MapReduce: Simplified Data Processing on Large Clusters

Jeffry Dean and Sanjay Ghemawat

ODSI 2004

Presented by Anders Miltner
Problem

- When applied on to big data, simple algorithms become more complicated
  - Parallelized computations
  - Distributed data
  - Failure handling
- How can we abstract away these common difficulties, allowing the programmer to be able to write simple code for their simple algorithms?
Core Ideas

• Require the programmer to state their problem in terms of two functions, map and reduce
• Handle the scalability concerns inside the system, instead of requiring the programmer to handle them
• Require the programmer to only write code about the easy to understand functionality
Map, Reduce, and MapReduce

- map: (k1,v1) -> list (k2, v2)
- reduce: (k2, list v2) -> list (v2)
- mapreduce: list (k1,v1) -> list (k2, list v2)
Example

• Word Count in a lot of documents

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));
Architecture

- Large clusters of commodity PCs
- Input files partitioned into M splits
- Intermediate keys partitioned into R distributions
- Written to disk, remotely read by workers
Master vs Worker

• Only 1 master, many (thousands) workers
• Workers do the computations of maps and reduces
• Masters handle failures, assign tasks to workers
Failure Detection and Recovery

• Master pings workers to determine if they are failed
• Map worker failure:
  • Reexecute map tasks
• Reduce worker failure:
  • Redo if not completed, don’t otherwise
• Master failure:
  • Do nothing
Locality

- Use GFS for file storage of input data
- Map tasks assigned to minimize network time
Task Granularity

- Map phase has M pieces
- Reduce phase has R pieces
- \( M + R > \# \) machines
  - Dynamic load balancing
  - Worker failure recovery
Backup Tasks

• Stragglers happen
  • Hard disk issues
  • Bad scheduling

• Have back up executions
  • Executions complete when primary or backup complete

• Occurs intelligently
Refinements

• Partitioning Function
• Ordering Guarantees
• Combiner Function
• Input and Output Types
• Side-effects
• Skipping Bad Records
• Local Execution
• Status Information
• Counters
Performance Setup

• Measured on 1800 machine cluster
  • 2GHz processors
  • 4GB memory
  • 2 160GB disks
  • Gigabit Ethernet
Performance

- Grep
  - $10^{10}$ 100 byte records in 150s
- Sort
  - $10^{10}$ 100 byte records in 891s

![Performance Graph](image1)

Figure 3: Data transfer rates over time for different executions of the sort program

<table>
<thead>
<tr>
<th>Number of jobs</th>
<th>29,423</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average job completion time</td>
<td>634 secs</td>
</tr>
<tr>
<td>Machine days used</td>
<td>79,186 days</td>
</tr>
<tr>
<td>Input data read</td>
<td>3,288 TB</td>
</tr>
<tr>
<td>Intermediate data produced</td>
<td>758 TB</td>
</tr>
<tr>
<td>Output data written</td>
<td>193 TB</td>
</tr>
<tr>
<td>Average worker machines per job</td>
<td>157</td>
</tr>
<tr>
<td>Average worker deaths per job</td>
<td>1.2</td>
</tr>
<tr>
<td>Average map tasks per job</td>
<td>3,351</td>
</tr>
<tr>
<td>Average reduce tasks per job</td>
<td>55</td>
</tr>
<tr>
<td>Unique map implementations</td>
<td>395</td>
</tr>
<tr>
<td>Unique reduce implementations</td>
<td>269</td>
</tr>
<tr>
<td>Unique map/reduce combinations</td>
<td>426</td>
</tr>
</tbody>
</table>

Table 1: MapReduce jobs run in August 2004
Pros/Cons

• Pros
  • Nice abstraction
  • Resilient to failures
  • Lots of different extensions for different functionality
  • Speedy for a certain amount of data

• Cons
  • Doesn’t scale to giant amounts of data (master requires O(M*R) info)
  • Lot of disk IO
  • Requires batches to be completed before going to next stage
  • Not all problems can be expressed in terms of map and reduce
Further Work

• MapReduce Online
  • Pipeline the stages

• Google’s Cloud DataFlow
  • “Simple, powerful model for building batch and streaming parallel data processing pipelines”

• Hadoop
  • Utilizes mapreduce alongside storage
  • JobTracker and TaskTracker

• Piccolo
  • In memory

• Dryad
  • Directed graph of vertices and channels