摘要

MapReduce：大型集群上的Simpliﬁ编辑数据处理Jeffrey Dean和Sanjay GhemawatJeff@google.com，Sanjay@google.com谷歌公司摘要MapReduce是一种编程模型，也是一个Acoci-用于处理和生成大数据集的有关联的实现。用户指定处理Akey/Value对以生成一组中间键/值的映射函数，以及合并与同一中间键相关联的所有中间值的约简函数。如本文所示，虚拟世界任务在这个模型中是可以表达的。

用这种功能风格编写的程序是自动的-Cally并行化并在大型COM-modity机器集群上执行。运行时系统负责对输入数据进行分区、在一组机器上调度过程执行、处理ma-chine故障以及管理所需的机器间通信等细节。这使得没有任何并行和分布式系统经验的程序员能够有效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具有高度可伸缩性的商品机器集群：一个典型的MapReduce计算处理成千上万台机器上的许多万亿字节的数据。程序员ﬁ和该系统易于使用：数百个MapReduce程序已经实现，超过一个你和MapReduce的工作是在谷歌的集群日执行。

导言在过去的ﬁ年中，作者和谷歌的许多其他人实现了数以百计的特殊目的的计算，它们处理大量原始数据，如爬行文档、web请求日志等，以计算各种派生数据，如无脊椎数据、web文档图表结构的各种表示、每台主机页面爬行数的汇总、一个主机中最常见的查询集。给定的一天，等等。大多数这样的计算都是简单明了的。然而，输入的数据通常是庞大的，计算必须分布在并行或数千台机器上，这样才能使ﬁ在合理的时间内完成。如何实现计算的等位化、数据的分发和操作故障等问题，使得原始的简单组合与大量的复杂代码相混淆，从而解决了这些问题。

作为对这种复杂性的反应，我们设计了一个新的抽象，允许我们表达我们试图执行的简单计算，但隐藏了库中的并行化、容错、数据分配和负载平衡等杂乱无章的解尾。我们的抽象是由映射提供的，减少了Lisp和许多其他功能语言中的原语。我们意识到，我们的大多数计算都涉及到对输入中的每个逻辑“记录”应用一个映射运算，以便计算一组中间键/值对，然后对共享相同键的所有值应用一个约简操作，以便将派生的数据完全结合起来。我们使用的功能模型与用户规格的ﬁed映射和约简操作，使我们可以很容易地并行大型计算，并使用重执行作为主要的容错机制。

这项工作的主要贡献是简单和强大的接口，允许大规模计算的自动并行化和分布，再加上这种接口的实现，在大型商品PC集群上实现高性能。

第2节描述了基本的编程模型和给出了几个例子。第3节描述了基于MapReduce接口定制的基于双酸集群的计算环境的集成.第4节介绍了一些我们认为有用的编程模型的ﬁ实例。第5节提供了实现各种任务的性能。第6节探讨了MapReduce在Google中的使用，包括我们使用它作为基础的经验来重写我们的生产索引系统。第七章讨论了相关工作和今后的工作

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库的用户以地图和约简两种功能表示计算。地图，由用户编写，采用输入对，并支持-引入一组中间键/值对。MapRe-duce库将所有中间值组合在一起，如-使用相同的中间键i插入并传递它们还原函数。也由用户编写的还原函数接受中间键I和该键的一组值。它将这些值合并在一起，形成一组可能较小的值。通常，每次减少调用只产生零或一个输出值。中间Val-ue通过ITER-ator提供给用户的减少功能.这允许我们处理内存中的ﬁt的工具值列表。

2.1实例考虑计算大量文档中每个单词的oc-currence数的问题。用户将编写类似于下面的伪代码的代码：映射(字符串键，字符串值)：

Map函数会发出每个单词加上一个相关的次数的计数(在这个简单的例子中只有‘1’)。此外，用户在MapReduce中将代码写入ﬁ11SPECIﬁ阳离子对象，具有输入和输出ﬁLES的名称，以及可选的调优参数。然后，用户调用MapReduce函数，并将speciﬁ-阳离子对象传递给它。用户的代码与MapReduce库(用C++实现)链接在一起。附录A包含此示例的完整程序文本。

2.2类型即使前面的伪代码是用字符串输入和输出来编写的，但从概念上讲，用户提供的映射和缩减函数具有关联类型：MAP(K1，v1)→列表(K2，v2)减少(K2，List(V2))→列表(V2)也就是说，输入键和值是从不同的域提取的，而不是输出键和值。此外，中间键和值来自与输出键和值相同的do-main。我们的C++实现将字符串传递给User-deﬁNed函数并将其留给用户代码，以在字符串和适当类型之间进行转换。

