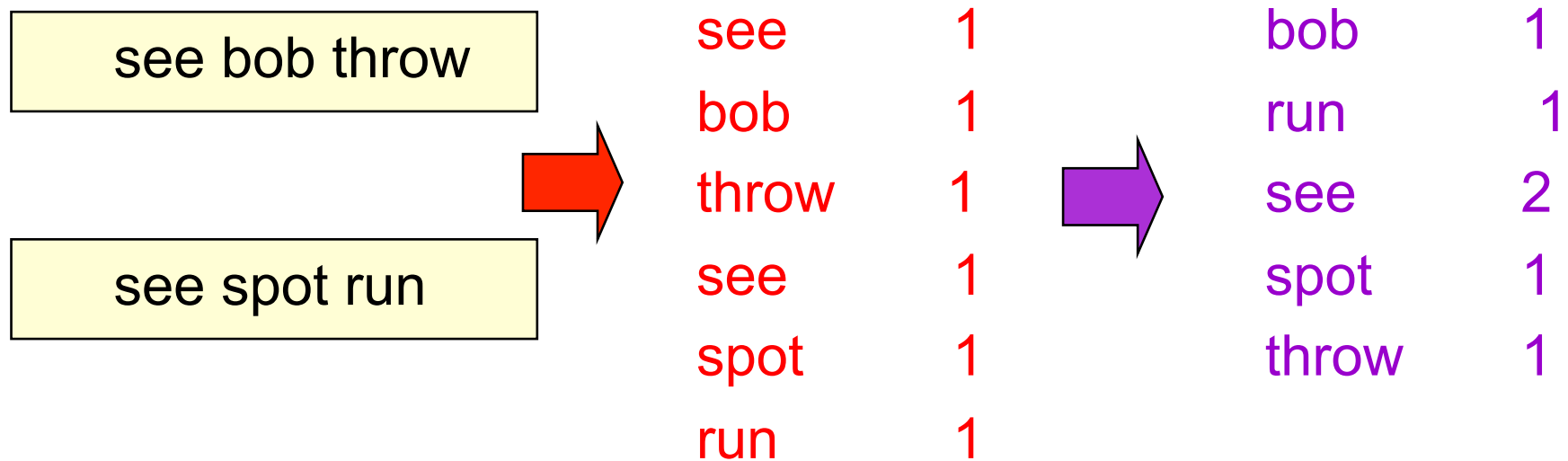


Introduction to Hadoop

Slides compiled from:

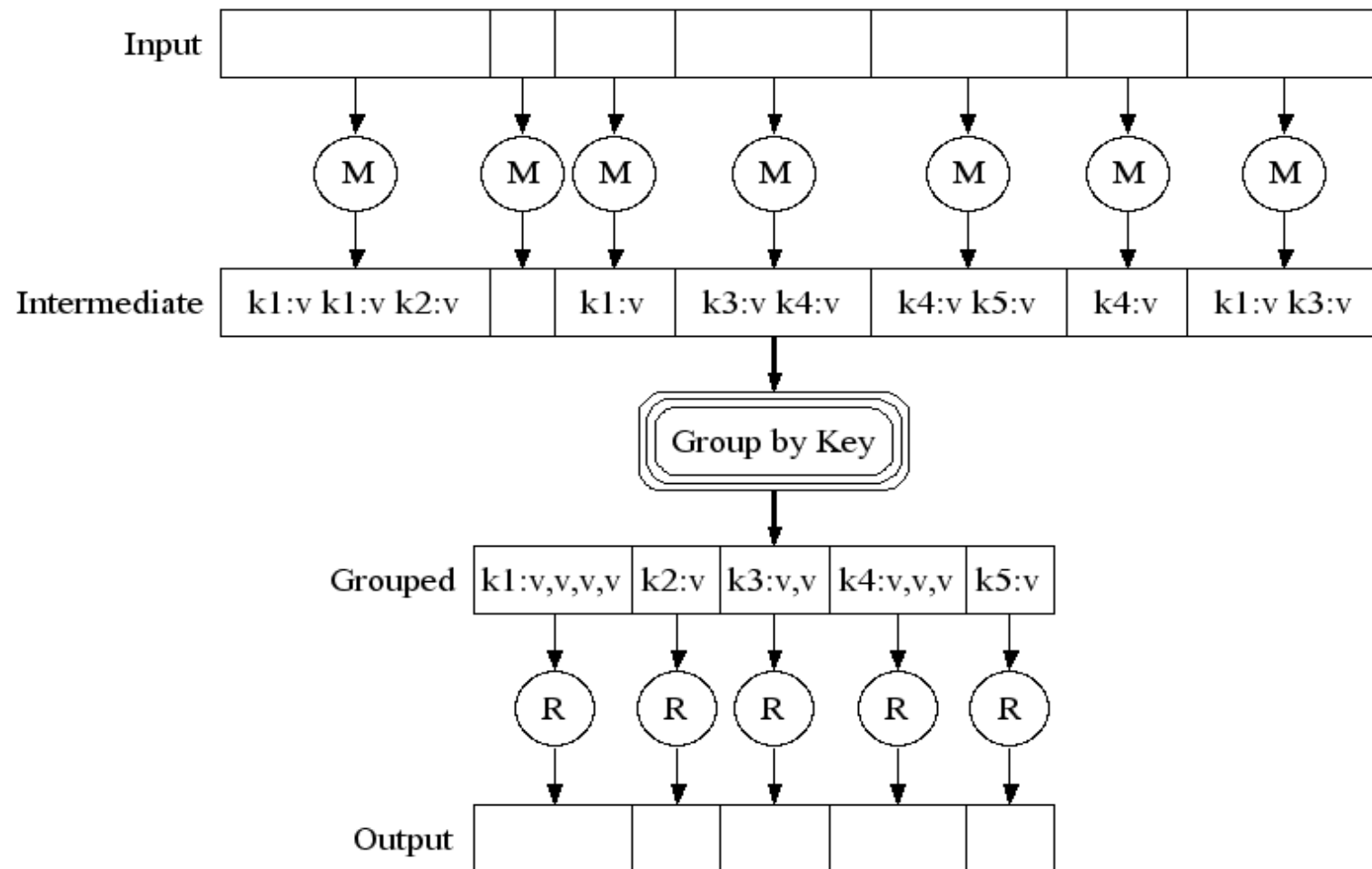
- Introduction to MapReduce and Hadoop
 - Shivrath Babu
- Experiences with Hadoop and MapReduce
 - Jian Wen

Word Count over a Given Set of Web Pages

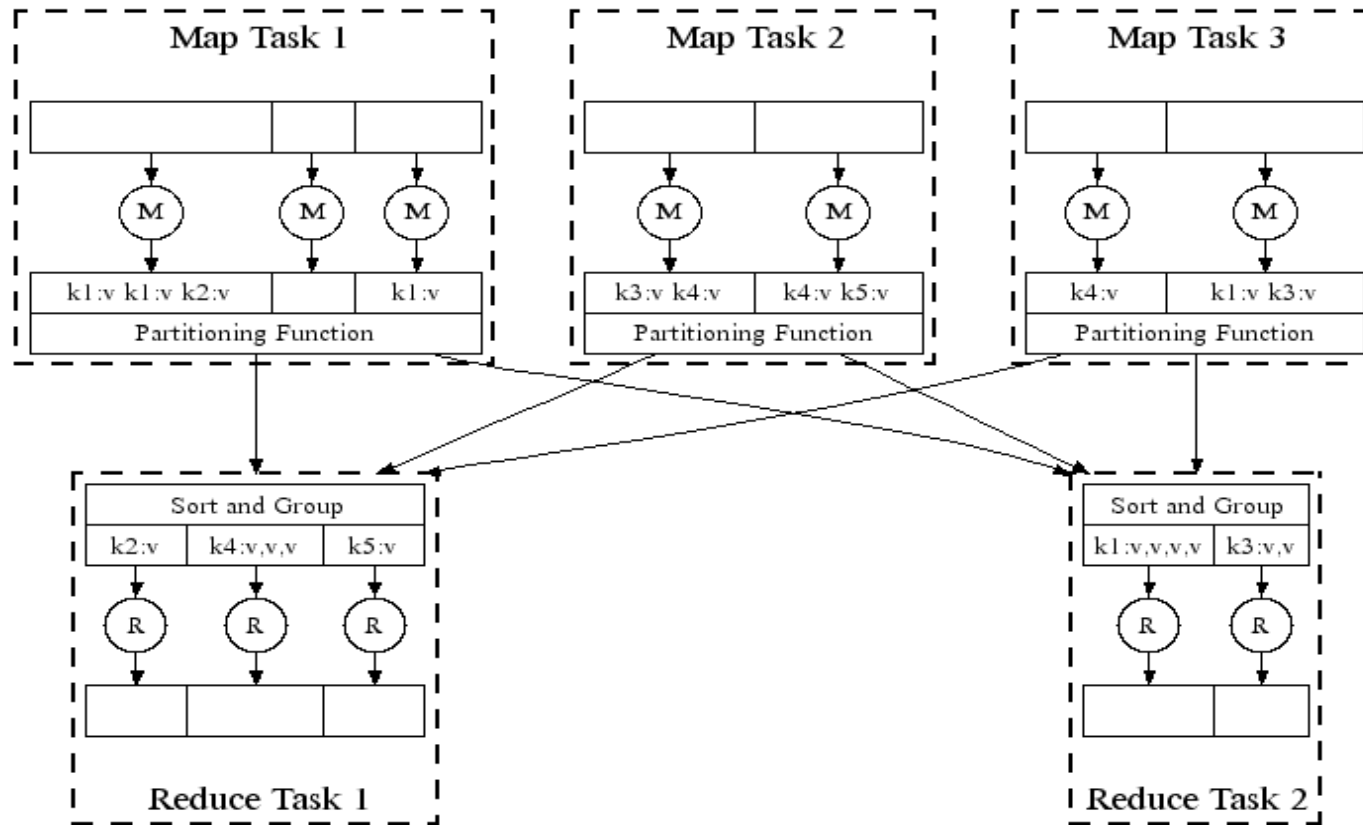


Can we do word count in parallel?

