Hadoop

Hadoop Installation

Generally Hadoop can be run in three modes:

1. **Standalone (or local) mode** (TEST AND DEBUG)
   - There are no daemons used in this mode. PROGRAMMI ESEGUITI ON-DEMAND
   - Hadoop uses the local file system as a substitute for HDFS file system.
   - The jobs will run as if there is 1 mapper and 1 reducer
2. **Pseudo-distributed mode**
   - All the daemons run on a single machine
   - This setting mimics the behaviour of a cluster
   - All the daemons run on your machine locally using the HDFS protocol.
   - There can be multiple mappers and reducers (1 reducer by default)
3. **Fully-distributed mode** (2 - 1000 MACHINES)
   - This is how Hadoop runs on a real cluster
   - All the daemons run on (a subset of) the cluster's machines using the HDFS protocol
   - There are multiple mappers and reducers (1 reducer by default)

Hadoop Program Execution

- Typically, Hadoop programs are developed with a text editors or an IDE
- The actual debugging and execution must be performed on Hadoop without IDE
- Once compiled, an Hadoop program is packaged in a JAR file and submitted to a Hadoop cluster
- On a client, we need to specify where HDFS and Hadoop daemons are running
- For simplicity, we will:
    - Develop programs locally using Maven to manage dependency
    - Package the Hadoop program in a JAR file
    - Copy the file into the Hadoop cluster using SSH
    - Perform debugging and execution on the Hadoop cluster using the Web browser interfaces and the log files
- Be prepared...

Word Count in Hadoop (I)

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    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);
        }
    }
}
```

Word Count in Hadoop (II)

```
public static class IntSumReducer 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);
    }
}
```

Word Count in Hadoop (I)

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2)
      System.err.println("Usage: wordcount <in> [<in>...] <out>"); System.exit(2);
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i)
        FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Hadoop Terminology

- A MapReduce **job** is a unit of work that the client wants to be performed. It consists of
  - **input** data
  - MapReduce **program**
  - **configuration** information
- Hadoop runs the MapReduce job by dividing it into **tasks**
  - There are two types of tasks: **map tasks** and **reduce tasks**
  - The tasks are **scheduled** using **YARN** (Yet Another Resource Negotiation) and run on nodes in the cluster.
  - If a task fails, it will be automatically rescheduled to run on a different node.
- Hadoop divides the input to a MapReduce job into **fixed-size pieces** called **input splits**
- Hadoop creates **one map task for each split**, which runs the user-defined map function for each **record** in the split

Hadoop Types (I)

Programmers specify **two functions**

- *map function*: from [key (K1), value (V1)] pair to list of [key (K2), value (V2)] pairs
- *reduce function*: from [key (K2), list of values (V2)] pair to list of [key (K3), values (V3)] pairs

```
public class Mapper<K1, V1, K2, V2> {
    public class Context extends MapContext<K1, V1, K2, V2> {
        // ...
    }
    protected void map(K1 key, V1 value, Context context)
                                     throws IOException, InterruptedException {
        // ...
    }
}

public class Reducer<K2, V2, K3, V3> {
    public class Context extends ReducerContext<K2, V2, K3, V3> {
        // ...
    }
    protected void reduce(K2 key, Iterable<V2> values, Context context)
                                     throws IOException, InterruptedException {
        // ...
    }
}
```

Hadoop Types (II)

- *map function*: from [key (K1), value (V1)] pair to list of [key (K2), value (V2)] pairs
- ***combiner function*: from [key (K2), list of values (V2)] pair to list of [key (K2), values (V2)] pairs**
- *reduce function*: from [key (K2), list of values (V2)] pair to list of [key (K3), values (V3)] pairs
- ***partitioner function*: from [key (K2), value (V2)] pair to integer**

- If a combiner function is used, then it has the same form as the reduce function
- The combiner function is an implementation of Reducer), except its output types are the intermediate key and value types (K2 and V2)
- Often the combiner and reduce functions are the same, in which case K3 is the same as K2, and V3 is the same as V2
- The partition function operates on the intermediate key and value types (K2 and V2) and returns the partition index
- In practice, the partition is determined solely by the key (the value is ignored)

