HBASE

目录

[1. HBase 2](#_Toc25124)

[1.1. HBase概述 2](#_Toc23491)

[1.2. Hbase的特点 3](#_Toc12927)

[1.3. HBase的应用场景 4](#_Toc20186)

[1.4. 非关系型数据库和关系型数据库的区别 6](#_Toc8001)

[1.4.1. 传统关系型数据库的缺陷 6](#_Toc2909)

[1.4.2. NoSQL数据库的区别 7](#_Toc15332)

[2. HBase的行存储与列存储 9](#_Toc28778)

[2.1. 行存储与列存储在数据读写的特点 10](#_Toc2907)

[2.1.1. 数据写入上的对比 10](#_Toc20155)

[2.1.2. 在数据读取上的对比 10](#_Toc21491)

[2.2. 行存储和列存储的优缺点 12](#_Toc29296)

[2.3. 行存储和列存储的使用场景 12](#_Toc30716)

[3. HBase的安装 15](#_Toc16893)

[3.1. Hbase的单机模式的安装 15](#_Toc6908)

[3.1.1. Hbase单机模式的特点 15](#_Toc22061)

[3.1.2. HBase的单机模式的安装 15](#_Toc20611)

[3.1.3. Hbase基础指令 18](#_Toc8821)

[3.2. HBase完全分布式的搭建 24](#_Toc23475)

[3.2.1. HBase的完全分布式的特点 24](#_Toc3307)

[3.2.2. Hbase完全分布式的安装 25](#_Toc25567)

[4. HBase的模式设计 35](#_Toc23265)

[4.1. Hbase的行键 35](#_Toc3886)

[4.2. HBase的列族 36](#_Toc16248)

[4.3. HBase的表设计 37](#_Toc15220)

[4.3.1. Hbase的行键设计 37](#_Toc934)

[4.3.2. HBase的列族设计 42](#_Toc3730)

[5. HBase API 43](#_Toc15229)

[5.1. API概述 43](#_Toc8990)

[5.2. API处理操作 43](#_Toc23233)

[5.3. 过滤器的概述 52](#_Toc31536)

[5.4. 过滤器的处理操作 53](#_Toc19211)

[6. Hbase的读写流程 57](#_Toc29299)

[6.1. HBase的写流程 57](#_Toc30721)

[6.2. Hbase的读流程 64](#_Toc29172)

[7. HBase的扩展内容 68](#_Toc10871)

[7.1. HBase的优化 68](#_Toc31561)

[7.2. Hbase的调优 70](#_Toc25630)

# 

# HBase

## HBase概述



HBase是一个分布式的、面向列的非关系型数据库，该技术来源于 Fay Chang 所撰写的Google论文《Bigtable》一个结构化数据的分布式存储系统"。就像Bigtable利用了Google文件系统(File System)所提供的分布式数据存储一样，HBase在Hadoop之上提供了类似于Bigtable的能力（低延迟的数据查询能力）。HBase是Apache的Hadoop项目的子项目。HBase不同于一般的关系数据库，HBase同BigTable一样，都是NoSQL数据库，即非关系型数据库。HBase是Apache Hadoop生态系统中的重要一员，主要用于海量结构化数据存储；从逻辑上讲，HBase将数据按照表、行和列进行存储。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的特点

1.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杂乱无章非常难按一个概念去进行抽取的数据适合用HBase，因为HBase支持动态添加列。

2.记录很稀疏：

RDBMS的行有多少列是固定的。为null的列浪费了存储空间。HBase为null的Column不会被存储，这样既节省了空间又提高了读性能。

3.多版本号数据：

依据Row key和Column key定位到的Value能够有随意数量的版本号值，因此对于须要存储变动历史记录的数据，用HBase是很方便的。比方某个用户的Address变更，用户的Address变更记录也许也是具有研究意义的。

4.仅要求最终一致性：

对于数据存储事务的要求不像金融行业和财务系统这么高，只要保证最终一致性就行。（比如HBase+elasticsearch时，可能出现数据不一致）

5.高可用和海量数据以及很大的瞬间写入量：

WAL解决高可用，支持PB级数据，put性能高。适用于插入比查询操作更频繁的情况。比如，对于历史记录表和日志文件。（HBase的写操作更加高效）

6.业务场景简单：

不需要太多的关系型数据库特性，列入交叉列，交叉表，事务，连接等。

单一RowKey固有的局限性决定了它不可能有效地支持多条件查询。不适合于大范围扫描查询。不直接支持 SQL 的语句查询。

## HBase的应用场景

HBase是一个分布式存储、数据库引擎，可以支持千万的QPS、PB级别的存储，这些都已经在生产环境验证，并且在广大的公司已经验证。特别是阿里、小米、京东、滴滴内部都有数千、上万台的HBase集群。选择一个技术的首要条件是对齐大公司，大公司会投入大量的人力去维护、改进、贡献社区。

HBase可以说是一个数据库，也可以说是一个存储。拥有双重属性的HBase天生就具备广阔的应用场景。在最近的一些版本中，引入了OffHeap降低gc影响，优化链路延迟，提供Replica等可以满足在线的需求。引入MOB，可以存储3M左右的对象，完全适应了对象存储。另外由于自身的并发能力、存储能力，可以说是具有最为竞争力的引擎。下面介绍一下HBase的应用场景，如图1-1所示。

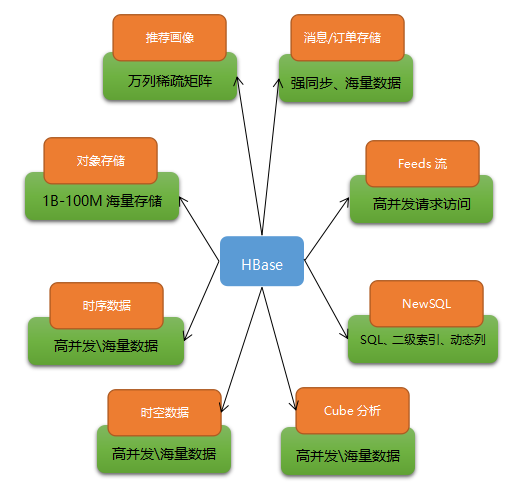


图1-1

 1、对象存储

比如一些app的海量的图片、网页、新闻等对象，可以存储在HBase中，有些病毒公司的病毒库也可以存储在HBase中。

2、时序数据

时序数据就是分布在时间上的一系列数值。HBase之上有OpenTSDB模块，可以满足时序类场景的需求。比如我们有很多的设备、传感器，产生很多数据，如果规模不是特别大的厂家有几千个风机，每个风机有几百个指标，那么就会有一百万左右的时序数据，如果用采样每一秒会产生一百万个时间点，如果用传统数据库，那么每一秒会产生一百万次，持续地往MQ做一百万次，它会崩裂。并且查询也是个大问题，除了多维查询以外，我们还会额外地增加时间纬度，查看一段时间的数据。这时候HBase很好了满足了时序类场景的需求。

3、时空数据

主要是轨迹、气象网格之类，比如滴滴打车的轨迹数据主要存在HBase之中，另外大数据量的车联网企业，数据也都是存在HBase中。比如互联网出行，智慧物流与外卖递送，传感网与实时GIS等场景。

4.CubeDB OLAP

Kylin一个cube分析工具，底层的数据就是存储在HBase之中，不少客户自己基于离线计算构建cube存储在HBase之中，满足在线报表查询的需求

5、推荐画像

特别是用户的画像，是一个比较大的稀疏矩阵，蚂蚁的风控就是构建在HBase上。用户画像有用户数据量大，用户标签多，标签统计维度不确定等特点，适合HBase特性的发挥。

6、消息/订单

在电信领域、银行领域，不少的订单查询底层的存储，另外不少通信、消息同步的应用构建在HBase之上。

7、Feed流

Feed流，是RSS中用来接收该信息来源更新的接口，简单的说就是持续更新并呈现给用户的内容。比如微信朋友圈中看到的好友的一条条状态，微博看到的你关注的人更新的内容，App收到的一篇篇新文章的推送，都算是feed流。

8、NewSQL

HBase上有Phoenix的插件，可以满足二级索引、SQL的需求，对接传统数据需要SQL非事务的需求。从NoSQL到NewSQL，Phoenix或许是新的趋势。

## 非关系型数据库和关系型数据库的区别

### 传统关系型数据库的缺陷

1）高并发读写的瓶颈

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

遍的业务需求。

2）可扩展性的限制

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

3）事务一致性的负面影响

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

4）复杂SQL查询的弱化

任何大数据量的Web系统都非常忌讳几个大表间的关联查询，以及复杂的数据分析类型的SQL查询，特别是SNS类型的网站，从需求以及产品设计角度就避免了这种情况的产生。更多的情况往往只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大地弱化了，所以这部分功能不能得到充分发挥。

### NoSQL数据库的区别

1）扩展性强

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

1. 并发性能好

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

3）数据模型灵活

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

**总结**

HBase作为NoSQL数据库的一种，当然也具备上面提到的种种优势。Hadoop最适合的应用场景是离线批量处理数据，其离线分析的效率非常高，Hadoop针对数据的吞吐量做了大量优化，能在分钟级别处理TB级的数据，但是Hadoop不能做到低延迟的数据访问，所以一般的应用系统并不适合批量模式访问，更多的还是用户的随机访问，就类似访问关系型数据库中的某条记录一样。比如Google这个搜索引擎，存储了海量的网页数据，当我们通过搜索引擎检索一个网页时，之所以Google能够快速的响应结果，核心的技术就是利用了BigTable，可以实现低延迟的数据访问。HBase的列式存储的特性支撑它实时随机读取、基于KEY的特殊访问需求。当然，HBase还有不少新特性，其中不乏有趣的特性，在接下来的内容中将会详细介绍。

# HBase的行存储与列存储

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

列式存储(column-based)是相对于传统关系型数据库的行式存储(Row-basedstorage)来说的。简单来说两者的区别就是如何组织表，如图2-1所示：

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

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

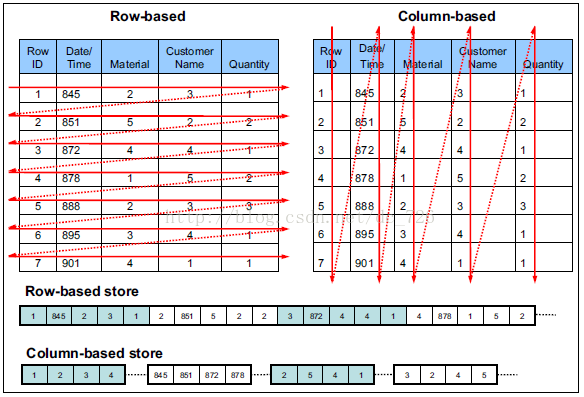


图2-1

从上图可以很清楚地看到，行式存储下一张表的数据都是放在一起的，但列式存储下都被分开保存了。

## 行存储与列存储在数据读写的特点

### 数据写入上的对比

1）行存储的写入是一次完成。如果这种写入建立在操作系统的文件系统上，可以保证写入过程的成功或者失败，数据的完整性因此可以确定。

2）列存储由于需要把一行记录拆分成单列保存，写入次数明显比行存储多（意味着磁头调度次数多，而磁头调度是需要时间的，一般在1ms~3ms)，再加上磁头需要在盘片上移动和定位花费的时间，实际时间消耗会更大。所以，行存储在写入上占有很大的优势。

