尚硅谷大数据技术之 HBase

（作者：尚硅谷大数据研发部）

版本：V1.3

第 1 章 HBase 简介

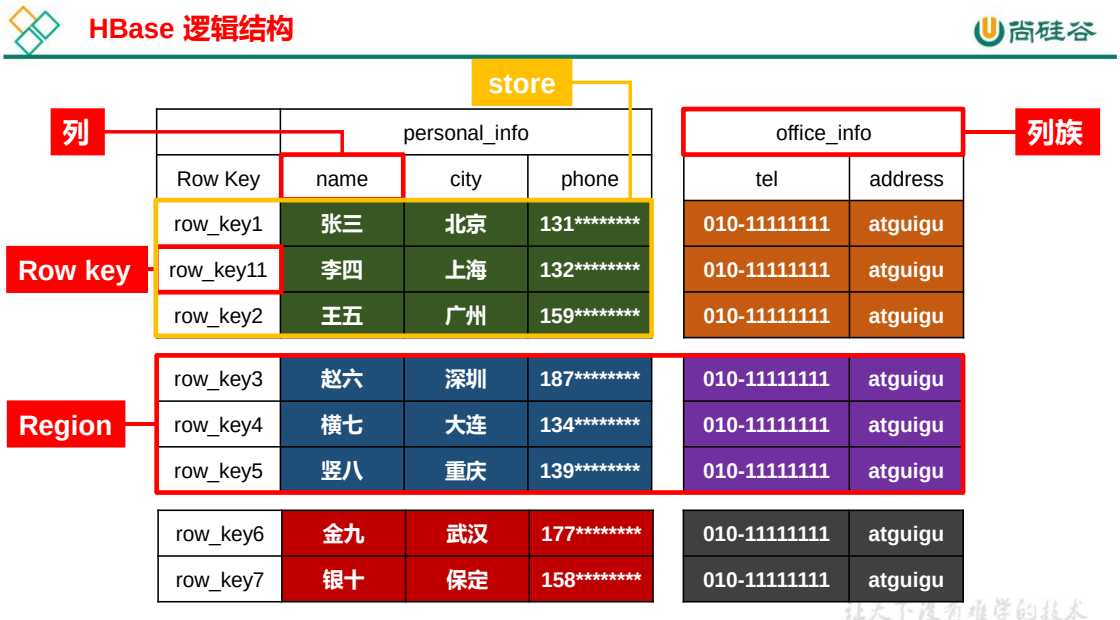
1.1 HBase 定义

HBase 是一种分布式、可扩展、支持海量数据存储的 NoSQL 数据库。

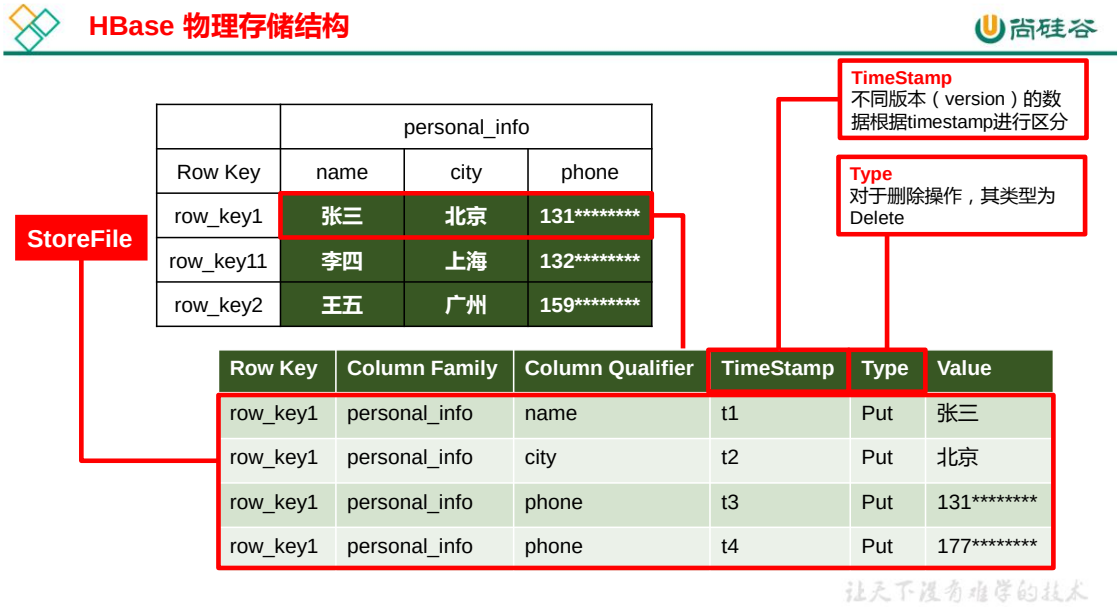
1.2 HBase 数据模型

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 HBase 逻辑结构



1.2.2 HBase 物理存储结构



1.2.3 数据模型

1. Name Space

命名空间，类似于关系型数据库的 DatabBase 概念，每个命名空间下有多个表。HBase 有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. Region

