Where we left off… MapReduce Development

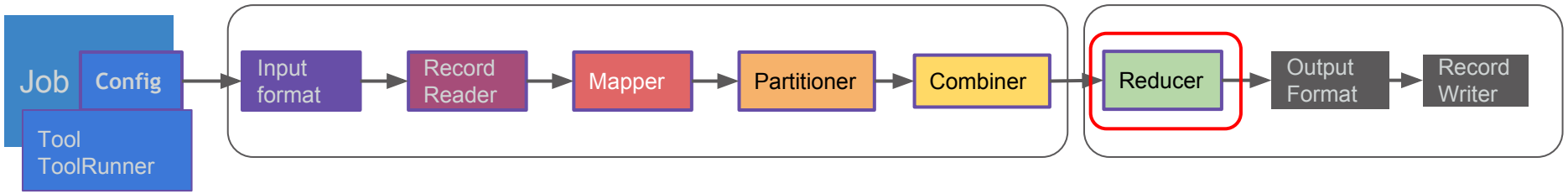
A stroll down the MapReduce API

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

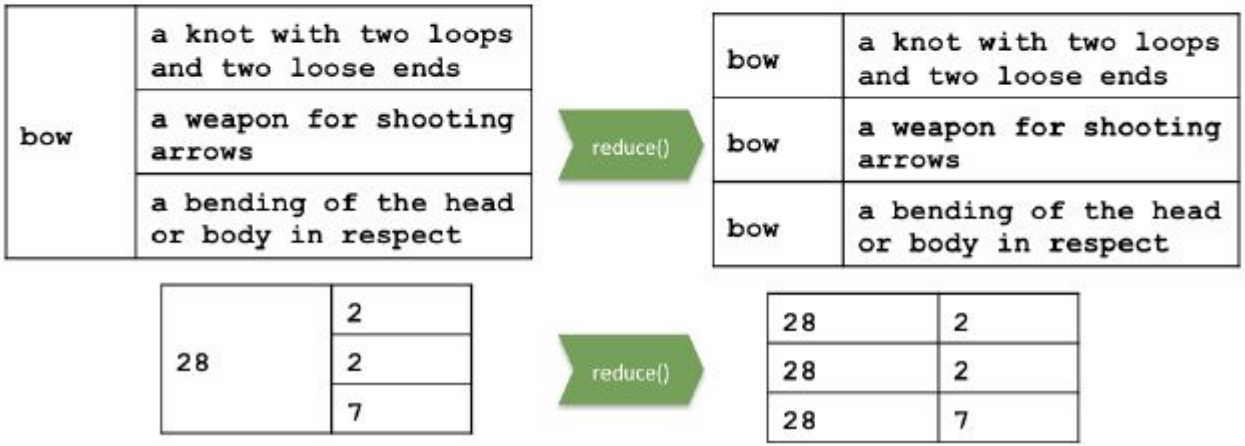
note: value.get() retrieves the integer.

