1.1.1. Explain 执行计划

1.1.1.1. 什么是执行计划

有了慢查询语句后，就要对语句进行分析。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。EXPLAIN 语句来帮助我们查看某个查询语句的具体执行计划，我们需要搞懂 EPLATNEXPLAIN 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。

通过使用 EXPLAIN 关键字可以模拟优化器执行 SQL 查询语句，从而知道 MySQL 是如何处理你的 SQL 语句的。分析查询语句或是表结构的性能瓶颈，总的来说通过 EXPLAIN 我们可以：

- 表的读取顺序
- 数据读取操作的操作类型
- 哪些索引可以使用
- 哪些索引被实际使用
- 表之间的引用
- 每张表有多少行被优化器查询

1.1.1.2. 执行计划的语法

执行计划的语法其实非常简单：在 SQL 查询的前面加上 EXPLAIN 关键字就行。比如：EXPLAIN select * from table1

重点的就是 EXPLAIN 后面你要分析的 SQL 语句

除了以 SELECT 开头的查询语句，其余的 DELETE、INSERT、REPLACE 以及 UPOATE 语句前边都可以加上 EXPLAIN，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 EsxPLAIN 语句的用法。

1.1.1.3. 执行计划详解

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：

explain select * from order_exp;

```
mysql> explain select * from order_exp;
+----+-------------+-----------+------------+------+---------------+------+---------+------+-------+----------+-------+
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra |
+----+-------------+-----------+------------+------+---------------+------+---------+------+-------+----------+-------+
|  1 | SIMPLE      | order_exp | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 10248 |   100.00 | NULL  |
+----+-------------+-----------+------------+------+---------------+------+---------+------+-------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

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

看到这里，是不是一脸懵逼，这是正常的，这里把它们都列出来只是为了描述一个轮廓，随着我们课程的进行，我们会仔细讲解每个列的含义，显示值的含义。

为了方便学习，我们使用范例表 order_exp：

```
CREATE TABLE `order_exp`  (
  `id` bigint(22) NOT NULL AUTO_INCREMENT COMMENT '订单的主键',
  `order_no` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的编号',
  `order_note` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的说明',
  `insert_time` datetime(0) NOT NULL DEFAULT CURRENT_TIMESTAMP(0) ON UPDATE CURRENT_TIMESTAMP(0) COMMENT
  '插入订单的时间',
  `expire_duration` bigint(22) NOT NULL COMMENT '订单的过期时长，单位秒',
  `expire_time` datetime(0) NOT NULL COMMENT '订单的过期时间',
  `order_status` smallint(6) NOT NULL DEFAULT 0 COMMENT
  '订单的状态，0：未支付；1：已支付；-1：已过期，关闭',
  PRIMARY KEY (`id`) USING BTREE,
  UNIQUE INDEX `u_idx_day_status`(`insert_time`, `order_status`, `expire_time`) USING BTREE,
  INDEX `idx_order_no`(`order_no`) USING BTREE,
  INDEX `idx_expire_time`(`expire_time`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 10819 CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT =
Dynamic;
```

这个表在库中有个三个派生表 s1，s2，order_exp_cut，表结构基本一致，有少许差别：

```
CREATE TABLE `order_exp_cut`  (
  `id` bigint(22) NOT NULL AUTO_INCREMENT COMMENT '订单的主键',
  `order_no` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT NULL COMMENT '订单的编
  `order_note` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的说明',
  `insert_time` datetime(0) NULL DEFAULT CURRENT_TIMESTAMP(0) ON UPDATE CURRENT_TIMESTAMP(0) COMMENT
  '插入订单的时间',
  `expire_duration` bigint(22) NOT NULL COMMENT '订单的过期时长，单位秒',
  `expire_time` datetime(0) NOT NULL COMMENT '订单的过期时间',
  `order_status` smallint(6) NOT NULL DEFAULT 0 COMMENT
  '订单的状态，0：未支付；1：已支付；-1：已过期，关闭',
  PRIMARY KEY (`id`) USING BTREE,
  UNIQUE INDEX `u_idx_day_status`(`insert_time`, `order_status`, `expire_time`) USING BTREE,
  INDEX `idx_order_no`(`order_no`) USING BTREE

