HBase：Hadoop Database是高可靠性、高性能、面向列、可伸缩、实时读写的分布式数据库，使用Hadoop HDFS作为文件存储系统，Hadoop MapReduce处理HBase中海量数据，ZooKeeper作为分布式协同服务，存储非结构化和结构化的松散数据，Columnar Store列式存储的数据库，可以在大数据里进行快速查询，列式数据库，可集群化，可以使用shell，web，api等多种方式访问，适合搞读写（insert）的场景，HQL查询语言，NoSQL的典型代表产品

HBase以表的形式存放数据，表由行与列组成，每个列属于某个列族，由行和列确定的存储单元称为元素，每个元素保存了同一份数据的多个版本，由时间戳来标识区分

行键（Row Key）：行键是数据行在表里的唯一标识，并作为检索记录的主键，访问表里的行只有三种方式：通过单一行键访问、给行键的范围访问、全表扫描，行键可以是最大长度不超过64KB的任意字节数据，并按照字典序（升序）存储

列族与列：列表示为<列族>:<限定符>，HBase创建表时必须指定至少一个列族（由于HBase的I/O问题，列族个数一般在1到3个之间），每个列族可以有多个列成员（限定符），权限控制、存储以及调优都是在列族层面进行，HBase把在磁盘上按照列族存储数据，将同一列族的数据存放在同一目录下，这种列式数据库的设计非常适合数据分析的情形，列族里的元素最好具有相同的读写方式（例如等长的字符串），以提高性能

时间戳（Time Stamp）：64位整型精确到毫秒的系统时间，HBase每个存储单元对同一份数据通过时间戳有多个版本，数据按照时间戳倒叙排序，对数据库的删除和修改实质是插入一个新的时间戳记录，对应每次数据操作的时间，可由系统自动生成，也可以由用户显示的赋值，HBase支持两种数据版本回收方式：通过设定每个数据单元只存储指定个数的最新版本，保存指定时间长度的版本

元素Cell单元格由行键，列族:限定符，时间戳唯一决定，以字节码（未解析的字节数组）形式存放，没有类型之分

表在行的方向上，按照行键范围划分成若干的Region，每个表最初只有一个Region，一个Region保存一个表里某段连续的数据，当记录数增加到超过某个阈值时会分裂成两个Region，Region数量达到一定阈值会被分配到其他的HRegionServer上，物理上所有数据存放在HDFS，由Region服务器提供管理，一台物理节点只能运行一个HRegionServer，一个HRegionServer可以管理多个Region实例，一个Region实例包括一个Hlog日志（包含数据的日志文件）和多个存放数据的Store，Store包含一个位于内存汇总的内存缓冲区MemStore和多个位于磁盘上的StoreFile（HDFS中的名称为HFile），Hmaster作为总控节点，ZooKeeper负责调度

Client负责HBase接口的访问和维护cache加快对HBase的访问

ZooKeeper负责集群任何时候有一个活跃的Master，存储所有Region的寻址入口和HBase的schema以及table元数据，实时监控RegionServer的上线和下线信息通知Master

Master负责管理RegionServer的负载均衡分配和管理Region和table的增删改操作

RegionServer负责维护和处理Region的I/O请求，切分超过阈值的Region

HLog用于灾难恢复，预写式日志，记录所有更新操作，操作先记录进日志，数据才会写入

HBase中有两张特殊的Table，-ROOT-和.META.，.META.记录了用户表的Region信息，.META.可以有多个Region，-ROOT-记录了.META.表的Region信息，-ROOT-只有一个Region，ZooKeeper记录了-ROOT-表的location位置

HRegion是HBase中分布式存储和负载均衡的最小单元，一个Region由多个Store组成，每个Store包含一个列族CF的所有数据，Store包括位于内存的Memstore和位于磁盘的Storefile，写操作先写入Memstore，当Memstore中的数量达到某个阈值，HRegionServer启动Flashcache进程写入storefile，每次写入形成单独一个storefile，当storefile文件的数量增长到一定阈值后，系统会进行合并，在合并过程中会进行版本合并和删除工作，形成更大的storefile，当storefile大小超过一定阈值后，会把当前的Region分割为两个，并有HMaster分配到相应的Region服务器，实现负载均衡

客户端检索数据时，使用的是Memstore的写缓存机制，先在Memstore找，找不到再找storefile，读/写缓存机制是将读/写的数据先放入缓存区，方便下次查询时快速查找

