MySQL的索引（中）

原创：小孩子 我们都是小青蛙 4月12日

连着发了几天文章，从我收到的反映来说，大家觉着还不错，可以很清晰的看到知识的脉络，但是这个还不错是针对传统的文章的无聊、不明确、完全不考虑考虑用户体验的角度上对比出来的。掌握一门知识还是不容易的，有的同学认为大致看一遍我的文章就可以了解到原理，所以毛毛躁躁，十来分钟读完一篇文章，最后再读下一篇的时候发现有的问题不会，然后就各种问，最后回头一看原来文章里已经写过了，而且使用红字飘红的那种！所以再次提醒大家，我的文章不像别的技术公众号那种讨论程序猿八卦，或者啥啥啥公司发生了啥事儿，谁家的老总又换了，或者java10出来了通知你一下，或者针对某一个知识点做一个很短的介绍，还介绍不清楚~。我的每一篇文章就是一个主题，拿MySQL举例来说，如果你想熟悉MySQL的基础知识，那大约需要了解十几个或者几十个主题的东西才算入门，比如基本操作、字符集、索引、事务、权限、数据目录、复制、备份、日志等等等等，在介绍这些主题的时候，我会严格遵守下边的原则：

- 不会在很短的篇幅里连续介绍很多个新概念。
- 后边文章出现的概念绝对不会出现在前边的文章里。
- 有的文章存在依赖关系，每篇文章的开头我都会注明你在读本篇文章时需要读过的文章，如果你没读过，那必须先读完后再来读本篇文章。
- 文章尽量用通俗易懂的语言描述，用汉字描述不来的我就去画图了。
- 我的文章尽量覆盖的越详细越好，如果你觉得某个重要的概念还没有被介绍，那它应该会在后边的文章中出现，我不想同时出现多个概念影响用户体验。
- 同一篇文章中的内容是强耦合的，文章前边出现的概念很有可能被后边使用到，所以不要跳着看，不要跳着看！

你可以看到，一篇既简单又详细的的文章背后都是满满的套路，而对于用户来说，只要你不是智障，一行一行的把我写的文章读完，相信我的努力能换来你学习的事半功倍，我对你的要求只有一点：认真点，不要跳着看！

开始正文，非常重要的几条阅读建议：

1. 最好使用电脑观看。

2. 如果你非要使用手机观看，那请把字体调整到最小，这样观看效果会好一些。

3. 碎片化阅读并不会得到真正的知识提升，要想有提升还得找张书桌认认真真看一会书，或者我们公众号的文章🥹🥹。

4. 如果觉得不错，各位帮着转发转发，如果觉得有问题或者写的哪不清晰，务必私聊我～

5. 本公众号的文章都是需要被系统性学习的，这篇文章是建立在之前的几篇文章之上的，如果你没有读过，**务必读后再看本篇**，要不然可能会有阅读不畅的体验：

- InnoDB记录存储结构
- InnoDB数据页结构
- MySQL的索引

上集回顾

上集主要唠叨了 `InnoDB` 存储引擎的索引，我们必须熟悉下边这些结论：

- 每个索引都对应一棵 `B+` 树，所有真实的用户记录都存储在 `B+` 树的叶子节点，所有 `目录项记录` 都存储在内节点。

- `InnoDB` 存储引擎会自动为主键（如果没有它会自动帮我们添加）建立 `聚簇索引` ，聚簇索引的叶子节点包含完整的用户记录。

- 我们可以为自己感兴趣的列建立 `二级索引` ， `二级索引` 的叶子节点包含的记录由 `索引列 + 主键` 组成，所以如果想通过 `二级索引` 来查找完整的用户记录的话，需要通过 `回表` 操作，也就是在通过 `二级索引` 找到主键值之后再到 `聚簇索引` 中查找完整的用户记录。

- `B+` 树中每层节点都是按照索引列值从小到大的顺序排序而组成了双向链表，而且每个页内的记录（不论是用户记录还是目录项记录）都是按照索引列的值从小到大的顺序而形成了一个单链表。如果是 `联合索引` 的话，则节点和记录按照先按照 `联合索引` 前边的列排序，如果该列值相同，再按照 `联合索引` 后边的列排序。

- 通过索引查找记录是从 `B+` 树的根节点开始，一层一层向下搜索。由于每个页面都按照索引列的值建立了 `Page Directory` （页目录），所以在这些页面中的查找非常快。

