# Bigtable

<http://blog.csdn.net/opennaive/article/details/7532589>

帮助理解Bigtable和HBase的数据模型《Understanding HBase and Bigtable》

<http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable>

2006年的OSDI有两篇Google的论文，分别是Bigtable和Chubby。Chubby是一个分布式锁服务，基于Paxos算法；Bigtable是一个用于管理结构化数据的分布式存储系统，构建在GFS、Chubby、SSTable等google技术之上。相当多的Google应用使用了Bigtable，比如Google Earth和Google Analytics。

Bigtable是一个为管理大规模结构化数据而设计的分布式存储系统，可以扩展到PB级数据和上千台服务器。很多google的项目使用Bigtable存储数据，这些应用对Bigtable提出了不同的挑战，比如数据规模的要求、延迟的要求。Bigtable能满足这些多变的要求，为这些产品成功地提供了灵活、高性能的存储解决方案。Bigtable的设计理念是构建在廉价的硬件之上，通过软件层面提供自动化容错和线性可扩展性能力。

Bigtable看起来像一个数据库，采用了很多数据库的实现策略。但是Bigtable并不支持完整的关系型数据模型；而是为客户端提供了一种简单的数据模型，客户端可以动态地控制数据的布局和格式，并且利用底层数据存储的局部性特征。Bigtable将数据统统看成无意义的字节串，客户端需要将结构化和非结构化数据串行化再存入Bigtable。

## 数据模型

Bigtable系统由很多表格组成，每个表格包含很多行，每行通过一个主键（row key）唯一标识，每行包含很多列。某一行的某一列构成一个单元（cell），**每个单元包含多个版本的数据**。多个列组织成列族（column family），这样，列名由两个部分组成：（column family，qulifier）。**列族**是Bigtable中访问控制的基本单元，也就是说，访问**权限是在列族这一级别上进行的**。Bigtable中列族在创建表格的时候需要预定义好，个数不可以过多；然而列族包含哪些qulifier是不需要预先定义的，qualifier可以任意多个，适合表示半结构化数据。

Bigtable不是关系型数据库，但是却沿用了很多关系型数据库的术语，像table（表）、row（行）、column（列）等。这容易让读者误入歧途，将其与关系型数据库的概念对应起来，从而难以理解论文。

本质上说，Bigtable是一个键值（key-value）映射。按作者的说法，Bigtable是一个稀疏的，分布式的，持久化的，多维的排序映射。Bigtable有三维，分别是行键（row key）、列键（column key）和时间戳（timestamp），行键和列键都是字节串，时间戳是64位整型；而值是一个字节串。可以用 (row:string, column:string, time:int64)→string 来表示一条键值对记录。行键可以是任意字节串，通常有10-100字节。行的读写都是原子性的。Bigtable按照行键的字典序存储数据。Bigtable的表会根据行键自动划分为片（tablet），片是负载均衡的单元。最初表都只有一个片，但随着表不断增大，片会自动分裂，片的大小控制在100-200MB。行是表的第一级索引，我们可以把该行的列、时间和值看成一个整体，简化为一维键值映射，类似于：

table{

"aaaaa" : {sth.}, //一行

"aaaab" : {sth.},

"xyz" : {sth.},

}

列是第二级索引，每行拥有的列是不受限制的，可以随时增加减少。为了方便管理，列被分为多个列族（column family，是访问控制的单元），一个列族里的列一般存储相同类型的数据。一行的列族很少变化，但是列族里的列可以随意添加删除。列键按照family:qualifier格式命名的。这次我们将列拿出来，将时间和值看成一个整体，把列族当做一层新的索引，类似于：

table{

"aaa" : { //一行

"A" : { //列族A

"foo" : {sth.}, //一列

"bar" : {sth.}

},

"B" : { //列族B

"" : {sth.}

}

},

"bbb" : { //一行

"A" : {

"foo" : {sth.},

},

"B" : {

"" : "ocean"

}

},

// ...

