MapReduce: Simplified Data Processing on Large Clusters

MapReduce:大型集群上的简化数据处理

Jeffrey Dean and Sanjay Ghemawat

杰弗里·迪恩和桑杰·格玛瓦特

jeff@google.com , sanjay@google.com

sanjay@google.com jeff@google.com

Google, Inc.

谷歌公司。

Abstract

摘要

MapReduce is a programming model and an associ-ated 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.

MapReduce 是一个用于处理和生成大型数据集的编程模型和相关实现。用户指定一个映射函数，该函数处理一个键/值对以生成一组中间键/值对，以及一个归约函数，该函数合并与同一中间键相关联的所有中间值。许多现实世界的任务都可以在这个模型中表达出来，如本文所示。

Programs written in this functional style are automati-cally parallelized and executed on a large cluster of com-modity machines.The run-time system takes care of the details of partitioning the input data, scheduling the pro-gram's execution across a set of machines, handling ma-chine failures, and managing the required inter-machine communication.This allows programmers without any experience with parallel and distributed systems to eas-ily utilize the resources of a large distributed system.

以这种功能风格编写的程序被自动并行化，并在一大群并行机器上执行。运行时系统负责输入数据的划分、在一组机器上调度程序的执行、处理机器故障以及管理所需的机器间通信等细节。这使得对并行和分布式系统没有任何经验的程序员能够轻松地利用大型分布式系统的资源。

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

我们的 MapReduce 实现运行在一个大型商用机器集群上，并且具有很高的可伸缩性:一个典型的 MapReduce 计算在数千台机器上处理大量数据。程序员发现这个系统很容易使用:数百个 MapReduce 程序已经实现，每天有超过 1000 个 MapReduce 作业在谷歌的集群上执行。

1 Introduction

1 导言

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

在过去的五年里，作者和谷歌的许多其他人已经实现了数百种特殊用途的计算，处理大量原始数据，如被抓取的文档、网络请求日志等。，以计算各种派生数据，例如倒排索引、web 文档的图形结构的各种表示、每个主机爬网的页数的摘要、

given day, etc.Most such computations are conceptu-ally 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 par-allelize the computation, distribute the data, and handle failures conspire to obscure the original simple compu-tation with large amounts of complex code to deal with these issues.

给定日期等。大多数这样的计算概念上很简单。然而，输入数据通常很大，计算必须分布在数百或数千台机器上，以便在合理的时间内完成。如何并行计算、分发数据和处理故障的问题，共同掩盖了最初的简单计算，需要大量复杂的代码来处理这些问题。

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computa-tions we were trying to perform but hides the messy de-tails of parallelization, fault-tolerance, data distribution and load balancing in a library.Our abstraction is in-spired 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 op-eration 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 ap-propriately.Our use of a functional model with user-specified map and reduce operations allows us to paral-lelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

作为对这种复杂性的反应，我们设计了一种新的抽象，它允许我们表达我们试图执行的简单计算，但是隐藏了并行化、容错、数据分布和库中负载平衡的混乱的缺点。我们的抽象是由 Lisp 和许多其他函数语言中的映射和简化原语所激发的。我们意识到，我们的大部分计算涉及到对输入中的每个逻辑“记录”应用映射运算，以便计算一组中间键/值对，然后对共享同一键的所有值应用一个约简运算，以便适当地组合导出的数据。我们使用具有用户指定的映射和简化操作的功能模型，允许我们容易地并行化大型计算，并使用重新执行作为容错的主要机制。

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.

这项工作的主要贡献是提供了一个简单而强大的接口，能够实现大规模计算的自动并行化和分布，并实现了在大型商用电脑集群上实现高性能的接口。

Section 2 describes the basic programming model and gives several examples.Section 3 describes an imple-mentation of the MapReduce interface tailored towards our cluster-based computing

environment.Section 4 de-scribes 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

第 2 节描述了基本的编程模型并给出了几个例子。第 3 节描述了为我们基于集群的计算环境定制的 MapReduce 接口的实现。第 4 节描述了我们发现有用的编程模型的几种改进。第 5 节针对各种任务对我们的实现进行了性能测量。第 6 节探讨了地图缩减在谷歌中的使用，包括我们使用它作为基础的经验

137

for a rewrite of our production indexing system.Sec-tion 7 discusses related and future work.

重写我们的生产索引系统。第 7 节讨论了相关的和未来的工作。

2 Programming Model

2 编程模型

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.

该计算采用一组输入键/值对，并产生一组输出键/值对。MapReduce 库的用户将计算表示为两个函数:Map 和 Reduce。

Map, written by the user, takes an input pair and pro-duces a set of intermediate key/value pairs.The MapRe-duce library groups together all intermediate values asso-ciated with the same intermediate key I and passes them to the Reduce function.

由用户编写的 Map 获取一个输入对，并生成一组中间键/值对。映射简化库将所有与同一中间键相关联的中间值组合在一起，并将它们传递给简化函数。

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 val-ues are supplied to the user's reduce function via an iter-ator.This allows us to handle lists of values that are too large to fit in memory.

同样由用户编写的 Reduce 函数接受一个中间键 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。通常每个缩减调用只产生零个或一个输出值。中间值通过调节器提供给用户的还原功能。这使我们能够处理太大而无法存储的值列表。

2.1 Example

2.1 示例

Consider the problem of counting the number of oc-currences of each word in a large collection of docu-ments.The user would write code similar to the follow-ing pseudo-code:

考虑一下在大量的文档中计算每个单词的 oc-currency 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:

映射(字符串键，字符串值)://键:文档名称//值:每个单词的文档内容值:

EmitIntermediate(w, "1");

发射中间体(w，“1”)；

reduce(String key, Iterator values): // key: a word

缩减(字符串键，迭代器值)://键:一个单词

// values: a list of counts int result = 0;for each v in values:

//值:计数列表 int 结果= 0；对于每个 v 值:

result += ParseInt(v);Emit(AsString(result));

结果+= ParSeint(v)；发出(字符串(结果))；

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.

map 函数发出每个单词加上相关的出现次数(在这个简单的例子中只有“1”)。reduce 函数将特定单词发出的所有计数相加。

In addition, the user writes code to fill in a mapreduce specification object with the names of the input and out-put files, and optional tuning parameters.The user then invokes the MapReduce function, passing it the specifi-cation 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.

此外，用户编写代码，用输入和输出文件的名称以及可选的优化参数填充 mapreduce 规范对象。然后，用户调用 MapReduce 函数，将指定对象传递给它。用户代码与 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 Types

2.2 类型

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:

尽管前面的伪代码是根据字符串输入和输出编写的，但从概念上讲，用户提供的映射和简化函数有相关的类型:

map (k1,v1) → list(k2,v2)

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 do-main as the output keys and values.

reduce (k2，list(v2)) → list(v2)，即输入键和值来自与输出键和值不同的域。此外，中间键和值与输出键和值来自同一个 do-main。

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.

我们的 C++实现将字符串传入和传出用户定义的函数，并让用户代码在字符串和适当的类型之间进行转换。

2.3 More Examples

2.3 更多示例

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computa-tions.

这里有几个有趣程序的简单例子，可以很容易地用 MapReduce 计算来表示。

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 intermedi-ate data to the output.

分布式 Grep:如果匹配提供的模式，映射函数会发出一条线。reduce 函数是一个标识函数，它只是将提供的中间数据复制到输出中。

Count of URL Access Frequency: The map func-tion processes logs of web page requests and outputs hURL, 1i.The reduce function adds together all values for the same URL and emits a hURL, total counti pair.

网址访问频率的计数:映射功能处理网页请求的日志并输出 1i。缩减函数将同一网址的所有值相加，并发出一个 HOLl，总计数对。

Reverse Web-Link Graph: The map function outputs htarget, sourcei pairs for each link to a target URL found in a page named source.The reduce function concatenates the list of all source URLs as-sociated with a given target URL and emits the pair: htarget, list(source)i

反向网络链接图:映射函数为每个链接输出一个目标源对，该链接指向一个名为 source 的页面中的目标 URL。reduce 函数将所有与给定目标网址相关联的源网址列表连接起来，并发出一对:目标，列表(源)

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 hword, frequencyi pairs.The map function emits a hhostname, term vectori pair for each input document (where the hostname is extracted from the URL of the document).The re-duce 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 hhostname, term vectori pair.

