POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

MR

Marco Brambilla

marco.brambilla@polimi.it

@marcobrambi

Agenda

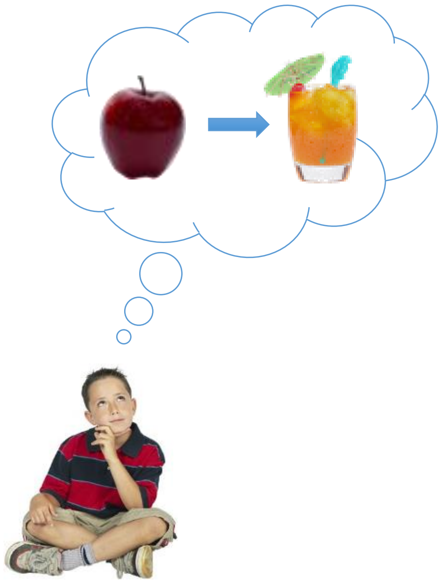
Intro & Intuition

Definition

Typical Problems

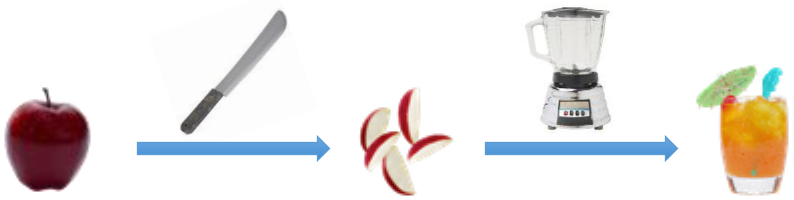
The Juice-making business

1. Wish



The Juice-making business

2. Prototype



Next Day

Apply it to all the fruits in the *fruit basket*

Industrialization

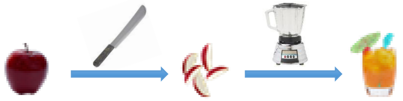
A juice making giant is making juice

- whole container of fruits

- juice of different fruits separately

Why?

The operations themselves are conceptually simple

- Making juice 
- Indexing
- Recommendations etc

But, the data to process is **HUGE!!!**

- Google processes over 50 PB of data every day

Sequential execution just won't scale up

Why?

Parallel execution achieves greater efficiency

But, parallel programming is hard

- Parallelization
  - Race Conditions
  - Debugging
- Fault Tolerance
- Data Distribution
- Load Balancing

MapReduce

> “*MapReduce is a programming model and an associated implementation for processing and generating large data sets*”

Programming model

- Abstractions to express simple computations

Implementation (existing)

- Takes care of the gory stuff: Parallelization, Fault Tolerance, Data Distribution and Load Balancing

MapReduce Advantages

- Automatic parallelization, distribution
- I/O scheduling
  - Load balancing
  - Network and data transfer optimization
- Fault tolerance
  - Handling of machine failures

Need more power: Scale out, not up!

Map? Reduce?

Mappers read in data from the filesystem, and output (typically) modified data

Reducers collect all of the mappers output on the keys, and output (typically) reduced data

