

HBase

2019年7月3日 10:43

HBASE介绍

2016年1月23日 10:48

一、HBASE概述



官方网址: <http://hbase.apache.org/>

HBase是一个**分布式的、面向列**的开源数据库，该技术来源于Fay Chang所撰写的Google论文《Bigtable》一个结构化数据的**分布式存储系统**。就像Bigtable利用了Google文件系统(File System)所提供的分布式数据存储一样，HBase在Hadoop之上提供了类似于Bigtable的能力（**低延迟的数据查询能力**）。HBase是Apache的Hadoop项目的子项目。HBase不同于一般的关系数据库，Hbase同BigTable一样，都是NoSQL数据库，即非关系型数据库，此外，HBase和BigTable一样，**是基于列**的而不是基于行的模式。

非关系型数据库和关系型数据库

传统关系型数据库的缺陷

随着互联网Web 2.0的兴起，传统的关系数据库在应付Web 2.0网站，特别是超大规模和高并发的SNS类型动态网站时已经力不从心，暴露了很多难以克服的问题。

1) 高并发读写的瓶颈

Web 2.0网站要根据用户个性化信息来实时生成动态页面和提供动态信息，所以基本上无法使用静态化技术，因此数据库并发负载非常高，可能峰值会达到每秒上万次读写请求。关系型数据库应付上万次SQL查询还勉强顶得住，但是应付上万次SQL写数据请求，硬盘I/O却无法承受。其实对于普通的BBS网站，往往也存在相对高并发写请求的需求，例如，人人网的实时统计在线用户状态，记录热门帖子的点击次数，投票计数等，这是一个相当普遍的业务需求。

2) 可扩展性的限制

在基于Web的架构中，数据库是最难以进行横向扩展的，当应用系统的用户量和访问量与日俱增时，数据库系统却无法像Web Server和App Server那样简单地通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供24小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移，而不能通过横向添加节点的方式实现无缝扩展。

3) 事务一致性的负面影响

事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。保证数据库一致性是指当事务完成时，必须使所有数据都具有一致的状态。在关系型数据库中，所有的规则必须应用到事务的修改上，以便维护所有数据的完整性，这随之而来的是性能的大幅度下降。很多Web系统并不需要严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求也不高。因此数据库事务管理成了高负载下的一个沉重负担。

4) 复杂SQL查询的弱化

任何大数据量的Web系统都非常忌讳几个大表间的关联查询，以及复杂的数据分析类型的SQL查询，特别是SNS类型的网站，从需求以及产品设计角度就避免了这

种情况的产生。更多的情况往往只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大地弱化了，所以这部分功能不能得到充分发挥。

NoSQL数据库的优势

1) 扩展性强

NoSQL数据库种类繁多，但是一个共同的特点就是去掉关系型数据库的关系特性，若数据之间是弱关系，则非常容易扩展。一般来说，NoSql数据库的数据结构都是Key-Value字典式存储结构。例如，HBase、Cassandra等系统的水平扩展性能非常优越，非常容易实现支撑数据从TB到PB级别的过渡。

2) 并发性能好

NoSQL数据库具有非常良好的读写性能，尤其在大数据量下，同样表现优秀。当然这需要有优秀的数据结构和算法做支撑。

3) 数据模型灵活

NoSQL无须事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系型数据库中，增删字段是一件非常麻烦的事情。对于数据量非常大的表，增加字段简直就是一场噩梦。NoSQL允许使用者随时随地添加字段，并且字段类型可以是任意格式。

总结

HBase作为NoSQL数据库的一种，当然也具备上面提到的种种优势。

Hadoop最适合的应用场景是离线批量处理数据，其离线分析的效率非常高，Hadoop针对数据的吞吐量做了大量优化，能在分钟级别处理TB级的数据，但是Hadoop不能做到低延迟的数据访问，所以一般的应用系统并不适合批量模式访问，更多的还是用户的随机访问，就类似访问关系型数据库中的某条记录一样。比如Google这个搜索引擎，存储了海量的网页数据，当我们通过搜索引擎检索一个网页时，之所以Google能够快速响应结果，核心的技术就是利用了BigTable，可以实现低延迟的数据访问。

HBase的列式存储的特性支撑它实时随机读取、基于KEY的特殊访问需求。当然，HBase还有不少新特性，其中不乏有趣的特性，在接下来的内容中将会详细介绍。

BigTable介绍

BigTable是Google设计的分布式数据存储系统，用来处理海量的数据的一种非关系型的数据库。

BigTable是非关系的数据库,是一个稀疏的、分布式的、持久化存储的多维度排序Map。Bigtable的设计目的是可靠的处理PB级别的数据，并且能够部署到上千台机器上。Bigtable已经实现了下面的几个目标：适用性广泛、可扩展、高性能和高可用性。Bigtable已经在超过60个Google的产品和项目上得到了应用，包括 Google Analytics、Google Finance、Orkut、Personalized Search、Writely和Google Earth。这些产品对Bigtable提出了迥异的需求，有的需要高吞吐量的批处理，有的则需要及时响应，快速返回数据给最终用户。它们使用的Bigtable集群的配置也有很大的差异，有的集群只有几台服务器，而有的则需要上千台服务器、存储几百TB的数据。

Bigtable的用三维表来存储数据，分别是行键（row key）、列键（column key）和时间戳（timestamp），

本质上说，Bigtable是一个键值（key-value）映射。按作者的说法，Bigtable是一个稀疏的，分布式的，持久化的，多维的排序映射。可以用(row:string, column:string, time:int64)→string 来表示一条键值对记录。

行存储 VS 列存储

2017年11月29日 19:04

概述

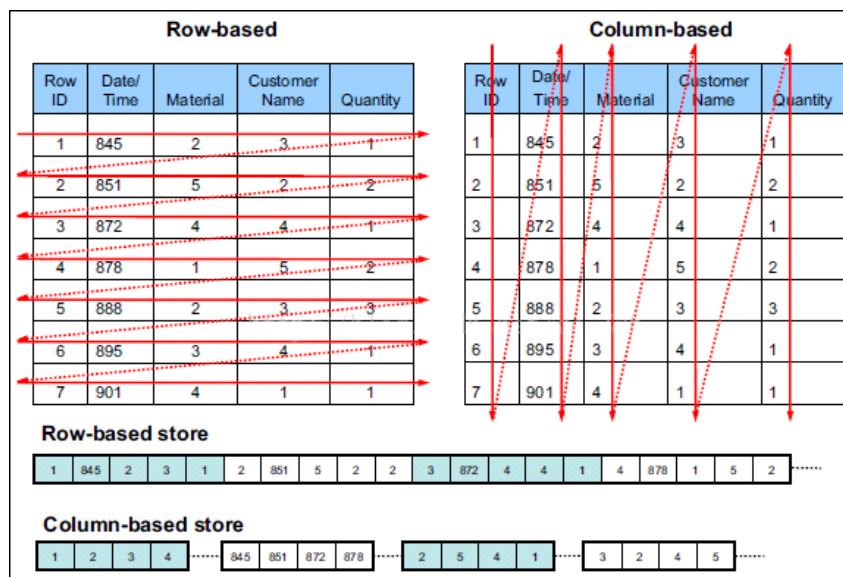
目前大数据存储有两种方案可供选择：行存储（Row-Based）和列存储（Column-Based）。业界对两种存储方案有很多争持，集中焦点是：谁能够更有效地处理海量数据，且兼顾安全、可靠、完整性。从目前发展情况看，关系数据库已经不适应这种巨大的存储量和计算要求，基本是淘汰出局。在已知的几种大数据处理软件中，**Hadoop的HBase采用列存储，MongoDB是文档型的行存储，Lexst是二进制型的行存储。**

什么是列存储？

列式存储(column-based)是相对于传统关系型数据库的行式存储(Row-based storage)来说的。简单来说两者的区别就是如何组织表：

Ø Row-based storage stores a table in a sequence of rows.

Ø Column-based storage stores a table in a sequence of columns.



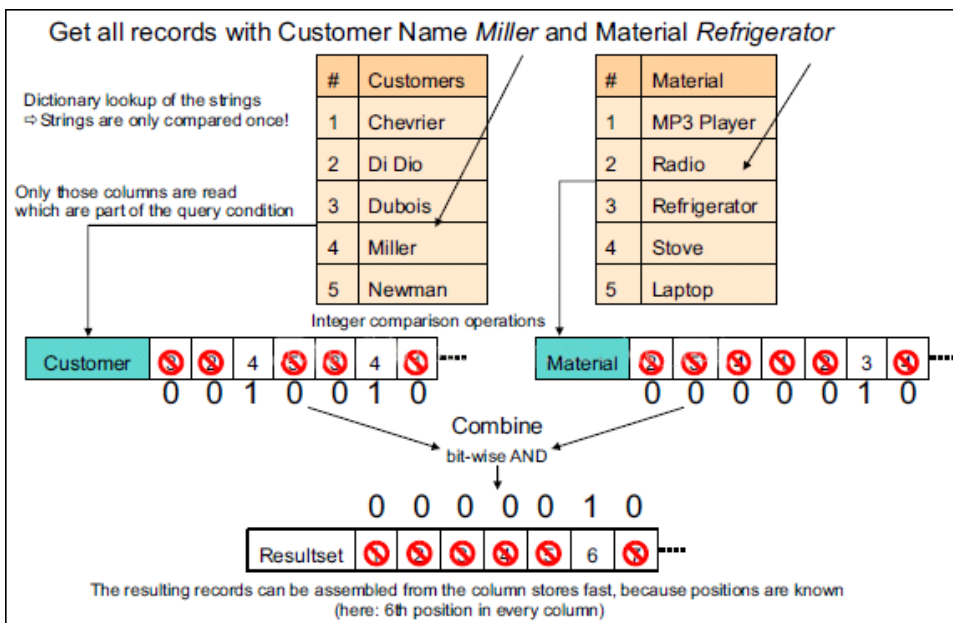
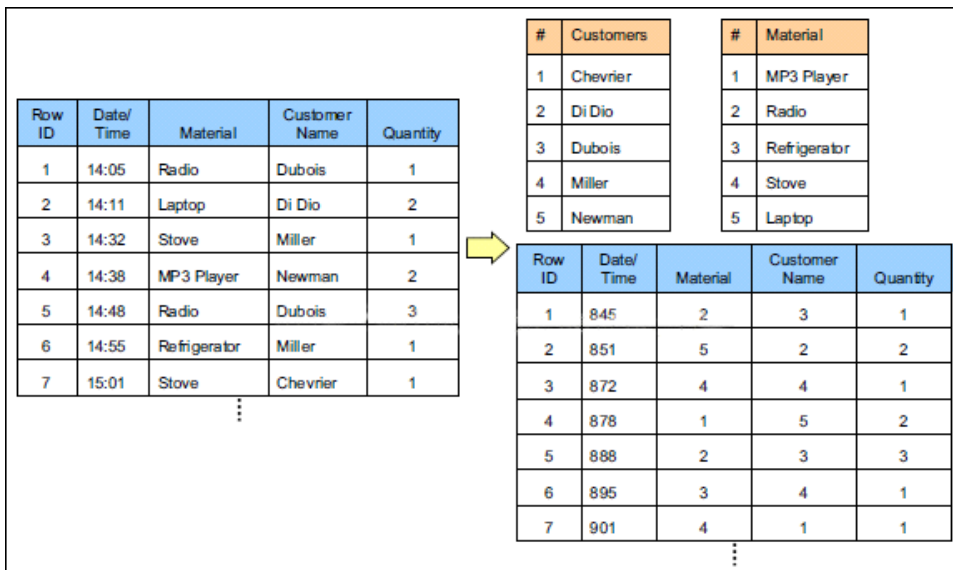
从上图可以很清楚地看到，行式存储下一张表的数据都是放在一起的，但列式存储下都被分开保存了。所以它们就有了如下这些优缺点对比：

在数据写入上的对比

- 1) 行存储的写入是一次完成。如果这种写入建立在操作系统的文件系统上，可以保证写入过程的成功或者失败，数据的完整性因此可以确定。
- 2) 列存储由于需要把一行记录拆分成单列保存，**写入次数明显比行存储多（意味着磁头调度次数多，而磁头调度是需要时间的，一般在1ms~10ms）**，再加上磁头需要在盘片上移动和定位花费的时间，实际时间消耗会更大。所以，行存储在写入上占有很大的优势。
- 3) 还有数据修改,这实际也是一次写入过程。不同的是，数据修改是对磁盘上的记录做删除标记。行存储是在指定位置写入一次，列存储是将磁盘定位到多个列上分别写入，这个过程仍是行存储的列数倍。所以，数据修改也是以行存储占优。

在数据读取上的对比

- 1) 数据读取时，行存储通常将一行数据完全读出，如果只需要其中几列数据的情况，就会存在冗余列，出于缩短处理时间的考量，消除冗余列的过程通常是在内存中进行的。
- 2) 列存储每次读取的数据是集合的一段或者全部，不存在冗余性问题。
- 3) 两种存储的数据分布。由于列存储的每一列数据类型是同质的，不存在二义性问题。比如说某列数据类型为整型(int)，那么它的数据集合一定是整型数据。这种情况使数据解析变得十分容易。相比之下，行存储则要复杂得多，因为在一行记录中保存了多种类型的数据，数据解析需要在多种数据类型之间频繁转换，这个操作很消耗CPU，增加了解析的时间。所以，列存储的解析过程更有利于分析大数据。
- 4) 从数据的压缩以及更性能的读取来对比



优缺点

显而易见，两种存储格式都有各自的优缺点：

- 1) 行存储的写入是一次性完成，消耗的时间比列存储少，并且能够保证数据的完整性，缺点是数据读取过程中会产生冗余数据，如果只有少量数据，此影响可以忽略；数量大可能会影响到数据的处理效率。
- 2) 列存储在写入效率、保证数据完整性上都不如行存储，它的优势是在读取过程，不会产生冗余数据，这对数据完整性要求不高的大数据处理领域，比如互联网，尤为重要。

两种存储格式各自的特性都决定了它们的使用场景。

列存储的适用场景

- 1) 一般来说，一个OLAP类型的查询可能需要访问几百万甚至几十亿个数据行，且该查询往往只关心少数几个数据列。例如，查询今年销量最高的前20个商品，这个查询只关心三个数据列：时间（date）、商品（item）以及销售量（sales amount）。商品的其他数据列，例如商品URL、商品描述、商品所属店铺，等等，对这个查询都是没有意义的。

而列式数据库只需要读取存储着“时间、商品、销量”的数据列，而行式数据库需要读取所有的数据列。因此，列式数据库大大地提高了OLAP大数据量查询的效率

OLTP OnLine Transaction Processor 在线联机事务处理系统（比如Mysql，Oracle等产品）

OLAP OnLine Analysier Processor 在线联机分析处理系统（比如Hive Hbase等）

表 4.1 OLTP 系统与 OLAP 系统的比较

特征	OLTP	OLAP
特性	操作处理	信息处理
面向	事务	分析
用户	办事员、DBA、数据库专业人员	知识工人（如经理、主管、分析人员）
功能	日常操作	长期信息需求、决策支持
DB 设计	基于 E-R，面向应用	星形/雪花、面向主题
数据	当前的、确保最新	历史的、跨时间维护
汇总	原始的、高度详细	汇总的、统一的
视图	详细、一般关系	汇总的、多维的
工作单元	短的、简单事务	复杂查询
访问	读/写	大多为读
关注	数据进入	信息输出
操作	主码上索引/散列	大量扫描
访问记录数量	数十	数百万
用户数	数千	数百
DB 规模	GB 到高达 GB	≥TB
优先	高性能、高可用性	高灵活性、终端用户自治
度量	事务吞吐量	查询吞吐量、响应时间

注：该表部分基于 Chaudhuri 和 Dayal[CD97]。

2) 很多列式数据库还支持列族（column group，Bigtable系统中称为locality group），即将多个经常一起访问的数据列的各个值存放在一起。如果读取的数据列属于相同的列族，列式数据库可以从相同的地方一次性读取多个数据列的值，避免了多个数据列的合并。列族是一种行列混合存储模式，这种模式能够同时满足OLTP和OLAP的查询需求。

3) 此外，由于同一个数据列的数据重复度很高，因此，列式数据库压缩时有很大的优势。

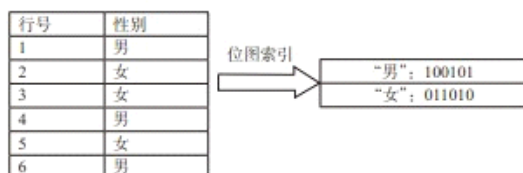
例如，Google Bigtable列式数据库对网页库压缩可以达到15倍以上的压缩率。另外，可以针对列式存储做专门的索引优化。比如，性别列只有两个值，“男”和“女”，可以对这一列建立位图索引：

如下图所示

“男”对应的位图为100101，表示第1、4、6行值为“男”

“女”对应的位图为011010，表示第2、3、5行值为“女”

如果需要查找男性或者女性的个数，只需要统计相应的位图中1出现的次数即可。另外，建立位图索引后0和1的重复度高，可以采用专门的编码方式对其进行压缩。



当然，如果每次查询涉及的数据量较小或者大部分查询都需要整行的数据，列式数据库并不适用。

最后总结如下

传统行式数据库的特性如下：

- ①数据是按行存储的。
- ②没有索引的查询使用大量I/O。比如一般的数据库表都会建立索引，通过索引加快查询效率。

③建立索引和物化视图需要花费大量的时间和资源。

④面对查询需求，数据库必须被大量膨胀才能满足需求。

列式数据库的特性如下：

①数据按列存储，即每一列单独存放。

②数据即索引。

③只访问查询涉及的列，可以大量降低系统I/O。

④每一列由一个线程来处理，即查询的并发处理性能高。

⑤数据类型一致，数据特征相似，可以高效压缩。比如有增量压缩、前缀压缩算法都是基于列存储的类型定制的，所以可以大幅度提高压缩比，有利于存储和网络输出数据带宽的消耗。

HBASE单机安装

2016年1月23日 15:36

单机模式安装

特点：不依赖于Hadoop的HDFS，配置完既可使用，好处是便于测试。坏处是不具备分布式存储数据的能力。

安装配置

1.安装JDK及配置环境变量

2.上传解压Hbase安装包

3.修改Hbase的配置文件，（修改安装目录下的conf/hbase-site.xml）

□ 配置示例：

```
<property>
    <name>hbase.rootdir</name>
    <value>file:///home/software/hbase/tmp</value>
</property>
```

这个是配置hbase存储数据的目录，如果不配置，默认是放在Linux的/tmp目录下。

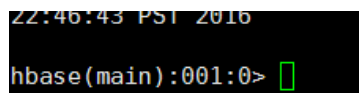
4.启动hbase,进入bin目录

执行：sh start-hbase.sh

然后通过jps查看是否有HMaster进程，如果有，证明hbase启动成功

5.在bin目录执行：

./hbase shell 进入shell客户端操作hbase



```
22:46:43 PST 2016
hbase(main):001:0> █
```

