MapReduce

1. 摘要。

MapReduce 是一个编程模式，它是与处理 / 产生海量数据集的实现相关。用户指定一个 map 函数，通过这个 map 函数处理 key/value （键 / 值）对，并且产生一系列的中间 key/value 对，并且使用 reduce 函数来合并所有的具有相同 key 值的中间键值对中的值部分。使用这样的函数形式实现的程序可以自动分布到一个由普通机器组成的超大几群上并发执行。 run-time 系统会解决输入数据的分布细节，跨越机器集群的程序执行调度，处理机器的失效，并且管理机器之间的通讯请求。这样的模式允许程序员可以不需要有什么并发处理或者分布式系统的经验，就可以处理超大的分布式系统得资源。

**二、介绍。**

google 的创造者和其他人实现了上百个用于特别计算目的的程序来出来海量的原始数据，比如蠕虫文档， web 请求 log ，等等，用于计算出不同的数据，比如降序索引，不同的图示展示的 web 文档，蠕虫采集的每个 host 的 page 数量摘要，给定日期内最常用的查询等等。绝大部分计算都是概念上很简洁的。不过，输入的数据通常是非常巨大的，并且为了能在合理时间内执行完毕，其上的计算必须分布到上百个或者上千个计算机上去执行。如何并发计算，如何分布数据，如何处理失败等等相关问题合并在一起就会导致原本简单的计算掩埋在为了解决这些问题而引入的很复杂的代码中。

因为这种复杂度，我们设计了一种新的东西来让我们能够方便处理这样的简单计算。这些简单计算原本很简单，但是由于考虑到并发处理细节，容错细节，以及数据分布细节，负载均衡等等细节问题，而导致代码非常复杂。所以我们抽象这些公共的细节到一个 lib 中。这种抽象是源自 Lisp 以及其他很多面向功能的语言的 map 和 reduce 概念。我们认识到大部分操作都和 map 操作相关，这些 map 操作都是运算在输入记录的每个逻辑 ”record” 上，并且 map 操作为了产生一组中间的 key/value 键值对 , 并且接着在所有相同 key 的中间结果上执行 reduce 操作，这样就可以合并适当的数据。我们得函数模式是使用用户定义的 map 和 reduce 操作，这样可以让我们并发执行大规模的运算，并且使用重新执行的方式作为容错的优先机制。

MapReduce 的主要贡献在于提供了一个简单强大的接口，通过这个接口，可以把大尺度的计算自动的并发和分布执行。使用这个接口，可以通过普通 PC 的巨大集群，来达到极高的性能。

第二节讲述了基本的编程模式，并且给出了一些例子。第三节讲述了一个面向我们基于集群的计算环境的 MapReduce 的实现。第四节讲述了一些我们建议的精巧编程模式。第五节讲述了在不同任务下我们的 MapReduce 实现的性能比较。第六节讲述了在 Google 中的 MapReduce 应用以及尝试重写了我们产品的索引系统。第七节讲述了相关工作和未来的工作。

# 2 编程模式

运算处理一组输入的（ input ）键值对（ key/valuepairs ） , 并且产生一组输出的（ output ）键值对。 MapReduce 函数库德用户用两个函数来表达这样的计算： Map 和 Reduce 。

