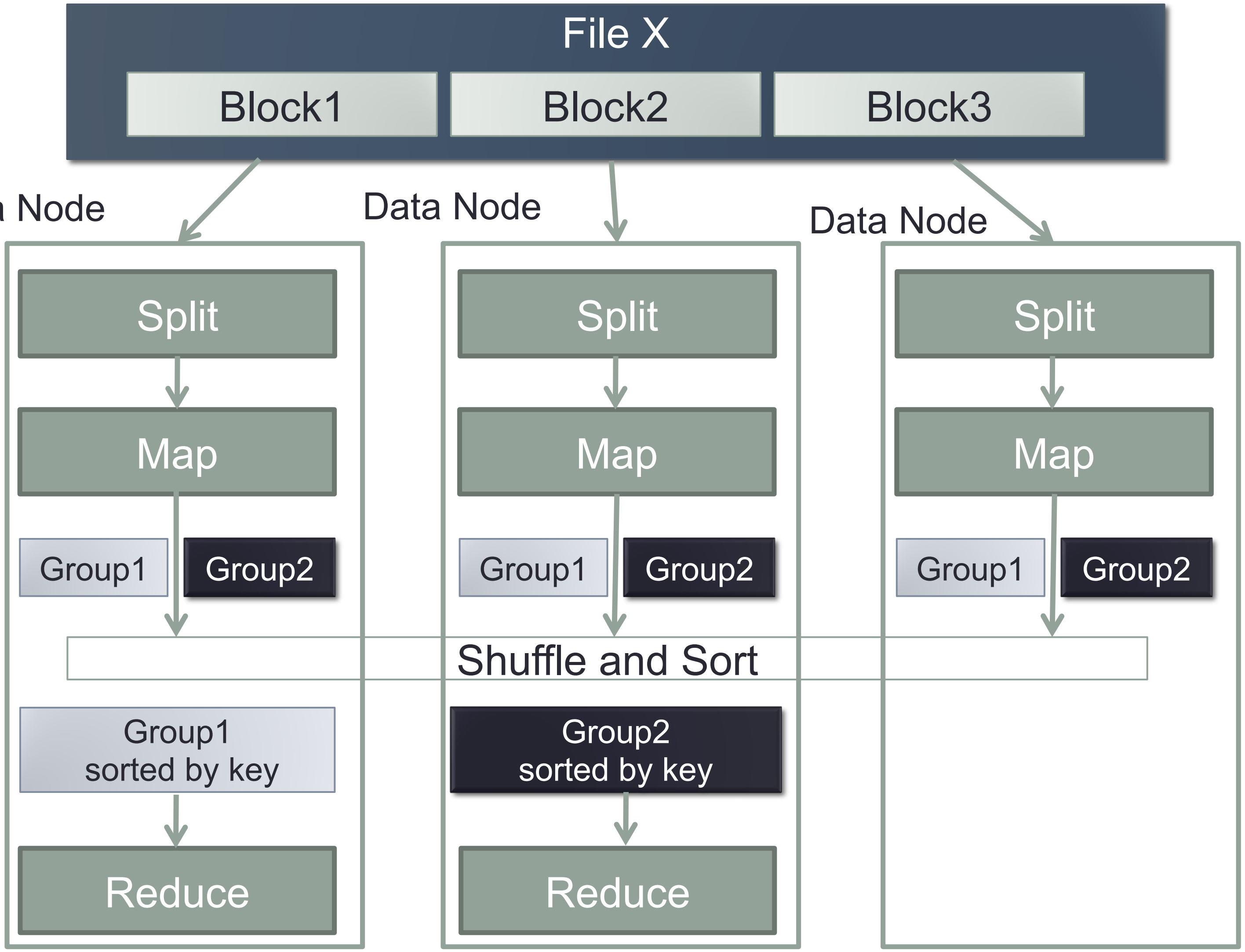
Split blocks at record boundaries

Process the split block

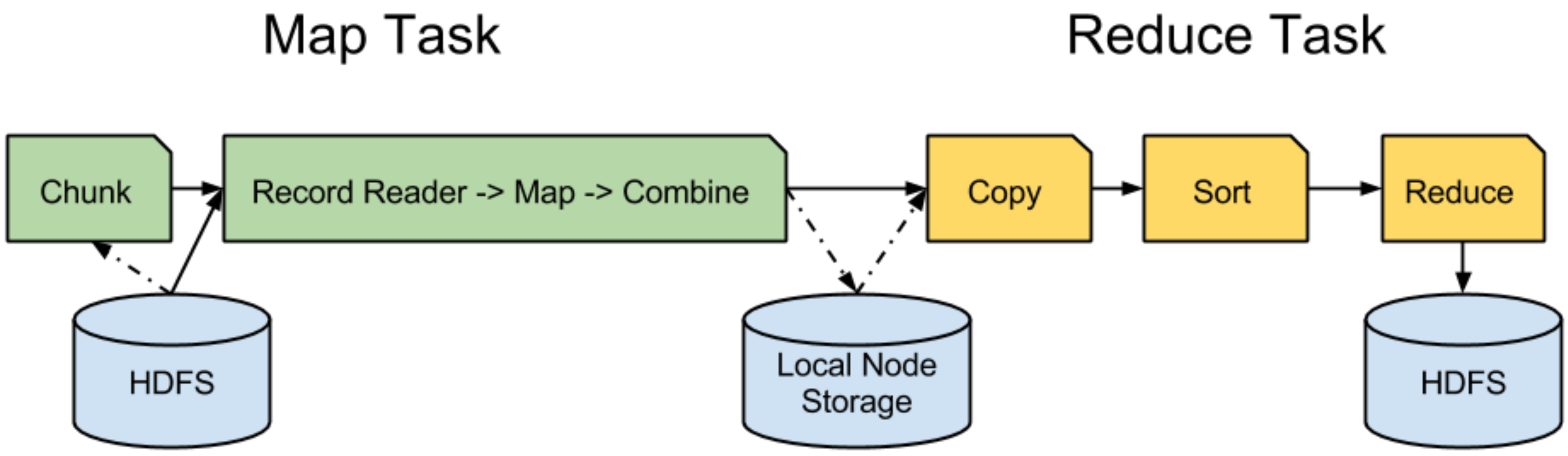
Place data into groups

Combine and sort like groups by key

Process the grouped data



MapReduce Framework
Dataflow View



MapReduce Framework
General Characteristics

- Maintains a separation of *what* computations are to be performed and *how* those computations are actually carried out on a cluster of machines.
- The first is under the control of the programmer, while the second is exclusively the responsibility of the execution framework
- The advantage is that the execution framework only needs to be designed once and verified for correctness
- As long as the developer expresses computations in the programming model, code is guaranteed to behave as expected.
- The upshot is that the developer is freed from having to worry about system-level details and can instead focus on algorithm or application design
  - No more debugging race conditions and addressing lock contention

MapReduce Framework
Libraries

- Java
  - Maximum flexibility
  - Fastest performance
  - Native to Hadoop
  - Most difficult to write
- Hadoop Streaming
  - A utility that comes with the Hadoop distribution
  - Allows you to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer
  - Transparent communication with Hadoop though stdin/stdout