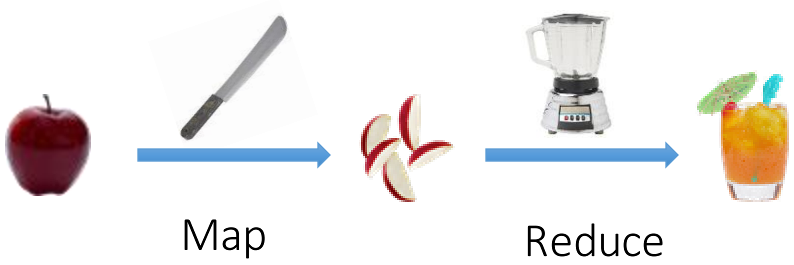
The output data is written to disk

All data is in terms of *key-value* pairs

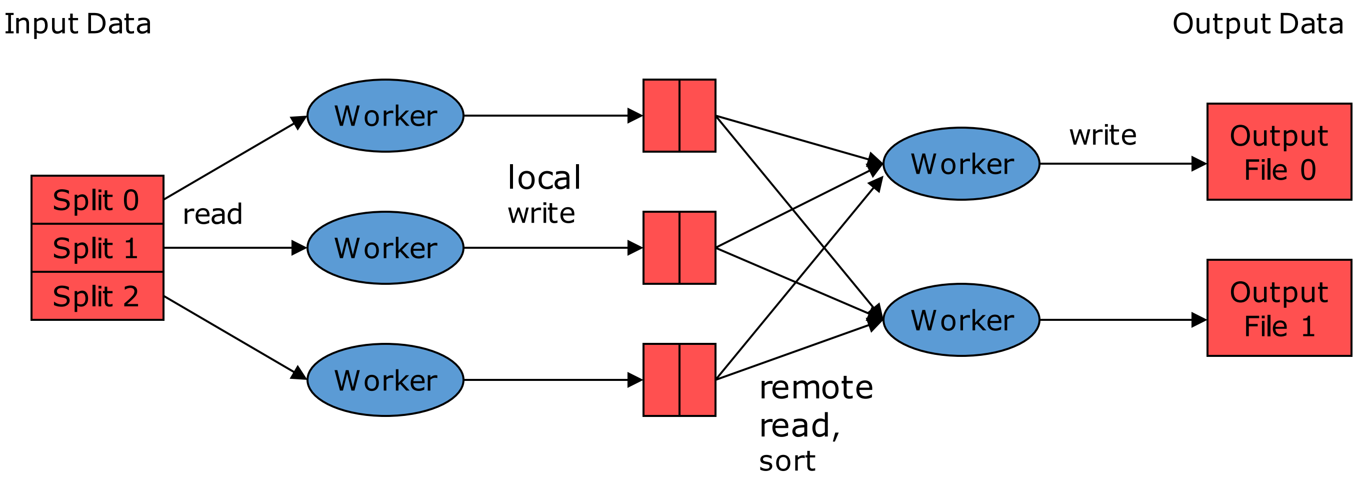
Typical problem solved by MapReduce

- Read a lot of data
- Map: extract something you care about from each record
- Shuffle and Sort
- Reduce: aggregate, summarize, filter, or transform
- Write the results

Example of Map-Reduce



MapReduce workflow



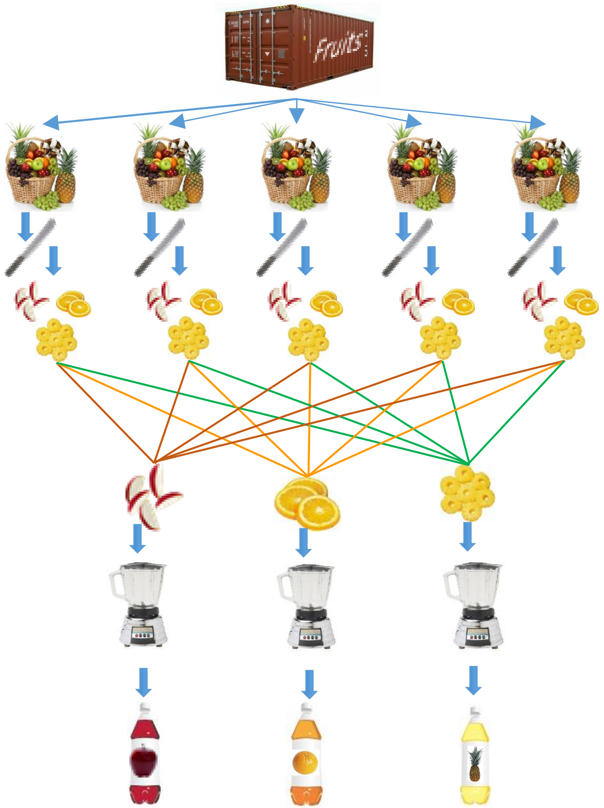
Map
extract something you care about from each record

Reduce
aggregate, summarize, filter, or transform

Marco Brambilla.

Complete MapReduce Example

A *parallel* version of the process



Marco Brambilla.

Programming Model

To generate a set of output key-value pairs from a set of input key-value pairs

- $\{ < k_i, v_i >\} \rightarrow \{ < k_o, v_o >\}$

Expressed using two abstractions:

- Map task
  - $<k_i, v_i> \rightarrow \{ < k_{int}, v_{int} > \}$
- Reduce task
  - $< k_{int}, \{v_{int}\} > \rightarrow < k_o, v_o >$

Library

- aggregates all the all intermediate values associated with the same intermediate key
- passes the intermediate key-value pairs to ***reduce*** function

Mapper

Reads in input pair <Key,Value>
Outputs a pair <K’, V’>

Let’s count number of each word in user queries (or Tweets/Blogs)
The input to the mapper will be <queryID, QueryText>:

```
<Q1,“The teacher went to the store. The store was closed; the
store opens in the morning. The store opens at 9am.” >
```

The output would be:

```
<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store,1> <the, 1> <store,
1> <was, 1> <closed, 1> <the, 1> <store,1> <opens, 1> <in, 1> <the, 1>
<morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>
```

Reducer

Accepts the Mapper output, and aggregates values on the key

- For our example, the reducer input would be:
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> **<store**, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> **<store**, 1> <opens,1> <in, 1> <the, 1> <morning, 1> <the 1> **<store**, 1> <opens, 1> <at, 1> <9am, 1>
- The output would be:
  - <The, 6> <teacher, 1> <went, 1> <to, 1> **<store, 3>** <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Key Components

Input Splitter

- Is responsible for splitting your input into multiple chunks
- These chunks are then used as input for your mappers
- Splits on logical boundaries. The default is 64MB per chunk
  - Depending on what you're doing, 64MB might be a LOT of data! You can change it
- Typically, you can just use one of the built in splitters, unless you are reading in a specially formatted file

Mapper

Reads in input pair <K,V> (a section as split by the input splitter)

Outputs a pair <K’, V’>

Ex. For our Word Count example, with the following input: “The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.”

