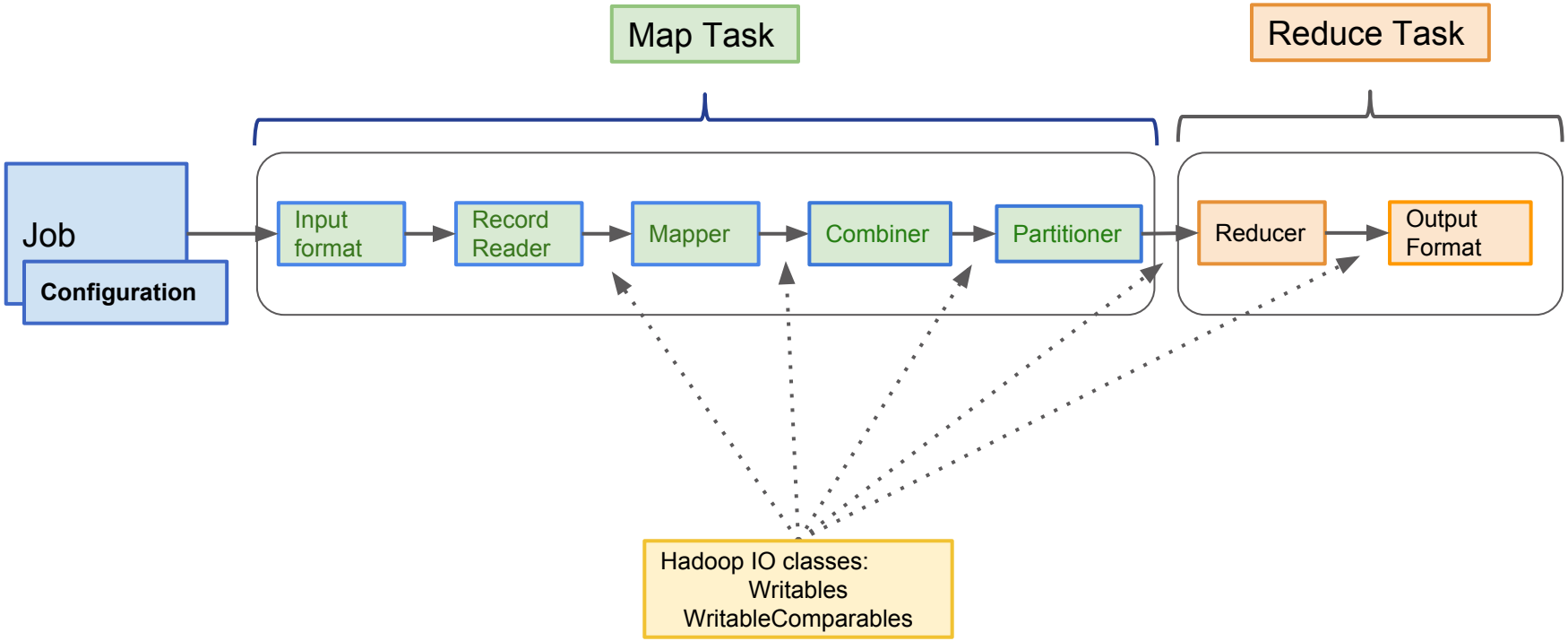
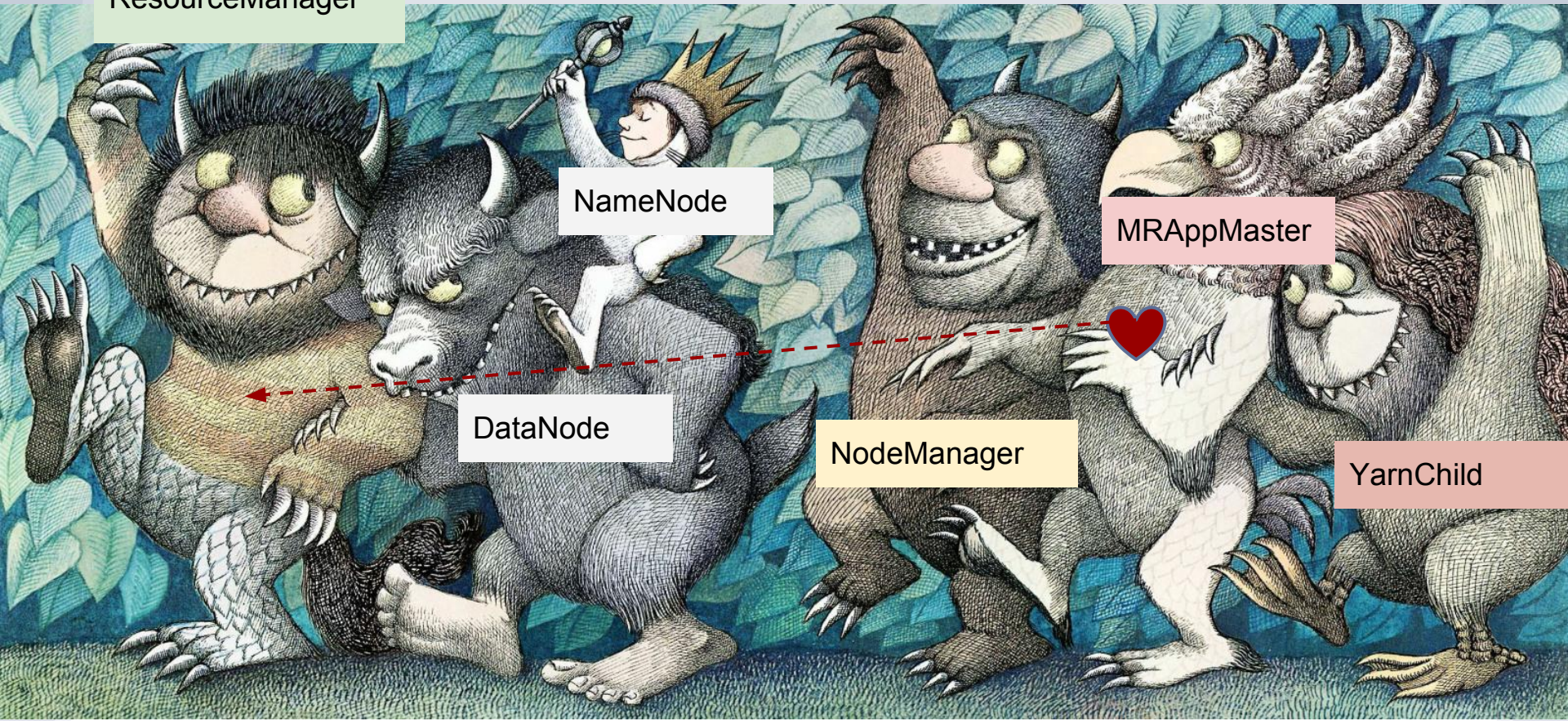
MapReduce Development

