1. 基本的数据结构和算法
   1. 顺序数组 二分查找
      1. 存储成顺序数组的方式使用二分查找对数据进行检索，检索复杂度为o(logn)。
      2. 顺序表的插入和删除时间复杂度为o(n)。
   2. Hash表
      1. 定义：根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。
      2. 为什么需要hash表？传统的数组和链表当作存储接口的时候并不能同时保证寻址（链表只能顺序查找）和存取（数组只能顺序存储）的效率，使用hash表可以同时均衡存取和寻址。
      3. Hash表要达到的效果：
         * 单向推导，不可以反推导出原始数据（数据的安全性）
         * 对于数据的微小变化，hash值可以充分变化
         * 从原始数据计算hash值的时候效率高
         * Hash映射函数的冲突概率要小
      4. hash映射函数：
         * 加法Hash：将值中的每一位进行相加然后取余一个很大的素数
         * 位运算Hash： Hash函数通过利用各种位运算（常见的是移位和异或）来充分的混合输入元素。
         * 乘法Hash：利用乘法的不相关性（FNV算法）
         * 除法Hash（效率太低没人用）
         * 查表hash：根据表格映射查找出hash值
         * 混合hash：结合上述几种方式的hash（MD5；Tiger）
      5. 如何解决冲突：
         * 开放定址法（散列法）

顺序再散列：如果发生冲突则找该位置的下一个位置

二次探测再散列，如果碰到冲突，就在左右进行跳跃式探测

* + - * 再哈希法：同时构造多个hash函数，hash冲突的时候，在执行另外的hash函数进行再hash（不易冲突但是计算量大）
      * 链地址法：这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。
    1. 使用场景
       - 版权校验：用于保证数据的唯一性，所有数据以第一次提交的为主（避免用户重复提交）
       - 大文件分块校验：将文件分块上传的时候怎么验证数据的完整性，将原始数据进行分块然后进行hash处理后生成hashlist并将数据和hashlist上传，到了服务器后将文件再次进行hash，将结果拼接成rootHash，与之前的hashlist拼成的roothash对比，如果一致则表明数据一致，如果不一致则对比每一块数据的hash判断哪块数据损坏了。
       - 负载均衡（一致性Hash）

参考：<https://iwiki.woa.com/pages/viewpage.action?pageId=74061517&from=km_search>

* + 1. Hash应用：地图距离以及地图范围筛选操作的GeoHash（比如现在的外卖服务，使用geohash在数据库中进行城市范围或者片区范围内的小哥筛选like geohash字符串中的前n位数据），大数据过滤的布隆过滤器
  1. 二叉查找树（BST树）
     + - 定义：若左子树不空，则左子树上所有结点的值均小于它的根结点的值；若右子树不空，则右子树上所有结点的值均大于它的根结点的值；左、右子树也分别为二叉排序树；没有键值相等的结点。
       - 最优情况：类似于一颗AVL树，查找效率为logn
       - 最差情况：一个递增或者递减的序列，查询效率为n
  2. 红黑树

红黑树(RB Tree)是平衡二叉树，其优点就是树到叶子节点深度一致，查找的效率也就一样，为logN.在实行查找，插入，删除的效率都一致，而当是全部静态数据时，没有太多优势，可能采用hash表各合适。hash\_map是一个hash table占用内存更多，查找效率高一些，但是hash的时间比较费时。

* 1. AVL树

AVL树本质上是一颗二叉查找树，但是它又具有以下特点：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。在AVL树中任何节点的两个子树的高度最大差别为一，所以它也被称为平衡二叉树。查找速度永远保持在logn。BST树->红黑树->AVL树查找效率越来越高，因为对树的左右节点高度差控制越来越严格，这也需要树在增加节点和删除节点的时候进行树的旋转调整。

* 1. B 树

根节点至少有两个子节点；每个节点有M-1个key，并且以升序排列；位于M-1和M key的子节点的值位于M-1 和M key对应的Value之间；其它节点至少有M/2个子节点。B树会将数据存储到它的所有节点中。插入删除细节：https://www.cnblogs.com/nullzx/p/8729425.html

