Getting stuff done with Big Data
Lecture Two: Map Reduce and Hadoop

Miles Osborne

School of Informatics
University of Edinburgh
miles@inf.ed.ac.uk

February 11, 2012
Map Reduce
  Major Components
  Critique

MR Programming Model
  Examples
  Efficiency

Hadoop
  Examples
Background

MR is a parallel programming model and associated infrastructure introduced by Google in 2004:

- Assumes large numbers of cheap, commodity machines.
- Failure is a part of life.
- Tailored for dealing with Big Data
- Simple
- Scales well
Background

Early Google Server (source: nialkennedy, flickr)
Background

Who uses it?

- Google (more than 1 million cores, rumours have it)
- Yahoo! (more than 100K cores)
- Facebook (8.8k cores, 12 PB storage)
- Twitter
- IBM
- Amazon Web services
- Edinburgh (!)
- Many many small start-ups

http://wiki.apache.org/hadoop/PoweredBy
MapReduce inside Google

Googlers' hammer for 80% of our data crunching

- Large-scale web search indexing
- Clustering problems for Google News
- Produce reports for popular queries, e.g. Google Trend
- Processing of satellite imagery data
- Language model processing for statistical machine translation
- Large-scale machine learning problems
- Just a plain tool to reliably spawn large number of tasks
  - e.g. parallel data backup and restore

The other 20%? e.g. Pregel

Source: Zhao et al, Sigmetrics 09
## Use of MapReduce inside Google

<table>
<thead>
<tr>
<th>Stats for Month</th>
<th>Aug.'04</th>
<th>Mar.'06</th>
<th>Sep.'07</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of jobs</td>
<td>29,000</td>
<td>171,000</td>
<td>2,217,000</td>
</tr>
<tr>
<td>Avg. completion time (secs)</td>
<td>634</td>
<td>874</td>
<td>395</td>
</tr>
<tr>
<td>Machine years used</td>
<td>217</td>
<td>2,002</td>
<td>11,081</td>
</tr>
<tr>
<td>Map input data (TB)</td>
<td>3,288</td>
<td>52,254</td>
<td>403,152</td>
</tr>
<tr>
<td>Map output data (TB)</td>
<td>758</td>
<td>6,743</td>
<td>34,774</td>
</tr>
<tr>
<td>reduce output data (TB)</td>
<td>193</td>
<td>2,970</td>
<td>14,018</td>
</tr>
<tr>
<td>Avg. machines per job</td>
<td>157</td>
<td>268</td>
<td>394</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unique implementations</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapper</td>
<td>395</td>
<td>1958</td>
<td>4083</td>
</tr>
<tr>
<td>Reducer</td>
<td>269</td>
<td>1208</td>
<td>2418</td>
</tr>
</tbody>
</table>

*From "MapReduce: simplified data processing on large clusters"*
Components

Major components:

1. MR task scheduling and environment
   - Running jobs, dealing with moving data, coordination, failures etc

2. Distributed File System (DFS)
   - Storing data in a robust manner across a network; moving data to nodes

3. Distributed Hash Table (BigTable)
   - Random-access to data that is shared across the network

Hadoop is an open-source version of 1 and 2; HBase (etc) are similar to 3
Tasks are run in parallel across the cluster(s):

- Computation moves to the data.
- Multiple instances of a task may be run at once
  - *Speculative execution* guards against task failure
- Tasks can be run *rack-aware*:
  - Tasks access data that is within the rack they are running on
Data is stored across one or more clusters:

- Files are stored in *blocks*
- Blocks size is optimised for disk-cache size (often 64M)
- Blocks are replicated across the network
  - Replication adds fault tolerance
  - Replication increases the chance that the data is on the same machine as the task needing it
- Blocks are read sequentially and written sequentially
- Blocks are also spread evenly across the cluster
Files are often big:

- 100s of GB or more
- Few, big files mean less overheads
- Hadoop currently does not support appending
  - Appending to a file is natural for streaming input
- Under Hadoop, blocks are write-only.
MR

Tasks and data are centrally managed:
  ► Dash-board to monitor and manage progress
  ► Under Hadoop, this is a single-point of failure
Possibility of moving jobs across data centres
  ► Take advantage of cheap electricity
  ► Deals with load-balancing, disasters etc
BigTable is a form of Database:

- Based on *shared-nothing* architecture
- Petabyte scaling, across thousands of machines
- Has a simple data model
- Designed for managing structured data
  - Storing Web pages, URLs, etc
  - Key-value pairs
- BigTable provides random access to data
- Can be used as a source and sink for MR jobs
Programming Model

MR offers one restricted version of parallel programming:
- Coarse-grained.
- No inter-process communication.
- Communication is (generally) through files.
Programming Model

Mapping:

- The input data is divided into *shards*.
- The *Map* operation works over each shard and *emits key-value* pairs.
- Each mapper works in parallel.

