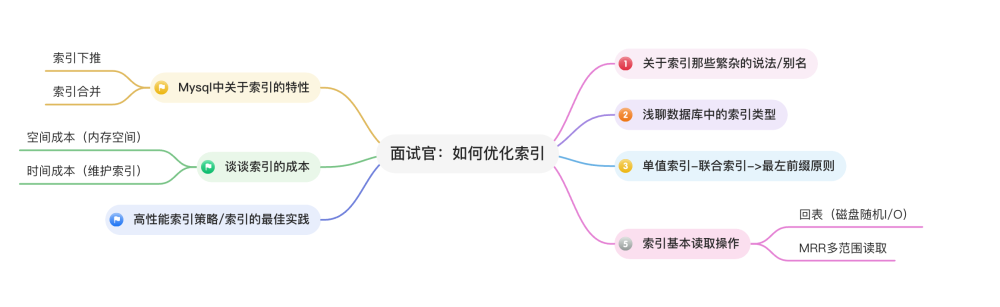
# 面试官：如何优化索引

大家好，我是🐟老师，今天我们一起来讨论一下如何优化索引。

文章阅读时长约15分钟。



在讨论优化索引前，我们应该先了解一下数据库中索引的类型，怎么样选择索引，索引的成本代价有那些，如何权衡性能和成本，索引内部存在的一些优化操作，到最后索引的最佳优化实践等。

## 关于索引那些繁杂的说法/别名

关于索引分类的问题，一直让我们很头疼，因为索引根据不同的维度，有很多种不同繁杂的说法。如果我们不懂这些繁杂的说法，那么在面试的时候不知道面试官在说什么，有可能会一脸懵逼。