MapReduce Framework
Libraries

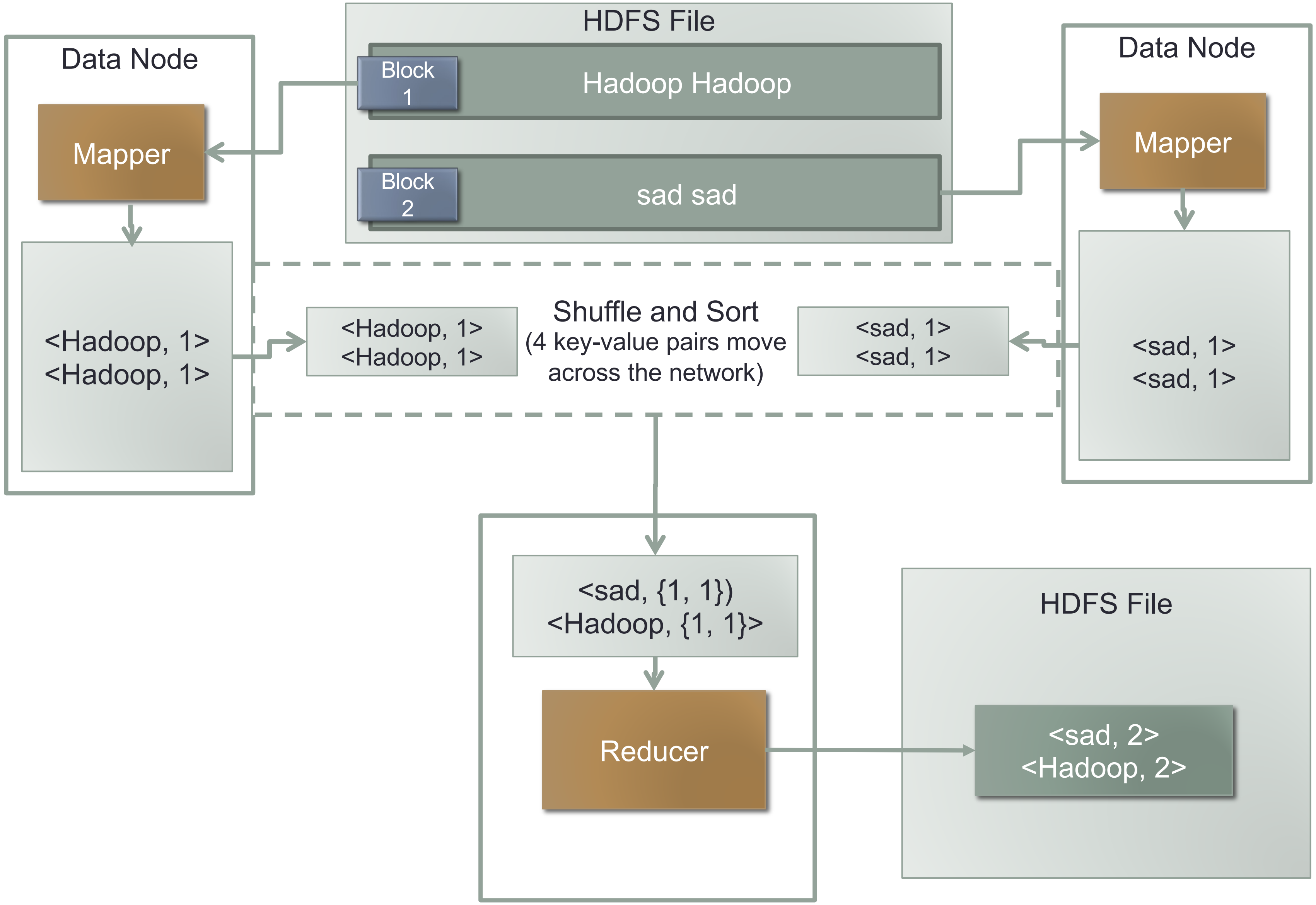
- mrjob
  - Write MapReduce jobs in Python!
  - Wraps “Hadoop Streaming” in Python
  - Open sourced and maintained by Yelp
  - Well documented
  - Can run locally, in Amazon EMR, or Hadoop

- We will use mrjob for our class work

MapReduce Example

- WordCount is a simple program which counts the number of occurrences of each word in a given text input data set.
- WordCount fits very well with the MapReduce model making it a great example to understand the Hadoop Map/Reduce programming style.
- Our implementation consists of three main parts:
  - Mapper function
  - Reducer function
  - Main program (hooking our functions into the framework)

MapReduce Example



WordCount
Driver

- The first chunk of code we'll look at is the driver.
- The driver takes all of the components that we've built for our MapReduce job and pieces them together to be submitted to execution
- This code is usually pretty generic and considered "boiler plate."
- You'll find that in all of our patterns the driver stays the same for the most part

WordCount
Driver

```
from mrjob.job import MRJob

class MRClassName(MRJob):

    def mapper(self, _, value):
     // some mapper logic

    def reducer(self, key, values):
        // some reducer logic

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

This (_) is a Python idiom to ignore the function argument

"_" is the key of the input key value pair

"value" is the value of the input key value pair

"key" is the key of the input key value pair

"values" is an iterable to the collection value of the input key value pair

WordCount
Mapper

- Next is the mapper code that parses and prepares the text
- The text string is split up into a list of words
- Then the intermediate key produced is the word and the value produced is simply "1."
- This means we've seen this word once.
- Even if we see the same word twice in one line, we'll output the word and "1" twice and it'll be taken care of in the end
- Eventually, all of these ones will be summed together into the global count of that word

WordCount
Mapper

- The mapper is where we'll see most of the work done.
- We don't care about the key of the input in this case which is the file offset of the next record read
- The input value (and key) are strings
- If you need to process the value (or part of it) as a float, integer or some other type you need to do the conversion yourself
- Our output key and value must also be strings

WordCount
Mapper

```
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)
```

WordCount
Reducer

- Next comes the reducer code, which is relatively simple
- The reduce function gets called once per key grouping, in this case each word
- We’ll iterate through the values, which will be numbers, and take a running sum
- The final value of this running sum will be the sum of the ones
- As in the mapper the key is a string

WordCount
Reducer

- The reduce function has a different signature from map, though
- It gives you an Iterator over values instead of just a single value
- This is because you are now iterating over all values that have that key instead of just one at a time
- The key is very important in the reducer of pretty much every MapReduce job, unlike the input key in the map
- Each reducer will create one file
    - so if you want to coalesce them together you'll have to write a post-processing step to concatenate them

WordCount
Reducer

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

Canonical Word Count

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

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

Canonical Word Count

Read from an HDFS file or S3 object split

```
from mrjob.job import MRJob
import re

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

<some-offset, The quick brown fox jumps over the lazy dog>

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

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


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

One input key value pair can result in zero or more output key value pairs

<the, 1>
<quick, 1>
<brown, 1>
<fox, 1>
<jumps, 1>
<over, 1>
<the, 1>
<lazy, 1>
<dog, 1>

