MapReduce

These slides follow the presentations given in Mining Massive Datasets by Rajaraman and Ullman and class notes by Rajaraman (Stanford) and Weld (U. Washington) , D. Weld (Google) and S. Ghemawat (Google)

Introduction

- MapReduce is a methodology for exploiting parallelism in computing clouds (racks of interconnected processors)
- It has become a common way to analyze very large amounts of data
- MapReduce was developed at Google
- In 2004 Google was using MapReduce to process 100TB/day of data
- By 2008 Google was using MapReduce to process 20PB/day of data

Motivation Beyond Search Engines

- **Modern Internet applications have created a need to manage immense amounts of data quickly.**
- **In many of these applications, the data is extremely regular, and there is ample opportunity to exploit parallelism.**
- **Some examples**

1. **Dish network collecting every click of the remote**
   - Dish network supplies TV reception via satellite; they collect data on their set top box and send it back to headquarters
2. **Tesla collecting every usage of the car**
   - Tesla's are connected to the cellular network; the car reports back all of its actions to Tesla

How is MapReduce Used by Search Engines?

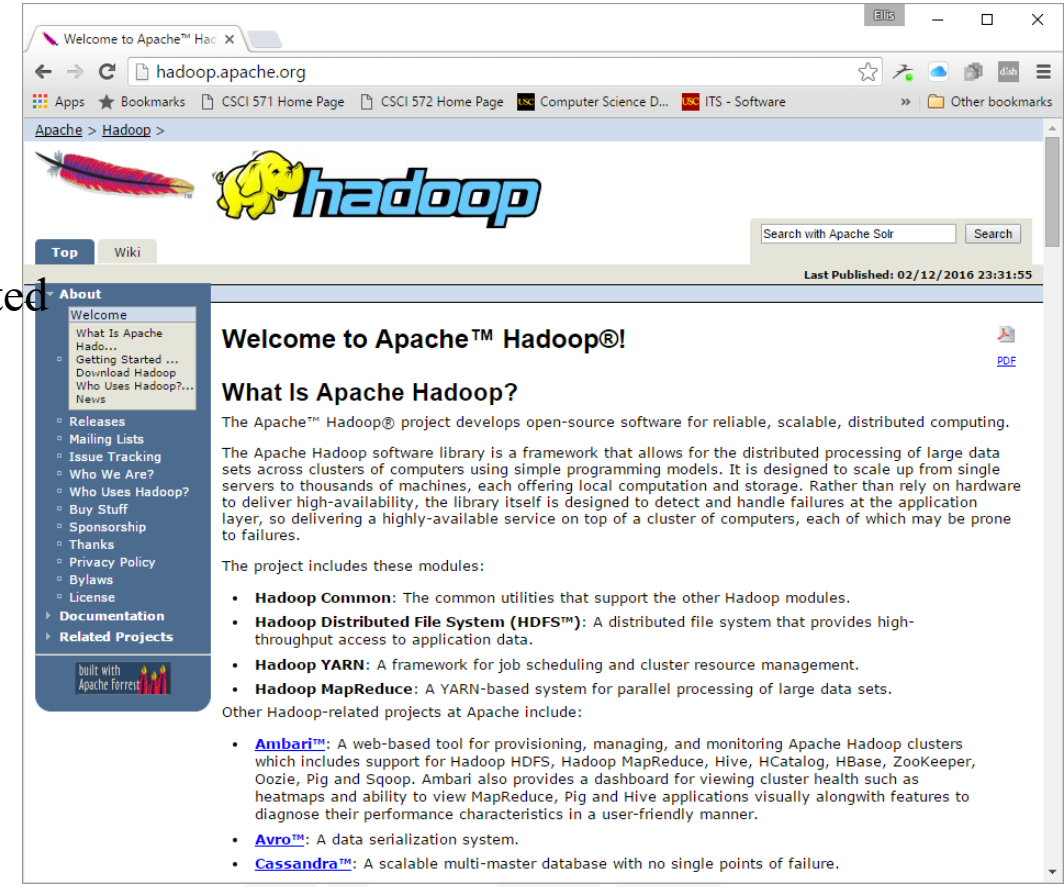
- **At Google:**
  - Building Google's Search Index
  - Article clustering for Google News
  - Statistical machine translation
- **At Yahoo!:**
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail
- **At Facebook:**
  - Data mining
  - Ad optimization
  - Spam detection

Hadoop - A MapReduce Implementation

Apache Hadoop is an open-source implementation of map/reduce written in Java for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware.