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。

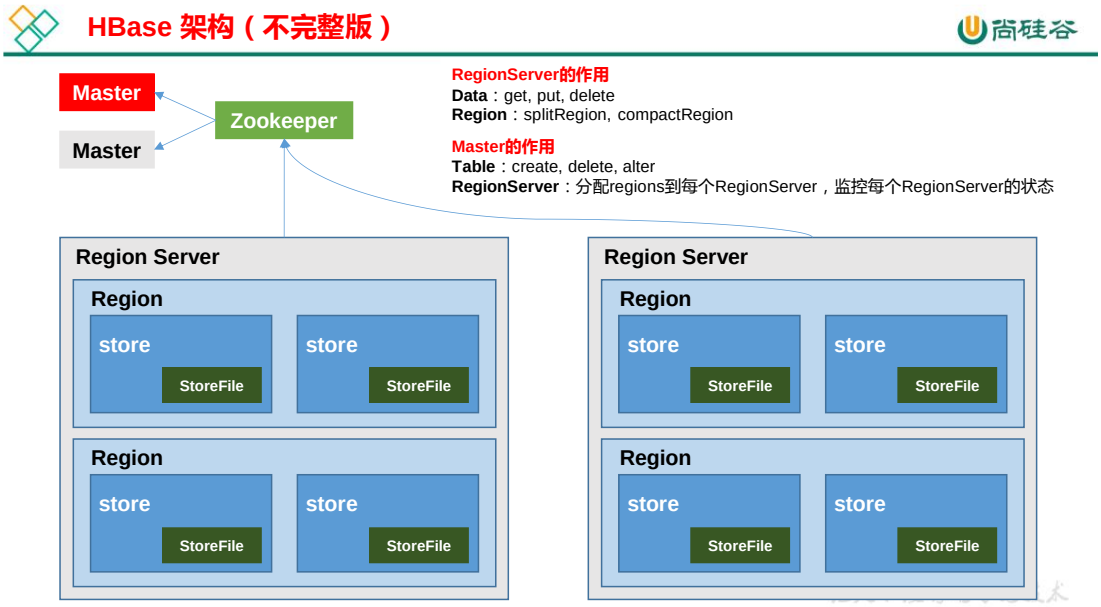
5. Time Stamp

用于标识数据的不同版本（version），每条数据写入时，如果不指定时间戳，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 中的数据是没有类型的，全部是字节数组形式存贮。

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 快速入门

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：

```
[atguigu@hadoop102 zookeeper-3.5.7]$ bin/zkServer.sh start
[atguigu@hadoop103 zookeeper-3.5.7]$ bin/zkServer.sh start
[atguigu@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：

```
[atguigu@hadoop102 hadoop-3.1.3]$ sbin/start-dfs.sh
[atguigu@hadoop103 hadoop-3.1.3]$ sbin/start-yarn.sh
```

2.1.3 HBase 的解压

解压 Hbase 到指定目录：

```
[atguigu@hadoop102 software]$ tar -zxvf hbase-2.2.4-bin.tar.gz -C
/opt/module
[atguigu@hadoop102 software]$ mv /opt/module/hbase-2.2.4 /opt/module/hbase
```

配置环境变量

```
[atguigu@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>

    <property>
        <name>hbase.unsafe.stream.capability.enforce</name>
        <value>false</value>
    </property>

    <property>
        <name>hbase.wal.provider</name>
        <value>filesystem</value>
    </property>
</configuration>
```

3. regionservers：

```
hadoop102
hadoop103
hadoop104
```

2.1.5 HBase 远程发送到其他集群

```
[atguigu@hadoop102 module]$ xsync hbase/
```

2.1.6 HBase 服务的启动

1. 启动方式

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

请参看帮助文档：《尚硅谷大数据技术之 Hadoop 入门》

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. **启动方式 2**

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

对应的停止服务：

```
[atguigu@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

2.1.8 高可用(可选)

