

**非关系数据库论文读后感**

**姓名:童振宇**

**学号:18231500**

**《BigTable,结构化数据的分布式存储系统》读书笔记**

**1.介绍与摘要**

BigTable是一个管理海量结构化数据(PB级别=1099511627776KB)的分布式系统，在实际应用的场景中，如谷歌的生态中，不同的应用需要不同级别的级别的响应速度和数据存储规模，但是BigTable面对这些不同情况的问题仍然给出了灵活且性能不错的处理方案。我认为这种存储系统在现今的互联网领域意义匪浅，比如在同一app中不同的产品服务需要不同的响应速度和数据规模，比如一些多人合作游戏中需要快速的响应时间而在后台一些如用户数据的加载响应速度就不需要这么快，再比如一些用户输入网址和用户画像所需的数据，这两者的数据规模就大相径庭，总之这种BigTable的存储方式在我们现在的互联网开发中至关重要。

之后，作者介绍了近年来BigTable在谷歌生态中的应用和成果，BigTable实现了在大数据和大量分布式机器中的广泛适用性、可扩展、高性能和高可用性。

作者也提到，BigTable和传统的数据库实现方式的最大不同在于Bigtable并不支持完整的关系数据模型(BigTable也不是传统意义上的关系型数据库)，而是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布局和格式，换言之BigTable中的数据是没有格式的，也就是就是数据没有表,视图等多种对象的集合，即Schema，用户需要自己去定义Schema。BigTable中数据数据用行和列的名字进行索引，名字可以是任意的字符串，数据本身也会被BigTable视为未被解析的字符串。

**2.数据模型**

正如作者之前提到的，BigTable中的数据是一个未被解析的字符串，并且由行列的名字来决定，也就是一个特殊的映射（Map），作者也提到Bigtable是一个稀疏的、分布式的、持久化存储的多维度排序Map，具体映射规则如下：

**(row:string, column:string,time:int64)->string**

原文中作者举了网站存储作为例子，因为在当今社会互联网网站的互联量非常的大，网站之间需要用到链接互联，所以作者在设计这个数据类型的时候，行名是一个反向URL。contents列族容纳的是网页的内容，anchor列族容纳引用该网页的锚链接文本不仅能存储网站自身的内容，还可以存储网站之间的联系内容，同时时间戳可以记录不同时间下网站的相关信息。

table{

"1" : {“header”:”1321”, “content”:54854},//一行

"aaaaa" : {sth.},

"aaaab" : {sth.},

"xyz" : {sth.},

"zzzzz" : {sth.}

}

接下来作者单独介绍了行关键字，列集合(列簇)和时间戳。

行关键字：在BigTable中使用字符串来存储，同时在同一行关键字下的读写操作都是原子性的，在BigTable中将一定范围内的行进行分区，BigTable以分区(Tablets)为单位进行数据分布和平衡各方需求的负载，这样做最大的好处在于，当需要查找处于一定范围即在同一分区内的数据时，分区的设计就会大大减少访问的次数。

列簇：列簇就是列关键字的集合，在BigTable中同一列的数据类型是相同的，这样非常方便存储和压缩。

时间戳：在数据存储中，时间戳起到了类似于版本控制的内容，因为不同的时间戳代表着不同的索引，从而可以代表同一数据在不同时间下的状态，如数据为网站的访问量，行和列可以代表URL和网站内部构造，通过不同的时间戳代表不同的索引从而代表同一网站不同时间的访问量。同时增加了一些其他机制来减少版本控制的负担，如设置参数标记废弃版本，把爬虫爬过的时间戳保留下来等等。

**3.API**

Bigtable 提供了建立和删除表以及列族的 API 函数，还提供了修改集群、表和列族的元数据的 API函数，以及根据行列进行查找遍历的函数。文中作者用了C++进行演示。同时可以和MapReduce ——一个 Google 开发的大规模并行计算框架联动，使得Bigtable作为MapReduce的输入输出。

**4.相关模块**

BigTable建立在其他的google配件之上，比如它使用google GFS来存储日志和数据，使用google SSTable这一map结构来进行存储数据，当需求查询Bigtable时，首先遍历SSTable找到块索引，之后把块索引加载到内存，这样找寻数据的速度就会快很多。并且BigTable还利用了Chubby这一锁服务，它不仅能保证在BitTable中的读写是原子的，同时提供了特殊的机制使得总有一个活动是主要的(Master), 即主(Master)服务器和tablet服务器技术.

