BigTable论文解读

0. 摘要

BigTable是一个分布式存储系统，他可以支持PB级别的数据，包括几千个商业服务器。Google的许多项目都存储在BigTable上，包括WEB索引、Google Earth 和Google Finance。这些应用对BigTable提出了截然不同的需求，无论是从数据量（从URL到网页到卫星图像）而言，还是从延迟需求（从后端批量处理到实时数据服务）而言。尽管这些不同的需求，BigTable已经为所有的Google产品提供了一个灵活的、高性能的解决方案。本文中，我们描述了BigTable提供的简单数据模型，它允许客户端对数据部署和格式进行动态控制，我们描述了BigTable的设计和实施。

1. 前言

在过去的两年半时间里，我们已经设计、实施和部署了一个分布式存储系统BigTable，来管理Google当中的结构化数据。BigTable被设计成可以扩展到PB的数据和上千个机器。BigTable已经达到了几个目标：广泛应用性、可扩展性、高性能和高可用性。Google的六十多款产品和项目都存储在BigTable中，包括Google Analytics和Google Finance，Orkut，Personalized Search，Writely和Google Earth。这些产品使用BigTable来处理不同类型的工作负载，包括面向吞吐量的批处理作业以及对延迟

敏感的终端用户数据服务。这些产品所使用的BigTable的簇，涵盖了多种配置，从几个到几千个服务器，并且存储了几百TB的数据。

在许多方面，BigTable都和数据库很相似，它具有和数据库相同的实施策略。并行数据库[14]和内存数据库[13]已经取得了可扩展性和高性能，但是BigTable提供了和这些系统不一样的接口。BigTable不能支持完整的关系型数据模型，相反，它为客户提供了一个简单数据模型，该数据模型可以支持针对数据部署和格式的动态控制，并且可以允许用户去推理底层存储所展现的数据的位置属性(比如具有相同前缀key的数据位置很接近，读取时候可进行一定的预取来进行优化)。BigTable使用行和列名称对数据进行索引，这些名称可以是任意字符串。BigTable把数据视为未经解释的字符串，客户可能经常把不同格式的结构化数据和非结构化数据都序列化成字符串。最后，BigTable模式参数允许用户动态地控制，是从磁盘获得数据还是从内存获得数据。

2. 数据模型

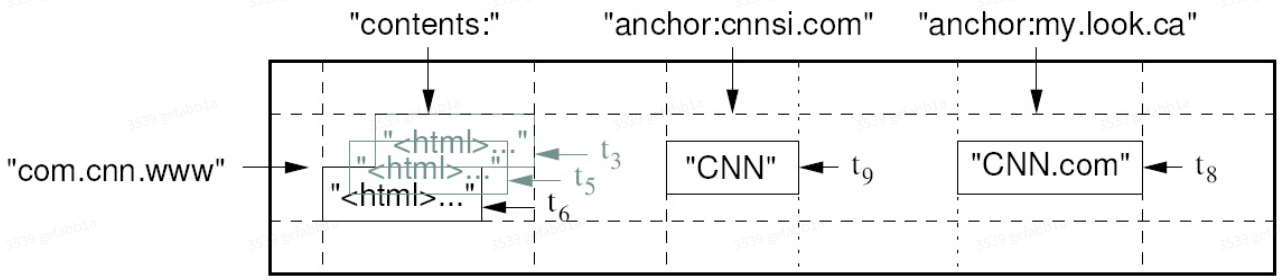
BigTable是一个稀疏的、分布式的、持久化存储的多维排序Map。Map的索引是行关键字、列关键字以及时间戳；Map中的每个value都是一个未经解析的byte数组。

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

2.1 使用例子来介绍BigTable的用途：

背景：

假设我们想要拷贝一个可能被很多项目都使用的、很大的网页集合以及相关的信息，让我们把这个特定的表称为Webtable。在Webtable当中，我们使用URL作为行键，网页的不同方面作为列键，并把网页的内容存储在contents:column中，如图1所示




```Markdown
*存储了网页数据的Webtable的一个片段。行名称是反转的URL，contents列家族包含了网页内容，anchor列家族包含了任何引用这个页面的anchor文本。CNN的主页被Sports Illustrated和MY-look主页同时引用，因此，我们的行包含了名称为"anchor:cnnsi.com"和"anchor:my.look.ca"的列。每个anchor单元格都只有一个版本，contents列有三个版本，分别对应于时间戳t3,t5和t6。*
```

2.2 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。

2.3 列

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。

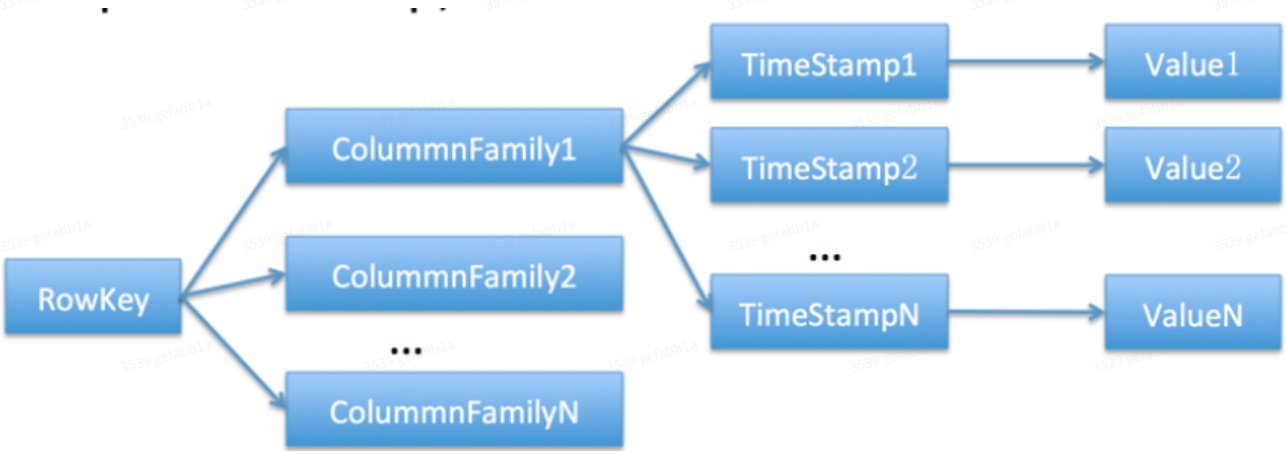
访问控制、磁盘和内存的使用统计都是在列族层面进行的。在我们的Webtable的例子中，上述的控制权限能帮助我们管理不同类型的应用：我们允许一些应用可以添加新的基本数据、一些应用可以读取基本数据并创建继承的列族、一些应用则只允许浏览数据（甚至可能因为隐私的原因不能浏览所有数据）。