HBASE完全分布式安装

2016年1月25日 9:41

实现步骤

- 1.准备三台虚拟机, 01作为主节点, 02、03作为从节点。(把每台虚拟机防火墙都关掉, 配置免密码登录, 配置每台的主机名和hosts文件。)
- 2.01节点上安装和配置: Hadoop+Hbase+JDK+Zookeeper
- 3.02、03节点上安装和配置: Hbase+JDK+Zookeeper
- 4.修改conf/hbase-env.sh

配置示例:

```
#修改JAVA_HOME
export JAVA_HOME=xxxx
#修改Zookeeper和Hbase的协调模式, hbase默认使用自带的zookeeper, 如果需要使用外部zookeeper, 需要先关闭。
export HBASE_MANAGES_ZK=false
```

- 5.修改hbase-site.xml, 配置开启完全分布式模式

配置示例:

```
<property>
  <name>hbase.rootdir</name>
  <value>hdfs://hadoop01:9000/hbase</value>
</property>
<property>
  <name>hbase.cluster.distributed</name>
  <value>true</value>
</property>
#配置Zookeeper的连接地址与端口号
<property>
  <name>hbase.zookeeper.quorum</name>
  <value>hadoop01:2181,hadoop02:2181,hadoop03:2181</value>
</property>
```

- 6.配置region服务器,修改conf/regionservers文件,每个主机名独占一行, hbase启动或关闭时会按照该配置顺序启动或关闭主机中的hbase

配置示例:

```
hadoop01
hadoop02
hadoop03
```

- 7.将01节点配置好的hbase通过远程复制拷贝到02,03节点上
- 8.启动01,02,03的Zookeeper服务
- 9.启动01节点的Hadoop
- 10.启动01节点的Hbase, 进入到hbase安装目录下的bin目录
执行: sh start-hbase.sh
- 11.查看各节点的java进程是否正确
- 12.通过浏览器访问http://xxxxx:60010来访问web界面, 通过web界面管理hbase
- 13.关闭Hmaster,进入到hbase安装目录下的bin目录
执行: stop-hbase.sh
- 14.关闭regionserver, 进入到hbase安装目录下的bin目录
执行: sh hbase-daemon.sh stop regionserver

注: HBASE配置文件说明

hbase-env.sh配置HBase启动时需要的相关环境变量

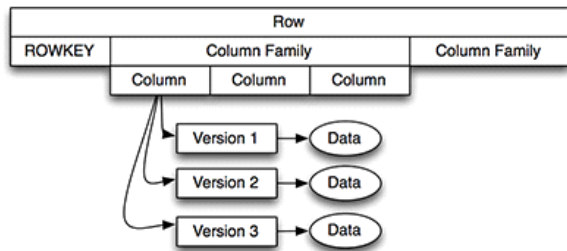
hbase-site.xml配置HBase基本配置信息

HBASE启动时默认使用hbase-default.xml中的配置, 如果需要可以修改hbase-site.xml文件, 此文件中的配置将会覆盖hbase-default.xml中的配置
修改配置后要重启hbase才会起作用

HBASE基本概念

2016年1月25日 9:41

HBase以表的形式存储数据。表有行和列族组成。列族划分为若干个列



1) Row Key

hbase本质上也是一种Key-Value存储系统。Key相当于RowKey，Value相当于列族数据的集合。

与nosql数据库们一样,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：

- 1 通过单个row key访问
- 2 通过row key的range
- 3 全表扫描

Row key行键 (Row key)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。

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, ..., 9, 91, 92, 93, 94, 95, 96, 97, 98, 99。要保持整形的自然序，行键必须用0作左填充。

2) 列族 (列族)

hbase表中的每个列，都归属某个列族。列族是表的schema的一部分(而不是)，列族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。实际应用中，列族上的控制权限能帮助我们管理不同类型的应用：我们允许一些应用可以添加新的基本数据、一些应用可以读取基本数据并创建继承的列族、一些应用则只允许浏览数据（甚至可能因为隐私的原因不能浏览所有数据）。

3) cell与时间戳

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。

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。一是保存数据的最后n个版本，二是保存最近一段时间内的版本（比如最近七天）。用户可以针对每个列族进行设置。

HBASE基础指令

2016年1月23日 17:08

常用命令

指令	说明	示例
create	创建表, t1指表名, c1, c2 列族名	create 'tab1','colfamily1','colfamily2'
list	查看一共有哪些表	list
put	t1指表名, r1指行键名, c1指列名, value指单元格值。ts1指时间戳, 一般都省略掉了。注意, 行键名在一张表里要全局唯一	put 'tab1','row-1','colfamily1:co11','aaa' put 'tab1','row-1','colfamily1:co12','bbb' put 'tab1','row-1','colfamily2:co11','ccc' put 'tab1','row-1','colfamily2:co12','ddd'
get	根据表名和行键查询	get 'tab1','row-1' get 'tab1','row-1','colfamily1' get 'tab1','row-1','colfamily1','colfamily2' get 'tab1','row-1','colfamily1:co11'
scan	扫描所有数据, 也可以跟指定条件	scan 'tab1' #扫描整表数据, 会查询出所有的行数据 scan 'tab1',{COLUMNS=>['colfamily1']} scan 'tab1',{COLUMNS=>['cf1:name']} scan 'tab1',{COLUMNS=>['cf1:name','cf2:salary']} scan 'tab1',{COLUMNS=>['colfamily1','colfamily2']} scan 'tab1',{RAW=>true,VERSIONS=>3} 可以在查询时加上RAW=>true来开启对历史版本数据的查询, VERSIONS=>3指定查询最新的几个版本的数据
deleteall	根据表名、行键删除整行数据	deleteall 'tab1','row-1'
drop	删除表, 前提是先禁用表	drop 'tab1'
disable	禁用表	disable 'tab1'
create 指令补充	建表时可以指定VERSIONS, 配置的是当前列族在持久化到文件系统中时, 要保留几个最新的版本数据, 这并不影响内存中的历史数据版本	create 'tab1',{NAME=>'cf1',VERSIONS=>3},{NAME=>'cf2',VERSIONS=>3}
exit	推出shell客户端	

补充说明:

hbase命令行下不能使用删除,
可以使用 ctrl+删除键 来进行删除
修改xshell配置:
文件->属性->终端->键盘

->delete键序列[VT220Del]
->backspace键序列[ASCII127]



HBASE API

2016年1月25日 9:41

实现步骤:

1.导入开发包将hbase安装包中lib下包导入java项目

创建表:

```
@Test
public void testCreateTable() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HBaseAdmin admin = new HBaseAdmin(conf);
    //指定表名
    HTableDescriptor tab1=new HTableDescriptor(TableName.valueOf("tab1"));
    //指定列族名
    HColumnDescriptor colfam1=new HColumnDescriptor("colfam1".getBytes());
    HColumnDescriptor colfam2=new HColumnDescriptor("colfam2".getBytes());
    //指定历史版本存留上限
    colfam1.setMaxVersions(3);

    tab1.addFamily(colfam1);
    tab1.addFamily(colfam2);
    //创建表
    admin.createTable(tab1);

    admin.close();

}
```

插入数据:

```
@Test
public void testInsert() throws Exception{
```

```

Configuration conf=HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum",
        "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");
//尽量复用Htable对象
HTable table=new HTable(conf,"tab1");
Put put=new Put("row-1".getBytes());
//列族, 列,值
put.add("colfam1".getBytes(),"col1".getBytes(),"aaa".getBytes());
put.add("colfam1".getBytes(),"col2".getBytes(),"bbb".getBytes());
table.put(put);
table.close();
}

```

📖 试验：100万条数据的写入

```

@Test
public void testInsertMillion() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HTable table=new HTable(conf,"tab1");

    List<Put> puts=new ArrayList<>();

    long begin=System.currentTimeMillis();

    for(int i=1;i<1000000;i++){
        Put put=new Put(("row"+i).getBytes());
        put.add("colfam1".getBytes(),"col".getBytes(),""+i.getBytes());
        puts.add(put);

        //批处理, 批大小为:10000
        if(i%10000==0){
            table.put(puts);
            puts=new ArrayList<>();
        }
    }

    long end=System.currentTimeMillis();
}

```



```

        System.out.println(end-begin);
    }

```

获取数据：

```

@Test
public void testGet() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HTable table=new HTable(conf,"tab1");
    Get get=new Get("row1".getBytes());
    Result result=table.get(get);
    byte[] col1_result=result.getValue("colfam1".getBytes(),"col".getBytes());
    System.out.println(new String(col1_result));
    table.close();
}

```

获取数据集：

```

@Test
public void testScan() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HTable table = new HTable(conf,"tab1");
    //获取row100及以后的行键的值
    Scan scan = new Scan("row100".getBytes());
    ResultScanner scanner = table.getScanner(scan);
    Iterator it = scanner.iterator();
    while(it.hasNext()){
        Result result = (Result) it.next();
        byte [] bs = result.getValue(Bytes.toBytes("colfam1"),Bytes.toBytes("col"));
        String str = Bytes.toString(bs);
        System.out.println(str);
    }
}

```

```
table.close();
```

```
}
```

删除数据：

@Test

```
public void testDelete() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HTable table = new HTable(conf,"tab1");
    Delete delete=new Delete("row1".getBytes());
    table.delete(delete);
    table.close();

}
```

删除表

@Test

```
public void testDeleteTable() throws Exception, IOException{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HBaseAdmin admin=new HBaseAdmin(conf);
    admin.disableTable("tab1".getBytes());
    admin.deleteTable("tab1".getBytes());
    admin.close();

}
```

HBASE完全分布式安装

2016年1月25日 9:41

实现步骤

1.准备三台虚拟机, 01作为主节点, 02、03作为从节点。(把每台虚拟机防火墙都关掉, 配置免密码登录, 配置每台的主机名和hosts文件。)

2.01节点上安装和配置: Hadoop+Hbase+JDK+Zookeeper

3.02、03节点上安装和配置: Hbase+JDK+Zookeeper

4.修改conf/hbase-env.sh

配置示例:

#修改JAVA_HOME

export JAVA_HOME=xxxx

#修改Zookeeper和Hbase的协调模式, hbase默认使用自带的zookeeper, 如果需要使用外部zookeeper, 需要先关闭。

export HBASE_MANAGES_ZK=false

5.修改hbase-site.xml, 配置开启完全分布式模式

配置示例:

<property>

<name>hbase.rootdir</name>

<value>**hdfs://hadoop01:9000/hbase**</value>

</property>

<property>

<name>hbase.cluster.distributed</name>

<value>true</value>

</property>

#配置Zookeeper的连接地址与端口号

<property>

<name>hbase.zookeeper.quorum</name>

<value>hadoop01:2181,hadoop02:2181,hadoop03:2181</value>

</property>

6.配置region服务器,修改conf/regionservers文件,每个主机名独占一行, hbase启动或关闭时会按照该配置顺序启动或关闭主机中的hbase

配置示例:

hadoop01

hadoop02

hadoop03

- 7.将01节点配置好的hbase通过远程复制拷贝到02,03节点上
- 8.启动01,02,03的Zookeeper服务
- 9.启动01节点的Hadoop
- 10.启动01节点的Hbase, 进入到hbase安装目录下的bin目录
执行: sh start-hbase.sh
- 11.查看各节点的java进程是否正确
- 12.通过浏览器访问http://xxxxx:60010来访问web界面, 通过web界面管理hbase
- 13.关闭Hmaster,进入到hbase安装目录下的bin目录
执行: stop-hbase.sh
- 14.关闭regionserver, 进入到hbase安装目录下的bin目录
执行: sh hbase-daemon.sh stop regionserver

注: HBASE配置文件说明

hbase-env.sh配置HBase启动时需要的相关环境变量

hbase-site.xml配置HBase基本配置信息

HBASE启动时默认使用hbase-default.xml中的配置, 如果需要可以修改hbase-site.xml文件, 此文件中的配置将会覆盖hbase-default.xml中的配置
修改配置后要重启hbase才会起作用

HBASE API

2016年1月25日 9:41

实现步骤:

1.导入开发包将hbase安装包中lib下包导入java项目

创建表:

```
@Test
public void testCreateTable() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HBaseAdmin admin = new HBaseAdmin(conf);
    //指定表名
    HTableDescriptor tab1=new HTableDescriptor(TableName.valueOf("tab1"));
    //指定列族名
    HColumnDescriptor colfam1=new HColumnDescriptor("colfam1".getBytes());
    HColumnDescriptor colfam2=new HColumnDescriptor("colfam2".getBytes());
    //指定历史版本存留上限
    colfam1.setMaxVersions(3);

    tab1.addFamily(colfam1);
    tab1.addFamily(colfam2);
    //创建表
    admin.createTable(tab1);

    admin.close();

}
```

插入数据:

```
@Test
public void testInsert() throws Exception{
```

```

Configuration conf=HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum",
        "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");
//尽量复用Htable对象
HTable table=new HTable(conf,"tab1");
Put put=new Put("row-1".getBytes());
//列族, 列,值
put.add("colfam1".getBytes(),"col1".getBytes(),"aaa".getBytes());
put.add("colfam1".getBytes(),"col2".getBytes(),"bbb".getBytes());
table.put(put);
table.close();
}

```

试验：100万条数据的写入

```

@Test
public void testInsertMillion() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HTable table=new HTable(conf,"tab1");

    List<Put> puts=new ArrayList<>();

    long begin=System.currentTimeMillis();

    for(int i=1;i<1000000;i++){
        Put put=new Put(("row"+i).getBytes());
        put.add("colfam1".getBytes(),"col".getBytes(),""+i.getBytes());
        puts.add(put);

        //批处理, 批大小为:10000
        if(i%10000==0){
            table.put(puts);
            puts=new ArrayList<>();
        }
    }

    long end=System.currentTimeMillis();
}

```

```

        System.out.println(end-begin);
    }

```

获取数据：

```

@Test
public void testGet() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HTable table=new HTable(conf,"tab1");
    Get get=new Get("row1".getBytes());
    Result result=table.get(get);
    byte[] col1_result=result.getValue("colfam1".getBytes(),"col".getBytes());
    System.out.println(new String(col1_result));
    table.close();
}

```

获取数据集：

```

@Test
public void testScan() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HTable table = new HTable(conf,"tab1");
    //获取row100及以后的行键的值
    Scan scan = new Scan("row100".getBytes());
    ResultScanner scanner = table.getScanner(scan);
    Iterator it = scanner.iterator();
    while(it.hasNext()){
        Result result = (Result) it.next();
        byte [] bs = result.getValue(Bytes.toBytes("colfam1"),Bytes.toBytes("col"));
        String str = Bytes.toString(bs);
        System.out.println(str);
    }
}

```

```
table.close();
```

```
}
```

删除数据：

@Test

```
public void testDelete() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HTable table = new HTable(conf,"tab1");
    Delete delete=new Delete("row1".getBytes());
    table.delete(delete);
    table.close();

}
```

删除表

@Test

```
public void testDeleteTable() throws Exception, IOException{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");

    HBaseAdmin admin=new HBaseAdmin(conf);
    admin.disableTable("tab1".getBytes());
    admin.deleteTable("tab1".getBytes());
    admin.close();

}
```


HBASE过滤器 API

2017年11月24日 20:45

Scanner代码:

@Test

```
public void testScanner() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.234.11:2181,192.168.234.210:2181,192.168.234.211:2181");
    HTable table = new HTable(conf,"tab1".getBytes());
    Scan scan=new Scan();
    scan.setStartRow("row1".getBytes());
    scan.setStopRow("row50".getBytes());

    ResultScanner rs=table.getScanner(scan);

    Result r = null;
    while((r = rs.next())!=null){
        String rowKey=new String(r.getRow());
        String col1Value=new String(r.getValue("colfam1".getBytes(), "col".getBytes()));
        System.out.println(rowKey+"."+col1Value);
    }

}
```

使用扫描器查询数据:

@Test

```
public void scanData() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.150.137:2181,192.168.150.138:2181,192.168.150.139:2181");
    HTable table=new HTable(conf, "tab2".getBytes());

    //--指定扫描行键的起始范围和终止范围
    Scan scan=new Scan();
    scan.setStartRow("row1".getBytes());
    scan.setStopRow("row30".getBytes());

    ResultScanner scanner= table.getScanner(scan);
    //--获取结果的迭代器
    Iterator<Result> it=scanner.iterator();
```

```

while(it.hasNext()){
    Result result=it.next();
    //--通过result对象获取指定列族的列的数据
    byte[] value=result.getValue("colfam1".getBytes(),"col1".getBytes());
    System.out.println(new String(value));
}
scanner.close();
}

```

正则过滤器:

@Test

```

public void scanData() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.150.137:2181,192.168.150.138:2181,192.168.150.139:2181");
    HTable table=new HTable(conf, "tab2".getBytes());

    Scan scan=new Scan();

    //--正则过滤器，匹配行键含3的行数据
    Filter filter=new RowFilter(CompareOp.EQUAL,new RegexStringComparator("^.*3.*$"));
    //--加入过滤器
    scan.setFilter(filter);

    ResultScanner scanner= table.getScanner(scan);
    //--获取结果的迭代器
    Iterator<Result> it=scanner.iterator();
    while(it.hasNext()){
        Result result=it.next();
        //--通过result对象获取指定列族的列的数据
        byte[] value=result.getValue("colfam1".getBytes(),"col1".getBytes());
        System.out.println(new String(value));
    }
    scanner.close();
}

```

行键比较过滤器:

@Test

```

public void scanData() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.150.137:2181,192.168.150.138:2181,192.168.150.139:2181");

```

```

HTable table=new HTable(conf, "tab2".getBytes());

Scan scan=new Scan();

//--行键比较过滤器，下例是匹配小于或等于指定行键的行数据。
Filter filter=new RowFilter(CompareOp.LESS_OR_EQUAL,new BinaryComparator("row90".getBytes()));
scan.setFilter(filter);

ResultScanner scanner= table.getScanner(scan);

Iterator<Result> it=scanner.iterator();
while(it.hasNext()){
    Result result=it.next();
    byte[] value=result.getValue("colfam1".getBytes(),"col1".getBytes());
    System.out.println(new String(value));
}
scanner.close();
}

```

📖 行键前缀过滤器:

@Test

```

public void scanData() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
        "192.168.150.137:2181,192.168.150.138:2181,192.168.150.139:2181");
    HTable table=new HTable(conf, "tab2".getBytes());

    Scan scan=new Scan();

    //--行键前缀过滤器
    Filter filter=new PrefixFilter("row3".getBytes());
    scan.setFilter(filter);

    ResultScanner scanner= table.getScanner(scan);

    Iterator<Result> it=scanner.iterator();
    while(it.hasNext()){
        Result result=it.next();
        byte[] value=result.getValue("colfam1".getBytes(),"col1".getBytes());
        System.out.println(new String(value));
    }
    scanner.close();
}

```

```
}
```

列值过滤器:

```
hbase(main):008:0> scan 'tab3'
ROW COLUMN+CELL
row1 column=cf1:age, timestamp=1538853690448, value=23
row1 column=cf1:name, timestamp=1538853675668, value=rose
row2 column=cf1:age, timestamp=1538853716285, value=25
row2 column=cf1:name, timestamp=1538853708926, value=jary
```

