RDBMS VS HBASE

## 关系数据库的困难：

  简单的事情只要上了量就会变得无比复杂的事情（上亿条插入查询非常慢）

   Orderby耗费很多性能

   大量发生，但又无法分布式处理

  顾客需要实时看到自己的足迹，因此不能使用缓存技术

## [Hbase](http://lib.csdn.net/base/hbase)迎接挑战

**天生就是面向时间戳的查询**

行键的设计非常关键（**因为HBase里面没有去重，关联等操作**）。基于行键的查询异常迅速，**特别是最近的数据被存放在内存的memstore里，完全没有IO开销**。

  分布式化解负荷

HBase简介：

官网：http://hbase.apache.org/book.html#arch.overview

Hbase是一个分布式的 面向列的数据存储系统，通过在HDFS上提供随机读写来解决Hadoop不能处理**的**问题，Hbase自底层设计开始即聚焦于各种可伸缩性问题：表可以很“高”（数**十亿**个数据行）；表可以很“宽”（数**百万**个列）；水平分区并在上千个普通商用机节点上自动复制。

HBase

非结构化数据存储的数据库.

所谓非结构化数据存储就是说HBase是基于列的而不是基于行的模式

HBase：类似于现在流行的Memcache，但不仅仅是简单的一个key对应一个 value，你很可能需要存储多个属性的[**数据结构**](http://lib.csdn.net/base/datastructure)，但没有传统数据库表中那么多的关联关系，这就是所谓的松散数据。

简单来说，你在HBase中的表创建的可以看做是一张很大的表，而这个表的属性可以根据需求去动态增加，在HBase中没有表与表之间关联查询。你只需要 告诉你的数据存储到Hbase的那个column families 就可以了，不需要指定它的具体类型：char,varchar,int,tinyint,text等等。

## 使用情况

facebook， ebay，taobao，百度

阿里在大量使用中，用的场景还比较丰富。全面服务于淘宝、天猫、蚂蚁金服、菜鸟、阿里云、高德、优酷等各个领域，满足业务对于大数据分布式存储的基本需求。

[云数据库HBase，云时代的大会数据存储 - 阿里云](https://link.zhihu.com/?target=https%3A//promotion.aliyun.com/ntms/act/hbase.html)

<https://promotion.aliyun.com/ntms/act/hbase.html>

[什么情况下使用HBase](http://blog.csdn.net/jiangshouzhuang/article/details/48595891)

成熟的数据分析主题，查询模式已经确立并且不轻易改变

传统的关系型[**数据库**](http://lib.csdn.net/base/mysql)已经无法承受负荷，**高速插入，大量读取**

适合海量的，但同时也是简单的操作（例如value-key）

[http://hbase.apache.org/book.html#arch.overview](http://hbase.apache.org/book.html" \l "arch.overview)

## HBase数据模型

Table & Column Family

|  |  |  |  |
| --- | --- | --- | --- |
| **Row Key** | **Timestamp** | **Column Family** | |
| **URI** | **Parser** |
| r1 | t3 | url=http://www.taobao.com | title=天天特价 |
| t2 | host=taobao.com |  |
| t1 |  |  |
| r2 | t5 | url=http://www.alibaba.com | content=每天… |
| t4 | host=alibaba.com |  |

