第二节-B+树索引实战

鲁班学院-周瑜

曾参与大型电商平台、互联网金融产品等多家互联网公司的开发，曾就职于大众点评，任项目经理等职位，参与并主导千万级并发电商网站与系统架构搭建

B+树索引总结

1. 每个索引都对应一棵B+树，B+树分为好多层，最下边一层是叶子节点，其余的是内节点。所有用户记录都存储在B+树的叶子节点，所有目录项记录都存储在内节点。
2. 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。
3. 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用户记录由索引列 + 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。
4. 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，则页面和记录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。
5. 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页目录，所以在这些页面中的查找非常快。

索引的代价

空间上的代价

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，那是很大的一片存储空间。

时间上的代价

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+树索引实战

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 t1 where b = 1 and c = 1 and d = 1;
```

MySQL中有查询优化器，会分析这些搜索条件并且按照可以使用的索引中列的顺序来决定先使用哪个搜索条件。

匹配左边的列

```
select * from t1 where b = 1;
select * from t1 where b = 1 and c = 1;
```

下面这个sql是用不到索引的

```
select * from t1 where c = 1;
```

因为B+树的数据页和记录先是按照b列的值排序的，在b列的值相同的情况下才使用c列进行排序，也就是说b列的值不同的记录中c的值可能是无序的。而现在你跳过b列直接根据c的值去查找，这是做不到的。

匹配列前缀

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 t1 where b like '%101%';
```

MySQL就无法快速定位记录位置了，因为字符串中间有'101'的字符串并没有排好序，所以只能全表扫描了。有时候我们有一些匹配某些字符串后缀的需求，比方说某个表有一个url列，该列中存储了许多url：

```
www.baidu.com
www.google.com
www.qq.com
```

假设已经对该url列创建了索引，如果我们想查询以com为后缀的网址的话可以这样写查询条件：WHERE url LIKE '%com'，但是这样的话无法使用该url列的索引。为了在查询时用到这个索引而不至于全表扫描，我们可以把后缀查询改写成前缀查询，不过我们就得把表中的数据全部逆序存储一下，也就是说我们可以这样保存url列中的数据：

```
moc.udiab.www
moc.elgoog.www
moc.qq.www
```

这样再查找以com为后缀的网址时搜索条件便可以这么写：WHERE url LIKE 'moc%'，这样就可以用到索引了。

匹配范围值

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 t1 where b > 1 and b < 20000;
```

由于B+树中的数据页和记录是先按b列排序的，所以我们上边的查询过程其实是这样的：

- 找到b值为1的记录。
- 找到b值为20000的记录。
- 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
- 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，比方说这样：

```
select * from t1 where b > 1 and c > 1;
```

上边这个查询可以分成两个部分：

1. 通过条件b > 1来对b进行范围，查找的结果可能有多条b值不同的记录，
2. 对这些b值不同的记录继续通过c > 1继续过滤。

这样子对于联合索引来说，只能用到b列的部分，而用不到c列的部分，因为只有b值相同的情况下才能用c列的值进行排序，而这个查询中通过b进行范围查找的记录中可能并不是按照c列进行排序的，所以在搜索条件中继续以c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 t1 where b = 1 and c > 1;
```

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），这些排序操作非常慢。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 t1 order by b, c, d;
```

这个查询的结果集需要先按照b值排序，如果记录的b值相同，则需要按照c来排序，如果c的值相同，则需要按照d排序。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。

分组

```
select b, c, d, count(*) from t1 group by b, c, d;
```

这个查询语句相当于做了3次分组操作：

1. 先把记录按照b值进行分组，所有b值相同的记录划分为一组。
2. 将每个b值相同的分组里的记录再按照c的值进行分组，将title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d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，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。

使用联合索引进行排序或分组的注意事项

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 c, b, d` 的顺序，那也是用不了B+树索引的。

同理， `order by b, order by b, c` 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 t1 where b = 1 order by c, d;
```

这个查询能使用联合索引进行排序是因为b列的值相同的记录是按照c, d排序的。

不可以使用索引进行排序或分组的几种情况

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。

```
select * from t1 order by b ASC, c DESC;
```

这个查询时用不到索引的。

如何建立索引

考虑索引选择性

索引的选择性（Selectivity），是指不重复的索引值（也叫基数，Cardinality）与表记录数的比值：

```
选择性 = 基数 / 记录数
```

选择性的取值范围为(0, 1]，选择性越高的索引价值越大。如果选择性等于1，就代表这个列的不重复值和表记录数是一样的，那么对这个列建立索引是非常合适的，如果选择性非常小，那么就代表这个列的重复值是很多的，不适合建立索引。

考虑前缀索引

用列的前缀代替整个列作为索引key，当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。

> 使用mysql官网提供的示例数据库：https://dev.mysql.com/doc/employee/en/employees-installation.html
> github地址：https://github.com/datacharmer/test_db.git

employees表只有一个索引，那么如果我们想按名字搜索一个人，就只能全表扫描了：

```
EXPLAIN SELECT * FROM employees.employees WHERE first_name='Eric' AND
last_name='Anido';
```

那么可以对或建立索引，看下两个索引的选择性：

```
SELECT count(DISTINCT(first_name))/count(*) AS Selectivity FROM employees.employees; -
- 0.0042
SELECT count(DISTINCT(concat(first_name, last_name)))/count(*) AS Selectivity FROM
employees.employees; -- 0.9313
```

显然选择性太低，选择性很好，但是first_name和last_name加起来长度为30，有没有兼顾长度和选择性的办法？可以考虑用first_name和last_name的前几个字符建立索引，例如，看看其选择性：

```
SELECT count(DISTINCT(concat(first_name, left(last_name, 3))))/count(*) AS Selectivity
FROM employees.employees; -- 0.7879
```

选择性还不错，但离0.9313还是有点距离，那么把last_name前缀加到4：

```
SELECT count(DISTINCT(concat(first_name, left(last_name, 4))))/count(*) AS Selectivity
FROM employees.employees; -- 0.9007
```

这时选择性已经很理想了，而这个索引的长度只有18，比短了接近一半，建立前缀索引的方式为：

```
ALTER TABLE employees.employees ADD INDEX `first_name_last_name4` (first_name,
last_name(4));
```

前缀索引兼顾索引大小和查询速度，但是其缺点是不能用于ORDER BY和GROUP BY操作，也不能用于覆盖索引。

总结

- 索引列的类型尽量小
- 利用索引字符串值的前缀
- 主键自增
- 定位并删除表中的重复和冗余索引
- 尽量使用覆盖索引进行查询，避免回表带来的性能损耗。