# 绪论

MapReduce是一个处理和生成大规模数据集的编程模型。用户定义一个map函数来处理key/value对，一个reduce函数来归并拥有相同key的所有中间值.许多现实世界中的任务表现出了这种方法，在这片论文中有所列举。 使用这种函数式编程的方法写出的程序在大规模集群中自动并行化，这个实时系统关注输入数据的分割，在一系列集群中进行程序调度执行，管理集群之间的数据通信。允许没有任何并行和分布式系统经验的程序员简单地去处理大规模的资源。 我们的MapReduce的实现运行在规模可以灵活调整的由普通机器组成的机群上：一个典型的MapReduce计算过程处理几千台机器上的以TB计算的数据。程序员发现这个系统很容易使用：每天的google集群上都有成千上百的程序已经使用并且高达几千的MapReduce程序

# 1.简介

在过去的五年中，作者和google的其他同事已经实现了几百个特定功能的计算模型来处理大规模数据，例如抓取文档，网络请求日志等，还有计算各种衍生数据，例如倒排索引,Web文档的图结构的各种表示,每个主机中文档抓取的总结，一天之中最频繁访问的链接等。绝大部分计算是直观的。但是，输入数据通常是很大的而且为了在一个合情合理的时间内完成计算，通常将计算分布在成百上千的机器中。怎样并行计算,分发数据,处理错误,所有这些问题综合在一起,使得原本很简介的计算,因为要大量的复杂代码来处理这些问题,而变得让人难以处理. 作为对这个复杂性的回应，我们设计了一个允许我们去表达简单的计算但是将复杂的并行细节，容错处理，数据分布和负载均衡屏蔽掉的新的抽象概念。我们的抽象概念的灵感来自于最先出现在LISP和其他函数式编程语言中的中的map和reduce。我们意识到大多数我们的计算模型涉及到应用一个map操作来对每一个逻辑“记录”在我们的输入中来确认计算一系列的中间的key/value对，然后使用reduce为了精确地结合衍生的数据，对所有的有相同key的数据进行操作。我们使用用户特定的函数式模型map和reduce允许我们轻便的进行并行大规模计算和使用再次执行作为初级机制来实现容错. 这个工作主要贡献是使用简单并且强大的接口来自动进行并行和大规模分布式计算，通过结合这个接口能够在大量的普通PC上实现高性能的运算。 第二部分描述了基本的编程模型，并且给定了一些例子。第三部分描述了MapReduce得一个应用，定义了基于集群的计算环境。第四章描述了一些我们发现非常有用的对这一编程模型的改进，第五部分包含大量的任务中性能的衡量。第六部分探索了在Google内部使用MapReduce作为基础来重写我们的索引系统产品。第七章讨论了相关内容和未来的工作。

**2.编程模型**

计算模型接受一系列的key/value对，生成一系列key/value对。MapReduce库的使用者通过Map和Reduce函数来表达计算。 Map，由使用者定义。接受一个key/value列表，生成一系列的**中间值** key/value对。MapReduce库 集合所有具有相同中间值 key I的序对并且将他们传递给Reduce函数。 Reduce也是由使用者定义，接受一个中间值key I和一系列它的值。reduce函数将这些值合并起来并且形成一个比较小的value集，通常每次reduce只会产生0或者1。这些中间值将会通过一个迭代器应用到用户的reduce函数中。这允许我们处理在内存中过于多的列表值。

**2.1 举例**

考虑这个问题:计算在一个大的文档集合中每个词出现的次数.用户将写和下面类似的伪代码:

map(String key,String value):

//key : 文档名

//value: 文档内容

for each word w in value:

EmitIntermediate(w,"1");

reduce(String key,Iterator values):

//key:单词

//values:计数列表

int result = 0;

for each v in values:

result += ParseInt(v);

Emit(AsString(result));

map函数产生每个词和这个词的出现次数(在这个简单地例子里就是1)reduce函数将特定词的所有次数加在一起 另外，用户用输入输出文件的名字和可选的调节参数来填充一个mapreduce规范对象。然后用户引用MapReduce函数，向它传递特定的对象。用户的代码和MapReduce的库（C++实现）相互连接。附录A包含这个例子的一个完整的程序。

**2.2 类型**

精工我们之前提供的伪代码采用字符串的输入输出，概念上来讲，map和reduce函数有关联的类型：

map (k1,v1) -> list(k2,v2)

reduce (k2,list(v2)) ->list(v2)

