WRITING MAP REDUCE FOR HADOOP (BASICS)

Master Program in Computer Science
University of Calabria

Prof. F. Ricca

Data Types

- The MapReduce Types
  - Framework won't allow any arbitrary class
  - Needs serializing the key/value pairs
- The **Writable** interface
  - *Implemented by values*
  - Values are simply passed through
- The **WritableComparable<T>** interface
  - A combination of Writable and java.lang.Comparable<T>
  - *Implemented by keys or values*
  - Keys must be comparable because
    - They will be sorted at the reduce stage

Some standard types

Class	Description
`BooleanWritable`	Wrapper for a standard Boolean variable
`ByteWritable`	Wrapper for a single byte
`DoubleWritable`	Wrapper for a Double
`FloatWritable`	Wrapper for a Float
`IntWritable`	Wrapper for a Integer
`LongWritable`	Wrapper for a Long
`Text`	Wrapper to store text using the UTF8 format
`NullWritable`	Placeholder when the key or value is not needed

Custom types

```
public class Edge implements WritableComparable<Edge>{

    private String departureNode;
    private String arrivalNode;

    public String getDepartureNode() { return departureNode;}

    @Override
    public void readFields(DataInput in) throws IOException {
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }

    @Override
    public int compareTo(Edge o) {
        return (departureNode.compareTo(o.departureNode) != 0)
            ? departureNode.compareTo(o.departureNode)
            : arrivalNode.compareTo(o.arrivalNode);
    }
}
```

1. **Specify how to read data in**
2. **Specify how to write data out**
3. **Define ordering of data**