HBase与Oracle对比：索引不同造成行为的差异，HBase适合大量插入同事又有读的情况，HBase的瓶颈是硬盘传播速度，Oracle的瓶颈是硬盘寻道时间，HBase很适合寻找按照时间排序的指定时间范围内的场景

HBase伪分布式：

安装对应HBase版本的JDK版本

解压HBase安装包，配置Java环境变量和HBase环境变量

修改启动文件HBase安装目录/conf/hbase-env.sh中Java环境变量

hbase-site.xml：

<configuration>

<property>

<name>hbase.rootdir</name>

<value><file://HBase>数据存放路径</value>

</property>

<property>

<name>hbase.zookeeper.property.dataDir</name>

<value>ZooKeeper数据存放路径</value>

</property>

<property>

<name>hbase.unsafe.stream.capability.enforce</name>

<value>false</value>

</property>

</configuration>

HBase进程jps查看Java进程对应HMaster进程，HBase需要用到ZooKeeper进行管理，伪分布式会启动HBase自带的ZooKeeper以及相应的端口

HBase完全分布式：

时间同步，测试网络连通性，配置主机之间免秘钥登录，修改hosts文件，检查防火墙，解压安装文件，配置Java环境变量和HBase环境变量

修改启动文件HBase安装目录/conf/hbase-env.sh中Java环境变量以及设置export HBASE\_MANAGES\_ZK=false默认true使用HBase自带的ZooKeeper

hbase-site.xml：

<configuration>

<property>

<name>hbase.rootdir</name>（配置HBase的数据存放目录，可以使用活跃的HDFS主机节点，一般高可用使用自定义的名称服务名）

<value>hdfs://自定义的名称服务名/hbase</value>

</property>

<property>

<name>hbase.cluster.distributed</name>（指定HBase是否为分布式模式）

<value>true</value>

</property>

<property>

<name>hbase.zookeeper.quorum</name>（指定管理HBase的ZooKeeper集群）

<value>ZooKeeper主机名[,ZooKeeper主机名]</value>

</property>

<property>

<name>hbase.unsafe.stream.capability.enforce</name>（控制HBase是否检查流功能hflush/hsync）

<value>false</value>

</property>

</configuration>

修改regionservers文件加入集群中运行RegionServer的节点列表

新建backup-masters文件加入集群中备用HMaster的节点列表

HBase加入HDFS配置，可以修改hbase-env.sh中HADOOP\_CONF\_DIR和HBASE\_CLASSPATH环境变量，也可以在hbase-site.xml文件中加入HDFS的配置，一般是复制hdfs-site.xml到HBase安装目录/conf目录下

HBase2.1.3版本由于编译问题可能需要将HBase安装目录/lib/client-facing-thirdparty/htrace-core-3.1.0-incubating.jar文件复制到HBase安装目录/lib/目录下

hbase shell：进入HBase shell界面

HBase shell命令：

help：查看帮助命令

status：查看数据库状态

version：查看数据库版本

whoami：查看当前用户

create '表名','列族名'：创建表，列在列族内，插入数据时创建不用再建表时创建

list：列出所有表

describe '表名'：查看表结构，每一个列族对应一段描述，{NAME => '列族名', VERSIONS => '最大版本数默认1', EVICT\_BLOCKS\_ON\_CLOSE => 'false', NEW\_VERSION\_BEHAVIOR => 'false', KEEP\_DELETED\_CELLS => 'FALSE', CACHE\_DATA\_ON\_WRITE => 'false', DATA\_BLOCK\_ENCODING => 'NONE', TTL => '（生存时间timetolive，FOREVER永久）', MIN\_VERSIONS => '最小版本数默认0', REPLICATION\_SCOPE => '0', BLOOMFILTER => 'ROW', CACHE\_INDEX\_ON\_WRITE => 'false', IN\_MEMORY => '加载到内存缓冲区默认false', CACHE\_BLOOMS\_ON\_WRITE => 'false', PREFETCH\_BLOCKS\_ON\_OPEN => 'false', COMPRESSION => 'NONE', BLOCKCACHE => '读缓存默认true开启', BLOCKSIZE => '缓存大小默认65536'}

