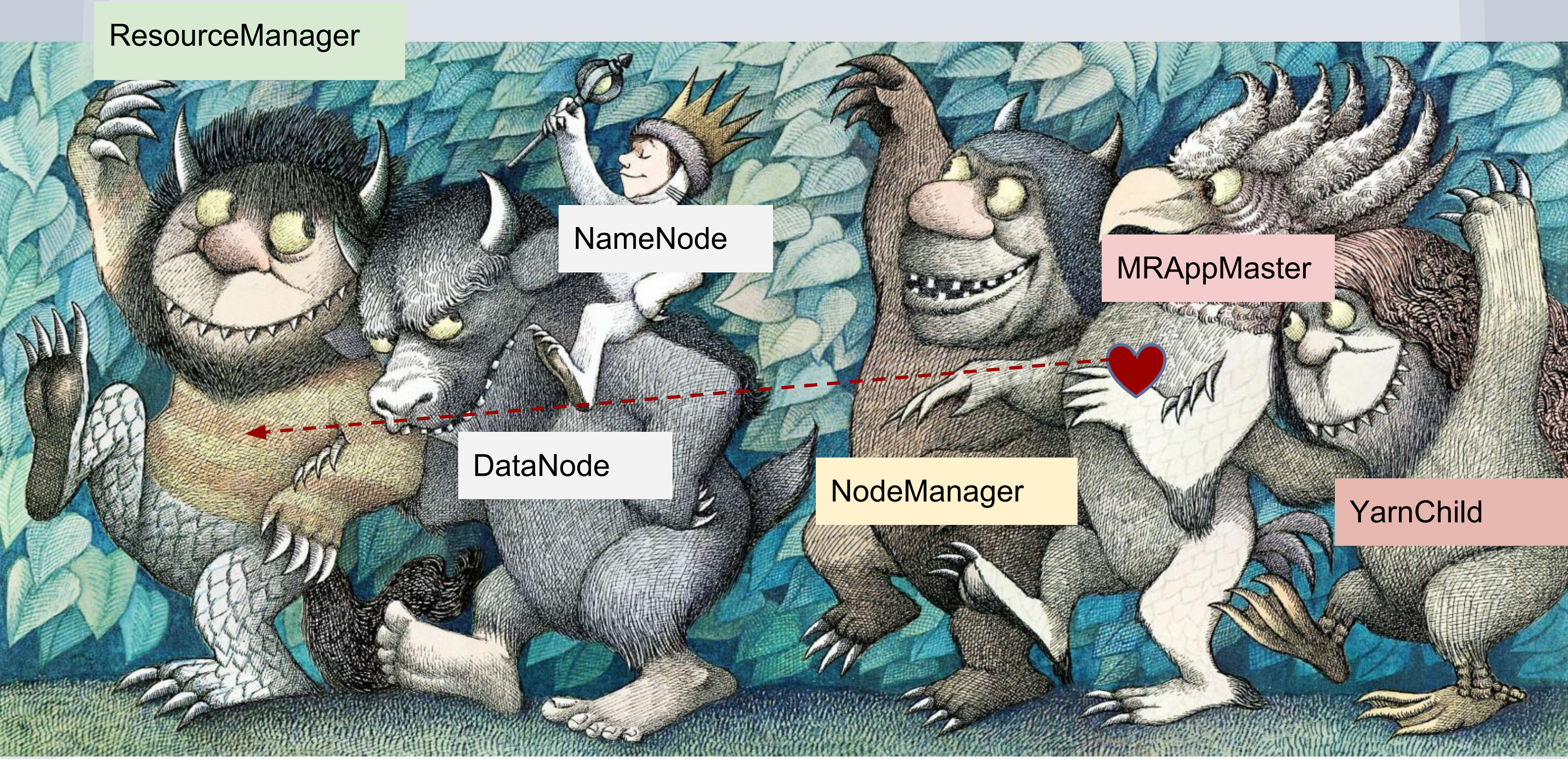
Practice using MapReduce

“Let the wild rumpus begin!”
- Maurice Sendak


ResourceManager
NameNode
MRAppMaster
DataNode
NodeManager
YarnChild

Key points from the last lecture

Name nodes know everything about the data (metadata, FS images)

- data nodes are dumb
- blocks are BIG - 128 M
- block contents are unstructured

Writing HDFS data is distributed and efficient

- pipelines are created for replication
- writes are thoroughly verified

The NameNode

- manages the “metadata” for data stored on the cluster
- monitor the health of Datanodes

The ResourceManager schedules Jobs and monitors their health

Advanced: References for HDFS

- **HDFS in more depth:**

  Hadoop: The Definitive Guide, 4rd Edition, by Tom White,Chapter 3.

- **File system shell guide**

  http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html

- **Namenode startup**

  http://hortonworks.com/blog/understanding-namenode-startup-operations-in-hdfs/

- **Namenode availability: checking-pointing explained**

  http://blog.cloudera.com/blog/2014/03/a-guide-to-checkpointing-in-hadoop/

- **HDFS Metadata directories explained**

  http://hortonworks.com/blog/hdfs-metadata-directories-explained/

Agenda

- In-class practice 1: setup VM and work with HDFS
- **What is Map Reduce**
- **Intro to Spark**
- In-class practice 2: running Map Reduce jobs
    - Learn to run both Spark and MR2 jobs in Eclipse
    - Learn to submit Spark and MR2 jobs to a cluster

Practice 1

Setup and use the VM

Flash drive

- **Contains a file called "fall2016.ova"**
- ***Practice 1 - see* install directions for fall2016.ova**
  - For the cowboys:
    - Don't unzip or try to open this file.
    - Don't try to create a "new" VM, use the import function as described in *Practice 1*.

Important: copy the .ova to your computer before you import it!

- Flashdrives have a limited number of "writes"
- If you try to run the fall2016.ova from your flashdrive it will be intolerably SLOW
- These flashdrives get HOT. Really hot.

The next 45 - 60 minutes ...

Import the VM

- **time to copy fall2016 to your computer: 5-10 minutes**
- **time to import fall2016.ova to your VM: 10-15 minutes**
- **time to boot VM: 5-10 minutes**

Do some things in the VM:

- **get comfortable with using the VM**
- **work with HDFS**

FOLLOW THE PRACTICE 1 instructions closely and in order...

Deep dive MapReduce

APIs and job submission

The 3 phases of a Map-Reduce job in MR2

Map phase

- **MapTask sets up the Mapper by running Mapper's**
  - **setup() method**
  - **run() method**
  - **cleanup method**
- **The Mapper's run() method executes map() for each record in a "split"**

Shuffle-sort - buckets up the mapped data, sorts and ships to the reducers

Reduce phase

- **ReduceTask merges the data buckets fetched from the Mappers**
- **ReduceTask runs Reducer's set-up, run and tear-down methods**
- **Output of each Reducer is saved as a file in HDFS**

The MapReduce API for traditional MR2

Developers customize MapReduce jobs by extending:

- InputFormat
- RecordReader
- Mapper
- Combiner
- Partitioner

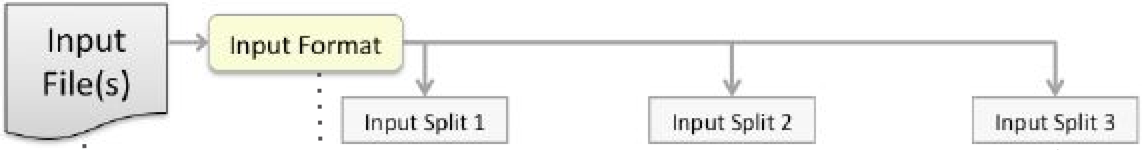
Instantiated and run in the MapTask

- Reducer
- OutputFormat
- RecordWriter

Instantiated and run in the ReduceTask

These classes are exposed as part of MR2's API.

MapReduce Processing



Input Files are stored in blocks

- blocks of bytes
- no info about contents

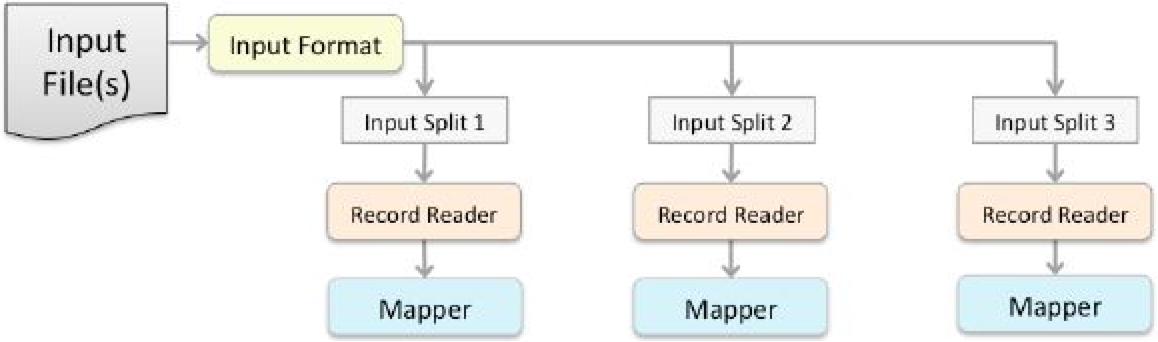
InputFormat is a class

- knows how to read a block
- turns blocks into “splits”

An **Input Split** is a set of data

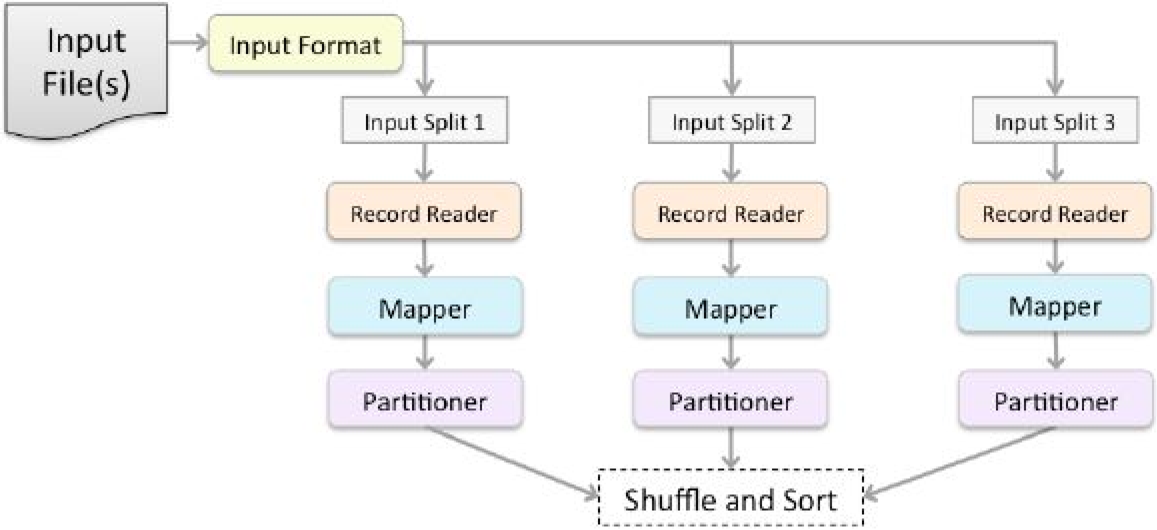
- Built from blocks
- Preserves record boundaries
- Splits are about the same size as a block*

MapReduce Processing



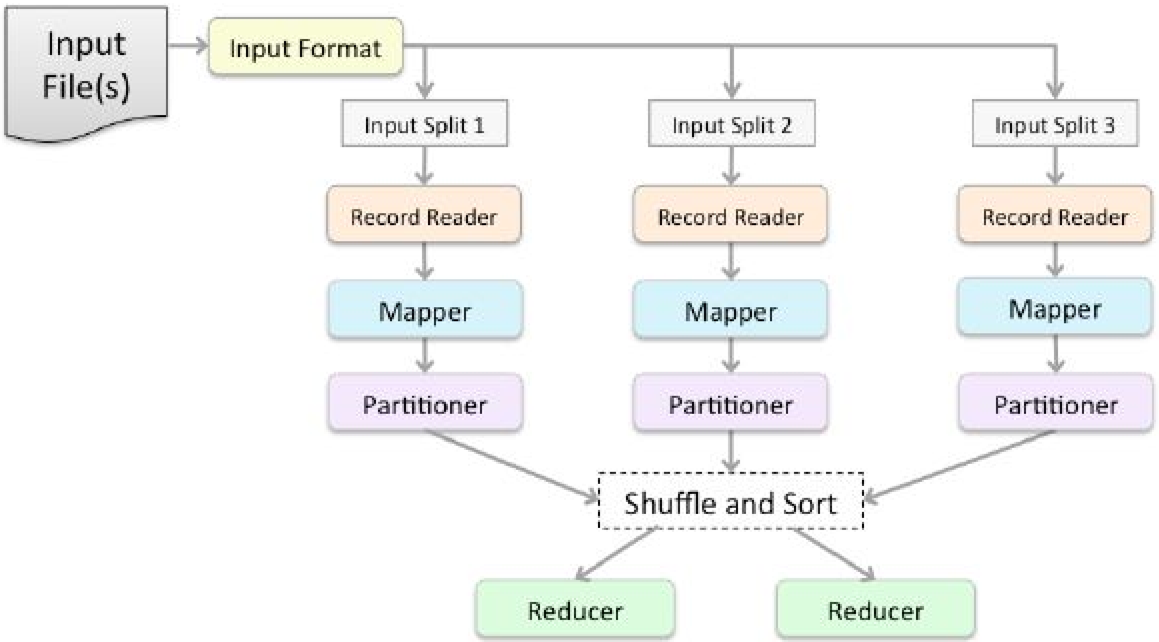
- Every **Split** is readable by a **RecordReader**
    - RecordReaders turn Splits into a iterable list of Records
- Every **MapTask** uses the **RecordReader** to feed Records to a **Mapper**
- Each **Mapper** processes one **Split**

