Word Frequency Exercise

- What if we want to compute the word **frequency** instead of the word **count**?

- **Input**: large number of text documents
- **Output**: the word frequency of each word across all documents

- **Note**: Frequency is calculated using the **total word count**

- **Hint 1**: We know how to compute the total word count
- **Hint 2**: Can we use the word count output as input?

- **Solution**: Use two MapReduce tasks
  - MR1: count number of all words in the documents
  - MR2: count number of each word and divide it by the total count from MR1

Basic HADOOP API (1.x or 0.20.x)

- **Package org.apache.hadoop.mapreduce**
- **Class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>**
  - void setup(Mapper.Context context)
  - void cleanup(Mapper.Context context)
  - void map(KEYIN key, VALUEIN value, Mapper.Context context)
  - output is generated by invoking context.collect(key, value);
- **Class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>**
  - void setup(Reducer.Context context)
  - void cleanup(Reducer.Context context)
  - void reduce(KEYIN key, Iterable<VALUEIN> values, Reducer.Context context)
  - output is generated by invoking context.collect(key, value);
- **Class Partitioner<KEY, VALUE>**
  - abstract int getPartition(KEY key, VALUE value, int numPartitions)

JOB

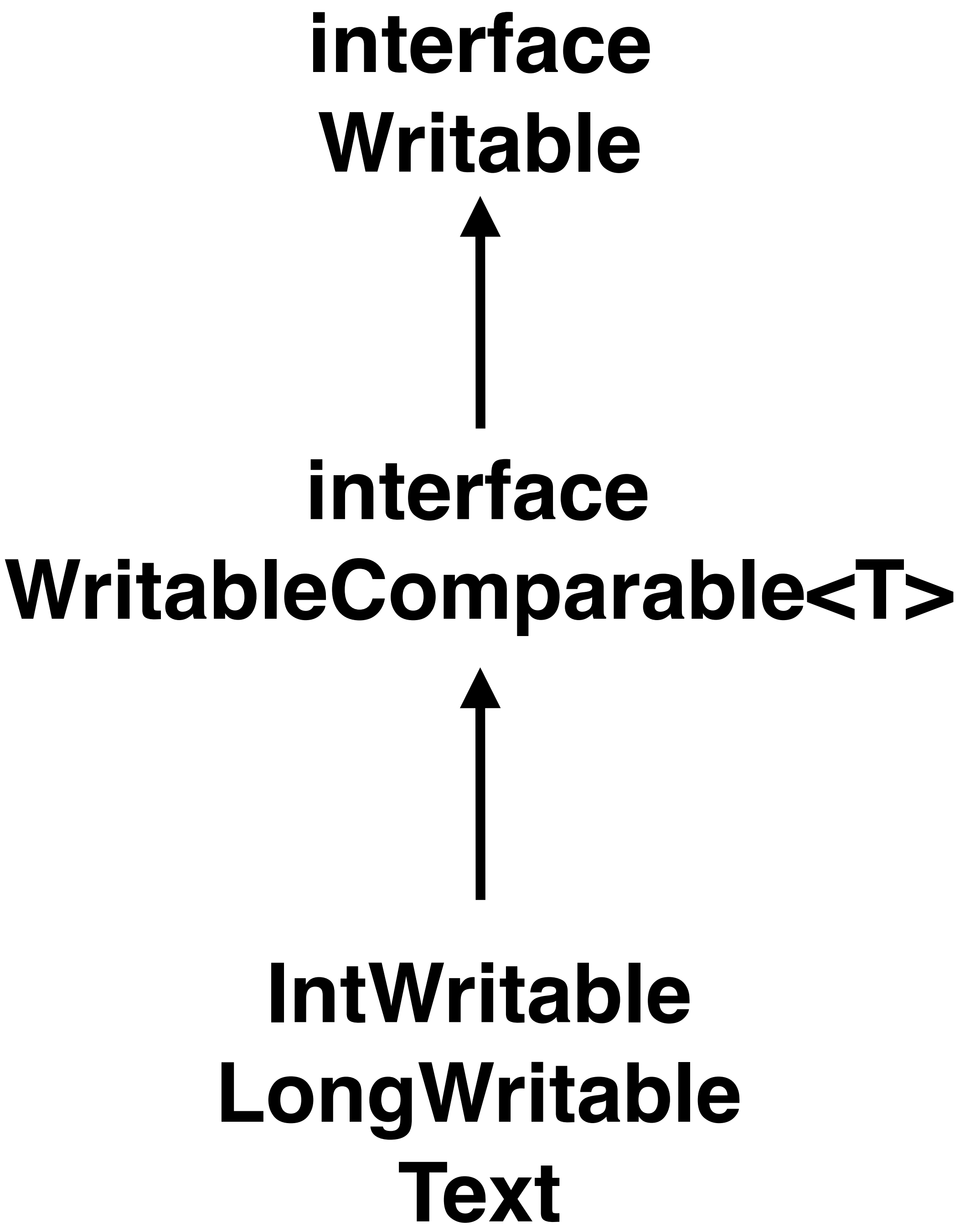
- Represents a packaged Hadoop job for submission to cluster
- Need to specify input and output paths
- Need to specify input and output formats
- Need to specify mapper, reducer, combiner, partitioner classes
- Need to specify intermediate/final key/value classes
- Need to specify number of reducers (but not mappers, why?)
- Don’t depend of defaults!

