MapReduce: Simplified Data Processing on Large Clusters

Stacey Heideloff
CIS 601
9/18/2018
Motivation

- Big Data Processing Challenges
  - Size of Data
  - Speed of Processing

- Why using a single machine is too slow
  - CPU contention
    - Data must wait until CPU is ready for it
  - RAM limitations
    - Data is way too big to fit in RAM
    - Disk I/O wastes a lot of time
Solution

- Parallel processing
  - Distribute the job among many machines
  - Machines concurrently process the portions they are assigned

- Benefits
  - Speed
    - Add computational power: more CPUs working at once
    - Reduce the size of data processed by each machine
  - Fault Tolerance
    - Can replicate portions of work to reduce the impact of a machine failure
What is MapReduce?

- MapReduce Programming Model
  - Conception
    - Proposed by Jeffrey Dean and Sanjay Ghemawat (Google)
    - Paper published in 2004
  - Paradigm
    - Parallelizes and executes one computation on a large cluster of machines
    - Programmer only needs to know the logic of his computation
      - Does not need to know how to parallelize that computation
    - Uses commodity machines to process the data
  - Scale
    - Can process TBs of data on 1000s of machines
What is MapReduce?

Components

- MapReduce Library
  - Parallelization logic
  - Fault-tolerance
  - Data distribution
  - Load balancing

- Runtime System
  - Partitions input data
  - Schedules execution across cluster of machines
  - Manages inter-machine communication
  - Handles machine failures

- User-Written Map and Reduce Functions
  - Contains the user’s computation logic
  - Code that extracts desired information from the input data
MapReduce Programming Model

**USER CODE**

Map Function
- Transforms data into intermediate data
- Input: key/value pair
- Output: intermediate key/value pair

Reduce Function
- Invoked via an iterator
- Input: intermediate key/value pair
- Output: 0 or 1 output value per invocation of Reduce

**MAPREDUCE CODE**

MapReduce Library
- Groups together all intermediate values associated with the same intermediate key
- Passes results to Reduce function

MapReduce Specification Object
- Provide names of input and output files
- Optional tuning parameters for MapReduce
MapReduce Parameters

- **M**: Number of Input files
  - Pick a number that will divide each task into 16-64MB input data
  - Should be much larger than the number of worker machines
- **R**: Number of Output files
  - Make a small multiple of the number of worker machines
- **Example Values**
  - Worker machines = 2,000
  - $M = 200,000$
  - $R = 5,000$
Word Count Example

Find the number of occurrences of each word in a large collection of documents

<table>
<thead>
<tr>
<th>Input</th>
<th>Map</th>
<th>Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Document Name, Text Content of Document)</td>
<td>(bar, {&quot;1&quot;} ), (foo, {&quot;1&quot;, &quot;1&quot;, &quot;1&quot;} )</td>
<td>bar: &quot;1&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>foo: &quot;3&quot;</td>
</tr>
</tbody>
</table>

Output

{ (foo, "1"), (foo, "1"), (bar, "1"), (foo, "1") }

Pseudocode

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

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 Implementation

Individual Machine Specifications
- Dual-processor x86
- 2-4 GB Memory
- Linux

Number of Machines
- Large number of clusters
- Per cluster: 100s or 1,000s

Network
- Switched Ethernet
- 100 Mbps or 1 Gbps speed at machine-level

Storage
- Inexpensive IDE disks attached directly to individual machines

File System
- GFS (Google File System): Distributed file system
Google File System (GFS)

- Distributed File System
  - Created to work with large files
  - Designed for scalability
  - Uses clusters of commodity machines

- Nodes
  - Master: holds metadata for files
  - Chunkservers: store and serve files
  - Clients: request files

- Files
  - Broken into chunks
  - Chunk size: 64 MB
  - 64-bit chunk handle: global identifier
  - Replicated - 3 copies of each file

https://computer.howstuffworks.com/internet/basics/google-file-system.htm
Execution Steps

- MapReduce Library splits input files into M pieces
  - Optional user parameter for piece size
  - Typically 16-64 MB per piece

- MapReduce Library starts up multiple copies of the program on a cluster of machines
  - One copy designated “master”
  - All other copies “workers” assigned to the master

- “Master” assigns each worker a Map or Reduce Task
Execution Steps

- “Map” workers read contents of one piece of the split input
  - Parse key/value pairs out of the data
  - Pass each pair to the user-defined “Map” function
  - “Map” function produces intermediate key/value pairs
  - Intermediate pairs buffered in memory
Execution Steps

- Partitioning function divides pairs into R regions
  - Buffered pairs periodically written to disk
  - Locations of pairs on local disk given to “Master”
  - “Master” responsible for forwarding the locations to the “Reduce” workers
Execution Steps

- “Reduce” workers read intermediate data from local disks of the “Map” workers with remote procedure calls
  - Sort intermediate data by key
  - Group all occurrences of same key together
  - If data is too large to fit in memory, external sort is used
- “Reduce” worker finds each unique intermediate key
  - Passes key/value pair to user’s “Reduce” function
  - Output of “Reduce” function appended to final output file for this partition
Execution Steps

