

INGI2145: CLOUD COMPUTING (Fall 2015)


MapReduce

8 October 2015

Announcements

- HW1 is available
 - Due **6th** November at 11:59pm
 - **Please start early!**
 - Debugging Hadoop programs can be a bit tricky
 - You should try to attend tomorrow's lab session
 - Ask questions in case you encounter problems
 - Try to finish a few days before the deadline
- Tomorrow's lab session in INTEL room
 - Introduction to Hadoop and exercises

Plan for today

- Introduction 
 - Census example
- MapReduce architecture
 - Programming model, data flow
 - Details, fault tolerance, challenges, etc.
- Single-pass algorithms in MapReduce
 - Filtering, aggregation, intersections and joins
 - Sorting
 - Strengths and weaknesses
- A brief overview of Hadoop

Analogy: National census

- Suppose we have 10,000 employees, whose job is to collate census forms and to determine how many people live in each city
- How would you organize this task?



National census “data flow”

Making things more complicated

- Suppose people take vacations, get sick, work at different rates
- Suppose some forms are incorrectly filled out and require corrections or need to be thrown away
- What if the supervisor gets sick?
- How big should the piles be?
- How do we monitor progress?
- ...

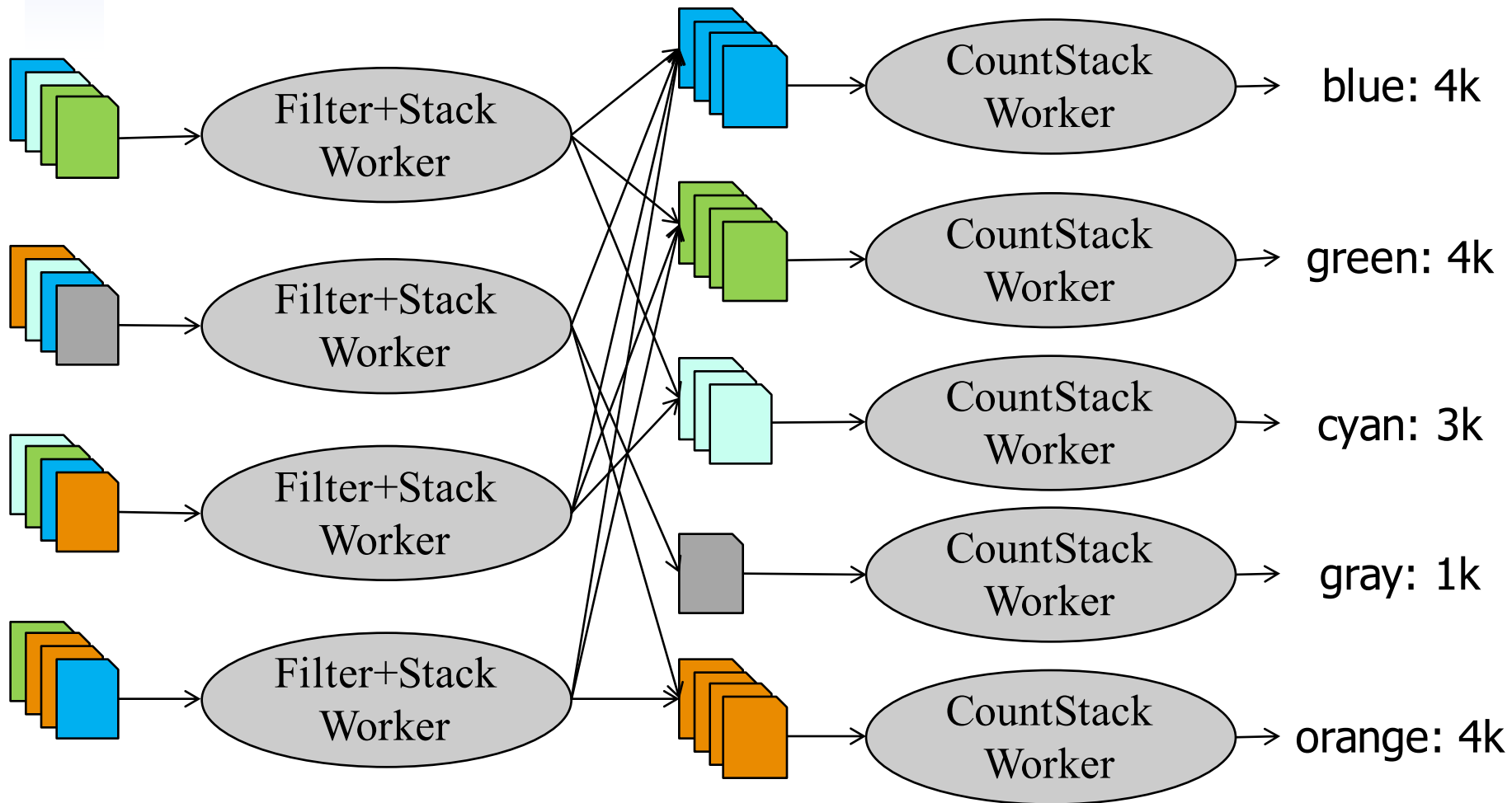
A bit of introspection

- What is the main challenge?
 - Are the individual tasks complicated?
 - If not, what makes this so challenging?
- How resilient is our solution?
- How well does it balance work across employees?
 - What factors affect this?
- How general is the set of techniques?

I don't want to deal with all this!!!

- Wouldn't it be nice if there were some system that took care of all these details for you?
- Ideally, you'd just tell the system what needs to be done
- **That's the MapReduce framework:**
 - A distributed data processing model and execution environment that runs on large clusters of machines

Abstracting into a digital data flow



Abstracting once more

- There are two kinds of workers:
 - Those that take input data items and produce output items for the “stacks”
 - Those that take the stacks and **aggregate** the results to produce outputs on a per-stack basis
 - We'll call these:
 - **map**: takes (item_key, value), produces one or more (stack_key, value') pairs
 - **reduce**: takes (stack_key, {set of value'}), produces one or more output results – typically (stack_key, agg_value)
- We will refer to this key
as the **reduce key**

Why MapReduce?

- Scenario:

- You have a huge amount of data, e.g., all the Google searches of the last three years
- You would like to perform a computation on the data, e.g., find out which search terms were the most popular

- How would you do it?




- Analogy to the census example:

- The computation isn't necessarily difficult, but parallelizing and distributing it, as well as handling faults, is challenging

- Idea: A programming model!