如果你读上边的几点结论有些一头雾水的话，那本篇文章不适合你，回过头先去看前边几篇，尤其是MySQL的索引这一篇。

索引的代价

在熟悉了 `B+` 树索引原理之后，本篇文章的主题是唠叨如何更好的使用索引，虽然索引是个好东西，可不能乱建，在介绍如何更好的使用索引之前先要了解一下使用这玩意儿的代价，它在空间和时间上都会拖后腿：

- 空间上的代价

  这个是显而易见的，每建立一个索引都为要它建立一棵 `B+` 树， `B+` 树的每一个节点都是一个数据页，一个页可是要占用 `16KB` 的存储空间的，一棵很大的 `B+` 树由许多数据页组成，那可是很大的一片存储空间呢。

- 时间上的代价

  我们讲过，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，而且每个页内的记录（不论是用户记录还是目录项记录）都是按照索引列的值从小到大的顺序而形成了一个单向链表。在对表中数据进行增、删、改操作的时候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、页面回收啥的操作来维护好节点和记录的排序。如果我们建了许多索引，每个索引对应的 `B+` 树都要进行相关的维护操作，这还能不给性能拖后腿么？

B+树索引适用的条件

下边我们将唠叨许多种让 `B+` 树索引发挥最大效能的技巧和注意事项，不过大家要清楚，所有的技巧都是源自你对 `B+` 树索引的理解，所以如果你还不能保证对 `B+` 树索引充分的理解，那么先回去温习一遍再来吧，要不然读文章对你来说是一种折磨。首先， `B+` 树索引并不是万能的，并不是所有的查询语句都能用到我们建立的索引。下边介绍几个我们可以使用 `B+` 树索引来进行查询的情况。为了故事的顺利发展，我们需要先创建一个表，这个表是用来存储人的一些基本信息的：

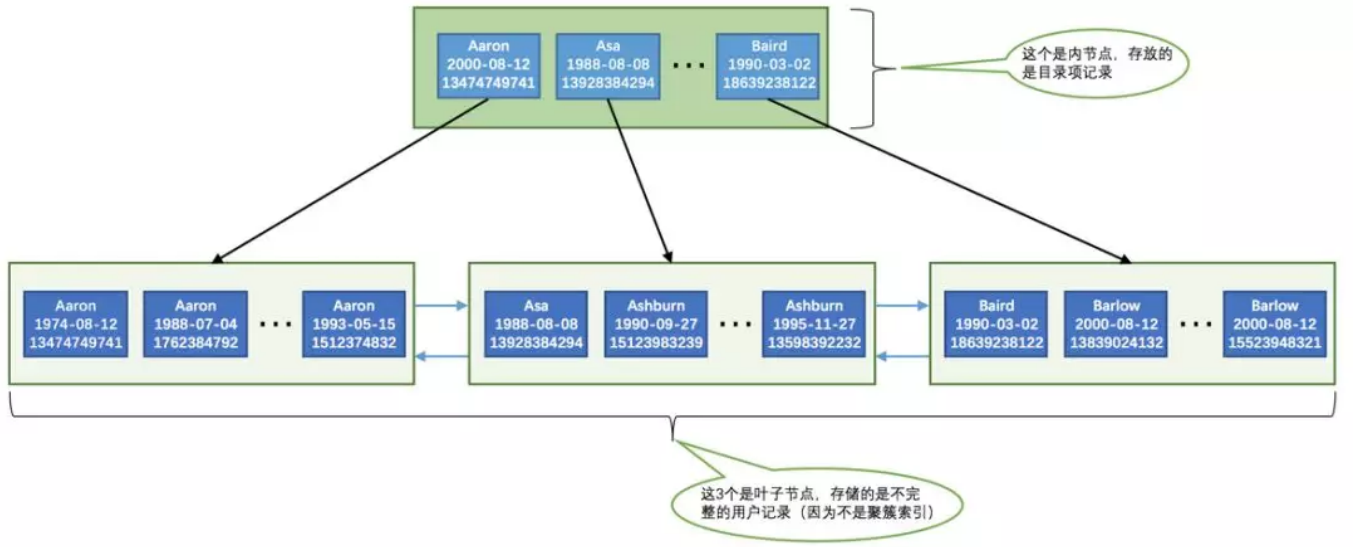
```
CREATE TABLE person_info(
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    KEY idx_name_age_birthday (name, birthday, phone_number)
);
```

