浅谈 Google 三驾马车

江山 19301146

北京交通大学 软件学院，北京 100044

【摘　要】 Google 在 2003 至 2006 年发表了三篇论文，《MapReduce: Simplified Data Processing on Large Clusters》，《Bigtable: A Distributed Storage System for Structured Data》和《The Google File System》，介绍了 Google 如何对大规模数据进行存储和分析。这三篇论文开启了工业界的大数据时代，被称为 Google 的三驾马车。本文将根据这三篇论文浅谈我对 Google 数据存储和分析方式的理解。

【关键词】 **大数据**；MapReduce；BigTable；GFS

1　引 言

21 世纪初期，大数据时代还未到来，大多数企业需要存储的数据量并不大。但 Google 作为全世界最大的搜索服务提供商，其搜索引擎的数据来源于爬虫。随着网页数量的逐渐增加，爬虫得到的数据量也急速膨胀，单机或简单的分布式方案已经无法满足 Google 的业务需求，因此 Google 需要设计一种全新的数据存储系统，GFS[1]（Google File System）应运而生。与庞大的数据量一同而来的是耦合程度的提高，为了尽可能解耦，GFS 只进行数据的存储，而不提供类似数据库的服务。即 GFS 只存数据，但并不知道存储数据的含义，也就无法提供基于数据内容的检索服务。因此，Google 开发了 BigTable[2] 作为数据库，向上层应用程序提供基于数据内容的各项功能。此外，Google 的搜索结果依赖于 PageRank 算法的排序，该算法需要存储一些额外信息，因此他们还开发了数据处理工具 MapReduce[3]，读取 BigTable 中的内容，根据具体业务需求，对数据进行运算和操作。

总体上看，GFS 能够有效利用多个 Linux 服务器上的磁盘，并向上掩盖分布式系统的细节。BigTable 在 GFS 的基础上对数据内容进行识别和存储，并向上提供类似数据库的服务。MapReduce 使用 BigTable 中的数据进行运算，将结果提供给具体的业务使用。

报告时间: 2021-12-10
指导教师: **李在**
*学号：19301146
*E-mail：hilljiang@bjtu.edu.cn

2　GFS

2.1　简介

GFS 的设计理念上做了两个重要假设，一是 GFS 只处理大文件（TB/PB 级）。二是 GFS 不支持更新只支持添加。在这两个假设的基础上，GFS 进一步假设可以把大文件切成若干个 Chunk。Google 用 64MB 的 Chunk 来代替传统的 Block, 以更好的存储大文件，减少元数据（MetaData）的存储空间。然后以 Master + ChunkServer 的分布式架构，来将文件存储在多个机器上。

2.2　效率保证

如果 Master 的 Index 信息里记载所有 ChunkServer 对所存数据的增删查改行为，ChunkServer 每一次存储在本地硬盘的数据变化都要和 Master 通讯汇报的话，Master 的通信消耗就太大了。因此 Google 让 Master 只记录文件存储在哪些 Chunk 上，每个 Chunk 在哪个 ChunkServer 上，而具体在硬盘上的位置由 ChunkServer 本地的 Index 来记录。这样既减少了 Master 的元数据信息的大小，还减少了 Master 和 ChunkServer 的通信消耗。

此外，Master 会统计 ChunkServer 的可用空间和网络传输带宽，以及每个数据 Chunk 的访问频率。基于热点平衡原则，会优先选择空闲的 ChunkServer 来存储新的 Chunk 副本。当某个 Chunk 的副本访问频率特别高时，Master 会把这个

Chunk 的副本复制到不相邻的空闲 ChunkServer，以方便另一区域（物理距离更近）更快地访问这个热点数据，而不会拥挤。

2.3 数据一致性

Chunk server 是 GFS 的主体，它们存在的目的是为了保存各种各样的 Chunk。这些 Chunk 代表了不同文件的不同部分。为了保证文件的完整性不受到机器损坏的影响，通常来说这些 chunk 都有冗余，这个冗余标准的来说是 3 份。整个 GFS 分布式存储架构基于数据存储服务器的位置组成了一个树形结构，Google 把数据存储在不同城市，每个城市是一个树的不同分支节点，而城市里有多个存储中心，每个存储中心有多个存储机器。对于 Chunk 的 3 个副本，Google 会基于跨中心跨机架的原则，把数据存储在 2 不同分支的中心上，而同一个分支的 2 份副本被存储在不同的机架上，以此来预防一个大规模的崩溃发生时会丢失数据的所有副本。

