BigTable论文学习笔记

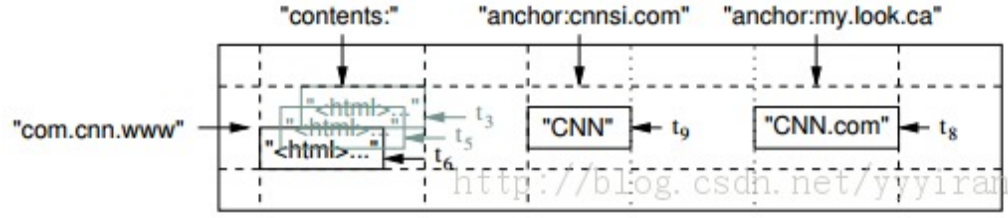
Bigtable为Google设计的一个分布式结构化数据存储系统，用来处理Google的海量数据。Google内包括Web索引、Google地球等项目都在使用Bigtable存储数据。尽管这些应用需求差异很大，但是Bigtable还是提供了一个灵活的、高性能的解决方案。

一、简介

* 设计目标：可靠的处理PB级别的数据，适用性广泛、可扩展、高性能和高可用性。
* 很多方面Bigtable和数据库类似，其也使用了数据库很多实现策略，但是Bigtable提供了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，但为用户提供了一种简单的数据模型，用户可以动态控制数据的分布和格式

二、数据模型

* Bigtable是一个稀疏的、分布式的、持久化存储的多维排序Map（Key=>Value）。Map的索引（Key）是行关键字、列关键字和时间戳，Map的值（Value）都是未解析的Byte数组：
  - Key (row:string, col:string, time:int64) => Value (string)
* 下图是Bigtable存储网页信息的一个例子：



  - 行："com.cn.www"为网页的URL
  - 列："contents:"为网页的文档内容，"anchor:"为网页的锚链接文本（anchor:为列族，包含2列cnnsi.com和my.look.ca）
  - 时间戳：t3、t5、t6、t8和t9均为时间戳

1、行

* 行和列关键字都为字符串类型，目前支持最大64KB，但一般10~100个字节就足够了
* 对同一个行关键字的读写操作都是原子的，这里类似Mysql的行锁，锁粒度并没有达到列级别
* Bigtable通过行关键字的字典序来组织数据，表中每行都可以动态分区。每个分区叫做一个"Tablet"，故Tablet是数据分布和负载均衡调整的最小单位。这样做的好处是读取行中很少几列数据的效率很高，而且可以有效的利用数据的位置相关性（局部性原理）

2、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，存放在同一列族的数据通常都属于同一类型。
* 一张表列族不能太多（最多几百个），且很少改变，但列却可以有无限多
* 列关键字的命名语法：列族:限定词。
* 访问控制、磁盘和内存的使用统计都是在列族层面进行的

3、时间戳

* 在Bigtable中，表的每个数据项都可包含同一数据的不同版本，不同版本通过时间戳来

索引（64位整型，可精确到毫秒）
* 为了减轻各版本数据的管理负担，每个列族有2个设置参数，可通过这2个参数可以对废弃版本数据进行自动垃圾收集，用户可以指定只保存最后n个版本数据
三、API
* 在表操作方面，提供建表、删表、建列族、删列族，以及修改集群、表和列族元数据（如访问权限等）等基本API。一个例子：

```
// 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.
* 在数据操作方面，提供写入、删除、读取、遍历等基础API。一个例子：

```
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.
* 根据具体需求，Bigtable还开发出支持一些其他的特性，比如：1 支持单行上的事务处理，2 允许把数据项做整数计数器 3 允许用户在Bigtable服务器地址空间上执行脚本程序
四、基础构件
* Bigtable是建立在其他几个Google基础构件上的，有GFS、SSTable、Chubby等
1、基础存储相关
* Bigtable使用GFS存储日志文件和数据文件，集群通常运行在共享机器池（cloud）中，依靠集群管理系统做任务调度、资源管理和机器监控等
2、数据文件格式相关
* Bigtable的内部储存文件为Google SSTable格式的，SSTable是一个持久化、排序的、不可更改的Map结构
* 从内部看，SSTable是一系列的数据块，并通过块索引定位，块索引在打开SSTable时加载到内存中，用于快速查找到指定的数据块
3、分布式同步相关
* Bigtable还依赖一个高可用的、序列化的分布式锁服务组件Chubby（类zookeeper）。
* Chubby服务维护5个活动副本，其中一个选为Master并处理请求，并通过Paxos算法

