Bigtable 翻译版本

2009-06-08 09:58

1.http://my.donews.com/eraera/2006/09/26/swogzstwtqdnwlfrzgsljctkjsbrtuiumxzj/

2.http://my.donews.com/eraera/2006/09/27/gvzwfjvzagmxymaynvemdgwgfjgzicnxnwps/

3.http://my.donews.com/eraera/2006/09/28/feahrnvxnoquabudxnkxuvbbvdbkkflvfftt/

大表(Bigtable):结构化数据的分布存储系统

http://labs.google.com/papers/bigtable-osdi06.pdf
{中是译者评论,程序除外}

{本文的翻译可能有不准确的地方,详细资料请参考原文.}
摘要
bigtable 是设计来分布存储大规模结构化数据的，从设计上它可以扩展到上 2 ^50字节，分布存储在几千个普通服务器上． G oogle 的很多项目使用 B T 来存储数据，包括网页查询，google earth 和 google 金融．这些应用程序对 B T 的要求各不相同：数据大小（从 URL 到网页到卫星图象）不同，反应速度不同（从后端的大批处理到实时数据服务）．对于不同的要求，B T 都成功的提供了灵活高效的服务．在本文中，我们将描述 B T 的数据模型．这个数据模型让用户动态的控制数据的分布和结构．我们还将描述 B T 的设计和实现．

１．介绍

在过去两年半里，我们设计，实现并部署了 B T． B T 是用来分布存储和管理结构化数据的． B T 的设计使它能够管理2^50 bytes(petabytes)数据，并可以部署到上千台机器上． B T 完成了以下目标：应用广泛，可扩展，高性能和高可用性（high availability）．包括 google analytics, google finance, orkut, personalized search, writely 和 google earth 在内的60多个项目都使用 BT.这些应用对 B T 的要求各不相同，有的需要高吞吐量的批处理，有的需要快速反应给用户数据．它们使用的 B T 集群也各不相同，有的只有几台机器，有的有上千台，能够存储2^40字节(terabytes)数据．

B T 在很多地方和数据库很类似：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］有可扩展性和高性能，但是 B T 的界面不同． B T 不支持完全的关系数据模型；而是为客户提供了简单的数据模型，让客户来动态控制数据的分布和格式**{就是只存储字串，格式由客户来解释}**，并允许客户推断底层存储数据的局部性**｛以提高访问速度｝**．数据下标是行和列的名字，数据本身可以是任何字串． B T 的数据是字串，没有解释**｛类型等｝**．客户会在把各种结构或者半结构化的数据串行化**｛比如说日期串｝**到数据中．通过仔细选择数据表示，客户可以控制数据的局部化．最后，可以使用 B T 模式来控制数据是放在内存里还是在硬盘上．**｛就是说用模式，你可以把数据放在离应用最近的地方．毕竟程序在一个时间只用到一块数据．在体系结构里，就是：locality, locality, locality｝**

第二节描述数据模型细节．第三节关于客户ＡＰＩ概述．第四节简介ＢＴ依赖的 google 框架．第五节描述ＢＴ的实现关键部分．第6节叙述提高ＢＴ性能的一些调整．第7节提供ＢＴ性能的数据．在第8节，我们提供ＢＴ的几个使用例子，第9节是经验教训．在第10节，我们列出相关研究．最后是我们的结论．

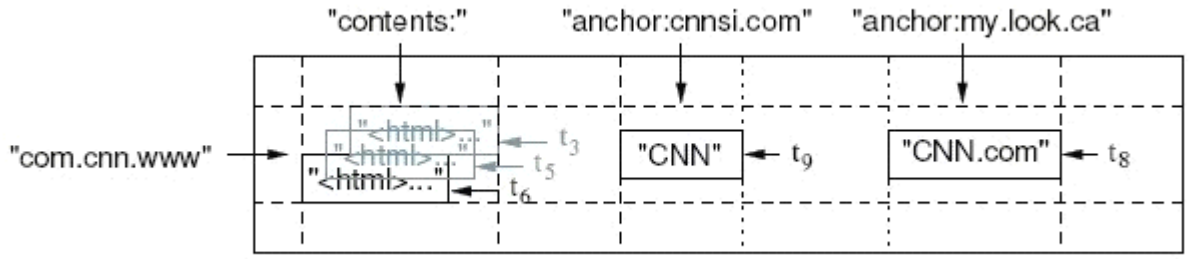
２．数据模型

ＢＴ是一个稀疏的，长期存储的｛存在硬盘上｝，多维度的，排序的映射表．这张表的索引是行关键字，列关键字和时间戳．每个值是一个不解释的字符数组．**｛数据都是字符串，没类型，客户要解释就自力更生吧｝**.