alter '表名',{NAME=>'列名',METHOD=>'修改方法'}：修改表结构，在修改或者删除表之前需要将表不可用

disable '表名'：设置不可用表

enable '表名'：设置可用表

drop '表名'：删除表

exists '表名'：查看表是否存在

is\_enabled '表名'：查看表是否可用

is\_disabled '表名'：查看表是否不可用

put '表名', '行键（即主键）', '列族名:列名', '值'：插入值，每次插入值系统会自动插入一个时间戳

get '表名', '行键', '列族:列名'：查询指定行键列族的列名的所有值

get '表名','行键',{COLUMN=>'列族:列名',TIMESTAMP=>时间戳}：查询指定行键列族列名时间戳的值

scan '表名'：进行全表扫描查看全表数据

list\_namespace：列出所有命名空间，命名空间hbase类似MySQL数据库中的mysql库保存管理HBase的信息

flush：强制内存缓冲区Memstore进行溢写到硬盘上，StoreFile文件对应伪分布式数据保存目录/data/命名空间名/表名/Region标识/列族名/HBase数据文件名

hbase hfile -p -f HBase数据文件名：解析指定的HBase数据文件并以key-value形式输出显示

delete '表名', '行键', '列族:列名'：删除指定行键列族列名的值

deleteall '表名', '行键'：删除指定行键的整行数据

count '表名'：查询表的行数，按照行键统计

truncate '表名'：清空表数据，执行顺序是先将表设置不可用，删除表，创建表，这是由于hadoop不能修改数据

HBase没有关系型数据库的条件查询即where语句，但可以通过过滤器filter实现条件查询的功能

HBase的使用场景：数据分析主题成熟，查询模式确定并且不轻易改变，传统数据库已经无法承受负荷，告诉插入，大量读取，适合海量的简单的操作

对于最近浏览历史HBase的优势：面向时间查询速度快（因为HBase有时间戳），基于行键的查询速度快，特别是最近的数据被放在内存的MemStore里，完全没有IO开销，分布式化解负荷

HBase是根据行键的范围进行分布，所以刚开始插入数据由于行键的分布范围小，会被集中在少量节点上，不能体现平衡负载均衡，可以使用将行键值数字倒排，或者使用哈希值

对于浏览过某件商品的用户还浏览了哪些商品的HBase的表设计与查询实现：一张行键为用户id列族和列为商品id，另一张表行键为商品id列族和列为用户id，先查询商品表根据商品表中的用户id查用户表里的商品id，需要使用应用程序去重和统计（因为HBase不像传统数据，没有去重和统计语法）

辅助索引：由于HBase没有索引，只能通过行键进行查找，可以自己创建辅助索引（表），将其他列设置为行键来充当索引

复合行键：将两个列设置为行键，便于分布，便于多条件伸缩查询，查询多条记录可以使用范围查询

应用程序与HBase的对接：

Thrift是一个跨语言的服务部署框架，Thrift通过一个中间语言（IDL，接口定义语言）来定义RPC的接口和数据类型，然后通过编译器生产不同语言的代码（目前支持C++、Java、Python、PHP、Ruby、Erlang、Perl、Haskell、C#、Cocoa、Smalltalk和OCaml），并由生产的代码负责RPC协议层和传输层的实现

public class HbaseStudy {

HBaseAdmin hBaseAdmin;//HBase操作对象

HTable hTable; //HBase表操作对象

String tablename="phone";

@Before

public void begin()throws Exception{

Configuration configuration=new Configuration();

configuration.set("hbase.ZooKeeper.quorum","hadoop0,hadoop1,hadoop2,hadoop3");

hBaseAdmin=new HBaseAdmin(configuration);

hTable=new HTable(configuration,tablename);

}

@After

public void end(){

if (hBaseAdmin != null) {

try {

hBaseAdmin.close();

} catch (IOException e) {

e.printStackTrace();

}

}

if (hTable!=null){

try {

hTable.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

@Test

public void insert(){//插入数据

String string="13148948393\_2016345555555";

Put put=new Put(string.getBytes());

put.add("cf1".getBytes(),"type".getBytes(),"1".getBytes());

put.add("cf1".getBytes(),"time".getBytes(),"2016".getBytes());

put.add("cf1".getBytes(),"pnum".getBytes(),"13332434533".getBytes());

try {

hTable.put(put);

} catch (InterruptedIOException e) {

e.printStackTrace();

} catch (RetriesExhaustedWithDetailsException e) {

e.printStackTrace();

}

}

@Test

public void get(){//获取数据

String string="13148948393\_2016345555555";