3）还有数据修改,这实际也是一次写入过程。不同的是，数据修改是对磁盘上的记录做删除标记。行存储是在指定位置写入一次，列存储是将磁盘定位到多个列上分别写入，这个过程仍是行存储的列数倍。所以，数据修改也是以行存储占优。

### 在数据读取上的对比

1）数据读取时，行存储通常将一行数据完全读出，如果只需要其中几列数据的情况，就会存在冗余列，出于缩短处理时间的考量，消除冗余列的过程通常是在内存中进行的。

2）列存储每次读取的数据是集合的一段或者全部，不存在冗余性问题。

3）两种存储的数据分布。由于列存储的每一列数据类型是同质的，不存在二义性问题。比如说某列数据类型为整型(int)，那么它的数据集合一定是整型数据。这种情况使数据解析变得十分容易。相比之下，行存储则要复杂得多，因为在一行记录中保存了多种类型的数据，数据解析需要在多种数据类型之间频繁转换，这个操作很消耗CPU，增加了解析的时间。所以，列存储的解析过程更有利于分析大数据。

4）从数据的压缩以及更性能的读取来对比‘’；

下面才是那张表本来的样子。经过字典表进行数据压缩后，表中的字符串才都变成数字了。正因为每个字符串在字典表里只出现一次了，所以达到了压缩的目的，如图2-2，图2-3所示。

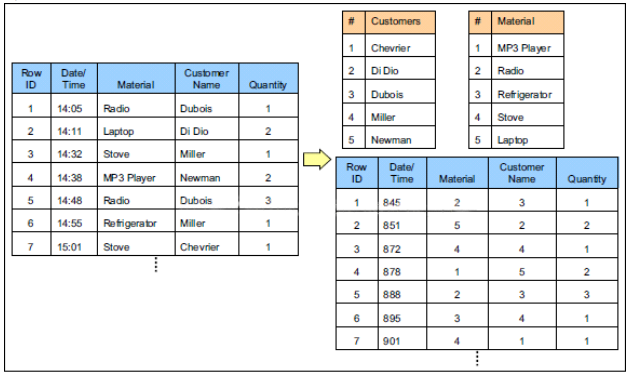


图2-2

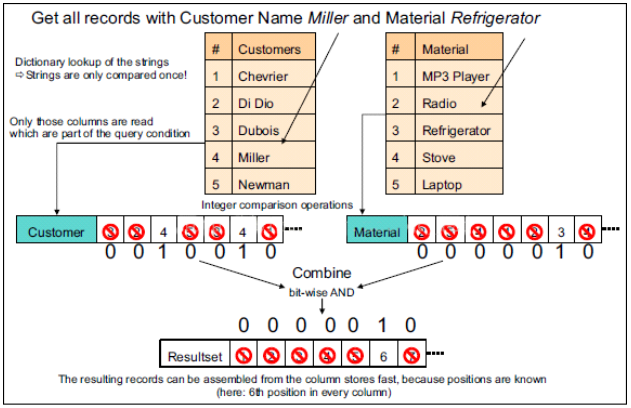


图2-3

关键步骤如下：

1.     去字典表里找到字符串对应数字(只进行一次字符串比较)。

2.     用数字去列表里匹配，匹配上的位置设为1。

3.     把不同列的匹配结果进行位运算得到符合所有条件的记录下标。

4.     使用这个下标组装出最终的结果集。

## 行存储和列存储的优缺点

1）行存储的写入是一次性完成，消耗的时间比列存储少，并且能够保证数据的完整性，缺点是数据读取过程中会产生冗余数据，如果只有少量数据，此影响可以忽略;数量大可能会影响到数据的处理效率。

2）列存储在写入效率、保证数据完整性上都不如行存储，它的优势是在读取过程，不会产生冗余数据，这对数据完整性要求不高的大数据处理领域，比如互联网，犹为重要。

## 行存储和列存储的使用场景

1）一般来说，一个OLAP类型的查询可能需要访问几百万甚至几十亿个数据行，且该查询往往只关心少数几个数据列。例如，查询今年销量最高的前20个商品，这个查询只关心三个数据列：时间（date）、商品（item）以及销售量（sales amount）。商品的其他数据列，例如商品URL、商品描述、商品所属店铺，等等，对这个查询都是没有意义的。而列式数据库只需要读取存储着“时间、商品、销量”的数据列，而行式数据库需要读取所有的数据列。因此，列式数据库大大地提高了OLAP大数据量查询的效率。

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

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

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

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

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

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

如下图所示

“男”对应的位图为3031，表示第1、4、6行值为“男”

“女”对应的位图为0133，表示第2、3、5行值为“女”

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

|  |  |
| --- | --- |
| 行号 | 性别 |
| 1 | 男   |  | | --- | | “男”：3031 | | “女”：0133 | |
| 2 | 女 |
| 3 | 女 |
| 4 | 男 |
| 5 | 女 |
| 6 | 男 |

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

最后总结如下

传统行式数据库的特性如下：

①数据是按行存储的。

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

③建立索引和物化视图需要花费大量的时间和资源。

④面对查询需求，数据库必须被大量膨胀才能满足需求。

列式数据库的特性如下：

①数据按列存储，即每一列单独存放。

②数据即索引。

③只访问查询涉及的列，可以大量降低系统I/O。

④每一列由一个线程来处理，即查询的并发处理性能高。

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

# HBase的安装

HBase有两种运行模式：单机模式和分布式模式。无论启动哪种模式，都必须要编辑conf/HBase-env.sh文件以指定运行HBase的java安装目录，只需要设置JAVA\_HOME参数指向Java安装的根目录。在这个文件中，用户还能设置HBase的环境变量，如堆大小和其他一些JVM参数，以及日志的输出目录。

## Hbase的单机模式的安装

### Hbase单机模式的特点

单机模式是默认模式，在单机模式中，HBase并不使用Hadoop的HDFS，仅使用本地文件系统，配置完既可使用，好处是便于测试。坏处是不具备分布式存储数据的能力。

### HBase的单机模式的安装

1.安装JDK及配置环境变量

执行命令如下：

vi /etc/profile

1）在尾行添加如下内容，然后保存退出

#set java environment

JAVA\_HOME=/usr/local/src/java/jdk1.7.0\_51

PATH=$JAVA\_HOME/bin:$PATH

CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar

export JAVA\_HOME PATH CLASSPATH

2）执行命令如下：

source /etc/profile

使更改的配置立即生效。

3）执行命令如下：

java -version

查看JDK版本信息。如显示1.7.0证明成功。

1. 上传解压HBase安装包

利用FZ软件将HBase的安装包上传到/root/usr/soft目录下，选中HBase的安装包右击点击上传，如图3-1所示。



图3-1

解压HBase的安装包，执行命令tar -zxvf HBase-0.98.17-hadoop2-bin.tar.gz 。如图3-2所示。



图3-2

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

配置示例：

<property>

<name>HBase.rootdir</name>

<value>file:///home/software/HBase/tmp</value>

</property>

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

4.启动HBase,进入bin目录，执行：sh start-HBase.sh，启动HBase。然后可以通过jps查看是否有HMaster进程，如果有，证明HBase启动成功。如图3-3所示

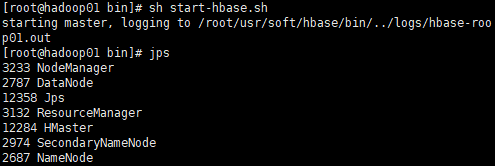


图3-3

5.在bin目录执行：./HBase shell 进入shell客户端操作HBase。如图3-4所示。

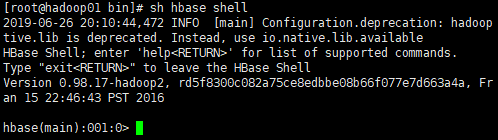


图3-4

6.解压所有文件之后，用户可以了解到项目目录中都包含哪些文件

执行命令：ls -lr 查看目录结构如下：

drwxr-xr-x. 8 root root 4096 Jun 30 19:15 tmp

-rw-r--r--. 1 300 300 1377 Jan 15 2016 README.txt

-rw-r--r--. 1 300 300 23149 Jan 15 2016 NOTICE.txt

drwxr-xr-x. 2 root root 4096 Jul 1 01:5 logs

-rw-r--r--. 1 300 300 57267 Jan 15 2016 LICENSE.txt

drwxr-xr-x. 3 root root 4096 Jun 4 06:58 lib

-rw-r--r--. 1 300 300 261 Jan 15 2016 LEGAL

drwxr-xr-x. 7 300 300 4096 Jan 15 2016 HBase-webapps

drwxr-xr-x. 5 300 300 4096 Jan 15 2016 docs

drwxr-xr-x. 2 300 300 4096 Jul 1 00:43 conf

-rw-r--r--. 1 300 300 209002 Jan 15 2016 CHANGES.txt

drwxr-xr-x. 4 300 300 4096 Jan 15 2016 bin

根目录包含了一些文本文件，CHANGES.txt文件是变更日志的静态快照页面，它包含了当前下载版本中所有的变更记录。

bin：即二进制文件，此目录包含着HBase的所有脚本，可以完成启动和停止进程。

conf：这个目录是配置目录，包含了定义HBase的配置文件。

docs：这个目录包含了HBase工程网页的副本，以及工具、API和项目自身的文档信息。打开浏览器把docs/index.html文件拖入浏览器，双击该文件使用文件-->打开菜单就可以使用文档。

HBase-webapps：HBase提供了Java实现的Web接口，其中所有的文件都在这个目录下。用户在布置HBase到生产环境中或者使用HBase时，很少有机会接触到这个目录中的文件。

lib：Java应用程序依赖很多类库，这些类库包含了实际的执行程序，并且这些文件都放在lib目录中。

logs：HBase进程通常以守护进程的形式运行，即在操作系统的后台运行在生命周期内它会将一些状态、程度、异常等信息打印到日志文件中。如果HBase已经运行，可以在logs目录的子目录中找到HBase的运行日志文件。

### Hbase基础指令

1.创建一张带有列族的表，t1指表名，c1,c2是列族名

执行语句：create 'tb1','cf1','cf2'