A stroll down the MapReduce API

Important MapReduce Classes

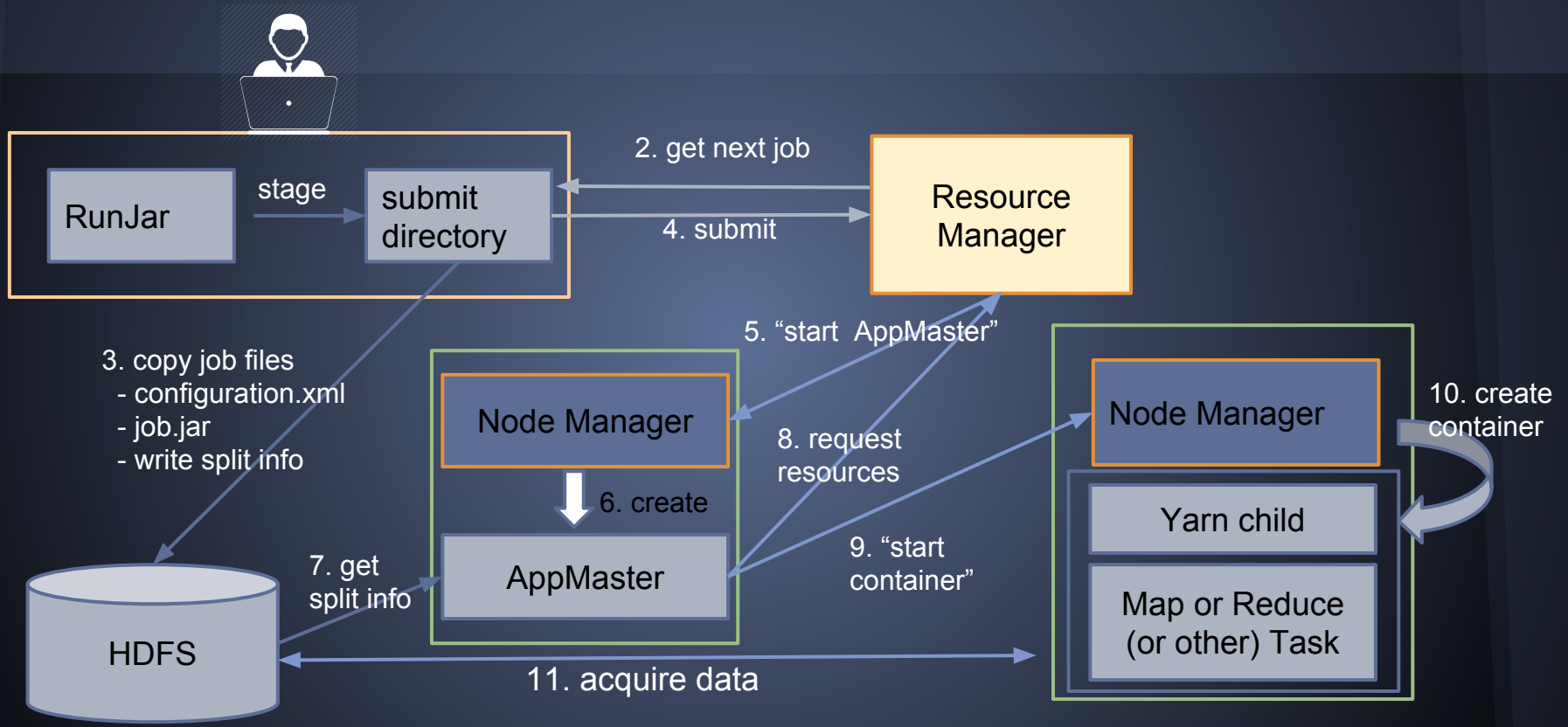



ResourceManager
NameNode
MRAppMaster
DataNode
NodeManager
YarnChild

Today’s topics

- Job submission (internals)
- Stroll down the processing line
  - ToolRunners
  - Input Formats and RecordReaders
  - Mappers and the Mapper methods: setup, run and cleanup
  - Partitioners - with an in-class exercise
  - Combiners
  - Reducers and the Reducer methods: setup, run and cleanup
  - OutputFormat and RecordWriters
- MRUnit testing - with an in-class exercise
- The Distributed Cache (or FileCache)

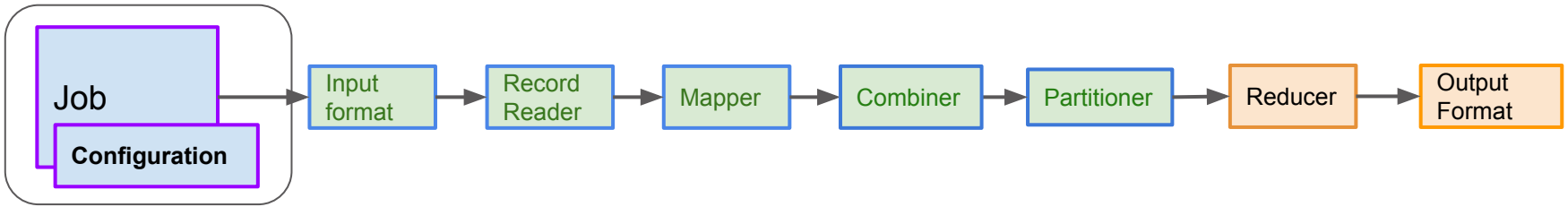
job launch details



Job launch - some key points

- **Programs are contained in a Java “jar” file and an XML file containing program configuration options.**
- **Submitting a job puts the job information into HDFS.**
- **Node Managers are told where to fetch the job info.**
- **The job info is fetched to all processing nodes before tasks start running.**
- **MapTasks acquire fetch data from HDFS as the *last* step in the job submission process.**

Job configuration via the driver



The driver - the class that submits a job

- **The Driver 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)

Configuration files

Default configuration (~1000 properties) and site specific (~200 properties)

- Site-specific files: core-site.xml, hdfs-site.xml, mapred-site.xml, yarn-site.xml
- Site-specific files are on VM - `/etc/hadoop/conf.pseudo`
- Default: core-default.xml, hdfs-default.xml, mapred-default.xml, yarn-default.xml

When running from command-line, see complete configuration using **Hue->JobBrowser->Job->Metadata**

(see cheat sheet)

Configuration properties

- In Eclipse, it can be difficult to see the configuration - the config files are embedded in jars.
- To view: use “Code for dumping the job’s configuration”
- See Lecture 3 cheat sheet.

Job configuration: Drivers

Previously, drivers were just main methods that

- Define a job's configuration
- Set context - input and output locations
- Submit the job.

The Tool driver does the same thing and also:

- Has a built-in parser for handling Hadoop specific flags
- Uses a ToolRunner

Implementing a driver as a so-called Tool is a best practice

ToolRunners parse flags on the commandline

```
$ hadoop jar myToolRunner.jar SectorCount \
-D sector=all <input> <output>
```

- **Used for specifying configuration values**
  - Use -D to set:
    - Hadoop configuration properties such as
      - mapreduce.job.reduces=12
      - mapred.sort.class=<the class used to sort keys>
    - User-defined properties and values
      - caseSensitive=true
  - Use -conf to supply a customized configuration file

Class examples (class-examples-1.zip)

Starting to study the stock market

- **data in companies/companiesListNASDAQ.csv**

We want to count the number of stocks are in a sector.

The first thing we are going to do is implement our code using Tool and ToolRunner...

```java
public class SectorCountWithSetupCleanup extends Configured implements Tool {

    private static final Log LOG = LogFactory.getLog(SectorCountWithSetupCleanup.class);

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

        Configuration conf = new Configuration();
        int exitCode = ToolRunner.run(conf, new SectorCountWithSetupCleanup(), args);
        System.out.println("Job completed with status:  " + exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {

        if (args.length != 2) {
            for (String arg : args)
                System.err.println(arg);
            System.err.println("Usage: -D sector=<sectorname|all> <in> <out>");
            System.exit(0);
        }

        Configuration conf = getConf();
        LOG.info("input: " + args[0] + " output: " + args[1]);

        LOG.info("---- counting stocks by sector ------- ");

        Job job = Job.getInstance(conf, "stock count");
        job.setJarByClass(SectorCountWithSetupCleanup.class);
        job.setMapperClass(StockSectorMapper.class);
        job.setReducerClass(StockCountReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]
                + "_" + Calendar.getInstance().getTimeInMillis()));

        boolean result = job.waitForCompletion(true);
        return (result) ? 0 : 1;
    }
```

Examples: using -D and -conf

Using -D to pass in an option

```
>> hadoop jar <mySnapshot>.jar stock.intro.SectorCountWithSetupCleanup \
 -D mapred.reduce.tasks=2 input output
```

Using -D to pass in a property you just created

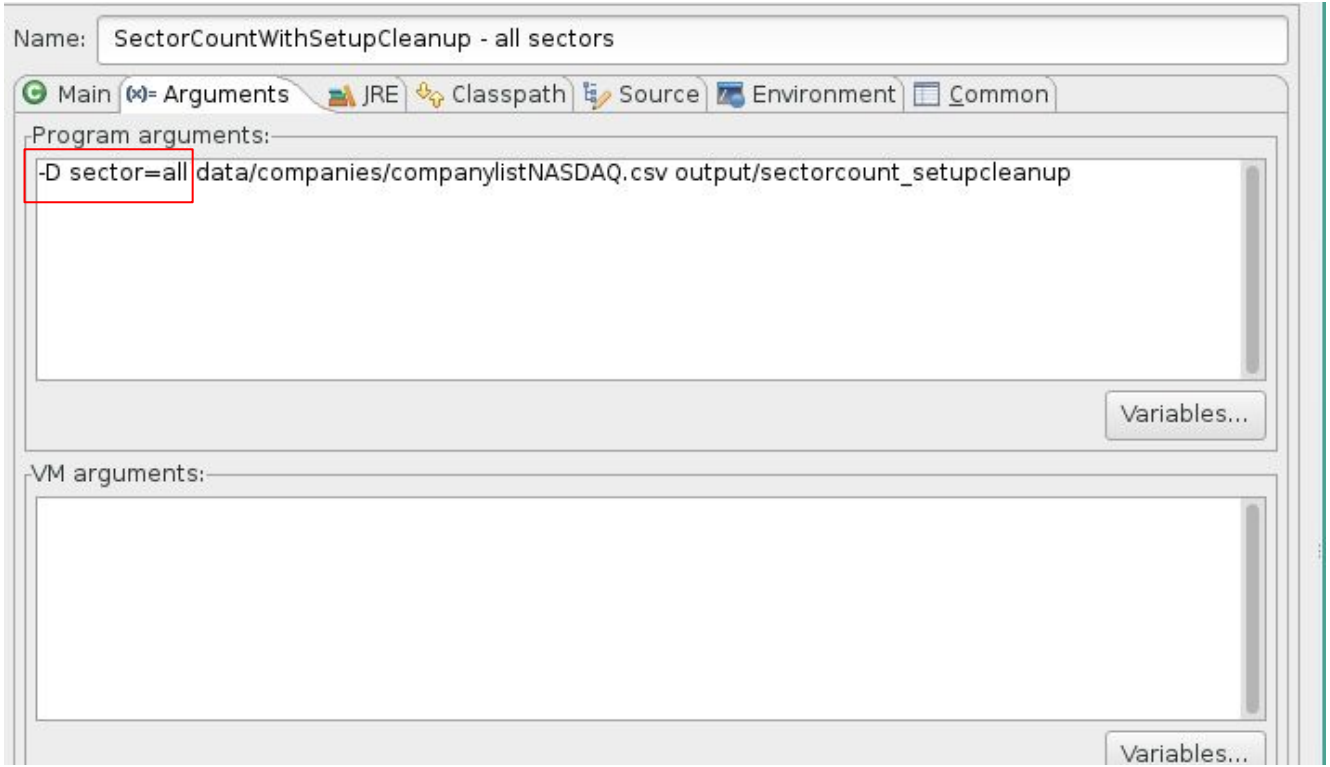
```
>> hadoop jar <mySnapshot>.jar stock.intro.SectorCountWithSetupCleanup \
 -D caseSensitive=true input output
```