MapReduce Processing



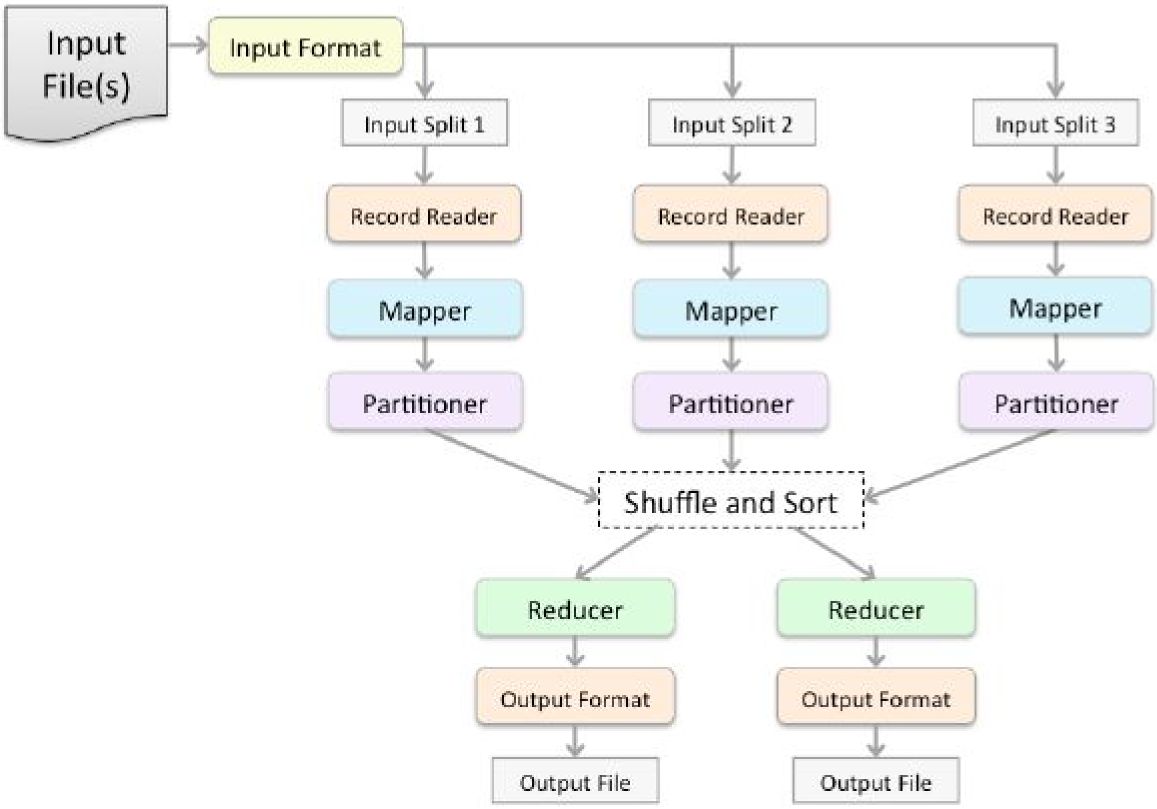
- **Mapper outputs are given a partition index**
- **Shuffle-sort (in the MapTask):**
    - **Mapper** results are sorted and **partitioned** into files
    - The **partitioned** files are available for HTTP fetch by Reducer Tasks

MapReduce Processing

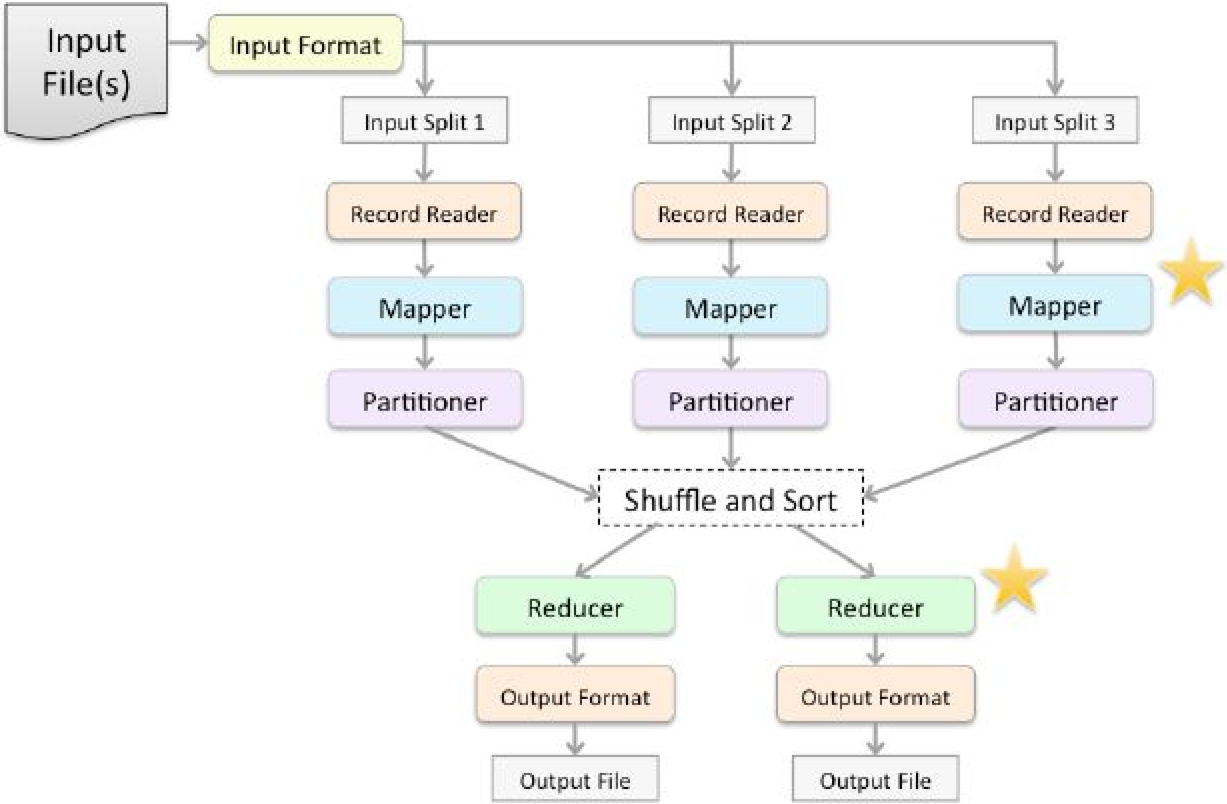


- **ShuffleSort** (In the ReduceTask) - The ReducerTask fetches and merges partitioned files into a mergeFile
- The **Reducer** reads and processes the mergeFile

MapReduce Processing



MapReduce Processing



Review: MapTask steps

For each block of data, create a MapTask. Each MapTask does the following:

1. **InputFormat:** convert blocks into input splits that respect record boundaries.
2. **RecordReader:** parse split into records, write record as <k,v> pair.
3. **Mapper:** run function on each <k,v> pair and write <k',v'>, as desired.
4. **Partitioner:** assigns an index, p, to each <k, v> pair => {p, k, v}

map-side shuffle-sort

5. **In the background, MapTask runs the spill-thread**:
    - aggregates all the values for a key into a list
    - partitions map output into buckets
    - writes each bucket to a different file, one for each reducer

Review: ReduceTask steps

reduce-side shuffle-sort

1. **Network phase:** ReduceTasks fetch MapTask output files
    - Each ReduceTask is assigned a partition
    - A ReduceTask fetches the buckets for its partition from the MapTasks
    - Reduce tasks do not start ‘reducing’ until ***all*** the MapTasks are done.
2. **Merge-sort:** The ReduceTask merge-sorts all fetched buckets into one file
3. **Reducer:** processes each <k, {v1, v2, v3…}> and outputs results: <key, result>
4. **OutputFormat:** determines format and writes the output. Usually, each Reducer produces a separate file.

Example: Average word length

(Covered in more detail in Practice 3)

- **Read in Shakespeare plays**
- **Mapper: find the first letter of each word**
  - **write out the first letter as key**
  - **write out the length of the word as value**
- **Reducer: read in the letter and list of lengths**
  - **iterate through the list, summing**
  - **calculate the average**
  - **output the key (letter) and the average length**

AvgWordLength - main method

```
public static void main(String[] args) throws Exception {

    Job job = Job.getInstance();
    job.setJobName("Average Word Length");
    job.setJarByClass(AvgWordLength.class);

    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(LetterMapper.class);
    job.setReducerClass(AverageReducer.class);

    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(DoubleWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);

}
```

Create and name the job
Find the jar containing this class

Define input and output locations

Set the Mapper and Reducer classes

Set the output classes for the Mapper and Job

LetterMapper

```java
package averageWordLength.solution;

public class LetterMapper extends Mapper<LongWritable, Text, Text, IntWritable>{

    @Override
    public void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {

        String line = value.toString();

        for (String word : line.split("\\W+")) {

            if (word.length() > 0) {

                String letter = word.substring(0, 1);

                context.write(new Text(letter), new IntWritable(word.length()));
            }
        }
    }
}
```

Letter Mapper in action

Input split

Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal

Key, value inputs

Key	Value
0	Four score and seven years ago our
34	fathers brought forth on this continent
74	a new nation, conceived in liberty
109	and dedicated to the proposition that

Letter Mapper

map output

F	4
s	5
a	3
s	5
y	5
a	3
o	3

n	3
n	5
c	9
i	2
l	7

f	7
b	7
f	5
o	2
t	4
c	10
a	1

Map-side shuffle-sort

merge-sort

add partition index

key	values	partition index
a	1,3,3,3,3	0
b	7	0
c	9,7	0
d	9	0
e	5	0
f	4,7	0
i	7	1
l	9	1
m	3	1
n	3, 5	1
o	3	1
p	11	1

partition 0

part-m-00000

a	1,3,3,3,3
b	7
c	9,7
d	9
e	5
f	4,7

partition 1

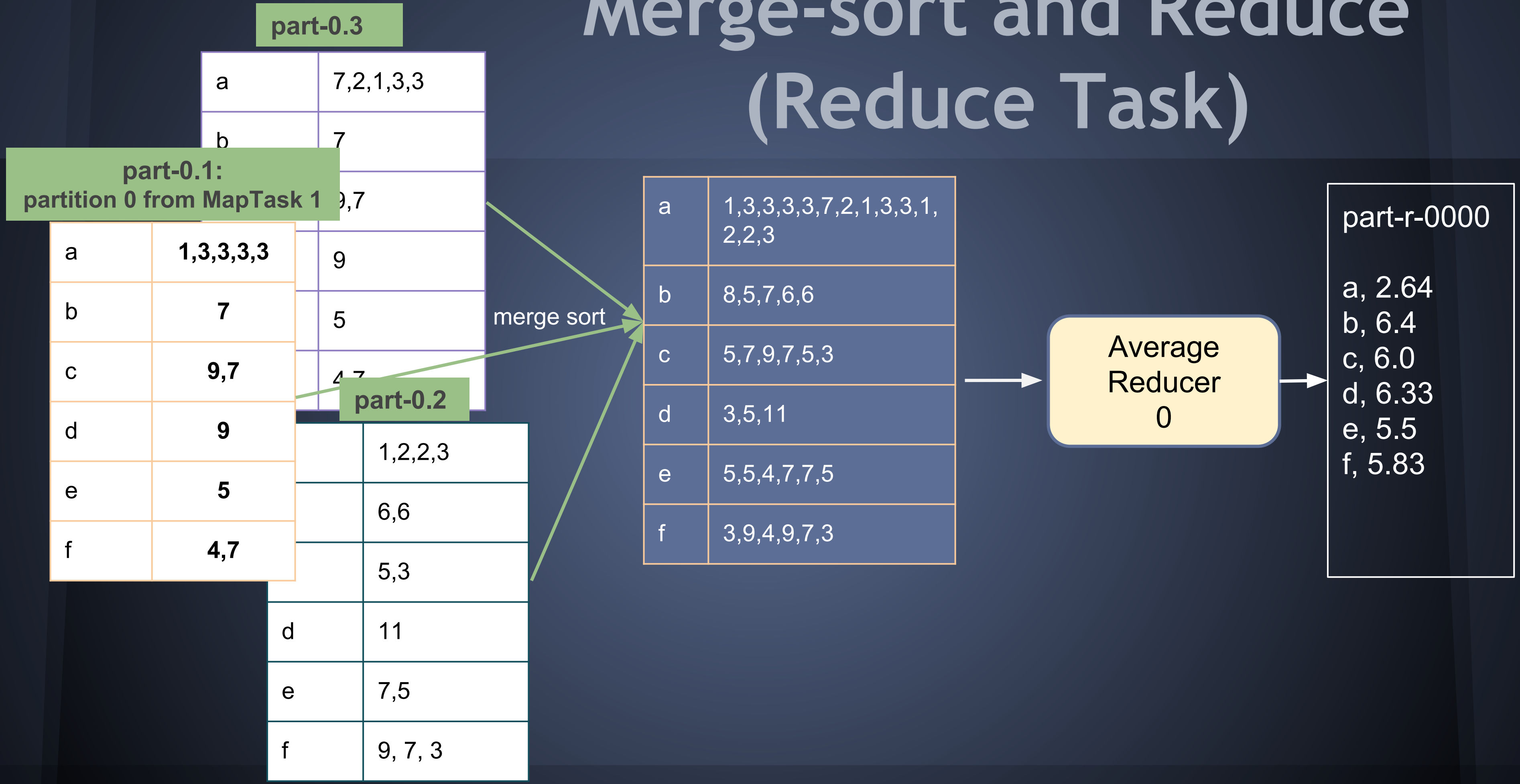
part-m-00001

i	7
l	9
m	3
n	3, 5
o	3
p	11

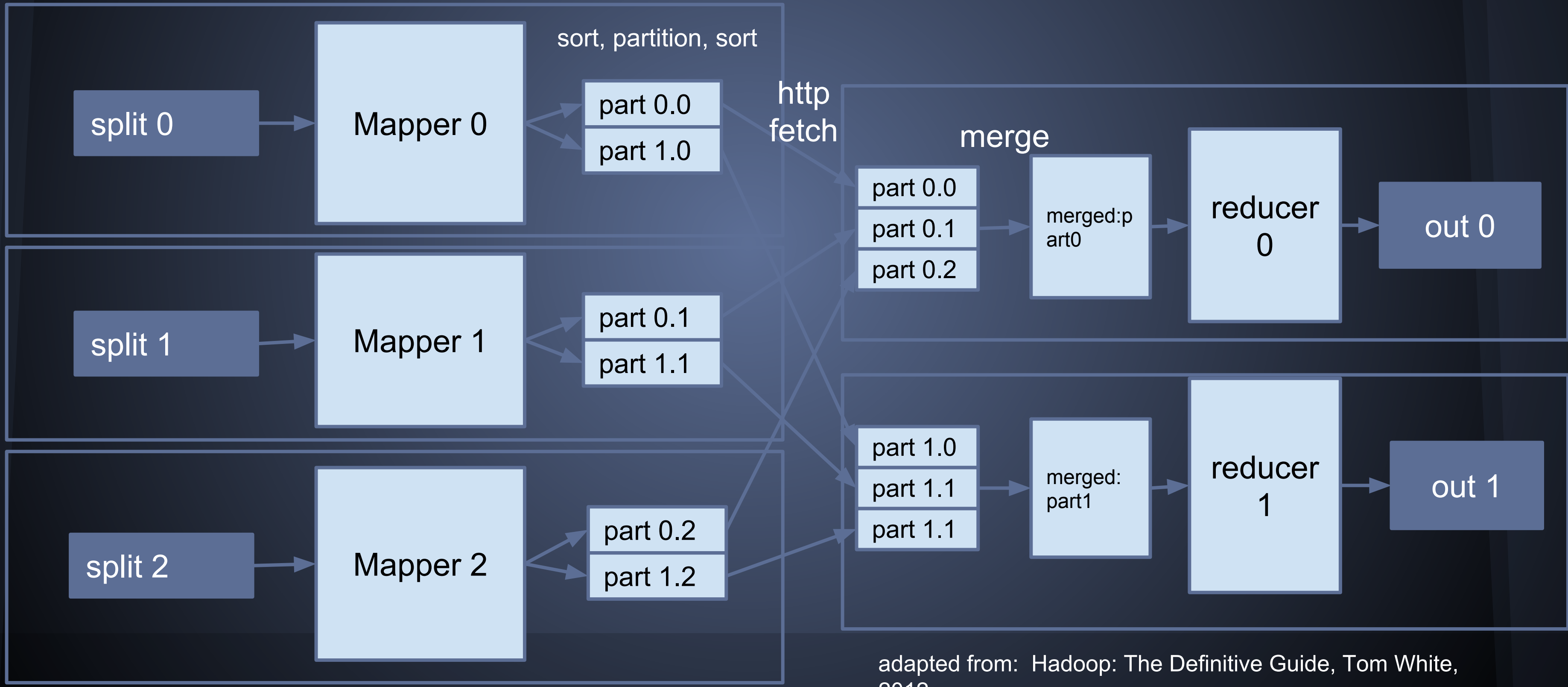
AverageReducer

```
public class AverageReducer extends Reducer<Text, IntWritable, Text, DoubleWritable>
{
  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    long sum = 0, count = 0;

    for (IntWritable value : values) {

      sum += value.get();
      count++;
    }
    if (count != 0) {

      double result = (double)sum / (double)count;
      context.write(key, new DoubleWritable(result));
    }
  }
}
```