例如，输入的key,value和输出的key,value的域不同.或者，中间值的key和value和输出的key和value来自相同的域。 我们的C++应用通过传递字符串来和用户定义的函数进行通信，并且将它留给了用户的代码在字符串和合适的类型中去转换。

**更多的例子**

这里有一些简单地有趣的程序能够解释MapReduce计算模型 **分布式Grep** map函数操做匹配到提供的模式的行。reduce函数是一个标记函数，仅仅复制并且应用到中间值作为输出 **计算URL连接的频繁度** map函数处理网页请求的日志并且输出<URL,1>然后reduce函数增加所有具有相同url的值在一起并且触发一个<URL，total count>序对。 **倒转网络链接图**map函数为每个url页面中的任何一个链接生成<target,source>序对，target是一个URL叫做目标,包含这个URL的页面叫做源.reduce函数根据给定的相关目标URLs连接所有的源URLs形成一个列表,产生target,list(source)对. **每个主机的术语向量**一个术语向量用一个(词,频率)列表来概述出现在一个文档或一个文档集中的最重要的一些词.map函数为每一个输入文档产生一个(主机名,术语向量)对(主机名来自文档的URL).reduce函数接收给定主机的所有文档的术语向量.它把这些术语向量加在一起,丢弃低频的术语,然后产生一个最终的(主机名,术语向量)对. [图片1] **倒排索引** map函数分析每个文档,然后产生一个(词,文档号)对的序列.reduce函数接受一个给定词的所有对,排序相应的文档IDs,并且产生一个(词,文档ID列表)对.所有的输出对集形成一个简单的倒排索引.它可以简单的增加跟踪词位置的计算. **分布式排序**:map函数从每个记录提取key,并且产生一个(key,record)对.reduce函数不改变任何的对.这个计算依赖分割工具(在4.1描述)和排序属性(在4.2描述).

**3. 应用**

MapReduce的接口有很多实现，正确的选择依赖于环境。例如，一个实现可能适合于小型的共享内存机器，另一个适用于大型的NUMA处理器，或者另一个for一个大型的网络机器的集合 这章描述一个基于目标环境的在google中广为使用的实现：用交换机连接的普通PC机的大机群，在我们的环境中： 1. 机器通常是多核处理器X86架构运行Linux的内存在2-4GB 2. 通用网络硬件使用100mb/s或者1gb/s的机器级别，但是平均小于全部带宽的一半 3. 一个集群包含成千上百的机器，因此机器失败是很普遍的 4. 直接连到每个机器上的廉价IDE硬盘。一个内部分布式文件系统来处理这些存在硬盘上的数据。这个数据系统使用冗余复制的方式在不可靠的的机器上保证可靠性和有效性. 5. 用户提交工作给调度系统。每个工作包含一系列的任务,每个工作被调度者映射到机群中一个可用的机器集上

**3.1 执行总览**

map调用通过自动分割输入数据成一个有M个split的集被分布到多台机器上.输入分割可能被并行的使用在不同的机器中。reduce也是分布式的，调用使用一个分割函数（例如：hash(key)mod R）通过分割中间值的key为R片。分片数目R和分片函数由用户来定义。 图1展示了我们的实现中整个MapReduce操作的流程。当用户程序调用MapReduce函数时，以下的行动序列发生（图一中的数字序列对应于下面列表中的数字） 1. 用户程序中的MapReduce库开始分割文件为M块，大小通常为16MB到64MB每块（通过配置参数由用户定义）。然后它开始在集群中执行此程序 2. 其中一个程序是特殊的--master，其他的程序的工作都是由master分配的。这里有M个map任务和R个reduce任务去分配，master挑选出其中空闲的机器进行map任务或者reduce任务。 3. 被分配了map任务的worker读取相关输入切分的内容。它从输入数据中解析key/value对，然后将它们传递给用户定义的Map函数。Map函数生成的中间值key/value对存在内存的缓存中。 4. 周期性的，缓存中的序对被写到硬盘中，被分割函数分割成R个区域。这些缓存在本地磁盘的位置被返回到master中，以便master分配reduce任务的时候使用 5. 当一个reduce机器被master通知这些位置时，它从map任务的本地磁盘中使用远程读取这些存取的数据。当一个reduce机器读完所有的中间值，它通过中间key进行排序，以便所有的具有相同key的值聚在一起。排序是必须的，因为许多不同keys的map有相同的reduce任务。如果中间值的数据在内存中太大，还需要一个外部排序. 6. reduce机器迭代排序过的中间数据，对每个独立的中间key值进行累加，它传递了key和相符合的中间值的集合给用户的Reduce程序。 7. 当所有的map任务和reduce任务都完成。master执行用户的程序，在此刻，管理者唤醒用户程序.在这个时候,在用户程序里的MapReduce调用返回到用户代码.