(row:string, column:string,time:int64)->string **{能编程序的都能读懂，不翻译了}**

//彼岸翻译的第二节

我们仔细查看过好些类似 bigtable 的系统之后定下了这个数据模型。举一个具体例子（它促使我们做出某些设计决定）， 比如我们想要存储大量网页及相关信息，以用于很多不同的项目；我们姑且叫它 Webtable。在 Webtable 里，我们将用 URL 作为行关键字，用网页的某些属性作为列名，把网页内容存在 contents:列中并用获取该网页的时间戳作为标识，如图一所示。



图一：一个存储 Web 网页的范例列表片断。行名是一个反向 URL｛即 com.cnn.www｝。contents 列族｛原文用 family，译为族，详见**列族**｝存放网页内容，anchor 列族存放引用该网页的锚链接文本。CNN 的主页被 Sports Illustrater｛即所谓 SI，CNN 的王牌体育节目｝和 MY-look 的主页引用，因此该行包含了名叫“anchor:cnnsi.com”和 “anchhor:my.look.ca”的列。每个锚链接只有一个版本｛由时间戳标识，如 t9，t8｝；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字符串（目前支持最多64KB，多数情况下10－100字节足够了）。在一个行关键字下的每一个读写操作都是原子操作（不管读写这一行里多少个不同列），这是一个设计决定，这样在对同一行进行并发操作时，用户对于系统行为更容易理解和掌控。

Bigtable 通过行关键字的字典序来维护数据。一张表可以动态划分成多个连续行。连续行在这里叫做“子表”｛tablet｝，是数据分布和负载均衡的单位。这样一来，读较少的连续行就比较有效率，通常只需要较少机器之间的通信即可。用户可以利用这个属性来选择行关键字，

从而达到较好数据访问地域性｛locality｝。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组织成连续行。具体来说，可以把 maps.google.com/index.html 中的数据存放在关键字 com.google.maps/index.html 下。按照相同或属性相近的域名来存放网页可以让基于主机和基于域名的分析更加有效。

列族

一组列关键字组成了“列族”，这是访问控制的基本单位。同一列族下存放的所有数据通常都是同一类型（同一列族下的数据可压缩在一起）。列族必须先创建，然后在能在其中的列关键字下存放数据；列族创建后，族中任何一个列关键字均可使用。我们希望，一张表中的不同列族不能太多（最多几百个），并且列族在运作中绝少改变。作为对比，一张表可以有无限列。

列关键字用如下语法命名：*列族：限定词*。 列族名必须是看得懂｛printable｝的字串，而限定词可以是任意字符串。比如，Webtable 可以有个列族叫 language，存放撰写网页的语言。我们在 language 列族中只用一个列关键字，用来存放每个网页的语言标识符。该表的另一个有用的列族是 anchor；给列族的每一个列关键字代表一个锚链接，如图一所示。而这里的限定词则是引用该网页的站点名；表中一个表项存放的是链接文本。

访问控制，磁盘使用统计，内存使用统计，均可在列族这个层面进行。在 Webtable 举例中，我们可以用这些控制来管理不同应用：有的应用添加新的基本数据，有的读取基本数据并创建引申的列族，有的则只能浏览数据（甚至可能因为隐私权原因不能浏览所有数据）。

时间戳

Bigtable 表中每一个表项都可以包含同一数据的多个版本，由时间戳来索引。Bigtable 的时间戳是64位整型。可以由 Bigtable 来赋值，表示准确到毫秒的“实时”；或者由用户应用程序来赋值。需要避免冲突的应用程序必须自己产生具有唯一性的时间戳。不同版本的表项内容按时间戳倒序排列，即最新的排在前面。

为了简化对于不同数据版本的数据的管理，我们对每一个列族支持两个设定，以便于 Bigtable 对表项的版本自动进行垃圾清除。用户可以指明只保留表项的最后 n 个版本，或者只保留足够新的版本（比如，只保留最近7天的内容）。

在 Webtable 举例中，我们在 contents:列中存放确切爬行一个网页的时间戳。如上所述的垃圾清除机制可以让我们只保留每个网页的最近三个版本。

//我开始翻译3,4节

3.API
BT 的ＡＰＩ提供了建立和删除表和列族的函数．还提供了函数来修改集群，表和列族的元数据，比如说访问权限．

```
// 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);
```
图 2: 写入 Bigtable.

