Map-Reduce Programming Model

Chandan Mazumdar

Jadavpur University

Map Reduce

- A style of computing implemented in several systems including Google's internal implementation and the popular open-source implementation Hadoop.

- Implementation of MapReduce uses two functions called Map and Reduce.

MapReduce Computation

- A number of Map tasks each with one or more chunks turn the chunk into a sequence of <key, value> pairs depending upon the code written by the user for the Map function.

- The key-value pairs are collected by a master controller and sorted by key. The keys are divided among all Reduce tasks such that all the key-value pairs with same key get associated with same Reduce tasks.

- The Reduce tasks work on one key at a time combining the key values in a manner defined by the code written by the user for the Map Reduce function.

The Map Tasks

- Input files of Map tasks can consist of elements of any type, may be a tuple or a document.
- A chunk is a collection of elements, and no element is stored across two chunks.
- The Map function takes an input element as its argument and produces zero or more key-value pairs.
- The types of key and values so produced are arbitary and keys are not "keys" the usual sense ; i.e. they do not have to be unique.
- Rather Map tasks produce several key-value pairs with the same key, possibly even from the same element.

Grouping by key

- After the Map tasks the key-value pairs are grouped by keys by the system.
- The master controller process knowing the number r of Reduce tasks , where r is defined by user, picks a hash function that applies to keys and produces a bucket number from 0 to r − 1.
- Each key that is output by a Map task is hashed and its key-value pair is put in one of r local files and these files are used by Reduce tasks.

Grouping by key

- For grouping by key, the master controller merges the files from each Map task destined for a particular Reduce task and feeds the merged file to that reducer process as a sequence of <key, list-of-value> pairs.

- That is, for each key k, the input to the Reduce task that handles key k is a pair of the form (k, [v1, v2, . . . ,vn]), where(k, v1), (k, v2), . . . , (k, vn) are all the key-value pairs with key k coming form the Map tasks.

