摘要

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来处理kev/value对以生成一组中间kev/value对，以及一个reduce函数来合并与同一中间键相关联的所有中间值。如本文所示，许多现实世界的任务可以在该模型中表达。

在这个功能stvle中编写的程序被自动并行化，并在一个大型的商用机器集群上执行。运行时系统负责对输入数据进行分区，在一组机器上安排程序的执行。处理机器故障。以及管理所需的机器间通信。这使得没有anv经验的程序员可以轻松地利用大型分布式系统的资源。

我们的MapReduce实现运行在一个大型的商用机器集群上，并且具有高度的可伸缩性:一个典型的ManReduce计算在数千台机器上处理许多万亿字节的数据。程序员发现这个系统很容易使用:数百个MapReduce pro grams已经被实现，每天在谷歌的集群上执行超过1000个ManReduce iobs。

介绍

在过去的五年里，作者和谷歌的其他人已经实现了数百种特殊用途的计算，这些计算处理大量的原始数据，比如爬行文档。web请求日志。等。计算各种派生数据，如倒排索引，weh文档图结构的各种表示。每台主机抓取的nages数量的摘要。a中最频繁的查询的集合

一天,等等。大多数这样的计算在概念上是简单的。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。如何将计算并行化、数据分布和故障处理等问题，用大量复杂的代码来处理这些问题，掩盖了原来简单的计算。

为了应对这种复杂性，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，但隐藏了并行化的混乱细节。容错。库中的数据分布和负载平衡。我们的抽象受到了Lisn和许多其他函数语言中存在的map和reduce原语的启发。我们意识到我们的大部分计算涉及应用操作映射到每个逻辑“记录”在我们的输入来计算一组中间kev /值对,然后将减少操作应用于所有共享理智的键的值,为了结合适当的导出数据。我们使用带有用户指定map和reduce操作的功能模型，这允许我们轻松地并行化大型计算，并使用重新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，使自动并行化和大规模计算的分布，与此接口的实现，以实现高性能的大型集群的商品pc。

第2节描述了基本的编程模型并给出了几个示例。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节描述了我们认为有用的编程模型的几个改进。第5节对我们实现的各种任务进行性能度量。第6节探讨了MapReduce在谷歌中的使用，包括我们使用它作为基础的经验

重写我们的生产索引系统。第7节讨论相关的和未来的工作

2 编程模型

计算需要一组inpur kev/值对。并生成一个由输出键/值对组成的sct。MapReduce库的用户用两个函数表示计算:Map和Reduce。

Map由用户编写，接受一个输入对并生成一组中间kev/值对。ManReduce库将与同一中间键I关联的所有中间值分组在一起，并将它们传递给Reduce函数

Reduce函数。也由用户编写。重读中间kev I和该键的一组值。它将这些值合并在一起，以创建一个可能很小的值集，通常每次Reduce调用都会产生0或1个输出值。中间值通过一个迭代器提供给用户的reduce函数。这允许我们处理那些大到无法装入内存的值列表。

2.1的例子

考虑计算一个大型文档集合中每个单词的oc当前数的问题。用户将编写类似下面的伪编码器的代码

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函数发出cach word和相关的出现次数计数(在这个简单的cxample中，iust为1’)。

reduce函数将针对特定单词发出的所有计数相加。

此外。用户编写代码，用输入和输出文件的名称以及可选的调优参数填充maprduce snecification实例。然后，用户调用ManReduce函数，将规范参数传递给它。用户的代码链接在一起

MapReducc libay 在 C++） 中C++。阿普库迪克斯 A

包含本示例的完整程序文本。

2类型

虽然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map和reduce函数有关联的类型:

map (k1,v1) → list(k2,v2)

reduce (k2,list(v2)) → list(v2)

L.c . .输入kevs和值与输出kevs和值来自不同的域。此外。

中间的kevs和值与输出键和值来自相同的域。

我们的c++ t实现将字符串传递给用户定义函数，并将字符串和anpronriate类型之间的转换留给用户代码。

2.3更多的例子

这里有几个有趣的程序的简单例子，可以用MapReduce计算来表达。

分布式Gren: map函数会发出一条与所提供的模式匹配的线。reduce函数是一个识别函数，它只需要将所提供的中间数据复制到输出端。

URL访问频率计数:map函数处理web页面请求和输出的日志(URL, 1)， reduce函数将同一URL的所有值相加，发出(URL, total Count)对。

