[第5章 HBASE数据库 237](#_Toc515544325)

[5.1 数据库 OLAP、OLTP的介绍和比较 237](#_Toc515544326)

[5.2 Hbase基础 238](#_Toc515544327)

[什么是OLTP 238](#_Toc515544328)

[5.2.1 hbase数据库介绍 241](#_Toc515544329)

[5.2.2 hbase集群结构 244](#_Toc515544330)

[5.2.3 hbase集群搭建 (简易步骤) 245](#_Toc515544331)

[5.2.4 命令行演示 245](#_Toc515544332)

[5.2.5 hbase代码开发（基本，过滤器查询） 246](#_Toc515544333)

[5.2.6 hbase内部原理 247](#_Toc515544334)

[5.2.7 物理存储 249](#_Toc515544335)

[5.2.8 寻址机制 253](#_Toc515544336)

[5.2.9 读写过程 254](#_Toc515544337)

[5.2.10 Region管理 256](#_Toc515544338)

[5.2.11 Master工作机制 257](#_Toc515544339)

[5.2.12 HBase容错性 257](#_Toc515544340)

[5.3 Hbase高级应用 258](#_Toc515544341)

[5.3.1 建表高级属性 258](#_Toc515544342)

[5.3.2 HBase的设计原则 260](#_Toc515544343)

[5.3.3 rowkey长度原则 260](#_Toc515544344)

[5.3.4 什么是热点 261](#_Toc515544345)

[1）加盐 261](#_Toc515544346)

[2）哈希 262](#_Toc515544347)

[3）反转 262](#_Toc515544348)

[4）时间戳反转 262](#_Toc515544349)

[5.4 HBase性能优化完全版 263](#_Toc515544350)

[5.4.1 垃圾回收优化 263](#_Toc515544351)

[5.4.2 启用压缩，详情自行搜索，暂时未曾尝试，后面持续更新。 264](#_Toc515544352)

[5.4.3 优化Region拆分合并以及与拆分Region 264](#_Toc515544353)

[5.4.4 4.客户端入库调优 265](#_Toc515544354)

[5.4.5 HBase配置文件 266](#_Toc515544355)

[5.4.6 HDFS优化部分 270](#_Toc515544356)

[5.5 大数据Hbase 面试题 271](#_Toc515544357)

# HBASE数据库

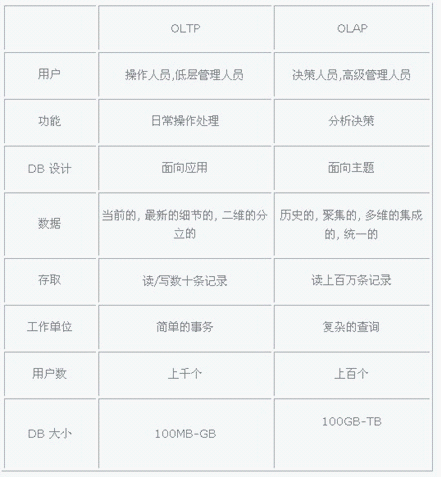
## 数据库 OLAP、OLTP的介绍和比较

数据处理大致可以分成两大类：联机事务处理OLTP（on-line transaction processing）、联机分析处理OLAP（On-Line Analytical Processing）。OLTP是传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。OLAP是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。

OLTP 系统强调数据库内存效率，强调内存各种指标的命令率，强调绑定变量，强调并发操作；

OLAP 系统则强调数据分析，强调SQL执行市场，强调磁盘I/O，强调分区等。

**OLTP与OLAP之间的比较：**



## Hbase基础

### 什么是OLTP

OLTP，也叫联机事务处理（Online Transaction Processing），表示事务性非常高的系统，一般都是高可用的在线系统，以小的事务以及小的查询为主，评估其系统的时候，一般看其每秒执行的Transaction以及Execute SQL的数量。在这样的系统中，单个数据库每秒处理的Transaction往往超过几百个，或者是几千个，Select 语句的执行量每秒几千甚至几万个。典型的OLTP系统有电子商务系统、银行、证券等，如美国eBay的业务数据库，就是很典型的OLTP数据库。

OLTP系统最容易出现瓶颈的地方就是CPU与磁盘子系统。

（1）CPU出现瓶颈常表现在逻辑读总量与计算性函数或者是过程上，逻辑读总量等于单个语句的逻辑读乘以执行次数，如果单个语句执行速度虽然很快，但是执行次数非常多，那么，也可能会导致很大的逻辑读总量。设计的方法与优化的方法就是减少单个语句的逻辑读，或者是减少它们的执行次数。另外，一些计算型的函数，如自定义函数、decode等的频繁使用，也会消耗大量的CPU时间，造成系统的负载升高，正确的设计方法或者是优化方法，需要尽量避免计算过程，如保存计算结果到统计表就是一个好的方法。

（2）磁盘子系统在OLTP环境中，它的承载能力一般取决于它的IOPS处理能力. 因为在OLTP环境中，磁盘物理读一般都是db file sequential read，也就是单块读，但是这个读的次数非常频繁。如果频繁到磁盘子系统都不能承载其IOPS的时候，就会出现大的性能问题。

OLTP比较常用的设计与优化方式为Cache技术与B-tree索引技术，Cache决定了很多语句不需要从磁盘子系统获得数据，所以，Web cache与Oracle data buffer对OLTP系统是很重要的。另外，在索引使用方面，语句越简单越好，这样执行计划也稳定，而且一定要使用绑定变量，减少语句解析，尽量减少表关联，尽量减少分布式事务，基本不使用分区技术、MV技术、并行技术及位图索引。因为并发量很高，批量更新时要分批快速提交，以避免阻塞的发生。

OLTP 系统是一个数据块变化非常频繁，SQL 语句提交非常频繁的系统。 对于数据块来说，应尽可能让数据块保存在内存当中，对于SQL来说，尽可能使用变量绑定技术来达到SQL 重用，减少物理I/O 和重复的SQL 解析，从而极大的改善数据库的性能。

这里影响性能除了绑定变量，还有可能是热快（hot block）。 当一个块被多个用户同时读取时，Oracle 为了维护数据的一致性，需要使用Latch来串行化用户的操作。当一个用户获得了latch后，其他用户就只能等待，获取这个数据块的用户越多，等待就越明显。 这就是热快的问题。 这种热快可能是数据块，也可能是回滚端块。 对于数据块来讲，通常是数据库的数据分布不均匀导致，如果是索引的数据块，可以考虑创建反向所以来达到重新分布数据的目的，对于回滚段数据块，可以适当多增加几个回滚段来避免这种争用。

#### 什么是OLAP

OLAP，也叫联机分析处理（Online Analytical Processing）系统，有的时候也叫DSS决策支持系统，就是我们说的数据仓库。在这样的系统中，语句的执行量不是考核标准，因为一条语句的执行时间可能会非常长，读取的数据也非常多。所以，在这样的系统中，考核的标准往往是磁盘子系统的吞吐量（带宽），如能达到多少MB/s的流量。

磁盘子系统的吞吐量则往往取决于磁盘的个数，这个时候，Cache基本是没有效果的，数据库的读写类型基本上是db file scattered read与direct path read/write。应尽量采用个数比较多的磁盘以及比较大的带宽，如4Gb的光纤接口。

**在OLAP系统中，常使用分区技术、并行技术。**

分区技术在OLAP系统中的重要性主要体现在数据库管理上，比如数据库加载，可以通过分区交换的方式实现，备份可以通过备份分区表空间实现，删除数据可以通过分区进行删除，至于分区在性能上的影响，它可以使得一些大表的扫描变得很快（只扫描单个分区）。另外，如果分区结合并行的话，也可以使得整个表的扫描会变得很快。总之，分区主要的功能是管理上的方便性，它并不能绝对保证查询性能的提高，有时候分区会带来性能上的提高，有时候会降低。

