1、Hadoop的主要组件：

HDFS负责整个分布式文件的存储

MapReduce专门做离线计算和批处理，MapReduce是基于磁盘计算的，MapReduce在做数据处理的时候都是要把数据写到磁盘的，处理结束后还要把结果写入到分布式文件系统中去。

YARN负责调度内存、CPU、带宽等计算资源

Tez是运行在YARN之上的下一代Hadoop查询处理框架，Tez会把很多的MapReduce作业进行分析优化

Hive是数据仓库，支持SQL语句，

2、Hadoop核心组件：HDFS和MapReduce

HDFS核心组件NameNode和DataNode

**NameNode就像一个目录服务器一样，应用来取数据的时候，首先去访问这个NameNode，来获取我们所要的数据存储在哪个DataNode上面**，就获取数据存储的具体位置后，再去具体的节点上面去取数据。对于分布式文件系统HDFS而言，NameNode是总管家，他要管理各种元数据并提供服务。NameNode里面的元数据都保存在内存中，NameNode会告诉你一个文件被分成多少块，每个块都保存在哪个节点上面，他会建立一个映射表，这些映射表的元数据都保存在内存中。

**名称节点中保存的元数据有两个核心的数据结构：文件系统树的镜像文件FsImage，日志文件EditLog，这两个文件都是保存在磁盘上的。**

**FsImage用于保存文件树以及文件树中所有文件和文件夹的元数据，包括文件的复制等级，修改访问时间、访问权限，块大小以及组成文件的块**，通过FsImage就知道整个文件目录是什么样子的，需要注意的是，FsImage没有具体记录块在哪个数据，刚才说的名称节点保存了块存储到哪个节点上，这些信息没有保存在FsImage里面，而是单独的在内存中的一个区域维护（映射信息），映射信息会告诉你一个大的文件节点上到底包含了哪些块，一个块到底被放到哪个数据节点。**EditLog告诉你在系统运行的过程中，你对那些数据进行了创建、删除、复制等操作记录。**

**DataNode主要就是存储数据。数据节点的数据最终是要存储到磁盘上去的，也就是保存在数据节点所在的Linux系统中**。

当一个数据节点加入到集群中后，数据节点会向管家节点即名称节点汇报自己保存了哪些数据块信息，名称节点根据这些信息构建一个清单，它就知道这些数据块到底保存在哪个节点上，所以这些信息不是通过FsImage来保存的，而是通过名称节点和数据节点在运行过程中不断的沟通来实时的维护这些信息的，而且这些信息都是保存在内存当中的映射信息上。

每次当名称节点启动以后，他要把FsImage从底层磁盘上加载到内存中，和EditLog的各项操作进行合并，因为FsImage里面记录的是历史的相关的元数据的信息，EditLog记录的是系统运行期间你对元数据的修改操作记录日志。只有把这两个数据结构进行合并，你才能得到最新的元数据信息。名称节点会把这个新版的FsImage进行保存，然后把旧版的FsImage进行删除，再新建一个空白的EditLog，系统启动以后就是按照这个模式来运行的。以后随着新的数据的修改、删除，新的数据结构的修改全部都保存在EditLog里面。

也就是说，只有在每次启动Namenode时，才会把edits中的操作增加到fsimage中，并且把edits清空。所以fsimage总是记录启动Namenode时的状态，而edits在每次启动时也是空的，它只记录本次启动后的操作日志。

为什么不把每次修改的元数据信息直接保存在FsImage里面，而要另外设置个EditLog呢？因为对于大规模的HDFS文件系统来说，文件规模较大，每次发生数据修改都对FsImage进行修改，会导致系统运行较慢。因此HDFS做了一个优化处理，就是把每次的更新单独记录在EditLog，因为EditLog也是在内存中，规模较小，所以运行效率较高，但是也存在一个问题，就是随着数据的不断修改，EditLog也会不断的增大，从而影响系统的性能。这个时候怎么处理呢？就是用第二名称节点（SecondaryNameNode），当然第二名称节点也有对名称节点冷备份的作用。

