1.Hbase与传统关系数据库的区别主要体现在几个方面？请用自己的语言分别阐述这几个区别。

HBase作为一种NoSQL的数据库，和传统的类似于mysql这样的关系型数据库是有很大区别的，本文来对他们做一个对比分析，以便更加深入的了解HBase。它们的区别体现在六个方面：数据类型、数据操作、存储模式、数据索引、数据维护、可伸缩性。

HBase作为一种NoSQL的数据库，和传统的类似于mysql这样的关系型数据库是有很大区别的，本文来对他们做一个对比分析，以便更加深入的了解HBase。

主要区别体现在以下六个方面：

1、数据类型

关系型数据库采用关系模型，具有丰富的数据类型和存储方式，比如我们熟知的mysql可以存储char、varchar、int等等各种类型的数据，而HBase则采用了更简单的数据模型，将数据存储为未解释的字符数组，所有数据都没有类型，都直接作为字符数组来看待。

2、数据操作

关系型数据库有丰富的数据操作，最基本的如：增删改查，并且这其中涉及复杂的多表连接，而HBase操作则不存在复杂的表与表之间的连接关系，只有简单的插入、查询、删除、清空等操作，因为HBase在设计上就避免了复杂的表和表之间的关系。

3、存储模式

关系型数据库是基于行模式来存储的，而HBase正如我们介绍的那样，它是基于列存储的，每个列族的数据都保存在一起，不同列族的文件是相互分离的。

4、数据索引

在关系型数据库中，有模式、表、视图、索引等多个概念，RDBMS可以根据不同列构建复杂的多个索引，以此来提高数据访问的性能，而HBase我们看到，实际上它只有一个索引，那就是行键，HBase表的所有访问都要通过行键，通过巧妙的设计，HBase中的所有访问方法，或者通过行键访问，或者通过行键扫描。

 5、数据维护

 关系型数据库中，更新操作会用当前的最新值去替换记录中的旧值，旧值被覆盖后就不再存在了，而HBase中的每个单元格是有一个以时间戳表示的版本号的，它在执行更新操作时，并不会删除数据的旧版本，而是生成一个具有更新的版本号的新数据，原有旧数据仍然会保存。

6、可伸缩性

关系型数据库大多受限于一台机器的存储容量，只能进行垂直扩展（更换大容量的设备、提升设备性能等等），很难做到横向扩展，而HBase设计的初衷本来也就是为了实现灵活的横向扩展，能够通过简单的增加节点数量来实现扩展，从而可以存储超大规模数据集，实现超大规模数据的随机访问。

2.请描述网页搜索的过程

Part1.分词

分词就是对一段文本，通过规则或者算法分出多个词，每个词作为搜索的最细粒度一个个单字或者单词。只有分词后有这个词，搜索才能搜到，分词的正确性非常重要。分词粒度太大，搜索召回率就会偏低，分词粒度太小，准确率就会降低。如何恰到好处的分词，是搜索引擎需要做的第一步。

正确性&粒度

分词正确性

“他说的确实在理”，这句话如何分词？

“他-说-的确-实在-理” [错误语义]

“他-说-的-确实-在理” [正确语义]

分词的粒度

“中华人民共和国宪法”，这句话如何分词？

“中华人民共和国-宪法”，[搜索 中华、共和国 无结果]

“中华-人民-共和国-宪法”，[搜索 共和 无结果]

“中-华-人-民-共-和-国-宪-法”，[搜索其中任意字都有结果]

分词的粒度并不是越小越好，他会降低准确率，比如搜索 “中秋” 也会出现上条结果，而且粒度越小，索引词典越大，搜索效率也会下降，后面会细说。

如何准确的把控分词，涉及到 NLP 的内容啦，这里就不展开了。

停用词

很多语句中的词都是没有意义的，比如 “的”，“在” 等副词、谓词，英文中的 “a”，“an”，“the”，在搜索是无任何意义的，所以在分词构建索引时都会去除，降低不不要的索引空间，叫停用词 (StopWord)。

通常可以通过文档集频率和维护停用词表的方式来判断停用词。

词项处理

词项处理，是指在原本的词项上在做一些额外的处理，比如归一化、词形归并、词干还原等操作，以提高搜索的效果。并不是所有的需求和业务都要词项处理，需要根据场景来判断。

1、归一化

USA - U.S.A.[缩写]

