# 第1章 HBase简介

## 1.1 HBase定义

HBase是一种分布式、可扩展、支持海量数据存储的NoSQL数据库。

## 1.2 HBase数据模型

逻辑上，HBase的数据模型同关系型数据库很类似，数据存储在一张表中，有行有列。但从HBase的底层物理存储结构（K-V）来看，HBase更像是一个multi-dimensional map。

### 1.2.1 数据模型

**1）Name Space**

命名空间，类似于关系型数据库的database概念，每个命名空间下有多个表。HBase两个自带的命名空间，分别是hbase和default，hbase中存放的是HBase内置的表，default表是用户默认使用的命名空间。

**2）Table**

类似于关系型数据库的表概念。不同的是，HBase定义表时只需要声明列族即可，不需要声明具体的列。这意味着，往HBase写入数据时，字段可以动态、按需指定。因此，和关系型数据库相比，HBase能够轻松应对字段变更的场景。

**3）Row**

HBase表中的每行数据都由一个**RowKey**和多个**Column**（列）组成，数据是按照RowKey的字典顺序存储的，并且查询数据时只能根据RowKey进行检索，所以RowKey的设计十分重要。

**4）Column**

HBase中的每个列都由Column Family(列族)和Column Qualifier（列限定符）进行限定，例如info：name，info：age。建表时，只需指明列族，而列限定符无需预先定义。

**5）Time Stamp**

用于标识数据的不同版本（version），每条数据写入时，系统会自动为其加上该字段，其值为写入HBase的时间。

**6）Cell**

由{rowkey, column Family：column Qualifier, time Stamp} 唯一确定的单元。cell中的数据全部是字节码形式存贮。

## 1.3 HBase基本架构

**架构角色：**

**1）Region Server**

Region Server为 Region的管理者，其实现类为HRegionServer，主要作用如下:

对于数据的操作：get, put, delete；

对于Region的操作：splitRegion、compactRegion。

**2）Master**

Master是所有Region Server的管理者，其实现类为HMaster，主要作用如下：

对于表的操作：create, delete, alter

对于RegionServer的操作：分配regions到每个RegionServer，监控每个RegionServer的状态，负载均衡和故障转移。

**3）Zookeeper**

HBase通过Zookeeper来做master的高可用、RegionServer的监控、元数据的入口以及集群配置的维护等工作。

**4）HDFS**

HDFS为Hbase提供最终的底层数据存储服务，同时为HBase提供高可用的支持。

# 第2章 HBase安装

## 2.1 HBase安装部署

### 2.1.1 Zookeeper正常部署

首先保证Zookeeper集群的正常部署，并启动之：

[user1@hadoop102 zookeeper-3.5.7]$ bin/zkServer.sh start

[user1@hadoop103 zookeeper-3.5.7]$ bin/zkServer.sh start

[user1@hadoop104 zookeeper-3.5.7]$ bin/zkServer.sh start

### 2.1.2 Hadoop正常部署

Hadoop集群的正常部署并启动：

[user1@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh

[user1@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh

### 2.1.3 HBase的解压

解压Hbase到指定目录：

[user1@hadoop102 software]$ tar -zxvf hbase-2.0.5-bin.tar.gz -C /opt/module

[user1@hadoop102 software]$ mv /opt/module/hbase-2.0.5 /opt/module/hbase

配置环境变量

[user1@hadoop102 ~]$ sudo vim /etc/profile.d/my\_env.sh

添加

#HBASE\_HOME

export HBASE\_HOME=/opt/module/hbase

export PATH=$PATH:$HBASE\_HOME/bin

### 2.1.4 HBase的配置文件

修改HBase对应的配置文件。

1.hbase-env.sh修改内容：

export HBASE\_MANAGES\_ZK=false

2.hbase-site.xml修改内容：

<configuration>

<property>

<name>hbase.rootdir</name>

<value>hdfs://hadoop102:8020/hbase</value>

</property>

<property>

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

<value>true</value>

</property>

<property>

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

<value>hadoop102,hadoop103,hadoop104</value>

</property>

</configuration>

3.regionservers：

hadoop102

hadoop103

hadoop104

### 2.1.5 HBase远程发送到其他集群

[user1@hadoop102 module]$ xsync hbase/

### 2.1.6 HBase服务的启动

1.单点启动

[user1@hadoop102 hbase]$ bin/hbase-daemon.sh start master

[user1@hadoop102 hbase]$ bin/hbase-daemon.sh start regionserver

提示：如果集群之间的节点时间不同步，会导致regionserver无法启动，抛出ClockOutOfSyncException异常。

修复提示：

a、同步时间服务

b、属性：hbase.master.maxclockskew设置更大的值

|  |
| --- |
| <property>  <name>hbase.master.maxclockskew</name>  <value>180000</value>  <description>Time difference of regionserver from master</description>  </property> |

2.群启

[user1@hadoop102 hbase]$ bin/start-hbase.sh

对应的停止服务：

[user1@hadoop102 hbase]$ bin/stop-hbase.sh

### 2.1.7 查看HBase页面

启动成功后，可以通过“host:port”的方式来访问HBase管理页面，例如：

[http://hadoop102:16010](http://linux01:16010)

### 2.1.8 高可用

在HBase中HMaster负责监控HRegionServer的生命周期，均衡RegionServer的负载，如果HMaster挂掉了，那么整个HBase集群将陷入不健康的状态，并且此时的工作状态并不会维持太久。所以HBase支持对HMaster的高可用配置。

1.关闭HBase集群（如果没有开启则跳过此步）

[user1@hadoop102 hbase]$ bin/stop-hbase.sh

2.在conf目录下创建backup-masters文件

[user1@hadoop102 hbase]$ touch conf/backup-masters

3.在backup-masters文件中配置高可用HMaster节点

[user1@hadoop102 hbase]$ echo hadoop103 > conf/backup-masters

4.将整个conf目录scp到其他节点

[user1@hadoop102 hbase]$ scp -r conf/ hadoop103:/opt/module/hbase/

[user1@hadoop102 hbase]$ scp -r conf/ hadoop104:/opt/module/hbase/

5.打开页面测试查看

[http://hadooo102:16010](http://linux01:16010)

## 2.2 HBase Shell操作

### 2.2.1 基本操作

**1．进入HBase客户端命令行**

[user1@hadoop102 hbase]$ bin/hbase shell

**2．查看帮助命令**

hbase(main):001:0> help

### 2.2.2 namespace的操作

**1．查看当前Hbase中有哪些namespace**

hbase(main):002:0> list\_namespace

NAMESPACE

default(创建表时未指定命名空间的话默认在default下)

hbase(系统使用的，用来存放系统相关的元数据信息等，勿随便操作)

**2．创建namespace**

hbase(main):010:0> create\_namespace "test"

hbase(main):010:0> create\_namespace "test01", {"author"=>"wyh", "create\_time"=>"2020-03-10 08:08:08"}

**3.查看namespace**

hbase(main):010:0> describe\_namespace "test01"

**4.修改namespace的信息（添加或者修改属性）**

hbase(main):010:0> alter\_namespace "test01", {METHOD => 'set', 'author' => 'weiyunhui'}

**添加或者修改属性:**

alter\_namespace 'ns1', {METHOD => 'set', 'PROPERTY\_NAME' => 'PROPERTY\_VALUE'}

**删除属性:**

alter\_namespace 'ns1', {METHOD => 'unset', NAME => ' PROPERTY\_NAME '}

**5.删除namespace**

hbase(main):010:0> drop\_namespace "test01"

注意: 要删除的namespace必须是空的，其下没有表。

### 2.2.3 表的操作

**0．查看当前数据库中有哪些表**

hbase(main):002:0> list

**1．创建表**

hbase(main):002:0> create 'student','info'

