MapReduce:在大型集群上简化数据处理

Jeffrey Dean 和 Sanjay Ghemawat

jeff@google.com, sanjay@google.com
谷歌，Inc.

摘要

MapReduce 是一种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，处理一个键值对，生成一组中间键值对，以及一个 *reduce* 函数，合并与同一个中间键相关联的所有中间值。~~许多现实世界的任务都可以在这个模型中表达，如论文中所示。~~

以这种函数式风格编写的程序被自动并行化，并在一个大的商用机器集群上执行。运行时系统负责处理输入数据的分区、跨一组机器调度程序的执行、处理机器故障以及管理所需的机器间通信等细节。~~这使得没有任何并行和分布式系统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，并且具有高度的可扩展性:一个典型的 MapReduce 计算在数千台机器上处理许多 tb 级的数据。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，每天在谷歌的集群上执行超过 1000 个 MapReduce 作业。

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个处理大量原始数据(如爬取的文档、web 请求日志等)的专用计算，以计算各种派生数据，如倒排索引、web 文档图结构的各种表示、每个主机爬取的页面数的摘要、a中最频繁查询的集合给定的一天，等等。大多数这样的计算在概念上都很简单。然而，输入数据通常很大，为了在合理的时间内完成计算，计算必须分布在数百或数千台机器上。如何并行化计算、分布数据、处理故障等问题共同作用，用大量复杂的代码来掩盖原始的简单计算，以处理这些问题。

作为对这种复杂性的反应，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，但隐藏了库中并行化、容错、数据分布和负载平衡的混乱细节。我们的抽象受到了 Lisp 和许多其他函数式语言中存在的 *map* 和 *reduce* 原语的启发。我们意识到，我们大多数的计算都涉及到对输入中的每个逻辑“记录”应用 *map* 操作，以便计算一组中间的键/值对，然后对共享相同键的所有值应用 *reduce* 操作，以便适当地组合派生数据。我们使用具有用户指定的 map 和 reduce 操作的功能模型，使我们能够轻松地将大型计算并行化，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，可以实现大规模计算的自动并行化和分发，结合这个接口的实现，可以在大宗商品 pc 的大型集群上实现高性能。

第 2 节描述了基本的编程模型，并给出了几个例子。第 3 节描述了针对我们基于集群的计算环境量身定制的 MapReduce 接口的实现。第 4 节描述了我们发现有用的编程模型的几个改进。第 5 节对我们在各种任务中的实现进行了性能测量。第 6 节探讨了在谷歌中使用 MapReduce，包括我们使用它作为基础的经验

申请重写我们的生产索引系统。第 7 节讨论了相关和未来的工作。

2 编程模型

计算需要一组*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*。

Map 是由用户编写的，它接受一个输入对，并生成一组*中间的*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 函数也是由用户编写的，它接受一个中间键 I 和该键对应的一组值。它将这些值合并在一起，形成一个可能更小的值集合。通常，每次 *Reduce* 调用只产生 0 或 1 个输出值。中间值通过迭代器提供给用户的 reduce 函数。这使我们能够处理过大的值列表，无法放入内存。

2.1 的例子

考虑计算每个单词在大量文档集合中的出现次数的问题。用户将编写类似于以下伪代码的代码:

```
map(字符串键，字符串值)://键:
  文档名称//值:文档内容
  对于 value 中的每个单词 w:
    EmitIntermediate(w，
      “1”);

reduce(字符串键，迭代器值)://键:
  一个单词
  //values:计数的列表
  Int result = 0;
  对于 values 中的每
  个 v:
    result += ParseInt(v);
  发出(AsString 结尾(结
  果));
```

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就是`1`)。reduce 函数将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和可选的调优参数填写 *mapreduce 规范*对象。然后，用户调用 *MapReduce* 函数，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 包含了这个示例的完整程序文本。

2.2 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 map 和 reduce 函数有关联的类型:

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

即，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，并将其留给用户代码在字符串和适当的类型之间进行转换。

2.3 更多示例

下面是一些有趣的程序的简单示例，可以很容易地表示为 MapReduce 计算。

分布式 Grep: map 函数如果匹配提供的模式，就会发出一行代码。reduce 函数是一个恒等函数，它只是将提供的中间数据复制到输出。

URL 访问频率的计数:map 函数处理网页请求的日志并输出 hURL, 1i。reduce 函数将相同 URL 的所有值加在一起，并发出一个 hURL, total counti 对。

反向网络链接图:map 函数为每个链接输出 htarget, sourcei 对，指向在名为 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:htarget, list(source)i

每个主机的词向量:一个词向量将出现在一篇文档或一组文档中最重要的单词总结为 hword, frequencyi 对的列表。map 函数为每个输入文档发出一个 hhostname, term vectori 对(其中从文档的 URL 中提取主机名)。reduce 函数传递给定主机的所有文档词向量。它把这些词向量加在一起，扔掉不频繁的词，然后发出一个最终的 hhostname, term vectori 对。

