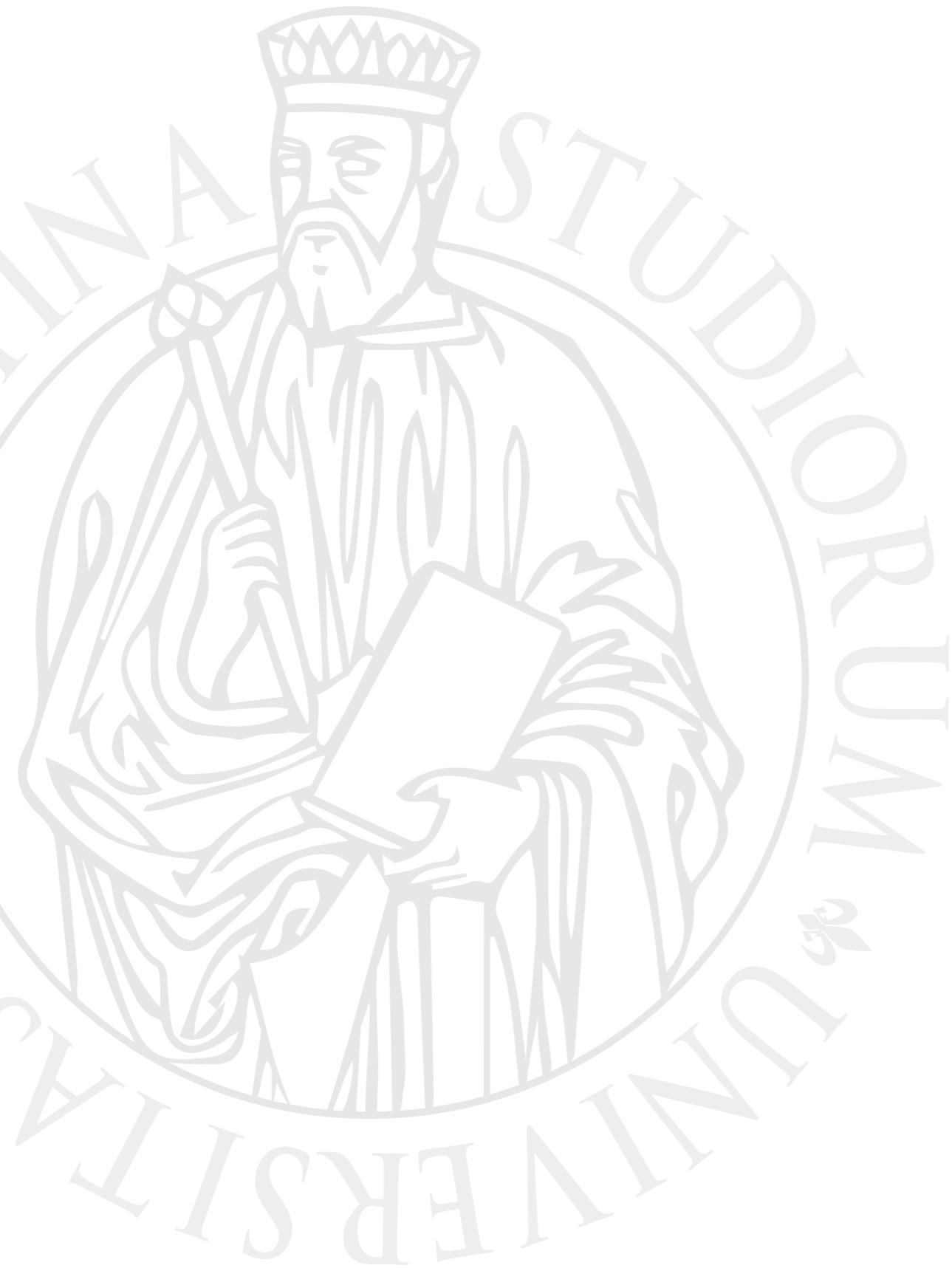




UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini





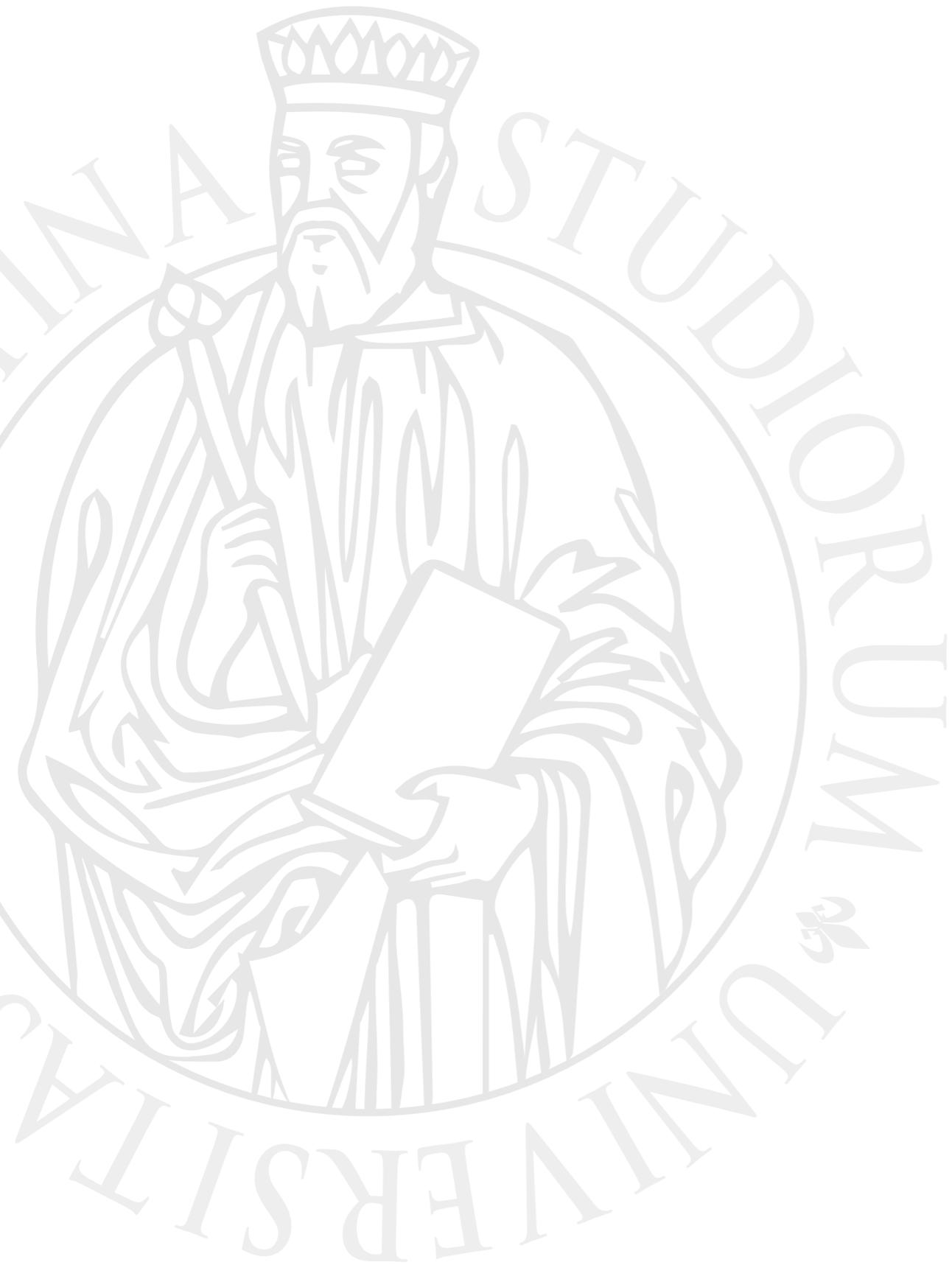
UNIVERSITÀ
DEGLI STUDI
FIRENZE



Apache Hadoop



UNIVERSITÀ
DEGLI STUDI
FIRENZE



Chaining jobs

Chaining MapReduce jobs

- Many complex tasks need to be broken down into simpler subtasks, each accomplished by an individual MapReduce job.
- You can chain MapReduce jobs to run sequentially, with the output of one MapReduce job being the input to the next, similarly to Unix pipes.
- As `JobClient.runJob()` blocks until the end of a job, chaining MapReduce jobs involves calling the driver of one MapReduce job after another.

Complex dependencies

- We may have a job that depends on the outcomes of two other preceding jobs. We can't use simple chaining
- In addition to holding job configuration information, Job also holds dependency information, specified through the `addDependingJob()` method. For Job objects `job1` and `job2`:
`job1.addDependingJob(job2)`
- Whereas Job objects store the configuration and dependency information, `JobControl` objects do the managing and monitoring of the job execution. You can add jobs to a `JobControl` object via the `addJob()` method. After adding all the jobs and dependencies, call `JobControl`'s `run()` method to spawn a thread to submit and monitor jobs for execution. `JobControl` has methods like `allFinished()` and `getFailedJobs()` to track the execution of various jobs within the batch.

