Kudu vs HBase

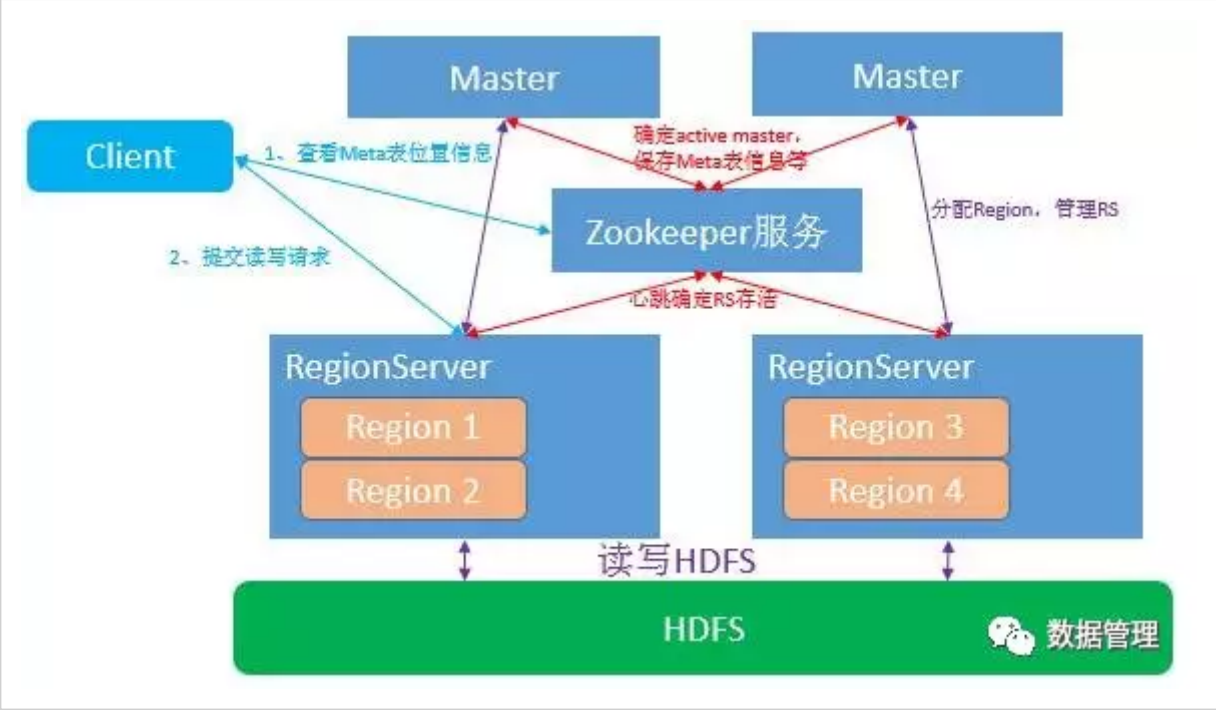
原创 2017-04-22 闵涛 数据管理

背　景

Cloudera在2016年发布了新型的分布式存储系统——kudu，kudu目前也是apache下面的开源项目。Hadoop生态圈中的技术繁多，HDFS作为底层数据存储的地位一直很牢固。而HBase作为Google BigTable的开源产品，一直也是Hadoop生态圈中的核心组件，其数据存储的底层采用了HDFS，主要解决的是在超大数据集场景下的随机读写和更新的问题。Kudu的设计有参考HBase的结构，也能够实现HBase擅长的快速的随机读写、更新功能。那么同为分布式存储系统，HBase和Kudu二者有何差异？两者的定位是否相同？我们通过分析HBase与Kudu整体结构和存储结构等方面对两者的差异进行比较。

