

ISIT312 Big Data Management

Java MapReduce Application

Dr Fenghui Ren

School of Computing and Information Technology -
University of Wollongong

Java MapReduce Application

Outline

Building blocks of MapReduce program

Word Count: The "Hello, World" of MapReduce

Hadoop datatype objects

Input and output formats

Java code of Driver

Java code of Mapper

Java code of Reducer

Java code of ToolRunner

Setting up a local running environment

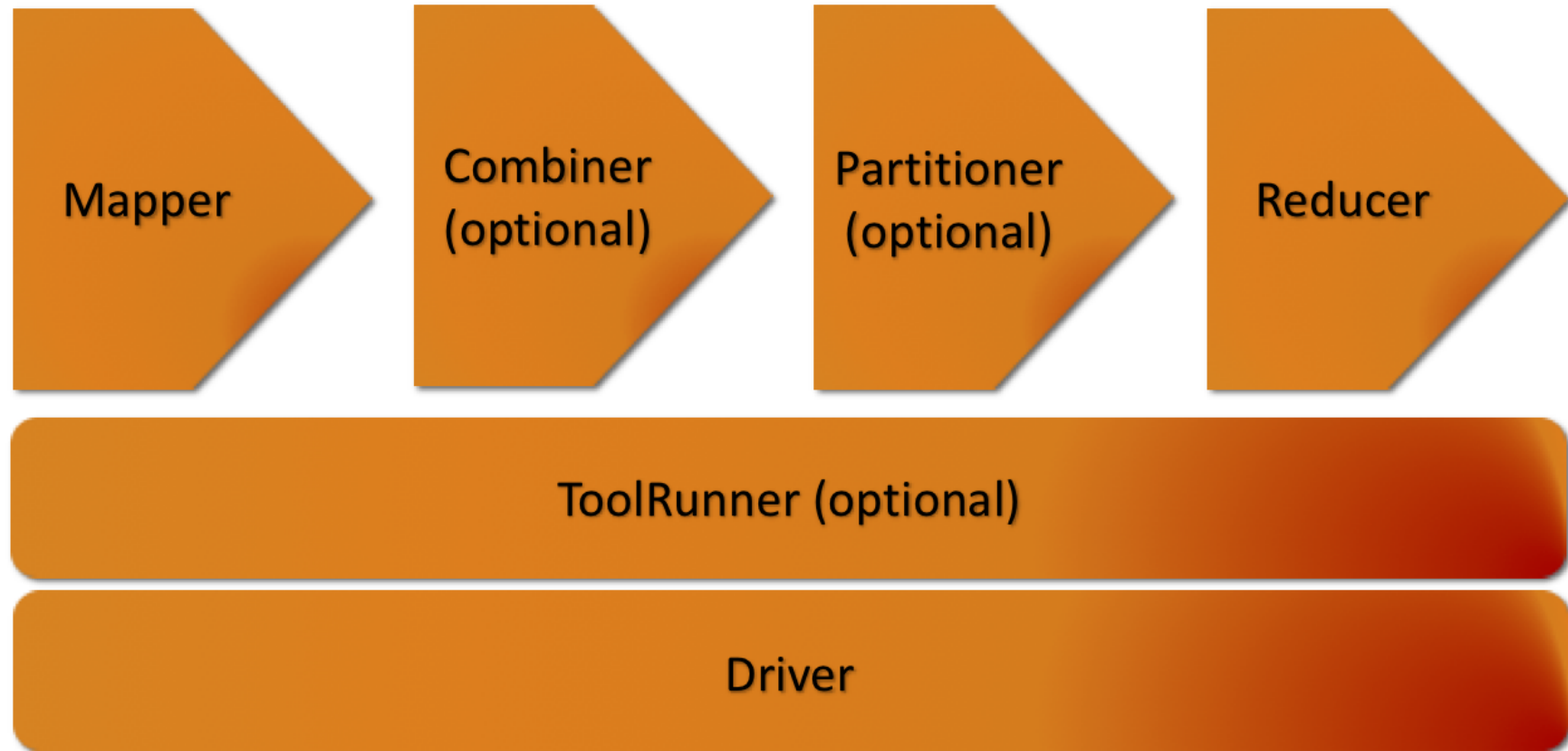
Combiner API

Partitioner API

Miscellaneous

Building blocks of MapReduce program

Structure of MapReduce Java Application



Building blocks of MapReduce program

Mapper, **Reducer**, **Combiner**, and **Partitioner** classes correspond to their counterparts in the **MapReduce** model