Chaining pre/post-processing steps

- A lot of data processing tasks involve record-oriented preprocessing and postprocessing.
 - For example, in processing documents for information retrieval, you may have one step to remove stop words, and another step for stemming. Chaining MapReduce jobs where the reducer is an identity and chaining everything is impractical.
- It is better to write the mapper to call all the preprocessing steps beforehand and the reducer to call all the postprocessing steps afterward
- Hadoop has the `ChainMapper` and the `ChainReducer` classes to simplify the composition of pre- and postprocessing.
 - We can have this processing flow: `MAP+ | REDUCE | MAP*`

Chaining pre/post-processing steps

- A lot of data processing tasks involve record-oriented preprocessing and postprocessing.
 - For example, in processing documents for information retrieval, you may have one step to remove stop words, and another step for stemming. Chaining MapReduce jobs where the reducer is an identity and chaining everything is impractical.

The job runs multiple mappers in sequence to preprocess the data, and after running reduce it can optionally run multiple mappers in sequence to postprocess the data.

afterward

- Hadoop has the `ChainMapper` and the `ChainReducer` classes to simplify the composition of pre- and postprocessing.
 - We can have this processing flow: `MAP+ | REDUCE | MAP*`



ChainMapper and ChainReducer

```
Configuration conf = getConf();
JobConf job = new JobConf(conf);
job.setJobName("ChainJob");
job.setInputFormat(TextInputFormat.class);
job.setOutputFormat(TextOutputFormat.class);
FileInputFormat.setInputPaths(job, in);
FileOutputFormat.setOutputPath(job, out);

JobConf map1Conf = new JobConf(false);
ChainMapper.addMapper(job, Map1.class, LongWritable.class, Text.class, Text.class, Text.class,
                      true, map1Conf);
JobConf map2Conf = new JobConf(false);
ChainMapper.addMapper(job, Map2.class, Text.class, Text.class, LongWritable.class, Text.class,
                      true, map2Conf);
JobConf reduceConf = new JobConf(false);
ChainReducer.setReducer(job, Reduce.class, LongWritable.class, Text.class, Text.class,
                       Text.class, true, reduceConf);
JobConf map3Conf = new JobConf(false);
ChainReducer.addMapper(job, Map3.class, Text.class, Text.class, LongWritable.class, Text.class,
                       true, map3Conf);
JobConf map4Conf = new JobConf(false);
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class, LongWritable.class, Text.class,
                       true, map4Conf);
JobClient.runJob(job);
```

The chaining is Map1 | Map2 | Reduce | Map3 | Map4 and we can consider Map2 followed by Reduce the core of MapReduce job, while the other rappers are pre and post processing steps. Between Map2 and Reduce there's the usual shuffling. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.

ChainMapper and ChainReducer

```
Configuration conf = getConf();  
JobConf job = new JobConf(conf);  
job.setJobName("ChainJob");  
job.setInputFormat(TextInputFormat.class);
```

```
public static <K1,V1,K2,V2> void  
    addMapper(JobConf job,  
              Class<? extends Mapper<K1,V1,K2,V2>> klass,  
              Class<? extends K1> inputKeyClass,  
              Class<? extends V1> inputValueClass,  
              Class<? extends K2> outputKeyClass,  
              Class<? extends V2> outputValueClass,  
              boolean byValue,  
              JobConf mapperConf)
```

```
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class, LongWritable.class, Text.class,  
                       true, map4Conf);  
JobClient.runJob(job);
```

The chaining is Map1 | Map2 | Reduce | Map3 | Map4 and we can consider Map2 followed by Reduce the core of MapReduce job, while the other rappers are pre and post processing steps. Between Map2 and Reduce there's the usual shuffling. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.

