读谷歌三驾马车论文有感

作者: 田森茂 学号:19301099
学院班级：软件学院 1904 班

引 言

谷歌在 2003 到 2006 年间发表了三篇论文，《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。

关键词：大数据　谷歌　数据存储与分析

一、引言

在 21 世纪初，互联网上的内容，大多数企业需要存储的数据量并不大。但是 Google 不同，Google 的搜索引擎的数据基于爬虫，而由于网页的大量增加，爬虫得到的数据也随之急速膨胀，单机或简单的分布式方案已经不能满足业务的需求，所以 Google 必须设计新的数据存储系统，其产物就是 Google File System（GFS）。不过，在 Google 的设计中，为了尽可能的解耦，GFS 仅负责数据存储而不提供类似数据库的服务。也就是说，GFS 只存数据，而对数据的具体内容一无所知，自然也就不能提供基于内容的检索功能。所以，更进一步，Google 开发了 Bigtable 作为数据库，向上层服务提供基于内容的各种功能。此外，Google 的搜索结果依赖于 PageRank 算法的排序，而该算法又需要一些额外的数据，比如某网页的被引用次数，所以他们还开发了对于的数据处理工具 MapReduce，在读取了 Bigtable 数据的技术上，根据业务需求，对数据内容进行运算。其总体架构如下，GFS 能充分利用多个 Linux 服务器的磁盘，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，向上提供类似数据库的各种操作。MapReduce 则使用 Bigtable 中的数据进行运算，再提供给具体的业务使用。

二、MapReduce

2.1 编程模型

总的来讲，Google MapReduce 所执行的分布式计算会以一组键值对作为输入，输出另一组键值对，用户则通过编写 Map 函数和 Reduce 函数来指定所要进行的计算。由用户编写的 Map 函数将被应用在每一个输入键值对上，并输出若干键值对作为中间结果。之后，MapReduce 框架则会将与同一个键相关联的值都传递到同一次 Reduce 函数

调用中。同样由用户编写的 Reduce 函数以键以及与该键相关联的值的集合作为参数，对传入的值进行合并并输出合并后的值的集合。形式化地说，由用户提供的 Map 函数和 Reduce 函数应有如下类型：

$$\begin{array}{lll} \text{map} & (k1, v1) & \rightarrow \text{list}(k2, v2) \\ \text{reduce} & (k2, \text{list}(v2)) & \rightarrow \text{list}\,(v_2) \end{array} \tag{1}$$

值得注意的是，在实际的实现中 MapReduce 框架使用 Iterator 来代表作为输入的集合，主要是为了避免集合过大，无法被完整地放入到内存中。

2.2 计算执行过程

首先，用户通过 MapReduce 客户端指定 Map 函数和 Reduce 函数，以及此次 MapReduce 计算的配置，包括中间结果键值对的 Partition 数量以及用于切分中间结果的哈希函数。用户开始 MapReduce 计算后，整个 MapReduce 计算的流程可总结如下：作为输入的文件会被分为个 Split，每个 Split 的大小通常在 16 64 MB 之间如此，整个 MapReduce 计算包含个 Map 任务和个 Reduce 任务。Master 结点会从空闲的 Worker 结点中进行选取并为其分配 Map 任务和 Reduce 任务收到 Map 任务的 Worker 们（又称 Mapper）开始读入自己对应的 Split，将读入的内容解析为输入键值对并调用由用户定义的 Map 函数。由 Map 函数产生的中间结果键值对会被暂时存放在缓冲内存区中在 Map 阶段进行的同时，Mapper 们周期性地将放置在缓冲区中的中间结果存入到自己的本地磁盘中，同时根据用户指定的 Partition 函数（默认为）将产生的中间结果分为个部分。任务完成时，Mapper 便会将中间结果在其本地磁盘上的存放位置报告给 MasterMapper 上报的中间结果存放位置会被 Master 转发给 Reducer。当 Reducer 接收到这些信息后便会通过 RPC 读取存储在 Mapper 本地磁盘上属于对应 Partition 的中间结果。在读取完毕后，Reducer 会对读取到的数据进行排序以令拥有相同键的键值对能够连续分布之后，Reducer 会为每个键收集与其关联的值的集合，并以之调用用户定义的 Reduce 函数。Reduce 函数的结果会被放入到对应的 Reduce Partition 结果文件实际上，在一个 MapReduce 集群中，Master 会记录每一个 Map 和 Reduce 任务的当前完成状态，以及所分配的 Worker。除此之外，Master 还负责将 Mapper 产生的中间结果文件的位置和大小转发给 Reducer。值得注意的是，每次 MapReduce 任务执行时，和的值都应比集群中的 Worker 数量要高得多，以达成集群内负载均衡的效果。

