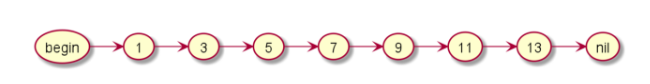
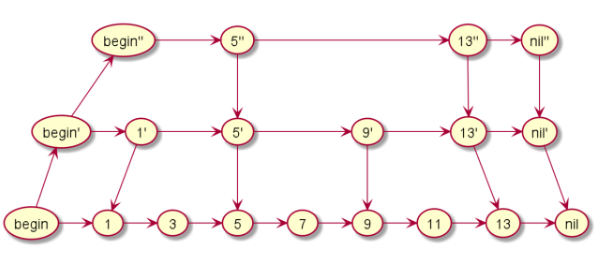
## HBase数据结构

在讲解HBase的LSM合并树之前，我们需要来了解一些常用的数据结构知识。

### 跳表



上图是一个有序链表，我们要检索一个数据就挨个遍历。如果想要再提升查询效率，可以变种为以下结构：



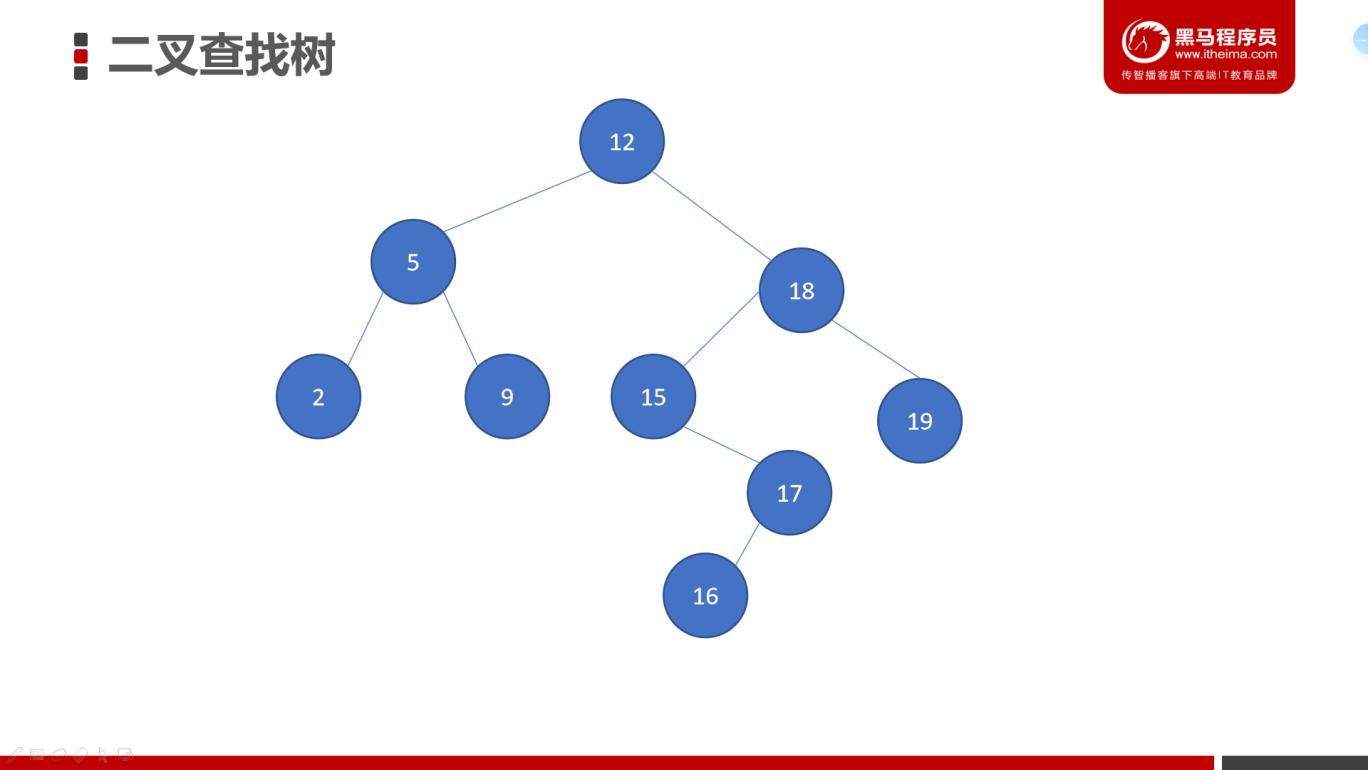
现在，我们要查询11，可以跳着来查询，从而加快查询速度。

### 常见树结构（扩展了解）

#### 二叉搜索树（Binary Search Tree）

##### 什么是二叉搜索树？

二叉搜索树也叫二叉查找树。它是一种比较特殊的二叉树。



##### 树的高度、深度、层数

* 深度

节点的深度是根节点到这个节点所经历的边的个数，深度是从上往下数的

* 高度

节点的高度是该节点到叶子节点的最长路径（边数），高度是从下往上数的

* 层数

根节点为第一层，往下依次递增

上图：

* 节点12的深度为0，高度为4，在第1层
* 节点15的深度为2，高度为2，在第3层

##### 二叉搜索树的特点

树中的每个节点，它的左子树中所有关键字值小于该节点关键字值，右子树中所有关键字值大于该节点关键字值

##### 二叉搜索树的查询方式

* 首先和根节点进行比较，如果等于根节点，则返回