Row Key: 行键，Table的主键，Table中的记录按照Row Key排序（字典排序）

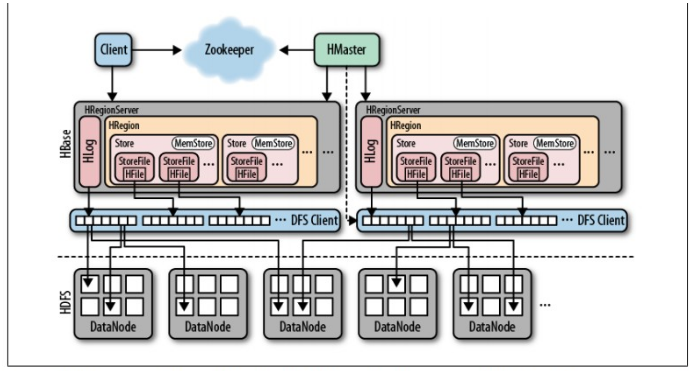
即如有1->11->112->2…

Timestamp: 时间戳，每次数据操作对应的时间戳，可以看作是数据的version number，默认最近的数据放在最前面。

Column Family：列簇，Table在水平方向有一个或者多个Column Family组成，一个Column Family中可以由任意多个Column组成，即Column Family支持动态扩展，无需预先定义Column的数量以及类型，所有Column均以二进制格式存储，用户需要自行进行类型转换。

HBase架构：

<http://blog.jobbole.com/91913/>



### client:

hbase访问入口

与hmaster通信，进行管理类操作（ddl，如创建表等，管理hdfs namespace等）

与hregionserver进行通信，数据读写类操作。（即读写操作不经过hmaster，直接与regionserver通信）

### zookeeper:

保证任何时候，只有一个hmaster,无单点故障

存储hbase的schema和table元数据

1、hbase regionserver 向zookeeper注册，提供hbase regionserver状态信息（是否在线）,并报告给master

2、hmaster启动时候会将hbase系统表-ROOT- 加载到 zookeeper cluster，通过zookeeper cluster可以获取当前系统表.META.的存储所对应的regionserver信息。

zookeeper是hbase集群的"协调器"。由于zookeeper的轻量级特性，因此我们可以将多个hbase集群共用一个zookeeper集群，以节约大量的服务器。多个hbase集群共用zookeeper集群的方法是使用同一组ip，修改不同hbase集群的"zookeeper.znode.parent"属性，让它们使用不同的根目录。比如cluster1使用/hbase-c1,cluster2使用/hbase-c2，等等。

### HMaster

启动多个，只有一个在运行，即主备模式（高可用）

管理regionserver的负载均衡

region split后，新region的分布

管理用户对表的增删改查操作。

其中HMaster节点用于：

1. 管理HRegionServer，实现其负载均衡。

2. 管理和分配HRegion，比如在HRegion split时分配新的HRegion；在HRegionServer退出时迁移其内的HRegion到其他HRegionServer上。