2.3 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。



举个例子：

```
CSS
```

“contents”列下保存了网页的三个版本，我们可以用 (“com.cnn.www”, “contents:”, t5) 来找到CNN主页在t5时刻的内容。

3. 接口

Bigtable API提供了创建和删除表和列族的功能。它还提供用于修改群集，表和列族元数据的功能，比如如访问控制权限。

3.1 修改或删除

```
C++
```

```
// 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);
```

3.2 写入

PHP

```
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());
 }
```

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允许把数据项用作整数计数器。

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. 组件

4.1 GFS

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。

4.2 chubby

BigTable依赖一个高可用的、持久性的分布式锁服务Chubby[8]。一个Chubby服务包含5个动态副本，其中一个被选作主副本对外提供服务。当大部分副本处于运行状态并且能够彼此通信时，这个服务就是可用的。Chubby使用Paxos算法[9][23]来使它的副本在失败时保持一致性。

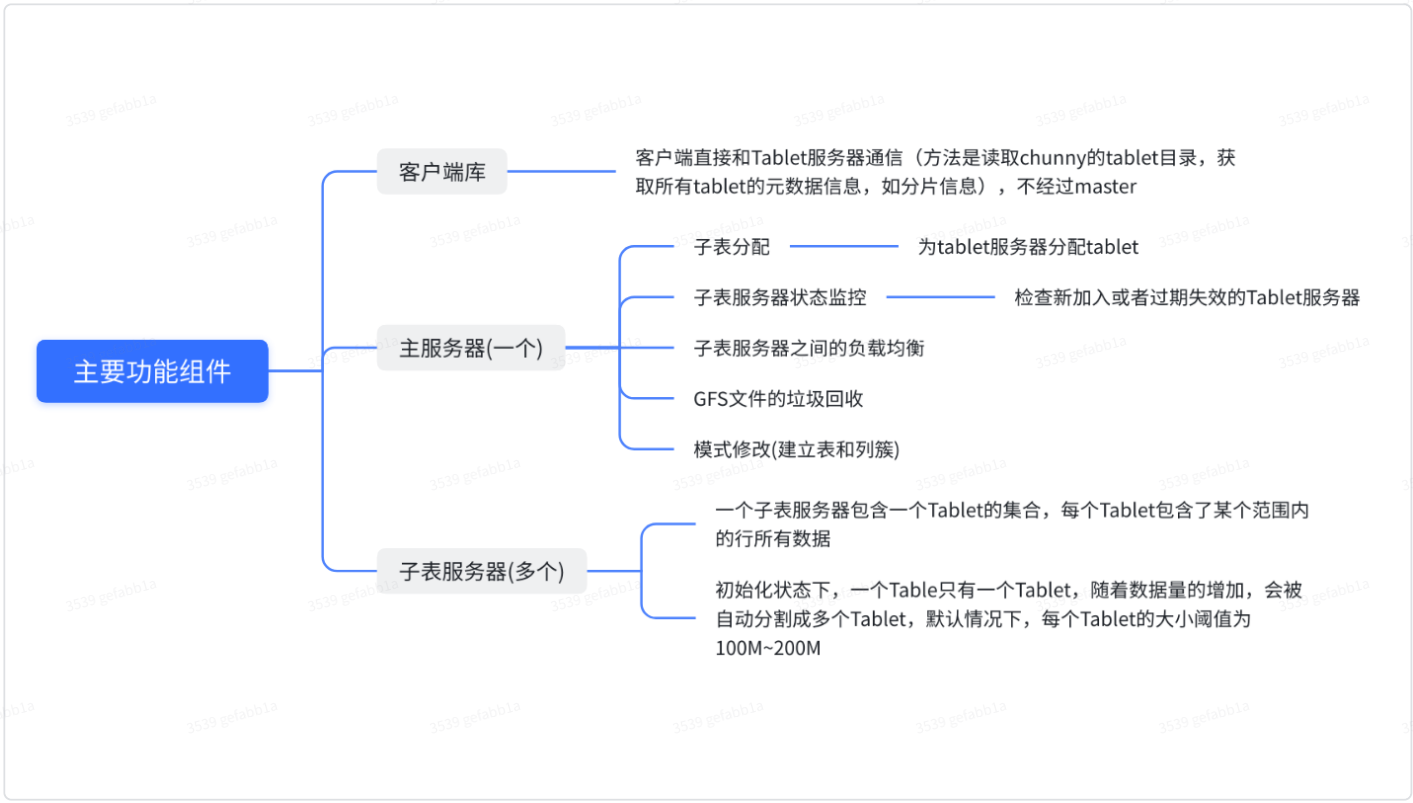
Chubby提供了一个名字空间，它包含了目录和小文件。每个目录和文件可以被用作一个锁，针对文件的读和写操作都是原子的。Chubby客户端函数库提供了针对Chubby文件的持久性缓存。每个Chubby客户端维护一个session，这个session具备Chubby服务。如果租约过期以后不能及时更新session的租约，那么这个客户端的session就会过期。当一个客户端的session过期时，它会丢失所有锁，并且放弃句柄。Chubby客户端也可以注册针对Chubby文件和目录的回调服务（callback），从而通知session过期或其他变化。

