第14章、基于规则的优化

标签： MySQL是怎样运行的

大家别忘了MySQL本质上是一个软件，设计MySQL的大叔并不能要求使用这个软件的人个个都是数据库高高手，就像我写这本书的时候并不能要求各位在学之前就会了里边儿的知识。

> 吐槽一下：都会了的人谁还看呢，难道是为了精神上受感化？

也就是说我们无法避免某些同学写一些执行起来十分耗费性能的语句。即使是这样，设计MySQL的大叔还是依据一些规则，竭尽全力的把这个很糟糕的语句转换成某种可以比较高效执行的形式，这个过程也可以被称作查询重写（就是人家觉得你写的语句不好，自己再重写一遍）。本章详细唠叨一下一些比较重要的重写规则。

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。为了方便大家理解，我们后边举例子的时候都使用诸如a、b、c之类的简单字母代表某个表的列名。

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)))
```

看着就很烦，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
```

> 小贴士： 为啥用OR连接起来的表达式就不能进行常量传递呢？自己想想哈～

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。

HAVING子句和WHERE子句的合并

如果查询语句中没有出现诸如SUM、MAX等等的聚集函数以及GROUP BY子句，优化器就把HAVING子句和WHERE子句合并起来。

常量表检测

设计MySQL的大叔觉得下边这两种查询运行的特别快：

- 查询的表中一条记录没有，或者只有一条记录。

  小贴士：大家有没有觉得这一条有点儿不对劲，我还没开始查表呢咋就知道这表里边有几条记录呢？哈哈，这个其实依靠的是统计数据。不过我们说过InnoDB的统计数据数据不准确，所以这一条不能用于使用InnoDB作为存储引擎的表，只能适用于使用Memory或者MyISAM存储引擎的表。

- 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。

设计MySQL的大叔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

我们前边说过，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。为了故事的顺利发展，我们还是把之前介绍连接原理时用过的t1和t2表请出来，为了防止大家早就忘掉了，我们再看一下这两个表的结构：

```
CREATE TABLE t1 (
    m1 int,
    n1 char(1)
) Engine=InnoDB, CHARSET=utf8;

CREATE TABLE t2 (
    m2 int,
    n2 char(1)
) Engine=InnoDB, CHARSET=utf8;
```

为了唤醒大家的记忆，我们再把这两个表中的数据给展示一下：

```
mysql> SELECT * FROM t1;
+------+------+
| m1   | n1   |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
+------+------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM t2;
+------+------+
| m2   | n2   |
+------+------+
|    2 | b    |
|    3 | c    |
|    4 | d    |
```

```
+------+------+
3 rows in set (0.00 sec)
```

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：

```
mysql> SELECT * FROM t1 INNER JOIN t2 ON t1.m1 = t2.m2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
+------+------+------+------+
2 rows in set (0.00 sec)


mysql> SELECT * FROM t1 LEFT JOIN t2 ON t1.m1 = t2.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t1中m1=1, n1='a'的记录无法在被驱动表t2中找到符合ON子句条件t1.m1 = t2.m2的记录，所以就直接把这条记录加入到结果集，对应的t2表的m2和n2列的值都设置为NULL。

> 小贴士： 右（外）连接和左（外）连接其实只在驱动表的选取方式上是不同的，其余方面都是一样的，所以优化器会首先把右（外）连接查询转换成左（外）连接查询。我们后边就不再唠叨右（外）连接了。

我们知道WHERE子句的杀伤力比较大，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：

```
mysql> SELECT * FROM t1 LEFT JOIN t2 ON t1.m1 = t2.m2 WHERE t2.n2 IS NOT NULL;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
|    3 | c    |    3 | c    |
+------+------+------+------+
2 rows in set (0.01 sec)
```

由于指定了被驱动表t2的n2列不允许为NULL，所以上边的t1和t2表的左（外）连接查询和内连接查询是一样一样的。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：

```
mysql> SELECT * FROM t1 LEFT JOIN t2 ON t1.m1 = t2.m2 WHERE t2.m2 = 2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
+------+------+------+------+
1 row in set (0.00 sec)
```

在这个例子中，我们在WHERE子句中指定了被驱动表t2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：

```
mysql> SELECT * FROM t1 INNER JOIN t2 ON t1.m1 = t2.m2 WHERE t2.m2 = 2;
+------+------+------+------+
| m1   | n1   | m2   | n2   |
+------+------+------+------+
|    2 | b    |    2 | b    |
+------+------+------+------+
1 row in set (0.00 sec)
```

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。

子查询优化

我们的主题本来是唠叨MySQL查询优化器是如何处理子查询的，但是我还是有一万个担心好多同学连子查询的语法都没掌握全，所以我们就先唠叨唠叨什么是个子查询（当然不会面面俱到啦，只是说个大概哈），然后再唠叨关于子查询优化的事儿。

子查询语法

想必大家都是妈妈生下来的吧，连孙猴子都有妈妈——石头人。怀孕妈妈肚子里的那个东东就是她的孩子，类似的，在一个查询语句里的某个位置也可以有另一个查询语句，这个出现在某个查询语句的某个位置中的查询就被称为子查询（我们也可以称它为宝宝查询哈哈），那个充当“妈妈”角色的查询也被称之为外层查询。不像人们怀孕时宝宝们都只在肚子里，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：