```
public abstract class Partitioner<K2, V2>
{
    public abstract int getPartition(K2 key, V2 value, int numPartitions);
}
```

Hadoop Job Configuration (I)

Property	Job setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
`mapreduce.job.inputformat.class`	`setInputFormatClass()`	•	•				
`mapreduce.map.output.key.class`	`setMapOutputKeyClass()`			•			
`mapreduce.map.output.value.class`	`setMapOutputValueClass()`				•		
`mapreduce.job.output.key.class`	`setOutputKeyClass()`					•	
`mapreduce.job.output.value.class`	`setOutputValueClass()`						•
Properties that must be consistent with the types:							
`mapreduce.job.map.class`	`setMapperClass()`	•	•	•	•		
`mapreduce.job.combine.class`	`setCombinerClass()`			•	•		
`mapreduce.job.partitioner.class`	`setPartitionerClass()`			•	•		
`mapreduce.job.output.key.comparator.class`	`setSortComparatorClass()`			•			
`mapreduce.job.output.group.comparator.class`	`setGroupingComparatorClass()`			•			
`mapreduce.job.reduce.class`	`setReducerClass()`			•	•	•	•
`mapreduce.job.outputformat.class`	`setOutputFormatClass()`					•	•

Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

Hadoop Job Configuration (II)

- Java generics' **type erasure** means that the type information isn't always present at runtime, so Hadoop has to be given it explicitly
- Input types are set by the input format.
  - For instance, a **`TextInputFormat`** generates keys of type **`LongWritable`** and values of type **`Text`**
- The other types are set explicitly by calling the methods on the Job
- If not set explicitly, the intermediate types default to the (final) output types, which default to **`LongWritable`** and **`Text`**.
  - For instance, if K2 and K3 are the same, you don't need to call **`setMapOutputKey Class()`**
  - Similarly, if V2 and V3 are the same, you only need to use **`setOutputValueClass()`**
- Due to type erasure it's possible to configure an Hadoop job with incompatible types, because the configuration isn't checked at compile time.

Hadoop Configuration Defaults

- The only configuration that we set is an **input path** and an output path.
- The **default input format** is `TextInputFormat`
  - The output key is of type `LongWritable`, and the output value is of type `Text`
- The **default mapper** is the `Mapper` class, which writes the input key and value unchanged to the output
  - The output key is of type `LongWritable`, and the output value is of type `Text`
- The **default reducer** is the `Reducer` class, which simply writes all its input to its output
  - The output key is of type `LongWritable`, and the output value is of type `Text`
- The **default partitioner** is `HashPartitioner`, which hashes an intermediate key to determine which partition the key belongs in
  - Each partition is processed by a reduce task
  - So the **number of partitions** is equal to the **number of reduce tasks** for the job
  - **By default there is a single reducer**, and therefore a single partition
- We did not set the **number of map tasks**
  - The number is equal to the **number of splits** that the input is turned into
  - It depends on the size of the input and the file's block size (if the file is in HDFS)

Serialization in Hadoop

- **Serialization** is the process of turning **structured objects** into a **byte stream**
  - for transmission over a network or for writing to persistent storage
- **Deserialization** is the reverse process of turning a **byte stream** back into a series of **structured objects**.
- Serialization is used in two quite distinct areas of distributed data processing
  - for interprocess communication
  - for persistent storage
- In Hadoop, **interprocess communication** between nodes in the system is implemented using **remote procedure calls** (RPCs)
- Hadoop uses its own serialization format called **Writables**
  - compact and fast
  - not so easy to extend or use from languages other than Java
  - There are other serialization frameworks supported in Hadoop, such as **Avro, Thrift, Protobuffers**

Writable Inferfaces

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable
{
    void write(DataOutput out) throws IOException;

    void readFields(DataInput in) throws IOException;
}
```

```
package org.apache.hadoop.io;

public interface WritableComparable<T> extends Writable, Comparable<T>
{
}
```

