THE UNIVERSITY of EDINBURGH
informatics

Extreme Computing

Hadoop MapReduce in more detail

How will I actually learn Hadoop?

- This class session
- Hadoop: The Definitive Guide
- RTFM
- There is a lot of material out there
  - There is also a lot of useless material
  - You need to filter it
  - Just because some random guy wrote a blog post about something does not make it right
  - We will have lab sessions
    - Attend them
    - Ask questions
    - Ask more questions

Basic Hadoop API

- Mapper
  - `void setup(Mapper.Context context)`
    Called once at the beginning of the task
  - `void map(K key, V value, Mapper.Context context)`
    Called once for each key/value pair in the input split
  - `void cleanup(Mapper.Context context)`
    Called once at the end of the task
- Reducer/Combiner
  - `void setup(Reducer.Context context)`
    Called once at the start of the task
  - `void reduce(K key, Iterable<V> values, Reducer.Context context)`
    Called once for each key
  - `void cleanup(Reducer.Context context)`
    Called once at the end of the task

Basic Hadoop API

- Partitioner
  - `int getPartition(K key, V value, int numPartitions)`
    Get the partition number given total number of partitions
- Job
  - Represents a packaged Hadoop job for submission to cluster
  - Need to specify input and output paths
  - Need to specify input and output formats
  - Need to specify mapper, reducer, combiner, partitioner classes
  - Need to specify intermediate/final key/value classes
  - Need to specify number of reducers (but not mappers, why?)
  - Don’t depend of defaults!

Data types in Hadoop: keys and values

Writable

Defines a de/serialization protocol. Every data type in Hadoop is a Writable.

WritableComparable

Defines a sort order. All keys must be of this type (but not values).

IntWritable
LongWritable
Text
…

Concrete classes for different data types.

SequenceFiles

Binary encoded of a sequence of key/value pairs

“Hello World”: Word count

```
Map(String docid, String text):
     for each word w in text:
          Emit(w, 1);

Reduce(String term, Iterator<Int> values):
     int sum = 0;
     for each v in values:
          sum += v;
          Emit(term, value);
```

“Hello World”: Word Count

```
private static class MyMapper extends
    Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable ONE = new IntWritable(1);
  private final static Text WORD = new Text();

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    String line = ((Text) value).toString();
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      WORD.set(itr.nextToken());
      context.write(WORD, ONE);
    }
  }
}
```

"Hello World": Word Count

```
private static class MyReducer extends
    Reducer<Text, IntWritable, Text, IntWritable> {

  private final static IntWritable SUM = new IntWritable();

  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context) throws IOException, InterruptedException {
    Iterator<IntWritable> iter = values.iterator();
    int sum = 0;
    while (iter.hasNext()) {
      sum += iter.next().get();
    }
    SUM.set(sum);
    context.write(key, SUM);
  }
}
```

Getting data to mappers and reducers

- Configuration parameters
  - Directly in the Job object for parameters
- Side data
  - DistributedCache
  - Mappers/reducers read from HDFS in setup method
- Avoid object creation at all costs
  - Reuse Writable objects, change the payload
- Execution framework reuses value object in reducer
- Passing parameters via class statics

