COMP 330: BIG DATA: AN INTRO TO MAPREDUCE

Chris Jermaine
cmj4@cs.rice.edu

15 Years Ago...

- Say you had a “big” data set, wanted platform to analyze it
  - — Analysis: report generation, customer profiling, statistical modelling, etc.
- What do I mean by “big”?
  - — Too large to fit in aggregate RAM of a standard distributed/parallel system
- Big is 100GB+ in 2002, TB+ now (all the way to dozens of PB)

Hardware: Agreement on "Shared Nothing"

- Store/analyze data on a large number of commodity machines
- Local, non-shared storage attached to each of them
- Only link is via a LAN
- "Shared nothing" refers to no sharing of RAM, storage
- Why preferred?
  - — Inexpensive: built out of commodity components (same stuff as in desktop PCs)
  - — You are leveraging price/tech wars among Dell, HP, Intel, AMD, etc.
  - — Compute resources scales nearly linearly with $$
  - — Contrast this to a shared RAM machine with uniform memory access

But What About the Software?

- 15 years ago, you'd have two primary options
  - — Put your data into an SQL database, or
  - — Roll your own software stack to perform your analysis

Clearly, Building Own Software Not Desirable

- Costly, time consuming
  - — A $10M software feature might eat up most of the IT budget for a single firm
  - — But Oracle can spread those costs across 100K customers
- Requires expertise not always found in house
- Risky: high potential for failure

But People Not Happy With SQL Databases

- Also quite expensive: even today, pay $5K to $50K/year/TB
- Performance often unpredictable, or just flat out poor
  - — Only now are there systems that mortals can get to work in TB+ range
- In 2004, not a lot of options for commodity shared nothing
- Software insanely complicated to use correctly
  - — Hundreds or even thousands of “knobs” to turn
- Software stack too big/deep, not possible to unbundle
  - — If you are doing analysis, ACID not important
  - — And yet, you pay for it ($$, complexity, performance)
- Difficult to put un- or semi-structured data into an SQL DB
  - — How does an archive of 10M emails get put into a set of relations?

And, Many People Just Don't Like SQL

- It is "declarative"
  - — In some ways, very nice, since parallelism is implicit
  - — But user doesn't really know what's happening under the hood... people don't like
- Also, not easy/natural to specify important computations
  - — Such as Google's PageRank, or rule mining, or data clustering, etc.

By Early-Mid 2000's...

- The Internet companies (Google, Yahoo!, etc.)...
  - — ...had some of the largest databases in the world
- But they never used classical SQL databases for webscale data
- How'd they deal with all of the data they had to analyze?
  - — Many ways
  - — But paradigm with most widespread impact was **MapReduce**
  - — First described in a 2004 academic paper, appeared in OSDI
  - — Easy read:

    http://research.google.com/archive/mapreduce.html

What Is MapReduce?

- It is a simple data processing paradigm
- To process a data set, you have two pieces of user-supplied code:
  - A **map** code
  - And a **reduce** code
- These are run (potentially over a large compute cluster) using three data processing phases
  - A **map** phase
  - A **shuffle** phase
  - And a **reduce** phase

The Map Phase

- Assume that the input data are stored in a huge file

  This file contains a simple list of pairs of type `(key1,value1)`

- And assume we have a user-supplied function of the form

  ```
  map (key1,value1)
  ```

- That outputs a list of pairs of the form `(key2, value2)`
- In the **map** phase of the MapReduce computation
  - — this `map` function is called for every record in the input data set
  - — Instances of `map` run in parallel all over the compute cluster

The Shuffle Phase

- The **shuffle** phase accepts all of the `(key2, value2)` pairs from the **map** phase
- And it groups them together
- So that all of the pairs
  - — From all over the cluster
  - — Having the same `key2` value
  - — Are merged into a single `(key2, list <value2>)` pair
- Called a **shuffle** because this is where a potential all-to-all data transfer happens

The Reduce Phase

- Assume we have a user-supplied function of the form

```
reduce (key2,list <value2>)
```

- That outputs a list of `value2` objects
- In the **reduce** phase of the MapReduce computation
  - this `reduce` function is called for every `key2` value output by the shuffle
  - Instances of `reduce` run in parallel all over the compute cluster
  - The output of all of those instances is collected in a (potentially) huge output file

The Distributed File System

- Now MapReduce is a **compute** paradigm
- It is not a **data storage** paradigm
- But any MapReduce system must read/write data from some storage system
- As a result, the MapReduce programming paradigm is tightly integrated with the idea of a **distributed file system (DFS)**
- A DFS is a storage system that allows data to be stored/accessed across machines in a network
- And abstracts away differences between local and remote data
  - — Same mechanism to read/write data
  - — No matter where data is located in the network

Distributed File Systems for MR

- DFSs have been around for a long time
  - — First widely used DFS was Sun's NFS, first introduced in 1985
- How is a DFS for MapReduce going to be different?
- Unlike classical DFSs, it sits on top of each machine's OS
- The OS is not aware of the DFS; you can't "mount" it anywhere
  - — So the files in the DFS are not accessible from "My Computer" in Windows
- Why "on top of" rather than "in" the OS?
  - — Ease of use, portability, means no worries with a heterogeneous cluster
  - — Just start up a process on each machine in the cluster
  - — No need to tell the OS about anything
  - — Means you can have a DFS up and running on a cluster in minutes/hours

