Google三驾马车阅读小结

19301162 袁绍潭

前言

当下，互联网已经发展到了空前的高度，为用户带来便利，诞生了很多高端的技术：人工智能、的深度学习，让无数程序员失去头发。然而，早在二十一世纪初，当互联网仍可以被称作是“刀耕火种”的年代时，Google 为了应对自身的业务发展，而发表的一系列关于数据处理的论文，可以说奠定了整个数据处理领域的基石，因此又被誉为 Google 的三驾马车。

这三篇论文分别提出了：

- GFS (Google File System)
- MapReduce
- Bigtable

分别聚焦了文件系统、数据处理、数据库方面的问题，并提出了相应的解决方案，对互联网的发展有着深远的影响。

Google File System

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，可以提供容错功能。它可以给大量的用户提供总体性能较高的服务。

这篇文章提出了如下的场景：

- 故障无处不在
- 巨大的文件
- 修改文件不会覆盖原来数据，而是在文件尾部追加
- 应用程序和文件系统 API 的协同设计提高了整个系统的灵活性

基于此，论文提出了一个分布式文件系统，它可以在普通的硬件设备上运行，但是在低廉的成本下又提供了强大的容灾能力；同时也具有可伸缩性，可以根据实际需求改变容量的大小。

GFS 的架构由 Master、Chunk Server 和 Client 组成，它们其实是运行在 Linux 系统上的一些普通的应用程序（这也实现了在廉价的普通设备上运行的目标）。在存储文件时，所有文件都被分割成固定大小的 Chunk，写入 Chunk 服务器的磁盘上此外，一个 Chunk 是比较大的，有64MB，这可以显著减少 IO 次数和网络连接次数；虽然这也有一些负面的影响，但是总体来说利大于弊。

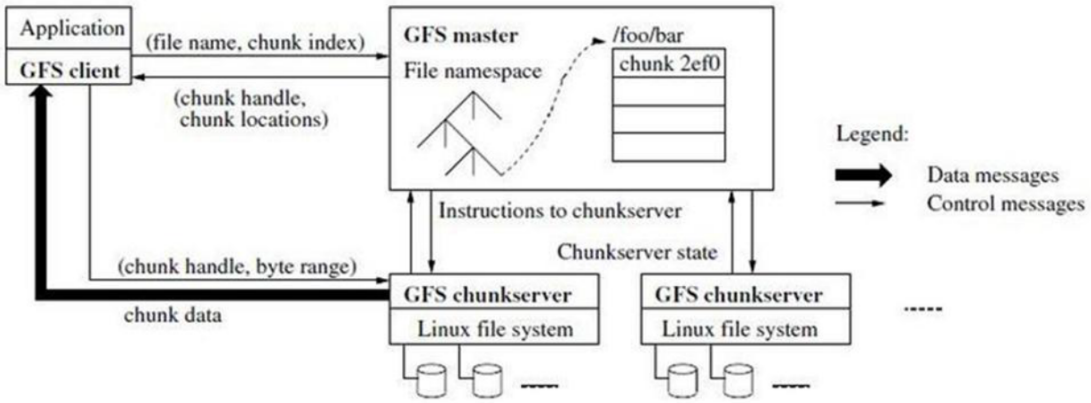


Figure 1: GFS Architecture

这种 Master-Chunk 的设计跟 Master-Slave 的设计很像，但是两者其实还是有所区别，后者 Master 作为整个系统的入口点，很可能在这一环影响到全局的性能。为了提升 Master 的效率，避免成为整个系统的单点故障，论文在 Master 的设计上下了很大功夫，包括：

- Master 只用来保存 Metadata，并且将它们保存在内存里
- Client 读取过 Metadata 后，会将它们缓存一段时间，减轻 Master 的压力
- 压缩 Metadata 的大小，使得 Master 的内存大小不会成为系统的瓶颈
- Chunk 的信息不会持久化存储，而是在启动时/定期进行轮询，出现问题也可以及时发现
- 定期做 Checkpoint，提升灾难恢复速度
- Master 做多台镜像，一台 down 掉了，别的可以迅速衔接