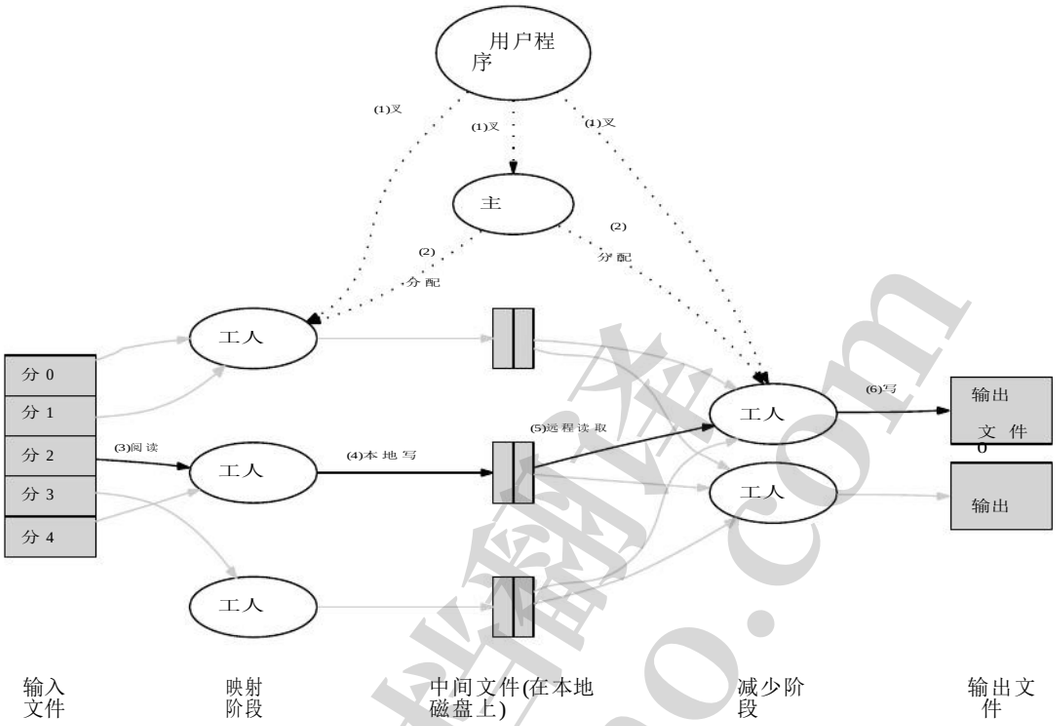


图 1:执行概览

倒排索引:map 函数解析每个文档，并发出 hword 序列，文档 IDi 对。reduce 函数接受给定单词的所有对，对相应的文档 ID 进行排序，并发出一个 hword, list(文档 ID)i 对。所有输出对的集合形成一个简单的倒排索引。很容易增加这种计算来跟踪单词的位置。

分布式排序:map 函数从每条记录中提取键，并发出一个 hkey, recordi 对。reduce 函数会原原不动地发出所有的对。这种计算依赖于 4.1 节中描述的划分机制和 4.2 节中描述的排序属性。

3 实现

MapReduce 接口的许多不同实现是可能的。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上广泛使用的计算环境的一种实现通过交换式以太网[4]将大量的商用 pc 连接在一起。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 内存。

(2)使用了商用网络硬件——通常在机器级别上是 100 兆/秒或 1 兆/秒，但在整体二分带宽上平均要少得多。

(3)集群由数百或数千台机器组成，因此机器故障很常见。

存储是由直接附接到各个机器上的廉价 IDE 磁盘提供的。使用内部开发的分布式文件系统[8]来管理存储在这些磁盘上的数据。该文件系统利用复制在不可靠的硬件之上提供可用性和可靠性。

(5)用户提交作业到一个调度系统。每个作业由一组任务组成，并被调度器映射到一个集群内的一组可用机器。

3.1 执行概览

通过自动划分输入数据，*Map* 调用分布在多台机器上

分成 M 组。输入切分可以由不同的机器并行处理。*Reduce* 调用是通过使用分区函数(例如，hash(key) **mod**R)将中间键空间划分为 R 块来分布的。分区数(R)和分区函数由用户指定。

图 1 展示了在我们的实现中一个 MapReduce 操作的整体流程。当用户程序调用 MapReduce 函数时，会发生以下操作序列(图 1 中编号的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成 M 块，通常每块 16 MB 到 64 MB (MB)(由用户通过可选参数进行控制)。然后它在集群机器上启动程序的许多副本。

2.这个程序的一个副本是特别的——master。其余的都是被主程序分配工作的工人。要分配的 map 任务有 M 个，reduce 任务有 R 个。master 挑选空闲的 worker，给每个 worker 分配一个 map 任务或者一个 reduce 任务。

3.被分配 map 任务的 worker 读取相应输入分割的内容。它从输入数据中解析出键/值对，并将每个键/值对传递给用户定义的 *Map* 函数。由 *Map* 函数产生的中间键/值对被缓冲在内存中。

4.周期性地将缓冲对写入本地磁盘，通过分区函数划分为 R 个区域。这些缓冲对在本地磁盘上的位置被传回给 master, master 负责将这些位置转发给 reduce worker。

5.当一个 reduce worker 被 master 通知这些位置时，它使用远程过程调用从 map worker 的本地磁盘读取缓冲数据。当 reduce worker 读取完所有中间数据后，它会根据中间键进行排序，这样所有出现相同键的地方就会被分组在一起。排序是需要的，因为通常许多不同的键映射到同一个 reduce 任务。如果中间数据量太大，无法放入内存，则使用外部排序。

6. reduce worker 遍历已排序的中间数据，对于遇到的每个唯一中间键，它将键和相应的中间值集合传递给用户的 *reduce* 函数。*Reduce* 函数的输出会被附加到这个 Reduce 分区的最终输出文件中。

7.当所有 map 任务和 reduce 任务完成后，master 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，mapreduce 执行的输出在 R 输出文件中可用(每个 reduce 任务一个，文件名由用户指定)。通常情况下，用户不需要将这些 R 输出文件组合成一个文件——他们经常将这些文件作为输入传递给另一个 MapReduce 调用，或者从另一个能够处理被划分为多个文件的输入的分布式应用程序中使用它们。

