1.HBase的操作数据的步骤

1、Hbase写数据过程：

Client写入 -> 存入MemStore，一直到MemStore满 -> Flush成一个StoreFile，直至增长到一定阈值 -> 出发Compact合并操作 -> 多个StoreFile合并成一个StoreFile，同时进行版本合并和数据删除 -> 当StoreFiles Compact后，逐步形成越来越大的StoreFile -> 单个StoreFile大小超过一定阈值后，触发Split操作，把当前Region Split成2个Region，Region会下线，新Split出的2个孩子Region会被HMaster分配到相应的HRegionServer 上，使得原先1个Region的压力得以分流到2个Region上由此过程可知，HBase只是增加数据，有所得更新和删除操作，都是在Compact阶段做的，所以，用户写操作只需要进入到内存即可立即返回，从而保证I/O高性能。

2、Hbase读数据：

client->zookeeper->.ROOT->.META-> 用户数据表zookeeper记录了.ROOT的路径信息（root只有一个region），.ROOT里记录了.META的region信息，（.META可能有多个region），.META里面记录了region的信息。

2.HDFS和HBase各自使用场景

> 首先一点需要明白：Hbase是基于HDFS来存储的。

HDFS：

(1)一次性写入，多次读取。

(2)保证数据的一致性。

(3)主要是可以部署在许多廉价机器中，通过多副本提高可靠性，提供了容错和恢复机制。

HBase：

(1)瞬间写入量很大，数据库不好支撑或需要很高成本支撑的场景。

(2)数据需要长久保存，且量会持久增长到比较大的场景

(3)HBase不适用与有join，多级索引，表关系复杂的数据模型

(4)大数据量 （100s TB级数据） 且有快速随机访问的需求。

如：淘宝的交易历史记录。数据量巨大无容置疑，面向普通用户的请求必然要即时响应。

(5)容量的优雅扩展

大数据的驱使，动态扩展系统容量。例如：webPage DB。

(6)业务场景简单，不需要关系数据库中很多特性（例如交叉列、交叉表，事务，连接等等）

(7)优化方面：合理设计rowkey。

3.热点现象及解决办法。

1. **热点现象**：某个小的时段内，对HBase的读写请求集中到极少数的Region上，导致这些region所在的RegionServer处理请求量骤增，负载量明显偏大，而其他的RgionServer明显空闲。
2. **热点现象出现的原因**：HBase中的行是按照rowkey的字典顺序排序的，这种设计优化了scan操作，可以将相关的行以及会被一起读取的行存取在临近位置，便于scan。然而**糟糕的rowkey设计是热点的源头**。 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。 设计良好的数据访问模式以使集群被充分，均衡的利用。
3. **热点现象解决办法**：为了避免写热点，设计rowkey使得不同行在同一个region，但是在更多数据情况下，数据应该被写入集群的多个region，而不是一个。常见的方法有以下这些：

a. **加盐**：在rowkey的前面增加随机数，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。

b. **哈希**：哈希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

c. **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

d. **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。

i. 比如需要保存一个用户的操作记录，按照操作时间倒序排序，在设计rowkey的时候，可以这样设计[userId反转]

[Long.Max_Value - timestamp]，在查询用户的所有操作记录数据的时候，直接指定反转后的userId，startRow是[userId反转][000000000000],stopRow是[userId反转][Long.Max_Value - timestamp]

ii. 如果需要查询某段时间的操作记录，startRow是[user反转][Long.Max_Value - 起始时间]，stopRow是[userId反转][Long.Max_Value - 结束时间]

e. HBase建表预分区：创建HBase表时，就预先根据可能的RowKey划分出多个region而不是默认的一个，从而可以将后续的读写操作负载均衡到不同的region上，避免热点现象。

4.RowKey的设计原则?