Keys and values can be anything which can be represented as a string.
Reducing:

- After mapping, each key-value pair is hashed on the key.
- Hashing sends that key-value pair to a given reducer.
  - All keys that hash to the same value are sent to the same reducer.
- The input to a reducer is sorted on the key.
  - Sorted input means that related key-value pairs are locally grouped together.
Programming Model
Programming Model

Note:

- Each Mapper and Reducer runs in parallel.
- There is no state sharing between tasks.
  - Task communication is achieved using either external resources or at start-time
- There need not be the same number of Mappers as Reducers.
  - It is possible to have no Reducers.
Programming Model

Note:

- Tasks read their input sequentially.
  - Sequential disk reading is far more efficient than random access
- Reducing starts once Mapping ends.
  - Sorting and merging etc can be interleaved.
Aside: Map-Reduce in one line

Under Unix, you can quickly test a MR job:

```
% cat input | mapper | sort -0 +1 | reducer > output
```

`mapper` is your Mapper and `reducer` is the Reducer
Example: Tokenisation

Example

Convert *John’s* to *John +s*

- The input data will be a list of documents
- The output will be a list of tokenised documents
- There is no need to run a reducing stage
Example: Tokenisation

Mapper:
- Tokeniser reads input
- Emits tokenised output

Sentence ordering may not be honoured (how can we do this?)
Example: Word Counting

Example

Count the number of words in a collection of documents

- Our Mapper counts words in each shard.
- The Reducer gathers together partial counts for a given word and sums them
Example: Word Counting

Mapper:

- For each sentence, emit word, 1 pair.
  - The key is the word
  - The value is the number 1
Example: Word Counting

Reducer:

- Each Reducer will see all instances of a given word.
- Sequential reads of the reducer input give partial counts of a word.
- Partial counts can be summed to give the total count.
Example: Word Counting

Input sentences:

- the cat
- the dog

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>dog</td>
<td>1</td>
</tr>
</tbody>
</table>

Mapper output
Example: Word Counting

Reducer 1 input
- the, 1
- the, 1
- dog, 1

Reducer 1 output
- the, 2
- dog, 1

Reducer 2 input
- cat, 1

Reducer 2 output
- cat, 1
Map Reduce Efficiency

MR algorithms involve a lot of disk and network traffic:

- We typically start with Big Data
- Mappers can produce intermediate results that are *bigger* than the input data.
- Task input may not be on the same machine as that task.
  - This implies network traffic
- Per-reducer input needs to be sorted.
Sharding might not produce a balanced set of inputs for each Reducer:

- Often, the data is heavily skewed
  - Eg all function words might go to one Reducer
- Having an imbalanced set of inputs turns a parallel algorithm into a sequential one
Map Reduce Efficiency

Selecting the right number of Mappers and Reducers can improve speed

- More tasks mean each task might fit in memory / require less network access
- More tasks mean that failures are quicker to recover from.
- Fewer tasks have less of an over-head.

This is a matter of guess-work
Algorithmically, we can:

- Emit fewer key-value pairs
  - Each task can locally aggregate results and periodically emit them.
  - (This is called *combining*)
- Change the key
  - Key selection implies we partition the output. Some other selection might partition it more evenly
Midway Summary

- Introduced MR and the MR programming model
- Sample MR applications
- Looked at efficiency
History

*Nutch* started in 2002 by Doug Cutting and Mike Cazfarella

- Early open-source web-search engine
- Written in Java
- Realisation that it would not scale for the Web
  - 2004: Google MR and GFS papers appeared
  - 2005: Working MR implementation for Nutch
  - 2006: Hadoop became standalone project
- 2008: Hadoop broke world record for sorting 1TB of data
Hadoop Overview

Set of components (Java), implementing most of MR-related ecosystem:

- MapReduce
- HDFS (Hadoop distributed filesystem)
- Services on top:
  - HBase (BigTable)
  - Pig (sql-like job control)
- Job-control
Hadoop supports a variety of ways to implement MR jobs:

- Natively, as java
- Using the ‘streaming’ interface
  - Mappers and reducers can be in any language
  - Performance penalty, restricted functionality
- C++ hooks etc
Word Counting