}

时间戳是第三级索引。Bigtable允许保存数据的多个版本，版本区分的依据就是时间戳。时间戳可以由Bigtable赋值，代表数据进入Bigtable的准确时间，也可以由客户端赋值。数据的不同版本按照时间戳降序存储，因此先读到的是最新版本的数据。我们加入时间戳后，就得到了Bigtable的完整数据模型，类似于：

table{

// ...

"aaa" : { //一行

"A:foo" : { //一列

15 : "y", //一个版本

4 : "m"

},

"A:bar" : { //一列

15 : "d",

},

"B:" : { //一列

6 : "w"

3 : "o"

1 : "w"

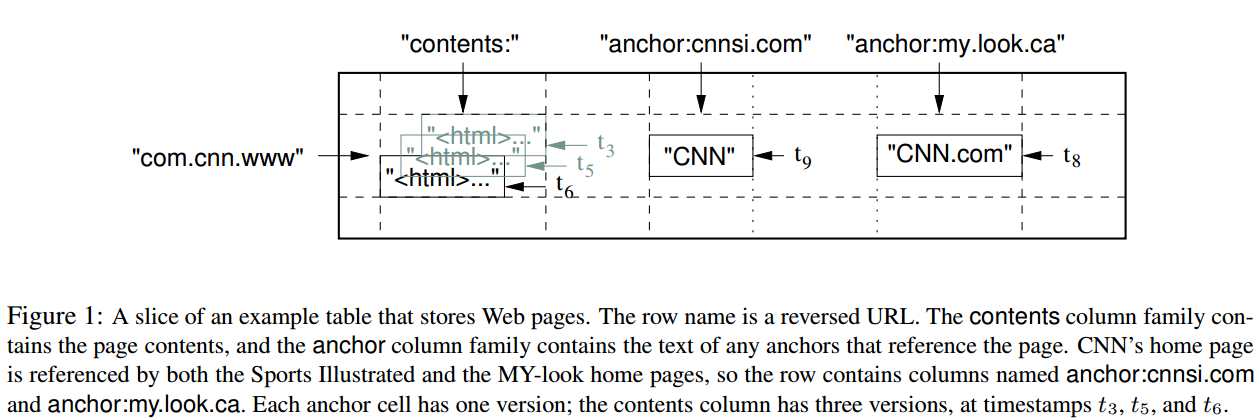
}

},

// ...

}

在查询时，如果只给出行和列，那么返回的是最新版本的数据；如果给出了行、列和时间戳，那么放回的是时间小于或等于时间戳的数据。比如，查询aaa/A:foo，返回的就是最新版本结果“y”；如果查询aaa/A:foo/10，返回的结果是“m”；如果查询的是aaa/A:foo/2，返回的结果是空。

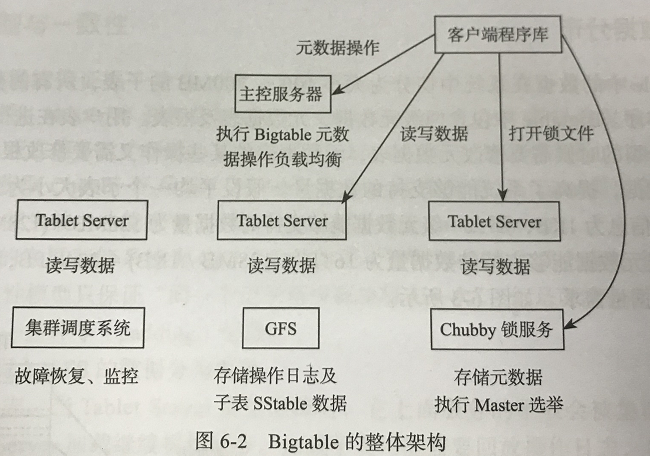


上面是Bigtable论文里给出的例子，表中存储了大量的网页和相关信息。每一行存储一个网页，其反转的URL作为行键，比如maps.google.com/index.html的数据存储在键为com.google.maps/index.html的行里，反转的原因是为了让同一个域名下的子域名网页能聚集在一起。列族“anchor”保存了该网页的引用站点，qualifier是引用站点的名称，而数据是链接文本；列族“content”保存的是网页的内容，这个列族只有一个空列“contents:”。图中“contents:”列下保存了网页的三个版本，比如可以使用com.cnn.www/contents:/t5来找到CNN主页在t5时刻的内容。