来保证副本一致性。另外Chubby提供一个名字空间，提供对Chubby文件的一致性缓存等
* Bigtable使用Chubby来完成几个任务，比如：1 确保任意时间只有一个活动Master副本，2 存储数据的自引导指令位置，3 查找Tablet服务器信息等 4 存储访问控制列表等
五、实现
* Bigtable包括3个主要的组件：链接到用户程序的库，1个Master服务器和多个Tablet服务器。Tablet服务器可根据工作负载动态增减
* Master服务器：为Tablet服务器分配Tablets，对Tablet服务器进行负载均衡，检测Tablet服务器的增减等
* Tablet服务器：管理一个Tablets集合（十到上千个Tablet），并负责它们的读写操作。与一般Single-Master类型的分布式存储系统类似，客户端可直接和Tablet服务器通信并进行读写，故Master的负载并不大
* 初始情况下，每个表只含一个Tablet，随着表数据的增长，它会被自动分割成多个Tablet，使得每个Table一般为100~200MB
1、Tablet的位置信息
* 我们使用三层的、类B+树的结构存储Tablet的位置信息，如下图所示：

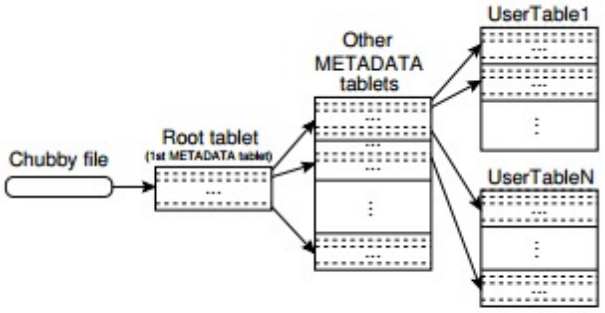


Figure 4: Tablet location hierarchy.

* 第一层为存储于Chubby中的Root Tablet位置信息。Root Tablet包含一个MetaData表，MetaData表每个Tablet包含一个用户Tablet集合
* 在MetaData表内，每个Tablet的位置信息都存储在一个行关键字下，这个行关键字由Tablet所在表的标识符和最后一行编码而成
* MetaData表每一行都存储约1KB内存数据，即在一个128MB的MetaData表中，采用这种3层存储结构，可标识2^32个Tablet地址
* 用户程序使用的库会缓存Tablet的位置信息，如果某个Tablet位置信息没有缓存或缓存失效，那么客户端会在树状存储结构中递归查询。故通常会通过预取Tablet地址来减少访问开销
2、Tablet的分配
* 在任何时刻，一个Tablet只能分配给一个Tablet服务器，这个由Master来控制分配（一个Tablet没分配，而一个Tablet服务器用足够空闲空间，则Master会发给该Tablet服务器装载请求）
* Bigtable通过Chubby跟踪Tablet服务器的状态。当Tablet服务器启动时，会在Chubby注册文件节点并获得其独占锁，当Tablet服务器失效或关闭时，会释放这个独占锁

* 当Tablet服务器不提供服务时，Master会通过轮询Chubby上Tablet服务器文件锁的状态检查出来，确认后会删除其在Chubby注册的节点，使其不再提供服务。最后Master会重新分配这个Tablet服务器上的Tablet到其他未分配的Tablet集合内
* 当集群管理系统启动一个Master服务器之后，这个Master会执行以下步骤：
  - 1 从Chubby获取一个唯一的Master锁，保证Chubby只有一个Master实例
  - 2 扫描Chubby上的Tablet文件锁目录，获取当前运行的Tablet服务器列表
  - 3 和所有Tablet服务器通信，获取每个Tablet服务器上的Tablet分配信息
  - 4 扫描MetaData表获取所有Tablet集合，如果发现有还没分配的Tablet，就会将其加入未分配Tablet集合等待分配

