CS 6423

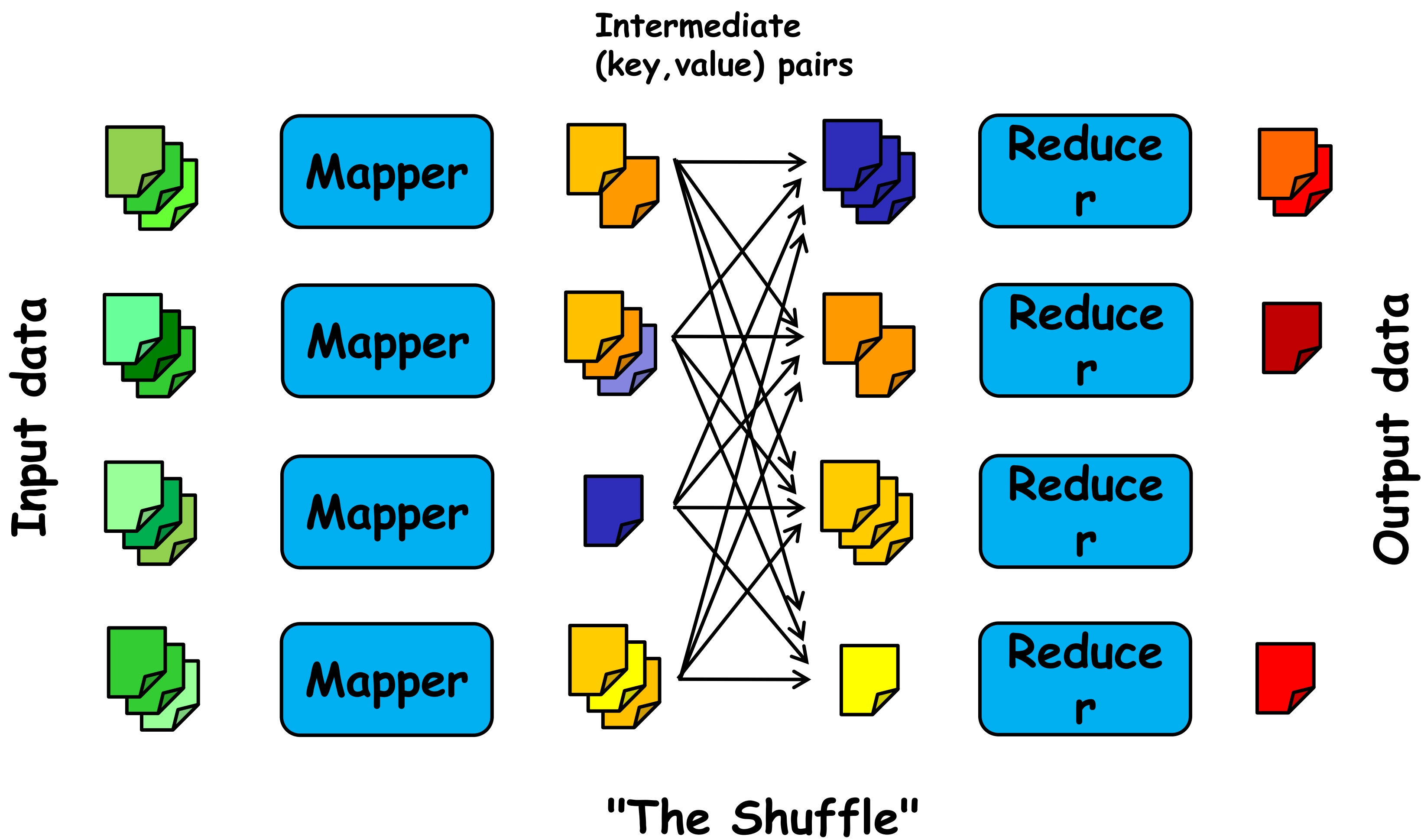
Scalable Computing for Big Data Analytics

Lecture 3: MapReduce: Aggregation Algorithhms

Prof. Gregory Provan
Department of Computer Science
University College Cork

Lecture adapted from:NETS 212: Scalable and Cloud Computing

Recap: MapReduce dataflow



Recap: MapReduce

These types depend on the input data

```
map(key:URL, value:Document)
{
  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 <u>all</u> the intermediate values with the same rkey

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

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

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

Plan for today

- Single-pass algorithms in MapReduce NEXT
  - Filtering algorithms
  - Aggregation algortihms
  - Intersections and joins
  - Partial Cartesian products
  - Sorting

The basic idea

- Single-pass algorithms
- Break algorithm 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

Word Count: Baseline

class Mapper
    method Map(docid a, doc d)
        for all term $t \in$ doc d **do**
            Emit(term t, count 1)

class Reducer
    method Reduce(term t, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ **do**
            $sum \leftarrow sum + c$
        Emit(term t, count s)

Word Count: Version 1

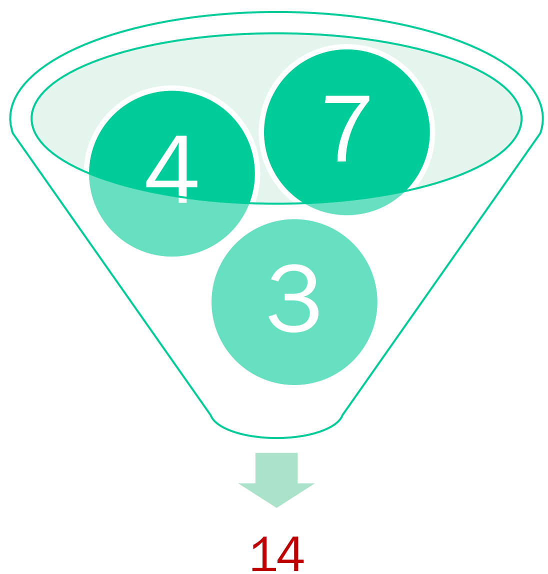
class Mapper
    method Map(docid a, doc d)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc d do
            $H\{t\} \leftarrow H\{t\} + 1$ ▷ Tally counts for entire document
        for all term $t \in H$ do
            Emit(term t, count $H\{t\}$)

MapReduce and Monoids

- MapReduce assumes *commutative Monoids* for the underlying algebraic set operations
- Monoid
  - Suppose that *S* is a set and • is some binary operation *S* × *S* ⟶ *S*, then *S* with • is a **monoid** if it satisfies the following two axioms:
    - **Associativity**: For all *a*, *b* and *c* in *S*, the equation $(a \bullet b) \bullet c = a \bullet (b \bullet c)$ holds.
    - **Identity element**: There exists an element *e* in *S* such that for every element *a* in *S*, the equations $e \bullet a = a \bullet e = a$ hold.
  - A binary operation • on a set *S* is called *commutative* if: $x \bullet y = y \bullet x$ for all $x, y \in S$.

Commutative Monoid and MapReduce

$$\underbrace{(1+1+1)}_{3} + \underbrace{(1+1+1+1+1+1+1)}_{7} + \underbrace{(1+1+1+1)}_{4}$$



Closure

Takes type X and returns type X

- 3 + 4 = 7 (int + int = int)
- 5 / 2 = 2.5 (int + int != float)

Identity

"concept of nothing"

- 5 + 0 = 5
- 5 * 1 = 5
- {3, 11, 9} + {} = {3, 11, 9}

- Initializing a counter to zero

Associativity

Add parenthesis anywhere

- 1 + 2 + 3 = (1 + 2) + 3
- 10 / 2 / 5 != 10 / (2 / 5)

- Huge jobs can become many small jobs

Commutativity

Reordering

- 1 + 2 + 3 = 2 + 3 + 1
- 10 / 2 != 2 /10

Monoid

- Closure (int + int = int)
- Identity (1 + 0 = 1)
- Associativity (1 + 2 + 3 = (1 + 2) + 3)

- Commutative Monoid

Design Pattern for Local Aggregation

- "In-mapper combining"
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Faster than actual combiners
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Remember: combiner are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of all integers associated with the same key

Filtering algorithms

- Goal: Find lines/files/tuples with a particular characteristic

- Examples:
  - grep Web logs for requests to *.ucc.ie/*
  - find in the Web logs the hostnames accessed by 192.168.2.1
  - locate all the files that contain the words 'Apple' and 'Jobs'

- 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 *.ucc.ie/*
  - Find the most popular domain
  - Average the number of requests per page per Web site