2.3 容错机制

2.3.1 Worker 失效

在 MapReduce 集群中，Master 会周期地向每一个 Worker 发送 Ping 信号。如果某个 Worker 在一段时间内没有响应，Master 就会认为这个 Worker 已经不可用。任何分配给

该 Worker 的 Map 任务，无论是正在运行还是已经完成，都需要由 Master 重新分配给其他 Worker，因为该 Worker 不可用也意味着存储在该 Worker 本地磁盘上的中间结果也不可用了。Master 也会将这次重试通知给所有 Reducer，没能从原本的 Mapper 上完整获取中间结果的 Reducer 便会开始从新的 Mapper 上获取数据。如果有 Reduce 任务分配给该 Worker，Master 则会选取其中尚未完成的 Reduce 任务分配给其他 Worker。鉴于 Google MapReduce 的结果是存储在 Google File System 上的，已完成的 Reduce 任务的结果的可用性由 Google File System 提供，因此 MapReduce Master 只需要处理未完成的 Reduce 任务即可。

2.3.2 Master 失效

整个 MapReduce 集群中只会有一个 Master 结点，因此 Master 失效的情况并不多见。Master 结点在运行时会周期性地将集群的当前状态作为保存点（Checkpoint）写入到磁盘中。Master 进程终止后，重新启动的 Master 进程即可利用存储在磁盘中的数据恢复到上一次保存点的状态。

2.3.3 落后的 Worker

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（Straggler）。MapReduce 在整个计算完成到一定程度时就会将剩余的任务进行备份，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。

三、Bigtable

BigTable 其实就是 Google 设计的分布式结构化数据表.

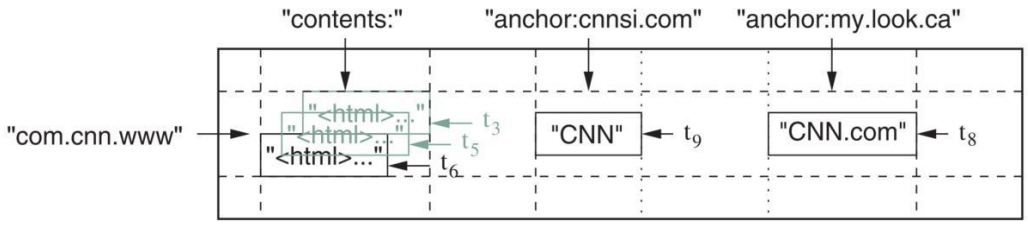
Bigtable 的设计动机: 1. 需要存储的数据种类繁多, 包括 URL、网页内容、用户的个性化设置在内的数据都是 Google 需要经常处理的需要存储的数据种类繁多海量的服务请求,Google 运行着目前世界上最繁忙的系统, 它每时每刻处理的客户服务请求数量是普通的系统根本无法承受的. 2. 商用数据库无法满足需求, 一方面现有商用数据库的设计着眼点在于其通用性。另一方面对于底层系统的完全掌控会给后期的系统维护、升级带来极大的便利