如图3-5所示。

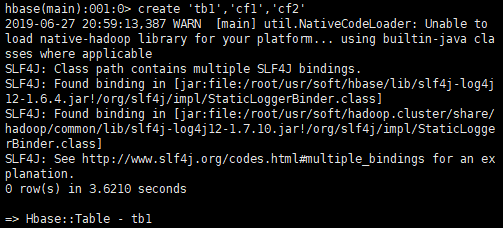


图3-5

2.查看一共有哪些表，执行list，目前只有一张表，如图3-6所示。

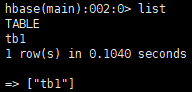


图3-6

3.向表中添加插入数据tb1指表名，row1指行键名，value指单元格值。ts1

指时间戳，一般都省略掉了。注意，行键名在一张表里要全局唯一。

执行命令： put 'tb1','row1','cf1:name','tom'

put 'tb1','row1','cf2:salary','15000'

put 'tb1','row2','cf1:name','tom'

put 'tb1','row2','cf1:age','30'

put 'tb1','row2','cf1:gender','F'

put 'tb1','row2','cf2:salary','3000'

put 'tb1','row2','cf2:addr','bj'

如图3-7、3-8、3-9、3-3、3-11、3-5、3-13所示。



图3-7



图3-8



图3-9



图3-3



图3-11



图3-5



图3-13

4.查看整表的数据，会查看出所有的行数据

执行命令：scan 'tb1'

如图3-14所示。

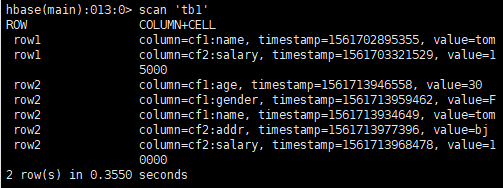


图3-14

5.向HBase中插入行数据时，必须为每一行的数据指定行健，具备唯一性的特点。HBase本质是一种基于Key-Value的存储系统。其中，Key是行健，Value是列族数据的集合。

查看第一行数据，执行语句：get 'tb1','row1'

如图3-15所示。

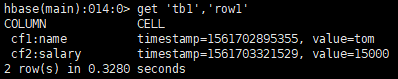


图3-15

查看第一行第一列数据，执行语句：get 'tb1','row1','cf1'

如图3-16所示。

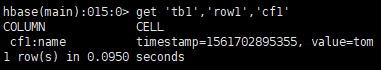


图3-16

查看表tb1的列族，执行语句：scan 'tb1',{COLUMNS=>['cf1']}

如图3-17所示。

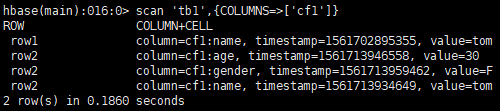


图3-17

查看多个列族，执行语句：scan 'tb1',{COLUMNS=>['cf1']}

如图3-18所示。

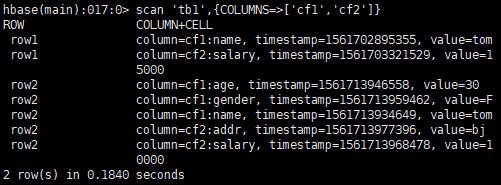


图3-18

历史版本数据：HBase会存储历史版本的数据，可以通过执行语句：

scan 'tb1',{RAW=>TRUE,VERSIONS=>3}

来开启查询。如图3-19所示。

其中VERSIONS=>3表示查看3个版本的数据（包含当前版本+历史版本=3），默认，HBase最多能够存储的版本数=3，版本数大于3时，会被替换，历史数据不易太多，会浪费。

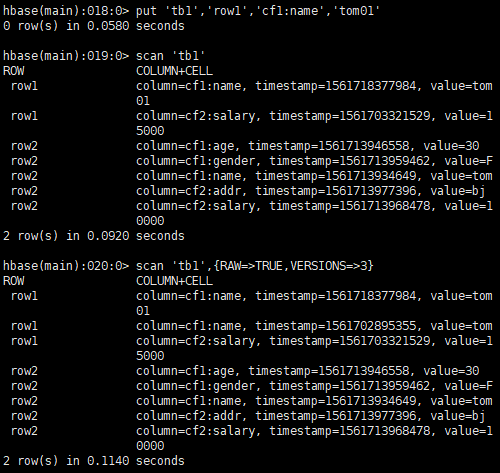


图3-19

先插入row4、然后插入row3，查看表中的数据，row3在前面，由此可知：HBase会按RowKey做字典顺序升序排序。如图3-20所示。

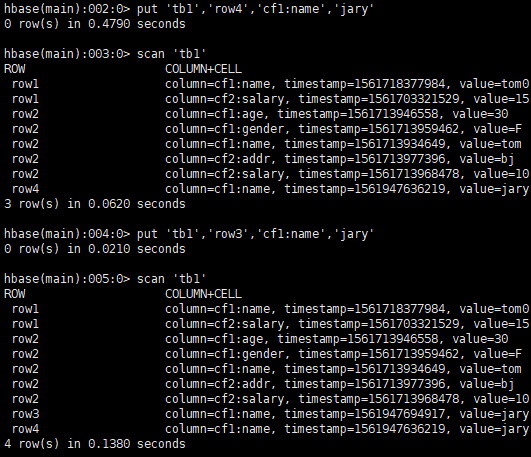


图3-20

根据表名、行键删除整行数据，执行命令：deleteall 'tb1','row4'

如图3-21所示。

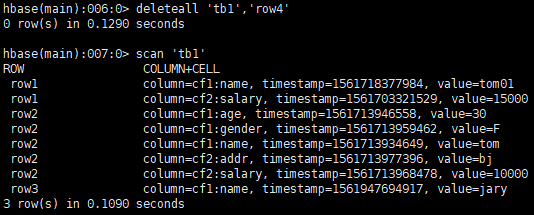


图3-21

删除表，根据表名、行键删除整行数据，首先要禁用表，

先执行命令：disable 'tb2'，

然后执行指令：drop 'tb2'

如图3-22所示。

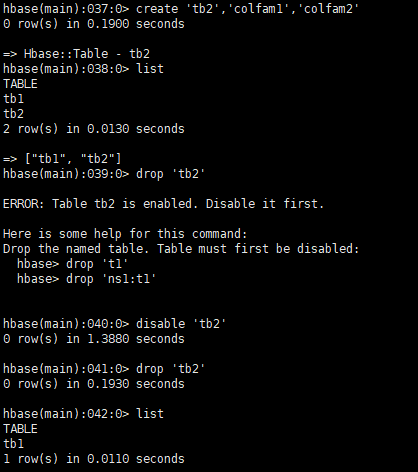


图3-22

## HBase完全分布式的搭建

### HBase的完全分布式的特点

用户需要在多台主机上进行完全分布式的操作，和单机模式相比，HBase单机模式是存在本地，而完全分布式模式是存储在HDFS上。完全分布式模式具备分布式存储的能力。

完全分布式的HBase需要依赖于Zookeeper集群。所有的节点和客户端都需要正常的访问Zookeeper集群(Zookeeper能够以一个单独的节点启动)，用户可以通过启动和关闭脚本就可以吧Zookeeper当做HBase的一部分来启动和关闭进程。HBase的集群模式结构，如图3-23所示。

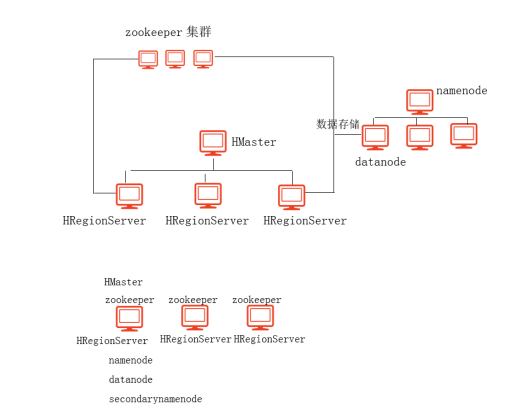


图3-23

### Hbase完全分布式的安装

1.准备三台虚拟机，01作为主节点，02、03作为从节点。

永久关闭每台机器的防火墙

执行命令：service iptables stop

再次执行命令：chkconfig iptables off

如图3-23所示。

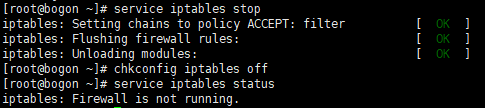


图3-23

2.为每台机器配置主机名以及hosts文件

配置主机名=》执行： vim /etc/sysconfig/network =》然后执行 hostname 主机名=》达到不重启生效目的

配置hosts文件=》执行：vim /etc/hosts

示例：

57.0.0.1 localhost

::1 localhost

192.168.234.21 hadoop01

192.168.234.22 hadoop02

192.168.234.23 hadoop03

3.为每台机器配置ssh免秘钥登录

先配置第一台机器

执行命令：ssh-keygen

然后执行命令：

ssh-copy-id root@hadoop01

ssh-copy-id root@hadoop02

ssh-copy-id root@hadoop03

如图3-24所示。

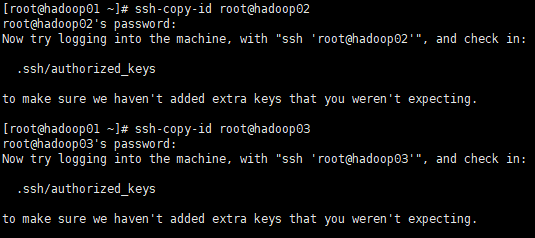


图3-24

配置完第一台机器之后，重复以上步骤，依次配置第二台、第三台机器。

4.01节点上安装和配置：Hadoop+HBase+JDK+Zookeepers.

