MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time 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 inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have 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, the set of most frequent queries in a given day, etc. 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 re-execution as the primary mechanism for fault tolerance.

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.

Section 2 describes the basic 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 programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

MapReduce：简化fi大型集群上的数据处理

杰弗里·迪安和桑杰伊·盖玛沃特

jeff@google.com, sanjay@google.com

谷歌公司

摘要

MapReduce是一种编程模型及其相关实现，用于处理和生成大型数据集。用户指定一个map函数，该函数处理键值对以生成一组中间键值对，以及一个reduce函数，该函数合并所有与相同中间键关联的中间值。许多现实世界的任务都可以用这种模型表达，如论文所示。用这种函数式风格编写的程序会自动并行化并在大型集群的通用机器上执行。运行时系统负责处理输入数据的分区、跨多台机器调度程序执行、处理机器故障以及管理所需的跨机通信。这使得没有并行和分布式系统经验的程序员能够轻松利用大型分布式系统的资源。我们的MapReduce实现运行在大型通用机器集群上，并且具有高度可扩展性：典型的MapReduce计算会在数千台机器上处理许多TB数据。程序员fi和系统易于使用：已有数百个MapReduce程序实现，并且每天谷歌集群上会执行超过一千个MapReduce作业。

1 引言

在过去的 fi 年里，作者和谷歌的许多其他人已经实现了数百种专门用途的计算，这些计算处理大量原始数据，例如爬取的文档、网络请求日志等，以计算各种派生数据，例如倒排索引、网络文档的图结构的各种表示形式、每个主机爬取的页面数量的摘要、某一天中最频繁查询的集合等。大多数此类计算在概念上是直接的。然而，输入数据通常很大，并且必须将计算分布在数百或数千台机器上，才能在合理的时间内完成。如何并行化计算、分发数据以及处理故障等问题相互交织，使得原本简单的计算被大量复杂的代码所掩盖，以处理这些问题。

为应对这种复杂性，我们设计了一种新的抽象机制，使我们能够表达我们试图执行的简单计算，同时将并行化、容错、数据分发和负载均衡等杂乱的细节隐藏在库中。我们的抽象机制受限于Lisp和许多其他函数式语言中存在的map和reduce原语。我们意识到，我们的大部分计算都涉及对输入中的每个逻辑 "记录" 应用map操作，以计算一组中间键值对，然后对所有共享相同键的值应用reduce操作，以适当组合衍生数据。我们使用具有用户自定义map和reduce操作的函数式模型，使我们能够轻松地并行化大型计算，并将重执行作为容错的主要机制。

这项工作的主要贡献是一个简单而强大的接口，它支持大规模计算的自动并行化和分发，以及实现该接口的实现，该实现能够在大型商用PC集群上实现高性能。

第二节描述了基本的编程模型并给出了几个示例。第三节描述了一个针对我们集群式计算环境的MapReduce接口的实现。第四节描述了我们发现的几个编程模型的改进。第五节展示了我们针对各种任务实现的性能指标。第六节探讨了MapReduce在谷歌内的使用情况，包括我们将其作为基础的使用经验

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 together these values 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 pseudo-code:

```
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 the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code 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)
```

I.e., 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.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of 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 URL Access Frequency: The map function processes logs of web page requests and outputs $\langle$`URL, 1`$\rangle$. The reduce function adds together all values for the same URL and emits a $\langle$`URL, total count`$\rangle$ pair.

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

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 $\langle word, frequency \rangle$ pairs. The map function emits a $\langle$`hostname, term vector`$\rangle$ 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 together, throwing away infrequent terms, and then emits a final $\langle$`hostname, term vector`$\rangle$ pair.

为了重写我们的生产索引系统。第7节讨论了相关和未来工作。

2 编程模型

计算接收一组输入键值对，并产生一组输出键值对。MapReduce 库的使用者将计算表示为两个函数：Map 和 Reduce。

Map 由用户编写，接收一个输入对并产生一组中间键值对。MapReduce 库将所有与相同中间键 I 相关的中间值组合在一起，并将它们传递给 Reduce 函数。

Reduce 函数也由用户编写，接收一个中间键 I 和该键的一组值。它将这些值合并在一起，形成一组可能更小的值。通常每个 Reduce 调用只产生零个或一个输出值。中间值通过迭代器提供给用户的 reduce 函数。这允许我们处理内存中无法容纳的值列表。

2.1 示例

考虑在大量文档中统计每个单词出现次数的问题。用户将编写类似于以下伪代码的代码：

```
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));
```

map函数会发出每个单词以及与之关联的计数（在这个简单的例子中只是‘1’）。reduce函数会将所有针对特定单词发出的计数加总在一起。

此外，用户会编写代码以 fill 在一个 mapreduce 规范对象中，包含输入和输出 files 的名称，以及可选的调优参数。然后用户调用 MapReduce 函数，并将规范对象传递给它。用户的代码与 MapReduce 库（用 C++ 实现）链接在一起。附录 A 包含此示例的完整程序文本。

2.2 类型

尽管之前的伪代码是以字符串输入和输出来描述的，但概念上用户提供的 map 和 reduce 函数具有相关的类型：

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

也就是说，输入的键和值来自不同的域，而输出的键和值来自同一个域。此外，中间的键和值与输出的键和值来自同一个域。

我们的C++ 实现将字符串传递给用户定义的函数，并让用户代码负责在字符串和适当类型之间进行转换。

2.3 更多示例

这里有一些简单的示例，展示了可以轻松表达为 MapReduce计算的有趣程序。

分布式 Grep：map 函数如果匹配到提供的模式，则发射一行。reduce 函数是一个恒等函数，它只是将提供的中间数据复制到输出。

URL 访问频率计数：map 函数
处理网页请求日志并输出
⟨URL, 1⟩。reduce 函数将所有值加在一起
对于相同的 URL，并发出一个 ⟨URL, 总数⟩ 对。

反向链接图：map函数为每个指向目标的链接输出⟨目标, 源⟩ 对
在名为 source 的页面中找到 URL。reduce 函数将所有与给定目标 URL 关联的 source URL 列表连接起来，并输出该对：

