Bigtable：一个分布式的结构化数据存储系统

摘要

Bigtable 是一个管理结构化数据的分布式存储系统，它被设计用来处理海量数据：分布在数千台通用服务器上的 PB 级的数据。Google 的很多项目将数据存储在 Bigtable 中，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，无论是在数据规模（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，针对所有 Google 这些产品，Bigtable 还是成功地提供了一个灵活的、高性能的解决方案。本文描述了 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布局和格式；并且我们还将描述 Bigtable 的设计和实现。

1 介绍

在过去两年半时间里，我们设计、实现并部署了一个用于管理结构化数据的分布式的存储系统—在 Google，我们称之为 Bigtable。Bigtable 的设计目的是可靠地适应 PB 级别的数据和成千上万台机器。Bigtable 已经实现了下面的几个目标：广泛的适用性、可扩展、高性能和高可用性。已经有超过 60 个 Google 的产品和项目在使用 Bigtable，包括 Google Analytics、Google Finance、Orkut、Personalized Search、Writely 和 Google Earth。这些产品使用 Bigtable 完成迥异的工作负载需求，这些需求从面向吞吐量的批处理作业到对终端用户而言延时敏感的数据服务。它们使用的 Bigtable 集群的配置也有很大的差异，从少数机器到成千上万台服务器，这些服务器里最多可存储几百 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布局和格式（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)在底层存储中展示的数据的位置属性(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)。数据用行和列的名字进行索引，名字可以是任意的字符串。虽然客户程序通常会在把各种结构化或半结构化的数据串行化到字符串里，Bigtable 同样将数据视为未经解析的字符串。通过仔细选择数据的模式，客户可以控制数据的位置。最后，可以通过 BigTable 的模式参数动态地控制数据读或写(control whether to serve data out of memory or from disk)。

第二节更详细地描述了数据模型，第三节概要介绍了客户端 API；第四节简要介绍了 BigTable 依赖的底层 Google 基础框架；第五节描述了 BigTable 实现的基本原理；第 6 节描述了为了提高 BigTable 的性能而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 节讨论了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 节介绍了相关工作，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 由行关键字、列关键字以及时间戳索引；Map 中的每个 value 都是一个未经解析的字节数组。
(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，决定选用这个数据模型。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；假设我们想要备份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，使用网页的各种属性（aspect）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 列族容纳的是网页的内容，anchor 列族容纳引用该网页的锚链接文本。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。每个锚链接数据项只有一个版本（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
表中的行关键字是任意字符串（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）。在单一行关键字下的每一个读或者写操作都是原子的（不管在这一行里被读或者写的不同列的数目），这个设计决策能够使用户很容易地推测（reason about）对同一个行进行并发更新操作时的系统行为。
Bigtable 通过行关键字的字典顺序来维护数据。表中一定范围内的行被动态分区。每个分区叫做一个”Tablet”，Tablet 是数据分布和负载均衡的单位。这样做的结果是，读取一定范围内的少数行很高效，并且往往只需要跟少数机器通信。用户可以通过选择他们的行关键字来开发这种特性，这样可以为他们的数据访问获得好的本地性(get good locality)。举例来说，我们在关键字 com.google.maps/index.html 的索引下为 maps.google.com/index.htm 存储数据。把相同的域中的网页存储在连续的区域可以让一些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“，列族构成了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（我们把同一个列族下的数据压缩在一起）。列族必须先创建，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。我们的意图是，一张表中不同列族的数目要小（最多几百个），并且列族在操作中很少改变。与此相反，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，但是限定词可以是任意字符串。比如，Webtable 有个列族 language，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单独一个锚链接，如图一所示。限定词是引用该网页的站点名；数据项内容是链接文本。
访问控制、磁盘和内存的计数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：一些应用可以添加新的基本数据、一些可

以读取基本数据并创建派生的列族、一些则只允许浏览现存数据（甚至可能因为隐私的原因不能浏览所有现存列族）。

时间戳

在 Bigtable 中，每一个数据项都可以包含同一数据的不同版本；这些版本通过时间戳来索引。Bigtable 时间戳是 64 位整型数。时间戳可由 Bigtable 指定，这种情况下时间戳代表精确到毫秒的“实时”时间，或者该值由库户程序明确指定。需要避免冲突的程序必须自己生成一个唯一的时间戳。数据项中不同版本按照时间戳倒序排列，所以最新的版本可以被先读到。

为了减轻多个版本数据的管理负担，我们对每一个列族提供两个设置参数，Bigtable 通过这两个参数可以对废弃版本的数据进行自动垃圾收集。用户既可以指定只保存最后 n 个版本的数据，也可以只保存“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）。

在我们的例子 Webtable 中，我们将存储在 contents:列中爬虫经过的页面的时间戳设置为这个版本的页面被实际爬过的时（alex 注: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每个网页的最近三个版本。

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);
```

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、从个别行中查找值、或者遍历表中的一个数据子集。图 2 中的 C++代码使用 RowMutation 抽象对象执行一系列的更新操作。（为了保持示例简洁，我们省略了无关细节）。对 Apply 的调用执行了了对Webtable 的一个原子修改（mutation）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());
```