每台主机的术语向量:术语向量将出现在一个或一组文档中的最重要的单词总结为一系列词序、频率对。映射函数为每个输入文档发出一个 hhostname，术语向量对(其中主机名是从文档的网址中提取的)。给定主机的所有每个文档的术语向量都被传递给 re-duce 函数。它将这些术语向量加在一起，丢弃不常用的术语，然后发出一个最终的 hhostname，术语向量对。

138

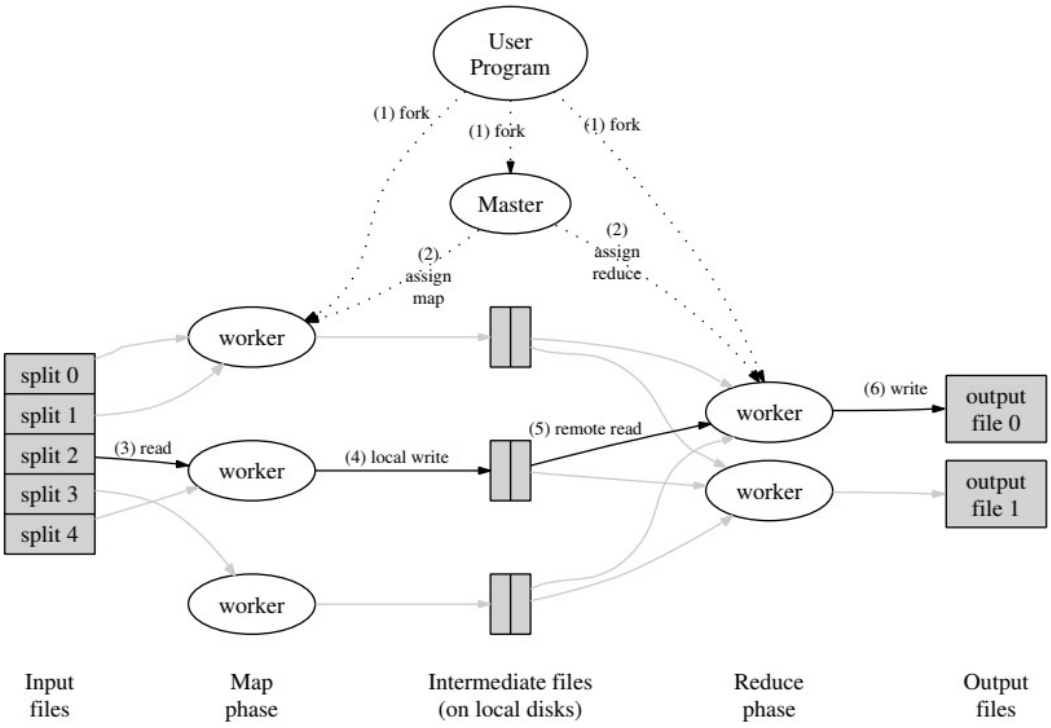


Figure 1: Execution overview

图 1:执行概述

Inverted Index: The map function parses each docu-ment, and emits a sequence of hword, document IDi pairs.The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a hword, list(document ID)i 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.

倒排索引:映射函数解析每个文档，并发出一个 hword，文档 IDi 对的序列。reduce 函数接受给定单词的所有对，对相应的文档标识进行排序，并发出一个 hword，list(文档标识)I 对。所有输出对的集合形成一个简单的倒排索引。增加这个计算来跟踪单词位置是很容易的。

Distributed Sort: The map function extracts the key from each record, and emits a hkey, recordi pair.The reduce function emits all pairs unchanged.This compu-tation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Sec-tion 4.2.

分布式排序:映射函数从每个记录中提取关键字，并发出一个 hkey，recordi 对。reduce 函数不变地发出所有对。这种计算取决于第 4.1 节中描述的划分工具和第 4.2 节中描述的排序属性。

3 Implementation

3 实施

Many different implementations of the MapReduce in-terface 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.

MapReduce 界面的许多不同实现都是可能的。正确的选择取决于环境。例如，一种实现可能适用于小型共享存储器机器，另一种适用于大型 NUMA 多处理器，还有一种适用于更大的联网机器集合。

This section describes an implementation targeted to the computing environment in wide use at Google:

本节描述了针对谷歌广泛使用的计算环境的实施:

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 over-all bisection bandwidth.

用交换式以太网连接在一起的大型商用电脑集群[4]。在我们的环境中:(1)机器通常是运行 Linux 的双处理器 x86 处理器，每台机器有 2-4 GB 的内存。(2)使用商用网络硬件-通常在机器级别上为 100 兆位/秒或 1 千兆位/秒，但平均整体二等分带宽要小得多。

(3) A cluster consists of hundreds or thousands of ma-chines, and therefore machine failures are common.(4) Storage is provided by inexpensive IDE disks at-tached 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.

(3)集群由数百或数千台机器组成，因此机器故障很常见。(4)存储由廉价的集成开发环境磁盘提供，直接连接到单独的机器。内部开发的分布式文件系统[8]用于管理存储在这些磁盘上的数据。文件系统使用复制在不可靠的硬件上提供可用性和可靠性。

(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.

(5)用户向调度系统提交作业。每个作业由一组任务组成，并由调度程序映射到群集中的一组可用机器。

3.1 Execution Overview

3.1 执行概述

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

通过自动划分输入数据，映射调用分布在多台机器上

into a set of M splits.The input splits can be pro-cessed in parallel by different machines.Reduce invoca-tions 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.

分成一组 M 个部分。不同的机器可以并行处理输入分割。通过使用划分函数(例如，散列(密钥)模块)将中间密钥空间划分成 R 个片段来分布减少的调用。分区数量和分区功能由用户指定。

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

图 1 显示了我们的实现中的 MapReduce 操作的整体流程。当用户程序调用 MapReduce 函数时，会出现以下操作序列(图 1 中的编号标签对应于下面列表中的编号):

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 (con-trollable by the user via an optional parameter).It then starts up many copies of the program on a clus-ter of machines.

1.用户程序中的 MapReduce 库首先将输入文件分成 M 个块，每个块通常为 16 到 64 兆字节(可由用户通过可选参数控制)。然后它在一堆机器上启动程序的许多副本。

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.

2.程序的一个副本是特殊的——主程序。其余的是由主人分配工作的工人。有 M 个地图任务和 R 个简化任务要分配。主人挑选空闲的工人，并给每个人分配一个地图任务或减少任务。

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 interme-diate key/value pairs produced by the Map function are buffered in memory.

3.分配有地图任务的工作人员读取相应输入分割的内容。它从输入数据中解析出键/值对，并将每一对传递给用户定义的映射函数。映射函数产生的中间键/值对被缓冲在内存中。

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.

4.定期地，缓冲的对被写入本地磁盘，通过分区功能被划分成 R 个区域。这些缓冲对在本地磁盘上的位置被传递回主服务器，主服务器负责将这些位置转发给精简工作人员。

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 in-termediate 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.

5.当主节点通知缩减工作节点这些位置时，它使用远程过程调用从映射工作节点的本地磁盘读取缓冲数据。当缩减工作器读取了所有中间数据后，它会按照中间键对其进行排序，以便将同一键的所有出现组合在一起。需要排序，因为通常许多不同的键映射到同一个简化任务。如果中间数据量太大，内存无法容纳，则使用外部排序。

6.The reduce worker iterates over the sorted interme-diate data and for each unique intermediate key en-countered, it passes the key and the corresponding set of intermediate values to the user's Reduce func-tion.The output of the Reduce function is appended to a final output file for this reduce partition.

6.归约工作器遍历已排序的中间数据，对于每个唯一的中间键值，它将键值和相应的中间值集传递给用户的归约函数。缩减函数的输出被附加到这个缩减分区的最终输出文件中。

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 pro-gram returns back to the user code.

7.当所有地图任务和缩小任务完成后，主程序会唤醒用户程序。此时，用户程序中的 MapReduce 调用返回到用户代码。

After successful completion, the output of the mapre-duce 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 dis-tributed application that is able to deal with input that is partitioned into multiple files.