3.2 主数据结构

主数据结构保持几个数据结构。对于每个 map 任务和 reduce 任务，它存储状态(*空闲*、*进行中*或*完成*)，以及 worker machine 的身份(对于非空闲任务)。

master 是中间文件区域的位置从 map 任务传播到 reduce 任务的通道。因此，对于每一个完成的 map 任务，master 都会存储 map 任务产生的 R 中间文件区域的位置和大小。随着地图任务的完成，这个位置和大小信息的更新就会被接收。这些信息被渐进地推送给正在执行 *reduce* 任务的工人。

3.3 容错

由于 MapReduce 库旨在帮助使用数百或数千台机器处理非常大的数据，因此该库必须优雅地容忍机器故障。

工人失败

主进程周期性地 ping 每个工作进程。如果在一定的时间内没有收到 worker 的响应，master 会将 worker 标记为失败。任何由 worker 完成的 map 任务都会被重置为初始的*空闲*状态，因此可以被调度到其他 worker 上。类似地，失败的 worker 上正在进行的任何 map 任务或 reduce 任务也会重置为*空闲状态*，从而有资格重新调度。

完成的 map 任务在发生故障时被重新执行，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。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 map 任务首先由 worker a 执行，然后由 worker B 执行(因为 a 失败)，所有

执行 reduce 任务的工人会被通知重新执行。任何还没有从 worker A 读取数据的 reduce 任务，都会从 worker B 读取数据。

MapReduce 对于大规模的 worker 故障具有弹性。例如，在一次 MapReduce 操作中，一个正在运行的集群的网络维护导致同时有 80 台机器的组在几分钟内变得不可访问。MapReduce master 只是重新执行不可达的 worker 机器所做的工作，并继续向前推进，最终完成 MapReduce 操作。

主服务器失败

让 master 写上面描述的主数据结构的周期性检查点是很容易的。如果主任务死亡，可以从上一个检查点状态开始一个新的副本。然而，考虑到只有一个 master，其失败的可能性不大;因此，如果 master 失败，我们当前的实现就会中止 MapReduce 的计算。客户端可以检查这种情况，如果他们愿意，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与整个程序的无错误顺序执行所产生的输出相同。

我们依靠 map 和 reduce 任务输出的原子提交来实现这一属性。每个进行中的任务将其输出写入私有临时文件。一个 reduce 任务产生一个这样的文件，一个 map 任务产生 R 个这样的文件(每个 reduce 任务一个)。当一个 map 任务完成后，worker 会向 master 发送一条消息，并在消息中包含 R 临时文件的名称。如果 master 收到一个已经完成的 map 任务的完成消息，它会忽略这个消息。否则，它会在一个主数据结构中记录 R 文件的名称。

当一个 reduce 任务完成时，reduce worker 会自动将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相同的 reduce 任务，则会对同一个最终输出文件执行多个重命名调用。我们依赖于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只包含一次执行 reduce 任务所产生的数据。

我们绝大多数的 *map* 和 *reduce* 操作符都是确定性的，在这种情况下，我们的语义相当于顺序执行，这使得它非常程序员很容易推理出他们程序的行为。当 *map* 和/或 *reduce* 操作符是不确定的，我们提供较弱但仍然合理的语义。在不确定操作符存在的情况下，特定 reduce 任务 R1 的输出相当于顺序执行不确定程序所产生的针对 R1 的输出。然而，不同 reduce 任务 R2 的输出可能对应于不确定程序的不同顺序执行所产生的 R2 的输出。

考虑 map 任务 M 和 reduce 任务 R1 和 R2。设 e(Ri)为提交的 Ri 的执行(正好有一个这样的执行)。出现较弱的语义是因为 e(R1)可能读取了一次执行 M 所产生的输出，而 e(R2)可能读取了一次不同执行 M 所产生的输出。

3.4 位置

在我们的计算环境中，网络带宽是一个相对稀缺的资源。我们利用输入数据(由 GFS[8]管理)存储在组成我们集群的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 64 MB 的块，并在不同的机器上存储每个块的若干副本(通常为 3 份副本)。MapReduce master 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机器上调度一个 map 任务。如果做不到这一点，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工作节点上运行大型 MapReduce 操作时，大多数输入数据都是在本地读取的，不会消耗网络带宽。

3.5 任务粒度

我们将 map 阶段细分为 M 块，reduce 阶段细分为 R 块，如上所述。理想情况下，M 和 R 应该比 worker machines 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，也可以在 worker 失败时加快恢复速度:它已经完成的许多 map 任务可以分散到所有其他 worker 机器上。

在我们的实现中，M 和 R 可以有多大的实际界限，因为 master 必须做出 O(M + R)调度决策，并如上所述在内存中保持 O(M* R)状态。(然而，内存使用的常量因素很小:状态的 O(M* R)块由每个 map 任务/reduce 任务对大约 1 字节的数据组成。)

此外，R 经常受到用户的限制，因为每个 reduce 任务的输出最终都在一个单独的输出文件中。在实践中，我们倾向于选择 M，这样每个单独的任务大约是 16 MB 到 64 MB 的输入数据(这样上面描述的局部性优化是最有效的)，并且我们使 R 成为我们期望使用的 worker 机器数量的一个小倍数。我们经常使用 2000 台工作机器，在 M = 200000 和 R = 5000 的情况下执行 MapReduce 计算。