CREATE TABLE `s2`  (
  `id` bigint(22) NOT NULL AUTO_INCREMENT COMMENT '订单的主键',
  `order_no` varchar(50) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的编
  `order_note` varchar(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL COMMENT '订单的
  `insert_time` datetime(0) NOT NULL DEFAULT CURRENT_TIMESTAMP(0) ON UPDATE CURRENT_TIMESTAMP(
  '插入订单的时间',
  `expire_duration` bigint(22) NOT NULL COMMENT '订单的过期时长，单位秒',
  `expire_time` datetime(0) NOT NULL COMMENT '订单的过期时间',
  `order_status` smallint(6) NOT NULL DEFAULT 0 COMMENT
  '订单的状态，0：未支付；1：已支付；-1：已过期，关闭',
  PRIMARY KEY (`id`) USING BTREE,
  UNIQUE INDEX `u_idx_day_status`(`insert_time`, `order_status`, `expire_time`) USING BTREE,
  INDEX `idx_order_no`(`order_no`) USING BTREE,
  INDEX `idx_insert_time`(`insert_time`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 10814 CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FOR
Dynamic;
```

注意：为了方便讲述，我们可能会适当调整对列的讲解顺序，不会完全按照 EXPLAIN 语句输出列顺序来讲解。

table

不论我们的查询语句有多复杂，里边包含了多少个表，到最后也是需要对每个表进行单表访问的，MySQL 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。

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+
| id | select_type | table | partitions | type |
+----+-------------+-------+------------+------+
|  1 | SIMPLE      | s2    | NULL       | ALL  |
|  1 | SIMPLE      | s1    | NULL       | ALL  |
+----+-------------+-------+------------+------+
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';
+----+-------------+-------+------------+------+------
| id | select_type | table | partitions | type | possi
+----+-------------+-------+------------+------+------
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_o
+----+-------------+-------+------------+------+------
```

可以看见，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，而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.

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，

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 J0IN s2

ON s1.id = s2.id

WHERE s1.order_status = 0 ;

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:

1、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 id IN (SELECT * FROM s2);

2、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，MySQL 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。

单 SELECT 关键字

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：

EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';

```
mysql> EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';
+----+-------------+-------+------------+------+---------------+-------------
| id | select_type | table | partitions | type | possible_keys | key
+----+-------------+-------+------------+------+---------------+-------------
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_order_no  | idx_order_no
+----+-------------+-------+------------+------+---------------+-------------
1 row in set, 1 warning (0.00 sec)
```

连接查询

对于连接查询来说，一个 SELEOT 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:

EXPLAIN SELECT * FROM s1 INNER JOIN s2;

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---
| id | select_type | table | partitions | type | possible_keys | key  | ke
+----+-------------+-------+------------+------+---------------+------+---
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NU
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NU
+----+-------------+-------+------------+------+---------------+------+---
2 rows in set, 1 warning (0.00 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。

包含子查询

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:

EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2) OR order_no = 'a';

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2) OR order_no = 'a';
+----+-------------+-------+------------+-------+---------------+------------------+
| id | select_type | table | partitions | type  | possible_keys | key              |
+----+-------------+-------+------------+-------+---------------+------------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_order_no  | NULL             |
|  2 | SUBQUERY    | s2    | NULL       | index | PRIMARY       | u_idx_day_status |
+----+-------------+-------+------------+-------+---------------+------------------+
2 rows in set, 1 warning (0.00 sec)
```

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:

EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2 WHERE order_no = 'a');

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2 WHERE order_no = 'a');
+----+-------------+-------+------------+--------+---------------------+--------------+
| id | select_type | table | partitions | type   | possible_keys       | key          |
+----+-------------+-------+------------+--------+---------------------+--------------+
|  1 | SIMPLE      | s2    | NULL       | ref    | PRIMARY,idx_order_no | idx_order_no |
|  1 | SIMPLE      | s1    | NULL       | eq_ref | PRIMARY             | PRIMARY      |
+----+-------------+-------+------------+--------+---------------------+--------------+
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，

包含 UNION 子句

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:

EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;

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

这个语句的执行计划的第三条记录为什么这样？UNION 子句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:

EXPLAIN SELECT * FROM s1 UNION ALL SELECT * FROM s2;

```
mysql> EXPLAIN SELECT * FROM s1 UNION ALL SELECT * FROM s2;
+----+-------------+-------+------------+------+---------------+------+
| id | select_type | table | partitions | type | possible_keys | key  |
+----+-------------+-------+------------+------+---------------+------+
|  1 | PRIMARY     | s1    | NULL       | ALL  | NULL          | NULL |
|  2 | UNION       | s2    | NULL       | ALL  | NULL          | NULL |
+----+-------------+-------+------------+------+---------------+------+
2 rows in set, 1 warning (0.00 sec)
```

select_type 列

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。

MySQL 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，select_type 取值如下：

SIMPLE：简单的 select 查询,不使用 union 及子查询

PRIMARY：最外层的 select 查询

UNION：UNION 中的第二个或随后的 select 查询,不依赖于外部查询的结果集

UNION RESULT：UNION 结果集

SUBQUERY：子查询中的第一个 select 查询,不依赖于外 部查询的结果集

DEPENDENT UNION：UNION 中的第二个或随后的 select 查询,依赖于外部查询的结果集

DEPENDENT SUBQUERY：子查询中的第一个 select 查询,依赖于外部查询的结果集

DERIVED： 用于 from 子句里有子查询的情况。 MySQL 会 递归执行这些子查询，把结果放在临时表里。

MATERIALIZED：物化子查询

UNCACHEABLE SUBQUERY:　结果集不能被缓存的子查询,必须重新为外层查询的每一行进行评估，出现极少。

UNCACHEABLE UNION：UNION 中的第二个或随后的 select 查询,属于不可缓存的子查询，出现极少。

SIMPLE

EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';

简单的 select 查询，查询中不包含子查询或者 UNION