IntWritable Example (I)

```java
package org.apache.hadoop.io;

import java.io.*;

public class IntWritable implements WritableComparable {

  private int value;

  public IntWritable() {}
  public IntWritable(int value) { set(value); }

  public void set(int value) {
    this.value = value;
  }

  public int get() {
    return value;
  }

  public void readFields(DataInput in) throws IOException {
    value = in.readInt();
  }

  public void write(DataOutput out) throws IOException {
    out.writeInt(value);
  }
```

IntWritable Example (II)

```
public boolean equals(Object o) {
  if (!(o instanceof IntWritable))
    return false;

  IntWritable other = (IntWritable)o;
  return this.value == other.value;
}

public int hashCode() {
  return value;
}

public int compareTo(Object o) {
  int thisValue = this.value;
  int thatValue = ((IntWritable)o).value;
  return (thisValue<thatValue ? -1 : (thisValue==thatValue ? 0 : 1));
}

public String toString() {
  return Integer.toString(value);
}
```

IntWritable Example (III)

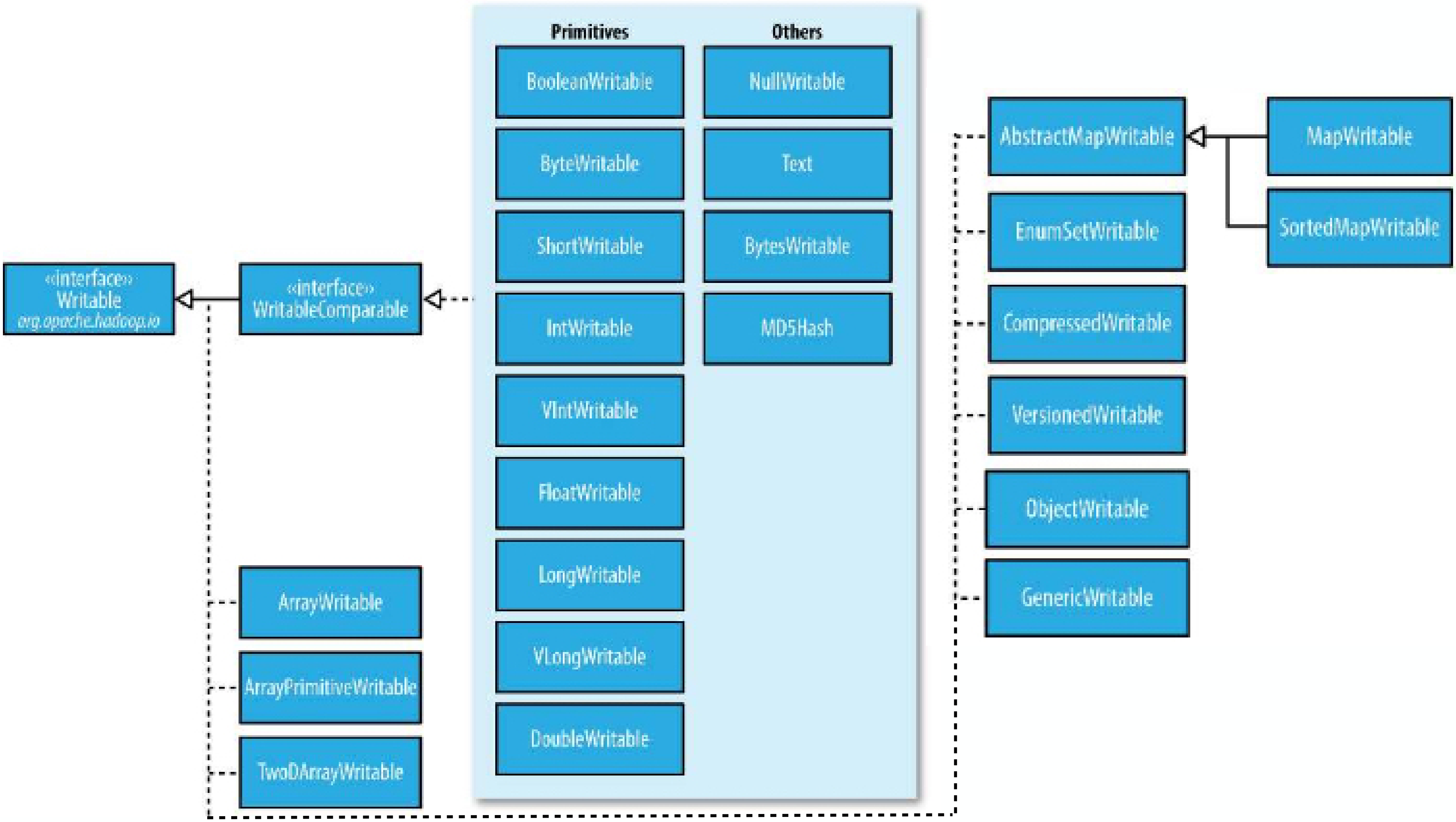
```
  /** A Comparator optimized for IntWritable. */

  public static class Comparator extends WritableComparator {

    public Comparator() {
      super(IntWritable.class);
    }

    public int compare(byte[] b1, int s1, int l1,
                       byte[] b2, int s2, int l2) {

      int thisValue = readInt(b1, s1);
      int thatValue = readInt(b2, s2);

      return (thisValue < thatValue ? -1 : (thisValue == thatValue ? 0 : 1));
    }
  }
}
```

