"

Explain 详解（上）

标签： MySQL 是怎样运行的

一条查询语句在经过MySQL查询优化器的各种基于成本和规则的优化会后生成一个所谓的执行计划，这个执行计划展示了接下来具体执行查询的方式，比如多表连接的顺序是什么，对于每个表采用什么访问方法来具体执行查询等等。设计MySQL的大叔贴心的为我们提供了EXPLAIN语句来帮助我们查看某个查询语句的具体执行计划，本章的内容就是为了帮助大家看懂EXPLAIN语句的各个输出项都是干嘛使的，从而可以有针对性的提升我们查询语句的性能。

如果我们想看看某个查询的执行计划的话，可以在具体的查询语句前边加一个EXPLAIN，就像这样：

```
mysql> EXPLAIN SELECT 1;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | No tables used |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
1 row in set, 1 warning (0.01 sec)
```

然后这输出的一大坨东西就是所谓的执行计划，我的任务就是带领大家看懂这一大坨东西里边的每个列都是干啥用的，以及在这个执行计划的辅助下，我们应该怎样改进自己的查询语句以使查询执行起来更高效。其实除了以SELECT开头的查询语句，其余的DELETE、INSERT、REPLACE以及UPDATE语句前边都可以加上EXPLAIN这个词儿，用来查看这些语句的执行计划，不过我们这里对SELECT语句更感兴趣，所以后边只会以SELECT语句为例来描述EXPLAIN语句的用法。为了让大家先有一个感性的认识，我们把EXPLAIN语句输出的各个列的作用先大致罗列一下：

列名	描述
id	在一个大的查询语句中每个SELECT关键字都对应一个唯一的id
select_type	SELECT关键字对应的那个查询的类型
table	表名
partitions	匹配的分区信息
type	针对单表的访问方法
possible_keys	可能用到的索引
key	实际上使用的索引
key_len	实际使用到的索引长度
ref	当使用索引列等值查询时，与索引列进行等值匹配的对象信息
rows	预估的需要读取的记录条数
filtered	某个表经过搜索条件过滤后剩余记录条数的百分比
Extra	一些额外的信息

需要注意的是，大家如果看不懂上边输出列含义，那是正常的，千万不要纠结～。我在这里把它们都列出来只是为了描述一个轮廓，让大家有一个大致的印象，下边会细细道来，等会儿说完了不信你不会～ 为了故事的顺利发展，我们还是要请出我们前边已经用了n遍的single_table表，为了防止大家忘了，再把它的结构描述一遍：

```
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;
```

我们仍然假设有两个和single_table表构造一模一样的s1、s2表，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。为了让大家有比较好的阅读体验，我们下边并不准备严格按照EXPLAIN输出列的顺序来介绍这些列分别是干嘛的，大家注意一下就好了。

执行计划输出中各列详解

table

不论我们的查询语句有多复杂，里边儿包含了多少个表，到最后也是需要对每个表进行单表访问的，所以设计MySQL的大叔规定EXPLAIN语句输出的每条记录都对应着某个单表的访问方法，该条记录的table列代表着该表的表名。所以我们看一条比较简单的查询语句：

```
mysql> EXPLAIN SELECT * FROM s1;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

这个查询语句只涉及对s1表的单表查询，所以EXPLAIN输出中只有一条记录，其中的table列的值是s1，表明这条记录是用来说明对s1表的单表访问方法的。

下边我们看一下一个连接查询的执行计划：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL                                               |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | Using join buffer (Block Nested Loop)              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

可以看到这个连接查询的执行计划中有两条记录，这两条记录的table列分别是s1和s2，这两条记录用来分别说明对s1表和s2表的访问方法是什么。

id

我们知道我们写的查询语句一般都以SELECT关键字开头，比较简单的查询语句里只有一个SELECT关键字，比如下边这个查询语句：

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

稍微复杂一点的连接查询中也只有一个SELECT关键字，比如：

```
SELECT * FROM s1 INNER JOIN s2
    ON s1.key1 = s2.key1
    WHERE s1.common_field = 'a';
```

但是下边两种情况下在一条查询语句中会出现多个SELECT关键字：

- 查询中包含子查询的情况

  比如下边这个查询语句中就包含2个SELECT关键字：