⟨目标, $list$(source)⟩

主机术语向量：术语向量将文档或一组文档中出现的最重要单词总结为 $\langle word, frequency \rangle$ 对列表。映射函数为每个输入文档发出一个 ⟨主机名,术语向量⟩ 对（其中主机名从文档的URL中提取）。归约函数接收给定主机的所有每个文档的术语向量。它将这些术语向量相加，丢弃不频繁的术语，然后发出一个 fi最终⟨主机名,术语向量⟩ 对。

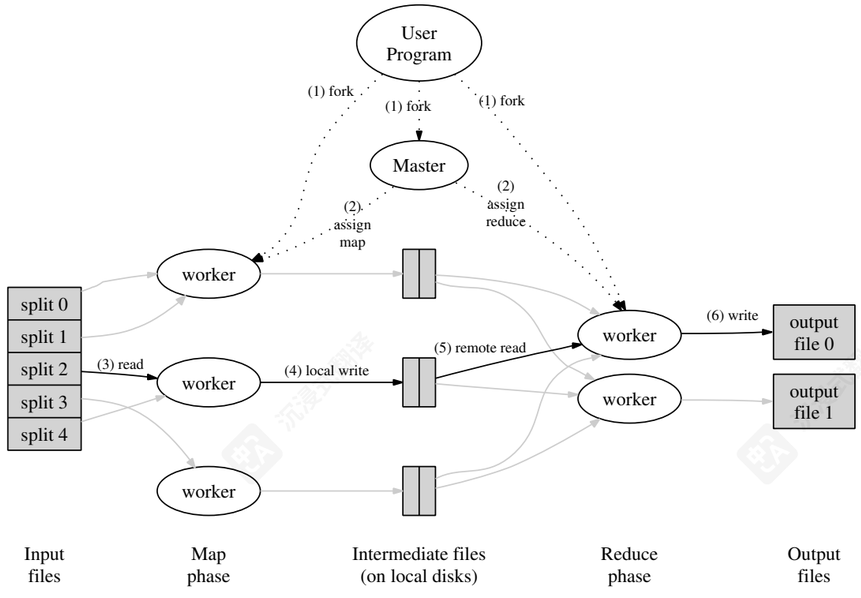


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of ⟨`word, document ID`⟩ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a ⟨`word,` $list$(`document ID`)⟩ 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.

Distributed Sort: The map function extracts the key from each record, and emits a ⟨`key, record`⟩ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

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 multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

(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.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] 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

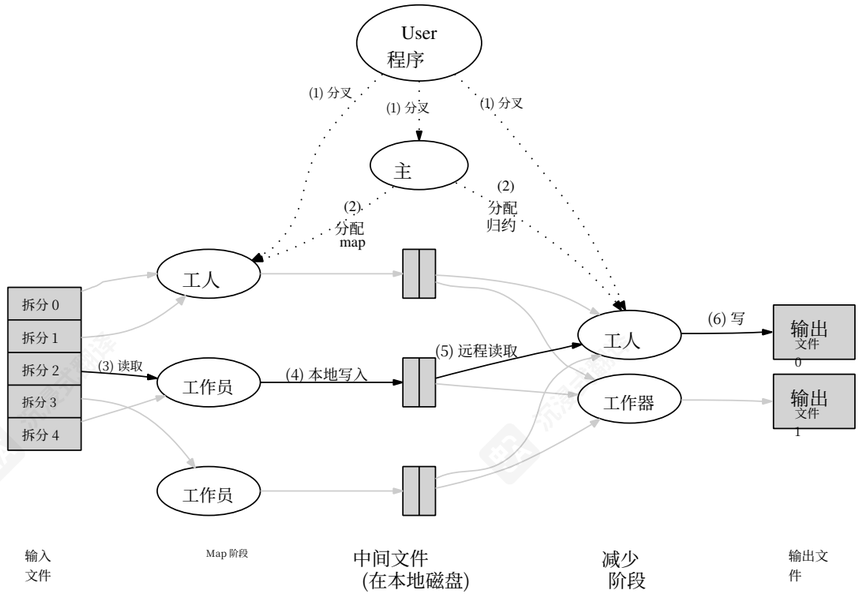


图1：执行概述

倒排索引：map函数解析每个文档，并发出一个由 ⟨单词,文档ID⟩对组成的序列。reduce函数接受给定文档的所有对

单词，对相应的文档ID进行排序并发出一个
⟨单词,$list$(文档ID)⟩ 对。所有输出对的集合形成一个简单的倒排索引。很容易扩展此计算以跟踪单词的位置。

分布式排序：map函数从每条记录中提取键，并输出一个 ⟨键,记录⟩ 对。reduce函数不变地输出所有对。此计算依赖于第4.1节中描述的分区功能，以及第4.2节中描述的排序属性。

第4.1节和第4.2节中描述的排序属性。
第 4.2 节

3 实现

MapReduce接口有许多不同的实现方式。正确的选择取决于环境。例如，一种实现方式可能适用于小型共享内存机器，另一种适用于大型NUMA多处理器，还有一种适用于更大的网络机器集合。

本节描述了一个针对Google广泛使用的计算环境的实现方案：

大量商用PC通过交换式以太网 [4]连接在一起。在我们的环境中：

(1) 机器通常是双处理器x86架构的Linux系统，每台机器拥有2-4GB的内存。

(2) 商品网络硬件在设备层面通常使用 100 兆比特/秒或 1 吉比特/秒，但在整体二分带宽中平均要低得多。

(3) 集群由数百或数千台设备组成，
因此设备故障很常见，.

(4) 存储由直接连接到单个设备的经济型 IDE 硬盘提供。一个分布式 fi 文件系统 [8] 由内部开发用于管理这些磁盘上存储的数据。该 fi 文件系统使用复制来在不可靠的硬件基础上提供可用性和可靠性。

(5) 用户将作业提交给调度系统。每个作业由一组任务组成，并由调度器
映射到集群内一组可用的设备上。

3.1 执行概述

地图调用通过自动分区输入数据，在多台机器上分布。

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 list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) 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 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, 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.

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 as 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 non-idle 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.