在ＢＴ中，客户应用可以写或者删除值，从每个行中找值，或者遍历一个表中的数据子集.图2的Ｃ＋＋代码是使用 RowMutation 抽象表示来进行一系列的更新（为保证代码精简，没有包括无关的细节）. 调用 Apply 函数，就对 W ebtable 进行了一个原子修改：它为 http://www.cnn.com/增加了一个锚点，并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: 从 Bigtable 读数据.

图3的Ｃ＋＋代码是使用 Scanner 抽象来遍历一个行内的所有锚点．客户可以遍历多个列族．有很多方法可以限制一次扫描中产生的行，列和时间戳．例如，我们可以限制上面的扫描，让它只找到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.

ＢＴ还支持其他一些更复杂的处理数据的功能．首先，ＢＴ支持单行处理．这个功能可以用来对存储在一个行关键字下的数据进行原子的读-修改-写操作．ＢＴ目前不支持跨行关键字的处理，但是它有一个界面，可以用来让客户进行批量的跨行关键字处理操作．其次，ＢＴ允许把每个表项用做整数记数器．最后，ＢＴ支持在服务器的地址空间内执行客户端提供的脚本程序．脚本程序的语言是 google 开发的 Sawzall[28]数据处理语言．目前，我们基于的 Sawzall 的ＡＰＩ还不允许客户脚本程序向ＢＴ内写数据，但是它允许多种形式的数据变换，基于任何表达式的过滤和通过多种操作符的摘要.

ＢＴ可以和 MapReduce[12]一起使用．MapReduce 是 google 开发的大规模并行计算框架．我们为编写了一套外层程序，使ＢＴ可以作为 MapReduce 处理的数据源头和输出结果.

4.建立ＢＴ的基本单元

ＢＴ是建立在其他数个 google 框架单元上的． ＢＴ使用 google 分布式文件系统(GFS)[17]来

存储日志和数据文件**{yeah, right, what else can it use, FAT32?}**．一个ＢＴ集群通常在一个共享的机器池中工作，池中的机器还运行其他的分布式应用**{虽然机器便宜的跟白菜似的，可是一样要运行多个程序，命苦的象小白菜}**，ＢＴ和其他程序共享机器**｛ＢＴ的瓶颈是ＩＯ/内存，可以和 CPU 要求高的程序并存｝**．ＢＴ依赖集群管理系统来安排工作，在共享的机器上管理资源，处理失效机器并监视机器状态**｛典型的 server farm 结构，ＢＴ是上面的应用之一｝**.

ＢＴ内部存储数据的格式是 google SSTable 格式．一个 SSTable 提供一个从关键字到值的映射，关键字和值都可以是任意字符串．映射是排序的，存储的**｛不会因为掉电而丢失｝**，不可改写的．可以进行以下操作：查询和一个关键字相关的值；或者根据给出的关键字范围遍历所有的关键字和值．在内部，每个 SSTable 包含一列数据块（通常每个块的大小是64KB, 但是大小是可以配置的**｛索引大小是16 bits，应该是比较好的一个数｝**）．块索引（存储在 SSTable 的最后）用来定位数据块；当打开 SSTable 的时候，索引被读入内存**｛性能｝**．每次查找都可以用一个硬盘搜索完成**｛根据索引算出数据在哪个道上，一个块应该不会跨两个道，没必要省那么点空间｝**：首先在内存中的索引里进行二分查找找到数据块的位置，然后再从硬盘读去数据块．最佳情况是：整个 SSTable 可以被放在内存里，这样一来就不必访问硬盘了．**｛想的美，前面是谁口口声声说要跟别人共享机器来着？你把内存占满了别人上哪睡去？｝**