7月30日 - 7/30 [中英文]

color - colour [通假词]

开心 - 高兴 [同义词扩展范畴]

这样查询 U.S.A.也能得到 USA 的结果，同义词可以算作归一化处理，不过同义词还可以有其他的处理方式。

2、词形归并（Lemmatization）

针对英语同一个词有不同的形态，可以做词形归并成一个，如：

am, are, is -> be

car, cars, car's, cars' -> car

the boy's cars are different colors -> the boy car be different color

3、词干还原（Stemming）

通常指的就粗略的去除单词两端词缀的启发式过程

automate(s), automatic, automation -> automat.

高高兴兴 -> 高兴 [中文重叠词还原]

明明白白 -> 明白

英文的常见词干还原算法，Porter算法。

Part2、倒排索引

要了解倒排索引，先看一下什么是正排索引。比如有下面两句话：

id1, “搜索引擎提供检索服务”

id2, “搜索引擎是信息检索系统”

正排索引

正排索引就是 MySQL 里的 B+ Tree，索引的结果是：

“搜索引擎是信息检索系统” -> id2

“搜索引擎提供检索服务” -> id1

表示对完整内容按字典序排序，得到一个有序的列表，以加快检索的速度。

倒排索引

第一步 分词

“搜索引擎-提供-检索-服务” -> id1

“搜索引擎-信息-检索-系统” -> id2

第二步 将分词项构建一个词典

搜索引擎

提供

检索

服务

信息

系统

第三步 构建倒排链

搜索引擎 -> id1, id2

提供 -> id1

检索 -> id1, id2

服务 -> id1

信息 -> id2

系统 -> id2

由此，一个倒排索引就完成了，搜索 “检索” 时，得到 id1, id2，说明这两条数据都有，搜索 “服务” 只有 id1 存在。但如果搜索 “检索系统”，此时会先建搜索词按照与构建同一种策略分词，得到 “检索-系统”，两个词项，分别搜索 检索 -> id1, id2 和 系统 -> id2，然后对其做一个交集，得到 id2。同理，通过求并集可以支持更复杂的查询。

以 Lucene 为例，简单说明一下 Lucene 的存储结构。从大到小是Index -> Segment -> Doc -> Field -> Term，类比 MySQL 为 Database -> Table -> Record -> Field -> Value。

Part 3、查询结果排序

搜索结果排序是根据 关键字 和 Document 的相关性得分排序，通常意义下，除了可以人工的设置权重 boost，也存在一套非常有用的相关性得分算法，看完你会觉得非常有意思。

3.面向行的存储有什么优势和缺点？

在数据写入上的对比

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~10ms)，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

行存储 VS 列存储[通俗易懂]

行存储 VS 列存储[通俗易懂]

行存储 VS 列存储[通俗易懂]

行存储 VS 列存储[通俗易懂]

优缺点

显而易见，两种存储格式都有各自的优缺点：

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。

两种存储格式各自的特性都决定了它们的使用场景。

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。

4.请描述Region的三层结构定位过程。

Region：就是要查找的数据所在的Region

.META.：是一张元数据表，记录了用户表的Region信息以及RegionServer的服务器地址，.META.可以有多个regoin。.META.表中的一行记录就是一个Region，记录了该Region的起始行、结束行和该Region的连接信息。

-ROOT-：是一张存储.META.表的表，记录了.META.表的Region信息，-ROOT-只有一个region

Client访问用户数据之前需要首先访问zookeeper，然后访问-ROOT-表，接着访问.META.表，最后才能找到用户数据的位置去访问，中间需要多次网络操作，不过client端会做cache缓存。

步骤：

（1）用户通过查找zk（zookeeper）的/hbase/root-region-server节点来知道-ROOT-表在什么RegionServer上。

（2）访问-ROOT-表，查看需要的数据在哪个.META.表上，这个.META.表在什么RegionServer上。

（3）访问.META.表查看查询的行健在什么Region范围里面。

（4）连接具体的数据所在的RegionServer，这回就真的开始用Scan来遍历row了。

5.简述Region、Hlog、store、memstore和Hfile的功能和作用：

1、Region

table在行的方向上分隔为多个Region。Region是HBase中分布式存储和负载均衡的最小单元，即不同的region可以分别在不同的Region Server上，但同一个Region是不会拆分到多个server上。

