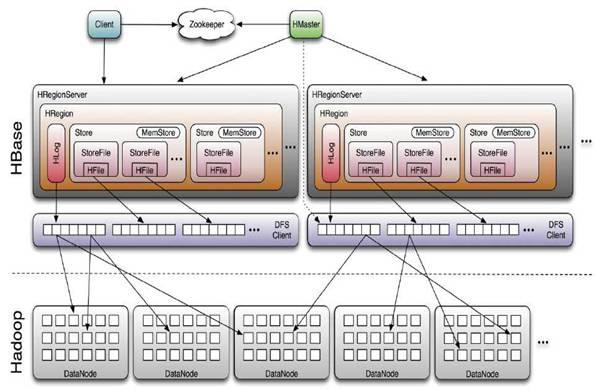
**Hbase写数据，存数据，读数据的详细过程**

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。

**Hbase写数据和存数据的过程**



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。

**对上述流程的补充：**

补充1：HStore存储是HBase存储的核心，其中由两部分组成，一部分是MemStore，一部分是StoreFiles。

补充2：HLog的功能：

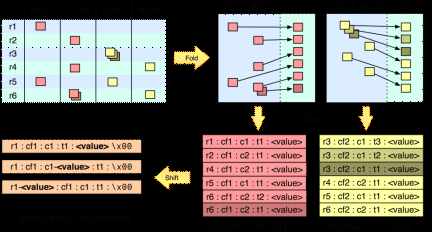
在分布式系统环境中，无法避免系统出错或者宕机，一旦HRegionServer以外退出，

MemStore中的内存数据就会丢失，引入HLog就是防止这种情况。

工作机制：每 个HRegionServer中都会有一个HLog对象，HLog是一个实现Write Ahead Log的类，每次用户操作写入Memstore的同时，也会写一份数据到HLog文件，HLog文件定期会滚动出新，并删除旧的文件(已持久化到 StoreFile中的数据)。当HRegionServer意外终止后，HMaster会通过Zookeeper感知，HMaster首先处理遗留的 HLog文件，将不同region的log数据拆分，分别放到相应region目录下，然后再将失效的region（带有刚刚拆分的log）重新分配，领取到这些region的 HRegionServer在Load Region的过程中，会发现有历史HLog需要处理，因此会Replay HLog中的数据到MemStore中，然后flush到StoreFiles，完成数据恢复。

补充3：Region就是StoreFiles，StoreFiles里由HFile构成，Hfile里由hbase的data块构成，一个data块里面又有很多keyvalue对，每个keyvalue里存了我们需要的值。

补充4：



我们观察上面这一幅图：

一 张表，有两个列族（红颜色的一个，黄颜色的一个），一个列族有两个列，从图中可以看出，这就是列式数据库的最大特点，同一个列族的数据在在一起的，我们还 发现如果是有多个版本，同时也会存多个版本。最后我们还发现里面存了这样的值：r1:键值，cf1:列族的名字，c1:列明。t1：版本号，value值 （最后一幅图说明的是value值可以存放的位置）。通过这样的看法，我们发现如果我们设计表的时候把这几个东西：r1:键值，cf1:列族的名 字，c1:列明的名字取短一点是不是我们会省出好多存储的空间！

还有，我们从这一幅图中还应该得到这样的认识：

我 们看倒数第二张图，字段筛选的效率从左到右明显下降，所以在keyvalue的设计时用户可以考虑把一些重要的筛选信息左移到合适的位置，从而在不改变数 据量的情况下，提高查询性能。那么简单的说就是用户应当尽量把查询维度或信息存储在行健中，因为它筛选数据的效率最高。

得到上面的认识后，我们应该还要会有这样的觉悟：

HBase 的数据存储时会被有顺序的存储到一个特定的范围，因为我们存储的时候一般都是按顺序的，所以会一直存到同一个region上，由于一个region只能由 一个服务器管理，这样我们老是添加到同一个region上，会造成读写热点，从而使集群性能下降。那么解决这个的办法还是有的，我能想到的就是，比如我们 有9台服务器，那么我们就回去当前时间，然后摸9，加到行健前缀，这样就会被平均的分到不同的region服务器上了，这样带来的好处是，因为相连的数据 都分布到不同的服务器上了，用户可以多线程并行的读取数据，这样查询的吞吐量会提高。

关于我们版本的控制，我们要么就让多台服务器上的时间都同步，要么干脆就在put插入数据的时候，就设置一个客户端的时间戳来代替。（因为我们要是不显示的添加，人家就给我们在自己的服务器上添加了自己的时间了。）