对于这个 `person_info` 表我们需要注意两点：

- 我们并没有在表中定义主键，所以 `innoDB` 存储引擎会默认为我们的表添加一行 `row_id` 列作为主键。

- 重点注意我们定义的索引 `idx_name_age_birthday` ，它是由3个列组成的联合索引。所以在这个索引对应的 `B+` 树的叶子节点处存储的记录只保留 `name` 、 `birthday` 、 `phone_number` 这三个列的值以及主键值。

下边我们画一下索引 `idx_name_age_birthday` 的示意图，不过既然我们已经掌握了 `InnoDB` 的 `B+` 树索引原理，那我们在画图的时候为了让图更加清晰，所以在记录结构中就省略一些不必要的部分，比如单链表的箭头、记录的主键值啥的，只保留 `name` 、 `birthday` 、 `phone_number` 这三个列的值，所以示意图就长这样（留心的同学看出来了，这其实和《高性能MySQL》里举的例子的图差不多，我觉得这个例子特别好，所以就借鉴了一下）：



为了方便大家理解，我们特意用不同颜色区分了一下内节点和叶子节点。并且用不同颜色区分了一下内节点中的目录项记录和叶子节点中的用户记录（由于不是聚簇索引，所以用户记录是不完整的，缺少 `country` 列的值）。从图中可以看出，这个 `idx_name_age_birthday` 索引对应的 `B+` 树中节点和记录的排序方式就是这样的：

- 先按照 `name` 列的值进行排序。
- 如果 `name` 列的值相同，则按照 `birthday` 列的值进行排序。
- 如果 `birthday` 列的值也相同，则按照 `phone_number` 的值进行排序。

这个排序方式十分、特别、非常、巨、very very very重要，因为只要节点和记录是排好序的，我们就可以通过二分法来快速定位查找。下边的内容都仰仗这个图了，大家对照着图理解。

全值匹配

如果我们的搜索条件中的列和索引列一致的话，这种情况就称为全值匹配，比方说下边这个查找语句：

```
SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday = '1990-09-27' AND phone_number = '15123983239';
```

我们建立的 `idx_name_age_birthday` 索引包含的3个列在这个查询语句中都展现出来了，而且**搜索条件中列的顺序和定义索引列时的顺序是一致的**。大家可以想象一下这个查询过程：

- 因为 `B+` 树的数据页和记录先是按照 `name` 列的值进行排序的，所以可以很快定位 `name` 列的值是 `Ashburn` 的记录位置。
- 在 `name` 列相同的记录里又是按照 `birthday` 列的值进行排序的，所以在 `name` 列的值是 `Ashburn` 的记录里又可以快速定位 `birthday` 列的值是 `'1990-09-27'` 的记录。
- 如果很不幸， `name` 和 `birthday` 列的值都是相同的，那记录是按照 `phone_number` 列的值排序的，所以联合索引中的三个列都可能被用到。

如果搜索条件中的列的顺序和索引列的顺序不一致就不太好了，比方说先对 `birthday` 列的值进行匹配的话，由于 `B+` 树中的数据页和记录是先按 `name` 列的值进行排序的，不能直接使用 `二分法` 快速定位记录，所以只能扫描所有的记录页。

匹配左边的列

其实在我们的搜索语句中也可以不用包含全部联合索引中的列，只包含左边的就行，比方说下边的查询语句：

```
SELECT * FROM person_info WHERE name = 'Ashburn';
```

或者包含多个左边的列也行：

```
SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday = '1990-09-27';
```

那为什么搜索条件中必须出现左边的列才可以使用到这个 `B+` 树索引呢？比如下边的语句就用不到这个 `B+` 树索引么？

```
SELECT * FROM person_info WHERE birthday = '1990-09-27';
```

是的，的确用不到，因为 B+ 树的数据页和记录先是按照 name 列的值排序的，你直接根据 birthday 的值去查找，臣妾做不到呀~ 那如果我就想在只使用 birthday 的值去通过 B+ 树索引进行查找咋办呢？这好办，你再对 birthday 列建一个 B+ 树索引就行了，创建索引的语法不用我唠叨了吧，如果不会出门左转看别的文章。