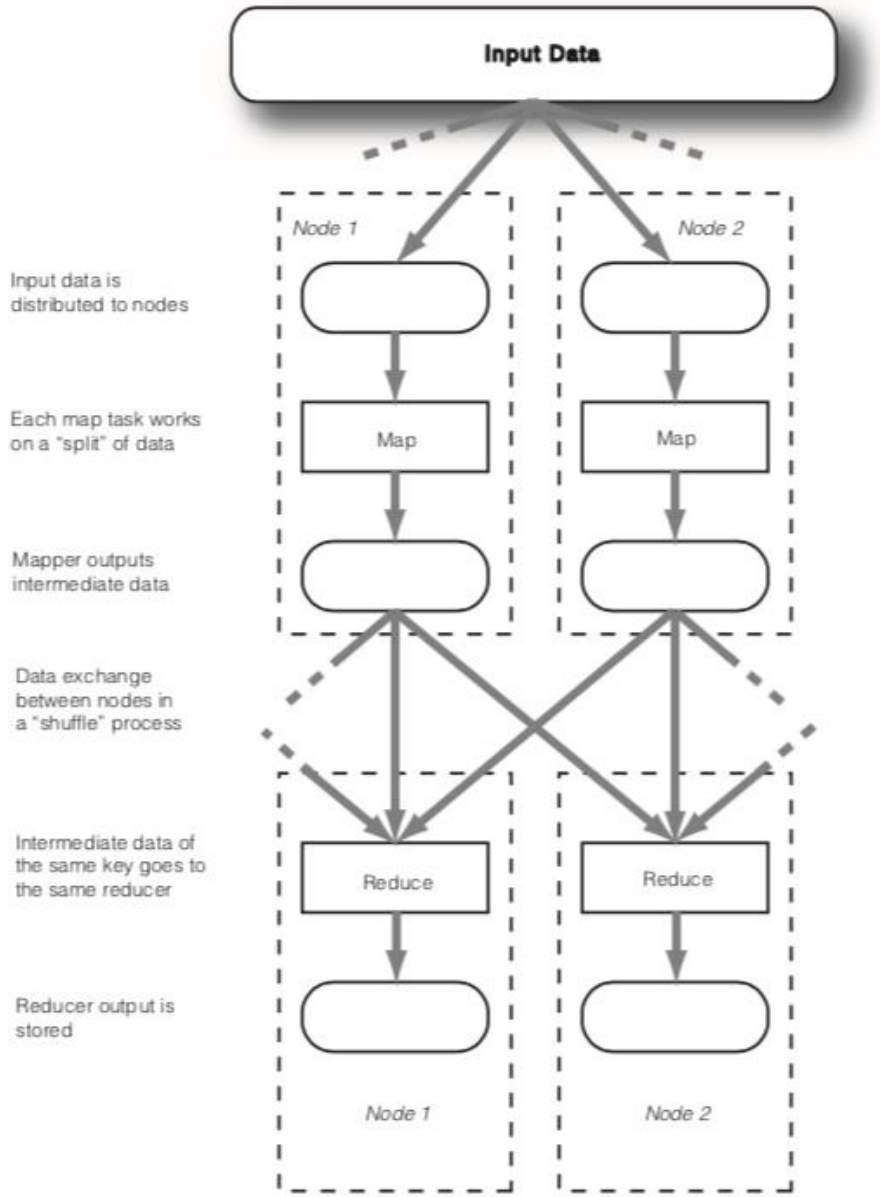

Figure 3.1 **The general MapReduce data flow. Note that after distributing input data to different nodes, the only time nodes communicate with each other is at the "shuffle" step. This restriction on communication greatly helps scalability.**

Mapper

- BEWARE: Old interface vs new Interface
  - A matter of package!!

- *Maps input key/value pairs to a set of intermediate key/value pairs*

```
import org.apache.hadoop.mapreduce.Mapper;
public class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
public void map(Object key, Text value, Context context)
```

- Es.
  - **public class** TokenCounterMapper extends Mapper<Object, Text, Text, IntWritable>

Some standard mappers

Class	Description
`IdentityMapper<K,V>`	Implements Mapper<K,V,K,V> and maps inputs directly to outputs
`InverseMapper<K,V>`	Implements Mapper<K,V,V,K> and reverses the key/value pair
`RegexMapper<K>`	Implements Mapper<K,Text,Text,LongWritable> and generates a (match, 1) pair for every regular expression match
`TokenCountMapper<K>`	Implements Mapper<K,Text,Text,LongWritable> and generates a (token, 1) pair when the input value is tokenized

Reducer

- *Reduces a set of intermediate values which share a key to a smaller set of values*
  - *Shuffle & Sort*
    - *copies the sorted output from each Mapper using HTTP across the network*
    - *merge sorts Reducer inputs by keys*
  - *SecondarySort*
    - *extend the key with the secondary key and define a grouping comparator)*
  - *The output of the Reducer is not re-sorted*

```
import org.apache.hadoop.mapreduce.Reducer;
public clas Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
public void reduce(Text key, Iterable<IntWritable> values,
                                    Context context)
```

- Es.
  - **public class** IntSumReducer<Key> extends Reducer<Key,IntWritable, Key,IntWritable>

Some standard reducers

Class	Description
`IdentityReducer<K,V>`	Implements Reducer<K,V,K,V> and maps inputs directly to outputs
`LongSumReducer<K>`	Implements Reducer<K,LongWritable,K,LongWritable> and determines the sum of all values corresponding to the given key

