MAPREDUCE OVERVIEW

MapReduce Overview

- **Apache Hadoop**
- MapReduce Programming Paradigm
- Components of a MapReduce Program
- Writing a MapReduce Program

Apache Hadoop

- MapReduce, a part of the core Apache Hadoop project, allows us writing distributed programs processing vast amounts of data
  - in a parallel and distributed manner
  - on large (of thousands of nodes) clusters
  - reliably

analyticscenter

Apache Hadoop

- MapReduce, a part of the core Apache Hadoop project, allows us writing distributed programs processing vast amounts of data
  - in a parallel and distributed manner
  - on large (of thousands of nodes) clusters
  - reliably

Apache Hadoop

- MapReduce, a part of the core Apache Hadoop project, allows us writing distributed programs processing vast amounts of data
  - in a parallel and distributed manner
  - on large (of thousands of nodes) clusters
  - reliably

MapReduce Overview

- Apache Hadoop
- **MapReduce Programming Paradigm**
- Components of a MapReduce Program
- Writing a MapReduce Program

MapReduce Programming Paradigm

- MapReduce, a part of the core Apache Hadoop project, allows us writing distributed programs processing vast amounts of data
  - in a parallel and distributed manner
  - on large (of thousands of nodes) clusters
  - reliably

MapReduce Programming Paradigm

- MapReduce framework is an implementation for efficient processing of data sets of records of pairs, where:
  - **Map tasks** run a user defined function on each record in a completely parallel manner and emit new pairs
  - The framework defined **Partitioner** sends the Map results with the same key to the same **Reducer** nodes
    - It is common to override this behavior (custom partitioning) in advanced use cases

MapReduce Programming Paradigm

- MapReduce framework is an implementation for efficient processing of data sets of records of pairs, where:
  - The framework defined **grouping&sorting** mechanisms group the pairs with the same key (first element) together
  - **Reduce tasks** run a user defined function on each group of values associated with the same key in a completely parallel fashion and emit new pairs
    - It is also common to override the grouping behavior in advanced use cases