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）**

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

2. **在 conf 目录下创建 backup-masters 文件**

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

3. **在 backup-masters 文件中配置高可用 HMaster 节点**

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

4. **将整个 conf 目录 scp 到其他节点**

```
[atguigu@hadoop102 hbase]$ scp -r conf/ hadoop103:/opt/module/hbase/
[atguigu@hadoop102 hbase]$ scp -r conf/ hadoop104:/opt/module/hbase/
```

5. **打开页面测试查看**

http://hadooo102:16010

2.2 HBase Shell 操作

2.2.1 基本操作

1. 进入 HBase 客户端命令行

```
[atguigu@hadoop102 hbase]$ bin/hbase shell
```

2. 查看帮助命令

```
hbase(main):001:0> help
```

3. 查看当前数据库中有哪些表

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

2.2.2 表的操作

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 API

在 IDEA 中创建 hbase-demo 模块

3.1 DDL

创建 HBase_DDL 类

3.1.1 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;
    }

}
```

3.1.2 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();
    }

}
```

3.1.3 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();
    }

}
```

3.1.4 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();

    }
}
```

3.2 DML

创建类 HBase_DML

3.2.1 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();
    }

}
```

3.2.2 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();

    }

}
```

3.2.3 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("CF:" +
Bytes.toString(cell.getFamilyArray()) +
                        ",CN:" + Bytes.toString(cell.getQualifierArray()) +
                        ",Value:" + Bytes.toString(cell.getValueArray()));
            }
        }

        //7.关闭资源
        table.close();
        connection.close();