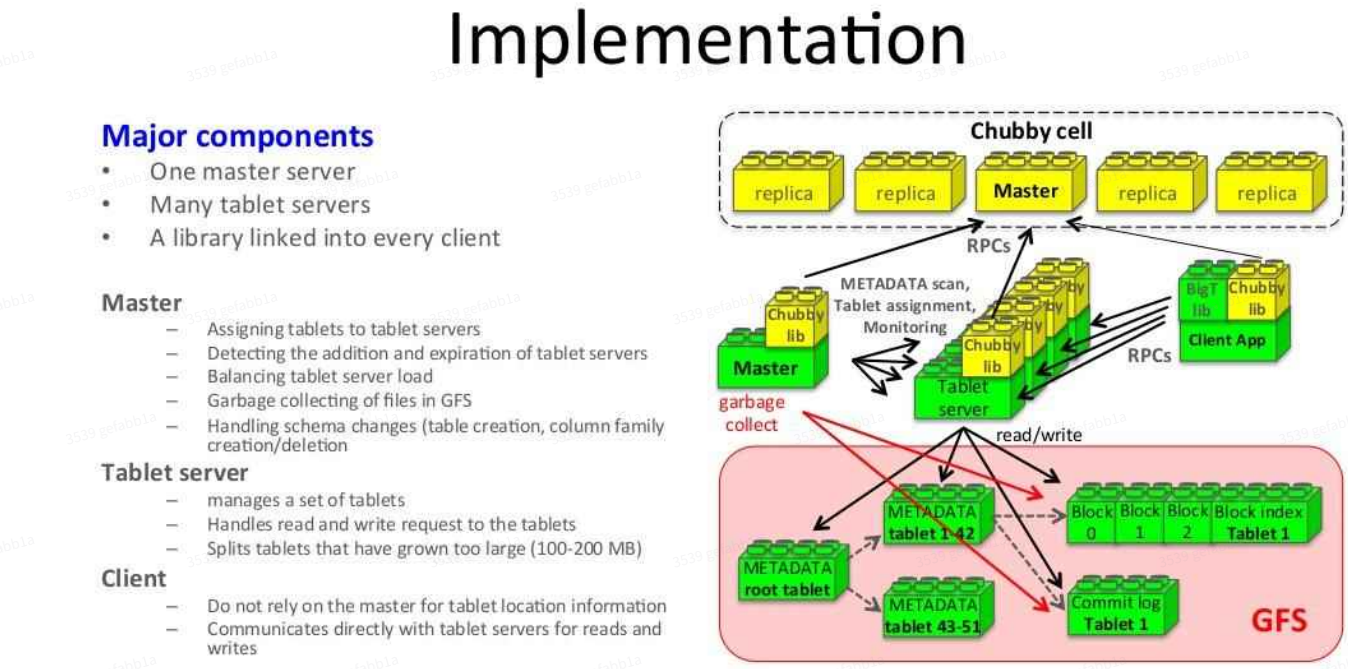
BigTable使用Chubby来完成许多任务：

- 保证在每个时间点只有一个主副本是活跃的
- 来存储BigTable数据的自引导的位置（见5.1节）
- 来发现tablet服务器,包括tablet服务器加入和下线
- 存储BigTable模式信息（即每个表的列家族信息）
- 存储访问控制列表。

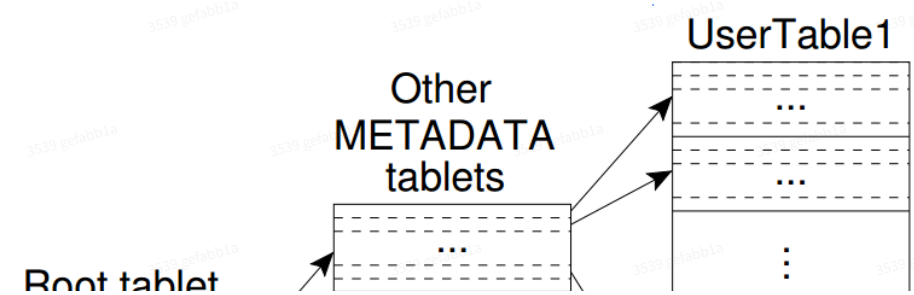
如果在一段时间以后，Chubby变得不可用，BigTable就不可用了。我们最近对涵盖11个Chubby实例的14个BigTable簇进行了这方面的效果测试。由于Chubby的不可用（可能由于Chubby过时，或者网络故障），而导致一些存储在BigTable中的数据变得不可用，这种情形占到BigTable服务小时的平均比例值是0.0047%。单个簇的百分比是0.0326%。