2.3更多的例子以下是一些有趣程序的简单示例，它们可以很容易地表示为MapReduce计算。分布式grep：如果映射函数发出一行匹配提供的模式。减少函数是一个标识函数，它只是将所提供的Intermedi-ate数据复制到输出。URL访问频率的计数：地图功能-处理网页请求和输出的日志用力，我。减少函数将相同URL的所有值加在一起，并发出一个掷数、总数一对。反向Web链接图：映射函数输出指向目标的每个链接的Sourcei对在名为Source的页面中找到的URL。还原函数将所有源URL的列表连接起来，这些URL与给定的目标URL相关联，并发出这一对：HTarget，L.I(来源)i术语-每个宿主的向量：一个术语向量总结了发生在文档或集合中的最重要的单词将文件作为Hword、f requencyi对的列表。Themap函数发出一个hhostname，术语vectori。对每个输入文档(其中主机名从文档的URL中提取)。还原函数传递给给定主机的所有每个文档项向量。它将这些项向量相加在一起，丢弃不频繁的项，然后发出一个ﬁnal。Hhostname，术语vectori对。

倒排索引：映射函数解析每个docu-和发射一系列的Hword，文件Idi成对。还原函数接受一个givenword的所有对，对相应的文档ID进行排序，并发出一个Hword，列表(文档ID)i对。所有输出的集合对构成一个简单的倒排索引。这是很容易增加这一计算，以保持跟踪词的位置。分布式排序：map函数提取密钥从每个记录中发出一个hkey记录对。这个减少函数发射所有对不变。这种组合取决于第4.1节中描述的分区设施和Sec-4.2中描述的排序属性。

3执行MapReduce在地面上的许多不同的实现是可能的。正确的选择取决于环境。例如，一个实现可能适用于一个小型共享内存机器，另一个用于大型NUMA多处理器，另一个用于更大的联网机器集合。本节描述目标为目标的实现。对于谷歌广泛使用的计算环境：大量商品PC与交换以太网连接在一起[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)使用的是商品网络硬件--典型的100兆比特/秒或1G/秒，但在所有的二分带宽中平均要小得多。(3)集群由数百或数千个ma-chines组成，因此机器故障很常见。(4)存储是由直接存储到各个机器的廉价IDE磁盘提供的。内部开发的分布式ﬁLASS系统[8]用于管理这些磁盘上的数据传输。ﬁLE系统在不可靠的硬件之上使用复制来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。

3.1执行概览地图调用通过自动划分输入数据而分布在多个数据线上。成一组M分裂。输入分裂可以是支持的-塞住被不同的机器并行。减少客座-通过对中间键进行分区来分发操作。使用分区函数(例如，散列(键)模R)分区数(R)和分区函数是由用户编写的speciﬁ。图1显示了MapReductop的总体ﬂow-在我们的实施中。当用户编程调用MapReduce函数时，会发生以下顺序操作(图1 Corre-Spond中的编号标签与下面列表中的数字)：

1.用户程序ﬁrst中的MapReduce库将输入的ﬁLES分割成典型的16的M个片段每片64 MB(用户可通过可选参数进行协商)。然后，它在一台机器上启动了许多程序的副本。

2.节目的其中一份是特别的-师父。其余的是被分配工作的工人。被主人。有M个MAP任务和R减少要分配的任务。主程序选择空闲的工作人员，并为每个工作人员分配一个映射任务或一个约简任务。

3.分配给地图任务的工作人员将读取相应输入分割的内容。它从输入数据中解析键/值对，并将每个值对传递给用户deﬁNedMap函数。Map函数生成的Interme-diate键/值对被缓冲在内存中。

4.定期将缓冲对写入本地。磁盘，通过分区划分为R区域。功能。本地磁盘上的这些缓冲对的位置被传回给主机，后者负责将这些位置转发给相应的工作人员。

5.当减少工人不被主ﬁ编辑时关于这些位置，它使用远程过程调用从内存工作者的本地磁盘读取缓冲数据。当减少工作人员读取了所有中间数据时，它会按中间键对其进行排序，这样所有出现的相同键都会被聚在一起。之所以需要排序，是因为通常情况下，许多不同的键映射到相同的约简任务。如果中间数据量太大，无法在内存中ﬁt，则使用外部排序。