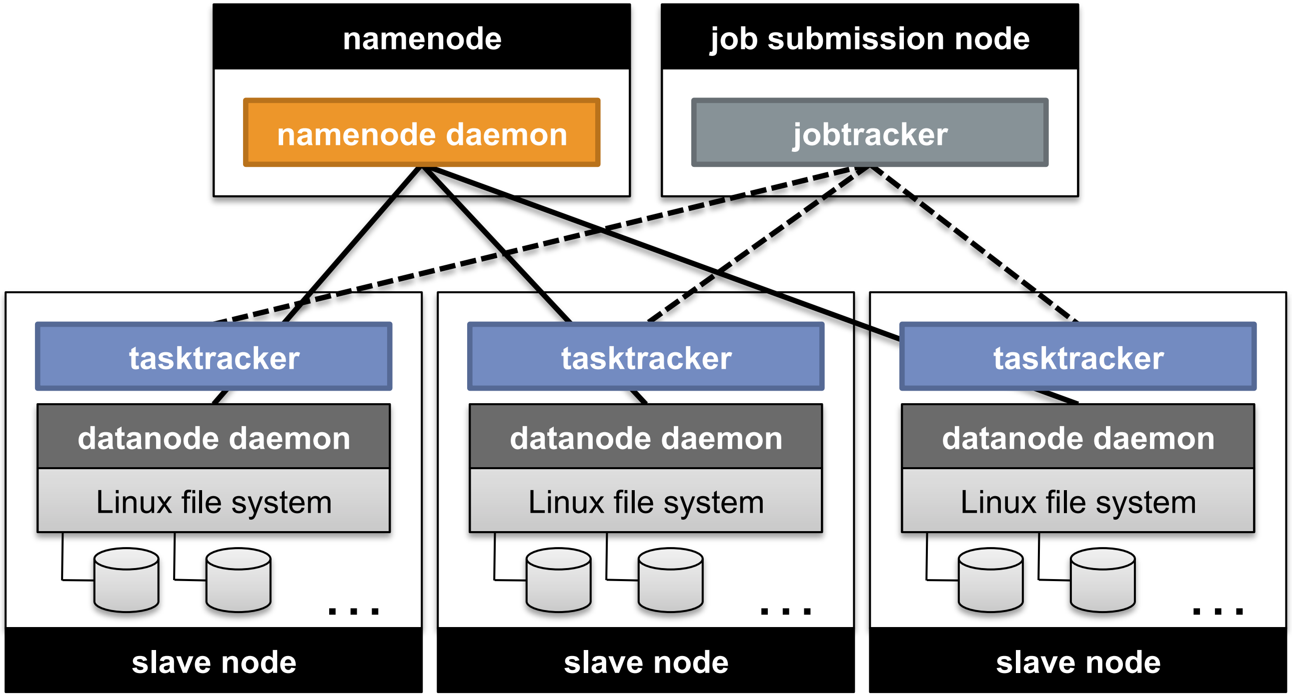
Complex Data Types in Hadoop

- How do you implement complex data types?
- The easiest way:
  - Encoded it as Text, e.g., (a, b) = “a:b”
  - Use regular expressions to parse and extract data
  - Works, but pretty hack-ish
- The hard way:
  - Define a custom implementation of Writable(Comparable)
  - Must implement: readFields, write, (compareTo)
  - Computationally efficient, but slow for rapid prototyping
  - Implement WritableComparator hook for performance
- Somewhere in the middle:
  - Some frameworks offers JSON support and lots of useful Hadoop types

Basic cluster components

- One of each:
  - Namenode (NN): master node for HDFS
  - Jobtracker (JT): master node for job submission
- Set of each per slave machine:
  - Tasktracker (TT): contains multiple task slots
  - Datanode (DN): serves HDFS data blocks

