快速理解为啥这个查询使用索引，那个查询不使用索引，学会了才发现：真tm简单

原创 小孩子4919 我们都是小青蛙 2019-07-19

小青蛙想做的事儿：用最短的时间学会那些曾经晦涩难懂的东西！

上一篇文章中唠叨了在WHERE子句中出现IS NULL、IS NOT NULL、!=这些条件时仍然可能使用索引：

- MySQL中IS NULL、IS NOT NULL、!=不能用索引？胡扯！

其中强调了一个查询成本的问题，不少同学反映对这个查询成本还是没啥概念，我们今天再来稍微深入的唠叨一下。

B+树结构

我们说对于 InnoDB 存储引擎来说，表中的数据都存储在所谓的B+树中，我们每多建立一个索引，就相当于多建立一棵B+树。

- 对于聚簇索引对应的B+树来说，叶子节点处存储了完整的用户记录（所谓完整用户记录，就是指一条聚簇索引记录中包含所有用户定义的列已经一些内建的列），并且这些聚簇索引记录按照主键值从小到大排序。

- 对于二级索引对应的B+树来说，叶子节点处存储了不完整的用户记录（所谓不完整用户记录，就是指一条二级索引记录只包含索引列和主键），并且这些二级索引记录按照索引列的值从小到大排序。

我们向表中存储了多少条记录，每一棵B+树的叶子节点中就包含多少条记录（注意是“每一棵”，包括聚簇索引对应的B+树以及二级索引对应的B+树）。

示例

我们举个例子：

```
CREATE TABLE t (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    key1 INT,
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1)
) Engine=InnoDB CHARSET=utf8;
```

这个表就包含2个索引（也就是2棵B+树）：

- 以 id 列为主键对应的聚簇索引。
- 为 key1 列建立的二级索引 idx_key1 。

我们向表中插入一些记录：

```
INSERT INTO t VALUES
    (1, 30, 'b'),
    (2, 80, 'b'),
    (3, 23, 'b'),
    (4, NULL, 'b'),
    (5, 11, 'b'),
    (6, 53, 'b'),
    (7, 63, 'b'),
    (8, NULL, 'b'),
    (9, 99, 'b'),
    (10, 12, 'b'),
    (11, 66, 'b'),
    (12, NULL, 'b'),
    (13, 66, 'b'),
    (14, 30, 'b'),
    (15, 11, 'b'),
    (16, 90, 'b');
```

所以现在 s1 表的聚簇索引示意图就是这样：

s1 表的二级索引示意图就是这样：

从图中可以看出，值为 NULL 的二级索引记录都被放到了B+树的最左边，这是因为设计InnoDB的大叔们有规定：

> We define the SQL null to be the smallest possible value of a field.

也就是认为 NULL 值是最小的。

> 小贴士：原谅我们把B+树的结构做了一个如此这般的简化，我们省略了页面的结构，省略了所有的内节点（只画了了三角形替代），省略了记录之间的链表，因为这些不是本文的重点，画成如果所示的样子只是为了突出叶子节点处的记录是按照给定索引的键值进行排序的。

比方说我们现在执行下边这个查询语句：

```
SELECT * FROM t WHERE key1 = 53;
```

那么语句的执行过程就如下图所示：

用文字描述一下这个过程也就是：

- 先通过二级索引 `idx_key1` 对应的 B+ 树快速定位到 `key1` 列值为 `53` 的那条二级索引记录。
- 然后通过二级索引记录上的主键值，也就是 `6` 到执行 回表 操作，也就是到聚簇索引中再找到 `id` 列值为 `6` 的聚簇索引记录。

> 小贴士：B+树叶子节点中的记录都是按照键值按照从小到大的顺序排好序的，通过B+树索引定位到叶子节点中的一条记录是非常快速的。不过由于我们并没有唠叨内节点、页目录这些东西，所以通过B+树索引定位到叶子节点中的一条记录的过程就不详

> 细唠叨了，这些东西其实都在《MySQL是怎样运行的：从根儿上理解MySQL》的掘金小册里详细讲述过。

像下边这个查询：

```
SELECT * FROM t WHERE key1 > 20 AND key1 < 50;
```

它的执行示意图就是这样：

用文字表述就是这样：

- 先通过二级索引 idx_key1 对应的 B+ 树快速定位到满足 key1 > 20 的第一条记录，也就是我们图中所示的 key1 值为 23 的那条记录，然后根据该二级索引中的主键值 3 执行回表操作，得到完整的用户记录后发送到客户端。

- 然后根据上一步骤中获取到的 key1 列值为 23 的二级索引记录的 next_record 属性，找到紧邻着的下一条二级索引记录，也就是 key1 列值为 30 的记录，然后执行回表操作，得到完整用户记录后发送到客户端。

- 然后再找上一步骤中获取到的 key1 列值为 30 的二级索引记录的下一条记录，该记录的 key1 列值也为 30 ，继续执行回表操作将完整的用户记录发送到客户端。

- 然后再找上一步骤中获取到的 key1 列值为 30 的二级索引记录的下一条记录，该记录的 key1 列值为 53 ，不满足 key1 < 50 的条件，所以查询就此终止。

从上边的步骤中也可以看出来：需要扫描的二级索引记录越多，需要执行的回表操作也就越多。如果需要扫描的二级索引记录占全部记录的比例达到某个范围，那优化器就可能选择使用全表扫描的方式执行查询（一个极端的例子就是扫描全部的二级索引记录，那么将对所有的二级索引记录执行回表操作，显然还不如直接全表扫描）。