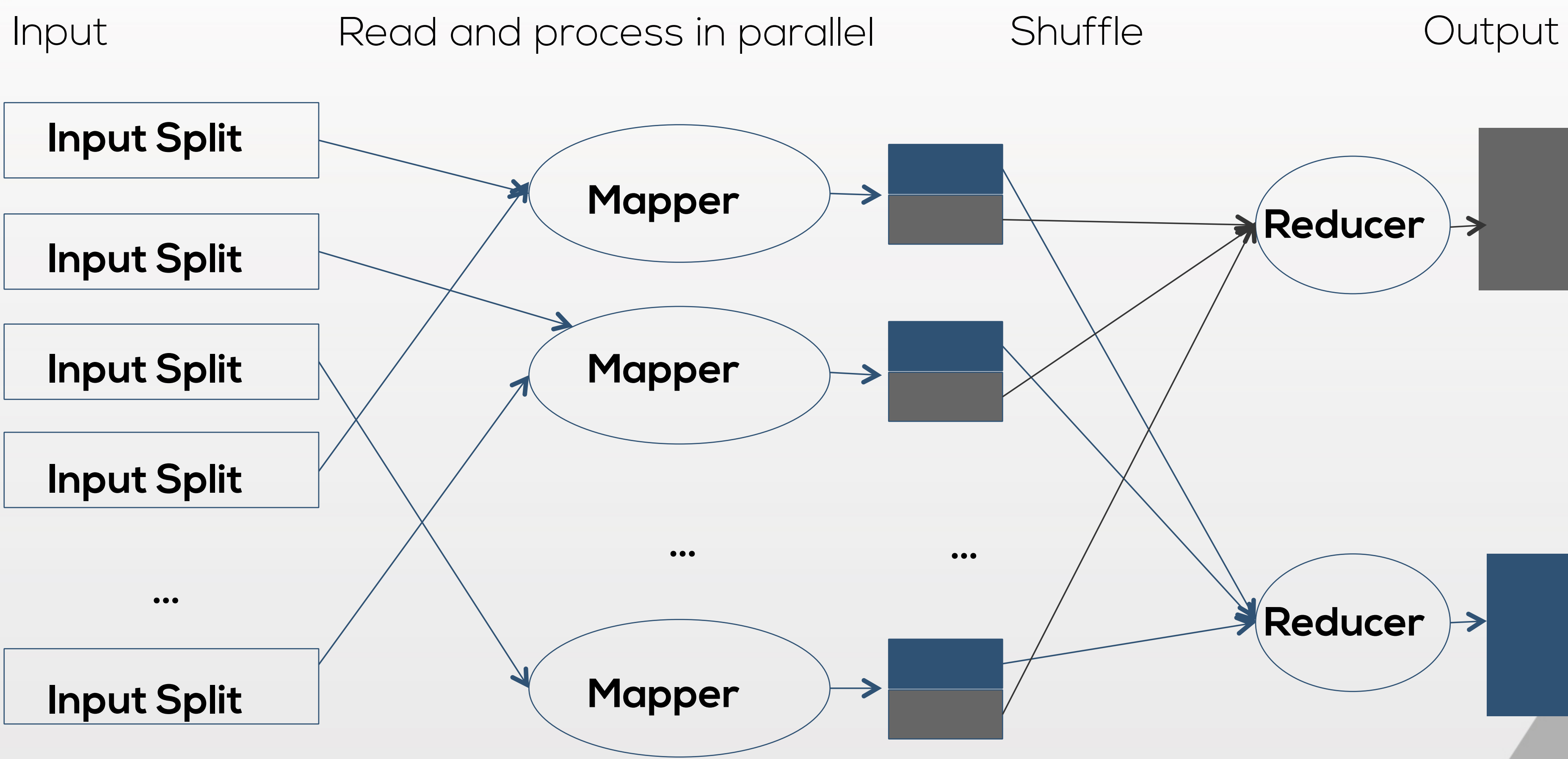
MapReduce Programming Paradigm

- For a MapReduce job to run:
  - How the input data is split into chunks, for parallel execution
  - The iteration over records behavior
  - How a record is interpreted as a pair should be defined
- This behavior is defined by setting an **InputFormat** for a MapReduce job
- The framework takes care of scheduling (of multiple MapReduce jobs) and monitoring tasks, and re-executing the failed ones