Get get=new Get(string.getBytes());

get.addColumn("cf1".getBytes(),"type".getBytes());

get.addColumn("cf1".getBytes(),"time".getBytes());

Result result= null;

try {

result = hTable.get(get);

} catch (IOException e) {

e.printStackTrace();

}

Cell cell=result.getColumnLatestCell("cf1".getBytes(),"type".getBytes());

System.out.println(new String(CellUtil.cloneValue(cell)));

}

Random random=new Random();

/\*\*

\* 随机生成手机号码

\* @param prefix

\* @return

\*/

public String getPhoneNum(String prefix){

return prefix+String.format("%08d",random.nextInt(99999999));

}

/\*\*

\* 随机生成时间

\* @param year

\* @return

\*/

public String getDate(String year){

return year+String.format("%02d%02d%02d%02d%02d",

new Object[]{

random.nextInt(12)+1,

random.nextInt(28),

random.nextInt(60),

random.nextInt(60),

random.nextInt(60),

});

}

/\*\*

\* 随机生成时间，年月日

\*/

public String getDate2(String prefix){

return prefix+String.format("%02d%02d%02d",

new Object[]{

random.nextInt(60),

random.nextInt(60),

random.nextInt(60),

});

}

@Test

public void insertDB(){

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

for (int i = 0; i < 10; i++) {

String rowkey;

String phoneNum=getPhoneNum("186");

for (int j = 0; j < 100; j++) {

String phoneDate=getDate("2016");

SimpleDateFormat simpleDateFormat=new SimpleDateFormat("yyyyMMddHHmmss");

long datalong= 0;

try {

datalong = simpleDateFormat.parse(phoneDate).getTime();

} catch (ParseException e) {

e.printStackTrace();

}

rowkey=phoneNum+(Long.MAX\_VALUE-datalong);

System.out.println(rowkey);

Put put=new Put(rowkey.getBytes());

put.add("cf1".getBytes(),"type".getBytes(),(random.nextInt(2)+"").getBytes());

put.add("cf1".getBytes(),"time".getBytes(),(phoneDate).getBytes());

put.add("cf1".getBytes(),"pnum".getBytes(),(getPhoneNum("170")).getBytes());

puts.add(put);

}

}

try {

hTable.put(puts);

} catch (InterruptedIOException e) {

e.printStackTrace();

} catch (RetriesExhaustedWithDetailsException e) {

e.printStackTrace();

}

}

/\*\*

\* 10个手机号一天100条通话记录

\*/

@Test

public void insertDB2(){

Phone.pday.Builder pday=Phone.pday.newBuilder();

for (int i = 0; i < 10; i++) {

String rowkey;

String phoneNum=getPhoneNum("186");

rowkey=phoneNum+"\_"+(Long.MAX\_VALUE-Long.parseLong("20161110"));

for (int j = 0; j < 100; j++) {

String phoneDate=getDate2("20161110");

Phone.pdetail.Builder detail=Phone.pdetail.newBuilder();//实例化Protobuf创建的序列化对象

detail.setPnum(getPhoneNum("177"));

detail.setTime(phoneDate);

detail.setType((random.nextInt(2)+""));

pday.addPlist(detail);

}

Put put=new Put(rowkey.getBytes());

put.add("cf1".getBytes(),"pday".getBytes(),pday.build().toByteArray());

try {

hTable.put(put);

} catch (InterruptedIOException e) {

e.printStackTrace();

} catch (RetriesExhaustedWithDetailsException e) {

e.printStackTrace();

}

}

}

/\*\*

\* 获取指定手机号一天的所有通话记录

\*/

@Test

public void getPhoneData(){

String string="18642755933\_9223372036834614697";

Get get=new Get(string.getBytes());

get.addColumn("cf1".getBytes(),"pday".getBytes());

Result result= null;

try {

result = hTable.get(get);

} catch (IOException e) {

e.printStackTrace();

}

Cell cell=result.getColumnLatestCell("cf1".getBytes(),"pday".getBytes());

try {

Phone.pday pday=Phone.pday.parseFrom(CellUtil.cloneValue(cell));

for (Phone.pdetail detail : pday.getPlistList()) {

System.out.println(detail.getPnum()+"-"+detail.getTime()+"-"+detail.getType());

}

} catch (InvalidProtocolBufferException e) {

e.printStackTrace();

}

}

/\*\*

\* 查询某个手机号某个月份下的所有通话详单

\*/

@Test

public void scanDB() throws Exception{

Scan scan=new Scan();