3.1 数据模型

Bigtable 会把数据存储在若干个 Table（表）中，Table 中的每个 Cell（数据单元）的形式如1：

Cell 内的数据由字节串（string）构成，使用行、列和时间戳三个维度进行定位。

图 1　数据模型



Bigtable 在存储数据时会按照 Cell 的 Row Key 对 Table 进行字典排序，并提供行级事务的支持（类似于 MongoDB，不支持跨行事务）。作为分布式的存储引擎，Bigtable 会把一个 Table 按 Row 切分成若干个相邻的 Tablet，并将 Tablet 分配到不同的 Tablet Server 上存储。如此一来，客户端查询较为接近的 Row Key 时 Cell 落在同一个 Tablet 上的概念也会更大，查询的效率也会更高。

3.2 系统原理

一个完整的 Bigtable 集群由两类节点组成：Master 和 Tablet Server。

Master 负责检测集群中的 Tablet Server 组成以及它们的加入和退出事件，会将 Tablet 分配至 Tablet Server，并负责均衡 Tablet Server 间的存储负载以及从 GFS 上回收无用的文件。除外，Master 还负责管理如 Table、Column Family 的创建和删除等 Schema 修改操作。

每个 Tablet Server 会负责管理若干个由 Master 指定的 Tablet，负责处理针对这些 Tablet 的读写请求，并负责在 Tablet 变得过大时对其进行切分。

Bigtable 集群会管理若干个 Table，每个 Table 由若干个 Tablet 组成，每个 Tablet 都会关联一个指定的 Row Key 范围，那么这个 Tablet 就包含了该 Table 在该范围内的所有数据。初始时，Table 会只有一个 Tablet，随着 Tablet 增大被 Tablet Server 自动切分，Table 就会包含越来越多的 Tablet。

3.3 集群成员变化与 Tablet 分配

Bigtable Master 利用了 Chubby 来探测 Tablet Server 加入和离开集群的事件。每个 Tablet Server 在 Chubby 上都会有一个对应的唯一文件，Tablet Server 在启动时便会拿到该文件在 Chubby 上的互斥锁，Master 则通过监听这些文件的父目录来检测 Tablet Server 的加入。如果 Tablet Server 失去了互斥锁，那么 Master 就会认为 Tablet Server 已退出集群。尽管如此，只要该文件仍然存在，Tablet Server 就会不断地尝试再次获取它的互斥锁；如果该文件已被删除（见下文），那么 Tablet Server 就会自行关闭。

在了解了集群中有哪些 Tablet Server 后，Master 便需要将 Tablet 分配给 Tablet Server。同一时间，一个 Tablet 只能被分配给一个 Tablet Server。Master 会通过向 Tablet Server 发

送 Tablet 载入请求来分配 Tablet。除非该载入请求在 Master 失效前仍未被 Tablet Server 接收到，那么就可以认为此次 Tablet 分配操作已成功：Tablet Server 只会接受来自当前 Master 的节点的请求。当 Tablet Server 决定不再负责某个 Tablet 时，它也会发送请求通知 Master。

Master 在检测到 Tablet Server 失效（互斥锁丢失）后，便会将其负责的 Tablet 重新分配。为此，Master 会尝试在 Chubby 上获取该 Tablet Server 对应的文件的互斥锁，并在成功获取后删除该文件，确保 Tablet Server 能够正确下线。之后，Master 便可顺利将 Tablet 分配至其他 Tablet Server。

如果 Master 与 Chubby 之间的通信连接断开，那么 Master 便会认为自己已经失效并自动关闭。Master 失效后，新 Master 恢复的过程如下：

在 Chubby 上获取 Master 独有的锁，确保不会有另一个 Master 同时启动利用 Chubby 获取仍有效的 Tablet Server 从各个 Tablet Server 处获取其所负责的 Tablet 列表，并向其表明自己作为新 Master 的身份，确保 Tablet Server 的后续通信能发往这个新 Master Master 确保 Root Tablet 及 METADATA 表的 Tablet 已完成分配 Master 扫描 METADATA 表获取集群中的所有 Tablet，并对未分配的 Tablet 重新进行分配