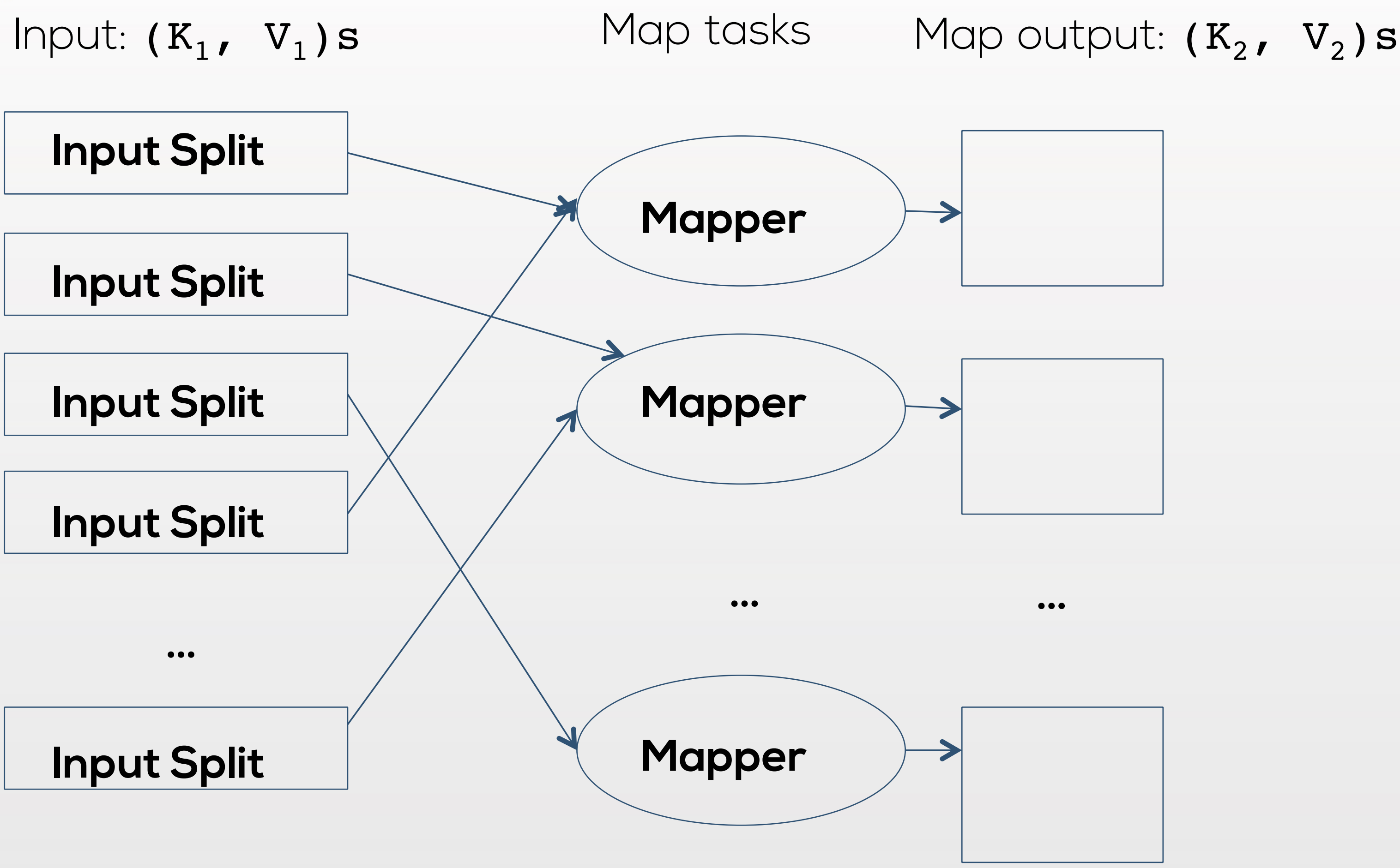
MapReduce Programming Paradigm

- Applications specify
  - the I/O Formats,
  - input/output locations
  - *map* and *reduce* functions by implementing the appropriate interfaces (`Mapper#map`, `Reducer#reduce`)
    - To transform/filter records, `map` functions are defined to run on each input record
    - `reduce` functions are defined such that they run on groups of values attached to a key