并行技术除了与分区技术结合外，在Oracle 10g中，与RAC结合实现多节点的同时扫描，效果也非常不错，可把一个任务，如select的全表扫描，平均地分派到多个RAC的节点上去。

在OLAP系统中，不需要使用绑定（BIND）变量，因为整个系统的执行量很小，分析时间对于执行时间来说，可以忽略，而且可避免出现错误的执行计划。但是OLAP中可以大量使用位图索引，物化视图，对于大的事务，尽量寻求速度上的优化，没有必要像OLTP要求快速提交，甚至要刻意减慢执行的速度。

绑定变量真正的用途是在OLTP系统中，这个系统通常有这样的特点，用户并发数很大，用户的请求十分密集，并且这些请求的SQL 大多数是可以重复使用的。

对于OLAP系统来说，绝大多数时候数据库上运行着的是报表作业，执行基本上是聚合类的SQL 操作，比如group by，这时候，把优化器模式设置为all\_rows是恰当的。 而对于一些分页操作比较多的网站类数据库，设置为first\_rows会更好一些。 但有时候对于OLAP 系统，我们又有分页的情况下，我们可以考虑在每条SQL 中用hint。 如：

Select /\*+first\_rows(10) \*/ a.\* from table a;

#### 分开设计与优化

在设计上要特别注意，如在高可用的OLTP环境中，不要盲目地把OLAP的技术拿过来用。

如分区技术，假设不是大范围地使用分区关键字，而采用其它的字段作为where条件，那么，如果是本地索引，将不得不扫描多个索引，而性能变得更为低下。如果是全局索引，又失去分区的意义。

并行技术也是如此，一般在完成大型任务时才使用，如在实际生活中，翻译一本书，可以先安排多个人，每个人翻译不同的章节，这样可以提高翻译速度。如果只是翻译一页书，也去分配不同的人翻译不同的行，再组合起来，就没必要了，因为在分配工作的时间里，一个人或许早就翻译完了。

位图索引也是一样，如果用在OLTP环境中，很容易造成阻塞与死锁。但是，在OLAP环境中，可能会因为其特有的特性，提高OLAP的查询速度。MV也是基本一样，包括触发器等，在DML频繁的OLTP系统上，很容易成为瓶颈，甚至是Library Cache等待，而在OLAP环境上，则可能会因为使用恰当而提高查询速度。

对于OLAP系统，在内存上可优化的余地很小，增加CPU 处理速度和磁盘I/O 速度是最直接的提高数据库性能的方法，当然这也意味着系统成本的增加。

比如我们要对几亿条或者几十亿条数据进行聚合处理，这种海量的数据，全部放在内存中操作是很难的，同时也没有必要，因为这些数据快很少重用，缓存起来也没有实际意义，而且还会造成物理I/O相当大。 所以这种系统的瓶颈往往是磁盘I/O上面的。

对于OLAP系统，SQL 的优化非常重要，因为它的数据量很大，做全表扫描和索引对性能上来说差异是非常大的。

### hbase数据库介绍

#### 简介

hbase是基于Google BigTable模型开发的，典型的key/value系统。是建立在hdfs之上，提供高可靠性、高性能、列存储、可伸缩、实时读写nosql的数据库系统。主要用于海量结构化和半结构化数据存储。

它介于nosql和RDBMS之间，仅能通过主键(row key)和主键的range来检索数据，仅支持单行事务(可通过hive支持来实现多表join等复杂操作)。

Hbase查询数据功能很简单，不支持join等复杂操作，不支持复杂的事务（行级的事务）

与hadoop一样，Hbase目标主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。

#### **HBase中的表的特点：**

* 大：一个表可以有上十亿行，上百万列
* 无模式：每行都有一个可排序的主键和任意多的列，列可以根据需要动态的增加，同一张表中不同的行可以有截然不同的列；
* 面向列:面向列(族)的存储和权限控制，列(族)独立检索。
* 稀疏:对于为空(null)的列，并不占用存储空间，因此，表可以设计的非常稀疏。
* 数据多版本：每个单元中的数据可以有多个版本，默认情况下版本号自动分配，是单元格插入时的时间戳
* 数据类型单一：Hbase中的数据都是字节数组 byte[]。

#### 表结构逻辑视图

HBase以表的形式存储数据。表有行和列组成。列划分为若干个列族(column family)



**1、Row Key**

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。

**Hbase会对表中的数据按照rowkey排序(字典顺序)**

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。

行的一次读写是原子操作 (不论一次读写多少列)。这个设计决策能够使用户很容易的理解程序在对同一个行进行并发更新操作时的行为。

**2、列族**

hbase表中的每个列，都归属与某个列族。列族是表的schema的一部分(而列不是)，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。

列族越多，在取一行数据时所要参与IO、搜寻的文件就越多，所以，如果没有必要，不要设置太多的列族。一般设置2-3个比较合理。

**3、时间戳**

HBase中通过row和columns确定的为一个存贮单元称为cell。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：

* 保存数据的最后n个版本
* 保存最近一段时间内的版本（设置数据的生命周期TTL）。

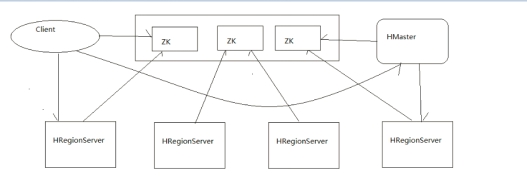
用户可以针对每个列族进行设置。

**4、Cell**

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。

### hbase集群结构



Hbase基本组件说明：

|  |
| --- |
| **Client：**  包含访问Hbase的接口，并维护cache来加快对Hbase的访问，比如region的位置信息。  **HMaster：**  是hbase集群的主节点，可以配置多个，用来实现HA  为RegionServer分配region  负责RegionServer的负载均衡  发现失效的RegionServer并重新分配其上的region  **RegionServer：**  Regionserver维护region，处理对这些region的IO请求  Regionserver负责切分在运行过程中变得过大的region  **Region:**  分布式存储的最小单元。  **Zookeeper作用:**  通过选举，保证任何时候，集群中只有一个活着的HMaster，HMaster与RegionServers 启动时会向ZooKeeper注册  存贮所有Region的寻址入口  实时监控Region server的上线和下线信息。并实时通知给HMaster  存储HBase的schema和table元数据  Zookeeper的引入使得HMaster不再是单点故障 |

### hbase集群搭建 (简易步骤)

--先部署一个zookeeper集群 ,并同步所有服务器的时间

1.配置hbase集群，要修改3个文件 hbase-env.sh, hbase-site.xml, regionservers, backup-masters来指定备用的主节点

注意：要把hadoop的hdfs-site.xml和core-site.xml 放到hbase/conf下

2.启动: ./zkServer.sh start, start-dfs.sh, start-hbase.sh

3.访问: itcast01:16010

### 命令行演示

#### 基本shell命令

|  |
| --- |
| 进入hbase命令行  ./hbase shell  Create ‘tablename’，’column family’  Put ‘tablename’,’rowkey’,’column family:’,’value’  Put ‘tablename’,’rowkey’,’column family:column’,’value’  Scan ‘tablename’  scan 'user\_test',{COLUMNS =>'info:username',LIMIT =>10, STARTROW => 'test',STOPROW=>'test2'}  scan 'tablename',{COLUMNS => 'column family'}  Get ‘tablename’,’rowkey’  Get ‘tablename’,’rowkey’,’column family’  Get ‘tablename’,’rowkey’,’column family:column’  delete 'tablename','rowkey','column family:column'  deleteall ‘tablename’,’rowkey’  删除列族: disable ‘tablename’  Alter ‘tablename’,{NAME=>’column family’,METHOD=>’delete’}  删除表: disable ‘tablename’  drop ‘tablename’ |

