HBase简介

HBase 是一个分布式的、面向列的开源数据库。建立在 HDFS 之上。Hbase的名字的来源是 Hadoop database，即 Hadoop 数据库。HBase 的计算和存储能力取决于 Hadoop 集群。

它介于 NoSql 和 RDBMS 之间，仅能通过主键(row key)和主键的 range 来检索数据，仅支持单行事务(可通过 Hive 支持来实现多表 join 等复杂操作)。

HBase中表的特点：

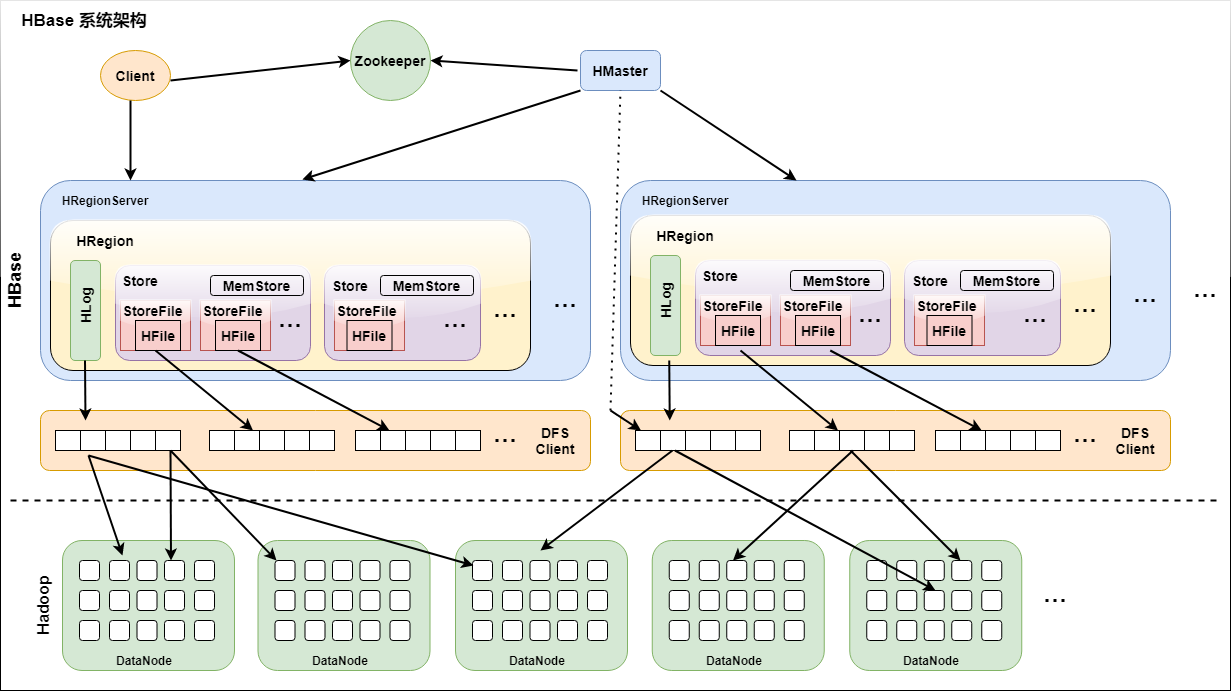
大：一个表可以有上十亿行，上百万列

面向列：面向列(族)的存储和权限控制，列(族)独立检索。

稀疏：对于为空(null)的列，并不占用存储空间，因此，表可以设计的非常稀疏。

HBase底层原理

系统架构



根据这幅图，解释下HBase中各个组件

Client

包含访问hbase的接口，Client维护着一些cache来加快对hbase的访问，比如regione的位置信息.