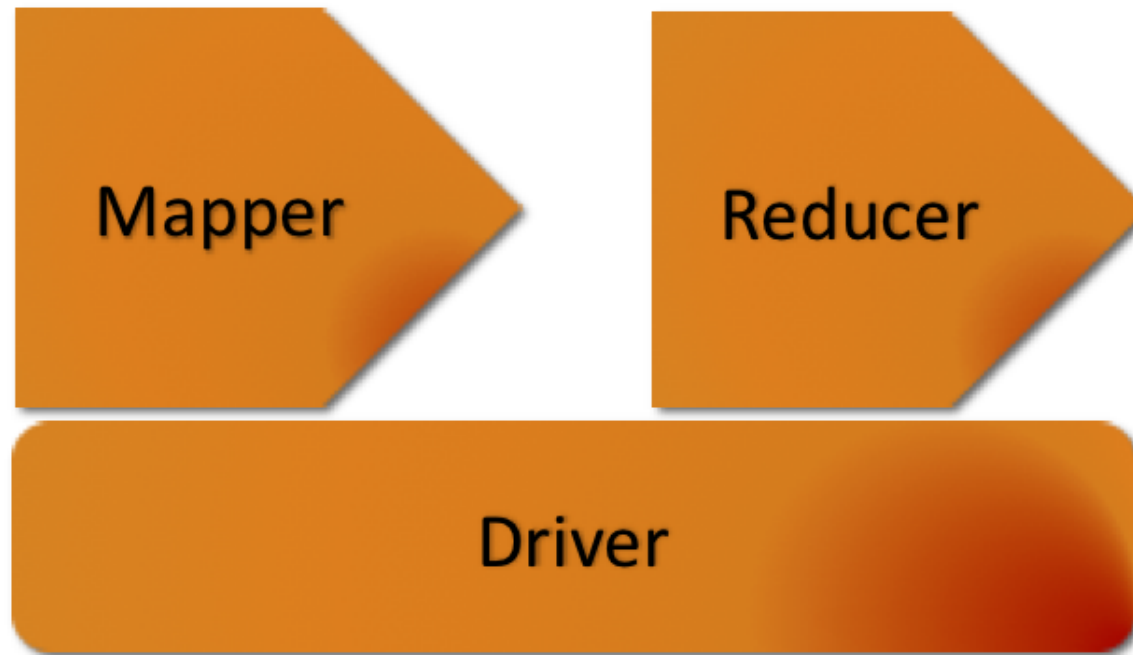
- These classes implement the **MapReduce** logic

The **Driver** or **ToolRunner** in a **MapReduce** program represents the client program

- The main method of a **MapReduce** program is in the **Driver** or **ToolRunner**
- The code of the two is very standard

Building blocks of MapReduce program

An elementary MapReduce program consists only of a Mapper class, a Reducer class and a Driver



As the main method is contained in the Driver, sometimes (but not always) it is convenient to make Mapper and Reducer as inner classes in Driver, which contains routine codes

Building blocks of MapReduce program: Driver

Driver is the program which sets up and starts a **MapReduce** application

Driver code is executed on the client; this code submits the application to the **ResourceManager** along with the application's configuration

Driver can submit the job asynchronously (in a non-blocking fashion) or synchronously (waiting for the application to complete before performing another action)– the second way will be used in our examples

Driver can also configure and submit more than one application; for instance, running a workflow consisting of multiple **MapReduce** applications

Building blocks of MapReduce program: Mapper

Mapper Java class contains a `map()` method

Its object instance iterates through the input to execute a `map()` method, using the **InputFormat** and its associated **RecordReader**

The number of **HDFS** blocks for the file determines the number of input splits, which, in turn, determines the number of **Mapper** objects (or **Map** tasks) in a **MapReduce** application (also see the previous lecture)