    }
```

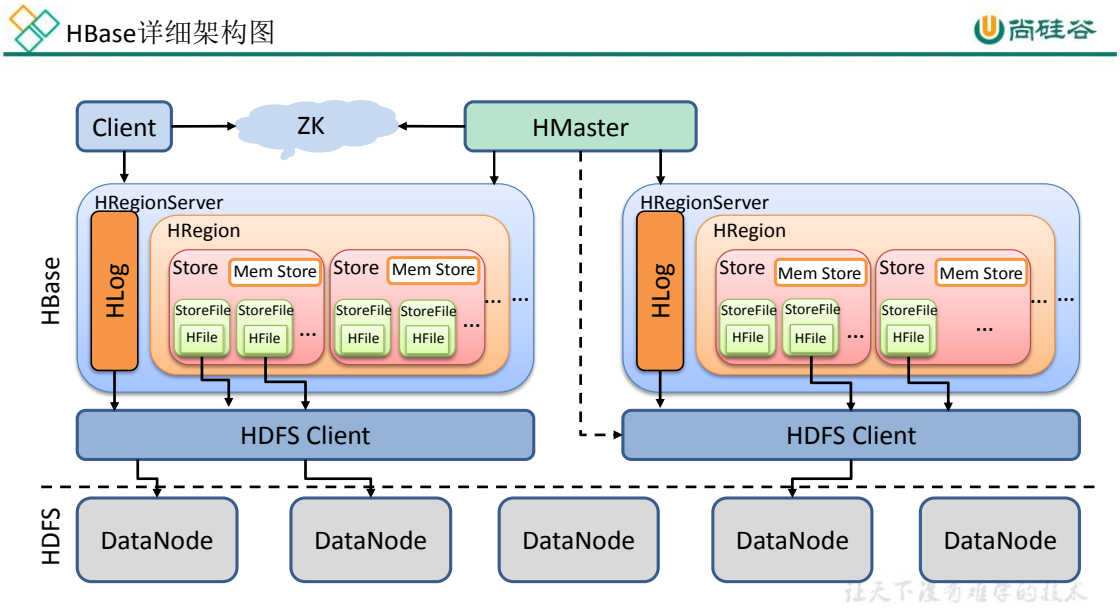
```
}
```

3.2.4 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 4 章 HBase 进阶

4.1 架构原理



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。

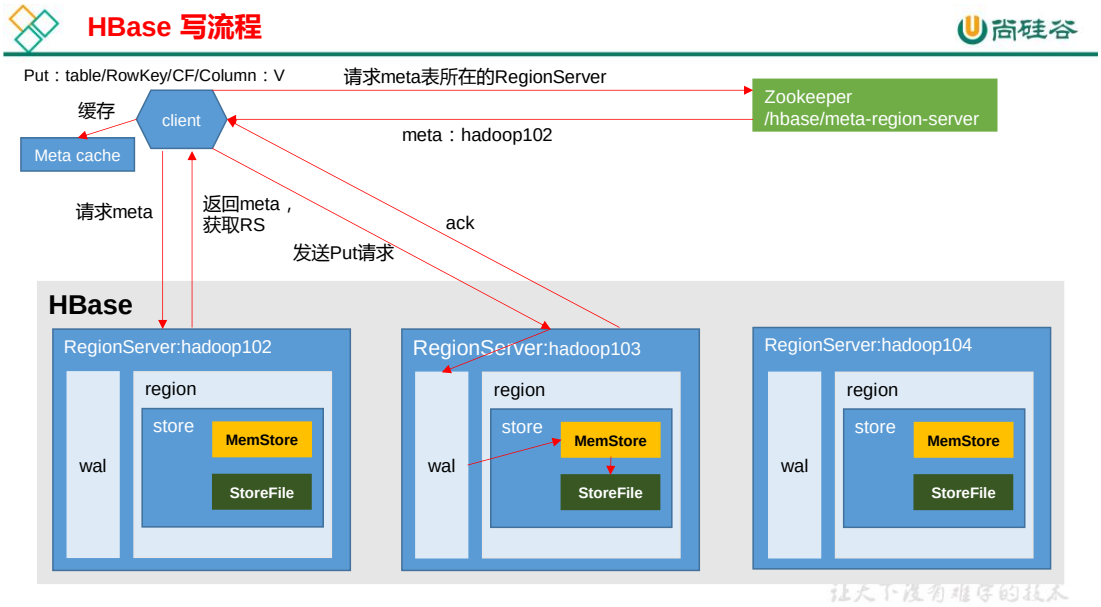
2. MemStore

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。

4.2 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；

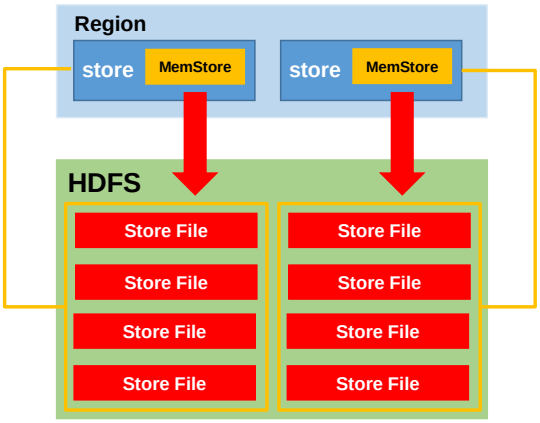
4）将数据顺序写入（追加）到 WAL；

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。

4.3 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

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

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

region server 会把它的所有 region 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 hbase.regionserver.global.memstore.size.lower.limit 以下。

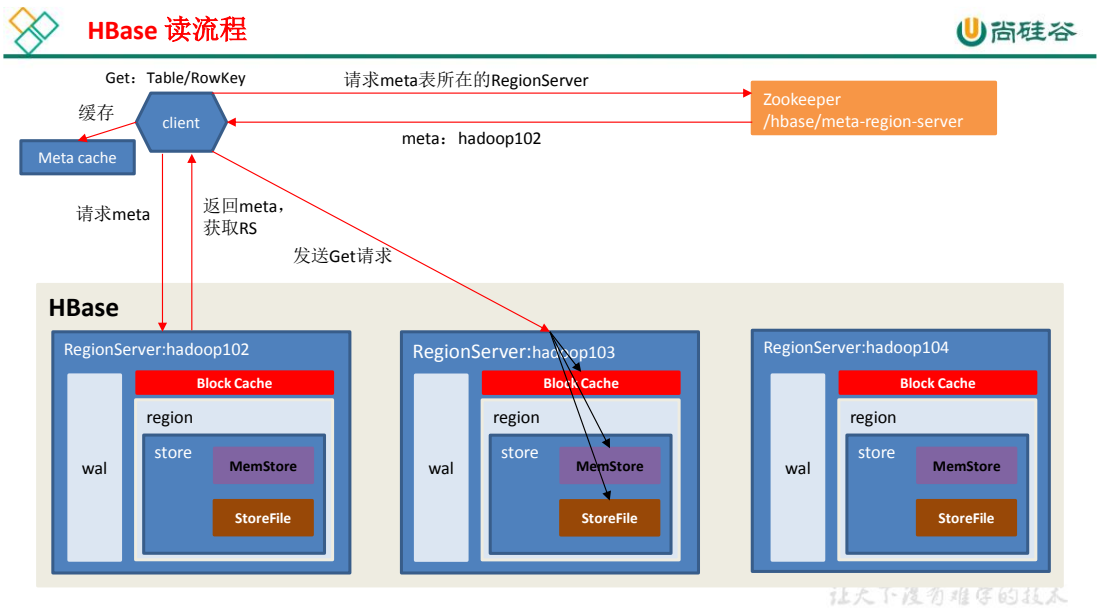
当 region server 中 memstore 的总大小达到

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.log** 以下（该属性名已经废弃，现无需手动设置，最大值为 32）。

4.4 读流程



