MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean, Sanjay Ghemawat

Presented by: Kevin Ji

Introduction: Large Scale Data Processing

Many tasks: Process lots of data to produce other data
Example: 20+ billion webpages $\times$20KB $=$ 400+ terabytes

- one computer can read $30 - 35$ MB/sec from disk
- $\sim$ 4 months to read the web

Solution: Distribute the work over thousands of machines
Problem: Programming Complexity

- Communication and coordination
- Recovering from machine failure
- Status reporting
- Debugging
- Optimization
- Locality

Must solve these problems for every new problem

MapReduce

A simple programming model that enables automatic parallelization and distribution of large-scale computations

- Specific implementation of the interface for commodity computing clusters

Hide messy details in MapReduce runtime library:

- Automatic parallelization
- Fault-tolerance
- Load balancing
- Data distribution

Programming Model

Input: A set of *input* key/value pairs

Output: A set of *output* key/value pairs

Programmer expresses computation as *Map* and *Reduce* functions

map$(k, v) \rightarrow list(\langle k', v' \rangle)$

- Extract something you care about from each record
- Processes input key/value pair
- Produces set of *intermediate* key/value pairs

Shuffle and Sort

- MapReduce library groups intermediate values according to intermediate key and passes them to the *Reduce* function

reduce$(k', list(v')) \rightarrow list(v')$

- Aggregate, summarize, filter, or transform related values
- Combines all intermediate values for a particular key
- Produces a set of merged output values

Example: Counting Word Occurrences

Pseudocode:

```
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));
```

The `map` function emits each word with a count of 1

The `reduce` function sums together all counts for a particular word

More Examples

Distributed Grep:

- *Map*: Emits a line if it matches a supplied pattern
- *Reduce*: Identity function

Count of URL Access Frequency:

- *Map*: Processes logs of page requests and outputs `<URL, 1>`
- *Reduce*: Sums together all values for the same URL and emits `<URL, total_count>`

Reverse Web-Link Graph:

- *Map*: Outputs `<target, source>` pairs for each link to `target` from `source`
- *Reduce*: Concatenates the list of source URLs for a particular target URL and emits `<target, list(source)>`

More Examples

Term-Vector per Host:

- *Map*: Emits `<hostname, term_vector>` pair for each input document
- *Reduce*: Sums all term vectors for a given host, discards infrequent terms, and emits `<hostname, term_vector>`

Inverted Index:

- *Map*: Emits `<word, document_ID>` pairs
- *Reduce*: Sorts all document IDs and emits `<word, list(document_ID)>` pairs

Distributed Sort:

- *Map*: Emits `<key, record>` pair for each record
- *Reduce*: Emits all pairs unchanged

Implementation

Computing Clusters

- Thousands of computers connected by switched Ethernet
- Commodity networking hardware
  - 100 megabits/second or 1 gigabit/second at machine level
  - Averaging substantially less in overall bisection bandwidth
- Users submit jobs to a scheduling system

Machines

- 2 CPUs: typically hyperthreaded or dual-core
- Several locally-attached disks
- 4GB-16GB of RAM
- Typical machine runs:
  - Google File System (GFS)
  - Scheduler daemon for starting user tasks
  - User tasks

Execution Overview

One master, many workers

- Input data divided into M *splits* (typically 64 MB in size), each one corresponds to a map task
- Reduce phase partitioned into R reduce tasks
  - Partition function over intermediate key space
- Tasks are assigned to workers dynamically
- Often: $M = 200000$, $R = 4000$, workers=2000

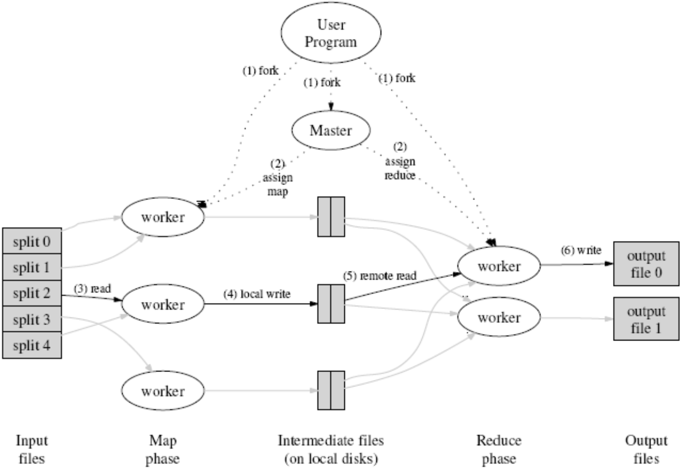
Master assigns each map task to a free worker