Mappers do most of the heavy lifting in data processing in **MapReduce**, as they read the entire input file for the application

Mappers can also include setup and cleanup code to run in any given object lifespan

Setup is called before the `map()` method; cleanup is called after it

Building blocks of MapReduce program: Reducer

Reducer runs against a partition and each key and its associated values are passed to a `reduce ()` method inside **Reducer class**

Reduce's InputFormat matches **Mapper's** OutputFormat

While **Mapper** usually do the data preparation, for example, filtering and extracting), **Reducer** usually contains the main application logic

- For example, summation, counting, and averaging operations are implemented in **Reducers**

The runtime of **Reducer** instances is usually faster (and much faster in some cases) than the runtime of **Mapper** instances

Java MapReduce Application

Outline

[Building blocks of MapReduce program](#)

[Word Count: The "Hello, World" of MapReduce](#)

[Hadoop datatype objects](#)

[Input and output formats](#)

[Java code of Driver](#)

[Java code of Mapper](#)

[Java code of Reducer](#)

[Java code of ToolRunner](#)

[Setting up a local running environment](#)

[Combiner API](#)

[Partitioner API](#)

[Miscellaneous](#)

Word Count: The "Hello, World" of MapReduce

WordCount: Read a text file and count occurrences of each word

Consider a text document containing a fragment of the works of Shakespeare

Romeo and Juliet

```
O Romeo, Romeo! wherefore art thou Romeo?  
Deny thy father, and refuse thy name
```

The input format is **TextInputFormat**

After the text is read, the input to **Map** task, i.e. the process running **Mapper** is the following

Romeo and Juliet

```
(0, 'O Romeo , Romeo ! wherefore art thou Romeo ?')  
(45, 'Deny thy father, and refuse thy name')
```

Word Count: The "Hello, World" of MapReduce

The output of the **Map** task is the following

```
( 'O' , 1 )  
( 'Romeo' , 1 )  
( 'Romeo' , 1 )  
( 'wherefore' , 1 )  
( 'art' , 1 )  
( 'thou' , 1 )  
( 'Romeo' , 1 )  
( 'Deny' , 1 )  
( 'thy' , 1 )  
( 'father' , 1 )  
( 'and' , 1 )  
( 'refuse' , 1 )  
( 'thy' , 1 )  
( 'name' , 1 )
```

Key-Value pairs

Note that if you want to filter out some trivial words such as "a", "and", ... then it can be done in a **Mapper**

Word Count: The "Hello, World" of MapReduce

Before sending data to **Reduce** task, there is a **shuffle-and-sort** stage

Shuffle-and-sort is usually hidden from a programmer

The following is the input to **Reduce** task

Key-value pairs after shuffle-and sort

```
('and', [1])
('art', [1])
('Deny', [1])
('father', [1])
('name', [1])
('O', [1])
('refuse', [1])
('Romeo', [1, 1, 1])
('thou', [1])
('thy', [1, 1])
('wherefore', [1])
```

Word Count: The "Hello, World" of MapReduce

The following is the final output from **Reduce** task:

```
( 'and' , 1)
( 'art' , 1)
( 'Deny' , 1)
( 'father' , 1)
( 'name' , 1)
( 'O' , 1)
( 'refuse' , 1)
( 'Romeo' , 3)
( 'thou' , 1)
( 'thy' , 2)
( 'wherefore' , 1)
```

Final output from Reduce

Note that we use plain texts to illustrate the data passing through the **MapReduce** stages, but in the **Java** implementation, all texts are wrapped in some object that implements the **Writable** interface

Java MapReduce Application

Outline

[Building blocks of MapReduce program](#)

[Word Count: The "Hello, World" of MapReduce](#)

[Hadoop datatype objects](#)

[Input and output formats](#)

[Java code of Driver](#)

[Java code of Mapper](#)

[Java code of Reducer](#)

[Java code of ToolRunner](#)

[Setting up a local running environment](#)

[Combiner API](#)

[Partitioner API](#)

[Miscellaneous](#)

Hadoop datatype objects

Some imported package members

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

Imports

`org.apache.hadoop.io.IntWritable` contains a specification of `Writable` interface