整体结构

Hbase的整体结构



HBase的主要组件包括Master，zookeeper服务，RegionServer，HDFS。

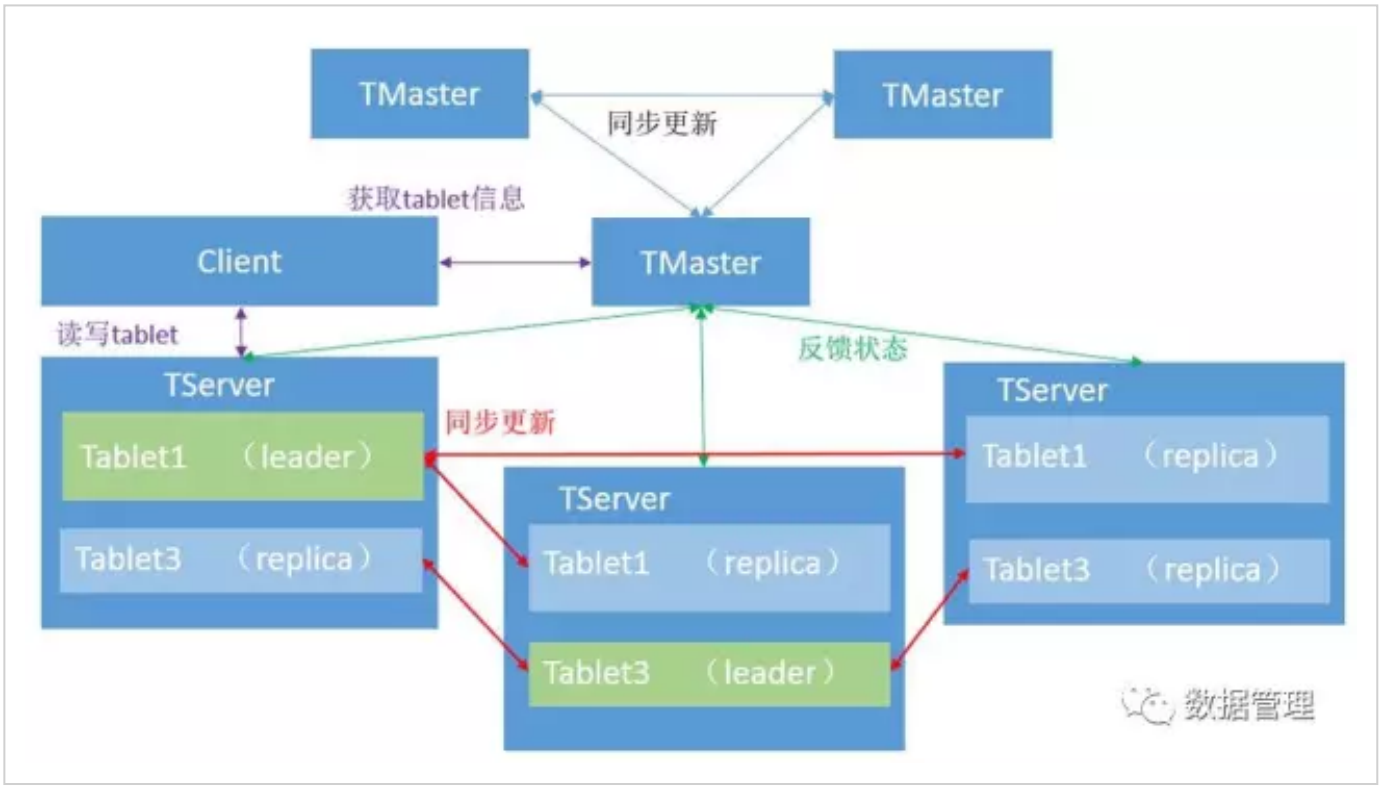
（1）Master：用来管理与监控所有的HRegionServer，也是管理HBase元数据的模块。

（2）zookeeper：作为分布式协调服务，用于保存meta表的位置，master的位置，存储RS当前的工作状态。

（3）RegionServer：负责维护Master分配的region，region对应着表中一段区间内的内容，直接接受客户端传来的读写请求。

（4）HDFS：负责最终将写入的数据持久化，并通过多副本复制实现数据的高可靠性。

Kudu的整体结构



Kudu的主要组件包括TServer和TMaster。

（1）TServer：负责管理Tablet，tablet是负责一张表中某块内容的读写，接收其他TServer中leader tablet传来的同步信息。

（2）TMaster：集群中的管理节点，用于管理tablet的基本信息，表的信息，并监听TServer的状态。多个TMaster之间通过Raft协议实现数据同步和高可用。

主要区别

Kudu结构看上去跟HBase差别并不大，主要的区别包括：

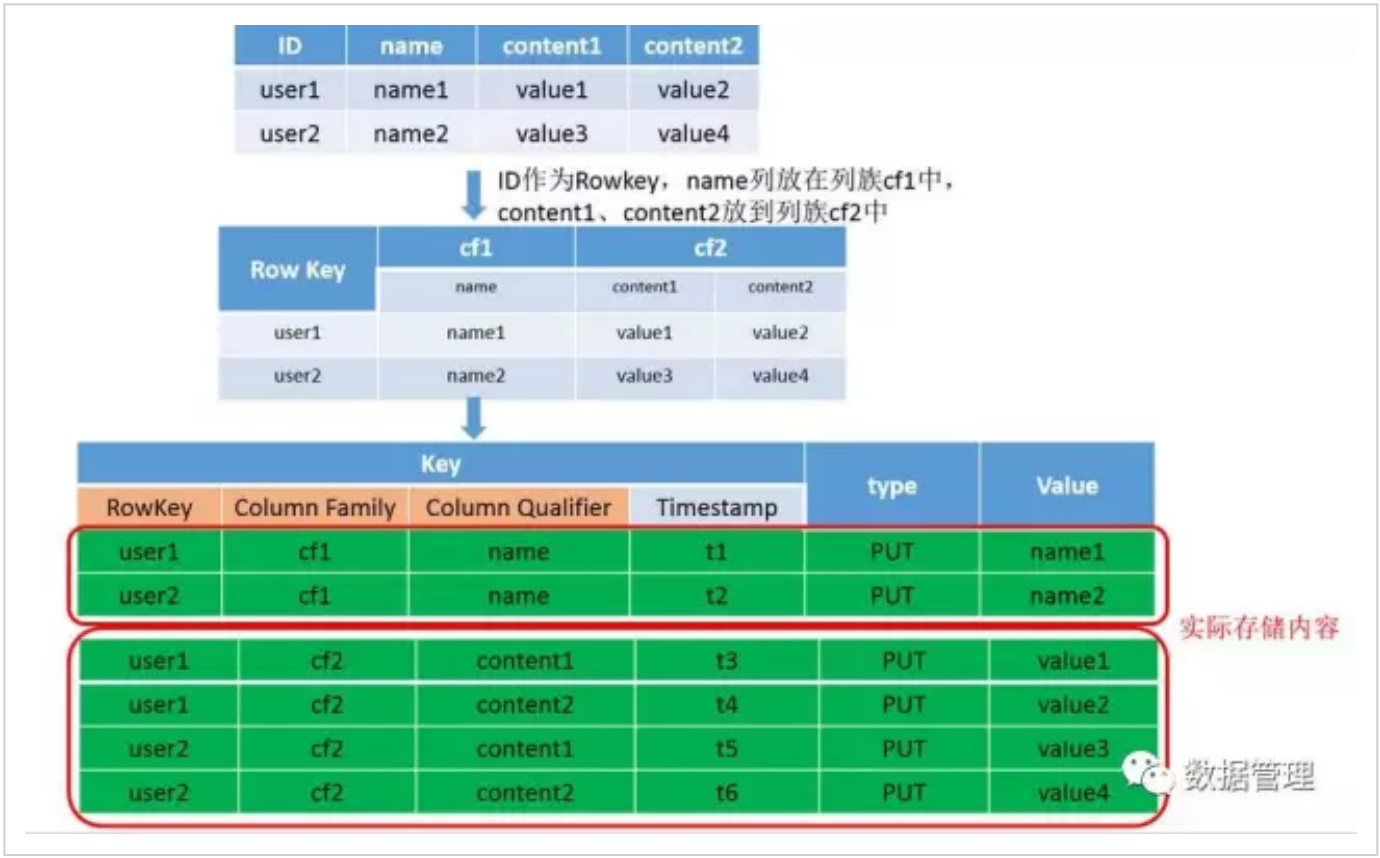
（1）Kudu将HBase中zookeeper的功能放进了TMaster内，Kudu中TMaster的功能比HBase中的Master任务要多一些。