**2．插入数据到表**

hbase(main):003:0> put 'student','1001','info:sex','male'

hbase(main):004:0> put 'student','1001','info:age','18'

hbase(main):005:0> put 'student','1002','info:name','Janna'

hbase(main):006:0> put 'student','1002','info:sex','female'

hbase(main):007:0> put 'student','1002','info:age','20'

**3．扫描查看表数据**

hbase(main):008:0> scan 'student'

hbase(main):009:0> scan 'student',{STARTROW => '1001', STOPROW => '1001'}

hbase(main):010:0> scan 'student',{STARTROW => '1001'}

**4．查看表结构**

hbase(main):011:0> describe 'student'

**5．更新指定字段的数据**

hbase(main):012:0> put 'student','1001','info:name','Nick'

hbase(main):013:0> put 'student','1001','info:age','100'

**6．查看“指定行”或“指定列族:列”的数据**

hbase(main):014:0> get 'student','1001'

hbase(main):015:0> get 'student','1001','info:name'

**7．统计表数据行数**

hbase(main):021:0> count 'student'

**8．删除数据**

删除某rowkey的全部数据：

hbase(main):016:0> deleteall 'student','1001'

删除某rowkey的某一列数据：

hbase(main):017:0> delete 'student','1002','info:sex'

**9．清空表数据**

hbase(main):018:0> truncate 'student'

提示：清空表的操作顺序为先disable，然后再truncate。

**10．删除表**

首先需要先让该表为disable状态：

hbase(main):019:0> disable 'student'

然后才能drop这个表：

hbase(main):020:0> drop 'student'

提示：如果直接drop表，会报错：ERROR: Table student is enabled. Disable it first.

**11．变更表信息**

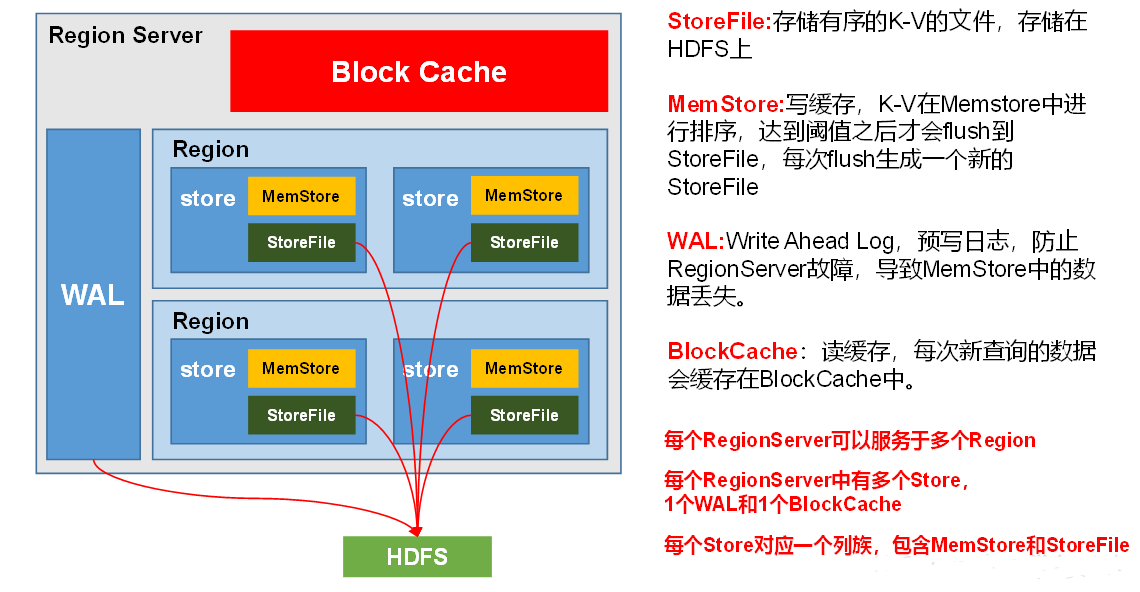
将info列族中的数据存放3个版本：

hbase(main):022:0> alter 'student',{NAME=>'info',VERSIONS=>3}

hbase(main):022:0> get 'student','1001',{COLUMN=>'info:name',VERSIONS=>3}

# 第3章 HBase进阶

## 3.1 RegionServer 架构



**1）StoreFile**

保存实际数据的物理文件，StoreFile以Hfile的形式存储在HDFS上。每个Store会有一个或多个StoreFile（HFile），数据在每个StoreFile中都是有序的。

**2）MemStore**

写缓存，由于HFile中的数据要求是有序的，所以数据是先存储在MemStore中，排好序后，等到达刷写时机才会刷写到HFile，每次刷写都会形成一个新的HFile。

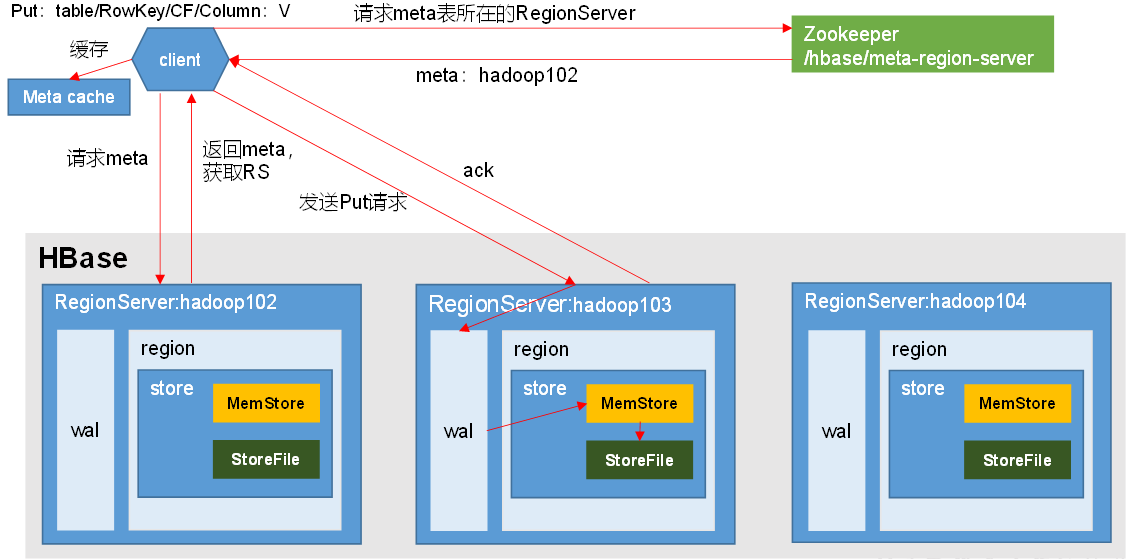
**3）WAL**

由于数据要经MemStore排序后才能刷写到HFile，但把数据保存在内存中会有很高的概率导致数据丢失，为了解决这个问题，数据会先写在一个叫做Write-Ahead logfile的文件中，然后再写入MemStore中。所以在系统出现故障的时候，数据可以通过这个日志文件重建。

**4）BlockCache**

读缓存，每次查询出的数据会缓存在BlockCache中，方便下次查询。

## 3.2 写流程



写流程：

1）Client先访问zookeeper，获取hbase:meta表位于哪个Region Server。

2）访问对应的Region Server，获取hbase:meta表，根据读请求的namespace:table/rowkey，查询出目标数据位于哪个Region Server中的哪个Region中。并将该table的region信息以及meta表的位置信息缓存在客户端的meta cache，方便下次访问。