成功完成后，映射生成执行的输出在 R 输出文件中可用(每个缩减任务一个，文件名由用户指定)。通常，用户不需要将这些 R 输出文件合并到一个文件中——他们通常将这些文件作为输入传递给另一个 MapReduce 调用，或者从另一个分布式应用程序中使用它们，该应用程序能够处理被划分成多个文件的输入。

3.2 Master Data Structures

3.2 主数据结构

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).

主服务器保留几个数据结构。对于每个映射任务和缩减任务，它存储状态(空闲、正在进行或已完成)和工作机的标识(对于非空闲任务)。

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 inter-mediate 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 incre-mentally to workers that have in-progress reduce tasks.

主文件是一个管道，通过它可以从地图任务中传播中间文件区域的位置，以减少任务。因此，对于每个已完成的地图任务，主机存储由地图任务产生的中间文件区域的位置和大小。地图任务完成后，会收到此位置和大小信息的更新。这些信息被渐进地推给正在减少任务的员工。

3.3 Fault Tolerance

3.3 容错

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.

由于 MapReduce 库旨在帮助使用数百或数千台机器处理大量数据，因此该库必须能够容忍机器故障。

Worker Failure

工人失败

The master pings every worker periodically.If no re-sponse 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 ini-tial idle state, and therefore become eligible for schedul-ing 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.

师傅定期给每个工人打电话。如果在一定时间内没有收到工作人员的响应，主机会将该工作人员标记为失败。由工作人员完成的任何映射任务都被重置回其初始空闲状态，因此有资格在其他工作人员上进行调度。类似地，在失败的工作机上正在进行的任何映射任务或缩减任务也将被重置为空闲，并可重新调度。

Completed map tasks are re-executed on a failure be-cause 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.

已完成的映射任务将在故障时重新执行，因为它们的输出存储在故障机器的本地磁盘上，因此不可访问。已完成的缩减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。

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

当一个映射任务首先由工作人员 A 执行，然后由工作人员 B 执行(因为 A 失败了)，所有

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.

执行 reduce 任务的工作人员会收到重新执行的通知。任何尚未从工作进程 A 读取数据的缩减任务都将从工作进程 b 读取数据

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 sev-eral minutes.The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually complet-ing the MapReduce operation.

MapReduce 能够应对大规模的工作人员故障。例如，在一次 MapReduce 操作中，正在运行的群集上的网络维护导致一次有 80 台机器的组在几分钟内变得不可访问。MapReduce 主机只是简单地重新执行无法访问的工作机所做的工作，并继续向前推进，最终完成 MapReduce 操作。

Master Failure

主失败

It is easy to make the master write periodic checkpoints of the master data structures described above.If the mas-ter 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 cur-rent implementation aborts the MapReduce computation if the master fails.Clients can check for this condition and retry the MapReduce operation if they desire.

很容易使主写周期性地检查上述主数据结构。如果 mas-ter 任务终止，可以从最后一个检查点状态开始新的拷贝。然而，鉴于只有一个主人，它的失败是不可能的；因此，如果主服务器失败，我们当前的实现将中止 MapReduce 计算。客户端可以检查这种情况，如果需要，可以重试 MapReduce 操作。

Semantics in the Presence of Failures

出现故障时的语义

When the user-supplied map and reduce operators are de-terministic 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.

当用户提供的 map 和 reduce 操作符是其输入值的决定函数时，我们的分布式实现将产生与整个程序的无错顺序执行所产生的输出相同的输出。

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.

我们依靠映射的原子提交和减少任务输出来实现这个特性。每个进行中的任务将其输出写入私有临时文件。缩减任务产生一个这样的文件，而映射任务产生 R 个这样的文件(每个缩减任务一个)。当映射任务完成时，工作人员向主服务器发送一条消息，并在消息中包含 R 个临时文件的名称。如果主机收到已完成的地图任务的完成消息，它将忽略该消息。否则，它会在主数据结构中记录 R 文件的名称。

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 multi-ple 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 guar-antee that the final file system state contains just the data produced by one execution of the reduce task.

当缩减任务完成时，缩减工作器会自动将其临时输出文件重命名为最终输出文件。如果在多台机器上执行相同的缩减任务，将对同一个最终输出文件执行多个重命名调用。我们依靠底层文件系统提供的原子重命名操作来确保最终文件系统状态只包含一次执行缩减任务产生的数据。

The vast majority of our map and reduce operators are deterministic, and the fact that our semantics are equiv-alent to a sequential execution in this case makes it very

我们的 map 和 reduce 操作符的绝大部分是确定性的，事实上我们的语义等同于这种情况下的顺序执行，这使得它非常

easy for programmers to reason about their program's be-havior.When the map and/or reduce operators are non-deterministic, we provide weaker but still reasonable se-mantics.In the presence of non-deterministic operators, the output of a particular reduce task R1 is equivalent to the output for R1 produced by a sequential execution of the non-deterministic program.However, the output for a different reduce task R2 may correspond to the output for R2 produced by a different sequential execution of the non-deterministic program.

程序员很容易对他们程序的行为进行推理。当映射和/或约简算子不确定时，我们提供较弱但仍然合理的语义。在存在非确定性操作符的情况下，特定缩减任务 R1 的输出相当于由非确定性程序的顺序执行产生的 R1 的输出。然而，不同缩减任务 R2 的输出可以对应于由非确定性程序的不同顺序执行产生的 R2 的输出。

Consider map task M and reduce tasks R1 and R2.Let e(Ri) be the execution of Ri that committed (there is exactly one such execution).The weaker semantics arise because e(R1) may have read the output produced by one execution of M and e(R2) may have read the output produced by a different execution of M.

考虑地图任务 M，减少任务 R1 和 R2。让 e(Ri)是所犯 Ri 的执行(有一个这样的执行)。较弱的语义出现是因为 e(R1)可能已经读取了由 M 的一次执行产生的输出，而 e(R2)可能已经读取了由 M 的不同执行产生的输出

3.4 Locality

3.4 地点

Network bandwidth is a relatively scarce resource in our computing environment.We conserve network band-width 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 corre-sponding 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.

在我们的计算环境中，网络带宽是相对稀缺的资源。我们利用输入数据(由 GFS [8]管理)存储在组成集群的机器的本地磁盘上这一事实来节省网络带宽。GFS 将每个文件分成 64 MB 的块，并在不同的机器上存储每个块的多个副本(通常为 3 个副本)。MapReduce 主机将输入文件的位置信息考虑在内，并尝试在包含相应输入数据副本的机器上调度地图任务。否则，它会尝试在任务输入数据的副本附近(例如，在与包含数据的机器位于同一网络交换机上的工作机上)调度映射任务。当在集群中的很大一部分工作人员上运行大型 MapReduce 操作时，大多数输入数据都是在本地读取的，不会消耗网络带宽。

3.5 Task Granularity

3.5 任务粒度

We subdivide the map phase into M pieces and the re-duce 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.

如上所述，我们将映射阶段细分为 M 个部分，将转换阶段细分为 R 个部分。理想情况下，机器的数量应该比工人机器的数量多得多。让每个工作人员执行许多不同的任务可以改善动态负载平衡，还可以在工作人员失败时加快恢复速度:它已经完成的许多地图任务可以分布在所有其他工作人员计算机上。

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 fac-tors 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.)

在我们的实现中，M 和 R 的大小是有实际限制的，因为主机必须做出 O(M + R)调度决策，并如上所述在内存中保持 O(M * R)状态。(然而，内存使用的常量因子很小:状态的 0(M * R)部分由每个映射任务/缩减任务对大约一个字节的数据组成。(

141

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate out-put 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 num-ber of worker machines we expect to use.We often per-form MapReduce computations with M = 200, 000 and R = 5, 000, using 2,000 worker machines.

此外，R 经常受到用户的限制，因为每个缩减任务的输出都在一个单独的输出文件中结束。在实践中，我们倾向于选择 M，这样每个单独的任务大约有 16 MB 到 64 MB 的输入数据(这样上面描述的局部优化是最有效的)，并且我们使 R 是我们期望使用的工作机数量的一个很小的倍数。我们经常使用 2000 台工作机，用 M = 200，000 和 R = 5，000 来进行每窗体的 MapReduce 计算。

3.6 Backup Tasks

3.6 备份任务

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a ma-chine 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 ex-ample, a machine with a bad disk may experience fre-quent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling sys-tem 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 net-work bandwidth.A recent problem we experienced was a bug in machine initialization code that caused proces-sor caches to be disabled: computations on affected ma-chines slowed down by over a factor of one hundred.

