Mapreduce: 即 Map（映射）和 reduce（归约）

Jeffrey dean 和 sanjay ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

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 map 函数来处理一个键/值对以生成一组中间键/值对，还指定一个 reduce 函数来合并所有与同一个中间键相关的中间值。许多现实世界的任务可以在这个模型中表达出来，正如论文中所述

用这种函数样式编写的程序自动并行化，并在大型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。这使得没有任何并行和分布式系统经验的程序员可以方便地利用大型分布式系统的资源。

我们的 mapreduce 实现运行在大量的商用机器上，并且具有高度的可扩展性：典型的 mapreduce 计算处理在数千台机器上的大量数据。程序员发现这个系统很容易使用：已经实现了数百个 mapreduce 程序，每天有超过 1000 个 mapreduce 工作被执行。

1 引言

在过去的五年里，作者和 google 的其他人已经实现了数百种特殊的、无目的的、处理大量原始数据的计算方法，例如爬行文档、 web 请求日志等，来计算各种派生的数据，例如反演的刻度、 web 文档图形结构的各种表示、每个主机上爬行的页面数量的总结、代

码日期中最常用的一组查询等等，这些计算方法大多数都是直观的。然而，输入的数据通常都很大，为了在合理的时间内完成，计算必须分布到多个分支或数千台机器上。如何并行化计算、分布数据、处理失败等问题共同掩盖了原有的简单计算和大量复杂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，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、数据分发和库负载平衡的混乱细节。我们的抽象是受到了 lisp 和许多其他函数式语言中的 map 和 reduce 原语的启发。我们意识到，大多数计算都涉及到对输入中的每个逻辑“记录”应用映射操作，以便计算一组中间键/值对，然后对所有共享同一键的值应用 reduce 操作，以便适当地组合派生的数据。我们使用了一个具有 userspecificmap 和 reduce 操作的函数模型，这使得我们可以轻松地并行化大型计算，并使用重执行作为容错的主要机制。

这项工作的主要贡献是一个简单而强大的界面，可以实现大规模计算的自动并行化和分布，结合这个界面的实现，可以在大型商品 pc 集群上获得高性能。

第二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 mapreduce 接口的一个实现，该接口适合于基于集群的计算环境。第 4 节描述了我们发现有用的编程模型的几个改进。第五部分有我们执行各种任务的执行情况。第 6 节探讨了 mapreduce 在 google 中的使用，包括我们使用它作为重写生产索引系统的基础的经验。第 7 节讨论相关的和未来的工作。

2 编程模型

计算采用一组输入键/值对，并生成一组输出键/值对。Mapreduce 库的用户将计算表

示为两个功能: 映射和 reduce。

Map 由用户编写，它接受一个输入对并生成一组中间键/值对。Mapreduce 库将与同一个中间键 i 相关联的所有中间值组合在一起，并将它们传递给 reduce 函数。

Reduce 函数也是由用户、 acceptsan 中间键 i 和该键的一组值编写的。它将这些值合并在一起，形成一个可能较小的值集。每次 reduce 调用通常只生成零个或一个输出值。中间值通过迭代器提供给用户的 reduce 函数。这允许我们处理内存中容纳的值列表。

2.1 例如

考虑计算大量文档中每个单词出现的次数的问题。用户可以编写类似于以下伪代码的代码:

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

Map 函数发出每个单词以及出现次数的相关计数(在这个简单的例子中只有“1”)。归约函数求和所有为一个特定单词发出的计数。

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映射/还原/还原/还原对象。然后用户调用 mapreduce 函数，将指定分类对象传递给它。用户代码与themapreduce 库(在 c + + 中实现)链接在一起。附录附上了这个例子的全部程序文本。

2.2 类型

即使前面的伪代码是以字符串输入和输出的 termsof 形式编写的，从概念上讲，用户提供的 map 和 reduce 函数具有相关类型:

map

(k1,v1)

→ list(k2,v2)

reduce

(k2,list(v2))

→ list(v2)

例如，输入键和值是从不同的域中绘制的，而不是从输出键和值中绘制的。此外，中间键和值与输出键和值来自同一个域。

我们的 c 实现将字符串传递给用户定义函数，并将其留给用户代码在字符串和适当类型之间进行转换。

2.3 更多列子

这里有一些有趣的程序的简单例子，可以很容易地表示为 mapreduce 计算。

Distributed grep: map 函数在匹配提供的模式时发出一行。Reduce 函数是一个同一函数，它只是将提供的中间数据复制到输出。

统计 url 访问频率: map 函数处理 web 页面请求和 outputshurl 的日志，1i。函数将同一个 url 的所有值加在一起，并发出一个 hurl，total countipair。

