# 概述

分布式表格系统对外提供表格模型，每个表格由很多行组成，通过主键唯一标识，每一行包含很多列。整个表格在系统中全局有序。

Google Bigtable是分布式表格系统的始祖，它采用双层结构，底层采用GFS作为持久化存储层。GFS+Bigtable双层架构是一种里程碑式的架构，其他系统，包括Microsoft分布式存储系统Windows Azure Storage以及开源的Hadoop系统，均为其模仿者。Bigtable的问题在于对外接口不够丰富，因此，Google后续开发了两套系统，一套是Megastore，构建在Bigtable之上，提供更加丰富的使用接口；另一套是Spanner，支持跨多个数据中心的数据库事务。

# 分类

## Bigtable

Bigtable是Google开发的寄语GFS和Chubby的分布式表格系统。Google的很多数据，包括Web索引、卫星凸显数据等在内的海量结构化和半结构化数据，都存储在Bigtable中。与Google的其他系统一样，Bigtable的设计理念是构建在廉价的硬件之上的，通过软件层面提供自动化容错和线性可扩展性能力。

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，每行又包含很多列（Column）。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。整体上看，Bigtable是一个分布式多维映射表，如下所示：

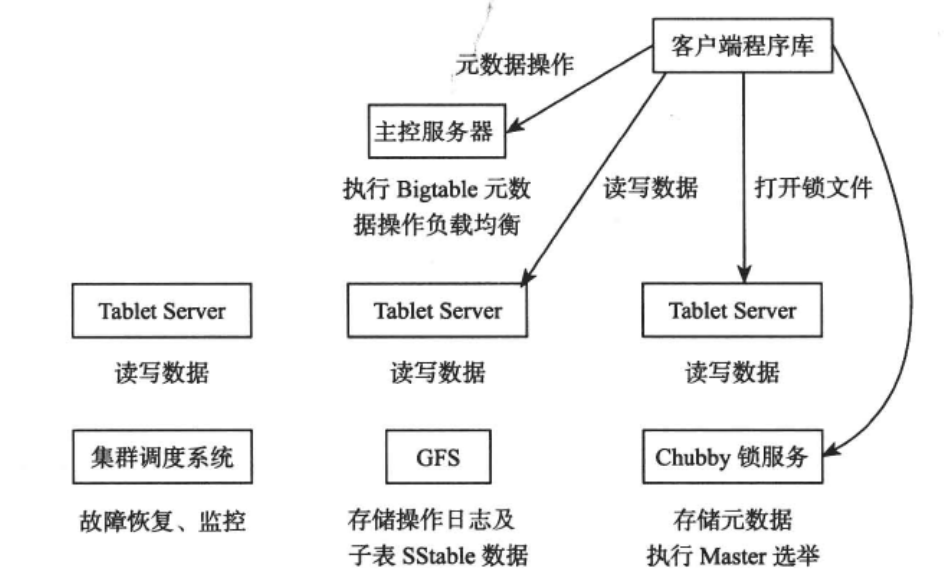


注：与Redis相比虽然都是存储非结构化数据，但是Redis没有主键的概念，是纯粹的key-value存储。

### 架构

Bigtable构建在GFS之上，为文件系统增加一层分布式索引层。另外，Bigtable依赖Google的Chubby（即分布式锁服务）进行服务器选举及全局信息维护。

如图所示，Bigtable将大表划分为大小在100~200MB的子表（tablet），每个子表对应一个连续的数据范围。Bigtable主要由三个部分组成：客户端程序库（Client）、一个主控服务器（Master）和多个子表服务器（Tablet Server）。



1. 客户端程序库（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，但是所有表格的数据内容都在客户端与子表服务器之间直接传送；
2. 主控服务器（Master）：管理所有的子表服务器，包括分配子表给子表服务器，指导子表服务器实现子表的合并，接受来自子表服务器的子表分裂消息，监控子表服务器，在子表服务器之间进行负载均衡并实现子表服务器的故障恢复等。
3. 子表服务器（Tablet Server）：实现子表的装载/卸出、表格内容的读和写，子表的合并和分裂。Tablet Server服务的数据包括操作日志以及每个子表上的sstable数据，这些数据存储在底层的GFS上。

### 数据分布

### 复制与一致性

注：之所以采用基于内存的写操作，其实就是为了适应底层的GFS存储，在内存中将随机的写操作转换为顺序写操作，然后直接追加到GFS（因为GFS对于append支持比较好，随机写非常不好）。