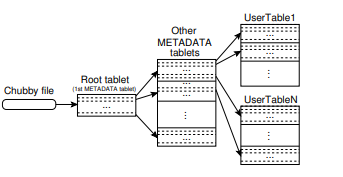
**5.部署**

Bigtable的相关组成有:连接到客户端的库(library),一个主(Master)服务器和大量的片(Tablet)服务器, 主服务器的功能包括为片服务器分配存储片(Tablets), 检测片服务器的相关信息(比如加入时间和是否过期)，以及和之前说的对读写负载进行均衡，垃圾处理等待。

每个片(Tablets)服务器都是一个Tablets的管理操控者，它的功能包括通过索引读写Tablets，并且在Tablets过大时，进行分割。

同时用户和BigTable的交互是跳过主(Master)服务器直接和片(Tablet)服务器交互的，所以主服务器可以更轻便的处理读写负载均衡和管理Tablets服务器的操作。（一个table大概为100MB到200MB之间）。

接下来作者详细的介绍了Bigtable中服务器的具体实现



首先我们使用三层B+树的数据结构来实现存储Tablet，一个Tablet服务器只能放一个Tablet，主(Master)服务器会实时监视Tablet服务器，不仅为了平衡负载，而且是为了把一些数据分配给有足够空余位置的Tablet服务器，通过扫描不同的Tablet服务器上不同Chubby锁的状态来决定如何分配，当Tablet被分配给Tablet服务器后,BigTable会使用Chubby锁来监视这个服务器来决定当前Tablet的状态(比如是否被合并,分开等)，之后重复该过程。

Tablet的持久化状态信息保存在GFS上。更新操作提交到日志中。更新操作保存在memtable中。随着写操作的执行，memtable的大小不断增加。当memtable的尺寸到达一个数值时，这个memtable就会被锁定，然后创建一个新的memtable；被锁定住的memtable会被写入GFS。客户程序可以将多个列族组合成一个局部性群族。对Tablet中的每个局部性群族都生成一个单独的SSTable。

当对 Tablet 服务器进行读写操作时，Tablet 服务器首先要根据 Chubby 锁里读取出来的具有写权限的操作者列表来进行验证。成功的修改操作会记录在提交日志里。可以采用批 量提交方式来提高包含大量小的修改操作的应用程序的吞吐量。当一个写操作提交后，写的内容 插入到 memtable 里面。

**6.优化与性能评价**

作者还介绍了相关的性能优化方法论和相关的评价。

优化的相关方法如下：

部分化分组，就是将不同的列族合成一个，之后对每个部分化分组生成一个单独的SSTable，这样最大的好处是可以把相同性质或者相同访问频率的数据放在一起，方便查询同时加快了查询速度。同时不同的分组可以有不同的标记参数，常访问的分组可以放在内存中，就不需要再次从磁盘读取(这个是redis的重要特性,也是redis速度如此之快的原因)。

压缩，我们可以通过一些“定制”的算法来压缩SSTable，这样可以减少读取索引所需要的时间，在本文中作者介绍了二次压缩方式，依次是Bentley and Mcllroy’s 算法和快速压缩算法。

通过缓存优化，为了提高读操作的性能，Tablet 服务器使用多级缓存。本文中作者介绍了二级缓存，扫描缓存是第一级缓存，主要缓存Table服务器通过 SSTable 接口获取键值对；Block 缓存是二级缓存，缓存的是从 GFS 读取的 SSTable 的Block，这对于一些经常访问的数据来说可以提升很高的效率。其中布隆过滤器在缓存优化中起到了至关重要的作用，布隆过滤器是为了检验大量数据集中有没有需要查找的数据，原理就是通过多次哈希变换尽可能的减少哈希碰撞的几率，这样就可以方便缓存策略，牺牲一部分的内存利用来换取读取硬盘的消耗，但是需要注意，就算经过多次哈希仍然存在小概率哈希碰撞的可能性。除此之外，还有提交日志，恢复提速，利用不变性等等方法也常用来优化缓存策略。

之后作者介绍了通过实验证明了这种存储方式对分布式系统读写大数据的性能提升，在机器的数量从一个提升到500台后，作者形容读写速度有了“梦幻般的提升”，但是由于CPU限制和分配任务算法的问题导致性能的提升仍然有瓶颈。但是也可以证明BigTable思想的想法对大数据分布式系统性能提升之大。

最后作者介绍了BigTable在谷歌地图，谷歌分析，个性化查询中的应用。

**《MapReduce:大型集群的数据简化处理》读书笔记**

1. **摘要与介绍**

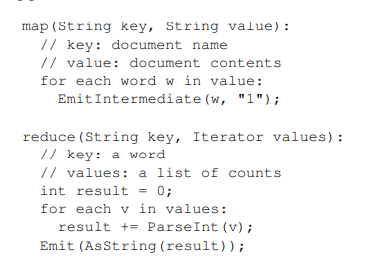
作者在一开始就介绍了当时大数据开发的困境，虽然可以通过各种集群操作获得大量的数据，比如通过爬虫得到网站的信息，用户搜索的关键词，web各种图示结构等，但是在对这些数据的运算却又是个大问题，比如两个数据集合求交并集，求出被用户搜索最多的关键字(类似于现在社交平台的热搜)等等，虽然概念上这些操作很好理解，但是因为数据量巨大，在对这些数据进行相关运算时，计算时间不能控制在可以接收的范围内，在本论文中作者就提供了MapReduce的方法来解决这个问题， MapReduce计算往往由几千台机器组成、处理TB数量级的数据。截至目前MapReduce已经实现了数以百计 的 MapReduce 程序，在 Google 的集群上，每天都有 1000 多个 MapReduce 程序在执行。

