谷歌技术"三宝"之谷歌文件系统

2012年04月21日 17:28:31

阅读数：50648

题记：初学分布式文件系统，写篇博客加深点印象。GFS的特点是使用一堆廉价的商用计算机支撑大规模数据处理。

虽然"*The Google File System* " 是03年发表的老文章了，但现在仍被广泛讨论，其对后来的分布式文件系统设计具有指导意义。然而，作者在设计GFS时，是基于过去很多实验观察的，并提出了很多假设作为前提，这等于给出了一个GFS的应用场景。所以我们自己在设计分布式系统时，一定要注意自己的应用场景是否和GFS相似，不能盲从GFS。

GFS的主要假设如下：

1. GFS的服务器都是普通的商用计算机，并不那么可靠，集群出现结点故障是常态。因此必须时刻监控系统的结点状态，当结点失效时，必须能检测到，并恢复之。
2. 系统存储适当数量的大文件。理想的负载是几百万个文件，文件一般都超过100MB，GB级别以上的文件是很常见的，必须进行有效管理。支持小文件，但不对其进行优化。
3. 负载通常包含两种读：大型的流式读（顺序读），和小型的随机读。前者通常一次读数百KB以上，后者通常在随机位置读几个KB。
4. 负载还包括很多连续的写操作，往文件追加数据（append）。文件很少会被修改，支持随机写操作，但不必进行优化。
5. 系统必须实现良好定义的语义，用于多客户端并发写同一个文件。同步的开销必须保证最小。
6. 高带宽比低延迟更重要，GFS的应用大多需要快速处理大量的数据，很少会严格要求单一操作的响应时间。

从这些假设基本可以看出GFS期望的应用场景应该是大文件，连续读，不修改，高并发。国内的淘宝文件系统（TFS）就不一样，专门为处理小文件进行了优化。

1 体系结构

GFS包括一个master结点（元数据服务器），多个chunkserver（数据服务器）和多个client（运行各种应用的客户端）。在可靠性要求不高的场景，client和chunkserver可以位于一个结点。图1是GFS的体系结构示意图，每一结点都是普通的Linux服务器，GFS的工作就是协调成百上千的服务器为各种应用提供服务。