在所有成功进行后，mapreduce输出结果已经可以使用（每一个都是reduce的任务，文件名由用户来定义）。特别的，用户不需要合并这个R个输出文件到一个文件，他们通常把这些文件当做输入文件放到另外一个mapreduce程序中，或者用户使用其他的分布式应用，他们能够处理这些分块输入文件。

**3.2 master的数据结构**

master有许多数据结构。每个map任务和reduce任务，它存储任务的状态(空闲，处理中，完成)，和用户机的身份识别(对于非空闲的任务) master就像一个管道,通过它,中间文件区域的位置从map任务传递到reduce任务.因此，对于每一项完成的任务,master存储地址和map过程生成的R中间值文件区域和大小。当map任务完成时，更新这些地址和大小信息，这些信息被逐步增加的传递给那些正在工作的reduce任务

**3.3 容错处理**

MapReduce库被设计为去帮助分布式的运行超大规模的数据，所以这个库必须要能很好的处理机器故障. **worker机器故障**master周期性的ping每个worker.如果master在一个确定的时间段内没有收到worker返回的信息,那么它将把这个worker标记成失效.因为每一个由这个失效的worker完成的map任务被重新设置成它初始的空闲状态,所以它可以被安排给其他的worker.同样的,每一个在失败的worker上正在运行的map或reduce任务,也被重新设置成空闲状态,并且将被重新调度. 在一个失败机器上已经完成的map任务将被再次执行,因为它的输出存储在它的磁盘上,所以不可访问.已经完成的reduce任务将不会再次执行,因为它的输出存储在全局文件系统中. 当一个map任务首先被worker A执行之后,又被B执行了(因为A失效了),重新执行这个情况被通知给所有执行reduce任务的worker.任何还没有从A读数据的reduce任务将从worker B读取数据. MapReduce可以处理大规模worker失败的情况.例如,在一个MapReduce操作期间,在正在运行的机群上进行网络维护引起80台机器在几分钟内不可访问了,MapReduce master只是简单的再次执行已经被不可访问的worker完成的工作,继续执行,最终完成这个MapReduce操作.

**master失败** 可以很容易的让管理者周期的写入上面描述的数据结构的checkpoints.如果这个master任务失效了,可以从上次最后一个checkpoint开始启动另一个master进程.然而,因为只有一个master,所以它的失败是比较麻烦的,因此我们现在的实现是,如果master失败,就中止MapReduce计算.客户可以检查这个状态,并且可以根据需要重新执行MapReduce操作

**在错误面前的处理机制** 当用户提供的map和reduce操作对它的输出值是确定的函数时,我们的分布式实现产生,和全部程序没有错误的顺序执行一样,相同的输出. 我们依赖对map和reduce任务的输出进行原子提交来完成这个性质.每个工作中的任务把它的输出写到私有临时文件中.一个reduce任务产生一个这样的文件,而一个map任务产生R个这样的文件(一个reduce任务对应一个文件).当一个map任务完成的时候,worker发送一个消息给master,在这个消息中包含这R个临时文件的名字.如果master从一个已经完成的map任务再次收到一个完成的消息,它将忽略这个消息.否则,它在master的数据结构里记录这R个文件的名字. 当一个reduce任务完成的时候,这个reduce worker原子的把临时文件重命名成最终的输出文件.如果相同的reduce任务在多个机器上执行,多个重命名调用将被执行,并产生相同的输出文件.我们依赖由底层文件系统提供的原子重命名操作来保证,最终的文件系统状态仅仅包含一个reduce任务产生的数据. 我们的map和reduce操作大部分都是确定的,并且我们的处理机制等价于一个顺序的执行的这个事实,使得程序员可以很容易的理解程序的行为.当map或/和reduce操作是不确定的时候,我们提供虽然比较弱但是合理的处理机制.当在一个非确定操作的前面,一个reduce任务R1的输出等价于一个非确定顺序程序执行产生的输出.然而,一个不同的reduce任务R2的输出也许符合一个不同的非确定顺序程序执行产生的输出. 考虑map任务M和reduce任务R1,R2的情况.我们设定e(Ri)为已经提交的Ri的执行(有且仅有一个这样的执行).这个比较弱的语义出现,因为e(R1)也许已经读取了由M的执行产生的输出,而e(R2)也许已经读取了由M的不同执行产生的输出.