- Considers locality of data to worker when assigning task
- Worker reads the corresponding input (often from local disk)
- Worker produces R local files containing intermediate key/value pairs

Master assigns each reduce task to a free worker

- Worker reads intermediate key/value pairs from map workers
- Worker sorts and applies Reduce to produce the output

Execution Overview



Master Data Structures

For each map and reduce task

- State: idle, in-progress, completed
- Identity of the worker machine for non-idle tasks

Locations and sizes of the intermediate file regions

Fault Tolerance: Handled via Re-execution

On worker failure:

- Detect failure via periodic ping
- Re-execute completed and in-progress map tasks
- Re-execute in-progress reduce tasks
- Task completion committed through master

On Master failure:

- Abort computation
- Could have master write checkpoints to GFS and have new master recover from checkpoint

Semantics in the Presence of Failures

Deterministic *map* and *reduce* operators

- Equivalent output as sequential execution
- Rely on atomic commits of map and reduce task outputs
  - Private temporary output files
  - Map tasks notify master of these files on completion
  - Reduce tasks atomically rename temp to final output file
- Rely on atomic rename operation from GFS
  - Multiple instances of the same reduce task produce multiple rename calls for the same final output file.

Non-Deterministic *map* and *reduce* operators

- Output for a particular reduce task R_1 is equivalent to the output for R_1 produced by *some* sequential execution
- Output for another reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution

Locality

Problem: Network bandwidth is scarce
Solution: Master Scheduling Policy

- Asks GFS for locations of replicas of input file blocks
- Map tasks typically split into 64MB (GFS block size)
- Map tasks scheduled so GFS input block replica are on same machine or network switch

Effect: Most input data is read from local disk

Task Granularity

Fine granularity tasks: many more map tasks than machines

- Minimizes time for fault recovery
- Better dynamic load balancing

Practical bounds on M and R

- $O(M + R)$ scheduling decisions
- $O(M \cdot R)$ state in memory
- R separate output files

Typically choose M so each individual task is 16MB to 64MB of input data, and R a small multiple of the expected number of worker machines to be used

- Often use $M = 200000$, $R = 5000$, with 2000 machines

Backup Tasks

Problem: Slow workers significantly lengthen completion time

- Other jobs consuming resources on machine
- Bad disks with soft errors transfer data very slowly
- Weird things: processor caches disabled...

Solution: Near end of MapReduce operation, spawn backup copies of tasks

- Whichever one finishes first wins

Effect: Dramatically shortens job completion time

Refinements

Partitioning Function

- Allows reduce operations on different keys to be parallelized
- By default, a simple hash function modulo R
  - e.g. $hash(key) \mod R$
- Users may specify custom partition functions:
  - e.g. $hash(Hostname(urlkey)) \mod R$

Ordering Guarantees

- Within a given partition the intermediate key/value pairs are processed in increasing key order

Combiner Function

- Can run on the same machine as a mapper
  - Operates locally on the output of individual mapper
- Aggregates data before passing to reducer functions

Refinements

Input and Output Types

- Input type implementations know how to split themselves meaningfully for map tasks
- Output types for producing formatted data

Side-effects

- Programmer must themselves make side-effects (e.g. auxiliary files) atomic and idempotent

Refinements

Skipping Bad Records

- Map/Reduce functions sometimes fail for particular records
  - Not always possible to debug and fix
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for the same record:
    - Next worker is told to skip the record
- Effect: Can work around bugs in third-party libraries

Local Execution

- Alternative implementation of MapReduce library to sequentially execute all work locally.
- Controls to specify particular map tasks

Refinements

Status Information

- Exports status pages showing progress of computation
- Links to standard error and standard output files
- Worker failure information

Counters

- Named counter objects incremented in *map* and/or *reduce*
- Counter values from workers are propagated to master
  - Piggybacked on ping response
- Master aggregates counters from successful tasks and returns them to the user code (eliminates duplicate executions)

Performance

Tests run on cluster of 1800 machines:

- 4GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

- Grep: Scan 10^{10} 100-byte records to extract records matching a rare pattern (92K matching records)
- Sort: Sort 10^{10} 100-byte records (modeled after TeraSort benchmark)