- Often: **`map`** may be simple or the identity

Computing the Mean

class Mapper
  method Map(string t, integer r)
    Emit(string t, integer r)

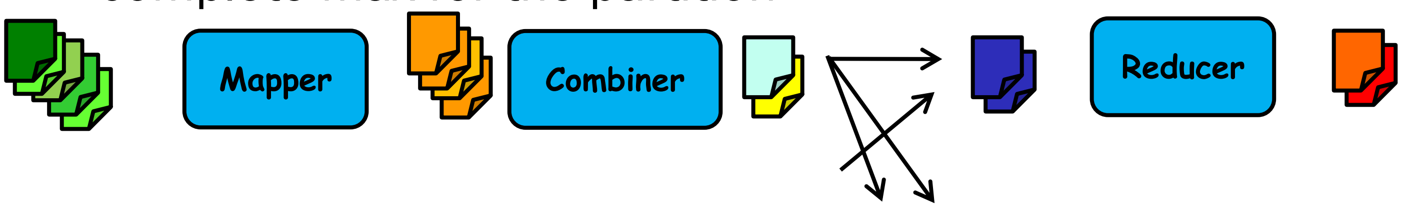
class Reducer
  method Reduce(string t, integers $[r_1, r_2, \ldots]$)
    $sum \leftarrow 0$
    $cnt \leftarrow 0$
    for all integer $r \in$ integers $[r_1, r_2, \ldots]$ **do**
      $sum \leftarrow sum + r$
      $cnt \leftarrow cnt + 1$
    $r_{avg} \leftarrow sum/cnt$
    Emit(string t, integer r_{avg})

A more complex example

- Goal: Billing for 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

Advanced Aggregation: Combiners

- Certain functions can be decomposed into partial steps:
  - Can take counts of two sub-partitions, sum them up to get a complete count for the partition
  - Can take maxes of two sub-partitions, max them to get a complete max for the partition



- Multiple map jobs on the same machine may write to the same reduce key
  - Example: map(1,"Apple juice") -> ("apple",1), ("juice",1)
  - map(2, "Apple sauce") -> ("apple",1),("sauce",1)
  - combiner: ("apple", [1,1]) -> ("apple", 2)

Intersections and joins

- Goal: Intersect multiple different inputs on some shared values
  - Values can be equal, or meet a certain predicate

- Examples:
  - Find all documents with the words “data” and “centric” given an inverted index
  - Find all professors and students in common courses and return the pairs <professor,student> for those cases

Partial Cartesian products

- Goal: Find some complex relationship, e.g., based on pairwise distance
- Examples:
  - Find all pairs of sites within 100m of each other
- Generally hard to parallelize
  - But may be possible if we can divide the input into bins or tiles, or link it to some sort of landmark
  - Overlap the tiles? (how does this scale?)
  - Generate landmarks using clustering?