Worker Failure

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 re-executed 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 re-executed 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

将输入数据分成一组 M 分片。输入分片可以被不同的机器并行处理。归约调用通过分区中间键空间，使用分区函数（例如，$hash(key)$ mod R）将中间键空间分成 R 块进行分布。分区数量 (R) 和分区函数由用户指定。

图1展示了我们实现中MapReduce操作的总体fl流程。当用户程序调用MapReduce函数时，会发生以下一系列操作（图1中的编号标签对应下方列表中的数字）：

1. 用户程序中的MapReduce库首先将输入 fi文件分割成通常为16兆字节到64兆字节（MB）的 M 块（可通过可选参数由用户控制）。然后，它在机器集群上启动多个程序副本。

2. 程序副本中的一个副本是特殊的 – 主节点。其余的是由主节点分配工作的工作节点。有 M 个map任务和 R 个reduce任务需要分配。主节点会选择空闲的工作节点，并为每个节点分配一个map任务或reduce任务。

3. 被分配了map任务的工作节点会读取相应输入块的内容。它会从输入数据中解析键值对，并将每个对传递给用户定义的Map函数。Map函数产生的中间键值对会缓存在内存中。

4. 定期地，缓存的键值对会被写入本地磁盘，由分区函数将其划分为 R 个区域。这些缓存的键值对在本地磁盘上的位置会被传回主节点，主节点负责将这些位置转发给 reduce 工作节点。

5. 当 reduce 工作节点被主节点通知这些位置时，它使用远程过程调用从 map 工作节点的本地磁盘读取缓存的键值对。当 reduce 工作节点读取完所有中间数据后，它会按中间键对数据进行排序，以便所有相同键的值都聚集在一起。排序是必要的，因为通常许多不同的键会映射到同一个 reduce 任务。如果中间数据量太大无法 fi 存放在内存中，则会使用外部排序。

6. reduce 工作节点会遍历排序后的中间数据，对于遇到的每个唯一中间键，它会将键和对应的中间值集合传递给用户的 Reduce 函数。Reduce 函数的输出会被追加到 fi 个最终输出 fi 文件中，用于这个 reduce 分区。

7. 当所有地图任务和归约任务都已完成时，主节点唤醒用户程序。此时，用户程序中的 MapReduce 调用返回到用户代码。

成功完成后，MapReduce 执行的输出会出现在 R 输出 fi件中（每个归约任务一个，文件名由用户指定）。通常，用户不需要将这些 R 输出fi件合并为一个 fi件 –，他们通常会把这些 fi件作为输入传递给另一个 MapReduce 调用，或从另一个能够处理分片为多个 fi件的输入的分布式应用程序中使用它们。

3.2 主节点数据结构

主节点维护多个数据结构。对于每个地图任务和归约任务，它存储状态（空闲、进行中或已完成），以及非空闲任务的工作机器标识。

主节点是负责将中间件区域的位置从地图任务传播到归约任务的通道。因此，对于每个完成的地图任务，主节点会存储由地图任务生成的中间件区域的位置和大小。当地图任务完成时，会接收到这些位置和大小信息的更新。这些信息会逐步推送给正在执行归约任务的工人。

3.3 容错机制

由于MapReduce库的设计目的是帮助使用数百或数千台机器处理海量数据，因此该库必须能够优雅地容忍机器故障。

工人故障

主节点会定期向每个工人发送心跳。如果在一定时间内没有收到某个工人的响应，主节点会将其标记为故障。该工人完成的任何地图任务都会被重置回初始空闲状态，因此可以重新调度到其他工人上。类似地，在故障工人在进行的任何地图任务或归约任务也会被重置为空闲状态，并可以重新调度。

任务失败时，已完成的地图任务会重新执行，因为它们的输出存储在故障机器的本地磁盘上，因此无法访问。已完成的归约任务不需要重新执行，因为它们的输出存储在全局文件系统中。

当 map 任务首先由 worker A 执行，然后稍后由 worker B 执行（因为 A 失败）时，所有执行 reduce 任务的 worker 都会收到重新执行的通 fi知。任何尚未从 worker A 读取数据的 reduce 任务都将从 worker B 读取数据。

workers executing reduce tasks are notified of the re-execution. 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 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 make the master write periodic checkpoints of the master data structures described above. 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 unlikely; 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.

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 non-faulting 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 just 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 non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic 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 non-deterministic 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 non-deterministic program.

Consider map task M and reduce tasks R_1 and R_2. Let $e(R_i)$ be the execution of R_i 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 [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB 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 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 $O(M + R)$ scheduling decisions and keeps $O(M * R)$ state in memory as described above. (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.)

执行 reduce 任务的 worker 都会收到重新执行的通fi知。任何尚未从 worker A 读取数据的 reduce 任务都将从 worker B读取数据。

MapReduce 对大规模工作节点故障具有弹性。例如，在一次 MapReduce 操作期间，运行集群的网络维护导致每组 80 台机器在几分钟内变得无法访问。MapReduce 主节点仅重新执行了无法访问的工作节点机器所完成的工作，并继续向前推进，最终完成了 MapReduce 操作。

主节点故障

可以轻松让主节点定期保存上述主数据结构的检查点。如果主节点任务死亡，新的副本可以从最后一个检查点状态启动。然而，由于只有一个主节点，其故障可能性很小；因此我们当前的实现如果主节点失败，会中止 MapReduce 计算。客户端可以检查此条件，如果需要，可以重试 MapReduce 操作。

故障存在时的语义

当用户提供的 map 和 reduce 算子是其输入值的确定性函数时，我们的分布式实现会产生与整个程序的非故障顺序执行相同的输出。

我们依靠 map 和 reduce 任务输出的原子提交来实现这一特性。每个进行中的任务将其输出写入私有的临时文件 fi。一个 reduce 任务产生一个这样的 fi文件，而一个 map 任务产生 R 个这样的fi文件（每个 reduce 任务一个）。当 map 任务完成时，工作节点向主节点发送消息，并在消息中包含 R 临时文件 fi的名称。如果主节点收到已完成 map 任务的完成消息，它会忽略该消息。否则，它将文件名称记录在主数据结构中。

当reduce任务完成时，reduce工作进程会原子性地将其临时输出 fi文件重命名为最终输出 fi文件。如果同一个reduce任务在多台机器上执行，则对于同一个最终输出 fi文件 fi，会执行多次重命名调用。我们依赖于底层 fi文件系统提供的原子重命名操作，以确保最终的 fi文件系统状态仅包含reduce任务的一次执行所产生的数据。

