Advanced Cloud Computing

Hadoop

Wei Wang
CSE@HKUST
Spring 2025

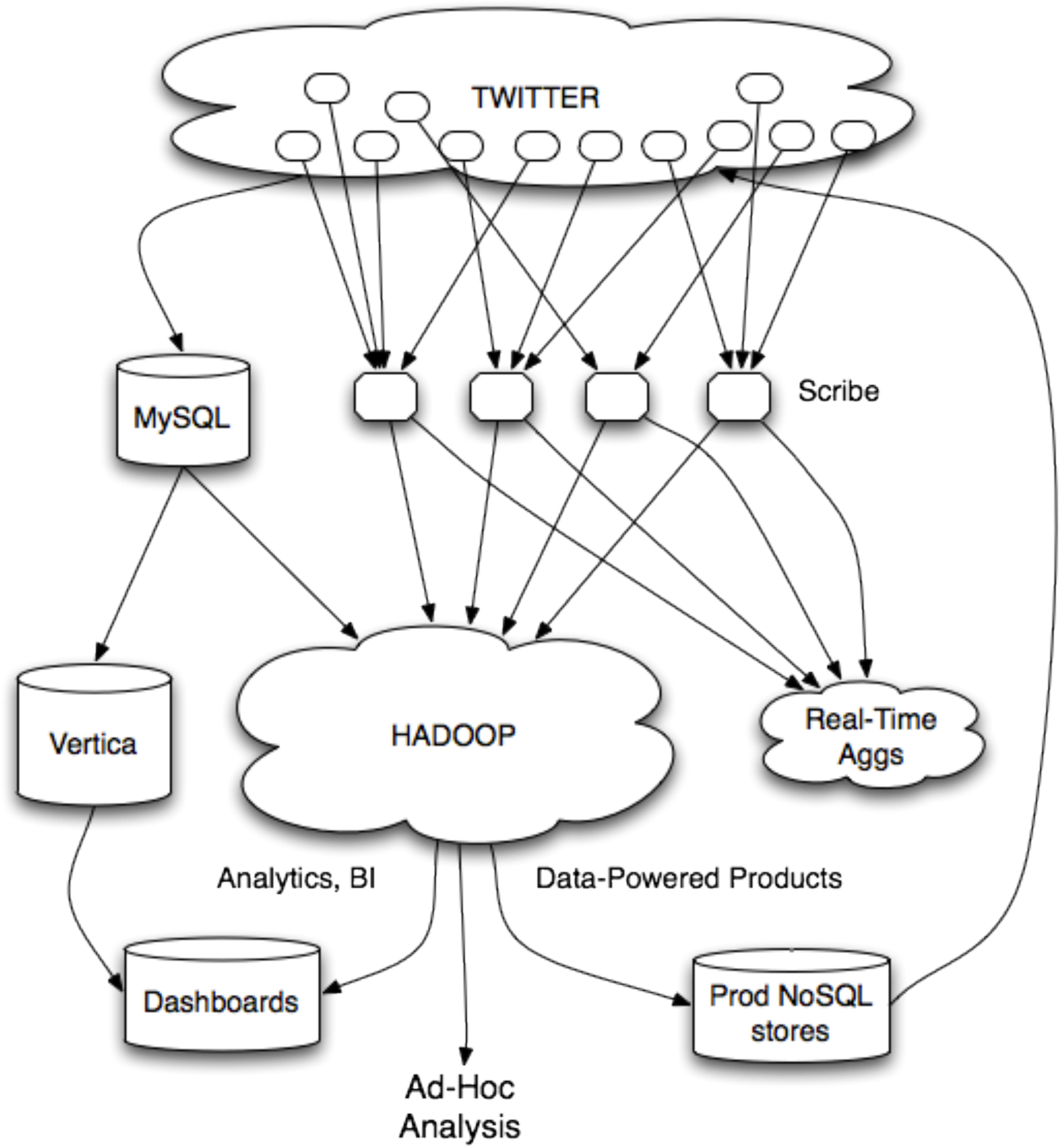
THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系

You say, “tomato…”

Google’s proprietary implementation	Open-source equivalent
MapReduce	Hadoop
GFS	HDFS
BigTable	HBase
Chubby	ZooKeeper