- Write a simple program to express the (simple) computation, and let the system runtime do all the hard work

Plan for today

- Introduction 
 - Census example 
- MapReduce architecture 
 - Programming model, data flow
 - Details, fault tolerance, challenges, etc.
- Single-pass algorithms in MapReduce
 - Filtering, aggregation, intersections and joins
 - Sorting
 - Strengths and weaknesses
- A brief overview of Hadoop

What is MapReduce?

- A famous distributed programming model
- In many circles, considered *the* key building block for much of Google's data analysis
 - A programming language built on it: Sawzall, <http://labs.google.com/papers/sawzall.html>
 - ... *Sawzall has become one of the most widely used programming languages at Google. ... [O]n one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 18,636 failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of 3.2×10^{15} bytes of data (2.8PB) and wrote 9.9×10^{12} bytes (9.3TB).*
 - Other similar languages: Yahoo's Pig Latin and Pig; Microsoft's Dryad
- Cloned in open source: Hadoop, <http://hadoop.apache.org/>

The MapReduce programming model

- Modeled after simple functional programming primitives:
 - **map** (apply function to all items in a collection) and
 - **reduce** (apply function to set of items with a common key)
- We start with:
 - A user-defined function to be applied to all data,
map: $(\text{key}, \text{value}) \rightarrow (\text{reduce_key}, \text{value})$
 - Another user-specified operation
reduce: $(\text{reduce_key}, \{\text{set of values}\}) \rightarrow \text{result}$
 - A set of n nodes, each with data
- All nodes run map on all of their data, producing new data with reduce keys
 - This data is collected by key, then **shuffled**, and finally **reduced**
 - Dataflow is through temporary files

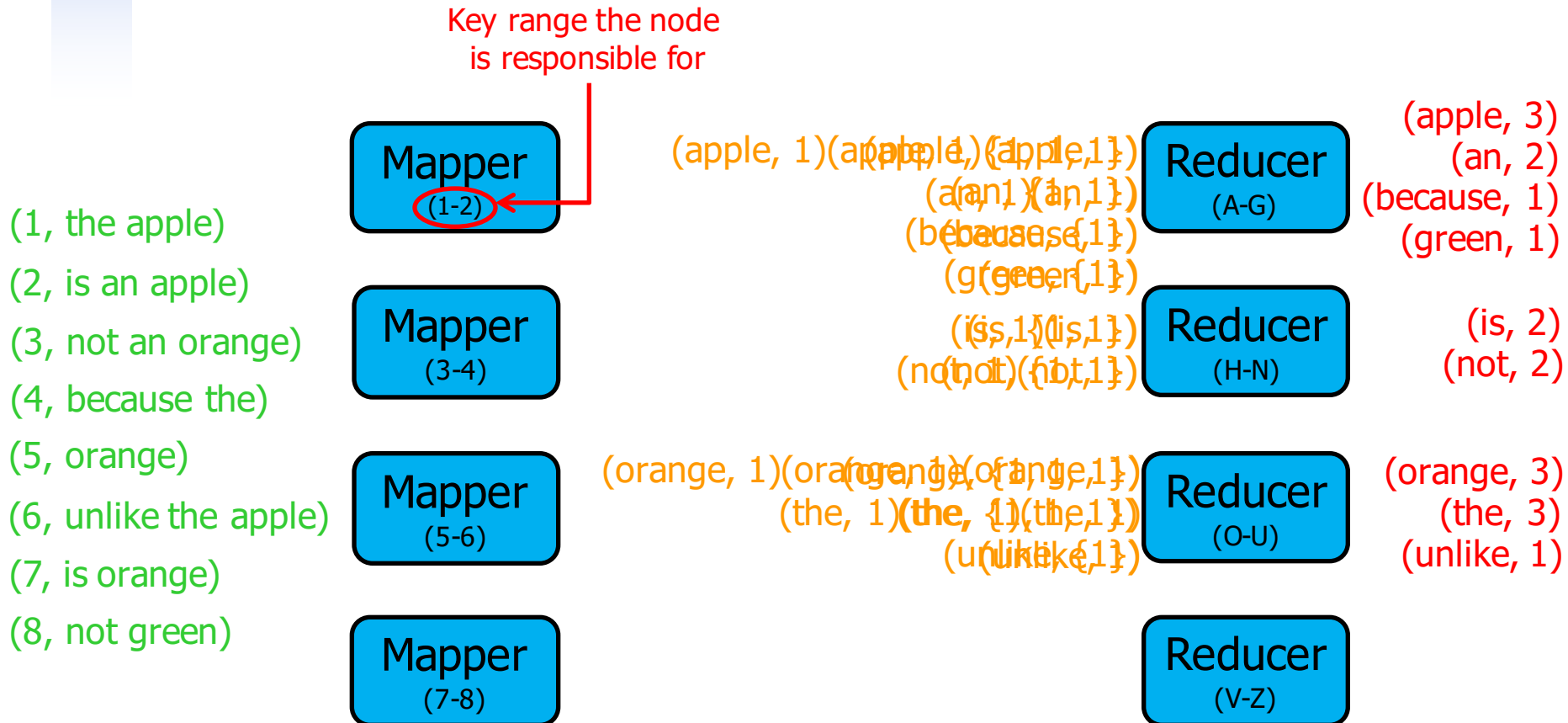
Simple example: Word count

```
map(String key, String value) {  
  // key: document name, line no  
  // value: contents of line  
  for each word w in value:  
    emit(w, 1)  
}
```

```
reduce(String key, Iterator values) {  
  // key: a word  
  // values: a list of counts  
  int result = 0;  
  for each v in values:  
    result += v;  
  emit(key, result)  
}
```