1. 总的原则：避免热点现象，提高读写性能；

2. 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。目前操作系统都是64位系统，内存8字节对齐，控制在16个字节，8字节的整数倍利用了操作系统的最佳特性；

3. 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；

4. 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。

5. Rowkey设计结合业务：在满足rowkey设计原则的基础上，往往需要将经常用于查询的字段正和岛rowkey上，以提高检索查询效率。

5.hbase.hregion.max.filesize应该设置多少合适

默认是256，HStoreFile的最大值。如果任何一个Column Family（或者说HStore）的HStoreFiles的大小超过这个值，那么，其所属的HRegion就会Split成两个。

众所周知hbase中数据一开始会写入memstore，当memstore满64MB以后，会flush到disk上而成为storefile。当storefile数量超过3时，会启动compaction过程将它们合并为一个storefile。这个过程中会删除一些timestamp过期的数据，比如update的数据。而当合并后的storefile大小大于hfile默认最大值时，会触发split动作，将它切分成两个region。

6.autoflush=false的影响

无论是官方还是很多blog都提倡为了提高hbase的写入速度而在应用代码中设置autoflush=false，然后在在线应用中应该谨慎进行该设置，原因如下：

(1)autoflush=false的原理是当客户端提交delete或put请求时，将该请求在客户端缓存，直到数据超过2M(hbase.client.write.buffer决定)或用户执行了hbase.flushcommits()时才向regionserver提交请求。因此即使htable.put()执行返回成功，也并非说明请求真的成功了。假如还没有达到该缓存而client崩溃，该部分数据将由于未发送到regionserver而丢失。这对于零容忍的在线服务是不可接受的。

(2)autoflush=true虽然会让写入速度下降2-3倍，但是对于很多在线应用来说这都是必须打开的，也正是hbase为什么让它默认值为true的原因。当该值为true时，每次请求都会发往regionserver,而regionserver接收到请求后第一件事就是写hlog，因此对io的要求是非常高的，为了提高hbase的写入速度，应该尽可能高地提高io吞吐量，比如增加磁盘、使用raid卡、减少replication因子数等。

7对于传统关系型数据库中的一张table，在业务转换到hbase上建模时，从性能的角度应该 如何设置family和qualifier呢?

1、最极端的，①每一列都设置成一个family，②一个表仅有一个family，所有列都是其中的一个qualifier，那么有什么区别呢?

1. 从读的方面考虑：

a. family越多，那么获取每一个cell数据的优势越明显，因为io和网络都减少了。如果只有一个family，那么每一次读都会读取当前rowkey的所有数据，网络和io上会有一些损失。

b. 当然如果要获取的是固定的几列数据，那么把这几列写到一个family中比分别设置family要更好，因为只需一次请求就能拿回所有数据。

2. 从写的角度考虑：

a. 内存方面来说，对于一个Region，会为每一个表的每一个Family分配一个Store，而每一个Store，都会分配一个MemStore，所以更多的family会消耗更多的内存。

b. 从flush和compaction方面说，目前版本的hbase，在flush和compaction都是以region为单位的，也就是说当一个family达到flush条件时，该region的所有family所属的memstore都会flush一次，即使memstore中只有很少的数据也会触发flush而生成小文件。这样就增加了compaction发生的机率，而compaction也是以region为单位的，这样就很容易发生compaction风暴从而降低系统的整体吞吐量。

c. 从split方面考虑，由于hfile是以family为单位的，因此对于多个family来说，数据被分散到了更多的hfile中，减小了split发生的机率。这是把双刃剑。更少的split会导致该region的体积比较大，由于balance是以region的数目而不是大小为单位来进行的，因此可能会导致balance失效。而从好的方面来说，更少的split会让系统提供更加稳定的在线服务。而坏处我们可以通过在请求的低谷时间进行人工的split和balance来避免掉。

2、因此对于写比较多的系统，如果是离线应该，我们尽量只用一个family好了，但如果是在线应用，那还是应该根据应用的情况合理地分配family

8.Hbase行健列族的概念，物理模型，表的设计原则?