* 如果小于根节点，则在根节点的左子树进行查找
* 如果大于根节点，则在根节点的右子树进行查找

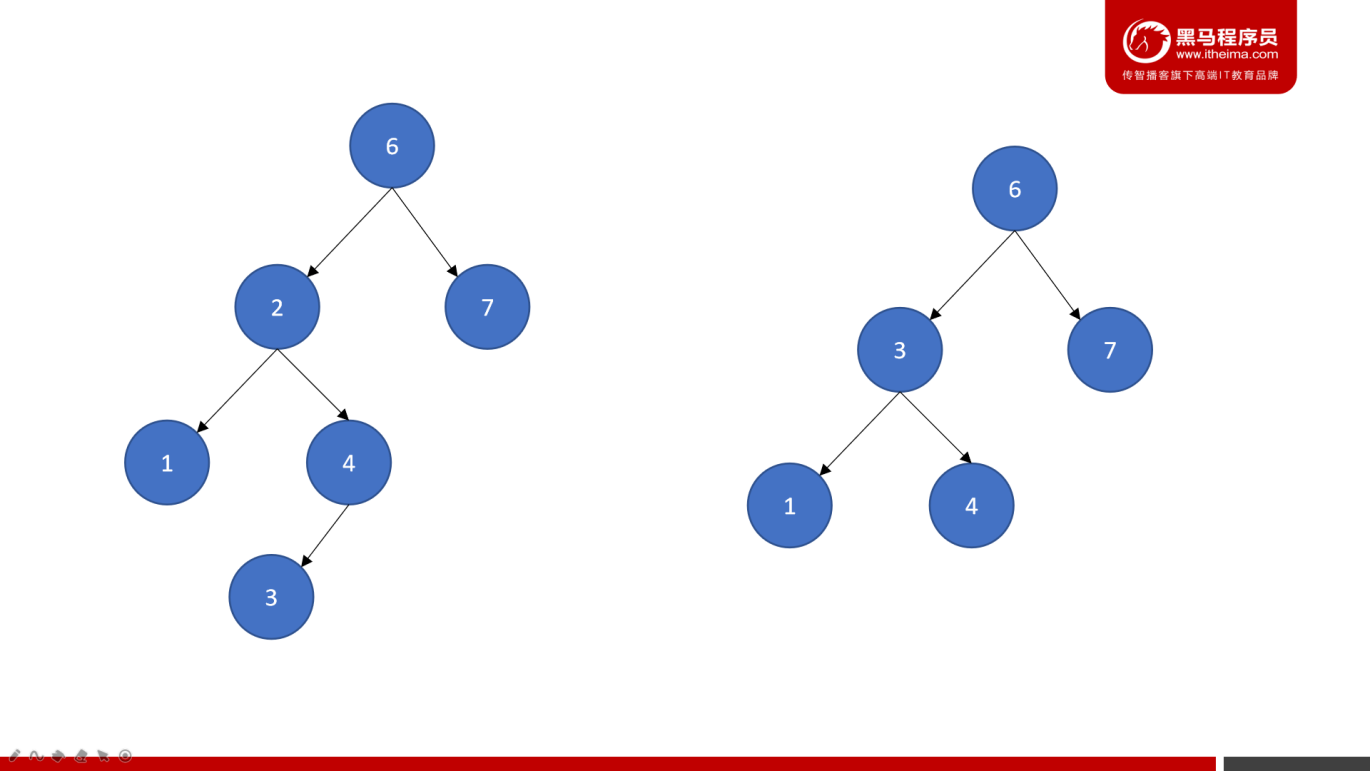
##### 二叉搜索树的缺点

因为二叉搜索树是一种二叉树，每个节点只能有两个子节点，但有较多节点时，整棵树的高度会比较大，树的高度越大，搜索的性能开销也就越大

#### 平衡二叉树（Balance Binary Tree）

##### 简介

* 平衡二叉树也称为AVL树
* 它是一颗空数，或者它的任意节点左右两个子树的高度差绝对值**不超过1**
* 平衡二叉树很好地解决了二叉查找树退化成链表的问题



上图：

1. 两棵树都是二叉查找树
2. 左边的不是平衡二叉树

节点6的子节点：节点2的高度为：2，节点7的高度为：0，| 2 – 0 | = 2 > 1）

1. 右边的是平衡二叉树

节点6的子节点：节点3的高度为：1，节点7的高度为：0，| 1 – 0 | = 1 = 1 ）

##### 平衡二叉树的特点

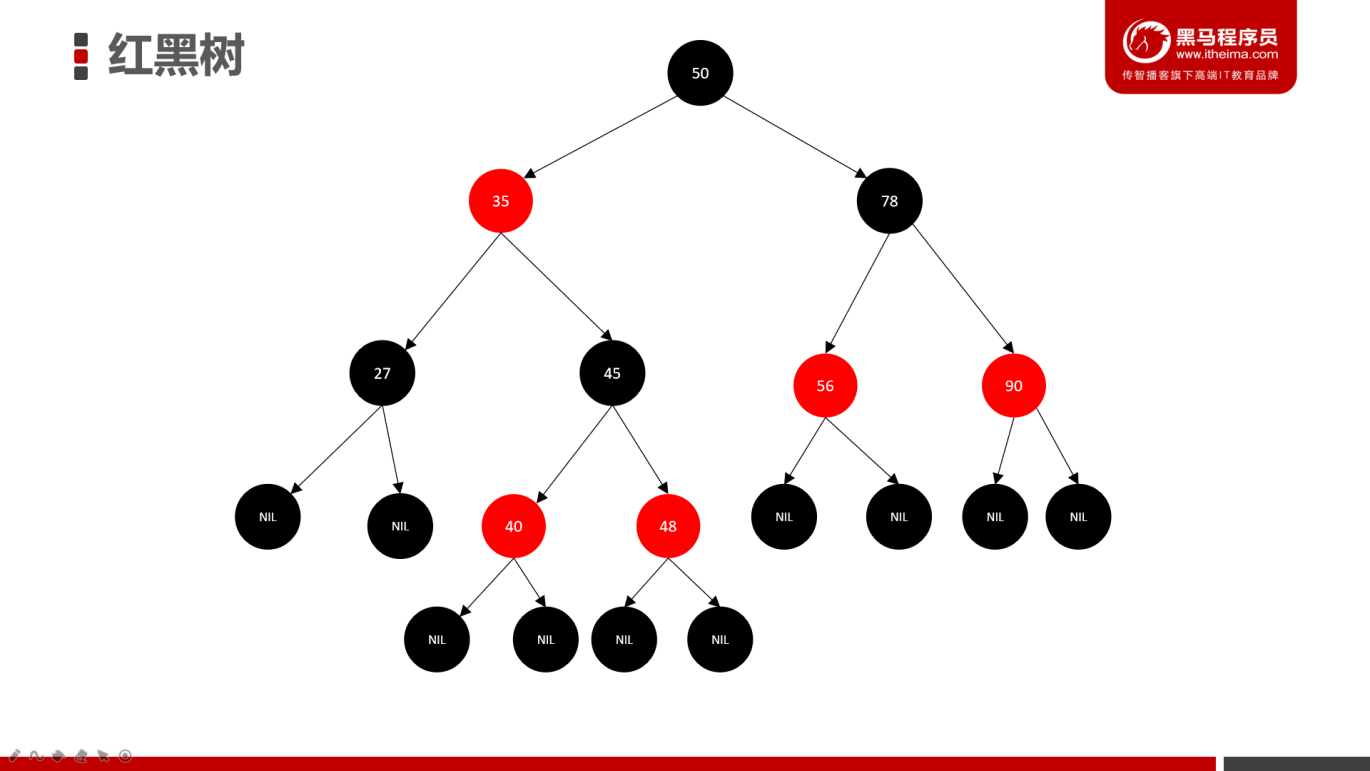
AVL树是高度平衡的（严格平衡），频繁的插入和删除，会引起频繁的rebalance，导致效率下降，它比较使用与插入/删除较少，查找较多的场景

#### 红黑树

##### 简介

红黑树是一种含有红黑节点并能自平衡的二叉搜索树，它满足以下性质：