（2）Hbase将数据持久化这部分的功能交给了Hadoop中的HDFS，最终组织的数据存储在HDFS上。Kudu自己将存储模块集成在自己的结构中，内部的数据存储模块通过Raft协议来保证leader Tablet和replica Tablet内数据的强一致性，和数据的高可靠性。为什么不像HBase一样，利用HDFS来实现数据存储，笔者猜测可能是因为HDFS读小文件时的时延太大，所以Kudu自己重新完成了底层的数据存储模块，并将其集成在TServer中。

数据存储方式

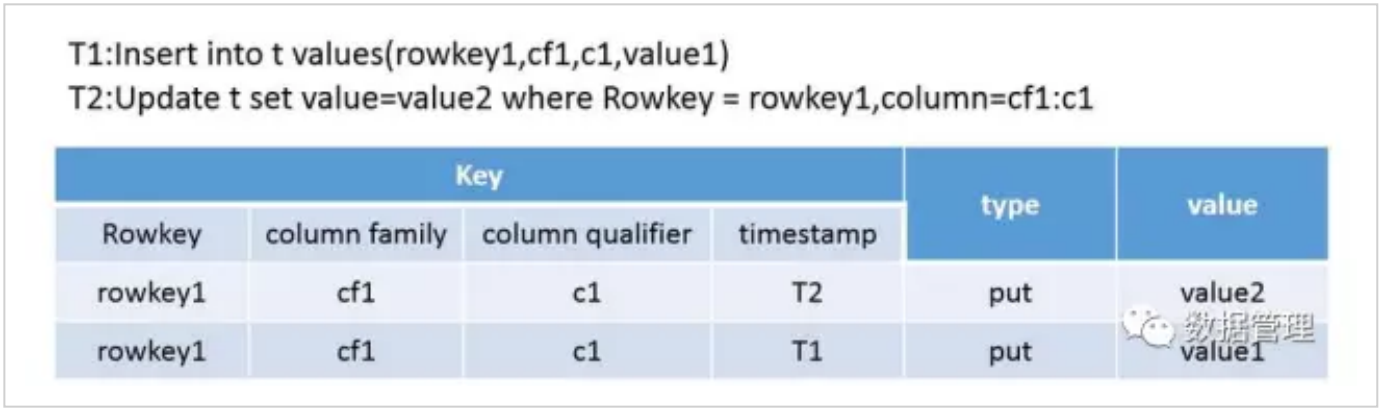
HBase

HBase是一款Nosql数据库，典型的KV系统，没有固定的schema模式，建表时只需指定一个或多个列族名即可，一个列族下面可以增加任意个列限定名。一个列限定名代表了实际中的一列，HBase将同一个列族下面的所有列存储在一起，所以HBase是一种面向列族式的数据库。