}
图 3 中的 C++代码使用 Scanner 抽象对象遍历一个特定行内的所有锚点。客户程序可以遍历多个列族，有几种机制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，让它只输出那些列匹配正则表达式*.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，这些特性允许用户以更复杂的方法操作数据。首先，Bigtable 支持单行上事务处理，利用这个功能，用户可以对存储在一个单独行关键字下的数据执行原子性的读取-更新-写入操作。虽然 Bigtable 提供了一个允许用户跨行关键字（at the clients?）批量写入的接口，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 支持在服务器的地址空间内执行脚本程序。脚本程序使用 Google 开发的用于数据处理的 Sawzall 语言【28】书写。目前，我们基于 Sawzall 的 API 还不允许客户脚本将数据写回 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、以及通过多种操作符的汇总归纳。

Bigtable 可以和 MapReduce【12】一起使用，MapReduce 是 Google 开发的运行大规模并行计算的框架。我们已经开发了一套封装器（wrapper），这些封装器使 Bigtable 既可以作为 MapReduce 作业的源输入也可以作为目标输出输出。

4 BigTable 构件

Bigtable 是建立在一些其他 Google 基础架构之上的。BigTable 使用 Google 分布式文件系统(GFS)【17】存储日志和数据文件。BigTable 集群往往运行在一个共享的机器池中，池中的机器还会运行其它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 依赖集群管理系统在共享机器上调度作业、管理资源、处理机器的故障、以及监视机器的状态。

SSTable 文件格式

BigTable 数据在内部使用 Google SSTable 文件格式存储。SSTable 提供一个从键（key）到值（value）的持久化的、已排序、不可更改的映射（Map），这里的 key 和 value 的都是任意的字节（Byte）串。对 SSTable 提供了如下操作：查询与一个指定 key 值相关的 value，或者遍历指定 key 值范围内的所有键值对。从内部看，SSTable 是一连串的数据块（通常每个块的大小是 64KB，但是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。一次查找可以通过一次磁盘搜索完成：首先执行二分查找在内存索引里找到合适数据块的位置，然后在从硬盘中读取合适的数据块。也可以选择把整个 SSTable 都映射到内存中，这样就可以在不用访问硬盘的情况下执行查询搜索了。

Chubby 分布式锁服务

BigTable 还依赖一个高可用的、持久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，其中一个副本被选为 Master，并且积极处理请求。只有在大多数副本正常运行，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，出现故障时 Chubby 使用 Paxos 算法【9,23】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使用，对文件的读写操作都是原子的。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租约，这个会话就过期失效了(A client’s session expires

if it is unable to renew its session lease within the lease expiration time.)。当一个客户会话失效时，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 Chubby 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 完成以下各种任务：保证在任意时间最多只有一个活动的 Master；存储 BigTable 数据的引导程序的位置（参考 5.1 节）；发现 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。最近我们在跨越 11 个 Chubby 服务实例的 14 个 BigTable 集群上测量了这个影响。Bigtable 服务器时钟的平均比率是 0.0047%,在这期间由于 Chubby 不可用而导致 BigTable 中的部分数据不能访问（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%。

5 实现

Bigtable 的实现有三个主要的组件：链接到每个客户程序的库、一个 Master 服务器和多个 tablet 服务器。在一个集群中可以动态地添加（或者删除）一个 tablet 服务器来适应工作负载的变化。
Master 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、平衡 tablet 服务器的负载、以及对 GFS 中的文件进行垃圾收集。除此之外，它还处理模式修改操作，例如建立表和列族。
每个 Tablet 服务器都管理一组 tablet（通常每个 tablet 服务器有大约数十个至上千个 tablet）。tablet 服务器处理它所加载的 tablet 的读写操作，以及分割增长的过大的 tablet。和很多单主节点（Single-Master）类型的分布式存储系统【17.21】类似，客户数据都不经过 master 服务器：客户程序直接和 tablet 服务器通信来进行读写操作。由于 BigTable 的客户程序不依赖 master 服务器来获取 tablet 的位置信息，大多数客户程序甚至完全不和 master 通信。因此，在实际应用中 master 的负载是很轻的。
一个 BigTable 集群存储了很多表，每个表包含了一组 tablet，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，每个表只有一个 tablet 组成。随着表中数据的增长，它被自动分割成多个 tablet，默认情况下每个 tablet 的大小大约是 100MB 到 200MB。
5.1 Tablet 的位置信息
我们使用一个三层的、类似于 B+树[10]的结构存储 tablet 的位置信息(如图 4)。
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 在一个特殊的元数据（METADATA）表包含了里所有的 tablet 的位置信息。每一个元数据 tablet 包含了一组用户 tablet 的的位置信息。root tablet 实际上只是元数据表的第一个 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 的位置层次不会超过三层。
元数据表将每个 tablet 的位置信息存储在一个行关键字下，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。每一个元数据行在内存中大约存储了 1KB 数据。在一个大小适中的、大小限制为 128MB 的元数据 tablet 中，我们的三层结构位置信息模式足够寻址 2^{34}（三层 $2^{7+10}*2^{7+10}$）个 tablet（或者说在 128M 的元数据中可以存储 2^{61} 个字节）。
客户程序库会缓存 tablet 的位置信息。如果客户程序不知道一个 tablet 的位置信息，或者发现它缓存的地址信息不正确，那么客户程序就递归移动到 tablet 位置层次；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，那么寻址算法可能进行多达 6 次（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）网络来回通信，因为过期缓存条目只有在没有查到数据（upon misses）的时候才能发现 (假设元数据 tablet 没有被频繁的移动）。尽管 tablet 的位置信息是存放在内存里的，所以不需访问 GFS，但是，通常我们会通过预取 tablet 地址来进一步的减少访问开销：无论何时读取元数据表，都会为不止一个 tablet 读取元数据。