  ```
  SELECT * FROM s1
      WHERE key1 IN (SELECT * FROM s2);
  ```

- 查询中包含UNION语句的情况

比如下边这个查询语句中也包含2个SELECT关键字：

```
SELECT * FROM s1  UNION SELECT * FROM s2;
```

查询语句中每出现一个SELECT关键字，设计MySQL的大叔就会为它分配一个唯一的id值。这个id值就是EXPLAIN语句的第一个列，比如下边这个查询中只有一个SELECT关键字，所以EXPLAIN的结果中也就只有一条id列为1的记录：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.03 sec)
```

对于连接查询来说，一个SELECT关键字后边的FROM子句中可以跟随多个表，所以在连接查询的执行计划中，每个表都会对应一条记录，但是这些记录的id值都是相同的，比如：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL                                               |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | Using join buffer (Block Nested Loop)              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

可以看到，上述连接查询中参与连接的s1和s2表分别对应一条记录，但是这两条记录对应的id值都是1。这里需要大家记住的是，在连接查询的执行计划中，每个表都会对应一条记录，这些记录的id列的值是相同的，出现在前边的表表示驱动表，出现在后边的表表示被驱动表。所以从上边的EXPLAIN输出中我们可以看出，查询优化器准备让s1表作为驱动表，让s2表作为被驱动表来执行查询。

对于包含子查询的查询语句来说，就可能涉及多个SELECT关键字，所以在包含子查询的查询语句的执行计划中，每个SELECT关键字都会对应一个唯一的id值，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_key3      | NULL     | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | SUBQUERY    | s2    | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9954 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.02 sec)
```

从输出结果中我们可以看到，s1表在外层查询中，外层查询有一个独立的SELECT关键字，所以第一条记录的id值就是1，s2表在子查询中，子查询有一个独立的SELECT关键字，所以第二条记录的id值就是2。

但是这里大家需要特别注意，查询优化器可能对涉及子查询的查询语句进行重写，从而转换为连接查询。所以如果我们想知道查询优化器对某个包含子查询的语句是否进行了重写，直接查看执行计划就好了，比如说：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key3 FROM s2 WHERE common_field = 'a');
+----+-------------+-------+------------+------+---------------+----------+---------+-----------------------+------+----------+--------------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref                   | rows | filtered | Extra                    |
+----+-------------+-------+------------+------+---------------+----------+---------+-----------------------+------+----------+--------------------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key3      | NULL     | NULL    | NULL                  | 9954 |    10.00 | Using where; Start temporary |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s2.key3 |    1 |   100.00 | End temporary            |
+----+-------------+-------+------------+------+---------------+----------+---------+-----------------------+------+----------+--------------------------+
2 rows in set, 1 warning (0.00 sec)
```

可以看到，虽然我们的查询语句是一个子查询，但是执行计划中s1和s2表对应的记录的id值全部是1，这就表明了查询优化器将子查询转换为了连接查询。

对于包含UNION子句的查询语句来说，每个SELECT关键字对应一个id值也是没错的，不过还是有点儿特别的东西，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1  UNION SELECT * FROM s2;
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL            |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | NULL            |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL | Using temporary |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
3 rows in set, 1 warning (0.00 sec)
```

这个语句的执行计划的第三条记录是个什么鬼？为毛id值是NULL，而且table列长的也怪怪的？大家别忘了UNION子句是干嘛用的，它会把多个查询的结果集合并起来并对结果集中的记录进行去重，怎么去重呢？MySQL使用的是内部的临时表。正如上边的查询计划中所示，UNION子句是为了把id为1的查询和id为2的查询的结果集合并起来并去重，所以在内部创建了一个名为<union1, 2>的临时表（就是执行计划第三条记录的table列的名称），id为NULL表明这个临时表是为了合并两个查询的结果集而创建的。

跟UNION对比起来，UNION ALL就不需要为最终的结果集进行去重，它只是单纯的把多个查询的结果集中的记录合并成一个并返回给用户，所以也就不需要使用临时表。所以在包含UNION ALL子句的查询的执行计划中，就没有那个id为NULL的记录，如下所示：