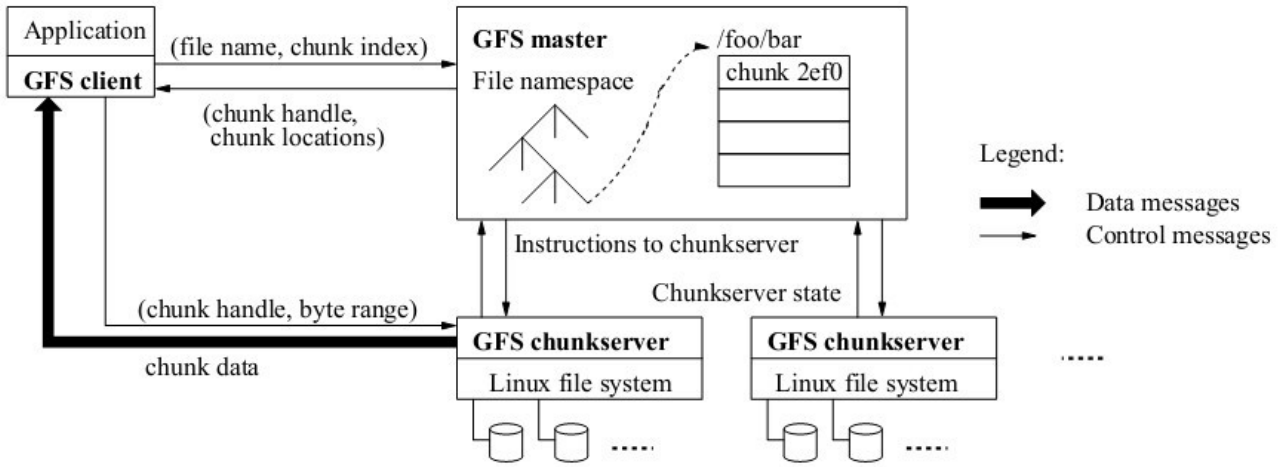


Figure 1: GFS Architecture

- chunkserver提供存储。GFS会将文件划分为定长数据块，每个数据块都有一个全局唯一不可变的id（chunk_handle），数据块以普通Linux文件的形式存储在chunkserver上，出于可靠性考虑，每个数据块会存储多个副本，分布在不同chunkserver。
- GFS master就是GFS的元数据服务器，负责维护文件系统的元数据，包括命名空间、访问控制、文件-块映射、块地址等，以及控制系统级活动，如垃圾回收、负载均衡等。
- 应用需要链接client的代码，然后client作为代理与master和chunkserver交互。master会定期与chunkserver交流（心跳），以获取chunkserver的状态并发送指令。

图1还描述了应用读取数据的流程。1.应用指定读取某个文件的某段数据，因为数据块是定长的，client可以计算出这段数据跨越了几个数据块，client将文件名和需要的数据块索引发送给master；2.master根据文件名查找命名空间和文件-块映射表，得到需要的数据块副本所在的地址，将数据块的id和其所有副本的地址反馈给client；3.client选择一个副本，联系chunkserver索取需要的数据；4.chunkserver返回数据给client。

2 数据的布局

GFS将文件条带化，按照类似RAID0的形式进行存储，可以提高聚合带宽。事实上，大多数分布式存储系统都会采取这种策略。GFS将文件按固定长度切分为数据块，master在创建一个新数据块时，会给每个数据块分配一个全局唯一且不可变的64位id。每个数据块以Linux文件的形式存储在chunkserver的本地文件系统里。

GFS为数据块设置了一个很大的长度，64MB，这比传统文件系统的块长要大多了。大块长会带来很多好处：1.减少client和master的交互次数，因为读写同一个块只需要一次交互，在GFS假设的顺序读写负载的场景下特别有用；2.同样也减少了client和chunkserver的交互次数，降低TCP/IP连接等网络开销；3.减少了元数据的规模，因此master可以将元数据完全放在内存，这对于集中式元数据模型的GFS尤为重要。

大数据块也有缺点。最大的缺点可能就是内部碎片了，不过考虑到文件一般都相当大，所以碎片也只存在于文件的最后一个数据块。还有一个缺点不是那么容易看出来，由于小文件可能只有少量数据块，极端情况只有一个，那么当这个小文件是热点文件时，存储该文件数据块的chunkserver可能会负载过重。不过正如前面所说，小文件不在GFS的优化范围。

为了提高数据的可靠性和并发性，每一个数据块都有多个副本。当客户端请求一个数据块时，master会将所有副本的地址都通知客户端，客户端再择优（距离最短等）选择一个副本。一个典型的GFS集群可能有数百台服务器，跨越多个子网，因此在考虑副本的放置时，不仅要考虑机器级别的错误，还要考虑整个子网瘫痪了该怎么办。将副本分布到多个子网去，还可以提高系统的聚合带宽。因此创建一个数据块时，主要考虑几个因素：1.优先考虑存储利用率低于平均水平的结点；2.限制单个结点同时创建副本的数量；3.副本尽量跨子网。

3 元数据服务

GFS是典型的集中式元数据服务，所有的元数据都存放在一个master结点内。元数据主要包括三种：文件和数据块的命名空间，文件-数据块映射表，数据块的副本位置。所有的元数据都是放在内存里的。

前两种元数据会被持久化到本地磁盘中，以操作日志的形式。操作日志会记录下这两种元数据的每一次关键变化，因此当master宕机，就可以根据日志恢复到某个时间点。日志的意义还在于，它提供了一个时间线，用于定义操作的先后顺序，文件、数据块的版本都依赖于这个时间顺序。