Using -conf to pass in a config file

```
>> hadoop jar <mySnapshot>.jar stock.intro.SectorCountWithSetupCleanup \
-conf conf/hadoop-localhost.xml input output
```

To see code: GenericOptionsParser.processGeneralOptions(Configuration conf, CommandLine line)

In Eclipse, using Run Configurations

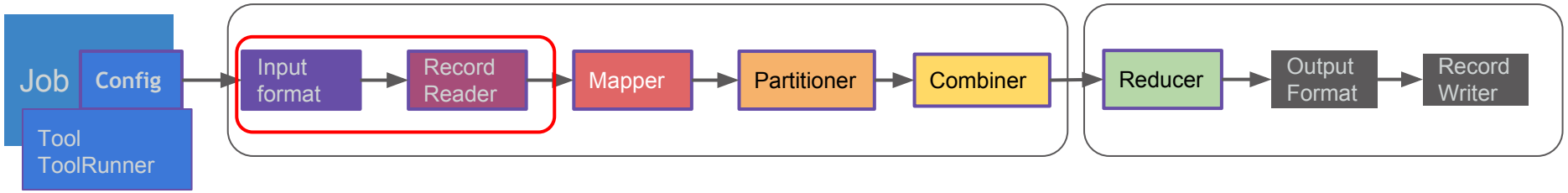


Recommended practice:

Practice 4: Using ToolRunner and Passing Parameters

*** Note: The practice is on Canvas under practiceForVM files ***

Input formats, data splitting and record readers



Specifying input locations for files

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

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

- **This will read the files in <dir>**
  - **Won't read files that start with "." or "_" (hidden files)**
  - **Can use wildcards to restrict input: /2010/*/Jan/***
  - **Note: <dir> can be a directory or a file**
  - **To read subdirectories, must use a regex. Example: shakespeare/***
- **To add multiple paths:**

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

InputFormat default

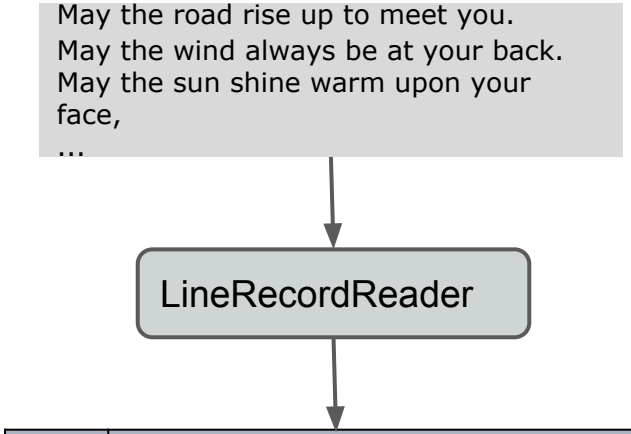
`TextInputFormat class`

- **Very general, frequently used**
- **To override the default, specify:**
  - `job.setInputFormatClass(<InputFormat>)`
- **`TextInputFormat`** reads text files; descends from **`FileInputFormat`**

The InputFormat's RecordReader

- **Most common: LineRecordReader**
  - output value = readline() string
  - output key is byte offset of line

- **Used by the default - TextInputFormat**



Key	Value
0	May the road rise up to meet you
32	May the wind always be at your back
67	May the sun shine warm upon your face

InputFormats and RecordReaders for text

- **FileInputFormat** - base class for file-based InputFormats
  - TextInputFormat (default)
    - **LineRecordReader**: Treats '\n'-terminated lines as a value.
  - KeyValueTextInputFormat
    - Parses delimited records as "key (delimiter) value".
    - Uses **KeyValueLineReader:** Uses default delimiter: tab.
  - WholeFileInputFormat
    - Whole files are read - each file is assigned to a MapTask
    - Key is number of bytes in file, Value is the file itself
    - Uses **WholeFileRecordReader**

InputFormats for structured data

- **SequenceFileInputFormats**
  - Binary files, (key, value) pairs with some additional metadata.
  - Useful for inputting binary data to Hadoop Streaming (Python)
  - Uses **SequenceFile readers**
- **AvroInputFormat**
  - Uses schemas to describe input and/or output files
  - Avro files just contain a sequence of values - therefore:
    - For Avro input files, Mapper must subclass **AvroMapper**
    - For Avro output files, Reducer must subclass **AvroReducer**
- **DBInputFormat - uses a DBRecordReader**
  - SQLStatement, creates a split based on rows in the statement.

Handling semi-structured text files

- **Read CSV or JSON files:**
  - **TextFileInputFormat**
    - **Read in with text file reader**
    - **Map over the values with a JSON or CSV parser.**

- **Reading XML files:**
  - **StreamingInputFormat**
    - Whole files are read - each file is assigned to a Mapper
    - Uses **StreamXMLRecordReader**
    - **Add Maven dependency to pom.xml**

```
<dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-streaming</artifactId>
        <version>2.5.0</version>
</dependency>
```

InputFormats and RecordReaders

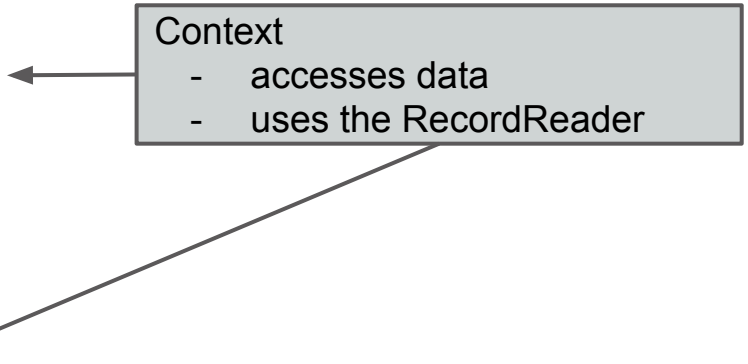
- `Input Format`
  - Contains the location of the input, often a file or directory.
  - Implements two methods:
    - **getSplits(): constructs splits and return information about where they are stored - and if they are in-memory.**
    - **createRecordReader(): creates the RecordReader for processing splits**

- `RecordReader`
  - Converts InputSplits into records
  - Parses records into <key, value> pairs
    - **Implements nextKeyValue(), getCurrentKey(), getCurrentValue()**

Processing records in a split

Application Master says:

```
MapTask.runNewMapper(){
        ...
        mapper.run(mapperContext)
        ...
}
```

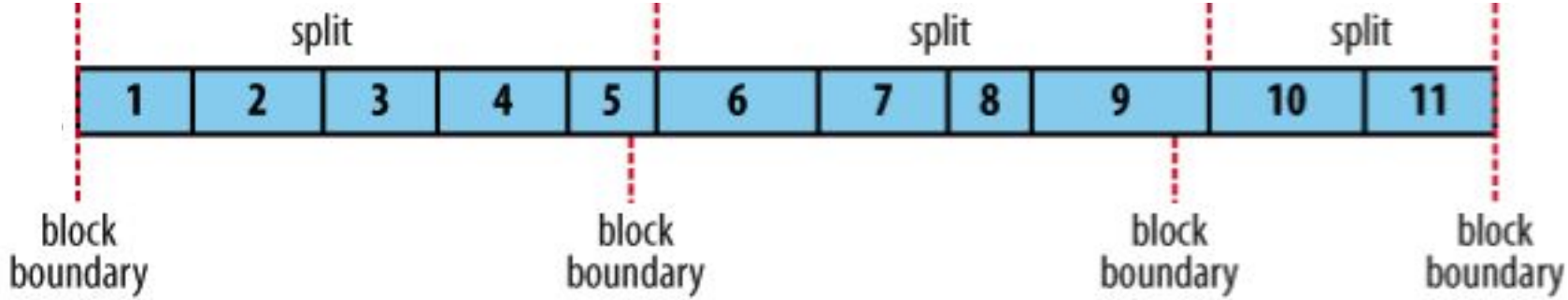


Mapper. run method:

```
Mapper.run(Context mapperContext) {
    while(context.nextKeyValue())
        map(context.getCurrentKey, context.getCurrentValue(), context);
}
```