我们的大多数map和reduce算子都是确定性的，而我们的语义在这种情况下等同于顺序执行，这使得它非常容易让程序员推理他们程序的行为。当map和/或reduce算子是非确定性的，我们提供较弱但仍然合理的语义。在非确定性算子的存在下，特定reduce任务 R_1 的输出等同于非确定性程序顺序执行产生的 R_1 输出。然而，不同reduce任务 R_2 的输出可能对应非确定性程序不同顺序执行产生的 R_2 输出。

考虑 map 任务 M 和 reduce 任务 R_1 以及 R_2。令 $e(R_i)$ 为 R_i 的执行（只有一个这样的执行）已提交。较弱的语义产生是因为 $e(R_1)$ 可能读取了 M 执行的输出，而 $e(R_2)$ 可能读取了 M 执行的不同输出的结果。

3.4 位置性

网络带宽在我们的计算环境中是一种相对稀缺的资源。我们通过利用输入数据（由 GFS [8] 管理）存储在我们集群组成机器的本地磁盘上的事实来节省网络带宽。GFS 将每个 fi 文件划分为 64 MB 块，并将每个块的多份副本（通常 3 份副本）存储在不同的机器上。MapReduce 主节点会考虑输入 fi 文件的位置信息，并尝试在包含相应输入数据副本的机器上调度一个 map 任务。如果不行，它会尝试在包含该任务输入数据副本附近的机器上调度一个 map 任务（例如，在同一个网络交换机上的工作节点上）。当在集群中大部分工作节点上运行大型 MapReduce 操作时，大多数输入数据都是本地读取的，不会消耗网络带宽。

3.5 任务粒度

我们将映射阶段细分为 M 个片段，将规约阶段细分为 R 个片段，如前所述。理想情况下，M 和 R 应该远大于工作机器的数量。让每个工作节点执行许多不同的任务可以提高动态负载均衡，并且在工作节点失败时也能加快恢复速度：它已经完成的许多映射任务可以被分配到所有其他工作机器上。

在我们的实现中，M 和 R 的大小存在实际限制，因为主节点必须做出$O(M + R)$ 调度决策，并按照上述描述将 $O(M * R)$状态保存在内存中。（不过，内存使用的常数因子很小：$O(M * R)$部分的状态状态大约每个map任务/reduce任务对包含约一个字节数据。）

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 MB 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 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: 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 are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (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. "$hash(key)$ **mod** 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, 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 user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey))$ **mod** R" as the partitioning function causes all URLs from the same host to 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 makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

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 Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, "text"

此外，R 通常受用户约束，因为每个reduce任务的输出最终会存放在一个独立的输出 fi文件中。在实践中，我们倾向于选择 M，使得每个独立任务大约包含16 MB到64MB的输入数据（以便上述的局部性优化效果最佳），并使 R 成为我们预期使用的工人数量的一个较小倍数。我们经常使用 $M = 200,000$ 和$R = 5,000$执行MapReduce计算，使用2,000台工人机器。

3.6 备份任务

MapReduce 操作总耗时延长的一个常见原因是 " 拖延者 ": 一台机器在计算中最后几个 map 或 reduce 任务中，完成某个任务花费了异常长的时间。拖延者可能由多种原因引起。例如，一台磁盘性能差的机器可能会频繁出现可纠正错误，导致其读取性能从 30 MB/s 降至 1 MB/s。集群调度系统可能已在该机器上调度了其他任务，由于 CPU、内存、本地磁盘或网络带宽的竞争，导致其执行 MapReduce 代码速度变慢。我们最近遇到的一个问题是机器初始化代码中的错误，导致处理器缓存被禁用：受影响机器上的计算速度慢了超过一百倍。

我们有一种通用的机制来缓解拖延者问题。当 MapReduce 操作接近完成时，主节点会调度剩余进行中任务的备份执行。无论主执行或备份执行完成，任务都会被标记为完成。我们已经调整了此机制，使其通常不会使操作使用的计算资源增加超过几个百分点。我们发现这显著减少了完成大型 MapReduce 操作的时间。例如，第 5.3 节中描述的排序程序在禁用备份任务机制时，完成时间延长了 44%。

4 Refinements

尽管仅通过编写 Map 和 Reduce 函数提供的基本功能足以满足大多数需求，但我们发现了一些扩展很有用。这些扩展在本节中描述。

4.1 分区函数

MapReduce 的用户指定他们期望的 reduce 任务/输出 fi 件数量 (R)。数据通过一个分区函数在这些任务之间进行分区，该函数基于中间键。

提供了一个默认的分区函数，该函数使用哈希（例如 "$hash(key)$ mod R"）。这通常会导致分区比较均衡。然而，在某些情况下，按键的某些其他函数进行分区可能更有用。例如，有时输出键是 URL，而我们希望同一主机下的所有条目都出现在同一个输出 fi件中。为了支持这种情况，MapReduce 库的用户可以提供一个特殊的分区函数。例如，使用 "$hash(Hostname(urlkey))$ mod R" 作为分区函数会导致来自同一主机的所有 URL 都出现在同一个输出 fi件中。

4.2 排序保证

我们保证在给定分区中，中间的键值对按键的升序处理。这种排序保证可以轻松为每个分区生成排序的输出 file，这在输出 file 格式需要支持基于键的高效随机访问查找时很有用，或者用户希望输出 find 数据是排序的。

4.3 合并函数

在某些情况下，每个 Map 任务生成的中间键存在显著重复，并且用户指定的 Reduce 函数是可交换和可结合的。一个很好的例子是第 2.1 节中的词频统计示例。由于词频往往遵循 Zipf 分布，每个 Map 任务都会生成数百或数千条形如 `<the, 1>` 的记录。所有这些计数都会通过网络发送到一个 Reduce 任务，然后由 Reduce 函数将它们相加，生成一个数字。我们允许用户指定一个可选的 Combiner 函数，该函数在数据通过网络发送之前进行部分合并。

Combiner 函数在每个执行 Map 任务的机器上执行。通常，相同的代码用于实现 Combiner 和 Reduce 函数。Reduce 函数和 Combiner 函数之间的唯一区别在于 MapReduce 库如何处理函数的输出。Reduce 函数的输出写入到 fi最终输出 fi文件。Combiner 函数的输出写入到一个中间 fi文件，该文件将被发送到 Reduce 任务。

部分合并显著加速了某些类别的 MapReduce 操作。附录 A 包含一个使用合并器的示例。

4.4 输入和输出类型

MapReduce 库提供支持以多种不同格式读取输入数据的功能。例如，"文本"

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). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

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 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 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 prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example 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 skips these 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.