```
mysql> EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';
+----+-------------+-------+------------+------+-----
| id | select_type | table | partitions | type | poss
+----+-------------+-------+------------+------+-----
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_
+----+-------------+-------+------------+------+-----
1 row in set, 1 warning (0.00 sec)
```

连接查询也算是 SIMPLE 类型

EXPLAIN SELECT * FROM s1 INNER JOIN s2;

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+
| id | select_type | table | partitions | type |
+----+-------------+-------+------------+------+
|  1 | SIMPLE      | s2    | NULL       | ALL  |
|  1 | SIMPLE      | s1    | NULL       | ALL  |
+----+-------------+-------+------------+------+
2 rows in set, 1 warning (0.00 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：

EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;

```
mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
+----+--------------+------------+------------+------+---------------+------+-
| id | select_type  | table      | partitions | type | possible_keys | key  |
+----+--------------+------------+------------+------+---------------+------+-
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL
+----+--------------+------------+------------+------+---------------+------+-
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 * FROMN 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，其余的查询的 select_type 值就是 UNION，可以对比上一个例子的效果。

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。

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:

EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2) OR order_no = 'a';

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2) OR order_no = 'a';
+----+-------------+-------+------------+-------+---------------+------------------+
| id | select_type | table | partitions | type  | possible_keys | key              |
+----+-------------+-------+------------+-------+---------------+------------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_order_no  | NULL             |
|  2 | SUBQUERY    | s2    | NULL       | index | PRIMARY       | u_idx_day_status |
+----+-------------+-------+------------+-------+---------------+------------------+
2 rows in set, 1 warning (0.00 sec)
```

可以看到，外层查询的 select_type 就是 PRTIMARY，子查询的 select_type 就是 SUBOUERY。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。

TIPS

上面的说明里出现了几个新名字，这里稍微解释下，以后会有详解讲解。

***semi-join：**半连接优化技术，本质上是把子查询上拉到父查询中，与父查询的表做 join 操作。关键词是“上拉”。对于子查询，其子查询部分相对于父表的每个符合条件的元组，都要把子查询执行一轮。效率低下。用半连接操作优化子查询，是把子查询上拉到父查询中，这样子查询的表和父查询中的表是并列关系，父表的每个符合条件的元组，只需要在子表中找符合条件的元组即可。简单来说，就是通过将子查询上拉对父查询中的数据进行筛选，以使获取到最少量的足以对父查询记录进行筛选的信息就足够了。*

***子查询物化：**子查询的结果通常缓存在内存或临时表中。*

***关联/相关子查询：**子查询的执行依赖于外部查询。多数情况下是子查询的 WHERE 子句中引用了外部查询的表。自然“非关联/相关子查询”的执行则不依赖与外部的查询。*

DEPENDENT UNION、DEPENDENT SUBQUERY

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。比方说下边这个查询:

EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2 WHERE id = 716 UNION SELECT id FROM s1 WHERE id = 718);

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2 WHERE id = 716 UNION SELECT id FROM s1 WHERE id = 718);
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+-------+------
| id | select_type        | table      | partitions | type  | possible_keys | key     | key_len | ref   | rows  | filte
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+-------+------
|  1 | PRIMARY            | s1         | NULL       | ALL   | NULL          | NULL    | NULL    | NULL  | 10664 |   100
|  2 | DEPENDENT SUBQUERY | s2         | NULL       | const | PRIMARY       | PRIMARY | 8       | const |     1 |   100
|  3 | DEPENDENT UNION    | s1         | NULL       | const | PRIMARY       | PRIMARY | 8       | const |     1 |   100
| NULL | UNION RESULT     | <union2,3> | NULL       | ALL   | NULL          | NULL    | NULL    | NULL  |  NULL |
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+-------+------
4 rows in set, 1 warning (0.00 sec)
```

这个查询比较复杂，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 id FROM s2 WHERE id = 716** 这个小查询由于是子查询中第一个查询，所以它的 select_type 是 OEPENDENT SUBOUERY，而 **SELECT id FROM s1 WHERE id = 718** 这个查询的 select_type 就是 DEPENDENT UNION。

