Apache Hadoop – A course for undergraduates

Lecture 6

Partitioners and Reducers

Chapter 6.1

Partitioners and Reducers

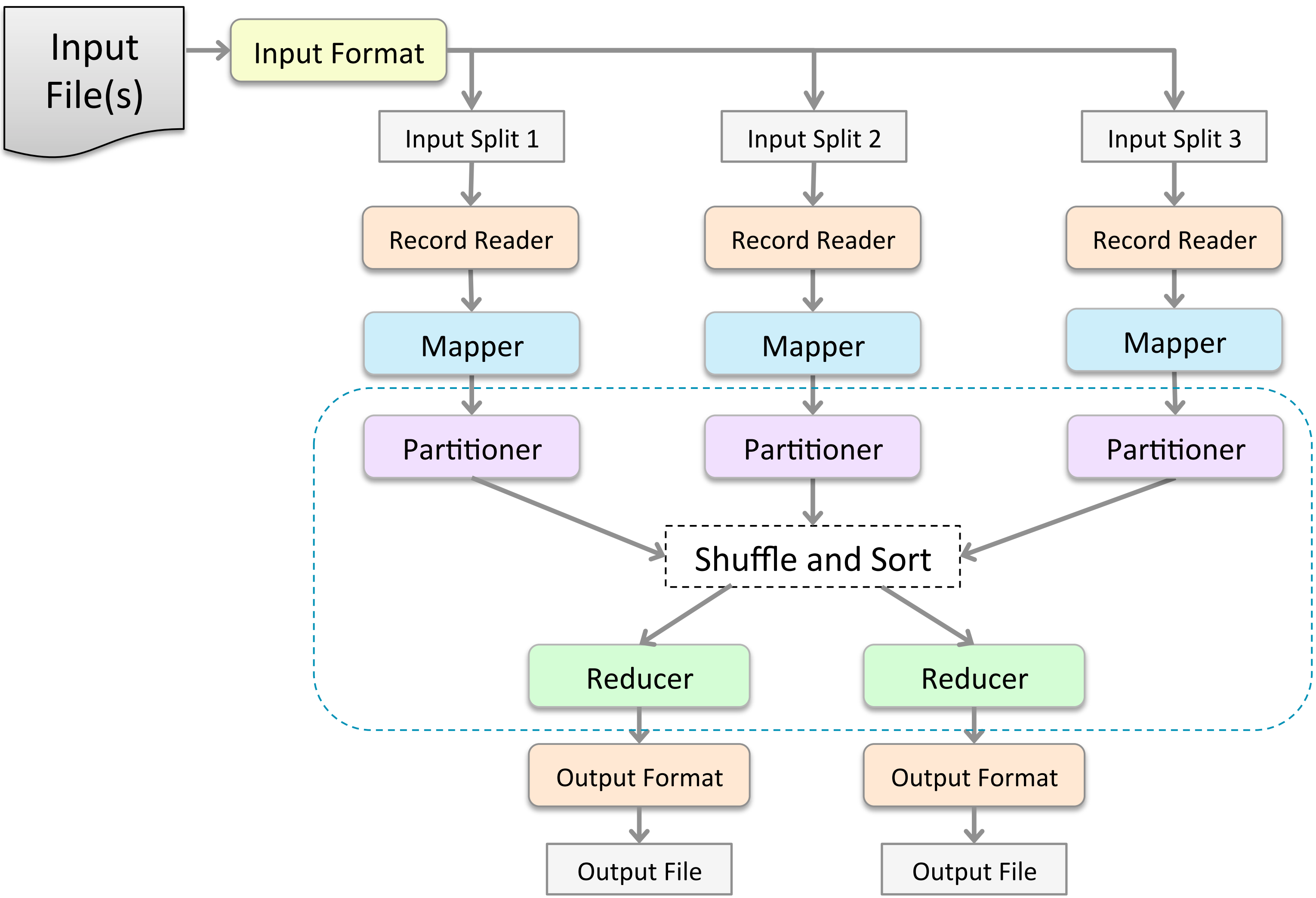
- **Writing custom Partitioners**
- **Determining how many Reducers are needed**

Chapter Topics

Partitioners and Reducers

- **How Partitioners and Reducers Work Together**
- Determining the Optimal Number of Reducers for a Job
- Writing Custom Partitioners