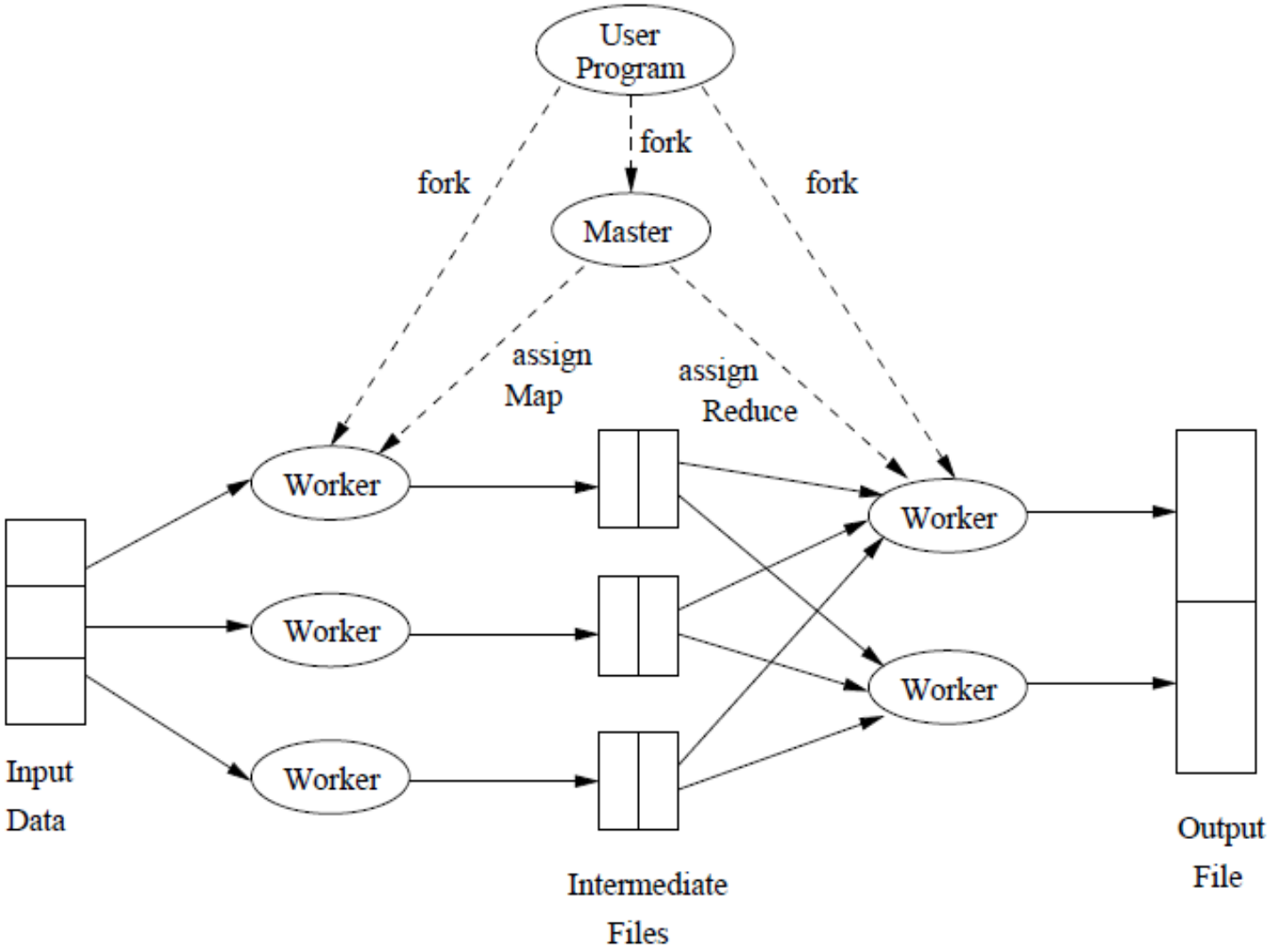
Reduce Tasks

- This function takes as argument a pair consisting of a key and its list of associated values.
- The output produced is a sequence of zero or more key-value pairs , where the pairs so obtained can be a type different from those sent from Map tasks to Reduce tasks .
- A Reduce task receives one or more keys and their associated value lists and the outputs of the Reduce tasks are merged into a single file.
- Reducers are partitioned among smaller number of Reduce tasks by hashing the keys and associating each Reduce task with one of the buckets of the hash function.