**3.4存储位置**

在我们的计算机环境里,网络带宽是一个相当缺乏的资源.我们利用把输入数据(由GFS管理)存储在机器的本地磁盘上来保存网络带宽.GFS把每个文件分成64MB的一些块,然后每个块的几个拷贝存储在不同的机器上(一般是3个拷贝).MapReduce的master考虑输入文件的位置信息,并且努力在一个包含相关输入数据的机器上安排一个map任务.如果这样做失败了,它尝试在那个任务的输入数据的附近安排一个map任务(例如,分配到一个和包含输入数据块在一个switch里的worker机器上执行).当运行巨大的MapReduce操作在一个机群中的一部分机器上的时候,大部分输入数据在本地被读取,从而不消耗网络带宽.

**3.5任务粒度**

象上面描述的那样,我们细分map阶段成M个片,reduce阶段成R个片.M和R应当比worker机器的数量大许多.每个worker执行许多不同的工作来提高动态负载均衡,也可以加速从一个worker失效中的恢复,这个机器上的许多已经完成的map任务可以被分配到所有其他的worker机器上. 在我们的实现里,M和R的范围是有大小限制的,因为master必须做O(M+R)次调度,并且保存O(M*R)个状态在内存中.(这个因素使用的内存是很少的,在O(M*R)个状态片里,大约每个map任务/reduce任务对使用一个字节的数据). 此外,R经常被用户限制,因为每一个reduce任务最终都是一个独立的输出文件.实际上,我们倾向于选择M,以便每一个单独的任务大概都是16到64MB的输入数据(以便上面描述的位置优化是最有效的),我们把R设置成我们希望使用的worker机器数量的小倍数.我们经常执行MapReduce计算,在M=200000,R=5000,使用2000台工作者机器的情况下.

**3.6备用任务**

一个落后者是延长MapReduce操作时间的原因之一:一个机器花费一个异乎寻常地的长时间来完成最后的一些map或reduce任务中的一个.有很多原因可能产生落后者.例如,一个有坏磁盘的机器经常发生可以纠正的错误,这样就使读性能从30MB/s降低到3MB/s.机群调度系统也许已经安排其他的任务在这个机器上,由于计算要使用CPU,内存,本地磁盘,网络带宽的原因,引起它执行MapReduce代码很慢.我们最近遇到的一个问题是,一个在机器初始化时的Bug引起处理器缓存的失效:在一个被影响的机器上的计算性能有上百倍的影响. 我们有一个一般的机制来减轻这个落后者的问题.当一个MapReduce操作将要完成的时候,master调度备用进程来执行那些剩下的还在执行的任务.无论是原来的还是备用的执行完成了,工作都被标记成完成.我们已经调整了这个机制,通常只会占用多几个百分点的机器资源.我们发现这可以显著的减少完成大规模MapReduce操作的时间.作为一个例子,将要在5.3描述的排序程序,在关闭掉备用任务的情况下,要比有备用任务的情况下多花44%的时间.

**4. 优化**

尽管提供的基础的方法来书写map函数和reduce函数在大多数条件下是很有效的。我们一经发现一些有用的扩展，我们将在这章呈现。

**4.1 分割函数**

MapReduce指出了reduce任务/输出文件的数量为R，数据通过这些任务所提供的中间值的key使用分割函数来进行分割。一个默认的分割函数使用hash（例如：hash(key) mod R），它能够得到完全均衡的分片。尽管在一些例子中，通过其它函数的key来进行分割也是非常有用的。例如：有些时候输出的键是URL，我们想要一个主机下所有入口输出到同一个文件。为了支持这种情况。MapReduce库提供了一种特别的分割函数，例如，使用hash(Hostname(urlkey) mod R)当做分割函数，这样将所有来自同一个host的url存放到一个输出文件中

**4.2 顺序保证**

我们保证在一个给定的分片内，它的中间值key/value对会按照升序的顺序处理。这个顺序保证使得每个分隔片生成一个有序输出文件变得简单，当输出文件需要提供通过key的高效的随机连接或者输出的用户去排序这些数据是非常有用。

**4.3 Combiner 函数**

在一些情况下，每个map任务将会产生大量重复的中间key值，而且用户定义的Reduce函数满足交换律和结合律,2.1节的统计字数就是一个好的例子。因为词频趋向于符合Zipf分布，每一个map任务将会生产出形式为<the,1>的成百上千的记录。所有的记录将会通过网络传递到一个reduce任务处，然后将他们累加起来，通过Reduce函数生产出一个数字。我们允许用户使用一个可选择的Combiner函数在通过网络传输之前来进行分片的合并工作。 这个Combiner函数在每个机器上表现为一个map任务去执行，典型的相同的代码用在了Combiner和reduce函数中。combiner函数和reduce函数的区别在于MapReduce库如何处理这些函数的输出。这些reduce函数的输出被写入最终的输出文件。combiner函数的输出被写入中间文件，然后combiner函数会被送到reduce任务中去。 局部的结合大大增加了MapReduce操作的速度。附录A包含了一个使用combiner的例子。