在元数据表中还存储了次级信息(alex 注：secondary information)，包括与 tablet 有关的所有事件日志（例如，什么时候一个服务器开始为该 tablet 提供服务）。这些信息有助于排除故障和性能分析。

5.2 Tablet 分配

每个 tablet 一次分配给一个 tablet 服务器。master 服务器记录活跃的 tablet 服务器、当前 tablet 到 tablet 服务器的分配、包括哪些 tablet 还没有被分配。当一个 tablet 还没有被分配、并且有一个 tablet 服务器有足够的空闲空间来装载该 tablet 并且可用，master 通过给这个 tablet 服务器发送一个 tablet 装载请求分配该 tablet。

BigTable 使用 Chubby 跟踪记录 tablet 服务器。当一个 tablet 服务器启动时，它在一个指定的 Chubby 目录下建立一个有唯一名字的文件，并且获取该文件的独占锁。master 监控着这个目录（服务器目录）y 以便范闲 tablet 服务器。如果 tablet 服务器失去了 Chubby 上的独占锁—比如由于网络断开导致 tablet 服务器丢失 Chubby 会话—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 服务器能够在不招致网络拥堵的情况下检查其是否还持有该锁）。只要该文件还存在，tablet 服务器就会试图重新获得对该独占锁；如果文件不存在了，那么 tablet 服务器就永远不能再提供服务了，它会自行退出（so it kills itself）。只要 tablet 服务器终止（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锁，以便 master 尽快重新分配它的 tablet。

Master 负责探测一个 tablet 服务器何时不再为它的 tablet 提供服务，并且尽快重新分配那些 tablet。master 通过轮询 tablet 服务器锁的状态来探测 tablet 服务器何时不再为 tablet 提供服务。如果一个 tablet 服务器报告它丢失了锁，或者 master 最近几次尝试都无法和该服务器通信，master 就会尝试获取该 tablet 服务器文件的独占锁；如果 master 能够获取独占锁，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，为了保证该 tablet 服务器不能再提供服，master 就删除该 tablet 服务器在 Chubby 上的服务器文件。一旦服务器文件被删除了，master 就把之前分配给该服务器的所有的 tablet 放入未分配的 tablet 集合中。为了确保 Bigtable 集群面对 master 和 Chubby 之间网络问题不那么脆弱，master 在它的 Chubby 会话过期时会主动退出。但是不管怎样，如上所述，master 的故障不会改变现有 tablet 到 tablet 服务器的分配。

当集群管理系统启动了一个 master 之后，master 首先要了解当前 tablet 的分配状态，之后才能够修改它们。master 在启动的时候执行以下步骤：（1）master 在 Chubby 中获取一个唯一的 master 锁，用来阻止并发的 master 实例；（2）master 扫描 Chubby 的服务器目录，获取寻找正在运行的服务器；（3）master 和每一个正在运行的 tablet 服务器通信，搜寻哪些 tablet 已经分配到了 tablet 服务器中；（4）master 服务器扫描元数据表获取

tablet 的集合。只要扫描发现了一个还没有分配的 tablet，master 就将这个 tablet 加入未分配的 tablet 集合，该集合使该 talbet 有机会参与 talbet 分配。
有一种复杂情况是：元数据 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），如果在第三步中没有发现对 root tablet 的分配，master 就把 root tablet 加入到未分配的 tablet 集合中。这个附加操作确保了 root tablet 会被分配。由于 root tablet 包括了所有元数据 tablet 的名字，master 在扫描完 root tablet 以后才了解所有元数据信息。
现存 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、两个现存 tablet 合并组成一个大的 tablet、或者一个现存 tablet 被分割成两个小的 tablet。master 可以跟踪这些改变，因为除了最后一个事件外的两个事件都是由它初始化的。tablet 分割事件需要特殊处理，因为它是由 tablet 服务器初始化的。tablet 服务器通过在元数据表中为新的 tablet 记录信息的方式提交分割操作。在分割操作提交之后 tablet 服务器会通知 master。假如分割操作通知丢失（tablet 服务器或者 master 宕机），master 在请求 tablet 服务器装载已经被分割的 tablet 的时候会探测到一个新的 tablet。由于在元数据 tablet 中发现的 tablet 条目只是列举了 master 请求加载的 tablet 的一部分，tablet 服务器会通知 master 分割信息。