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>
```

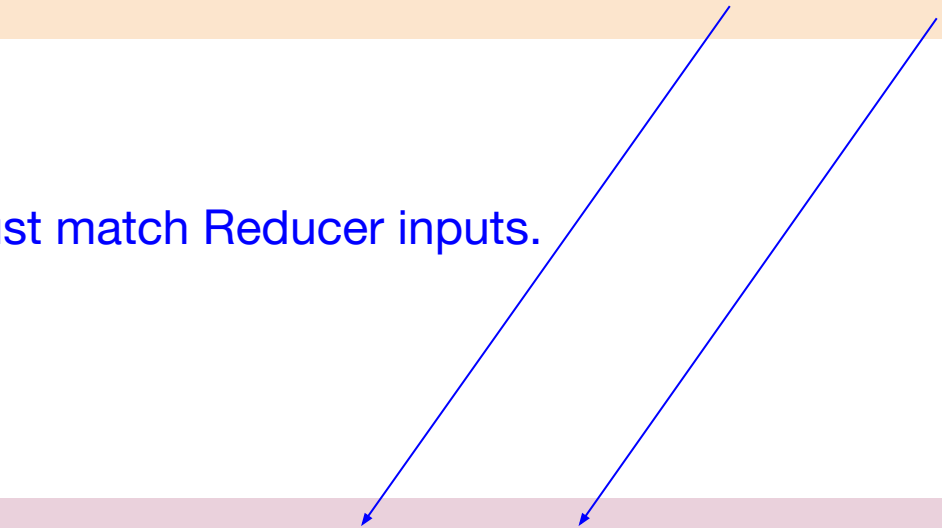
```
public class MaxTempReducer extends Reducer<Text, IntWritable, Text, FloatWritable>
```

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

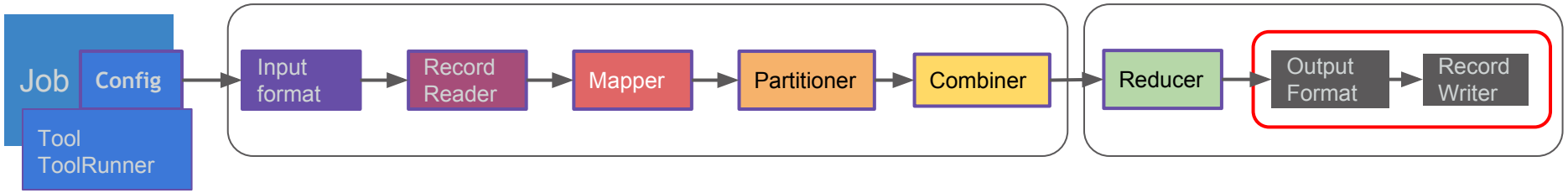
```
public class MaxTempMapper extends Mapper<LongWritable, Text, Text, IntWritable>
```

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:

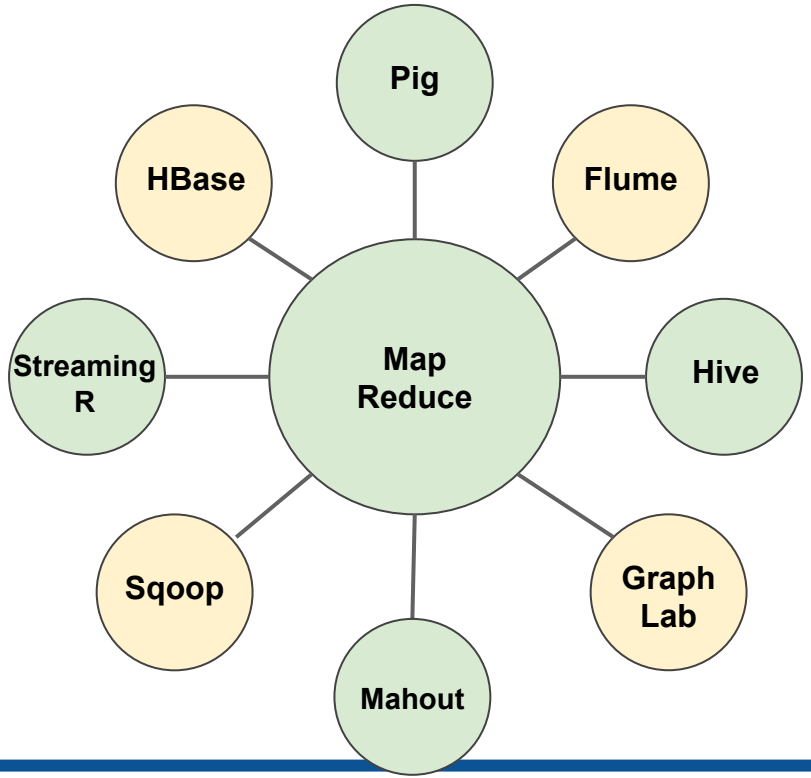
add this to the end of your driver

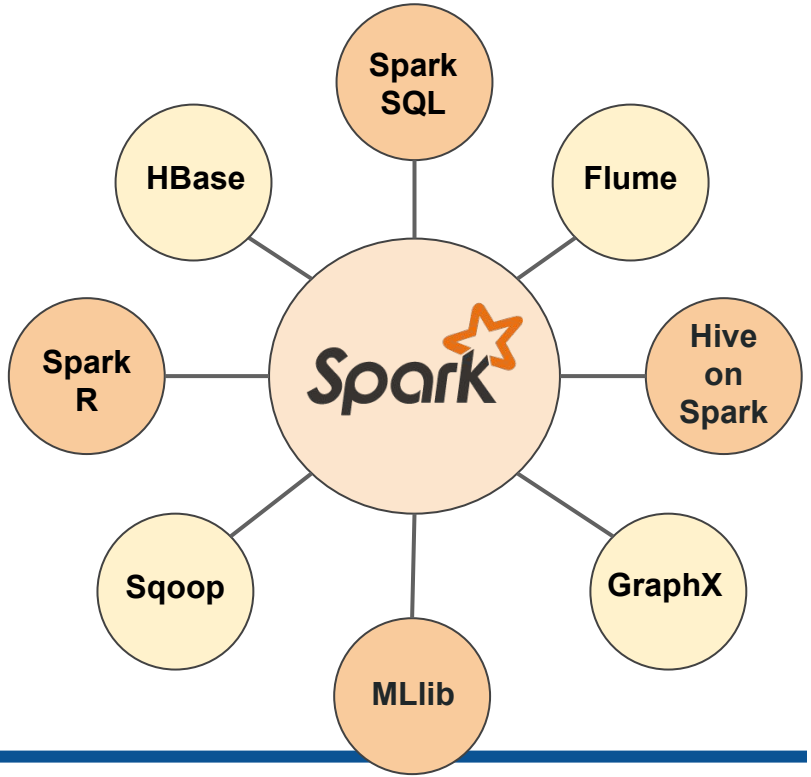
```
FileUtil.copyMerge(fs, srcPath, fs, dstPath, false, config, null);
```

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

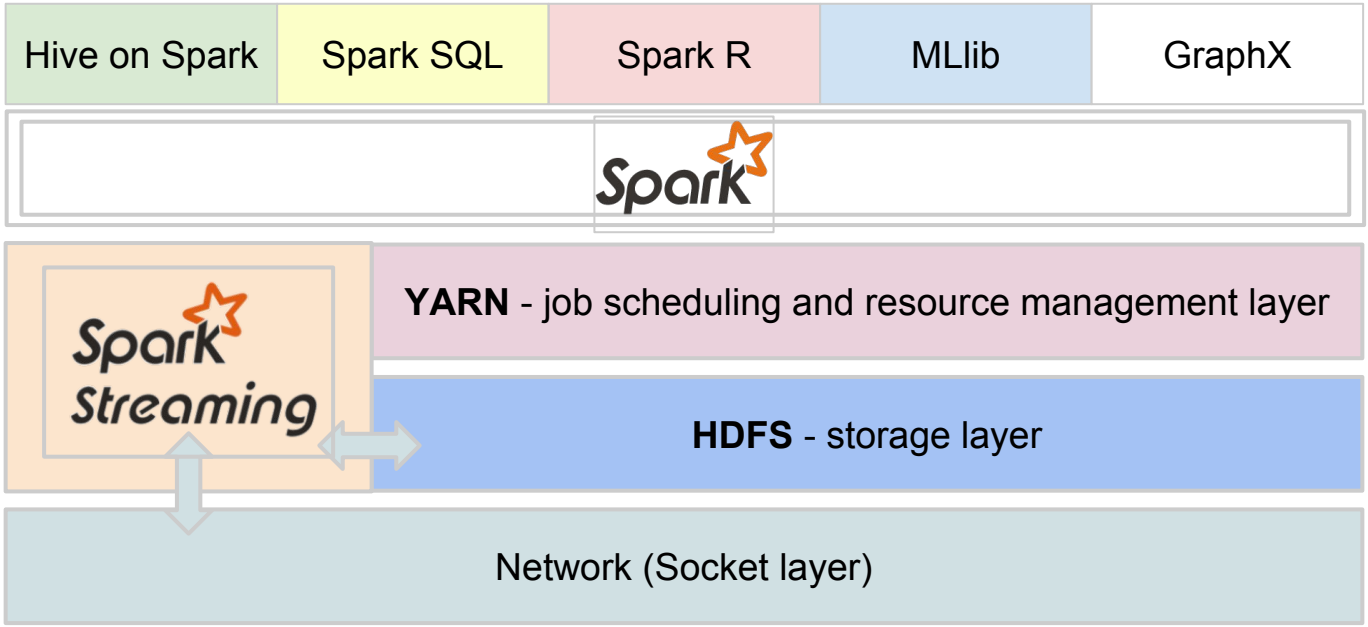
Spark

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





Spark Analytics



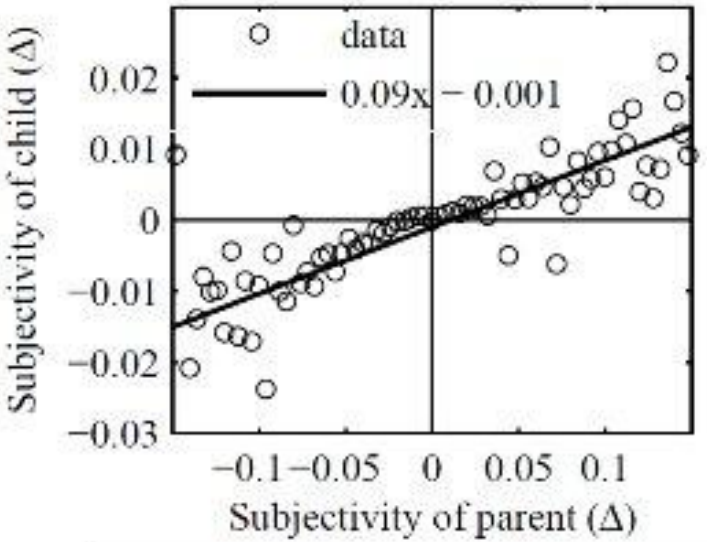
databricks

- 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

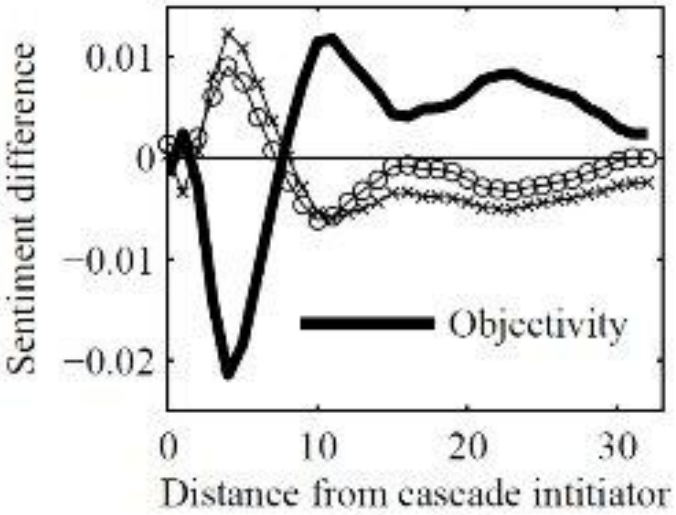

There’s hype...
hadoop + Spark = A Winning Combination

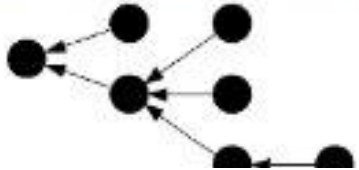
The course of sentiment

- Cascades "heats" up early and then cool off



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