- “Master” wakes up user program and returns control to user code
  - output is received in R output files
    - one file per reduce task
  - output can be used:
    - in a distributed application that expects partitioned files
    - as input to another MapReduce call
MapReduce - Map Function

Distributed File System

Input File

Data Chunk
Data Chunk
Data Chunk
Data Chunk

Machine 1
Run
Map

Machine 2
Run
Map

Distributed File System

Input File

Data Chunk
Data Chunk
Data Chunk
Data Chunk

Buffered Result

MapResult
MapResult
MapResult

Machine 1

Machine 2
MapReduce - Partitioning Function

Distributed File System

Input File
- Data Chunk
- Data Chunk
- Data Chunk
- Data Chunk

Buffered Result
- MapResult
- MapResult
- MapResult
- MapResult

Intermediate Data
- IntResult
- IntResult

Partition

Machine 1
Machine 2
Machine 1
Machine 2
MapReduce - Reduce Function

Distributed File System

Input File
- Data Chunk
- Data Chunk
- Data Chunk
- Data Chunk

Intermediate Data
- IntResult
- IntResult

Global File System

Output File
- Reduce Result

Output File
- Reduce Result

Machine 1
- request data
- read Machine 1 data

Machine 2
- request data
- read Machine 2 data

Machine 3
- Run Reduce

Machine 4
- Run Reduce
Beyond Map and Reduce Functions

- Partitioning Function
  - Decides which output file a reduce result is placed in
  - Default partitioning function = hash(intermediate key) mod R
    - R = number of output files specified by user
  - User can provide their own function
  - Ordering guarantee: output files will be sorted within each partition

- Combiner Function
  - Optional user-specified function to be performed on “Map” workers
  - Partially processes “Map” results to reduce amount of data sent over network
  - Typically is same logic used in “Reduce” function
  - Significantly speeds up some MapReduce operations
Beyond Map and Reduce Functions

 Skipping Bad Records
  - Bugs in user code can cause “Map” or “Reduce” to crash on certain records
  - MapReduce Library can detect which records cause crashes and skip them
  - Option is best for cases when it is acceptable to ignore a few records
    - Ex: Statistical analysis on a large data set

 Status Information
  - “Master” task runs a server that exports status pages for users
    - Tasks completed
    - Tasks in progress
    - Bytes of input / intermediate data / output
    - Processing rates
    - Worker failures (with tasks being processed during failure)
  - Helps to debug user code
Implementation Challenges

- Machine Failures
  - With more machines involved, failure of at least one machine more likely

- Communication Costs
  - Network bandwidth major factor in speed of computation
  - Slow communication of results between machines could erode speed advantages of faster sub-computations

- “Straggler” Machines
  - Machines that take an unusually long time to complete its map or reduce task
  - Lengthens the total time taken by MapReduce computation
Google’s Solutions

- Machine Failures -> Rescheduling
  - “Master” periodically pings workers
  - “Master” marks unresponsive workers as failed
    - Reschedules failed worker’s tasks on another machine
    - Completed map tasks are re-executed because output is unavailable to the network
    - Completed reduce tasks are not re-executed because output is safe on global file system
  - If “master” task fails, entire MapReduce computation is aborted

- Communication Costs -> GFS
  - Input data is stored on local disks of machines in cluster
  - GFS stores several copies of each data block on different machines
  - “Master” can schedule map tasks on/near machines that contain the input data
    - Network bandwidth not used on local reads
Google’s Solutions

- “Straggler” Machines -> Backup Tasks
  - “Master” schedules backup executions of in-progress tasks
    - Near the end of the MapReduce operation when few tasks remain
  - Task marked as complete when either original or backup execution finishes
  - Tends to only increase computational resources by a few percent
  - Significantly reduces time to complete large MapReduce operations
The same MapReduce computation is run in (a) and (b)

- Sort ~1 TB of data

Effect of Backup Tasks

- (a) Total time: 891 sec
- (b) Total time: 1283 sec
  - 5 stragglers remain around 960 sec
  - 44% increase!
Performance Measurements

- The same MapReduce computation is run in (a) and (b)
  - Sort ~1 TB of data

- Effect of Killing Tasks
  - (a) Total time: 891 sec
  - (b) Total time: 933 sec
    - tasks re-executed on the killed workers themselves
    - only 5% increase

(a) Normal execution
(c) 200 tasks killed
Influence of This Paper

- **Apache Hadoop**
  - Open-source implementation of MapReduce
  - Designed for clusters of commodity machines
  - Written in Java
  - HDFS: Hadoop Distributed File System
  - First released in 2006

- **Cloud Vendors**
  - Microsoft: HDInsight on Azure
  - Amazon: Elastic MapReduce (EMR)
  - IBM: InfoSphere Insights
  - Cloudera
  - HortonWorks
  - MapR

https://en.wikipedia.org/wiki/Apache_Hadoop
Paper Referenced

- Title: MapReduce: Simplified Data Processing on Large Clusters
- Year: 2004
- Authors: Jeffrey Dean and Sanjay Ghemawat (Google, Inc.)
Thank You!