@Test

```
public void scanData() throws Exception{
    Configuration conf=HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum",
            "192.168.150.137:2181,192.168.150.138:2181,192.168.150.139:2181");
    HTable table=new HTable(conf, "tab3".getBytes());

    Scan scan=new Scan();

    //--列值过滤器
    Filter filter = new SingleColumnValueFilter("cf1".getBytes(),"name".getBytes(), CompareOp.EQUAL, "rose".getBytes());

    scan.setFilter(filter);

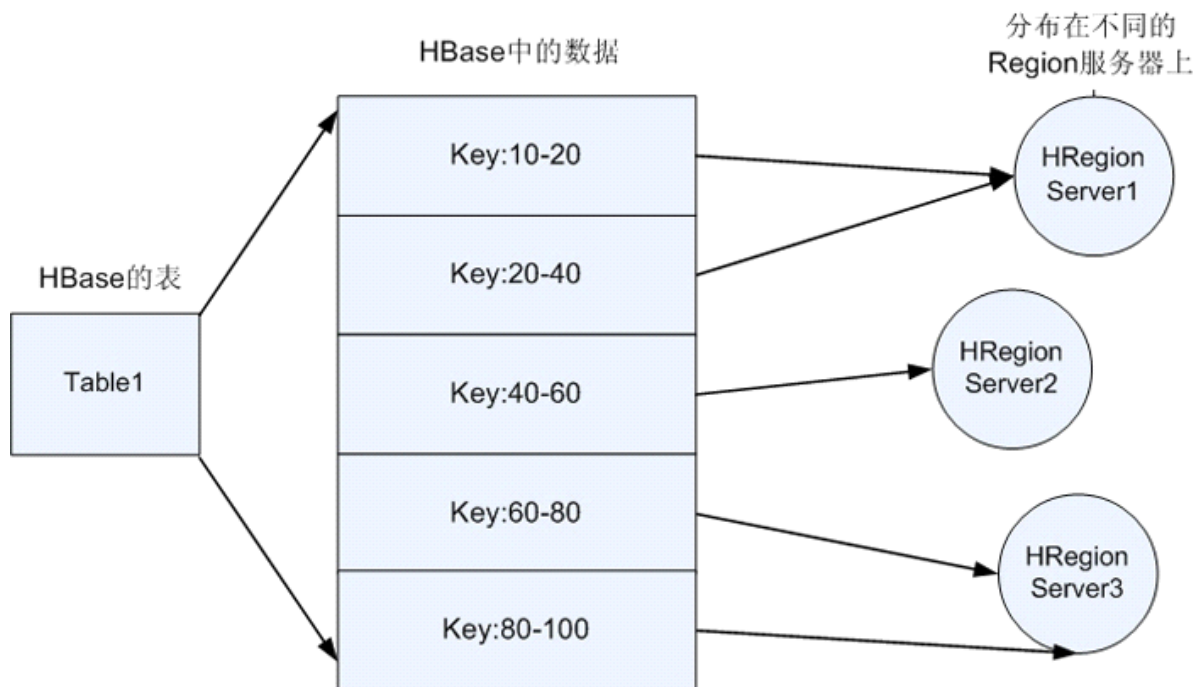
    ResultScanner scanner= table.getScanner(scan);
    //--获取结果的迭代器
    Iterator<Result> it=scanner.iterator();
    while(it.hasNext()){
        Result result=it.next();
        byte[] name=result.getValue("cf1".getBytes(),"name".getBytes());
        byte[] age=result.getValue("cf1".getBytes(),"age".getBytes());
        System.out.println(new String(name)+":"+new String(age));
    }
    scanner.close();
}
```

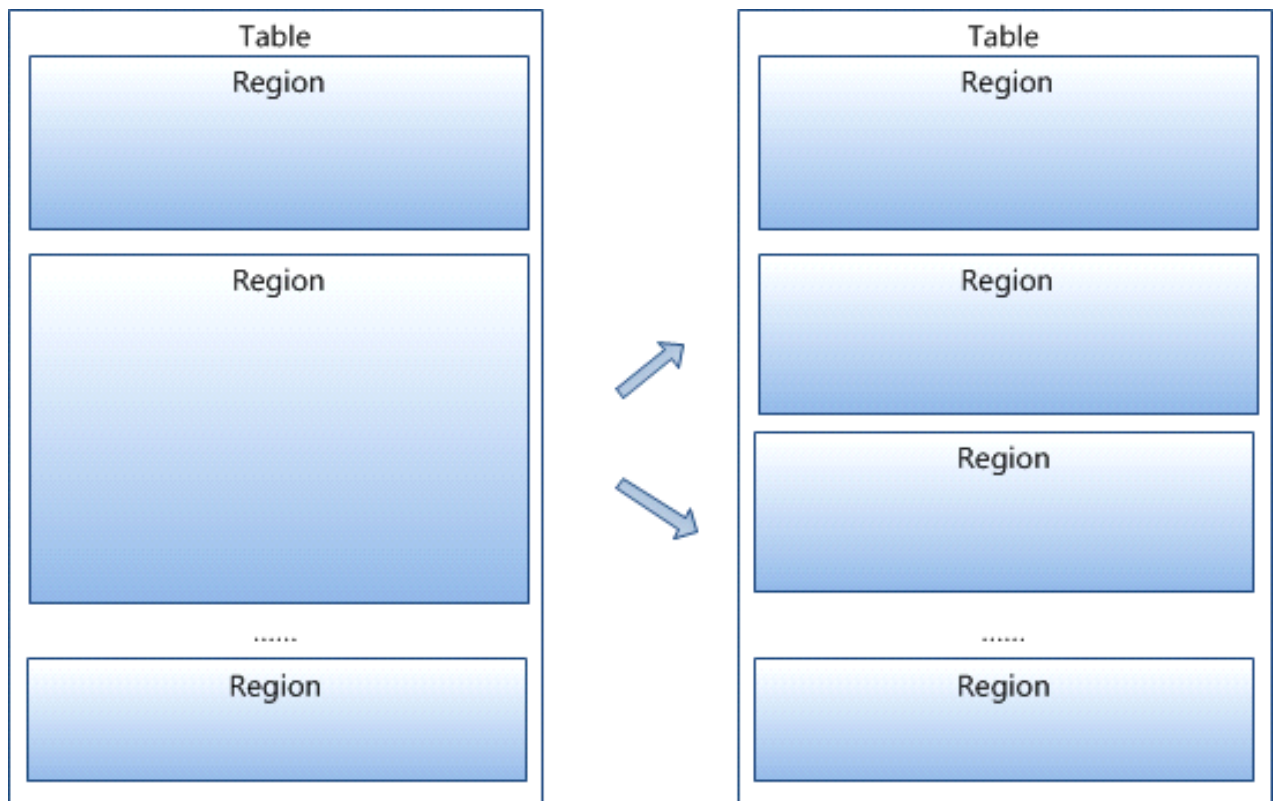
HBASE物理存储原理

Hbase里的一个Table 在行的方向上分割为多个Hregion。

即HBase中一个表的数据会被划分成很多的HRegion, HRegion可以动态扩展并且HBase保证

HRegion的负载均衡。HRegion**实际上是行键排序后的按规则分割的连续的存储空间。**





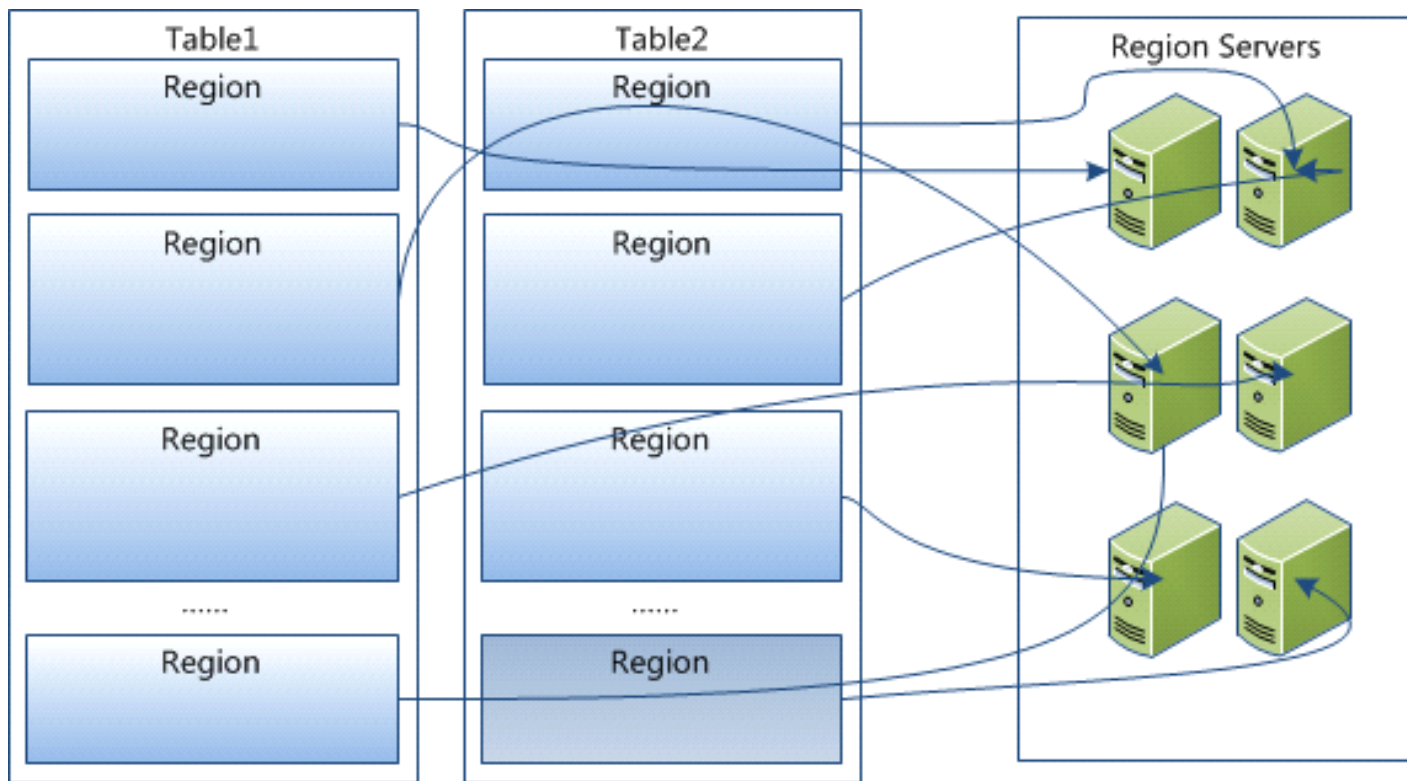
总结：一张Hbase表，可能有多个HRegion，每个HRegion达到一定大小（默认是10GB）时，进行分裂。

拆分流程图：

Hregion是按大小分割的，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，当增大到一个阈值的时候，Hregion就会**等分**两个新的Hregion。当table中的行不断增多，就会有越来越多的Hregion。

按照现在主流硬件的配置，每个HRegion的大小可以是1~20GB。这个大小由hbase.hregion.max.filesize指定，默认为10GB。HRegion的拆分和转移是由HBase（HMaster）自动完成的，用户感知不到。

Hregion是Hbase中分布式存储和负载均衡的最小单元



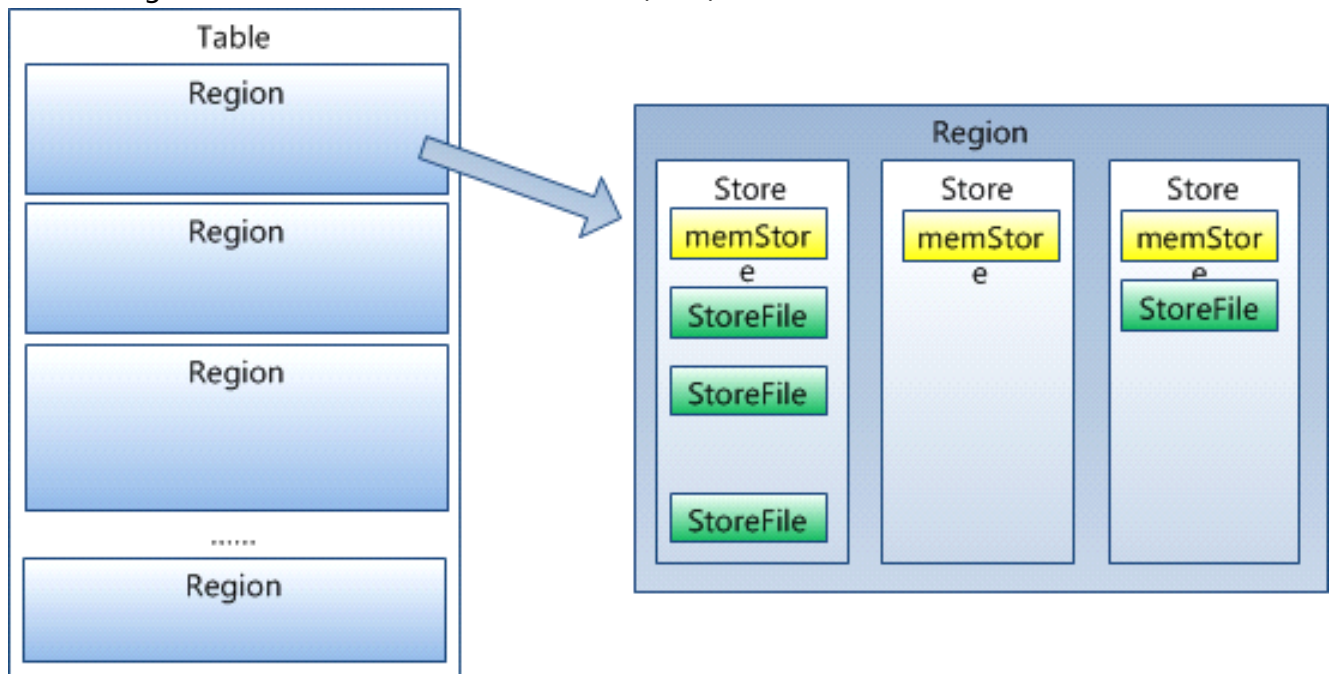
HRegion虽然是分布式存储的最小单元，但并不是存储的最小单元。

事实上，HRegion由一个或者多个HStore组成，每个Hstore保存一个columns family。

- 每个Hstore又由一个memStore和0至多个StoreFile组成。如图：

StoreFile以HFile格式保存在HDFS上。

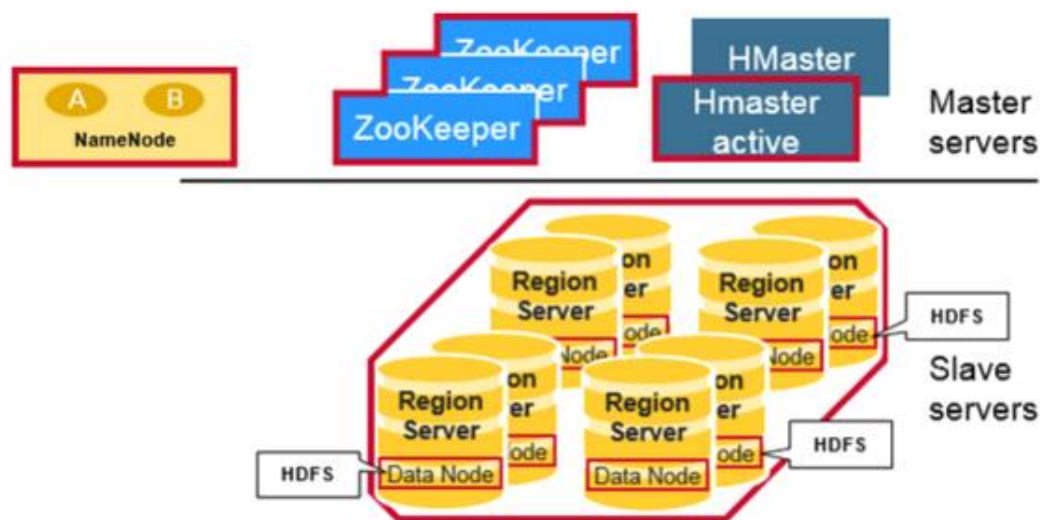
总结：HRegion是分布式的存储最小单位，StoreFile(Hfile)是存储最小单位。



HBASE系统架构

2016年1月28日 13:37

Hbase与Hadoop架构图



HBase架构组成

HBase采用Master/Slave架构搭建集群，它隶属于Hadoop生态系统，由以下类型节点组成：

- 1.HMaster节点
- 2.HRegionServer节点
- 3.ZooKeeper集群
- 4.Hbase的数据存储于HDFS中，因而涉及到HDFS的NameNode、DataNode等。RegionServer和DataNode一般会放在相同的Server上**实现数据的本地化（避免或减少数据在网络中的传输，节省带宽）**。

HMaster节点

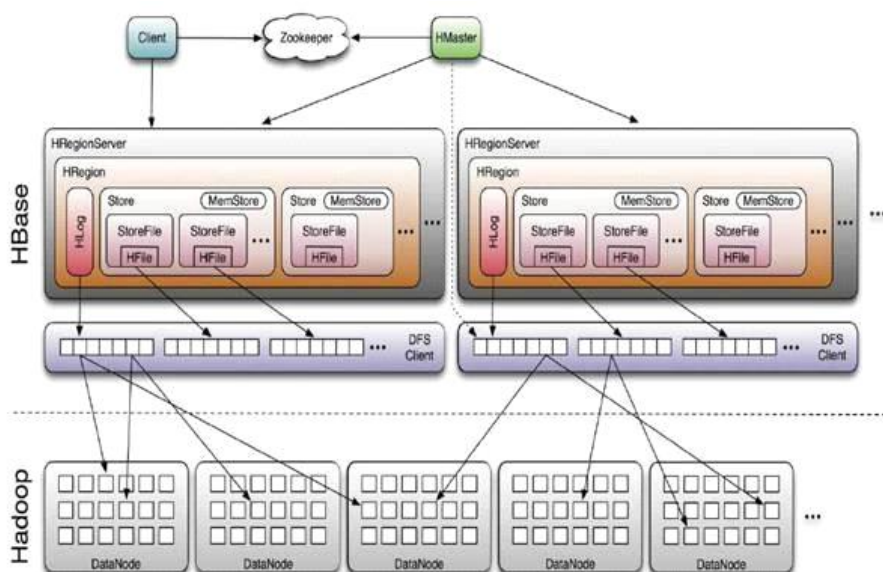
- 1.管理HRegionServer，实现其负载均衡。
- 2.管理和分配HRegion，比如在HRegion split时分配新的HRegion；在HRegionServer退出时迁移其内的HRegion到其他HRegionServer上。
- 3.实现DDL操作（Data Definition Language，namespace和table的增删改，column family的增删改等）。
- 4.管理namespace和table的元数据（实际存储在HDFS上）。
- 5.权限控制（ACL）。