MapReduce的思想来源于类似于Lisp的函数式编程语言中的Map和Reduce简单且不严谨的说就是首先通过Map函数把信息转化成key/value形式，对于大数据操作来说就是把大数据分成一个key/value集合，再通过Reduce操作将相同key进行合并，得到合并中间值，这样可以简化运算得到结果。

同时这个系统经过一系列的优化，可以做到只关心如何分割输入数据，在大量计算机组成的集群上的调度，集群中计算机的错误处理，管理集群中计算机之间 必要的通信，从而使程序员专注于开发。同时 MapReduce 架构可以使那些没有并行计算和分布式处理系统开发经验的程序员有效利用分布式系统的丰富资源进行大规模并行运算。

1. **编程模型**

MapReduce的函数模型也很简单，正如MapReduce的名字一样，就是map和reduce，首先是map函数它可以接收一个key值集和一个value值集，产生一个key/value的中间集，之后把同一个key的value中间集，之后把这个中间集发送给reduce函数，reduce函数会检查这个从map传来的key/value值，之后把同一个key下的value合并起来，形成一个更小的集合。以下是伪代码段：



需要注意，作者在伪代码中使用了字符串来做演示，但是事实上可以用户自定义数据类型。

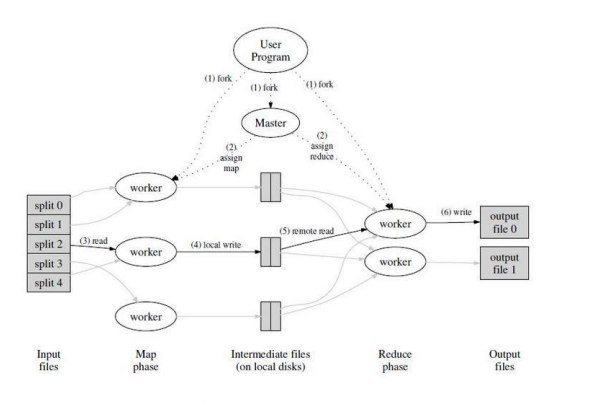
之后作者举了一些例子来说明MapReduce的思想，最好理解的就是用户对URL的点击率，, 网站管理端获得不同点击量初始化key/value即为(URL,1)这是Map操作，之后Reduce操作把所有相同的URL相加起来即为(URL,访问量).

1. **具体部署**

MapReduce的具体部署方式很多，可以使用小型机器共享，也可以使用大型网络连接集群，在本论文作者介绍了一种谷歌内部的部署方式，通过以太网连接PC来部署。

以下是基本配置x86架构、运行Linux 操作系统、双处理器、2-4GB 内存的机器。 普通的网络硬件设备，每个机器的带宽为百兆或者千兆，但是远小于网络的平均带宽的一半。 集群中包含成百上千的机器， 存储IDE 硬盘。一个内部分布式文件系统用来管理存储在这些磁盘上的数据，文件系统通过数据复制在硬件上保证数据的可靠性和有效性。用户提交任务给调度系统。

以下是大致图解



输入数据后，系统会调用Map函数，Map函数会将输入的key/value为自动分割 M 个数据片段的集合，因为是集群式，所以Map被分布到多台机器上执行使用分区函数将 Map 调用产生的中间 key 值分成 R 个不同分区，同样的Reduce 调用也被分布到多台机器上执行。分区数量（R）和分区函数可以由用户来指 定。

程序中的master类似于BigTable中的主(Master)服务器。它是用来控制其他 worker 程序，例如master 将一个 Map 任务或 Reduce 任务分配 给一个 worker。

被分配了map 任务的 worker 会读取相关的输入数据片段，从输入的数据片段中解析出 key/value，然后把 key/value pair 传递给用户自定义的 Map 函数，由 Map 函数生成并输出的中间 key/value pair，并保存在内存中。 缓存中的 key/value pair 通过分区函数分成 R 个区域，之后周期性的写入到本地磁盘上。缓存的 key/value pair 在本地磁盘上的存储位置将被回传给 master，由 master 负责把任务再传送给 Reduce worker。之后master会把消息发送给Reduce worker 程序，Reduce会读取这些缓存数据。读取完毕后，让具有相同 key 值的数据聚合在一起。从而完成reduce任务。