从数据结构角度：

* B+树索引：查询效率比较平均，一般是3～4的I/O，支持范围查询，范围查询快

* 哈希索引：单条查询快，但是不支持范围查询，有hash碰撞问题

* FULLTEXT索引：目前MyISAM和InnoDB 引擎都支持了，分词算法，用于模糊检索字符串

* R-Tree索引：用于对GIS数据类型创建SPATIAL索引，B+树多维化

从物理存储角度：

* 聚集索引/聚簇索引：数据文件和索引文件存储在一个文件，也就是索引文件即数据文件，数据文件即是索引文件

* 非聚集索引/非聚簇索引：数据文件和索引文件分开存储，分别在不同的文件

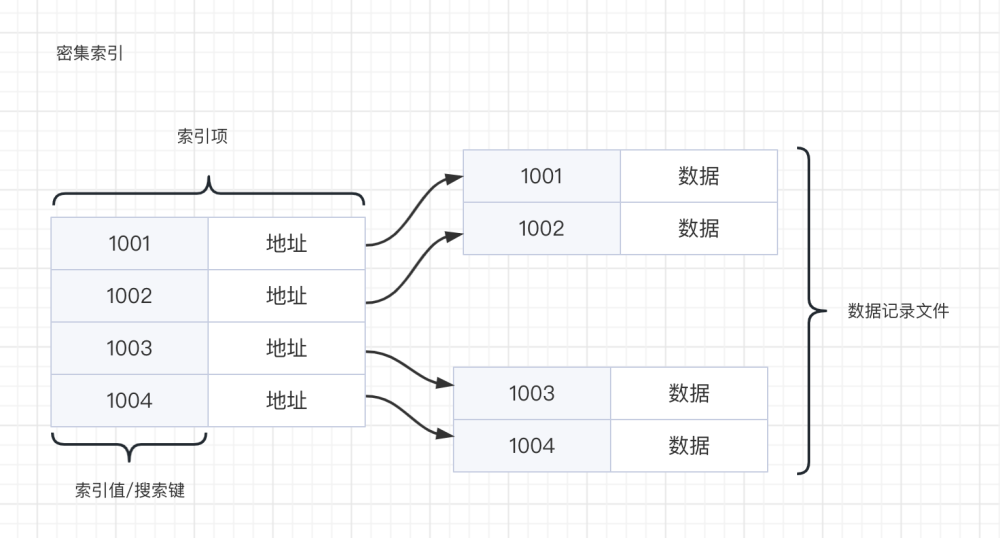
* 密集索引/稠密索引：

文件中的索引项包含搜索键值/索引值和指向磁盘上实际记录的指针。

每条索引项的行对应一条记录，数据查询速度非常快，但是维护成本大。

如何定位数据：

通过二分查找，搜索值 = 索引项的索引值，就能准确的定位到数据



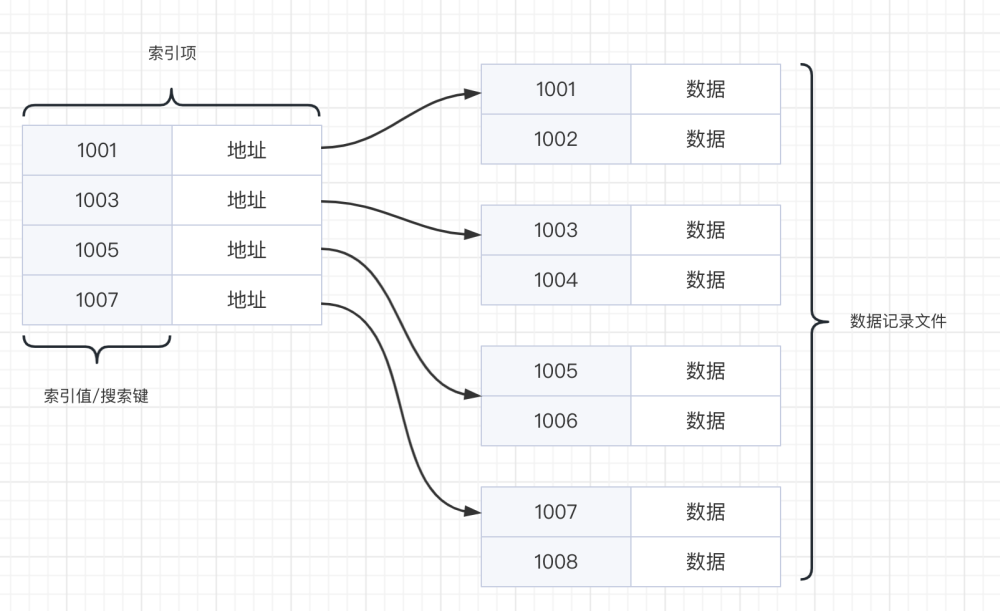
* 稀疏索引：

要求索引字段是按照顺序排序的，否则无法有序索引

没有为每条记录建立对应的索引项的行，数据查询速度较慢，但是存储空间小，维护成本低。

如何定位数据：

通过二分查找，搜索值 > 索引值中的最大值，然后根据记录所在的记录集合按照偏移量进行顺序查找，如果不在当前记录，则继续使用二分法查找，直到找到为止。



在myisam存储引擎中，不管是什么索引都是非聚集索引/非聚簇索引，稀疏索引。

在innodb存储引擎：有且只有一个聚集索引/聚簇索引，密集索引/稠密索引，而这个索引就是主键索引。innodb存储引擎里的普通二级索引是非聚集索引/非聚簇索引，稀疏索引。

从逻辑角度：

主键索引：使用主键来组织完整的行数据记录

普通索引：在创建索引时，不附加任何限制条件（唯一、非空等限制）

单列索引/单值索引/单一索引：构建索引树的索引列个数为1（创建索引的字段个数为1）

多列索引/联合索引/辅助索引/复合索引/二级索引：构建索引树的索引列个数超过1（创建索引的字段个数 > 1）

窄索引：构建索引树的索引列个数为1～2

宽索引：构建索引树的索引列个数超过2

唯一索引：构建的索引树里面的索引列的值是唯一的，不存在重复值

非唯一索引：构建的索引树里面的索引列的值是唯一的，存在重复值

## 浅聊数据库中的索引类型



索引的类型分为四种：

FULLTEXT（全文索引）[了解]：

* **主要用来查找文本中的关键字，只能在 CHAR、VARCHAR 或 TEXT 类型的列上创建**。**在 MySQL 中只有 MyISAM 存储引擎支持全文索引。**