HRegionServer节点

- 1.存放和管理本地HRegion。
- 2.读写HDFS，管理Table中的数据。
- 3.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）。

1. 存放整个 HBase 集群的元数据以及集群的状态信息。以及 RS 服务器的运行状态
2. 实现 HMaster 主备节点的 failover。

下图更全面展示了 Hbase 于 Hadoop 的体系图

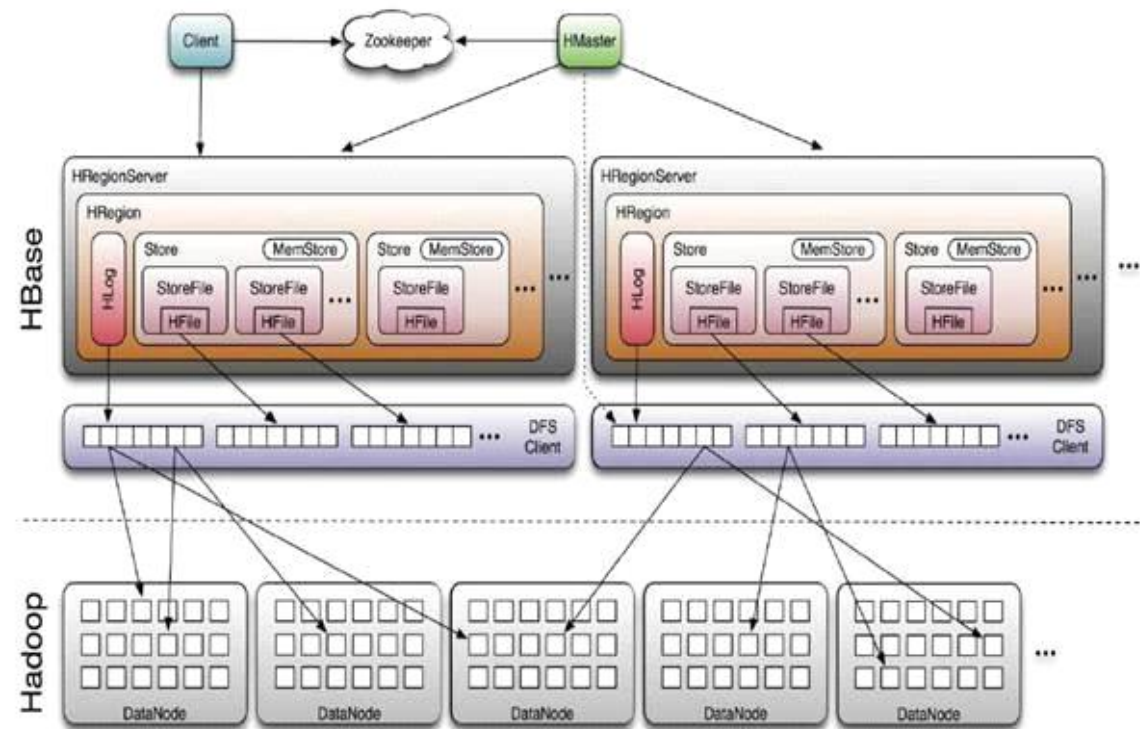


HBase Client通过RPC方式和HMaster、HRegionServer通信；一个HRegionServer可以存放1000个HRegion（1000个数字的由来是来自于Google的Bigtable论文）；底层Table数据存储于HDFS中，而HRegion所处理的数据尽量和数据所在的DataNode在一起，实现数据的本地化；**数据本地化并不是总能实现，比如在HRegion移动(如因Split)时，需要等下一次Compact才能继续回到本地化。**

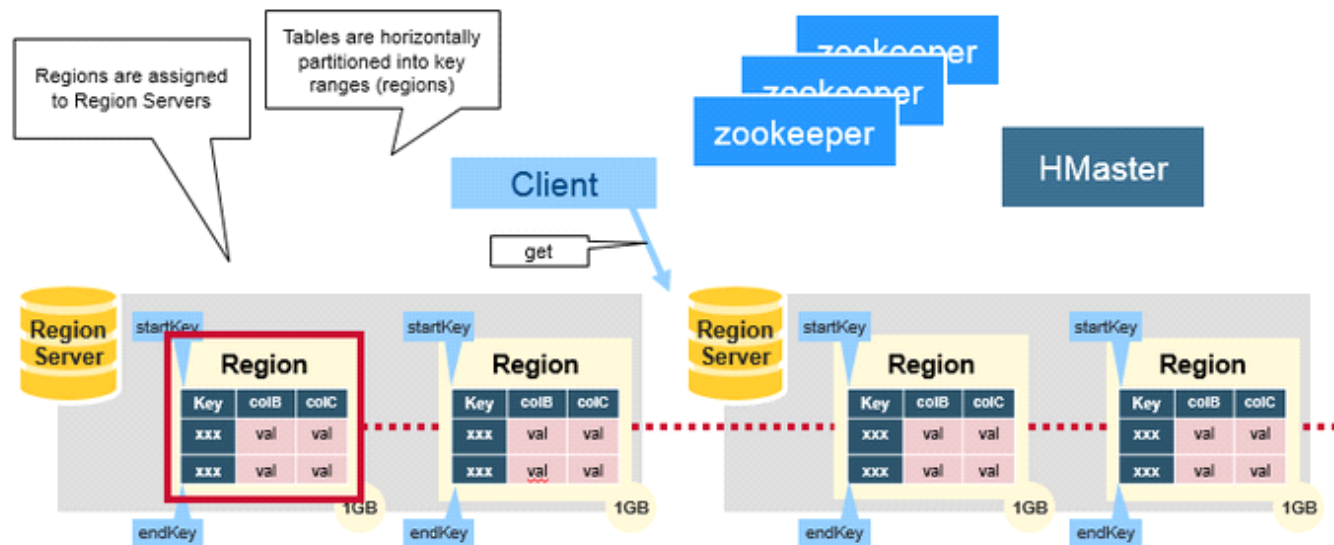
HBASE架构原理详解

2015年3月22日 17:35

整体架构图



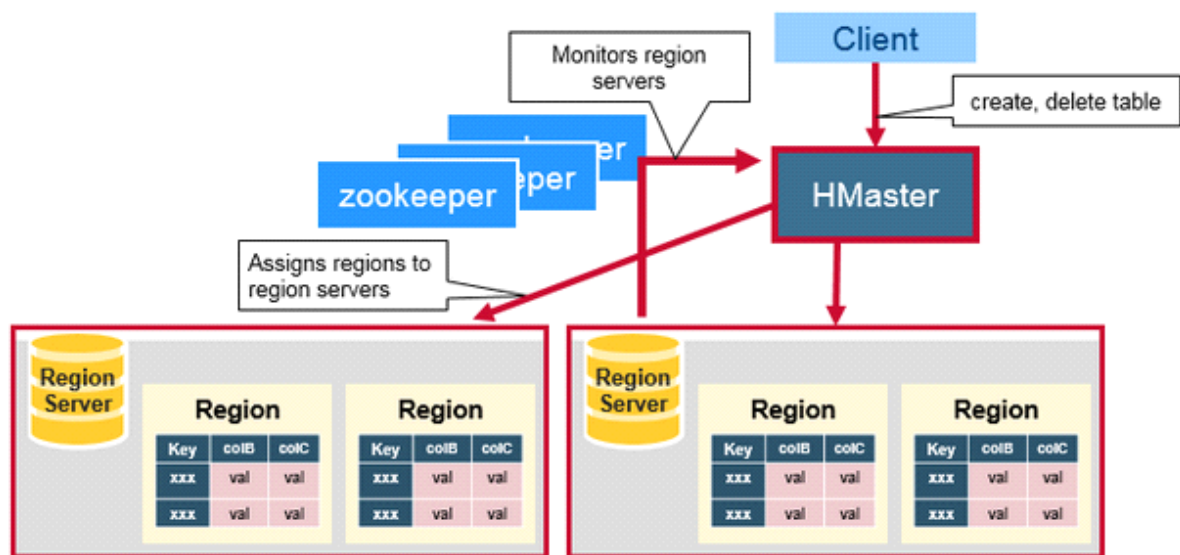
Hregion



HBase使用RowKey将表水平切割成多个HRegion，从HMaster的角度，每个HRegion都纪录了它的StartKey和EndKey，由于RowKey是排序的，因而Client可以通过HMaster快速的定位每个

RowKey在哪个HRegion中。 HRegion由HMaster分配到相应的HRegionServer中，然后由HRegionServer负责HRegion的启动和管理，和Client的通信，负责数据的读(使用HDFS)。每个HRegionServer可以同时管理1000个左右的HRegion（这个数字怎么来的？超过1000个会引起性能问题？这个1000的数字是从BigTable的论文中来的（5 Implementation节）：Each tablet server manages a set of tablets(typically we have somewhere between ten to a thousand tablets per tablet server))。

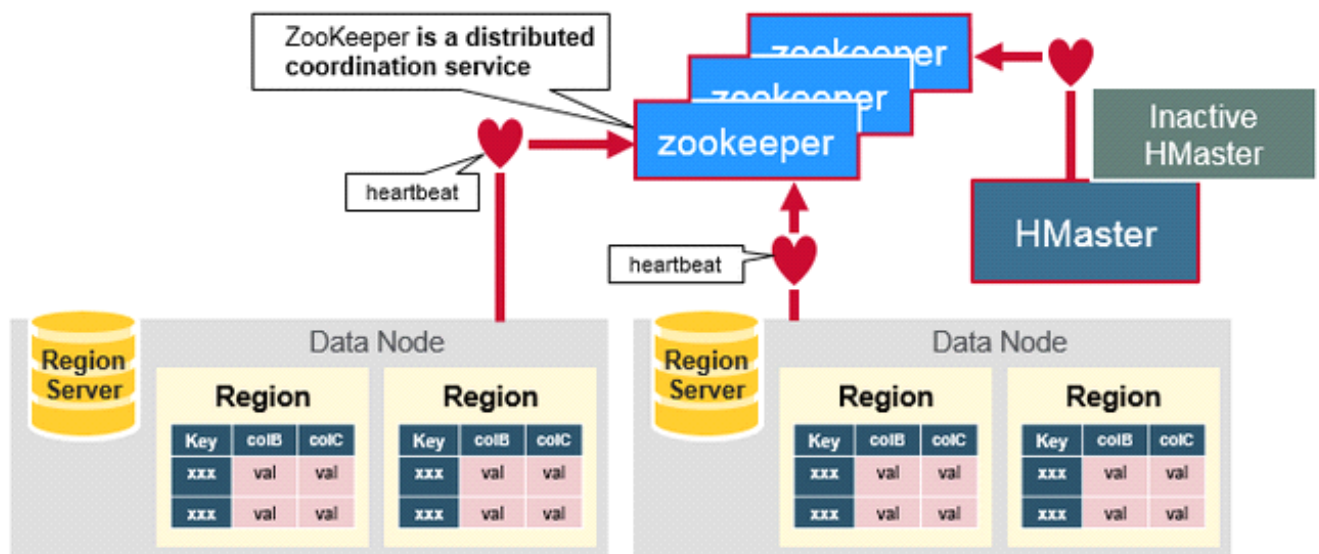
HMaster



HMaster没有单点故障问题，可以启动多个HMaster，通过ZooKeeper的Master Election机制保证同时只有一个HMaster处于Active状态，其他的HMaster则处于热备份状态。一般情况下会启动两个HMaster，非Active的HMaster会定期的和Active HMaster通信以获取其最新状态，从而保证它是实时更新的，因而如果启动了多个HMaster反而增加了Active HMaster的负担。前文已经介绍了HMaster的主要用于HRegion的分配和管理，DDL(Data Definition Language，既Table的新建、删除、修改等)的实现等，既它主要有两方面的职责：

1. 协调HRegionServer
 1. 启动时HRegion的分配，以及负载均衡和修复时HRegion的重新分配。
 2. 监控集群中所有HRegionServer的状态(通过Heartbeat和监听ZooKeeper中的状态)。
2. Admin职能
 1. 创建、删除、修改Table的定义。

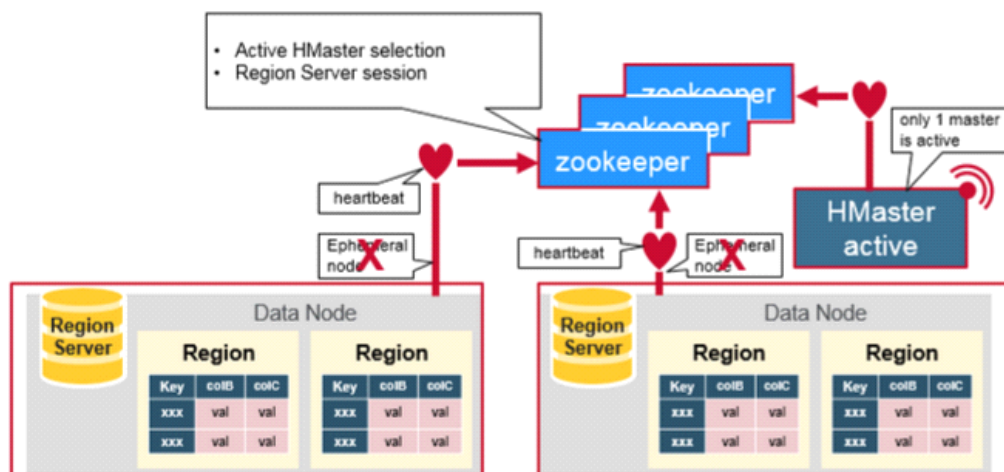
ZooKeeper：协调者



ZooKeeper（根据Google的《The Chubby lock service for loosely coupled distributed System》）为HBase集群提供协调服务，它管理着HMaster和HRegionServer的状态 (available/alive等)，并且会在它们宕机时通知给HMaster，从而HMaster可以实现HMaster之间的failover(失败恢复)，或对宕机的HRegionServer中的HRegion集合的修复(将它们分配给其他的HRegionServer)。ZooKeeper集群本身使用一致性协议(PAXOS协议，PAXOS算法的思想是：在分布式的环境下，如何就某个决议达成一致性的算法，PAXOS算法的缺点是存在活锁问题，ZK是基于PAXOS实现的Fast PAXOS)保证每个节点状态的一致性。

How The Components Work Together

ZooKeeper协调集群所有节点的共享信息，在HMaster和HRegionServer连接到ZooKeeper后创建Ephemeral(临时)节点，并使用Heartbeat机制维持这个节点的存活状态，如果某个Ephemeral节点失效，则HMaster会收到通知，并做相应的处理。



另外，HMaster通过监听ZooKeeper中的Ephemeral节点(默认：/hbase/rs/*)来监控HRegionServer的加入和宕机。在第一个HMaster连接到ZooKeeper时会创建Ephemeral节点(默

认：/hbase/master)来表示Active的HMaster，其后加进来的HMaster则监听该Ephemeral节点，如果当前Active的HMaster宕机，则该节点消失，因而其他HMaster得到通知，而将自身转换成Active的HMaster，在变为Active的HMaster之前，它会创建在/hbase/back-masters/下创建自己的Ephemeral节点。

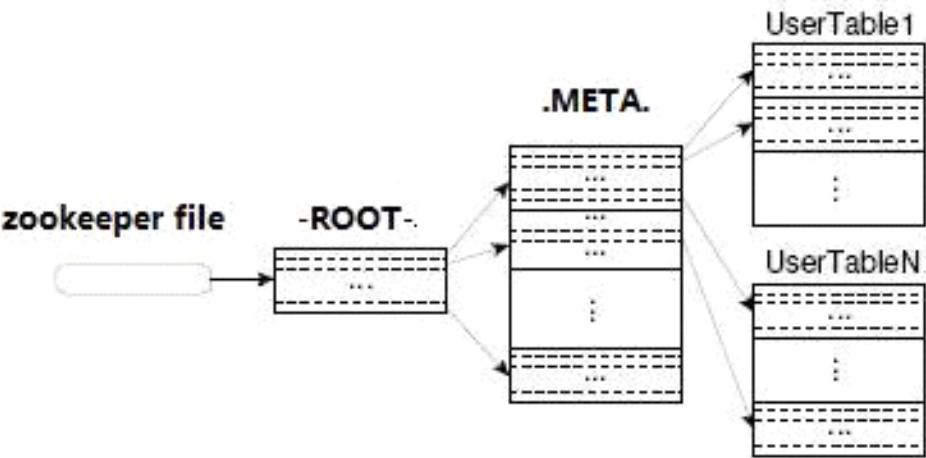
HBase的第一次读写

在HBase 0.96以前，HBase有两个特殊的Table：-ROOT-和.META.（如[BigTable](#)中的设计），其中-ROOT- Table的位置存储在ZooKeeper，它存储了.META. Table的RegionInfo信息，并且它只能存在一个HRegion，而.META. Table则存储了用户Table的RegionInfo信息，它可以被切分成多个HRegion，因而对第一次访问用户Table时，首先从ZooKeeper中读取-ROOT- Table所在HRegionServer；然后从该HRegionServer中根据请求的TableName，RowKey读取.META. Table所在HRegionServer；最后从该HRegionServer中读取.META. Table的内容而获取此次请求需要访问的HRegion所在的位置，然后访问该HRegionSever获取请求的数据，这需要三次请求才能找到用户Table所在的位置，然后第四次请求开始获取

.META.