3. 实现DDL操作（Data Definition Language，namespace和table的增删改，column familiy的增删改等）。

4. 管理namespace和table的元数据（实际存储在HDFS上）。

5. 权限控制（ACL）。

### hregion server

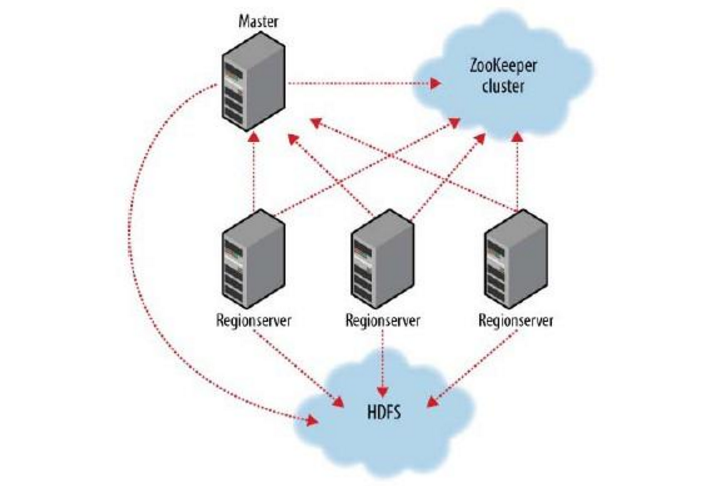
管理region,处理region io请求切分变得过大的region

client访问hbase的数据过程并不需要master参与 。

读写HDFS，管理Table中的数据。

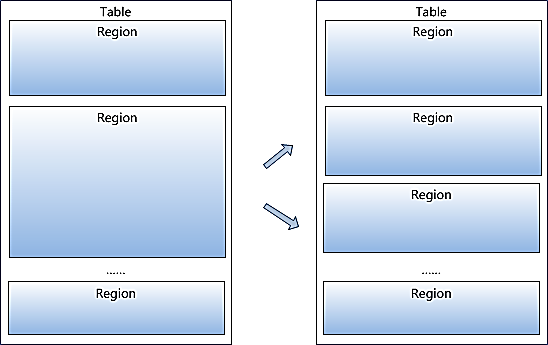
Client直接通过HRegionServer读写数据（从HMaster中获取元数据，找到RowKey所在的HRegion/HRegionServer后）。

 HBase部署相对是一个较大的动作，其依赖于zookeeper cluster,hadoop HDFS。



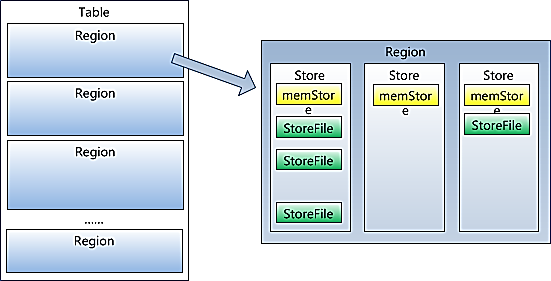
### region

开始的时候每个Table只有一个Region，当Region超过一个阀值的时候就会分割成两个Region。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002ZyuBSzy6R0eUwwOQ10)

​

这些Region会被分布到不同的RegionServer上，Region是分布式和负载均衡的最小单元，但并不是存储的最小单元，每个Region中又包含多个Store，每个Store存储一个列簇，而每个Store中由一个memStore和多个StoreFile组成，每个StoreFile有一个HFile

[](http://photo.blog.sina.com.cn/showpic.html#blogid=&url=http://album.sina.com.cn/pic/002ZyuBSzy6R0eV7xiJ25)

​由之前的HBase的架构知道，在HRegionServer中存储了多个Region这些Region是按照行进行分割的，而每个RegionServer中都会有一个HLOG和多个Store，Store我们知道每个Store中存储的是一个列族。HLOG也成为预写式日志write ahead log （WAL），每个Region中都有一个Hlog，当存储数据的时候只有当把数据写到WAL和每个Store中的memStore才算成功，那为什么这么做呢，因为，在大型分布式中硬件故障非常常见，如果memStore还没有刷新(一会说memStore)，服务器就崩溃了，内存中的数据没有写入到硬盘中就丢失了，HBase应对办法就是在写动作完成之前先写入到WAL，而WAL是底层文件系统的一个文件，这样如果内存中的数据还没有写入到硬盘上即便发生了崩溃也可以根据Hlog进行恢复，这就是为什么要当WAL和memStore写完之后才算完成。memStore是什么呢，memStore是内存中的写入缓冲区，HBase写入数据到硬盘之前是将数据积累在memStore这个内存缓冲区上面的，当memStore缓冲区写满之后会将数据写到StoreFile上，storeFile是分布式上的存储空间，只有当storeFile写满之后才会将数据存储到HFile中，HFile是Hbase中底层的存储格式，一个类簇(一个Store上)有多个HFile，但是一个HFile只能在一个列族中，一个store有一个memStore

### HLog

wal：预写日志文件

灾难备份

一个RegionServer维护一个LOG

与一个HRGION维护一个HLOG区别  
IO流，一个RegionServer管理几百个REGION，增加io

实际上regionserver是1个用的多，也不会经常regionserver挂掉

### blockCache

相对于regionserver的

增加读取性能

分级的

single（相对不重要，新数据过来，优先清除）

multi（用的次数比较多，新数据过来，其次清除）

InMemory（元数据，不会清除）

hbase(main):007:0> describe 'user'