All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common and should be automatically handled by the framework.

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part called MapReduce.

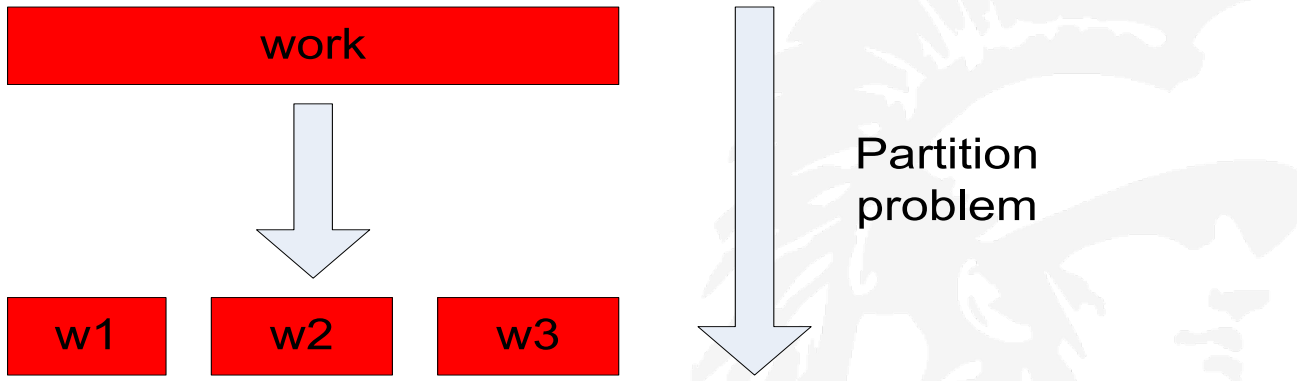


Large Data Requires Large, Distributed Files

- **File systems have to change - to handle much larger sizes, and replication of data**
- **Assumptions**
  1. **Files are distributed (DFS)**
     - Google File System (files divided into chunks, often read or appended to; rarely overwritten, distributed across clusters of machines)
     - Hadoop Distributed File System (written in Java)
     - CloudStore by Kosmiz (C++ implementation of GFS)
  2. **Files are rarely updated, often read and sometimes appended to**
  3. **Files are divided into chunks and chunks are replicated**

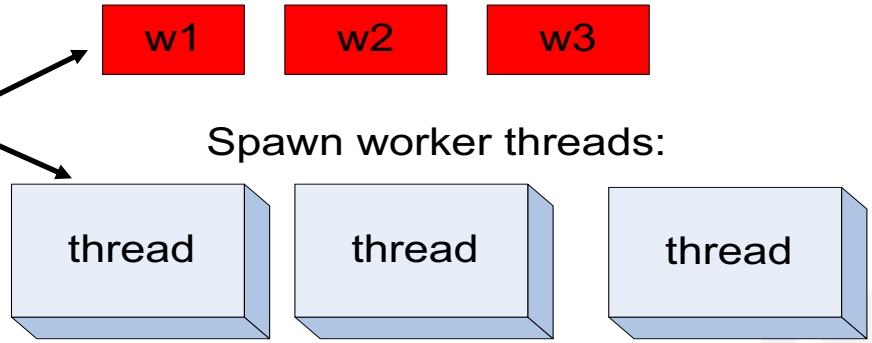
Why Parallelization is Hard

- **Parallelization is "easy" if processing can be cleanly split into n units:**



Why Parallelization is Hard

we would like to have as many threads as we have processors



But there are complicated issues to deal with!

- **How do we assign work units to worker threads?**
- **What if we have more work units than threads?**
- **How do we aggregate/combine the results at the end?**
- **How do we know all the workers have finished?**
- **What if the work cannot be divided into completely separate tasks?**

- ***MapReduce solves all of these problems***

Programming with Multiple Threads Poses Challenges

Thread 1:

```
void foo() {
  x++;
  y = x;
}
```

Thread 2:

```
void bar() {
  y++;
  x++;
}
```

If the initial state is x = 6, y = 0, what are the final values of x and y after the threads finish running? Possible solutions include: (8,8) and (8,7)

Multithreaded = Unpredictability

- Many things that look like "one step" operations actually take several steps under the hood:

Thread 1:

```
void foo() {
  eax = mem[x];
  inc eax;
  mem[x] = eax;
  ebx = mem[x];
  mem[y] = ebx;
}
```

