ABSTRACT

我们已经设计和实现了 Google File System，一个适用于大规模分布式数据处理相关应用的，可扩展的分布式文件系统。它运行在廉价且普通的硬件设备上，并提供了容错的设计，并且为大量的客户端提供极高的聚合处理性能。

尽管我们的设计目标和上一个版本的分布式文件系统有很多相同的地方，我们的设计是依据我们应用的工作量以及技术环境来设计的，包括现在和预期的，都有一部分和早先的文件系统的约定有所不同。这就要求我们重新审视传统的设计选择，以及探索一些在根本上不同的设计要点。

这个文件系统成功的满足了我们的存储需求。这个文件系统作为那些需要大数据集服务的数据生成处理的基础存储平台而广泛部署在谷歌内部。最大的集群通过上千个计算机的数千个硬盘，提供了数百 TB 的存储，并且这些数据被数百个客户端并行同时操作。

在这个论文里，我们展示了用于支持分布式应用的扩展文件系统接口设计，讨论了许多我们设计的方面，并且列出了我们的 micro-benchmarks 以及真实应用性能指标。

1.INTRODUCTION

为了满足 google 快速增长的数据处理需求，我们设计实现了 google 文件系统(GFS)。GFS 和上一个分布式文件系统有着很多相同的设计目标，比如性能，扩展性，可靠性，以及可用性。然而，它的设计是由我们的具体应用的负载类型以及当前甚至未来技术环境的观察驱动的，所以与早期文件系统的设计假设具有明显的区别。这就要求我们重新审视传统的设计选择，探索出一些在根本上不同的设计观点。

首先，组件失败成为一种常态而不是一种错误(或者说异常)。整个文件系统是由成百上千台廉价的普通机器组成的存储机器，可以被大量的客户端访问。组件的数量和质量在本质上决定了在某一时间有一些是不可用的，并且某些机器无法从当前失败的状态中恢复。我们观察到，应用程序的 bug，操作系统 bug，人为的错误，硬盘的失败，内存，连接器，网络，电力供应都可以引起这样的问题。因此经常性的监控，错误检测，容错和自动恢复必须集成到系统中。

第二，与传统的标准相比，文件是巨大的。在这里，好几个 G 的文件是很普通的。每个文件通常包含很多的应用程序处理的对象比如网页文档。当我们日常处理的快速增长的数据集合总是达到好几个 TB 的大小(包含数十亿的数据)，

即使文件系统能够支持,我们也不希望去管理数十亿个 KB 级别的文件。这样设计中的一些假设和参数，比如 IO 操作和块大小就必须重新定义。

第三，大部分文件都是只会在文件尾新增加数据，而少见修改已有数据的。对一个文件的随机写操作在实际上几乎是不存在的。一旦写完,文件就是只读的，并且一般都是顺序读取的。大量的数据都具有这样的特点。有些数据可能组成很大的数据仓库，并且数据分析程序从头扫描到尾。有些可能是运行应用而不断的产生的数据流。有些是归档的数据。有些是一个机器为另一个机器产生的中间结果，另一个机器及时或者随后处理这些中间结果。假设在大文件上数据访问具有这样的模式，那么当缓存数据在客户端失效后，append 操作就成为性能优化和原子性的关键。

第四，应用程序和文件系统 api 的协同设计，增加了整个系统的灵活性。比如我们通过放松了 GFS 的一致性模型大大简化了文件系统，同时也没有给应用程序带来繁重的负担。我们也提供了一个原子性的 append 操作，这样多个客户端就可以对同一个文件并行的进行 append 操作而不需要彼此间进行额外的同步操作。这些都会在后面进行详细的讨论。

2.DESIGN OVERVIEW

2.1 Assumptions

在设计一个满足我们需求的文件系统时,我们以一些充满了挑战和机遇的假设作为指南，之前我们曾间接的提到过一些关键的点，现在我们把这些假设再详细的列出来。

- 系统是建立在大量廉价的普通计算机上，这些计算机经常故障。必须对这些计算机持续进行检测，并且在运行的系统上进行：检查，容错，以及从快速故障恢复。
- 系统存储了大量的超大文件。我们期望有数百万个文件，每个 100mb 或者更大。上 GB 的文件大小应该是很普通的情况而且能被有效的管理。小文件也应该被支持，但我们不需要为它们进行优化。
- 工作负载主要由两种类型的读取组成：大的流式读取和小的随机读取。在大的流式读取中，单个操作通常要读取数百 k，甚至 1m 或者更大的数据。对于同一个客户端来说，往往会发起连续的读取操作顺序读取一个文件。小的随机读取通常在某个任意的偏移位置读取几 kb 的数据。小规模的随机读取通常在文件的不同位置，读取几 k 数据。对于性能有过特

别考虑的应用通常会作批处理并且对他们读取的内容进行排序，这样可以使得他们的读取始终是单向顺序读取，而不需要往回读取数据。

- 通常基于 GFS 的操作都有很多超大的，例如顺序写入(大的流式读取)的文件操作。通常写入操作的数据量和读取的数据量是相当。一旦完成写入，文件就很少会被再次修改。支持文件中任意位置的小规模写入操作，但是不需要为此作特别的优化。
- 系统对多客户端并行添加同一个文件必须非常有效以及明确语义细节的进行支持。我们的文件经常使用生产者/消费者队列模式，或者作为多路合并模式进行操作。好几百个运行在不同机器上的生产者，将会并行增加一个文件。其本质就是最小的原子操作的定义。读取操作可能接着生产者操作之后进行，消费者会同时读取这个文件。
- 高性能的稳定带宽的网络要比低延时更加重要。我们大多数的目标应用程序都非常重视高速批量处理数据，而很少有人对单个读写操作有严格的响应时间要求。

2.2 Interface

GFS 虽然他没有实现一些标准的 API 比如 POSIX,但它提供了常见的文件系统的接口。文件是通过 pathname 来通过目录进行分层管理的。我们支持的一些常见操作：create,delete,open,close,read，write 等文件操作。

另外，GFS 有 snapshot，record append 等操作。Snapshort(快照)以低成本创建一个文件或者一个目录树的副本。Record append 允许很多个客户端同时对一个文件增加数据，同时保证每一个客户端的添加操作的原子操作性。这个对于多路合并操作和多个客户端同时操作的生产者/消费者队列的实现非常有用，它不用额外的加锁处理。我们发现这种文件对于构造大型分布式应用来说，是不可或缺的。snapshot 和 record append 在后边的 3.4 和 3.3 节有单独讲述。

2.3 Architecture

GFS 集群由一个单个的 master 和多个 chunkserver（块服务器）组成，GFS 集群会有很多客户端 client 访问（图 1）。每一个节点都是一个普通的 Linux 计算机，运行的是一个用户级别（user-level）的服务器进程。只要机器资源允许，并且允许不稳定的应用代码导致的低可靠性，我们就在同一台机器上运行 chunkserver 和 client。

在 GFS 下,每一个文件都拆成固定大小的 chunk(块)。每一个块都由 master 根据块创建的时间产生一个全局唯一的 64 位的 chunk handle 标志。Chunkservers 在本地磁盘上用 Linux 文件系统保存这些 chunk，并且根据 chunk handle 和字节区间，通过 Linux 文件系统读/写这些 chunk 的数据。出于可靠性的考虑，每一个块都会在不同的 chunkserver 上保存备份。默认情况下，我们保存 3 个备份，不过用户对于不同的文件 namespace 区域，可以指定不同的复制级别。

master 负责管理所有的文件系统的元数据(metadata,元数据是指描述数据属性的信息，包括存储位置，历史数据等等)。包括 namespace，访问控制信息，文件到 chunk 的映射关系，当前 chunk 的位置等等信息。master 也同样控制系统级别的活动，比如 chunk 的分配管理，孤点 chunk 的垃圾回收机制，chunkserver 之间的 chunk 镜像管理。master 和这些 chunkserver 之间会有周期性的的心跳检测，并且在检测的过程中向其发出指令并收集其状态。

连接到各个应用系统的 GFS 客户端代码包含了文件系统的 API，并且会和 master 和 chunkserver 进行通讯处理，代表应用程序进行读/写数据的操作。客户端和 master 进行元数据的操作，但是所有的数据相关的通讯是直接和 chunkserver 进行的。我们并没有提供 POSIX API,因此不需要连接到 Linux 的 vnode 层。

客户端或者 chunkserver 都不会缓存文件数据。客户端缓存机制没有什么好处,这是因为大部分的应用都是流式访问超大文件或者操作的数据集太大而不能被缓存。不设计缓存系统使得客户端以及整个系统都大大简化了（不用设计解决缓存的一致性的问题，也就是缓存同步机制）(不过客户端缓存元数据)。chunkserver 不需要缓存文件数据,因为 chunks 已经跟本地文件一样的被保存了，所以 Linux 的 buffer cache 已经把常用的数据缓存到了内存里。