6.减少工人在排序的中间层上迭代-对于每个唯一的中间键，它将密钥和对应的中间值集传递给用户的约简功能。这个约简函数的输出被附加到这个减少分区的ﬁnal输出ﬁle中。

7.当所有映射任务和减少任务已完成，主人唤醒了用户程序。此时，用户程序中的MapReduce调用返回到用户代码。在成功完成后，地图的输出-Duce执行可在R输出ﬁLES中使用(每一个一个使用ﬁle名称作为用户编辑的speciﬁ)。通常，用户不需要组合这些R输出ﬁ将Les转换成一个ﬁle--它们通常将这些ﬁLES作为输入传递给另一个MapReduce调用，或者从另一个能够处理被划分为多个ﬁLES的输入的分布式应用程序中使用它们。

3.2主数据结构主机保存了几个数据结构。对于每个映射任务和约简任务，它存储状态(空闲、正在进行或已完成)，以及工作机器的标识(对于非空闲任务)。主机是位置所在的管道。中间ﬁ的LE区域从映射任务传播到减少任务。因此，对于每个已完成的地图任务，主存储器存储R间的位置和大小-中介映射任务生成的ﬁle区域。在完成映射任务时，将收到对此位置和大小的更新信息。这些信息在精神上被推送给正在进行中的减少任务的工人。

3.3容错由于MapReduce库的设计目的是帮助使用数百台或数千台机器处理大量数据，因此该库必须能够很好地容忍机器故障。工人失败主人定期给每个工人打电话。如果在一定时间内没有从工人那里接收到应答，主会将工作人员标记为失败。任何由工人完成的映射任务都会被重置回他们的初始空闲状态，因此有资格在其他工作人员上执行计划。类似地，在失败员工上正在进行的任何映射任务或还原问题也会被重置为idleand，并有资格重新安排时间。完成的映射任务在失败时重新执行-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。Completedreduce任务不需要重新执行，因为它们的日志存储在全局ﬁle系统中。在执行映射任务时，ﬁrst由工作人员A和然后由工作人员B执行(因为A失败)，所有执行减少任务的工作线程将收到重新执行通知。任何尚未从辅助角色 A 读取数据的缩减任务都将从辅助角色 B 读取数据。例如，在一次的MapReduce操作期间，一个正在运行的群集上的网络维护导致一次有80台机器组在几分钟内无法访问。MapReduceMaster 简单执行了无法到达的工人机器完成的工作，并继续向前推进，最终完成 MapReduce 操作MasterFailure 很容易使主写入主数据结构的周期检查点。如果主任务已死，可以从上次检查状态启动新副本。但是，鉴于只有一个主，其失败的可能性不大;因此，如果主函数失败，我们的当前实现将中止 MapReduce 计算。客户端可以检查此条件，并重试 MapReduce 操作（如果需要）。当用户提供的映射器是确定性函数软海林普卢值值时，我们的分布式实现输出结果可能由整个程序的无故障顺序执行产生。我们依靠映射的原子提交和减少任务输出来实现此属性。每个在进行中的任务都编写其输出的临时文件。Areducetask 生成一个这样的文件，地图任务生成 R 这样的文件（每个减少任务一个）。当映射任务完成时，工作人员向主服务器发送一条消息，并在消息中包括 R 临时文件的名称。如果主服务器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。当减少任务完成时，减少工作线程以原子方式将其临时输出文件重命名为最终输出文件。如果在多台计算机上执行相同的减少任务，则多个重命名调用将 b当映射和/或减少运算符是非确定性时，我们提供更简约但仍合理的语义。在存在非确定性运算符的情况下，特定减少任务 R1 的输出等效于通过顺序执行非确定性程序产生的 R1 输出。但是，不同减少任务 R2 的输出可能与非确定性程序的不同顺序执行产生的 R2 的输出相对应。考虑映射任务 M 并减少任务 R1 和 R2。让 e （Ri） 执行 Ri 承诺 （正好有一个这样的执行）.语义较弱是因为 e（R1） 可能读取了 M 的一次执行产生的输出，e（R2） 可能读取了 M 的不同执行产生的输出。