四、The Google File System

Google File System(GFS) 是一种适用于大型分布式数据密集型应用的可扩展分布式文件系统。它在大量普通机器上运行并提供容错能力，并为大量客户端提供高性能。

4.1 设计原则

节点失效是常态。系统由成百上千个普通机器组成，大量用户同时进行访问，这使得节点很容易因程序 bug、磁盘故障、内存故障等原因失效。因此，GFS 必须能够持续地监控自身状态，进行异常检测，同时具有很高的容错性，可以自动地从节点失效中快速回复。存储内容以大文件为主。系统存储上百万个大文件，每个文件通常几百 MB 或几 GB。系统需要支持小文件，但不需要对其进行优化。系统主要文件操作为大容量连续读、小容量随机读以及追加连续写。系统应该支持原子的文件追加操作，使得大量用户可以并行追加文件，而不需要额外的加锁机制。在 Goole 应用场景中，这些文件常用于生产者-消费者队列或者多路归并。系统的高吞吐量比低延时更重要。

4.2 集群架构

一个 GFS 集群由一个 Master 节点和若干个 Chunk Server 节点组成，可以被若干个客户端访问。每个节点作为用户级进程运行在 Linux 机器上。在存储文件时，GFS 会把文件切分成 64MB 的 Chunk 进行存储。Chunk 由一个 Master 分配的不可改变且全局唯一的 64 位 Chunk 句柄进行标识。Chunk Server 在本地磁盘上存储 Chunk，通过指定

Chunk 句柄和字节范围读取或写入 Chunk。为了保证可靠性，每一个 Chunk 会被备份到多个 Chunk Server 上，通常情况下存储三份 Replica。Master 负责维护整个集群的元数据，包括文件和 Chunk 的命名空间、访问控制信息、文件与 Chunk 的映射和 Chunk Replica 的位置信息。此外，Master 还负责 Chunk 的租期管理、回收和迁移。Master 也会周期地通过心跳包和 Chunk Server 进行通信，以此收集 Chunk Server 的状态并向其发送指令。客户端与 Master 交互获得集群的元数据。当客户端需要进行数据读写时，不会通过 Master 直接进行，而是询问 Master 应该与哪些 Chunk Server 通信，然后直接与 Chunk Server 通信进行数据读写，以此避免 Master 成为整个集群数据传输的瓶颈，同时，客户端会在一定时间内缓存 Chunk Server 信息，后续操作可以直接与 Chunk Server 进行通信。

4.3 高可用机制

首先是快速恢复。GFS 的组件被设计为可以在数秒钟内恢复它们的状态并重新启动。GFS 的组件实际上并不区分正常退出和异常退出：要关闭某个组件时直接 kill 进程即可。

然后是 Chunk Server。作为集群中的 Slave 角色，Chunk Server 失效的几率比 Master 要大得多。当 Chunk Server 失效时，其所持有的 Replica 对应的 Chunk 的 Replica 数量便会降低，当 Master 发现 Replica 数量低于用户指定阈值时，会进行重备份。此外，当 Chunk Server 失效时，用户的写入操作还会不断地进行，那么当 Chunk Server 重启后，Chunk Server 上的 Replica 便有可能是过期的。为此，Master 会为每个 Chunk 维持一个版本号，以区分正常的和过期的 Replica。每当 Master 将 Chunk Lease 分配给一个 Chunk Server 时，Master 便会提高 Chunk 的版本号，并通知其他的 Replica 更新自己的版本号。如果此时有 Chunk Server 失效，那么它上面的 Replica 的版本号就不会变化。在 Chunk Server 重启时，Chunk Server 会向 Master 汇报自己所持有的 Chunk Replica 及对应的版本号。如果 Master 发现某个 Replica 版本号过低，过期的 Replica 便会在下一次的 Replica 回收过程中被移除。此外，Master 向客户端返回 Replica 位置信息时也会返回 Chunk 当前的版本号，客户端在执行操作时都会验证版本号以确保不会读取到旧的数据。