是不是很奇怪这条语句并没有依赖外部的查询？MySQL 优化器对 IN 操作符的优化会将 IN 中的非关联子查询优化成一个关联子查询。我们可以在执行上面那个执行计划后，马上执行 show warnings\G，可以看到 MySQL 对 SQL 语句的大致改写情况：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2 WHERE id = 716 UNION SELECT id FROM s1 WHERE id = 718);
+----+--------------------+-----------+------------+-------+---------------+---------+---------+------+-------+----------+-----------------+
| id | select_type        | table     | partitions | type  | possible_keys | key     | key_len | ref  | rows  | filtered | Extra           |
+----+--------------------+-----------+------------+-------+---------------+---------+---------+------+-------+----------+-----------------+
|  1 | PRIMARY            | s1        | NULL       | ALL   | NULL          | NULL    | NULL    | NULL | 10664 |   100.00 | Using where     |
|  2 | DEPENDENT SUBQUERY | s2        | NULL       | const | PRIMARY       | PRIMARY | 8       | const |     1 |   100.00 | Using index     |
|  3 | DEPENDENT UNION    | s1        | NULL       | const | PRIMARY       | PRIMARY | 8       | const |     1 |   100.00 | Using index     |
| NULL | UNION RESULT     | <union2,3> | NULL       | ALL   | NULL          | NULL    | NULL    | NULL |  NULL |     NULL | Using temporary |
+----+--------------------+-----------+------------+-------+---------------+---------+---------+------+-------+----------+-----------------+
4 rows in set, 1 warning (0.00 sec)

mysql> show warnings\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `mysqladv`.`s1`.`id` AS `id`,`mysqladv`.`s1`.`order_no` AS `order_no`,`mysqladv`.`s1`.`order_note` AS `order_note`,`mysqladv`.`s1`.`in
sert_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`.`id`,<exists>(/* select#2 */ select 1 from `mysqladv`.`s2` where ((<cache>(`mysqladv`.`s1`.`i
d`) = 716)) union /* select#3 */ select 1 from `mysqladv`.`s1` where ((<cache>(`mysqladv`.`s1`.`id`) = 718))))
1 row in set (0.00 sec)
```

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。

EXPLAIN SELECT * FROM (SELECT id, count(*) as c FROM s1 GROUP BY id) AS derived_s1 where c >1;

```
mysql> EXPLAIN SELECT * FROM (SELECT id, count(*) as c FROM s1 GROUP BY id) AS derived_s1 where c >1;
+----+-------------+------------+------------+-------+---------------------------------------+---------
| id | select_type | table      | partitions | type  | possible_keys                         | key
+----+-------------+------------+------------+-------+---------------------------------------+---------
|  1 | PRIMARY     | <derived2> | NULL       | ALL   | NULL                                  | NULL
|  2 | DERIVED     | s1         | NULL       | index | PRIMARY,u_idx_day_status,idx_order_no | PRIMARY
+----+-------------+------------+------------+-------+---------------------------------------+---------
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。

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：

EXPLAIN SELECT * FROM s1 WHERE order_no IN (SELECT order_no FROM s2);

```
mysql> EXPLAIN SELECT * FROM s1 WHERE order_no IN (SELECT order_no FROM s2);
+----+--------------+--------------+------------+--------+---------------+-----
| id | select_type  | table        | partitions | type   | possible_keys | key
+----+--------------+--------------+------------+--------+---------------+-----
|  1 | SIMPLE       | s1           | NULL       | ALL    | idx_order_no  | NULL
|  1 | SIMPLE       | <subquery2>  | NULL       | eq_ref | <auto_key>    | <aut
|  2 | MATERIALIZED | s2           | NULL       | index  | idx_order_no  | idx_
+----+--------------+--------------+------------+--------+---------------+-----
3 rows in set, 1 warning (0.00 sec)
```

执行计划的第三条记录的 id 值为 2，从它的 select_type 值为 NATERIALIED 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 iad 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、UNCACHEABLE UNION

出现极少，不做深入讲解，比如

explain select * from s1 where id = (select id from s2 where order_no=@@sql_log_bin);

```
mysql>  explain select * from s1 where id = ( select id from s2 where order_no=@@sql_log_bin);
+----+----------------------+-------+------------+-------+---------------+-------------+-----
| id | select_type          | table | partitions | type  | possible_keys | key         | key_
+----+----------------------+-------+------------+-------+---------------+-------------+-----
|  1 | PRIMARY              | NULL  | NULL       | NULL  | NULL          | NULL        | NULL
|  2 | UNCACHEABLE SUBQUERY | s2    | NULL       | index | idx_order_no  | idx_order_no | 152
+----+----------------------+-------+------------+-------+---------------+-------------+-----
2 rows in set, 10565 warnings (0.00 sec)
```

partitions

和分区表有关，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/访问类型，其中的 type 列就表明了这个访问方法/访问类型是个什么东西，是较为重要的一个指标，结果值从最好到最坏依次是：

system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL

出现比较多的是 system>const>eq_ref>ref>range>index>ALL

一般来说，得保证查询至少达到 range 级别，最好能达到 ref。

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。

explain select * from test_myisam;

```
mysql> explain select * from test_myisam;
+----+-------------+-------------+------------+--------+
| id | select_type | table       | partitions | type   |
+----+-------------+-------------+------------+--------+
|  1 | SIMPLE      | test_myisam | NULL       | system |
+----+-------------+-------------+------------+--------+
1 row in set, 1 warning (0.00 sec)
```