延长 MapReduce 操作总时间的一个常见原因是“散兵游勇”:一台机器花费了异常长的时间来完成最后几个地图中的一个或减少计算中的任务。掉队者的出现有很多原因。例如，具有坏磁盘的计算机可能会遇到可纠正的错误，从而使其读取性能从 30 兆字节/秒降低到 1 兆字节/秒。群集调度系统可能已经在该计算机上调度了其他任务，由于对中央处理器、内存、本地磁盘或网络带宽的竞争，导致其执行 MapReduce 代码的速度更慢。我们最近遇到的一个问题是机器初始化代码中的一个错误，它导致进程排序缓存被禁用:在受影响的机器上的计算速度降低了 100 多倍。

We have a general mechanism to alleviate the prob-lem 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 exam-ple, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

我们有一个通用的机制来缓解掉队者的问题。当一个 MapReduce 操作接近完成时，主服务器会调度剩余正在进行的任务的备份执行。只要主执行或备份执行完成，任务就会被标记为已完成。我们已经调整了这种机制，因此它通常会将操作所使用的计算资源增加不超过几个百分点。我们发现这大大减少了完成大型 MapReduce 操作的时间。例如，当备份任务机制被禁用时，第 5.3 节中描述的排序程序需要花费 44%的时间来完成。

4 Refinements

4 项改进

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.

尽管简单地编写映射和缩减函数所提供的基本功能对于大多数需求来说已经足够了，但是我们已经发现了一些有用的扩展。这些将在本节中介绍。

4.1 Partitioning Function

4.1 划分功能

The users of MapReduce specify the number of reduce tasks/output files that they desire (R).Data gets parti-tioned across these tasks using a partitioning function on

MapReduce 的用户指定他们想要的缩减任务/输出文件的数量。使用上的分区函数跨这些任务对数据进行分区

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 par-titioning function causes all URLs from the same host to end up in the same output file.

中间键。提供了使用散列(例如，“散列(密钥)模块”)的默认分区功能。这往往会产生相当平衡的分区。然而，在某些情况下，用键的其他功能来划分数据是有用的。例如，有时输出关键字是 URL，我们希望单个主机的所有条目都在同一个输出文件中结束。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“哈希(主机名(urlkey)) mod R”作为分区函数会导致来自同一主机的所有 URL 最终出现在同一个输出文件中。

4.2 Ordering Guarantees

4.2 订购保证

We guarantee that within a given partition, the interme-diate key/value pairs are processed in increasing key or-der.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 con-venient to have the data sorted.

我们保证在给定的分区内，中间的键/值对以递增的键或顺序进行处理。这种排序保证使得为每个分区生成一个已排序的输出文件变得容易，当输出文件格式需要支持有效的按键随机访问查找，或者输出的用户发现对数据进行排序很方便时，这是很有用的。

4.3 Combiner Function

4.3 组合器功能

In some cases, there is significant repetition in the inter-mediate keys produced by each map task, and the user-specified Reduce function is commutative and associa-tive.A good example of this is the word counting exam-ple 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 re-duce 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.

在某些情况下，由每个映射任务产生的中间键有显著的重复，并且用户指定的缩减函数是可交换的和关联的。这方面的一个很好的例子是第 2.1 节中的单词计数例子。由于词频倾向于遵循齐夫分布，每个地图任务将产生数百或数千条< the，1 >形式的记录。所有这些计数将通过网络发送到一个单一的减少任务，然后通过减少功能加在一起产生一个数字。我们允许用户指定一个可选的合并器功能，在数据通过网络发送之前对其进行部分合并。

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 func-tions.The only difference between a reduce function and a combiner function is how the MapReduce library han-dles 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.

组合器功能在执行地图任务的每台机器上执行。通常使用相同的代码来实现合并器和缩减功能。缩减函数和组合函数之间的唯一区别是 MapReduce 库如何处理函数的输出。缩减函数的输出被写入最终输出文件。组合器函数的输出被写入一个中间文件，该文件将被发送到一个简化任务。

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

部分合并显著加快了某些类别的 MapReduce 操作。附录 A 包含一个使用组合器的示例。

4.4 Input and Output Types

4.4 输入和输出类型

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

MapReduce 库支持读取几种不同格式的输入数据。例如，“文本”

142

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 mean-ingful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits oc-cur only at line boundaries).Users can add support for a new input type by providing an implementation of a sim-ple reader interface, though most users just use one of a small number of predefined input types.

模式输入将每一行视为键/值对:键是文件中的偏移量，值是行的内容。另一种常见的支持格式存储按键排序的键/值对序列。每种输入类型的实现都知道如何将其自身分割成有意义的范围，以便作为单独的地图任务进行处理(例如，文本模式的范围分割确保范围分割仅在线边界处进行)。用户可以通过提供一个简单的阅读器接口的实现来增加对新输入类型的支持，尽管大多数用户只使用少量预定义输入类型中的一种。

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 struc-tures mapped in memory.

读取器不一定需要提供从文件中读取的数据。例如，很容易定义一个从数据库或从内存中映射的数据结构中读取记录的读取器。

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.

以类似的方式，我们支持一组输出类型来产生不同格式的数据，并且用户代码很容易添加对新输出类型的支持。

4.5 Side-effects

4.5 副作用

In some cases, users of MapReduce have found it con-venient 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 tempo-rary file and atomically renames this file once it has been fully generated.

在某些情况下，MapReduce 的用户发现从他们的地图和/或 Reduce 操作符中生成辅助文件作为附加输出很方便。我们依靠应用程序编写器来使这样的副作用成为原子的和幂等的。通常，应用程序会写入一个节奏文件，并在文件完全生成后自动重命名该文件。

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

我们不支持由单个任务产生的多个输出文件的原子两阶段组合。因此，产生具有跨文件一致性要求的多个输出文件的任务应该是确定性的。这种限制在实践中从来都不是问题。

4.6 Skipping Bad Records

4.6 跳过不良记录

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 unavail-able.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 execu-tion where the MapReduce library detects which records cause deterministic crashes and skips these records in or-der to make forward progress.

有时，用户代码中存在错误，导致映射或缩减函数在某些记录上崩溃。这些错误会阻止 MapReduce 操作的完成。通常的做法是修复错误，但有时这是不可行的；也许这个 bug 是在第三方库中，源代码是不可用的。此外，有时忽略一些记录也是可以接受的，例如在对大型数据集进行统计分析时。我们提供了一种可选的执行模式，在这种模式下，MapReduce 库检测哪些记录会导致决定性的崩溃，并跳过这些记录，以便向前推进。

Each worker process installs a signal handler that catches segmentation violations and bus errors.Before invoking a user Map or Reduce operation, the MapRe-duce library stores the sequence number of the argument in a global variable.If the user code generates a signal,

每个工作进程安装一个信号处理器，捕捉分段冲突和总线错误。在调用用户映射或减少操作之前，映射减少库将参数的序列号存储在全局变量中。如果用户代码产生信号，

the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce mas-ter.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 corre-sponding Map or Reduce task.

信号处理程序向 MapReduce master 发送一个包含序列号的“最后一口气”UDP 数据包。当主机在一个特定记录上看到多个故障时，它指示当它发出相应的映射或缩减任务的下一次重新执行时，应该跳过该记录。

4.7 Local Execution

4.7 本地执行

Debugging problems in Map or Reduce functions can be tricky, since the actual computation happens in a dis-tributed 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 im-plementation 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).

在映射或简化函数中调试问题可能很棘手，因为实际的计算发生在一个分布式系统中，通常在几千台机器上，工作分配决策由主机动态做出。为了便于调试、分析和小规模测试，我们开发了一个 MapReduce 库的替代实现，它在本地机器上顺序执行 MapReduce 操作的所有工作。向用户提供控件，以便计算可以限于特定的地图任务。用户用一个特殊的标志来调用他们的程序，然后可以很容易地使用任何他们认为有用的调试或测试工具(例如 gdb)。

4.8 Status Information

4.8 状态信息

The master runs an internal HTTP server and exports a set of status pages for human consumption.The sta-tus 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 gen-erated by each task.The user can use this data to pre-dict 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 com-putation is much slower than expected.

