Bigtable：一个分布式的结构化数据存储系统[中文版]

2010年3月27日 blademaster 没有评论

Bigtable：一个分布式的结构化数据存储系统

*译者：**alex***

摘要

Bigtable是一个分布式的结构化数据存储系统，它被设计用来处理海量数据：通常是分布在数千台普通服务器上的PB级的数据。Google的很多项目使用Bigtable存储数据，包括Web索引、Google Earth、Google Finance。这些应用对Bigtable提出的要求差异非常大，无论是在数据量上（从URL到网页到卫星图像）还是在响应速度上（从后端的批量处理到实时数据服务）。尽管应用需求差异很大，但是，针对Google的这些产品，Bigtable还是成功的提供了一个灵活的、高性能的解决方案。本论文描述了Bigtable提供的简单的数据模型，利用这个模型，用户可以动态的控制数据的分布和格式；我们还将描述Bigtable的设计和实现。

1 介绍

在过去两年半时间里，我们设计、实现并部署了一个分布式的结构化数据存储系统 — 在Google，我们称之为Bigtable。Bigtable的设计目的是可靠的处理PB级别的数据，并且能够部署到上千台机器上。Bigtable已经实现了下面的几个目标：适用性广泛、可扩展、高性能和高可用性。Bigtable已经在超过60个Google的产品和项目上得到了应用，包括Google Analytics、Google Finance、Orkut、Personalized Search、Writely和Google Earth。这些产品对Bigtable提出了迥异的需求，有的需要高吞吐量的批处理，有的则需要及时响应，快速返回数据给最终用户。它们使用的Bigtable集群的配置也有很大的差异，有的集群只有几台服务器，而有的则需要上千台服务器、存储几百TB的数据。

在很多方面，Bigtable和数据库很类似：它使用了很多数据库的实现策略。并行数据库【14】和内存数据库【13】已经具备可扩展性和高性能，但是Bigtable提供了一个和这些系统完全不同的接口。Bigtable不支持完整的关系数据模型；与之相反，Bigtable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分布和格式*（alex注：也就是对BigTable而言，数据是没有格式的，用数据库领域的术语说，就是数据没有Schema，用户自己去定义Schema），*用户也可以自己推测*(alex注：reason about)*底层存储数据的位置相关性*(alex注：位置相关性可以这样理解，比如树状结构，具有相同前缀的数据的存放位置接近。在读取的时候，可以把这些数据一次读取出来)*。数据的下标是行和列的名字，名字可以是任意的字符串。Bigtable将存储的数据都视为字符串，但是Bigtable本身不去解析这些字符串，客户程序通常会在把各种结构化或者半结构化的数据串行化到这些字符串里。通过仔细选择数据的模式，客户可以控

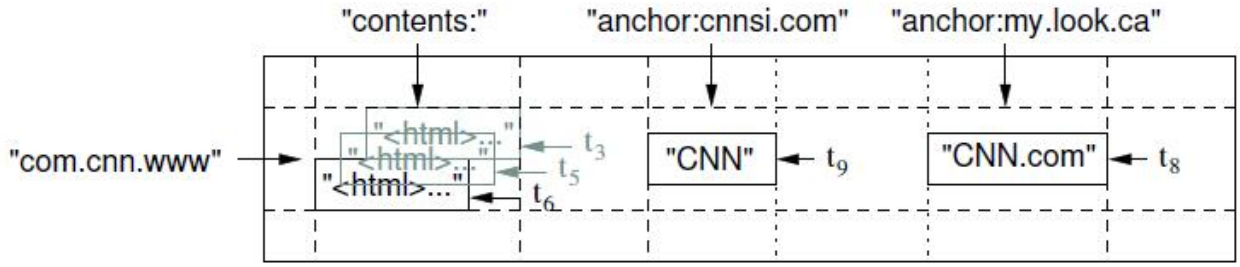
制数据的位置相关性。最后，可以通过BigTable的模式参数来控制数据是存放在内存中、还是硬盘上。第二节描述关于数据模型更多细节方面的东西；第三节概要介绍了客户端API；第四节简要介绍了BigTable底层使用的Google的基础框架；第五节描述了BigTable实现的关键部分；第6节描述了我们为了提高BigTable的性能采用的一些精细的调优方法；第7节提供了BigTable的性能数据；第8节讲述了几个Google内部使用BigTable的例子；第9节是我们在设计和后期支持过程中得到一些经验和教训；最后，在第10节列出我们的相关研究工作，第11节是我们的结论。

2 数据模型

Bigtable是一个稀疏的、分布式的、持久化存储的多维度排序Map*（alex注：对于程序员来说，Map应该不用翻译了吧。Map由key和value组成，后面我们直接使用key和value，不再另外翻译了）*。Map的索引是行关键字、列关键字以及时间戳；Map中的每个value都是一个未经解析的byte数组。

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

我们在仔细分析了一个类似Bigtable的系统的种种潜在用途之后，决定使用这个数据模型。我们先举个具体的例子，这个例子促使我们做了很多设计决策；假设我们想要存储海量的网页及相关信息，这些数据可以用于很多不同的项目，我们姑且称这个特殊的表为Webtable。在Webtable里，我们使用URL作为行关键字，使用网页的某些属性作为列名，网页的内容存在“contents:”列中，并用获取该网页的时间戳作为标识*(alex注：即按照获取时间不同，存储了多个版本的网页数据)*，如图一所示。



图一：一个存储Web网页的例子的表的片断。行名是一个反向URL。contents列族存放的是网页的内容，anchor列族存放引用该网页的锚链接文本*（alex注：如果不知道HTML的Anchor，请Google一把）*。CNN的主页被Sports Illustrater和MY-look的主页引用，因此该行包含了名为“anchor:cnnsi.com”和 “anchhor:my.look.ca”的列。每个锚链接只有一个版本*（alex注：注意时间戳标识了列的版本，t9和t8分别标识了两个锚链接的版本）*；而contents列则有三个版本，分别由时间戳t3，t5，和t6标识。

行

表中的行关键字可以是任意的字符串（目前支持最大64KB的字符串，但是对大多数用户，10-100个字节就足够了）。对同一个行关键字的读或者写操作都是原子的（不管读或者写这一行里多少个不同列），这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

Bigtable通过行关键字的字典顺序来组织数据。表中的每个行都可以动态分区。每个分区叫做一个”Tablet”，Tablet是数据分布和负载均衡调整的最小单位。这样做的结果是，当操作只读取行中很少几列的数据时效率很高，通常只需要很少几次机器间的通信即可完成。用户可以通过选择合适的行关键字，在数据访问时有效利用数据的位置相关性，从而更好的利用这个特性。举例来说，在Webtable里，通过反转URL中主机名的方式，可以把同一个域名下的网页聚集起来组织成连续的行。具体来说，我们可以把maps.google.com/index.html的数据存放在关键字com.google.maps/index.html下。把相同的域中的网页存储在连续的区域可以让基于主机和域名的分析更加有效。

列族

列关键字组成的集合叫做“列族“，列族是访问控制的基本单位。存放在同一列族下的所有数据通常都属于同一个类型（我们可以把同一个列族下的数据压缩在一起）。列族在使用之前必须先创建，然后才能在列族中任何的列关键字下存放数据；列族创建后，其中的任何一个列关键字下都可以存放数据。根据我们的设计意图，一张表中的列族不能太多（最多几百个），并且列族在运行期间很少改变。与之相对应的，一

张表可以有无限多个列。

列关键字的命名语法如下：*列族：限定词*。 列族的名字必须是可打印的字符串，而限定词的名字可以是任意的字符串。比如，Webtable有个列族language，language列族用来存放撰写网页的语言。我们在language列族中只使用一个列关键字，用来存放每个网页的语言标识ID。Webtable中另一个有用的列族是anchor；这个列族的每一个列关键字代表一个锚链接，如图一所示。Anchor列族的限定词是引用该网页的站点名；Anchor列族每列的数据项存放的是链接文本。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。在我们的Webtable的例子中，上述的控制权限能帮助我们管理不同类型的应用：我们允许一些应用可以添加新的基本数据、一些应用可以读取基本数据并创建继承的列族、一些应用则只允许浏览数据（甚至可能因为隐私的原因不能浏览所有数据）。

时间戳

在Bigtable中，表的每一个数据项都可以包含同一份数据的不同版本；不同版本的数据通过时间戳来索引。Bigtable时间戳的类型是64位整型。Bigtable可以给时间戳赋值，用来表示精确到毫秒的“实时”时间；用户程序也可以给时间戳赋值。如果应用程序需要避免数据版本冲突，那么它必须自己生成具有唯一性的时间戳。数据项中，不同版本的数据按照时间戳倒序排序，即最新的数据排在最前面。

为了减轻多个版本数据的管理负担，我们对每一个列族配有两个设置参数，Bigtable通过这两个参数可以对废弃版本的数据自动进行垃圾收集。用户可以指定只保存最后n个版本的数据，或者只保存“足够新”的版本的数据（比如，只保存最近7天的内容写入的数据）。

在Webtable的举例里，contents:列存储的时间戳信息是网络爬虫抓取一个页面的时间。上面提及的垃圾收集机制可以让我们只保留最近三个版本的网页数据。

3 API

Bigtable提供了建立和删除表以及列族的API函数。Bigtable还提供了修改集群、表和列族的元数据的API，比如修改访问权限。

```
// Open the table
Table *T = OpenOrDie(“/bigtable/web/webtable”);
// Write a new anchor and delete an old anchor
RowMutation r1(T, “com.cnn.www”);
r1.Set(“anchor:www.c-span.org”, “CNN”);
r1.Delete(“anchor:www.abc.com”);
Operation op;
Apply(&op, &r1);
```

Figure 2: Writing to Bigtable.

客户程序可以对Bigtable进行如下的操作：写入或者删除Bigtable中的值、从每个行中查找值、或者遍历表中的一个数据子集。图2中的C++代码使用RowMutation抽象对象进行了一系列的更新操作。（为了保持示例代码的简洁，我们忽略了一些细节相关代码）。调用Apply函数对Webtable进行了一个原子修改操作：它为www.cnn.com增加了一个锚点，同时删除了另外一个锚点。

```
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily(“anchor”);
stream->SetReturnAllVersions();
scanner.Lookup(“com.cnn.www”);
for (; !stream->Done(); stream->Next()) {
    printf(“%s %s %lld %s\n”,
```

```
    scanner.RowName(),
    stream->ColumnName(),
    stream->MicroTimestamp(),
    stream->Value());
}
```

Figure 3: Reading from Bigtable.

图3中的C++代码使用Scanner抽象对象遍历一个行内的所有锚点。客户程序可以遍历多个列族，有几种方法可以对扫描输出的行、列和时间戳进行限制。例如，我们可以限制上面的扫描，让它只输出那些匹配正则表达式*.cnn.com的锚点，或者那些时间戳在当前时间前10天的锚点。

Bigtable还支持一些其它的特性，利用这些特性，用户可以对数据进行更复杂的处理。首先，Bigtable支持单行上的事务处理，利用这个功能，用户可以对存储在一个行关键字下的数据进行原子性的读-更新-写操作。虽然Bigtable提供了一个允许用户跨行批量写入数据的接口，但是，Bigtable目前还不支持通用的跨行事务处理。其次，Bigtable允许把数据项用做整数计数器。最后，Bigtable允许用户在服务器的地址空间内执行脚本程序。脚本程序使用Google开发的Sawzall【28】数据处理语言。虽然目前我们基于的Sawzall语言的API函数还不允许客户的脚本程序写入数据到Bigtable，但是它允许多种形式的数据转换、基于任意表达式的数据过滤、以及使用多种操作符的进行数据汇总。

Bigtable可以和MapReduce【12】一起使用，MapReduce是Google开发的大规模并行计算框架。我们已经开发了一些Wrapper类，通过使用这些Wrapper类，Bigtable可以作为MapReduce框架的输入和输出。

4 BigTable构件

Bigtable是建立在其它的几个Google基础构件上的。BigTable使用Google的分布式文件系统(GFS)【17】存储日志文件和数据文件。BigTable集群通常运行在一个共享的机器池中，池中的机器还会运行其它的各种各样的分布式应用程序，BigTable的进程经常要和其它应用的进程共享机器。BigTable依赖集群管理系统来调度任务、管理共享的机器上的资源、处理机器的故障、以及监视机器的状态。

BigTable内部存储数据的文件是Google SSTable格式的。SSTable是一个持久化的、排序的、不可更改的Map结构，而Map是一个key-value映射的数据结构，key和value的值都是任意的Byte串。可以对SSTable进行如下的操作：查询与一个key值相关的value，或者遍历某个key值范围内的所有的key-value对。从内部看，SSTable是一系列的数据块（通常每个块的大小是64KB，这个大小是可以配置的）。SSTable使用块索引（通常存储在SSTable的最后）来定位数据块；在打开SSTable的时候，索引被加载到内存。每次查找都可以通过一次磁盘搜索完成：首先使用二分查找法在内存中的索引里找到数据块的位置，然后再从硬盘读取相应的数据块。也可以选择把整个SSTable都放在内存中，这样就不必访问硬盘了。

BigTable还依赖一个高可用的、序列化的分布式锁服务组件，叫做Chubby【8】。一个Chubby服务包括了5个活动的副本，其中的一个副本被选为Master，并且处理请求。只有在大多数副本都是正常运行的，并且彼此之间能够互相通信的情况下，Chubby服务才是可用的。当有副本失效的时候，Chubby使用Paxos算法【9,23】来保证副本的一致性。Chubby提供了一个名字空间，里面包括了目录和小文件。每个目录或者文件可以当成一个锁，读写文件的操作都是原子的。Chubby客户程序库提供对Chubby文件的一致性缓存。每个Chubby客户程序都维护一个与Chubby服务的会话。如果客户程序不能在租约到期的时间内重新签订会话的租约，这个会话就过期失效了*(alex注：又用到了lease。原文是：A client's session expires if it is unable to renew its session lease within the lease expiration time.)*。当一个会话失效时，它拥有的锁和打开的文件句柄都失效了。Chubby客户程序可以在文件和目录上注册回调函数，当文件或目录改变、或者会话过期时，回调函数会通知客户程序。

Bigtable使用Chubby完成以下的几个任务：确保在任何给定的时间内最多只有一个活动的Master副本；存储BigTable数据的自引导指令的位置（参考5.1节）；查找Tablet服务器，以及在Tablet服务器失效时进行善后（5.2节）；存储BigTable的模式信息（每张表的列族信息）；以及存储访问控制列表。如果

Chubby长时间无法访问，BigTable就会失效。最近我们在使用11个Chubby服务实例的14个BigTable集群上测量了这个影响。由于Chubby不可用而导致BigTable中的部分数据不能访问的平均比率是0.0047%（Chubby不能访问的原因可能是Chubby本身失效或者网络问题）。单个集群里，受Chubby失效影响最大的百分比是0.0326%*（alex注：有点莫名其妙，原文是： The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.）*。

5 介绍

Bigtable包括了三个主要的组件：链接到客户程序中的库、一个Master服务器和多个Tablet服务器。针对系统工作负载的变化情况，BigTable可以动态的向集群中添加（或者删除）Tablet服务器。

Master服务器主要负责以下工作：为Tablet服务器分配Tablets、检测新加入的或者过期失效的Table服务器、对Tablet服务器进行负载均衡、以及对保存在GFS上的文件进行垃圾收集。除此之外，它还处理对模式的相关修改操作，例如建立表和列族。

每个Tablet服务器都管理一个Tablet的集合（通常每个服务器有大约数十个至上千个Tablet）。每个Tablet服务器负责处理它所加载的Tablet的读写操作，以及在Tablets过大时，对其进行分割。

和很多Single-Master类型的分布式存储系统【17.21】类似，客户端读取的数据都不经过Master服务器：客户程序直接和Tablet服务器通信进行读写操作。由于BigTable的客户程序不必通过Master服务器来获取Tablet的位置信息，因此，大多数客户程序甚至完全不需要和Master服务器通信。在实际应用中，Master服务器的负载是很轻的。

一个BigTable集群存储了很多表，每个表包含了一个Tablet的集合，而每个Tablet包含了某个范围内的行的所有相关数据。初始状态下，一个表只有一个Tablet。随着表中数据的增长，它被自动分割成多个Tablet，缺省情况下，每个Tablet的尺寸大约是100MB到200MB。

5.1 Tablet的位置

我们使用一个三层的、类似B+树[10]的结构存储Tablet的位置信息(如图4)。

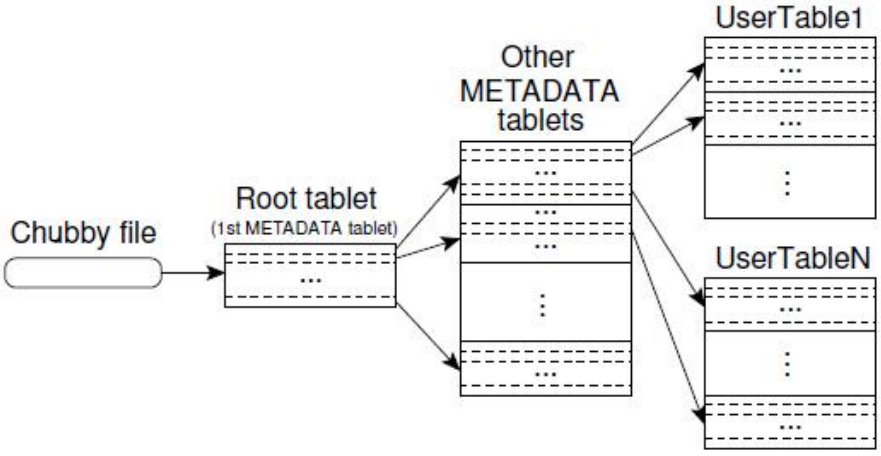


Figure 4: Tablet location hierarchy.

第一层是一个存储在Chubby中的文件，它包含了Root Tablet的位置信息。Root Tablet包含了一个特殊的METADATA表里所有的Tablet的位置信息。METADATA表的每个Tablet包含了一个用户Tablet的集合。Root Tablet实际上是METADATA表的第一个Tablet，只不过对它的处理比较特殊 — Root Tablet永远不会被分割 — 这就保证了Tablet的位置信息存储结构不会超过三层。

在METADATA表里面，每个Tablet的位置信息都存放在一个行关键字下面，而这个行关键字是由Tablet所

在的表的标识符和Tablet的最后一行编码而成的。METADATA的每一行都存储了大约1KB的内存数据。在一个大小适中的、容量限制为128MB的METADATA Tablet中，采用这种三层结构的存储模式，可以标识2^34个Tablet的地址（如果每个Tablet存储128MB数据，那么一共可以存储2^61字节数据）。

客户程序使用的库会缓存Tablet的位置信息。如果客户程序没有缓存某个Tablet的地址信息，或者发现它缓存的地址信息不正确，客户程序就在树状的存储结构中递归的查询Tablet位置信息；如果客户端缓存是空的，那么寻址算法需要通过三次网络来回通信寻址，这其中包括了一次Chubby读操作；如果客户端缓存的地址信息过期了，那么寻址算法可能需要最多 6 次网络来回通信才能更新数据，因为只有在缓存中没有查到数据的时候才能发现数据过期*（alex注：其中的三次通信发现缓存过期，另外三次更新缓存数据）*(假设METADATA的Tablet没有被频繁的移动）。尽管Tablet的地址信息是存放在内存里的，对它的操作不必访问GFS文件系统，但是，通常我们会通过预取Tablet地址来进一步的减少访问的开销：每次需要从METADATA表中读取一个Tablet的元数据的时候，它都会多读取几个Tablet的元数据。

在METADATA表中还存储了次级信息*(alex注：secondary information)*，包括每个Tablet的事件日志（例如，什么时候一个服务器开始为该Tablet提供服务）。这些信息有助于排查错误和性能分析。

5.2 Tablet分配

在任何一个时刻，一个Tablet只能分配给一个Tablet服务器。Master服务器记录了当前有哪些活跃的Tablet服务器、哪些Tablet分配给了哪些Tablet服务器、哪些Tablet还没有被分配。当一个Tablet还没有被分配、并且刚好有一个Tablet服务器有足够的空闲空间装载该Tablet时，Master服务器会给这个Tablet服务器发送一个装载请求，把Tablet分配给这个服务器。

BigTable使用Chubby跟踪记录Tablet服务器的状态。当一个Tablet服务器启动时，它在Chubby的一个指定目录下建立一个有唯一性名字的文件，并且获取该文件的独占锁。Master服务器实时监控着这个目录（服务器目录），因此Master服务器能够知道有新的Tablet服务器加入了。如果Tablet服务器丢失了Chubby上的独占锁 — 比如由于网络断开导致Tablet服务器和Chubby的会话丢失 — 它就停止对Tablet提供服务。（Chubby提供了一种高效的机制，利用这种机制，Tablet服务器能够在不增加网络负担的情况下知道它是否还持有锁）。只要文件还存在，Tablet服务器就会试图重新获得对该文件的独占锁；如果文件不存在了，那么Tablet服务器就不能再提供服务了，它会自行退出*（alex注：so it kills itself）*。当Tablet服务器终止时（比如，集群的管理系统将运行该Tablet服务器的主机从集群中移除），它会尝试释放它持有的文件锁，这样一来，Master服务器就能尽快把Tablet分配到其它的Tablet服务器。

Master服务器负责检查一个Tablet服务器是否已经不再为它的Tablet提供服务了，并且要尽快重新分配它加载的Tablet。Master服务器通过轮询Tablet服务器文件锁的状态来检测何时Tablet服务器不再为Tablet提供服务。如果一个Tablet服务器报告它丢失了文件锁，或者Master服务器最近几次尝试和它通信都没有得到响应，Master服务器就会尝试获取该Tablet服务器文件的独占锁；如果Master服务器成功获取了独占锁，那么就说明Chubby是正常运行的，而Tablet服务器要么是宕机了、要么是不能和Chubby通信了，因此，Master服务器就删除该Tablet服务器在Chubby上的服务器文件以确保它不再给Tablet提供服务。一旦Tablet服务器在Chubby上的服务器文件被删除了，Master服务器就把之前分配给它的所有的Tablet放入未分配的Tablet集合中。为了确保Bigtable集群在Master服务器和Chubby之间网络出现故障的时候仍然可以使用，Master服务器在它的Chubby会话过期后主动退出。但是不管怎样，如同我们前面所描述的，Master服务器的故障不会改变现有Tablet在Tablet服务器上的分配状态。

当集群管理系统启动了一个Master服务器之后，Master服务器首先要了解当前Tablet的分配状态，之后才能够修改分配状态。Master服务器在启动的时候执行以下步骤：（1）Master服务器从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；（2）Master服务器扫描Chubby的服务器文件锁存储目录，获取当前正在运行的服务器列表；（3）Master服务器和所有的正在运行的Tablet表服务器通信，获取每个Tablet服务器上Tablet的分配信息；（4）Master服务器扫描METADATA表获取所有的Tablet的集合。在扫描的过程中，当Master服务器发现了一个还没有分配的Tablet，Master服务器就将这个Tablet加入未分配的Tablet集合等待合适的时机分配。

可能会遇到一种复杂的情况：在METADATA表的Tablet还没有被分配之前是不能够扫描它的。因此，在开始扫描之前（步骤4），如果在第三步的扫描过程中发现Root Tablet还没有分配，Master服务器就把Root Tablet加入到未分配的Tablet集合。这个附加操作确保了Root Tablet会被分配。由于Root Tablet包括了所有METADATA的Tablet的名字，因此Master服务器扫描完Root Tablet以后，就得到了所有的

METADATA表的Tablet的名字了。

保存现有Tablet的集合只有在以下事件发生时才会改变：建立了一个新表或者删除了一个旧表、两个Tablet被合并了、或者一个Tablet被分割成两个小的Tablet。Master服务器可以跟踪记录所有这些事件，因为除了最后一个事件外的两个事件都是由它启动的。Tablet分割事件需要特殊处理，因为它是由Tablet服务器启动。在分割操作完成之后，Tablet服务器通过在METADATA表中记录新的Tablet的信息来提交这个操作；当分割操作提交之后，Tablet服务器会通知Master服务器。如果分割操作已提交的信息没有通知到Master服务器（可能两个服务器中有一个宕机了），Master服务器在要求Tablet服务器装载已经被分割的子表的时候会发现一个新的Tablet。通过对比METADATA表中Tablet的信息，Tablet服务器会发现Master服务器要求其装载的Tablet并不完整，因此，Tablet服务器会重新向Master服务器发送通知信息。

5.3 Tablet服务

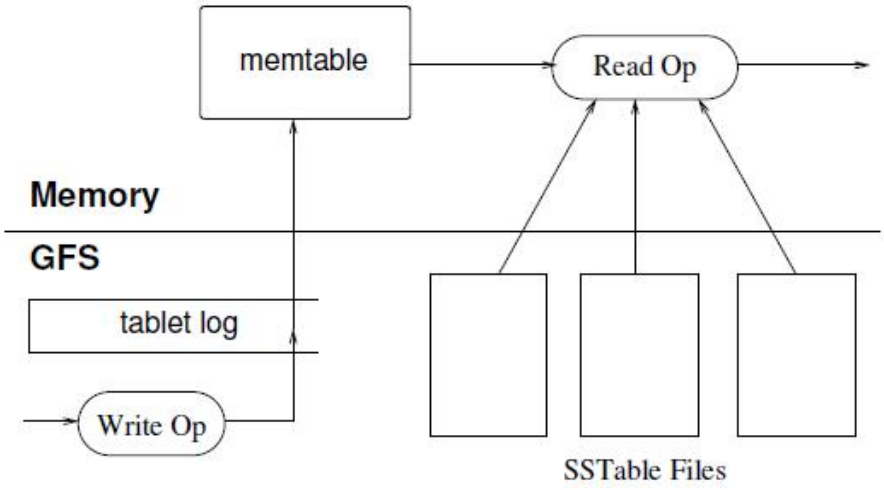


Figure 5: Tablet Representation

如图5所示，Tablet的持久化状态信息保存在GFS上。更新操作提交到REDO日志中*（alex注：Updates are committed to a commit log that stores redo records）*。在这些更新操作中，最近提交的那些存放在一个排序的缓存中，我们称这个缓存为memtable；较早的更新存放在一系列SSTable中。为了恢复一个Tablet，Tablet服务器首先从METADATA表中读取它的元数据。Tablet的元数据包含了组成这个Tablet的SSTable的列表，以及一系列的Redo Point*（alex注：a set of redo points）*，这些Redo Point指向可能含有该Tablet数据的已提交的日志记录。Tablet服务器把SSTable的索引读进内存，之后通过重复Redo Point之后提交的更新来重建memtable。