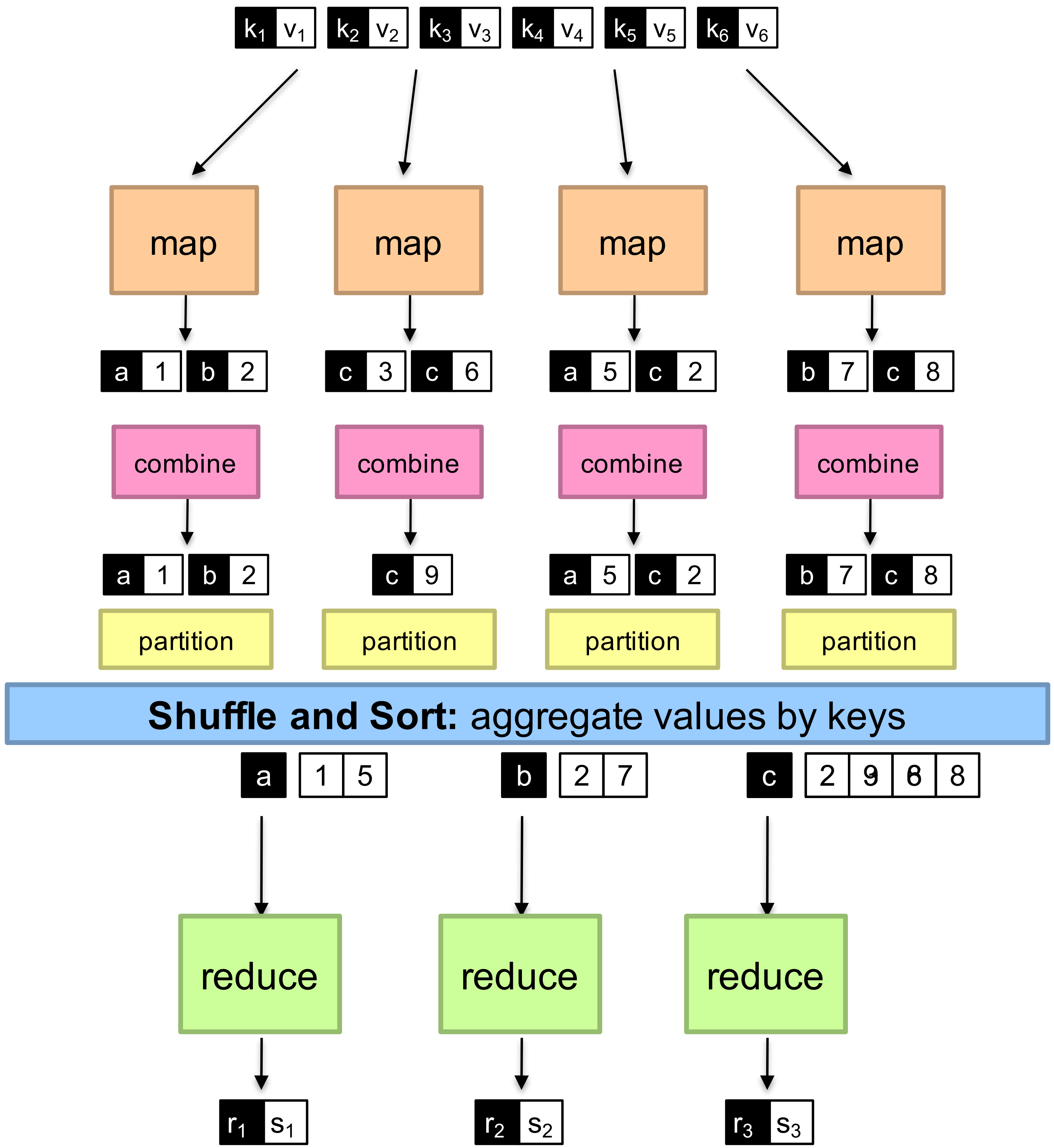
An open-source implementation of MapReduce in Java

- development led by Yahoo!, now an Apache project
- used in production at Yahoo!, Facebook, Twitter, LinkedIn, Netflix, …
- the *de facto* big data processing platform
- large and expanding software ecosystem
- lots of custom research implementations



Twitter’s data warehousing architecture

Source: Wikipedia (Mahout)



k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

Hadoop has two versions of API

org.apache.hadoop.mapreduce

org.apache.hadoop.mapred

Source: Wikipedia (Budapest)

Basic Hadoop API

Mapper

- **void setup(Mapper.Context context)**
  called once at the start 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

Basic Hadoop API

Reducer/Combiner

- **void setup(Reducer.Context context)**
  called once at the start of the task
- **void reduce(K key, Iterable<V> value, 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

Hadoop terminology

Job

- a packaged Hadoop program 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
- need to specify intermediate/final key/value classes
- need to specify number of reducers (but not mappers, why?)

Hadoop terminology

Task

- an execution of a mapper or a reducer on a slice of data, a.k.a., Task-In-Progress (TIP)

Task attempt

- a particular instance of an attempt to execute a task on a machine
- a particular task will be attempted at least once, possibly more times if it crashes

Terminology example

Running a WordCount across 20 files is one **job**

20 files to be mapped imply 20 **map tasks** + some number of **reduce tasks**

At least 20 map **task attempts** will be performed

- more if a machine crashes, etc.