DESCRIPTION ENABLED

'user', {NAME => 'info', DATA\_BLOCK\_ENCODING => 'NONE', BLOOMFILTER => ' true

ROW', REPLICATION\_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE',

MIN\_VERSIONS => '0', TTL => 'FOREVER', KEEP\_DELETED\_CELLS => 'false', BL

OCKSIZE => '65536', IN\_MEMORY => 'false', BLOCKCACHE => 'true'}

1 row(s) in 0.1070 seconds

每load一个block到cache时，都会检查当前cache的size是否已经超过了“警戒线”，这个“警戒线”是一个规定的当前block cache总体积占额定体积的安全比例，默认该值是0.85，即当加载了一个block到cache后总大小超过了既定的85%就开始触发异步的evict操作了。  
  
evict的逻辑是这样的：遍历cache中的所有block,根据它们所属的级别(single,multi,in-memory)分拨到三个优先级队列中，队头元素是最旧（最近访问日间值最小）的那个block。对这个三队列依次驱逐对头元素，释放空间。  
  
所以说:in-memory的block与其他类型的block并无本质上的不同，它不会长久驻留cache而不被逐出cache, 当不断有新的in-memory的block被访问，而现有in-memory cache已达到上限时，旧的in-memory block就会被替换出去，除非，所有in-memory的block的总体积小于in-memory cache。  
  
但是in-memory的block确实不同于其他两种block的地方在于它的这个“in-memory”特征是静态指定的（在column family上设置），不会像其他两种cache会因访问频率而发生改变，这就决定了它的独立性，另外两种block访问次数再多也不会被放到in-memory的区段里去，in-memory的block不管是第几次访问，总是被放置到in-memory的区段中。  
  
从in-memory cache的这些特性上来看，需要特别强调的是：  
  
1. 标记IN\_MEMORY=>'true'的column family的总体积最好不要超过in-memory cache的大小（in-memory cache = heap size \* hfile.block.cache.size \* 0.85 \* 0.25），特别是当总体积远远大于了in-memory cache时，会在in-memory cache上发生严重的颠簸。  
  
2. 换个角度再看，普遍提到的使用in-memory cache的场景是把元数据表的column family声明为IN\_MEMORY=>'true。实际上这里的潜台词是：元数据表都很小。其时我们也可以大胆地把一些需要经常访问的，总体积不会超过in-memory cache的column family都设为IN\_MEMORY=>'true'从而更加充分地利用cache空间。就像前面提到的，普通的block永远是不会被放入in-memory cache的，只存放少量metadata是对in-memory cache资源的浪费（未来的版本应该提供三种区段的比例配置功能）。

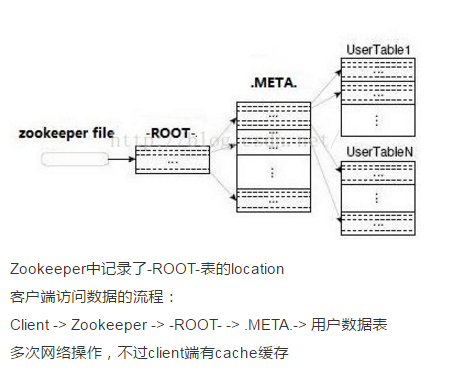
## 查询数据：

client->zookeeper->hbase:meta->user-table->put/get

首先到zookeeper上找到系统自带的hbase的meta表的元数据信息，此表记录hbase:meta 表的region在regionserver上的位置。在regionserver上找到对应的位置读取meta表，此表存放用户表的regionserver位置，从而获得regionserver的位置，到对应的位置进行put/get