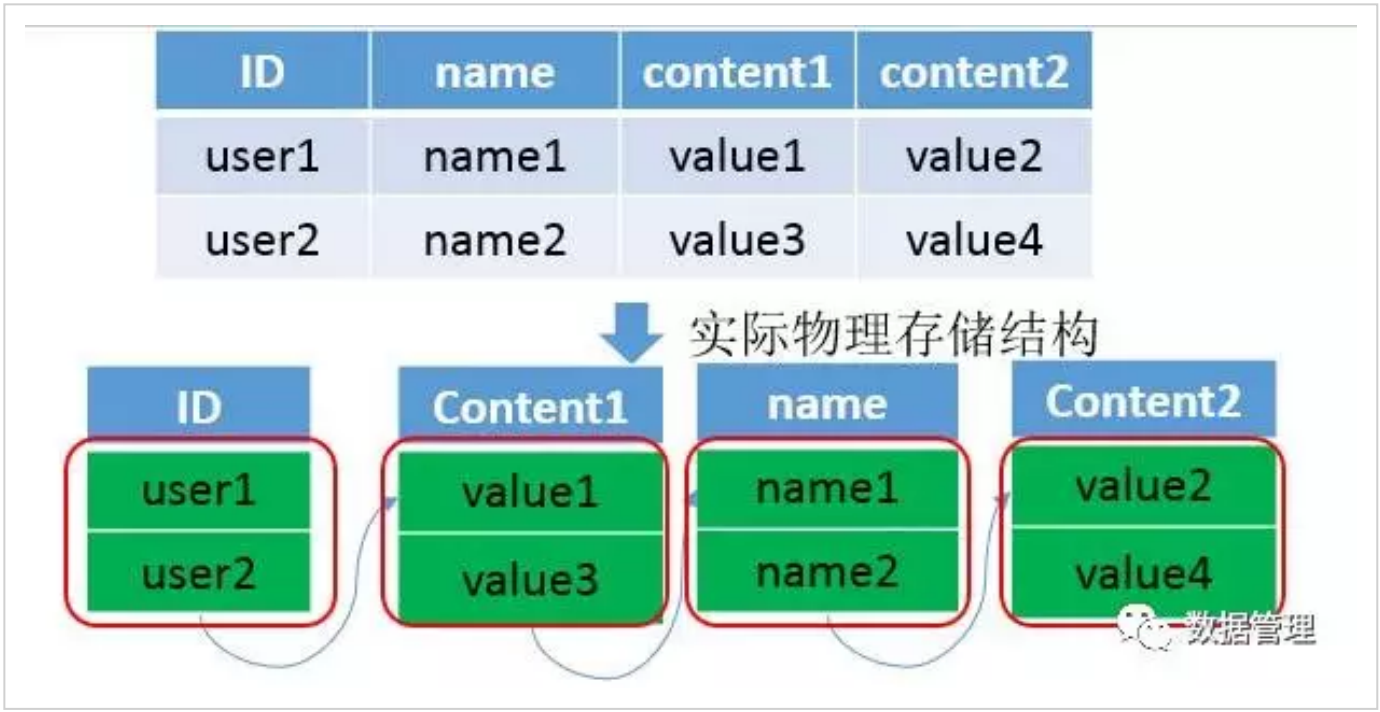
HBase将每个列族中的数据分别存储，一个列族中的每行数据中，将rowkey、列族名、列名、timestamp组成最终存取的key值，另外为了支持修改，删除，增加了一个表征该行数据是否删除的标记。在同一个列族中的所有数据，按照rowkey:columnfamily:columnQulifier:timestamp组成的key值大小进行升序排列，其中rowkey、columnfamily、columnQulifier采用的是字典顺序，其值越大，key越大，而timestamp是值越大，key越小。HBase通过按照列族分开存储，相对于行式存储能够实现更高的压缩比，这也是其比较重要的一个特性。