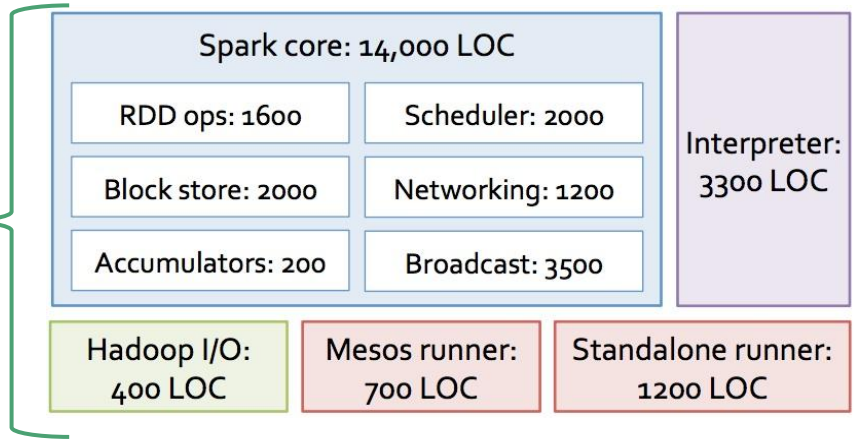




There's change: Spark growth

Core:

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



Speed - by objective observers

Graysort competition

3x faster than Hadoop MR2

Clash of the Titans (see site)

3-7x faster than Hadoop MR2

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
  - Often, no need for Cascading or Oozie

MR2 tasks vs Spark transformations

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

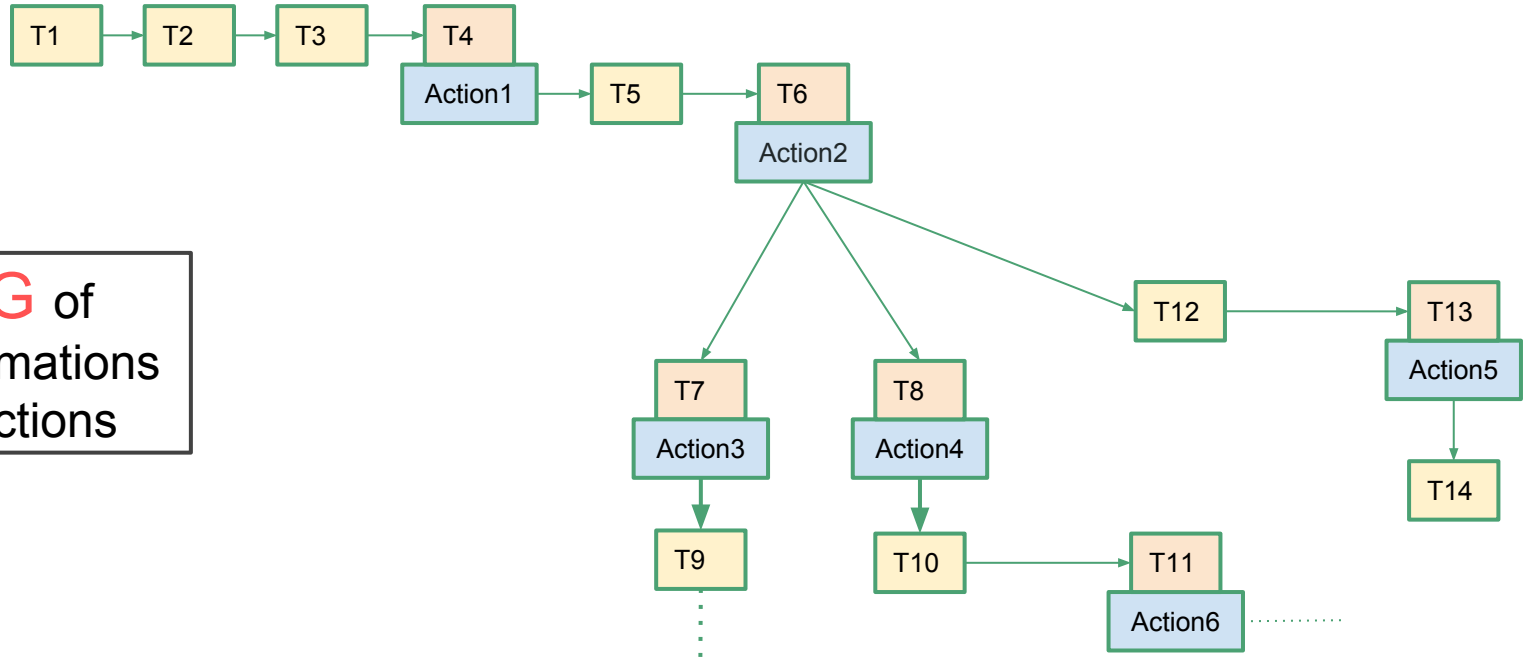
Spark executes a job with many transformations.

- Narrow transformations are like MapTasks - they do not involve shuffle-sort
- Wide transformations are like ReduceTasks - they trigger a shuffle during processing

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

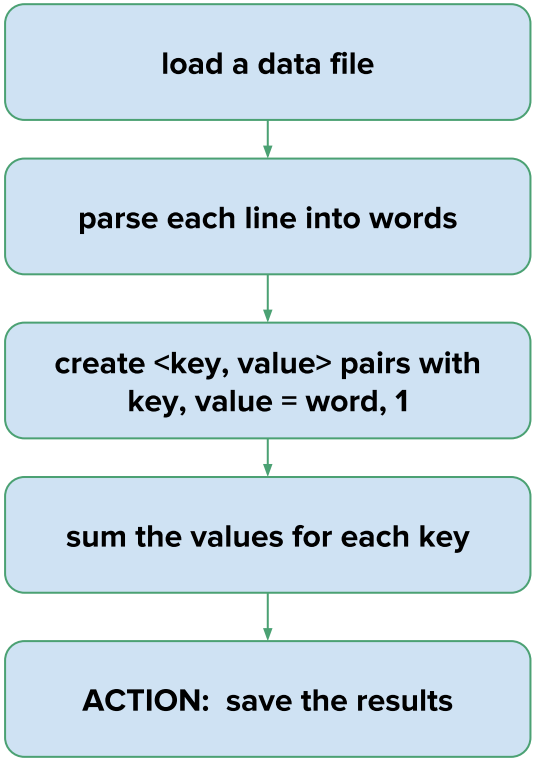
- Solves the hard problems
  - Easily moves data from one MR job to another
  - Shuffles in-memory
  - Caches variables and recycles JVMs for tasks
- Solves user problems
  - Multiple languages
    - Python and Scala shell
    - Applications in Java, Python and Scala
  - Well-integrated with Spark R,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

WordCount setup