ChainMapper and ChainReducer

```
Configuration conf = getConf();  
JobConf job = new JobConf(conf);  
job.setJobName("ChainJob");  
job.setInputFormat(TextInputFormat.class);
```

```
public static <K1,V1,K2,V2> void  
    addMapper(JobConf job, Global JobConf  
        Class<? extends Mapper<K1,V1,K2,V2>> klass,  
        Class<? extends K1> inputKeyClass,  
        Class<? extends V1> inputValueClass,  
        Class<? extends K2> outputKeyClass,  
        Class<? extends V2> outputValueClass,  
        boolean byValue,  
        JobConf mapperConf)
```

```
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class, LongWritable.class, Text.class,  
    true, map4Conf);  
JobClient.runJob(job);
```

The chaining is Map1 | Map2 | Reduce | Map3 | Map4 and we can consider Map2 followed by Reduce the core of MapReduce job, while the other rappers are pre and post processing steps. Between Map2 and Reduce there's the usual shuffling. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.

ChainMapper and ChainReducer

```
Configuration conf = getConf();  
JobConf job = new JobConf(conf);  
job.setJobName("ChainJob");  
job.setInputFormat(TextInputFormat.class);
```

```
public static <K1,V1,K2,V2> void  
addMapper(JobConf job, Global JobConf  
Class<? extends Mapper<K1,V1,K2,V2>> klass,  
Class<? extends K1> inputKeyClass,  
Class<? extends V1> inputValueClass,  
Class<? extends K2> outputKeyClass,  
Class<? extends V2> outputValueClass,  
boolean byValue,  
JobConf mapperConf) Local JobConf
```

```
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class, LongWritable.class, Text.class,  
true, map4Conf);  
JobClient.runJob(job);
```

The chaining is Map1 | Map2 | Reduce | Map3 | Map4 and we can consider Map2 followed by Reduce the core of MapReduce job, while the other rappers are pre and post processing steps. Between Map2 and Reduce there's the usual shuffling. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.

ChainMapper and ChainReducer

```
Configuration conf = getConf();  
JobConf job = new JobConf(conf);  
job.setJobName("ChainJob");  
job.setInputFormat(TextInputFormat.class);
```

```
public static <K1,V1,K2,V2> void  
addMapper(JobConf job, Global JobConf  
    Class<? extends Mapper<K1,V1,K2,V2>> klass, Mapper class  
    Class<? extends K1> inputKeyClass,  
    Class<? extends V1> inputValueClass,  
    Class<? extends K2> outputKeyClass,  
    Class<? extends V2> outputValueClass,  
    boolean byValue,  
    JobConf mapperConf) Local JobConf
```

```
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class, LongWritable.class, Text.class,  
    true, map4Conf);  
JobClient.runJob(job);
```

The chaining is Map1 | Map2 | Reduce | Map3 | Map4 and we can consider Map2 followed by Reduce the core of MapReduce job, while the other rappers are pre and post processing steps. Between Map2 and Reduce there's the usual shuffling. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.

ChainMapper and ChainReducer

```
Configuration conf = getConf();  
JobConf job = new JobConf(conf);  
job.setJobName("ChainJob");  
job.setInputFormat(TextInputFormat.class);
```

```
public static <K1,V1,K2,V2> void  
addMapper(JobConf job, Global JobConf  
Class<? extends Mapper<K1,V1,K2,V2>> klass, Mapper class  
Class<? extends K1> inputKeyClass,  
Class<? extends V1> inputValueClass,  
Class<? extends K2> outputKeyClass,  
Class<? extends V2> outputValueClass,  
boolean byValue,  
JobConf mapperConf) Local JobConf
```

input/output class
types of the Mapper
class

```
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class, LongWritable.class, Text.class,  
true, map4Conf);  
JobClient.runJob(job);
```

The chaining is Map1 | Map2 | Reduce | Map3 | Map4 and we can consider Map2 followed by Reduce the core of MapReduce job, while the other rappers are pre and post processing steps. Between Map2 and Reduce there's the usual shuffling. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.

In the standard Mapper model, the output key/value pairs are serialized and written to disk, this is considered to be passed by value, as a copy of the key/value pair is sent over. In the current case where we can chain one Mapper to another, we can execute the two in the same JVM thread. Therefore, it's possible for the key/value pairs to be passed by reference, where the output of the initial Mapper stays in place in memory and the following Mapper refers to it directly in the same memory location.