HBase对一行数据进行更新时，HBase也是相当于插入一行新数据，在读数据时HBase按照timestamp的大小得到经过更新过的最新数据。

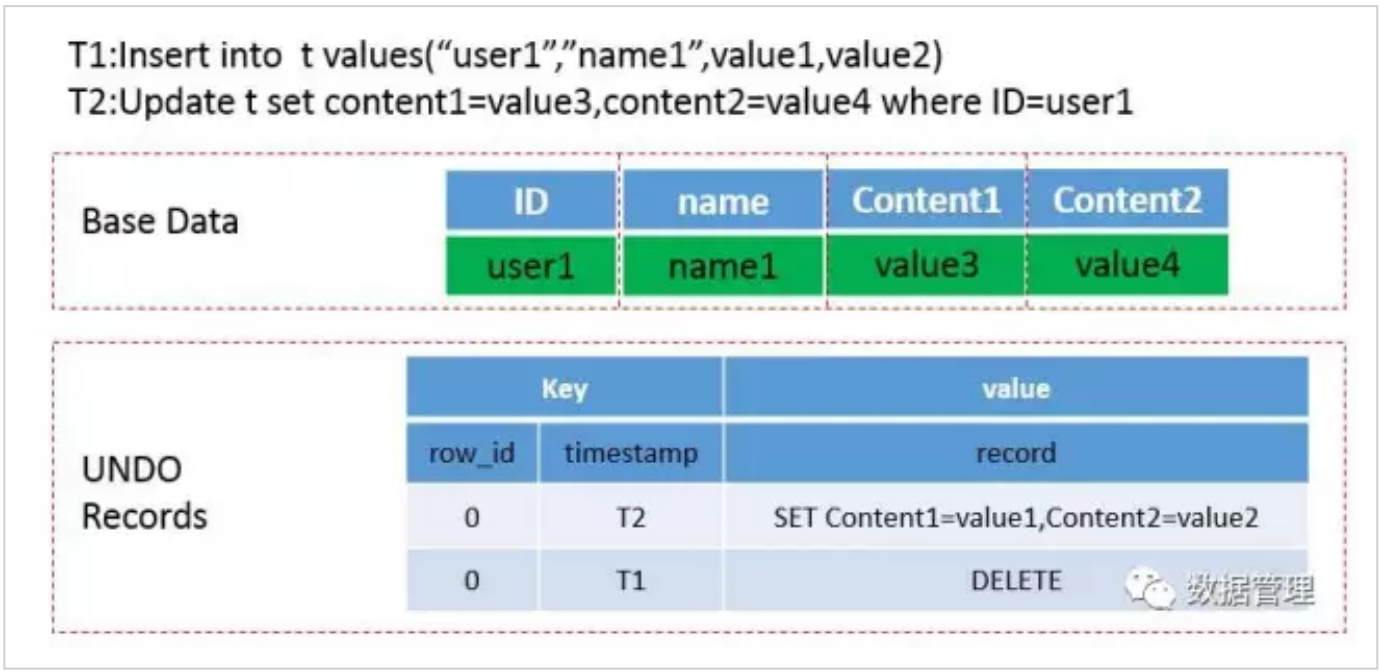


Kudu

Kudu是一种完全的列式存储引擎，表中的每一列数据都是存放在一起，列与列之间都是分开的。



为了能够保存一部分历史数据，并实现MVCC，Kudu将数据分为三个部分。一个部分叫做base data，是当前的数据；第二个部分叫做UNDO records，存储的是从插入数据时到形成base data所进行的所有修改操作，修改操作以一定形式进行组织，实现快速查看历史数据；第三个部分是REDO records，存储的是还未merge到当前数据中的更新操作。下图中表示的是在Kudu中插入一条数据、更新数据两个操作的做法，当然做法不唯一，不唯一的原因是Kudu可以选择先不将更新操作合并到base data中。



差异分析

（1）HBase是面向列族式的存储，每个列族都是分别存放的，HBase表设计时，很少使用设计多个列族，大多情况下是一个列族。这个时候的HBase的存储结构已经与行式存储无太大差别了。而Kudu，实现的是一个真正的面向列的存储方式，表中的每一列都是单独存放的；所以HBase与Kudu的差异主要在于类似于行式存储的列族式存储方式与典型的面向列式的存储方式的差异；

（2）HBase是一款NoSQL类型的数据库，对表的设计主要在于rowkey与列族的设计，列的类型可以不指定，因为HBase在实际存储中都会将所有的value字段转换成二进制的字节流。因为不需要指定类型，所以在插入数据的时候可以任意指定列名（列限定名），这样相当于可以在建表之后动态改变表的结构。Kudu因为选择了列

式存储，为了更好的提高列式存储的效果，Kudu要求在建表时指定每一列的类型，这样的做法是为了根据每一列的类型设置合适的编码方式，实现更高的数据压缩比，进而降低数据读入时的IO压力；

（3）HBase对每一个cell数据中加入了timestamp字段，这样能够实现记录同一rowkey和列名的多版本数据，另外HBase将数据更新操作、删除操作也是作为一条数据写入，通过timestamp来标记更新时间，type来区分数据是插入、更新还是删除。HBase写入或者更新数据时可以指定timestamp，这样的设置可以完成某些特定的操作；

Kudu也在数据存储中加入了timestamp这个字段，不像HBase可以直接在插入或者更新数据时设置特殊的timestamp值，Kudu的做法是由Kudu内部来控制timestamp的写入。不过Kudu允许在scan的时候设置timestamp参数，使得客户端可以scan到历史数据；

