MapReduce：简化大型群集上的数据处理

杰弗里·迪恩和桑杰·格马瓦特

**理论知识**

MapReduce 是一种编程模型，也是用于处理和生成大型数据集的关联实现。用户指定一个映射函数，该函数处理键/值层元中间键/值对，以及合并与同一中间键关联的所有中间值的减少函数。许多现实世界的任务在这个模型中是可以表达的，如本文所示。

这种功能样式中编写的程序是自动的，并执行程序。运行时系统负责对输入数据进行分区、在一组计算机上调度程序的交互、处理计算机故障以及管理所需的机器间通信等详细信息。这使得程序员无需任何并行和分布式系统经验，可以轻松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，并且是高度可伸缩的:一个典型的MapReduce计算在数千台机器上处理数万亿字节的数据。程序员和系统管理员使用:已经实现了一百个MapReduce程序，每天有一千个MapReduce作业在谷歌的集群上执行。

1. **介绍**

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、 给定日期等。大多数这样的计算在概念上很简单。但是，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。如何并行化计算、分发数据和处理故障的问题，共同掩盖了原始的简单计算，需要大量复杂的代码来处理这些问题。

作为对这种复杂性的反应，我们设计了一个新的抽象层次，我们试图执行的简单计算隐藏了并行化、容错、数据分布和库中负载平衡的混乱细节。我们的抽象受到Lisp和许多其他函数语言中的映射和简化原语的启发。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”应用一个映射操作，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简化操作，以便适当地组合派生数据。我们使用具有用户特定映射和减少操作的功能模型，使我们能够轻松地并行化大型计算，并将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，它能够实现大规模计算的自动并行化和分布，并结合了这个接口的实现，在大型商用电脑集群上实现了高性能。

第二节描述了基本程序设计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界面的实现。第4节描述了我们发现有用的编程模型的几个缺点。第5节针对各种任务对我们的实现进行了性能测量。第6节探讨了在谷歌中使用Map缩减，包括我们使用它作为重写我们的产品索引系统的基础的经验。第7节讨论了相关和未来的工作。

1. **程序设计模型**

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。

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

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。中间值通过迭代器提供给用户的reduce函数。这允许我们处理太大而无法在内存中记录的值列表。

* 1. **例子**

考虑在大量文档中计算每个单词出现的次数的问题。用户将编写类似以下伪代码的代码:

map(String key, String value):

// key: document name

// value: document contents

for each word w in value:

EmitIntermediate(w, "1");

reduce(String key, Iterator values):

// key: a word

// values: a list of counts

int result = 0;

for each v in values:

result+= ParseInt(v);

Emit(AsString(result));

map函数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。

此外，用户还可以编写代码，用输入和输出文件的名称以及可选的调整参数来填充mapreduce特定对象。然后，用户调用MapReduce函数，将特定对象传递给它。用户的发现与MapReduce库结合在一起(用C++实现)。附录A包含本例的完整程序文本。

* 1. **类型**

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)

reduce (k2,list(v2)) → list(v2)

