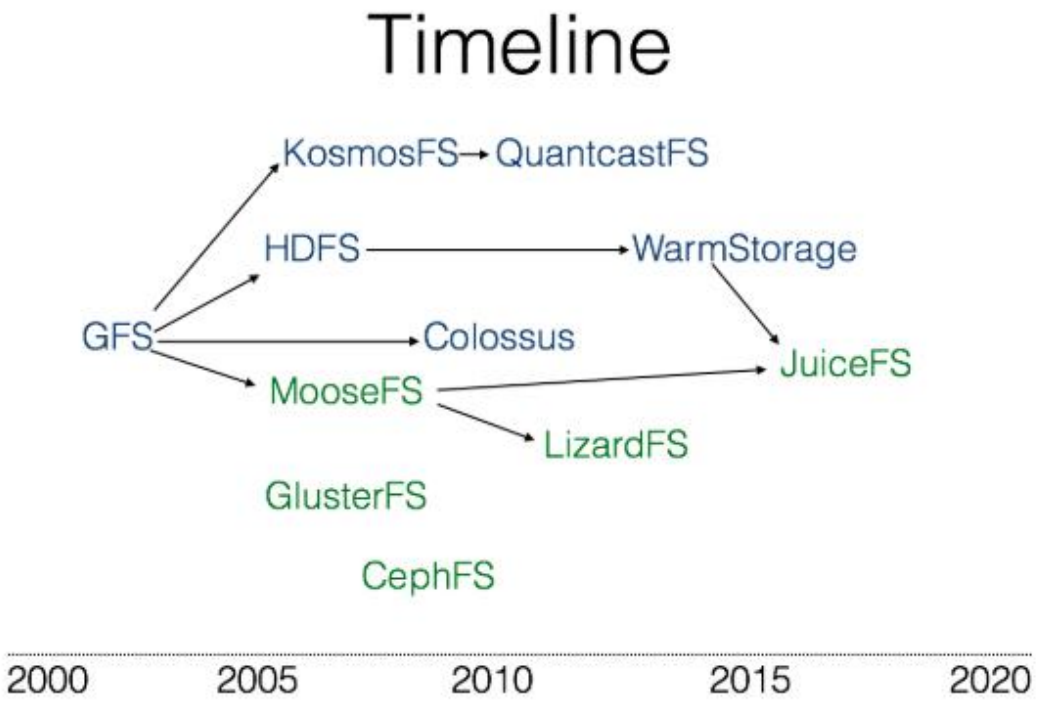
The Google File System

目录

一、摘要（Abstract）

Google File System 是面向大规模数据密集型应用、**可伸缩（scalable）**的*分布式文件系统*。具 **fault tolerance（容灾）**和**高性能（high aggregate performance）特点**。



（注：本笔记不按照绝对顺序）

二、简介(Introduction)

GFS 包含传统分布式系统的设计目标：performance、scalability、reliability and availability。GFS 的设计思路包括下面这些：

第一，（component failures are the norm rather than the exception），因此，GFS 集成了持续监控、错误侦测、*灾难冗余*以及自动恢复的机制。（constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.）*注:我们的系统也需要这些功能，并且认为这些功能是必不可少的。*

第二，GFS 设计时考虑文件大小在 GB 以上。*注：小文件读写性能可能受限。*

第三，数据访问模式：大部分文件修改是尾部追加，一次写完，顺序只读。Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while *caching data* blocks in the *client* loses its appeal. *注：超大的数据集下使用 caching data 的可能性不大？为何不使用缓存机制?*

第四，保证 *N-1:多客户机对同一文件访问*下的数据一致性。*（追加操作）*

三、设计预期（Assumptions）

（1）快速侦测错误、冗余并恢复。

（2）只要针对大文件的存储，小文件存储无专门优化。

（3）系统工作负载：大规模流式读取和小规模随机读取。

（4）系统对小规模的随机位置写入效率不明显。

（5）N-1:多客户端并行追加数据到同一个文件。（最小同步开销同时实现原子的多路追加数据操作）。

（6）稳定的网络带宽。

注：简介和设计预期里面都提到 GFS 设计的背景和目的，从这些侧面都能够知道 GFS 并不能够解决所有问题，但是其提供的基础模型是有用的。我们需要特别留意 GFS 是针对大文件（GB 以上）存储的解决方案，系统使用的是宽松的一致性标准（对覆写操作无一致性优化）。这些一方面是 GFS 简约的原因，另外一方面也是系统的缺陷。后续继续研究其他分布式文件系统时可以刻意去看看有没有解决对应的问题。