3）与目标Region Server进行通讯；

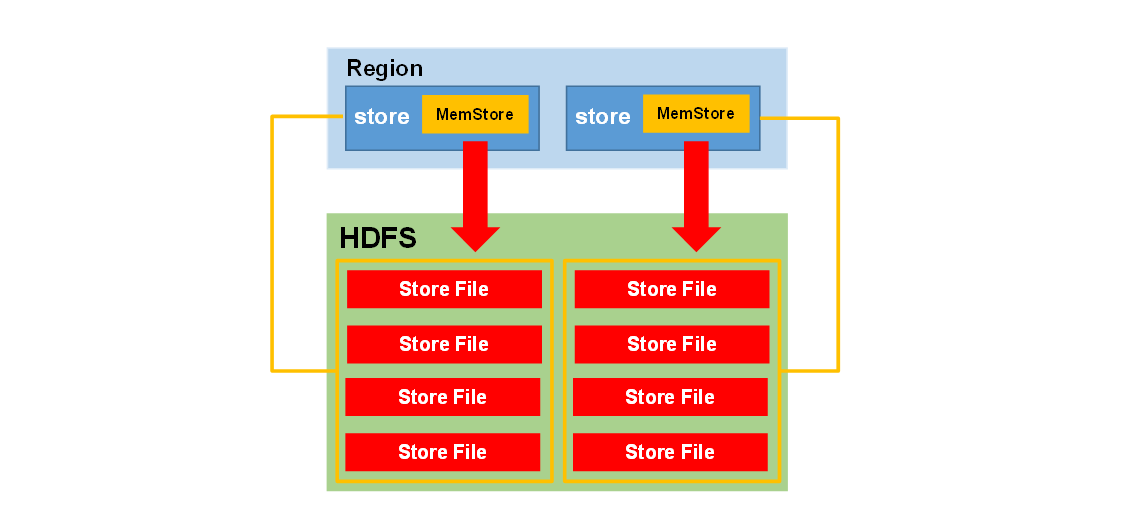
4）将数据顺序写入（追加）到WAL；

5）将数据写入对应的MemStore，数据会在MemStore进行排序；

6）向客户端发送ack；

7）等达到MemStore的刷写时机后，将数据刷写到HFile。

## 3.3 MemStore Flush



**MemStore刷写时机：**

1.当某个memstore的大小达到了**hbase.hregion.memstore.flush.size（默认值128M）**，其所在region的所有memstore都会刷写。

当memstore的大小达到了

**hbase.hregion.memstore.flush.size（默认值128M）**

**\*** **hbase.hregion.memstore.block.multiplier（默认值4）**

时，会阻止继续往该memstore写数据。

2.当region server中memstore的总大小达到

**java\_heapsize(默认1G)**

**\*hbase.regionserver.global.memstore.size（默认值0.4）**

**\*hbase.regionserver.global.memstore.size.lower.limit（默认值0.95）**，

region会按照其所有memstore的大小顺序（由大到小）依次进行刷写。直到region server中所有memstore的总大小减小到上述值以下。

当region server中memstore的总大小达到

**java\_heapsize**

**\*hbase.regionserver.global.memstore.size（默认值0.4）**

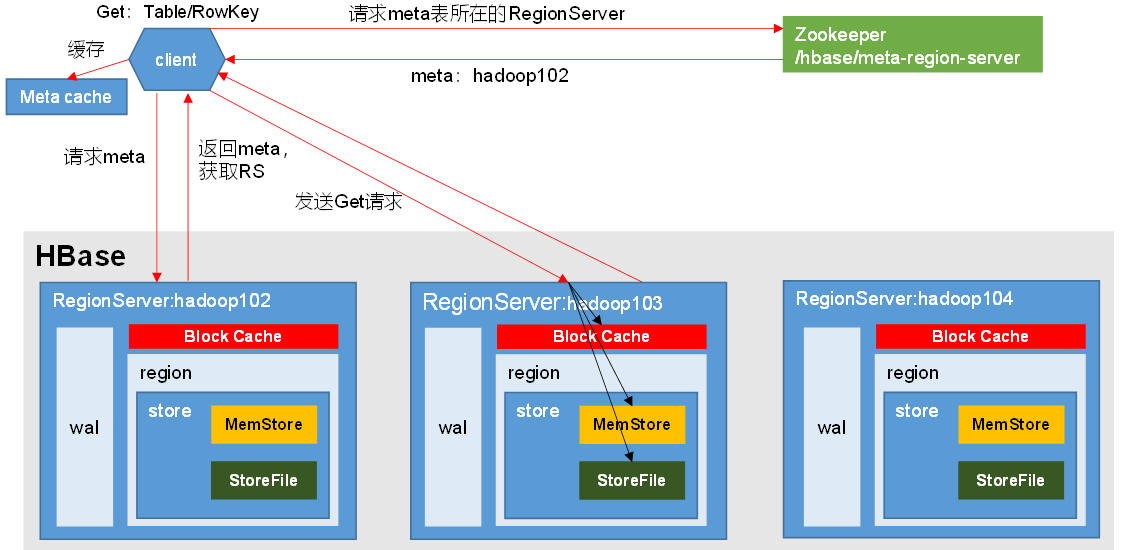
时，会阻止继续往所有的memstore写数据。

3. 到达自动刷写的时间，也会触发memstore flush。自动刷新的时间间隔由该属性进行配置**hbase.regionserver.optionalcacheflushinterval（默认1小时）**。

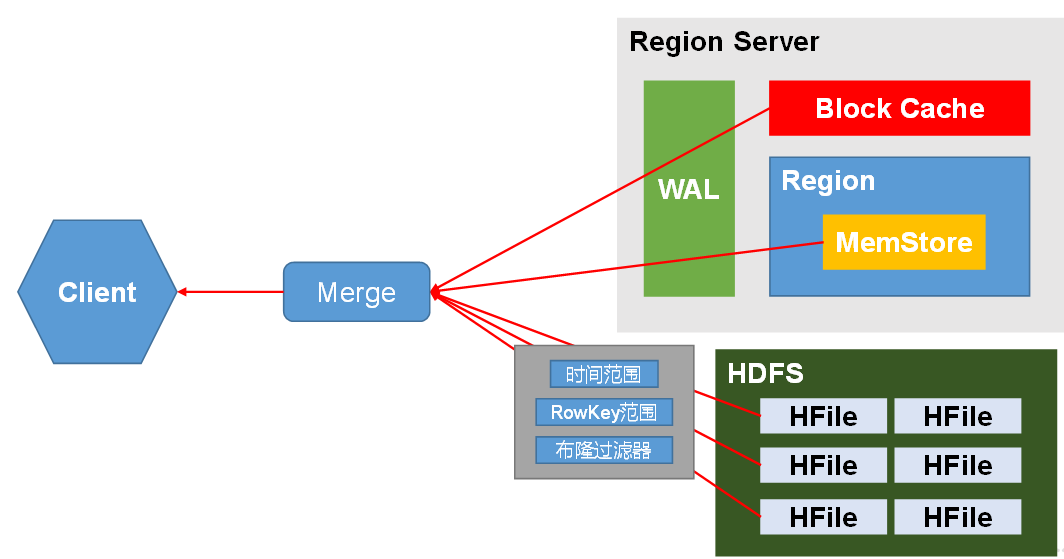
4.当WAL文件的数量超过**hbase.regionserver.max.logs**，region会按照时间顺序依次进行刷写，直到WAL文件数量减小到**hbase.regionserver.max.logs**以下（该属性名已经废弃，现无需手动设置，最大值为32）。

## 3.4 读流程

1）整体流程



2）Merge细节



**读流程**

1）Client先访问zookeeper，获取hbase:meta表位于哪个Region Server。

2）访问对应的Region Server，获取hbase:meta表，根据读请求的namespace:table/rowkey，查询出目标数据位于哪个Region Server中的哪个Region中。并将该table的region信息以及meta表的位置信息缓存在客户端的meta cache，方便下次访问。

