索引

啥叫索引?

查字典的时候, 你是一页页翻去找字, 还是先翻目录, 通过目录定位到你想查的那个字呢? 你肯定通过目录查呀, 因为目录快呀. 这里的目录其实就是一种索引, 通过空间换时间的思想加快查询速度.

从数据库层面来说, 索引就是一种能快速查找的数据结构, 能加快sql执行速度的. 而字典一页一页翻找就是全表扫描, 通过目录查数据, 就是索引查询. 所以执行sql走索引肯定快呀, 全表扫描肯定慢呀.

如果让你设计, 数据库应该使用哪种数据结构作为索引呢?

常见的能快速搜索的数据结构有哈希表, 二叉排序树, 平衡二叉树, 红黑树, B树, B+树.

hash表用唯一的key查对应的value, 时间复杂度O(1), 它很快, 但是没办法做范围查询, 也没法排序.

二叉排序树, 左节点小于根, 右节点大于根, 中序遍历就是有序的. 可以做排序, 也可以范围查询. 但是二叉排序树在顺序插入的情况下, 会退化为链表. 性能就会很差.

平衡二叉树, 插入节点时使用旋转操作让二叉树保持平衡, 左右子树高度差不大于1. 这样就能避免极端情况下退化为链表. 但是由于平衡二叉树追求完美的平衡, 会导致频繁的左旋右旋来保持平衡. 那么应用于数据库中, 当插入删除时, 频繁旋转就会带来大量磁盘IO, 降低性能.

那红黑树. 红黑树也是一种自平衡的二叉排序树, 通过插入删除数据时, 进行变色和旋转操作来让树保持平衡. 红黑树并不追求绝对平衡, 只要大致平衡就可以了, 这就大大降低了插入删除需要的旋转操作, 应用于数据库中, 就减少了磁盘IO. 但是红黑树是一种二叉树, 一个节点只能有两个孩子, 存储大量数据时, 树高会很高. 排序树的查找性能和树高直接相关, 这就导致存储大量数据时, 红黑树查找性能没那么好, 需要多次磁盘IO. 所以红黑树其实更适合于存储少量数据的内存操作, 像Java的HashMap, 当某个桶数据量数据量超过了阈值时, 链表就会变成红黑树. 当整个HashMap数据量比较多, 也能平均地分布到HashMap每个桶里, 每个桶都可以对应一个链表或者红黑树, 这样单个红黑树存的数据也不会很多, 而且纯内存操作, 也比较快.

红黑树是二叉树, 高度太高不行, 那B树呢? B树是多路平衡排序树, 一个节点可以有n个孩子, 数据量大的情况下, 树高也不会像红黑树那么高. 但是B树的节点又存数据又存索引值, 这就导致了B树一个节点存不了多少索引值, 树高就会变高. 而且B树做范围查询, 需要回溯树结构, 效率低, 还会产生很多随机IO.

B树也不行, 那上B+树, B+树也是多路平衡排序树, 而且B+树的非叶子节点上只存索引值, 不存数据, 那么B+树的单个节点能存的索引值就变多了, 树的高度更低, 查询性能更高. B+树的数据只存到最下面的叶子节点中, 而且由双向链表连接, 范围查询只需要遍历链表即可, 不用回溯树结构.

所以MySQL就选了B+树来保存索引.

那B+树具体是咋存储数据的呢

mysql索引从存储结构上划分主要分两种, 聚集索引和非聚集索引, 也有叫聚簇索引和非聚簇索引.

- 聚集索引就是索引值和数据一块存.
- 非聚集索引就是索引值和数据分开存.

举个例子, 像主键索引就是聚集索引. 非叶子节点都是索引值, 叶子节点保存全部索引值和对应的行数据. 因为主键在B+树中, 这也决定了主键应该自增插入, 能避免页分裂带来的性能问题.

像mysql中的唯一索引, 普通索引, 前缀索引就是二级索引, 或者叫非聚集索引. 这种索引非叶子节点存储索引值, 叶子节点存储索引值和对应的主键id. 所以查询时候, 先根据索引列查出对应的主键id, 然后再根据id查主键索引, 查出对应的行数据. 这也就是为啥叫二级索引. 因为它需要先索引树查出id, 再查主键索引树, 查出行数据. 这个过程叫做回表. 所以这里也可以看到回表会降低查询效率, 所以很多sql优化就是要避免回表.

那查二级索引一定会回表吗

不一定. 如果我们只查id, 那二级索引树的叶子节点就有id呀, 所以就可以只查二级索引树, 就没必要回表查询了, 这就提高了性能.

