# MapReduce：大型集群上的简化数据处理

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com，sanjay@google.com

*谷歌公司*

## 摘要

MapReduce 是一种编程模型以及与之相关的实现方式，用于处理和生成大型数据集。用户指定一个*映射*函数，该函数处理键/值对以生成一组中间键/值对，还指定一个归约函数，用于合并所有与同一中间键相关联的中间值。正如论文中所展示的，许多实际任务都可以用这种模型来表达。

用这种函数式风格编写的程序会自动并行化，并在大量商品机器组成的集群上执行。运行时系统会处理输入数据的分区、在一组机器上调度程序执行、处理机器故障以及管理所需的机器间通信等细节。这使得没有并行和分布式系统经验的程序员也能轻松利用大型分布式系统的资源。

我们实现的 MapReduce 系统运行在由大量普通商用机器组成的集群上，具有高度的可扩展性：一个典型的 MapReduce 计算任务每天会在数千台机器上处理数太字节的数据。程序员发现该系统易于使用：已经实现了数百个 MapReduce 程序，并且每天在谷歌的集群上执行的 MapReduce 任务多达一千个。

## 1 简介

在过去的五年里，作者和谷歌的许多其他人员已经实现了数百种专门用途的计算，这些计算处理大量的原始数据，例如抓取的文档、网络请求日志等，以计算各种派生数据，比如倒排索引、网络文档图结构的各种表示、每个主机抓取的页面数量汇总、给定一天中最频繁的查询集合等等。大多数此类计算在概念上都很简单。然而，输入数据通常规模庞大，为了在合理的时间内完成计算，这些计算必须在数百或数千台机器上进行分布式处理。如何并行化计算、分配数据以及处理故障等问题交织在一起，使得原本简单的计算被大量复杂的代码所掩盖，这些代码用于处理这些问题。

针对这种复杂性，我们设计了一种新的抽象概念，它能够让我们表达出想要执行的简单计算，同时将并行化、容错、数据分布和负载均衡等复杂细节隐藏在一个库中。我们的抽象概念受到了 Lisp 和许多其他函数式语言中 *map* 和 *reduce* 原语的启发。我们意识到，我们的大多数计算都涉及对输入中的每个逻辑“记录”应用 *map* 操作以计算一组中间的键/值对，然后对具有相同键的所有值应用 *reduce* 操作，以便适当地组合派生的数据。通过使用用户指定的 map 和 reduce 操作的函数模型，我们能够轻松地并行化大型计算，并将重新执行作为主要的容错机制。

这项工作的主要贡献在于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式处理，同时实现了该接口在由大量商用 PC 组成的集群上达到高性能。

第 2 节描述了基本的编程模型，并给出了几个示例。第 3 节介绍了针对我们的集群计算环境定制的 MapReduce 接口实现。第 4 节描述了我们认为有用的编程模型的若干改进。第 5 节给出了我们针对各种任务的实现性能测量结果。第 6 节探讨了 MapReduce 在谷歌内部的应用，包括我们将其作为基础所获得的经验。

关于对我们的生产索引系统的重写。第 7 节讨论了相关工作和未来的工作。

## 2 编程模型

该计算处理一组*输入*的键值对，并生成一组*输出*的键值对。使用 MapReduce 库的用户将计算表达为两个函数：*Map* 函数和 *Reduce 函数*。

由用户编写的 *Map* 函数接收一个输入对，并生成一组*中间*键值对。MapReduce 库将所有与同一个中间键 I 相关联的中间值组合在一起，并将它们传递给 *Reduce* 函数。

用户编写的 *Reduce* 函数接受一个中间键 I 以及该键的一组值。它将这些值合并在一起，形成一个可能更小的值集。通常，每次调用 *Reduce* 函数只会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。这使我们能够处理那些太大而无法放入内存的值列表。

### 2.1 示例

考虑这样一个问题：计算大量文档中每个单词出现的次数。用户会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 函数会输出每个单词及其出现次数（在这个简单的示例中，每次出现的计数都是“1”）。reduce 函数会将针对特定单词输出的所有计数相加。

此外，用户编写代码来填充一个 *MapReduce 规范*对象，其中包含输入和输出文件的名称以及可选的调优参数。然后，用户调用 *MapReduce* 函数，并将规范对象传递给它。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

尽管前面的伪代码是以字符串输入和输出的形式编写的，但从概念上讲，用户提供的映射和归约函数都有相关的类型：

```
map     (k1,v1)          → list(k2,v2)
reduce  (k2,list(v2))    → list(v2)
```

也就是说，输入的键和值与输出的键和值来自不同的域。此外，中间的键和值与输出的键和值来自相同的域。

我们的 C++ 实现将字符串传递给用户定义的函数，并从这些函数接收字符串，而将字符串与适当类型的转换工作留给用户代码来完成。

### 2.3 更多示例

这里有几个简单的有趣程序示例，它们可以很容易地用 MapReduce 计算来表达。

**分布式 Grep：**如果 map 函数匹配到提供的模式，则输出一行。reduce 函数是一个恒等函数，只是将提供的中间数据复制到输出中。

**URL 访问频率统计**：map 函数处理网页请求日志，并输出 hURL， 1i。reduce 函数将相同 URL 的所有值相加，并输出 hURL， 总计数i 对。

**反向网页链接图：**映射函数为每个在名为 source 的页面中找到的指向目标 URL 的链接输出 htarget, sourcei 对。归约函数将与给定目标 URL 相关联的所有 source URL 列表连接起来，并输出对：htarget, list(source)i

