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

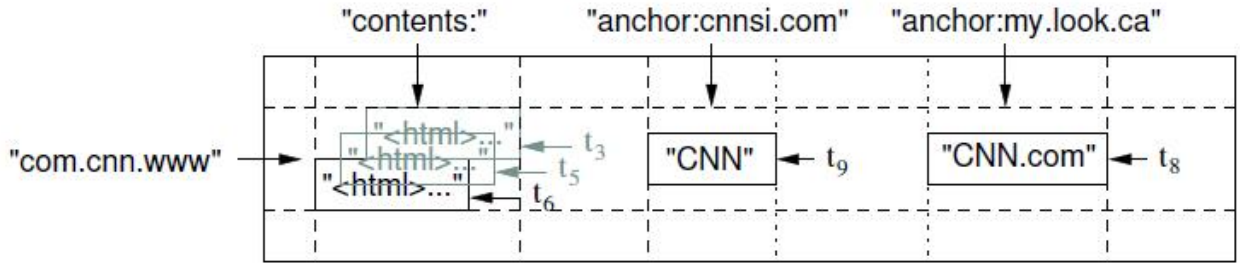
制数据的位置相关性。最后，可以通过BigTable的模式参数来控制数据是存放在内存中、还是硬盘上。第二节描述关于数据模型更多细节方面的东西；第三节概要介绍了客户端API；第四节简要介绍了BigTable底层使用的Google的基础框架；第五节描述了BigTable实现的关键部分；第6节描述了我们为了提高BigTable的性能采用的一些精细的调优方法；第7节提供了BigTable的性能数据；第8节讲述了几个Google内部使用BigTable的例子；第9节是我们在设计和后期支持过程中得到一些经验和教训；最后，在第10节列出我们的相关研究工作，第11节是我们的结论。

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)。

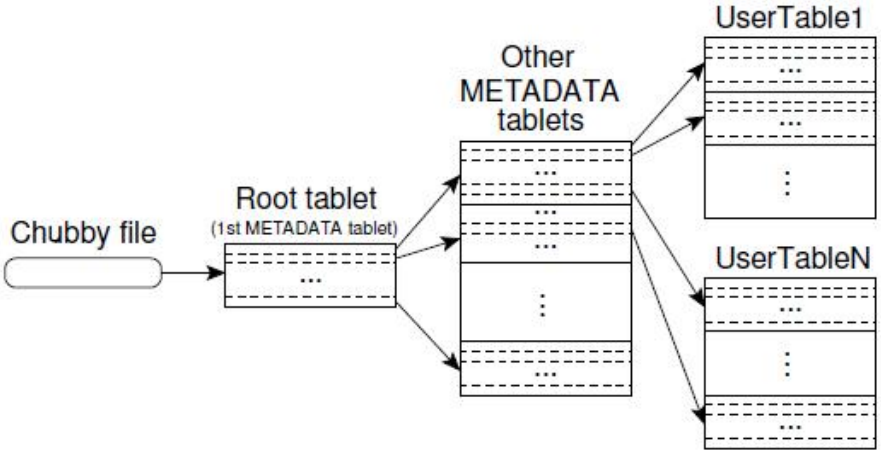


Figure 4: Tablet location hierarchy.

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

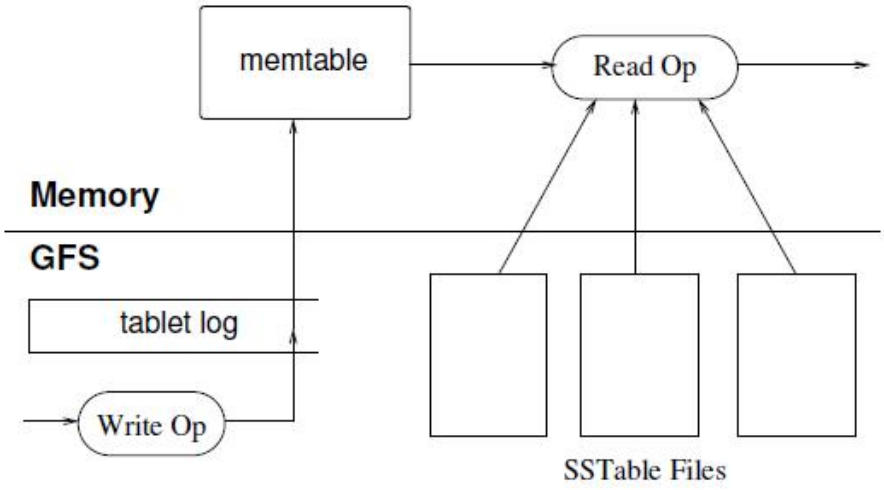


Figure 5: Tablet Representation

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

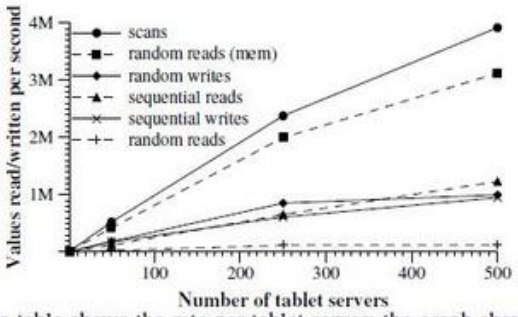


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

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.