反向web链接图:map函数输出(目标、源:对在名为源的页面中找到的目标URL的每个链接)。reduce函数将与给定目标URL关联的所有源URL列表连接起来，并发出这一对:

目标(。目前居住(source))

每个主机的术语向量:术语向量将文档或一组文档中出现的最重要的单词汇总为一个列表(word)。frequencu)对。map函数发出(主机名)。每个输入文档(其中主机名从文档的URL中提取)的term vector对。reduce函数传递给定主机的每个文档的所有术语向量，然后将这些术语向量相加，去掉前缀，最后得到一个(主机名，术语向量)对。

在这儿，土地标记

ment，并发出(word、文档ID)对序列。reduce函数接受给定单词的所有对，对相应的文档ID进行排序，并发出(单词、列表(文档ID))对。所有输出对的集合形成一个简单的反向索引。很容易增加这个计算来跟踪单词的位置。

分布式排序:map函数从每个记录中提取密钥，并发出(密钥，记录)对。reduce函数不加修改地发出所有对。这种计算依赖于4.1节中描述的分区工具和4.2节中描述的排序属性。

3实现

MapReduce接口有许多不同的实现。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联网机器集合。

本节描述了一个针对广泛应用于谷歌的计算环境的实现:

用交换式以太网[4]连接的大型商品pc集群。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)商品网络硬件被使用-通常是100兆位/秒或1千位/秒在机器水平上，但在总体平分带宽平均相当少。

(3)一个集群由成百上千个mahines组成，因此机器故障很常见。

(4)存储由便宜的IDE磁盘直接提供给各个machires。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)用户向调度系统提交作业。每个作业由一组任务组成，并由调度器映射到集群中的一组可用机器上。

3.1执行概述

通过自动分区输入数据，映射调用分布在多台机器上

变成一组M叉。输入分割可以是pro。

被不同的机器并联。Reduce调用是通过使用一个分区函数将intermediatc kev空间分成R个部分来分配的。

分区的数量(R)和分区函数由用户指定。

图1显示了在我们的实现中ManReduce操作的总体流程。当用户程序调用MapReduce函数时。接下来会发生一系列的动作(图1中编号的标签对应于下面列表中的数字):

1. MapReduce库在用户程序firs将输入文件分成M块，一般是10兆字节到64兆字节(MB)每块(由用户通过一个可选参数控制)。然后在一组机器上启动该程序的多个副本

这个程序的副本中有一份是特别的——主程序。其余的是被主人分配工作的工人。有M个map任务和R个reduce任务需要分配。主程序挑选空闲的工人，并为每个工人分配一个map任务或reduce任务

3.分配map任务的工作人员读取相应的输入分割的内容。它解析:kev/值对输出的输入数据并传递每一个

对用户定义的映射函数。Map函数产生的interme diate kev/value对被缓冲在内存中

周期性地，缓冲对被写入loca磁盘，分区函数将其划分为R区域。本地磁盘上这些缓冲对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reduce工作程序。

5. 当一个reduce工作程序被maste通知这些位置时，它使用远程过程调用:从map工作程序的本地磁盘读取缓存的数据。当reduce worker读取了所有的间接数据时，它将按中间键对其进行排序:以便将所有出现的相同键分组在一起。之所以需要排序，是因为通常有许多不同的键映射到同一个reduce任务。如果中间数据量太大而无法装入内存，则使用外部排序。

reduce worker迭代已排序的intermediate数据，对于每一个惟一的中间值，它将键和相应的中间值集传递给用户的reduce函数。Reduce函数的输出被附加到这个Reduce分区的最终输出文件中。

7. 当所有map任务和reduce任务完成时，主程序唤醒用户程序。

此时，用户程序中的MapReduce调用返回到用户oode。

在成功完成之后。mapreduce执行的cutput在R输出文件中可用(每个reduce任务一个，文件名由ser指定)。

Tvpically上，用户不需要将这些R cutput文件合并到一个文件中——用户经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分割成多个文件的输入