```
public class WordCount {

    private final static String recordRegex = "\\W+";
    private final static Pattern REGEX = Pattern.compile(recordRegex);

    public static void main(String[] args) {
        /*
         * Validate that one variable is passed from the command line.
         */
        if (args.length != 2) {
            System.out.printf("Usage: Provide <input dir> <output dir> \n");
            System.exit(-1);
        }

        /*
         * setup job configuration and context
         */
        SparkConf conf = new SparkConf();
        conf.setMaster("local");
        conf.setAppName("Word count");
        JavaSparkContext sc = new JavaSparkContext(conf);

        /*
         * setup input and output
         */
        String output = args[1] + "_" + Calendar.getInstance().getTimeInMillis();
```

WordCount processing

```
JavaRDD<String> lines = sc.textFile(args[0]);

JavaRDD<String> words = lines.flatMap(
        line -> Arrays.asList(REGEX.split(line)));

JavaPairRDD<String, Integer> wordPairs = words.mapToPair
        (w -> new Tuple2<String, Integer>(w, 1));

JavaPairRDD<String, Integer> wordCounts = wordPairs.reduceByKey(
        (x, y) -> x + y);

wordCounts.saveAsTextFile(output);
```



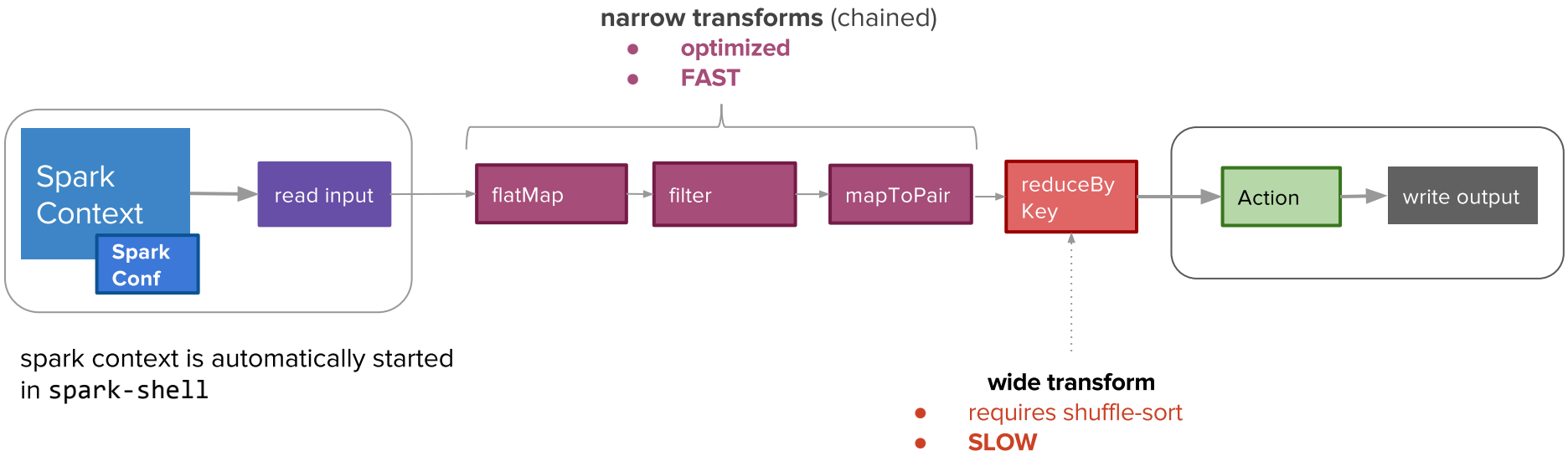
Let's filter

```
JavaRDD<String> lines = sc.textFile(args[0]);

JavaRDD<String> words = lines.flatMap(line -> Arrays.asList(REGEX.split(line)));
JavaRDD<String> filteredWords = words.filter(word -> word.toLowerCase().contains("love"));
JavaPairRDD<String, Integer> wordPairs = words.mapToPair(w -> new Tuple2<String, Integer>(w, 1));

JavaPairRDD<String, Integer> wordCounts = wordPairs.reduceByKey((x, y) -> x + y);

wordCounts.saveAsTextFile(output);
```

map functions

combine *and* reduce function

Spark workflow for “filtered” count



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

Filebacked RDD
<values only>

Single RDD
<values only>

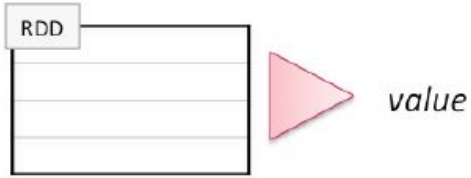
Pair RDD
<key, value>

Pair RDD
<key, value>

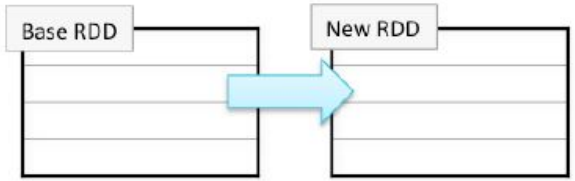
```
JavaRDD<String> lines = sc.textFile(args[0]);

JavaRDD<String> words = lines.flatMap(
        line -> Arrays.asList(REGEX.split(line)));

JavaPairRDD<String, Integer> wordPairs = words.mapToPair
        (w -> new Tuple2<String, Integer>(w, 1));

JavaPairRDD<String, Integer> wordCounts = wordPairs.reduceByKey(
        (x, y) -> x + y);