Canonical Word Count

```
from mrjob.job import MRJob
import re

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

class MRWordFreqCount(MRJob):

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

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

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

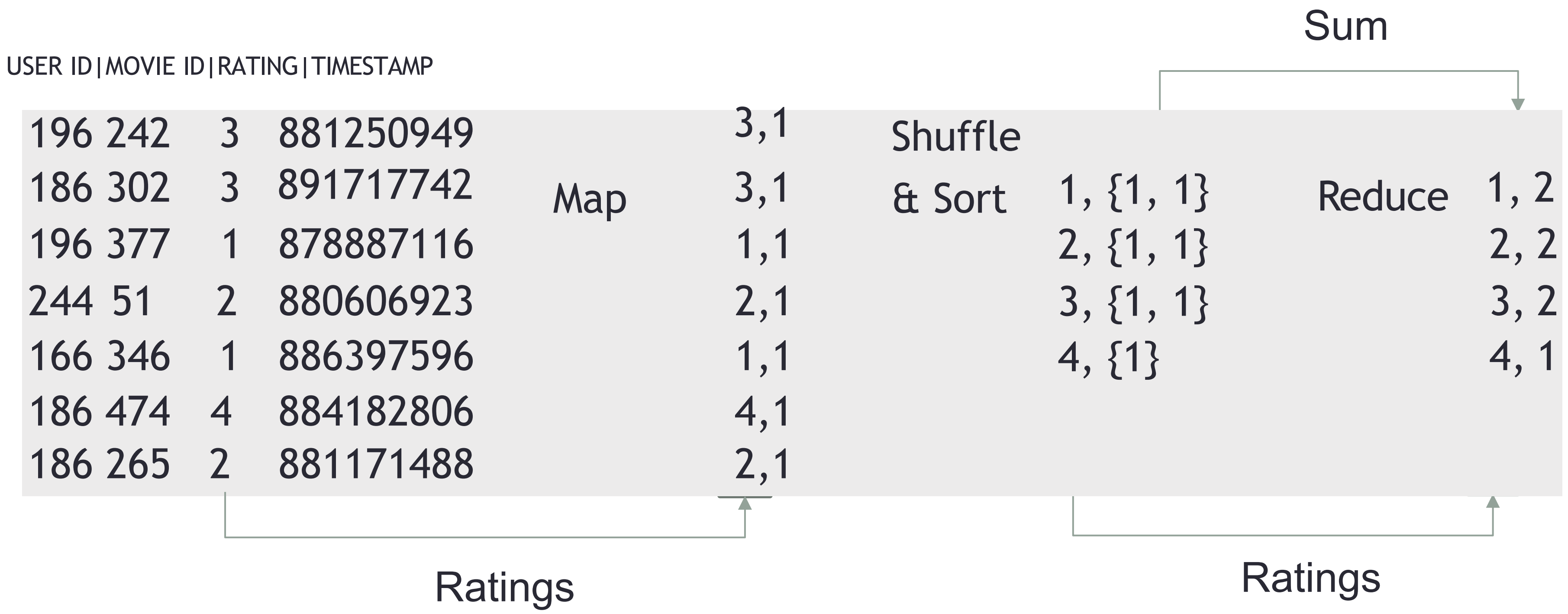
Created by the shuffle and sort phase grouping Mapper output

<dog, {1, 1, 1, 1, 1, 1}>

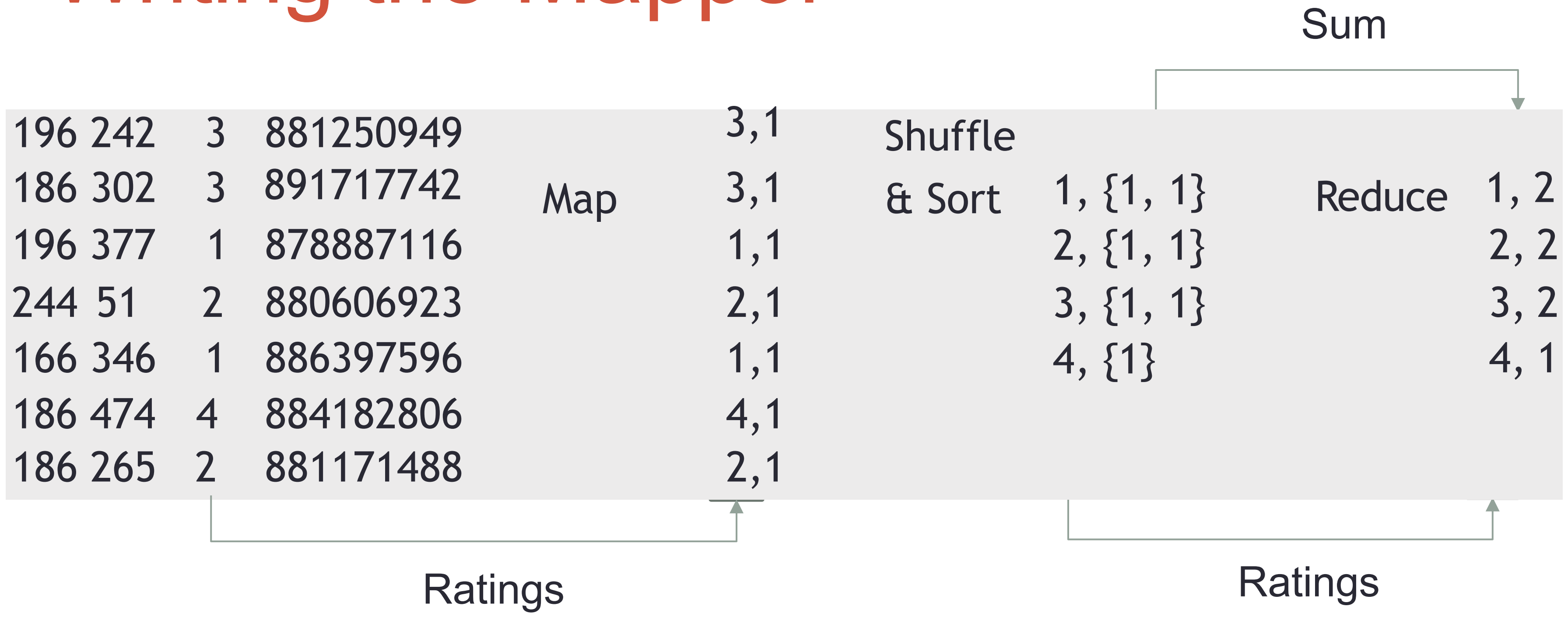
<dog, 6>

Another Example: How many of each movie rating exist?

- MAP each input line to (rating, 1)
- REDUCE each rating with the sum of all the 1's



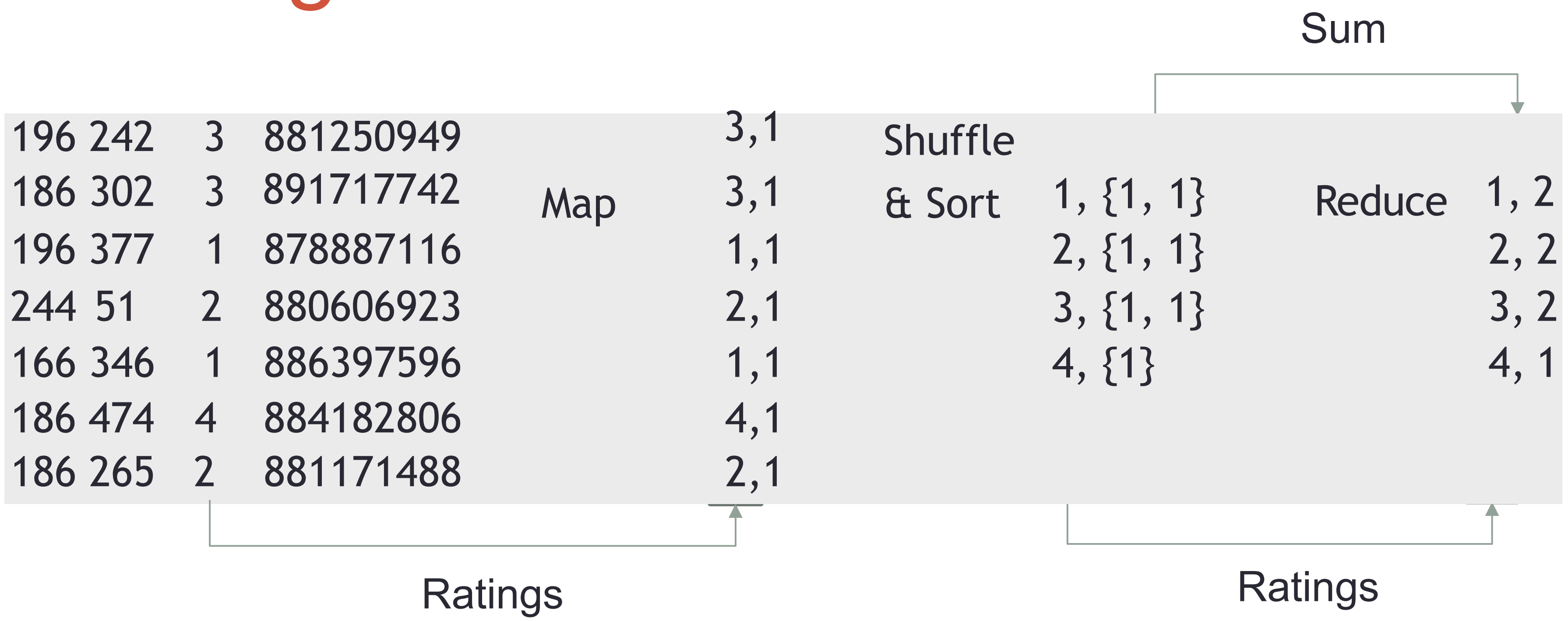
Writing the Mapper



```
196 242    3    881250949                3,1      Shuffle
186 302    3    891717742      Map       3,1      & Sort    1, {1, 1}      Reduce    1, 2
196 377    1    878887116                1,1                2, {1, 1}                2, 2
244 51     2    880606923                2,1                3, {1, 1}                3, 2
166 346    1    886397596                1,1                4, {1}                   4, 1
186 474    4    884182806                4,1
186 265    2    881171488                2,1
```