3.6 备份任务

导致 MapReduce 操作总时间延长的一个常见原因是“掉队者”(straggler):在计算中，最后几个 map 或 reduce 任务中的一个需要花费异常长的时间。出现“掉队者”的原因有很多。例如，有坏磁盘的机器可能会频繁地出现可纠正的错误，从而使其读取性能从 30 MB/s 降至 1 MB/s。集群调度系统可能已经在这台机器上调度了其他任务，由于对 CPU、内存、本地磁盘或网络带宽的竞争，导致它执行 MapReduce 代码的速度变慢。我们最近经历的一个问题是机器初始化代码的一个 bug，导致处理器缓存被禁用:受影响的机器上的计算速度慢了一百倍以上。

我们有一个通用机制来缓解掉队者的问题。当一个 MapReduce 操作接近完成时，主调度剩余*正在*执行的任务的备份执行。每当主执行或备份执行完成时，该任务就被标记为完成。我们已经调整了这种机制，使其通常增加操作所使用的计算资源不超过几个百分点。我们发现，这大大减少了完成大型 MapReduce 操作的时间。举个例子，5.3 节中描述的 sort 程序在禁用备份任务机制的情况下需要多花 44%的时间才能完成。

4 改进

虽然简单地编写 *Map* 和 *Reduce* 函数提供的基本功能已经足够满足大多数需求，但我们发现了一些有用的扩展。这些将在本节中进行描述。

4.1 分区函数

MapReduce 的用户指定他们想要的 reduce 任务/输出文件的数量(R)。数据通过这些任务使用分区函数 on 进行分区中间密钥。提供了一个默认的分区函数，使用哈希(例如“hash(key) **mod** R”)。这往往会导致相当平衡的分区。然而，在某些情况下，通过键的某些其他函数来划分数据是有用的。例如，有时输出的键是 url，我们希望单个主机的所有条目最终都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(urlkey)) **mod**R”作为分区函数，会导致来自同一主机的所有 url 最终出现在同一个输出文件中。

4.2 排序保证

我们保证在给定的分区内，中间的键/值对按键递增的顺序处理。这种排序保证使得每个分区很容易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高效的随机访问查找时，或者输出的用户发现对数据进行排序很方便时，这是有用的。

4.3 Combiner 函数

在某些情况下，每个 map 任务产生的中间键有显著的重复，用户指定的 *Reduce* 函数是交换性和结合性的。这方面的一个很好的例子是 2.1 节中的单词计数例子。由于词频倾向于遵循 Zipf 分布，每个 map 任务将产生数百或数千条形式为<the, 1>的记录。所有这些计数将通过网络发送到单个 reduce 任务，然后由 *reduce* 函数加在一起，产生一个数字。我们允许用户指定一个可选的 *Combiner* 函数，在这些数据通过网络发送之前对其进行部分合并。

Combiner 函数在每台执行 map 任务的机器上执行。通常使用相同的代码来实现 combiner 和 reduce 函数。reduce 函数和 combiner 函数之间唯一的区别是 MapReduce 库如何处理函数的输出。reduce 函数的输出会写入到最终的输出文件中。combiner 函数的输出被写入一个中间文件，该文件将被发送给 reduce 任务。

部分合并显著加速了某些类别的 MapReduce 操作。附录 A 包含了一个使用 combiner 的例子。

4.4 输入和输出类型

MapReduce 库提供了读取几种不同格式的输入数据的支持。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。每个输入类型的实现都知道如何将自己分割成有意义的范围，以便作为单独的 map 任务处理(例如，文本模式的范围分割确保范围分割只发生在行边界)。用户可以通过提供一个简单的*阅读器*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量预定义输入类型中的一种。

*阅读器*不一定需要提供从文件中读取的数据。例如，很容易定义一个从数据库或映射在内存中的数据结构中读取记录的 reader。

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的输出类型的支持。

4.5 的副作用

在某些情况下，MapReduce 的用户发现用 map 和/或 reduce 操作生成辅助文件作为额外的输出是很方便的。我们依赖于应用程序编写器来使这种副作用具有原子性和幂等性。通常，应用程序写入临时文件，并在该文件完全生成后原子性地重命名该文件。

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，用户代码中的 bug 会导致 *Map* 或 *Reduce* 函数在处理某些记录时崩溃。这样的 bug 会阻止 MapReduce 操作的完成。通常的做法是修复 bug，但有时这是不可行的;也许 bug 在第三方库中，而这个库的源代码是不可用的。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，MapReduce 库检测哪些记录会导致确定性的崩溃，并跳过这些记录，以便向前推进。

每个 worker 进程安装一个信号处理程序来捕获分段违规和总线错误。在调用用户 *Map* 或 *Reduce* 操作之前，MapReduce 库会将参数的序号存储在一个全局变量中。如果用户代码生成一个信号，信号处理程序发送一个“last gasp”UDP 数据包，其中包含了 MapReduce master 的序列号。当 master 在某条记录上看到多个失败时，它指示当它发出下一次重新执行相应的 Map 或 Reduce 任务时，该记录应该被跳过。

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，通常在数千台机器上，工作分配决策是由 master 动态做出的。为了帮助方便调试、分析和小规模测试，我们开发了 MapReduce 库的另一种实现，它在本地机器上顺序执行 MapReduce 操作的所有工作。控件被提供给用户，这样计算就可以被限制在特定的 map 任务中。用户通过一个特殊的标志调用他们的程序，然后可以轻松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

master 运行一个内部 HTTP 服务器，并导出一组状态页面供人类使用。状态页显示了计算的进度，比如已经完成了多少任务，有多少在进行中，输入的字节数，中间数据的字节数，输出的字节数，处理速率等。这些页面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。这些页面也可以用来判断何时计算比预期慢得多。