当对Tablet服务器进行写操作时，Tablet服务器首先要检查这个操作格式是否正确、操作发起者是否有执行这个操作的权限。权限验证的方法是通过从一个Chubby文件里读取出来的具有写权限的操作者列表来进行验证（这个文件几乎一定会存放在Chubby客户缓存里）。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*（alex注：group commit）*来提高包含大量小的修改操作的应用程序的吞吐量【13，16】。当一个写操作提交后，写的内容插入到memtable里面。

当对Tablet服务器进行读操作时，Tablet服务器会作类似的完整性和权限检查。一个有效的读操作在一个由一系列SSTable和memtable合并的视图里执行。由于SSTable和memtable是按字典排序的数据结构，因此可以高效生成合并视图。

当进行Tablet的合并和分割时，正在进行的读写操作能够继续进行。

5.4 Compactions

(alex注：这个词挺简单，但是在这节里面挺难翻译的。应该是空间缩减的意思，但是似乎又不能完全概括它在上下文中的意思，干脆，不翻译了)

随着写操作的执行，memtable的大小不断增加。当memtable的尺寸到达一个门限值的时候，这个memtable就会被冻结，然后创建一个新的memtable；被冻结住memtable会被转换成SSTable，然后写入GFS *（alex注：我们称这种Compaction行为为Minor Compaction）*。Minor Compaction过程有两个目的：shrink*(alex注：shrink是数据库用语，表示空间收缩)*Tablet服务器使用的内存，以及在服务器灾难恢复过程中，减少必须从提交日志里读取的数据量。在Compaction过程中，正在进行的读写操作仍能继续。

每一次Minor Compaction都会创建一个新的SSTable。如果Minor Compaction过程不停滞的持续进行下去，读操作可能需要合并来自多个SSTable的更新；否则，我们通过定期在后台执行Merging Compaction过程合并文件，限制这类文件的数量。Merging Compaction过程读取一些SSTable和memtable的内容，合并成一个新的SSTable。只要Merging Compaction过程完成了，输入的这些SSTable和memtable就可以删除了。

合并所有的SSTable并生成一个新的SSTable的Merging Compaction过程叫作Major Compaction。由非Major Compaction产生的SSTable可能含有特殊的删除条目，这些删除条目能够隐藏在旧的、但是依然有效的SSTable中已经删除的数据 *（alex注：令人费解啊，原文是SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live）*。而Major Compaction过程生成的SSTable不包含已经删除的信息或数据。Bigtable循环扫描它所有的Tablet，并且定期对它们执行Major Compaction。Major Compaction机制允许Bigtable回收已经删除的数据占有的资源，并且确保BigTable能及时清除已经删除的数据 *（alex注：实际是回收资源。数据删除后，它占有的空间并不能马上重复利用；只有空间回收后才能重复使用），*这对存放敏感数据的服务是非常重要。

6 优化

上一章我们描述了Bigtable的实现，我们还需要很多优化工作才能使Bigtable到达用户要求的高性能、高可用性和高可靠性。本章描述了Bigtable实现的其它部分，为了更好的强调这些优化工作，我们将深入细节。

局部性群组

客户程序可以将多个列族组合成一个局部性群族。对Tablet中的每个局部性群组都会生成一个单独的SSTable。将通常不会一起访问的列族分割成不同的局部性群组可以提高读取操作的效率。例如，在Webtable表中，网页的元数据（比如语言和Checksum）可以在一个局部性群组中，网页的内容可以在另外一个群组：当一个应用程序要读取网页的元数据的时候，它没有必要去读取所有的页面内容。

此外，可以以局部性群组为单位设定一些有用的调试参数。比如，可以把一个局部性群组设定为全部存储在内存中。Tablet服务器依照惰性加载的策略将设定为放入内存的局部性群组的SSTable装载进内存。加载完成之后，访问属于该局部性群组的列族的时候就不必读取硬盘了。这个特性对于需要频繁访问的小块数据特别有用：在Bigtable内部，我们利用这个特性提高METADATA表中具有位置相关性的列族的访问速度。

压缩

客户程序可以控制一个局部性群组的SSTable是否需要压缩；如果需要压缩，那么以什么格式来压缩。每个SSTable的块（块的大小由局部性群组的优化参数指定）都使用用户指定的压缩格式来压缩。虽然分块压缩浪费了少量空间 *（alex注：相比于对整个SSTable进行压缩，分块压缩压缩率较低），*但是，我们在只读取SSTable的一小部分数据的时候就不必解压整个文件了。很多客户程序使用了“两遍”的、可定制的压缩方式。第一遍采用Bentley and McIlroy’s方式[6]，这种方式在一个很大的扫描窗口里对常见的长字符串进行压缩；第二遍是采用快速压缩算法，即在一个16KB的小扫描窗口中寻找重复数据。两个压缩的算法都很快，在现在的机器上，压缩的速率达到100-200MB/s，解压的速率达到400-1000MB/s。

虽然我们在选择压缩算法的时候重点考虑的是速度而不是压缩的空间，但是这种两遍的压缩方式在空间压缩率上的表现也是令人惊叹。比如，在Webtable的例子里，我们使用这种压缩方式来存储网页内容。在

一次测试中，我们在一个压缩的局部性群组中存储了大量的网页。针对实验的目的，我们没有存储每个文档所有版本的数据，我们仅仅存储了一个版本的数据。该模式的空间压缩比达到了10:1。这比传统的Gzip在压缩HTML页面时3:1或者4:1的空间压缩比好的多；“两遍”的压缩模式如此高效的原因是由于Webtable的行的存放方式：从同一个主机获取的页面都存在临近的地方。利用这个特性，Bentley-McIlroy算法可以从来自同一个主机的页面里找到大量的重复内容。不仅仅是Webtable，其它的很多应用程序也通过选择合适的行名来将相似的数据聚簇在一起，以获取较高的压缩率。当我们在Bigtable中存储同一份数据的多个版本的时候，压缩效率会更高。

通过缓存提高读操作的性能

为了提高读操作的性能，Tablet服务器使用二级缓存的策略。扫描缓存是第一级缓存，主要缓存Tablet服务器通过SSTable接口获取的Key-Value对；Block缓存是二级缓存，缓存的是从GFS读取的SSTable的Block。对于经常要重复读取相同数据的应用程序来说，扫描缓存非常有效；对于经常要读取刚刚读过的数据附近的数据的应用程序来说，Block缓存更有用（例如，顺序读，或者在一个热点的行的局部性群组中随机读取不同的列）。

***Bloom**过滤器*

(alex注：Bloom，又叫布隆过滤器，什么意思？请参考Google黑板报http://googlechinablog.com/2007/07/bloom-filter.html请务必先认真阅读)

如5.3节所述，一个读操作必须读取构成Tablet状态的所有SSTable的数据。如果这些SSTable不在内存中，那么就需要多次访问硬盘。我们通过允许客户程序对特定局部性群组的SSTable指定Bloom过滤器【7】，来减少硬盘访问的次数。我们可以使用Bloom过滤器查询一个SSTable是否包含了特定行和列的数据。对于某些特定应用程序，我们只付出了少量的、用于存储Bloom过滤器的内存的代价，就换来了读操作显著减少的磁盘访问的次数。使用Bloom过滤器也隐式的达到了当应用程序访问不存在的行或列时，大多数时候我们都不需要访问硬盘的目的。

***Commit**日志的实现*

如果我们把对每个Tablet的操作的Commit日志都存在一个单独的文件的话，那么就会产生大量的文件，并且这些文件会并行的写入GFS。根据GFS服务器底层文件系统实现的方案，要把这些文件写入不同的磁盘日志文件时*(alex注：different physical log files)*，会有大量的磁盘Seek操作。另外，由于批量提交*(alex注：group commit)*中操作的数目一般比较少，因此，对每个Tablet设置单独的日志文件也会给批量提交本应具有的优化效果带来很大的负面影响。为了避免这些问题，我们设置每个Tablet服务器一个Commit日志文件，把修改操作的日志以追加方式写入同一个日志文件，因此一个实际的日志文件中混合了对多个Tablet修改的日志记录。

使用单个日志显著提高了普通操作的性能，但是将恢复的工作复杂化了。当一个Tablet服务器宕机时，它加载的Tablet将会被移到很多其它的Tablet服务器上：每个Tablet服务器都装载很少的几个原来的服务器的Tablet。当恢复一个Tablet的状态的时候，新的Tablet服务器要从原来的Tablet服务器写的日志中提取修改操作的信息，并重新执行。然而，这些Tablet修改操作的日志记录都混合在同一个日志文件中的。一种方法新的Tablet服务器读取完整的Commit日志文件，然后只重复执行它需要恢复的Tablet的相关修改操作。使用这种方法，假如有100台Tablet服务器，每台都加载了失效的Tablet服务器上的一个Tablet，那么，这个日志文件就要被读取100次（每个服务器读取一次）。

为了避免多次读取日志文件，我们首先把日志按照关键字（table，row name，log sequence number）排序。排序之后，对同一个Tablet的修改操作的日志记录就连续存放在了一起，因此，我们只要一次磁盘Seek操作、之后顺序读取就可以了。为了并行排序，我们先将日志分割成64MB的段，之后在不同的Tablet服务器对段进行并行排序。这个排序工作由Master服务器来协同处理，并且在一个Tablet服务器表明自己需要从Commit日志文件恢复Tablet时开始执行。

在向GFS中写Commit日志的时候可能会引起系统颠簸，原因是多种多样的（比如，写操作正在进行的时候，一个GFS服务器宕机了；或者连接三个GFS副本所在的服务器的网络拥塞或者过载了）。为了确保在GFS负载高峰时修改操作还能顺利进行，每个Tablet服务器实际上有两个日志写入线程，每个线程都写自己的日志文件，并且在任何时刻，只有一个线程是工作的。如果一个线程的在写入的时候效率很低，Tablet服务器就切换到另外一个线程，修改操作的日志记录就写入到这个线程对应的日志文件中。每

个日志记录都有一个序列号，因此，在恢复的时候，Tablet服务器能够检测出并忽略掉那些由于线程切换而导致的重复的记录。

***Tablet**恢复提速*

当Master服务器将一个Tablet从一个Tablet服务器移到另外一个Tablet服务器时，源Tablet服务器会对这个Tablet做一次Minor Compaction。这个Compaction操作减少了Tablet服务器的日志文件中没有归并的记录，从而减少了恢复的时间。Compaction完成之后，该服务器就停止为该Tablet提供服务。在卸载Tablet之前，源Tablet服务器还会再做一次（通常会很快）Minor Compaction，以消除前面在一次压缩过程中又产生的未归并的记录。第二次Minor Compaction完成以后，Tablet就可以被装载到新的Tablet服务器上了，并且不需要从日志中进行恢复。

利用不变性

我们在使用Bigtable时，除了SSTable缓存之外的其它部分产生的SSTable都是不变的，我们可以利用这一点对系统进行简化。例如，当从SSTable读取数据的时候，我们不必对文件系统访问操作进行同步。这样一来，就可以非常高效的实现对行的并行操作。memtable是唯一一个能被读和写操作同时访问的可变数据结构。为了减少在读操作时的竞争，我们对内存表采用COW(Copy-on-write)机制，这样就允许读写操作并行执行。

因为SSTable是不变的，因此，我们可以把永久删除被标记为“删除”的数据的问题，转换成对废弃的SSTable进行垃圾收集的问题了。每个Tablet的SSTable都在METADATA表中注册了。Master服务器采用“标记-删除”的垃圾回收方式删除SSTable集合中废弃的SSTable【25】，METADATA表则保存了Root SSTable的集合。

最后，SSTable的不变性使得分割Tablet的操作非常快捷。我们不必为每个分割出来的Tablet建立新的SSTable集合，而是共享原来的Tablet的SSTable集合。

7 性能评估

为了测试Bigtable的性能和可扩展性，我们建立了一个包括N台Tablet服务器的Bigtable集群，这里N是可变的。每台Tablet服务器配置了1GB的内存，数据写入到一个包括1786台机器、每台机器有2个IDE硬盘的GFS集群上。我们使用N台客户机生成工作负载测试Bigtable。（我们使用和Tablet服务器相同数目的客户机以确保客户机不会成为瓶颈。） 每台客户机配置2GZ双核Opteron处理器，配置了足以容纳所有进程工作数据集的物理内存，以及一张Gigabit的以太网卡。这些机器都连入一个两层的、树状的交换网络里，在根节点上的带宽加起来有大约100-200Gbps。所有的机器采用相同的设备，因此，任何两台机器间网络来回一次的时间都小于1ms。

Tablet服务器、Master服务器、测试机、以及GFS服务器都运行在同一组机器上。每台机器都运行一个GFS的服务器。其它的机器要么运行Tablet服务器、要么运行客户程序、要么运行在测试过程中，使用这组机器的其它的任务启动的进程。

R是测试过程中，Bigtable包含的不同的列关键字的数量。我们精心选择R的值，保证每次基准测试对每台Tablet服务器读/写的数据量都在1GB左右。

在序列写的基准测试中，我们使用的列关键字的范围是0到R-1。这个范围又被划分为10N个大小相同的区间。核心调度程序把这些区间分配给N个客户端，分配方式是：只要客户程序处理完上一个区间的数据，调度程序就把后续的、尚未处理的区间分配给它。这种动态分配的方式有助于减少客户机上同时运行的其它进程对性能的影响。我们在每个列关键字下写入一个单独的字符串。每个字符串都是随机生成的、因此也没有被压缩*（alex注：参考第6节的压缩小节）*。另外，不同列关键字下的字符串也是不同的，因此也就不存在跨行的压缩。随机写入基准测试采用类似的方法，除了行关键字在写入前先做Hash，Hash采用按R取模的方式，这样就保证了在整个基准测试持续的时间内，写入的工作负载均匀的分布在列存储空间内。

序列读的基准测试生成列关键字的方式与序列写相同，不同于序列写在列关键字下写入字符串的是，序列

读是读取列关键字下的字符串（这些字符串由之前序列写基准测试程序写入）。同样的，随机读的基准测试和随机写是类似的。

扫描基准测试和序列读类似，但是使用的是BigTable提供的、从一个列范围内扫描所有的value值的API。由于一次RPC调用就从一个Tablet服务器取回了大量的Value值，因此，使用扫描方式的基准测试程序可以减少RPC调用的次数。

随机读（内存）基准测试和随机读类似，除了包含基准测试数据的局部性群组被设置为“in-memory”，因此，读操作直接从Tablet服务器的内存中读取数据，不需要从GFS读取数据。针对这个测试，我们把每台Tablet服务器存储的数据从1GB减少到100MB，这样就可以把数据全部加载到Tablet服务器的内存中了。

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

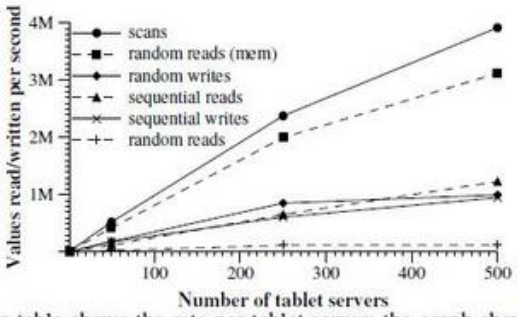


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

图6中有两个视图，显示了我们的基准测试的性能；图中的数据和曲线是读/写 1000-byte value值时取得的。图中的表格显示了每个Tablet服务器每秒钟进行的操作的次数；图中的曲线显示了每秒种所有的Tablet服务器上操作次数的总和。

*单个**Tablet**服务器的性能*

我们首先分析下单个Tablet服务器的性能。随机读的性能比其它操作慢一个数量级或以上*（alex注：by the order of magnitude or more）*。每个随机读操作都要通过网络从GFS传输64KB的SSTable到Tablet服务器，而我们只使用其中大小是1000 byte的一个value值。Tablet服务器每秒大约执行1200次读操作，也就是每秒大约从GFS读取75MB的数据。这个传输带宽足以占满Tablet服务器的CPU时间，因为其中包括了网络协议栈的消耗、SSTable解析、以及BigTable代码执行；这个带宽也足以占满我们系统中网络的链接带宽。大多数采用这种访问模式BigTable应用程序会减小Block的大小，通常会减到8KB。

内存中的随机读操作速度快很多，原因是，所有1000-byte的读操作都是从Tablet服务器的本地内存中读取数据，不需要从GFS读取64KB的Block。

随机和序列写操作的性能比随机读要好些，原因是每个Tablet服务器直接把写入操作的内容追加到一个Commit日志文件的尾部，并且采用批量提交的方式，通过把数据以流的方式写入到GFS来提高性能。随机写和序列写在性能上没有太大的差异，这两种方式的写操作实际上都是把操作内容记录到同一个Tablet服务器的Commit日志文件中。

序列读的性能好于随机读，因为每取出64KB的SSTable的Block后，这些数据会缓存到Block缓存中，后续的64次读操作直接从缓存读取数据。

扫描的性能更高，这是由于客户程序每一次RPC调用都会返回大量的value的数据，所以，RPC调用的消耗基本抵消了。

性能提升

随着我们将系统中的Tablet服务器从1台增加到500台，系统的整体吞吐量有了梦幻般的增长，增长的倍率超过了100。比如，随着Tablet服务器的数量增加了500倍，内存中的随机读操作的性能增加了300倍。之所以会有这样的性能提升，主要是因为这个基准测试的瓶颈是单台Tablet服务器的CPU。

尽管如此，性能的提升还不是线性的。在大多数的基准测试中我们看到，当Tablet服务器的数量从1台增加到50台时，每台服务器的吞吐量会有一个明显的下降。这是由于多台服务器间的负载不均衡造成的，大多数情况下是由于其它的程序抢占了CPU。 我们负载均衡的算法会尽量避免这种不均衡，但是基于两个主要原因，这个算法并不能完美的工作：一个是尽量减少Tablet的移动导致重新负载均衡能力受限（如果Tablet被移动了，那么在短时间内 — 一般是1秒内 — 这个Tablet是不可用的），另一个是我们的基准测试程序产生的负载会有波动*（alex注：the load generated by our benchmarks shifts around as the benchmark progresses）*。

随机读基准测试的测试结果显示，随机读的性能随Tablet服务器数量增加的提升幅度最小（整体吞吐量只提升了100倍，而服务器的数量却增加了500倍）。这是因为每个1000-byte的读操作都会导致一个64KB大的Block在网络上传输。这样的网络传输量消耗了我们网络中各种共享的1GB的链路，结果导致随着我们增加服务器的数量，每台服务器上的吞吐量急剧下降。

8 实际应用

# of tablet servers	# of clusters
0 .. 19	259
20 .. 49	47
50 .. 99	20
100 .. 499	50
> 500	12

Table 1: Distribution of number of tablet servers in Bigtable clusters.

截止到2006年8月，Google内部一共有388个非测试用的Bigtable集群运行在各种各样的服务器集群上，合计大约有24500个Tablet服务器。表1显示了每个集群上Tablet服务器的大致分布情况。这些集群中，许多用于开发目的，因此会有一段时期比较空闲。通过观察一个由14个集群、8069个Tablet服务器组成的集群组，我们看到整体的吞吐量超过了每秒1200000次请求，发送到系统的RPC请求导致的网络负载达到了741MB/s，系统发出的RPC请求网络负载大约是16GB/s。

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

表2提供了一些目前正在使用的表的相关数据。一些表存储的是用户相关的数据，另外一些存储的则是用于批处理的数据；这些表在总的大小、 每个数据项的平均大小、从内存中读取的数据的比例、表的Schema的复杂程度上都有很大的差别。本节的其余部分，我们将主要描述三个产品研发团队如何使用Bigtable的。

8.1 Google Analytics

Google Analytics是用来帮助Web站点的管理员分析他们网站的流量模式的服务。它提供了整体状况的统

计数据，比如每天的独立访问的用户数量、每天每个URL的浏览次数；它还提供了用户使用网站的行为报告，比如根据用户之前访问的某些页面，统计出几成的用户购买了商品。

为了使用这个服务，Web站点的管理员只需要在他们的Web页面中嵌入一小段JavaScript脚本就可以了。这个Javascript程序在页面被访问的时候调用。它记录了各种Google Analytics需要使用的信息，比如用户的标识、获取的网页的相关信息。Google Analytics汇总这些数据，之后提供给Web站点的管理员。

我们粗略的描述一下Google Analytics使用的两个表。Row Click表（大约有200TB数据）的每一行存放了一个最终用户的会话。行的名字是一个包含Web站点名字以及用户会话创建时间的元组。这种模式保证了对同一个Web站点的访问会话是顺序的，会话按时间顺序存储。这个表可以压缩到原来尺寸的14%。

Summary表（大约有20TB的数据）包含了关于每个Web站点的、各种类型的预定义汇总信息。一个周期性运行的MapReduce任务根据Raw Click表的数据生成Summary表的数据。每个MapReduce工作进程都从Raw Click表中提取最新的会话数据。系统的整体吞吐量受限于GFS的吞吐量。这个表的能够压缩到原有尺寸的29%。

8.2 Google Earth

Google通过一组服务为用户提供了高分辨率的地球表面卫星图像，访问的方式可以使通过基于Web的Google Maps访问接口（maps.google.com），也可以通过Google Earth定制的客户端软件访问。这些软件产品允许用户浏览地球表面的图像：用户可以在不同的分辨率下平移、查看和注释这些卫星图像。这个系统使用一个表存储预处理数据，使用另外一组表存储用户数据。

数据预处理流水线使用一个表存储原始图像。在预处理过程中，图像被清除，图像数据合并到最终的服务数据中。这个表包含了大约70TB的数据，所以需要从磁盘读取数据。图像已经被高效压缩过了，因此存储在Bigtable后不需要再压缩了。

Imagery表的每一行都代表了一个单独的地理区域。行都有名称，以确保毗邻的区域存储在了一起。Imagery表中有一个列族用来记录每个区域的数据源。这个列族包含了大量的列：基本上市每个列对应一个原始图片的数据。由于每个地理区域都是由很少的几张图片构成的，因此这个列族是非常稀疏的。

数据预处理流水线高度依赖运行在Bigtable上的MapReduce任务传输数据。在运行某些MapReduce任务的时候，整个系统中每台Tablet服务器的数据处理速度是1MB/s。

这个服务系统使用一个表来索引GFS中的数据。这个表相对较小（大约是500GB），但是这个表必须在保证较低的响应延时的前提下，针对每个数据中心，每秒处理几万个查询请求。 因此，这个表必须在上百个Tablet服务器上存储数据，并且使用in-memory的列族。

8.3 个性化查询

个性化查询（www.google.com/psearch）是一个双向服务；这个服务记录用户的查询和点击，涉及到各种Google的服务，比如Web查询、图像和新闻。用户可以浏览他们查询的历史，重复他们之前的查询和点击；用户也可以定制基于Google历史使用习惯模式的个性化查询结果。

个性化查询使用Bigtable存储每个用户的数据。每个用户都有一个唯一的用户id，每个用户id和一个列名绑定。一个单独的列族被用来存储各种类型的行为（比如，有个列族可能是用来存储所有的Web查询的）。每个数据项都被用作Bigtable的时间戳，记录了相应的用户行为发生的时间。个性化查询使用以Bigtable为存储的MapReduce任务生成用户的数据图表。这些用户数据图表用来个性化当前的查询结果。

个性化查询的数据会复制到几个Bigtable的集群上，这样就增强了数据可用性，同时减少了由客户端和Bigtable集群间的“距离”造成的延时。个性化查询的开发团队最初建立了一个基于Bigtable的、“客户侧”的复制机制为所有的复制节点提供一致性保障。现在的系统则使用了内建的复制子系统。

个性化查询存储系统的设计允许其它的团队在它们自己的列中加入新的用户数据，因此，很多Google服务使用个性化查询存储系统保存用户级的配置参数和设置。在多个团队之间分享数据的结果是产生了大量的列族。为了更好的支持数据共享，我们加入了一个简单的配额机制*(alex注：quota，参考AIX的配额机制)*

限制用户在共享表中使用的空间；配额也为使用个性化查询系统存储用户级信息的产品团体提供了隔离机制。

9 经验教训

在设计、实现、维护和支持Bigtable的过程中，我们得到了很多有用的经验和一些有趣的教训。

一个教训是，我们发现，很多类型的错误都会导致大型分布式系统受损，这些错误不仅仅是通常的网络中断、或者很多分布式协议中设想的fail-stop类型的错误*（alex注：fail-stop failture，指一旦系统fail就stop，不输出任何数据；fail-fast failture，指fail不马上stop，在短时间内return错误信息，然后再stop）*。比如，我们遇到过下面这些类型的错误导致的问题：内存数据损坏、网络中断、时钟偏差、机器挂起、扩展的和非对称的网络分区*（alex注：extended and asymmetric network partitions，不明白什么意思。partition也有中断的意思，但是我不知道如何用在这里）*、我们使用的其它系统的Bug（比如Chubby）、GFS配额溢出、计划内和计划外的硬件维护。我们在解决这些问题的过程中学到了很多经验，我们通过修改协议来解决这些问题。比如，我们在我们的RPC机制中加入了Checksum。我们在设计系统的部分功能时，不对其它部分功能做任何的假设，这样的做法解决了其它的一些问题。比如，我们不再假设一个特定的Chubby操作只返回错误码集合中的一个值。