The output would be:

<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

Reducer

Accepts the Mapper output, and collects values on the key

- All inputs with the same key *must* go to the same reducer!

Input is typically sorted, output is output exactly as is

For our example, the reducer input would be:

- <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

The output would be:

- <The, 6> <teacher, 1> <went, 1> <to, 1> <store, 3> <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Partitioner (Shuffler)

- Decides which pairs are sent to which reducer
- Default is simply:
  - Key.hashCode() % numOfReducers

- Custom partitioning is often required, for example, to produce a total order in the output. User can override to:
  - Provide (more) uniform distribution of load between reducers
  - Some values might need to be sent to the same reducer
    - Ex. To compute the relative frequency of a pair of words <W1, W2> you would need to make sure all of word W1 are sent to the same reducer
  - Binning of results
- How?
  - Should implement *Partitioner* interface
  - Set by calling `conf.setPartitionerClass(MyPart.class)`

Combiner

- Essentially an intermediate reducer
- Is optional
- Reduces output from each mapper, reducing bandwidth and sorting
- Cannot change the type of its input
  - Input types must be the same as output types

Output Committer

Is responsible for taking the reduce output, and committing it to a file

Typically, this committer needs a corresponding input splitter (so that another job can read the input)

Again, usually built-in committers are good enough, unless you need to output a special kind of file

Master

Responsible for scheduling & managing jobs

Scheduled computation should be close to the data if possible

- Bandwidth is expensive! (and slow)
- This relies on a Distributed File System (GFS / HDFS)!

If a task fails to report progress (such as reading input, writing output, etc), crashes, the machine goes down, etc, it is assumed to be stuck, and is killed, and the step is re-launched (with the same input)

The Master is handled by the framework, no user code is necessary

Master

- HDFS can replicate data to be local if necessary for scheduling
- Because our nodes are (or at least should be) deterministic
  - The Master can restart failed nodes
    - Nodes should have no side effects!
  - If a node is the last step, and is completing slowly, the master can launch a second copy of that node
    - This can be due to hardware isuses, network issues, etc.
    - First one to complete wins, then any other runs are killed

Data Typing: Writables

Are types that can be serialized / deserialized to a stream

Are required to be input/output classes, as the framework will serialize your data before writing it to disk

User can implement this interface, and use their own types for their input/output/intermediate values

There are default for basic values, like Strings, Integers, Longs, etc.

Can also handle store, such as arrays, maps, etc.