* 1. B+树

在B树的基础上，叶子结点增加了顺序指针，所有的数据都存储到了叶子结点上。

* 1. 总结几种数据结构

为什么mysql索引采用的是B+树和hash，而不是上述的其他类型？

BST树，当数据库存储大量数据的时候，存在极端情况整个BST树极度不平衡，导致查找时间达到o(n)。

AVL树：该树在查找插入和删除的消耗都是O(lgn),再插入数据的时候最多旋转1次，但是该树在删除的时候会导致树的失衡而进行旋转，量级为O(lgn)

红黑树：会造成树的高度过高，在内存的时候存取性能可以，但是在磁盘存取的时候会导致磁盘IO问题。为什么树太高的时候会影响磁盘IO？因为每次只能读取一个节点导致会进行h（树的高度）次单个节点的IO查找。

B树：磁盘IO有个有个特点，就是从磁盘读取1B数据和1KB数据所消耗的时间是基本一样的，所以尽可能在一次磁盘IO中多读一点数据到内存。这一点很符合B树的一个节点多个key的构造，从结构上看，B树比较矮，进行的磁盘io数量会比较少。B树有以下几个优点：1.优秀检索速度，时间复杂度：B树的查找性能等于O（h\*logn），其中h为树高，n为每个节点关键词的个数；2.尽可能少的磁盘IO，加快了检索速度；3.可以支持范围查找。

既然B树已经这么优秀了为什么mysql还要选择B+树呢？

* + - * B树一个节点里存的是数据，而B+树存储的是索引（地址），所以B树里一个节点存不了很多个数据，但是B+树一个节点能存很多索引，B+树叶子节点存所有的数据。
      * B+树的叶子节点是数据阶段用了一个链表串联起来，便于范围查找。

Hash与B+树为何选择B+树不选择查找更快的Hash？

1. 扩展性差，需要提前预测数据量的大小

哈希表是基于数组的，数组创建后难于扩展某些哈希表被基本填满时，性能下降得非常严重，所以程序员必须要清楚表中将要存储多少数据（或者准备好定期地把数据转移到更大的哈希表中，这是个费时的过程）。因为动态Hash一直是一个比较难的问题，所以开始为了保证较合适的填充因子，所以不得不开一个比较大的空间来存储索引，此时数据内容的条数可能并不是很多。而B树，扩展性比较好。

1. 不能有序遍历数据

没有一种简便的方法可以以任何一种顺序〔例如从小到大）遍历表中数据项。B树阔以轻松搞定（中序遍历即可：O（N \* log(ceil(m/2) N)）），B+树阔以更轻松地搞定（扫一遍叶子结点即可：O（N））。

1. B树磁盘IO次数少

而hash如果同一个桶里的数据如果比较多，难免增加不少IO次数。而磁盘IO次数往往决定了索引速度。

AHI自适应hash索引详见后面的innodb存储引擎详解

1. Mysql两种存储引擎的索引结构

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。下图是MyISAM索引的原理图：

手机屏幕截图

描述已自动生成

InnoDB引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据。下图是InnoDB索引的原理图：

手机屏幕的截图

描述已自动生成

1. 多列索引存储结构

先看单个的辅助索引；对于MyISam来说单个的辅助索引和主键索引没有区别只是辅助索引的key不是唯一的，具体结构如下图：

一些文字和图片的手机截图

描述已自动生成

对于Innodb来说辅助索引和主键索引的区别是，辅助索引的data域存储的是主键的值而不是地址，具体结构如下图：

手机屏幕的截图

描述已自动生成

对于多列索引来说：数据会先按照最左边到最右边的列一层一层的进行排序，先按照第一列排序结果，如果第一列一样就按照第二个列排序，以此类推排序出B+树，叶子结点存储的值和单列索引一致，具体结构如下图：

图片包含 游戏机, 文字

描述已自动生成

1. 索引失效情况（为啥失效）

首先明确一个原则就是最左前缀原则，为什么是最左原则呢？可以看到上面的多列索引的索引树结构，如果说你不找最左边前缀的姓名的话，直接通过出生年月进行搜索的话是不能保证数据是顺序排列的，所以没办法利用索引进行扫描。