If you're sure that Map1's map() method doesn't use the content of k and v after output, or that Map2 doesn't change the value of its k and v input, you can achieve some performance gains by setting `byValue` to `false`. If you're not sure of the Mapper's internal code, it's best to play safe and let `byValue` be `true`, maintaining the pass-by-value model

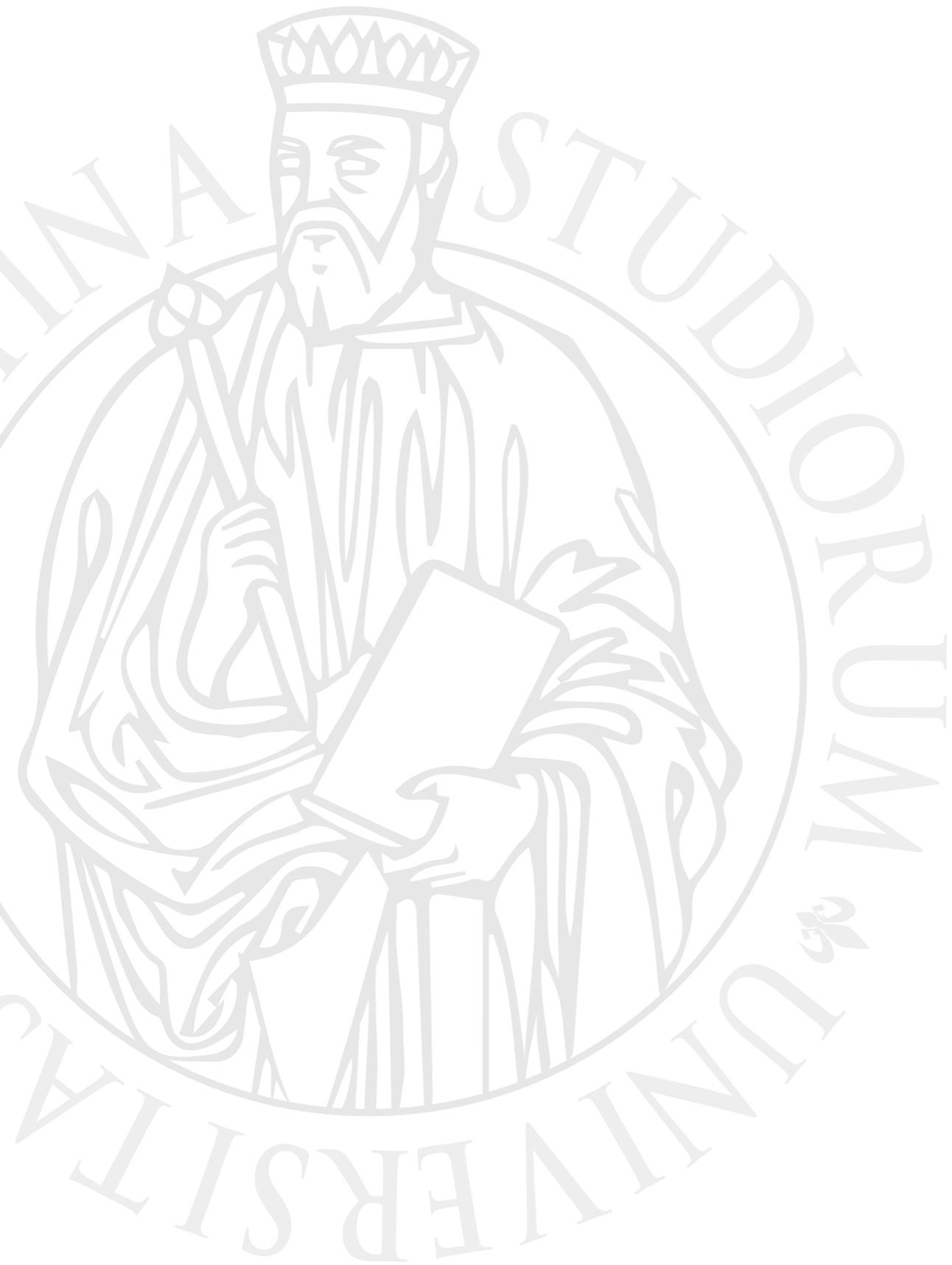
```
Class<? extends K2> outputKeyClass,  
Class<? extends V2> outputValueClass,  
boolean byValue,  
JobConf mapperConf) Local JobConf
```

```
ChainReducer.addMapper(job, Map4.class, LongWritable.class, Text.class, LongWritable.class, Text.class,  
                        true, map4Conf);  
JobClient.runJob(job);
```

The chaining is Map1 | Map2 | Reduce | Map3 | Map4 and we can consider Map2 followed by Reduce the core of MapReduce job, while the other mappers are pre and post processing steps. Between Map2 and Reduce there's the usual shuffling. Running all the pre- and postprocessing steps in a single job leaves no intermediate file and there's a dramatic reduction in I/O.