SimpleDateFormat simpleDateFormat=new SimpleDateFormat("yyyyMMddHHmmss");

String startRowkey="18698031814"+(Long.MAX\_VALUE-simpleDateFormat.parse("20160301000000").getTime());

scan.setStartRow(startRowkey.getBytes());

String stopRowkey="18698031814"+(Long.MAX\_VALUE-simpleDateFormat.parse("20160201000000").getTime());

scan.setStopRow(stopRowkey.getBytes());

ResultScanner resultScanner=hTable.getScanner(scan);

for (Result rs:resultScanner) {

System.out.println(new String(CellUtil.cloneValue(rs.getColumnLatestCell("cf1".getBytes(),"type".getBytes())))+"-"+

new String(CellUtil.cloneValue(rs.getColumnLatestCell("cf1".getBytes(),"time".getBytes())))+"-"+

new String(CellUtil.cloneValue(rs.getColumnLatestCell("cf1".getBytes(),"pnum".getBytes()))));

}

}

/\*\*

\* 查询某个手机号所有主叫type=0的通话详单

\*/

@Test

public void scanDB2(){

FilterList filterList=new FilterList(FilterList.Operator.MUST\_PASS\_ALL); //过滤器列表，过滤列表中MUST\_PASS\_ONE（任何一个）| MUST\_PASS\_ALL（所有）条件

PrefixFilter prefixFilter=new PrefixFilter("18698031814".getBytes());//前置过滤器，对HBase行键进行过滤

filterList.addFilter(prefixFilter);

SingleColumnValueFilter singleColumnValueFilter=new SingleColumnValueFilter("cf1".getBytes(),"type".getBytes(), CompareFilter.CompareOp.EQUAL,"0".getBytes());列过滤器，对指定列族列名的值按照EQUAL（相等）进行过滤

filterList.addFilter(singleColumnValueFilter);

Scan scan=new Scan();

scan.setFilter(filterList);

ResultScanner resultScanner= null;

try {

resultScanner = hTable.getScanner(scan);

} catch (IOException e) {

e.printStackTrace();

}

for (Result rs:resultScanner) {

String rowkey=new String(rs.getColumnLatestCell("cf1".getBytes(),"type".getBytes()).getRow());

System.out.println(rowkey+"-"+new String(CellUtil.cloneValue(rs.getColumnLatestCell("cf1".getBytes(),"type".getBytes())))+"-"+

new String(CellUtil.cloneValue(rs.getColumnLatestCell("cf1".getBytes(),"time".getBytes())))+"-"+

new String(CellUtil.cloneValue(rs.getColumnLatestCell("cf1".getBytes(),"pnum".getBytes()))));

}

}

@Test

public void createTb1()throws Exception{

if (hBaseAdmin.tableExists(tablename)){

hBaseAdmin.disableTable(tablename);

hBaseAdmin.deleteTable(tablename);

}

HTableDescriptor hTableDescriptor=new HTableDescriptor(TableName.valueOf(tablename)); //HBase表描述对象

HColumnDescriptor hColumnDescriptor=new HColumnDescriptor("cf1");

hColumnDescriptor.setBlockCacheEnabled(true);

hColumnDescriptor.setInMemory(true);

hColumnDescriptor.setMaxVersions(1);

hTableDescriptor.addFamily(hColumnDescriptor);

hBaseAdmin.createTable(hTableDescriptor);