即输入键和值是从不同于输出键和值的域中提取的。此外，中间键和值与输出键和值来自同一个域。

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。

* 1. **更多示例**

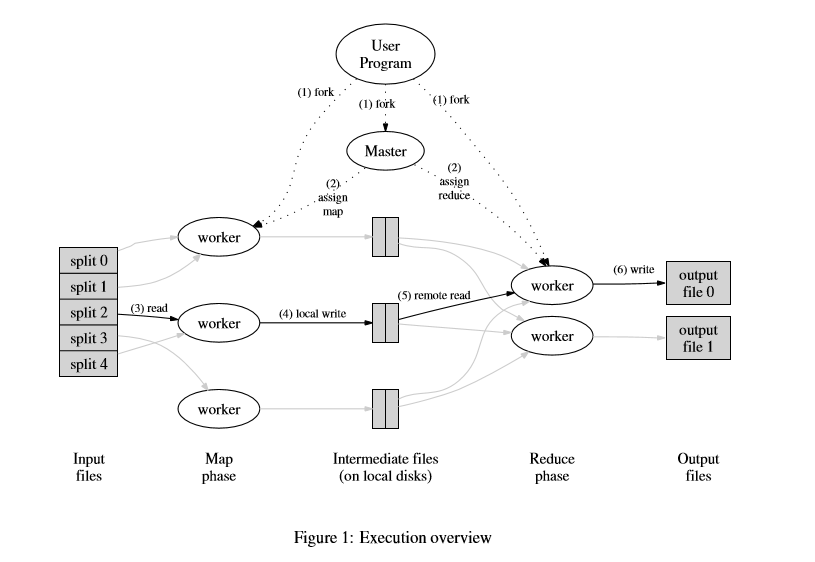
这里有几个有趣程序的简单例子，可以很容易地用MapReduce计算来表达。

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。

网址访问频率计数:映射功能处理网页请求和输出的日志(网址，1)。减少函数将同一网址的所有值相加，并发出一个(网址，总计数对。

反向网络链接图:映射函数为每个链接输出(目标，源)对到在一个名为source的页面中找到的目标URL。reduce函数连接与给定目标URL相关联的所有源URL的列表，并发出该对:(目标，列表(源))

每台主机的术语向量:术语向量将文档或一组文档中出现的最重要的单词总结为一系列(单词、频率)对。映射函数为每个输入文档发出一个(主机名，术语向量)对(其中主机名是从文档的网址中提取的)。给定主机的所有每文档术语向量都传递给reduce函数。它将这些术语向量加在一起，丢弃不常用的术语，然后发出一个术语(主机名，术语向量)对。



倒排索引:映射函数解析每个文档，并发出一系列(单词，文档标识)对。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(单词、列表(文档标识))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。

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(关键字，记录)对。reduce函数不变地发出所有对。该计算取决于第4.1节中描述的划分工具和第4.2节中描述的排序属性。

1. **安装启用**

MapReduce接口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。

本节描述了针对谷歌广泛使用的计算环境的实施: 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，具有2-4 GB的内存。

(2)使用商用网络硬件–通常在机器级别为100兆位/秒或1千兆位/秒，但平均总二等分带宽要小得多。

(3)集群由数百或数千台机器组成，因此机器故障很常见。

(4)存储由直接连接到单个机器的廉价集成开发环境磁盘提供。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。每个作业都由一组集群中的可用机器组成一个任务调度器。

**3.1 执行概述**

通过将输入数据自动划分为一组M个分割，映射调用分布在多台机器上。输入拆分可以由不同的机器并行处理。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个片段来分布减少调用。分区数量和分区功能由用户指定。

图1显示了我们的实现中MapReduce操作的总体流程。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数字):

1.用户程序中的MapReduce库首先将输入文件分割成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，它在一组机器上启动程序的许多副本。

2. 程序的一个副本是特殊的——主人。其余的是由主人分配的工作。有M个映射任务和r个减少分配的任务。主人挑选空闲的工人，并给每个人分配一个映射任务或缩减任务。

3. 分配有Map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. 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. 当主服务器没有通知缩减工作器这些位置时，它使用远程过程调用从Map工作器的本地磁盘读取缓冲数据。当reduce工作进程读取了所有中间数据后，它会按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，无法存储在内存中，则使用外部排序。

6. 缩减工作器遍历已排序的中间数据，对于遇到的每个唯一的中间键，它将键和相应的中间值集传递给用户的缩减函数。缩减功能的输出被附加到该缩减分区的最终输出文件中。

7. 当所有Map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。

成功完成后，可在R输出文件中获得Map缩减执行的输出(每个缩减任务一个，文件名称由用户指定)。通常，用户不需要将这些R输出文件合并为一个文件，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为多个文件的输入。

**3.2 主数据结构**

主服务器保留几个数据结构。对于每个映射任务和减少任务，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)。

主Map是一个管道，通过它可以从Map任务中传播中间文件区域的位置，从而减少任务。因此，对于每个已完成的Map任务，主存储由Map任务产生的中间区域的位置和大小。Map任务完成后，将接收此位置和大小信息的更新。该信息被递增地推送给正在减少任务的工作人员。

**3.3 容错**

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。

**人工故障**

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。工作人员完成的任何Map任务都将重置回其初始空闲状态，因此可以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。

已完成的映射任务会在故障时重新执行，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。

当一个映射任务首先由工作人员甲执行，然后由工作人员乙执行(因为甲失败)，所有执行缩减任务的工作人员都不知道要重新执行。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无法访问。MapReduce主机只需重新执行不可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。

**主要失败**

很容易使主写周期性地检查上述主数据结构。如果主任务终止，可以从最后一个检查点状态开始一个新的副本。然而，鉴于只有一个主人，它的失败是不可能的；因此，如果主机失败，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。

**出现故障时的语义**

当用户提供的map和reduce操作符是其输入值的确定性函数时，我们的分布式实现产生的输出将与整个程序的无故障顺序执行产生的输出相同。

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)。Map任务完成后，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Map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文件的名称。