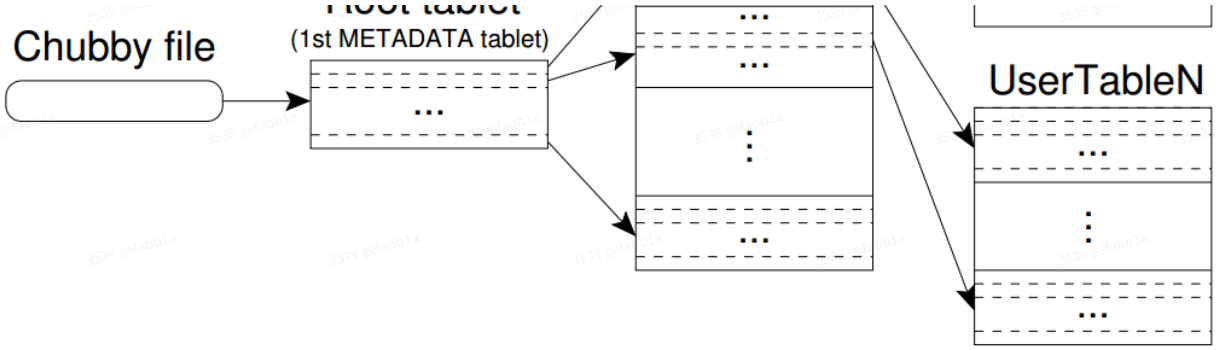
5. 实现


主要功能组件
客户端库
客户端直接和Tablet服务器通信（方法是读取chunny的tablet目录，获取所有tablet的元数据信息，如分片信息），不经过master
主服务器(一个)
子表分配
为tablet服务器分配tablet
子表服务器状态监控
检查新加入或者过期失效的Tablet服务器
子表服务器之间的负载均衡
GFS文件的垃圾回收
模式修改(建立表和列簇)
子表服务器(多个)
一个子表服务器包含一个Tablet的集合，每个Tablet包含了某个范围内的行所有数据
初始化状态下，一个Table只有一个Tablet，随着数据量的增加，会被自动分割成多个Tablet，默认情况下，每个Tablet的大小阈值为100M~200M


Implementation
Major components
One master server
Many tablet servers
A library linked into every client
Master
Assigning tablets to tablet servers
Detecting the addition and expiration of tablet servers
Balancing tablet server load
Garbage collecting of files in GFS
Handling schema changes (table creation, column family creation/deletion
Tablet server
manages a set of tablets
Handles read and write request to the tablets
Splits tablets that have grown too large (100-200 MB)
Client
Do not rely on the master for tablet location information
Communicates directly with tablet servers for reads and writes
Chubby cell
replica
replica
Master
replica
replica
RPCs
METADATA scan,
Tablet assignment,
Monitoring
Chubby lib
Master
Tablet server
BigT lib
Client App
RPCs
garbage collect
read/write
METADATA tablet 1-42
Block 0
Block 1
Block 2
Block index
Tablet 1
METADATA root tablet
METADATA tablet 43-51
Commit log Tablet 1
GFS

5.1 Tablet实现


UserTable1
Other
METADATA
tablets
Root tablet



第一层：一个Chubby文件，该文件存储了*root tablet*的位置信息，由于该文件是Chubby文件，也就意味着，一旦Chubby服务不可用，整个BigTable就丢失了*root tablet*的位置，整个服务也就不可用了。

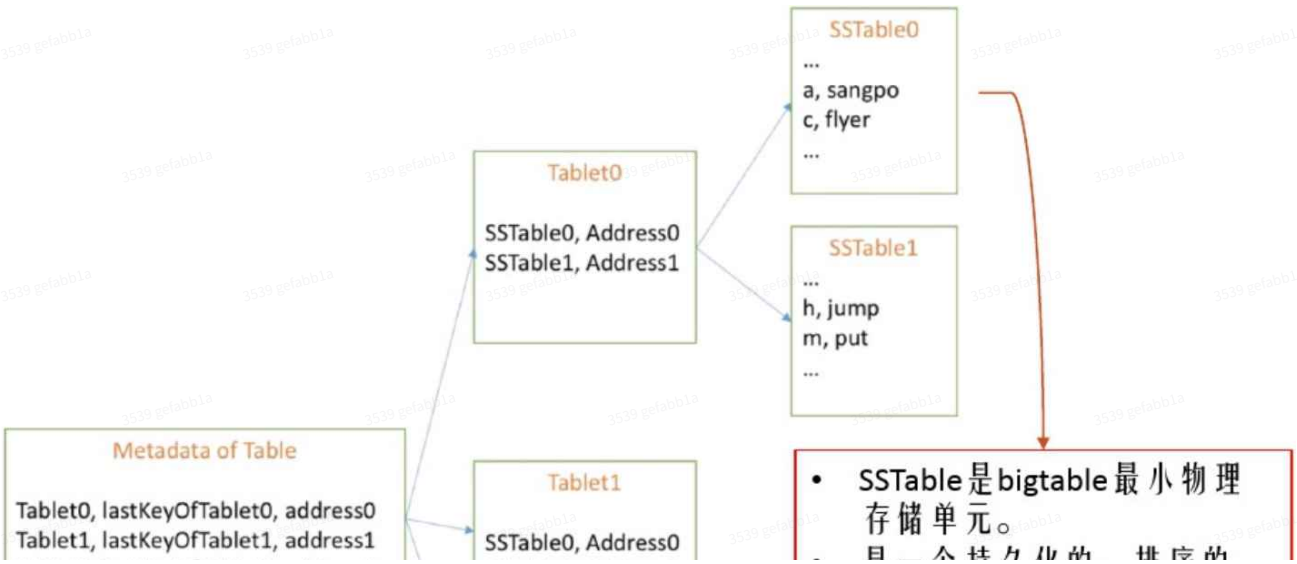
第二层：*root tablet*，*root tablet*其实就是元数据表 `METADATA Table` 的第一个Tablet，该Tablet中保存着元数据表其他Tablet的位置信息，*root tablet*很特殊，为了保证整个树的深度不变，*root tablet*从不分裂。

注意：对于元数据表 `METADATA Table` 来说，除了第一个特殊的Tablet来说，其余每个Tablet包含一组用户Tablet位置信息集合。

注意：`METADATA Table` 存储Tablet位置信息时，`Row Key` 是通过对 `Tablet Table Identifier` 和该Tablet的 `End Row` 生成的。

注意：每个 `METADATA Table` 的 `Row Key` 大约占用1KB的内存，一般情况下，配置 `METADATA Table` 的大小限制为128MB，也就是说，三层的定位模式大约可以寻址2^34个Tablets。

第三层：其他元数据表的Tablet，这些Tablet与*root tablet*共同构成整个元数据表。



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。

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。

1. 当一个Tablet服务器启动流程：

在Chubby的一个指定目录下建立一个唯一的文件，并且获取该文件的独占锁。

2. Master如何管理Tablet服务器

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。当Tablet服务器终止时（比如，集群的管理系统将该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。