但是需要特别注意的一点是，**搜索条件中的列的顺序必须和索引列的定义顺序一致**，比方说索引列的定义顺序是 name 、 birthday 、 phone_number ，如果我们的搜索条件中只有 name 和 phone_number ，而没有 birthday ，这样只能用到 name 列的索引， birthday 和 phone_number 的索引就用不上了（因为 name 值相同的记录先按照 birthday 进行排序， birthday 值相同的记录才按照 phone_number 值进行排序）。

匹配列前缀

对于字符串类型的索引列来说，我们没必要对该列的值进行精确匹配，只匹配它的前缀也是可以的，因为前缀本身就已经是排好序的。比方说我们想查询名字以 'As' 开头的记录，那就可以这么写查询语句：

```
SELECT * FROM person_info WHERE name LIKE 'As%';
```

B+ 树中的数据页和记录都先是按照 name 列排序的，只给出前缀也是可以通过 二分法 快速定位的。但是需要注意的是，如果只给出后缀或者中间的某个字符串，比如这样：

```
SELECT * FROM person_info WHERE name LIKE '%As%';
```

MySQL 就无法通过 二分法 来快速定位记录位置了，因为字符串中间有 'As' 的字符串并没有排好序，所以只能全表扫描了。

匹配范围值

回头看我们 idx_name_age_birthday 索引的 B+ 树示意图，**所有记录都是按照索引列的值从小到大的顺序排好序的**，所以这极大的方便我们查找索引列的值在某个范围内的记录。比方说下边这个查询语句：

```
SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow';
```

由于 B+ 树中的节点和数据页是先按 name 列排序的，所以我们上边的查询过程其实是这样的：

- 找到 `name` 值为 `Asa` 的记录。
- 找到 `name` 值为 `Barlow` 的记录。
- 哦啦，由于所有记录都是由连链表连起来的（记录之间用单链表，数据页之间用双链表），所以他们之间的记录都可以很容易的取出来喽~
- 找到这些记录的主键值，再到 `聚簇索引` 中 `回表` 查找完整的记录。

不过进行范围查找的时候需要注意，**如果对多个列同时进行范围查找的话，只有对索引最左边的那个列进行范围查找的时候才能用到 `B+` 树索引**，比方说这样：

```
SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow' AND birthday > '1980-01-01';
```

为啥不能呢？因为上边这个查询可以分成两个部分：

1. 通过条件 `name > 'Asa' AND name < 'Barlow'` 来对 `name` 进行范围，查找的结果可能有多条 `name` 值不同的记录，

2. 对这些 `name` 值不同的记录继续通过 `birthday > '1980-01-01'` 条件继续过滤。

但是值得注意的是，只有 `name` 值相同的情况下才能用 `birthday` 列的值进行排序，也就是说**通过 `name` 进行范围查找的结果并不是按照 `birthday` 列进行排序的**，所以在搜索条件中继续以 `birthday` 列进行查找时是用不到 `B+` 树索引的。

精确匹配某一列并范围匹配另外一列

对于同一个联合索引来说，虽然对多个列都进行范围查找时只能用到最左边那个索引列，但是如果左边的列是精确查找，则右边的列可以进行范围查找，比方说这样：

```
SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday > '1980-01-01' AND birthday < '2000-12-31' AND phone_number > '15100000000';
```

这个查询的条件可以分为3个部分：

1. `name = 'Ashburn'` ，对 `name` 列进行精确查找，当然可以使用 `B+` 树索引了。

2. `birthday > '1980-01-01' AND birthday < '2000-12-31'` ，由于 `name` 列是精确查找，所以通过 `name = 'Ashburn'` 条件查找后得到的结果的 `name` 值都是相同的，它们会再按照 `birthday` 的值进行排序。所以此时对 `birthday` 列进行范围查找是可以用到 `B+` 树索引的。

3. `phone_number > '15100000000'` ，通过 `birthday` 的范围查找的记录的 `birthday` 的值可能不同，所以这个条件无法再利用 `B+` 树索引了，只能遍历上一步查询得到的记录。