### hbase代码开发（基本，过滤器查询）

#### 1.基本增删改查java实现 –例子

|  |
| --- |
| private Configuration conf = null;  @Before  public void init(){  conf = HBaseConfiguration.create();  conf.set("hbase.zookeeper.quorum", "itcast01:2181,itcast02:2181,itcast03:2181");  }    @Test  public void testDrop() throws Exception{  HBaseAdmin admin = new HBaseAdmin(conf);  admin.disableTable("account");  admin.deleteTable("account");  admin.close();  } |
|  |

#### 2.过滤器查询 (基于scan)

引言：过滤器的类型很多，但是可以分为两大类——比较过滤器，专用过滤器

过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；

hbase过滤器的比较运算符：

|  |
| --- |
| LESS < EQUAL= GREATER > NO\_OP 排除所有 not <> |

Hbase过滤器的比较器（指定比较机制）：

|  |
| --- |
| BinaryComparator 按字节索引顺序比较指定字节数组，采用Bytes.compareTo(byte[])  BinaryPrefixComparator 跟前面相同，只是比较左端的数据是否相同  NullComparator 判断给定的是否为空  BitComparator 按位比较  RegexStringComparator 提供一个正则的比较器，仅支持 EQUAL 和非EQUAL  SubstringComparator 判断提供的子串是否出现在value中。 |

Hbase的过滤器分类

* 比较过滤器
  1. 行键过滤器RowFilter --为例
  2. 列族过滤器FamilyFilter
  3. 列过滤器QualifierFilter
  4. 值过滤器 ValueFilter

|  |
| --- |
| Filter filter1 = new RowFilter(CompareOp.LESS\_OR\_EQUAL, new BinaryComparator(Bytes.toBytes("row-22")));  scan.setFilter(filter1); |

* 专用过滤器

2.1 单列值过滤器 SingleColumnValueFilter ----会返回满足条件的整行

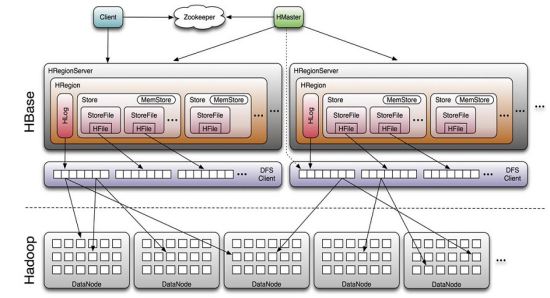
2.2前缀过滤器 PrefixFilter----针对行键

2.3 列前缀过滤器 ColumnPrefixFilter

|  |
| --- |
| SingleColumnValueFilter filter = new SingleColumnValueFilter( Bytes.toBytes("colfam1"), Bytes.toBytes("col-5"),  CompareFilter.CompareOp.NOT\_EQUAL, new SubstringComparator("val-5"));  filter.setFilterIfMissing(true); //如果不设置为true，则那些不包含指定column的行也会返回  scan.setFilter(filter1); |

### hbase内部原理

#### 系统架构



Client

1 包含访问hbase的接口，client维护着一些cache来加快对hbase的访问，比如region的位置信息。

Zookeeper

1 保证任何时候，集群中只有一个master

2 存贮所有Region的寻址入口----root表在哪台服务器上。

3 实时监控Region Server的状态，将Region server的上线和下线信息实时通知给Master

4 存储Hbase的schema,包括有哪些table，每个table有哪些column family

Master职责

1 为Region server分配region

2 负责region server的负载均衡

3 发现失效的region server并重新分配其上的region

4 HDFS上的垃圾文件回收

5 处理schema更新请求

Region Server职责

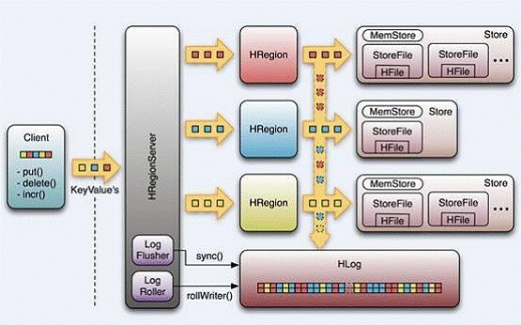
1 Region server维护Master分配给它的region，处理对这些region的IO请求

2 Region server负责切分在运行过程中变得过大的region

可以看到，client访问hbase上数据的过程并不需要master参与（寻址访问zookeeper和region server，数据读写访问regione server），master仅仅维护者table和region的元数据信息，负载很低。

### 物理存储

##### 1、整体结构



1 Table中的所有行都按照row key的字典序排列。

2 Table 在行的方向上分割为多个Hregion。

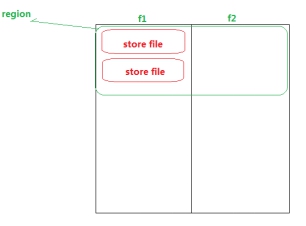
3 region按大小分割的(默认10G)，每个表一开始只有一个region，随着数据不断插入表，region不断增大，当增大到一个阀值的时候，Hregion就会等分会两个新的Hregion。当table中的行不断增多，就会有越来越多的Hregion。

4 Hregion是Hbase中分布式存储和负载均衡的最小单元。最小单元就表示不同的Hregion可以分布在不同的HRegion server上。但一个Hregion是不会拆分到多个regionserver上的。

5 HRegion虽然是负载均衡的最小单元，但并不是物理存储的最小单元。

事实上，HRegion由一个或者多个Store组成，每个store保存一个column family。

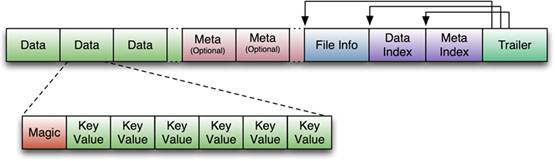
每个Strore又由一个memStore和0至多个StoreFile组成。如上图



##### 2、STORE FILE & HFILE结构

StoreFile以HFile格式保存在HDFS上。

附：HFile的格式为：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0080.jpg)

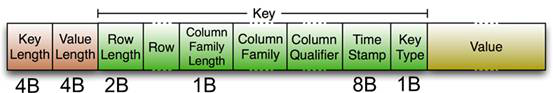
首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。正如图中所示的，Trailer中有指针指向其他数据块的起始点。

File Info中记录了文件的一些Meta信息，例如：AVG\_KEY\_LEN, AVG\_VALUE\_LEN, LAST\_KEY, COMPARATOR, MAX\_SEQ\_ID\_KEY等。

Data Index和Meta Index块记录了每个Data块和Meta块的起始点。

Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询。 每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成, Magic内容就是一些随机数字，目的是防止数据损坏。

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：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0090.jpg)

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。

HFile分为六个部分：

Data Block 段–保存表中的数据，这部分可以被压缩

Meta Block 段 (可选的)–保存用户自定义的kv对，可以被压缩。

File Info 段–Hfile的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。

Data Block Index 段–Data Block的索引。每条索引的key是被索引的block的第一条记录的key。

Meta Block Index段 (可选的)–Meta Block的索引。

Trailer–这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先 读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰。

HFile的Data Block，Meta Block通常采用压缩方式存储，压缩之后可以大大减少网络IO和磁盘IO，随之而来的开销当然是需要花费cpu进行压缩和解压缩。

目标Hfile的压缩支持两种方式：Gzip，Lzo。

##### 3、Memstore与storefile

一个region由多个store组成，每个store包含一个列族的所有数据。

Store包括位于内存的memstore和位于硬盘的storefile。

写操作先写入memstore,当memstore中的数据量达到某个阈值，Hregionserver启动flashcache进程写入storefile,每次写入形成单独一个storefile。