为了保证 ChunkServer 没有丢失，每个 ChunkServer 会定期向 Master 发送“存活”的时间戳。当 Master 发现某个 ChunkServer 很久没有消息了，就会让临近的服务器发消息询问，如果没有回复，就启动这个 ChunkServer 上存储的所有数据的恢复流程。同时，Google 将每个大 Chunk 分为固定个数的小 Block，每个组成的 Block 都有一个检验和（CheckSum），而每个 Chuck 都有一个元数据记载着组成的每个 Block 检验和是多少，每次读取数据时执行对当前 Block 检验和的验证操作。当 ChunkServer 验证 Chunk 的检验和时，发现不一致，ChunkServer 会向 Master 求助，发送 Chunk 损坏的信息给 Master。Master 记载着 Chunk 的副本在哪 3 个服务器上，会把另两个 ChunkServer 发送给请求恢复损坏 Chunk 的服务器。而这个服务器会直接通信 2 个副本服务器，请求他们把自己损坏 Chunk 的副本传输过来。当 2 份副本一致时，恢复损坏的 Chunk。当 2 份 Chunk 不一致时，同时保留 2 份，等待 Master 之后的处理。

2.4 文件读写

文件读取方面，GFS 客户端会向 GFS Master 询问想要的文件名称。GFS Master 有一个 namespace 的数据结构，用树形结构记载了目标文件的路径和存储地址。GFS Master 会确认数据权限请求许可并返回数据的地址。GFS 客户端在拿着权限许可和地址向 ChunkServer 请求目标数据文件，ChunkServer 会把对应的数据文件发送给 GFS 客户端。

文件写入方面，GFS 客户端会向 GFS Master 询问数据所在的服务器，GFS Master 会返回一个主服务器和 2 个副本服务器。然后 GFS 客户端会把数据发送给临近的服务器，由服务器之间来传输数据副本，而不是直接发送 3 份数据副本，这涉及数据一致性和通信问题。而收到的数据也不是立刻写入硬盘，而是先缓存起来，以防数据传输出了问题导致写入硬盘无意义。等 3 个服务器确认收到数据后，由主服务器发出指令统一写入硬盘，3 个服务器都完成写入操作后，由主服务器向 GFS 客户端发送写入完成信息。

3 BigTable

3.1 简介

Bigtable 依托于 GFS、Chubby 及 SSTable 诞生，用于解决 Google 内部不同产品在对数据存储的容量和响应时延需求的差异化，力求在确保能够容纳大量数据的同时减少数据的查询耗时。与 GFS 不同，BigTable 关心数据内容，根据数据的内容建立数据模型，并对外提供读写数据的接口。

3.2 数据模型

Bigtable 基本的数据结构和关系型数据库类似，都是以行列构成的表，但是，它还另外增加了新的维度——时间。也就是说，在行列确定的情况下，一个单元格（Cell）中有多个以事件为版本的数据。Bigtable 用 (row:string, column:string, time:int64) → string 表示映射关系。如果想要在表中查询指定版本的内容，我们需要指出行、列及版本。增加时间这个维度是因为“三驾马车”被设计出来的时候主要是为了支持搜索引擎，搜索引擎可能需要保留多个时间段的网页数据，而 GFS 也使用追加（Append）作为数据的主要修改方式，所以增加时间戳作为版本既充分利用了 GFS 的特性，也能满足业务的需求。

另外，Bigtable 还把多个 Column Keys 并入到被称为 Column Family 的集合中，并将 Column Family 作为访问控制的基础单元。我认为，这种方案其实是一种事务（Transaction）的实现方案。传统的事务以行为基本操作单位，在读写时对行上

锁以实现隔离，而 Bigtable 则是以 Column Family 为单位，这里的访问控制其实就是锁的思想。

3.3 数据定位

在 Bigtable 中，由于单个表（Table）存储的数据可能相当地多，那么读写的效率就会十分低下，于是 Bigtable 将 Table 分割为固定大小的 Tablet，将其作为数据存储和查找的基本单位。每当 Table 增加了这里要说明的是，tablet 是数据存储的基本单元，是用户感知不到的。而 Column Family 则是访问的基本单元，是编程时指定的，两者一前一后，不是一个概念。