3.2主数据结构

主程序保存几个数据结构。对于每个map任务和reduce任务，它存储状态(idle)。正在进行的或已完成的)和工作机器的标识(对于非空闲任务)。

master是将中间文件区域的位置从mar任务传播到reduce任务的管道。因此。为cach完成了人的任务。

主文件存储由map任务生成的R中间文件区域的位置和大小。到这个位置和大小的更新信息在映射任务合并时被接收，这些信息被逐步地混合到有正在进行的reduce任务的工人那里。

3.3容错

因为MapReduce库被设计用来帮助处理使用成百上千台机器的大量数据。库必须优雅地容忍机器故障。

工人失败

主处理器定期ping每一个worker。如果在一定的时间内没有收到worker的响应，则master将该worker标记为failed。任何由worker完成的map任务都被重置回它们的初始空闲状态，因此可以在其他worker上进行调度。类似地，在一个失败的worker上进行中的任何map任务或reduce任务也会被重置为空闲，并有资格重新调度。

完成的人工任务将在失败时重新执行。

因为它们的输出存储在故障机器的本地磁盘上，因此无法访问。完成的reduce任务不需要重新执行，因为它们的输出存储在一个全局文件系统中。

map任务首先由worker a执行，然后由worker B执行(因为a失败)

执行reduce任务的工人被通知re。

执行。任何还没有从worker A读取数据的reduce任务都将从worker B读取数据。

MapReduce对大规模工人失败具有弹性。

例如在一个MapReduce操作期间。运行集群上的网络维护导致80台机器在数分钟内无法到达，ManReduce主机简单地重新执行无法到达的工作机器所做的工作，并继续向前推进，最终完成ManReduce作业。

主服务器失败

对于上面描述的主数据模型，让主写入定期检查点是很容易的。如果主任务终止，则可以从最后一个检查点状态启动一个新副本。然而，假设只有一个主服务器。它的失败是不像elv:因此我们当前的实现中止ManReduce计算如果主失败，客户端可以检查这个条件，并重试MapReduce操作如果他们愿意

出现故障时的语义

当用户提供的man和reduce onerators是它们输入值的确定性函数时。我们的分布式实现产生的输出与整个程序的无故障顺序执行产生的输出相同。

我们依赖于map的原子提交和reduce任务输出来实现这个属性。每个正在进行的任务将其输出写入临时文件。reduce任务生成这样一个文件。map任务生成R个这样的文件(每个reduce任务一个)，当一个man任务完成时，工作人员向master发送一条消息，并在消息中包含R个临时文件的名称。如果主服务器收到一个已经完成的人工任务的完成消息。它忽略消息。否则。

它在主数据结构中记录R文件的名称。

当一个reduce任务完成时，reduce工作人员会自动将其临时的cutput文件重命名为最终的输出文件。如果相同的减少在多台机器上执行任务,将执行多个重命名调用相同的最后我们依赖于原子outnut文件重命名操作underlving提供的文件系统来保证最终的文件系统状态产生的数据只包含一个减少任务的执行。

我们庞大的地图和减少的单位是决定性的。事实上，我们的语义在这种情况下与顺序执行是等价的

程序员很容易推断出他们的程序是什么。

havior，当map和/或reduce操作符是不确定的。我们提供了较弱但仍然合理的语义。在存在非确定性因子的情况下。

一个特定的reduce任务R的运行速度，等于由非确定性程序的本质解产生的R的输出。然而，对于不同的reduce任务R的输出，可能对应于由非确定性程序的不同顺序执行所产生的Ro的输出。

考虑映射任务M和减少任务Bi和Ra。

L.et e(R)是R的执行，即被执行(therd就是这样的执行之一)。语义较弱是因为e(R) mav读取了一次M执行产生的输出，而e(R)可能读取了另一次M执行产生的输出。

3.4

位置

网络带宽在我们的计算环境中是一个相对稀缺的资源。我们利用这样一个事实，即输入数据(由GFS[81管理)存储在组成集群的机器的本地磁盘上，从而节省网络带宽。GFS将每个文件划分为64 MB的块，并在不同的机器上存储每个块的几个副本(tvpically 3个副本)。MapReduce主程序考虑到输入文件的locarion信息，并尝试在一台包含相应输入数据副本的机器上调度一个map任务，如果失败，它尝试在该任务的输入数据副本附近调度一个map任务(例如。在与包含数据的机器在同一网络交换机上的工作机器上)。当在集群中相当一部分的工作上运行大型MapReduce操作时，大多数输入数据都是在本地读取的，不消耗网络带宽