数据块的副本位置则没有持久化，因为动辄数以百计的chunkserver是很容易出错的，因此只有chunkserver对自己存储的数据块有绝对的话语权，而master上的位置信息很容易因为结点失效等原因而过时。取而代之的方法是，master启动时询问每个chunkserver的数据块情况，而且chunkserver在定期的心跳检查中也会汇报自己存储的部分数据块情况。

GFS物理上没有目录结构，也不支持链接操作，使用一张表来映射文件路径名和元数据。

4 缓存和预取

GFS的客户端和chunkserver都不会缓存任何数据，这是因为GFS的典型应用是顺序访问大文件，不存在时间局部性。空间局部性虽然存在，但是数据集一般很大，以致没有足够的空间缓存。

我们知道集中式元数据模型的元数据服务器容易成为瓶颈，应该尽量减少客户端与元数据服务器的交互。因此GFS设计了元数据缓存。client需要访问数据时，先询问master数据在哪儿，然后将这个数据地址信息缓存起来，之后client对该数据块的操作都只需直接与chunkserver联系了，当然缓存的时间是有限的，过期作废。

master还会元数据预取。因为空间局部性是存在，master可以将逻辑上连续的几个数据块的地址信息一并发给客户端，客户端缓存这些元数据，以后需要时就可以不用找master的麻烦了。

5 出错了肿么办

引用：“We treat component failures as the norm rather than the exception."

分布式系统整体的可靠性是至关重要的。GFS集群使用的都是普通的商用计算机，而且机器的数量众多，设备故障经常出现，如何处理结点失效的问题是GFS最大的挑战。

5.1 完整性

GFS使用数以千计的磁盘，磁盘出错导致数据被破坏时有发生，我们可以用其它副本来恢复数据，但首先必须能检测出错误。chunksever会使用校验和来检测错误数据。每一个块（chunk）都被划分为64KB的单元（block），每个block对应一个32位的校验和。校验和与数据分开存储，内存有一份，然后以日志的形式在磁盘备一份。

chunkserver在发送数据之前会核对数据的校验和，防止错误的数据传播出去。如果校验和与数据不匹配，就返回错误，并且向master反映情况。master会开始克隆副本的操作，完成后就命令该chunkserver删除非法副本。

5.2 一致性

一致性指的是master的元数据和chunkserver的数据是否一致，多个数据块副本之间是否一致，多个客户端看到的数据是否一致。

先来看看元数据一致性。GFS的命名空间操作是原子性的，并且用日志记录下操作顺序。虽然GFS没有目录结构，但是仍然有一颗逻辑的目录树，树的每个结点都有自己的读写锁，每个元数据操作都需要获得一系列的锁，应该是写锁会阻塞其它的锁，而读锁只阻塞写锁而不阻塞读锁。比如/home/user "目录" 正在创建快照，需要获得/home的读锁和/home/user的写锁，这时如果想创建文件/home/user/foo会被阻塞，因为需要获得/home、/home/user的读锁以及/home/user/foo的写锁，快照会阻塞创建操作获取/home/user的读锁。如果是在一个有传统目录树结构的文件系统里，创建一个文件需要修改父目录的数据，因此需要获得父目录的写锁。这种锁机制允许在一个目录里并发修改数据（如并发创建文件等），这在传统文件系统里是不允许的。