The MapReduce Framework (pioneered by Google)



Automatic Parallel Execution in MapReduce (Google)



Handles failures automatically, e.g., restarts tasks if a node fails; runs multiples copies of the same task to avoid a slow task slowing down the whole job

MapReduce in Hadoop (1)

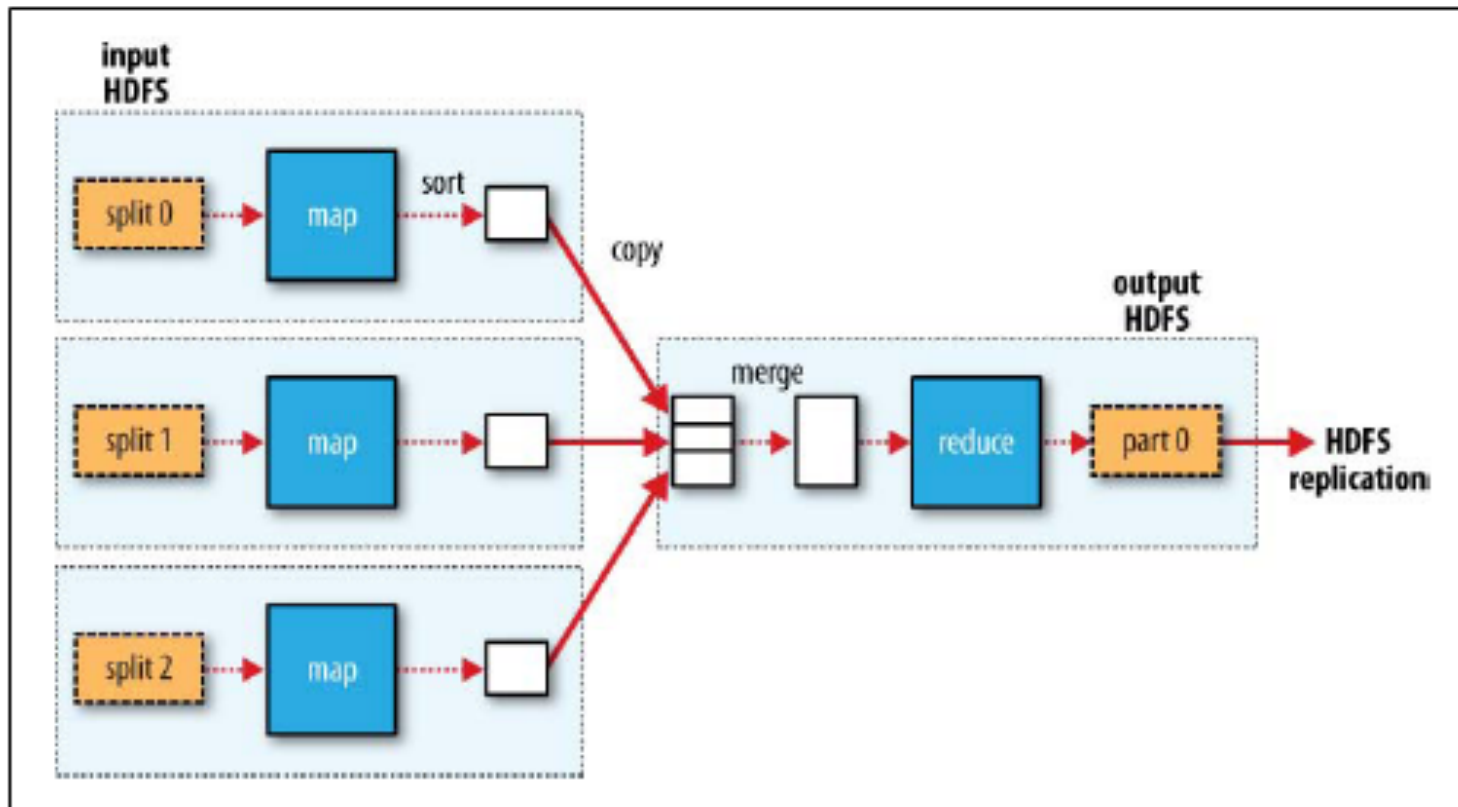


Figure 2-2. MapReduce data flow with a single reduce task

MapReduce in Hadoop (2)

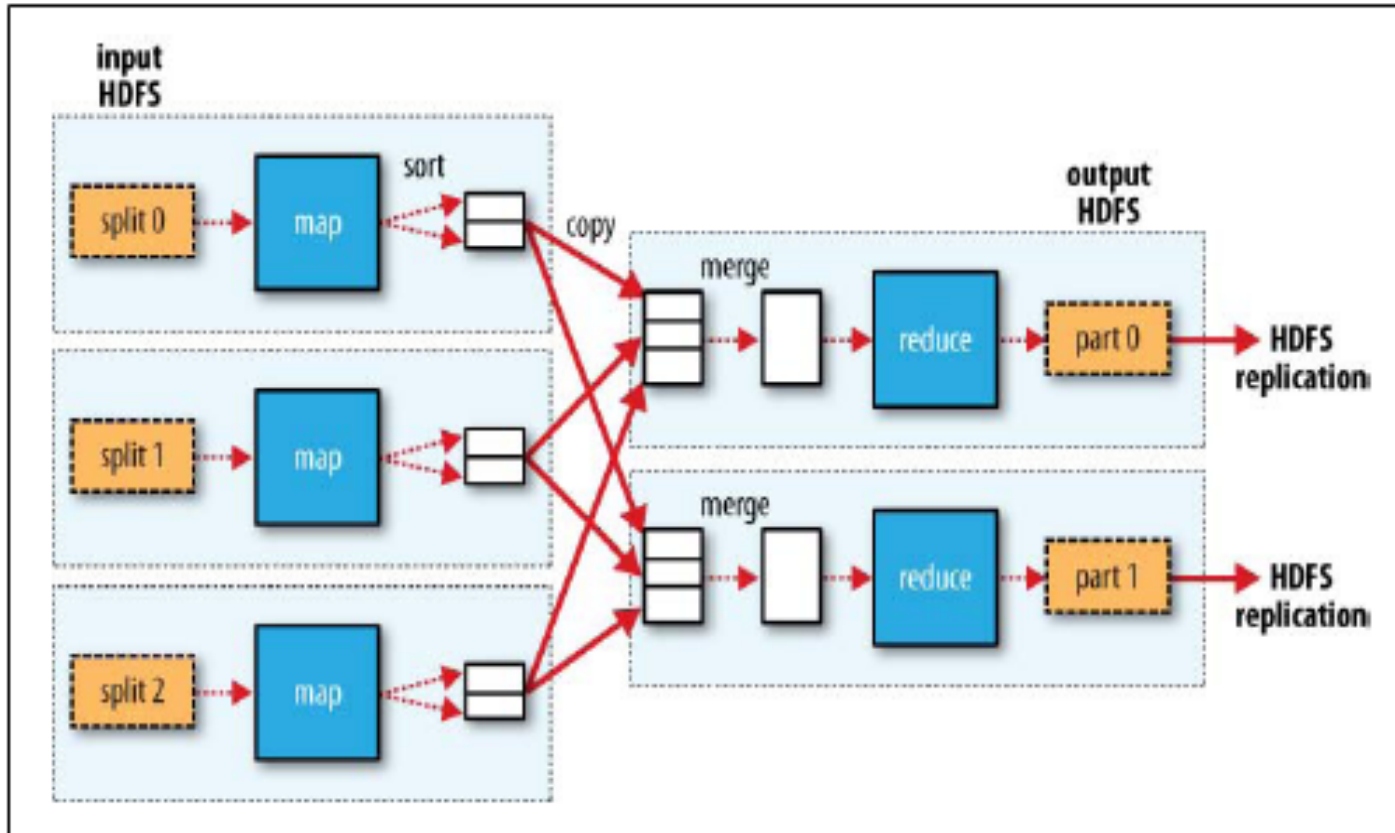


Figure 2-3. MapReduce data flow with multiple reduce tasks

MapReduce in Hadoop (3)

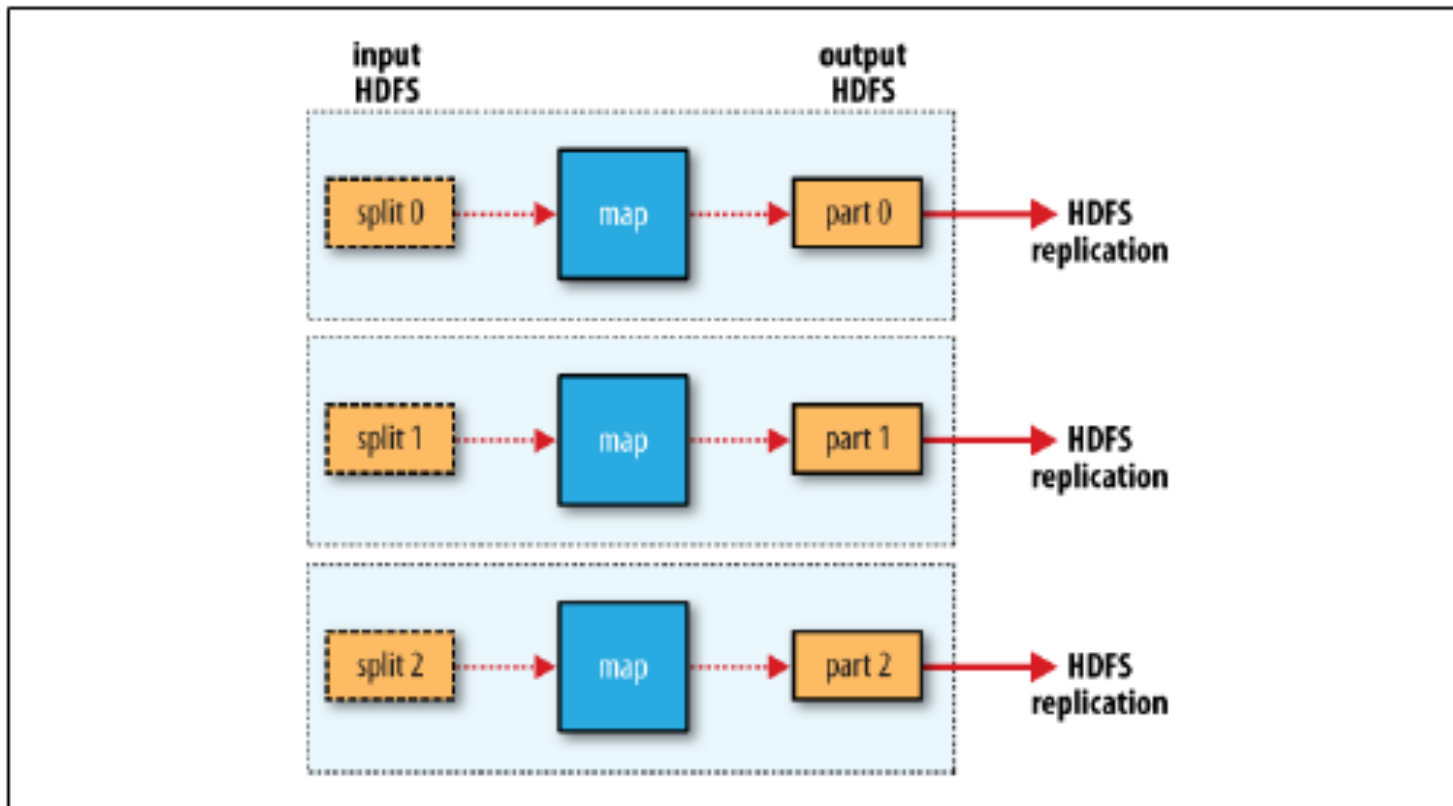
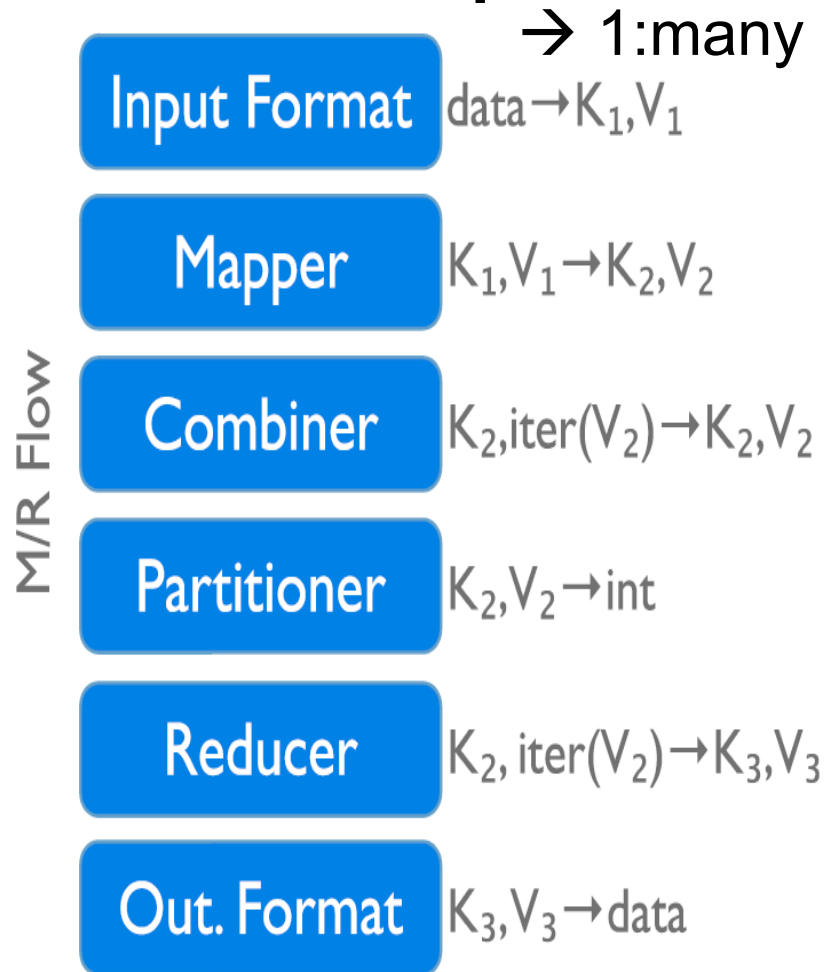