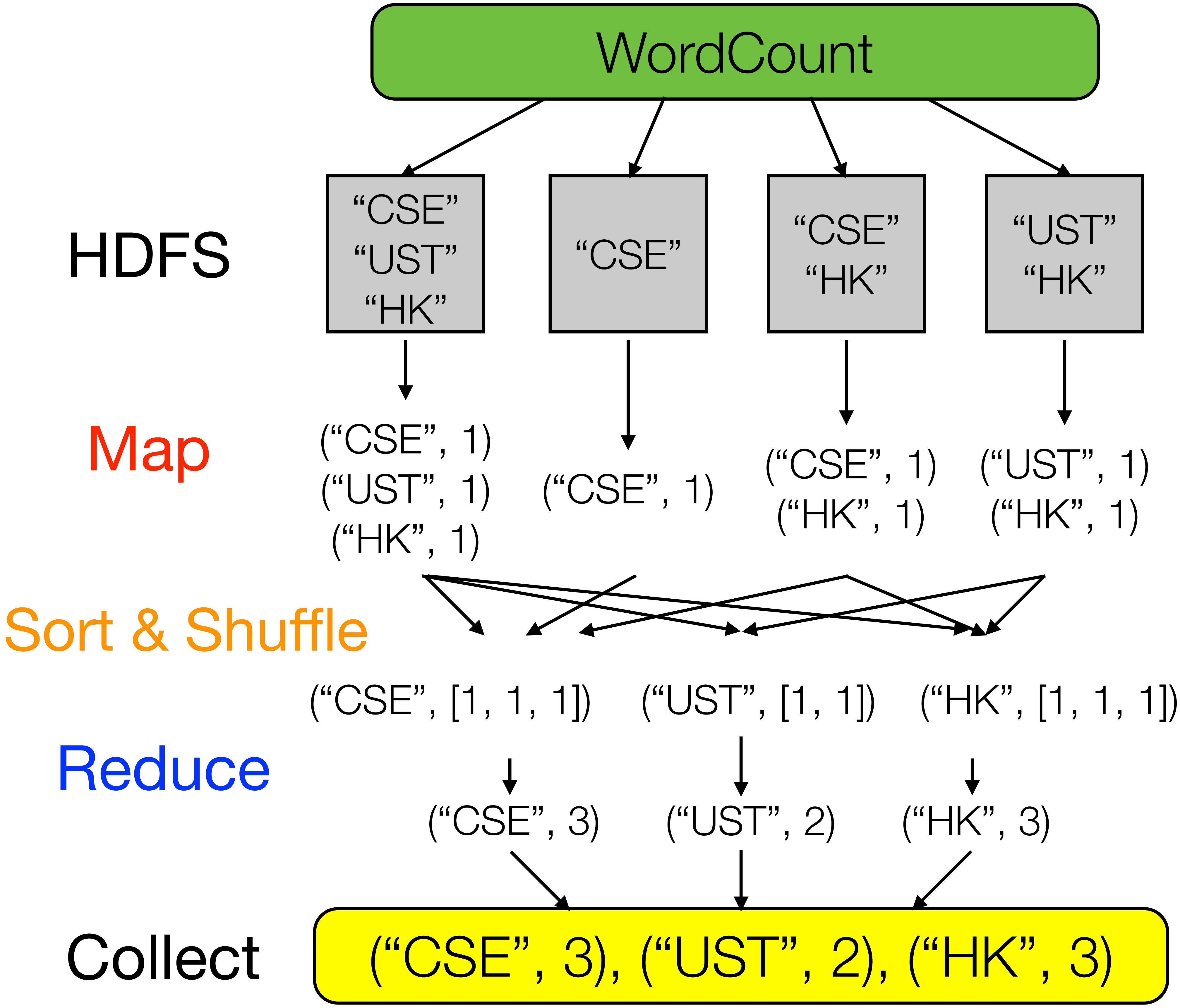
Data types in Hadoop

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.
SequenceFile	Binary encoded of a sequence of key/value pairs

Why not use Java Serialization?

Java comes with its own serialization mechanism. Why reinvent the wheel?

> "Because it (Java Serialization) looked **big** and **hairy** and I thought we needed something **lean** and **mean**, where we had precise control over exactly how objects are written and read, since that is central to Hadoop."
>
> —Dough Cutting


WordCount
HDFS
“CSE” “UST” “HK”
“CSE”
“CSE” “HK”
“UST” “HK”
Map
(“CSE”, 1) (“UST”, 1) (“HK”, 1)
(“CSE”, 1)
(“CSE”, 1) (“HK”, 1)
(“UST”, 1) (“HK”, 1)
Sort & Shuffle
(“CSE”, [1, 1, 1])
(“UST”, [1, 1])
(“HK”, [1, 1, 1])
Reduce
(“CSE”, 3)
(“UST”, 2)
(“HK”, 3)
Collect
(“CSE”, 3), (“UST”, 2), (“HK”, 3)

WordCount: pseudo code

```
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, sum)
```

WordCount: Mapper

Custom mapper inherits from the Mapper class:

map(k, v) → [<k', v'>]

```
private static class MyMapper
     extends Mapper<LongWritable, Text, Text, IntWritable> {
    // avoid creating objects on the fly
    private final static IntWritable ONE = new IntWritable(1);
    private final static Text WORD = new Text();

    @Override      // key = byte offset of each line; value = line text
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
      String line = ((Text) value).toString();
      String[] words = line.trim().split("\\s+");
      for (String w: words) {
        WORD.set(w);
        context.write(WORD, ONE);
      }
    }
  }
```

WordCount: Reducer

Custom reducer inherits from the Reducer class:

reduce(k, [v]) → <k', v'>

```
private static class MyReducer
     extends Reducer<Text, IntWritable, Text, IntWritable> {
   // avoid creating objects on the fly
   private final static IntWritable SUM = new IntWritable();

   @Override   // values with the same key are reduced together
   public void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
     int sum = 0;
     for (IntWritable v: values) {
          sum += v.get();
     }
     SUM.set(sum);
     context.write(key, SUM);
   }
 }
```

Three Gotchas

Avoid object creation whenever possible

- reuse **Writable** objects, change the payload

Execution framework reuses value object in reducer

Passing parameters via class statics

Configure the job and run it

```
// Create and configure a MapReduce job
Configuration conf = getConf();
Job job = Job.getInstance(conf);
job.setJobName("Word Count");
job.setJarByClass(WordCount.class);

job.setNumReduceTasks(reduceTasks); // Optional

// Specify inputs, outputs
FileInputFormat.setInputPaths(job, new Path(inputPath));
FileOutputFormat.setOutputPath(job, new Path(outputPath));

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

// Specify mapper, combiner, and reducer class
job.setMapperClass(WordCountMapper.class);
job.setCombinerClass(WordCountReducer.class);
job.setReducerClass(WordCountReducer.class);

// Run job and wait for its completion
System.exit(job.waitForCompletion(true) ? 0: 1);
```

Try it in Assignment-3

Sometimes, you may need complex data types, e.g., key as a pair of strings

Complex data types

The easiest way:

- encode it as **Text**, e.g., (a, b) = “a:b”
- use regular expressions to parse and extract data
- works but pretty “hack-ish" and hard to read

Complex data types

The standard (and 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

- third-party implementations: there are plenty of them!

