Hadoop MapReduce (part a)

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

Slides are derived from online materials, mostly the Yahoo! Hadoop tutorial:
https://developer.yahoo.com/hadoop/tutorial/

Learning objectives

A brief history of MapReduce and Hadoop.

MapReduce programming basics:

- mappers, reduces, and combiners
- simple programing patterns
- data flow and job lifecycle
- input format and output format
- record readers and record writers

Brief history: lambda calculus

From Wikipedia:

"Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.

… The λ-calculus provides a simple semantics for computation, enabling properties of computation to be studied formally. The first simplification is that the λ-calculus treats functions "anonymously", without giving them explicit names.

… The second simplification is that the λ-calculus only uses functions of a single input."

Examples:

$$(x, y) \mapsto x \times x + y \times y$$

$$x \mapsto (y \mapsto x \times x + y \times y)$$

Brief history: MR at Google

From introduction of MapReduce paper (OSDI'04):

"Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues."

Brief history: Hadoop

- The most prominent open-source implementation of Google's MapReduce system.
- Created in 2005 by Doug Cutting and Mike Cafarella.
- Doug Cutting was working for Yahoo! at the time and named the system after his son's toy elephant.

Image source: http://www.boiledbeans.net/2011/03/24/not-the-water-droplets-kind-and-not-native-american-either/