02、03节点上安装和配置：HBase+JDK+Zookeeper

利用FZ上传HBase安装包到/root/usr/soft的目录下，如图3-25所示。



图3-25

安装Zookeeper

利用FZ将zookeeper的安装包上传到usr/soft的目录下，选中zookeeper右键点击上传，如图3-26所示。



图3-26

解压zookeeper的安装包

执行命令：tar -zxvf zookeeper-3.4.7.tar.gz

如图3-27所示。



图3-27

配置集群模式

①切换到zookeeper安装目录的conf目录，其中有一个zoo\_sample.cfg的配置文件，这个一个配置模板文件，我们需要复制这个文件，并重命名为 zoo.cfg。zoo.cfg才是真正的配置文件。

执行命令：cp zoo\_sample.cfg zoo.cfg

②配置zoo.cfg=》vim zoo.cfg 更改如下几个参数配置：

dataDir。这个参数是存放zookeeper集群环境配置信息的。这个参数默然是配置在/tmp/zookeeper下的 。但是注意，tmp是一个临时文件夹，这个是linux自带的一个目录，是linux本身用于存放临时文件用的目录。但是这个目录极有可能被清空,所以，重要的文件一定不要存在这个目录下。

所以改成：/usr/soft/zookeeper-3.4.7/tmp

注意：这个路径是自定义的，所以目录需要手动创建

在zookeeper-3.4.7下创建tmp目录，

执行命令：mkdir tmp

如图3-28、3-29所示。

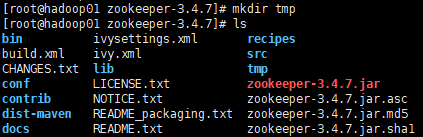


图3-28

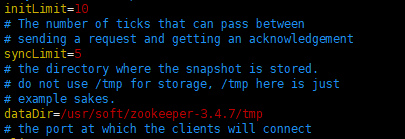


图3-29

clientport。客户端连接服务器的端口，默认是2181，一般不用修改

在配置文件里，需要在加上如下的配置：

server.1=192.168.230.133:2888:3888

server.2=192.168.230.134:2888:3888

server.3=192.168.230.135:2888:3888

①server是关键字，写死

②后面的数字是选举id，在zk集群的选举过程中会用到。

补充：此数字不固定，但是需要注意选举id不能重复，相互之间要能比较大小

然后保存退出

③192.168.168.133:2888:3888

说明：2888原子广播端口，3888选举端口

zookeeper有几个节点，就配置几个server，如图3-30所示。

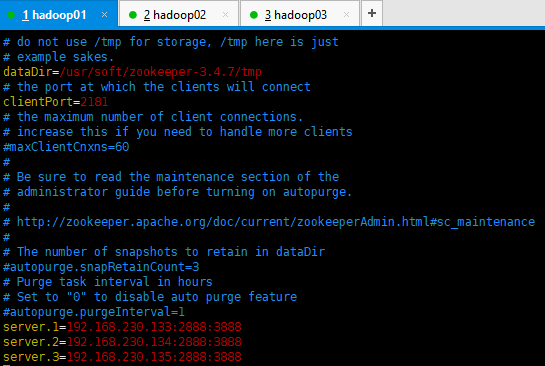


图3-30

③配置文件配置好，需要在dataDir目录下创建一个文件

即在：/root/usr/soft/zookeeper-3.4.7/tmp 目录下，创建 myid

执行命令：vim myid

如图3-31所示。



图3-31

给当前的节点编号。zookeeper节点在启动时，就会到这个目录下去找myid文件，得知自己的编号。如图3-32所示。

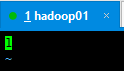


图3-32

:wq保存退出

配置的集群环境发送到其他节点。

scp -r 目录 远程ip地址：存放的路径。

scp -r zookeeper 192.168.230.134: /usr/soft

如图3-33所示。



图3-33

注意：

①更改节点的ip

②更改myid的id号

③关闭防火墙 执行命令：service iptables stop;

启动zookeeper

进入zookeeper的安装目录的bin目录

执行命令：sh zkServer.sh start

如图3-34所示。

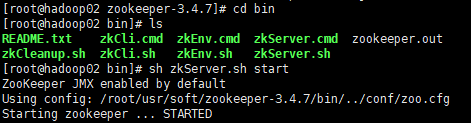


图3-34

然后可以输入jps命令，查看有哪些java进程，如图3-35所示。

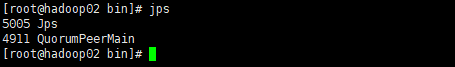


图3-35

执行命令：sh zkServer.sh status

查看当zookeeper的节点状态，如图3-36、3-37、3-38所示。



图3-36



图3-37



图3-38

可以观看到hadoop02为leader，hadoop01、hadoop03为follower

证明zookeeper集群启动成功。

5.修改conf/HBase-env.sh，HBase的环境变量等信息需要在这个文件中设置，Zookeeper和HBase的协调模式也是需要在这个文件中设置的。

执行命令：vim HBase-env.sh

如图3-39所示。



图3-39

配置示例：

#修改JAVA\_HOME

export JAVA\_HOME=/root/usr/soft/jdk1.8.0\_65

#修改Zookeeper和HBase的协调模式，HBase默认使用自带的zookeeper，如果需要使用外部zookeeper，需要先关闭。true表示用HBase内置的zookeeper。false表示用户自己的zookeeper环境。

export HBase\_MANAGES\_ZK=false

6.在Hadoop中，如果用户需要配置增加HDFS的特定配置就要添加到hdfs-site.xml文件中。与此类似的，在HBase中，用户需要增加信息就需要配置添加到conf/HBase-site.xml文件中。修改HBase-site.xml，配置HBase存储数据的目录，开启完全分布式模式，配置Zookeeper集群的地址。

执行命令：vim HBase-site.xml

如图3-40所示



图3-40

配置示例：

<!--指定HBase存储数据的目录，单机模式是存在本地，集群模式是存储在HDFS上-->

<property>

<name>HBase.rootdir</name>

<value>hdfs://hadoop01:9000/HBase</value>

</property>

<!--开启HBase的集群模式-->

<property>

<name>HBase.cluster.distributed</name>

<value>true</value>

</property>

<!--配置Zookeeper集群的地址-->

<property>

<name>HBase.zookeeper.quorum</name>

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

</property>

7.配置region服务器,修改conf/regionservers文件，这个文件罗列了所有region服务器的主机名，它是个纯文本文件，文件中的每一行都是主机名。HBase的运维脚本会一次迭代访问每一行来启动region服务器进程。每个主机名独占一行，HBase启动或关闭时会按照该配置顺序启动或关闭主机中的HBase。

执行命令：vim regionservers

如图3-41所示。



图3-41

配置示例：

hadoop01

hadoop02

hadoop03

8.将01节点配置好的HBase通过远程复制拷贝到02,03节点上。

执行命令：scp -r HBase hadoop02:/root/usr/soft

如图3-42所示。



图3-42

9.启动01,02,03的Zookeeper服务

执行命令：sh zkServer.sh start

1. 启动01节点的Hadoop

执行命令：./start-all.sh

如图3-43所示。

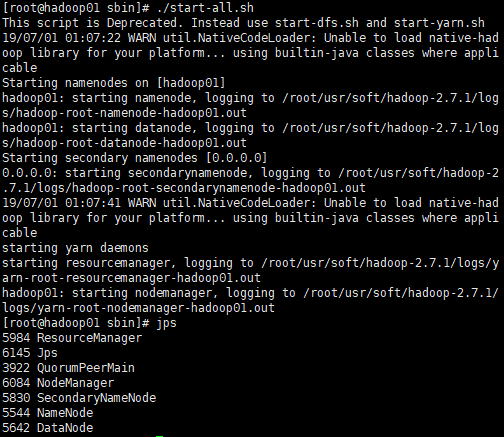


图3-43

11.启动01节点的HBase，进入到HBase安装目录下的bin目录

执行：sh start-HBase.sh

如图3-44所示。

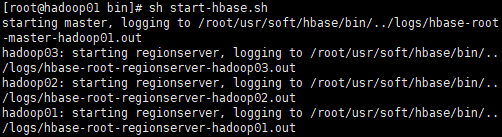


图3-44

5.查看各节点的java进程是否正确

如图3-45、3-46、3-47所示。

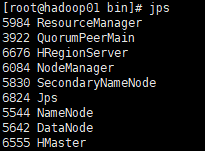


图3-45



图3-46



图3-47

13.通过浏览器访问http://192.168.230.133:6003来访问web界面，通过web见面管理HBase，如图3-48所示。

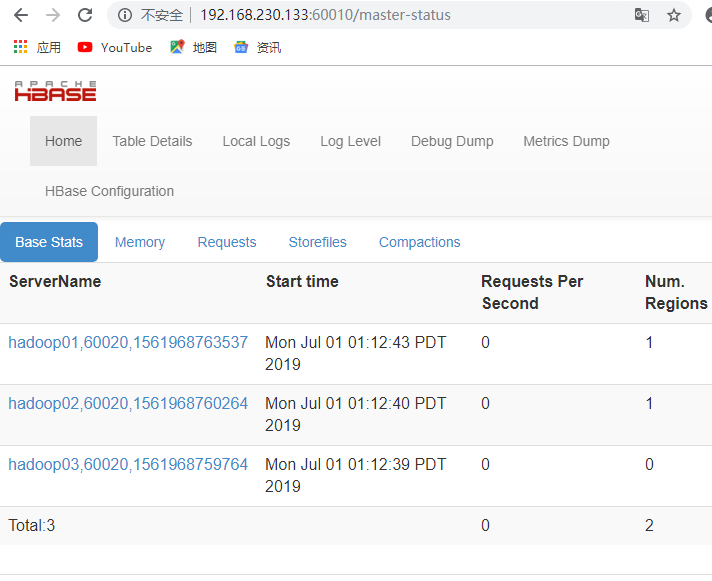


图3-48

这个页面可以查看到HBase集群的当前的状态。

# HBase的模式设计

## Hbase的行键

HBase本质上也是一种Key-Value存储系统。那么RowKey就是KeyValue的key了,表示唯一一行，Value相当于列族数据的集合。RowKey也是一段二进制码流，最大长度为64KB内容可以由使用的用户自定义，数据加载时，也是根据RowKey的二进制序由小到大的进行。与nosql数据库们一样，HBase是根据RowKey来进行检索的,系统通过找到某个RowKey所在的Region然后将查询数据的请求路由到该Region获取数据。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RowKey访问，即按照某个RowKey键值进行get操作,这样获取唯一一条记录。

2.通过RowKey的range进行scan ,即通过设置startRowKey和endRowKey,在这个范围内进行扫描。

3.全表扫描，即直接扫描整张表中所有行记录。

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)，不要超过16个字节。在HBase内部，row key保存为字节数组。存储时，数据按照Row key的字典序(byte order)排序存储。设计key时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。(位置相关性)

注意：

字典序对int排序的结果是：

1,10,100,11,5,13,14,15,16,17,18,19,2,20,21,…,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。

## HBase的列族

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，列族必须在使用表之前定义。列名都以列族作为前缀。例如courses:history ， courses:math 都属于 courses 这个列族。访问控制、磁盘和内存的使用统计都是在列族层面进行的。实际应用中，列族上的控制权限能帮助我们管理不同类型的应用：我们允许一些应用可以添加新的基本数据、一些应用可以读取基本数据并创建继承的列族、一些应用则只允许浏览数据（甚至可能因 为隐私的原因不能浏览所有数据）。

一行有若干列组成，若干列又构成一个列族（column family），这不仅有助于构建数据的语义边界或者局部边界，还有助于给他们设置某些特性（如压缩），或者指示他们存贮在内存中，一个列族的所有列存贮在同一个底层的存储文件中，这个存储文件叫做HFile。列族需要在创建表时就定义好，并且修改的不能太频繁，数量也不能太多，理论上是限制在几十个，但是实际中可能会更少。列族的名称必须是由可打印的字符组成，这个是与其他的值或名字的命名规范显著不同的地方。​

