MapReduce: An Abstraction for Large-Scale Processing of Data

Jeffrey Dean and Sanjay Ghemawat

Google, Inc.*
May 24, 2004

Abstract

TODO

1 Introduction

For the past five years, we have been involved with developing and improving Google's core web search services. Over this time, we and many of our colleagues have implemented hundreds of special-purpose computations that process our raw data (files of records containing crawled documents, logs of web requests, etc.) to compute various kinds of derived data (e.g. inverted indices, various representations of the graph structure of web documents, summaries of the number for pages crawled per host, the set of most frequent queries in a given day, etc.). In the majority of these computations the actual processing for each input record is relatively straightforward. However, the fact that the input data is usually large (computations that process 20 TB of input data are not unusual) means that we want to distribute the computation across hundreds or thousands of machines in order to finish the computation quickly. At this point, the issues of how to parallelize the computation, data distribution, fault-tolerance, and load balancing conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As we gained experience and distaste for repeatedly writing complex code to perform conceptually straightforward computations, we started to search for a common abstraction that would allow us express the simple computations we were trying to perform, but would hide the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our inspiration is from the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately.

*The authors can be reached at the following addresses: {*jeff,sanjay*}*@google.com*.

The major contributions of this work are a simple-but-powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves very high performance in a cluster-based computing environment based on commodity PCs. Section 2 describes the basic MapReduce programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the basic MapReduce programming model that we have found useful. Section 5 has performance measurements of our MapReduce implementation for a variety of tasks. Section 6 discusses more about the general usage of MapReduce within Google including our experiences in using it as the basis for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map (written by the user) takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same key and passes them to the *Reduce* function. The *Reduce* function (written by the user) merges all of its arguments to produce a set of output pairs (typically just one output pair is produced per *Reduce* invocation).

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String name, String contents):
  for each word w in contents:
    EmitIntermediate(w, "1");
```

```
reduce(String word, list<String> values):
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  EmitFinal(word, AsString(result));
```

The `map` function emits each word plus an associated count of occurrences (just '1' in this simple example). The `reduce` function sums together all counts emitted for a particular word.

2.2 Usage

To use the library, the user implements the *Map* and *Reduce* functions as described earlier. In addition, the user links in the MapReduce library (implemented in C++) into their program, and invokes it as follows:

```
map_function(...) { ... }
reduce_function(...) { ... }

MapReduceSpecification spec;
MapReduceResult result;

add list of input files to spec
add output file name(s) to spec
specify map function to use
specify reduce function to use
set tuning parameters in spec