High-level architecture

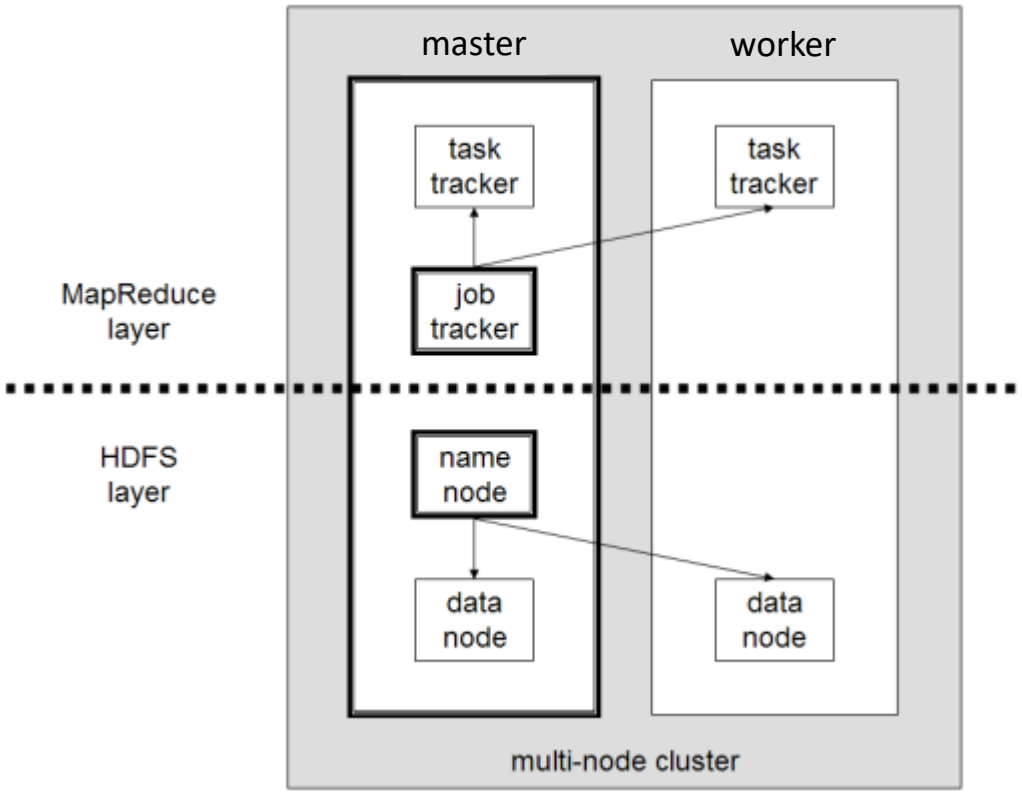


Image source:
https://en.wikipedia.org/wiki/Apache_Hadoop

MapReduce basics

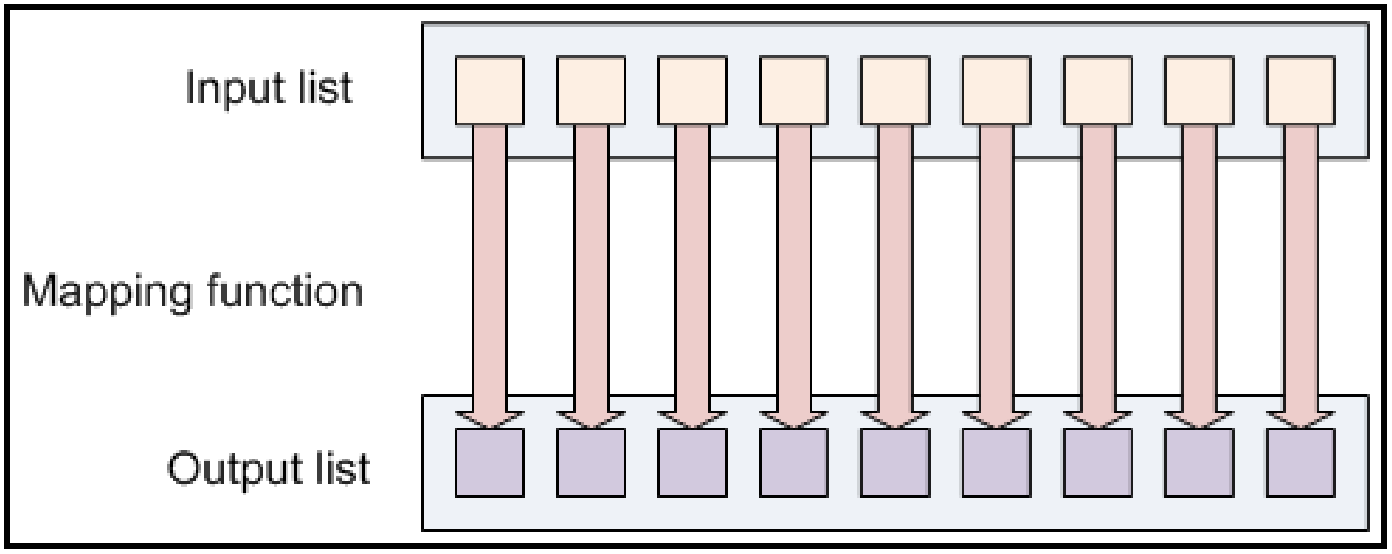
Material from this point onward is from the Yahoo! Hadoop tutorial:

- MapReduce performs parallel computation on large volumes of data.
- Components are not allowed to share data arbitrarily as the communication overhead required to keep data synchronized across nodes would prevent the system from performing reliably or efficiently at large scale.
- Data elements in MapReduce are immutable. Communication occurs only by generating new outputs, which are then forwarded by the Hadoop system into the next phase of execution.
- Conceptually, MapReduce programs transform lists of input data elements into lists of output data elements. A single MapReduce program will usually do this twice, using two different list processing idioms: map, and reduce. These terms are borrowed from functional programming languages.

MapReduce basics: mapper

The first phase of a MapReduce program is called mapping. A list of data elements are provided, one at a time, to a function called the Mapper, which transforms each element individually to one output data element, or sometimes zero or more output elements.