```
mysql> EXPLAIN SELECT * FROM s1  UNION ALL SELECT * FROM s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | PRIMARY     | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL  |
|  2 | UNION       | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
2 rows in set, 1 warning (0.01 sec)
```

select_type

通过上边的内容我们知道，一条大的查询语句里边可以包含若干个SELECT关键字，每个SELECT关键字代表着一个小的查询语句，而每个SELECT关键字的FROM子句中都可以包含若干张表（这些表用来做连接查询），每一张表都对应着执行计划输出中的一条记录，对于在同一个SELECT关键字中的表来说，它们的id值是相同的。

设计MySQL的大叔为每一个SELECT关键字代表的小查询都定义了一个称之为select_type的属性，意思是我们只要知道了某个小查询的select_type属性，就知道了这个小查询在整个大查询中扮演了一个什么角色，口说无凭，我们还是先来见识见识这个select_type都能取哪些值（为了精确起见，我们直接使用文档中的英文做简要描述，随后会进行详细解释的）：

名称	描述
SIMPLE	Simple SELECT (not using UNION or subqueries)
PRIMARY	Outermost SELECT
UNION	Second or later SELECT statement in a UNION
UNION RESULT	Result of a UNION
SUBQUERY	First SELECT in subquery
DEPENDENT SUBQUERY	First SELECT in subquery, dependent on outer query
DEPENDENT UNION	Second or later SELECT statement in a UNION, dependent on outer query
DERIVED	Derived table
MATERIALIZED	Materialized subquery
UNCACHEABLE SUBQUERY	A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query
UNCACHEABLE UNION	The second or later select in a UNION that belongs to an uncacheable subquery (see UNCACHEABLE SUBQUERY)

英文描述太简单，不知道说了啥？来详细瞅瞅里边儿的每个值都是干啥吃的：

- SIMPLE

  查询语句中不包含UNION或者子查询的查询都算作是SIMPLE类型，比方说下边这个单表查询的select_type的值就是SIMPLE：

  ```
  mysql> EXPLAIN SELECT * FROM s1;
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
  | id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
  |  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL  |
  ```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

当然，连接查询也算是SIMPLE类型，比如：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL                                               |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | Using join buffer (Block Nested Loop)              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

- PRIMARY

对于包含UNION、UNION ALL或者子查询的大查询来说，它是由几个小查询组成的，其中最左边的那个查询的select_type值就是PRIMARY，比方说：

```
mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL            |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | NULL            |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL | Using temporary |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
3 rows in set, 1 warning (0.00 sec)
```

从结果中可以看到，最左边的小查询SELECT * FROM s1对应的是执行计划中的第一条记录，它的select_type值就是PRIMARY。

- UNION

对于包含UNION或者UNION ALL的大查询来说，它是由几个小查询组成的，其中除了最左边的那个小查询以外，其余的小查询的select_type值就是UNION，可以对比上一个例子的效果，这就不多举例子了。

- UNION RESULT

MySQL选择使用临时表来完成UNION查询的去重工作，针对该临时表的查询的select_type就是UNION RESULT，例子上边有，就不赘述了。

- SUBQUERY

如果包含子查询的查询语句不能够转为对应的semi-join的形式，并且该子查询是不相关子查询，并且查询优化器决定采用将该子查询物化的方案来执行该子查询时，该子查询的第一个SELECT关键字代表的那个查询的select_type就是SUBQUERY，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_key3      | NULL     | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | SUBQUERY    | s2    | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9954 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

可以看到，外层查询的select_type就是PRIMARY，子查询的select_type就是SUBQUERY。需要大家注意的是，由于select_type为SUBQUERY的子查询由于会被物化，所以只需要执行一遍。

- DEPENDENT SUBQUERY

如果包含子查询的查询语句不能够转为对应的semi-join的形式，并且该子查询是相关子查询，则该子查询的第一个SELECT关键字代表的那个查询的select_type就是DEPENDENT SUBQUERY，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE s1.key2 = s2.key2) OR key3 = 'a';
+----+--------------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type        | table | partitions | type | possible_keys | key      | key_len | ref               | rows | filtered | Extra       |
+----+--------------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL  | idx_key3      | NULL     | NULL    | NULL              | 9688 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | ref  | idx_key2,idx_key1 | idx_key2 | 5       | xiaohaizi.s1.key2 |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
2 rows in set, 2 warnings (0.00 sec)
```