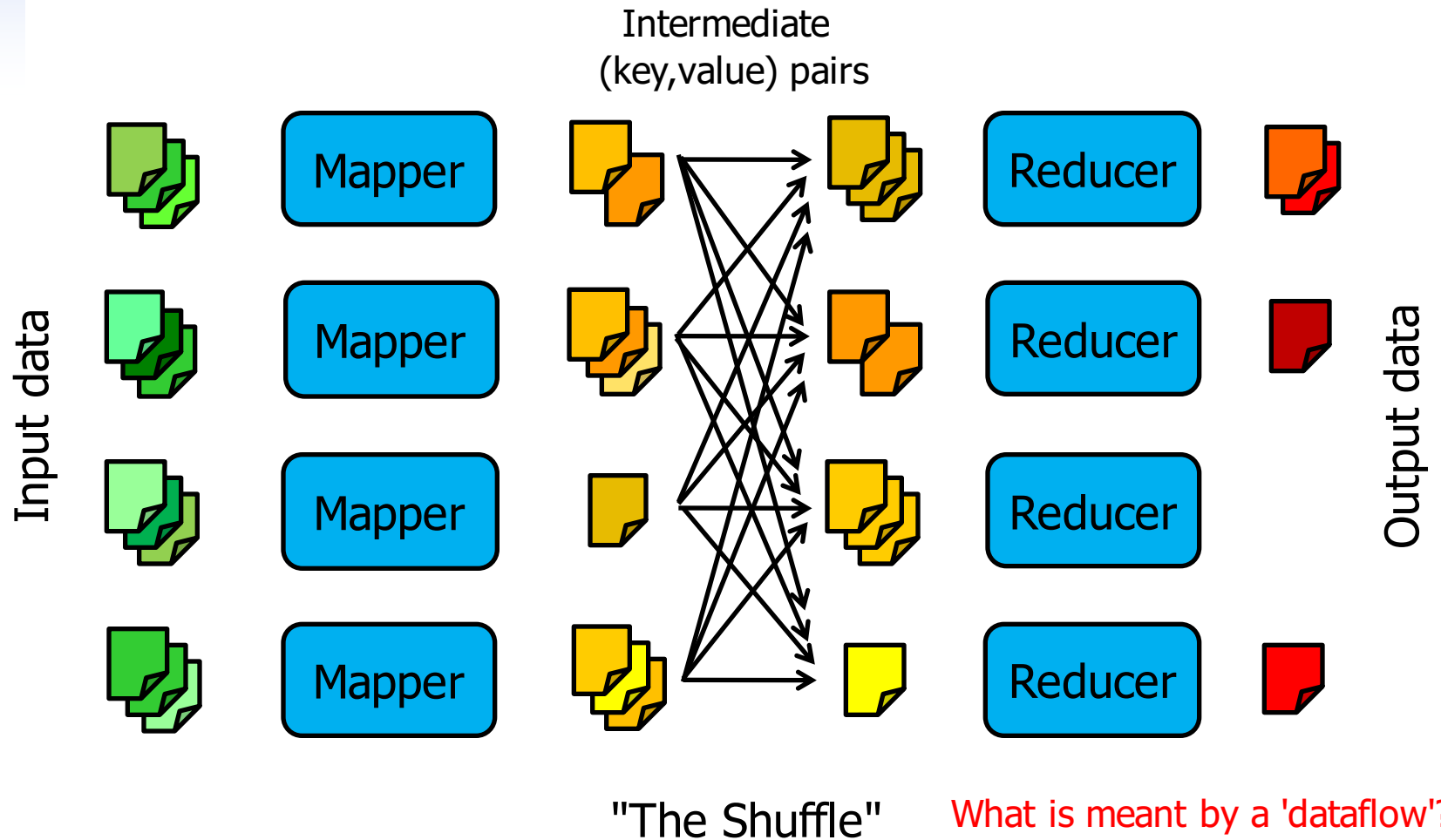
- Goal: Given a set of documents, count how often each word occurs
 - Input: Key-value pairs (document:lineNumber, line of text)
 - Output: Key-value pairs (word, #occurrences)
 - What should be the intermediate key-value pairs?

Simple example: Word count



- Each mapper receives some of the KV-pairs as input
- The mappers process the KV-pairs one by one
- Each KV-pair output by the mapper is sent to the reducer that is responsible for it
- The reducers sort their input by key and group it
- The reducers process their input one group at a time

MapReduce dataflow



What is meant by a 'dataflow'?
What makes this so scalable?

More examples

- Distributed grep – all lines matching a pattern
 - Map: filter by pattern
 - Reduce: output set
- Count URL access frequency
 - Map: output each URL as key, with count 1
 - Reduce: sum the counts
- Reverse web-link graph
 - Map: output *(target, source)* pairs when link to target found in source
 - Reduce: concatenate values and emit *(target, list(source))*
- Inverted index
 - Map: emit *(word, documentID)*
 - Reduce: combine these into *(word, list(documentID))*

Common mistakes to avoid

■ Mapper and reducer should be **stateless**

- Don't use static variables - after map + reduce return, they should remember nothing about the processed data!
- Reason: No guarantees about which key-value pairs will be processed by which workers!

```
HashMap h = new HashMap();  
map(key, value) {  
    if (h.contains(key)) {  
        h.add(key, value);  
        emit(key, "X");  
    }  
}
```

Wrong!

■ Don't try to do your own **I/O!**

- Don't try to read from, or write to, files in the file system
- The MapReduce framework does all the I/O for you:

```
map(key, value) {  
    File foo =  
        new File("xyz.txt");  
    while (true) {  
        s = foo.readLine();  
        ...  
    }  
}
```

Wrong!

- All the incoming data will be fed as arguments to map and reduce
- Any data your functions produce should be output via emit

More common mistakes to avoid

```
map(key, value) {  
    emit("FOO", key + " " + value);  
}
```

Wrong!


```
reduce(key, value[]) {  
    /* do some computation on  
    all the values */  
}
```

- Mapper must not map too much data to the same key
 - In particular, don't map *everything* to the same key!
 - Otherwise the reduce worker will be overwhelmed!
 - It's okay if some reduce workers have more work than others
 - Example: In WordCount, the reduce worker that works on the key 'and' has a lot more work than the reduce worker that works on 'rare'.

