Bigtable 论文阅读笔记

姓名	学号
米家龙	18342075

> 论文选择为 Bigtable: A Distributed Storage System for Structured Data

- 总结

简介

Google 在2003年到2006年期间发布的三篇关于大规模数据储存和分析的论文，分别是：

- MapReduce: Simplified Data Processing on Large Clusters
- Bigtable: A Distributed Storage System for Structured Data
- The Google File System

这“三驾马车”的存在，使得 google 在很长一段时间内担当着工业时代的大数据时代的领头羊的身份。

研究背景

在当时，由于技术和硬件性能的局限，大多数企业对大量的数据储存需求并没有很高的要求，但对于基于爬虫获取数据的 google 搜索引擎来说，互联网的快速发展带来的网页数量大规模膨胀，爬虫获取到的数据也随之大量增加，于是，gfs 顺理成章的诞生了；但由于解耦合的设计思路，gfs 只负责**数据的存储**，对于数据的具体内容不关心，因此无法提供**基于内容**的类似数据库的服务

为了满足内容相关的需求，google 后续开发了 Bigtable 作为数据库，为上层服务提供内容的各种功能

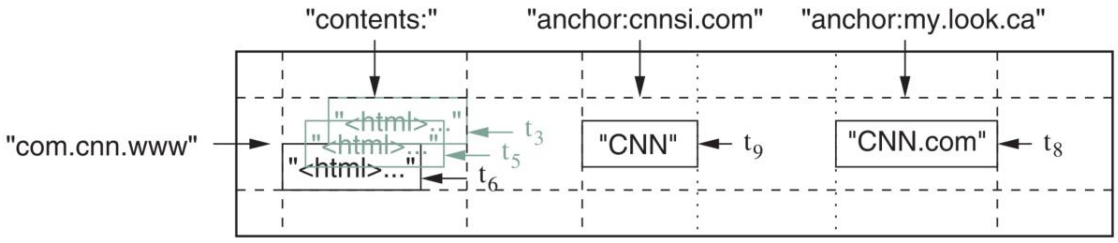
Bigtable 依托于 Google File System 、Chubby 和 SSTable 而诞生，主要用于解决内部产品在对数据储存的容量和响应时延需求的差异化，尽可能追求大容量数据的同时减少查询时间。

方法介绍

数据模型

一个 Bigtable 是一个稀疏的、分布式的、持久化的、多维的有序 Map ， Map 中含有若干个 table（表），表中的 cell（数据单元）通过行、列的键以及时间戳进行索引，形式如下：

$$(row : \text{string},\ column : \text{string},\ time : \text{int64}) \rightarrow string$$



行 Row

行标识有如下特性：

1. 行键是上限为 64kb 的字符串
2. 任意单行的读写都是原子性的
3. 能够按照关键字进行**字典序**排序
4. Bigtable 能根据行的关键字动态划分数据成相邻的 *tablet（子表）* ，并储存在不同的服务器上，从而实现负载均衡

列族 Column Family

传统的关系型数据库按照列为粒度进行权限管理，但在拥有较多列的情况下，管理难度很大，因此 Bigtable 作出了改进，使用 Column Family（列族）进行管理，具有如下特性：

1. 列键的形式为 $(family:\text{qualifier})$
2. 用户使用前必须声明有哪些列族，声明完成后才能在对应的列族中创建对应的列
3. 同一个列族中储存的数据通常属于同一类型
4. Bigtable 会对同属一个列族的数据进行合并压缩
5. Bigtable 允许用户以列族为单位，为其他用户设定数据访问权限

时间戳 timestamp

为了避免数据更新带来的版本冲突，Bigtable 按照时间关系给同行同列的数据赋予一个时间戳，并且：

1. 时间戳以64位整数形式储存
2. 时间戳可以通过客户端应用程序设置，也可以由时间的决定
3. 默认降序排列
4. 两种版本回收机制：
    1. 保留最新的几个版本
    2. 保留一定时间内的所有版本

架构

一个完整的 Bigtable 集群由两种节点组成：

- Master 主服务器
- Tablet Server 子表服务器

Master

主服务器不储存子表，也不提供子表的定位信息（这一点和 gfs 有所不同），主要负责：

1. 子表服务器的分配组成、负载均衡
2. 监控子表服务器的状态，包括加入和退出集群事件
3. 子表服务器租约超时无回应时安排新的子表服务器替代
4. 子表服务器中的子表过大时，重新划分并分配子表服务器
5. 管理 *表（Table）* 、列族的创建删除等修改操作