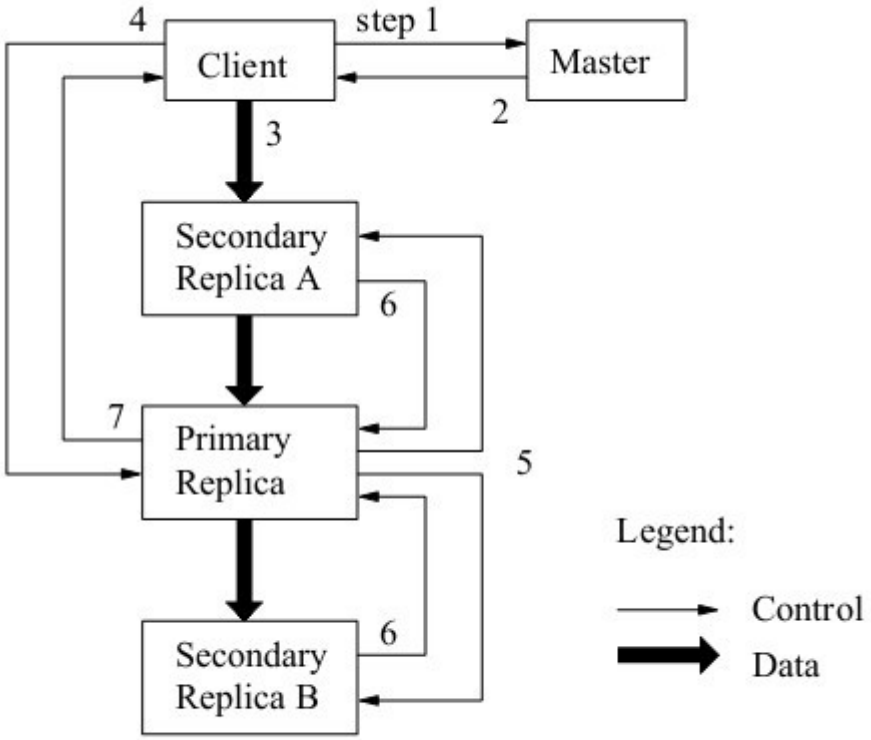


Figure 2: Write Control and Data Flow

再来看看GFS是如何并发写（write）的，GFS必须将对数据块的修改同步到每一个副本。考虑一下多个应用同时修改同一数据块的情况，我们必须为修改操作定义统一的时序，不然多个副本会出现不一致的情况，那么定义时序由谁做呢？还记得前面提到的元数据缓存么，为了减少master的负担，client在获得副本位置后就不再和master交互，所以必然需要选出一个master代理来完成这个任务。事实上GFS采用了租约（lease）的机制，master会将租约授权给某个副本，称为primary，由这个primary来确定数据修改的顺序，其它副本照做就是。

图2是写操作的控制流和数据流：

1. client需要更新一个数据块，询问master谁拥有该数据块的租约（谁是primary）；
2. master将持有租约的primary和其它副本的位置告知client，client缓存之；
3. client向所有副本传输数据，这里副本没有先后顺序，根据网络拓扑情况找出最短路径，数据从client出发沿着路径流向各个chunkserver，这个过程采用流水线（网络和存储并行）。chunkserver将数据放到LRU缓存；
4. 一旦所有的副本都确定接受数据，client向primary发送写请求，primary为这个前面接受到的数据分配序列号（primary为所有的写操作分配连续的序列号表示先后顺序），并且按照顺序执行数据更新；
5. primary将写请求发送给其它副本，每个副本都按照primary确定的顺序执行更新；
6. 其它副本向primary汇报操作情况；

7. primary回复client操作情况，任何副本错误都导致此次请求失败，并且此时副本处于不一致状态（写操作完成情况不一样）。client会尝试几次3到7的步骤，实在不行就只能重头来过了。

如果一个写请求太大了或者跨越了chunk，GFS的client会将其拆分为多个写请求，每个写请求都遵循上述过程，但是可能和其它应用的写操作交叉在一起。所以这些写操作共享的数据区域就可能包含几个写请求的碎片（就是下文提到的undefined状态）。