3.5任务粒度

我们将map阶段划分为M个picces，将reduce阶段划分为R个片段，如上所述。在理想的情况下。M和R应该比工人机器的数量大得多。让每个worker执行许多不同的任务可以改善动态负载平衡。当一个worker发生故障时，还会加快恢复速度:它已经完成的许多映射任务可以分散到所有其他worker机器上

有实用范围多大M和Rcan在我们的实现中,由于主必须O (M + R)调度决策和O (M \* R)状态保存在存储器中如上所述,(然而内存使用量很小的常数因素:O (MR)的状态由大约一个字节的数据/任务/减少任务对人。)

此外。R常常受到用户的限制，因为每个reduce任务的eutput最终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，这样每个任务的输入数据大约为16mb到64mb(这样上面描述的局部性优化是最有效的)，我们使R成为我们要使用的工作机器数量的一个小倍数。我们经常用M=200.000和R=5.000来执行MapReduce计算，使用的是2000台工人机器

3.6

备份任务

延长MapReduce操作所花费的总时间的一个常见原因是“掉队者”:如果一台机器在完成最后几个map或reduce任务时花费了异常长的时间，那么出现掉队者的原因有很多。前女友。

充足的。有坏磁盘的机器可能会经历频繁的可纠正的错误，使其读取性能从30mb /s降低到1mb /s。集群调度系统。

由于CPU、内存、本地磁盘或网络带宽的竞争，tem可能已经在机器上调度了其他任务，从而导致它执行MapReduce代码的速度变慢。我们最近遇到的一个问题是机器初始化代码中的一个错误导致了proces。

或将缓存禁用:受影响机器上的计算速度减慢超过100倍。

我们有一个普遍的机制来缓解掉队者的问题。MapReduce操作接近完成时,主人安排备份执行其余的正在进行的任务,任务被标记为完成时主或backun执行完成后,我们有调整这个机制,让它通常增加使用的计算资源操作不超过几个百分点我们发现,这大大减少了时间来完成大型MapReduce操作。例如，当备份任务nechanism被禁用时，5.3节中描述的排序程序要多花44%的时间来完成

4

细化

尽管简单地编写Map和Reduce函数所提供的基本功能足以满足大多数需求，但我们发现一些有用的扩展。这些将在本节中进行描述

4.1分区功能

MapReduce的用户指定reduce任务/输出文件的数量(R)

中间键。提供了一个使用哈希的默认分区函数。“散列(键)mod R”这往往会导致相当均衡的分区。但是，在某些情况下，根据键的其他函数对数据进行分区是有用的。为例。有时输出键是url，我们希望单个主机的所有条目都出现在同一个输出文件中。为了支持这样的情况，MapReduce库的用户可以提供一个特殊的分区函数。例如，使用“hash(Hostname(urlkeu)) mod R”作为分区函数会导致来自同一主机的所有url最终出现在同一输出文件中

4.2要求担保

我们保证在一个给定的分区,中间键/值对处理增加关键这种排序保证便于gererate排序输出文件每个分区,这是有用的,当输出文件格式nceds支持高效的随机访问查找键,或用户输出方便找到它的数据排序

4.3合路器Functior

在某些情况下，每个map任务产生的中间键有大量重复，并且userspecified Reduce函数是可交换的和关联的。一个很好的例子是2.1节中的单词计数示例。由于单词freduencies倾向于遵循Zipf分发，所以每个map任务将生成表单<the, 1>的数百条或数千条记录。所有这些计数将通过网络发送到一个reduce任务，然后由reduce函数相加产生一个数字。我们允许用户soecify一个可选的组合函数，在数据通过网络发送之前完成部分合并。

组合函数在执行映射任务的每台机器上执行。通常使用相同的代码来实现组合器和reduce函数。reduce函数和合并函数之间的唯一区别是MapReduce库如何处理函数的oulpul。reluce函数的oulut被写入最终的输出文件。组合器函数的输出被写入中间文件，该文件将被发送给reduce任务

