MapReduce and Hadoop

Alan Sussman, Department of Computer Science

With thanks to MMDS authors
Leskovec, Rajaraman, Ullman,
www.mmds.org
Announcements

• Assignment 4 posted Tuesday, due May 2 at 11:59 pm

• Quiz 2 done, 3rd quiz likely last week of class
MapReduce

- **Challenges:**
  - How to distribute computation?
  - Distributed/parallel programming is hard

- **Map-reduce** addresses this problem for certain kinds of computations
  - Started as Google’s computational/data manipulation model
  - Overall, an elegant way to work (in parallel) with big data
Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- If 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to do something useful with the data!
- **A standard architecture for such problems emerged several years ago:**
  - Cluster of commodity Linux nodes
  - Commodity network (e.g., Ethernet) to connect them
Cluster Architecture several years ago

1 Gbps between any pair of nodes in a rack

2-10 Gbps backbone between racks

Each rack contains 16-64 nodes

In 2011 it was guesstimated that Google had 1M machines, http://bit.ly/Shh0RO
Large-scale Computing

- **Large-scale computing for data analytics** problems on commodity hardware

- **Challenges:**
  - **How do you distribute computation?**
  - **How can we make it easy to write distributed programs?**

- **Machines fail:**
  - One server may stay up 3 years (1,000 days)
  - If you have 1,000 servers, expect to lose 1/day
  - People estimated Google had ~1M machines in 2011
    - 1,000 machines fail every day!
Idea and Solution

- **Issue:** Copying data over a network takes time

- **Idea:**
  - Bring computation close to the data
  - Store files multiple times in multiple locations for reliability

- **Map-reduce** addresses these problems
  - Google’s computational/data manipulation model
  - Elegant way to work with big data

- **Storage Infrastructure – File system**
  - Google: GFS  Hadoop: HDFS

- **Programming model**
  - Map-Reduce
Storage Infrastructure

• **Problem:**
  - If nodes fail, how to store data persistently?

• **Answer:**
  - **Distributed File System:**
    - Provides global file namespace
    - Google GFS; Hadoop HDFS;
  - **Typical usage pattern**
    - Huge files (100s of GB to TB)
    - Data is rarely updated in place
    - Reads and appends are common
Distributed File System

• **Chunk servers**
  
  • File is split into contiguous chunks
  
  • Typically each chunk is 16-64MB
  
  • Each chunk replicated (usually 2x or 3x)
  
  • Try to keep replicas in different racks in the cluster

• **Primary node**
  
  • a.k.a. Name Node in Hadoop’s HDFS
  
  • Stores metadata about where files are stored
  
  • Might be replicated

• **Client library for file access**
  
  • Talks to primary to find chunk servers
  
  • Connects directly to chunk servers to access data
Distributed File System

- Reliable distributed file system
- Data kept in “chunks” spread across machines
- Each chunk replicated on different machines
- Seamless recovery from disk or machine failure

Bring computation directly to the data!

Chunk servers also serve as compute servers
Programming Model: MapReduce

Warm-up task:

• We have a huge text document
• Count the number of times each distinct word appears in the file

Sample application:

• Analyze web server logs to find popular URLs
Task: Word Count

The Problem:

- Count occurrences of words in a document:
  - `words(doc.txt) | sort | uniq -c`
  - where `words` takes a file and outputs the words in it, one per line

- This pipeline captures the essence of **MapReduce**
  - Great thing is that it is naturally parallelizable
MapReduce: Overview

• Inspired by LISP

• Map(function, set of values)
  • Applies function to each value in the set
    (map 'length '((()) (a) (a b) (a b c))) ⇒ (0 1 2 3)

• Reduce(function, set of values)
  • Combines all the values using a binary function (e.g.,+)
    (reduce #'+ '(1 2 3 4 5)) ⇒ 15
MapReduce: Overview

- Sequentially read a lot of data

- **Map:**
  - Extract something you care about

- **Group by key:** Sort and Shuffle

- **Reduce:**
  - Aggregate, summarize, filter or transform

- Write the result

Outline stays the same, Map and Reduce change to fit the problem
MapReduce: The Map Step