Figure 2-4. MapReduce data flow with no reduce tasks

Data Flow in a MapReduce Program in Hadoop

- InputFormat
- Map function
- Partitioner
- Sorting & Merging
- Combiner
- Shuffling
- Merging
- Reduce function
- OutputFormat



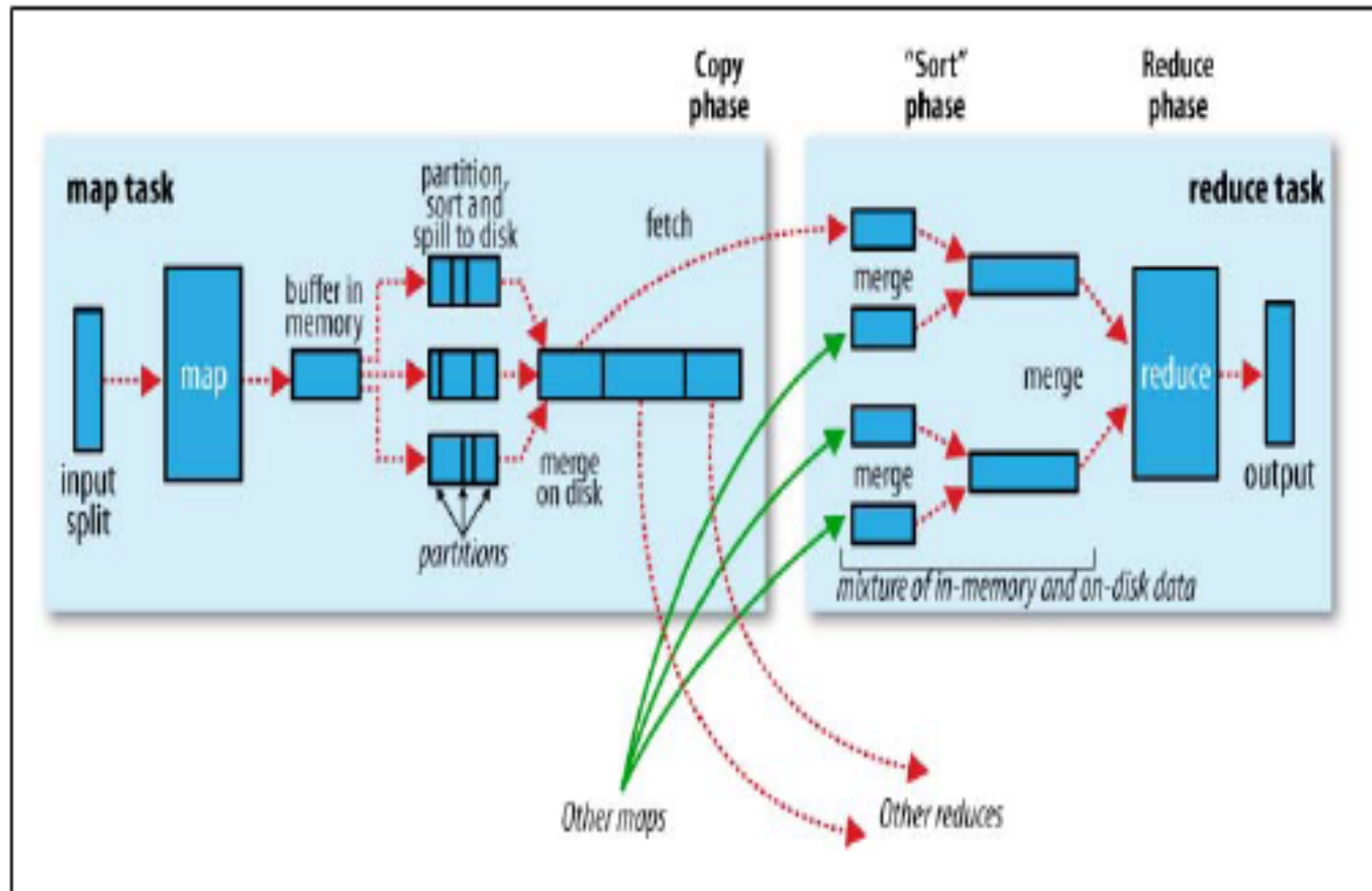


Figure 6-4. Shuffle and sort in MapReduce

Lifecycle of a MapReduce Job

The image shows a Java IDE window with the following code and annotations:

```
public class WordCount {  
  
    public static class Map extends MapReduceBase implements  
        Mapper<LongWritable, Text, Text, IntWritable> {  
        private final static IntWritable one = new IntWritable(1);  
        private Text word = new Text();  
  
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>  
            output, Reporter reporter) throws IOException {  
            String line = value.toString();  
            StringTokenizer tokenizer = new StringTokenizer(line);  
            while (tokenizer.hasMoreTokens()) {  
                word.set(tokenizer.nextToken());  
                output.collect(word, one);  
            }  
        }  
    }  
  
    public static class Reduce extends MapReduceBase implements  
        Reducer<Text, IntWritable, Text, IntWritable> {  
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,  
            IntWritable> output, Reporter reporter) throws IOException {  
            int sum = 0;  
            while (values.hasNext()) { sum += values.next().get(); }  
            output.collect(key, new IntWritable(sum));  
        }  
    }  
  
    public static void main(String[] args) throws Exception {  
        JobConf conf = new JobConf(WordCount.class);  
        conf.setJobName("wordcount");  
        conf.setOutputKeyClass(Text.class);  
        conf.setOutputValueClass(IntWritable.class);  
        conf.setMapperClass(Map.class);  
        conf.setCombinerClass(Reduce.class);  
        conf.setReducerClass(Reduce.class);  
        conf.setInputFormat(TextInputFormat.class);  
        conf.setOutputFormat(TextOutputFormat.class);  
        FileInputFormat.setInputPaths(conf, new Path(args[0]));  
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
  
        JobClient.runJob(conf);  
    }  
}
```

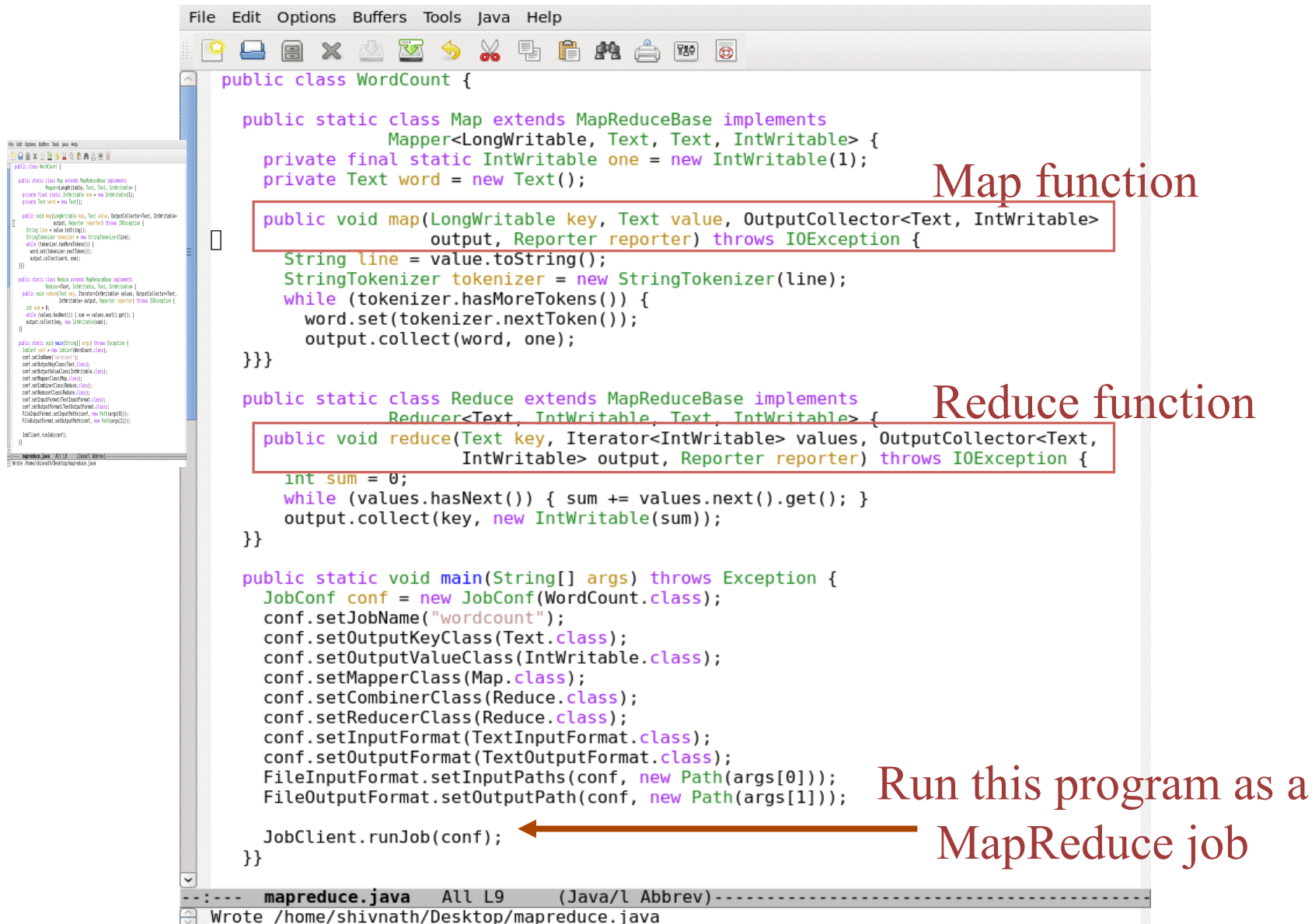
Map function: A red box highlights the `map` method in the `Map` class, with a red arrow pointing to it from the label.