```python
def mapper_get_ratings(self, _, line):
    (userID, movieID, rating, timestamp) = line.split('\t')
    yield rating, 1
```

Writing the Reducer



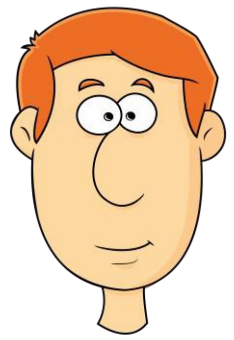
```
def reducer_count_ratings(self, key, values):
    yield key, sum(values)
```

Putting it all together

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class RatingsBreakdown(MRJob):
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_ratings,
                   reducer=self.reducer_count_ratings)
        ]

    def mapper_get_ratings(self, _, line):
        (userID, movieID, rating, timestamp) = line.split('\t')
        yield rating, 1

    def reducer_count_ratings(self, key, values):
        yield key, sum(values)

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

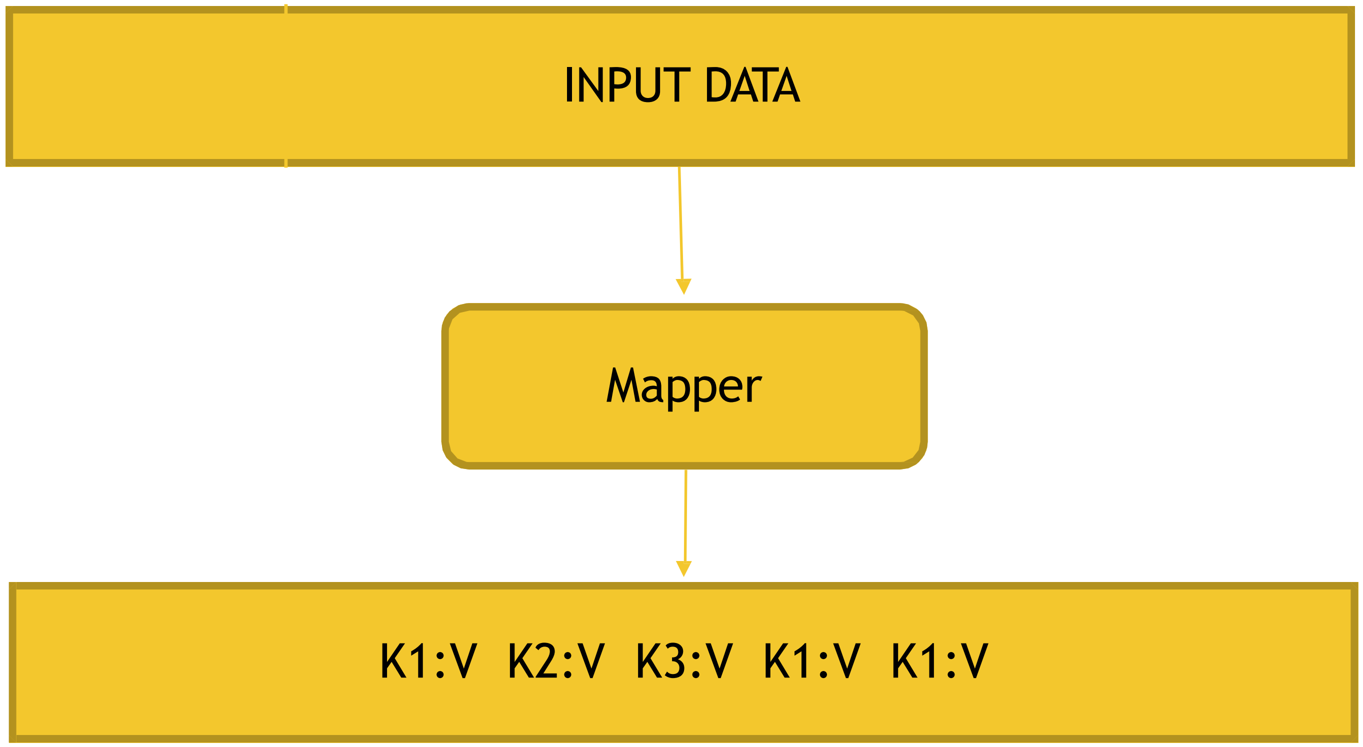
Let’s illustrate with another example

- How many movies did each user rate in the MovieLens data set?



How MapReduce Works: Mapping

- The MAPPER converts raw source data into **key/value** pairs



Example: MovieLens Data (u.data file)

USER ID|MOVIE ID|RATING|TIMESTAMP

196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

Map users to movies they watched

USER ID	MOVIE ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

<fileOffset1, “196 242 3 881250949”>

…

<fileOffsetN, “186 265 2 881171488”>

Mapper

<user-id, movie-id>

<196, 242> <186, 302> <196, 377> <244, 51> <166, 346> <186, 274> < 186, 265>

Extract and Organize What We Care About

<user-id, movie-id>

<196, 242> <186, 302> <196, 377> <244, 51> <166, 346> <186, 274> < 186, 265>

MapReduce Sorts and Groups the Mapped Data (“Shuffle and Sort”)

<user-id, movie-id>

<196, 242> <186, 302> <196, 377> <244, 51> <166, 346> <186, 274> < 186, 265>

Shuffle & Sort

<user-id, {movie-id-1, movie-id-2, …}>

<166, {346}> <186, {302,274,265}> <196, {242,377}> <244, {51}>

After shuffle and sort, all values associated with the same key are grouped together into unordered collections

The key-value pairs are presented to reducers in sorted order by their keys

The REDUCER Processes Each Key's Values

<166, {346}> <186, {302,274,265}> <196, {242,377}> <244, {51}>

↓

Reducer output <key, len(value)>

↓ <user-id, count-of-movies-reviewed-by-this-user>

<166, 1> <186, 3> <196, 2> <244,1>

Putting it All Together

USER ID	MOVIE ID	RATING	TIMESTAMP
196	242	3	881250949
186	302	3	891717742
196	377	1	878887116
244	51	2	880606923
166	346	1	886397596
186	474	4	884182806
186	265	2	881171488

↓

MAPPER

↓