4.7 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 of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to 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`).

4.8 Status Information

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, such as 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 pages also contain links to the standard error and standard output files generated by each task. 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. These pages can also be used to figure out when the computation is 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.

4.9 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 named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

模式输入将每一行视为键值对：键是 fi文件中的偏移量，值是行的内容。另一种常见的支持格式按键排序存储键值对序列。每种输入类型实现都了解如何将其拆分为有意义的范围，以便作为单独的 Map 任务进行处理（例如，文本模式的范围拆分确保拆分仅在行边界发生）。用户可以通过提供简单读者接口的实现来添加对新输入类型的支持，尽管大多数用户只是使用少数几个预定义的输入类型。

读者不一定需要提供从 fi 文件中读取的数据。例如，很容易定义一个从数据库读取记录的读者，或者从内存中映射的数据结构读取数据的读者。

类似地，我们支持一组输出类型，用于以不同格式生成数据，并且用户代码很容易添加对新输出类型的支持。

4.5 侧效应

在某些情况下，MapReduce 的用户发现将辅助 fi 文件作为其 map 和/或 reduce 算子的额外输出是有用的。我们依赖于应用程序编写者使此类副作用原子化和幂等。通常，应用程序会写入一个临时 fi 文件，并在生成完成后原子地重命名这个 fi 文件。

我们不提供对单个任务生成的多个输出 fi 文件的原子两阶段提交的支持。因此，需要跨 fi 文件一致性要求的多个输出 fi 文件的任务应该是确定性的。这个限制在实践中从未成为一个问题。

4.6 跳过错误记录

有时用户代码中存在错误，会导致 Map 或 Reduce 函数在某些记录上确定性地崩溃。这类错误会阻止 MapReduce 操作完成。通常的做法是修复 fix 这个错误，但有时这并不可行；也许错误存在于源代码不可用的第三方库中。此外，有时可以忽略少量记录，例如在大型数据集上进行统计分析时。我们提供了一种可选的执行模式，MapReduce 库会检测哪些记录会导致确定性地崩溃，并跳过这些记录以继续执行。

每个工作进程安装一个信号处理器，用于捕获段错误和总线错误。在调用用户 Map 或 Reduce 操作之前，MapReduce 库将参数的序列号存储在一个全局变量中。如果用户代码生成信号，信号处理器会发送一个包含序列号的 "最后挣扎" UDP 数据包给 MapReduce 主节点。当主节点在一个特定记录上检测到多次失败时，它会在下一次重新执行相应的 Map 或 Reduce 任务时指示跳过该记录。

4.7 本地执行

在 Map 或 Reduce 函数中调试问题可能很棘手，因为实际计算发生在分布式系统中，通常涉及数千台机器，而工作分配决策由主节点动态做出。为了便于调试、性能分析和小规模测试，我们开发了一种 MapReduce 库的替代实现，该实现将 MapReduce 操作的所有工作在本地机器上顺序执行。用户可以通过特殊 flag 控制计算仅限于特定的 Map 任务。用户使用特殊 flag 启动程序后，可以轻松使用任何他们认为有用的调试或测试工具（例如 gdb）。

4.8 状态信息

主进程运行一个内部 HTTP 服务器，并导出一套供人类使用的状态页面。状态页面显示计算的进度，例如已完成多少任务、多少任务正在进行中、输入字节数、中间数据字节数、输出字节数、处理速率等。页面还包含每个任务生成的标准错误和标准输出 fi 文件链接。用户可以利用这些数据预测计算将花费多长时间，以及是否需要为计算添加更多资源。这些页面还可以用来 fi 判断计算是否比预期慢得多。

此外，顶级状态页面会显示哪些工作节点失败了，以及它们在失败时正在处理的 Map 和 Reduce 任务。当尝试诊断用户代码中的错误时，这些信息很有用。

4.9 计数器

MapReduce 库提供计数器功能来统计各种事件的发生次数。例如，用户代码可能希望统计处理的总词数或索引的德语文档数量等。

要使用此功能，用户代码创建一个命名计数器对象，然后在 Map 和/或 Reduce 函数中适当地递增计数器。例如：

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

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      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 fraction of the total number of documents processed.

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 data set.

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 100-200 Gbps 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.

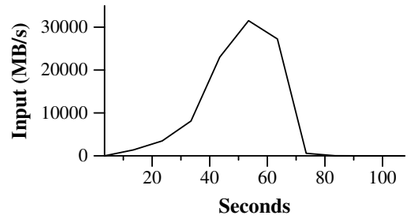


Figure 2: Data transfer rate over time

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 ($R = 1$).

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 [10].

The sorting program consists of less than 50 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


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

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


来自单个工作机器的计数器值会定期传播到主节点（作为 ping 响应的一部分）。主节点聚合成功 Map 和 Reduce 任务的计数器值，并在 MapReduce 操作完成后将它们返回给用户代码。当前计数器值也显示在主节点状态页面上，以便人类可以监控实时计算的进度。在聚合计数器值时，主节点会消除相同 Map 或 Reduce 任务重复执行的影响，以避免重复计数。（重复执行可能源于我们使用备份任务以及由于故障而重新执行任务。）

MapReduce 库会自动维护一些计数器值，例如处理过的输入键值对数量和产生的输出键值对数量。

用户发现计数器功能对于检查 MapReduce 操作的行为很有用。例如，在某些 MapReduce 操作中，用户代码可能希望确保产生的输出对数量正好等于处理的输入对数量，或者处理过的德语文档比例在处理过的文档总数中处于可接受的范围内。

5 性能

在本节中，我们在大型集群机器上运行的两个计算中测量了 MapReduce 的性能。一个计算是在大约一TB的数据中搜索特定模式，另一个计算是对大约一TB的数据进行排序。

这两个程序代表了 MapReduce 用户编写的真实程序的一个大子集。–一类程序将数据从一个表示形式混洗到另一个表示形式，另一类程序从大型数据集中提取少量有趣的数据。

5.1 集群配置

所有程序都在一个由大约1800台机器组成的集群上执行。每台机器配备了两颗2GHz的Intel Xeon处理器（已启用超线程技术）、4GB内存、两个160GB IDE硬盘，以及一个千兆以太网连接。