3）与目标Region Server进行通讯；

4）分别在MemStore和Store File（HFile）中查询目标数据，并将查到的所有数据进行合并。此处所有数据是指同一条数据的不同版本（time stamp）或者不同的类型（Put/Delete）。

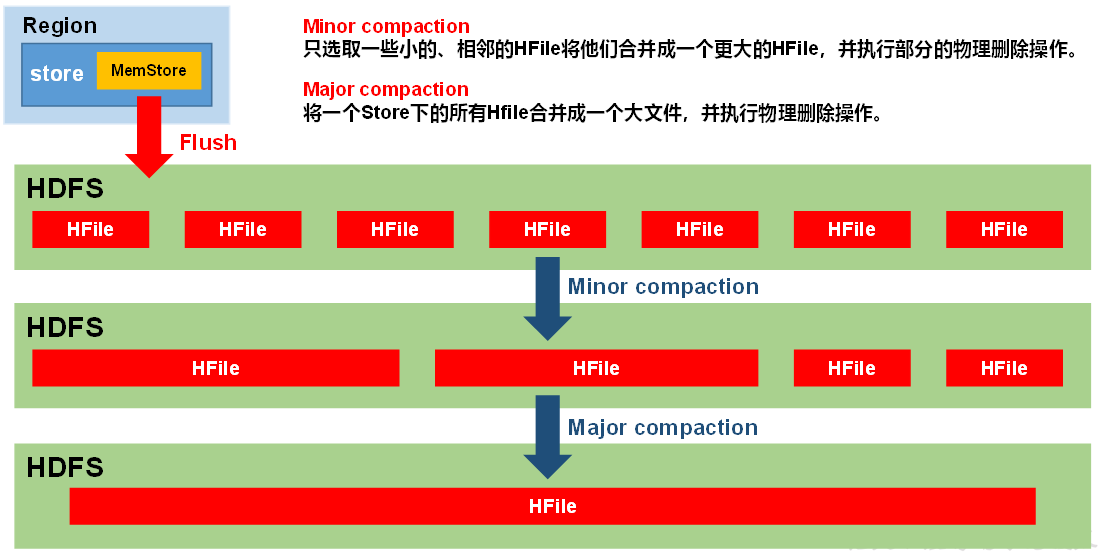
5）将查询到的新的数据块（Block，HFile数据存储单元，默认大小为64KB）缓存到Block Cache。

6）将合并后的最终结果返回给客户端。

## 3.5 StoreFile Compaction

由于memstore每次刷写都会生成一个新的HFile，且同一个字段的不同版本（timestamp）和不同类型（Put/Delete）有可能会分布在不同的HFile中，因此查询时需要遍历所有的HFile。为了减少HFile的个数，以及清理掉过期和删除的数据，会进行StoreFile Compaction。

Compaction分为两种，分别是Minor Compaction和Major Compaction。Minor Compaction会将临近的若干个较小的HFile合并成一个较大的HFile，并清理掉部分过期和删除的数据。Major Compaction会将一个Store下的所有的HFile合并成一个大HFile，并且**会**清理掉所有过期和删除的数据。



## 3.6 Region Split

默认情况下，每个Table起初只有一个Region，随着数据的不断写入，Region会自动进行拆分。刚拆分时，两个子Region都位于当前的Region Server，但处于负载均衡的考虑，HMaster有可能会将某个Region转移给其他的Region Server。

Region Split时机：

1.当1个region中的某个Store下所有StoreFile的总大小超过hbase.hregion.max.filesize，该Region就会进行拆分（0.94版本之前）。

2.当1个region中的某个Store下所有StoreFile的总大小超过Min(initialSize\*R^3 ,hbase.hregion.max.filesize")，该Region就会进行拆分。其中initialSize的默认值为2\*hbase.hregion.memstore.flush.size，R为当前Region Server中属于该Table的Region个数（0.94版本之后）。

具体的切分策略为：

第一次split：1^3 \* 256 = 256MB

第二次split：2^3 \* 256 = 2048MB

第三次split：3^3 \* 256 = 6912MB

第四次split：4^3 \* 256 = 16384MB > 10GB，因此取较小的值10GB

后面每次split的size都是10GB了。

3.Hbase 2.0引入了新的split策略：如果当前RegionServer上该表只有一个Region，按照2 \* hbase.hregion.memstore.flush.size分裂，否则按照hbase.hregion.max.filesize分裂。



# 第4章 HBase API

## 4.1 环境准备

新建项目后在pom.xml中添加依赖：

|  |
| --- |
| <dependency>  <groupId>org.apache.hbase</groupId>  <artifactId>hbase-server</artifactId>  <version>2.0.5</version>  </dependency>  <dependency>  <groupId>org.apache.hbase</groupId>  <artifactId>hbase-client</artifactId>  <version>2.0.5</version>  </dependency> |

## 4.2 DDL

创建HBase\_DDL类

### 4.2.1 判断表是否存在

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.NamespaceDescriptor;

import org.apache.hadoop.hbase.NamespaceExistException;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBase\_DDL {

//TODO 判断表是否存在

public static boolean isTableExist(String tableName) throws IOException {

//1.创建配置信息并配置

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

//2.获取与HBase的连接

Connection connection = ConnectionFactory.createConnection(configuration);

//3.获取DDL操作对象

Admin admin = connection.getAdmin();

//4.判断表是否存在操作

boolean exists = admin.tableExists(TableName.valueOf(tableName));

//5.关闭连接

admin.close();

connection.close();

//6.返回结果

return exists;

}

}

### 4.2.2 创建表

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.NamespaceDescriptor;

import org.apache.hadoop.hbase.NamespaceExistException;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBase\_DDL {

//TODO 创建表

public static void createTable(String tableName, String... cfs) throws IOException {

//1.判断是否存在列族信息

if (cfs.length <= 0) {

System.out.println("请设置列族信息！");

return;

}

//2.判断表是否存在

if (isTableExist(tableName)) {

System.out.println("需要创建的表已存在！");

return;

}

//3.创建配置信息并配置

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

//4.获取与HBase的连接

Connection connection = ConnectionFactory.createConnection(configuration);

//5.获取DDL操作对象

Admin admin = connection.getAdmin();

//6.创建表描述器构造器

TableDescriptorBuilder tableDescriptorBuilder = TableDescriptorBuilder.newBuilder(TableName.valueOf(tableName));

//7.循环添加列族信息

for (String cf : cfs) {

ColumnFamilyDescriptorBuilder columnFamilyDescriptorBuilder = ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes(cf));

tableDescriptorBuilder.setColumnFamily(columnFamilyDescriptorBuilder.build());

}

//8.执行创建表的操作

admin.createTable(tableDescriptorBuilder.build());

//9.关闭资源

admin.close();

connection.close();

}

}

### 4.2.3 删除表

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.NamespaceDescriptor;

import org.apache.hadoop.hbase.NamespaceExistException;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBase\_DDL {

//TODO 删除表

public static void dropTable(String tableName) throws IOException {

//1.判断表是否存在

if (!isTableExist(tableName)) {

System.out.println("需要删除的表不存在！");

return;

}

//2.创建配置信息并配置

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

//3.获取与HBase的连接

Connection connection = ConnectionFactory.createConnection(configuration);

//4.获取DDL操作对象

Admin admin = connection.getAdmin();

//5.使表下线

TableName name = TableName.valueOf(tableName);

admin.disableTable(name);

//6.执行删除表操作

admin.deleteTable(name);

//7.关闭资源

admin.close();

connection.close();

}

}

### 4.2.4 创建命名空间

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.NamespaceDescriptor;

import org.apache.hadoop.hbase.NamespaceExistException;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBase\_DDL {

//TODO 创建命名空间

public static void createNameSpace(String ns) throws IOException {

//1.创建配置信息并配置

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

//2.获取与HBase的连接

Connection connection = ConnectionFactory.createConnection(configuration);

