Mapreduce: Simplified Data Processing on Large Clusters

Speakers: Sach Vaidya, Liang Zhang

Authors: Jeffrey Dean, Sanjay Ghemawat
Outline

- Motivation/Intuition
- Methodology
- Execution Model
- Fault Tolerance/Advantages
- Refinements
- Performance/Experience
- Conclusion
Motivation

- Some computations are required to process large amounts of input data
  - Ex. Generating inverted index, Analyzing graph structure

- These computations must be distributed across multiple machines in order to finish in a reasonable amount of time

- Goal: Programming model that expresses this computation but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing
Intuition

- Break down computation into *map* and *reduce* stages
- **Map** stage applies some operation to each record in input and produce intermediate key/value pairs
- **Reduce** stage applies some operation to all values with the same key
- Enables computation to be distributed across many machines
Example: Word Count

Input:
Document 1: [“eecs”, “591”, “mapreduce”]
Document 2: [“mapreduce”, “map”, “reduce”]
Document 3: [“mapreduce”, “map”]
Document 4: [“mapreduce”, “reduce”]

Output:
(eecs, 1)
(591, 1)
(mapreduce, 4)
(map, 2)
(reduce, 2)

● One machine can go through all input at once
● Multiple machines can break up the work and aggregate results at the end
Example: Word Count

map(String key, String value):

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

Input:
Document 1 : ["eecs", "591", "mapreduce"]
Document 2 : ["mapreduce", "map", "reduce"]
Document 3 : ["mapreduce", "map"]
Document 4 : ["mapreduce", "reduce"]

Output:
(eecs, 1)     // Doc 1
(591, 1)
(mapreduce, 1)
(mapreduce, 1)    // Doc 2
(map, 1)
(reduce, 1)
(mapreduce, 1)    // Doc 3
(map, 1)
(mapreduce, 1)    // Doc 4
(reduce, 1)
Example: Word Count

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(key, AsString(result));

Input:
(eecs, 1) (591, 1) (mapreduce, 1)
(mapreduce, 1) (map, 1) (reduce, 1)
(mapreduce, 1) (map, 1) (mapreduce, 1)
(reduce, 1)

Output:
(eecs, 1)
(591, 1)
(mapreduce, 4)
(map, 2)
(reduce, 2)
Setup

- 1 Master, remaining machines are all Workers
- M: number of map tasks
- R: number of reduce tasks
Map Phase

- Input is split into M pieces
- Master assigns map task to idle workers
  - Will assign M total map tasks
- Mapper reads corresponding input split and runs user defined map function on that split
- Mapper output is written to local disk partition $\text{hash(key)} \% R$
- Output location is sent back to the master
Reduce Phase

- Master assigns reduce task to idle workers
  - Will assign R total reduce tasks
- Reducer reads all mapper output using RPC calls
- Reducer sorts all intermediate data by key
Reduce Phase

- Reducer iterates over sorted intermediate data
- Reducer runs user defined reduce function for each unique intermediate key and all the values associated with that key
- Reducer output is appended to output file at $hash(key) \% R$
Master

Assign map tasks

Split Input

Input 0 Input 1 Input 2 Input 3

Worker 0 Worker 1 Worker 2 Worker 3

Read

Word Count
(Map, Reduce, Input)
M=4, R=2

Client

Intermediate Output Files
$M = 4, R = 2$

Master

Map Done!

Worker 0  Worker 1  Worker 2  Worker 3

Input 0  Input 1  Input 2  Input 3

Intermediate Output Files

Client
M = 4, R = 2

Assign Reduce Tasks

Remote Read

Intermediate Output Files
$M = 4, R = 2$

Master

Back to user code

Client

Input 0  Input 1  Input 2  Input 3

Reduce Done!

Worker 0  Worker 1  Worker 2  Worker 3

Output 0  Output 1

Intermediate Output Files

P0  P1  P0  P1  P0  P1  P0  P1
Fault Tolerance - Worker Failure

- Master detects worker failure by heartbeats
- Any map task completed by failed worker is reset and rescheduled
  - Map output was stored on local disk
- Any map or reduce task in progress on the failed worker is reset and rescheduled
- All other reducers are notified of a failed map task
  - Must be updated on where they should read map output from
Fault Tolerance - Master Failure

- Not discussed in paper
- Have master periodically write periodic checkpoints of its data structures
- New master can resume execution using the log if needed
- Google chooses to abort execution with master failure
Advantages - Locality

- Most input data is stored on the local disk of the machines in the cluster
- Master attempts to schedule a map task on a machine that contains the corresponding input data locally
- If that is not possible, it attempts to schedule on a nearby machine instead
- Conserves network bandwidth by reading data locally
Advantages - Task Granularity