当缩减任务完成时，缩减工作进程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同一个缩减任务，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保证最终文件系统状态只包含一次执行缩减任务产生的数据。

我们的语义相当于这种情况下的顺序执行，这一事实使得它非常 程序员很容易对他们程序的行为进行推理。当映射和/或约简操作符不确定时，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，一个特定的缩减任务R1的输出与由非确定性程序的顺序执行产生的R1的输出相等。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。

考虑Map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 位置**

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将每个文件划分为64 MB的数据块，并将每个数据块的多个拷贝(通常为3个拷贝)存储在不同的计算机上。MapReduce主机会考虑输入文件的位置信息，并尝试在包含相应输入数据副本的机器上调度Map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。当对集群中很大一部分工作人员运行大型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 任务粒度**

如上所述，我们将映射阶段细分为M个部分，将缩减阶段细分为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Map任务可以分布在所有其他工作人员计算机上。

在我们的实现中，M和R的大小是有实际限制的，因为如上所述，主机必须做出O(M + R)调度决策并在内存中保持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。

此外，R经常受到用户的限制，因为每个缩减任务的输出最终会出现在单独的输出文件中。在实践中，我们倾向于选择M，这样每个单独的任务大约有16mb到64MB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成为我们期望使用的工作机数量的一个小倍数。我们经常使用2000个工人的机器，在M = 200，000和R =5，000的情况下进行MapReduce计算。

**3.6 备份任务**

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Map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经常遇到可纠正的错误，这些错误会使其读取性能从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导致处理器缓存被禁用:受影响机器上的计算速度降低了100倍以上。

我们有一个通用的机制来缓解这些问题。当一个预执行操作即将完成时，主机会安排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，这显著减少了完成大型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。

1. **限制**

虽然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是足够的，但是我们发现了一些有用的扩展。这些将在本节中介绍。

**4.1分割函数**

MapReduce的用户指定他们想要的缩减任务/输出文件的数量。使用上的分区功能跨这些任务对数据进行分区中间键。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最终出现在同一输出文件中。

**4.2 订购保证**

我们保证在给定的分区内，中间键/值对以递增的键顺序进行处理。这种排序保证使得为每个分区生成一个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时，或者当输出文件的用户需要对数据进行排序时，这是非常有用的。

**4.3 组合器功能**

在某些情况下，每个映射任务产生的中间键有明显的重复，用户指定的Reduce函数是可交换和关联的。一个很好的例子是单词计数示例2.1。由于单词频率倾向于遵循Zipf分布，每个映射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进行部分合并。

组合器功能在执行任务的每台机器上执行。通常，same code用于实现组合器和归约功能。Reduce函数和组合函数的区别在于MapReduce库如何处理函数的输出。缩减功能的输出被写入最终输出文件。一个组合函数的输出将被发送到一个简化任务。

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 输入输出类型**

MapReduce库支持读取几种不同格式的输入数据。例如，“文本”模式输入将每行视为键/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。每个输入类型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Map任务进行处理(例如，文本模式的范围分割确保范围分割仅发生在线边界处)。用户可以通过提供一个简单的阅读器界面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先定义的输入类型中的一种。

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

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 副作用**

在某些情况下，MapReduce的用户发现从他们的Map和/或简化运算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，应用程序会写入临时文件，并在文件完全生成后自动重命名该文件。

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提交。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 跳过不良记录**

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。通常的做法是消除缺陷，但有时这是不可行的；也许这个bug在第三方库中，源代码不可用。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确定性崩溃，并跳过这些记录，以便向前推进。

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， 信号处理器向MapReduce主机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 本地执行**

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。为了有助于调试、规划和小规模测试，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Map任务。用户用一个特殊的文件调用他们的程序，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 状态信息**

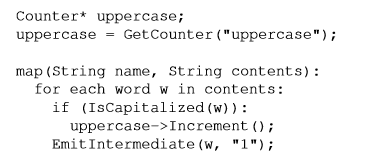
主服务器运行一个内部的超文本传输协议服务器，并导出一组供人类使用的状态页面。状态页显示了计算的进度，例如有多少任务已经完成，有多少正在进行中，字节数，字节数，输出字节数，处理速率等。这些页面还包含每个任务生成的标准误差和标准输出文件的链接。用户可以使用这些数据来预测计算需要多长时间，以及是否应该向计算中添加更多资源。当计算比预期慢得多时，页面也可以用来配置。

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 计数器**

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。

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:



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。聚合计数器值时，主机消除重复执行同一Map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，或者由于失败而重新执行任务。)

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。

用户已经发现计数器工具对于检查MapReduce操作的行为的健全性很有用。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。

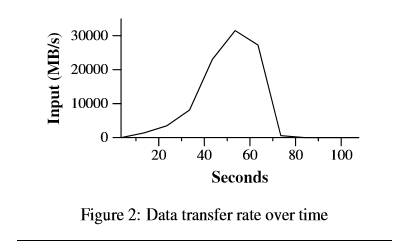
1. **表现**

在本节中，我们将在大型计算机集群上运行的两个计算上测量MapReduce的性能。一次计算搜索大约1tb的数据，寻找特定的模式。另一种计算对大约1tb的数据进行排序。

这两个程序是由MapReduce用户编写的真实程序的一个大子集的代表——一类程序将数据从一个表示形式转换到另一个表示形式，另一类程序从一个大数据集提取少量有趣的数据。

* 1. **群集配置**

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。每台机器都有两个支持超线程的2GHz英特尔至强处理器、4GB内存、两个160GB集成开发环境磁盘和一个千兆以太网链路。这些机器被安排在一个两级树形交换网络中，在根节点有大约100-200 Gbps的总带宽。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。



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。

* 1. **Grep**

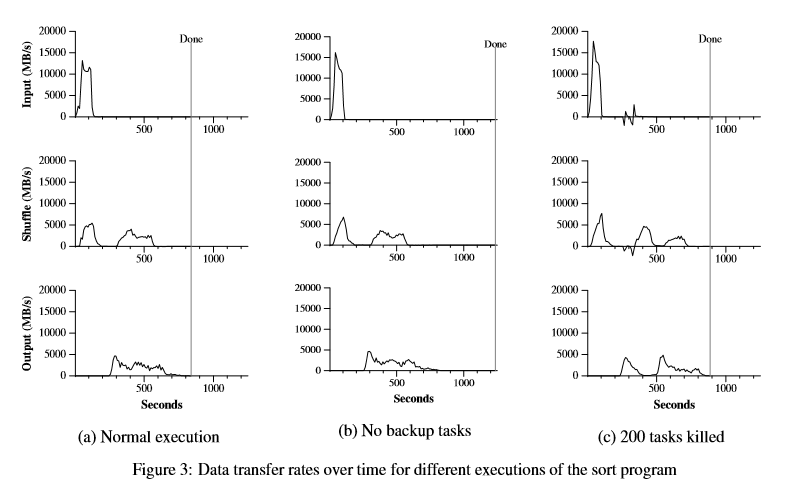
grep程序扫描10的10次方个100字节的记录，寻找一个相对罕见的三字符模式(该模式出现在92，337条记录中)。输入分成大约64MB的部分(M = 15000)，整个输出放在一个文件中(R =1)。

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30GB/s。随着Map任务的完成，速率开始下降，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启动开销。开销是由于程序传播到所有工作机，以及延迟与GFS交互以打开1000个输入文件集并获得局部优化所需的信息。

* 1. **分类**

排序程序对10的10次方个100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。

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，并将该键和原始文本行作为中间键/值对发出。我们使用了一个内置的身份函数作为减少操作符。此函数将中间键/值对作为输出键/值对不变地传递。最终的排序输出被写入一组双向复制的GFS文件(即，2tb被写入作为程序的输出)。



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(R = 4000)。分割函数使用密钥的初始字节将它分割成R个片段中的一个。

我们在这个基准测试中的分区函数已经内置了密钥分配的知识。在一般的排序程序中，我们将添加一个预通过MapReduce操作，该操作将收集键的样本，并使用样本键的分布来计算最终排序通过的分界点。

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Map任务都在200秒之前完成。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。

左中图显示了数据通过网络从Map任务发送到缩减任务的速率。第一个Map任务一完成，此洗牌就开始。图表中的第一个峰是 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秒后，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整理数据。所有的洗牌都是在大约600秒内完成的。

左下角的图表显示了缩减任务将排序后的数据写入最终输出文件的速率。由于机器忙于对中间数据进行排序，因此在第一个洗牌周期的结束和写入周期的开始之间有一个延迟。写入会以大约2-4 GB/s的速度持续一段时间。所有的写操作都需要大约850秒的计算时间。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]。