//3.获取DDL操作对象

Admin admin = connection.getAdmin();

//4.创建命名空间描述器

NamespaceDescriptor namespaceDescriptor = NamespaceDescriptor.create(ns).build();

//5.执行创建命名空间操作

try {

admin.createNamespace(namespaceDescriptor);

} catch (NamespaceExistException e) {

System.out.println("命名空间已存在！");

} catch (Exception e) {

e.printStackTrace();

}

//6.关闭连接

admin.close();

connection.close();

}

}

## 4.3 DML

创建类HBase\_DML

### 4.3.1 插入数据

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.Cell;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBase\_DML {

//TODO 插入数据

public static void putData(String tableName, String rowKey, String cf, String cn, String value) throws IOException {

//1.获取配置信息并设置连接参数

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

//2.获取连接

Connection connection = ConnectionFactory.createConnection(configuration);

//3.获取表的连接

Table table = connection.getTable(TableName.valueOf(tableName));

//4.创建Put对象

Put put = new Put(Bytes.toBytes(rowKey));

//5.放入数据

put.addColumn(Bytes.toBytes(cf), Bytes.toBytes(cn), Bytes.toBytes(value));

//6.执行插入数据操作

table.put(put);

//7.关闭连接

table.close();

connection.close();

}

}

### 4.3.2 单条数据查询

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.Cell;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBase\_DML {

//TODO 单条数据查询(GET)

public static void getDate(String tableName, String rowKey, String cf, String cn) throws IOException {

//1.获取配置信息并设置连接参数

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

//2.获取连接

Connection connection = ConnectionFactory.createConnection(configuration);

//3.获取表的连接

Table table = connection.getTable(TableName.valueOf(tableName));

//4.创建Get对象

Get get = new Get(Bytes.toBytes(rowKey));

// 指定列族查询

// get.addFamily(Bytes.toBytes(cf));

// 指定列族:列查询

// get.addColumn(Bytes.toBytes(cf), Bytes.toBytes(cn));

//5.查询数据

Result result = table.get(get);

//6.解析result

for (Cell cell : result.rawCells()) {

System.out.println("CF:" + Bytes.toString(cell.getFamilyArray()) +

",CN:" + Bytes.toString(cell.getQualifierArray()) +

",Value:" + Bytes.toString(cell.getValueArray()));

}

//7.关闭连接

table.close();

connection.close();

}

}

### 4.3.3 扫描数据

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.Cell;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBase\_DML {

//TODO 扫描数据(Scan)

public static void scanTable(String tableName) throws IOException {

//1.获取配置信息并设置连接参数

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

//2.获取连接

Connection connection = ConnectionFactory.createConnection(configuration);

//3.获取表的连接

Table table = connection.getTable(TableName.valueOf(tableName));

//4.创建Scan对象

Scan scan = new Scan();

//5.扫描数据

ResultScanner results = table.getScanner(scan);

//6.解析results

for (Result result : results) {

for (Cell cell : result.rawCells()) {

System.out.println(

Bytes.toString(CellUtil.cloneRow(cell))+":"+

Bytes.toString(CellUtil.cloneFamily(cell))+":" +

Bytes.toString(CellUtil.cloneQualifier(cell)) +":" +

Bytes.toString(CellUtil.cloneValue(cell))

);

}

}

//7.关闭资源

table.close();

connection.close();

}

}

### 4.3.4 删除数据

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.Cell;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.\*;

import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HBase\_DML {

//TODO 删除数据

public static void deletaData(String tableName, String rowKey, String cf, String cn) throws IOException {

//1.获取配置信息并设置连接参数

Configuration configuration = HBaseConfiguration.create();

configuration.set("hbase.zookeeper.quorum", "hadoop102,hadoop103,hadoop104");

//2.获取连接

Connection connection = ConnectionFactory.createConnection(configuration);

//3.获取表的连接

Table table = connection.getTable(TableName.valueOf(tableName));

//4.创建Delete对象

Delete delete = new Delete(Bytes.toBytes(rowKey));

// 指定列族删除数据

// delete.addFamily(Bytes.toBytes(cf));

// 指定列族:列删除数据(所有版本)

// delete.addColumn(Bytes.toBytes(cf), Bytes.toBytes(cn));

// 指定列族:列删除数据(指定版本)

// delete.addColumns(Bytes.toBytes(cf), Bytes.toBytes(cn));

//5.执行删除数据操作

table.delete(delete);

//6.关闭资源

table.close();

connection.close();

}

}

# 第5章 HBase优化

## 5.1 预分区

每一个region维护着startRow与endRowKey，如果加入的数据符合某个region维护的rowKey范围，则该数据交给这个region维护。那么依照这个原则，我们可以将数据所要投放的分区提前大致的规划好，以提高HBase性能。

**1.手动设定预分区**

hbase> create 'staff1','info',SPLITS => ['1000','2000','3000','4000']

**2.生成16进制序列预分区**

create 'staff2','info',{NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}

**3.按照文件中设置的规则预分区**

创建splits.txt文件内容如下：

|  |
| --- |
| aaaa  bbbb  cccc  dddd |

然后执行：

create 'staff3','info',SPLITS\_FILE => 'splits.txt'

**4.使用JavaAPI创建预分区**

|  |
| --- |
| //自定义算法，产生一系列Hash散列值存储在二维数组中  byte[][] splitKeys = 某个散列值函数  //创建HbaseAdmin实例  HBaseAdmin hAdmin = new HBaseAdmin(HbaseConfiguration.create());  //创建HTableDescriptor实例  HTableDescriptor tableDesc = new HTableDescriptor(tableName);  //通过HTableDescriptor实例和散列值二维数组创建带有预分区的Hbase表  hAdmin.createTable(tableDesc, splitKeys); |

## 5.2 RowKey设计

一条数据的唯一标识就是rowkey，那么这条数据存储于哪个分区，取决于rowkey处于哪个一个预分区的区间内，设计rowkey的主要目的 ，就是让数据均匀的分布于所有的region中，在一定程度上防止数据倾斜。接下来我们就谈一谈rowkey常用的设计方案。

**1.生成随机数、hash、散列值**

|  |
| --- |
| 比如：  原本rowKey为1001的，SHA1后变成：dd01903921ea24941c26a48f2cec24e0bb0e8cc7  原本rowKey为3001的，SHA1后变成：49042c54de64a1e9bf0b33e00245660ef92dc7bd  原本rowKey为5001的，SHA1后变成：7b61dec07e02c188790670af43e717f0f46e8913  在做此操作之前，一般我们会选择从数据集中抽取样本，来决定什么样的rowKey来Hash后作为每个分区的临界值。 |

**2.字符串反转**

|  |
| --- |
| 20170524000001转成10000042507102  20170524000002转成20000042507102 |

这样也可以在一定程度上散列逐步put进来的数据。

**3.字符串拼接**

|  |
| --- |
| 20170524000001\_a12e  20170524000001\_93i7 |

## 5.3 内存优化

HBase操作过程中需要大量的内存开销，毕竟Table是可以缓存在内存中的，但是不建议分配非常大的堆内存，因为GC过程持续太久会导致RegionServer处于长期不可用状态，一般16~36G内存就可以了，如果因为框架占用内存过高导致系统内存不足，框架一样会被系统服务拖死。

## 5.4 基础优化

**1.Zookeeper会话超时时间**

hbase-site.xml

属性：zookeeper.session.timeout

解释：默认值为90000毫秒（90s）。当某个RegionServer挂掉，90s之后Master才能察觉到。可适当减小此值，以加快Master响应，可调整至600000毫秒。

**2.设置RPC监听数量**

hbase-site.xml

|  |
| --- |
| 属性：hbase.regionserver.handler.count  解释：默认值为30，用于指定RPC监听的数量，可以根据客户端的请求数进行调整，读写请求较多时，增加此值。 |