3. Tablet服务器如何工作?

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。此时Master服务器将会把Tablet迁移。

4. Master如何管理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(给其它Tablet服务器)。

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。

5. Master服务器启动过程

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集合并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

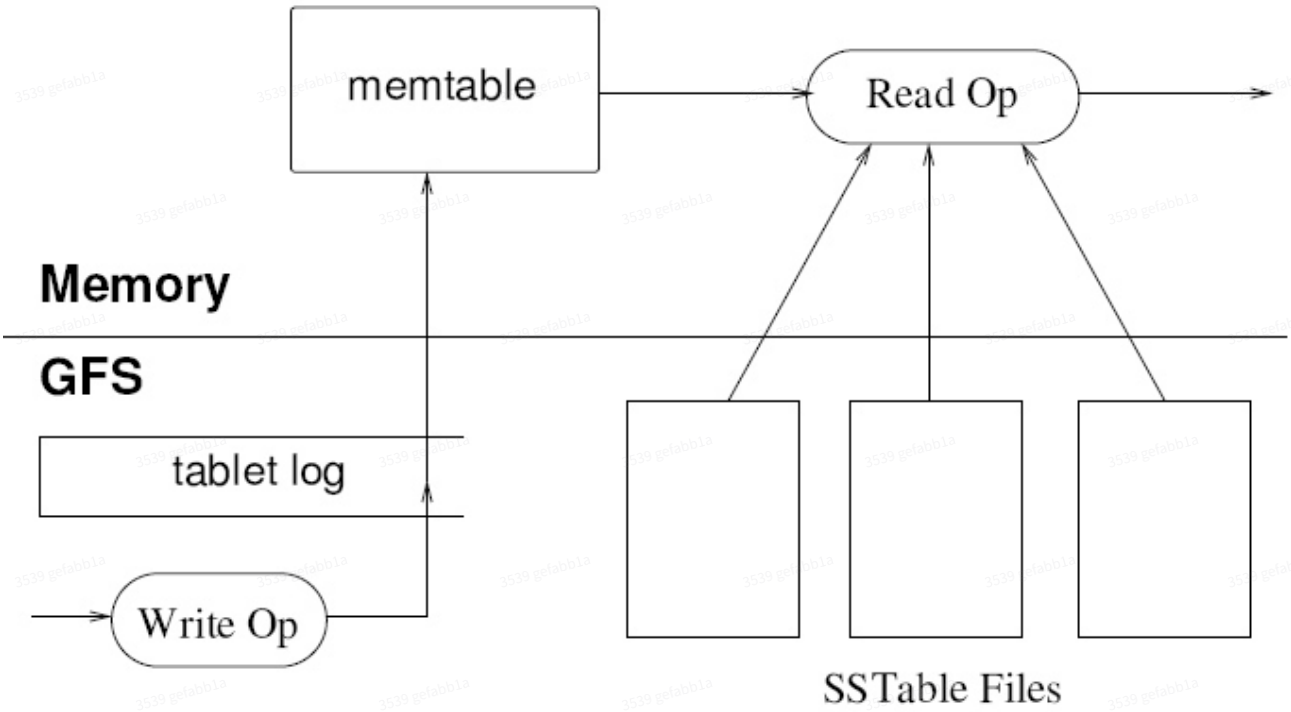
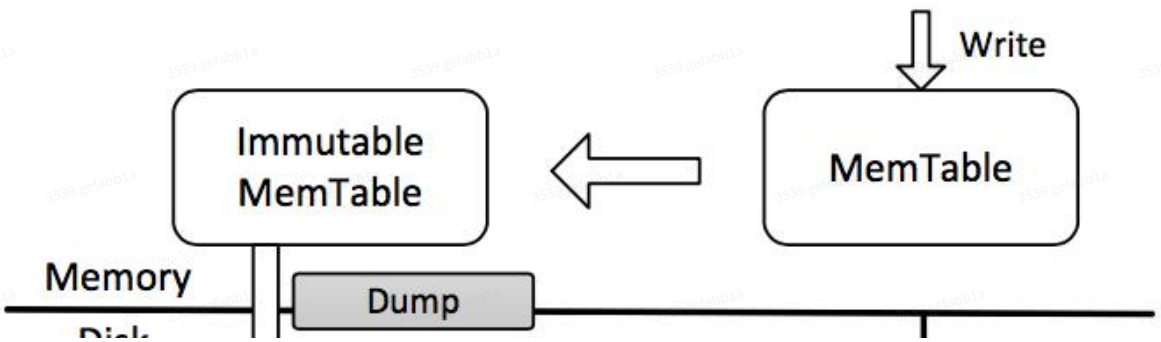
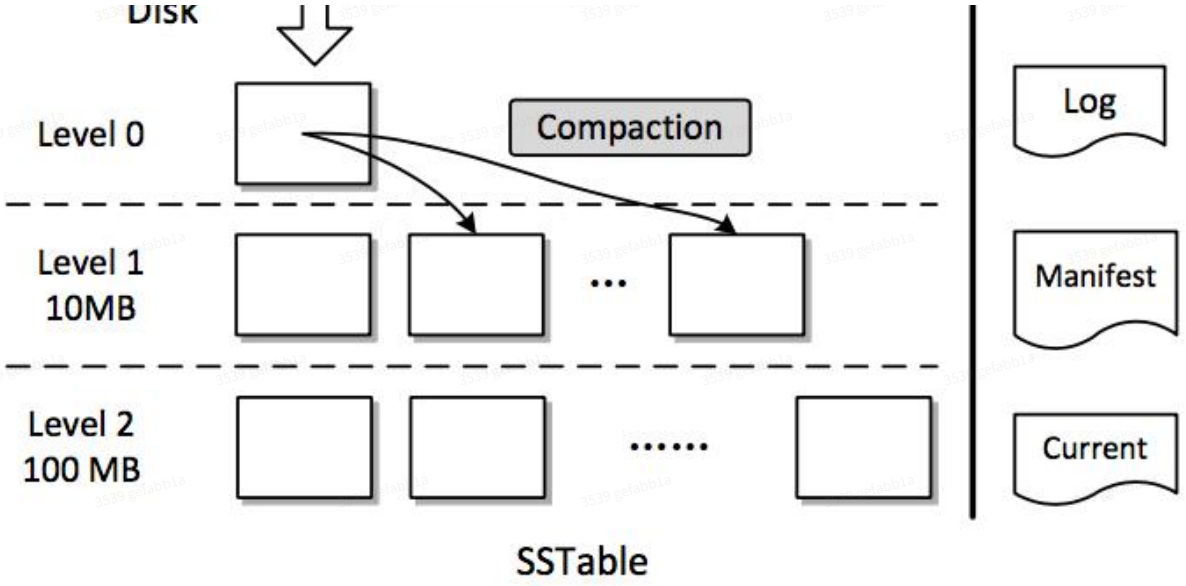