wordCounts.saveAsTextFile(output);
```

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 data ops
- 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

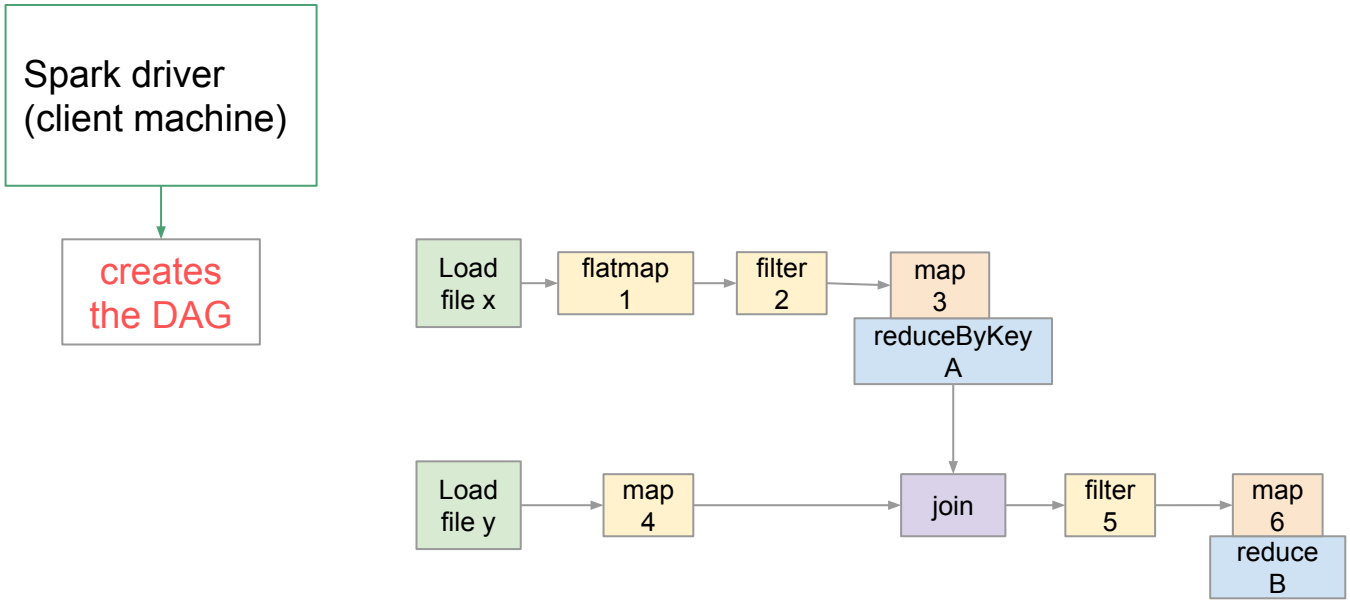
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-dagscheduler.html

Spark has a planning and execution engine

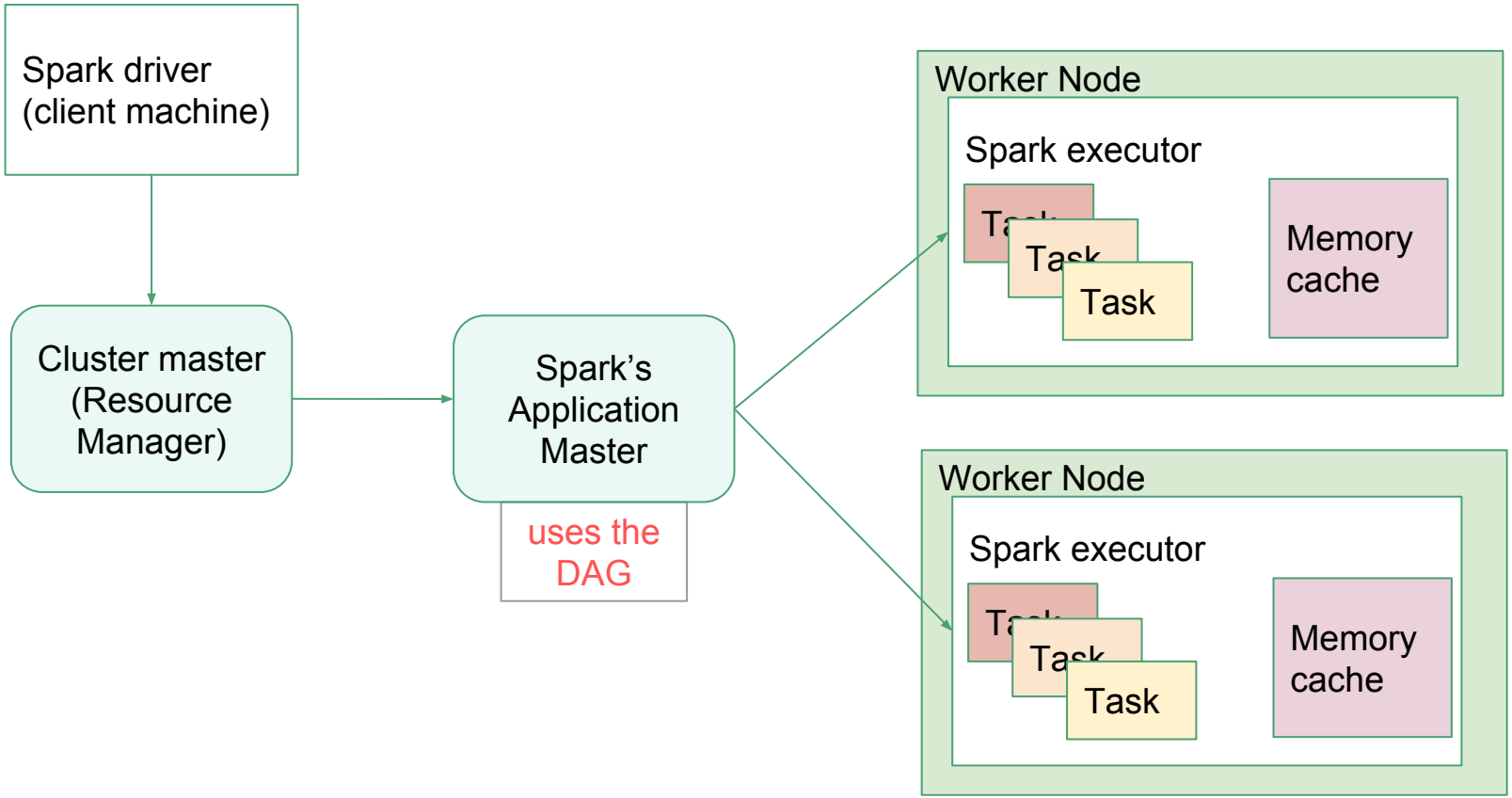
- Creates a plan for the job
- The job is executed in stages
- Narrow transformations are optimized (chained)
  - run on the same executor
  - use the same data
  - process the data line-by-line through the whole chain

Execution is managed by the DAGScheduler

Spark execution(1)



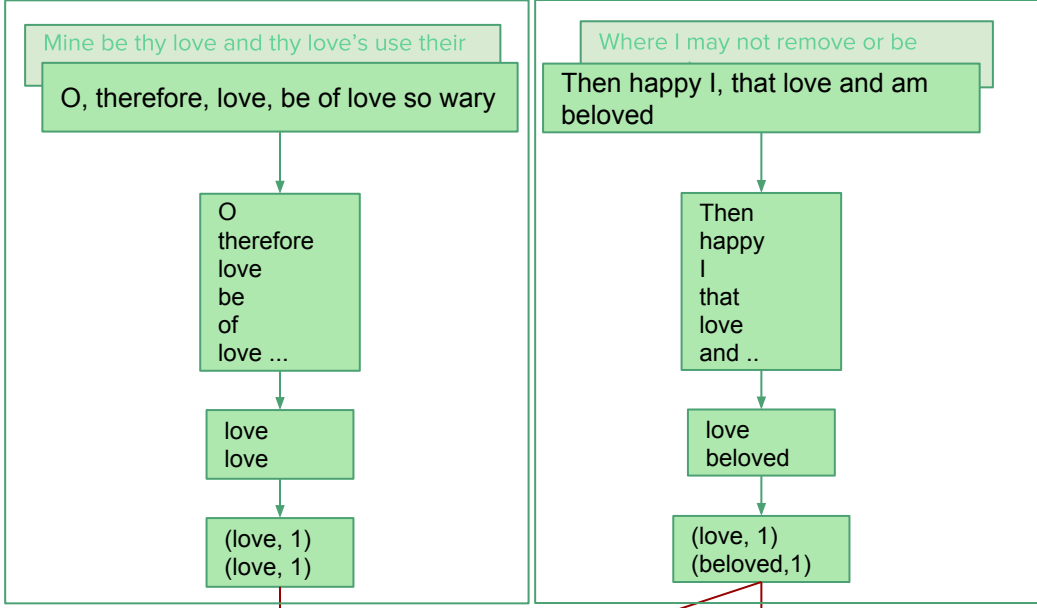
Spark execution(2)



So, Spark jobs run in stages?

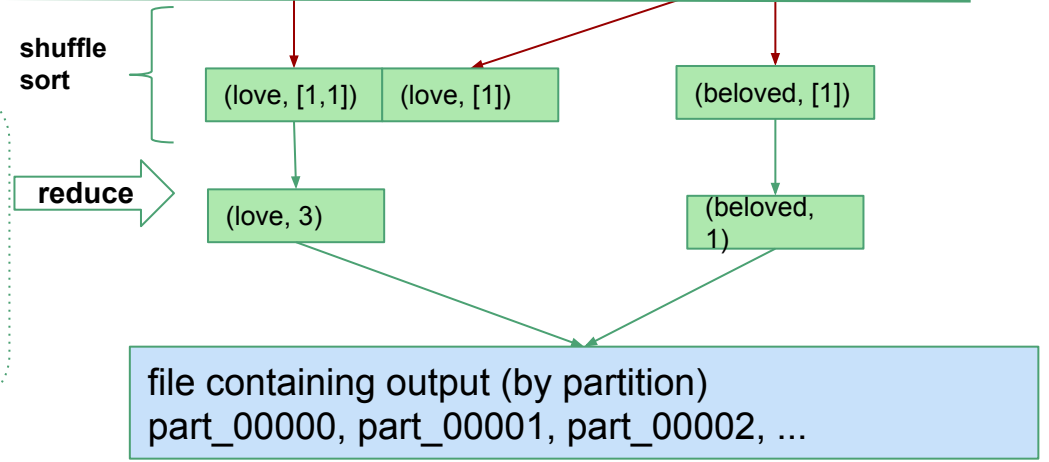
Stage 1

- **textFile** - read line from file
- **flatMap -** parse line into words
- **filter** words containing “love”
- **mapToPair** words to (word, 1) pairs



Stage 2

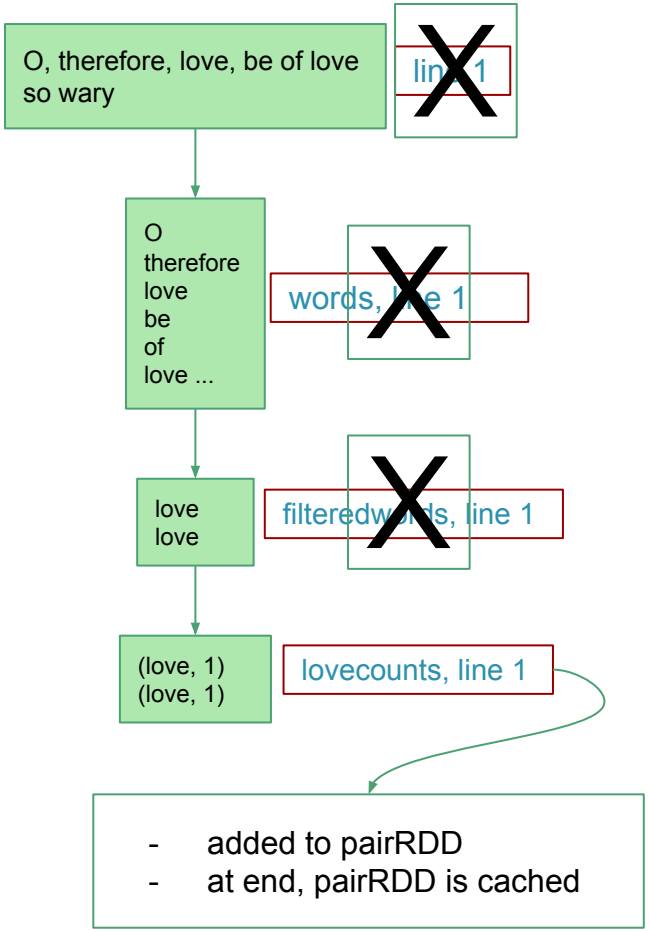
- **reduceByKey** to get counts
- **saveAsFile** to start processing



Stage 1: a chain of narrow transforms:

textFile -> flatmap -> filter -> mapToPair

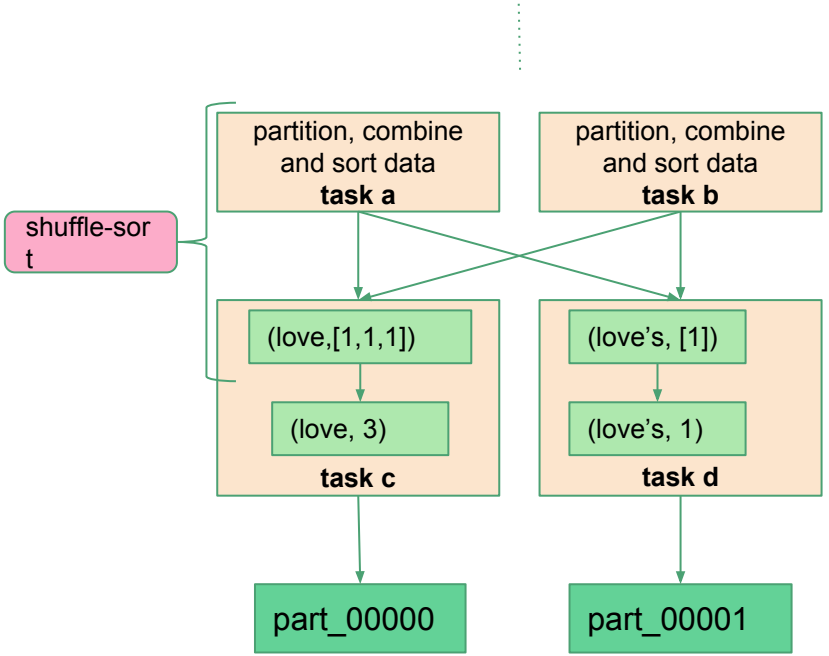
- Assign an executor to each input split
  - executes the chain of transforms
  - each line in a split is processed thru the chain
  - optimizes to chain to look/act like a single task
- Intermediate results (RDDs) are NOT stored
  - Records process consecutively
  - Each record is fed through all transforms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



Stage 2:

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

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

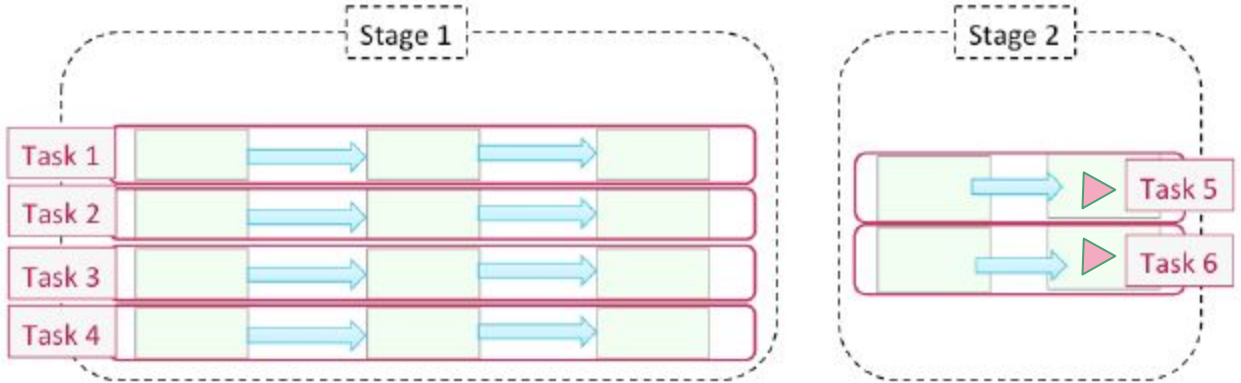


Summary