因为是大量的集群机器做并行运算，所以出错是不可避免的，作者在第三章的剩余篇幅讨论着重讨论了关于容错处理，针对普通的worker故障，mapreduce的处理策略是通过master周期性的对所有的worker进行ping操作，如果没能在给定的时间内得到反馈，则认定该worker实现，本该分配给该worker的任务就会被分配给其他worker。至于master的失效，这个是非常难处理的，论文中的处理策略是一旦master失效，立刻取消整个mapreduce进程。同时根据MapReduce的特性，我们需要，我们的分布 式机群在任何情况下的输出都和程序不出任何错误、并且顺序的执行产生的输出是一样的，论文中作者提到通过map和reduce函数的读写设为原子操作，同时MapReduce中还有类似于多重备份操作，将一次读写的结果保存在多个临时文件中，读写完毕后把临时文件保存。

因为MapReduce是针对大数据的处理运算，这样对网络的宽带利用十分的大，作者通过把输入数据存储在集群中机器的本地磁盘上来节省网络带宽，并且通过GFS进行文件保存，这样以来所需的文件都从本地读取就可以减少网络宽带消耗。

就像我们生活中的木桶效应一样，mapreduce的效率很大程度上取决于系统中的掉队者，这种情况出现的可能性很多，比如一个硬盘质量等原因需要经常纠错，或者最GPU或者宽带的争夺可能会导致任务掉队。我们有一个通用的机制来减少掉队者的出现。当一个 MapReduce 操作接近完成的时候，master 调度备用任务进程来执行剩下仍处于处理中状态的任务。无论是最初的执行进程、还是备用任务进程完成了任务，我们都把这个任务标记成为已经完成。通俗点就是当已经有任务被报告完成后，就会有一个机制来帮助未完成的任务。

1. **优化**

作者接下来介绍了针对MapReduce的相关拓展功能进行优化。

分区函数，MapReduce在中间 key 上使 用分区函数来对生成的数据进行分区，之后再输入到后续任务执行进程。作者介绍了哈希函数进行分区。虽然哈希函数大部分时间可以产生平衡分区，但是有的时候，其它的一些分区函数对 key 值进行的分区将非常有用。这时我们就可能不需要平衡的分区，而是有侧重的。

顺序保证，在系统中key/value是按照key递增的顺序生成的，这样对后续对在排序输出的数据集也很有帮助。

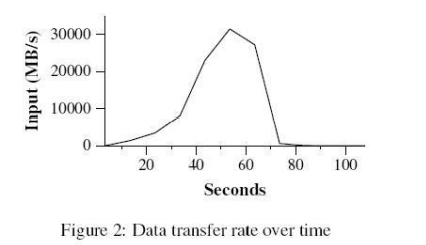
组合函数，reduce函数就是针对key对应的value进行合并，但是有些时候key对应的value量过大，同时value的分布符合一定的规律，这种情况下就可以通过组合函数首先对key/value对进行一次合并，将结果输出到中间文件，之后再输出到reduce函数中，事实证明这种方式的确可以增强效率。在论文的附件一中就有针对这一问题的具体实例——字数统计的例子。

忽略坏数据，正如之前提到，系统出现bug或者数据出问题时，就会导致掉队者现象，这时可以对很可能是错误的数据进行选择性的忽略，在大数据的背景下，这种选择性的忽略还是可以接受的，那么为了找出错误的数据，mapreduce在每个 worker 进程都设置了信号处理函数捕获内存段异常和总线错误检测，通过这些操作可以找出存在异常的数据并进行选择性跳过。

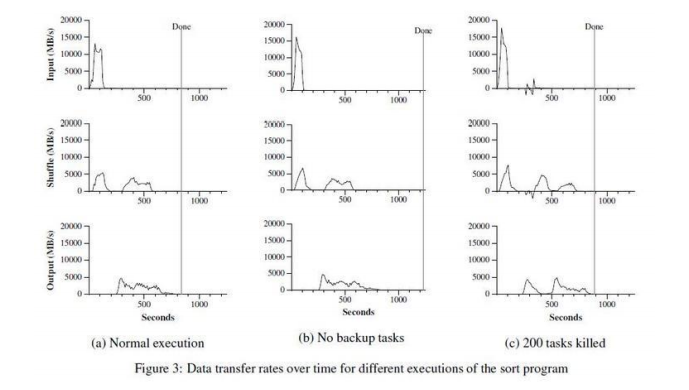
状态信息，用户可以通过master监控各种执行状态。状态信息页面包括已经完成了多少任务、有多少任务正在处理、输入的字节数、中间数据的字节数、输出的字节数、处理百分比等等。页面还包含了指向每个任务的链接。用户根据这些数据预测计算需要执行大约多长时间、是否需要增加额外的计算资源。这些页面也可以用来分析谁是掉队者。 另外，处于最顶层的状态页面显示了哪些worker 失效了，以及他们失效的时候正在运行的 Map 和 Reduce 任务，这个信息可以被我之前说的异常处理利用，把这个失效的worker的工作转移给其他worker。