Input Splits

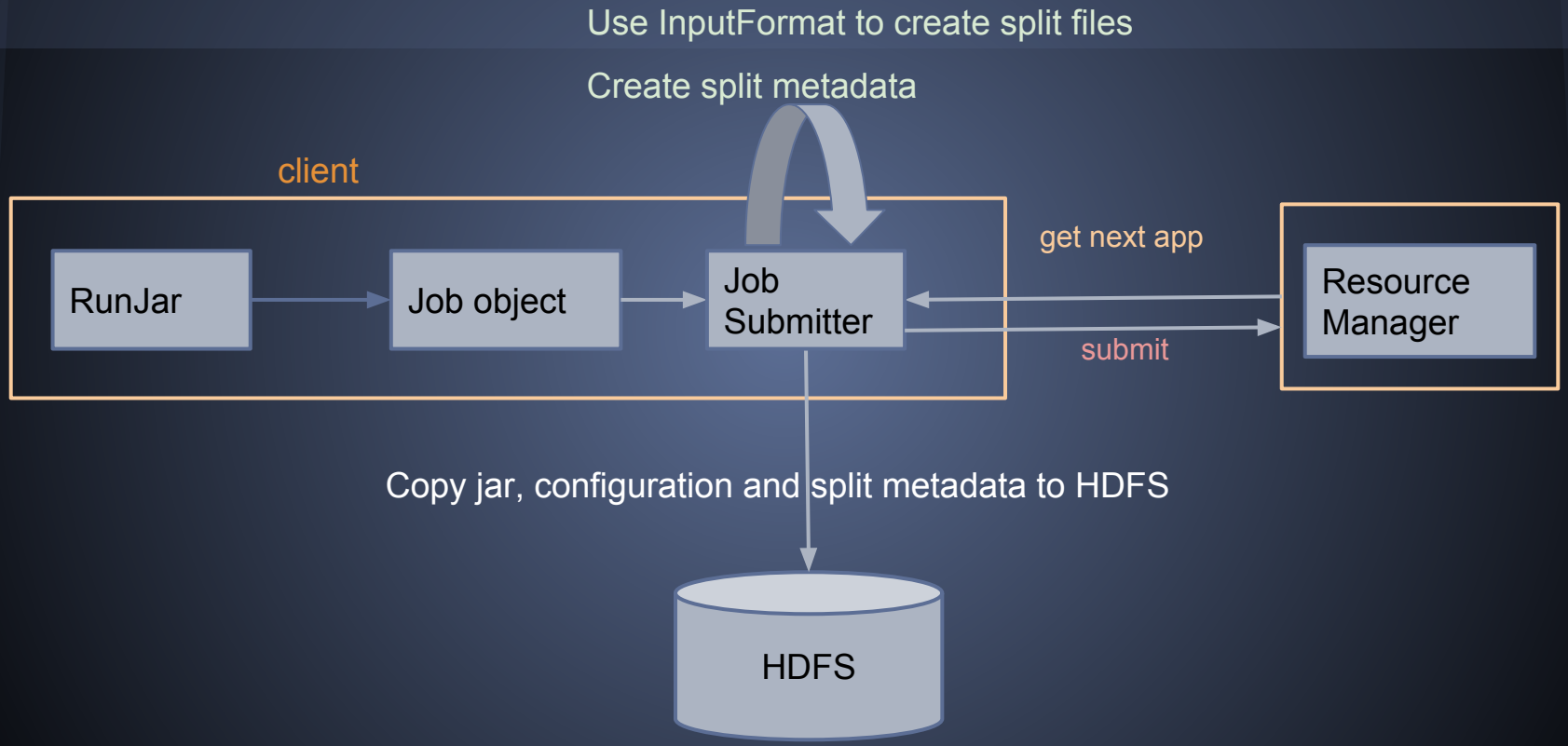
FileInputFormat.getSplits(): calculates splits from blocks in a file.



- A split provides whole records for the RecordReader.
- For FileSplits, split size >= block size
- Each Map Task processes one split
  - A split contains: host location, file location, position and length of the split
  - The RecordReader actually reads the split from the filesystem

Courtesy of Hadoop: The Definitive Guide, Tom White, 2012

JobSubmitter calculates splits



Is all this relevant to newer implementations? yes.

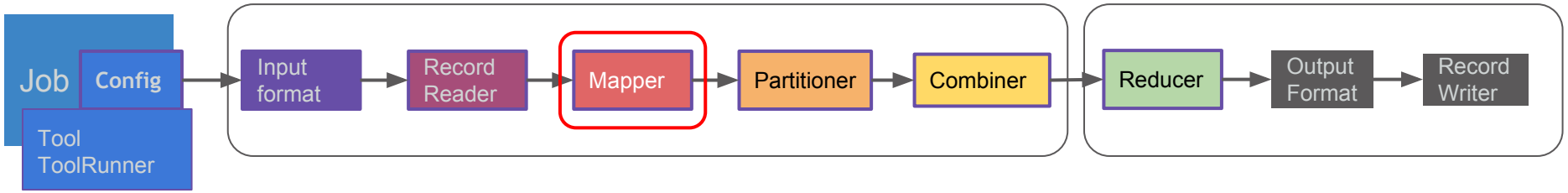
Spark internals: input functions in Spark wrap Hadoop InputFormats and Record Readers

```
def textFile(path: String, minPartitions: Int =
defaultMinPartitions): RDD[String] = {
   assertNotStopped()
   hadoopFile(path,
            classOf[TextInputFormat],
            classOf[LongWritable],
            classOf[Text],minPartitions)
            .map(pair => pair._2.toString).setName(path)
}
```

Flink internals: input functions in Flink wrap Hadoop MapReduce classes

For more on Spark input-output functions see: `http://www.bigsynapse.com/spark-input-output`

Mappers


Job
Config
Tool
ToolRunner
Input format
Record Reader
Mapper
Partitioner
Combiner
Reducer
Output Format
Record Writer

The Mapper - Review

- **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**

Last week we saw several examples

- **Upper Case Mapper -** transforming key-value pairs
- **Explode Mapper -** creating multiple key-value pairs from a single pair
- **Filter Mapper -** only output key-value pairs that pass the “filter”
- **Changing keys -** create and output a new keys
- **Identity Mapper -** the default Mapper (the parent)

Mapper for sector count

- **In our code, you'll see the regular expression for a record**

recordline in the NASDAQ company list files

"GOOG","GoogleInc.","523.4","$357.66B","2004","Technology","Computer Software", "http://www.nasdaq.com/symbol/goog"

- **You will also see the setup method used to read a configuration property.**

```
public static class StockSectorMapper extends Mapper<Object, Text, Text, IntWritable> {

    /*
     * Create holders for output so we don't recreate on every map
     */
    private final static IntWritable ONE = new IntWritable(1);
    private final static Text SECTOR = new Text();
    /*-
     * recordline in the NASDAQ company list files
     * "GOOG","GoogleInc.","523.4","$357.66B","2004","Technology","Computer Software", "http://www.nasdaq.com/syr
     *
     * regular expression for record:
     */
    private final static String recordRegex = ",(?=([^\"]*\"[^\"]*\")*[^\"]*$)";

    private final static int sectorIndex = 5;
    public final static String NO_INFO = "n/a";
    String sector = new String();

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

        String[] tokens = value.toString().split(recordRegex, -1);
        String str = tokens[sectorIndex].replace("\"", "");

        if (str.equals(sector) || sector.equals("all")) {
            SECTOR.set(str);
            context.write(SECTOR, ONE);
        }
    }

    @Override
    public void setup(Context context) {

        // pull the sector property from job's configuration object
        Configuration conf = context.getConfiguration();
        sector = conf.get("sector", "all");

    }

}
```

Things to remember about setup

setup is first phase of a Mapper or Reducer run

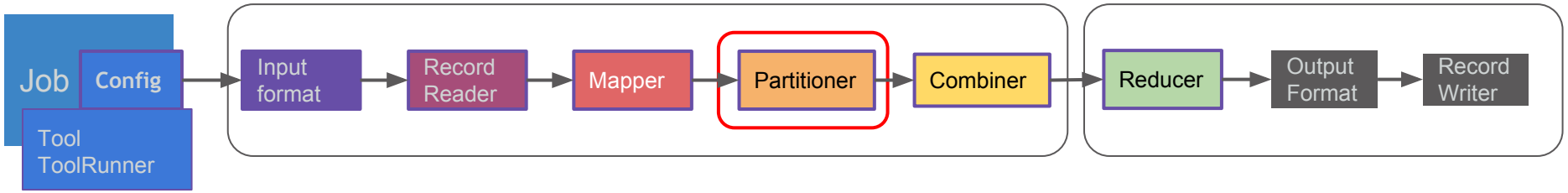
- Runs before any data is processed
- Use setup method to read in configuration properties
- Example reads the `sector` property and sets it to "`all`" if not found.

```
@Override
public void setup(Context context) {

    // pull the sector property from job's configuration object
    Configuration conf = context.getConfiguration();
    sector = conf.get("sector", "all");

}
```

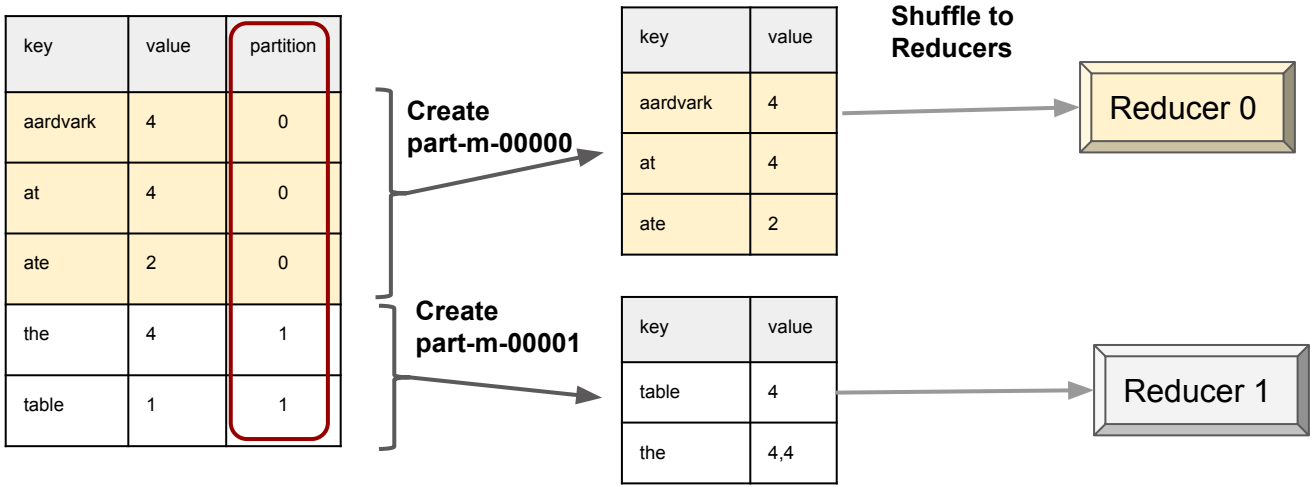
Partitioners and the number of Reducers



What does a Partitioner do?