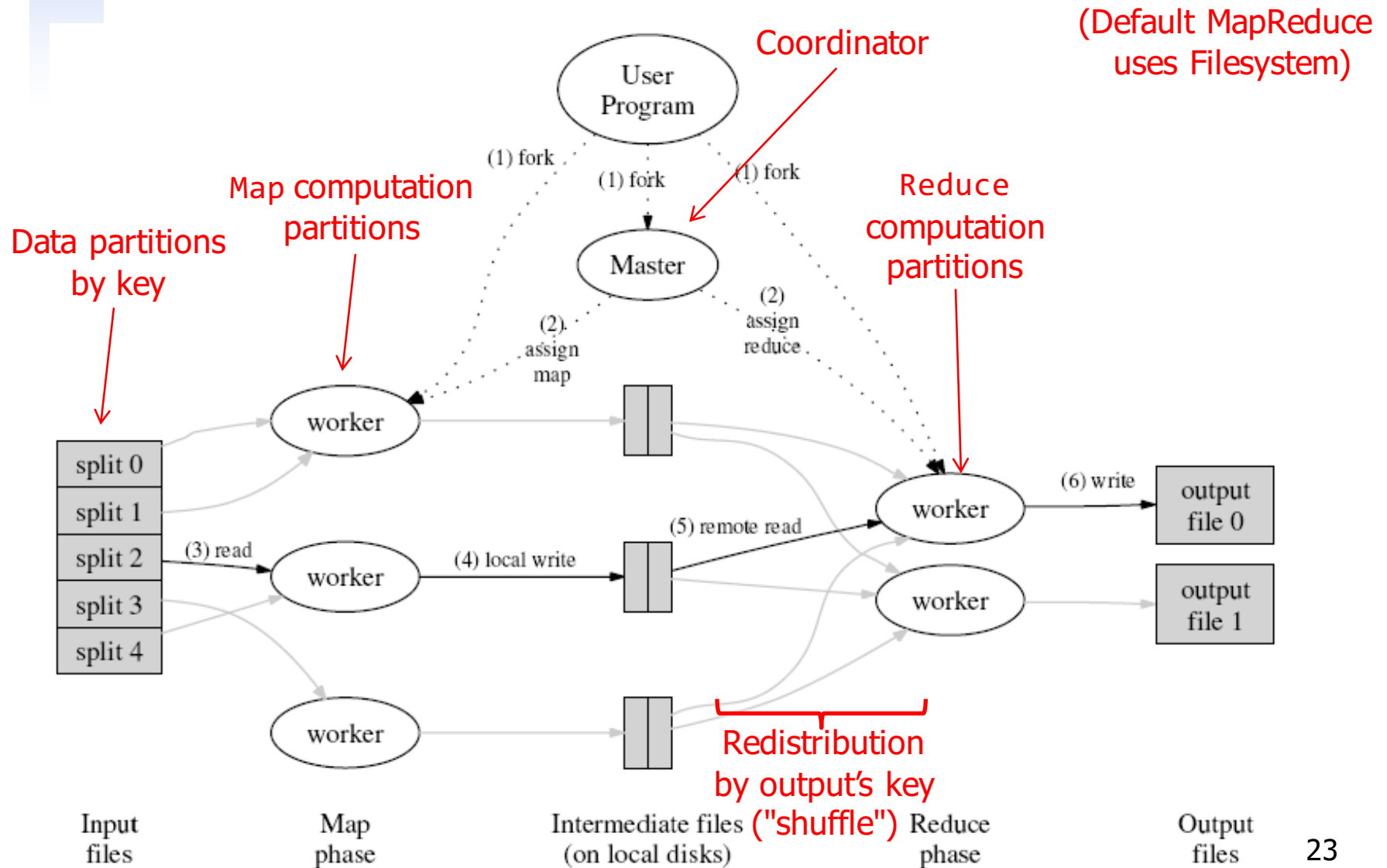
Designing MapReduce algorithms

- Key decision: What should be done by map, and what by reduce?
 - map can do something to each individual key-value pair, but it can't look at other key-value pairs
 - Example: Filtering out key-value pairs we don't need
 - map can emit more than one intermediate key-value pair for each incoming key-value pair
 - Example: Incoming data is text, map produces (word,1) for each word
 - reduce can aggregate data; it can look at multiple values, as long as map has mapped them to the same (intermediate) key
 - Example: Count the number of words, add up the total cost, ...
- Need to get the intermediate format right!
 - If reduce needs to look at several values together, map must emit them using the same key!

Plan for today

- Introduction ✓
 - Census example ✓
- MapReduce architecture
 - Programming model, data flow ✓
 - Details, fault tolerance, challenges, etc. 
- Single-pass algorithms in MapReduce
 - Filtering, aggregation, intersections and joins
 - Sorting
 - Strengths and weaknesses
- A brief overview of Hadoop

More details on the MapReduce data flow



Some additional details

- To make this work, we need a few more parts...
- The **file system** (distributed across all nodes):
 - Stores the inputs, outputs, and temporary results
- The **driver program** (executes on one node):
 - Specifies where to find the inputs, the outputs
 - Specifies what mapper and reducer to use
 - Can customize behavior of the execution
- The **runtime system** (controls nodes):
 - Supervises the execution of tasks

Some details

- Fewer computation partitions than data partitions
 - All data is accessible via a distributed filesystem with replication
 - Worker nodes produce data in key order (makes it easy to merge)
 - The master is responsible for scheduling, keeping all nodes busy
 - The master knows how many data partitions there are, which have completed – atomic commits to disk
- **Locality:** Master tries to do work on nodes that have replicas of the data
- Master can deal with stragglers (slow machines) by re-executing their tasks somewhere else

What if a worker crashes?

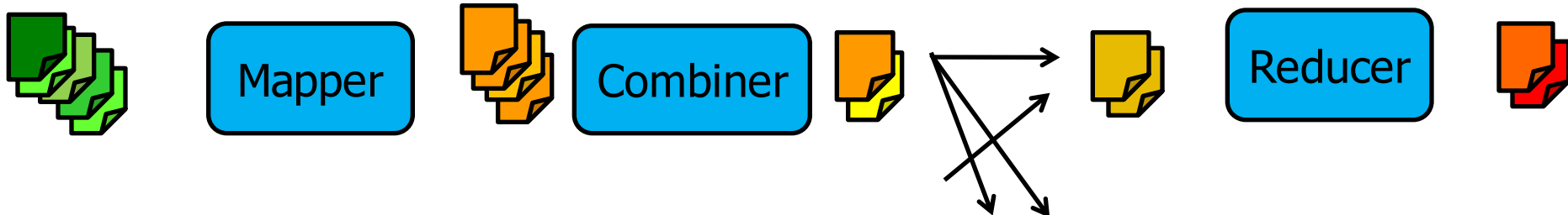
- We rely on the file system being shared across all the nodes
- Two types of (crash) faults:
 - Node wrote its output and then crashed
 - Here, the file system is likely to have a copy of the complete output
 - Node crashed before finishing its output
 - The master sees that the job isn't making progress, and restarts the job elsewhere on the system
- (Of course, we have fewer nodes to do work...)
- But what if the master crashes?