Reduce function: A red box highlights the `reduce` method in the `Reduce` class, with a red arrow pointing to it from the label.

Run this program as a MapReduce job: A red arrow points from this text to the `JobClient.runJob(conf);` line in the `main` method.

The IDE status bar at the bottom shows: `--:--- mapreduce.java All L9 (Java/l Abbrev) -----` and `Wrote /home/shivnath/Desktop/mapreduce.java`.

Lifecycle of a MapReduce Job



```
public class WordCount {  
  
    public static class Map extends MapReduceBase implements  
        Mapper<LongWritable, Text, Text, IntWritable> {  
        private final static IntWritable one = new IntWritable(1);  
        private Text word = new Text();  
  
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>  
            output, Reporter reporter) throws IOException {  
            String line = value.toString();  
            StringTokenizer tokenizer = new StringTokenizer(line);  
            while (tokenizer.hasMoreTokens()) {  
                word.set(tokenizer.nextToken());  
                output.collect(word, one);  
            }  
        }  
  
        public static class Reduce extends MapReduceBase implements  
            Reducer<Text, IntWritable, Text, IntWritable> {  
            public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,  
                IntWritable> output, Reporter reporter) throws IOException {  
                int sum = 0;  
                while (values.hasNext()) { sum += values.next().get(); }  
                output.collect(key, new IntWritable(sum));  
            }  
        }  
  
        public static void main(String[] args) throws Exception {  
            JobConf conf = new JobConf(WordCount.class);  
            conf.setJobName("wordcount");  
            conf.setOutputKeyClass(Text.class);  
            conf.setOutputValueClass(IntWritable.class);  
            conf.setMapperClass(Map.class);  
            conf.setCombinerClass(Reduce.class);  
            conf.setReducerClass(Reduce.class);  
            conf.setInputFormat(TextInputFormat.class);  
            conf.setOutputFormat(TextOutputFormat.class);  
            FileInputFormat.setInputPaths(conf, new Path(args[0]));  
            FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
  
            JobClient.runJob(conf);  
        }  
    }  
}
```

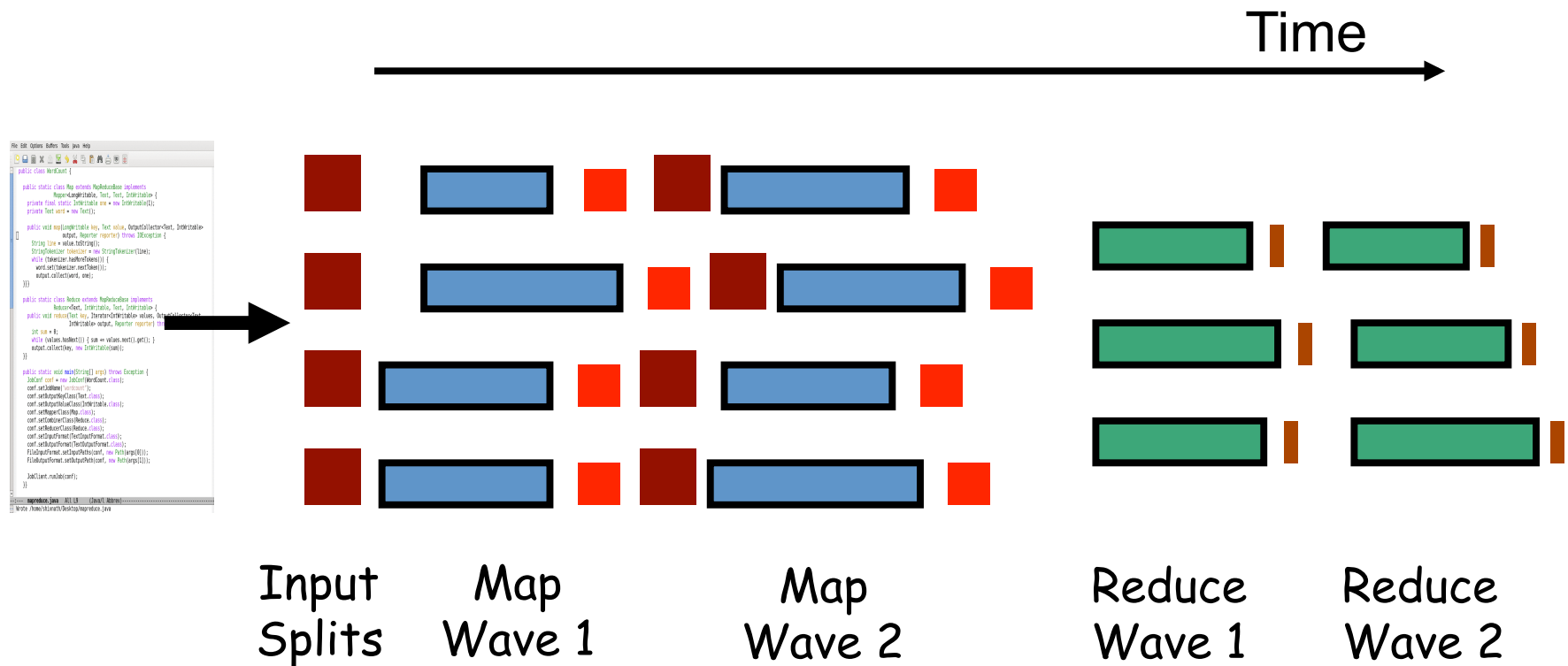
Map function

Reduce function

Run this program as a MapReduce job

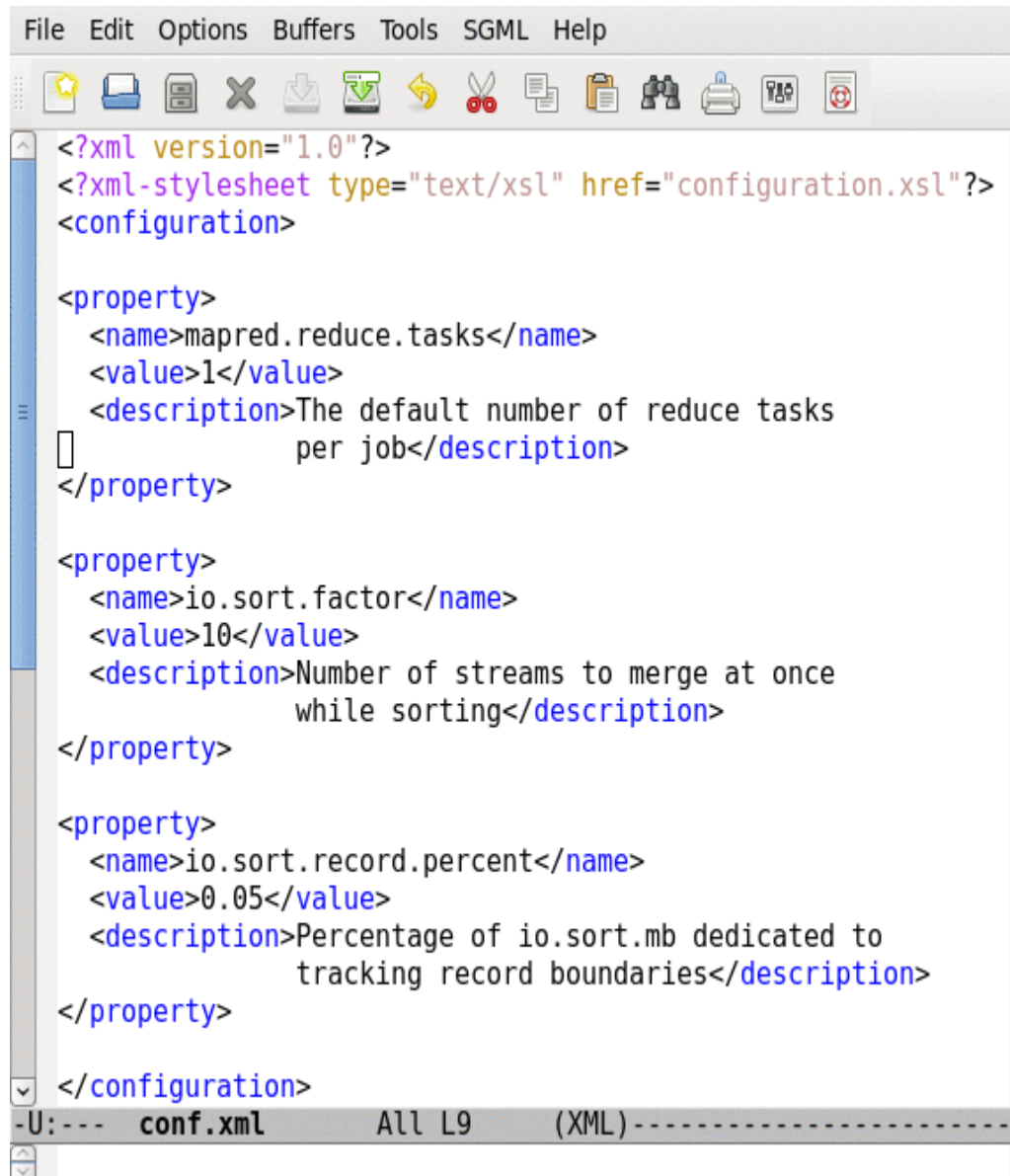
mapreduce.java All L9 (Java/l Abbrev) -----
Wrote /home/shivnath/Desktop/mapreduce.java

Lifecycle of a MapReduce Job



How are the number of splits, number of map and reduce tasks, memory allocation to tasks, etc., determined?