analyticscenter

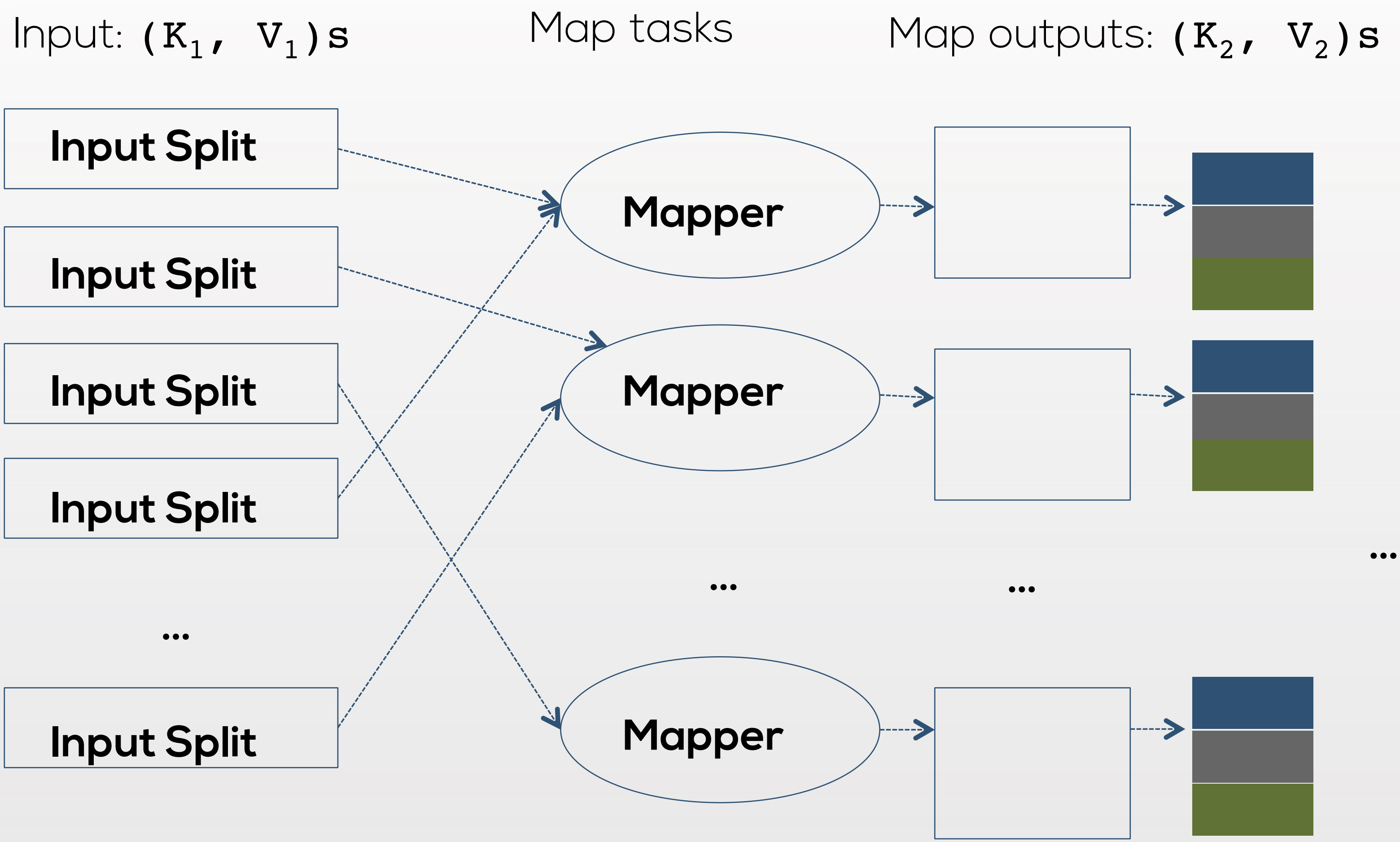


Google's MapReduce

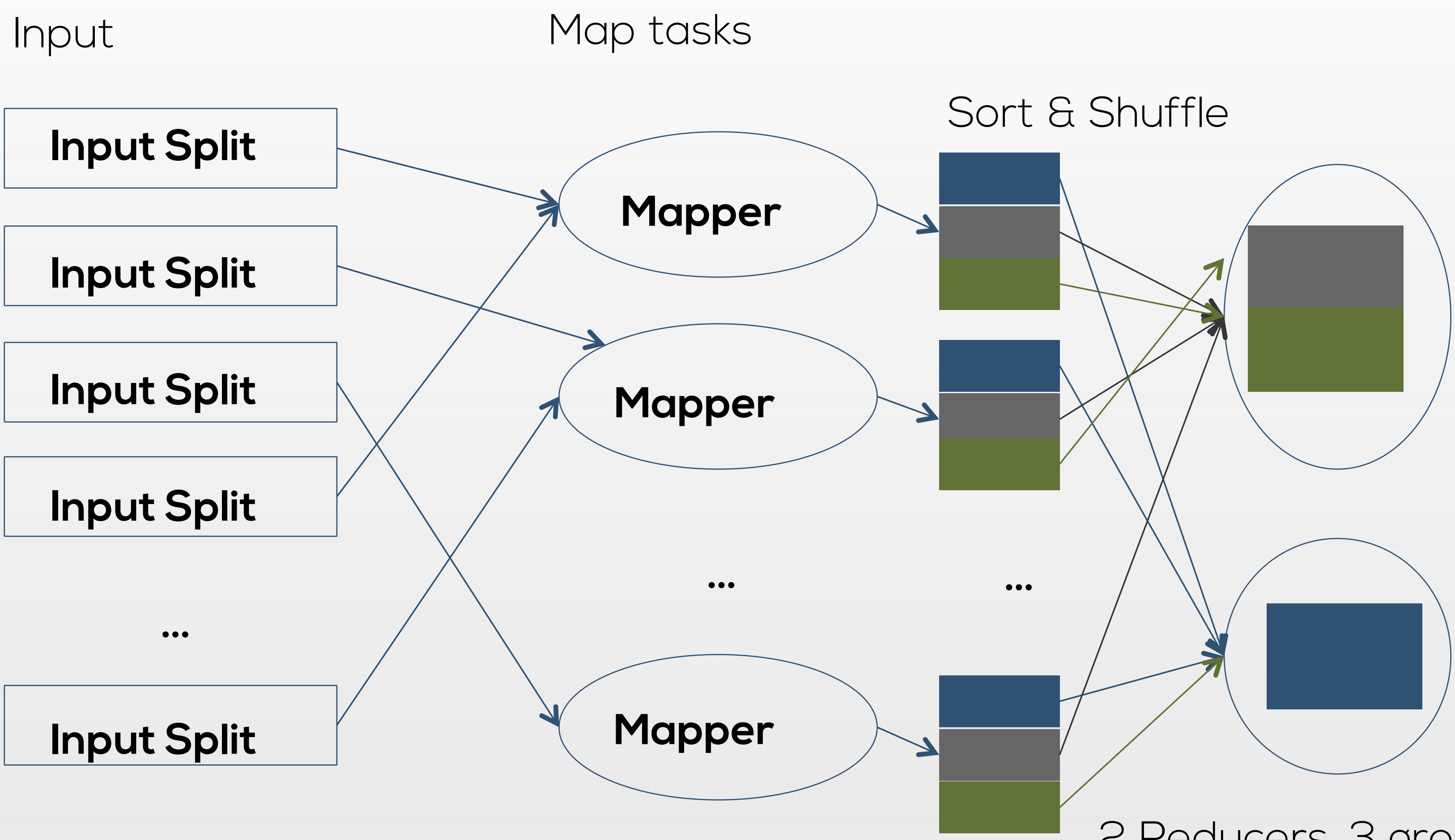


Each split is assigned to a Map task, each record of which are processed individually