3.4 局部网络带宽是我们计算环境中相对具有的存储资源。我们利用输入数据（由 GFS [8] 管理）存储在我们群集的计算机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 块，并在不同的计算机上存储每个块的多个副本（通常为 3 个副本）。MapReduce 主服务器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计算机上安排地图任务。否则，它会尝试安排 amaptaskearareplicaofthattask 的 sinputdata（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器上）。当对群集中很大一部分工作人员运行大型 MapReduce 操作时，大多数输入数据在本地读取，并且不会消耗网络带宽。

3.5 任务范围我们将地图阶段细分为 M 片和减面，上面描述。理想情况下，M 和 R 应该比工作机器的数量大得多。让每个工作人员执行许多不同的任务可改善动态负载平衡，并在工作人员发生故障时加快恢复速度：它完成的许多地图任务可以分散到所有其他辅助角色机器上。我们的实现中可以实现实际边界，因为主服务器必须做出 O（M + R） 调度决策，并如上所述，将 O（M +R） 状态保留在内存中。（常量因子形式设备小然而：状态的 O（M+R） 部分由每个地图任务/减少任务对的大约一字节的数据组成。此外，R 通常受到用户的限制，因为每个减少任务的输出最终都位于单独的输出文件中。在实践中，我们倾向于选择 M，以便每个单独的任务大约是 16 MBto 64MB 的input 数据（因此，在最有效之前，我们使 R 成为我们期望使用的工作机器数量的小倍。我们经常使用 M = 200，000 和 R = 5 ，000，使用 2，000 个工机执行 MapReduce 计算。

3.6 BackupTasks 延长 MapReduce 操作总时间的常见原因之一是"切换器"：需要非常长时间才能完成最后几个映射之一或减少计算任务的机器。由于种种原因，可能会出现流浪汉。例如，磁盘损坏的机器可能会遇到频繁的可纠正错误，这些错误会将其读取性能从 30 MB/s 降低到 1 MB/s。群集调度系统可能计划在计算机上安排其他任务，由于 CPU、内存、本地磁盘或网络带宽的竞争，因此它执行 MapReduce 代码速度较慢。我们最近遇到的问题是计算机初始化代码中的一个错误，导致处理器缓存被禁用：受影响计算机上的计算速度减慢，超过一百个因子。我们有一个总体机制来缓解问题。当 aMapReduce 操作即将完成时，主任务将计划对剩余在进行中的任务的备份执行。每当主执行或备份执行完成时，任务都会标记为已完成。我们调整了此机制，以便它通常将操作使用的计算资源增加不超过百分之几。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更长的时间才能完成。

4 优化 虽然简单编写 Map 和 Reduce 函数提供的基本功能足以满足大多数需求，但我们发现很少扩展。本节将介绍这些内容。

4.1 分区功能 MapReduce 的用户指定所需的减少任务/输出文件数 （R）。数据在中间键上跨这些任务进行分区，使用分区功能。提供了使用哈希（例如"哈希（键）mod R"的默认分区函数。这往往会导致相当均衡的分区。但是，在某些情况下，通过密钥的一些其他函数对数据进行分区是很有用的。例如，有时输出键是 URL，我们希望单个主机的所有条目都最终出现在同一个输出文件中。为了支持这样的情况，MapReduce 库的用户可以提供特殊功能。例如，使用"哈希（主机名（urlkey）））mod R"作为分区函数，导致来自同一主机的allURL最终出现在同一个输出文件中。

4.2 订购保证我们保证在给定的分区中，中间键/值对按增加键顺序进行处理。这种排序保证使得每个分区可以轻松生成一个排序的输出文件，当输出文件格式需要支持按键进行有效的随机访问查找时，或者输出用户发现对数据进行排序很方便时，这非常有用。

4.3 组合功能 在某些情况下，每个地图任务产生的中间键存在显著重复，用户指定"减少"功能是交换和关联。这方面的一个很好的例子是第2.1节中的单词计数示例。由于字频率，以遵循Zipf分布，每个地图任务将产生成百上千的记录的形式 <the, 1="">。所有这些计数将通过网络发送到一个单一的减少塔斯坎德塞格格比比功能产生一个数字。我们允许用户指定一个可选的组合器函数，该函数在通过网络发送此数据之前执行部分合并。</the,> 在执行amaptask的每台机器上执行组合器功能。通常使用芝麻代码来实现组合器和减少函数。"映射函数"和"组合器"函数之间的差异是 MapReduce 库如何处理函数的输出。减少函数的输出写入最终输出文件。将发送到减少任务的一个组合程序函数的输出被写给中间文件。部分组合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 输入和输出类型 MapReduce 库为读取输入数据提供了支持。这些格式不同。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。另一种常用支持的格式存储按键排序的键/值对序列。每个输入类型实现演示显示自我扩展的自分范围作为单独的地图任务处理（例如，文本模式的范围拆分确保范围拆分仅在线边界发生）。用户可以通过提供新输入类型来添加对新输入类型的支持，尽管大多数用户只使用少量的预定义丁普类型之一。读取器不一定需要提供从文件读取的数据。例如，从数据库或从内存中映射的数据结构读取记录，易从分析器。以类似的方式，我们支持一组输出类型，用于以不同格式生成数据，并且用户代码很容易添加对新输出类型的支持。