用于排序

我们在写查询语句的时候经常需要对查询出来的记录按照某种规则进行排序，对于不适用 `B+` 树索引进行排序的情况，我们只能把记录都加载到内存中，再用一些排序算法，比如快速排序、归并排序、吧啦吧啦排序等等在内存中对这些记录进行排序，然后再把排好序的结果集返回到客户端。但是如果 `ORDER BY` 子句里使用到了我们的索引列，就有可能省去在内存中排序的步骤，比如下边这个简单的查询语句：

```
SELECT * FROM person_info ORDER BY name, birthday, phone_number LIMIT 10;
```

这个查询的结果集需要先按照 `name` 值排序，如果记录的 `name` 值相同，则需要按照 `birthday` 来排序，如果 `birthday` 的值相同，则需要按照 `phone_number` 排序。大家可以回过头去看我们建立的 `idx_name_age_birthday` 索引的示意图，因为这个 `B+` 树索引本身就是排好序的，所以直接从索引中提取数据就好了。简单吧？是的，索引就是这么diao~

但是有个问题需要注意， `ORDER BY` 的子句后边的列的顺序也必须按照索引列的顺序给出，如果给出 `ORDER BY phone_number, birthday, name` 的顺序，那也是用不了 `B+` 树索引，这种颠倒顺序就不能使用索引的原因我们上边详细说过了，这就不赘述了。同理， `ORDER BY name` 、 `ORDER BY name, birthday` 这种匹配索引左边的列的形式可以使用部分的 `B+` 树索引。

用于分组

有时候我们为了方便统计表中的一些信息，会把表中的记录按照某些列进行分组。比如下边这个分组查询：

```
SELECT name, birthday, phone_number, COUNT(*) FROM person_info GROUP BY name,
birthday, phone_number
```

这个查询语句相当于做了3次分组操作：

1. 先把记录按照 `name` 值进行分组，所有 `name` 值相同的记录划分为一组。

2. 将每个 `name` 值相同的分组里的记录再按照 `birthday` 的值进行分组，将 `birthday` 值相同的记录放到一个小分组里，所以看起来就像在一个大分组里又化分了好多小分组。

3. 再将上一步中产生的小分组按照 `phone_number` 的值分成更小的分组，所以整体上看起来就像是先把记录分成一个大分组，然后把 `大分组` 分成若干个 `小分组` ，然后把若干个 `小分组` 再细分成更多的 `小小分组` 。

然后针对那些 `小小分组` 进行统计，比如在我们这个查询语句中就是统计每个 `小小分组` 包含的记录条数。如果没有索引的话，这个分组过程全部需要在内存里实现，而如果有了索引的话，恰巧这个分组顺序又和我们的 `B+` 树中的索引列的顺序是一致的，而我们的 `B+` 树索引又是按照索引列排好序的，这不正好么，所以可以直接使用 `B+` 树索引进行分组。

和使用 `B+` 树索引进行排序是一个道理，分组列的顺序也需要和索引列的顺序一致，也可以只使用索引列中左边的列进行分组，吧啦吧啦的~

如何挑选索引

上边我们以 `idx_name_age_birthday` 索引为例对索引的适用条件进行了详细的唠叨，下边看一下我们在建立索引时或者编写查询语句时就应该注意的一些事项。

只为用于搜索、排序或分组的列创建索引

也就是说，只为出现在 `WHERE` 子句中的列、连接子句中的连接列，或者出现在 `ORDER BY` 或 `GROUP BY` 子句中的列创建索引。而出现在查询列表中的列就没必要建立索引了：

```
SELECT birthday, country FROM person_name WHERE name = 'Ashburn';
```

像查询列表中的 `birthday` 、 `country` 这两个列就不需要建立索引，我们只需要为出现在 `WHERE` 子句中的 `name` 列创建索引就可以了。

考虑列的基数

`列的基数` 指的是某一列中不重复数据的个数，比方说某个列包含值 `2, 5, 8, 2, 5, 8, 2, 5, 8` ，虽然有 `9` 条记录，但该列的基数却是 `3` 。也就是说，**在记录行数一定的情况下，列的基数越大，该列中的值越分散，列的基数越小，该列中的值越集中**。这个 `列的基数` 指标非常

重要，直接影响我们是否能有效的利用索引。假设某个列的基数为 1 ，也就是所有记录在该列中的值都一样，那为该列建立索引是没有用的，因为所有值都一样就无法排序，无法进行二分法查找了~ 所以结论就是：最好为那些列的基数大的列建立索引，为基数太小列的建立索引效果可能不好。