**3.手动控制Major Compaction**

hbase-site.xml

属性：hbase.hregion.majorcompaction

解释：默认值：604800000秒（7天）， Major Compaction的周期，若关闭自动Major Compaction，可将其设为0

**4.优化HStore文件大小**

hbase-site.xml

|  |
| --- |
| 属性：hbase.hregion.max.filesize  解释：默认值10737418240（10GB），如果需要运行HBase的MR任务，可以减小此值，因为一个region对应一个map任务，如果单个region过大，会导致map任务执行时间过长。该值的意思就是，如果HFile的大小达到这个数值，则这个region会被切分为两个Hfile。 |

**5.优化HBase客户端缓存**

hbase-site.xml

|  |
| --- |
| 属性：hbase.client.write.buffer  解释：默认值2097152bytes（2M）用于指定HBase客户端缓存，增大该值可以减少RPC调用次数，但是会消耗更多内存，反之则反之。一般我们需要设定一定的缓存大小，以达到减少RPC次数的目的。 |

**6.指定scan.next扫描HBase所获取的行数**

hbase-site.xml

|  |
| --- |
| 属性：hbase.client.scanner.caching  解释：用于指定scan.next方法获取的默认行数，值越大，消耗内存越大。 |

**7.BlockCache占用RegionServer堆内存的比例**

hbase-site.xml

属性：hfile.block.cache.size

解释：默认0.4，读请求比较多的情况下，可适当调大

**8.MemStore占用RegionServer堆内存的比例**

hbase-site.xml

属性：hbase.regionserver.global.memstore.size

解释：默认0.4，写请求较多的情况下，可适当调大

# 第6章 整合Phoenix

## 6.1 Phoenix简介

### 6.1.1 Phoenix定义

Phoenix是HBase的开源SQL皮肤。可以使用标准JDBC API代替HBase客户端API来创建表，插入数据和查询HBase数据。

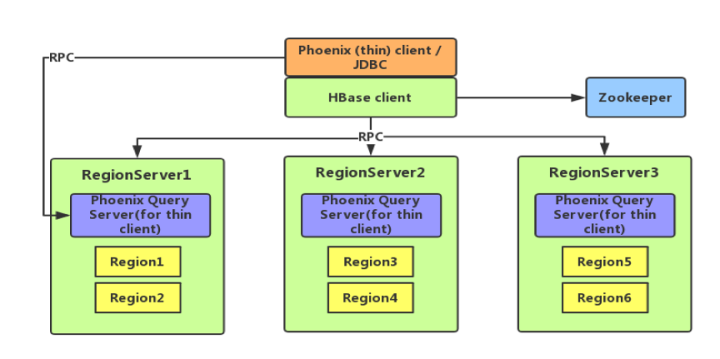
### 6.1.2 Phoenix特点

1）容易集成：如Spark，Hive，Pig，Flume和Map Reduce；

2）操作简单：DML命令以及通过DDL命令创建和操作表和版本化增量更改；

3）支持HBase二级索引创建。

### 6.1.3 Phoenix架构



## 6.2 Phoenix快速入门

### 6.2.1 安装

1.官网地址

<http://phoenix.apache.org/>

2.Phoenix部署

1）上传并解压tar包

[user1@hadoop102 module]$ tar -zxvf apache-phoenix-5.0.0-HBase-2.0-bin.tar.gz -C /opt/module/

[user1@hadoop102 module]$ mv apache-phoenix-5.0.0-HBase-2.0-bin phoenix

2）复制server包并拷贝到各个节点的hbase/lib

[user1@hadoop102 module]$ cd /opt/module/phoenix/

[user1@hadoop102 phoenix]$ cp /opt/module/phoenix/phoenix-5.0.0-HBase-2.0-server.jar /opt/module/hbase/lib/

[user1@hadoop102 phoenix]$ xsync /opt/module/hbase/lib/phoenix-5.0.0-HBase-2.0-server.jar

4）配置环境变量

#phoenix

export PHOENIX\_HOME=/opt/module/phoenix

export PHOENIX\_CLASSPATH=$PHOENIX\_HOME

export PATH=$PATH:$PHOENIX\_HOME/bin

5）重启HBase

[user1@hadoop102 ~]$ stop-hbase.sh

[user1@hadoop102 ~]$ start-hbase.sh

6) 连接Phoenix

[user1@hadoop101 phoenix]$ /opt/module/phoenix/bin/sqlline.py hadoop102,hadoop103,hadoop104:2181

### 6.2.2 Phoenix Shell操作

**1. schema的操作**

1）创建schema

默认情况下，在phoenix中不能直接创建schema。需要将如下的参数添加到Hbase中conf目录下的hbase-site.xml 和 phoenix中bin目录下的 hbase-site.xml中

<property>

<name>phoenix.schema.isNamespaceMappingEnabled</name>

<value>true</value>

</property>

重新启动Hbase和连接phoenix客户端.

[user1@hadoop102 ~]$ stop-hbase.sh

[user1@hadoop102 ~]$ start-hbase.sh

[user1@hadoop102 ~]$ /opt/module/phoenix/bin/sqlline.py hadoop102,hadoop103,hadoop104:2181

创建schema

create schema bigdata;

注意:在phoenix中，schema名，表名，字段名等会自动转换为大写，若要小写，使用双引号，如"student"。

**2.表的操作**

1）显示所有表

!table 或 !tables

2）创建表

直接指定单个列作为RowKey

CREATE TABLE IF NOT EXISTS student(

id VARCHAR primary key,

name VARCHAR,

addr VARCHAR);

指定多个列的联合作为RowKey

CREATE TABLE IF NOT EXISTS us\_population (

State CHAR(2) NOT NULL,

City VARCHAR NOT NULL,

Population BIGINT

CONSTRAINT my\_pk PRIMARY KEY (state, city));

3）插入数据

upsert into student values('1001','zhangsan','beijing');

4）查询记录

select \* from student;

select \* from student where id='1001';

5）删除记录

delete from student where id='1001';

6）删除表

drop table student;

7）退出命令行

!quit

**3.表的映射**

1）表的关系

默认情况下，直接在HBase中创建的表，通过Phoenix是查看不到的。如果要在Phoenix中操作直接在HBase中创建的表，则需要在Phoenix中进行表的映射。映射方式有两种：视图映射和表映射。

2）命令行中创建表test

HBase 中test的表结构如下，两个列族info1、info2。

|  |  |  |
| --- | --- | --- |
| Rowkey | info1 | info2 |
| id | name | address |

启动HBase Shell

[user1@hadoop102 ~]$ /opt/module/hbase/bin/hbase shell

创建HBase表test

hbase(main):001:0> create 'test','info1','info2'

3）视图映射

Phoenix创建的视图是只读的，所以只能用来做查询，无法通过视图对源数据进行修改等操作。在phoenix中创建关联test表的视图

0: jdbc:phoenix:hadoop101,hadoop102,hadoop103> create view "test"(id varchar primary key,"info1"."name" varchar, "info2"."address" varchar);

删除视图

0: jdbc:phoenix:hadoop101,hadoop102,hadoop103> drop view "test";

4）表映射

使用Apache Phoenix创建对HBase的表映射，有两种方法：

（1）HBase中不存在表时，可以直接使用create table指令创建需要的表,系统将会自动在Phoenix和HBase中创建person\_infomation的表，并会根据指令内的参数对表结构进行初始化。

（2）当HBase中已经存在表时，可以以类似创建视图的方式创建关联表，只需要将create view改为create table即可。

0: jdbc:phoenix:hadoop101,hadoop102,hadoop103> create table "test"(id varchar primary key,"info1"."name" varchar, "info2"."address" varchar) column\_encoded\_bytes=0;

**4.表映射中数值类型的问题**

Hbase中存储数值类型的值(如int,long等)会按照正常数字的补码进行存储. 而phoenix对数字的存储做了特殊的处理. phoenix 为了解决遇到正负数同时存在时，导致负数排到了正数的后面（负数高位为1，正数高位为0，字典序0 < 1）的问题。 phoenix在存储数字时会对高位进行转换.原来为1,转换为0， 原来为0，转换为1.