因为是在分布式系统中，那么每个 Tablet 所在的机器不同，需要记录相关信息（METADATA）对其进行管理。而存储这些 METADATA 又需要分布式的系统，所以 Bigtable 又将这些 METADATA 的 METADATA 记录在一个文件中，并将这个文件的位置保存在 Chubby 中。即 Chubby 保存着 Root Tablet 的位置，Root Tablet 中保存着 METADATA Table 中所有 Tablet 的位置，METADATA Table 中保存着所有存储数据的 Tablet 的位置。这其中有几点值得注意。由于 Root Tablet 的特殊性，哪怕它的数据量再大，它也不允许被分割。METADATA tables 被读取到内存中以加快速度，其中存储的是以开始和结尾的 Row Key 作为键，tablet 位置作为值的映射。如果客户端希望读取特定的数据，那么它会以此读取 Chubby 中的文件，Root Tablet，METADATA Tablet，最后读取存储改数据的 Tablet。同时，为了加快读取的速度，它会将这些信息缓存到本地，直到信息失效。

3.4 数据分配

Bigtable 使用 Chubby 来检测 Tablet Server 的变化。这里的操作和 Zookeeper 的用法类似，当有新节点加入时，它需要在 Chubby 中新建一个对应的文件，并获取该文件的锁。由于所有的节点在 Chubby 中都有对应的文件，那么 Master 可以通过监听 Chubby 来获取所有 Tablet Server 的信息。这里有两种节点失效的情况，一种是仅仅回收了锁但是文件还在，这种情况很可能是节点崩溃了。由于节点不能自己退出，所以在 Master 节点得到该文件的锁后，它会将文件删除，以此表示节点退出。另一种情况是，文件已经被删除，这种情况说明节点是主动退出系统，那么可以直接重新分配 Tablet 给其他节点即可。

在正常的情况下，系统中会有大量数据写入，Master 需要负责将这些数据分配到合适的 Tablet Server。Bigtable 并没有明确指出分配所使用的的算法，但是它提出了一个要求。为了保证数据的一致性，同一时间，一个 Tablet 只能被分配给一个 Tablet Server。Master 通过向 Tablet Server 发送载入请求来分配 Tablet。如果该载入请求被 Tablet Server 接收到前 Master 仍是有效的，那么就可以认为此次 Tablet 分配操作已成功。

3.5 数据读写

每个 Tablet 由若干个位于 GFS 上的 SSTable、一个位于内存内的 MemTable 以及一份 Tablet Log 组成。为了保证系统可恢复，Google 首先使用 Table Log（即 WAL）将客户端发出的写操作请求记录在磁盘中，那么，一旦系统崩溃，仍然可以从磁盘读取数据，继续执行命令。然后，相关的数据被放入位于内存中的 Memtable 中，因为内存的速度相当快，那么执行排序等操作就要快得多。当 Memtable 的大小达到设定的值后，它就会以 SSTable 的形式被存储到 GFS 中，这被称为 Minor Compaction。

客户端的读操作请求则要综合考虑 Memtable 和 SSTable 中的数据，如果 Memtable 中已经有需要读的数据，就无需读取 SSTable。由于 Memtable 和 SSTable 都是有序的，所以读取的速度都相当快。由于 SSTable 中的数据有可能被标记为删除，那么我们需要定期对其进行处理，Bigtable 将其称为 Major Compaction。在这个过程中，Bigtable 会将过期或者被删除的数据删除，并合并多个 SSTable。这里似乎和 GFS 的 Garbage Collection 有点类似，但是我认为这可能是两个层面的活动。Bigtable 清理的是单个 Chunk 中的数据，而 GFS 清理的是磁盘中的单个 Chunk。

4 MapReduce

4.1 简介

MapReduce 是一种编程模式，其基本思想是把所有的计算都拆分成两个基本的计算操作，即 Map 和 Reduce。其中 Map 函数以一系列键值对（key-value）作为输入，然后输出一个中间文件（Intermediate）。这个中间态是另一种形式的键值对。然后，Reduce 函数将这个中间态作为输入，计算

得出结果。其中，Map 函数和 Reduce 函数的逻辑均由开发人员自行定义。以 WordCount 为例，准备要统计一本书中所有单词出现的次数。在 Map 函数中，我们每遇到一个单词 W，就往中间文件中写入（W，1）。然后，在 Reduce 函数中，把所有（W，1）出现的次数相加，就能得到 W 的出现次数 V。