Merge-sort and Reduce (Reduce Task)



OVERVIEW: map-reduce data flow



adapted from: Hadoop: The Definitive Guide, Tom White, 2012

Creating and Running a MapReduce Job

Development

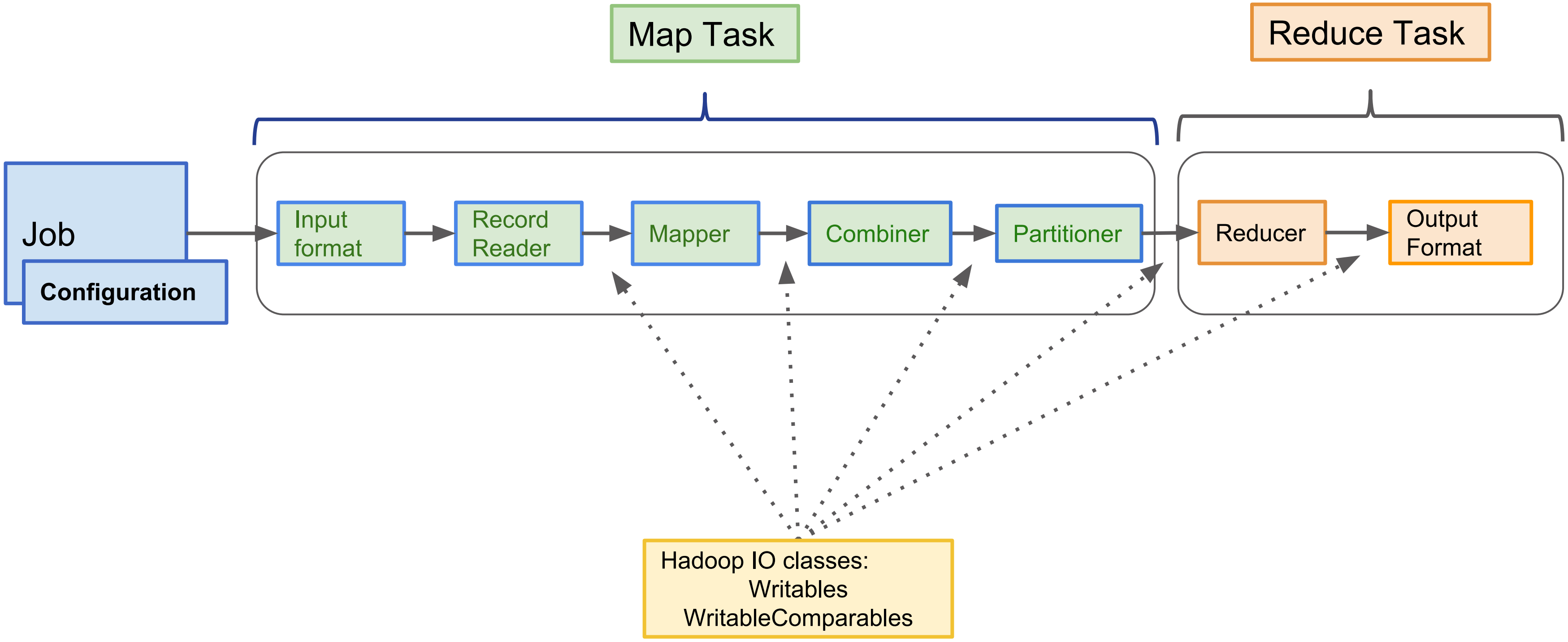
- **Write the code in Eclipse**
- **Let Eclipse compile the code (automatic)**
- **Test the code by running it in Eclipse (use Run Configurations)**

Submitting to the cluster

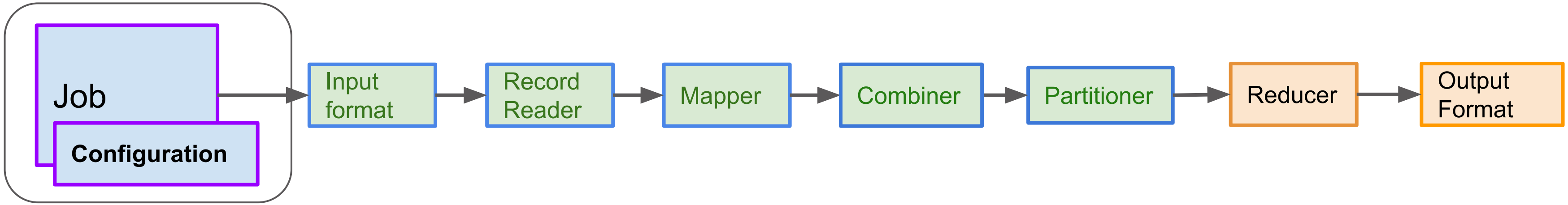
- **Create a jar file from the compiled code**
- **Use the ‘hadoop jar’ command to submit**

Detailed instructions in Practice 2

Important MapReduce Classes



Job configuration in the main method



The main method submits a job

- **The main method creates a Job.**
- **The Job wraps a Configuration object.**
- **We define a Job using setters in the Job class.**
  - set configuration properties
  - define the MR classes the developer has extended
  - define IO paths and IO types

Going deeper: The Job implements MRJobConfig

(see org.apache.hadoop.mapreduce.MRJobConfig)

The simplest job driver

```
public static void main(String[] args) throws Exception {

    Job job = Job.getInstance();
    job.setJobName("Identity Job");
    job.setJarByClass(IdentityDriver.class);

    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Note: No Mapper or Reducer classes defined⟶ Top-level Mapper and Reducers are used:

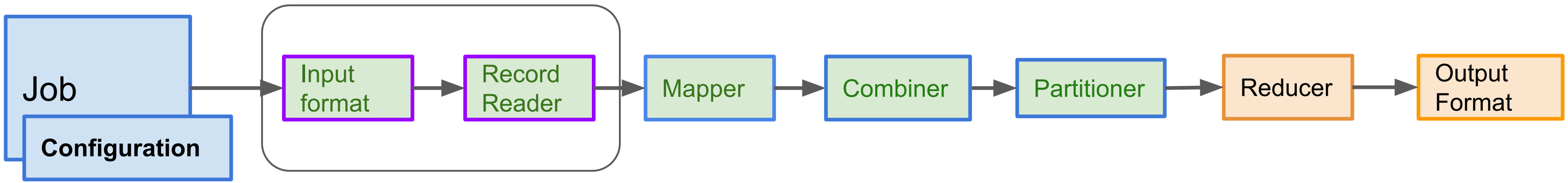
- the top-level Mapper and Reducer do not change the data.
- result is the same as the input data except it is sorted (it went through shuffle-sort)

Job setters and their defaults

Setter	**Default value**
setInputFormatClass	TextInputFormat
setMapperClass	Mapper
setMapOutputKeyClass	LongWritable
setMapOutputValueClass	Text
setPartitionerClass	HashPartitioner
setNumReduceTasks	1
setReducerClass	Reducer
setOutputKeyClass	LongWritable
setOutputValueClass	Text
setOutputFormatClass	TextOutputFormat

If you do <u>not</u> use the setter, then the default is used.

Finding and formatting input data



Specifying input locations

- **Set the input location using InputFormat.**

```
FileInputFormat.setInputPaths(job, new Path(<dir>))
```

- **This will read the files in <dir> and execute them in MapReduce.**
    - **Won't read files that start with "." or "_" (hidden files)**
    - **Can use wildcards to restrict input: /2010/*/Jan/***
    - **<dir> can be a directory or a file**

- **To add multiple paths:**

```
FileInputFormat.addInputPath(job, new Path(<file>))
```

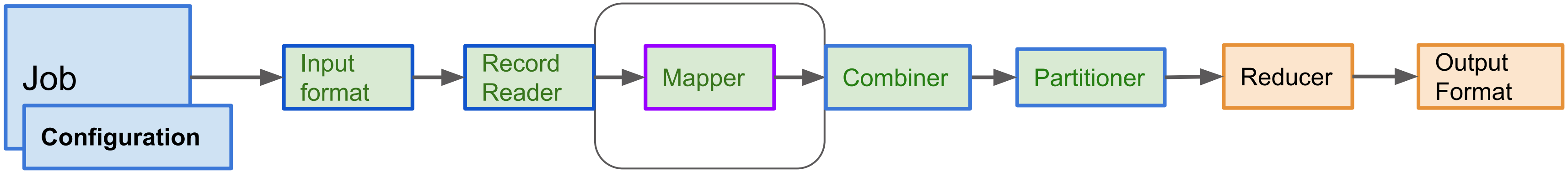
What is an InputFormat?

- **Holds the location of the input, usually a file or directory.**
- **Formats data blocks into splits and ten splits into records.**
- **InputFormats and OutputFormats are part of Hadoop IO**
  - **Hadoop IO is used by *both MR2 and Spark***
  - **Details on InputFormats are covered in a later lecture**

For now, we'll just be using the default TextInputFormat

Format type	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
Key-value text	Semi	Format for key-value pairs. Delimiters can be configured.
Whole file input		Processes one file per split.
Sequence File	Yes	A compressible format for binary data. Header holds metadata about contents. Has sync points for splitting.
Avro File	Yes	A flexible, compressible format with associate schema for handling complex, structured, evolving data.
DB (SQL statement)	Yes	A format for SQL statements from databases. (Most database tables are loaded using Sqoop)
Other: JSON, CSV, XML	Semi	Read in using TextFileFormat; parse in map using json/XML/csv specific libraries. May also use Hadoop streaming.

Mappers


Job
Configuration
Input format
Record Reader
Mapper
Combiner
Partitioner
Reducer
Output Format

The Mapper runs on every MapTask

- **Hadoop attempts to ensure that Mappers run on nodes which hold their portion of the data locally, to avoid network traffic**
  - Multiple Mappers run in parallel, each processing a portion of the input data
- **The Mapper reads data in the form of key/value pairs**
  - The Mapper may use or completely ignore the input key
  - For example, a standard pattern is to read one line of a file at a time
    - The key is the byte offset into the file at which the line starts
    - The value is the contents of the line itself
    - Typically the key is considered irrelevant
- **If the Mapper writes anything out, the output must be in the form of key/value pairs**

Example Mapper: Upper Case Mapper

- **Turn input into upper case (pseudo-code):**

```
let map(k, v) =
    emit(k.toUpper(), v.toUpper())
```

bugaboo	an object of fear or alarm	map()	BUGABOO	AN OBJECT OF FEAR OR ALARM
mahout	an elephant driver	map()	MAHOUT	AN ELEPHANT DRIVER
bumbershoot	umbrella	map()	BUMBERSHOOT	UMBRELLA

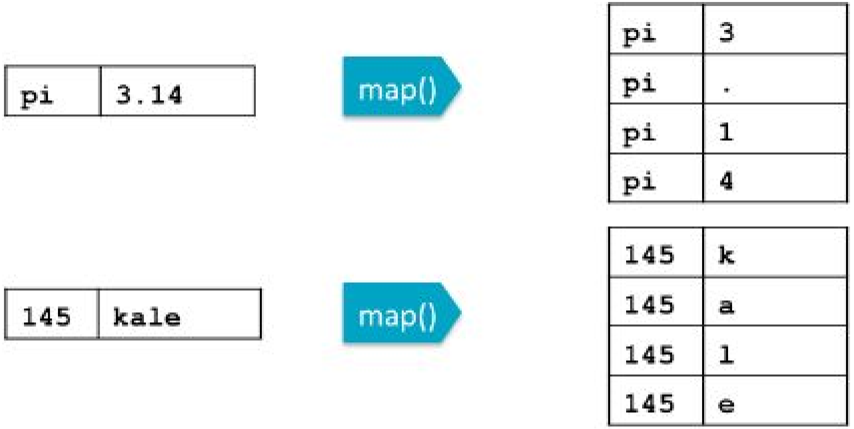
UpperCaseMapper

```
public class UpperCaseMapper extends Mapper<Text, Text, Text, Text> {

    private Text uKey = new Text();
    private Text uValue = new Text();

    @Override
    public void map(Text key, Text value, Context context) {

        uKey.set(key.toString().toUpperCase());
        uValue.set(value.toString().toUpperCase());
        context.write(ukey, uvalue);

    }

}
```

Example Mapper: ‘Explode’ Mapper

- **Output each input character separately (pseudo-code):**

```
let map(k, v) =
    foreach char c in v:
        emit (k, c)
```



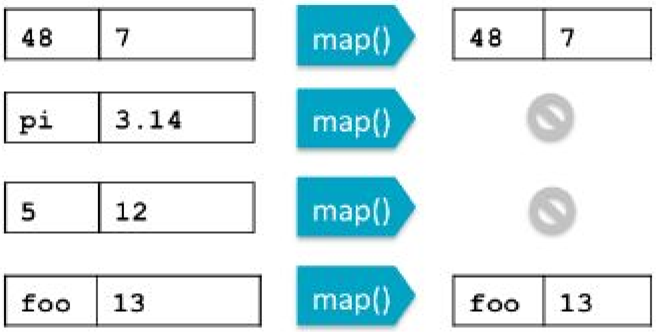
ExplodeMapper

```
public class ExplodeMapper extends Mapper<Text, Text, Text, Text> {

    private Text c = new Text();

    @Override
    public void map(Text key, Text value, Context context) {

        char[] array = value.toString().toCharArray();
        for (int i = 0; i < array.length; i++)
            context.write(key, c.set(array[i]);

    }

}
```

Example Mapper: ‘Filter’ mapper

- Only output key/value pairs where the input value is a prime number (pseudo-code):

```
let map(k, v) =
    if (isPrime(v)) then emit(k, v)
```



FilterMapper

```
public class FilterMapper extends Mapper<Text, IntWritable, Text, IntWritable> {

    @Override
    public void map(Text key, IntWritable value, Context context) {

        if (isPrime(value))
            context.write(key, value);