Joining data from different sources



Joins

- It may happen the necessity to pull in data from different sources.
- MapReduce can perform joins between large datasets, but writing the code to do joins from scratch is fairly involved.
 - Rather than writing MapReduce programs, you might consider using a higher-level framework such as Hive, in which join operations are a core part of the implementation.
- If the join is performed by the mapper, it is called a **map-side join**, whereas if it is performed by the reducer it is called a **reduce-side join**.

Map-side joins

- A map-side join between large inputs works by performing the join before the data reaches the map function.
- The inputs to each map must be partitioned and sorted in a particular way.
- Each input dataset must be divided into the same number of partitions, and it must be sorted by the same key (the join key) in each source.
- All the records for a particular key must reside in the same partition.

Reduce-side joins

- A reduce-side join is more general than a map-side join, in that the input datasets don't have to be structured in any particular way, but it is less efficient because both datasets have to go through the MapReduce shuffle.
- The mapper tags each record with its source and uses the join key as the map output key, so that the records with the same key are brought together in the reducer.
- We need to use two techniques:
 - Multiple inputs
 - Secondary sort

Multiple Inputs

- The input sources for different datasets generally have different formats, so it is very convenient to use the `MultipleInputs` class to separate the logic for parsing and tagging each source.
- `MultipleInputs` class allows you to specify which `InputFormat` and `Mapper` to use on a per-path basis.



Secondary sort

- The reducer will see the records from both sources that have the same key, but they are not guaranteed to be in any particular order. However, to perform the join, it is important to have the data from one source before another.
- We do not want to buffer data because it could be too big
- The goal of secondary sort is to perform an additional sort of keys, beyond that of Hadoop, that follows our criteria

Secondary sort

The result will be the records from both sources

There is a recipe to perform secondary sort:

- Make the key a composite of the natural key and a secondary key we need e.g. the value.
 - The sort comparator should order by the composite key, that is, the natural key and secondary key.
 - The partitioner and grouping comparator for the composite key should consider only the natural key for partitioning and grouping.
-
- **The goal of secondary sort is to perform an additional sort of keys, beyond that of Hadoop, that follows our criteria**



Example

- Let us consider a MapReduce job that joins stock symbols of companies with their trade data
- Let's assume we have a `TextPair` class that implements `WritableComparable` and models couples of Text strings.
This class allows secondary sort.



Stock symbol mapper

```
public class JoinStockNMapper
extends Mapper<LongWritable, Text, TextPair, Text> {

    private SymbolParser parser = new SymbolParser();

    @Override

    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {

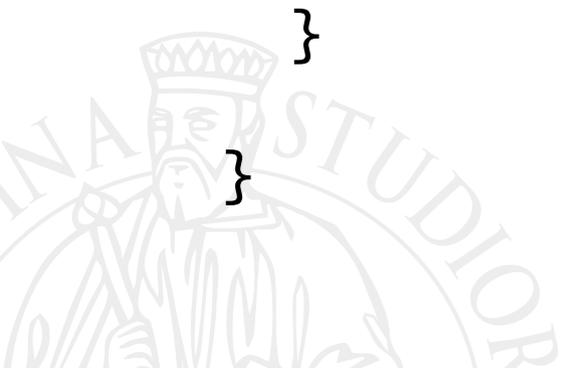
        if (parser.parse(value)) {

            context.write(new TextPair(parser.getSymbolId(), "0"),
                new Text(parser.getSymbolName()));

        }

    }

}
```



Stock symbol mapper

```
public class JoinStockNMapper
extends Mapper<LongWritable, Text, TextPair, Text> {

    private SymbolParser parser = new SymbolParser();

    @Override

    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {

        if (parser.parse(value)) {

            context.write(new TextPair(parser.getSymbolId(), "0"),
                new Text(parser.getSymbolName()));

        }

    }

}
```

This data comes from table "0", i.e. trade symbol table

Stock symbol mapper

```
public class JoinStockNMapper
extends Mapper<LongWritable, Text, TextPair, Text> {

    private SymbolParser parser = new SymbolParser();

    @Override

    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {

        if (parser.parse(value)) {

            context.write(new TextPair(parser.getSymbolId(), "0"),
                new Text(parser.getSymbolName()));

        }

    }

}
```