4.2 具体流程

MapReduce 客户端会将输入的文件会分为 M 个片段，每个片段的大小通常在 16 64 MB 之间。然后在多个机器上开始运行 MapReduce 程序。系统中会有一个机器被选为 Master 节点，整个 MapReduce 计算包含 M 个 Map 任务和 R 个 Reduce 任务。Master 节点会为空闲的 Worker 节点分配 Map 任务和 Reduce 任务。执行 Map 任务的 Worker 开始读入自己对应的片段并将读入的数据解析为输入键值对。然后调用由用户定义的 Map 任务。最后，Worker 会将 Map 任务输出的结果存在内存中。在执行 Map 的同时，Map Worker 根据 Partition 函数将产生的中间结果分为 R 个部分，然后定期将内存中的中间文件存入到自己的本地磁盘中。任务完成时，Mapper 便会将中间文件在其本地磁盘上的存放位置报告给 Master。Master 会将中间文件存放位置通知给 Reduce Work。Reduce Worker 接收到这些信息后便会通过 RPC 读取中间文件。在读取完毕后，Reduce Worker 会对读取到的数据进行排序，保证拥有相同键的键值对能够连续分布。最后，Reduce Worker 会为每个键收集与其关联的值的集合，并调用用户定义的 Reduce 函数。Reduce 函数的结果会被放入到对应的结果文件。当所有 Map 和 Reduce 都结束后，程序会换新客户端并返回结果。值得说明的是，无论是输入文件到 Map Worker 的映射还是中间文件到 Reduce Worker 的映射都可以通过自定义的哈希函数来确定，论文中默认使用 Hash(key) mod R 来确定。另外，M 和 R 的值都是由用户指定的，应当比实际的机器数量要多一些，以此实现均衡负载。

4.3 容错及优化

在 MapReduce 中，容错也考虑 Master 和 Work 两种情况。

Master 节点会定期地将当前运行状态存为快照，当 Master 节点崩溃，就从最近的快照恢复然后重新执行任务。Master 节点会定期地 Ping 每个 Work 节点，一旦发现 Work 节点不可达，针对其当前执行的是 Map 还是 Reduce 任务，会有不同的策略。

如果是 Map 任务，无论任务已完成或是未完成，都会废除当前节点的任务。之后，Master 会将任务重新分配给其他节点，同时由于已经生成的中间文件不可访问，还会通知还未拿到中间文件的 Reduce Worker 去新的节点拿数据。如果是 Reduce 任务，由于结果文件存在 GFS 中，文件的可用性和一致性由 GFS 保证，所以 Master 仅将未完成的任务重新分配。

优化方面，如果集群中有某个 Worker 花了特别长的时间来完成最后的几个 Map 或 Reduce 任务，整个 MapReduce 计算任务的耗时就会因此被拖长，这样的 Worker 也就成了落后者。MapReduce 在整个计算完成到一定程度时就会将剩余的任务即同时将其分配给其他空闲 Worker 来执行，并在其中一个 Worker 完成后将该任务视作已完成。

5 结论

纵观 Google 的三篇论文，《Google File System》、《Google Bigtable》和《Google MapReduce》其实描述的就是 Google 的三种技术，GFS 分布式文件系统，Bigtable 分布式数据存储系统，MapReduce 编程模型，都是基于分布式并行运行的，都是部署在大量普通机器组成的集群之上，其实相互之间都有相似之处，也能协调在一起运行和工作。

通俗地讲，Google 的三驾马车解决的核心问题就是，如何用大量的廉价并且不稳定的 PC 去搭建一套健壮，高效的大规模数据存储和分析系统。

参考文献

[1] GHEMAWAT S, GOBIOFF H, LEUNG S T. The google file system [C]//Proceedings of the 19th ACM Symposium on Operating Systems Principles. Bolton Landing, NY, 2003: 20-43.

[2] CHANG F, DEAN J, GHEMAWAT S, et al. Bigtable: A distributed storage system for structured data[C]//7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2006: 205-218.

[3] DEAN J, GHEMAWAT S. Mapreduce: Simplified data processing on large clusters[C]//OSDI'04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA, 2004: 137-150.