当然，如果改成使用 InnoDB 存储引擎，试试看执行计划的 type 列的值是什么。

```
mysql> explain select * from test;
+----+-------------+-------+------------+------+
| id | select_type | table | partitions | type |
+----+-------------+-------+------------+------+
|  1 | SIMPLE      | test  | NULL       | ALL  |
+----+-------------+-------+------------+------+
1 row in set, 1 warning (0.00 sec)
```

const

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。因为只匹配一行数据，所以很快。

例如将主键置于 where 列表中

EXPLAIN SELECT * FROM s1 WHERE id = 716;

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 716;
+----+-------------+-------+------------+-------+
| id | select_type | table | partitions | type  |
+----+-------------+-------+------------+-------+
|  1 | SIMPLE      | s1    | NULL       | const |
+----+-------------+-------+------------+-------+
1 row in set, 1 warning (0.00 sec)
```

B+树叶子节点中的记录是按照索引列排序的，对于的聚簇索引来说，它对应的 B+树叶子节点中的记录就是按照 id 列排序的。B+树矮胖，所以这样根据主键值定位一条记录的速度很快。类似的，我们根据唯一二级索引列来定位一条记录的速度也很快的，比如下边这个查询：

SELECT * FROM order_exp WHERE insert_time='' and order_status='' and expire_time='' ;

这个查询的执行分两步，第一步先从 u_idx_day_status 对应的 B+树索引中根据索引列与常数的等值比较条件定位到一条二级索引记录，然后再根据该记录的 id 值到聚簇索引中获取到完整的用户记录。

MySQL 把这种通过主键或者唯一二级索引列来定位一条记录的访问方法定义为：const，意思是常数级别的，代价是可以忽略不计的。

不过这种 const 访问方法只能在主键列或者唯一二级索引列和一个常数进行等值比较时才有效，如果主键或者唯一二级索引是由多个列构成的话，组成索引的每一个列都是与常数进行等值比较时，这个 const 访问方法才有效。

对于唯一二级索引来说，查询该列为 NULL 值的情况比较特殊，因为唯一二级索引列并不限制 NULL 值的数量，所以上述语句可能访问到多条记录，也就是说 is null 不可以使用 const 访问方法来执行。

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。

TIPS

驱动表与被驱动表：*A 表和 B 表 join 连接查询，如果通过 A 表的结果集作为循环基础数据，然后一条一条地通过该结果集中的数据作为过滤条件到 B 表中查询数据，然后合并结果。那么我们称 A 表为驱动表，B 表为被驱动表*

比方说：

EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
+----+-------------+-------+------------+--------+-------------
| id | select_type | table | partitions | type   | possible_key
+----+-------------+-------+------------+--------+-------------
|  1 | SIMPLE      | s2    | NULL       | ALL    | PRIMARY
|  1 | SIMPLE      | s1    | NULL       | eq_ref | PRIMARY
+----+-------------+-------+------------+--------+-------------
2 rows in set, 1 warning (0.00 sec)
```

从执行计划的结果中可以看出，MySQL 打算将 s2 作为驱动表，s1 作为被驱动表，重点关注 s1 的访问方法是 eq_ref，表明在访问 s1 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。

