1. 介绍

2.HBase的实现原理

2.1 表和Region

2.2 Region的定位

3. HBase系统架构

4. 二级索引

4.1 前言

4.2 基于Coprocessor方案

4.3 非Coprocessor方案

1. 介绍

HBase是一个分布式的、面向列的开源数据库，

HBase是建立在Hadoop文件系统之上的分布式面向列的数据库。HBase是一个数据模型，类似于谷歌的bigTable设计，可以提供快速随机访问海量结构化数据。它利用了Hadoop的文件系统（HDFS）提供的容错能力。

来自〈https://www.yiibai.com/hbase/〉

HBase是一个面向列的数据库，在表中它由行排序(HBase只有一个索引——行键)。表模式定义只能列族，也就是键值对。一个表有多个列族以及每一个列族可以有任意数量的列。后续列的值连续地存储在磁盘上。表中的每个单元格值都具有时间戳,(记录上次修改的时间,hbase删除不是真正的删除,只是不显示(或者覆盖显示),到最后由hbase自己回收这些不用的数据)

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。

总之，在一个HBase：

- 表是行的集合。(许多行组成表)
- 行是列族的集合。(许多列族组成行)(每一行都有一个索引,称为行健)
- 列族是列的集合。(许多列组成列族)
- 列是单元格的集合。(许多单元格组成列)

下面给出的表中是HBase模式的一个例子。

Rowide	Column Family			Column Family			Column Family			Column Family		
	col1	col2	col3	col1	col2	col3	col1	col2	col3	col1	col2	col3
1												
2												
3												

来自〈https://www.yiibai.com/hbase/〉

Row Key	Time Stamp	Column Family:c1		Column Family:c2	
		列	值	列	值

r1	t7	c1:1	value1-1/1		
	t6	c1:2	value1-1/2		
	t5	c1:3	value1-1/3		
	t4			c2:1	value1-2/1
	t3			c2:2	value1-2/2
t2	t2	c1:1	value2-1/1		
	t1			c2:1	value2-1/1

从上表可以看出，test表有r1和r2两行数据，并且c1和c2两个列族，在r1中，列族c1有三条数据，列族c2有两条数据；在r2中，列族c1有一条数据， 列族c2有一条数据，每一条数据对应的时间戳都用数字来表示，编号越大表示数据越旧，反而表示数据越新。

> https://www.cnblogs.com/duanxz/p/4660784.html

(单元格)Cell：由{row key, column(=<family> + <label>), version}是唯一确定的单元 cell中的数据是没有类型的，全部是字节码形式存贮

在HBase中，表被分割成(多个)区域(Region)，
一个region由[startkey,endkey)表示，不同的region会被Master分配给相应的RegionServer进行管理。区域被列族垂直分为Stores”。Stores被保存在HDFS文件

2.HBase的实现原理

HBase的实现包括三个主要的功能组件：

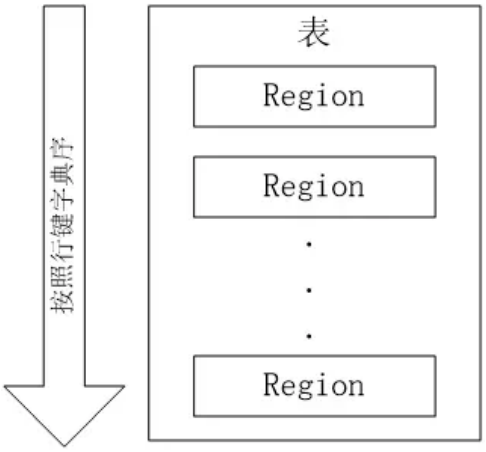
- 1、库函数：链接到每个客户端
- 2、一个Master主服务器
- 3、许多个Region服务器

> 来自：https://www.jianshu.com/p/53864dc3f7b4

主服务器Master 负责管理和维护Hbase表的分区信息，维护Region服务器列表，分配Region，负载均衡。
Region服务器 负责存储和维护分配给自己的Region，处理来自客户端的读写请求。