真正的数据。当然为了提升性能，客户端会缓存-ROOT- Table位置以及-ROOT-/ .META. Table的内容。如下图所示：



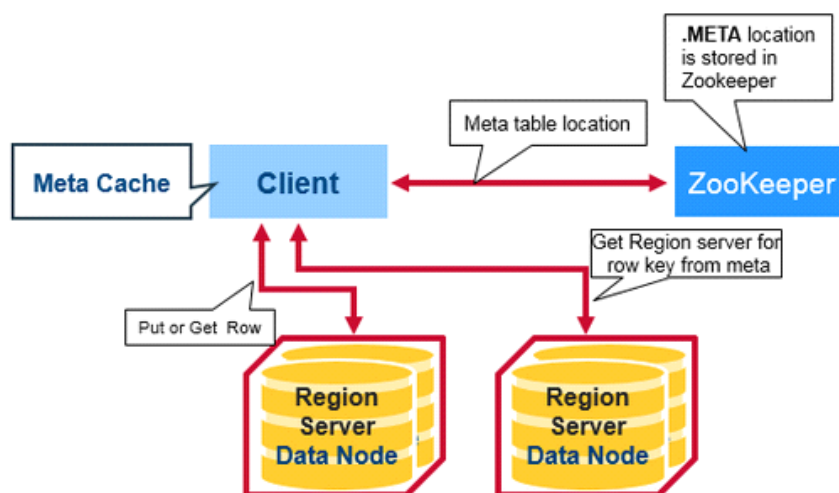
可是即使客户端有缓存，在初始阶段需要三次请求才能直到用户Table真正所在的位置也是性

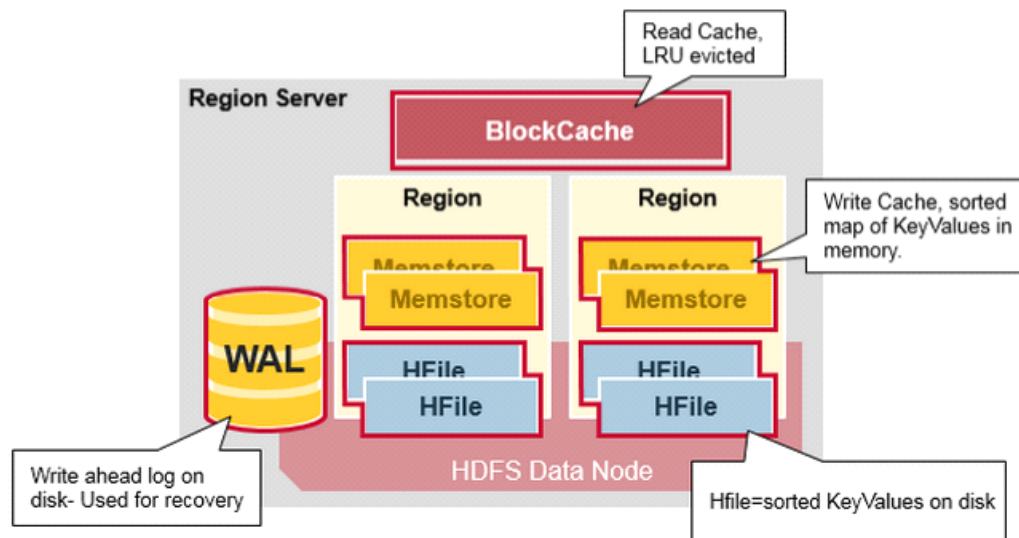
能低下的，而且真的有必要支持那么多的HRegion吗？或许对Google这样的公司来说是需要，但是对一般的集群来说好像并没有这个必要。在BigTable的论文中说，每行METADATA存储1KB左右数据，中等大小的Tablet(HRegion)在128MB左右，3层位置的Schema设计可以支持 2^{34} 个Tablet(HRegion)。即使去掉-ROOT-Table，也还可以支持 2^{17} (131072)个HRegion，如果每个HRegion还是128MB，那就是16TB，这个貌似不够大，但是现在的HRegion的最大大小都会设置的比较大，比如我们设置了2GB，此时支持的大小则变成了4PB，对一般的集群来说已经够了，**因而在HBase 0.96以后去掉了-ROOT-Table，只剩下这个特殊的目录表叫做Meta**

Table(hbase:meta)，它存储了集群中所有用户HRegion的位置信息，而ZooKeeper的节点中(/hbase/meta-region-server)存储的则直接是这个Meta Table的位置，并且这个Meta Table如以前的-ROOT-Table一样是不可split的。**这样，客户端在第一次访问用户Table的流程就变成了：**

1. 从ZooKeeper(/hbase/meta-region-server)中获取hbase:meta的位置（HRegionServer的位置），缓存该位置信息。
2. 从HRegionServer中查询用户Table对应请求的RowKey所在的HRegionServer，**缓存该位置信息。**
3. 从查询到HRegionServer中读取Row。

从这个过程中，我们发现客户会缓存这些位置信息，然而第二步它只是缓存当前RowKey对应的HRegion的位置，因而如果下一个要查的RowKey不在同一个HRegion中，则需要继续查询hbase:meta所在的HRegion，**然而随着时间的推移，客户端缓存的位置信息越来越多，以至于不需要再次查找hbase:meta Table的信息，除非某个HRegion因为宕机或Split被移动，此时需要重新查询并且更新缓存。**





HRegionServer一般和DataNode在同一台机器上运行，**实现数据的本地性**。HRegionServer存活和管理多个HRegion，由WAL(HLog)、BlockCache、MemStore、HFile组成。

1. **WAL即Write Ahead Log**，在早期版本中称为HLog，它是HDFS上的一个文件，如其名字所表示的，所有**写操作都会先保证将写操作写入这个Log文件后，才会真正更新MemStore**，最后写入HFile中。采用这种模式，可以保证HRegionServer宕机后，我们依然可以从该Log文件中恢复数据，Replay所有的操作，而不至于数据丢失。

HLog文件就是一个普通的Hadoop Sequence File，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 Sequence File的Value是HBase的KeyValue对象，即对应HFile中的KeyValue。

这个Log文件会定期Roll出新的文件而删除旧的文件(那些已持久化到HFile中的Log可以删除)。WAL文件存储在/hbase/WALs/\${HRegionServer_Name}的目录中(在0.94之前，存储在/hbase/.logs/目录中)，**一般一个HRegionServer只有一个WAL实例**，也就是说一个HRegionServer的所有WAL写都是串行的(就像log4j的日志写也是串行的)。一个RS服务器只有一个HLOG文件，在0.94版本之前，写HLOG的操作是串行的，所以效率很低，所以1.0版本之后，Hbase引入多管道并行写技术，从而提高性能。

2.**BlockCache是一个读缓存**，即“引用局部性”原理（也应用于CPU，[分空间局部性和时](#)

空间局部性，空间局部性是指CPU在某一时刻需要某个数据，那么有很大的概率在一下时刻它需要的数据在其附近；**时间局部性**是指某个数据在被访问过一次后，它有很大的概率在不久的将来会被再次的访问），将数据预读取到内存中，以提升读的性能。

这样设计的目的是为了**提高读缓存的命中率**。

HBase中默认on-heap LruBlockCache。（LRU -evicted，是一种数据的回收策略， LRU-最近最少使用的：移除最长时间不被使用的对象。）

3.HRegion是一个Table中的一个Region在一个HRegionServer中的表达。一个Table可以有一个或多个HRegion，他们可以在一个相同的HRegionServer上，也可以分布在不同的HRegionServer上，一个HRegionServer可以有多个HRegion，他们分别属于不同的Table。HRegion由多个Store(HStore)构成，每个HStore对应了一个Table在这个HRegion中的一个Column Family，即每个Column Family就是一个集中的存储单元，**因而最好将具有相近I/O特性的Column存储在一个Column Family，以实现高效读取(数据局部性原理，可以提高缓存的命中率)**。HStore是HBase中存储的核心，它实现了读写HDFS功能，一个HStore由一个MemStore 和0个或多个StoreFile组成。

4.MemStore是一个写缓存(In Memory Sorted Buffer)，所有数据的写在完成WAL日志写后，会写入MemStore中，由MemStore**根据一定的算法（LSM-TREE算法，这个算法的作用是将数据顺序写磁盘，而不是随机写，减少磁头调度时间，从而提高写入性能）**将数据Flush到底层的HDFS文件中(HFile)，通常每个HRegion中的每个 Column Family有一个自己的MemStore。

5.HFile(StoreFile) 用于存储HBase的数据(Cell/KeyValue)。在HFile中的数据是按RowKey、Column Family、Column排序，对相同的Cell(即这三个值都一样)，则按timestamp倒序排列。

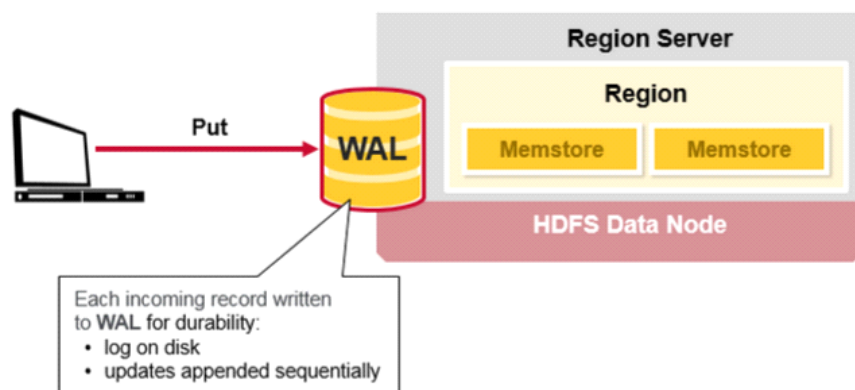
因为Hbase的HFile是存到HDFS上，所以Hbase实际上是具备数据的副本冗余机制的。

HBASE写流程

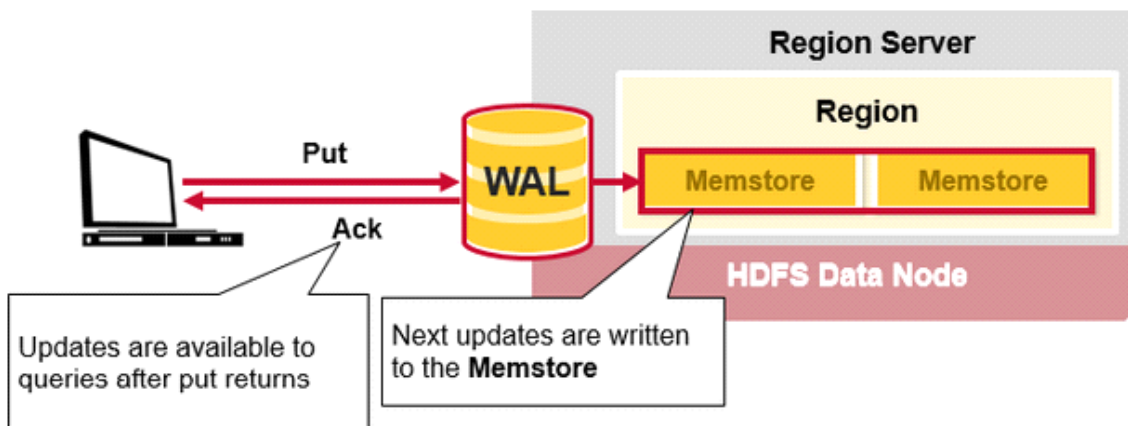
2017年11月29日 19:45

HRegionServer中数据写流程图解

当客户端发起一个Put请求时，首先它从hbase:meta表中查出该Put数据最终需要去的HRegionServer。然后客户端将Put请求发送给相应的HRegionServer，在HRegionServer中它首先会将该Put操作写入WAL日志文件中(Flush到磁盘中)。

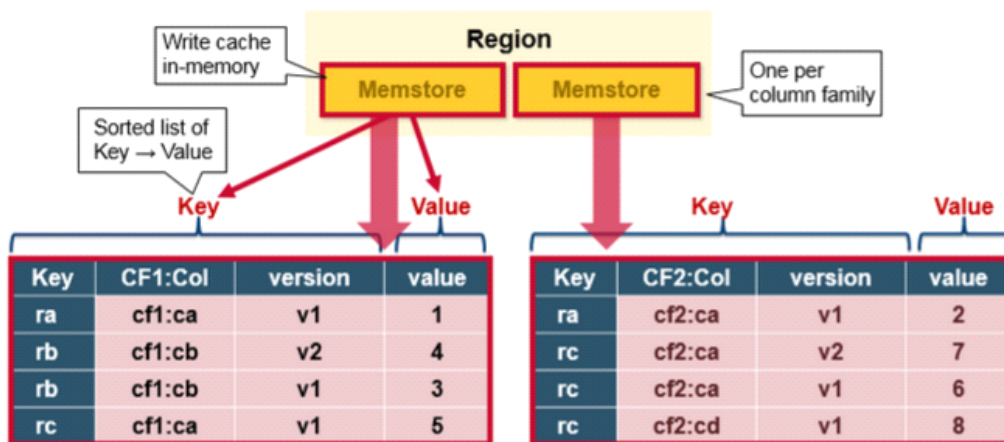


写完WAL日志文件后，然后将数据写到Memstore，在Memstore按Rowkey排序，以及用LSM-TREE对数据做合并处理。HRegionServer根据Put中的TableName和RowKey找到对应的HRegion，并根据Column Family找到对应的HStore，并将Put写入到该HStore的MemStore中。此时写成功，并返回通知客户端。



MemStore Flush

MemStore是一个In Memory Sorted Buffer，在每个HStore中都有一个MemStore，即它是一个HRegion的一个Column Family对应一个实例。它的排列顺序以RowKey、Column Family、Column的顺序以及Timestamp的倒序，如下所示：



每一次Put/Delete请求都是先写入到MemStore中，当MemStore满后会Flush成一个新的StoreFile(底层实现是HFile)，即一个HStore(Column Family)可以有0个或多个StoreFile(HFile)。有以下三种情况可以触发MemStore的Flush动作：

1. 当一个HRegion中的MemStore的大小超过了：

`hbase.hregion.memstore.flush.size`的大小，默认128MB。

此时当前的MemStore会Flush到HFile中。

2. 当RS服务器上所有的MemStore的大小超过了：`hbase.regionserver.global.memstore.upperLimit`的大小，默认35%的内存使用量。

此时当前HRegionServer中所有HRegion中的MemStore可能都会Flush。**从最大的Memstore开始flush**

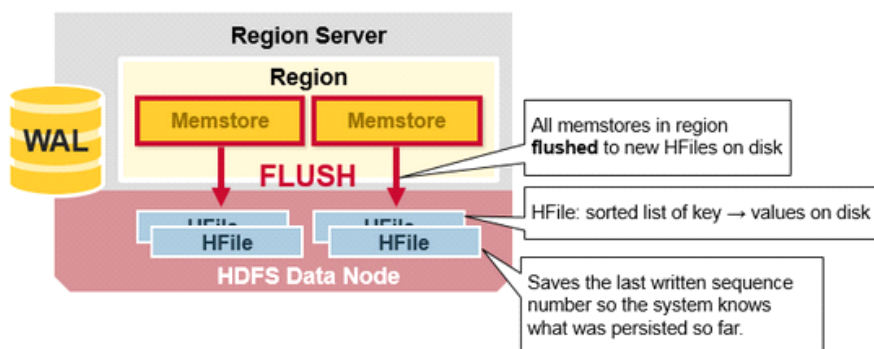
3. 当前HRegionServer中WAL的大小超过了 1GB

`hbase.regionserver.hlog.blocksize(32MB) * hbase.regionserver.max.logs(32)`的数量，当前HRegionServer中所有HRegion中的MemStore都会Flush

这里指的是两个参数相乘的大小。

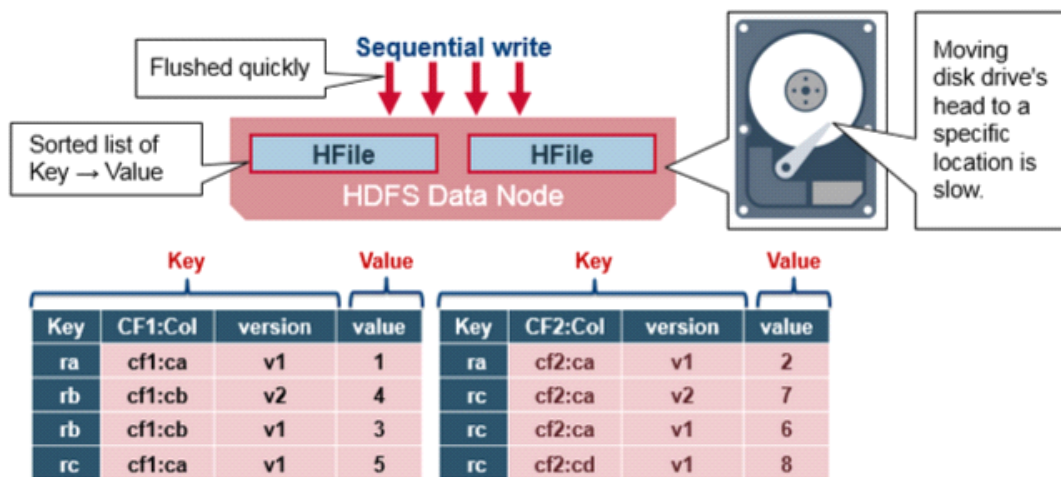
查代码发现：`hbase.regionserver.max.logs`默认值是32，而`hbase.regionserver.hlog.blocksize`是HDFS的默认blocksize，32MB

此外，在MemStore Flush过程中，还会在尾部追加一些meta数据，其中就包括Flush时最大的WAL sequence值，以告诉HBase这个StoreFile写入的最新数据的序列，那么在Recover时就直到从哪里开始。在HRegion启动时，这个sequence会被读取，并取最大的作为下一次更新时的起始sequence。



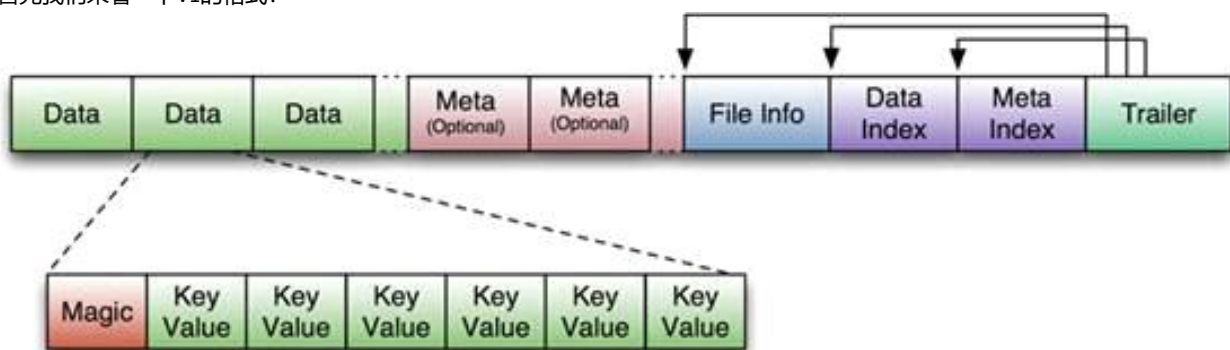
HFile格式

HBase的数据以KeyValue(Cell)的形式顺序的存储在HFile中，在MemStore的Flush过程中生成HFile，由于MemStore中存储的Cell遵循相同的排列顺序，因而Flush过程是**顺序写**，我们知道磁盘的顺序写性能很高，因为不需要不停的移动磁盘指针。



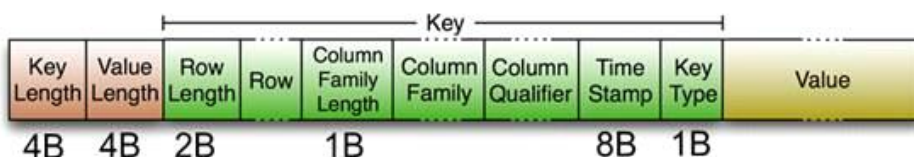
HFile参考BigTable的SSTable和Hadoop的TFile实现，从HBase开始到现在，HFile经历了三个版本，其中V2在0.92引入

首先我们来看一下v1的格式：



V1的HFile由多个Data Block、Meta Block、FileInfo、Data Index、Meta Index、Trailer组成，其中Data Block是HBase的最小存储单元，在前文中提到的BlockCache就是基于Data Block的缓存的。一个Data Block由一个魔数和一系列的KeyValue(Cell)组成，魔数是一个随机的数字，用于表示这是一个Data Block类型，以快速检测这个Data Block的格式，防止数据的破坏。Data Block的大小可以在创建Column Family时设置(HColumnDescriptor.setBlockSize())，默认值是64KB，大号的Block有利于顺序Scan，小号Block利于随机查询，因而需要权衡。Meta块是可选的，FileInfo是固定长度的块，它纪录了文件的一些Meta信息，例如：AVG_KEY_LEN, AVG_VALUE_LEN, LAST_KEY, COMPARATOR, MAX_SEQ_ID_KEY等。Data Index和Meta Index纪录了每个Data块和Meta块的起始点、未压缩时大小、Key(起始RowKey)等。Trailer纪录了FileInfo、Data Index、Meta Index块的起始位置，Data Index和Meta Index索引的数量等。其中FileInfo和Trailer是固定长度的。