在 Chunk 的完整性方面，GFS 也做了大量的努力：

- 一个 Chunk 镜像到多台服务器上（默认 3 台）
- Chunk 划为更小的文件块（64KB），并计算相应的 checksum，读取时如果 checksum 对不上，就从别的服务器上读取
- Chunk Server 空闲的时候扫描空闲 Chunk，避免它们损坏了还没被发现

此外，GFS 在保证整个分布式系统的一致性上，除了以上提到的这些点外，还采取了一些 brilliant 的方法：

- 对于垃圾回收的机制，GFS 采用了一种延迟删除的机制，不会马上回收文件系统的空间，而是将文件改名，直到 3 天后才真正将它删除。这避免了在删除消息没有传到别的服务器上时，还可以继续将它们删除掉。这会带来一些空间浪费，如果用户的空间紧张，也可以进行手动的显式删除。
- Master 和 Chunk 之间使用一种叫做“租约”的机制来同步信息，Master 给每个 Chunk 分配一个租约，同时 Chunk 和 Master 之间使用心跳维持连接。如果主 Chunk 和 Master 失联了，Master 可以和另外的 Chunk 签订新的租约。

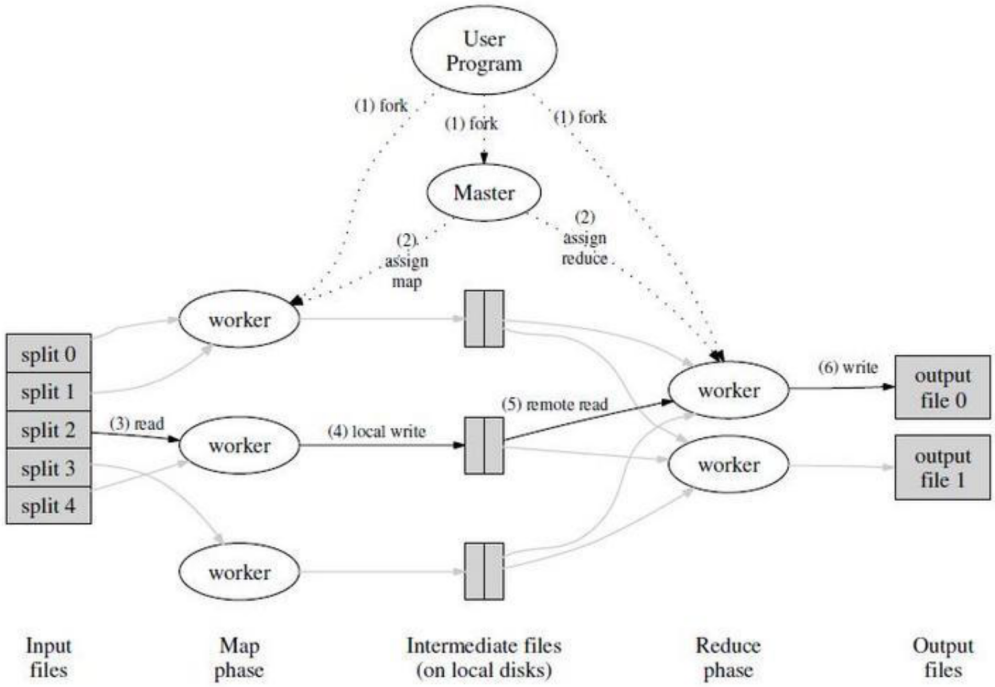
综上，GFS 虽然是 20 年前的论文，但是其中蕴含的很多思想至今仍被沿用，也为我们搭建一个分布式的文件系统提供了很多思路。