* 每个节点要么是黑色，要么是红色
* 根节点是黑色
* 每个叶子节点（NIL）是黑色
* 每个红色结点的两个子结点一定都是黑色
* 任意一结点到每个叶子结点的路径都包含数量相同的黑结点



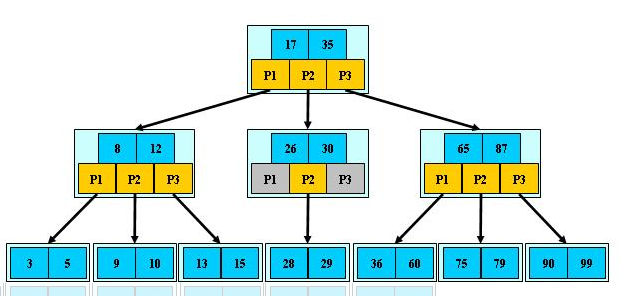
##### 红黑树的特点

和AVL树不一样，红黑树是一种弱平衡的二叉树，它的插入/删除效率更高，所以对于插入、删除较多的情况下，就用红黑树，而且查找效率也不低。例如：Java中的TreeMap就是基于红黑树实现的。

#### B树

##### 什么是B树

* B树是一种**平衡多路搜索树**
* 与二叉搜索树不同的是，B树的节点可以有多个子节点，不限于最多两个节点
* 它的子节点可以是几个或者是几千个



##### B树的特点

* 所有节点关键字是按递增次序排列，并遵循左小右大原则
* B-树有个最大的特点是有多个查找路径，而不像二叉搜索树，只有两路查找路径。
* 所有的叶子节点在同一层
* **逐层查找，找到节点后返回**

##### B-树的查找方式

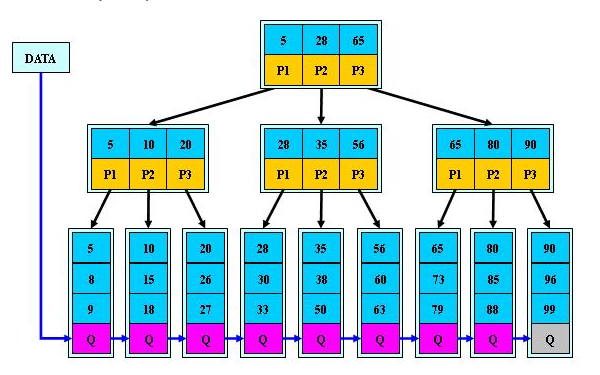
* 从根节点的关键字开始比较，例如：上图为13，判断大于还是小于
* 继续往下查找，因为节点可能会有多个节点，所以需要判断属于哪个区间
* 不断往下查找，直到找到为止或者没有找到返回Null