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。随着HFile版本迁移，KeyValue(Cell)的格式并未发生太多变化，只

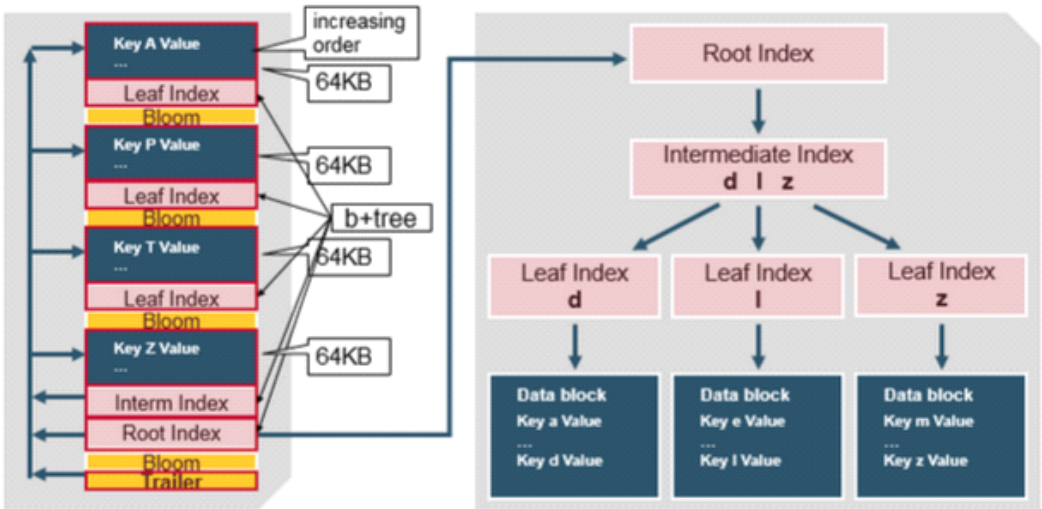
是在v3版本，尾部添加了一个可选的Tag数组。

HFileV1版本的在实际使用过程中发现它占用内存多，因而增加了启动时间。为了解决这些问题，在0.92版本中引入HFileV2版本：

“Scanned block” section	Data Block		
	...		
	Leaf index block / Bloom block		
	...		
	Data Block		
	...		
	Leaf index block / Bloom block		
“Non-scanned block” section	Data Block		
	...		
“Load-on-open” section	Meta block	...	Meta block
	Intermediate Level Data Index Blocks (optional)		
	Root Data Index		Fields for midkey
	Meta Index		
Trailer	File Info		Bloom filter metadata (interpreted by StoreFile)
	Trailer fields		Version

在这个版本中，为了提升启动速度，还引入了延迟读的功能，即在HFile真正被使用时才对其进行解析。

对HFileV2格式具体分析，它是一个多层的类B+树索引，采用这种设计，可以实现查找不需要读取整个文件：



Data Block中的Cell都是升序排列，每个block都有它自己的Leaf-Index，每个Block的最后一个Key被放入Intermediate-Index中，Root-Index指向Intermediate-Index。在HFile的末尾还有Bloom Filter(布隆过滤)用于快速定位那么没有在某Data Block中的Row；TimeRange信息用于给那些使用时间查询的参考。在HFile打开时，这些索引信息都被加载并保存在内存中，以增加以后的读取性能。

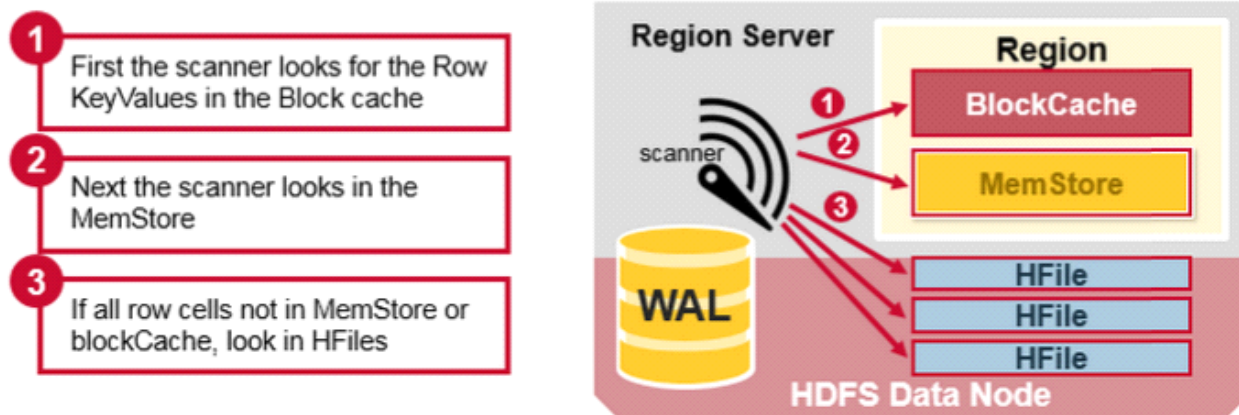
HBASE读的实现

2015年3月22日 18:14

HBase读的实现

我们先来分析一下相同的Cell（数据）可能存在的位置：首先对新写入的Cell，它会存在于MemStore中；然后对之前已经Flush到HFile中的Cell，它会存在于某个或某些StoreFile(HFile)中；最后，对刚读取过的Cell，它可能存在于BlockCache中。既然相同的Cell可能存储在三个地方，在读取的时候只需要扫描这三个地方，然后将结果合并即可(Merge Read)，在HBase中扫描的顺序依次是：

BlockCache、MemStore、StoreFile(HFile)（这个扫描顺序的目的也是为了减少磁盘的I/O次数）。其中StoreFile的扫描先会使用Bloom Filter(布隆过滤算法) 过滤那些不可能符合条件的HFile，然后使用Block Index快速定位Cell，并将其加载到BlockCache中，然后从BlockCache中读取。我们知道一个HStore可能存在多个StoreFile(HFile)，此时需要扫描多个HFile，如果HFile过多又是会引起性能问题。



Compaction机制

MemStore每次Flush会创建新的HFile，而过多的HFile会引起读的性能问题，那么如何解决这个问题呢？HBase采用Compaction机制来解决这个问题。在HBase中Compaction分为两种：**Minor Compaction**和**Major Compaction**。

1. Minor Compaction是指选取一些小的、相邻的StoreFile将他们合并成一个更大的StoreFile，在这个过程中不会处理已经Deleted或Expired的Cell。一次Minor Compaction的结果是更少并且更大的StoreFile。（这个是对的吗？BigTable中是这

样描述Minor Compaction的: As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This *minor compaction* process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur. 也就是说它将memtable的数据flush的一个HFile/SSTable称为一次Minor Compaction)

2. Major Compaction是指将所有的StoreFile合并成一个StoreFile, 在这个过程中, 标记为Deleted的Cell会被删除, 而那些已经Expired的Cell会被丢弃, 那些已经超过最多版本数的Cell会被丢弃。一次Major Compaction的结果是一个HStore只有一个StoreFile存在。Major Compaction可以手动或自动触发, 然而由于它会引起很多的I/O操作而引起性能问题, 因而它一般会被安排在周末、凌晨等集群比较闲的时间。

如何实现Compaction:

- 1.通过API:

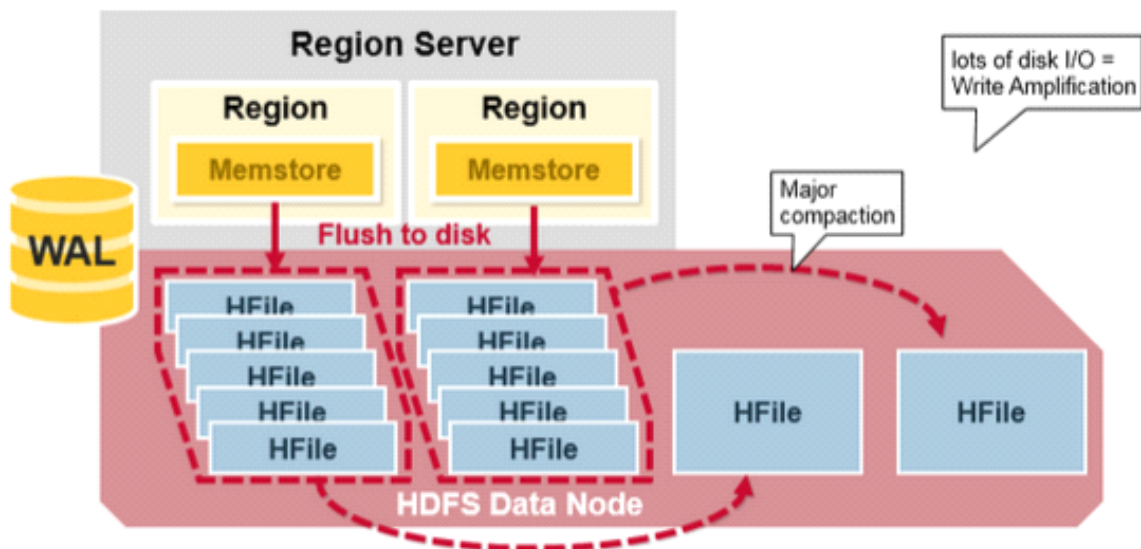
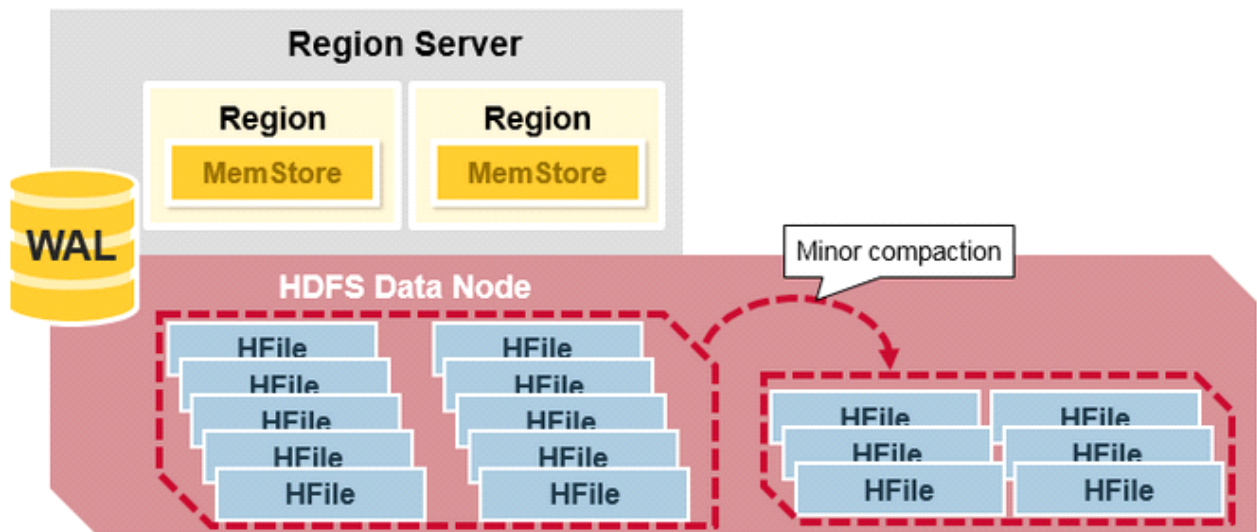
```
//--minor compact  
admin.compact("tab2".getBytes());  
  
//--major compact  
admin.majorCompact("tab2".getBytes());
```

- 2.通过指令:

```
compact('tab2')  
major_compact('tab2')
```

更形象一点, 如下面两张图分别表示Minor Compaction和Major Compaction。

Hbase默认用的是Minor compaction。之所以默认不用Major Compaction的原因是在于，Major Compaction可能会代理大量的磁盘I/O，从而阻塞Hbase其他的读写操作。所以对于Major Compaction,一般选择在业务峰值低的时候执行。



Major compaction的使用注意事项

虽然这种合并可以有效的减少HFile数量，但是可能的副作用是产生的磁盘I/O负载极大，所以可能会影响到正常的HBase服务端和客户端的交互。所以一般选在业务峰值低，比如凌晨或周末来执行。

补充：当向HBase表数据更新数据时，表面是更新数据，但是最后数据以追加到HFile中的

这符合HDFS存储文件的原则（不允许修改，但允许追加）

此外，当执行删除时，数据并不是马上从HFile中删除的，还在，只不过被加上了删除标记。

当执行Compactio时，才会将带有删除标记的数据清除掉。

而且针对超过历史版本上限的数据，也会被打上删除标记的。

BloomFilter

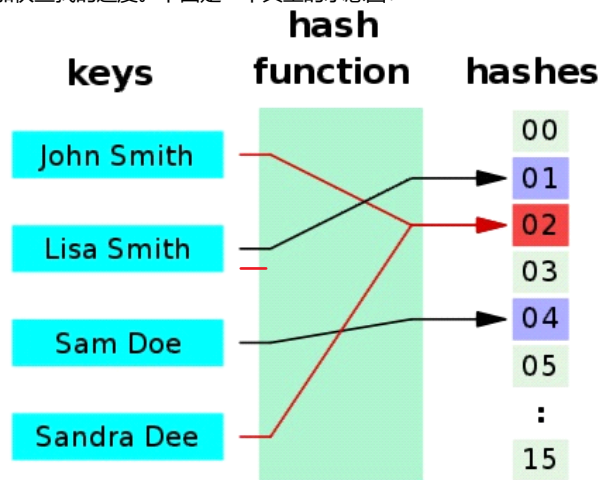
2017年11月29日 21:41

背景说明

Hash 函数在计算机领域，尤其是数据快速查找领域，加密领域用的极广。

其作用是将一个大的数据集映射到一个小的数据集上面（这些小的数据集叫做哈希值，或者散列值）。

Hash table（散列表，也叫哈希表），是根据哈希值(Key value)而直接进行访问的数据结构。也就是说，它通过把哈希值映射到表中一个位置来访问记录，以加快查找的速度。下面是一个典型的示意图：



但是这种简单的Hash Table存在一定的问题，就是Hash冲突的问题。假设 Hash 函数是良好的，如果我们的位阵列长度为 m 个点，那么如果我们想将冲突率降低到例如 1%，这个散列表就只能容纳 $m * 1\%$ 个元素。显然这就不叫空间有效了（Space-efficient）。

Bloom Filter概述

Bloom Filter是1970年由布隆（Burton Howard Bloom）提出的。它实际上是一个很长的二进制向量和一系列随机映射函数（Hash函数）。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法。Bloom Filter广泛的应用于各种需要查询的场合中，如：

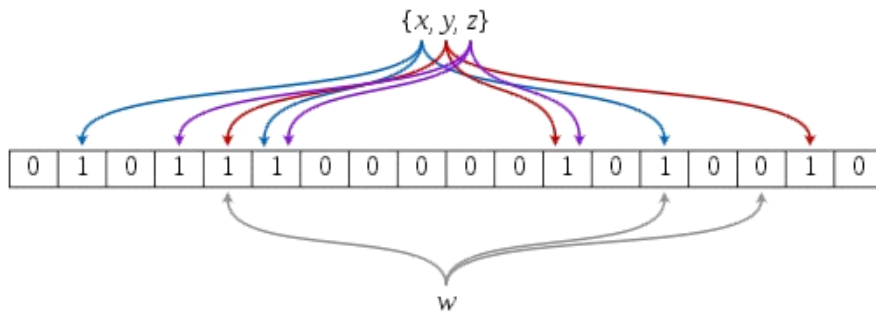
Google 著名的分布式数据库 Bigtable 使用了布隆过滤器来查找不存在的行或列，以减少磁盘查找的IO次数。

在很多Key-Value系统中也使用了布隆过滤器来加快查询过程，如 Hbase，Accumulo，Leveldb，一般而言，Value 保存在磁盘中，访问磁盘需要花费大量时间，然而使用布隆过滤器可以快速判断某个Key对应的Value是否存在，因此可以避免很多不必要的磁盘IO操作，只是引入布隆过滤器会带来一定的内存消耗。

Bloom Filter 原理

如果想判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比较确定。链表，树等等数据结构都是这种思路。但是随着集合中元素的增加，我们需要的存储空间越来越大，检索速度也越来越慢。

一个Bloom Filter是基于一个 m 位的位向量（ b_1, \dots, b_m ），这些位向量的初始值为0。另外，还有一系列的hash函数（ h_1, \dots, h_k ），这些hash函数的值域属于 $1 \sim m$ 。下图是一个bloom filter插入x,y,z并判断某个值w是否在该数据集的示意图：



但是布隆过滤器的缺点和优点一样明显。误算率（False Positive）是其中之一。随着存入的元素数量增加，误算率随之增加。但是如果元素数量太少，则使用散列表足矣。

总结：Bloom Filter 通常应用在一些需要快速判断某个元素是否属于集合，但是并不严格要求100%正确的场合。此外，引入布隆过滤器会带来一定的内存消耗。

HBase表设计

2016年1月25日 9:42

Rowkey设计

Rowkey是不可分割的字节数，按字典排序由低到高存储在表中。

在设计HBase表时，Rowkey设计是最重要的事情，应该基于预期的访问模式来为Rowkey建模。Rowkey决定了访问HBase表时可以得到的性能，原因有两个：

- 1) Region基于Rowkey为一个区间的行提供服务，并且负责区间的每一行；
- 2) HFile在硬盘上存储有序的行。

这两个因素是相互关联的。当Region将内存中数据刷写为HFile时，这些行已经排过序，也会有序地写到硬盘上。Rowkey的有序特性和底层存储格式可以保证HBase表在设计Rowkey之后的良好性能。

关系型数据库可以在多列上建立索引，**但是HBase只能在Rowkey上建立索引**。（可以通过ES为Hbase的列建立索引）而设计Rowkey有各种技巧，而且可以针对不同访问模式进行优化，我们接下来就研究一下。

1.将Rowkey以字典顺序从大到小排序

原生HBase只支持从小到大的排序，但是现在有个需求想展现影片热度排行榜，这就要求实现从大到小排列，针对这种情况可以采用 $\text{Rowkey} = \text{Integer.MAX_VALUE} - \text{Rowkey}$ 的方式将Rowkey进行转换，最大的变最小，最小的变最大，在应用层再转回来即可完成排序需求。