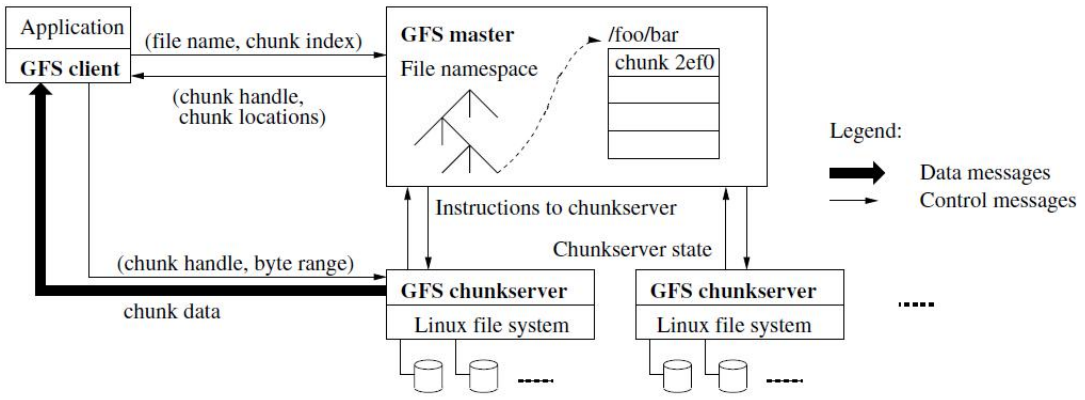


Figure 1: GFS Architecture

2.4 Single Master

引入一个单个 master 的设计可以大大简化我们的设计，并且也让 master 能够基于全局的角度来复杂的 chunk 存放和作出复制的决策。不过，我们必须尽量减少 master 的读和写操作，以避免它成为瓶颈。客户端永远不会通过 master 来做文件的数据读写。客户端只是问 master 它应当访问那一个 chunkserver 来访问数据。客户端在一定时间内缓存这个信息，并且在后续的操作中都直接和 chunkserver 进行操作。

这里我们简单介绍一下图 1 中的读取操作。首先，客户端把应用要读取的文件名和偏移量，根据固定的 chunk 大小，转换成为文件的 chunk index。然后向 master 发送这个包含了文件名和 chunkindex 的请求。master 返回相关的 chunk handle 以及对应的位置。客户端 cache 这些信息，把文件名和 chunkindex 作为 cache 的关键索引字。

于是这个客户端就像对应的位置的 chunkserver 发起请求，通常这个会是离这个客户端最近的一个。请求给定了 chunk handle 以及一个在这个 chunk 内需要读取得字节区间。在这个 chunk 内，再次操作数据将不用再通过客户端-master 的交互，除非这个客户端本身的 cache 信息过期了，或者这个文件重新打开了。实际上，客户端通常都会在请求中附加向 master 询问多个 chunk 的信息，master 于是接着会立刻给这个客户端回应这些 chunk 的信息。这个附加信息是通过几个几乎没有任何代价的 client-master 的交互完成的。

2.5 Chunk Size

chunk 的大小是一个设计的关键参数。我们选择这个大小为 64M，远远大于典型的文件系统的 block 大小。每一个 chunk 的实例（复制品，快照）都是作为在 chunk server 上的 Linux 文件格式存放的，并且只有当需要的情况下才会增长。滞后分配空间(Lazy space allocation)的机制可以通过文件内部分段来避免空间浪费，可能对于这样大的 chunk size 来说，(内部分段 fragment) 这可能是一个最大的缺陷了。

选择一个很大的 chunk 大小提供了一些重要的好处。首先，它减少了客户端和 master 的交互，因为在同一个 chunk 内的读写操作之需要客户端初始询问一次 master 关于 chunk 位置信息就可以了。这个减少访问量对于我们的系统来说是很显著的，因为我们的应用大部分是顺序读写超大文件的。即使是对小范围的随机读，客户端可以很容易 cache 一个好几个 TB 数据文件的所有的位置信息。

其次，由于是使用一个大的 chunk，客户端可以在一个 chunk 上完成更多的操作，它可以通过维持一个到 chunk server 的 TCP 长连接来减少网络管理量(overhead,负载?)。第三，它减少了元数据在 master 上的大小。这个使得我们可以把元数据保存在内存，这样带来一些其他的好处，详细的讨论请见 2.6.1 节。

在另一方面，即时采用了 lazy space allocation 的大 chunk size 也有它的不好的地方。小文件可能仅由一些 chunk 组成，也许只有一个 chunk。如果很多的 client 都需要访问这个文件，这样那些存储了这些 chunk 的 chunkserver 就会变成热点。实际中，热点还没有成为一个主要考虑的问题因为我们的应用绝大部分都是在顺序读取多个大型 chunk 文件(large multi-chunck files)。

然而，当 GFS 第一次使用在一个批处理队列系统时，热点确实出现了：一个可执行文件作为一个 chunk 的文件写到 GFS，然后同时在数百台机器上开始执行。只有两三个 chunkservers 存储这些可执行文件，而这些 chunkserver 被数百个并发请求瞬间变成超载。我们通过更高的备份级别存储这样的可执行文件(多存几份?)以及错开队列系统的应用程序启动时间解决了这个问题。一个潜在的长远的解决方案是在这种情况下，允许客户端从其他客户端读取数据。

2.6 Metadata

master 节点保存这样三个主要类型的元数据:①file 和 chunk namespace，②从 files 到 chunks 的映射关系③每一个 chunk 的副本的位置。所有的元数据都是保存在 master 的内存里的。前两个类型(namep spaces 和文件到 chunk 的映射)还通过将更新操作的日志保存在本地硬盘，并且日志也会在远端机器上保存副本。使用 log 允许我们简单可靠地更新 master 的状态，不用担心当 master crash 的时候的不一致性。master 并不持久化保存 chunk 位置信息。相反，他在启动地时候以及 chunkserver 加入集群的时候，向每一个 chunkserver 询问他的 chunk 信息。

2.6.1 In-Memory Data Structures

因为元数据都是在内存保存的,master 的操作很快。另外 master 也很容易，有效地定时在后台扫描所有的内部状态。这个周期性的扫描是用来实现 chunk 垃圾回收，chunkserver 出现失败时进行的重复制，以及为了平衡负载和磁盘空间在 chunkserver 间的 chunk 迁移。4.3 4.4 将进一步讨论这些活动。

这种内存保存数据的方式有一个潜在的问题,就是说整个系统的 chunk 数量以及对应的系统容量是受到 master 机器的内存限制的。这个在实际生产中并不是一个很严重的限制。master 为每 64Mchunk 分配的空间不到 64 个字节的元数据。大部分的 chunks 是满的，因为大部分文件都是很大的，包含很多个 chunk，只有文件的最后部分可能是未满的。类似的，每个文件名字空间数据通常需要少于 64byte 因为文件名称存储时会使用前缀压缩算法进行压缩。

如果有需要支持到更大的文件系统，因为我们是采用内存保存元数据的方式，所以我们可以很简单，可靠，高效，灵活的通过增加 master 机器的内存就可以了。

2.6.2 Chunk Locations

master 并不持久化保存 chunkserver 上保存的 chunk 的记录。它只是在启动的时候简单的从 chunkserver 取得这些信息。master 可以在启动之后一直保持自己的这些信息是最新的，因为它控制所有的 chunk 的位置，并且使用普通心跳检测监视 chunkserver 的状态。

我们最开始尝试想把 chunk 位置信息持久化保存在 master 上，但是我们后来发现如果在启动时候，以及定期性从 chunkserver 上读取 chunk 位置信息会使得设计简化很多。因为这样可以消除 master 和 chunkserver 之间进行 chunk 信息的同步问题，当 chunkserver 加入和离开集群，更改名字，失效，重新启动等等时候，如果 master 上要求保存 chunk 信息，那么就会存在信息同步的问题。在一个数百台机器的组成的集群中，这样的发生 chunserver 的变动实在是太平常了。

此外，不在 master 上保存 chunk 位置信息的一个重要原因是因为只有 chunkserver 对于 chunk 到底在不在自己机器上有着最后的话语权。另外，在 master 上保存这个信息也是没有必要的,因为有很多原因可以导致 chunserver 可能忽然就丢失了这个 chunk（比如磁盘坏掉了等等），或者 chunkserver 忽然改了名字，那么 master 上保存这个资料啥用处也没有。

2.6.3 Operation Log

操作日志保存了关键元数据变化的历史记录。它是 GFS 的核心。不仅仅因为这是唯一持久化的元数据记录，而且也是因为操作日志也是作为逻辑时间基线，定义了并行操作的顺序。chunks 以及 Files，连同他们的版本（参见 4.5 节），

都是用他们创建时刻的逻辑时间基线来作为唯一的标志。

由于操作日志是极其关键的，我们必须可靠保存它，在元数据改变并且持久化之前，对于客户端来说都是不可见的（也就是说保证原子性）。否则，就算是chunkserver 完好的情况下，我们也可能会丢失整个文件系统，或者最近的客户端操作。因此，我们把这个文件保存在多个远程主机上，并且只有当刷新这个相关的操作日志到本地和远程磁盘之后，才会给客户端操作应答。master 可以在刷新之前将多个操作日志批量处理，以减少刷新和复制这个日志导致的系统吞吐量。

master 通过反演操作日志来回复自身文件系统状态。为了减少启动时间，我们必须保证操作日志的文件尽可能的小。master 在日志增长超过某一个大小的时候，执行 checkpoint 动作，卸出自己的状态，这样可以使下次启动的时候从本地硬盘读出这个最新的 checkpoint，然后反演有限记录数。checkpoint 是一个类似 B-树的格式，可以直接映射到内存，而不需要额外的分析。这更进一步加快了恢复的速度，提高了可用性。

因为建立一个 checkpoint 可能会花一点时间，于是我们这样设定 master 的内部状态，就是说新建立的 checkpoint 可以不阻塞新的状态变化。master 切换到一个新的 log 文件，并且在一个独立的线程中创建新的 checkpoint。新的 checkpoint 包含了在切换到新 log 文件之前的状态变化。当这个集群有数百万文件的时候，创建新的 checkpoint 会花上几分钟的时间。当 checkpoint 建立完毕，会写到本地和远程的磁盘。