主服务器运行一个内部的超文本传输协议服务器，并导出一组供人类使用的状态页面。sta-tus 页面显示了计算的进度，如已完成的任务数、正在进行的任务数、输入字节数、中间数据字节数、输出字节数、处理速率等。这些页面还包含指向每个任务生成的标准错误和标准输出文件的链接。用户可以使用这些数据来预测计算需要多长时间，以及是否应该向计算中添加更多的资源。这些页面也可以用来计算什么时候计算比预期的要慢。

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 informa-tion is useful when attempting to diagnose bugs in the user code.

此外，顶级状态页面显示哪些工作人员失败，以及失败时他们正在处理的哪些映射和缩减任务。当试图诊断用户代码中的错误时，此信息很有用。

4.9 Counters

4.9 计数器

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.

MapReduce 库提供了一个计数器工具来计算各种事件的发生次数。例如，用户代码可能希望计算已处理的单词总数或已索引的德语文档数等。

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:

为了使用这个工具，用户代码创建一个命名的计数器对象，然后在映射和/或减少函数中适当地增加计数器。例如:

143

```
Counter* uppercase;

计数器*大写；

uppercase = GetCounter("uppercase");

大写= GetCounter(“大写”)；

map(String name, String contents): for each word w in contents: if (IsCapitalized(w)):
```

映射(字符串名称，字符串内容):对于内容中的每个单词:

uppercase->Increment();EmitIntermediate(w, "1");

大写->增量()；发射中间体(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 dis-played on the master status page so that a human can watch the progress of the live computation.When aggre-gating 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.)

来自单个工作机的计数器值被周期性地传播到主机(搭载在 ping 响应上)。主服务器从成功的映射和缩减任务中聚合计数器值，并在映射缩减操作完成时将它们返回给用户代码。当前的计数器值也显示在主状态页面上，以便人们可以观察实时计算的进度。当聚合计数器值时，主机消除重复执行同一地图的影响或减少任务以避免重复计数。(重复执行可能是由于我们使用了备份任务以及由于失败而重新执行任务造成的。(

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

一些计数器值由 MapReduce 库自动维护，例如处理的输入键/值对的数量和产生的输出键/值对的数量。

Users have found the counter facility useful for san-ity 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 pro-cessed, or that the fraction of German documents pro-cessed is within some tolerable fraction of the total num-ber of documents processed.

用户发现计数器工具对检查 MapReduce 操作的行为很有用。例如，在一些 MapReduce 操作中，用户代码可能希望确保生成的输出对的数量正好等于处理的输入对的数量，或者处理的德语文档的比例在处理的文档总数的某个容许比例内。

5 Performance

5 性能

In this section we measure the performance of MapRe-duce on two computations running on a large cluster of machines.One computation searches through approxi-mately one terabyte of data looking for a particular pat-tern.The other computation sorts approximately one ter-abyte of data.

在这一节中，我们测量了在大型计算机集群上运行的两个计算的 MapReduce 性能。一次计算在大约 1tb 的数据中搜索一个特定的模式。另一种计算方法是对大约 1/3 的数据进行排序。

These two programs are representative of a large sub-set of the real programs written by users of MapReduce - one class of programs shuffles data from one representa-tion to another, and another class extracts a small amount of interesting data from a large data set.

这两个程序代表了由 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混排到另一个表示，另一类程序从一个大数据集中提取少量有趣的数据。

5.1 Cluster Configuration

5.1 集群配置

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

所有的程序都是在一个由大约 1800 台机器组成的集群上执行的。每台机器都有两个支持超线程技术的 2GHz 英特尔至强处理器、4GB 内存和两个 160GB 集成开发环境

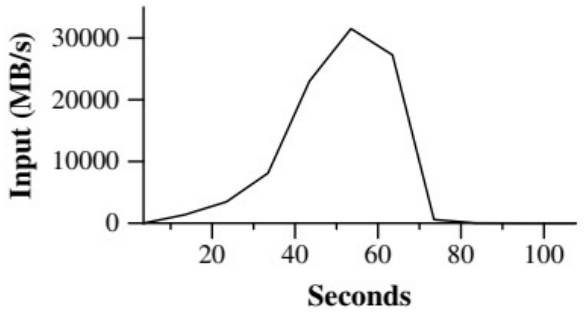


Figure 2: Data transfer rate over time

图 2:一段时间内的数据传输率

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 band-width 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 mil-lisecond.

磁盘和千兆以太网链路。这些机器被安排在一个两级树形交换网络中，其根端的总带宽约为 100-200 Gbps。所有的机器都在同一个托管设施中，因此任何一对机器之间的往返时间都不到一毫秒。

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.

在 4GB 的内存中，大约有 1-1.5GB 被集群上运行的其他任务保留。这些程序是在一个周末的下午执行的，那时中央处理器、磁盘和网络大部分都是空闲的。

5.2 Grep

5.2 Grep

The grep program scans through 10100-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 en-tire output is placed in one file (R = 1).

grep 程序扫描 10100 字节的记录，寻找相对罕见的三字符模式(该模式出现在 92337 条记录中)。输入被分成大约 64MB 的块(M = 15000)，轮胎内输出被放在一个文件中(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 over-head.The overhead is due to the propagation of the pro-gram 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.

图 2 显示了计算随时间的进展。Y 轴显示输入数据的扫描速率。随着越来越多的机器被分配到这个 MapReduce 计算中，速度逐渐加快，当分配了 1764 名工作人员时，速度达到峰值，超过 30 GB/s。随着地图任务的完成，速度开始下降，并在大约 80 秒后达到零。整个计算从开始到结束大约需要 150 秒。这包括大约一分钟的过度启动。开销是由于程序传播到所有工作机器，以及延迟与 GFS 交互以打开 1000 个输入文件的集合并获得局部优化所需的信息。

5.3 Sort

5.3 分类

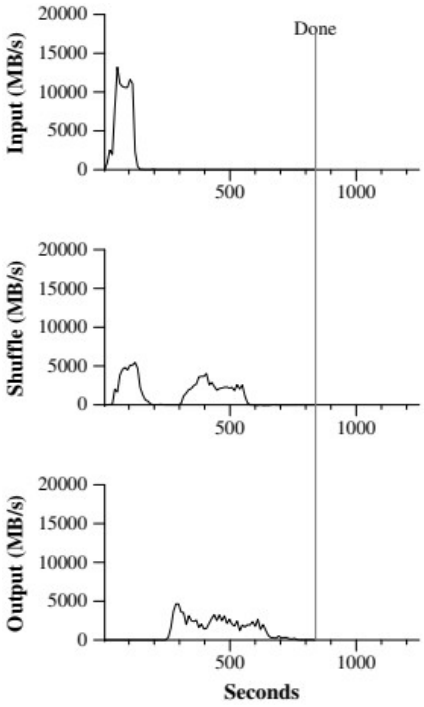
The sort program sorts 10100-byte records (approxi-mately 1 terabyte of data).This program is modeled after the TeraSort benchmark [10].

排序程序对 10100 字节的记录(大约 1tb 的数据)进行排序。该程序是以 TeraSort 基准[10]为模型的。

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

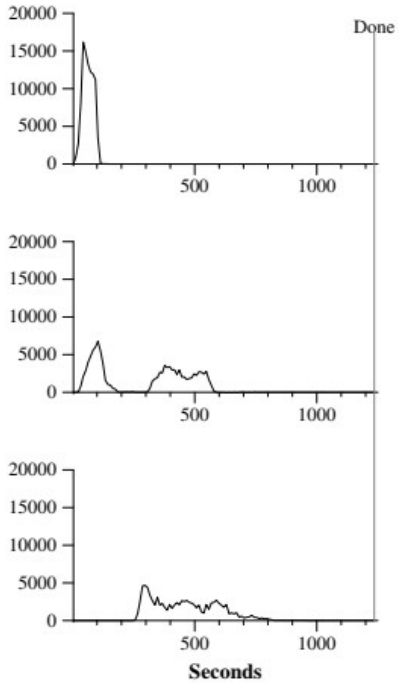
排序程序由少于 50 行的用户代码组成。三行映射函数从文本行中提取一个 10 字节的排序键，并发出该键和

144



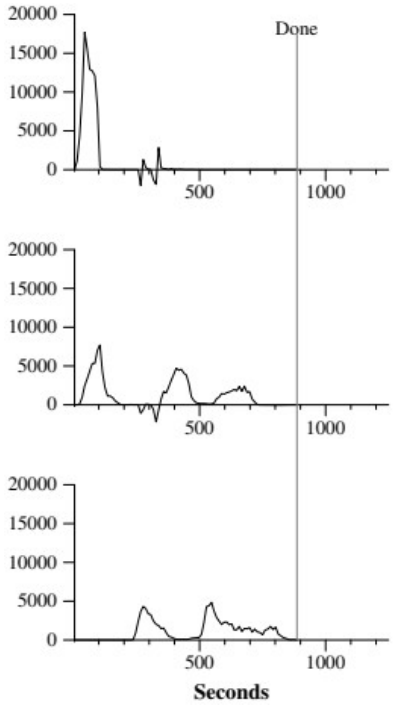
(a) Normal execution

正常执行



(b) No backup tasks

没有备份任务



(c) 200 tasks killed

(c) 200 项任务被取消

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

图 3:不同排序程序执行时间的数据传输率

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 un-changed 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).