索引列的类型尽量小

我们在定义表结构的时候要显式的指定列的类型，以整数类型为例，有 TINYINT 、 MEDIUMINT 、 INT 、 BIGINT 这么几种，它们占用的存储空间依次递增，能表示的整数范围当然也是依次递增，我们这里所说的 类型大小 指的就是该类型表示的数据范围的大小。如果我们想要对某个整数列建立索引的话，在表示的整数范围允许的情况下，尽量让索引列使用较小的类型，比如我们能使用 INT 就不要使用 BIGINT ，能使用 MEDIUMINT 就不要使用 INT ~ 这是因为：

- 数据类型越小，在查询时进行的比较操作越快（这是CPU层次的东东）
- 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/O 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。

这个建议对于表的主键来说更加适用，因为不仅是聚簇索引中会存储主键值，其他所有的二级索引的叶子节点处都会存储一份记录的主键值，如果主键适用更小的数据类型，也就意味着节省更多的存储空间。

索引字符串值的前缀

我们知道一个字符串其实是由若干个字符组成，如果我们在 MySQL 中适用 utf8 字符集去存储字符串的话，编码一个字符需要占用 1~3 个字节。假设我们的字符串很长，那存储一个字符串就需要占用很大的存储空间。在我们需要为这个字符串列建立索引时，那就意味着在对应的 B+ 树中有这么两个问题：

- B+ 树索引中的记录需要把该列的完整字符串存储起来，而且字符串越长，在索引中占用的存储空间越大。

- 如果 B+ 树索引中索引列存储的字符串很长，那在做字符串比较时会占用更多的时间。

所以索引的设计者提出了个方案 --- 只对字符串的前几个字符进行索引。通过字符串的前几个字符我们已经能大概排序字符串了，剩下不能排序的可以通过遍历进行查找啊，这样只在

`B+` 树中存储字符串的前几个字符的编码，既节约空间，又减少了字符串的比较时间，还大概能解决排序的问题，何乐而不为，比方说我们在建表语句中只对 `name` 列的前10个字符进行索引可以这么写：

```
CREATE TABLE person_info(
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    KEY idx_name_age_birthday (name(10), birthday, phone_number)
);
```

`name(10)` 就表示在建立的 `B+` 树索引中只保留记录的前 `10` 个字符的编码，这种只索引字符串值的前缀的策略是我们非常鼓励的，尤其是在字符串类型能存储的字符比较多的时候。

尽量使用联合索引

如果我们的搜索条件中有多个列的话，最好为这些列建立一个 `联合索引` ，而不是分别为每个列建立一个索引（因为每建一个索引都会维护一棵 `B+` 树），就像我们 `person_info` 表的 `idx_name_age_birthday` 索引，它是 `name` 、 `birthday` 、 `phone_number` 这三个列的联合索引，所以这个联合索引可以用于搜索下边几种列组合：

```
name, birthday, phone_number
name, birthday
name
```

如果我们的确有对其他列搜索的需求，那就为其他列单独再建一个索引吧，虽然创建索引是有代价的，但是还是要保证高效的查询比较重要。

让索引列在比较表达式中单独出现

假设表中有一个整数列 `my_col` ，我们为这个列建立了索引。下边的两个 `WHERE` 子句虽然语义是一致的，但是在效率上却有差别：

1. `WHERE my_col * 2 < 4`
2. `WHERE my_col < 4/2`

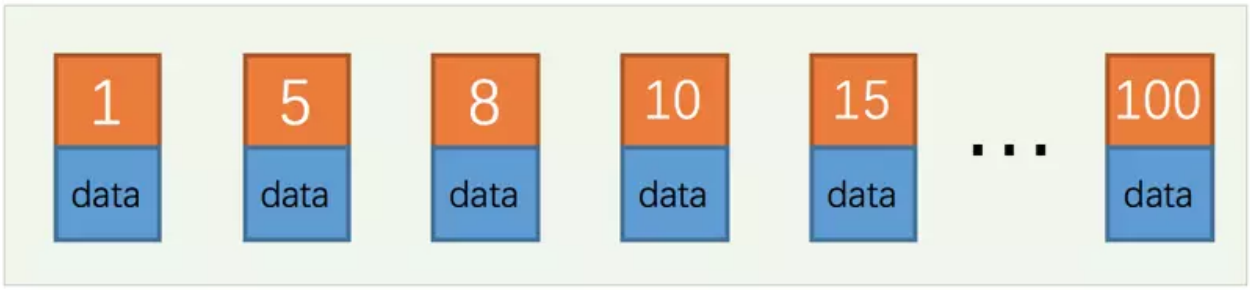
第1个 `WHERE` 子句中 `my_col` 列并不是以单独列的形式出现的，而是以 `my_col * 2` 这样的表达式的形式出现的，存储引擎会依次遍历所有的记录，计算这个表达式的值是不是小于 `4` ，所