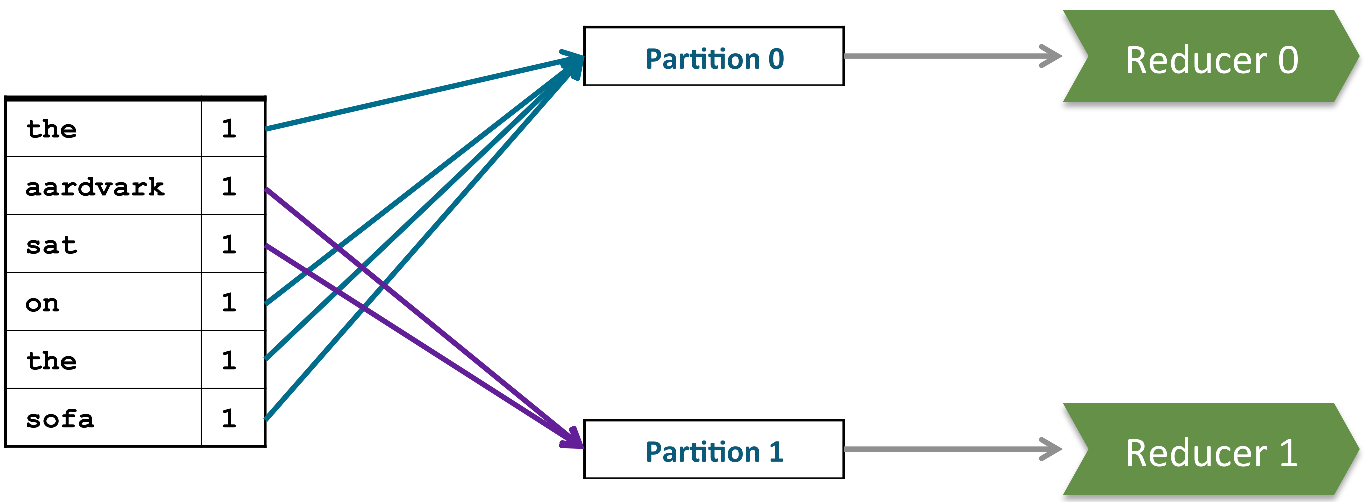
Review: The Big Picture



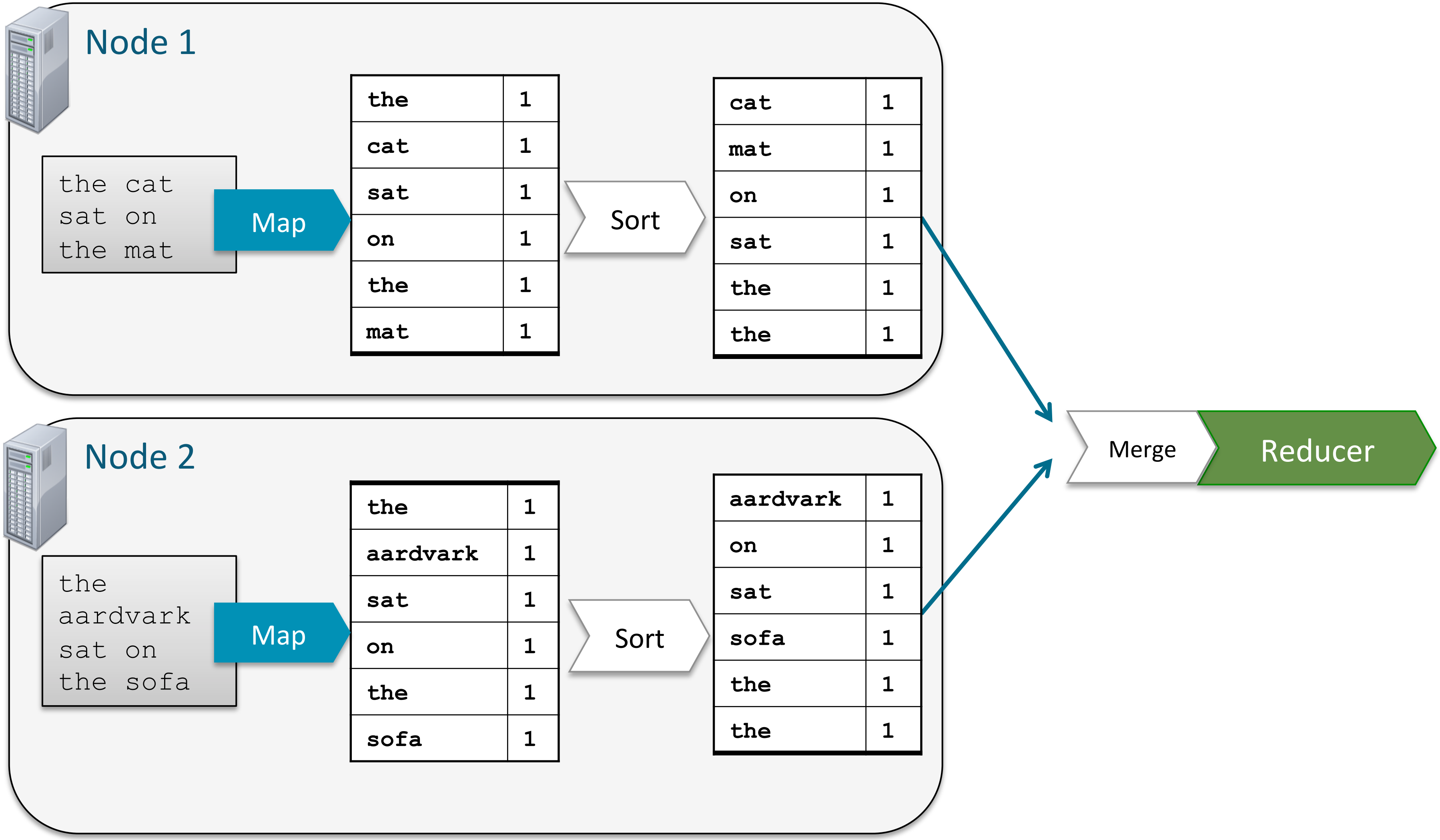
What Does the Partitioner Do?