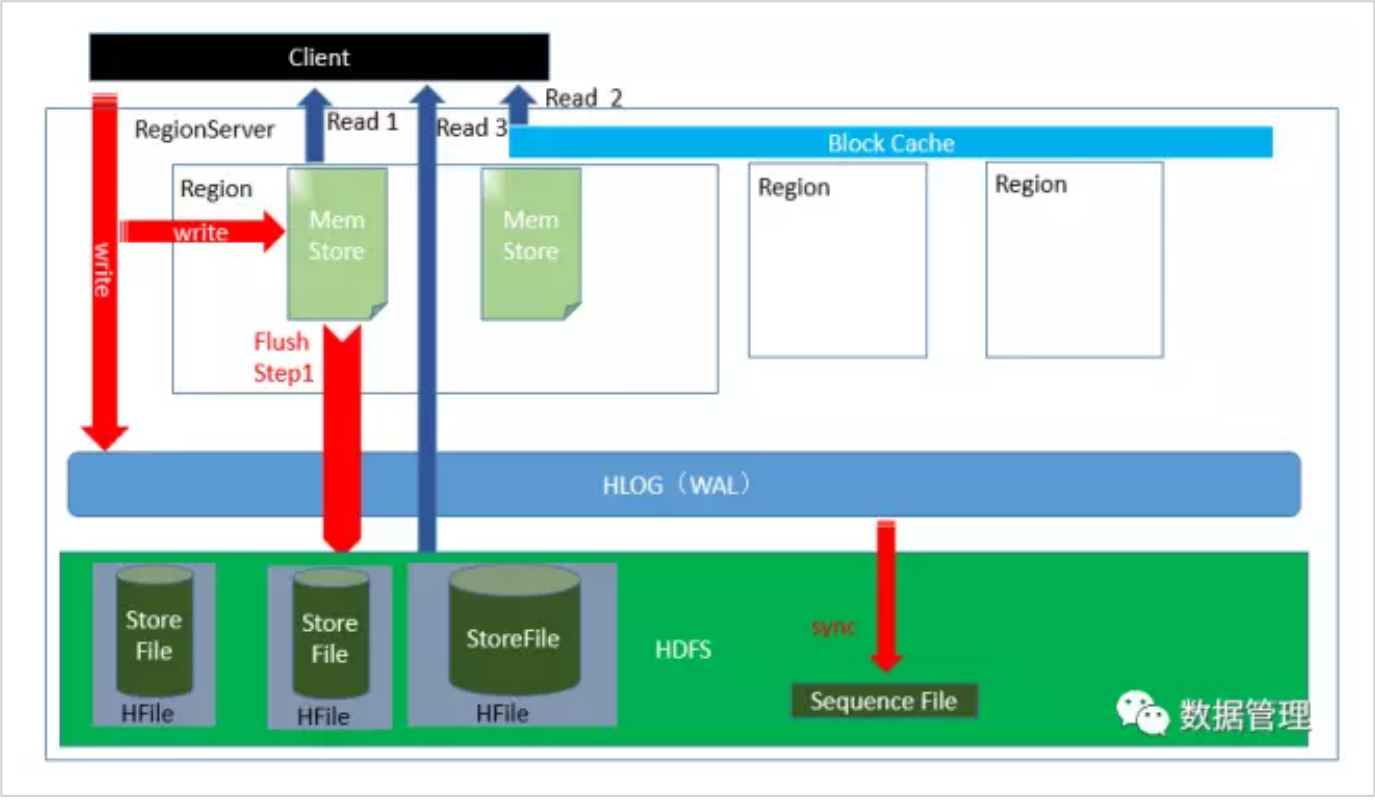
（4）相对于HBase允许多版本的数据存在，Kudu为了提高批量读取数据时的效率，要求设计表时提供一列或者多列组成一个主键，主键唯一，不允许多个相同主键的数据存在。这样的设置下，Kudu不能像HBase一样将更新操作直接转换成插入一条新版本的数据，Kudu的选择是将写入的数据，更新操作分开存储；

（5）当然还有一些其他的行式存储与列式存储之间在不同应用场景下的性能差异。

写入和读取过程

HBase

HBase作为一种非常典型的LSM结构的分布式存储系统，是Google bigtable的apache开源版本。经过近10年的发展，HBase已经成为了一个成熟的项目，在处理OLTP型的应用如消息日志，历史订单等应用较适用。在HBase中真正接受客户端读写请求的RegionServer的结构如下图所示：



关于HBase的几个关键点：

（1）在HBase中，充当写入缓存的这个结构叫做Memstore，另外会将写入操作顺序写入HLOG（WAL）中以保证数据不丢失；

（2）为了提高读的性能，HBase在内存中设置了blockcache，blockcache采用LRU策略将最近使用的数据块放在内存中；

（3）作为分布式存储系统，为保证数据不因为集群中机器出现故障而导致数据丢失，HBase将实际数据存放在HDFS上，包括storefile与HLOG。HBase与HDFS低耦合，HBase作为HDFS的客户端，向HDFS读写数据。

1. HBase写过程

（1）客户端通过客户端上保存的RS信息缓存或者通过访问zk得到需要读写的region所在的RS信息；

（2）RS接受客户端写入请求，先将写入的操作写入WAL，然后写入Memstore，这时HBase向客户端确认写入成功；

（3）HBase在一定情况下将Memstore中的数据flush成storefile（可能是Memstore大小达到一定阈值或者region占用的内存超过一定阈值或者手动flush之类的），storefile以HFile的形式存放在HDFS上；

（4）HBase会按照一定的合并策略对HDFS上的storefile进行合并操作，减少storefile的数量。

2. Hbase读过程

HBase读数据的过程比较麻烦，原因包括：

（1）HBase采用了LSM-tree的多组件算法作为数据组织方式，这种算法会导致一个region中有多个storefile；

（2）HBase中采用了非原地更新的方式，将更新操作和删除操作转换成插入一条新数据的形式，虽然这样能够较快的实现更新与删除，但是将导致满足指定rowkey，列族、列名要求的数据有多个，并且可能分布在不同的storefile中；