MapReduce Programming Model

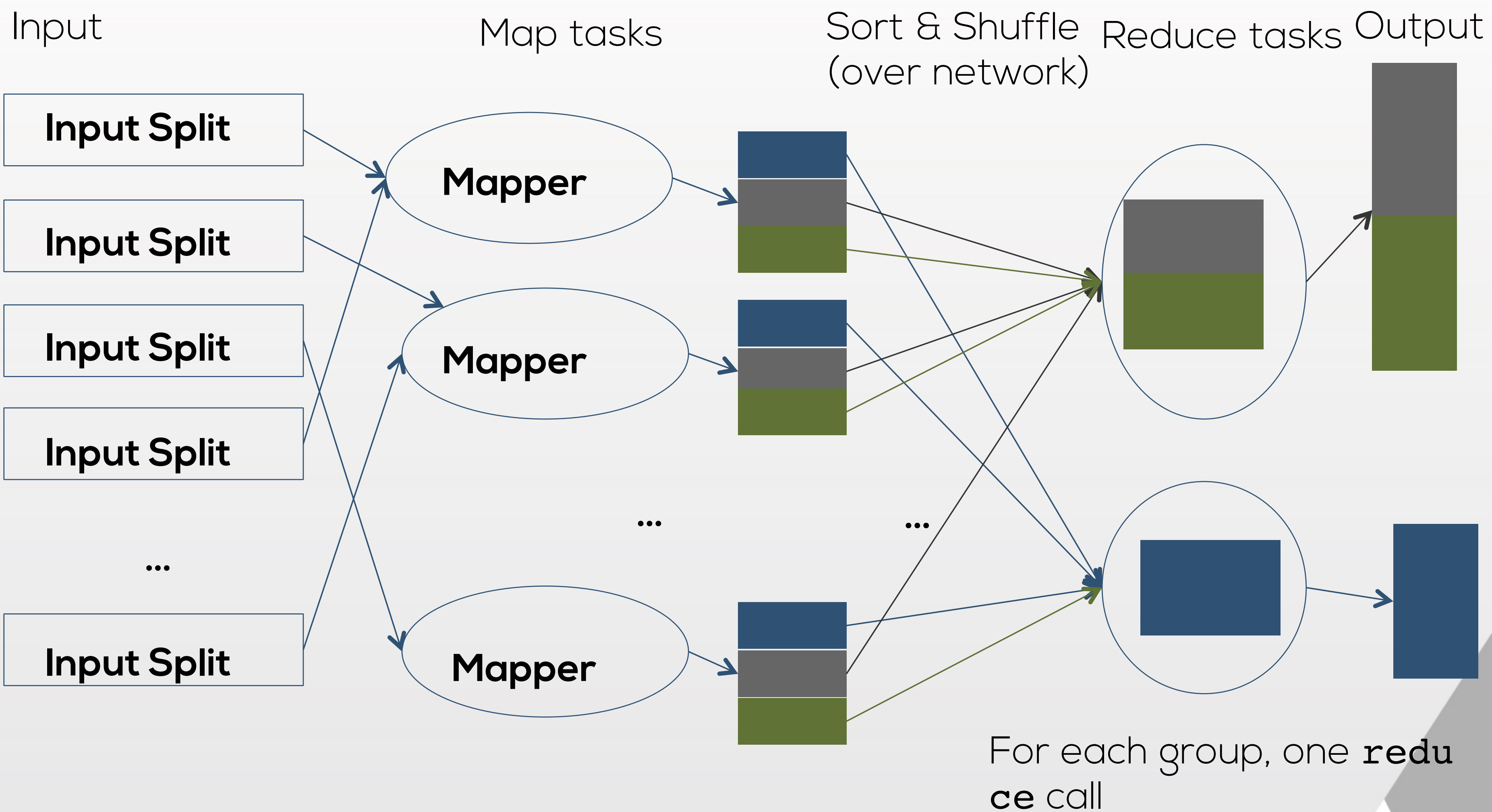


MapReduce Programming Model



2 Reducers, 3 groups, distant groups with the same keys collected together

MapReduce Programming Model



MapReduce Programming Model

MapReduce Overview

- Apache Hadoop
- MapReduce Programming Paradigm
- **Components of a MapReduce Program**
- Writing a MapReduce Program

Mapper

- Developers need to implement the abstract method `org.apache.hadoop.mapreduce.Mapper#map`
- The input for a map function is just a single record (as a pair of key/value)
- The map function writes zero or more key/values to the `org.apache.hadoop.mapreduce.MapContext`

analyticscenter

Mapper & InputFormat

- The record boundaries and its key-value separation is defined by the `org.apache.hadoop.mapreduce.RecordReader`
- `RecordReader` is a part of the `org.apache.hadoop.mapreduce.InputFormat` specified by the user
- `RecordReader` is a part of the `org.apache.hadoop.mapreduce.InputFormat` specified by the user

Mapper & InputFormat

- A `RecordReader` is used to iterate over an `org.apache.hadoop.mapreduce.InputSplit`
  - There would be many `InputSplit`s, each representing a partition of the entire input data set
  - Number of Mappers is determined by the number of `InputSplit`s of the data set
  - `InputSplit`s are also determined by the `InputFormat`

Mapper & InputFormat

- To summarize, using the **`InputFormat`**, the input data set is partitioned into multiple **`InputSplit`**s, and for each **`InputSplit`** in parallel:
  - A **`RecordReader`** is created to iterate over it and is passed to a **`MapContext`**
  - A Map task (An app running the **`Mapper`**) is created from this **`MapContext`**, and its **`map`** method is called for each of the corresponding **`InputSplit`**'s records
  - The output key/values are written to the **`MapContext`**

Shuffle & Sort

- Recall that number of Map tasks are determined by the number of `InputSplits`
- This is not the case for the Reduce tasks, number of Reducers are set by the submitter of the job
  - This is a configuration property (`mapred.reduce.tasks`) that can be set for a job, and it has a default value
- The outputs from Map tasks are partitioned into numReducers number of files, based on the `Partitioner` behavior
  - The default Partitioning behaviors is hash-modulo-numReducers
- After a set of key/values for each Reduce task are created from Map outputs, these sets are transferred to the appropriate Reduce tasks

Partitioner

- The outputs from Map tasks are partitioned into numReducers number of files, based on the `Partitioner` behavior
  - The default `Partitioner` is the `HashPartitioner` (the behavior is hash-modulo-numReducers)
  - Users can specify their own `Partitioner` (`Job#setPartitionerClass`)