```
lines = sc.textFile(“data/shakespeare/poetry/*”)
words = lines.flatMap(line -> Arrays.asList(REGEX.split(line)));
filteredWords = words.filter(word -> word.toLowerCase().contains("love"));
wordPairs = words.mapToPair(w -> new Tuple2<String, Integer>(w, 1));
-----------------------------------------------------------------------
wordCounts = wordPairs.reduceByKey((x, y) -> x + y);
counts.saveAsFile(output);
```

Stage 1

Stage 2



That saveAsFile action - huh?

- saveAsFile() 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.saveAsFile()” is the **action** that started the job
    - if you use a REPL, you only see processing after an action

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

Code examples

Logging

Passing Parameters

Some comments on optimization

Logging

Logging is configured in log4j.properties

- file must be first log4j.properties file in classpath
- right now, for eclipse, it is in src/main/resources/conf

Log level is specified using:

- log4j.rootCategory=INFO, logfile, console
- change INFO to DEBUG for more information
- familiarize yourself with Levels (ERROR, WARNING, INFO, DEBUG, TRACE)

Output file is determined in the properties file using:

- log4j.appender.logfile.File=spark.log

```java
public class StockCountWithLogging {

    static Logger logger = Logger.getLogger(StockCountWithLogging.class);

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

        // validate input
        if (args.length != 1) {
            System.out.printf("Usage: Provide <input dir>\n");
            System.exit(-1);
        }

        // setup job configuration and context
        SparkConf conf = new SparkConf();
        conf.setMaster("local");
        conf.setAppName("Stock count");
        JavaSparkContext sc = new JavaSparkContext(conf);

        // create a log for messages generated by this class
        ConfigurationUtil.setupClassLogging(logger, StockCountWithLogging.class);

        // read the file
        JavaRDD<String> lines = sc.textFile(args[0]);

        // count the lines in file
        long n = lines.count();