另外一个教训是，我们明白了在彻底了解一个新特性会被如何使用之后，再决定是否添加这个新特性是非常重要的。比如，我们开始计划在我们的API中支持通常方式的事务处理。但是由于我们还不会马上用到这个功能，因此，我们并没有去实现它。现在，Bigtable上已经有了很多的实际应用，我们可以检查它们真实的需求；我们发现，大多是应用程序都只是需要单个行上的事务功能。有些应用需要分布式的事务功能，分布式事务大多数情况下用于维护二级索引，因此我们增加了一个特殊的机制去满足这个需求。新的机制在通用性上比分布式事务差很多，但是它更有效（特别是在更新操作的涉及上百行数据的时候），而且非常符合我们的“跨数据中心”复制方案的优化策略。

还有一个具有实践意义的经验：我们发现系统级的监控对Bigtable非常重要（比如，监控Bigtable自身以及使用Bigtable的客户程序）。比如，我们扩展了我们的RPC系统，因此对于一个RPC调用的例子，它可以详细记录代表了RPC调用的很多重要操作。这个特性允许我们检测和修正很多的问题，比如Tablet数据结构上的锁的内容、在修改操作提交时对GFS的写入非常慢的问题、以及在METADATA表的Tablet不可用时，对METADATA表的访问挂起的问题。关于监控的用途的另外一个例子是，每个Bigtable集群都在Chubby中注册了。这可以帮助我们跟踪所有的集群状态、监控它们的大小、检查集群运行的我们软件的版本、监控集群流入数据的流量，以及检查是否有引发集群高延时的潜在因素。

对我们来说，最宝贵的经验是简单设计的价值。考虑到我们系统的代码量（大约100000行生产代码*（alex注：non-test code）*），以及随着时间的推移，新的代码以各种难以预料的方式加入系统，我们发现简洁的设计和编码给维护和调试带来的巨大好处。这方面的一个例子是我们的Tablet服务器成员协议。我们第一版的协议很简单：Master服务器周期性的和Tablet服务器签订租约，Tablet服务器在租约过期的时候Kill掉自己的进程。不幸的是，这个协议在遇到网络问题时会大大降低系统的可用性，也会大大增加Master服务器恢复的时间。我们多次重新设计这个协议，直到它能够很好的处理上述问题。但是，更不幸的是，最终的协议过于复杂了，并且依赖一些Chubby很少被用到的特性。我们发现我们浪费了大量的时间在调试一些古怪的问题*(alex注：obscure corner cases)*，有些是Bigtable代码的问题，有些事Chubby代码的问题。最后，我们只好废弃了这个协议，重新制订了一个新的、更简单、只使用Chubby最广泛使用的特性的协议。

10 相关工作

Boxwood【24】项目的有些组件在某些方面和Chubby、GFS以及Bigtable类似，因为它也提供了诸如分布式协议、锁、分布式Chunk存储以及分布式B-tree存储。Boxwood与Google的某些组件尽管功能类似，但是Boxwood的组件提供更底层的服务。Boxwood项目的目的是提供创建类似文件系统、数据库等高级服务的基础构件，而Bigtable的目的是直接为客户程序的数据存储需求提供支持。

现在有不少项目已经攻克了很多难题，实现了在广域网上的分布式数据存储或者高级服务，通常是“Internet规模”的。这其中包括了分布式的Hash表，这项工作由一些类似CAN【29】、Chord【32】、Tapestry【37】和Pastry【30】的项目率先发起。这些系统的主要关注点和Bigtable不同，比如应对各

种不同的传输带宽、不可信的协作者、频繁的更改配置等；另外，去中心化和Byzantine灾难冗余*(alex注：Byzantine，即拜占庭式的风格，也就是一种复杂诡秘的风格。Byzantine Fault表示：对于处理来说，当发错误时处理器并不停止接收输出，也不停止输出，错就错了，只管算，对于这种错误来说，这样可真是够麻烦了，因为用户根本不知道错误发生了，也就根本谈不上处理错误了。在多处理器的情况下，这种错误可能导致运算正确结果的处理器也产生错误的结果，这样事情就更麻烦了，所以一定要避免处理器产生这种错误。)*也不是Bigtable的目的。

就提供给应用程序开发者的分布式数据存储模型而言，我们相信，分布式B-Tree或者分布式Hash表提供的Key-value pair方式的模型有很大的局限性。Key-value pair模型是很有用的组件，但是它们不应该是提供给开发者唯一的组件。我们选择的模型提供的组件比简单的Key-value pair丰富的多，它支持稀疏的、半结构化的数据。另外，它也足够简单，能够高效的处理平面文件；它也是透明的（通过局部性群组），允许我们的使用者对系统的重要行为进行调整。

有些数据库厂商已经开发出了并行的数据库系统，能够存储海量的数据。Oracle的RAC【27】使用共享磁盘存储数据（Bigtable使用GFS），并且有一个分布式的锁管理系统（Bigtable使用Chubby）。IBM并行版本的DB2【4】基于一种类似于Bigtable的、不共享任何东西的架构（a shared-nothing architecture）【33】。每个DB2的服务器都负责处理存储在一个关系型数据库中的表中的行的一个子集。这些产品都提供了一个带有事务功能的完整的关系模型。

Bigtable的局部性群组提供了类似于基于列的存储方案在压缩和磁盘读取方面具有的性能；这些以列而不是行的方式组织数据的方案包括C-Store【1，34】、商业产品Sybase IQ【15，36】、SenSage【31】、KDB+【22】，以及MonetDB/X100【38】的ColumnDM存储层。另外一种在平面文件中提供垂直和水平数据分区、并且提供很好的数据压缩率的系统是AT&T的Daytona数据库【19】。局部性群组不支持Ailamaki系统中描述的CPU缓存级别的优化【2】。

Bigtable采用memtable和SSTable存储对表的更新的方法与Log-Structured Merge Tree【26】存储索引数据更新的方法类似。这两个系统中，排序的数据在写入到磁盘前都先存放在内存中，读取操作必须从内存和磁盘中合并数据产生最终的结果集。

C-Store和Bigtable有很多相似点：两个系统都采用Shared-nothing架构，都有两种不同的数据结构，一种用于当前的写操作，另外一种存放“长时间使用”的数据，并且提供一种机制在两个存储结构间搬运数据。两个系统在API接口函数上有很大的不同：C-Store操作更像关系型数据库，而Bigtable提供了低层次的读写操作接口，并且设计的目标是能够支持每台服务器每秒数千次操作。C-Store同时也是个“读性能优化的关系型数据库”，而Bigtable对读和写密集型应用都提供了很好的性能。

Bigtable也必须解决所有的Shared-nothing数据库需要面对的、类型相似的一些负载和内存均衡方面的难题（比如，【11，35】）。我们的问题在某种程度上简单一些：（1）我们不需要考虑同一份数据可能有多个拷贝的问题，同一份数据可能由于视图或索引的原因以不同的形式表现出来；（2）我们让用户决定哪些数据应该放在内存里、哪些放在磁盘上，而不是由系统动态的判断；（3）我们的系统中没有复杂的查询执行或优化工作。

11 结论

我们已经讲述完了Bigtable，Google的一个分布式的结构化数据存储系统。Bigtable的集群从2005年4月开始已经投入使用了，在此之前，我们花了大约7人年设计和实现这个系统。截止到2006年4月，已经有超过60个项目使用Bigtable了。我们的用户对Bigtable提供的高性能和高可用性很满意，随着时间的推移，他们可以根据自己的系统对资源的需求增加情况，通过简单的增加机器，扩展系统的承载能力。

由于Bigtable提供的编程接口并不常见，一个有趣的问题是：我们的用户适应新的接口有多难？新的使用者有时不太确定使用Bigtable接口的最佳方法，特别是在他们已经习惯于使用支持通用事务的关系型数据库的接口的情况下。但是，Google内部很多产品都成功的使用了Bigtable的事实证明了，我们的设计在实践中行之有效。

我们现在正在对Bigtable加入一些新的特性，比如支持二级索引，以及支持多Master节点的、跨数据中心复制的Bigtable的基础构件。我们现在已经开始将Bigtable部署为服务供其它的产品团队使用，这样不同的产品团队就不需要维护他们自己的Bigtable集群了。随着服务集群的扩展，我们需要在Bigtable系统内

部处理更多的关于资源共享的问题了【3，5】。

最后，我们发现，建设Google自己的存储解决方案带来了很多优势。通过为Bigtable设计我们自己的数据模型，是我们的系统极具灵活性。另外，由于我们全面控制着Bigtable的实现过程，以及Bigtable使用到的其它的Google的基础构件，这就意味着我们在系统出现瓶颈或效率低下的情况时，能够快速的解决这些问题。

Acknowledgements

We thank the anonymous reviewers, David Nagle, and our shepherd Brad Calder, for their feedback on this paper.The Bigtable system has benefited greatly from the feedback of our many users within Google. In addition,we thank the following people for their contributions to Bigtable: Dan Aguayo, Sameer Ajmani, Zhifeng Chen,Bill Coughran, Mike Epstein, Healfdene Goguen, Robert Griesemer, Jeremy Hylton, Josh Hyman, Alex Khesin, Joanna Kulik, Alberto Lerner, Sherry Listgarten, Mike Maloney, Eduardo Pinheiro, Kathy Polizzi, Frank Yellin,and Arthur Zwiegincew.

References

[1] ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. C. Integrating compression and execution in columnoriented database systems. Proc. of SIGMOD (2006).
[2] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. Weaving relations for cache performance.In The VLDB Journal (2001), pp. 169-180.
[3] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In Proc. of the 3rd OSDI (Feb. 1999), pp. 45-58.
[4] BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO, H., JHINGRAN, A., PADMANABHAN, S., COPELAND,G. P., AND WILSON, W. G. DB2 parallel edition. IBM Systems Journal 34, 2 (1995), 292-322.
[5] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In Proc. of the 1st NSDI(Mar. 2004), pp. 253-266.
[6] BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In Data Compression Conference (1999), pp. 287-295.
[7] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. CACM 13, 7 (1970), 422-426.
[8] BURROWS, M. The Chubby lock service for looselycoupled distributed systems. In Proc. of the 7th OSDI (Nov. 2006).
[9] CHANDRA, T., GRIESEMER, R., AND REDSTONE, J.Paxos made live ? An engineering perspective. In Proc. of PODC (2007).
[10] COMER, D. Ubiquitous B-tree. Computing Surveys 11, 2 (June 1979), 121-137.
[11] COPELAND, G. P., ALEXANDER, W., BOUGHTER, E. E., AND KELLER, T. W. Data placement in Bubba. In Proc. of SIGMOD (1988), pp. 99-108.
[12] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In Proc. of the 6th OSDI (Dec. 2004), pp. 137-150.
[13] DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D. Implementation techniques for main memory database systems. In Proc. of SIGMOD (June 1984), pp. 1-8.
[14] DEWITT, D. J., AND GRAY, J. Parallel database systems: The future of high performance database systems. CACM 35, 6 (June 1992), 85-98.
[15] FRENCH, C. D. One size ts all database architectures do not work for DSS. In Proc. of SIGMOD (May 1995), pp. 449-450.
[16] GAWLICK, D., AND KINKADE, D. Varieties of concurrency control in IMS/VS fast path. Database Engineering Bulletin 8, 2 (1985), 3-10.
[17] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In Proc. of the

19th ACM SOSP (Dec.2003), pp. 29-43.
[18] GRAY, J. Notes on database operating systems. In Operating Systems ? An Advanced Course, vol. 60 of Lecture Notes in Computer Science. Springer-Verlag, 1978.
[19] GREER, R. Daytona and the fourth-generation language Cymbal. In Proc. of SIGMOD (1999), pp. 525-526.
[20] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In Proc. of the 11th SOSP (Dec. 1987), pp. 155-162.
[21] HARTMAN, J. H., AND OUSTERHOUT, J. K. The Zebra striped network file system. In Proc. of the 14th SOSP(Asheville, NC, 1993), pp. 29-43.
[22] KX.COM. kx.com/products/database.php. Product page.
[23] LAMPORT, L. The part-time parliament. ACM TOCS 16,2 (1998), 133-169.
[24] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In Proc. of the 6th OSDI (Dec. 2004), pp. 105-120.
[25] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. CACM 3, 4 (Apr. 1960), 184-195.
[26] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (1996), 351-385.
[27] ORACLE.COM. www.oracle.com/technology/products/database/clustering/index.html. Product page.
[28] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming Journal 13, 4 (2005), 227-298.
[29] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In Proc. of SIGCOMM (Aug. 2001), pp. 161-172.
[30] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. In Proc. of Middleware 2001(Nov. 2001), pp. 329-350.
[31] SENSAGE.COM. sensage.com/products-sensage.htm. Product page.
[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In Proc. of SIGCOMM (Aug. 2001), pp. 149-160.
[33] STONEBRAKER, M. The case for shared nothing. Database Engineering Bulletin 9, 1 (Mar. 1986), 4-9.
[34] STONEBRAKER,M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. C-Store: A columnoriented DBMS. In Proc. of VLDB (Aug. 2005), pp. 553-564.
[35] STONEBRAKER, M., AOKI, P. M., DEVINE, R., LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture for distributed data. In Proc. of the Tenth ICDE(1994), IEEE Computer Society, pp. 54-65.
[36] SYBASE.COM. www.sybase.com/products/databaseservers/sybaseiq. Product page.
[37] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, CS Division, UC Berkeley, Apr. 2001.
[38] ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. MonetDB/X100 ?A DBMS in the CPU cache. IEEE Data Eng. Bull. 28, 2 (2005), 17-22.

分类: 未分类 标签:

The Google File System中文版

2010年3月27日 blademaster 没有评论

The Google File System中文版

*译者：**alex***

摘要

我们设计并实现了Google GFS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虽然运行在廉价的普遍硬件设备上，但是它依然了提供灾难冗余的能力，为大量客户机提供了高性能的服务。

虽然GFS的设计目标与许多传统的分布式文件系统有很多相同之处，但是，我们的设计还是以我们对自己的应用的负载情况和技术环境的分析为基础的，不管现在还是将来，GFS和早期的分布式文件系统的设想都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计思路。

GFS完全满足了我们对存储的需求。GFS作为存储平台已经被广泛的部署在Google内部，存储我们的服务产生和处理的数据，同时还用于那些需要大规模数据集的研究和开发工作。目前为止，最大的一个集群利用数千台机器的数千个硬盘，提供了数百TB的存储空间，同时为数百个客户机服务。

在本论文中，我们展示了能够支持分布式应用的文件系统接口的扩展，讨论我们设计的许多方面，最后列出了小规模性能测试以及真实生产系统中性能相关数据。

分类和主题描述

D [4]: 3—D分布文件系统

常用术语

设计，可靠性，性能，测量

关键词

容错，可伸缩性，数据存储，集群存储

1. 简介

为了满足Google迅速增长的数据处理需求，我们设计并实现了Google文件系统(Google File System – GFS)。GFS与传统的分布式文件系统有着很多相同的设计目标，比如，性能、可伸缩性、可靠性以及可用性。但是，我们的设计还基于我们对我们自己的应用的负载情况和技术环境的观察的影响，不管现在还是将来，GFS和早期文件系统的假设都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计思路。

首先，组件失效被认为是常态事件，而不是意外事件。GFS包括几百甚至几千台普通的廉价设备组装的存

储机器，同时被相当数量的客户机访问。GFS组件的数量和质量导致在事实上，任何给定时间内都有可能发生某些组件无法工作，某些组件无法从它们目前的失效状态中恢复。我们遇到过各种各样的问题，比如应用程序bug、操作系统的bug、人为失误，甚至还有硬盘、内存、连接器、网络以及电源失效等造成的问题。所以，持续的监控、错误侦测、灾难冗余以及自动恢复的机制必须集成在GFS中。

其次，以通常的标准衡量，我们的文件非常巨大。数GB的文件非常普遍。每个文件通常都包含许多应用程序对象，比如web文档。当我们经常需要处理快速增长的、并且由数亿个对象构成的、数以TB的数据集时，采用管理数亿个KB大小的小文件的方式是非常不明智的，尽管有些文件系统支持这样的管理方式。因此，设计的假设条件和参数，比如I/O操作和Block的尺寸都需要重新考虑。

第三，绝大部分文件的修改是采用在文件尾部追加数据，而不是覆盖原有数据的方式。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，比如：数据分析程序扫描的超大的数据集；正在运行的应用程序生成的连续的数据流；存档的数据；由一台机器生成、另外一台机器处理的中间数据，这些中间数据的处理可能是同时进行的、也可能是后续才处理的。对于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，数据的追加操作是性能优化和原子性保证的主要考量因素。

第四，应用程序和文件系统API的协同设计提高了整个系统的灵活性。比如，我们放松了对GFS一致性模型的要求，这样就减轻了文件系统对应用程序的苛刻要求，大大简化了GFS的设计。我们引入了原子性的记录追加操作，从而保证多个客户端能够同时进行追加操作，不需要额外的同步操作来保证数据的一致性。本文后面还有对这些问题的细节的详细讨论。

Google已经针对不同的应用部署了多套GFS集群。最大的一个集群拥有超过1000个存储节点，超过300TB的硬盘空间，被不同机器上的数百个客户端连续不断的频繁访问。

2.设计概述

2.1设计预期

在设计满足我们需求的文件系统时候，我们的设计目标既有机会、又有挑战。之前我们已经提到了一些需要关注的关键点，这里我们将设计的预期目标的细节展开讨论。

- 系统由许多廉价的普通组件组成，组件失效是一种常态。系统必须持续监控自身的状态，它必须将组件失效作为一种常态，能够迅速地侦测、冗余并恢复失效的组件。
- 系统存储一定数量的大文件。我们预期会有几百万文件，文件的大小通常在100MB或者以上。数个GB大小的文件也是普遍存在，并且要能够被有效的管理。系统也必须支持小文件，但是不需要针对小文件做专门的优化。
- 系统的工作负载主要由两种读操作组成：大规模的流式读取和小规模的随机读取。大规模的流式读取通常一次读取数百KB的数据，更常见的是一次读取1MB甚至更多的数据。来自同一个客户机的连续操作通常是读取同一个文件中连续的一个区域。小规模的随机读取通常是在文件某个随机的位置读取几个KB数据。如果应用程序对性能非常关注，通常的做法是把小规模的随机读取操作合并并排序，之后按顺序批量读取，这样就避免了在文件中前后来回的移动读取位置。
- 系统的工作负载还包括许多大规模的、顺序的、数据追加方式的写操作。一般情况下，每次写入的数据的大小和大规模读类似。数据一旦被写入后，文件就很少会被修改了。系统支持小规模的随机位置写入操作，但是可能效率不彰。
- 系统必须高效的、行为定义明确的*（alex注：well-defined）*实现多客户端并行追加数据到同一个文件里的语意。我们的文件通常被用于”生产者-消费者“队列，或者其它多路文件合并操作。通常会有数百个生产者，每个生产者进程运行在一台机器上，同时对一个文件进行追加操作。使用最小的同步开销来实现的原子的多路追加数据操作是必不可少的。文件可以在稍后读取，或者是消费者在追加的操作的同时读取文件。
- 高性能的稳定网络带宽远比低延迟重要。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。

2.2 接口

GFS提供了一套类似传统文件系统的API接口函数，虽然并不是严格按照POSIX等标准API的形式实现的。文件以分层目录的形式组织，用路径名来标识。我们支持常用的操作，如创建新文件、删除文件、打开文件、关闭文件、读和写文件。

另外，GFS提供了快照和记录追加操作。快照以很低的成本创建一个文件或者目录树的拷贝。记录追加操作允许多个客户端同时对一个文件进行数据追加操作，同时保证每个客户端的追加操作都是原子性的。这对于实现多路结果合并，以及"生产者-消费者"队列非常有用，多个客户端可以在不需要额外的同步锁定的情况下，同时对一个文件追加数据。我们发现这些类型的文件对于构建大型分布应用是非常重要的。快照和记录追加操作将在3.4和3.3节分别讨论。

2.3 架构

一个GFS集群包含一个单独的Master节点*（alex注：这里的一个单独的Master节点的含义是GFS系统中只存在一个逻辑上的Master组件。后面我们还会提到Master节点复制，因此，为了理解方便，我们把Master节点视为一个逻辑上的概念，一个逻辑的Master节点包括两台物理主机，即两台Master服务器）*、多台Chunk服务器，并且同时被多个客户端访问，如图1所示。所有的这些机器通常都是普通的Linux机器，运行着用户级别(user-level)的服务进程。我们可以很容易的把Chunk服务器和客户端都放在同一台机器上，前提是机器资源允许，并且我们能够接受不可靠的应用程序代码带来的稳定性降低的风险。

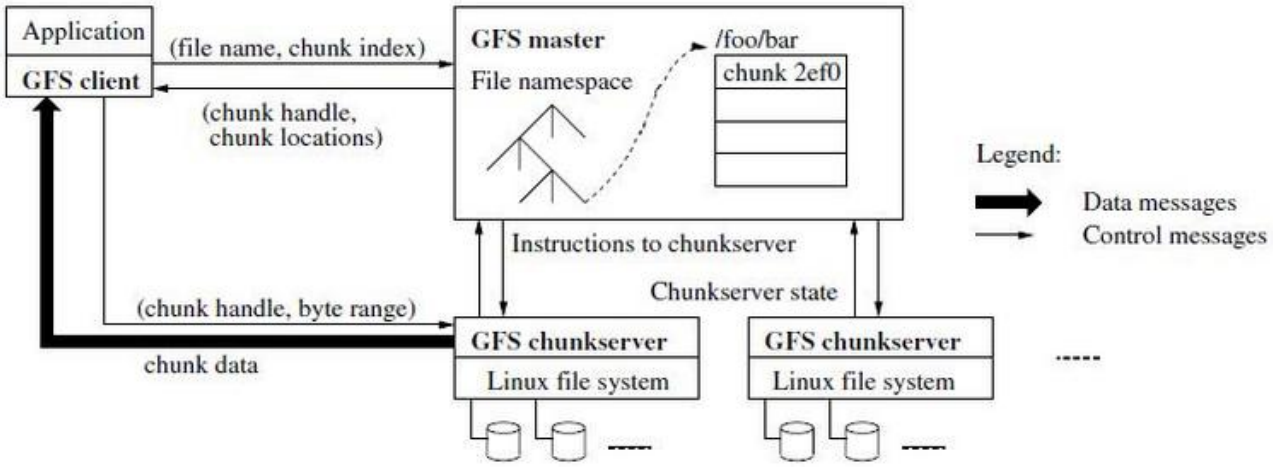


Figure 1: GFS Architecture

GFS存储的文件都被分割成固定大小的Chunk。在Chunk创建的时候，Master服务器会给每个Chunk分配一个不变的、全球唯一的64位的Chunk标识。Chunk服务器把Chunk以linux文件的形式保存在本地硬盘上，并且根据指定的Chunk标识和字节范围来读写块数据。出于可靠性的考虑，每个块都会复制到多个块服务器上。缺省情况下，我们使用3个存储复制节点，不过用户可以为不同的文件命名空间设定不同的复制级别。

Master节点管理所有的文件系统元数据。这些元数据包括名字空间、访问控制信息、文件和Chunk的映射信息、以及当前Chunk的位置信息。Master节点还管理着系统范围内的活动，比如，Chunk租用管理*(alex注：BDB也有关于lease的描述，不知道是否相同)*、孤儿Chunk*(alex注：orphaned chunks)*的回收、以及Chunk在Chunk服务器之间的迁移。Master节点使用心跳信息周期地和每个Chunk服务器通讯，发送指令到各个Chunk服务器并接收Chunk服务器的状态信息。

GFS客户端代码以库的形式被链接到客户程序里。客户端代码实现了GFS文件系统的API接口函数、应用程序与Master节点和Chunk服务器通讯、以及对数据进行读写操作。客户端和Master节点的通信只获取元数据，所有的数据操作都是由客户端直接和Chunk服务器进行交互的。我们不提供POSIX标准的API的功能，因此，GFS API调用不需要深入到Linux vnode级别。

无论是客户端还是Chunk服务器都不需要缓存文件数据。客户端缓存数据几乎没有什么用处，因为大部分程序要么以流的方式读取一个巨大文件，要么工作集太大根本无法被缓存。无需考虑缓存相关的问题也简

化了客户端和整个系统的设计和实现。（不过，客户端会缓存元数据。）Chunk服务器不需要缓存文件数据的原因是，Chunk以本地文件的方式保存，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。

2.4 单一Master节点

单一的Master节点的策略大大简化了我们的设计。单一的Master节点可以通过全局的信息精确定位Chunk的位置以及进行复制决策。另外，我们必须减少对Master节点的读写，避免Master节点成为系统的瓶颈。客户端并不通过Master节点读写文件数据。反之，客户端向Master节点询问它应该联系的Chunk服务器。客户端将这些元数据信息缓存一段时间，后续的操作将直接和Chunk服务器进行数据读写操作。

我们利用图1解释一下一次简单读取的流程。首先，客户端把文件名和程序指定的字节偏移，根据固定的Chunk大小，转换成文件的Chunk索引。然后，它把文件名和Chunk索引发送给Master节点。Master节点将相应的Chunk标识和副本的位置信息发还给客户端。客户端用文件名和Chunk索引作为key缓存这些信息。

