This chapter describes the design and implementation of MapReduce, a programming system for large-scale data processing problems. MapReduce was developed as a way of simplifying the development of large-scale computations at Google. MapReduce programs are automatically parallelized and executed on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing the required intermachine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

A Motivating Example

Suppose that you have 20 billion documents, and you want to generate a count of how often each unique word occurs in the documents. With an average document size of 20 KB, just reading through the 400 terabytes of data on one machine will take roughly four months. Assuming we were willing to wait that long and that we had a machine with sufficient memory, the code would be relatively simple. Example 23-1 (all the examples in this chapter are pseudocode) shows a possible algorithm.
One way of speeding up this computation is to perform the same computation in parallel across each individual document, as shown in Example 23-2.

**Example 23-1. Naive, nonparallel word count program**

```cpp
map<string, int> word_count;
for each document d {
    for each word w in d {
        word_count[w]++;
    }
}
... save word_count to persistent storage ...
```

**Example 23-2. Parallelized word count program**

```cpp
Mutex lock; // Protects word_count
map<string, int> word_count;
for each document d in parallel {
    for each word w in d {
        lock.Lock();
        word_count[w]++;
        lock.Unlock();
    }
}
... save word_count to persistent storage ...
```

The preceding code nicely parallelizes the input side of the problem. In reality, the code to start up threads would be a bit more complex, since we’ve hidden a bunch of details by using pseudocode. One problem with Example 23-2 is that it uses a single global data structure for keeping track of the generated counts. As a result, there is likely to be significant lock contention with the `word_count` data structure as the bottleneck. This problem can be fixed by partitioning the `word_count` data structure into a number of buckets with a separate lock per bucket, as shown in Example 23-3.

**Example 23-3. Parallelized word count program with partitioned storage**

```cpp
struct CountTable {
    Mutex lock;
    map<string, int> word_count;
};
const int kNumBuckets = 256;
CountTable tables[kNumBuckets];
for each document d in parallel {
    for each word w in d {
        int bucket = hash(w) % kNumBuckets;
        tables[bucket].lock.Lock();
        tables[bucket].word_count[w]++;
        tables[bucket].lock.Unlock();
    }
}
for (int b = 0; b < kNumBuckets; b++) {
    ... save tables[b].word_count to persistent storage ...
}
```
The program is still quite simple. However, it cannot scale beyond the number of processors in a single machine. Most affordable machines have eight or fewer processors, so even with perfect scaling, this approach will still require multiple weeks of processing to complete. Furthermore, we have been glossing over the problem of where the input data is stored and how fast it can be read by one machine.

Further scaling requires that we distribute the data and the computation across multiple machines. For the moment, let’s assume that the machines do not fail. One way to increase scaling is to start many processes on a cluster of networked machines. We will have many input processes, each one responsible for reading and processing a subset of the documents. We will also have many output processes, each responsible for managing one of the word_count buckets. Example 23-4 shows the algorithm.

**Example 23-4. Parallelized word count program with partitioned processors**

```c++
const int M = 1000;   // Number of input processes
const int R = 256;    // Number of output processes

main() {  
  // Compute the number of documents to assign to each process
  const int D = number of documents / M;
  for (int i = 0; i < M; i++) {
    fork InputProcess(i * D, (i + 1) * D);
  }
  for (int i = 0; i < R; i++) {
    fork OutputProcess(i);
  }
  ... wait for all processes to finish ...
}

void InputProcess(int start_doc, int end_doc) {
  map<string, int> word_count[R];     // Separate table per output process
  for each doc d in range [start_doc .. end_doc-1] do {
    for each word w in d {
      int b = hash(w) % R;
      word_count[b][w]++;
    }
  }
  for (int b = 0; b < R; b++) {
    string s = EncodeTable(word_count[b]);
    ... send s to output process b ...
  }
}

void OutputProcess(int bucket) {
  map<string, int> word_count;
  for each input process p {
    string s = ... read message from p ...
    map<string, int> partial = DecodeTable(s);
    for each <word, count> in partial do {
      word_count[word] += count;
    }
  }
  ... save word_count to persistent storage ...
}
```
This approach scales nicely on a network of workstations, but is significantly more complicated and hard to understand (even though we’ve hidden the details of marshaling and unmarshaling, as well as starting and synchronizing different processes). It also does not deal gracefully with machine failures. To deal with failures, we would extend Example 23-4 to re-execute processes that failed before completion. To avoid double-counting data when we re-execute an input process, we would mark each piece of intermediate data with a generation number of the input process and modify the output processing so that it uses these generation numbers to discard duplicates. As you can imagine, adding this failure-handling support would further complicate things.

