The MapReduce Framework

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

November 8, 2007
Google, Yahoo, etc. deal with
- very large amounts of data (many terabytes)
- need to process data fairly quickly (within a day, e.g.)
- use very large numbers of commodity machines (thousands)

Google developed an infrastructure consisting of
- the Google distributed file system GFS
- the MapReduce computational model

Other implementations include Hadoop from Apache.
Requirements and Constraints

- Want to run on 1,000–10,000 nodes.
- With that many nodes
  - some will fail
  - some will go down for maintenance

Fault tolerance is essential.

- Want to work on petabytes of data
- Data will need to be distributed across many disks.
- Data access speeds will depend on location:
  - local disk will be fastest
  - same rack may be faster than different rack
- Replication is needed for performance and fault tolerance.
Requirements and Constraints

- Want an infrastructure that takes care of management tasks
  - distribution of data
  - management of fault tolerance
  - collecting results
- For a specific problem
  - user writes a few routines
  - routines plug into the general interface
- Goal: identify a class of computations that is
  - general enough to cover many problems
  - structured enough to allow development of an infrastructure
  - reasonably easy to tailor to specific problems
- MapReduce seems to fit this goal reasonably well.
Related to two concepts from functional programming:

- **Mapping**: applying a function to each element of a structure and returning a comparable structure of results. R/S use the term `apply`.
- **Reducing or folding**: Applying a binary operation, usually associative, often commutative, to an initial element and every successive element of a structure to produce a single reduced result, e.g. a sum.

R 2.6.0 has recently introduced some functional programming primitives, including `Map` and `Reduce`.

The names come from the Lisp world.

A useful running example: Counting word frequencies in a collection of documents.
MapReduce operations work with key/value pairs, e.g.
- document name/document content
- word/count

A general MapReduce computation has several components:
- Input reader: reads input files and divides into chunks for the map function
- map function: receives key/value pair and emits 0 or more key value pairs.
- Partition function: allocates output of maps to particular reduce functions.
- Comparison function: used in sorting map output by keys.
- reduce function: takes a key and a collection of values and produces a key/value pair.
- Output writer: writes results to storage.

All components can be customized.
Often only **map** and **reduce** need to be written.

For simple word counting, **map** might look like

```java
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

The key is ignored.

The **reduce** function might look like

```java
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```
Google’s MapReduce is implemented as a C++ library.

- Operates on commodity hardware and standard networking.
- Input data, intermediate results, and final results are stored in GFS.
- A master scheduler process distributes map, reduce tasks to workers.

Fault tolerance:
- The master pings workers periodically.
- Workers that do not respond are marked as failed.
- Jobs assigned to failed workers are rerun.
- Master failure aborts the computation.
Hadoop is part of the Apache Lucene project for open-source search software.

Hadoop is used heavily by Yahoo, among others.

There is support for running Hadoop jobs on Amazon EC2/Amazon S3.

Hadoop includes
- a distributed file system, HDFS.
- a MapReduce framework.
- a web monitoring interface.

Hadoop is written in Java and can be extended in Java.

A mechanism for extension via C/C++ is also available.

A streaming interface using standard I/O can also be used.

The streaming interface is the easiest way to use Python or R.
A Python example from the Wiki is easily adapted to R.
I set up a simple test framework on my workstation.
Eventually we may wish to add this to beowulf.
The streaming interface uses batches of lines from text files as inputs.
It requires mapper and reducer executables or scripts.
The mapper produces lines of the form key<tab>value for the reducer.
R script `mapper.R` to read lines from standard input and print
word<tab>1
for each word to standard output:

```r
#! /usr/bin/env Rscript

trimWhiteSpace <- function(line) gsub("(^ +)|( +$)", "", line)
splitIntoWords <- function(line) unlist(strsplit(line, "[[:space:]]+"))

con <- file("stdin", open = "r")
while (length(line <- readLines(con, n = 1, warn = FALSE)) > 0) {
    line <- trimWhiteSpace(line)
    words <- splitIntoWords(line)
    cat(paste(words, "\t1\n", sep=""), sep="")
}
close(con)
```

R script `reducerer.R` to read word/count pairs and emit word/sum pairs
is a little longer.
Apache Hadoop
Word Count Example

- Data are files from project Gutenberg.
- Steps to running the example:
  - Start up hadoop.
  - Copy data to HDFS.
  - Run MapReduce.
  - Copy results back from HDFS.
  - Shut down hadoop

- Starting up code:
  ```bash
  setenv HADOOP_INSTALL /home/luke/hadoop/hadoop
  $HADOOP_INSTALL/bin/start-all.sh
  ```

- Copying data to HDFS:
  ```bash
  $HADOOP_INSTALL/bin/hadoop dfs -copyFromLocal gutenberg gutenberg
  ```
Running the MapReduce:

```
$HADOOP_INSTALL/bin/hadoop jar \
   $HADOOP_INSTALL/contrib/hadoop-streaming.jar \
   -mapper /home/luke/hadoop/mapper.R \ 
   -reducer /home/luke/hadoop/reducer.R \ 
   -input 'gutenberg/*' -output gutenberg-output
```

Looking at the results:

```
$HADOOP_INSTALL/bin/hadoop dfs -cat \
   gutenberg-output/part-00000 | more
...
Abaft   1
abandon 7
abandoned 7
abandoned, 2
...
```
● With minor modifications we can count the number of movies reviewed by each customer in the Netflix data.

● It is useful to change the movie files from

```
17767:
1428688,3,2005-08-09
656399,3,2005-08-19
1356914,4,2005-05-27
1526449,4,2005-10-20
...
```

to

```
17767,1428688,3,2005-08-09
17767,656399,3,2005-08-19
17767,1356914,4,2005-05-27
17767,1526449,4,2005-10-20
...
```
The mapper script `nmapper.R` is

```r
#!/usr/bin/env Rscript

con <- file("stdin", open = "r")
while (length(line <- readLines(con, n = 1, warn = FALSE)) > 0) {
    vals <- unlist(strsplit(line, ","))
    cat(vals[2], "\t", 1, "\n", sep="")
}

close(con)
```

The reducer remains the same.

The results for 3 movie files:

```
$HADOOP_INSTALL/bin/hadoop dfs -cat netflix-output/part-00000 | more
...  
1001833 1
1001928 2
...
1664010 3
1664458 1
...  
```
Many statistical computations can be expressed via MapReduce:
- simple summaries
- least squares regression
- $k$-means clustering
- logistic regression (needs a sequence of MapReduce operations)

Languages for managing MapReduce computations are in development:
- Apache PIG project
- Google Sawzall

Some extensions are also under consideration
- map-reduce-merge

A number of frameworks supporting MapReduce are in development.