2.RowKey尽量散列设计

最重要的是要保证散列，这样就会保证所有的数据都不是在一个Region上，从而避免读写时

候负载会集中在个别Region上。比如ROWKEY_Random

3.RowKey的长度尽量短

如果Rowkey太长，第一存储开销会增加，影响存储效率；第二内存中Rowkey字段过长，会导致内存的利用率降低，进而降低索引命中率。

Rowkey是一个二进制码流，Rowkey的长度被很多开发者建议说设计在10~100个字节，不过建议是越短越好，**不要超过16个字节**。

原因如下：

- 1) 数据的持久化文件HFile中是按照KeyValue存储的，如果Rowkey过长比如100个字节，1000万列数据光Rowkey就要占用 $100 \times 1000\text{万} = 10\text{亿}$ 个字节，将近1G数据，这会极大影响HFile的存储效率；
- 2) MemStore将缓存部分数据到内存，如果Rowkey字段过长内存的有效利用率会降低，系统将无法缓存更多的数据，这会降低检索效率。因此Rowkey的字节长度越短越好。

4.RowKey唯一

5.RowKey建议用String类型

虽然行键在HBase中是以byte[]字节数组的形式存储的，但是建议在系统开发过程中将其数据类型设置为String类型，保证通用性。

常用的行键字符串有以下几种：

- 1) 纯数字字符串，譬如9559820140512；
- 2) 数字+特殊分隔符，譬如95598-20140512；
- 3) 数字+英文字母，譬如city20140512；
- 4) 数字+英文字母+特殊分隔符，譬如city_20140512

6.RowKey设计得最好有意义

RowKey的主要作用是为了进行数据记录的唯一性标示，但是唯一性并不是其全部，具有明确意义的行键对于应用开发、数据检索等都具有特殊意义。

譬如数字字符串：9559820140512，其实际意义是这样：95598（电网客服电话）+ 20140512（日期）。

行键往往由多个值组合而成，而各个值的位置顺序将影响到数据存储和检索效率，所以在设计行键时，需要对日后的业务应用开发有比较深入的了解和前瞻性预测，才能设计出可尽量高效率检索的行键。

7.具有定长性

行键具有有序性的基础便是定长，譬如20140512080500、20140512083000，这两个日期时间形式的字符串是递增的，不管后面的秒数是多少，我们都将其设置为14位数字形式，如果我们把后面的0去除了，那么201405120805将大于20140512083，其有序性发生了变更。所以我们建议，行键一定要设计成定长的。

此外，目前操作系统都是64位系统，内存**8字节对齐**。控制在16个字节，8字节的整数倍利用操作系统的最佳特性。

列族的设计

在设计hbase表时候，列族不宜过多，尽量要少使用列族。

经常要在一起查询的数据最好放在一个列族中，尽量减少跨列族的数据访问。

案例一 网络数据访问

比如我们现在数据库里有两张表：

用户表

id	name	age	gender	email
001	zhang	19	男	zhang@qq.com
002	wang	20	男	wang@qq.com

用户访问的网页表

host	viewtime	content	userid
www.baidu.com	2016-12-20	xxxx	001
www.sina.com	2016-11-10	xxxx	001
www.souhu.com	2016-11-09	xxxx	001
www.baidu.com	2016-12-20	xxxx	002
www.163.com	2016-12-20	xxxx	002

我们现在要将上述两表设计为Hbase表，如何设计行键？

行键怎么设计，直接决定了查询语句怎么写，如果我将行键设计为：日期_姓名这种形式：

2016-12-20_zhang

2016-11-10_zhang

2016-11-09_zhang

2016-12-20_wang

2016-12-20_wang

则在查询时，我们可以利用hbase提供的过滤器进行日期范围的查询：

```
Filter filter=new PrefixFilter("2016-12-20".getBytes());
```

也可以查询以人名为过滤条件的查询。所以在hbase中，行键的设计很重要，要结合具体业务。

实例2 动物分类

如果一位生物科学家要存储一些动物相关信息，其中要包括动物的大类，以及一些大类动物下包括的小类，这样可以方便今后查询某种具体动物属于哪一类别，以及动物名字具体是什么。

下面简单列举一些动物分类如下：

Animal

Pig

Cat

Monkey

Dog：Red dog和Black dog

Tiger：Tiger of northeast

其中，Animal是顶级分类，Pig、Cat、Monkey、Dog和Tiger属于一级分类，而Dog中的Red dog和Black dog，以及Tiger中的Tiger of northesst属于二级分类。

RDBMS表结构设计

动物分类在关系型数据库的设计中只需要考虑主键的映射关系即可，需要一个分类表结构。每种动物都有几个关键字段：id、name、parent_id和child_id，如表所示。

RDBMS中的动物分类表结构

id PK	name	parent_id	child_id
1	animal		2,3,4,5
2	pig	1	
3	cat	1	
4	monkey	1	
5	dog	1	7,8
6	red-dog	1,5	
7	black-dog	1,5	
8	tiger	1	9
9	tiger-of -northeast	1,8	

HBase中表结构设计

对于动物分类的HBase表结构设计，Rowkey对应RDBMS中的id，共有三个列族：

name、parent和child，详细设计如表

Rowkey	Column Family		
<id>	name	parent	child
		parent:<id>	child:<id>
1	animal		child:2=cat child:3=monkey child:4=dog child:5=tiger child:6=pig
4	dog	parent:1=animal	child:7=reddog child:8=blackdog
7	reddog	parent:1=animal parent:4=dog	

通过这个简单的例子可以看出，这两种表结构设计有本质的区别，一个是行式存储，一个是列式存储，所以刚刚接触HBase的读者需要转换思维设计表结构，这样才能够更好地掌握HBase的表模式设计。

HBASE优化

硬件和操作系统调优

1) 配置内存

HBase对于内存的消耗是非常大的，主要是其LSM树状结构、缓存机制和日志记录机制决定的，所以物理内存当然是越大越好。并且现在内存的价格已经降到可以批量配置的程度，例如一条三星DDR3的16GB内存，价格大约在1000元左右。

在互联网领域，服务器内存方面的主流配置已经是64GB，所以一定要根据实际的需求和预算配备服务器内存。如果资源很紧张，推荐内存最小在32GB，如果再小会严重影响HBase集群性能。

2) 配置CPU

HBase给使用者的印象可能更偏向于“内存型” NoSQL数据库，从而忽略了CPU方面的需求，其实HBase在某些应用上对CPU的消耗非常大，例如频繁使用过滤器，因为在过滤器中包含很多匹配、搜索和过滤的操作；多条件组合扫描的场景也是CPU密集型的；压缩操作很频繁等。如果服务器CPU不够强悍，会导致整个集群的负载非常高，很多线程都在阻塞状态（非网络阻塞和死锁的情况）。

一般CPU的品牌有Intel、AMD、IBM，Intel是主流。

现在的服务器支持1、2、3、4、6、8、10路CPU，而每路CPU的核心有双核、四核、六核、八核、十二核。CPU数量和核心数之间可以互相搭配，当然值越大相应的价格越高。建议每台物理节点至少使用双路四核CPU（2×4），主流是2~8路，一般单颗CPU至少四核。一颗四核心CPU，便宜的，价格在1500元左右，还是可以接受的。所以，对于CPU密集型的集群，当然是越多越好。

3) 垃圾回收器（GC）的选择

对于运行HBase相关进程JVM的垃圾回收器，不仅仅关注吞吐量，还关注停顿时间，而且两者之间停顿时间更为重要，因为HBase设计的初衷就是解决大规模数据集下实时访问的问题。那么按照首位是停顿时间短，从这个方面**CMS**和G1有着非常大的优势。

而CMS作为JDK1.5已经出现的垃圾收集器，已经成熟应用在互联网等各个行业。所以，选用CMS作为老年代的垃圾回收器。与CMS搭配的新生代收集器有Serial和ParNew，而对比这两个收集器，明显ParNew具有更好的性能，所以新生代选用ParNew作为垃圾收集器。那么，**最终选用的垃圾收集器搭配组合是CMS+ParNew**。而且很多成熟应用已经验证了这种组合搭配的优势。

与CMS收集器相关的几个重要参数的具体含义、默认值和相关说明详见表。

参数名称	含 义	默认值	说 明
-XX:+UseConcMarkSweepGC	使用 CMS 垃圾收集器		
-XX:+UseParNewGC	新生代采用并行 GC 策略		JDK5.0 以上，JVM 会根据系统配置自行设置，所以无须再设置此值
-XX:CMSFullGCsBeforeCompaction	多少次 Full GC 后进行内存压缩		由于并发收集器不对内存空间进行压缩整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩整理
-XX:+CMSParallelRemarkEnabled	降低标记停顿		
-XX+UseCMSCompactAtFullCollection	在 FULL GC 的时候，对老年代的压缩		CMS 是不会移动内存的，非常容易产生碎片，导致内存不够用，因此启动内存压缩参数

配置方式：需要添加到hbase-env.sh文件中

```
export HBASE_OPTS="-XX: +UseConcMarkSweepGC" -XX: CMSInitiatingOccupancyFraction=70 -XX:
+UseCMSCompactAtFullCollection
```

4) JVM堆大小设置

堆内存大小参数hbase-env.sh文件中设置，设置的代码如下：

```
export HBASE_HEAPSIZE=16384
```

在上面代码中指定堆内存大小是16284，单位是MB，即16GB。当然，这个值需要根据节点实际的物理内存来决定。一般不超过实际物理内存的1/2。

服务器内存的分配，比如服务器内存64GB，为操作系统预留出8G~16GB。此外给Yarn留出8G~16GB，如果没有其他框架，把剩余的留给HBase

Hbase调优

1) 调节数据块 (data block) 的大小

HFile数据块大小可以在列族层次设置。这个数据块不同于之前谈到的HDFS数据块，其默认值是65536字节，或64KB。数据块索引存储每个HFile数据块的起始键。数据块大小的设置影响数据块索引的大小。**数据块越小，索引越大**，从而占用更大内存空间。同时加载进内存的数据块越小，**随机查找性能更好**。但是，如果需要更好的序列扫描性能，那么一次能够加载更多HFile数据进入内存更为合理，这意味着应该将数据块设置为更大的值。相应地，索引变小，将在随机读性能上付出更多的代价。

可以在表实例化时设置数据块大小，代码如下：

```
hbase (main) : 002: 0> create 'mytable', {NAME => 'colfam1', BLOCKSIZE => '65536'}
```

2) 适当时机关闭数据块缓存

把数据放进读缓存，并不是一定能够提升性能。如果一个表或表的列族只被顺序化扫描访问或很少被访问，则Get或Scan操作花费时间长一点是可以接受的。在这种情况下，可以选择关闭列族的缓存。

关闭缓存的原因在于：如果只是执行很多顺序化扫描，会多次使用缓存，并且可能会滥用缓存，从而把应该放进缓存获得性能提升的数据给排挤出去。

所以如果关闭缓存，不仅可以避免上述情况发生，而且可以让出更多缓存给其他表和同一表的其他列族使用。数据块缓存默认是打开的。

可以在新建表或更改表时关闭数据块缓存属性：

```
hbase (main) : 002: 0> create 'mytable', {NAME => 'colfam1', BLOCKCACHE => 'false'}
```

3) 开启布隆过滤器

数据块索引提供了一个有效的方法getDataBlockIndexReader ()，在访问某个特定的行时用来查找应该读取的HFile的数据块。但是该方法的作用有限。HFile数据块的默认大小是64KB，一般情况下不能调整太多。

如果要查找一个很短的行，只在整个数据块的起始行键上建立索引是无法给出更细粒度的索引信息的。例如，某行占用100字节存储空间，一个64KB的数据块包含 $(64 \times 1024) / 100 = 655.53$ ，约700行，只能把起始行放在索引位上。要查找的行可能落在特定数据块上的行区间，但也不能肯定存放在那个数据块上，

这就导致多种可能性：该行在表中不存在，或者存放在另一个HFile中，甚至在MemStore中。这些情况下，从硬盘读取数据块会带来I/O开销，也会滥用数据块缓存，这会影响性能，尤其是当面对一个巨大的数据集且有很多并发读用户时。

布隆过滤器 (Bloom Filter) 允许对存储在每个数据块的数据做一个反向测验。当查询某行时，先检查布隆过滤器，看看该行是否不在这个数据块。布隆过滤器要么确定回答该行不在，要么回答不知道。因此称之为反向测验。布隆过滤器也可以应用到行内的单元格上，当访问某列标识符时先使用同样的反向测验。

使用布隆过滤器也不是没有代价，相反，存储这个额外的索引层次占用额外的空间。布隆过滤器的占用空间大小随着它们的索引对象数据增长而增长，所以**行级布隆过滤器**比**列标识符级布隆过滤器**占用空间要少。当空间不是问题时，它们可以压榨整个系统的性能潜力。

可以在列族上打开布隆过滤器，代码如下：

```
hbase (main) : 007: 0> create 'mytable', {NAME => 'colfam1', BLOOMFILTER => 'ROWCOL'}
```

布隆过滤器参数的默认值是NONE。另外，还有两个值：ROW表示行级布隆过滤器；ROWCOL表示列标识符级布隆过滤器。行级布隆过滤器在数据块中检查特定行键是否存在，列标识符级布隆过滤器检查行和列标识符联合体是否存在。ROWCOL布隆过滤器的空间开销高于ROW布隆过滤器。

4) 开启数据压缩

HFile可以被压缩并存放在HDFS上，这有助于节省硬盘I/O，但是读写数据时压缩和解压缩会抬高CPU利用率。压缩是表定义的一部分，可以在建表或模式改变时设定。除非确定压缩不会提升系统的性能，否则推荐打开表的压缩。只有在数据不能被压缩，或者因为某些原因服务器的CPU利用率有限制要求的情况下，有可能需要关闭压缩特性。

HBase可以使用多种压缩编码，包括LZO、SNAPPY和GZIP，LZO和SNAPPY是最流行的两种。

当建表时可以在列族上打开压缩，代码如下：

```
hbase (main) : 002: 0>
```

```
create 'mytable', {NAME => 'colfam1', COMPRESSION => 'SNAPPY'}
```

注意，数据只在硬盘上是压缩的，在内存中（MemStore或BlockCache）或在网络传输时是没有压缩的。

5) 设置Scan缓存

HBase的Scan查询中可以设置缓存，定义一次交互从服务器端传输到客户端的行数，设置方法是使用Scan类中setCaching（）方法，这样能有效地减少服务器端和客户端的交互，更好地提升扫描查询的性能。

下面的代码展示了如何使用setCaching（）方法。

代码示例：

```
HTable table = new HTable (config, Bytes.toBytes (tableName) ) ;
```

```
Scan scanner = new Scan ( ) ;
```

```
/* batch and caching */
```

```
scanner.setBatch (0) ;
```

```
scanner.setCaching (10000) ;
```

```
ResultScanner rsScanner = table.getScanner (scanner) ;
```

```
for (Result res : rsScanner) {
```

```
    final List<KeyValue> list = res.list ( ) ;
```

```
    String rk = null;
```

```
    StringBuilder sb = new StringBuilder ( ) ;
```

```
    for (final KeyValue kv : list) {
```

```
        sb.append (Bytes.toStringBinary (kv.getValue ( ) ) + ", " ) ;
```

```
        rk = getRealRowKey (kv) ;
```

```
    }
```

```
    if (sb.toString ( ) .length ( ) > 0)
```

```
        sb.setLength (sb.toString ( ) .length ( ) - 1) ;
```

```
    System.out.println (rk + "\t" + sb.toString ( ) ) ;
```

```
}
```

```
rsScanner.close ( ) ;
```

6) 显式地指定列

当使用Scan或Get来处理大量的行时，最好确定一下所需要的列。因为服务器端处理完的结果，需要通过网络传输到客户端，而且此时，传输的数据量成为瓶颈，如果能有效地过滤部分数据，使用更精确的需求，能够很大程度上减少网络I/O的花费，否则会造成很大的资源浪费。如果在查询中指定某列或者某几列，**能够有效地减少网络传输量**，在一定程度上提升查询性能。下面代码是使用Scan类中指定列的addColumn () 方法。

代码示例：

```
HTable table = new HTable (config, Bytes.toBytes (tableName) ) ;
Scan scanner = new Scan () ;

/* 指定列 */
scanner.addColumn (Bytes.toBytes (columnFamily) , Bytes.toBytes (column) ) ;

ResultScanner rsScanner = table.getScanner (scanner) ;

for (Result res : rsScanner) {
    final List<KeyValue> list = res.list () ;
    String rk = null;
    StringBuilder sb = new StringBuilder () ;
    for (final KeyValue kv : list) {
        sb.append (Bytes.toStringBinary (kv.getValue () ) + ", " ) ;
        rk = getRealRowKey (kv) ;
    }
    if (sb.toString () .length () > 0)
        sb.setLength (sb.toString () .length () - 1 ) ;

    System.out.println (rk + "\t" + sb.toString () ) ;
}
rsScanner.close () ;
```

7) 关闭ResultScanner

ResultScanner类用于存储服务端扫描的最终结果，可以通过遍历该类获取查询结果。但是，如果不关闭该类，可能会出现服务端在一段时间内一直保存连接，资源无法释放，从而导致服务器端某些资源的不可用，还有可能引发RegionServer的其他问题。所以在使用完该类之后，需要执行关闭操作。这一点与JDBC操作MySQL类似，需要关闭连接。代码的最后一行rsScanner.close () 就是执行关闭ResultScanner。

8) 使用批量读

通过调用HTable.get (Get) 方法可以根据一个指定的行键获取HBase表中的一行记录。同样HBase提供了另一个方法，通过调用HTable.get (List<Get>) 方法可以根据一个指定的行键列表，批量获取多行记录。使用该方法可以在服务器端执行完批量查询后返回结果，降低网络传输的速度，节省网络I/O开销，对于数据实时性要求高且网络传输RTT高的场景，能带来明显的性能提升。

代码示例：

```
HTable table = new HTable (config, Bytes.toBytes (tableName) ) ;
```

```

Get get1 = new Get (ROW1) ;
Get get2 = new Get (ROW2) ;
Get get3 = new Get (ROW3) ;
List<Get> gets = new ArrayList<Get> () ;
gets.add (get1) ;
gets.add (get2) ;
gets.add (get3) ;

try {
    Result[] result = table.get (gets) ;
    return result;
} catch (IOException e) {
    e.printStackTrace () ;
    return null;
} finally {
    try {
        table.close () ;
    } catch (IOException e) {
        e.printStackTrace () ;
    }
}

```

9) 使用批量写

通过调用HTable.put (Put) 方法可以将一个指定的行键记录写入HBase, 同样HBase提供了另一个方法, 通过调用HTable.put (List<Put>) 方法可以将指定的多个行键批量写入。这样做的好处是批量执行, 减少网络I/O开销。

□□ 对于批量写入方法的使用见下面代码:

```

HTable table = new HTable (config, Bytes.toBytes (tableName) ) ;
Put put1 = new Put (ROW1) ;
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("mid") , Bytes.toBytes (123456) ) ;
Put put2 = new Put (ROW2) ;
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("mid") , Bytes.toBytes (123456) ) ;
Put put3 = new Put (ROW3) ;
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("mid") , Bytes.toBytes (123456) ) ;

List<Put> puts = new ArrayList<Put> () ;
puts.add (put1) ;
puts.add (put2) ;
puts.add (put3) ;