- **The Partitioner determines which Reducer each intermediate key and its associated values goes to**

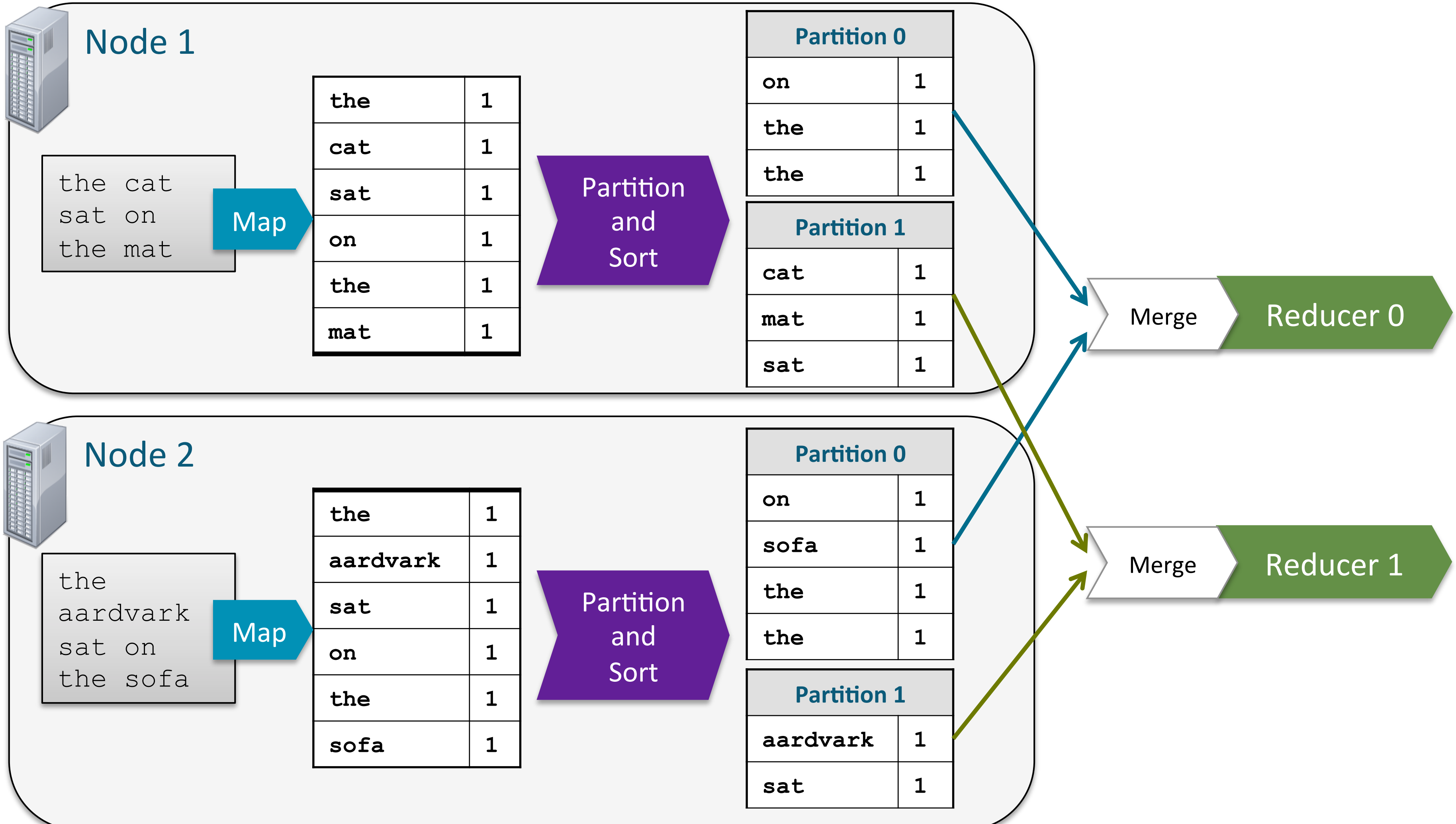
```
getPartion:
    (inter_key, inter_value, num_reducers)→ partition
```