Recap

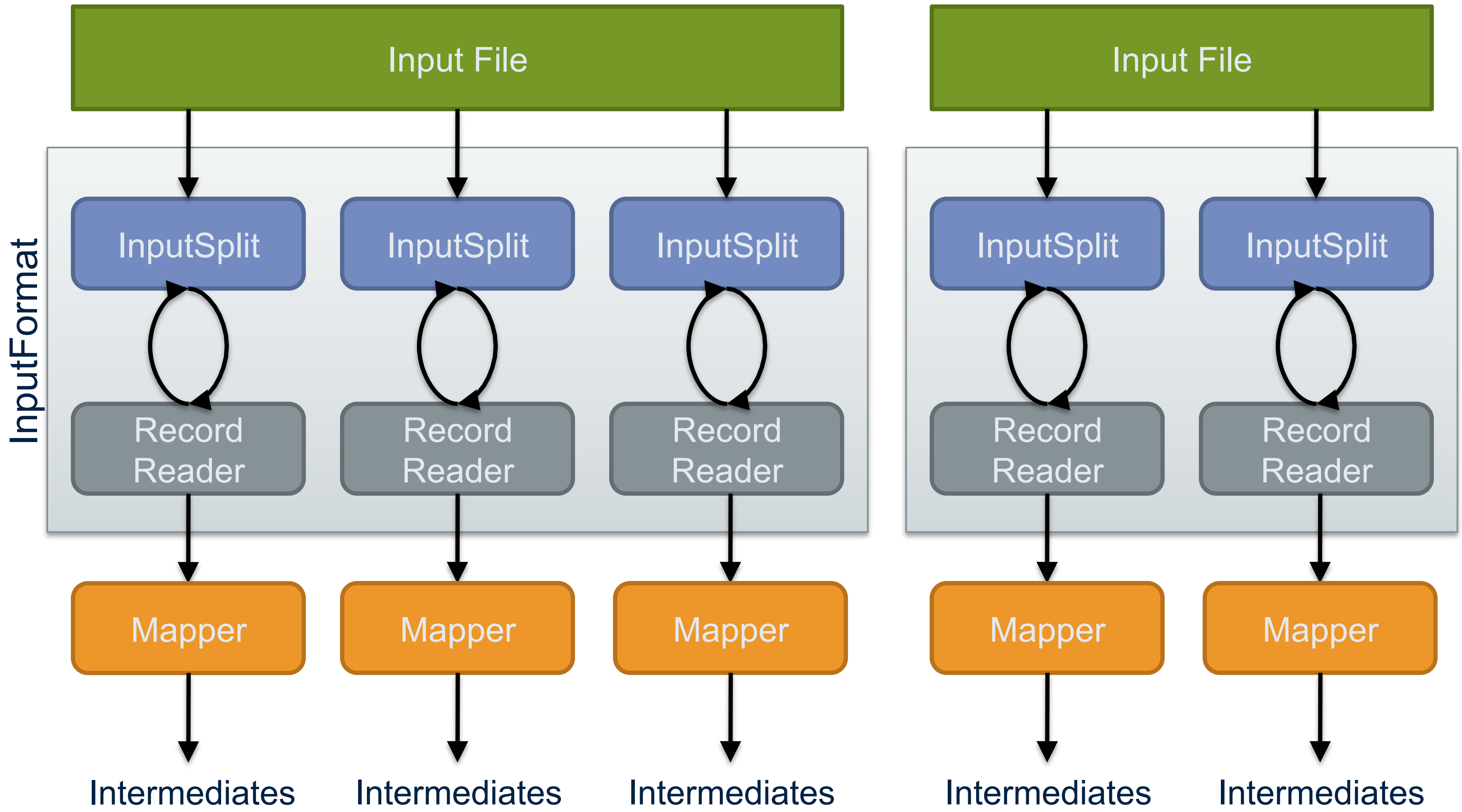

namenode
namenode daemon
job submission node
jobtracker
tasktracker
datanode daemon
Linux file system
slave node
tasktracker
datanode daemon
Linux file system
slave node
tasktracker
datanode daemon
Linux file system
slave node