Thread 2:

```
void bar() {
  eax = mem[y];
  inc eax;
  mem[y] = eax;
  eax = mem[x];
  inc eax;
  mem[x] = eax;
}
```

the same programs, but at the assembly code level

- **When we run a multithreaded program, we don't know what order threads run in, nor do we know when they will interrupt one another.**

The "corrected" example

Thread 1:

```
void foo() {
  sem.lock();
  x++;
  y = x;
  sem.unlock();
}
```

Thread 2:

```
void bar() {
  sem.lock();
  y++;
  x++;
  sem.unlock();
}
```

Global var "Semaphore sem = new Semaphore();" guards access to x & y

Unpredictability on Many Levels

This applies to more than just low level operations:

- **Pulling work units from a queue**
- **Reporting work back to master unit**
- **Telling another thread that it can begin the "next phase" of processing**

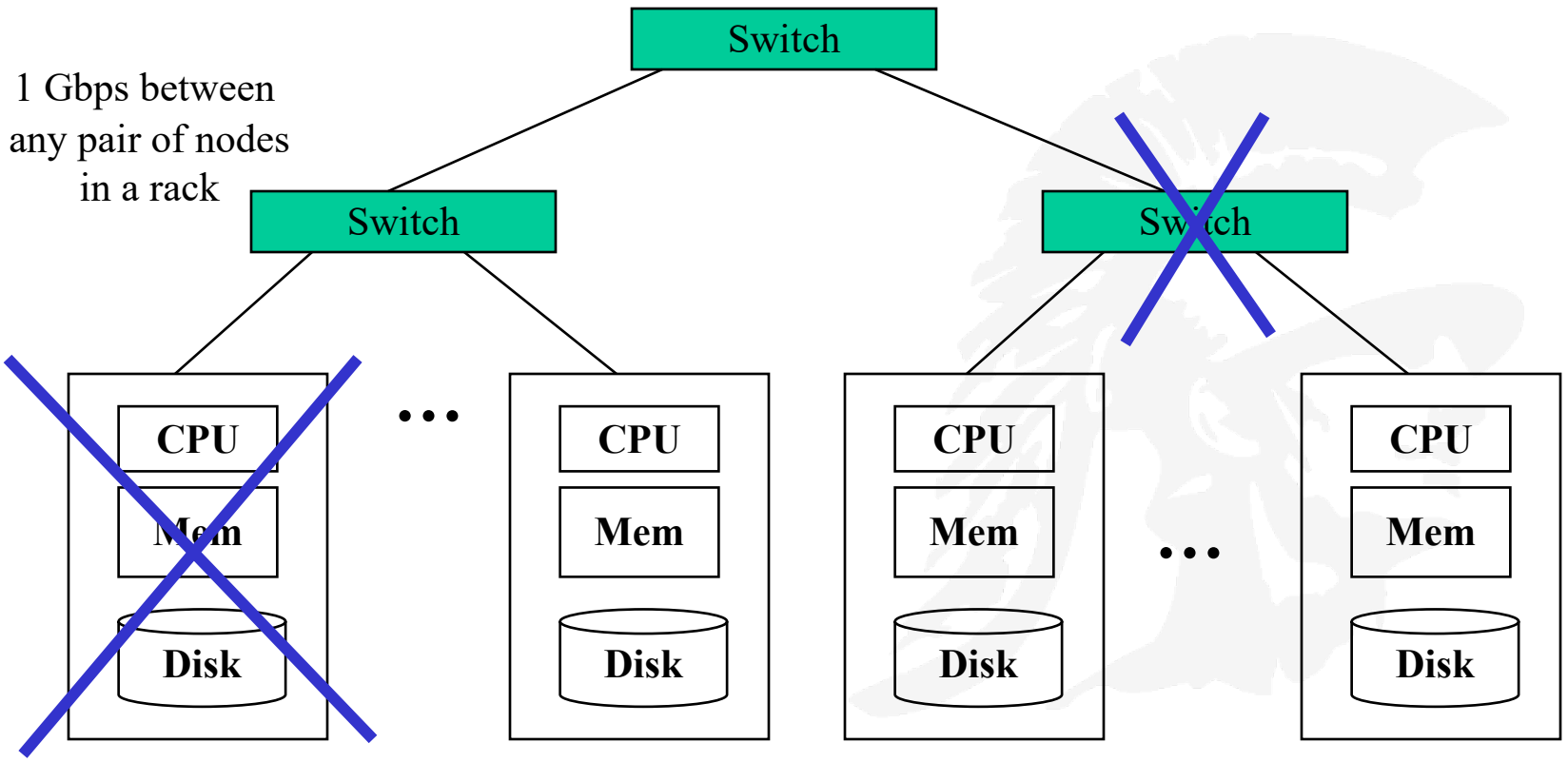
… All require synchronization!