2.1 表和Region

一个HBase表被划分成多个Region。



图片.png

开始只有一个Region，随着内容增多后台不断分裂。Region拆分操作非常快，接近瞬间，因为拆分之后Region读取的仍然是原存储文件，直到“合并”过程把存储文件异步地写到独立的文件之后，才会读取新文件。

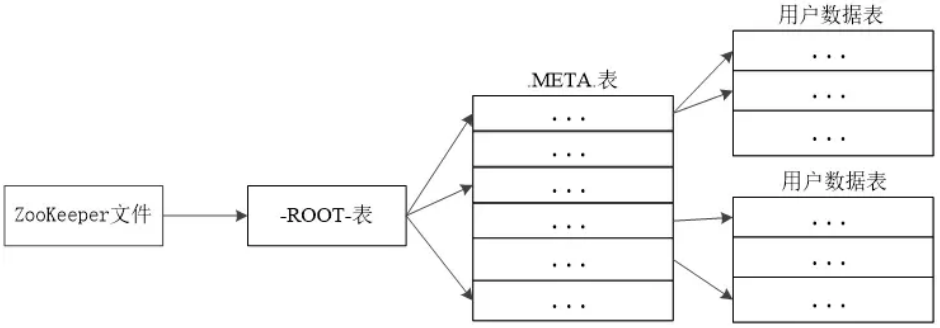
2.2 Region的定位

> 来自 <https://www.yiibai.com/hbase/hbase_architecture.html>

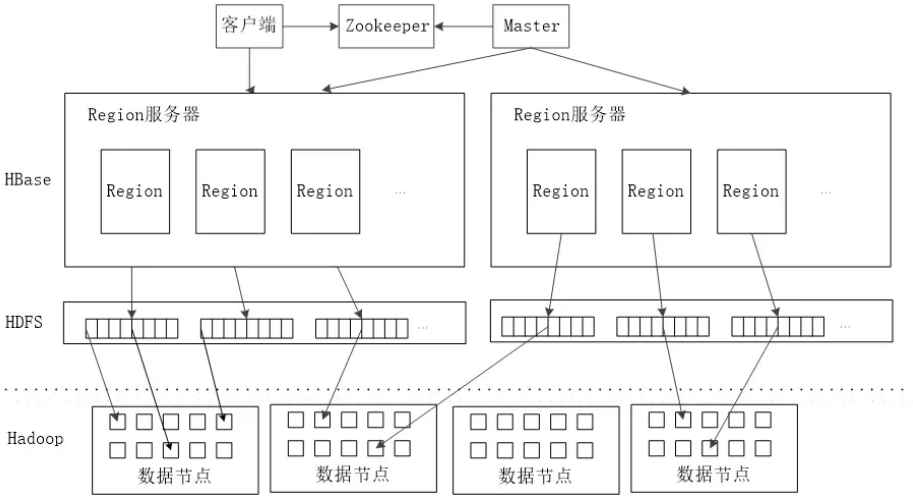
HBase中有两张特殊的Table，`-ROOT-`和`.META.`

- .META.：记录了用户表的Region信息，.META.表本身可以有多个regoin
- -ROOT-：记录了.META.表的Region信息，-ROOT-表本身只有一个region(名字被写死)
- Zookeeper中记录了-ROOT-表的位置

Client访问用户数据之前需要首先访问zookeeper，然后访问-ROOT-表，接着访问.META.表，最后才能找到用户数据的位置去访问，中间需要多次网络操作，不过client端会做cache缓存。