Other challenges

- **Locality**
 - Try to schedule map task on machine that already has data
- **Task granularity**
 - How many map tasks? How many reduce tasks?
- **Dealing with stragglers**
 - Schedule some backup tasks
- **Saving bandwidth**
 - E.g., with combiners (see later)

Combiners

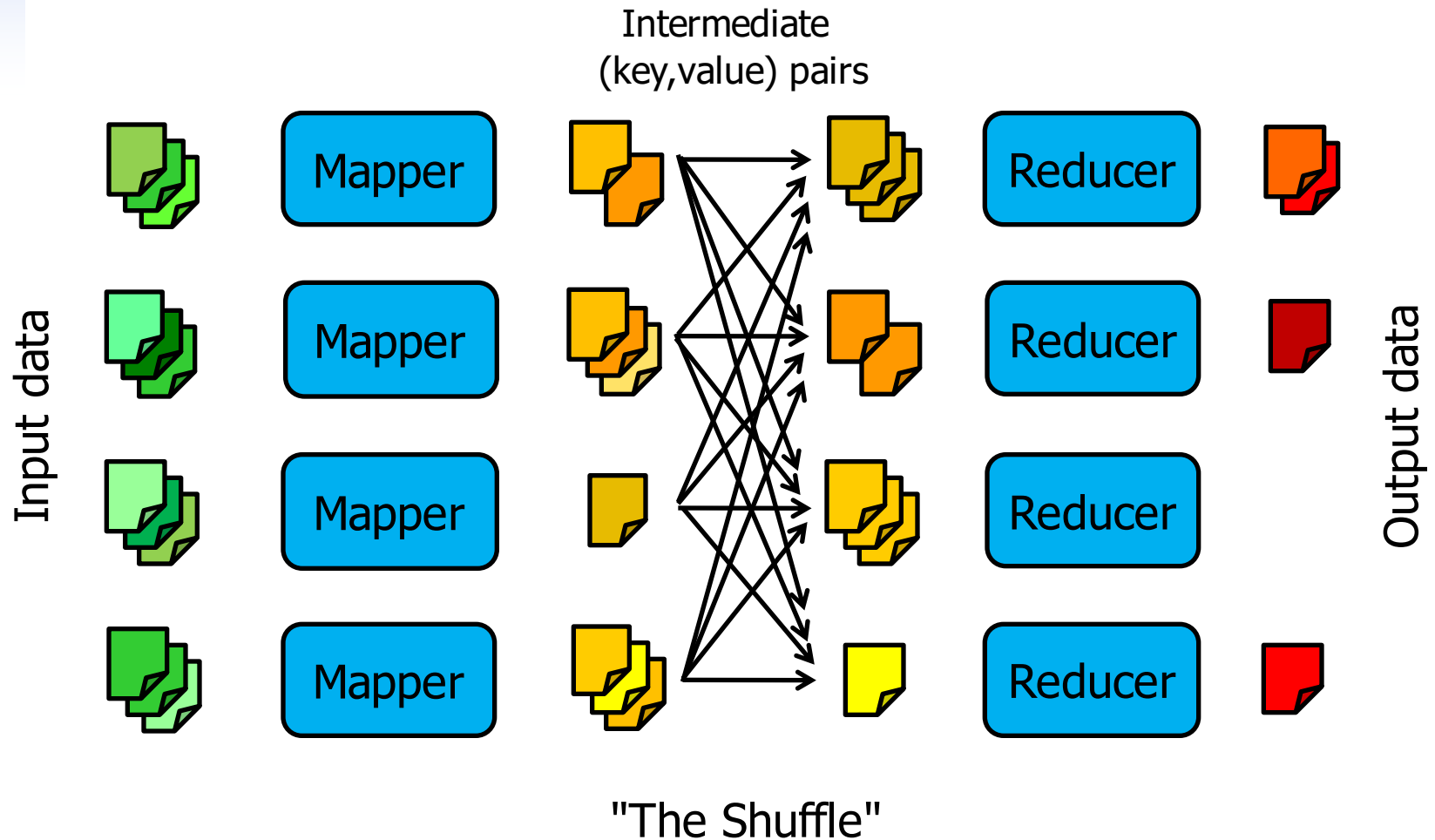
- Optional component that can be inserted after the mappers
 - Input: All data emitted by the mappers on a given node
- Why is this useful?
 - Suppose your mapper counts words by emitting $(xyz, 1)$ pairs for each word xyz it finds
 - If a word occurs many times, it is much more efficient to pass (xyz, k) to the reducer, than passing k copies of $(xyz, 1)$



Scale and MapReduce

- From a particular Google paper on a language built over MapReduce:
 - ... Sawzall has become one of the most widely used programming languages at Google. ...
[O]n one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each.
While running those jobs, **18,636** failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of **3.2×10^{15} bytes of data (2.8PB)** and wrote **9.9×10^{12} bytes (9.3TB)**.
- We will see some of MapReduce-based languages, like Pig Latin, later in the semester

Recap: MapReduce dataflow



Recap: MapReduce

```
map(key:URL, value:Document)
{
```

These types depend on
the input data

```
    String[] words = value.split(" ");
    for each w in words
        emit(w, 1);
}
```

Produces intermediate
key-value pairs that
are sent to the reducer

These types can be (and often are)
different from the ones in map()


reduce gets all the
intermediate values
with the same rkey

```
reduce(rkey:String, rvalues:Integer[])
{
    Integer result = 0;
    foreach v in rvalues
        result = result + v;
    emit(rkey, result);
}
```

Both map() and reduce() are
stateless: Can't have a global
variable that is preserved
across invocations!

All key-value pairs emitted
by the reducer are added to
the final output

Plan for today

- Introduction ✓
 - Census example ✓
- MapReduce architecture ✓
 - Programming model, data flow ✓
 - Details, fault tolerance, challenges, etc. ✓
- Single-pass algorithms in MapReduce 
 - Filtering, aggregation, intersections and joins
 - Sorting
 - Strengths and weaknesses
- A brief overview of Hadoop

The basic idea

- Let's consider single-pass algorithms
- Need to take the algorithm and break it into filter/collect/aggregate steps
 - Filter/collect becomes part of the **map** function
 - Collect/aggregate becomes part of the **reduce** function
- Note that sometimes we may need multiple map / reduce stages – chains of maps and reduces
- Let's see some examples

Filtering algorithms

- Goal: Find lines/files/tuples with a particular characteristic
- Examples:
 - grep Web logs for requests to *.uclouvain.ne/*
 - find in the Web logs the hostnames accessed by 192.168.2.1
 - locate all the files that contain the words 'Beer' and 'Fries'
- Generally: **map** does most of the work, **reduce** may simply be the identity

Aggregation algorithms

- Goal: Compute the maximum, the sum, the average, ..., over a set of values
- Examples:
 - Count the number of requests to *.uclouvain.be/*
 - Find the most popular domain
 - Average the number of requests per page per Web site
- Often: **map** may be simple or the identity