5.3 Tablet 服务

如图 5 所示，tablet 的持久化状态信息保存在 GFS 上。更新操作提交到存储撤销(REDO)记录的提交日志中。在这些更新操作中，最近提交的那些存放在一个叫做 memtable 的排序的缓冲区中；较早的更新存放在一系列 SSTable 中。为了恢复一个 tablet，tablet 服务器在元数据表中读取它的元数据。这些元数据包含组成一个 tablet 的 SSTable 列表和一组还原点（redo points），这些点是指向包含 tablet 数据的任一提交日志的指针。tablet 服务器把 SSTable 的索引读进内存，之后通过应用还原点之后提交的所有更新来重构 memtable。
当写操作到达 tablet 服务器时，tablet 服务器首先要检查这个操作格式是否正确、发送者是否有执行这个改变的权限。权限验证是通过从一个 Chubby 文件里读取具有写权限的操作者列表来进行的（这个文件几乎总会在 Chubby 客户缓存里命中）。有效的修改操作会记录在提交日志里。可以采用组提交方式（alex 注：group commit）来提高大量小的修改操作的吞吐量【13，16】。当一个写操作提交后，它的内容被插入到 memtable 里面。
当读操作到达 tablet 服务器时，它同样会检查良构性和适当的权限。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，引入（incoming）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 的尺寸到达一个临界值的时候，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 并写入 GFS（alex 注：我们称这种 Compaction 行为为 Minor Compaction）。Minor Compaction 过程有两个目的：一是收缩 tablet 服务器内存使用，二是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，引入（incoming）的读写操作仍能继续。

每一次 Minor Compaction 都会创建一个新的 SSTable。如果这个行为未经检查地持续下去，读操作可能需要合并来任意个 SSTable 的更新；反之，我们通过定期在后台执行 Merging Compaction 过程限制这类文件（shijin：SStable）的数量。Merging Compaction 过程读取一些 SSTable 和 memtable 的内容，输出一个新的 SSTable。只要 Merging Compaction 过程完成了，作为输入的 SSTable 和 memtable 就可以丢弃了。
重写所有的 SSTable 到一个新的 SSTable 的 Merging Compaction 过程叫作 Major Compaction。由非 Major Compaction 产生的 SSTable 可以包含特殊的删除条目，这些删除条目能够禁止仍然可用的较早 SSTable 中已删除的数据（STables produced by non-major compactions can contain special deletion entries that suppres s deleted data in older SSTables that are still live）。另一方面，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 并且定期对它们应用 Major Compaction。Major Compaction 机制允许 Bigtable 回收已经删除的数据使用的资源，并且确保已删除的数据及时从系统内消失（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），这对存储敏感数据的服务是非常重要的。
6 优化
上一节描述的实现需要很多优化来达到用户要求的高性能、高可用性和高可靠性的目标。为了突出这些优化，本节描述了部分实现的详细细节。
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。对每个 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单独的局部性群组使读取操作更高效。例如，在 Webtable 表中，网页的元数据（比如语言和检验和 Checksum）可以在一个局部性群组中，网页的内容可以在不同的群组：要读取网页元数据的应用程序没有必要读取整个页面内容。
此外，可以以局部性群组为单位指定一些有用的调整参数。比如，可以把一个局部性群组设定为全部存储在内存中。设定为存入内存的局部性群组的 SSTable 依照惰性加载的策略装载进 tablet 服务器内存。加载完成之后，属于该局部性群组的列族的不用访问硬盘即可读取。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，我们利用这个特性进行元数据表内的列族定位（for the location column family in the metadata table）。
压缩
客户程序可以控制一个局部性群组的 SSTable 是否压缩；如果压缩,用什么格式压缩。用户指定的压缩格式应用到每个 SSTable 的块中（块的大小由局部性群组的调整参数操纵）。尽管为每个分别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），我们却受益于在只读取小部分数据 SSTable 的时候就不必解压整个文件了。许多客户程序使用双步（two-pass）定制压缩模式。第一步采用 Bentley and McIlroy’s 模式[6]，这种模式横跨一个很大窗口压缩常见的长字符串；第二步采用快速压缩算法，即在一个 16KB 数据的小窗口中寻找重复数据。两步压缩都很快，在现代的机器上，编码的速率达到 100-200MB/s，解码的速率达到 400-1000MB/s。
虽然我们在选择压缩算法的时候强调的是速度而不是压缩的空间，但是这种两步压缩模式效果却惊人的好。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。在一次实验中，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，对每个文档我们限制其只有一个版本，而不是存储对我们可用的所有版本。该模式在空间上

达到了 10:1 的压缩比。这比经典的 Gzip 在压缩 HTML 页面时 3:1 或者 4:1 的空间压缩比好的多；这是由于 Webtable 的行的布局方式：从单一主机获取的所有页面紧密存储。利用这个特性，Bentley-McIlroy 算法可以从来自同一个主机的页面里识别大量共享的引用。不仅仅是 Webtable，很多应用程序也通过选择行名来将相似的数据集聚，从而获取较高的压缩率。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。对 tablet 服务器代码而言（to tablet server code），扫描缓存是第一级缓存，其缓存 SSTable 接口返回的键值对；Block 缓存是二级缓存，其缓存从 GFS 读取的 SSTable 块。对于趋向于重复读取相同数据的应用程序来说，扫描缓存非常有效；对于趋向于读取刚读过的数据附近的数据的应用程序来说，Block 缓存很有用（例如，顺序读，或者在一个热点的行的同一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-filte r.html 请务必先认真阅读)