  ```
  mysql> SELECT (SELECT m1 FROM t1 LIMIT 1);
  +----------------------------+
  | (SELECT m1 FROM t1 LIMIT 1) |
  +----------------------------+
  |                          1 |
  +----------------------------+
  1 row in set (0.00 sec)
  ```

  其中的(SELECT m1 FROM t1 LIMIT 1)就是我们唠叨的所谓的子查询。

- FROM子句中

  比如：

  ```
  SELECT m, n FROM (SELECT m2 + 1 AS m, n2 AS n FROM t2 WHERE m2 > 2) AS t;
  +------+------+
  | m    | n    |
  +------+------+
  |    4 | c    |
  |    5 | d    |
  +------+------+
  2 rows in set (0.00 sec)
  ```

  这个例子中的子查询是：(SELECT m2 + 1 AS m, n2 AS n FROM t2 WHERE m2 > 2)，很特别的地方是它出现在了FROM子句中。FROM子句里边儿不是存放我们要查询的表的名称么，这里放进来一个子查询是个什么鬼？其实这里我们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，设计MySQL的大叔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：

  ```
  mysql> SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2);
  +------+------+
  | m1   | n1   |
  +------+------+
  |    2 | b    |
  |    3 | c    |
  +------+------+
  2 rows in set (0.00 sec)
  ```

  这个查询表明我们想要将(SELECT m2 FROM t2)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t1表中的某些记录，这些记录的m1列的值能在t2表的m2列找到匹配的值。

- ORDER BY子句中

  虽然语法支持，但没啥子意义，不唠叨这种情况了。

- GROUP BY子句中

  同上~

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 t1 LIMIT 1);
```

或者这样：

```
SELECT * FROM t1 WHERE m1 = (SELECT MIN(m2) FROM t2);
```

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 t1 WHERE (m1, n1) = (SELECT m2, n2 FROM t2 LIMIT 1);
  ```

  其中的`(SELECT m2, n2 FROM t2 LIMIT 1)`就是一个行子查询，整条语句的含义就是要从`t1`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喽，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 t1 WHERE m1 IN (SELECT m2 FROM t2);
  ```

  其中的`(SELECT m2 FROM t2)`就是一个列子查询，表明查询出`t2`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 t1 WHERE (m1, n1) IN (SELECT m2, n2 FROM t2);
  ```

  其中的`(SELECT m2, n2 FROM t2)`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，所以也就不举例子了哈。

- 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 t1 WHERE m1 IN (SELECT m2 FROM t2 WHERE n1 = n2);
  ```

  例子中的子查询是`(SELECT m2 FROM t2 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`t1`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。

子查询在布尔表达式中的使用

你说写下边这样的子查询有啥意义：

```
SELECT (SELECT m1 FROM t1 LIMIT 1);
```

貌似没啥意义～ 我们平时用子查询最多的地方就是把它作为布尔表达式的一部分来作为搜索条件用在`WHERE`子句或者`ON`子句里。所以我们这里来总结一下子查询在布尔表达式中的使用场景。

- 使用`=`、`>`、`<`、`>=`、`<=`、`<>`、`!=`、`<=>`作为布尔表达式的操作符

  这些操作符具体是啥意思就不用我多介绍了吧，如果你不知道的话，那我真的很佩服你是靠着啥勇气一口气看到这里的～ 为了方便，我们就把这些操作符称为`comparison_operator`吧，所以子查询组成的布尔表达式就长这样：

  ```
  操作数 comparison_operator (子查询)
  ```

  这里的操作数可以是某个列名，或者是一个常量，或者是一个更复杂的表达式，甚至可以是另一个子查询。但是需要注意的是，这里的子查询只能是标量子查询或者行子查询，也就是子查询的结果只能返回一个单一的值或者只能是一条记录。比如这样（标量子查询）：

  ```
  SELECT * FROM t1 WHERE m1 < (SELECT MIN(m2) FROM t2);
  ```

  或者这样（行子查询）：

  ```
  SELECT * FROM t1 WHERE (m1, n1) = (SELECT m2, n2 FROM t2 LIMIT 1);
  ```

- [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`comparison_operator`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t1表中的某些记录，这些记录存在于子查询的结果集中：

```
SELECT * FROM t1 WHERE (m1, n1) IN (SELECT m2, n2 FROM t2);
```