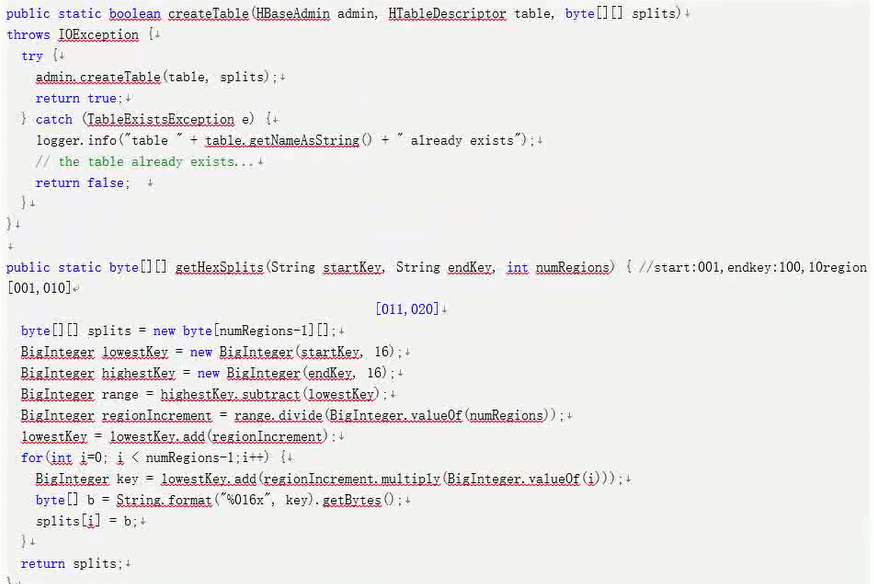
}

}

HBase优化：

Pre-Creating Regions（预分区）：

默认情况下，在创建HBase表的时候会自动创建一个Region分区，当导入数据的时候，所有的HBase客户端都向这一个Region写数据，数据量到达一定阈值后才会进行切分，可以通过预先创建一些空的Regions，按照数据区分写入不同的Region中实现数据负载均衡



Row key：

HBase中 row key用来检索表中的记录,支持三种方式：键值进行get操作、row key的range进行指定范围scan即设置startRowkey和endRow key和全表扫描

Row key的设计可以加入时间戳timestamp，Long.MAX\_VALUE-timestamp作为时间戳可以提高新写入的数据在读取时快速命中，Row key设计的一般规则是在满足业务的同时Row key尽量小和散列性（取反或哈希值，散列可以将数据均分分布但影响范围查询效率）

Column Family：

目前HBase表的列族最好在1到3个，某个列族flush时会因关联效应触发临近的列族也flush，最终导致系统产生更多的I/O影响效率

In Memory：

创建表的时候，可以通过 HColumnDescriptor.setInMemory(true)将表放到Regionserver的缓存中，保证在读取的时候被cache命中。

Max Version：

创建表的时候，可以通过 ColumnDescriptor.setMaxversions(int maxVersions)设置表中数据的最大版本数

Time To Live

创建表的时候，可以通过 ColumnDescriptor.setTimetolive(int timeToLive)设置表中数据的存储生命期，过期数据将自动被除

Compact & split：

MemStore进行flush持久化到StoreFile（minor compact），StoreFile合并（major compact），StoreFile分割（split），StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，通常合并过程比较快，可以手动进行minor compact，将同一个row key的修改进行合并形成一个大的StoreFile，以及将StoreFile设置大些，减少split

HBase的compaction操作将小StoreFile文件合并成大StoreFile文件，compaction的两种类型minor compaction（合并StoreFile文件小数量少速度快）和major compaction（合并所有StoreFile文件，合并文件大速度慢，合并过程中会影响HBase其他性能），通过major compact命令、majorCompact() API和RegionServer自动运行(相关参数: hbase.hregion.majorcompaction默认为24小时执行一次合并和hbase.hreqion.majorcompaction.jetter默认值为0.2阈值，防止所有RegionServer在同一时间进行major compaction) ，一般为了防止合并影响HBase其他性能进行手动控制major compaction关闭不使用RegionServer的自动major compaction

手动控制major compaction可以通过Java的Timer类或者Linux shell脚步contab

minor compaction相关设置参数（一般不控制minor compaction）:

hbase.hstore.compaction.min默认值为3，表示至少需要三个满足条件的StoreFile时, minor compaction才会启动

hbase.hstore.compaction.max默认值为10，表示一次minor compaction最多选取10个StoreFile

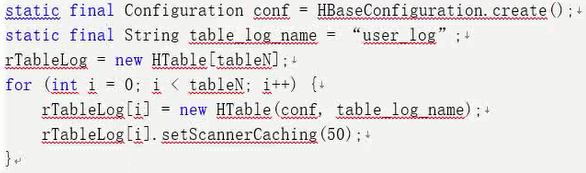
hbase.hstore.compaction.min.size表示文件大小小于指定值的StoreFile一定会加入到minor compaction的StoreFile中

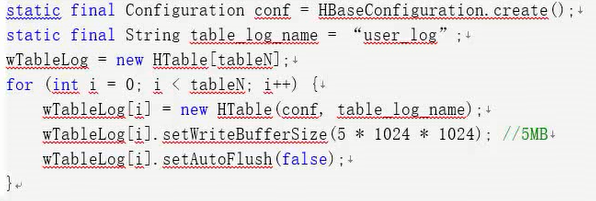
hbase.hstore.compaction.max.size表示文件大小大于指定值的StoreFile一定会被minor compaction排除

hbase.hstore.compaction.ratio将StoreFile按照文件年齡排序(older to younger)，minor compaction总是从older StoreFile开始选择