## HBase的表设计

### Hbase的行键设计

行健（Rowkey）是不可分割的字节数，按字典排序由低到高存储在表中。这对扫描（Scan）操作是一个优化。行健是HBase最有效的索引，与MySQl数据库相比，MySQL数据库支持多字段的索引，HBase不支持二级索引，因此MySQL通过多重、多列支持的复杂数据库查询操作对HBase可能是个灾难。当HBase也支持对列的条件过滤（参见5.3节），但是因为需要读取存储文件做字符或者字节对比，所以效率很低，对性能影响很大。

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

1.Region基于Rowkey为一个区间的行提供服务，并且负责区间的每一行；

2.HFile在硬盘上存储有序的行。

这两个因素是相互关联的。当Region将内存中数据刷写为HFile时，这些行已经排过序，也会有序地写到硬盘上。Rowkey的有序特性和底层存储格式可以保证HBase表在设计Rowkey之后的良好性能。关系型数据库可以在多列上建立索引，但是HBase只能在Rowkey上建立索引。（可以通过ES为HBase的列建立索引） 而设计Rowkey有各种技巧，而且可以针对不同访问模式进行优化，我们接下来就研究一下。

1.将Rowkey以字典顺序从大到小排序

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

2.RowKey尽量散列设计

最重要的是要保证散列，这样就会保证所有的数据都不是在一个Region上，从而避免读写的时候负载会集中在个别Region上。比如ROWKEY\_Random。

3.RowKey的长度尽量短

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

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

原因如下：

1）数据的持久化文件HFile中是按照KeyValue存储的，如果Rowkey过长比如100个字节，1000万列数据光Rowkey就要占用100\*1000万=10亿个字节，将近1G数据，这会极大影响HFile的存储效率；

2）MemStore将缓存部分数据到内存，如果Rowkey字段过长内存的有效利用率会降低，系统。将无法缓存更多的数据，这会降低检索效率。因此Rowkey的字节长度越短越好。

4.RowKey唯一：行健对应关系型数据库的唯一键，系统设计之初必须考虑有足够的唯一行健去支持业务的数据量。

5.RowKey建议用String类型

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

常用的行键字符串有以下几种：

1）纯数字字符串，譬如955982014055；

2）数字+特殊分隔符，譬如95598-2014055;

3）数字+英文字母，譬如city2014055；

4）数字+英文字母+特殊分隔符，譬如city\_2014055

6.RowKey设计得最好有意义

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

譬如数字字符串：955982014055，其实际意义是这样：95598（电网客服电话）+2014055（日期）。行键往往由多个值组合而成，而各个值的位置顺序将影响到数据存储和检索效率，所以在设计行键时，需要对日后的业务应用开发有比较深入的了解和前瞻性预测，才能设计出可尽量高效率检索的行键。

7.具有定长性

行键具有有序性的基础便是定长，譬如2014055080500、2014055083000，这两个日期时间形式的字符串是递增的，不管后面的秒数是多少，我们都将其设置为14位数字形式，如果我们把后面的0去除了，那么20140550805将大于2014055083，其有序性发生了变更。所以我们建议，行键一定要设计成定长的。此外，目前操作系统是都是64位系统，内存8字节对齐。控制在16个字节，8字节的整数倍利用操作系统的最佳特性。

下面从统计需求来看用户行为日志系统表s\_behavior的行键设计。

查询同一个用户一段时间内浏览过的商品:该需求有两个数据查询维度，第

一 个是用户，第二个是时间。为了提高查询性能，需要把同一个用户的数据聚簇地放在一起，因此可以把用户ID作为行键的开始。类似于MySQL的组合索引，用户ID作为组合索引的引导列，时间则可以作为组合索引的第二列，这样得到的行键格式为“[用户ID]\_[时间戳]”。在某些极端并发情况下，例如，用户浏览器同时打开多个商品，那么这些浏览记录时间戳可能相同。考虑到唯一性原则，可以在行键最后添加一个序列号来实现，因此最后得到的行键格式为“[用户ID]\_[时间戳]\_ [序列号]”。用户浏览的商品ID、订单ID等可以作为列值数据存储。

统计某个商品某天的转化率:转化率的定义是商品下单数除以商品浏览数，该需求有3个数据查询维度，第一个是商品，第二个是数据类型，第三个是时间。可以把商品ID作为行键的开始，接下来用一个数字表示用户的数据类型(如0:商品浏览记录; 1:商品下单记录)，再加上时间戳。同一个商品同一时间会有很多

用户浏览和下单，根据唯一性原则，行健最后也需要添加一个序列号，因此最后得到的行键格式为“[商品ID]\_[数据类型]\_[时间戳]\_[序列号]”，用户ID等可作为列值数据存储。  
 当然一个完整的用户日志系统还要考虑很多其他需求，但是目前为止已经足够麻烦了，因为两个需求得到的行键完全没有共同点。下面分析如何在保持性能基本稳定的同时满足以上两个需求。  
 使用时间戳过滤: 注意，两个需求的行键都包括了时间戳， 那么是否可以把时间戳顺序提前用作行键的开始呢?假如把时间戳用作行键，那么无论统计哪个需求，都需要把整个时间段内所有的数据都扫描出来，然后在服务端或者客户端过滤，这样会导致大量非必要的数据读取。例如，查询某个用户在2018年1月1日这一天所有的商品浏览数据，则需要将整个用户行为系统在这一天产生的所有数据全部扫描一次再过滤，性能无疑非常低下，无法满足需求。  
 使用二级索引: HBase本身不支持二级索引，唯一的索引就是行键。由于HBase对复杂查询过滤条件支持的局限性，因此开源社区对HBase二级索引提出了多个解决方案，如IHBase、华为的HIndex等。这些解决方案可能实现方式不一样(如华为HIndex使用协处理器实现)，各有优劣，但是最终目标都是实现HBase对复杂查询条件的支持，同时在性能、数据一致性、 代码侵入性方面能够达到平衡。回到用户行为管理系统，为了满足两个查询需求，可以考虑按用户维度构建行键存储数据，按商品维度建立二级索引满足商品维度统计需求，如表11-1所示。

|  |  |  |
| --- | --- | --- |
| 行健 | 列族：列浏览记录  （cf:v） | 列族：列下单记录  （cf:0） |
| 5345\_1510720956000\_1 | 1001 | 1001 |
| 5345\_1510721056000\_2 | 1002 |  |
| 5345\_1510721086000\_3 | 1001 |  |
| 5345\_1510721096000\_4 |  | 1001 |
| ... | ... | 1001 |
| 5345\_idx1\_1001\_0\_1510720956000\_5345\_1510720956000\_1 |  |  |
| 5345\_idx1\_1001\_0\_1510721086000\_5345\_1510721086000\_3 |  |  |
| 5345\_idx1\_1001\_1\_1510720956000\_5345\_1510720956000\_1 |  |  |
| 5345\_idx1\_1001\_1\_1510721096000\_5345\_1510721096000\_4 |  |  |
| 5345\_idx1\_1002\_0\_1510721086000\_5345\_1510721056000\_2 |  |  |

表11-1前4行为数据部分，后4行为数据索引部分，下面解释每行数据的含义。

第一行:用户5345在时间戳1510720956000浏览了商品1001，同时下单了商品1001。

第二行:用户5345在时间戳1510721056000浏览了商品1002。

第三行:用户5346在时间戳1510721086000浏览了商品1001。

第四行:用户5346在时间戳1510721096000下单了商品1001。

从第六行开始就是对前四行数据的索引，索引数据只有行键，列值数据均为空，索引格式为“[分区开始行键]\_[索引名称]\_[商品ID]\_[数据类型]\_[时间戳]\_[数据行键]”，使用分区开始行键作为行键的前缀是为了让索引和数据在同一个分区，这样读取索引和数据的请求就会落在同一个分区服务器,否则可能需要在分区服务器A读取索引后，再跨机器去分区服务器B读取数据。下面看看如何使用二级索引满足系统的两个统计需求。

查询同一个用户一段时间内浏览过的商品:数据的行键以用户ID开始，因此支持对该用户数据的统计查询，扫描请求只需指定该用户的最小行键和最大行键。例如用户查询用户5345从2017年5月1日到2018年1月1日之间的商品浏览记录，只需指定查询的行键区间[5345\_ 155086400000, 5345\_ 1514764800000)。

统计某个商品某天的转化率:这里二级索引就可以发挥作用了，查询的主要维度是商品，因此可以用到以商品ID开始的索引idx1。假设需要统计商品1001在2018年1月1日这天的转化率，需要扫描的行健区间为[5345\_idx1\_1001 0\_151474800000, 5345\_idxl\_1001\_1151485200000并且只需扫描所有的索引行键。拿到所有的索引行键后循环遍历，根据数据类型判断是浏览记录还是下单记录，对浏览记录和下单记录，分别求和即可得到转化率。注意，这里的场最比较特殊，通常情况下还会需要根据索引行键得到数据行键，再根据数据行键去查询数据，因此索引行键的最后是数据行键。

### HBase的列族设计

1.同一个列族里存储相似访问模式的所有数据。

    经常要在一起查询的数据最好放在一个列族中，尽量的减少跨列族的数据访问。关系数据库中一对一关系，一对一关系的两张表，在HBase中可以存储在一张表中，放到不同的列族。

2.在设计HBase表时候，列族不宜过多，尽量的要少使用列族。

建议将HBase列族的数量设置的越少越好。对于两个或两个以上的列族HBase并不能处理的很好。这是由于HBase的Flushing和压缩是基于Region的。当一个列族所存储的数据达到Flushing的阈值时，该表中所有列族将同时进行Flushing操作。这将带来不必要的I/O开销，列族越多，该特性带来的影响越大。

此外，还要考虑到同一个表中不同列族所存储的记录数量的差别，即列族的势(Cardinality)。当两个列族数量差别过大时会使包含记录数量较少列族的数据分散在多个Region上，而Region有可能存储在不同的RegionServer上。这样，当进行查询或scan操作的时候，系统效率将会受到影响。在多列簇的情况下，注意各列簇数据的数量级要一致。如果两个列簇的数量级相差太大，会使数量级少的列簇的数据扫描效率低下。将经常查询和不经常查询的数据放到不同的列簇。

# HBase API

## API概述

HBase是由Java进行编写的，原生的API也是由Java开发的，但是这也不意味着一定要通过Java访问HBase。也是可以通过其他的编程语言来使用HBase的。

HBase的主要客户端接口是org.apache.hadoop.HBase.client包中HTable类提供的，通过这个类我们向HBase中存储和检索数据，以及删除一些无效的数据。

数据库的基本操作具体指增、删、改、查操作。在HBase中也有与之对应的一组操作，这些方法都是由HTable类提供的。

## API处理操作

实现步骤：

1.右键点击New，然后选择java Project，在弹出的New java Project的对话框中，填写项目名称，点击Finish完成项目的创建，如图5-1所示。