> 小贴士：我们这里还是定型的分析成本，而不定量分析。定量分析的过程比较复杂，不过小册里有写，有兴趣的同学可以去看。

所以现在的结论就是：判定某个查询是否可以使用索引的条件就是需要扫描的二级索引记录占全部记录的比例是否比较低，较低的话说明成本较低，那就可以使用二级索引来执行查询，否则要采用全表扫描。

具体的查询条件分析

我们分别看一下WHERE子句中出现 IS NULL 、 IS NOT NULL 、 != 这些条件时优化器是怎么做决策的。

IS NULL的情况

比方说这个查询：

```
SELECT * FROM t WHERE key1 IS NULL;
```

优化器在真正执行查询前，会首先少量的访问一下索引，调查一下 key1 在 [NULL, NULL] 这个区间的记录有多少条：

> 小贴士：[NULL, NULL]这个区间代表区间里只有一个NULL值。

优化器经过调查得知，需要扫描的二级索引记录占总记录条数的比例是 3/16 ，它觉得这个查询使用二级索引来执行比较靠谱，所以在执行计划中就显示使用这个 idx_key1 来执行查询：

IS NOT NULL的情况

比方说这个查询：

```
SELECT * FROM t WHERE key1 IS NOT NULL;
```

优化器在真正执行查询前，会首先少量的访问一下索引，调查一下 key1 在 (NULL, +∞) 这个区间内记录有多少条：

> 小贴士：我们这里把NULL当作是最小值对待，你可以认为它比-∞都小。另外注意区间(NULL, +∞)是开区间，也就意味这不包括NULL值。

优化器经过调查得知，需要扫描的二级索引记录占总记录条数的比例是 13/16 ，跟显然这个比例已经非常大了，所以优化器决定使用全表扫描的方式来执行查询：

那怎么才能让使用 IS NOT NULL 条件的查询使用到二级索引呢？这还不简单，让表中符合 IS NOT NULL 条件的记录少不就行了，我们可以执行一下：

```
UPDATE t SET key1 = NULL WHERE key1 < 80;
```

这样再去执行这个查询：

```
SELECT * FROM t WHERE key1 IS NOT NULL;
```

优化器在真正执行查询前，会首先少量的访问一下索引，调查一下 key1 在 (NULL, +∞) 这个区间内记录有多少条：：

优化器经过调查得知，需要扫描的二级索引记录占总记录条数的比例是 3/16 ，它觉得这个查询使用二级索引来执行比较靠谱，所以在执行计划中就显示使用这个 idx_key1 来执行查询：

!= 的情况

比方说这个查询：

```
SELECT * FROM t WHERE key1 != 80;
```

优化器在真正执行查询前，会首先少量的访问一下索引，调查一下 key1 在 (NULL, 80) 和 (80, +∞) 这两个区间内记录有多少条：

优化器经过调查得知，需要扫描的二级索引记录占总记录条数的比例是 2/16 ，它觉得这个查询使用二级索引来执行比较靠谱，所以在执行计划中就显示使用这个 idx_key1 来执行查询：

且慢！为啥执行计划的 rows 列的值为3呢？？？这是个什么鬼，明明只有2条记录符合条件嘛。哈哈，我们罗列一下每个区间找到的符合条件的记录数量：

- (NULL, 80) 区间中有0条记录满足条件 key1 != 80 。
- (80, +∞) 区间中有2条记录满足条件 key1 != 80 。

可是设计优化器的大叔在这里有个规定：当某个范围区间符合给定条件的记录数量为0时，硬生生的把它掰成1。也就是说实际优化器认为在 (NULL, 80) 这个范围区间中有1条记录符合条件 key1 != 80 。所以执行计划的 rows 列才显示了 3 。

> 小贴士：下边是设计优化器的大叔自己对当某个范围区间符合给定条件的记录数量为0时硬生生的把它掰成1的解释（能看懂的就看，看不懂赶紧跳过）：The MySQL optimizer seems to believe an estimate of 0 rows is always accurate and may return the result 'Empty set' based on that. The accuracy is not guaranteed, and even if it were, for a locking read we should anyway perform the search to set the next-key lock. Add 1 to the value to make sure MySQL does not make the assumption!

总结

至此，我们分别分析了拥有 `IS NULL` 、 `IS NOT NULL` 、 `!=` 这三个条件的查询是在什么情况下使用二级索引来执行的，核心结论就是：成本决定执行计划，跟使用什么查询条件并没有什么关系。优化器会首先针对可能使用到的二级索引划分几个范围区间，然后分别调查这些区间内有多少条记录，在这些范围区间内的二级索引记录的总和占总共的记录数量的比例达到某个值时，优化器将放弃使用二级索引执行查询，转而采用全表扫描。

> 小贴士：其实范围区间划分的太多也会影响优化器的决策，比方说IN条件中有太多个参数，将会降低优化器决定使用二级索引执行查询的几率。另外，优化器调查在某个范围区间内的索引记录的条数的方式有两种，一种是所谓的index dive（这种方式在数据少的时候是精确的，在数据多时会有些偏差），一种是依赖index statistics，也就是统计数据来做调查（这种方式的统计是很不精确的，有时候偏差是超级巨大的），反正不论采用哪种方式，优化器都会将各个范围区间中的索引记录数量给计算出来。关于这两种调查方式在小册中都给出了详细的算法，当然都占用了相当大的篇幅，写在公众号文章里就有点杀鸡用牛刀了。

小青蛙历史文章：

- MySQL中IS NULL、IS NOT NULL、!=不能用索引？胡扯！
- 死锁分析
- 收藏版MySQL语句加锁分析
- MySQL小册创作之心路历程

长按关注小青蛙，都是干货喔

注：这篇文章写了4个多钟头（是的，我计了一下时），原创不好写，如果有帮助，一定要记得点个“在看”，帮着转发一下喔，谢谢各位么么哒。