#### B+树结构

##### B+树简介

B+树是B树的升级版。B+树常用在**文件系统和数据库**中，B+树通过**对每个节点存储**数据的个数进行扩展，**可以让连续的数据进行快速访问，有效减少查询时间**，减少IO操作。它能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度

例如：Linux的Ext3文件系统、Oracle、MySQL、SQLServer都会使用到B+树。



* B+ 树是一种树数据结构，是一个n叉树
* 每个节点通常有多个孩子
* 一颗B+树包含根节点、内部节点和叶子节点
* **只有叶子节点包含数据（所有数据都是在叶子节点中出现）**

##### B+树的特点

* 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的

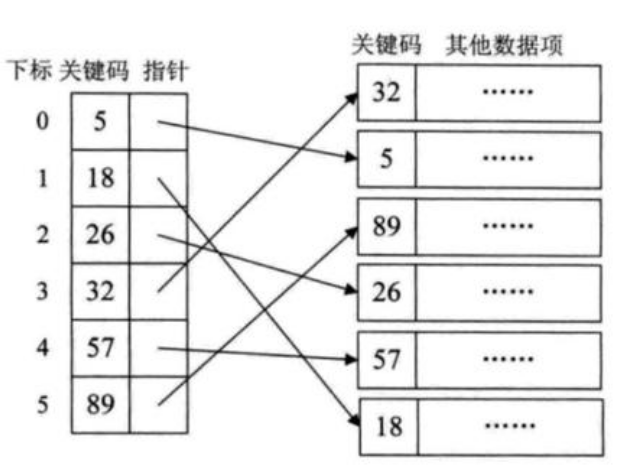
如果执行的是：select \* from user order by id，要全表扫描数据，那么B树就比较费劲了，但B+树就容易了，只要遍历最后的链表就可以了。

* 只会在叶子节点上搜索到数据
* 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储
* 数据库的B+树高度大概在 2-4 层，也就是说查询到某个数据最多需要2到4次IO，相当于0.02到0.04s