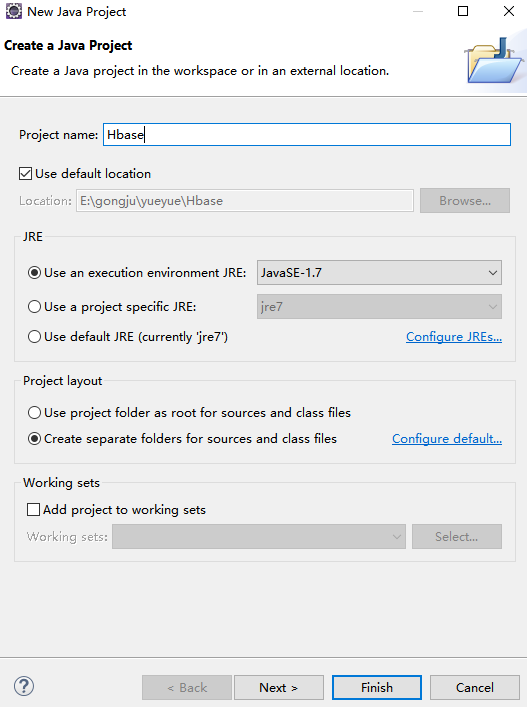


图5-1

2.选中HBase项目，右键点击New，然后选择Folder，在弹出的对话框New Folder中，填写目录名称lib，点击Finish完成目录创建，如图5-2所示。

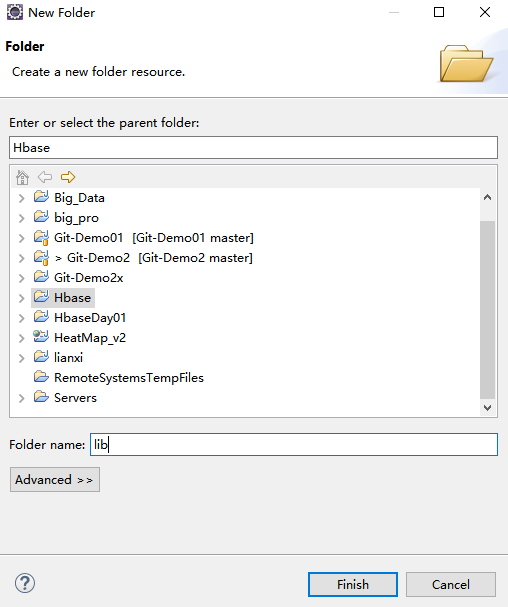


图5-2

3.将HBase的相关jar导入的lib目录下，然后右键Build Path，如图5-3所示。

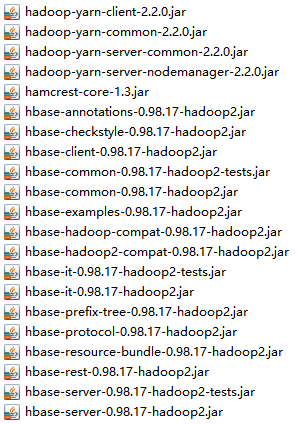


图5-3

4.右键点击New，然后点击class，在弹出的对话框New Java Class中，填写类的名称TestDemo，如图5-4所示。

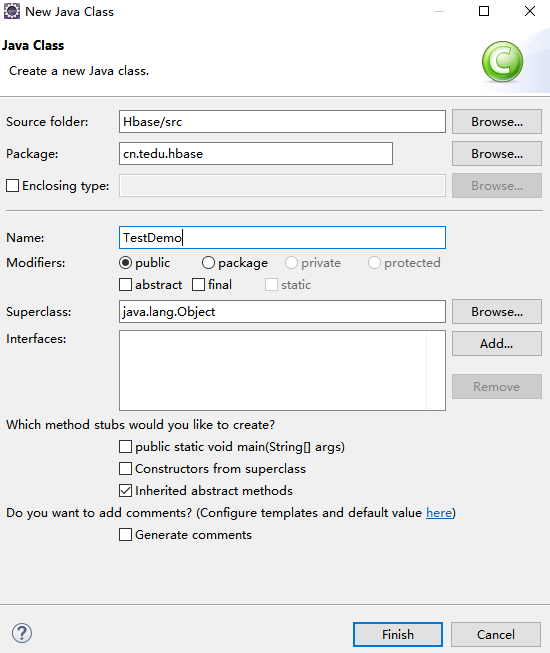


图5-4

5.因当HBase启动的时候，每一个节点都会自动的在Zookeeper上来注册一个临时节，在注册临时节点的时候，注册的不是IP地址而是主机名。需要在host文件中配置相应的IP地址和主机名。host文件所在的目录是：

C:\Windows\System32\drivers\etc，配置如下：

192.168.230.133 hadoop01

192.168.230.134 hadoop02

192.168.230.135 hadoop03

6.利用Java代码创建表tbb1，代码如下：



右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，如图5-5所示。

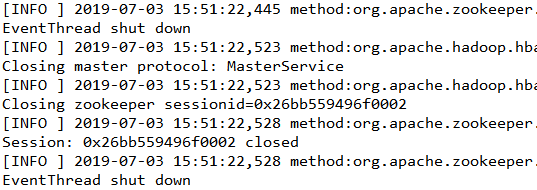


图5-5

在HBase中可以查看表tbb1创建成功，如图5-6所示。

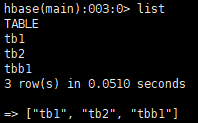


图5-6

7.向指定的HBase表插入数据，代码如下：

右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，查看tbb1表数据的插入情况，执行命令：scan 'tbb1'，如图5-7所示。

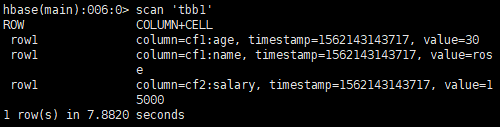


图5-7

8.右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，查看tbb1表数据的插入情况，执行命令：scan 'tbb1'，如图5-8所示。

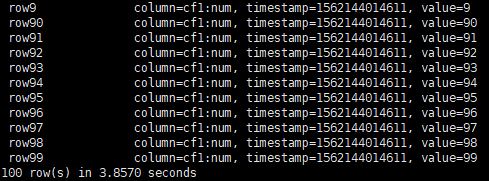


图5-8

9.通过指定行键读取数据，代码如下：



右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，如图5-9所示。

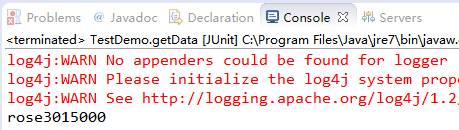


图5-9

10.通过scan扫描表数据,也可以通过scan做范围查询



右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，如图5-10所示。

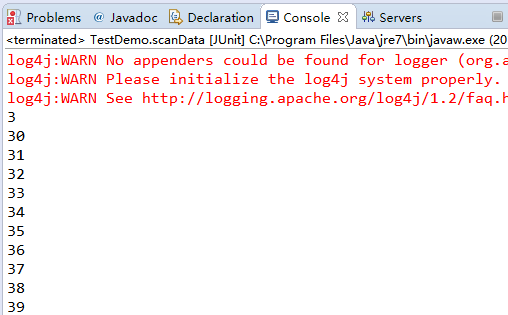


图5-10

11.删除表操作，代码如下：



右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，如图5-11所示。

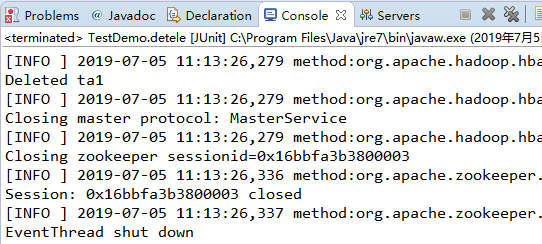


图5-11

查看HBase中的表，输入list，可以查看到ta2已经被删除，如图5-5所示。

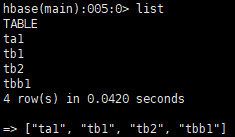


图5-5

## 过滤器的概述

HBase过滤器是一套为完成一些较高级的需求所提供的API借口。从过滤器的名称我们就可以看出：过滤器就是对数据库获取的数据进行过滤，将符合条件的数据返回客户端，从而减少从region服务器向客户端发送的数据，从而减少数据传输，提高效率。

HBase中主要的数据读取函数是get()和scan()，它们都支持直接访问数据和通过指定起止行键访问数据的功能。我们可以在查询中添加更多的限制条件来减少查询得到的数据量，这些限制可以是指定列族、列、时间戳以及版本号。这些方法可以帮助我们控制哪些数据在查询时被包含其中，但是它们缺少一些细粒度的筛选功能，比如基于正则表达式对行键或是值进行筛选。Get和Scan两个类都支持过滤器，理由如下：这类对象提供的基本API不能对行键、列名或列值进行过滤，但是通过过滤器可以达到这个目的。过滤器最基本的接口叫Filter，除此之外，还有一些由HBase提供的无需编程就可以直接使用的类。

同时，我们还可以通过继承Filter类来实现自己的需求。所有的过滤器都在服务器端生效，叫做谓词下推。这样可以保证被过滤掉的数据不会被传送到客户端。我们可以在客户端代买中实现过滤的功能（但会影响系统性能），因为在这种情况下服务器端需要传输更多的数据到客户端，我们应当尽量避免这种情况。

在过滤器层级结构的最底层是Filter接口和FilterBase抽象类，它们实现了过滤器的空壳和骨架，这使得实际的过滤器类可以许多重复的结构代码。大部分实体过滤器类一般都直接继承自FilterBase，也有一些间接继承自该类。不过它们的使用流程都是相同的，用户定义一个需要的过滤器实例，同时把定义好的过滤器实例传递给Get或Scan实例：setFilter(filter)。在实例化过滤器的时候，用户需要提供一些参数来设定过滤器的用途。其基本格式是：所需类型的过滤器（例如：行过滤器，值过滤器，依赖列过滤器等等），在过滤器中包含三个基本的参数，第一个是要过滤什么（比如说过滤id，过滤name等等），第二个是要比较过滤的方式（例如等于，不等于，大于等于等等），第三个就是要比较过滤的值。