**每个主机的词向量**：词向量总结了文档或文档集中出现的最重要的词，以“词，频率”对的形式列出。映射函数为每个输入文档输出一个“主机名，词向量”对（其中主机名从文档的 URL 中提取）。归约函数接收给定主机的所有文档词向量。它将这些词向量相加，舍弃出现频率低的词，然后输出一个最终的“主机名，词向量”对。

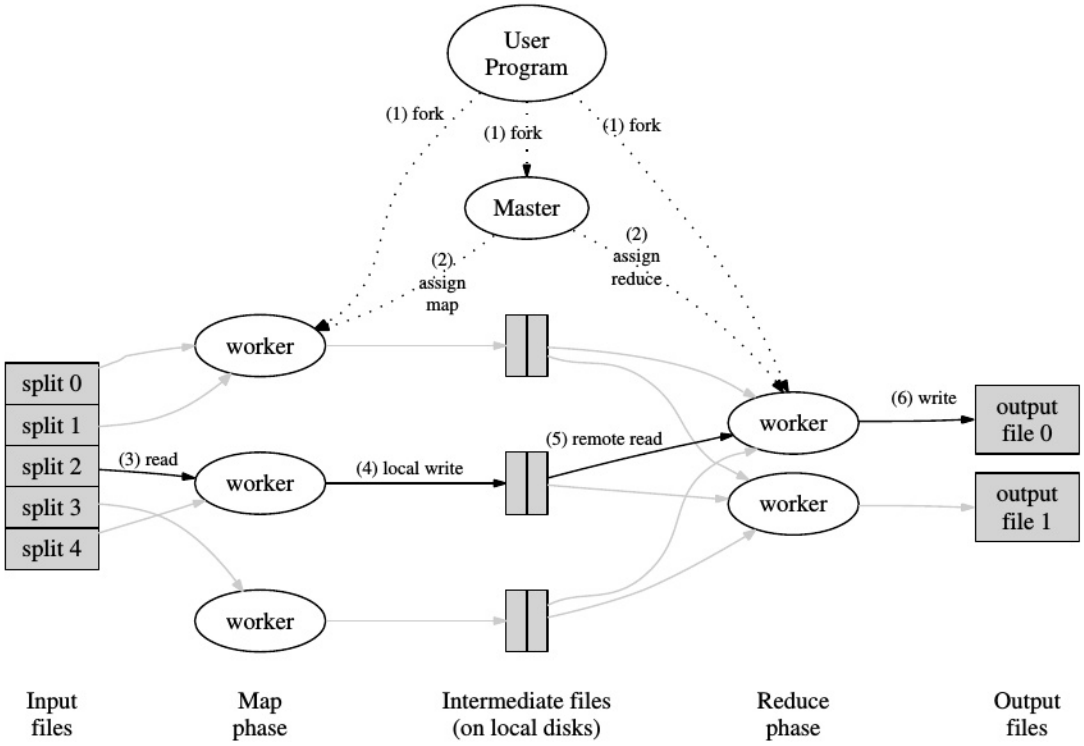


图 1：执行概览

**倒排索引：**映射函数解析每个文档，并输出一系列由单词和文档 ID 组成的键值对。归约函数接收给定单词的所有键值对，对相应的文档 ID 进行排序，并输出由单词和文档 ID 列表组成的键值对。所有输出的键值对集合构成一个简单的倒排索引。很容易对这种计算进行扩展，以跟踪单词的位置。

**分布式排序：**映射函数从每条记录中提取键，并输出一个键值对 <hkey, recordi>。归约函数将所有键值对原样输出。此计算依赖于第 4.1 节所述的分区设施以及第 4.2 节所述的排序特性。

## 3 实施

MapReduce 接口有许多不同的实现方式。正确选择取决于环境。例如，一种实现方式可能适用于小型共享内存机器，另一种适用于大型非统一内存访问（NUMA）多处理器，还有一种适用于更大规模的联网机器集群。

本节描述了一种针对谷歌广泛使用的计算环境的实现方案：

大量通过交换式以太网连接在一起的商用 PC 集群[4]。在我们的环境中：

机器通常是双处理器的 x86 处理器，运行 Linux 操作系统，每台机器配备 2 至 4GB 内存。

（2）通常使用商品化的网络硬件——在机器层面一般为每秒 100 兆位或每秒 1 千兆位，但整体的二分带宽平均要低得多。

（3）一个集群由数百台或数千台机器组成，因此机器故障是常见的。

（4）存储由直接连接到各台机器上的廉价 IDE 硬盘提供。内部开发的一种分布式文件系统[8]用于管理存储在这些硬盘上的数据。该文件系统通过复制来在不可靠的硬件之上提供可用性和可靠性。

（5）用户向调度系统提交作业。每个作业都包含一组任务，并由调度器映射到集群内的一组可用机器上。

### 3.1 执行概览

通过自动划分输入数据，*Map* 调用被分配到多台机器上。

将其划分为 M 个输入分*片*。这些输入分片可以由不同的机器并行处理。通过使用分区函数（例如，hash (key) **mod** R）将中间键空间划分为 R 个部分来分配 *Reduce* 调用。分区的数量（R）和分区函数由用户指定。

图 1 展示了我们实现中 MapReduce 操作的整体流程。当用户程序调用 MapReduce 函数时，会按以下顺序执行一系列操作（图 1 中的编号标签与下面列表中的编号相对应）：

1. 用户程序中的 MapReduce 库首先将输入文件分割成 M 个大小通常为 16 兆字节到 64 兆字节（MB）的片段（用户可通过一个可选参数进行控制）。然后，它在一组机器上启动该程序的多个副本。

2. 其中一个程序副本是特殊的——即主节点。其余的是由主节点分配任务的工作者。有 M 个映射任务和 R 个归约任务需要分配。主节点挑选空闲的工作者，并给每个工作者分配一个映射任务或一个归约任务。