Input: AAPL, Apple

Output: <AAPL, 0,> , Apple

This data comes from table "0", i.e. trade symbol table



Trade Closing Mapper

```
public class JoinTradeMapper
extends Mapper<LongWritable, Text, TextPair, Text> {

    private TradePriceParser parser = new TradePriceParser();

    @Override

    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {

        parser.parse(value);

        context.write(new TextPair(parser.getSymbolId(), "1"),
            new Text(parser.getClosingValue()+"\t"+
                parser.getTimestamp()));

    }
}
```



Trade Closing Mapper

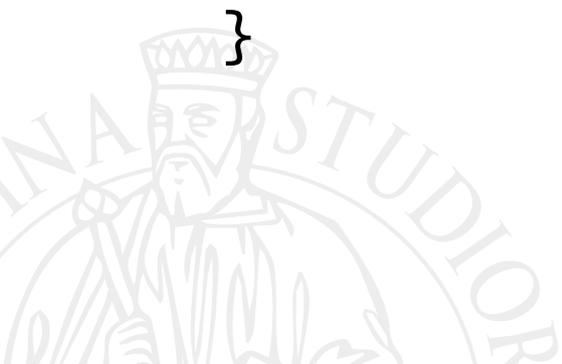
```
public class JoinTradeMapper
extends Mapper<LongWritable, Text, TextPair, Text> {

    private TradePriceParser parser = new TradePriceParser();

    @Override

    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {
        parser.parse(value);
        context.write(new TextPair(parser.getSymbolId(), "1"),
            new Text(parser.getClosingValue()+"\t"+
                parser.getTimestamp()));
    }
}
```

This data comes from table "1", i.e. trade prices table



Trade Closing Mapper

```
public class JoinTradeMapper
extends Mapper<LongWritable, Text, TextPair, Text> {

    private TradePriceParser parser = new TradePriceParser();

    @Override

    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {
        parser.parse(value);
        context.write(new TextPair(parser.getSymbolId(), "1"),
            new Text(parser.getClosingValue()+"\t"+
                parser.getTimestamp()));
    }
}
```

This data comes from table "1", i.e. trade prices table

Input: AAPL, 100€, 2016-12-01
Output: <AAPL, 1>, 100€ \t 2016-12-01



Join Reducer

```
public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {  
  
    @Override  
  
    protected void reduce(TextPair key, Iterable<Text> values, Context context)  
        throws IOException, InterruptedException {  
  
        Iterator<Text> iter = values.iterator();  
  
        Text symbolName = new Text(iter.next());  
  
        while (iter.hasNext()) {  
  
            Text record = iter.next();  
  
            Text outValue = new Text(symbolName.toString() + "\t" + record.toString());  
  
            context.write(key.getFirst(), outValue);  
  
        }  
  
    }  
  
}
```



Join Reducer

```
public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {
```

```
    @Override
```

```
    protected void reduce(TextPair key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
```

```
        Iterator<Text> iter = values.iterator();
```

```
        Text symbolName = new Text(iter.next());
```

```
        while (iter.hasNext()) {
```

```
            Text record = iter.next();
```

```
            Text outValue = new Text(symbolName.toString() + "\t" + record.toString());
```

```
            context.write(key.getFirst(), outValue);
```

```
        }
```

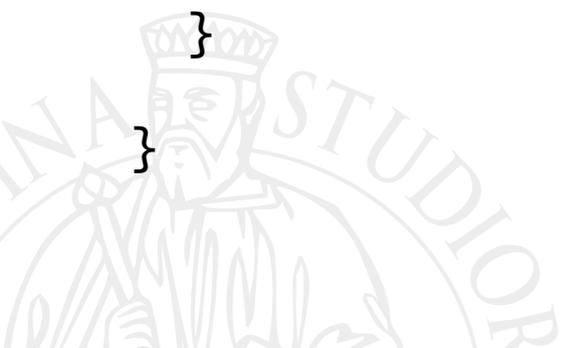
The reducer knows that it will receive the stock symbol record first, so it extracts its name from the value and writes it out as a part of every output record



The code assumes that every trade price ID in the trade records has exactly one matching record in the stock names dataset. If this were not the case, we would need to generalize the code to put the tag into the value objects, by using another TextPair. The reduce() method would then be able to tell which entries were station names and detect (and handle) missing or duplicate entries before processing the weather records.

```
public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {  
  
    @Override  
  
    protected void reduce(TextPair key, Iterable<Text> values, Context context)  
        throws IOException, InterruptedException {  
  
        Iterator<Text> iter = values.iterator();  
  
        Text symbolName = new Text(iter.next());  
  
        while (iter.hasNext()) {  
  
            Text record = iter.next();  
  
            Text outValue = new Text(symbolName.toString() + "\t" + record.toString());  
  
            context.write(key.getFirst(), outValue);  
  
        }  
  
    }  
  
}
```