- **The Partitioner assigns a number to each map-output key**
- **Partitioner has a method:**

```
int getPartition(inter_key, inter_value, num_reducers)
```



Partitioners

- getPartition function: returns an partition index for a key-value pair.
- The **default Partitioner** is the **HashPartitioner**

```
public class HashPartioner<K, V> extends Partitioner<K, V> {

        public int getPartition(K key, V value,int nReducers)  {
                return (key.hashCode() & Integer.MAX_VALUE) % nReducers;

        }
}
```

- Only rule: return a number between 0 and nReducers
- The number of partitions = nReducers

If you need strict order: TotalOrderPartitioner

Partitions data so that all keys which go to the first Reducer are smaller than any that go to the second, etc.

- Order is maintained when multiple reducers can be used.
- Results can be concatenated into a ordered list.
- How: Uses an externally generated file dictating how keys are to be split into partitions.

Partitioners: writing your own

- **Custom partitioners are necessary:**
  - Load balancing: if data is distributed unevenly across keys.
  - Within-key processing in the Reducer: e.g. secondary sorting.
- **You can partition by many methods*:**
  - by the mantissa of a key
  - by the first part of a multipart key
  - by any function of the key - or possibly (key, value) pair.

*** but there are a few rules you should follow.**

Partitioners: rules for writing your own

- **Rule: partition the data into same-size datasets.**
  - **Ignore if some keys must be processed by the same reducer.**
  - **May need to sample the data to understand distribution over keys.**
- **Do not ignore the number of reduce tasks.**
  - **If Partitioner gives an index > number reduce tasks: exception.**

how many reducers? (1)

Is there an optimal number?

- **Default is 1**
  - Can be useful if the final output must be completely sorted
  - Inefficient if Mappers produce a large amount of data
    - **not enough disk on Reducer's node for intermediate data**
    - **Reducer has long running time**
- **Some jobs have an implicit number**
  - If key is week day, then number of partitions = 7
  - If key is treatment, then number of partitions = number of treatment types (or diseases)

how many reducers? (2)

- **Balancing act:**
  - too few reducers ⇒ too much data for each
  - too many reducers ⇒ too much overhead creating and maintaining reduce tasks.
- **If there are more Reducers than resources available:**
  - A second ‘wave’ of reducers will run
  - Can double time in reducer phase
  - In this case, increasing the number of Reducers will cut down on the time it takes to run each wave.

how many reducers - the prevailing wisdom

- **Set number of reducers to 0.95 or 1.75 multiplied by (<*number of nodes*> * <*number of maximum containers per node*>).**
  - **Usually, maximum containers is 100-200.**
- **If you use 0.95**
  - all of the reducers can launch immediately
  - start transferring map outputs as the maps finish.
- **If you use 1.75**
  - faster nodes will finish their first round of reducers
  - then launch a second wave of reducers
  - Much better load balancing.
- **Increasing the number of reduces**
  - increases the framework overhead
  - increases load balancing and lowers the cost of failures.
- **The scaling factors above less than whole numbers to reserve a few reducers in the framework for speculative-tasks and failed tasks.**

- 1000 node cluster
- 128 max containers
- Usual load is 90%
  - 10 available nodes
  - 100 containers

=> 950 to 1750 reducers

how many reducers? Data-driven approach

- **Test with small data sets**
  - Determine average ratio of map-input and map-output.
  - Use rule of thumb: Give each reducer about a block of data

num-reducers = (size of map-output/size of map-input) * nInputSplits

- Example:
  - Data input is 8200 splits => 8200 Mappers
  - Avg input record = 1K
  - Avg map output = 200 bytes
  - Number of reducers = 8200*(200/1000) = 1640

Map only jobs

Zero reducers: setNumReducer(0)

What if we just don't need the reduce step?

- **File reformatting**
- **Data sampling**
- **ETL (Extract, transform, load)**

A Map-only job is a job with no reducers ...

```java
public class MapOnlySectorCounter extends Configured implements Tool {

    private static final Log LOG = LogFactory.getLog(MapOnlySectorCounter.class);

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

        int exitCode = ToolRunner.run(new MapOnlySectorCounter(), args);
        System.out.println("Job completed with status:  " + exitCode);
    }

    @Override
    public int run(String[] args) throws Exception {

        if (args.length != 2) {
            for (String arg : args)
                System.err.println(arg);
            System.err.println("Usage: maponly_sectorcount <input> <output>");
            System.exit(0);
        }

        String timeStampedOutput = args[1] + "_" + Calendar.getInstance().getTimeInMillis();

        LOG.info("input: " + args[0] + " output: " + timeStampedOutput);
        LOG.info("---- map only sector counter ------- ");

        Job job = Job.getInstance(getConf(), "sector count with counters");
        job.setJarByClass(MapOnlySectorCounter.class);
        job.setMapperClass(SectorMapperWithCounter.class);

        job.setNumReduceTasks(0);

        // everything is happening using counters - but, at the end, we write
        // something to the output file to help avoid confusion
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(timeStampedOutput));

        boolean result = job.waitForCompletion(true);

        return (result) ? 0 : 1;
    }
```

Counters

To create a counter, simply use it:

- define the group (for example, SectorCount)
- define the names
  - you can use a String variable, getCounter will create a new counter for each String the variable passes in.

```
context.getCounter("SectorCount", sectorStr).increment(1);
```

Output will go to the logs (and your console)

Counters are aggregated over all tasks

- you can use the same counter in a Mapper and a Reducer
- you cannot use a counter in a Partitioner

```
public static class SectorMapperWithCounter extends Mapper<Object, Text, Text, NullWritable> {

    private final static String recordRegex = ",(?=([^\"]*\"[^\"]*\")*[^\"]*$)";
    private final static int sectorIndex = 5;
    private final static int capIndex = 3;
    public final static String NO_INFO = "n/a";

    @Override
    public void cleanup(Context context) throws IOException, InterruptedException {

        context.write(new Text("Nothing to view here - all the info is in the counters."), NullWritable.get());

    }

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

        String[] tokens = value.toString().split(recordRegex, -1);
        String sectorStr = tokens[sectorIndex].replace("\"", "");
        String capStr = tokens[capIndex].replace("\"", "");

        if (tokens.length == 9 && !tokens[1].equals("Sector") && !sectorStr.equals(NO_INFO)
                && capStr.endsWith("B")) {

            context.getCounter("SectorCount", sectorStr).increment(1);
            context.getCounter("SectorCount", "Total companies processed successfully").increment(1);

        }
        context.getCounter("SectorCount", "Total companies attempted").increment(1);
    }

}
```

```
2016-10-27 15:48:55,210 INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1384)) -  map 100% reduce (
2016-10-27 15:48:55,212 INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1395)) - Job job_local1822(
2016-10-27 15:48:55,220 INFO  [main] mapreduce.Job (Job.java:monitorAndPrintJob(1402)) - Counters: 29
        File System Counters
                FILE: Number of bytes read=409181
                FILE: Number of bytes written=273866
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
        Map-Reduce Framework
                Map input records=3030
                Map output records=1
                Input split bytes=165
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=0
                Total committed heap usage (bytes)=154664960
        SectorCount
                Basic Industries=14
                Capital Goods=41
                Consumer Durables=17
                Consumer Non-Durables=28
                Consumer Services=142
                Energy=20
                Finance=110
                Health Care=143
                Miscellaneous=30
                Public Utilities=18
                Technology=197
                Total companies attempted=3030
                Total companies processed successfully=779
                Transportation=19
        File Input Format Counters
                Bytes Read=408959
        File Output Format Counters
                Bytes Written=68
Job completed with status:  0
```

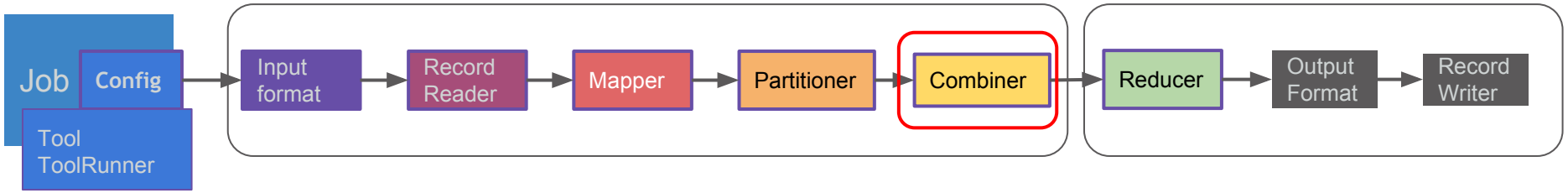
Review: Map-only Jobs with counters

- **Driver code:**
  - **Set number of reducers to zero:** `job.setNumReduceTasks(0)`
  - **Use job.setOutputKeyClass and setOutputValueClass**
    - don't use *setMapOutputKeyClass* or *setMapOutputValueClass*.
- **Mapper/Reducer code:**
  - **Create and use counters with the same command:**

```
context.getCounters("groupname", "countername").increment(n)
```