部分结合显著地加速了某些类的MapReduce操作。附录A包含一个使用组合器的例子。

4.4

mapreduce库支持以下几种不同的格式读取输入数据。列如，“文本”模式输入将每一行视为一个键/值对：键是文件中的偏移量，值是行的内容。另一种常用的支持格式存储按键排序的键/值对序列。每个输入类型实现都知道如何将自身拆分成有意义的范围，以便作为单独的映射任务（例如文本）进行处理mode的范围分割确保了仅在行边界处进行范围拆分）。用户可以通过提供简单读取器接口的实现来添加对新输入类型的支持，尽管大多数用户只使用少数预定义输入类型中的一种。

读卡器不一定需要提供从文件中读取的数据。例如，很容易定义一个读卡器，它可以从数据库或内存中映射的数据结构中读取记录。

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

4.5 副作用

在某些情况下，MapReduce的用户发现有必要生成辅助文件作为他们的map和/或reduce运算符的附加输出。我们依赖于应用程序编写器使这些副作用原子化和幂等。通常，应用程序写入一个临时文件，并在文件完全完成后自动重命名该文件生成。

我们不支持由单个任务生成的多个输出文件的原子两阶段组件。因此，生成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的，这种限制在实践中从来不是一个问题。

4.6 跳过不良记录

有时用户代码中存在导致映射的错误或者Reduce函数在某些记录上发生决定性的崩溃。这样的错误会阻止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；可能错误位于第三方库中，而该库的源代码不可用。此外，有时，忽略一些记录是可以接受的，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库会检测哪些记录会导致确定性崩溃，并跳过这些记录，以便向前推进。

每个工作进程都安装一个信号处理程序，用于捕获分段冲突和总线错误。在调用用户映射或Reduce操作之前，映射还原库将参数的序列号存储在全局变量中。如果用户代码生成信号处理程序向MapReduce mas发送一个包含序列号的“lastgas”UDP数据包。当主机在某个特定记录上发现多个故障时，它指示在下一次重新执行相应的Map或Reduce任务时应跳过该记录。

4.7 本地执行

在Map或Reduce函数中调试问题可能很棘手，因为实际的计算发生在分布式系统中，通常在几千台机器上，由主机动态地做出工作分配决策。为了帮助调试、分析和小规模测试，我们开发了一个MapReduce库的替代实现，它顺序地执行本地机器上MapReduce操作的所有工作。提供了一些控件，使计算可以限制到特定的映射任务。用户可以使用特殊标志调用其程序，然后可以轻松地使用任何调试或测试工具。

4.8 状态信息

主机运行一个内部的HTTP服务器并导出一组状态页供用户使用。状态页显示计算的进度，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。这些页页还包含指向每个任务生成的标准错误和标准输出文件。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。这些页面还可用于计算何时比预期慢得多。

此外，顶层状态页显示哪些工作线程失败，以及在失败时映射和减少正在处理的任务。当尝试诊断用户代码中的错误时，此信息非常有用。

4.9 计数器

MapReduce库提供了一个计数器功能来统计各种事件。例如，用户代码可能需要统计处理的总字数或索引的德语文档数等。

要使用此功能，用户代码将创建一个命名的counter对象，然后在Map和/或reduce函数中适当地增加计数器。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");

来自单个工作计算机的计数器值定期传播到主计算机（ping响应中的附带值）。主程序从成功的map和reduce任务聚合计数器值，并在MapReduce操作完成后将其返回给用户代码。当前的计数器值也显示在主状态页上，这样人类就可以观察实时计算的进度。当聚集计数器值时，主机会消除同一映射的重复执行的影响，或减少任务以避免重复计算。（在使用备份任务和由于失败而重新执行任务时，可能会出现重复执行。）

有些计数器值由MapReduce库自动维护，例如处理的输入键/值对的数量和生成的输出键/值对的数量。

用户发现计数器功能对san检查MapReduce操作的行为非常有用。例如，在某些MapReduce操作中，用户代码可能希望确保生成的输出对数与处理的输入对数完全相等，或者处理的德国文件的比例在处理的文件总数的某个可容忍的部分之内。

5 性能

在本节中，我们将测量在大型计算机集群上运行的两个计算的性能。一个计算在大约1TB的数据中搜索特定模式，另一个计算对大约1/3字节的数据进行排序。

这两个程序代表了MapReduce用户编写的真实程序的一个大的子集合——一类程序将数据从一个表示转移到另一个表示，另一类程序从一个大的数据集中取出少量有趣的数据。