以这种情况下是使用不到为 `my_col` 列建立的 `B+` 树索引的。而第2个 `WHERE` 子句中 `my_col` 列并是以单独列的形式出现的，这样的情况可以直接使用 `B+` 树索引。

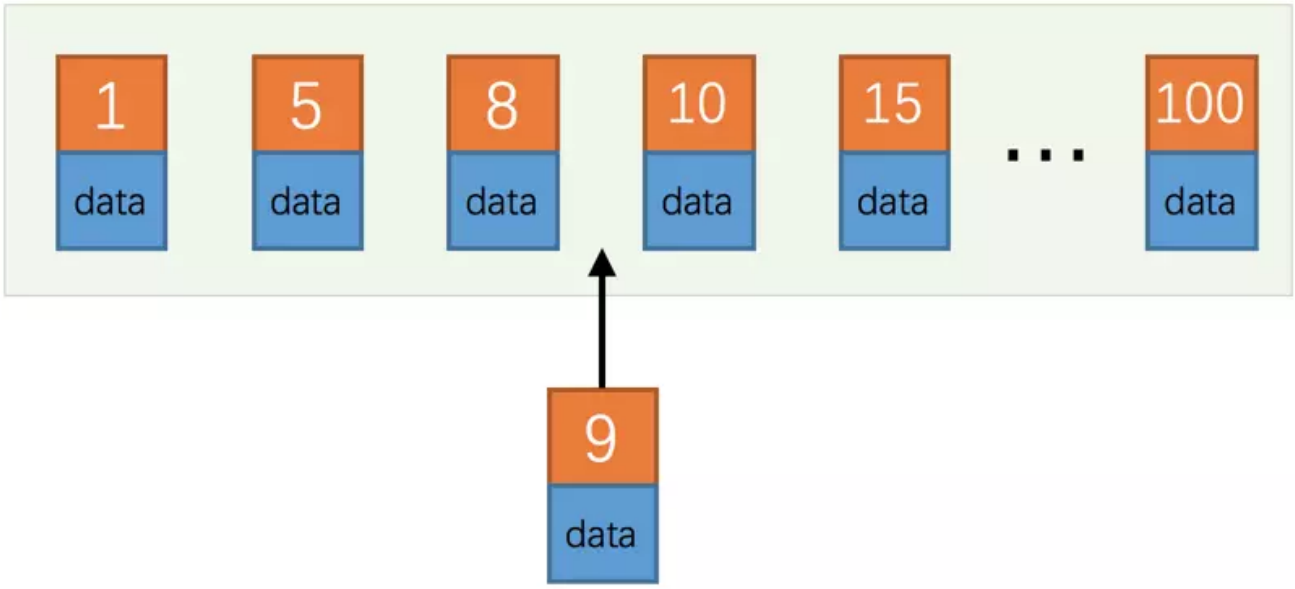
所以结论就是：如果索引列在比较表达式中不是以单独列的形式出现，而是以某个表达式，或者函数调用形式出现的话，是用不到索引的。

主键插入顺序

我们知道，对于一个使用 `InnoDB` 存储引擎的表来说，在我们没有显式的创建索引时，表中的数据实际上都是存储在 `聚簇索引` 的叶子节点的。而记录又是存储在数据页中的，数据页和记录又是按照记录主键值从小到大的顺序进行排序，所以如果我们插入的记录的主键值是依次增大的话，那我们每插满一个数据页就换到下一个数据页继续插，而如果我们插入的主键值忽大忽小的话，这就比较麻烦了，假设某个数据页存储的记录已经满了，它存储的主键值在 `1~100` 之间：