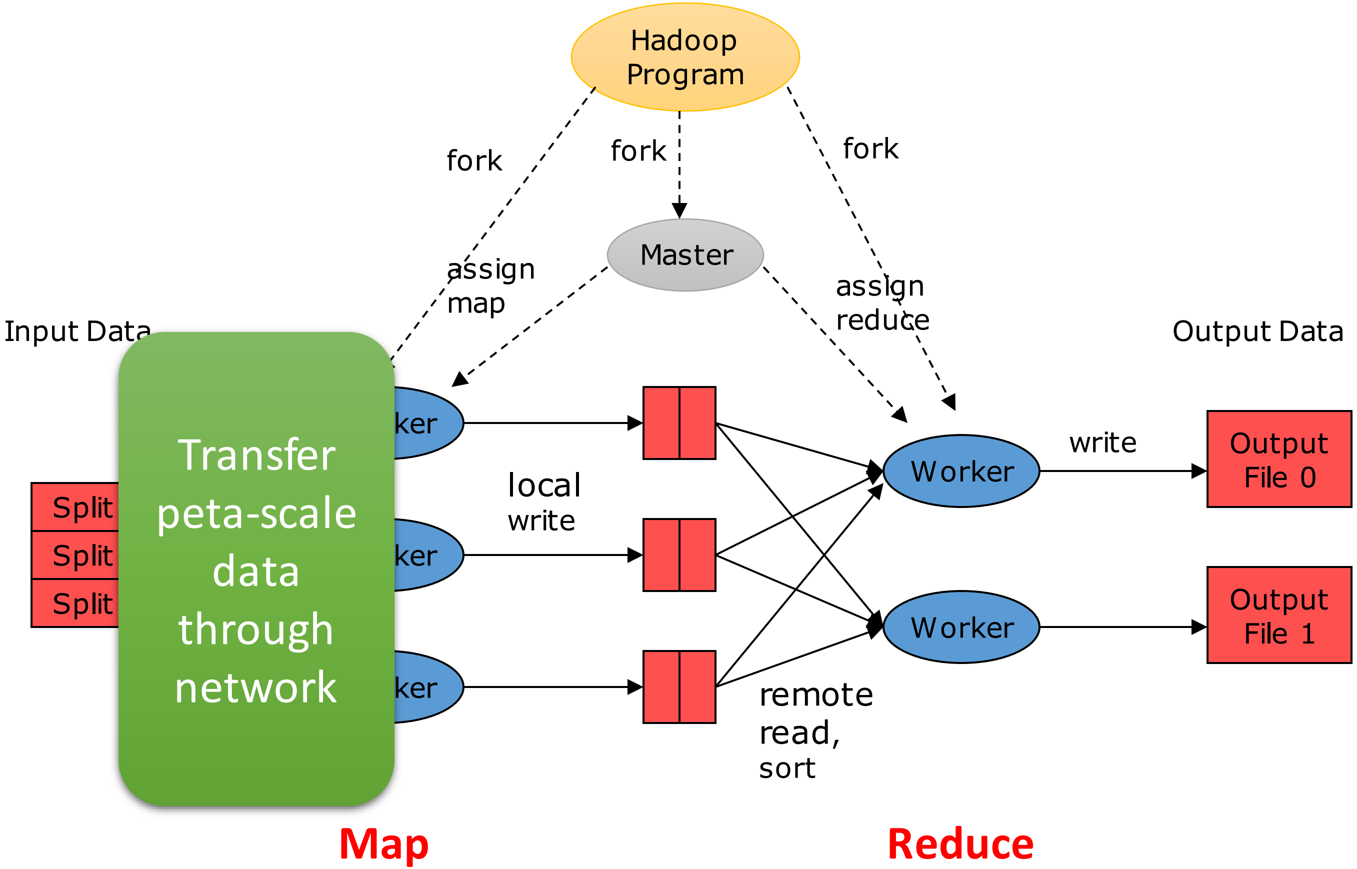
Your application needs at least six writables

- 2 for your input
- 2 for your intermediate values (Map <-> Reduce)
- 2 for your output

