Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

本文翻译：厦门大学计算机系 林子雨（http://www.cs.xmu.edu.cn/linziyu）

翻译时间：2010 年 7 月

本文英文论文引用方式：

[ChangDGHWBCFG06]Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert Gruber: Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). OSDI 2006:205-218.

本文英文原始目录：

[本文翻译的原始出处：厦门大学计算机系数据库实验室网站林子雨老师的云数据库技术资料专区 http://dblab.xmu.edu.cn/cloud_database_view]

林子雨老师翻译内容如下：

Abstract

BigTable 是一个分布式存储系统，它可以支持扩展到很大尺寸的数据：PB 级别的数据，包含几千个商业服务器。Google 的许多项目都存储在 BigTable 中，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 Introduction

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。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，虽然，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。

本文第 2 部分详细描述了数据模型，第 3 部分大概介绍了用户 API，第 4 部分简要介绍了 BigTable 所依赖的 Google 底层基础设施，第 5 部分描述了 BigTable 的实施方法，第 6 部分描述了我们针对 BigTable 做的性能改进，第 7 部分提供了 BigTable 的性能衡量方法，第 8 部分给出了几个实例来介绍 Google 如何使用 BigTable，第 9 部分介绍了我们在设计和支持 BigTable 过程中得到的经验教训。最后，在第 10 部分介绍相关工作，第 11 部分给出结论。

2 Data model

一个 BigTable 是一个稀疏的、分布的、永久的多维排序图。我们采用行键盘（row key）、列键（column key）和时间戳（timestamp）对图进行索引。图中的每个值都是未经解释的字节数组。

(row:string, column string, time:int64)→string

我们在检查了类似 BigTable 的系统的多种应用以后，才决定采用这种数据模型。这里给出一个实际的例子来阐释为什么我们采用数据模型设计。假设我们想要拷贝一个可能被很多项目都是用的、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。

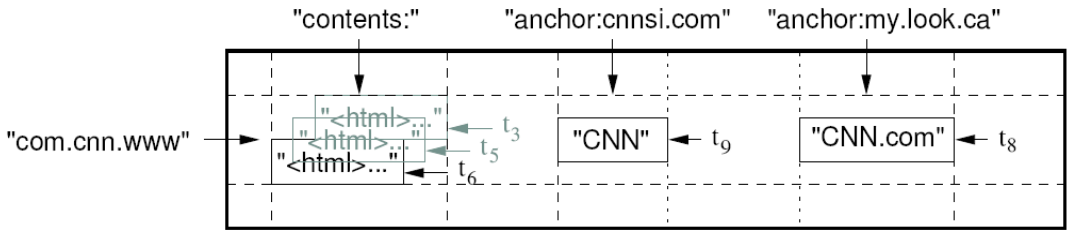

图 1 存储了网页数据的 Webtable 的一个片段。行名称是反转的 URL，contents 列家族包含了网页内容，anchor 列家族包含了任何引用这个页面的 anchor 文本。CNN 的主页被 Sports Illustrated 和 MY-look 主页同时引用，因此，我们的行包含了名称为"anchor:cnnsi.com"和"anchor:my.look.ca"的列。每个 anchor 单元格都只有一个版本，contents 列有三个版本，分别对应于时间戳 t3,t5 和 t6。

ROWS

一个表中的行键，是任意的字符串（当前在尺寸上有 64KB，虽然 10-100 字节是用户最常用的尺寸）。对每个行键下所包含的数据的读或写都是一个原子操作，不管这个行中所包含的列的数量是多少。这种设计决定可以使得当针对同一行发生并发更新行为时，用户很容易知道系统的行为。

BigTable 在行键上根据字典顺序对数据进行维护。对于一个表而言，行区间是动态划分的。每个行区间称为一个 Tablet，它是负载均衡和数据分发的基本单位。因而，读取一个比较短的行区间是非常高效的，通畅只需要和少数几个机器通讯。用户可以利用这种属性，也就是说，用户可以选择分布具有局部性的行区间。例如，在 Webtable 中，通过对 URL 地址进行反转，属于同一个领域的网页都会被分组到连续的行中。例如，我们在键 com.google.maps/index.html 下面存储 com.google.maps/index.html 中包含的数据。把来自同一个领域的数据彼此临近存储，使得一些领域分析更加高效。

Column Families

列键被分组成称为“列家族”的集合，它成为基本的访问控制单元。存储在一个列家族当中的所有数据，通常都属于同一个数据类型（我们对同一个列家族中的数据一起进行压缩）。数据可以被存放到列家族的某个列键下面，但是，在把数据存放到这个列家族的某个列键下面之前，必须首先创建这个列家族。在创建完成一个列家族以后，就可以使用同一个家族当中的列键。我们的意愿是，让一个表当中所包含的列家族的数量尽可能少（至多几百个列家族），而且，在操作过程当中，列家族很少发生变化。相反，一个表可以包含无限数量的列。

列键采用下面的语法命名：family:qualifier。列家族名字必须是可打印的，但是，修饰符 qualifier 可以是任意字符串。比如，对于 Webtable 而言，有一个列家族是 language，它存储了网页所用语言的信息。在 language 列家族中，我们只使用一个列键，它存储了每个网页语言的 ID。Webtable 当中另一个有用的列家族就是 anchor，这个列家族中的每个列键都代表了一个单个的 anchor，如图 1 所示。它的修饰符 qualifier 是引用网站的名称，这个单元格内容是链接文本。

访问控制以及磁盘和内存审计是在列家族层面上进行的。以 Webtable 为例，这些控制允许我们管理几种不同类型的应用，一些应用负责增加新的基本数据，一些应用负责读取基

本数据并且创建衍生的列家族，一些应用则只被允许浏览现有的数据（甚至，如果出于隐私保护考虑，无法浏览全部列家族）。

Timestamps

在 BigTable 中的每个单元格当中，都包含相同数据的多个版本，这些版本采用时间戳进行索引。BitTable 时间戳是 64 位整数。BigTable 对时间戳进行分配，时间戳代表了真实时间，以微秒来计算。客户应用也可以直接分配时间戳。需要避免冲突的应用必须生成唯一的时间戳。一个单元格的不同版本是根据时间戳降序的顺序进行存储的，这样，最新的版本可以被最先读取。

为了减轻版本数据的管理负担，我们支持两种“每列家族”设置，它会告诉 BigTable 来自动垃圾收集（garbage-collect）单元格版本。用户可以设定只保存单元格中数据的最近 n 个版本，或者只保存足够新版本（比如只保存最近 7 天内的数据版本）。