Anatomy of a job

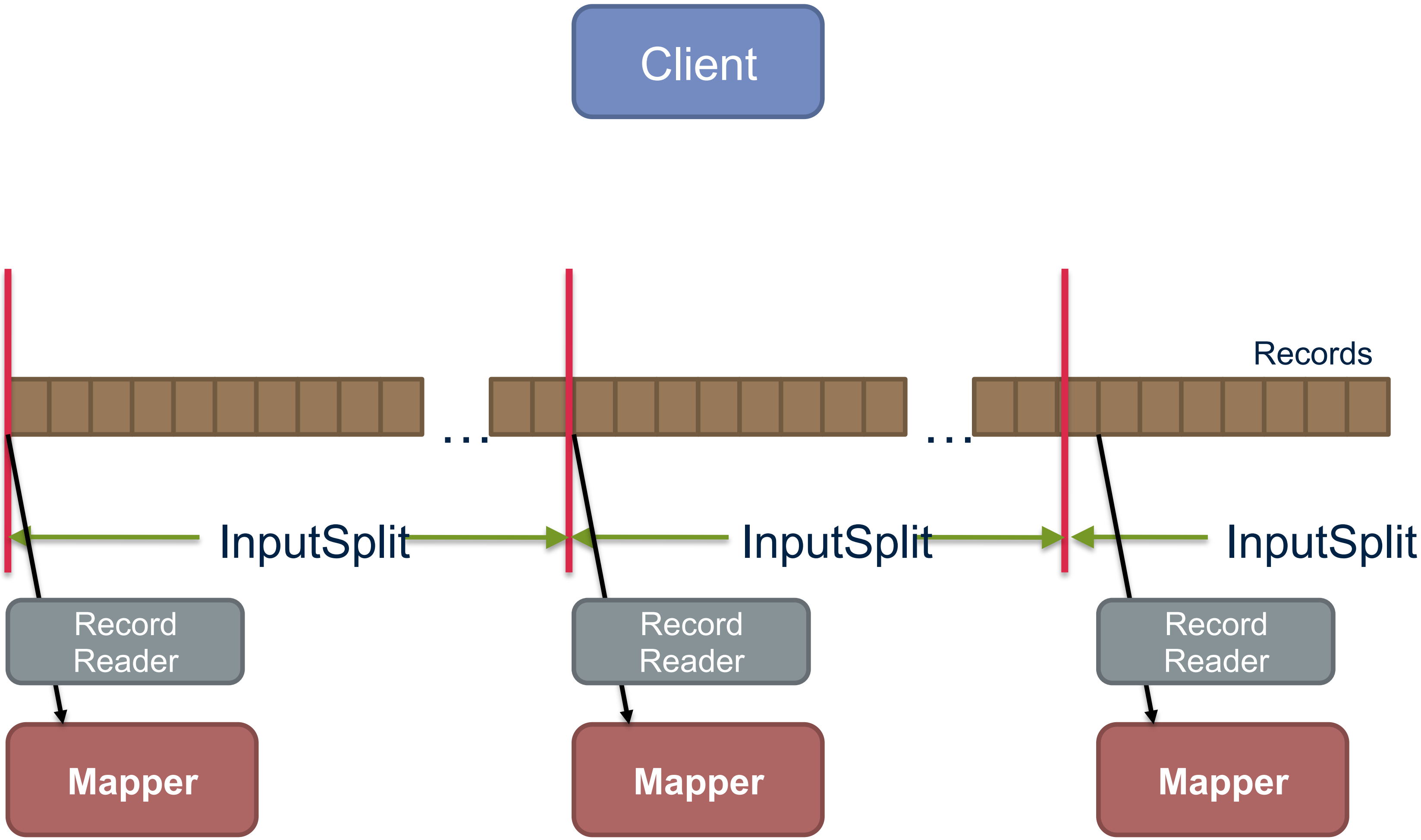
- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of running a task is called a task attempt (occupies a slot)
  - Multiple jobs can be composed into a workflow
- Job submission:
  - Client (i.e., driver program) creates a job, configures it, and submits it to jobtracker
  - That's it! The Hadoop cluster takes over