Figure 5: Tablet Representation

对比bigtable的单机版本——leveldb





wiscKey论文精读

一个tablet的持久化状态是存储在GFS中的，如图5所示。更新被提交到一个提交日志，日志中记录了redo记录。在这些更新当中，最近提交的更新被存放到内存当中的一个被称为memtable的排序缓冲区，比较老的更新被存储在一系列SSTable中。为了恢复一个tablet，tablet服务器从METADATA表当中读取这个tablet的元数据。这个元数据包含了SSTable列表，其中每个SSTable都包括一个tablet和一个重做点（redo point）的集合，这些redo point是一些指针，它们指向那些可能包含tablet所需数据的重做日志。服务器把SSTable索引读入内存，执行重做点以后的所有已经提交的更新来重新构建memtable。

当一个写操作到达tablet服务器时，服务器首先检查它定义是否良好，并且发送者是否被授权执行该变更。执行授权检查时，会从一个Chubby文件中读取具有访问权限的写入者的列表，这个Chubby文件通常总能够在Chubby客户端缓存中找到。一个有效的变更会被写入提交日志中。分组提交是为了优化许多小变更[13,16]操作的吞吐量。在写操作已经被提交以后，它的内容就会被插入到memtable。

当一个读操作到达Tablet服务器，与写操作类似，服务器也会首先检查它是否是良好定义和得到授权的。一个有效地读操作是在一系列SSTable和memtable的合并的视图上执行的。由于SSTable和memtable是字典排序的数据结构，视图合并的执行效率很高。

当tablet发生合并或分割操作时，正在到达的读写操作仍然可以继续进行。

5.4 压缩

wiscKey论文精读

随着写操作的执行，memtable的大小逐渐增加。当memtable的大小到达一个阈值的时候，memtable就会被冻结然后创建一个新的memtable，被冻结的memtable就转化成一个SSTable并被写入到GFS。这个*小压缩（minor compaction）*过程有两个目标：它缩小了tablet服务器的内存使用率；当发生服务器宕机需要恢复时可以减少了需要从重做日志中读取的数据量。当压缩过程正在进行时，正在到达的读写操作仍然可以继续进行。

每一次小压缩都会创建一个新的SSTable，如果这种行为没有约束地持续进行，读操作可能需要从任意数量的SSTable中合并更新。相反，我们会对这种文件的数量进行限制，我们在后台周期性地运行一个合并压缩程序。一个合并压缩程序从一些SSTable和memtable中读取内容，并且写出一个新的SSTable。一旦压缩过程完成，这个输入的SSTable和memtable就可以被删除。

一个合并压缩程序，把所有的SSTable的数据重写到一个SSTable，这个合并压缩被称为“主压缩”（major compaction）。非主压缩所产生的SSTable可以包含特殊的删除入口（entry），它把被删除的数据压缩在仍然存活的比较老的SSTable当中。另一方面，一个主压缩过程，产生一个不包含删除信息或被删除的数据的SSTable。BigTable定期检查它的所有tablet，并执行主压缩操作。这些主压缩过程可以允许BigTable收回被删除数据占用的资源，并且保证被删除数据在一定时间内就可以及时的从系统中消失，这对于一些存储敏感数据的服务来说是非常重要的。

6 优化

6.1 局域性群组

客户端可以把多个列族一起分组到一个*locality group*中。我们会为每个tablet中的每个locality group都生成一个单独的SSTable。把那些通常不会被一起访问的列族分割在不同的locality group，可以实现更高效的读。例如，在WebTable当中网页的元数据（比如语言和校验码），可以被放置到同一个locality group当中，网页的内容可以被放置到另一个locality group当中。这样想要读取页面元数据的应用就不需要去访问所有的页面内容。

另外，可以针对每个locality group来指定一些有用的参数。例如，一个locality group可以设置成存放在内存中。常驻内存的locality group的SSTable采用懒加载的方式加载到tablet服务器的内存中。一旦加载，属于这些locality group的列族，就可以被应用直接访问，而不需要读取磁盘。这个特性对于那些被频繁访问的小量数据来说是非常有用的：我们在内部使用其获取METADATA表的列族位置。

6.2 压缩

客户端可以控制是否对相应于某个locality group的SSTable进行压缩，如果压缩，应该采用什么格式。用户自定义的压缩格式可以被应用到每个SSTable块中（块的大小可以通过locality group指定的参数来进行控制）。虽然对每个块进行单独压缩会损失一些空间，但是我们可以从另一个方面受益，当解压缩时只需要对小部分数据进行解压而不需要解压全部数据。许多客户端都使用两段自定义压缩模式。第一遍使用Bentley and McIlroy[6]模式，它对一个大窗口内的长的公共字符串进行压缩。第二遍使用一个快速的压缩算法，这个压缩算法在一个16KB数据量的窗口内寻找重复数据。两个压缩步骤都非常快，在现代计算机上运行，他们编码的速度是100-200MB/S，解码的速度在400-1000MB/S。