本质上也是一种索引访问，它返回所有匹配某个单独值的行，然而，它可能会找到多个符合条件的行，所以他属于查找和扫描的混合体

EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';
+----+-------------+-------+------------+------+--------
| id | select_type | table | partitions | type | possibl
+----+-------------+-------+------------+------+--------
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_orc
+----+-------------+-------+------------+------+--------
```

对于这个查询，我们当然可以选择全表扫描来逐一对比搜索条件是否满足要求，我们也可以先使用二级索引找到对应记录的 id 值，然后再回表到聚簇索引中查找完整的用户记录。

由于普通二级索引并不限制索引列值的唯一性，所以可能找到多条对应的记录，也就是说使用二级索引来执行查询的代价取决于等值匹配到的二级索引记录条数。如果匹配的记录较少，则回表的代价还是比较低的，所以 MySQL 可能选择使用索引而不是全表扫描的方式来执行查询。这种搜索条件为二级索引列与常数等值比较，采用二级索引来执行查询的访问方法称为：ref。

对于普通的二级索引来说，通过索引列进行等值比较后可能匹配到多条连续的记录，而不是像主键或者唯一二级索引那样最多只能匹配 1 条记录，所以这种 ref 访问方法比 const 要差些，但是在二级索引等值比较时匹配的记录数较少时的效率还是很高的（如果匹配的二级索引记录太多那么回表的成本就太大了）。不过需要注意下边两种情况：

二级索引列值为 NULL 的情况

不论是普通的二级索引，还是唯一二级索引，它们的索引列对包含 NULL 值的数量并不限制，所以我们采用 key IS NULL 这种形式的搜索条件最多只能使用 ref 的访问方法，而不是 const 的访问方法。

对于某个包含多个索引列的二级索引来说，只要是最左边的连续索引列是与常数的等值比较就可能采用 ref 的访问方法，比方说下边这几个查询：

SELECT * FROM order_exp WHERE insert_time = '2021-03-22 18:28:23';

SELECT * FROM order_exp WHERE insert_time = '2021-03-22 18:28:23' AND order_status = 0;

但是如果最左边的连续索引列并不全部是等值比较的话，它的访问方法就不能称为 ref 了，比方说这样：

SELECT * FROM order_exp WHEREinsert_time = '2021-03-22 18:28:23' AND order_status > -1;

fulltext

全文索引，跳过 ~ 。

ref_or_null

有时候我们不仅想找出某个二级索引列的值等于某个常数的记录，还想把该列的值为 NULL 的记录也找出来，就像下边这个查询：

explain SELECT * FROM order_exp_cut WHERE order_no= 'abc' OR order_no IS NULL;

```
mysql> explain SELECT * FROM order_exp_cut WHERE order_no= 'abc' OR order_no IS NULL;
+----+-------------+---------------+------------+-------------+---------------+--------
----------+
| id | select_type | table         | partitions | type        | possible_keys | key
          |
+----+-------------+---------------+------------+-------------+---------------+--------
----------+
|  1 | SIMPLE      | order_exp_cut | NULL       | ref_or_null | idx_order_no  | idx_or
condition |
+----+-------------+---------------+------------+-------------+---------------+--------
----------+
1 row in set, 1 warning (0.00 sec)
```

注意，上面的表改为了 order_exp_cut，order_exp_cut 相对于 order_exp 就是把一些列改为了允许 null，其他的无变化。

id	bigint	22	0	☑	☐	1	订单的主键
order_no	varchar	50	0	☐	☐		订单的编号
order_note	varchar	100	0	☑	☐		订单的说明
insert_time	datetime	0	0	☐	☐		插入订单的
expire_duration	bigint	22	0	☑	☐		订单的过期
expire_time	datetime	0	0	☑	☐		订单的过期
order_status	smallint	6	0	☑	☐		订单的状态

这个查询相当于先分别从 order_exp_cut 表的 idx_order_no 索引对应的 B+树中找出 order_no IS NULL 和 order_no= 'abc'的两个连续的记录范围，然后根据这些二级索引记录中的 id 值再回表查找完整的用户记录。

index_merge

一般情况下对于某个表的查询只能使用到一个索引，在某些场景下可以使用索引合并的方式来执行查询：

EXPLAIN SELECT * FROM s1 WHERE order_no = 'a' OR insert_time = '2021-03-22 18:36:47';

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE order_no = 'a' OR insert_time = '2021-03-22 18:36:47';
+----+-------------+-------+------------+-------------+-----------------------------------------------+
---+-----------------------------------------------------------+
| id | select_type | table | partitions | type        | possible_keys                                 |
ed | Extra                                                     |
+----+-------------+-------+------------+-------------+-----------------------------------------------+
---+-----------------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | index_merge | u_idx_day_status,idx_order_no,idx_insert_time |
00 | Using union(idx_order_no,idx_insert_time); Using where |
+----+-------------+-------+------------+-------------+-----------------------------------------------+
---+-----------------------------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

什么是索引合并，我们以后再说。

unique_subquery

类似于两表连接中被驱动表的 eg_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:

EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2 where s1.insert_time = s2.insert_time) OR order_no = 'a';

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE id IN (SELECT id FROM s2 where s1.insert_time = s2.insert_time) OR order_no = 'a';
+----+--------------------+-------+------------+-----------------+-----------------------------------------+---------+---
---+
| id | select_type        | table | partitions | type            | possible_keys                           | key     | ke
   |
+----+--------------------+-------+------------+-----------------+-----------------------------------------+---------+---
---+
|  1 | PRIMARY            | s1    | NULL       | ALL             | idx_order_no                            | NULL    | NU
re |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | unique_subquery | PRIMARY,u_idx_day_status,idx_insert_time | PRIMARY | 8
re |
+----+--------------------+-------+------------+-----------------+-----------------------------------------+---------+---
---+
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：

EXPLAIN SELECT * FROM s1 WHERE order_no IN (SELECT order_no FROM s2 where s1.insert_time = s2.insert_time) OR order_no = 'a';

这个语句和 unique_subquery 章节中的唯一不同是什么？就是 in 子句的字段由 id 变成了 order_no。