MapReduce

MapReduce 是一种将大规模任务在集群上进行分布式计算的实现，可以在大量的普通廉价计算机上进行超大规模的计算。

在模型的构建上，它的思路基本上是基于分治的，但是引入了 Map 和 Reduce 两个概念，Map 将输入的数据进行初步处理，得到一个规范化的、可被进一步处理的中间值，然后 Reduce 将这些中间值进一步进行运算，最终得出结果。

对于整个架构的实现，它其实也是运行在 Linux 上的普通用户程序，跟系统解耦合了。它采用的也是类似于 Master-Slave 的架构，有一个 master 负责分配任务：每个 Map 和 Reduce 都被切成许多小片，分配给 worker 执行。执行 Map 的 worker 将中间数据写入内存中，并且将位置回传给 master；master 再将这些位置传给 Reduce worker 执行相应的任务。Reduce 执行完任务后，将结果写入磁盘即可。



而对于任务划分的粒度，上面提到 Map 和 Reduce 会切分成 M 和 R 个小片。一个很直观的想法是，M 和 R 设的越大越好，当 M 和 R 的值很大时，每一个任务片的大小就降下来了，这意味着如果发生了错误，影响到的数据量并不大，可以迅速恢复；此外，一台 worker 可以执行多个任务，这对于负载均衡来说也是有益的。但是，考虑到物理条件的限制，M 和 R 并不能无限大：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；此外，考虑到输出结果是分为 R 个文件，这里也要考虑到下一步操作的影响。论文中提出了一组数据：用这样的比例来执行MapReduce：M=200000，R=5000，使用2000台worker机器。

考虑到分布式系统，容错也是一个大的挑战。对于 worker 来说，master 会定期 ping worker，如果超时未收到 worker 的信息，则就将其标记为失效，并且将分给它的任务标记为未完成，等待下一次调度。但对于 master 的失效，这篇论文并没有好的解决方案，而是直接将整个 MapReduce 任务结束，并让用户决定是否继续执行。这里有一点疑问：为何不参照 GFS 的设计理念，将 master 做多机镜像呢？这样，如果主 master 挂掉了，可以立即切换到镜像 master，并且镜像 master 向所有 worker 广播自己的信息，应当是可以实现无缝切换的。

除去容错，另一个常见的问题是，如果有某几个任务分片执行的非常慢，则可能别的任务都已经完成了，整个MapReduce还在等待这些慢任务执行完毕。为了解决这个问题，MapReduce 引入了一种 backup 的机制，在任务快执行结束时，master 会调度一些 backup 的 worker 来执行未完成的任务；同时，无论是 backup 执行结束，还是原来的主 worker 执行结束，这个任务都算完成。

MapReduce 这种思想现在已经被集成到很多现代的编程语言中了：例如，现在大热的 Python 中就内建了map()和 reduce()的函数，足以见 Google 的这篇论文对整个业界影响之深远。

Bigtable

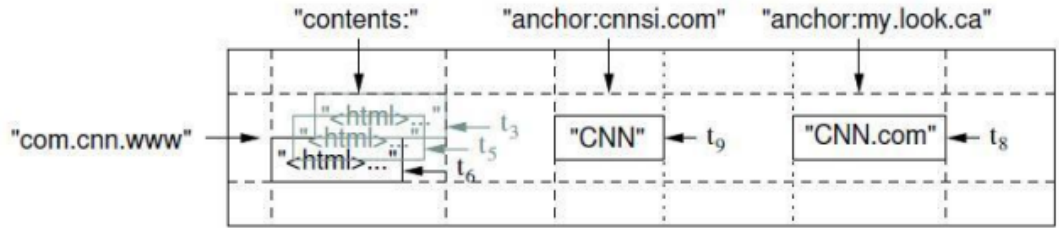
Google在Bigtable出现之前面临着很大的问题，就是很难解决对所有网站列表的快速储存和访问。谷歌层采用"网页存储"和"站点统计"存储，但是它们都有几个共同的特点：