如果此时再插入一条主键值为 `9` 的记录，那它插入的位置就如下图：



可这个数据页已经满了啊，再插进来咋办呢？我们需要把当前页面分裂成两个页面，把本页中的一些记录移动到新创建的这个页中。页面分裂和记录移位意味着什么？意味着：性能损耗！所以如果我们想尽量避免这样无谓的性能损耗，最好让插入的记录的主键值依次递增，这样就不会发生这样的性能损耗了。所以我们建议：让主键具有 `AUTO_INCREMENT` ，让存储引擎自己为表生成主键，而不是我们手动插入 ，比方说我们可以这样定义 `person_info` 表：

```
CREATE TABLE person_info(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    PRIMARY KEY (id),
    KEY idx_name_age_birthday (name(10), birthday, phone_number)
);
```

我们自定义的主键列 `id` 拥有 `AUTO_INCREMENT` 属性，在插入记录时存储引擎会自动为我们填入自增的主键值。

冗余和重复索引

有时候有的同学有意或者无意的就对同一个列创建了多个索引，比方说这样写建表语句：

```
CREATE TABLE person_info(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    PRIMARY KEY (id),
    KEY idx_name_age_birthday (name(10), birthday, phone_number),
    KEY idx_name (name(10))
);
```

我们知道，通过 `idx_name_age_birthday` 索引就可以对 `name` 列进行快速搜索，再创建一个专门针对 `name` 列的索引就算是一个 `冗余` 索引，维护这个索引只会增加维护的成本，并不会对搜索有什么好处。

另一种情况，我们可能会对某个列重复建立索引，比方说这样：

```
CREATE TABLE repeat_index_demo (
    c1 INT PRIMARY KEY,
    c2 INT,
    UNIQUE uidx_c1 (c1),
    INDEX idx_c1 (c1)
);
```

我们看到，`c1`既是主键、又给它定义为一个唯一索引，还给它定义了一个普通索引，可是主键本身就会生成聚簇索引，所以定义的唯一索引和普通索引是重复的，这种情况要避免。

覆盖索引

比方说这个查询：

```
SELECT * FROM person_info WHERE name = 'Ashburn';
```

`person_info`表的`idx_name_age_birthday`的索引列只有3个，而表中一共有4个列，所以为了获得完整的用户记录，在通过`idx_name_age_birthday`索引得到对应的主键值后还得到`聚簇索引`中做一次`回表`操作。`回表`操作也是要性能损耗的啊，所以我们建议：**最好在查询列表里只包含索引列**，比如这样：

```
SELECT name, birthday, phone_number FROM person_info WHERE name = 'Ashburn';
```

因为我们只查询`name`，`birthday`，`phone_number`这三个索引列的值，所以在通过`idx_name_age_birthday`索引得到结果后就不必到`聚簇索引`中再查找记录的剩余列，也就是`country`列的值了。这样就省去了`回表`操作带来的性能损耗，我们把这种只需要用到索引的查询方式称为`索引覆盖`。

当然，如果业务需要查询出索引以外的列，那还是以保证业务需求为重。但是**我们很不鼓励用`*`号作为查询列表，最好把我们需要查询的列依次标明**。

总结

上边只是我们在创建和使用`B+`树索引的过程中需要注意的一些点，后边我们还会陆续介绍更多的优化方法和注意事项，敬请期待。本集内容总结如下：

1. `B+`树索引在空间和时间上都有代价，所以没事儿别瞎建索引。

2. `B+`树索引适用于下边这些情况：

    - 全值匹配
    - 匹配左边的列
    - 匹配范围值

- 精确匹配某一列并范围匹配另外一列
- 用于排序
- 用于分组

3. 在使用索引时需要注意下边这些事项：

- 只为用于搜索、排序或分组的列创建索引
- 为列的基数大的列创建索引
- 索引列的类型尽量小
- 可以只对字符串值的前缀建立索引
- 只有索引列在比较表达式中单独出现才可以适用索引
- 为了尽可能少的让 聚簇索引 发生页面分裂和记录移位的情况，建议让主键拥有 AUTO_INCREMENT 属性。
- 定位并删除表中的重复和冗余索引
- 尽量适用 覆盖索引 进行查询，避免 回表 带来的性能损耗。

题外话

写文章挺累的，有时候你觉得阅读挺流畅的，那其实是背后无数次修改的结果。如果你觉得不错请帮忙转发一下，万分感谢 ~