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。

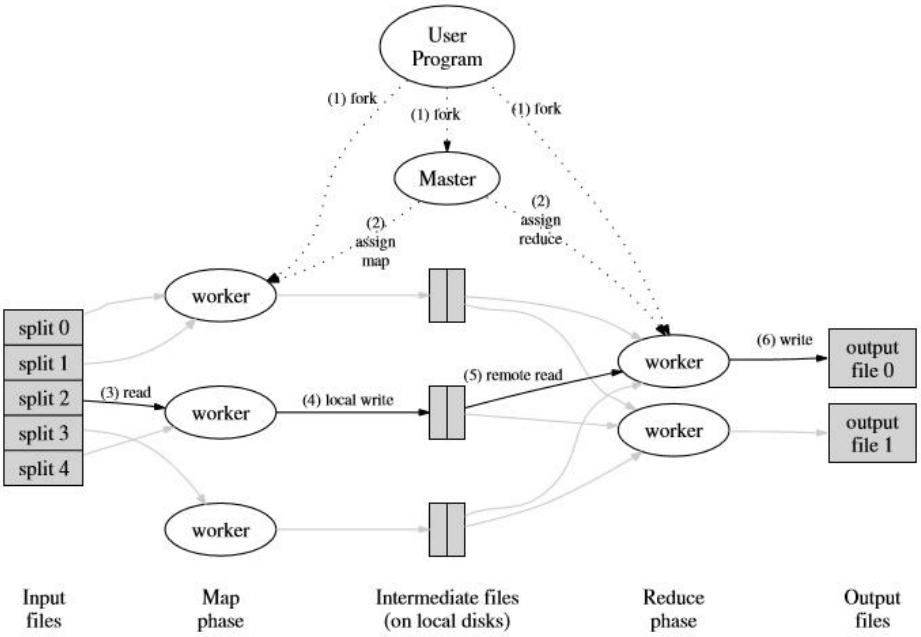

Figure 1: Execution overview

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。

3 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 reduce 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 reduce 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. reduce 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 reduce 函数。reduce 功能的输出被附加到该 reduce 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 Mapreduce 执行的输出(每个 reduce 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 reduce 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 reduce 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 reduce 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 reduce 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。reduce 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 reduce 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 reduce 任务完成时，reduce 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 reduce 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 reduce 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 reduce 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 reduce 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 reduce 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/reduce 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 reduce 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。

4 限制

虽然简单地编写映射和 reduce 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 reduce 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。reduce 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 reduce 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 reduce 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 reduce 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:

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

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

来自单个工作机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 reduce 任务中聚合计数器值，并在映射 reduce 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。

5 表现

在本节中，我们将测量 mapreduce 在大型机群上运行的两个计算上的性能。一次计算搜索大约 1 兆兆字节的数据，寻找特定的模式。这两个程序代表了由 mapreduce 用户编写的实际程序的一个大子集——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中提取一小部分有趣的数据。

5.1 群集配置

所有的程序都在一个由大约 1800 台机器组成的集群中执行。每台机器都有两个 2ghz 的 intel xeon 处理器，支持超线程，4 gb 内存，2 个 160gb 的 ide 磁盘，以及一个吉比特以太网链接。这些机器被放置在一个两级树形交换网络中，根目录大约有 100-200gbps 的聚合带宽。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 1-1.5 gb 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，当时 cpu、磁盘和网络大多处于空闲状态。

5.2 Grep

Grep 程序扫描 1010100 字节的记录，搜索相对罕见的三字符模式(模式出现在 92,337 条记录中)。输入被分成大约 64mb 的部分(m = 15000) ，整个输出被放置在一个文件中(r = 1)。图 2 显示超时运算的进度。Y 轴显示输入数据被扫描的速率。随着分配给这个 mapreduce 计算的机器越来越多，这个速率逐渐加快，当 1764 个工作人员被分配时，速率达到 30gb/s 以上。整个计算从开始到结束大约需要 150 秒。包括一分钟的启动开销。开销是由于程序传播到所有工作机器，并延迟与 gfs 交互以打开 1000 个输入文件集并获取局部性优化所需的信息。

5.3 Sort

排序程序对 1010100 字节的记录(大约 1 太字节的数据)进行排序。这个程序是模仿 terasort 基准[10]。排序程序只有不到 50 行的用户代码。一个三行 map 函数从文本行中提取一个 10 字节的键，并将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标识函数作为 reduce 操作符。最终的分类输出被写入一组双向复制的 gfs 文件(也就是说，写入 2 太字节作为程序的输出)。与前面一样，输入数据被分割成 64mb 的部分(m = 15000)。我们将排序后的输出分成 4000 个文件(r = 4000)。分区函数使用键的初始字节将其隔离成一个 r 段。我们的基准分区函数已经建立了键的分布知识。在通用排序程序中，