```
mysql> EXPLAIN SELECT * FROM s1 WHERE order_no IN (SELECT order_no FROM
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: s1
   partitions: NULL
         type: ALL
possible_keys: idx_order_no
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 10573
     filtered: 100.00
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: s2
   partitions: NULL
         type: index_subquery
possible_keys: u_idx_day_status,idx_order_no,idx_insert_time
          key: idx_order_no
      key_len: 152
          ref: func
         rows: 1
     filtered: 10.00
        Extra: Using where
2 rows in set, 2 warnings (0.00 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，一般就是在你的 where 语句中出现了 between、<、>、in 等的查询。

这种范围扫描索引扫描比全表扫描要好，因为它只需要开始于索引的某一点，而结束语另一点，不用扫描全部索引。

EXPLAIN SELECT * FROM s1 WHERE order_no IN ('a', 'b', 'c');

EXPLAIN SELECT * FROM s1 WHERE order_no > 'a' AND order_no < 'b';

```
mysql> EXPLAIN SELECT * FROM s1 WHERE order_no IN ('a', 'b', 'c');
+----+-------------+-------+------------+-------+---------------+-----
| id | select_type | table | partitions | type  | possible_keys | key
+----+-------------+-------+------------+-------+---------------+-----
|  1 | SIMPLE      | s1    | NULL       | range | idx_order_no  | idx_
+----+-------------+-------+------------+-------+---------------+-----
1 row in set, 1 warning (0.01 sec)

mysql> EXPLAIN SELECT * FROM s1 WHERE order_no > 'a' AND order_no < 'b
+----+-------------+-------+------------+-------+---------------+-----
| id | select_type | table | partitions | type  | possible_keys | key
+----+-------------+-------+------------+-------+---------------+-----
|  1 | SIMPLE      | s1    | NULL       | range | idx_order_no  | idx_
+----+-------------+-------+------------+-------+---------------+-----
1 row in set, 1 warning (0.00 sec)
```

这种利用索引进行范围匹配的访问方法称之为：range。

此处所说的使用索引进行范围匹配中的 `索引` 可以是聚簇索引，也可以是二级索引。

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。

EXPLAIN SELECT insert_time FROM s1 WHERE expire_time = '2021-03-22 18:36:47';

```
mysql> EXPLAIN SELECT insert_time FROM s1 WHERE expire_time = '2021-03-22 18:36:47';
+----+-------------+-------+------------+-------+---------------+------------------+
| id | select_type | table | partitions | type  | possible_keys | key              |
+----+-------------+-------+------------+-------+---------------+------------------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | u_idx_day_status |
+----+-------------+-------+------------+-------+---------------+------------------+
1 row in set, 1 warning (0.00 sec)
```

all

最熟悉的全表扫描，将遍历全表以找到匹配的行

EXPLAIN SELECT * FROM s1;

```
mysql> EXPLAIN SELECT * FROM s1;
+----+-------------+-------+------------+------+
| id | select_type | table | partitions | type |
+----+-------------+-------+------------+------+
|  1 | SIMPLE      | s1    | NULL       | ALL  |
+----+-------------+-------+------------+------+
1 row in set, 1 warning (0.00 sec)
```

possible_keys 与 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，如果为 NULL，则没有使用索引。比方说下边这个查询:。

EXPLAIN SELECT order_note FROM s1 WHERE insert_time = '2021-03-22 18:36:47';

```
mysql> EXPLAIN SELECT order_note FROM s1 WHERE insert_time = '2021-03-22 18:36:47';
+----+-------------+-------+------------+------+-------------------------------+------------------+
| id | select_type | table | partitions | type | possible_keys                 | key              |
+----+-------------+-------+------------+------+-------------------------------+------------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | u_idx_day_status,idx_insert_time | u_idx_day_status |
+----+-------------+-------+------------+------+-------------------------------+------------------+
1 row in set, 1 warning (0.00 sec)
```

上述执行计划的 possible keys 列的值表示该查询可能使用到 u_idx_day_status,idx_insert_time 两个索引，然后 key 列的值是 u_idx_day_status，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 u_idx_day_status 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，可能会出现 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:

EXPLAIN SELECT insert_time FROM s1 WHERE expire_time = '2021-03-22 18:36:47';