因此，如果hbase表中的数据的写是由phoenix写入的,不会出现问题，因为对数字的编解码都是phoenix来负责。如果hbase表中的数据不是由phoenix写入的，数字的编码由hbase负责. 而phoenix读数据时要对数字进行解码。 因为编解码方式不一致。导致数字出错.

1. 在hbase中创建表，并插入数值类型的数据

hbase(main):001:0> create 'person','info'

hbase(main):001:0> put 'person','1001', 'info:salary',Bytes.toBytes(123456)

注意: 如果要插入数字类型，需要通过Bytes.toBytes(123456)来实现。

2）在phoenix中创建映射表并查询数据

create table "person"(id varchar primary key,"info"."salary" integer )

column\_encoded\_bytes=0;

select \* from "person"

会发现数字显示有问题

1. 解决办法:

在phoenix中创建表时使用无符号的数值类型. unsigned\_long

create table "person"(id varchar primary key,"info"."salary" unsigned\_long )

column\_encoded\_bytes=0;

### 6.2.3 Phoenix JDBC操作

1.Thin Client

1）启动query server

[user1@hadoop102 ~]$ queryserver.py start

2）创建项目并导入依赖

<dependencies>

<dependency>

<groupId>org.apache.phoenix</groupId>

<artifactId>phoenix-queryserver-client</artifactId>

<version>5.0.0-HBase-2.0</version>

</dependency>

</dependencies>

3）编写代码

package com.user1;

import java.sql.\*;

import org.apache.phoenix.queryserver.client.ThinClientUtil;

public class PhoenixTest {

public static void main(String[] args) throws SQLException {

String connectionUrl = ThinClientUtil.getConnectionUrl("hadoop102", 8765);

Connection connection = DriverManager.getConnection(connectionUrl);

PreparedStatement preparedStatement = connection.prepareStatement("select \* from student");

ResultSet resultSet = preparedStatement.executeQuery();

while (resultSet.next()) {

System.out.println(resultSet.getString(1) + "\t" + resultSet.getString(2));

}

//关闭

connection.close();

}

}

2.Thick Client

1）在pom中加入依赖

<dependencies>

<dependency>

<groupId>org.apache.phoenix</groupId>

<artifactId>phoenix-core</artifactId>

<version>5.0.0-HBase-2.0</version>

<exclusions>

<exclusion>

<groupId>org.glassfish</groupId>

<artifactId>javax.el</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.glassfish</groupId>

<artifactId>javax.el</artifactId>

<version>3.0.1-b06</version>

</dependency>

</dependencies>

2）编写代码

package com.user1.phoenix.thin;

import java.sql.\*;

import java.util.Properties;

public class TestThick {

public static void main(String[] args) throws SQLException {

String url = "jdbc:phoenix:hadoop102,hadoop103,hadoop104:2181";

Properties props = new Properties();

props.put("phoenix.schema.isNamespaceMappingEnabled","true");

Connection connection = DriverManager.getConnection(url,props);

PreparedStatement ps = connection.prepareStatement("select \* from \"test\"");

ResultSet rs = ps.executeQuery();

while(rs.next()){

System.out.println(rs.getString(1)+":" +rs.getString(2));

}

}

}

## 6.3 Phoenix二级索引

### 6.3.1 二级索引配置文件

1.添加如下配置到HBase的HRegionserver节点的hbase-site.xml

<!-- phoenix regionserver 配置参数-->

<property>

<name>hbase.regionserver.wal.codec</name>

<value>org.apache.hadoop.hbase.regionserver.wal.IndexedWALEditCodec</value>

</property>

<property>

<name>hbase.region.server.rpc.scheduler.factory.class</name>

<value>org.apache.hadoop.hbase.ipc.PhoenixRpcSchedulerFactory</value>

<description>Factory to create the Phoenix RPC Scheduler that uses separate queues for index and metadata updates</description>

</property>

<property>

<name>hbase.rpc.controllerfactory.class</name>

<value>org.apache.hadoop.hbase.ipc.controller.ServerRpcControllerFactory</value>

<description>Factory to create the Phoenix RPC Scheduler that uses separate queues for index and metadata updates</description>

</property>

### 6.3.2 全局二级索引

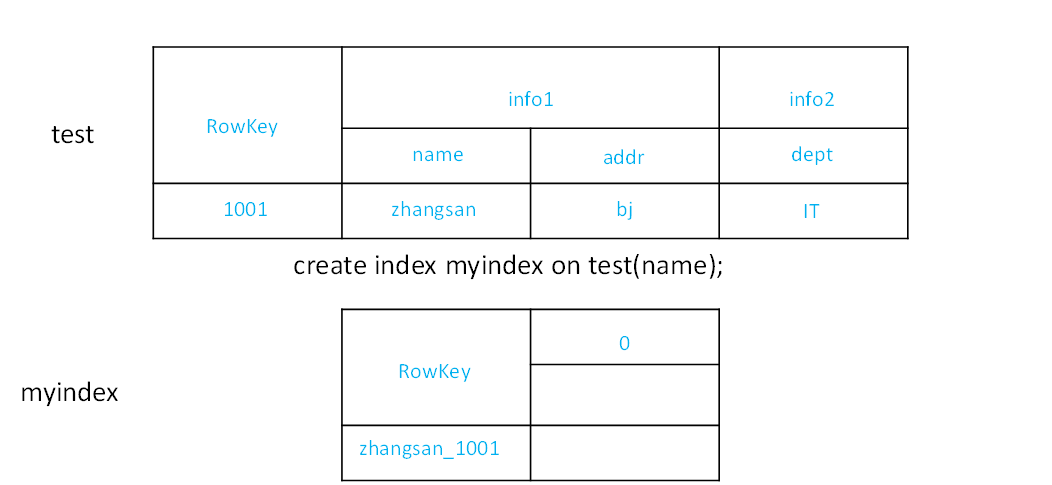
Global Index是默认的索引格式，创建全局索引时，会在HBase中建立一张新表。也就是说索引数据和数据表是存放在不同的表中的，因此全局索引适用于多读少写的业务场景。

写数据的时候会消耗大量开销，因为索引表也要更新，而索引表是分布在不同的数据节点上的，跨节点的数据传输带来了较大的性能消耗。

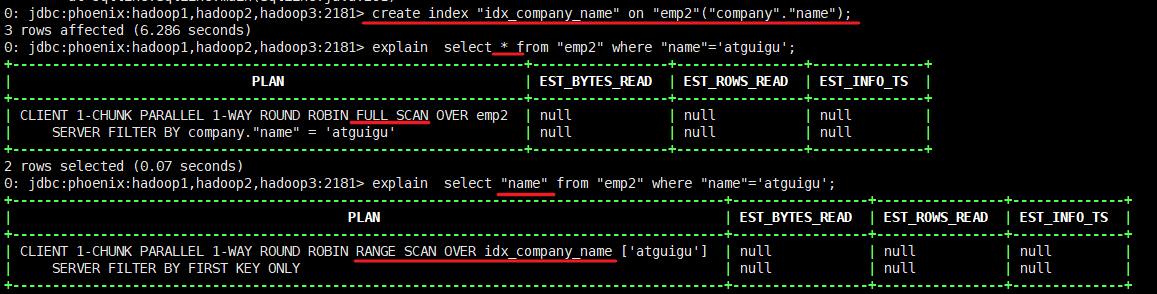
在读数据的时候Phoenix会选择索引表来降低查询消耗的时间。

**1.创建单个字段的全局索引**

CREATE INDEX my\_index ON my\_table (my\_col);