4.5 副作用 在某些情况下，MapReduce 的用户发现从地图和/或减少运算符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使这种副作用具有原子和幂等性。通常，应用程序写入完全生成的临时文件，从上重新名称此文件。我们不提供由单个任务产生的多个输出文件的原子两相承诺。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性的。这种限制在实践上从来不是问题。

4.6 跳过坏记录有时有bugsinusercode，导致地图或减少函数崩溃确定某些记录。这种虫子预防从完成操作。通常的操作过程是修复 Bug，但有时这是不可行的;可能该 Bug 位于源代码不可用的第三方库中。此外，有时忽略一些记录是可以接受的，例如，对大型数据集进行统计分析时。我们提供一种可选的执行模式，其中地图恢复图书馆检测哪个记录导致确定性裂纹和斯基普斯的前进。每个工作进程安装一个信号处理程序，捕获分段冲突和总线错误。在调用用户映射或减少操作之前，MapReduce 库将参数的序列编号存储到全局变量中。如果用户代码生成信号，信号处理程序会向 MapReduce 主函数发送包含序列号的"最后喘息"UDP 数据包。当主服务器在特定记录上看到多个故障时，它指示在重新执行应答映射或减少任务时应跳过记录。

4.7 映射或减少函数中的本地执行调试问题可能比较棘手，因为实际计算发生在分布式系统中，通常发生在数千台计算机上，由主服务器动态做出工作分配决策。为了帮助调试、分析和小规模测试，我们先后对MapReduce库实施，该库按顺序执行本地计算机上的MapReduce操作的所有工作。控件提供给用户，以便计算可以限制为特定的地图任务。用户使用特殊标志投票程序，然后可以轻松地使用他们认为有用的任何调试或测试工具（例如 gdb）。

4.8 状态信息 主机运行内部 HTTP 服务器并导出一组状态页供人工使用。状态页显示计算的进度，例如已完成的任务数、正在进行中的任务数、字节、中间数据、输出字节、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。用户可以使用此数据来预测计算需要多长时间，以及是否应向计算中添加更多资源。当计算速度比预期的要慢得多时， 这些页面的糖贝因多图弹出。此外，顶级状态页显示哪些工作人员失败，以及在失败时映射和减少他们正在处理的任务。当尝试诊断用户代码中的 Bug 时，此信息非常有用。

4.9 计数器"地图减少库"提供了用于计算各种事件发生的计数器工具。例如，用户代码可能希望计算总数字的字处理或编号的German文档索引，等等。若要使用此工具，用户代码将创建一个命名的计数器对象，然后在"映射"和/或"减少"函数中适当增加计数器。例如：计数器\* 大写字母;大写 = 获取计数器（"大写"）;映射（字符串名称、字符串内容）：对于内容中每个单词 w：如果（大写（w））：大写=增量（）;发射中介（w，"1"）; 各个工作机器的计数器值会定期传播到主计算机（在响应响应上背负）。当 MapReduce 操作完成时，从成功映射和生成任务值的 themaster 将它们转导到用户代码。当前计数器值也显示在母版状态页上，以便人类可以观看这些计算过程。当聚合计数器值时，主计对重复执行同一地图或减少任务以避免重复计算效果。（重复执行可能来自我们使用备份任务以及因失败而重新执行任务。某些计数器值由 MapReduce 库自动维护，例如处理的输入键/值对数和产生的输出键/值输出数。用户发现计数器设施对于理智检查行为映射减少操作很有用。例如，在某些 MapReduce 操作中，用户代码可能希望确保产生的输出对数完全等于处理的输入对数，或者处理的德国文档的分数与处理的文档总数不一。

5. 性能

在本节中，我们将通过在大型计算机集群上运行的两个计算来测试MapReduce的性能。其中一个计算任务是在1TB的数据中查找某个特定的东西。另一个计算任务则是对1TB左右大小的数据进行排序。