四、架构（Architecture）

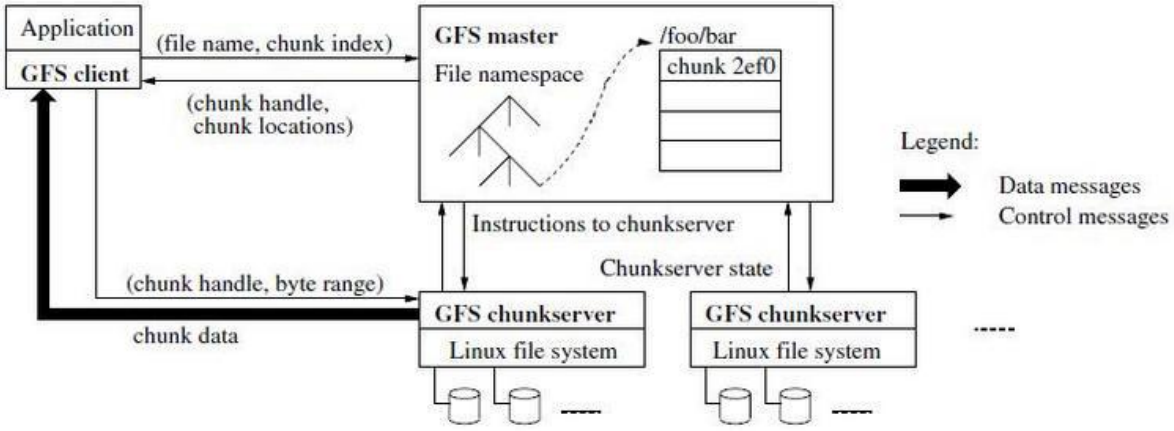


Figure 1: GFS Architecture

GFS 架构有几个组成部分,GFS master、chunserver、client、Application，master 节点管理文件系统的元数据，chunserver 实际存储文件内容，client 节点通常询问文件数据，APP 即为应用层。

client 和 chunserver 通常是在同一台物理主机上的两个独立进程。（*注：通常 chunkserver 与 chunkserver 之间要进行数据同步/复制，涉及到访问数据，这些 chunk 数据交互时，自由定义 client、chunkserver 的逻辑概念。*）

GFS 存储文件时分割成固定大小的 chunk（*注:研究簇大小对于文件存储的影响*），chunk 创建时，Master 服务器给 chunk 分配不变的、全局唯一的 64 位 chunk 标识。chunkserver 将 chunk 以文件形式保存在本地磁盘（*注：研究文件如何保存在磁盘上的*）。

GFS 不适用***缓存机制***，无论 client 还是 chunkserver 都不能存在缓存加速。由于 GFS 设计背景是以流方式读取一个超大文件/超大工作集访问，只要问题在于无法被缓存。（*注：研究超大数据集的缓存加速机制*）但是客户端可以缓存数 TB 级别的工作数据集的 Chunk 位置信息。Chunkserver 不需要缓存文件数据的原因是，Chunk 以本地文件的形式保存，而 Linux 操作系统的文件系统缓存会把经常访问的数据缓存在内存中。（注：研究 Linux 的缓存加速机制）

五、Master 节点（Single Master）

这部分设计策略是使 Master 单节点负荷不至于成为系统的性能瓶颈。Client 过一段时间会向 Master 询问元数据，**然后将元数据存放到本地缓存中**，后续进行数据读写由 Client 对最近的 chunkserver 发起，不需要再经过 Master，从而减轻 Master 的负荷。Client 一次请求可以查询多个 Chunk 信息，由于 Chunk size 体量大（通常 64MB），因此元数据尺寸小，一次查询可以获取的信息量变大（逻辑上这样解释是没问题的）。