A more complex example

- Goal: Billing for a CDN like Amazon CloudFront
 - Input: Log files from the edge servers. Two files per domain:
 - access_log-www.foo.com-20111006.txt: HTTP accesses
 - ssl_access_log-www.foo.com-20111006.txt: HTTPS accesses
 - Example line:

```
158.130.53.72 - - [06/Oct/2011:16:30:38 -0400] "GET /largeFile.ISO HTTP/1.1" 200 8130928734 "-" "Mozilla/5.0 (compatible; MSIE 5.01; Win2000)"
```
 - Mapper receives (filename, line) tuples
 - Billing policy (simplified):
 - Billing is based on a mix of request count and data traffic (why?)
 - 10,000 HTTP requests cost \$0.0075
 - 10,000 HTTPS requests cost \$0.0100
 - One GB of traffic costs \$0.12
 - Desired output is a list of (domain, grandTotal) tuples

Intersections and joins

- Intersection: take like items, return only one copy of each item
- Join: take unlike items, combine if they satisfy a relationship (like meet a certain predicate)
- Examples:
 - [Intersection] Find all documents with the words “data” and “centric” given an inverted index
 - [Join] Find all professors and students in common courses and return the pairs <professor,student> for those cases

Intersection

Generally easy:

- send the “like” objects to the same reducer
- only return one result for each unique item in the set

```
reduce(key, values) {  
    Set items = new Set();  
    for (v in values) {  
        if (!items.contains(v))  
            emit(k, v);  
        items.add(v);  
    }  
}
```

Join (1/2)


Two main ways to do this:

- (not counting loading into RAM on every mapper)
- 1. Reduce-side join: Roughly the same as intersection
 - map over both source datasets and send to the same reducer by using emit ("join key", tuple)
 - reduce check that there are tuples from different datasets
 - e.g., for one-to-one join there must be 2 tuples, one from each dataset
 - if they are of dissimilar types, proceed to join them and emit
 - e.g., merge, filter on a predicate, compute aggregates

Join (2/2)

- 2. Map-side join: Need a way of ensuring both sources are partitioned the same way
 - typically requires that we directly access files from within map
 - e.g., the outputs of prior MapReduce runs
 - as map gets called with an argument, merge it with the contents of the (hopefully local) file
- Which should be more efficient?

Plan for today

- Introduction ✓
 - Census example ✓
- MapReduce architecture ✓
 - Programming model, data flow ✓
 - Details, fault tolerance, challenges, etc. ✓
- Single-pass algorithms in MapReduce
 - Filtering, aggregation, intersections and joins ✓
 - **Sorting** 
 - Strengths and weaknesses
- A brief overview of Hadoop

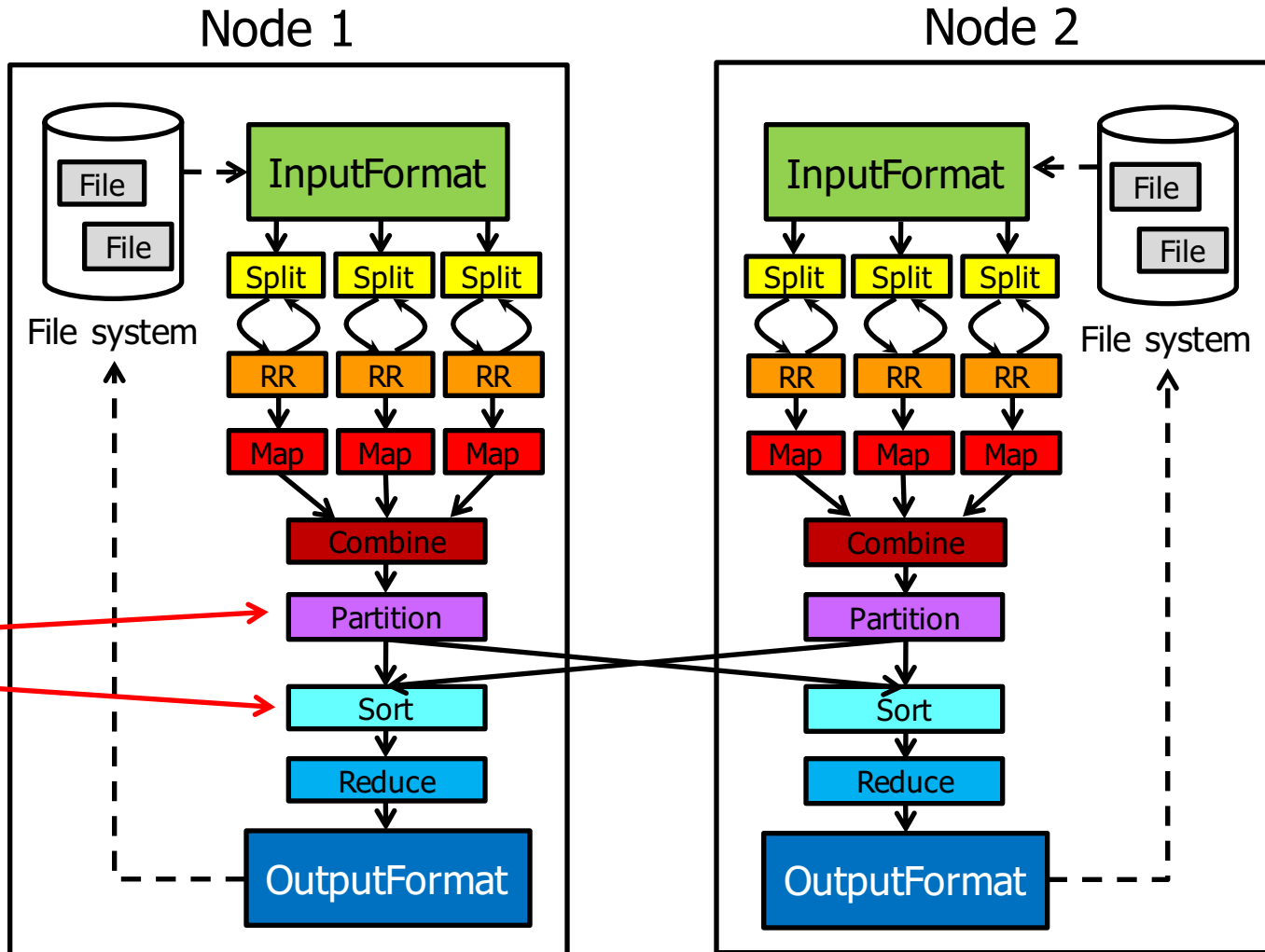
Sorting

- Runtime guarantees that reduce keys will be presented to reduce in sorted order
 - Convenient result of the Shuffle stage
 - Why?