* 全文索引允许在索引列中插入重复值和空值。

* 不过对于大容量的数据表，生成全文索引非常消耗时间和硬盘空间。

SPATIAL（空间索引）[了解]：

* **空间索引是对空间数据类型的字段建立的索引**

* 创建空间索引的列必须将其声明为 NOT NULL，**空间索引只能在存储引擎为 MyISAM 的表中创建**。

* 空间索引主要用于地理空间数据类型 GEOMETRY。这类索引相对来说很少会被用到。

NORMAL（普通索引）：

* MySQL 中最基本的索引类型，加快系统对数据的访问速度。

* **没有任何的限制，允许在定义索引的列中插入重复值和空值。**

UNIQUE（唯一索引）：

* 唯一索引与普通索引类似，但它的目的不是为了提高访问速度，而是**为了避免数据出现重复**。

* **唯一索引列的值必须唯一，允许有空值**。如果是组合索引，则列值的组合必须唯一。

## 单值索引-联合索引->最左前缀原则

从依据创建索引的字段的个数我们可以分为单值索引和联合索引。

单值索引：根据一个字段来建立索引

联合索引：根据超过一个字段来创建索引

联合索引如图所示：

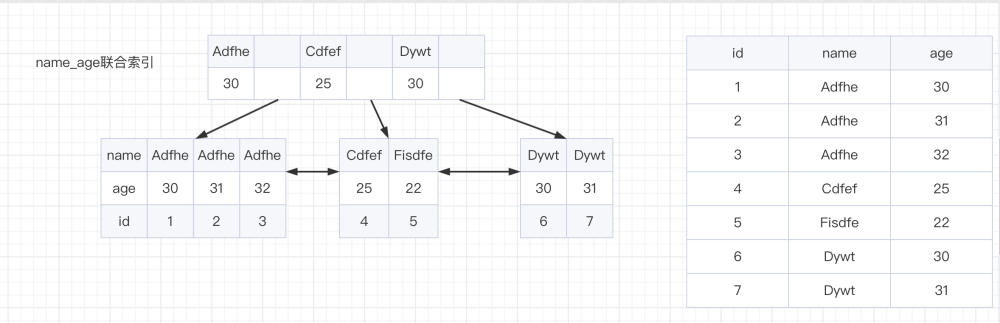
下面我们根据name和age按照顺序建立联合索引。

在B+树中，会按照第一个索引列name进行排序大小，在第一个索引列name的相同的有序的情况下在去排序下一个索引列age。

当我们执行select \* from user where name = "Adfhe" and age = 30的时候

* 遍历找到name = "Adfhe"的节点，然后根据空白指针往下找，根据age进行比对

* 最后获取对应的主键id，根据主键id到主键索引进行回表操作



最左前缀原则：

最左前缀原则这个说法只是针对联合索引的一种建立规则策略。

如果索引包含了多个列，那么我们最好遵循最左前缀原则。

我们都知道当我们建立多个索引列的时候，会按照索引列建立的顺序进行构建索引树。最左前缀原则指的是尽量要让查询会从索引的最左列开始匹配，按照每一个建立的索引的顺序，按照顺序依次匹配。期间不能跳过中间的某个索引列。且索引匹配严格按照第一个索引开始排序，依次往后。

假设我们建立了（name，task，age）的联合索引，按照最左前缀的原则要求：

执行sql： select \* from user where name = "a" and task = "b" and age =18

* 我们需要从最左列name，找到name匹配值

* 然后在name相同的情况基础下，task有序，然后找到task匹配值

* 然后在task匹配的情况基础下，age有序，然后找到age的匹配值

注意：如果我们没有按照顺序依次去遍历匹配的情况，跳过了中间某一个索引列的匹配，则该索引列后续的顺序都是乱的，因为在B+树中，联合索引的建立的有序性都是依据上一个索引列为相同的情况下，当前索引列才有序（这个跟B+树的存储结构原理有关）。

假设我们建立了（name，task，age）的联合索引，不按照最左前缀的原则要求，跳过中间task索引：

执行sql： select \* from user where name = "a" and age = 18

* 我们需要从最左列name，找到name匹配值

* 然后在name相同的情况基础下，task是有序，但是由于我们不需要查找task的值，只能节点往下全部遍历

* 由于age有序的前提是task为有序，因为task不需要查找指定的值，因此会全部遍历，导致age无序

因此这个索引只走了name有序，剩下的age和task都没有被利用到索引树加快检索效率。

假设我们建立了（name，task，age）的联合索引，不按照最左前缀的原则要求，跳过中间age索引：

执行sql： select \* from user where task = "b" and age = 18

* 我们需要从最左列name，找到name匹配值，由于name没有指定匹配的值，因此会全部遍历，导致task无序

* 由于task无序，则age也无序

因此这个索引没有走到任何的列，由于name无序，导致task无序，age无序，无法走索引列。

下面我们可以一起来尝试做一下以下的题目（面试官最爱问系列）：

假设我们建立了索引（a，b，c）

|  |  |
| --- | --- |
| where语句 | 索引是否被使用 |
| where a = 3 | Y,使用到a |
| where a =3 and b =5 | Y,使用到a，b |
| where a= 3 and b = 5 and c = 4 | Y,使用到a,b,c |
| where b = 3  where b = 3 and c = 5  where c =4 | N，因a无序，b无序  N，因a无序，b无序  N，因a无序，b无序，c无序 |
| where a = 3 and b > 4 and c =5 | Y,使用到a，b，使用不到c  因为b是某个范围不是某个确定的值，因此在b无序，c无序 |