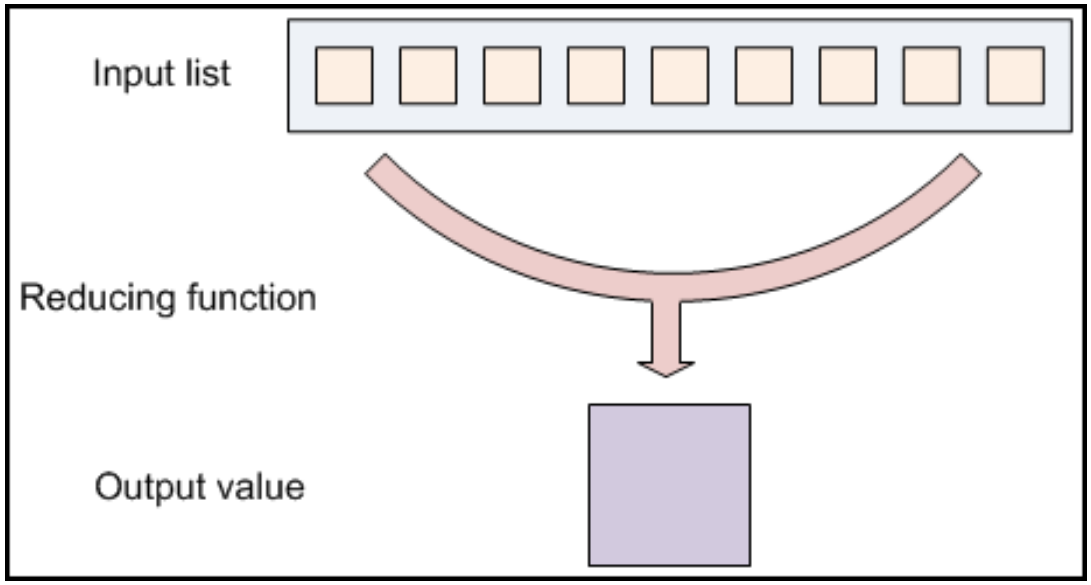
Example 1: convert each line of input text to uppercase.
Example 2: for each line of input text, output each word individually.



