Mysql内核查询优化规则详解

有道云链接：http://note.youdao.com/noteshare?id=fbd0885f891bbcc6033eaeab1b837fe0&sub=674C8D7C45774906B2EFE35A23B96157

在前面的课程中我们看到当我们把SQL语句交给MySQL执行后，MySQL在执行的过程中会有很多的优化措施，比如索引条件下推，回表中的MRR、索引合并等等。但是在执行之前，MySQL会依据一些规则，竭尽全力的把我们的SQL语句进行转换，以期可以达到成某种可以更高效执行的形式，这个过程也可以被称作查询重写或者查询优化，很多时候这些优化可以由我们自行完成以减少MySQL的优化时间。

条件化简

我们编写的查询语句的搜索条件本质上是一个表达式，这些表达式可能比较繁杂，或者不能高效的执行，MySQL的查询优化器会为我们简化这些表达式。

移除不必要的括号

有时候表达式里有许多无用的括号，比如这样：

```
((a = 5 AND b = c) OR ((a > c) AND (c < 5)))
```

优化器会把那些用不到的括号给干掉，就是这样：

```
(a = 5 and b = c) OR (a > c AND c < 5)
```

常量传递（constant_propagation）

有时候某个表达式是某个列和某个常量做等值匹配，比如这样：

```
a = 5
```

当这个表达式和其他涉及列a的表达式使用AND连接起来时，可以将其他表达式中的a的值替换为5，比如这样：

```
a = 5 AND b > a
```

就可以被转换为：

```
a = 5 AND b > 5
```

等值传递（equality_propagation）

有时候多个列之间存在等值匹配的关系，比如这样：

```
a = b and b = c and c = 5
```

这个表达式可以被简化为：

```
a = 5 and b = 5 and c = 5
```

移除没用的条件（trivial_condition_removal）

对于一些明显永远为TRUE或者FALSE的表达式，优化器会移除掉它们，比如这个表达式：

```
(a < 1 and b = b) OR (a = 6 OR 5 != 5)
```

很明显，b = b这个表达式永远为TRUE，5 != 5这个表达式永远为FALSE，所以简化后的表达式就是这样的：

(a < 1 and TRUE) OR (a = 6 OR FALSE)

可以继续被简化为

a < 1 OR a = 6

表达式计算

在查询开始执行之前，如果表达式中只包含常量的话，它的值会被先计算出来，比如这个：

a = 5 + 1

因为5 + 1这个表达式只包含常量，所以就会被化简成：

a = 6

但是这里需要注意的是，如果某个列并不是以单独的形式作为表达式的操作数时，比如出现在函数中，出现在某个更复杂表达式中，就像这样：

ABS(a) > 5

或者：

-a < -8

优化器是不会尝试对这些表达式进行化简的。我们前边说过只有搜索条件中索引列和常数使用某些运算符连接起来才可能使用到索引，所以如果可以的话，最好让索引列以单独的形式出现在表达式中。

常量表检测

MySQL觉得下边这种查询运行的特别快：

使用主键等值匹配或者唯一二级索引列等值匹配作为搜索条件来查询某个表。

MySQL觉得这两种查询花费的时间特别少，少到可以忽略，所以也把通过这两种方式查询的表称之为常量表（英文名：constant tables）。优化器在分析一个查询语句时，先首先执行常量表查询，然后把查询中涉及到该表的条件全部替换成常数，最后再分析其余表的查询成本，比方说这个查询语句：

```
SELECT * FROM table1 INNER JOIN table2
    ON table1.column1 = table2.column2
    WHERE table1.primary_key = 1;
```

很明显，这个查询可以使用主键和常量值的等值匹配来查询table1表，也就是在这个查询中table1表相当于常量表，在分析对table2表的查询成本之前，就会执行对table1表的查询，并把查询中涉及table1表的条件都替换掉，也就是上边的语句会被转换成这样：

SELECT table1表记录的各个字段的常量值, table2.* FROM table1 INNER JOIN table2 ON table1表column1列的常量值 = table2.column2;

外连接消除

我们知道，内连接的驱动表和被驱动表的位置可以相互转换，而左（外）连接和右（外）连接的驱动表和被驱动表是固定的。这就导致内连接可能通过优化表的连接顺序来降低整体的查询成本，而外连接却无法优化表的连接顺序。

我们之前说过，外连接和内连接的本质区别就是：对于外连接的驱动表的记录来说，如果无法在被驱动表中找到匹配ON子句中的过滤条件的记录，那么该记录仍然会被加入到结果集中，对应的被驱动表记录的各个字段使用NULL值填充；而内连接的驱动表的记录如果无法在被驱动表中找到匹配ON子句中的过滤条件的记录，那么该记录会被舍弃。查询效果就是这样：

