读后感

何兰兰 18301039

谷歌在03到06年间连续发表了三篇很有影响力的文章，分别是03年SOSP的GFS，04年OSDI的MapReduce，和06年OSDI的BigTable。GFS为上层提供高效的非结构化存储服务，BigTable是提供结构化数据服务的分布式数据库，Hadoop MapReduce是一种并行计算的编程模型，用于作业调度。MapReduce论文大概可以分成两个部分，一个叫做MapReduce的编程模型，另一个是大规模数据处理的体系架构的实现。从文中看，解决问题只要都严格遵循Map Shuffle Reduce三个阶段就好。其中Shuffle是系统自己提供的而Map和Reduce是用户自定义的。我认为它实现的在廉价PC端实现超大规模的数据处理在当时具有十分重要的意义，GFS功不可没。GFS可以简单理解为使用一大群廉价的PC作为底层Server，对于超大数据存放，比如几百万的文件的话。我们可以分成好几个chunk server,每个chunk server可以简单的理为一个廉价PC。然后由一个Master server来做索引。首先能够知道文件在哪个chunk server上，先去那个server，然后再让chunk server从它自己的index里找东西。为了防止server crash,每个文件要复制在3个server里。判断chunk server有没有挂掉用的是心，每个server会定期给Master发信息，如果长时间没法就说明chunk server可以挂掉了。在阅读完三篇文章后，我对GFS、MapReduce、Big Table有了部分了解：

1.GFS

GFS是一个可扩展的分布式文件系统，用于大型的、分布式的、对大量数据进行访问的应用。它运行于廉价的普通硬件上，提供容错功能。

在GFS 系统中，文件以分层目录的形式组织，用路径来标识。创建新文件、删除文件、打开文件、关闭文件、读/写文件等;快照：低成本创建一个文件或目录树的拷贝；记录追加：记录追加允许多个客户端同时追加操作一个文件，并保证每个客户端的操作都是原子性；此外允许多个客户端在不需要额外同步锁的情况下，同时对一个文件追加记录。

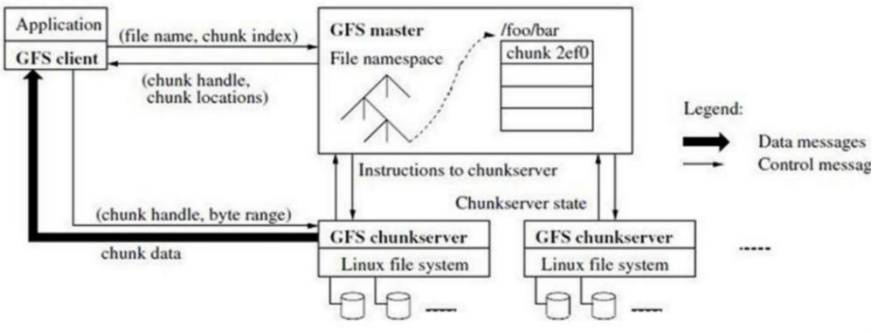


Figure 1: GFS Architecture

GFS 集群主要由三个部分构成：一个单独的 GFS Master 节点、多台 GFS Chunk（块） 服务器、多个 GFS 客户端。一个单独的 GFS Master 节点：即 Hadoop 底层存储 HDFS 的 nameNode 节点的前身。一个单独的 Master 节点主要管理所有文件系统元数据（名称空间、访问控制信息、文件和 Chunk 的映射信息、当前的 Chunk 位置信息）、管理系统范围内的活动（Chunk 租用管理、孤儿（orphaned） Chunk 回收、Chunk 在 Chunk 服务器上的迁移）、

发送心跳信息保持与每个 Chunk 服务器通讯（发送指令，接受服务器状态信息）。多台 GFS Chunk（块）服务器：即 Hadoop 底层存储 HDFS 的 dataNode 节点的前身。GFS Chunk（块）服务器即 Linux 机器，允许用户级别的服务进程，包含由 Master 服务器分配一个不变且唯一的 Chunk 标识。由于存储在 GFS 中的文件被分割为固定大小的 Chunk，以 Linux 文件的形式保存在硬盘中，根据 GFS 客户端指定的 Chunk 标识与字节范围读写数据。在可靠性方面，由于每个块都会复制到多个块服务器上（用户可设定不同的复制级别），有了冗余备份，系统的可靠性与容灾能力就有了保障。多个 GFS 客户端：当多个 GFS 客户端发送请求时，请求顺序如下：对 Master 发送文件名称与块索引 => 获取指定的 Chunk 标识与所在位置 => 客户端以文件名与 Chunk 索引作为 key 缓存这些信息 => 对最近的 Chunk 服务器发送 Chunk 标识与字节范围 => 获取块数据流。特点：无需缓存数据（Linux 文件系统自动缓存经常访问的数据到内存）。