## 过滤器的处理操作

1.HBase的过滤器将近有15种，需要重点掌握正则过滤器，正则过滤器是最常用的，代码如下：



右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，如图5-13所示。

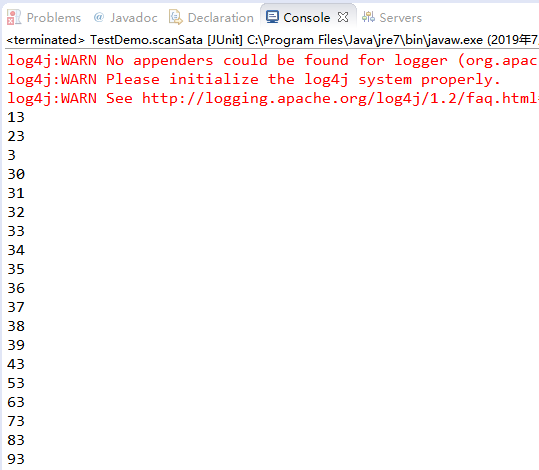


图5-13

2.行健过滤器基于行健来过滤数据的，以下代码展示了如何通过行健比较过滤器来获取需要的行数据的。示例代码如下：



右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，如图5-14所示。

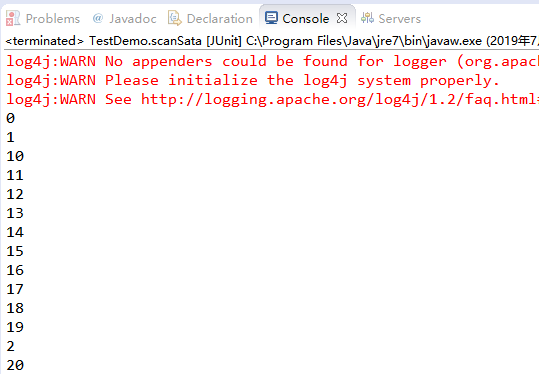


图5-14

行键前缀过滤器，所有和前缀匹配的过滤器都会被返回到客户端，示例代码如下：



右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，如图5-15所示。

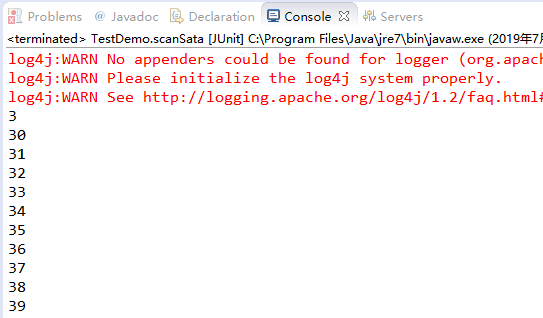


图5-15

4.列值过滤器和行健过滤器相似，只不过它是比较的列值而不是比较的行健来返回结果的，示例代码如下。





右键点击Run As，然后点击JUnit Test，运行程序，程序运行成功，如图5-16所示。

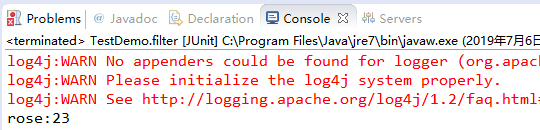


图5-16

# Hbase的读写流程

## HBase的写流程

当客户端发起一个Put请求时，首先它从HBase:meta表中查出该Put数据最终需要去的HRegionServer。然后客户端将Put请求发送给相应的HRegionServer，在HRegionServer中它首先会将该Put操作写入WAL日志文件中(Flush到磁盘中)。如图6-1所示。

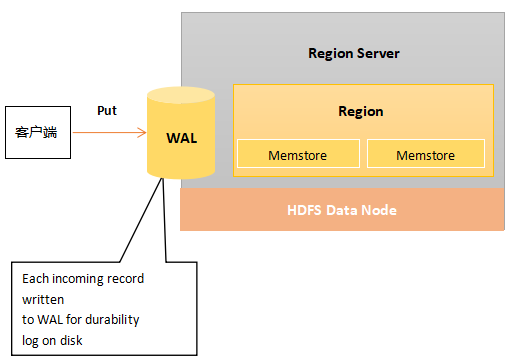


图6-1

写完WAL日志文件后，然后会将数据写到Memstore，在Memstore按Rowkey排序，以及用LSM-TREE对数据做合并处理。HRegionServer根据Put中的TableName和RowKey找到对应的HRegion，并根据Column Family找到对应的HStore，并将Put写入到该HStore的MemStore中。此时写成功，并返回通知客户端。如图6-2所示。

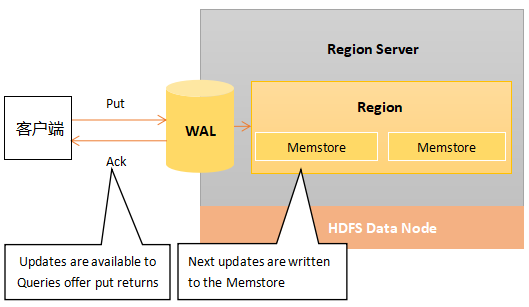


图6-2

MemStore是一个In Memory Sorted Buffer，在每个HStore中都有一个MemStore，即它是一个HRegion的一个Column Family对应一个实例。它的排列顺序以RowKey、Column Family、Column的顺序以及Timestamp的倒序，如图6-3所示。

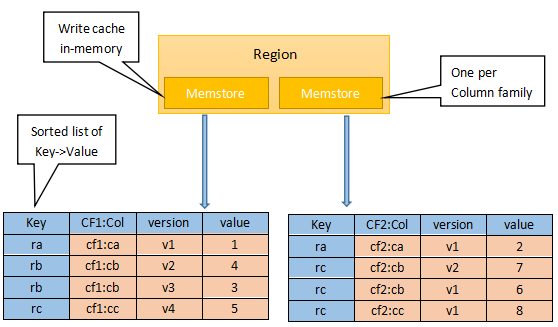


图6-3

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

当一个HRegion中的MemStore的大小超过了：

HBase.hregion.memstore.flush.size的大小，默认128MB。此时当前的MemStore会Flush到HFile中。

当RS服务器上所有的MemStore的大小超过了：HBase.regionserver.global.memstore.upperLimit的大小，默认35％的内存使用量。此时当前HRegionServer中所有HRegion中的MemStore可能都会Flush。从最大的Memostore开始flush。

当前HRegionServer中WAL的大小超过了 1GB

HBase.regionserver.hlog.blocksize(32MB) \*HBase.regionserver.max.logs(32)的数量，当前HRegionServer中所有HRegion中的MemStore都会Flush。这里指的是两个参数相乘的大小。

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

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

如图6-4所示。

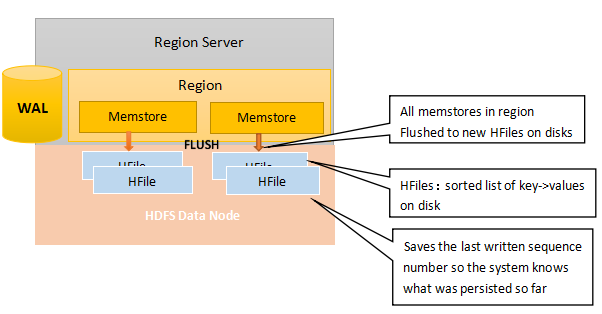


图6-4

HBase的数据以KeyValue(Cell)的形式顺序的存储在HFile中，在MemStore的Flush过程中生成HFile，由于MemStore中存储的Cell遵循相同的排列顺序，因而Flush过程是顺序写，我们知道磁盘的顺序写性能很高，因为不需要不停的移动磁盘指针。如图6-5所示。

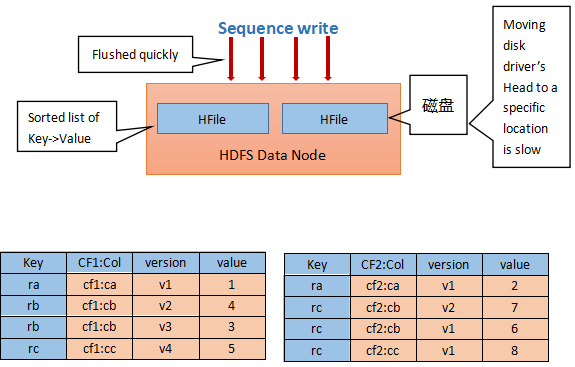


图6-5

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

首先我们来看一下V1的格式，如图6-6所示。

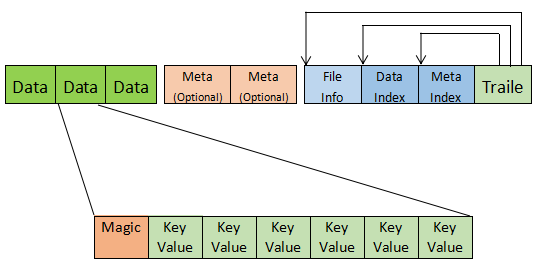


图6-6

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

是固定长度的。

HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构，如图6-7所示。

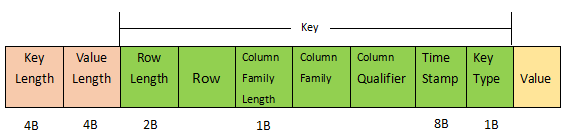


图6-7

开始是两个固定长度的数值，分别表示Key的长度和Value的长度。紧接着是Key，开始是固定长度的数值，表示RowKey的长度，紧接着是 RowKey，然后是固定长度的数值，表示Family的长度，然后是Family，接着是Qualifier，然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。随着HFile版

本迁移，KeyValue(Cell)的格式并未发生太多变化，只是在V3版本，尾部添加了一个可选的Tag数组。

HFileV1版本的在实际使用过程中发现它占用内存多，因而增加了启动时间。为了解决这些问题，在0.92版本中引入HFileV2版本，如图6-8所示。

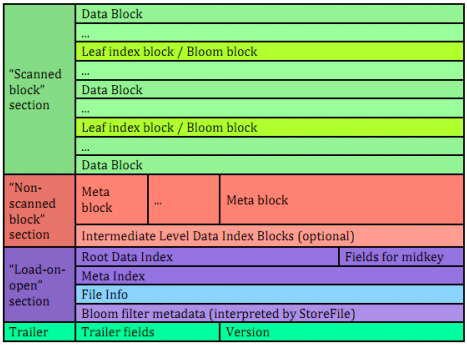
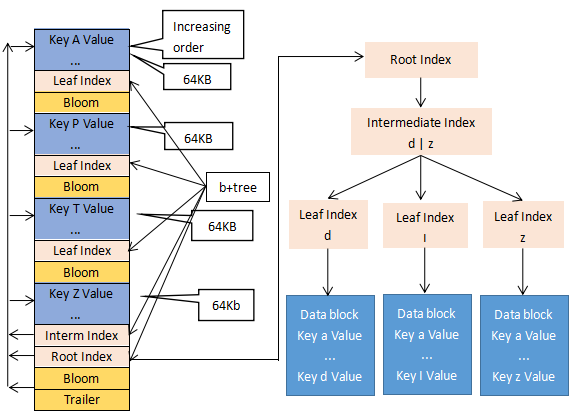


图6-8

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

对HFileV2格式具体分析，它是一个多层的类B+树索引，采用这种设计，可以实现查找不需要读取整个文件，如图6-9所示。

图6-9

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

## Hbase的读流程

先来分析一下相同的Cell（数据）可能存在的位置：首先对新写入的Cell，它会存在于MemStore中；然后对之前已经Flush到HFile中的Cell，它会存在于某个或某些StoreFile(HFile)中；最后，对刚读取过的Cell，它可能存在于BlockCache中。既然相同的Cell可能存储在三个地方，在读取的时候只需要扫瞄这三个地方，然后将结果合并即可(Merge Read)，在HBase中扫瞄的顺序依次是：BlockCache、MemStore、StoreFile(HFile)（这个扫描顺序的目的也是为了减少磁盘的I/O次数）。如图6-10所示。