SELECT * FROM e1 INNER JOIN e2 ON e1.m1 = e2.m2;

```
mysql> SELECT * FROM e1 INNER JOIN e2 ON e1.m1 = e2.m2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
+------+------+------+------+
2 rows in set (0.00 sec)
```

SELECT * FROM e1 LEFT JOIN e2 ON e1.m1 = e2.m2;

```
mysql> SELECT * FROM e1 LEFT JOIN e2 ON e1.m1 = e2.m2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
|    1 | a    | NULL | NULL |
+------+------+------+------+
3 rows in set (0.00 sec)
```

对于上边例子中的（左）外连接来说，由于驱动表e1中m1=1, n1='a'的记录无法在被驱动表e2中找到符合ON子句条件e1.m1 = e2.m2的记录，所以就直接把这条记录加入到结果集，对应的e2表的m2和n2列的值都设置为NULL。

因为凡是不符合WHERE子句中条件的记录都不会参与连接。只要我们在搜索条件中指定关于被驱动表相关列的值不为NULL，那么外连接中在被驱动表中找不到符合ON子句条件的驱动表记录也就被排除出最后的结果集了，也就是说：在这种情况下：外连接和内连接也就没有什么区别了！比方说这个查询：

SELECT * FROM e1 LEFT JOIN e2 ON e1.m1 = e2.m2 WHERE e2.n2 IS NOT NULL;

```
mysql> SELECT * FROM e1 LEFT JOIN e2 ON e1.m1 = e2.m2 WHERE e2.n2 IS NOT NULL;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
+------+------+------+------+
2 rows in set (0.00 sec)
```

由于指定了被驱动表e2的n2列不允许为NULL，所以上边的e1和e2表的左（外）连接查询和内连接查询是一样的。当然，我们也可以不用显式的指定被驱动表的某个列IS NOT NULL，只要隐含的有这个意思就行了，比方说这样：

```
SELECT * FROM e1 LEFT JOIN e2 ON e1.m1 = e2.m2 WHERE e2.m2 = 2;
```

在这个例子中，我们在WHERE子句中指定了被驱动表e2的m2列等于2，也就相当于间接的指定了m2列不为NULL值，所以上边的这个左（外）连接查询其实和下边这个内连接查询是等价的：

```
SELECT * FROM e1 INNER JOIN e2 ON e1.m1 = e2.m2 WHERE e2.m2 = 2;
```

我们把这种在外连接查询中，指定的WHERE子句中包含被驱动表中的列不为NULL值的条件称之为空值拒绝（英文名：reject-NULL）。在被驱动表的WHERE子句符合空值拒绝的条件后，外连接和内连接可以相互转换。这种转换带来的好处就是查询优化器可以通过评估表的不同连接顺序的成本，选出成本最低的那种连接顺序来执行查询。

子查询MySQL内部优化规则

子查询语法

在一个查询语句A里的某个位置也可以有另一个查询语句B，这个出现在A语句的某个位置中的查询B就被称为子查询，A也被称之为外层查询。子查询可以在一个外层查询的各种位置出现，比如：

SELECT子句中

也就是我们平时说的查询列表中，比如这样：

```
SELECT (SELECT m1 FROM e1 LIMIT 1);
```

其中的(SELECT m1 FROM e1 LIMIT 1)就是子查询。

FROM子句中

比如：

```
SELECT m, n FROM (SELECT m2 + 1 AS m, n2 AS n FROM e2 WHERE m2 > 2) AS t;
```

这个例子中的子查询是：(SELECT m2 + 1 AS m, n2 AS n FROM e2 WHERE m2 > 2)，这里可以把子查询的查询结果当作是一个表，子查询后边的AS t表明这个子查询的结果就相当于一个名称为t的表，这个名叫t的表的列就是子查询结果中的列，比如例子中表t就有两个列：m列和n列。这个放在FROM子句中的子查询本质上相当于一个表，但又和我们平常使用的表有点儿不一样，MySQL把这种由子查询结果集组成的表称之为**派生表**。

WHERE或ON子句中

把子查询放在外层查询的WHERE子句或者ON子句中可能是我们最常用的一种使用子查询的方式了，比如这样：

```
SELECT * FROM e1 WHERE m1 IN (SELECT m2 FROM e2);
```

这个查询表明我们想要将(SELECT m2 FROM e2)这个子查询的结果作为外层查询的IN语句参数，整个查询语句的意思就是我们想找e1表中的某些记录，这些记录的m1列的值能在e2表的m2列找到匹配的值。