- **Output from context.write will write to HDFS.**
  - One file per Mapper **without sorting** by keys.
- **Output from Counters will write to the Console and to logs.**

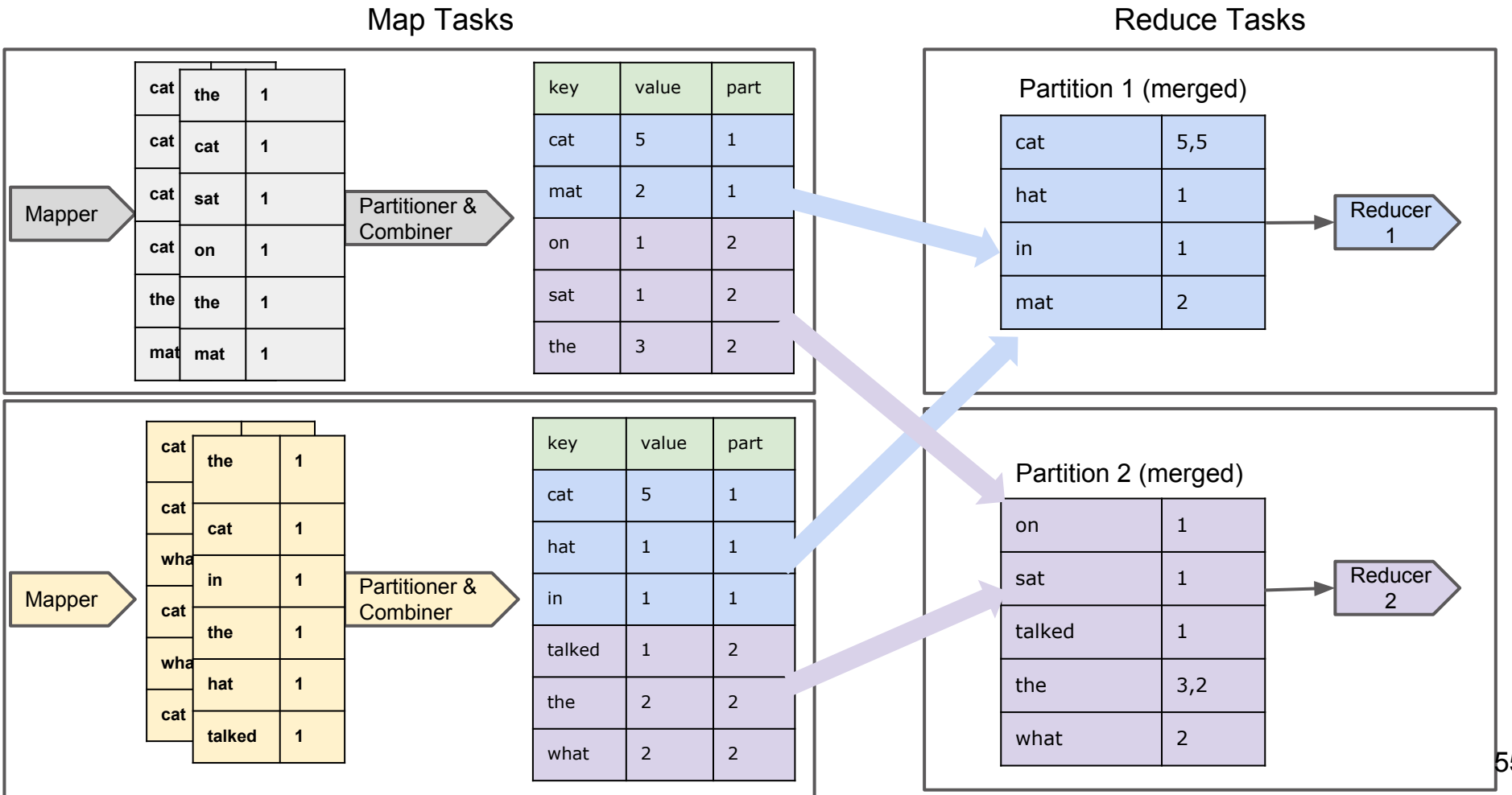
Combiners


Job
Config
Tool
ToolRunner
Input format
Record Reader
Mapper
Partitioner
Combiner
Reducer
Output Format
Record Writer

When are combiners used?

- **Mappers produce large amount of intermediate data**
  - Can create too much network traffic
  - Mappers map one-to-one to Combiners

- **A Combiner can reduce intermediate data**
  - Runs locally on Mapper output
  - Combiner output is sent to Reducers


Map Tasks
Reduce Tasks
Mapper
cat the 1
cat cat 1
cat sat 1
cat on 1
the the 1
mat mat 1
Partitioner & Combiner
key value part
cat 5 1
mat 2 1
on 1 2
sat 1 2
the 3 2
Partition 1 (merged)
cat 5,5
hat 1
in 1
mat 2
Reducer 1
Mapper
cat the 1
cat cat 1
wha in 1
cat the 1
wha hat 1
cat talked 1
Partitioner & Combiner
key value part
cat 5 1
hat 1 1
in 1 1
talked 1 2
the 2 2
what 2 2
Partition 2 (merged)
on 1
sat 1
talked 1
the 3,2
what 2
Reducer 2

Combiner: it's a Reducer

- **Combiner signature is same as Reducer:**

```
public static class MyCombiner extends Reducer
```

- **Overrides the reduce method, e.g.:**

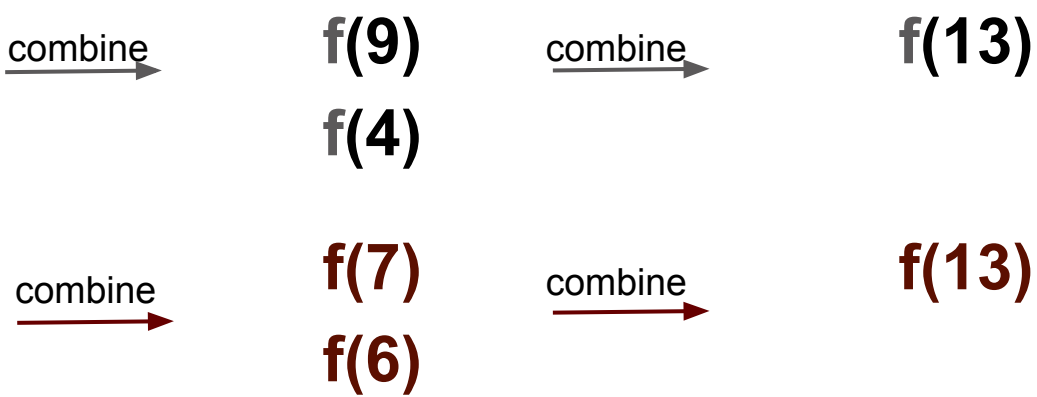
```
public void reduce(Text key, Iterable<Text> values, Context context)
```

- **Because data is spilled and then merged multiple times, the Combiner may run on data a,b,c,d in different ways:**
  - **combine(combine(a,b),combine(c,d))**
  - **combine(combine(a,b,c,d))**
  - **combine(combine(a), combine(b,c,d))**
- **Therefore, Combiners must be associative (unlike Reducers)**

Combiners are associative and commutative

- **Combiners may run more than once on output from the same Mapper.**

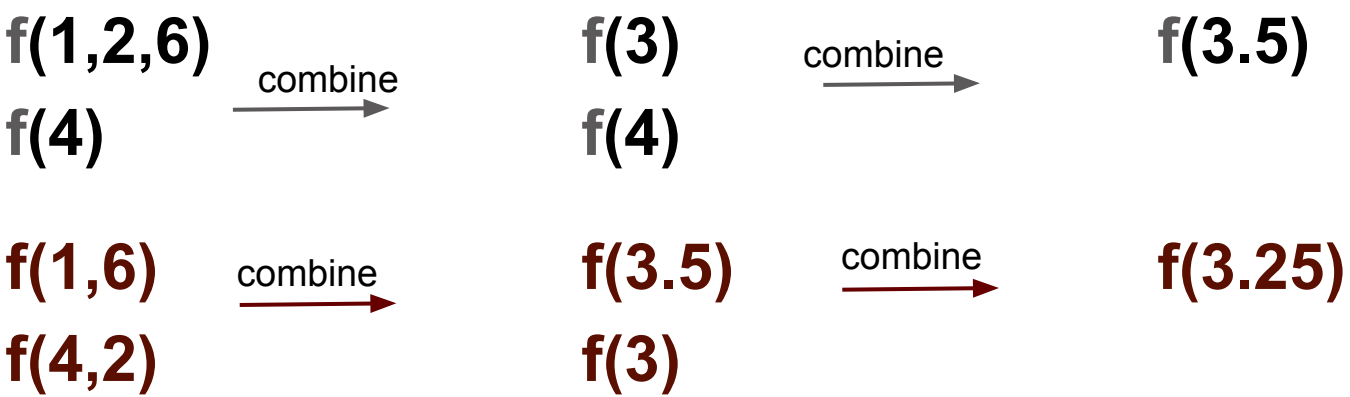
this works: SumReducer (addition)