How it works

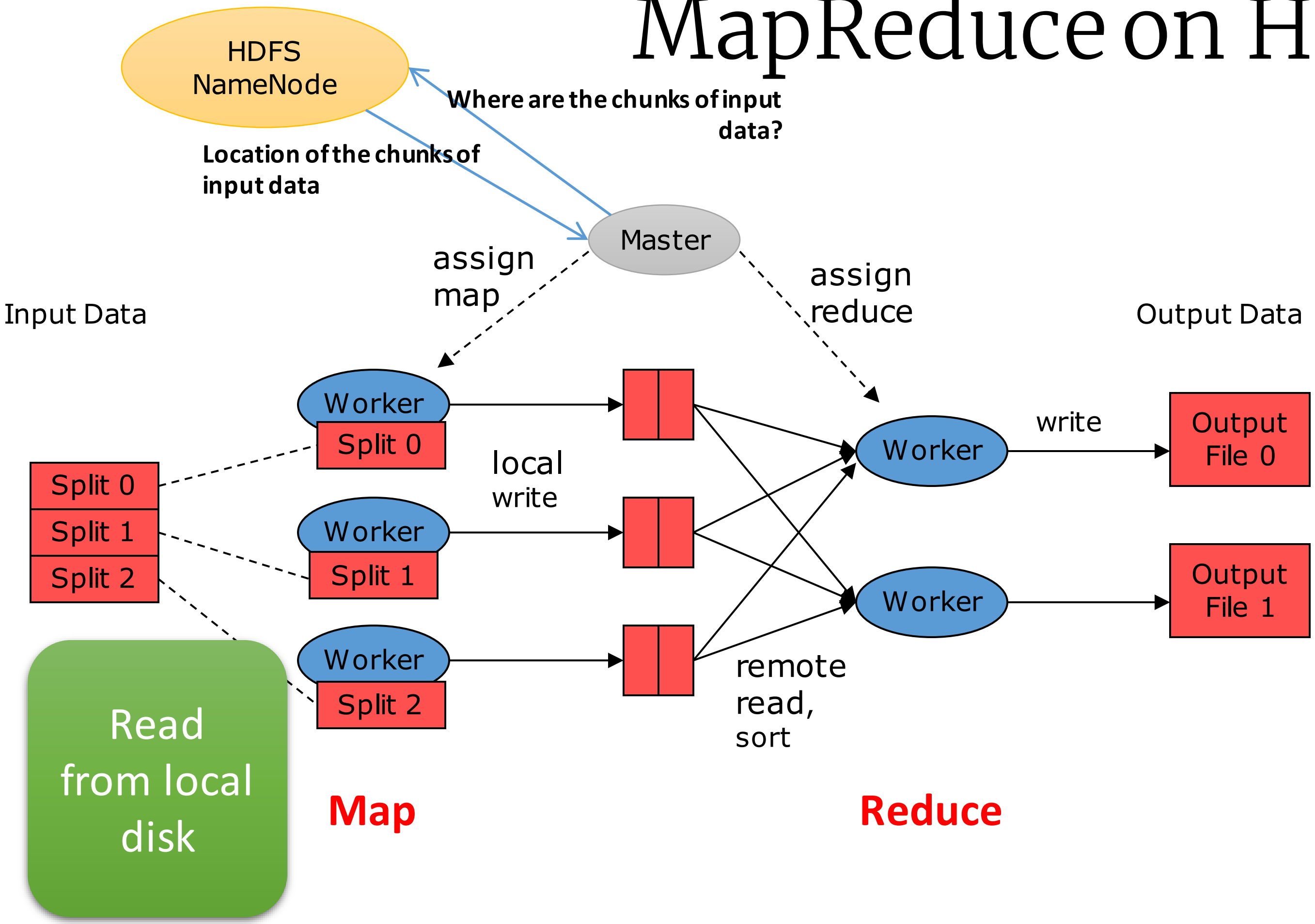
Hadoop Map-Reduce

MapReduce



Marco Brambilla.

MapReduce on HDFS



Marco Brambilla.

Execution

System determines:

- M: no. of map tasks

User specifies:

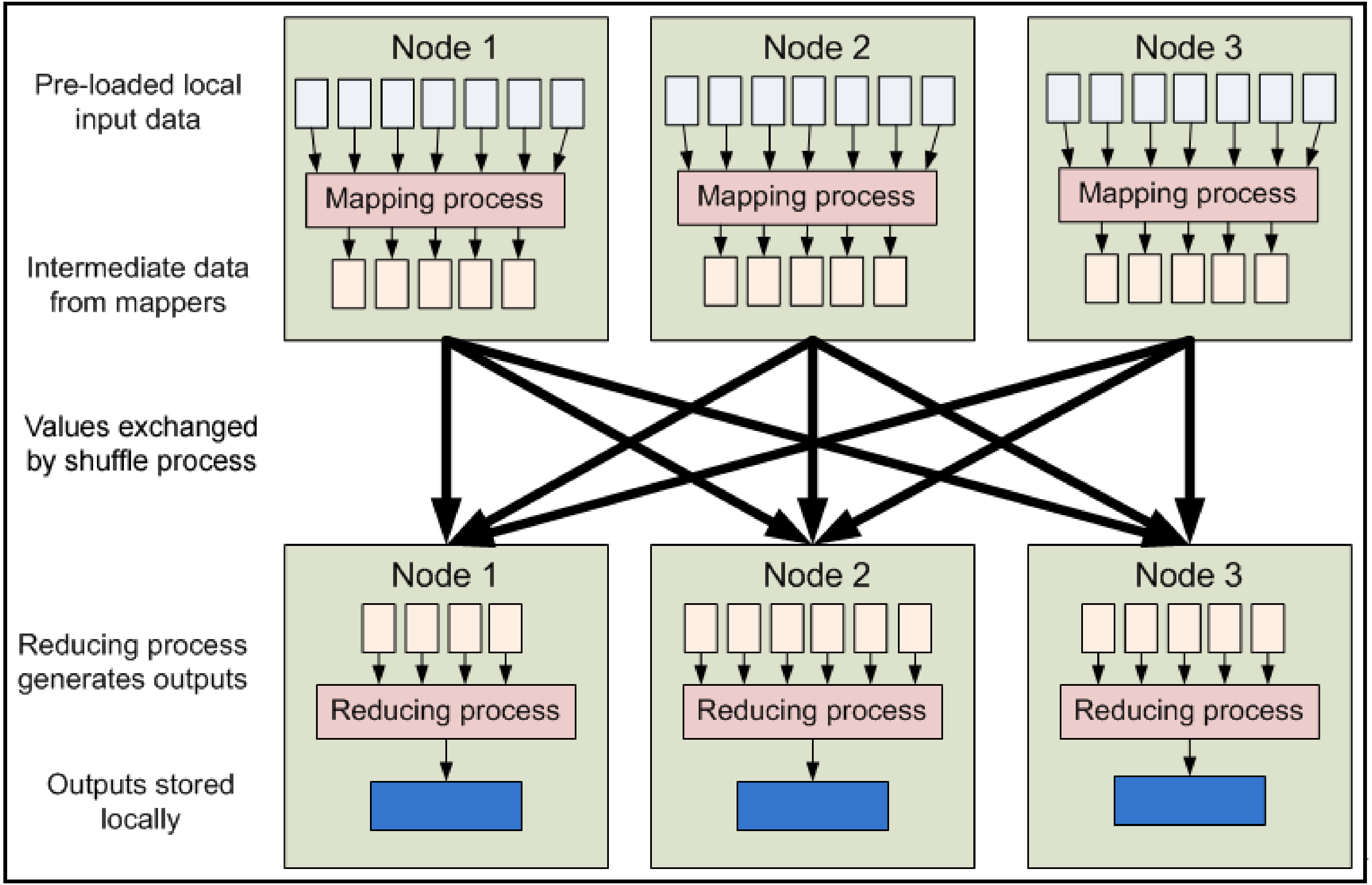
- R: no. of reduce tasks

Map Phase

- input is partitioned into M splits
- map tasks are distributed across multiple machines

