# MapReduce\_Simplified Data Processing on Large Clusters

**2、编程模型（Programing Model）**

**Map：**map函数是用户自定义的函数，处理输入的key-value对，并且产生一组中间的key-value对，MapReduce函数库聚合所有相同的中间key-value对的values，并且发送给Reduce函数进行处理。

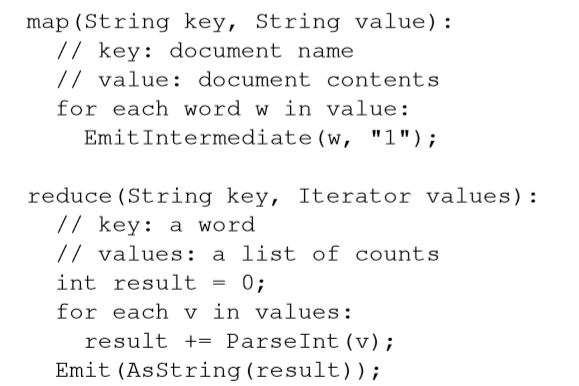
Map函数主要任务是读取原始数据，形成中间结果key-value数据。

**Reduce：**reduce函数同样也是用户提供的，它处理中间key-value，以及这个中间key-value相关的值集合。这个函数合并这些值，最后形成一个相对较小的值集合。通常一个单次Reduce执行会产生0个或者1个输出值。提供给Reduce函数的中间key-value是通过一个iterator来提供的，这就让我们可以处理超过内存容量的值列表。

Reduce函数主要任务是将相同key的数据聚合到一组。

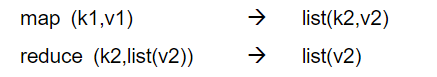
**例子**

在很大文档集合中统计每一个单词出现的次数：

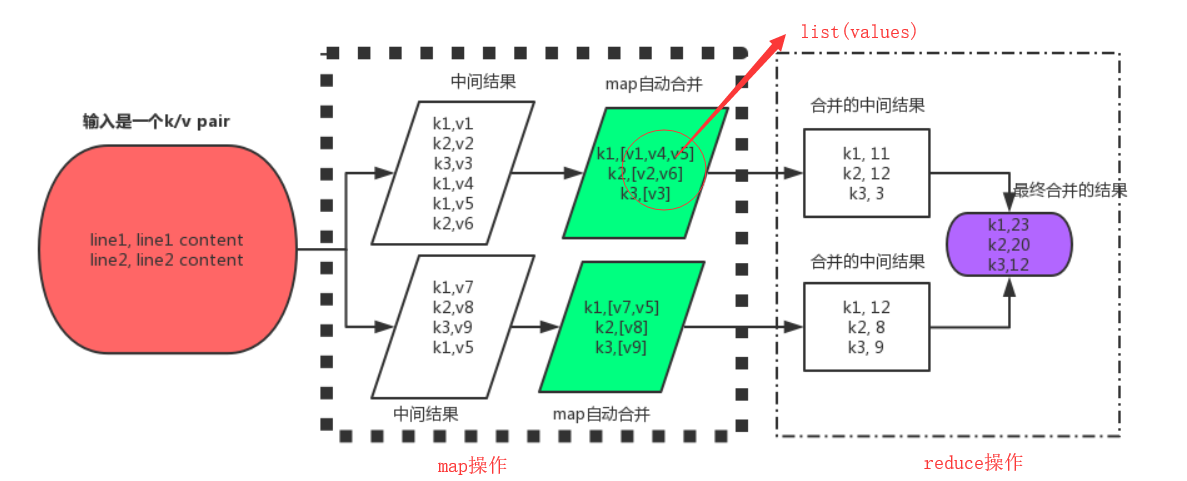


在这个例子中，map函数检查每一个单词，并且对每一个单词增加1到其对应的计数器，Reduce函数把特定单词的所有出现的次数进行合并。

用户提供的map和reduce函数有如下相关类型：



也就是说，输入的键和值和输出的键值是属于不同区域的，进一步说，中间的键值是和输出的键值属于相同区域的。（比如map的输出就是作为reduce的输入）



**3、实现（Implementation）**

MapReduce接口可以有很多不同的实现，根据不同的环境来进行选择。比如说，一个可以适用于小型共享内存机器，另一个可能是基于大型NUMA多处理器系统，还可能有为大规模计算机集群的实现。

