MapReduce: Simplified Data Processing on Large Clusters

Authors: Jeffrey Dean and Sanjay Ghemawat

Presented by: Luna (Lilong) Teng
Outline

- Motivation & Overview
- Word Count Example & Implementation
- Structure & Execution Overview
- Fault Tolerance
- Refinements
- Performance
Outline

- **Motivation & Overview**
- Word Count Example & Implementation
- Structure & Execution Overview
- Fault Tolerance
- Refinements
- Performance
Motivation

- **Large amount** of raw data to process
- Conceptually straightforward computation
- Distributed computations → Finish in reasonable time
- Problems:
  - Parallelize computations
  - Distribute data
  - Handle failures
MapReduce: Overview

- A programming model & an associated implementation for processing and generating large data sets.
MapReduce: Overview

- First version in 2003 by Google
- Significant growth of usage

Applications:
- Large-scale machine learning problems
- Clustering problems for the Google News
- Extraction of data in queries &
- Extraction of properties of web pages
- Large-scale graph computations
- ...
- Rewrite the production indexing system for the Google web search service
What is MapReduce

- Inspired by the *map* and *reduce* primitives
  - In Lisp & other languages

- Advantages:
  - Allow **user defined** computations
  - **Hides messy details** in a library:
    - Parallelization
    - Fault-tolerance
    - Data distribution
    - Load balancing
Main idea

- **Map** operation:
  - Each Input record → key/value pair

- **Reduce** operation:
  - (same key) values → Derived data
Map Function

- User specified

- In: An input pair

- Out: A set of intermediate key/value pairs
Intermediate key/value pairs

- MapReduce library **groups** intermediate values with same key /
Reduce Function

- User specified

- In: Intermediate key \( I \) and a set of values for key \( I \)
- Out: Smaller set of values

- Typically 0 or 1 output value per Reduce invocation
User specification

- **Map & Reduce** functions
- Names of input & output files
- Optional tuning parameters
Outline

● Motivation & Overview
● **Word Count Example & Implementation**
● Structure & Execution Overview
● Fault Tolerance
● Refinements
● Performance
Example: Word Count

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

<table>
<thead>
<tr>
<th>DOC1</th>
<th>DOC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC</td>
<td>AAAA</td>
</tr>
</tbody>
</table>

**Intermediates**

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
</tr>
</tbody>
</table>
Example: Word Count

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));
Implementation of MapReduce

● Many possible implementations depend on the environment

● Here: A MapReduce interface tailored towards Google’s cluster-based computing environment
  ○ Build on Commodity PCs connected with switched Ethernet
  ○ Machine failures are common
Outline

- Motivation & Overview
- Word Count Example & Implementation
- Structure & Execution Overview
- Fault Tolerance
- Refinements
- Performance
Structure

- Single master
- A set of workers
Structure

- **M**: Partition input data into $M$ splits
  - Typically 16-64 MB per piece

- **R**: Partition intermediate key space into $R$ pieces
  - Using a partitioning function
  - E.g. $(\text{hash(key)} \mod R)$

- All specified by user
Execution Overview

- One master program
  - The rest are workers
  - Master assign map/reduce tasks to idle workers

Figure 1: Execution overview
Execution Overview

- One master program
  - The rest are workers
  - Master assign map/reduce tasks to idle workers
Execution Overview

- **Workers** perform **Map** task
  - Intermediate key/value pairs buffered in memory
  - Periodically write buffered pairs into **local disk**
  - Locations of buffered pairs → Master

Figure 1: Execution overview
Execution Overview

- **Workers** perform *Map* task
  - Intermediate key/value pairs buffered in memory
  - Periodically write buffered pairs into *local disk*
  - Locations of buffered pairs → Master
Execution Overview