        // log the count
        logger.info("The number of lines in the input file is:  " + n);

        sc.close();
    }
```

Parameter passing

- parse parameter input from args[] in main
- create a static final holder for the parameter
- the holder must be read-only because it is used across JVMs
- use freely, you can be confident that the information is present

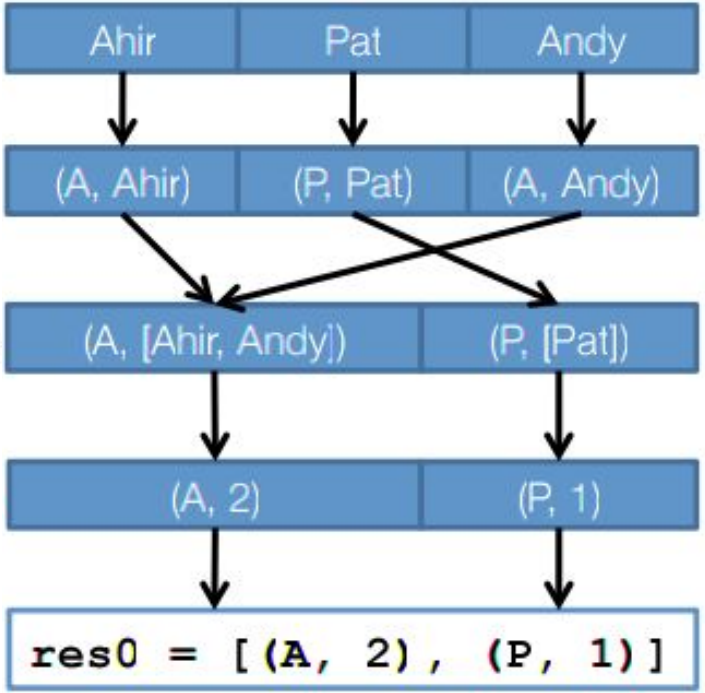
```
// these are used across all tasks, they MUST be read-only
final boolean sort = Boolean.valueOf(args[2].replace("sort=", ""));
final boolean save = Boolean.valueOf(args[3].replace("save=", ""));


JavaRDD<String> lines = sc.textFile(args[0]);

JavaPairRDD<String, Integer> sectorCounts = lines.mapToPair(new PairFunction<String, String, Integer>() {
    @Override
    public Tuple2<String, Integer> call(String line) {

        String[] tokens = REGEX.split(line, -1);

        if (tokens.length == 9) {
            String sector = tokens[sectorIndex].replace("\"", "");
            if (!sector.equals("n/a"))
                return new Tuple2<String, Integer>(sector, 1);
        }

        return new Tuple2<String, Integer>("Invalid record", 1);
    }
});


JavaPairRDD<String, Integer> totalCounts = sectorCounts.reduceByKey(new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer a, Integer b) throws Exception {
        return a + b;
    }
});

/*-
 * sort the output for legibility
 */
if (sort)
    totalCounts = totalCounts.sortByKey();

/*-
 * and action!
 */
if (save)
    totalCounts.saveAsTextFile(outputPath);
else {
    List<Tuple2<String, Integer>> output = totalCounts.collect();
    for (Tuple2<?, ?> tuple : output) {
        // show in red so it is easy to see on the console
        System.err.println(tuple._1() + ": " + tuple._2());
    }
}
```

Yet another coding example

A few optimizations to consider

Find number of distinct names per “first letter”

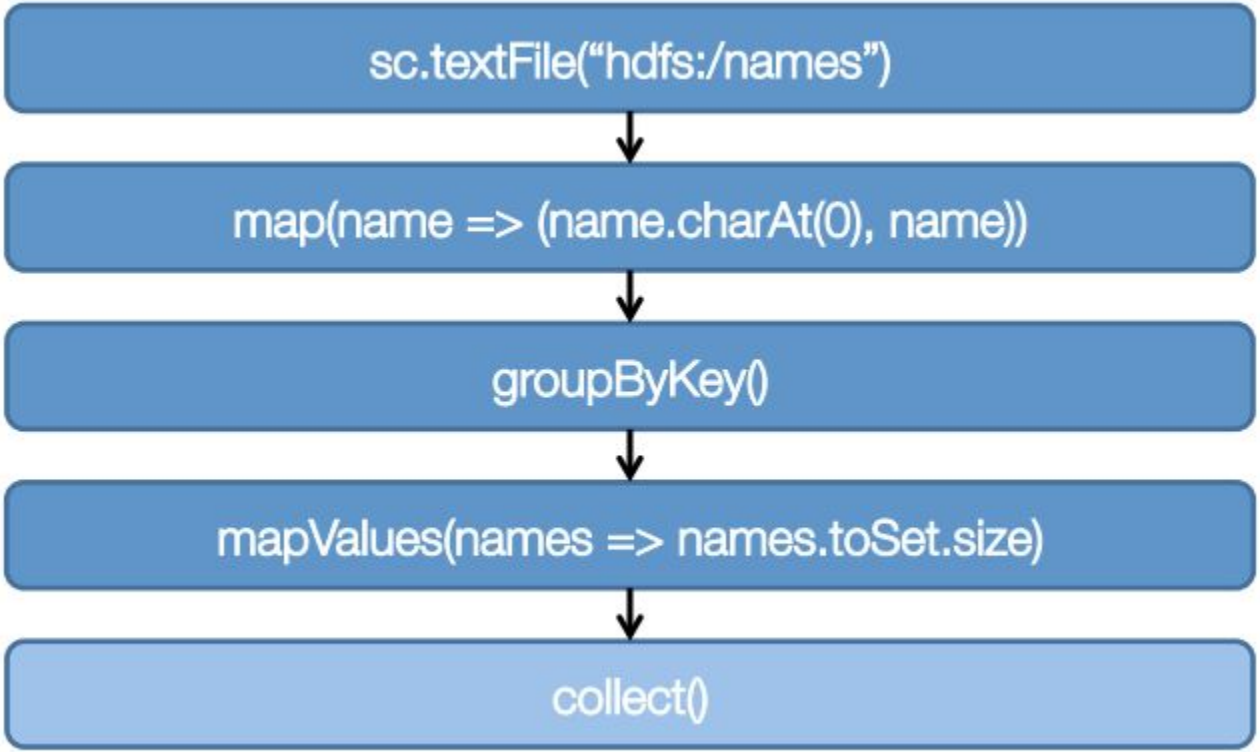
```
JavaRDD<String> lines = sc.textFile(args[0]);


JavaPairRDD<String, String> pairs = lines.mapToPair
        (w -> new Tuple2<String, String>(w.substring(0, 1), w));



JavaPairRDD<String, Iterable<String>> groupedPair = pairs.groupByKey();


JavaPairRDD<String, Integer> nameOccurs = pairs.mapValues
        (new Function<String, Integer>() {
    @Override
    public Integer call(String names) throws Exception {
        return new LinkedHashSet<String>(Arrays.asList(names)).size();
    }

});