Example: PairOfStrings

Anatomy of Hadoop

Basic cluster components

One of each cluster:

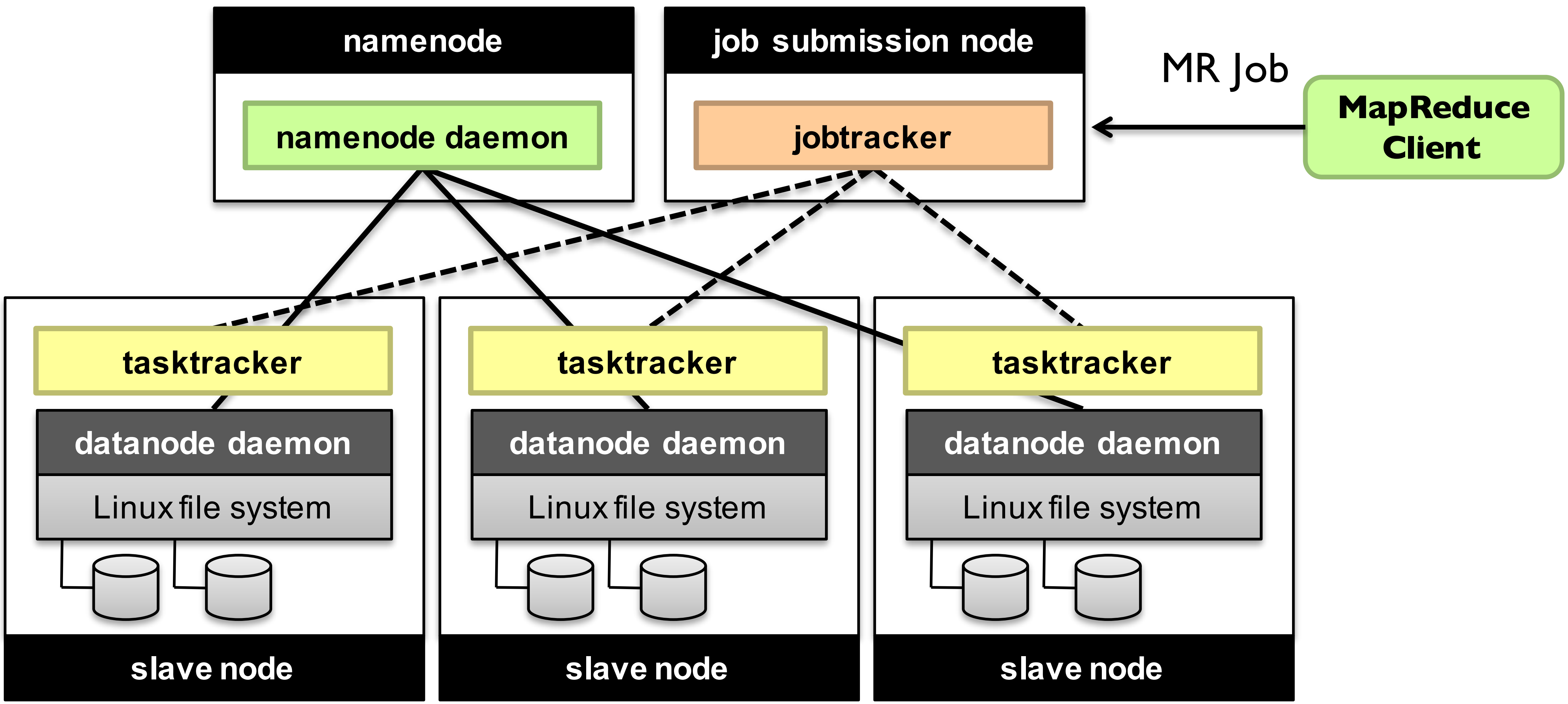
- **NameNode** (NN): master node for HDFS
- **JobTracker** (JT): master node for job submission

Set of each per slave machine:

- **DataNode** (DN): serves HDFS data blocks
- **TaskTracker** (TT): contains multiple task slots

Hadoop = HDFS + MapReduce

When HDFS meets MapReduce



Node-to-node communications

Hadoop uses its own RPC protocol

All communication begins in slave nodes

- prevents circular-wait deadlock
- slaves periodically poll for “status” message

Classes must provide explicit serialization

- that’s why Hadoop data type must inherits from **Writable**

Nodes, trackers, tasks

Master node runs **JobTracker** instance, which accepts **Job** requests from clients

TaskTracker instances run on slave nodes

TaskTracker forks separate Java process for task instances

Anatomy of a job

MapReduce program in Hadoop = Hadoop job

- jobs are divided into map and reduce tasks
- multiple jobs can be composed into a **workflow**
    - map->reduce->map->reduce->…

Job distribution

Job submission:

- client (i.e., driver program) creates a job, configures it, and submits it to **JobTracker**
- “jar” file + an XML file containing serialized program configuration options

Running a MapReduce job

- places “jar” file and XML file into the HDFS
- notifies **TaskTrackers** where to retrieve the relevant program code

Under the hood

Input splits are computed (on the client end)