需要大家注意的是，select_type为DEPENDENT SUBQUERY的查询可能会被执行多次。

- DEPENDENT UNION

在包含UNION或者UNION ALL的大查询中，如果各个小查询都依赖于外层查询的话，那除了最左边的那个小查询之外，其余的小查询的select_type的值就是DEPENDENT UNION。说的有些绕哈，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE key1 = 'a' UNION SELECT key1 FROM s1 WHERE key1 = 'b');
+----+--------------------+------------+------------+------+---------------+----------+---------+-------+------+----------+-----------------------+
| id | select_type        | table      | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra                 |
+----+--------------------+------------+------------+------+---------------+----------+---------+-------+------+----------+-----------------------+
|  1 | PRIMARY            | s1         | NULL       | ALL  | NULL          | NULL     | NULL    | NULL  | 9688 |   100.00 | Using where           |
|  2 | DEPENDENT SUBQUERY | s2         | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |   12 |   100.00 | Using where; Using index |
|  3 | DEPENDENT UNION    | s1         | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | Using where; Using index |
| NULL | UNION RESULT     | <union2,3> | NULL       | ALL  | NULL          | NULL     | NULL    | NULL  | NULL |     NULL | Using temporary       |
+----+--------------------+------------+------------+------+---------------+----------+---------+-------+------+----------+-----------------------+
4 rows in set, 1 warning (0.03 sec)
```

这个查询比较复杂啊，大查询里包含了一个子查询，子查询里又是由UNION连起来的两个小查询。从执行计划中可以看出来，SELECT key1 FROM s2 WHERE key1 = 'a'这个小查询由于是子查询中第一个查询，所以它的select_type是DEPENDENT SUBQUERY，而SELECT key1 FROM s1 WHERE key1 = 'b'这个查询的select_type就是DEPENDENT UNION。

- DERIVED

对于采用物化的方式执行的包含派生表的查询，该派生表对应的子查询的select_type就是DERIVED，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM (SELECT key1, count(*) as c FROM s1 GROUP BY key1) AS derived_s1 where c > 1;
+----+-------------+------------+------------+------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table      | partitions | type | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+------------+------------+------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | <derived2> | NULL       | ALL  | NULL          | NULL     | NULL    | NULL | 9688 |    33.33 | Using where |
|  2 | DERIVED     | s1         | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9688 |   100.00 | Using index |
+----+-------------+------------+------------+------+---------------+----------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

从执行计划中可以看出，id为2的记录就代表子查询的执行方式，它的select_type是DERIVED，说明该子查询是以物化的方式执行的。id为1的记录代表外层查询，大家注意看它的table列显示的是<derived2>，表示该查询是针对将派生表物化之后的表进行查询的。

> 小贴士：如果派生表可以通过和外层查询合并的方式执行的话，执行计划又是另一番景象，大家可以试试哈～

- MATERIALIZED

当查询优化器在执行包含子查询的语句时，选择将子查询物化之后与外层查询进行连接查询时，该子查询对应的select_type属性就是MATERIALIZED，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2);
+----+--------------+-------------+------------+--------+---------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type  | table       | partitions | type   | possible_keys | key      | key_len | ref               | rows | filtered | Extra       |
+----+--------------+-------------+------------+--------+---------------+----------+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE       | s1          | NULL       | ALL    | idx_key1      | NULL     | NULL    | NULL              | 9688 |   100.00 | Using where |
|  1 | SIMPLE       | <subquery2> | NULL       | eq_ref | <auto_key>    | <auto_key> | 303     | xiaohaizi.s1.key1 |    1 |   100.00 | NULL        |
|  2 | MATERIALIZED | s2          | NULL       | index  | idx_key1      | idx_key1 | 303     | NULL              | 9954 |   100.00 | Using index |
+----+--------------+-------------+------------+--------+---------------+----------+---------+-------------------+------+----------+-------------+
3 rows in set, 1 warning (0.01 sec)
```

执行计划的第三条记录的id值为2，说明该条记录对应的是一个单表查询，从它的select_type值为MATERIALIZED可以看出，查询优化器是要把子查询先转换成物化表。然后看执行计划的前两条记录