Hadoop data type objects

In most programming languages, when defining most data elements, we usually use simple, or primitive, datatypes such as `int`, `long`, or `char`

However, in Hadoop a key or a value is an object that is an instantiation of a class, with attributes and defined methods

A key or a value contains (or encapsulates) the data with methods defined for reading and writing data from and to the object

Hadoop data type objects

`Writable` interface

Hadoop serialisation format is `Writable` interface

For example, a class that implements `Writable` is `IntWritable`, which is a wrapper for a Java `int`

One can create such a class and set its value in the following way

```
IntWritable writable = new IntWritable();  
writable.set(163);
```

Creating IntWritable object

Alternatively, we can write

```
IntWritable writable = new IntWritable(163);
```

Creating IntWritable object

Hadoop data type objects

`WritableComparable` interface

`IntWritable` implements `WritableComparable` interface

It is interface of `Writable` and `java.lang.Comparable` interfaces

Creating `WritableComparable` interface

```
package org.apache.hadoop.io;  
public interface WritableComparable <T> extends Writable,  
                                                Comparable<T> { ... }
```

Comparison is crucial for `MapReduce`, because `MapReduce` contains a sorting phase during which keys are compared with one another

`WritableComparable` permits to compare records read from a stream without deserialising them into objects, thereby avoiding any overhead of object creation

Hadoop data type objects

Hadoop primitive `Writable` wrappers

Writable Wrappers	
Writable Wrapper	Java Primitive
BooleanWritable	boolean
ByteWritable	byte
IntWritable	int
FloatWritable	float
LongWritable	long
DoubleWritable	double
NullWritable	null
Text	String

Java MapReduce Application

Outline

Building blocks of MapReduce program

Word Count: The "Hello, World" of MapReduce

Hadoop datatype objects

Input and output formats

Java code of Driver

Java code of Mapper

Java code of Reducer

Java code of ToolRunner

Setting up a local running environment

Combiner API

Partitioner API

Miscellaneous

Input and output formats

FileInputFormat (the base class of **InputFormat**) reads data (keys and values) from a given path, using the default or user-defined format

- The default input format is **LongWritable** for the keys and **Text** for the values

FileOutputFormat (the base class of **OutputFormat**) writes data into a file in a given path

- The output format is usually defined by a programmer
- For example, the output format is **Text** for the keys and **IntWritable** for the values

Java MapReduce Application

Outline

Building blocks of MapReduce program

Word Count: The "Hello, World" of MapReduce

Hadoop datatype objects

Input and output formats

Java code of Driver

Java code of Mapper

Java code of Reducer

Java code of ToolRunner

Setting up a local running environment

Combiner API

Partitioner API

Miscellaneous

Java code of Driver

Java code of Driver

```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        Configuration conf = new Configuration();  
        Job job = Job.getInstance(conf, "word count");  
        job.setJarByClass(WordCount.class);  
        job.setMapperClass(MyMapper.class);    //the Mapper class  
        job.setReducerClass(MyReducer.class); //the Reducer class  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        System.exit(job.waitForCompletion(true) ? 0 : 1);  
    }  
}
```

Java code of Driver

Job object and configuration

Driver class instantiates a **Job** object

A **Job** object creates and stores the configuration options for a **Job**, including the classes to be used as **Mapper** and **Reducer**, input and output directories, etc

The configuration options are specified in (one or more of) the following places

- **Hadoop** defaults (***-default.xml**, e.g., **core-default.xml**)
- A default configuration is documented in the Apache Hadoop documentation
- The ***-site.xml** files on the client node where **Driver** code is processed
- The ***-site.xml** files on the slave nodes where **Mapper** runs on
- **Configuration** properties set at the command line as arguments to a **MapReduce** application (in a **ToolRunner** object)
- **Configuration** properties set explicitly in code and compiled through a **Job** object

Java code of Driver

Driver routines:

- Parses the command line for positional arguments - `input` file(s)/directory and `output` directory
- Creates a new `Job` object instance, using `getConf()` method to obtain configuration from the various sources (`*-default.xml` and `*-site.xml`)
- Gives a `Job` a friendly name (the name you will see in the ResourceManager UI)
- Sets the `InputFormat` and `OutputFormat` for a `Job` and determines the input splits for a `Job`
- Defines `Mapper` and `Reducer` classes to be used for a Job (They must be available in the Java classpath where `Driver` is run - typically these classes are packaged alongside the Driver)
- Sets the final output key and value classes, which will be written out files in the output directory
- Submits a `Job` object through `job.waitForCompletion(true)`