The shuffle stage revisited

Shuffle really consists of two parts:

- Partition
- Sort




Sorting

- Runtime guarantees that reduce keys will be presented to reduce in sorted order
 - Convenient result of the Shuffle stage
- What if in addition to sorting by key, we also need to sort by value?
- Obvious solution is to buffer all values in the reducer and then sort them before additional processing

Shuffle as a sorting mechanism

- We can exploit the per-node sorting operation done by the Shuffle stage
- Perform a “value-to-key conversion”
 - map: Instead of emitting (reduce key, value), move the field in value that we want to sort on to the reduce key
 - e.g., to sort by time: emit(<id, timestamp>, value)
 - Need a custom partitioner to ensure that all pairs with the same original reduce key are shuffled to the same reducer
 - reduce: Receive all key-value pairs in the sorted order but values are split across multiple keys
 - Need to preserve state across invocations of reduce to maintain the right processing context
 - e.g., know when current item identified by id ends and next one starts

Plan for today

- Introduction ✓
 - Census example ✓
- MapReduce architecture ✓
 - Programming model, data flow ✓
 - Details, fault tolerance, challenges, etc. ✓
- Single-pass algorithms in MapReduce
 - Filtering, aggregation, intersections and joins ✓
 - Sorting ✓
 - Strengths and weaknesses 
- A brief overview of Hadoop

Strengths and weaknesses

- What problems can you solve well with MapReduce?
 - ... in a single pass?
 - ... in multiple passes?
- Are there problems you cannot solve efficiently with MapReduce?
- Are there problems it can't solve at all?
- How does it compare to other ways of doing large-scale data analysis?
 - Is MapReduce always the fastest/most efficient way?

Recap: MapReduce algorithms

- A variety of different tasks can be expressed as a single-pass MapReduce program
 - Filtering and aggregation + combinations of the two
 - Joins on shared elements
 - If we allow multiple MapReduce passes or even fixpoint iteration, we can do even more
 - We will see examples later, including PageRank
- But it does not work for all tasks
 - Sorting doesn't work at all, at least in the abstract model, but the implementations support it
 - Algorithms that depend on shared global state during processing are difficult to implement

Additional references


Data-Intensive Text Processing with MapReduce

Jimmy Lin and Chris Dyer

Morgan & Claypool Publishers, 2010

<http://lintool.github.io/MapReduceAlgorithms/>

Plan for today

- Introduction ✓
 - Census example ✓
- MapReduce architecture ✓
 - Programming model, data flow ✓
 - Details, fault tolerance, challenges, etc. ✓
- Single-pass algorithms in MapReduce ✓
 - Filtering, aggregation, intersections and joins ✓
 - Sorting ✓
 - Strengths and weaknesses ✓
- A brief overview of Hadoop 

Hadoop

- An open-source software framework for large-scale processing
 - Implementation of MapReduce
 - Hadoop Distributed File System (HDFS)
 - YARN – a resource-management platform for managing compute resources and scheduling applications



2002-2004: Lucene and Nutch

- Early 2000s: Doug Cutting develops two open-source search projects:

- Lucene: Search indexer
 - Used e.g., by Wikipedia
- Nutch: A spider/crawler (with Mike Carafella)



- Nutch

- Goal: Web-scale, crawler-based search
- Written by a few part-time developers
- Distributed, 'by necessity'
- Demonstrated 100M web pages on 4 nodes, but true 'web scale' still very distant



2004-2006: GFS and MapReduce

- 2003/04: GFS, MapReduce papers published
 - Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: "The Google File System", SOSP 2003
 - Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004
 - Directly addressed Nutch's scaling issues
- GFS & MapReduce added to Nutch
 - Two part-time developers over two years (2004-2006)
 - Crawler & indexer ported in two weeks
 - Ran on 20 nodes at IA and UW
 - Much easier to program and run, scales to several 100M web pages, but still far from web scale

2006-2008: Yahoo

- 2006: Yahoo hires Cutting
 - Provides engineers, clusters, users, ...
 - Big boost for the project; Yahoo spends tens of M\$
 - Not without a price: Yahoo has a slightly different focus (e.g., security) than the rest of the project; delays result
- Hadoop project split out of Nutch
 - Finally hit web scale in early 2008
- Cutting is now at Cloudera
 - Startup; started by three top engineers from Google, Facebook, Yahoo, and a former executive from Oracle
 - Has its own version of Hadoop; software remains free, but company sells support and consulting services
 - Was elected chairman of Apache Software Foundation

Who uses Hadoop?

- Hadoop is running search on some of the Internet's largest sites:
 - Amazon Web Services: Elastic MapReduce
 - AOL: Variety of uses, e.g., behavioral analysis & targeting
 - EBay: Search optimization (532-node cluster)
 - Facebook: Reporting/analytics, machine learning (1100 m.)
 - Fox Interactive Media: MySpace, Photobucket, Rotten T.
 - Last.fm: Track statistics and charts
 - IBM: Blue Cloud Computing Clusters
 - LinkedIn: People You May Know (2x50 machines)
 - Rackspace: Log processing
 - Twitter: Store + process tweets, log files, other data
 - Yahoo: >36,000 nodes; biggest cluster is 4,000 nodes

Writing jobs for Hadoop

- What do we need to write?
- A mapper
 - Accepts (key,value) pairs from the input
 - Produces intermediate (key,value) pairs, which are then shuffled
- A reducer
 - Accepts intermediate (key,value) pairs
 - Produces final (key,value) pairs for the output
- A driver
 - Specifies which inputs to use, where to put the outputs
 - Chooses the mapper and the reducer to use
- Hadoop takes care of the rest!
 - Default behaviors can be customized by the driver

Hadoop data types

Name	Description	JDK equivalent
IntWritable	32-bit integers	Integer
LongWritable	64-bit integers	Long
DoubleWritable	Floating-point numbers	Double
Text	Strings	String

- Hadoop uses its own serialization
 - Java serialization is known to be very inefficient
- Result: A set of special data types
 - All implement the 'Writable' interface
 - Most common types shown above; also has some more specialized types (SortedMapWritable, ObjectWritable, ...)

The Mapper

Input format
(file offset, line)

Intermediate format
can be freely chosen

```
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;

public class FooMapper extends Mapper<LongWritable, Text, Text, Text> {
    public void map(LongWritable key, Text value, Context context) {
        context.write(new Text("foo"), value);
    }
}
```

- Extends abstract 'Mapper' class
 - Input/output types are specified as type parameters
- Implements a 'map' function
 - Accepts (key,value) pair of the specified type
 - Writes output pairs by calling 'write' method on context

The Reducer

```
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;

public class FooReducer extends Reducer<Text, Text, IntWritable, Text> {
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws java.io.IOException, InterruptedException
    {
        for (Text value: values)
            context.write(new IntWritable(4711), value);
    }
}
```

Intermediate format
(same as mapper output)

Output format

Note: We may get multiple values for the same key!