的id值都为1，说明这两条记录对应的表进行连接查询，需要注意的是第二条记录的table列的值是<subquery2>，说明该表其实就是id为2对应的子查询执行之后产生的物化表，然后将s1和该物化表进行连接查询。

- UNCACHEABLE SUBQUERY

  不常用，就不多唠叨了。

- UNCACHEABLE UNION

  不常用，就不多唠叨了。

partitions

由于我们压根儿就没唠叨过分区是个啥，所以这个输出列我们也就不说了哈，一般情况下我们的查询语句的执行计划的partitions列的值都是NULL。

type

我们前边说过执行计划的一条记录就代表着MySQL对某个表的执行查询时的访问方法，其中的type列就表明了这个访问方法是个啥，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.04 sec)
```

可以看到type列的值是ref，表明MySQL即将使用ref访问方法来执行对s1表的查询。但是我们之前只唠叨过对使用InnoDB存储引擎的表进行单表访问的一些访问方法，完整的访问方法如下：system，const，eq_ref，ref，fulltext，ref_or_null，index_merge，unique_subquery，index_subquery，range，index，ALL。当然我们还要详细唠叨一下哈：

- system

  当表中只有一条记录并且该表使用的存储引擎的统计数据是精确的，比如MyISAM、Memory，那么对该表的访问方法就是system。比方说我们新建一个MyISAM表，并为其插入一条记录：

  ```
  mysql> CREATE TABLE t(i int) Engine=MyISAM;
  Query OK, 0 rows affected (0.05 sec)

  mysql> INSERT INTO t VALUES(1);
  Query OK, 1 row affected (0.01 sec)
  ```

  然后我们看一下查询这个表的执行计划：

  ```
  mysql> EXPLAIN SELECT * FROM t;
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  | id | select_type | table | partitions | type   | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  |  1 | SIMPLE      | t     | NULL       | system | NULL          | NULL | NULL    | NULL |    1 |   100.00 | NULL  |
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  1 row in set, 1 warning (0.00 sec)
  ```

  可以看到type列的值就是system了。

  小贴士： 你可以把表改成使用InnoDB存储引擎，试试看执行计划的type列是什么。

- const

  这个我们前边唠叨过，就是当我们根据主键或者唯一二级索引列与常数进行等值匹配时，对单表的访问方法就是const，比如：

  ```
  mysql> EXPLAIN SELECT * FROM s1 WHERE id = 5;
  +----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
  | id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
  +----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
  |  1 | SIMPLE      | s1    | NULL       | const | PRIMARY       | PRIMARY | 4       | const |    1 |   100.00 | NULL  |
  +----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
  1 row in set, 1 warning (0.01 sec)
  ```

- eq_ref

  在连接查询时，如果被驱动表是通过主键或者唯一二级索引列等值匹配的方式进行访问的（如果该主键或者唯一二级索引是联合索引的话，所有的索引列都必须进行等值比较），则对该被驱动表的访问方法就是eq_ref，比方说：

  ```
  mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
  +----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
  | id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref             | rows | filtered | Extra |
  +----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
  |  1 | SIMPLE      | s1    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NULL            | 9688 |   100.00 | NULL  |
  |  1 | SIMPLE      | s2    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | xiaohaizi.s1.id |    1 |   100.00 | NULL  |
  +----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
  2 rows in set, 1 warning (0.01 sec)
  ```

  从执行计划的结果中可以看出，MySQL打算将s1作为驱动表，s2作为被驱动表，重点关注s2的访问方法是eq_ref，表明在访问s2表的时候可以通过主键的等值匹配来进行访问。

- ref

  当通过普通的二级索引列与常量进行等值匹配时来查询某个表，那么对该表的访问方法就可能是ref，最开始举过例子了，就不重复举例了。

- fulltext

  全文索引，我们没有细讲过，跳过～

- ref_or_null

  当对普通二级索引进行等值匹配查询，该索引列的值也可以是NULL值时，那么对该表的访问方法就可能是ref_or_null，比如说：