The MapReduce Programming Model

If you compare Example 23-1 with Example 23-4, you’ll find that the simple task of counting words has been buried under lots of details about managing parallelism. If we can somehow separate the details of the original problem from the details of parallelization, we may be able to produce a general parallelization library or system that can be applied not just to this word-counting problem, but other large-scale processing problems. The parallelization pattern that we are using is:

- For each input record, extract a set of key/value pairs that we care about from each record.
- For each extracted key/value pair, combine it with other values that share the same key (perhaps filtering, aggregating, or transforming values in the process).

Let’s rewrite our program to implement the application-specific logic of counting word frequencies for each document and summing these counts across documents in two functions that we’ll call Map and Reduce. The result is Example 23-5.

**EXAMPLE 23-5. Division of word counting problem into Map and Reduce**

```java
void Map(string document) {
    for each word w in document {
        EmitIntermediate(w, "1");
    }
}

void Reduce(string word, list<string> values) {
    int count = 0;
    for each v in values {
        count += StringToInt(v);
    }
    Emit(word, IntToString(count));
}
```

A simple driver program that uses these routines to accomplish the desired task on a single machine would look like Example 23-6.
The Map function is called once for each input record. Any intermediate key/value pairs emitted by the Map function are collected together by the driver code. Then, the Reduce function is called for each unique intermediate key, together with a list of intermediate values associated with that key.

We’re now back to an implementation that runs on a single machine. However, with things separated in this manner, we can now change the implementation of the driver program to make it deal with distribution, automatic parallelization, and fault tolerance without affecting the application-specific logic in the Map and Reduce functions. Furthermore, the driver is independent of the particular application logic implemented by the Map and Reduce functions, and therefore the same driver program can be reused with other Map and Reduce functions to solve different problems. Finally, notice that the Map and Reduce functions that implement the application-specific logic are nearly as understandable as the simple sequential code shown in Example 23-1.

Other MapReduce Examples

We’ll examine the implementation of a much more sophisticated driver program that automatically runs MapReduce programs on large-scale clusters of machines in a moment, but first, let’s consider a few other problems and how they can be solved using MapReduce:

Distributed grep

The Map function emits a line if it matches a supplied regular expression pattern. The Reduce function is an identity function that just copies the supplied intermediate data to the output.
Reverse web-link graph
A forward web-link graph is a graph that has an edge from node URL1 to node URL2 if the web page found at URL1 has a hyperlink to URL2. A reverse web-link graph is the same graph with the edges reversed. MapReduce can easily be used to construct a reverse web-link graph. The Map function outputs \(<\text{target}, \text{source}>\) pairs for each link to a target URL found in a document named \text{source}. The Reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair \(<\text{target}, \text{list of source URLs}>\).

Term vector per host
A term vector summarizes the most important words that occur in a document or a set of documents as a list of \(<\text{word}, \text{frequency}>\) pairs. The Map function emits a \(<\text{hostname}, \text{term vector}>\) pair for each input document (where the hostname is extracted from the URL of the document). The Reduce function is passed all per-document term vectors for a given host. It adds these term vectors, throwing away infrequent terms, and then emits a final \(<\text{hostname}, \text{term vector}>\) pair.