- ANY/SOME（ANY和SOME是同义词）

  具体的语法形式如下：

  ```
  操作数 comparison_operator 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comparison_operator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 t1 WHERE m1 > ANY(SELECT m2 FROM t2);
  ```

  这个查询的意思就是对于t1表的某条记录的m1列的值来说，如果子查询(SELECT m2 FROM t2)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 t1 WHERE m1 > (SELECT MIN(m2) FROM t2);
  ```

  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 comparison_operator 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comparison_operator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 t1 WHERE m1 > ALL(SELECT m2 FROM t2);
  ```

  这个查询的意思就是对于t1表的某条记录的m1列的值来说，如果子查询(SELECT m2 FROM t2)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 t1 WHERE m1 > (SELECT MAX(m2) FROM t2);
  ```

  小贴士：觉得ANY和ALL有点晕的同学多看两遍哈～

- 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：

  ```
  [NOT] EXISTS (子查询)
  ```

  我们举一个例子啊：

  ```
  SELECT * FROM t1 WHERE EXISTS (SELECT 1 FROM t2);
  ```

  对于子查询(SELECT 1 FROM t2)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 t2)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。

  不扩起来的子查询是非法的，比如这样：

  ```
  mysql> SELECT SELECT m1 FROM t1;

  ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'S
  ```

- 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，也就是查询列表中，比如这样就是非法的：

  ```
  mysql> SELECT (SELECT m1, n1 FROM t1);

  ERROR 1241 (21000): Operand should contain 1 column(s)
  ```

- 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。

  比如这样是非法的：

  ```
  mysql> SELECT * FROM t1 WHERE m1 IN (SELECT * FROM t2 LIMIT 2);
  ```

```
ERROR 1235 (42000): This version of MySQL doesn't yet support 'LIMIT & IN/ALL/ANY/SOME subquery'
```

为啥不合法？人家就这么规定的，不解释～ 可能以后的版本会支持吧。正因为`[NOT] IN/ANY/SOME/ALL`子查询不支持`LIMIT`语句，所以子查询中的这些语句也就是多余的了：

- `ORDER BY`子句

  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一点儿都不重要，比如下边这个语句中的`ORDER BY`子句简直就是画蛇添足：

  ```
  SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2 ORDER BY m2);
  ```

- `DISTINCT`语句

  集合里的值去不去重也没啥意义，比如这样：

  ```
  SELECT * FROM t1 WHERE m1 IN (SELECT DISTINCT m2 FROM t2);
  ```

- 没有聚集函数以及`HAVING`子句的`GROUP BY`子句。

  在没有聚集函数以及`HAVING`子句时，`GROUP BY`子句就是个摆设，比如这样：

  ```
  SELECT * FROM t1 WHERE m1 IN (SELECT m2 FROM t2 GROUP BY m2);
  ```

对于这些冗余的语句，查询优化器在一开始就把它们给干掉了。

- 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。

  比方说这样：

  ```
  mysql> DELETE FROM t1 WHERE m1 < (SELECT MAX(m1) FROM t1);


  ERROR 1093 (HY000): You can't specify target table 't1' for update in FROM clause
  ```

子查询在MySQL中是怎么执行的

好了，关于子查询的基础语法我们用最快的速度温习了一遍，如果想了解更多语法细节，大家可以去查看一下`MySQL`的文档哈，现在我们就假设各位都懂了啥是个子查询了喔，接下来就要唠叨具体某种类型的子查询在`MySQL`中是怎么执行的了，想想就有点儿小激动呢～ 当然，为了故事的顺利发展，我们的例子也需要跟随形势鸟枪换炮，还是要祭出我们用了n遍的`single_table`表：

```
CREATE TABLE single_table (
    id INT NOT NULL AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    KEY idx_key1 (key1),
    UNIQUE KEY idx_key2 (key2),
    KEY idx_key3 (key3),
    KEY idx_key_part(key_part1, key_part2, key_part3)
) Engine=InnoDB CHARSET=utf8;
```

为了方便，我们假设有两个表`s1`、`s2`与这个`single_table`表的构造是相同的，而且这两个表里边儿有10000条记录，除id列外其余的列都插入随机值。下边正式开始我们的表演。

小白们眼中子查询的执行方式

在我还是一个单纯无知的少年时，觉得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 key1 IN (SELECT common_field FROM s2);
  ```