这两个程序能代表了MapReduce的用户编写的程序的很大一部分用途。其中一类是将数据从一种表示变为另一种表示。另一类则是从一个大型数据集中提取出小部分人们所感兴趣的数据。

5.1集群配置

所有的程序都会在一个包含了大约1800台机器的集群中执行。每台机器都有两颗允许超线程技术的2GHz的Intel至强处理器，4GB内存，两块160GB大小的IDE硬盘，以及1Gbit速度的以太网络连接。这些机器被安排在两层树形交换网络中，根节点处可提供大约100-200Gbps的总带宽。所有机器都存放在同一托管机房中，因此，任意一对机器间的往返时间都小于1毫秒。

在4GB的内存中，大约有1-1.5GB左右的空间被运行在该集群中的其他任务所占用。这些程序会在周末下午执行，因为那时的CPU，磁盘以及网络大部分都处于空闲状态。

5.2 Grep

grep 程序可以从10^10条大小为100byte的数据中找到一条相对少见的数据（该数据在这个数据集中出现了92337次）。该输入数据被拆分为大约64MB大小的片段（总共有15000个这样的片段），并且整个输出结果会放在一个文件中。

Figure 2中所展示的是在一段时间内的计算进展。Y轴所表示的是扫描输入数据的速率。随着分配给用于MapReduce计算的机器数量的增加，速率也随之逐渐提高。并且当分配了1764台机器时，峰值速率超过了30GB/s。当map任务结束后，速率开始下掉，在大约80秒处，速率变为0。整个计算过程从开始到结束花了大约150秒左右。这其中包含了一分钟左右的启动开销。这种开销是由于需要将程序分发到所有测试机器上，以及与GFS进行交互以打开1000个输入文件的集合，并获取进行位置优化所需的信息而导致的延迟。

5.3 Sort

sort程序是用来对10^10条大小为100byte（大约为1TB大小的数据）的记录进行排序处理。该程序是以TeraSort基准为模型。

该排序程序所包含的代码行数少于50行。一个三行的Map函数能从一行文本中提取10byte大小的排序键，并将该键和该原始文本作为中间键值对。我们使用一个内置的Identity函数当作Reduce操作符来执行。该函数对传入的中间键值对不做改变，直接作为输出键值对输出。最终排序后的输出结果会被写入一组双向备份的GFS文件中（例如，要向文件中写入2TB大小的输出数据）。

和之前一样，输入数据会被拆分为15000个大小为64MB左右的数据块。我们将排序后的输出数据放入4000个文件中。分区函数会使用key中的初始字节，将它分为4000份。

我们针对该基准的分区函数是建立在key分配的相关知识之上。在一个普通的排序程序中，我们会在里面添加一个预传递的MapReduce操作，该操作用来收集样本键，并使用所采样的样本键的分布情况来计算出最终排序过程中的分割点。

Figure 3(a)中向我们展示了该排序程序的一次正常执行过程。左上方的图表示的是输入文件读取的速率。这张图上的峰值速率大约为13GB/s，由于所有的Map任务都在200秒前完成，所以它的速度很快就降了下来，变成了0。注意，此处读取输入文件的速率要小于grep中的速率。这是因为排序Map任务花费了大约一半的时间和I/O带宽将中间输出写入它们的本地硬盘中。grep所对应的中间数据的大小可以忽略不计。

中间左边的图像展示的是map任务所产生的数据通过网络传送给reduce任务时的速率。一旦第一个map任务完成后，shuffle就会开始。图中第一个驼峰处代表的是第一批大概1700个Reduce任务（整个MapReduce计算会被分配给1700机器，每台机器最多一次只会执行一个reduce任务）。大概执行计算300秒后，第一批reduce任务中的部分任务会完成，接着，我们对剩下的reduce任务进行shuffle。shuffle总共花了600秒完成。

左下方的图像表示的是Reduce任务将排序后的数据写入到最终的输入文件时的速度。在第一次shuffle阶段结束后和写入阶段开始前，它们之间存在了延迟。因为机器会忙于对中间数据进行排序。硬盘的写入速度会处于2-4GB/s，并且这个速度会持续一段时间。所有的写入需要花费850秒才能完成。包括启动开销，整个计算过程花费了891秒。这个速度和TeraSort benchmark中的当前最佳纪录差不多。