Job Configuration Parameters

A screenshot of an XML editor window. The title bar shows 'File Edit Options Buffers Tools SGML Help'. The toolbar contains icons for file operations and editing. The main text area displays an XML configuration file named 'conf.xml'. The XML structure includes a root element 'configuration' containing three 'property' elements. Each 'property' element has 'name', 'value', and 'description' sub-elements. The status bar at the bottom shows '-U: --- conf.xml All L9 (XML)'.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>

  <property>
    <name>mapred.reduce.tasks</name>
    <value>1</value>
    <description>The default number of reduce tasks
    per job</description>
  </property>

  <property>
    <name>io.sort.factor</name>
    <value>10</value>
    <description>Number of streams to merge at once
    while sorting</description>
  </property>

  <property>
    <name>io.sort.record.percent</name>
    <value>0.05</value>
    <description>Percentage of io.sort.mb dedicated to
    tracking record boundaries</description>
  </property>

</configuration>
```

- 190+ parameters in Hadoop
- Set manually or defaults are used

Hadoop Streaming

- Allows you to create and run map/reduce jobs with any executable
- Similar to unix pipes, e.g.:
 - format is: Input | Mapper | Reducer
 - echo “this sentence has five lines” | cat | wc

Hadoop Streaming

- Mapper and Reducer receive data from stdin and output to stdout
- Hadoop takes care of the transmission of data between the map/reduce tasks
 - It is still the programmer's responsibility to set the correct key/value
 - Default format: "key \t value\n"
- Let's look at a Python example of a MapReduce word count program...

Streaming_Mapper.py

```
# read in one line of input at a time from stdin
```

```
for line in sys.stdin:
```

```
    line = line.strip()      # string
```

```
    words = line.split()    # list of strings
```

```
# write data on stdout
```

```
for word in words:
```

```
    print '%s\t%i' % (word, 1)
```

Hadoop Streaming

- What are we outputting?
 - Example output: “the 1”
 - By default, “the” is the key, and “1” is the value
- Hadoop Streaming handles delivering this key/value pair to a Reducer
 - Able to send similar keys to the same Reducer or to an intermediary Combiner

Streaming_Reducer.py

```
wordcount = { }          # empty dictionary
# read in one line of input at a time from stdin
for line in sys.stdin:
    line = line.strip()    # string
    key,value = line.split()
    wordcount[key] = wordcount.get(key, 0) + value