当storefile的个数超过一定阈值后（默认参数hbase.hstore.blockingStoreFiles=10），多个storeFile会进行合并，当该region的所有store的storefile大小之和，即所有store的大小超过hbase.hregion.max.filesize=10G时，这个region会被拆分会把当前的region分割成两个，并由Hmaster分配给相应的region服务器，实现负载均衡。

客户端检索数据时，先在memstore找，找不到再找storefile。

##### 4、HLog(WAL log)

WAL 意为Write ahead log(http://en.wikipedia.org/wiki/Write-ahead\_logging)，该机制用于数据的容错和恢复，Hlog记录数据的所有变更,一旦数据修改，就可以从log中进行恢复。

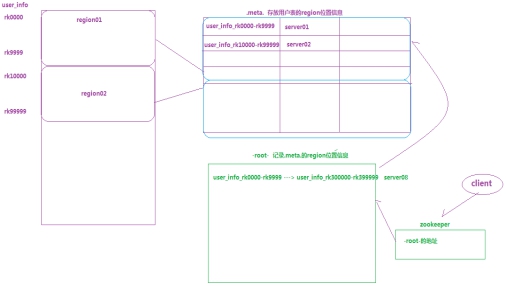
每个HRegionServer中都有一个HLog对象，HLog是一个实现Write Ahead Log的类，在每次用户操作写入MemStore的同时，也会写一份数据到HLog文件中（HLog文件格式见后续），HLog文件定期会滚动出新的，并删除旧的文件（已持久化到StoreFile中的数据）。当HRegionServer意外终止后，HMaster会通过Zookeeper感知到，HMaster首先会处理遗留的 HLog文件，将其中不同Region的Log数据进行拆分，分别放到相应region的目录下，然后再将失效的region重新分配，领取到这些region的HRegionServer在Load Region的过程中，会发现有历史HLog需要处理，因此会Replay HLog中的数据到MemStore中，然后flush到StoreFiles，完成数据恢复。

HLog文件就是一个普通的Hadoop Sequence File：

* HLog 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 Sequece File的Value是HBase的KeyValue对象，即对应HFile中的KeyValue，可参见上文描述。

### 寻址机制

#### 1、寻址示意图

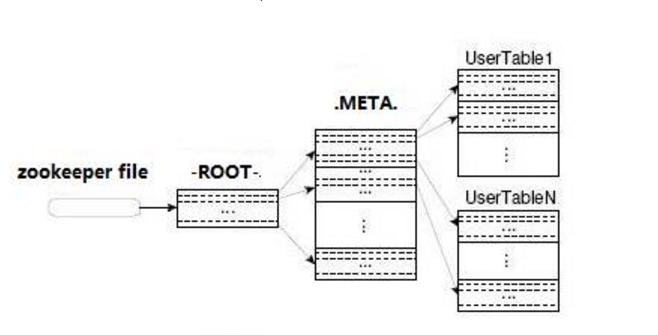


#### -ROOT-和.META.表结构

-ROOT-表结构 : rowkey info(regioninfo server serverstartcode ) historiam

**.META.行记录结构** : rowkey info(regioninfo server serverstartcode ) historiam

#### 3、寻址流程



系统如何找到某个row key (或者某个 row key range)所在的region

bigtable 使用三层类似B+树的结构来保存region位置。

第一层：保存zookeeper里面的文件，它持有root region的位置。

第二层：root region是.META.表的第一个region其中保存了.META.表其它region的位置。通过root region，我们就可以访问.META.表的数据。

第三层：.META.表它是一个特殊的表，保存了hbase中所有数据表的region 位置信息。

说明：

(1) root region永远不会被split，保证了最需要三次跳转，就能定位到任意region 。

(2).META.表每行保存一个region的位置信息，row key 采用表名+表的最后一行编码而成。

(3) 为了加快访问，.META.表的全部region都保存在内存中。

(4) client会将查询过的位置信息保存缓存起来，缓存不会主动失效，因此如果client上的缓存全部失效，则需要进行最多6次网络来回，才能定位到正确的region(其中三次用来发现缓存失效，另外三次用来获取位置信息)。

### 读写过程

#### 1、读请求过程：

（1） 客户端通过zookeeper以及root表和meta表找到目标数据所在的regionserver

（2）联系regionserver查询目标数据

（3）regionserver定位到目标数据所在的region，发出查询请求

（4）region先在memstore中查找，命中则返回

（5）如果在memstore中找不到，则在storefile中扫描（可能会扫描到很多的storefile----bloomfilter布隆过滤器）

补充：布隆过滤器参数类型有2种：

Row、row+col

#### 2、写请求过程：

（1）client向region server提交写请求

（2）region server找到目标region

（3）region检查数据是否与schema一致

（4）如果客户端没有指定版本，则获取当前系统时间作为数据版本

（5）将更新写入WAL log

（6）将更新写入Memstore

（7）判断Memstore的是否需要flush为StoreFile文件。

细节描述：

hbase使用MemStore和StoreFile存储对表的更新。

数据在更新时首先写入Log(WAL log)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时，系统会在zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了。

当系统出现意外时，可能导致内存(MemStore)中的数据丢失，此时使用Log(WAL log)来恢复checkpoint之后的数据。

StoreFile是只读的，一旦创建后就不可以再修改。因此Hbase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并(minor\_compact, major\_compact),将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行split，等分为两个StoreFile。

由于对表的更新是不断追加的，compact时，需要访问Store中全部的 StoreFile和MemStore，将他们按row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，合并的过程还是比较快。

### Region管理

(1) region分配

任何时刻，一个region只能分配给一个region server。master记录了当前有哪些可用的region server。以及当前哪些region分配给了哪些region server，哪些region还没有分配。当需要分配的新的region，并且有一个region server上有可用空间时，master就给这个region server发送一个装载请求，把region分配给这个region server。region server得到请求后，就开始对此region提供服务。

(2) region server上线

master使用zookeeper来跟踪region server状态。当某个region server启动时，会首先在zookeeper上的server目录下建立代表自己的znode。由于master订阅了server目录上的变更消息，当server目录下的文件出现新增或删除操作时，master可以得到来自zookeeper的实时通知。因此一旦region server上线，master能马上得到消息。

(3) region server下线

当region server下线时，它和zookeeper的会话断开，zookeeper而自动释放代表这台server的文件上的独占锁。master就可以确定：

1 region server和zookeeper之间的网络断开了。

2 region server挂了。

无论哪种情况，region server都无法继续为它的region提供服务了，此时master会删除server目录下代表这台region server的znode数据，并将这台region server的region分配给其它还活着的同志。

### Master工作机制

* master上线

master启动进行以下步骤:

（1） 从zookeeper上获取唯一一个代表active master的锁，用来阻止其它master成为活着的master。

（2）扫描zookeeper上的server父节点，获得当前可用的region server列表。

（3）和每个region server通信，获得当前已分配的region和region server的对应关系。

（4）扫描.META.region的集合，计算得到当前还未分配的region，将他们放入待分配region列表。

* master下线

由于master只维护表和region的元数据，而不参与表数据IO的过程，master下线仅导致所有元数据的修改被冻结(无法创建删除表，无法修改表的schema，无法进行region的负载均衡，无法处理region 上下线，无法进行region的合并，唯一例外的是region的split可以正常进行，因为只有region server参与)，表的数据读写还可以正常进行。因此master下线短时间内对整个hbase集群没有影响。