ORDER BY子句、GROUP BY子句中

虽然语法支持，但没啥意义。

按返回的结果集区分子查询

因为子查询本身也算是一个查询，所以可以按照它们返回的不同结果集类型而把这些子查询分为不同的类型：

标量子查询

那些只返回一个单一值的子查询称之为标量子查询，比如这样：

```
SELECT (SELECT m1 FROM e1 LIMIT 1);
```

或者这样：

```
SELECT * FROM e1 WHERE m1 = (SELECT MIN(m2) FROM e2);
SELECT * FROM e1 WHERE m1 < (SELECT MIN(m2) FROM e2);
```

这两个查询语句中的子查询都返回一个单一的值，也就是一个标量。这些标量子查询可以作为一个单一值或者表达式的一部分出现在查询语句的各个地方。

行子查询

顾名思义，就是返回一条记录的子查询，不过这条记录需要包含多个列（只包含一个列就成了标量子查询了）。比如这样：

```
SELECT * FROM e1 WHERE (m1, n1) = (SELECT m2, n2 FROM e2 LIMIT 1);
```

其中的(SELECT m2, n2 FROM e2 LIMIT 1)就是一个行子查询，整条语句的含义就是要从e1表中找一些记录，这些记录的m1和n1列分别等于子查询结果中的m2和n2列。

列子查询

列子查询自然就是查询出一个列的数据，不过这个列的数据需要包含多条记录（只包含一条记录就成了标量子查询了）。比如这样：

```
SELECT * FROM e1 WHERE m1 IN (SELECT m2 FROM e2);
```

其中的(SELECT m2 FROM e2)就是一个列子查询，表明查询出e2表的m2列的值作为外层查询IN语句的参数。

表子查询

顾名思义，就是子查询的结果既包含很多条记录，又包含很多个列，比如这样：

```
SELECT * FROM e1 WHERE (m1, n1) IN (SELECT m2, n2 FROM e2);
```

其中的(SELECT m2, n2 FROM e2)就是一个表子查询，这里需要和行子查询对比一下，行子查询中我们用了LIMIT 1来保证子查询的结果只有一条记录，表子查询中不需要这个限制。

按与外层查询关系来区分子查询

不相关子查询

如果子查询可以单独运行出结果，而不依赖于外层查询的值，我们就可以把这个子查询称之为不相关子查询。我们前边介绍的那些子查询全部都可以看作不相关子查询。

相关子查询

如果子查询的执行需要依赖于外层查询的值，我们就可以把这个子查询称之为相关子查询。比如：

```
SELECT * FROM e1 WHERE m1 IN (SELECT m2 FROM e2 WHERE n1 = n2);
```

例子中的子查询是(SELECT m2 FROM e2 WHERE n1 = n2)，可是这个查询中有一个搜索条件是n1 = n2，别忘了n1是表e1的列，也就是外层查询的列，也就是说子查询的执行需要依赖于外层查询的值，所以这个子查询就是一个相关子查询。

[NOT] IN/ANY/SOME/ALL子查询

对于列子查询和表子查询来说，它们的结果集中包含很多条记录，这些记录相当于是一个集合，所以就不能单纯的和另外一个操作数使用操作符来组成布尔表达式了，MySQL通过下面的语法来支持某个操作数和一个集合组成一个布尔表达式：

IN或者NOT IN

具体的语法形式如下：

```
操作数 [NOT] IN (子查询)
```

这个布尔表达式的意思是用来判断某个操作数在不在由子查询结果集组成的集合中，比如下边的查询的意思是找出e1表中的某些记录，这些记录存在于子查询的结果集中：

```
SELECT * FROM e1 WHERE (m1, n1) IN (SELECT m2, n2 FROM e2);
```

ANY/SOME（ANY和SOME是同义词）

具体的语法形式如下：

```
操作数 比较符 ANY/SOME(子查询)
```

这个布尔表达式的意思是只要子查询结果集中存在某个值和给定的操作数做比较操作，比较结果为TRUE，那么整个表达式的结果就为TRUE，否则整个表达式的结果就为FALSE。比方说下边这个查询：

```
SELECT * FROM e1 WHERE m1 > ANY(SELECT m2 FROM e2);
```

这个查询的意思就是对于e1表的某条记录的m1列的值来说，如果子查询(SELECT m2 FROM e2)的结果集中存在一个小于m1列的值，那么整个布尔表达式的值就是TRUE，否则为FALSE，也就是说只要m1列的值大于子查询结果集中最小的值，整个表达式的结果就是TRUE，所以上边的查询本质上等价于这个查询：

```
SELECT * FROM e1 WHERE m1 > (SELECT MIN(m2) FROM e2);
```