How MapReduce Solves the Parallelization Problems

- **So MapReduce provides**
  - Automatic parallelization of code & distribution across multiple processors
  - Fault tolerance in the event of failure of one or more nodes
  - I/O scheduling
  - Monitoring & Status updates

Typical MapReduce Cluster Architecture

- **Each rack of cpu's contains between 16-64 nodes**
- **Nodes within a single rack are connected by gigabyte Ethernet**
- **Each rack is connected to another rack by a switch with speeds of 2-10 Gbps**
- **Individual cpu's can fail; switches between racks can fail**



Typical Data Center Cluster

Distributed File Systems

- *cluster computing* is defined as a collection of compute nodes stored on racks with racks connected by switches
- The inter-rack bandwidth is generally faster than the intra-rack bandwidth
- It is assumed that components will *often* fail, e.g.
  - Loss of single compute node
  - Loss of a single disk, causing a node to fail
  - Loss of an entire rack (the interconnecting switch fails)
- The solution to the constant failure problem is
  - Files are stored redundantly
  - Computations are divided into tasks such that if any one task fails it can be restarted without affecting other tasks
  - machines are constantly pinged

Map/Reduce - Beginnings

- **Map/Reduce**
  - Is a programming model borrowed from the programming language Lisp
  - (and other functional languages, e.g. ML)
- **Many problems can be phrased this way**
- **Easy to distribute computation across nodes**
- **Nice retry/failure semantics**

Map and Reduce in LISP (Scheme)

- (map *f list* [*list*$_2$ *list*$_3$ …]) General formulation
- Specific example
  - (map square '(1 2 3 4))
  - (1 4 9 16)

- (reduce *f id list*)
- Specific example
  - (reduce + 0 '(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 (+ 1 0))))
  - 30

The Map/Reduce Paradigm

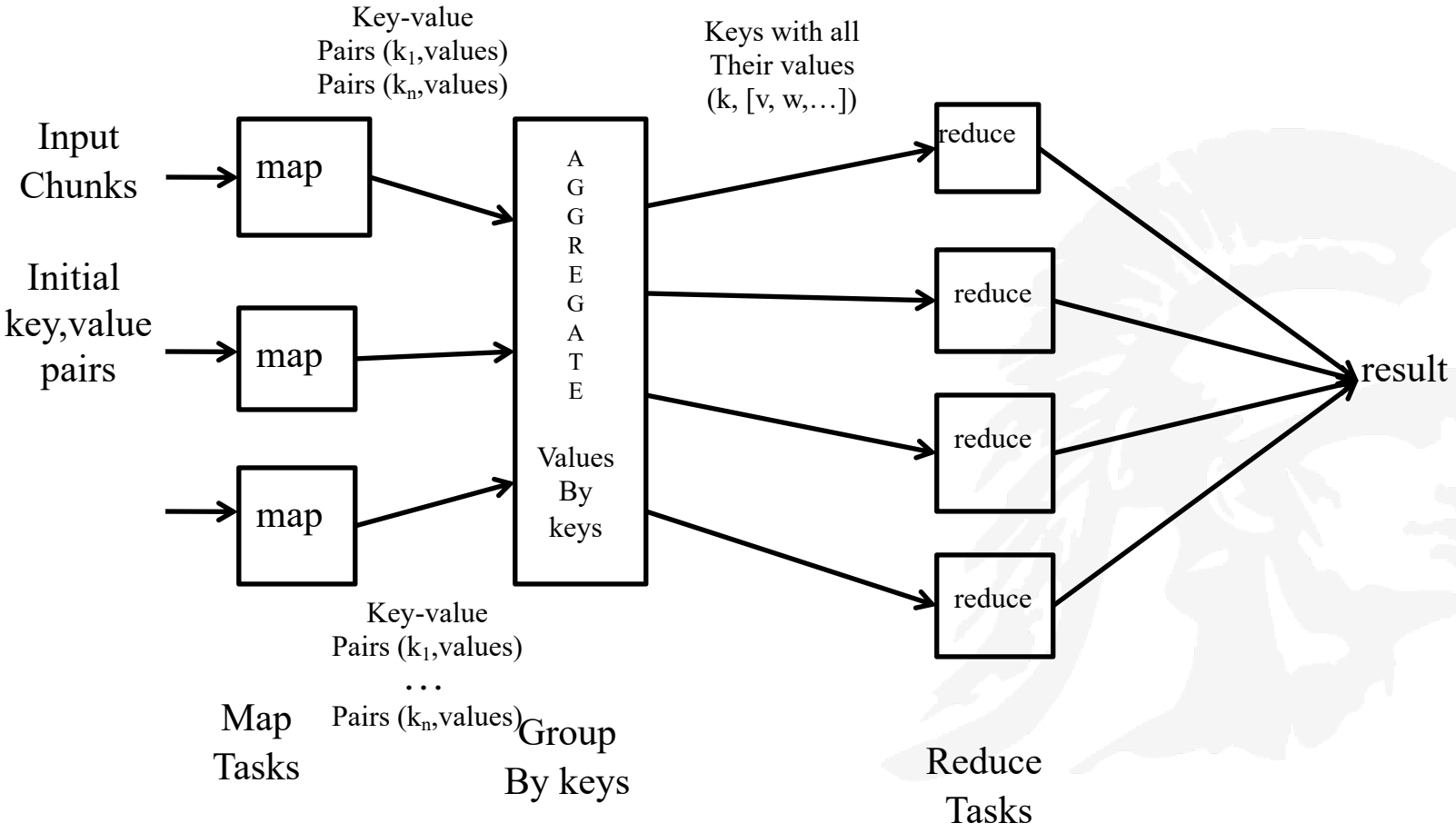
1. A large number of records are broken into segments
2. Map: extract something of interest from each segment
3. Group and sort intermediate results from each segment
4. Reduce: aggregate intermediate results
5. Generate final output