Input key-value pairs

Intermediate key-value pairs

map

...
MapReduce: The **Reduce** Step

**Intermediate key-value pairs**

- \( k \) \( v \)
- \( k \) \( v \)
- \( k \) \( v \)

- Group by key

**Key-value groups**

- \( k \) \( v \) \( v \) \( v \)
- \( k \) \( v \) \( v \)

**Output key-value pairs**

- \( k \) \( v \)
- \( k \) \( v \)
- \( k \) \( v \)
More Specifically

- **Input:** a set of key-value pairs

- Programmer specifies two methods:
  - **Map**(k, v) → <k’, v’>*
    - Takes a key-value pair and outputs a set of key-value pairs
    - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every (k,v) pair
  - **Reduce**(k’, <v’>* ) → <k’, v”>*
    - All values v’ with same key k’ are reduced together and processed in v’ order
    - There is one Reduce function call per unique key k’
MapReduce: Word Counting

**Big document**

(The, 1)

(crew, 1)

(of, 1)

(the, 1)

(space, 1)

(shuttle, 1)

(Endeavor, 1)

(recently, 1)

....

**MAP:**
Read input and produces a set of key-value pairs

**Group by key:**
Collect all pairs with same key

The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/machine partnership.

"The work we're doing now -- the robotics we're doing -- is what we're going to need

**Reduce:**
Collect all values belonging to the key and output

(crew, 2)

(space, 1)

(the, 3)

(shuttle, 1)

(recently, 1)

...
Word Count Using MapReduce

```java
map(key, value):
// key: document name; value: text of the document
for each word w in value:
    emit(w, 1)

reduce(key, values):
// key: a word; value: an iterator over counts
result = 0
for each count v in values:
    result += v
emit(key, result)
```
Map-Reduce: Environment

Map-Reduce environment (runtime system) takes care of:

• Partitioning the input data

• Scheduling the program’s execution across a set of machines

• Performing the group by key step

• Handling machine failures

• Managing required inter-machine communication
Map-Reduce: A diagram

**MAP:**
Read input and produces a set of key-value pairs

**Group by key:**
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

**Reduce:**
Collect all values belonging to the key and output

**Input**
Big document

**Intermediate**

- M
- k1:v k1:v k2:v
- k1:v
- k3:v k4:v
- k4:v k5:v
- k4:v
- k1:v k3:v

**Group by Key**

- Grouped
  - k1:v,v,v,v
  - k2:v
  - k3:v,v
  - k4:v,v,v
  - k5:v

**Output**

- R
- R
- R
- R
- R
Map-Reduce: In Parallel

All phases are distributed with many tasks doing the work.
Map-Reduce

- **Programmer specifies:**
  - Map and Reduce and input files

- **Workflow:**
  - Read inputs as a set of key-value-pairs
  - **Map** transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to output nodes
  - All k'v'-pairs with a given k' are sent to the same **reduce**
  - **Reduce** processes all k'v'-pairs grouped by key into new k"v"-pairs
  - Write the resulting pairs to files

- All phases are distributed with many tasks doing the work
Hadoop
Announcements

• Assignment 4 due May 2 at 11:59 pm
  • Questions?

• Quiz 3 last week of class
Data Flow – Hadoop architecture

• **Input and final output are stored in a distributed file system (FS):**
  • Scheduler tries to schedule map tasks “close” to physical storage location of input data

• **Intermediate results are stored on local FS of Map and Reduce workers**

• **Output is often input to another MapReduce task**
Input Files

- **Input files** are where the data for a MapReduce task is initially stored.

- The input files typically reside in a distributed file system (e.g., HDFS).