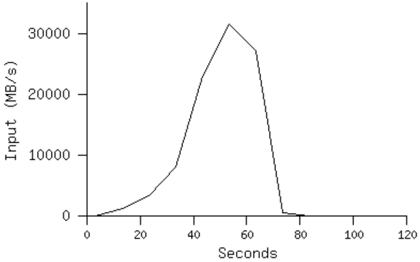
Grep

Large *map*, small *reduce*

- 64 MB splits, $M = 15000$
- Single output file, $R = 1$

Locality optimization helps:

- Peak transfer rate of 30+GB/s with 1764 workers assigned



Startup overhead is significant for short jobs

- 150 seconds total
- 1 minute startup overhead
    - Program propagation to worker machines
    - GFS interaction

Sort

Implementation:

- *Map*: Extracts 10-byte sort key from a line, emits key/line pair
- *Reduce*: Built-in identity function

Custom partitioning function:

- Uses the initial bytes of the sort key
- Built-in knowledge of the distribution of keys

Tuning parameters:

- 64MB splits, $M = 15000$, $R = 4000$
- Final output written to 2-way replicated GFS files
  - 2TB written as output

Sort

More detailed experiment:

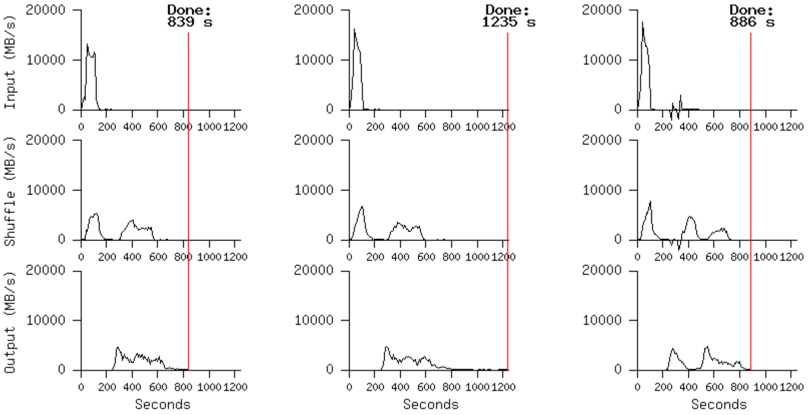
- Custom partitioning
- Removed backup tasks
- Induced machine failures

Results:

- Normal execution:
  - 891 seconds (850 excluding startup)
- No backup tasks:
  - 1283 seconds
- 200 induced failures:
  - 933 seconds

Sort

Normal | No backups | 200 processes killed



Sort

Normal

- Input rate is lower than Grep
  - Time writing intermediate output to local disk
- Delay between shuffling and output because of sorting
- Input rate is higher than shuffle and output rates
  - Locality optimization
- Shuffle rate is higher than output rate
  - 2-way replicated GFS files

No Backups

- Increased computation time
  - All but 5 tasks finish by 960 seconds
  - 5 stragglers add 300 seconds
  - 44% increase: 1283 seconds

Induced Failures

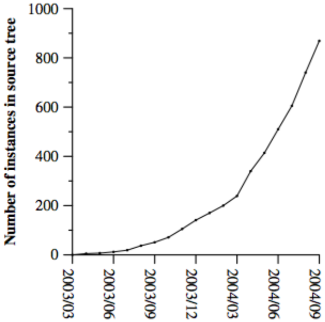
- Killed 200 of 1800 workers
- Re-execution begins immediately
- Only 5% total time increase

Experience

Rewrite of Production Indexing

- $5 - 10$ MapReduce operations for $20+$ terabytes of data
- New code is simpler, easier to understand
- MapReduce handles failures, slow machines
- Easy to improve performance by adding more machines

Broad applications



Related Work

- Programming model inspired by functional programming
- Partitioning/shuffling similar to many large-scale sorting systems
  - NOW-Sort [1]
- Re-execution for fault tolerance and locality-aware scheduling
  - BAD-FS [5] and TACC [7]
- Locality optimization has parallels with Active Disks
  - Active Disks [12, 15]
- Backup tasks similar to Eager Scheduling in Charlotte system
  - Charlotte [3]
- Dynamic load balancing solves similar problems as River's distributed queues
  - River [2]
- Cluster management system is similar to Condor
  - Condor [16]

Conclusions

- MapReduce has proven to be a useful abstraction
- Greatly simplifies large-scale computations
- Restricted programming model can make fault-tolerant parallelization easy
- Network bandwidth is scarce
- Redundant execution can help provide fault-tolerance