另外，=ANY相当于判断子查询结果集中是否存在某个值和给定的操作数相等，它的含义和IN是相同的。

ALL

具体的语法形式如下：

操作数 比较操作 ALL(子查询)

这个布尔表达式的意思是子查询结果集中所有的值和给定的操作数做比较操作比较结果为TRUE，那么整个表达式的结果就为TRUE，否则整个表达式的结果就为FALSE。比方说下边这个查询：

```
SELECT * FROM e1 WHERE m1 > ALL(SELECT m2 FROM e2);
```

这个查询的意思就是对于e1表的某条记录的m1列的值来说，如果子查询(SELECT m2 FROM e2)的结果集中的所有值都小于m1列的值，那么整个布尔表达式的值就是TRUE，否则为FALSE，也就是说只要m1列的值大于子查询结果集中最大的值，整个表达式的结果就是TRUE，所以上边的查询本质上等价于这个查询：

```
SELECT * FROM e1 WHERE m1 > (SELECT MAX(m2) FROM e2);
```

EXISTS子查询

有的时候我们仅仅需要判断子查询的结果集中是否有记录，而不在乎它的记录具体是个啥，可以使用把EXISTS或者NOT EXISTS放在子查询语句前边，就像这样：

```
SELECT * FROM e1 WHERE EXISTS (SELECT 1 FROM e2);
```

对于子查询(SELECT 1 FROM e2)来说，我们并不关心这个子查询最后到底查询出的结果是什么，所以查询列表里填*、某个列名，或者其他啥东西都无所谓，我们真正关心的是子查询的结果集中是否存在记录。也就是说只要(SELECT 1 FROM e2)这个查询中有记录，那么整个EXISTS表达式的结果就为TRUE。

子查询语法注意事项

子查询必须用小括号扩起来。

在SELECT子句中的子查询必须是标量子查询，如果子查询结果集中有多个列或者多个行，都不允许放在SELECT子句中，在想要得到标量子查询或者行子查询，但又不能保证子查询的结果集只有一条记录时，应该使用LIMIT 1语句来限制记录数量。

对于[NOT] IN/ANY/SOME/ALL子查询来说，子查询中不允许有LIMIT语句，而且这类子查询中ORDER BY子句、DISTINCT语句、没有聚集函数以及HAVING子句的GROUP BY子句没有什么意义。因为子查询的结果其实就相当于一个集合，集合里的值排不排序等一点儿都不重要。

不允许在一条语句中增删改某个表的记录时同时还对该表进行子查询。

子查询在MySQL中是怎么执行的

想象子查询的执行方式

想象中子查询的执行方式是这样的：

如果该子查询是不相关子查询，比如下边这个查询：

```
SELECT * FROM s1 WHERE order_note IN (SELECT order_note FROM s2);
```

先单独执行(SELECT order_note FROM s2)这个子查询。然后在将上一步子查询得到的结果当作外层查询的参数再执行外层查询SELECT * FROM s1 WHERE order_note IN (...)。

如果该子查询是相关子查询，比如下边这个查询：

```
SELECT * FROM s1 WHERE order_note IN (SELECT order_note FROM s2 WHERE
s1.order_no= s2.order_no);
```

这个查询中的子查询中出现了s1.order_no= s2.order_no这样的条件，意味着该子查询的执行依赖着外层查询的值，先从外层查询中获取一条记录，本例中也就是先从s1表中获取一条记录，然后执行子查询。

最后根据子查询的查询结果来检测外层查询WHERE子句的条件是否成立，如果成立，就把外层查询的那条记录加入到结果集，否则就丢弃。

再次执行第一步，获取第二条外层查询中的记录，依次类推。

但真的是这样吗？其实MySQL用了一系列的办法来优化子查询的执行，下边我们来看看各种不同类型的子查询具体是怎么执行的。

标量子查询、行子查询的执行方式

对于**不相关标量子查询或者行子查询**来说，它们的执行方式很简单，比方说下边这个查询语句：

```
SELECT * FROM s1 WHERE order_note = (SELECT order_note FROM s2 WHERE key3 =
'a' LIMIT 1);
```

它的执行方式和我们前面想象的一样：先单独执行(SELECT order_note FROM s2 WHERE key3 = 'a' LIMIT 1)这个子查询。然后在将上一步子查询得到的结果当作外层查询的参数再执行外层查询SELECT * FROM s1 WHERE order_note= ...。