**4.4 输入和输出的类型**

MapReduce类提供了几种不同的输入数据的方法，例如，“text”输入模式将每一行看成是key/value对：key是在文件中的偏移量，value是这行的内容。另一个常见的是通过key的排序来格式化存储一序列的key/value对。每一个输入类型的实现知道如何将自己分割成有意义的供不同的map任务处理的范围（例如，text模式的分割范围保证仅仅在线性关系下进行分割）。用户可以通过实现一个简单地reader接口来增加一个新的输入类型的支持。尽管大部分用户仅仅使用小数量提前定义的输入类型中的一个。 reader函数不需要从文件中提供数据，可以很轻易的定义一个reader来从数据库中读取记录，或者从内容的数据结构中。 同样，我们支持一系列的输出类型去生产不同格式化的数据，并且用户编码区增加新的输出类型也是很简单的。

**4.5 副作用**

在某些方面来说，MapReduce的用户已经发现从他们的map或者reduce操作中编写辅助文件当做额外的输入是非常简单地。我们依赖编码者去让这些副作用原子化。通常应用被写到了一个临时文件此时当它被完全生成时已经自动重新命名了这个文件。 在一个单独的任务中生成多输出文件时，我们没有分别提供以上两个歩奏。因此，通过跨文件的一致性需求来生产多重输出文件是合情合理的。这个问题在实际使用中不是问题。

**4.6 跳过错误记录**

有时候用户的代码中的错误会导致map函数或者reduce函数在某些特定的记录中奔溃。一些错误防止mapReduce操作完成。通常的举措是弥补这些错误，但是有时是不现实的。也许这个错误由源码不可达的第三方的库文件提供。或者有时跳过几个记录是可以接受的，当我们做大规模数据集的统计分析时，我们提供了一个可选的模式来运行，当MapReduce库发现记录导致程序奔溃之后，跳过这些记录来进行后续的程序。 每一个worker进程启动一个信号处理器来捕捉内存段异常和总线错误。在应用用户的Map或者Reduce操作之前，MapReduce库也将序列的长度存储在了全局变量中。如果用户的代码抛出一个信号量， 然后信号量处理器发送一个包含序号的“last gasp”UDP包给master

**4.7 本地执行**

在MapReduce函数中排出错误是很难的，因为实际的计算发生在一个分布式系统中，通常在几千个机器中，由master动态的分配任务。为了方便排出错误和进行调试，我们开发了一个MapReduce的一个替代的应用，能够在本地机器上执行所有的MapReduce操作。由用户提供控制，因此计算能够执行有限的特定的map任务。用户通过一个特定的标志来调用他们的代码。然后能够轻松的使用任何他们觉得有用的任何调试和测试工具（例如.gdb）

**4.8 状态信息**

主机启动一个内部的HTTP服务器，并且为用户导出一系列的状态页面。状态页面表现出计算的进度，例如有多少任务被完成了，多少在进行之中，输入数据的大小，中间值数据的大小，输出数据的大小，进行的速率等。这个页面也包含了一些标准错误和标准输出文件的链接。用户可以使用这些数据去预测计算所花费的时间和是否有更多的数据资源应该被加入到计算中。这些页面也能够支出什么时候计算数据比期待的要少 另外，首要的状态页面输出哪个worker失败了，哪一个map/reduce任务在进行中失败。当我们需要解决问题时这些信息是非常有用的。

**4.9 计数器**

MapReduce库提供了一个计数器的工具箱,来计算各种事件的发生次数。例如，用户的代码有可能希望计算所有单词的数量或者被索引的德文文档的数量等。 为了使用这个工具箱，用户代码创建一个计数器对象然后合适的在map或者reduce函数中递增。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");