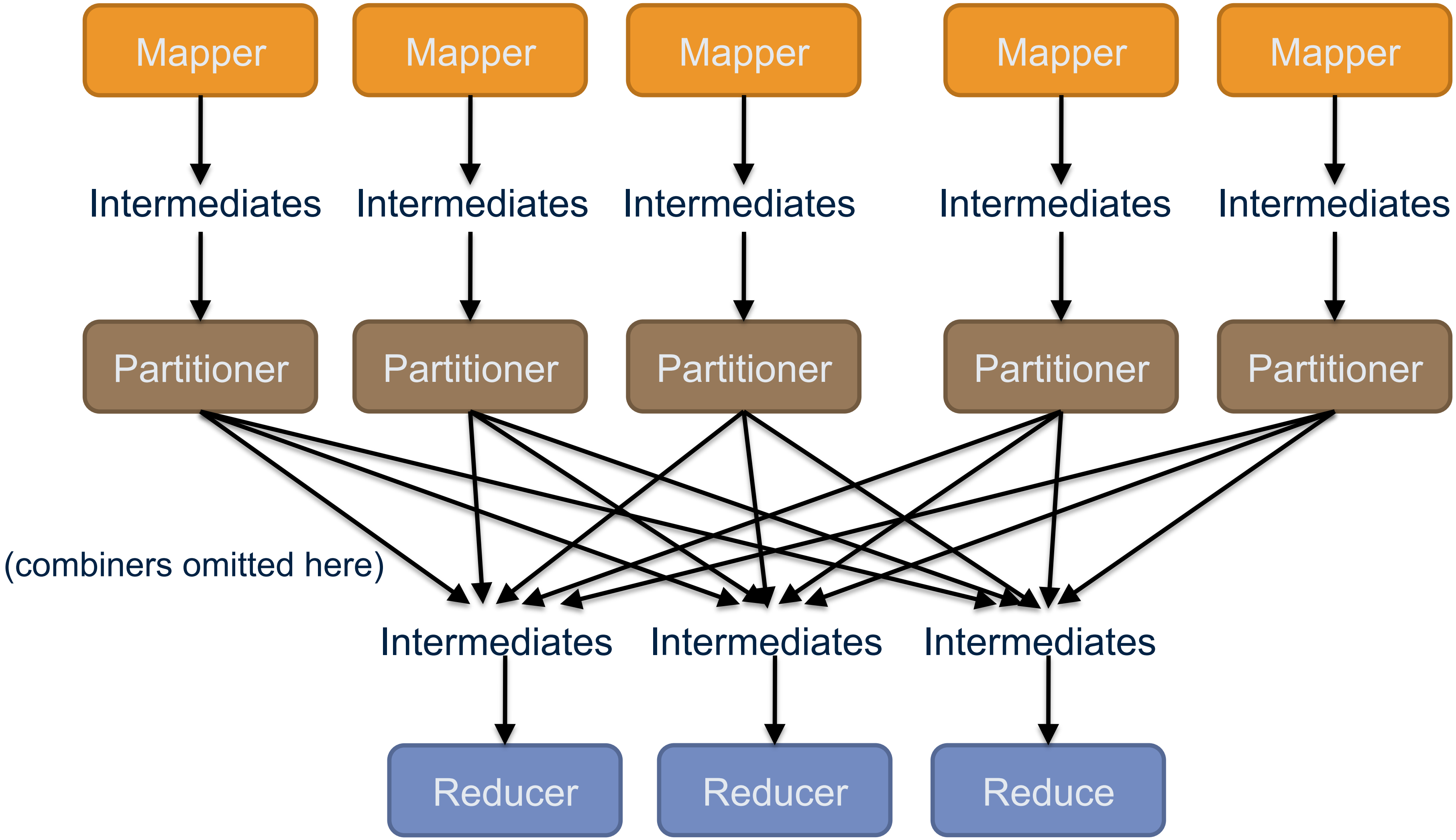
Anatomy of a job

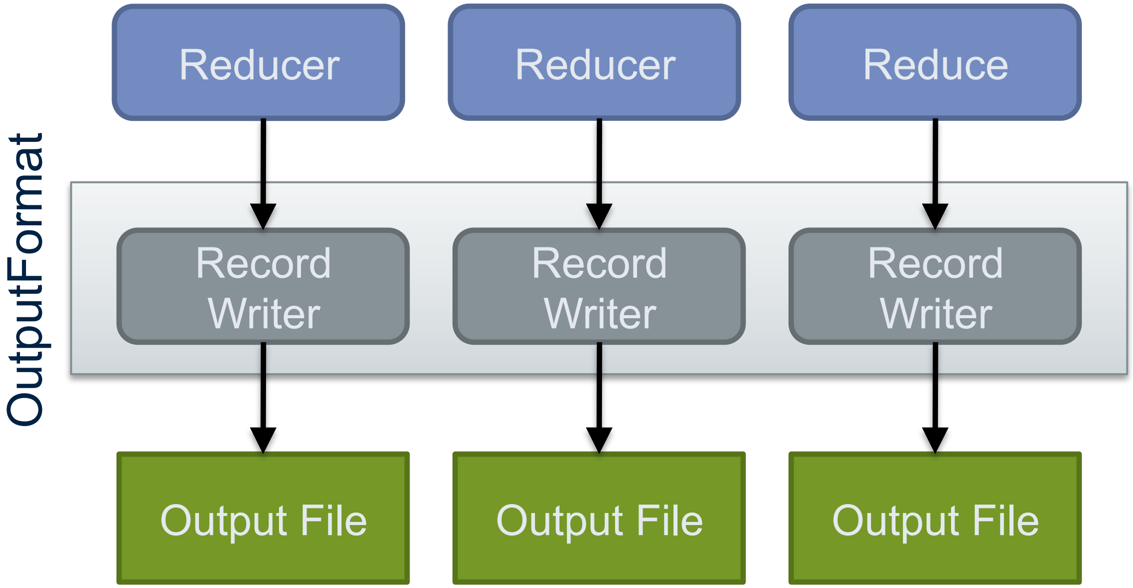
- Behind the scenes:
  - Input splits are computed (on client end)
  - Job data (jar, configuration XML) are sent to JobTracker
  - JobTracker puts job data in shared location, enqueues tasks
  - TaskTrackers poll for tasks
  - Off to the races


Input File
Input File
InputFormat
InputSplit
InputSplit
InputSplit
InputSplit
InputSplit
Record Reader
Record Reader
Record Reader
Record Reader
Record Reader
Mapper
Mapper
Mapper
Mapper
Mapper
Intermediates
Intermediates
Intermediates
Intermediates
Intermediates

THE UNIVERSITY of EDINBURGH
informatics




Mapper
Mapper
Mapper
Mapper
Mapper
Intermediates
Intermediates
Intermediates
Intermediates
Intermediates
Partitioner
Partitioner
Partitioner
Partitioner
Partitioner
(combiners omitted here)
Intermediates
Intermediates
Intermediates
Reducer
Reducer
Reduce


Reducer
Reducer
Reduce
OutputFormat
Record Writer
Record Writer
Record Writer
Output File
Output File
Output File