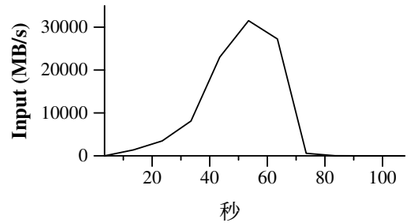


图2：数据传输速率随时间变化

这些机器被安排在一个两层树状交换网络中，总带宽在根节点处约为100-200 Gbps。所有机器都位于同一个托管设施内，因此任何两台机器之间的往返时间都小于1毫秒。

在4GB的内存中，大约有1-1.5GB被集群上运行的其他任务预留。程序是在一个周末下午执行的，当时CPU、磁盘和网络大多处于空闲状态。

5.2 Grep

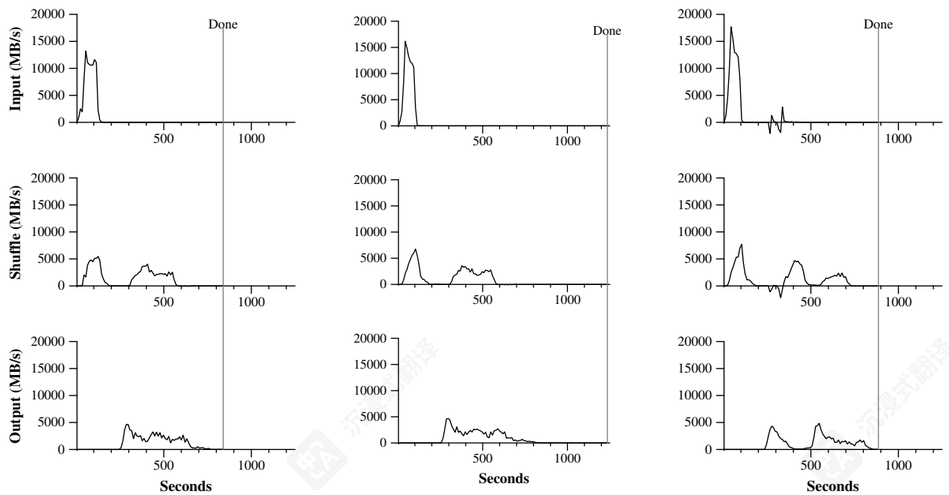
grep程序扫描 10^{10} 100-字节的记录，搜索一个相对罕见的三字符模式（该模式出现在92,337条记录中）。输入被分成大约64MB的块 ($M = 15000$)，整个输出被放在一个 fi文件 ($R = 1$)中。

图2显示了计算随时间的进展情况。Y轴显示了输入数据的扫描速率。随着分配给此MapReduce计算的机器越来越多，速率逐渐提高，在分配了1764个工作者时达到峰值，超过30 GB/s。随着映射任务 fi完成，速率开始下降，在计算开始约80秒时降至零。整个计算从开始到 fi完成大约需要150秒。这包括大约一分钟的启动开销。开销是由于程序传播到所有工作者机器，以及与GFS交互以打开1000个输入 fi文件并获取用于局部性优化的所需信息而产生的延迟。

5.3 排序

排序程序对 10^{10} 100字节的记录进行排序（大约1TB数据）。该程序模仿了TeraSort基准 [10]。

排序程序由不到 50 行用户代码组成。一个三行的 Map 函数从文本行中提取 10 字节的排序键，并输出该键和



(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

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This functions passes the intermediate key/value pair unchanged as the output key/value pair. 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 R pieces.

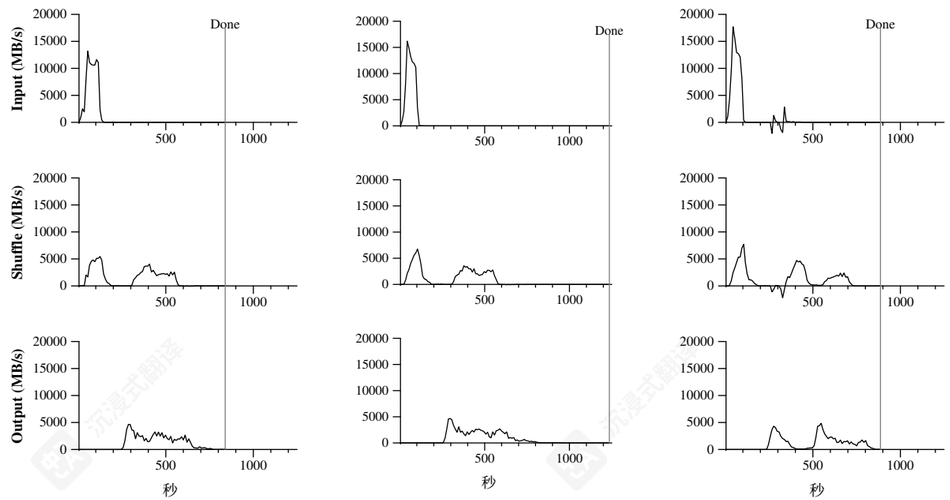
Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting 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 executes at most one reduce task at a time). Roughly 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. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

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 [14] rather than replication.



(a) 正常执行 (b) 无备份任务 (c) 200个任务被终止

图3：不同排序程序执行的数据传输速率随时间变化

排序程序由不到 50 行用户代码组成。一个三行的 Map 函数从文本行中提取 10 字节的排序键，并输出该键以及原始文本行作为中间键值对。我们使用了一个内置的 Identity 函数作为 Reduce 操作符。该函数将中间键值对原样传递作为输出键值对。最终的排序输出写入到一组双向复制的 GFS 文件（即，2TB 作为程序的输出写入）。

如前所述，输入数据被分割成64MB的块（$M = 15000$）。我们将排序后的输出划分为4000fi个块（$R = 4000$）。分区函数使用键的初始字节将其隔离到 R 个块中。

我们此基准测试的分区函数具有关于键分布的内置知识。在一个通用的排序程序中，我们会添加一个预处理的MapReduce操作，该操作会收集键的样本，并使用样本键的分布来计算最终排序遍历的分割点。

图3（a）显示了排序程序正常执行的进度。左上角的图表显示了输入读取的速率。该速率在约13 GB/s时达到峰值，并且由于所有映射任务在200秒之前都已完成，因此很快下降。请注意，输入速率低于grep。这是因为排序映射任务大约花费了它们一半的时间和I/O带宽将中间输出写入其本地磁盘。grep对应的中间输出大小可以忽略不计。