1. **性能**

之后作者用在一个集群上运行的两个任务来衡量 MapReduce 的性能。一个是计算在大约 1TB 的数据中进行模式匹配运算，另一个计算对大约 1TB 的数据进行排序。下图展示了随着时间处理速度的变化。



下图展示了不同状态下排序的性能问题。



之后在论文最后作者介绍了该技术在谷歌相关项目上的应用。

**《Google File System谷歌文件系统》读书笔记**

1. **简介与摘要**

作者在本文中阐述了GFS即谷歌文件系统，一个面向大规模数据密集型应用的、可伸缩的分布式文件系统。 GFS运行在廉价的普遍硬件设备上，但是它依然能对灾难问题进行应对，为集群系统提供了高性能的服务。GFS和传统的早期文件系统有着不同之处，作者根据谷歌开发的实际情况，因地制宜，开发出了GFS。并且经过实践检验，GFS 完全满足了谷歌应用对存储的需求。GFS 作为存储平台已经被广泛的部署在 Google 应用，存储数据，同时还用于其他大规模数据集。

作者根据以下几点阐述了为什么在谷歌的实际情况下传统的早期传统文件系统难以胜任，以及设计出来的GFS的合理性:

组件失效在大数据的集群处理中是常态事件，而不是意外事件。千级的设备组成的集群存储同时并发的被大量的客户端访问，很难不发生错误，比如应用程序 bug、人为输入错误，硬盘、内存、连接器故障、网络信号波动，断电等造成的问题。所以，在GFS在持续的监控、错误侦测、灾难冗余以及自动恢复做了很多工作。

然后就是文件非常的大，大多数文件都是GB级别的，数以亿计的对象比比皆是，采用传统的用小文件来保存大文件是不明智的。

绝大部分的修改文件操作追加写文件，而不是覆盖原有数据。对文件的随机写入操作在实际中几乎不存在。一旦写完之后，对文件的操作就只有读，而且通常是按顺序读。大量的数据符合这些特性，针对这种针对海量文件的访问模式，客户端对数据块缓存是没有意义的，数据的追加操作是区别传统的文件系统和GFS的关键。

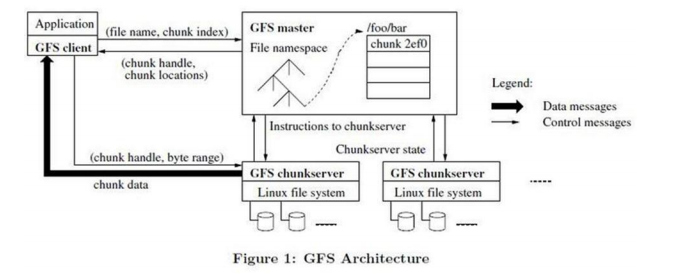
文件系统的API 需要更高的灵活性，比如减轻文件系统对应用的苛刻要求，以及GFS增加了原子性的读写这样就不需要单独在程序中增加同步操作。

1. **设计概述**

正如在上一节说的，GFS和传统的文件系统有着大的差别，比如GFS需要考虑随时出现的组件失效，大文件，大多数操作都是追加写，API的高灵活性，高并发性和高性能的稳定网络带宽，我们的目标程序绝大部分要求能够高速率的、大批量的处理数据，极少有程序对单一的读写操作有严格的响应时间要求。

在GFS中，作者设计了一系列类似于传统的文件系统的API，比如创建，删除，打开，关闭，读写文件，文件在这里以分层目录的形式组织，我们可以通过路径名来访问。另外，GFS 提供了快照和记录追加操作。这两者是原子性的读写操作，为了方便并行读写和追加写而设计的，之后会详细介绍到。

下图是GFS的具体架构



如图所示，一个GFS集群中有一个唯一的Master节点，一些运行着Linux系统的Chunk(组块)机器，并且同时被多个客户端访问，GFS存储的文件会被分割成大小相同的Chunk，之后Master会给每个Chunk分配标记，之后以Linux文件的格式保存在本地，并且根据标记和Chunk内部的范围读写数据，为了保证数据不被意外丢失，数据可能保存在多个Chunk机器上，同时正如之前说的，GFS中的API不同于POSIX标准，所以调用GFS的API比较灵活不需要深入Linux的代码层的级别，同时GFS只有一个master结点，用户通过master查询需要寻找的Chunk的位置，标志等信息，master将相关信息反馈给用户后，用户与Chunk进行读写的交互操作，需要注意的是，我们应该尽量避免对master进行读写。