补充5：

设 计表的时候，有两种设计方式，一种是高表设计，一种是胖表设计。根据HBase的拆分规则，我们的高表设计更容易拆分（使用组合键），不过，如果我们设计 成胖表，而我们的这个胖里的数据需要经常修改，这样设计是很合理的，因为我们的HBase保证了行级的原子性，如果设计成高表，反而就不合适了，因为不能 保证跨行的原子性。

补充6：

写缓存

每 一个put的操作实际上是RPC的操作，它将客户端的数据传送到服务器然后返回，这只适合小数据量的操作，如果有个应用程序需要每秒存储上千行数据到 HBase表中，这样处理就不太合适了。HBase的ＡＰＩ配备了一个客户端的写缓冲区，缓冲区负责收集ｐｕｔ操作，然后调用RPC操作一次性将put送 往服务器。默认情况下，客户端缓冲区是禁止的。可以通过自动刷写设置为FALSE来激活缓冲区。 table.setAutoFlush(false);void flushCommits () throws IOException这个方法是强制 将数据写到服务器。用户还可以根据下面的方法来配置客户端写缓冲区的大小。 void setWritaeBufferSize(long writeBufferSize) throws IOException;默认大小是 2MB，这个也是适中的，一般用户插入的数据不大，不过如果你插入的数据大的话，可能要考虑增大这个值。从而允许客户端更高效地一定数量的数据组成一组通 过一次RPC请求来执行。给每个用户的HTable设置一个写缓冲区也是一件麻烦的事，为了避免麻烦，用户可以在

Hbase-site.xml中给用户设置一个较大的预设值。

<property>

<name>hbase.client.write.buffer</name>

<value>20971520</value>

</property>

补充7：

hbase支持大量的算法，并且支持列族级别以上的压缩算法，除非有特殊原因，不然我们应该尽量使用压缩，压缩通常会带来较好的 性能。通过一些测试，我们推荐使用SNAPPY这种算法来进行我们hbase的压缩。

**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。

补充1：

在 HBase中，所有的存储文件都被划分成了若干个小存储块，这些小存储块在get或scan操作时会加载到内存中，他们类似于RDBMS中的存储单元页。 这个参数的默认大小是64K。通过以上方式设置：void setBlocksize(int s);（HBase中Hfile的默认大小就是64K跟 HDFS的块是64M没关系）HBase顺序地读取一个数据块到内存缓存中，其读取相邻的数据时就可以再内存中读取而不需要从磁盘中再次读取，有效地减少 了磁盘I/O的次数。这个参数默认为TRUE，这意味着每次读取的块都会缓存到内存中。但是，如果用户顺序读取某个特定的列族，最好将这个属性设置为 FALSE，从而禁止使用缓存快。上面这样描述的原因：如果我们访问特定的列族，但是我们还是启用了这个功能，这个时候我们的机制会把我们其它不需要的列 族的数据也加载到了内存中，增加了我们的负担，我们使用的条件是，我们获取相邻数据。 void setBlockCacheEnabled(boolean blockCacheEnable);

补充2：

1：禁止自动刷写。

我们有大批数据要插入时，如果我们没有禁止，Put实例会被逐个的传送到regio服务器

，如果用户禁止了自动刷写的功能，put操作会在写缓冲区被填满时才会被送出。

2：使用扫描缓存。

如果HBase被用作一个mapreduce作业的输入源，请最好将作为mapreduce作业输入扫描

器实例的缓存用setCaching()方法设置为比默认值1更大的数。使用默认值意味着map

任务会在处理每条记录时都请求region服务器。不过，这个值要是500的话，则一次

可传送500条数据到客户端进行处理，当然了这数据也是根据你的情况定的。

这个是行级的，在我们的119页有说明。

3：限定扫描范围。

这个是很好理解的，比如我们要处理大量行（特别是作为mapreduce的输入源），其中

用到scan的时候我们有Scan.addFamily();的方法，这个时候我们如果只是需要到

这个列族中的几个列，那么我们一定要精确。因为过多的列会导致效率的损失。

4：关闭resultScanner

当然了这个不能提高我们的效率，但是如果没关就会对效率有影响。

5：块缓存的用法

首先我们的块缓存是通过Scan.setCacheBolcks();的启动的，那些被频繁访问的行

我们应该使用缓存块，但是mapreduce作业使用扫描大量的行，我们就不该使用这个

了。（这个块缓存跟我在第四节中提到的那个块是不一样的）。

6：优化获取行健的方式

当然用这个的前提是，我们只需要表中的行健时，才能用。那么怎么用在411页有说明。

7：关闭Put上的WAL

书上是这么说，但是我个人觉得这个功能还是不用的好，因为我们关闭了这个功能，

服务器就不会把put写入到WAL，而是直接写到memstore里，这样一旦服务器出现故障

我们的数据就丢失了。