对于 master 的恢复，只需要最新的 checkpoint 以及后续的 log 文件。旧的 checkpoint 及其 log 文件可以删掉了，虽然我们还是保存几个 checkpoint 以及 log，用来防止比较大的故障产生。在 checkpoint 的时候得故障并不会导致正确性受到影响，因为恢复的代码会检查并且跳过不完整的 checkpoint。

2.7 Consistency Model

GFS 是一个松散的一致性检查的模型，通过简单高效的实现，来支持我们的高度分布式计算的应用。我们在这里讨论的 GFS 的可靠性以及对应用的可靠性。我们也强调了 GFS 如何达到这些可靠性，实现细节在本论文的其他部分实现。

2.7.1 Guarantees by GFS

文件名字空间的改变（比如，文件的创建）是原子操作。他们是由 master 来专门处理的。名字空间的锁定保证了操作的原子性以及正确性（4.1 节）；

master 的操作日志定义了这些操作的全局顺序(2.6.3)。

当数据变更后，文件区域(文件区就是在文件中的一小块内容)的状态取决于变更的类型，变更是否成功以及是否是并发进行的。表 1 是对结果的一个概述。

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

Table 1: File Region State After Mutation

如果所有的客户端都看到的是相同的数据的时候，并且与这些客户端从哪个数据的副本读取无关的时候，那我们就称这个 file region 具有一致性的。当数据变更前后具有一致性，同时客户端将会看到完整的变更，我们就称该文件区已定义。当一个更改操作成功完成，没有并发写冲突，那么受影响的 region 就是已定义的了（肯定是一致性）：所有客户端都可以看到这个变化是什么。并发成功操作会使 region 的状态进入未定义，但还是一致性的：所有客户端都看到了相同的数据，但它可能无法看到所有的变化(无法区分到底发生了什么变化，如果变更是针对相同的数据写这样有的变更就会被新的变更所覆盖，这样用户就无法看到最先的变更了，同时发生在跨 chunk 的操作会被拆分成两个操作，这样这个操作的一部分可能会被其他操作覆盖，而另一部分则保留下来，如 3.1 节末尾所述)。通常它看到的是多个变更组合后的结果。一个失败的变更会使区域进入非一致的状态(因此也是未定义的状态):不同的客户端在不同的访问中可能看到不同的数据。我们下面描述下我们的应用程序如何区分定义良好的区域和未定义的区域。应用程序不需要进一步区分未定义区域的各种不同的类型。

数据变更可能是 *write* 或者 *record append*。写操作会使数据在应用程序指定的偏移位置写入。record append 操作会使数据原子性的 append，如果是并发性的话则至少会被 append 一次，但是偏移位置是由 GFS 决定的(然而，通常的理解可能是在客户端想写入的那个文件的尾部)。偏移位置会被返回给客户端，同时标记包含这条记录的那个定义良好的文件区域的起始位置。另外 GFS 可能会在它们之间插入一些 padding 或者记录的副本。它们会占据那些被认为是不一致的区域，通常它们比用户数据小的多。

在一系列成功的变更之后，变更的文件区域被保证是已定义的，同时包含了最后一次变更的数据写入。GFS 通过两种方式来实现这种结果:(a).将这些变更以相同的操作顺序应用在该 chunk 的所有的副本上(3.1 小节);(b).使用 chunk

的版本号来检测那些老旧的副本可能是由于它的 chunkserver 挂掉了而丢失了一些变更。陈旧的副本永远都不会参与变更或者返回给那些向 master 询问 chunk 位置的 client。它们会优先参与垃圾回收。

因为客户端会缓存 chunk 的位置，在信息更新之前它们可能会读到陈旧的副本。时间窗口由缓存值的超时时间以及文件的下一次打开而限制，文件的打开会清除缓存中该文件相关的 chunk 信息。此外，由于我们的大部分操作都是 append，因此一个陈旧副本通常会返回一个过早结束的 chunk 而不是过时的数据。当读取者重试并与 master 联系时，它会立即得到当前的 chunk 位置。

成功的变更很久之后，组件失败仍有可能破坏或者污染数据。GFS 通过周期性的在 master 和所有 chunkserver 间握手找到那些失败的 chunkserver，同时通过校验和(5.2 节)来检测数据的污染。一旦发现问题，会尽快的利用正确的副本恢复(4.3 节)。只有一个块的所有副本在 GFS 做出反应之前，全部丢失，这个块才会不可逆转的丢失，而通常 GFS 的反应是在几分钟内的。即使在这种情况下，块不可用，而不是被污染：应用程序会收到清晰的错误信息而不是被污染的数据。

2.7.2 Implications for Application

GFS 应用程序可以通过使用简单的技术来适应这种松散的一致性模型，这些技术已经为其他目的所需要：依赖与 append 操作而不是覆盖，检查点，写时自我验证，自己标识记录。

实际中，我们所有的应用程序都是通过 append 而不是覆盖来改变文件。在一个普通的应用中，程序员生成一个文件都是从头到尾直接生成的。当写完所有数据后它自动的将文件重命名为一个永久性的名称，或者通过周期性的检查点检查已经有多少数据被成功写入了。检查点可能会设置应用级的 checksum。读取者仅验证和处理最后一个检查点之前的文件区域，这些区域处于已定义的状态。无论什么样的并发和一致性要求，这个方法都工作的很好。Append 操作比随机写对于应用程序的失败处理起来总是要更加有效和富有弹性。Checkpoint 使得写操作者增量的进行写操作并且防止读操作者处理已经成功写入，但是对于应用程序角度看来并未提交的数据。

另一种常见的应用中，很多写操作同时向一个文件 append 为了归并文件或者是作为一个生产者消费者队列。记录的 append 的 append-at-least-once 语义预服务每个写者的输出。Reader 对偶然的空白填充(padding)和重复数据的处理如下：writer 为每条记录准备一些额外信息，比如 checksums，这样它

的合法性就可以验证。Reader 可以识别和丢弃额外的 padding，并使用 checksum 记录片段。如果不能容忍重复的数据(比如它们可能触发非幂等操作)，可以通过在记录中使用唯一标识符来过滤它们，很多时候都需要这些标识符命名相应的应用程序实体，比如网页文档。这些用于 record 输入输出的功能函数(除了重复删除)是以库的形式被我们的应用程序共享的，同时应用于 google 其他的文件接口实现。所以，相同系列的记录，加上一些罕见的重复，总是直接被分发给记录的 Reader。

3.System interactions

我们设计的一个原则是尽量在所有操作中减少与 master 的交互。基于该条件下我们现在阐述 client，master 以及 chunkserver 如何通过交互来实现数据变更，记录 append 以及快照。

3.1 Leases and Mutation Order

租约和变更顺序?令牌和变化顺序?

一个变更是指一个改变 chunk 的内容或者 metadata 的操作，比如写操作或者 append 操作。每个变更都需要在所有 chunk 的副本上执行。我们使用租约来保持多个副本间变更顺序的一致性。Master 授权给其中的一个副本一个该 chunk 的租约，我们把它叫做主副本(*primary*)。这个 primary 对所有对 chunk 更改进行序列化。然后所有的副本根据这个顺序执行变更。因此，全局的变更顺序首先是由 master 选择的租约授权顺序来确定的(可能有多个 chunk 需要进行修改)，而同一个租约内的变更顺序则是由那个主副本来定义的。

租约机制是为了最小化 master 的管理开销而设计的。一个租约有一个初始化为 60s 的超时时间设置。然而只要这个 chunk 正在变更，那个主副本就可以向 master 请求延长租约。这些请求和授权通常是与 master 和 chunkserver 间的心跳信息一起发送的。有时候 master 可能想在租约过期前撤销它(比如，master 可能想使对一个正在重命名的文件的变更无效)。即使 master 无法与主副本进行通信，它也可以在旧的租约过期后安全的将租约授权给另一个新的副本。

如图 2，我们将用如下的数字标识的步骤来表示一个写操作的控制流程。

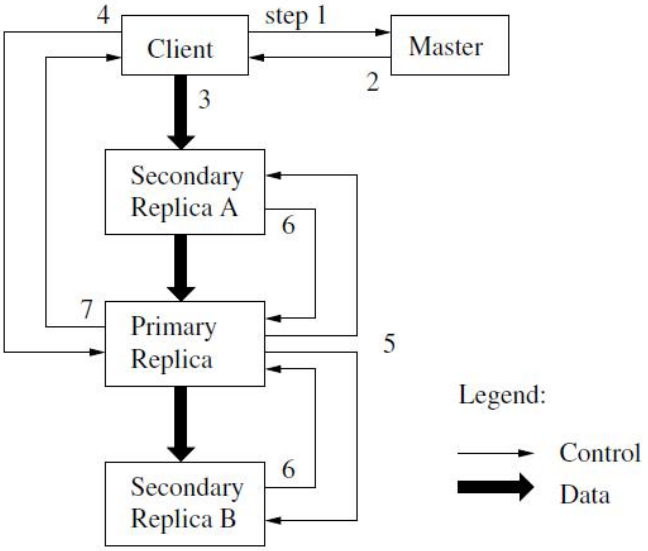


Figure 2: Write Control and Data Flow

1.client 向 master 询问那个 chunkserver 获取了当前 chunk 的租约以及其他副本所在的位置。如果没有人得到租约，master 将租约授权给它选择的一个副本。

2.master 返回该主副本的标识符以及其他副本的位置。Client 为未来的变更缓存这个数据。只有当主副本没有响应或者租约到期时它才需要与 master 联系。

3.client 将数据推送给所有的副本，client 可以以任意的顺序进行推送。每个 chunkserver 会将数据存放在内部的 LRU buffer cache 里，直到数据被使用或者过期。通过将控制流与数据流分离，我们可以通过将昂贵的数据流基于网络拓扑进行调度来提高性能，而不用考虑哪个 chunkserver 是主副本。3.2 节更深入地讨论了这点。