作为中间键/值对的原始文本行。我们使用了一个内置的身份函数作为减少操作符。此函数将未更改的中间键/值对作为输出键/值对传递。最终的排序输出被写入一组双向复制的 GFS 文件(即，2tb 的字节被写入作为程序的输出)。

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 ini-tial bytes of the key to segregate it into one of R pieces.

像以前一样，输入数据被分割成 64MB 的片段(M = 15000)。我们将排序后的输出分成 4000 个文件。分区函数使用密钥的初始字节将它分成 R 个部分。

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.

我们针对这个基准测试的分区函数内置了密钥分发的知识。在一般的排序程序中，我们会添加一个预通过 MapReduce 操作，该操作将收集关键字的样本，并使用样本关键字的分布来计算最终排序通过的分割点。

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 be-fore 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 in-termediate output to their local disks.The corresponding intermediate output for grep had negligible size.

图 3 (a)显示了排序程序的正常执行过程。左上角的图表显示了读取输入的速率。速率峰值约为 13 GB/s，并很快消失，因为所有地图任务都在 200 秒前完成。请注意，输入速率小于 grep。这是因为排序映射任务花费大约一半的时间和输入/输出带宽将中间输出写入本地磁盘。grep 的相应中间输出的大小可以忽略不计。

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the re-duce tasks.This shuffling starts as soon as the first map task completes.The first hump in the graph is for

左中图显示了数据通过网络从地图任务发送到减少任务的速率。第一个地图任务一完成，洗牌就开始。图表中的第一个驼峰是为

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.

第一批约 1700 个缩减任务(整个 MapReduce 分配了约 1700 台机器，每台机器一次最多执行一个缩减任务)。计算进行了大约 300 秒，第一批缩减任务中的一些完成了，我们开始为剩余的缩减任务洗牌。所有的洗牌都是在计算过程的 600 秒内完成的。

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 pe-riod and the start of the writing period because the ma-chines 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].

左下角的图表显示了 reduce 任务将排序后的数据写入最终输出文件的速率。在第一次洗牌周期的结束和写入周期的开始之间有一个延迟，因为机器正忙于对中间数据进行分类。写入会以大约 2-4 GB/s 的速度持续一段时间。所有的写操作都在大约 850 秒内完成。包括启动开销，整个计算需要 891 秒。这与目前报道的 TeraSort 基准测试 1057 秒的最佳结果相似[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 avail-ability reasons).We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system.Network bandwidth re-quirements for writing data would be reduced if the un-derlying file system used erasure coding [14] rather than replication.

需要注意的几件事:由于我们的局部优化，输入速率高于混洗速率和输出速率——大多数数据是从本地磁盘读取的，绕过了相对带宽受限的网络。混洗速率高于输出速率，因为输出阶段写入了已排序数据的两个副本(出于可靠性和可用性的原因，我们制作了输出的两个副本)。我们编写两个副本，因为这是底层文件系统提供的可靠性和可用性机制。如果当前文件系统使用擦除编码[14]而不是复制，写入数据所需的网络带宽将会减少。

5.4 Effect of Backup Tasks

5.4 备份任务的影响

In Figure 3 (b), we show an execution of the sort pro-gram 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 fin-ish until 300 seconds later.The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

在图 3 (b)中，我们显示了禁用备份任务的排序程序的执行。执行流程类似于图 3 (a)所示的流程，只是有一个很长的尾部，几乎没有任何写活动发生。960 秒后，除了 5 个缩减任务外，所有任务都完成了。然而，这最后几个掉队者直到 300 秒后才发现。整个计算耗时 1283 秒，耗时增加了 44%。

5.5 Machine Failures

5.5 机器故障

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 pro-cesses were killed, the machines were still functioning properly).

在图 3 (c)中，我们展示了一个排序程序的执行，在这个程序中，我们故意在计算的几分钟内杀死了 1746 个工作进程中的 200 个。底层集群调度器立即在这些机器上重新启动新的工作进程(因为只有进程被终止，机器仍然正常运行)。

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 fin-ishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

工人死亡显示为负输入率，因为一些先前完成的地图工作消失了(因为相应的地图工人被杀死了)并且需要重做。地图工作的重新执行相对较快。整个计算在 933 秒内完成，包括启动开销(只比正常执行时间增加了 5%)。

6 Experience

6 经验

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 li-brary has been for the kinds of problems we work on.It has been used across a wide range of domains within Google, including:

我们在 2003 年 2 月编写了 MapReduce 库的第一个版本，并在 2003 年 8 月对其进行了重大改进，包括局部优化、跨工作机的任务执行的动态负载平衡等。从那时起，我们对 MapReduce li-brary 在我们所研究的问题上的广泛应用感到惊喜。它已经在谷歌内部广泛的领域使用，包括:

• large-scale machine learning problems,

大规模机器学习问题，

• clustering problems for the Google News and Froogle products,

谷歌新闻和 Froogle 产品的聚类问题，

• extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),

提取用于生成热门查询报告的数据(如谷歌时代精神)，

• extraction of properties of web pages for new exper-iments and products (e.g. extraction of geographi-cal locations from a large corpus of web pages for localized search), and

提取新实验和产品的网页属性(例如，从用于本地化搜索的大量网页中提取地理位置)，以及

• large-scale graph computations.

大规模图形计算。

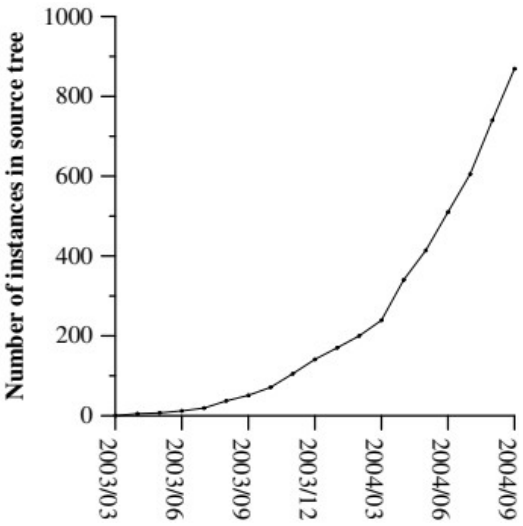


Figure 4: MapReduce instances over time

图 4:随着时间的推移，MapReduce 实例

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

工作数量 29，423 平均作业完成时间 634 秒机器天数用了 79，186 天
输入数据读取 3，288 TB 中间数据产生了 758 TB 输出数据写入 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 每个作业的平均减少任务数 55
唯一地图实现 395 唯一缩减实现 269 唯一地图/缩减组合 426

Table 1: MapReduce jobs run in August 2004

表 1: MapReduce 作业在 2004 年 8 月运行

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 be-cause 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.

图 4 显示了随着时间的推移，检入我们的主要源代码管理系统的独立 MapReduce 程序数量的显著增长，从 2003 年初的 0 增加到 2004 年 9 月底的近 900 个独立实例。MapReduce 非常成功，因为它使得编写一个简单的程序并在半个小时内在一千台机器上高效运行成为可能，大大加快了开发和原型制作周期。此外，它允许没有分布式和/或并行系统经验的程序员轻松地开发大量资源。

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.