The reducer knows that it will receive the stock symbol record first, so it extracts its name from the value and writes it out as a part of every output record



The code assumes that every trade price ID in the trade records has exactly one matching record in the stock names dataset. If this were not the case, we would need to generalize the code to put the tag into the value objects, by using another TextPair. The reduce() method would then be able to tell which entries were station names and detect (and handle) missing or duplicate entries before processing the weather records.

```
public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {  
  
    @Override  
  
    protected void reduce(TextPair key, Iterable<Text> values, Context context)  
        throws IOException, InterruptedException {  
  
        Iterator<Text> iter = values.iterator();  
  
        Text symbolName = new Text(iter.next());  
  
        while (iter.hasNext()) {  
  
            Text record = iter.next();  
  
            Text outValue = new Text(symbolName.toString() + "\t" + record.toString());  
  
            context.write(key.getFirst(), outValue);  
  
        }  
  
    }  
  
}
```

The reducer knows that it will receive the stock symbol record first, so it extracts its name from the value and writes it out as a part of every output record

```
Input:  
  
<AAPL, 0>, Apple  
<AAPL, 1>, 100€ \t 2016-1201  
  
Output:  
AAPL, Apple \t 100 \t 2016-12-01
```





Driver

```
public class JoinTradePriceRecordWithTradeName extends Configured implements Tool {

    public static class KeyPartitioner extends Partitioner<TextPair, Text> {
        @Override
        public int getPartition(TextPair key, Text value, int numPartitions) {
            return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;
        }
    }

    @Override
    public int run(String[] args) throws Exception {
        Job job = new Job(getConf(), "Join stock name records with trade prices"); job.setJarByClass(getClass());
        Path stocknInputPath = new Path(args[0]); Path tradepInputPath = new Path(args[1]);
        Path outputPath = new Path(args[2]);

        MultipleInputs.addInputPath(job, stocknInputPath, TextInputFormat.class, JoinStockNMapper.class);
        MultipleInputs.addInputPath(job, tradepInputPath, TextInputFormat.class, JoinTradeMapper.class);
        FileOutputFormat.setOutputPath(job, outputPath);

        job.setPartitionerClass(KeyPartitioner.class);
        job.setGroupingComparatorClass(TextPair.FirstComparator.class);
        job.setMapOutputKeyClass(TextPair.class);
        job.setReducerClass(JoinReducer.class);
        job.setOutputKeyClass(Text.class);
        return job.waitForCompletion(true) ? 0 : 1;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new JoinRecordWithStationName(), args);
        System.exit(exitCode);
    }
}
```



Driver

```
public class JoinTradePriceRecordWithTradeName extends Configured implements Tool {  
  
    public static class KeyPartitioner extends Partitioner<TextPair, Text> {  
        @Override  
        public int getPartition(TextPair key, Text value, int numPartitions) {  
            return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;  
        }  
    }  
}
```

The essential point is that we partition and group on the first part of the key, the station ID, which we do with a custom Partitioner (**KeyPartitioner**) and a custom group comparator, **FirstComparator** (from **TextPair**).

```
MultipleInputs.addInputPath(job, stocknInputPath, TextInputFormat.class, JoinStockNMapper.class);  
MultipleInputs.addInputPath(job, tradepInputPath, TextInputFormat.class, JoinTradeMapper.class);  
FileOutputFormat.setOutputPath(job, outputPath);
```

```
job.setPartitionerClass(KeyPartitioner.class);  
job.setGroupingComparatorClass(TextPair.FirstComparator.class);  
job.setMapOutputKeyClass(TextPair.class);  
job.setReducerClass(JoinReducer.class);  
job.setOutputKeyClass(Text.class);  
return job.waitForCompletion(true) ? 0 : 1;  
}
```

```
public static void main(String[] args) throws Exception {  
    int exitCode = ToolRunner.run(new JoinRecordWithStationName(), args);  
    System.exit(exitCode);  
}
```

Books

- Learning Hadoop 2, Garry Turkington and Gabriele Modena, Packt Publishing - Chapt. 2
- Hadoop The Definitive Guide, Tom White, O'Reilly - Chapt. 8
- Hadoop in Action, Chuck Lam, Manning - Chapt. 5