MapReduce(&spec, &result);
```

2.3 More Examples

Here are a few simple exampls of some interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of Query Frequency: The map function outputs $\langle query, 1\rangle$. The reduce function adds together all values for the same query and emits a $\langle query, totalcount\rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle targetURL, sourceURL\rangle$ pairs for each link to $targetURL$ found in a page named $sourceURL$. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle targetURL, list(sourceURL)\rangle$.

Term-Vector per Host: The map function emits a $\langle hostname, termvector\rangle$ pair for each input document. The reduce function is passed all per-document term vectors for a given host. It merges these term vectors together, throwing away infrequent terms, and then emits a final $\langle hostname, termvector\rangle$ pair.

Simple Inverted Index: The map function parses each document, and emits a sequence of $\langle word, documentID\rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle word, list(documentID)\rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice will depend on the environment. For example, one implementation may be suitable for a small shared memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

Below we describe an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched ethernet [**?**]. Some of the salient characteristics of this environment are:

(1) Machines are typically dual processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common events.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [**?**] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data into

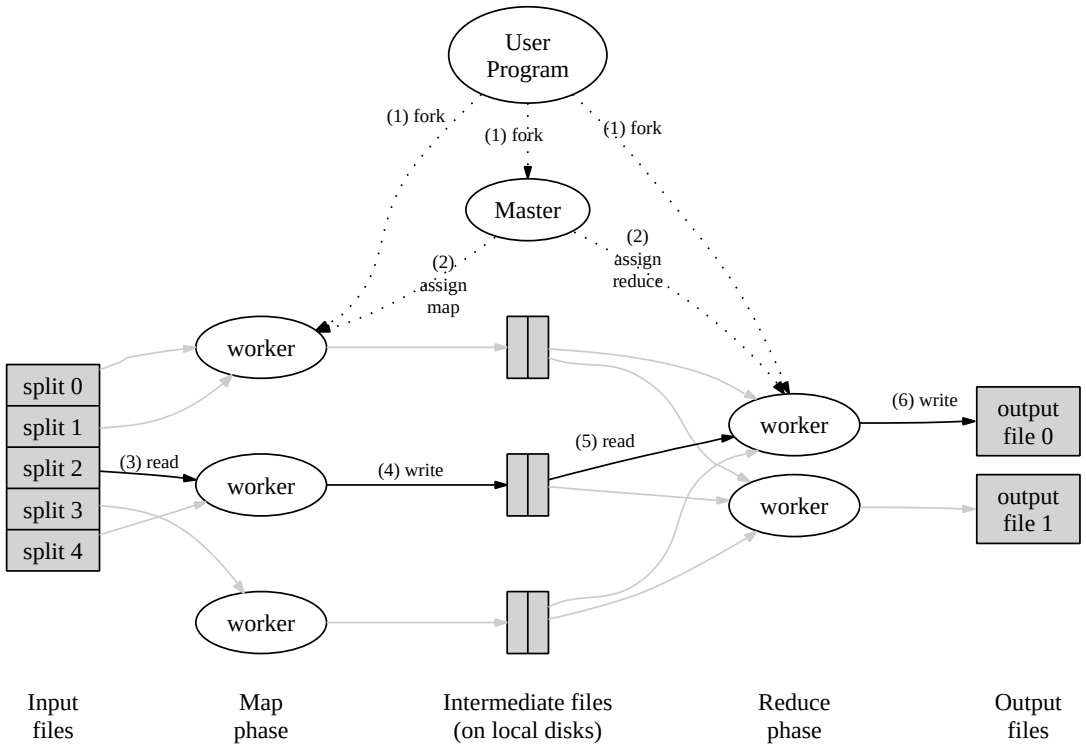


Figure 1: Execution overview

a set of M splits. Multple input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $hash(key)\%R$).

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the `MapReduce` function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16-64MB in size per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program is special – the master. The rest are workers who are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is written to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the *MapReduce()* call in the user program returns back to the user code.

3.2 Master Data Structures

The master keeps several data structures. For each map task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle map tasks).

For each reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle reduce tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed and are pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must gracefully tolerate machine failures.

Worker Failure

The master periodically pings every worker. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to periodically checkpoint the master data structures described above to persistent storage. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is very unlikely, and therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation, if they desire.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [**?**]) is actually stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores multiple replicas of each block on different machines (typically 3 replicas). The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g. on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails – the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large M and R can be in our implementation, since the master must make a total of $O(M+R)$ scheduling decisions and keeps $O(M*R)$ state in memory as described above. (The constant factors for memory usage are very small however – the $O(M*R)$ piece of the state keeps approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a"straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent, but correctable errors, that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling

system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: the MapReduce computation (and all other computation) on affected machines slowed down by over a factor of ten.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations (the final paper will include specific details, but our anecdotal evidence is that it improves completion time for our production indexing tasks by roughly XXX (20%?) XXX IS this right?).

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partition Function

The user of MapReduce specifies the number of reduce tasks/output files that they desired (R). Data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (e.g. $hash(key)\%R$). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, we have sometimes computed data whose keys were URLs of documents, but where we wanted all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide their own partitioning function (e.g. $hash(Hostname(urlkey))\%R$ for partitioning so that URLs from the same host end up in the same output file).