3、Tablet的服务

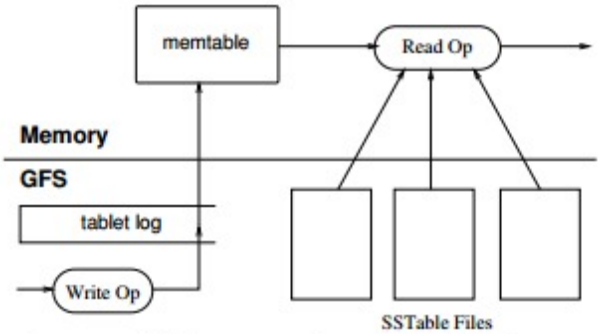


Figure 5: Tablet Representation

* 如图所示，Tablet的持久化状态信息保存在GFS上。更新操作会提交Redo日志，更新操作分2类：
  - 最近提交的更新操作会存放在一个排序缓存中，称为memtable
  - 较早提交的更新操作会存放在SSTable中，落地在GFS上
* Tablet的恢复：Tablet服务器从MetaData中读取这个Tablet的元数据，元数据里面就包含了组成这个Tablet的SSTable和RedoPoint，然后通过重复RedoPoint之后的日志记录来重建（类似Mysql的binlog）
* 对Tablet服务器写操作：首先检查操作格式正确性和权限（从Chubby拉取权限列表）。之后有效的写记录会提交日志，也支持批量提交，最后写入的内容插入memtable内
* 对Tablet服务器读操作：也首先检查格式和权限，之后有效的读操作在一系列SSTable和memtable合并的视图内执行（都按字典序排序，可高效生成合并视图）

4、Compactions

* 当memtable增大达到一个门限值时，这个memtable会转换为SSTable并创建新的memtable，这个过程称为Minor Compaction。
* Minor Compaction过程为了减少Tablet服务器使用的内存，以及在灾难恢复时减少从提交日志读取的数据量
* 如果Minor Compaction过程不断进行下去，SStable数量会过多而影响读操作合并多个SSTable，所以Bigtable会定期合并SStable文件来限制其数量，这个过程称为Major Compaction。
* 除此之外，Major Compaction过程生产的新SStable不会包含已删除的数据，帮助

Bigtable来回收已删除的资源
六、优化
1、局部性群族
* 用户可将多个列族组合成一个局部性群族，Tablet中每个局部性群族都会生产一个SSTable，将通常不会一起访问的分割成不同局部性群族，可以提高读取操作的效率
* 此外，可以局部性群族为单位专门设定一些调优参数，如是否存储于内存等
2、压缩
* 用户可以控制一个局部性群族的SSTable是否压缩
* 很多用户使用”两遍可定制“的压缩方式：第一遍采用Bentley and McIlroy（大扫描窗口内常见长字符串压缩），第二遍采用快速压缩算法（小扫描窗口内重复数据），这种方式压缩速度达到100~200MB/s，解压速度达到400~1000MB/s，空间压缩比达到10:1
3、缓存
* Tablet服务器使用二级缓存策略来提高读操作性能。两级的缓存针对性不同：
* 第一级缓存为扫描缓存：缓存Tablet服务器通过SSTable接口获取的Key-Value对（时间局部性）
* 第二季缓存为块缓存：缓存从GFS读取的SSTable块（空间局部性）
4、布隆过滤器
* 一个读操作必须读取构成Tablet状态的所有SSTable数据，故如果这些SSTable不在内存便需多次访问磁盘
* 我们允许用户使用一个Bloom过滤器来查询SStable是否包含指定的行和列数据，付出少量Bloom过滤器内存存储代价，换来显著减少访问磁盘次数
5、Commit日志实现
* 如果每个Tablet操作的Commit日志单独写一个文件，会导致日志文件数过多，写入GFS会产生大量的磁盘Seek操作而产生负面影响
* 优化：设置为每个Tablet服务器写一个公共的日志文件，里面混合了各个Tablet的修改日志。
* 这个优化显著提高普通操作性能，却让恢复工作复杂化。当一台Tablet服务器挂了，需要将其上面的tablet均匀恢复到其他Tablet服务器，则其他服务器都得读取完整的Commit日志。为了避免多次读Commit日志，我们将日志按关键字排序(table, row, log_seq)，让同一个Tablet的操作日志连续存放
6、Tablet恢复提速
* Master转移Tablet时，源Tablet服务器会对这个Tablet做一次Minor Compaction，减少Tablet服务器日志文件没有归并的记录，从而减少了恢复时间
7、利用不变性
* 在使用Bigtable时，除了SSTable缓存外其他部分产生的SSTable都是不变的，可以利用这个不变性对系统简化