Google广泛使用的计算环境：用交换机网络连接的，由普通PC构成的超大集群

环境：

1. 每个节点通常是双x86处理器，运行Linux，每台机器有2-4GB内存。
2. 使用的网络设备都是常用的，一般在节点上使用的是带宽为100M或者1GB的网络，一般情况下都用不到一般的网络带宽。
3. 一个cluster中常常有成百上千台机器，所以机器故障是常态而不是异常。
4. 存储时使用便宜的IDE硬盘，直接放在每一个机器上，并且有一个分布式的文件系统来管理这些分布在各个机器上的硬盘。文件系统通过复制的方式在不可靠的硬件上保证可用性和可靠性。
5. 用户向调度系统提交请求时，每一个请求都包含一组任务，映射到这个计算机cluster里的一组机器上执行。

**执行概览（Execution Overview）：**

Map操作通过把输入数据进行分区（partition）（比如分为M块），就可以分布到不同的机器上执行了。把输入块拆成多块，可以并行在不同机器上执行。Reduce操作是通过对中间产生的key的分布来进行分布的，中间产生的key可以根据某种分区函数进行分布（比如 hash(key) mod R），分布成为R块，分区(R)的数量和分区函数都是由用户指定。

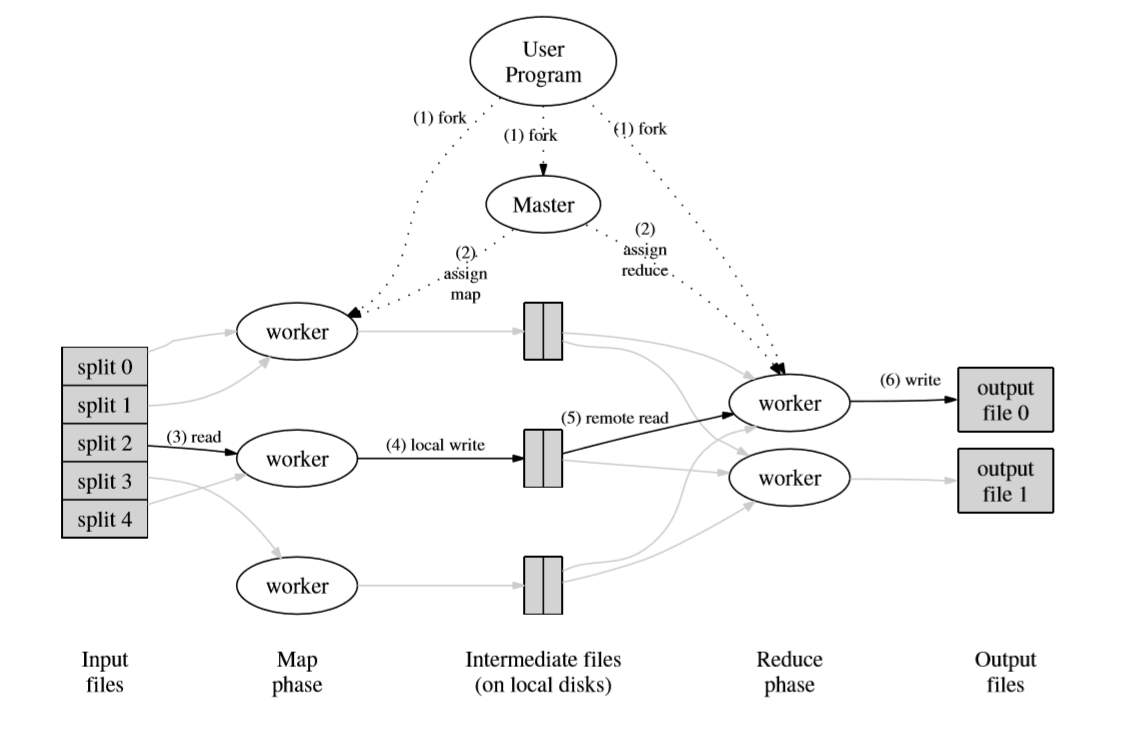


图1是我们实现的MapReduce操作的整体数据流，当用户程序调用MapReduce函数，就会引起图中的各项操作：

1. MR自动地将输入文件分成M个分片，每个分片16-64MB大小（可以进行调节），然后启动每台机器的用户程序。
2. 其中一台机器的程序比较特殊，它就是master，master将其他机器指定为worker，需要分配M个map任务和R个reduce任务（一般M是R的几十倍，比如40倍），master给闲置的worker分配map任务或者reduce任务，也就是说每台worker要么干map的活儿要么干reduce的活儿。
3. 收到map任务的worker读相应的输入分片（input split），从输入文件中读取出一个个键值对传给用户定义的map函数，map函数将一个键值对变成很多个键值对，将这些中间结果缓存在内存中。
4. 缓存的键值对会定时地写入到本地磁盘中，键值对会被分区函数（一般是hash(key) mod R）分成R个region，这些缓存在本地磁盘的键值对会回传到master，master负责将这些键值对的位置信息传给reduce worker。
5. Master将位置信息给reduce worker，reduce worker通过RPC去读map worker本地磁盘缓存的数据，读完所有的中间数据后，reduce会将中间数据的key进行排序以便于相同的key可以分到同一组中，排序是很有必要的，因为许多不同的key会映射到同一个reduce任务里，如果中间数据太多以至于reduce worker没有足够的内存进行存放时，就需要外部排序了（数据倒换到磁盘来排序，速度非常慢）。
6. reduce worker遍历每一个排序后的中间数据，将key和相应的values传递给用户的reduce函数，reduce函数将结果添加到这个reduce分片的输出文件里。
7. 当所有的map任务和reduce任务都已经完成的时候，master激活用户程序，在这时候MapReduce返回用户程序的调用点。