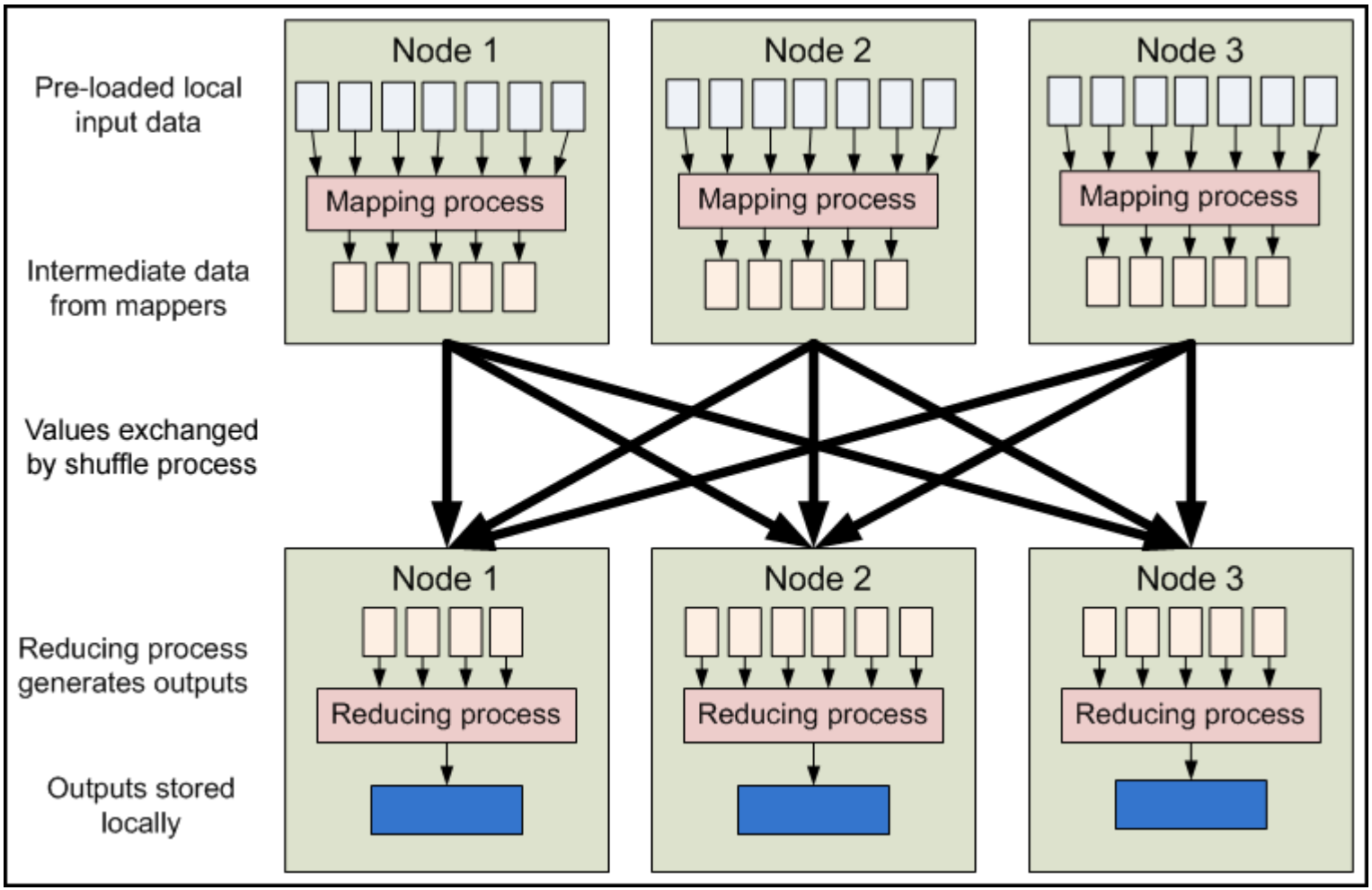
MapReduce basics: reducer

Reducing lets you aggregate values together. A reducer function receives an iterator of input values from an input list. It then combines these values together, returning a single output value.

Example: computing a sum of elements in the input list.



Data flow


Pre-loaded local input data
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Outputs stored locally
Node 1
Mapping process
Node 2
Mapping process
Node 3
Mapping process
Node 1
Reducing process
Node 2
Reducing process
Node 3
Reducing process