Writable Wrappers

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

Writable Class Hierarchy



Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

Hadoop Input (I)

- An **input split** is a chunk of the input that is processed by a single map task
- Each split is divided into **records**, and the map task processes each record – a key-value pair – in turn
- Splits and records are **logical**
  - Not required to be files, although commonly they are.
  - In a database context, a split might correspond to a range of rows from a table and a record to a row in that range
  - Input splits are represented by the Java class `InputSplit`
- An `InputSplit` has a **length** in bytes and a **set of storage locations** (i.e., hostname strings)
  - A split doesn't contain the input data;
  - A split is just a reference to the data.
  - The **storage locations** are used by Hadoop to **place map tasks** as close to the split's data as possible,
  - The **size** is used to order the splits so that the largest get processed first, to **minimize the job runtime**

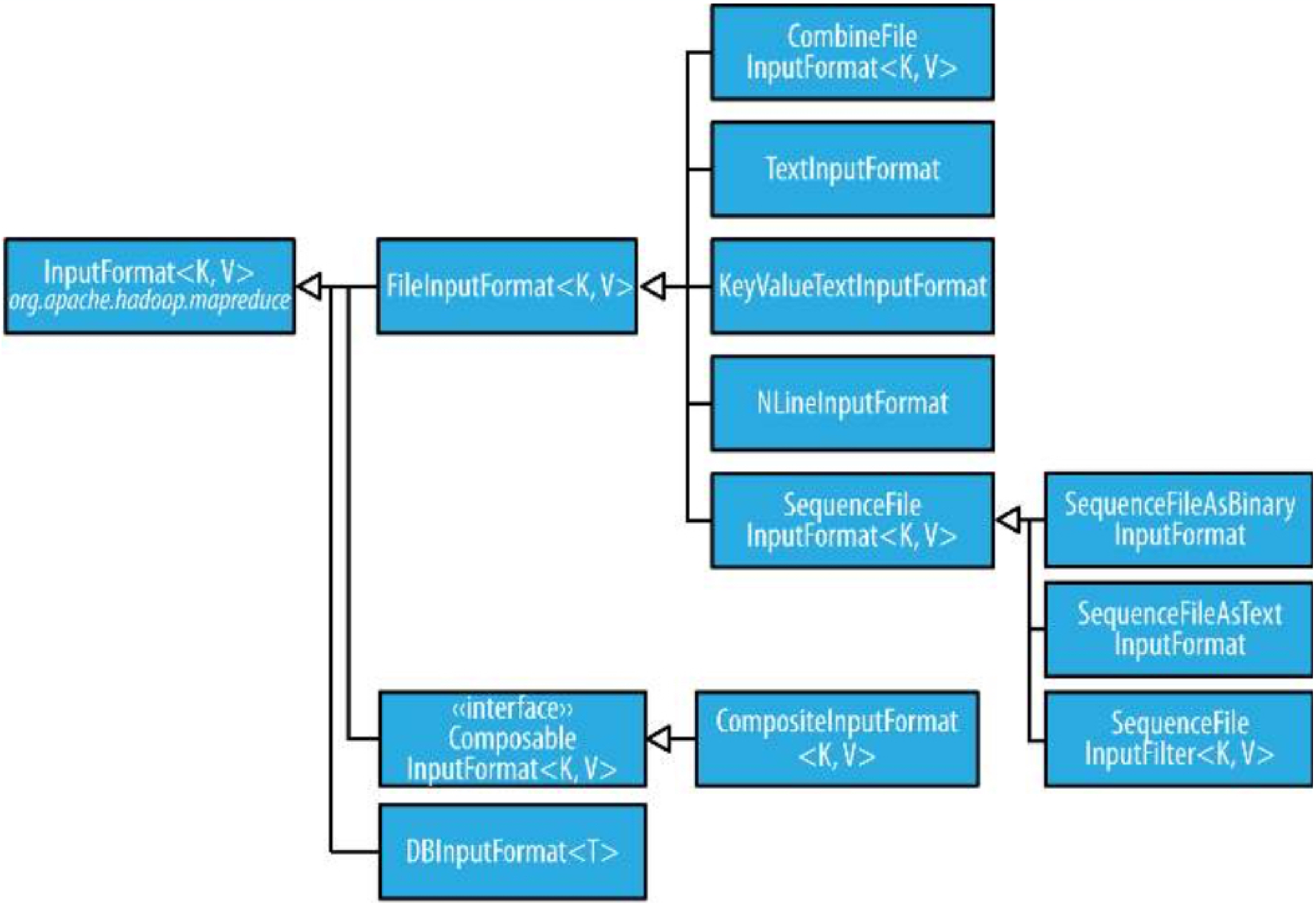
Hadoop Input (II)