196:242 186:302 196:377 244:51 166:346 186:274 186:265

↓

SHUFFLE AND SORT

↓

166:346 186:302,274,265 196:242,377 244:51

↓

REDUCER

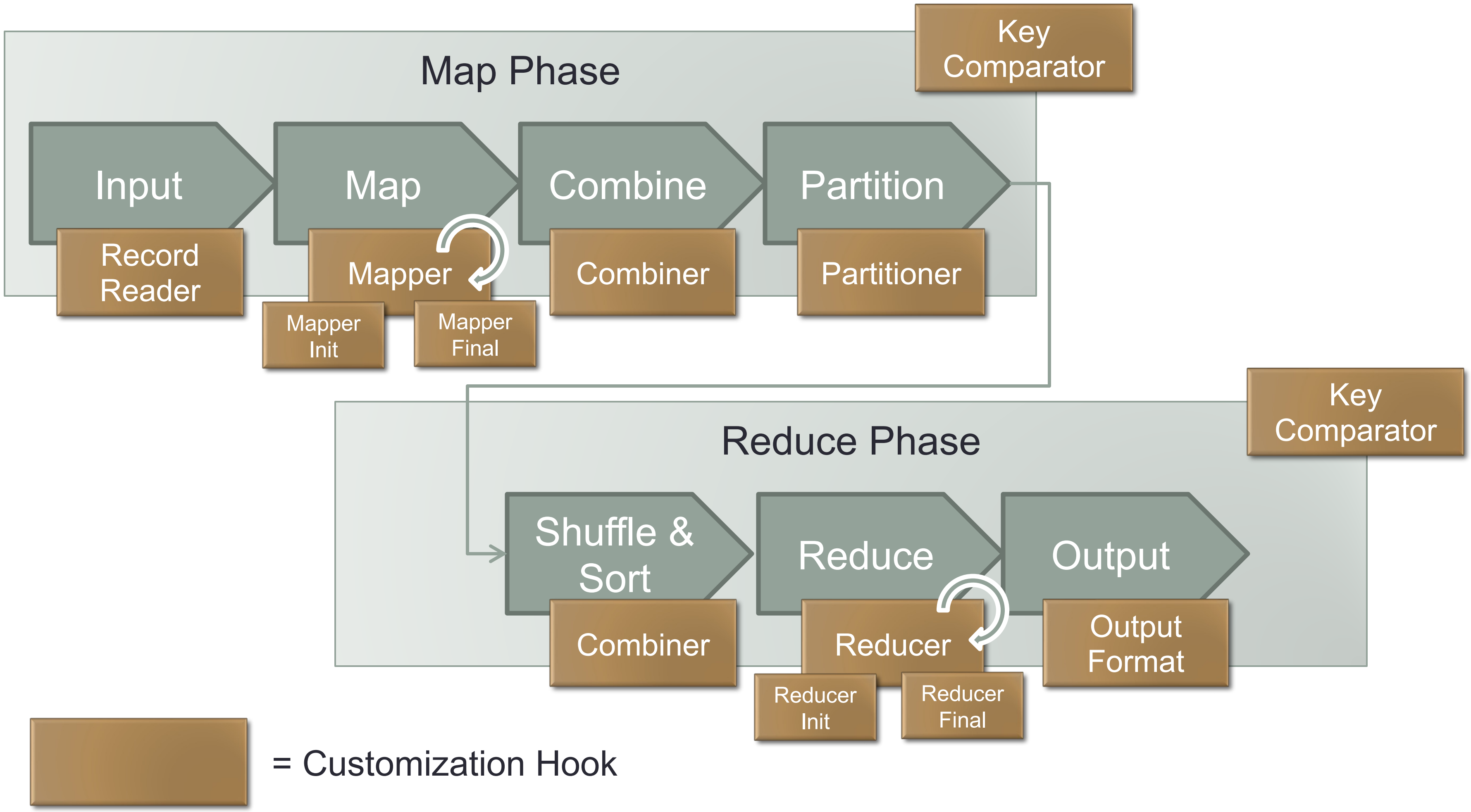
↓

166:1 186:3 196:2 244:1

MapReduce
A More Complete Perspective

- 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
- Each map task in Hadoop is broken into the following phases: *record reader*, *mapper*, *combiner*, and *partitioner*.
- The output of the map tasks, called the intermediate keys and values, are sent to the reducers.
- The reduce tasks are broken into the following phases *shuffle*, *sort*, *reducer*, and *output format*.
- The nodes in which the map tasks run are optimally on the nodes in which the data rests.
- This way, the data typically does not have to move over the network and can be computed on the local machine.

MapReduce
A More Complete Perspective

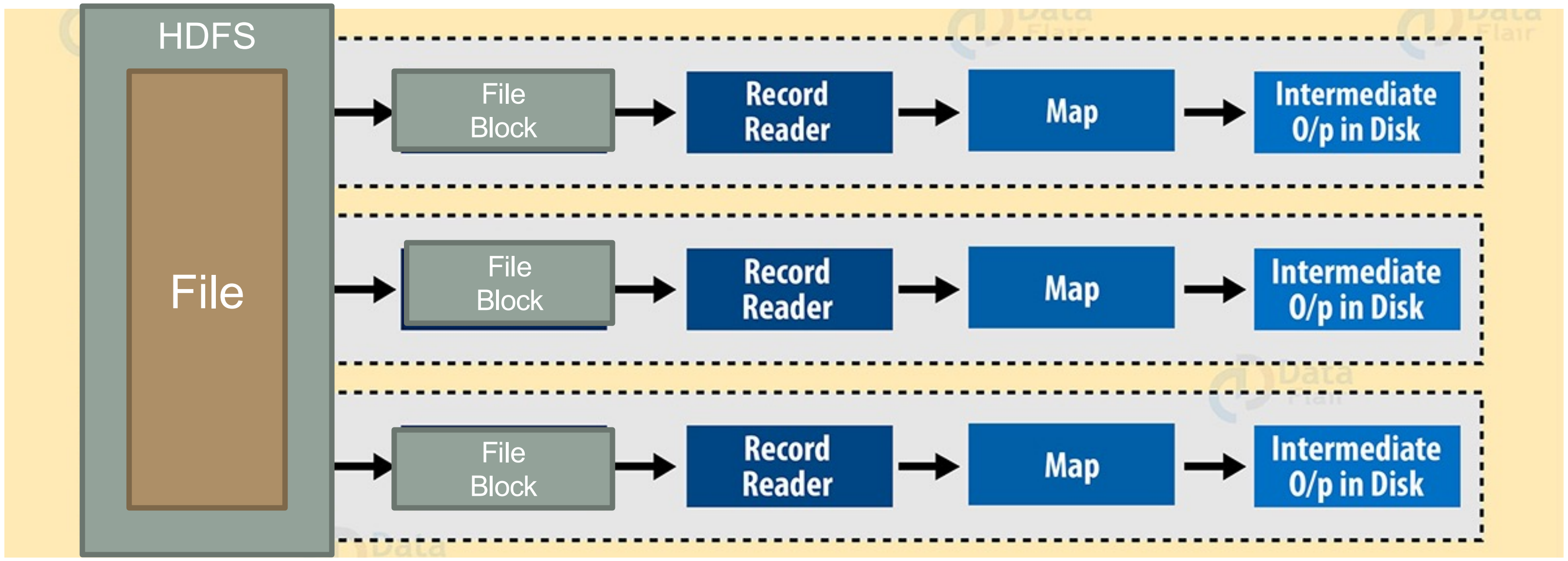


Input Format
Record Reader

- 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.

Mapper

- 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 pairs, called the intermediate pairs.
- The decision of what is the key and value here is not arbitrary and is very important to what the MapReduce job is accomplishing
- The key is what the data will be grouped on and the value is the information pertinent to the analysis in the reducer



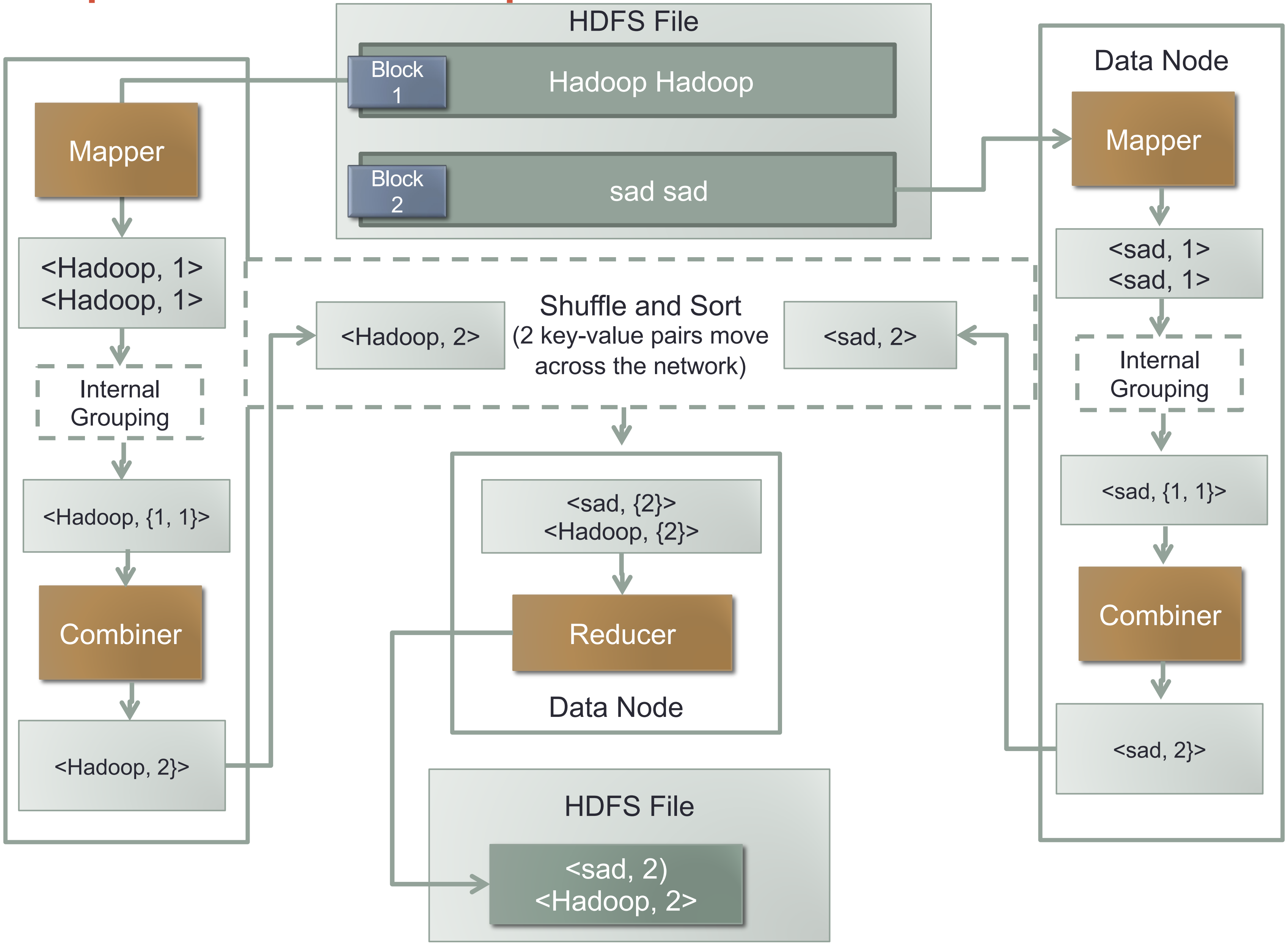
Combiner

- A Combiner does local aggregation of mapper output
- This helps to minimize data transfer (network overhead) between mapper and reducer
- It also decreases the amount of data that needed to be processed by the reducer.
- The result is an increase to MapReduce job performance
- But the execution of combiner is not guaranteed
  - Hadoop may or may not execute a combiner
  - Also if required it may execute it more than 1 time
  - **So... combiner can be be executed zero, one or many times**

Combiner

- 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.
- For example, because the count of an aggregation is the sum of the counts of each part, you can produce an intermediate count and then sum those intermediate counts for the final result.
- In many situations, this significantly reduces the amount of data that has to move over the network.
- Sending (hello world, 3) requires fewer bytes than sending (hello world, 1) three times over the network.
- A combiner is not guaranteed to execute, so it is an optional part of the overall algorithm

MapReduce Example



Combiner
Word Count Example