Reduce Phase

- reduce tasks are distributed across multiple machines
- intermediate keys are partitioned (using partitioning function) to be processed by desired reduce task



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

Example: Word Count

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");
```

<“Sam”, “1”>, <“Apple”, “1”>,
<“Sam”, “1”>, <“Mom”, “1”>,
<“Sam”, “1”>, <“Mom”, “1”>,

```
reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

<“Sam” , [“1”,”1”,”1”]>,
<“Apple” , [“1”]>,
<“Mom” , [“1”, “1”]>

“3”
“1”
“2”

Marco Brambilla.

Complete MapReduce Job

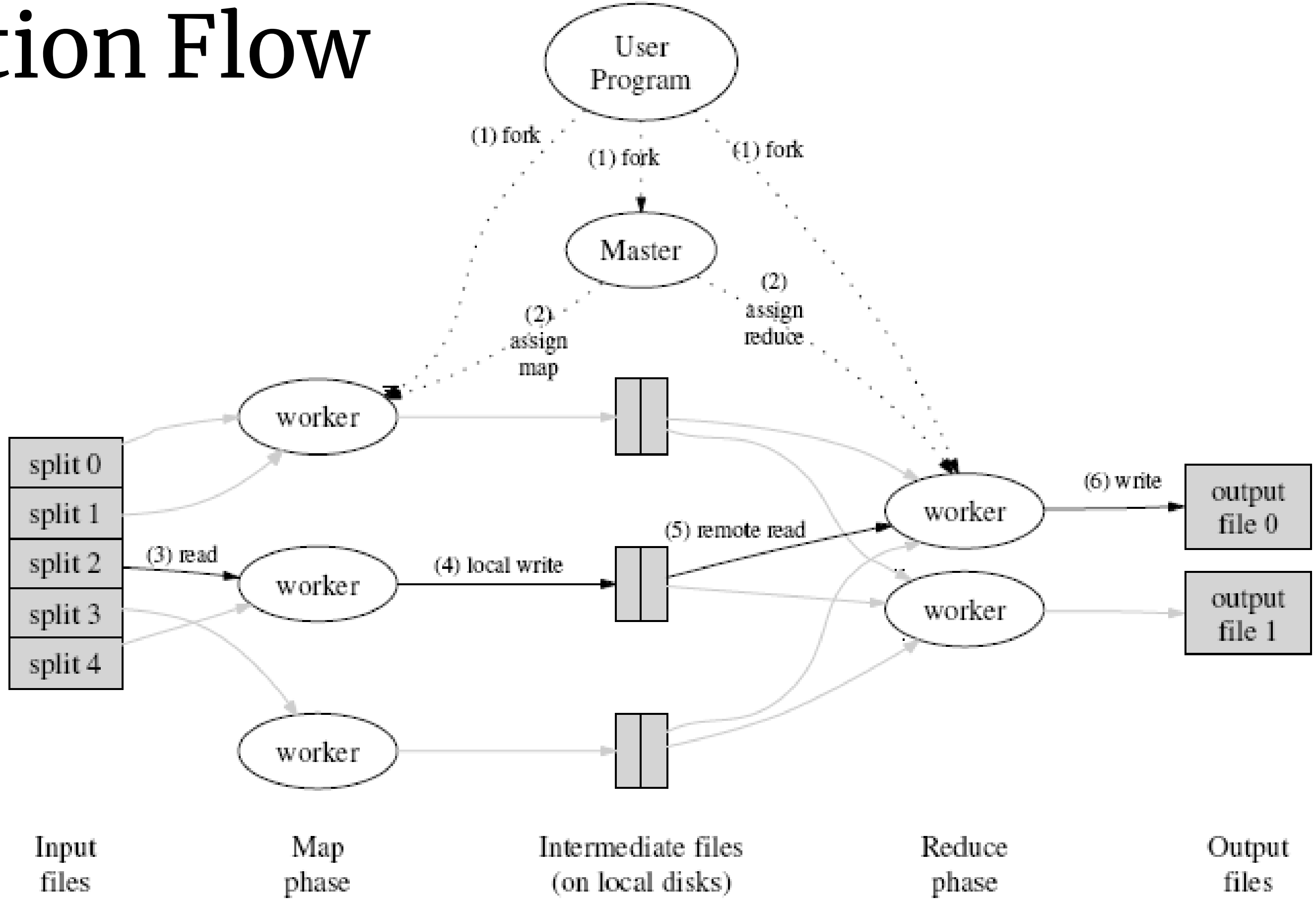
```java
public class WordCount {

  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);
  }}
```