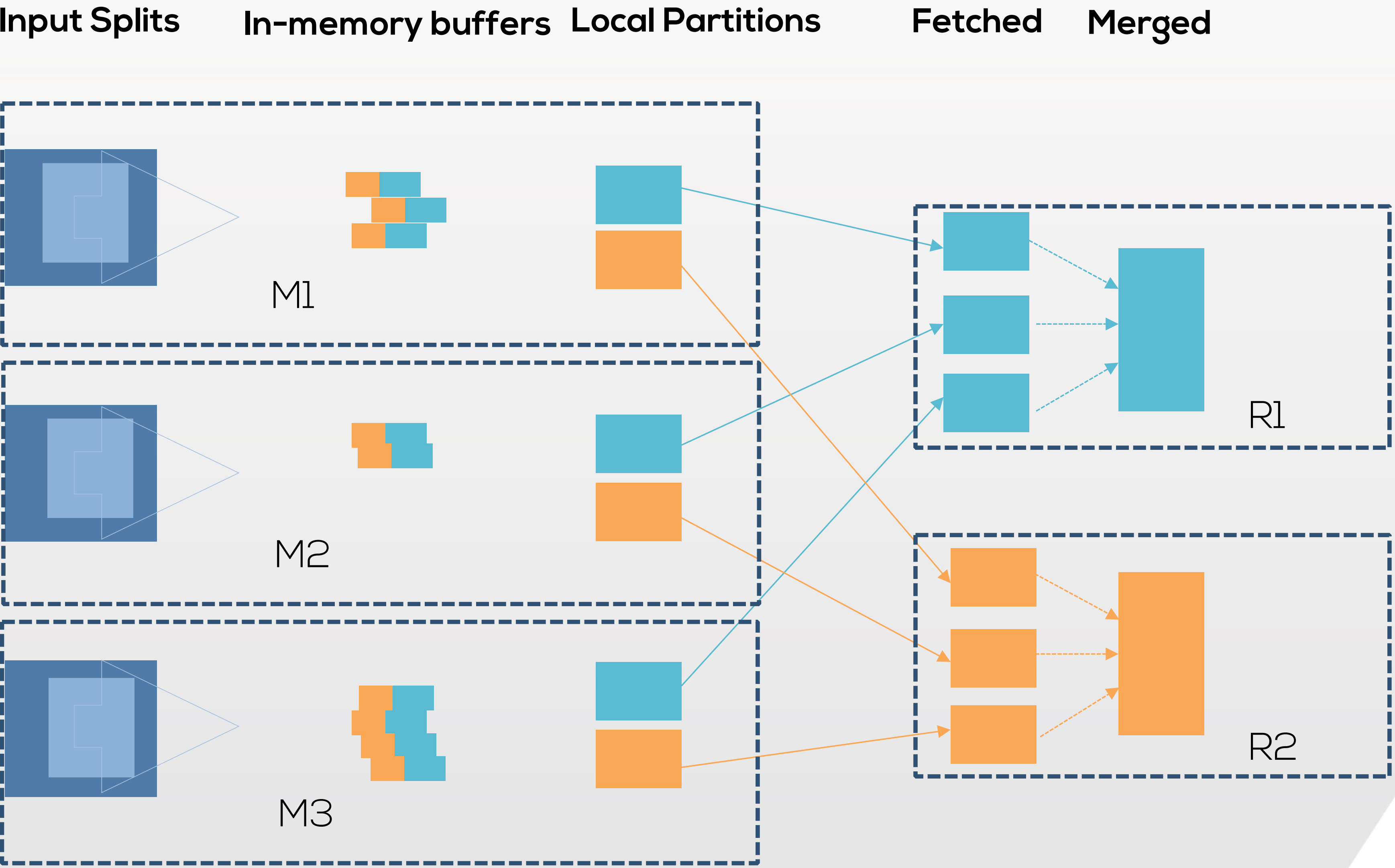
Shuffle & Sort & Reducer

- For each global partition (set of corresponding local partitions transferred from the Map task outputs), a Reduce task (an app running a `org.apache.hadoop.mapreduce.Reducer`) is created
- All key/values assigned to a Reduce task are grouped by key (and sorted by key)
- For each key/{set of values} group, `Reducer#reduce` method is called
- The reduce function writes zero or more key/values to the `org.apache.hadoop.mapreduce.ReduceContext`

Grouping and Sorting Behavior

- The input to a Reduce task is grouped by key and sorted
  - The comparator that controls which keys are grouped together for a single call can be overridden (`Job#setGroupingComparatorClass`)
  - The comparator that controls how the keys are sorted can also be overridden, independently from the above (`Job#setSortComparatorClass`)
- By default, Map outputs are sent to the reducers, and sorting and grouping are all performed by key
- There are some advanced use cases where changing this behavior is beneficial (such as implementing a SecondarySort)

Shuffle & Sort



analyticscenter

MapReduce Overview

- Apache Hadoop
- MapReduce Programming Paradigm
- Components of a MapReduce Program
- **Writing a MapReduce Program**

