Data-Intensive Computing with MapReduce

Session 2: Hadoop Nuts and Bolts

Jimmy Lin
University of Maryland
Thursday, January 31, 2013

Source: Wikipedia (The Scream)

Source: Wikipedia (Japanese rock garden)

How will I actually learn Hadoop?

- This class session
- Hadoop: The Definitive Guide
- RTFM
- RTFC(!)

Source: Wikipedia (Mahout)

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

*Note that there are two versions of the API!

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
  - Don't depend of defaults!

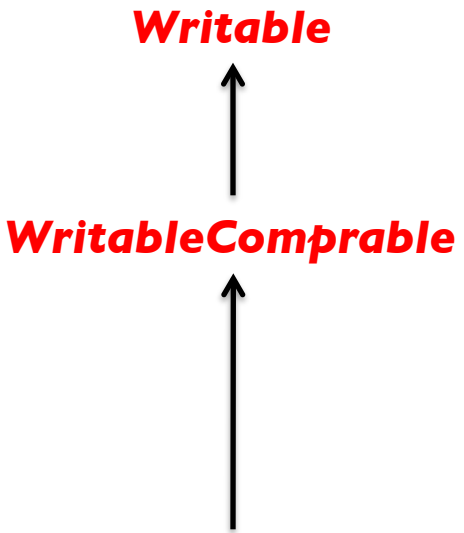
*Note that there are two versions of the API!

A tale of two packages...

org.apache.hadoop.mapreduce
org.apache.hadoop.mapred

Source: Wikipedia (Budapest)