在我们的 Webtable 实例当中，我们为存储在 contents:column 中的网页设置时间戳，时间戳的数值就是这个网页的这个版本被抓取的真实时间。上面所描述的垃圾收集机制，允许我们只保留每个网页的最近三个版本。

3 API

BigTable 的 API 提供了删除和创建表和列家族的功能。它还提供了改变簇、表和列家族的元数据，比如访问控制权限。

客户应用可以书写和删除 BigTable 中的值，从单个行中查询值，或者对表中某个数据子集进行遍历。图 2 显示了一段 C++代码，它使用了 RowMutation 来执行一系列的更新（为了更好地理解这个例子，已经忽略了不相关的细节）。对 Apply 的调用，会执行一个针对 Webtable 的原子更新操作：它增加一个 anchor 到 www.cnn.com 中去，并且删除一个不同的 anchor。

```
// 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.

图 3 显示了一段 C++代码，它使用了 Scanner 来遍历某个行中的所有 anchor。客户端可以遍历多个列家族，并且有几种机制可以用来对一次扫描中所产生的行、列和时间戳的数量进行限制。例如，我们可以对上面的扫描进行限制，让所产生的 anchor 所在的列与正则表达式匹配 anchor:*.cnn.com，或者只产生那些时间戳距离当前时间 10 天以内的 anchor。

```
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.

BigTable 支持几种其他的功能，允许用户以更加复杂的方式来操作数据。首先，BigTable 支持单行事务，可以允许对存储在某个行键下面的数据执行原子的“读-修改-写”操作。BigTable 当前不支持通用的跨行键的事务，虽然它在客户端提供了跨行键批量写入数据的接口。其次，BigTable 允许单元格被用来作为整数计数器。最后，BigTable 支持在服务器的地址空间内执行客户端提供的脚本。这种脚本是用称为 Sawzall 的语言开发的，这种语言是 Google 开发出来进行数据处理的。目前，基于 Sawzall 的 API 不允许客户端脚本对 BigTable 执行回写操作，但是，它确实允许不同类型的数据转换、基于任意表达式的过滤以及针对不同类型操作符的总结。

BigTable 可以和 MapReduce[12]一起使用，MapReduce 是 Google 开发的、用来运行大规模并行计算的框架。我们已经书写了一个 Wrapper 集合，它允许 BigTable 被用来作为一个 MapReduce 作业的输入源或者输出目标。

4 Building Blocks

BigTable 是构建在其他几个 Google 基础设施之上的。BigTable 使用了分布式 Google 文件系统（GFS[17]）来存储日志和数据文件。BigTable 的一个簇通常在一个共享机器池内进行操作，这个共享机器池会运行其他一些分布式应用。BigTable 的进程通常和其他应用的进程共享同样的机器。BigTable 依赖一个簇管理系统来调度作业、在共享机器上调度资源、处理机器失败和监督机器状态。

Google SSTable 文件格式作为存储 BigTable 数据的内部格式。一个 SSTable 提供一个持久化的、排序的、不可变的、从键到值的映射，其中，键和值都是任意的字节字符串。BigTable 提供了查询与一个指定键相关的值的操作，以及在一个指定的键区间内遍历所有的“键/值对”的操作。在内部，每个 SSTable 都包含一个块序列。通常，每个块是 64KB，不过块尺寸是可配置的。存储在 SSTable 结尾的块索引，可以用来快速定位块的位置。当 SSTable 被打开时，块索引就会被读入内存。一个查询操作只需要进行一次磁盘扫描，我们首先在内存的块索引当中使用二分查找方法找到合适的块，然后从磁盘中读取相应的块。可选地，一个 SSTable 可以被完全读入内存，这样，我们在进行查找操作时，就不需要读取磁盘。

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：（1）保证在每个时间点只有一个主副本是活跃的，（2）来存储 BigTable 数据的 bootstrap 的位置（见 5.1 节），（3）来发现 tablet 服务器，（4）宣告 tablet 服务器死亡，（5）存储 BigTable 模式信息（即每个表的列家族信息），以及（6）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 Implementation

BigTable 实现包括三个主要的功能组件：(1)库函数：链接到每个客户端，（2）一个主服务器，（3）许多 Tablet 服务器。Tablet 服务器可以根据工作负载的变化，从一个簇中动态地增加或删除。主服务器负责把 Tablet 分配到 Tablet 服务器，探测 Tablet 服务器的增加和过期，进行 Table 服务器的负载均衡，以及 GFS 文件系统中的垃圾收集。除此以外，它还处理模式变化，比如表和列家族创建。

每个 Tablet 服务器管理一个 Tablet 集合，通常，在每个 Tablet 服务器上，我们会放置 10 到 1000 个 Tablet。Tablet 服务器处理针对那些已经加载的 Tablet 而提出的读写请求，并且会对过大的 Tablet 进行划分。

就像许多单服务器分布式存储系统一样[17,21]，客户端并不是直接从主服务器读取数据，而是直接从 Tablet 服务器上读取数据。因为 BigTable 客户端并不依赖于主服务器来获得 Tablet 的位置信息，所以，大多数客户端从来不和主服务器通信。从而使得在实际应用中，主服务器负载很小。

一个 BigTable 簇存储了许多表。每个表都是一个 Tablet 集合，每个 Tablet 包含了位于某个域区间内的所有数据。在最初阶段，每个表只包含一个 Tablet。随着表的增长，它会被自动分解成许多 Tablet，每个 Tablet 默认尺寸大约是 100 到 200MB。