当一个索引包含所有需要查询的字段值, 就可以不回表, 这种情况叫覆盖索引. 所以减少回表查询就是要尽可能做到覆盖索引. 如果一个索引列有多个字段, 那就能大大增加覆盖索引的概率. 这种索引就叫联合索引, 所谓联合索引就是针对表中的多个字段去创建索引.

举个例子, 我对姓名和年龄两个建立联合索引, 此时索引列为姓名, 年龄, 索引树的叶子节点还挂了id, 如果我 `select id, 姓名, 年龄 where 姓名=xxx` 那么就会走联合索引, 直接查到id, 姓名, 和年龄, 就能避免回表, 提升性能.

所以减少回表查询的方式之一, 就是创建联合索引, 然后尽可能避免使用 `select *` , `select *` 查所有的列数据就很容易触发回表.

联合索引最左前缀匹配法则是怎么回事呢

联合索引因为有多个索引列, 所以排序的时候会根据索引列的顺序去排序, 查找的时候, 会根据索引列的顺序, 从左到右依次匹配.

比如建立联合索引(a, b, c), mysql底层建b+树时, 会根据a先排序, 再根据b排序, 再根据c排序. 所以查询匹配时, 得先从a开始查, 如果a不存在, b c就是乱序的, 无法跳过前面的索引列去匹配, 必须从左到右依次使用索引列匹配.

举个例子, 建立联合索引(a,b,c)

- 查询条件为where a=1 and b=2, 会走索引吗?
  - 显然是会的, 最左边的a和b都有.
- 那查询条件为where b=2 and a=1, 会走索引吗?
  - 也是会的, 只要有a 有b就行, 书写顺序无所谓, mysql优化器会自动把索引列重排序的.
- 那查询条件为where a=1 and c=2, 会走索引吗?
  - 会走部分索引, a会走索引, c的部分无法走索引, 因为按索引顺序c的左边时b, 查询没有b, c的部分就无法走索引.
- 那查询条件为 where b=1 and c=2, 会走索引吗?
  - 完全不会走, 因为a在最左边, b c的有序是建立在a上的, 没有a, 那么b c无法保证有序, 就无法走索引.

那还有其他索引失效的情况吗?

- 比如模糊查询场景中, `where name like '王%'` 尾部模糊, 索引生效, 因为索引是从左往右匹配的, `where name like '%三'` 头部模糊, 索引失效, 前面的数据没有了, 就没法搜索匹配了.
- 索引列运算/函数会使索引失效, 因为b+树存的是索引值, 你一计算, 或一使用函数, 那就无法确定在B+树的位置了.

- 在发展隐式类型转换时也会让索引失效, 因为隐式类型转换就会触发CAST()函数做转换, 这就间接调函数了.
- 再比如用or的时候, 出现了非索引列. or一边有索引, 一边没索引, 那就没法走索引了. 因为你不能同时做索引扫描和全表扫描. 所以直接退化为全表扫描看是否符合条件了.

索引失效的例子还有很多, 但是只要你理解索引查找的基本原理, 这些索引失效你就都能看懂了.

那索引这么快, 给全加上不就好了?

索引就像字典里的目录, 如果你加过多索引, 那目录就会越来越多了, 占得空间就越来越大了. 如果目录占的空间都赶上正文了, 那还要正文干嘛? 那就有问题了.

而且索引会降低增删改的性能, 因为你增删改了数据, 你同时也要去更新对应的索引.

所以索引肯定不是越多越好, 也不是万能的.

那怎么加索引比较好呢

- 数据量大的, 查询频繁的适合加索引. 经常where, order by, group by的列建立索引, 增删改频繁的就不适合加索引了.
- 对于选择区分度高的列做索引. 身份证号适合索引, 性别就不适合加索引
- 对于比较长的字符串, 可以使用前缀索引, 能够避免索引太大占用太多空间
- 尽量使用联合索引, 而不是单列索引, 联合索引很多时候可以覆盖索引, 避免回表

- 要注意一张表的索引不要太多, 最多别超过5个, 否则会大大降低插入删除的效率.

那怎么知道要不要加索引呢? 那怎么判断是否走索引呢?

- 可以打开MySQL的慢查询日志, 慢查询日志记录了执行时间超过阈值的所有查询语句. 这就能找到慢sql了.
- 然后可以使用explain执行计划来对慢 SQL 进行分析, 查询是否走索引, 走了什么索引. 然后针对性地做优化即可.