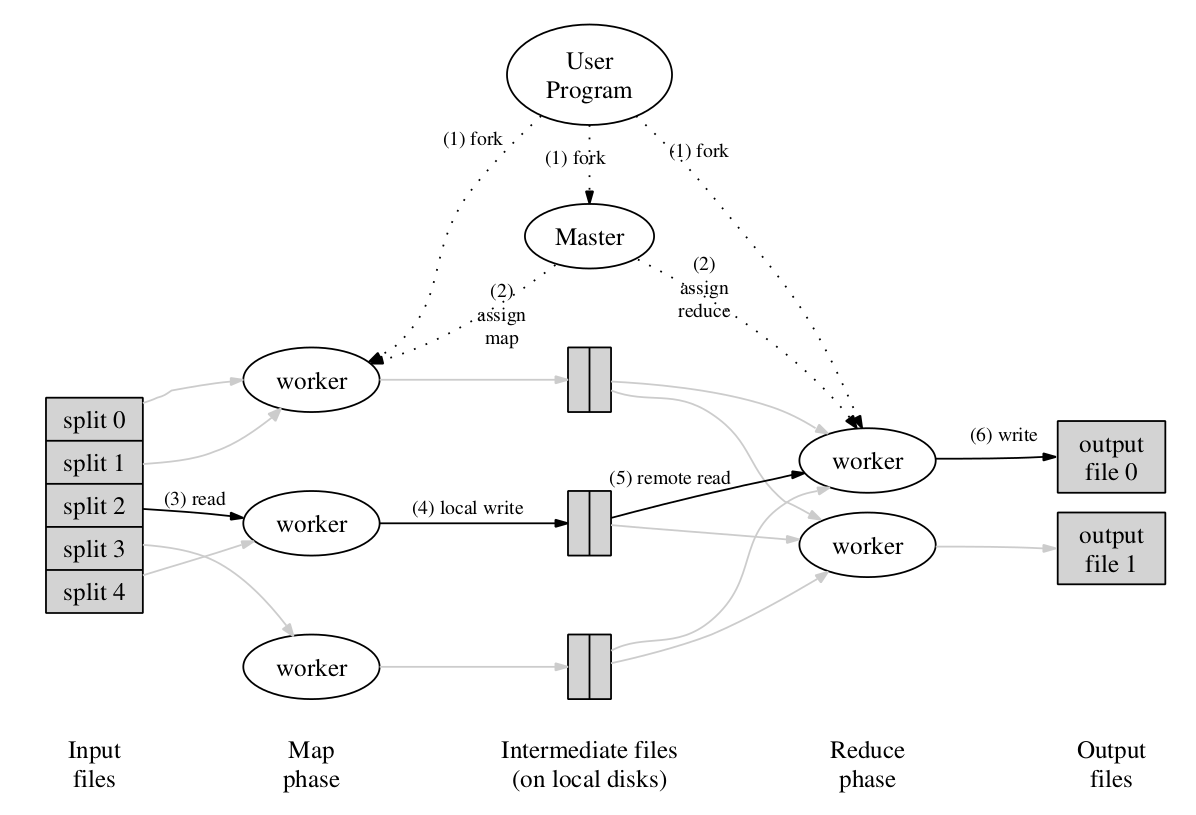
Map 函数，是用户自定义的的函数，处理输入的键值对，并且产生一组中间的（ intermediate ）键值对。 MapReduce 函数库稽核所有相同的中间键值键 I 的值 ，并且发送给 Reduce 函数进行处理。

Reduce 函数同样也是用户提供的，它处理中间键值 I ，以及这个中间键值相关的值集合。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 函数的中间值是通过一个 iterator 来提供的。这就让我们可以处理超过内存容量的值列表。

**三、执行概括。**

*MapReduce* 模型可以有多种不同的实现方式，论文主要介绍了一种在 *Google* 内部广泛使用的计算环境下（通过以太网交换机连接，并由商用服务器所组成的大型集群）使用的 *MapReduce* 实现 。

### 3.1 执行流程



上图为此 *MapReduce* 框架实现的示意图，下文基于此图对 *MapReduce* 的执行过程进行描述，描述的序号与图中的序号相对应

1. *MapReduce* 库会先把文件切分成 M 个片段（ 每个大小为 *16MB~64MB* ），存储在 GFS 文件系统 ，接着，它会在集群中启动多个 程序副本 。
2. 这些程序副本中，一个为 master ，剩余为 worker ，*master* 对 *worker* 进行任务分配，共有 M个 *map* 任务以及 R 个 *reduce* 任务（ *M* 同时为文件片段数 ， *R* 由用户指定），*master* 会给每个空闲的 *worker* 分配一个 *map* 任务或者一个 *reduce* 任务 。
3. 被分配了 *map* 任务的 *worker* 会读取相关的输入数据片段，这些数据片段一般位于该 *worker* 所在的服务器上（ *master* 调度时会优先使 *map* 任务执行在存储有相关输入数据的服务器上，通过这种 本地执行 的方式降低服务器间网络通信，节约网络带宽 ）。它会解析出输入数据中的 键值对 ，并将它们传入用户定义的 *Map* 函数中，*Map* 函数所生成的 中间键值对 会被缓存在内存中 。（ 要将 *map* 任务和用户定义的 *Map* 函数区分开来，*map* 任务包含了一些前置处理以及 *Map* 函数的执行 ，*reduce* 任务和 *Reduce* 函数同理 ）
4. 每隔一段时间，被缓存的中间键值对会被写入到本地硬盘，并通过分区函数（一般是哈希后取模）分到 *R*个区域内 。这些被缓存的键值对在本地硬盘的位置会被传回 *master* ，*master* 负责将这些位置转发给执行 *reduce* 任务的 *worker* 。
5. 所有 *map* 任务执行结束后，*master* 才开始分发 *reduce* 任务 。当某个执行 *reduce* 任务的 *worker* 从 *master* 获取到了这些位置信息，该 *worker* 就会通过 RPC 的方式从保存了对应缓存中间数据的 *map workers* 的本地硬盘中读取数据 （ 输入一个 *reduce* 任务中的中间数据会产生自所有 *map* 任务 ）。当一个 *reduce worker* 读完所有中间数据后，会 根据中间键进行排序，使得具有相同中间键的数据可以聚合在一起 。（需要排序是因为中间 *key* 的数量一般远大于 *R* ，许多不同 *key* 会映射到同一个 *reduce* 任务中 ）如果中间数据的数据量太大而无法放到内存中，需要使用外部排序 。
6. *reduce worker* 会对排序后的中间数据进行遍历，对于每个唯一的中间键，将该中间键和对应的中间值的集合传入用户提供的 *Reduce* 函数中，*Reduce* 函数生成的输出会被追加到这个 *reduce* 任务分区的输出文件中 （ 即一个 *reduce* 任务对应一个输出文件，即 *R* 个输出文件，存储在 *GFS* 文件系统，需要的话可作为另一个 *MapReduce* 调用的输入 ）。
7. 当所有的 *map* 和 *reduce* 任务完成后，*master* 会唤醒用户程序 。此时，用户程序会结束对 *MapReduce*的调用 。