    }

    private boolean isPrime(IntWritable value) {
    … your sieves here …
    }

}
```

Example Mapper: Changing Keyspaces

- The key output by the Mapper does not need to be identical to the input key
- Example: output the word length as the key (pseudo-code):

```
let map(k, v) =
    emit(v.length(), v)
```

001	hadoop	map()	6	hadoop
002	aim	map()	3	aim
003	ridiculous	map()	10	ridiculous

Changing Key spaces: LengthMapper

```
public class LengthMapper extends Mapper<IntWritable, Text, IntWritable, Text> {

    IntWritable lengthKey = new IntWritable();

    @Override
    public void map(IntWritable key, Text value, Context context) {

        lengthKey.set(value.toString().length());
        context.write(lengthKey, value);

    }

 }
```

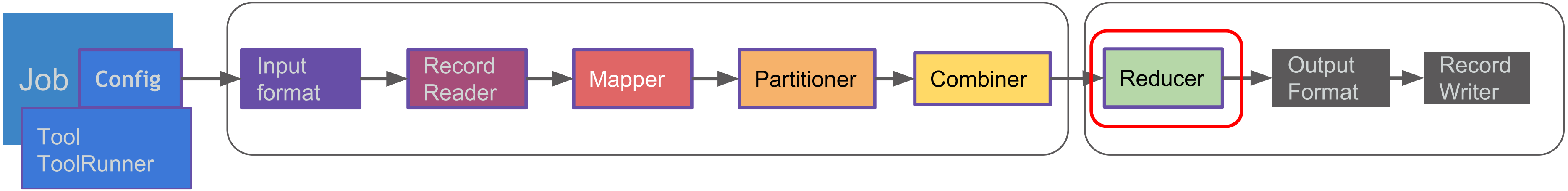
Example Mapper: Identity Mapper

- **Emit the key,value pair (pseudo-code):**

```
let map(k, v) =
    emit(k,v)
```

bugaboo	an object of fear or alarm	map()	bugaboo	an object of fear or alarm
mahout	an elephant driver	map()	mahout	an elephant driver
bumbershoot	umbrella	map()	bumbershoot	umbrella

Reducers



For each input key, the Reducer *reduces* the list of values to a smaller set of values.

The Reducer

- **After the Map phase is over, all intermediate values for a given intermediate key are combined together into a list**
- **This list is given to a Reducer**
  - There may be a single Reducer, or multiple Reducers
  - All values associated with a particular intermediate key are guaranteed to go to the same Reducer
  - The intermediate keys, and their value lists, are passed to the Reducer in sorted key order
- **The Reducer outputs zero or more final key/value pairs**
  - These are written to HDFS
  - In practice, the Reducer usually emits a single key/value pair for each input key

Example Reducer: Sum Reducer

- Add up all the values associated with each intermediate key (pseudo-code):

```
let reduce(k, vals) =
    sum = 0
    foreach int i in vals:
        sum += i
    emit(k, sum)
```

SumReducer code

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      context.write(key, new IntWritable(sum));
    }
  }
```

Example Reducer: Average Reducer

- **Find the mean of all the values associated with each intermediate key (pseudo-code):**

```
let reduce(k, vals) =
    sum = 0; counter = 0;
    foreach int i in vals:
        sum += i; counter += 1;
    emit(k, sum/counter)
```

Average Reducer code

```
public class AvgReducer extends Reducer<IntWritable, IntWritable, IntWritable, DoubleWritable> {

      DoubleWritable average = new DoubleWritable();

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

            int sum = 0;
            int count = 0;
            for (IntWritable value : values) {
                  sum += value.get();
                  count++;
            }
            average.set(sum / (double) count);
            context.write(key, average);
      }
}
```

create a holder for average

Iterate through the values in the list, adding to the sum and incrementing the counter.

value.get() retrieves the integer.
average.set() sets the double.

Example Reducer: Identity Reducer

- **The Identity Reducer is very common (pseudo-code):**

```
let reduce(k, vals) =
    foreach v in vals:
        emit(k, v)
```

bow	a knot with two loops and two loose ends
	a weapon for shooting arrows
	a bending of the head or body in respect

reduce()

bow	a knot with two loops and two loose ends
bow	a weapon for shooting arrows
bow	a bending of the head or body in respect

28	2
	2
	7

reduce()

28	2
28	2
28	7

Caveat: keep track of types

the output types of the mapper
must match
the input types of the reducer

Mapper and Reducer outputs must match main settings

```
public static void main(String[] args) throws Exception {

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        …
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FloatWritable.class);
```

job defined in main

```
public class MaxTempMapper extends Mapper<LongWritable, Text, Text, IntWritable>

public class MaxTempReducer extends Reducer<Text, IntWritable, Text, FloatWritable>
```

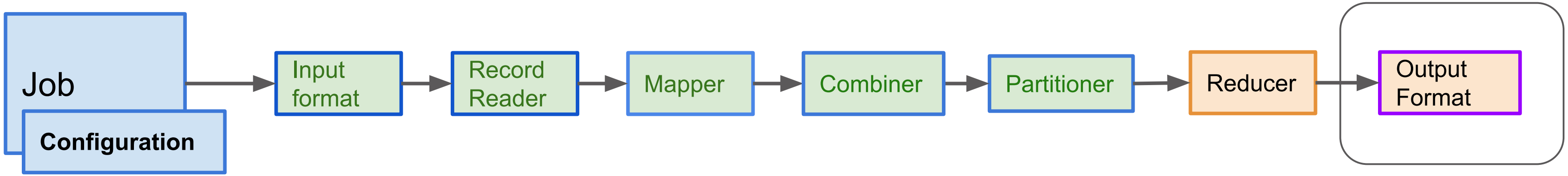
Mapper output must match Reducer input

```
public class MaxTempMapper extends Mapper<LongWritable, Text, Text, IntWritable>
```

Map outputs must match Reducer inputs.

```
public class MaxTempReducer extends Reducer<Text, IntWritable, Text, IntWritable>
```

Writing formatted output data



Specifying output locations

- **define the output location using OutputFormat:**

```
FileOutputFormat.setOutputPath(job, new Path(<dir>))
```

- **This defines the directory that receive the final (reduced) results.**
- **This directory must not exist - MapReduce will create it.**

Output formats

- **For now, we will use the default for the OutputFormat.**

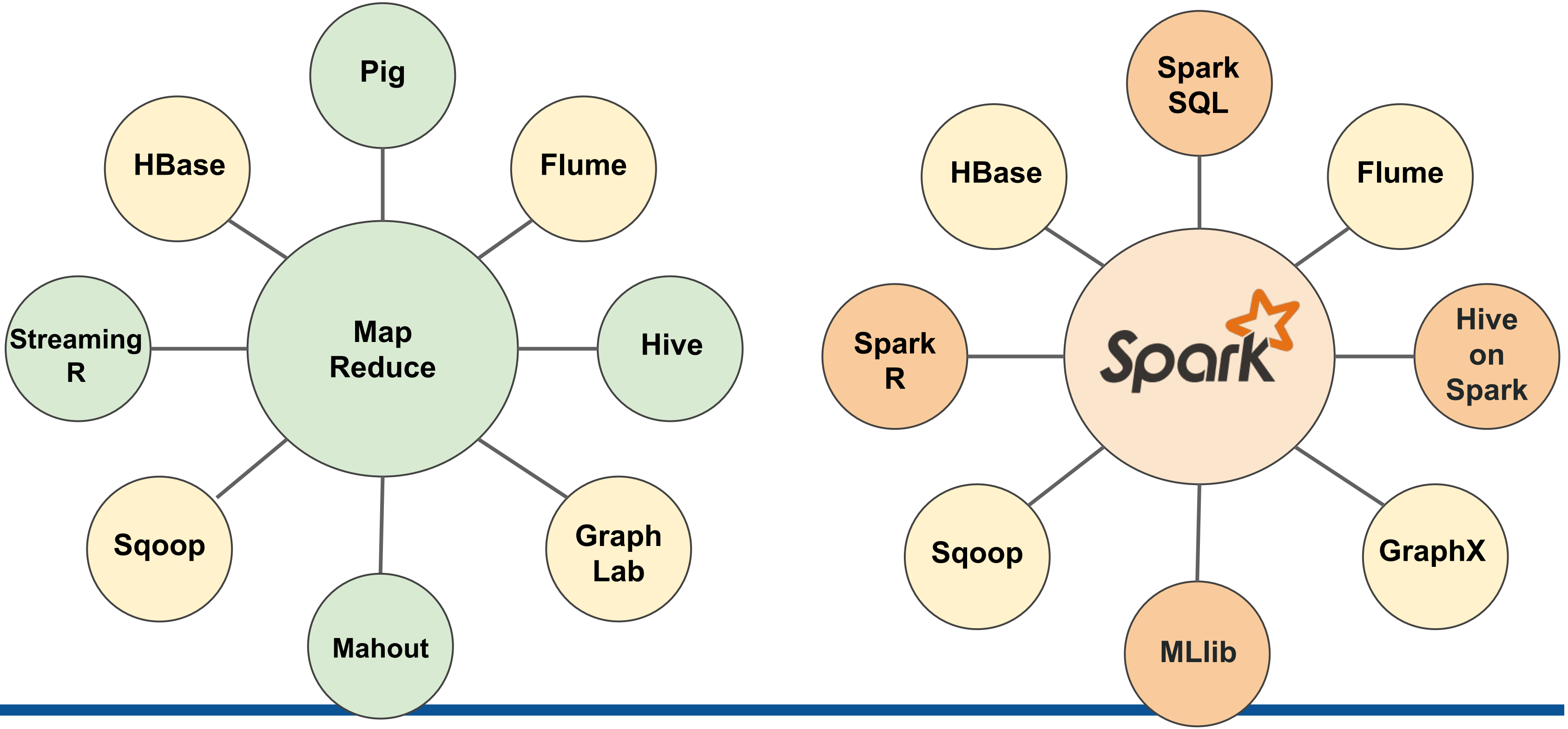
```
TextOutputFormat.class;
```

  - Regardless of the output classes used, it will write out the results as Text (String)

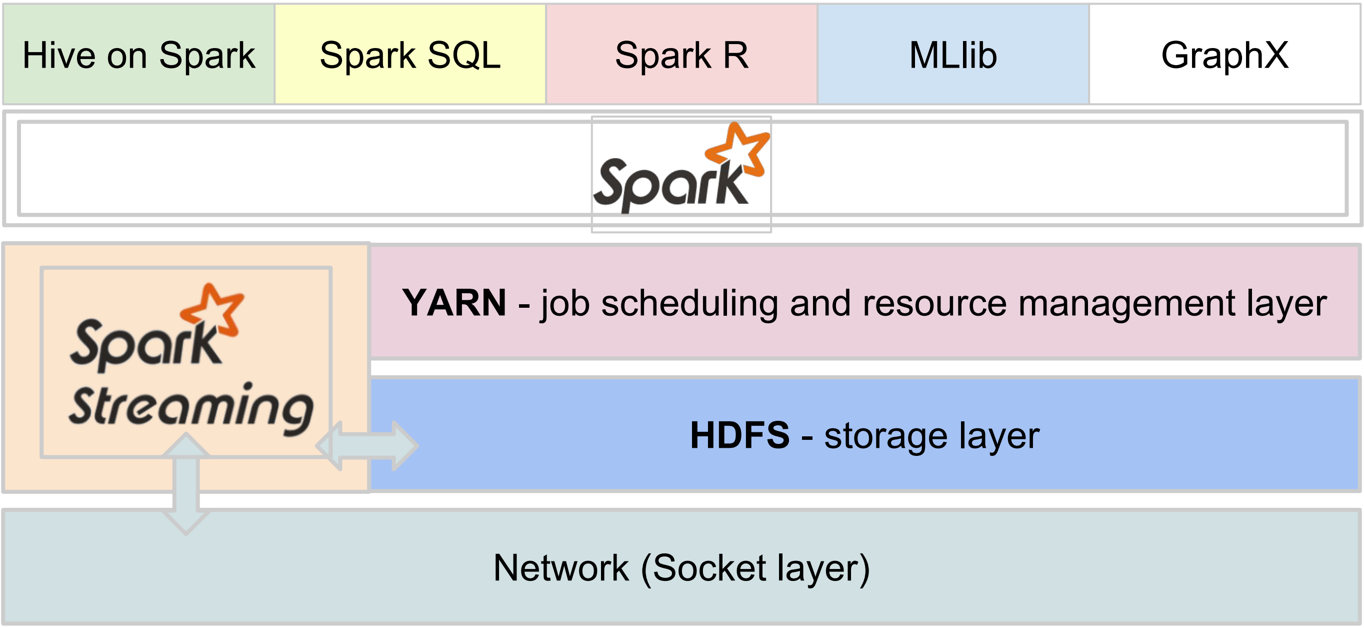
  - Very forgiving...

Spark

https://cwiki.apache.org/confluence/display/SPARK/Committers



Spark Analytics



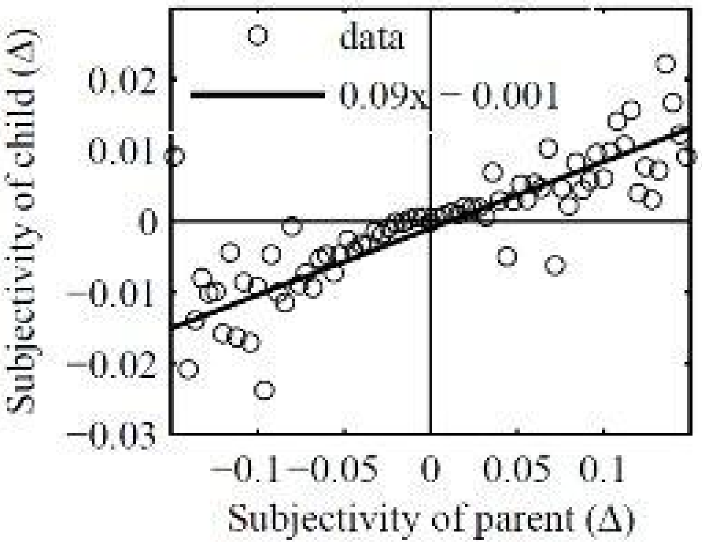
databricks

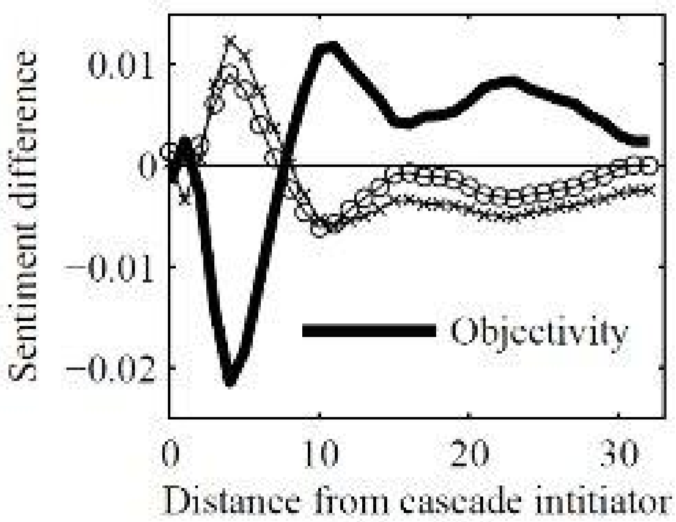
- Founded in late 2013
- by the creators of Apache Spark (Matei Zaharia’s PhD dissertation)
- Original team from UC Berkeley AMPLab
- Raised $47 million in 2 rounds
- <100 employees, 100% recommend on Glassdoor
- They’re hiring! (https://databricks.com/company/careers)
- Contributed more than 75% of the code in Spark

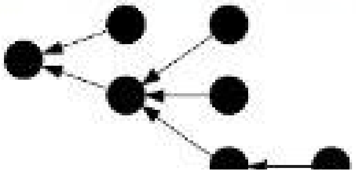
There's hype…

The course of sentiment

- Cascades "heats" up early and then cool off





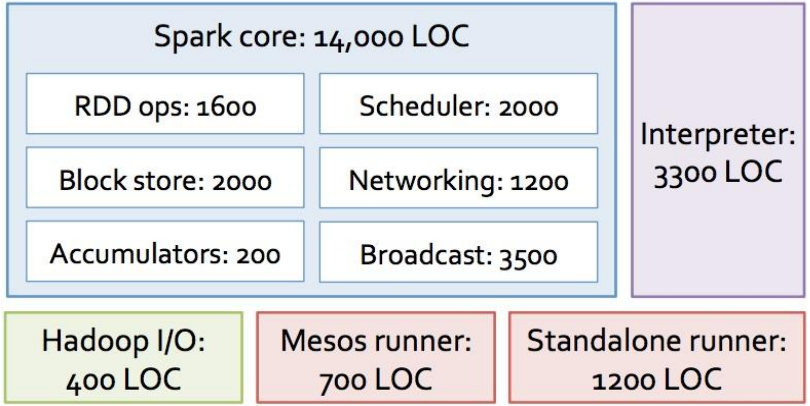


Subjectivity of the child and the parent are correlated. Sentiment flows!