## 架构

Bigtable构建在GFS之上，为文件系统增加一层分布式索引成，GFS用来存储日志和数据文件，按照SSTable文件格式存储数据；依赖Google的chubby（分布式锁服务）进行服务器选举及全局信息维护。

SSTable（Sorted Strings Table），是一种不可修改的有序的键值映射，提供了查询、遍历等功能。每个SSTable由一系列的块组成，Bigtable将块默认设置为64KB。在SSTable的尾部存储着块索引，在访问SSTable时，整个索引会被装入内存。Bigtable论文中没有给出SSTable的具体结构。每个片（tablet）在GFS里都是按照SSTable的格式存储的，每个片可能对应多个SSTable。



Bigtable将大表划分为大小在100-200MB的片（tablet），每个片对应一个连续的数据范围。一旦片大小超出了范围就分裂成更小的片，或者合并成更大的片。每个片服务器负责一定量的片，处理对其片的读写请求以及片的分裂和合并。片服务器可以根据负载随时增加或者删除。片服务器并不是真正的在存储数据，数据永远都是存储在GFS中的，片服务器相当于一个连接Bigtable和GFS的代理，客户端的一些数据操作都通过片服务器代理间接访问GFS。

Bigtable主要由三部分组成：client、主控服务器（master）和多个片服务器（tablet server）。

master：管理所有的片服务器，包括分配片给片服务器、指导片服务器实现片的合并、接受来自片服务器的片分裂消息、监控片服务器、在片服务器之间进行负载均衡并实现片服务器故障恢复等。主服务器不存储任何片，不提供任何数据服务，也不提供片的定位信息；

tablet server：实现片的装载/卸出、表格内容的读写、片的合并和分裂。tablet server服务的数据包括操作日志以及每个片上的sstable数据，这些都是存储在GFS中；

client：提供Bigtable到应用程序的接口，应用程序通过客户端程序库对表格的数据单元进行增删改查等操作，客户端通过chubby锁服务获取一些控制信息，但所有表格的数据内容都在客户端与片服务器之间直接传送，因此大多数客户端从来不需要访问主服务器，主服务器的负载一般很轻。

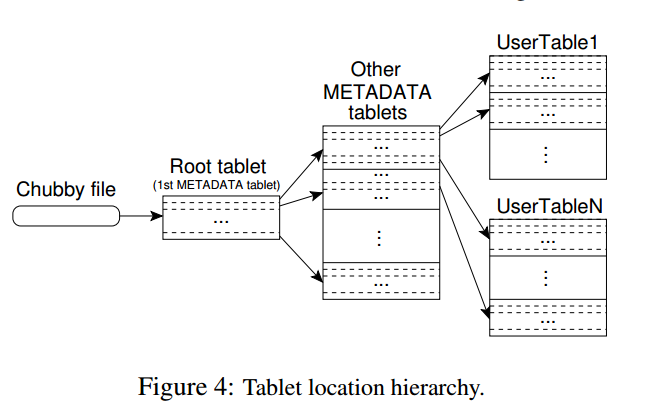
chubby是底层采用paxos为核心算法的分布式锁服务，在Bigtable中提供一下功能：

* 选取并保证同一时间内只有一个主控服务器；
* 存储Bigtable系统引导信息；
* 用于配合主控服务器发现片服务器加入和下线；
* 获取Bigtable表格的schema信息及控制访问信息。

chubby有5个活跃副本，但同时只有一个主副本提供服务，副本之间用paxos算法维持一致性

## 数据分布

Bigtable包含3钟类型的表格：用户表（user table）、元数据表（Meta table）和根表（root table）。其中，用户表存储用户实际数据；元数据表存储用户的元数据，如片位置信息、SSTable及操作日志文件编号、日志回放点等；根表存储元数据表的元数据，根表的元数据，也就是根表的位置信息，又称Bigtable引导信息，存放在chubby系统中。client、master以及tablet server执行过程都需要依赖chubby服务，如果chubby出现故障，Bigtable系统整体不可用。