如 5.3 节所述，一个读操作必须读取组成 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。通过 bloom 过滤器我们可以查询一个 SSTable 是否包含了特定行/列对的数据。对于某些应用程序，只使用了少量的 tablet 服务器内粗来存储 Bloom 过滤器，却大幅度减少了读操作需要的磁盘访问次数。Bloom 过滤器的使用也意味着对不存在的行或列的大多数查询不需要访问硬盘。

提交日志的实现

如果我们为每个 tablet 在一个单独的文件里保存提交日志，那么大量的文件会并发地写入 GFS。取决于每个 GFS 服务器底层文件系统实现方案，为了写入不同的物理日志文件，这些写操作会引起大量的询盘操作。另外，由于批量提交(group)中操作的数目趋向于比较少，每个 tablet 拥有单独的日志文件也会降低批量提交优化的效果。为了修正这些问题，我们对每个 tablet 服务器的唯一提交日志追加修改，不同 tablet 的修改操作协同混合到一个相同的物理日志文件中。

在普通操作中使用单个日志提供了重大的性能收益，但是将恢复的工作复杂化了。当一个 tablet 服务器宕机时，它服务的 tablet 将会被移动到大量其它的 tablet 服务器上：每个 tablet 服务器通常都装载少量原来的服务器的 tablet。为了恢复一个 tablet 的状态，新的 tablet 服务器要为该 tablet 重新应用原来的 tablet 服务器写的提交日志中的修改操作。然而，这些 tablet 修改操作被混合在同一个物理日志文件中。一种方法可以是对每一个新的 tablet 服务器读取完整的提交日志文件，然后只重新应用它需要恢复的 tablet 的相关条目。然而，在这种模式下，假如 100 台机器中每台都加载了来自失效的 tablet 服务器的一个单独的 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）。

为了避免重复读取日志文件，我们首先把提交日志的条目按照关键字（table，row name，log sequence number）排序。排序之后，对一个特定 tablet 的修改操作连续存放，因此，随着一次询盘操作之后的顺序读取，修改操作的读取将更高效。为了并行排序，我们将日志文件分割成 64MB 的段，之后在不同的 tablet 服务器对每段进行并行排序。这个排序过

程由 master 来协调，并且当一个 tablet 服务器指出它需要从一些提交日志文件中回复修改时排序被初始化。
在向 GFS 写提交日志时有时引起性能颠簸，原因是多种多样的（比如，写操作相关 GFS 服务器崩溃；或者穿过到达特定组合的三个 GFS 服务器的网络拥塞或者过载）。为了使修改操作免受 GFS 瞬时延迟的影响，每个 tablet 服务器实际上有两个日志写入线程，每个线程写自己的日志文件，并且同一时刻，两个线程只有其中之一是活跃的。如果写入活跃日志文件的效率很低，日志文件写入切换到另外一个线程，在提交日志队列中的修改操作就会由新的活跃日志写入线程写入。日志条目包含序列号，这使得恢复进程可以省略掉由于日志进程切换而造成的重复条目。
Tablet 恢复提速
如果 master 将一个 tablet 从一个 tablet 服务器移到另外一个 tablet 服务器，源 tablet 服务器会对这个 tablet 做一次 Minor Compaction。这个 Compaction 操作减少了 tablet 服务器日志文件中没有压缩的状态的数目，从而减少了恢复的时间。Compaction 完成之后，该 tablet 服务器停止为该 tablet 提供服务。在真正卸载 tablet 之前，tablet 服务器还会再做一次（通常会很快）Minor Compaction，以消除 tablet 服务器日志中第一次 minor compaction 执行过程中产生的未压缩的状态残留。当第二次 minor compaction 完成以后，tablet 就在不需要任何日志条目恢复的情况下被装载到另一个 tablet 服务器上了。
利用不变性
我们在使用 Bigtable 时，除了 SSTable 缓存之外，实际上所有我们产生的 SSTable 都是不变的，因而 Bigtable 系统的其它部分就被简化了。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发操作。memtable 是唯一一个能被读和写操作同时访问的可变数据结构。为了减少对 memtable 进行读操作时的竞争，我们让每个 memtable 表的行写备份 copy-on-write，这样就允许读写操作并行执行。
因为 SSTable 是不变的，所以永久移除已被删除数据的问题就转换成对废弃的 SSTable 进行垃圾收集的问题了。每个 tablet 的 SSTable 都在注册在元数据表中。Master 将废弃的 SSTable 作为对 SSTable 集合的“标记-清除”的垃圾回收而删除【25】，元数据表则保存了 root 的集合。
最后，SSTable 的不变性使得分割 tablet 的操作非常快捷。与为每个子 tablet 生成新的 SSTable 集合相反，我们让子 tablet 共享父 tablet 的 SSTable。
7 性能评估
我们建立一个包括 N 台 tablet 服务器的 Bigtable 集群，通过改变 N 的值来测量 Bigtable 的性能和可扩展性。tablet 服务器配置了 1GB 的内存，数据写入到一个包含 1786 台机器、每台机器有 2 个 400G IDE 硬盘驱动的 GFS 单元上。N 台客户机生成测试 Bigtable 工作负载。（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。）每台机器有主频 2GHZ 的双核 Opteron 处理器，配置了足以容纳所有进程工作集的物理内存，以及一张千兆以太网卡。这些机器都分配到一个两层的、树状的交换网络里，在根节点上的可用总带宽大约 100-200Gbps。所有的机器采用相同的主机设备，因此，任何两台机器间的往返时间都小于 1ms。