Java MapReduce Application

Outline

Building blocks of MapReduce program

Word Count: The "Hello, World" of MapReduce

Hadoop datatype objects

Input and output formats

Java code of Driver

Java code of Mapper

Java code of Reducer

Java code of ToolRunner

Setting up a local running environment

Combiner API

Partitioner API

Miscellaneous

Java code of Mapper

Java code of Mapper

```
public static class MyMapper
extends Mapper{
    private final static IntWritable one = new IntWritable(1);
    private Text wordObject = new Text();
    public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
        String line = value.toString();
        for (String word : line.split("\\W+")) {
            if (word.length() > 0) {
                wordObject.set(word);
                context.write(wordObject, one);
            }
        }
    }
}
```

Java code of Mapper

`MyMapper` class

A class `MyMapper` extends a base `Mapper` class included within the `Hadoop` libraries

In the example, the four generics in `Mapper<Object, Text, Text, IntWritable>` represent `<map_input_key, map_input_value, map_output_key, map_output_value>`

These generics must correspond to:

- `Key-value pair` types as defined by `InputFormat` in Driver (may be the default one)
- `job.setMapOutputKeyClass` and `job.setMapOutputValueClass` defined in Driver
- Input and output to the `map()` method

Java code of Mapper

In `map()` method, before performing any functions against a key or a value (such as `split()`), we need to get the value contained in the serialised `Writable` or `WritableComparable` object, by using the `value.toString()` method

After performing operations against the input data (`key-value pairs`), the output data (intermediate data, also `key-value pairs`) are `WritableComparable` and `Writable` objects, both of which are emitted using a `Context` object

In the case of a `Map-only` job, the output from `Map phase`, namely the set of `key-value pairs` emitted from all `map()` methods in all map tasks, is the final output, without intermediate data or `Shuffle-and-Sort phase`

Java code of Mapper

Context object

A **Context** object is used to pass information between processes in Hadoop

We mostly invoke its **write()** method to write the output data from **Mapper** and **Reducer**

Other functions of **Context** object are the following

- It contains configuration and state needed for processes within the **MapReduce** application, including enabling parameters to be passed to distributed processes
- It is used in the optional **setup()** and **cleanup()** methods within a **Mapper** or **Reducer**

Java MapReduce Application

Outline

[Building blocks of MapReduce program](#)

[Word Count: The "Hello, World" of MapReduce](#)

[Hadoop datatype objects](#)

[Input and output formats](#)

[Java code of Driver](#)

[Java code of Mapper](#)

[Java code of Reducer](#)

[Java code of ToolRunner](#)

[Setting up a local running environment](#)

[Combiner API](#)

[Partitioner API](#)

[Miscellaneous](#)

Java code of Reducer

Java code of Reducer

```
public static class MyReducer
    extends Reducer {
private IntWritable result = new IntWritable();
public void reduce(Text key,
                    Iterable values,
                    Context context)
    throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values)
    {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
}
```


Java code of Reducer

`MyReducer` class

A class `MyReducer` extends the based `Reducer` class included with the `Hadoop` libraries.

The four generics in

`Reducer<Text, IntWritable, Text, IntWritable>` represents
<reduce_input_key, reduce_input_value, reduce_output_key,
reduce_output_value>

A `reduce()` method accepts a key and an `Iterable` list of values as input

The values have type `Iterable<T>` (e.g., `Iterable<IntWritable>`)

Java code of Reducer

As in [Mapper](#), to operate or perform Java string or numeric operations against keys or values from the input list of values, we first extract a value included in [Hadoop](#) object