来自不同worker机器上的计数器值被周期性的传送给master(在ping回应里).master把来自成功的计数器的map值和reduce任务的值加起来并且当mapreduce操作完成时返回用户代码。当前的计数器的值也表现在master的状态页中，因此用户能够看到计算实时运行。当聚集计数的值，master消除相同map或者reduce任务重复执行的影响以避免重复求值。多重执行产生于用户使用的备份文件和 由于错误重新执行的任务 一些计数器的值是由MapReduce类自动均衡，例如输入的key/value对的数量和生产处的key/value对的数目。 用户已经发现在保证MapReduce操作正常进行的情况下，计数工具箱是非常有用的。例如：一些MapReduce操作，用户代码有可能想要保证输入的key/value对和输出的key/value对必须相等。或者处理过的德文文档的数量是在全部被处理的文档数量中属于合理的范围

**5.性能**

在这章，我们用在一个大型集群上运行的两个计算来衡量MapReduce的性能。一个计算从1TB的数据中找寻特定的匹配。另一个排序大约1TB的数据。 这两个程序代表了MapReduce的用户实现的真实的程序的一个大子集.一类是,把数据从一种表示转化到另一种表示.另一类是,从一个大的数据集中提取少量的关心的数据.

**5.1 集群配置**

所有的程序都在一个包含1800台机器的集群中执行。每一个机器有2个2GHz的Intel Xeon超线程处理器，4GB内存，2个160GB的硬盘和1GB的以太网连接。这个机器被排列在有一个根节点大约在100-200Gbps的带宽的一个二级树形交换网络上。所有的机器都有相同部署，因此任意两点之间的速度小于1ms 在4GB内存中，大约1-1.5GB被其他运行在集群中的任务占据。这些程序运行在一个周末的下午，当CPU,硬盘，网络都基本处于空闲状态。

**5.2 Grep**

grep程序大概扫描了10^10，每个100比特的记录，查出相对少的3个字母的组合（存在于92337个记录中）。输入数据被分成大概64MB的片（M=15000）,整个的输出是一个文件（R=1） 图表2表现了这个过程计算的进度，Y轴表示输入数据被扫描的速度，随着更多的机群被分配给这个MapReduce计算,速度在逐步的提高，当有1764个worker被分配时达到顶峰为超过30GB/s。当map任务完成时，速率逐渐降低，当到80秒时，速率为0.整个计算花费了大概150秒。包含一秒钟的启动时间。启动时间用来把程序传播到所有的机器上。等待GFS打开1000个输入文件,得到必要的位置优化信息.

**5.3 排序**

sort程序排序10^10，每个100比特的记录（大概1TB数据）。这个程序是模仿TeraSort的。 排序程序包含少于50行的用户数据，一个三行的Map函数从文件行得到10比特的排序key，并且产生一个由这个key和原始文本行组成的中间key/value对。我们使用内置的identity函数当做reduce操作符。这个函数传入中间key/value对当做reduce的output对。最终的排序输出2路复制的GFS文件中(也就是,程序的输出会写2TB的数据) 从前面来看，输入数据被分成了64MB的分片（M=15000），我们分开这些排序过的输出为4000个文件（R=4000）。分割函数使用key的原始字节来把数据分区到R个小片中。 我们以这个基准的分割函数,知道key的分布情况.在通常的排序程序中，我们能够加入提前通过的MapReduce操作符，他能够收集一些keys的样本然后使用这些样本Key的分布为最终的排序来计算分割点。 图标3（a）表现出一般执行排序程序的过程，左上角的图表现出输入数据读取速率，速度达到13GB/s的峰值然后迅速下降，因为所有的map任务都已经在200s内完成。注意到输入的速率小于grep的速率。因为排序函数花费一半的时间和I/O带宽在向本地磁盘写中间值的输出结果。在同种情况下，grep的中间值输出结果可以忽略不计。 左边中间的表格表现出数据通过网络从map任务到reduce任务的速率。当第一个map任务完成时开始有速率。图中的第一个峰是启动了大概1700个reduce任务（所有的MapReduce被分配给1700个机器，每一个机器在一个时间内最多执行一个reduce任务），计算时间大概花费了300秒。一些reduce任务完成了，然后我们开始执行剩下的的reduce任务。所有的网络任务在600秒内完成。 左边下面的图表明了reduce任务中排序过得数据被写入到最终输出文件的速率。第一个计算时期的末尾和开始写入之间有一个延时。因为机器正在忙着排序中间数据。写入持续的速率为2-4GB/s一段时间。所有的写入完成在850秒左右。包括启动的时间，整个计算过程花费891秒的时间。这个和TeraSort benchmark的最好纪录1057秒差不多。 一些注意的事项：因为我们区域的优化，大部分的数据是从本地磁盘而不是有限的带宽，所以输入速率比计算速率和输出速率要高。计算速率比输出速率要高因为输出阶段写入两份排好序的数据（处于可靠性和可用性的原因，我们复制了一份输出数据），我们写两份的原因是因为底层文件系统的可靠性和可用性的要求。如果底层文件系统用类似容错编码(erasure coding)的方式,而不采用复制写的方式,在写盘阶段可以降低网络带宽的要求