Tablet Server

子表服务器负责管理主服务器指定的子表（tablet），处理对应的读写请求，并在子表过大的时候进行切分

为了减小主服务器负载，数据请求不会经过主服务器而是直接到子表服务器上

并且子表服务器并不会直接储存数据，实际储存数据的其实是 gfs ，子表服务器只是进行分片管理

其他

SSTable

SSTable 是 Bigtable 内部使用的文件格式，有如下特点：

1. 该文件提供了不变、有序的键值映射，键值都是任意字符串
2. SSTable 的默认块大小为64kb，这些块的索引存放在文件末尾
3. 每个子表能够对应多个 SSTable

Chubby

Chubby 是 google 设计的锁服务，在 Bigtable 中起到非常关键的作用，以至于一旦服务失效，整个 Bigtable 都无法继续工作

Chubby 主要应用在：

1. 子表服务器的定位
2. 子表服务器的分配
3. 表的权限控制等

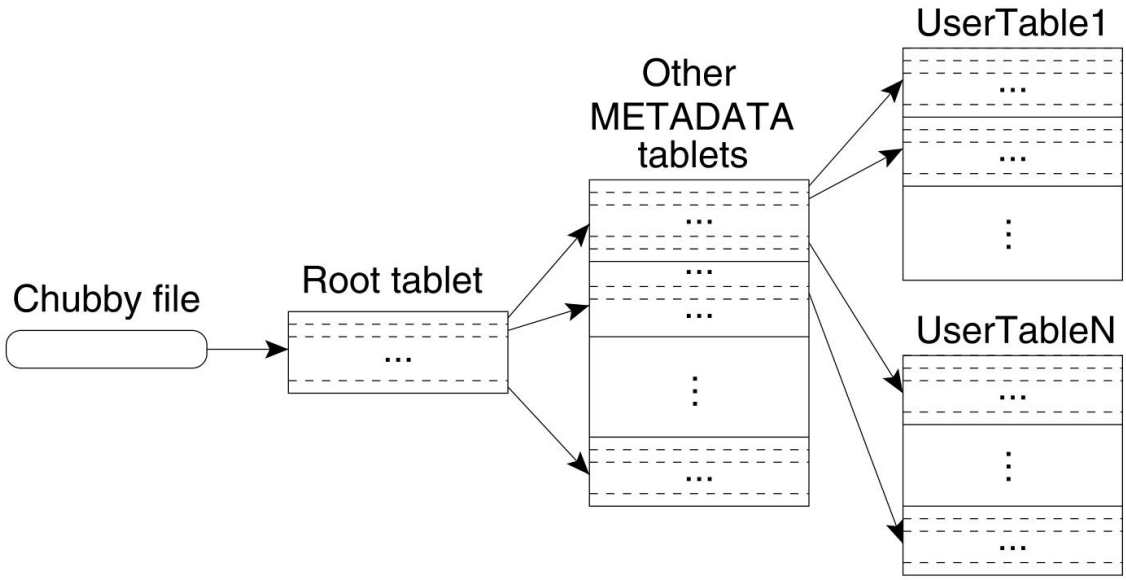
相关操作

子表定位

在 Bigtable 中，子表之间会按照一个三层的结构进行组织，具体为：

1. 第一层：Chubby 的一个文件中保存着 `根子表（Root Tablet）` 的位置
2. 第二层：根子表是 `元数据子表（METADATA Tablets）` 中的第一个子表，储存着元数据表中其他所有子表的位置，并且该子表不会随着表的增长而分割
3. 第三层：元数据子表保存其他所有表中子表的信息

结构图为



因此，在查找子表的过程如下：

- 客户端检查自己的缓存，看是否存在对应子表位置缓存：
  - 若缓存**有效**：客户端就直接根据对应的位置访问子表信息
  - 若缓存**错误**：客户端会递归向上一层的子表服务器进行查询
  - 若缓存**为空**：
    1. 客户端从 Chubby file 处获取根子表的位置
    2. 从根子表中查询对应元数据子表的位置
    3. 再在元数据子表中找到需要的子表的位置
    4. 完成完整的询问