Example: WordCount with One Reducer



Example: WordCount with Two Reducers



The Default Partitioner

- **The default Partitioner is the `HashPartitioner`**
  - Uses the Java `hashCode` method
  - Guarantees all pairs with the same key go to the same Reducer

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {

    public int getPartition(K key, V value, int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

Chapter Topics

Partitioners and Reducers

- How Partitioners and Reducers Work Together
- **Determining the Optimal Number of Reducers for a Job**
- Writing Custom Partitioners

How Many Reducers Do You Need?

- **An important consideration when creating your job is to determine the number of Reducers specified**
- **Default is a single Reducer**
- **With a single Reducer, one task receives *all* keys in sorted order**
  - This is sometimes advantageous if the output must be in completely sorted order
  - Can cause significant problems if there is a large amount of intermediate data
    - Node on which the Reducer is running may not have enough disk space to hold all intermediate data
    - The Reducer will take a long time to run

Jobs Which Require a Single Reducer

- **If a job needs to output a file where all keys are listed in sorted order, a single Reducer must be used**
- **Alternatively, the `TotalOrderPartitioner` can be used**
  - Uses an externally generated file which contains information about intermediate key distribution
  - Partitions data such that all keys which go to the first Reducer are smaller than any which go to the second, etc
  - In this way, multiple Reducers can be used
  - Concatenating the Reducers' output files results in a totally ordered list

Jobs Which Require a Fixed Number of Reducers

- **Some jobs will require a specific number of Reducers**
- **Example: a job must output one file per day of the week**
  - Key will be the weekday
  - Seven Reducers will be specified
  - A Partitioner will be written which sends one key to each Reducer

Jobs With a Variable Number of Reducers (1)

- **Many jobs can be run with a variable number of Reducers**
- **Developer must decide how many to specify**
  - Each Reducer should get a reasonable amount of intermediate data, but not too much
  - Chicken-and-egg problem
- **Typical way to determine how many Reducers to specify:**
  - Test the job with a relatively small test data set
  - Extrapolate to calculate the amount of intermediate data expected from the 'real' input data
  - Use that to calculate the number of Reducers which should be specified

Jobs With a Variable Number of Reducers (2)

- **Note: you should take into account the number of Reduce slots likely to be available on the cluster**
  - If your job requires one more Reduce slot than there are available, a second 'wave' of Reducers will run
    - Consisting just of that single Reducer
    - Potentially doubling the amount of time spent on the Reduce phase
  - In this case, increasing the number of Reducers further may cut down the time spent in the Reduce phase
    - Two or more waves will run, but the Reducers in each wave will have to process less data

Chapter Topics

Partitioners and Reducers

- How Partitioners and Reducers Work Together
- Determining the Optimal Number of Reducers for a Job
- **Writing Custom Partitioners**

Custom Partitioners (1)