我们会添加一个预通道映射/约简操作，它将收集键的样本，并使用采样键的分布来计算最终排序通道的分割点。

图 3(a)显示排序程序的正常执行的进度。左上角的图表显示了读取输入的速度。由于所有 Map 任务在 200 秒之前完成，速率峰值约为 13gb/sand，因此相当快地消失。请注意，输入率小于 grep。这是因为排序映射任务花费大约一半的时间和 i/o 带宽将中间输出写入本地磁盘。左中间的图表显示了数据通过网络从 map 任务发送到 reduce 任务的速率。一旦第一个 Map 任务完成，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，每台机器最多每次执行一个 reduce 任务)。计算进行大约 300 秒后，有些第一批 reduce 任务完成，我们开始对剩余 reduce 任务的数据进行重组。左下角的图表显示了 reduce 任务将排序数据写入最终输出文件的速率。由于机器正忙于对中间数据进行排序，所以在第一个洗牌周期结束和写入周期开始之间存在延迟。写入速度在 2-4gb/s 之间保持一段时间。所有的写操作在计算开始后 850 秒内完成，包括启动开销，整个计算耗时 891 秒。这类似于目前最好的报告/基准测试 1057 秒的结果[18]。需要注意的是: 由于我们的局部优化，输入速率高于分频率和输出速率——大多数数据从本地磁盘读取，绕过了我们相对带宽有限的网络。我们编写两个副本，因为这是由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，则写入数据的网络带宽要求将会减少。

5.4 备份任务的影响

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，只是有一个很长的尾巴，几乎没有任何写入活动。960 秒后，除了 5 个 reduce 任务外，所有任务都完成了。然而这些最后的落伍者要 300 秒后才能完成。整个计算耗时 1283 秒，

占用时间增加了 44% 。

5.5 机器故障

在图 3(c)中，我们展示了一个排序程序的执行过程，在这个程序中，我们在计算的几分钟内故意杀死了 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新的工作进程(因为只有进程被终止，所以这些机器仍然能够正常工作)。由于之前完成的一些 Map 工作消失了(因为相应的 Map 工作者被杀死了) ，需要重做，因此工人的死亡显示为负输入级别。这个 Map 工作的重新执行速度相对较快。整个计算在 933 秒内完成，包括启动开销(比正常执行时间增加 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 月对其进行了重大改进，包括位置优化、跨工作机任务执行的动态负载平衡等:?大规模机器学习问题，?谷歌新闻和搜索产品的聚类问题，?用于生成流行查询报告的数据提取(如谷歌时代思想) ，用于新实验和产品的网页属性提取(如从大量网页语料库中提取地理位置以进行局部搜索) ，以及?大规模图形计算。

图 4 显示了随着时间的推移，在我们的基本源代码管理系统中检入的单独 mapreduce 程序数量的显著增长，从 2003 年初的 0 个到 2004 年 10 月底的将近 900 个单独的实例。Mapreduce 之所以如此成功，是因为它使编写一个简单的程序成为可能，并在一千台机器上以半小时的时间高效地运行它，极大地加快了开发和原型设计的周期。此外，它还允许没有分布式和/或并行系统经验的程序员轻松地利用大量资源。在每个作业结束时，mapreduce 库日志/作业使用的计算资源统计。在表 1 中，我们显示了 2004 年 8 月 google 运行的 mapreduce 作业子集的一些统计数据。

6.1 大规模索引

我们使用 mapreduce to date 最重要的用途之一就是完全改写了生产索引系统，该系统生成用于 google 网络搜索服务的数据结构。索引系统需要输入一系列的文档，这些文档已经被我们的爬行系统检索，并存储为一组 gfs 文件。这些文件的内容有超过 20tb 的数据。索引过程按照 5 到 10 个 mapreduce 操作的顺序运行。使用 mapreduce (而不是先前版本的索引系统中的 ad-hoc 分布式传递)提供了几个好处:索引代码更简单，更小，更容易理解，因为处理错误处理，分发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，以避免对数据进行额外的传递。这样就很容易改变索引过程。例如，在旧的索引系统中做一个改变花了几个月的时间，而在新系统中只花了几天的时间。?索引过程已经变得更容易操作，因为大多数由机器故障、机器运行缓慢和网络//cups 引起的问题都由 mapreduce 库自动处理，不需要操作员的干预。此外，通过在索引集群中添加新的机器，可以很容易地改进索引过程的性能。

7 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 reduce 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。

8 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。

```
#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;
}
```