索引的使用

索引的代价

- 空间代价
  - 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，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，那可是很大的一片存储空间呢。
- 时间代价
  - 每次对表中的数据进行增、删、改操作时，都需要去修改各个`B+`树索引。而且我们讲过，`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。不论是叶子节点中的记录，还是内节点中的记录（也就是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。而增、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。

B+树索引适用条件

- 首先建立一个表：CREATE TABLE person_info(id INT 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_birthday_phone_number (name, birthday, phone_number));
- 对于这个表的注意点
  - 表中的主键是`id`列，它存储一个自动递增的整数。所以`InnoDB`存储引擎会自动为`id`列建立聚簇索引。
  - 我们额外定义了一个二级索引`idx_name_birthday_phone_number`，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用户记录只保留`name`、`birthday`、`phone_number`这三个列的值以及主键`id`的值，并不会保存`country`列的值。
- 一个表中有多少索引就会建立多少棵B+树
- 内节点中存储的是`目录项记录`，叶子节点中存储的是`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_birthday_phone_number`索引对应的`B+`树中页面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，因为只要页面和记录是排好序的，我们就可以通过二分法来快速定位查找。时间复杂度log（N）

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
- 比如这条查找语句`SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday = '1990-09-27' AND phone_number = '15123983239';`跟上面定义的`idx_name_birthday_phone_number`正好一致。
- where子句中搜索条件的顺序对查询结果不会有啥影响，因为MySQL查询优化器会根据索引中的列顺序去决定先使用哪个搜索条件。

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，只包含左边或者多个左边的就行，比方说下边的查询语句：`SELECT * FROM person_info WHERE name = 'Ashburn';`，`SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday = '1990-09-27';`，这样的select语句也会使用到我们的定义的`idx_name_birthday_phone_number`，但`SELECT * FROM person_info WHERE birthday = '1990-09-27';`就不会用到这个 索引，因为`B+`树的数据页和记录先是按照`name`列的值排序的，在`name`列的值相同的情况下才使用`birthday`列进行排序，也就是说`name`列的值不同的记录中`birthday`的值可能是无序的。
- **特别注意**的是**如果我们想使用联合索引中尽可能多的列，搜索条件中的各个列必须是联合索引中从最左边连续的列。**比方说联合索引`idx_name_birthday_phone_number`中列的定义顺序是`name`、`birthday`、`phone_number`，如果我们的搜索条件中只有`name`和`phone_number`，而没有中间的`birthday`，比方说这样：

  ```
  SELECT * FROM person_info WHERE name = 'Ashburn' AND phone_number = '15123983239';
  ```

  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`的值进行排序，`birthday`值相同的记录才按照`phone_number`值进行排序。

匹配列前缀

- 字符串排序的本质就是比较哪个字符串大一点儿，哪个字符串小一点，比较字符串大小就用到了该列的字符集和比较规则，所以对于字符串类型的索引列来说，我们只匹配它的前缀也是可以快速定位记录的，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%';
```

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

- 针对这种查询`SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow';`由于`B+`树中的数据页和记录是先按`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。
- 不过在使用联合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**
  - 例如：`SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow' AND birthday > '1980-01-01';`
  - 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。

    这样子对于联合索引`idx_name_birthday_phone_number`来说，只能用到`name`列的部分，而用不到`birthday`列的部分，因为只有`name`值相同的情况下才能用`birthday`列的值进行排序，而这个查询中通过`name`进行范围查找的记录中可能并不是按照`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这个`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，虽然对多个列进行范围查找时只能用到最左边的那个索引列，但如果左边列是精确查找，则右边的列可以进行范围查找。
- 比如这样的查询`SELECT * FROM person_info WHERE name = 'Ashburn' AND birthday > '1980-01-01' AND birthday < '2000-12-31' AND phone_number > '15100000000';`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。

- 同理，下边的查询也是可能用到这个`idx_name_birthday_phone_number`联合索引的：

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

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通过`ORDER BY`子句按照某种规则进行排序。一般情况下，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、归并排序等等在内存中对这些记录进行排序，有的时候可能查询的结果集太大以至于不能在内存中进行排序的话，还可能暂时借助磁盘的空间来存放中间结果，排序操作完成后再把排好序的结果集返回到客户端。在`MySQL`中，把这种在内存中或者磁盘上进行排序的方式统称为**文件排序**（英文名：`filesort`），跟`文件`这个词儿一沾边儿，就显得这些排序操作非常慢了（磁盘和内存的速度比起来，就像是飞机和蜗牛的对比）。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或文件中排序的步骤，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_birthday_phone_number`索引的示意图，因为这个`B+`树索引本身就是按照上述规则排好序的，所以直接从索引中提取数据，然后进行`回表`操作取出该索引中不包含的列就好了。

使用联合索引进行排序的注意事项

- 对于`联合索引`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。
- 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。当联合索引左边列的值为常量，也可以使用后边的列进行排序，比如这样：