Distributed File Systems for MR

- But (in theory) they still give you most of what a classic DFS does
- Replication
  - — Put each block at n locations in the cluster
  - — That way, if a disk/machine goes down, you are still OK
- Network awareness
  - — Smart enough to try to satisfy a data request locally, or from same rack
- Easy to add/remove machines
  - — You buy 10 more machines, just tell the DFA about them, and it'll add data to 'em
  - — Can take machines off the network; no problem, DFS will realize this and handle
- Load balancing
  - — If one machine is getting hit, go to another machine that has the data

Take Home Message From Last 10 Slides

- MapReduce is a distributed programming paradigm
- Needs to run on top of some storage system--a DFS
- DFS should be lightweight, easy to install, OS agnostic
- Thus, you can expect most MR softwares to be tightly integrated with a particular DFS
  - — And that DFS will typically run **on top of** the OS of each machine

MapReduce Had a Huge Impact

- One of the key technologies in the "NoSQL movement"
- What do people like about it?

Why Popular? (1)

- Schema-less
- You write code that operates over raw data
- No need to spend time/$$ loading the data into a database system
- Really nice for unstructured/semi-structured data: text, logs, for example

Why Popular? (2)

- Easier to code than classical HPC system
- Distributed/parallel computing **very** difficult to program correctly
  - — pthreads, MPI, semaphores, monitors, condition variables...
  - — All are hard to use!
- But MapReduce is a super-simple compute model
- All communication is done during shuffle phase
- All scheduling is taken care of by the MapReduce system
- Radically reduces complexity for the programmer

Why Popular? (3)

- Much more control than an SQL database
- You write the actual code that touches the data
  - — In a standard language, such as Java
- You control what happens during the map and reduce phases
- Contrast to an SQL database
  - — Where you write SQL that can be compiled to an arbitrary execution plan

Why Popular? (4)

- Fits very nicely into the “cloud computing” paradigm
- Why? So simple, lightweight, hardware agnostic
- Need to run a MapReduce computation?
  - — Just rent a bunch of machines from Amazon
  - — Give ‘em back when you are done
- Contrast this with a multi-terabyte Oracle database
  - — Most SQL databases are NOT lightweight and simple to get going in a few mins

Why Popular? (5)

- Software is free!

By 2010, Hadoop Was Widely Used

- Open-source implementation of Google MapReduce
  - — Had Hadoop MapReduce
  - — Plus Hadoop distributed file system
- But the cracks were appearing in the edifice...

Hadoop MR Word Count Java Code

```
import java.util.*;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;

public class WordCount {

  public static int main(String[] args) throws Exception {

    // if we got the wrong number of args, then exit
    if (args.length != 4 || !args[0].equals ("-r")) {
      System.out.println("usage: WordCount -r <num reducers> <input> <output>");
      return -1;
    }

    // Get the default configuration object
    Configuration conf = new Configuration ();

    // now create the MapReduce job
    Job job = new Job (conf);
    job.setJobName ("WordCount");

    // we'll output text/int pairs (since we have words as keys and counts as values)
    job.setMapOutputKeyClass (Text.class);
    job.setMapOutputValueClass (IntWritable.class);

    // again we'll output text/int pairs (since we have words as keys and counts as values)
    job.setOutputKeyClass (Text.class);
    job.setOutputValueClass (IntWritable.class);

    // tell Hadoop the mapper and the reducer to use
    job.setMapperClass (WordCountMapper.class);
    job.setCombinerClass (WordCountReducer.class);
    job.setReducerClass (WordCountReducer.class);

    // we'll be reading in a text file, so we can use Hadoop's built-in TextInputFormat
    job.setInputFormatClass (TextInputFormat.class);

    // we can use Hadoop's built-in TextOutputFormat for writing out the output text file
    job.setOutputFormatClass (TextOutputFormat.class);

    // set the input and output paths
    TextInputFormat.setInputPaths (job, args[2]);
    TextOutputFormat.setOutputPath (job, new Path (args[3]));

    // set the number of reduce paths
    try {
      job.setNumReduceTasks (Integer.parseInt (args[1]));
    } catch (Exception e) {
      System.out.println("usage: WordCount -r <num reducers> <input> <output>");
      return -1;
    }

    // force the mappers to handle one megabyte of input data each
    TextInputFormat.setMinInputSplitSize (job, 1024 * 1024);
    TextInputFormat.setMaxInputSplitSize (job, 1024 * 1024);

    // this tells Hadoop to ship around the jar file containing "WordCount.class" to all of
the different
    // nodes so that they can run the job
    job.setJarByClass(WordCount.class);

    // submit the job and wait for it to complete!
    int exitCode = job.waitForCompletion (true) ? 0 : 1;
    return exitCode;
```

Not pretty!
Programmer burden too high...

Many Felt Hadoop MR Too Slow

- Data read from HDFS again for each MR job
- Bad for iterative data processing
  - — Many analytics tasks process and reprocess data

And API Too Restrictive

- Can only do Map
- Or MapReduce
- Everything else in terms of those operations... annoying!

So Hadoop MR Used Less and Less

- Issues led to other dataflow platforms to replace Hadoop MR
  - — Notable are Spark and Flink
  - — We will study Spark
- Interesting fact:
  - — While Hadoop MR is dying...
  - — Hadoop DFS is going strong
  - — De-facto standard for Big Data management
  - — Looks set to stay that way for a very long time

Apache Spark

- #1 Hadoop MR killer
- What is Spark?
  - — Platform for efficient, distributed data analytics
  - — More in the future!!

Questions?