## 索引基本读取操作

### 回表（磁盘随机I/O）

* 在innodb存储引擎中，由主键索引去组织完整用户记录数据，而辅助索引来组织其他字段的排序记录。

* 当我们查询需要通过辅助索引来查找数据的时候，innodb存储引擎会先到辅助索引去找到匹配的记录对应的主键。在通过主键到主键索引找到相关的完整用户记录，这个过程就叫做回表。

* 因此根据辅助索引来查询数据，需要使用到两颗B+树（辅助索引 + 主键索引）。

为什么需要回表，为什么不能把完整用户记录存储到二级索引呢（用二级索引来组织完整用户记录数据）？

* 如果我们把完整用户记录存储到二级索引的话，确实是不需要回表，按道理上来讲效率确实会高一些。

* 但是我们建立多个索引的情况下，每个索引都存储完整用户记录的话，会浪费大量的存储空间。

* 其次在项目中我们进行增删改某一条完整用户记录中的某个字段数据的时候，我们还需要去维护表中所有的B+树的索引叶子节点的数据，可想而知，这个性能很低的。

### MRR多范围读取（二级索引范围查询后主键无序，排序后回表）

MRR全名称：Disk-Sweep Multi-Range Read

当表的数据量很大的时候，在二级索引使用范围查询可能会导致的大量的磁盘随机I/O（回表导致）。

使用MRR，可以在扫描索引后，将索引对应的主键keys进行排序后，通过有序的主键去访问主键索引

这样可以减少对磁盘的随机I/O，进行对主键索引更多的顺序读

在回表中，我们根据辅助索引中筛选匹配，根据记录的主键值到主键执行回表操作。那么在辅助索引中筛选匹配的主键记录，存储两种情况。第一种情况主键值有序，这种一般是等值匹配查询（例如建立索引name，那么name字段等值匹配相同的情况，主键id是有序的，建立索引name其实跟建立索引（name，id）是一个意思）。第二种情况是主键值无序，这种一般是范围查询。

当我们每次回表都会去随机读取一个聚簇索引的页面（磁盘随机I/O）。这些随机I/O带来的性能开销比较大。因此就出现了MMR多范围读取来进行优化这一步骤。针对于以上第二种情况而言，每次读取一部分的主键值，进行排好序后在统一执行回表操作。这样虽然是随机I/O，但是有序相连数据，磁盘的页与页之间的寻址就不需要太久。

使用MMR多范围读取，相比每次从辅助索引读取一条主键值，就去回表会减少一些磁盘随机I/O。

## Mysql中关于索引的特性

### 索引下推(单个联合索引过滤数据，减少回表)

**指的是在辅助的联合索引查询的时候，同时过滤字段数据，有效的减少回表效果的操作**

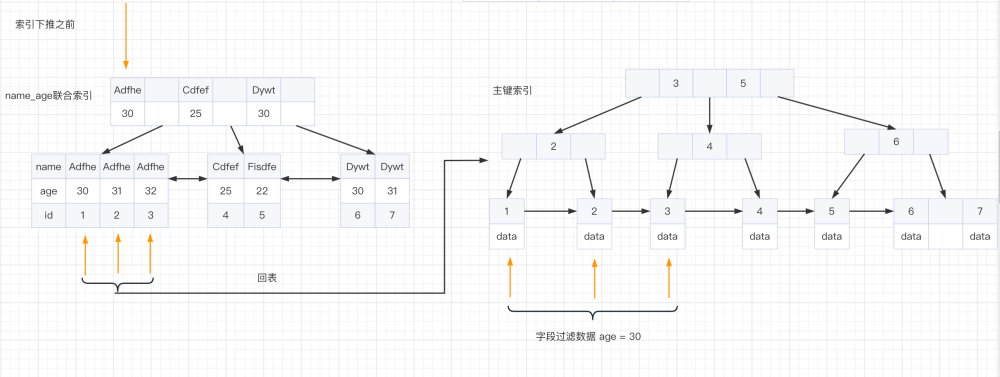
对于辅助的联合索引(name,age)，正常情况按照最左前缀原则。按照select \* from user where name like 'A%' and age = 30这种情况只会走name字段索引，但是根据name字段过滤完，得到的索引行里的age是无序的，无法很好的利用索引，那么索引下推就出现了。下面我们来对比一下有使用索引下推和没有使用索引下推的区别。