从4，5两步可以看出MR慢就慢在将中间结果放在了本地磁盘里，然后把位置信息发送给master，最后reduce worker收到位置信息后直接去读map worker的本地磁盘，写磁盘和读磁盘其实是很慢的。

**Master的数据结构**

Master需要一定的数据结构来保存map和reduce任务的状态（idle，in-progress或者completed），并且识别不同的worker机器（对于非idle的任务状态来说）。

Master是一个由map产生的中间区域文件位置信息到reduce任务的一个管道，因此，对于每一个能够完成的map任务，master都会保存这个map任务产生的R中间区域文件的位置和大小信息。对于保存文件的位置和大小信息是当接收到map任务能够完成时做的，这些信息是渐增推送到处于in-progress状态的reduce任务的worker上。

**容错考虑**

**Worker Failure：**

Master会定期地向每一个worker机器进行ping操作。如果在一定时间内没有worker机器的回应，master就认为这个worker失效了。所以这台worker完成的map任务都被设置成为它们的初始idle状态，并且因此可以被其他worker调度执行。类似的，所有这台机器上正在处理的map任务或者reduce任务都被设置为idle状态，可以被其他worker重新执行。

在失效机器上的已经完成的map任务还需要再次重新执行，这是因为中间结果存放在这个失效的机器上，所以导致中间结果无法访问。已经完成的reduce任务不需要再次执行，因为它们的结果已经保存在全局的文件系统中。

可以看出map任务失败时代价是很大的，所以尽量不要在map任务执行的时候掉链子。MR与GFS的关系也很明显，就是map的中间结果存放在本地文件系统里，而reduce的结果存放在GFS文件系统里。

当map任务首先由worker A 执行，随后被worker B 执行的时候（因为A失效，B开始执行），所有执行reduce任务的worker都会被通知，所有还没有来得及从A上读取数据的worker都会从B上读取数据。

这里注意一个问题：对于同一个输入，每次运算的结果都是一样的，那么称这个算子是确定的，否则就是不确定的。比如A失败了，由B顶上，如果A的算子（map函数）是确定的，那么在B机器上重新执行map操作肯定会产生同样的结果，然后把这个中间结果传递给后续的reduce操作；但是如果A的算子是不确定的，在B机器上重新执行map操作就可能产生不同的结果，比如上一次map在A上输出的是5，这次在B上输出了6.

一般情况下，避免这种问题有两种方式：

1. 把map函数定义成确定的函数
2. 保证机器A绝不会出现问题

MapReduce可以有效地支持到很大尺度的worker失效的情况，比如在一个MapReduce操作中，在一个网络例行维护中，可能会导致每次大约有80台机器在几分钟内不能访问。MapReduce的master简单地把这些不能访问的worker上的工作再执行一次，并且继续调度进程，最后完成MapReduce的操作。

**Master失效**

在master中，定期会设定checkpoint，写出master的数据结构，如果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。不过，由于只有一个master在运行，所以他如果失效就比较麻烦，因此我们当前的实现上，如果master失效了，就终止MapReduce执行。客户端可以检测这种失效并且如果需要就重新尝试MapReduce操作。

**失效的处理设计**

当用户提供的map和reduce函数对于它们的输入来说是确定性的函数，我们的分布式的输出就应当和没有失败连续执行的程序相同。

我们依靠对map和reduce任务的而输出进行原子提交来完成这样的可靠性，每一个in-progress任务把输出写到一个私有的临时文件中，reduce任务产生一个这样的文件，map任务产生R个这样的任务（每一个任务对应一个reduce任务）。当一个map任务完成时，worker发送一个消息给master，并且这个消息中包含了这个R临时文件的名字，如果master又收到一个已经完成的map任务的完成消息，他就忽略了这个消息，否则他就在master数据结构中记录这个R文件。