Bigtable中包含两级元数据，root tablet和other METADATA tablets。用户表进行某些操作，比如片分裂，需要修改元数据表，元数据表的某些操作又需要修改根表。通过两级元数据，提高了系统能够支持的数据量。假设一个片128MB，每个片的元信息为1KB，那么一级元数据能够支持的数据量为：

128MB\*(128MB/1KB)=16TB

两级元数据能够支持的数据量为：

16TB\*(128MB/1KB)=2048PB

可以满足几乎所有的业务需求。

Bigtable使用一个类似B+树的数据结构来存储片的位置信息。

第一层是chubby file。这一层是一个chubby文件，保存着root tablet的位置信息。这个chubby文件属于chubby服务的一部分，一旦chubby不可用，就意味着丢失了root tablet的位置，整个Bigtable不可用。

第二层是root tablet。root tablet其实是元数据表（METADATA tablet）的第一个分片，它保存着元数据表其它片的位置。root tablet很特别，为了保证树的深度不变，root tablet从不分裂。

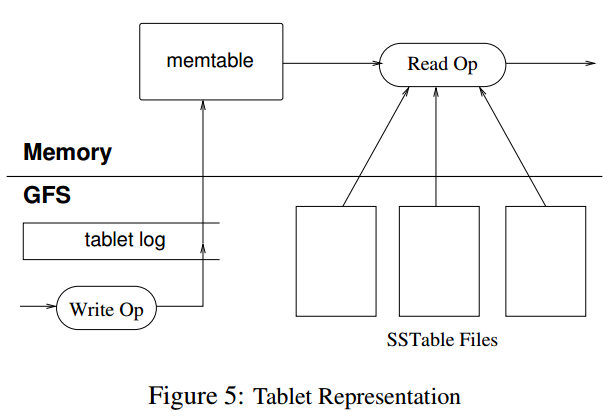
第三层是其它的元数据片，它们和root tablet一起组成完整的元数据表。每个元数据片都包含了许多用户片的位置信息。

可以看出整个定位系统其实只是两部分，一个chubby文件，一个元数据表（root tablet和other METADATA tablets）。注意元数据表虽然特殊，但也仍然服从前文的数据模型，每个分片也都是由专门的片服务器负责，这就是不需要主服务器提供位置信息的原因。客户端会缓存片的位置信息，如果在缓存里找不到一个片的位置信息，就需要查找这个三层结构了。

client查询时，首先从chubby中读取根表的位置，接着从根表读取所需的元数据片的位置，最后可以从元数据片中找到待查询的用户片的位置。为了减少访问开销，客户端采用**缓存（cache）和预取（prefetch）**技术。片的位置信息被缓存在client，client寻址时首先查找cache，一旦cache为空或者cache信息过期，客户端就需要请求片服务器的上一级元数据表获取位置信息，比如用户片cache过期就需要请求元数据表，元数据表cache过期就需要请求根表，根表缓存过期需要读取chubby中的引导信息。可见如果cache为空，最多需要3次请求；如果缓存过期，最多需要6次请求（其中3次请求用来确定信息是否过期，另外3次获取新的地址信息）。预取是在每次访问元数据表时不仅仅读取所需片的元数据，而是读取连续的多个片元数据，这样查找下一个片时就不需要再次访问元数据表。

## 片的存储访问

片的数据最终还是写到GFS里的，片在GFS里的物理形态就是若干个SSTable文件。



当片服务器收到一个写请求，片服务器首先检查请求是否合法。如果合法，先将写请求提交到日志去，然后将数据写入内存中的memtable。memtable相当于SSTable的缓存，当memtable成长到一定规模会被冻结，Bigtable随之创建一个新的memtable，并且将冻结的memtable转换为SSTable格式写入GFS，这个操作称为minor compaction。

当片服务器收到一个读请求，同样要检查请求是否合法。如果合法，这个读操作会查看所有SSTable文件和memtable的合并视图，因为SSTable和memtable本身都是已排序的，所以合并相当快。

每一次minor compaction都会产生一个新的SSTable文件，SSTable文件太多读操作的效率就降低了，所以Bigtable定期执行merging compaction操作，将几个SSTable和memtable合并为一个新的SSTable。BigTable还有个更厉害的叫major compaction，它将所有SSTable合并为一个新的SSTable。