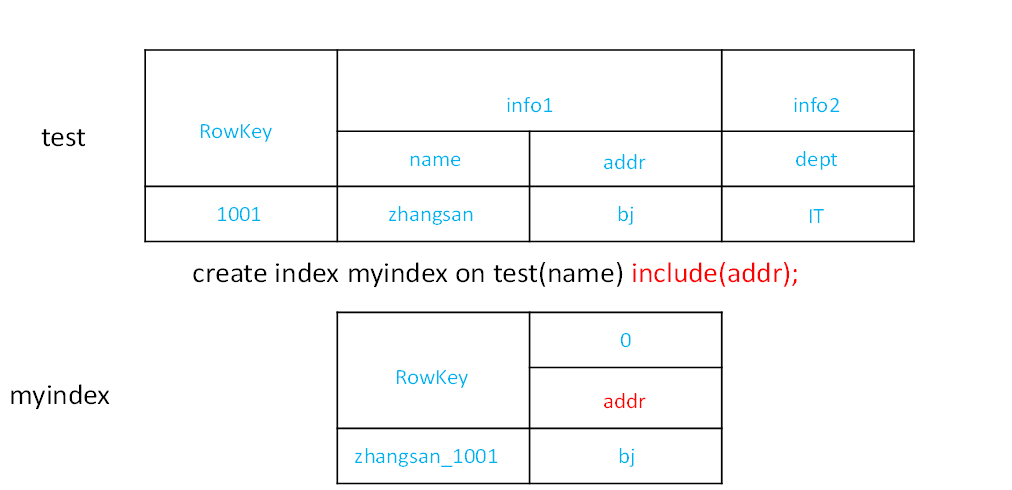


如果想查询的字段不是索引字段的话索引表不会被使用，也就是说不会带来查询速度的提升。



**2.创建携带其他字段的全局索引**

CREATE INDEX my\_index ON my\_table (v1) INCLUDE (v2);

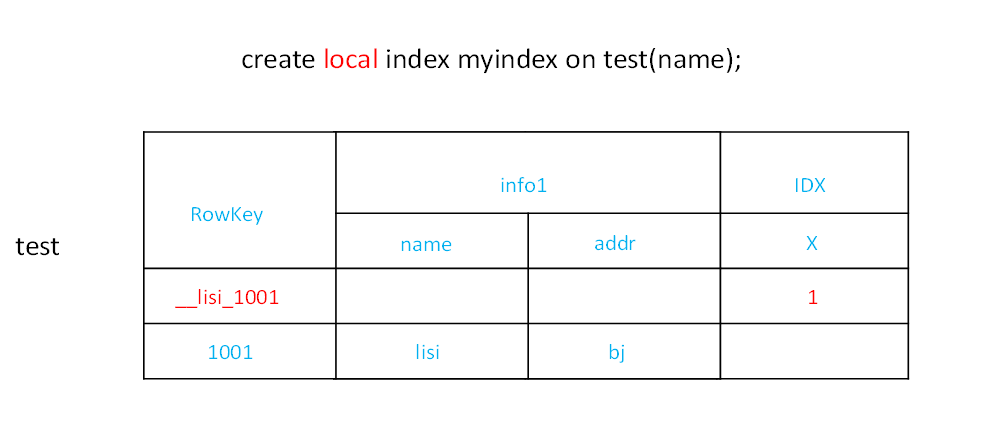


### 6.3.3 本地二级索引

Local Index适用于写操作频繁的场景。

索引数据和数据表的数据是存放在同一张表中（且是同一个Region），避免了在写操作的时候往不同服务器的索引表中写索引带来的额外开销。

CREATE LOCAL INDEX my\_index ON my\_table (my\_column);



# 第7章 与Hive的集成

## 7.1 HBase与Hive的对比

1.Hive

(1) 数据仓库

Hive的本质其实就相当于将HDFS中已经存储的文件在Mysql中做了一个双射关系，以方便使用HQL去管理查询。

(2) 用于数据分析、清洗

Hive适用于离线的数据分析和清洗，延迟较高。

(3) 基于HDFS、MapReduce

Hive存储的数据依旧在DataNode上，编写的HQL语句终将是转换为MapReduce代码执行。

2．HBase

(1) 数据库

是一种面向列族存储的非关系型数据库。

(2) 用于存储结构化和非结构化的数据

适用于单表非关系型数据的存储，不适合做关联查询，类似JOIN等操作。

(3) 基于HDFS

数据持久化存储的体现形式是HFile，存放于DataNode中，被ResionServer以region的形式进行管理。

(4) 延迟较低，接入在线业务使用

面对大量的企业数据，HBase可以直线单表大量数据的存储，同时提供了高效的数据访问速度。

## 7.2 HBase与Hive集成使用

在hive-site.xml中添加zookeeper的属性，如下：

|  |
| --- |
| <property>  <name>hive.zookeeper.quorum</name>  <value>hadoop102,hadoop103,hadoop104</value>  </property>  <property>  <name>hive.zookeeper.client.port</name>  <value>2181</value>  </property> |

**案例一**

**目标：**建立Hive表，关联HBase表，插入数据到Hive表的同时能够影响HBase表。

分步实现：

1.在Hive中创建表同时关联HBase

|  |
| --- |
| CREATE TABLE hive\_hbase\_emp\_table(  empno int,  ename string,  job string,  mgr int,  hiredate string,  sal double,  comm double,  deptno int)  STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'  WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:comm,info:deptno")  TBLPROPERTIES ("hbase.table.name" = "hbase\_emp\_table"); |

提示：完成之后，可以分别进入Hive和HBase查看，都生成了对应的表

2.在Hive中创建临时中间表，用于load文件中的数据

提示：不能将数据直接load进Hive所关联HBase的那张表中

|  |
| --- |
| CREATE TABLE emp(  empno int,  ename string,  job string,  mgr int,  hiredate string,  sal double,  comm double,  deptno int)  row format delimited fields terminated by '\t'; |

3.向Hive中间表中load数据

|  |
| --- |
| hive> load data local inpath '/home/admin/softwares/data/emp.txt' into table emp; |

4.通过insert命令将中间表中的数据导入到Hive关联Hbase的那张表中

|  |
| --- |
| hive> insert into table hive\_hbase\_emp\_table select \* from emp; |

5.查看Hive以及关联的HBase表中是否已经成功的同步插入了数据

Hive：

|  |
| --- |
| hive> select \* from hive\_hbase\_emp\_table; |

HBase：

|  |
| --- |
| Hbase> scan 'hbase\_emp\_table' |

**案例二**

**目标：**在HBase中已经存储了某一张表hbase\_emp\_table，然后在Hive中创建一个外部表来关联HBase中的hbase\_emp\_table这张表，使之可以借助Hive来分析HBase这张表中的数据。

**注：**该案例2紧跟案例1的脚步，所以完成此案例前，请先完成案例1。

分步实现：

**1.在Hive中创建外部表**

|  |
| --- |
| CREATE EXTERNAL TABLE relevance\_hbase\_emp(  empno int,  ename string,  job string,  mgr int,  hiredate string,  sal double,  comm double,  deptno int)  STORED BY  'org.apache.hadoop.hive.hbase.HBaseStorageHandler'  WITH SERDEPROPERTIES ("hbase.columns.mapping" =  ":key,info:ename,info:job,info:mgr,info:hiredate,info:sal,info:comm,info:deptno")  TBLPROPERTIES ("hbase.table.name" = "hbase\_emp\_table"); |

**2.关联后就可以使用Hive函数进行一些分析操作了**

|  |
| --- |
| hive (default)> select \* from relevance\_hbase\_emp; |