# write data on stdout
for word, count in sorted(wordcount.items()):
    print '%s\t%i' % (word, count)
```

Hadoop Streaming

- Streaming Reducer receives single lines (which are key/value pairs) from stdin
 - Regular Reducer receives a collection of all the values for a particular key
 - It is still the case that all the values for a particular key will go to a single Reducer

Using Hadoop Distributed File System (HDFS)

- Can access HDFS through various shell commands (see Further Resources slide for link to documentation)
 - ❑ `hadoop -put <localsrc> ... <dst>`
 - ❑ `hadoop -get <src> <localdst>`
 - ❑ `hadoop -ls`
 - ❑ `hadoop -rm file`

Configuring Number of Tasks

- Normal method
 - ❑ `jobConf.setNumMapTasks(400)`
 - ❑ `jobConf.setNumReduceTasks(4)`
- Hadoop Streaming method
 - ❑ `-jobconf mapred.map.tasks=400`
 - ❑ `-jobconf mapred.reduce.tasks=4`
- Note: # of map tasks is only a hint to the framework. Actual number depends on the number of InputSplits generated

Running a Hadoop Job

- Place input file into HDFS:
 - ❑ `hadoop fs -put ./input-file input-file`
- Run either normal or streaming version:
 - ❑ `hadoop jar Wordcount.jar org.myorg.Wordcount input-file output-file`
 - ❑ `hadoop jar hadoop-streaming.jar \`
 - `-input input-file \`
 - `-output output-file \`
 - `-file Streaming_Mapper.py \`
 - `-mapper python Streaming_Mapper.py \`
 - `-file Streaming_Reducer.py \`
 - `-reducer python Streaming_Reducer.py \`

Output Parsing

- Output of the reduce tasks must be retrieved:
 - ❑ `hadoop fs -get output-file hadoop-output`
- This creates a directory of output files, 1 per reduce task
 - ❑ Output files numbered part-00000, part-00001, etc.
- Sample output of Wordcount
 - ❑ `head -n5 part-00000`

| | |
|---------|---|
| "tis | 1 |
| "come | 2 |
| "coming | 1 |
| "edwin | 1 |
| "found | 1 |

Case study 1

NetflixHadoop

NetflixHadoop: Problem Definition

- From Netflix Competition
 - Data: 100480507 rating data from 480189 users on 17770 movies.
 - Goal: Predict unknown ratings for any given user and movie pairs.
 - Measurement: Use RMSE to measure the precision.
- The approach: Singular Value Decomposition (SVD)

NetflixHadoop: SVD algorithm

- A feature means...
 - User: Preference (I like sci-fi or comedy...)
 - Movie: Genres, contents, ...
 - Abstract attribute of the object it belongs to.
- Feature Vector
 - Each user has a user feature vector;
 - Each movie has a movie feature vector.
- Rating for a (user, movie) pair can be estimated by a linear combination of the feature vectors of the user and the movie.
- Algorithm: Train the feature vectors to minimize the prediction error!

NetflixHadoop: SVD Pseudocode

- Basic idea:
 - Initialize the feature vectors;
 - Recursively: calculate the error, adjust the feature vectors.

LEARN (user,movie,rating,userFeatureVector,movieFeatureVector):

Find error in prediction using $\text{userFeature}(\text{rating.u})$ & $\text{movieFeature}(\text{rating.m})$

For each feature f :