在选择我们的压缩算法时，即使我们强调速度而不是空间压缩率，这个两段压缩模式也表现出了惊人的性能。例如，在WebTable中，我们使用这种压缩模式来存储网页内容。在其中一个实验当中，我们

在一个压缩后的locality group中存储了大量的文档。为了达到实验的目的，我们只为每个文档存储一个版本，而不是存储我们可以获得的所有版本。这个压缩模式获得了10：1的空间压缩率。这比传统的GZip方法的效果要好得多，GZip针对HTML数据通常只能获得3:1到4:1的空间压缩率。**这种性能上的改进是和WebTable中的行的存储方式紧密相关的，即所有来自同一个站点的网页都存储在相近的位置。**这就使得Bentley and McIlroy算法可以从同一个站点的网页中确定大量相似的内容。许多应用，不只是WebTable，都会很好地选择行名称，从而保证相似的数据可以被存放到同一个集群当中，这样就可以取得很好的压缩率。当我们在BigTable中存储同一个值的多个不同版本时，可以取得更好的压缩率。

6.3 读缓存

为了读操作的性能，tablet服务器使用双层缓存。扫描缓存是高层缓存，它缓存了tablet服务器代码使用SSTable接口获取的key-value对；块缓存是底层缓存，它缓存了从GFS上读取的SSTables块。扫描缓存主要用于倾向重复访问相同数据的应用。块缓存主要用于倾向读取近期数据附近数据的应用（如：顺序读取或随机读取同一个局域性群组的一个频繁访问行的不同列）。

6.4 布隆过滤器

如5.3中描述的，一个读操作必须从所有的组成tablet的SSTable中读取数据。如果这些SSTable没有在内存中，则我们最终会多次访问硬盘。我们通过允许客户端对特定局域性群组的SSTable指定Bloom过滤器来降低访问次数。一个Bloom过滤器允许我们查询一个SSTable是否含有特定的行/列对的数据。对于某些特定应用，虽然存储Bloom过滤器占用了tablet服务器少量的内存，但能够彻底的减少读操作对磁盘的查询次数。我们使用Bloom过滤器也可以隐式的达到了当查询的行和列不存在时，不需要访问磁盘。

6.5 Commit日志的实现

1. 每个 tablet 还是每个 tablet server 一个 log 文件

如果为每个 tablet 维护一个单独的 log 文件，那会导致底层 GFS 大量文件的并发写。考 虑到 GFS 的具体实现，这些并发写进而会导致大量的磁盘访问，以完成不同物理文件的并 发写入。另外，每个 tablet 一个 log 文件的设计还会降低组提交（group commit，批量 提交）优化的有效性，因为每个组（group）都会很小。

因此，为了克服以上问题，我们为**每个 tablet server 维护一个 commit log**，将属于 这个 tablet server 的不同的 tablet 操作都写入这同一个物理上的 log 文件 [18, 20]。

2. 恢复起来比较麻烦

这种方式使得常规操作（normal operations）的性能得到了很大提升，但是，它使 tablet 恢复过程变得复杂。

当一个 tablet server 挂掉后，它负责的那些 tablets 就会重新分配给其他（大量）的 tablet servers：通常情况下每个 tablet server 只会分到其中的一小部分。恢复一个 tablet 的状态时，新的 tablet server 需要从原 tablet server 的 commit log 里重新 应用（reapply）这个 tablet 的修改（mutation）。然而，**这些 tablet 的 mutation 都 混在同一个物理的 log 文件内**。

一种方式是每个新的 tablet server 都去读完整的 commit log，将自己需要的部分过滤出 来。但是，如果有 100 个机器分到了 tablet 的话，这个 log 文件就要被读 100 次。

3. 如何解决这个问题呢?

为了避免这种重复读，我们**将 commit log 内容**以 `(table; row name; log sequence number)` 为键（key）**进行排序。在排序后的 commit log 中，每个 tablet 的所有 mutation 都是连续的**，因此可以实现高效的读取：**只需一次磁盘寻址加随后的顺序读**。 为了加速排序过程，我们还将 commit log 分割成 64 MB 的段（segment），分散到多个 tablet server 上并发地进行排序。

这个排序过程是由 **master 协调（coordinate）、tablet server 触发**的： tablet server 向 master 汇报说需要从一些 commit log 中恢复一些 mutation。

写提交记录到 GFS 有时会遇到性能卡顿，这可能有多方面原因。例如，负责写操作的 GFS server 挂了，或者到三个指定的 GFS master 的网络发生了拥塞或过载。为了减少这些 GFS 导致的延迟抖动，**每个 tablet server 为 commit log 使用了两个写线程**：每个 线程写到各自的 log 文件，但同时只会有一个线程是活跃的。 如果当前的活跃线程写性能非常差，写操作就会切换到另一个线程，由这个新线程负责之后 的写。

log 中的记录（entry）都有序列号，恢复的时候可以根据序列号过滤由于 log 切换导致 的重复数据。

6.6 加速 tablet 恢复过程

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。

6.7 利用不可变性

除了 SSTable 缓存之外，Bigtable 系统其他一些部分也因 SSTable 的不可变性而得到简化。例如，从 SSTable 读取数据时，对文件系统的访问不需要任何同步。因此，对行的并发控制可以实现地非常高效。

读和写操作涉及的唯一可变数据结构是 memtable。为减少 memtable 的读竞争，我们 将 memtable 的行（row）设计为**写时复制**（copy-on-write），这样读和写就可以并行进行。

因为 SSTable 是不可变的，所以**彻底删除数据**（permanently removing deleted data ）的问题就变成了**对过期的 SSTable 进行垃圾回收**（garbage collecting obsolete SSTables）。