4.一旦所有的副本应答接收到了数据，client 发送一个写请求给主副本，这个请求标识了先前推送给所有副本的数据。主副本会给它收到的所有变更(可能来自多个 client)安排一个连续的序列号来进行必需的串行化。它将这些变更根据序列号应用在本地副本上。

5.主副本将写请求发送给所有的次副本,每个次副本以与主副本相同的串行化顺序应用这些变更。

6.所有的次副本完成操作后向主副本返回应答

7.主副本向 client 返回应答。任何副本碰到的错误都会返回给 client。出现错误时，该写操作可能已经在主副本以及一部分次副本上执行成功。(如果主副本失败，那么它不会安排一个序列号并且发送给其他人)。客户端请求将会被认为是失败的，被修改的区域将会处在非一致状态下。我们的客户端代码会通

过重试变更来处理这样的错误。它会首先在 3-7 步骤间进行一些尝试后在重新从头重试这个写操作。

如果应用程序的一个写操作很大或者跨越了 chunk 的边界，GFS client 代码会将它转化为多个写操作。它们都会遵循上面的控制流程，但是可能会被来自其他 client 的操作插入或者覆盖。因此共享的文件区域可能会包含来自不同 client 的片段,虽然这些副本是一致的(因为所有的操作都按照相同的顺序在所有副本上执行成功了),但是文件区域会处在一种一致但是未定义的状态，正如 2.7 节描述的那样。

3.2 Data Flow

为了更有效的使用网络我们将数据流和控制流解耦。控制流从 client 到达主副本，然后到达其他的所有次副本，而数据则是线性地通过一个精心选择的 chunkserver 链，某种程度上像是管道流水线那样推送过去的。我们的目标是充分利用每个机器的网络带宽，避免网络瓶颈和高延时链路，并且最小化数据推送的延时(最小化同步数据的时间)。

为了充分利用每个机器的网络带宽，数据通过 chunkserver 链线性的推送过去,而不是以其他的拓扑结构进行推送(例如树形)。因此每个机器的带宽都是用于尽可能快地传送数据，而不是在多个接收者之间进行分配。

为了尽可能的避免网络瓶颈和高延时链路(比如 inter-switch 连接通常既是瓶颈延时也高)，每个机器向网络中还没有收到该数据的最近的那个机器推送数据。假设 client 将数据推送给 S1- S4，它会首先将数据推送给最近的 chunkserver 假设是 S1，S1 推送给最近的，假设 S2，S2 推送给 S3，S4 中离他最近的那个。我们网络拓扑足够简单，以至于距离可以通过 IP 地址估计出来。

最后为了最小化延时，我们通过将 TCP 数据传输进行流水化。一旦一个 chunkserver 收到数据，它就开始立即往下发送数据。流水线对我们来说尤其有用，因为我们使用了一个全双工链路的交换网络。立即发送数据并不会降低数据接受速率。如果没有网络拥塞，向 R 个副本传输 B 字节的数据理想的时间耗费是 B/T+RL,T 代表网络吞吐率，L 是机器间的网络延时。我们的网络连接是 100Mbps(T),L 远远低于 1ms，因此 1MB 的数据理想情况下需要 80ms 就可以完成。

3.3 Atomic Record Appends

GFS 提供一个原子性的 append 操作叫做 *record append*(注意这与传统的

append 操作也是不同的)。在传统的写操作中,用户指定数据需要写的偏移位置。对于相同区域的并行写操作是不可串行的：该区域的末尾可能包含来自多个 client 的数据片段。但在一个 record append 操作中，client 唯一需要说明的只有数据。GFS 会将它至少原子性地 append 到文件中一次，append 的位置是由 GFS 选定的，同时会将这个位置返回给 client。这很类似于 unix 文件打开模式中的 O_APPEND，当多个写者并发操作时不会产生竞争条件。

Record append 在我们的分布式应用中被大量的使用。在我们的应用中很多在不同机器的 client 并发地向同一个文件 append。如果使用传统的写操作，client 将需要进行复杂而又昂贵的同步化操作,比如通过一个分布式锁管理器。在我们的工作负载中，这样的文件通常作为一个多生产者/单消费者队列或者用来保存来自多个不同 client 的归并结果。

Record append 是一种类型的变更操作，除了一点在主副本上的额外的逻辑外依然遵循 3.1 节的控制流。Client 将所有的数据推送给所有副本后，它向主副本发送请求。主副本检查将该记录 append 到该 chunk 后是否会导致该 chunk 超过它的最大值(64MB)。如果超过了，它就将该 chunk 填充到最大值，告诉所有的次副本做同样的工作，然后告诉客户端该操作应该在下一个 chunk 上重试。(append 的 Record 大小需要控制在最大 chunk 大小的四分之一以内，这样可以保证最坏情况下的碎片可以保持在一个可以接受的水平上)。如果 record 没有超过最大尺寸，就按照普通情况处理，主副本将数据 append 到它的副本上，告诉次副本将数据写在相同的偏移位置上，最后向 client 返回成功应答。

如果 record append 在任何一个副本上失败，client 就会重试这个操作。这样，相同 chunk 的多个副本就可能包含不同的数据，这些数据可能包含了相同记录的整个或者部分的重复值。GFS 并不保证所有的副本在 byte 级别上的一致性，它只保证数据作为一个原子单元最少写入一次。这个属性是由如下的简单的观察中得出，当操作报告成功时，数据必须写在所有副本的相同 chunk 的相同偏移量写入。此外，所有的副本都必须起码和纪录结束点等长，并且因此即使另外一个副本成了主副本(primary),所有后续的纪录都会被分配在一个较高的偏移量或者在另外一个 chunk 中。在我们的一致性保证里，record append 操作成功后写下的数据区域是已定义的(肯定是一致的),然而介于其间的数据则是不一致的(因此也是未定义的)。我们的应用程序可以处理这样的不一致区域，正如我们在 2.7.2 里讨论的那样。

3.4 Snapshot

快照操作在尽量不影响正在执行的变更操作的情况下，几乎即时产生一个文件或者目录树（“源”）。用户经常用它来创建大数据集的分支拷贝（经常还有拷贝的拷贝，递归拷贝），或者在提交变动前做一个当前状态的 checkpoint，这样可以使得接下来的 commit 或者回滚容易一点。

像 AFS，我们使用标准的 copy-on-writer 技术来实现快照。当 master 收到一个快照请求时，它首先撤销将要进行快照的那些文件对应的 chunk 的所有已发出的租约。这就使得对于这些 chunk 的后续写操作需要与 master 交互来得到租约持有者。这就首先给 master 一个机会创建该 chunk 的新的拷贝。

当这些租约被撤销或者过期后，master 将这些操作以日志形式写入磁盘。然后复制该文件或者目录树的元数据，然后将这些日志记录应用到内存中的复制后的状态上，新创建的快照文件与源文件一样指向相同的 chunk。

当 client 在快照生效后第一次对一个 chunk C 进行写入时，它会发送请求给 master 找到当前租约拥有者。Master 注意到对于 chunk C 的引用计数大于 1。它延迟回复客户端的请求，选择一个新的 chunk handle C`。然后让每个拥有 C 的那些 chunkserver 创建一个新的叫做 C`的 chunk。通过在相同的 chunkserver 上根据原始的 chunk 创建新 chunk，就保证了数据拷贝是本地，而不是通过网络(我们的硬盘比 100Mbps 网络快大概三倍)。这样，对于任何 chunk 的请求处理都没有什么不同：master 为新才 chunk C`的副本中的一个授权租约，然后返回给 client，这样它就可以正常的写这个 chunk 了，client 不需要知道该 chunk 实际上是从一个现有的 chunk 创建出来的。

4.Master Operation

Master 执行所有的 namespace 操作。此外，它还管理整个系统的 chunk 备份：决定如何放置，创建新的 chunk 和相应的副本，协调整个系统的活动保证 chunk 都是完整备份的，在 chunkserver 间进行负载平衡，回收没有使用的存储空间。我们现在讨论这些主题。

4.1 Namespace Management and Locking

很多 master 操作都需要花费很长时间：比如，一个快照操作要撤销该快照所包含的 chunk 的所有租约。我们并不想耽误其他运行中的 master 操作，因此

我们允许多个操作处于活动状态并通过在 namespace 区域使用锁来保证正确的串行化。

不像传统的文件系统，GFS 的目录并没有一种数据结构用来列出该目录下所有文件,而且也不支持文件或者目录别名(像 unix 的硬链接或者软连接那样)。GFS 在逻辑上通过一个路径全称到元数据映射的查找表来表示它的名字空间。通过采用前缀压缩，这个表可以有效地在内存中表示。namespace 树中的每个节点(要么是文件的绝对路径名称要么是目录的)具有一个相关联的读写锁。

每个 master 操作在它运行前，需要获得一个锁的集合。比如,如果它想操作/d1/d2…/dn/leaf，那么它需要获得/d1,/d1/d2……/d1/d2…/dn 这些目录的读锁，然后才能得到路径/d1/d2…/dn/leaf 的读锁或者写锁。注意 Leaf 可能是个文件或者目录，这取决于具体的操作。

我们现在解释一下，当为/home/user 创建快照/save/user 时，锁机制如何防止文件/home/user/foo 被创建。快照操作需要获得在/home 和/save 上的读锁，以及/home/user 和/save/user 上的写锁。文件创建需要获得在/home 和/home/user 上的读锁，以及在/home/user/foo 上的写锁。这两个操作将会被正确的串行化，因为它们试图获取在/home/user 上的相冲突的锁。文件创建并不需要父目录的写锁，因为实际上这里并没有”目录”或者说是类似于 inode 的数据结构，需要防止被修改。读锁已经足够用来防止父目录被删除。