（3）HBase中允许设置插入和删除数据行的timestamp属性，这样导致按顺序落盘的storefile内数据的timestamp可能不是递增的。

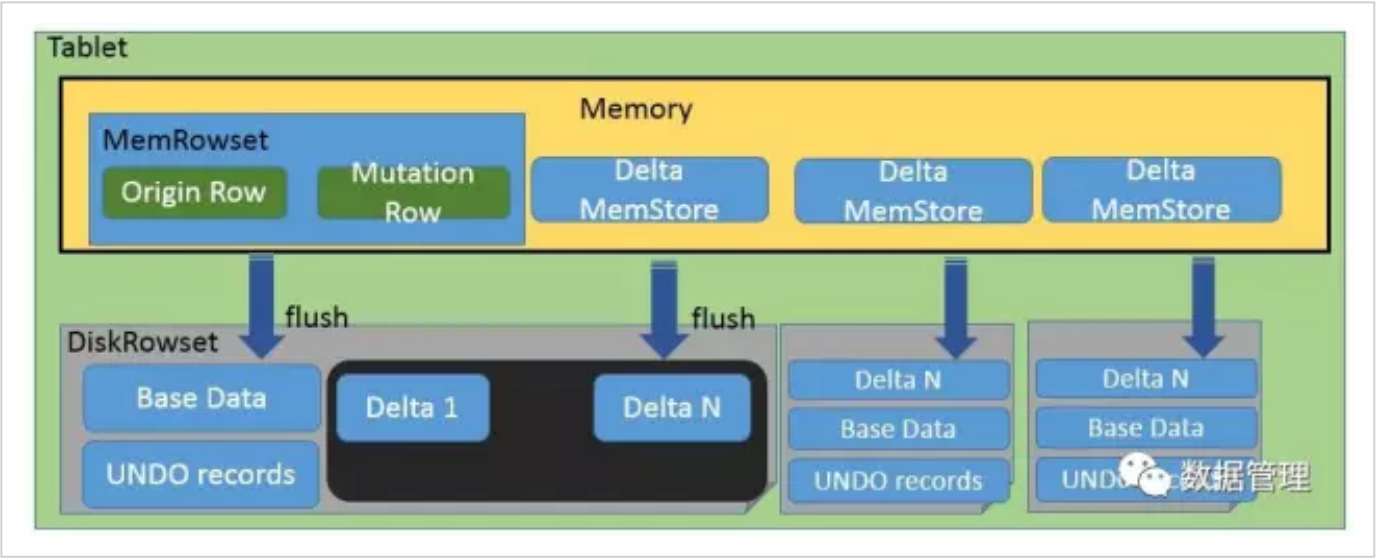
下面介绍从HBase中读取一条指定（rowkey，column family，column）的记录：

（1）读过程与HBase客户端写过程第一步一样，先尝试获取需要读的region所在的RS相关信息；

（2）RS接收读请求，因为HBase中支持多版本数据（允许存在rowkey、列族名、列名相同的数据，不同版本的数据通过timestamp进行区分），另外更新与删除数据都是通过插入一条新数据实现的。所以要准确的读到数据，需要找到所有可能存储有该条数据的位置，包括在内存中未flush的memstore，已经flush到HDFS上的storefile，所以需要在1 memstore +N storefile中查找；

（3）在找到的所有数据中通过判断timestamp值得到最终的数据。

Kudu



（1）Kudu中的Tablet是负责表中一块内容的读写工作，Tablet由一个或多个Rowset组成。其中有一个Rowset处于内存中，叫做Memrowset，Memrowset主要负责处理新的数据写入请求。DiskRowSet是MemRowset达到一定程序刷入磁盘后生成的，实质上是由一个CFile（Base Data）、多个DeltaFile（UNDO records &REDO records）和位于内存的DeltaMemStore组成。Base data、UNDO records、和REDO records都是不可修改的，DeltaMemStore达到一定大小后会将数据刷入磁盘生成新的REDO records。Kudu后台会有一个类似HBase的compaction线程按照一定的compaction 策略对tablet进行合并处理：

a. 将多个DeltaFile（REDO records）合并成一个大的DeltaFile；

b. 将多个REDO reccords文件与Base data进行合并，并生成新的 UNDO records；

c. 将多个DiskRowset之间进行合并，减少DiskRowset的数量。

（2）Kudu将最终的数据存储在本地磁盘上，为了保证数据可靠性，Kudu为一个tablet设置了多个副本（一般为3或5个）。所以一个tablet会由多个TServer负责维护，其中有个副本称为leader tablet，写入的请求只能通过leader tablet来处理，副本之间通过Raft协议保证其他副本与leader tablet的强一致性。

1. Kudu写过程

Kudu与HBase不同，Kudu将写入操作分为两种，一种是插入一条新数据，一种是对一条已插入数据的更新。

Kudu插入一条新数据

（1）客户端连接TMaster获取表的相关信息，包括分区信息，表中所有tablet的信息；

（2）客户端找到负责处理读写请求的tablet所负责维护的TServer。Kudu接受客户端的请求，检查请求是否符合要求（表结构）；

（3）Kudu在Tablet中的所有rowset（memrowset,diskrowset）中进行查找，看是否存在与待插入数据相同主键的数据，如果存在就返回错误，否则继续；

（4）Kudu在MemRowset中写入一行新数据，在MemRowset数据达到一定大小时，MemRowset将数据落盘，并生成一个diskrowset用于持久化数据，还生成一个memrowset继续接收新数据的请求。