Master 节点的工作任务包括：名称空间管理和锁、决定位置、创建和复制 chunk、协调系统操作(读、写、快照)、管理 chunkserver 负载均衡、垃圾回收机制。

（1）Namespace management and locking-命名空间管理和锁机制

GFS 使用细粒度的锁机制，比如/home 被上了 read lock，但是/home/user/bar 没有被锁，意味着/home/user/bar 下的文件可读。

（2）Creation,Re-replication,Rebalancing

新建 chunk，会尽量将 replica 放置于磁盘空间利用率低于平均水平的 chunkserver。

Re-replication 的原因有两个，要么是 replica 故障，要么是动态增加了 replication 的个数。需要限制 GFS 集群同时进行 clone 的操作数量，以减轻 chone 对带宽的消耗。

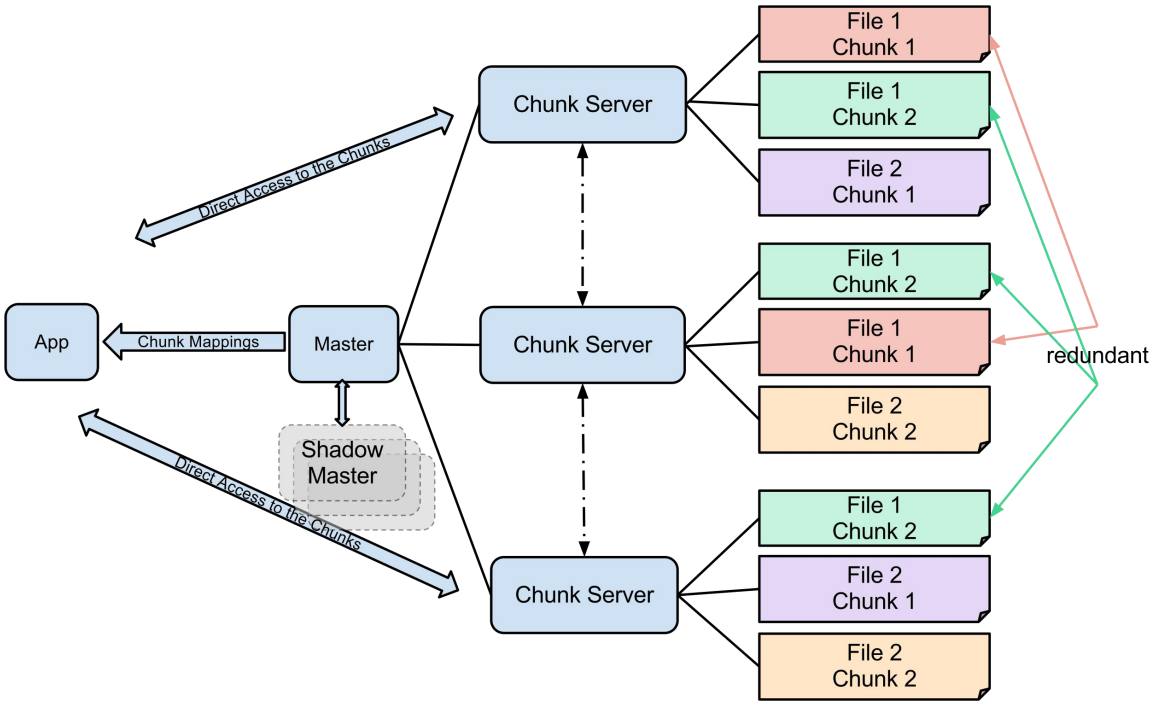
Rebalancing 机制是 master 负责检查当前 replica 分布，将 replica 移动到更好的磁盘位置(通常挑选磁盘空闲水平低于平均值的 chunkserver）。

（3）Garbage Collection-垃圾回收

客户端发起 master 节点删除一个文件后，GFS 不立刻回收文件的物理磁盘空间。GFS 提供 regular 的垃圾收集机制，用于回收 file 和 chunk 级别的物理磁盘空间。这是一种 lazily garbage collection。

Master 会将需要删除的文件重命名为 hidden files，再定期扫描，如果时间戳超过 3 天以上,则进行空间回收。撤销删除也只需要进行恢复命名。chunkserver 上相关的 chunk 删除，通过 heartbeat 机制进行。