1. 行健：是hbase表自带的，每个行健对应一条数据。
2. 列族：是创建表时指定的，为列的集合，每个列族作为一个文件单独存储，存储的数据都是字节数组，其中的数据可以有很多，通过时间戳来区分。
3. 物理模型：整个hbase表会拆分为多个region，每个region记录着行健的起始点保存在不同的节点上，查询时就是对各个节点的并行查询，当region很大时使用.META表存储各个region的起始点，-ROOT又可以存储.META的起始点。
4. rowkey的设计原则：各个列簇数据平衡，长度原则、相邻原则，创建表的时候设置表放入regionserver缓存中，避免自动增长和时间，使用字节数组代替string，最大长度64kb，最好16字节以内，按天分表，两个字节散列，四个字节存储时分毫秒。
5. 列族的设计原则：尽可能少（按照列族进行存储，按照region进行读取，不必要的io操作），经常和不经常使用的两类数据放入不同列族中，列族名字尽可能短。

9.HBase存储单元Cell?

1. 单元{row key，column(=<family>+<label>),version}
2. 唯一性；数据是没有类型的，以字节码形式存储
3. 表：(行key，列族+列名，版本(timestamp))->值

10.说说HBase物理模型?

(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Region；

(3)Region按大小分割的，每个表开始只有一个region，随着数据增多，region不断增大，当增大到一个阈值的时候，region就会等分成两个新的region，之后会有越来越多的region；

(4)Region是HBase中分布式存储和负载均衡的最小单元。不同region分布到不同的regionserver上；

(5)region虽然是分布式存储的最小单元，但并不是存储的最小单元。

Region由一个或者多个store组成，每个store保存一个columns family；

每个store又由一个memstore和0至多个storeFile组成；

Memstore存储在内存中，storefile存储在HDFS上；

(6)每个column family存储在HDFS上的一个单独文件中；

(7)Key和version number在每个column family中都保存一份；

(8)空值不会被保存。

11.HBase的客户端Client?

1. 整个HBase集群的访问入口；
2. 使用HBase RPC机制与HMaster和HRegionServer进行通信；
3. 与HMaster通信进行管理类操作；
4. 与HRegionServer进行数据读写类操作；
5. 包含访问HBase接口，并维护cache来加快对HBase的访问。

12.介绍HBase二级索引?

1. HBase提供了检索数据的功能，不过原有系统仅提供了通过rowkey检索数据的功能，过于单一，不灵活，一旦查询条件改变了往往涉及到要全表扫描过滤，极大浪费机器物理资源，又达不到实时的一个效果。HBase二级索引功能解决了原有HBase系统中仅能够通过rowkey检索数据的问题，使得用户能够指定多种条件，在HBase表中进行数据的实时检索与统计。
2. HBase只提供了一个基于字典排序的主键索引，在查询中只能通过行键查询或者扫描全表来获取数据。
3. HBase基于rowkey的全局索引：Zookeeper的znode节点/hbase/meta-region-server-->hbase:mata

13.每天百亿数据存入HBase，如何保证数据的存储正确和在规定的时间里全部录入完毕，不残留数据?

看到这个题目的时候我们要思考的是它在考查什么知识点?

我们来看看要求：

1）百亿数据：证明数据量非常大

2）存入HBase：证明是跟HBase的写入数据有关

3）保证数据的正确：要设计正确的数据结构保证正确性

4）在规定时间内完成：对存入速度是有要求的

那么针对以上的四个问题我们来一一分析

1）数据量百亿条，什么概念呢？假设一整天60x60x24 = 86400秒都在写入数据，那么每秒的写入条数高达100万条，HBase当然是支持不了每秒百万条数据的，所以这百亿条数据可能不是通过实时地写入，而是批量地导入。批量导入推荐使用BulkLoad方式（推荐阅读：Spark之读写HBase），性能是普通写入方式几倍以上