此外，顶层状态页面显示哪些 worker 失败了，以及他们失败时正在处理哪些 map 和 reduce 任务。当试图诊断用户代码中的 bug 时，这些信息非常有用。

4.9 计数器

MapReduce 库提供了一个计数器来对各种事件的出现次数进行计数。例如，用户代码可能想要计算处理的单词总数或德语文档编入索引的数量，等等。

为了使用这个功能，用户代码创建一个命名的计数器对象，然后在 *Map* 和/或 *Reduce* 函数中适当地增加计数器。例如:

```
反*大写;
uppercase =
GetCounter( “ uppercase ” );

map(字符串名称， 字符串内容):
  对于 contents 中的每个单词
  w:
    if (iscapitalize (w)):大写-
      >Increment();

    EmitIntermediate  (w, " 1
    ");
```

来自单个工作机器的计数器值周期性地传播到主节点(承载在 ping 响应上)。master 聚合成功的 map 和 reduce 任务的计数器值，并在 MapReduce 操作完成时返回给用户代码。当前的计数器值也会显示在 master 状态页面上，这样人类就可以看到实时计算的进度。在聚合计数器值时，master 消除了重复执行同一 map 或 reduce 任务的影响，以避免重复计算。(重复执行可能是由于我们使用备份任务以及由于失败而重新执行任务而产生的。)

一些计数器值是由 MapReduce 库自动维护的，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。

用户发现计数器对于对 MapReduce 操作的行为进行完整性检查很有用。例如，在一些 MapReduce 操作中，用户代码可能希望确保产生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些可容忍的比例内。

5 的性能

在本节中，我们将测量在大型集群上运行的两个计算的 MapReduce 性能。一次计算通过大约 1tb 的数据来寻找特定的模式。另一个计算对大约 1tb 的数据进行排序。

这两个程序代表了 MapReduce 用户编写的真实程序的一个大子集——一类程序将数据从一种表示形式转移到另一种表示形式，另一类程序从一个大数据集中提取少量有趣的数据。

5.1 集群配置

所有的程序都在一个由大约 1800 台机器组成的集群上执行。每台机器都有两个 2GHz 的 Intel Xeon 处理器，支持超线程，4GB 内存，两个 160GB IDE

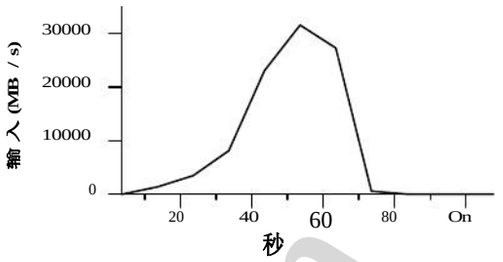


图 2:随时间变化的数据传输率

磁盘，以及千兆以太网链路。这些机器被安排在一个两级的树形交换网络中，在根节点有大约 100-200 Gbps 的总带宽可用。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少于 1 毫秒。

在 4GB 的内存中，大约有 1-1.5GB 是由集群上运行的其他任务分配的。这些程序是在周末的下午执行的，这个时候 cpu、磁盘和网络大部分都是空闲的。

5.2 Grep

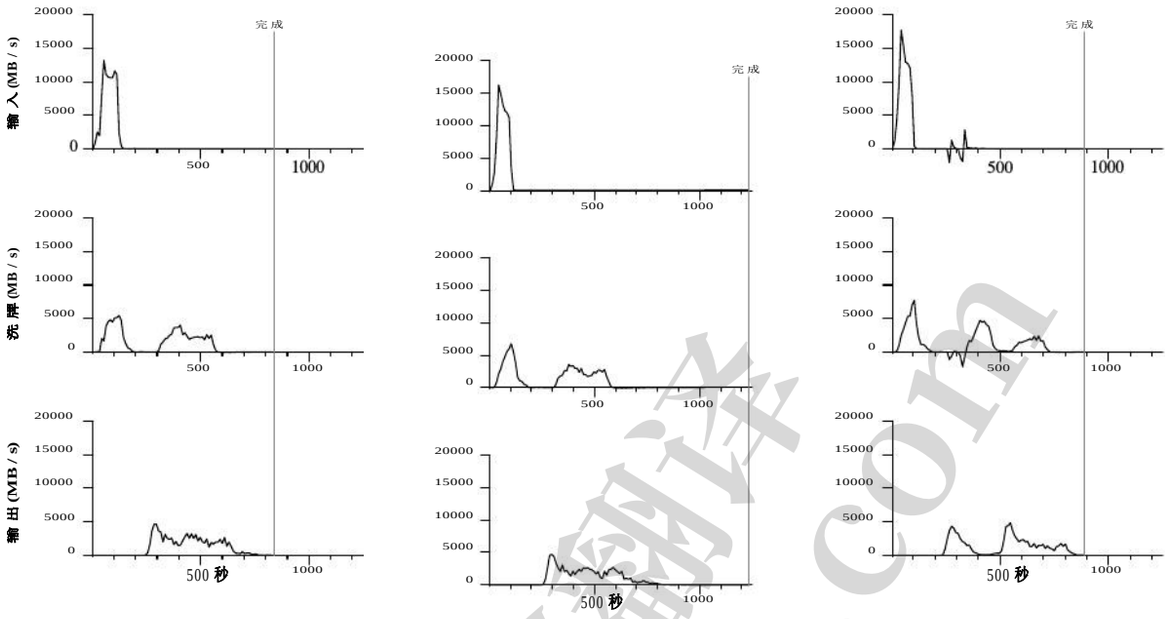
grep 程序扫描 10^{10} 条 100 字节记录，搜索一个相对罕见的三字符模式(该模式出现在 92,337 条记录中)。输入被分割成大约 64MB 的片段(M = 15000)，整个输出被放置在一个文件中(R = 1)。