六、Chunk 尺寸（Chunk Size）



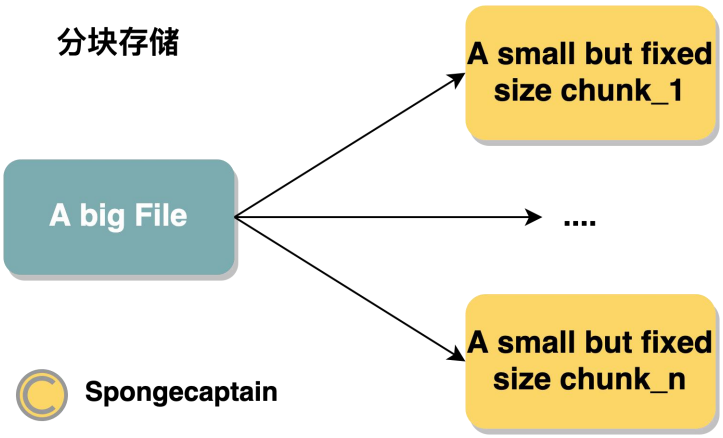
Chunk 以 64MB 为固定的 Chunk Size（默认）。大的 chunk 的优点：

1. 减小 metadata 的尺寸，master 可以存储更多的元数据。

2. 概率上，数据读取有连续读取的倾向，如果 chunk 体积大，下一次读取在当前 offset 后的位置进行读取数据的概率增大。

3. 减少 client 与 master 的交互次数，一次查询可拿到多个 chunk 的位置信息。

缺陷也明显，对于小文件的存储，通常不分块，意味着一个文件对应一个 chunk，当多用户同时对该文件进行数据记录时，将造成热点现象（hot spot）。



GFS 引入文件分块存储，这里说明其特点：

1. Chunk 大小固定，默认 64MB。

2. Chunk1-n 在逻辑上连续，磁盘存储上不一定连续（同一物理磁盘不同地址，同一主机不同磁盘，不同主机磁盘）

注：将数据存储在同一主机不同磁盘上的方式需要研究，WIndows 上有 RAID 机制，那么 Linux 上是否利用同样的方式，又或者在部署 GFS 时，是否已经将不同磁盘的空间统一利用。

3. Chunk 有复制机制，通常有两份副本（replicas）。GFS 默认为每个 chunk 保存两份 replicas。

七、元数据（Metadata）

NUM.	Key	Value	ALG.
1	filename	An array of chunk handler (nv)	HashMap
2	Chunk	√A list	B+Tree

	handler	of chunkserver (v) √Chunk version number(nv) √Which chunkserver is primary node(v) √Lease expiration time(v)	

表格 1. Metadata

Table1. and Table2. 数据都存放在 master 的内存上，nv(non-volatile)数据需要定期存储到 Master 的磁盘上。（数据持久化使用日志+checkpoint）

注：论文中提到相关的 In-Memory Data Structures、Chunk Locations、Operation Log。其中内存数据结构主要提及元数据需要保存在内存中，这样设计的优点是 master 操作速度很快，当然系统也受限于内存大小。Chunk Locations 主要讲了 master 不需要持久化 replicas 的位置信息，而是通过定期轮询进行数

据同步，不需要在 master 上维护全局视图（由于 Master 并不能决定 chunk 是否存在某个硬盘上）。Operation Log 则描述了系统的操作日志，记录元数据变更的历史记录，并且将其作为判断同步操作顺序的逻辑时间基线，在元数据被持久化后，操作日志才会对客户端可见。

同时，Master 服务器在灾难恢复时，需要通过重做 Operation Log 将文件系统进行恢复，为了减轻日志体量和重做的任务量，使用 checkpoint 文件，只需要重做 checkpoint 点往后的日志操作。（对于一个包含数百万个文件的集群，创建一个 Checkpoint 文件需要 1min 左右时间）。

八、一致性模型(Consistency Model)