ＢＴ还依赖一个高度可用的，存储的分布式数据锁服务 Chubby[8]**｛看你怎么把这个 high performance 给说圆喽｝**．一个 Chubby 服务由5个活的备份｛机器｝构成，其中一个被这些备份选成主备份，并且处理请求．这个服务只有在大多数备份都活着并且互相通信的时候才是活的**｛绕口令？去看原文吧，是在有出错的前提下的冗余算法｝**．当有机器失效的时候，Chubby 使用 Paxos 算法[9,23]来保证备份的一致性**｛这个问题还是比较复杂的，建议去看引文了解一下问题本身｝**．Chubby 提供了一个名字空间，里面包括了目录和小文件｛万变不离其宗｝．每个目录或者文件可以当成一个锁来用，读写文件操作都是原子化的．Chubby 客户端的程序库提供了对 Chubby 文件的一致性缓存**｛究竟是提高性能还是降低性能？如果访问是分布的，就是提高性能｝**．每个 Chubby 客户维护一个和 Chubby 服务的会话．如果一个客户不能在一定时间内更新它的会话，这个会话就过期失效了**｛还是针对大 server farm 里机器失效的频率设计的｝**．当一个会话失效时，其拥有的锁和打开的文件句柄都失效**｛根本设计原则：失效时回到安全状态｝**．Chubby 客户可以在文件和目录上登记回调函数，以获得改变或者会话过期的通知．**｛翻到这里，有没有人闻到 java 的味道了？｝**

ＢＴ使用 Chubby 来做以下几个任务：保证任何时间最多只有一个活跃的主备份；来存储ＢＴ数据的启动位置（参考5.1节）；发现小表（tablet）服务器，并完成 tablet 服务器消亡的善后（5.2节）；存储ＢＴ数据的模式信息（每张表的列信息）；以及存储访问权限列表．如果有相当长的时间 Chubby 不能访问，ＢＴ就也不能访问了**｛任何系统都有其弱点｝**．最近我们在使用11个 Chubby 服务实例的14个ＢＴ集群中度量了这个效果，由于 Chubby 不能访问而导致 BT 中部分数据不能访问的平均百分比是0.0047%,这里 Chubby 不能访问的原因是 Chubby 本身失效或者网络问题．单个集群里，受影响最大的百分比是0.0326%**｛基于文件系统的 Chubby 还是很稳定**

//这里是第二部分.我发现如果一篇文章太大,我的 blog 就打开特别慢,不知是 WORDPRESS 的问题还是什么.所以这里另起一篇.

５．实现

ＢＴ的实现有三个主要组件：客户程序库，一个主服务器和多个子表服务器．针对负载的变化，可以动态的从服务器群中添加（或者去除）子表服务器．主服务器的任务是：给子表服务器指定子表，检测加入或者失效的子表服务器，子表服务器负载均衡，以及对 google 文件系统的文件进行垃圾收集．除此之外，它还处理诸如建立表和列族之类的表模式改变工作．

每个子表服务器管理一个子表集合（通常每个服务器处理数十乃至上千个子表）．子表服务器负责处理对它管理的子表进行的读写操作，当子表变的太大时，服务器会将子表分割．和很多单个主服务器分布式系统［17.21］一样，客户数据不经过主服务器．客户的读写操作是通过直接和子表服务器通信完成的．由于ＢＴ的客户不必通过主服务器获取子表位置信息，大多数客户完全不和主服务器通信．因此，实际使用中主服务器的负载很轻．

一个ＢＴ集群存储多个表．每个表由一些子表组成，每个子表包含一个行域内的所有数据．在起始状态下，一个表只有一个子表．当一个表长大以后，它自动的分割成多个子表，每个子表的缺省大小是100到200MB.

5.1 子表的地址

子表地址信息是存储在一个三层类似Ｂ＋树[10]的结构中的(图4).