5.1 Tablet Location

我们使用了一个类似于 B+树的三层架构（如图 4 所示），来存储 Tablet 位置信息。

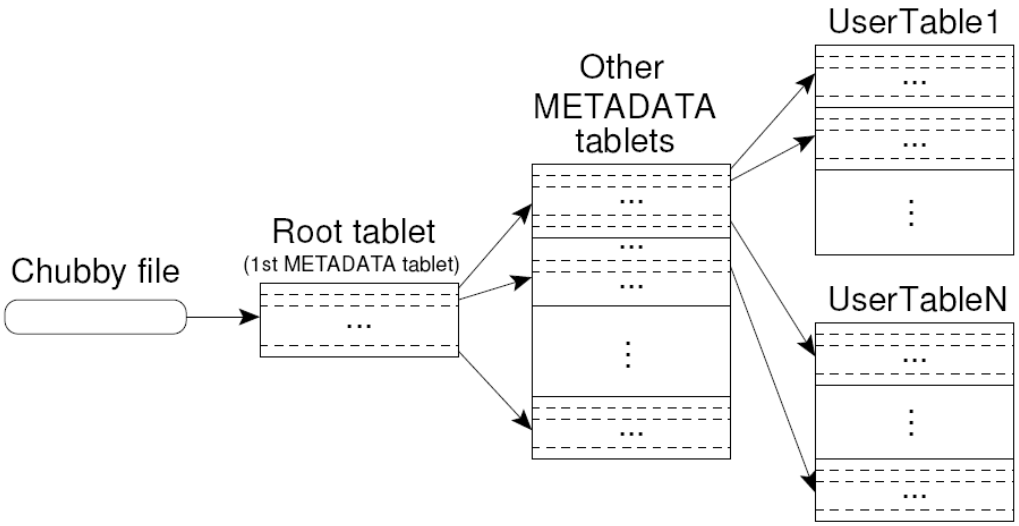

图 4 Tablet 位置层次结构

第一个层次是一个文件，存储在 Chubby 中，它包含了 Toot Tablet 的位置信息。Root Tablet 把 Tablet 的所有位置信息都保存在一个特定的 METADATA 表中。每个 METADATA 表都包含了一个 user tablet 集合的位置信息。Root Tablet 其实就是 METADATA 表当中的第

一个 Tablet，但是，它被区别对待，它在任何情况下都不会被拆分，从而保证 Tablet 位置层次结构不会超过三层。

METADATA 表存储了属于某个行键的 Tablet 的位置信息，所谓行键，就是关于 Tablet 表标识符和它的最后一行这二者的编码。每个 METADATA 行，大约在内存中存储了 1KB 的数据。由于采用了 128M 大小的 METADATA Tablet 的适当限制，我们的三层位置模式足够用来存放 2 的 34 此方的 Tablet 的位置信息。

客户端函数库会缓存 Tablet 位置信息。如果客户端不知道一个 Tablet 的位置信息，或者它发现，它所缓存的 Tablet 位置信息部正确，那么，它就会在 Tablet 位置层次结构中依次向上寻找。如果客户端缓存是空的，那么定位算法就需要进行三次轮询，其中就包括一次从 Chubby 中读取信息。如果客户端的缓存是过期的，定位算法就要进行六次轮询，因为，只有在访问无效的时候才会发现缓存中某个 entry 是过期的（这里假设 METADATA Tablets 不会频繁移动）。虽然，Tablets 位置信息是保存在缓存中，从而不需要访问 GFS，但是，我们仍然通过让客户端库函数预抓取 tablet 位置信息，来进一步减少代价，具体方法是：每次读取 METADATA 表时，都要读取至少两条以上的 Tablet 位置信息。

我们也在 METADATA 表中存储了二级信息，包括一个日志，它记载了和每个 tablet 有关的所有事件，比如，一个服务器什么时候开始提供这个 tablet 服务。这些信息对于性能分析和程序调试是非常有用的。

5.2 Tablet Assignment

在每回，每个 Tablet 可以被分配到一个 tablet 服务器。主服务器跟踪 tablet 服务器的情况，掌握当前 tablet 被分配到 tablet 服务器的情况，其中包括哪个 tablet 还没有被分配。当一个 tablet 没有被分配，并且一个具有足够空间可以容纳该 tablet 的 tablet 服务器是可用时，主服务器就把当前这个 tablet 分配给这个 tablet 服务器，主服务器会向 tablet 服务器发送一个 tablet 负载请求。

BigTable 使用 Chubby 来跟踪 tablet 服务器。当一个 Tablet 服务器启动的时候，它创建并且获得一个独占的排他锁，这个锁会锁住一个特定的 Chubby 目录中的一个唯一命名的文件。主服务器监视这个目录（服务器目录），来发现 tablet 服务器。如果一个 tablet 服务器停止服务，它就会丢失这个锁，比如，由于网络故障，导致这个 tablet 服务器丢失了这个 Chubby 会话。（Chubby 提供了一个完善的机制，来允许一个 tablet 服务器检查自己是否已经丢失了这个独占排他锁）。如果丢失了锁，那么，只要目录中的这个文件还存在，那么一个 tablet 服务器就会努力去获得这个锁。如果文件不再存在，那么，这个 tablet 服务器就不再能够对外提供服务，因此，它就自杀。一旦一个 tablet 服务器终止了服务（比如，簇管理系统把这个 tablet 服务器从簇中移除），它就会努力释放锁，这样，主服务器就可以更快地重新分配这个 tablet。

主服务器需要探测，什么时候 tablet 服务器不再提供 tablet 服务，并且要负责尽快对这些 tablet 进行重新分配。为了探测什么时候 tablet 服务器不再提供 tablet 服务，主服务器会周期性地询问每个 tablet 服务器，了解他们的锁的状态。如果一个 tablet 服务器报告，它已经丢失了锁；或者，在最近的几次尝试中，主服务器都无法与 tablet 服务器取得联系，主服务器就会努力获得一个针对这个服务器文件的独占排他锁。如果主服务器可以获得这个锁，那么，Chubby 就是可用的，相应地，这个 tablet 服务器或者已经死亡，或者有些故障导致它无法到达 Chubby。因此，主服务器就从 Chubby 中删除这个 tablet 服务器的文件，从而确保这个 tablet 服务器不再能够提供服务。一旦一个服务器文件被删除，主服务器就可以把所

有以前分配给该服务器的 tablet，都移动到“待分配”tablet 集合。为了保证一个 BigTable 簇不会轻易受到主服务器和 Chubby 之间的网络故障的影响，如果一个主服务器的 Chubby 会话过期了，这个主服务器就会自杀。但是，正如上所述，主服务器失效，不会改变 tablet 到 table 的分配。

当一个主服务器被簇管理系统启动时，在它能够改变 tablet 分配之前，它必须首先了解当前的 tablet 分配信息。为此，在启动的时候，主服务器会执行以下步骤：（1）主服务器在 Chubby 中抓取一个独特的 master lock，这就防止了多个主服务器并发启动的情形。（2）主服务器扫描 Chubby 中的服务器目录，从而发现当前可用的服务器。（3）主服务器和当前每个可用的 tablet 服务器通信，来发现哪些 tablets 已经被分配到哪个 tablet 服务器。（4）主服务器扫描 METADATA 表来学习 tablets 表集合。一旦在扫描过程中，主服务器发现某个 tablet 还没有被分配，主服务器就把这个 tablet 放置到“待分配”tablet 集合，这就使得这些 tablet 可以进入待分配状态。