- 数据量极大，TB，PB级别
- 和时间维度相关
- 同一个主键，属性与值有映射

传统二维small table，无法解决Google面临的存储问题，于是Google搞了一个big table来解决Google对这些业务模型进行分析，在二维table的基础上扩充，抽象了一个新的"三维table"：

- 主键，使用URL

- 属性，schema的列名，例如content，author等
- 时间，timestamp
- 值，不同URL的内容与作者等值



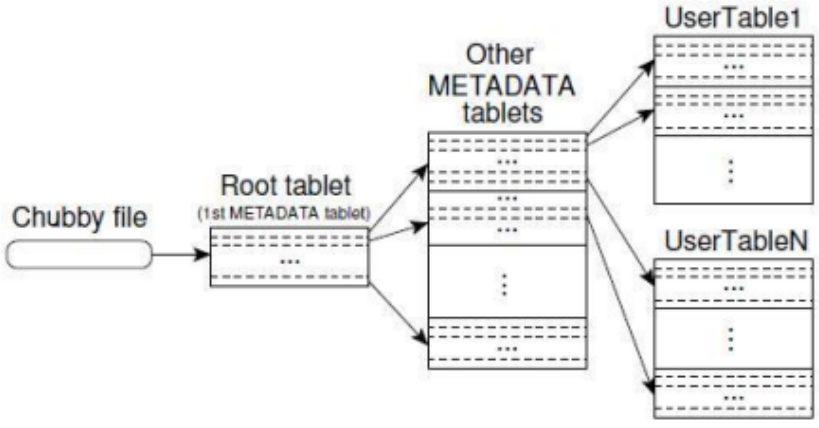
同一个key，不同属性，不同时间，会存储一个value。不像以行为单位进行存储的传统关系型数据库，这个三维的大表格BigTable是一个稀疏列存储系统。也就是说，Bigtable实现了空间的压缩，使数据能用更小的控件容纳。

它的**数据模型**的本质是一个map：

$$key + column + time => value$$

BigTable使用分布式Google文件系统（**GFS**）来存储日志文件和数据文件。

- 存储的文件格式为Google **SSTable文件格式。**一个SSTable提供一个持久的、排序的、不可变的、从键到值的映射。一个SSTable的大小是64KB（大小可配）。块的索引在SSTable的结尾，用来快速定位块的位置。当一个SSTable打开时，块索引就会先被读取进入内存。利用二分查找找到合适的块地址，最终才从磁盘读取相应的块。当然如果SSTable足够小可以读取进去内存，那么BigTable就会直接将该SSTable读取进去内存操作。
- 同时，BigTable还依赖**Chubby（一个高可用、持久性的分布式锁服务）**。Chubby使用Paxos算法来保持一致性。



Tablet的位置通过维护元数据表实现，每个元数据表分别记录了对应的表的位置。**元数据表存放在分布式锁服务Chubby中**，保证了数据的一致性。上述操作采用三层结构实现。**由于客户端不会直接与主服务器进行交互，所以客户端函数库会缓存Tablet位置信息。**客户端的Tablet位置信息保存在缓存中，速度快。同时，定期更新Tablet位置信息。

主服务器根据全局的负载情况分别Tablet到对应的Tablet服务器。BigTable使用Chubby来跟中Tablet服务器。主服务器见识对应的Chubby上的目录，来发现Tablet服务器。

显然主服务器把Tablet的位置信息转交到Chubby上，这样可以更专注的进行负载均衡，表、列创建等其他事情。

阅读感悟

阅读了Google三驾马车的论文后，我总结了以下几点：

- 大的分布式系统很容易发生多种错误，不仅是其他分布式系统遇到的网络分割和故障，而且还包括：内存与网络故障、时钟不对齐、机器挂起、依赖的其他系统组件问题、非计划之外的硬件维护。
- 不要盲目开发新的特性，需要根据实际需求进行开发。
- 需要一套合理的自我监控系统，包括监视进程。
- 遵循简单设计原则。