There's change: Spark growth

Core:

- **2010: 14,000 LOC**
- **2012: 20,000 LOC**
- **2014: 50,000 LOC**
- **2016: 70,000+ LOC**
    - including new libraries: 300,000+ LOC



What is Spark?

- A distributed in-memory compute system
- Can use Hadoop/YARN
- Uses the Map Reduce paradigm

  Load -> Split -> Map -> Partition -> Shuffle-sort -> Reduce -> Output

  - Read/write to HDFS
  - Uses Hadoop IO (input and output formats, writables)

What is Spark?

- High-level functionality (joins, aggregates, group by, filter)
- Amazing job choreography
  - MR2 can only execute two tasks, in order: Map and Reduce
  - Spark can create a jobs executing many data transformations
  - While a complex problem might require several MR2 jobs, Spark can execute the same problem in one job.
- Spark’s succinct code can represents complex jobs
  - No need for Cascading or Oozie

MR2 tasks vs Spark transformations

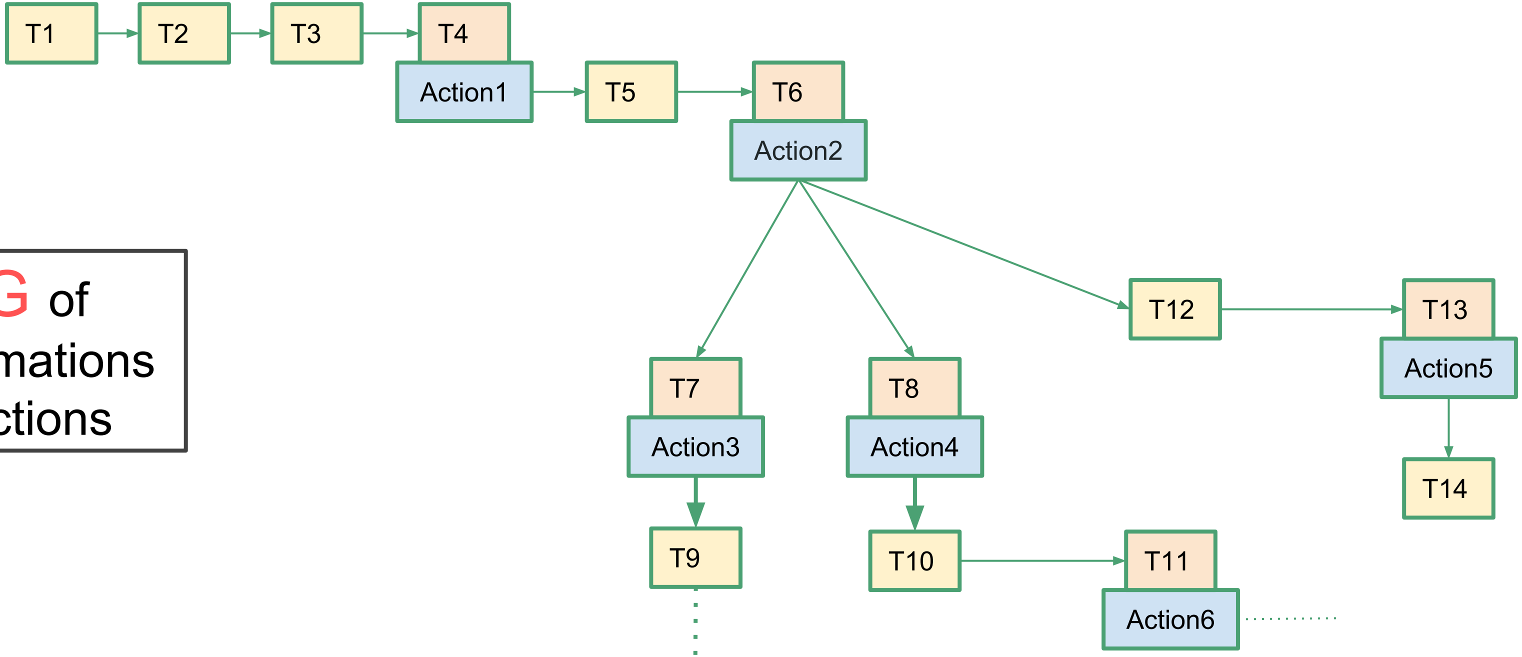
MR2 executes each job with just two tasks: MapTask and ReduceTask

Spark executes a job with many transformations.

- Narrow transformations are like MapTasks - they do not involve shuffle-sort
- Wide transformations are like ReduceTasks - they trigger a shuffle before the transformation begins.

Use case - Twitter

The processing pipeline for a Spark application with transformations (T) and Actions.



DAG of transformations and actions

More about Spark’s allure

- Solves the hard problems
    - Easily moves data from one MR job to another
    - Shuffles in-memory
    - Caches variables and recycles JVMs for tasks
- Solves user problems
    - Multiple languages
        - Python and Scala shell
        - Applications in Java, Python and Scala
    - Well-integrated with Spark R, Spark SQL, MLlib and GraphX
    - Easy data loads, powerful keywords, **concise**

Example: Word count (written in Scala)

```scala
val text = sparkContext.textFile("file:///...")

val words = text.flatMap(line => line.split(" "))

val wordPairs = words.map(word => (word, 1))

val counts = wordPairs.reduceByKey(_ + _)

counts.saveAsTextFile("hdfs:/...")
```

load a data file

parse each line into words

create <key, value> pairs with value = 1

sum the values for each key

ACTION: save the results

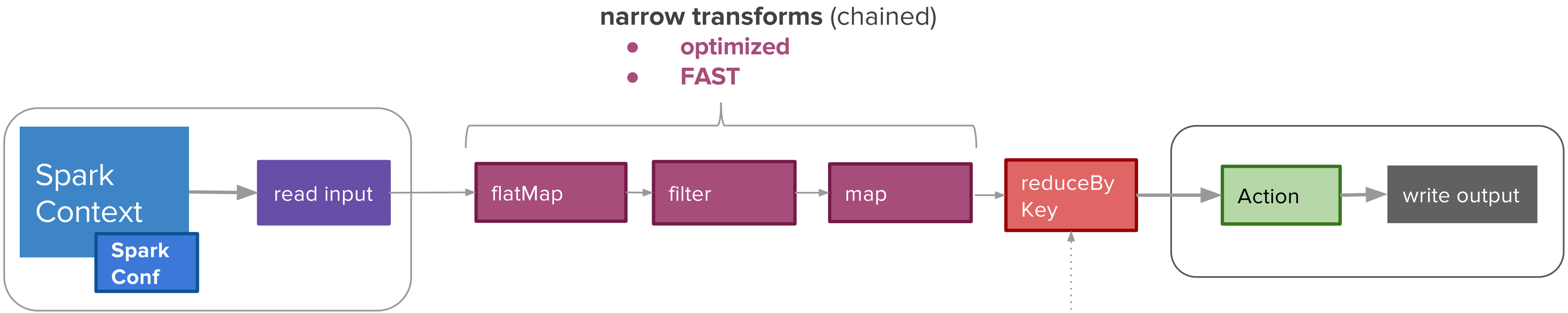
Counting the number of times “love” appears in Shakespeare’s poetry

```
> val lines = sc.textFile(“file:///home/cloudera/datasets/shakespeare/poetry”)

> val words = lines.flatMap(line => line.split(“\\W+”))

> val loveWords = words.filter(word=> word.contains(“love”))

> val wordCount = loveWords.map(lword=> (lword, 1))

> val counts = wordCount.reduceByKey(_+_)

> counts.collect()
```

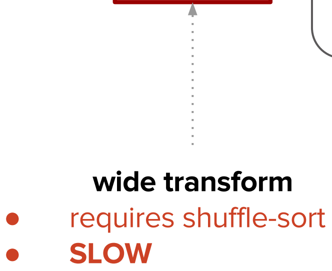
we could write this in one Mapper in MR2

this requires a shuffle before we start

Spark workflow for “love” count



spark context is automatically started in `spark-shell`



At home practice

1. Start your VM
2. Open a terminal
3. Start the spark REPL:

```
$ spark-shell
```

4. Run the example by typing this

```
scala> val lines = sc.textFile("file:///home/cloudera/datasets/shakespeare/poetry")

scala> val words = lines.flatMap(line => line.split("\\W+"))

scala> val loveWords = words.filter(word => word.contains("love"))

scala> val wordCount = loveWords.map(word => (word, 1))

scala> val counts = wordCount.reduceByKey(_+_)

scala> counts.collect()
```

Stopping

5. stop the context

```
scala> sc.stop()
```

6. stop the REPL

```
scala> :quit
```

RDD - the core class of Spark

- compile-time type-safe
- lazy…
  - transform operations describe how to change the data
    - **map, filter, join, reduceByKey**
  - action operations actually start the processing and produce output
    - **take, count, first, foreach, collect**
- based on the Scala collections API
- most Spark RDD operations == Hive and MapReduce functionality

RDDs in the example

> val lines = sc.textFile(“file:///home/cloudera/datasets/shakespeare/poetry”)

lines is a file-backed RDD

> val words = lines.flatMap(line => line.split(“\\W+”))
> val loveWords = words.filter(word => word.contains(“love”))
> val wordCount = loveWords.map(word => (word, 1))
> val counts = wordCount.reduceByKey(_+_)

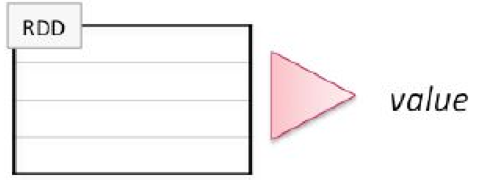
words, **loveWords**, **wordCount** and **counts** are RDDs derived from lines

> counts.collect()

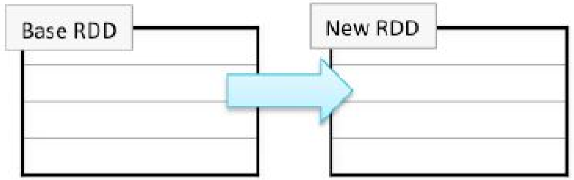
collect() is an action - it starts processing and prints out **counts**

RDD operations

- **Two types of RDD operations**
  - Actions – return values



  - Transformations – define a new RDD based on the current one(s)



- **Pop quiz:**
  - Which type of operation is `count()`?

Learning Spark

- Learn RDDs - focus on depth, learn the “how” of the processing
- Learn DataFrames and Datasets - focus on “use”, best for analysis
- Focus on Spark and Spark Streaming - for dataops
- Focus on Spark ecosystem (MLlib, SparkR, SparkSQL) - for analysis

Job execution

How the “count love words” example is processed

Job scheduling and execution

Very broad overview

Much more detail here - current, free book

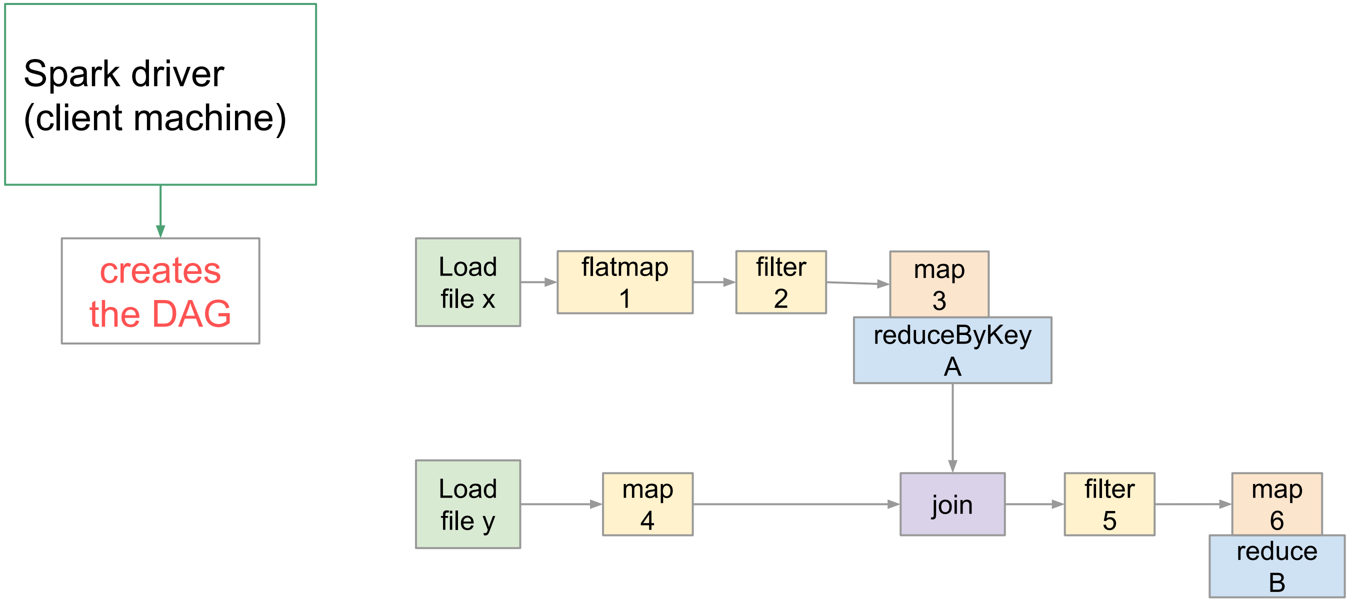
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-dagscheduler.html