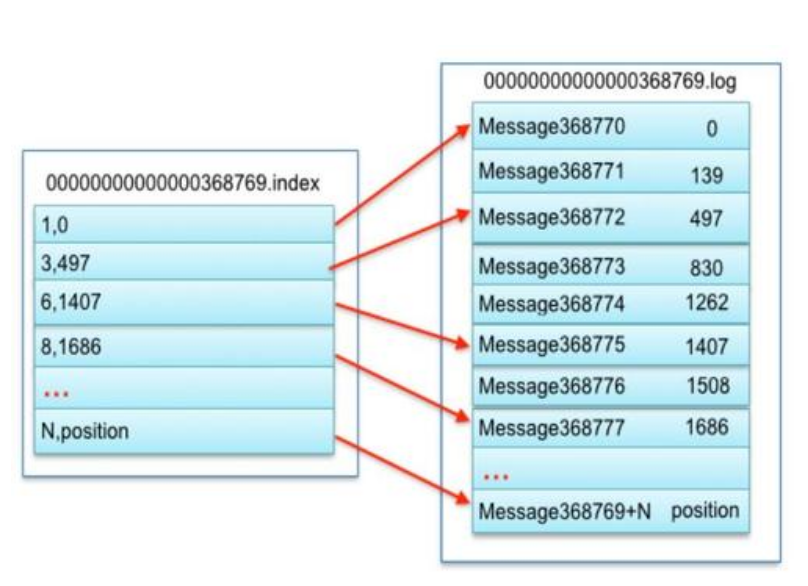
##### 稠密索引和稀疏索引

* 稠密索引文件中的每个搜索码值都对应一个索引值
* 稀疏索引文件只为索引码的某些值建立索引项

稠密索引：



稀疏索引：



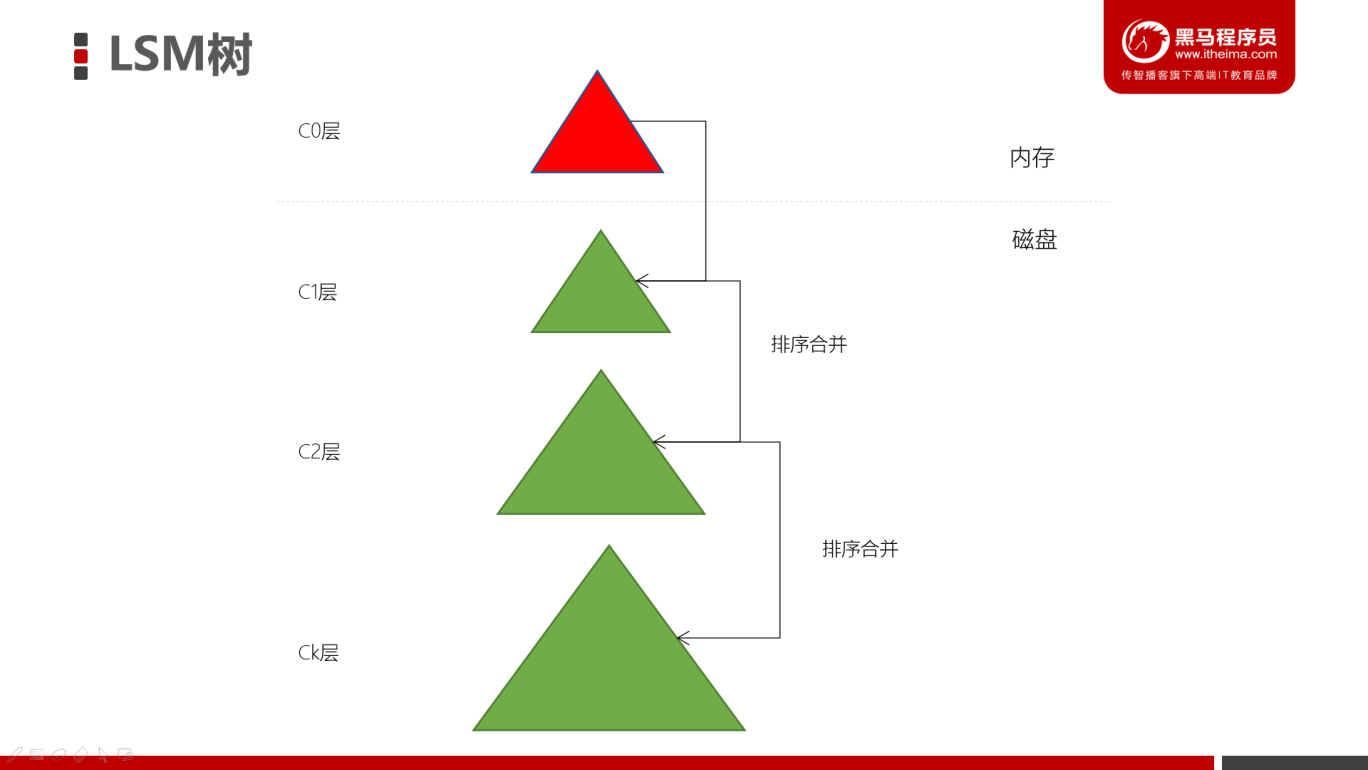
### LSM树数据结构

#### 简介

传统关系型数据库，一般都选择使用B+树作为索引结构，而在大数据场景下，HBase、Kudu这些存储引擎选择的是LSM树。LSM树，即日志结构合并树(Log-Structured Merge-Tree)。

* LSM树主要目标是快速建立索引
* B+树是建立索引的通用技术，但如果并发写入压力较大时，B+树需要大量的磁盘**随机IO**，而严重影响索引创建的速度，在一些写入操作非常频繁的应用场景中，就不太适合了
* LSM树通过磁盘的**顺序写**，来实现最好的写性能

#### LSM树设计思想



* LSM 的主要思想是划分不同等级的结构，换句话来理解，就是LSM中不止一个数据结构，而是存在多种结构
* 一个结构在内存、其他结构在磁盘（HBase存储结构中，有内存——MemStore、也有磁盘——StoreFile）
* 内存的结构可以是B树、红黑树、跳表等结构（HBase中是跳表），磁盘中的树就是一颗B+树
* C0层保存了最近写入的数据，数据都是有序的，而且可以随机更新、随机查询
* C1到CK层的数据都是存在磁盘中，每一层中key都是有序存储的

#### LSM的数据写入操作

* 首先将数据写入到WAL（Write Ahead log），写日志是顺序写，效率相对较高（PUT、DELETE都是顺序写）
* 数据项写入到内存中的C0结构中
* 只有内存中的C0结构超过一定阈值的时候，将内存中的C0、和C1进行合并。这个过程就是Compaction（合并）
* 合并后的新的C1顺序写磁盘，替换之前的C1
* 但C1层达到一定的大小，会继续和下层合并，合并后旧的文件都可以删除，只保留最新的
* 整个写入的过程只用到了内存结构，Compaction由后台异步完成，不阻塞写入

#### LSM的数据查询操作

* 先在内存中查C0层
* 如果C0层中不存在数据，则查询C1层
* 不断逐层查询，最早的数据在CK层
* C0层因为是在内存中的结构中查询，所以效率较高。因为数据都是分布在不同的层结构中，所以一次查询，可能需要多次跨层次结构查询，所以读取的速度会慢一些。
* 根据以上，LSM树结构的程序适合于写密集、少量查询的场景

### 布隆过滤器

#### 简介

客户端：这个key存在吗？

服务器：不存在/不知道

本质上，布隆过滤器是一种数据结构，是一种比较巧妙的概率型数据结构。它的特点是高效地插入和查询。但我们要检查一个key是否在某个结构中存在时，通过使用布隆过滤器，我们可以快速了解到「这个key一定不存在或者可能存在」。相比于以前学习过的List、Set、Map这些数据结构，它更加高效、占用的空间也越少，但是它返回的结果是概率性的，是不确切的。

#### 应用场景

##### 缓存穿透

* 为了提高访问效率，我们会将一些数据放在Redis缓存中。当进行数据查询时，可以先从缓存中获取数据，无需读取数据库。这样可以有效地提升性能。
* 在数据查询时，首先要判断缓存中是否有数据，如果有数据，就直接从缓存中获取数据。
* 但如果没有数据，就需要从数据库中获取数据，然后放入缓存。如果大量访问都无法命中缓存，会造成数据库要扛较大压力，从而导致数据库崩溃。而使用布隆过滤器，当访问不存在的缓存时，可以迅速返回避免缓存或者DB crash。