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 Block Cache（读缓存），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）。

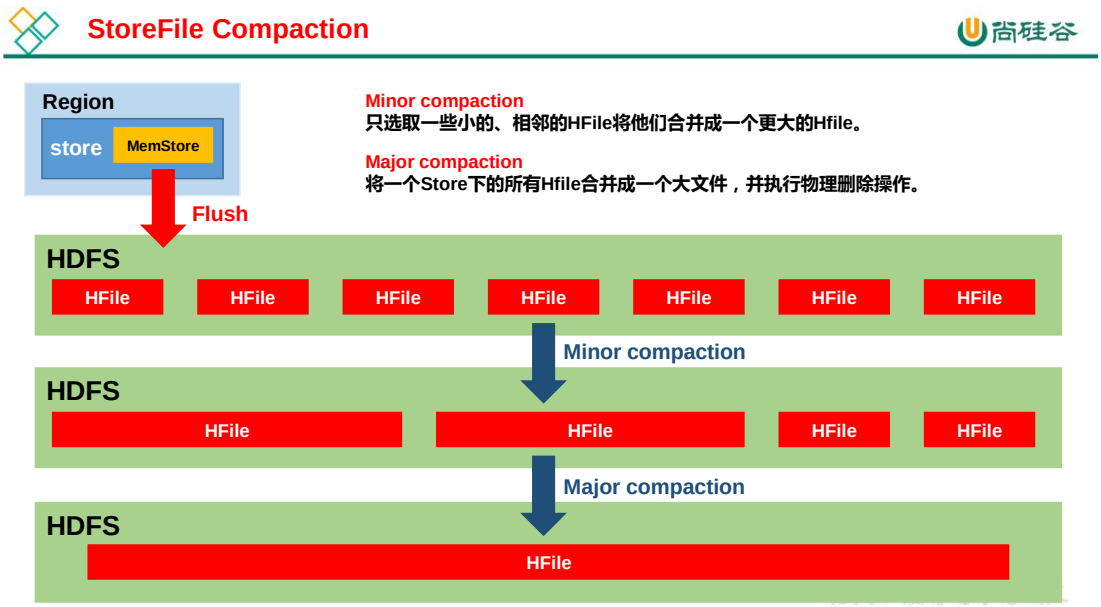
5）将从文件中查询到的数据块（Block，HFile 数据存储单元，默认大小为 64KB）缓存到 Block Cache。

6）将合并后的最终结果返回给客户端。

4.5 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，但不会清理过期和删除的数据。Major Compaction 会将一个 Store 下的所有的 HFile 合并成一个大 HFile，并且会清理掉过期和删除的数据。



4.6 Region Split

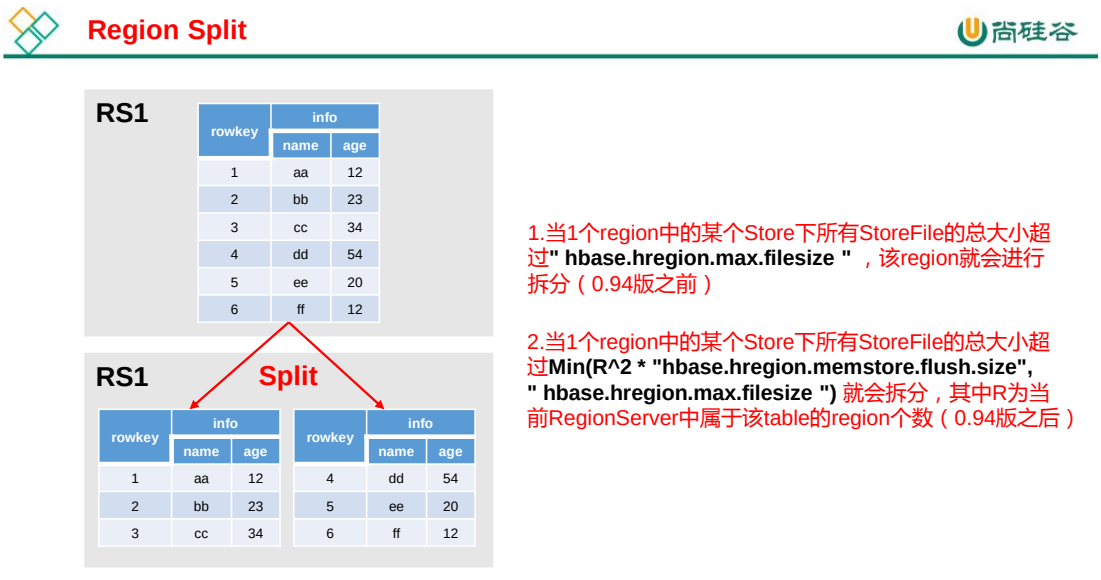
默认情况下，每个 Table 起初只有一个 Region，随着数据的不断写入，Region 会自动进行拆分。刚拆分时，两个子 Region 都位于当前的 Region Server，但处于负载均衡的考虑，HMaster 有可能会将某个 Region 转移给其他的 Region Server。

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(R^3 * 2 *"hbase.hregion.memstore.flush.size",hbase.hregion.max.filesize")，该 Region 就会进行拆分，其中 R 为当前 Region Server 中属于该 Table 的个数（0.94 版本之后）。

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 5 章 HBase 优化

5.1 预分区

每一个 region 维护着 StartRow 与 EndRow，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','partition1',SPLITS =>
['1000','2000','3000','4000']
```