一个比较复杂的情况是，在 METADATA tablets 被分配之前，我们是不能扫描 METADATA 表的。因此，在开始扫描（步骤 4）之前，如果主服务器在步骤 3 的扫描中发现 root tablet 没有被发现，主服务器把 root tablet 增加到待分配 tablet 集合。这个增加，保证了 root tablet 一定会被分配。因为，root tablet 包含了所有 METADATA tablets 的名字。主服务器只有在扫描了 root tablet 以后，才可能知道所有这些 METADATA tablets 的名字。
现有的 tablet 集合，只有在以下情形才会发生改变：（1）当一个 tablet 被创建或删除；（2）对两个现有的 tablet 进行合并得到一个更大的 tablet；（3）一个现有的 tablet 被分割成两个较小的 tablet。主服务器可以跟踪这些变化。Tablet 分裂会被特殊对待，因为，它是由一个 tablet 服务器发起的。Tablet 服务器通过把信的 tablet 信息记录在 METADATA 表中，来提交分裂操作。当分裂被提交以后，这个 tablet 服务器会通知主服务器。为了防止分裂通知丢失（或者由于主服务器死亡或者由于 tablet 服务器死亡），当主服务器要求一个 tablet 服务器加载已经被分裂的 tablet 时，主服务器会对这个新的 tablet 进行探测。Tablet 服务器会把分裂的情况告知主服务器，因为，它在 METADATA 表中所找到的 tablet entry，只能确定主服务器要求它加载的数据的一部分。

5.3 Tablet Serving

一个 tablet 的持久化存储是存在 GFS 当中，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，并且重构 memtable，方法是，执行重做点以后的所有已经提交的更新。

当一个写操作到达 tablet 服务器，服务器首先检查它是否是良好定义的，并且发送者是否被授权执行该操作。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。一个有效的变化，会被写到提交日志中。分组提交是为了改进许多小更新[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。

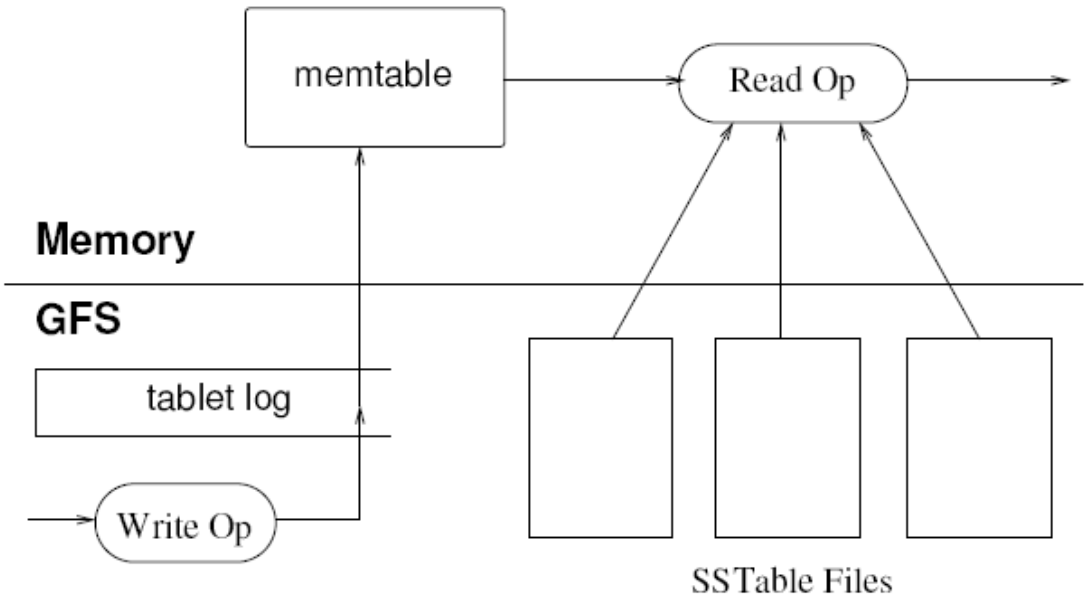

图 5 Tablet Representation

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以下二者的合并的基础上执行的，即一系列 SSTable 和 memtable。由于 SSTable 和 memtable 是字典排序的数据结构，合并视图的执行是非常高效的。

当 tablet 发生合并或分解操作时，正在到达的读写操作仍然可以继续进行。

5.4 Compactions

随着写操作的执行，memtable 的尺寸逐渐增加。当 memtable 的尺寸到达一个门槛值的时候，memtable 就被冻结，就创建一个新的 memtable，被冻结的 memtable 就转化成一个 SSTable，并被写入到 GFS。这个“次压缩”（minor compaction）过程有两个目标：（1）它缩减了 tablet 服务器的内存使用率；（2）当发生服务器死亡需要恢复时，它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，如果这种行为没有限制地持续进行，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 SSTable，它不包含删除信息或被删除的数据。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 Refinements

以前章节所描述的实现，需要一系列完善措施从而获得高性能、可用性和可靠性，这些都是我们的用户所要求的。这部分内容更详细介绍实现细节。

Locality groups

客户端可以把多个列家族一起分组到一个 locality group 中。我们会为每个 tablet 中的每个 locality group 大都创建一个单独的 SSTable。把那些通常不被一起访问的列家族分割到不同的 locality group，可以实现更高效的读。例如，在 WebTable 当中的页元数据（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。那些想要读取页面元数据的应用，就不需要访问所有的页面内容。

除此以外，一些有用的参数，可以针对每个 locality group 来设定。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，采用被动加载的方式被加载 tablet 服务器的内存，即只有应用请求 SSTable 中的数据，而这些数据又不在内存中时，才把 SSTable 加载到内存。一旦加载，属于这些 locality group 的列家族，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。

Compression

客户端可以决定是否对相应于某个 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（块的尺寸可以采用与 locality group 相关的参数来进行控制）。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。

当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 当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簇当中，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。

Caching for read performance

为了改进读性能，tablet 服务器使用两个层次的缓存。Scan 缓存是一个高层次的缓存，它缓存了“键-值”对，这些“键-值”对是由 tablet 服务器代码的 SSTable 接口返回的。Block 缓存是比较低层次的缓存，它缓存了从 GFS 当中读取的 SSTable 块。Scan 缓存对于那些频繁读取相同数据的应用来说是非常有用的。Block 缓存对于那些倾向于读取与自己最近读取数据临近的数据的应用来说，是比较有用的，比如顺序读取，或者随机读取属于同一个 locality group 中的不同的列。