Combiners are associative and commutative

this does not work: AvgReducer

avg(avg(a,b),c) != avg(a, avg(b,c))



Driver code for Combiner

- **job.setCombiner(MyCombiner.class);**

- **You can use the same class for both Combiner and Reducer:**
  - **job.setCombinerClass(SumReducer.class);**
  - **job.setReducerClass(SumReducer.class);**

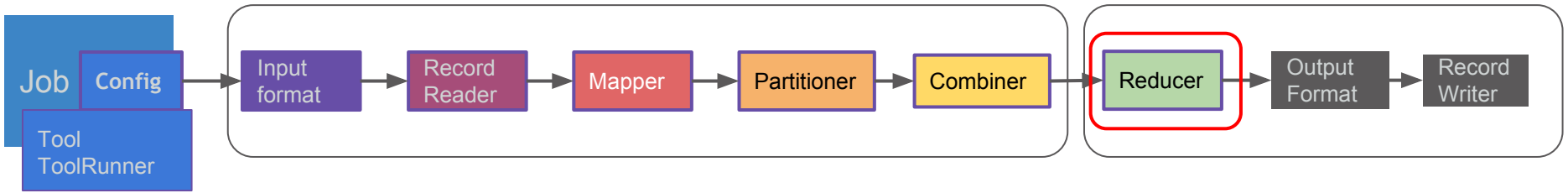
- **Input/output types of the Combiner must match input/output types of the Reducer.**

Practice 5

Practice 5: Writing a Partitioner

**** Note: The practice is on Canvas under practiceForVM files ****

Reducers



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

The Reducer (1)

- **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

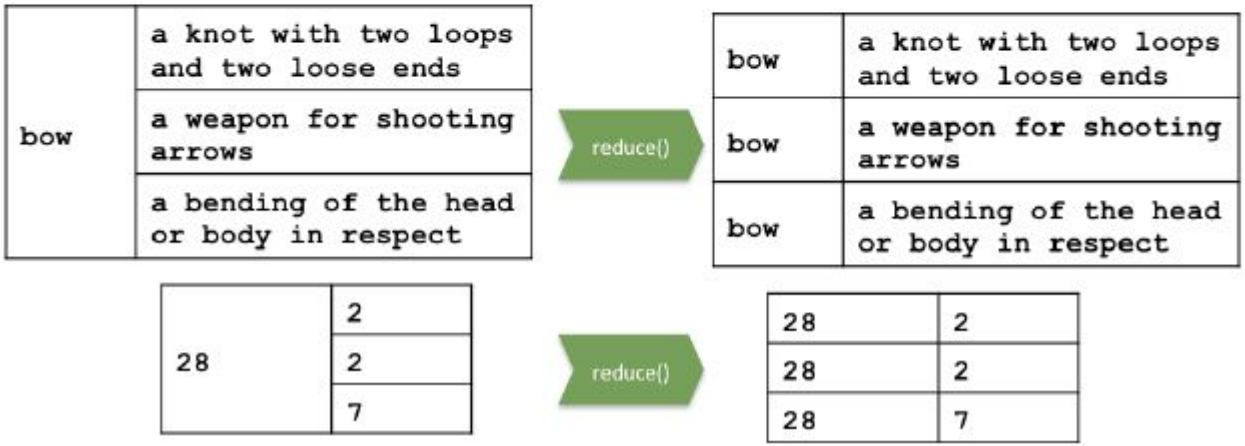
note: value.get() retrieves the integer.

average.set sets the double in DoubleWritable.

Example Reducer: Identity Reducer

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

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



Mapper/Reducer methods

setup

runs *before* any data is processed

- override is optional
- initializes data structures and parameters
- **define local variables using Configuration properties**

run

processes all the data in a split

```
for each (key,value) in the split
    mapper.map(key, value)
```

cleanup

runs *after* all the data is processed

- override is optional
- **use to write out summary info:**
  - counters, sums, errors, etc.
- close files

```
public static class StockCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private static int totalReducerCount = 0;
    private IntWritable result = new IntWritable();

    @Override
    public void cleanup(Context context) throws IOException, InterruptedException {
        Text describe = new Text(
                "------------- output from reducer's cleanup -----------------\nTotal count for reducer:  ");
        IntWritable totalCount = new IntWritable(totalReducerCount);
        context.write(describe, totalCount);
    }

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

        int count = 0;
        for (IntWritable value : values) {
            count += value.get();
        }
        totalReducerCount += count;
        result.set(count);
        context.write(key, result);
    }
}
```

Caveat: keep track of types

Output types in driver must match Mapper and Reducer

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

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

Driver code

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

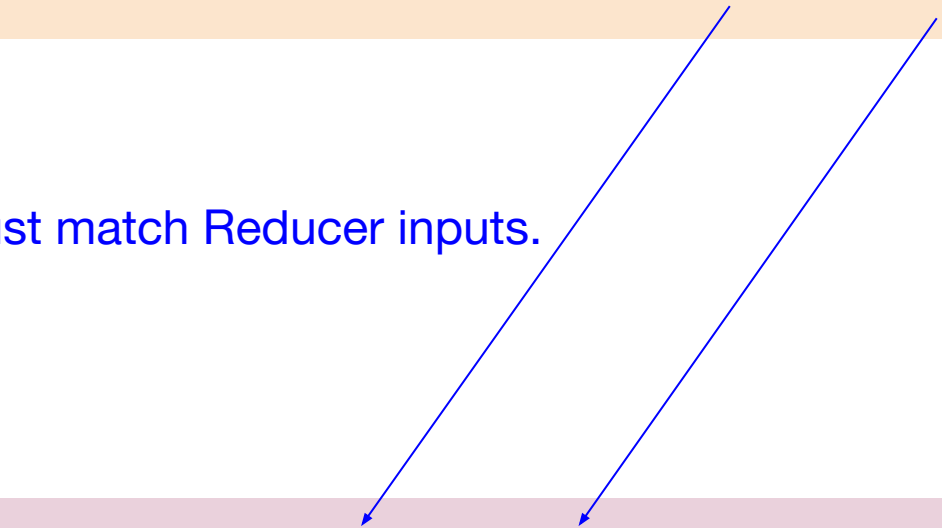
```
public class MaxTempReducer extends Reducer<Text, IntWritable, Text, FloatWritable>
```

Mapper and Reducer outputs must match output setting in the driver.

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>
```

Mapper outputs must also match Partitioner inputs

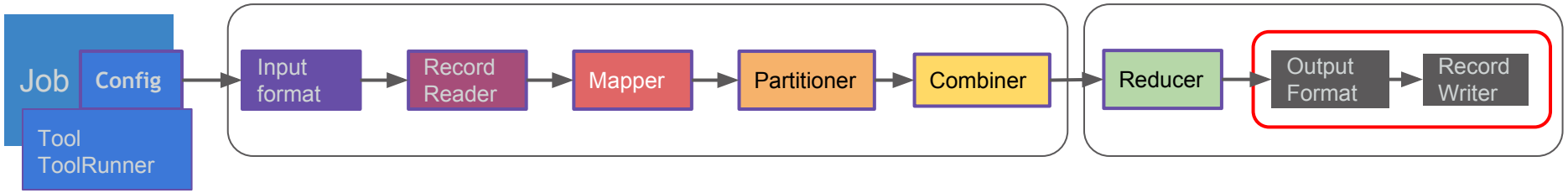
```
public class MaxTempMapper extends Mapper<LongWritable, Text, Text, IntWritable>
```

Map outputs must match Partitioner and Reducer inputs.

```
public class MaxTempPartitioner<K,V> extends Partitioner<Text, IntWritable>
     implements Configurable
```

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

Output locations and OutputFormats



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 format default

- **Defaults for the OutputFormat.**

```
TextOutputFormat.class;
```

  - (very general, often used)

- **To override the default, specify:**
  - job.setOutputFormatClass(<OutputFormat>)

Commonly used OutputFormats

(Default) TextOutputFormat: Writes plain text files

MultipleOutputFormat: The reducer writes data to different files depending on the keys

SequenceFileOutputFormat: Writes output in compressed format

- We will have a section of sequence files and compression next week

DBOutputFormat:

- Configure a job so it can create a DB connection using JDBC.
- DBOutputFormat generates a set of INSERT statements in each reducer.
- The reducer's close() method then executes them in a bulk transaction against the database.
- https://archanaschangale.wordpress.com/tag/dbinputformat/

most common question - merging reducer output

answer one:

use the getmerge command:

```
hadoop fs -getmerge <...>
```

- warning: doesn't work for sequence files or Avro
- creates file on local filesystem, not HDFS

answer two:

add this to the end of your driver