当一个reduce任务完成的时候，reduce worker自动把临时输出的文件名改为正式的输出文件，如果在多台机器上有相同的reduce任务执行，那么就会有多个针对最终输出文件的更名操作。我们依靠文件系统提供的原子操作‘rename’，来保证最终的文件系统状态中记录的是其中一个reduce任务的输出。

我们的绝大部分map和reduce操作都是确定性的，实际上在语义角度，这个map和reduce并发执行和顺序执行是一样的，这就使得程序员很容易推测程序的行为。当map和reduce操作是非确定性的时候，我们有稍弱的但是依旧是有道理的错误处理机制，对于非确定性操作来说，特定reduce任务R1的输出，与非确定性的顺序执行的程序对R1的输出是等价的。另外，另一个reduce任务R2的输出，是和另一个顺序执行的非确定性程序对应的R2输出相关的。

考虑map任务M和reduce任务R1，R2，我们设定e(Ri)为已经提交的Ri执行（有且只有一个这样的执行），当e(R1)处理得是M的依次执行，而e(R2)是处理M的另一次执行的时候没那么就会导致稍弱的失效处理了。

**存储位置**

在我们的环境下，网络带宽资源是相对缺乏的，我们尽量将输入的数据（input data）保存在构成集群的本地硬盘上（通过GFS进行管理），通过这种方式来减少网络带宽的开销。GFS把文件分成64M一块，并且每一块都有几个拷贝（通常是3份），分布到不同的机器上，MapReduce的master有输入文件组的位置信息，并且尝试分派map任务在对应包含了相关输入数据块的设备上执行。如果不能分配map任务到对应其输入数据的机器上执行，他就会尝试分配map任务到靠近这个任务输入数据库的机器上执行（比如，分配到一个包含输入数据块的一个switch网段的worker上执行）。当在一个足够大的cluster上运行大型的MapReduce操作的时候，大部分输入数据都是在本地机器读取的，它们消耗比较少的网络带宽。

**任务粒度**

把map阶段拆分到M小块，reduce阶段拆分到R小块执行，在理想状态下，M和R应当比worker机器数量要多得多。每一个worker机器都通过执行大量的任务来提高动态的负载均衡能力，并且能够加快故障恢复的速度：这个失效机器上执行的大量map任务都可以分布到所有其他worker机器上执行。

但是我们在实现的过程中，实际上对于M和R的取值有一定的限制，因为master必须执行O(M+R)次调度，并且在内存中保存O(M\*R)个状态。（对影响内存使用的因素还是比较小的：O(M\*R)块状态，大概每对map/reduce任务一个字节就可以的）

进一步来讲，用户通常会指定R的值，因为每一个reduce任务最终都是一个独立的输出文件，在实际中，我们倾向于调整M的值，使得每一个独立任务都是处理大约16M到64M的输入数据。另外，我么们使R比较小，这样使得R占用不多的worker机器，我们通常按照这样的比例来执行MapReduce：M=200,000，R=5,000，使用2,000台worker机器。

**备用任务**

通常情况下，一个MapReduce的总执行时间会收到最后的几个“拖后腿”任务的影响：在计算过程中会有一个机器过了比正常时间长得多得时间还没有执行完map或者reduce任务，导致MapReduce总任务不能按时完成。出现拖后腿得情况有很多原因，比如说一个机器得硬盘有点问题，经常需要反复读取纠错，然后把读取输入数的性能从30M/s降低到1M/s。cluster调度系统已经在某台机器上调度了其他的任务，所以因为CPU/内存/本地硬盘/网络带宽等竞争的关系，导致执行MapReduce的代码性能比较慢。我们最近出现的一个问题是机器的启动代码有问题，导致关闭了CPU的cache：在这些机器上的任务性能有上百倍的影响。

我们有一个通用的机制来减少拖后腿的情况，当MapReduce操作接近完成的时候，master调度备用进程来执行那些剩下的in-progress状态的任务。无论当最初的任务还是backup任务执行完成的时候，都把这个任务标记成为已经完成，我们调优了这个机制，通常只会占用几个百分点的机器资源。但是我们发现这样做以后对于减少超大MapReduce操作的总处理时间来说非常有效。

**4、技巧（Refinements）**

**分区函数**

MapReduce的使用者通过指定（R）来给出reduce任务/输出文件的数量，他们处理的数据在这些任务上通过对中间结果key的分区函数来进行分区，缺省分区函数时使用hash函数（hash（key）mod R）。这一般就可以得到分散均匀的分区，不过在某些情况下，对key用其他的函数进行分区可以更有用，比如某些情况下key时URL，那么我们希望所有对单个host的入口URL都保存在相同的输出文件。为了支持类似的情况，MapReduce函数库可以让用户提供一个特定的分区函数，比如使用hash(hostname(URL key))mod R作为分区函数，这样可以让只想同一个hostname的URL分配到相同的输出文件中。