Data flow

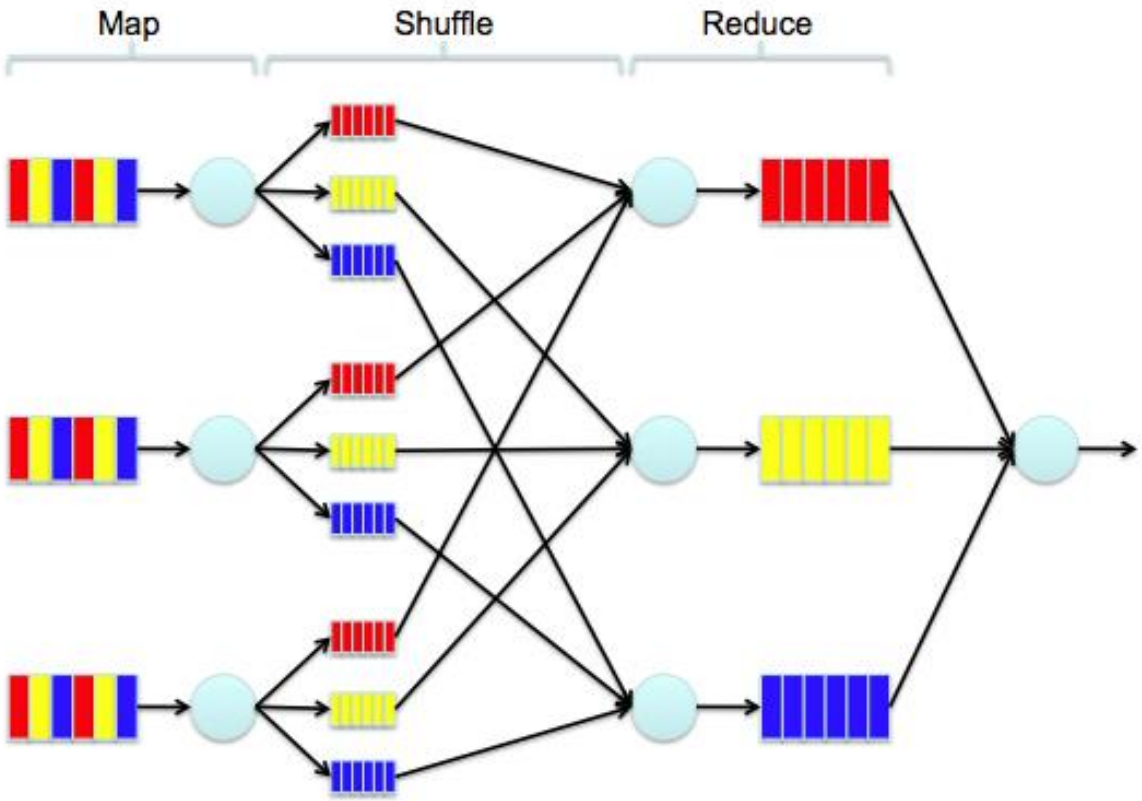