Basic HADOOP main (1.x or 0.20.x)

```
public static void main(String[] args) throws Exception
{
    Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");
    job.setJarByClass(WordCount.class);

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

    job.setMapperClass(NewMapper.class);
    job.setReducerClass(NewReducer.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Basic HADOOP Data Types (1.x or 0.20.x)

- **Package org.apache.hadoop.io**



interface Writable

Defines a de/serialization protocol

Any key or value type in the Hadoop Map-Reduce framework implements this interface

interface WritableComparable<T>

WritableComparables can be compared to each other, typically via Comparators

Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface

IntWritable
LongWritable
Text

Concrete classes for common data types

Complex HADOOP Data Types

- **Quick & Dirty way**
  - Encode key and value as Text object with custom separator
  - Example: ("blue", 14) becomes "blue_14" or "blue$14)
  - Use regular expressions or split() to extract data
  - Good for rapid prototyping, bad for performance
- **Standard way**
  - Define a custom implementation of WritableComparable<T>
  - Must implement
    - public void write(DataOutput out) throws IOException
    - public void readFields(DataInput in) throws IOException
    - public int compareTo(T o)
  - Should implement
    - public int hashCode()
    - public boolean equals(Object obj)
  - Good for performance, bad for rapid prototyping

Hello World in Hadoop (I)

```
class MAPPER
    method MAP(docid $a$, doc $d$)
        for all term $t \in$ doc $d$ do
            EMIT(term $t$, count 1)

class REDUCER
    method REDUCE(term $t$, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
            $sum \leftarrow sum + c$
        EMIT(term $t$, count $sum$)
```

Data-Intensive Text Processing with MapReduce

Hello World in Hadoop (II)

```
public static class MyMapper extends Mapper<Object, Text, Text, IntWritable>
{
   private final static IntWritable ONE = new IntWritable(1);
   private Text WORD = new Text();