5.1 集群配置

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。每台主机都有两个2GHz的Intel Xeon处理器，支持超线程，4GB内存，两个160GB的IDE磁盘和一个千兆以太网链路。这些机器安装在一个两级树形交换网络中，在根部有大约100-200 Gbps的总带宽。所有的机器都在同一个主机设备中，因此任何一对机器之间的往返时间不到一毫秒。

在4GB内存中，大约有1-1.5GB被运行在群集上的其他任务保留，这些程序是在一个周末的下午执行的，那时CPU、磁盘和网络大多处于空闲状态。

5.2 绿色环保

grep程序扫描1010个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92337个记录中），输入被分成大约64MB的片段（M15000），而en-tire输出则被放在一个文件中（R=1）。

图2显示了一段时间内计算的进度。Y轴显示了扫描输入数据的速率。当分配给此MapReduce计算的机器越来越多时，扫描速度逐渐加快，当分配1764个工人时，扫描速度的峰值超过30GB/s。当map任务完成时，计算速度开始下降，大约80秒到达0。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从头启动。开销是由于程序传播到所有工作机，以及延迟与GFS交互打开1000个输入文件集并获取局部优化所需的信息。

5.3 分类

sort程序对100字节的记录进行排序（大约相当于1万亿字节的数据）。

这个程序是在TeraSort基准测试[10]之后建模的，排序程序由不到50行用户代码组成。三行映射函数从文本行中提取一个10字节的排序键，并将该键和原始文本行作为中间键/值对。我们使用了一个内置的身份函数作为Reduce接线员。这个函数将未更改的中间键/值对作为输出键/值对传递。最后的sortedoutput被写入一组双向复制的GFS文件中（即，2兆字节被写入程序的输出）。

如前所述，输入数据被分成64MB（M=15000），我们将排序后的输出分成4000个文件（R=4000），分区函数使用密钥的初始字节将其分隔成R块之一。

我们在这个基准测试中的分区函数有关于键分布的内置知识，在一个普通的排序程序中，我们将添加一个pre-pass MapReduce操作，该操作将收集一个键的样本，并使用采样的键的分布来计算最终排序过程的拆分点。

图3（a）显示了排序程序的正常执行情况。左上角的图表显示了读取输入的速率。该速率的峰值约为13gb/s，并且由于所有映射任务都在200秒前完成，所以很快就消失了。请注意，输入速率低于grep。这是因为排序映射任务花费了大约一半的时间它们将中间输出写入本地磁盘的时间和I/O和宽度。对应的grep中间输出的大小可以忽略不计。

左中图显示了通过网络将数据从映射任务发送到导入任务的速率。当第一个map任务完成时，这种洗牌就开始了。图中的第一个驼峰是第一批大约1700个reduce任务（整个MapReduce被分配了大约1700台机器，每台机器每次最多执行一个reduce任务）。在计算大约300秒后，第一批reduce任务中的一些完成，我们开始为剩余的reduce任务处理数据。所有的转换都是在600秒左右完成的。

左下角的图表显示了reduce将sorteddata写入最终输出文件的速率任务。那里是第一次洗牌周期结束到写入周期开始之间的延迟，因为主机忙于对中间数据进行排序。写操作会以大约2-4gb/s的速度持续一段时间。所有的书写都在850秒后完成计算。包括启动开销，整个计算需要891秒。这与TeraSort基准测试当前1057秒的最佳报告结果相似[18]。

有几点需要注意的是：由于我们的局部优化，输入速率高于shuffle速率和输出速率–大多数数据是从本地磁盘读取的，并且绕过了我们相对带宽受限的限制网络洗牌率较高因为输出阶段会写入两个已排序数据的副本（出于可靠性和可用性的原因，我们制作两个输出副本）。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。如果不兼容的文件系统使用擦除编码而不是复制，则写入数据的网络带宽要求将降低。

5.4 备份任务的影响

在图3（b）中，我们展示了在禁用备份任务的情况下执行sort程序的过程，其执行流程与图3（a）中所示的类似，只是有一个很长的尾部，几乎没有任何写入活动发生。960秒后，除了5个reduce任务外，其他任务都完成了。然而，最后的几个掉队者直到300秒后才完成。整个计算耗时1283秒，所用时间增加了44%。

5.5 机器故障

在图3（c）中，我们展示了一个sort程序的执行过程，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。底层的集群调度程序立即在这些机器上重新启动了新的工作进程（因为只有进程被终止，机器仍然正常工作）。