**顺序保证**

我们确保在给定的分区中，中间键值对 key/value的处理顺序是根据key增量处理的。这样的顺序保证可以很容易生成每一个分区有序的输出文件，这对于输出文件格式需要支持客户端的对key的随机存储时就很有用，或者对输出数据集再作排序就很容易。

**合并函数（Combiner Function）**

在某些情况下，允许中间结果key重复占据相当的比重，并且用户定义的reduce函数满足结合率和交换率。Combiner函数在每一个map任务的机器上执行，通常这个combiner函数的代码和reduce的代码实现上是一样的，它们之间唯一的不同就是MapReduce对于这两个函数的输出处理上不同，对于reduce函数的输出是直接写到最终的输出文件，对于combiner函数来说输出是写到中间文件，并且会被发送到reduce任务中去。

**输入和输出类型**

MapReduce函数库提供了读取几种不同格式的输入支持。例如，“text”模式下，每行输入都被看成一个key-value对：key是在文件的偏移量，value是行的内容。另一个格式保存了根据key进行排序key-value对的顺序。每一个输入类型的实现都知道如何把输入进行分隔（为了得到map任务），比如text模式下的分隔就是要确保分隔得边界只能按照行来进行分隔。用户可以通过简单得提供reader接口来进行新得输入类型的支持，不过大部分用户都只用一小部分预先定义的输入类型。

Reader函数不需要提供从文件读取数据，例如我们很容易定义一个reader函数从数据库提取数据，或者从保存在内存中的数据结构中读取数据。

类似的，我们提供了一组用于输出的类型，可以产生不同格式的数据，并且用户也可以很简单地增加新的数据类型。

**边缘效应**

在某些情况下，MapReduce的使用上，如果在map操作或者reduce操作时，增加辅助的输出文件会比较游泳，我们依靠程序来提供这样的边缘原子操作。通常应用程序写一个临时文件并且用系统的原子操作：改名操作，在这个文件写完的时候一次把文件名改掉。

对于单个任务产生的多个输出文件来说，我们没有提供其上的两阶段提交的原子操作支持。因此，对于产生的多个输出文件和跨文件有一致性要求的任务，都必须是确定性的任务。

**跳过不良记录**

某些情况下，用户代码中存在的一些错误（bugs）会导致在确定某些记录时使map和reduce功能崩溃，这种情况下MapReduce的操作就不能完成。一般的做法就是对错误（bugs）进行修复之后再执行，但是有时候不太可行，因为有些bug使再第三方包中，它的源代码不存在等等情况，并且很多时候忽略一些记录不进行处理也是可以接收的，比如在一个大数据集上进行统计分析的时候，就可以忽略有问题的少量记录。我们提供了一种执行模式，在这种模式下，MapReduce会检测到那些导致确定性崩溃的记录，并且跳过这些记录不做处理，使得整个程序可以继续执行。

每一个worker处理进程都有一个signal handler，可以捕获内存段异常和总线错误，在执行用户map或者reduce操作之前，MapReduce函数库通过全局变量保存记录序号，如果用户代码产生了这个信号，signal handler就会用“最后一口气”通过UDP包向master发送上次处理的最后一条记录的序号。当master看到在这个特定记录上，有不止一个失效的时候，它就标志着那条记录需要被跳过，并且在下次重新执行相关的map和reduce任务时跳过这条记录。

**本地执行**

因为实际执行操作是在分布式系统上执行的，通常是在好几千台计算机上执行，并且是由master机器进行动态调度任务，所以对map和reduce函数的调试就比较麻烦。为了能够方便地调试、配置和小规模的测试，我们开发了一套MapReduce的本地实现，也就是说，MapReduce函数库在本地机器上顺利执行所有的MapReduce操作，用户可以控制执行，这样计算可以限制到特定的map任务上，用户可以通过设定特别的标志来执行他们的程序，同时也可以很容易的使用调试和测试工具。（比如gdb）

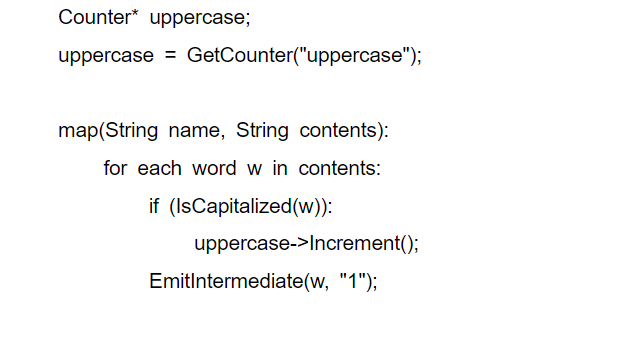
**状态信息**

Master内部有一个用于输出状态报告的HTTP服务器。状态页提供了计算的进度报告：1）有多少任务已经完成；2）有多少任务正在处理；3）输入的字节数；4）中间数据的字节数；5）处理百分比；等等。这些页面也包括了指向每个任务输出的标准错误和输出的标准文件的连接，用户可以根据这些数据来预测计算大约需要多长的时间，是否需要为这个计算机增加额外的计算资源。这些页面还可以用来分析为何计算执行的会比预期的慢。