- **`InputSplits`** are created by an **`InputFormat`** interface implementation

```
public abstract class InputFormat<K, V>
{
  public abstract List<InputSplit> getSplits(JobContext context)
    throws IOException, InterruptedException;
  public abstract RecordReader<K, V> createRecordReader(InputSplit split, TaskAttemptContext context)
    throws IOException, InterruptedException;
}
```

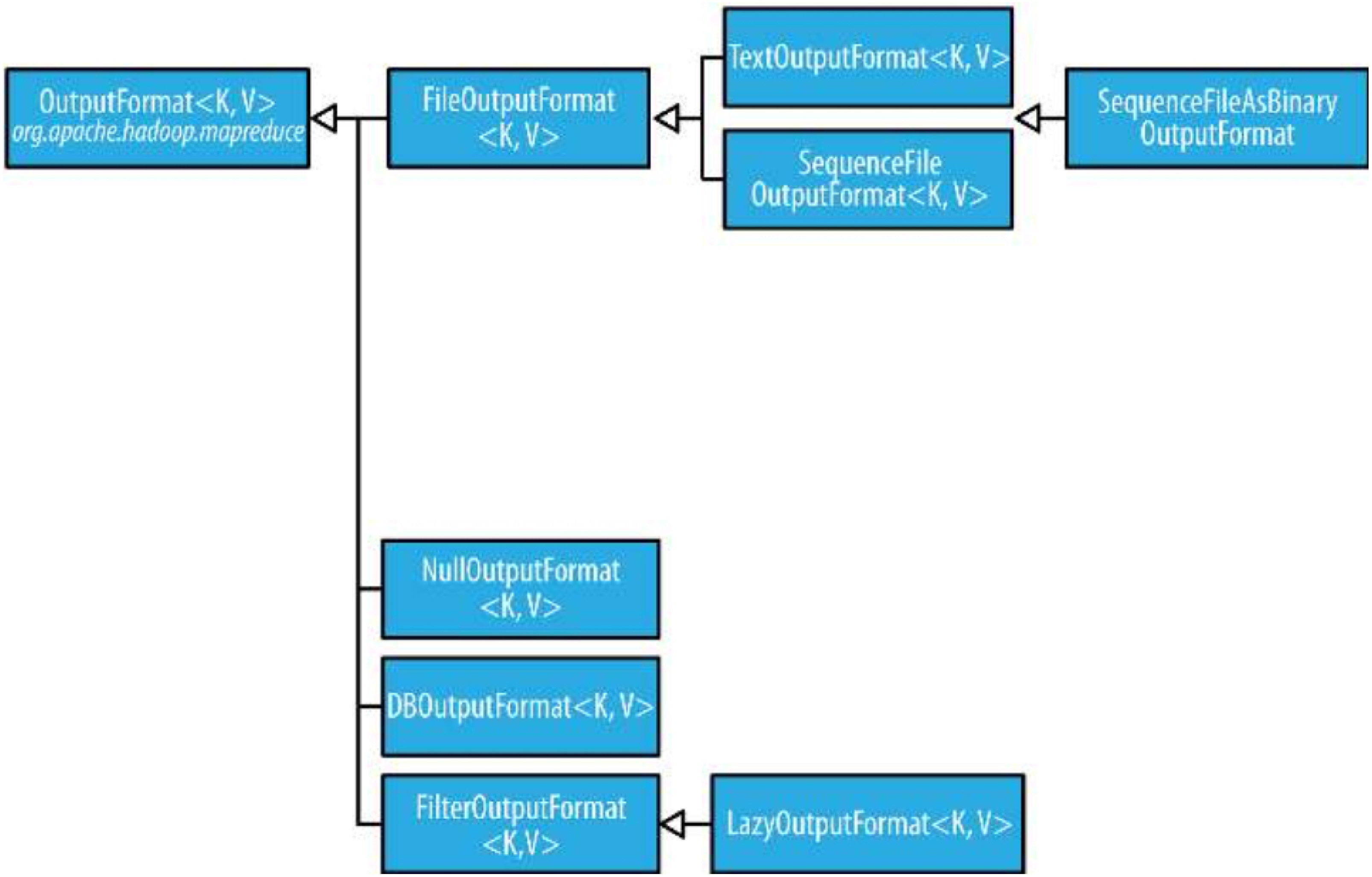
- The client running the job calculates the splits for the job by calling **`getSplits()`**
- It sends them to the application master, which uses their storage locations to schedule map tasks that will process them on the cluster
- The map task passes the split to the **`createRecordReader()`** method on InputFormat to obtain a **`RecordReader`** for that split.
- A **`RecordReader`** is little more than an iterator over records, and the map task uses one to generate record key-value pairs, which it passes to the map function.

InputFormat Class Hierarchy



Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

OutputFormat Class Hierarchy



Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

The setup and cleanup methods

- It is common to want your `Mapper` or `Reducer` to execute some code before the `map()` or `reduce()` method is called for the first time
  - Initialize data structures
  - Read data from an external file
  - Set parameters
- The `setup()` method is run before the `map()` or `reduce()` method is called for the first time

```
public void setup(Context context)
```

- Similarly, you may wish to perform some action(s) after all the records have been processed by your `Mapper` or `Reducer`
- The `cleanup()` method is called before the `Mapper` or `Reducer` terminates

```
public void cleanup(Context context)
```

Passing parameters

```
public class MyDriverClass
{
    public int main(String[] args) throws Exception
    {
        int value = 42;
        Configuration conf = new Configuration();
        conf.setInt ("paramname", value);
        Job job = new Job(conf);
        // ...
        return job.waitForCompletion(true);
    }
}
```

```
public class MyMapper extends Mapper
{
    public void setup(Context context)
    {
        Configuration conf = context.getConfiguration();
        int myParam = conf.getInt("paramname", 0);
        // ...
    }

    public void map...
}
```

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)
- **An `Iterable<V>` object is not equal to a `List<V>` object**
  - You don't know the number of values **a priori**
  - Accessing an iterable object return a **reference** to the current object
  - Iterable's current object **changes state**, perform a **deep copy** if you need to save it
  - Be careful in allocating a list to store the iterable's elements (memory!)