```
FileUtil.copyMerge(fs, srcPath, fs, dstPath, false, config, null);
```

there is no *official* output format that merges output files - care to guess why?

Hadoop Streaming

The MaxTemp MapReducer written in Python

Hadoop Streaming

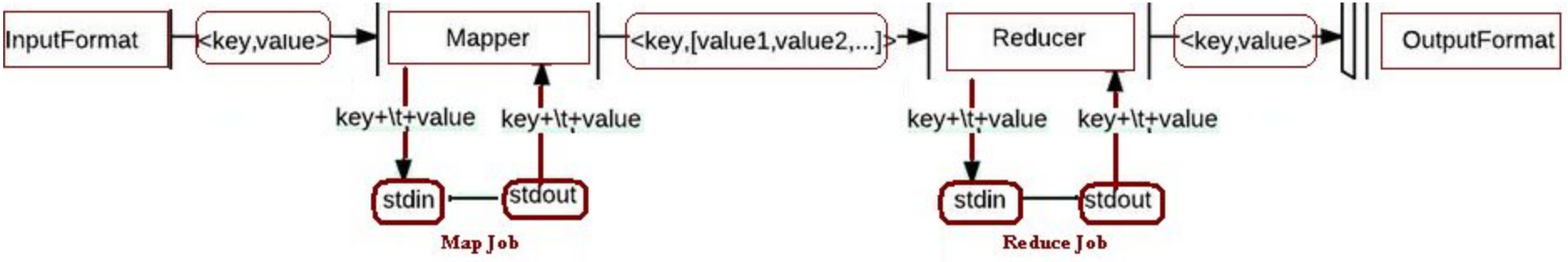
- **Features**
- **Example using Python**
  - **python mapper**
  - **python reducer**
- **How to run**

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**

Recommended Practice

SpecialPractice: HadoopStreaming

*** Note: This practice is already on your VM on the desktop ***

Next time: Hadoop IO

Using sqoop, flume, the filecache and more...

Additional slides you might find interesting

Requested: PDF files example

- Input Method 1 - controls where splits occur:
  - Create a **SequenceFile** containing the PDF files.
  - Create a class derived from **Writable** to contain the PDF
  - Job.setInputFormatClass(**SequenceFileInputFormat**.class)
- Input Method 2 - prevents splitting entirely:
  - Use Job.setInputFormatClass(**WholeFileInputFormat**.class)
    - One file is assigned to each MapTask
    - Used only for large PDF files
- In your Map or Reduce code: use any java PDF library such as PDFBox to manipulate the PDFs.

How many mappers?

- **Usually, the number of maps ~= total number of blocks used by the input files**
- **The right level of parallelism for maps: 10-100 mappers per-node**
  - **Can go up to 300 mappers for very cpu-light map tasks.**

Example:

1TB of input data

blocksize of 128MB ⇒ 8,200 mappers

To override: Configuration.set(MRJobConfig.NUM_MAPS, int)

- used by InputFormat
- may only provide a hint or not be used at all
- depends on the splitting method (splitting by records, files, DB buffers, or marked real-time data)

What if the InputFormat won't take the hint? What can you change?

Maximum number of containers per node

- Depends on how much memory is assigned to the NodeManager
- Depends on MapReduce specific requirements.

Will cover when we turn to configuration of the cluster.

Jumpstart:

- **A newbie asks: what is the maximum number of containers per node**
- **Hortonworks suggestions for configuration**

Testing code with MR Unit

Unit Testing

A ‘unit’ is a small bit of code with function.

Verify correctness of function.

Function needs clear definition or a contract.

Unit should test that “contract”.

Should take < 1 sec. to complete.

- Allows incremental development.
- Can be joined into suites.
- Catches side-effects of other code changes.

unit testing: important for Hadoop

Normally, MR are huge jobs:

- hours to complete.
- many machines.

We normally code and test on one development machine:

- single-node/pseudo-distributed mode
- VMs running clusters

Even so, tests running the whole code stream can take a long time (and stall anything else on your machine).

unit testing: JUnit and MRUnit Frameworks

- JUnit tests of MR code are really tedious to set up...
- MRUnit is built on JUnit and mocks up the MapReduce framework.
- Like JUnit, can be run from Eclipse.

unit testing: introduction to JUnit

Can generate a stubbed out JUnit test with a wizard in Eclipse - and select the methods to test.

Then:

- create the test and decorate outputs with assertEquals, assertTrue and assertFalse.
- run by right-clicking on the test and selecting “Run as JUnit Test”

Clean, clear results.

unit testing: using mrunit

- mrunit runs on top of JUnit
- Provides a mock InputSplit and other classes
- Can test just the Mapper, just the Reducer or the full MapReduce flow.

Testing with MRunit - imports

```
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.apache.hadoop.mrunit.mapreduce.ReduceDriver;
import org.apache.hadoop.mrunit.mapreduce.MapReduceDriver;

import org.junit.Before;
import org.junit.Test;
```

Import the relevant junit classes and the mrunit MapDriver classes.

Testing a Mapper with MRUnit

```
public class WordMapperTest {

    MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;

    @Before
    public void setup() {
        WordMapper mapper = new WordMapper();
        mapDriver = new MapDriver(mapper);
    }

    @Test
    public void TestMapper() throws IOException {

        mapDriver.withInput(new LongWritable(1), new Text("duck duck goose"));

        mapDriver.withOutput(new Text("duck"), new IntWritable(1));
        mapDriver.withOutput(new Text("duck"), new IntWritable(1));
        mapDriver.withOutput(new Text("goose"), new IntWritable(1));

        mapDriver.runTest();

    }
}
```

Match Mapper's template

Define the test driver.

define test inputs and expected results.

run the test, checking results == outputs.

Testing a Reducer with MRUnit

```
public class SumReducerTest {

    ReduceDriver<Text, IntWritable, Text, IntWritable> reduceDriver;

    @Before
    public void setup() {
        SumReducer reducer = new SumReducer();
        reduceDriver = new ReduceDriver<Text, IntWritable, Text, IntWritable>(reducer);
    }

    @Test
    public void TestReducer() throws IOException {

        List<IntWritable> values = new ArrayList<IntWritable>();
        values.add(new IntWritable(1));
        values.add(new IntWritable(1));

        reduceDriver.withInput(new Text("duck"), values);
        reduceDriver.withOutput(new Text("duck"), new IntWritable(2));

        reduceDriver.runTest();

    }
}
```

Match Reducer's template

Define driver.

define test inputs and expected results.

run the test, checking results == outputs.

Testing both the Mapper and Reducer

```
public class WordCountTest {

    MapReduceDriver<LongWritable, Text, Text, IntWritable, Text, IntWritable> mapReduceDriver;

    @Before
    public void setup() {

        // define the test driver for both the Mapper and Reducer
        WordMapper mapper = new WordMapper();
        SumReducer reducer = new SumReducer();
        mapReduceDriver = new MapReduceDriver<LongWritable, Text, Text, IntWritable, Text, IntWritable>(mapper,
                reducer);
    }

    @Test
    /**
     * Actual test for the Mapper and Reducer
     *
     * @throws IOException
     */
    public void TestMapReduce() throws IOException {

        // very simple input based on a single Key, Value to be passed to the Mapper
        mapReduceDriver.withInput(new LongWritable(1), new Text("duck duck goose"));

        // expected outputs from the reducer
        mapReduceDriver.withOutput(new Text("duck"), new IntWritable(2));
        mapReduceDriver.withOutput(new Text("goose"), new IntWritable(1));

        // runTest will compare test outputs with expected outputs
        mapReduceDriver.runTest();

    }
```

MRUnit drivers and their methods

- **3 drivers: MapDriver, ReduceDriver, MapReduceDriver.**
- **Methods to specify test input and output:**
  - **testing one (key,value) pair at a time**
    - `withInput(inputKey, inputValue)`
    - `withOutput(resultKey, resultValue)`
  - **testing a list of (key,value) pairs at a time**
    - `withAll:` list of input key,value pairs
    - `withAllOutput:` output list of expected key, value pairs
- **Test other components:** withCombiner, withCounters, withComparators...

MRUnit tests - running the tests

- **methods for running tests:**
  - **driver.runTest() - Runs the test and verifies output**
  - **driver.run() - Runs the test and returns the results**

- **Running multiple tests**
  - **Call driver.resetOutput() between calls.**
  - **If not, the test will fail.**

Summary

- **unit testing is critical to swift development**
- **mrunit is a framework for testing MapReduce programs**
- **mrunit is beautifully configured and insanely easy to install**
- **you can write tests for Mappers and Reducers individually, or together.**
- **Best practice: always write unit tests!**

Information on MRUnit

For full information on MRUnit see http://mrunit.apache.org/

Other good links are:

- **MRUnit Tutorial: https://cwiki.apache.org/confluence/display/MRUNIT/MRUnit+Tutorial**
- **MRUnit Javadoc: http://mrunit.apache.org/documentation/javadocs/0.9.0-incubating/overview-summary.html**
- **Getting Started with MRUnit: http://mrunit.apache.org/documentation/javadocs/0.9.0-incubating/org/apache/hadoop/mrunit/package-summary.html**

For future reference:

Installing MRUnit

Download the latest version of MRUnit jar from Apache website is version 1.1.0:

http://mrunit.apache.org/general/downloads.html

If you are using the Hadoop version 2.2.0, download mrunit-1.1.0-hadoop2.jar

http://www.apache.org/dyn/closer.cgi/mrunit/mrunit-1.1.0/apache-mrunit-1.1.0-hadoop2-bin.tar.gz

I use the sonic.net mirror because sonic rules.

From Eclipse:

1. Select File -> Import -> Maven -> Existing Maven Project...

2. After the project has been imported, right click the project in the project explorer and select Maven -> Update Project Configuration.