2. 生成 16 进制序列预分区

```
create 'staff2','info','partition2',{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','partition3',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，一般会分配整个可用内存的 70%给 HBase 的 Java 堆。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 16~48G 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. 允许在 HDFS 的文件中追加内容

hdfs-site.xml、hbase-site.xml

```
属性：dfs.support.append
解释：开启 HDFS 追加同步，可以优秀的配合 HBase 的数据同步和持久化。默认值为 true。
```

2. 优化 DataNode 允许的最大文件打开数

hdfs-site.xml

```
属性：dfs.datanode.max.transfer.threads
解释：HBase 一般都会同一时间操作大量的文件，根据集群的数量和规模以及数据动作，设置为 4096 或者更高。默认值：4096
```

3. 优化延迟高的数据操作的等待时间

hdfs-site.xml

```
属性：dfs.image.transfer.timeout
解释：如果对于某一次数据操作来讲，延迟非常高，socket 需要等待更长的时间，建议把该值设置为更
```

大的值（默认 60000 毫秒），以确保 socket 不会被 timeout 掉。

4. 优化数据的写入效率

mapred-site.xml

属性：
mapreduce.map.output.compress
mapreduce.map.output.compress.codec
解释：开启这两个数据可以大大提高文件的写入效率，减少写入时间。第一个属性值修改为 true，第二个属性值修改为：org.apache.hadoop.io.compress.GzipCodec 或者其他压缩方式。

5. 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。

6. 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。

7. 优化 HBase 客户端缓存

hbase-site.xml

属性：hbase.client.write.buffer
解释：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。

8. 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。

9. flush、compact、split 机制

当 MemStore 达到阈值，将 Memstore 中的数据 Flush 进 Storefile；compact 机制则是把 flush 出来的小文件合并成大的 Storefile 文件。split 则是当 Region 达到阈值，会把过大的 Region 一分为二。

涉及属性：

即：128M 就是 Memstore 的默认阈值

```
hbase.hregion.memstore.flush.size: 134217728
```

即：这个参数的作用是当单个 HRegion 内所有的 Memstore 大小总和超过指定值时，flush 该 HRegion 的所有 memstore。RegionServer 的 flush 是通过将请求添加一个队列，

模拟生产消费模型来异步处理的。那这里就有一个问题，当队列来不及消费，产生大量积压请求时，可能会导致内存陡增，最坏的情况是触发 OOM。

```
hbase.regionserver.global.memstore.upperLimit: 0.4
hbase.regionserver.global.memstore.lowerLimit: 0.38
```

即：当 MemStore 使用内存总量达到 hbase.regionserver.global.memstore.upperLimit 指定值时，将会有多个 MemStores flush 到文件中，MemStore flush 顺序是按照大小降序执行的，直到刷新到 MemStore 使用内存略小于 lowerLimit

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。

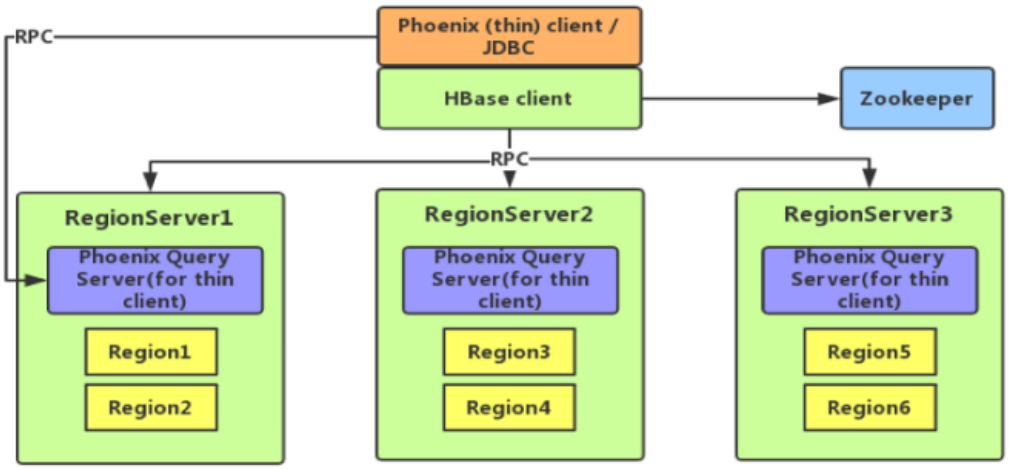
6.1.2 Phoenix 特点

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