2）存入HBase：普通写入是用JavaAPI put来实现，批量导入推荐使用BulkLoad

3）保证数据的正确：这里需要考虑RowKey的设计、预建分区和列族设计等问题

4）在规定时间内完成也就是存入速度不能过慢，并且当然是越快越好，使用BulkLoad

14.你知道哪些HBase优化方法

优化手段主要有以下四个方面

1）减少调整

减少调整这个如何理解呢？HBase中有几个内容会动态调整，如region（分区）、HFile，所以通过一些方法来减少这些会带来I/O开销的调整

- Region

  如果没有预建分区的话，那么随着region中条数的增加，region会进行分裂，这将增加I/O开销，所以解决方法就是根据你的RowKey设计来进行预建分区，减少region的动态分裂

- HFile

  HFile是数据底层存储文件，在每个memstore进行刷新时会生成一个HFile，当HFile增加到一定程度时，会将属于一个region的HFile进行合并，这个步骤会带来开销但不可避免，但是合并后HFile大小如果大于设定的值，那么HFile会重新分裂。为了减少这样的无谓的I/O开销，建议估计项目数据量大小，给HFile设定一个合适的值

2）减少启停

数据库事务机制就是为了更好地实现批量写入，较少数据库的开启关闭带来的开销，那么HBase中也存在频繁开启关闭带来的问题。

- 关闭Compaction，在闲时进行手动Compaction

  因为HBase中存在Minor Compaction和Major Compaction，也就是对HFile进行合并，所谓合并就是I/O读写，大量的HFile进行肯定会带来I/O开销，甚至是I/O风暴，所以为了避免这种不受控制的意外发生，建议关闭自动Compaction，在闲时进行compaction

- 批量数据写入时采用BulkLoad

  如果通过HBase-Shell或者JavaAPI的put来实现大量数据的写入，那么性能差是肯定并且还可能带来一些意想不到的问题，所以当需要写入大量离线数据时建议使用BulkLoad

3）减少数据量

虽然我们是在进行大数据开发，但是如果可以通过某些方式在保证数据准确性同时减少数据量，何乐而不为呢?

- 开启过滤，提高查询速度

  开启BloomFilter，BloomFilter是列族级别的过滤，在生成一个StoreFile同时会生成一个MetaBlock，用于

查询时过滤数据

- 使用压缩：一般推荐使用Snappy和LZO压缩

4）合理设计

在一张HBase表格中RowKey和ColumnFamily的设计是非常重要，好的设计能够提高性能和保证数据的准确性

- RowKey设计：应该具备以下几个属性
  - 散列性：散列性能够保证相同相似的rowkey聚合，相异的rowkey分散，有利于查询
- 简短性：rowkey作为key的一部分存储在HFile中，如果为了可读性将rowKey设计得过长，那么将会增加存储压力
- 唯一性：rowKey必须具备明显的区别性
- 业务性：举些例子
  - 假如我的查询条件比较多，而且不是针对列的条件，那么rowKey的设计就应该支持多条件查询
  - 如果我的查询要求是最近插入的数据优先，那么rowKey则可以采用叫上Long.Max-时间戳的方式，这样rowKey就是递减排列
- 列族的设计

  列族的设计需要看应用场景
  - 多列族设计的优劣
    - 优势：HBase中数据时按列进行存储的，那么查询某一列族的某一列时就不需要全盘扫描，只需要扫描某一列族，减少了读I/O；其实多列族设计对减少的作用不是很明显，适用于读多写少的场景
  - 劣势：降低了写的I/O性能。原因如下：数据写到store以后是先缓存在memstore中，同一个region中存在多个列族则存在多个store，每个store都一个memstore，当其实memstore进行flush时，属于同一个region的store中的memstore都会进行flush，增加I/O开销

15.HRegionServer宕机如何处理?