子表分配

主服务器通过 Chubby 监控子表服务器的状态（加入和离开集群的事件）与子表的分配（包括子表被分配给哪个子表服务器以及哪些子表还未被分配）

监控子表服务器的状态是通过**互斥锁**来实现的：

- **加入集群**事件：
  - 当子表服务器启动并和 Chubby 建立连接后，会在 Chubby 的特定目录下建立一个文件并获取到互斥锁；
  - 主服务器通过**周期性**尝试获取这些互斥锁来确认子表服务器是否正常工作
- **退出集群**事件：
  - 子表服务器和 Chubby 断开连接后，互斥锁会失去，但只要子表服务器上的数据和 Chubby 上对应的文件存在，子表服务器还会不断请求这个互斥锁；
  - 当主服务器获取到这个互斥锁之后，便会删除对应的 Chubby 文件，最终使得子表服务器停止
  - 子表服务器主动停止服务时，会主动释放互斥锁

主服务器和 Chubby 断开连接之后，当前主服务器会主动关闭，集群则会重新选择一个新的主服务器作为替代用于恢复，过程如下：

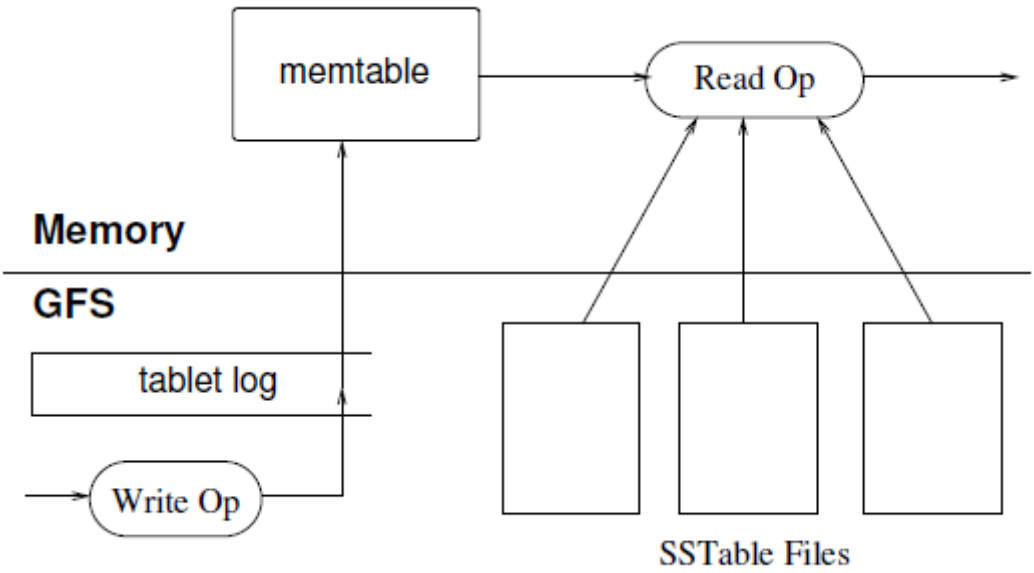
1. 新的主服务器启动后，会获取 Chubby 上对应 Master 的互斥锁，避免其他的主服务器启动
2. 主服务器通过扫描 Chubby 特定目录并尝试获取互斥锁来获得当前有效的子表服务器
3. 主服务器向子表服务器询问被分配的子表，并表明主服务器身份，使得子表服务器的后续通信能够发往新的主服务器
4. 统计元数据子表中未分配的子表，并完成分配

子表读写

一个子表组成为：

- 若干持久化在 gfs 中的 SSTable 文件
- 一个位于内存中的 memtable
- 一份 tablet log 日志文件

读写操作示意图：



读操作

读操作步骤：

1. 子表服务器进行权限检查，相关用户权限列表可以从 Chubby 的特定文件中获取
2. 尝试从 memtable 中获取最新的数据
3. 如果无法从 memtable 中获取数据，那么将从 SSTable 中进行数据查找

写操作

写操作步骤：

1. 先通过写日志（Write-Ahead Log, WAL）的方式把变更记录到 table log 中
2. 将插入的数据放入对应的 memtable 中，memtable 需要保持内部数据的有序性
3. 已经持久化的数据则会作为 SSTable 持久化保存在 gfs 中

载入

子表服务器在载入子表时，需要进行如下操作：

1. 从元数据表中获取子表对应的 SSTable 文件以及 table log 日志
2. 通过 table log 日志中的条目逐步恢复子表的 memtable

子表压缩

> 为什么需要子表压缩?