Kudu对原有数据的更新

（1）客户端连接TMaster获取表的相关信息，包括分区信息，表中所有tablet的信息；

（2）Kudu接受请求，检查请求是否符合要求；

（3）因为待更新数据可能位于memrowset中，也可能已经flush到磁盘上，形成diskrowset。因此根据待更新数据所处位置不同，kudu有不同的做法：

a. 当待更新数据位于memrowset时，找到待更新数据所在行，然后将更新操作记录在所在行中一个mutation链表中；在memrowset将数据落盘时，Kudu会将更新合并到base data，并生成UNDO records用于查看历史版本的数据和MVCC,UNDO records实际上也是以DeltaFile的形式存放；

b. 当待更新数据位于DiskRowset时，找到待更新数据所在的DiskRowset，每个DiskRowset都会在内存中设置一个DeltaMemStore，将更新操作记录在DeltaMemStore中，在DeltaMemStore达到一定大小时，flush在磁盘，形成Delta并存在方DeltaFile中。

实际上Kudu提交更新时会使用Raft协议将更新同步到其他replica上去，当然如果在memrowset和diskrowset中都没有找到这条数据，那么返回错误给客户端；另外当DiskRowset中的deltafile太多时，Kudu会采用一定的策略对一组deltafile进行合并。

2. Kudu读过程

（1）客户端连接TMaster获取表的相关信息，包括分区信息，表中所有tablet的信息；

（2）客户端找到需要读取的数据的tablet所在的TServer，Kudu接受读请求，并记录timestamp信息，如果没有显式指定，那么表示使用当前时间；

（3）Kudu找到待读数据的所有相关信息，当目标数据处于memrowset时，根据读取操作中包含的timestamp信息将该timestamp前提交的更新操作合并到base data中，这个更新操作记录在该行数据对应的mutation链表中；

（4）当读取的目标数据位于diskrowset中，在所有DeltaFile中找到所有目标数据相关的UNDO record和REDO records，REDO records可能位于多个DeltaFile中，根据读操作中包含的timestamp信息判断是否需要将base data进行回滚或者利用REDO records将base data进行合并更新。

1. 写过程

（1）HBase写的时候，不管是新插入一条数据还是更新数据，都当作插入一条新数据来进行；而Kudu将插入新数据与更新操作分别看待；

（2）Kudu表结构中必须设置一个唯一键，插入数据的时候必须判断一些该数据的主键是否唯一，所以插入的时候其实有一个读的过程；而HBase没有太多限制，待插入数据将直接写进memstore；

（3）HBase实现数据可靠性是通过将落盘的数据写入HDFS来实现，而Kudu是通过将数据写入和更新操作同步在其他副本上实现数据可靠性。

结合以上几点，可以看出Kudu在写的性能上相对HBase有一定的劣势。

2. 读过程

（1）在HBase中，读取的数据可能有多个版本，所以需要结合多个storefile进行查询；Kudu数据只可能存在于一个DiskRowset或者MemRowset中，但是因为可能存在还未合并进原数据的更新，所以Kudu也需要结合多个DeltaFile进行查询；

（2）HBase写入或者更新时可以指定timestamp，导致storefile之间timestamp范围的规律性降低，增加了实际查询storefile的数量；Kudu不允许人为指定写入或者更新时的timestamp值，DeltaFile之间timestamp连续，可以更快的找到需要的DeltaFile；

（3）HBase通过timestamp值可以直接取出数据；而Kudu实现多版本是通过保留UNDO records（已经合并过的操作）和REDO records（未合并过的操作）完成的，在一些情况下Kudu需要将base data结合UNDO records进行回滚或者结合REDO records进行合并然后才能得到真正所需要的数据。

结合以上三点可以得出，不管是HBase还是Kudu，在读取一条数据时都需要从多个文件中搜寻相关信息。相对于HBase，Kudu选择将插入数据和更新操作分开，一条数据只可能存在于一个DiskRowset或者memRowset中，只需要搜寻到一个rowset中存在指定数据就不用继续往下找了，用户不能设置更新和插入时的timestamp值，减少了在rowset中DeltaFile的读取数量。这样在scan的情况下可以结合列式存储的优点实现较高的读性能，特别是在更新数量较少的情况下能够有效提高scan性能。

另外，本文在描述HBase读写过程中没有考虑读写中使用的优化技术如Bloomfilter、timestamp range等。其实Kudu中也有使用类似的优化技术来提高读写性能，本文只是简单的分析，因此就不再详细讨论读写过程。如有需要了解HBase的详细读写过程，

3. 其它差异

HBase：使用的java，内存的释放通过GC来完成，在内存比较紧张时可能引发full GC进而导致服务不稳定；

Kudu：核心模块用的C++来实现，没有full gc的风险。

总　结

本文主要简单介绍了一下Kudu，并在整体结构、数据存储结构和读写过程等方面上对HBase和Kudu这两款分布式存储系统进行大体上的比较。Kudu通过要求完整的表结构设置，主键的设定，以列式存储作为数据在磁盘上的组织方式，更新和数据分开等技巧，使得Kudu能够实现像HBase一样实现数据的随机读写之外，在HBase不太擅长的批量数据扫描（scan）具有较好的性能。而批量读数据正是olap型应用所关注的重点，正如Kudu官网主页上描述的，Kudu实现的是既可以实现数据的快速插入与实时更新，也可以实现数据的快速分析。Kudu的定位不是取代HBase，而是以降低写的性能为代价，提高了批量读的性能，使其能够实现快速在线分析。