### HBase容错性

**Master容错**：Zookeeper重新选择一个新的Master  
 无Master过程中，数据读取仍照常进行；  
 无Master过程中，region切分、负载均衡等无法进行；  
**RegionServer容错**：定时向Zookeeper汇报心跳，如果一旦时间内未出现心跳，Master将该RegionServer上的Region重新分配到其他RegionServer上，失效服务器上“预写”日志由主服务器进行分割并派送给新的RegionServer  
**Zookeeper容错**：Zookeeper是一个可靠地服务，一般配置3或5个Zookeeper实例

## Hbase高级应用

### 建表高级属性

下面几个shell 命令在hbase操作中可以起到很到的作用，且主要体现在建表的过程中，看下面几个create 属性

1、BLOOMFILTER 默认是NONE 是否使用布隆过虑及使用何种方式

布隆过滤可以每列族单独启用。

使用 HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL) 对列族单独启用布隆。

* Default = ROW 对行进行布隆过滤。
* 对 ROW，行键的哈希在每次插入行时将被添加到布隆。
* 对 ROWCOL，行键 + 列族 + 列族修饰的哈希将在每次插入行时添加到布隆

使用方法: create 'table',{BLOOMFILTER =>'ROW'}

启用布隆过滤可以节省读磁盘过程，可以有助于降低读取延迟

2、VERSIONS 默认是1 这个参数的意思是数据保留1个版本，如果我们认为我们的数据没有这么大的必要保留这么多，随时都在更新，而老版本的数据对我们毫无价值，那将此参数设为1 能节约2/3的空间

使用方法: create 'table',{VERSIONS=>'2'}

附：MIN\_VERSIONS => '0'是说在compact操作执行之后，至少要保留的版本

3、COMPRESSION 默认值是NONE 即不使用压缩

这个参数意思是该列族是否采用压缩，采用什么压缩算法

使用方法: create 'table',{NAME=>'info',COMPRESSION=>'SNAPPY'}

建议采用SNAPPY压缩算法

HBase中，在Snappy发布之前（Google 2011年对外发布Snappy），采用的LZO算法，目标是达到尽可能快的压缩和解压速度，同时减少对CPU的消耗；

在Snappy发布之后，建议采用Snappy算法（参考《HBase: The Definitive Guide》），具体可以根据实际情况对LZO和Snappy做过更详细的对比测试后再做选择。

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **% remaining** | **Encoding** | **Decoding** |
| GZIP | 13.4% | 21 MB/s | 118 MB/s |
| LZO | 20.5% | 135 MB/s | 410 MB/s |
| Zippy/Snappy | 22.2% | 172 MB/s | 409 MB/s |

如果建表之初没有压缩，后来想要加入压缩算法，可以通过alter修改schema

4、alter

使用方法：如 修改压缩算法

disable 'table'

alter 'table',{NAME=>'info',COMPRESSION=>'snappy'}

enable 'table'

但是需要执行major\_compact 'table' 命令之后 才会做实际的操作。

5、TTL

默认是 2147483647 即:Integer.MAX\_VALUE 值大概是68年

这个参数是说明该列族数据的存活时间，单位是s

这个参数可以根据具体的需求对数据设定存活时间，超过存过时间的数据将在表中不在显示，待下次major compact的时候再彻底删除数据.

注意的是TTL设定之后 MIN\_VERSIONS=>'0' 这样设置之后，TTL时间戳过期后，将全部彻底删除该family下所有的数据，如果MIN\_VERSIONS 不等于0那将保留最新的MIN\_VERSIONS个版本的数据，其它的全部删除，比如MIN\_VERSIONS=>'1' 届时将保留一个最新版本的数据，其它版本的数据将不再保存。

6、describe 'table' 这个命令查看了create table 的各项参数或者是默认值。

7、disable\_all 'toplist.\*' disable\_all 支持正则表达式，并列出当前匹配的表：

toplist\_a\_total\_1001

...

Disable the above 25 tables (y/n)? 并给出确认提示.

8、drop\_all 这个命令和disable\_all的使用方式是一样的

9、hbase 表预分区----手动分区

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，直到这个region足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的regions，这样当数据写入HBase时，会按照region分区情况，在集群内做数据的负载均衡。

命令方式:

create 't1', 'f1', {NUMREGIONS => 15, SPLITALGO => 'HexStringSplit'}

也可以使用api的方式:

bin/hbase org.apache.hadoop.hbase.util.RegionSplitter test\_table HexStringSplit -c 10 -f info

参数：

HexStringSplit:split 方式 -c 是分10个region -f 是family

这样就可以将表预先分为15个区，减少数据达到storefile 大小的时候自动分区的时间消耗，并且还有以一个优势，就是合理设计rowkey 能让各个region 的并发请求平均分配(趋于均匀) 使IO 效率达到最高，但是预分区需要将filesize 设置一个较大的值，设置哪个参数呢, hbase.hregion.max.filesize 这个值默认是10G 也就是说单个region 默认大小是10G,

这个参数的默认值在0.90 到0.92到0.94.3各版本的变化：256M--1G--10G

但是如果MapReduce Input类型为TableInputFormat 使用hbase作为输入的时候，就要注意了，每个region一个map，如果数据小于10G 那只会启用一个map 造成很大的资源浪费，这时候可以考虑适当调小该参数的值，或者采用预分配region的方式，并将检测如果达到这个值，再手动分配region。

### [HBase的设计原则](http://blog.csdn.net/javajxz008/article/details/51892967)

HBase是三维有序存储的，通过rowkey（行键），column key（column family和qualifier）和TimeStamp（时间戳）这个三个维度可以对HBase中的数据进行快速定位。

HBase中rowkey可以唯一标识一行记录，在HBase查询的时候，有以下几种方式：

1. 通过get方式，指定rowkey获取唯一一条记录
2. 通过scan方式，设置startRow和stopRow参数进行范围匹配
3. 全表扫描，即直接扫描整张表中所有行记录

### rowkey长度原则

rowkey是一个二进制码流，可以是任意字符串，最大长度64kb，实际应用中一般为10-100bytes，以byte[]形式保存，一般设计成定长。

建议越短越好，不要超过16个字节，原因如下：

* 数据的持久化文件HFile中是按照KeyValue存储的，如果rowkey过长，比如超过100字节，1000w行数据，光rowkey就要占用100\*1000w=10亿个字节，将近1G数据，这样会极大影响HFile的存储效率；
* MemStore将缓存部分数据到内存，如果rowkey字段过长，内存的有效利用率就会降低，系统不能缓存更多的数据，这样会降低检索效率。

##### 1.rowkey散列原则

如果rowkey按照时间戳的方式递增，不要将时间放在二进制码的前面，建议将rowkey的高位作为散列字段，由程序随机生成，低位放时间字段，这样将提高数据均衡分布在每个RegionServer，以实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息，所有的数据都会集中在一个RegionServer上，这样在数据检索的时候负载会集中在个别的RegionServer上，造成热点问题，会降低查询效率。

##### 2.rowkey唯一原则

必须在设计上保证其唯一性，rowkey是按照字典顺序排序存储的，因此，设计rowkey的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

### 什么是热点

HBase中的行是按照rowkey的字典顺序排序的，

热点发生在大量的client直接访问集群的一个或极少数个节点（访问可能是读，写或者其他操作）。大量访问会使热点region所在的单个机器超出自身承受能力，引起性能下降甚至region不可用，这也会影响同一个RegionServer上的其他region，由于主机无法服务其他region的请求。

避免热点的方法

### 1）加盐

在rowkey的前面增加随机数，具体就是给rowkey分配一个随机前缀以使得它和之前的rowkey的开头不同。分配的前缀种类数量应该和你想使用数据分散到不同的region的数量一致。加盐之后的rowkey就会根据随机生成的前缀分散到各个region上，以避免热点。