也就是说，对于包含不相关的标量子查询或者行子查询的查询语句来说，MySQL会分别独立的执行外层查询和子查询，就当作两个单表查询就好了。

对于**相关的标量子查询或者行子查询**来说，比如下边这个查询：

```
SELECT * FROM s1 WHERE
    order_note = (SELECT order_note FROM s2 WHERE s1.order_no= s2.order_no
LIMIT 1);
```

事情也和我们前面想象的一样，它的执行方式就是这样的：

先从外层查询中获取一条记录，本例中也就是先从s1表中获取一条记录。

然后从上一步骤中获取的那条记录中找出子查询中涉及到的值，本例中就是从s1表中获取的那条记录中找出s1.order_no列的值，然后执行子查询。

最后根据子查询的查询结果来检测外层查询WHERE子句的条件是否成立，如果成立，就把外层查询的那条记录加入到结果集，否则就丢弃。

再次执行第一步，获取第二条外层查询中的记录，依次类推。

也就是说对于两种使用标量子查询以及行子查询的场景中，MySQL优化器的执行方式并没有什么新鲜的。

MySQL对IN子查询的优化

物化表

对于不相关的IN子查询，比如这样：

```
SELECT * FROM s1 WHERE order_note IN (SELECT order_note FROM s2 WHERE order_no = 'a');
```

我们最开始的感觉就是这种不相关的IN子查询和不相关的标量子查询或者行子查询是一样一样的，都是把外层查询和子查询当作两个独立的单表查询来对待。但是MySQL为了优化IN子查询下了很大力气，所以整个执行过程并不像我们想象的那么简单。

对于不相关的IN子查询来说，如果子查询的结果集中的记录条数很少，那么把子查询和外层查询分别看成两个单独的单表查询效率很高，但是如果单独执行子查询后的结果集太多的话，就会导致这些问题：

1、结果集太多，可能内存中都放不下。

2、对于外层查询来说，如果子查询的结果集太多，那就意味着IN子句中的参数特别多，这就导致：无法有效的使用索引，只能对外层查询进行全表扫描。

在对外层查询执行全表扫描时，由于IN子句中的参数太多，这会导致检测一条记录是否符合和IN子句中的参数匹配花费的时间太长。

比如说IN子句中的参数只有两个：

```
SELECT * FROM tbl_name WHERE column IN (a, b);
```

这样相当于需要对tbl_name表中的每条记录判断一下它的column列是否符合column = a OR column = b。在IN子句中的参数比较少时这并不是什么问题，如果IN子句中的参数比较多时，比如这样：

```
SELECT * FROM tbl_name WHERE column IN (a, b, c ..., ...);
```

那么这样每条记录需要判断一下它的column列是否符合column = a OR column = b OR column = c OR ...，这样性能耗费可就多了。

MySQL的改进是不直接将不相关子查询的结果集当作外层查询的参数，而是将该结果集写入一个临时表里。写入临时表的过程是这样的：

1、该临时表的列就是子查询结果集中的列。

2、写入临时表的记录会被去重，临时表也是个表，只要为表中记录的所有列建立主键或者唯一索引。

一般情况下子查询结果集不会大的离谱，所以会为它建立基于内存的使用Memory存储引擎的临时表，而且会为该表建立哈希索引。

如果子查询的结果集非常大，超过了系统变量tmp_table_size或者max_heap_table_size，临时表会转而使用基于磁盘的存储引擎来保存结果集中的记录，索引类型也对应转变为B+树索引。

MySQL把这个将子查询结果集中的记录保存到临时表的过程称之为**物化**（英文名：Materialize）。为了方便起见，我们就把那个存储子查询结果集的临时表称之为物化表。正因为物化表中的记录都建立了索引（基于内存的物化表有哈希索引，基于磁盘的有B+树索引），通过索引执行IN语句判断某个操作数在不在子查询结果集中变得非常快，从而提升了子查询语句的性能。

物化表转连接

事情到这就完了？我们还得重新审视一下最开始的那个查询语句：

```
SELECT * FROM s1 WHERE order_note IN (SELECT order_note FROM s2 WHERE order_no = 'a');
```

当我们把子查询进行物化之后，假设子查询物化表的名称为materialized_table，该物化表存储的子查询结果集的列为m_val，那么这个查询

就相当于表s1和子查询物化表materialized_table进行内连接：

```
SELECT s1.* FROM s1 INNER JOIN materialized_table ON order_note = m_val;
```

转化成内连接之后就有意思了，查询优化器可以评估不同连接顺序需要的成本是多少，选取成本最低的那种查询方式执行查询。我们分析一下上述查询中使用外层查询的表s1和物化表materialized_table进行内连接的成本都是由哪几部分组成的：