### 容错

### 负载均衡

### 分裂与合并

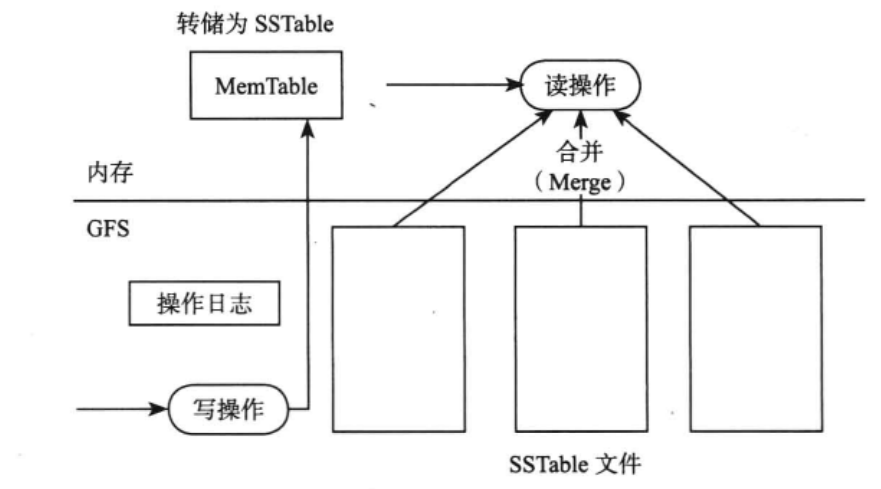
随着数据不断写入和删除，某些子表可能太大，某些子表可能太小，需要执行子表分裂和合并操作（这个要比Sharding方案好，不需要修改分发规则就可以实现表数据的迁移，Sharding数据与分发规则过于耦合）。顺序分布与哈希分布的区别在于哈希分布往往是静态的，而顺序分布式动态的，需要通过分裂与合并操作动态调整。

Bigtable每个子表的数据分为内存中的MemTable和GFS中的多个SSTable，由于Bigtable中同一个子表只被一台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为两份，比如分裂前子表的范围为（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，某个子表（1,10]的分裂主键为5，那么，分裂后生成的两个子表的数据范围为：（1,5]和[5,10）。分裂以后两个子表各自写不同的MemTable，等到执行comparation操作时再根据分裂后的子表范围生成不同的SSTable，无用的数据自然成为垃圾回收。

### 单机存储

如图所示，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，很好地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转储（Dump）到磁盘中生成SSTable文件。由于数据同时存在MemTable和多个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。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写两种需求。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，我们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上都是成立的。

插入、删除、更新、增加（Add）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中记录的只是操作，而不是最终的结果，需要等到读取（随机或者顺序）时才合并得到最终结果。



### 垃圾回收

Compaction后生成新的SSTable，原有的SSTable成为垃圾需要被回收掉。每个子表正在引用的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所有正在使用的SSTable文件，如果GFS中的SSTable没被任何一个子表使用，说明可以被回收掉。这里需要注意，由于Tablet Server执行Compaction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元数据中的SSTable文件。一种比较简单的做法是垃圾回收只删除至少一段时间，比如1小时没有被使用的SSTable文件。

### 讨论

GFS+Bigtable两层架构以一种很优雅的方式兼顾系统的强一致性和可用性。底层文件系统GFS是弱一致性系统，可用性和性能很大，但是多客户端追加可能出现重复记录等数据不一致问题；上层的表格系统Bigtable通过多级分布式索引的方式使得系统对外整体表现为强一致性。Bigtable最大的优势在于线性可扩展，单台机器出现故障可将服务迅速（一分钟以内）迁移到整个集群。Bigtable架构最多可支持几千台的集群规模，通过自动化容错技术大大降低了存储成本。

Bigtable架构也面临一些问题，如下所示：

1. 单副本服务。Bigtable架构非常适合离线或者半线上应用，然而，Tablet Server节点出现故障时部分数据短时间内无法提供读写服务，不适合实时性要求特别高的业务，如交易类业务。
2. SSD使用。Google整体架构的设计理念为通过廉价机器构建自动容错的大集群，然而，随着所说的等硬件技术的发展，机器宕机的概率变得更小，SSD和SAS混合存储也变得非常常见，存储和服务分离的架构有些不太适应。
3. 架构的复杂性导致bug定位很难。Bigtable依赖GFS和Chubby，这些依赖一系统本身比较复杂，另外，Bigtable多级分布式索引和容错等机制内部实现都非常复杂，工程量巨大，使用的过程中如果发现问题很难定位。

总体来说，Bigtable架构把可扩展性和成本做到了极致，但在线实时服务能力有一定的改进空间，适合通用的离线和半线上应用场合。

## Megastore

## Azure Storage