先到memstore中读到数据，再到blockcache->storefile,合并两者结果。

|  |
| --- |
| 第一步：client 查询数据get ‘rowkey’  get:the fast  **hbase**(main):014:0> get 'user','10001'  COLUMN CELL  info:address timestamp=1472122892614, value=shanghai  info:age timestamp=1472122845656, value=25  info:name timestamp=1472122821525, value=zhangsan  info:sex timestamp=1472122866991, value=male  4 **row**(s) **in** 0.0350 seconds  **hbase**(main):015:0> get 'user','10001','info:name'  COLUMN CELL  info:name timestamp=1472122821525, value=zhangsan  第二步：  **hbase**(main):001:0> list\_namespace  NAMESPACE  default  hbase  2 **row**(s) **in** 1.0230 seconds  **hbase**(main):004:0> list\_namespace\_tables 'hbase'  TABLE  meta  namespace  2 **row**(s) **in** 0.0290 seconds    hbase(main):005:0> scan 'hbase:meta'  ROW COLUMN+CELL  ...  user,,1472122399908.011ff289dad7472fed3b1f9fd99 column=info:regioninfo, timestamp=1472122399990, value={ENCODED => 011ff289dad7472fed3b1f9fd997f885, NAME => 'user,,1472122399908.011ff289da  7f885. d7472fed3b1f9fd997f885.', STARTKEY => '', ENDKEY => ''}  user,,1472122399908.011ff289dad7472fed3b1f9fd99 column=info:seqnumDuringOpen, timestamp=1494488996969, value=\x00\x00\x00\x00\x00\x00\x00\x0D  7f885.  user,,1472122399908.011ff289dad7472fed3b1f9fd99 column=info:server, timestamp=1494488996969, value=hadoop.study.com:60020  7f885.  user,,1472122399908.011ff289dad7472fed3b1f9fd99 column=info:serverstartcode, timestamp=1494488996969, value=1494488965309  7f885.  ....      [zk: localhost:2181(CONNECTED) 2] get /hbase/meta-region-server  �regionserver:60020������]PBUF    hadoop.study.com����ﳿ+  cZxid = 0x190  ctime = Thu May 11 15:49:52 CST 2017  mZxid = 0x190  mtime = Thu May 11 15:49:52 CST 2017  pZxid = 0x190  cversion = 0  dataVersion = 0  aclVersion = 0  ephemeralOwner = 0x0  dataLength = 68  numChildren = 0 |



scan range:

scan :

delete:

http://blog.jobbole.com/91913/

row key:字典排序

## 写入数据方式：

client->hlog->MemStore，满 ->flush 成一个StoreFile，阈值->Compact合并操作->多个storefile合并成一个storefile,同时版本合并，数据删除->strore file 越来越大->单个storefile超过一定阈值后,split->一个region变成2个->原来 的下线，新生的被master分配到regionserver上->分流，减轻压力

版本合并：对HDFS而言，就是放数据,put

HBASE中的数据删除首先是逻辑删除，即只是打个标签，并未真正删除，只所以检索不到，实际上只是不显示。

但是当strorfile合并成一个操作时，会真正将数据删除

## 索引：

默认是rowkey

针对某些需求，可以创建二级索引

coreprocess/solor

## 文件压缩

文件压缩：SNAPPY

查看底层的HADOOP

## hbase flush

刷新memstore中的数据

## Compact

**minor/major合并**

minor:较多的小文件重写为数量较少的大文件.多路归并，速度快。

major：将一个region多个hfile重写为一个新的hfile.

与minor不同：

扫描所有的key/value

重写全部的数据，略过做了删除标记的数据。如超过版本号限制的，做过删除标记的，超过生命周期的就不再写入。即此时是真正的删除更新数据。

major block客户端对该regionr的操作。

查看是否真正删除：

scan ‘hbase:meta’

在compact时，当选中的文件数小于这个store所有文件数\*ratio(hbase.hstore.compaction.ratio)时，进行Minor compact.  
当

boolean isMajor = (filesToCompact.getFilesToCompact().size() == this.storefiles.size());

时，也就是当选中的文件数等于该store数时，会进行Major compact.

## 生活案例分析