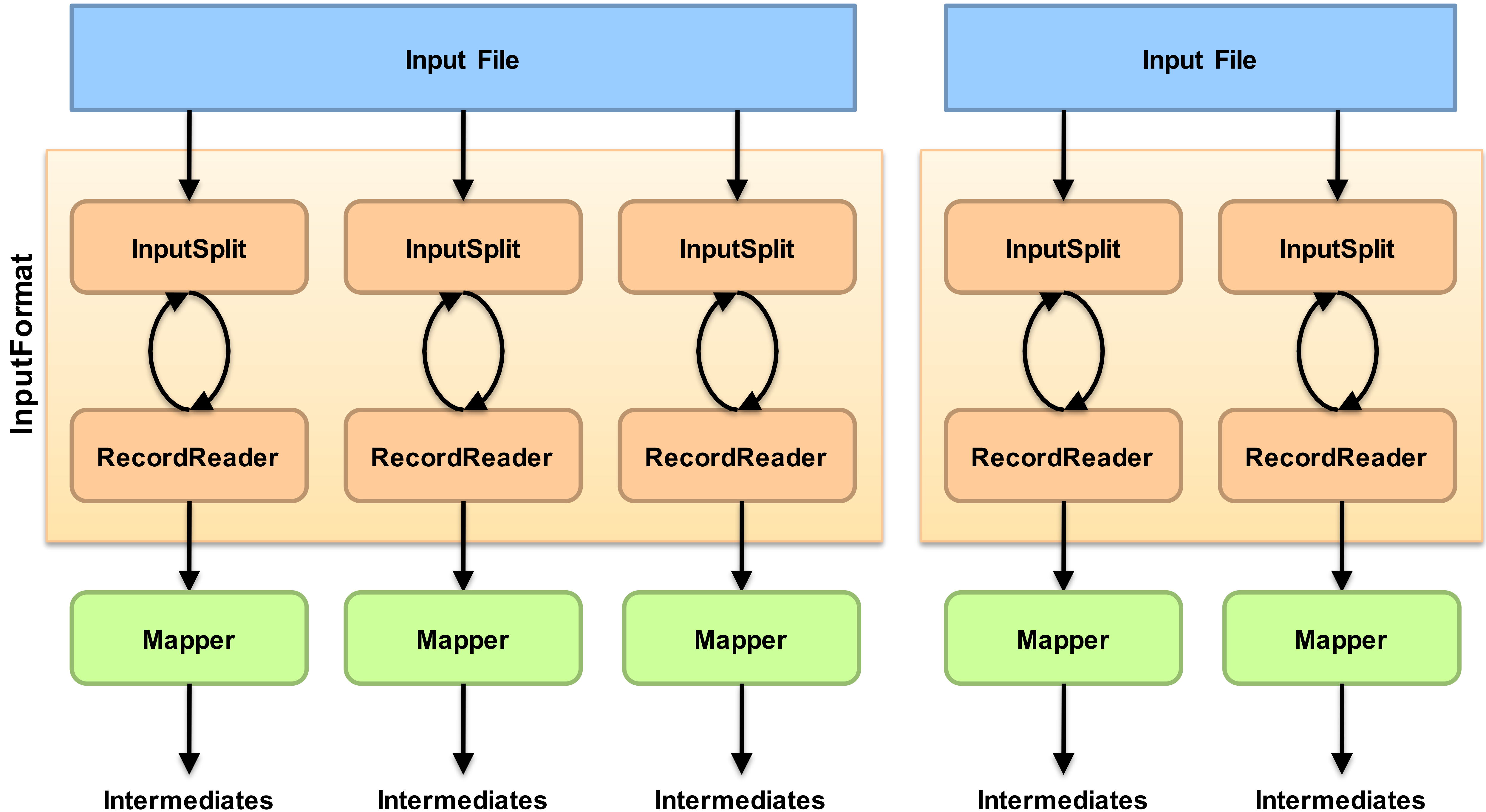
Job data sent to **JobTracker**

- jar + configuration XML

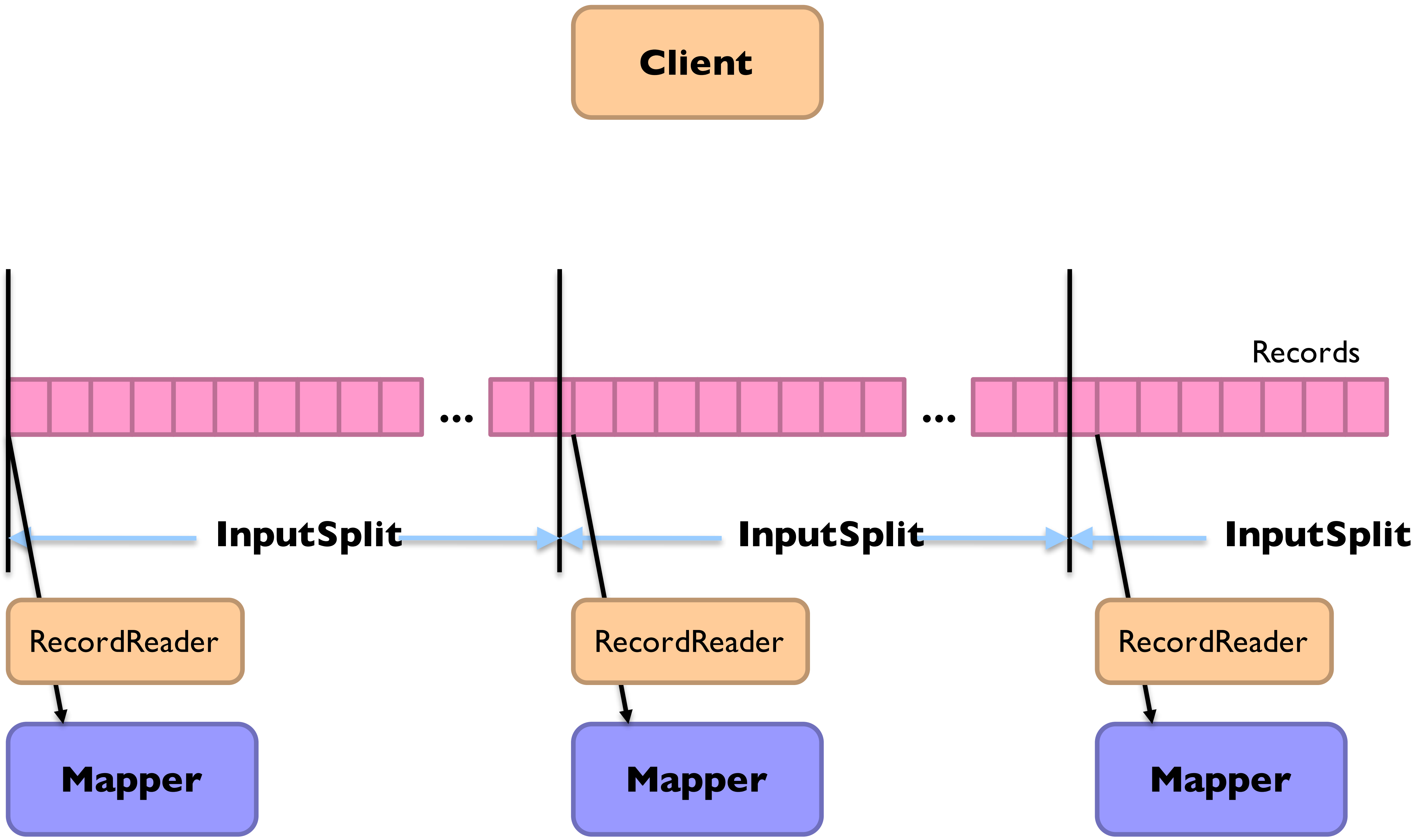
JobTracker puts job data in shared location (HDFS), enqueues tasks