Key idea: to re-phrase problems in such a way that the input can be divided into parts and operated on in parallel and the results combined to produce a solution to the original problem

Map-Reduce Overview

- Using map-reduce one must write 2 functions called *Map* and *Reduce*
- The system manages the parallel execution and coordination of tasks; it is all done automatically
- A map-reduce computation proceeds as follows:
  1. Some number of map tasks each are given one or more chunks to process
  2. These map tasks turn the chunk into a sequence of key-value pairs; the way the pairs are produced depends upon the code for the Map function
  3. Key-value pairs from each Map task are collected by a master controller and sorted by key; keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task
  4. Reduce tasks work on one key at a time, and combine all the values associated with that key in some way; the manner of combination depends upon the Reduce code

Schematic of a Map-Reduce Computation



A MapReduce Example – Counting Word Occurrences

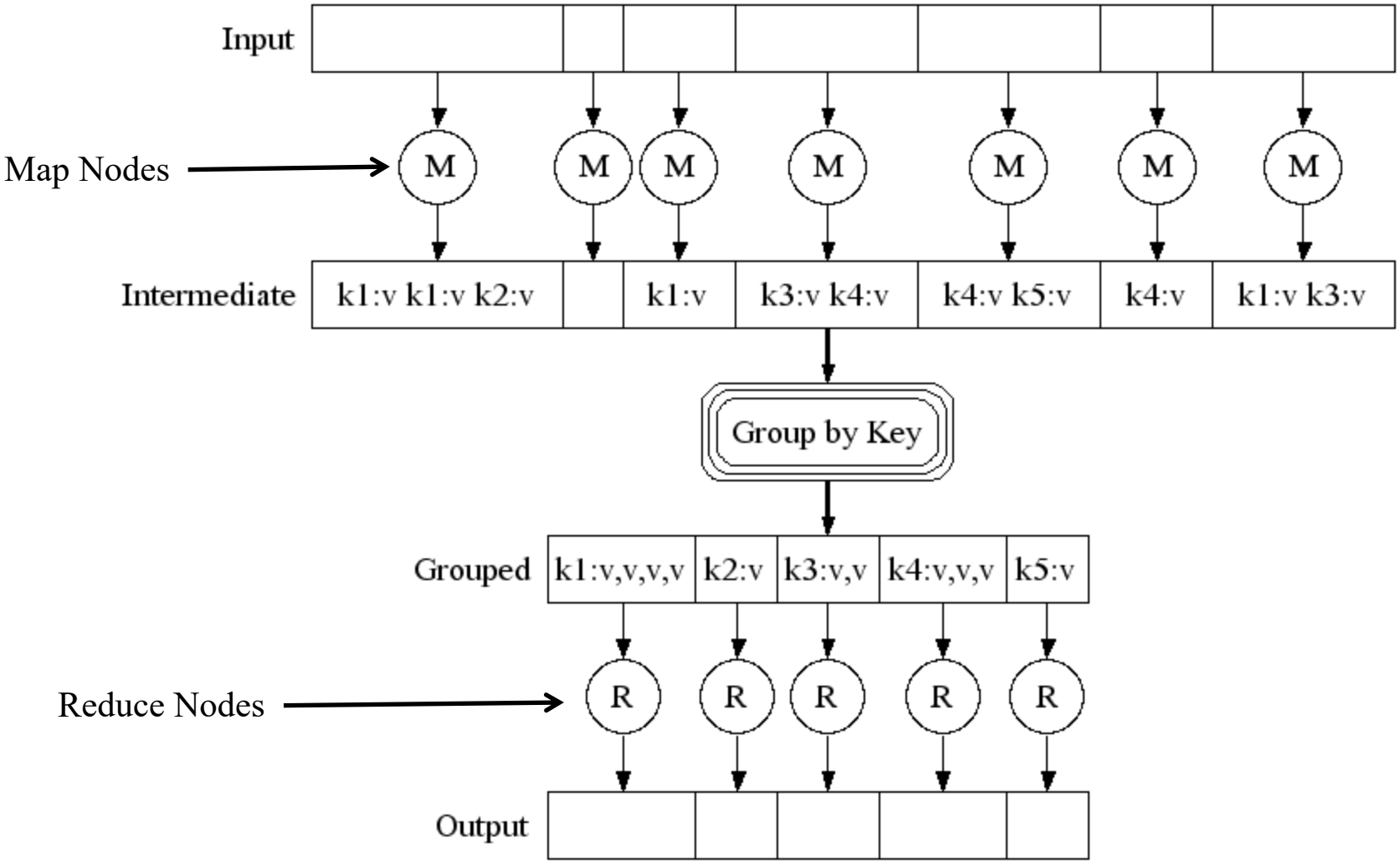
- Counting the number of occurrences for each word in a collection of documents
- The input file is a repository of documents
- Each document is an element
- The Map function
  - Uses keys of type String (the words obtained by parsing), $w_1, w_2, \ldots$
  - Uses values that are integers
  - Reads a document and breaks it into its sequence of words, $w_1, \ldots, w_n$
  - Outputs key-value pairs where the value is always 1, namely $(w_1,1), (w_2,1), \ldots, (w_n,1)$
- The Map task handles many documents so its output will be more than the sequence for one document
- If a word w appears m times among all documents, then there will be m key-value pairs $(w,1)$ in the output

Count Word Occurrences Code

```
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");

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));
```