1、如果使用s1表作为驱动表的话，总查询成本由下边几个部分组成：

物化子查询时需要的成本

扫描s1表时的成本

s1表中的记录数量 × 通过m_val = xxx对materialized_table表进行单表访问的成本（我们前边说过物化表中的记录是不重复的，并且为物化表中的列建立了索引，所以这个步骤显然是非常快的）。

2、如果使用materialized_table表作为驱动表的话，总查询成本由下边几个部分组成：

物化子查询时需要的成本

扫描物化表时的成本

物化表中的记录数量 × 通过order_note= xxx对s1表进行单表访问的成本（如果order_note列上建立了索引，这个步骤还是非常快的）。

MySQL查询优化器会通过运算来选择上述成本更低的方案来执行查询。

将子查询转换为semi-join

将子查询进行物化之后再执行查询都会有建立临时表的成本，能不能不进行物化操作直接把子查询转换为连接呢？让我们重新审视一下上边的查询语句：

```
SELECT * FROM s1 WHERE order_note IN (SELECT order_note FROM s2 WHERE order_no = 'a');
```

我们可以把这个查询理解成：对于s1表中的某条记录，如果我们能在s2表（准确的说是执行完WHERE s2.order_no= 'a'之后的结果集）中找到一条或多条记录，这些记录的order_note的值等于s1表记录的order_note列的值，那么该条s1表的记录就会被加入到最终的结果集。这个过程其实和把s1和s2两个表连接起来的效果很像：

```
SELECT s1.* FROM s1 INNER JOIN s2
    ON s1.order_note = s2.order_note
    WHERE s2.order_no= 'a';
```

只不过我们不能保证对于s1表的某条记录来说，在s2表（准确的说是执行完WHERE s2.order_no= 'a'之后的结果集）中有多少条记录满足s1.order_no = s2.order_no这个条件，不过我们可以分三种情况讨论：

情况一：对于s1表的某条记录来说，s2表中没有任何记录满足s1.order_note = s2.order_note 这个条件，那么该记录自然也不会加入到最后的结果集。

情况二：对于s1表的某条记录来说，s2表中有且只有1条记录满足s1.order_note = s2.order_note 这个条件，那么该记录会被加入最终的结果集。

情况三：对于s1表的某条记录来说，s2表中至少有2条记录满足s1.order_note = s2.order_note 这个条件，那么该记录会被多次加入最终的结果集。

对于s1表的某条记录来说，由于我们只关心s2表中是否存在记录满足s1.order_no = s2.order_note这个条件，而不关心具体有多少条记录与之匹配，又因为有情况三的存在，我们上边所说的IN子查询和两表连接之间并不完全等价。但是将子查询转换为连接又真的可以充分发挥优化器的作用，所以MySQL在这里提出了一个新概念 --- **半连接**（英文名：semi-join）。

将s1表和s2表进行半连接的意思就是：对于s1表的某条记录来说，我们只关心在s2表中是否存在与之匹配的记录，而不关心具体有多少条记录与之匹配，最终的结果集中只保留s1表的记录。为了让大家有更直观的感受，我们假设MySQL内部是这么改写上边的子查询的：

```
SELECT s1.* FROM s1 SEMI JOIN s2
    ON s1.order_note = s2.order_note
    WHERE order_no= 'a';
```

注意： semi-join只是在MySQL内部采用的一种执行子查询的方式，MySQL并没有提供面向用户的semi-join语法。

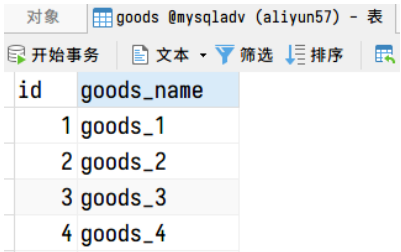
semi-join实例

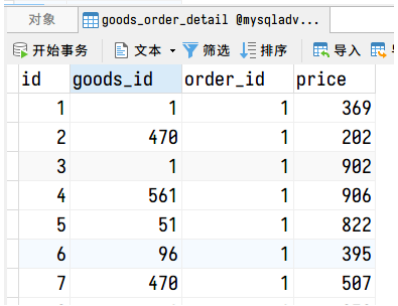
现在有三个表：商品表goods、订单表goods_order、订单详情表goods_order_detail。三个表中的数据有两种方式导入：

1、执行goods_stru.sql、goods_order_stru.sql、goods_order_detail_stru.sql建立原始表，然后执行存储过程create_goods产生原始数据，再执行存储过程randon_detail进行数据更新；

2、执行SQL脚本goods_all.sql、 goods_order_all.sql、goods_order_detail_all.sql，同时导入表结构和数据。