之后客户端发送请求到其中的一个副本处，一般会选择最近的。请求信息包含了Chunk的标识和字节范围。在对这个Chunk的后续读取操作中，客户端不必再和Master节点通讯了，除非缓存的元数据信息过期或者文件被重新打开。实际上，客户端通常会在一次请求中查询多个Chunk信息，Master节点的回应也可能包含了紧跟着这些被请求的Chunk后面的Chunk的信息。在实际应用中，这些额外的信息在没有任何代价的情况下，避免了客户端和Master节点未来可能会发生的几次通讯。

2.5 Chunk尺寸

Chunk的大小是关键的设计参数之一。我们选择了64MB，这个尺寸远远大于一般文件系统的Block size。每个Chunk的副本都以普通Linux文件的形式保存在Chunk服务器上，只有在需要的时候才扩大。惰性空间分配策略避免了因内部碎片造成的空间浪费，内部碎片或许是对选择这么大的Chunk尺寸最具争议一点。

选择较大的Chunk尺寸有几个重要的优点。首先，它减少了客户端和Master节点通讯的需求，因为只需要一次和Mater节点的通信就可以获取Chunk的位置信息，之后就可以对同一个Chunk进行多次的读写操作。这种方式对降低我们的工作负载来说效果显著，因为我们的应用程序通常是连续读写大文件。即使是小规模的随机读取，采用较大的Chunk尺寸也带来明显的好处，客户端可以轻松的缓存一个数TB的工作数据集所有的Chunk位置信息。其次，采用较大的Chunk尺寸，客户端能够对一个块进行多次操作，这样就可以通过与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，在2.6.1节我们会讨论元数据全部放在内存中带来的额外的好处。

另一方面，即使配合惰性空间分配，采用较大的Chunk尺寸也有其缺陷。小文件包含较少的Chunk，甚至只有一个Chunk。当有许多的客户端对同一个小文件进行多次的访问时，存储这些Chunk的Chunk服务器就会变成热点。在实际应用中，由于我们的程序通常是连续的读取包含多个Chunk的大文件，热点还不是主要的问题。

然而，当我们第一次把GFS用于批处理队列系统的时候，热点的问题还是产生了：一个可执行文件在GFS上保存为single-chunk文件，之后这个可执行文件在数百台机器上同时启动。存放这个可执行文件的几个Chunk服务器被数百个客户端的并发请求访问导致系统局部过载。我们通过使用更大的复制参数来保存可执行文件，以及错开批处理队列系统程序的启动时间的方法解决了这个问题。一个可能的长效解决方案是，在这种的情况下，允许客户端从其它客户端读取数据。

2.6 元数据

Master服务器*（alex注：注意逻辑的Master节点和物理的Master服务器的区别。后续我们谈的是每个Master服务器的行为，如存储、内存等等，因此我们将全部使用物理名称）*存储3种主要类型的元数据，包括：文件和Chunk的命名空间、文件和Chunk的对应关系、每个Chunk副本的存放地点。所有的元数

据都保存在Master服务器的内存中。前两种类型的元数据（命名空间、文件和Chunk的对应关系）同时也会以记录变更日志的方式记录在操作系统的系统日志文件中，日志文件存储在本地磁盘上，同时日志会被复制到其它的远程Master服务器上。采用保存变更日志的方式，我们能够简单可靠的更新Master服务器的状态，并且不用担心Master服务器崩溃导致数据不一致的风险。Master服务器不会持久保存Chunk位置信息。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。

2.6.1 内存中的数据结构

因为元数据保存在内存中，所以Master服务器的操作速度非常快。并且，Master服务器可以在后台简单而高效的周期性扫描自己保存的全部状态信息。这种周期性的状态扫描也用于实现Chunk垃圾收集、在Chunk服务器失效的时重新复制数据、通过Chunk的迁移实现跨Chunk服务器的负载均衡以及磁盘使用状况统计等功能。4.3和4.4章节将深入讨论这些行为。

将元数据全部保存在内存中的方法有潜在问题：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。但是在实际应用中，这并不是一个严重的问题。Master服务器只需要不到64个字节的元数据就能够管理一个64MB的Chunk。由于大多数文件都包含多个Chunk，因此绝大多数Chunk都是满的，除了文件的最后一个Chunk是部分填充的。同样的，每个文件的在命名空间中的数据大小通常在64字节以下，因为保存的文件名是用前缀压缩算法压缩过的。

即便是需要支持更大的文件系统，为Master服务器增加额外内存的费用是很少的，而通过增加有限的费用，我们就能够把元数据全部保存在内存里，增强了系统的简洁性、可靠性、高性能和灵活性。

2.6.2 Chunk位置信息

Master服务器并不保存持久化保存哪个Chunk服务器存有指定Chunk的副本的信息。Master服务器只是在启动的时候轮询Chunk服务器以获取这些信息。Master服务器能够保证它持有的信息始终是最新的，因为它控制了所有的Chunk位置的分配，而且通过周期性的心跳信息监控Chunk服务器的状态。

最初设计时，我们试图把Chunk的位置信息持久的保存在Master服务器上，但是后来我们发现在启动的时候轮询Chunk服务器，之后定期轮询更新的方式更简单。这种设计简化了在有Chunk服务器加入集群、离开集群、更名、失效、以及重启的时候，Master服务器和Chunk服务器数据同步的问题。在一个拥有数百台服务器的集群中，这类事件会频繁的发生。

可以从另外一个角度去理解这个设计决策：只有Chunk服务器才能最终确定一个Chunk是否在它的硬盘上。我们从没有考虑过在Master服务器上维护一个这些信息的全局视图，因为Chunk服务器的错误可能会导致Chunk自动消失(比如，硬盘损坏了或者无法访问了)，亦或者操作人员可能会重命名一个Chunk服务器。

2.6.3 操作日志

操作日志包含了关键的元数据变更历史记录。这对GFS非常重要。这不仅仅是因为操作日志是元数据唯一的持久化存储记录，它也作为判断同步操作顺序的逻辑时间基线*（alex注：也就是通过逻辑日志的序号作为操作发生的逻辑时间，类似于事务系统中的LSN）*。文件和Chunk，连同它们的版本(参考4.5节)，都由它们创建的逻辑时间唯一的、永久的标识。

操作日志非常重要，我们必须确保日志文件的完整，确保只有在元数据的变化被持久化后，日志才对客户端是可见的。否则，即使Chunk本身没有出现任何问题，我们仍有可能丢失整个文件系统，或者丢失客户端最近的操作。所以，我们会把日志复制到多台远程机器，并且只有把相应的日志记录写入到本地以及远程机器的硬盘后，才会响应客户端的操作请求。Master服务器会收集多个日志记录后批量处理，以减少写入磁盘和复制对系统整体性能的影响。

Master服务器在灾难恢复时，通过重演操作日志把文件系统恢复到最近的状态。为了缩短Master启动的时间，我们必须使日志足够小*（alex注：即重演系统操作的日志量尽量的少）*。Master服务器在日志增长到一定量时对系统状态做一次Checkpoint*(alex注：Checkpoint是一种行为，一种对数据库状态作一次快照的行为)*，将所有的状态数据写入一个Checkpoint文件*（alex注：并删除之前的日志文件）*。在灾难

恢复的时候，Master服务器就通过从磁盘上读取这个Checkpoint文件，以及重演Checkpoint之后的有限个日志文件就能够恢复系统。Checkpoint文件以压缩B-树形势的数据结构存储，可以直接映射到内存，在用于命名空间查询时无需额外的解析。这大大提高了恢复速度，增强了可用性。

由于创建一个Checkpoint文件需要一定的时间，所以Master服务器的内部状态被组织为一种格式，这种格式要确保在Checkpoint过程中不会阻塞正在进行的修改操作。Master服务器使用独立的线程切换到新的日志文件和创建新的Checkpoint文件。新的Checkpoint文件包括切换前所有的修改。对于一个包含数百万个文件的集群，创建一个Checkpoint文件需要1分钟左右的时间。创建完成后，Checkpoint文件会被写入在本地和远程的硬盘里。

Master服务器恢复只需要最新的Checkpoint文件和后续的日志文件。旧的Checkpoint文件和日志文件可以被删除，但是为了应对灾难性的故障*（alex注：catastrophes，数据备份相关文档中经常会遇到这个词，表示一种超出预期范围的灾难性事件），*我们通常会多保存一些历史文件。Checkpoint失败不会对正确性产生任何影响，因为恢复功能的代码可以检测并跳过没有完成的Checkpoint文件。

2.7 一致性模型

GFS支持一个宽松的一致性模型，这个模型能够很好的支撑我们的高度分布的应用，同时还保持了相对简单且容易实现的优点。本节我们讨论GFS的一致性的保障机制，以及对应用程序的意义。我们也着重描述了GFS如何管理这些一致性保障机制，但是实现的细节将在本论文的其它部分讨论。

2.7.1 GFS一致性保障机制

文件命名空间的修改（例如，文件创建）是原子性的。它们仅由Master节点的控制：命名空间锁提供了原子性和正确性（4.1章）的保障；Master节点的操作日志定义了这些操作在全局的顺序（2.6.3章）。

	写	记录追加
串行成功	已定义	已定义
并行成功	一致但是未定义	部分不一致
失败	不一致	

表1 操作后的文件状态

数据修改后文件region*（alex注：region这个词用中文非常难以表达，我认为应该是修改操作所涉及的文件中的某个范围）*的状态取决于操作的类型、成功与否、以及是否同步修改。表1总结了各种操作的结果。如果所有客户端，无论从哪个副本读取，读到的数据都一样，那么我们认为文件region是“一致的”；如果对文件的数据修改之后，region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region是“已定义的”。当一个数据修改操作成功执行，并且没有受到同时执行的其它写入操作的干扰，那么影响的region就是已定义的（隐含了一致性）：所有的客户端都可以看到写入的内容。并行修改操作成功完成之后，region处于一致的、未定义的状态：所有的客户端看到同样的数据，但是无法读到任何一次写入操作写入的数据。通常情况下，文件region内包含了来自多个修改操作的、混杂的数据片段。失败的修改操作导致一个region处于不一致状态（同时也是未定义的）：不同的客户在不同的时间会看到不同的数据。后面我们将描述应用如何区分已定义和未定义的region。应用程序没有必要再去细分未定义region的不同类型。

数据修改操作分为写入或者记录追加两种。写入操作把数据写在应用程序指定的文件偏移位置上。即使有多个修改操作并行执行时，记录追加操作至少可以把数据原子性的追加到文件中一次，但是偏移位置是由GFS选择的（3.3章）*（alex注：这句话有点费解，其含义是所有的追加写入都会成功，但是有可能被执行了多次，而且每次追加的文件偏移量由GFS自己计算）*。（相比而言，通常说的追加操作写的偏移位置是文件的尾部。）GFS返回给客户端一个偏移量，表示了包含了写入记录的、已定义的region的起点。另外，GFS可能会在文件中间插入填充数据或者重复记录。这些数据占据的文件region被认定是不一致的，这些数据通常比用户数据小的多。

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region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：（a） 对Chunk的所有副本的修改操作顺序一致（3.1章），（b）使用Chunk的版本号来检测副本是否因为它所在的Chunk服务器宕机（4.5章）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。它们会被垃圾收集系统尽快回收。

由于Chunk位置信息会被客户端缓存，所以在信息刷新前，客户端有可能从一个失效的副本读取了数据。在缓存的超时时间和文件下一次被打开的时间之间存在一个时间窗，文件再次被打开后会清除缓存中与该文件有关的所有Chunk位置信息。而且，由于我们的文件大多数都是只进行追加操作的，所以，一个失效的副本通常返回一个提前结束的Chunk而不是过期的数据。当一个Reader *（alex注：本文中将用到两个专有名词，Reader和Writer，分别表示执行GFS读取和写入操作的程序）*重新尝试并联络Master服务器时，它就会立刻得到最新的Chunk位置信息。

即使在修改操作成功执行很长时间之后，组件的失效也可能损坏或者删除数据。GFS通过Master服务器和所有Chunk服务器的定期“握手”来找到失效的Chunk服务器，并且使用Checksum来校验数据是否损坏（5.2章）。一旦发现问题，数据要尽快利用有效的副本进行恢复（4.3章）。只有当一个Chunk的所有副本在GFS检测到错误并采取应对措施之前全部丢失，这个Chunk才会不可逆转的丢失。在一般情况下GFS的反应时间 *（alex注：指Master节点检测到错误并采取应对措施）*是几分钟。即使在这种情况下，Chunk也只是不可用了，而不是损坏了：应用程序会收到明确的错误信息而不是损坏的数据。

2.7.2 程序的实现

使用GFS的应用程序可以利用一些简单技术实现这个宽松的一致性模型，这些技术也用来实现一些其它的目标功能，包括：尽量采用追加写入而不是覆盖，Checkpoint，自验证的写入操作，自标识的记录。

在实际应用中，我们所有的应用程序对文件的写入操作都是尽量采用数据追加方式，而不是覆盖方式。一种典型的应用，应用程序从头到尾写入数据，生成了一个文件。写入所有数据之后，应用程序自动将文件改名为一个永久保存的文件名，或者周期性的作Checkpoint，记录成功写入了多少数据。Checkpoint文件可以包含程序级别的校验和。Readers仅校验并处理上个Checkpoint之后产生的文件region，这些文件region的状态一定是已定义的。这个方法满足了我们一致性和并发处理的要求。追加写入比随机位置写入更加有效率，对应用程序的失败处理更具有弹性。Checkpoint可以让Writer以渐进的方式重新开始，并且可以防止Reader处理已经被成功写入，但是从应用程序的角度来看还并未完成的数据。

我们再来分析另一种典型的应用。许多应用程序并行的追加数据到同一个文件，比如进行结果的合并或者是一个生产者-消费者队列。记录追加方式的“至少一次追加”的特性保证了Writer的输出。Readers使用下面的方法来处理偶然性的填充数据和重复内容。Writers在每条写入的记录中都包含了额外的信息，例如Checksum，用来验证它的有效性。Reader可以利用Checksum识别和抛弃额外的填充数据和记录片段。如果应用不能容忍偶尔的重复内容(比如，如果这些重复数据触发了非幂等操作)，可以用记录的唯一标识符来过滤它们，这些唯一标识符通常用于命名程序中处理的实体对象，例如web文档。这些记录I/O功能 *（alex注：These functionalities for record I/O）*（除了剔除重复数据）都包含在我们的程序共享的库中，并且适用于Google内部的其它的文件接口实现。所以，相同序列的记录，加上一些偶尔出现的重复数据，都被分发到Reader了。

3. 系统交互

我们在设计这个系统时，一个重要的原则是最小化所有操作和Master节点的交互。带着这样的设计理念，我们现在描述一下客户机、Master服务器和Chunk服务器如何进行交互，以实现数据修改操作、原子的记录追加操作以及快照功能。

3.1 租约（**lease**）和变更顺序

（alex注：lease是数据库中的一个术语）

变更是一个会改变Chunk内容或者元数据的操作，比如写入操作或者记录追加操作。变更操作会在Chunk的所有副本上执行。我们使用租约（lease）机制来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，我们把这个副本叫做主Chunk。主Chunk对Chunk的所有更改操作进行序列化。所有的副本都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节点选择的租约的顺序决定，然后由租约中主Chunk分配的序列号决定。

设计租约机制的目的是为了最小化Master节点的管理负担。租约的初始超时设置为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（例如，Master节点想取消在一个已经被改名的文件上的修改操作）。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。

在图2中，我们依据步骤编号，展现写入操作的控制流程。

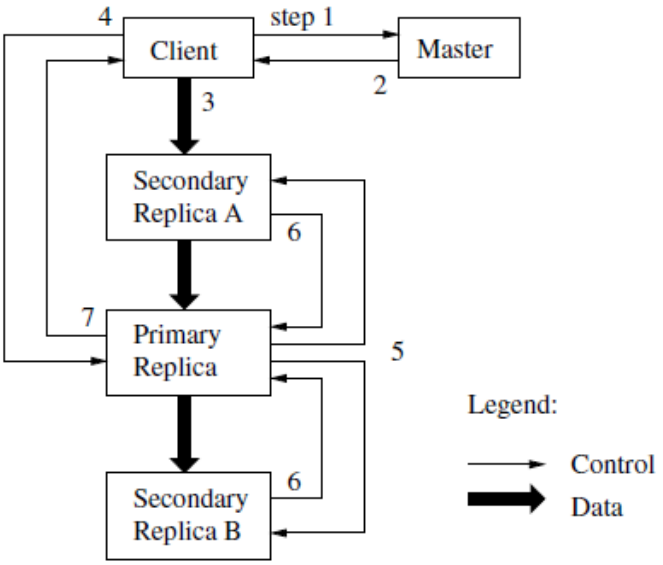


Figure 2: Write Control and Data Flow

1. 客户机向Master节点询问哪一个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（这个步骤在图上没有显示）。
2. Master节点将主Chunk的标识符以及其它副本（又称为secondary副本、二级副本）的位置返回给客户机。客户机缓存这些数据以便后续的操作。只有在主Chunk不可用，或者主Chunk回复信息表明它已不再持有租约的时候，客户机才需要重新跟Master节点联系。
3. 客户机把数据推送到所有的副本上。客户机可以以任意的顺序推送数据。Chunk服务器接收到数据并保存在它的内部LRU缓存中，一直到数据被使用或者过期交换出去。由于数据流的网络传输负载非常高，通过分离数据流和控制流，我们可以基于网络拓扑情况对数据流进行规划，提高系统性能，而不用去理会哪个Chunk服务器保存了主Chunk。3.2章节会进一步讨论这点。
4. 当所有的副本都确认接收到了数据，客户机发送写请求到主Chunk服务器。这个请求标识了早前推送到所有副本的数据。主Chunk为接收到的所有操作分配连续的序列号，这些操作可能来自不同的客户机，序列号保证了操作顺序执行。它以序列号的顺序把操作应用到它自己的本地状态中*（alex注：也就是在本地执行这些操作，这句话按字面翻译有点费解，也许应该翻译为“它顺序执行这些操作，并更新自己的状态”）*。
5. 主Chunk把写请求传递到所有的二级副本。每个二级副本依照主Chunk分配的序列号以相同的顺序

执行这些操作。
6. 所有的二级副本回复主Chunk，它们已经完成了操作。
7. 主Chunk服务器*（alex注：即主Chunk所在的Chunk服务器）*回复客户机。任何副本产生的任何错误都会返回给客户机。在出现错误的情况下，写入操作可能在主Chunk和一些二级副本执行成功。（如果操作在主Chunk上失败了，操作就不会被分配序列号，也不会被传递。）客户端的请求被确认为失败，被修改的region处于不一致的状态。我们的客户机代码通过重复执行失败的操作来处理这样的错误。在从头开始重复执行之前，客户机会先从步骤（3）到步骤（7）做几次尝试。

如果应用程序一次写入的数据量很大，或者数据跨越了多个Chunk，GFS客户机代码会把它们分成多个写操作。这些操作都遵循前面描述的控制流程，但是可能会被其它客户机上同时进行的操作打断或者覆盖。因此，共享的文件region的尾部可能包含来自不同客户机的数据片段，尽管如此，由于这些分解后的写入操作在所有的副本上都以相同的顺序执行完成，Chunk的所有副本都是一致的。这使文件region处于2.7节描述的一致的、但是未定义的状态。

3.2 数据流

为了提高网络效率，我们采取了把数据流和控制流分开的措施。在控制流从客户机到主Chunk、然后再到所有二级副本的同时，数据以管道的方式，顺序的沿着一个精心选择的Chunk服务器链推送。我们的目标是充分利用每台机器的带宽，避免网络瓶颈和高延时的连接，最小化推送所有数据的延时。

为了充分利用每台机器的带宽，数据沿着一个Chunk服务器链顺序的推送，而不是以其它拓扑形式分散推送（例如，树型拓扑结构）。线性推送模式下，每台机器所有的出口带宽都用于以最快的速度传输数据，而不是在多个接受者之间分配带宽。

为了尽可能的避免出现网络瓶颈和高延迟的链接（eg，inter-switch最有可能出现类似问题），每台机器都尽量的在网络拓扑中选择一台还没有接收到数据的、离自己最近的机器作为目标推送数据。假设客户机把数据从Chunk服务器S1推送到S4。它把数据推送到最近的Chunk服务器S1。S1把数据推送到S2，因为S2和S4中最接近的机器是S2。同样的，S2把数据传递给S3和S4之间更近的机器，依次类推推送下去。我们的网络拓扑非常简单，通过IP地址就可以计算出节点的“距离”。

最后，我们利用基于TCP连接的、管道式数据推送方式来最小化延迟。Chunk服务器接收到数据后，马上开始向前推送。管道方式的数据推送对我们帮助很大，因为我们采用全双工的交换网络。接收到数据后立刻向前推送不会降低接收的速度。在没有网络拥塞的情况下，传送B字节的数据到R个副本的理想时间是 B/T+RL ，T是网络的吞吐量，L是在两台机器数据传输的延迟。通常情况下，我们的网络连接速度是100Mbps（T），L将远小于1ms。因此，1MB的数据在理想情况下80ms左右就能分发出去。

3.3 原子的记录追加

GFS提供了一种原子的数据追加操作–记录追加。传统方式的写入操作，客户程序会指定数据写入的偏移量。对同一个region的并行写入操作不是串行的：region尾部可能会包含多个不同客户机写入的数据片段。使用记录追加，客户机只需要指定要写入的数据。GFS保证至少有一次原子的写入操作成功执行（即写入一个顺序的byte流），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机。这类似于在Unix操作系统编程环境中，对以O_APPEND模式打开的文件，多个并发写操作在没有竞态条件时的行为。

记录追加在我们的分布应用中非常频繁的使用，在这些分布式应用中，通常有很多的客户机并行地对同一个文件追加写入数据。如果我们采用传统方式的文件写入操作，客户机需要额外的复杂、昂贵的同步机制，例如使用一个分布式的锁管理器。在我们的工作中，这样的文件通常用于多个生产者/单一消费者的队列系统，或者是合并了来自多个客户机的数据的结果文件。

记录追加是一种修改操作，它也遵循3.1节描述的控制流程，除了在主Chunk有些额外的控制逻辑。客户机把数据推送给文件最后一个Chunk的所有副本，之后发送请求给主Chunk。主Chunk会检查这次记录

追加操作是否会使Chunk超过最大尺寸（64MB）。如果超过了最大尺寸，主Chunk首先将当前Chunk填充到最大尺寸，之后通知所有二级副本做同样的操作，然后回复客户机要求其对下一个Chunk重新进行记录追加操作。（记录追加的数据大小严格控制在Chunk最大尺寸的1/4，这样即使在最坏情况下，数据碎片的数量仍然在可控的范围。）通常情况下追加的记录不超过Chunk的最大尺寸，主Chunk把数据追加到自己的副本内，然后通知二级副本把数据写在跟主Chunk一样的位置上，最后回复客户机操作成功。

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。重新进行记录追加的结果是，同一个Chunk的不同副本可能包含不同的数据–重复包含一个记录全部或者部分的数据。GFS并不保证Chunk的所有副本在字节级别是完全一致的。它只保证数据作为一个整体原子的被至少写入一次。这个特性可以通过简单观察推导出来：如果操作成功执行，数据一定已经写入到Chunk的所有副本的相同偏移位置上。这之后，所有的副本至少都到了记录尾部的长度，任何后续的记录都会追加到更大的偏移地址，或者是不同的Chunk上，即使其它的Chunk副本被Master节点选为了主Chunk。就我们的一致性保障模型而言，记录追加操作成功写入数据的region是已定义的（因此也是一致的），反之则是不一致的（因此也就是未定义的）。正如我们在2.7.2节讨论的，我们的程序可以处理不一致的区域。

3.4 快照

(alex注：这一节非常难以理解，总的来说依次讲述了什么是快照、快照使用的COW技术、快照如何不干扰当前操作)
快照操作几乎可以瞬间完成对一个文件或者目录树（“源”）做一个拷贝，并且几乎不会对正在进行的其它操作造成任何干扰。我们的用户可以使用快照迅速的创建一个巨大的数据集的分支拷贝（而且经常是递归的拷贝拷贝），或者是在做实验性的数据操作之前，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。

就像AFS *（alex注：AFS，即Andrew File System，一种分布式文件系统），*我们用标准的copy-on-write技术实现快照。当Master节点收到一个快照请求，它首先取消作快照的文件的所有Chunk的租约。这个措施保证了后续对这些Chunk的写操作都必须与Master交互交互以找到租约持有者。这就给Master节点一个率先创建Chunk的新拷贝的机会。

租约取消或者过期之后，Master节点把这个操作以日志的方式记录到硬盘上。然后，Master节点通过复制源文件或者目录的元数据的方式，把这条日志记录的变化反映到保存在内存的状态中。新创建的快照文件和源文件指向完全相同的Chunk地址。

在快照操作之后，当客户机第一次想写入数据到Chunk C，它首先会发送一个请求到Master节点查询当前的租约持有者。Master节点注意到Chunke C的引用计数超过了1*(alex注：不太明白为什么会大于1.难道是Snapshot没有释放引用计数？)*。Master节点不会马上回复客户机的请求，而是选择一个新的Chunk句柄C`。之后，Master节点要求每个拥有Chunk C当前副本的Chunk服务器创建一个叫做C`的新Chunk。通过在源Chunk所在Chunk服务器上创建新的Chunk，我们确保数据在本地而不是通过网络复制（我们的硬盘比我们的100Mb以太网大约快3倍）。从这点来讲，请求的处理方式和任何其它Chunk没什么不同：Master节点确保新Chunk C`的一个副本拥有租约，之后回复客户机，客户机得到回复后就可以正常的写这个Chunk，而不必理会它是从一个已存在的Chunk克隆出来的。

4. Master节点的操作

Master节点执行所有的名称空间操作。此外，它还管理着整个系统里所有Chunk的副本：它决定Chunk的存储位置，创建新Chunk和它的副本，协调各种各样的系统活动以保证Chunk被完全复制，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。本节我们讨论上述的主题。