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

Table 1: File Region State After Mutation

这部分原文写的简略且晦涩，术语一堆，只能够从全局观点了解。文件在写入时会遇到一致性问题，这个容易理解，GFS 将文件修改后所处的状态进行了划分，解释如下：

术语	解释
Write	覆写文件
Record Append	从某个位置进行追加
Serial Success	串行写成功后
Concurrent Success	并行写成功后
Failure	失败（发生写冲突）
defined	所有 client 上看到的数据都是一致的，完全一样的。并且修改的内容是可见的。*（有点类似于按照一定的顺序进行执行，结果一致。再来一次同样写入也一样结果。）*
Consistent	所有 client 上看到的数据都是一致的。
Undefined	所有 client 上看到的数据都是一致的，但是执行次序导致结果未定义，并不能确定修改后是什么结果。
Failed	完全混乱

这里要提一下，按照原文的解释，实在令人难以理解上述术语，defined 的翻译是：一个文件数据修改之后 file region 保持 consistent，也就是所有 client 看到同样的数据，**此外，所有 client 能够看到全部修改（且写入 chunkserver）的内容。**

这是难理解的，按照我的理解，defined 和 undefined 的状态下，所有 client 上看到的数据都是一致的，但是唯一区别在于，defined 状态表示结果是可以逻辑推理出来的唯一状态，而 undefined 并不能够按照逻辑推理，状态不确定。这两个术语是按照***执行次序的层面***进行描述的。

同时，按照这样的逻辑解释，不难理解 consistent but undefined 状态，即看到数据一致，但是不能定义状态，也就是*发生了写冲突，执行次序不确定。*

另外一个令人费解的地方是“defined interpersed with inconsistent”。

也由于这部分术语难以理解，我个人认为完全没必要再去深究。接下来介绍 GFS 如果确保文件修改操作的原子性。GFS 确保被修改的文件是一致的，采用以下措施：

（1）对 Chunk 所有副本修改操作顺序一致；

（2）使用 Chunk 版本号检测副本是否因为所在 chunkserver 宕机导致错过执行而失效。（失效的副本会被 master 回收）

出现的相关问题：

（1）缓存未过期，replica 出现过时问题：client 存在 cache 缘故，在缓存被刷新之前，client 有可能从 stale replica 读取文件数据。重要的一点是，由于 GFS 中的大多数文件都是 append-only，因此 stale replica 上读的数据都是一个 premature end of chunk，并不会读到最新追加的 chunk，这样将影响降到最小。

（2）组件故障问题：master 节点与 chunkserver 进行 regular handshake 检测出现故障的 chunkserver，通过 checksumming（校验和）来检测数据是否损坏。一旦出现问题，GFS 会从有效的 replicas 上进行数据恢复。