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

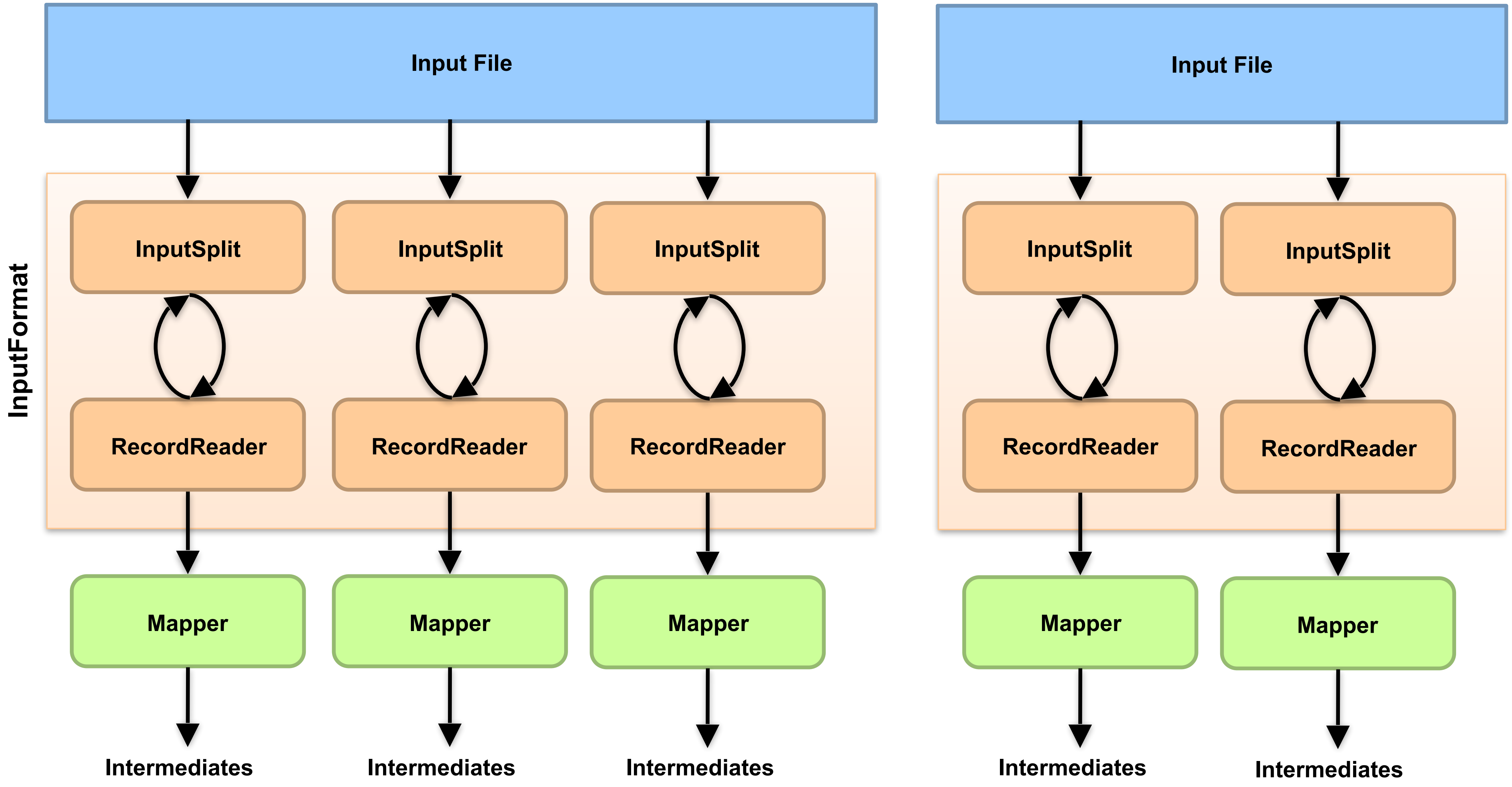
Hello World in Hadoop (III)

```
public static class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable>
{
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException
    {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

HADOOP tricks

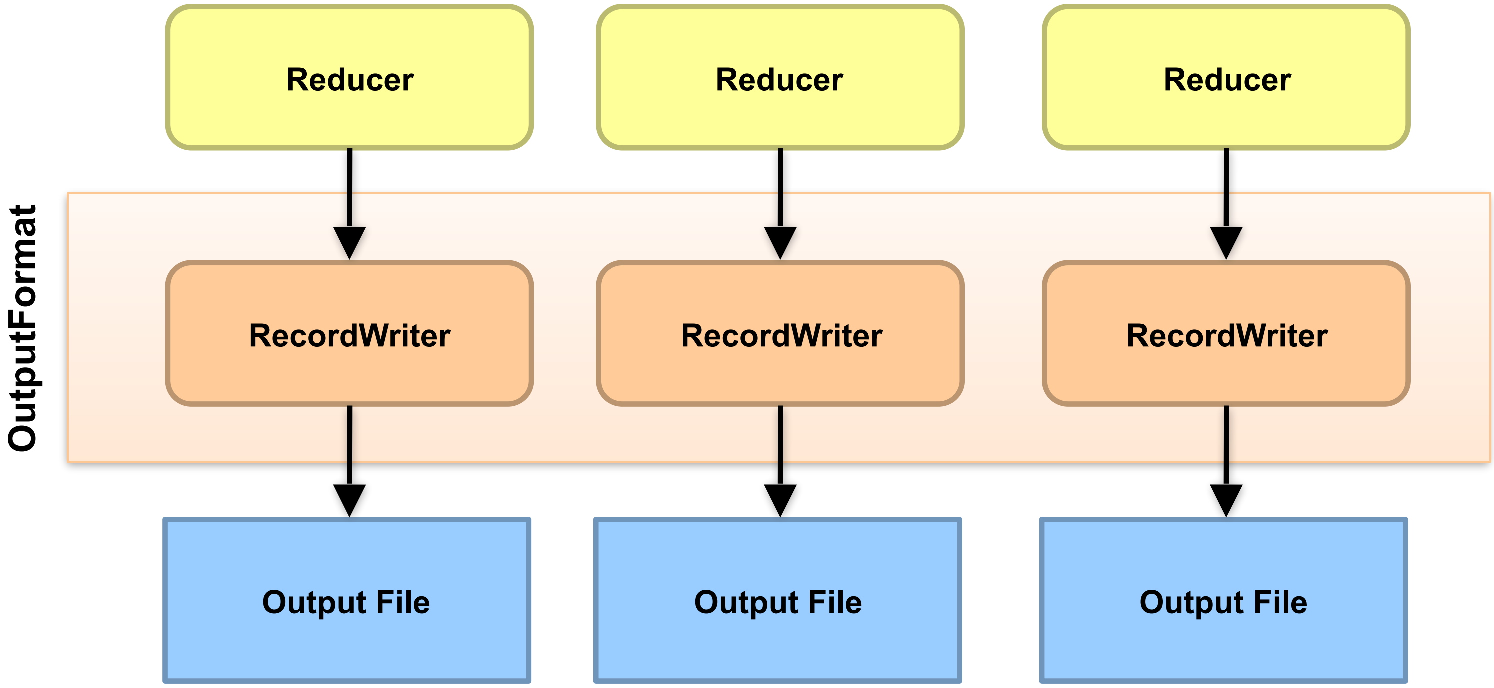
- Limit as much as possible the memory footprint
  - **Avoid** storing reducer values in **local lists** if possible
  - Use **static final** objects
  - Reuse **Writable** objects
- A single reducer is a powerful friend
  - Object fields are **shared** among reduce() invocations.
  - The framework **reuses** value object in reducer, so make deep copies if needed
- Passing parameters via class statics doesn't work!
  - Use configuration parameters (through Job configuration)
  - Use external data sources/sinks (files on HDFS, cache service)

Hadoop Dataflow (I)



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

Hadoop Dataflow (II)



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

HADOOP Data Reading (1.x or 0.20.x)

- Data sets are specified by **InputFormats**
  - Defines input data (e.g., a directory)
  - Identifies partitions of the data that form an **InputSplit**, each of which will be assigned to a mapper
  - Provide the **RecordReader** implementation to extract (k, v) records from the input source
- Base class implementation is **FileInputFormat**
  - Will read all files out of a specified directory and send them to the mappers
  - **TextInputFormat** – Treats each '\n'-terminated line of a file as a value
  - **KeyValueTextInputFormat** – Maps '\n'- terminated text lines of "k SEP v"
  - **SequenceFileInputFormat** – Binary file of (k, v) pairs with some add'l metadata
  - **SequenceFileAsTextInputFormat** – Same, but maps (k.toString(), v.toString())

HADOOP Data Writing (1.x or 0.20.x)

- Data sets are specified by **OutputFormats**
  - Analogous to InputFormat
- Base class implementation is **FileOutputFormat**
  - TextOutputFormat – Writes “key val\n” strings to output file
  - SequenceFileOutputFormat – Uses a binary format to pack (k, v) pairs
- Other implementation is **NullOutputFormat**
  - Discards output to /dev/null