Bloom filters

正如 5.3 节阐述的那样，一个读操作必须从构成一个 tablet 的当前状态的所有 SSTable 中读取数据。如果这些 SSTable 不在内存中，我们就不得不需要很多磁盘访问。我们通过下面的方式来减少磁盘访问，即允许客户端来确定，为某个特定 locality group 中的 SSTable 创建 Bloom filter[7]。一个 Bloom filter 允许我们询问，一个 SSTabble 是否包含属于指定的“行-列队”的特定的数据。对于某个特定的应用，一个用来存储 Bloom filter 的很少量的 tablet 服务器内存空间，都可以极大减少读操作的磁盘访问次数。我们使用 Bloom filter 也意味着，许多针对目前不存在的行或列的查询，根本就不需要访问磁盘。

Commit-log implementation

如果我们为每个 tablet 都设置一个单独的文件来保存提交日志，那么，在 GFS 中，会有大量的文件并发写操作。取决于每个 GFS 服务器底层文件系统的实现方式，这些写操作会引起大量的磁盘访问。除此以外，让每个 tablet 都有一个单独的日子文件，会降低分组提交优化的效率。为了解决这些问题，我们对每个 tablet 服务器具备一个独立日志文件这种方式进行了补充，不是把更新都写入相应 tablet 的独立文件中，而是把几个不同 tablet 的更新内容都添加到一个同样的物理日志文件中[18][20]。

使用单个日志文件，明显改善了普通操作时的性能收益，但是，它使得故障恢复变得复杂起来。当一个 tablet 服务器死亡，它上面承载的 tablet 会被重新分配到其他多个 tablet 服务器，每个 tablet 服务器通常只接收一部分 tablet。为了给一个 tablet 恢复状态，新的 tablet 服务器需要根据原来 tablet 服务器中记载的提交日志文件，为这个 tablet 重新执行一遍更新操作。但是，针对这些 tablet 的更新操作的日子记录，都混合存放在一个日志文件中。一种方法是，让新的 tablet 服务器完整地读取一份提交日志文件，然后，只把恢复这个 tablet 时所需要的日志记录读取出来，完成恢复。但是，在这种机制下，如果有 100 个机器，每个都被分配了来自死亡 tablet 服务器的一个单独的 tablet，那么，这个日志文件就必须被重复读取 100 遍。

我们通过以下的方式来避免日志的重复读写：首先以键（表，行名称，日志顺序号）的顺序对日志文件的条目（entry）进行排序。在排序得到的结果中，所有针对某个特定 tablet 的更新都是连续存放的，因此，可以通过一次磁盘寻找，然后顺序批量读取数据，这种方法具有较高的效率。为了实现并行排序，我们把日子文本分割成 64MB 为单位的分段，在不同的 tablet 服务器上对每个分段进行并行排序。主服务器会对排序过程进行协调，当一个

tablet 服务器表示需要从一些提交日志文件中恢复数据时，就需要启动排序过程。

书写提交日志到 GFS 中去，由于很多原因（比如 GFS 服务器上包含很多冲突操作，或者网络拥塞），有时会带来性能上的瓶颈。为了使得更新免受 GFS 延迟的影响，每个 tablet 服务器实际上有两个日子书写线程，每个线程都书写到它自己的日志中。在运行时，只有一个进程处于工作状态。如果当前活跃的日志书写线程性能比较差，那么，就切换到另一个日志书写线程。在更新日志队列中的更新就会被这个新激活的线程书写。日志条目包含了序号，这就可以帮助恢复过程删除那些由于线程切换而导致的重复的日志记录。

Speeding up tablet recovery

如果主服务器把一个 tablet 从一个 tablet 服务器转移到另一个 tablet 服务器。这个源 tablet 服务器就对这个 tablet 做一个次压缩（minor compaction）。通过减少 tablet 服务器中的提交日志中的未压缩状态的数量，压缩过程减少了恢复时间。在完成这个压缩过程以后，这个源 tablet 服务器就停止提供针对这个 tablet 的服务。在它实际上卸载这个 tablet 之前，这个源 tablet 服务器要做另一个次压缩，来删除本 tablet 服务器的日志中任何未压缩的状态，这些未压缩状态是在第一个次压缩进行过程中产生的。当第二个次压缩完成时，这个 tablet 就可以被加载到另一个 tablet 服务器，而不需要任何日志条目的恢复。

Exploiting immutability

除了 SSTable 缓存，BigTable 系统的其他部分也已经被简化，这些简化基于这样一个事实，即我们所产生的所有 SSTable 都是不变的。例如，当我们从 SSTable 中读取数据时，不需要进行任何文件系统访问的同步。结果是，针对行级别的并发控制可以高效地执行。唯一发生变化的数据结构是 memtable，它同时被读操作和写操作访问。为了减少读取 memtable 过程中的冲突，我们使得每个 memtable 行采取”copy-on-write”，并且允许读和写操作并行执行。

由于 SSTable 是不可变的，永久性删除数据的问题就转变成，收集废弃的 SSTable。每个 tablet 的 SSTable，都会在 METADATA 表中进行注册。主服务器移除废弃的 SSTable，把它标记为垃圾数据集合。

最后，SSTable 的不可变性，允许我们更快地分裂 tablet，而不是为每个子 tablet 生成一个新的 SSTable 集合。我们让子 tablet 和父 tablet 共享一个 SSTable 集合。

7 Performance Evaluation

我们建立了一个 BigTable 簇，它具有 N 个 tablet 服务器，我们改变 N 的值，从而测试 BigTable 的性能和可扩展性。Tablet 服务器的配置为，1GB 内存，并且把数据写到 GFS 单元格中，这个 GFS 单元格包含了 1786 个机器，每个机器具备 400GB IDE 硬盘。我们让 N 个客户端机器产生针对 BigTable 的负载，从而实现测试。需要指出的是，这里我们采用的客户端的数量和 tablet 服务器的数量是相等的，从而保证客户端不会成为瓶颈。每个机器都有两个双核 Opteron 2GHz 芯片、足够的物理内存（从而能够容纳工作集产生的数据）和 GB

带宽的网络连接。这些机器被安排为两层树型交换网络，在根部的带宽可以达到100-200Gbps。所有这些机器都采用相同的配置，都在同一个域内，因此，任何两台机器之间的通讯时间都不会超过 1 微秒。

Tablet 服务器、主服务器、测试客户端和 GFS 服务器都在同一个机器集合上运行。每个机器都运行一个 GFS 服务器。同时，一些机器还会另外运行一个 tablet 服务器，一个客户端进程。

R 是 BigTable 的不同值的行键的数量，在选择 R 的值的时候，需要确保每个 benchmark 在每个 tablet 服务器上读或写大约 1GB 的数据量。