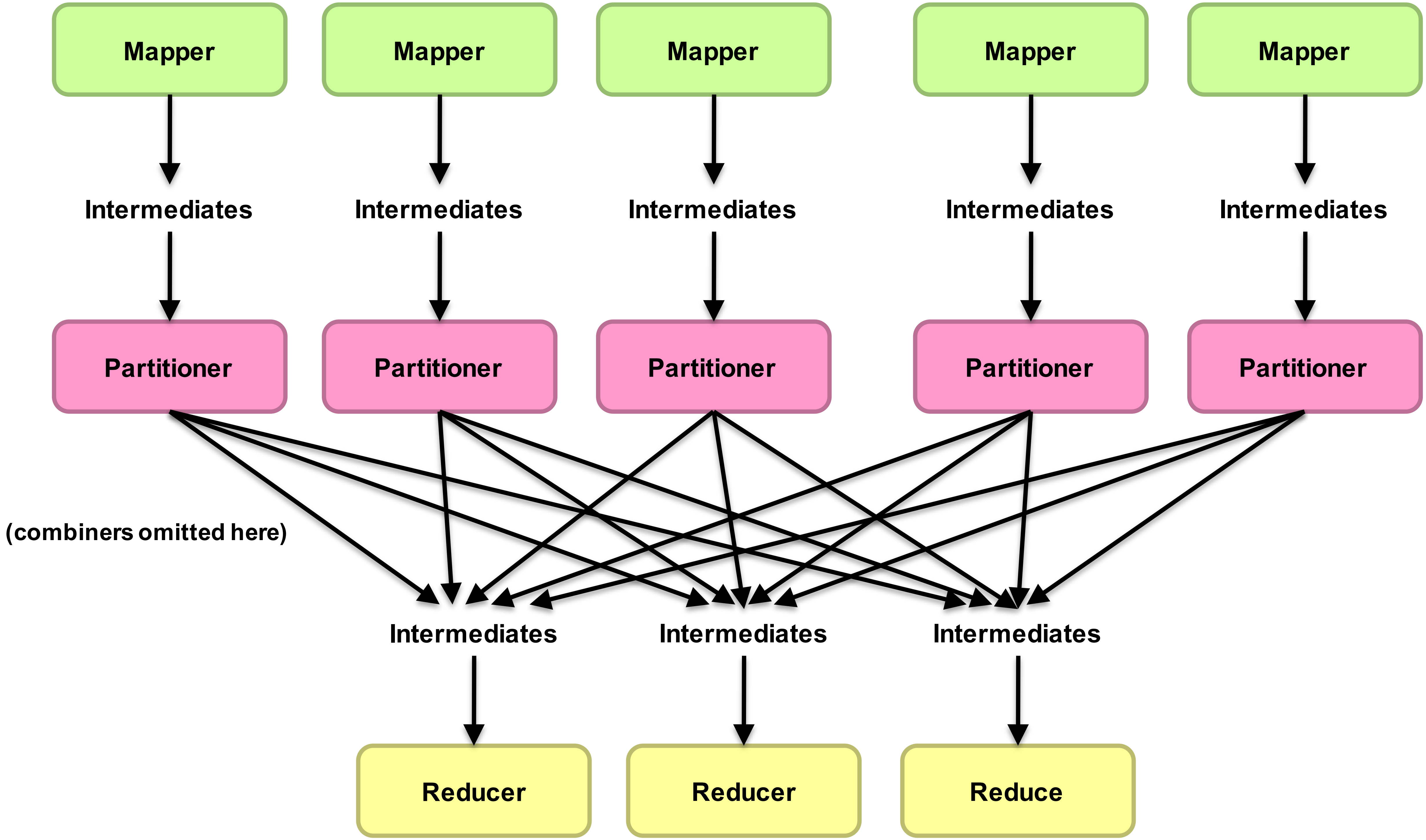
TaskTrackers poll for tasks

Off to the races…

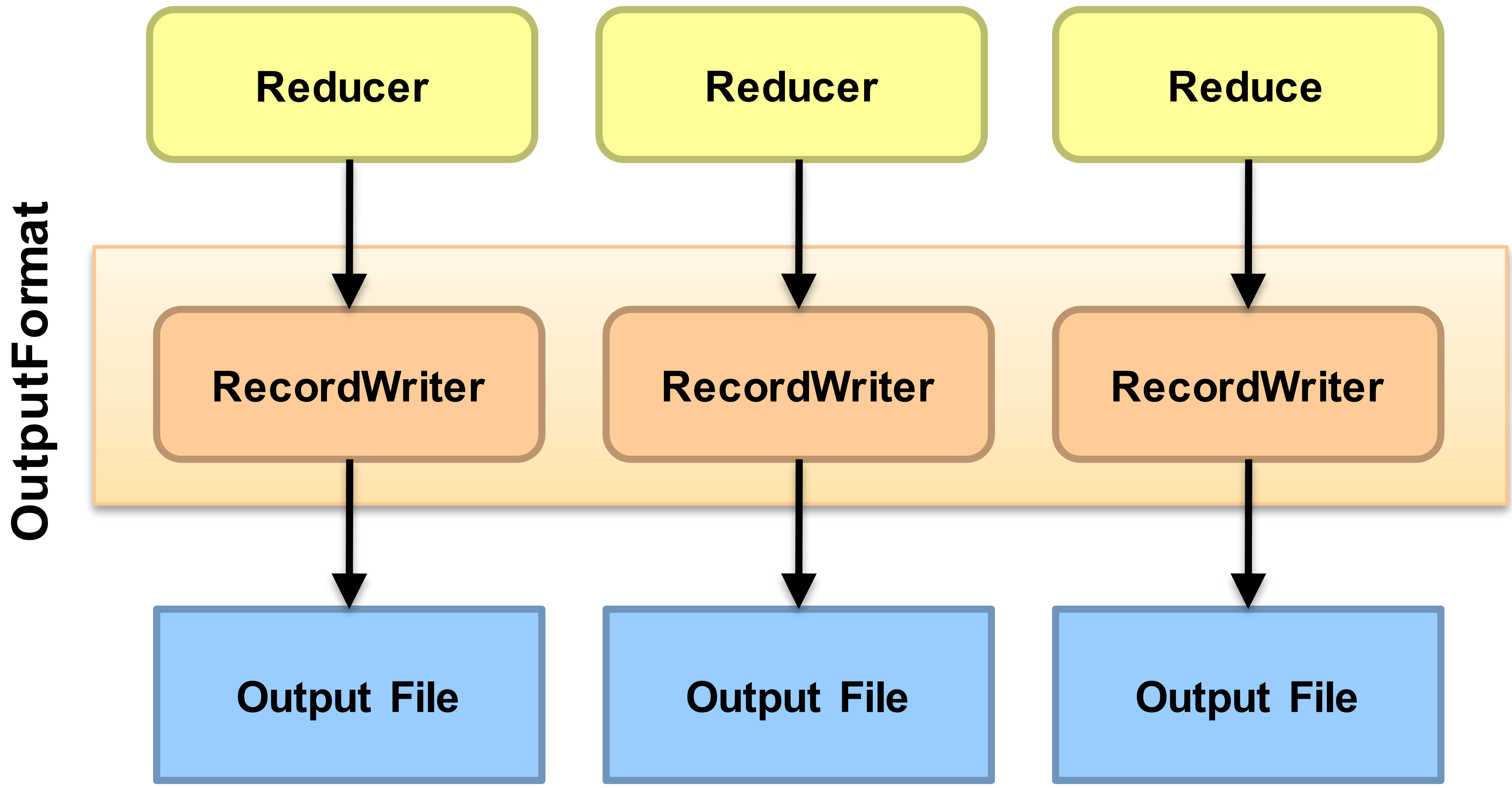


Source: redrawn from a slide by Cloduera, cc-licensed


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

Hadoop I/O

InputFormat:

- **TextInputFormat**: treats each '\n'-terminated line of a file as a value
- **KeyValueTextInputFormat**: maps '\n'-terminated text lines of "k v"
- **SequenceFileInputFormat**: binary file of (k, v) pairs
- …

OutputFormat:

- **TextOutputFormat**: writes "key val\n" strings to output file
- **SequenceFileOutputFormat**: uses a binary format to pack (k, v) pairs
- …

Shuffle and sort in Hadoop

Probably the most complex aspect of MapReduce

Map

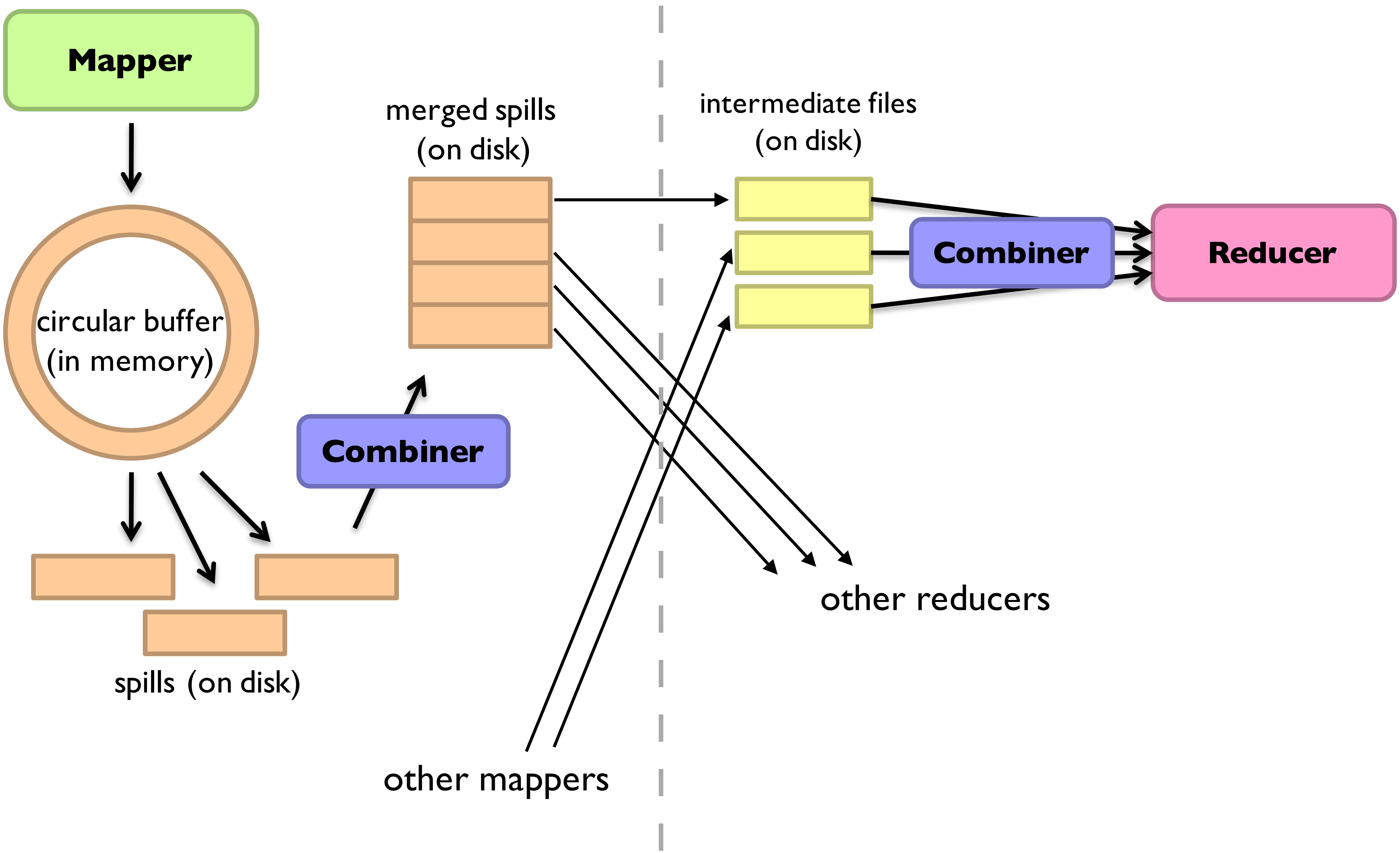
- Map outputs are buffered in memory in a circular buffer
- when buffer reaches threshold, contents are spilled to disk
- spills merged in a single, partitioned file (sorted within each partition): combiner runs during the merges