- The format of input files is arbitrary:
  - Line-based log files
  - Binary files
  - Multi-line input records
  - Or something else entirely, e.g., a database

Gregory Kesden, Carnegie Mellon U.
InputFormat

• How the input files are split up and read is defined by the InputFormat

• InputFormat is a class that does the following:
  • Selects the files that should be used for input
  • Defines the InputSplits that break a file
  • Provides a factory for RecordReader objects that read the file

Files loaded from local HDFS store

Gregory Kesden, Carnegie Mellon U.
**InputFormat Types**

- Several InputFormats are provided with Hadoop:

<table>
<thead>
<tr>
<th>InputFormat</th>
<th>Description</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextInputFormat</td>
<td>Default format; reads lines of text files</td>
<td>The byte offset of the line</td>
<td>The line contents</td>
</tr>
<tr>
<td>KeyValueInputFormat</td>
<td>Parses lines into (K, V) pairs</td>
<td>Everything up to the first tab character</td>
<td>The remainder of the line</td>
</tr>
<tr>
<td>SequenceFileInputFormat</td>
<td>A Hadoop-specific high-performance binary format</td>
<td>user-defined</td>
<td>user-defined</td>
</tr>
</tbody>
</table>

Gregory Kesden, Carnegie Mellon U.
Input Splits

• An *input split* describes a unit of work that comprises a single map task in a MapReduce program

• By default, the InputFormat breaks a file up into 64MB splits

• By dividing the file into splits, allow several map tasks to operate on a single file in parallel

• If the file is very large, this can improve performance significantly through parallelism

• Each map task corresponds to a *single* input split

Files loaded from local HDFS store

Gregory Kesden, Carnegie Mellon U.
RecordReader

- The input split defines a slice of work but does not describe how to access it
- The `RecordReader` class actually loads data from its source and converts it into (K, V) pairs suitable for reading by Mappers
- The RecordReader is invoked repeatedly on the input until the entire split is consumed
- Each invocation of the RecordReader lead to another call of the map function defined by the programmer

Files loaded from local HDFS store

Gregory Kesden, Carnegie Mellon U.
OutputFormat

- The `OutputFormat` class defines the way (K,V) pairs produced by Reducers are written to output files.

- The instances of `OutputFormat` provided by Hadoop write to files on the local disk or in HDFS.

- Several `OutputFormat`s are provided by Hadoop:

<table>
<thead>
<tr>
<th><code>OutputFormat</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TextOutputFormat</code></td>
<td>Default; writes lines in &quot;key \t value&quot; format</td>
</tr>
<tr>
<td><code>SequenceFileOutputFormat</code></td>
<td>Writes binary files suitable for reading into subsequent MapReduce jobs</td>
</tr>
<tr>
<td><code>NullOutputFormat</code></td>
<td>Generates no output files</td>
</tr>
</tbody>
</table>
Coordination: Master

- **Master node takes care of coordination:**
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its $R$ intermediate files, one for each reducer
  - Master pushes this info to reducers
  - **Master pings workers periodically to detect failures**
Dealing with Failures

- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker

- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted

- **Master failure**
  - MapReduce task is aborted and client is notified
How many Map and Reduce jobs?

- $M$ map tasks, $R$ reduce tasks

- **Rule of a thumb:**
  - Make $M$ much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures

- **Usually $R$ is smaller than $M$**
  - Because output is spread across $R$ files
Task Granularity & Pipelining

- **Fine granularity tasks:** map tasks >> machines

  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing

![Diagram showing task allocation and execution process]

<table>
<thead>
<tr>
<th>Process</th>
<th>Time ------------------------&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Program</td>
<td>MapReduce()</td>
</tr>
<tr>
<td>Master</td>
<td>... wait ...</td>
</tr>
<tr>
<td>Worker 1</td>
<td>Assign tasks to worker machines...</td>
</tr>
<tr>
<td>Worker 2</td>
<td></td>
</tr>
<tr>
<td>Worker 3</td>
<td></td>
</tr>
<tr>
<td>Worker 4</td>
<td></td>
</tr>
</tbody>
</table>
Refinements: Backup Tasks

• **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things

• **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first “wins”

• **Effect**
  - Dramatically shortens job completion time
Refinement: Combiners

- Often a Map task will produce many pairs of the form \((k,v_1), (k,v_2), \ldots\) for the same key \(k\)
  - E.g., popular words in the word count example

- **Can save network time by pre-aggregating values in the mapper:**
  - \(\text{combine}(k, \text{list}(v_1)) \rightarrow v_2\)
  - Combiner is usually same as the reduce function

- Works only if reduce function is commutative and associative
Refinement: Combiners