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee allows us to easily generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipfian distribution, each map task will produce hundreds or thousands of records of the form <''the'',''1''>. The *Reduce* function simply adds up the count values. To help conserve network bandwidth for MapReduce operations that satisfy these properties, we allow the user to specify a *Combiner* function that will get invoked with each unique intermediate key and a *partial* set of the intermediate values with this key. This is similar to a *Reduce* function, except that it gets executed on each machine that performs a map task, as a way of partially summarizing the intermediate key/value pairs (indeed, when using a Combiner function, the typical approach is to specify the same function for the Combiner and Reduce operations). We have found that this partial combining significantly speeds up certain classes of MapReduce operations.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text" mode input treats each line as a key/value pair – the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). User code can add support for a new input type by providing an implementation of simple *reader* interface.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to deterministically crash on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible (e.g. perhaps the bug is in a third-party library for which source code is unavailable, etc.). Also, sometimes it is acceptable to ignore a few records in a very large input set (e.g. when doing

statistical analysis on a large data set). We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and can skip such records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal, the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.6 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all the work for a MapReduce operation on the local machine (it can also be told to execute just particular map tasks). Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. `gdb`).

4.7 Status Page

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation (how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc.). The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. In addition, it can be useful to figure out when something is going much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.8 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```
Counter* uppercase;
uppercase = GetCounter(''uppercase'');

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

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable bound of the total number of documents processed.

5 Performance

In this section we show the performance of MapReduce by providing measurements of two large computations running on a large cluster of machines. One computation searches through approximately a TB of data looking for a particular pattern. The other computation sorts approximately a TB of data.

These two programs represent a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 available machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled; 4GB of memory; two 160GB

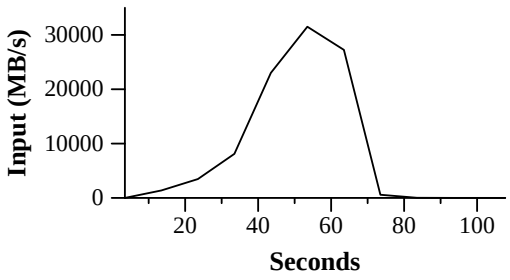


Figure 2: Data transfer rate over time

IDE disks; and a gigabit ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately XXX Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore round-trip times between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1 - 1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

5.2 Grep

The *grep* program scans through 10^{10} 100-bytes records, searching for a relatively rare pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish (this includes about a minute of startup overhead).

5.3 Sort

The *sort* program sorts 10^{10} 100-bytes records (approximately 1 TB of data).

The sorting program consists of less than 60 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the original text line as the intermediate key/value pair. We used a builtin *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair.

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Note: our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for grep had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine only executes one reduce task at a time). At around 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

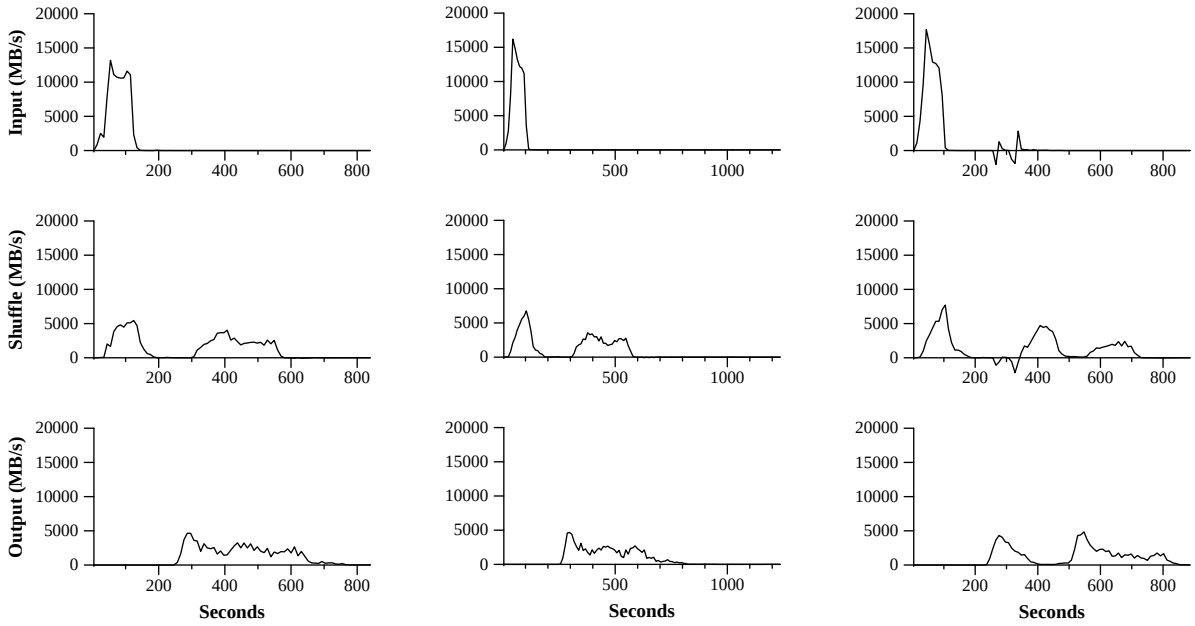
The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

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

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February, 2003, and made significant enhancements to it in August, 2003 (including the locality optimization, dynamic load balancing of task execution across worker machines, etc.). Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations

Figure 4 shows the significant growth in the number of separate MapReduce uses checked into our primary source code management system over time, from 0 in early 2003 to approximately 500 separate instances as of late May, 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to easily exploit large amounts of computational resources.

We have recently instrumented the MapReduce library to gather statistics about the computing resources and I/O utilization across all MapReduce computations, and will report on this data in the final paper.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production indexing system that produces the data structures used for the www.google.com web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 27 TB of uncompressed data (more than 8 TB afer compression). The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead

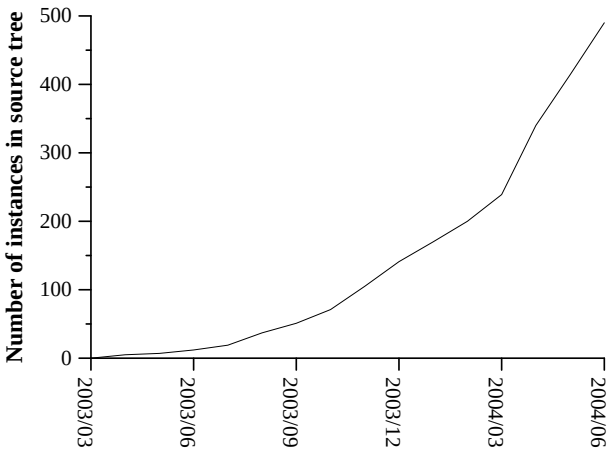


Figure 4: MapReduce instances over time

of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are automatically dealt with by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related and Future Work