Minor Compaction

每进行一次写操作后，产生的新条目都会以 COW(Copy On Write) 的方式放入 memtable 中

当 memtable 中的条目数量较大，达到预定的阈值后，Bigtable 会将新的请求写入到另一个 memtable 中，同时将旧的 memtable 写入到新的 SSTable 文件中

对于已有的 SSTable 中的旧数据，Bigtable 不会进行溢出

这一系列操作被称为 Minor Compaction

Merging Compaction

由于 Minor Compaction 会不断增加 SSTable 的数量，极大程度地影响文件维护的性能与效率，因此需要周期性地对 SSTable 和 memtable 进行合并操作，将若干 SSTable 和 memtable 中的数据原样合并成一个 SSTable

Major Compaction

Major Compaction 是一种特殊的 Merging Compaction 操作，在这个操作中，Bigtable 会：

- 将若干 SSTable 合并为一个 SSTable
- 将 SSTable 中因后续变更/删除操作而被标记成无效的条目移除

优化

操作日志

一个还是多个日志文件

Bigtable 使用了 WAL 的做法来保证高可用性，但如果为不同的子表使用不同的日志，会出现大量日志同时写入的情况，从而使得底层硬盘的寻址时间大大增加；因此，Bigtable 使子服务器把接收到的所有子表写操作都写入到**同一个** table log 中

重复读取

当一个子表服务器下线时，其负责的子表可能会被重新分配到其他的子表服务器上，当其他子表服务器在恢复 memtable 的过程中会重复读取上一个子表服务器产生的 table log ，从而消耗不必要的时间和性能

为了解决这个问题，Bigtable 进行了如下操作：

- 子表服务器在读取 table log 前会向主服务器发送信号，主服务器会对原 table log 进行排序操作
- 原 table log 会按照 64MB 的大小进行切分，每个块并发地按照 $(table, row, name, log\ sequence\ number)$ 的规则进行排序
- 完成排序后，子服务器只需要读取对应的部分即可

读缓存与布隆过滤器

Bigtable 使用的储存方式是 LSM 树，通过牺牲读取性能换取写操作性能的增加，因此需要通过其他方式来确保读性能

读缓存

Bigtable 的读缓存分为两层：

- Block Cache：Block Cache 缓存 gfs 中读取的 SSTable 的块，用于提高客户端在读取某个数据附近的其他数据的效率
- Scan Cache：Scan Cache 在 Block Cache 上，缓存 SSTable 返回给子服务器的键值对，在客户端重复读取时提高效率

布隆过滤器

Bigtable 允许用户开启布隆过滤器机制，通过消耗一定的内存保存为 SSTable 构建的布隆过滤器，以便在哭护短检索记录时快速排除无关 SSTable ，减少需要搜索的 SSTable 数

Locality Group

Bigtable 允许客户端为列族指定 Locality Group ，并以此为基础指定文件的储存格式和压缩方式

在进行 Compaction 操作时，Bigtable 给每个子表中的每个 Locality Group 生成独立的 SSTable 文件，因此可以将较少同时访问的列族放入不同的 Locality Group 中，从而提高查询效率

SSTable 压缩

Bigtable 对 SSTable 的压缩不是对整个文件进行压缩，而是针对 SSTable 中的块进行的，这使得用户在读取数据时只需要对特定的 SSTable 的块进行解压

这种压缩方式虽然提高了读操作的效率，但是会导致压缩率的下降，是一种取舍

总结

Bigtable 使用了排序大表的方式来设计数据库，使用 LSM 树将随机写操作转化为顺序写，并利用 gfs 提供数据冗余，为后续各种数据库的设计提供了思路

Chubby 的使用也证明了分布式协调组件的重要性，引导了后续相关组件的设计

上述两点是 Bigtable 对于工业界最有影响的两点，为现代工业分布式数据库的设计提供了方向与思路，阅读 Bigtable 论文能对分布式的历史与理解更加了解，受益匪浅