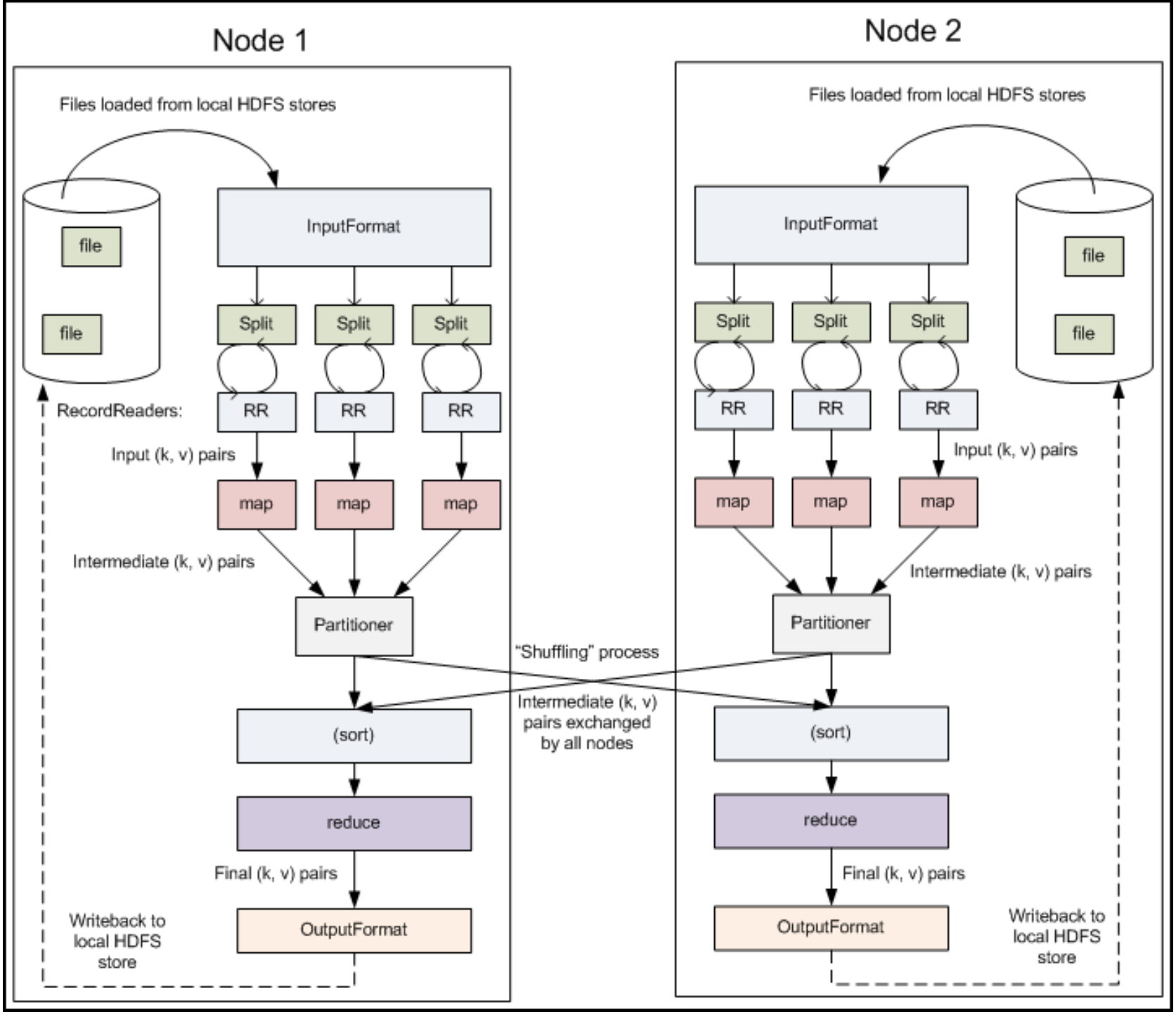
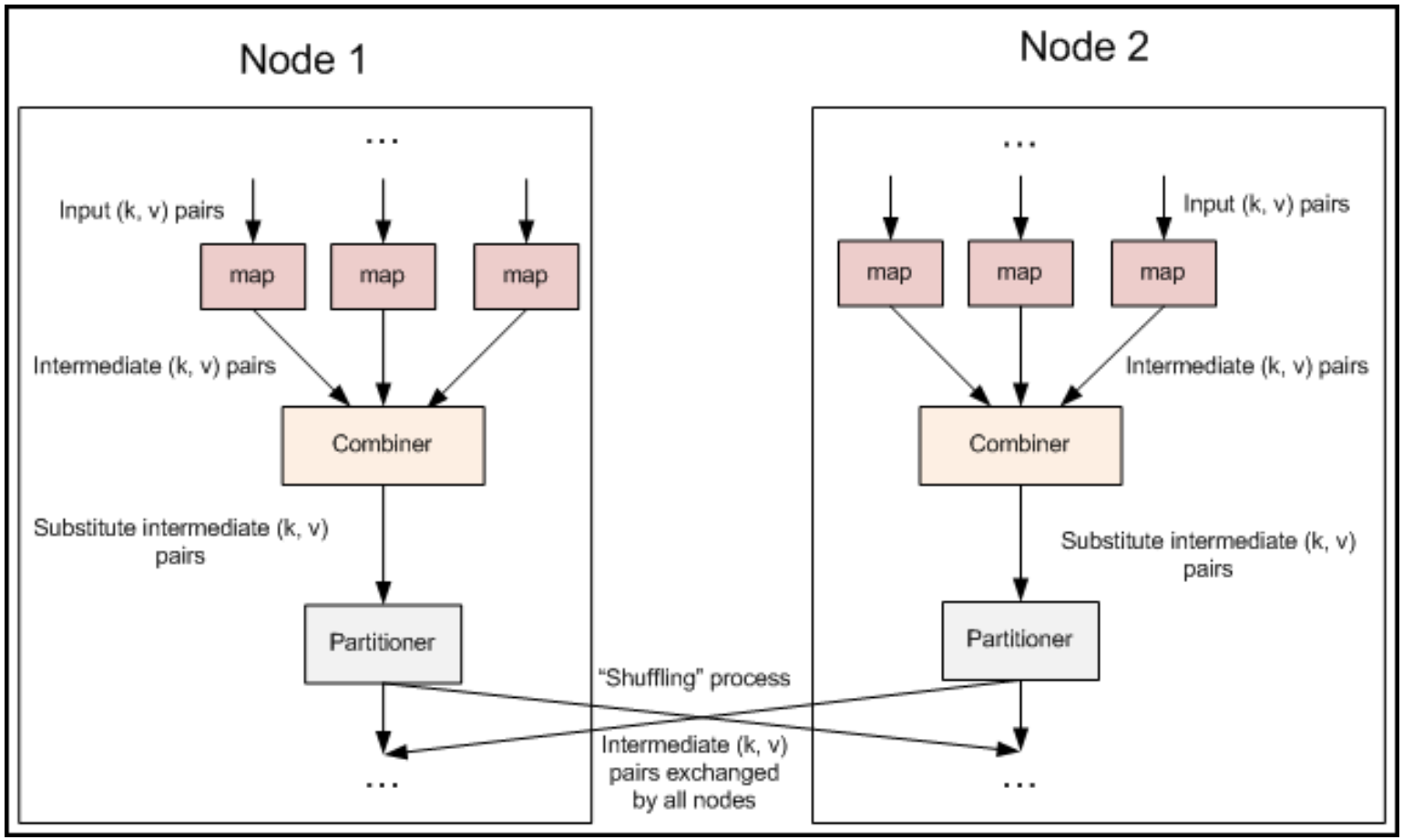