```
. Direct-attached storage
  http://www.pdl.cmu.edu/Active/#Talks-Berkeley98
. TODO
```

8 Conclusions

MapReduce is an abstraction that successfully hides the details of parallelization, fault-tolerance, locality optimization and load balancing from programmers. MapReduce is broadly applicable to a wide range of computations we perform as part of running Google's production services and in developing new services. We have applied it to the problem of generating the data for Google's production web search service, data mining, machine learning, and many prototypes. We believe it to be applicable and useful across a broad range of data processing problems beyond those found in the information retrieval domain.

We have described an implementation of MapReduce suitable for execution on a large cluster of machines. The implementation is scalable, provides good performance, and handles machine and other failures gracefully.

9 Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce relies fairly heavily on both the Google File System and Global Work Queue systems to help with the parallelization, locality optimization, fault-tolerance and distribution of data. We would like to thank Howard Gobioff, Shun-tak Leung, Mohit Aron, David Kramer, Markus Gutschke, and Josh Redstone for their work in developing GFS, the large-scale distributed file system that serves as the backing store for MapReduce input and output data. Percy Liang (currently at MIT) developed an initial version of the Global Work Queue working with the first author during a summer internship at Google in 2002. Olcan Sercinoglu has implemented many enhancements to the Global Work Queue since 2002, and has also provided many helpful suggestions in the development of MapReduce. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback and suggestions (and bug reports :).

TODO: References

[**?**]

References