1）ZooKeeper会监控HRegionServer的上下线情况，当ZK发现某个HRegionServer宕机之后会通知HMaster进行失效备援；

2）该HRegionServer会停止对外提供服务，就是它所负责的region暂时停止对外提供服务

3）HMaster会将该HRegionServer所负责的region转移到其他HRegionServer上，并且会对HRegionServer上存在memstore中还未持久化到磁盘中的数据进行恢复

4）这个恢复的工作是由WAL重播来完成，这个过程如下：

- wal实际上就是一个文件，存在/hbase/WAL/对应RegionServer路径下
- 宕机发生时，读取该RegionServer所对应的路径下的wal文件，然后根据不同的region切分成不同的临时文件recover.edits
- 当region被分配到新的RegionServer中，RegionServer读取region时会进行是否存在recover.edits，如果有则进行恢复

16.HBase简单读写流程

读：

找到要读取数据的region所在的RegionServer，然后按照以下顺序进行读取：先去BlockCache读取，若BlockCache没有，则到Memstore读取，若MemStore中没有，则到HFile中读取。

写：

找到要写入数据的region所在的RegionServer，然后将数据先写到WAL中，然后再将数据写到MemStore等待刷新，回复客户端写入完成。

17.HBase和Hive的对比

	HBase	Hive
类型	列式数据库	数据仓库
内部机制	数据库引擎	MapReduce
增删改查	都支持	只支持导入和查询
Schema	只需要预先定义列族，不需要具体到列列可以动态修改	需要预先定义表格
应用场景	实时	离线处理
特点	以K-V形式存储	类SQL

18.HBase与传统关系型数据库(如MySQL)的区别

1. 数据类型：没有数据类型，都是字节数组（有一个工具类Bytes，将java对象序列化为字节数组）。
2. 数据操作：HBase只有很简单的插入、查询、删除、清空等操作，表和表之间是分离的，没有复杂的表和表之间的关系，而传统数据库通常有各式各样的函数和连接操作。
3. 存储模式：Hbase适合于非结构化数据存储，基于列存储而不是行。
4. 数据维护：HBase的更新操作不应该叫更新，它实际上是插入了新的数据，而传统数据库是替换修改
5. 时间版本：Hbase数据写入cell时，还会附带时间戳，默认为数据写入时RegionServer的时间，但是也可以指定一个不同的时间。数据可以有多个版本。
6. 可伸缩性，Hbase这类分布式数据库就是为了这个目的而开发出来的，所以它能够轻松增加或减少硬件的数量，并且对错误的兼容性比较高。而传统数据库通常需要增加中间层才能实现类似的功能

19.什么时候适合使用HBase（应用场景）？

1. 半结构化或非结构化数据:
2. 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。
3. 记录很稀疏:
4. RDBMS的行有多少列是固定的。为null的列浪费了存储空间。而如上文提到的，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。
5. 多版本号数据:
6. 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。
7. 仅要求最终一致性:
8. 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）
9. 高可用和海量数据以及很大的瞬间写入量:
10. WAL解决高可用，支持PB级数据，put性能高
11. 索引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）
12. 业务场景简单:
13. 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。

20.Client会缓存.META.的数据,该数据更新了怎么办?

其实，Client的元数据缓存不更新，当.META.的数据发生更新。比如因为region重新均衡，某个Region的位置发生了变化，Client再次根据缓存去访问的时候，会出现错误，当出现异常达到最大重试次数后，client就会重新去.META.所在的RegionServer获取最新的Region信息，如果.META.所在的RegionServer也变了，Client就会重新去ZK上获取.META.所在的RegionServer的最新地址。

21.请描述如何解决Hbase中region太小和region太大带来的冲突?

Region过大会发生多次compaction，将数据读一遍并重写一遍到hdfs 上，占用io，region过小会造成多次split，region

会下线，影响访问服务，调整hbase.hregion.max.filesize 为256m.

22.解释下 hbase 实时查询的原理

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。

23.HBase 宕机如何处理?

宕机分为 HMaster 宕机和 HRegisoner 宕机。

a. 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。

b. 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。