  我年少时觉得这个查询是的执行方式是这样的：

  - 先单独执行`(SELECT common_field 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 key1 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 key1 IN (SELECT common_field FROM s2 WHERE s1.key2 = s2.key2);
  ```

这个查询中的子查询中出现了`s1.key2 = s2.key2`这样的条件，意味着该子查询的执行依赖着外层查询的值，所以我年少时觉得这个查询的执行方式是这样的：

- 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.key2`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～

告诉我不只是我一个人是这样认为的，这样认为的同学请举起你们的双手～～～ 哇唔，还真不少～

其实设计`MySQL`的大叔想了一系列的办法来优化子查询的执行，大部分情况下这些优化措施其实挺有效的，但是保不齐有的时候马失前蹄，下边我们详细唠叨各种不同类型的子查询具体是怎么执行的。

小贴士： 我们下边即将唠叨的关于MySQL优化子查询的执行方式的事儿都是基于MySQL5.7这个版本的，以后版本可能有更新的优化策略！

标量子查询、行子查询的执行方式

我们经常在下边两个场景中使用到标量子查询或者行子查询：

- `SELECT`子句中，我们前边说过的在查询列表中的子查询必须是标量子查询。
- 子查询使用`=`、`>`、`<`、`>=`、`<=`、`<>`、`!=`、`<=>`等操作符和某个操作数组成一个布尔表达式，这样的子查询必须是标量子查询或者行子查询。

对于上述两种场景中的不相关标量子查询或者行子查询来说，它们的执行方式是简单的，比方说下边这个查询语句：

```
SELECT * FROM s1
    WHERE key1 = (SELECT common_field FROM s2 WHERE key3 = 'a' LIMIT 1);
```

它的执行方式和年少的我想的一样：

- 先单独执行`(SELECT common_field 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 key1 = ...`。

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
    key1 = (SELECT common_field FROM s2 WHERE s1.key3 = s2.key3 LIMIT 1);
```

事情也和年少的我想的一样，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.key3`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～

也就是说对于一开始唠叨的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。

IN子查询优化

物化表的提出

对于不相关的`IN`子查询，比如这样：

```
SELECT * FROM s1
    WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = '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，可是很遗憾的是设计`MySQL`的大叔为了优化`IN`子查询倾注了太多心血（毕竟`IN`子查询是我们日常生活中最常用的子查询类型），所以整个执行过程并不像我们想象的那么简单(>_<)。

其实说句老实话，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还是蛮高的，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：

- 结果集太多，可能内存中都放不下～
- 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。

于是乎设计MySQL的大叔想了一个招：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：

- 该临时表的列就是子查询结果集中的列。
- 写入临时表的记录会被去重。

  我们说IN语句是判断某个操作数在不在某个集合中，集合中的值重不重复对整个IN语句的结果并没有啥子关系，所以我们在将结果集写入临时表时对记录进行去重可以让临时表变得更小，更省地方～

  > 小贴士： 临时表如何对记录进行去重？这不是小意思嘛，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就好了嘛～

- 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。

  > 小贴士： IN语句的本质就是判断某个操作数在不在某个集合里，如果集合中的数据建立了哈希索引，那么这个匹配的过程就是超级快的。 有同学不知道哈希索引是什么？我这里就不展开了，自己上网找找吧，不会了再来问我～

  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。

设计MySQL的大叔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。

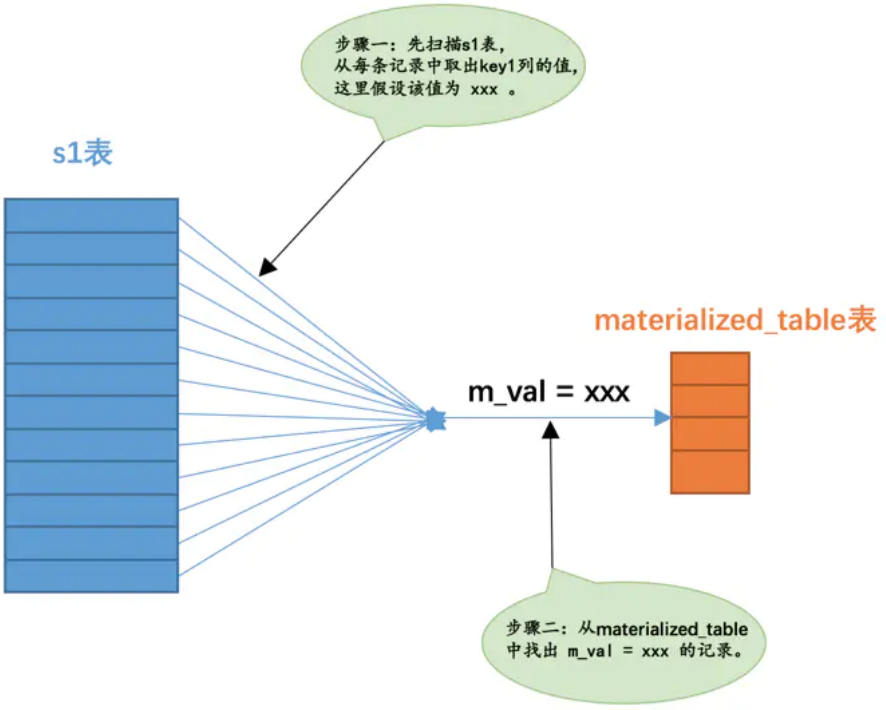
物化表转连接

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：

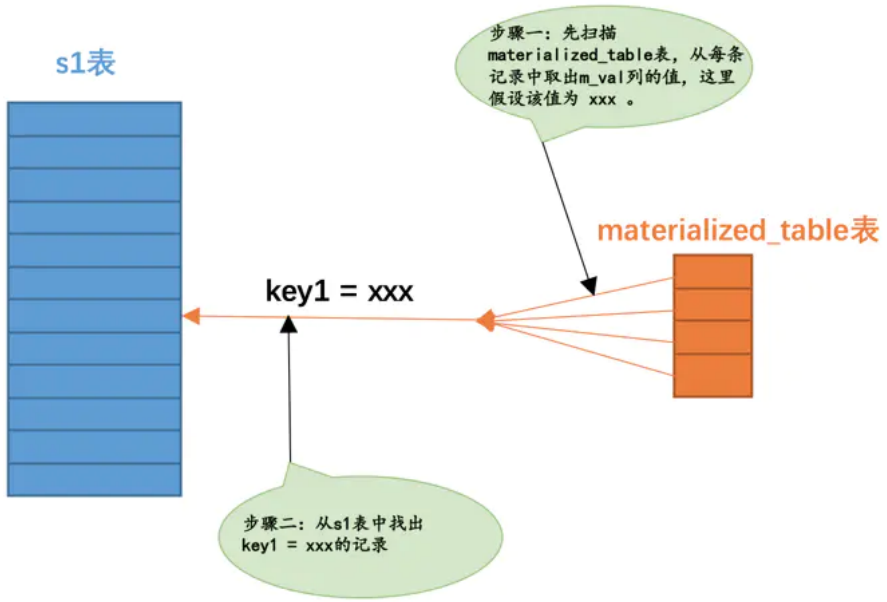
```
SELECT * FROM s1
    WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a');
```

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，那么这个查询其实可以从下边两种角度来看待：

- 从表s1的角度来看待，整个查询的意思其实是：对于s1表中的每条记录来说，如果该记录的key1列的值在子查询对应的物化表中，则该记录会被加入最终的结果集。画个图表示一下就是这样：



- 从子查询物化表的角度来看待，整个查询的意思其实是：对于子查询物化表的每个值来说，如果能在s1表中找到对应的key1列的值与该值相等的记录，那么就把这些记录加入到最终的结果集。画个图表示一下就是这样：



也就是说其实上边的查询就相当于表s1和子查询物化表materialized_table进行内连接：