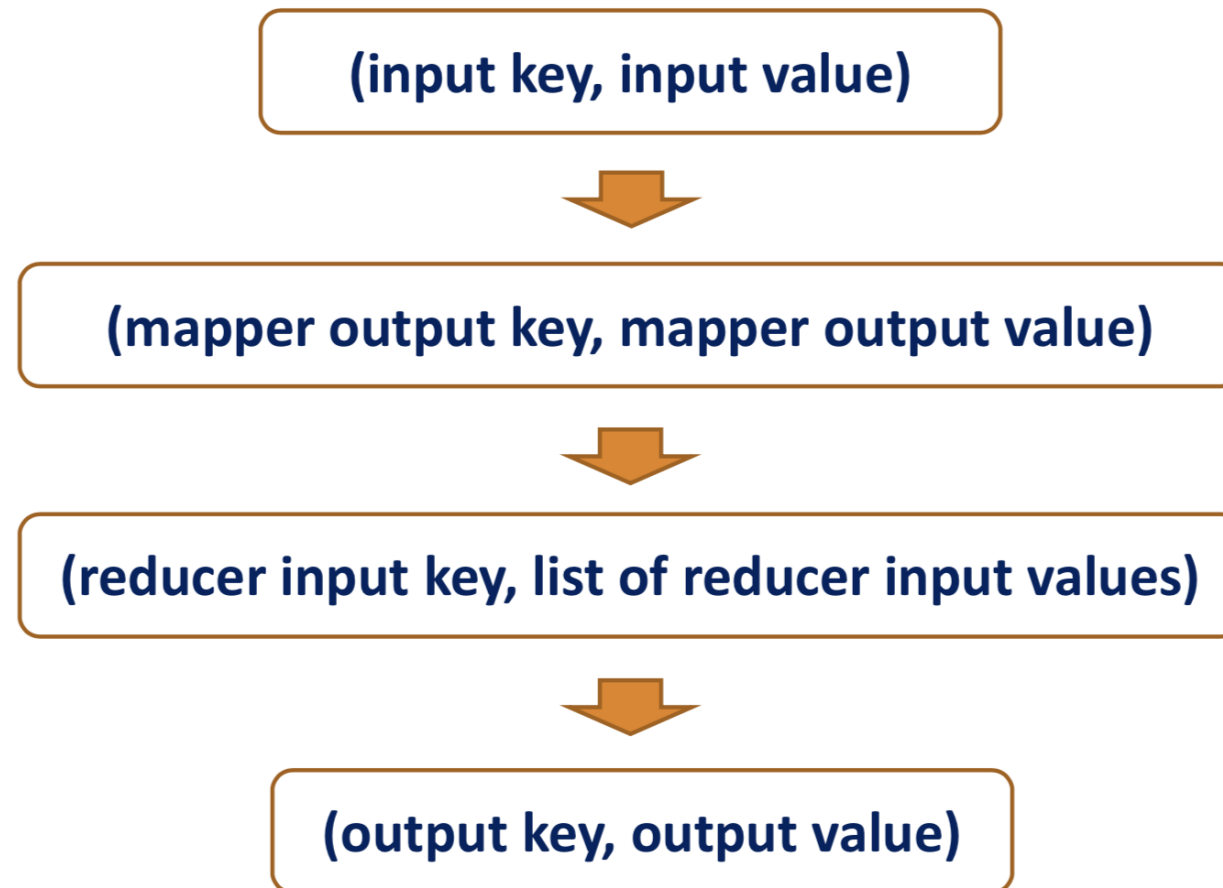
Also, the emit of **key-value pairs** in the form of `WritableComparable` objects for keys and values uses a `Context` object

The Reducer class implements the main application logic

For example, the actually counting of words is implemented in **Reducer**

Java code of Reducer

Data flow of keys and values



Java MapReduce Application

Outline

[Building blocks of MapReduce program](#)

[Word Count: The "Hello, World" of MapReduce](#)

[Hadoop datatype objects](#)

[Input and output formats](#)

[Java code of Driver](#)

[Java code of Mapper](#)

[Java code of Reducer](#)

[Java code of ToolRunner](#)

[Setting up a local running environment](#)

[Combiner API](#)

[Partitioner API](#)

[Miscellaneous](#)

Java code of ToolRunner

Despite optional, **Driver** can leverage a class called **ToolRunner**, which is used to parse command-line options