### 2）哈希

哈希会使同一行永远用一个前缀加盐。哈希也可以使负载分散到整个集群，但是读却是可以预测的。使用确定的哈希可以让客户端重构完整的rowkey，可以使用get操作准确获取某一个行数据。

### 3）反转

第三种防止热点的方法时反转固定长度或者数字格式的rowkey。这样可以使得rowkey中经常改变的部分（最没有意义的部分）放在前面。这样可以有效的随机rowkey，但是牺牲了rowkey的有序性。

反转rowkey的例子以手机号为rowkey，可以将手机号反转后的字符串作为rowkey，这样的就避免了以手机号那样比较固定开头导致热点问题

### 4）时间戳反转

一个常见的数据处理问题是快速获取数据的最近版本，使用反转的时间戳作为rowkey的一部分对这个问题十分有用，可以用 Long.Max\_Value - timestamp 追加到key的末尾，例如 [key][reverse\_timestamp] , [key] 的最新值可以通过scan [key]获得[key]的第一条记录，因为HBase中rowkey是有序的，第一条记录是最后录入的数据。

列族尽可能越短越好，最好是一个字符，冗长的属性名虽然可读性好，但是更短的属性名存储在HBase中会更好。

## HBase性能优化完全版

### **垃圾回收优化**

**Java本身提供了垃圾回收机制，依靠JRE对程序行为的各种假设进行垃圾回收，但是HBase支持海量数据持续入库，非常占用内存，因此繁重的负载会迫使内存分配策略无法安全地依赖于JRE的判断：需要调整JRE的参数来调整垃圾回收策略。有关java内存回收机制的问题具体请参考：**<http://my.oschina.net/sunnywu/blog/332870>**。**

（1）HBASE\_OPTS或者HBASE\_REGIONSERVER\_OPT变量来设置垃圾回收的选项，后面一般是用于配置RegionServer的，需要在每个子节点的HBASE\_OPTS文件中进行配置。

1）首先是设置新生代大小的参数，不能过小，过小则导致年轻代过快成为老生代，引起老生代产生内存随便。同样不能过大，过大导致所有的JAVA进程停止时间长。-XX:MaxNewSize=256m-XX:NewSize=256m 这两个可以合并成为-Xmn256m这一个配置来完成。

2）其次是设置垃圾回收策略：-XX:+UseParNewGC -XX:+UseConcMarkSweepGC也叫收集器设置。

3）设置CMS的值，占比多少时，开始并发标记和清扫检查。-XX:CMSInitiatingOccupancyFraction=70

4）打印垃圾回收信息：-verbose:gc -XX: +PrintGCDetails -XX:+PrintGCTimeStamps

-Xloggc:$HBASE\_HOME/logs/gc-$(hostname)-hbase.log

最终可以得到：HBASE\_REGIONSERVER\_OPT="-Xmx8g -Xms8g –Xmn256m -XX:+UseParNewGC -XX:+UseConcMarkSweepGC  \

-XX:CMSInitiatingOccupancyFraction=70   -verbose:gc \

-XX:+PrintGCDetails -XX:+PrintGCTimeStamps \

-Xloggc:$HBASE\_HOME/logs/gc-$(hostname)-hbase.log

（2）hbase.hregion.memstore.mslab.enabled默认值：true，这个是在hbase-site.xml中进行配置的值。

说明：减少因内存碎片导致的Full GC，提高整体性能。

### **启用压缩，详情自行搜索，暂时未曾尝试，后面持续更新。**

### **优化Region拆分合并以及与拆分Region**

（1）hbase.hregion.max.filesize默认为256M（在hbase-site.xml中进行配置），当region达到这个阈值时，会自动拆分。可以把这个值设的无限大，则可以关闭HBase自动管理拆分，手动运行命令来进行region拆分，这样可以在不同的region上交错运行，分散I/O负载。

（2）预拆分region

用户可以在建表的时候就制定好预设定的region，这样就可以避免后期region自动拆分造成I/O负载。

### **4.客户端入库调优**

（1）用户在编写程序入库时，HBase的自动刷写是默认开启的，即用户每一次put都会提交到HBase server进行一次刷写，如果需要高速插入数据，则会造成I/O负载过重。在这里可以关闭自动刷写功能，setAutoFlush(false)。如此，put实例会先写到一个缓存中，这个缓存的大小通过hbase.client.write.buffer这个值来设定缓存区，当缓存区被填满之后才会被送出。如果想要显示刷写数据，可以调用flushCommits()方法。

**此处引申：采取这个方法要估算服务器端内存占用则可以：hbase.client.write.buffer\*hbase.regionserver.handler.count得出内存情况。**

（2）第二个方法，是关闭每次put上的WAL（writeToWAL(flase)）这样可以刷写数据前，不需要预写日志，但是如果数据重要的话建议不要关闭。

（3）hbase.client.scanner.caching：默认为1

这是设计客户端读取数据的配置调优，在hbase-site.xml中进行配置，代表scanner一次缓存多少数据（从服务器一次抓取多少数据来scan）默认的太小，但是对于大文件，值不应太大。

（4）hbase.regionserver.lease.period默认值：60000

说明：客户端租用HRegion server 期限，即超时阀值。

调优：这个配合hbase.client.scanner.caching使用，如果内存够大，但是取出较多数据后计算过程较长，可能超过这个阈值，适当可设置较长的响应时间以防被认为宕机。

（5）还有诸多实践，如设置过滤器，扫描缓存等，指定行扫描等多种客户端调优方案，需要在实践中慢慢挖掘。

### **HBase配置文件**

**上面涉及到的调优内容或多或少在HBase配置文件中都有所涉及，因此，下面的配置不涵盖上面已有的配置。**

(1) zookeeper.session.timeout（默认3分钟）

ZK的超期参数，默认配置为3分钟，在生产环境上建议减小这个值在1分钟或更小。

设置原则：这个值越小，当RS故障时Hmaster获知越快，Hlog分裂和region 部署越快，集群恢复时间越短。 但是，设置这个值得原则是留足够的时间进行GC回收，否则会导致频繁的RS宕机。一般就做默认即可

（2）hbase.regionserver.handler.count（默认10）

对于大负载的put（达到了M范围）或是大范围的Scan操作，handler数目不易过大，易造成OOM。 对于小负载的put或是get，delete等操作，handler数要适当调大。根据上面的原则，要看我们的业务的情况来设置。（具体情况具体分析）。

（3）HBASE\_HEAPSIZE（hbase-env.sh中配置）

我的前两篇文章Memstoresize40%（默认） blockcache 20%（默认）就是依据这个而成的，总体HBase内存配置。设到机器内存的1/2即可。

（4）选择使用压缩算法，目前HBase默认支持的压缩算法包括GZ，LZO以及snappy（hbase-site.xml中配置）

（5）hbase.hregion.max.filesize默认256M

上面说过了，hbase自动拆分region的阈值，可以设大或者无限大，无限大需要手动拆分region，懒的人别这样。

（6）hbase.hregion.memstore.flush.size

单个region内所有的memstore大小总和超过指定值时，flush该region的所有memstore。

（7）hbase.hstore.blockingStoreFiles  默认值：7

说明：在flush时，当一个region中的Store（CoulmnFamily）内有超过7个storefile时，则block所有的写请求进行compaction，以减少storefile数量。

调优：block写请求会严重影响当前regionServer的响应时间，但过多的storefile也会影响读性能。从实际应用来看，为了获取较平滑的响应时间，可将值设为无限大。如果能容忍响应时间出现较大的波峰波谷，那么默认或根据自身场景调整即可。

（8）hbase.hregion.memstore.block.multiplier默认值：2

说明：当一个region里总的memstore占用内存大小超过hbase.hregion.memstore.flush.size两倍的大小时，block该region的所有请求，进行flush，释放内存。