Inverted index
An inverted index is a data structure that maps from each unique word to a list of documents that contain the word (where the documents are typically identified with a numeric identifier to keep the inverted index data relatively compact). The Map function parses each document and emits a sequence of \(<\text{word}, \text{docid}>\) pairs. The Reduce function accepts all docids for a given word, sorts the corresponding document IDs, and emits a \(<\text{word}, \text{list of docids}>\) 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 within each document.

Distributed sort
MapReduce can also be used to sort data by a particular key. The Map function extracts the key from each record, and emits a \(<\text{key}, \text{record}>\) pair. The Reduce function emits all pairs unchanged (i.e., the identity Reduce function). This computation depends on the partitioning facilities and ordering properties described later in this chapter.

There are many more examples of computations that can easily be expressed as a MapReduce computation. For more complex computations, it is often easy to express them as a sequence of MapReduce steps or as an iterative application of a MapReduce computation, where the output of one MapReduce step is the input to the next MapReduce step.

One you start thinking of data processing problems in terms of MapReduce, they are often relatively easy to express. As some testament to this, over the last four years, the number of MapReduce programs at Google has gone from a small handful of candidate problems in March 2003 (when we started to design MapReduce) to more than 6,000 distinct MapReduce programs in December 2006. These programs were written by more than a thousand different software developers, many of whom had never written a parallel or distributed program before using MapReduce.
A Distributed MapReduce Implementation

Much of the benefit of the MapReduce programming model is that it nicely separates the expression of the desired computation from the underlying details of parallelization, failure handling, etc. Indeed, different implementations of the MapReduce programming model are possible for different kinds of computing platforms. 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.

A very simple single-machine implementation that supports the programming model was shown in the code fragment in Example 23-6. This section describes a more complex implementation that is targeted to running large-scale MapReduce jobs on the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet (see “Further Reading,” at the end of this chapter). In this environment:

- Machines are typically dual-processor x86 processors running Linux, with 2–4 GB of memory per machine.
- Machines are connected using commodity-networking hardware (typically 1 gigabit/second switched Ethernet). Machines are organized into racks of 40 or 80 machines. These racks are connected to a central switch for the whole cluster. The bandwidth available when talking to other machines in the same rack is 1 gigabit/second per machine, while the per-machine bandwidth available at the central switch is much smaller (usually 50 to 100 megabits/second per machine).
- Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed filesystem called GFS (see the reference to “The Google File System” under “Further Reading,” at the end of this chapter) is used to manage the data stored on these disks. GFS uses replication to provide availability and reliability on top of unreliable hardware by breaking files into chunks of 64 megabytes and storing (typically) 3 copies of each chunk on different machines.
- 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.

**Execution Overview**

The Map function 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., $\text{hash(key)} \% R$).

Figure 23-1 shows the actions that occur when the user program calls the MapReduce function (the numbered labels in Figure 23-1 correspond to the numbers in the following list).
1. The MapReduce library first splits the input files into $M$ pieces (typically 16 megabytes to 64 megabytes per piece). It then starts up many copies of the program on a cluster of machines, by making a request to the cluster scheduling system.

2. One of the copies is special and is called the MapReduce master. The remaining tasks are assigned chunks of Map and Reduce work by the master. There are $M$ map tasks and $R$ reduce tasks. The master picks idle workers and assigns a map and/or a reduce task to each.

3. A worker that is assigned a map task reads the contents of the corresponding input split. It passes each input record 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 $R$ separate buckets by the partitioning function. When the map task is completed, the worker notifies the master. The master forwards information about the location of the intermediate data generated by this map task to any workers that have been assigned reduce tasks. If there are remaining map tasks, the master assigns one of the remaining tasks to the newly idle worker.

5. When a reduce worker is told the locations of intermediate data for its reduce task, it issues remote procedure calls to read the buffered intermediate data from the local disk of the map workers. When a reduce worker has finished reading all intermediate data for its reduce task, it sorts it by the intermediate keys so that all occurrences of the same intermediate key are grouped together. If the intermediate data is too large to fit in memory on the reduce worker, an external sort is used.