Word counting using Hadoop:
- Use HDFS
- Specify the MR program
- Run the job

Note: all commands are for Hadoop 0.19
Word Counting

First need to upload data to HDFS
  ▶ Hadoop has a set of filesystem-like commands to do this:
    ▶ hadoop dfs -mkdir data
    ▶ hadoop dfs -put file.txt data/
  ▶ This creates a new directory and uploads the file file.txt to HDFS
  ▶ We can verify that it is there:
    ▶ hadoop dfs -ls data/
Word Counting

Mapper:

- Using Streaming, a Mapper reads from STDIN and writes to STDOUT
- Keys and Values are delimited (by default) using tabs.
- Records are split using newlines
Word Counting

Mapper:

```java
while !eof(STDIN) do
    line = readLine(STDIN)
    wordList = split(line)
    foreach word in wordList do
        print word TAB 1 NEWLINE
    end
end
```
Word Counting

Reducer

1 prevWord = ";" ; valueTotal = 0
2 while !eof(STDIN) do
3     line = readLine(STDIN); (word,value) = split(line)
4     if word eq prevWord or prevWord eq "" then
5         valueTotal += value
6         prevWord = word
7     else
8         print prevWord valueTotal NEWLINE
9         prevWord = word; valueTotal = value
10     end
11 end
12 print word valueTotal NEWLINE
Word Counting

Improving the Mapper

wordsCounts = {}

while !eof(STDIN) do
  line = readLine(STDIN)
  wordList = split(line)
  foreach word in wordList do
    wordCounts{word}++
  end
end

foreach word in keys(wordCounts) do
  count = wordCounts{word}
  print word TAB count NEWLINE
end
Our improved Mapper:

- Only emits one word-count pair, per word and shard
- This uses a *Combiner* technique
- Uses an unbounded amount of memory

Word counting 2 million tokens (Unix MR simulation)

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>1 minute 5 sec</td>
</tr>
<tr>
<td>Combiner</td>
<td>10 sec</td>
</tr>
</tbody>
</table>

How can you change it to use a *bounded* amount of memory?
Secondary Sorting

At times, we may want to resort the Reducer input:

- Hadoop only guarantees that the same keys are grouped together
- We may want to ensure that some key occurs before other ones
  - Eg when estimating the parameters of models we may want a normalising constant first
## Secondary Sorting Example

Reducer input:

<table>
<thead>
<tr>
<th>Ordinary</th>
<th>Resorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>loves mary 1</td>
<td>loves NULL 3</td>
</tr>
<tr>
<td>loves NULL 3</td>
<td>loves bob 2</td>
</tr>
<tr>
<td>loves bob 2</td>
<td>loves mary 1</td>
</tr>
</tbody>
</table>

After reading each line we can immediately emit probabilities
Critique

MR has generated a lot of interest:

- It solves all scaling problems!
- Google use it, so it must be great
- Start-ups etc love it and they generate a lot of chatter in the Tech Press
  - Big companies use DBs and they don’t talk about it
- Who needs complicated, expensive DBs anyway
Google Trends: Blue (DBMS mentions), Red (Hadoop mentions)
Critique

Stonebraker et al considered whether MR can replace parallel databases

- P-DBs have been in development for 20+ years
- Robust, fast, scalable
- Based upon declarative data models
Critique

Which application classes might MR be a better choice than a P-DB?

► *Extract-transform-load* problems
  ► Read data from multiple sources
  ► Parse and clean it
  ► Transform it
  ► Store some of the data

► *Complex analytics*
  ► Multiple passes over the data
  ► Computing complex reports etc

► *Semi-structured data*
  ► No single scheme for the data (eg logs from multiple sources)

► *Quick-and-dirty analyses*
  ► Asking questions over the data, with minimum fuss and effort
Critique

Resulted indicated:

- For a range of core tasks, a P-DB was faster than Hadoop
  - P-DBs are flexible enough to deal eg with semi-structured data
- (Unclear whether this is implementation-specific)
- Hadoop was criticised as being too low-level
  - Higher-level abstractions such as Pig might help
- Hadoop was easier for quick-and-dirty tasks
  - Writing MR jobs can be easier than complex SQL queries
  - Non-specialists can quickly write MR jobs
- Hadoop is a lot cheaper
Critique

MR is not really suited for low-latency problems:

- Batch nature and lack of real-time guarantees means you shouldn’t use it for front-end tasks

MR is not a good fit for problems which need global state information:

- Many Machine Learning algorithms require maintenance of centralised information and this implies a single task

Use the right tool for the job
Summary

- History
- Looked at major components
- Two examples
- Critique