Data Types in Hadoop: Keys and Values



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

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

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);
  }
}
```

Three Gotchas

- Avoid object creation at all costs
  - Reuse Writable objects, change the payload
- Execution framework reuses value object in reducer
- Passing parameters into mappers and reducers:
  - Directly in the Job object for parameters
  - DistributedCache for larger static data

Complex Data Types in Hadoop

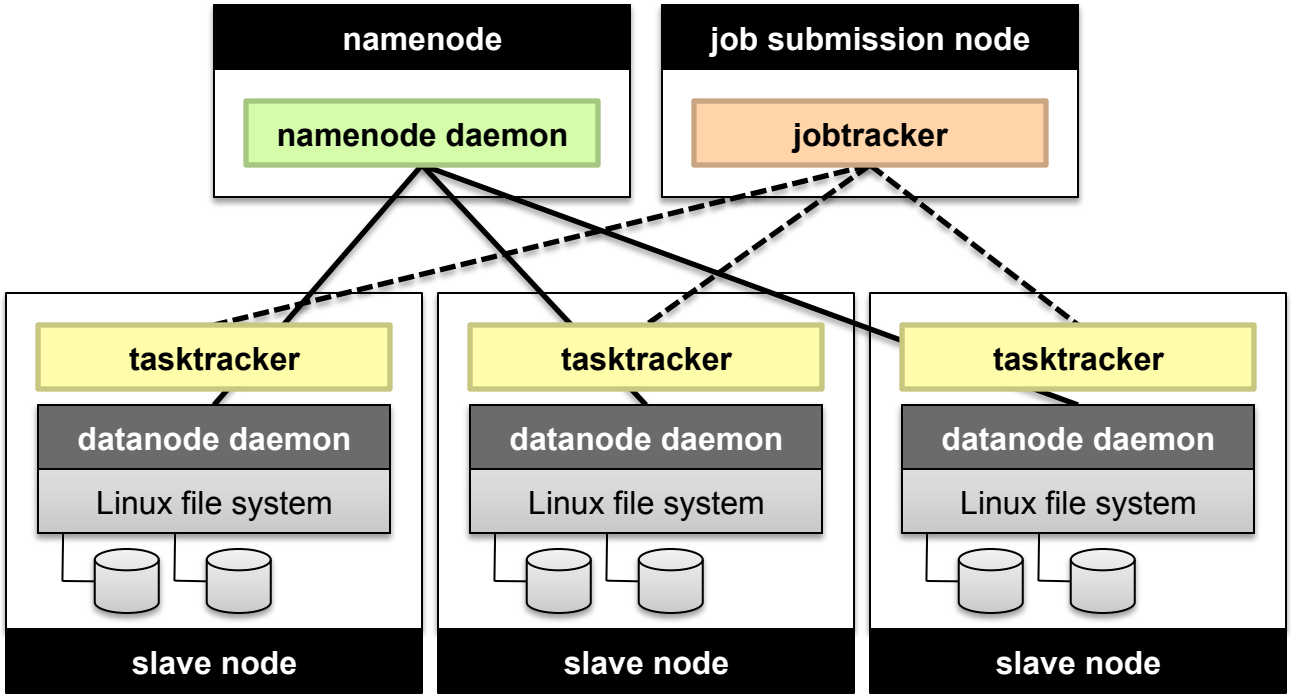
- How do you implement complex data types?
- The easiest way:
  - Encoded it as Text, e.g., (a, b) = "a:b"
  - Use regular expressions to parse and extract data
  - Works, but pretty hack-ish
- The hard way:
  - Define a custom implementation of Writable(Comprable)