Sorting

- Goal: Sort input

- Examples:
  - Return all the domains covered by Google's index and the number of pages in each, ordered by the number of pages

- The programming model does not support this per se, but the implementations do
  - Let’s take a look at what happens in the Shuffle stage

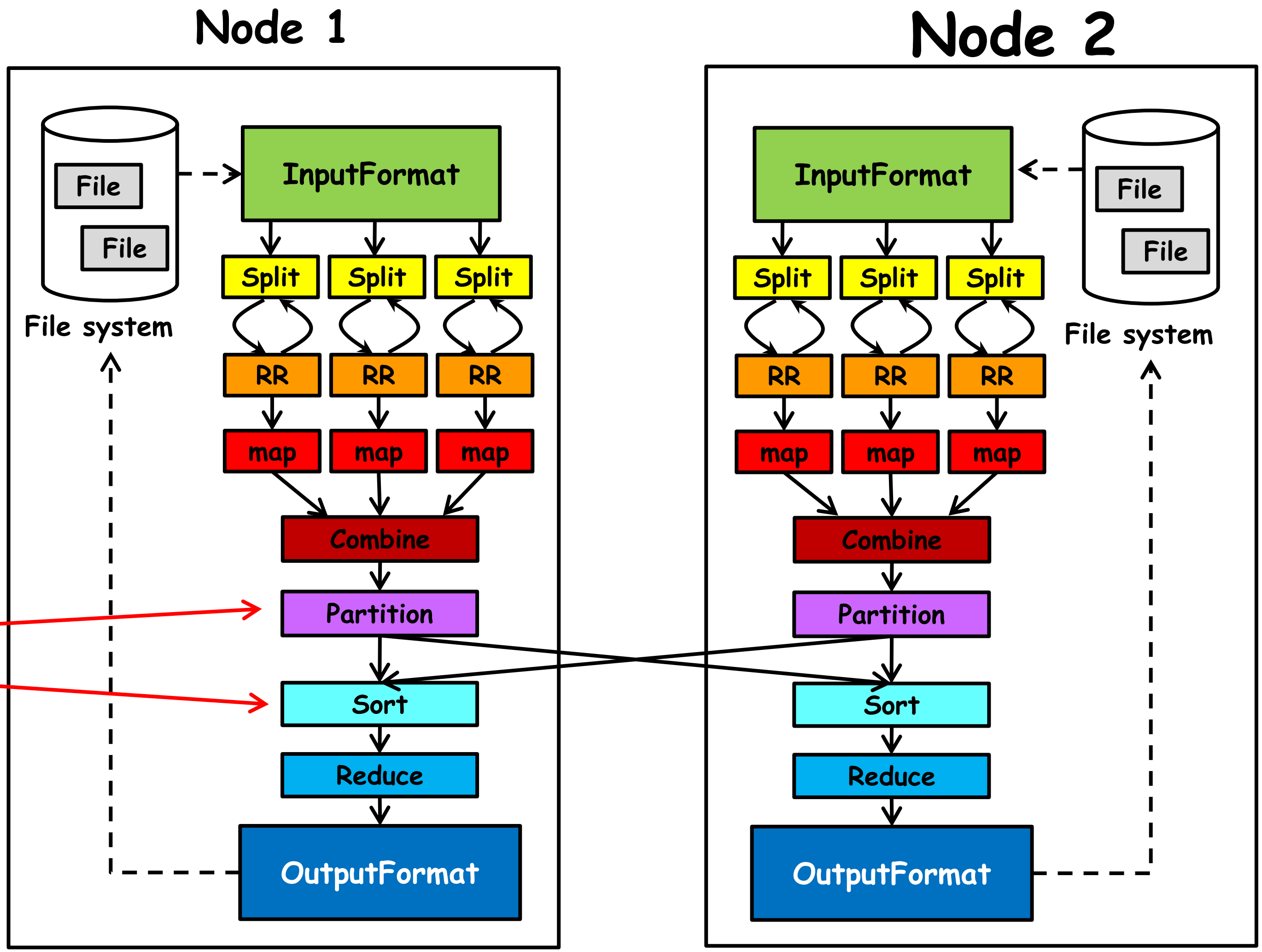
Plan for today

- Single-pass algorithms in MapReduce
  - Filtering algorithms
  - Aggregation algortihms
  - Intersections and joins
  - Partial Cartesian products
  - Sorting

NEXT

The shuffle stage revisited

Shuffle really consists of two parts:
- Partition
- Sort



Example: Hadoop

Shuffle as a sorting mechanism

- We can exploit the per-node sorting operation done by the Shuffle stage
  - If we have a single reducer, we will get sorted output
  - If we have multiple reducers, we can get partly sorted output (or better – consider an order-preserving hash)
    - Note it’s quite easy to write a last-pass file that merges all of the part-r-000x files

- Example
  - Return all the domains covered by Google's index and the number of pages in each, ordered by the number of pages

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 (see later)

- But it does not work for all tasks
  - Partial Cartesian product not an ideal fit, but can be made to work with binning and tiling
  - Sorting doesn't work at all, at least in the abstract model, but the implementations support it

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

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

Wrong!

More common mistakes to avoid

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

Wrong!