这种锁模式的一个好处就是它允许对相同目录的并发变更操作。比如多个文件的创建可以在相同目录下并发创建：每个获得该目录的一个读锁，以及文件的一个写锁。目录名称上的读锁足够可以防止目录被删除，重命名或者快照。文件名称上的写锁将会保证重复创建相同名称的文件的操作只会被执行一次。

因为 namespace 有很多节点,所以读写锁对象只有在需要时才会被分配,(懒加载)一旦不再使用用就删除。为了避免死锁，锁是按照一个一致的全序关系进行获取的：首先根据所处的 namespace 树的级别，相同级别的则根据字典序。

4.2 Replica Placement

副本位置。GFS 集群是高度分布在多个层次上的。它拥有数百个散步在多个机柜中的 chunkserver。这些 chunkserver 又可以被来自不同或者相同机柜上的 client 访问。处在不同机柜的机器间的通信可能需要穿过一个或者更多的网络交换机。此外，进出一个机柜的带宽可能会小于机柜内所有机器的带宽总和。多级的分布式带来了数据分布式时的扩展性，可靠性和可用性方面的挑战。

Chunk 的备份放置策略服务于两个目的：最大化数据可靠性和可用性，最小化网络带宽的使用。为了达到这两个目的,仅仅将备份放在不同的机器是不够的,

这只能应对机器或者硬盘失败，以及最大化利用每台机器的带宽。我们必须在机柜间存放备份。这样能够保证当一个机柜整个损坏或者离线(比如网络交换机故障或者电路出问题)时，该 chunk 的存放在其他机柜的某些副本仍然是可用的。这也意味着对于一个 chunk 的流量，尤其是读取操作可以充分利用多个机柜的带宽。另一方面，写操作需要在多个机柜间进行，这是我们权衡之后认为可以接受的。

4.3 Creation,Re-replication,Rebalancing

Chunk 副本的创建主要有三个原因：chunk 的创建，重备份，重平衡。

当 master 创建一个 chunk 时，它会选择放置初始化空白副本的位置。它会考虑几个因素：1.尽量把新的 chunk 放在那些低于平均磁盘空间使用值的那些 chunkserver 上。随着时间的推移，这会使得 chunkserver 的磁盘使用趋于相同 2.尽量限制每个 chunkserver 上的最近的文件创建数，虽然创建操作是很简单的，但是它后面往往跟着繁重的写操作，因为 chunk 的创建通常是因为写者的需要而创建它。在我们的一次 append 多次读的工作负载类型中，一旦写入完成，它们就会变成只读的。3.正如前面讨论的，我们希望在机柜间存放 chunk 的副本。

当 chunk 的可用备份数低于用户设定的目标值时，Master 会进行重复制。有多个可能的原因导致它的发生：chunkserver 不可用，chunkserver 报告它的某个备份已被污染，一块硬盘由于错误而不可用或者用户设定的目标值变大了。需要重复制的 chunk 根据几个因素确定优先级。一个因素是它与备份数的目标值差了多少，比如我们给那些丢失了 2 个副本的 chunk 比丢失了 1 个的更高的优先级。另外，比起最近被删除的文件的 chunk，我们更想备份那些仍然存在的文件的 chunk(参考 4.4 节)。最后，为了最小化失败对于运行中的应用程序的影响，我们提高那些阻塞了用户进度的 chunk 的优先级。

Master 选择最高优先级的 chunk,通过给某个 chunkserver 发送指令告诉它直接从一个现有合法部分中拷贝数据来进行克隆。新备份的放置与创建具有类似的目标：平均磁盘使用，限制在单个 chunkserver 上进行的 clone 操作数，使副本存放在不同机柜间。为了防止 clone 的流量淹没 client 的流量，master 限制整个集群已经每个 chunkserver 上处在活动状态的 clone 操作数。另外每个 chunkserver 还会限制它用在 clone 操作上的带宽，通过控制它对源 chunkserver 的读请求。

最后，master 会周期性的对副本进行重平衡。它检查当前的副本分布，然后为了更好的磁盘空间使用和负载均衡，将副本进行移动。而且在这个过程中，

master 是逐步填充一个新的 chunkserver,而不是立即将新的 chunk 以及大量沉重的写流量使他忙的不可开交。对于一个新副本的放置，类似于前面的那些讨论。另外，master 必须选择删除哪个现有的副本。通常来说，它更喜欢那些存放在低于平均磁盘空闲率的 chunkserver 上的 chunk，这样可以使磁盘使用趋于相等。

4.4 Garbage Collection

文件删除后，GFS 并不立即释放可用的物理存储。它会将这项工作推迟到文件和 chunk 级别的垃圾回收时做。我们发现，这种方法使得系统更简单更可靠。

4.4.1 Mechanism

当文件被应用程序删除时,master 会将这个删除操作立刻写入日志(就像其他变更操作)。但是文件不会被立即删除(回收)，而是被重命名为一个包含删除时间戳的隐藏名称。在 master 对文件系统进行常规扫描时，它会删除那些存在时间超过 3 天(这个时间是可以配置的)的隐藏文件。在此之前，文件依然可以用那个新的特殊名称进行读取，或者重命名回原来的名称来取消删除。当隐藏文件从名字空间删除后，它的元数据会被擦除。这样就有效地切断了它与所有 chunk 的关联。

在 chunk 的类似的常规扫描中，master 找到那些孤儿块(无法从任何文件到达)，擦除这些块的元数据。在 chunkserver 与 master 周期性心跳信息中，chunkserver 报告它所拥有的 chunk 的那个子集，然后 master 返回那些不在 master 的元数据中出现的 chunk 的标识。Chunkserver 就可以自由的删除这些 chunk 的那些副本了。

4.4.2 Discussion

虽然分布式的垃圾回收是一个艰巨的问题,在程序设计的时候需要复杂的解决，但是在我们的系统中却是比较简单的。我们可以轻易辨别出对一个 chunk 的全部引用：它们都唯一保存在 master 的 file-to-chunk 映射中。我们也可以容易辨别所有的 chunk 副本：它们是在各个 chunkserver 上的指定目录下的 linux 文件。所有不被 master 知道的副本就是”垃圾”。

采用垃圾回收方法收回存储空间与直接删除相比，提供了几个优势：首先.在经常出现组件失败的大规模分布式系统中，它是简单而且可靠的。Chunk 创建

可能在某些 chunkserver 上成功，在另外一些失败，这样就留下一些 master 所不知道的副本。副本删除消息可能丢失，master 必须记得在出现失败时进行重发。垃圾回收提供了一种统一，可信赖的清除无用副本的方式。其次，它将存储空间回收与 master 常规的后台活动结合在一起，比如名字空间扫描，与 chunkserver 的握手。因此它们是绑在一块执行的，这样开销会被平摊。而且只有当 master 相对空闲时才会执行。Master 就可以为那些具有时间敏感性的客户端请求提供更好的响应。第三，空间回收的延迟为意外的不可逆转的删除提供了一道保护网。

根据我们的经验，主要的缺点是：滞后删除会导致阻碍我们尝试调整磁盘使用情况的效果。那些频繁创建和删除中间文件的应用程序不能够立即重用磁盘空间。我们通过当已删除的文件被再次删除时加速它的存储回收来解决这个问题。我们也允许用户在不同的 namespace 内使用不同的重备份和回收策略。比如用户可以指定某个目录树下的文件的 chunk 使用无副本存储，任何已删除的文件立刻并且不可撤销的从文件系统状态中删除。

4.5 Stale Replica Detection

如果 chunkserver 失败或者在它停机期间丢失了某些更新，chunk 副本就可能变为过期的。对于每个 chunk，master 维护一个版本号来区分最新和过期的副本。

无论何时只要 master 为一个 chunk 授权一个新的租约，那么它的版本号就会增加，然后通知副本进行更新。在一致的状态下，Master 和所有副本都会记录这个新的版本号。这发生在任何 client 被通知以前，因此也就是 client 开始向 chunk 中写数据之前。如果另一个副本当前不可用，它的 chunk 版本号就不会被更新。当 chunkserver 重启或者报告它的 chunk 和对应的版本号的时候，master 会检测该 chunkserver 是否包含过期副本。如果 master 发现有些版本号大于它的记录，master 就认为它在授权租约时失败了，所以采用更高的版本号的那个进行更新。

Master 通过周期性的垃圾回收删除过期副本。在删除之前,它需要确认在它给所有客户端的 chunk 信息请求的应答中都没有包含这个过期的副本。作为另外一种保护措施，当 master 通知客户端那个 chunkserver 包含某 chunk 的租约或者当它在 clone 操作中让 chunkserver 从另一个 chunkserver 中读取 chunk 时，会将 chunk 的版本号包含在内。当 clinet 和 chunkserver 执行操作时，总是会验证版本号，这样就使得它们总是访问最新的数据。

5.Fault Tolerance And Diagnosis

容错和诊断。在设计系统时，一个最大的挑战就是频繁的组件失败。组件的数量和质量使得这些问题变成一种常态而不再是异常。我们不能完全信任机器也不能完全信任磁盘。组件失败会导致系统不可用，甚至是损坏数据。我们讨论下如何面对这些挑战，以及当它们不可避免的发生时，在系统中建立起哪些工具来诊断问题。

5.1 High Availability

在 GFS 的数百台服务器中，在任何时间总是有一些是不可用的。我们通过两个简单有效的策略来保持整个系统的高可用性：快速恢复和备份。

5.1.1 Fast Recovery

快速恢复机制。经过设计后 Master 和 chunkserver 无论何时并以任意的方式被终止,我们都可以在在几秒内恢复它们的状态并启动。事实上，我们并没有区分正常和异常的终止。服务器通常都是通过杀死进程来关闭。客户端和其他服务器的请求超时后会经历一个小的停顿，然后重连那个重启后的服务器，进行重试。6.2.2 报告了观测到的启动时间。