- **Sometimes you will need to write your own Partitioner**
- **Example: your key is a custom `WritableComparable` which contains a pair of values `(a, b)`**
  - You may decide that all keys with the same value for **`a`** need to go to the same Reducer
  - The default Partitioner is not sufficient in this case

Custom Partitioners (2)

- **Custom Partitioners are needed when performing a secondary sort (see later)**
- **Custom Partitioners are also useful to avoid potential performance issues**
  - To avoid one Reducer having to deal with many very large lists of values
  - Example: in our word count job, we wouldn't want a single Reducer dealing with all the three- and four-letter words, while another only had to handle 10- and 11-letter words

Creating a Custom Partitioner

1. **Create a class that extends Partitioner**
2. **Override the `getPartition` method**
   - Return an int between 0 and one less than the number of Reducers
     - e.g., if there are 10 Reducers, return an int between 0 and 9

```
import org.apache.hadoop.mapreduce.Partitioner;

public class MyPartitioner<K,V> extends Partitioner<K,V> {

    @Override
    public int getPartition(K key, V value, int numReduceTasks) {
        //determine reducer number between 0 and numReduceTasks-1
        //...
        return reducer;
    }
}
```

Using a Custom Partitioner

- **Specify the custom Partitioner in your driver code**

```
job.setPartitionerClass(MyPartitioner.class);
```

Aside: Setting up Variables for your Partitioner (1)

- **If you need to set up variables for use in your partitioner, it should implement `Configurable`**
- **If a Hadoop object implements `Configurable`, its `setConf()` method will be called once, when it is instantiated**
- **You can therefore set up variables in the `setConf()` method which your `getPartition()` method will then be able to access**

Aside: Setting up Variables for your Partitioner (2)

```
class MyPartitioner extends Partitioner<K, V> implements Configurable {

    private Configuration configuration;
    // Define your own variables here

    @Override
    public void setConf(Configuration configuration) {
        this.configuration = configuration;
        // Set up your variables here
    }
    @Override
    public Configuration getConf() {
        return configuration;
    }
    public int getPartition(K key, V value, int numReduceTasks) {
        // Use variables here
    }
}
```

Key Points

- **Developers need to consider how many Reducers are required for a job**
- **Partitioners divide up intermediate data to pass to Reducers**
- **Write custom Partitioners for better load balancing**
  - `getPartition` method returns an integer indicating which Reducer to send the data to

Bibliography

The following offer more information on topics discussed in this chapter

- **Use of the TotalOrderPartitioner is described in detail on pages 274-277 of TDG 3e.**
- **See TDG 3e page 254 for more discussion on considerations when designing a Partitioner.**

Data Input and Output

Chapter 6.2

Data Input and Output

- **How to create custom Writable and WritableComparable implementations**
- **How to save binary data using SequenceFile and Avro data files**
- **What issues to consider when using file compression**

Chapter Topics

Data Input and Output

- **Creating Custom Writable and WritableComparable Implementations**
- Saving Binary Data Using SequenceFiles and Avro Data Files
- Issues to Consider When Using File Compression

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 `WritableComparable`

`IntWritable`
`LongWritable`
`Text`
…

Concrete classes for different data types

'Box' Classes in Hadoop

- **Hadoop's built-in data types are 'box' classes**
  - They contain a single piece of data
    - `Text`: `String`
    - `IntWritable`: `int`
    - `LongWritable`: `long`
    - `FloatWritable`: `float`
    - etc.
- **`Writable` defines the wire transfer format**
  - How the data is serialized and deserialized

Creating a Complex `Writable`

- **Example: say we want a tuple (a, b) as the value emitted from a Mapper**
  - We could artificially construct it by, for example, saying

```
Text t = new Text(a + "," + b);
...
String[] arr = t.toString().split(",");
```

- **Inelegant**
- **Problematic**
  - If `a` or `b` contained commas, for example
- **Not always practical**
  - Doesn't easily work for binary objects
- **Solution: create your own `Writable` object**

The `Writable` Interface

```
public interface Writable {
    void readFields(DataInput in);
    void write(DataOutput out);
}
```

- **The `readFields` and `write` methods will define how your custom object will be serialized and deserialized by Hadoop**
- **The `DataInput` and `DataOutput` classes support**
  - `boolean`
  - `byte`, `char` (Unicode: 2 bytes)
  - `double`, `float`, `int`, `long`,
  - `String` (Unicode or UTF-8)
  - Line until line terminator
  - unsigned `byte`, `short`
  - `byte` array

A Sample Custom `Writable`: DateWritable

```java
class DateWritable implements Writable {
  int month, day, year;

  // Constructors omitted for brevity

  public void readFields(DataInput in) throws IOException {
    this.month = in.readInt();
    this.day = in.readInt();
    this.year = in.readInt();
  }

  public void write(DataOutput out) throws IOException {
    out.writeInt(this.month);
    out.writeInt(this.day);
    out.writeInt(this.year);
  }
}
```

What About Binary Objects?