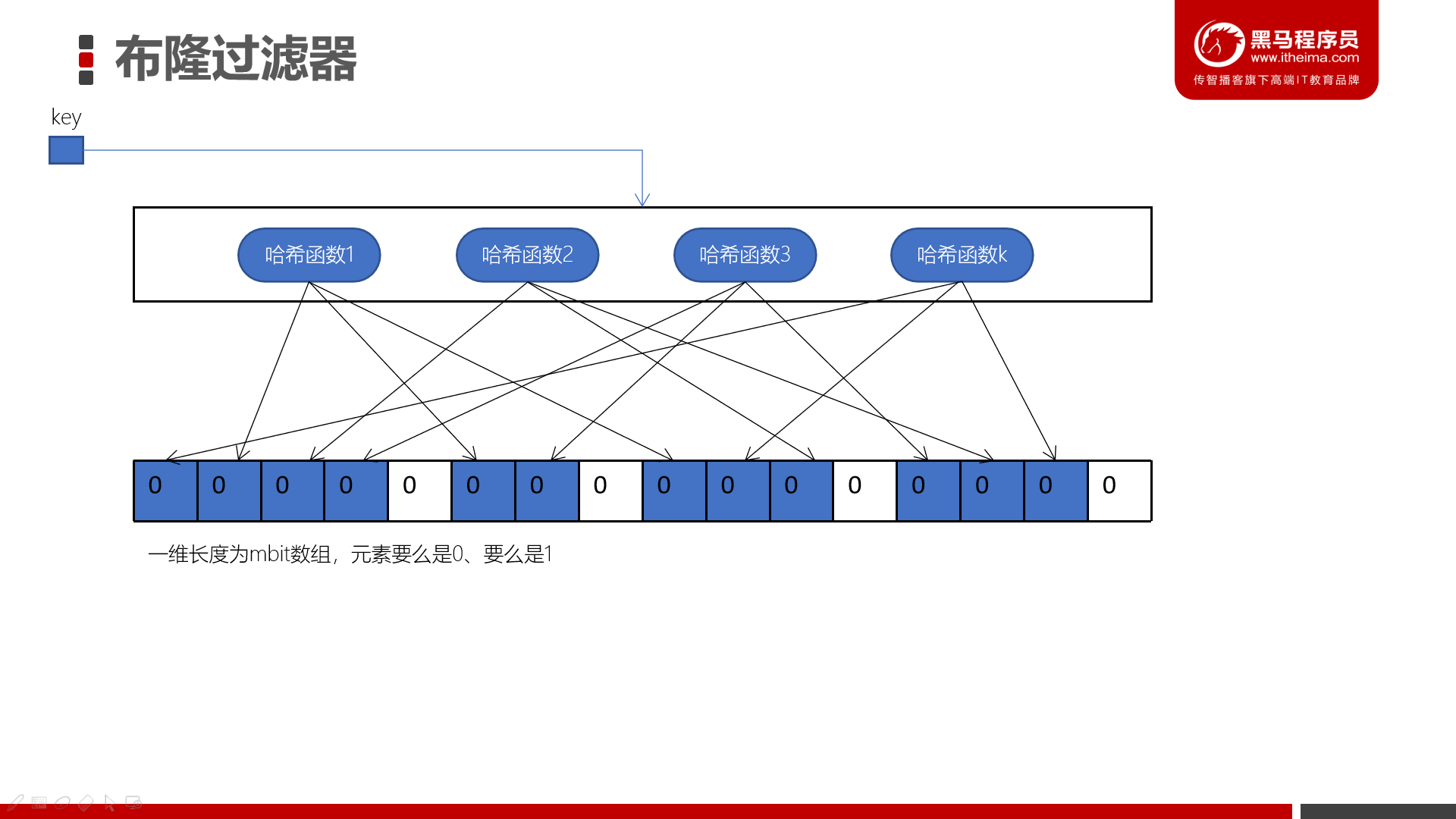
##### 判断某个数据是否在海量数据中存在

HBase中存储着非常海量数据，要判断某个ROWKEYS、或者某个列是否存在，使用布隆过滤器，可以快速获取某个数据是否存在。但有一定的误判率。但如果某个key不存在，一定是准确的。

#### HashMap的问题

* 要判断某个元素是否存在其实用HashMap效率是非常高的。HashMap通过把值映射为HashMap的Key，这种方式可以实现O(1)常数级时间复杂度。
* 但是，如果存储的数据量非常大的时候（例如：上亿的数据），HashMap将会耗费非常大的内存大小。而且也根本无法一次性将海量的数据读进内存。

#### 理解布隆过滤器



* 布隆过滤器是一个bit数组或者称为一个bit二进制向量
* 这个数组中的元素存的要么是0、要么是1
* k个hash函数都是彼此独立的，并将每个hash函数计算后的结果对数组的长度m取模，并将对一个的bit设置为1（蓝色单元格）
* 我们将每个key都按照这种方式设置单元格，就是「**布隆过滤器**」