Zookeeper

HBase可以使用内置的Zookeeper，也可以使用外置的，在实际生产环境，为了保持统一性，一般使用外置Zookeeper。

Zookeeper在HBase中的作用：

保证任何时候，集群中只有一个master

存贮所有Region的寻址入口

实时监控Region Server的状态，将Region server的上线和下线信息实时通知给Master

HMaster

为Region server分配region

负责region server的负载均衡

发现失效的region server并重新分配其上的region

HDFS上的垃圾文件回收

处理schema更新请求

HRegion Server

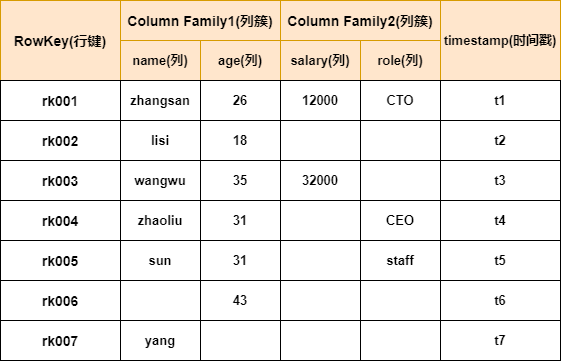
HRegion server维护HMaster分配给它的region，处理对这些region的IO请求

HRegion server负责切分在运行过程中变得过大的region

从图中可以看到，Client访问HBase上数据的过程并不需要HMaster参与（寻址访问Zookeeper和HRegion server，数据读写访问HRegione server）

HMaster仅仅维护者table和HRegion的元数据信息，负载很低。

HBase的表数据模型



行键 Row Key

与nosql数据库一样,row key是用来检索记录的主键。访问hbase table中的行，只有三种方式：

通过单个row key访问

通过row key的range

全表扫描

Row Key 行键可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10-100bytes)，在hbase内部，row key保存为字节数组。

Hbase会对表中的数据按照rowkey排序(字典顺序)

存储时，数据按照Row key的字典序(byte order)排序存储。设计key时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)。

注意：

字典序对int排序的结果是

1,10,100,11,12,13,14,15,16,17,18,19,2,20,21 … 。要保持整形的自然序，行键必须用0作左填充。

行的一次读写是原子操作 (不论一次读写多少列)。这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

列族 Column Family

HBase表中的每个列，都归属与某个列族。列族是表的schema的一部分(而列不是)，必须在使用表之前定义。

列名都以列族作为前缀。例如 courses:history ， courses:math 都属于 courses 这个列族。

访问控制、磁盘和内存的使用统计都是在列族层面进行的。

列族越多，在取一行数据时所要参与IO、搜寻的文件就越多，所以，如果没有必要，不要设置太多的列族。

列 Column

列族下面的具体列，属于某一个ColumnFamily，类似于在mysql当中创建的具体的列。

时间戳 Timestamp

HBase中通过row和columns确定的为一个存贮单元称为cell。每个 cell都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64位整型。时间戳可以由hbase(在数据写入时自动 )赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 cell中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。

为了避免数据存在过多版本造成的的管理 (包括存贮和索引)负担，hbase提供了两种数据版本回收方式：

保存数据的最后n个版本

保存最近一段时间内的版本（设置数据的生命周期TTL）。

用户可以针对每个列族进行设置。

单元 Cell

由{row key, column( =<family> + <label>), version} 唯一确定的单元。

cell中的数据是没有类型的，全部是字节码形式存贮。

版本号 VersionNum

数据的版本号，每条数据可以有多个版本号，默认值为系统时间戳，类型为Long。

物理存储

1. 整体结构

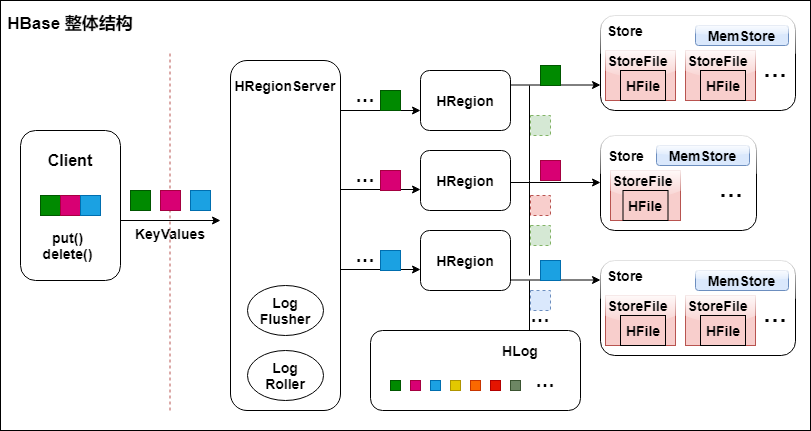


Table 中的所有行都按照 Row Key 的字典序排列。

Table 在行的方向上分割为多个 HRegion。

HRegion按大小分割的(默认10G)，每个表一开始只有一 个HRegion，随着数据不断插入表，HRegion不断增大，当增大到一个阀值的时候，HRegion就会等分会两个新的HRegion。当Table 中的行不断增多，就会有越来越多的 HRegion。