```
SELECT s1.* FROM s1 INNER JOIN materialized_table ON key1 = 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：

- 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）。
- 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key1 = xxx对s1表进行单表访问的成本（非常庆幸key1列上建立了索引，所以这个步骤是非常快的）。

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

虽然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，但是不管怎么说，我们见识到了将子查询转换为连接的强大作用，设计MySQL的大叔继续开脑洞：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 key1 IN (SELECT common_field FROM s2 WHERE key3 = '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.key3 = 'a'之后的结果集）中找到一条或多条记录，这些记录的common_field的值等于s1表记录的key1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.key1 = s2.common_field
    WHERE s2.key3 = '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.key3 = 'a'之后的结果集）中有多少条记录满足s1.key1 = s2.common_field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.key1 = s2.common_field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.key1 = s2.common_field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.key1 = s2.common_field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.key1 = s2.common_field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，所以设计MySQL的大叔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.key1 = s2.common_field
    WHERE key3 = 'a';
```

小贴士： 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，所以我们不需要，也不能尝试把上边这个语句放到黑框框里运行，我只是想说明一下上边的子查询在MySQL内部会被转换为类似上边语句的半连接～

概念是有了，怎么实现这种所谓的半连接呢？设计MySQL的大叔准备了好几种办法。

- 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，比如这个

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

  由于key2列是s2表的唯一二级索引列，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？哎呀，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的记录呀～

- DuplicateWeedout execution strategy （重复值消除）

  对于这个查询来说：

  ```
  SELECT * FROM s1
      WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a');
  ```

  转换为半连接查询后，s1表中的某条记录可能在s2表中有多条匹配的记录，所以该条记录可能多次被添加到最后的结果集中，为了消除重复，我们可以建立一个临时表，比方说这个临时表长这样：