nameOccurs.collect();
```



Spark Execution Model

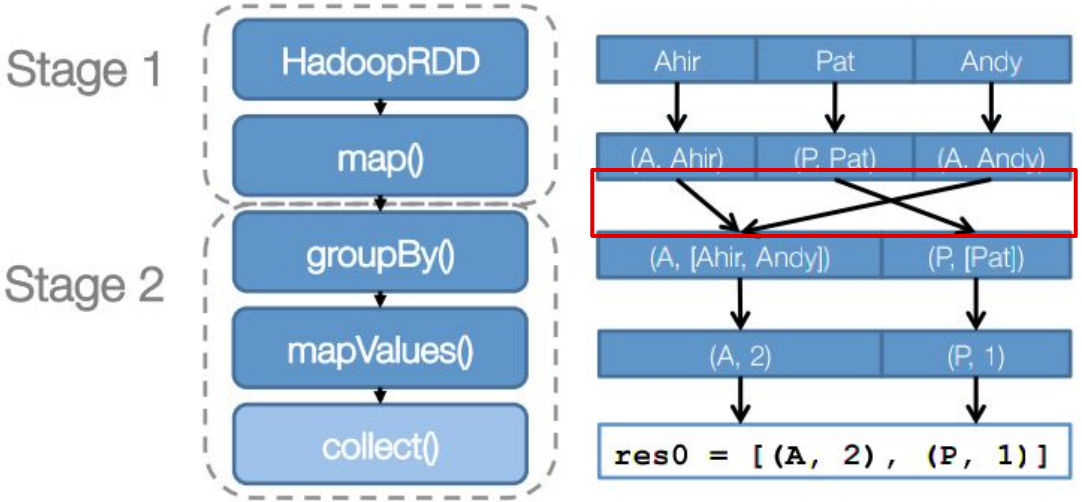
1. Create DAG of RDDs to represent computation
2. Create logical execution plan for DAG
3. Schedule and execute individual tasks

Step 1: DAG for job



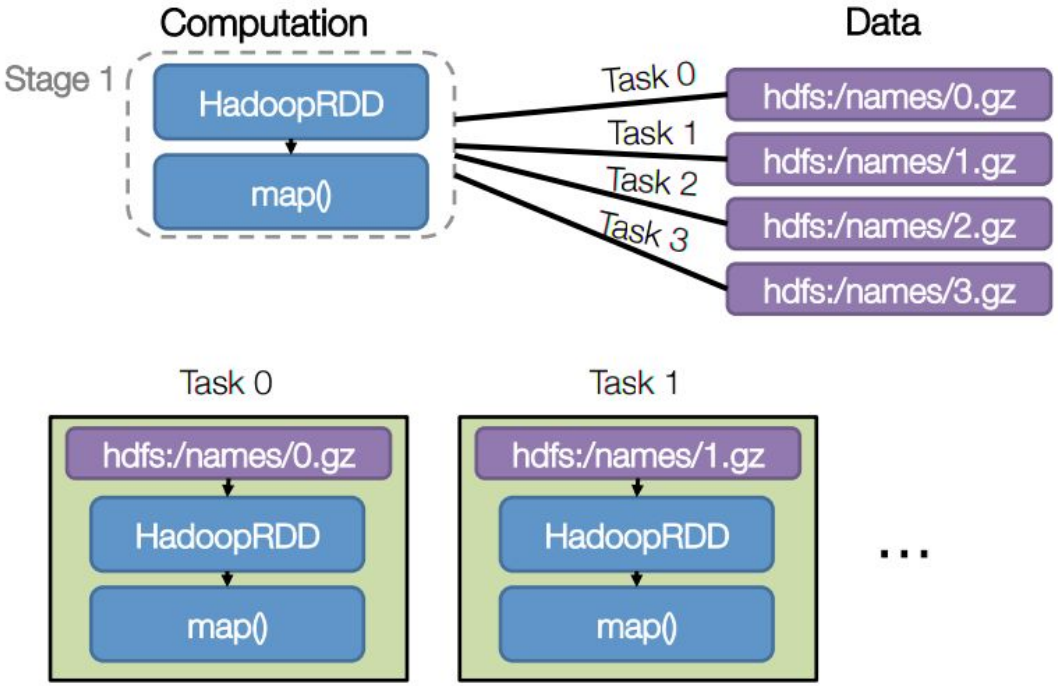
Step 2: Execution plan

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



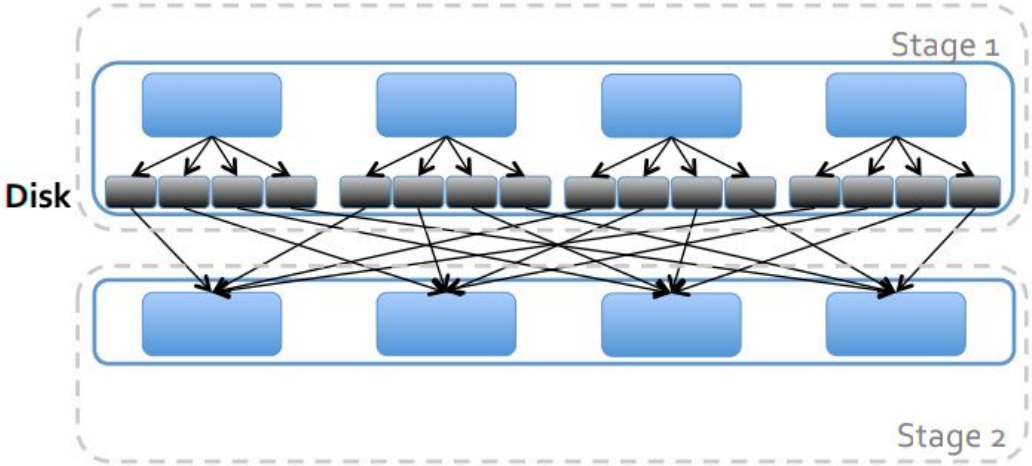
Step 3: Scheduling

- 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 an operation to remove duplicates near end of job

- we usually 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

Question: Is there a better way to “GroupBy” and “MapValues”?

- Yes, there is “ReduceByKey”

Revised code with optimizations

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

Original:

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

At home practice - Practice 3

The aggregateByKey is the most difficult to learn,

- that’s why it is in the practice

Two parts to practice 3 - running in MR2 and in Spark.

- spend time on Spark

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

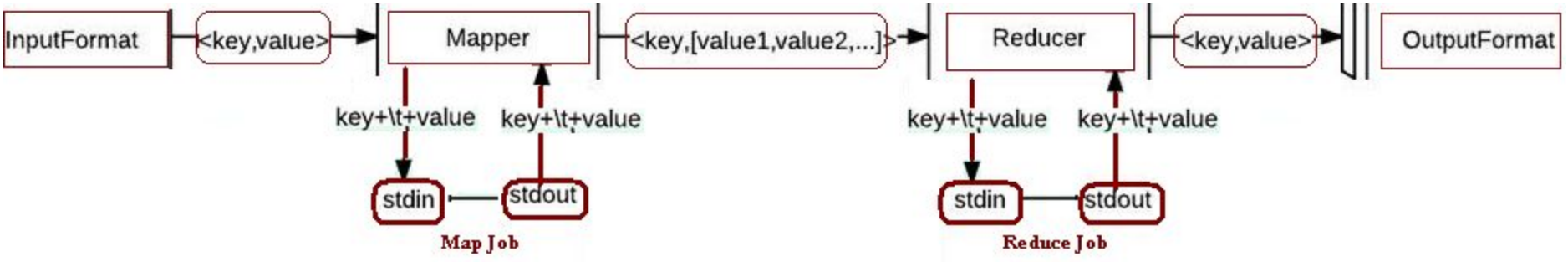
- **Features**
- **Example using Python**
  - **python mapper**
  - **python reducer**
- **How to run**

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**

Hadoop Streaming

The MaxTemp MapReducer written in Python

Hadoop Streaming

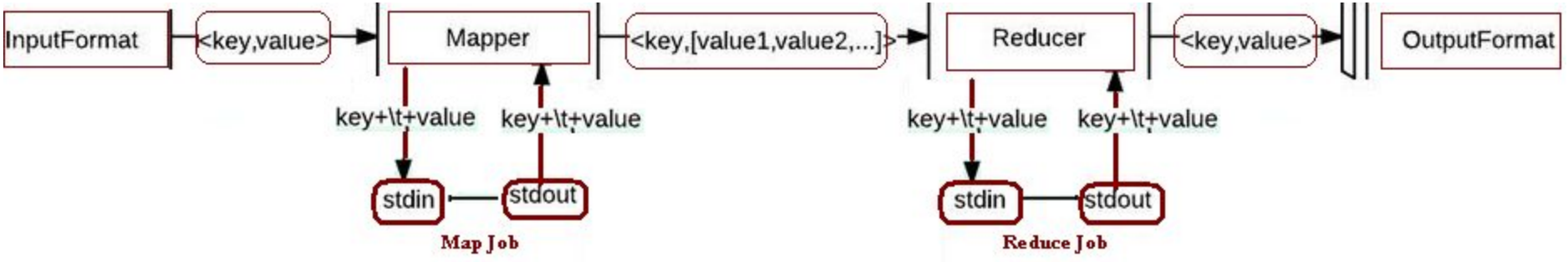
- **Features**
- **Example using Python**
  - **python mapper**
  - **python reducer**
- **How to run**

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 ***