3. HBase系统架构

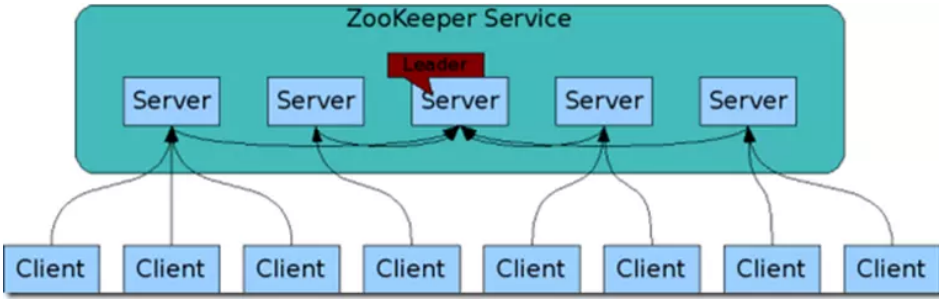


客户端

客户端(去连HBase的都叫客户端)包含访问Hbase的接口，同时在缓存中维护着已经访问过的Region位置信息，用来加快后续数据访问过程。

Zookeeper服务器

Zookeeper可以帮助选举出一个Master作为集群的总管，并保证在任何时刻总有唯一一个Master在运行，这就避免了Master的“单点失效”的问题。



Master服务器

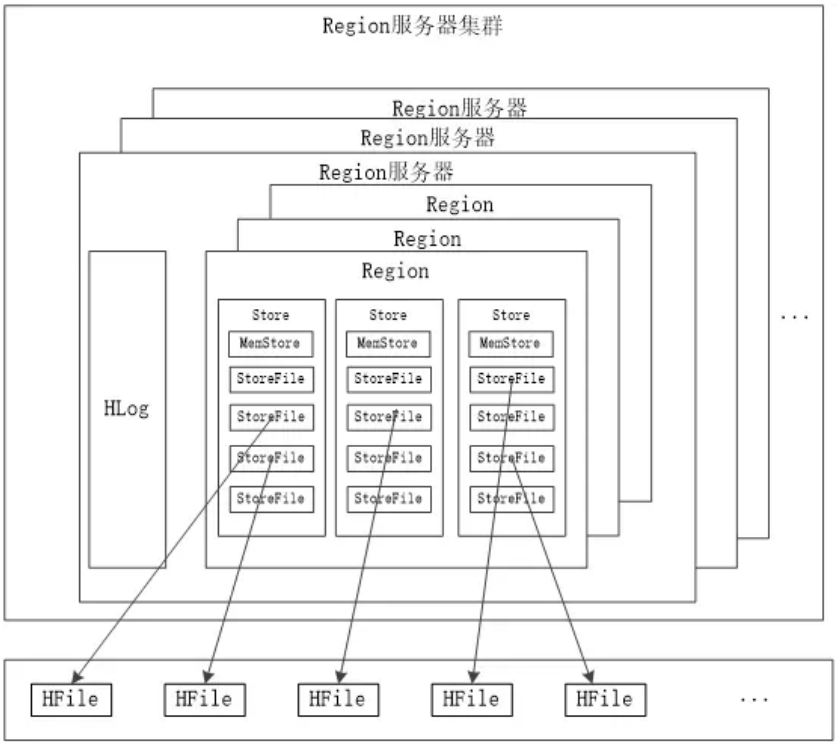
主服务器Master主要负责表和Region的管理工作：(怎么选主???)

- 管理用户对表的增加、删除、修改、查询等操作
- 实现不同Region服务器之间的负载均衡
- 在Region分裂或合并后，负责重新调整Region的分布
- 对发生故障失效的Region服务器上Region进行迁移

Region服务器

Region服务器是Hbase中最核心的模块，负责维护分配给自己的Region，并响应用户的读写请求。

Region服务器工作原理



Region服务器向HDFS文件系统中读写数据过程：

- 1、用户读写数据过程

读：

① 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。

② 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。

③ 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。

④ 最后 HRegionServer 把查询到的数据响应给 Client。

写：

① Client 先访问 zookeeper，找到 Meta 表，并获取 Meta 表元数据。

② 确定当前将要写入的数据所对应的 HRegion 和 HRegionServer 服务器。

③ 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。

④ Client 先把数据写入到 HLog，以防止数据丢失。

⑤ 然后将数据写入到 Memstore。

⑥ 如果 HLog 和 Memstore 均写入成功，commit()则这条数据写入成功（只有commit后,数据就算存下来了,毕竟写日志了嘛,,服务器恢复时,会回访日志,进行数据恢复,这叫WAL(预写式日志)）