三个表目前都没有索引：





三个表目前的记录数如下：

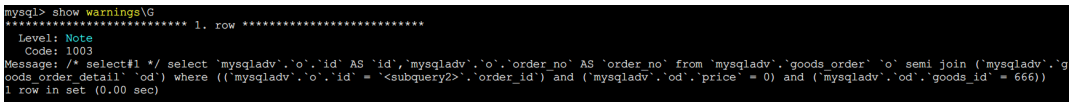
```
mysql> select count(*) from goods;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from goods_order;
+----------+
| count(*) |
+----------+
|   998863 |
+----------+
1 row in set (0.38 sec)

mysql> select count(*) from goods_order_detail;
+----------+
| count(*) |
+----------+
|  9988621 |
+----------+
1 row in set (2.76 sec)
```

现在有这么个业务场景：查找包含赠送商品且赠送商品编号是666的订单

```
select o.* from goods_order o where o.id in(select order_id from goods_order_detail od where od.goods_id=666 and od.price=0);
```

我们查看下MySQL对这个查询语句的改写：

```
mysql> show warnings;
/* select#1 */ select `mysqladv`.`o`.`id` AS `id`,`mysqladv`.`o`.`order_no` AS `order_no` from `mysqladv`.`goods_order` `o` semi join (`mysqladv`.`goods_order_detail` `od`) where ((`mysqladv`.`o`.`id` = `<subquery2>`.`order_id`) and (`mysqladv`.`od`.`price` = 0) and (`mysqladv`.`od`.`goods_id` = 666))
```



可以看到MySQL将这个子查询改造为了半连接semi join。

半连接的实现方法概述

怎么实现这种所谓的半连接呢？MySQL准备了好几种办法，比如Table pullout （子查询中的表上拉）、DuplicateWeedout execution strategy （重复值消除）、LooseScan execution strategy （松散扫描）、Semi-join Materializationa半连接物化、FirstMatch execution strategy （首次匹配）等等。

Table pullout （子查询中的表上拉）

当子查询的查询列表处只有主键或者唯一索引列时，可以直接把子查询中的表上拉到外层查询的FROM子句中，并把子查询中的搜索条件合并到外层查询的搜索条件中，比如假设s2中存在这个一个key2列，列上有唯一性索引：

```
SELECT * FROM s1
    WHERE key2 IN (SELECT key2 FROM s2 WHERE key3 = 'a');
```

由于key2列是s2表的一个唯一性二级索引列，所以我们可以直接把s2表上拉到外层查询的FROM子句中，并且把子查询中的搜索条件合并到外层查询的搜索条件中，上拉之后的查询就是这样的：

```
SELECT s1.* FROM s1 INNER JOIN s2
    ON s1.key2 = s2.key2
    WHERE s2.key3 = 'a';
```

为啥当子查询的查询列表处只有主键或者唯一索引列时，就可以直接将子查询转换为连接查询呢？因为主键或者唯一索引列中的数据本身就是不重复的嘛！所以对于同一条s1表中的记录，你不可能找到两条以上的符合s1.key2 = s2.key2的记录。

至于其他的，重复值消除、松散扫描等等我们就不更深入的讨论了，感兴趣的同学可以自行研究。

不能转为semi-join查询的子查询优化

注意，并不是所有包含IN子查询的查询语句都可以转换为semi-join，对于不能转换的，MySQL有这几种方法：

1、对于不相关子查询来说，会尝试把它们物化之后再参与查询

比如我们上边提到的这个查询：

SELECT * FROM s1 WHERE order_note NOT IN (SELECT order_note FROM s2 WHERE order_no= 'a')

先将子查询物化，然后再判断order_note是否在物化表的结果集中可以加快查询执行的速度。

2、不管子查询是相关的还是不相关的，都可以把IN子查询尝试转为EXISTS子查询

其实对于任意一个IN子查询来说，都可以被转为EXISTS子查询，通用的例子如下：

outer_expr IN (SELECT inner_expr FROM ... WHERE subquery_where)

可以被转换为：

EXISTS (SELECT inner_expr FROM ... WHERE subquery_where AND outer_expr=inner_expr)

为啥要转换呢？这是因为不转换的话可能用不到索引，比方说下边这个查询：

SELECT * FROM s1 WHERE order_no IN (SELECT order_no FROM s2 where s1.order_note = s2.order_note) OR insert_time > '2021-03-22 18:28:28' ;

这个查询中的子查询是一个相关子查询，而且子查询执行的时候不能使用到索引，但是将它转为EXISTS子查询后却可以使用到索引：

SELECT * FROM s1 WHERE EXISTS (SELECT 1 FROM s2 where s1.order_note = s2.order_note AND s2.order_no= s1.order_no) OR insert_time > '2021-03-22 18:28:28' ;

