MapReduce: Simplified Data Processing on Large Clusters

by Jeffrey Dean and Sanjay Ghemawat

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks. Users specify the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Programmers find the system easy to use: more than ten thousand distinct MapReduce programs have been implemented internally at Google over the past four years, and an average of one hundred thousand MapReduce jobs are executed on Google's clusters every day, processing a total of more than twenty petabytes of data per day.

1 Introduction

Prior to our development of MapReduce, the authors and many others at Google implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, Web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of Web documents, summaries of the number of pages crawled per host, and the set of most frequent queries in a given day. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault tolerance, data distribution and load balancing in a library. Our abstraction is inspired by 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. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use reexecution as the primary mechanism for fault tolerance.

Biographies

Jeff Dean (jeff@google.com) is a Google Fellow and is currently working on a large variety of large-scale distributed systems at Google's Mountain View, CA, facility.

Sanjay Ghemawat (sanjay@google.com) is a Google Fellow and works on the distributed computing infrastructure used by most the company's products. He is based at Google's Mountain View, CA, facility.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs. The programming model can also be used to parallelize computations across multiple cores of the same machine.

Section 2 describes the basic programming model and gives several examples. In Section 3, we describe an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. In Section 6, we explore the use 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 intermediate key *I* and passes them to the reduce function.

The reduce function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges these values together to form a possibly smaller set of values. Typically just zero or one output value is produced per reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

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

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files and optional tuning parameters. The user then invokes the *MapReduce* function, passing it to the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Our original MapReduce paper contains the full program text for this example [8].

More than ten thousand distinct programs have been implemented using MapReduce at Google, including algorithms for large-scale graph processing, text processing, data mining, machine learning, statistical machine translation, and many other areas. More discussion of specific applications of MapReduce can be found elsewhere [8, 16, 7].

2.2 Types

Even though the previous pseudocode is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types.

map	(k1,v1)	→ list(k2,v2)
reduce	(k2,list(v2))	→ list(v2)

That is, the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

3. Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multiprocessor, and yet another for an even larger collection of networked machines. Since our original article, several open source implementations of MapReduce have been developed [1, 2], and the applicability of MapReduce to a variety of problem domains has been studied [7, 16].

This section describes our implementation of MapReduce that is targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Gigabit Ethernet [4]. In our environment, machines are typically dual-processor x86 processors running Linux, with 4-8GB of memory per machine. Individual machines typically have 1 gigabit/second of network bandwidth, but the overall bisection bandwidth available per machine is considerably less than 1 gigabit/second. A computing cluster contains many thousands of machines, and therefore machine failures are common. Storage is provided by inexpensive IDE disks attached directly to individual machines. GFS, a distributed file system developed in-house [10], 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.

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 a set of *M splits*. The 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*) **mod** *R*). The number of partitions (*R*) and the partitioning function are specified by the user.

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 following list).

1. The MapReduce library in the user program first splits the input files into *M* pieces of typically 16-64MB 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—the master— is special. The rest are workers that 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 for its partition, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. 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 appended to a final output file for this reduce partition.