Tablet 服务器、master、测试客户机、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 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，同时使用这个机器池的其它作业的进程。
R 是与测试相关的相异 Bigtable 行关键字的数量。我们精心选择 R 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名字从 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：一旦客户程序处理完上一个分配给它的区间，调度程序就把下一个可用的的区间分配给它。这种动态分配的方式有助于减轻客户机上运行的其它进程对性能影响的变化。我们在每个行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，不同行关键字下的字符串也是不同的，因此也就不可能有跨行的压缩。随机写入基准测试采用类似的方法，只是行关键字在写入前先做按 R 取模的 Hash，这样就保证了在整个基准测试期间，写入的工作负载大致均匀地分布在行存储空间内。
序列读的基准测试生成行关键字的方式与序列写相同，只是说不是在行关键字下写入，而是读取行关键字下的字符串（这些字符串由之前序列写基准测试调用写入）。同样的，随机读的基准测试也是附于随机写操作之上的。
扫描基准测试和序列读类似，但是使用的是 BigTable 提供的、从一个行范围内扫描所有值的 API。由于一次 RPC 从一个 tablet 服务器取回了大量的值，因此，使用扫描减少了基准测试执行 RPC 的次数。
随机读（内存）基准测试和随机读类似，但是包含基准测试数据的局部性群组被标记为“in-memory”，因此，要读的数据从 tablet 服务器的内存中即可得到满足，而不需要从 GFS 读取数据。仅对这个测试，我们把每台 tablet 服务器存储的数据从 1GB 减少到 100MB，这样就充裕地满足了 tablet 服务器的可用内存。
图 6 中的两个视图展示了我们在 Bigtable 中读写 1000-byte 时基准测试的性能。图表显示了每个 tablet 服务器每秒钟进行的操作次数；图中的曲线显示了每秒种操作次数的总和。
单个 tablet 服务器的性能
我们首先考虑单个 tablet 服务器的性能。随机读的性能比其它所有操作慢一个数量级或以上（by the order of magnitude or more）。每个随机读操作都涉及通过网络从 GFS 传输 64KB 的 SSTable 到 tablet 服务器，这其中只有一个 1000-byte 的值被用到。Tablet 服务器每秒大约执行 1200 次读操作，也就是说每秒大约从 GFS 读取 75MB（64*1200/1024）的数据。由于网络协议层的消耗、SSTable 解析、以及 BigTable 代码执行，这个传输带宽足以占满 tablet 服务器的 CPU，这个带宽也几乎足以占满我们系统中使用的网络链接。大多数采用这种访问模式的 BigTable 应用程序减小 Block 到一个很小的值，通常取 8KB。
从内存随机读速度快很多，原因是这样的，每个 1000-byte 的读操作都是由 tablet 服务器的本地内存满足的，不需要从 GFS 读取 64KB 的大数据块。
随机和序列写操作性能比随机读要好些，原因是每个 tablet 服务器把写入的内容追加到一个提交日志上，并且采用批量提交的方式，高效地流式写入 GFS。随机写和序列写的性能没有显著的差别，两种方式下对 tablet 服务器的写操作都记录到同一提交日志中。
序列读的性能好于随机读，原因是每次从 GFS 获取的 64KB 的 SSTable 数据块会缓存到 Block 缓存中，这些缓存用来服务下一次 64KB 的读请求。

扫描的性能更高，这是由于 tablet 服务器响应一次客户 RPC 就会返回大量值，所以，RPC 开销基本被大量的数据摊销了。

扩大规模（Scaling）

随着我们将系统中的 tablet 服务器的数目从 1 增加到 500，系统的整体吞吐量得到显著提高，增长了超过 100 倍。比如，随着 tablet 服务器的数量增加 500 倍，随机内存读的性能增加了几乎 300 倍。之所以会有这样的性能提升，是因为这个基准测试的性能瓶颈是单台 tablet 服务器的 CPU。

尽管如此，性能并不是线性增长。在大多数的基准测试中，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。这种下降是由于多台服务器配置中的负载不均衡导致的，通常是由于其它的程序争夺 CPU 和网络。我们的负载均衡算法试图处理这种不均衡，但是基于两个主要原因导致这个算法效果不尽如人意：一个是由于减少 tablet 的移动而导致重新均衡负载能力受限（当 tablet 被移动了，那么短时间内—通常是 1 秒内—这个 tablet 是不可用的），另一个是在基准测试进行中其产生的负载会有波动（alex 注：the load generated by our benchmarks shifts around as the benchmark progresses）。

随机读基准测试在扩大规模后表现最差（整体吞吐量只提升了 100 倍，而服务器的数量却增加了 500 倍）。这种现象的出现是因为（就像上面解释的那样）每读 1000-byte 我们都会在网络上传输一个 64KB 大的块。这样的网络传输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

截止到 2006 年 8 月有 388 个非测试用的 Bigtable 集群运行在各种各样的 Google 机器集群上，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，因此在引人注意的一段时期内比较空闲。通过观察一个由 14 个忙碌集群、8069 个 tablet 服务器组成的群组，我们看到整体的流量超过了每秒 120 万次请求，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。