转为EXISTS子查询时便可能使用到s2表的idx_order_no索引了。

所以如果执行

EXPLAIN SELECT * FROM s1 WHERE order_no IN (SELECT order_no FROM s2 where s1.order_note = s2.order_note) OR insert_time > '2021-03-22 18:28:28';

show WARNINGS;

可以看见，MySQL确实把这个子查询转换为了exists子查询

```
/* select#1 */ select `mysqladv`.`s1`.`id` AS `id`,`mysqladv`.`s1`.`order_no` AS `order_no`,`mysqladv`.`s1`.`order_note`
AS `order_note`,`mysqladv`.`s1`.`insert_time` AS `insert_time`,`mysqladv`.`s1`.`expire_duration` AS `expire_duration`,
`mysqladv`.`s1`.`expire_time` AS `expire_time`,`mysqladv`.`s1`.`order_status` AS `order_status` from `mysqladv`.`s1`
where (<in_optimizer>(`mysqladv`.`s1`.`order_no`,<exists>(<index_lookup>(<cache>(`mysqladv`.`s1`.`order_no`) in s2 on
idx_order_no where ((`mysqladv`.`s1`.`order_note` = `mysqladv`.`s2`.`order_note`) and (<cache>(`mysqladv`.`s1`.`order_no`
) = `mysqladv`.`s2`.`order_no`))))) or (`mysqladv`.`s1`.`insert_time` > '2021-03-22 18:28:28'))
```

需要注意的是，如果IN子查询不满足转换为semi-join的条件，又不能转换为物化表或者转换为物化表的成本太大，那么它就会被转换为EXISTS查询。

在MySQL5.5以及之前的版本没有引进semi-join和物化的方式优化子查询时，优化器都会把IN子查询转换为EXISTS子查询，所以很多技术书籍或者博客都是建议大家把子查询转为连接，不过随着MySQL的发展，最近的版本中引入了非常多的子查询优化策略，内部的转换工作优化器会为大家自动实现。

IN子查询小结

如果IN子查询符合转换为semi-join的条件，查询优化器会优先把该子查询转换为semi-join，然后从前面所说的5种执行半连接的策略（既子查询中的表上拉、重复值消除等等）中选择成本最低的那种执行策略来执行子查询。

如果IN子查询不符合转换为semi-join的条件，那么查询优化器会从下边两种策略中找出一种成本更低的方式执行子查询：

先将子查询物化之后再执行查询

执行IN to EXISTS转换。

ANY/ALL子查询优化

如果ANY/ALL子查询是不相关子查询的话，它们在很多场合都能转换成我们熟悉的方式去执行，比方说：

原始表达式　　转换为

< ANY (SELECT inner_expr ...)　< (SELECT MAX(inner_expr) ...)

> ANY (SELECT inner_expr ...)　> (SELECT MIN(inner_expr) ...)

< ALL (SELECT inner_expr ...)　< (SELECT MIN(inner_expr) ...)

> ALL (SELECT inner_expr ...)　> (SELECT MAX(inner_expr) ...)

[NOT] EXISTS子查询的执行

如果[NOT] EXISTS子查询是不相关子查询，可以先执行子查询，得出该[NOT] EXISTS子查询的结果是TRUE还是FALSE，并重写原先的查询语句，比如对这个查询来说：

SELECT * FROM s1 WHERE EXISTS (SELECT 1 FROM s2 WHERE expire_time= 'a')
OR order_no> '2021-03-22 18:28:28' 0;

因为这个语句里的子查询是不相关子查询，所以优化器会首先执行该子查询，假设该EXISTS子查询的结果为TRUE，那么接着优化器会重写查询为：

SELECT * FROM s1 WHERE TRUE OR order_no> '2021-03-22 18:28:28' 0;

进一步简化后就变成了：

SELECT * FROM s1 WHERE TRUE;

对于相关的[NOT] EXISTS子查询来说，比如这个查询：

SELECT * FROM s1 WHERE EXISTS (SELECT 1 FROM s2 WHERE s1.order_note = s2.order_note);

很不幸，这个查询只能按照我们想象中的那种执行相关子查询的方式来执行。不过如果[NOT] EXISTS子查询中如果可以使用索引的话，那查询速度也会加快不少，比如：

```
SELECT * FROM s1 WHERE EXISTS (SELECT 1 FROM s2 WHERE s1.order_note = s2.order_no);
```

上边这个EXISTS子查询中可以使用idx_order_no来加快查询速度。