Image source: Lynn Langit
http://www.slideshare.net/lynnlangit/hadoop-mapreduce-fundamentals-21427224


Node 1
Node 2
Files loaded from local HDFS stores
Files loaded from local HDFS stores
file
file
file
file
InputFormat
InputFormat
Split
Split
Split
Split
Split
Split
RecordReaders:
RR
RR
RR
RR
RR
RR
Input (k, v) pairs
Input (k, v) pairs
map
map
map
map
map
map
Intermediate (k, v) pairs
Intermediate (k, v) pairs
Partitioner
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
(sort)
reduce
reduce
Final (k, v) pairs
Final (k, v) pairs
Writeback to local HDFS store
OutputFormat
OutputFormat
Writeback to local HDFS store

Adding combiners



Technical terms

InputSplit: A unit of work assigned to one map task. Usually corresponds to a chunk of an input file. Each record in a file belongs to exactly one input split and the framework takes care of dealing with record boundaries.

InputFormat: Determines how the input files are parsed, and defines the input splits. Factory for RecordReader objects. Examples: TextInputFormat, KeyValueInputFormat, SequenceFileInputFormat.

RecordReader: Loads data from an input split and creates key-value pairs for the mapper.

Partitioner: Determines which partition a given key-value pair will go to. The default partitioner simply hashes the key emitted by the mapper.

OutputFormat: Determines how the output files are formatted. Factory for RecordWriter objects. Examples: TextOutputFormat, SequenceFileOutputFormat, NullOutputFormat.

RecordWriter: Writes records (e.g., key-value pairs) to output files.

Fault tolerance

- Hadoop achieves fault tolerance primarily by restarting tasks. Individual task nodes (TaskTrackers) are in constant communication with the head node of the system, called the JobTracker.
- If a TaskTracker fails to communicate with the JobTracker for a period of time (by default, 1 minute), the JobTracker will assume that the TaskTracker in question has crashed.
- If the job is still in the mapping phase, then other TaskTrackers will be asked to re-execute all map tasks previously run by the failed TaskTracker.
- If the job is in the reducing phase, then other TaskTrackers will re-execute all reduce tasks that were in progress on the failed TaskTracker.

Fault tolerance (cont.)

- Mappers and reducers must be side-effect free. If Mappers and Reducers had individual identities and communicated with one another or the outside world, then restarting a task would require the other nodes to communicate with the new instances of the map and reduce tasks, and the re-executed tasks would need to reestablish their intermediate state.
- One problem with the Hadoop system is that by dividing the tasks across many nodes, it is possible for a few slow nodes (called stragglers) to rate-limit the rest of the program.
- By forcing tasks to run in isolation from one another, individual tasks do not know *where* their inputs come from. Therefore, the same input can be processed *multiple times in parallel*. As most of the tasks in a job are coming to a close, the Hadoop platform will schedule redundant copies of the remaining tasks across several nodes which do not have other work to perform. This process is known as speculative execution.

Example: word count problem

Input: scattered across one or more files

foo.txt: “sweet this is the foo file”

bar.txt: “this is the bar file”

Output: scattered across one or more files (one per reducer)

bar	1
file	2
foo	1
is	2
sweet	1
the	2
this	2

Example: pseudocode

```
mapper (position, line):
        for each word in line:
                emit (word, 1)

reducer (word, values):
        sum := 0
        for each value in values:
                sum := sum + value
        emit (word, sum)
```

Example: mapper Java code

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

Example: reducer Java code

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

Example: driver Java code

```
public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

How to run on eceubuntu in local mode

```
bash
wget https://ece.uwaterloo.ca/~wgolab/WordCount.java
export JAVA_HOME=/usr/lib/jvm/default-java
HADOOP=/opt/hadoop-3.1.2
export HADOOP_COMMON_HOME=$HADOOP
export HADOOP_HDFS_HOME=$HADOOP
export HADOOP_YARN_HOME=$HADOOP
export HADOOP_MAPRED_HOME=$HADOOP
export CLASSPATH=`$HADOOP/bin/hadoop classpath`
javac WordCount.java
jar cf wc.jar WordCount*.class
INPUT=/usr/share/mythes/th_en_US_v2.dat
# Delete the output subdirectory prior to running the next command!
$HADOOP/bin/hadoop jar wc.jar WordCount $INPUT output
ls -l output/
```