表 2 提供了一些目前正在使用的表的相关数据。一些表存储的是服务用户的数据，然而另一些存储的则是用于批处理的数据；这些表在总的大小、平均数据项大小、从内存中读取的数据的比例、表的模式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 站点的管理员在他们网站上分析流量模式的服务。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 的浏览量；它还提供了站点追踪报告，比如假定用户之前访问了一个指定页面，购买商品的用户的比例。为了使用这个服务，Web 站点的管理员需要在他们的 Web 页面中嵌入一小段 JavaScript 脚本。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。

我们简略描述 Google Analytics 使用的两个表。原始点击 Raw Click 表（200TB）为每一个终端用户会话维护一行数据。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。这种模式保证了访问同一个 Web 站点的会话是连续的，会话按时间顺序存储。这个表压缩到原来尺寸的 14%。
汇总表 summary（20TB）包含了每个 Web 站点的、各种类型的预定义汇总信息。通过周期性地调度 MapReduce 作业，从 raw click 表中生成 summary 表的数据。每个 MapReduce 作业从 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。这个表的压缩到原有尺寸的 29%。

8.2 Google Earth

Google 运转着一批为用户提供高分辨率地球表面卫星图像的服务，既可以通过基于 Web 的 Google Maps 接口（maps.google.com），也可以通过 Google Earth 可定制客户端软件访问。这些软件产品允许用户浏览地球表面：用户可以在许多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。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，然后被合并到最终的服务数据中。这个表包含了大约 70TB 的数据，因此需要从磁盘读取数据。图像已经被高效压缩过了，因此 Bigtable 压缩被禁用。
Imagery 表的每一行与一个单独的地理区块对应。行都有名称，以确保毗邻的区域存储在了一起。Imagery 表中有一个记录每个区块的数据源的列族。这个列族包含了大量的列：基本上是一个原始数据图像一列。由于每个区块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。
预处理流水线高度依赖运行在 Bigtable 上的 MapReduce 传输数据。在一些 MapReduce 作业中，整个系统中每台 tablet 服务器的处理速度是 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（500GB），但是这个表必须在低延迟下，针对每个数据中心每秒处理几万个查询请求。因此，这个表必须存储在上百个 tablet 服务器上，并且包含 in-memory 的列族。

8.3 个性化查询

个性化查询（www.google.com/psearch）是一个选择性加入服务；这个服务记录用户对各种 Google 属性的查询和点击，比如 Web 查询、图像和新闻。用户可以浏览他们查询的历史，重新访问他们之前的查询和点击；用户也可以请求基于他们历史上的 Google 惯用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，然后分配一个以该 id 命名的行。所有的用户操作都存储在表里。一个单独的列族被用来储存各种类型的行为（比如，有个列族可能是用来存储所有 Web 查询的）。每个数据项将相应的用户动作发生的时间作为 Bigtable 时间戳。个性化查询在 Bigtable 上使用 MapReduce 生成用户配置文件。这些用户配置文件用来个性化当前（live）查询结果。
个性化查询的数据在多个 Bigtable 的集群上备份，以便提高数据可用性同时减少由客户端的距离而造成的延时。个性化查询的开发团队最初在 Bigtable 之上建立了一个客户侧（client side）的复制机制，该机制保证了所有副本的最终一致性。现在的系统则使用了内嵌的复制子系统。
个性化查询存储系统的设计允许其它团队在它们自己的列中加入新的用户级（per-user）数据，因此，很多其他需要按用户配置选项和设置的 Google 属性使用了该系统。在多个团队之间共享表的结果是产生了非比寻常的众多列族。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)，限制任意特定客户在共享表中消耗的空间；该机制也为使用该系统存储用户级信息的产品团体提供了隔离。

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，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 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 partitio ns，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。随着我们在这些问题中得到更多经验，我们通过修改各种协议来解决（address）这些问题。比如，我们在 RPC 机制中加入了检验和 Checksum。我们通过移除系统的其他部分针对另一部分作出的假设来解决这些问题。例如，我们不再假设一个给定的 Chubby 操作只返回固定错误码集合中的一个值。

我们学到的另外一个教训是，我们明白了在彻底了解一个新特性如何使用之前，延迟添加这个新特性是非常重要的。比如，我们最初计划在我们的 API 中支持通用目的的事务处理。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 上已经运行了很多实际应用，我们已经能够检查它们的真实需求；然后发现大多数应用程序只是需要单行事务处理。在人们需要分布式的事务处理的地方，最重要的使用时用来维护二级索引，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，但是更高效（特别是在更新遍布上百行或者更多数据的时候），而且与我们积极“跨数据中心”备份模式交互的很好。

我们从支持 Bigtable 中得到的一个实际的经验就是，适当的系统级监控是非常重要的（比如，既监控 Bigtable 自身，也监控使用 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 的例子，它详细记录了代表 RPC 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 数据结构上的锁的竞争、在提交 Bigtable 修改操作时对 GFS 的写入非常慢的问题、以及在元数据表的 tablet 不可用时，元数据表无法访问的问题。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。这可以允许我们追踪所有集群，发现它们的大小、检查运行的我们软件的版本、他们接收的流量，以及检查是否有类似于意外高延迟等问题。