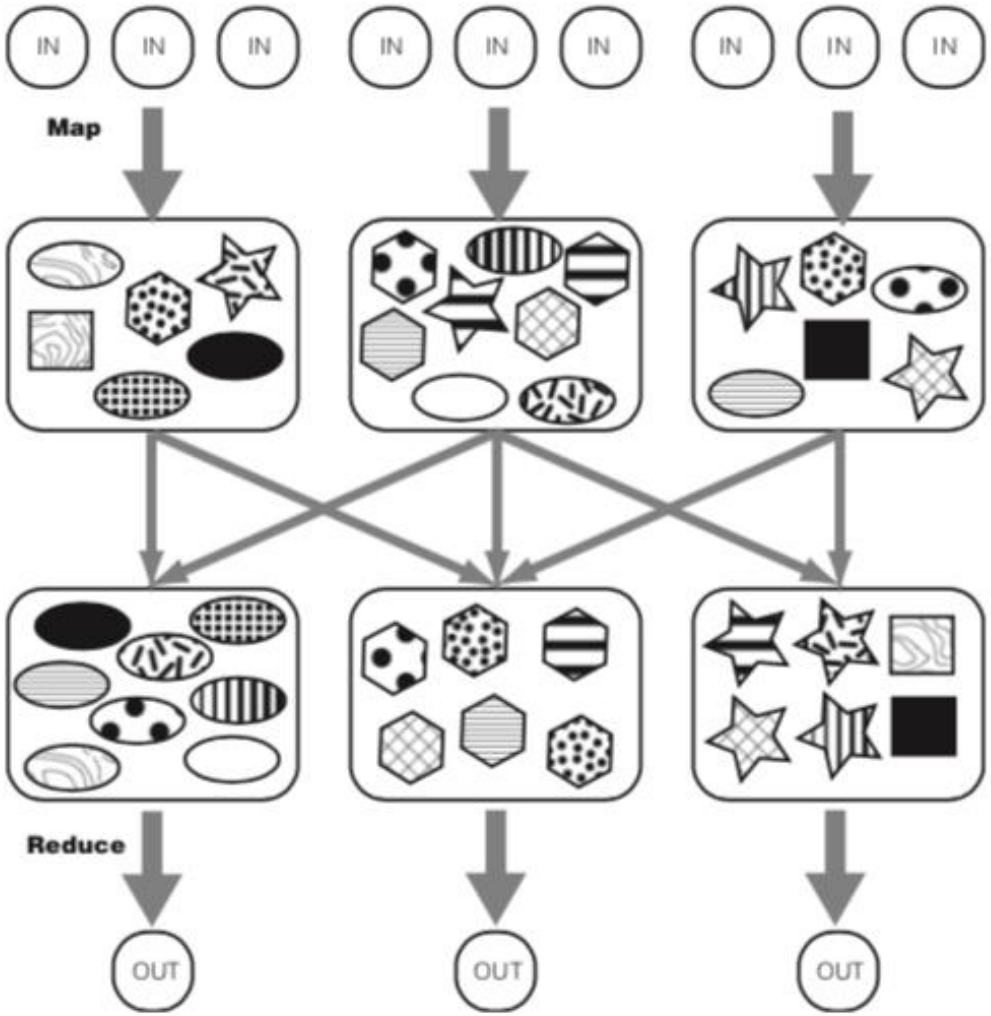

Figure 3.2 The MapReduce data flow, with an emphasis on partitioning and shuffling. Each icon is a key/value pair. The shapes represents keys, whereas the inner patterns represent values. After shuffling, all icons of the same shape (key) are in the same reducer. Different keys can go to the same reducer, as seen in the rightmost reducer. The partitioner decides which key goes where. Note that the leftmost reducer has more load due to more data under the "ellipse" key.

Combiner: a "local" reducer

- Perform a “local reduce” before we distribute the mapper results
  - Reduce network traffic and increase performance
  - Load Balancing: avoid overwhelming a single reducer
- Programmatically, the combiner implements Reducer
  - The very same interface of a reducer
- A combiner doesn’t necessarily improve performance
  - But might provide huge gains!
- Not always applicable (more on this later)

Partitioner

- You often need multiple reducers
- The partitioner
  - Determines where to send a (key/value) pair outputted by a mapper
  - Partitioner controls the partitioning of the keys of the intermediate map-outputs.
    - key (or a subset of the key) used to derive the partition by a hash function
    - The total number of partitions is the same as the number of reduce tasks for the job. Note:
  - A Partitioner is created only when there are multiple reducers
- Hadoop default: HashPartitioner class
- You can provide your custom Partitioner!

public abstract class Partitioner<KEY,VALUE>

Input formats

- The **InputFormat** interface
  - Defines the way an input file is split up and read by Hadoop
  - Many implementations are available
  - Many new can be added
- InputFormat class
  - Describes the input-specification for a Map-Reduce job
  - Validate the input-specification of the job
  - Split-up the input file(s) into logical InputSplits, each of which is then assigned to an individual Mapper
    - RecordReader implementation to be used to glean input records from the logical InputSplit for processing by the Mapper

public abstract class InputFormat<K,V>

InputFormat	Description
`TextInputFormat`	Each line in the text files is a record. Key is the byte offset of the line, and value is the content of the line. key: LongWritable value: Text
`KeyValueTextInputFormat`	Each line in the text files is a record. The first separator character divides each line. Everything before the separator is the key, and everything after is the value. The separator is set by the key.value.separator.in.input.line property, and the default is the tab (\t) character. key: Text value: Text
`SequenceFileInputFormat<K,V>`	An InputFormat for reading in *sequence files*. Key and value are user defined. Sequence file is a Hadoop-specific compressed binary file format. It's optimized for passing data between the output of one MapReduce job to the input of some other MapReduce job. key: K (user defined) value: V (user defined)
`NLineInputFormat`	Same as TextInputFormat, but each split is guaranteed to have exactly *N* lines. The mapred.line.input.format.linespermap property, which defaults to one, sets *N*. key: LongWritable value: Text