工人死亡显示为工人的负输入率，因为以前完成的一些地图工作消失了（因为相应的地图工作人员被杀），需要重新进行。此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（只是增加了比正常执行时间长5%）。

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月对其进行了重大的改进，包括局部性优化、跨工作机执行任务的动态负载平衡等。从那时起，我们惊喜地发现MapReduce库的应用范围之广对于我们正在处理的问题。它已经被广泛应用于谷歌的各个领域，包括：

大规模的学习问题

Google新闻和Froogle产品的聚类问题，

提取用于生成热门查询报告的数据（如Google Zeitgeist），

为新的实验和产品提取网页的属性（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）

随着时间的推移，从2003年初的0到2004年9月下旬的900个独立实例，MapReduce已经成功地登录到我们的主源代码管理系统中。MapReduce非常成功，因为它可以在半小时内编写一个简单的程序并在上千台机器上高效运行，大大加快了开发和原型设计的周期，而且，它允许没有分布式和/或并行系统经验的程序员轻松地利用大量资源。

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，我们展示了2004年8月在Google上运行的MapReduce作业子集的一些统计数据。

6.1 大规模索引

到目前为止，MapReduce最重要的用途之一是对生产索引系统进行了彻底重写，该系统生成用于google web搜索服务的数据结构。索引系统输入了一大组文档，这些文档已被我们的爬行系统检索到，存储为一组GFS文件。这些文件的原始内容是20多个字节的数据。索引过程以5到10个MapReduce操作的顺序运行。使用MapReduce（而不是以前版本的索引系统中的即席分布式过程）提供了几个好处：

索引代码更简单、更小、更易于理解，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 .

MapReduce库的性能足够好，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。这使得更改索引过程变得很容易。例如，在我们的旧索引系统中，花了几个月的时间进行的一项更改只花了几天时间就在新系统中实现了。

索引过程变得更容易操作，因为大多数由机器故障、机器速度慢和网络中断引起的问题都是由MapRe-Reduce库自动处理的，无需操作员干预。此外，通过在索引集群中添加新的机器，可以很容易地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，在N个处理器上，通过并行前缀计算，可以在对数N时间内对N个元素数组的所有前缀进行关联函数计算[6，MapReduce可以被认为是基于我们在大型实际计算方面的经验对其中一些模型的简化和提炼。更重要的是，我们提供了一个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，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是MapReduce利用受限的pro编程模型自动并行化用户程序并提供透明的容错。

我们的备份任务机制类似于夏洛特系统[3]中使用的急切调度机制，简单急切调度的一个缺点是，如果一个给定的任务重复失败，整个计算都无法完成。我们用跳过坏记录的机制来解决这个问题的一些情况。

MapReduce的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。尽管不是本文的重点，但集群管理系统在精神上与其他系统类似，如[16]。

作为MapReduce库一部分的排序功能在操作上与现在的排序[1]类似。源计算机（mapworkers）将要排序的数据分区并将其发送给其中一个reduce worker。每个reduce worker都会在本地（如有可能，在内存中）对其数据进行排序。当然，现在的Sort没有用户可定义的Map和reduce使我们的库得到广泛应用的函数。

River[2]提供了一个编程模型，在该模型中，进程通过分布式队列发送数据来相互通信。与MapReduce一样，Rivers系统试图提供良好的平均情况下的性能，即使在存在由异构硬件或系统扰动引起的不一致性的情况下。River通过精心安排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pro gramming模型，MapReduce框架能够将问题划分为大量细粒度任务。这些任务在可用的工作线程上动态调度，以便更快的工人处理更多的任务。受限编程模型还允许我们在接近作业结束时调度任务的冗余执行，这在存在非一致性（如慢工或死工）时大大缩短了完成时间。

BAD-FS[5]有一个非常不同的编程模型

与MapReduce不同，它的目标是跨广域网执行作业。然而，有两个基本的相似之处。（1） 两者兼而有之系统使用冗余执行从数据中恢复故障造成的损失。

（2） 两者都使用位置感知调度以减少通过托管网络链路发送的数据量。

TACC[7]是一个旨在简化高可用性网络服务结构的系统，与MapReduce一样，它依靠重新执行作为实现容错的机制。

8 结论

MapRuduce编程模型已经成功在Google上被广泛使用有很多不同的目的。我们把这个成功归因于几个原因。首先，这个模型很容易使用，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，各种各样的问题很容易用MapReduce计算来表达。例如，MapReduce用于为Google的生产性web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，它可以扩展到大型机器集群该实现有效地利用了这些机器资源，因此适用于许多在谷歌遇到的大型计算问题。