  ```
  CREATE TABLE tmp (
      id PRIMARY KEY
  );
  ```

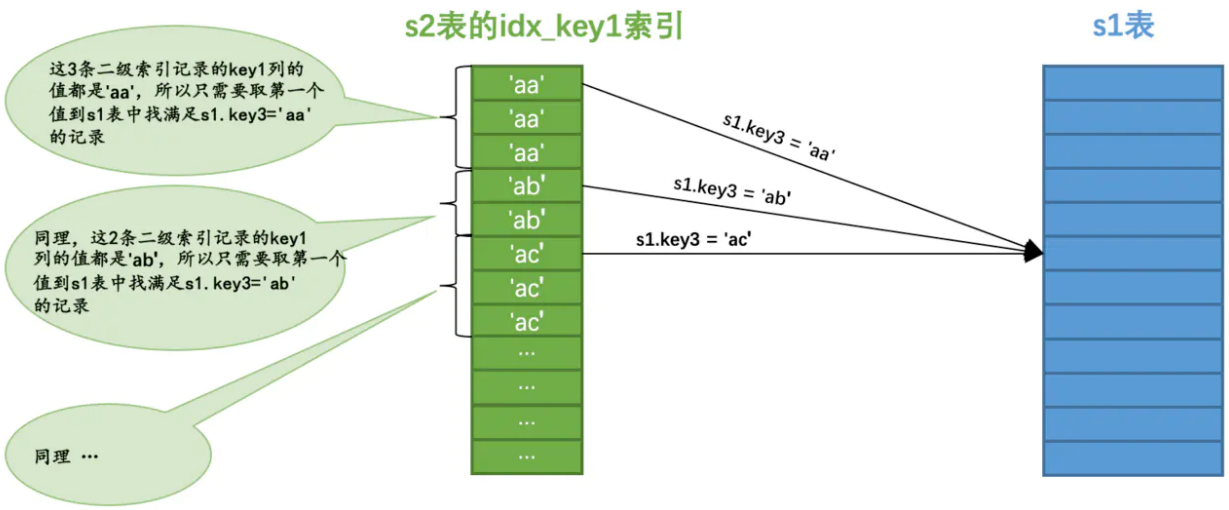
  这样在执行连接查询的过程中，每当某条s1表中的记录要加入结果集时，就首先把这条记录的id值加入到这个临时表里，如果添加成功，说明之前这条s1表中的记录并没有加入最终的结果集，现在把该记录添加到最终的结果集；如果添加失败，说明之前这条s1表中的记录已经加入过最终的结果集，这里直接把它丢弃就好了，这种使用临时表消除semi-join结果集中的重复值的方式称之为DuplicateWeedout。

- LooseScan execution strategy （松散扫描）

  大家看这个查询：

  ```
  SELECT * FROM s1
      WHERE key3 IN (SELECT key1 FROM s2 WHERE key1 > 'a' AND key1 < 'b');
  ```

  在子查询中，对于s2表的访问可以使用到key1列的索引，而恰好子查询的查询列表处就是key1列，这样在将该查询转换为半连接查询后，如果将s2作为驱动表执行查询的话，那么执行过程就是这样：



如图所示，在s2表的idx_key1索引中，值为'aa'的二级索引记录一共有3条，那么只需要取第一条的值到s1表中查找s1.key3 = 'aa'的记录，如果能在s1表中找到对应的记录，那么就把对应的记录加入到结果集。依此类推，其他值相同的二级索引记录，也只需要取第一条记录的值到s1表中找匹配的记录，这种虽然是扫描索引，但只取值相同的记录的第一条去做匹配操作的方式称之为松散扫描。

- Semi-join Materialization execution strategy

  我们之前介绍的先把外层查询的IN子句中的不相关子查询进行物化，然后再进行外层查询的表和物化表的连接本质上也算是一种semi-join，只不过由于物化表中没有重复的记录，所以可以直接将子查询转为连接查询。

- FirstMatch execution strategy（首次匹配）

  FirstMatch是一种最原始的半连接执行方式，跟我们年少时认为的相关子查询的执行方式是一样一样的，就是说先取一条外层查询的中的记录，然后到子查询的表中寻找符合匹配条件的记录，如果能找到一条，则将该外层查询的记录放入最终的结果集并且停止查找更多匹配的记录，如果找不到则把该外层查询的记录丢弃掉；然后再开始取下一条外层查询中的记录，重复上边这个过程。

对于某些使用IN语句的相关子查询，比方这个查询：

```
SELECT * FROM s1
    WHERE key1 IN (SELECT common_field FROM s2 WHERE s1.key3 = s2.key3);
```

它也可以很方便的转为半连接，转换后的语句类似这样：

```
SELECT s1.* FROM s1 SEMI JOIN s2
    ON s1.key1 = s2.common_field AND s1.key3 = s2.key3;
```

然后就可以使用我们上边介绍过的DuplicateWeedout、LooseScan、FirstMatch等半连接执行策略来执行查询，当然，如果子查询的查询列表处只有主键或者唯一二级索引列，还可以直接使用table pullout的策略来执行查询，但是需要大家注意的是，由于相关子查询并不是一个独立的查询，所以不能转换为物化表来执行查询。

semi-join的适用条件

当然，并不是所有包含IN子查询的查询语句都可以转换为semi-join，只有形如这样的查询才可以被转换为semi-join：