*Figure 23.1. Relationships between processes in MapReduce*
6. The reduce worker iterates over the sorted intermediate key/value pairs. For each unique intermediate key encountered, it passes the key and the corresponding list of intermediate values to the user’s Reduce function. Any key/value pairs generated by the user’s Reduce function are appended to a final output file for this reduce partition. When the reduce task is done, the worker notifies the master. If there are remaining reduce tasks, the master assigns one of the remaining reduce tasks to the newly idle worker.

When all map tasks and reduce tasks have been completed, the MapReduce function call in the user program returns, giving control back to the user code. At this point, the output of the MapReduce job is available in the \( R \) output files (one file per reduce task).

Several details of the implementation allow it to perform well in our environment.

Load balancing
A MapReduce job typically has many more tasks than machines, which means that each worker will be assigned many different tasks by the master. The master assigns a new task to a machine when it finishes its previous task. This means that a faster machine will be assigned more tasks than a slower machine. Therefore, the assignment of tasks to machine is properly balanced even in a heterogeneous environment, and workers tend to be kept busy with useful work throughout the computation.

Fault tolerance
Because this implementation of MapReduce is designed to run jobs distributed across hundreds or thousands of machines, the library must transparently handle machine failures.

The master keeps state about which map and reduce tasks have been done by which workers. The master periodically sends a ping remote procedure call to each worker. If a worker does not respond to several consecutive pings, the master declares that worker as dead and assigns any work that was done by that worker to other machines for re-execution. Since a typical MapReduce execution might have 50 times as many map tasks as worker machines, recovery is very fast, because 50 separate machines can each pick up one map task for re-execution when a machine fails.

The master logs all updates of its scheduling state to a persistent logfile. If the master dies (a rare occurrence, since there is only one master), it is restarted by the cluster scheduling system. The new master instance reads the logfile to reconstruct its internal state.

Locality
Our MapReduce implementation conserves network bandwidth by taking advantage of the fact that the input data (managed by GFS) is stored on the same machines or racks on which the map computation is executed. For any given Map task, the MapReduce master finds the locations of the input data (there are typically multiple locations due to GFS’s replication). The master then tries to schedule the map task on a machine that is close to one of the replicas of the tasks’s input data. For large MapReduce jobs that use thousands of workers, most input data is read directly from local disk.
Backup tasks

The running time of MapReduce is often dominated by a few stragglers. (A straggler is any machine that takes a long time to execute one of the last few map or reduce tasks.) A task may take a long time to execute either because it is intrinsically expensive, or because it is running on a slow machine.

A machine might be slow for a wide variety of reasons. For example, the machine might be busy with other unrelated CPU-intensive processes, or the machine might have a faulty hard drive that causes frequent retries of read operations that slow disk reads by factors of 10 or 100.

We use backup tasks to solve the problem of stragglers. When there are only a few map tasks left, the master schedules (on idle workers) one backup execution for each of the remaining in-progress map tasks. Each remaining map task is marked as completed whenever one of the instances of the task finishes (the primary or the backup). A similar strategy is used for reduce tasks. We typically use just 1–2 percent additional computational resources for backup tasks, but have found that they significantly shorten the typical completion time of large MapReduce operations.

Extensions to the Model

Although most uses of MapReduce require just writing Map and Reduce functions, we have extended the basic model with a few features that we have found useful in practice.

Partitioning function

MapReduce users specify the number of reduce tasks/output files that they desire (R). Intermediate data gets partitioned across these tasks using a partitioning function on the intermediate key. A default partitioning function is provided that uses hashing (hash(key) % R) to evenly balance the data across the R partitions.

In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the users of the MapReduce library can provide their own custom partitioning function. For example, using hash(Hostname(urlkey)) % R as the partitioning function causes all URLs from the same host to end up in the same output file.

Ordering guarantees

Our MapReduce implementation sorts the intermediate data to group together all intermediate values that share the same intermediate key. Since many users find it convenient to have their Reduce function called on keys in sorted order, and we have already done all of the necessary work, we expose this to users by guaranteeing this ordering property in the interface to the MapReduce library.