其次明确哪些地方会用到索引：

1. where条件
2. 排序
3. 分组
4. 聚合minmax…
5. 连接join

还有一个重要的考量就是索引的区分度：如果count(distinct(column))/count(column)<80以下一般不建立索引。

·where查询

Not in ：一般不走索引，为什么呢？反过来想一定要走索引是怎么走呢？假设有个column是A where A not in（1， 2）；A的类别有100000种分布很均匀，相当于A！=1and A！=2要去扫描整个b+树找A > 1 and A < 2 or A < 1 or A > 2;

这就变得很慢。但是如果数据只有0 1 2 3 分布过于局部，用索引的范围扫描就会变得很快这都是相对的。

No exist：No Exist这个用法跟内外表大小有关系，如果内表很大外表很小，noexist就会变得很优秀，反之性能就变得很差；为什么呢？因为no exist的外表是走循环的，内表是走索引的。

In：in语句优化器会把它看成区间查询如A in（a，b）优化器会把它看成A in [a,a],B in [b,b]两个去查然后unionall，这个时候如果in里面不是常量了而是一个语句的话，mysql会先执行内部的子查询，如果内部的子查询返回结果很大的时候in语句的总体查询性能就会急剧降低，反之如果很小就会像常量查询那么优秀。

Exist： 同no exist。

Or：会触发mysql的优化器将or语句转为in语句

And：需要注意最左原则，多列索引的时候需要从左往右加索引

Like：like语句其实就是模糊匹配，分为3种情况like’%query’ ‘query%’ ‘%query%’;先分析第一种情况，结合上文的b+树模型的Alien名字，找姓名后缀lien的人即sql为：like %lien，这样无法使用顺序的索引树去查找出数据。第二种情况，我们找Ali开头的姓名，我们可以从第一层找到Alien 和Astaire，发现Astaire不符合所以索引会走到第二层Akroyd也不符合直接返回Alien通过索引扫瞄很快。第三种情况就是集合1/2种情况会和第一种一样无法使用到索引。

Between：采用索引范围查找符合最左原则。

* 其他失效情况：

1.类型转换

先介绍一下Mysql隐式转换规则：

* 如果一个或两个参数都是NULL，比较的结果是NULL，除了NULL安全的<=>相等比较运算符。对于NULL <=> NULL，结果为true。不需要转换
* 如果比较操作中的两个参数都是字符串，则将它们作为字符串进行比较。
* 如果两个参数都是整数，则将它们作为整数进行比较。
* 如果不与数字进行比较，则将十六进制值视为二进制字符串
* 如果其中一个参数是十进制值，则比较取决于另一个参数。 如果另一个参数是十进制或整数值，则将参数与十进制值进行比较，如果另一个参数是浮点值，则将参数与浮点值进行比较
* 如果其中一个参数是TIMESTAMP或DATETIME列，另一个参数是常量，则在执行比较之前将常量转换为时间戳。
* 在所有其他情况下，参数都是作为浮点数（实数）比较的。

类型转换和索引失效的关系：

* in 参数包含多个类型（索引失效）
* 左边是数字右边是字符串（索引不失效，都会转为实数比较）整数转实数是唯一的；右边是数字左边是字符串（索引失效）；因为字符串转为浮点数是不确定的列如‘10001’，‘10001a’,'10001b'都将转化为浮点数10001。
* Join的时候where左右两边字段字符集类型不同。

索引列出现函数或者计算用不到索引，因为出现了计算和函数mysql必须进行全表扫描并且进行计算才能确定是否相等。

OrderBy GroupBy 违反最左原则用不到索引原理同where

1. 倒排索引

正向索引：

图片包含 游戏机, 截图

描述已自动生成

倒排索引

手机屏幕截图

描述已自动生成

那么倒排的瓶颈就是word的查找，在elasticseach中是利用B树或者B+树进行存储，也可以hash+链表的存储方式，也可以java1.8以后的hashmap的实现方式。

1. 如何优化索引
2. 考虑覆盖索引 ---在高并发查询简单列表字段的时候可以采用
3. SQLAdvisor
4. Explain语句优化