  ```
  mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key1 IS NULL;
  +----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
  | id | select_type | table | partitions | type        | possible_keys | key      | key_len | ref   | rows | filtered | Extra                 |
  +----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
  |  1 | SIMPLE      | s1    | NULL       | ref_or_null | idx_key1      | idx_key1 | 303     | const |    9 |   100.00 | Using index condition |
  +----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
  1 row in set, 1 warning (0.01 sec)
  ```

- index_merge

  一般情况下对于某个表的查询只能使用到一个索引，但我们唠叨单表访问方法时特意强调了在某些场景下可以使用Intersection、Union、Sort-Union这三种索引合并的方式来执行查询，忘掉的回去补一下哈，我们看一下执行计划中是怎么体现MySQL使用索引合并的方式来对某个表执行查询的：

  ```
  mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key3 = 'a';
  +----+-------------+-------+------------+-------------+-----------------------+-----------------------+---------+------+------+----------+-----------------------------------------------------
  | id | select_type | table | partitions | type        | possible_keys         | key                   | key_len | ref  | rows | filtered | Extra
  +----+-------------+-------+------------+-------------+-----------------------+-----------------------+---------+------+------+----------+-----------------------------------------------------
  |  1 | SIMPLE      | s1    | NULL       | index_merge | idx_key1,idx_key3 | idx_key1,idx_key3 | 303,303 | NULL |   14 |   100.00 | Using union(idx_key1,idx_key3); Using where
  +----+-------------+-------+------------+-------------+-----------------------+-----------------------+---------+------+------+----------+-----------------------------------------------------
  1 row in set, 1 warning (0.01 sec)
  ```

  从执行计划的type列的值是index_merge就可以看出，MySQL打算使用索引合并的方式来执行对s1表的查询。

- unique_subquery

  类似于两表连接中被驱动表的eq_ref访问方法，unique_subquery是针对在一些包含IN子查询的查询语句中，如果查询优化器决定将IN子查询转换为EXISTS子查询，而且子查询可以使用到主键进行等值匹配的话，那么该子查询执行计划的type列的值就是unique_subquery，比如下边的这个查询语句：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 IN (SELECT id FROM s2 where s1.key1 = s2.key1) OR key3 = 'a';
+----+--------------------+-------+------------+--------------------+-----------------+---------+---------+------+----------+-------------+
| id | select_type        | table | partitions | type               | possible_keys   | key     | key_len | ref  | rows | filtered | Extra       |
+----+--------------------+-------+------------+--------------------+-----------------+---------+---------+------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL                | idx_key3        | NULL    | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | unique_subquery    | PRIMARY,idx_key1 | PRIMARY | 4       | func |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+--------------------+-----------------+---------+---------+------+------+----------+-------------+
2 rows in set, 2 warnings (0.00 sec)
```

可以看到执行计划的第二条记录的type值就是unique_subquery，说明在执行子查询时会使用到id列的索引。

- index_subquery

  index_subquery与unique_subquery类似，只不过访问子查询中的表时使用的是普通的索引，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE common_field IN (SELECT key3 FROM s2 where s1.key1 = s2.key1) OR key3 = 'a';
+----+--------------------+-------+------------+----------------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type        | table | partitions | type           | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+--------------------+-------+------------+----------------+---------------+---------+---------+------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL            | idx_key3      | NULL    | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | index_subquery | idx_key1,idx_key3 | idx_key3 | 303     | func |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+----------------+---------------+---------+---------+------+------+----------+-------------+
2 rows in set, 2 warnings (0.01 sec)
```

- range

  如果使用索引获取某些范围区间的记录，那么就可能使用到range访问方法，比如下边的这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN ('a', 'b', 'c');
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |   27 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.01 sec)
```

  或者：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'a' AND key1 < 'b';
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  294 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

- index

  当我们可以使用索引覆盖，但需要扫描全部的索引记录时，该表的访问方法就是index，比如这样：

```
mysql> EXPLAIN SELECT key_part2 FROM s1 WHERE key_part3 = 'a';
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
| id | select_type | table | partitions | type  | possible_keys | key          | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | idx_key_part | 909     | NULL | 9688 |    10.00 | Using where; Using index |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

  上述查询中的搜索列表中只有key_part2一个列，而且搜索条件中也只有key_part3一个列，这两个列又恰好包含在idx_key_part这个索引中，可是搜索条件key_part3不能直接使用该索引进行ref或者range方式的访问，只能扫描整个idx_key_part索引的记录，所以查询计划的type列的值就是index。

  > 小贴士： 再一次强调，对于使用InnoDB存储引擎的表来说，二级索引的记录只包含索引列和主键列的值，而聚簇索引中包含用户定义的全部列以及一些隐藏列，所以扫描二级索引的代价比直接全表扫描，也就是扫描聚簇索引的代价更低一些。

- ALL

  最熟悉的全表扫描，就不多唠叨了，直接看例子：