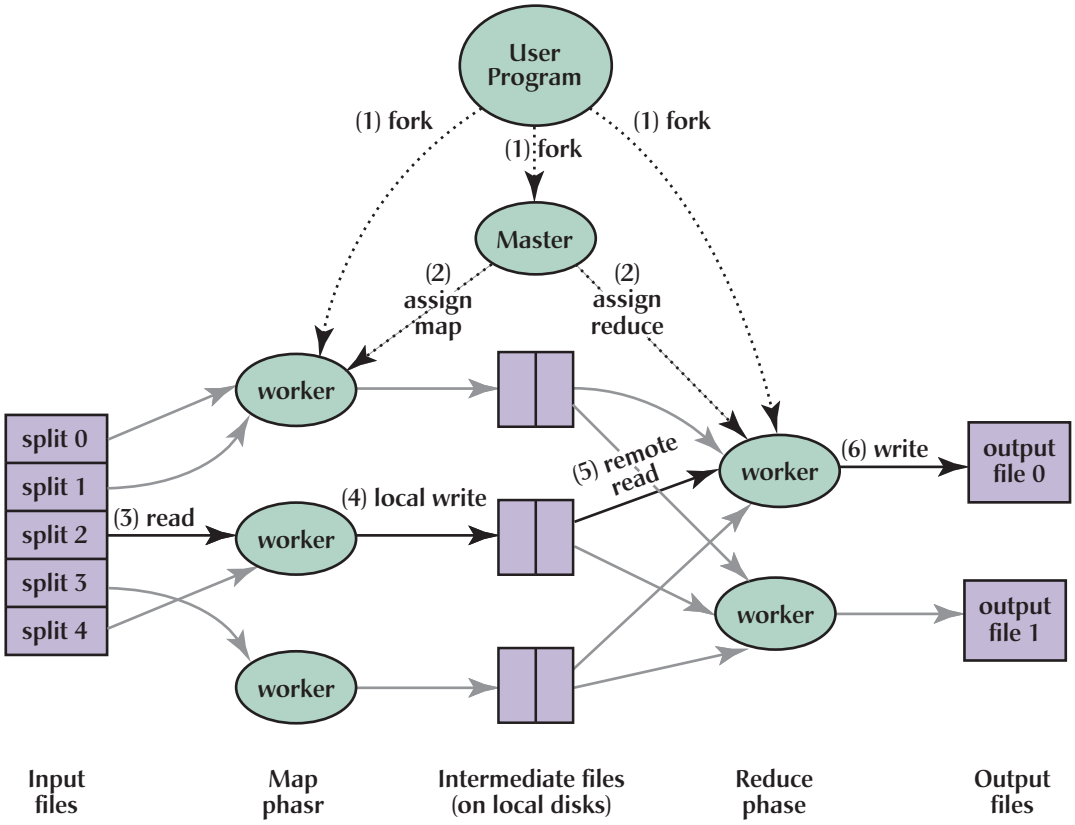

Fig. 1. Execution overview.

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.

After successful completion, the output of the mapreduce execution is available in the *R* output files (one per reduce task, with file names specified by the user). Typically, users do not need to combine these *R* output files into one file; they often pass these files as input to another MapReduce call or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*) and the identity of the worker machine (for nonidle 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. The information is 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 tolerate machine failures gracefully.

Handling Worker Failures

The master pings every worker periodically. 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.

Completed map tasks are reexecuted on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be reexecuted since their output is stored in a global file system.

When a map task is executed first by worker *A* and then later executed by worker *B* (because *A* failed), all workers executing reduce tasks are notified of the reexecution. Any reduce task that has not already read the data from worker *A* will read the data from worker *B*.

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 reexecuted the work done by the unreachable worker machines and continued to make forward progress, eventually completing the MapReduce operation.

Semantics in the Presence of Failures

When the user-supplied map and reduce operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a nonfaulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces R such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the R temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of R files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains only the data produced by one execution of the reduce task.

The vast majority of our map and reduce operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very easy for programmers to reason about their program's behavior. When the map and/or reduce operators are nondeterministic, we provide weaker but still reasonable semantics. In the presence of nondeterministic operators, the output of a particular reduce task R_1 is equivalent to the output for R_1 produced by a sequential execution of the nondeterministic program. However, the output for a different reduce task R_2 may correspond to the output for R_2 produced by a different sequential execution of the nondeterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_1 that committed (there is exactly one such execution). The weaker semantics arise because $e(R_1)$ may have read the output produced by one execution of M, and $e(R_2)$ may have read the output produced by a different execution of M.

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 [10]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64MB blocks and stores several copies of each block (typically 3 copies) on different machines. 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 previously. 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 $O(M+R)$ scheduling decisions and keep $O(M*R)$ state in memory as described. (The constant factors for memory usage are small, however. The $O(M*R)$ piece of the state consists of 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 16MB to 64MB of input data (so that the locality optimization described previously 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, that is, 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 correctable errors that slow its read performance from 30MB/s to 1MB/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: computations on affected machines slowed down by over a factor of one hundred.