我们得到的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，代码以难以预料的方式演变的现实，我们发现清晰的设计和编码给维护和调试带来的巨大帮助。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：maste 周期性地和 tablet 服务器签订租约，tablet 服务器在租约过期自动退出。不幸的是，这个协议在网络问题面前大大降低系统的可用性，并且对 master 服务器恢复时间很敏感。我们多次重新设计这个协议，直到它表现优异。然而，最终的协议太复杂，并且依赖一些 Chubby 很少被其他应用程序运用的特性的行为。我们发现我们花费了过量的时间调试一些古怪的边角问题(obscure corner cases)，不仅在 Bigtable 代码中，也在 Chubby 代码中。最后，我们废弃了这个协议，转向了一个新的简单的协议，该协议仅仅依赖最广泛使用 Chubby 的特性。

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 重叠，因为它也提供了诸如分布式协议、锁、分布式块存储以及分布式 B-tree 存储。在有重叠部分的实例中，看来 Boxwood 组件的定位要比相应 Google 低。Boxwood 项目的目标是为诸如文件系统、数据库等高级服务提供基础架构，然而 Bigtable 的目标是直接为希望存储数据的客户程序提供支持。

许多近期的项目已经处理了很多难题，例如在广域网上提供了分布式存储或者高级服务，通常是“Internet 规模”的。这其中包括了分布式的 Hash 表方面的工作，这项工作由一些诸如 CAN【29】、Chord【32】、Tapestry【37】和 Pastry【30】的项目发起。这些系统的强调的关注点不是由于 Bigtable 出现的，比如高度变化的带宽、不可信的参与者、频繁的更改配置等；去中心化和拜占庭式容错(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入，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 的目标。

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 表提供的键值对模型有很大的局限性。键值对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。我们选择的模型比简单的键值对丰富的多，它支持稀疏的、半结构化的数据。尽管如此，它也足够简单，可以标榜为高效普通文件的代表（it lends itself to a very efficient flat-file representation）；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了并行的数据库系统，能够存储海量的数据。Oracle 的实时应用集群数据库 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 的 DB2 并行版本【4】基于一种类似于 Bigtable 的、无共享的架构（a shared-nothing architecture）【33】。每个 DB2 的服务器都负责处理存储在一个本地关系型数据库中的表中的行的一个子集。这两种产品都提供了一个带有事务功能的完整的关系模型。

Bigtable 的局部性群组认识到了类似于压缩和磁盘读取方面性能的收益，这是从其他系统观察到的，这些系统以基于列而不是行的存储方式组织数据，这种系统包括 C-Store【1，34】、商业产品例如 Sybase IQ【15，36】、SenSage【31】、KDB+【22】，以及 MonetDB/X100【38】的 ColumnDM 存储层。另外一种在普通文件中进行垂直和水平分区并获得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 级缓存优化【2】。

Bigtable 采用 memtable 和 SSTable 存储对 tablet 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缓冲在内存中，读取操作必须从内存和磁盘中合并数据。

C-Store 和 Bigtable 共享很多特点：两个系统都采用无共享架构，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，并且提供一种机制将数据从一种格式转换为另一种。两个系统在 API 上有很大的不同：C-Store 操作更像关系型数据库，而 Bigtable 提供跟低层次的读写接口，并且设计用来支持每台服务器每秒数千次

操作。C-Store 也是个“读性能优化的关系型数据库管理系统”，然而 Bigtable 对密集型读写应用提供了很好的性能。

Bigtable 的负载均衡器也必须解决无共享数据库面对的一些类似的负载和内存均衡问题（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副本的问题，同一份数据可能由于视图或索引的原因以替换的形式表现出来；（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，而不是试图动态的决断；（3）我们的不用执行复杂的查询或者对其进行优化。

11 结论

我们已经讲述完了 Bigtable，Google 中一个存储结构化数据的分布式系统。Bigtable 的集群从 2005 年 4 月开始已经投入产业使用了，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 8 月，有超过 60 个项目使用 Bigtable。我们的用户对 Bigtable 实现提供的高性能和高可用性很满意，随着时间的推移，当资源需求改变时，他们可以通过简单的多增加机器来扩展集群的容量。

考虑到 Bigtable 的编程接口并不常见，一个有趣的问题是：我们的用户适应它有多难？新的使用者有时不太确定如何最好地使用 Bigtable 接口，特别是如果他们已经习惯于支持通用目的的事务处理的关系型数据库。然而，许多 Google 产品都成功的使用了 Bigtable 的事实证明，我们的设计在实践中行之有效。

我们现在正在实现对 Bigtable 加入一些新的特性，比如支持二级索引，以及多 master 副本的、跨数据中心复制的 Bigtable 的基础架构。我们也已经开始将 Bigtable 部署为服务供其它的产品团队使用，这样个人团队就不需要维护他们自己的 Bigtable 集群了。随着我们服务集群的扩大，我们需要在 Bigtable 系统内部处理更多的资源共享问题了【3，5 】。

最后，我们发现，建设 Google 自己的存储解决方案有很多重大优势。通过为 Bigtable 设计我们自己的数据模型，我们获得了很大的灵活性。另外，我们对 Bigtable 实现，以及 Bigtable 依赖的其它的 Google 的基础组件的控制，意味着我们在系统出现瓶颈或效率低下的情况时，可以清除这些问题。