A class ToolRunner

```
// ... Originally imported package members of WordCount
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.conf.Configured;
public class WordCountTR extends Configured implements Tool {
    public static void main(String[] args)
        throws Exception {
        int res = ToolRunner.run(new Configuration(), new WordCount(), args);
        System.exit(res);
    }
// "run" method of ToolRunner (in the next slide)
}
```

Java code of ToolRunner

run method of ToolRunner

run method of ToolRunner

```
@Override
public int run(String[] args) throws Exception {
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf, "word count with ToolRunner");
    job.setJarByClass(WordCountTR.class);
    job.setMapperClass(MyMapper.class); //the Mapper class
    job.setReducerClass(MyReducer.class); //the Reducer class
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    return job.waitForCompletion(true) ? 0 : 1;
}
```

Java code of ToolRunner

ToolRunner in a command line

ToolRunner enables flexibility in supplying configuration parameters at the command line when submitting a **MapReduce** job

The following submission has a command to specify the number of **Reduce** tasks from the command line

ToolRunner in a command line

```
hadoop jar mr.jar MyDriver -D mapreduce.job.reduces=10 myinputdir myoutputdir
```

The following submission specifies the location of **HDFS**

ToolRunner in a command line

```
hadoop jar mr.jar MyDriver  
-D fs.defaultFS=hdfs://localhost:9000 myinputdir myoutputdir
```

Java code of ToolRunner

ToolRunner options

ToolRunner options

Option	Description
<code>-D property=value</code>	Sets the given Hadoop configuration property to the given value. Overrides any default or site properties in the configuration and any properties set via the <code>-conf</code> option.
<code>-conf filename ...</code>	Adds the given files to the list of resources in the configuration. This is a convenient way to set site properties or to set a number of properties at once.
<code>-fs uri</code>	Sets the default filesystem to the given URI. Shortcut for <code>-D fs.defaultFS=uri</code> .
<code>-jt host:port</code>	Sets the YARN resource manager to the given host and port. (In Hadoop 1, it sets the jobtracker address, hence the option name.) Shortcut for <code>-D yarn.resourcemanager.address=host:port</code> .

Java code of ToolRunner

ToolRunner options

ToolRunner options

Option	Description
<code>-files file1, file2, ...</code>	Copies the specified files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (e.g. HDFS) and makes them available to MapReduce programs in the working directory of task.
<code>-archives archive1, archive2, ...</code>	Copies the specified archives from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS), unarchives them, and makes them available to MapReduce programs in the working directory of task.
<code>-libjars jar1, jar2, ...</code>	Copies the specified JAR files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and adds them to the MapReduce classpath of task. This option is a useful way of shipping JAR files that a job is dependent on.

Java MapReduce Application

Outline

Building blocks of MapReduce program

Word Count: The "Hello, World" of MapReduce

Hadoop datatype objects

Input and output formats

Java code of Driver

Java code of Mapper

Java code of Reducer

Java code of ToolRunner

Setting up a local running environment

Combiner API

Partitioner API

Miscellaneous

Setting up a local running environment

A local running environment is often convenient in MapReduce application development

- Before sending it out to the whole cluster.
- Used for debugging and testing

Create a configuration file, say, `hadoop-local.xml`

```
<property>
  <name>fs.defaultFS</name>
  <value>file:///</value>
</property>
<property>
  <name>mapreduce.framework.name</name>
  <value>local</value>
</property>
```

hadoop-local.xml

Setting local environment

```
hadoop jar mr.jar MyDriver -conf hadoop-local.xml myinputdir myoutputdir
```

Java MapReduce Application

Outline

[Building blocks of MapReduce program](#)

[Word Count: The "Hello, World" of MapReduce](#)

[Hadoop datatype objects](#)

[Input and output formats](#)

[Java code of Driver](#)

[Java code of Mapper](#)

[Java code of Reducer](#)

[Java code of ToolRunner](#)

[Setting up a local running environment](#)

[Combiner API](#)

[Partitioner API](#)

[Miscellaneous](#)

Combiner API

Combiner functions can decrease the amount of intermediate data sent between **Mappers** and **Reducers** as part of a **Shuffle-and-Sort** process

- In a sense, **Combiner** is the "map-side reducers"

One can reuse **Reducer** code to implement **Combiner** if

- A **combiner()** function is identical to a **reduce()** function defined in your **Reducer** class
- The output key and value object types from a **map()** function implemented in **Mapper** match the input to the function used in **Combiner**
- The output key and value object types from the function used in **Combiner** match the input key and value object types used in **Reducer's** **reduce()** method
- The operation to be performed is commutative and associative.