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. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

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 include:

- user-specified partitioning functions for determining the mapping of intermediate key values to the R reduce shards;
- ordering guarantees: Our implementation guarantees that within each of the R reduce partitions, the intermediate key/value pairs are processed in increasing key order;
- user-specified combiner functions for doing partial combination of generated intermediate values with the same key within the same map task (to reduce the amount of intermediate data that must be transferred across the network);
- custom input and output types, for reading new input formats and producing new output formats;
- a mode for execution on a single machine for simplifying debugging and small-scale testing.

The original article has more detailed discussions of each of these items [8].

5 Performance

In this section, we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of 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 dataset.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time 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-byte records, searching for a relatively rare three-character 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).

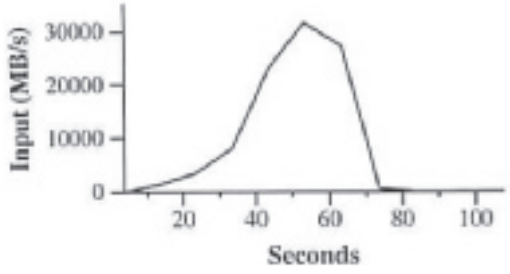

Fig. 2. Data transfer rate over time (mr-grep).

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. The overhead is due to the propagation of the program to all worker machines and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The sort program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [12].

The sorting program consists of less than 50 lines of user code. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

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 pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a prepass 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.

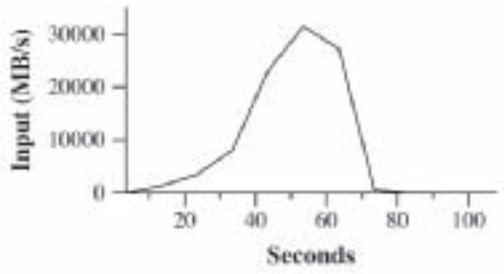

Figure 2: Data transfer rate over time (mr-grep)

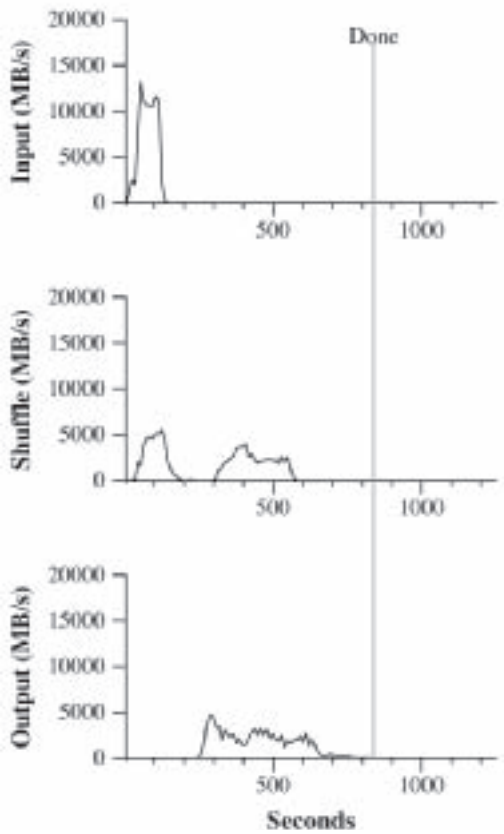

Fig. 3. Data transfer rate over time (mr-sort).

Figure 3 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 13GB/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.

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization; most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [15] rather than replication.

The original article has further experiments that examine the effects of backup tasks and machine failures [8].

6 Experience

We wrote the first version of the MapReduce library in February of 2003 and made significant enhancements to it in August of 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,
- extracting data to produce reports of popular queries (e.g. Google Zeitgeist and Google Trends),
- extracting 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),
- processing of satellite imagery data,
- language model processing for statistical machine translation, and
- large-scale graph computations.

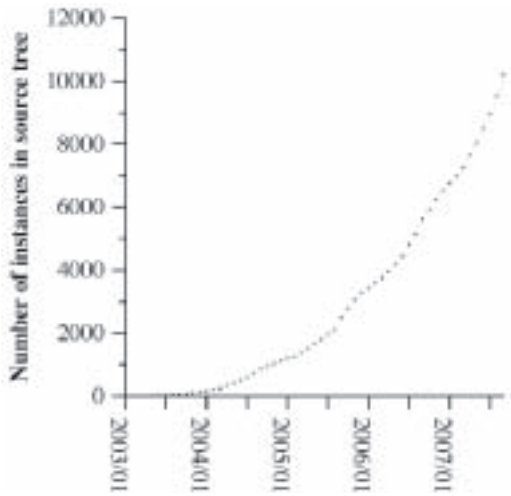

Fig. 4. MapReduce instances over time.