4.1 名称空间管理和锁

Master节点的很多操作会花费很长的时间：比如，快照操作必须取消Chunk服务器上快照所涉及的所有的Chunk的租约。我们不希望在这些操作的运行时，延缓了其它的Master节点的操作。因此，我们允许多个操作同时进行，使用名称空间的region上的锁来保证执行的正确顺序。

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。GFS也不支

持文件或者目录的链接（即Unix术语中的硬链接或者符号链接）。在逻辑上，GFS的名称空间就是一个全路径和元数据映射关系的查找表。利用前缀压缩，这个表可以高效的存储在内存中。在存储名称空间的树型结构上，每个节点（绝对路径的文件名或绝对路径的目录名）都有一个关联的读写锁。

每个Master节点的操作在开始之前都要获得一系列的锁。通常情况下，如果一个操作涉及/d1/d2/…/dn/leaf，那么操作首先要获得目录/d1，/d1/d2，…，/d1/d2/…/dn的读锁，以及/d1/d2/…/dn/leaf的读写锁。注意，根据操作的不同，leaf可以是一个文件，也可以是一个目录。

现在，我们演示一下在/home/user被快照到/save/user的时候，锁机制如何防止创建文件/home/user/foo。快照操作获取/home和/save的读取锁，以及/home/user和/save/user的写入锁。文件创建操作获得/home和/home/user的读取锁，以及/home/user/foo的写入锁。这两个操作要顺序执行，因为它们试图获取的/home/user的锁是相互冲突。文件创建操作不需要获取父目录的写入锁，因为这里没有”目录”，或者类似inode等用来禁止修改的数据结构。文件名的读取锁足以防止父目录被删除。

采用这种锁方案的优点是支持对同一目录的并行操作。比如，可以再同一个目录下同时创建多个文件：每一个操作都获取一个目录名的上的读取锁和文件名上的写入锁。目录名的读取锁足以的防止目录被删除、改名以及被快照。文件名的写入锁序列化文件创建操作，确保不会多次创建同名的文件。

因为名称空间可能有很多节点，读写锁采用惰性分配策略，在不再使用的时候立刻被删除。同样，锁的获取也要依据一个全局一致的顺序来避免死锁：首先按名称空间的层次排序，在同一个层次内按字典顺序排序。

4.2 副本的位置

GFS集群是高度分布的多层布局结构，而不是平面结构。典型的拓扑结构是有数百个Chunk服务器安装在许多机架上。Chunk服务器被来自同一或者不同机架上的数百个客户机轮流访问。不同机架上的两台机器间的通讯可能跨越一个或多个网络交换机。另外，机架的出入带宽可能比机架内所有机器加和在一起的带宽要小。多层分布架构对数据的灵活性、可靠性以及可用性方面提出特有的挑战。

Chunk副本位置选择的策略服务两大目标：最大化数据可靠性和可用性，最大化网络带宽利用率。为了实现这两个目的，仅仅是在多台机器上分别存储这些副本是不够的，这只能预防硬盘损坏或者机器失效带来的影响，以及最大化每台机器的网络带宽利用率。我们必须在多个机架间分布储存Chunk的副本。这保证Chunk的一些副本在整个机架被破坏或掉线（比如，共享资源，如电源或者网络交换机造成的问题）的情况下依然存在且保持可用状态。这还意味着在网络流量方面，尤其是针对Chunk的读操作，能够有效利用多个机架的整合带宽。另一方面，写操作必须和多个机架上的设备进行网络通信，但是这个代价是我们愿意付出的。

4.3 创建，重新复制，重新负载均衡

Chunk的副本有三个用途：Chunk创建，重新复制和重新负载均衡。

当Master节点创建一个Chunk时，它会选择在哪里放置初始的空的副本。Master节点会考虑几个因素。（1）我们希望在低于平均硬盘使用率的Chunk服务器上存储新的副本。这样的做法最终能够平衡Chunk服务器之间的硬盘使用率。（2）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。虽然创建操作本身是廉价的，但是创建操作也意味着随之会有大量的写入数据的操作，因为Chunk在Writer真正写入数据的时候才被创建，而在我们的”追加一次，读取多次”的工作模式下，Chunk一旦写入成功之后就会变为只读的了。（3）如上所述，我们希望把Chunk的副本分布在多个机架之间。

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。这可能是由几个原因引起的：一个Chunk服务器不可用了，Chunk服务器报告它所存储的一个副本损坏了，Chunk服务器的一个磁盘因为错误不可用了，或者Chunk副本的复制因数提高了。每个需要被重新复制的Chunk都会根据几个因素进行排序。一个因素是Chunk现有副本数量和复制因数相差多少。例如，丢失两个副本的Chunk比丢失一个副本的Chunk有更高的优先级。另外，我们优先重新复制活跃（live）文件的Chunk而不是最近刚被删除的文件的Chunk（查看4.4节）。最后，为了最小化失效的Chunk对正在运行的应用程

序的影响，我们提高会阻塞客户机程序处理流程的Chunk的优先级。

Master节点选择优先级最高的Chunk，然后命令某个Chunk服务器直接从可用的副本”克隆”一个副本出来。选择新副本的位置的策略和创建时类似：平衡硬盘使用率、限制同一台Chunk服务器上的正在进行的克隆操作的数量、在机架间分布副本。为了防止克隆产生的网络流量大大超过客户机的流量，Master节点对整个集群和每个Chunk服务器上的同时进行的克隆操作的数量都进行了限制。另外，Chunk服务器通过调节它对源Chunk服务器读请求的频率来限制它用于克隆操作的带宽。

最后，Master服务器周期性地对副本进行重新负载均衡：它检查当前的副本分布情况，然后移动副本以便更好的利用硬盘空间、更有效的进行负载均衡。而且在这个过程中，Master服务器逐渐的填满一个新的Chunk服务器，而不是在短时间内用新的Chunk填满它，以至于过载。新副本的存储位置选择策略和上面讨论的相同。另外，Master节点必须选择哪个副本要被移走。通常情况，Master节点移走那些剩余空间低于平均值的Chunk服务器上的副本，从而平衡系统整体的硬盘使用率。

4.4 垃圾回收

GFS在文件删除后不会立刻回收可用的物理空间。GFS空间回收采用惰性的策略，只在文件和Chunk级的常规垃圾收集时进行。我们发现这个方法使系统更简单、更可靠。

4.4.1 机制

当一个文件被应用程序删除时，Master节点象对待其它修改操作一样，立刻把删除操作以日志的方式记录下来。但是，Master节点并不马上回收资源，而是把文件名改为一个包含删除时间戳的、隐藏的名字。当Master节点对文件系统命名空间做常规扫描的时候，它会删除所有三天前的隐藏文件（这个时间间隔是可以设置的）。直到文件被真正删除，它们仍旧可以用新的特殊的名字读取，也可以通过把隐藏文件改名为正常显示的文件名的方式“反删除”。当隐藏文件被从名称空间中删除，Master服务器内存中保存的这个文件的相关元数据才会被删除。这也有效的切断了文件和它包含的所有Chunk的连接*（alex注：原文是This effectively severs its links to all its chunks）*。

在对Chunk名字空间做类似的常规扫描时，Master节点找到孤儿Chunk（不被任何文件包含的Chunk）并删除它们的元数据。Chunk服务器在和Master节点交互的心跳信息中，报告它拥有的Chunk子集的信息，Master节点回复Chunk服务器哪些Chunk在Master节点保存的元数据中已经不存在了。Chunk服务器可以任意删除这些Chunk的副本。

4.4.2 讨论

虽然分布式垃圾回收在编程语言领域是一个需要复杂的方案才能解决的难题，但是在GFS系统中是非常简单的。我们可以轻易的得到Chunk的所有引用：它们都只存储在Master服务器上的文件到块的映射表中。我们也可以很轻易的得到所有Chunk的副本：它们都以Linux文件的形式存储在Chunk服务器的指定目录下。所有Master节点不能识别的副本都是”垃圾”。

垃圾回收在空间回收方面相比直接删除有几个优势。首先，对于组件失效是常态的大规模分布式系统，垃圾回收方式简单可靠。Chunk可能在某些Chunk服务器创建成功，某些Chunk服务器上创建失败，失败的副本处于无法被Master节点识别的状态。副本删除消息可能丢失，Master节点必须重新发送失败的删除消息，包括自身的和Chunk服务器的*（alex注：自身的指删除metadata的消息）*。垃圾回收提供了一致的、可靠的清除无用副本的方法。第二，垃圾回收把存储空间的回收操作合并到Master节点规律性的后台活动中，比如，例行扫描和与Chunk服务器握手等。因此，操作被批量的执行，开销会被分散。另外，垃圾回收在Master节点相对空闲的时候完成。这样Master节点就可以给那些需要快速反应的客户机请求提供更快捷的响应。第三，延缓存储空间回收为意外的、不可逆转的删除操作提供了安全保障。

根据我们的使用经验，延迟回收空间的主要问题是，延迟回收会阻碍用户调优存储空间的使用，特别是当存储空间比较紧缺的时候。当应用程序重复创建和删除临时文件时，释放的存储空间不能马上重用。我们通过显式的再次删除一个已经被删除的文件的方式加速空间回收的速度。我们允许用户为命名空间的不同部分设定不同的复制和回收策略。例如，用户可以指定某些目录树下面的文件不做复制，删除的文件被即时的、不可恢复的从文件系统移除。

4.5 过期失效的副本检测

当Chunk服务器失效时，Chunk的副本有可能因错失了一些修改操作而过期失效。Master节点保存了每个Chunk的版本号，用来区分当前的副本和过期副本。

无论何时，只要Master节点和Chunk签订一个新的租约，它就增加Chunk的版本号，然后通知最新的副本。Master节点和这些副本都把新的版本号记录在它们持久化存储的状态信息中。这个动作发生在任何客户机得到通知以前，因此也是对这个Chunk开始写之前。如果某个副本所在的Chunk服务器正好处于失效状态，那么副本的版本号就不会被增加。Master节点在这个Chunk服务器重新启动，并且向Master节点报告它拥有的Chunk的集合以及相应的版本号的时候，就会检测出它包含过期的Chunk。如果Master节点看到一个比它记录的版本号更高的版本号，Master节点会认为它和Chunk服务器签订租约的操作失败了，因此会选择更高的版本号作为当前的版本号。

Master节点在例行的垃圾回收过程中移除所有的过期失效副本。在此之前，Master节点在回复客户机的Chunk信息请求的时候，简单的认为那些过期的块根本就不存在。另外一重保障措施是，Master节点在通知客户机哪个Chunk服务器持有租约、或者指示Chunk服务器从哪个Chunk服务器进行克隆时，消息中都附带了Chunk的版本号。客户机或者Chunk服务器在执行操作时都会验证版本号以确保总是访问当前版本的数据。

5. 容错和诊断

我们在设计GFS时遇到的最大挑战之一是如何处理频繁发生的组件失效。组件的数量和质量让这些问题出现的频率远远超过一般系统意外发生的频率：我们不能完全依赖机器的稳定性，也不能完全相信硬盘的可靠性。组件的失效可能造成系统不可用，更糟糕的是，还可能产生不完整的数据。我们讨论我们如何面对这些挑战，以及当组件失效不可避免的发生时，用GFS自带工具诊断系统故障。

5.1 高可用性

在GFS集群的数百个服务器之中，在任何给定的时间必定会有些服务器是不可用的。我们使用两条简单但是有效的策略保证整个系统的高可用性：快速恢复和复制。

5.1.1 快速恢复

不管Master服务器和Chunk服务器是如何关闭的，它们都被设计为可以在数秒钟内恢复它们的状态并重新启动。事实上，我们并不区分正常关闭和异常关闭；通常，我们通过直接kill掉进程来关闭服务器。客户机和其它的服务器会感觉到系统有点颠簸*(alex注：a minor hiccup)*，正在发出的请求会超时，需要重新连接到重启后的服务器，然后重试这个请求。6.6.2章节记录了实测的启动时间。

5.1.2 Chunk复制

正如之前讨论的，每个Chunk都被复制到不同机架上的不同的Chunk服务器上。用户可以为文件命名空间的不同部分设定不同的复制级别。缺省是3。当有Chunk服务器离线了，或者通过Chksum校验（参考5.2节）发现了已经损坏的数据，Master节点通过克隆已有的副本保证每个Chunk都被完整复制*（alex注：即每个Chunk都有复制因子制定的个数个副本，缺省是3）*。虽然Chunk复制策略对我们非常有效，但是我们也在寻找其它形式的跨服务器的冗余解决方案，比如使用奇偶校验、或者Erasure codes*（alex注：Erasure codes用来解决链接层中不相关的错误，以及网络拥塞和buffer限制造成的丢包错误）*来解决我们日益增长的只读存储需求。我们的系统主要的工作负载是追加方式的写入和读取操作，很少有随机的写入操作，因此，我们认为在我们这个高度解耦合的系统架构下实现这些复杂的冗余方案很有挑战性，但并非不可实现。

5.1.3 Master服务器的复制

为了保证Master服务器的可靠性，Master服务器的状态也要复制。Master服务器所有的操作日志和checkpoint文件都被复制到多台机器上。对Master服务器状态的修改操作能够提交成功的前提是，操作

日志写入到Master服务器的备节点和本机的磁盘。简单说来，一个Master服务进程负责所有的修改操作，包括后台的服务，比如垃圾回收等改变系统内部状态活动。当它失效的时，几乎可以立刻重新启动。如果Master进程所在的机器或者磁盘失效了，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。客户端使用规范的名字访问Master（比如gfs-test）节点，这个名字类似DNS别名，因此也就可以在Master进程转到别的机器上执行时，通过更改别名的实际指向访问新的Master节点。

此外，GFS中还有些“影子”Master服务器，这些“影子”服务器在“主”Master服务器宕机的时候提供文件系统的只读访问。它们是影子，而不是镜像，所以它们的数据可能比“主”Master服务器更新要慢，通常是不到1秒。对于那些不经常改变的文件、或者那些允许获取的数据有少量过期的应用程序，“影子”Master服务器能够提高读取的效率。事实上，因为文件内容是从Chunk服务器上读取的，因此，应用程序不会发现过期的文件内容。在这个短暂的时间窗内，过期的可能是文件的元数据，比如目录的内容或者访问控制信息。

“影子”Master服务器为了保持自身状态是最新的，它会读取一份当前正在进行的操作的日志副本，并且依照和主Master服务器完全相同的顺序来更改内部的数据结构。和主Master服务器一样，“影子”Master服务器在启动的时候也会从Chunk服务器轮询数据（之后定期拉数据），数据中包括了Chunk副本的位置信息；“影子”Master服务器也会定期和Chunk服务器“握手”来确定它们的状态。在主Master服务器因创建和删除副本导致副本位置信息更新时，“影子”Master服务器才和主Master服务器通信来更新自身状态。

5.2 数据完整性

每个Chunk服务器都使用Checksum来检查保存的数据是否损坏。考虑到一个GFS集群通常都有好几百台机器、几千块硬盘，磁盘损坏导致数据在读写过程中损坏或者丢失是非常常见的（第7节讲了一个原因）。我们可以通过别的Chunk副本来解决数据损坏问题，但是跨越Chunk服务器比较副本来检查数据是否损坏很不实际。另外，GFS允许有歧义的副本存在：GFS修改操作的语义，特别是早先讨论过的原子纪录追加的操作，并不保证副本完全相同*(alex注：副本不是byte-wise完全一致的)*。因此，每个Chunk服务器必须独立维护Checksum来校验自己的副本的完整性。

我们把每个Chunk都分成64KB大小的块。每个块都对应一个32位的Checksum。和其它元数据一样，Checksum与其它的用户数据是分开的，并且保存在内存和硬盘上，同时也记录操作日志。

对于读操作来说，在把数据返回给客户端或者其它的Chunk服务器之前，Chunk服务器会校验读取操作涉及的范围内的块的Checksum。因此Chunk服务器不会把错误数据传递到其它的机器上。如果发生某个块的Checksum不正确，Chunk服务器返回给请求者一个错误信息，并且通知Master服务器这个错误。作为回应，请求者应当从其它副本读取数据，Master服务器也会从其它副本克隆数据进行恢复。当一个新的副本就绪后，Master服务器通知副本错误的Chunk服务器删掉错误的副本。

Checksum对读操作的性能影响很小，可以基于几个原因来分析一下。因为大部分的读操作都至少要读取几个块，而我们只需要读取一小部分额外的相关数据进行校验。GFS客户端代码通过每次把读取操作都对齐在Checksum block的边界上，进一步减少了这些额外的读取操作的负面影响。另外，在Chunk服务器上，Chunksum的查找和比较不需要I/O操作，Checksum的计算可以和I/O操作同时进行。

Checksum的计算针对在Chunk尾部的追加写入操作作了高度优化（与之对应的是覆盖现有数据的写入操作），因为这类操作在我们的工作中占了很大比例。我们只增量更新最后一个不完整的块的Checksum，并且用所有的追加来的新Checksum块来计算新的Checksum。即使是最后一个不完整的Checksum块已经损坏了，而且我们不能够马上检查出来，由于新的Checksum和已有数据不吻合，在下次对这个块进行读取操作的时候，会检查出数据已经损坏了。

相比之下，如果写操作覆盖已经存在的一个范围内的Chunk，我们必须读取和校验被覆盖的第一个和最后一个块，然后再执行写操作；操作完成之后再重新计算和写入新的Checksum。如果我们不校验第一个和最后一个被写的块，那么新的Checksum可能会隐藏没有被覆盖区域内的数据错误。

在Chunk服务器空闲的时候，它会扫描和校验每个不活动的Chunk的内容。这使得我们能够发现很少被读取的Chunk是否完整。一旦发现有Chunk的数据损坏，Master可以创建一个新的、正确的副本，然后把

损坏的副本删除掉。这个机制也避免了非活动的、已损坏的Chunk欺骗Master节点，使Master节点认为它们已经有了足够多的副本了。

5.3 诊断工具

详尽的、深入细节的诊断日志，在问题隔离、调试、以及性能分析等方面给我们带来无法估量的帮助，同时也只需要很小的开销。没有日志的帮助，我们很难理解短暂的、不重复的机器之间的消息交互。GFS的服务器会产生大量的日志，记录了大量关键的事件（比如，Chunk服务器启动和关闭）以及所有的RPC的请求和回复。这些诊断日志可以随意删除，对系统的正确运行不造成任何影响。然而，我们在存储空间允许的情况下会尽量的保存这些日志。

RPC日志包含了网络上发生的所有请求和响应的详细记录，但是不包括读写的文件数据。通过匹配请求与回应，以及收集不同机器上的RPC日志记录，我们可以重演所有的消息交互来诊断问题。日志还用来跟踪负载测试和性能分析。

日志对性能的影响很小（远小于它带来的好处），因为这些日志的写入方式是顺序的、异步的。最近发生的事件日志保存在内存中，可用于持续不断的在线监控。

6. 度量

本节中，我们将使用一些小规模基准测试来展现GFS系统架构和实现上的一些固有瓶颈，还有些来自Google内部使用的真实的GFS集群的基准数据。

6.1 小规模基准测试

我们在一个包含1台Master服务器，2台Master服务器复制节点，16台Chunk服务器和16个客户机组成的GFS集群上测量性能。注意，采用这样的集群配置方案只是为了易于测试。典型的GFS集群有数百个Chunk服务器和数百个客户机。

所有机器的配置都一样：两个PIII 1.4GHz处理器，2GB内存，两个80G/5400rpm的硬盘，以及100Mbps全双工以太网连接到一个HP2524交换机。GFS集群中所有的19台服务器都连接在一个交换机，所有16台客户机连接到另一个交换机上。两个交换机之间使用1Gbps的线路连接。

6.1.1 读取

N个客户机从GFS文件系统同步读取数据。每个客户机从320GB的文件集合中随机读取4MB region的内容。读取操作重复执行256次，因此，每个客户机最终都读取1GB的数据。所有的Chunk服务器加起来总共只有32GB的内存，因此，我们预期只有最多10%的读取请求命中Linux的文件系统缓冲。我们的测试结果应该和一个在没有文件系统缓存的情况下读取测试的结果接近。

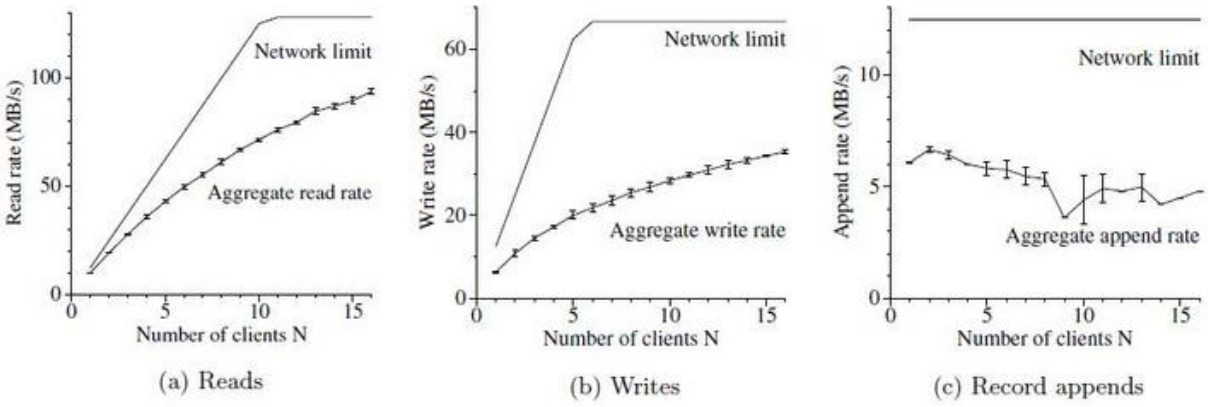


Figure 3: Aggregate Throughputs.

图三：合计吞吐量：上边的曲线显示了我们网络拓扑下的合计理论吞吐量上限。下边的曲线显示了观测到的吞吐量。这个曲线有着95%的可靠性，因为有时候测量会不够精确。

图3（a）显示了N个客户机整体的读取速度以及这个速度的理论极限。当连接两个交换机的1Gbps的链路饱和时，整体读取速度达到理论的极限值是125MB/S，或者说每个客户机配置的100Mbps网卡达到饱和时，每个客户机读取速度的理论极限值是12.5MB/s。实测结果是，当一个客户机读取的时候，读取的速度是10MB/s，也就是说达到客户机理论读取速度极限值的80%。对于16个客户机，整体的读取速度达到了94MB/s，大约是理论整体读取速度极限值的75%，也就是说每个客户机的读取速度是6MB/s。读取效率从80%降低到了75%，主要的原因是当读取的客户机增加时，多个客户机同时读取一个Chunk服务器的几率也增加了，导致整体的读取效率下降。

6.1.2 写入

N个客户机同时向N个不同的文件中写入数据。每个客户机以每次1MB的速度连续写入1GB的数据。图3（b）显示了整体的写入速度和它们理论上的极限值。理论上的极限值是67MB/s，因为我们需要把每一byte写入到16个Chunk服务器中的3个上，而每个Chunk服务器的输入连接速度是12.5MB/s。

一个客户机的写入速度是6.3MB，大概是理论极限值的一半。导致这个结果的主要原因是我们的网络协议栈。它与我们推送数据到Chunk服务器时采用的管道模式不相适应。从一个副本到另一个副本的数据传输延迟降低了整个的写入速度。

16个客户机整体的写入速度达到了35MB/s（即每个客户机2.2MB/s），大约只是理论极限值的一半。和多个客户机读取的情形很类型，随着客户机数量的增加，多个客户机同时写入同一个Chunk服务器的几率也增加了。而且，16个客户机并行写入可能引起的冲突比16个客户机并行读取要大得多，因为每个写入都会涉及三个不同的副本。

写入的速度比我们想象的要慢。在实际应用中，这没有成为我们的主要问题，因为即使在单个客户机上能够感受到延时，它也不会在有大量客户机的时候对整体的写入带宽造成显著的影响。

6.1.3 记录追加

图3（c）显示了记录追加操作的性能。N个客户机同时追加数据到一个文件。记录追加操作的性能受限于保存文件最后一个Chunk的Chunk服务器的带宽，而与客户机的数量无关。记录追加的速度由一个客户机的6.0MB/s开始，下降到16个客户机的4.8MB/s为止，速度的下降主要是由于不同客户端的网络拥塞以及网络传输速度的不同而导致的。

我们的程序倾向于同时处理多个这样的文件。换句话说，即N个客户机同时追加数据到M个共享文件中，这里N和M都是数十或者数百以上。所以，在我们的实际应用中，Chunk服务器的网络拥塞并没有成为一个严重问题，如果Chunk服务器的某个文件正在写入，客户机会去写另外一个文件。

6.2 实际应用中的集群

我们现在来仔细评估一下Google内部正在使用的两个集群，它们具有一定的代表性。集群A通常被上百个

工程师用于研究和开发。典型的任务是被人工初始化后连续运行数个小时。它通常读取数MB到数TB的数据，之后进行转化或者分析，最后把结果写回到集群中。集群B主要用于处理当前的生产数据。集群B的任务持续的时间更长，在很少人工干预的情况下，持续的生成和处理数TB的数据集。在这两个案例中，一个单独的"任务"都是指运行在多个机器上的多个进程，它们同时读取和写入多个文件。

Cluster	A	B
Chunkservers	342	227
可用硬盘空间 已用硬盘空间	72TB 55TB	180TB 155TB
文件数量 死文件数量 Chunk数量	735k 22k 992k	737k 232k 1550k
chunkserver元数据 master元数据	13GB 48MB	21GB 60MB

表2：两个GFS集群特性

6.2.1 存储

如上表前五行所描述的，两个集群都由上百台Chunk服务器组成，支持数TB的硬盘空间；两个集群虽然都存储了大量的数据，但是还有剩余的空间。"已用空间"包含了所有的Chunk副本。实际上所有的文件都复制了三份。因此，集群实际上各存储了18TB和52TB的文件数据。