- **Solution: use byte arrays**
- **Write idiom:**
  - Serialize object to byte array
  - Write byte count
  - Write byte array
- **Read idiom:**
  - Read byte count
  - Create byte array of proper size
  - Read byte array
  - Deserialize object

`WritableComparable`

- **`WritableComparable` is a sub-interface of `Writable`**
  - Must implement `compareTo`, `hashCode`, `equals` methods
- **All keys in MapReduce must be `WritableComparable`**

Making DateWritable a WritableComparable (1)

```
class DateWritable implements WritableComparable<DateWritable> {
  int month, day, year;

  // Constructors omitted for brevity

  public void readFields (DataInput in) . . .

  public void write (DataOutput out) . . .

  public boolean equals(Object o) {
    if (o instanceof DateWritable) {
      DateWritable other = (DateWritable) o;
      return this.year == other.year && this.month == other.month
      && this.day == other.day;
    }
    return false;
  }
```

Making `DateWritable` a `WritableComparable` (2)

```
    public int compareTo(DateWritable other) {
                              // Return -1 if this date is earlier
                              // Return 0 if dates are equal
                              // Return 1 if this date is later

      if (this.year != other.year) {
        return (this.year < other.year ? -1 : 1);
      } else if (this.month != other.month) {
        return (this.month < other.month ? -1 : 1);
      } else if (this.day != other.day) {
        return (this.day < other.day ? -1 : 1);
      }
      return 0;
    }

    public int hashCode() {
      int seed = 163;                  // Arbitrary seed value
      return this.year * seed + this.month * seed + this.day * seed;
    }
  }
```

Using Custom Types in MapReduce Jobs

- **Use methods in `Job` to specify your custom key/value types**
- **For output of Mappers:**

```
job.setMapOutputKeyClass()
job.setMapOutputValueClass()
```

- **For output of Reducers:**

```
job.setOutputKeyClass()
job.setOutputValueClass()
```

- **Input types are defined by `InputFormat`**
  - Covered later

Chapter Topics

Data Input and Output

- Creating Custom Writable and WritableComparable Implementations
- **Saving Binary Data Using SequenceFiles and Avro Data Files**
- Issues to Consider When Using File Compression

What Are SequenceFiles?

- **SequenceFiles are files containing binary-encoded key-value pairs**
  - Work naturally with Hadoop data types
  - SequenceFiles include metadata which identifies the data types of the key and value
- **Actually, three file types in one**
  - Uncompressed
  - Record-compressed
  - Block-compressed
- **Often used in MapReduce**
  - Especially when the output of one job will be used as the input for another
    - `SequenceFileInputFormat`
    - `SequenceFileOutputFormat`

Directly Accessing SequenceFiles

- **It is possible to directly access `SequenceFiles` from your code:**

```
Configuration config = new Configuration();
SequenceFile.Reader reader =
    new SequenceFile.Reader(FileSystem.get(config), path, config);

Text key = (Text) reader.getKeyClass().newInstance();
IntWritable value = (IntWritable) reader.getValueClass().newInstance();

while (reader.next(key, value)) {
    // do something here
}
reader.close();
```

Problems With SequenceFiles

- **SequenceFiles are useful but have some potential problems**
- **They are only typically accessible via the Java API**
  - Some work has been done to allow access from other languages
- **If the definition of the key or value object changes, the file becomes unreadable**

An Alternative to SequenceFiles: Avro

- **Apache Avro is a serialization format which is becoming a popular alternative to SequenceFiles**
  - Project was created by Doug Cutting, the creator of Hadoop
- **Self-describing file format**
  - The schema for the data is included in the file itself
- **Compact file format**
- **Portable across multiple languages**
  - Support for C, C++, Java, Python, Ruby and others
- **Compatible with Hadoop**
  - Via the `AvroMapper` and `AvroReducer` classes