虽然我们设置了region所占用的memstores总内存大小，比如64M，但想象一下，在最后63.9M的时候，我Put了一个200M的数据，此时memstore的大小会瞬间暴涨到超过预期的hbase.hregion.memstore.flush.size的几倍。这个参数的作用是当memstore的大小增至超过hbase.hregion.memstore.flush.size 2倍时，block所有请求，遏制风险进一步扩大。

调优： 这个参数的默认值还是比较靠谱的。如果你预估你的正常应用场景（不包括异常）不会出现突发写或写的量可控，那么保持默认值即可。如果正常情况下，你的写请求量就会经常暴长到正常的几倍，那么你应该调大这个倍数并调整其他参数值，比如hfile.block.cache.size和hbase.regionserver.global.memstore.upperLimit/lowerLimit，以预留更多内存，防止HBase server OOM。

（9）hbase.regionserver.global.memstore.upperLimit：默认40%

当ReigonServer内所有region的memstores所占用内存总和达到heap的40%时，HBase会强制block所有的更新并flush这些region以释放所有memstore占用的内存。

hbase.regionserver.global.memstore.lowerLimit：默认35%

同upperLimit，只不过lowerLimit在所有region的memstores所占用内存达到Heap的35%时，不flush所有的memstore。它会找一个memstore内存占用最大的region，做个别flush，此时写更新还是会被block。lowerLimit算是一个在所有region强制flush导致性能降低前的补救措施。在日志中，表现为 “\*\* Flushthread woke up with memory above low water.”。

调优：这是一个Heap内存保护参数，默认值已经能适用大多数场景。

（10）hfile.block.cache.size：默认20%

  这是涉及hbase读取文件的主要配置，BlockCache主要提供给读使用。读请求先到memstore中查数据，查不到就到blockcache中查，再查不到就会到磁盘上读，并把读的结果放入blockcache。由于blockcache是一个LRU,因此blockcache达到上限(heapsize \* hfile.block.cache.size)后，会启动淘汰机制，淘汰掉最老的一批数据。对于注重读响应时间的系统，应该将blockcache设大些，比如设置blockcache=0.4，memstore=0.39，这会加大缓存命中率。

（11）hbase.regionserver.hlog.blocksize和hbase.regionserver.maxlogs

之所以把这两个值放在一起，是因为WAL的最大值由hbase.regionserver.maxlogs\*hbase.regionserver.hlog.blocksize (2GB by default)决定。一旦达到这个值，Memstore flush就会被触发。所以，当你增加Memstore的大小以及调整其他的Memstore的设置项时，你也需要去调整HLog的配置项。否则，WAL的大小限制可能会首先被触发，因而，你将利用不到其他专门为Memstore而设计的优化。抛开这些不说，通过WAL限制来触发Memstore的flush并非最佳方式，这样做可能会会一次flush很多Region，尽管“写数据”是很好的分布于整个集群，进而很有可能会引发flush“大风暴”。

提示：最好将hbase.regionserver.hlog.blocksize\* hbase.regionserver.maxlogs 设置为稍微大于hbase.regionserver.global.memstore.lowerLimit\* HBASE\_HEAPSIZE。

### **HDFS优化部分**

**HBase是基于hdfs文件系统的一个数据库，其数据最终是写到hdfs中的，因此涉及hdfs调优的部分也是必不可少的。**

（1）dfs.replication.interval:默认3秒

可以调高，避免hdfs频繁备份，从而提高吞吐率。

（2）dfs.datanode.handler.count:默认为10

可以调高这个处理线程数，使得写数据更快

（3）dfs.namenode.handler.count：默认为8

（4）dfs.datanode.socket.write.timeout：默认480秒，并发写数据量大的时候可以调高一些，否则会出现我另外一篇博客介绍的的错误。

1. dfs.socket.timeout:最好也要调高，默认的很小。

## 大数据Hbase 面试题

#### hbase 的特点是什么

1. Hbase一个分布式的基于列式存储的数据库,基于Hadoop的hdfs存储，zookeeper进行管理。
2. Hbase适合存储半结构化或非结构化数据，对于数据结构字段不够确定或者杂乱无章很难按一个概念去抽取的数据。
3. Hbase为null的记录不会被存储.
4. 基于的表包含rowkey，时间戳，和列族。新写入数据时，时间戳更新，同时可以查询到以前的版本.
5. hbase是主从架构。hmaster作为主节点，hregionserver作为从节点

#### hbase如何导入数据

使用MapReduce Job方式，根据Hbase API编写java脚本，将文本文件用文件流的方式截取，然后存储到多个字符串数组中，在put方法下，通过对表中的列族进行for循环遍历列名，用if判断列名后进行for循环调用put.add的方法对列族下每一个列进行设值，每个列族下有几个了就赋值几次！没有表先对先创建表。

#### hbase的存储结构？

答：Hbase中的每张表都通过行键(rowkey)按照一定的范围被分割成多个子表（HRegion），默认一个HRegion超过256M就要被分割成两个，由HRegionServer管理，管理哪些HRegion由Hmaster分配。 HRegion存取一个子表时，会创建一个HRegion对象，然后对表的每个列族（Column Family）创建一个store实例，每个store都会有0个或多个StoreFile与之对应，每个StoreFile都会对应一个HFile，HFile就是实际的存储文件，因此，一个HRegion还拥有一个MemStore实例。

#### Hbase和hive 有什么区别

**问题导读：**

**hive与hbase的底层存储是什么？**

**hive是产生的原因是什么？**  
**habase是为了弥补hadoop的什么缺陷**

**答案:**

**共同点：**

1.hbase与hive都是架构在hadoop之上的。都是用hadoop作为底层存储

**区别：**

2.Hive是建立在Hadoop之上为了减少MapReducejobs编写工作的批处理系统，HBase是为了支持弥补Hadoop对实时操作的缺陷的项目 。

3.想象你在操作RMDB数据库，如果是全表扫描，就用Hive+Hadoop,如果是索引访问，就用HBase+Hadoop 。

4.Hive query就是MapReduce jobs可以从5分钟到数小时不止，

HBase是非常高效的，肯定比Hive高效的多。

5.Hive本身不存储和计算数据，它完全依赖于HDFS和MapReduce，Hive中的表纯逻辑。

6.hive借用hadoop的MapReduce来完成一些hive中的命令的执行

7.hbase是物理表，不是逻辑表，提供一个超大的内存hash表，搜索引擎通过它来存储索引，方便查询操作。

8.hbase是列存储。

9.hdfs作为底层存储，hdfs是存放文件的系统，而Hbase负责组织文件。

10.hive需要用到hdfs存储文件，需要用到MapReduce计算框架。

#### 解释下hbase实时查询的原理

**答：实时查询，可以认为是从内存中查询，一般响应时间在1秒内。HBase的机制是数据先写入到内存中，当数据量达到一定的量（如128M），再写入磁盘中， 在内存中，是不进行数据的更新或合并操作的，只增加数据，这使得用户的写操作只要进入内存中就可以立即返回，保证了HBase I/O的高性能。**

#### 描述Hbase的rowKey的设计原则

**问题导读：联系region 和rowkey 关系说明,设计可参考以下三个原则.**

1 rowkey长度原则

rowkey是一个二进制码流，可以是任意字符串，最大长度64kb，实际应用中一般为10-100bytes，以byte[]形式保存，一般设计成定长。建议越短越好，不要超过16个字节, 原因如下：