在MySQL5.6之前的版本，没有索引下推的概念：

* 首先会根据联合索引（二级索引）匹配符合"A"开头的索引节点

* 根据相关的索引节点对应的主键值进行依次回表

* 到主键索引查找对应的完整的行数据记录，进行数据过滤对比age的值是否符合要求

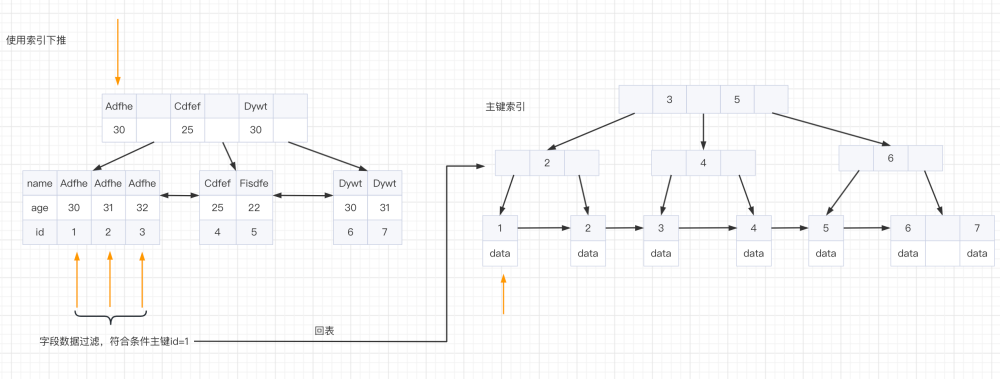


MySQL 5.6引入了索引下推优化（在索引遍历过程中，对所有字段进行数据过滤比对记录）：

* 首先会根据联合索引（二级索引）匹配符合"A"开头的索引节点

* 同时过滤字段age的数据，根据过滤后的索引节点进行回表

* 到主键索引查找对应的完整的行数据记录



对比是否使用索引下推：

* 不使用索引下推，则字段的数据过滤在**回表查询完整用户记录后进行过滤**。

* 使用索引下推，则是在回表前**通过联合索引进行匹配索引值的同时进行字段的数据过滤**。

* 因此使用索引下推可以有效的减少回表次数。回表的次数减少，磁盘I/O也相应减少，提高了性能。

提问1: 索引下推只能用在二级索引吗，为什么主键索引没有？

假设主键索引有，那么当我们回表，到主键索引查找对应的完整的行数据记录去过滤字段数据。因为主键索引是聚集索引，它的叶子节点包含了完整的行记录。按照我们这种玩法跟不使用索引下推的效果一样，而且并没有减少回表，也查询了一样多的行数据。

提问2: mysql一定会使用索引下推吗？

在mysql中，查询优化器会进行不同查询方式的成本估算来决定是否是否索引下推。

* 如果查询的过滤的结果集太大（并没有明显的减少回表的次数），不一定会使用索引下推，可能不使用索引下推的查找速度更快。

* 如果查询的过滤结果集很小（有效的减少回表次数），则可能会使用索引下推。

我们知道索引下推是联合索引没法走全部索引列，所以需要过滤数据，如果这样的话，我们可以根据业务需求建立一个更好的索引，让他们全部走索引，就不存在数据过滤问题，性能更高。