5.1.2 Chunk Replication

chunk 备份。正如之前讨论的，每个 chunk 备份在不同机柜上的多个 chunkserver 上。用户可以在不同名字空间内设置不同的备份级别，默认是 3. 当 chunkserver 离线或者通过检验和检测到某个 chunk 损坏后(5.2 节)，master 会克隆现有的副本使得副本的数保持充足。尽管副本已经很好的满足了我们的需求，我们还探寻一些其他的具有同等或者更少 code 的跨机器的冗余方案，来满足我们日益增长的只读存储需求。我们期望在我们的非常松散耦合的系统中实现这些更复杂的冗余模式是具有挑战性但是可管理的。因为我们的负载主要是 append 和读操作而不是小的随机写操作。

5.1.3 Master Replication

为了可靠性，master 的状态需要进行备份。它的操作日志和检查点备份在多台机器上。对于状态的变更只有当它的操作日志被写入到本地磁盘和所有的远程备份后，才认为它完成。为了简单起见，master 除了负责进行各种后台活动比如：垃圾回收外，还要负责处理所有的变更。当它失败后，几乎可以立即重启。如果它所在的机器或者硬盘坏了，独立于 GFS 的监控设施会利用备份的操作日志在别处重启一个新的 master 进程。Client 仅仅使用 master 的一个典型名称(比如 gfs-test)来访问它，这是一个 DNS 名称，如果 master 被重新部署到一个新的机器上，可以改变它。

此外，当主 master down 掉之后，还有多个影子 master 可以提供对文件系统的只读访问。它们是影子，而不是镜像，这意味着它们可能比主 master 要滞后一些，通常可能是几秒。对于那些很少发生变更的文件或者不在意轻微过时的应用程序来说，它们增强了读操作的可用性。实际上，因为文件内容是从 chunkserver 中读取的，应用程序并不会看到过期的文件内容。文件元数据可能在短期内是过期的，比如目 录内容或者访问控制信息。

5.2 Data Integrity

数据完整性。每个 chunkserver 通过校验和来检测存储数据中的损坏。GFS 集群通常具有分布在几百台机器上的数千块硬盘，这样它就会经常出现导致数据损坏或丢失的硬盘失败。我们可以从 chunk 的其他副本中恢复被损坏的数据，但是如果通过在 chunkserver 间比较数据来检测数据损坏是不现实的。另外，有分歧的备份仍然可能是合法的：根据 GFS 的变更语义，尤其是前面提到的原子性的 record append 操作，并不保证所有副本是完全一致的。因此每个 chunkserver 必须通过维护一个检验和来独立的验证它自己的拷贝的完整性。

一个 chunk 被划分为 64kb 大小的块。每个块有一个相应的 32bit 的校验和。与其他的元数据一样，校验和与用户数据分离的，它被存放在内存中，同时通过日志进行持久化存储。

对于读操作，chunkserver 在向请求者(可能是一个 client 或者其他的 chunkserver)返回数据前，需要检验与读取边界重叠的那些数据库的校验和。因此 chunkserver 不会将损坏数据传播到其他机器上去。如果一个块的校验和与记录中的不一致，chunkserver 会向请求者返回一个错误，同时向 master 报告这个不匹配。之后，请求者会向其他副本读取数据，而 master 则会用其他

副本来 clone 这个 chunk。当这个合法的新副本创建成功后，master 向报告不匹配的那个 chunkserver 发送指令删除它的副本。

校验和对于读性能的影响很小，因为：我们大部分的读操作至少跨越多个块，我们只需要读取相对少的额外数据来进行验证。GFS client 代码通过尽量在校验边界上对齐读操作大大降低了开销。另外在 chunkserver 上校验和的查找和比较不需要任何的 IO 操作，校验和的计算也可以与 IO 操作重叠进行。

校验和计算对于 append 文件末尾的写操作进行了特别的优化。因为它们在工作负载中占据了统治地位。我们仅仅增量性的更新最后一个校验块的校验值，同时为那些 append 尾部的全新的校验块计算它的校验值。即使最后一个部分的校验块已经损坏，而我们现在无法检测出它，那么新计算出来的校验和将不会与存储数据匹配，那么当这个块下次被读取时，就可以检测到这个损坏。(也就是说这里并没有验证最后一个块的校验值，而只是更新它的值，也就是说这里省去了验证的过程，举个例子假设最后一个校验块出现了错误，由于我们的校验值计算时是增量性的，也就是说下次计算不会重新计算已存在的这部分数据的校验和，这样该损坏就继续保留在校验和里，关键是因为这里采用了增量型的校验和计算方式)。

与之相对的，如果一个写操作者覆盖了一个现有 chunk 的边界，我们必须首先读取和验证操作边界上的第一个和最后一个块，然后执行写操作，最后计算和记录新的校验和。如果在覆盖它们之前不验证第一个和最后一个块，新的校验和就可能隐藏掉那些未被覆盖的区域的数据损坏。(因为这里没有采用增量计算方式，因为它是覆盖不是 append 所以现有的检验和就是整个块的没法从中取出部分数据的校验和，必须重新计算)。

在空闲期间，chunkserver 可以扫描验证处在非活动状态的 trunk 的内容。这允许我们检测到那些很少被读取的数据的损失。一旦损坏被发现，master 就可以创建一个新的未损坏副本并且删除损坏的副本。这就避免了一个不活跃的坏块骗过 master，让之以为块有足够的好的副本。

5.3 Diagnostic Tools

诊断工具。全面而详细的诊断性的日志以很小的成本提供了问题分解，调试，性能分析上不可估量的帮助。没有日志，就很难理解那些机器间偶然出现的不可重复的交互。GFS 生成一个诊断日志用来记录很多重要事件(比如 chunkserver 的启动停止)以及所有 RPC 请求和应答。这些诊断日志可以自由的删除而不影响系统的正常运行。然而，只要磁盘空间允许，我们会尽量保存这些日志。

除了正在读写的文件数据,RPC 日志包含了精确(exact,所有???)的请求和

响应信息。通过匹配请求和响应，整理不同机器上的 RPC 日志，我们可以重新构建出整个交互历史来诊断一个问题。这些日志也可以用来进行负载测试和性能分析。

因为日志是顺序异步写的，因此写日志对于性能的影响是很小的，得到的好处却是大大的。最近的事件也会保存在内存中，可以用于持续的在线监控。

6.Measurements

在这一节，我们用一些小规模的测试来展示 GFS 架构和实现固有的一些瓶颈，有一些数字来源于 google 的实际集群。

6.1 Mirco-benchmarks

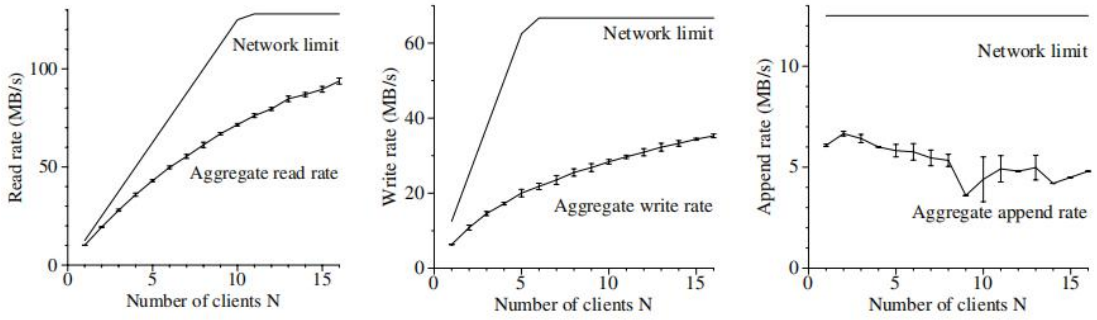
我们在一个 master，两个 master 备份，16 个 chunkserver，16 个 client 组成的 GFS 集群上进行了性能测量。这个配置是为了方便测试，实际中的集群通常会有数百个 chunkserver，数百个 client。

所有机器的配置是，双核 PIII 1.4GHz 处理器，2GB 内存，两个 80G，5400rpm 硬盘，以及 100Mbps 全双工以太网连接到 HP2524 交换机。所有 19 个 GFS 服务器连接在一个交换机，所有 16 个客户端连接在另一个上。两个交换机用 1Gbps 的线路连接。

6.1.1 Reads

N 个客户端从文件系统中并发读。每个客户端在一个 320GB 的文件集合里随机 4MB 进行读取。然后重复 256 次，这样每个客户端实际上读取了 1GB 数据。Chunkserver 总共只有 32GB 内存，因此我们估计在 linux 的 buffer cache 里最多有 10%的命中率。我们的结果应该很接近一个几乎无缓存的结果。

图 3(a)展示了对于 N 个客户端的总的读取速率以及它的理论上的极限。当 2 个交换机通过一个 1Gbps 的链路连接时，它的极限峰值是 125MB/s，客户端通过 100Mbps 连接，那么换成单个客户端的极限就是 12.5MB/s。当只有一个客户端在读取时，观察到的读取速率是 10MB/s，达到了单个客户端极限的 80%。当 16 个读取者时，总的读取速率的 94 MB/s，大概达到了链路极限(125MB/s)的 75%，换成单个客户端就是 6 MB/s。效率从 80%降到了 75%，是因为伴随着读取者的增加，多个读者从同一个 chunkserver 并发读数据的概率也随之变大。