单一 Master 节点的策略简化了设计，减少对 Master 读写，避免其成为系统瓶颈，Master 节点可以通过全局的信息精确定位 Chunk 位置及进行复制决策，另外客户端在一次请求 Master 节点时会查询多个 Chunk 信息，避免多次通讯。

Master 服务器存储 3 种主要类型的元数据，包括：文件和 Chunk 的命名空间、文件和 Chunk 的对应关系、每个 Chunk 副本的存放地点。所有的元数据都保存在 Master 服务器的内存中。前两种类型的元数据（命名空间、文件和 Chunk 的对应关系）同时也会以记录变更日志的方式记录在操作系统的系统日志文件中，日志文件存储在本地磁盘上，同时日志会被复制到其它的远程 Master 服务器上。

操作日志包含了关键的元数据变更历史记录。这对 GFS 非常重要。这不仅仅是因为操作日志是元数据唯一的持久化存储记录，它也作为判断同步操作顺序的逻辑时间基线文件和 Chunk，连同它们的版本，都由它们创建的逻辑时间唯一的、永久的标识。Master 服务器在灾难恢复时，通过重演操作日志把文件系统恢复到最近的状态。为了缩短 Master 启动的时间，我们必须使日志足够小。

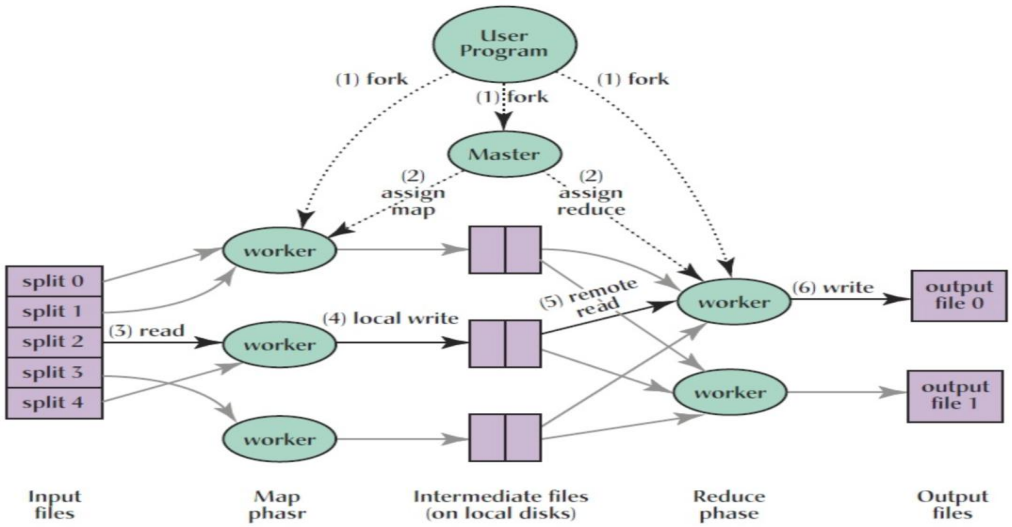
2.MapReduce

MapReduce 就是一个数据处理过程，将数据转换成相应个<Key, Value>对， 然后将相同的 Key 值的<Key, Value>划分成组，中间 Key 空间分成 R 个 pieces（Hash(Key)mod R），然后分别交给 Reduce 进行处理。

MapReduce 分为两个过程，Map 和 Reduce。Map 过程对一批 key/value 数据进行处理，中间的处理过程由用户自定义，经过 Map 过程之后会生成一批新的 key/value 的数据。Reduce 的过程是归约，将 Map 输出的结果进行处理。

map 函数和 reduce 函数是交给用户实现的，这两个函数定义了任务本身。map 函数：接受一个键值对，产生一组中间键值对。MapReduce 框架会将 map 函数产生的中间键值对里键相同的值传递给一个 reduce 函数。reduce 函数：接受一个键，以及相关的一组值，将这组值进行合并产生一组规模更小的值（通常只有一个或零个值）。Map 作业处理一个输入数据的分片，可能需要调用多次 map 函数来处理每个输入键值对；Reduce 作业处理一个分区的中间键值对，期间要对每个不同的键调用一次 reduce 函数，Reduce 作业最终也对应一个输出文件。

