Mysql执行原理之索引合并详解

有道云链接：http://note.youdao.com/noteshare?id=b2dfb5a96ae8a8378ea6f11d07a27e0e&sub=EE7F74C223B1410DA902E1876C2E56A3

我们前边说过MySQL在一般情况下执行一个查询时最多只会用到单个二级索引，但存在有特殊情况，在这些特殊情况下也可能在一个查询中使用到多个二级索引，MySQL中这种使用到多个索引来完成一次查询的执行方法称之为：索引合并/index merge，在前面的成本计算中我们说到过这个概念：“我们需要分别分析单独使用这些索引执行查询的成本，最后还要分析是否可能使用到索引合并”。

其实optimizer trace输出的文本中就有这个片段：

```
"analyzing_roworder_intersect": {
  "usable": false,
  "cause": "too_few_roworder_scans"
}
```

具体的索引合并算法有下边三种。

Intersection合并

Intersection翻译过来的意思是交集。这里是说某个查询可以使用多个二级索引，将从多个二级索引中查询到的结果取交集，比方说下边这个查询：

SELECT * FROM order_exp WHERE order_no = 'a' AND expire_time = 'b';

假设这个查询使用Intersection合并的方式执行的话，那这个过程就是这样的：

从idx_order_no二级索引对应的B+树中取出order_no= 'a'的相关记录。

从idx_insert_time二级索引对应的B+树中取出insert_time= 'b'的相关记录。

二级索引的记录都是由索引列 + 主键构成的，所以我们可以计算出这两个结果集中id值的交集。

按照上一步生成的id值列表进行回表操作，也就是从聚簇索引中把指定id值的完整用户记录取出来，返回给用户。

为啥不直接使用idx_order_no或者idx_insert_time只根据某个搜索条件去读取一个二级索引，然后回表后再过滤另外一个搜索条件呢？这里要分析一下两种查询执行方式之间需要的成本代价。

只读取一个二级索引的成本：

按照某个搜索条件读取一个二级索引，根据从该二级索引得到的主键值进行回表操作，然后再过滤其他的搜索条件

读取多个二级索引之后取交集成本：

按照不同的搜索条件分别读取不同的二级索引，将从多个二级索引得到的主键值取交集，然后进行回表操作。

虽然读取多个二级索引比读取一个二级索引消耗性能，但是大部分情况下读取二级索引的操作是顺序I/O，而回表操作是随机I/O，所以如果只读取一个二级索引时需要回表的记录数特别多，而读取多个二级索引之后取交集的记录数非常少，当节省的因为回表而造成的性能损耗比访问多个二级索引带来的性能损耗更高时，读取多个二级索引后取交集比只读取一个二级索引的成本更低。

MySQL在某些特定的情况下才可能会使用到Intersection索引合并，哪些情况呢？

情况一：等值匹配

二级索引列是等值匹配的情况，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。

而下边这两个查询就不能进行Intersection索引合并：

```
SELECT * FROM order_exp WHERE order_no> 'a' AND insert_time = 'a' AND
order_status = 'b' AND expire_time = 'c';
SELECT * FROM order_exp WHERE order_no = 'a' AND insert_time = 'a';
```

第一个查询是因为对order_no进行了范围匹配，第二个查询是因为联合索引u_idx_day_status中的order_status和expire_time列并没有出现在搜索条件中，所以这两个查询不能进行Intersection索引合并。

情况二：主键列可以是范围匹配

比方说下边这个查询可能用到主键和u_idx_day_status进行Intersection索引合并的操作：

```
SELECT * FROM order_exp WHERE id > 100 AND insert_time = 'a';
```

对于InnoDB的二级索引来说，记录先是按照索引列进行排序，如果该二级索引是一个联合索引，那么会按照联合索引中的各个列依次排序。而二级索引的用户记录是由索引列 + 主键构成的，二级索引列的值相同的记录可能会有好多条，这些索引列的值相同的记录又是按照主键的值进行排序的。

所以重点来了，之所以在二级索引列都是等值匹配的情况下才可能使用Intersection索引合并，是因为只有在这种情况下根据二级索引查询出的结果集是按照主键值排序的。

Intersection索引合并会把从多个二级索引中查询出的主键值求交集，如果从各个二级索引中查询的到的结果集本身就是已经按照主键排好序的，那么求交集的过程就很容易。

假设某个查询使用Intersection索引合并的方式从idx_order_no和idx_expire_time这两个二级索引中获取到的主键值分别是：

从idx_order_no中获取到已经排好序的主键值：1、3、5

从idx_expire_time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)。

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。

另外，不仅是多个二级索引之间可以采用Intersection索引合并，索引合并也可以有聚簇索引参加，也就是我们上边写的情况二：在搜索条件中有主键的范围匹配的情况下也可以使用Intersection索引合并索引合并。为啥主键这就可以范围匹配了？还是得回到应用场景里：

```
SELECT * FROM order_exp WHERE id > 100 AND order_no = 'a';
```