Writing a MapReduce Program

- To write a MapReduce program, the user typically
  - Determines the `InputFormat`
  - Determines the number of Reduce tasks
  - Writes a `Mapper` for performing Map tasks
  - Writes a `Reducer` for performing Reduce tasks
  - Creates a `Job`, configures it, and runs

Determining the InputFormat

- For file base `InputFormat`s, the base class is the `FileInputFormat`
- `FileInputFormat` calculates splits based on the HDFS blocks (should check if the file is splittable)
- It is the `RecordReader`'s responsibility to respect record boundaries to present a record-oriented view to the Map tasks
- The concrete `TextInputFormat` for example, is for input data sets of plain text files. It is assumed that the files are broken into lines, and each line is interpreted as a record
  - Keys are position in the file, and values are the line of text

Note on the InputFormat

- MapReduce, and all prominent processing engines (such as Spark) can understand the `InputFormat`
  - That is, whenever and `InputFormat` is defined on a kind of dataset, whether or not it actually resides on your cluster, it can be treated as a distributed dataset, it is available to MapReduce, Spark, etc.
- Similarly, virtually all popular distributed storage systems offer InputFormats for the processing engines to read from

Determining Number of Reduce Tasks

- This value is set via the configuration parameter `mapred.reduce.tasks`
- It can be set on a per-job basis
- A default value can be put into `mapred-site.xml`

Writing the Mapper

- Typically, the map method is overridden
- A map method should be designed such that it can run on an arbitrary record
  - The default implementation is the identity mapper
- There are also two more methods, setup and cleanup, that are called before this Map task starts to process records, and after all records are processed, recpectively
  - They do nothing by default, and might be overridden

Writing the Reducer

- Typically, the reduce method is overridden
- A reduce method should be designed such that it can run on a group of values associated with the same key, collected from all Map outputs

An Example Mapper

```
public 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);
    }
  }
}
```


analyticscenter

An Example Reducer

```
public 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);
  }
}
```

Configuring and Submitting a Job

- The usual way to setup and submit a job is by using a `Job` instance
  - The `Mapper`, `Reducer`, `Combiner`, `Configuration`, `InputFormat`, `OutputFormat`, output key/value classes are passed to the created `Job` instance
  - `Job` is then submitted

Running the Example MapReduce Job

```
public class WordCount {
  public static void main(String[] args){
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, “my word count app”);

    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

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

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

Configuring and Submitting a Job

- Alternatively, a **`JobConf`** instance and the **`JobClient`** can be used to configure and submit the job
  - The **`Mapper`**, **`Reducer`**, **`Combiner`**, **`Configuration`**, **`InputFormat`**, **`OutputFormat`**, output key/value classes are passed to the created **`JobConf`** instance
  - **`JobClient.runJob`** is then submitted

analyticscenter

Configuring and Submitting a Job

```
public class WordCount {
  public static void main(String[] args){
     JobConf job = new JobConf(new Configuration(),
                                 MyJob.class);
     job.setJobName(”wordcount");
     job.setInputPath(new Path(args[0]));
     job.setOutputPath(new Path(args[1]));
     job.setMapperClass(Tokenizer.class);
     job.setReducerClass(IntSumReducer.class);
     job.setCombinerClass(IntSumReducer.class);
     job.setOutputKeyClass(Text.class)
     job.setOutputValueClass(IntWritable.class);

     JobClient.runJob(job);

  }
}
```


analyticscenter

Job Submission

- Users can bundle their job code in a jar file and execute it using the hadoop jar utility:

```
$ hadoop jar <jar> [MainClass] args...
```

- Besides the arguments for the program, MapReduce jobs can take generic command-line options
- For handling of generic command-line options, **`Tool`** interface can be used

analyticscenter

Tool Interface

- `Tool` interface supports handling of generic command-line options
- A typical `Tool` implements the `org.apache.hadoop.util.Tool` interface (which has an abstract `run` method)

Tool Interface

- Using the **`Tool`** interface with the **`GenericOptionParser`** makes it easy, via, **`ToolRunner`** to handle the generic options, such as:
  - **`-conf <configuration_file>`**: Used to specify a configuration file for the application
  - **`-D <property>=<value>`**: Used to specify properties
  - **`-files <comma_separated_list_of_files>`**: Used to specify files to be copied to the M/R cluster
  - **`-libjars <comma_separated_list_of_jars>`**: Used to specify jar files to include in the classpath

  and run the jobs

Tool Interface

```
//from hadoop.apache.org
public class MyApp extends Configured implements Tool {

       public int run(String[] args) throws Exception {
         Configuration conf = getConf();

         // Generic options are automatically handled by the GenericOptionParser
         String[] otherArgs = new GenericOptionParser(conf, args).getRemainingArgs()

         // Create a JobConf using the processed conf
         JobConf job = new JobConf(conf, MyApp.class);

         // Process custom command-line options
         Path in = new Path(otherArgs[1]);
         Path out = new Path(otherArgs[2]);

         // Specify various job-specific parameters
         job.setJobName("my-app");
         job.setInputPath(in);
         job.setOutputPath(out);
         job.setMapperClass(MyMapper.class);
         job.setReducerClass(MyReducer.class);

