1. 讲一下HBase的存储结构,这样的存储结构有什么优缺点

	Row Key	column-family1	column-family2	column-family3			
	column1	column1	column1	column2	column3	column1	
	key1	t1:abc t2:gdxdf		t4:dfads t3:hello t2:world			
	key2	t3:abc t1:gdxdf		t4:dfads t3:hello		t2:dfdsfa t3:dfdf	
	key3		t2:dfadfasd t1:dfdasddsf				t2:dfxxdfasd t1:taobao.com

HBase的优点及应用场景:

1. 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。

HBase的缺点:

1. 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
2. 不适合于大范围扫描查询
3. 不直接支持 SQL 的语句查询

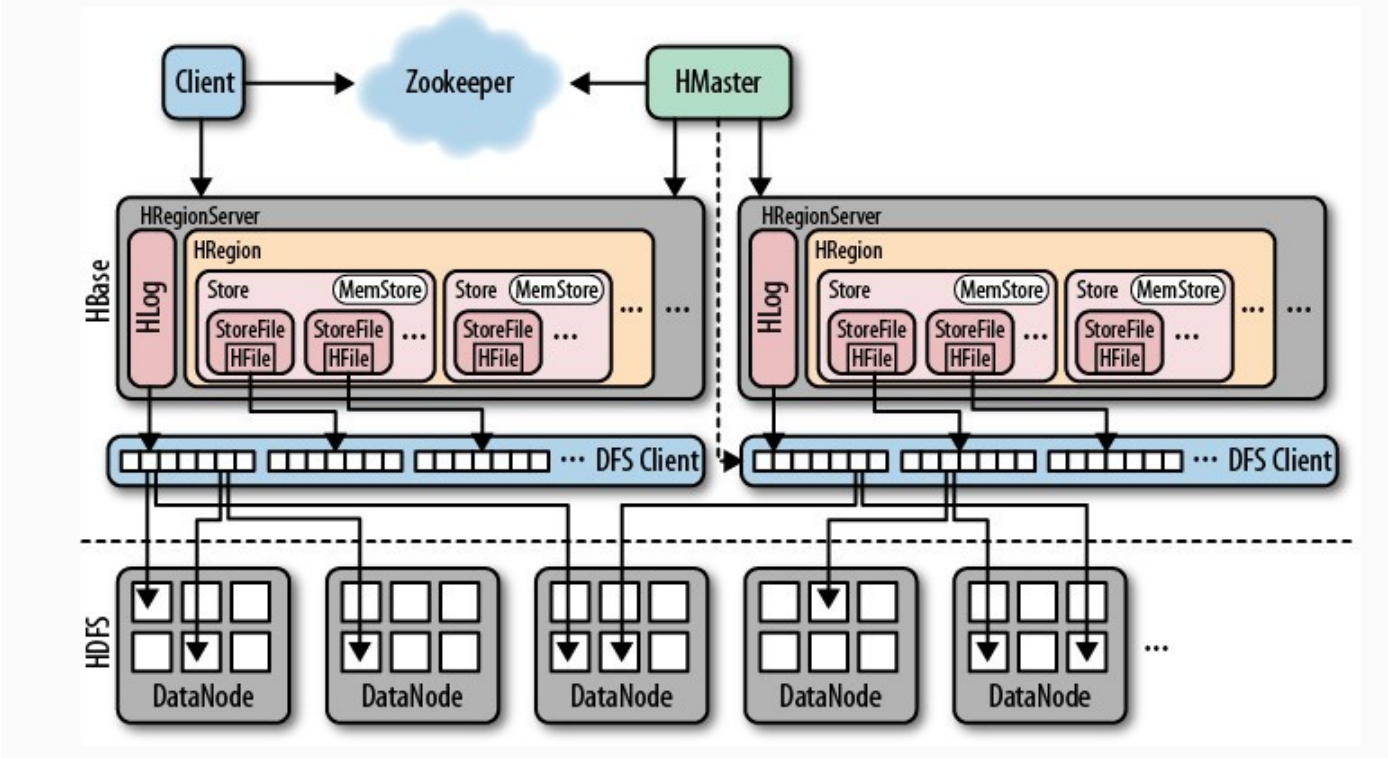
2. 讲一下HBase的写数据的流程

1. Client先访问zookeeper，从.META.表获取相应region信息，然后从meta表获取相应region信息
2. 根据namespace、表名和rowkey根据meta表的数据找到写入数据对应的region信息
3. 找到对应的regionserver 把数据先写到WAL中，即HLog，然后写到MemStore上
4. MemStore达到设置的阈值后则把数据刷成一个磁盘上的StoreFile文件。
5. 当多个StoreFile文件达到一定的大小后(这个可以称之为小合并，合并数据可以进行设置，必须大于等于2，小于10——hbase.hstore.compaction.max和hbase.hstore.compactionThreshold，默认为10和3)，会触发Compact合并操作，合并为一个StoreFile，（这里同时进行版本的合并和数据删除。）
6. 当Storefile大小超过一定阈值后，会把当前的Region分割为两个（Split）【可称之为大合并，该阈值通过hbase.hregion.max.filesize设置，默认为10G】，并由Hmaster分配到相应的HRegionServer，实现负载均衡