图4:子表地址结构

第一层是 Chubby 中的一个文件，它存储根子表的地址．根子表里存储一个特殊的表里的所有子表的地址，地址这个特殊的表是元数据表．每个元数据子表里存储一组用户子表的地址．根子表其实是元数据表里的第一个子表，但是对它的处理比较特殊：根子表永远不会被分割，这样一来保证了子表地址结构不会超过三层．

元数据表里面，每个子表的地址都对应一个行关键字，这个关键字是由子表所在的表的标识符，和子表的最后一行编码而成的．每个元数据行在内存里存储大约1kb 的数据．元数据子表的大小限制是128MB，限制看似不大,不过已经可以让这个三层的地址树足够表示2^34个子表了（如果每个子表存储128MB 数据，一共是2^61字节数据）．

客户程序库缓存了子表地址．如果客户没有一个子表的地址，或者它发现地址不正确，客户就递归的查询子表地址树．如果缓存是空的，那么寻址算法需要三次网络来回通信来寻址，其中包括一次 Chubby 读操作．如果缓存数据过期，那么寻址算法可能最多需要 6 次网络来回通信才能更新数据，因为只有在缓存不命中的时候才能发现数据过期**｛三次通信发现过期，另外三次更新数据｝**（这里的假定是，元数据子表没有频繁的移动）．子表的地址是放在内存里的，所以不必访问 google 文件系统ＧＦＳ，我们通过预取子表地址来进一步的减少了访问开销｛体系结构里的老花招：缓存，预取｝：每次读取子表的元数据的时候，都读取几个子表的元数据**｛为什么不说预取几个子表地址？俩？四个？这里就是有价值的东西了，需要时间去积累经验｝.**

在元数据表中还存储了次要信息，包括每个子表的事件日志（例如，什么时候一个服务器开始服务该子表）．这些信息有助于排错和性能分析**｛一笔代过重要信息，比如都存了什么事件，事件属性是什么等｝.**

5.2子表分配

在任一时刻,一个子表只会分配给一个子表服务器.主服务器知道当前有哪些活跃的子表服务器，还知道哪些子表分配到哪些子表服务器，哪些以及哪些子表没有被分配．当一个子表没有被分配到服务器，同时又有一个服务器的空闲空间足够装载该子表，主服务器就给这个子表服务器材发送一个装载请求，把子表分配给这个服务器．

｛这里是协议描述｝ＢＴ使用 Chubby 来追踪子表服务器．当一个子表服务器启动时，它在一个特定的 Chubby 目录里建立一个有唯一名字的文件，并获取该文件的独占的锁．主服务器监视这个目录**{服务器目录}**，就可以发现新的子表服务器．一个子表服务器如果丧失了对文件的锁，就停止对它的子表们的服务．服务器可能丧失锁的原因很多，例如：网络断开导致服务器丢失了 Chubby 会话（Chubby 提供一种有效的服务，使子表服务器不必通过网络就能查询它是否还拥有文件锁**｛这个比较神，难道是 tablet server 自己只查本地文件，chubby server 来帮它在本地建立文件？要认真看看 chubby 的协议才知道｝**）．如果文件依然存在，子表服务器会试图重新获得对文件的独占锁．如果文件不存在了，那么子表服务器就不能服务了，它就退出．当子表服务器终止时（例如，集群管理系统将子表服务器的机器从集群中移除），它会试图释放文件锁，这样一来主服务器就能更快的把子表分配给其他服务器．

当一个子表服务器不再服务它的子表的时候，主服务器有责任发现问题，并把子表尽快重新分配．主服务器发现问题的方法是定期询问子表服务器的文件锁状态．如果一个子表服务器报告丢失了文件锁，或者过去几次询问都没有反应，主服务器就会试图获取子表服务器的文件锁．如果主服务器能够获取锁，说明 Chubby 是好的，子表服务器或者是死了，或者不能和 Chubby 通信，主服务器就删除子表服务器的文件，以确保子表服务器不再服务子表．一旦一个服务器的文件被删除，主服务器就可以把它的子表都放入未分配的子表集合中．为了保证在主服务器和 Chubby 之间有网络故障的时候ＢＴ仍然可以使用，主服务器的 Chubby 会话一旦过期，主服务器就退出．但是，如前所述，主服务器故障不影响子表到子表服务器的分配．

当一个集群管理系统启动一个主服务器时，它需要发现当前的子表分配状态，然后才能修改分配状态｛设计思想：永远考虑失效＋恢复｝．主服务器执行以下启动步骤：(１)主服务器在 Chubby 中获取唯一的主文件锁，来阻止其他主服务器实例**｛singlton｝**．(2)主服务器材扫描 Chubby 服务器目录,获取当前活跃服务器列表.(3)主服务器和活跃子表服务器通信，获取子表分配状态**｛注意子表分配不是主服务器存储的，保证了失效时主服务器不会成为性能瓶颈｝**.（４）主服务器扫

描元数据表，每次遇到一个没有分配的子表，就加入未分配子表集合，这个子表就可以被分配了.