Word Count Execution
An Alternative View



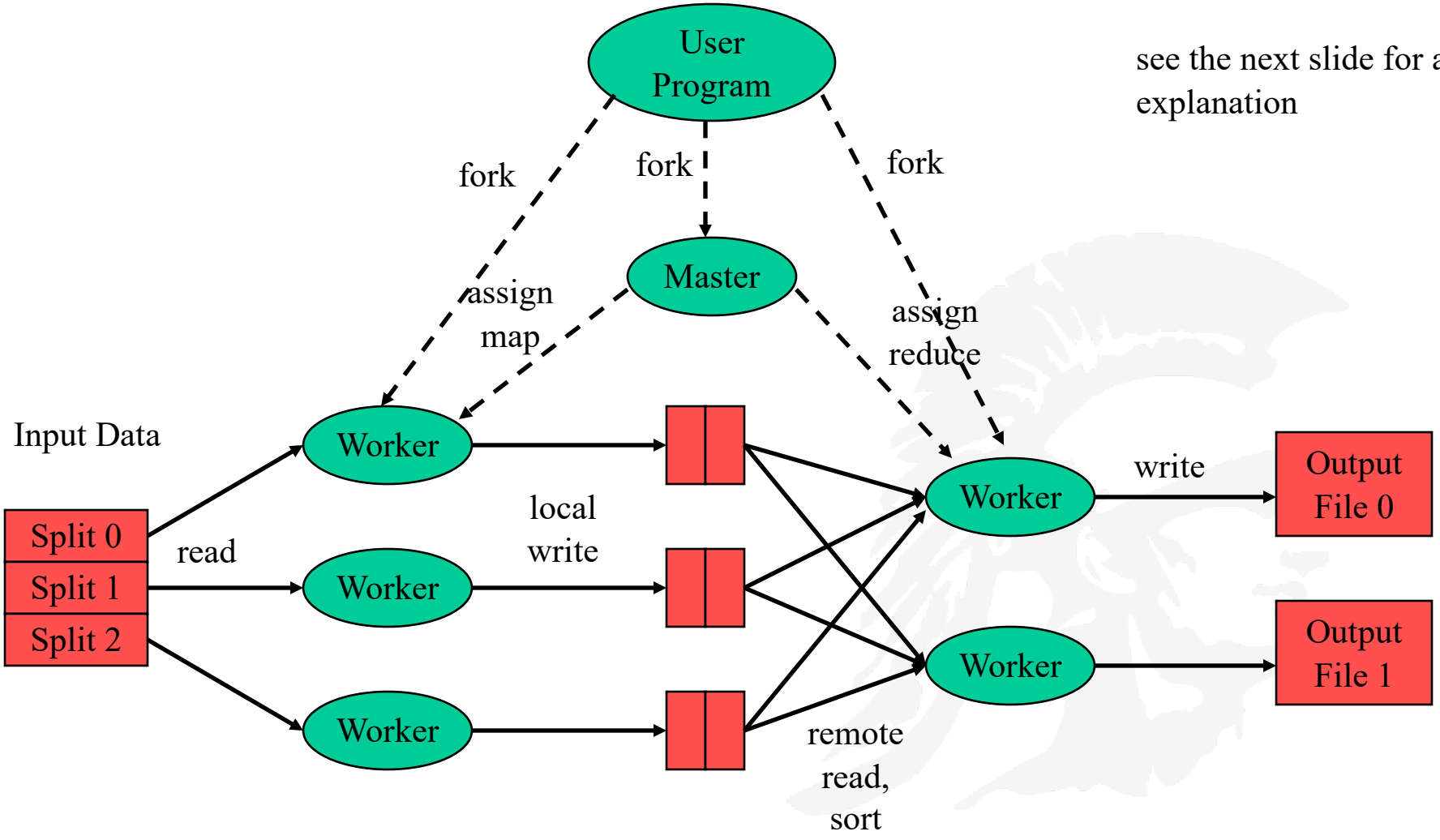
Explanation of Grouping and Aggregation

- There is a *master controller* process that knows how many Reduce tasks there will be, say *r*
- The user defines *r*
- The master controller picks a hash function that applies to keys and produces a bucket number from *0* to *r-1*
- Each key output by a Map task is hashed and its key-value pair is put in one of *r* local files
  - Each file will be processed by a Reduce task
- After all Map tasks have completed successfully, the master controller merges the file from each Map task that are destined for a particular Reduce task and feeds the merged file to that process
- For each key *k*, the input to the Reduce task that handles key *k* is a pair $(k, [v_1, \ldots, v_n])$ where $(k,v_1),(k,v_2),\ldots,(k,v_n)$ are all the key-value pairs with key *k* coming from all the Map tasks

Explanation of the Reduce Task

- The *Reduce function* is written to take pairs consisting of a key and a list of associated values, and combines them in some way
- The *Reduce function* output is a sequence of key-value pairs consisting of each input key *k* paired with the combined value
- Outputs from all Reduce tasks are merged into a single file
- *Reduce function* adds up all the values and outputs a sequence of *(w,m)* pairs where *w* is a word that appears at least once in the documents and *m* is the total number of occurrences
- The *Reduce function* is generally associative and commutative implying values can be combined in any order yielding the same result

Distributed Execution Overview



see the next slide for an explanation

Parallel Execution of Map-Reduce - Looking Under the Hood

- The user program forks a *Master* controller process and some number of *Worker* processes at different compute nodes;
- A *Worker* handles either Map tasks or Reduce tasks, but not both
- The *Master* must
  - Create some number of Map and Reduce tasks
  - These tasks are assigned to *Worker* processes by the *Master*
  - Typically there is one Map task for every chunk of the input
  - Keeps track of the status of each Map and Reduce task (states are: idle, executing on a Worker, completed)
- Each Map task is assigned one or more chunks of the input file(s) and executes on it the code
- The Map task creates a file for each Reduce task on the local disk of the Worker that executes the Map task
- The *Master* is told of the location and sizes of each of these files and the Reduce task for which each is destined