3. 被分配了映射任务的工作者会读取相应输入分片的内容。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*函数生成的中间键值对会在内存中进行缓冲。

4. 定期地，缓冲区中的键值对会被写入本地磁盘，并通过分区函数划分为 R 个区域。这些键值对在本地磁盘上的位置会被反馈给主节点，主节点负责将这些位置转发给归约工作节点。

5. 当一个归约工作进程从主节点收到这些位置的通知后，它会使用远程过程调用从映射工作进程的本地磁盘读取缓冲数据。当一个归约工作进程读取完所有中间数据后，它会按照中间键对其进行排序，以便将所有相同的键归为一组。排序是必要的，因为通常许多不同的键会映射到同一个归约任务。如果中间数据量过大而无法放入内存，则会使用外部排序。

6. 归约工作进程会对排序后的中间数据进行迭代，对于遇到的每个唯一的中间键，它会将该键以及对应的中间值集合传递给用户的*归*约函数。*归*约函数的输出会被追加到此归约分区的最终输出文件中。

7. 当所有映射任务和归约任务都已完成时，主节点会唤醒用户程序。此时，用户程序中的 MapReduce 调用将返回到用户代码。

成功完成后，MapReduce 执行的输出结果将保存在 R 输出文件中（每个归约任务一个文件，文件名由用户指定）。通常，用户无需将这些 R 输出文件合并为一个文件——他们常常将这些文件作为输入传递给另一个 MapReduce 调用，或者在能够处理分隔为多个文件的输入的其他分布式应用程序中使用它们。

## 3.2 掌握数据结构

主节点维护着几个数据结构。对于每个映射任务和归约任务，它都存储着任务的状态（空闲、*进行中*或已*完成*），以及执行任务的从节点机器的身份（对于非空闲任务）。

主节点是中间文件区域位置从映射任务传播到归约任务的通道。因此，对于每个已完成的映射任务，主节点都会存储该映射任务生成的 R 个中间文件区域的位置和大小。随着映射任务的完成，会收到对此位置和大小信息的更新。这些信息会逐步推送给正在*进行*归约任务的工作者节点。

## 3.3 容错性

由于 MapReduce 库旨在帮助使用数百台或数千台机器处理海量数据，因此该库必须能够优雅地应对机器故障。

### 工人故障

主节点会定期向每个工作节点发送心跳请求。如果在一定时间内未收到某个工作节点的响应，主节点就会将其标记为故障。该工作节点已完成的任何映射任务都会被重置为初始的*空闲*状态，从而有资格在其他工作节点上重新调度。同样，故障工作节点上正在进行的任何映射任务或归约任务也会被重置为*空闲*状态，并有资格重新调度。

已完成的映射任务在出现故障时会重新执行，因为它们的输出存储在故障机器的本地磁盘上，因而无法访问。已完成的归约任务则无需重新执行，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节点 A 执行，之后又由工作节点 B 执行（因为 A 失败了），所有

执行归约任务的工作者会收到重新执行的通知。任何尚未从工作者 A 读取数据的归约任务都将从工作者 B 读取数据。

MapReduce 具有应对大规模工作节点故障的能力。例如，在一次 MapReduce 操作期间，正在运行的集群进行网络维护，导致每次有 80 台机器在几分钟内无法访问。MapReduce 主节点只是重新执行了那些无法访问的工作节点所完成的工作，并继续向前推进，最终完成了 MapReduce 操作。

### 主故障

要让主节点定期写入上述主数据结构的检查点很容易。如果主任务终止，可以从最后一个检查点的状态重新启动一个新的副本。不过，由于只有一个主节点，其出现故障的可能性不大；因此，我们当前的实现方式是，如果主节点出现故障，就终止 MapReduce 计算。客户端可以检查这种情况，如果需要，可以重新尝试 MapReduce 操作。

### 故障存在情况下的语义学

当用户提供的*映射*和*归*约操作符是其输入值的确定性函数时，我们的分布式实现所产生的输出与整个程序在无故障的顺序执行中所产生的输出相同。

我们依靠对映射和归约任务输出的原子提交来实现这一特性。每个正在运行的任务将其输出写入私有临时文件。归约任务生成一个这样的文件，而映射任务生成 R 个这样的文件（每个归约任务一个）。当映射任务完成时，工作节点会向主节点发送一条消息，并在消息中包含 R 个临时文件的名称。如果主节点收到已完成映射任务的完成消息，则忽略该消息。否则，它会在主节点的数据结构中记录 R 个文件的名称。

当一个归约任务完成时，归约工作进程会将它的临时输出文件原子性地重命名为最终输出文件。如果同一个归约任务在多台机器上执行，针对同一个最终输出文件将会有多个重命名调用。我们依靠底层文件系统提供的原子性重命名操作来保证最终的文件系统状态仅包含归约任务执行一次所产生的数据。

我们的大多数*映射*和*归*约操作符都是确定性的，这种情况下我们的语义等同于顺序执行，这使得程序员能够非常轻松地推断出程序的行为。当*映射*和/或*归*约操作符是非确定性的，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，特定归约任务 R1 的输出等同于非确定性程序顺序执行时 R1 的输出。然而，对于不同的归约任务 R2，其输出可能对应于非确定性程序的不同顺序执行时 R2 的输出。

考虑映射任务 M 和归约任务 R1 和 R2。设 e（Ri） 为 Ri 的执行过程（恰好存在这样一个执行过程）。较弱的语义出现是因为 e（R1） 可能读取了 M 的某次执行所生成的输出，而 e（R2） 可能读取了 M 的另一次执行所生成的输出。