图 2 显示了随着时间的推移计算的进度。y 轴表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，速率逐渐加快，当分配了 1764 个 worker 时，速率最高超过 30 GB/s。随着 map 任务的完成，速率开始下降，并在计算大约 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。这种开销是由于将程序传播到所有工作机器，并延迟与 GFS 交互，以打开 1000 个输入文件的集合，并获得本地化优化所需的信息。

5.3 类

sort 程序对 10^{10} 条 100 字节的记录(大约 1 tb 的数据)进行排序。这个程序模仿了 TeraSort 基准[10]。

该排序程序由少于 50 行用户代码组成。一个 3 行 *Map* 函数从文本行中提取一个 10 字节的排序键，并发出键和



(a)正常执行　　(b)无备份任务　　(c) 200 个任务被杀死

图 3:sort 程序不同执行的数据传输率随时间的变化

原始文本行作为中间键值对。我们使用了内置的*恒等*函数作为 *Reduce* 运算符。这个函数传递没有改变的中间键值对作为输出键值对。最终排序的输出被写入一组 2 路复制的 GFS 文件(即 2 tb 被写入作为程序的输出)。

和之前一样，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)。分区函数使用密钥的初始字节将其分割为 R 块之一。

我们用于此基准测试的分区函数具有键分布的内置知识。在一般的排序程序中，我们会添加一个 pre-pass MapReduce 操作，该操作将收集键的样本，并使用采样键的分布来计算最终排序 pass 的分割点。

图 3 (a)显示了排序程序正常执行的进度。左上角的图表显示了读取输入的速度。速率峰值约为 13 GB/s，并很快消失，因为所有 map 任务都在 200 秒内完成。注意，输入速率低于 *grep*。这是因为 sort map 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。grep 对应的中间输出的大小可以忽略不计。

中左的图显示了数据通过网络从 map 任务发送到 reduce 任务的速率。这种混洗在第一个 map 任务完成时就开始了。图中的第一个驼峰是 for

第一批约 1700 个 reduce 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 reduce 任务)。大约在计算的 300 秒后，第一批 reduce 任务中的一些完成了，我们开始为剩下的 reduce 任务进行数据洗牌。所有的洗牌都是在计算大约 600 秒后完成的。

左下角的图表显示了 reduce 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，在第一个洗牌周期结束和写入周期开始之间存在延迟。写入会以大约 2-4 GB/s 的速率持续一段时间。所有的写操作在计算的 850 秒内完成。算上启动开销，整个计算耗时 891 秒。这与 TeraSort 基准[18]目前报告的最佳结果 1057 秒类似。

有几点需要注意:由于我们的局部性优化，输入速率高于洗牌速率和输出速率——大多数数据是从本地磁盘读取，并绕过我们相对带宽受限的网络。shuffle 速率比 output 速率高是因为输出阶段写了两个排序数据的副本(出于可靠性和可用性的原因，我们制作了两个输出副本)。我们写两个副本，因为这是我们底层文件系统提供的可靠性和可用性的机制。如果底层文件系统使用纠删码[14]而不是复制，写入数据的网络带宽要求将会降低。

5.4 备份任务效果

在图 3 (b)中，我们展示了在禁用备份任务的情况下 sort 程序的执行情况。执行流程与图 3 (a)所示类似，除了有一个非常长的尾，几乎没有发生任何写活动。960 秒后，除 5 个 reduce 任务外，其他任务都完成了。然而，这最后几名掉队者直到 300 秒后才完成。整个计算耗时 1283 秒，运行时间增加了 44%。

5.5 机器故障

在图 3 (c)中，我们展示了 sort 程序的执行过程，在计算进行了几分钟后，我们故意杀死了 1746 个 worker 进程中的 200 个。底层的集群调度器立即在这些机器上重启新的工作进程(因为只有进程被杀死，机器仍然正常运行)。

由于一些之前完成的 map 工作消失了(因为对应的 map 工作被杀死了)并且需要重做，工作进程的死亡会以负输入率的形式显示出来。这种地图工作的重新执行相对较快。包括启动开销在内，整个计算在 933 秒内完成(仅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 月对其进行了重大增强，包括局部性优化、跨 worker 机器执行任务的动态负载平衡等。从那时起，我们惊喜地发现 MapReduce 库对于我们所处理的这类问题的适用范围是如此广泛。在谷歌中，它已经被广泛应用于各个领域，包括:

- 大规模机器学习问题，
- 谷歌新闻和 Froogle 产品的聚类问题，
- 提取用于生成流行查询报告的数据(例如谷歌时代精神)，
- 为新实验和产品提取 web 页面的属性(例如，从大量 web 页面语料库中提取地理位置，用于本地化搜索)
- 大规模的图计算。