Region按大小分隔，表中每一行只能属于一个region。随着数据不断插入表，region不断增大，当region的某个列族达到一个阈值（默认256M）时就会分成两个新的region。

2、Store

每一个region有一个或多个store组成，至少是一个store，hbase会把一起访问的数据放在一个store里面，即为每个ColumnFamily建一个store（即有几个ColumnFamily，也就有几个Store）。一个Store由一个memStore和0或多个StoreFile组成。

HBase以store的大小来判断是否需要切分region。

3、MemStore

memStore 是放在内存里的。保存修改的数据即keyValues。当memStore的大小达到一个阀值（默认64MB）时，memStore会被flush到文件，即生成一个快照。目前hbase 会有一个线程来负责memStore的flush操作。

4、StoreFile

memStore内存中的数据写到文件后就是StoreFile（即memstore的每次flush操作都会生成一个新的StoreFile），StoreFile底层是以HFile的格式保存。

5、 HFile

HFile是HBase中KeyValue数据的存储格式，是hadoop的二进制格式文件。一个StoreFile对应着一个HFile。而HFile是存储在HDFS之上的。HFile文件格式是基于Google Bigtable中的SSTable。

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。Trailer中又指针指向其他数据块的起始点，FileInfo记录了文件的一些meta信息。

6.简述客户端的一个写请求是如何把信息写入到Hfile的，请从Hbase系统架构说起。

1、 HBase 写入流程

从整理架构的视角来看，HBase写入流程整体分为3个阶段：

客户端处理阶段：客户端将用户的写入请求进行预处理，并根据集群元数据定位写入数据所在的RegionServer，将请求发送给对应的RegionServer。

Region写入阶段：RegionServer接收到写入请求之后将数据解析出来，首先写入WAL，再写入对应Region列簇的MemStore。

MemStore Flush阶段：当Region中MemStore容量超过一定阈值，系统会异步执行flush操作，将内存中的数据写入文件，形成HFile。

注意：用户写入请求在完成MemStore的写入后就会返回成功，MemStore Flush是一个异步执行过程。

1、1、 客户端处理流程

客户端处理写入请求的核心流程基本可以概括为三步：

1、1、1、步骤1

用户提交put请求后，HBase客户端会将写入的数据添加到本地缓冲区中，符合一定条件就会通过AsyncProcess异步批量提交。HBase默认设置autoflush=true，表示put请求直接会提交给服务器进行处理；用户可以设置autoflush=false，这样，put请求会首先放到本地缓冲区，等到本地缓冲区大小超过一定阈值（默认为2M，可以通过配置文件配置）之后才会提交。

很显然，后者使用批量提交请求，可以极大地提升写入吞吐量，但是因为没有保护机制，如果客户端崩溃，会导致部分已经提交的数据丢失。

1、1、2、步骤2

在提交之前，HBase会在元数据表hbase:meta中根据rowkey找到它们归属的RegionServer，这个定位的过程是通过HConnection的locateRegion方法完成的。

如果是批量请求，还会把这些rowkey按照HRegionLocation分组，不同分组的请求意味着发送到不同的RegionServer，因此每个分组对应一次RPC请求。

客户端根据写入的表以及rowkey在元数据缓存中查找，如果能够查找出该rowkey所在的RegionServer以及Region，就可以直接发送写入请求（携带Region信息）到目标RegionServer。

如果客户端缓存中没有查到对应的rowkey信息，需要首先到ZooKeeper上/hbase/meta-region-server节点查找HBase元数据表所在的RegionServer。

向hbase:meta所在的RegionServer发送查询请求，在元数据表中查找rowkey所在的RegionServer以及Region信息。客户端接收到返回结果之后会将结果缓存到本地，以备下次使用。

客户端根据rowkey相关元数据信息将写入请求发送给目标RegionServer，RegionServer接收到请求之后会解析出具体的Region信息，查到对应的Region对象，并将数据写入目标Region的MemStore中。

1、1、3、 步骤3

HBase会为每个HRegionLocation构造一个远程RPC请求MultiServerCallable。将请求经过Protobuf序列化后发送给对应的RegionServer。