### 索引合并（多个二级索引记录的主键id过滤，减少回表）

指的是将多个索引的结果集的主键进行合并，减少回表次数

假设我们建立了name和age的二级索引，执行select \* from user where name = "Adfhe" and age = 30

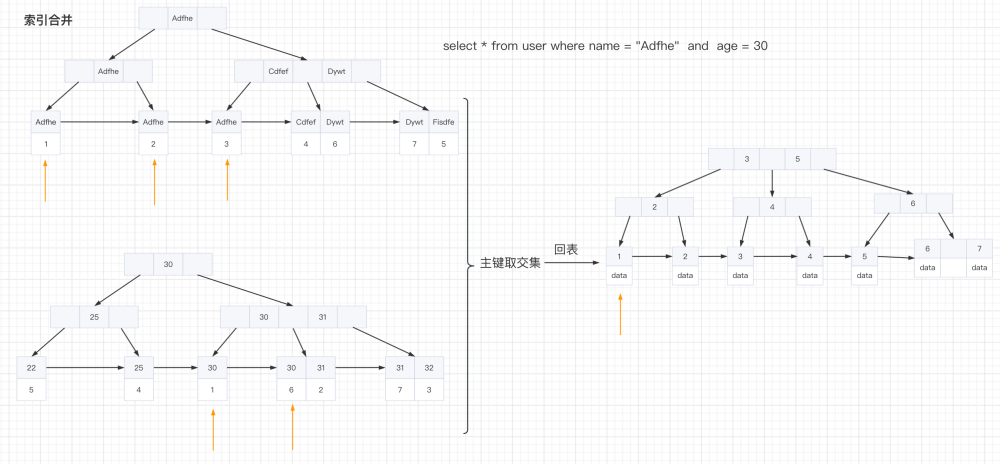
* 根据name二级索引从B+树中取出name = "Adfhe" 相关的记录（索引列 + 主键）

* 根据age二级索引从B+树中取出 age = 30 相关的记录（索引列 + 主键）

* 由于链接符是and，所以根据两部分索引的过滤记录按照主键取交集

* 根据交集的id = 1 进行到主键索引进行回表操作，根据主键查询完整行数据记录

关于索引合并，根据sql语句的连接符，and则取交集（Intersection），or则取并（Union）。目前当前案例演示的是and取交集



#### 普通索引合并需要满足的条件

**1.用于等值查询**

对于联合索引来说，在联合索引中的每个列都必须等值匹配，不能出现只匹配部分列的情况。 这种情况下，查询出来的索引的主键key是有序的。

假设查询使用索引合并的方式从a和b这两个二级索引中获取到的主键值分别是：

从a中获取到已经排好序的主键值：1、3、5

从b中获取到已经排好序的主键值：2、3、4

那么求交集的过程就是这样：

* 逐个取出这两个结果集中最小的主键值，如果两个值相等， 则加入最后的交集结果中

* 否则丢弃当前较小的主键值，再取该丢弃的主键值所在结果集的后一个主键值来比较，直到某个结果集中的主键值用完了，时间复杂度是O(n)。

取出两个结果集的最小主键值：1，2，丢弃最小的主键值1，a结果集为3，5，b结果集为2，3，4

获取a结果集中的最大主键值5对比b结果集中的2，不相同

获取a结果集中的主键值3对比b结果集中的2，不相同

进行下一轮b结果集中的3进行对比，按照此顺序对比

但是如果从各个二级索引中查询出的结果集并不是按照主键排序的话，那就要先把结果集中的主键值排序完再来做上边的那个过程，就比较耗时了。

按照有序的主键值去回表取记录有个专有名词，叫：Rowid Ordered Retrieval，简称 ROR

**2.主键列可以是范围查询**

因为如果按照联合索引，进行排序的话，获取到的主键key是有序的，我们可以根据主键列的范围查询（例如id > 5 ），进行过滤主键

#### 特殊的sort索引合并（需要将无序转为有序）

相对于普通的索引合并（二级索引查询之后的主键都是有序的），当多个二级索引的索引列是范围查询的话，得出来的主键id是无序的，那么会将无序的主键id进行排序完，后进行索引取交集或者并集，最后在回表。特殊的索引合并比普通的索引合并多了一步二级索引记录的主键值排序的过程。

满足了索引合并的条件就会触发索引合并吗？

当查询优化器根据二级索引中获取的记录数比较少，通过索引合并后进行访问的代价比全表扫描更小时才会使用索引合并。

#### 对比是否使用索引合并

不使用索引合并，只使用一个索引：

按照我们目前建立的索引和查询的sql语句，如果我们不使用索引合并只是读取一个索引，那么我们需要根据name二级索引过滤数据匹配name的值后的所有主键id，根据主键id到主键索引进行回表，查询完整行数据记录。之后需要在进行过滤一遍 age的值**（单个索引查询 + 回表过滤查询条件）**。