Combiners

- A Combiner is a Reduce function which is associative and commutative, values can be combined in any order yielding the same result.

- When the Reduce function is associative and commutative, we can push some of what the reducers do to the Map tasks.


User Program
fork
fork
fork
Master
assign Map
assign Reduce
Worker
Worker
Worker
Worker
Worker
Input Data
Intermediate Files
Output File

MapReduce - What?

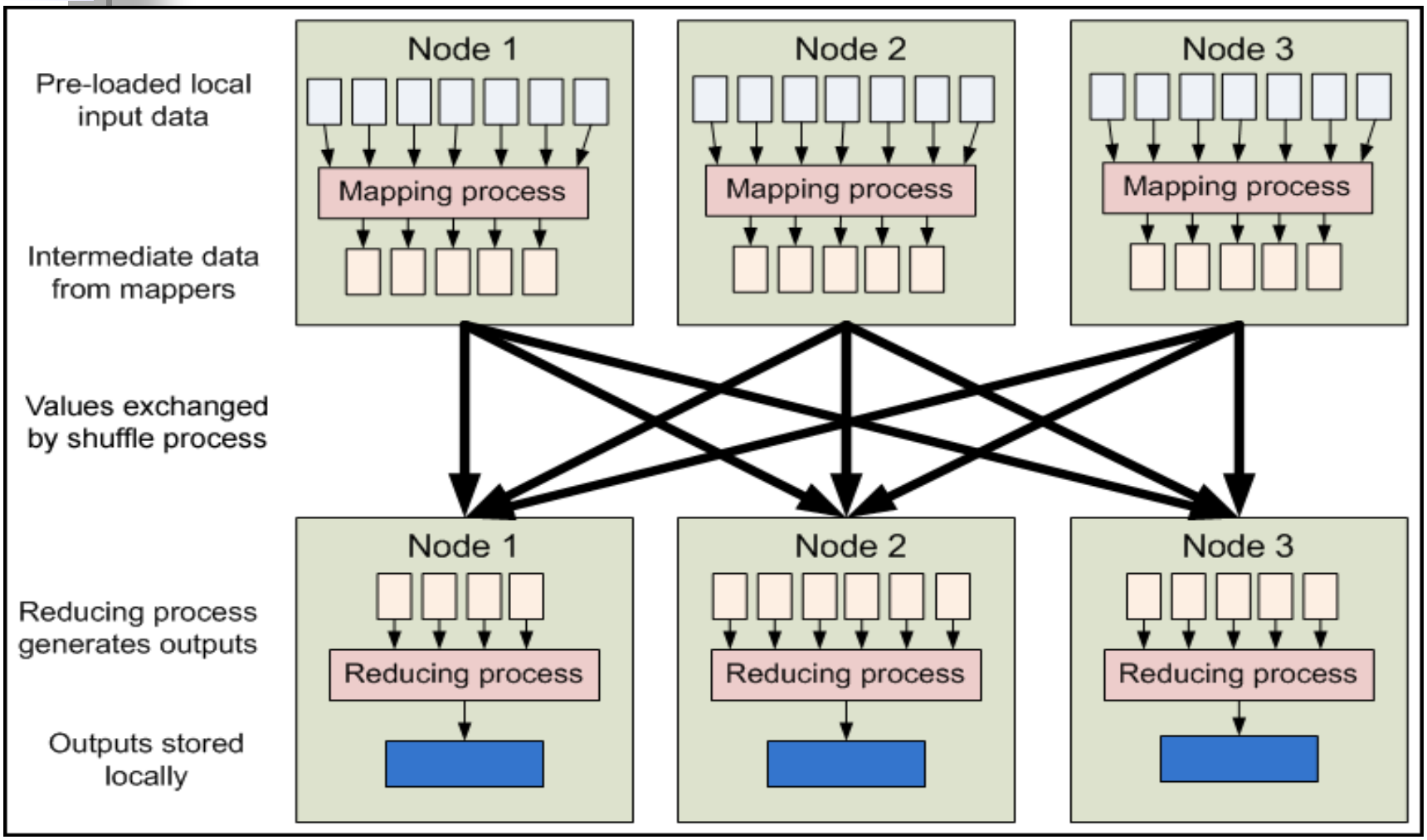
- MapReduce is a programming model for efficient distributed computing