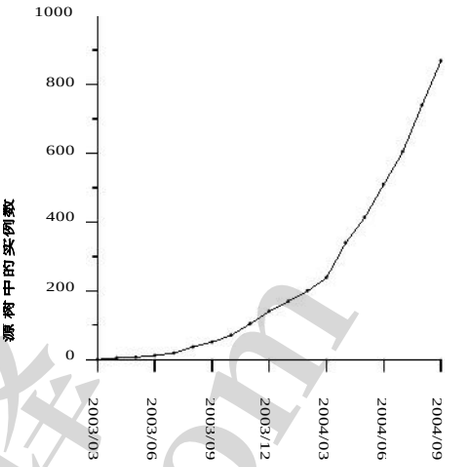


图 4:MapReduce 实例随时间的变化

作业数量	29423 年
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 年结核病
产生的中间数据	758 年结核病
输出数据写入	193 年结核病
平均工人每工作机器	157
每项工作的平均工人死亡人数	1.2
每个作业的平均 map 任务数	3351 年
减少每个工作的平均任务数	55
独特的*地图*实现	395
独特的 *reduce* 实现	269
独特的 *map/reduce* 组合	426

表 1:MapReduce 作业运行于 2004 年 8 月

图 4 显示了随着时间的推移，检入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 到 2004 年 9 月底的近 900 个独立实例。MapReduce 之所以如此成功，是因为它使编写一个简单的程序成为可能，并在半小时的时间内在 1000 台机器上高效地运行它，大大加快了开发和原型开发周期。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利用大量的资源。

在每个作业结束时，MapReduce 库会记录该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 月在谷歌上运行的 MapReduce 作业子集的一些统计数据。

6.1 大规模索引

到目前为止，我们对 MapReduce 最重要的用途之一是完全重写了生产索引-

ing 系统，它产生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，存储为一组 GFS 文件。这些文档的原始内容是超过 20tb 的数据。索引过程以 5 到 10 个 MapReduce 操作的序列运行。使用 MapReduce(而不是索引系统先前版本中的临时分布式传递)提供了几个好处

适合:

- 索引代码更简单、更小、更容易理解，因为处理容错、分布和并行的代码隐藏在 MapReduce 库中。例如，当使用 MapReduce 表达时，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，而不是将它们混在一起，以避免对数据的额外传递。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需要几个月才能完成的一个更改，在新系统中只需要几天就可以实现。

- 索引过程变得更容易操作，因为大多数由机器故障、机器慢速和网络故障引起的问题都由 MapReduce 库自动处理，而无需操作员干预。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。

7 相关工作

许多系统提供了受限的编程模型，并利用这些限制来自动并行化计算。例如，可以在 N 个处理器上使用并行前缀计算[6,9,13]，在 log N 时间内对 N 个元素数组的所有前缀进行关联函数计算。MapReduce 可以被认为是基于我们对大型真实世界计算的经验，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。这些系统和 MapReduce 的一个关键区别是，MapReduce 利用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活动磁盘[12,15]等技术中获得灵感，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。我们在少量磁盘直接连接的商用处理器上运行，而不是直接在磁盘控制器处理器上运行，但一般方法是类似的。

我们的备份任务机制类似于 Charlotte 系统[3]中采用的及早调度机制。简单的及早调度的缺点之一是，如果给定的任务导致重复失败，则整个计算无法完成。我们用我们的跳过坏记录的机制来解决这个问题的一些实例。

MapReduce 的实现依赖于一个内部的集群管理系统，该系统负责在大量的共享机器集合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 Condor[16]等其他系统相似。

MapReduce 库中的排序功能与 NOW-Sort[1]的操作类似。Source machines (map worker)对要排序的数据进行分区，并将其发送给 R reduce worker 中的一个。每个 reduce worker 在本地(如果可能的话在内存中)对其数据进行排序。当然现在 sort 没有用户自定义的 Map 和 Reduce 函数，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。与 MapReduce 一样，River 系统试图提供良好的平均案例性能，即使是在异构硬件或系统扰动所引入的不均匀性存在的情况下。River 通过仔细调度磁盘和网络传输来实现这一点，以实现平衡的完成时间。MapReduce 则有不同的方法。通过限制编程模型，MapReduce 框架能够将问题划分为大量的细粒度任务。这些任务被动态地调度到可用的 worker 上，以便更快的 worker 处理更多的任务。受限编程模型还允许我们在临近作业结束时调度任务的冗余执行，这在存在不均匀(例如缓慢或卡住的工人)的情况下大大减少了完成时间。

BAD-FS[5]与 MapReduce 有着非常不同的编程模型，与 MapReduce 不同的是，它的目标是

在广域网上执行作业。然而，有两个基本的相似之处。(1)两个系统都使用冗余执行来从故障造成的数据丢失中恢复。(2)两种系统都使用局部感知调度来减少在拥塞的网络链路上发送的数据量。

TACC[7]是一种旨在简化高可用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 的结论

MapReduce 编程模型已经在谷歌上成功地用于许多不同的目的。我们将这种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、局部性优化和负载平衡的细节。第二，各种各样的问题都可以很容易地表达为 MapReduce 计算。例如，MapReduce 被用于为谷歌的生产 web 搜索服务生成数据，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，我们开发了一个 MapReduce 的实现，它可以扩展到由数千台机器组成的大型机器集群。该实现有效利用了这些机器资源，因此适合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，限制编程模型使并行化和分布式计算变得容易，并使这样的计算具有容错性。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针对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，将中间数据的一份副本写入本地磁盘可以节省网络带宽。第三，可以使用冗余执行来减少机器速度慢的影响，并处理机器故障和数据丢失。

确认

Josh Levenberg 在修改和扩展用户级 MapReduce API 方面发挥了重要作用，基于他使用 MapReduce 的经验和其他人的增强建议，他提供了许多新功能。MapReduce 从谷歌文件系统[8]读取输入，并将输出写入。我们要感谢 Mohit Aron, Howard Gobioff, Markus Gutschke，