使用索引合并，多个二级索引取交集：

按照不同的筛选条件读取对应的二级索引，将多个索引的主键值取交集，最后进行回表操作（**多个二级索引查询过滤主键 + 回表）**。

### 使用联合索引代替索引合并

大部分使用索引合并的原因是多个二级索引为单值索引，所以需要进行主键的合并。如果我们把索引列都放在一起，建立一个联合索引，直接使用联合索引就完事了。使用联合索引又快又好，不需要像索引合并一样读取多颗B+树，然后合并结果！！！

提问1 ：哪种方式性能更优？

* 使用多个二级索引可以减少回表的次数，但是查询多个二级索引比查询一个二级索引的性能要慢。

* 读取二级索引的操作是顺序I/O，而回表操作是随机I/O。顺序I/O是比随机I/O要快几个数量级。关于哪种性能更优没有绝对的说法，这取决于所需回表的次数有多少，回表次数越多，随机I/O越多越慢。

单索引成本： 一个索引的检索消耗 + 回表消耗

多索引成本： 多个索引取交集或者并集的消耗 + 回表消耗

* 如果只读取一个索引的回表的次数很多，且多个二级索引取交集的回表次数很少。且回表带好的性能消耗 > 二级索引取交集的性能消耗，那么索引合并的性能更优。

* 如果只读取一个索引的回表的次数很少，且回表带好的性能消耗 < 二级索引取交集的性能消耗，那么只使用一个索引的性能更优。

## 谈谈索引的成本

### 空间成本（内存空间）

每当我们建立一个索引，就要构建一颗对应的B+树，每一棵B+树的每一个节点都是一个数据页，一个页默认会占用16KB的存储空间，一棵很大的B+树由许多数据页组成会占据很多的存储空间。

### 时间成本（维护索引）

B+树每层节点都是按照索引列的值从小到大的顺序排序而组成了双向链表。而节点内部的记录按照索引列的值从小到大的顺序而形成了一个单向链表。 每次对表中的数据进行增、删、改操作时，都需要去维护各个B+树。因为增、删、改操作可能会对节点和记录的排序造成破坏，所以存储引擎需要额外的时间进行一些记录移位，页面分裂、页面回收的操作来维护好节点和记录的排序，这些维护操作也会产生性能影响。

## 高性能索引策略/索引的最佳实践

讨论话题前的思考: 有索引一定会走索引吗？

有索引不一定会走索引，因为mysql会根据查询优化器，对表中的数据量多少，筛选匹配后的数据量多少，全表扫描，使用各个索引的成本，回表的记录数多少等等进行成本分析，以此来选择最优的方式来执行sql。

正确地创建和使用索引是实现高性能查询的基础，下面我们来看看如何创建高性能的索引：

### 1.选择区分度高的字段建立索引

索引字段的区分度/离散度越高，则检索的次数越少。如果字段的区分度不高，可能整个查询下来会查询超过一半多的数据，同时这样索引页内部的用户分组没法快速筛选数据，这样也就没法快速利用二分查找的算法。此外区分度高的索引可以利用好二分查找算法，检索性能更高，在检索数据时过滤掉更多的数据，减少回表的次数。

那么如何判断表内某些字段的离散度高不高呢？我们可以通过数据库表字段不重复的个数除于所有行数来计算，值越大说明离散度越高。

SELECT COUNT(DISTINCT 字段) / COUNT(\*) from 表;

### 2.在允许的情况下，尽量选择占用存储空间少的字段建立索引

* 数据类型越小，在查询时进行的比较操作越快（CPU层次)

* 数据类型越小，索引占用的存储空间就越少，在一个数据页内就可以放下更多的记录，从而减少磁盘I/0带来的性能损耗，也就意味着可以把更多的数据页缓存在内存中， 从而加快读写效率。

### 3.where语句中经常被使用到的字段应该建立索引，分组字段或排序字段应创建索引，多表连接字段应该创建索引

只为出现在where 子句中的列、连接子句中的连接列创建索引，而出现在查询列表中的列一般就没必要建立索引了，除非是需要使用覆盖索引。又或者为出现在order by或group by子句中的列创建索引。

### 4.更新频繁的字段不适合建立索引

如果这个字段更新很频繁，那么需要经常去维护这个字段建立的相关索引，最坏的结果可能会导致页分裂，合并等，性能开销很大。

### 5.建立索引遵循最左前缀原则

遵循最左前缀原则，能够让查询条件尽最大可能的命中更多的索引列，最大化的命中更多索引列。

### 6.尽量使用前缀索引

有时候我们需要对一些字符串很大的字段建立索引 ，这会让索引变的很大，内存的开销很大，cpu操作大的数据也很慢且一个数据页存储的字段数据更少，查询更慢。我们可以将字段的开始的部分字符建立索引，这样大大减少了内存开销，也提高了索引的效率。选择使用前缀索引的同时，也失去了字段的区分度。因此我们要保证前缀索引有足够高的区分度，同时也不能保证索引太大。