读取的流程如下。客户发送数据，之后服务器把文件名和程序指定的字节偏移，再根据 Chunk大小，转换成文件的Chunk索引。然后，它把文件名和文件对应的Chunk 索引发送给 Master服务器，Master处理后，将相应的 Chunk 标识发还给客户端。之后客户用文件名和 Chunk 索引作为 key 缓存这些信息。

Chunk 的大小被设计成了64MB，大于传统文件系统的大小，虽然超过正常尺寸的内存块极有可能导致内部碎片浪费，但是考虑到它对于GFS系统的巨大用处，内部碎片的损失是可以接接受的，大容量的内存块可以只需要一次和 Mater 节点的通信就可以获取 Chunk 的索引，位置信息等，之后就可以对同一个 Chunk 进行多次的读写操作，这样可以大量减少工作负载，在大数据的背景下很有用，同时因为Chunk块很大，很多文件内容保存在同一个Chunk块中，这样客户方便对一个块进行多次操作就可以通过与 Chunk 服务器保持较长时间的 TCP 连接来减少网络负载。第三，选用较大的 Chunk 尺寸减少了 Master 节点需要保存的元数据的数量。这就允许我们把元数据全部放在内存中，以加速读写。

Chunk还有一个麻烦的问题就是热点问题，小文件包含很少的Chunk节点甚至只有一个Chunk节点，在对这个文件的并行访问量增加时，这个Chunk节点就有可能成为热点，即可能超载 ，论文中作者提及的解决方案就是允许客户端从其他客户端读取数据，而不是从Chunk节点读取，这样就避免了某个或某几个Chunk节点过载。

Master中存储 3 种元数据，分别是命名空间、对应关系(索引)、每个 Chunk节点的存放位置。所有的元数据都保存在 Master 服务器的内存中。前两种数据的变化会被保存系统日志文件中，日志文件存储在本地磁盘上，日志也会被同步到其它的远程Master服务器上。这样就可以避免因为不可避免的设备出错导致的信息丢失等问题。

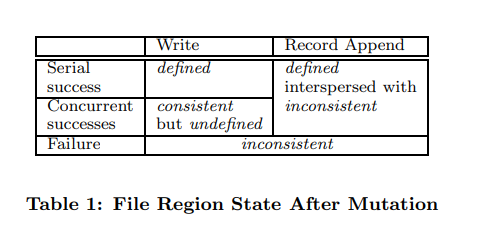
Master 服务器不会持久保存 Chunk 相关信息。Master 服务器在启动时，或者有新的 Chunk 服务器加入时，向各个 Chunk 服务器轮询它们所存储的 Chunk 的信息

因为Master的相关数据保存在内存中，这样可以保证Master的高速读写，虽然这样做，Chunk的数量以及整个系统的承载能力都受限于 Master 服务器所拥有的内存大小，但是为了保持系统的简洁，可伸缩和高性能，这是完全可以接受的，就算是需要支持更大的文件系统，为 Master 服务器增加额外内存付出的代价是很少的。

操作日志是元数据唯一的持久化记录，它包含了关键的元数据变更历史记录。在服务器在出现问题，进行恢复时，通过回滚操作日志把文件系统恢复上一个状态。为了缩短Master 启动的时间，尽快重新开始服务，我们必须使日志足够小。同时在日志达到一个数量阈值的时候进行一次检查点(CheckPoint)，这样在进行回滚的时候，读取检查点就行，可以加快整个进程，尽快使系统恢复工作，同时因为检查点的存在，我们可以实时的删除多余的日志文件以减少系统负载。

GFS 支持一个灵活的一致性模型，这个模型能够很好的支撑我们的分布式的应用，同时还保持了简单，易实现的优点。

文件命名空间的创建，删除都是是原子性的。它们仅由 Master 节点的控制：命名空间中的锁提供了原子性和正确性保障；Maste节点的操作日志定义了这些操作在全局的顺序。数据修改后文件数据的状态取决于操作的类型、成功与否、以及是否同步修改，下表列出了不同操作对应的结果。



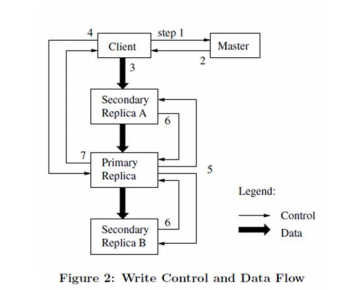
我们把“无论哪个客户端从哪个文件开始读取，读到的内容都是相同的”称之为“一致性”，如果对文件的数据修改之后，数据是一致的，并且客户端能够看到写入操作全部的内容，那么这个数据是“定义性的”。那么为了位置一致性和定义性，并且为了保持GFS中多为追加写的特性，我们在每次写完文件后记录下Chunk的偏移量作为索引的一部分，方便下次再写能直接找到追加的位置，并且经过了一系列的成功的修改操作之后，GFS 确保被修改的文件 数据是已定义的，并且包含最后一次修改操作写入的数据。GFS 通过以下措施确保上述行为：对 Chunk 的所有副本的修改操作顺序一致，使用 Chunk 的版本号来检测副本是否因为它所在的 Chunk 服务器出现问题而错过了修改操作 而导致其失效。失效的副本不会再进行任何修改操作，Master 服务器也不再返回这个 Chunk 副本的位置信息给客户端。它们会被垃圾系统回收。