GFS还提供另一种写操作append record，append只在文件的尾部以record为单位（为了避免内部碎片，record一般不会很大）写入数据。append是原子性的，GFS保证将数据顺序地写到文件尾部至少一次。append record的流程和图2类似，只是在primary有点区别，GFS必须保证一个record存储在一个chunk内，所以当primary判断当前chunk无足够空间时，就通知所有副本将chunk填充，然后汇报失败，client会申请创建新chunk并重新执行一次append record操作。如果该chunk大小合适，primary会将数据写到数据块的尾部，然后通知其它副本将数据写到一样的偏移。任何副本append失败，各个副本会处于不一致状态（完成或未完成），这时primary必然是成功写入的（不然就没有4以后的操作了）。客户端重试append record操作时，因为primary的chunk长度已经变化了，primary就必须在新的偏移写入数据，而其它副本也是照做。这就导致上一个失败的偏移位置，各个副本处于不一致状态，应用必须自己区分record正确与否，我称这为无效数据区。

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

表1说明了GFS的一致性保证，明白write和append操作后就容易理解了。consistent指的是多个副本之间是完全一致的；defined指的是副本数据修改后不仅一致而且client能看到其修改的数据全貌。

- 成功的连续write是已定义的，各个副本数据一致；
- 成功的并发write能保证一致性性，即各个副本是一样的，但数据并不一定如用户所期望，如前所述，多个用户的修改可能交错在一起；
- 失败的write操作，使得副本之间不一致，而且数据undefined，不同client可能看到不同的数据（注意区别defined、undefined数据的方法）；
- 成功的append操作，不管是顺序还是并发都是defined，因为GFS保证了append是原子性的（atomically at least once）。有效数据区确实是defined的，但是失败append操作留下的无效数据区可能会有不一致的情况，所以中间可能零散分布着不一致的数据。

如上所述，在primary的协调下，能保证并发write的一致性。但还有一些可能会导致数据不一致，比如chunkserver宕机错过了数据更新，这时就会出现新旧版本的数据，GFS为每个数据块分配版本来解决这个问题。master每次授权数据块租约给primary之前，都会增加数据块的版本号，并且通知所有副本更新版本号。客户端需要读数据时当然会根据这个版本号来判断副本的数据是否最新。

5.3 可用性

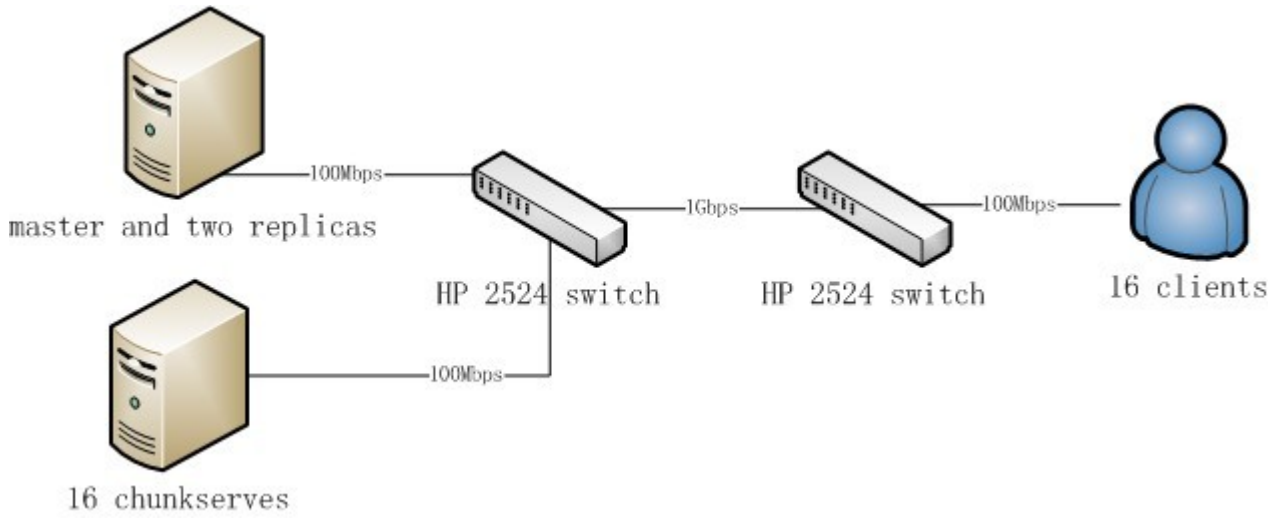
为了保证数据的可用性，GFS为每个数据块存储了多个副本，在数据的布局里有介绍，这里主要关注下元数据的可用性。