Chapter Topics

Data Input and Output

- Creating Custom Writable and WritableComparable Implementations
- Saving Binary Data Using SequenceFiles and Avro Data Files
- **Issues to Consider When Using File Compression**

Hadoop and Compressed Files

- **Hadoop understands a variety of file compression formats**
  - Including GZip
- **If a compressed file is included as one of the files to be processed, Hadoop will automatically decompress it and pass the decompressed contents to the Mapper**
  - There is no need for the developer to worry about decompressing the file
- **However, GZip is not a 'splittable file format'**
  - A GZipped file can only be decompressed by starting at the beginning of the file and continuing on to the end
  - You cannot start decompressing the file part of the way through it

Non-Splittable File Formats and Hadoop

- **If the MapReduce framework receives a non-splittable file (such as a GZipped file) it passes the *entire* file to a single Mapper**
- **This can result in one Mapper running for far longer than the others**
  - It is dealing with an entire file, while the others are dealing with smaller portions of files
  - Speculative execution could occur
    - Although this will provide no benefit
- **Typically it is not a good idea to use GZip to compress MapReduce input files**

Splittable Compression Formats: LZO

- **One splittable compression format is LZO**
- **Because of licensing restrictions, LZO cannot be shipped with Hadoop**
  - But it is easy to add
  - See `https://github.com/cloudera/hadoop-lzo`
- **To make an LZO file splittable, you must first index the file**
- **The index file contains information about how to break the LZO file into splits that can be decompressed**
- **Access the splittable LZO file as follows:**
  - In Java MapReduce programs, use the `LzoTextInputFormat` class
  - In Streaming jobs, specify `-inputformat com.hadoop.mapred.DeprecatedLzoTextInputFormat` on the command line

Splittable Compression for SequenceFiles and Avro Files Using the Snappy Codec

- **Snappy is a relatively new compression codec**
  - Developed at Google
  - Very fast
- **Snappy does not compress a SequenceFile and produce, e.g., a file with a `.snappy` extension**
  - Instead, it is a codec that can be used to compress data within a file
  - That data can be decompressed automatically by Hadoop (or other programs) when the file is read
  - Works well with SequenceFiles, Avro files
- **Snappy is now preferred over LZO**

Compressing Output SequenceFiles With Snappy

- **Specify output compression in the `Job` object**
- **Specify block or record compression**
  - Block compression is recommended for the Snappy codec
- **Set the compression codec to the Snappy codec in the `Job` object**
- **For example:**

```
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;
import org.apache.hadoop.io.SequenceFile.CompressionType;
import org.apache.hadoop.io.compress.SnappyCodec;
. . .
job.setOutputFormatClass(SequenceFileOutputFormat.class);
FileOutputFormat.setCompressOutput(job,true);
FileOutputFormat.setOutputCompressorClass(job,SnappyCodec.class);
SequenceFileOutputFormat.setOutputCompressionType(job,
  CompressionType.BLOCK);
```

Key Points

- **All keys in Hadoop are WritableComparable objects**
  - Writable: `write` and `readFields` methods provide serialization
  - Comparable: `compareTo` method compares two WritableComparable objects
- **Key/Value pairs can be encoded in binary SequenceFile and Avro data files**
  - Useful when one job's output is another job's input
- **Hadoop supports reading from and writing to compressed files**
  - Use "splittable" encoding for MapReduce input files (e.g., Snappy)

Bibliography

The following offer more information on topics discussed in this chapter

- **A thorough discussion of equals and hashCode can be found in Joshua Bloch's Effective Java book**
  - http://www.amazon.com/gp/product/0321356683/
- **SequenceFiles are described in TDG 3e from pages 130-137.**
- **For an example of writing a SequenceFile, see TDG 3e pages 131-132.**
- **Avro**
  - http://www.datasalt.com/2011/07/hadoop-avro/
  - TDG 3e on pages 110-130.
- **Compression is covered on pages 83-92 of TDG 3e.**

Bibliography (cont'd)

The following offer more information on topics discussed in this chapter

- **For more information on Snappy, see**
  - http://www.cloudera.com/blog/2011/09/snappy-and-hadoop/
- **For more information on SequenceFiles and Snappy, see:**
  - http://blog.cloudera.com/blog/2011/09/snappy-and-hadoop/
  - http://wiki.apache.org/hadoop/SequenceFile
  - https://ccp.cloudera.com/display/CDHDOC/Snappy+Installation