- Ideally $M$ and $R$ should be much larger than number of workers
- Improves dynamic load balancing, speeds up recovery when worker fails
- Master must schedule $O(M + R)$ tasks
- Master must maintain $O(M \times R)$ state information for each worker task
Task Granularity in Practice

- R is constrained by the user because it determines the number of output files.
- M is chosen so each task operates on roughly 16MB-64MB of data.
- Google has performed mapreduce computations with M = 200,000 and R = 5,000 using 2,000 worker machines.
Improvement - Backup Tasks

- Stragglers can lengthen the time needed for a mapreduce task
  - Machine taking an unusually long time to complete task
- Alleviated by scheduling backup executions of a task still in progress
  - Task is marked completed whenever either the primary or backup finishes execution
- Tradeoff between additional computation and overall time needed to execute.
- Backup execution improved sort execution by 44%
Refinements

1. Partitioning Function
2. Ordering Guarantees
3. Combiner Function
4. Input and Output Types
5. Side-effects
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. Counters
Partitioning Function

- As introduced in the former part, a default partitioning function is provided that uses hashing ("hash(key) % R") to result in fairly well-balanced partitions.

- In some cases, however, it is useful to partition data by some other function of the key. The user of the MapReduce library can provide other special partitioning function.
Partitioning Function

E.g.

The output keys are URLs, and we want all entries for a single host to end up in the same output file.

Solution:

Mapreduce support “hash(Hostname(urlkey)) % R” as the partitioning function.

This will cause all URLs from the same host to end up in the same output file.
Ordering Guarantees

- Guarantees that within a given partition, the intermediate key/value pairs are processed in increasing key order. This guarantee makes it easy to generate a sorted output file per partition.
Combiner Function

- The MapReduce library allows the user to specify an optional Combiner function that does partial merging of this data before it is sent over the network.
Compare Reduce Function and Combiner Function

- The only difference:
  - The output of a **reduce function** => the final output file

- The output of a **combiner function** => an intermediate file that will be sent to a reduce task
Input and Output Types

- The MapReduce library provides support for reading input data in several different formats.

- Besides reading from file, reader also can reads records from a database, or from data structures mapped in memory. Writer is similar.
Side-effects

- Users may find it convenient to produce auxiliary files as additional outputs.

- The mapreduce rely on the application writer to make such side-effects atomic and idempotent.

- Typically, the application writes to a temporary file and atomically rename this file once it has been fully generated.
Skipping Bad Records

- Bugs in user code => crash of the Map or Reduce functions on records

- The usual action is to fix the bug, but sometimes this is not feasible.

- Provide an optional mode of execution to make forward progress regardless the bug records.
Skipping Bad Records

How to detect bug/bad records:

● Each worker process installs a signal handler that catches segmentation violations and bus errors.

● Before invoking a Map or Reduce operation, stores the sequence number.

● The user code generates a signal => the signal handler sends a UDP packet that contains the sequence number to the MapReduce master.

● More than one failure been seen on a record indicates the record should be skipped.
Local execution

- Debugging problems in Map or Reduce functions can be tricky.

- Have developed an alternative implementation of the MapReduce library that sequentially executes all of the MapReduce work on local machine.

- Users invoke program with a special flag and can then easily use any debugging or testing tools.
Status Information

- Master runs an internal HTTP server and exports a set of status pages for human consumption.
- Status pages show the progress of the computation.
- This data can be used to predict how long the computation will take.
- The top-level status page shows which workers have failed and related information.
Counters

- The MapReduce library provides a counter facility to count occurrences of various events.
- Example to use counter:

```java
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
    for each word w in contents:
        if (IsCapitalized(w)):
            uppercase->Increment();
            EmitIntermediate(w, "1");
```
Performance

- Measures the performance on two computations running on a large cluster of machines.

- Measures the performance of MapReduce at:
  - Cluster Configuration
  - Grep
  - Sort
Evaluation of sort

Figure 3 (a) shows the progress of a normal execution of the sort program.

Figure 3 (b) shows an execution of the sort program with backup tasks disabled.

Figure 3 (c), shows an execution of the sort program where 200 out of 1746 worker processes were intentionally killed.
Experience

- MapReduce has been used across a wide range of domains within Google.

- Figure on the left shows the significant growth in the number of separate MapReduce programs over time at Google.

- MapReduce has been so successful because:
  - Makes it possible to write a simple program and run it efficiently on a thousand machines.
  - Allows programmers who have no experience with distributed or parallel systems to use it easily.
Conclusion

- The MapReduce programming model has been successfully used at Google for many different purposes.

- The success can be attributed to the following three reasons:
  - Easy to use
  - Variety of problems can be easily expressed as MapReduce computations
  - Scales to large clusters of machines comprising thousands of machines