## 3.4 本地性

在网络计算环境中，网络带宽是一种相对稀缺的资源。我们利用输入数据（由 GFS [8] 管理）存储在构成集群的各台机器的本地磁盘这一事实来节省网络带宽。GFS 将每个文件分成 64MB 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 主节点会考虑输入文件的位置信息，并尝试将映射任务调度到包含相应输入数据副本的机器上。如果无法做到这一点，它会尝试将映射任务调度到靠近该任务输入数据副本的机器上（例如，在与包含数据的机器位于同一网络交换机的作业节点上）。当在集群中相当一部分作业节点上运行大型 MapReduce 操作时，大多数输入数据都是本地读取的，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，将归约阶段细分为 R 个部分。理想情况下，M 和 R 应远大于工作机器的数量。让每个工作机器执行许多不同的任务有助于动态负载均衡，而且当某个工作机器出现故障时，还能加快恢复速度：它已完成的众多映射任务可以分配到所有其他工作机器上。

在我们的实现中，M 和 R 的大小存在实际限制，因为主节点必须做出 O（M + R） 次调度决策，并且如上所述在内存中保留 O（M * R） 的状态。（不过，内存使用的常量因子很小：状态中的 O（M * R） 部分大约为每个映射任务/归约任务对占用 1 字节的数据。）

此外，R 常常受到用户的限制，因为每个归约任务的输出最终都保存在一个单独的输出文件中。在实际操作中，我们倾向于选择 M 的值，使得每个单独的任务处理大约 16MB 到 64MB 的输入数据（这样上述的本地性优化才能发挥最大效果），并且我们让 R 等于预期使用的工人机器数量的一个小倍数。我们经常使用 M = 200,000 和 R = 5,000，配合 2,000 台工人机器来进行 MapReduce 计算。

## 3.6 备份任务

MapReduce 操作总耗时延长的一个常见原因是“拖后腿者”：即在计算的最后几个 Map 或 Reduce 任务中，有一台机器耗时异常长。拖后腿者出现的原因多种多样。例如，磁盘有问题的机器可能会频繁出现可纠正错误，导致其读取性能从每秒 30MB 降至每秒 1MB。集群调度系统可能会在该机器上安排其他任务，由于 CPU、内存、本地磁盘或网络带宽的竞争，导致其执行 MapReduce 代码的速度变慢。最近我们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下降了超过一百倍。

我们有一个通用机制来缓解“落后者”问题。当 MapReduce 操作接近完成时，主节点会安排剩余未完成任务的备份执行。只要主执行或备份执行中的任何一个完成，该任务就会被标记为已完成。我们对该机制*进行*了调整，使其通常只会使操作所使用的计算资源增加几个百分点。我们发现，这显著减少了大型 MapReduce 操作的完成时间。例如，第 5.3 节中描述的排序程序在禁用备份任务机制时，完成时间会延长 44%。

# 4 改进措施

尽管仅通过编写 *Map* 和 *Reduce* 函数所提供的基本功能已能满足大多数需求，但我们发现一些扩展还是很有用的。这些扩展将在本节中进行描述。

## 4.1 分区功能

MapReduce 的用户会指定他们所需的归约任务/输出文件的数量（R）。中间键通过分区函数在这些任务之间进行分区。默认的分区函数使用哈希算法（例如“hash(key) **mod** R”）。这通常会导致分区较为均衡。但在某些情况下，根据键的其他函数来分区会更有用。例如，有时输出键是 URL，我们希望来自同一主机的所有条目都出现在同一个输出文件中。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“hash(Hostname(urlkey)) **mod** R”作为分区函数，会导致来自同一主机的所有 URL 都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，中间的键/值对将按照键的升序进行处理。这种排序保证使得为每个分区生成一个排序后的输出文件变得容易，这在输出文件格式需要支持通过键进行高效的随机访问查找，或者输出文件的使用者觉得数据已排序更方便时很有用。

## 4.3 组合器函数

在某些情况下，每个 Map 任务生成的中间键存在大量重复，且用户指定的 *Reduce* 函数具有交换性和结合性。第 2.1 节中的单词计数示例就是一个很好的例子。由于单词频率通常遵循齐普夫分布，每个 Map 任务都会生成数百或数千条形如 <the, 1> 的记录。所有这些计数都将通过网络发送到单个 Reduce 任务，然后由 *Reduce* 函数相加得出一个数字。我们允许用户指定一个可选的 *Combiner* 函数，在数据通过网络发送之前对其进行部分合并。

*组合器*函数在执行映射任务的每台机器上运行。通常，实现组合器函数和归约函数所使用的代码是相同的。归约函数和组合器函数之间的唯一区别在于 MapReduce 库如何处理函数的输出。归约函数的输出会被写入最终的输出文件。组合器函数的输出会被写入一个中间文件，该文件将被发送到归约任务。

部分合并能显著加快某些类别的 MapReduce 操作。附录 A 包含了一个使用合并器的示例。

## 4.4 输入和输出类型

MapReduce 库提供了以多种不同格式读取输入数据的支持。例如，“文本”

模式输入将每一行视为键/值对：键是文件中的偏移量，值是该行的内容。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自身拆分为有意义的范围，以便作为单独的映射任务进行处理（例如，文本模式的范围拆分确保范围拆分仅在行边界处发生）。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，不过大多数用户只是使用少量预定义的输入类型之一。

读取*器*不一定需要提供从文件读取的数据。例如，很容易定义一个读取*器*，使其从数据库或映射到内存的数据结构中读取记录。

同样地，我们支持一组输出类型，以便以不同的格式生成数据，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 的用户发现从其映射和/或归约操作中生成辅助文件作为额外输出很方便。我们依靠应用程序编写者确保此类副作用具有原子性和幂等性。通常应用程序会将数据写入临时文件，并在文件完全生成后将其原子性地重命名。