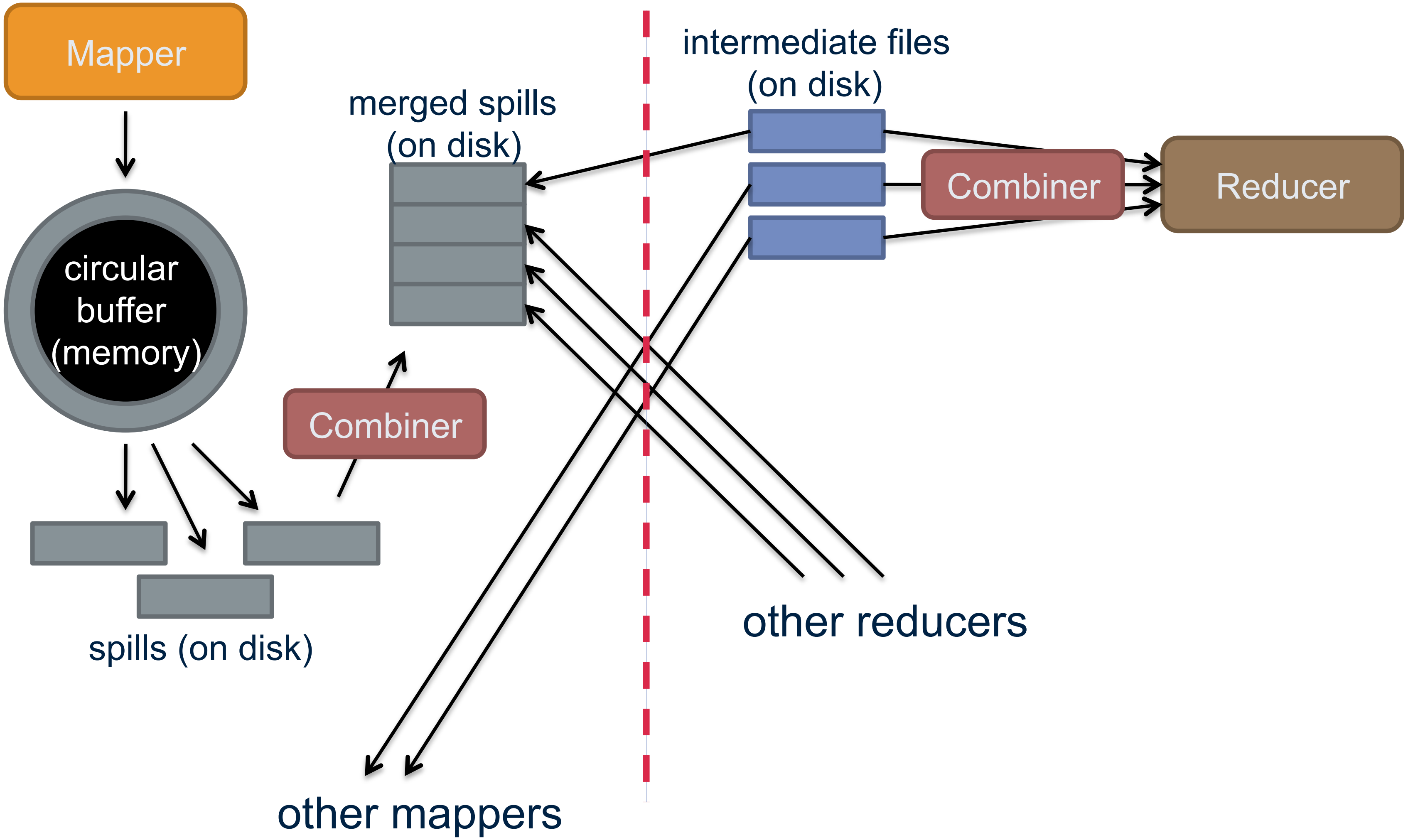
Input and output

- InputFormat:
  - TextInputFormat
  - KeyValueTextInputFormat
  - SequenceFileInputFormat
  - …
- OutputFormat:
  - TextOutputFormat
  - SequenceFileOutputFormat
  - …

Shuffle and sort in Hadoop

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

Shuffle and sort



Recommended workflow

- Here's one way to work
  - Develop code in your favourite IDE on host machine
  - Build distribution on host machine
  - Check out copy of code on VM
  - Copy (i.e., scp) jars over to VM (in same directory structure)
  - Run job on VM
  - Iterate
- Avoid using the UI of the VM
  - Directly ssh into the VM
- Deploying the job
- `$HADOOP_CLASSPATH`
- `hadoop jar MYJAR.jar -D k1=v1 … -libjars foo.jar,bar.jar my.class.to.run arg1 arg2 arg3 …`

Actually running the job

- `$HADOOP_CLASSPATH`
- hadoop jar MYJAR.jar

```
hadoop jar MYJAR.jar
  -D k1=v1 ...
  -libjars foo.jar,bar.jar
  my.class.to.run arg1 arg2 arg3 …
```

Debugging Hadoop

- First, take a deep breath
- Start small, start locally
- Build incrementally
- Different ways to run code:
  - Plain Java
  - Local (standalone) mode
  - Pseudo-distributed mode
  - Fully-distributed mode
- Learn what's good for what

We will start by using Python bindings before deploying Java jobs

Hadoop debugging strategies

- Good ol' `System.out.println`
  - Learn to use the webapp to access logs
  - Logging preferred over `System.out.println`
  - Be careful how much you log!
- Fail on success
  - Throw `RuntimeExceptions` and capture state
- Programming is still programming
  - Use Hadoop as the glue
  - Implement core functionality outside mappers and reducers
  - Independently test (e.g., unit testing)
  - Compose (tested) components in mappers and reducers

Summary

- Presented Hadoop in more detail
- Described the implementation of the various components
- Described the workflow of building and deploying applications
- Things are a lot more complicated than this
- We will next turn to algorithmic design for MapReduce