- **Back to the word counting example:**
  - Combiner combines the values of all keys of a single mapper (single machine):

  ![Diagram](image_url)

  - Much less data needs to be copied and shuffled!
Refinement: Partition Function

• **Want to control how keys get partitioned**
  
  • Inputs to map tasks are created by contiguous splits of input file
  
  • Reduce needs to ensure that records with the same intermediate key end up at the same worker

• **System uses a default partition function:**
  
  • hash(key) mod $R$

• **Sometimes useful to override the hash function:**
  
  • E.g., hash(hostname(URL)) mod $R$ ensures URLs from a host end up in the same output file
Problems Suited for Map-Reduce
Applications

• Three major classes:
  • Text tokenization, indexing, and search
  • Creation of other kinds of data structures (e.g., graphs)
  • Data mining and machine learning

• See list at https://cwiki.apache.org/confluence/display/HADOOP2/poweredby

• For Machine Learning algorithms, see MAHOUT at http://mahout.apache.org/
  • Default backend is now Spark
Example: Host size

• Suppose we have a large web corpus

• Look at the metadata file
  
  • Lines of the form: (URL, size, date, …)

• For each host, find the total number of bytes
  
  • That is, the sum of the page sizes for all URLs from that particular host

• Other examples:
  
  • Link analysis and graph processing
  
  • Machine Learning algorithms
Example: Language Model

• **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

• **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts
Example: Database Join By Map-Reduce

- Compute the natural join $R(A,B) \bowtie S(B,C)$
- $R$ and $S$ are each stored in files
- Tuples are pairs $(a,b)$ or $(b,c)$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$b_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_2$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$b_3$</td>
<td>$c_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_3$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$c_2$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$c_3$</td>
</tr>
</tbody>
</table>
Map-Reduce Join

• **Use a hash function** $h$ **from** $B$-values **to** $1...k$

• **A Map process turns:**
  
  • Each input tuple $R(a,b)$ into key-value pair $(b,(a,R))$
  
  • Each input tuple $S(b,c)$ into $(b,(c,S))$

• **Map processes** send each key-value pair with key $b$ to Reduce process $h(b)$
  
  • Hadoop does this automatically; just tell it what $k$ is.

• Each **Reduce process** matches all the pairs $(b,(a,R))$ with all $(b,(c,S))$ and outputs $(a,c)$.
Cost Measures for Algorithms

- In MapReduce we quantify the cost of an algorithm using

  1. **Communication cost** = total I/O of all processes

  2. **Elapsed communication cost** = max of I/O along any path

  3. **(Elapsed) computation cost** analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)
Example: Cost Measures

• For a map-reduce algorithm:

  • **Communication cost** = input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.

  • **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process.
What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud provider
- Elapsed cost is wall-clock time using parallelism
Cost of Map-Reduce Join

- **Total communication cost**
  \[ = \mathcal{O}(|R|+|S|+|R \bowtie S|) \]

- **Elapsed communication cost**
  \[ = \mathcal{O}(s) \]

- We’re going to pick \( k \) and the number of Map processes so that the I/O limit \( s \) is respected

- We put a limit \( s \) on the amount of input or output that any one process can have. \( s \) could be:
  - What fits in main memory
  - What fits on local disk

- With proper indexes, computation cost is linear in the input + output size

- So computation cost is like comm. cost
Pointers and Further Reading
Implementations

- **Google**
  - Not available outside Google

- **Hadoop**
  - An open-source implementation in Java
  - Uses HDFS for stable storage

- **Amazon Elastic MapReduce (EMR)**
  - Hadoop MapReduce running on Amazon EC2
  - Can also run Spark, HBase, Hive, Presto …
Reading

• Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters
  • [https://research.google.com/archive/mapreduce-osdi04.pdf](https://research.google.com/archive/mapreduce-osdi04.pdf)

• Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
Resources

• Hadoop Resources
  • Introduction
    • https://hadoop.apache.org/
  • Map/Reduce Overview
    • https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
  • Eclipse Environment
    • https://people.apache.org/~srimanth/hadoop-eclipse/
  • Javadoc
    • http://hadoop.apache.org/docs/stable/api/