```
SELECT * FROM person_info WHERE name = 'A' ORDER BY birthday, phone_number LIMIT 10;
```

这个查询能使用联合索引进行排序是因为`name`列的值相同的记录是按照`birthday`，`phone_number`排序的

不可以使用索引进行排序的几种情况

- **ASC,DESC混用**，对于使用联合索引进行排序的场景，我们要求各个排序列的排序顺序是一致的，也就是要么各个列都是`ASC`规则排序，要么都是`DESC`规则排序。ORDER BY子句后的列如果不加ASC或者DESC默认是按照ASC排序规则排序的，也就是升序排序的。
  - 如果查询中的各个排序列的排序顺序是一致的，比方说下边这两种情况：
    - `ORDER BY name, birthday LIMIT 10`这种情况直接从索引的最左边开始往右读10行记录就可以了。
    - `ORDER BY name DESC, birthday DESC LIMIT 10`， 这种情况直接从索引的最右边开始往左读10行记录就可以了。
  - 但是如果我们查询的需求是先按照`name`列进行升序排列，再按照`birthday`列进行降序排列的话，比如说这样的查询语句：

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

  这样如果使用索引排序的话过程就是这样的：

  1.先从索引的最左边确定`name`列最小的值，然后找到`name`列等于该值的所有记录，然后从`name`列等于该值的最右边的那条记录开始往左找10条记录。

  2.如果`name`列等于最小的值的记录不足10条，再继续往右找`name`值第二小的记录，重复上边那个过程，直到找到10条记录为止。

  累不累？累！重点是这样不能高效使用索引，而要采取更复杂的算法去从索引中取数据，设计`MySQL`的大叔觉得这样还不如直接文件排序来的快，所以就规定使用联合索引的各个排序列的排序顺序必须是一致的。
- **WHERE子句中出现非排序使用到的索引列**，例如这样的语句：`SELECT * FROM person_info WHERE country = 'China' ORDER BY name LIMIT 10;`
- **派序列包含非同一个索引的列**，有时候用来排序的多个列不是一个索引里的，这种情况也不能使用索引进行排序，比方说：`SELECT * FROM person_info ORDER BY name, country LIMIT 10;`
- **排序列使用了复杂的表达式**，要想使用索引进行排序操作，必须保证索引列是以单独列的形式出现，而不是修饰过的形式，比方说这样：`SELECT * FROM person_info ORDER BY UPPER(name) LIMIT 10;`

用于分组

- 例如这个查询：`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

回表的代价

- 针对这样的查询`SELECT * FROM person_info WHERE name > 'Asa' AND name < 'Barlow';`，在使用`idx_name_birthday_phone_number`索引进行查询时大致可以分为这两个步骤：
  1. 从索引`idx_name_birthday_phone_number`对应的`B+`树中取出`name`值在`Asa`～`Barlow`之间的用户记录。
  2. 由于索引`idx_name_birthday_phone_number`对应的`B+`树用户记录中只包含`name`、`age`、`birthday`、`id`这4个字段，而查询列表是`*`，意味着要查询表中所有字段，也就是还要包括`country`字段。这时需要把从上一步中获取到的每一条记录的`id`字段都到聚簇索引对应的`B+`树中找到完整的用户记录，也就是我们通常所说的`回表`，然后把完整的用户记录返回给查询用户。
- 由于索引`idx_name_birthday_phone_number`对应的`B+`树中的记录首先会按照`name`列的值进行排序，所以值在`Asa`～`Barlow`之间的记录在磁盘中的存储是相连的，集中分布在一个或几个数据页中，我们可以很快的把这些连着的记录从磁盘中读出来，这种读取方式我们也可以称为**顺序I/O**。根据第1步中获取到的记录的`id`字段的值可能并不相连，而在聚簇索引中记录是根据`id`（也就是主键）的顺序排列的，所以根据这些并不连续的`id`值到聚簇索引中访问完整的用户记录可能分布在不同的数据页中，这样读取完整的用户记录可能要访问更多的数据页，这种读取方式我们也可以称为**随机I/O**。一般情况下，顺序I/O比随机I/O的性能高很多，所以步骤1的执行可能很快，而步骤2就慢一些。所以这个使用索引`idx_name_birthday_phone_number`的查询有这么两个特点：
  - 会使用到两个`B+`树索引，一个二级索引，一个聚簇索引。
  - 访问二级索引使用**顺序I/O**，访问聚簇索引使用**随机I/O**。
- 需要回表的记录越多，使用二级索引的性能就越低，什么时候采用全表扫描，什么时候采用二级索引加回表的方式执行查询，查询优化器会事先对表中的记录计算一些统计数据，然后再利用这些统计数据根据查询的条件来计算一下需要回表的记录数，需要回表的记录数越多，就越倾向于使用全表扫描，反之倾向于使用`二级索引 + 回表`的方式。当然优化器做的分析工作不仅仅是这么简单，但是大致上是个这个过程。一般情况下，限制查询获取较少的记录数会让优化器更倾向于选择使

用`二级索引 + 回表`的方式进行查询，因为回表的记录越少，性能提升就越高。

覆盖索引

- 为了彻底告别回表操作带来的性能损耗，最好在查询列表里只包含索引列，我们把这种只需要用到索引的查询方式称为**索引覆盖**。排序操作也优先使用覆盖索引的方式。
- 比如这样：