九、系统交互

设计 GFS 系统时，有意识地保持一个原则：最小化所有操作和 Master 节点的交互，这节的内容描述了各节点间的交互模式、

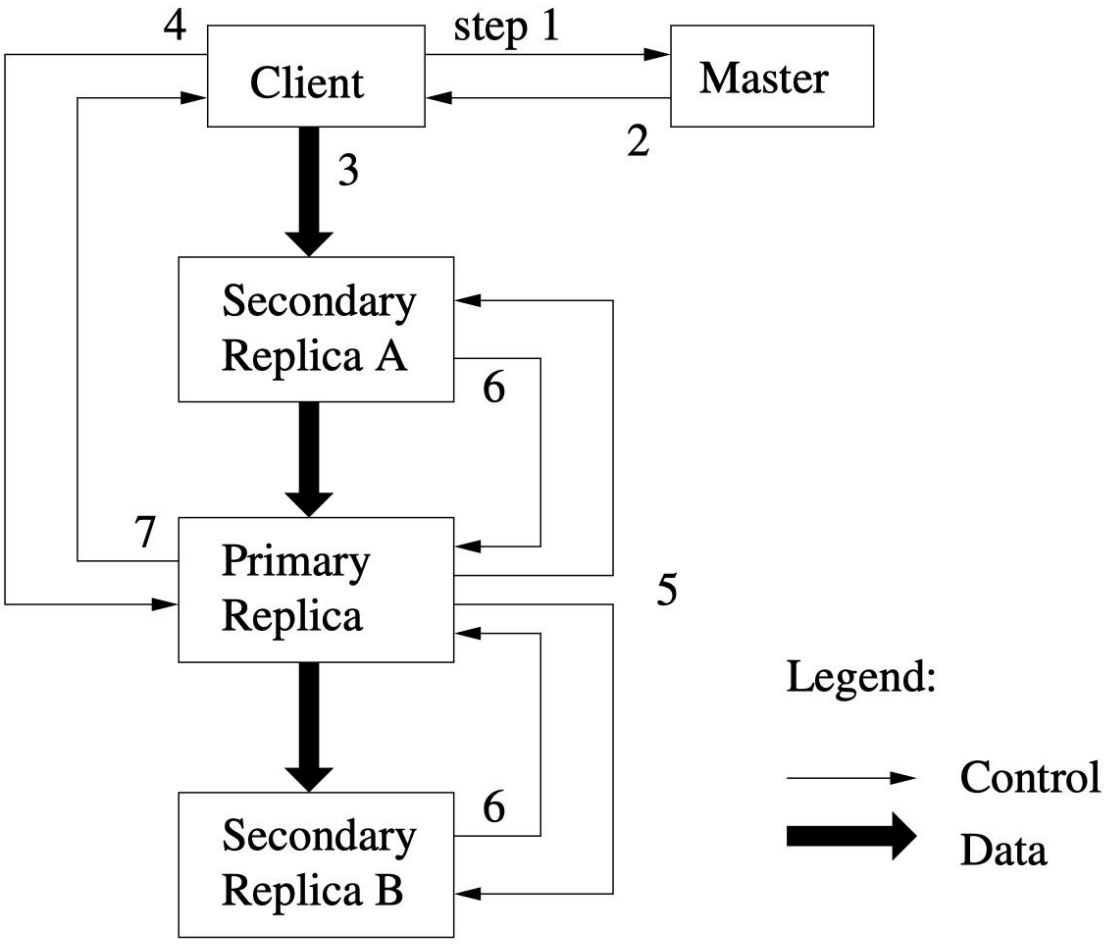


Figure 2: Write Control and Data Flow

如何实现数据修改操作、原子性的记录追加以及快照功能。

（1）Leases and Mutation Order（租约和修改顺序）

变更会改变 chunk 内容或者元数据，比如写入/记录追加。变更操作会在 chunk 的所有副本上执行。使用*租约机制*保持多个副本间变更顺序的一致性。

master 节点为 chunk 的一个副本建立一个租约，将这个副本叫做 primary chunk。primary chunk 对 chunk 的所有更改操作进行序列化，所有的副本遵从这个序列进行更改。

大概的流程步骤如下：

a)client 询问 master 节点当前的租约 chunk 节点（primary chunk）；

b)client 将数据推送到所有的副本上（保存在客户机的*LRU 缓存*中）；

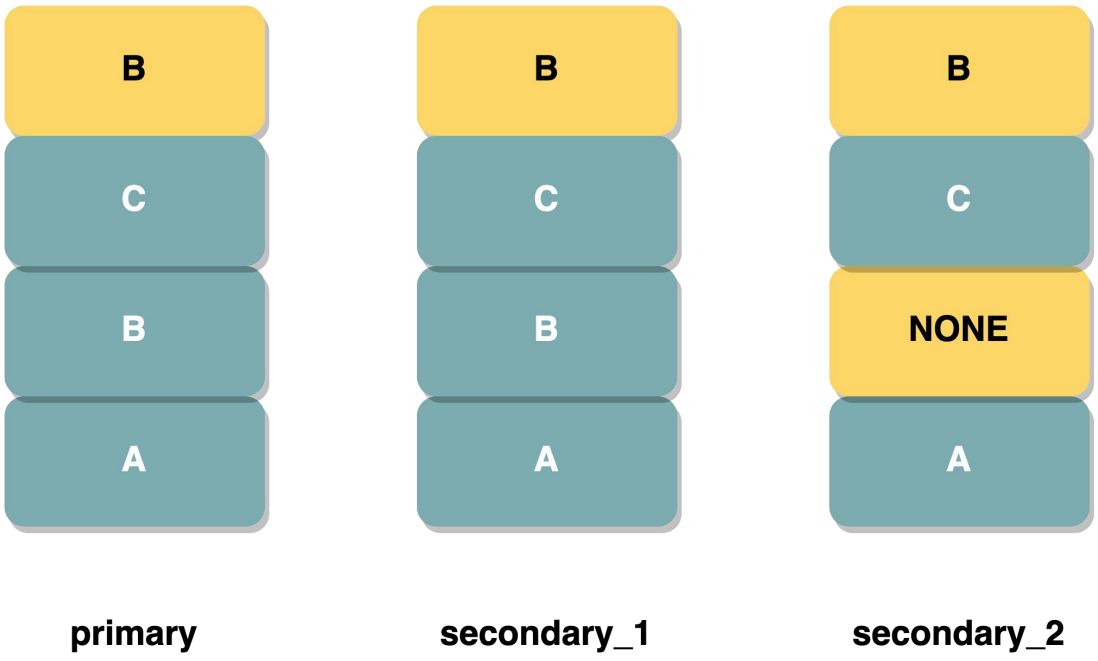
c)所有的副本都确认接收了数据，client 发送写请求到 primary chunk，由 primary chunk 决定每个副本的写入执行顺序。