负责顺序写的 benchmark，使用名称从 0 到 R-1 的行键。行键的空间被分区成 10N 个等尺寸区间。这些区间被中央调度器分配到 N 个客户端，一旦，某个客户端已经处理完前面分配到的区间后，中央调度器就立即给这个客户端分配另一个区间。这种动态分配，可以减轻其他运行于当前机器上的进程对该机器的影响。我们在每个行键下面，写了一个单个字符串。每个字符串都是随机生成的，因此，不具备可压缩性。除此以外，在不同行键下面的字符串都是不同的，这样就不会存在跨行压缩。负责随机写的 benchmark 也是类似的，初了行键在写之前要以 R 为模数进行哈希操作，从而使得在整个 benchmark 期间，写操作负载可以均匀分布到整个行空间内。

负责顺序读的 benchmark 产生行键的方式，也和负责顺序写的 benchmark 类似。但是，顺序读 benchmark 不是在行键下面写数据，而是读取存储在行键下面的字符串，这些字符串是由前面的顺序写 benchmark 写入的。类似地，随机读 benchmark 会以随机的方式读取随机写 benchmark 所写入的数据。

Scan benchmark 和顺序读 benchmark 类似，但是，使用了由 BigTable API 所提供的支持，来扫描属于某个域区间内的所有值。使用一个 scan，减少了由 benchmark 所执行的 RPC 的数量，因为，在 Scan 中，只需要一个 RPC 就可以从 tablet 服务器中获取大量顺序值。

随机读(mem) benchmark 和随机读 benchmark 类似，但是，包含这个 benchmark 数据的局部群组，被标记为 in-memory，因此，它是从内存中读取所需数据，而不是从 GFS 中读取数据。对这个 benchmark 而言，我们把每个 tablet 服务器所包含数据的数量，从 1GB 减少到 100MB，从而可以很好地装入 tablet 服务器可用内存中。

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

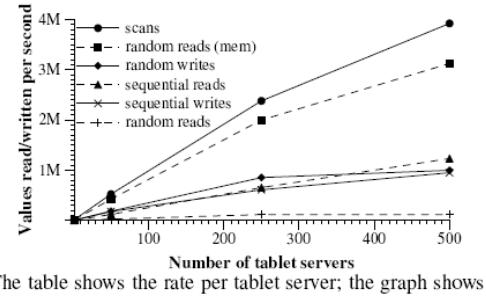

Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

图 6 显示了，当读或写 1000 字节的数据到 BigTable 中时，我们的 benchmark 的性能。表格内容显示了每个 tablet 服务器每秒的操作的数量。图显示了每秒的操作的总数量。

Single tablet server performance

让我们首先考虑只有一个 tablet 服务器的性能。随机读的性能要比其他类型的操作的性能都要差一个数量级，甚至差更多。每个随机读操作包含了，把一个 64KB 的 SSTable 块通

过网络从 GFS 传输到一个 tablet 服务器，在这个块中，只有 1000 字节的数据会被使用。Tablet 服务器大约每秒执行 1200 个操作，这会导致从 GFS 需要读取大约 75MB/s。在这个过程中，带宽是足够的。许多具有这种访问模式的 BigTable 应用都把块大小设置为更小的值 8KB。

从内存读取数据的随机读会更快一些，因为，每个 1000 字节读都是从 tablet 服务器内存当中读取，而不需要从 GFS 当中读取 64KB 的数据。

随机和顺序写执行的性能要好于随机读，因为，每个 tablet 服务器都把所有到达的写都追加到一个单独的提交日志中，并且使用分组提交，从而使得到 GFS 的写操作可以流水化。在随机写和顺序写之间，没有明显的性能差别。在两种情形下，所有的写都记录在同一个提交日志中。

顺序读的性能要好于随机读，因为从 GFS 中获取的每 64KB SSTable 块，会被存储到块缓存中，它可以为后面的 64 个读操作服务。

Scan 操作会更快，因为，针对单个 RPC，tablet 服务器可以返回大量的值，因此，RPC 开销就可以分摊到大量的值当中。

Scaling

当我们在系统中把 tablet 服务器的数量从 1 增加到 500 的过程中，累计吞吐量急剧增加，通常以 100 倍的规模。例如，随着 tablet 服务器增长了 500 倍，针对内存的随机读的性能，增长了大约 300 倍，之所以会发生这种情况，因为这个 benchmark 的性能瓶颈是单个 tablet 服务器的 CPU。

但是，性能不会线性增长。对于多数 benchmark 来说，当把 tablet 服务器的数量从 1 增加到 50 时，每个服务器的吞吐量有显著的降低。这个问题，是由多服务器环境中，负载不均衡引起的，通常由于有其他进程争抢 CPU 资源和网络带宽。我们的负载均衡算法努力解决这个问题，但是，无法实现完美的目标，主要原因在于：第一，重新负载均衡有时候会被禁止，这样做可以减少 tablet 迁移的数量（当一个 tablet 迁移时，在短时间内是不可用的，通常是一秒）；第二，我们的 benchmark 的负载是动态变化的。

随机读 benchmark 显示了最差的可扩展性。当服务器数量增加 500 倍时，累计吞吐量的增长只有 100 倍左右。导致这个问题的原因在于，对于每个 1000 字节的读操作，我们都会转移一个 64KB 的块。这个数据转移使得我们的 GB 级别的网络很快达到饱和，这样，当我们增加机器的数量时，单个服务器的吞吐量就会很快降低。

# 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.

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.

8 Real Applications

截止到 2006 年 8 月，已经又 388 个非测试的 BigTable 簇运行在不同的 Google 服务器簇里，包括总共大约 24500 个 tablet 服务器。表 1 显示了每个簇中的 tablet 服务器的大概分布。许多簇都是用于开发目的，因此，在很多时间内都是空闲的。一组包含 14 个繁忙簇（包含 8069 个 tablet 服务器），每秒钟的累积负载是 120 万个请求。其中，到达的 RPC 流量是 741MB/s，出去的 RPC 流量大约是 16GB/s。表 2 提供了一些关于当前正在使用的表的数据。一些表存储为用户服务的数据，而其他表则存储为批处理服务的数据。在总尺寸、平均单元格尺寸、从内存服务的数据的百分比以及表模式的复杂性方面，这些表区别很大。下面，我们将要分别描述三个产品团队如何使用 BigTable。

8.1 Google Analytics

Google Analytics 是一种服务，它帮助网站管理者分析网站流量模式。它提供了汇总分析，比如，每天不同访问者的数量，以及每天每个 URL 的网页视图的数量，以及网站流量报告，比如浏览了某个网页以后发生购买行为的用户的数量。