- Master sends these locations to **reduce workers**
- Reduce worker reads **intermediate data**
  - Sorts data by intermediate keys →
  - Intermediates with same key group together
  - If intermediate data too large:
    - External sort is used
- Atomically renames temporary output file →
  - Final output file
- Final file system: Data from **one** execution of each reduce task
- Therefore, **R** output files

![Diagram of execution overview]

Figure 1: Execution overview
Execution Overview

- Master sends these locations to **reduce workers**
- Reduce worker reads **intermediate data**
  - Sorts data by intermediate keys →
  - Intermediates with same key group together
  - If intermediate data too large:
    - External sort is used
Execution Overview

- Master sends these locations to **reduce workers**
- Reduce worker reads **intermediate data**
  - Sorts by intermediate keys →
  - Intermediates with same key group together
  - If intermediate data too large:
    - External sort is used
Execution Overview

- Master sends these locations to **reduce workers**
- Reduce worker reads **intermediate data**
  - Sorts by intermediate keys →
  - Intermediates with same key group together
  - If intermediate data too large:
    - External sort is used

- Atomically renames temporary output file → Final output file
- Final file system: Data from **one** execution of each reduce task
- Therefore, $R$ output files
Execution Overview

- When all tasks completed
- Master wakes up the user program
Execution Overview

- Master data structures:
  - State of each map & reduce task:
    - Idle, in-progress, completed
  - Identity of worker machine
  - \( R \) intermediate file locations for each completed map task
Outline

- Motivation & Overview
- Word Count Example & Implementation
- Structure & Execution Overview
- Fault Tolerance
- Refinements
- Performance
Fault Tolerance

● Why important:
  ○ Commodity machines: Failures are common
  ○ Hundreds and thousands of machines: Tolerate faults gracefully

● Primary mechanism: Re-execution

● Worker Failure
● Master Failure
Fault Tolerance

Worker Failure

- Master pings worker periodically
- No response → Mark worker failed
- Reset Map task **completed** by failed worker → idle state
- Reset **in-progress** Map and Reduce tasks by failed worker → idle state
- Idle tasks → eligible for rescheduling
Fault Tolerance

Worker Failure
- Master pings worker periodically
- No response → Mark worker failed
- Reset Map task completed by failed worker → idle state
- Reset in-progress Map and Reduce tasks by failed worker → idle state
- Idle tasks → eligible for rescheduling
Fault Tolerance

Worker Failure

- Master pings worker periodically
- No response → Mark worker failed
- Reset Map task **completed** by failed worker → idle state
- Reset **in-progress** Map and Reduce tasks → idle state
- Idle tasks → eligible for rescheduling
Fault Tolerance

Worker Failure

- Master pings worker periodically
- No response $\rightarrow$ Mark worker failed
- Reset Map task completed by failed worker $\rightarrow$ idle state
- Reset in-progress Map and Reduce tasks $\rightarrow$ idle state
- Idle tasks $\rightarrow$ eligible for rescheduling
Fault Tolerance

Worker Failure

- Completed Map tasks are re-executed on failure
  - Map outputs: on local disks of workers

- Completed Reduce tasks do not need
  - Reduce outputs: in global file system

- All the workers will be notified of a re-execution
  - Reduce worker read data from new location

- Resilient to large-scale worker failures
Fault Tolerance

Worker Failure

● Completed Map tasks are re-executed on failure
  ○ Map outputs: on local disks of workers

● Completed Reduce tasks do not need
  ○ Reduce outputs: in global file system

● All the workers will be notified of a re-execution
  ○ Reduce worker read data from new location

● Resilient to large-scale worker failures
Fault Tolerance

Master Failure

- Single master, rare failure
- If master fails, aborts the MapReduce computation
- Client can retry

Figure 1: Execution overview
Semantics in the Presence of Failures

- **Deterministic Map/Reduce Functions**
  - Atomic commit task output:
    - guarantee no duplicates of *Map* results
  - Atomic rename operation:
    - guarantee no duplicates of *Reduce* results