数据的持久化文件HFile中是按照KeyValue存储的，如果rowkey过长会极大影响HFile的存储效率

MemStore将缓存部分数据到内存，如果rowkey字段过长，内存的有效利用率就会降低，系统不能缓存更多的数据，这样会降低检索效率

2 rowkey散列原则

如果rowkey按照时间戳的方式递增，不要将时间放在二进制码的前面，建议将rowkey的高位作为散列字段，由程序随机生成，低位放时间字段，这样将提高数据均衡分布在每个RegionServer，以实现负载均衡的几率。如果没有散列字段，首字段直接是时间信息，所有的数据都会集中在一个RegionServer上，这样在数据检索的时候负载会集中在个别的RegionServer上，造成热点问题，会降低查询效率。

3 rowkey唯一原则

必须在设计上保证其唯一性，rowkey是按照字典顺序排序存储的，因此，设计rowkey的时候，要充分利用这个排序的特点，将经常读取的数据存储到一块，将最近可能会被访问的数据放到一块。

#### 列簇怎么创建比较好？ (<=2)

答：rowKey最好要创建有规则的rowKey，即最好是有序的。HBase中一张表最好只创建一到两个列族比较好，因为HBase不能很好的处理多个列族。

#### 描述Hbase中scan和get的功能以及实现的异同.

1.按指定RowKey 获取唯一一条记录，get方法（org.apache.hadoop.hbase.client.Get）Get 的方法处理分两种 : 设置了ClosestRowBefore 和没有设置的rowlock .主要是用来保证行的事务性，即每个get 是以一个row 来标记的.一个row中可以有很多family 和column.

2.按指定的条件获取一批记录，scan方法(org.apache.Hadoop.hbase.client.Scan)实现条件查询功能使用的就是scan 方式.

1)scan 可以通过setCaching 与setBatch 方法提高速度(以空间换时间)； 2)scan 可以通过setStartRow 与setEndRow 来限定范围([start，end]start 是闭区间，end 是开区间)。范围越小，性能越高。

3)scan 可以通过setFilter 方法添加过滤器，这也是分页、多条件查询的基础。 3.全表扫描，即直接扫描整张表中所有行记录

#### 请详细描述Hbase中一个Cell 的结构

HBase 中通过row 和columns 确定的为一个存贮单元称为cell。

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

#### 请描述Hbase中scan对象的setCache和setBatch 方法的使用.

<https://www.cnblogs.com/editice/archive/2013/04/22/3035728.html>

cache是面向行的优化处理，batch是面向列的优化处理

#### 简述 HBASE中compact用途是什么，什么时候触发，分为哪两种,有什么区别，有哪些相关配置参数？

在hbase中每当有memstore数据flush到磁盘之后，就形成一个storefile，当storeFile的数量达到一定程度后，就需要将 storefile 文件来进行 compaction 操作。

Compact 的作用：

1>.合并文件

2>.清除过期，多余版本的数据

3>.提高读写数据的效率

HBase 中实现了两种 compaction 的方式：minor and major. 这两种 compaction 方式的区别是：

1、Minor 操作只用来做部分文件的合并操作以及包括 minVersion=0 并且设置 ttl 的过

期版本清理，不做任何删除数据、多版本数据的清理工作。

2、Major 操作是对 Region 下的HStore下的所有StoreFile执行合并操作，最终的结果是整理合并出一个文件。

#### 简述Hbase filter的实现原理是什么？结合实际项目经验，写出几个使用filter的场景

HBase为筛选数据提供了一组过滤器，通过这个过滤器可以在HBase中的数据的多个维度（行，列，数据版本）上进行对数据的筛选操作，也就是说过滤器最终能够筛选的数据能够细化到具体的一个存储单元格上（由行键，列名，时间戳定位）。RowFilter、PrefixFilter。。。

hbase的filter是通过scan设置的，所以是基于scan的查询结果进行过滤.

过滤器的类型很多，但是可以分为两大类——比较过滤器，专用过滤器

过滤器的作用是在服务端判断数据是否满足条件，然后只将满足条件的数据返回给客户端；

如在进行订单开发的时候，我们使用rowkeyfilter过滤出某个用户的所有订单

#### Hbase内部是什么机制

在HBase 中无论是增加新行还是修改已有的行，其内部流程都是相同的。HBase 接到命令后存下变化信息，或者写入失败抛出异常。默认情况下，执行写入时会写到两个地方：预写式日志（write-ahead log，也称HLog）和MemStore（见图2-1）。HBase 的默认方式是把写入动作记录在这两个地方，以保证数据持久化。只有当这两个地方的变化信息都写入并确认后，才认为写动作完成。

MemStore 是内存里的写入缓冲区，HBase 中数据在永久写入硬盘之前在这里累积。当MemStore 填满后，其中的数据会刷写到硬盘，生成一个HFile。HFile 是HBase 使用的底层存储格式。HFile 对应于列族，一个列族可以有多个HFile，但一个HFile 不能存储多个列族的数据。在集群的每个节点上，每个列族有一个MemStore。

大型分布式系统中硬件故障很常见，HBase 也不例外。设想一下，如果MemStore还没有刷写，服务器就崩溃了，内存中没有写入硬盘的数据就会丢失。HBase 的应对办法是在写动作完成之前先写入WAL。HBase 集群中每台服务器维护一个WAL 来记录发生的变化。WAL 是底层文件系统上的一个文件。直到WAL 新记录成功写入后，写动作才被认为成功完成。这可以保证HBase 和支撑它的文件系统满足持久性。大多数情况下，HBase 使用Hadoop 分布式文件系统（HDFS）来作为底层文件系统。

如果HBase 服务器宕机，没有从MemStore 里刷写到HFile 的数据将可以通过回放WAL 来恢复。你不需要手工执行。Hbase 的内部机制中有恢复流程部分来处理。每台HBase 服务器有一个WAL，这台服务器上的所有表（和它们的列族）共享这个WAL。

你可能想到，写入时跳过WAL 应该会提升写性能。但我们不建议禁用WAL，除非你愿意在出问题时丢失数据。如果你想[**测试**](http://lib.csdn.net/base/softwaretest)一下，如下代码可以禁用WAL： 注意：不写入 WAL 会在RegionServer 故障时增加丢失数据的风险。关闭WAL，出现故障时HBase 可能无法恢复数据，没有刷写到硬盘的所有写入数据都会丢失。

#### HBase宕机如何处理

答：宕机分为HMaster宕机和HRegisoner宕机，如果是HRegisoner宕机，HMaster会将其所管理的region重新分布到其他活动的RegionServer上，由于数据和日志都持久在HDFS中，该操作不会导致数据丢失。所以数据的一致性和安全性是有保障的。

如果是HMaster宕机，HMaster没有单点问题，HBase中可以启动多个HMaster，通过Zookeeper的Master Election机制保证总有一个Master运行。即ZooKeeper会保证总会有一个HMaster在对外提供服务。

**补充 --- 知识点深度剖析(可选看):**

1.导致Hbase挂掉的场景

<https://blog.csdn.net/zlfprogram/article/details/74066585> 2.深入理解HBase的memestore、storeFile(HFile)

<https://blog.csdn.net/xiaoshunzi111/article/details/69844526>

3.HBase-7.hbase查询多版本数据&过滤器原则&批量导入Hbase&hbase预分区

<https://blog.csdn.net/shenfuli/article/details/50589496>

补充: 布隆过滤可以每列族单独启用。

使用 HColumnDescriptor.setBloomFilterType(NONE | ROW | ROWCOL) 对列族

启用布隆过滤可以节省读磁盘过程，可以有助于降低读取延迟

4. HBase原理－数据读取流程解析+HBase最佳实践－写性能优化策略

<http://hbasefly.com/2016/12/21/hbase-getorscan/>

<http://hbasefly.com/2016/12/10/hbase-parctice-write/>

5. HBase原理 – 所有Region切分的细节都在这里了

<http://hbasefly.com/2017/08/27/hbase-split/>

6. HBase Compaction的前生今世

<http://hbasefly.com/2016/07/13/hbase-compaction-1/>