两个集群存储的文件数量都差不多，但是集群B上有大量的死文件。所谓"死文件"是指文件被删除了或者是被新版本的文件替换了，但是存储空间还没有来得及被回收。由于集群B存储的文件较大，因此它的Chunk数量也比较多。

6.2.2 元数据

Chunk服务器总共保存了十几GB的元数据，大多数是来自用户数据的、64KB大小的块的Checksum。保存在Chunk服务器上其它的元数据是Chunk的版本号信息，我们在4.5节描述过。

在Master服务器上保存的元数据就小的多了，大约只有数十MB，或者说平均每个文件100字节的元数据。这和我们设想的是一样的，Master服务器的内存大小在实际应用中并不会成为GFS系统容量的瓶颈。大多数文件的元数据都是以前缀压缩模式存放的文件名。Master服务器上存放的其它元数据包括了文件的所有者和权限、文件到Chunk的映射关系，以及每一个Chunk的当前版本号。此外，针对每一个Chunk，我们都保存了当前的副本位置以及对它的引用计数，这个引用计数用于实现写时拷贝*（alex注：即COW，copy-on-write）*。

对于每一个单独的服务器，无论是Chunk服务器还是Master服务器，都只保存了50MB到100MB的元数据。因此，恢复服务器是非常快速的：在服务器响应客户请求之前，只需要花几秒钟时间从磁盘上读取这些数据就可以了。不过，Master服务器会持续颠簸一段时间–通常是30到60秒–直到它完成轮询所有的Chunk服务器，并获取到所有Chunk的位置信息。

6.2.3 读写速率

集群	A	B
读速率（最后一分钟）	583MB/s	380MB/s
读取率（最后一小时）	562MB/s	384MB/s
读速率（自从重启动）	589MB/s	49MB/s
写速率（最后一分钟）	1MB/s	101MB/s
写取率（最后一小时）	1MB/s	117MB/s
写速率（自从重启动）	25MB/s	13MB/s
master操作（最后一分钟）	325Ops/s	533Ops/s
master操作（最后一小时）	381Ops/s	518Ops/s
master操作（自从重启动）	202Ops/s	347Ops/s

表三：两个GFS集群的性能表

表三显示了不同时段的读写速率。在测试的时候，这两个集群都运行了一周左右的时间。（这两个集群最近都因为升级新版本的GFS重新启动过了）。

集群重新启动后，平均写入速率小于30MB/s。当我们提取性能数据的时候，集群B正进行大量的写入操作，写入速度达到了100MB/s，并且因为每个Chunk都有三个副本的原因，网络负载达到了300MB/s。读取速率要比写入速率高的多。正如我们设想的那样，总的工作负载中，读取的比例远远高于写入的比例。两个集群都进行着繁重的读取操作。特别是，集群A在一周时间内都维持了580MB/s的读取速度。集群A的网络配置可以支持750MB/s的速度，显然，它有效的利用了资源。集群B支持的峰值读取速度是1300MB/s，但是它的应用只用到了380MB/s。

6.2.4 Master服务器的负载

表3的数据显示了发送到Master服务器的操作请求大概是每秒钟200到500个。Master服务器可以轻松的应付这个请求速度，所以Master服务器的处理能力不是系统的瓶颈。

在早期版本的GFS中，Master服务器偶尔会成为瓶颈。它大多数时间里都在顺序扫描某个很大的目录（包含数万个文件）去查找某个特定的文件。因此我们修改了Master服务器的数据结构，通过对名字空间进行二分查找来提高效率。现在Master服务器可以轻松的每秒钟进行数千次文件访问。如果有需要的话，我们可以通过在名称空间数据结构之前设置名称查询缓冲的方式进一步提高速度。

6.2.5 恢复时间

当某个Chunk服务器失效了，一些Chunk副本的数量可能会低于复制因子指定的数量，我们必须通过克隆副本使Chunk副本数量达到复制因子指定的数量。恢复所有Chunk副本所花费的时间取决于资源的数量。在我们的试验中，我们把集群B上的一个Chunk服务器Kill掉。这个Chunk服务器上大约有15000个Chunk，共计600GB的数据。为了减小克隆操作对正在运行的应用程序的影响，以及为GFS调度决策提供修正空间，我们缺省的把集群中并发克隆操作的数量设置为91个（Chunk服务器的数量的40%），每个克隆操作最多允许使用的带宽是6.25MB/s（50mbps）。所有的Chunk在23.2分钟内恢复了，复制的速度高达440MB/s。

在另外一个测试中，我们Kill掉了两个Chunk服务器，每个Chunk服务器大约有16000个Chunk，共计660GB的数据。这两个故障导致了266个Chunk只有单个副本。这266个Chunk被GFS优先调度进行复制，在2分钟内恢复到至少有两个副本；现在集群被带入到另外一个状态，在这个状态下，系统可以容忍另外一个Chunk服务器失效而不丢失数据。

6.3 工作负荷分析(Workload Breakdown)

本节中，我们展示了对两个GFS集群工作负载情况的详细分析，这两个集群和6.2节中的类似，但是不完全相同。集群X用于研究和开发，集群Y用于生产数据处理。

6.3.1 方法论和注意事项

本章节列出的这些结果数据只包括客户机发起的原始请求，因此，这些结果能够反映我们的应用程序对GFS文件系统产生的全部工作负载。它们不包含那些为了实现客户端请求而在服务器间交互的请求，也不包含GFS内部的后台活动相关的请求，比如前向转发的写操作，或者重新负载均衡等操作。

我们从GFS服务器记录的真实的RPC请求日志中推导重建出关于IO操作的统计信息。例如，GFS客户程序可能会把一个读操作分成几个RPC请求来提高并行度，我们可以通过这些RPC请求推导出原始的读操作。因为我们的访问模式是高度程式化，所以我们认为任何不符合的数据都是误差*(alex注：Since our access patterns are highly stylized, we expect any error to be in the noise)*。应用程序如果能够记录更详尽的日志，就有可能提供更准确的诊断数据；但是为了这个目的去重新编译和重新启动数千个正在运行的客户机是不现实的，而且从那么多客户机上收集结果也是个繁重的工作。

应该避免从我们的工作负荷数据中过度的归纳出普遍的结论*(alex注：即不要把本节的数据作为基础的指导性数据)*。因为Google完全控制着GFS和使用GFS的应用程序，所以，应用程序都针对GFS做了优化，同时，GFS也是为了这些应用程序而设计的。这样的相互作用也可能存在于一般程序和文件系统中，但是在我们的案例中这样的作用影响可能更显著。

6.3.2 Chunk服务器工作负荷

操作	读		写		纪录增加	
集群	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

表4：操作大小百分比细目（%）。对于读操作，大小是实际读取的数据和传送的数据，而不是请求的数据量。

表4显示了操作按涉及的数据量大小的分布情况。读取操作按操作涉及的数据量大小呈现了双峰分布。小的读取操作（小于64KB）一般是由查找操作的客户端发起的，目的在于从巨大的文件中查找小块的数据。大的读取操作（大于512KB）一般是从头到尾顺序的读取整个文件。

在集群Y上，有相当数量的读操作没有返回任何的数据。在我们的应用中，尤其是在生产系统中，经常使用文件作为生产者-消费者队列。生产者并行的向文件中追加数据，同时，消费者从文件的尾部读取数据。某

些情况下，消费者读取的速度超过了生产者写入的速度，这就会导致没有读到任何数据的情况。集群X通常用于短暂的数据分析任务，而不是长时间运行的分布式应用，因此，集群X很少出现这种情况。

写操作按数据量大小也同样呈现为双峰分布。大的写操作（超过256KB）通常是由于Writer使用了缓存机制导致的。Writer缓存较小的数据，通过频繁的Checkpoint或者同步操作，或者只是简单的统计小的写入（小于64KB）的数据量*(alex注：即汇集多次小的写入操作，当数据量达到一个阈值，一次写入)*，之后批量写入。

再来观察一下记录追加操作。我们可以看到集群Y中大的记录追加操作所占比例比集群X多的多，这是因为集群Y用于我们的生产系统，针对GFS做了更全面的调优。

操作	读		写		纪录增加	
集群	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	0.1	46.8
1M...inf	6.4	30.1	3.3	28.0	53.9	7.4

表5：操作大小字节传输细目表（%）。对于读取来说，这个大小是实际读取的并且传输的，而不是请求读取的数量。两个不同点是如果读取尝试读超过文件结束的大小，那么实际读取大小就不是试图读取的大小，尝试读取超过文件结束的大小这样的操作在我们的负载中并不常见。

表5显示了按操作涉及的数据量的大小统计出来的总数据传输量。在所有的操作中，大的操作（超过256KB）占据了主要的传输量。小的读取（小于64KB）虽然传输的数据量比较少，但是在读取的数据量中仍占了相当的比例，这是因为在文件中随机Seek的工作负荷而导致的。

6.3.3 记录追加 vs. 写操作

记录追加操作在我们的生产系统中大量使用。对于集群X，记录追加操作和普通写操作的比例按照字节比是108:1，按照操作次数比是8:1。对于作为我们的生产系统的集群Y来说，这两个比例分别是3.7:1和2.5:1。更进一步，这一组数据说明在我们的两个集群上，记录追加操作所占比例都要比写操作要大。对于集群X，在整个测量过程中，记录追加操作所占比率都比较低，因此结果会受到一两个使用某些特定大小的buffer的应用程序的影响。

如同我们所预期的，我们的数据修改操作主要是记录追加操作而不是覆盖方式的写操作。我们测量了第一个副本的数据覆盖写的情况。这近似于一个客户机故意覆盖刚刚写入的数据，而不是增加新的数据。对于集群X，覆盖写操作在写操作所占据字节上的比例小于0.0001%，在所占据操作数量上的比例小于0.0003%。对于集群Y，这两个比率都是0.05%。虽然这只是某一片断的情况，但是仍然高于我们的预期。这是由于这些覆盖写的操作，大部分是由于客户端在发生错误或者超时以后重试的情况。这在本质上应该不算作工作负荷的一部分，而是重试机制产生的结果。

6.3.4 Master的工作负荷

集群	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
其他	0.5	0.8

表6：master请求类型明细（%）

表6显示了Master服务器上的请求按类型区分的明细表。大部分的请求都是读取操作查询Chunk位置信息（FindLocation）、以及修改操作查询lease持有者的信息（FindLease-Locker）。

集群X和Y在删除请求的数量上有着明显的不同，因为集群Y存储了生产数据，一般会重新生成数据以及用新版本的数据替换旧有的数据。数量上的差异也被隐藏在了Open请求中，因为旧版本的文件可能在以重新写入的模式打开时，隐式的被删除了（类似UNIX的open函数中的“w”模式）。

FindMatchingFiles是一个模式匹配请求，支持“ls”以及其它类似的文件系统操作。不同于Master服务器的其它请求，它可能会检索namespace的大部分内容，因此是非常昂贵的操作。集群Y的这类请求要多一些，因为自动化数据处理的任务进程需要检查文件系统的各个部分，以便从全局上了解应用程序的状态。与之不同的是，集群X的应用程序更加倾向于由单独的用户控制，通常预先知道自己所需要使用的全部文件的名称。

7. 经验

在建造和部署GFS的过程中，我们经历了各种各样的问题，有些是操作上的，有些是技术上的。

起初，GFS被设想为我们的生产系统的后端文件系统。随着时间推移，在GFS的使用中逐步的增加了对研究和开发任务的支持。我们开始增加一些小的功能，比如权限和配额，到了现在，GFS已经初步支持了这些功能。虽然我们生产系统是严格受控的，但是用户层却不总是这样的。需要更多的基础架构来防止用户间的相互干扰。

我们最大的问题是磁盘以及和Linux相关的问题。很多磁盘都声称它们支持某个范围内的Linux IDE硬盘驱动程序，但是实际应用中反映出来的情况却不是这样，它们只支持最新的驱动。因为协议版本很接近，所以大部分磁盘都可以用，但是偶尔也会有由于协议不匹配，导致驱动和内核对于驱动器的状态判断失误。这会导致数据因为内核中的问题意外的被破坏了。这个问题促使我们使用Checksum来校验数据，同时我们也修改内核来处理这些因为协议不匹配带来的问题。

较早的时候，我们在使用Linux 2.2内核时遇到了些问题，主要是fsync()的效率问题。它的效率与文件的大小而不是文件修改部分的大小有关。这在我们的操作日志文件过大时给出了难题，尤其是在我们尚未实现Checkpoint的时候。我们费了很大的力气用同步写来解决这个问题，但是最后还是移植到了Linux2.4内核上。

另一个和Linux相关的问题是单个读写锁的问题，也就是说，在某一个地址空间的任意一个线程都必须在从磁盘page in（读锁）的时候先hold住，或者在mmap()调用（写锁）的时候改写地址空间。我们发现即使我们的系统负载很轻的情况下也会有偶尔的超时，我们花费了很多的精力去查找资源的瓶颈或者硬件的问题。最后我们终于发现这个单个锁在磁盘线程交换以前映射的数据到磁盘的时候，锁住了当前的网络线程，阻止它把新数据映射到内存。由于我们的性能主要受限于网络接口，而不是内存copy的带宽，因此，我们用pread()替代mmap()，用了一个额外的copy动作来解决这个问题。

尽管偶尔还是有其它的问题，Linux的开放源代码还是使我们能够快速探究和理解系统的行为。在适当的

时候，我们会改进内核并且和公开源码组织共享这些改动。

8. 相关工作

和其它的大型分布式文件系统，比如AFS[5]类似，GFS提供了一个与位置无关的名字空间，这使得数据可以为了负载均衡或者灾难冗余等目的在不同位置透明的迁移。不同于AFS的是，GFS把文件分布存储到不同的服务器上，这种方式更类似Xfs[1]和Swift[3]，这是为了提高整体性能以及灾难冗余的能力。

由于磁盘相对来说比较便宜，并且复制的方式比RAID[9]方法简单的多，GFS目前只使用复制的方式来进行冗余，因此要比xFS或者Swift占用更多的裸存储空间*(alex注：Raw storage，裸盘的空间)*。

与AFS、xFS、Frangipani[12]以及Intermezzo[6]等文件系统不同的是，GFS并没有在文件系统层面提供任何Cache机制。我们主要的工作在单个应用程序执行的时候几乎不会重复读取数据，因为它们的工作方式要么是流式的读取一个大型的数据集，要么是在大型的数据集中随机Seek到某个位置，之后每次读取少量的数据。

某些分布式文件系统，比如Frangipani、xFS、Minnesota's GFS[11]、GPFS[10]，去掉了中心服务器，只依赖于分布式算法来保证一致性和可管理性。我们选择了中心服务器的方法，目的是为了简化设计，增加可靠性，能够灵活扩展。特别值得一提的是，由于处于中心位置的Master服务器保存有几乎所有的Chunk相关信息，并且控制着Chunk的所有变更，因此，它极大地简化了原本非常复杂的Chunk分配和复制策略的实现方法。我们通过减少Master服务器保存的状态信息的数量，以及将Master服务器的状态复制到其它节点来保证系统的灾难冗余能力。扩展能力和高可用性（对于读取）目前是通过我们的影子Master服务器机制来保证的。对Master服务器状态更改是通过预写日志的方式实现持久化。为此，我们可以调整为使用类似Harp[7]中的primary-copy方案，从而提供比我们现在的方案更严格的一致性保证。

我们解决了一个难题，这个难题类似Lustre[8]在如何在有大量客户端时保障系统整体性能遇到的问题。不过，我们通过只关注我们的应用程序的需求，而不是提供一个兼容POSIX的文件系统，从而达到了简化问题的目的。此外，GFS设计预期是使用大量的不可靠节点组建集群，因此，灾难冗余方案是我们设计的核心。

GFS很类似NASD架构[4]。NASD架构是基于网络磁盘的，而GFS使用的是普通计算机作为Chunk服务器，就像NASD原形中方案一样。所不同的是，我们的Chunk服务器采用惰性分配固定大小的Chunk的方式，而不是分配变长的对象存储空间。此外，GFS实现了诸如重新负载均衡、复制、恢复机制等等在生产环境中需要的特性。

不同于与Minnesota's GFS和NASD，我们并不改变存储设备的Model*(alex注：对这两个文件系统不了解，因为不太明白改变存储设备的Model用来做什么，这不明白这个model是模型、还是型号)*。我们只关注用普通的设备来解决非常复杂的分布式系统日常的数据处理。

我们通过原子的记录追加操作实现了生产者-消费者队列，这个问题类似River[2]中的分布式队列。River使用的是跨主机的、基于内存的分布式队列，为了实现这个队列，必须仔细控制数据流；而GFS采用可以被生产者并发追加记录的持久化的文件的方式实现。River模式支持m-到-n的分布式队列，但是缺少由持久化存储提供的容错机制，GFS只支持m-到-1的队列。多个消费者可以同时读取一个文件，但是它们输入流的区间必须是对齐的。

9. 结束语

Google文件系统展示了一个使用普通硬件支持大规模数据处理的系统的特质。虽然一些设计要点都是针对我们的特殊的需要定制的，但是还是有很多特性适用于类似规模的和成本的数据处理任务。

首先，我们根据我们当前的和可预期的将来的应用规模和技术环境来评估传统的文件系统的特性。我们的评估结果将我们引导到一个使用完全不同于传统的设计思路上。根据我们的设计思路，我们认为组件失效是常态而不是异常，针对采用追加方式（有可能是并发追加）写入、然后再读取（通常序列化读取）的大文件进行优化，以及扩展标准文件系统接口、放松接口限制来改进整个系统。

我们系统通过持续监控，复制关键数据，快速和自动恢复提供灾难冗余。Chunk复制使得我们可以对Chunk服务器的失效进行容错。高频率的组件失效要求系统具备在线修复机制，能够周期性的、透明的修复损坏的数据，也能够第一时间重新建立丢失的副本。此外，我们使用Checksum在磁盘或者IDE子系统级别检测数据损坏，在这样磁盘数量惊人的大系统中，损坏率是相当高的。

我们的设计保证了在有大量的并发读写操作时能够提供很高的合计吞吐量。我们通过分离控制流和数据流来实现这个目标，控制流在Master服务器处理，而数据流在Chunk服务器和客户端处理。当一般的操作涉及到Master服务器时，由于GFS选择的Chunk尺寸较大*(alex注：从而减小了元数据的大小)*，以及通过Chunk Lease将控制权限移交给主副本，这些措施将Master服务器的负担降到最低。这使得一个简单、中心的Master不会成为成为瓶颈。我们相信我们对网络协议栈的优化可以提升当前对于每客户端的写入吞吐量限制。

GFS成功的实现了我们对存储的需求，在Google内部，无论是作为研究和开发的存储平台，还是作为生产系统的数据处理平台，都得到了广泛的应用。它是我们持续创新和处理整个WEB范围内的难题的一个重要工具。

致谢

We wish to thankt he following people for their contributions to the system or the paper. Brain Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeff Dean, and David des-Jardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Edjlali worked on storage quota. Markus Gutschke worked on a testing frameworkan d security enhancements. David
Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new file system and gave us useful feedback. Yoshka helped with early testing.

参考

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless networkfil e systems. In Proceedings of the 15th ACM Symposium on Operating System Principles, pages 109–126, Copper Mountain Resort, Colorado, December 1995.
[2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99), pages 10–22, Atlanta, Georgia, May 1999.
[3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disks triping to provide high I/O data rates. Computer Systems, 4(4):405–436, 1991.

[4] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, ErikR iedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems, pages 92–103, San Jose, California, October 1998.

[5] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6(1):51–81, February 1988.
[6] InterMezzo. http://www.inter-mezzo.org, 2003.

[7] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In 13th Symposium on Operating System Principles, pages 226–238, Pacific Grove, CA, October 1991.
[8] Lustre. http://www.lustreorg, 2003.

[9] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, pages 109–116, Chicago, Illinois, September 1988.
[10] FrankS chmuck and Roger Haskin. GPFS: A shared-diskfi le system for large computing clusters. In Proceedings of the First USENIX Conference on File and Storage Technologies, pages 231–244, Monterey, California, January 2002.

[11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T.O'Keefe. The Gobal File System. In Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, College Park, Maryland, September 1996.
[12] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In Proceedings of the 16th ACM Symposium on Operating System Principles, pages 224–237, Saint-Malo, France, October 1997
分类: Google论文 标签:

MuleSoft公司的CloudCat支持在Amazon EC2和GoGrid的云上部署Web应用

2010年3月27日 blademaster 没有评论

作者 **Srini Penchikala** 译者 侯伯薇 发布于 2010年3月22日 上午8时33分

CloudCat是一种作为Apache Tomcat的servlet容器的云服务产品，它提供了虚拟镜像，允许开发者和QA团队在云环境中构建和测试web应用程序。MuleSoft，也就是创建了Mule ESB的公司，最近发布了CloudCat产品，它可以被用做在物理内部服务器上托管Tomcat的一种选择。MuleSoft还宣布，与云基础架构托管提供商GoGrid达成合作伙伴关系，从而以云服务的形式来提供CloudCat。他们之间的组合为开发人员提供了一种方式，可以同时提供云计算和开源软件的好处。

当还没有CloudCat的时候，在云中使用Apache Tomcat除了要安装其它必要的软件之外，还需要对Tomcat进行手动的安装和配置。通过在CloudCat中使用预配置的Apache Tomcat镜像，开发者和操作团队就可以在云中部署和测试他们的web应用程序，而不需要投资购买并存放物理服务器。目前CloudCat已经在Amazon Elastic Compute Cloud（Amazon EC2 AMI）和GoGrid（GoGrid GSI）的云环境中以云服务的形式提供。它包括了运行在Linux（在GoGrid使用Redhat，EC2上使用Ubuntu）和MySQL上的Apache Tomcat 6服务器。

Cloudcat服务器的主要特性包括：

- 为开发和测试Tomcat应用程序提供了Cloudcat。
- 使用MuleSoft的Tcat服务控制台提供了Cloudcat运行时的诊断工具
- 与Apache Maven集成，在开发和测试环境之间提供持续集成
- 与Tcat服务器的REST API集成，提供管理和控制
- 针对Tomcat应用程序的请求式运行时能力以及远程重启的能力。

InfoQ对Mulesoft的产品管理主管Sateesh Narahari进行采访，向其询问了关于Cloudcat服务器的发布以及新的与GoGrid之间和合作伙伴关系的问题。

InfoQ：发布CloudCat的主要动机是什么呢？

> 在Mulesoft，我们拥有唯一的主要动机，它推动了所有一切工作，那就是创建出在企业中和在云中都易于使用的中间件。这次，我们专注于Apache Tomcat，那是我们最喜欢的Web应用

> 程序服务器。在开发中Tomcat很可靠，并且应用广泛，但是对于IT管理员来说，却很难在生产环境中来管理，因为缺少好的操作工具和商业化的技术支持。当我们想在各种公有云中找到干净的、即时的对Tomcat的支持镜像时，发现根本没有。而CloudCat正是我们填补该项市场空白的初次尝试。通过使用我们的最佳实践和Tomcat的技术秘诀，还有我们在Amazon EC2和GoGrid提供工具以及为Tcat服务器管理服务提供附加价值的经验，我们相信CloudCat会得到试图寻找在云上部署应用的企业的青睐。

InfoQ：CloudCat能够被用于在生产环境中托管web应用程序吗，还是仅限于在云中在开发/QA环境中测试应用程序？

> CloudCat能够用于开发/测试环境，也可以用于生产环境。CloudCat包含了已经验证过的初始化脚本，可以为IT操作员提供可靠和合适的重启以及服务器控制。

InfoQ：CloudCat服务器环境也支持负载平衡吗？在CloudCat中故障排除是怎么做的呢？

> CloudCat可以与已经由云提供商所提供的负载平衡解决方案协同工作。我们在CloudCat自身中不提供负载平衡的能力，但是可以与基础架构提供的能力协同工作。例如，用户可以在EC2中使用Elastic负载平衡。

InfoQ：开发者和QA团队成员能使用新的CloudCat服务器来做性能测试吗？

> 可以。因为我们将CloudCat集成到任何其他云基础架构的提供过程中，这样就节省了提供新的CloudCat实例的时间，并且能够满足开发/QA团队成员使用CloudCat实例做性能测试或者模拟高负载场景的需要。当与Amazon EC2协同工作的时候，CloudCat实例还能够通过可选的EC 2插件从Tcat服务控制台直接创建。

InfoQ：对于开发、单元和集成测试、调试、应用程序概要分析等等，我们为想要在CloudCat环境部署应用程序的开发者提供了什么样的工具作为支持呢？

> 在CloudCat中可以直接使用Tcat服务器，而没有任何附加费用。Tcat服务器为运行在Tomcat实例上的web应用程序提供了深层次的诊断和调试能力。我们为CloudCat实例提供了这些能力，同时也在Tcat服务控制台提供了同样的能力。此外，Tcat服务器还提供了部署的能力，它使得将应用程序从开发环境迁移到测试环境最终到生产环境变得非常容易。

InfoQ：在新的CloudCat服务器上提供了什么样的监控工具呢？

> Cloudcat实例可以从默认的云监控工具中监控，或者还可以从Tcat服务控制台监控。当前，Cloudcat不提供任何警告的能力，但是任何行业领先的支持云的监控工具都能够监控Cloudcat服务实例。

InfoQ：关于新特性，Cloudcat服务器产品将来的路线图是怎样的呢？

> Cloudcat会始终是MuleSoft的战略投资所在。我们期望拓展Amazon EC2和GoGrid之外的云提供商。有了这个版本的Cloudcat，我们已经获得了大量用户反馈，那会对产品的路线图产生影响。我们还在寻找更易于在私有云中使用Cloudcat的方法。我们将会在未来和合作伙伴一起在这个领域发布更激动人心的产品。