需要注意的几点:由于我们的位置优化，输入速率高于缓冲速率和输出速率——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。混洗率高于输出率，因为输出阶段写入两个已排序数据的副本(出于可靠性和可用性原因，我们制作了两个输出副本)。我们编写两个副本，因为这是我们底层文件系统提供的可靠性和可用性机制。如果底层文件系统使用擦除编码[14]而不是复制，写入数据的网络带宽要求将会降低。

* 1. **备份任务的影响**

在图3 (b)中，我们显示了备份任务被禁用的排序程序的执行。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，这最后几个散兵游勇直到300秒后才结束。整个计算耗时1283秒，耗时增加了44%。

* 1. **机器故障**

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)。

工人死亡显示为负输入率，因为一些先前完成的Map工作消失了(因为相应的Map工人被杀死了)并且需要重做。Map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(仅比正常执行时间增加5%)。

1. **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库在我们所处理的问题上的广泛适用性感到惊喜。它已经在谷歌内部广泛的领域使用，包括:

a.大规模机器学习问题，

b.谷歌新闻和Froogle产品的聚类问题

c. 提取用于生成热门查询报告的数据(如谷歌时代精神)，

d. 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

e. 大规模图形计算

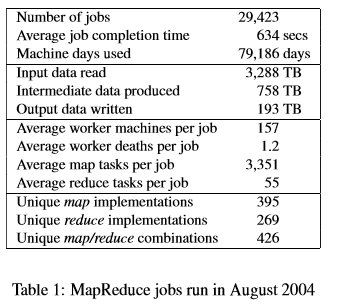
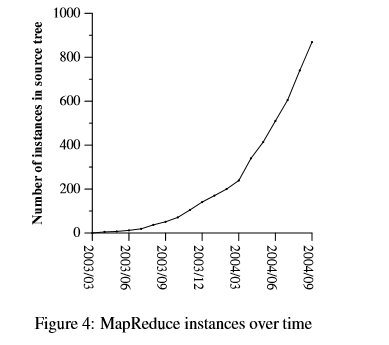


图4显示了随着时间的推移，检入我们的主要源代码管理系统的独立MapReduce程序数量的显著增长，从2003年初的0个增加到2004年9月底的近900个独立实例。MapReduce之所以如此成功，是因为它使得编写一个简单的程序并在半个小时内在一千台机器上高效运行成为可能，大大加快了开发和原型制作周期。此外，它允许没有分布式和/或并行系统经验的程序员轻松地开发大量资源。

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。

* 1. **大规模索引**

到目前为止，MapReduce最重要的用途之一是完全重写了生产索引系统，该系统生成了用于谷歌网络搜索服务的数据结构。索引系统将我们的爬行系统检索到的大量文档作为输入，存储成一组文件。这些文档的原始内容是超过20tb的数据。索引过程作为五到十个MapReduce操作的序列运行。使用MapReduce(而不是以前版本的索引系统中的临时分布式传递)提供了几个好处:

1. 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。
2. 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。
3. 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。
4. **相关著作**

许多系统提供了受限的编程模型，并使用这些限制来自动并行化计算。例如，一个关联函数可以在N个处理器上使用并行预运算在N个时间内对N个元素数组的所有预运算进行计算[6，9，13]。基于我们在大型现实世界计算中的经验，MapReduce可以被认为是其中一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。

批量同步编程[17]和一些MPI原语[11]提供了更高级别的抽象 让程序员更容易编写并行程序。这些系统和MapReduce之间的一个关键区别是，MapReduce利用一个受限的编程模型来自动并行化用户程序，并提供透明的容错。

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，计算被推进到靠近本地磁盘的处理单元中，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。

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找出了这个问题的一些实例。

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发和运行用户任务。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。

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(Map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，NOW-Sort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]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。MapReduce有一种不同的方法。通过限制编程模型，MapReduce框架能够将问题划分为大量未完成的任务。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]有一个与MapReduce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]是一个旨在简化高可用性网络服务构建的系统。像MapReduce一样，它依赖于重新执行作为实现容错的机制。

1. **结论**

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。第二，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实现，它可以扩展到包含数千台机器的大型机器集群。该实现充分利用了这些机器资源，因此适用于谷歌遇到的许多大型计算问题。

我们从这项工作中学到了一些东西。首先，对编程模型的限制使得并行化和分布式计算变得容易，并使这种计算具有容错性。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旨在减少通过网络发送的数据量:局部性优化允许我们从本地磁盘读取数据，将中间数据的单个副本写入本地磁盘可以节省网络带宽。第三，冗余执行可以用来减少慢速机器的影响，并处理机器故障和数据丢失。