#### 根据布隆过滤器查询元素

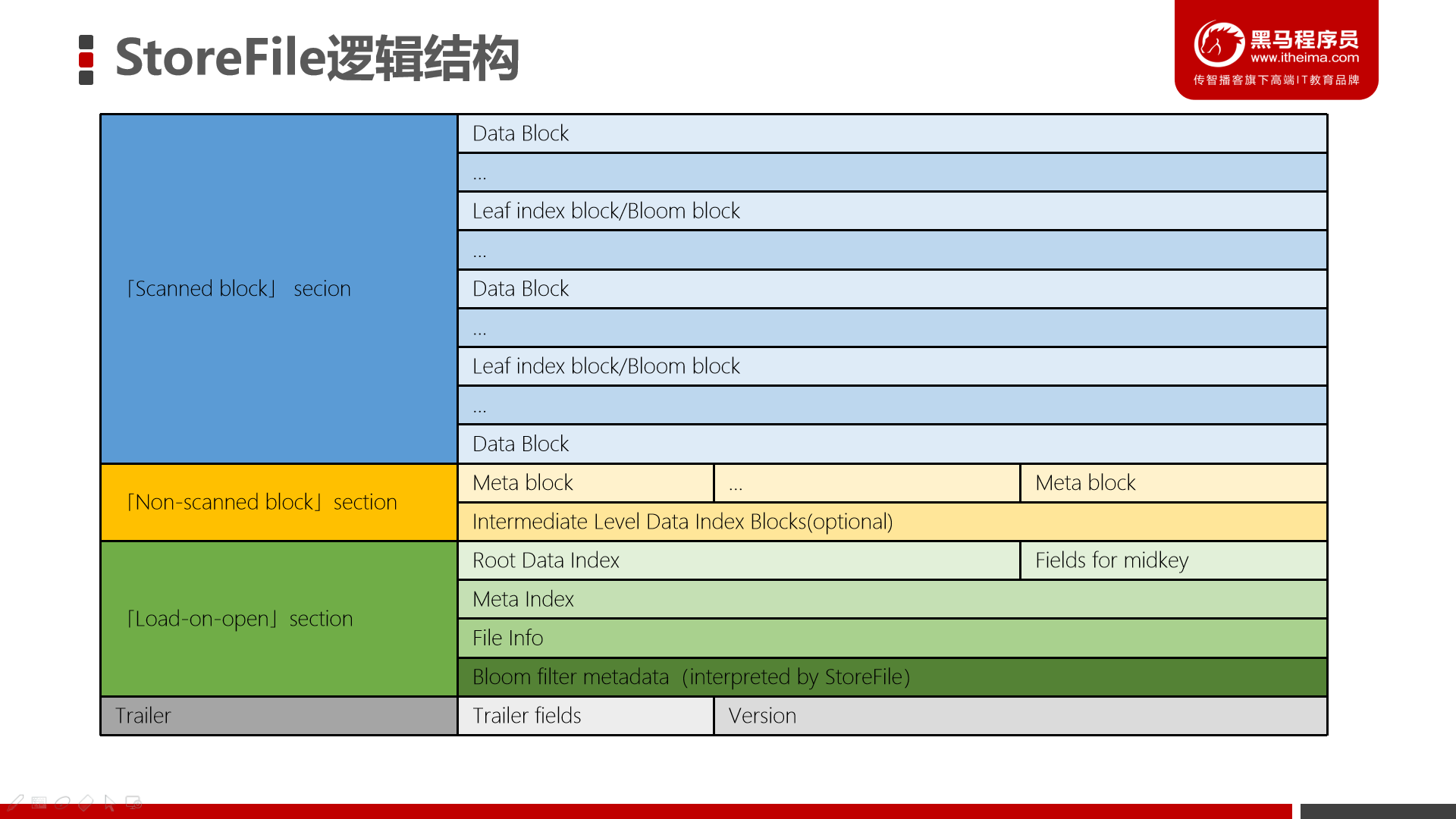
* 假设输入一个key，我们使用之前的k个hash函数求哈希，得到k个值
* 判断这k个值是否都为蓝色，如果有一个不是蓝色，那么这个key一定不存在
* 如果都有蓝色，那么key是可能存在（布隆过滤器会存在误判）
* 因为如果输入对象很多，而集合比较小的情况，会导致集合中大多位置都会被描蓝，那么检查某个key时候为蓝色时，刚好某个位置正好被设置为蓝色了，此时，会错误认为该key在集合中

### StoreFiles（HFile）结构

StoreFile是HBase存储数据的文件格式。

#### HFile的逻辑结构

##### HFile逻辑结构图



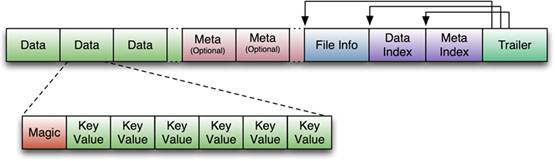
##### 逻辑结构说明

4大部分

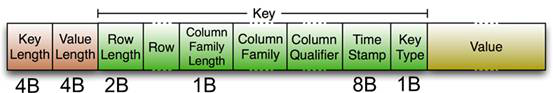
* Scanned block section
  + 扫描StoreFile时，所有的Data Block（数据块）都将会被读取
  + Leaf Index（LSM + C1树索引）、Bloom block（布隆过滤器）都会被读取
* Non-scanned block section
  + 扫描StoreFile时，不会被读取
  + 包含MetaBlock和Intermediate Level Data Index Blocks
* Opening-time data section
  + 在RegionServer启动时，需要将数据加载到内存中，包括数据块索引、元数据索引、布隆过滤器、文件信息。
* Trailer
  + 记录了HFile的基本信息
  + 各个部分的偏移值和寻址信息