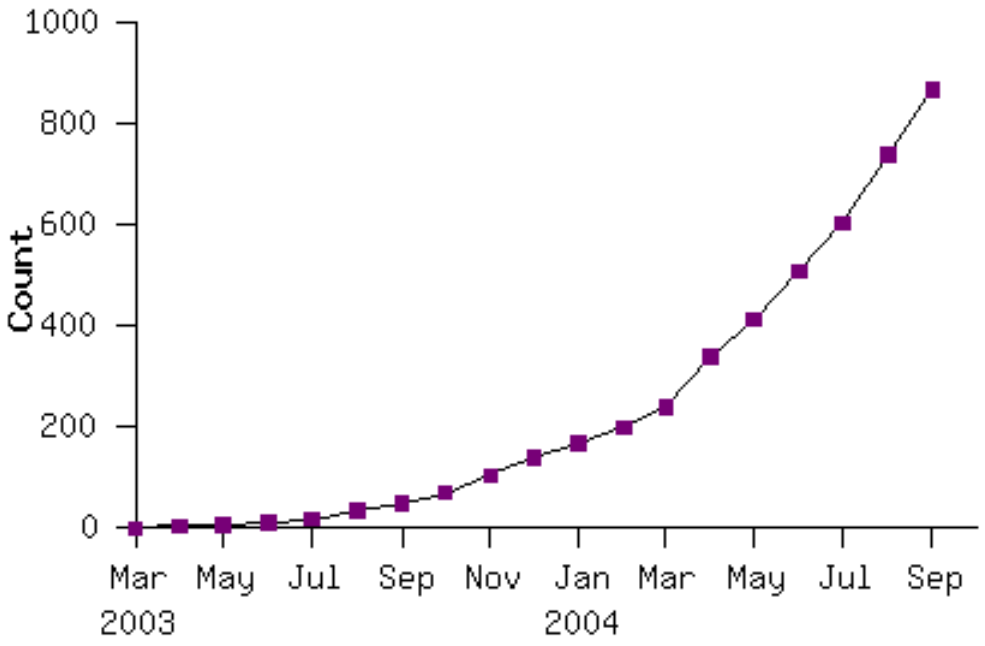
Fault Tolerance in MapReduce

1. **If a task crashes:**
   - Retry on another node
     - OK for a map because it had no dependencies
     - OK for reduce because map outputs are on disk
   - If the same task repeatedly fails, fail the job or ignore that input block
2. **If a node crashes:**
   - Relaunch its current tasks on other nodes
   - Relaunch any maps the node previously ran
     - Necessary because their output files were lost along with the crashed node
3. **If a task is going slowly (straggler):**
   - Launch second copy of task on another node
   - Take the output of whichever copy finishes first, and kill the other one

Coping with Node Failure

- Worst case: **the compute node where the Master is executing fails**
  - **Result**: the entire map-reduce job must be restarted
- Other failures are less severe and are handled by the Master
- **The compute node of a Map worker fails**
  - This is detected by the Master and all Map tasks that were assigned are re-done
  - The Master sets the status of each Map task to idle and re-schedules them when a worker becomes available
  - The Master informs each Reduce task of the location of its new input
- **The compute node of a Reduce worker fails**
  - The Master sets the status of its currently executing Reduce tasks to idle and they will be re-scheduled on another reduce worker later

The Use of the Map/Reduce Application in the Google Source Tree



Example uses:

distributed grep

distributed sort

web link-graph reversal

term-vector / host

web access log stats

inverted index construction

document clustering

machine learning

statistical machine translation

...

...

...

References

- **Google Videos on map/reduce**
  https://www.youtube.com/watch?v=yjPBkvYh-ss (Lecture 1, 46 min)
  https://www.youtube.com/watch?v=-vD6PUdf3Js (Lecture 2, 52 min)
- **Wikipedia,** http://en.wikipedia.org/wiki/MapReduce
- *Data-Intensive Text Processing with MapReduce*, Jimmy Lin and Chris Dyer, Morgan & Claypool Synthesis Lectures on Human Language Technologies, 2010
  http://www.umiacs.umd.edu/~jimmylin/MapReduce-book-final.pdf
- **Hadoop** is an open source implementation of MapReduce
  http://hadoop.apache.org/
- MapReduce: Simplified Data Processing on Large Clusters, by Jeffrey Dean and Sanjay Ghemawat, http://research.google.com/archive/mapreduce.html