```
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): # optimization: sum the words we've seen so far
   yield (word, sum(counts))

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

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

A combiner only aggregates data output by one mapper task operating on one split

The output of the combiner must be the same format as that of the Mapper
<key, value>
AND NOT
<key, {value1, value2, …}>

What Determines if a Combiner Will Execute?

- Each mapper has a memory buffer for sorting, and by default its size is 100MB
- When the utilization of the memory buffer reaches some threshold (80% full), a thread will start to write the buffer contents to disk
- The writing process is also called “spilling,” and the files created are called spill files and are each 100MB in size by default
- By default a combiner runs if there are more than 3 spill files written to the disk
- If there are only one or two spills, the potential reduction in map output size is not worth the overhead in invoking the combiner, so it is not run

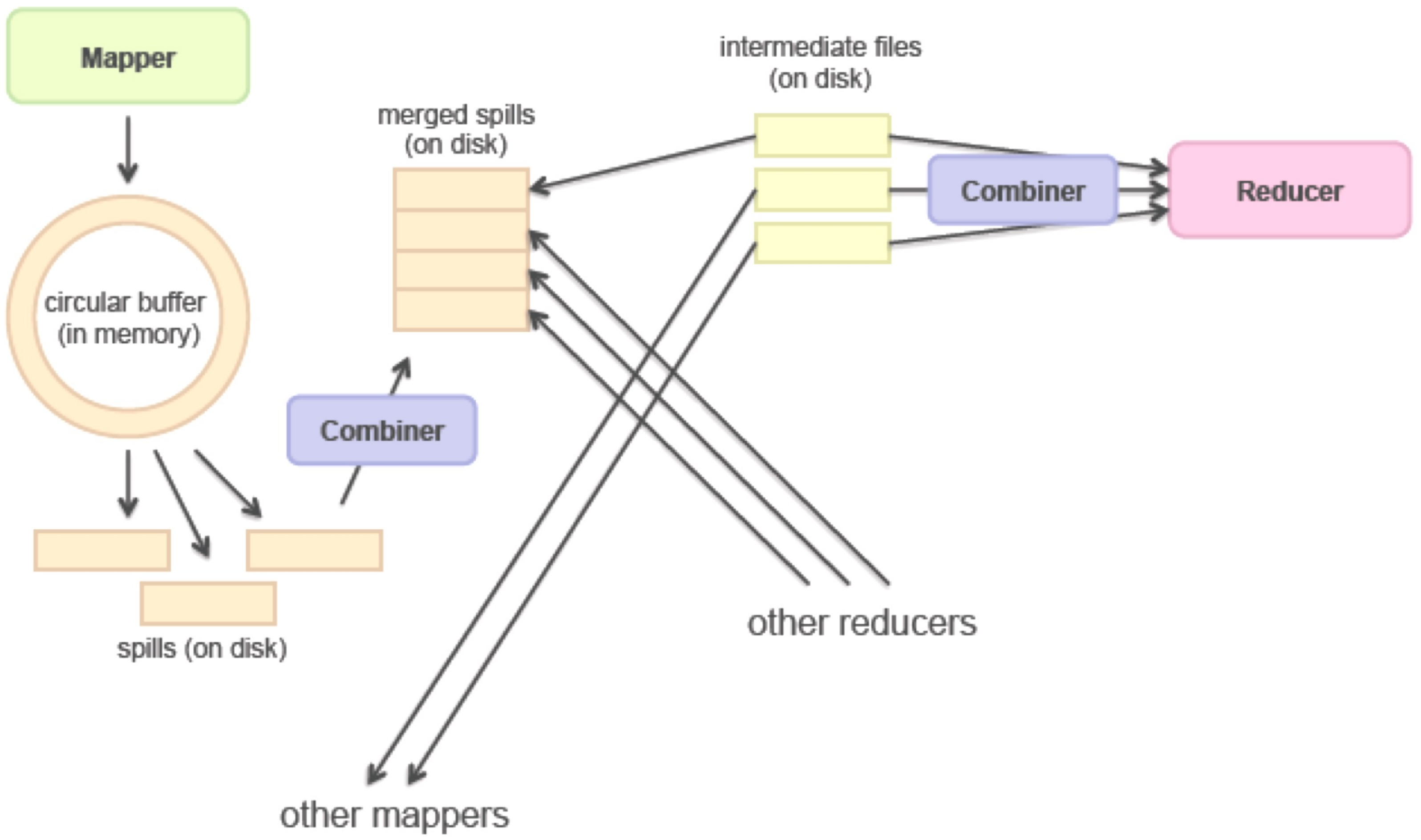
Partitioner

- 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 interrogates the object for its hash code, which is typically an md5sum.
- Then, the partitioner performs a modulus operation by the number of reducers: key.hashCode() % (number of reducers)
- This 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, and will be in some more advanced patterns, such as sorting
- However, changing the partitioner is rarely necessary.
- The partitioned data is written to the local file system for each map task and waits to be pulled by its respective reducer.

Shuffle and Sort

- Probably the most complex aspect of MapReduce!
- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are "spilled" to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs here
- Reduce side
  - First, *map outputs are copied over to reducer machine*
  - *"Sort" is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs here*
  - Final merge pass goes directly into reducer

Shuffle and Sort



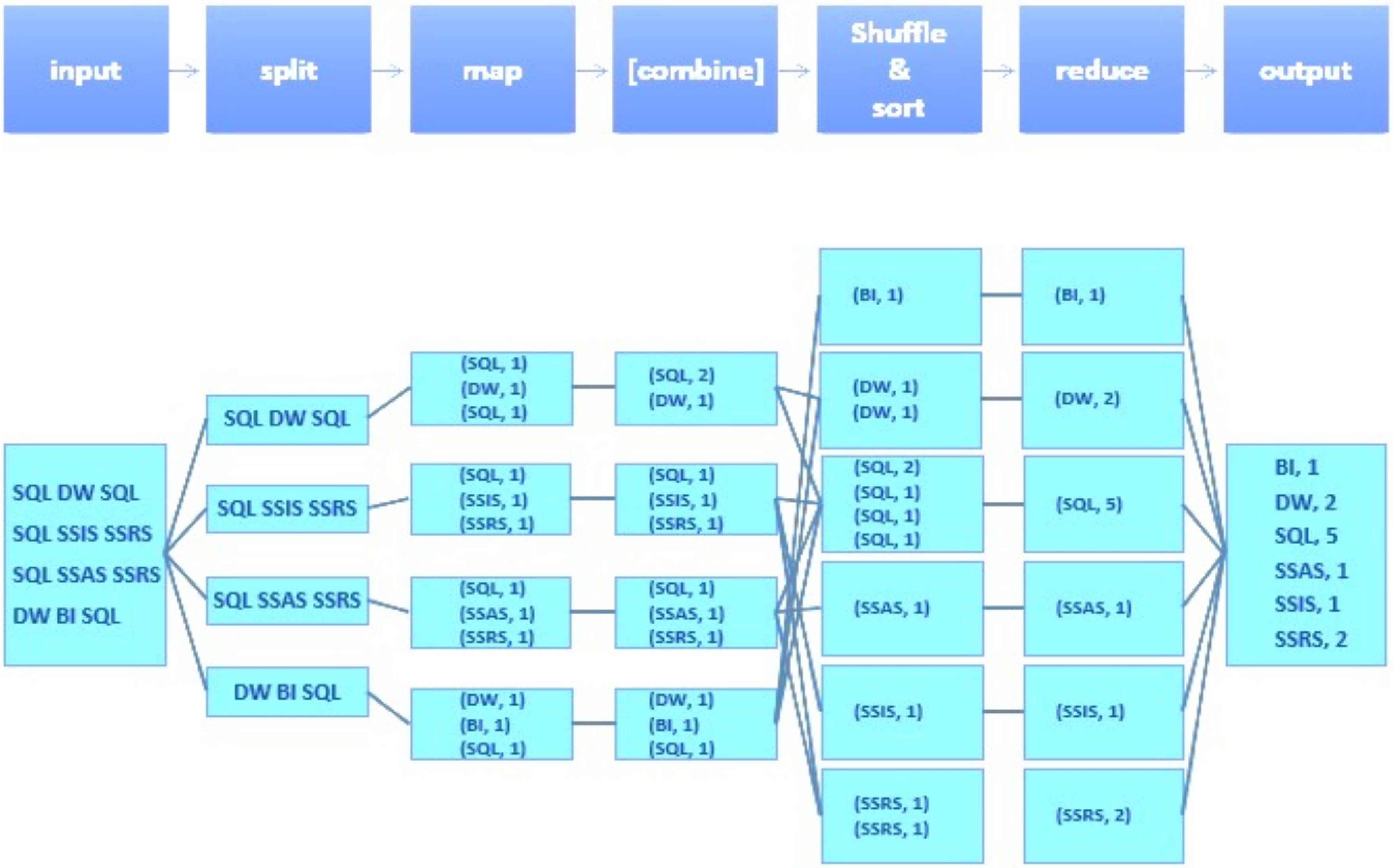
Reducer

- 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.
- This phase is not customizable and the framework handles everything automatically.
- The only control a developer has is how the keys are sorted and grouped by specifying a custom Comparator object.

Reducer

- 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.
- A wide range of processing can happen in this function, as we'll see in many of our patterns.
- The data can be aggregated, filtered, and combined in a number of ways.
- Once the reduce function is done, it sends zero or more key/value pair to the final step, the output format.
- Like the map function, the reduce function will change from job to job since it is a core piece of logic in the solution.