Mapper

Reducer

Run this program as a MapReduce job

Execution Flow



Master Data Structures

- For each task
  - State { *idle*, *in-progress*, *completed* }
  - Identity of the worker machine
- For each completed map task
  - Size and location of intermediate data

Some further handy tools

Combiners
Compression
Counters
Speculation
Zero Reduces

Combiners

When *maps* produce many repeated keys

- It is often useful to do a local aggregation following the *map*
- Done by specifying a *Combiner*
- Goal is to decrease size of the transient data
- Combiners have the same interface as Reduces, and often are the same class
- Combiners must **not** side effects, because they run an intermdiate number of times
- In *WordCount*, `conf.setCombinerClass(Reduce.class);`

Compression

Compressing the outputs and intermediate data will often yield huge performance gains

- Can be specified via a configuration file or set programmatically
- Set *mapred.output.compress* to *true* to compress job output
- Set *mapred.compress.map.output* to *true* to compress map outputs

Compression Types (*mapred(.map)?.output.compression.type)*

- "block" - Group of keys and values are compressed together
- "record" - Each value is compressed individually
- Block compression is almost always best

Compression Codecs *(mapred(.map)?.output.compression.codec)*

- Default (zlib) - slower, but more compression
- LZO - faster, but less compression

Counters

Often Map/Reduce applications have countable events

For example, framework counts records in to and out of Mapper and Reducer

To define user counters:

```
static enum Counter {EVENT1, EVENT2};
reporter.incrCounter(Counter.EVENT1, 1);
```

Define nice names in a MyClass_Counter.properties file

```
CounterGroupName=MyCounters
EVENT1.name=Event 1
EVENT2.name=Event 2
```

Speculative execution

The framework can run multiple instances of slow tasks

- Output from instance that finishes first is used
- Controlled by the configuration variable *mapred.speculative.execution*
- Can dramatically bring in long tails on jobs

Zero Reduces

Frequently, we only need to run a filter on the input data
- No sorting or shuffling required by the job
- Set the number of reduces to 0
- Output from maps will go directly to OutputFormat and disk

Fault Tolerance

- Worker failure – handled via re-execution
  - Identified by no response to heartbeat messages
  - *In-progress* and *Completed* map tasks are re-scheduled
  - Workers executing reduce tasks are notified of re-scheduling
  - *Completed* reduce tasks are not re-scheduled
- Master failure
  - Rare
  - Can be recovered from checkpoints
  - All tasks abort

Disk Locality

Leveraging HDFS

Map tasks are scheduled close to data

- on nodes that have input data
- if not, on nodes that are nearer to input data
  - Ex. Same network switch

Conserves network bandwidth

Task Granularity

- No. of map tasks > no. of worker nodes
  - Better load balancing
  - Better recovery
- But, increases load on Master
  - More scheduling decisions
  - More states to be saved
- M could be chosen w.r.t to block size of the file system
  - to effectively leverage locality
- R is usually specified by users
  - Each reduce task produces one output file

Stragglers

- Slow workers delay completion time
  - Bad disks with soft errors
  - Other tasks eating up resources
  - Strange reasons like processor cache being disabled
- Start back-up tasks as the job nears completion
  - First task to complete is considered

Refinement: Partitioning Function

- Identifies the desired reduce task
  - Given the intermediate key and R
- Default partitioning function
  - $hash(key) \textbf{ mod } R$
- Important to choose well-balanced partitioning functions
  - If not, reduce tasks may delay completion time

Refinement: Combiner Function

- Mini-reduce phase before the intermediate data is sent to reduce
- Significant repetition of intermediate keys possible
    - Merge values of intermediate keys before sending to reduce tasks
- Similar to reduce function
- Saves network bandwidth

Refinement: Skipping Bad Records

- Map/Reduce tasks may fail on certain records due to bugs
  - Ideally, debug and fix
  - Not possible if third-party code is buggy
- When worker dies, Master is notified of the record
- If more than one worker dies on the same record
  - Master re-schedules the task and asks to skip the record

New Trend: Disk-locality Irrelevant

Assumes disk bandwidth exceeds network bandwidth

Network speeds fast improving

Disk speeds have stagnated

Next step: attain memory-locality

Scheduling

By default, Hadoop uses FIFO to schedule jobs.