我们从这项工作中学到了几点。首先，限制编程模型使分配和分配变得容易，并使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是为了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，并将中间数据的一个副本写入本地磁盘可节省网络带宽。第三，冗余执行可用于减少慢速机器的影响，并处理机器故障和数据丢失。

致谢

Josh Levenberg根据他使用MapReduce的经验和其他人的改进建议，在修改和扩展用户级MapReduce API方面发挥了重要作用。MapReduce从Google文件系统读取输入并将其输出写入Google文件系统[8]。我们要感谢Mohit Aron，Howard Gobioff，Markus Gutschke，我们还要感谢Percy Liang和Olcan Sercinoglu在开发MapReduce使用的集群管理系统方面所做的工作。Mike Burrows，Wilson Hsieh，Josh Leven-berg，Sharon Perl，Rob Pike，Debby Wallach对这篇文章的早期草稿提供了有益的评论。匿名的OSDI评论员和我们的shepherd Eric Brewer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Google工程或组织中MapReduce的所有用户提供了有用的反馈，建议，和错误报告。

工具书类

[1]Andrea C.Arpaci Dusseau，Remzi H.Arpaci Dusseau，David E.Culler，Joseph M.Hellerstein，和David A.Pat-terson.工作站网络的高性能分类，1997年ACM SIGMOD国际数据管理会议论文集，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/Owith River:使快速案例变得常见。《第六届并行和分布式输入/输出研讨会论文集》（IOPADS’99），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:网络上的元计算.第九届并行与分布式计算系统国际会议论文集，1996。

[4]Luiz A.Barroso，Jeffrey Dean，and Urs Holzle.一个星球的网络搜索：谷歌集群架构。IEEE Micro，23（2）：22-282003年4月。

[5]John Bent，Douglas Thain，Andrea C.Arpaci Dusseau，Remzi H.Arpaci Dusseau，和Miron Livny.批处理感知分布式文件系统中的显式控制.Ist-USENIX网络系统设计与实现研讨会论文集，2004年3月。

[6]盖伊E.布莱洛赫。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和Paul Gauthier.基于群集的可扩展网络服务，载于第16届ACM操作系统原理研讨会论文集，78-91页，法国圣马洛，1997年。

[8]Sanjay Ghemawat，Howard Gobioff，Shun Tak Le-ung.谷歌文件系统，第19届操作系统原理研讨会，第2943页，纽约乔治湖，2003年。

[9]S.Gorlatch.扫描和其他列表同态的系统有效并行化.L.Bouge，P.Fraigni-aud，A.Mignotte和Y.Robert，编辑，1996年欧洲标准。并行处理，计算机科学1124讲义，第401-408页。斯普林格·韦拉格，1996年。

[10] 吉姆·格雷。排序基准主页。

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

[11]威廉·格罗普、尤因·罗斯克和安东尼·斯凯勒姆。《使用MPI：具有消息传递接口的可移植并行程序设计》。麻省理工学院出版社，剑桥，马萨诸塞州，1999年。

[12]L.Huston，R.Sukthankar，R.Wickremesinghe，M.Satya-narayanan，G.R.Ganger，E.Riedel，和A.Ailamaki.Di-amond：一种用于主动搜索中早期丢弃的存储体系结构。2004年USENIX文件和存储技术快速会议论文集，2004年4月。

[13]理查德·E·拉德纳和迈克尔·J·费舍尔。《计算》，ACM杂志，27（4）：831-8381980。

[14]迈克尔.O.拉宾.安全性、负载平衡和容错的有效信息分散.ACM杂志，36（2）：335-3481989。

[15]Erik Riedel，ChristosFaloutsos，.Gibson，David Nagle.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]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：Condor经验。并发与计算：实践与经验，2004。

[17]L.G.Valiant.并行计算的桥接模型。ACM通讯，33（8）：103-1111997。

[18]Jim Wyllie.Spsort:如何快速排序TB字节。

词频

本节包含一个程序，该程序统计命令行中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"

// 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;

}