此外，最上层的状态页面也显示了哪些worker失效了，以及他们失效的时候上面运行的map和reduce任务。这些信息对于调试用户代码中的bug很有帮助。

**计数器（Counters）**

MapReduce函数库提供了用于统计不同时间发生次数的计数器，用户创建counter对象并在map和reduce函数中适当的时候增加counter的值实现计数功能。



这些counter的值，会定时从各个单独的worker机器上传递给master（通过ping的应答包传递）。Master把执行成功的map或者reduce任务中的counter进行累计，并且当MapReduce操作完成之后，返回给用户代码。当前的counter值也会显示在master的状态页面上，这样人们可以看到计算现场的进度。当累计counter值得时候，master会检查是否有对同一个map或者reduce任务得相同累计，避免重复累计（backup任务或者机器失效导致的重新执行map任务或者reduce任务或导致这个counter重复执行，所以需要检查，避免master重复统计）

部分计数器的值是由MapReduce函数库进行自动维持的，比如已经处理的输入的key-value对的数量，或者输出的key-value键值对等等。

**性能（**P**erformance）**

测试：在一个大型集群上运行两个计算来衡量MapReduce的性能。

1. 在一个大概1TB的数据中查找特定的匹配串——>数据抽取（extract data）
2. 对大概1TB的数据进行排序——>数据清洗（shuffles data）

**集群配置（Cluster Conﬁguration）**

集群数量：1800台机器 处理器：2个2GB Inter Xeon（支持超线程）

内存：4GB 硬盘：160GB IDE硬盘 网卡：1000MB

这些机器部署在一个由两层组成的树形交换机网络中，在最上层大概有100-200G的聚合贷款，所有这些机器都有相同的部署（对等部署），因此任意两点之间的来回时间小于1毫秒。在4GB的内存里，大概有1-1.5G用于运行在集群上的其他任务。

**Grep（数据抽取/挖掘）**

Grep程序需要扫描大概10的10次方个由100字节组成的记录，查找比较少见的3个字符的查找串（这个查找串在92337个记录中存在）。输入的记录被拆分成大约64MB一个块（M=15000），整个输出方在一个文件中（R=1）.

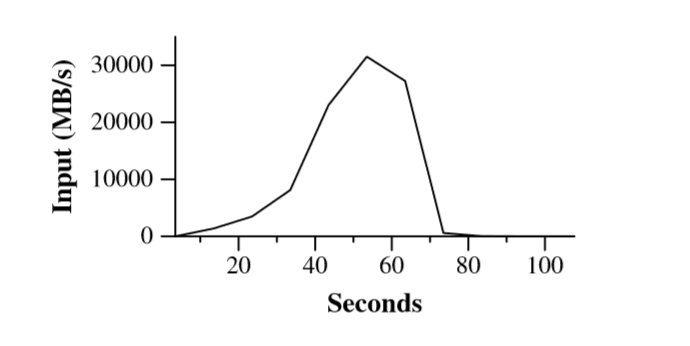


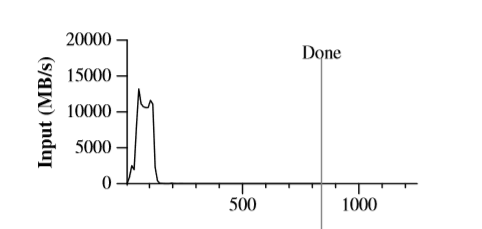
图2表示了这个程序随时间的处理过程，Y轴是输入数据的处理速度，处理速度逐渐随着参与MapReduce计算机器的增加而增加，当1764台worker开始工作的时候，达到了30GB/s的速度，当map任务结束的时候，在计算开始后80s，输入的速度降至0，整个计算过程从开始到结束一共花了大概150s，这包括了大约一分钟的启动部分。（启动部分是用来把整个程序传播到各个worker机器上的时间，并且等待GFS系统打开100个输入文件集合获得相关文件位置优化信息）