HRegion 是 HBase 中分布式存储和负载均衡的最小单元。最小单元就表示不同的 HRegion 可以分布在不同的 HRegion Server 上。但一个 HRegion 是不会拆分到多个 Server 上的。

HRegion 虽然是负载均衡的最小单元，但并不是物理存储的最小单元。

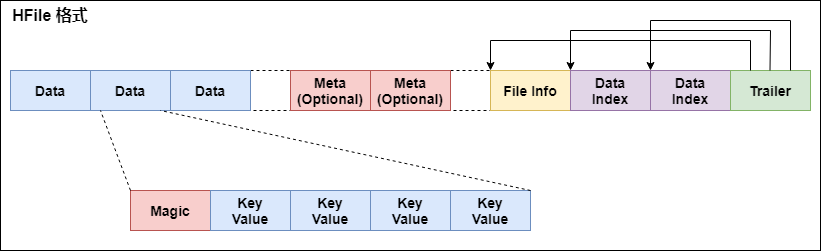
事实上，HRegion 由一个或者多个 Store 组成，每个 Store 保存一个 Column Family。

每个 Strore 又由一个 MemStore 和0至多个 StoreFile 组成。如上图。

2. StoreFile 和 HFile 结构

StoreFile以HFile格式保存在HDFS上。

HFile的格式为：



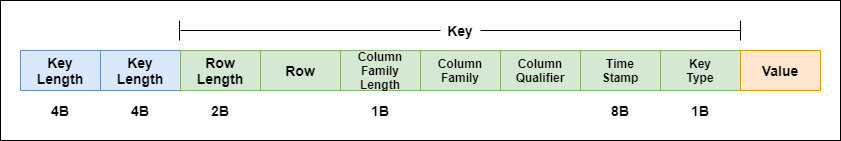
首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。正如图中所示的，Trailer中有指针指向其他数 据块的起始点。

File Info中记录了文件的一些Meta信息，例如：AVG\_KEY\_LEN, AVG\_VALUE\_LEN, LAST\_KEY, COMPARATOR, MAX\_SEQ\_ID\_KEY等。

Data Index和Meta Index块记录了每个Data块和Meta块的起始点。

Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询。 每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成, Magic内容就是一些随机数字，目的是防止数据损坏。

HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：



开始是两个固定长度的数值，分别表示Key的长度和Value的长度。紧接着是Key，开始是固定长度的数值，表示RowKey的长度，紧接着是 RowKey，然后是固定长度的数值，表示Family的长度，然后是Family，接着是Qualifier，然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。

HFile分为六个部分：

Data Block 段–保存表中的数据，这部分可以被压缩.

Meta Block 段 (可选的)–保存用户自定义的kv对，可以被压缩。

File Info 段–Hfile的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。

Data Block Index 段–Data Block的索引。每条索引的key是被索引的block的第一条记录的key。

Meta Block Index段 (可选的)–Meta Block的索引。

Trailer–这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰。

HFile的Data Block，Meta Block通常采用压缩方式存储，压缩之后可以大大减少网络IO和磁盘IO，随之而来的开销当然是需要花费cpu进行压缩和解压缩。

目前HFile的压缩支持两种方式：Gzip，Lzo。

3. Memstore与StoreFile

一个 HRegion 由多个 Store 组成，每个 Store 包含一个列族的所有数据

Store 包括位于内存的 Memstore 和位于硬盘的 StoreFile。

写操作先写入 Memstore，当 Memstore 中的数据量达到某个阈值，HRegionServer 启动 FlashCache 进程写入 StoreFile，每次写入形成单独一个 StoreFile