1. **系统交互**

接下来作者介绍了系统中客户，master节点和chunk交互的细节，以及数据修改，原

子性读写的操作，快照功能以及如何让所有的操作简洁化。

当变更Chunk的数据时，比如写入操作或者记录追加操作，该操作会在 Chunk 的所有副本上执行。我们使用租用机制来保持多个副本的变更顺序的一致性，具体过程如下，master会为所有的chunk节点创建一个建立一个租用，副本可以对Chunk的所有更改操作进行序列化。 所有的副本都必须遵从这个序列进行修改操作。因此，修改操作全局的顺序首先由 Master 节点选择的租用的顺序决定，然后由租用中主Chunk分配的序列号决定。这样就可以减少Master的负担。租用的初始超时设置为 60 秒。不过，只要 Chunk 被修改了，主 Chunk 就可以申请更长的租期，通常会得到 Master 节点的确认并收到租约延长的时间。 这些租约延长请求和批准的信息通常都是附加在 Master 节点和 Chunk 服务器之间的心跳消息中来传递。有时 Master 节点会试图提前取消租约。即使 Master节点和主Chunk失去联系，它仍然可以安全地在旧的租约到期后和另外一个Chunk副本签订新的租约。



上图就是当客户和Master节点交互时，产生的多级缓存来，Master节点查询哪一个 Chunk 服务器持有当前的租约，以及其它副本的位置。如果没有一个 Chunk 持有租约，Master 节点就选择其中一个副本建立一个租约。

为了提高网络效率，GFS把数据和控制指令分开。在控制指令从客户到主Chunk、然后再到所有二级副本的同时，数据顺序的沿着其他Chunk服务器推送。这样就能充分利用每台机器的带宽，充分利用集群优势，最小化推送所有数据的延时。

GFS同时根据具体的情况，制作了和传统写不同的原子追加写，每次追加写都根据上次写后标记的偏移量，同时写操作不是串行的，每次追加写都有不同客户写入的片段。写入的数据追加到 GFS 指定的偏移位置上，之后 GFS 返回这个偏移量给客户，如果采用传统的文件读写，这时就需要大量的同步锁机制。如果记录追加操作在任何一个副本上失败了，客户端就需要重新进行操作。这样就能保证数据能被原子性的写入一次。

快照操作可以瞬间对一个文件做一个拷贝，并且几乎不会对正在进行的其它操作造成干扰。用户可以使用快照创建一个巨大的数据集的分支拷贝，或者是在做数据操作之前，使用快照操作备份当前状态，这样之后就可以轻松的提交或者回滚到备份时的状态，GFS使用copy-on-write 技术实现快照。

1. **Master节点的操作**

Master 节点执行所有的命名空间操作，管理整个系统里所有 Chunk副本，决定 Chunk 的存储位置，以及创建新 Chunk 和它的副本，保证在所有的 Chunk 服务器之间的进行负载均衡，回收不再使用的存储空间，接下来就是关于上述问题的详述。

不同于许多传统文件系统，GFS 没有针对每个目录实现能够列出目录下所有文件的数据结构。事实上，GFS 的名称空间就是一个全路径和元数据映射关系表。这个表可以高效的存储在内存中。并且每个节点都有一个关联的读写锁来保证高并行读写的同步。

Master同时负责管理Chunk副本，Chunk副本有三个用途，方便创建，重新复制和负载均衡。 当创建一个 Chunk节点时，它会选择在哪里放置初始的空的副本选择的标准如下尽量选择低于平均硬盘使用率的 Chunk 服务器，这样的做法最终能够平衡 Chunk 服务器之间的硬盘使用率。我们希望限制在每个 Chunk 服务器上“最近”的 Chunk 创建操作的次数，我们希望把 Chunk的副本分布在多个机器之间。 当 Chunk 的有效副本数量少于用户指定的复制因数的时候，Master 节点会重新复制它。

1. **错误处理**

正如开头说的，在大数据集群中部件失效是常有的事，作者接下来给出了一些GFS的应对策略。

使用两条简单策略保证整个系统的可用性：快速恢复和复制。 首先是快速恢复，只要是服务器出现关闭现象，它们都能在数秒钟内恢复它们的状态并重新启动。然后是复制，每个Chunk节点的信息都被复制到不同的其他Chunk 服务器上，当有 Chunk 服务器出现故障或者发现损坏的数据，Master节点通过已有的副本保证每个 Chunk 都被完整复制回来。除此之外还有其他方案，比如使用奇偶校验、Erasure 码等策略。最后就是复制Master 服务器，为了保证 Master 服务器的可靠性，Master本身也要被复制，包括所有的操作日志，checkpoint文件。这样以来当master的操作失效时，就可以很快重新启动。如果Master 节点所在的机器出现故障，处于GFS系统外部的监控进程会在其它的存有完整操作日志的机器上启动一个新的 Master 进程。