（2）Data-Flow 数据流

GFS 采用 pipeline 的方式（数据达到一定阈值，传输给下一个节点）。这样由 master 给 primary chunk 传输数据，primary chunk 向下一个 replica 传数据，而下一个 replica 再向下一个 replica 传数据，分散网络负载，充分利用每台机器的带宽。

（3）At Lease Once 机制引发的问题

chunk 上进行 record append 失败后，客户端立刻进行重试（数据至少保证成功写入一次），会出现下面问题：*（注：通过*



校验码进行区别错误）

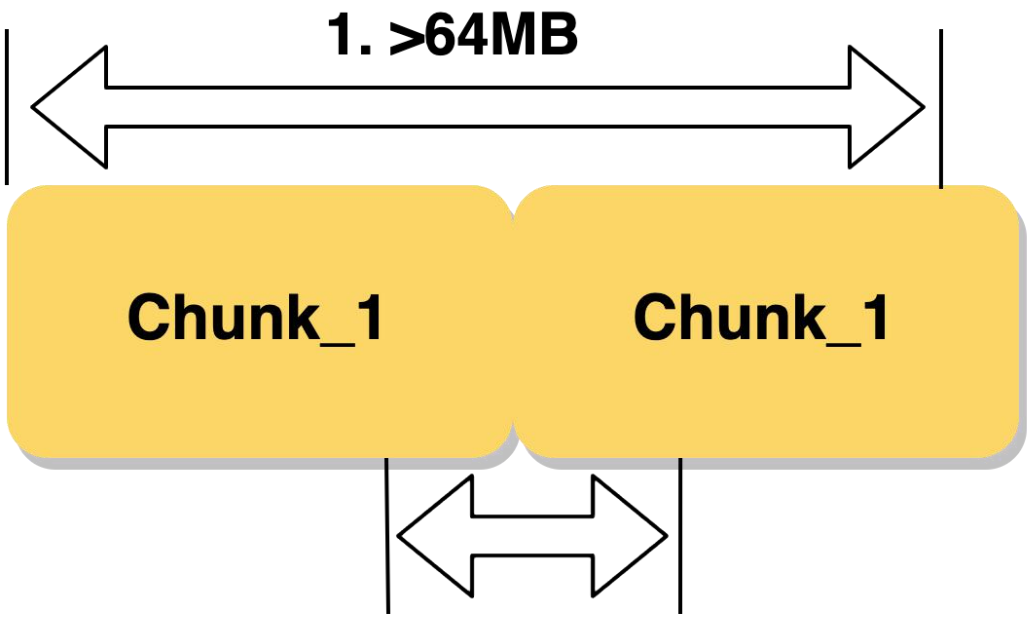
如果记录追加操作在任何一个 replica 上失败了，客户端需要重新操作。结果就是，同一个 chunk 的不同副本包含数据重复（比如上面的 B）。

GFS 并不能保证 Chunk 的所有副本在字节级别的完全一致性，只能够保证数据作为一个整体被至少写入一次。

我们会借助 Readers 机制处理上述出现的偶然性填充数据和重复内容。Writers 在每条写入记录中包含额外信息，例如 chunksum，用以检验有效性（Secondary_2 的 None 可以被判别为无效数据）。