当 StoreFile 大小超过一定阈值后，会把当前的 HRegion 分割成两个，并由 HMaster 分配给相应的 HRegion 服务器，实现负载均衡

客户端检索数据时，先在memstore找，找不到再找storefile。

4. HLog(WAL log)

WAL 意为Write ahead log，类似 mysql 中的 binlog,用来 做灾难恢复时用，Hlog记录数据的所有变更,一旦数据修改，就可以从log中进行恢复。

每个Region Server维护一个Hlog,而不是每个Region一个。这样不同region(来自不同table)的日志会混在一起，这样做的目的是不断追加单个文件相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对table的写性能。带来的麻烦是，如果一台region server下线，为了恢复其上的region，需要将region server上的log进行拆分，然后分发到其它region server上进行恢复。

HLog文件就是一个普通的Hadoop Sequence File：

HLog Sequence File 的Key是HLogKey对象，HLogKey中记录了写入数据的归属信息，除了table和region名字外，同时还包括 sequence number和timestamp，timestamp是”写入时间”，sequence number的起始值为0，或者是最近一次存入文件系统中sequence number。

HLog Sequece File的Value是HBase的KeyValue对象，即对应HFile中的KeyValue，可参见上文描述。

读写过程

1. 读请求过程：

HRegionServer保存着meta表以及表数据，要访问表数据，首先Client先去访问zookeeper，从zookeeper里面获取meta表所在的位置信息，即找到这个meta表在哪个HRegionServer上保存着。

接着Client通过刚才获取到的HRegionServer的IP来访问Meta表所在的HRegionServer，从而读取到Meta，进而获取到Meta表中存放的元数据。

Client通过元数据中存储的信息，访问对应的HRegionServer，然后扫描所在HRegionServer的Memstore和Storefile来查询数据。

最后HRegionServer把查询到的数据响应给Client。

查看meta表信息

hbase(main):011:0> scan 'hbase:meta'

1

2. 写请求过程：

Client也是先访问zookeeper，找到Meta表，并获取Meta表元数据。

确定当前将要写入的数据所对应的HRegion和HRegionServer服务器。

Client向该HRegionServer服务器发起写入数据请求，然后HRegionServer收到请求并响应。

Client先把数据写入到HLog，以防止数据丢失。

然后将数据写入到Memstore。

如果HLog和Memstore均写入成功，则这条数据写入成功

如果Memstore达到阈值，会把Memstore中的数据flush到Storefile中。

当Storefile越来越多，会触发Compact合并操作，把过多的Storefile合并成一个大的Storefile。

当Storefile越来越大，Region也会越来越大，达到阈值后，会触发Split操作，将Region一分为二。

细节描述：

HBase使用MemStore和StoreFile存储对表的更新。

数据在更新时首先写入Log(WAL log)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时，系统会在zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了。

当系统出现意外时，可能导致内存(MemStore)中的数据丢失，此时使用Log(WAL log)来恢复checkpoint之后的数据。

StoreFile是只读的，一旦创建后就不可以再修改。因此HBase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并(minor\_compact, major\_compact),将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行split，等分为两个StoreFile。

由于对表的更新是不断追加的，compact时，需要访问Store中全部的 StoreFile和MemStore，将他们按row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，合并的过程还是比较快。

HRegion管理

HRegion分配

任何时刻，一个HRegion只能分配给一个HRegion Server。HMaster记录了当前有哪些可用的HRegion Server。以及当前哪些HRegion分配给了哪些HRegion Server，哪些HRegion还没有分配。当需要分配的新的HRegion，并且有一个HRegion Server上有可用空间时，HMaster就给这个HRegion Server发送一个装载请求，把HRegion分配给这个HRegion Server。HRegion Server得到请求后，就开始对此HRegion提供服务。

HRegion Server上线

HMaster使用zookeeper来跟踪HRegion Server状态。当某个HRegion Server启动时，会首先在zookeeper上的server目录下建立代表自己的znode。由于HMaster订阅了server目录上的变更消息，当server目录下的文件出现新增或删除操作时，HMaster可以得到来自zookeeper的实时通知。因此一旦HRegion Server上线，HMaster能马上得到消息。