```
mysql> EXPLAIN SELECT * FROM s1;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

一般来说，这些访问方法按照我们介绍它们的顺序性能依次变差。其中除了All这个访问方法外，其余的访问方法都能用到索引，除了index_merge访问方法外，其余的访问方法都最多只能用到一个索引。

possible_keys和key

在EXPLAIN语句输出的执行计划中，possible_keys列表示在某个查询语句中，对某个表执行单表查询时可能用到的索引有哪些，key列表示实际用到的索引有哪些，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND key3 = 'a';
+----+-------------+-------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1,idx_key3 | idx_key3 | 303     | const |    6 |     2.75 | Using where |
+----+-------------+-------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.01 sec)
```

上述执行计划的possible_keys列的值是idx_key1,idx_key3，表示该查询可能使用到idx_key1,idx_key3两个索引，然后key列的值是idx_key3，表示经过查询优化器计算使用不同索引的成本后，最后决定使用idx_key3来执行查询比较划算。

不过有一点比较特别，就是在使用index访问方法来查询某个表时，possible_keys列是空的，而key列展示的是实际使用到的索引，比如这样：

```
mysql> EXPLAIN SELECT key_part2 FROM s1 WHERE key_part3 = 'a';
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
| id | select_type | table | partitions | type  | possible_keys | key          | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | idx_key_part | 909     | NULL | 9688 |    10.00 | Using where; Using index |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

另外需要注意的一点是，possible_keys列中的值并不是越多越好，可能使用的索引越多，查询优化器计算查询成本时就得花费更长时间，所以如果可以的话，尽量删除那些用不到的索引。

key_len

key_len列表示当优化器决定使用某个索引执行查询时，该索引记录的最大长度，它是由这三个部分构成的：

- 对于使用固定长度类型的索引列来说，它实际占用的存储空间的最大长度就是该固定值，对于指定字符集的变长类型的索引列来说，比如某个索引列的类型是VARCHAR(100)，使用的字符集是utf8，那么该列实际占用的最大存储空间就是100 × 3 = 300个字节。
- 如果该索引列可以存储NULL值，则key_len比不可以存储NULL值时多1个字节。
- 对于变长字段来说，都会有2个字节的空间来存储该变长列的实际长度。

比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 5;
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
```

```
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | PRIMARY       | PRIMARY | 4       | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

由于id列的类型是INT，并且不可以存储NULL值，所以在使用该列的索引时key_len大小就是4。当索引列可以存储NULL值时，比如：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 = 5;
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | idx_key2      | idx_key2 | 5       | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

可以看到key_len列就变成了5，比使用id列的索引时多了1。

对于可变长度的索引列来说，比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

由于key1列的类型是VARCHAR(100)，所以该列实际最多占用的存储空间就是300字节，又因为该列允许存储NULL值，所以key_len需要加1，又因为该列是可变长度列，所以key_len需要加2，所以最后ken_len的值就是303。

有的同学可能有疑问：你在前边唠叨InnoDB行格式的时候不是说，存储变长字段的实际长度不是可能占用1个字节或者2个字节么？为什么现在不管三七二十一都用了2个字节？这里需要强调的一点是，执行计划的生成是在MySQL server层中的功能，并不是针对具体某个存储引擎的功能，设计MySQL的大叔在执行计划中输出key_len列主要是为了让我们区分某个使用联合索引的查询具体用了几个索引列，而不是为了准确的说明针对某个具体存储引擎存储变长字段的实际长度占用的空间到底是占用1个字节还是2个字节。比方说下边这个使用到联合索引idx_key_part的查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key_part1 = 'a';
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key_part  | idx_key_part | 303     | const |   12 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