- **Non-deterministic Functions**
  - Weaker but still reasonable semantics
Outline

- Motivation & Overview
- Word Count Example & Implementation
- Structure & Execution Overview
- Fault Tolerance
- Refinements
- Performance
Locality

- Scarce resource: Network bandwidth
- Solution: Store input data (managed by GFS) on local disks of machines that makes up the cluster
- GFS:
  - Divides data into 64 MB blocks
  - Replicates data in different machines (usually 3)
- Master: Assign map tasks to machines contains the data or close to data locations (e.g. same network switch)
Refinements

- Customizable Partitioning Function
- Ordering Guarantees
- Input and Output Types
- Auxiliary additional outputs
- Skipping bad records
- Local Execution
- Status information
- Counters
Refinements

- Optional Combiner Function
  - Partial merging of data at the end of Map
  - Typically same code as Reduce
  - E.x. <the, 1> in Zipf distributed word count task
Outline

- Motivation & Overview
- Word Count Example & Implementation
- Structure & Execution Overview
- Fault Tolerance
- Refinements
- Performance
Performance

2 computation tasks:

- Search through appx. 1 terabytes data → rare 3-character pattern (Grep)
  - Extract small amount of interesting data from large dataset
  - Input → 64 MB pieces ($M = 15000$)
  - Output in 1 file ($R = 1$)
Grep

- Peaks at ~ 30 GB/s
  - 1764 Workers

- ~ 1 min startup overhead
  - Program propagation to workers
  - Delays when interacting with GFS for locality optimization

Figure 2: Data transfer rate over time
Performance

2 computation tasks:

- Sort approx. 1 terabyte of data (Sort)
  - Shuffles data from one representation to another
  - Modeled after the TeraSort benchmark
  - Map → word, text line
  - Reduce → Built-in Identity function
Performance

2 computation tasks:

- Sort approx. 1 terabyte of data (Sort)
  - Input → 64 MB pieces ($M = 15000$)
  - Final output: A set of 2-way replicated GFS files
  - $R = 4000$
Sort Performance

- **Input rate:**
  - Input is read

- **Shuffle rate:**
  - Data sent from map tasks to reduce tasks

- **Output rate**
  - Sorted data written to final output files by reduce tasks

- **Higher input rate**
  - Locality optimization

(a) Normal execution
Sort Performance

- Input rate less than that for grep
  - Spend half of time & bandwidth writing intermediates
Sort Performance

First batch of ~ 1700 reduce workers

Some of the first batch finish, Start shuffling for remaining reduce tasks

- Shuffle rate:
  - Data sent from map tasks to reduce tasks

(a) Normal execution
Sort Performance

- Output rate
  - Sorted data written to final output files by reduce tasks

First batch of ~ 1700 reduce workers
- Some of the first batch finish,
  - Start shuffling for remaining reduce tasks
- Delay due to busy sorting of intermediates
  - Finishes at ~850s
  - (891s including the startup overhead)

(a) Normal execution
Backup Task

- “Stragglers”: machines take unusually long time

- Solution:
  - *Map/Reduce* close to completion
  - Master schedule backups for remaining in-progress tasks

- 44% longer time when no backup tasks

Wait for 5 “stragglers” from 960s
Machine Failure

- Killed 200 out of 1746 workers
  - \(\approx 11.5\%\) workers

- 5% increase of execution time

- Neg values: Map work need to be redone in dead workers
Sort Performance

- Entire computation takes 891s
- Comparable to best reported results (1057s) for the TeraSort Benchmark
Conclusion

● Mapreduce is easy to use
  ○ Hides details of
    ■ Parallelization
    ■ Fault-tolerance
    ■ Locality optimization
    ■ Load balancing

● Powerful
  ○ A large variety of problems are expressible as MapReduce computations

● Scalable
  ○ Implementation of MapReduce using large cluster of machines
Thank you!

Questions?