**5.4 备份任务的影响**

在图3(b)中,显示我们不用备用任务的排序程序的执行情况.除了它有一个很长的几乎没有写动作发生的尾巴外,执行流程和图3(a)相似.在960秒后,只有5个reduce任务没有完成.然而,就是这最后几个落后者知道300秒后才完成.全部的计算任务执行了1283秒,多花了44%的时间.

**5.5 机器运行失败**

在图3(c)中,显示我们有意的在排序程序计算过程中停止1746台worker中的200台机器上的程序的情况.底层机群调度者在这些机器上马上重新开始新的worker程序(因为仅仅程序被停止,而机器仍然在正常运行). 因为已经完成的map工作丢失了(由于相关的map worker被杀掉了),需要重新再作，所以worker死掉会导致一个负数的输入速率.相关map任务的重新执行很快就重新执行了.整个计算过程在933秒内完成,包括了前边的启动时间(只比正常执行时间多了5%的时间).

**6.经验**

我们在2003年的2月写了MapReduce库的第一个版本,并且在2003年的8月做了显著的增强,包括位置优化,worker机器间任务执行的动态负载均衡,等等.从那个时候起,我们惊奇的发现MapReduce函数库广泛用于我们日常处理的问题.它现在在Google内部各个领域内广泛应用,包括:

* 大规模机器学习问题
* Google News和Froogle产品的机器问题.
* 提取数据产生一个流行查询的报告(例如,Google Zeitgeist).
* 为新的试验和产品提取网页的属性(例如,从一个web页的大集合中提取位置信息 用在位置查询).
* 大规模的图计算. 图4显示了我们主要的源代码管理系统中,随着时间推移,MapReduce程序的显著增加,从2003年早先时候的0个增长到2004年9月份的差不多900个不同的程序.MapReduce之所以这样的成功,是因为他能够在不到半小时时间内写出一个简单的能够应用于上千台机器的大规模并发程序,并且极大的提高了开发和原形设计的周期效率.并且,他可以让一个完全没有分布式和/或并行系统经验的程序员,能够很容易的利用大量的资源. 在每一个任务结束的时候,MapReduce函数库记录使用的计算资源的统计信息.在图1里,我们列出了2004年8月份在Google运行的一些MapReduce的工作的统计信息.

**6.1大规模索引**

到目前为止,最成功的MapReduce的应用就是重写了Google web 搜索服务所使用到的index系统.索引系统处理爬虫系统抓回来的超大量的文档集,这些文档集保存在GFS文件里.这些文档的原始内容的大小,超过了20TB.索引程序是通过一系列的,大概5到10次MapReduce操作来建立索引.通过利用MapReduce(替换掉上一个版本的特别设计的分布处理的索引程序版本)有这样一些好处: 索引的代码简单,量少,容易理解,因为容错,分布式,并行处理都隐藏在MapReduce库中了.例如,当使用MapReduce函数库的时候,计算的代码行数从原来的3800行C++代码一下减少到大概700行代码. MapReduce的函数库的性能已经非常好,所以我们可以把概念上不相关的计算步骤分开处理,而不是混在一起以期减少在数据上的处理.这使得改变索引过程很容易.例如,我们对老索引系统的一个小更改可能要好几个月的时间,但是在新系统内,只需要花几天时间就可以了. 索引系统的操作更容易了,这是因为机器的失效,速度慢的机器,以及网络失效都已经由MapReduce自己解决了,而不需要操作人员的交互.另外,我们可以简单的通过对索引系统增加机器的方式提高处理性能