**排序Sort（数据清洗/分析） 高效的backup任务 失效的机器**

Sort程序排序10的10次方个100字节组成的记录（大概1TB的数据），它由不到50行的用户代码组成，三行的map函数从文本中解出10个字节的排序key，并且把这个key和原始行作为中间结果key-value对输出。我们使用一个内嵌的identity函数作为reduce的操作，这个函数把中间结果key-value对作为输出的key-value对，最终排序输出写到一个两路复制的GFS文件中（也就是说，程序的输出会写2TB的数据）

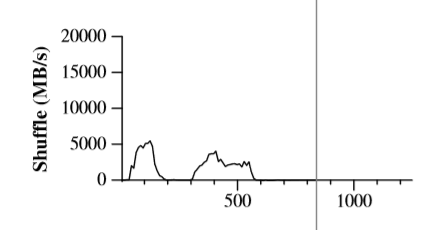
把输入数据分成64MB每块（M=15000），把排序后的输出分成4000个文件（R=4000），分区函数使用key的原始字节来把数据分区到R个小块中。

Benchmark中的分区函数自身知道key的分区情况，通常对于排序程序来说，我们会增加一个预处理的MapReduce操作，这个操作用于采样key的情况，并且用这个采用的key的分布情况来计算对最终排序处理的分区点。



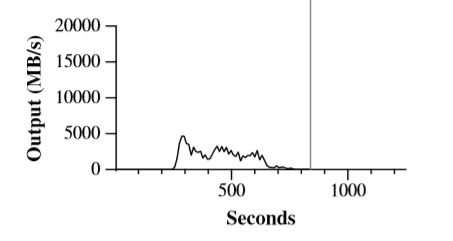
Input data

上图表示了输出数据读取的速度，数据读取速度峰值大概是13GB/s，并且在所有map任务完成之后（不到200s）读取速度迅速滑落到0。我们可以注意到数据读取速度小于grep粒子，这是因为排序map任务花了大概一半的时间和I/O带宽讲中间输出写到本地硬盘中，相对应的grep中间结果输出几乎可以忽略不记。



Shuffle data

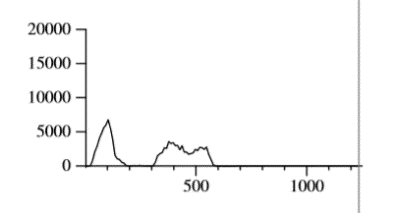
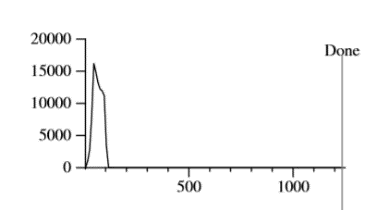
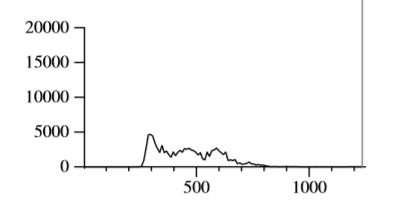
上图描述了map任务把中间数据发送到reduce任务的网络带宽速度与时间之间的关系。这个排序过程从第一个任务完成后就开始了，图中的第一个峰值时启动了第一批大概1700个reduce任务（整个MapReduce分布到1700台机器上，每台机器执行1个reduce任务）。大概计算开始300s后这些第一批reduce任务完成，之后开始执行剩下的reduce任务，所有这些排序任务会在计算开始后大概600s结束。



Output data

上图描述了reduce任务把排序后的数据写到最终的输出文件的速度与时间的关系。在第一个排序期结束后到写盘开始之前有一个小延时，这是因为机器正在忙于内部排序中间的数据，写盘的速度持续大概2-4GB/s，在计算开始后大概850s左右写盘完成。包括启动部分，整个计算用了891s。

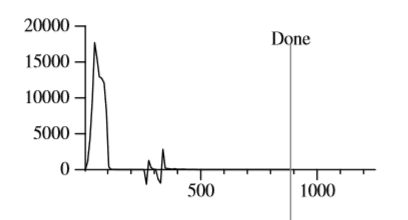
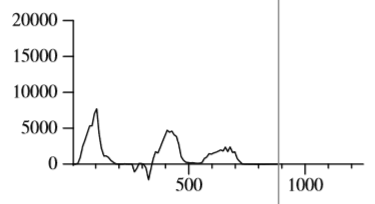
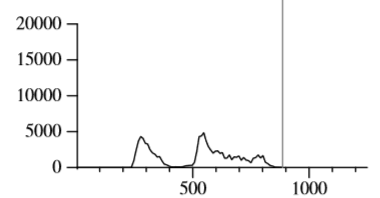
**高效的backup任务**