```
SELECT ... FROM outer_tables
    WHERE expr IN (SELECT ... FROM inner_tables ...) AND ...
```

或者这样的形式也可以：

```
SELECT ... FROM outer_tables
    WHERE (oe1, oe2, ...) IN (SELECT ie1, ie2, ... FROM inner_tables ...) AND ...
```

用文字总结一下，只有符合下边这些条件的子查询才可以被转换为semi-join：

- 该子查询必须是和IN语句组成的布尔表达式，并且在外层查询的WHERE或者ON子句中出现。
- 外层查询也可以有其他的搜索条件，只不过和IN子查询的搜索条件必须使用AND连接起来。
- 该子查询必须是一个单一的查询，不能是由若干查询由UNION连接起来的形式。
- 该子查询不能包含GROUP BY或者HAVING语句或者聚集函数。
- ... 还有一些条件比较少见，就不唠叨啦～

不适用于semi-join的情况

对于一些不能将子查询转位semi-join的情况，典型的比如下边这几种：

- 外层查询的WHERE条件中有其他搜索条件与IN子查询组成的布尔表达式使用OR连接起来

  ```
  SELECT * FROM s1
      WHERE key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a')
          OR key2 > 100;
  ```

- 使用NOT IN而不是IN的情况

  ```
  SELECT * FROM s1
      WHERE key1 NOT IN (SELECT common_field FROM s2 WHERE key3 = 'a')
  ```

- 在SELECT子句中的IN子查询的情况

  ```
  SELECT key1 IN (SELECT common_field FROM s2 WHERE key3 = 'a') FROM s1 ;
  ```

- 子查询中包含GROUP BY、HAVING或者聚集函数的情况

  ```
  SELECT * FROM s1
      WHERE key2 IN (SELECT COUNT(*) FROM s2 GROUP BY key1);
  ```

- 子查询中包含UNION的情况

  ```
  SELECT * FROM s1 WHERE key1 IN (
      SELECT common_field FROM s2 WHERE key3 = 'a'
  ```

```
    UNION
    SELECT common_field FROM s2 WHERE key3 = 'b'
);
```

MySQL仍然留了两手绝活来优化不能转为semi-join查询的子查询，那就是：

- 对于不相关子查询来说，可以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 key1 NOT IN (SELECT common_field FROM s2 WHERE key3 = 'a')
  ```

  先将子查询物化，然后再判断key1是否在物化表的结果集中可以加快查询执行的速度。

  > 小贴士： 请注意这里将子查询物化之后不能转为和外层查询的表的连接，只能是先扫描s1表，然后对s1表的某条记录来说，判断该记录的key1值在不在物化表中。

- 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)
  ```

  当然这个过程中有一些特殊情况，比如在outer_expr或者inner_expr值为NULL的情况下就比较特殊。因为有NULL值作为操作数的表达式结果往往是NULL，比方说：

  ```
  mysql> SELECT NULL IN (1, 2, 3);
  +-------------------+
  | NULL IN (1, 2, 3) |
  +-------------------+
  |              NULL |
  +-------------------+
  1 row in set (0.00 sec)

  mysql> SELECT 1 IN (1, 2, 3);
  +----------------+
  | 1 IN (1, 2, 3) |
  +----------------+
  |              1 |
  +----------------+
  1 row in set (0.00 sec)

  mysql> SELECT NULL IN (NULL);
  +----------------+
  | NULL IN (NULL) |
  +----------------+
  |           NULL |
  +----------------+
  1 row in set (0.00 sec)
  ```

  而EXISTS子查询的结果肯定是TRUE或者FASLE：

  ```
  mysql> SELECT EXISTS (SELECT 1 FROM s1 WHERE NULL = 1);
  +-----------------------------------------+
  | EXISTS (SELECT 1 FROM s1 WHERE NULL = 1) |
  +-----------------------------------------+
  |                                       0 |
  +-----------------------------------------+
  1 row in set (0.01 sec)

  mysql> SELECT EXISTS (SELECT 1 FROM s1 WHERE 1 = NULL);
  +-----------------------------------------+
  | EXISTS (SELECT 1 FROM s1 WHERE 1 = NULL) |
  +-----------------------------------------+
  ```

```
|                                          0 |
+--------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT EXISTS (SELECT 1 FROM s1 WHERE NULL = NULL);
+---------------------------------------------+
| EXISTS (SELECT 1 FROM s1 WHERE NULL = NULL) |
+---------------------------------------------+
|                                           0 |
+---------------------------------------------+
1 row in set (0.00 sec)
```

但是幸运的是，我们大部分使用IN子查询的场景是把它放在WHERE或者ON子句中，而WHERE或者ON子句是不区分NULL和FALSE的，比方说：