我们不支持单个任务生成的多个输出文件的原子两阶段提交。因此，生成多个具有跨文件一致性要求的输出文件的任务应是确定性的。在实际应用中，这一限制从未成为问题。

## 4.6 跳过错误记录

有时用户代码中存在一些错误，会导致 *Map* 或 *Reduce* 函数在处理某些记录时确定性地崩溃。此类错误会阻止 MapReduce 操作完成。通常的做法是修复错误，但有时这不可行；也许错误出现在无法获取源代码的第三方库中。此外，有时忽略少数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 库会检测出导致确定性崩溃的记录，并跳过这些记录以继续向前推进。

每个工作进程都会安装一个信号处理程序，用于捕获段错误和总线错误。在调用用户定义的 *Map* 或 *Reduce* 操作之前，MapReduce 库会将参数的序列号存储在一个全局变量中。如果用户代码产生信号，信号处理程序会发送一个包含序列号的“临终遗言”UDP 数据包给 MapReduce 主节点。当主节点发现某个特定记录出现多次故障时，在重新执行相应的 Map 或 Reduce 任务时，它会指示跳过该记录。

## 4.7 本地执行

在 *Map* 或 *Reduce* 函数中调试问题可能会很棘手，因为实际计算是在分布式系统中进行的，通常涉及数千台机器，并且工作分配决策由主节点动态做出。为了帮助进行调试、性能分析和小规模测试，我们开发了一种 MapReduce 库的替代实现方式，它会在本地机器上按顺序执行 MapReduce 操作的所有工作。为用户提供了控制选项，以便将计算限制在特定的映射任务上。用户通过使用一个特殊的标志来调用其程序，然后就可以轻松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

主节点运行一个内部 HTTP 服务器，并导出一组供人类查看的状态页面。这些状态页面展示了计算的进度，例如已完成的任务数量、正在进行的任务数量、输入字节数、中间数据字节数、输出字节数、处理速率等。页面中还包含每个任务生成的标准错误和标准输出文件的链接。用户可以利用这些数据预测计算所需的时间，以及是否应为计算添加更多资源。这些页面还可以用于判断计算速度是否比预期慢得多。