GFS是典型的集中式元数据模型，一个元数据服务器承担了巨大的压力，不仅有性能的问题，还有单点故障问题。master为了能快速从故障中恢复过来，采用了log和checkpoint技术，log记录了元数据的每一次变化。用咱们备份的话来说，checkpoint就相当于一次全量备份，log相当于连续数据保护，master宕机后，就先恢复到checkpoint，然后根据log恢复到最新状态。每次创建一个新的checkpoint，log就可以清空，这有效控制了log的大小。

这还不够，如果master完全坏了肿么办？GFS设置了“影子”服务器，master将日志备份到影子上，影子按照日志把元数据与master同步。如果master悲剧了，我们还有影子。平时正常工作时，影子可以分担一部分元数据访问工作，当然不提供直接的写操作。

6 测试

6.1 模拟



实验的网络拓扑图大概就是这样。集群包括一个master和它的两个影子，16个chunkserver，16个client，和两个HP交换机。两个交换机之间是1Gbps的链路，结点与交换机的链路是100Mbps。聚合带宽的理论上限是125MB/s，而单个client的理论带宽上限是12.5MB/s。

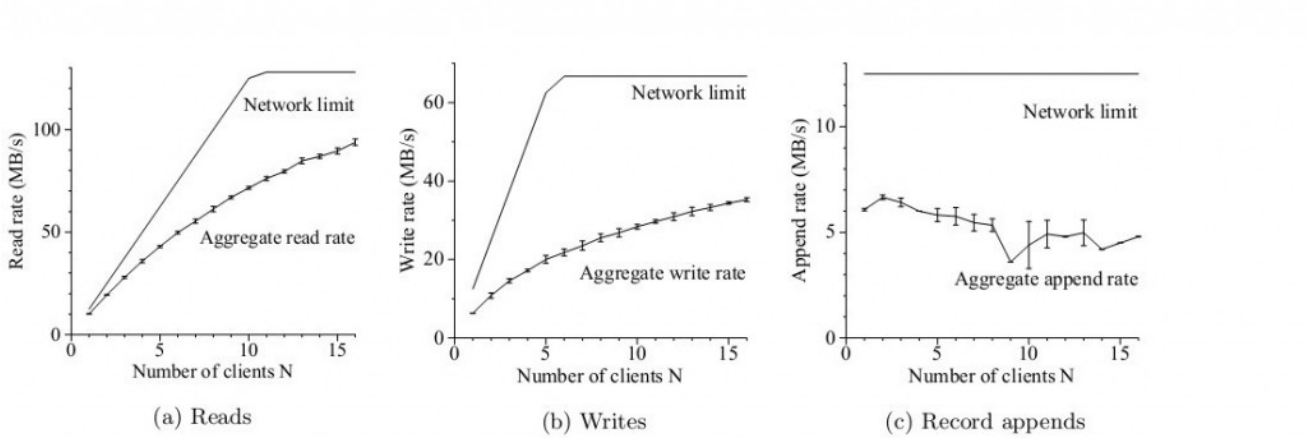


Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

实验一：N个client同时随机各自读取1GB数据。

图3(a)，x轴是N，y轴是聚合带宽，上面一条线是理论值，下面一条线是实际值。当N=1时，吞吐率是10MB/s，达到了理论值的80%。当N=16时，吞吐率是94MB/s，达到理论值的75%。此时瓶颈可能是在chunkserver，因为client数量很多，同时读一个chunkserver的概率很大。