遗憾的是，BigTable作者没有介绍memtable和SSTable的详细数据结构。

## 复制与一致性

Bigtable系统保证强一致性，同一时刻同一个片只能被一台tablet server服务，master将片分配给某个tablet server服务时，需要确保没有其他的table server正在服务这个片。这是通过chubby的互斥锁机制保证的，tablet server启动时需要获取chubby互斥锁，当tablet server出现故障，master需要等到tablet server互斥锁失效才能把它上面的片迁移到其他tablet server。

Bigtable写入GFS的数据分为两种：

* 操作日志。当tablet server出现故障，它上面服务的片会被集群中其他的tablet server加载继续提供服务。加载片可能需要回放操作日志，每条操作日志都有唯一的序号，通过它可以去除重复的操作日志。
* 每个片包含的SSTable数据。如果写入GFS失败可能重试并产生多条重复记录，但是Bigtable只会索引最后一条写入成功的记录。

Bigtable本质上是构建在GFS之上的一层分布式索引，解决了GFS遗留的一致性问题。

## 容错

master对tablet server的监控是通过chubby完成的，tablet server初始化时会从chubby获取一个独占锁，通过这种方式所有tablet server基本信息被保存在chubby中一个称为服务器目录（server directory）的特殊目录中。master通过检测这个目录随时获取最新的tablet server信息，包括目前活跃的tablet server以及每个table server上已分配的片。

对于每个tablet server，master会定期询问其独占锁的状态，如果tablet server的锁丢失或者没有回应，此时可能出现的问题是：chubby出现了问题，或者tablet server出现了问题。对此master首先自己尝试获取这个独占锁，如果失败说明是chubby服务器出现问题，否则说明是tablet server出现问题，master将终止这个tablet server并将其上的片全部迁移到其他tablet server。

片持久化的数据包含两个部分：操作日志及SSTable。tablet server发生故障时某些片的一些更新操作还在内存中，需要通过回放操作日志来恢复。为了提高性能，tablet server没有为它服务的每一个片写一个操作日志，而是把所有它服务的片操作日志混在一起写入GFS，每条日志通过<表格编号、行主键、日志序号>来标识。当某个tablet server宕机后，master将tablet server服务的片分配给其他tablet server。为了减少tablet server从GFS读取日志的数据量，master将选择一些tablet server对日志进行分段排序，排好序后同一个片的操作日志连续存放，tablet server恢复某个片时只需要读取该片对应的操作日志即可。master需要可能选择负载低的tablet server执行排序，并且需要处理排序任务失败的情况。

master启动时需要从chubby中获取一个独占锁，如果master发生故障，master的独占锁将过期，管理程序会自动指定一个新的master服务器，它从chubby成功获取独占锁后可以继续提供服务。

## 负载均衡

片是Bigtable负载均衡的基本单位。tablet server定期向master汇报状态，当发现某个tablet server负载过重时，master会自动对其进行负载均衡，即片迁移。片迁移分为两步：

1. 请求原有的tablet server卸载片；
2. 选择一台负载均衡较低的tablet server加载片。

master发送命令请求原有的tablet server卸载片，原有的tablet server卸载成功后应答master。接着，master才可以安全将该片分配给新的tablet server加载。如果原有的tablet server发生故障，master需要等待原有tablet server加在chubby上的互斥锁过期。片迁移前原有的tablet server会对其执行minor compaction操作，将内存中的更新操作以SSTable文件的形式转储到GFS中，因此负载均衡带来的片迁移在新的tablet server上不需要回放操作日志。

片迁移过程中有短暂的时间需要停服务，为了尽可能减少停服务的时间，Bigtable内部采用两次minor compaction的策略：

1. 原有tablet server对片进行一次minor compaction，操作过程中仍然允许写操作；
2. 停止片的写服务，对片再次执行minor compaction操作。由于第一次minor compaction过程中**写入的数据一般比较少**（意思可能是在这个过程中新到来的写入操作比较少），第二次minor compaction的时间会比较短。