此外，顶级状态页面会显示哪些工作节点出现故障，以及它们在出现故障时正在处理哪些映射和归约任务。这些信息在尝试诊断用户代码中的错误时非常有用。

## 4.9 计数器

MapReduce 库提供了一个计数器功能，用于统计各种事件的发生次数。例如，用户代码可能想要统计处理的单词总数或索引的德文文档数量等。

要使用此功能，用户代码需创建一个命名的计数器对象，然后在 *Map* 和/或 *Reduce* 函数中对其进行适当的递增操作。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");
```

各个工作节点的计数器值会定期传播到主节点（附带在 ping 响应中）。主节点会汇总成功完成的映射和归约任务的计数器值，并在 MapReduce 操作完成后将其返回给用户代码。当前的计数器值也会显示在主节点的状态页面上，以便人工查看正在进行的计算进度。在汇总计数器值时，主节点会消除同一映射或归约任务重复执行的影响，以避免重复计数。（重复执行可能源于我们对备份任务的使用以及由于任务失败而重新执行任务的情况。）

某些计数器值由 MapReduce 库自动维护，例如处理的输入键值对的数量以及生成的输出键值对的数量。

用户发现计数器功能对于检查 MapReduce 操作的行为是否正常非常有用。例如，在某些 MapReduce 操作中，用户代码可能希望确保生成的输出键值对数量恰好等于处理的输入键值对数量，或者处理的德文文档数量占处理的文档总数的比例在可接受的范围内。

# 5 性能

在本节中，我们将在一个大型机器集群上运行的两个计算任务来衡量 MapReduce 的性能。其中一个计算任务是在大约 1TB 的数据中搜索特定模式，另一个计算任务是对大约 1TB 的数据进行排序。

这两个程序代表了 MapReduce 用户编写的大量实际程序中的一大部分——一类程序将数据从一种表示形式转换为另一种表示形式，另一类程序从大量数据集中提取少量有趣的数据。

## 5.1 集群配置

所有程序均在一个由约 1800 台机器组成的集群上执行。每台机器配备两颗 2GHz 英特尔至强处理器（启用了超线程技术）、4GB 内存以及两个 160GB 的 IDE 硬盘。

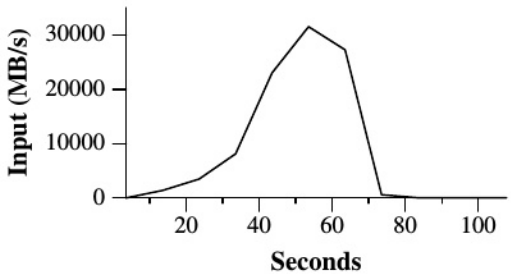


图 2：随时间变化的数据传输速率

这些机器配备了磁盘和千兆以太网链路。它们被布置成一个两级树形交换网络，在根节点处的总带宽约为 100 至 200 Gbps。所有机器都在同一托管设施内，因此任意两台机器之间的往返时间都不到一毫秒。

在 4GB 内存中，约有 1 - 1.5GB 被集群上运行的其他任务占用。这些程序是在周末下午执行的，当时 CPU、磁盘和网络大多处于空闲状态。

## 5.2 Grep

*grep* 程序扫描 10 亿条 100 字节的记录，查找一个相对罕见的三个字符的模式（该模式出现在 92337 条记录中）。输入被分割成大约 64MB 的块（M = 15000），而整个输出被放置在一个文件中（R = 1）。

图 2 展示了计算随时间推进的情况。Y 轴表示输入数据的扫描速率。随着分配给此 MapReduce 计算的机器增多，扫描速率逐渐上升，在分配了 1764 个工作节点时达到峰值，超过 30GB/秒。随着映射任务的完成，扫描速率开始下降，并在计算开始约 80 秒时降至零。整个计算过程从开始到结束大约需要 150 秒，其中包括约 1 分钟的启动开销。开销源于程序向所有工作节点的传播，以及与 GFS 交互打开 1000 个输入文件和获取本地性优化所需信息的延迟。

## 5.3 排序

该*排序*程序对 100 字节的 10 亿条记录进行排序（约 1 太字节的数据）。此程序是基于 TeraSort 基准测试 [10] 设计的。

该排序程序包含不到 50 行用户代码。一个三行的 *Map* 函数从文本行中提取出 10 字节的排序键，并输出该键以及

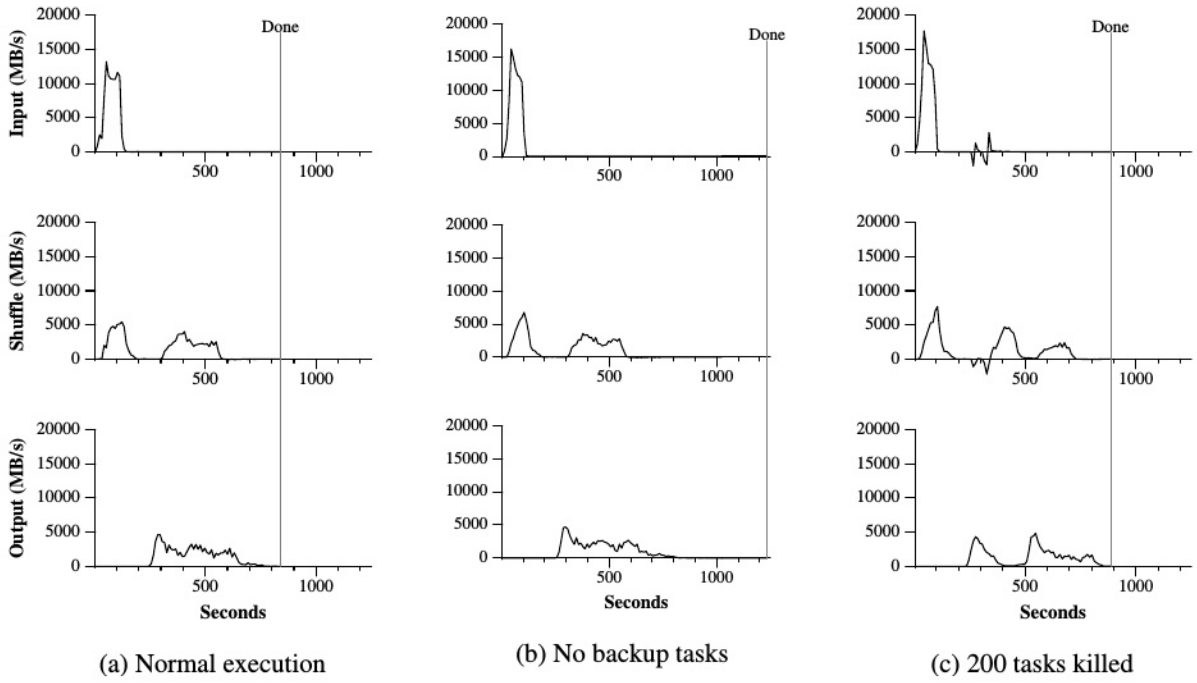


图 3：排序程序不同执行过程中随时间变化的数据传输速率

我们将原始文本行作为中间键/值对。我们使用了一个内置的*恒*等函数作为*归*约操作符。该函数将中间键/值对原封不动地作为输出键/值对传递。最终排序后的输出被写入一组两份复制的 GFS 文件（即，2TB 的数据作为程序的输出被写入）。

和之前一样，输入数据被分割成 64MB 的块（M = 15000）。我们把排序后的输出分成 4000 个文件（R = 4000）。分区函数使用键的初始字节将其划分到 R 个部分中的一个。

我们为该基准测试所设计的分区函数内置了对键值分布的了解。而在一般的排序程序中，我们会添加一个预处理的 MapReduce 操作，用于收集键值样本，并根据样本键值的分布情况来计算最终排序阶段的分割点。

图 3（a）展示了排序程序正常执行的进程。左上角的图表显示了输入读取速率。该速率在约 13GB/秒时达到峰值，然后迅速下降，因为所有映射任务在 200 秒内就已完成。请注意，输入速率低于 *grep* 程序。这是因为排序映射任务大约有一半的时间和 I/O 带宽用于将中间输出写入本地磁盘。而 grep 程序对应的中间输出大小可以忽略不计。

中间偏左的图表展示了从映射任务向归约任务传输数据的速率。这种数据重组操作在第一个映射任务完成后即开始。图表中的第一个峰值对应于第一批约 170 个归约任务（整个 MapReduce 作业分配到了约 1700 台机器，每台机器一次最多执行一个归约任务）。大约在计算开始 300 秒时，第一批归约任务中的一部分完成，于是我们开始为剩余的归约任务重组数据。所有数据重组操作在计算开始约 600 秒时完成。

左下角的图表展示了归约任务将排序后的数据写入最终输出文件的速率。在第一个数据混洗阶段结束与开始写入阶段之间存在一段延迟，这是因为机器正忙于对中间数据进行排序。写入操作以约 2 至 4GB/秒的速度持续了一段时间。所有写入操作在计算开始约 850 秒时完成。包括启动开销在内，整个计算过程耗时 891 秒。这与目前 TeraSort 基准测试中所报告的最佳结果 1057 秒相近[18]。

需要注意的是：由于我们进行了本地性优化，输入速率高于洗牌速率和输出速率——大部分数据是从本地磁盘读取的，从而绕过了我们相对带宽有限的网络。洗牌速率高于输出速率是因为输出阶段会写入两份排序后的数据（出于可靠性和可用性的考虑，我们对输出结果进行两份副本的制作）。我们写入两份副本是因为这是底层文件系统提供的可靠性和可用性的机制。如果底层文件系统使用纠删码[14]而非复制，那么写入数据所需的网络带宽将会减少。