  - Must implement: readFields, write, (compareTo)
  - Computationally efficient, but slow for rapid prototyping
  - Implement WritableComparator hook for performance
- Somewhere in the middle:
  - Cloud[9] 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

* Not quite… leaving aside YARN for now

Putting everything together…

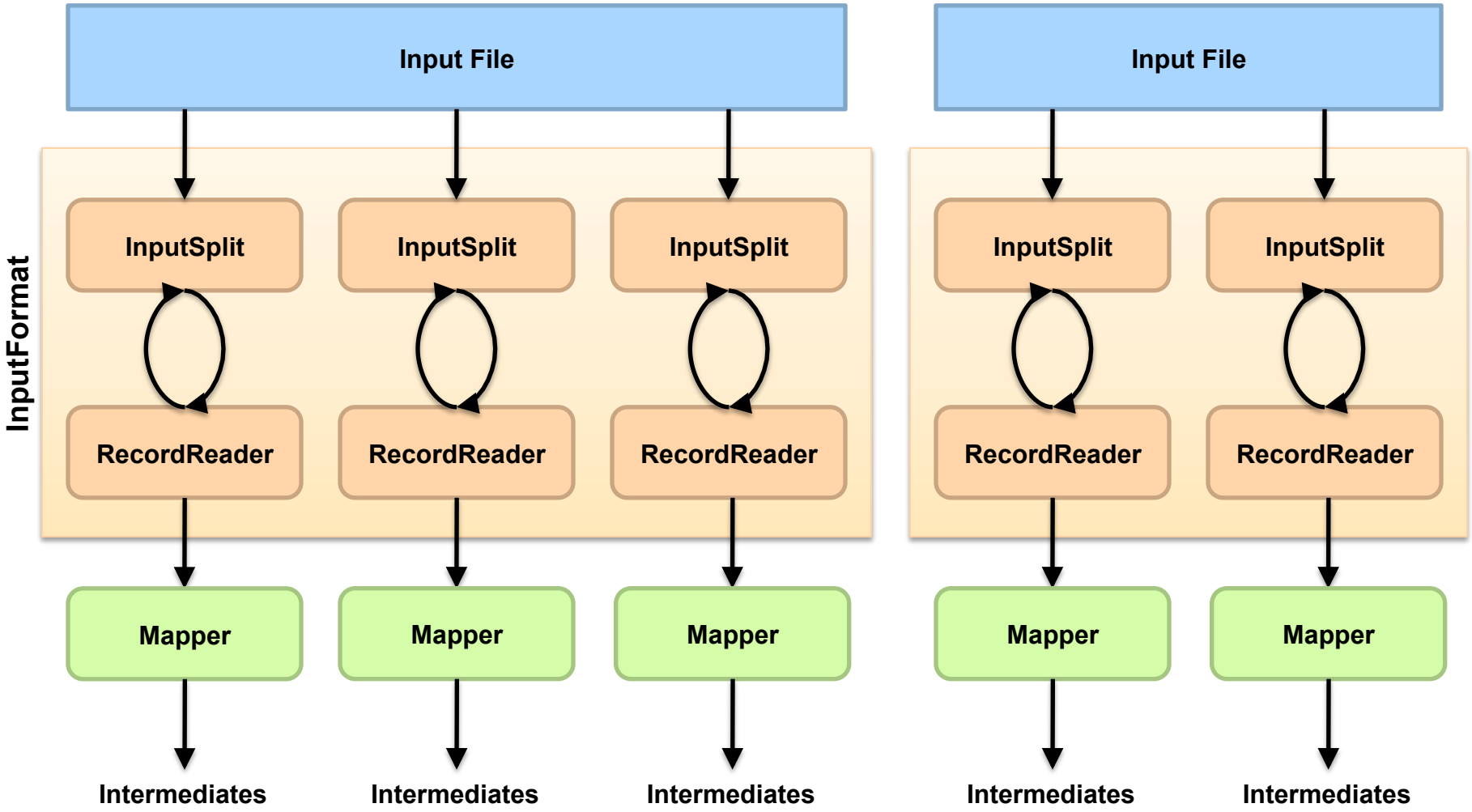


Anatomy of a Job