这里有一个复杂的情况：在元数据表没有被分配之前，是不能扫描元数据表的**{鸡和蛋}**. 因此, 在开始第四步的扫描之前，如果第三步的扫描没有发现根子表没分配，主服务器就把根子表加入未分配的子表集合. 这一附加步骤保证了根子表肯定会被分配. 由于根子表包括了所有元数据子表的名字，主服务器在扫描过根子表以后，就知道了所有的元数据子表.

现存子表集合仅在以下事件中才会改变：一个子表被建立或者删除，两个子表被合并，或者一个子表被分割成两个. 主服务器可以监控所有这些事件，因为前三个事件都是主服务器启动的. 最后一个事件：分割子表，是由子表服务器启动的. 这个事件是特别处理的. 子表服务器在分割完毕时，在元数据表中记录新的子表的信息. 当分割完成时，子表服务器通知主服务器. 如果分割的通知没有到达（两个服务器中间死了一个），主服务器在请求子表服务器装载已经分割的子表的时候，就会发现这个没有通知的分割操作**{设计的时候一定要考虑到错误和失败的恢复. 古人云，未思进,先思退}**. 子表服务器就会重新通知主服务器，因为在元数据表中找到的子表入口只包含要求装载的子表的部分信息**{细节，细节呢？}**.

//今天比较忙,只有这些了.抱歉.

//更新:彼岸翻译了第5章的剩余部分,贴在这里:

5.3 子表服务

图5：子表表示

子表的状态存放在 GFS 里，如图5所示。更新内容提交到存放 redo 记录的提交日志里**{比较绕, 看原文可能清楚点}**。在这些更新中，最近提交的那些存放在内存里一个叫 *memtable* 的有序缓冲里；老一点的更新则存放在一系列 SSTable 里。若要恢复一个子表，子表服务器从 METADATA 表中读取元数据。元数据包括了由一个子表和一系列 redo 点**{redo 怎么翻好？}**组成的 SSTable 列表，这些是指向可能含有该子表数据的提交日志的指针**{烦死定语从句了}**。 该服务器把这些 SSTable 的索引读进内存，并通过重复 redo 点之后提交的更新来重建 memtable。

当一个写操作到达子表服务器时，该服务器检查确信这个操作完整无误，而且发送方有权执行所描述的变换。授权是通过从一个 Chubby 文件里读取具有写权限的操作者列表来进行的（几乎一定会存放在 Chubby 客户缓存里）。合法的变换会写到提交日志里。可以用成组提交来提高大量小变换的吞吐量［13，16］。写操作提交后，写的内容就插入到 memtable 里。

当一个读操作到达子表服务器时，会作类似的完整性和授权检查。合法的读操作在一个由 SSTable 系列和 memtable 合并的视图里执行。由于 SSTable 和 memtable 是字典序的数据结构，合并视图可以很有效地形成。

进来方向的**{incoming}**读写操作在子表分拆和合并时仍能继续。

5.4 紧缩**{compaction}**

在执行写操作时，memtable 的大小不断增加。当 memtable 大小达到一定阈值时，memtable 就会被冻结，然后创建一个新的 memtable，冻结住的 memtable 则被转换成 SSTable 并写到 GFS 里。这种*次要紧缩*过程有两个目的：缩小了子表服务器的内存用度，以及减少了在服务器当机后恢复过程中必须从提交日志里读取的数据量。 进来方向的读写操作在紧缩进行当中仍能继续。

每一个次要紧缩会创建一个新的 SSTable。如果这种行为一直继续没有停止的迹象，读操作可能需要合并来自任意多 SSTable 的更新。相反，我们通过定期在后台执行*合并紧缩*来限定这类文件的数量。合并紧缩读取一些 SSTable 和 memtable 的内容，并写成一个新的 SSTable。一旦紧缩完成，作为输入的这些个 SSTable 和 memtable 就可以扔掉了。