我们可以通过计算，来找到合适建立的前缀长度。通过如下sql不断的调整前缀长度的大小，来查看离散度的大小的增幅变化，从而来找到最适合的前缀长度大小。

SELECT COUNT(DISTINCT LEFT(字段,前缀长度))/COUNT(\*) FROM 表;

### 7.尽量设计多列索引，避免创建重复的索引以及删除未使用的索引

* 尽量设计多列索引，根据索引列的区分度，查询字段的频率，sql进行权衡索引列的排序位置。

* 当联合索引包含了单值索引，则单值索引重复，例如索引（a，b）和索引a，则索引a重复删除即可

* 在项目中可能会因为sql的查询慢需要进行优化改动，不免会出现某些索引突然用不到的情况，我们应该删除，减少带好的内存开销。

### 8.尽可能设计三星索引

三星索引由《高性能MySQL》作者提出的一个概念，如果满足了三星索引则可以认为这是一个性能很高的索引

第一星（减少数据检索范围）：如果一个查询相关的索引行是相邻的或者至少相距足够靠近的话，必须扫描的索引片宽度就会缩至最短，也就是说，让索引片尽量变窄，也就是我们所说的索引的扫描范围越小越好。尽量将where的等值查询，区分度高的列放在前面。

第二星（排序星，索引按照查询语句中order by排序）：当查询需要排序，order by 中的排序和索引顺序一致，它能够减少耗时的文件排序。我们可以按照sql给order by相关的字段按照顺序加入索引列中。

第三星（宽索引星，覆盖索引）：索引行包含查询语句中所有的列。这样就不需要回表操作（磁盘随机I/O慢），只需要访问一个二级索引即可。我们可以将相关的查询列加入到索引当中去。

在大部分的情况下，我们不能同时满足三星，因此我们需要在三星里面做权衡。

select name,age,city from user where name="aaa" and age = 20;

最佳的索引是建复合索引 (name, age, city) ，这是一个三星索引。

第一星：很明显name的区分度比age高，为了过滤尽可能多的行，需要把把选择性高的索引放在前面name，age。减少索引片的大小以减少需要扫描的数据行

第二星：也就是说，当经过了name,age的筛选之后，筛选出来的行本身就是已排序的city。避免排序，减少磁盘 IO 和内存的使用；

第三星：通过宽索引实现索引覆盖，避免回表进行磁盘随机 I/O

在一些复杂的查询中，我们没法同时满足三星，只能在三星之中做权衡。

SELECT id, name, age FROM user  
WHERE age BETWEEN 18 AND 25  
 AND city = "aaa"  
ORDER BY name;

在这个情况下，city是等值索引，为了减少索引的检索范围，我们需要建立索引（city，age），此时获得一星。为了获取第三颗星，我们需要建立索引（city，age，name）。但是由于age是范围查询，导致name是是无序的，所以没法满足第二颗星。为了避免内存排序操作，我们更改索引为（city，name，age），满足了第二颗星和第三颗星。因此我们需要在第一颗星和第二颗之间做抉择。

因此大部分情况（复杂查询）下，我们需要在三颗星之间进行抉择。按照常规的思想来说，我们一般优先满足第三颗星，减少回表的随机I/O，性能提升最大。其次优先满足第一颗星，减少大部分的数据检索范围，让下一次查询减少数据量更少。当然这也不是绝对的标准答案，需要查看当前查询数据的数据量，假设满足第一颗星能筛选出大部分的数据的性能大于满足第二颗星的排序的性能，则选择第二颗星。需要根据查询的重点。

## 🎣

### 提问1: 如何设计一个高性能索引

1. 选择区分度高的字段建立索引

1. 在允许的情况下，尽量选择占用存储空间少的字段建立索引

1. where字句中经常被使用到的字段应该建立索引，分组字段或排序字段应创建索引，多表连接字段应创建索引

1. 更新频繁的字段不适合建立索引

1. 建立索引遵循最左前缀原则

1. 尽量使用前缀索引

1. 尽量设计多列索引，避免创建重复的索引以及删除未使用的索引

1. 尽可能设计三星索引