// LRATE and Q are learning constants, these two updates done atomically:

$\text{userFeature}[f] \leftarrow \text{LRATE} * (\text{err} * \text{movieFeature}[f] Q$
 $\quad * \text{userFeature}[f])$

$\text{movieFeature}[f] \leftarrow \text{LRATE} * (\text{err} * \text{userFeature}[f] Q$
 $\quad * \text{movieFeature}[f])$

NetflixHadoop: Implementation

- Data Pre-process
 - Randomize the data sequence.
 - Mapper: for each record, randomly assign an integer key.
 - Reducer: do nothing; simply output (automatically sort the output based on the key)
 - Customized RatingOutputFormat from FileOutputFormat
 - Remove the key in the output.

NetflixHadoop: Implementation

- Feature Vector Training
 - Mapper: From an input (user, movie, rating), adjust the related feature vectors, output the vectors for the user and the movie.
 - Reducer: Compute the average of the feature vectors collected from the map phase for a given user/movie.
- Challenge: Global sharing feature vectors!

NetflixHadoop: Implementation

- Global sharing feature vectors
 - Global Variables: fail!
 - Different mappers use different JVM and no global variable available between different JVM.
 - Database (DBInputFormat): fail!
 - Error on configuration; expecting bad performance due to frequent updates (race condition, query start-up overhead)
 - Configuration files in Hadoop: fine!
 - Data can be shared and modified by different mappers; limited by the main memory of each working node.

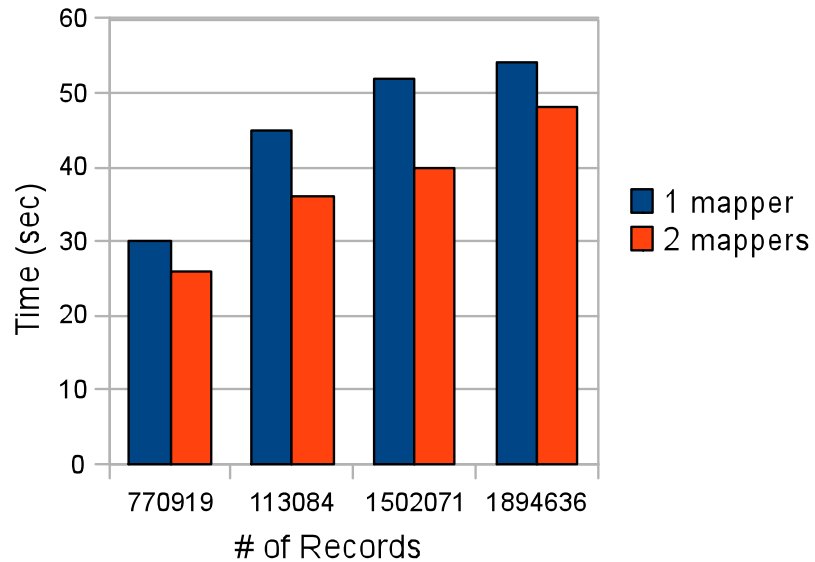
NetflixHadoop: Experiments

- Experiments using single-thread, multi-thread and MapReduce
- Test Environment
 - Hadoop 0.19.1
 - Single-machine, virtual environment:
 - Host: 2.2 GHz Intel Core 2 Duo, 4GB 667 RAM, Max OS X
 - Virtual machine: 2 virtual processors, 748MB RAM each, Fedora 10.
 - Distributed environment:
 - 4 nodes (should be... currently 9 node)
 - 400 GB Hard Driver on each node
 - Hadoop Heap Size: 1GB (failed to finish)

NetflixHadoop: Experiments

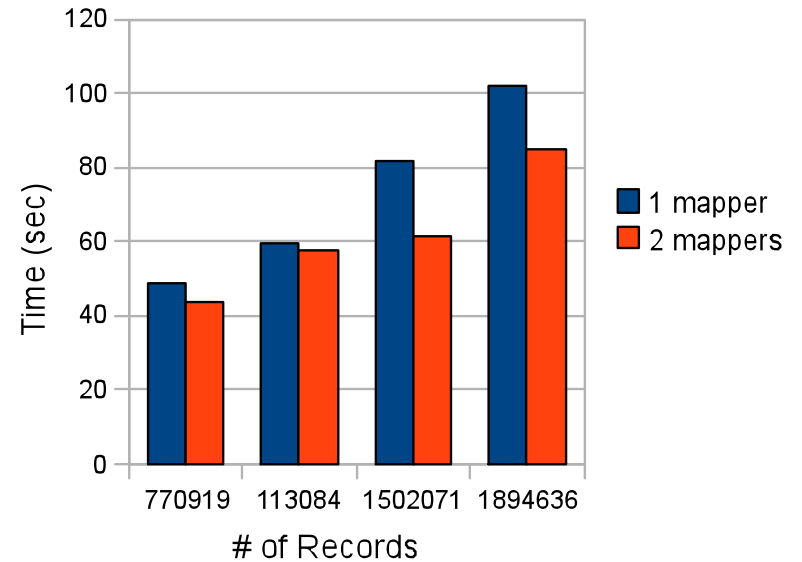
1 mapper v.s. 2 mappers

Randomizer



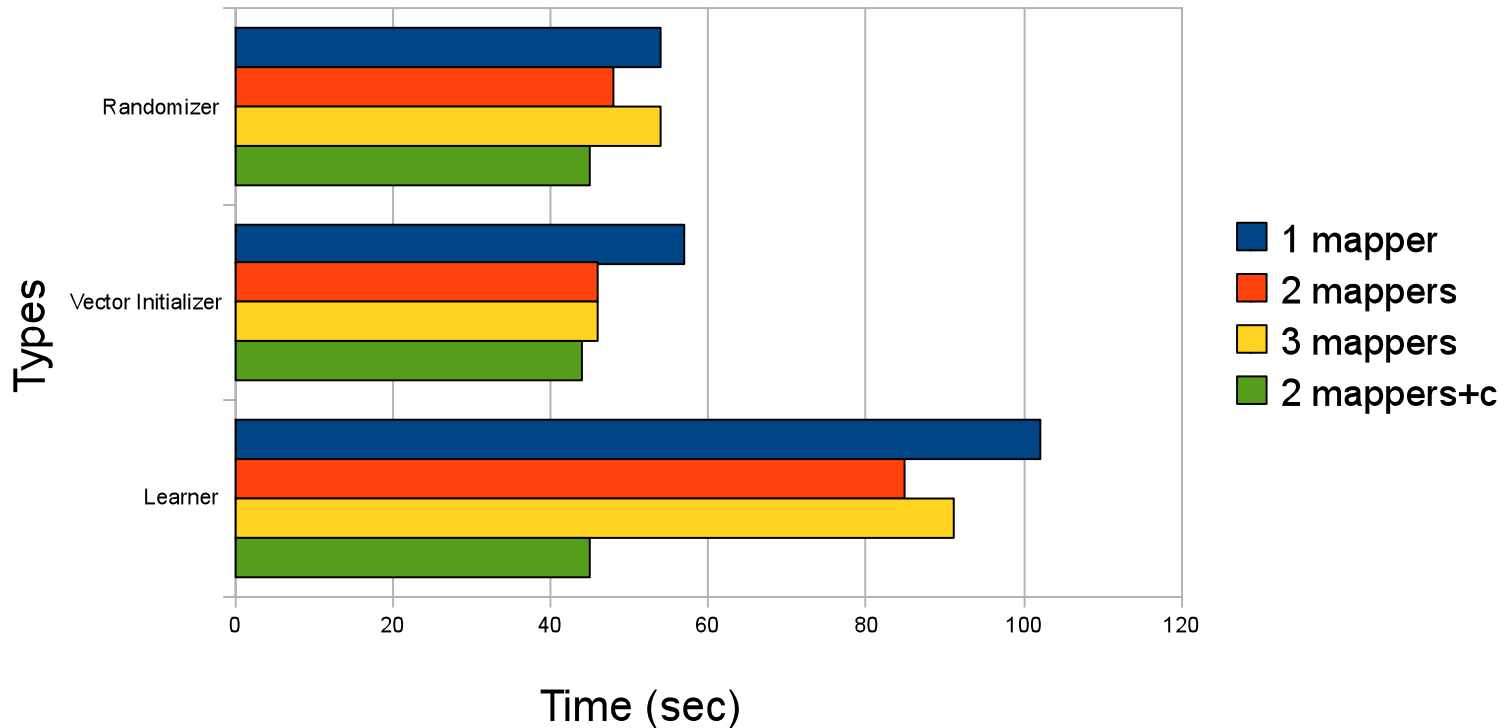
1 mapper v.s. 2 mapper2

Learner



NetflixHadoop: Experiments

Mappers 123
on 1894636 ratings



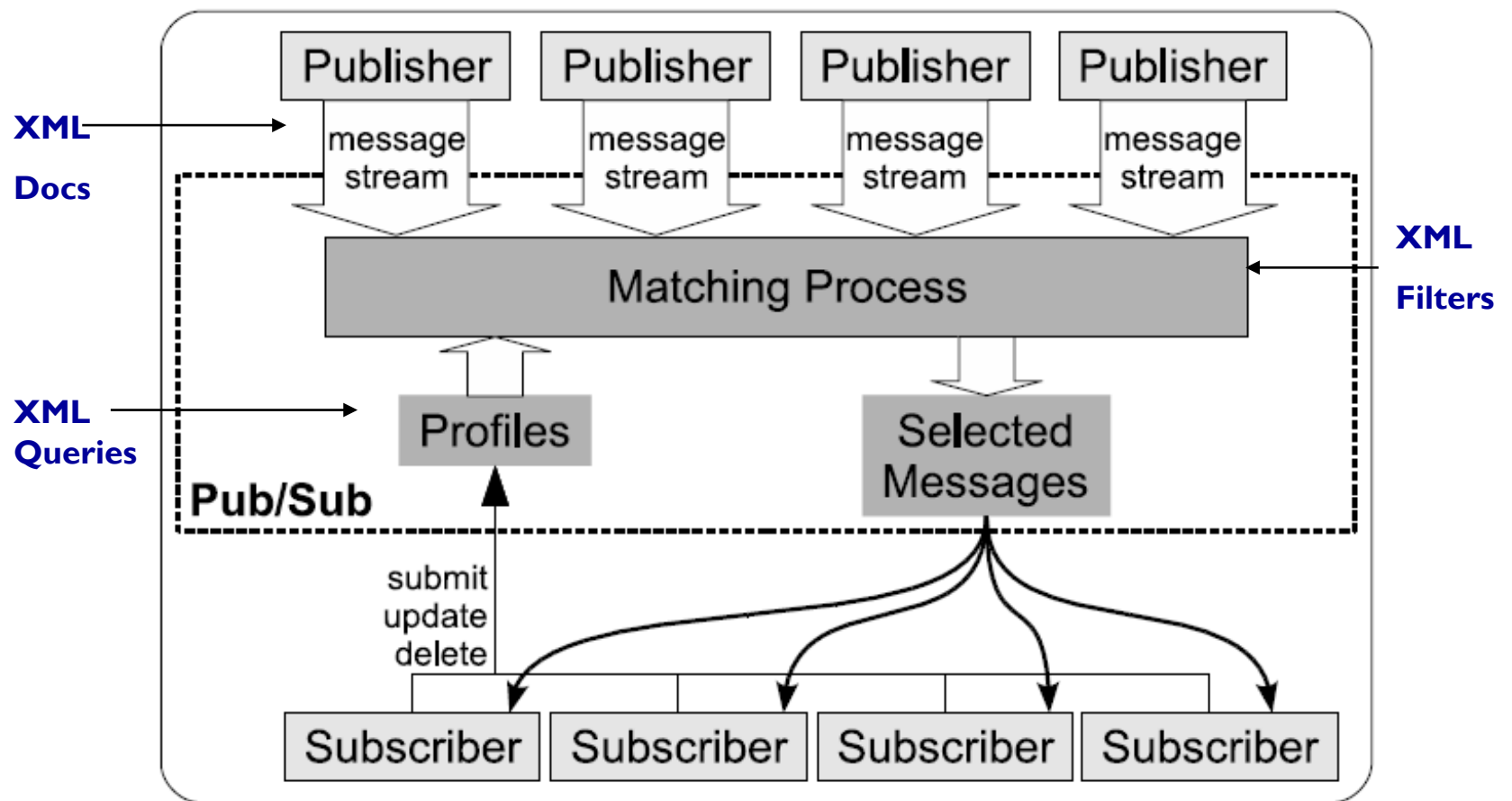
Case study 2

XML Filtering

XML Filtering: Problem Definition

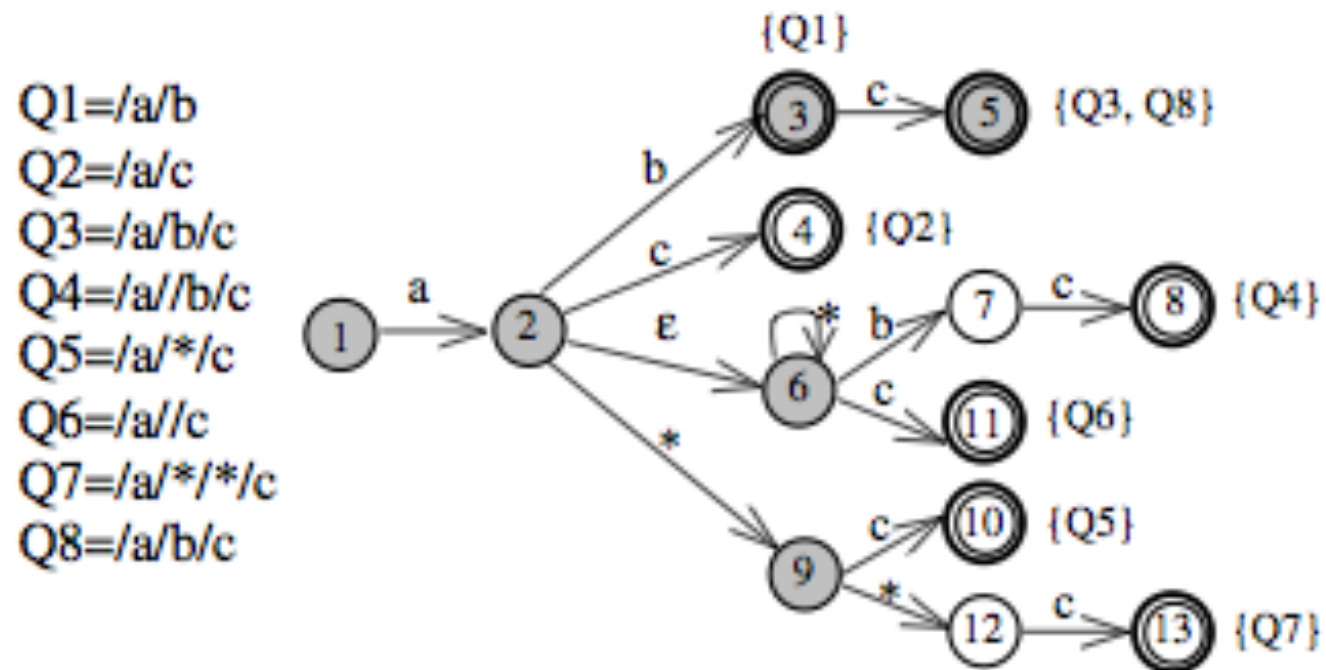
- Aimed at a publish/subscriber system utilizing distributed computation environment
 - Pub/sub: Queries are known, data are fed as a stream into the system (DBMS: data are known, queries are fed).

XML Filtering: Pub/Sub System



XML Filtering: Algorithms

- Use YFilter Algorithm
 - YFilter: XML queries are indexed as a NFA, then XML data is fed into the NFA and test the final state output.
 - Easy for parallel: queries can be partitioned and indexed separately.

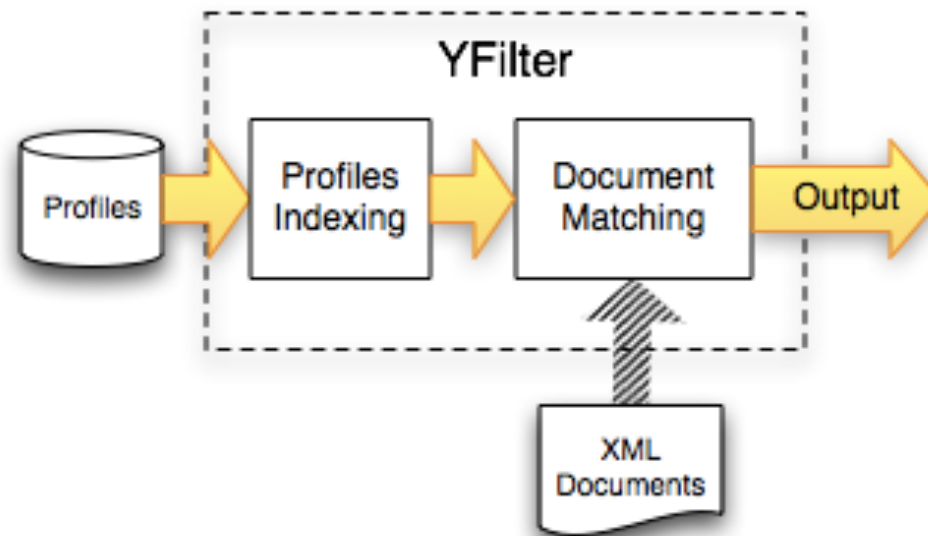


XML Filtering: Implementations

- Three benchmark platforms are implemented in our project:
 - Single-threaded: Directly apply the YFilter on the profiles and document stream.
 - Multi-threaded: Parallel YFilter onto different threads.
 - Map/Reduce: Parallel YFilter onto different machines (currently in pseudo-distributed environment).

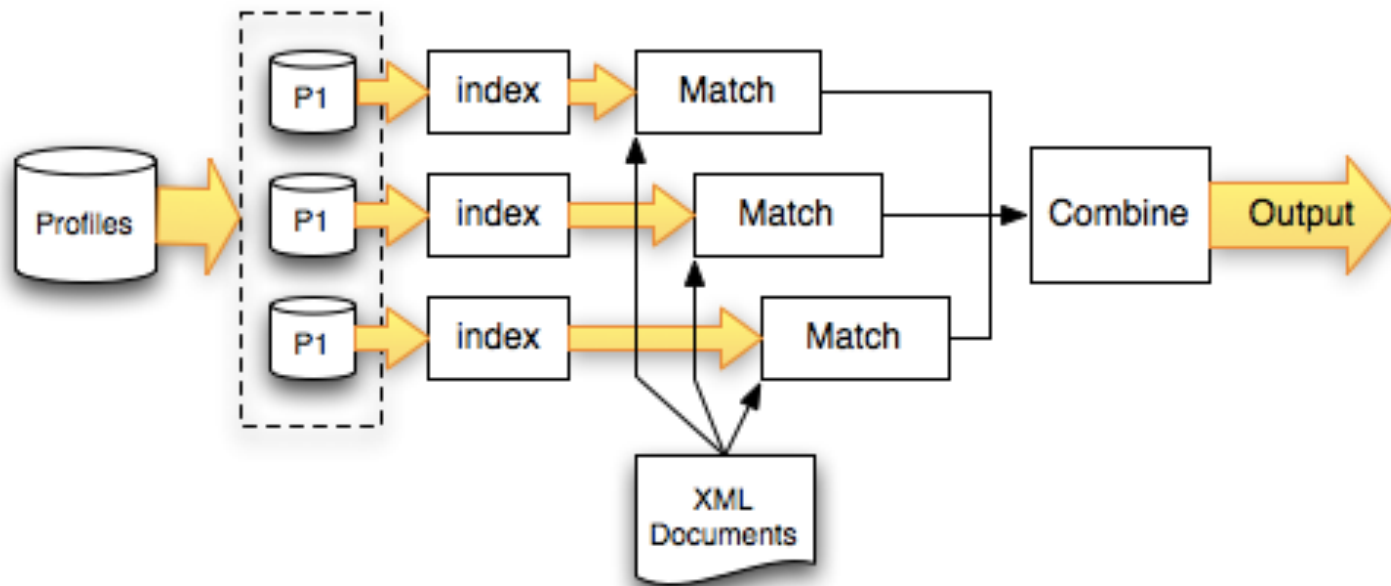
XML Filtering: Single-Threaded Implementation

- The index (NFA) is built once on the whole set of profiles.
- Documents then are streamed into the YFilter for matching.
- Matching results then are returned by YFilter.



XML Filtering: Multi-Threaded Implementation

- Profiles are split into parts, and each part of the profiles are used to build a NFA separately.
- Each YFilter instance listens a port for income documents, then it outputs the results through the socket.



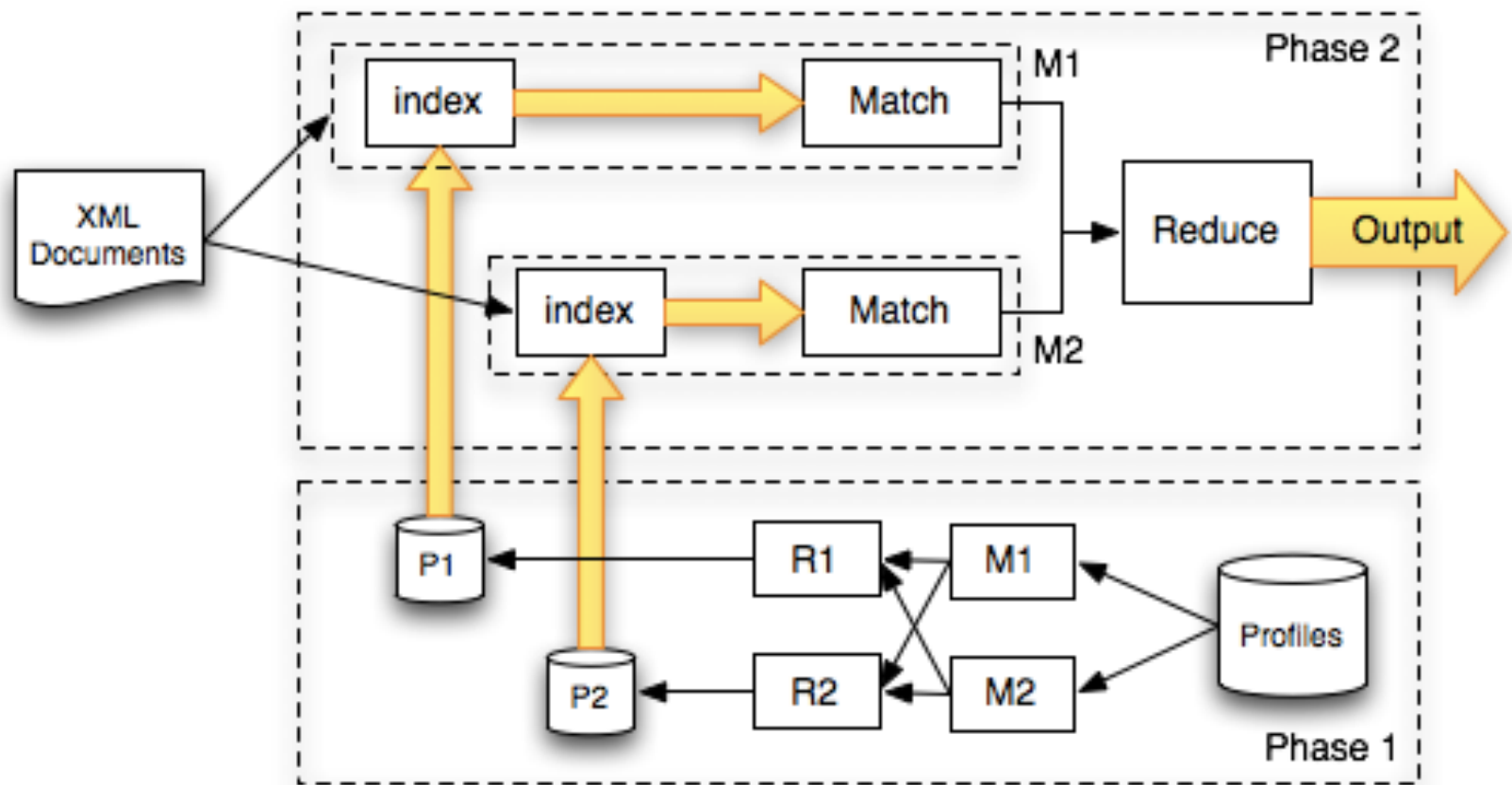
XML Filtering: Map/Reduce Implementation

- Profile splitting: Profiles are read line by line with line number as the key and profile as the value.
 - Map: For each profile, assign a new key using $(\text{old_key} \% \text{split_num})$
 - Reduce: For all profiles with the same key, output them into a file.
 - Output: Separated profiles, each with profiles having the same $(\text{old_key} \% \text{split_num})$ value.

XML Filtering: Map/Reduce Implementation

- Document matching: Split profiles are read file by file with file number as the key and profiles as the value.
 - Map: For each set of profiles, run YFilter on the document (fed as a configuration of the job), and output the old_key of the matching profile as the key and the file number as the values.
 - Reduce: Just collect results.
 - Output: All keys (line numbers) of matching profiles.

XML Filtering: Map/Reduce Implementation



XML Filtering: Experiments

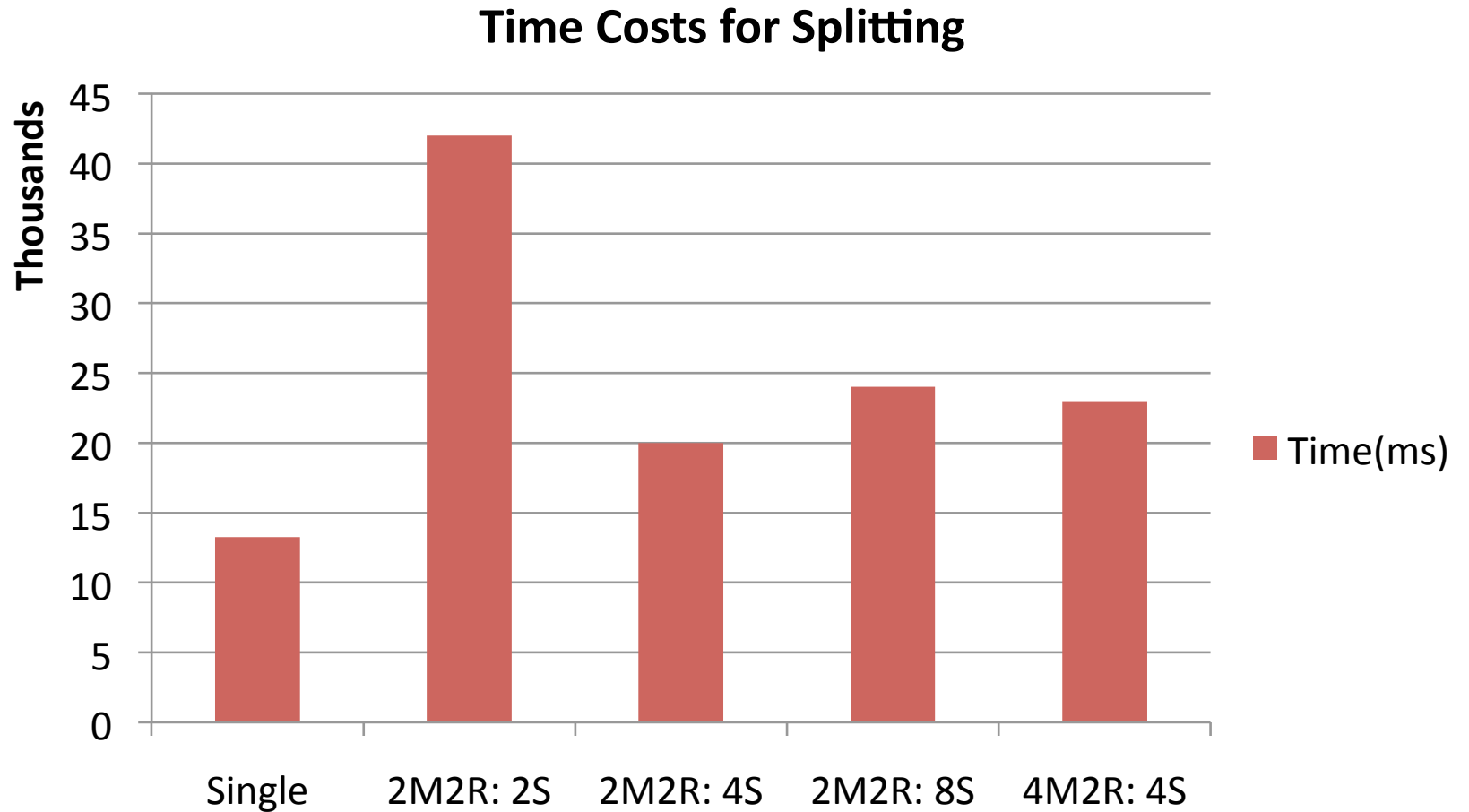
- Hardware:
 - Macbook 2.2 GHz Intel Core 2 Duo
 - 4G 667 MHz DDR2 SDRAM
- Software:
 - Java 1.6.0_17, 1GB heap size
 - Cloudera Hadoop Distribution (0.20.1) in a virtual machine.
- Data:
 - XML docs: SIGMOD Record (9 files).
 - Profiles: 25K and 50K profiles on SIGMOD Record.

| Data | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------|--------|--------|--------|--------|--------|-------|-------|-------|-------|
| Size | 478416 | 415043 | 312515 | 213197 | 103528 | 53019 | 42128 | 30467 | 20984 |

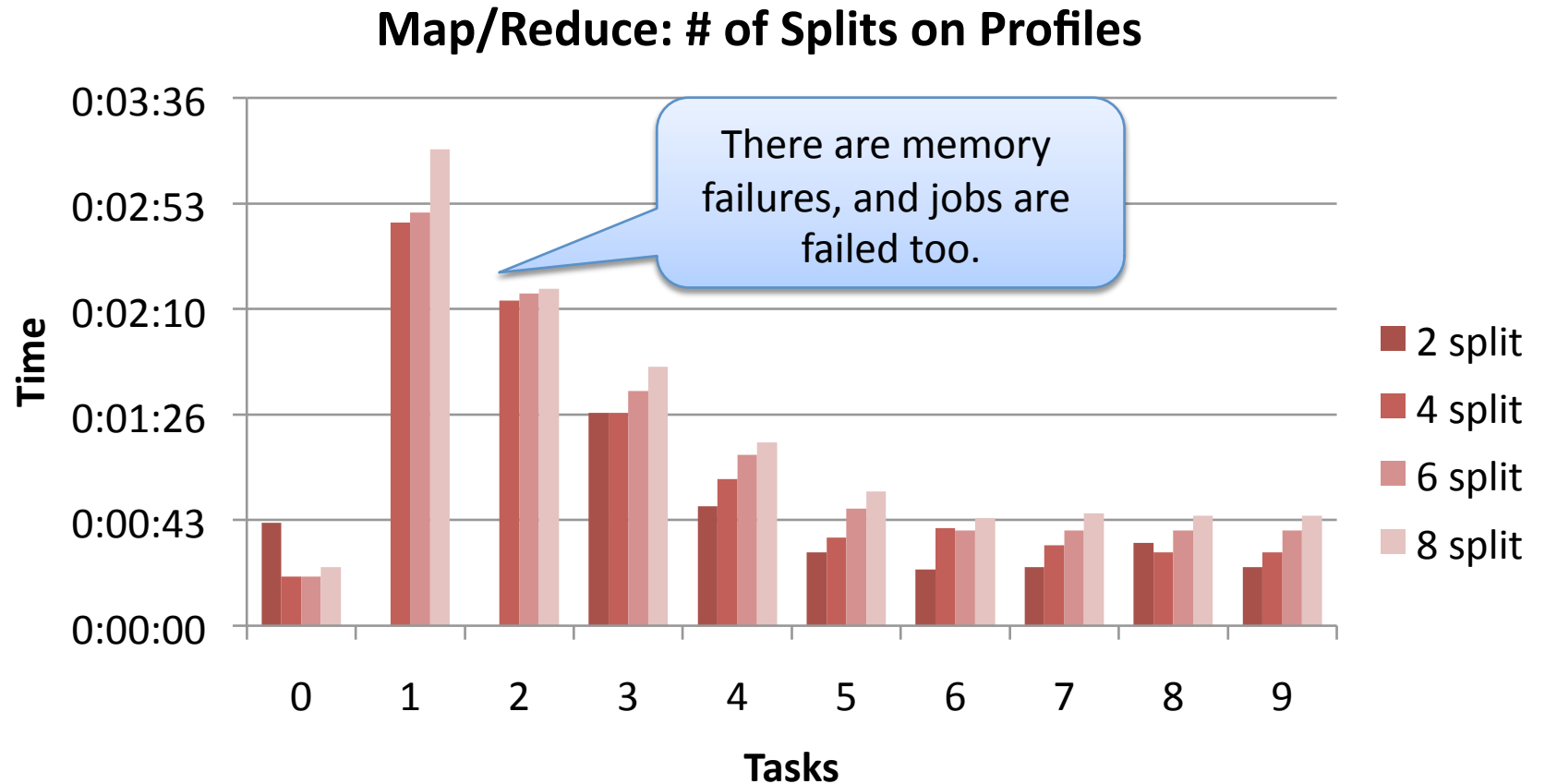
XML Filtering: Experiments

- Run-out-of-memory: We encountered this problem in all the three benchmarks, however Hadoop is much robust on this:
 - Smaller profile split
 - Map phase scheduler uses the memory wisely.
- Race-condition: since the YFilter code we are using is not thread-safe, in multi-threaded version race-condition messes the results; however Hadoop works this around by its shared-nothing run-time.
 - Separate JVM are used for different mappers, instead of threads that may share something lower-level.

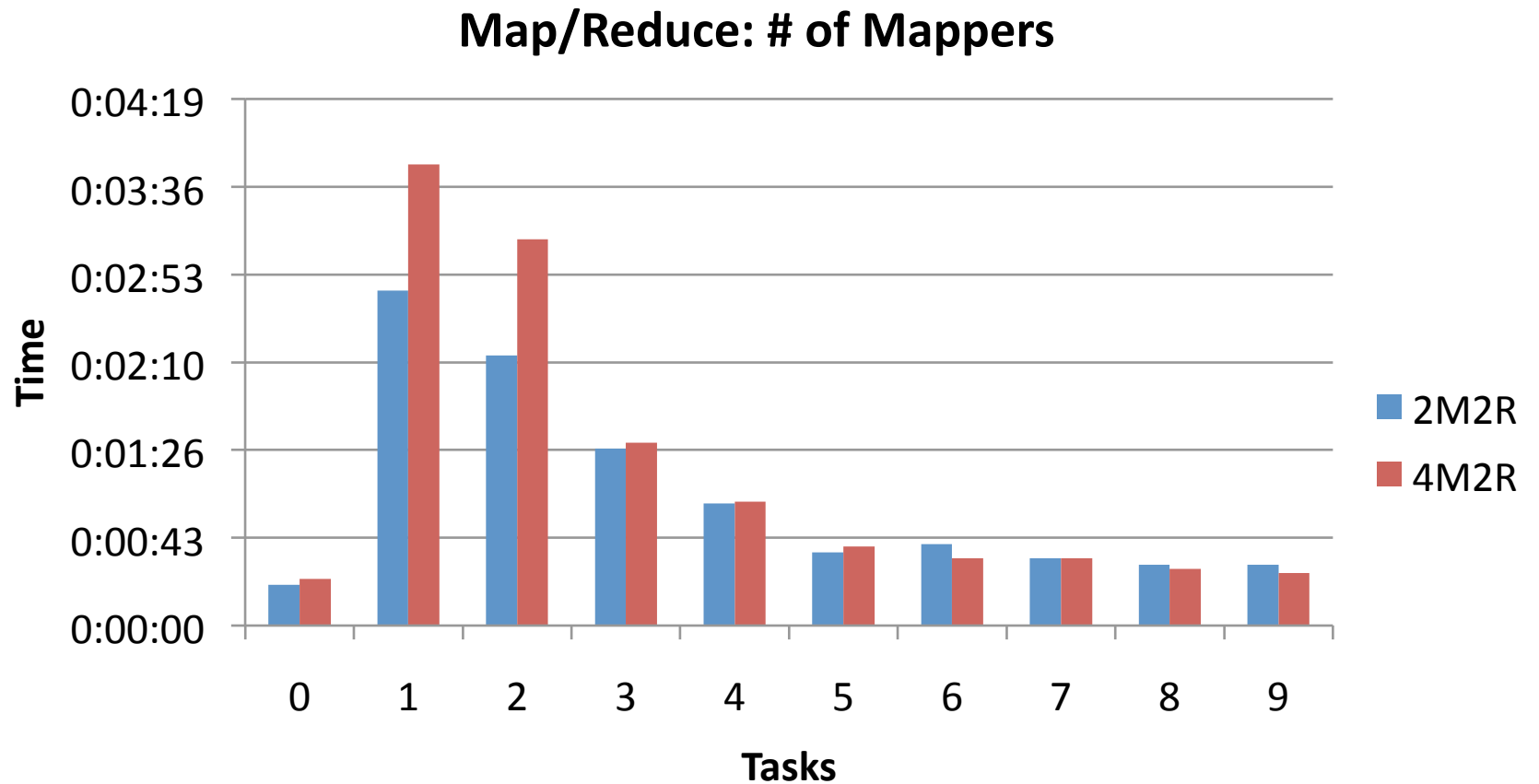
XML Filtering: Experiments



XML Filtering: Experiments



XML Filtering: Experiments



XML Filtering: Experiments