把所有 SSTable 重写成唯一一个 SSTable 的合并紧缩叫作*主要紧缩*。 由非主要紧缩产生的 SSTable 可以含有特殊的删除条目，它们使得老一点但仍活跃的 SSTable 中已删除的数据不再出现。而主要紧缩则产生不包含删除信息或删除数据的 SSTable。Bigtable 在它所有的子表中循环，并且定期对它们执行主要紧缩。这些主要紧缩使得 Bigtable 可以回收已删除数据占有的资源，并且还能保证已删除数据及时从系统里小时，这对存放敏感数据的服务很重要。

6.优化

前面一章描述了ＢＴ的实现，我们还需要很多优化工作来获得用户需要的高性能，高可用性和高可靠性．本章描述实现的一些部分，以强调这些优化．

局部性群组

客户可以将多个列族组合成局部性群族.对每个子表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，将会提高读取效率.例如，Webtable 中的网页元数据（语言和校验和之类的）可以在一个局部性群组，网页内容可以在另外一个群组：如果一个应用要读取元数据，它就没有必要去访问页面内容．

此外，每个群组可以设定不同的调试参数．例如，一个局部性群组可以被设定在内存中．内存中的局部性群组的 SSTable 在被子表服务器装载进内存的时候，使用的装载策略是懒惰型的．一旦属于该局部性群组的列族被装载进内存，再访问它们就不必通过硬盘了**{读不懂？知道机器翻译有多难了吧？人翻译都不行}**．这个特性对于需要频繁访问的小块数据特别有用：在ＢＴ内部，我们用这个特性来访问元数据表中的地址列族．

压缩

客户可以控制是否压缩一个局部性群组的 SSTable.每个 SSTable 块（块的大小由局部性群组的调试参数确定），都会使用用户指定的压缩格式．尽管这样分块压缩｛比全表压缩｝浪费了少量空间，但是在读取 SSTable 的一小部分数据的时候，就不必解压整个文件了**｛那是，你们文件巨大，硬盘又便宜｝**．很多客户使用两遍的订制压缩方式．第一遍是 Bentley and McIlroy’s 方式[6],该方式在一个大的扫描窗口中将常见的长串进行压缩．第二遍是一种快速压缩算法，在一个16KB 的小扫描窗口中寻找重复数据．两个算法都很快，现有机器上压缩速率为100到200MB/s,解压速率是400到1000MB/s.

尽管我们选择压缩算法的重点是速率，而非空间效率，这种两遍的压缩方式空间效率也令人

惊叹{**老大，别吹了，您老是在压字符串哪！你去压压运行代码看看？**}. 例如，在 Webtable, 我们用这种压缩方式来存储网页内容．针对实验的目的，我们对每个文档只存储一个版本，而非存储所有能访问的版本．该模式获得了10比1的压缩率．这比一般的 Gzip 的3比1或者4比1的 HTML 页面压缩率好很多，因为 Webtable 的行是这样安排的：从一个主机获取的页面都存在临近的地方，这种特性让 Bentley-McIlroy 算法可以从一个主机那里来的页面里找到大量的重复内容．不仅是 Webtable,其他很多应用也通过选择行名来将类似内容聚集在一起，因此压缩效果非常的好{**针对数据和程序特性选择的压缩算法**}．当在ＢＴ中存储同一数据的多个版本的时候，压缩效率更高．

使用缓存来提高读取性能

为了提高读操作性能，子表服务机构使用两层缓存．扫描缓存是高层，它缓存子表服务器代码从 SSTable 获取的关键字-值对．块缓存是底层，缓存的是从ＧＦＳ读取的 SSTable 块．对于经常要重复读取同一部分数据的应用程序来说，扫描缓存很有用．对于经常要读取前面刚读过的数据附近的其他数据（例如，顺序读{**性能提升老花招：预取**}，或者在一个热门的行中的同一局部性群组中，随机读取不同的列）的应用程序来说，块缓存很有用**{后面这句比较拗口，是说一个局部性群组里的列都在缓存里，可以随机读取}**.