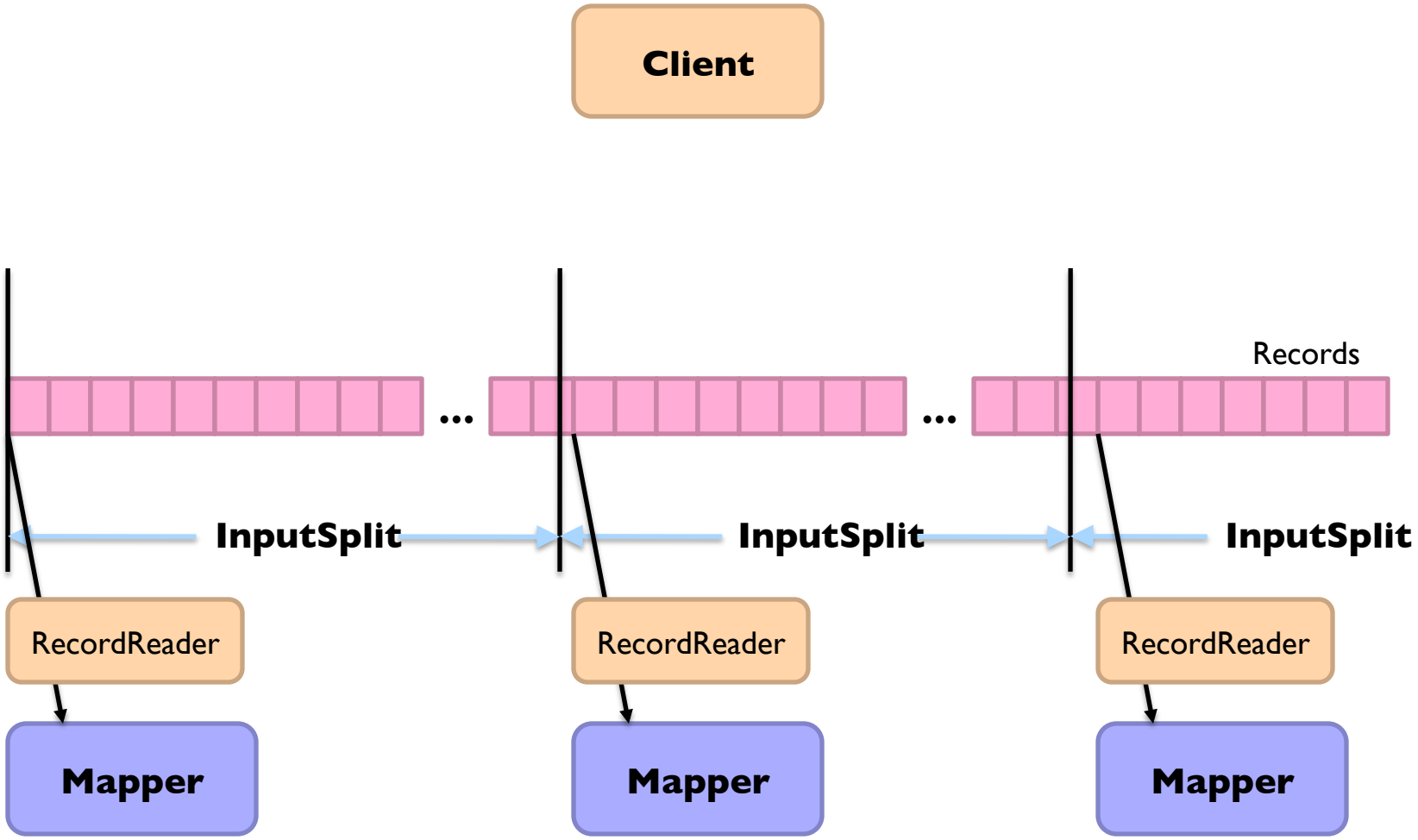
- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of running a task is called a task attempt (occupies a slot)
  - Multiple jobs can be composed into a workflow
- Job submission:
  - Client (i.e., driver program) creates a job, configures it, and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

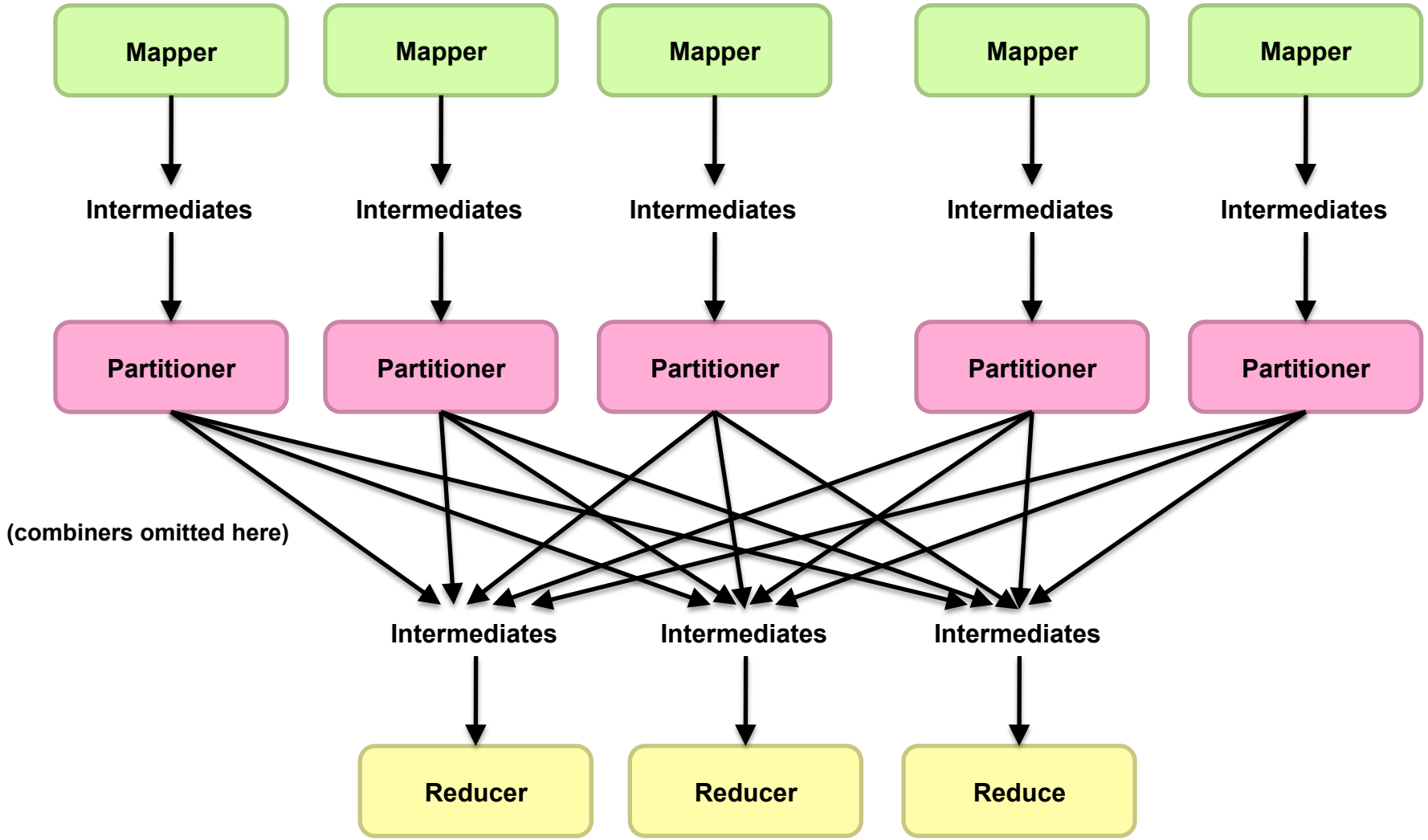
Anatomy of a Job