**7 相关工作**

很多系统都提供了严格的设计模式,并且通过对编程的严格限制来实现自动的并行计算.例如,一个结合函数可以通过N个元素的数组的前缀在N个处理器上使用并行前缀计算在log N的时间内计算完.MapReduce是基于我们的大型现实计算的经验,对这些模型的一个简化和精炼.并且,我们还提供了基于上千台处理器的容错实现.而大部分并发处理系统都只在小规模的尺度上实现,并且机器的容错还是程序员来控制的. Bulk Synchronous Programming以及一些MPI primitives提供了更高级别的抽象,可以更容易写出并行处理的程序.这些系统和MapReduce系统的不同之处在,MapReduce利用严格的编程模式自动实现用户程序的并发处理,并且提供了透明的容错处理. 我们本地的优化策略是受active disks等技术的启发,在active disks中,计算任务是尽量推送到靠近本地磁盘的处理单元上,这样就减少了通过I/O子系统或网络的数据量.我们在少量磁盘直接连接到普通处理机运行,来代替直接连接到磁盘控制器的处理机上,但是一般的步骤是相似的. 我们的备用任务的机制和在Charlotte系统上的积极调度机制相似.这个简单的积极调度的一个缺陷是,如果一个任务引起了一个重复性的失败,那个整个计算将无法完成.我们通过在故障情况下跳过故障记录的机制,在某种程度上解决了这个问题. MapReduce实现依赖一个内置的机群管理系统来在一个大规模共享机器组上分布和运行用户任务.虽然这个不是本论文的重点,但是集群管理系统在理念上和Condor等其他系统是一样的. 在MapReduce库中的排序工具在操作上和NOW-Sort相似.源机器(map worker)分割将要被排序的数据,然后把它发送到R个reduce worker中的一个上.每个reduce worker来本地排序它的数据(如果可能,就在内存中).当然,NOW-Sort没有用户自定义的map和reduce函数,使得我们的库可以广泛的应用. River提供一个编程模型,在这个模型下,处理进程可以靠在分布式的队列上发送数据进行彼此通讯.和MapReduce一样,River系统尝试提供对不同应用有近似平均的性能,即使在不对等的硬件环境下或者在系统颠簸的情况下也能提供近似平均的性.River是通过精心调度硬盘和网络的通讯,来平衡任务的完成时间.MapReduce不和它不同.利用严格编程模型,MapReduce构架来把问题分割成大量的任务.这些任务被自动的在可用的worker上调度,以便速度快的worker可以处理更多的任务.这个严格编程模型也让我们可以在工作快要结束的时候安排冗余的执行,来在非一致处理的情况减少完成时间(比如,在有慢机或者阻塞的worker的时候). BAD-FS是一个很MapReduce完全不同的编程模型,它的目标是在一个广阔的网络上执行工作.然而,它们有两个基本原理是相同的. 1. 这两个系统使用冗余的执行来从由失效引起的数据丢失中恢复. 2. 这两个系统使用本地化调度策略,来减少通过拥挤的网络连接发送的数据数量. TACC是一个被设计用来简化高有效性网络服务结构的系统.和MapReduce一样,它通过再次执行来实现容错.

**8结束语**

MapReduce编程模型已经在Google成功的用在不同的目的.我们把这个成功归于以下几个原因:第一,这个模型使用简单,甚至对没有并行和分布式经验的程序员也是如此,因为它隐藏了并行化,容错,位置优化和负载均衡的细节.第二,大量不同的问题可以用MapReduce计算来表达.例如,MapReduce被用来,为Google的产品web搜索服务,排序,数据挖掘,机器学习,和其他许多系统,产生数据.第三,我们已经在一个好几千台计算机的大型集群上开发实现了这个MapReduce.这个实现使得对于这些机器资源的利用非常简单,因此也适用于解决Google遇到的其他很多需要大量计算的问题. 从这个工作中我们也学习到了一些东西.首先,严格的编程模型使得并行化和分布式计算简单,并且也易于构造这样的容错计算环境.第二,网络带宽是系统的瓶颈.因此在我们的系统中大量的优化目标是减少通过网络发送的数据量,本地优化使用我们从本地磁盘读取数据,并且把中间数据写到本地磁盘,以保留网络带宽.第三,冗余的执行可以用来减少速度慢的机器的影响,和控制机器失效和数据丢失.

**感谢**

Josh Levenberg校定和扩展了用户级别的MapReduce API,并且结合他的适用经验和其他人的改进建议,增加了很多新的功能.MapReduce从GFS中读取和写入数据.我们要感谢Mohit Aron,Howard Gobioff,Markus Gutschke,David Krame,Shun-Tak Leung,和Josh Redstone,他们在开发GFS中的工作.我们还感谢Percy Liang Olcan Sercinoglu 在开发用于MapReduce的集群管理系统得工作.Mike Burrows,Wilson Hsieh,Josh Levenberg,Sharon Perl,RobPike,Debby Wallach为本论文提出了宝贵的意见.OSDI的无名审阅者,以及我们的审核者Eric Brewer,在论文应当如何改进方面给出了有益的意见.最后,我们感谢Google的工程部的所有MapReduce的用户,感谢他们提供了有用的反馈,建议,以及错误报告等等.