Custom Input Formats

- The functions that InputFormat has to perform:
  - Identify all the files used as input data and divide them into input splits
    - Each map task is assigned one split
  - Provide an object (RecordReader) to iterate through records in a given split and
    - to parse each record into key and value of predefined types
  - In practice, a split usually ends up being the size of a block, which defaults to 64 MB in HDFS

Provide new RecordReaders

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

public class CustomRecordReader extends RecordReader {

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context) throws IOException, Int
        // TODO Auto-generated method stub

    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public Object getCurrentKey() throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public Object getCurrentValue() throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public float getProgress() throws IOException, InterruptedException {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public void close() throws IOException {
        // TODO Auto-generated method stub

    }

}
```

Output formats

- MapReduce output files using the OutputFormat class
  - Analogous to the InputFormat class
  - The output has no splits
  - Each reducer writes its output only to its own file.
    - in a common directory
    - typically named part-*nnnnn* (*nnnnn* is the ID of the reducer)
  - RecordWriter objects format the output

Some standard OutputFormats

OutputFormat	Description
`TextOutputFormat<K,V>`	Writes each record as a line of text. Keys and values are written as strings and separated by a tab (\t) character, which can be changed in the mapred.textoutputformat.separator property.
`SequenceFileOutputFormat<K,V>`	Writes the key/value pairs in Hadoop's proprietary sequence file format. Works in conjunction with SequenceFileInputFormat.
`NullOutputFormat<K,V>`	Outputs nothing.

Configuration

- Provides access to configuration parameters

public class Configuration
extends Object implements iterable<Map.Entry<String,String>>, Writable

- Configurations are specified by resources
  - A resource contains a set of name/value pairs as XML data.
  - Hadoop by default specifies two resources
    - core-default.xml: Read-only defaults for hadoop.
    - core-site.xml: Site-specific configuration for a given hadoop installation.
  - Applications may add additional resources

Job

- The user
  - Creates the application
  - Describes various facets of the job via Job
  - Submits the job and monitor its progress

public class Job
    extends JobContextImpl implements JobContext, AutoCloseable

- With an instance of Job, you can
  - Configure the job
    - Set job name
    - Set jar containing the code, and the main to run
    - Set Input/Output
    - Set mapper(s) and rediucer(s)
  - Submit and control its execution
    - job.waitForCompletion(**true**)

```
// Create a new Job
Job job = Job.getInstance();
job.setJarByClass(MyJob.class);

// Specify various job-specific parameters
job.setJobName("myjob");

job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));

job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);

// Submit the job, then poll for progress until the job is complete
job.waitForCompletion(true);
```

LET'S START

(open, configure and run examples on eclipse)

Required Software

- Eclipse
  - www.eclipse.org
- Hadoop
  - Download it from http://hadoop.apache.org/
  - Obtain everything it via Maven (https://mvnrepository.com/artifact/org.apache.hadoop)
- Small examples do not need a cluster
  - Running Hadoop in Standalone mode
- Better testing
  - Pseudo-distributed mode
  - All daemons in you computer
- Serious testing & production
  - Install it on a cluster

POM settings

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>it.unical.mat.bigdata</groupId>
<artifactId>Examples</artifactId>
<version>0.0.1-SNAPSHOT</version>


<dependencies>
<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-mapreduce-client-core</artifactId>
<version>3.2.0</version>
</dependency>
<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-common</artifactId>
<version>3.2.0</version>
</dependency>
<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-mapreduce-client-jobclient</artifactId>
<version>3.2.0</version>
</dependency>
</dependencies>
</project>
```