Input data shuffle data output data

如上所示，当关闭所有的backup任务之后，sort程序的执行情况会发生改变。执行流和Normal execution很类似，但是执行结束的时间被大大延长了，并且执行的尾巴没有任何写盘的动作。在960s以后，除了5个reduce任务意外，其他的reduce任务都已经完成，不过这些拖后腿（straggler）的任务又执行了300s才完成，整个计算花了1283s，超出了44%的执行时间。

**失效的机器**

Input data shuffle data output data

在sort程序执行过程中故意杀死1746个worker中的200个worker进程的执行情况，底层的集群调度立刻在这些机器上重新创建了新的worker处理（因为我们只是把这些机器上的处理进程杀掉，而机器依旧是可以操作的）。

因为已经完成的map work丢失了（由于相关map worker被杀掉了），需要重新再作，所以worker死掉会导致一个附属的输入速率。相关map任务的重新执行很快重新执行了，整个计算过程再933s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（Experience）**

MapReduce函数库在Google内部各个领域内的广泛应用：

1. 大型机器学习问题
2. Google News 和 Froogle产品的集群问题
3. 提取用于生成热门查询报告的数据（例如：GoogleZeitgeist）
4. 提取新实验和产品的网页属性（例：从大量网页中提取地理位置以进行本地化搜索）
5. 大型图形计算

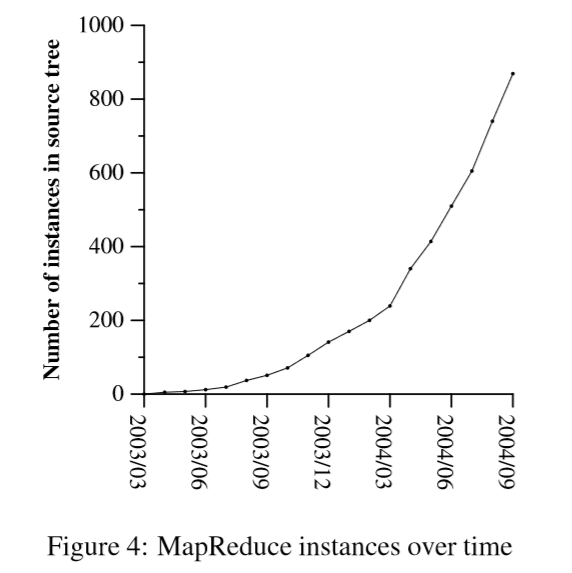
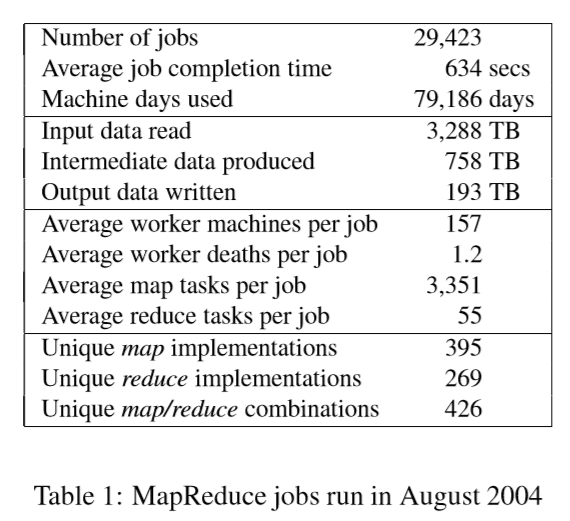
 

图4显示了随着时间的推移，在原始源代码管理系统中检查的单独MapReduce程序数量的显着增长，从2003年初的0增长到2004年9月下旬的近900个独立实例。MapReduce之所有这样成功是因为它能够在不到半小时的时间里写出一个简单而又能应用于上千台机器的大规模并发程序，并且极大地提高了开发和原形设计的周期效率。并且MapReduce可以让一个完全没有分布式或并行系统经验的程序员很容易地开发出能够处理海量数据的程序。

在每一个任务结束的时候，MapReduce函数库记录使用的计算资源状态如表1所示，表1就是2004年8月MapReduce运行的任务所占用的相关资源。

**大型索引**

到目前位置，最成功的MapReduce应用就是重写了Google web搜索服务所使用到的index系统，索引系统处理蠕虫系统（crawling system）抓取的超大量数据，这些数据保存在GFS文件里。这些文件的大小超过了20TB的数据，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的好处：

1. 索引代码更简单，更小，更容易理解，因为对于容错的处理代码，分布以及并行处理代码都通过MapReduce函数库进行封装。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++代码一下减少到了大概700行代码。
2. MapReduce的函数库的性能非常好，所以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处理次数，这使得我们能够容易地改变索引处理方式。比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，但在新系统内，只需要花几天时间就可以了。
3. 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，以及网络风暴等问题都由MapReduce自己解决了，而不需要操作人员的交互。此外，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。