```
SELECT name, birthday, phone_number FROM person_info WHERE name > 'Asa'
AND name < 'Barlow'
```

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_birthday_phone_number`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`。排序操作也优先使用`覆盖索引`的方式进行查询，比方说这个查询：

```
SELECT name, birthday, phone_number  FROM person_info ORDER BY name,
birthday, phone_number;
```

虽然这个查询中没有`LIMIT`子句，但是采用了`覆盖索引`，所以查询优化器就会直接使用`idx_name_birthday_phone_number`索引进行排序而不需要回表操作了。

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。

如何挑选索引

- **只为用于搜索，排序或分组的列创建索引**
  - 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，无法进行快速查找了～ 而且如果某个建立了二级索引的列的重复值特别多，那么使用这个二级索引查出的记录还可能要做回表操作，这样性能损耗就更大了。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，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。能表示的整数范围当然也是依次递增，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和更高效的`I/O`。
- **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`中使用`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。

    我们前边儿说过索引列的字符串前缀其实也是排好序的，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也就是说在二级索引的记录中只保留字符串前几个字符。这样在查找记录时虽然不能精确的定位到记录的位置，但是能定位到相应前缀所在的位置，然后根据前缀相同的记录的主键值回表查询完整的字符串值，再对比就好了。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_birthday_phone_number (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。

- **索引列字符串值前缀对排序的影响**
  - 如果使用了索引列字符串值前缀，比方说前边只把`name`列的前10个字符放到了二级索引中，下边这个查询可能就有点儿尴尬了：

    ```
    SELECT * FROM person_info ORDER BY name LIMIT 10;
    ```

    因为二级索引中不包含完整的`name`列信息，所以无法对前十个字符相同，后边的字符不同的记录进行排序，也就是使用索引列前缀的方式无法支持使用索引排序，只好乖乖的用文件排序喽。

总结

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。