第二名称节点和名称节点会定期进行通信，在某个阶段会请求名称节点停止使用EditLog，名称节点收到请求后会停止使用EditLog，会生成一个edits.new，也就是一个新的EditLog，然后把旧的EditLog放在一边，让第二名称节点取走，第二名称节点会通过Http Get方式把FsImage和EditLog下载到本地，然后进行合并得到一个新的FsImage，并把它重新发送给名称节点，在此期间，所有数据更新全部记录到edits.new，合并完以后再把edits.new改为EditLog，这样即解决了EditLog不断增大的问题，也解决了冷备份的问题。

**HDFS的Checkpoint**

Snapshot（快照）：在数据库或者文件系统中，一个快照表示对当前[系统状态](https://www.baidu.com/s?wd=%E7%B3%BB%E7%BB%9F%E7%8A%B6%E6%80%81&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的一个备份，当系统发生故障时，可以利用这个快照将系统恢复到产生快照时的样子。

Checkpoint（检查点）：因为数据库系统或者像HDFS这样的分布式文件系统，对文件数据的修改不是直接写回到磁盘的，很多操作是先缓存到内存的Buffer中，当遇到一个检查点Checkpoint时，系统会强制将内存中的数据写回磁盘，当然此时才会记录日志，从而产生持久的修改状态。

对于这两者的区别，个人认为主要是Snapshot是对数据的备份，而Checkpoint只是一个将数据修改持久化的机制。

那么对于HDFS来说，这两个技术是如何实现的呢？

## 1. Snapshot

首先来看Snapshot，很不幸，目前为止最新的Hadoop2.0.5-alpha也没有支持这一功能，个人感觉实现的可能性不大。因为HDFS是用来存储大数据的，对于这么大规模的数据，即便是采用增量备份的方式，也会带来很大的存储开销。不知道以后的Hadoop版本是否会支持，[拭目以待](https://www.baidu.com/s?wd=%E6%8B%AD%E7%9B%AE%E4%BB%A5%E5%BE%85&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)。

## 2. Checkpoint

对于Checkpoint，hadoop已经很好地支持了，不过它是将Checkpoint用在了Namenode元数据的持久化上。

### 2.1 Namenode

在介绍Checkpoint之前，先来看看Namenode上面有些什么数据：

edits：HDFS操作的日志记录，每次对HDFS进行修改操作后，都会往edits中记录一条日志；

fsimage：HDFS中命名空间、数据块分布、文件属性等信息都存放在fsimage中；

Namenode中主要就包含上面两个文件，HDFS是怎样处理这两个文件的呢？

前面已经说来，edits是在每次修改HDFS时都会插入记录，那么fsimage则在整个HDFS运行期间不会产生变化，用HDFS官方文档的说法就是：NameNode merges fsimage and edits files only during start up。（引自：<http://hadoop.apache.org/docs/stable/hdfs_user_guide.html>）也就是说，只有在每次启动Namenode时，才会把edits中的操作增加到fsimage中，并且把edits清空。所以fsimage总是记录启动Namenode时的状态，而edits在每次启动时也是空的，它只记录本次启动后的操作日志。

### 2.2 Checkpoint触发时机

说了这么多，还没有涉及到Checkpoint呢，别急。我们先来分析一下为什么需要Checkpoint？

按照fsimage和edits的工作机制，在一次启动后，edits的文件可能会增长到很大，这样在下次启动Namenode时需要花费很长时间来恢复；另一方面，如果在HDFS运行过程中发生Namenode的故障，那么edits中的记录就会丢失。所以，我们需要利用Checkpoint即使将修改操作持久化。

在配置文件中有两个参数：

fs.checkpoint.period：设置两次相邻checkpoint之间的时间间隔，默认是1小时；

fs.checkpoint.size：设置一个edits文件大小的阈值，达到这个阈值，就强制执行一此checkpoint（即使此时没有达到period的时限），默认为64MB。