(a) Reads (b) Writes (c) Record appends

6.1.2 Writes

N 个客户端并行向 N 个不同的文件写数据。每个客户端以 1MB 的单个写操作总共向一个新文件写入 1GB 数据。总的写速率以及它的理论上的极限如图 3(b)所示。极限值变成了 67 MB/s,是因为我们需要将每个字节写入到 16 个 chunkserver 中的 3 个，每个具有 12.5MB/s 的输入连接。

单个客户端的写入速率是 6.3 MB/s,大概是极限值的一半。主要原因是我们的网络协议栈。它不能充分利用我们用于 chunk 副本数据推送的流水线模式。将数据从一个副本传递到另一个副本的延迟降低了整体的写速率。

对于 16 个客户端，总体的写入速率达到了 35 MB/s，平均每个客户端 2.2 MB/s,大概是理论极限的一半。与写操作类似，伴随着写操作的增加，多个写操作从同一个 chunkserver 并发写数据的概率也随之变大。另外对于 16 个写操作比 16 个读者更容易产生碰撞，因为每个写操作将关联到 3 个不同的副本。

写操作比我们期望的要慢。在实际中，这还未变成一个主要问题，因为尽管它可能增加单个客户端的延时，但是当系统面对大量客户端时，其总的写入带宽并没有显著的影响。

6.1.3 Record Appends

图 3(c)展示了 record append 的性能。N 个客户端向单个文件并行的 append。性能取决于保存了该文件最后那个 chunk 的那些 chunkserver，与客户端的数目无关。当只有一个客户端时，能达到 6.0MB/s，当有 16 个客户端时就降到了 4.8 MB/s。主要是由于拥塞以及不同的客户端的网络传输速率不同造成的。

我们的应用程序倾向于并行创建多个这样的文件。换句话说，N 个客户端向 M 个共享文件并行 append，在这里 N 和 M 通常是几十甚至几百大小。因此在我

们的实验中出现的 chunkserver 的网络拥塞问题在实际中并不是一个显著的问题，因为当一个文件的 chunkserver 比较繁忙的时候，它可以去写另一个。

6.2 Real World Clusters

我们选择在 google 内部使用的两个集群进行测试作为相似的那些集群的一个代表。集群 A 主要用于 100 多个工程的日常研发。它会从数 TB 的数据中读取数 MB 的数据，对这些数据进行转化或者分析，然后将结果再写回集群。集群 B 主要用于产品数据处理。它上面的任务持续时间更长，持续地在生成和处理数 TB 的数据集合，只是偶尔可能需要人为的参与。在这两种情况下，任务都是由分布在多个机器上的很多进程组成，它们并行的读写很多文件。

6.2.1 Storage

正如表中前 5 个字段所展示的，两个集群都有数百个 chunkserver，支持 TB 级的硬盘空间，空间已经被充分使用但还没全满。已用的空间包含 chunk 的所有副本。通常文件存在三个副本，因此这两个集群实际分别存储了 18TB 和 52TB 的数据。

这两个集群的文件数目很接近，尽管 B 集群有大量的死文件(那些已经被删除或者被新版本文件所替换但空间还没有被释放的文件)。而且它具有更多的 trunk，因为它上面的文件通常更大。

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

6.2.2 Metadata

所有的 Chunkserver 总共存储了数十 G 的元数据，大部分是用户数据的 64kb 块的校验和。Chunkserver 上唯一的其他的元数据就是 4.5 节讨论的 chunk 的版本号。

保存在 master 上的元数据要更小一些，只有数十 MB，平均下来每个文件只有 100 来个字节。这也刚好符合我们的 master 的内存不会成为实际中系统容量限制的假设。每个文件的元数据主要是以前缀压缩格式存储的文件名称。还有一些其他的元数据比如文件所有者，权限，文件到 chunk 的映射以及 chunk 的当前版本。另外对于每个 chunk 我们还存储了当前的副本位置以及用于实现写时复制的引用计数。

每个独立的 server(chunkserver 和 master)只有 50-100MB 的元数据。因此，恢复是很快的：在 server 可以应答查询前只需要花几秒钟的时间就可以把它们从硬盘上读出来。然而，master 的启动可能要慢一些，通常还需要 30-60 秒从所有的 chunkserver 获得 chunk 的位置信息。

6.2.3 Read and Write Rates

表 3 展示了不同时期的读写速率。进行这些测量时，两个集群都已经运行了大约一周(为了更新到最新版本的 GFS，这两个集群被重启过)。

从启动开始看，平均写速率小于 30MB/s。当我们进行这些测量时，集群 B 正在以 100MB/s 的速率进行密集的写操作，同时产生了 300MB/s 的网络负载，因为写操作将会传给 3 个副本。

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

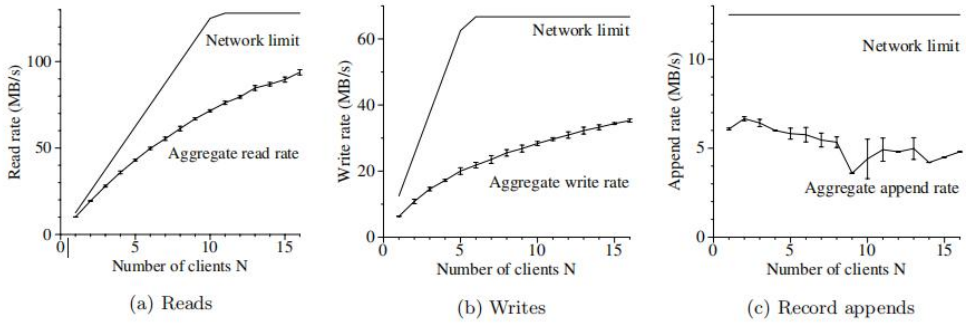


Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

读速率要远高于写速率。正如我们料想的那样，整个工作负载组成中，读要多于写。这两个集群都处在繁重的读活动中。尤其是，A 已经在过去的一个星期中维持了 580MB/s 的读速率。它的网络配置可以支持 750MB/s，因此它已经充分利用了资源。B 集群可支持 1300 MB/s 的峰值读速率，但是应用只使用了 380 MB/s。

6.2.4 Master Load

表 3 也表明发送给 master 的操作速率大概是每秒 200-500 个操作。Master 可以轻易的处理这个级别的速率，因此对于这些工作负载来说，它不会成为瓶颈。

在早期版本的 GFS 中，master 偶尔会成为某些工作负载的瓶颈。为了查找文件，花费大量的时间在巨大的目录(包含上千万的文件)中进行线性扫描。因此，我们改变了 master 的数据结构，使之可以在名字空间内进行有效的二分搜索。现在它可以简单的支持每秒上千次的文件访问。如果必要的话，我们可以通过在 namespace 数据结构前面放置名称查找缓存来进一步加快速度(If necessary, we could speed it up further by placing name lookup caches in front of the namespace data structures.)。

6.2.5 Recovery Time

一台 Chunkserver 失败后，它上面的那些 chunk 的副本数就会降低，必须进行 clone 以维持正常的副本数。恢复这些 chunk 的时间取决于资源的数量。在一个实验中，我们关闭集群 B 中的一个 chunkserver。该 chunkserver 大概有 15000 个 chunk，总共 600GB 的数据。为减少对于应用程序的影响以及为调度决策提供余地，我们的默认参数设置将集群的并发 clone 操作限制在 91 个(占 chunkserver 个数的 40%)，同时每个 clone 操作最多可以消耗

6.25MB/s(50Mbps)。所有的 chunk 在 23.2 分钟内被恢复，备份速率是 440MB/s。

在另一个实验中，我们关掉了两个 chunkserver，每个具有 16000 个 chunk，660GB 的数据。这次失败使得 266 个 chunk 降低到了一个副本，但是两分钟内，它们就恢复到了至少 2 个副本，这样就让集群能够容忍另一个 chunkserver 发生失败，而不产生数据丢失。

6.3 Workload Breakdown

工具负载解析。在这一节，我们将继续在两个新的集群上对工作负载进行细致的对比分析。集群 X 是用于研究开发的，集群 Y 是用于产品数据处理。

6.3.1 Methodology and Caveats

这些结果只包含了客户端产生的请求，因此它们反映了应用程序的对整个文件系统的工作负载。并不包含为了执行客户端的请求进行的 server 间的请求，或者是内部的后台活动，比如写推送或者是重平衡。

对于 IO 操作的统计是从 GFS 的 server 的 PRC 请求日志中重新构建出来的。比如为了增加并行性，GFS 客户端代码可能将一个读操作拆分为多个 RPC 请求，我们通过它们推断出原始请求。因为我们的访问模式高度的程式化，希望每个错误都可以出现在日志中。应用程序显式的记录可以提供更精确的数据，但是重新编译以及重启正在运行中的客户端在逻辑上是不可能这样做的。而且由于机器数很多，收集这些数据也会变得很笨重。

需要注意的是，不能将我们的工作负载过于泛化(generalize)。因为 GFS 和其应用程序是由 google 完全控制的，这些应用程序都是倾向于针对 GFS 进行专门调整，同时 GFS 也是专门为这些应用而设计的。这种相互的影响可能也存在于一般的文件系统及其应用程序中，但是在我们的案例中这种影响可能更加明显。

6.3.2 Chunkserver Workload

表 4 按大小显示了操作的分布。读操作的大小表现出双峰分布，小型读操作(小于 64kb)来自于那些在大量文件中查找小片数据的随机读客户端，大型读操作(超过 512kb)来自于穿越整个文件的线性读操作。