Spark has a planning and execution engine

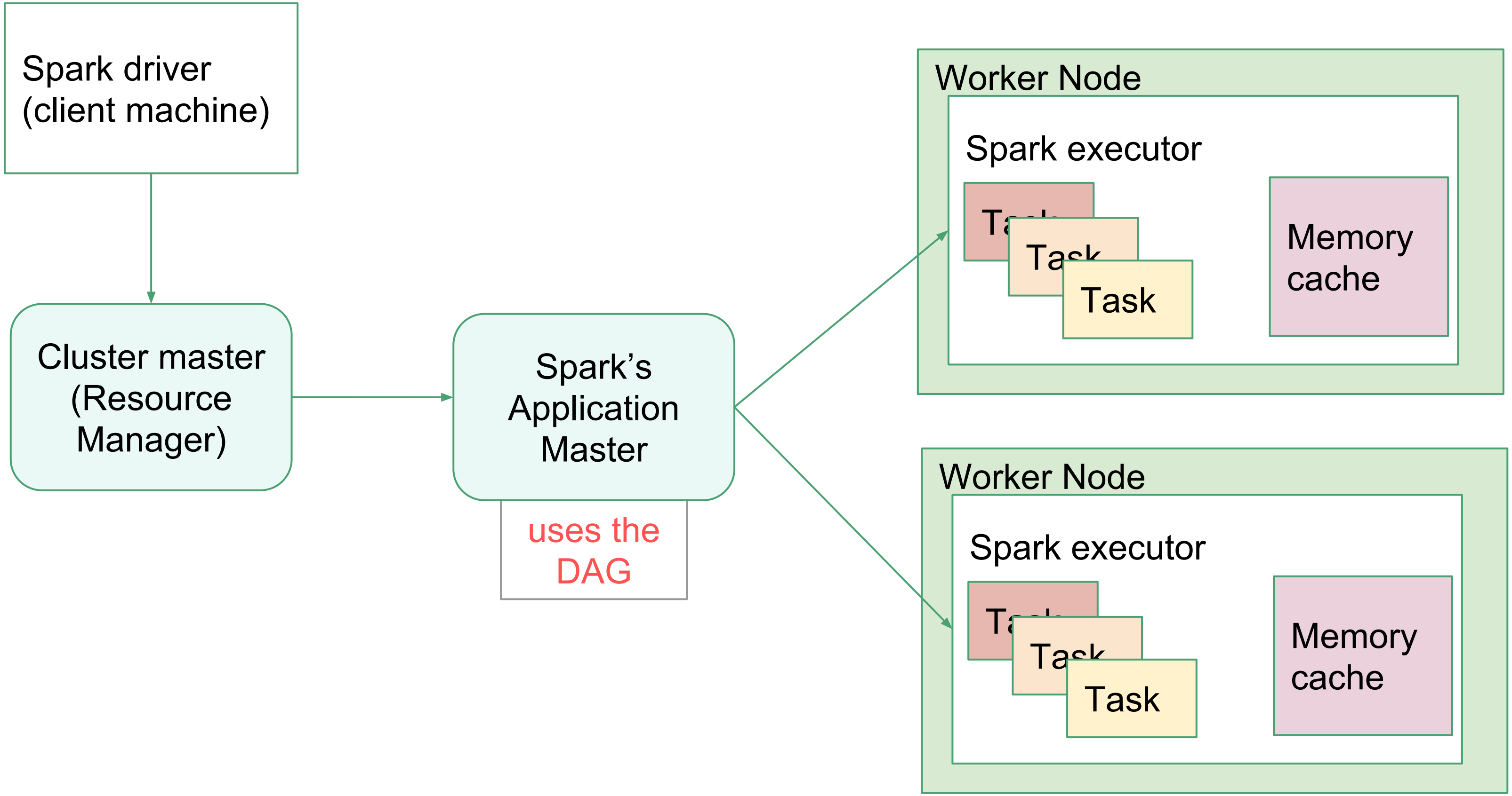
- Creates a plan for the job
- The job is executed in stages
- Narrow transformations are optimized (chained)
  - run on the same executor
  - use the same data
  - process the data line-by-line through the whole chain

Execution is managed by the DAGScheduler

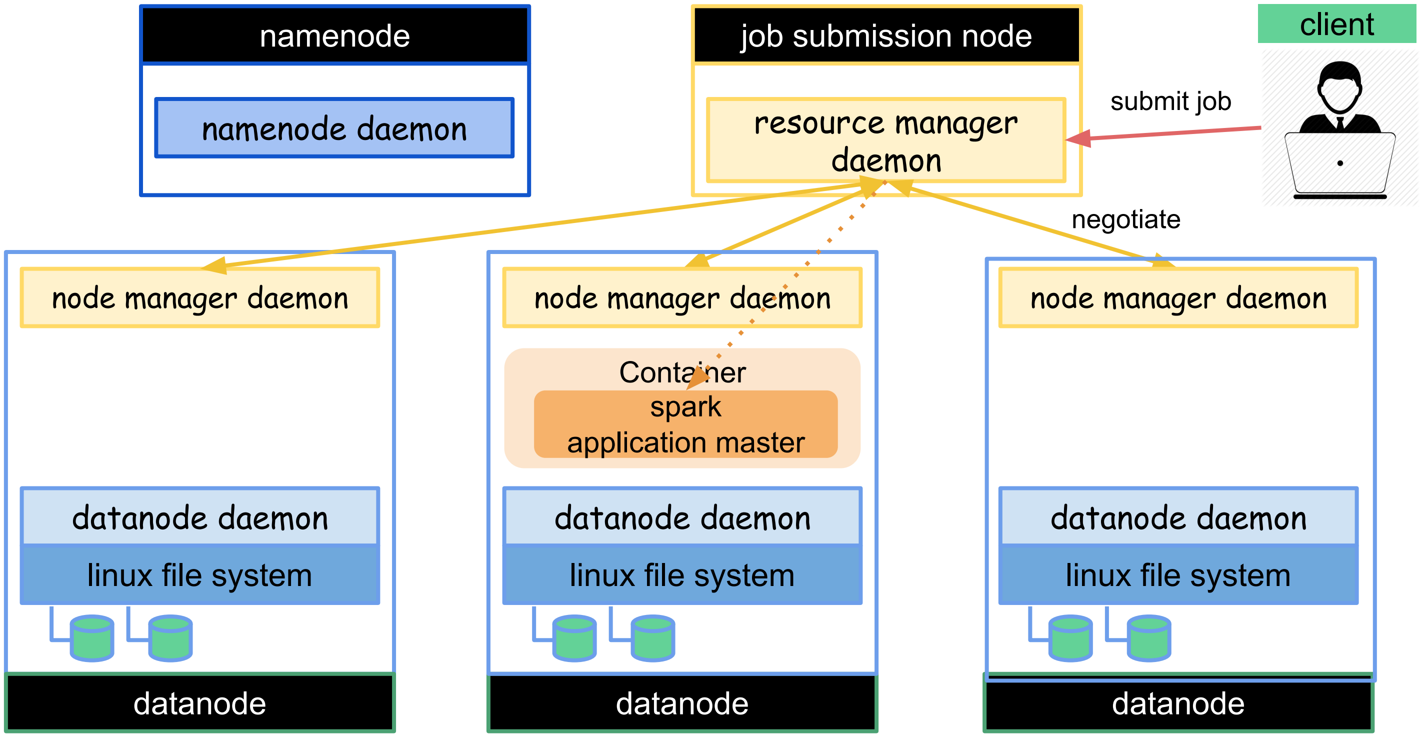
Spark execution(1)



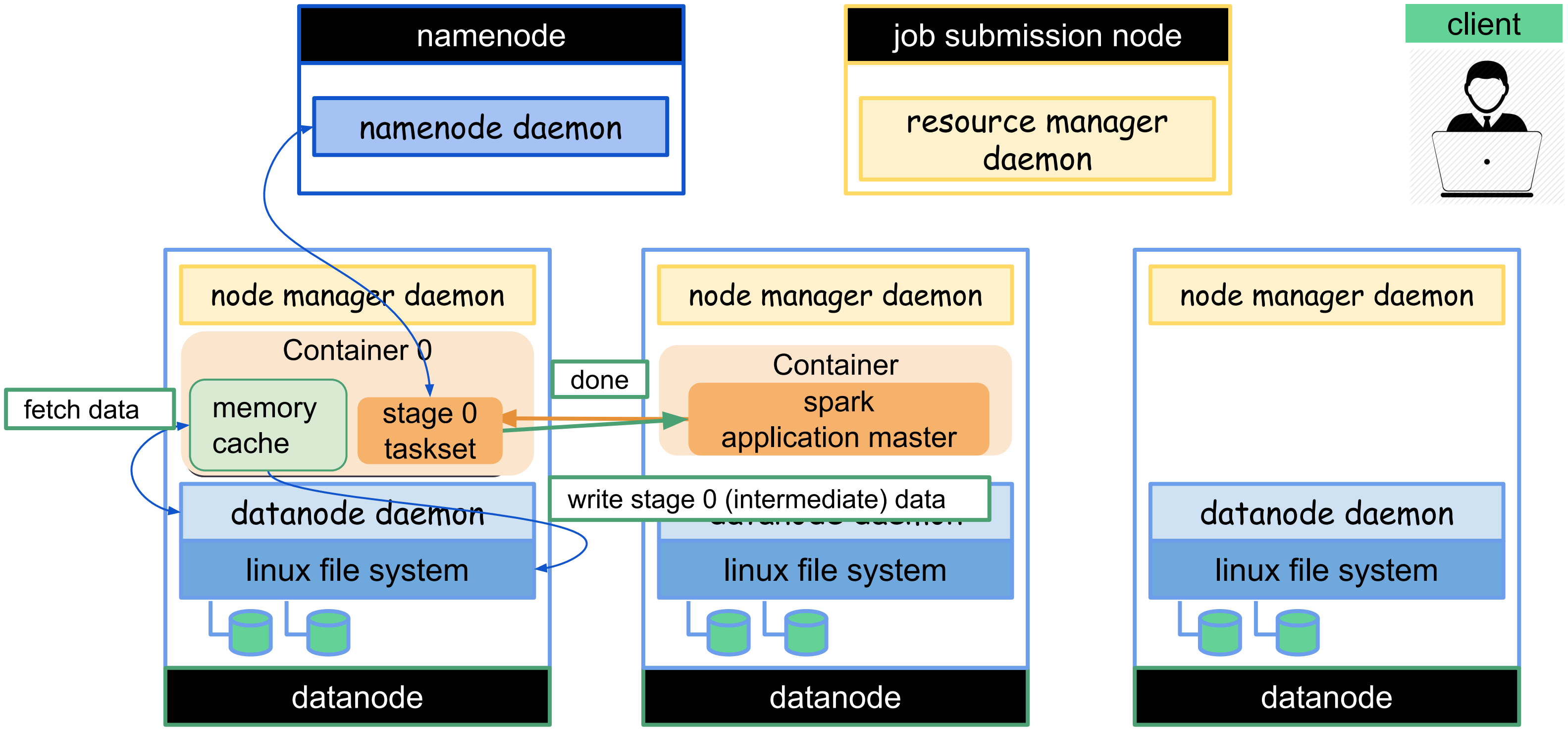
Spark execution(2)