在每个作业结束时，MapReduce 库会记录作业使用的计算资源的统计信息。在表 1 中，我们显示了 2004 年 8 月在谷歌运行的 MapReduce 作业子集的一些统计数据。

6.1 Large-Scale Indexing

6.1 大规模索引

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

到目前为止，MapReduce 最重要的用途之一是完全重写生产索引-

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 ter-abytes 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 ver-sion of the indexing system) has provided several bene-fits:

产生用于谷歌网络搜索服务的数据结构的系统。索引系统将我们的爬行系统检索到的大量文档作为输入，并存储为一组 GFS 文件。这些文件的原始内容是 20 多页的数据。索引过程作为五到十个 MapReduce 操作的序列运行。使用 MapReduce(而不是索引系统先前版本中的临时分布式传递)提供了几个好处:

• 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 approx-imately 700 lines when expressed using MapRe-duce.

索引代码更简单、更小、更容易理解，因为处理容错、分布和并行化的代码隐藏在 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++代码下降到大约 700 行。

• The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them to-gether 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.

MapReduce 库的性能非常好，我们可以将概念上不相关的计算分开，而不是将它们混合在一起，以避免额外的数据传递。这使得改变索引过程变得容易。例如，在我们的旧索引系统中，一个花了几个月时间进行的更改只花了几天时间就在新系统中实现了。

• 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 MapRe-duce library without operator intervention.Further-more, it is easy to improve the performance of the indexing process by adding new machines to the in-dexing cluster.

索引过程变得更加容易操作，因为大多数由机器故障、机器速度慢和网络故障引起的问题都是由 MapReduce 库自动处理的，无需操作员干预。此外，通过向索引集群添加新的机器，可以很容易地提高索引过程的性能。

7 Related Work

7 相关工作

Many systems have provided restricted programming models and used the restrictions to parallelize the com-putation automatically.For example, an associative func-tion 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 compu-tations.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.

许多系统都提供了受限的编程模型，并使用这些限制来自动并行计算。例如，使用并行前缀计算，可以在 N 个处理器上对 N 个元素数组的所有前缀进行对数 N 次运算[6，9，13]。基于我们对大型现实世界计算机的经验，MapReduce 可以被认为是这些模型的简化和提炼。更重要的是，我们提供了可扩展到数千个处理器的容错实现。相比之下，大多数并行处理系统只在较小的规模上实现，把处理机器故障的细节留给程序员。

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that

批量同步编程[17]和一些 MPI 原语[11]提供了更高级别的抽象

make it easier for programmers to write parallel pro-grams.A key difference between these systems and MapReduce is that MapReduce exploits a restricted pro-gramming model to parallelize the user program auto-matically and to provide transparent fault-tolerance.

让程序员更容易编写并行程序。这些系统和 MapReduce 的一个主要区别是，MapReduce 利用一个受限的编程模型自动并行化用户程序，并提供透明的容错。

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where compu-tation 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.

我们的局部性优化从主动磁盘[12，15]等技术中获得灵感，在这些技术中，计算被推入靠近本地磁盘的处理单元，以减少通过输入/输出子系统或网络发送的数据量。我们运行在有少量磁盘直接连接的商用处理器上，而不是直接运行在磁盘控制器处理器上，但是一般方法是相似的。

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte Sys-tem [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 in-stances of this problem with our mechanism for skipping bad records.

我们的备份任务机制类似于夏洛特系统[3]中采用的紧急调度机制。简单的紧急调度的缺点之一是，如果一个给定的任务导致重复失败，整个计算就无法完成。我们用跳过不良记录的机制解决了这个问题的一些情况。

The MapReduce implementation relies on an in-house cluster management system that is responsible for dis-tributing 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].

MapReduce 实现依赖于内部集群管理系统，该系统负责在大量共享机器上分配和运行用户任务。虽然不是本文的重点，但集群管理系统在精神上与其他系统(如 Condor [16])相似。

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 appli-cable.

作为 MapReduce 库一部分的排序工具在操作上类似于 NOW-Sort [1]。源机器(地图工作器)对要排序的数据进行分区，并将其发送给其中一个简化工作器。每个精简工作人员在本地对其数据进行排序(如果可能的话，在内存中)。当然，现在排序没有用户可定义的映射和缩减功能，这使得我们的库可以广泛应用。

River [2] provides a programming model where pro-cesses 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.MapRe-duce has a different approach.By restricting the pro-gramming 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).

River [2]提供了一个编程模型，其中进程通过分布式队列发送数据来相互通信。像 MapReduce 一样，River 系统试图提供良好的平均情况性能，即使存在由异构硬件或系统扰动引入的不均匀性。River 通过仔细调度磁盘和网络传输来实现这一点，以实现均衡的完成时间。MapRe-duce 有不同的方法。通过限制编程模型，MapReduce 框架能够将问题划分成大量细粒度的任务。这些任务被动态地调度给可用的工作人员，以便更快的工作人员处理更多的任务。受限编程模型还允许我们在工作接近结束时调度任务的冗余执行，这大大减少了在存在非均匀性(例如缓慢或停滞的工人)时的完成时间。

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

BAD-FS [5]的编程模型与 MapReduce 非常不同，与 MapReduce 不同，它的目标是

the execution of jobs across a wide-area network.How-ever, 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 con-gested network links.

通过广域网执行作业。然而，有两个基本的相似之处。(1)两个系统都使用冗余执行从故障造成的数据丢失中恢复。(2)两者都使用位置感知调度来减少通过拥塞的网络链路发送的数据量。

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

TACC [7]是一个设计用来简化高可用性网络服务的系统。像 MapReduce 一样，它依赖于重新执行作为实现容错的机制。

8 Conclusions

8 结论

The MapReduce programming model has been success-fully 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 opti-mization, and load balancing.Second, a large variety of problems are easily expressible as MapReduce com-putations.For example, MapReduce is used for the gen-eration of data for Google's production web search ser-vice, for sorting, for data mining, for machine learning, and many other systems.Third, we have developed an implementation of MapReduce that scales to large clus-ters of machines comprising thousands of machines.The implementation makes efficient use of these machine re-sources and therefore is suitable for use on many of the large computational problems encountered at Google.

MapReduce 编程模型已经在谷歌成功地用于许多不同的目的。我们将这一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、局部优化和负载平衡的细节。其次，大量的问题很容易用 MapReduce 计算来表达。例如，MapReduce 用于为谷歌的产品网络搜索服务生成数据，用于排序、数据挖掘、机器学习和许多其他系统。第三，我们开发了一个 MapReduce 的实现，它可以扩展到包含数千台机器的大型计算机集群。该实现有效地利用了这些机器资源，因此适用于在谷歌遇到的许多大型计算问题。

We have learned several things from this work.First, restricting the programming model makes it easy to par-allelize 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 al-lows 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 ma-chine failures and data loss.

我们从这项工作中学到了一些东西。首先，限制编程模型使并行和分布计算变得容易，并使这种计算容错。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旨在减少通过网络发送的数据量:局部优化允许我们从本地磁盘读取数据，而将中间数据的单个副本写入本地磁盘则节省了网络带宽。第三，冗余执行可以用来减少慢速机器的影响，并处理机器故障和数据丢失。

Acknowledgements

承认

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a num-ber of new features based on his experience with using MapReduce and other people's suggestions for enhance-ments.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,

乔希·莱文伯格根据他使用 MapReduce 的经验和其他人对增强的建议，用许多新的特性来修改和扩展用户级 MapReduce 应用编程接口。MapReduce 从谷歌文件系统中读取输入，并将输出写入其中[8]。我们要感谢穆希特·阿伦、霍华德·戈比夫、马库斯·古施克，

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 Leven-berg, Sharon Perl, Rob Pike, and Debby Wallach pro-vided helpful comments on earlier drafts of this pa-per.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 or-ganization for providing helpful feedback, suggestions, and bug reports.