为了支持这项服务，网站管理员在网页中嵌入了一个小的 Javascript 程序。每当页面被访问时，都会触发这个 Javascript 程序。它把关于这个用户访问的所有相关信息都记录在 Google Analytics 中，比如用户标识以及被访问的网页的信息。Google Analytics 分析这些数据，并提供给网站管理员访问。

我们简单地描述 Google Analytics 所使用的两个表。网站点击表（200TB）为每个用户访问都维护了一个行。行的名称是一个元祖，它包含了网站的名称以及 session 被创建的时间。这个表模式保证了，访问同一个 WEB 站点的 session 都是临近的，并且以时间顺序进行存储。这个表的压缩率可以达到 14%。

汇总表（大约 20TB）包含了针对每个网站的不同的汇总信息。这个表是从网站点击表中通过周期性地运行 MapReduce 作业而得到的。每个 MapReduce 作业从网站点击表当中抽取最近的 session 信息。系统的总吞吐量，受到了 GFS 吞吐量的限制。这个表的压缩率在 29%左右。

8.2 Google Earth

Google 提供很多服务，它支持用户访问高清晰度的卫星图片，或者通过基于浏览器的 Google Maps 接口，或者通过定制的客户端软件 Google Earth。这些产品允许用户在地球表

面进行导航。该产品支持在不同清晰度下查看和标注地图信息。该系统采用一个表进行数据预处理，用另一个表位用户提供数据服务。

这个预处理管道使用一个表来存储卫星图片。在预处理过程中，影像数据被清洗并转换成最后的服务数据。这个表包含了大约 70TB 的数据，因此，是从磁盘访问的。影像数据都是经过压缩的，因此，BigTable 就不存在数据压缩了。

影像表中的每行，都对应一个单个的地理片段。行被命名，以保证相邻的地理分段在存储时彼此接近。表包含了一个列家族来跟踪每个分段的数据。这个列家族包含了大量的列，基本上为每个原始影像创建一个列。因为，每个分段只是从少量几个列中的影像构建得到的，因此，这个列家族很稀疏。

这个预处理管道严重依赖于针对 BigTable 的 MapReduce 操作来转换数据。在这些 MapReduce 作业运行期间，整个系统的每个服务器大约处理 1MB/秒的数据。

这个服务系统使用一个表来对存储在 GFS 中的数据建立索引。这个表相对比较小，大约 500GB。但是，每个数据中心每秒钟必须以很低的延迟处理成千上万个请求。因而，这个表通常被重复存放到多个 tablet 服务器上，并且包含了内存列家族。

8.3 Personalized search

Personalized search 是一种服务，它记录了用户查询和点击数据，涵盖了各个方面的 Google 属性，比如，网页搜索、图片和新闻。用户可以浏览他们自己的访问历史，他们可以要求根据 Google 使用历史模式来获得个性化的搜索结果。

Personalized search 把所有的用户数据都存放在 BigTable 中。每个用户都有一个独特的用户 ID，并被分配了以这个用户 ID 为名字的行。每种类型的动作都会存储到特定的列家族中，比如，有一个列家族存储了所有的网页查询。每个数据元素都把这个动作发生的时间作为 BigTable 的时间戳。Personalized search 在 BigTable 上使用 MapReduce 产生用户的 profile。这些用户 profile 用来协助生成个性化的用户搜索结果。

Personalized Search 的数据被分布到多个 BigTable 簇，来增加可用性，减少由距离而产生的延迟。Personalized Search 最初构建了基于 BigTable 的客户端副本模式，从而保证所有副本的最终一致性。当前的系统使用了一个复制子系统，它是内嵌到服务器端的。

Personalized Search 存储系统的设计，允许其他分组增加新的用户信息到他们自己的列中，这个系统当前正在被许多 Google 属性使用，这些属性需要存储用户的配置信息和设置。在多个分组之间共享一个表，导致了列家族数量比较大。为了帮助支持共享，我们为 BigTable 增加了一个简单的配额机制，从而对共享表当中某个特定应用可以使用的存储空间进行限制。这种机制为每个用户在使用本系统的不同的产品分组之间建立了隔离。

9 Lessons

在设计、实现、维护和支持 BigTable 的过程中，我们收获了有益的经验，并且获得了几个有意思的教训。

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，我们已经遇到问题，他们是由以下原因引起的：内存和网络故障、大的时钟扭曲、机器挂起、我们所使用的其他系统（比如 Chubby）中存在的故障、GFS 配额当中的溢出以及计划或非计划之中的硬件维护。随着我们获得了更多的针对这些问题的经验，我们已经通过改变不同的协议来解决他们。例如，我们为 RPC 机制

增加了 checksumming。我们通过移除一部分系统针对另一部分系统所做的假设，也解决了一些问题。例如，我们取消了假设一个给定的 Chubby 操作只返回一个固定错误集合。

我们获得的另一个教训是，在很清楚地知道如何使用一个新特性之前，不要随便增加这个新特性。例如，我们最初计划在我们的应用 API 当中支持通用的事务。因为我们不会立即用到这种功能，所以，我们没有实现它。现在我们有许多应用都是运行在 BigTable 之上，我们就可以考察他们的应用需求，发现很多应用只需要针对单个记录的事务。当人们需要分布式事务时，最主要的用途就在于维护二级索引，我们就考虑增加一个特定的机制来满足这种需求。在通用性方面，这种特定机制会比通用事务模型差一些，但是，它更高效，尤其是对于那些需要跨越多个行的更新而言。

从支持 BigTable 运行中获得的一个比较实际的教训是，合适的系统级别的监视的重要性，即监视 BigTable 自己，也监视运行在 BigTable 上的进程。例如，我们扩展了我们的 RPC 系统，从而使得对一些 RPC 样本，它会详细跟踪记录针对该 RPC 的重要动作。这种特性已经允许我们探测和解决许多问题，比如针对 tablet 数据结构的锁冲突，当执行 BigTable 更新时的慢写，以及当 METADATA tablet 不可用时，访问 METADATA 表发生的阻塞。说明监视比较有用的另一个例子是，每个 BigTable 簇都是在 Chubby 中注册的。这就允许我们跟踪所有的簇，发现这些簇的大小，看看它们当前使用的是我们软件的哪个版本，以及是否存在一些预料之外的问题，比如很大的延迟。