垃圾回收有以下两种策略：master会定时检测没有被文件引用的chunk并删除，在master和chunk server的心跳包中，chunk server会报告目前拥有的chunk，master跟自己的对比之后返回可以删除的chunk，然后chunk进行删除操作。

master发放租约的时候，会带上一个版本号给所有的副本，如果某个副本机器挂了，没有拿到这个版本号，那么在该副本重启的过程中，master会发现版本号不一致，则会让该副本失效。

**读后感总结**

在阅读完介绍BigTable的论文后，我了解到当今社会的大数据需求正在急剧上升，无论是在现在让人成瘾的短视频和淘宝商家对买家心理的精确把控等等，这无不有着大数据的影子，同时现在的计算机系统不再是一个计算机单枪匹马而是大量的计算机进行集群，通俗来说就是需求变大了，从一些小的数据变成了大数据，同时我们的设备也变强了，从单个系统变成了集群系统，这种情况下，如何让每台设备都能发挥最大作用就成了很好的研究课题，BigTable就给出了一些思路。它的出现大幅度提升了大数据读写效率，它的思想影响了后世很多的NoSql数据库如HBase中按行存储和行族的概念等, Bigtable是一个PB数量级的分布式存储系统，它存储的数据可以是任意的字符串，由行列和时间戳决定，并且每行的读写操作都是原子的；表中的行都可以进行动态分区，每个分区叫片(tablet)，tablet是数据分布和负载均衡的最小单位。 在具体的部署中，使用了主服务器的Tablet服务器，用户越过主(Main)服务器直接访问Tablet服务器中的Tablet里的数据，至于主(Main)服务器，它用来管理Tablet服务器的内容比如通过加Chubby来监听服务器的状态来进行分配服务器新的任务或者拆分重组数据库。同时Master 服务器自身从Chubby 获取一个唯一的Master 锁，用来阻止创建其它的Master 服务器实例，同时使用多级缓存，布隆过滤器等手段来优化储存。

MapReduce这一系统运用于保证速度和准确性的前提上处理从大数据系统上得到的数据，这个在当今的信息社会显得尤为重要，比如新闻的热搜需要找到在用户大数据中点击率最高的关键字，比如监视针对某一关键字得舆情分析。MapReduce过程其实就是由Map和Reduce这两个过程组成，Map可以将得到的数据转化成key/value，之后Reduce可以把相同的key对应得value进行合并。同时利用集群，将多台计算机的计算力合并起来，通过master来管理每一个worker，执行map任务的叫map worker,当然执行reduce任务的就叫reduce worker了。模型很简单，但是具体的部署细节和优化策略却很多，比如当worker出现故障时，master会及时检测到，然后把任务给其他worker，在某些worker出现拖后腿的现象时，master会让一个备份和它一起运行，同时还有诸如分区函数和组合函数等来优化系统效率。

GFS的设计和传统的文件系统有很大的区别，是谷歌根据实际情况因地制宜设计出来的新的系统，它主要表现在大数据集群的环境下组件失效是常态所以自动恢复，回滚和预防就成了很重要的话题，然后就是数据量大，大部分操作都是追加写而不是覆盖，同时GFS的API需要灵活等。针对这些需求，谷歌设计了这款GFS，首先设计了master和chunk节点来进行部署，主要包括master和chunk server两部分组成。一个文件被分散成了多个chunk进行保存，每个chunk会在其他chunk冗余备份，其中，master机器只有一台，在内存中存储：内存结构，chunk索引等，这样的设计巧妙的解耦了元数据和数据，master能专注于维护元数据，并且提供chunk的索引供对chunk读写，对于每个chunk都有多个副本。当客户端要修改数据的时候，master通过使用租约来保证多个副本之间的一致性。master从所有副本中选出一个chunk发放租约，得到租约的副本，称为一级缓存，其余的就是二级缓存。一级缓存将同一个chunk的操作进行序列化。通过心跳包可以续租。master server可以主动收回租约。

以上就是我的NoSql课程三篇论文的读书报告，这三篇论文的历史意义非凡，毫不夸张的说，这三篇论文的思想影响了后世大部分的非关系型数据库，同时这也是我第一次阅读英文顶尖论文，在阅读时我查阅了一些资料，并且因为学术水平和英文水平不够，对一些概念的理解可能会有偏差，希望老师多多指教。