         // Submit the job, then poll for progress until the job is complete
         JobClient.runJob(job);
         return 0;
       }

       public static void main(String[] args) throws Exception {
         // Let ToolRunner handle generic command-line options
         int res = ToolRunner.run(new Configuration(), new MyApp(), args);

         System.exit(res);
       }
}
```

Tool Interface

- With generic options, job submission becomes like the following:

```
$ hadoop jar <jar-file> [MainClass] [generic options] args...
```


analyticscenter

ProgramDriver

- Hadoop also comes with a driver that is used to run programs added to it: **`org.apache.hadoop.util.ProgramDriver`**
- The application's entry point (The main method) instantiates a ProgramDriver object
  - To this object, programs are added (with a short name, preferably, and with concrete **`Tool`**s)
  - After additions are complete, **`ProgramDriver#run`** is called with the main arguments
  - The arguments are then parsed and passed to the correct instances by the framework, automatically

Example Driver

```
public class ExampleDriver {

  public static void main(String argv[]){
    int exitCode = -1;
    ProgramDriver pgd = new ProgramDriver();
    try {
      pgd.addClass("wordcount", WordCount.class,
                   "Counts the words in the input files.");
      pgd.addClass("grep", Grep.class,
                   "Counts the matches of a regex in the
                   input");
      exitCode = pgd.run(argv)
     }
     catch(Throwable e) {
       e.printStackTrace();
     }

    System.exit(exitCode);
  }
}
```

ProgramDriver

- In the example code, **`WordCount`** and **`Grep`** can still be **`Tool`** implementations with a main method calling **`ToolRunner.run`**
- Once the **`ExampleDriver`** is set as the main class of the bundled jar, the examples can be run using:

```
$ hadoop jar <jar> wordcount [generic options] args...
```

```
$ hadoop jar <jar> grep [generic options] args...
```


analyticscenter

Developing MapReduce Applications

- To summarize, to write a MapReduce program, a developer writes Mapper, Reducer, and driver code
- Usually, we use the default Configuration variables, or override the existing ones
- Hadoop uses its custom Configuration API, and a `Configuration` instance represents a collection of properties with names and values

Configuration

- A Configuration resource is an XML file, which looks like:

```
<configuration>
 <property>
  <name>mapreduce.reduce.tasks</name>
  <value>10</value>
 </property>

 <property>
  <name>mapreduce.job.user.classpath.first</name>
  <value>true</value>
 </property>

 <property>
  <name>my.configuration.param</name>
  <value>val</value>
 </property>
</configuration>
```

Configuration

- When such a resource is added using the

  `conf.addResource (path)`method, `conf.get("my.configuration.parameter")` and `conf.getInt("mapreduce.reduce.tasks")` calls can be made

- A Job's Configuration is shipped to the tasks, so for distributing small metadata (in a configuration format), Configuration can be used

- A single Configuration parameter can be set using the API, via `Configuration#set(String, String)`

analyticscenter

Configuration

- Multiple configuration resources can be defined for a job:
- The configuration that is applied the latest overrides the previously applied ones
- Hadoop by default specifies two resources from the classpath, loaded in the following order: **`core-default.xml, core-site.xml`**
- The practice is leaving the **`*-default.xml`** files for Hadoop defaults, and making site specific configurations in **`*-site.xml`**

Configuration

- Value Strings can be of expansions of other configuration property values
- Example the below configuration replaces **`${user.name}`** with **`dev`**

```
<configuration>
 <property>
  <name>user.name</name>
  <value>dev</value>
 </property>

 <property>
  <name>user.home.dir</name>
  <value>/home/${user.name}</value>
 </property>
</configuration>
```

Configuration

- Configuration files are picked from the classpath, but we can override the configuration resource by:
  - Passing **`--conf`** generic option
  - setting **`HADOOP_CONF_DIR`** environment variable
- The above alternatives make it easy to switch between different configurations

Developing MapReduce Applications

- Classes and Interfaces required to develop MapReduce programs are available in the **`hadoop-client`** project, which is available in central maven repository
  - Maven is an ASF project that is commonly used to manage the build lifecycle of –mainly Java– software
  - Most IDEs are tightly integrated with Maven
- In a client application simply adding **`hadoop-client`** as a dependency (from your favorite vendor, probably), packaging the project, and setting your driver class as the main class would suffice to use **`hadoop`** (or **`hadoop jar`**) utility to submit the a MapReduce job

analyticscenter

Developing MapReduce Applications

- *-default.xml and *-site.xml configuration resources would be added by Hadoop automatically, if they are available in the classpath
- If a 3rd party jar is used in the project, it should be added to the bundled jar, or it needs to be shipped to all tasks. These can be done as it is described below, respectively:
  - creating an uber jar containing all the dependencies
  - passing **-libjars <comma_separated_jar_files>** generic option

Demo

Developing the WordCount Maven Application and Submitting through ProgramDriver

MapReduce
Overview

End of Chapter