**也就说触发HDFS中Checkpoint的机制有两种，一是时间，另一个是日志记录的大小。**

### 2.3 Checkpoint执行过程

Chekpoint主要干的事情是，将Namenode中的edits和fsimage文件拷贝到Second Namenode上，然后将edits中的操作与fsimage文件merge以后形成一个新的fsimage，这样不仅完成了对现有Namenode数据的备份，而且还产生了持久化操作的fsimage。最后一步，Second Namenode需要把merge后的fsimage文件upload到Namenode上面，完成Namenode中fsimage的更新。

以上提到的文件都可以在hadoop系统的data目录下找到。

### 2.4 Checkpoint导入

当Namenode发生故障丢失元数据后，可以利用Second Namenode进行导入恢复，过程如下：

在Namenode的节点上面创建dfs.name.dir指定的目录；

指定配置文件中的fs.checkpoint.dir（应该是hdfs-site.xml文件）；

启动Namenode时带上选项 -importCheckpoint。

Namenode首先会将fs.checkpoint.dir中的文件拷贝到dfs.name.dir中，如果此时dfs.name.dir中已经包含了合法的fsimage文件（也就是Namenode没有发生元数据丢失却执行了导入操作），那么Namenode就会执行失败。否则，Namenode会检测导入的fsimage文件是否与文件系统中的数据一致，若一致则成功完成导入恢复。

另外一点，在HDFS官方文档中没有提到，就是Checkpoint对于edits文件很大时启动Namenode时间长的问题，因为需要花费更长的时间去将edits日志merge到fsimage中。我个人认为是否可以利用Second Namenode中的fsimage进行启动呢，也就是说，直接将Second Namenode中的image文件拷贝到Namenode中，这就节省了merge的时间。这样做可以吗？

后来仔细想想，貌似行不通。因为Second Namenode中的image文件不能保证是最新的，它只有在触发Checkpoint时才更新，而启动Namenode需要将edits中全部的操作都merge到fsimgae中。另外，即使拷贝Second数据的方案可行，也不能保证启动时间会减少，因为虽然节省Namenode本地merge操作的时间，却增加了网络数据传输的时间，两者大小还不确定。所以综合这两点原因，此方法不行。

解决方法为：

第二名称节点和名称节点会定期进行通信，在某个阶段会请求名称节点停止使用EditLog，名称节点收到请求后会停止使用EditLog，会生成一个edits.new，也就是一个新的EditLog，然后把旧的EditLog放在一边，让第二名称节点取走，第二名称节点会通过Http Get方式把FsImage和EditLog下载到本地，然后进行合并得到一个新的FsImage，并把它重新发送给名称节点，在此期间，所有数据更新全部记录到edits.new，合并完以后再把edits.new改为EditLog，这样即解决了EditLog不断增大的问题，也解决了冷备份的问题。

## 3. Recovery

除了上面介绍的Checkpoint机制，Namenode还提供了一种Recovery方式。

在配置文件hdfs-site.xml中有两个参数:

1. dfs.namenode.name.dir：指定namenode中元数据的本地存储位置，这里的位置可以指定多个，并且用逗号分割；
2. dfs.namenode.name.dir.restore：布尔变量，若为true，在每次checkpoint的时候会自动恢复丢失数据的dir。

可见，Recovery只是在本地存储了多个元数据的备份，虽说可以避免文件损害带来的故障，但是对于节点故障来说，本地的多个备份都将不能使用，所以还是有一定局限性的。不过一般情况下都会把多个本地备份设置在不同的物理磁盘分区上面，当一个分区坏掉以后可以使用另一个。

至于如何利用这些本地备份进行recovery

3、MapReduce核心组件JobTracker和TaskTracker

**JobTracker负责对整个作业进行管理，他会把一个用户的大作业拆分成很多小作业，然后把这些作业分发到不同的机器上面**，每个机器上面都部署了TaskTracker，**TaskTracker会对分配给它的这些作业进行跟踪和执行。TaskTracker就是负责完成分配给它的那一小部分作业，在内存中完成计算。**