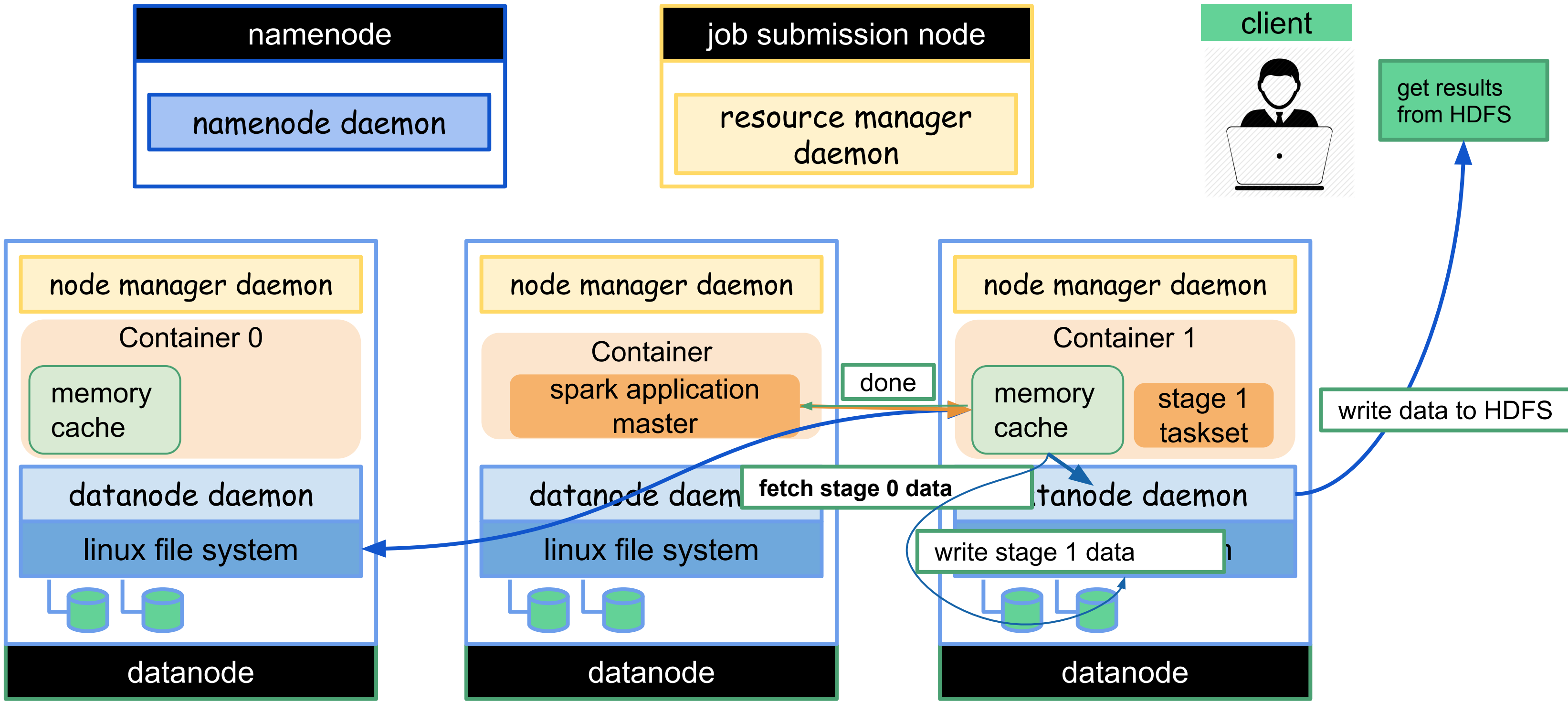
Running a 3-stage Spark job on YARN



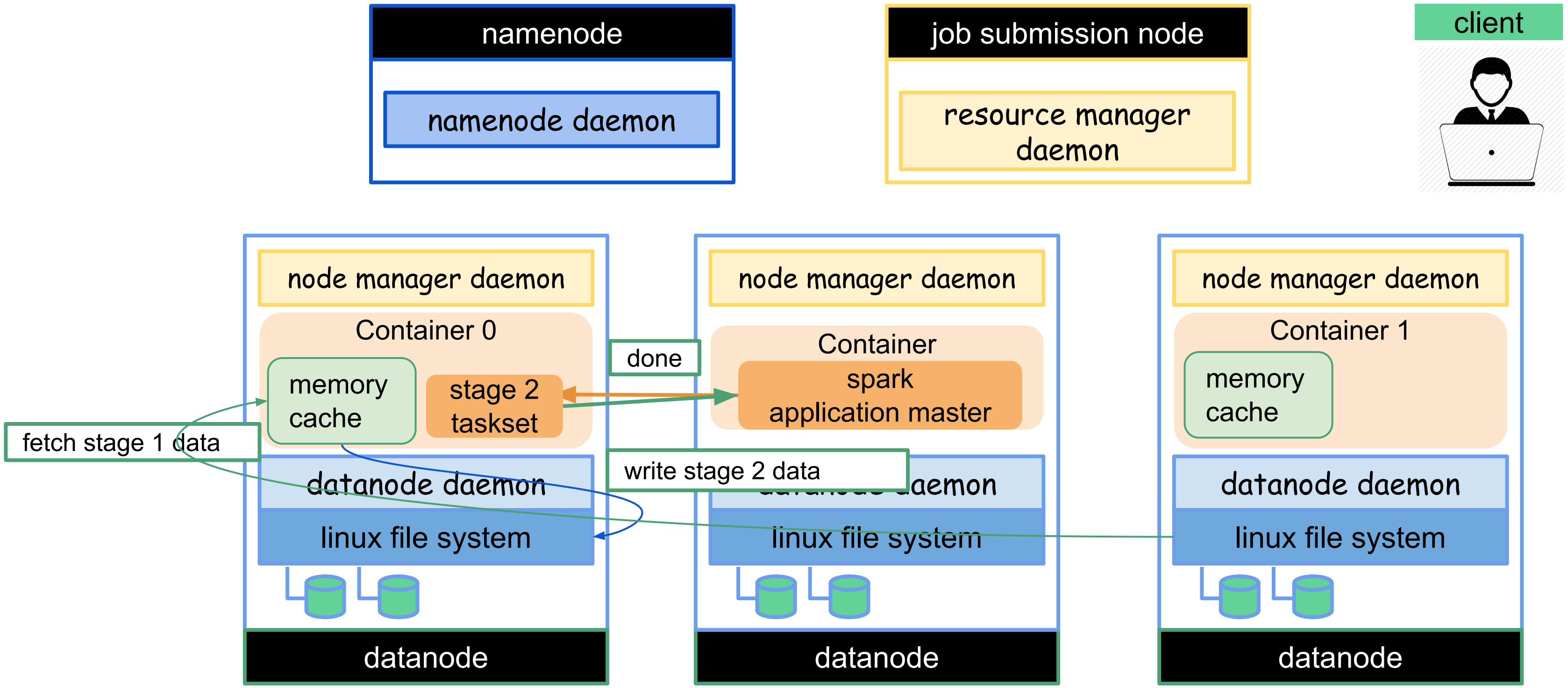
Running a 3-stage Spark job on YARN



Running a 3-stage Spark job on YARN



Running a 3-stage Spark job on YARN



So, Spark jobs run in stages?

Stage 1

textFile - read line from file

flatMap - parse line into words

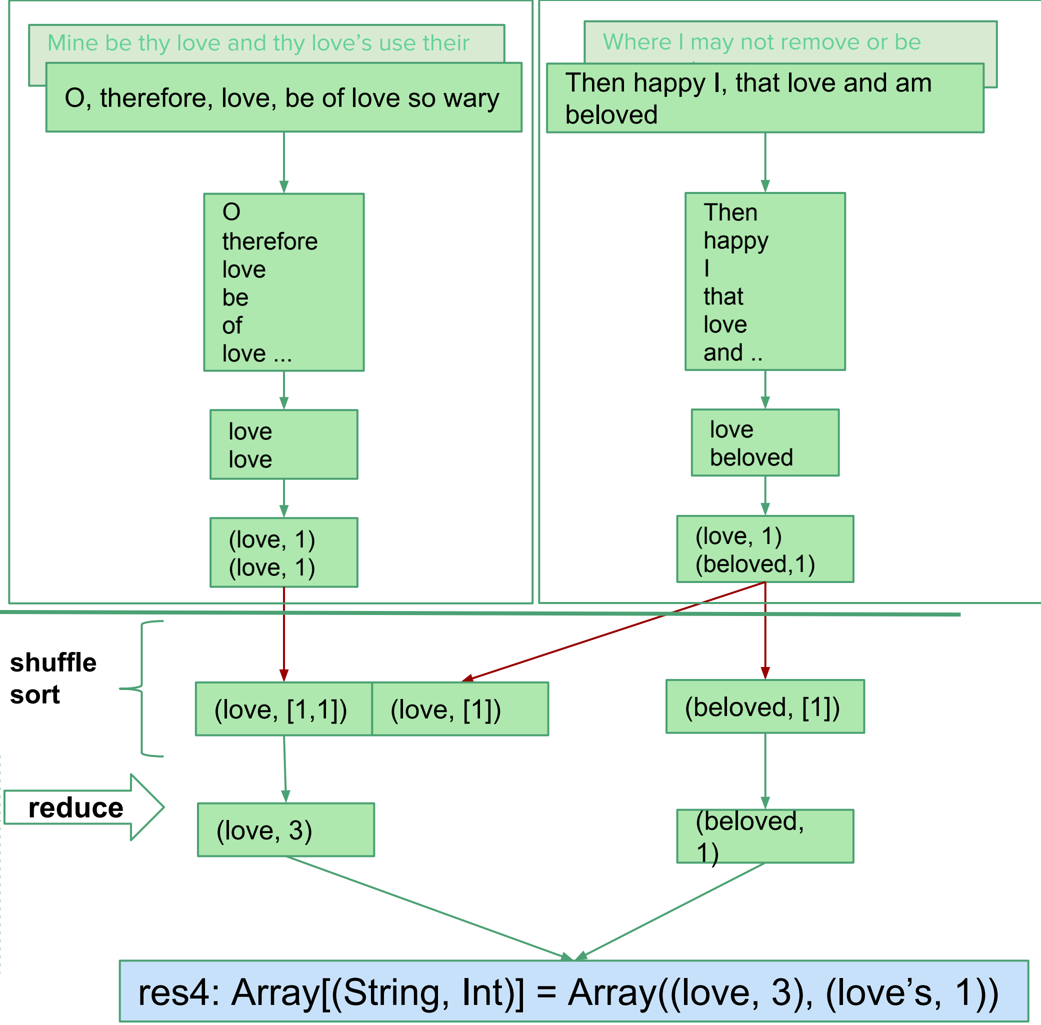
filter words containing "love"

map lwords to (lword, 1) pairs

Stage 2

reduceByKey to get counts

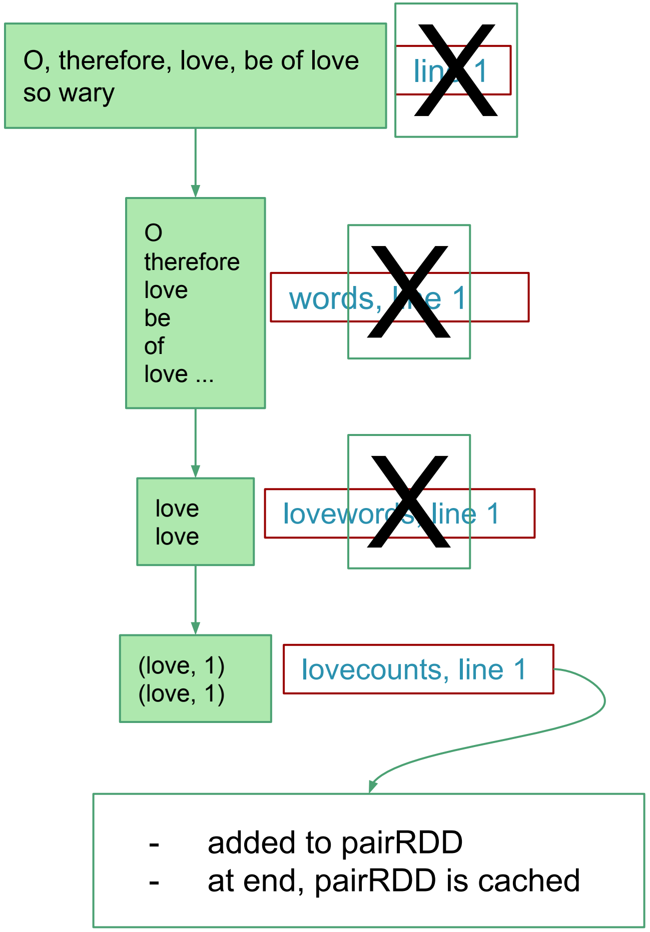
collect to start processing (!)
- data is returned as an array



Stage 1: a chain of narrow transforms:

textFile -> flatmap -> filter -> map

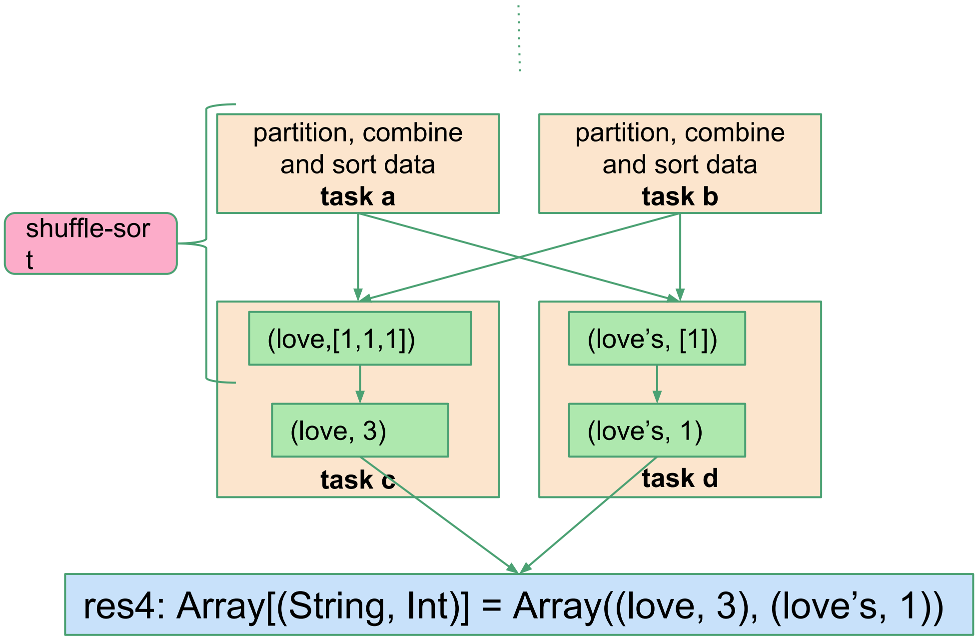
- Assign an executor to each input split
  - executes the chain of transforms
  - each line in a split is processed thru the chain
  - optimizes to chain to look/act like a single task
- Intermediate results (RDDs) are NOT stored
  - Records process consecutively
  - Each record is fed through all transforms
  - This is called “pipelining”
- At the end of a stage, the data is cached.
  - If we want to “save” an intermediate RDDs, use <name of rdd>.cache()



Stage 2:

- **a wide transformation: *reduceByKey***
  - *performs shuffle-sort*
  - *runs the ReduceByKey function*

- at end, **collect** *prints results:*
  - outputs an iterator (e.g. for an array)
  - used for small results



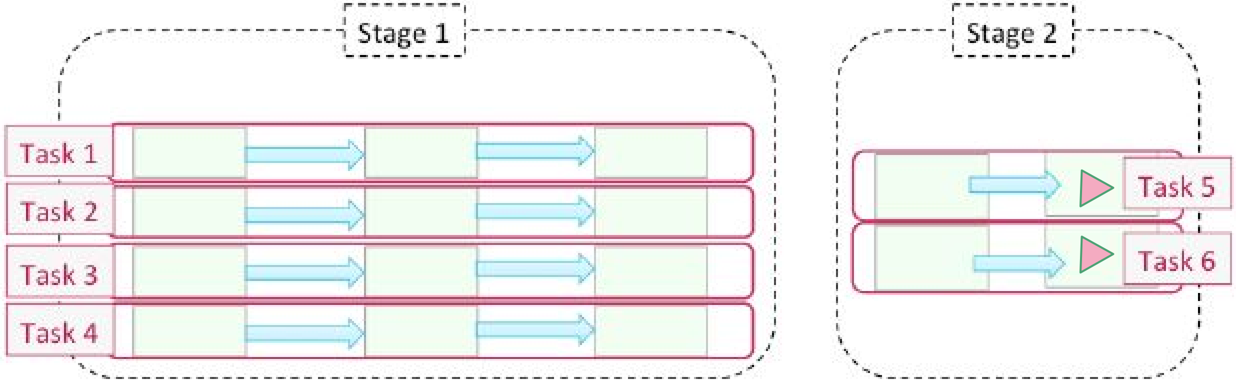
Summary

```
val lines = sc.textFile("/home/cloudera/datasets/shakespeare")
val words = lines.flatMap(line => line.split("\\W+"))
val loveWords = words.filter(word=> word.contains("love"))
val loveCounts = loveWords.map(lword=> (lword, 1))
```

Stage 1

```
val counts = loveCounts.reduceByKey(_+_)
counts.collect()
```

Stage 2



That collect() action - huh?