一切都是从最上方的 user program 开始的，user program 链接了 MapReduce 库，实现了最基本的 Map 函数和 Reduce 函数。



执行流程：

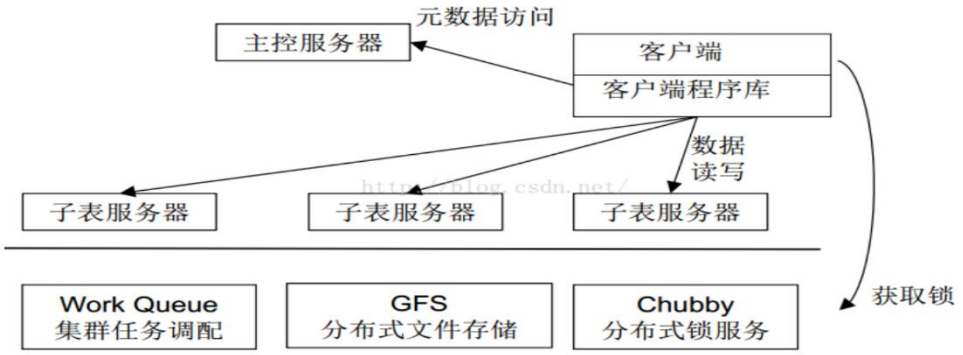
1. MapReduce 库先把 user program 的输入文件划分为 M 份（M 为用户定义），每一份通常有 16MB 到 64MB，如图左方所示分成了 split0~4；然后使用 fork 将用户进程拷贝到集群内其它机器上。
2. user program 的副本中有一个称为 master，其余称为 worker，master 是负责调度的，为空闲 worker 分配作业（Map 作业或者 Reduce 作业），worker 的数量也是可以由用户指定的。
3. 被分配了 Map 作业的 worker，开始读取对应分片的输入数据，Map 作业数量是由 M 决定的，和 split 一一对应；Map 作业从输入数据中抽取出键值对，每一个键值对都作为参数传递给 map 函数，map 函数产生的中间键值对被缓存在内存中。
4. 缓存的中间键值对会被定期写入本地磁盘，而且被分为 R 个区，R 的大小是由用户定义的，将来每个区会对应一个 Reduce 作业；这些中间键值对的位置会被通报给 master，master 负责将信息转发给 Reduce worker。
5. master 通知分配了 Reduce 作业的 worker 它负责的分区在什么位置（肯定不止一个地方，每个 Map 作业产生的中间键值对都可能映射到所有 R 个不同分区），当 Reduce worker 把所有它负责的中间键值对都读过来后，先对它们进行排序，使得相同键的键值对聚集在一起。因为不同的键可能会映射到同一个分区也就是同一个 Reduce 作业（谁让分区少呢），所以排序是必须的。
6. reduce worker 遍历排序后的中间键值对，对于每个唯一的键，都将键与关联的值传递给 reduce 函数，reduce 函数产生的输出会添加到这个分区的输出文件中。
7. 当所有的 Map 和 Reduce 作业都完成了，master 唤醒正版的 user program，MapReduce 函数调用返回 user program 的代码。

执行完毕后，MapReduce 输出放在了 R 个分区的输出文件中（分别对应一个 Reduce 作业）。用户通常并不需要合并这 R 个文件，而是将其作为输入交给另一个 MapReduce 程序处理。整个过程中，输入数据是来自底层分布式文件系统（GFS）的，中间数据是放在本地文件系统的，最终输出数据是写入底层分布式文件系统（GFS）的。

执行过程中为了检测可能的故障，master 周期性地 ping 各个 worker。如果某个 worker 响应超时，master 把 worker 标识为故障。这个 worker 处理的任何 map 操作结果需要回滚，回滚后的数据可由其他正常的 worker 进行处理。类似的，任何在故障机器上的 map 或 reduce 任务会被标识为空闲（未分配），master 重新对这些任务进行分配。因为 map 任务把处理后的数据存储在本地磁盘上，所以故障机器上的 map 任务需要重新执行。而 reduce 任务吧输出数据存储到全局文件系统，所以即时发生故障也不需重新执行。

3.BigTable

Big Table 是 Google 设计的分布式数据存储系统，用来处理海量的数据的一种非关系型的数据库。Big Table 是非关系型数据库，是一个稀疏的、分布式的、持久化存储的多维度排序 Map。Map 由 key 和 value 组成。Map 的索引是行关键字、列关键字以及时间戳。Bigtable 的设计目的是快速且可靠地处理 PB 级别的数据，并且能够部署到上千台机器上。