还有一些事情值得注意。输入速度要比shuffle速度和输出速度高得多，这是要归功于我们的地区性优化。大部分数据都是从本地硬盘中读取，这样就避开了通过受限的网络进行数据传输了。shuffle速度要比输出速度高不少。原因是因为输出阶段要输出排序后数据的两份备份（为了保证可靠性和可用性，我们要有两份输出备份）。我们要写入两个备份的原因是，我们的底层文件系统提供了可靠性和可用性的机制。如果底层文件系统使用类似容错编码的方式而不是复制的方式来保证数据的可靠性和可用性，那么在输出数据写入硬盘的时候，就可以降低网络带宽的使用。

5.4 高效的Backup任务

在Figure 3 (b)中，我们展示了一个禁用备用任务的排序任务的执行过程。该执行过程和Figure 3(a)中所示的类似，除了最后这段时间的尾巴很长，几乎没有任何写入活动发生。在960秒后，除了最后5个reduce任务以外，其他任务全部完成了。然而，这些最后几个拖后腿的任务花了300秒才完成。整个计算过程花了1283秒，多了44%的执行时间。

5.5 机器故障

在Figure 3(c)中演示的排序程序里，我们在程序开始后的几分钟内故意砍掉了1746个worker中的200个worker。底层集群调度器会立即在这些机器上重启新的worker进程（这些机器依然能够正常运行，只是worker进行被关闭）。

worker进程的关闭会在图中显示一个负的输入速度，这是因为之前一些已经完成的map任务丢失所导致的，这些丢失的map任务需要被再次执行。重新执行这种Map任务的速度会相对较快。整个计算任务包括启动开销在内，花了933秒完成（只比正常执行时间多用了5%的时间）。

6 经验

我们在2003年2月完成了MapReduce库的第一个版本，并且在2003年8月份我们对它进行了显著的优化，这其中包括了地区性优化，worker机器之间任务执行的动态负载均衡等等。从那时起我们发现，MapReduce库能广泛地应用于我们所遇见的各种问题。谷歌已经将其应用于非常多的领域之上：

大规模机器学习问题

谷歌新闻以及Froogle产品的集群问题

提取用于生成流行查询报告的数据（例如，谷歌的Zeitgeis）

从新实验和产品的网页中提取相关属性（例如从本地化搜索的语料库中提取位置信息）

以及大规模图计算问题

Figure 4 显示了在我们的主要源代码管理系统中，随着时间的推移，MapReduce程序在数量上显著的增长。从2003年一开始的0个MapReduce程序，到2004年9月已经有了差不多900个不同的MapReduce程序实例。MapReduce是如此的成果，因为它使得编写一个简单的程序并在半小时内在一千台计算机上高效运行它成为可能，从而极大地加快了开发和原型开发周期。此外，它允许没有分布式或者并行系统经验的程序员轻松地去利用这些资源。

在每项job完成后，MapReduce库会以日志的方式记录下该job所利用的计算资源的数据。在Table 1中，我们展示了谷歌在2004年8月所运行的MapReduce job中某个子集的运行数据。

6.1 大规模索引

到目前为止，MapReduce最成功的应用那就是重写了谷歌网络搜索服务所使用到的用来生成数据结构的索引系统。索引系统会将我们抓取系统所检索到的大量文档作为输入，并保存到一组GFS文件中。这些文档的原始内容的数据大小超过20TB。在索引过程中，通常会运行5-10次MapReduce操作。使用MapReduce已经为我们提供了以下便利（与之前版本中所使用的分布式索引系统相比）：

索引相关的代码会更加简单，小巧并且易于理解。因为那些用于处理容错，分布式以及并行的代码已经隐藏在MapReduce库中，所以无需去担心这些。例如，使用MapReduce库，处理计算相关的C++代码从原来的3800行减少到了大概700行代码。

MapReduce库的性能已经足够好了，这样我们可以将概念上无关的计算分离，而不是将它们全混在一起，来避免对数据的额外传递。这使得我们容易去改变索引处理过程。例如，原本在我们旧的索引系统中要去做一处修改需要花数月才能完成，但在新系统中只需花几天就能实现。

索引处理过程执行起来变得更加容易，因为大部分由机器故障，机器缓慢，网络瞬间阻塞所引发的问题已经自动被MapReduce库所解决了，这也不需要操作人员的介入了。此外，通过在索引集群中增加新的机器，这可以很容易的提升索引过程的性能。