Word Count Full MapReduce Flow



Compute the Mean

```
class Mapper
    method Map(string $t$, integer $r$)
        Emit(string $t$, integer $r$)

class Reducer
    method Reduce(string $t$, integers $[r_1, r_2, \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all integer $r \in$ integers $[r_1, r_2, \ldots]$ do
            $sum \leftarrow sum + r$
            $cnt \leftarrow cnt + 1$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string $t$, integer $r_{avg}$)
```

Compute the Mean (Combiner)

```
class Mapper
    method Map(string $t$, integer $r$)
        Emit(string $t$, pair $(r, 1)$)

class Combiner
    method Combine(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \dots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \dots]$ do
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        Emit(string $t$, pair $(sum, cnt)$)

class Reducer
    method Reduce(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \dots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \dots]$ do
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string $t$, integer $r_{avg}$)
```

Version 1: Count the number of lines in a file

This passes the most data from Mappers to Reducers

```
from mrjob.job import MRJob

class MRCountLines(MRJob):

    def mapper(self, _, line):
        yield “Line”, 1

    def reducer(self, key, values):
        yield key, sum(values)

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

The mapper is called once per record in the block with the key-value pair holding <offset-of-record-line in block, line/record>

For each input line/record the mapper outputs a key value pair of the form <“Line”, 1>

The reducer is called once per key-value pair holding <“Line”, {1, 1, 1, 1, 1, 1, 1, …}>

Sum up all the 1’s to give the total line count in a file

Version 2: Count the number of lines in a file

This uses mapper_init and mapper_file to decrease the amount of data passed to reducers

```
from mrjob.job import MRJob

class MRCountLines(MRJob):

    def mapper_init(self):
        self.num_lines = 0

    def mapper(self, _, line):
        self.num_lines += 1

    def mapper_final(self):
        yield None, self.num_lines

    def reducer(self, key, values):
        yield key, sum(values)

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

Executes one per mapper to set the block (partial) line counter to zero

The mapper is called once per record in the block with the key-value pair holding <offset-of-record-line in block, line/record>

Executes once per block line/record to increment the block (partial) line counter to zero

The reducer is called once per key-value pair holding <None, {number-of-lines-per-mapper}>

Executes once per key value pair to sum up the block (partial) line counts

Version 3: Count the number of lines in a file

This uses a combiner to to decrease the amount of data passed to reducers

```
from mrjob.job import MRJob

class MRCountLines(MRJob):

    def mapper(self, _, line):
        yield "Line", 1

    def combiner(self, key, values)
        yield key, sum(values)

    def reducer(self, key, values):
        yield key, sum(values)

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

The mapper is called once per record in the block with the key-value pair holding <offset-of-record-line in block, line/record>

For each input line/record the mapper outputs a key value pair of the form <"Line", 1>

The combiner is called with a key value pair of the form below just for a block <"Line", {1, 1, 1, 1, 1, 1, 1, …}>

The outputs a key value pair of the form <"Line", 1>

The reducer is called once per key-value pair holding <"Line", {1, 1, 1, 1, 1, 1, 1, …}>

Sum up all the 1's to give the total line count in a file

Mappers and Reducers
How Many?

- Picking the appropriate count of the map and reduce tasks for a job can radically change performance Increasing the number of tasks increases the framework overhead…
- But increases load balancing and lowers the cost of failures
- At one extreme is the 1 map and 1 reduce case where nothing is distributed
- The other extreme is to have 1,000,000 map and the 1,000,000 reduce case where framework runs out of resources

Mappers
How Many?

- An input split is a chunk of the input that is processed by a map task
  - Each map processes a single split
- Each split is divided into records, and the map processes each record—a key-value pair—in turn
- The MapReduce framework sets the number of the map tasks based on considerations of split size
- Except in special circumstances the number of mappers is just generally the following…
  - file-size / split-size

Mappers
How Many?

Pseudocode used:

```
num_splits = 0
for each input file f:
   remaining = f.length
   while remaining / split_size > split_slope:
      num_splits += 1
      remaining -= split_size
```

where:

split_slope = 1.1
split_size =~ blocksize

Mappers
How Many?

- Some files, generally those compressed using techniques like gzip, are not able to be split for processing by multiple mappers
- In this case the MapReduce execution engine arranges for only one map task to execute
- Some applications don't want files to be split, even where this is possible…
- As this allows a single mapper to process each input file in its entirety
- For example, a simple way to check if all the records in a file are sorted is…
- Go through the records in order, checking whether each record is not less than the preceding one
- Implemented as a map task, this algorithm will work only if one map processes the whole file

Reducers
How Many?

- By default there is a single reducer…
- And therefore a single partition of data holding all the intermediate key-value pairs generated by mappers
- But almost all real-world tasks should have more than one reducer…
- As otherwise the MapReduce job will be very slow to execute…
- Since all intermediate data flows through a single reduce task

Reducers
How Many?

- Choosing the number of reducers is often a matter of experience and trial and error
- Increasing the numbers of reducers makes the reduce phase shorter by allowing parallel execution
  - For n reduce tasks the intermediate data will be divided into n partitions
- But if you use too many reduce tasks each one may process a partition whose size is small
  - For a small partition the startup and shutdown of a reduce task may take more time than to process its partition
  - And in a shared environment with too many reduce tasks startup may be delayed by YARN until enough resources are available
  - Or other MapReduce jobs may be starved of needed resources

MapReduce Framework
Fault Tolerance

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce task is aborted and client is notified

MapReduce Framework
What is in Framework Control

- Limited control over data and execution flow
  - All algorithms must expressed in m, r, c, p
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

MapReduce Framework
What is in Developer Control

- The ability to construct complex data structures as keys and values to store and communicate partial results.
- The ability to execute user-specified initialization code at the beginning of a map or reduce task
- The ability to execute user-specified termination code at the end of a map or reduce task
- The ability to preserve state in both mappers and reducers across multiple input or intermediate keys
- The ability to control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys
- The ability to control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer

Appendix

MapReduce Explained Simply ☺

- Our goal is to process an HDFS files or S3 object as input and to produce an HDFS file or S3 object as output
- Why do this… to derive insight from a large volume of data impossible to explore manually or with standard programs
- So we write a program composed of several mandated or optional functions which is then distributed across a cluster
- This program is generally referred to as a MapReduce job
- The MapReduce framework distributes this program across the cluster for its functions to execute in parallel
- An executing function of this program/job is referred to as a task
- Functions (tasks) are called in a specific sequence to perform the programmer specified data processing activities

MapReduce Explained Simply ☺

- So we start with some input HDFS file or object
- Some part of the input file or object is assigned to a specific one of the tasks to process… One split per task
- This initial task executes one of the functions included in the program/job distributed across the cluster
- The initial function/task is referred to as the Mapper… and the function is generally named that
- This part of the file or object is called a split… Usually for HDFS files a split corresponds to a block
- The Mapper is given data from the input split with which it is associated, one record/line at a time
- The input to the Mapper function is a record from the split with which it is associated… But what is the format of that input?

MapReduce Explained Simply ☺

- The MapReduce framework exchanges data across Mappers and other functions in a very standardized way
- This is through the use of key value pairs, which we will write as

  <key, value>
- And which is provided to the Mapper function to represent the input record
- Each record read from a split is passed to a mapper as a key value pair
- And the Mapper can output zero, one or more key value pair for each input key value pair it is passed
- Key value pairs input to and output from the Mapper have a very simple form
- Keys and values are either a primitive data type such as int, float, string or double, or a single Java or Python object or a Python tuple

MapReduce Explained Simply ☺

- Each key value pair passed to the Mapper has the following general form

  <offset-of-record-in-the-file, record-content-as-a-string>
- In practice the key of the key value pair passed to the Mapper is usually ignored… we just care about the record content
- The Mapper accepts an input key value pair of the above form, parses out the record part…
- Then the Mapper then performs some specific transformation on the value part, and generates zero or more key value pairs as output
- The key value pairs output by the Mapper have the following general format

  <app-specific-key, single-app-specific-value>

MapReduce Explained Simply ☺

- The output of each Mapper is based on a partial view of the entire input HDFS file or S3 Object… a single split view
- But now we need to consolidate records from all of the Mappers together to create a summary or aggregated result
- So another function must be coded in our program/job to accept intermediate output from Mappers and do the summary processing
- This function/task is referred to as the Reducer…but the Reducer is not passed the exact output (key value pairs) produced by action of the Mappers
- The MapReduce execution engine (framework) does some additional processing of the key value pairs output by Mappers before providing them as input key value pairs to the Reducers
- This occurs during what is called the shuffle and sort step… the values associated with the same keys are grouped together into collections

MapReduce Explained Simply ☺

- Here an example will be helpful…
- Let’s say there are two Mappers and one Reducer function/task: the mapper task output the following simple key value pairs as follows:

  Mapper1 -> <hi, 1> <there, 1>

  Mapper2 -> <hi, 1> <you, 1>
- During the shuffle and sort phase these key value pairs are reorganized into the following, note we using curly brackets to indicate a collection:

  <hi, {1, 1}> <there, {1}> <you, {1}>
- That is all the values associated with the same key are grouped into a collection, and that collection becomes the value of the key value pair used as input to the reducer
- The shuffle and sort does one more thing…it sorts the key value pairs to be provided as input to the Reducer(s) in lexicographic order, by key (and NOT by value)

MapReduce Explained Simply ☺

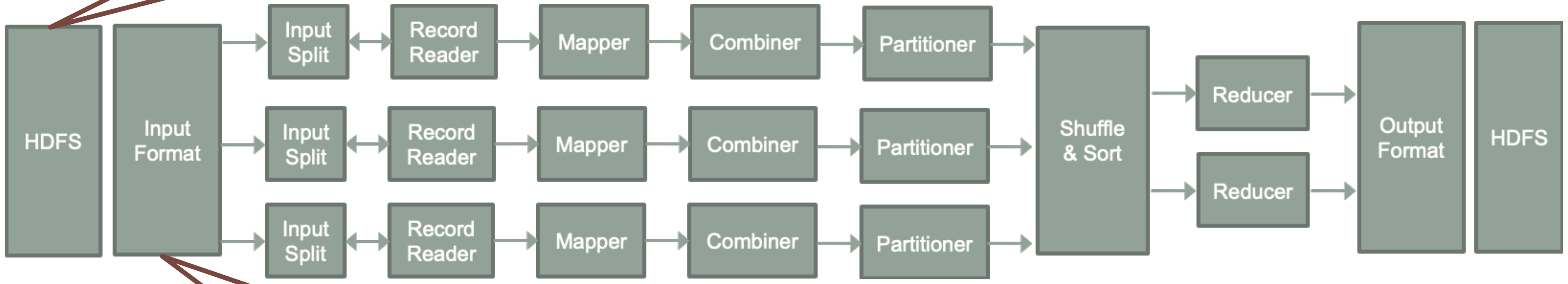
- So each key value pair passed to a Reducer has the following general form

  <app-specific-key, collection-of-values-for the-key>
- The Reducer accepts an input key value pair of the above form then performs some specific transformation and generates zero or more key value pairs as output
- The key value pairs output by the Reducer have the following general format

  <app-specific-key, single-app-specific-value>
- These key value pairs are written to an output HDFS file or S3 Object

MapReduce Processing Flow in More Detail

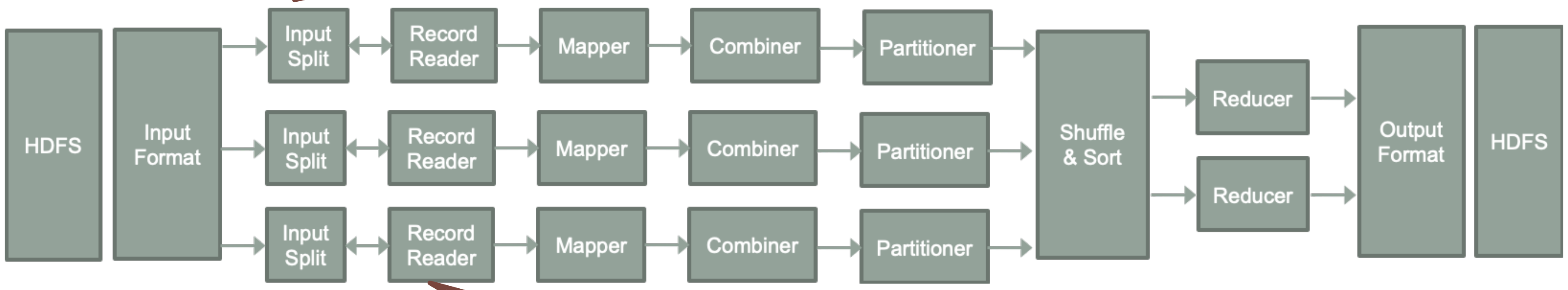
The data for a MapReduce task is stored in **input files**, and input files typically lives in **HDFS or S3**



InputFormat splits the HDFS Input file into InputSplit(s) and assigns them to individual Mapper(s)

MapReduce Processing Flow in More Detail

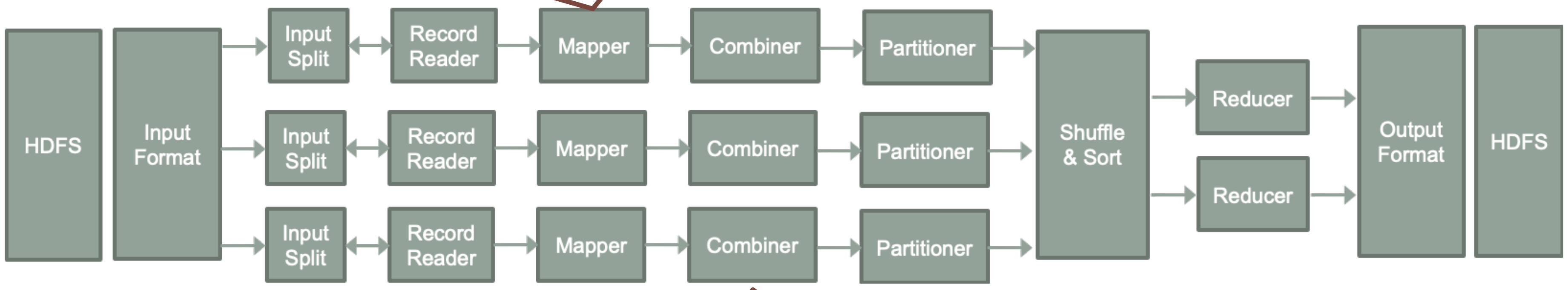
It is created by InputFormat, logically represent the data that will be processed by an individual Mapper. One map task is created for each split; so the number of map tasks will be equal to the number of InputSplits.



It communicates with an InputSplit, reads the next record and converts it into key-value pairs suitable for providing to the mapper

MapReduce Processing Flow in More Detail

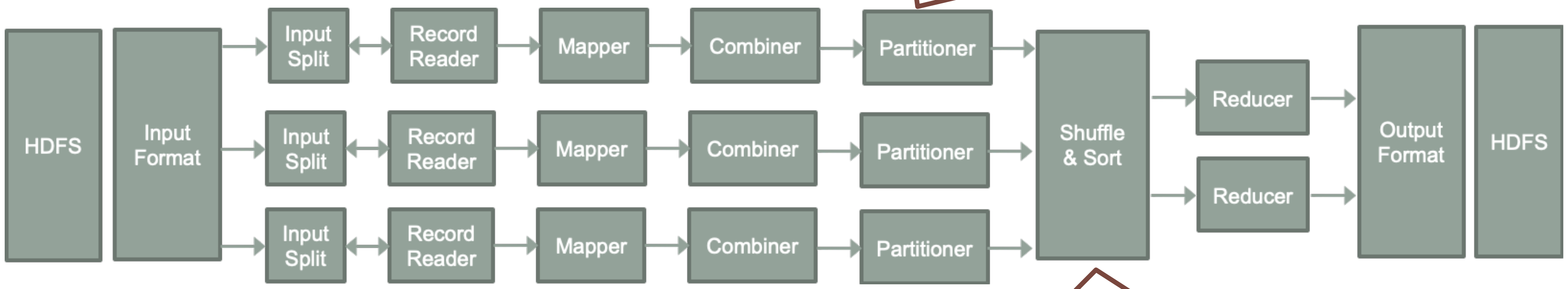
It executes once for each input record (from RecordReader) and generates one or more new key-value pairs. The output of each Mapper is first buffered into local memory and then, as needed, is written to the local disk



The Combiner performs local aggregation on the mappers' output, which helps to minimize the data transfer between mapper and **reducer**

MapReduce Processing Flow in More Detail

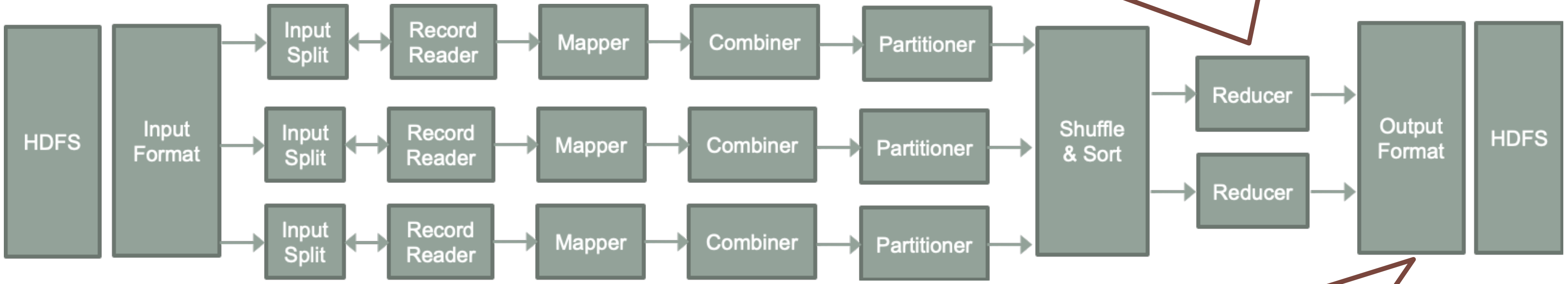
Each combiner output is partitioned, key-value pairs with the same key goes into the same partition, then each partition is sent to a reducer. If there is more than one reducer the partitioner applies an algorithm to chose to which reducer a partition is sent.



Shuffling is the physical movement of data partitions over the network to the reducer selected by partitioners. Once all the mappers are finished and their output is shuffled on the reducer nodes, then this intermediate output is merged and sorted by key, which is then provided as input to reducer

MapReduce Processing Flow in More Detail

It takes the set of intermediate key-value pairs produced by the mappers as the input and then runs a function on each of them to generate the output



The way output key-value pairs are written in output files by is determined by the OutputFormat