可以使用HTable.get(List)和HTable.put(List<Put>)方法代替HTable.get(Get)和HTable.put(Put)方法将指定的row key列表批量写入多行记录代替将一个指定的row key记录写入HBase减少网络I/O开销提高性能

可以创建多个HTable客户端并发读/写操作，提高读/写数据的吞吐量





可以开启多个HTabe写线程并发读/写操作（一般可以通过MapReduce的多线程代替HTable多线程并发读/写操作），每个写线程负责一个HTable对象的flush操作，既保证在数据量小的时候，数据可以在较短时间内被 flush，同时又保证在数据量大的时候，写buffer=满就及时进行flush



HTable参数设置：

Auto Flush:

通过调用HTable.setAutoFlush(false)方法将HTable写客户端的自动flush关闭，可以让put操作写满客户端缓存时才实际向HBase服务端发起写请求而不是put操作一次执行一次更新，默认auto flush是true

Write Buffer：

通过调用HTable.setWriteBufferSize(writeBufferSize)方法可以设置HTable客户端的写缓存大小，单位byte字节数

WAL Flag：

为了防止服务器宕机后可以恢复数据，HBase客户端向集群中的RegionServer提交数据时(Put/ Delete操作)，首先会先写WAL(Write Ahead Log)日志(即出HLog，一个RegionServer上的所有Region共享一个HLog)，WAL日志写成功后才会写MemStore通知客户端提交数据成功，写WAL日志失败通知客户端提交失败，可以在Put/ Delete操作时，调用Put.setWriteToWAL(false)或Delete.setWriteToWAL( false)函数关闭写WAL日志提高数据写入的性能，关闭写WAL日志后RegionServer宣机后Put/ Delete的数据无法根据WAL日志恢复，一般用于提高数据不重要写操作性能

Scanner Caching：

Hbase.client.scanner.caching配置项设置HBase scanner一次从服务端抓取的数据条数通过使用客户端的内存可以减少scan过程中next()的时间开销，默认一次一条，配置HBase scanner优先级从高到低：Scan.setCaching(int caching)、HTable.setScannerCaching(int scannerCaching)、HBase的conf配置文件中的配置

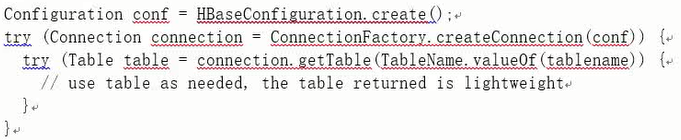
Scan Attribute Selection：指定scan操作获取的列族，可以减少网络传输数据量，默认返回表的所有列族数据

Close Resultscanner：通过scan取完数据后关闭ResultScanner，否则RegionServer可能会出现问题(对应的 Server资源无法释放)

HBase缓存在RegionServer的内存中分为写缓存Memstore（多个）和读缓存Blockcache（一个），一个Blockcache和多个Memstore总大小之和不能大于等于 heapsize \* 0.8，否则HBase不能启动，默认Blockcache为0.2，Memstore为0.4，Memstore满64MB会flush刷新到硬盘，Memstore大小超过阈值heapsize \* hbase.regionserver.global.memstore.upperLimit \* 0.9会被强制flush直至低于阈值

读请求先到Memstore中查数据，查不到会去Blockcache中查，再查不到就会到硬盘上读并把读的结果放入 Blockcache，由于Blockcache采用的是LRU策略，因此 Blockcache达到阈值heapsize \* hfile.block.cache.size \* 0.85后会启动淘汰机制删除最久的一批数据

可以通过HTablePool连接池管理HTable



HTable的创建需要消耗资源所以应避免频繁创建，HTable不是线程安全的，HTable之间共享Configuration（共享ZooKeeper的连接和公共的资源）

HBase与DBMS比较：

查询数据不灵活：不能使用column之间进行过滤查询，不支持全文索引，可以使用solr和HBase整合完成全文搜索，使用MapReduce批量读取HBase中的数据，在solr里面建立索引(no store)之保存rowley的值，根据关键词从索引中搜索到rowley(分页)，根据rowkey从HBase查询所有数据