7. 相关工作

许多系统都提供了受限制的编程模型，通过使用这些限制来让计算自动并行化执行。例如，一个函数可以把它的所有前缀放入一个长度为N的数组内，使用并行前缀计算，将它分布在N个处理器上，在log N时间内完成计算。MapReduce可以看做是根据我们在现实世界中大型计算的相关经验所得到的模型，对这些模型进行简化和精炼的成果。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，大部分并行处理系统的实现只适用于小规模场景使用，并且它们将处理机器故障的细节交由程序员处理。

Bulk同步编程以及一些MPI（消息传递接口）原语提供了更高级的抽象，这使得程序员能更容易得写出并行程序。这些系统和MapReduce之间的主要区别在于MapReduce利用受限制的编程模型来自动并行化用户程序，并且还提供了透明的容错能力。

我们的局部性优化从诸如活动磁盘[12,15]之类的技术中获得灵感，在这些技术中，计算被推到靠近本地磁盘的处理元素中，以减少通过I/O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品处理器上，而不是直接在磁盘控制器处理器上运行，但一般的方法是相似的。

我们的备用任务机制和Charlotte系统中提供的eager调度机制相似。Eager调度机制的一个缺点就是，如果一个任务反复失败，那么整个计算任务就无法完成。在我们的机制中，通过跳过有问题的记录，以此在某种程度上修复这种问题。

MapReduce的实现依赖于内部的集群管理系统。该系统负责在一个超大的共享机器的集群中分发和运行用户任务。虽然这并不是本文的重点，但是本质上来讲，该系统和Condor等其他系统类似。

排序机制是MapReduce库中的一部分，它和NOW-Sort中的操作类似。源机器（也就是map worker）将要排序的数据进行分区，然后将这些输入发送给R个reduce worker中的一个去处理。每个reduce worker会在本地将数据进行排序。当然，NOW-Sort并没有用户所定义的Map和Reduce函数，这也使得我们的库运用范围更加广泛。

River提供了一种编程模型，在该模型中，进程通过在分布式队列上发送数据来相互通信。和MapReduce类似，即使存在硬件故障或者系统问题，River系统也能提供良好的平均性能。River通过对磁盘和网络通讯进行仔细的调度，以此来平衡任务的完成时间。MapReduce则采用了不同的方法。通过限制编程模型，MapReduce框架能够将一个问题拆分成很多粒度良好的任务。这些任务会动态调度给可用的worker，这样执行速度快的worker就能处理更多的工作（能者多劳）。这种受限制的编程模型也允许我们在任务快要结束时安排备用任务，这大大减少了在配置不均衡的情况下（例如，又卡又慢的机器拖累整个进度）的任务完成时间。

BAD-FS则有一个和MapReduce非常不同的编程模型。与MapReduce不同，它是针对面向广域网的任务执行。但是，它们有两个基本相似点。(1)它们都使用冗余执行来恢复因为数据丢失而导致的任务失败。(2)它们都是用了数据本地化调度策略，减少通过拥堵的网络发生的数据量。

TACC是一个用于简化构建高可用网络服务的系统。和MapReduce一样，它也依靠重新执行机制来实现容错处理。

8. 总结

谷歌已经将MapReduce编程模型成功应用于许多不同的场景。我们将此成功归因于许多原因。首先，该模型易于使用，即便是没有并行和分布式系统经验的程序员也可以轻松使用。因为它隐藏了并行，容错，局部优化，负载均衡这些细节。其次，很大一部分问题都能以MapReduce计算进行表达。例如，在谷歌的许多产品中（例如，网页搜索服务、排序、数据挖掘、机器学习以及其他系统），MapReduce被用于生成数据。第三点则是我们已经开发了一种MapReduce的实现。该实现可以扩展应用到一个包含数千台机器的集群之上。该实现可以有效使用这些机器资源。因此，它适用于谷歌所遇到的许多大型计算问题。

从这项工作中我们学到了以下几点。首先，通过限制编程模型可以很容易进行并行和分布式计算，这样可以使这种计算具备容错性。其次，网络带宽是一种稀缺资源。因此，在我们的系统中存在着许多优化。主要针对于减少通过网络发送的数据量：其中，位置优化可以让我们能够从本地磁盘中读取数据，并且在本地磁盘中写入一份中间数据的副本，以此来节省网络带宽。第三，可以使用冗余执行来减少速度比较慢的机器所带来的影响，并且可以处理计算机故障和数据丢失。