接下来是 Master。Master 在对元数据做任何操作前对会用先写日志的形式将操作进行记录，只有当日志写入完成后才会响应客户端的请求，而这些日志也会备份到多个机器上。日志只有在写入到本地以及远端备份的持久化存储中才被视为完成写入。在重新启动时，Master 会通过重放已保存的操作记录来恢复自身的状态。为了保证 Master 能够快速地完成恢复，Master 会在日志达到一定大小后为自身的当前状态创建 Checkpoint，并删除 Checkpoint 创建以前的日志，重启时便从最近一次创建的 Checkpoint 及其后续的日志开始恢复。Checkpoint 以 B 树的形式进行组织，可以直接映射到内存中，且不做其他额外解析的情况下检索其所存储的 Namespace，进一步减少 Master 恢复所需的时

间。为了简化设计，同一时间只会有一个 Master 起作用。当 Master 失效时，外部的监控系统会侦测到这一事件，并在其他地方重新启动新的 Master 进程。此外，集群还提供具有只读功能的 Shadow Master。它们会同步 Master 的状态变更，但有可能延迟若干秒，其主要用于为 Master 分担读操作的压力。Shadow Master 会通过读取 Master 操作日志的某个备份来让自己的状态与 Master 同步；它也会像 Master 那样，在启动时轮询各个 Chunk Server，获知它们所持有的 Chunk Replica 信息，并持续监控它们的状态。在 Master 失效时，Shadow Master 仍能为整个集群提供只读功能，在 Master 因创建和删除副本导致副本位置信息更新时，Shadow Master 才和 Master 通信来更新自身状态。

最后是数据完整性。为了保证数据完整，每个 Chunk Server 都会使用校验和来检测自己保存的数据是否有损坏；在侦测到损坏数据后，Chunk Server 可以利用其它 Replica 来恢复数据。每个 Chunk 都分成 64KB 大小的 block。每个 block 都对应一个 32 位的校验和。和其它元数据一样，保存在内存和硬盘上，同时也记录操作日志。当进行数据读取操作时，Chunk Server 首先会利用校验和检查所需读取的数据是否有发生损坏，如此一来 Chunk Server 便不会把损坏的数据传递给其他请求发送者，无论它是客户端还是另一个 Chunk Server。发现损坏后，Chunk Server 会为请求发送者发送一个错误，并向 Master 告知数据损坏事件。接收到错误后，请求发送者会选择另一个 Chunk Server 重新发起请求，而 Master 则会利用另一个 Replica 为该 Chunk 进行重备份。当新的 Replica 创建完成后，Master 便会通知该 Chunk Server 删除这个损坏的 Replica。当进行数据追加操作时，Chunk Server 可以为位于 Chunk 尾部的校验和块的校验和进行增量式的更新，并且用所有追加来的新的校验和块来计算新的校验和。即使是被追加的校验和块在之前已经发生了数据损坏，增量更新后的校验和依然会无法与实际的数据相匹配，在下一次读取时依然能够检测到数据的损坏。当进行数据写入操作时，Chunk Server 必须读取并校验包含写入范围起始点和结束点的校验和块，然后进行写入，最后再重新计算校验和。此外，在空闲的时候，Chunk Server 也会周期地扫描并校验不活跃的 Chunk Replica 的数据，以确保某些 Chunk Replica 即使在不怎么被读取的情况下，其数据的损坏依然能被检测到，同时也确保了这些已损坏的 Chunk Replica 不至于让 Master 认为该 Chunk 已有足够数量的 Replica。

参考文献

[1] 知乎

[2] Dean, Jeffrey and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” OSDI (2004).

[3] Chang, Fay W. et al. “Bigtable: A Distributed Storage System for Structured Data.” ACM Trans. Comput. Syst. 26 (2008): 4:1-4:26.

[4] GhemawatSanjay et al. “The Google file system.” Operating Systems Review (2003): n. pag.