Bigtable负载均衡过程中会停一会服务，负载均衡策略不应当过于激进。负载均衡涉及的因素很多，tablet server通过心跳定时将读、写个数，磁盘，内存负载等信息发送给master，master根据负载计算公式计算出需要迁移的表，然后放入迁移队列中等待执行。

## 分裂与合并

数据不断写入和删除，可能某些片太大，某些片太小，需要执行片合并与分裂操作。

Bigtable每个片的数据分为内存中的 memtable和GFS中的多个 SSTable，由于Bigtable中同一个片只被一台tablet server服务，进行分裂时比较简单。Bigtable上执行分裂操作不需要进行实际的数据拷贝工作，只需要将内存中的索引信息分成两份，比如分裂前片的范围为(起始主键，结束主键]，在内存中将索引分成(起始主键，分裂主键]和(分裂主键，结束主键]两个范围。例如，某个片的(1，10]分裂主键为5，那么分裂后生成的两个片的数据范围为:(1，5]和(5，10]。分裂以后两个片各自写不同的 memtable,等到执行 Compaction操作时再根据分裂后的片范围生成不同的 SSTable,无用的数据自然成为垃圾被回收。

分裂操作由tablet server发起，需要修改元数据，即用户表的分裂需要修改元数据表，元数据表的分裂需要修改根表。分裂操作需要增加一个片，相当于在元数据表中增加一行，通过Bigtable的行事务保证原子性。只要修改元数据表成功，分裂操作就算成功。分裂成功后tablet server将向 master汇报，如果出现 tablet server故障, master可能丢失汇报分裂消息。新的 tablet server加载这个片时将发现它已经分裂，并通知master。

合并操作由master发起，相比分裂操作要更加复杂。由于待合并的两个片可能被不同的tablet server加载，合并的第一步需要迁移其中一个片，以使它们在同一个tablet server上，接着通知tablet server执行片合并。片合并操作具体实现时非常麻烦，需要自行思考。

## 单机存储

Bigtable采用merge-dump存储引擎，数据写入时需要先写操作日志，成功后应用到内存中的MemTable中，写操作日志是往磁盘中的日志文件追加数据，很好的利用了磁盘设备的特性。当内存中的MemTable达到一定大小，需要将MemTable转存（dump）到磁盘中生成SSTable文件。由于数据同时存在MemTable和可能多个SSTable中，读取操作需要按从旧到新的时间顺序合并SSTable和内存中的MemTable。数据在SSTable中连续存放，因此可以同时满足随机读取和顺序读取两种需求。为了防止磁盘中的SSTable文件过多，需要定时将多个SSTable通过compaction过程合并为一个SSTable，从而减少后续读操作需要读取的文件个数。一般情况下，如果写操作比较少，总是能够使得对每一份数据同时只存在一个SSTable和一个MemTable，也就是说，随机读取和顺序读取都只需要访问一次磁盘，这对于线上服务基本都是成立的。

插入、删除、更新、增加等操作在merge-dump引擎中都看成一回事，除了最早生成的SSTable外，SSTable只记录操作，而不是最终的结果，需要等到读取（随机或顺序）时才合并得到的最终结果。

## 垃圾回收

compaction操作会生成新的SSTable，原有的SSTable成为垃圾需要被回收。每个片正在引用的SSTable文件保存在元数据中。master定期执行垃圾回收任务，这是一个标记删除（mark-and-sweep）过程。首先扫描GFS获取所有的SSTable文件，接着扫描根表和元数据表获取正在使用的SSTable文件，如果GFS中的SSTable没有被任何一个片使用，说明可以被回收。需要注意的是，由于tablet server执行campaction操作生成一个全新的SSTable与修改元数据这两个操作不是原子的，垃圾回收需要避免删除刚刚生成但还没有记录到数据中的SSTable文件。一种简单的做法是垃圾回收只删除至少一段时间没有被使用的SSTable文件，比如1个小时没有使用的。

## Bigtable和GFS之间的关系

集群包括主服务器和片服务器，主服务器负责将片分配给片服务器，而具体的数据服务则全权由片服务器负责。但是不要误以为片服务器真的存储了数据（除了内存中memtable的数据），数据的真实位置只有GFS才知道，主服务器将片分配给片服务器的意思应该是，片服务器获取了片的所有SSTable文件名，片服务器通过一些索引机制可以知道所需要的数据在哪个SSTable文件，然后从GFS中读取SSTable文件的数据，这个SSTable文件可能分布在好几台chunkserver上。