## 5.4 备份任务的影响

在图 3（b）中，我们展示了排序程序在禁用备份任务情况下的执行情况。其执行流程与图 3（a）所示类似，只是有一个很长的尾部，在此期间几乎没有任何写入活动。960 秒后，除了 5 个归约任务外，其余任务均已完成。然而，这最后几个落后的任务直到 300 秒后才完成。整个计算过程耗时 1283 秒，比之前增加了 44%。

## 5.5 机器故障

在图 3（c）中，我们展示了排序程序的一次执行情况，在计算开始几分钟后，我们故意终止了 1746 个工作进程中的 200 个。底层的集群调度器随即在这些机器上重新启动了新的工作进程（因为只是进程被终止，机器仍能正常运行）。

工人死亡表现为负输入率，因为一些先前已完成的地图工作会消失（因为相应的地图绘制工人被杀），需要重新执行。重新执行这些地图工作相对迅速。整个计算过程包括启动开销在内共耗时 933 秒（仅比正常执行时间增加了 5%）。

# 6 经验

我们于 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，包括本地性优化、在工作节点之间动态平衡任务执行负载等。自那时起，我们惊喜地发现 MapReduce 库在我们所处理的各类问题中有着广泛的应用。它已在谷歌内部的众多领域得到使用，包括:

大规模机器学习问题

· 谷歌新闻和 Froogle 产品中的聚类问题，

· 用于生成热门查询报告（例如谷歌年度回顾）的数据提取，

· 从网页中提取用于新实验和产品的属性（例如，从大量网页语料库中提取地理位置以实现本地化搜索），以及

大规模图计算。

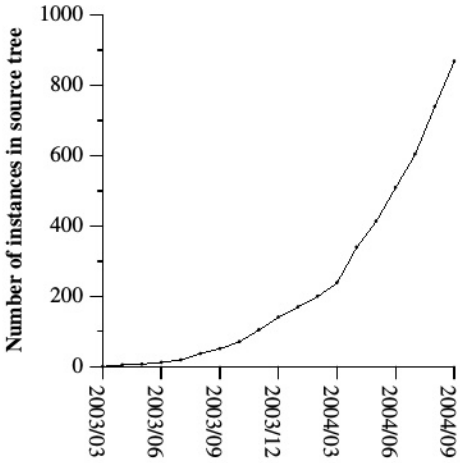


图 4：随时间变化的 MapReduce 实例

| | |
|---|---|
| 工作岗位数量 | 29,423 |
| 平均工作完成时间 | 634 秒 |
| 机器使用天数 | 79186 天 |
| 输入数据读取 | 3288 太字节 |
| 中间生成的数据 | 758 太字节 |
| 输出数据已写入 | 193 太字节 |
| 每份工作平均使用的机器数量 | 157 |
| 平均每份工作中的工人死亡人数 | 1.2 |
| 每个作业的平均映射任务数 | 3,351 |
| 平均每个作业的归约任务数 | 55 |
| 独特的*地图*实现方式 | 395 |
| 独特的*归约*实现方式 | 269 |
| 独特的*映射/归约*组合 | 426 |

表 1：2004 年 8 月运行的 MapReduce 作业

图 4 展示了随着时间推移，提交到我们主要源代码管理系统中的独立 MapReduce 程序数量显著增长，从 2003 年初的 0 个增加到 2004 年 9 月底的近 900 个。MapReduce 取得如此成功的原因在于，它使得编写一个简单的程序并在半小时内在上千台机器上高效运行成为可能，极大地加快了开发和原型设计的周期。此外，它还让那些没有分布式和/或并行系统经验的程序员能够轻松利用大量资源。

在每个作业结束时，MapReduce 库都会记录该作业所使用的计算资源的统计信息。表 1 展示了 2004 年 8 月在谷歌运行的部分 MapReduce 作业的统计信息。

## 6.1 大规模索引

到目前为止，我们对 MapReduce 最重要的应用之一是对生产索引的全面重写。

用于生成 Google 网络搜索服务所用数据结构的索引系统。该索引系统以由我们的抓取系统获取并存储为一组 GFS 文件的大量文档为输入。这些文档的原始内容超过 20 太字节的数据。索引过程作为五到十个 MapReduce 操作的序列运行。使用 MapReduce（而非先前版本索引系统中的临时分布式处理）带来了诸多好处：

索引代码更简单、更精简且更易于理解，因为处理容错、分布和并行化的代码都隐藏在 MapReduce 库中。例如，使用 MapReduce 表达时，计算的一个阶段的代码量从大约 3800 行 C++ 代码减少到了大约 700 行。

MapReduce 库的性能足够出色，我们能够将概念上不相关的计算分开进行，而无需将它们混合在一起以避免对数据进行额外的遍历。这使得更改索引过程变得十分容易。例如，在旧的索引系统中需要花费数月时间才能完成的一项更改，在新系统中仅需几天即可实现。

索引过程的操作变得容易多了，因为由机器故障、机器运行缓慢以及网络中断所引发的大多数问题，都由 MapReduce 库自动处理，无需操作人员干预。此外，通过向索引集群添加新机器，可以轻松提升索引过程的性能。

## 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实现计算的并行化。例如，使用并行前缀计算，可以在 N 个处理器上以 log N 的时间计算 N 元素数组的所有前缀上的关联函数[6, 9, 13]。基于我们在大规模实际计算中的经验，MapReduce 可以被视为对这些模型的一种简化和提炼。更重要的是，我们提供了一种容错实现，可扩展到数千个处理器。相比之下，大多数并行处理系统仅在较小规模上实现，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]提供了更高级别的抽象，使程序员更容易编写并行程序。这些系统与 MapReduce 的一个关键区别在于，MapReduce 利用一种受限的编程模型来自动并行化用户程序，并提供透明的容错能力。

我们的本地化优化从诸如主动磁盘[12, 15]等技术中汲取灵感，这些技术将计算任务推送到靠近本地磁盘的处理单元，以减少通过 I/O 子系统或网络传输的数据量。我们运行在直接连接少量磁盘的商用处理器上，而非直接在磁盘控制器处理器上运行，但总体方法是相似的。

我们的备份任务机制类似于夏洛特系统 [3] 中采用的急切调度机制。简单急切调度的一个缺点是，如果给定的任务导致反复失败，整个计算将无法完成。我们通过跳过坏记录的机制解决了部分此类问题。

MapReduce 的实现依赖于一个内部的集群管理系统，该系统负责在大量共享机器上分发和运行用户任务。虽然这并非本文的重点，但该集群管理系统在精神上类似于 Condor 等其他系统[16]。

MapReduce 库中的排序设施的操作方式与 NOW-Sort [1] 类似。源机器（映射工作器）将待排序的数据分区，并将其发送给 R 个归约工作器中的一个。每个归约工作器在其本地对数据进行排序（如果可能的话，在内存中进行）。当然，NOW-Sort 没有我们库中用户可定义的映射和归约函数，因此其适用范围较窄。