4、HDFS相关概念

块：是为了分摊磁盘读写开销，块的默认大小是64MB，也可以设置成更大的。

HDFS为何会有块的设计，因为他要支持大规模数据存储，同时也是为了降低磁盘的寻址开销。访**问HDFS文件，要经过三级寻址，首先要找到元数据目录，然后找到存储数据的节点，最后从数据节点上读取数据**。块设计的太小会导致磁盘寻址开销过大，块设计的过大，虽然磁盘寻址开销小了，但会影响到MapReduce的设计，使MapReduce的并行度降低。

HDFS采用这种抽象块的好处：（1）突破单机存储文件的限制，可以把文件分块后分布存储到不同机器上。（2）可以简化系统设计，通过块的设计，很方便元数据的管理。（3）以块为单位适合备份。

5、**HDFS存储原理：冗余数据保存问题、数据保存问题、数据恢复问题**

**数据冗余保存**：HDFS作为分布式文件系统，他把整个底层架构在比较廉价的机器集群上面，这些机器很可能会出现故障，所以数据的冗余保存是很有必要的，数据备份以块为单位，一般数据的冗余因子为3，就是每个数据块保存3份。

好处：（1）加快数据传输速度，可以实现并行传输。（2）可以很容易检查错误，因为三个副本形成相互对照，其中一个出现错误，会很容易检查得到。（3）保证数据可靠性，当系统周期探测到有一个副本出现错误时，会自动重新生成副本，达到要求的副本数量。

**数据存放策略**：当一个块到达系统后，如何一模一样的放三份？如果提交请求来自于集群内部，第一份放在上传文件的数据节点上，这样就不用通过内部网络把数据传输到其他节点上。如果提交请求来自于集群外部，那便随便找一个磁盘不太满，CPU不太忙的节点，在上面放置第一个副本。第二个副本放置在与第一个副本不同的机架上面的节点上，第三个副本放置在与第一个副本相同的机架上的不同节点上。如果有更多的副本，则采用随机放置的方法。

**数据读取策略**：读取数据的一个基本原则就是就近读取，HDFS提供一个API可以获取数据节点所属的机架ID，客户端也可以调用API获取自己所在机架ID，客户端首先会去寻找与自己ID相同的数据节点，ID相同则表示位于同一个机架内，这个时候就会优先选择这个节点去读取，若没有找到相同ID的数据节点，则随机读取。

**数据的错误与恢复：名称节点出错、数据节点出错、数据本身出错**

**名称节点出错**：

名称节点保存了最核心的两个数据结构FsImage和EditLog，这两个数据结构是访问分布式文件系统HDFS最基本的数据结构，没有他们根本无法访问HDFS，平时对他们有一个冷备份，就是第二名称节点（SecondaryNameNode），当名称节点出现错误后，他会对外暂停一段时间服务，然后从第二名称节点上获取相应的元数据信息进行恢复，恢复以后再对外提供服务。HDFS1.0出现这个错误后会暂停服务一段时间，HDFS2.0出现这个错误后马上就可以热备，不存在这个问题。

**数据节点出错**：

数据节点是负责具体数据存储的，在整个运行期间，数据节点会不断地给名称节点发送心跳信息，告诉名称节点，他还活着，一旦隔了一个周期，名称节点没有收到数据节点的心跳信息，就会把该节点标记为宕机。然后把凡是在这个机器上的数据全部复制一份到另一台机器上，所以是很容易恢复的。HDFS分布式文件系统与其他文件系统最大的区别还在于可以调整冗余数据的位置，不仅当发生故障的时候可以调整数据冗余位置，当负载不均衡的时候，，也可以调整冗余数据的位置。