Big table 为客户提供了简单的数据模型，利用这个模型，客户可以动态控制数据的分布和格式，用户也可以自己推测底层存储数据的位置相关性，数据的下标是行和列的名字，名字可以是任意的字符串。

行关键字可以是任意字符串，不管这一行有多少个列，对同一行关键字的读或者写操作都是原子的。Big Table 通过行关键字的字典顺序来组织数据，基于列式存储概念，表中的每一行都可以动态分区，每个分区叫做 Tablet，Tablet 是数据分布和负载均衡的最小单位，动态分区(Tablet)好处是在读取行中很少几列时效率很高，通常只需要很少几次机器间通讯即可。字段按列簇组织在一起。Bigtable 中加入列簇的主要考虑是值的稀疏性。因为 Big Table 是按列存储的，而列的值可能是稀疏的，而且列的数目非常多，如果只按照列来组织存储的话，可能会形成很多个小文件。而分布式文件系统对小文件的管理成本是比较高的。因此 Big Table 引入列簇的概念，把同一个列簇的列存储在一起。列簇是访问权限控制的单元。时间戳是 64 位整数，在 Big Table 中用来区分数据的不同版本。时间戳可以由数据库自动生成，也可以由应用自行指定。数据的存储按照时间戳的倒序排列，因此最近的版本会被最先读到。Bigtable 支持指定数据最多有多少个版本，或者数据的生存时间。过期的数据有自动的垃圾回收机制删除。应用程序不需要维护数据的删除问题。

Big table 提供了建立和删除表以及列族的 API 函数。Big table 还提供了修改集群、

表和列族的元数据的 API。Big Table 提供按 row key 查询，按 row key 范围查询， 按列簇过滤，按时间戳过滤，以及列的迭代器。写操作包括创建记录，更新记录，删除记录。也有批量写接口（但是不保证事务性）。管理操作包括管理集群，表，列簇，权限等。服务器端代码执行支持在服务器端脚本的执行。可以进行数据的过滤，表达式转换，数据聚合等操作。

BigTable 内部数据存储文件是 Google SSTable 格式，SSTable 是持久的，排序的，不可变更的 Map。BigTable 分为三部分:链接到客户程序的库、1 个 master 服务器、多个 tablet 服务器。master 职责：给 Tablet 服务器分配 tablets, 检测新加入或者过期的 Tablet 服务器，对 Tables 服务器做负载均衡，对保存在 GFS 上的文件进行垃圾回收，建立表和列族。tablet 服务器职责:每台 tablet 服务器都管理一个 tablet 集合(十个至上千个 tablet), 否则对其内的 tablet 的读写,在 tablet 过大时进行切割。客户端程序:读写数据直接和 tablet 服务器交互,实际应用中 master 服务器负载很轻。tablet 的持久化状态信息保存在 GFS 上。更新操作提交到 REDO 日志中。最近提交的那些存放在一个排序的缓存 memtable 中，较早的更新存放在一系列 SSTable 中。

Chubby 是一个高可用的、序列化的分布式锁服务组件。一个 Chubby 服务包括了 5 个活动的副本，其中的一个副本被选为 Master 并处理请求。只有在大多数副本都是正常运行且彼此之间能够互相通信，Chubby 服务才是可用的。当有副本失效的时候，Chubby 使用 Paxos 算法保证副本的一致性。Chubby 提供了一个名字空间，包括了目录和小文件。每个目录或者文件可以当成一个锁，读写操作是原子的。Chubby 客户程序库提供对 Chubby 文件的一致性缓存。每个 Chubby 客户程序都维护一个与 Chubby 服务的会话。如果客户程序不能在租约到期的时间内重新签订会话的租约，这个会话就过期失效。当一个会话失效时，它拥有的锁和打开的文件句柄都失效了。Chubby 客户程序可以在文件和目录上注册回调函数，当文件或目录改变、或者会话过期时，回调函数会通知客户程序。

参考文档：

https://www.cnblogs.com/AndyStudy/p/9048219.html
https://blog.csdn.net/QQ635785620/article/details/33737311
https://baike.sogou.com/v56166718.htm?fromTitle=BigTable
https://blog.csdn.net/HeyShHeyou/article/details/103538680
https://blog.csdn.net/HeyShHeyou/article/details/103526563