中间左侧的图表显示了从地图任务到归约任务通过网络发送数据的速率。这种混洗fl操作在第一个地图任务完成时立即开始。图表中的第一个峰值是针对大约1700个归约任务的第一个批次（整个MapReduce被分配了大约1700台机器，每台机器一次最多执行一个归约任务）。在计算开始后大约300秒，其中一些 fi个第一批次的归约任务 fi完成，我们开始为剩余的归约任务混洗fl数据。所有的混洗fl操作大约在计算开始后600秒完成。

左下角的图表显示了归约任务向最终输出 fi件 fi写入排序数据的速率。由于机器正在处理中间数据，因此在排序阶段结束和写入阶段开始之间存在延迟。写入速率在一段时间内保持在约2-4 GB/s。所有写入操作在大约计算开始850秒时完成。包括启动开销在内，整个计算过程耗时891秒。这与TeraSort基准测试 [18]当前最佳报告结果1057秒相似。

有几点需要注意：由于我们的本地优化，输入速率高于shuffle速率和输出速率，因为大部分数据是从本地磁盘读取，绕过了我们相对带宽受限的网络。shuffle速率高于输出速率，因为输出阶段会写入排序数据的两个副本（出于可靠性和可用性考虑，我们对输出数据制作了两个副本）。我们写入两个副本，因为这是我们底层 file系统提供的可靠性和可用性机制。如果底层 file系统使用纠删码 [14] 而不是复制，写入数据的网络带宽需求会降低。

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).

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 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,
- 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.

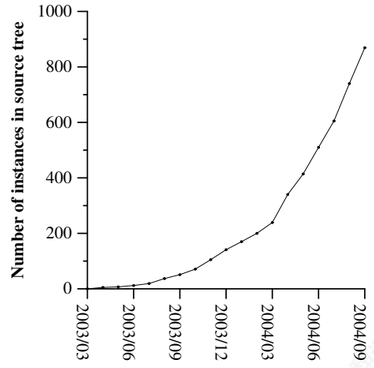


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

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 separate instances as of late September 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 exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

5.4 备份任务的影响

在图3 (b)中，我们展示了禁用备份任务的排序程序执行情况。执行 flow与图3 (a)中所示类似，除了有一个非常长的尾部，几乎没有任何写入活动。960秒后，除了5个reduce任务外，所有reduce任务都已完成。然而，最后这几个掉队者直到300秒后才 fin- ish。整个计算过程耗时1283秒，总耗时增加了44%。

5.5 机器故障

在图3（c）中，我们展示了排序程序的执行情况，其中我们在计算开始几分钟内故意杀死了1746个工作者进程中的200个。底层集群调度器立即在这些机器上重新启动了新的工作者进程（由于只杀死了进程，机器仍然正常运行）。

工作者进程的死亡表现为负的输入速率，因为一些之前完成的map工作消失了（因为相应的map工作者进程被杀死了），需要重新执行。这次重新执行map工作相对较快。整个计算 fi在933秒内完成，包括启动开销（只是正常执行时间的5%增加）。

6 经验

我们在2003年2月编写了MapReduce库的第一个版本，并在2003年8月对其进行了显著的增强，包括位置优化、跨工作者机器的任务执行动态负载均衡等。自那时以来，我们对MapReduce库在我们所处理的问题上的广泛适用性感到非常惊喜。它在谷歌内部被用于广泛的领域，包括：

- 大规模机器学习问题，
- Google News和Froogle产品的聚类问题，
- 用于生成热门查询报告（例如Google Zeitgeist）的数据提取，查询（例如Google Zeitgeist），
- 用于新实验和产品（例如从大量网页语料库中提取地理位置以进行本地化搜索）的网页属性提取，和
- 大规模图计算。

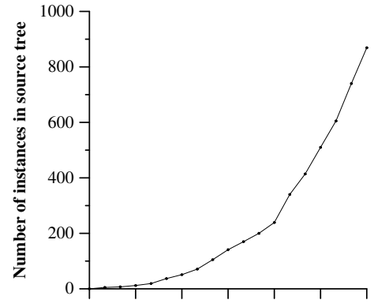


图4：随时间变化的MapReduce实例

工作数量	29,423
平均工作完成时间	634秒
使用的机器天数	79,186天
输入数据已读取	3,288 TB
中间数据已生成	758 TB
输出数据已写入	193 TB
每个作业的平均工人机器数	157
每个作业的平均工人死亡数	1.2
每个作业的平均地图任务数	3,351
每个作业的平均规约任务数	55
唯一的地图实现	395
独特的 reduce 实现	269
独特的 map/reduce 组合	426

表 1：2004 年 8 月运行的 MapReduce 任务

图4显示了从2003年初的0个到2004年9月底近900个独立MapReduce程序，我们主要源代码管理系统中的数量显著增长。MapReduce之所以如此成功，是因为它使得人们可以编写一个简单的程序，并在半小时内高效地运行在千台机器上，大大加快了开发和原型设计周期。此外，它还允许没有分布式和/或并行系统经验的程序员轻松利用大量资源。

每个任务结束时，MapReduce 库会记录关于任务使用的计算资源的统计数据。在表 1 中，我们展示了 2004 年 8 月在 Google 运行的一部分 MapReduce 任务的统计数据。

6.1 大规模索引

到目前为止，我们使用MapReduce最显著的用途之一是对生产索引系统进行了完全重写-

ing 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. The indexing process runs as a sequence of five to ten MapReduce operations. 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, 9, 13]. 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.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], 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. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. 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.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

该系统生成Google网络搜索服务所使用的数据结构。索引系统以我们爬虫系统检索到的文档集合作为输入，这些文档存储为一系列GFS files。这些文档的原始内容超过200TB数据。索引过程作为一系列 fi到十个MapReduce操作运行。使用MapReduce（而不是索引系统先前版本中的自定义分布式过程）提供了几个好处：

• 索引代码更简单、更小，更容易理解，因为处理容错、分布式和并行化的代码被隐藏在MapReduce库中。例如，当使用MapReduce表示时，计算的一个阶段的大小从大约3800行C++ 代码减少到大约700行。