实验二：N个client同时写N个不同文件，各自连续写1GB。

图3(b)。理论值上限是67MB/s（The limit plateaus at 67 MB/s because we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection），这个理论值我没有看明白是怎么算的。当N=1，吞吐率是6.3MB/s，达到极限的一半，主要的瓶颈是数据在副本之间传递。N=16时，聚合带宽为35MB/s（每个client有2.2MB/s），达到极限的一半，这里chunkserver同时接受多个请求的情况比读更严重，因为得写3个副本。写比期望的要慢。

实验三：N个client同时向一个文件append record。

图3(c)。理论上限值是一台chunkserver的带宽，即12.5MB/s。当N=1时，吞吐率有6.0MB/s。当N=16时，下降到4.8MB/s。这可能是因为拥塞。

6.2 现实

文章里还介绍了两个真实集群的使用情况，cluster A and B。

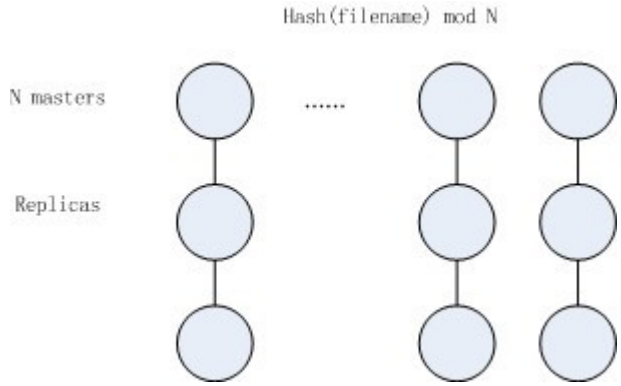
Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 2: Characteristics of two GFS clusters

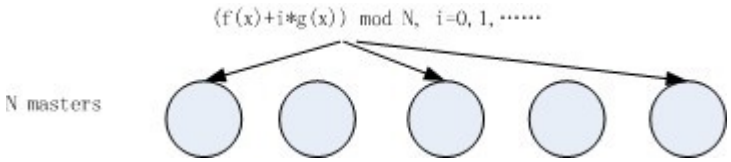
表2是两个集群的情况。A和B都有数百个chunkserver，存储利用率很高，冗余度是3，所以实际存储的数据是18TB和52TB。文件数相当，dead file是需要删除的文件，但还没被垃圾回收（GFS会为删除的文件保留三天才回收空间）。B的块数更多，意味着文件更大，不过A和B的文件平均都只有1到2个块。chunkserver的元数据主要是block（64KB）的校验和（4 Bytes），以及数据块的版本信息。master的元数据（文件和数据块的名字，文件-数据块映射，块位置等）相当小，平均每个文件只有100B元数据，大部分是用于存储文件名。平均下来，每个服务器有50~100MB的元数据。

7 胡说八道

画了个全分布式元数据模型。



这估计是最简单的，命名空间被划分为几个区域，master各管各的。为了可靠性，每个master做几个副本。映射方法可以是文件名计算哈希取模，好的哈希函数可以使文件随机分布，负载比较均衡。当然扩展性不是很好，加入新的结点，所有文件得重新映射。而且冗余度只能靠机器堆了，不能软件控制。



又构思了个复杂点的。将文件映射到r(<=N)个master，利用参数r控制冗余度，当r=N时就变成全对等元数据集群。这个模型需要r个不同的哈希函数，为了减少开销，可以用两个函数模拟多个函数。比如有随机性很好的f(x)和g(x)函数，我们用式子f(x)+i*g(x)来模拟，其中i为非负整数。当我们处理文件x时，就用前面式子求出r个不同的位置（有冲突的概率，实际可能不止计算r次）。

胡说八道，切勿当真；如有雷同，纯属巧合。

参考文献：

[1] The Google File System.