**数据本身出错**

数据本身出错也是在所难免的，客户端在读取数据的时候，会对它的数据进行校验码验证，如果发现校验码不对，则说明数据发生错误。校验码是整个文件创建时生成的，客户端每次往里面写数据的时候，都会生成一个校验码，校验码和文件在同一目录下进行存储，下次读数据的时候，会把数据块和校验码一起读，读到以后对这个数据块的校验码进行计算，与上次的校验码进行比较，要是不同的话，则说明数据发生了错误，他要进行恢复，进行冗余副本复制。

6、分布式数据库HBase

HBase是一个分布式文件系统，属于分布式数据库，主要存储半结构化和非结构化的松散数据。

结构化数据：是指可以使用关系型数据库表示和存储，表现为二维形式的数据。一般特点是：数据以行为单位，一行数据表示一个实体信息，每一行数据的属性是相同的。举一个例子

id name age gender

1 lyh 12 male

2 liangyh 13 female

3 liang 18 male

所以结构化数据的存储和排列是很有规律的，这对数据的查询和修改是很有帮助的，但是扩展性不好。

半结构化数据是结构化数据的一种形式，它并不符合关系型数据库或其他数据表的形式关联起来的数据结构，但包含相关标记，用来分割语义元素以及对记录和字段进行分层，因此他也被称为自描述结构。

半结构化数据，属于同一类实体可以有不同的属性，即使他们组合在一起，这些属性的顺序并不重要。常见的半结构化数据有XML和JSON，半结构化数据的扩展性很好。

非结构化数据：顾名思义就是没有固定结构的数据，例如图片、视频/音频、文档等都是非结构化数据，对于这类数据，我们一般直接整体进行存储，而且一般存储为二进制的数据格式。

底层分布式文件系统存储完全非结构化的数据，HBase可以比较好的支持半结构化数据，也就是松散数据的存储，他有很好的水平可扩展性。HBase是架构在底层分布式文件系统HDFS之上的，同时MapReduce可以对HBase中的数据进行处理，Pig、Hive等组件也可以访问HBase中的数据。

1. 数据类型

传统的关系数据库有很多的数据类型，但对于HBase来讲，它的数据类型非常简单，他就是把存储的值存储为未经解释的字符串。

1. 数据操作

在关系型数据库中，定义了非常多的数据操作，增删改查等操作，而HBase当中就没有那么多的操作，HBase在设计的时候，他不会对这些数据进行规范化，数据都存在一张完整的表里，不需要进行多表连接查询，所以效率非常高。

1. 存储模式

关系数据库是基于行存储的，HBase数据库是基于列存储的。

1. 数据索引

关系数据库一般来说可以针对不同的列构建复杂的索引，一般一个关系数据库都有一个主索引，同时也会构建很多的二级索引，也叫辅助索引，来满足相关数据的查询需求，可以快速定位到不同的列。HBase在设计之初，是没有那么多的索引的，它只支持对不同行键进行简单索引，不支持对列索引。

1. 数据维护

关系数据库在数据更新的时候，一般会把旧的值给替换掉，但HBase不会，他在进行数据更新的时候，仍然会保留旧版本的值，只有到一定时间以后，超过你设置的参数，在系统后台进行清理的时候，才会把旧的版本给清理掉。

1. 可伸缩方面

HBase不是在一台机器上进行存储，是完全借助分布式集群来进行分布式存储，所以它的水平可扩展性非常好。

7、HBase数据模型

HBase是一个稀疏的，多维度的排序映射表。他是通过四个元素行键、列族、列限定符（列）、时间戳去定位一个数据的，在HBase当中，每一个值都是一个未经解释的字符串，也就是Bytes数组，程序员要负责对它的解析。用户在表中存储数据的时候，一行可以有一个行键和任意多列，对于一个表来说，在水平方向，可以有一个或多个列族。列族是HBase里面一个非常重要的概念，列族支持动态扩展，可以很容易的对列族进行动态扩展，增加、减少列族都可以，另外一个特性就是在数据修改的时候，可以存储多个版本。这是因为HBase底层是架构在分布式文件系统HDFS之上的，它采用比较简单的数据模型，只允许追加，不允许修改，所以在数据修改的时候，会生成新的版本，标记时间戳，通过时间戳找到最新的版本。