**感谢**

Josh Levenberg根据他使用MapReduce的经验和其他人对增强功能的建议，在修改和扩展用户级MapReduce应用编程接口方面发挥了重要作用。MapReduce从谷歌文件系统中读取输入，并将输出写入其中[8]。我们要感谢莫希·阿隆、霍华德·戈比欧夫、马库斯·古施克、大卫·克莱默、梁信德和乔希·雷石东在开发全球金融服务方面所做的工作。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·莱文伯格、莎伦·佩尔、罗布·派克和王思然·瓦拉克对本文的早期草稿提供了有益的评论。匿名评论者和我们的牧羊人埃里克·布鲁尔为论文的改进提供了许多有用的建议。最后，我们感谢谷歌工程组织中的所有MapReduce用户提供了有用的反馈、建议和错误报告。

**参考**

[1] Andrea C. Arpaci-Dusseau，Remzi H. Arpaci-Dusseau，DavidE .约瑟芬·卡勒。工作站网络上的高性能排序。1997年5月在亚利桑那州图森举行的1997年ACM SIGMOD国际数据管理会议记录。

[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]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。1996年第九届并行和分布式计算系统国际会议录。

[4]路易斯·巴罗佐、杰弗里·迪恩和乌尔斯·奥兹勒。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]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，NSDI，第一届USENIX网络系统设计与实现研讨会论文集。

[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]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。基于集群的可扩展网络服务。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]桑杰·格玛瓦特、霍华德·戈比夫和梁信德。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。

[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]吉姆·格雷。对基准主页进行排序。<http://research.microsoft.com/barc/SortBenchmark/>。

[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。麻省剑桥，麻省理工学院出版社，1999。

[12] L.Huston，R . Sukthankar，R.Wickremesinghe，M.Satyanarayanan，G.R.Ganger，E.Riedel，AnDa . ai amaki . Diamond:交互式搜索中早期丢弃的存储体系结构.在2004年USENIX文件和存储技术会议记录中，2004年4月。

[13]理查德·拉德纳和迈克尔·费希尔。并行前置计算。美国计算机学会杂志，27(4):831-838，1980。

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

[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。

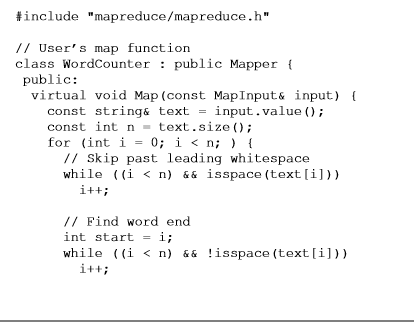
[16]道格拉斯·塞恩、托德·坦南鲍姆和米隆·利夫尼。实践中的分布式计算:秃鹰体验。并发和计算:实践和经验，2004。

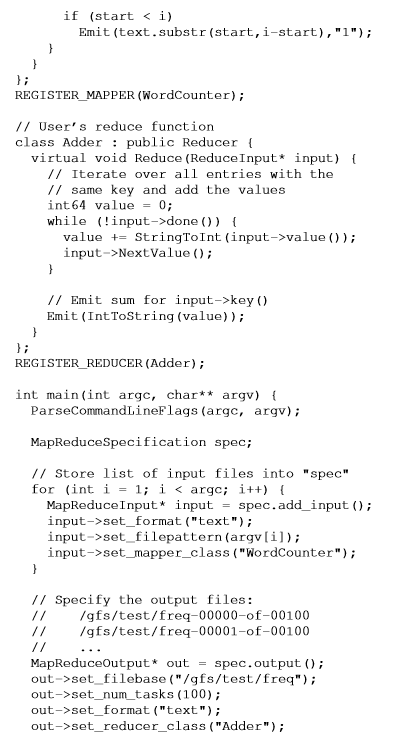
[17]瓦兰特·阿布里德金并行计算模型。美国计算机学会通讯，33(8):103-111，1997。

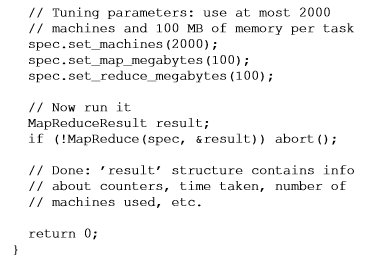
[18]吉姆·威利。Spsort:如何快速对万亿字节进行排序。<http://alme1.almaden.ibm.com/cs/spsort.pdf>。

**A 字频率**

本节包含一个程序，用于计算命令行上指定的一组输入文件中出现的唯一单词的数量。





****