其中StoreFile的扫瞄先会使用Bloom Filter(布隆过滤算法）过滤那些不可能符合条件的HFile，然后使用Block Index快速定位Cell，并将其加载到BlockCache中，然后从BlockCache中读取。我们知道一个HStore可能存在多个StoreFile(HFile)，此时需要扫瞄多个HFile，如果HFile过多又是会引起性能问题。

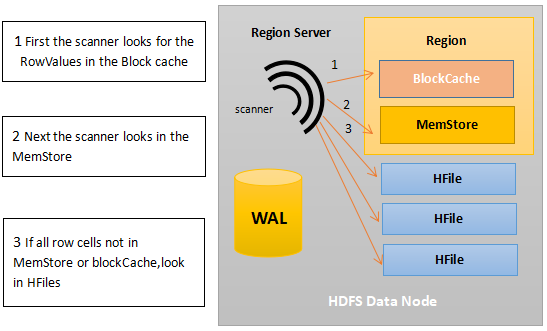


图6-10

Compaction机制主要分为Minor Compaction机制和Major Compaction机制。

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

Minor Compaction是指选取一些小的、相邻的StoreFile将他们合并成一个更大的StoreFile，在这个过程中不会处理已经Deleted或Expired的Cell。一次Minor Compaction的结果是更少并且更大的StoreFile。也就是说它将memtable的数据flush的一个HFile/SSTable称为一次Minor Compaction。

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。Minor compaction的运行机制要复杂一些，它由一下几个参数共同决定：   
　　HBase.hstore.compaction.min :默认值为 3，表示至少需要三个满足条件的store file时，minor compaction才会启动 。  
　　HBase.hstore.compaction.max 默认值为10，表示一次Minor compaction中最多选取10个store file 。  
　　HBase.hstore.compaction.min.size 表示文件大小小于该值的store file 一定会加入到Minor compaction的store file中 。  
　　HBase.hstore.compaction.max.size 表示文件大小大于该值的store file 一定会被Minor compaction排除 。  
　　HBase.hstore.compaction.ratio 将store file 按照文件年龄排序（older to younger），Minor compaction总是从older store file开始选择，如果该文件的size 小于它后面HBase.hstore.compaction.max 个store file size 之和乘以该ratio，则该store file 也将加入到Minor compaction中。

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

major compaction的功能是将所有的store file合并成一个，触发major compaction的可能条件有：major\_compact 命令、majorCompact()API、region server自动运行（相关参数：HBase.hregion.majoucompaction默认为24 小时、HBase.hregion.majorcompaction.jetter默认值为0.2，防止region server 在同一时间进行major compaction）。HBase.hregion.majorcompaction.jetter参数的作用是：对参数HBase.hregion.majoucompaction规定的值起到浮动的作用，假如两个参数都为默认值24和0,2，那么major compact最终使用的数值为：19.2~28.8 这个范围。

更形象一点，如下面两张图分别表示Minor Compaction和Major Compaction。如图6-11、6-12所示。

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

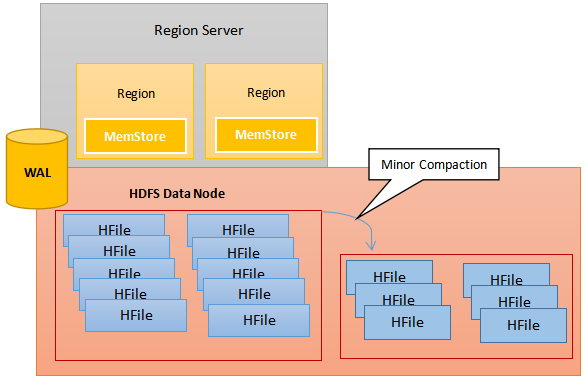


图6-11

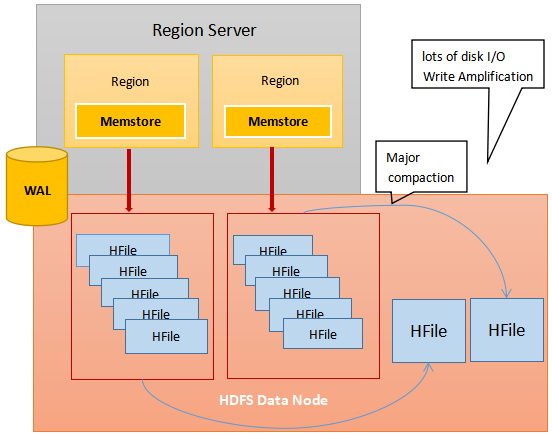


图6-12

# HBase的扩展内容

## HBase的优化

1.配置内存

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

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

2.配置CPU

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

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

3.垃圾回收器（GC）的选择

对于运行HBase相关进程JVM的垃圾回收器，不仅仅关注吞吐量，还关注停顿时间，而且两者之间停顿时间更为重要，因为HBase设计的初衷就是解决大规模数据集下实时访问的问题。那么按照首位是停顿时间短，从这个方面CMS和G1有着非常大的优势。而CMS作为JDK1.5已经出现的垃圾收集器，已经成熟应用在互联网等各个行业。所以，选用CMS作为老年代的垃圾回收器。与CMS搭配的新生代收集器有Serial和ParNew，而对比这两个收集器，明显ParNew具有更好的性能，所以新生代选用ParNew作为垃圾收集器。那么，最终选用的垃圾收集器搭配组合是CMS+ParNew。而且很多成熟应用已经验证了这种组合搭配的优势。

配置方式：需要添加到HBase-env.sh文件中

export HBase\_OPTS="-XX：+UseConcMarkSweepGC" -XX：CMSInitiatingOccupancyFraction=70 -XX：+UseCMSCompactAtFullCollection

4.JVM堆大小设置

堆内存大小参数HBase-env.sh文件中设置，设置的代码如下：

export HBase\_HEAPSIZE=16384，在上面代码中指定堆内存大小是16284，单位是MB，即16GB。当然，这个值需要根据节点实际的物理内存来决定。一般不超过实际物理内存的1/2。服务器内存的分配，比如服务器内存64GB，为操作系统预留出8G~16GB。此外给Yarn留出8G~16GB，如果没有其他框架，把剩余的留给HBase

## Hbase的调优

1.调节数据块（data block）的大小

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

可以在表实例化时设置数据块大小，代码如下：

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

2.适当时机关闭数据块缓存

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

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

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

可以在新建表或更改表时关闭数据块缓存属性：

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

3.开启布隆过滤器

数据块索引提供了一个有效的方法getDataBlockIndexReader（），在访问某个特定的行时用来查找应该读取的HFile的数据块。但是该方法的作用有限。HFile数据块的默认大小是64KB，一般情况下不能调整太多。如果要查找一个很短的行，只在整个数据块的起始行键上建立索引是无法给出更细粒度的索引信息的。例如，某行占用100字节存储空间，一个64KB的数据块包含（64×1024）/100=655.53，约700行，只能把起始行放在索引位上。要查找的行可能落在特定数据块上的行区间，但也不能肯定存放在那个数据块上，这就导致多种可能性：该行在表中不存在，或者存放在另一个HFile中，甚至在MemStore中。这些情况下，从硬盘读取数据块会带来I/O开销，也会滥用数据块缓存，这会影响性能，尤其是当面对一个巨大的数据集且有很多并发读用户时。

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

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

可以在列族上打开布隆过滤器，代码如下：

HBase（main）：007：0> create 'mytable'， {NAME => 'colfam1'， BLOOMFILTER => 'ROWCOL'}。布隆过滤器参数的默认值是NONE。另外，还有两个值：ROW表示行级布隆过滤器；ROWCOL表示列标识符级布隆过滤器。行级布隆过滤器在数据块中检查特定行键是否不存在，列标识符级布隆过滤器检查行和列标识符联合体是否不存在。ROWCOL布隆过滤器的空间开销高于ROW布隆过滤器。

4.开启数据压缩

HFile可以被压缩并存放在HDFS上，这有助于节省硬盘I/O，但是读写数据时压缩和解压缩会抬高CPU利用率。压缩是表定义的一部分，可以在建表或模式改变时设定。除非确定压缩不会提升系统的性能，否则推荐打开表的压缩。只有在数据不能被压缩，或者因为某些原因服务器的CPU利用率有限制要求的情况下，有可能需要关闭压缩特性。HBase可以使用多种压缩编码，包括LZO、SNAPPY和GZIP，LZO和SNAPPY是其中最流行的两种。

当建表时可以在列族上打开压缩，代码如下：

HBase（main）：002：0>

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

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

5.设置Scan缓存

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

下面的代码展示了如何使用setCaching（）方法。

代码示例：





6.显式地指定列

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

代码示例：



7.关闭ResultScanner

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

8.使用批量读

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

代码示例：



9.使用批量写

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

对于批量写入方法的使用见下面代码：



10.关闭写WAL日志

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

具体的设置方法如下面代码所示：





11.设置AutoFlush

HTable有一个属性是AutoFlush，该属性用于支持客户端的批量更新。该属性默认值是true，即客户端每收到一条数据，立刻发送到服务端。如果将该属性设置为false，当客户端提交Put请求时，将该请求在客户端缓存，直到数据达到某个阈值的容量时（该容量由参数HBase.client.write.buffer决定）或执行HBase.flushcommits（）时，才向RegionServer提交请求。这种方式避免了每次跟服务端交互，采用批量提交的方式，所以更高效。但是，如果还没有达到该缓存而客户端崩溃，该部分数据将由于未发送到RegionServer而丢失。这对于有些零容忍的在线服务是不可接受的。所以，设置该参数的时候要慎重。

HTable设置AutoFlush的示例代码如下：





12.预创建Region

在HBase中创建表时，该表开始只有一个Region，插入该表的所有数据会保存在该Region中。随着数据量不断增加，当该Region大小达到一定阈值时，就会发生分裂（Region Splitting）操作。并且在这个表创建后相当长的一

段时间内，针对该表的所有写操作总是集中在某一台或者少数几台机器上，这不仅仅造成局部磁盘和网络资源紧张，同时也是对整个集群资源的浪费。这个问题在初始化表，即批量导入原始数据的时候，特别明显。为了解决这个问题，可以使用预创建Region的方法。

HBase内部提供了RegionSplitter工具，使用命令如下：



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

13.调整ZooKeeper Session的有效时长

参数zookeeper.session.timeout用于定义连接ZooKeeper的Session的有效时长，这个默认值是180秒。这意味着

一旦某个RegionServer宕机，HMaster至少需要180秒才能察觉到宕机，然后开始恢复。