- 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…

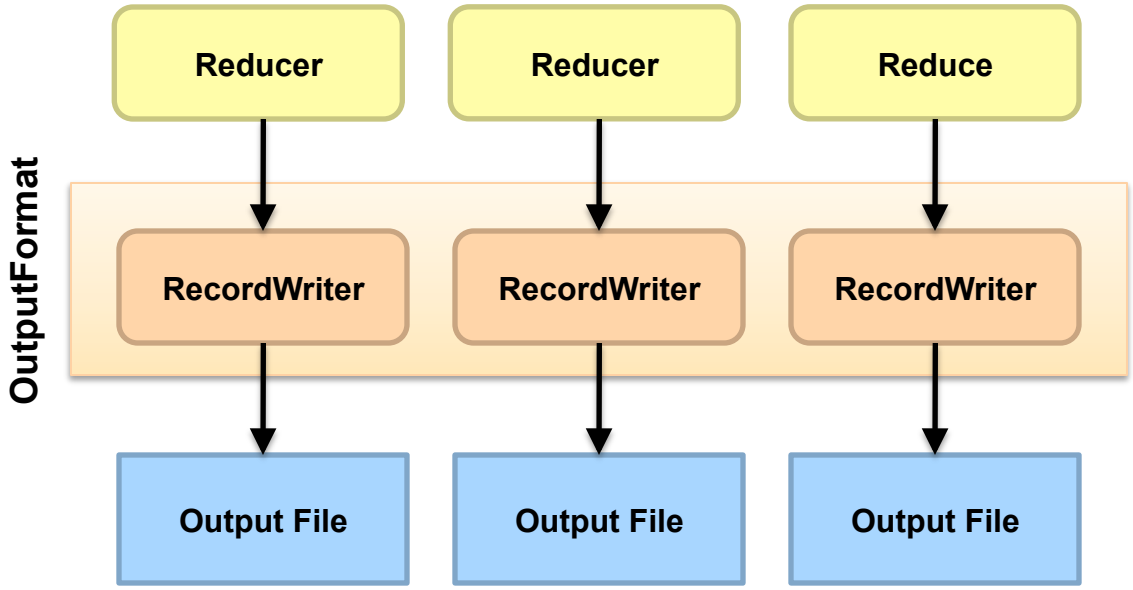


Source: redrawn from a slide by Cloduera, cc-licensed


Client
Records
InputSplit
InputSplit
InputSplit
RecordReader
RecordReader
RecordReader
Mapper
Mapper
Mapper



Source: redrawn from a slide by Cloduera, cc-licensed



Source: redrawn from a slide by Cloduera, cc-licensed

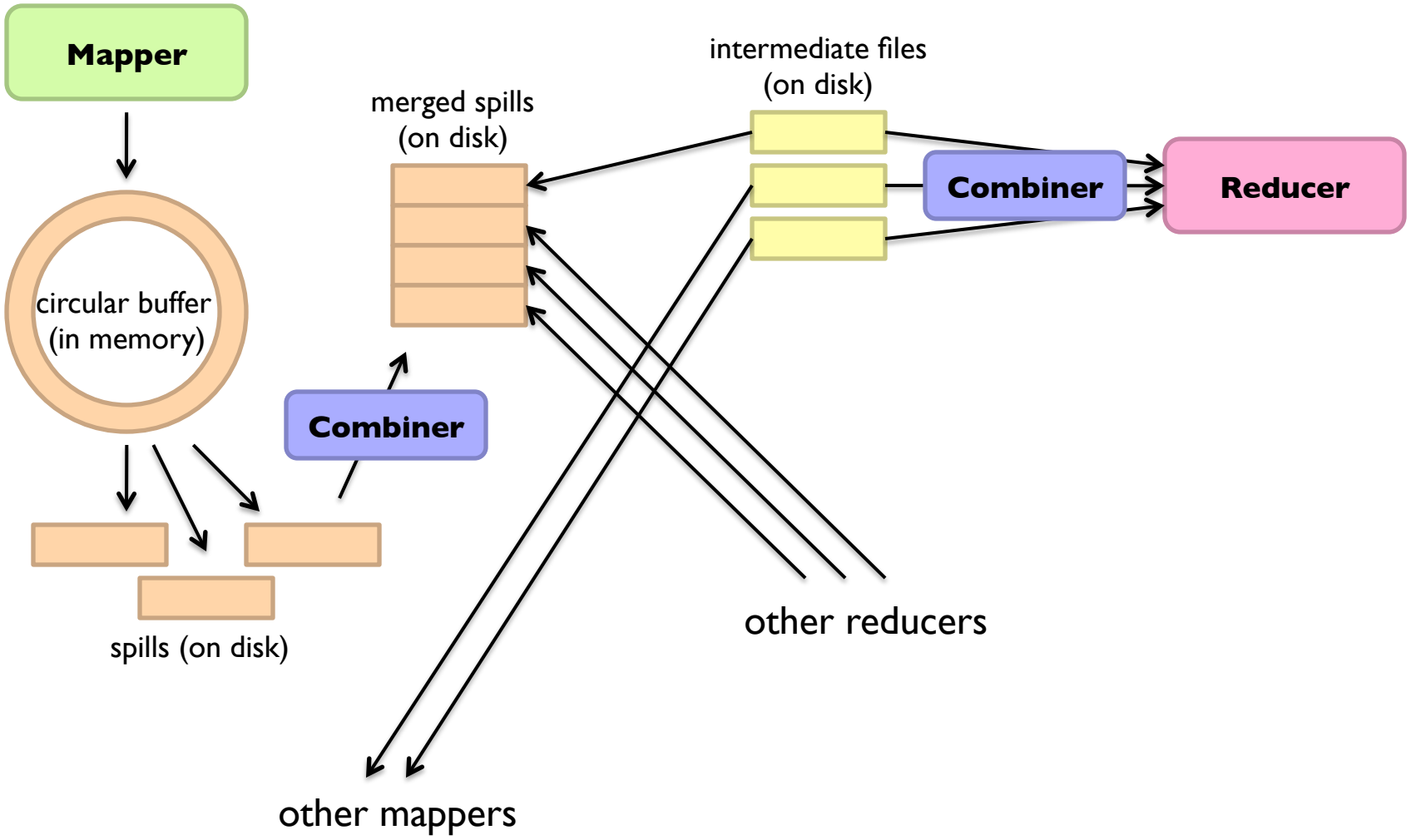