```
[atguigu@hadoop101 module]$ tar -zxvf /opt/software/phoenix-5.1.0-SNAPSHOT-hbase-2.2.tar.gz -C /opt/module

[atguigu@hadoop101 module]$ mv phoenix-5.1.0-SNAPSHOT-hbase-2.2 phoenix
```

2）复制 server 包并拷贝到各个节点的 hbase/lib

```
[atguigu@hadoop102 module]$ cd /opt/module/phoenix/

[atguigu@hadoop102 phoenix]$ cp phoenix-5.1.0-SNAPSHOT-hbase-2.2-server.jar /opt/module/hbase/lib/
[atguigu@hadoop102 phoenix]$ scp phoenix-5.1.0-SNAPSHOT-hbase-2.2-server.jar hadoop103:/opt/module/hbase/lib/
[atguigu@hadoop102 phoenix]$ scp phoenix-5.1.0-SNAPSHOT-hbase-2.2-server.jar hadoop104:/opt/module/hbase/lib/
```

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）连接 Phoenix

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

6.2.2 Phoenix Shell 操作

1. 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);
```

在 phoenix 中，表名等会自动转换为大写，若要小写，使用双引号，如"us_population"。

```
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');
```

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
```

2. 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

```
[atguigu@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;
```

6.2.3 Phoenix JDBC 操作

1. 创建项目并导入依赖

```
    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.apache.phoenix/phoenix-
queryserver-client -->
        <dependency>
            <groupId>org.apache.phoenix</groupId>
            <artifactId>phoenix-queryserver-client</artifactId>
            <version>5.0.0-HBase-2.0</version>
        </dependency>
    </dependencies>
```

2. 编写代码

```
package com.atguigu;

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);
        System.out.println(connectionUrl);
        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();

    }
}
```

6.3 Phoenix 二级索引

6.3.1 HBase 协处理器（扩展）

1. 案例需求

编写协处理器，实现在往 A 表插入数据的同时让 HBase 自身（协处理器）向 B 表中插入一条数据。

2. 实现步骤

1）创建一个 maven 项目，并引入以下依赖。

```
<dependencies>
    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-client</artifactId>
        <version>2.2.4</version>
</dependency>

    <dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-server</artifactId>
```

```
        <version>2.2.4</version>
    </dependency>
</dependencies>
```

2）定义 FruitTableCoprocessor 类并继承 BaseRegionObserver 类

```
package com.atguigu;

import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.coprocessor.BaseRegionObserver;
import org.apache.hadoop.hbase.coprocessor.ObserverContext;
import org.apache.hadoop.hbase.coprocessor.RegionCoprocessorEnvironment;
import org.apache.hadoop.hbase.regionserver.wal.WALEdit;

import java.io.IOException;

public class FruitTableCoprocessor extends BaseRegionObserver {

    @Override
    public void postPut(ObserverContext<RegionCoprocessorEnvironment> e, Put
put, WALEdit edit, Durability durability) throws IOException {

        //获取连接
        Connection connection =
ConnectionFactory.createConnection(HBaseConfiguration.create());

        //获取表对象
        Table table = connection.getTable(TableName.valueOf("fruit"));

        //插入数据
        table.put(put);

        //关闭资源
        table.close();
        connection.close();
    }
}
```

6.3.2 二级索引配置文件

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>
```