Bloom 过滤器{需要读参考文献才知道是什么意思．从标题看，bloom 是一种杂凑函数，有相对低的不命中率，所以可以用它来猜测一个关键字对应的存储数据在哪里，从而减少访问硬盘的次数，以提高性能}

如5.3节所述，一个读操作要知道一个子表的所有状态，必须从所有 SSTable 中读取数据．如果这些 SSTable 不在内存，那么就需要多次访问硬盘．我们通过允许客户对特定局部性群组的 SSTable 指定 bloom 过滤器[7]，来降低访问硬盘的次数．使用 bloom 过滤器，我们就可以猜测一个 SSTable 是否可能包含特定行和列的数据．对于某些特定应用程序，使用少量内存来存储 bloom 过滤器换来的，是显著减少的访问磁盘次数．我们使用 bloom 过滤器也使当应用程序要求访问不存在的行或列时，我们不会访问硬盘．

修改日志｛commit-log｝的实现

如果我们把对每个子表的修改日志都另存一个文件的话，就会产生非常多的文件，这些文件会并行的写入ＧＦＳ．根据每个ＧＦＳ底层实现的细节，这些写操作在最终写入实际日志文件的时候，可能造成大量的硬盘寻道动作．此外，由于群组不大，为每个子表建立一个新的日志文件也降低了群组修改的优化程度．为了避免这些问题，我们将对每个子表服务器的修改动作添加到一个修改日志中，将多个子表的修改混合到一个实际日志文件中［18,20］.

使用单个日志显著提高了正常使用中的性能，但是将恢复的工作复杂化了．当一个子表服务器死掉时，它以前服务的子表们将会被移动到很多其他的子表服务器上：它们每个都装载很少的几个子表．要恢复子表的状态，新的子表服务器要按照原来的服务器写的修改日志来重新进行修改．但是，这些修改记录是混合在一个实际日志文件中的．一种方法是把日志文件都读出来，然后只重复需要进行的修改项．但是，用这种方法，假如有100台机器都装载了要恢复的子表，那么这个日志文件要读取100次（每个服务器一次）.

避免这个问题的方法是先把日志按照关键字排序.在排序以后,所有的修改项都是连续的了，

只要一次寻道操作，然后顺序读取．为了并行的排序，我们将日志分割成64MB 的段，并在不同的子表服务器上并行的排序．这个排序工作是由主服务器来协同的，当一个子表服务器表示需要从某些日志文件中开始恢复修改，这个过程就开始了．

有时，向ＧＦＳ中写修改日志文件会导致性能不稳定，原因很多（例如，正在写的时候，一个ＧＦＳ服务器不行了，或者访问某三个ＧＦＳ服务器的网络路由断了或者拥塞）．为了在ＧＦＳ延迟的高峰时还能保证修改顺利进行，每个子表服务器实际上有两个线程：各自写不同的日志文件；两个线程里只有一个活跃．如果一个线程的写操作性能不好，就切换到另外一个线程，修改的记录就写入新的活跃线程的日志文件．每个日志项都有序列号，在恢复的时候，由于线程切换而导致的重复的序列号将被忽略．

加速子表的恢复

如果主服务器将一个子表从一个子表服务器移动到另外一个服务器，第一个子表服务器对子表进行轻度压缩．该压缩减少了子表服务器的日志文件中没有被紧缩的状态，从而减少了恢复时间．压缩完成以后，该服务器就停止服务该子表．然后，在卸载该子表前，该服务器再次进行一次（通常很快）轻度压缩，以消除在前面一次压缩时遗留下来的未紧缩的状态．第二次压缩做完以后，子表就可以被装载到另外一个服务器上，而不必请求从日志中恢复了．

利用不变性{immutability，不可写，可以并行读取}

除了 SSTable 缓存以外，由于所有生成的 SSTable 都是不变的，所以ＢＴ的很多其他部分都变的简单了．例如，当从 SSTable 读的时候，就不必进行同步．这样一来，对行的并行操作就可以非常有效的实现了．内存表是唯一一个被读和写操作同时访问的可变数据结构．为了减少在读操作中对内存表的竞争，内存表是写复制的，这样一来就可以并行进行读写操作．

因为 SSTable 是不变的，因此永久消除被删除的数据的问题，就转换成对过时的 SSTable 进行垃圾收集的问题了．每个子表的 SSTable 们都在元数据表进行注册．主服务器对 SSTable 集合进行标记-扫除的垃圾收集工作[25]，元数据表保存了根 SSTable 集合。

最后，SSTable 的不变性使分裂子表的操作更加快速。我们不必为每个分裂出来的子表建立新的 SSTable 集合，而是让分裂的子表集合共享原来子表的 SSTable 集合。