```

```

try {
    table.put (puts) ;
} catch (IOException e) {
    e.printStackTrace () ;
} finally {
    try {
        table.close () ;
    }
}

```

```

} catch (IOException e) {
    e.printStackTrace ();
}

```

10) 关闭写WAL日志

在默认情况下，为了保证系统的高可用性，写WAL日志是开启状态。写WAL开启或者关闭，在一定程度上确实会对系统性能产生很大影响，根据HBase内部设计，WAL是规避数据丢失风险的一种补偿机制，如果应用可以容忍一定的数据丢失的风险，可以尝试在更新数据时，关闭写WAL。该方法存在的风险是，当RegionServer宕机时，可能写入的数据会出现丢失的情况，且无法恢复。关闭写WAL操作通过Put类中的writeToWAL () 设置。

□ 具体的设置方法如下面代码所示：

```

long st = System.currentTimeMillis ();
Put put = new Put (Bytes.toBytes ("r1") );
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("mid") ,
Bytes.toBytes (123111) );
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("stat_hour") ,
Bytes.toBytes ("20") );
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("logdate") ,
Bytes.toBytes ("20121126") );
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("ditch") ,
Bytes.toBytes ("2") );
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("version") ,
Bytes.toBytes ("3.2.2.2") );
put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("type") ,
Bytes.toBytes ("2") );

```

```

put.setWriteToWAL (false) ;

```

```

table.put (put) ;
table.close () ;

```

```

long en = System.currentTimeMillis ();
System.out.println ("time: " + (en - st) + "... ms") ;

```

11) 设置AutoFlush

HTable有一个属性是AutoFlush，该属性用于支持客户端的批量更新。该属性默认值是true，即客户端每收到一条数据，立刻发送到服务端。如果将该属性设置为false，当客户端提交Put请求时，将该请求在客户端缓存，直到数据达到某个阈值的容量时（该容量由参数hbase.client.write.buffer决定）或执行hbase.flushcommits () 时，才向RegionServer提交请求。这种方式避免了每次跟服务端交互，采用批量提交的方式，所以更高效。

但是，如果还没有达到该缓存而客户端崩溃，该部分数据将由于未发送到RegionServer而丢失。这对于有些零容忍的在线服务是不可接受的。所

以，设置该参数的时候要慎重。

□ HTable设置AutoFlush的示例代码如下：

```
public static final boolean AUTO_FLUSH = false;
public static final int WRITE_BUFFER_SIZE = 12 * 1024 * 1024;

public void put () throws IOException {
    table.setAutoFlush (AUTO_FLUSH) ;
    table.setWriteBufferSize (WRITE_BUFFER_SIZE) ;

    long st = System.currentTimeMillis () ;
    Put put = null;

    for (int i = 0; i < 100000; i++) {
        put = new Put (Bytes.toBytes ("row1") , 10L) ;
        put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("mid") ,
            Bytes.toBytes (123111) ) ;
        put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("stat_hour") ,
            Bytes.toBytes ("20") ) ;
        put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("logdate") ,
            Bytes.toBytes ("20121126") ) ;
        put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("ditch") ,
            Bytes.toBytes ("2") ) ;
        put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("version") ,
            Bytes.toBytes ("3.2.2.2") ) ;
        put.add (Bytes.toBytes ("cf1") , Bytes.toBytes ("type") ,
            Bytes.toBytes ("2") ) ;
        put.setWriteToWAL (true) ;

        table.put (put) ;

        if ( (i % 1000) == 0) {
            System.out.println (i + " DOCUMENTS done! ") ;
        }
    }

    table.flushCommits () ;
    table.close () ;

    long en = System.currentTimeMillis () ;
    System.out.println ("time: " + (en - st) + "... ms") ;
}
```

12) 预创建Region

在HBase中创建表时，该表开始只有一个Region，插入该表的所有数据会保存在该Region中。随着数据量不断增加，当该Region大小达到一定阈值时，就会发生分裂（Region Splitting）操作。**并且在这个表创建后相当长的一段时间内，针对该表的所有写操作总是集中在某一台或者少数几台机器上**，这不仅仅造成局部磁盘和网络资源紧张，同时也是对整个集群资源的浪费。这个问题在初始化表，即批量导入原始数据的时候，特别明

显。为了解决这个问题，可以使用预创建Region的方法。

📖 **Hbase内部提供了RegionSplitter工具，使用命令如下：**

```
${HBASE_HOME}/bin/hbase org.apache.hadoop.hbase.util.RegionSplitter test2 HexStringSplit -c 10 -f cf1
```

其中，test2是表名，HexStringSplit表示划分的算法，参数-c 10表示预创建10个Region，-f cf1表示创建一个名字为cf1的列族。

13) 调整ZooKeeper Session的有效时长

参数zookeeper.session.timeout用于定义连接ZooKeeper的Session的有效时长，这个默认值是180秒。这意味着一旦某个RegionServer宕机，HMaster至少需要180秒才能察觉到宕机，然后开始恢复。或者客户端读写过程中，如果服务端不能提供服务，客户端直到180秒后才能觉察到。在某些场景中，这样的时长可能对生产线业务来讲不能容忍，需要调整这个值。

此参数在HBase-site.xml中，通过<property></property>

Phoenix介绍和安装


2017年11月24日 12:40

介绍

HBase基础上架构的SQL中间件。让我们可以通过SQL/JDBC来操作HBase

安装实现步骤:

1.上传Phoenix安装包到linux服务器并解压，这台linux服务器最好是Hbase Master节点。所以，如果是一个Hbase集群，我们不需要在全部的服务节点上来安装Phoenix。

 [apache-phoenix-4.8.1-HBase-0.98-bin.tar.gz](#)

2.将Phoenix安装目录下的两个jar包，拷贝到Hbase安装目录下的lib目录

phoenix-4.8.1-HBase-0.98-server.jar

phoenix-4.8.1-HBase-0.98-client.jar

cp phoenix-4.8.1-HBase-0.98-server.jar /home/software/hbase/lib

cp phoenix-4.8.1-HBase-0.98-client.jar /home/software/hbase/lib

3.在 etc/profile文件中 配置Hbase的目录路径

配置示例:

JAVA_HOME=/home/software/jdk1.8

HADOOP_HOME=/home/software/hadoop-2.7.1

HBASE_HOME=/home/software/hbase

CLASSPATH=.:\$JAVA_HOME/lib/dt.jar:\$JAVA_HOME/lib/tools.jar

PATH=\$JAVA_HOME/bin:\$HADOOP_HOME/bin:\$HADOOP_HOME/sbin:\$PATH

export JAVA_HOME PATH CLASSPATH HADOOP_HOME **HBASE_HOME**

注：配置完之后，别忘了source /etc/profile

4.启动Hadoop

5.启动Zk集群

6.启动Hbase集群

7.进入Phoenix安装目录的bin目录

执行: ./sqlline.py hadoop01,hadoop02,hadoop03:2181

```
Connected to: Phoenix (version 4.8)
Driver: PhoenixEmbeddedDriver (version 4.8)
Autocommit status: true
Transaction isolation: TRANSACTION_READ_COMMITTED
Building list of tables and columns for tab-completion (set fastconnect to true to skip)...
86/86 (100%) Done
Done
sqlline version 1.1.9
0: jdbc:phoenix:hadoop01,hadoop02,hadoop03:21>
```

如上图所以，证明Phoenix安装成功

此外，此时进入hbase，执行list查看，会发现多出如下的表：

```
=> ["SYSTEM.CATALOG", "SYSTEM.FUNCTION", "SYSTEM.SEQUENCE", "SYSTEM.STATS",
```

如何杀掉Phoenix进程

执行: pstree -p

查看 py进程，杀掉Sqlline的父进程

```
python(2449)
├─sshd(1948)─sshd(2437)─bash(2449)─pstree(3968)
└─python(3695)─java(3710)
```

Phoenix使用

2017年11月29日 15:22

1.创建表:

```
>create table tab1(id integer primary key,name varchar);
```

注:

- ①Phoenix建表必须有声明主键，否则报错
- ②Phoenix建表的表名，在hbase里的表名是大写的，此外，列名也是大写的。
- ③这条建表语句，并未声明表的列族，则默认就一个列族，且列族的名字为：0。
- ④在列族0中，除主键列外，其余的列都属于0列族里的列

2.查看所有表:

```
>! tables
```

```
0: jdbc:phoenix:hadoop01,hadoop02,hadoop03:21> !tables
```

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	TABLE_TYPE	REMARKS	TYPE_NAME	SELF_REFERENCING_COL_NAME
	SYSTEM	CATALOG	SYSTEM TABLE			
	SYSTEM	FUNCTION	SYSTEM TABLE			
	SYSTEM	SEQUENCE	SYSTEM TABLE			
	SYSTEM	STATS	SYSTEM TABLE			
		TAB1	TABLE			

3.插入数据:

```
upsert into tab1 values(1,'hello');
```

注: 字符串类型用 ' ' 包起来，不要用 " " ，否则报错。

4.查询数据:

```
select * from tab1;
```

ID	NAME
1	hello

5.删除数据:

```
delete from tab1 where id=2;
```

6.删除表:

```
drop table tab1;
```

7.创建小写的表名:

```
create table "tab2" (id integer primary key,name varchar);
create table "tab2" ("id" integer primary key,"name" varchar);
```

```
select * from "tab2";
```

注: CRUD都以 "tab2" 为表名来操作

8.自定义列族名

```
create table tab3 (id integer primary key,info.name varchar,info.age integer);
upsert into tab3 values(1,'tom',23);
```

然后在hbase里查看会发现：

describe 'TAB3'

```
{NAME => 'INFO', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KEEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'FAST_DIFF', TTL => 'FOREVER', COMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65536', REPLICATION_SCOPE => '0'}  
1 row(s) in 0.0650 seconds
```

scan 'TAB3'

```
hbase(main):016:0> scan 'TAB3'  
ROW COLUMN+CELL  
  \x80\x00\x00\x01 column=INFO:AGE, timestamp=1511999638782, value=\x80\x00\x00\x17  
  \x80\x00\x00\x01 column=INFO:NAME, timestamp=1511999638782, value=tom  
  \x80\x00\x00\x01 column=INFO:_0, timestamp=1511999638782, value=x  
1 row(s) in 0.0220 seconds
```

扩展：LSM-TREE

2017年11月29日 21:41

概述

众所周知传统磁盘I/O是比较耗性能的，优化系统性能往往需要和磁盘I/O打交道,而磁盘I/O产生的时延主要由下面3个因素决定:

- 1) 寻道时间（将磁盘臂移动到适当的柱面上所需要的时间，寻道时移动到相邻柱面移动所需时间1ms，而随机移动所需时间位5~10ms)
- 2) 旋转时间（等待适当的扇区旋转到磁头下所需要的时间)
- 3) 实际数据传输时间(低端硬盘的传输速率为5MB/ms，而高速硬盘的速率是10MB/ms)

近20年平均寻道时间改进了7倍，传输速率改进了1300倍，而容量的改进则高达50000倍，这一格局主要是因为磁盘中运动部件的改进相对缓慢和渐进，而记录表面则达到了相当高的密度。对于一个块的访问完全由寻道时间和旋转延迟所决定，所以花费相同时间访问一个盘块，那么取的数据越多越好。

磁盘I/O瓶颈可能出现在seek(寻道)和transfer(数据传输)上面。

根据磁盘I/O类型，关系型存储引擎中广泛使用的B树及B+树，而Bigtable的存储架构基础的会使用Log-Structured Merge Tree。

B- Tree和B+Tree

如果没有太多的写操作，B+树可以工作的很好，它会进行比较繁重的优化来保证较低的访问时间。而写操作往往是随机的，随机写到磁盘的不同位置上，更新和删除都是以磁盘seek的速率级别进行的。RDBMS通常都是Seek型的，主要是由用于存储数据的B树或者是B+树结构引起的，在磁盘seek的速率级别上实现各种操作，通常每个访问需要 $\log(N)$ 个seek操作

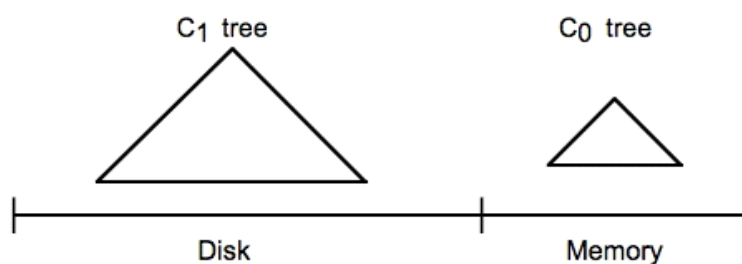
LSM-Tree

而LSM-tree工作在磁盘传输速率的级别上，可以更好地扩展到更大的数据规模上，**保证一个比较一致的插入速率，因为它会使用日志文件和一个内存存储结构，将随机写操作转化为顺序写。**

在传输等量数据场景下，随机写I/O的时延大部分花费在了seek操作上，数据库对磁盘进行零碎的随机写会产生多次seek操作；而顺序存取只需一次seek操作，便可以传输大量数据，针对批量写入大量数据的场景，顺序写比随机写具有明显的优势。

The Log-Structured Merge-Tree(LSM-Tree)的一个重要思想就是通过使用某种算法，**该算法会对索引变更进行延迟及批量处理，并通过一种类似于归并排序的方式高效地将更新迁移到磁盘，进行批量写入，利用磁盘顺序写性能远好于随机写这一特点，将随机写转变为顺序写，从而保证对磁盘的操作是顺序的**，以提升写性能，同时建立索引，以获取较快的读性能，在读和写性能之间做一个平衡。

LSM-Tree原理



c0 Tree 是存在内存的的树结构，可以是（B-树，B+树，二叉树，跳跃表）

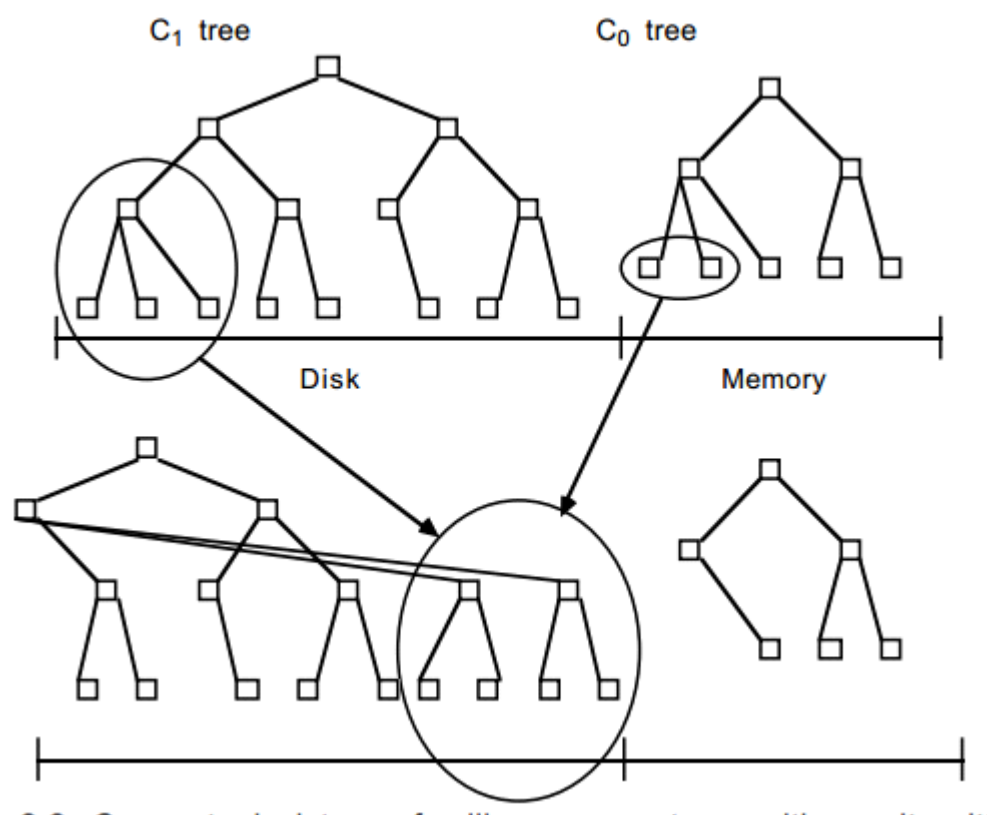
c1 Tree 是存在磁盘上的文件（本身也是一个树结构）

写入或者更新某条记录时，首先会预写日志，用于数据写入失败时进行数据恢复。之后该条记录会被插入到驻留在内存中的c0树，在符合某个条件的时候从被移到磁盘上的c1树中。

C0树不一定要具有一个类B-树的结构。HBase中采用了线程安全的ConcurrentSkipListMap

数据结构。

向内存中的C0树插入一个条目速度是非常快的，因为操作不会产生磁盘I/O开销。然而用于C0的内存成本要远高于磁盘，通常做法是限制它的大小。采用一种有效的方式来将记录迁移到驻留在更低成本的存储设备上的C1树中。为了实现这个目的，在当C0树因插入操作而达到接近某个上限的阈值大小时，就会启动一个rolling merge过程，**来将某些连续的记录段（保证是顺序写）**从C0树中删除，并merge到磁盘上的C1树中。



磁盘上的C1树是一个类似于B-Tree的数据结构，但是它是为顺序性的磁盘访问优化过的。

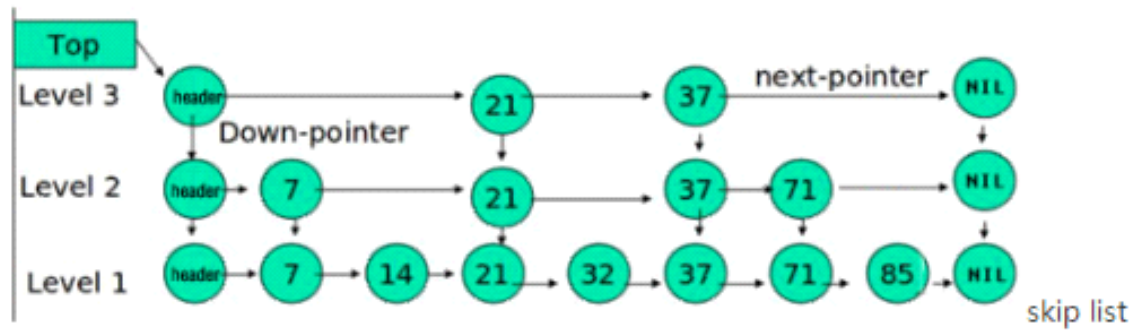
HBase的实现

MemStore

MemStore是HBase中C0的实现，向HBase中写数据的时候，首先会写到内存中的MemStore,当达到一定阈值之后，flush(顺序写)到磁盘，形成新的StoreFile（HFile），最后多个StoreFile（HFile）又会进行Compact。

memstore内部维护了一个数据结构：ConcurrentSkipListMap，数据存储是按照RowKey排

好序的跳跃列表。跳跃列表的算法有同平衡树一样的渐进的预期时间边界，并且更简单、更快和使用更少的空间。



HFile

HFile是lsm tree中C1的实现