3. 讲一下HBase读数据的流程

1. 首先，客户端需要获知其想要读取的信息的Region的位置，这个时候，Client访问hbase上数据时并不需要Hmaster参与（HMaster仅仅维护着table和Region的元数据信息，负载很低），只需要访问zookeeper，从meta表获取相应region信息(地址和端口等)。【Client请求ZK获取.META.所在的RegionServer的地址。】
2. 客户端会将该保存着RegionServer的位置信息的元数据表.META.进行缓存。然后在表中确定待检索rowkey所在的RegionServer信息（得到持有对应行键的.META表的服务器名）。【获取访问数据所在的RegionServer地址】
3. 根据数据所在RegionServer的访问信息，客户端会向该RegionServer发送真正的数据读取请求。服务器端接收到该请求之后需要进行复杂的处理。
4. 先从MemStore找数据，如果没有，再到StoreFile上读(为了读取的效率)。

4. 讲一下 HBase 架构



Hbase主要包含HMaster/HRegionServer/Zookeeper

- **HRegionServer 负责实际数据的读写. 当访问数据时, 客户端直接与RegionServer通信.**

  HBase的表根据Row Key的区域分成多个Region, 一个Region包含这这个区域内所有数据. 而Region server负责管理多个Region, 负责在这个Region server上的所有region的读写操作.

- **HMaster 负责管理Region的位置, DDL(新增和删除表结构)**
  - 协调RegionServer
  - 在集群处于数据恢复或者动态调整负载时,分配Region到某一个RegionServer中
  - 管控集群,监控所有Region Server的状态
  - 提供DDL相关的API, 新建(create),删除(delete)和更新(update)表结构.
- **Zookeeper 负责维护和记录整个Hbase集群的状态**

  zookeeper探测和记录Hbase集群中服务器的状态信息.如果zookeeper发现服务器宕机,它会通知Hbase的master节点.

5. HBase 的HA实现，ZooKeeper在其中的作用

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。
配置HBase高可用，只需要启动两个HMaster，让Zookeeper自己去选择一个Master Acitve即可

zk的在这里起到的作用就是用来管理master节点,以及帮助hbase做master选举

6. HBase数据结构

RowKey

与nosql数据库们一样,RowKey是用来检索记录的主键。访问HBASE table中的行，只有三种方式：

- 通过单个RowKey访问(get)
- 通过RowKey的range（正则）(like)
- 全表扫描(scan)

RowKey行键 (RowKey)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，RowKey保存为字节数组。存储时，数据按照RowKey的字典序(byte order)排序存储。设计RowKey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)。

Column Family

列族：HBASE表中的每个列，都归属于某个列族。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。

Cell

由{rowkey, column Family:columu, 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。

关键字：无类型、字节码

Time Stamp

HBASE 中通过rowkey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。一是保存数据的最后n个版本，二是保存最近一段 时间内的版本（比如最近七天）。用户可以针对每个列族进行设置。

命名空间

命名空间结构如下：

1. Table：表，所有的表都是命名空间的成员，即表必属于某个命名空间，如果没有指定，则在default默认的命名空间中。
2. RegionServer group：一个命名空间包含了默认的RegionServer Group。
3. Permission：权限，命名空间能够让我们来定义访问控制列表ACL（Access Control List）。例如，创建表，读取表，删除，更新等等操作。
4. Quota：限额，可以强制一个命名空间可包含的region的数量。

7. HBase 如何设计rowkey

- **RowKey长度原则**

  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：

  - 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；
  - 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。
  - 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。
- **RowKey散列原则**

  如果Rowkey是按时间戳的方式递增，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，由程序循环生成，低位放时间字段，这样将提高数据均衡分布在每个Regionserver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将产生所有新数据都在一个RegionServer上堆积的热点现象，这样在做数据检索的时候负载将会集中在个别RegionServer，降低查询效率。

- **RowKey唯一原则**

  必须在设计上保证其唯一性。

注：资料来源于网络。