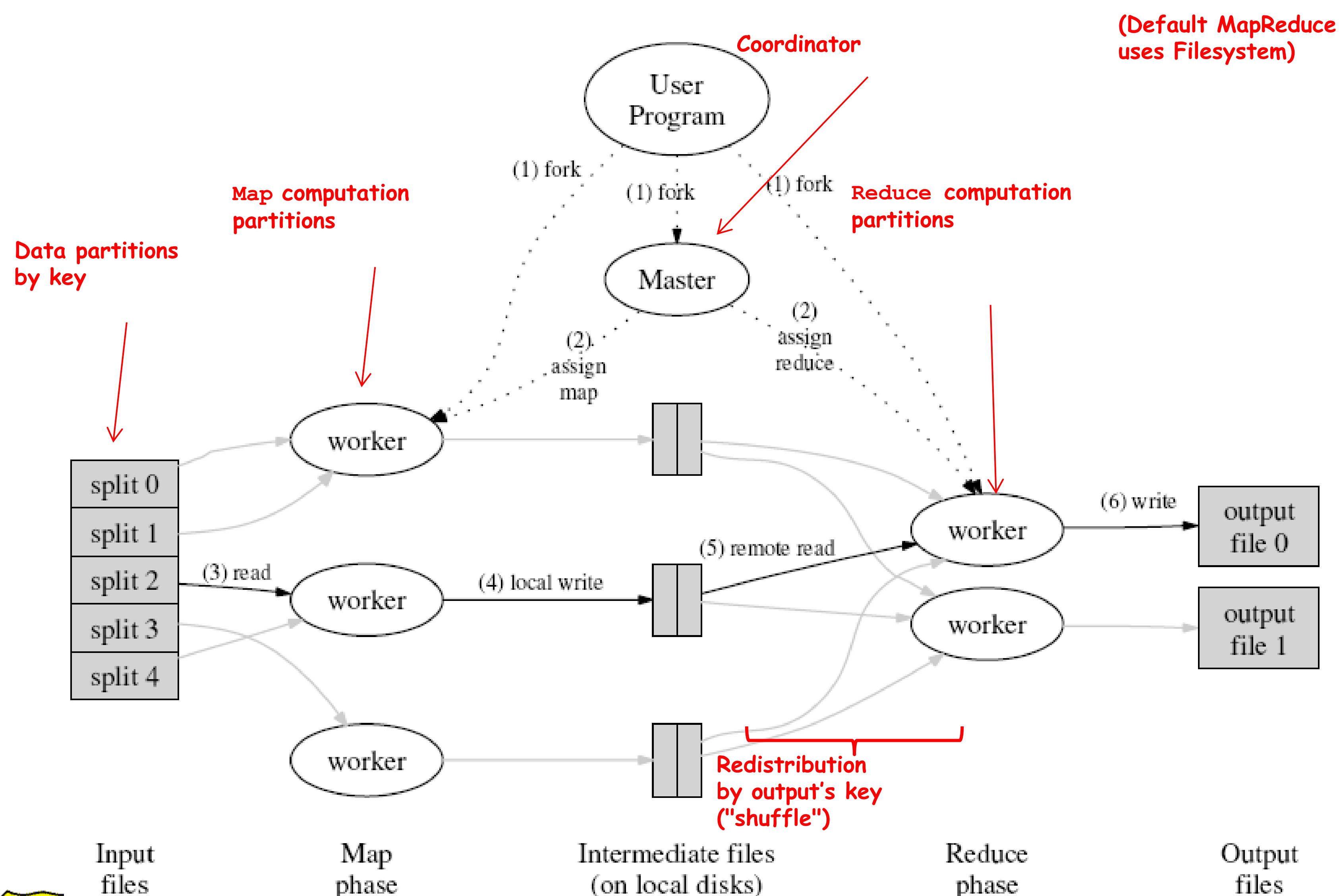
```
reduce(key, value[]) {
  /* do some computation on
  all the values */
}
```

- 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 'syzygy'.

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!

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
  - Esp. JobTracker

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 JobTracker 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
- Handling bad records
  - "Last gasp" packet with current sequence number

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).

Source: Interpreting the Data: Parallel Analysis with Sawzall (Rob Pike, Sean Dorward, Robert Griesemer, Sean