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source-code management system over time, from 0 in early 2003 to almost 900 in September 2004, to about 4000 in March 2006. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in a half hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

Table I. MapReduce Statistics for Different Months.

	Aug. '04	Mar. '06	Sep. '07
Number of jobs (1000s)	29	171	2,217
Avg. completion time (secs)	634	874	395
Machine years used	217	2,002	11,081
map input data (TB)	3,288	52,254	403,152
map output data (TB)	758	6,743	34,774
reduce output data (TB)	193	2,970	14,018
Avg. machines per job	157	268	394
Unique implementations			
map	395	1958	4083
reduce	269	1208	2418

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table I, we show some statistics for a subset of MapReduce jobs run at Google in various months, highlighting the extent to which MapReduce has grown and become the de facto choice for nearly all data processing needs at Google.

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 Google 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 20 terabytes of data. At the time we converted the indexing system to use MapReduce in 2003, it ran as a sequence of eight MapReduce operations. Since that time, because of the ease with which new phases can be added, many new phases have been added to the indexing system. Using MapReduce (instead 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 dealt with automatically 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 Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an *N* element array in log *N* time on *N* processors using parallel prefix computations [6, 11, 14]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Our locality optimization draws its inspiration from techniques such as active disks [13, 17], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network.

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [3]. Source machines (map workers) partition the data to be sorted and send it to one of *R* reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable map and reduce functions that make our library widely applicable.

BAD-FS [5] and TACC [9] are two other systems that rely on reexecution as a mechanism for implementing fault tolerance.

The original article has a more complete treatment of related work [8].

Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production Web search service, for sorting, data mining, machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

By restricting the programming model, we have made it easy to parallelize and distribute computations and to make such computations fault tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features. We would like to especially thank others who have worked on the system and all the users of MapReduce in Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

1. Hadoop: Open source implementation of MapReduce. http://lucene.apache.org/hadoop/.
2. The Phoenix system for MapReduce programming. http://csl.stanford.edu/~christos/sw/phoenix/.
3. Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Culler, D. E., Hellerstein, J. M., and Patterson, D. A. 1997. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. Tucson, AZ.
4 Barroso, L. A., Dean, J., and Urs Hölzle, U. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro 23*, 2, 22-28.
5. Bent, J., Thain, D., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Livny, M. 2004. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation* (*NSDI*).
6. Blelloch, G. E. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput. C-38*, 11.
7. Chu, C.-T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G., Ng, A., and Olukotun, K. 2006. Map-Reduce for machine learning on multicore. In *Proceedings of Neural Information Processing Systems Conference* (*NIPS*). Vancouver, Canada.
8. Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of Operating Systems Design and Implementation* (*OSDI*). San Francisco, CA. 137-150.
9. Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*. Saint-Malo, France. 78-91.
10. Ghemawat, S., Gobioff, H., and Leung, S.-T. 2003. The Google file system. In *19th Symposium on Operating Systems Principles*. Lake George, NY. 29-43.
11. Gorlatch, S. 1996. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds. *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science, vol. 1124. Springer-Verlag. 401-408
12. Gray, J. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.
13. Huston, L., Sukthankar, R., Wickremesinghe, R., Satyanarayanan, M., Ganger, G. R., Riedel, E., and Ailamaki, A. 2004. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*.
14. Ladner, R. E., and Fischer, M. J. 1980. Parallel prefix computation. *JACM 27*, 4. 831-838.
15. Rabin, M. O. 1989. Efficient dispersal of information for security, load balancing and fault tolerance. *JACM 36*, 2. 335-348.
16. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of 13th International Symposium on High-Performance Computer Architecture* (*HPCA*). Phoenix, AZ.
17. Riedel, E., Faloutsos, C., Gibson, G. A., and Nagle, D. Active disks for large-scale data processing. *IEEE Computer*. 68-74.