HRegion Server下线

当HRegion Server下线时，它和zookeeper的会话断开，zookeeper而自动释放代表这台server的文件上的独占锁。HMaster就可以确定：

HRegion Server和zookeeper之间的网络断开了。

HRegion Server挂了。

无论哪种情况，HRegion Server都无法继续为它的HRegion提供服务了，此时HMaster会删除server目录下代表这台HRegion Server的znode数据，并将这台HRegion Server的HRegion分配给其它还活着的节点。

HMaster工作机制

master上线

master启动进行以下步骤:

从zookeeper上获取唯一一个代表active master的锁，用来阻止其它HMaster成为master。

扫描zookeeper上的server父节点，获得当前可用的HRegion Server列表。

和每个HRegion Server通信，获得当前已分配的HRegion和HRegion Server的对应关系。

扫描.META.region的集合，计算得到当前还未分配的HRegion，将他们放入待分配HRegion列表。

master下线

由于HMaster只维护表和region的元数据，而不参与表数据IO的过程，HMaster下线仅导致所有元数据的修改被冻结(无法创建删除表，无法修改表的schema，无法进行HRegion的负载均衡，无法处理HRegion 上下线，无法进行HRegion的合并，唯一例外的是HRegion的split可以正常进行，因为只有HRegion Server参与)，表的数据读写还可以正常进行。因此HMaster下线短时间内对整个HBase集群没有影响。

从上线过程可以看到，HMaster保存的信息全是可以冗余信息（都可以从系统其它地方收集到或者计算出来）

因此，一般HBase集群中总是有一个HMaster在提供服务，还有一个以上的‘HMaster’在等待时机抢占它的位置。

HBase三个重要机制

1. flush机制

1.（hbase.regionserver.global.memstore.size）默认;堆大小的40%

regionServer的全局memstore的大小，超过该大小会触发flush到磁盘的操作,默认是堆大小的40%,而且regionserver级别的flush会阻塞客户端读写

2.（hbase.hregion.memstore.flush.size）默认：128M

单个region里memstore的缓存大小，超过那么整个HRegion就会flush,

3.（hbase.regionserver.optionalcacheflushinterval）默认：1h

内存中的文件在自动刷新之前能够存活的最长时间

4.（hbase.regionserver.global.memstore.size.lower.limit）默认：堆大小 \* 0.4 \* 0.95

有时候集群的“写负载”非常高，写入量一直超过flush的量，这时，我们就希望memstore不要超过一定的安全设置。在这种情况下，写操作就要被阻塞一直到memstore恢复到一个“可管理”的大小, 这个大小就是默认值是堆大小 \* 0.4 \* 0.95，也就是当regionserver级别的flush操作发送后,会阻塞客户端写,一直阻塞到整个regionserver级别的memstore的大小为 堆大小 \* 0.4 \*0.95为止

5.（hbase.hregion.preclose.flush.size）默认为：5M

当一个 region 中的 memstore 的大小大于这个值的时候，我们又触发了region的 close时，会先运行“pre-flush”操作，清理这个需要关闭的memstore，然后 将这个 region 下线。当一个 region 下线了，我们无法再进行任何写操作。 如果一个 memstore 很大的时候，flush 操作会消耗很多时间。“pre-flush” 操作意味着在 region 下线之前，会先把 memstore 清空。这样在最终执行 close 操作的时候，flush 操作会很快。

6.（hbase.hstore.compactionThreshold）默认：超过3个

一个store里面允许存的hfile的个数，超过这个个数会被写到新的一个hfile里面 也即是每个region的每个列族对应的memstore在flush为hfile的时候，默认情况下当超过3个hfile的时候就会对这些文件进行合并重写为一个新文件，设置个数越大可以减少触发合并的时间，但是每次合并的时间就会越长

2. compact机制

把小的storeFile文件合并成大的HFile文件。

清理过期的数据，包括删除的数据

将数据的版本号保存为1个。

split机制

当HRegion达到阈值，会把过大的HRegion一分为二。

默认一个HFile达到10Gb的时候就会进行切分。