查看英文原文：MuleSoft’s CloudCat Supports Web Application Deployment on Amazon EC2 and GoGrid Clouds

分类: 小道消息 标签:

Google MapReduce中文版

2010年3月27日 blademaster 没有评论

Google MapReduce中文版

*译者: **alex***

摘要

MapReduce是一个编程模型，也是一个处理和生成超大数据集的算法模型的相关实现。用户首先创建一个Map函数处理一个基于key/value pair的数据集合，输出中间的基于key/value pair的数据集合；然后再创建一个Reduce函数用来合并所有的具有相同中间key值的中间value值。现实世界中有很多满足上述处理模型的例子，本论文将详细描述这个模型。

MapReduce架构的程序能够在大量的普通配置的计算机上实现并行化处理。这个系统在运行时只关心：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。采用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。

我们的MapReduce实现运行在规模可以灵活调整的由普通机器组成的集群上：一个典型的MapReduce计算往往由几千台机器组成、处理以TB计算的数据。程序员发现这个系统非常好用：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。

1、介绍

在过去的5年里，包括本文作者在内的Google的很多程序员，为了处理海量的原始数据，已经实现了数以百计的、专用的计算方法。这些计算方法用来处理大量的原始数据，比如，文档抓取（类似网络爬虫的程序）、Web请求日志等等；也为了计算处理各种类型的衍生数据，比如倒排索引、Web文档的图结构的各种表示形势、每台主机上网络爬虫抓取的页面数量的汇总、每天被请求的最多的查询的集合等等。大多数这样的数据处理运算在概念上很容易理解。然而由于输入的数据量巨大，因此要想在可接受的时间内完成运算，只有将这些计算分布在成百上千的主机上。如何处理并行计算、如何分发数据、如何处理错误？所有这些问题综合在一起，需要大量的代码处理，因此也使得原本简单的运算变得难以处理。

为了解决上述复杂的问题，我们设计一个新的抽象模型，使用这个抽象模型，我们只要表述我们想要执行的简单运算即可，而不必关心并行计算、容错、数据分布、负载均衡等复杂的细节，这些问题都被封装在了一个库里面。设计这个抽象模型的灵感来自Lisp和许多其他函数式语言的Map和Reduce的原语。我们意识到我们大多数的运算都包含这样的操作：在输入数据的“逻辑”记录上应用Map操作得出一个中间key/value pair集合，然后在所有具有相同key值的value值上应用Reduce操作，从而达到合并中间的数据，得到一个想要的结果的目的。使用MapReduce模型，再结合用户实现的Map和Reduce函数，我们就可以非常容易的实现大规模并行化计算；通过MapReduce模型自带的“再次执行”（re-execution）功能，也提供了初级的容灾实现方案。

这个工作(实现一个MapReduce框架模型)的主要贡献是通过简单的接口来实现自动的并行化和大规模的分布式计算，通过使用MapReduce模型接口实现在大量普通的PC机上高性能计算。

第二部分描述基本的编程模型和一些使用案例。第三部分描述了一个经过裁剪的、适合我们的基于集群的计算环境的MapReduce实现。第四部分描述我们认为在MapReduce编程模型中一些实用的技巧。第五部分对于各种不同的任务，测量我们MapReduce实现的性能。第六部分揭示了在Google内部如何使用MapReduce作为基础重写我们的索引系统产品，包括其它一些使用MapReduce的经验。第七部分讨论相关的和未来的工作。

2、编程模型

MapReduce编程模型的原理是：利用一个输入key/value pair集合来产生一个输出的key/value pair集合。MapReduce库的用户用两个函数表达这个计算：Map和Reduce。

用户自定义的Map函数接受一个输入的key/value pair值，然后产生一个中间key/value pair值的集合。MapReduce库把所有具有相同中间key值I的中间value值集合在一起后传递给reduce函数。

用户自定义的Reduce函数接受一个中间key的值I和相关的一个value值的集合。Reduce函数合并这些value值，形成一个较小的value值的集合。一般的，每次Reduce函数调用只产生0或1个输出value值。通常我们通过一个迭代器把中间value值提供给Reduce函数，这样我们就可以处理无法全部放入内存中的大量的value值的集合。

2.1、例子

例如，计算一个大的文档集合中每个单词出现的次数，下面是伪代码段：

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, “1″);
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Map函数输出文档中的每个词、以及这个词的出现次数(在这个简单的例子里就是1)。Reduce函数把Map函数产生的每一个特定的词的计数累加起来。

另外，用户编写代码，使用输入和输出文件的名字、可选的调节参数来完成一个符合MapReduce模型规范的对象，然后调用MapReduce函数，并把这个规范对象传递给它。用户的代码和MapReduce库链接在一起(用C++实现)。附录A包含了这个实例的全部程序代码。

2.2、类型

尽管在前面例子的伪代码中使用了以字符串表示的输入输出值，但是在概念上，用户定义的Map和Reduce函数都有相关联的类型：

```
map(k1,v1) ->list(k2,v2)
  reduce(k2,list(v2)) ->list(v2)
```

比如，输入的key和value值与输出的key和value值在类型上推导的域不同。此外，中间key和value值与输出key和value值在类型上推导的域相同。

（alex注：原文中这个domain的含义不是很清楚，我参考Hadoop、KFS等实现，map和reduce都使用了泛型，因此，我把domain翻译成类型推导的域）。
我们的C++中使用字符串类型作为用户自定义函数的输入输出，用户在自己的代码中对字符串进行适当的类型转换。

2.3、更多的例子

这里还有一些有趣的简单例子，可以很容易的使用MapReduce模型来表示：

- 分布式的Grep：Map函数输出匹配某个模式的一行，Reduce函数是一个恒等函数，即把中间数据复制到输出。
- 计算URL访问频率：Map函数处理日志中web页面请求的记录，然后输出(URL,1)。Reduce函数把相同URL的value值都累加起来，产生(URL,记录总数)结果。
- 倒转网络链接图：Map函数在源页面（source）中搜索所有的链接目标（target）并输出为

(target,source)。Reduce函数把给定链接目标（target）的链接组合成一个列表，输出(target,list(source))。

- 每个主机的检索词向量：检索词向量用一个(词,频率)列表来概述出现在文档或文档集中的最重要的一些词。Map函数为每一个输入文档输出(主机名,检索词向量)，其中主机名来自文档的URL。Reduce函数接收给定主机的所有文档的检索词向量，并把这些检索词向量加在一起，丢弃掉低频的检索词，输出一个最终的(主机名,检索词向量)。
- 倒排索引：Map函数分析每个文档输出一个(词,文档号)的列表，Reduce函数的输入是一个给定词的所有（词，文档号），排序所有的文档号，输出(词,list（文档号）)。所有的输出集合形成一个简单的倒排索引，它以一种简单的算法跟踪词在文档中的位置。
- 分布式排序：Map函数从每个记录提取key，输出(key,record)。Reduce函数不改变任何的值。这个运算依赖分区机制(在4.1描述)和排序属性(在4.2描述)。

3、实现

MapReduce模型可以有多种不同的实现方式。如何正确选择取决于具体的环境。例如，一种实现方式适用于小型的共享内存方式的机器，另外一种实现方式则适用于大型NUMA架构的多处理器的主机，而有的实现方式更适合大型的网络连接集群。
本章节描述一个适用于Google内部广泛使用的运算环境的实现：用以太网交换机连接、由普通PC机组成的大型集群。在我们的环境里包括：
1.x86架构、运行Linux操作系统、双处理器、2-4GB内存的机器。
2.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。 *（alex注：这里需要网络专家解释一下了）*
3.集群中包含成百上千的机器，因此，机器故障是常态。
4.存储为廉价的内置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来在不可靠的硬件上保证数据的可靠性和有效性。
5.用户提交工作（job）给调度系统。每个工作（job）都包含一系列的任务（task），调度系统将这些任务调度到集群中多台可用的机器上。

3.1、执行概括

通过将Map调用的输入数据自动分割为M个数据片段的集合，Map调用被分布到多台机器上执行。输入的数据片段能够在不同的机器上并行处理。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（例如，hash(key) mod R），Reduce调用也被分布到多台机器上执行。分区数量（R）和分区函数由用户来指定。

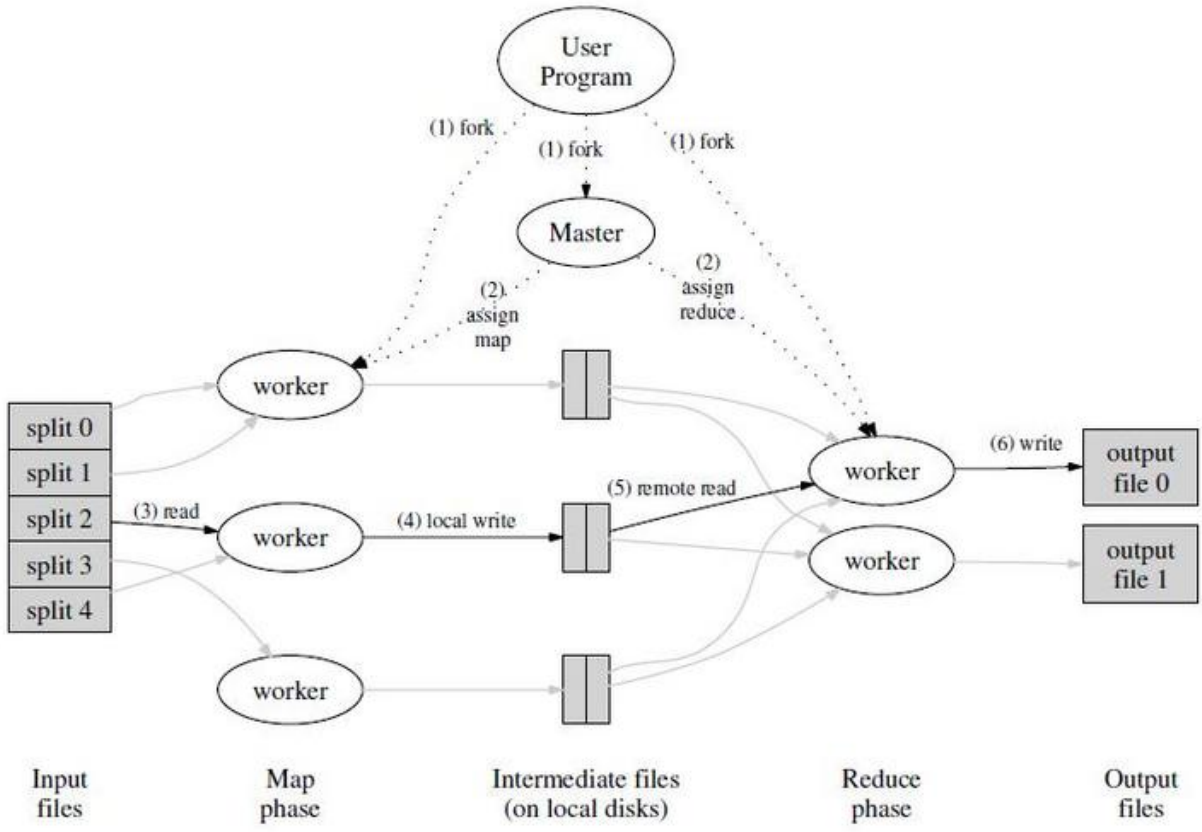


Figure 1: Execution overview

图1展示了我们的MapReduce实现中操作的全部流程。当用户调用MapReduce函数时，将发生下面的一系列动作（下面的序号和图1中的序号一一对应）：

1.用户程序首先调用的MapReduce库将输入文件分成M个数据片度，每个数据片段的大小一般从 16MB到64MB(可以通过可选的参数来控制每个数据片段的大小)。然后用户程序在机群中创建大量的程序副本。 *（alex：copies of the program还真难翻译）*

2.这些程序副本中的有一个特殊的程序–master。副本中其它的程序都是worker程序，由master分配任务。有M个Map任务和R个Reduce任务将被分配，master将一个Map任务或Reduce任务分配给一个空闲的worker。

3.被分配了map任务的worker程序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 pair，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，并缓存在内存中。

4.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负责把这些存储位置再传送给Reduce worker。

5.当Reduce worker程序接收到master程序发来的数据存储位置信息后，使用RPC从Map worker所在主机的磁盘上读取这些缓存数据。当Reduce worker读取了所有的中间数据后，通过对key进行排序后使得具有相同key值的数据聚合在一起。由于许多不同的key值会映射到相同的Reduce任务上，因此必须进行排序。如果中间数据太大无法在内存中完成排序，那么就要在外部进行排序。

6.Reduce worker程序遍历排序后的中间数据，对于每一个唯一的中间key值，Reduce worker程序将这个key值和它相关的中间value值的集合传递给用户自定义的Reduce函数。Reduce函数的输出被追加到所属分区的输出文件。

7.当所有的Map和Reduce任务都完成之后，master唤醒用户程序。在这个时候，在用户程序里的对MapReduce调用才返回。

在成功完成任务之后，MapReduce的输出存放在R个输出文件中（对应每个Reduce任务产生一个输出文件，文件名由用户指定）。一般情况下，用户不需要将这R个输出文件合并成一个文件–他们经常把这些文

件作为另外一个MapReduce的输入，或者在另外一个可以处理多个分割文件的分布式应用中使用。

3.2、Master数据结构

Master持有一些数据结构，它存储每一个Map和Reduce任务的状态（空闲、工作中或完成），以及Worker机器(非空闲任务的机器)的标识。

Master就像一个数据管道，中间文件存储区域的位置信息通过这个管道从Map传递到Reduce。因此，对于每个已经完成的Map任务，master存储了Map任务产生的R个中间文件存储区域的大小和位置。当Map任务完成时，Master接收到位置和大小的更新信息，这些信息被逐步递增的推送给那些正在工作的Reduce任务。

3.3、容错

因为MapReduce库的设计初衷是使用由成百上千的机器组成的集群来处理超大规模的数据，所以，这个库必须要能很好的处理机器故障。

worker故障

master周期性的ping每个worker。如果在一个约定的时间范围内没有收到worker返回的信息，master将把这个worker标记为失效。所有由这个失效的worker完成的Map任务被重设为初始的空闲状态，之后这些任务就可以被安排给其他的worker。同样的，worker失效时正在运行的Map或Reduce任务也将被重新置为空闲状态，等待重新调度。

当worker故障时，由于已经完成的Map任务的输出存储在这台机器上，Map任务的输出已不可访问了，因此必须重新执行。而已经完成的Reduce任务的输出存储在全局文件系统上，因此不需要再次执行。

当一个Map任务首先被worker A执行，之后由于worker A失效了又被调度到worker B执行，这个“重新执行”的动作会被通知给所有执行Reduce任务的worker。任何还没有从worker A读取数据的Reduce任务将从worker B读取数据。

MapReduce可以处理大规模worker失效的情况。比如，在一个MapReduce操作执行期间，在正在运行的集群上进行网络维护引起80台机器在几分钟内不可访问了，MapReduce master只需要简单的再次执行那些不可访问的worker完成的工作，之后继续执行未完成的任务，直到最终完成这个MapReduce操作。

master失败

一个简单的解决办法是让master周期性的将上面描述的数据结构*（alex注：指3.2节）*的写入磁盘，即检查点（checkpoint）。如果这个master任务失效了，可以从最后一个检查点（checkpoint）开始启动另一个master进程。然而，由于只有一个master进程，master失效后再恢复是比较麻烦的，因此我们现在的实现是如果master失效，就中止MapReduce运算。客户可以检查到这个状态，并且可以根据需要重新执行MapReduce操作。

在失效方面的处理机制

（alex注：原文为”semantics in the presence of failures”）

当用户提供的Map和Reduce操作是输入确定性函数（即相同的输入产生相同的输出）时，我们的分布式实现在任何情况下的输出都和所有程序没有出现任何错误、顺序的执行产生的输出是一样的。

我们依赖对Map和Reduce任务的输出是原子提交的来完成这个特性。每个工作中的任务把它的输出写到私有的临时文件中。每个Reduce任务生成一个这样的文件，而每个Map任务则生成R个这样的文件（一个Reduce任务对应一个文件）。当一个Map任务完成的时，worker发送一个包含R个临时文件名的完成消息给master。如果master从一个已经完成的Map任务再次接收到到一个完成消息，master将忽略这个消息；否则，master将这R个文件的名字记录在数据结构里。

当Reduce任务完成时，Reduce worker进程以原子的方式把临时文件重命名为最终的输出文件。如果同

一个Reduce任务在多台机器上执行，针对同一个最终的输出文件将有多个重命名操作执行。我们依赖底层文件系统提供的重命名操作的原子性来保证最终的文件系统状态仅仅包含一个Reduce任务产生的数据。

使用MapReduce模型的程序员可以很容易的理解他们程序的行为，因为我们绝大多数的Map和Reduce操作是确定性的，而且存在这样的一个事实：我们的失效处理机制等价于一个顺序的执行的操作。当Map或/和Reduce操作是不确定性的时候，我们提供虽然较弱但是依然合理的处理机制。当使用非确定操作的时候，一个Reduce任务R1的输出等价于一个非确定性程序顺序执行产生时的输出。但是，另一个Reduce任务R2的输出也许符合一个不同的非确定顺序程序执行产生的R2的输出。

考虑Map任务M和Reduce任务R1、R2的情况。我们设定e(Ri)是Ri已经提交的执行过程（有且仅有一个这样的执行过程）。当e(R1)读取了由M一次执行产生的输出，而e(R2)读取了由M的另一次执行产生的输出，导致了较弱的失效处理。

3.4、存储位置

在我们的计算运行环境中，网络带宽是一个相当匮乏的资源。我们通过尽量把输入数据(由GFS管理)存储在集群中机器的本地磁盘上来节省网络带宽。GFS把每个文件按64MB一个Block分隔，每个Block保存在多台机器上，环境中就存放了多份拷贝(一般是3个拷贝)。MapReduce的master在调度Map任务时会考虑输入文件的位置信息，尽量将一个Map任务调度在包含相关输入数据拷贝的机器上执行；如果上述努力失败了，master将尝试在保存有输入数据拷贝的机器附近的机器上执行Map任务(例如，分配到一个和包含输入数据的机器在一个switch里的worker机器上执行)。当在一个足够大的cluster集群上运行大型MapReduce操作的时候，大部分的输入数据都能从本地机器读取，因此消耗非常少的网络带宽。

3.5、任务粒度

如前所述，我们把Map拆分成了M个片段、把Reduce拆分成R个片段执行。理想情况下，M和R应当比集群中worker的机器数量要多得多。在每台worker机器都执行大量的不同任务能够提高集群的动态的负载均衡能力，并且能够加快故障恢复的速度：失效机器上执行的大量Map任务都可以分布到所有其他的worker机器上去执行。

但是实际上，在我们的具体实现中对M和R的取值都有一定的客观限制，因为master必须执行O(M+R)次调度，并且在内存中保存O(M*R)个状态（对影响内存使用的因素还是比较小的：O(M*R)块状态，大概每对Map任务/Reduce任务1个字节就可以了）。

更进一步，R值通常是由用户指定的，因为每个Reduce任务最终都会生成一个独立的输出文件。实际使用时我们也倾向于选择合适的M值，以使得每一个独立任务都是处理大约16M到64M的输入数据（这样，上面描写的输入数据本地存储优化策略才最有效），另外，我们把R值设置为我们想使用的worker机器数量的小的倍数。我们通常会用这样的比例来执行MapReduce：M=200000，R=5000，使用2000台worker机器。

3.6、备用任务

影响一个MapReduce的总执行时间最通常的因素是“落伍者”：在运算过程中，如果有一台机器花了很长的时间才完成最后几个Map或Reduce任务，导致MapReduce操作总的执行时间超过预期。出现“落伍者”的原因非常多。比如：如果一个机器的硬盘出了问题，在读取的时候要经常的进行读取纠错操作，导致读取数据的速度从30M/s降低到1M/s。如果cluster的调度系统在这台机器上又调度了其他的任务，由于CPU、内存、本地硬盘和网络带宽等竞争因素的存在，导致执行MapReduce代码的执行效率更加缓慢。我们最近遇到的一个问题是由于机器的初始化代码有bug，导致关闭了的处理器的缓存：在这些机器上执行任务的性能和正常情况相差上百倍。

我们有一个通用的机制来减少“落伍者”出现的情况。当一个MapReduce操作接近完成的时候，master调

度备用（backup）任务进程来执行剩下的、处于处理中状态（in-progress）的任务。无论是最初的执行进程、还是备用（backup）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。我们调优了这个机制，通常只会占用比正常操作多几个百分点的计算资源。我们发现采用这样的机制对于减少超大MapReduce操作的总处理时间效果显著。例如，在5.3节描述的排序任务，在关闭掉备用任务的情况下要多花44%的时间完成排序任务。

4、技巧

虽然简单的Map和Reduce函数提供的基本功能已经能够满足大部分的计算需要，我们还是发掘出了一些有价值的扩展功能。本节将描述这些扩展功能。

4.1、分区函数

MapReduce的使用者通常会指定Reduce任务和Reduce任务输出文件的数量（R）。我们在中间key上使用分区函数来对数据进行分区，之后再输入到后续任务执行进程。一个缺省的分区函数是使用hash方法(比如，hash(key) mod R)进行分区。hash方法能产生非常平衡的分区。然而，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。比如，输出的key值是URLs，我们希望每个主机的所有条目保持在同一个输出文件中。为了支持类似的情况，MapReduce库的用户需要提供专门的分区函数。例如，使用“hash(Hostname(urlkey)) mod R”作为分区函数就可以把所有来自同一个主机的URLs保存在同一个输出文件中。

4.2、顺序保证

我们确保在给定的分区中，中间key/value pair数据的处理顺序是按照key值增量顺序处理的。这样的顺序保证对每个分成生成一个有序的输出文件，这对于需要对输出文件按key值随机存取的应用非常有意义，对在排序输出的数据集也很有帮助。

4.3、Combiner函数

在某些情况下，Map函数产生的中间key值的重复数据会占很大的比重，并且，用户自定义的Reduce函数满足结合律和交换律。在2.1节的词数统计程序是个很好的例子。由于词频率倾向于一个zipf分布(齐夫分布)，每个Map任务将产生成千上万个这样的记录<the,1>。所有的这些记录将通过网络被发送到一个单独的Reduce任务，然后由这个Reduce任务把所有这些记录累加起来产生一个数字。我们允许用户指定一个可选的combiner函数，combiner函数首先在本地将这些记录进行一次合并，然后将合并的结果再通过网络发送出去。

Combiner函数在每台执行Map任务的机器上都会被执行一次。一般情况下，Combiner和Reduce函数是一样的。Combiner函数和Reduce函数之间唯一的区别是MapReduce库怎样控制函数的输出。Reduce函数的输出被保存在最终的输出文件里，而Combiner函数的输出被写到中间文件里，然后被发送给Reduce任务。

部分的合并中间结果可以显著的提高一些MapReduce操作的速度。附录A包含一个使用combiner函数的例子。

4.4、输入和输出的类型

MapReduce库支持几种不同的格式的输入数据。比如，文本模式的输入数据的每一行被视为是一个key/value pair。key是文件的偏移量，value是那一行的内容。另外一种常见的格式是以key进行排序来存储的key/value pair的序列。每种输入类型的实现都必须能够把输入数据分割成数据片段，该数据片段能够由单独的Map任务来进行后续处理(例如，文本模式的范围分割必须确保仅仅在每行的边界进行范围分割)。虽然大多数MapReduce的使用者仅仅使用很少的预定义输入类型就满足要求了，但是使用者依然可以通过提供一个简单的Reader接口实现就能够支持一个新的输入类型。

Reader并非一定要从文件中读取数据，比如，我们可以很容易的实现一个从数据库里读记录的Reader，或者从内存中的数据结构读取数据的Reader。

类似的，我们提供了一些预定义的输出数据的类型，通过这些预定义类型能够产生不同格式的数据。用户采用类似添加新的输入数据类型的方式增加新的输出类型。

4.5、副作用

在某些情况下，MapReduce的使用者发现，如果在Map和/或Reduce操作过程中增加辅助的输出文件会比较省事。我们依靠程序writer把这种“副作用”变成原子的和幂等的*（alex注：幂等的指一个总是产生相同结果的数学运算）*。通常应用程序首先把输出结果写到一个临时文件中，在输出全部数据之后，在使用系统级的原子操作rename重新命名这个临时文件。

如果一个任务产生了多个输出文件，我们没有提供类似两阶段提交的原子操作支持这种情况。因此，对于会产生多个输出文件、并且对于跨文件有一致性要求的任务，都必须是确定性的任务。但是在实际应用过程中，这个限制还没有给我们带来过麻烦。

4.6、跳过损坏的记录

有时候，用户程序中的bug导致Map或者Reduce函数在处理某些记录的时候crash掉，MapReduce操作无法顺利完成。惯常的做法是修复bug后再次执行MapReduce操作，但是，有时候找出这些bug并修复它们不是一件容易的事情；这些bug也许是在第三方库里边，而我们手头没有这些库的源代码。而且在很多时候，忽略一些有问题的记录也是可以接受的，比如在一个巨大的数据集上进行统计分析的时候。我们提供了一种执行模式，在这种模式下，为了保证保证整个处理能继续进行，MapReduce会检测哪些记录导致确定性的crash，并且跳过这些记录不处理。

每个worker进程都设置了信号处理函数捕获内存段异常（segmentation violation）和总线错误（bus error）。在执行Map或者Reduce操作之前，MapReduce库通过全局变量保存记录序号。如果用户程序触发了一个系统信号，消息处理函数将用“最后一口气”通过UDP包向master发送处理的最后一条记录的序号。当master看到在处理某条特定记录不止失败一次时，master就标志着条记录需要被跳过，并且在下次重新执行相关的Map或者Reduce任务的时候跳过这条记录。