Alternate scheduler options:

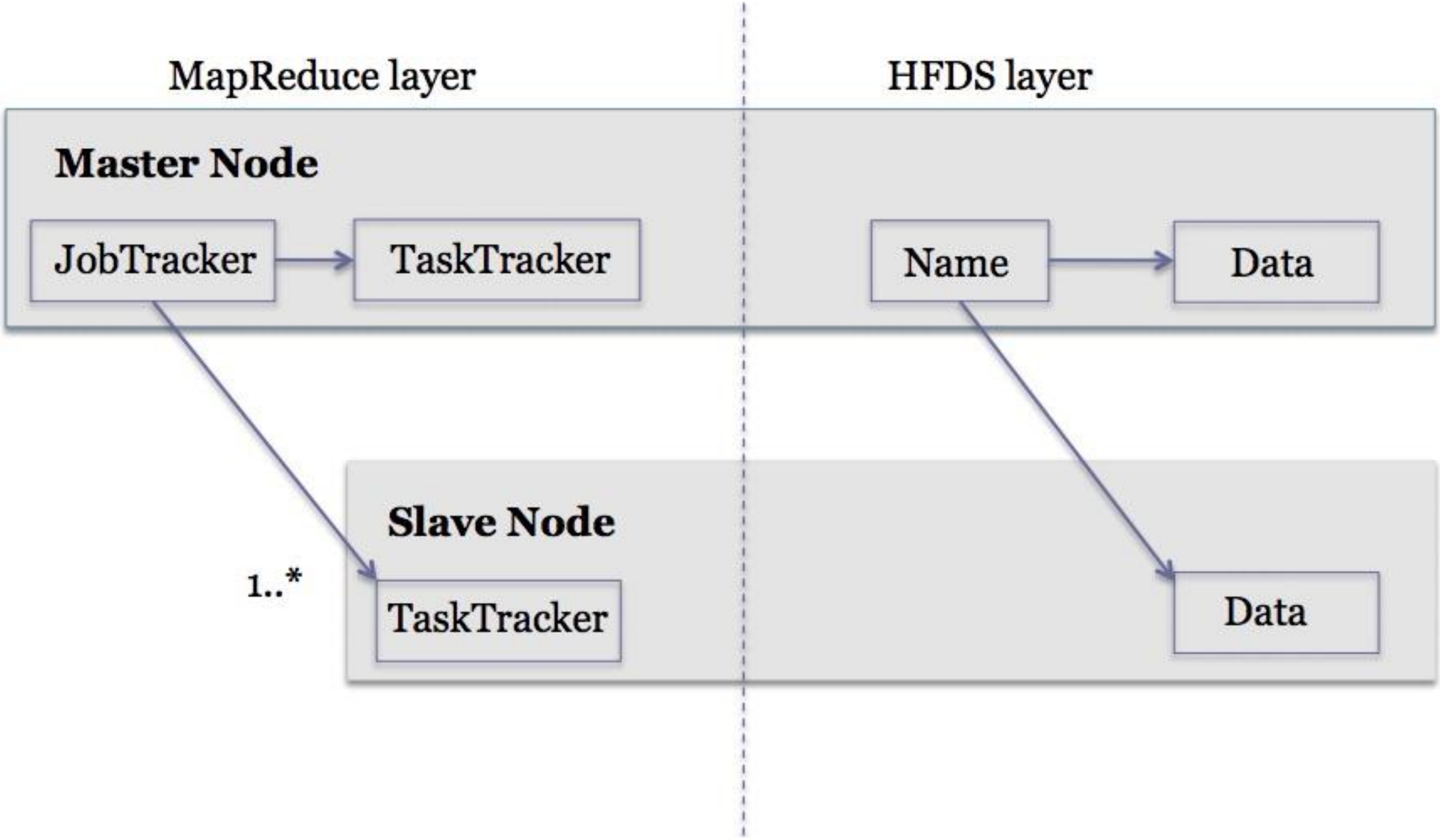
capacity and *fair*

Capacity Scheduler

- Developed by Yahoo
- Jobs are submitted to queues
- Jobs can be prioritized
- Queues are allocated a fraction of the total resource capacity
- Free resources are allocated to queues beyond their total capacity
- No preemption once a job is running

Fair scheduler

- Developed by Facebook
- Provides fast response times for small jobs
- Jobs are grouped into Pools
- Each pool assigned a guaranteed minimum share
- Excess capacity split between jobs
- By default, jobs that are uncategorized go into a default pool.
- Pools have to specify the minimum number of map slots, reduce slots, and a limit on the number of running jobs


MapReduce layer
HFDS layer
Master Node
JobTracker
TaskTracker
Name
Data
Slave Node
1..*
TaskTracker
Data

Conclusion

- Easy to use scalable programming model for large-scale data processing on clusters
  - Allows users to focus on computations
- Hides issues of
  - parallelization, fault tolerance, data partitioning & load balancing
- Achieves efficiency through disk-locality
- Achieves fault-tolerance through replication

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

Thanks

Marco Brambilla

marco.brambilla@polimi.it

@marcobrambi