大卫·克莱默、梁信德和乔希·雷石东，感谢他们在开发 GFS 方面所做的工作。我们还要感谢珀西·梁(Percy Liang)和奥尔坎·瑟奇诺格鲁(Olcan Sercinoglu)在开发 MapReduce 使用的集群管理系统方面所做的工作。迈克·布伦斯、威尔逊·谢、乔希·利文-伯格、莎伦·佩尔、罗布·派克和王思然·瓦拉克对本报告的早期草稿提供了有益的评论。匿名的 OSDI 评论者和我们的牧羊人埃里克·布鲁尔提供了许多有用的建议，指出了论文可以改进的地方。最后，我们感谢谷歌工程组织中的所有 MapReduce 用户，感谢他们提供了有用的反馈、建议和错误报告。

References

参考

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

[1]AnDrea c . Arpaci-Dussiau，Remzi h . Arpaci-Dussiau，David E. Culler，Joseph M. Hellerstein 和 David A. Pat-terson。工作站网络上的高性能排序。1997 年数据管理国际会议记录，亚利桑那州图森，1997 年 5 月。

[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.

[2]雷姆齐·阿尔帕奇-杜塞乌、埃里克·安德森、诺亚·特鲁霍夫、戴维·e·卡勒、约瑟夫·m·赫勒斯坦、戴维·帕特森和凯西·耶里克。利用河流进行集群输入/输出:使快速情况变得常见。《第六届并行和分布式系统输入输出研讨会论文集》，第 10-22 页，佐治亚州亚特兰大，1999 年 5 月。

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

[3]阿拉什·巴拉特洛、穆罕默德·卡拉乌尔、兹维·凯登和彼得·威科夫。夏洛特:网络上的元计算。在 1996 年第九届并行和分布式计算系统国际会议的开幕式上。

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

[4]路易斯·巴罗佐、杰弗里·迪恩和乌尔斯·奥兹勒。网络搜索一个星球:谷歌集群架构。《电气和电子工程师协会微》，23(2):22-28，2003 年 4 月。

[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 Pro-ceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI, March 2004.

[5]约翰·本特、道格拉斯·塞恩、安德烈亚·阿尔帕奇-杜塞乌、伦齐·阿尔帕奇-杜塞乌和米隆·利夫尼。批感知分布式文件系统中的显式控制。2004 年 3 月，在 NSDI 举行的第一届 USENIX 网络系统设计与实现研讨会上。

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

[6]盖伊·布尔洛克。扫描是原始的并行操作。IEEE 计算机事务，C-38(11)，1989 年 11 月。

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

[7]阿曼多·福克斯、史蒂文·格里布尔、亚丁·查瓦特、埃里克·布鲁尔和保罗·高蒂尔。基于集群的可伸缩网络服务。在第 16 届美国计算机学会操作系统原理研讨会会议录，第 78- 91 页，法国圣马洛，1997 年。

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

[8] Sanjay Ghemawat，Howard Gobioff 和 sung-Tak Le-ung。谷歌文件系统。第 19 届操作系统原理研讨会，第 29-43 页，乔治湖，纽约，2003 年。

148

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

[9]s . Gorrack。扫描和其他列表同态的系统有效并行化。1996 年欧洲期刊编辑。并行处理，计算机科学讲义 1124，第 401-408 页。斯普林格-弗拉格，1996 年。

[10] Jim Gray.Sort benchmark home page.

[10]吉姆·格雷。对基准主页进行排序。

http://research.microsoft.com/barc/SortBenchmark/.

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.

[11]威廉·格罗普、尤因·吕斯克和安东尼·斯克杰伦。使用 MPI:带有消息传递接口的可移植并行编程。麻省理工学院出版社，剑桥，麻省，1999。

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satya-narayanan, G. R. Ganger, E. Riedel, and A. Ailamaki.Di-amond: A storage architecture for early discard in inter-active search.In Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference, April 2004.

[12] L. Huston，R. Sukthankar，R. Wickremesinghe，M. Satya-narayanan，G. R. Ganger，E. Riedel 和 A. Ailamaki。双向:一种在交互搜索中早期丢弃的存储结构。在 2004 年 USENIX 文件和存储技术会议记录中，2004 年 4 月。

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

[13]理查德·拉德纳和迈克尔·费希尔。并行前缀计算。美国计算机学会杂志，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.

[14]迈克尔·拉宾。为了安全、负载平衡和容错，有效地分散信息。美国计算机学会杂志，36(2):335-348，1989。

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

[15]埃里克·里德尔、克里斯特斯·法鲁索斯、加斯·吉布森和戴维·纳格尔。用于大规模数据处理的活动磁盘。IEEE 计算机，第 68-74 页，2001 年 6 月。

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny.Distributed computing in practice: The Condor experi-ence.Concurrency and Computation: Practice and Ex-perience, 2004.

[16]道格拉斯·塞恩、托德·坦南鲍姆和米隆·利夫尼。实践中的分布式计算:秃鹰实验。《并发与计算:实践与经验》，2004 年。

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

[17]瓦兰特。并行计算的桥接模型。美国国会通讯，33(8):103-111，1997。

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

[18]吉姆·威利。Spsort:如何快速对万亿字节进行排序。http://alme1.almaden.ibm.com/cs/spsort.pdf.

A Word Frequency

词频

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.

本节包含一个程序，该程序计算命令行上指定的一组输入文件中每个唯一单词的出现次数。

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

#包括“mapreduce/mapreduce.h”

```
// User's map function
```

//用户地图功能

```
class WordCounter : public Mapper { public:
```

公共映射器{公共:

```
virtual void Map(const MapInput& input) { const string& text = input.value();const int n = text.size();for (int i = 0;i < n;) {
```

虚拟空间映射(常量映射输入和输入){常量字符串和文本=输入.值()；const int n = text . size()；for(int I = 0；I < n；){

```
// Skip past leading whitespace while ((i < n) && isspace(text[i]))
```

```
//跳过前导空格((i < n) && isspace(文本[i])

i++;

i++;

// Find word end int start = i;

//查找单词结束点开始点= I;

while ((i < n) && !isspace(text[i])) i++;

而((i < n) &&！ISS space(text[I])(i++);

if (start < i)

如果(开始< I)

Emit(text.substr(start,i-start),"1");

发出(text.substr(start，i-start)，" 1 ");

}

{}

}

{}

};

}；

REGISTER_MAPPER(WordCounter);

寄存器映射器(字计数器);

// User's reduce function class Adder : public Reducer {

//用户的缩减函数类加法器:公共缩减器{

virtual void Reduce(ReduceInput* input) { // Iterate over all entries with the // same key and add the values int64 value = 0;

虚拟无效减少(减少输入*输入){ //用//同一个键迭代所有条目，并添加 int64 值= 0 的值;

while (!input->done()) {

while(！输入->完成()){
```

value += StringToInt(input->value());input->NextValue();

值+=字符串点(输入->值())；输入->下一个值()；

}

{}

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

//发出输入的总和->键()发出(输入字符串(值))；

}

{}

};

}；

REGISTER_REDUCER(Adder);

寄存器_减压器(加法器)；

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

int main(int argc，char** argv) {

ParseCommandLineFlags(argc, argv);

ParseCommandLineFlags(argc，argv)；

MapReduceSpecification spec;

MapReduce 规范规范；

// Store list of input files into "spec" for (int i = 1;i < argc;i++) {

//将输入文件列表存储到(int I = 1；i < argci++) {

MapReduceInput* input = spec.add_input();input->set_format("text");input->set_filepattern(argv[i]);input->set_mapper_class("WordCounter");

MapReduce InPut * InPut = spec . add _ InPut()；输入->set_format(“文本”)；输入-> set _ file pattern(argv[I])；输入->设置映射器类(“计数器”)；

}

{}

// Specify the output files: // /gfs/test/freq-00000-of-00100 // /gfs/test/freq-00001-of-00100 // ...

//指定输出文件:///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");

MapReduce output * out = spec . output()；out-> set _ file base("/GFS/test/freq ")；out-> set _ num _ tasks(100)；out->set_format(“文本”)；out-> set _ reductor _ class(“加法器”)；

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

//可选:在 map //任务中进行部分求和以节省网络带宽->set_combiner_class(“加法器”)；

// 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);

//调整参数:每个任务最多使用 2000 个//机器和 100 兆内存。spec . set _ map _ MB(100)；spec . set _ reduce _ MB(100)；

// Now run it

//现在运行它

MapReduceResult result;

MapReduce 结果；

if (!MapReduce(spec, &result)) abort();

如果(！MapReduce(规范，&结果))中止()；

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

//完成:“结果”结构包含关于计数器、所用时间、所用//机器数量等的信息//。

return 0;

返回 0；

}

{}

149

149