David Kramer、Shun-Tak Leung 和 Josh Redstone 在开发 GFS 方面所做的工作。我们也要感谢 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 Levenberg、Sharon Perl、Rob Pike 和 Debby Wallach 对本文的早期草稿提供了有益的意见。匿名的 OSDI 审稿人和我们的牧羊人 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，我们感谢谷歌工程组织内所有 MapReduce 的用户，他们提供了有用的反馈、建议和 bug 报告。

参考文献

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein 和 David A. Patterson。工作站网络上的高性能排序。*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 和 Kathy Yelick。Cluster I/O with River: Making the fast case common。*第六届并行和分布式系统输入/输出研讨会论文集(IOPADS ' 99)*，第 10-22 页，乔治亚州亚特兰大，1999 年 5 月。

[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem 和 Peter Wyckoff。Charlotte: metcomputing on the web。发表于 1996 *年第 9 届国际并行与分布式计算系统会议论文集*。

[4] Luiz A. Barroso, Jeffrey Dean, and Urs H ¨ olzle。网络搜索星球:谷歌集群架构。*IEEE Micro*, 23(2): 22-28, 2003 年 4 月。

[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Miron Livny。批处理感知的分布式文件系统中的显式控制。*第一届 USENIX 网络系统设计与实现研讨会论文集，NSDI*, 2004 年 3 月。

[6] Guy E. Blelloch。扫描作为原始的并行操作。*IEEE 计算机汇刊*, C-38(11)， 1989 年 11 月。

[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, Paul Gauthier。基于集群的可扩展网络服务。第 16 *届 ACM 操作系统原理研讨会论文集*，第 78 - 91 页，法国圣马洛，1997 年。

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung。谷歌文件系统。*《第 19 届操作系统原理研讨会》*，第 29-43 页，纽约乔治湖，2003 年。

S. Gorlatch。扫描和其他列表同态的系统高效并行化。In L. Bouge, P. Fraigni- aud, A. Mignotte, and Y. Robert, editors, *Euro-Par ' 96。并行处理*，计算机科学讲义 1124，页 401-408。斯普林格出版社,1996 年版。

[10]吉姆·格雷。排序基准主页。http://research.microsoft.com/barc/SortBenchmark/。

[11] William Gropp, Ewing Lusk, Anthony Skjellum。*使用 MPI:带有消息传递接口的可移植并行编程*。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. Satyanarayanan、G. R. Ganger、E. Riedel、A. Ailamaki。Diamond:交互式搜索中早期丢弃物的存储架构。发表于 *2004 年 4 月的 USENIX 文件和存储技术快速会议论文集*。

[13] Richard E. Ladner和 Michael J. Fischer。并行前缀计算。*计算机学报，27*(4):831 - 838,1980。

[14] Michael O. Rabin。为了安全、负载均衡和容错而高效散布信息。*计算机学报，36*(2):335 - 348,1989。

Erik Riedel, Christos Faloutsos, Garth A. Gibson 和 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，米伦·利夫尼。分布式计算实践:秃鹰体验。*并发与计算:实践与经验*，2004。

[17] L. G. Valiant。并行计算的桥接模型。*计算机学报，33*(8):103-111,1997。

[18]吉姆·威利。Spsort:如何快速对 tb 级数据进行排序。http://alme1.almaden.ibm.com/cs/spsort.pdf。

一个词频

这一节包含一个程序，它计算命令行上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”

//用户的 map 函数

类 WordCounter: public Mapper
{public:

  虚拟虚空映射(const MapInput& input)
    {const string& text = input.value();Const
    int n = text.size();

    For (int I = 0;I < n;){

      //跳过前导空格

      While ((i < n) && isspace(text[i]))
        i++;

      //查找单词结尾

      Int start = i;

      While ((i < n) && !isspace(text[i]))
        i++;

        If (start < i)

          发出(text.substr(开始,我开始),"
          1 ");

    }

  }

};

REGISTER_MAPPER(wordcount);

//用户的 reduce 函数

class addr: public Reducer {

  virtual void Reduce(ReduceInput* input){//
    用相同的键遍历所有条目，并将值相
    加

    Int64 value = 0;

    While (!input->done()) {

      value += StringToInt(input->value());输
      入->NextValue ();

    }

    //输入和->key()

    发出(IntToString(值));}

};

REGISTER_REDUCER(加法器);

Int main(Int argc, char** argv) {

  ParseCommandLineFlags(命令行参数
  个 数 ,argv);MapReduceSpecification
  规范;

  //将输入文件列表存储到“spec”for
  (int i = 1;I< argc;我++){

    MapReduceInput* input = spec.add_input();
    输入->set_format(“文本”);

    输入->set_filepattern
    (argv[我]);

    输入-> set_mapper_class(“wordcount”);}

  //指定输出文件:

  // /gfs/test/freq-00000-of-00100 //
  /gfs/test/freq-00001-of-00100 //…

  MapReduceOutput* out =
  spec.output();——> set_filebase(“/ gfs
  /测试/频率”);——> set_num_tasks
  (100);

  ——>set_format(“文
  本”);

  ——>set_reducer_class(“毒
  蛇”);

  //可选:在 map 任务内做部分和//节省网
  络带宽 out-
  >set_combiner_class(“Adder”);

  //调优参数:每个任务最多使用 2000 台机器
  和 100 MB 内存 spec.set_machines(2000);

  spec.set_map_megabytes (100);

  spec.set_reduce_megabytes (100);

  //现在运行它

  MapReduceResult 结果;

  如果(!MapReduce(spec， &result))
  abort();

  // Done: `result `结构包含有关计数器、
  花费的时间、使用的机器数量等信息
```

```
返回 0;                                                    }
```