## 元数据表的的结构

元数据表（METADATA table）是一张特殊的表，它被用于数据的定位以及一些元数据服务，不可谓不重要。但是Bigtable论文里只给出了少量线索，而对表的具体结构没有说明。这里我试图根据论文的一些线索，猜测一下表的结构。首先列出论文中的线索：

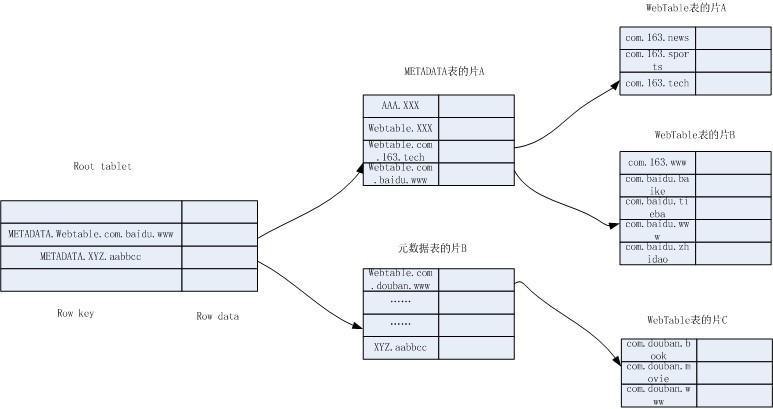
1. The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row.
2. Each METADATA row stores approximately 1KB of data in memory（因为访问量比较大，元数据表是放在内存里的，这个优化在论文的locality groups中提到）.This feature（将locality group放到内存中的特性） is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the METADATA table.
3. We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet(such as when a server begins serving it).

第一条线索，元数据表的行键是由片所属表名的id和片最后一行编码而成，所以每个片在元数据表中占据一条记录（一行），而且行键既包含了其所属表的信息也包含了其所拥有的行的范围。譬如采取最简单的编码方式，元数据表的行键等于strcat(表名，片最后一行的行键)。

第二点线索，除了知道元数据表的地址部分是常驻内存以外，还可以发现元数据表有一个列族称为location，我们已经知道元数据表每一行代表一个片，那么为什么需要一个列族来存储地址呢？因为每个片都可能由多个SSTable文件组成，列族可以用来存储任意多个SSTable文件的位置。一个合理的假设就是每个SSTable文件的位置信息占据一列，列名为location:filename。当然不一定非得用列键存储完整文件名，更大的可能性是把SSTable文件名存在值里。获取了文件名就可以向GFS索要数据了。

第三个线索告诉我们元数据表不止存储位置信息，也就是说列族不止location，这些数据暂时不是咱们关心的。

通过以上信息画了一个简化的Bigtable结构图：



结构图以Webtable表为例，表中存储了网易、百度和豆瓣的几个网页。当我们想查找百度贴吧昨天的网页内容，可以向Bigtable发出查询Webtable表的(com.baidu.tieba, contents:, yesterday)。

假设客户端没有该缓存，那么Bigtable访问root tablet的片服务器，希望得到该网页所属的片的位置信息在哪个元数据片中。使用METADATA.Webtable.com.baidu.tieba为行键在root tablet中查找，定位到最后一个比它大的是METADATA.Webtable.com.baidu.www，于是确定需要的就是元数据表的片A。访问片A的片服务器，继续查找Webtable.com.baidu.tieba，定位到Webtable.com.baidu.www是比它大的，确定需要的是Webtable表的片B。访问片B的片服务器，获得数据。

这里需要注意的是，每个片实际都由若干SSTable文件和memtable组成，而且这些SSTable和memtable都是已排序的。这就导致查找片B时，可能需要将所有SSTable和memtable都查找一遍；另外客户端应该不会直接从元数据表获得SSTable的文件名，而只是获得片属于片服务器的信息，通过片服务器为代理访问SSTable。