HBase在组织数据的时候也是以表的形式，且它避免了单表查询，以一个大表进行存储，列族实际上是HBase存储的基本单元，不同列族的数据存储到不同的文件里面。

8、HBase实现原理

**HBase最核心的组件有三个：一个库函数、一个Master和很多个Region。**

库函数一般用于链接客户端，用户客户端能够通过这些库函数来访问数据，Master服务器主要充当管家的作用。Region服务器负责存储不同的Region。通过Master服务器就可以对各个分区的信息进行管理，同时Master服务器还维护一个Region服务器列表，通过Master服务器可以知道哪个Region服务器在工作，哪些有了故障。同时还负责对Region进行分区，一个表会被分成多个分区，哪个分区被分到哪个Region服务器都是由Master负责，Master还要负责负载均衡。

Region服务器就是负责存储不同的Region。每个表都会被分成不同的Region，这些Region都是由不同的Region服务器负责维护和管理。客户端在访问数据的时候，也是直接和Region服务器打交道的，在获取相应Region的位置后，直接和相应的Region服务器打交道，进行数据存取。

客户端并不依赖Master去获取Region位置信息，客户端获取数据的时候，可以不通过Master就获得相应数据的具体位置，然后直接去存取数据。HBase是典型的数据库，会为每个应用都设计一个表，在刚开始的时候，因为数据比较小，只有一个Region，随着数据规模的慢慢变大，Region也会慢慢变大，增大到一定程度就会进行分裂，一个Region分裂成两个Region，在分裂的时候并不是物理分裂以后再进行数据拆分。而是在分开的瞬间，修改了数据的指向，实际访问的时候还是访问的旧的Region，这样可以保证快速拆分。后台会运行一个合并程序，对拆分的数据进行重新操作，最终会写到一个新的文件当中去，只有这些新的文件写完之后，才会告诉你新的Region信息，以后再访问的话就访问这些新的文件。在Region拆分的时候，多个Region会被分到不同的Region服务器上，但是对于同一个Region是绝对不会拆分到不同的Region服务器上的。

Region定位

当一个Region被拆分成多个Region时，他会被打散，随机分布在不同地方去存储。我们如何定位每一个Region被存储在哪个地方？HBase设计了一个三层结构实现Region的寻址和定位，它构建一个元数据表，这个元数据表有两列，一列是Region的id，第二列是目标服务器的id。通过这样一个映射表，就可以知道Region id被分到了那个Region服务器上。HBase在最初设计的时候，就建立了一个映射表.META.表，随着数据的增多，映射条目的增多，仅靠一个Region是无法存储.META.表的，这个时候.META.表也要进行分区，需要多个Region进行存储，同样，这时候分区的.META.表如何寻找，也是建立映射表，第一列是.META.表的Region id，第二列是存放目标Region服务器的id。这个映射表叫做-ROOT-

表，而且我们规定随着-ROOT-表不断增大，也不允许-ROOT-表进行分裂，即一个-ROOT-表必须存储在一个Region上。

所以HBase三层架构中，先是-ROOT-表，再是.META.表，最后是用户表。

-ROOT-表必须有地址，-ROOT-表只有一个，所以它的地址已经在程序中写死了，写死在Zookeeper服务器中，客户端在访问数据的时候，先去Zookeeper服务器找到-ROOT-表，然后根据-ROOT-表找到.META.表地址，再根据.META.表找到用户数据表地址，最后把数据取出来，这就是HBase的三层结构。

为了加速寻址，客户端一般会把这些寻址信息给缓存下来，第一次要经历三级寻址，当找个数据的位置后，客户端会把这次的寻址信息给缓存下来，下次再去取数据的时候，就不需要经过三级寻址了，这样加快寻址效率。但当数据发生改变的时候，缓存就会失效。所以HBase采用的机制是，当把数据的Region信息缓存之后，就一直使用这个缓存，当使用这个缓存找不到相应数据的时候，就重新经历三级寻址，找到数据的Region后，就再把该信息缓存下来。