⑦ 如果 Memstore 达到阈值，会把 Memstore 中的数据 flush 到 Storefile 中。

⑧ 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。

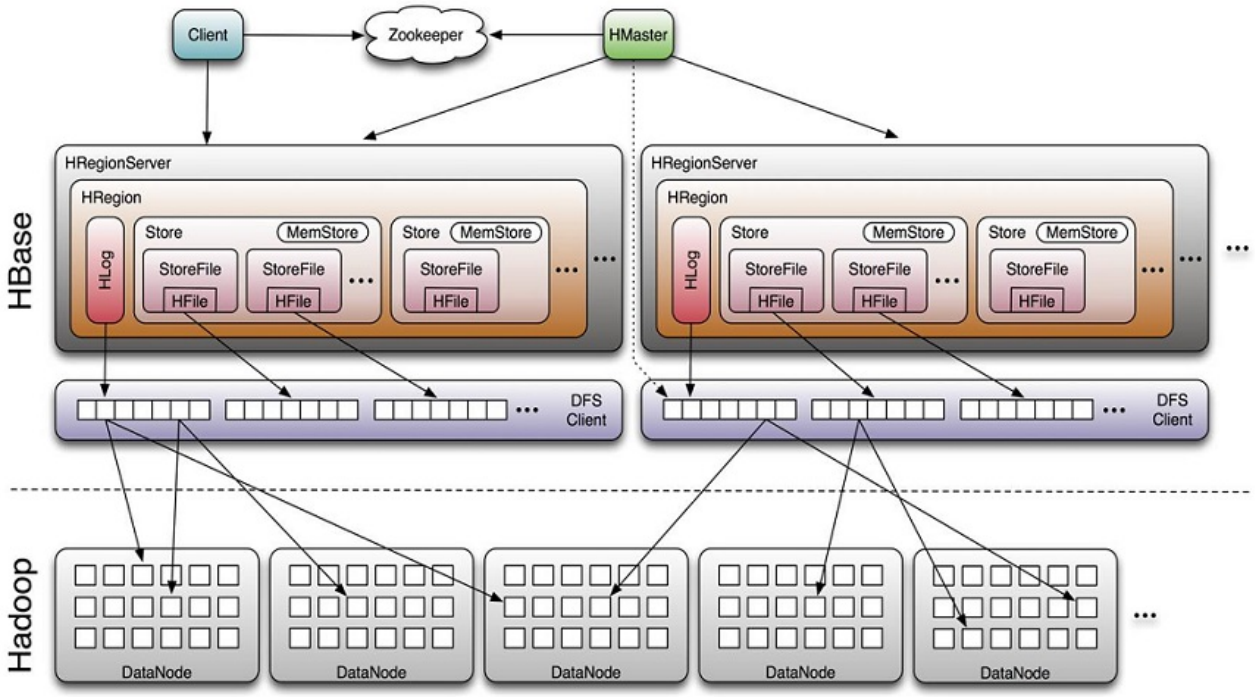
⑨ 当 Storefile 越来越大，Region 也会越来越大，达到阈值后，会触发 Split 操作，将 Region 一分为二。