我们可以从执行计划的key_len列中看到值是303，这意味着MySQL在执行上述查询中只能用到idx_key_part索引的一个索引列，而下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key_part1 = 'a' AND key_part2 = 'b';
+----+-------------+-------+------------+------+---------------+--------------+---------+-------------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len | ref         | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+--------------+---------+-------------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key_part  | idx_key_part | 606     | const,const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+--------------+---------+-------------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

这个查询的执行计划的ken_len列的值是606，说明执行这个查询的时候可以用到联合索引idx_key_part的两个索引列。

ref

当使用索引列等值匹配的条件去执行查询时，也就是在访问方法是const、eq_ref、ref、ref_or_null、unique_subquery、index_subquery其中之一时，ref列展示的就是与索引列作等值匹配的东东是个啥，比如只是一个常数或者是某个列。大家看下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

可以看到ref列的值是const，表明在使用idx_key1索引执行查询时，与key1列作等值匹配的对象是一个常数，当然有时候更复杂一点：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref             | rows | filtered | Extra |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NULL            | 9688 |   100.00 | NULL  |
|  1 | SIMPLE      | s2    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | xiaohaizi.s1.id |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
2 rows in set, 1 warning (0.00 sec)
```

可以看到对被驱动表s2的访问方法是eq_ref，而对应的ref列的值是xiaohaizi.s1.id，这说明在对被驱动表进行访问时会用到PRIMARY索引，也就是聚簇索引与一个列进行等值匹配的条件，于s2表的id作等值匹配的对象就是xiaohaizi.s1.id列（注意这里把数据库名也写出来了）。

有的时候与索引列进行等值匹配的对象是一个函数，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s2.key1 = UPPER(s1.key1);
+----+-------------+-------+------------+------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL     | NULL    | NULL | 9688 |   100.00 | NULL                  |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1      | idx_key1 | 303     | func |    1 |   100.00 | Using index condition |
+----+-------------+-------+------------+------+---------------+----------+---------+------+------+----------+-----------------------+
2 rows in set, 1 warning (0.00 sec)
```

我们看执行计划的第二条记录，可以看到对s2表采用ref访问方法执行查询，然后在查询计划的ref列里输出的是func，说明与s2表的key1列进行等值匹配的对象是一个函数。

rows

如果查询优化器决定使用全表扫描的方式对某个表执行查询时，执行计划的rows列就代表预计需要扫描的行数，如果使用索引来执行查询时，执行计划的rows列就代表预计扫描的索引记录行数。比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  266 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

我们看到执行计划的rows列的值是266，这意味着查询优化器在经过分析使用idx_key1进行查询的成本之后，觉得满足key1 > 'z'这个条件的记录只有266条。

filtered

之前在分析连接查询的成本时提出过一个condition filtering的概念，就是MySQL在计算驱动表扇出时采用的一个策略：

- 如果使用的是全表扫描的方式执行的单表查询，那么计算驱动表扇出时需要估计出满足搜索条件的记录到底有多少条。
- 如果使用的是索引执行的单表扫描，那么计算驱动表扇出的时候需要估计出满足除使用到对应索引的搜索条件外的其他搜索条件的记录有多少条。

比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND common_field = 'a';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------------------+
```

```
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+---------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  266 |    10.00 | Using index condition; Using where |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+---------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

从执行计划的key列中可以看出来，该查询使用idx_key1索引来执行查询，从rows列可以看出满足key1 > 'z'的记录有266条。执行计划的filtered列就代表查询优化器预测在这266条记录中，有多少条记录满足其余的搜索条件，也就是common_field = 'a'这个条件的百分比。此处filtered列的值是10.00，说明查询优化器预测在266条记录中有10.00%的记录满足common_field = 'a'这个条件。

对于单表查询来说，这个filtered列的值没什么意义，我们更关注在连接查询中驱动表对应的执行计划记录的filtered值，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.key1 = s2.key1 WHERE s1.common_field = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref               | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_key1      | NULL     | NULL    | NULL              | 9688 |    10.00 | Using where |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s1.key1 |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

从执行计划中可以看出来，查询优化器打算把s1当作驱动表，s2当作被驱动表。我们可以看到驱动表s1表的执行计划的rows列为9688， filtered列为10.00，这意味着驱动表s1的扇出值就是9688 × 10.00% = 968.8，这说明还要对被驱动表执行大约968次查询。