• MapReduce库的性能足够好，我们可以将概念上无关的计算保持分离，而不是将它们混合在一起以避免对数据进行额外遍历。这使得更改索引过程变得容易。例如，在我们旧的索引系统中需要花费几个月才能完成的一项更改，在新的系统中只需几天即可实现。

• 索引过程变得更容易操作，因为机器故障、慢速机器和网络波动引起的大部分问题都由MapReduce库自动处理，无需操作员干预。此外，通过向索引集群添加新机器，可以轻松提高索引过程的性能。

7 相关工作

许多系统提供了受限的编程模型，并利用这些限制自动并行化计算。例如，可以使用并行前缀计算在 $\log N$ 时间内对所有 N 元素数组的前fi缀进行关联函数计算 [6, 9, 13]。MapReduce 可以被视为基于我们对大型实际计算的经验，对其中一些模型进行简化和提炼。更重要的是，我们提供了一种容错实现，可扩展到数千个处理器。相比之下，大多数并行处理系统仅在小规模上实现，并将处理机器故障的细节留给程序员。

批量同步编程 [17] 和一些MPI原语 [11] 提供了更高层次的抽象，使程序员更容易编写并行程序。这些系统与MapReduce之间的一个关键区别在于，MapReduce利用受限的编程模型自动并行化用户程序并提供透明的容错能力。

我们的局部性优化借鉴了诸如主动磁盘 [12, 15], 等技术，将计算推送到靠近本地磁盘的处理单元，以减少跨 I/O 子系统或网络传输的数据量。我们在直接连接少量磁盘的商用处理器上运行，而不是直接在磁盘控制器处理器上运行，但总体方法相似。

我们的备份任务机制与Charlotte系统中采用的急速调度机制类似 [3]。简单急速调度的一个缺点是，如果某个任务导致重复失败，整个计算将无法完成。我们 fix通过我们的跳过坏记录机制来处理这个问题。

MapReduce实现依赖于一个内部集群管理系统，该系统负责在大量共享机器上分配和运行用户任务。虽然本文的重点不是集群管理系统，但它与其他系统（如Condor [16]）在精神上类似。

MapReduce库中的排序功能在操作上与NOW-Sort [1]类似。源机器（映射工作者）将待排序数据分区，并将其发送给其中一个 R 归约工作者。每个归约工作者在其本地（如果可能则在内存中）对数据进行排序。当然，NOW-Sort没有我们库中广泛适用的用户可配置的映射和归约功能。

River [2] 提供了一个编程模型，其中进程通过在分布式队列之间发送数据来相互通信。与MapReduce类似，River系统试图在异构硬件或系统扰动引入的非均匀性存在的情况下提供良好的平均性能。River通过仔细调度磁盘和网络传输以实现均衡的完成时间来实现这一点。MapReduce采用不同的方法。通过限制编程模型，MapReduce框架能够将问题划分为大量细粒度的 fi任务。这些任务在可用的工作器上动态调度，以便更快的工作器处理更多任务。受限的编程模型还允许我们在作业接近结束时调度任务的冗余执行，这大大减少了非均匀性（如慢速或卡住的工作器）存在时的完成时间。

BAD-FS [5] 的编程模型与MapReduce差异很大，并且与MapReduce不同，它面向的是跨广域网络的作业执行。

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

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, for data mining, for 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.

We have learned several things from this work. First, restricting the programming model makes 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 based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

然而，存在两个基本相似之处。(1) 两种系统都使用冗余执行来从由故障导致的数据丢失中恢复。(2) 两种系统都使用感知位置的调度来减少通过拥塞网络链路传输的数据量。

TACC [7] 是一个旨在简化高可用性网络服务构建的系统。与MapReduce类似，它依赖于重执行作为实现容错性的机制。

8 结论

MapReduce编程模型已在Google成功用于许多不同的目的。我们将这种成功归因于几个原因。首先，该模型易于使用，即使对于没有并行和分布式系统经验的程序员也是如此，因为它隐藏了并行化、容错性、局部性优化和负载均衡的细节。其次，各种问题都很容易表达为MapReduce计算。例如，MapReduce用于生成Google生产网络搜索服务的数据、排序、数据挖掘、机器学习以及其他许多系统。第三，我们开发了一个MapReduce的实现，可以扩展到包含数千台机器的大型集群。该实现有效地利用这些机器资源，因此适合用于Google遇到的许多大型计算问题。

我们从这项工作中学到了几件事。首先，限制编程模型使得并行化和分布式计算变得容易，并且可以使得此类计算具有容错性。其次，网络带宽是一种稀缺资源。因此，我们系统中的许多优化都旨在减少跨网络发送的数据量：局部性优化允许我们从本地磁盘读取数据，将中间数据的一个副本写入本地磁盘可以节省网络带宽。第三，冗余执行可以用来减少慢机器的影响，并处理机器故障和数据丢失。

致谢

Josh Levenberg 在修订和扩展用户级 MapReduce API 方面发挥了关键作用，他基于自己使用 MapReduce 的经验以及其他人对改进的建议添加了多个新功能。MapReduce 从 Google 文件系统 [8] 读取输入并写入输出。我们感谢 Mohit Aron、Howard Gobioff、Markus Gutschke、David Kramer、Shun-Tak Leung 和 Josh Redstone 为开发 GFS 所做的工作，我们同样感谢 Percy Liang 和 Olcan Sercinoglu 为开发 MapReduce 所使用的集群管理系统所做的工作。Mike Burrows、Wilson Hsieh、Josh Levenberg、Sharon Perl、Rob Pike 和 Debby Wallach 对本文初稿提出了有益的建议。匿名 OSDI 审稿人以及我们的指导者 Eric Brewer 提供了许多关于改进本文的建议。最后，我们感谢 Google 工程组织内所有使用 MapReduce 的用户，感谢他们提供的帮助反馈、建议和错误报告。

参考文献

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.

[6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#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++;
      if (start < i)
        Emit(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_REDUCER(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 2000
  // 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;
}
```


[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com/barc/SortBenchmark/.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

[17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.

[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com/cs/spsort.pdf.

词频统计

本节包含一个程序，该程序统计输入 files 中每个唯一单词的出现次数，这些 files 在命令行上指定。

```
#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++;
      if (start < i)
        Emit(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_REDUCER(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 2000
  // 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;
}
```