- collect() is an **action**
  - *starts* the processing by contacting DAGScheduler
    - acts like: job.waitForCompletion() in MapReduce
  - DAGScheduler is an event planner
    - everything is planned and ready to go
    - just needs to be told when to start the party
  - Example: “counts.collect()” was the **action** that started the job
    - Remember, you saw processing after “counts.collect()”

Important differences: traditional MR and Spark

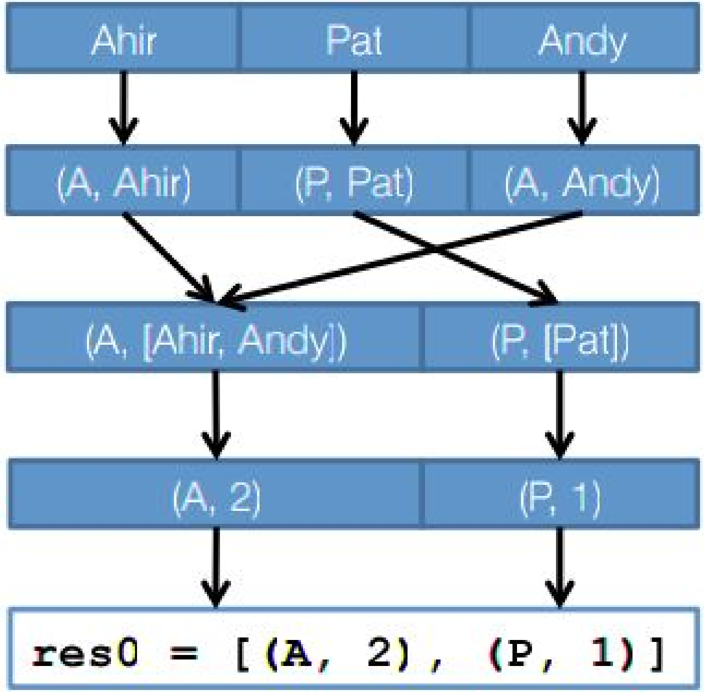
- Multiple stages - not just Map and then Reduce
    - Example: Map (stage 0), Reduce (stage 1), Map (stage 2)
- Memory cache
- Container (JVM) reuse
- Intermediate data contains serialized RDDs on disk

Yet another coding example

Another pass at computing in Spark

Find number of distinct names per “first letter”

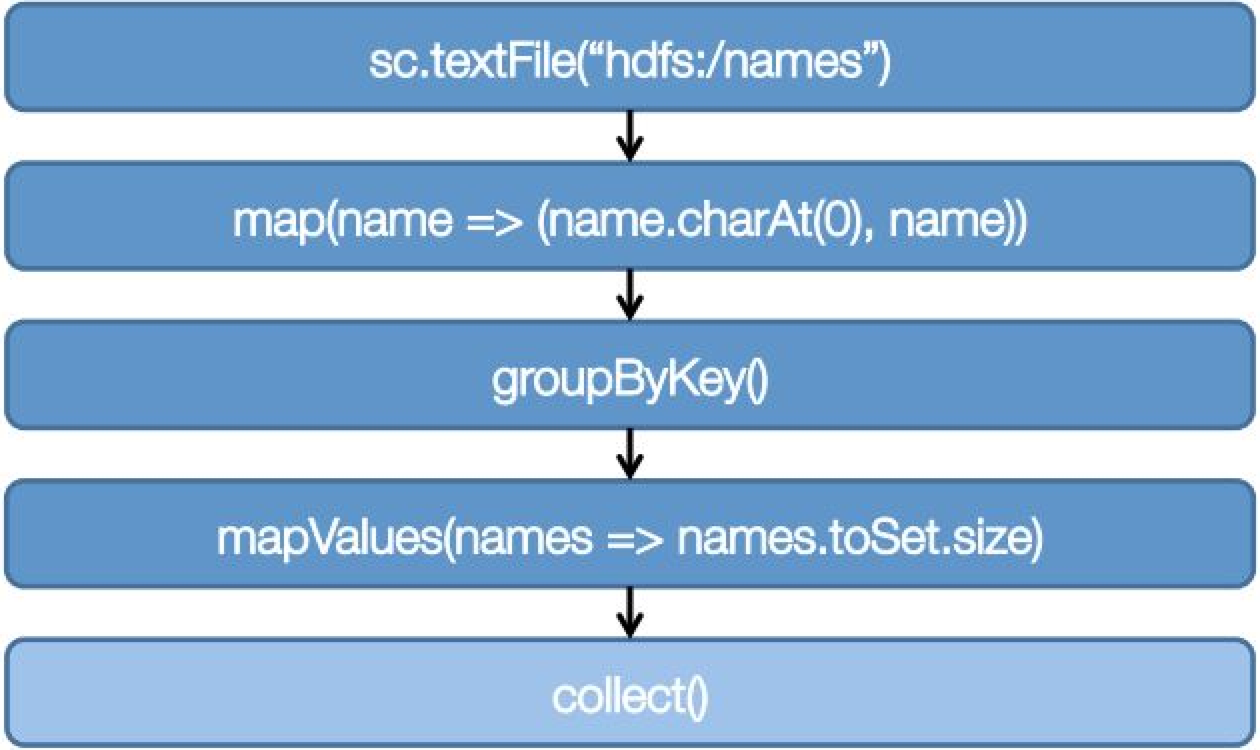
```
sc.textFile(“hdfs:/names”)

.map(name=>(name.charAt(0), name))

.groupByKey()

.mapValues(names=>names.toSet.size)

.collect()
```



Spark Execution Model

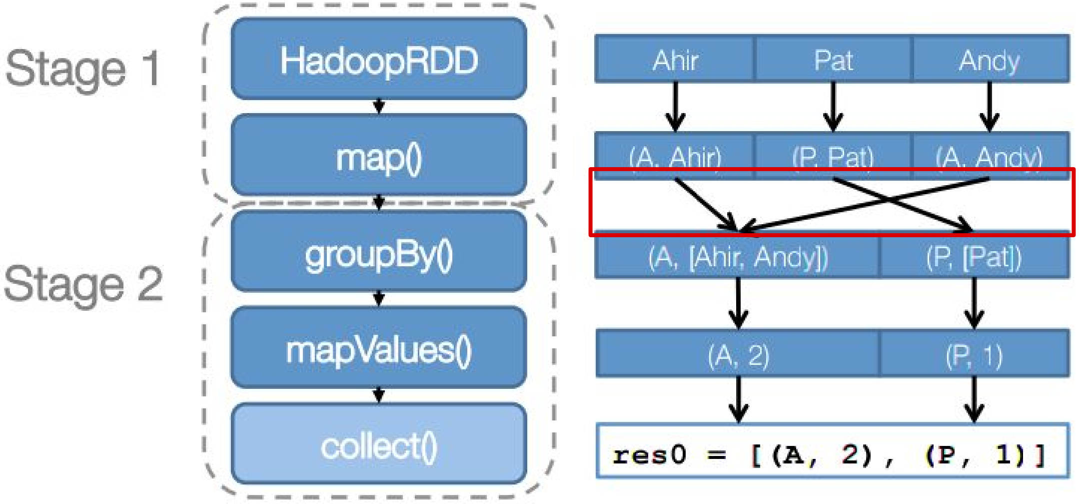
1. Create DAG of RDDs to represent computation
2. Create logical execution plan for DAG
3. Schedule and execute individual tasks

Step 1: DAG for job



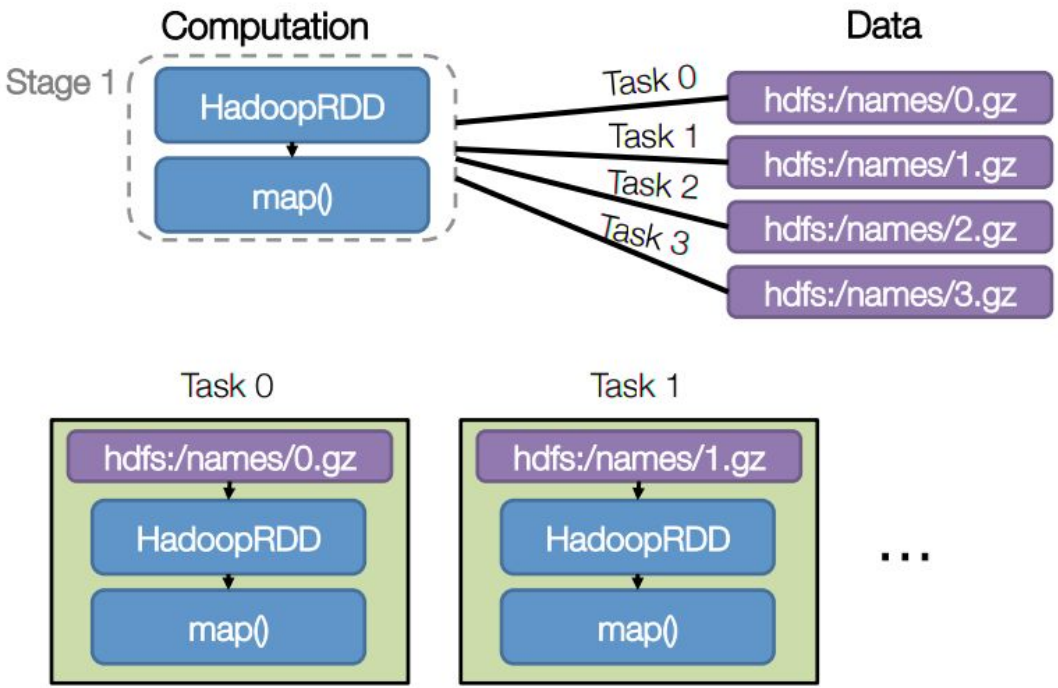
Step 2: Execution plan

- Pipeline as much as possible
- Split into "**stages**" based on need to reorganize data



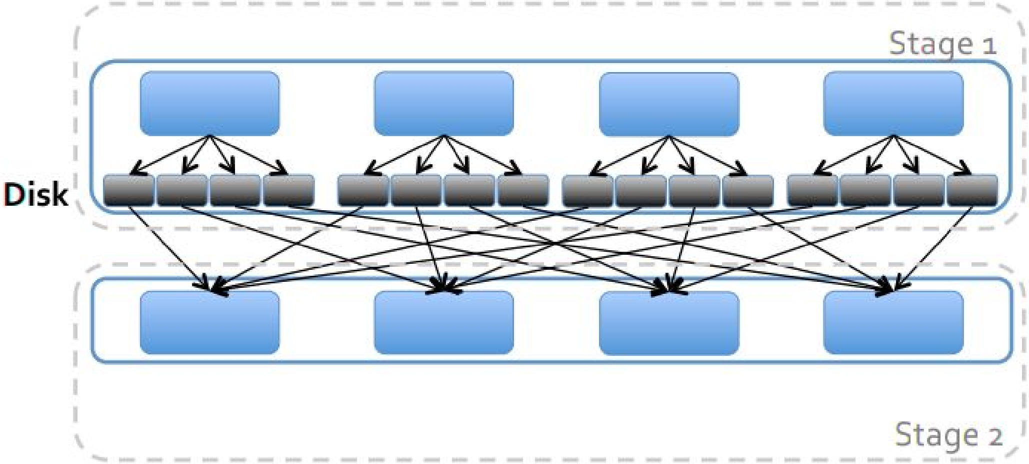
Step 3: Scheduling

- Split stages into tasks
- A task is data + computation
- Execute each task in a stage before moving on



Shuffle

1. Bucket up the data: Hash by key into buckets
2. Write buckets to disk
3. Pull bucket files to nodes used for stage 2



Optimizations

Problem: Ran the toSet() operation, to remove duplicates, at end

- probably want to cull the data earlier

Problem: May not be enough concurrency

- Need “reasonable number” of data partitions
- Commonly between 100 and 10,000 partitions
- Lower bound: At least ~2x number of cores in cluster
- Upper bound: Ensure tasks take at least 100ms

Revised code with optimizations

```
sc.textFile("hdfs:/names")
   .distinct(numPartitions = 6)
   .map(name => (name.charAt(0), 1))
   .reduceByKey(_ + _)
   .collect()
```

this is the SumReducer, in scala.

Original:

```
sc.textFile("hdfs:/names")
   .map(name => (name.charAt(0), name))
   .groupByKey()
   .mapValues { names => names.toSet.size }
   .collect()
```

Using Mapper setup and cleanup methods is a story for another day...

Hadoop Streaming

The MaxTemp MapReducer written in Python

Special Practice

On your VM’s Desktop, there is a folder called

“hadoop-streaming”.

This contains a SpecialPractice to help you learn

Hadoop Streaming

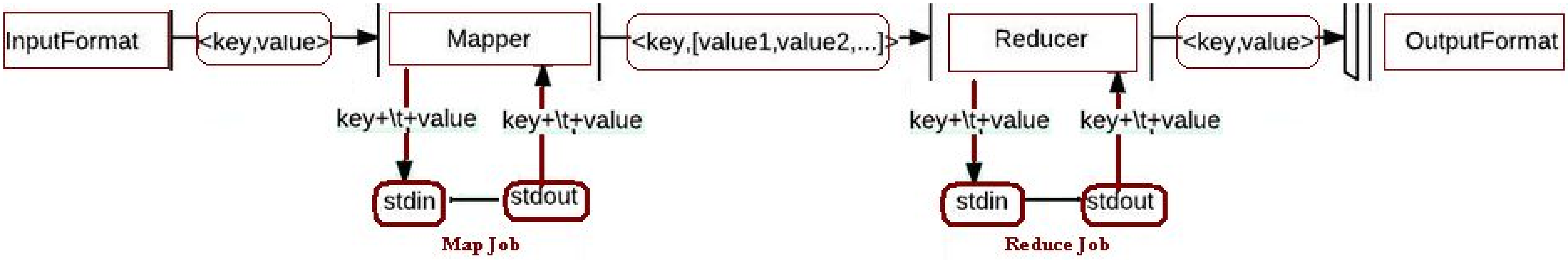
- **Features**
- **Example using Python**
  - **python mapper**
  - **python reducer**
- **How to run**

Hadoop Streaming: features

- **Run MapReduce using *any* language that can read from standard input and write to standard output.**

- **An important difference:**
    - **Hadoop MapReduce functions process one record at a time**
    - **Hadoop Streaming functions read from stdin and control the read process.**

How it works:
Streaming calls code from Mapper or Reducer



hadoop streaming: Python mapper

```
import re
import sys

for line in sys.stdin:
    val = line.strip()
    (year,temp,q)=val[15:19], val[87:92], val[92:93])
    if (temp != “+9999 and re.match(“[01459]”, q)):
        print “%s\t%s” % (year, temp)
```

hadoop streaming: Python Reducer

```
import sys

(last_key, max_val) = (None, -sys.maxint)
for line in sys.stdin:
        (key,val) = line.strip().split(“\t”)
        if last_key and last_key != key:
                print “%s \t %s” % (last_key, max_val)
                (last_key, max_val) = (key, int(val))
        else:
                (last_key, max_val) = (key, max(max_val, int(val)))
if last_key:
        print “%s \t%s” % (last_key, max_val)
```

hadoop streaming: running the job

```
$ hadoop jar /usr/lib/hadoop-<version>-mapreduce/\
contrib/streaming/hadoop-streaming-<version>.jar \
-input inputDir -output outputDir \
-file pathToMapScript -file pathToReduceScript \
-mapper mapBasename -reducer reduceBasename
```

Hadoop supplies the jar for streaming

Example: running hadoop streaming with Python in the studentVM

```
hadoop jar /usr/lib/hadoop-0.20-mapreduce/\
contrib/streaming/hadoop-streaming-2.0.0-mr1-cdh4.2.1.jar \
-input shakespeare -output avgwordstreaming \
-file mapper.py \
-file reducer.py \
-mapper mapper.py -reducer reducer.py
```

Key Points

- **To write a Mapper and a Reducer**
  - **can use any language that reads and writes to stdio**
  - **code must iterate through input data**

- **To run with “hadoop jar”:**
  - **use the hadoop-*-streaming.jar**
  - **use the -mapper and -reducer flags**

Extra slides

job launch details