Shuffle and sort in Hadoop

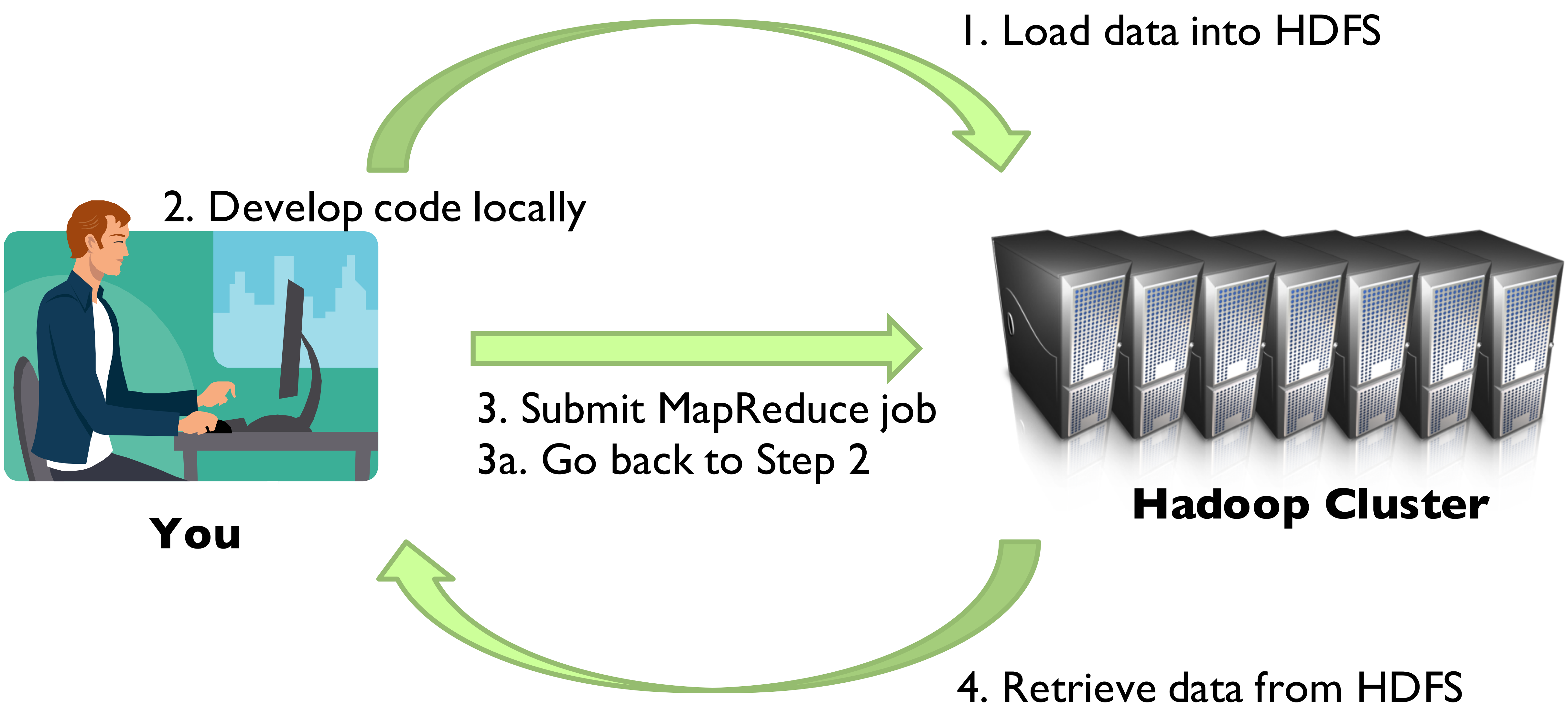
Reduce

- 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

Network


Mapper
circular buffer
(in memory)
spills (on disk)
Combiner
merged spills
(on disk)
other mappers
intermediate files
(on disk)
Combiner
Reducer
other reducers

Hadoop workflow



Hadoop workflow

Develop code in Eclipse on local machine

Build distribution on local machine

Check out copy of code on cluster

Compile and package

Run job on cluster

Iterate

Code execution environments

Different ways to run code:

- plain Java
- local (standalone) mode
- pseudo-distributed mode (emulate cluster nodes using multiple processes)
- fully-distributed mode

Debugging

- Start small, start locally
- Build incrementally

Credits

Slides are adapted from Prof. Jimmy Lin's slides at the University of Waterloo