注：对于重复数据应该认定是错误的重复还是正常的重复数据呢?在这方面原文不能让我彻底理解，我推测 GFS 在这个方面存在着漏洞。或者原文不愿意透露更多。

（4）读取操作



客户端并不了解 File 底层被 chunk 存储，所以即使读取数据大小大于数据块 64MB，或者数据位于两个 chunk 中间位置。都可以这样理解：底层还是与两个/多个 chunk 发生了数据请求。

（5）Snapshot-快照

COW(copy on write）机制实现快照，快速且不影响当前操作。多个调用者同时访问相同数据时，只有更新操作，才会复制一份数据并且更新替换，否则都是访问同一个资源。

注：如果对于某个客户端而言，进行了数据写入，并且有新的 chunk 副本产生，后续进行写入操作，再通过拥有这个 chunk 的 chunkserver 进行副本复制，那么之前的数据呢？还有就是其他客户端在这个过程中看到的数据是什么？

COW 在创建快照时，并不会发生物理数据拷贝，只是拷贝原始数据所在源数据块的物理位置元数据（指针信息），新创建的快照文件和源文件指向相同 Chunk 地址。因此，COW 快照创建很快，瞬间完成。

创建快照后，快照监控跟踪原始数据变化（即对源数据块的写操作），一旦源数据块中的原始数据被改写，则将源数据块上的数据块拷贝到新数据块上（新 replica），然后将*新数据写入到源数据块覆盖原始数据。其中所有的源数据块组成源数据卷，新数据块组成快照卷，快照卷只保留发生变换的数据块。*

原文不详细，建议重新研究 Snapshot 技术。可以用于数据回滚。需要注意快照文件大小。

十、过期副本检测(Stale Replica Detection)

通过 master 维护的 chunk version number（版本号）来区分哪些 chunk 是 up-to-date,哪些 chunk 是 stale。

Master 赋予一个 chunk 新的租约时，将 chunk version 自增，分发告知其他 replicas。

十一、容错和诊断（Fault tolerance and diagnosis）

（1）HA-高可用性(high availability)

Fast Recovery: 可以快速重启。

Chunk Replication: 数据冗余（默认 2 份冗余数据）。

Master replication; 提供 shadow master 节点。

（2）Data Integrity-数据完整性

十二、度量（Measurements）

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

对于每一个服务器，无论是 chunkserver 还是 master 节点，都只有几十 MB 大小的 metadata 数据，因此节点从故障到恢复速度很快（30~60sec）。chunkserver 从启动到提供 metadata 数据，只要几秒。

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Table 3: Performance Metrics for Two GFS Clusters

Read and write rates（读写速率测试）

注：写性能严重落后。

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

集群 X，Y。

十三、相关经验

不同于 AFS、xFS、Frangipani 以及 Intermezzo 等文件系统，GFS 在*文件系统层面*没有提供任何 Cache 机制。（在操作系统层面是有的，主要用于元数据）。

某些分布式文件系统是去中心的，比如 Frangipani、xFS、Minnesota’s GFS、GPFS，只依赖分布式算法保持一致性和可管理性。GFS 选择 Master-Client 架构，并且对 Master 减负荷，简化系统架构，并且增加稳定性和可扩展性。

如果我们的目标是，提供一个高速读写的分布式文件系统，支持多客户机进行大规模数据（包括 GB 以上文件、MB 以下文件）的非顺序性操作，通常需要做容灾，也需要支持良好的扩展性。也许 GFS 并不适用。

目前了解看来，GFS 的缺陷明显，首先，它对小文件读写不友好，其次在文件一致性的保障上采用了宽容的策略，甚至对于覆写彻底没有一致性保证。

当然，GFS 设计之初也只是针对其内部应用场景，子模块设计还是有些有趣的地方的，比如对 master 的元数据的持久化（使用了日志+checkpoint）、stale replicas 的回收机制、利用 leases 保证追加记录数据一致性、细粒度的分布式锁机制等等。以及设计所遇到的许多挑战也极其具有代表性，在往后研究里面可以多注意是否解决了对应的瓶颈问题。