6.3.3 全局二级索引

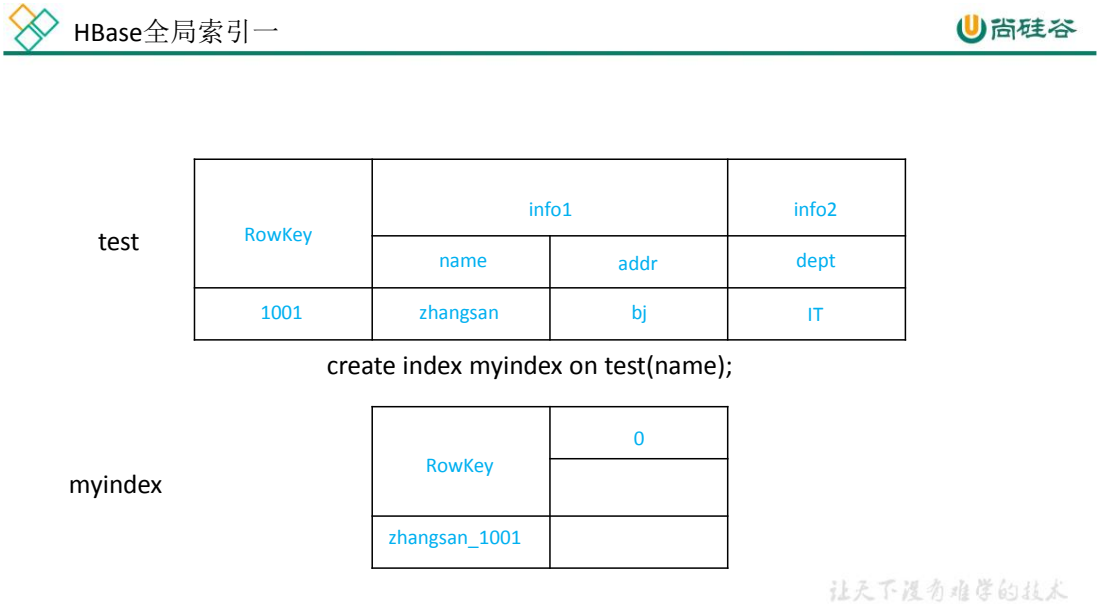
Global Index 是默认的索引格式，创建全局索引时，会在 HBase 中建立一张新表。也就是说索引数据和数据表是存放在不同的表中的，因此全局索引适用于多读少写的业务场景。

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。

```
0: jdbc:phoenix:hadoop1,hadoop2,hadoop3:2181> create index "idx_company_name" on "emp2"("company"."name");
3 rows affected (6.286 seconds)
0: jdbc:phoenix:hadoop1,hadoop2,hadoop3:2181> explain  select * from "emp2" where "name"='atguigu';
+-----------------------------------------------------------+------------------+------------------+----------------+
|                           PLAN                            | EST_BYTES_READ   | EST_ROWS_READ    | EST_INFO_TS    |
+-----------------------------------------------------------+------------------+------------------+----------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN FULL SCAN OVER emp2  | null             | null             | null           |
|     SERVER FILTER BY company."name" = 'atguigu'           | null             | null             | null           |
+-----------------------------------------------------------+------------------+------------------+----------------+
2 rows selected (0.07 seconds)
0: jdbc:phoenix:hadoop1,hadoop2,hadoop3:2181> explain  select "name" from "emp2" where "name"='atguigu';
+-----------------------------------------------------------------------+------------------+------------------+----------------+
|                                 PLAN                                  | EST_BYTES_READ   | EST_ROWS_READ    | EST_INFO_TS    |
+-----------------------------------------------------------------------+------------------+------------------+----------------+
| CLIENT 1-CHUNK PARALLEL 1-WAY ROUND ROBIN RANGE SCAN OVER idx_company_name ['atguigu']  | null             | null             | null           |
|     SERVER FILTER BY FIRST KEY ONLY                                   | null             | null             | null           |
+-----------------------------------------------------------------------+------------------+------------------+----------------+
```

2）创建携带其他字段的全局索引

```
CREATE INDEX my_index ON my_table (v1) INCLUDE (v2);
```

HBase全局索引二

test

RowKey	info1		info2
	name	addr	dept
1001	zhangsan	bj	IT

create index myindex on test(name) include(addr);

myindex

RowKey	0
	addr
zhangsan_1001	bj

6.3.4 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。查询的字段不是索引字段索引表也会被使用，这会带来查询速度的提升。

```
CREATE LOCAL INDEX my_index ON my_table (my_column);
```

create local index myindex on test(name);

test

RowKey	info1		IDX
	name	addr	X
__lisi_1001			1
1001	lisi	bj	

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 中修改 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>
```

1. 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:dep
tno")
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'
```

2. 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:dep
tno")
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;
```