```
mysql> SELECT 1 FROM s1 WHERE NULL;
Empty set (0.00 sec)

mysql> SELECT 1 FROM s1 WHERE FALSE;
Empty set (0.00 sec)
```

所以只要我们的IN子查询是放在WHERE或者ON子句中的，那么IN -> EXISTS的转换就是没问题的。说了这么多，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 key1 IN (SELECT key3 FROM s2 where s1.common_field = s2.common_field)
        OR key2 > 1000;
```

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.common_field = s2.common_field AND s2.key3 = s1.key1)
        OR key2 > 1000;
```

转为EXISTS子查询时便可以使用到s2表的idx_key3索引了。

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。

> 小贴士： 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，好多同学就惊呼我明明写的是一个不相关子查询，为啥要按照执行相关子查询的方式来执行呢？所以当时好多声音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，大家可以稍微放心的使用子查询了，内部的转换工作优化器会为大家自动实现。

小结一下

- 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，然后再考虑下边5种执行半连接的策略中哪个成本最低：
  - Table pullout
  - DuplicateWeedout
  - LooseScan
  - Materialization
  - FirstMatch

  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 key1 = 'a')
        OR key2 > 100;
```

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 key2 > 100;
```

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.common_field = s2.common_field);
```

很不幸，这个查询只能按照我们年少时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.common_field = s2.key1);
```

上边这个EXISTS子查询中可以使用idx_key1来加快查询速度。

对于派生表的优化

我们前边说过把子查询放在外层查询的FROM子句后，那么这个子查询的结果相当于一个派生表，比如下边这个查询：

```
SELECT * FROM  (
        SELECT id AS d_id,  key3 AS d_key3 FROM s2 WHERE key1 = 'a'
    ) AS derived_s1 WHERE d_key3 = 'a';
```

子查询(SELECT id AS d_id, key3 AS d_key3 FROM s2 WHERE key1 = 'a')的结果就相当于一个派生表，这个表的名称是derived_s1，该表有两个列，分别是d_id和d_key3。

对于含有派生表的查询，MySQL提供了两种执行策略：

- 最容易想到的就是把派生表物化。

  我们可以将派生表的结果集写到一个内部的临时表中，然后就把这个物化表当作普通表一样参与查询。当然，在对派生表进行物化时，设计MySQL的大叔使用了一种称为延迟物化的策略，也就是在查询中真正使用到派生表时才回去尝试物化派生表，而不是还没开始执行查询呢就把派生表物化掉。比方说对于下边这个含有派生表的查询来说：

  ```
  SELECT * FROM (
          SELECT * FROM s1 WHERE key1 = 'a'
      ) AS derived_s1 INNER JOIN s2
      ON derived_s1.key1 = s2.key1
      WHERE s2.key2 = 1;
  ```

  如果采用物化派生表的方式来执行这个查询的话，那么执行时首先会到s2表中找出满足s2.key2 = 1的记录，如果压根儿找不到，说明参与连接的s2表记录就是空的，所以整个查询的结果集就是空的，所以也就没有必要去物化查询中的派生表了。

- 将派生表和外层的表合并，也就是将查询重写为没有派生表的形式

  我们来看这个贼简单的包含派生表的查询：

  ```
  SELECT * FROM (SELECT * FROM s1 WHERE key1 = 'a') AS derived_s1;
  ```

  这个查询本质上就是想查看s1表中满足key1 = 'a'条件的的全部记录，所以和下边这个语句是等价的：

  ```
  SELECT * FROM s1 WHERE key1 = 'a';
  ```

  对于一些稍微复杂的包含派生表的语句，比如我们上边提到的那个：

  ```
  SELECT * FROM (
          SELECT * FROM s1 WHERE key1 = 'a'
      ) AS derived_s1 INNER JOIN s2
      ON derived_s1.key1 = s2.key1
      WHERE s2.key2 = 1;
  ```

  我们可以将派生表与外层查询的表合并，然后将派生表中的搜索条件放到外层查询的搜索条件中，就像这样：

  ```
  SELECT * FROM s1 INNER JOIN s2
  ```

```
    ON s1.key1 = s2.key1
    WHERE s1.key1 = 'a' AND s2.key2 = 1;
```

这样通过将外层查询和派生表合并的方式成功的消除了派生表，也就意味着我们没必要再付出创建和访问临时表的成本了。可是并不是所有带有派生表的查询都能被成功的和外层查询合并，当派生表中有这些语句就不可以和外层查询合并：

- 聚集函数，比如MAX()、MIN()、SUM()啥的
- DISTINCT
- GROUP BY
- HAVING
- LIMIT
- UNION 或者 UNION ALL
- 派生表对应的子查询的`SELECT`子句中含有另一个子查询
- ... 还有些不常用的情况就不多说了哈～

所以`MySQL`在执行带有派生表的时候，优先尝试把派生表和外层查询合并掉，如果不行的话，再把派生表物化掉执行查询。