Java MapReduce Application

Outline

[Building blocks of MapReduce program](#)

[Word Count: The "Hello, World" of MapReduce](#)

[Hadoop datatype objects](#)

[Input and output formats](#)

[Java code of Driver](#)

[Java code of Mapper](#)

[Java code of Reducer](#)

[Java code of ToolRunner](#)

[Setting up a local running environment](#)

[Combiner API](#)

[Partitioner API](#)

[Miscellaneous](#)

Partitioner API

Partitioner divides the output keyspace for a **MapReduce** application, controlling which **Reducers** get which intermediate data

- It is useful in process distribution or load balancing, for example, getting more **Reduce** tasks running in parallel
- It can be used to segregate the outputs, for example, creating a file for monthly data in a year

A default **Partitioner** is a **HashPartitioner**, which arbitrarily hashes the key space such that

- the same keys go to the same **Reducers** and
- the keyspace is distributed roughly equally among the number of **Reducers** when determined by a programmer

In case of one **Reduce** task (default in pseudo-distributed mode), the **Partitioner** is "academic" because all intermediate data goes to the same **Reducer**

Partitioner API

Example: LetterPartitioner

LetterPartitioner

```
// ... other imported package members
import org.apache.hadoop.mapreduce.Partitioner;
public static class LetterPartitioner
    extends Partitioner {
    @Override
    public int getPartition(Text key,
                           IntWritable value,
                           int numReduceTasks) {
        String word = key.toString();
        if (word.toLowerCase().matches("[a-m].*$")) {
//            if word starts with a to m, go to the first Reducer or partition
            return 0;
        } else {
//            else go to the second Reducer or partition
            return 1;
        }
    }
}
```


Partitioner API

Declare `Combiner` and `Partitioner` in a Driver

Declaring Combiner and Partitioner in a Driver

```
public class WordCountWithLetPar {  
    public static void main(String[] args) throws Exception {  
        ...  
        job.setMapperClass(MyMapper.class);    // Mapper class  
        job.setCombinerClass(MyReducer.class); // Combiner class, which is  
                                                same as Reducer class in this program  
        job.setPartitionerClass(MyPartitioner.class); // Partitioner class  
        job.setReducerClass(MyReducer.class);  // Reducer class  
        ...  
    }  
}
```

Java MapReduce Application

Outline

[Building blocks of MapReduce program](#)

[Word Count: The "Hello, World" of MapReduce](#)

[Hadoop datatype objects](#)

[Input and output formats](#)

[Java code of Driver](#)

[Java code of Mapper](#)

[Java code of Reducer](#)

[Java code of ToolRunner](#)

[Setting up a local running environment](#)

[Combiner API](#)

[Partitioner API](#)

[Miscellaneous](#)

Miscellaneous

Some points for Mappers & Reducers Implementation

- **No saving or sharing of state:** do not attempt to persist or share data between **Map** tasks and between **Reduce** tasks
- **No side effects:** as **Map** and **Reduce** tasks may be executed in any sequence and may be executed more than once, any task must not create any side effects, such as creating external events or triggering external processes
- **No attempt to communicate with other tasks:** the **Map** tasks run in an interleaving manner, and do the **Reduce** tasks

Miscellaneous

Compound **MapReduce** jobs

How to turn a (possibly complex) data processing problem into the **MapReduce model** ?

As a rule of thumb, for complex data processing problems, think about adding more jobs, rather than adding the complexity to jobs

For example, **to find the mean of maximum daily temperature in each month in different weather stations**, consider these two jobs:

- Compute the maximum daily temperature for every station-day key
- Compute the mean of the maximum daily temperature for every station-day-month key

A library **ChainMapper** allows to develop chained MapReduce Jobs

References

White T., Hadoop The Definitive Guide: Storage and Analysis at Internet Scale, O'Reilly, 2015