HBase存储数据并不是直接和磁盘打交道的，而是借助于底层的HDFS分布式文件系统来存储的。这里有几个核心组件，一个Zookeeper服务器，一个Master服务器，另外一个是访问数据的客户端以及存储数据的Region服务器。Zookeeper负责整个集群在任何时候都只有一个Master在运行，Master负责HBase当中的表和Region的管理工作，比如对表的增删改查都是通过Master来实现的，同时还负责Region服务器的负载均衡。Region服务器就是负责具体数据的存储，访问数据的时候就是直接和Region服务器打交道。

整个Region服务器集群是由多个Region服务器构成的，一个Region服务器上可以有10到1000个Region，Region就是具体的数据。这些Region共用一个日志文件HLog，每一个Region在写数据的时候，又会按照列族进行切分，列族是HBase存储数据的基本单位，每一个列族会存储为一个Store文件，列族Store当中的数据不是直接写到底层的，而是先写到缓存MemStore当中，在写缓存的时候，为了安全性和防止数据丢失，也需要写日志HLog，HLog是预写式日志，每次写MemStore的时候，先写日志，而且也得保证日志全部写到磁盘中时，才返回给客户端。当缓存满了之后，再刷写到磁盘StoreFile中，StoreFile是HBase当中的形式。StoreFile是存储在底层HDFS当中的，所以每一个StoreFile是通过HDFS里面的HFile去存储的。每次刷写的时候都会生成一个StoreFile文件，如果数量非常多的话，会影响查找速度，一般当StoreFile达到一定数量后，就对StoreFile进行合并。这时候又会出现一个问题，随着StoreFile的合并，会变得越来越大，所以当达到一定阈值的时候，又会进行分裂，分裂以后，一个Region分成两个。

用户读数据的时候，也是先去缓存MemStore里面去读取，因为最新的数据都在MemStore里面，如果缓存找不到，再到StoreFile里面去寻找。

9、HDFS体系结构设计的局限性

（1）命名空间限制（2）性能瓶颈（3）隔离问题（4）集群可用性

10、HLog的工作原理

HLog保障系统在出错时候能够恢复，系统恢复采用最经典的方法就是采用日志方法，通过日志来恢复相关的数据。

Zookeeper负责监视集群，会告诉你哪个服务器出现了故障，也会通知给相关的Master，Master知道哪个服务器出现故障后，就会把故障服务器上的数据进行迁移，Master去处理故障的时候，会把故障服务器上遗留的HLog进行复制，HLog里面记录了故障服务器上日志记录，怎么更新，怎么修改，怎么删除等等，都在日志文件中记录。故障服务器上就一个HLog文件，但Region有许多个，在恢复的时候如何恢复，HBase会把故障服务器上的Hlog拉取过来，然后进行拆解，把属于每个Region的日志相应的分配给它。因为故障服务器上的Region不能用了，于是就重新寻找一个新的Region服务器，然后按照Hlog日志重新操作一遍。

Spark具体的程序执行过程

MapReduce执行过程

两个文件中有不同的字符串，如何统计这两个文件中一样的字符串。

人工智能平台的比较

11、HBase性能优化

（1）可以用“系统最大值-时间戳”作为行键，这样在时间上比较近的数据可以放在一起，可以保证最新写入的数据最快被命中。

（2）在创建表的时候通过设置HColumnDescriptor.setInMemory选项为True，这样就可以把相关的Region服务器中的表放入到缓存中去，提高读写速度。

（3）通过HColumnDescriptor.setMaxVersions设置最大版本数，比如我们只需要保存最新版本，我们就可以设置最大版本数为1。这样他就不会在系统中保留其他版本，节省磁盘空间。

（4）我们也可以为数据设置生命周期TimeToLive，如果这些数据一旦超过这些声明周期，就会自动被删除。