#### StoreFile物理结构

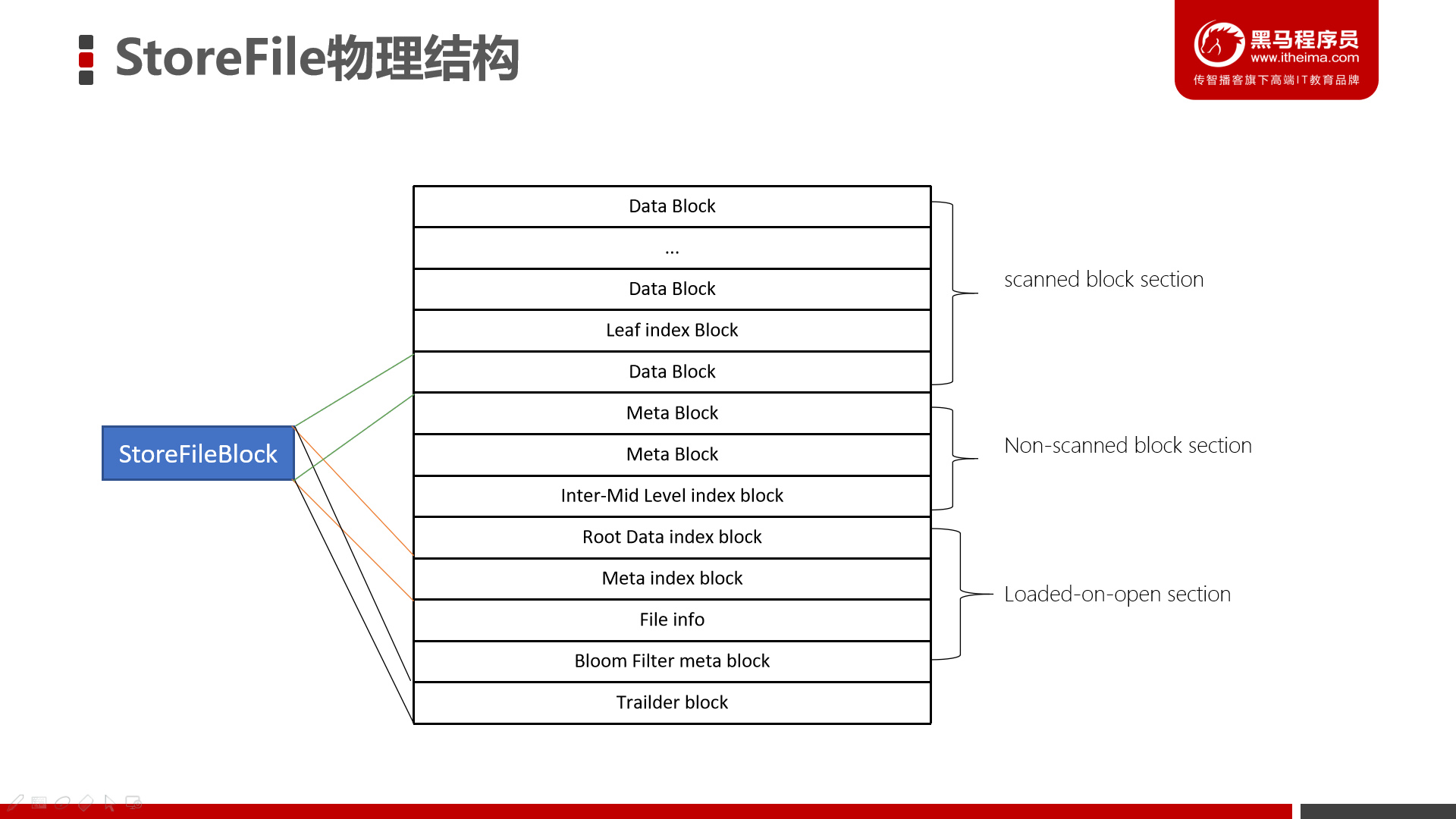
StoreFile是以Hfile的形式存储在HDFS上的。Hfile的格式为下图：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0080.jpg)

* HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。正如图中所示的，Trailer中有指针指向其他数 据块的起始点。
* File Info中记录了文件的一些Meta信息，例如：AVG\_KEY\_LEN, AVG\_VALUE\_LEN, LAST\_KEY, COMPARATOR, MAX\_SEQ\_ID\_KEY等
* Data Index和Meta Index块记录了每个Data块和Meta块的起始点。
* Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询。 每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成, Magic内容就是一些随机数字，目的是防止数据损坏。
* HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：

[](http://www.searchtb.com/wp-content/uploads/2011/01/image0090.jpg)

1. 开始是两个固定长度的数值，分别表示Key的长度和Value的长度
2. 紧接着是Key，开始是固定长度的数值，表示RowKey的长度
3. 紧接着是 RowKey，然后是固定长度的数值，表示Family的长度
4. 然后是Family，接着是Qualifier
5. 然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）——每一种操作都会生成一个Key-Value。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。



* Data Block段

保存表中的数据，这部分可以被压缩

* Meta Block段 (可选的)

保存用户自定义的kv对，可以被压缩。

* File Info段

Hfile的元信息，不被压缩，用户也可以在这一部分添加自己的元信息。

* Data Block Index段

Data Block的索引。每条索引的key是被索引的block的第一条记录的key。

* Meta Block Index段 (可选的)

Meta Block的索引。

* Trailer

这一段是定长的。保存了每一段的偏移量，读取一个HFile时，会首先 读取Trailer，Trailer保存了每个段的起始位置(段的Magic Number用来做安全check)，然后，DataBlock Index会被读取到内存中，这样，当检索某个key时，不需要扫描整个HFile，而只需从内存中找到key所在的block，通过一次磁盘io将整个 block读取到内存中，再找到需要的key。DataBlock Index采用LRU机制淘汰