> 原文链接：https://blog.csdn.net/shujuelin/article/details/89035272

- 2、缓存的刷新
  - 当MemStore缓存里的内容达到阈值时则会刷写到磁盘的StoreFile文件中，清空缓存，并在Hlog里面写入一个标记
  - 每次刷写都生成一个新的StoreFile文件，因此，每个Store包含多个StoreFile文件
  - 每个Region服务器都有一个自己的HLog 文件，每次启动都检查该文件，确认最近一次执行缓存刷新操作之后是否发生新的写入操作；如果发现更新，则先写入MemStore，再刷写到StoreFile，最后删除旧的Hlog文件，开始为用户提供服务。(这也是数据恢复(预写式日志))
- 3、StoreFile的合并
  - 每次刷写都生成一个新的StoreFile，数量太多，影响查找速度、
  - 调用Store.compact()把多个合并成一个
  - 合并操作比较耗费资源，只有数量达到一个阈值才启动合并

> 来自: https://www.jianshu.com/p/53864dc3f7b4

架构图：



> 来自 <http://www.tianshouzhi.com/api/tutorials/hbase>
> https://www.cnblogs.com/raphael5200/p/5229164.html

4. 二级索引

4.1 前言

数据库查询可简单分解为两个步骤：1）键的查找；2） 数据的查找

因这两种数据组织方式的不同，在RDBMS领域有两种常见的数据组织表结构：

索引组织表：键与数据存放在一起，查找到键所在的位置则意味着查找到数据本身。

堆表：键的存储与数据的存储是分离的。查找到键的位置，只能获取到数据的物理地址，还需要基于该地址去获取数据。

HBase数据表其实是一种**索引组织表结构**：查找到RowKey所在的位置则意味着找到数据本身。因此，**RowKey本身就是一种索引**。

一种业务模型的用户数据RowKey，只能采用单一结构设计。但事实上，查询场景可能是多纬度的。例如，在上面的场景基础上，还需要单独基于Phone列进行查询。这是HBase二级索引出现的背景。即，二级索引是为了让HBase能够提供更多纬度的查询能力。

4.2 基于Coprocessor方案

从0.94版本，HBase官方文档已经提出了HBase上面实现二级索引的一种路径：

- 基于Coprocessor（0.92版本引入，达到支持类似传统RDBMS的触发器的行为）。
- 开发自定义数据处理逻辑，采用数据“双写”策略，在有数据写入同时同步到二级索引表。

Apache Phoenix：功能围绕SQL On HBase，支持和兼容多个hbase版本，二级索引只是其中一块功能。二级索引的创建和管理直接有SQL语法支持，适用起来简便，该项目目前社区活跃度和版本更新迭代情况都比较好。

Apache Phoenix在目前开源的方案中，是一个比较优的选择，主打SQL On HBase，基于SQL能完成HBase的CRUD操作，支持JDBC协议。

Phoenix二级索引特点：

- Covered Indexes（覆盖索引）：把关注的数据字段也附在索引表上，只需要通过索引表就能返回所要查询的数据（列），所以索引的列必须包含所需查询的列（SELECT的列和WHERE的列）。
- Functional Indexes（函数索引）：索引不局限于列，支持任意的表达式来创建索引。
- Global Indexes（全局索引）：适用于读多写少场景。通过维护全局索引表，所有的更新和写操作都会引起索引的更新，写入性能受到影响。在读数据时，Phoenix SQL会基于索引字段，执行快速查询。
- Local Indexes（本地索引）：适用于写多读少场景。在数据写入时，索引数据和表数据都会存储在本地。在数据读取时，由于无法预先确定region的位置，所以在读取数据时需要检查每个region（以找到索引数据），会带来一定性能（网络）开销。

4.3 非Coprocessor方案

选择不基于Coprocessor开发，自行在外部构建和维护索引关系也是另外一种方式。
常见的是采用底层基于Apache Lucene的ElasticSearch（下面简称ES）或Apache Solr，来构建强大的索引能力、搜索能力，例如支持模糊查询、全文检索、组合查询、排序等。

下面显示了数说基于ES做二级索引的两种构建流程，包含：

- 增量索引：日常持续接入的数据源，进行增量的索引更新
- 全量索引：配套基于Spark/MR的批量索引创建/更新程序，用于初次或重建已有HBase库表的索引。

数据查询流程：