假设这个查询可以采用Intersection索引合并，我们理所当然的以为这个查询会分别按照id > 100这个条件从聚簇索引中获取一些记录，在通过order_no= 'a'这个条件从idx_order_no二级索引中获取一些记录，然后再求交集，其实这样就把问题复杂化了，没必要从聚簇索引中获取一次记录。别忘了二级索引的记录中都带有主键值的，所以可以在从idx_order_no中获取到的主键值上直接运用条件id > 100过滤就行了，这样多简单。所以涉及主键的搜索条件只不过是为了从别的二级索引得到的结果集中过滤记录罢了，是不是等值匹配不重要。

当然，上边说的情况一和情况二只是发生Intersection索引合并的必要条件，不是充分条件。也就是说即使情况一、情况二成立，也不一定发生Intersection索引合并，这得看优化器的心情。优化器只有在单独根据搜索条件从某个二级索引中获取的记录数太多，导致回表开销太大，而通过Intersection索引合并后需要回表的记录数大大减少时才会使用Intersection索引合并。

Union合并

我们在写查询语句时经常想把既符合某个搜索条件的记录取出来，也把符合另外的某个搜索条件的记录取出来，我们说这些不同的搜索条件之间是OR关系。有时候OR关系的不同搜索条件会使用到不同的索引，比方说这样：

```
SELECT * FROM order_exp WHERE order_no = 'a' OR expire_time = 'b'
```

Intersection是交集的意思，这适用于使用不同索引的搜索条件之间使用AND连接起来的情况；Union是并集的意思，适用于使用不同索引的搜索条件之间使用OR连接起来的情况。与Intersection索引合并类似，MySQL在某些特定的情况下才可能会使用到Union索引合并：

情况一：等值匹配

分析同Intersection合并

情况二：主键列可以是范围匹配

分析同Intersection合并

情况三：使用Intersection索引合并的搜索条件

就是搜索条件的某些部分使用Intersection索引合并的方式得到的主键集合和其他方式得到的主键集合取交集，比方说这个查询：

```
SELECT * FROM order_exp WHERE insert_time = 'a' AND order_status = 'b' AND expire_time = 'c' OR (order_no = 'a' AND expire_time = 'b');
```

优化器可能采用这样的方式来执行这个查询：

先按照搜索条件order_no = 'a' AND expire_time = 'b'从索引idx_order_no和idx_expire_time中使用Intersection索引合并的方式得到一个主键集合。

再按照搜索条件 insert_time = 'a' AND order_status = 'b' AND expire_time = 'c'从联合索引u_idx_day_status中得到另一个主键集合。

采用Union索引合并的方式把上述两个主键集合取并集，然后进行回表操作，将结果返回给用户。

当然，查询条件符合了这些情况也不一定就会采用Union索引合并，也得看优化器的心情。优化器只有在单独根据搜索条件从某个二级索引中获取的记录数比较少，通过Union索引合并后进行访问的代价比全表扫描更小时才会使用Union索引合并。

Sort-Union合并

Union索引合并的使用条件太苛刻，必须保证各个二级索引列在进行等值匹配的条件下才可能被用到，比方说下边这个查询就无法使用到Union索引合并：

```
SELECT * FROM order_exp WHERE order_no< 'a' OR expire_time> 'z'
```

这是因为根据order_no< 'a'从idx_order_no索引中获取的二级索引记录的主键值不是排好序的，根据expire_time> 'z'从idx_expire_time索引中获取的二级索引记录的主键值也不是排好序的，但是order_no< 'a'和expire_time> 'z''这两个条件又特别让我们动心，所以我们可以这样：

先根据order_no< 'a'条件从idx_order_no二级索引中获取记录，并按照记录的主键值进行排序

再根据expire_time> 'z'条件从idx_expire_time二级索引中获取记录，并按照记录的主键值进行排序

因为上述的两个二级索引主键值都是排好序的，剩下的操作和Union索引合并方式就一样了。

上述这种先按照二级索引记录的主键值进行排序，之后按照Union索引合并方式执行的方式称之为Sort-Union索引合并，很显然，这种Sort-Union索引合并比单纯的Union索引合

并多了一步对二级索引记录的主键值排序的过程。

联合索引替代Intersection索引合并

SELECT * FROM order_exp WHERE order_no= 'a' And expire_time= 'z';

这个查询之所以可能使用Intersection索引合并的方式执行，还不是因为idx_order_no和idx_expire_time是两个单独的B+树索引，要是把这两个列搞一个联合索引，那直接使用这个联合索引就把事情搞定了，何必用啥索引合并呢，就像这样：

ALTER TABLE order_exp drop index idx_order_no, idx_expire_time,

add index idx_order_no_expire_time(order_no, expire_time);

这样我们把idx_order_no, idx_expire_time都干掉，再添加一个联合索引idx_order_no_expire_time，使用这个联合索引进行查询简直是又快又好，既不用多读一棵B+树，也不用合并结果。