- Extends abstract 'Reducer' class
 - Must specify types again (must be compatible with mapper!)
- Implements a 'reduce' function
 - Values are passed in as an 'Iterable'
 - **Caution:** These are NOT normal Java classes. Do not store them in collections - content can change between iterations!

The Driver

```
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FooDriver {
    public static void main(String[] args) throws Exception {
        Job job = new Job();
        job.setJarByClass(FooDriver.class);

        FileInputFormat.addInputPath(job, new Path("in"));
        FileOutputFormat.setOutputPath(job, new Path("out"));

        job.setMapperClass(FooMapper.class);
        job.setReducerClass(FooReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Mapper&Reducer are
in the same Jar as
FooDriver

Input and Output
paths

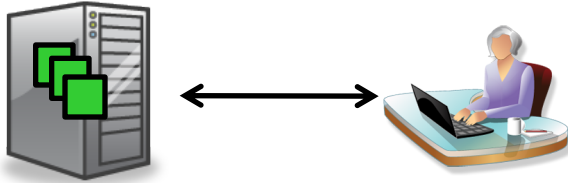
Format of the (key,value)
pairs output by the
reducer

- Specifies how the job is to be executed
 - Input and output directories; mapper & reducer classes

What is HDFS?

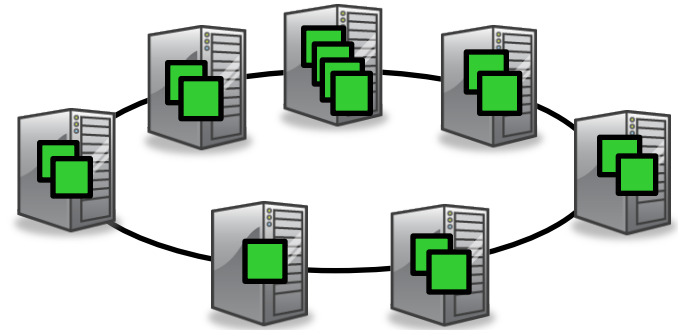
- HDFS is a distributed file system
 - Makes some unique tradeoffs that are good for MapReduce
- What HDFS does well:
 - Very large read-only or append-only files (individual files may contain Gigabytes/Terabytes of data)
 - Sequential access patterns
- What HDFS does not do well:
 - Storing lots of small files
 - Low-latency access
 - Multiple writers
 - Writing to arbitrary offsets in the file

HDFS versus NFS



Network File System (NFS)

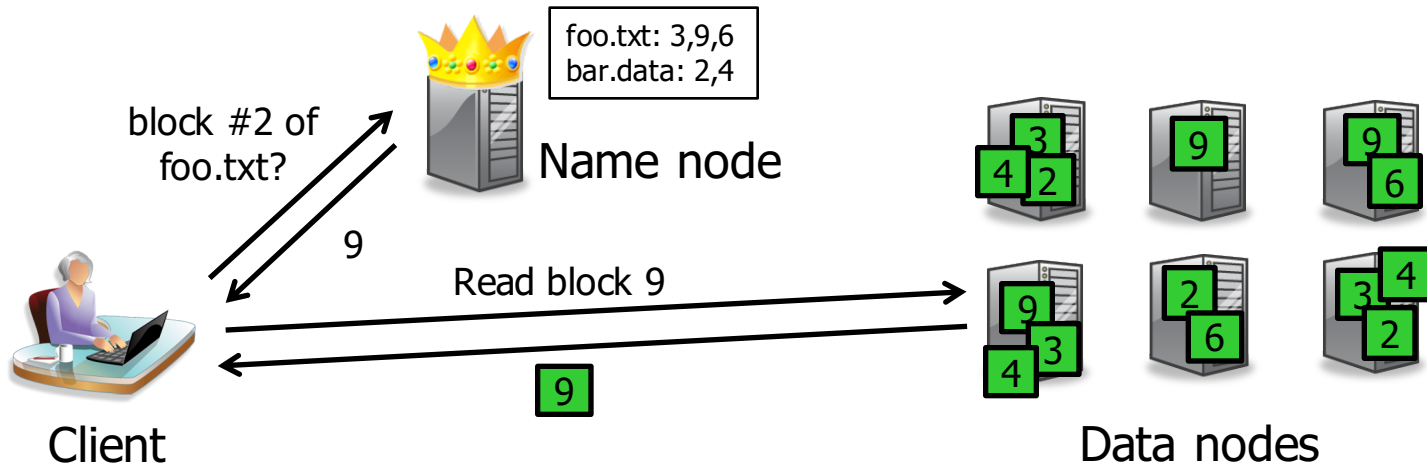
- Single machine makes part of its file system available to other machines
- Sequential or random access
- **PRO:** Simplicity, generality, transparency
- **CON:** Storage capacity and throughput limited by single server



Hadoop Distributed File System (HDFS)

- Single virtual file system spread over many machines
- Optimized for sequential read and local accesses
- **PRO:** High throughput, high capacity
- **"CON":** Specialized for particular types of applications

How data is stored in HDFS



- Files are stored as sets of (large) blocks
 - Default block size: 64 MB (ext4 default is 4kB!)
 - Blocks are replicated for durability and availability
 - What are the advantages of this design?
- Namespace is managed by a single name node
 - Actual data transfer is directly between client & data node
 - Pros and cons of this decision?

The Namenode



Name node

```
foo.txt: 3,9,6  
bar.data: 2,4  
blah.txt: 17,18,19,20  
xyz.img: 8,5,1,11
```

fsimage

```
Created abc.txt  
Appended block 21 to blah.txt  
Deleted foo.txt  
Appended block 22 to blah.txt  
Appended block 23 to xyz.img  
...
```

edits

- State stored in two files: fsimage and edits
 - fsimage: Snapshot of file system metadata
 - edits: Changes since last snapshot
- Normal operation:
 - When namenode starts, it reads fsimage and then applies all the changes from edits sequentially
 - Pros and cons of this design?

The Secondary Namenode

- What if the state of the namenode is lost?
 - Data in the file system can no longer be read!
- **Solution #1: Metadata backups**
 - Namenode can write its metadata to a local disk, and/or to a remote NFS mount
- **Solution #2: Secondary Namenode**
 - Purpose: Periodically merge the edit log with the fsimage to prevent the log from growing too large
 - Has a copy of the metadata, which can be used to reconstruct the state of the namenode
 - But: State lags behind somewhat, so data loss is likely if the namenode fails

Stay tuned



Next time you will learn about:
Algorithms in MapReduce