我们所获得的最重要的教训就是简单设计的价值。假设我们系统的尺寸是大约 10 万行非测试代码，以及这些代码会随着时间演化，我们发现代码和设计的清晰性对于代码和系统维护具有重要的帮助。一个例子是我们的 tablet 服务器成员协议。我们的第一个协议很简单，主服务器周期性地发布租约给 tablet 服务器，如果 tablet 服务器发现租约到期就自杀。不幸的是，当存在网络问题时，这种协议极大降低了系统的可用性，并且对于主服务器的恢复时间也很敏感。我们先后几次更改了协议的设计，直到找到一个好的协议。但是，这个比较好的协议，太复杂，并且依赖于那些很少被其他应用使用的 Chubby 功能。我们发现，我们花费了大量的时间来调试各种晦涩的案例。最后，我们废弃了这个协议，转向采用一个比较简单的新的协议，它只依赖那些经常被使用的 Chubby 功能。

10 Related Work

Boxwood 项目[24]具有一些和 Chubby、GFS 以及 BigTable 重叠的组件，因为 Boxwood 支持分布式协议和分布式 B 树存储。在每个发生重叠的领域中，Boxwood 看起来似乎针对的是 Google 所提供服务的低一层次的服务。Boxwood 项目的目标是提供构建高层次服务的基础架构，比如文件系统或数据库，而 BigTable 的目标是直接支持那些需要存储数据的客户端应用。

许多最近的计划，已经解决了在广域网内提供分布式存储和高层次服务的问题。这些计划包括，在分布式哈希表方面的工作，比如项目 CAN[29]、CHORD[32]、Tapestry[37]和 Pastry[30]。这些系统解决的问题，在 BigTable 中没有出现，比如高可用性带宽、不可信的参与者或频繁地重配置。分布式控制和理想的零错误不是 BigTable 的设计目标。

就可以提供给应用开发者的分布式存储模型而言，我们认为，分布式 B 树所提供的“键-值对”模型或者分布式哈希表具有很大的局限性。“键-值对”是一个有用的积木，但是，它不应该是可以提供给应用开发者的唯一的积木。我们所能提供的模型比简单的“键-值对”更加丰富，并且支持稀疏的办结构化数据。但是，它是非常简单的，并且足以支持非常高效的平面文件表示，并且它足够透明，允许用户来调节系统重要的行为。

有几个数据库开发商已经开发了并行数据库，它可以存储大量的数据。Oracle 的 Real Application Cluster 数据库[27]，使用共享的磁盘来存储数据（BigTable 使用 GFS），并且使用一个分布式的锁管理器（BigTable 使用 Chubby）。IBM 的 DB2 Parallel Edition[4]是基于非共享[33]的体系架构，这一点和 BigTable 类似。每个 DB2 服务器负责表中行的一个子集，它被存储在一个局部关系数据库中。IBM 的 DB2 Parallel Edition 和 Oracle 的 Real Application Cluster 都提供了完整的事务模型。

BigTable 局部分组实现了和其他一些系统类似的压缩和磁盘读性能，这些系统在磁盘上存储数据室，采用基于列而不是基于行的存储，包括 C-store[1,34]和商业化产品，比如 Sybase IQ[15,36]，SenSage[31]，KDB+[32]，以及在 MonetDB/X100[38]当中的 ColumnBM 存储层。另一个把数据垂直和水平分区到平面文件中并且取得了很好的数据压缩率的系统是，AT&T 的 Daytona 数据库[19]。局部分组不支持 CPU 缓存级别的优化，比如那些在 Ailamaki[2]中描述的。

BigTable 使用 memtable 和 SSTable 来存储 tablet 更新的方式，和 Log-Structured Merge Tree[26]用来存储索引更新的方式是类似的。在这两个系统中，排序的数据在写入到磁盘之前是存放在内存之中的，读操作必须把来自磁盘和内存的数据进行合并。

C-Store 和 BigTable 具有很多相似的特性：两个系统都使用非共享的体系架构，都有两个不同的数据结构，一个是为存放最近的写操作，一个是为存放长期的数据，并且设计了把数据从一个结构转移到另一个结构的机制。但是，这两个系统在 API 方面具有很大的区别：C-Store 就像一个关系型的数据库，而 BigTable 提供了一个低层次的读和写接口，并且被设计成在每个服务器上每秒钟内提供成千上万的这种操作。C-Store 也是一个读优化的关系型 DBMS，而 BigTable 对读敏感和写敏感的应用都提供了很好的性能。

BigTable 的负载均衡器，必须解决负载均衡和内存均衡问题，这也是非共享数据库[11,35]通常面临的问题。我们的问题更加简单一些：（1）我们不考虑同一个数据存在多个副本的情形，我们可能采用视图或索引的形式以另一种方式来看待副本；（2）我们让用户告诉我们，什么数据应该放在内存中，什么数据应该放在磁盘上，而不是系统自己尽力去猜测数据的存放位置；（3）我们没有复杂的查询优化机制，也不执行复杂的查询。

11 Conclusions

我们已经描述了 BigTable，一个为 Google 存储数据的分布式存系统。自从 2005 年 4 月开始，BigTable 簇已经在实际生产系统中使用。在那之前，我们已经投入了 7 个 person-years 在设计和开发上。截止 2006 年 8 月，有超过 60 项计划正在使用 BigTable。我们的用户喜欢 BigTable 提供的高性能和高可用性。当用户的资源需求随着时间变化时，他们只需要简单地往系统中添加机器，就可以实现服务器簇的扩展。

鉴于 BigTable 的不同寻常的接口，一个有意思的问题是，我们的用户适应 BigTable 的过程是多么艰难。新的用户有时候不知道如何很好地使用 BigTable 提供的接口，尤其是，他们已经习惯使用支持通用事务的关系型数据库。但是，许多 Google 产品成功地使用 BigTable 来存放数据这样一个事实，已经说明 BigTable 设计的成功。

我们正在考虑增加其他几个额外的 BigTable 功能，比如支持二级索引，以及一个基础框架，它可以支持构建跨数据中心分布 BigTable 数据。我们已经开始把 BigTable 作为一个服务提供给产品组，从而使得每个组都不需要维护他们自己的簇。随着我们的服务簇的扩展，我们需要处理更多的 BigTable 内部的资源共享的问题。

最后，我们发现，在 Google 内部构件自己的存储解决方案具有明显的优势。我们在为 BigTable

设计我们自己的数据模型中获得了很大的灵活性。除此以外，我们针对 BigTable 实现的控制，以及其他 BigTable 所依赖的 Google 基础设施，意味着，我们可以移除瓶颈和缺点。

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.（厦门大学林子雨翻译 标注：致谢就不翻译了啊）

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 fits 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. CACM3, 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.

（厦门大学计算机系 林子雨 翻译）

[本文翻译的原始出处：厦门大学计算机系数据库实验室网站林子雨老师的云数据库技术资料专区 http://dblab.xmu.edu.cn/cloud_database_view]