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 Y 中大量的读操作没有返回数据。我们应用程序，尤其是在产品系统中，经常使用文件作为生产者消费者队列。生产者并行的往文件中 append 数据，而消费者则从文件尾部读数据。有时候，如果消费者超过了生产者，就没有数据返回。集群 X 很少出现这种情况，因为它主要是用来进行短期数据分析，而不是长期的分布式应用。

写操作的大小也表现出双峰分布。大型的写操作(超过 256KB)通常来自于写操作者的缓冲。那些缓冲更少数据的写操作者，检查点或者经常性的同步或者简单的数据生成组成了小型的写操作(低于 64KB)。

对于记录的 append，Y 集群比 X 集群可以看到更大的大 record append 比率。因为使用 Y 集群的产品系统，针对 GFS 进行了更多的优化。

表 5 展示了不同大小的数据传输总量。对于各种操作来说，大型的操作(超过 256KB)构成了大部分的数据传输。但是小型(低于 64KB)的读操作虽然传输了比较少的数据但是在数据读中也占据了相当的一部分，主要是由于随机 seek 造成的。

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

6.3.3 Appends versus Writes

记录 append 操作被大量的应用尤其是在我们的产品系统中。对于集群 X 来说，按字节传输来算，write 与 append 的比例是 108：1，根据操作数来算它们的比例是 8：1。对于集群 Y，比例变成了 3.7：1 和 2.5：1。对于这两个集群来说，它们的 append 操作都要比 write 操作大一些{操作数的比要远大于字节数的比，说明单个的 append 操作的字节数要大于 write。对于集群 X 来说，在测量期间的记录 append 操作要低一些，这可能是由其中具有特殊缓冲大小设置的应用程序造成的。

正如期望的，我们的数据变更操作处于支配地位的是追加而不是重写(write 也可能是追加)。我们测量了在主副本上的数据重写数量。对于集群 X 来说，以字节大小计算的话重写大概占了整个数据变更的 0.0001%，以操作个数计算，大概小于 0.0003%。对于 Y 集群来说，这两个数字都是 0.05%，尽管这也不算大，但是还是要高于我们的期望。结果显示，大部分的重写是由于错误或者超时导致的客户端重写而产生的。它们并不是工作负载的一部分，而是属于重试机制。

6.3.4 Master Workload

表 6 展示了对于 master 各种请求类型的剖析。大部分请求是为了得到 chunk 位置以及数据变更需要的租约持有信息。

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

Table 6: Master Requests Breakdown by Type (%)

可以看到集群 X 和 Y 在 delete 请求上的限制区别，因为集群 Y 上存储的产品信息会周期性地生成被新版本数据所替换。这些不同被隐藏在 open 请求中，因为老版的数据在被写的时候的打开操作中被隐式的删除(类似与 Unix 的”w”打开模式)。

查找匹配文件是一个类似于 ls 的模式匹配请求。不像其他的请求，它可能需要处理很大部分的名字空间，因此可能是很昂贵的。在集群 Y 上可以更频繁地看到它，因为自动化的数据处理任务为了了解整个应用程序的状态可能需要检查

文件系统中的某些部分。与此相比，集群 X 需要更多显式的用户控制而且已经提前知道所需要的文件的名称。

7.Experiences

在构建和部署 GFS 的过程中，我们总结出了很多经验，观点和技术。

起初，GFS 只是考虑作为我们产品系统的后端文件系统。随着时间的推移，开始在研究和开发中使用。一开始它基本不支持像权限,磁盘配额(quota)这些东西，但是现在它们都已经有了。产品系统是很容易控制的，但是用户却不是。因此需要更多的设施来避免用户间的干扰。

我们最大的问题是硬盘和 linux 相关性。我们的很多硬盘声称支持各种 IDE 协议版本的 linux 驱动，但是实际上它们只能在最近的一些版本上才能可靠的工作。因此如果协议版本如果相差不大，硬盘大多数情况下都可以工作，但是有时候这种不一致会使得驱动和内核在硬盘状态上产生分歧。由于内核的问题，这将会导致数据被默默的污染。这个问题使得我们使用校验和来检测数据污染，如果出现这种情况，我们就需要修改内核来处理这种协议不一致的情况。

之前，由于 linux2.2 内核的 fsync()的花费，我们也碰到过一些问题。它的花费是与文件大小而不是被修改部分的大小相关的。这对于我们大的操作日志会是一个问题，尤其是在我们实现检查点之前。我们通过改用同步写来绕过了这个问题，最后迁移到 Linux2.4 来解决了它。

另一个由于 linux 产生的问题是与读写锁相关的。在一个地址空间里的线程在从硬盘中读页数据(读锁)或者在 mmap 调用中修改地址空间(写锁)的时候，必须持有一个读写锁。在系统负载很高，产生资源瓶颈或者出现硬件失败时，我们碰到了瞬态的超时。最后，我们发现当磁盘读写线程处理前面映射的数据时，这个锁阻塞了网络线程将新的数据映射到内存。由于我们的工作瓶颈主要是在网络带宽而不是内存带宽，因此我们通过使用 pread()加上额外的开销替代 mmap()绕过了这个问题。

尽管出现了一些问题，linux 代码的可用性帮助了我们探索和理解系统的行为。在适当的时机，我们也会改进内核并与开源社区共享这些变化。

8.Related work

像其他的大型分布式文件系统比如 AFS，GFS 提供了一个本地的独立名字空间，使得数据可以为了容错或者负载平衡而透明的移动。但与 AFS 不同的是，

为了提升整体的性能和容错能力，GFS 将文件数据在多个存储服务器上存储，这点更类似于 xFS 或者 Swift。

硬盘是相对便宜的，而且与复杂的 RAID 策略相比，副本策略更简单。由于 GFS 完全采用副本策略进行冗余因此它会比 xFS 或者 Swift 消耗更多的原始存储。

与 AFS,xFS,Frangipani,Intermezzo 这些系统相比，GFS 在文件系统接口下并不提供任何缓存。我们的目标工作负载类型对于通常的单应用程序运行模式来说，基本上是不可重用的，因为这种模式通常需要读取大量数据集合或者在里面进行随机的 seek，而每次只读少量的数据。

一些分布式文件系统比如 xFS,Frangipani,Minnesota’s GFS 和 GPFS 删除了中央服务节点，依赖于分布式的算法进行一致性和管理。我们选择中央化测量是为了简化设计增加可靠性，获取灵活性。尤其是，一个中央化的 master 更容易实现复杂的 chunk 放置和备份策略，因为 master 具有大部分的相关信息以及控制了它们的改变。我们通过让 master 状态很小以及在其他机器上进行备份来解决容错。当前通过影子 master 机制提供可扩展性和可用性。对于 master 状态的更新，通过 append 到 write-ahead 日志里进行持久化。因此我们可以通过类似于 Harp 里的主 copy 模式来提供一个比我们当前模式具有更强一致性的高可用性。我们未来将解决类似于 Lustre 的一个问题：大量客户端的整体性能。然而我们通过专注于我们自己的需求而不是构建一个 POSIX 兼容文件系统来简化了这个问题。另外，GFS 加速不可靠组件的数量是很大的，因此容错是我们设计的中心。GFS 很类似于 NASD 架构。但是 NASD 是基于网络连接的硬盘驱动器，GFS 则使用普通机器作为 chunkserver。与 NASD 不同，chunkserver 在需要时分配固定大小的 chunk，而没有使用变长对象。此外，GFS 还实现了诸如重平衡，副本，产品环境需要的快速恢复。

不像 Minnesota’s GFS 和 NASD，我们并没有寻求改变存储设备的模型。我们更专注于解决使用现有商品化组件组成的复杂分布式系统的日常的数据处理需求。

通过在生产者消费者队列中使用原子 record append 操作解决了与分布式操作系统 River 的类似问题。River 使用基于内存的跨机器分布式队列以及小心的数据流控制来解决这个问题，而 GFS 只使用了一个可以被很多生产者 append 数据的文件。River 模型支持 mton 的分布式队列，但是缺乏容错，GFS 目前只支持 m to 1。多个消费者可以读取相同文件，但是它们必须协调好对输入负载进行划分(各自处理不相交的一部分)。

9.Conclusions

GFS 包含了那些在商品化硬件上支持大规模数据处理的必要特征。尽管某些设计决定与我们特殊的应用类型相关，但是可以应用在具有类似需求和特征的数据处理任务中。

针对我们当前的应用负载类型，我们重新审视传统的文件系统的一些假设。我们的审视，使得我们的设计中产生了一些与之根本不同的观点。我们将组件失败看做常态而不是异常，为经常进行的在大文件上的 append 进行优化，然后是读(通常是顺序的)，为了改进整个系统我们扩展并且放松了标准文件系统接口。

我们的系统通过监控，备份关键数据，快速和自动恢复来提供容错。Chunk 备份使得我们可以容忍 chunkserver 的失败。这些经常性的失败，驱动了一个优雅的在线修复机制的产生，它周期性地透明的进行修复尽快的恢复那些丢失的副本。另外，我们通过使用校验和来检测数据损坏，当系统中硬盘数目很大的时候，这种损坏变得很正常。

我们的设计实现了对于很多执行大量任务的并发读者和写者的高吞吐率。通过从数据传输中分离文件系统控制，我们来实现这个目标，让 master 来处理文件系统控制，数据传输则直接在 chunkserver 和客户端之间进行。通过增大 chunk 的大小以及 chunk 的租约机制，降低了 master 在普通操作中的参与。这使中央的 master 不会成为瓶颈。我们相信在当前网络协议栈上的改进将会提供客户端写出速率的限制。

GFS 成功地满足了我们的存储需求，同时除了作为产品数据处理平台外，还作为研发的存储平台而被广泛使用。它是一个使我们可以持续创新以及面对整个 web 的海量数据挑战的重要工具