4.7、本地执行

调试Map和Reduce函数的bug是非常困难的，因为实际执行操作时不但是分布在系统中执行的，而且通常是在好几千台计算机上执行，具体的执行位置是由master进行动态调度的，这又大大增加了调试的难度。为了简化调试、profile和小规模测试，我们开发了一套MapReduce库的本地实现版本，通过使用本地版本的MapReduce库，MapReduce操作在本地计算机上顺序的执行。用户可以控制MapReduce操作的执行，可以把操作限制到特定的Map任务上。用户通过设定特别的标志来在本地执行他们的程序，之后就可以很容易的使用本地调试和测试工具（比如gdb）。

4.8、状态信息

master使用嵌入式的HTTP服务器（如Jetty）显示一组状态信息页面，用户可以监控各种执行状态。状态信息页面显示了包括计算执行的进度，比如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stderr和stdout文件的链接。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什么时候计算执行的比预期的要慢。

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。这些信息对于调试用户代码中的bug很有帮助。

4.9、计数器

MapReduce库使用计数器统计不同事件发生次数。比如，用户可能想统计已经处理了多少个单词、已经索引的多少篇German文档等等。

为了使用这个特性，用户在程序中创建一个命名的计数器对象，在Map和Reduce函数中相应的增加计数器的值。例如：

```
Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
 for each word w in contents:
  if (IsCapitalized(w)):
   uppercase->Increment();
  EmitIntermediate(w, "1");
```

这些计数器的值周期性的从各个单独的worker机器上传递给master（附加在ping的应答包中传递）。master把执行成功的Map和Reduce任务的计数器值进行累计，当MapReduce操作完成之后，返回给用户代码。

计数器当前的值也会显示在master的状态页面上，这样用户就可以看到当前计算的进度。当累加计数器的值的时候，master要检查重复运行的Map或者Reduce任务，避免重复累加（之前提到的备用任务和失效后重新执行任务这两种情况会导致相同的任务被多次执行）。

有些计数器的值是由MapReduce库自动维持的，比如已经处理的输入的key/value pair的数量、输出的key/value pair的数量等等。

计数器机制对于MapReduce操作的完整性检查非常有用。比如，在某些MapReduce操作中，用户需要确保输出的key value pair精确的等于输入的key value pair，或者处理的German文档数量在处理的整个文档数量中属于合理范围。

5、性能

本节我们用在一个大型集群上运行的两个计算来衡量MapReduce的性能。一个计算在大约1TB的数据中进行特定的模式匹配，另一个计算对大约1TB的数据进行排序。

这两个程序在大量的使用MapReduce的实际应用中是非常典型的 — 一类是对数据格式进行转换，从一种表现形式转换为另外一种表现形式；另一类是从海量数据中抽取少部分的用户感兴趣的数据。

5.1、集群配置

所有这些程序都运行在一个大约由1800台机器构成的集群上。每台机器配置2个2G主频、支持超线程的Intel Xeon处理器，4GB的物理内存，两个160GB的IDE硬盘和一个千兆以太网卡。这些机器部署在一个两层的树形交换网络中，在root节点大概有100-200GBPS的传输带宽。所有这些机器都采用相同的部署（对等部署），因此任意两点之间的网络来回时间小于1毫秒。

在4GB内存里，大概有1-1.5G用于运行在集群上的其他任务。测试程序在周末下午开始执行，这时主机的CPU、磁盘和网络基本上处于空闲状态。

5.2、GREP

这个分布式的grep程序需要扫描大概10的10次方个由100个字节组成的记录，查找出现概率较小的3个字符的模式（这个模式在92337个记录中出现）。输入数据被拆分成大约64M的Block（M=15000），整个输出数据存放在一个文件中（R=1）。

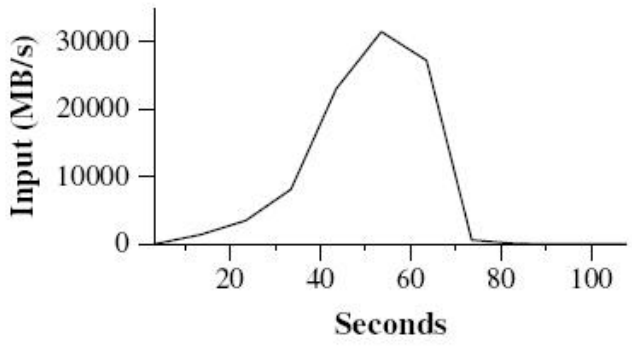


Figure 2: Data transfer rate over time

图2显示了这个运算随时间的处理过程。其中Y轴表示输入数据的处理速度。处理速度随着参与MapReduce计算的机器数量的增加而增加，当1764台worker参与计算的时，处理速度达到了30GB/s。当Map任务结束的时候，即在计算开始后80秒，输入的处理速度降到0。整个计算过程从开始到结束一共花了大概150秒。这包括了大约一分钟的初始启动阶段。初始启动阶段消耗的时间包括了是把这个程序传送到各个worker机器上的时间、等待GFS文件系统打开1000个输入文件集合的时间、获取相关的文件本地位置优化信息的时间。

5.3、排序

排序程序处理10的10次方个100个字节组成的记录（大概1TB的数据）。这个程序模仿TeraSort benchmark[10]。

排序程序由不到50行代码组成。只有三行的Map函数从文本行中解析出10个字节的key值作为排序的key，并且把这个key和原始文本行作为中间的key/value pair值输出。我们使用了一个内置的恒等函数作为Reduce操作函数。这个函数把中间的key/value pair值不作任何改变输出。最终排序结果输出到两路复制的GFS文件系统（也就是说，程序输出2TB的数据）。

如前所述，输入数据被分成64MB的Block（M=15000）。我们把排序后的输出结果分区后存储到4000个文件（R=4000）。分区函数使用key的原始字节来把数据分区到R个片段中。

在这个benchmark测试中，我们使用的分区函数知道key的分区情况。通常对于排序程序来说，我们会增加一个预处理的MapReduce操作用于采样key值的分布情况，通过采样的数据来计算对最终排序处理的分区点。

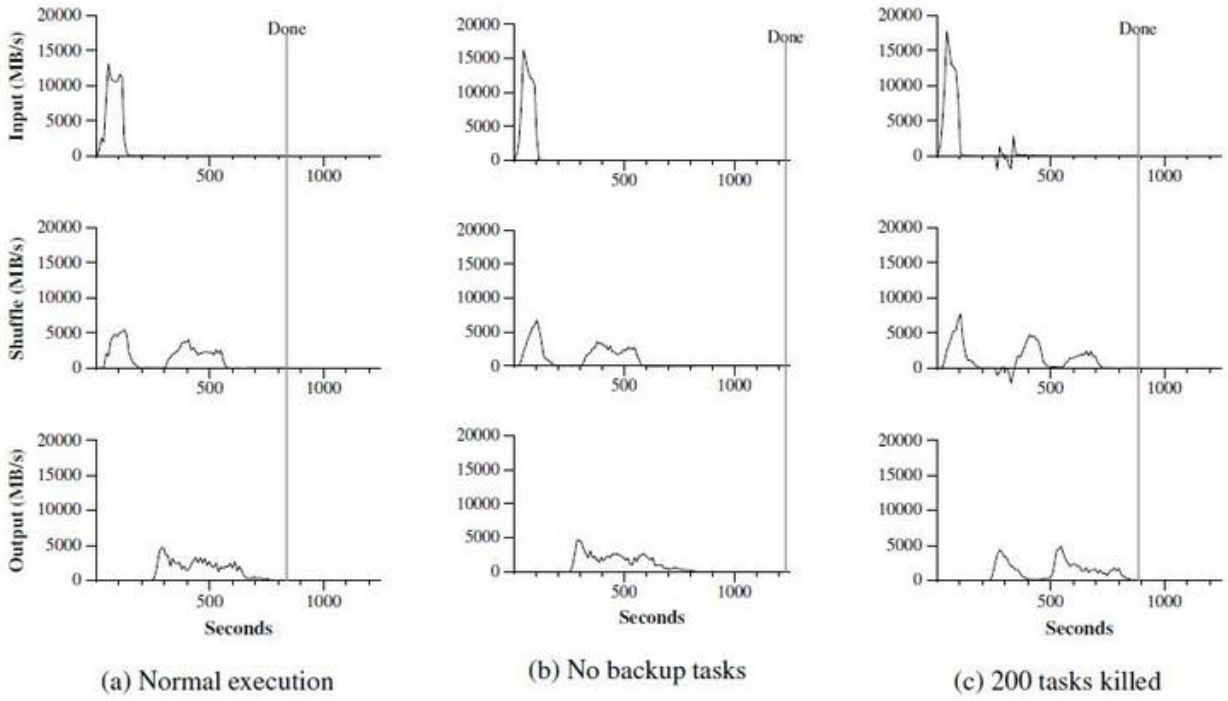


Figure 3: Data transfer rates over time for different executions of the sort program

图三（a）显示了这个排序程序的正常执行过程。左上的图显示了输入数据读取的速度。数据读取速度峰值会达到13GB/s，并且所有Map任务完成之后，即大约200秒之后迅速滑落到0。值得注意的是，排序程序输入数据读取速度小于分布式grep程序。这是因为排序程序的Map任务花了大约一半的处理时间和I/O带宽把中间输出结果写到本地硬盘。相应的分布式grep程序的中间结果输出几乎可以忽略不计。

左边中间的图显示了中间数据从Map任务发送到Reduce任务的网络速度。这个过程从第一个Map任务完成之后就开始缓慢启动了。图示的第一个高峰是启动了第一批大概1700个Reduce任务（整个MapReduce分布到大概1700台机器上，每台机器1次最多执行1个Reduce任务）。排序程序运行大约300秒后，第一批启动的Reduce任务有些完成了，我们开始执行剩下的Reduce任务。所有的处理在大约600秒后结束。

左下图表示Reduce任务把排序后的数据写到最终的输出文件的速度。在第一个排序阶段结束和数据开始写入磁盘之间有一个小的延时，这是因为worker机器正在忙于排序中间数据。磁盘写入速度在2-4GB/s持续一段时间。输出数据写入磁盘大约持续850秒。计入初始启动部分的时间，整个运算消耗了891秒。这个速度和TeraSort benchmark[18]的最高纪录1057秒相差不多。

还有一些值得注意的现象：输入数据的读取速度比排序速度和输出数据写入磁盘速度要高不少，这是因为我们的输入数据本地化优化策略起了作用 — 绝大部分数据都是从本地硬盘读取的，从而节省了网络带宽。排序速度比输出数据写入到磁盘的速度快，这是因为输出数据写了两份（我们使用了2路的GFS文件系统，写入复制节点的原因是为了保证数据可靠性和可用性）。我们把输出数据写入到两个复制节点的原因是因为这是底层文件系统的保证数据可靠性和可用性的实现机制。如果底层文件系统使用类似容错编码[14](erasure coding)的方式而不是复制的方式保证数据的可靠性和可用性，那么在输出数据写入磁盘的时候，就可以降低网络带宽的使用。

5.4、高效的backup任务

图三（b）显示了关闭了备用任务后排序程序执行情况。执行的过程和图3（a）很相似，除了输出数据写磁盘的动作在时间上拖了一个很长的尾巴，而且在这段时间里，几乎没有什么写入动作。在960秒后，只有5个Reduce任务没有完成。这些拖后腿的任务又执行了300秒才完成。整个计算消耗了1283秒，多了44%的执行时间。

5.5、失效的机器

在图三（c）中演示的排序程序执行的过程中，我们在程序开始后几分钟有意的kill了1746个worker中的200个。集群底层的调度立刻在这些机器上重新开始新的worker处理进程（因为只是worker机器上的处理进程被kill了，机器本身还在工作）。

图三（c）显示出了一个“负”的输入数据读取速度，这是因为一些已经完成的Map任务丢失了（由于相应的执行Map任务的worker进程被kill了），需要重新执行这些任务。相关Map任务很快就被重新执行了。整个运算在933秒内完成，包括了初始启动时间（只比正常执行多消耗了5%的时间）。

6、经验

我们在2003年1月完成了第一个版本的MapReduce库，在2003年8月的版本有了显著的增强，这包括了输入数据本地优化、worker机器之间的动态负载均衡等等。从那以后，我们惊喜的发现，MapReduce库能广泛应用于我们日常工作中遇到的各类问题。它现在在Google内部各个领域得到广泛应用，包括：

- 大规模机器学习问题
- Google News和Froogle产品的集群问题
- 从公众查询产品（比如Google的Zeitgeist）的报告中抽取数据。
- 从大量的新应用和新产品的网页中提取有用信息（比如，从大量的位置搜索网页中抽取地理位置信息）。
- 大规模的图形计算。

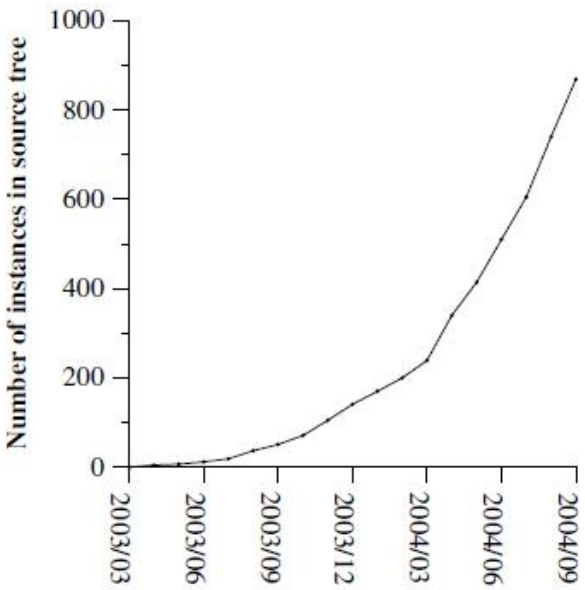


Figure 4: MapReduce instances over time

图四显示了在我们的源代码管理系统中，随着时间推移，独立的MapReduce程序数量的显著增加。从2003年早些时候的0个增长到2004年9月份的差不多900个不同的程序。MapReduce的成功取决于采用MapReduce库能够在不到半个小时时间内写出一个简单的程序，这个简单的程序能够在上千台机器的组成的集群上做大规模并发处理，这极大的加快了开发和原形设计的周期。另外，采用MapReduce库，可以让完全没有分布式和/或并行系统开发经验的程序员很容易的利用大量的资源，开发出分布式和/或并行处理的应用。

任务数	29423
平均任务完成时间	634 秒
使用的机器时间	79,186 天
读取的输入数据	3,288TB
产生的中间数据	758TB
写出的输出数据	193TB
每个 job 平均 worker 机器数	157
每个 job 平均死掉 work 数	1.2
每个 job 平均 map 任务	3,351
每个 job 平均 reduce 任务	55
map 唯一实现	395
reduce 的唯一实现	296
map/reduce 的 combiner 实现	426

表 1：MapReduce2004 年 8 月的执行情况

在每个任务结束的时候，MapReduce库统计计算资源的使用状况。在表1，我们列出了2004年8月份MapReduce运行的任务所占用的相关资源。

6.1、大规模索引

到目前为止，MapReduce最成功的应用就是重写了Google网络搜索服务所使用到的index系统。索引系统的输入数据是网络爬虫抓取回来的海量的文档，这些文档数据都保存在GFS文件系统里。这些文档原始内容*（alex注：raw contents，我认为就是网页中的剔除html标记后的内容、pdf和word等有格式文档中提取的文本内容等）*的大小超过了20TB。索引程序是通过一系列的MapReduce操作（大约5到10次）来建立索引。使用MapReduce（替换上一个特别设计的、分布式处理的索引程序）带来这些好处：

- 实现索引部分的代码简单、小巧、容易理解，因为对于容错、分布式以及并行计算的处理都是MapReduce库提供的。比如，使用MapReduce库，计算的代码行数从原来的3800行C++代码减少到大概700行代码。
- MapReduce库的性能已经足够好了，因此我们可以把在概念上不相关的计算步骤分开处理，而不是混在一起以期减少数据传递的额外消耗。概念上不相关的计算步骤的隔离也使得我们可以很容易改变索引处理方式。比如，对之前的索引系统的一个小更改可能要耗费好几个月的时间，但是在使用MapReduce的新系统上，这样的更改只需要花几天时间就可以了。
- 索引系统的操作管理更容易了。因为由机器失效、机器处理速度缓慢、以及网络的瞬间阻塞等引起的绝大部分问题都已经由MapReduce库解决了，不再需要操作人员的介入了。另外，我们可以通过在索引系统集群中增加机器的简单方法提高整体处理性能。

7、相关工作

很多系统都提供了严格的编程模式，并且通过对编程的严格限制来实现并行计算。例如，一个结合函数可以通过把N个元素的数组的前缀在N个处理器上使用并行前缀算法，在log N的时间内计算完[6，9，13]*（alex注：完全没有明白作者在说啥，具体参考相关6、9、13文档）*。MapReduce可以看作是我们结合在真实环境下处理海量数据的经验，对这些经典模型进行简化和萃取的成果。更加值得骄傲的是，我们还实现了基于上千台处理器的集群的容错处理。相比而言，大部分并发处理系统都只在小规模的集群上实现，并且把容错处理交给了程序员。

Bulk Synchronous Programming[17]和一些MPI原语[11]提供了更高级别的并行处理抽象，可以更容易写出并行处理的程序。MapReduce和这些系统的关键不同之处在于，MapReduce利用限制性编程模式实现了用户程序的自动并发处理，并且提供了透明的容错处理。

我们数据本地优化策略的灵感来源于active disks[12,15]等技术，在active disks中，计算任务是尽量推送到数据存储的节点处理*（alex注：即靠近数据源处理），*这样就减少了网络和IO子系统的吞吐量。我们在挂载几个硬盘的普通机器上执行我们的运算，而不是在磁盘处理器上执行我们的工作，但是达到的目的一样的。

我们的备用任务机制和Charlotte System[3]提出的eager调度机制比较类似。Eager调度机制的一个缺

点是如果一个任务反复失效，那么整个计算就不能完成。我们通过忽略引起故障的记录的方式在某种程度上解决了这个问题。

MapReduce的实现依赖于一个内部的集群管理系统，这个集群管理系统负责在一个超大的、共享机器的集群上分布和运行用户任务。虽然这个不是本论文的重点，但是有必要提一下，这个集群管理系统在理念上和其它系统，如Condor[16]是一样。

MapReduce库的排序机制和NOW-Sort[1]的操作上很类似。读取输入源的机器（map workers）把待排序的数据进行分区后，发送到R个Reduce worker中的一个进行处理。每个Reduce worker在本地对数据进行排序（尽可能在内存中排序）。当然，NOW-Sort没有给用户自定义的Map和Reduce函数的机会，因此不具备MapReduce库广泛的实用性。

River[2]提供了一个编程模型：处理进程通过分布式队列传送数据的方式进行互相通讯。和MapReduce类似，River系统尝试在不对等的硬件环境下，或者在系统颠簸的情况下也能提供近似平均的性能。River是通过精心调度硬盘和网络的通讯来平衡任务的完成时间。MapReduce库采用了其它的方法。通过对编程模型进行限制，MapReduce框架把问题分解成为大量的“小”任务。这些任务在可用的worker集群上动态的调度，这样快速的worker就可以执行更多的任务。通过对编程模型进行限制，我们可用在工作接近完成的时候调度备用任务，缩短在硬件配置不均衡的情况下缩小整个操作完成的时间（比如有的机器性能差、或者机器被某些操作阻塞了）。

BAD-FS[5]采用了和MapReduce完全不同的编程模式，它是面向广域网*（alex注：wide-area network）*的。不过，这两个系统有两个基础功能很类似。（1）两个系统采用重新执行的方式来防止由于失效导致的数据丢失。（2）两个都使用数据本地化调度策略，减少网络通讯的数据量。

TACC[7]是一个用于简化构造高可用性网络服务的系统。和MapReduce一样，它也依靠重新执行机制来实现的容错处理。

8、结束语

MapReduce编程模型在Google内部成功应用于多个领域。我们把这种成功归结为几个方面：首先，由于MapReduce封装了并行处理、容错处理、数据本地化优化、负载均衡等等技术难点的细节，这使得MapReduce库易于使用。即便对于完全没有并行或者分布式系统开发经验的程序员而言；其次，大量不同类型的问题都可以通过MapReduce简单的解决。比如，MapReduce用于生成Google的网络搜索服务所需要的数据、用来排序、用来数据挖掘、用于机器学习，以及很多其它的系统；第三，我们实现了一个在数千台计算机组成的大型集群上灵活部署运行的MapReduce。这个实现使得有效利用这些丰富的计算资源变得非常简单，因此也适合用来解决Google遇到的其他很多需要大量计算的问题。

我们也从MapReduce开发过程中学到了不少东西。首先，约束编程模式使得并行和分布式计算非常容易，也易于构造容错的计算环境；其次，网络带宽是稀有资源。大量的系统优化是针对减少网络传输量为目的的：本地优化策略使大量的数据从本地磁盘读取，中间文件写入本地磁盘、并且只写一份中间文件也节约了网络带宽；第三，多次执行相同的任务可以减少性能缓慢的机器带来的负面影响*（alex注：即硬件配置的不平衡），*同时解决了由于机器失效导致的数据丢失问题。

9、感谢

*（alex注：还是原汁原味的感谢词比较好，这个就不翻译了）*Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people’s suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke, David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg,

Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper.The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

10、参考资料

[1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau,David E. Culler, Joseph M. Hellerstein, and David A. Patterson.High-performance sorting on networks of workstations.In Proceedings of the 1997 ACM SIGMOD InternationalConference on Management of Data, Tucson,Arizona, May 1997.
[2] Remzi H. Arpaci-Dusseau, Eric Anderson, NoahTreuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River:Making the fast case common. In Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99), pages 10.22, Atlanta, Georgia, May 1999.
[3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, 1996. [4] Luiz A. Barroso, Jeffrey Dean, and Urs H¨olzle. Web search for a planet: The Google cluster architecture. IEEE Micro, 23(2):22.28, April 2003.
[5] John Bent, Douglas Thain, Andrea C.Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI, March 2004.
[6] Guy E. Blelloch. Scans as primitive parallel operations.IEEE Transactions on Computers, C-38(11), November 1989.
[7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In Proceedings of the 16th ACM Symposium on Operating System Principles, pages 78. 91, Saint-Malo, France, 1997.
[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In 19th Symposium on Operating Systems Principles, pages 29.43, Lake George, New York, 2003. To appear in OSDI 2004 12
[9] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, Euro-Par'96. Parallel Processing, Lecture Notes in Computer Science 1124, pages 401.408. Springer-Verlag, 1996.
[10] Jim Gray. Sort benchmark home page. http://research.microsoft.com /barc/SortBenchmark/.
[11] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1999.
[12] L. Huston, R. Sukthankar, R.Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference, April 2004.
[13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. Journal of the ACM, 27(4):831.838, 1980.
[14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. Journal of the ACM, 36(2):335.348, 1989.
[15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. IEEE Computer, pages 68.74, June 2001.
[16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. Concurrency and Computation: Practice and Experience, 2004.
[17] L. G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103.111, 1997.
[18] Jim Wyllie. Spsort: How to sort a terabyte quickly. http://alme1.almaden.ibm.com

/cs/spsort.pdf.

附录A、单词频率统计

本节包含了一个完整的程序，用于统计在一组命令行指定的输入文件中，每一个不同的单词出现频率。

```
#include “mapreduce/mapreduce.h”

// User’s map function
class WordCounter : public Mapper {
 public:
  virtual void Map(const MapInput& input) {
   const string& text = input.value();
   const int n = text.size();
   for (int i = 0; i < n; ) {
    // Skip past leading whitespace
    while ((i < n) && isspace(text[i]))
     i++;

   // Find word end
   int start = i;
   while ((i < n) && !isspace(text[i]))
    i++;
   if (start < i)
    Emit(text.substr(start,i-start),”1″);
  }
 }
};

REGISTER_MAPPER(WordCounter);

// User’s reduce function
class Adder : public Reducer {
 virtual void Reduce(ReduceInput* input) {
  // Iterate over all entries with the
  // same key and add the values
  int64 value = 0;
  while (!input->done()) {
   value += StringToInt(input->value());
   input->NextValue();
  }

  // Emit sum for input->key()
  Emit(IntToString(value));
 }
};

REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
 ParseCommandLineFlags(argc, argv);

 MapReduceSpecification spec;

 // Store list of input files into “spec”
 for (int i = 1; i < argc; i++) {
```

```
 MapReduceInput* input = spec.add_input();
 input->set_format("text");
 input->set_filepattern(argv[i]);
 input->set_mapper_class("WordCounter");
}

// Specify the output files:
// /gfs/test/freq-00000-of-00100
// /gfs/test/freq-00001-of-00100
// ...
MapReduceOutput* out = spec.output();
out->set_filebase("/gfs/test/freq");
out->set_num_tasks(100);
out->set_format("text");
out->set_reducer_class("Adder");

// Optional: do partial sums within map
// tasks to save network bandwidth
out->set_combiner_class("Adder");

// Tuning parameters: use at most 2000
// machines and 100 MB of memory per task
spec.set_machines(2000);
spec.set_map_megabytes(100);
spec.set_reduce_megabytes(100);

// Now run it
MapReduceResult result;
if (!MapReduce(spec, &result)) abort();

// Done: 'result' structure contains info
// about counters, time taken, number of
// machines used, etc.
 return 0;
}
```

分类: Google论文 标签:

About alex

2010年3月26日 blademaster 没有评论

热衷于技术，特别是分布式技术、Unix操作系统、C/C++/Perl/Python，最近有迷上了迷一般的Erlang

分类: About 标签:
订阅

Categories

- About
- Google论文
- 小道消息
- 未分类

Blogroll

Archives

- 2010年03月

Meta

- 注册
- 登录