### 3.2 容错

#### 3.2.1 Woker 故障

master 会周期性地 ping 每个 worker ，若在一定时间内无法收到某个 worker 的响应，那么 master 将该 worker标记为 fail ：

* 此 worker 上 **完成** 的所有 map 任务都被重设为 idle 状态，交由别的 worker 去执行这些 map 任务
* 此 worker 上 **正在执行** 的 map 任务或 reduce 任务重设为 idle 状态，并等待重新调度

该 worker 上完成的 map 任务必须重新执行，因为 map 任务数据结果保存在 worker 的本地硬盘中，worker 无法访问了，则输出数据也无法访问；该 worker 上完成的 reduce 任务不需要重新执行，因为输出结果已存储在全局文件系统中 。

#### 3.2.2 Master 故障

目前的实现选择中断 MapReduce 计算，客户端可检查该 master 的状态，并根据需要重新执行 MapReduce 操作 。

### 3.3 数据存储位置

此模式是为了 **节约网络带宽**。

将输入数据（ 由 GFS 系统管理 ）存储在集群中服务器的本地硬盘上，GFS 将每个文件分割为大小为 64MB 的 Block ，并且对每个 Block 保存多个副本（通常3个副本，分散在不同机器上）。master 调度 map 任务时会考虑输入数据文件的位置信息，尽量在包含该相关输入数据的拷贝的机器上执行 map 任务 。若任务失败，master尝试在保存输入数据副本的邻近机器上执行 map 任务，以此来节约网络带宽 。

### 3.4 备用任务

此模式是为了缓解 **straggler (掉队者) 问题**，即 ：一台机器花费了异常多的时间去完成 **最后几个**  map 或 reduce 任务，导致整个计算时间延长的问题 。可能是由于硬盘问题，可能是 CPU 、内存、硬盘和网络带宽的竞争而导致的 。

解决此问题的方法是：当一个 MapReduce 计算 **接近完成** 时，master  **为正在执行中的任务执行 备用任务** ，当此任务完成时，无论是主任务还是备用任务完成的，都将此任务标记为完成 。这种方法虽然多使用了一些计算资源，但是有效降低了 MapReduce Job 的执行时间 。

### 3.5 Combiner 函数

某些情况下，每个 map 任务生成的中间 key 会有明显重复，可使用 **Combiner 函数**在 map worker 上将数据进行部分合并，再传往 reduce worker 。

Combiner 函数 和 Reduce 函数的实现代码一样，区别在于两个函数输出不同，Combiner 函数的输出被写入中间文件，Reduce 函数的输出被写入最终输出文件 。

这种方法可以提升某些类型的 MapReduce 任务的执行速度（ 如 word count 任务）。

### 3.6 临时中间文件

对于有服务器故障而可能导致的 **reduce 任务可能读到部分写入的中间文件** 的问题 。可以使用 **临时中间文件** ，即 map 任务将运算结果写入临时中间文件，一旦该文件完全生成完毕，以原子的方式对该文件重命名 。

四、MapReduce 的优点

* 适合PB级以上海量数据的离线处理
* 隐藏了并行化、容错、数据分发以及负载均衡等细节
* 允许没有分布式或并行系统经验的程序员轻松开发分布式任务程序
* 伸缩性好，使用更多的服务器可以获得更多的吞吐量

## 五、MapReduce 的限制

 不擅长实时计算

 无法进行流式计算，因为 *MapReduce* 的输入数据是静态的

 无多阶段管道，对于先后依赖的任务，*MapReduce* 必须把数据写入硬盘，再由下一个 *MapReduce* 任务调用这些数据，造成了多余的磁盘 I/O

## 六、相关问题总结

### 6.1 MapReduce 如何节约网络带宽

1. 集群中所有服务器既执行 GFS ，也执行 MapReduce 的 worker
2. master 调度时会优先使 map 任务执行在存储有相关输入数据的服务器上
3. reduce worker 直接通过 RPC 从 map worker 获取中间数据，而不是通过 GFS ，因此中间数据只需要进行一次网络传输
4. R 远小于中间 key 的数量，因此中间键值对会被划分到一个拥有很多 key 的文件中，传输更大的文件（ 相对于一个文件拥有更少的 key ）效率更高

### 6.2 MapReduce 如何获得好的负载均衡

1. 通过备用任务缓解 straggler 问题
2. 使 task 数远多于 worker 数，master 将空闲任务分给已经完成任务的 worker