Skipping bad records

Sometimes there are bugs in user code that cause the Map or Reduce functions to crash deterministically on certain records. Such bugs may cause a large MapReduce execution to fail after doing large amounts of computation. The preferred course of action is
to fix the bug, but sometimes this is not feasible; for instance, the bug may be in a
third-party library for which source code is not available. Also, it is sometimes accept-
able to ignore a few records, such as when doing statistical analysis on a large data set.
Thus, we provide an optional mode of execution where the MapReduce library detects
records that cause deterministic crashes and skips these records in subsequent re-
executions, 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 record 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.

A number of other extensions are discussed in a lengthier paper about MapReduce (see
“Further Reading,” below).

Conclusion

MapReduce has proven to be a valuable tool at Google. As of early 2007, we have more
than 6,000 distinct programs written using the MapReduce programming model, and run
more than 35,000 MapReduce jobs per day, processing about 8 petabytes of input data per
day (a sustained rate of about 100 gigabytes per second). Although we originally devel-
oped the MapReduce programming model as part of our efforts to rewrite the indexing
system for our web search product, it has shown itself to be useful across a very broad
range of problems, including machine learning, statistical machine translation, log analy-
sis, information retrieval experimentation, and general large-scale data processing and
computation tasks.

Further Reading

A more detailed description of MapReduce appeared in the OSDI ’04 conference:

“MapReduce: Simplified Data Processing on Large Clusters.” Jeffrey Dean and Sanjay
Ghemawat. Appeared in OSDI ’04: Sixth Symposium on Operating System Design and
com/papers/mapreduce.html.

A paper about the design and implementation of the Google File System appeared in the
SOSP ’03 conference:

A paper describing the general hardware infrastructure at Google appeared in IEEE Micro:


A language called Sawzall developed at Google for logs analysis runs on top of MapReduce:


**Acknowledgments**

A number of people have made substantial contributions to the continued development and improvement of MapReduce, including Tom Annau, Matt Austern, Chris Colohan, Frank Dabek, Walt Drummond, Xianping Ge, Victoria Gilbert, Shan Lei, Josh Levenberg, Nahush Mahajan, Greg Malewicz, Russell Power, Will Robinson, Ioannis Tsoukalidis, and Jerry Zhao. MapReduce builds on a number of pieces of infrastructure developed at Google, including the Google File System and our cluster scheduling system. We would like to especially thank the developers of those systems. Finally, we thank all the users of MapReduce within Google’s engineering organization for providing helpful feedback, suggestions, and bug reports.

**Appendix: Word Count Solution**

This section contains the full C++ implementation of the word frequency counting example that was used in the early part of this chapter. The code can also be found on the O’Reilly web site for this book ([http://www.oreilly.com/catalog/9780596510046](http://www.oreilly.com/catalog/9780596510046)):

```cpp
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
public:
  virtual void Map(const MapInput& input) {
    const string& text = input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while ((i < n) && isspace(text[i]))
        i++;

      // Find word end
      int start = i;
      while ((i < n) && !isspace(text[i]))
        i++;
```

```cpp```
if (start < i)
    EmitIntermediate(text.substr(start,i-start),"1");
}
);
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // Iterate over all entries with the
        // same key and add the values
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};
REGISTER_REDUCTER(Adder);

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);

    MapReduceSpecification spec;

    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }

    // Specify the output files:
    //   /gfs/test/freq-00000-of-00100
    //   /gfs/test/freq-00001-of-00100
    //   ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Optional: do partial sums within map
    // tasks to save network bandwidth
    out->set_combiner_class("Adder");

    // Tuning parameters: use at most 2,000
    // machines and 100 MB of memory per task
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);
// Now run it
MapReduceResult result;
if (!MapReduce(spec, &result)) abort();

// Done: 'result' structure contains info
// about counters, time taken, number of
// machines used, etc.

return 0;
}