每个 tablet 的 SSTable 会注册到 `METADATA` table。master 会对过期的 SSTable 进行**“先标记后清除”**（mark-and-sweep）[25]，其中 `METADATA` table 记录了这些 SSTable 的对应的 tablet 的 root。

最后，**SSTable 的不可变性使得 tablet 分裂过程更快**。我们直接让子 tablet 共享父 tablet 的 SSTable ，而不是为每个子 tablet 分别创建一个新的 SSTable。

9 经验

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，也学习到了 很多有趣的教训。

9.1 大的分布式系统会发生各种错误

我们得到的一个教训是，大的分布式系统很发生多种错误，不仅是其他分布式系统经常遇到的标准的网络分割和故障。例如，我们就遇到过如下场景引起的故障：

- 内存和网络损坏
- 很大的时钟偏差（clock skew）
- 机器死机（hung）
- 更复杂的和非对称的网络分裂
- 依赖的基础服务的 bug（例如 Chubby）
- GFS 配额溢出（overflow）
- 计划及非计划的硬件维护

随着对这一问题的了解的深入，我们开始修改各种的协议来应对这一问题。例如，我们给 RPC 机制添加了校验和。另外，我们还去掉了系统的一个部分对另一部分的假设。例如，我们不再假设一次 Chubby 操作只会返回固定的错误。

9.2谨慎添加使用场景不明确的新特性

我们得到的另一重要经验是：如果还不是非常清楚一个新特性将被如何使用，那就不要着急 添加到系统中。

例如，我们最初有计划在 API 中支持广义事物模型（general-purpose transaction）。但 因为当时没有迫切的使用场景，因此没有立即去实现。现在有了很多真实应用跑在 Bigtable 之后，我们审视了这些应用的真实需求，发现大部分应用其实只需要单行事务（single-row transaction）。

对于真的有分布式事务需求的人，我们发现他们最核心的需求其实是维护二级索引（ secondary indices），因此我们计划通过添加一个特殊的机制来满足这个需求。这个机制 没有分布式事务通用，但性能会更好（尤其是跨上百行以上的更新），而且对于乐观跨数据 中心复制（optimistic cross-data-center replication）来说，和我们系统的集成会更好。

系统级监控非常重要

在日常支持 Bigtable 中学到的实际一课是：合理的系统级监控（例如监控 Bigtable 本身 ，以及使用 Bigtable 的客户端）非常重要。

例如，我们扩展了我们的 RPC 系统，可以记录重要动作的详细跟踪信息。这个特性帮助我 们检测和解决了很多问题，包括：

1. tablet 数据结构上的锁竞争

2. 提交 Bigtable mutation 时 GFS 写很慢
3. `METADATA` tablets 不可用时访问 `METADATA` 表时被卡住（stuck）

监控的另一个例子是每个 Bigtable 集群都注册到了 Chubby。这使得我们可以跟踪所有的集 群，看到集群有多大，各自运行的是什么版本，接收到的流量有多大，是否有异常的大延迟 等等。

9.3保持设计的简洁

我们学到的最重要经验是：简单设计带来的价值（the value of simple designs）。

考虑到我们的系统规模（10 万行代码，不包括测试），以及代码都会随着时间以难以 意料的方式演进，我们发现代码和设计的简洁性对代码的维护和 debug 有着巨大的帮助。

> Given both the size of our system (about 100,000 lines of non-test code), as well as the fact that code evolves over time in unexpected ways, we have found that code and design clarity are of immense help in code maintenance and debugging.

一个例子是我们的 tablet server 成员（membership）协议。我们的第一版非常简单： master 定期向 tablet server 提供租约，如果一个 tablet server 的租约到期，它就自 杀。不幸的是，这个协议在发生网络问题时可用性非常差，而且对 master 恢复时间也很敏感。

接下来我们重新设计了好几版这个协议，直到它令我们满意。但是，这时的协议已经变得过 于复杂，而且依赖了一些很少被其他应用使用的 Chubby 特性。最后发现我们花了大量的时 间来 debug 怪异的边界场景，不仅仅是 Bigtable 代码，还包括 Chubby 代码。

最终，我们放弃了这个版本，重新回到了一个新的更简单的协议，只依赖使用广泛的 Chubby 特性。

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34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35【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 总结

我们在 Google 设计了 Bigtable，一个存储**结构化数据**的分布式系统。

Bigtable 从 2005 年 4 月开始用于生产环境，而在此之前，我们花了大约 **7 个人年** （person-year）的时间在设计和实现上。到 2006 年 8 月，已经有超过 60 个项目在使用 Bigtable。

我们的用户很喜欢 Bigtable 提供的性能和高可用性，当集群面临的负载不断增加时，他们只需简单地向集群添加更多的节点就可以扩展 Bigtable 的容量。

考虑到 Bigtable 的接口不是太常规（unusual），一个有趣的问题就是，我们的用户需要 花多长时间去适应 Bigtable。新用户有时不太确定如何使用 Bigtable 最合适，尤其是如 果之前已经习惯了关系型数据库提供的广义事务。然后，很多 Google 产品成功地使用了 Bigtable 还是说明了，我们的设计在实际使用中还是非常不错的。

当前我们正在添加一些新的特性，例如支持 secondary indices，以及构建跨数据中心复制 的、有多个 master 副本的 Bigtable。我们还在做的是将 Bigtable 作为一个服务提供给 各产品组，以后每个组就不需要自己维护他们的集群。随着服务集群的扩展，我们将 需要处理更多 Bigtable 内部的资源共享问题 [3, 5]。

最后，我们发现**构建我们自己的存储解决方案可以带来非常大的优势**。为 Bigtable 设 计自己的数据模型已经给我们带来非常多的便利性。另外，我们**对 Bigtable 的实现，以 及 Bigtable 所依赖的其他 Google 基础设施有足够的控制权**，因此任何一个地方有瓶颈 了，我们都可以及时解决。

12 引用

https://wenku.baidu.com/view/7425b34b00020740be1e650e52ea551810a6c903.html