1、2、Region 写入阶段

数据写入Region的流程可以抽象为两步：追加写入HLog，随机写入MemStore。

服务器端RegionServer接收到客户端的写入请求后，首先会反序列化为put对象，然后执行各种检查操作，比如检查Region是否是只读、MemStore大小是否超过blockingMemstoreSize等。检查完成之后，执行一系列核心操作：

Acquire locks ：获取行锁，HBase中使用行锁保证对同一行数据的更新都是互斥操作，用以保证更新的原子性，要么更新成功，要么更新失败。

Update LATEST\_TIMESTAMP timestamps ：更新所有待写入/更新KeyValue的时间戳为当前系统时间。

Build WAL edit ：HBase使用WAL机制保证数据可靠性，即首先写日志再写缓存，即使发生宕机，也可以通过恢复HLog还原出原始数据。

该步骤就是在内存中构建WALEdit对象，为了保证Region级别事务的写入原子性，一次写入操作中所有KeyValue会构建成一条WALEdit记录。

Append WALEdit To WAL ：将步骤3中构造在内存中的WALEdit记录顺序写入HLog中，此时不需要执行sync操作。当前版本的HBase使用了disruptor实现了高效的生产者消费者队列，来实现WAL的追加写入操作。

Write back to MemStore：写入WAL之后再将数据写入MemStore。

Release row locks：释放行锁。

Sync wal ：HLog真正sync到HDFS，在释放行锁之后执行sync操作是为了尽量减少持锁时间，提升写性能。如果sync失败，执行回滚操作将MemStore中已经写入的数据移除。

结束写事务(Advance mvcc)：此时该线程的更新操作才会对其他读请求可见，更新才实际生效。

1、2、1、 追加写入 HLog

之前的版本中，HLog写入都需要经过三个阶段：首先将数据写入本地缓存，然后将本地缓存写入文件系统，最后执行sync操作同步到磁盘。

当前版本中，HBase使用LMAX Disruptor框架实现了无锁有界队列操作。

(1) 最左侧部分是Region处理HLog写入的两个前后操作：append和sync。

当调用append后，WALEdit和HLogKey会被封装成FSWALEntry类，进而再封装成RingBufferTruck类放入Disruptor无锁有界队列中。

当调用sync后，会生成一个SyncFuture，再封装成RingBufferTruck类放入同一个Disruptor无锁有界队列中，然后工作线程会被阻塞，等待notify()来唤醒。

(2) 最右侧部分是消费者线程，在LMAX Disruptor框架中有且仅有一个消费者线程工作。这个框架会从Disruptor无锁有界队列中依次取出RingBufferTruck对象，然后根据如下选项来操作：

如果RingBufferTruck对象中封装的是FSWALEntry，就会执行文件append操作，将记录追加写入HDFS文件中。需要注意的是，此时数据有可能并没有实际落盘，而只是写入到文件缓存。

如果RingBufferTruck对象是SyncFuture，会调用线程池的线程异步地批量刷盘，刷盘成功之后唤醒工作线程完成HLog的sync操作。

1、2、2、随机写入MemStore

MemStore使用数据结构ConcurrentSkipListMap来实际存储KeyValue，优点是能够非常友好地支持大规模并发写入，同时跳跃表本身是有序存储的，这有利于数据有序落盘，并且有利于提升MemStore中的KeyValue查找性能。

KeyValue写入MemStore并不会每次都随机在堆上创建一个内存对象，然后再放到ConcurrentSkipListMap中，这会带来非常严重的内存碎片，进而可能频繁触发Full GC。HBase使用MemStore-Local Allocation Buffer（MSLAB）机制预先申请一个大的（2M）的Chunk内存，写入的KeyValue会进行一次封装，顺序拷贝这个Chunk中，这样，MemStore中的数据从内存flush到硬盘的时候，JVM内存留下来的就不再是小的无法使用的内存碎片，而是大的可用的内存片段。基于这样的设计思路，MemStore的写入流程可以表述为以下3步。

检查当前可用的Chunk是否写满，如果写满，重新申请一个2M的Chunk。

将当前KeyValue在内存中重新构建，在可用Chunk的指定offset处申请内存创建一个新的KeyValue对象。

将新创建的KeyValue对象写入ConcurrentSkipListMap中。