七、性能评估
* 实验设计：N台Tablet服务器集群（N=1、50、250、500...），每台Tablet服务器1G内存，数据写入一个含1786台机器的GFS集群。使用N台Client产生工作负载，这些机器都连入一个两层树状网络，根节点带宽约100~200Gbps。

* 一共有6组基准测试：序列写、随机写、序列读、随机读、随机读（内存）和扫描，测试结果如下图所示：

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

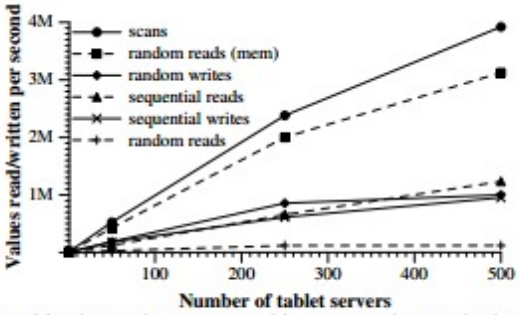


Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.

测试均为读/写1000字节value的数据，图1显示了1/50/250/500台Tablet服务器，每台服务器的每秒操作次数，图2曲线显示随着Tablet服务器数量增加，所有服务器的每秒操作次数总和

* 对于图1单个Tablet服务器性能维度，有下面几个特点：
  - 随机读性能最慢，这是因为每个随机读操作都要通过网络从GFS集群拉回64KB（1块）数据到Tablet服务器
  - 随机读（内存）性能很快，因为这些读操作的数据都从Tablet服务器的内存读取
  - 序列读性能好于随机读，因为每次从GFS取出64KB数据，这些数据会缓存，序列读很多落到同个块上而减少GFS读取次数
  - 写操作比读操作高，因为写操作实质上为Tablet服务器直接把写入内容追加到Commit日志文件尾部（随机写和序列写性能相近的原因），最后再采用批量提交的方式写入GFS
  - 扫描的性能最高，因为Client的每一次RPC调用都会返回大量value数据，抵消了RPC调用消耗

* 对于图2Tablet服务器集群性能维度，有下面几个特点：
  - 随着Tablet服务器的增加，系统整体吞吐量有了梦幻般的增加，之所以会有这样的性能提升，主要是因为基准测试的瓶颈是单台Tablet服务器的CPU
  - 尽管如此，性能的增加也不是线性的，这是由于多台Tablet服务器间负载不均衡造成的
  - 随机读的性能提升最小，还是由于每个1000字节value的读操作都会导致一个64KB块的网络传输，消耗了网络的共享带宽

八、实际应用

* 截止到2006年，Google内部一共运行了388个非测试的Bigtable集群，约24500台Tablet服务器，这些应用以及应用数据大致如下：

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.

* 如上图所示，可以了解到Google分析，Google地图，Google个性化查询等应用的Bigtable使用情况

九、经验教训

* 很多类型的错误都会导致大型分布式系统受损，而不仅仅是网络中断等“常规”错误。我们使用修改协议来解决这些问题（容错性），如在RPC机制中加入Checksum等
* 需要在彻底了解一个新特性如何使用后，再决定添加这个新特性是否是重要的。
* 系统级的监控对Bigtable非常重要，能有效跟踪集群状态、检查引发集群高时延的潜在因素等
* 简单的设计和编码给维护和调试带来了巨大的好处

来自：http://blog.csdn.net/yyyiran/article/details/12918921