Input and Output

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

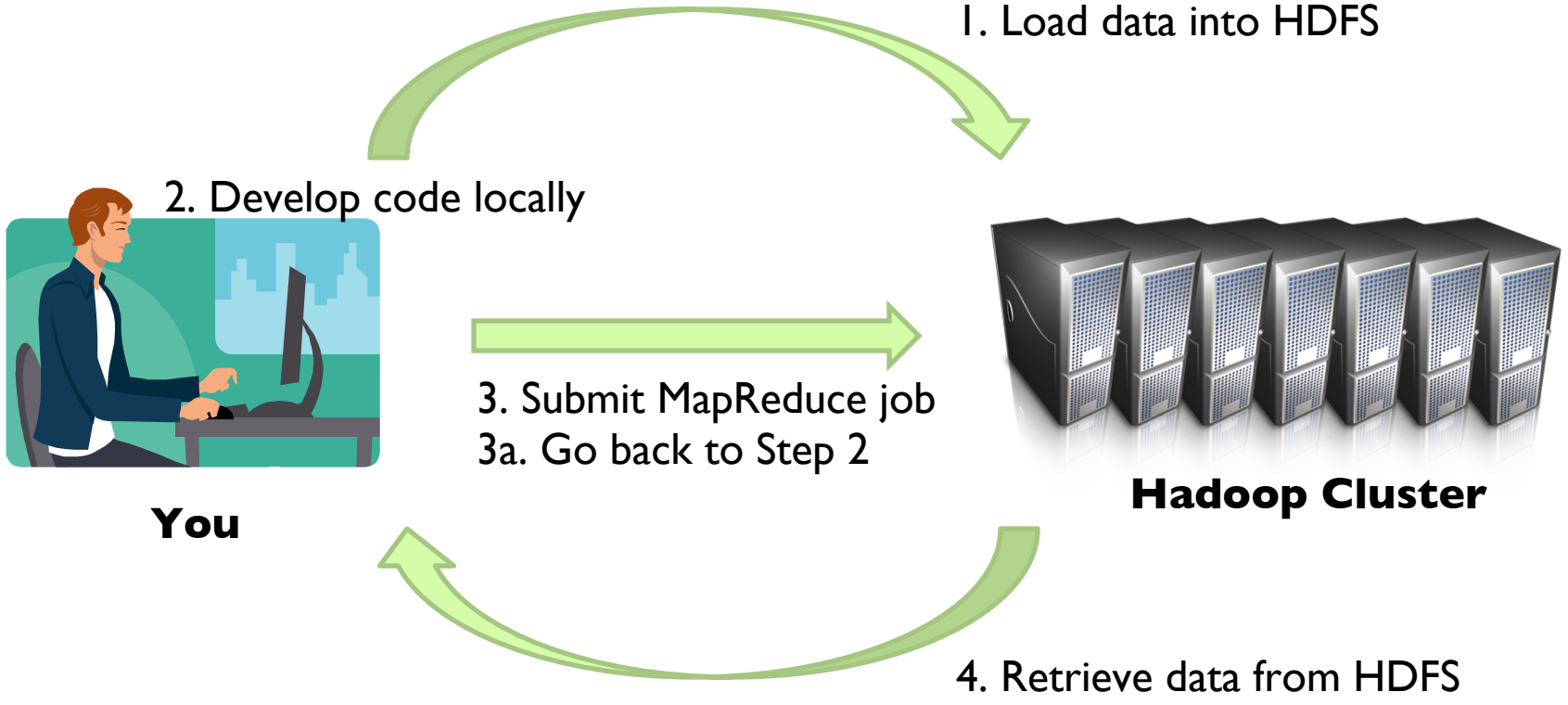
Shuffle and Sort in Hadoop

- 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



Hadoop Workflow



Debugging Hadoop

- First, take a deep breath
- Start small, start locally
- Build incrementally

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

Source: Wikipedia (Japanese rock garden)