- It works like a Unix pipeline
  - cat input | grep | sort | uniq -c | cat > output
  - **Input | Map |** Shuffle & Sort **| Reduce | Output**

- Efficiency from
  - Streaming through data, reducing seeks
  - Pipelining

- A good fit for a lot of applications
  - Log processing
  - Web index building

MapReduce - Dataflow



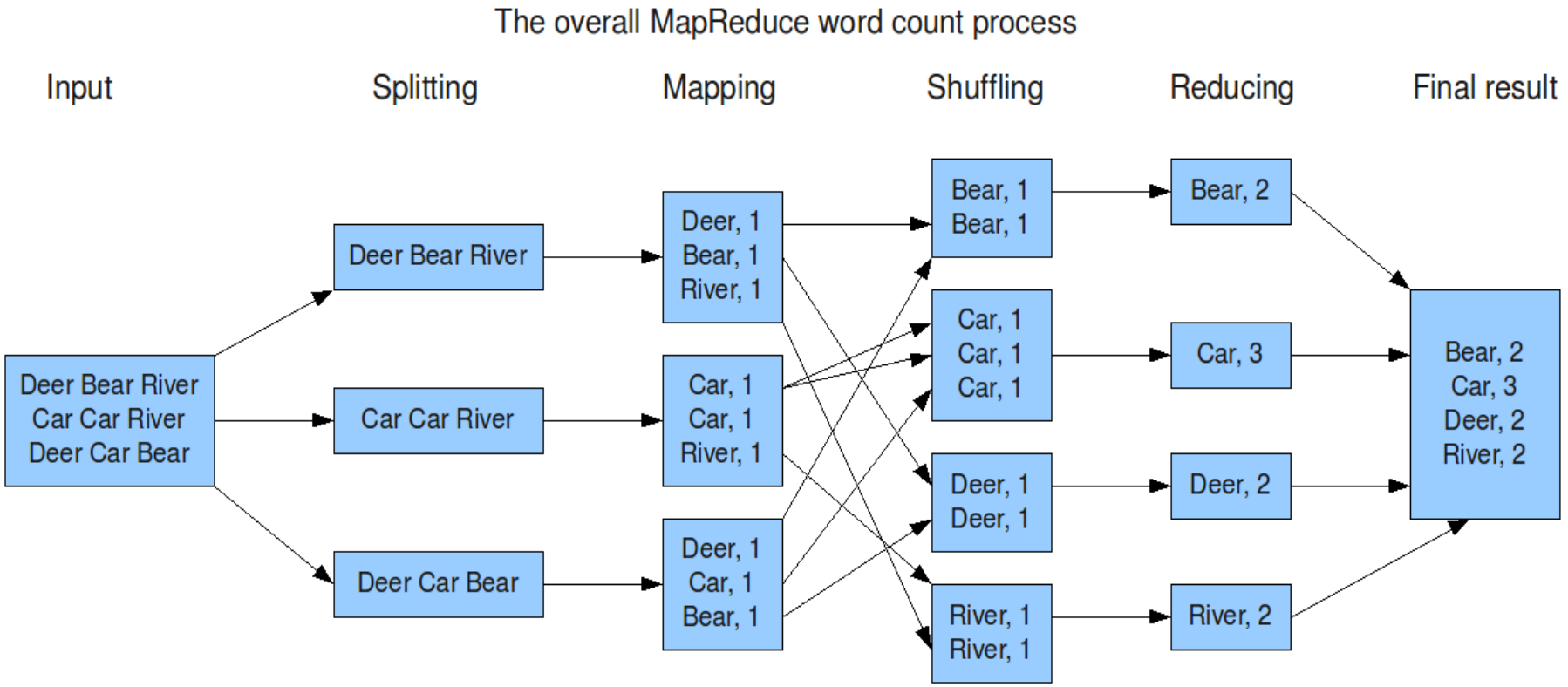
MapReduce - Features

- Fine grained Map and Reduce tasks
  - Improved load balancing
  - Faster recovery from failed tasks
- Automatic re-execution on failure
  - In a large cluster, some nodes are always slow or flaky
  - Framework re-executes failed tasks
- Locality optimizations
  - With large data, bandwidth is a problem
  - Map-Reduce + HDFS is a very effective solution
  - Map-Reduce queries HDFS for locations of input data
  - Map tasks are scheduled close to the inputs when possible

Word Count Example

- Mapper
  - Input: value: lines of text of input
  - Output: key: word, value: 1
- Reducer
  - Input: key: word, value: set of counts
  - Output: key: word, value: sum
- Launching program
  - Defines this job
  - Submits job to cluster

Word Count Dataflow



Word Count Mapper

```
public static class Map extends MapReduceBase implements
    Mapper<LongWritable,Text,Text,IntWritable> {
  private static final IntWritable one = new IntWritable(1);
  private Text word  = new Text();

  public static void map(LongWritable key, Text value,
    OutputCollector<Text,IntWritable> output, Reporter reporter) throws
    IOException {
     String line = value.toString();
     StringTokenizer = new StringTokenizer(line);
     while(tokenizer.hasNext()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
        }
     }
}
```

Word Count Reducer

```
public static class Reduce extends MapReduceBase implements
    Reducer<Text,IntWritable,Text,IntWritable> {
public static void map(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));
    }
 }
```

Word Count Example

- Jobs are controlled by configuring *JobConfs*
- JobConfs are maps from attribute names to string values
- The framework defines attributes to control how the job is executed
  - `conf.set("mapred.job.name", "MyApp");`
- Applications can add arbitrary values to the JobConf
  - `conf.set("my.string", "foo");`
  - `conf.set("my.integer", 12);`
- JobConf is available to all tasks

Putting it all together

- Create a launching program for your application
- The launching program configures:
  - The *Mapper* and *Reducer* to use
  - The output key and value types (input types are inferred from the *InputFormat*)
  - The locations for your input and output
- The launching program then submits the job and typically waits for it to complete

Putting it all together

```
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.setReducer(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);
```

Input and Output Formats

- A Map/Reduce may specify how it's input is to be read by specifying an *InputFormat* to be used
- A Map/Reduce may specify how it's output is to be written by specifying an *OutputFormat* to be used
- These default to *TextInputFormat* and *TextOutputFormat*, which process line-based text data
- Another common choice is *SequenceFileInputFormat* and *SequenceFileOutputFormat* for binary data
- These are file-based

How many Maps and Reduces

- Maps
  - Usually as many as the number of HDFS blocks being processed, this is the default
  - Else the number of maps can be specified as a hint
  - The number of maps can also be controlled by specifying the *minimum split size*
  - The actual sizes of the map inputs are computed by:
    - *max(min(block_size,data/#maps), min_split_size)*

- Reduces
  - Unless the amount of data being processed is small
    - *0.95*num_nodes*mapred.tasktracker.tasks.maximum*

Tools

- Pig
  - High-level language for data analysis
- HBase
  - Table storage for semi-structured data
- Zookeeper
  - Coordinating distributed applications
- Hive
  - SQL-like Query language and Metastore
- Mahout
  - Machine learning
- Storm, SPARK
  - Stream Processing

THANK YOU !