River [2] 提供了一种编程模型，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 类似，River 系统即使在存在由异构硬件或系统扰动引入的非均匀性的情况下，也试图提供良好的平均性能。River 通过精心安排磁盘和网络传输来实现这一点，以达到平衡的完成时间。MapReduce 则采取了不同的方法。通过限制编程模型，MapReduce 框架能够将问题划分为大量细粒度的任务。这些任务会动态地调度到可用的工作节点上，以便更快的工作节点处理更多的任务。这种受限的编程模型还允许我们在作业接近尾声时安排任务的冗余执行，这在存在非均匀性（例如缓慢或卡住的工作节点）的情况下极大地减少了完成时间。

BAD-FS [5] 的编程模型与 MapReduce 大不相同，并且与 MapReduce 不同的是，它旨在

在广域网中执行任务。然而，两者有两个基本的相似之处。（1）两个系统都使用冗余执行来从故障导致的数据丢失中恢复。（2）两者都使用具有本地性感知的调度来减少通过拥塞网络链路传输的数据量。

TACC [7] 是一个旨在简化高可用网络服务构建的系统。与 MapReduce 类似，它依靠重新执行来实现容错机制。

## 8 结论

MapReduce 编程模型已在谷歌成功应用于多种不同的用途。我们认为其成功的原因有以下几点。首先，该模型易于使用，即使是没有并行和分布式系统经验的程序员也能轻松上手，因为它隐藏了并行化、容错、局部性优化和负载均衡等细节。其次，大量问题都能轻松地用 MapReduce 计算来表达。例如，MapReduce 用于生成谷歌生产网络搜索服务的数据、用于排序、用于数据挖掘、用于机器学习以及许多其他系统。第三，我们开发了一种 MapReduce 实现，能够扩展到由数千台机器组成的大型集群。该实现能高效利用这些机器资源，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几件事。首先，限制编程模型使得并行化和分布式计算变得容易，并且能够使这些计算具有容错性。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措施都旨在减少网络传输的数据量：局部性优化使我们能够从本地磁盘读取数据，而将中间数据仅写入本地磁盘一次则节省了网络带宽。第三，冗余执行可用于减轻慢速机器的影响，并处理机器故障和数据丢失。

## 致谢

乔希·莱文伯格（Josh Levenberg）在修订和扩展用户级 MapReduce API 方面发挥了重要作用，他根据自身使用 MapReduce 的经验以及其他人提出的改进建议，新增了许多功能。MapReduce 从 Google 文件系统（GFS）[8] 中读取输入，并将输出写入该系统。我们感谢莫希特·阿龙（Mohit Aron）、霍华德·戈比奥夫（Howard Gobioff）、马库斯·古茨克（Markus Gutschke）、大卫·克莱默（David Kramer）、梁顺德（Shun-Tak Leung）和乔希·雷德斯通（Josh Redstone）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·瑟尔乔格鲁（Olcan Sercinoglu）在开发 MapReduce 所使用的集群管理系统方面所做的工作。迈克·伯罗斯（Mike Burrows）、威尔逊·谢（Wilson Hsieh）、乔希·莱文伯格（Josh Levenberg）、莎伦·珀尔（Sharon Perl）、罗布·派克（Rob Pike）和黛比·瓦拉赫（Debby Wallach）对本文的早期草稿提出了有益的意见。匿名的 OSDI 审稿人以及我们的指导人埃里克·布鲁尔（Eric Brewer）提供了许多有助于改进本文的建议。最后，我们感谢谷歌工程组织内所有使用 MapReduce 的用户，他们提供了有用的反馈、建议和错误报告。

## 参考文献

[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.

[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.

[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.

[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.

[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.

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

[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.

[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.

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

[10] Jim Gray. Sort benchmark home page. 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.

[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.

[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 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.

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

[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.

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

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

## 词频

本节包含一个程序，该程序会统计在命令行中指定的一组输入文件中每个唯一单词出现的次数。

```
#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;
}
```