具体原因请自行思考。

```
mysql> EXPLAIN SELECT insert_time FROM s1 WHERE expire_time = '2021-03-22 18:36:47';
+----+-------------+-------+------------+-------+---------------+------------------+
| id | select_type | table | partitions | type  | possible_keys | key              |
+----+-------------+-------+------------+-------+---------------+------------------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | u_idx_day_status |
+----+-------------+-------+------------+-------+---------------+------------------+
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，计算方式是这样的：

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 x 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：

EXPLAIN SELECT * FROM s1 WHERE id = 718;

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 718;
+----+-------------+-------+------------+-------+---------------+---------+---------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len |
+----+-------------+-------+------------+-------+---------------+---------+---------+
|  1 | SIMPLE      | s1    | NULL       | const | PRIMARY       | PRIMARY | 8       |
+----+-------------+-------+------------+-------+---------------+---------+---------+
1 row in set, 1 warning (0.00 sec)
```

由于 id 列的类型是 bigint，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 8。

对于可变长度的索引列来说，比如下边这个查询：

EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';

```
mysql> EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';
+----+-------------+-------+------------+------+---------------+--------------+---------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len |
+----+-------------+-------+------------+------+---------------+--------------+---------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_order_no  | idx_order_no | 152     |
+----+-------------+-------+------------+------+---------------+--------------+---------+
```

由于 order_no 列的类型是 VARCHAR(50)，所以该列实际最多占用的存储空间就是 50*3 字节，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 152。

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，MySQL 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(复合索引有最左前缀的特性，如果复合索引能全部使用上，则是复合索引字段的索引长度之和，这也可以用来判定复合索引是否部分使用，还是全部使用)，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。

Key_len 表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。在不损失精确性的情况下，长度越短越好

key_len 显示的值为索引字段的最大可能长度，并非实际使用长度，即 key_len 是根据表定义计算而得，不是通过表内检索出的。

注意：char 和 varchar 跟字符编码也有密切的联系，比如 latin1 占用 1 个字节，gbk 占用 2 个字节，utf8 占用 3 个字节。

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、eg_ref、ref、ref_or_null、unique_sutbquery、index_subopery 其中之一时,ref 列展示的就是与索引列作等值匹配的是谁，比如只是一个常数或者是某个列。比如：

EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE order_no = 'a';
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len | ref   |
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_order_no  | idx_order_no | 152     | const |
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+
1 row in set, 1 warning (0.00 sec)
```

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

EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;

```
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             |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+
|  1 | SIMPLE      | s2    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NULL            |
|  1 | SIMPLE      | s1    | NULL       | eq_ref | PRIMARY       | PRIMARY | 8       | mysqladv.s2.id |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+
```

可以看到对被驱动表 s2 的访问方法是 eg_ref，而对应的 ref 列的值是 mysqladv.s2.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，与 s2 表的 id 作等值匹配的对象就是 mysqladv.s2.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，比如

EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s2.order_no= UPPER(s1.order_no);

可以看到在查询计划的 ref 列里输出的是 func。

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s2.order_no= UPPER(s1.order_no);
+----+-------------+-------+------------+------+---------------+--------------+---------+------+-------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len | ref  | rows  |
+----+-------------+-------+------------+------+---------------+--------------+---------+------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL         | NULL    | NULL | 10573 |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_order_no  | idx_order_no | 152     | func |     1 |
+----+-------------+-------+------------+------+---------------+--------------+---------+------+-------+
2 rows in set, 1 warning (0.01 sec)
```

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。比如下边两个个查询:

EXPLAIN SELECT * FROM s1 WHERE order_no > 'z';

EXPLAIN SELECT * FROM s1 WHERE order_no > 'a';

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE order_no > 'z';
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key          | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_order_no  | idx_order_no | 152     | NULL |    1 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)

mysql>  EXPLAIN SELECT * FROM s1 WHERE order_no > 'a';
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_order_no  | NULL | NULL    | NULL | 10573 |    50.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

我们看到执行计划的 rows 列的值是分别是 1 和 10573，这意味着查询优化器在经过分析使用 idx_order_no 进行查询的成本之后，觉得满足 order_no> ' a '这个条件的记录只有 1 条，觉得满足 order_no> ' a '这个条件的记录有 10573 条。

filtered

查询优化器预测有多少条记录满足其余的搜索条件，什么意思呢？看具体的语句：

EXPLAIN SELECT * FROM s1 WHERE id > 5890 AND order_note = 'a';

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE order_no > 'r' AND order_note = 'a';
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+
| id | select_type | table | partitions | type  | possible_keys | key          | key_len | ref  | rows | filtered |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_order_no  | idx_order_no | 152     | NULL |    1 |    10.00 |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+
```

从执行计划的 key 列中可以看出来，该查询使用 PRIMARY 索引来执行查询，从 rows 列可以看出满足 id > 5890 的记录有 5286 条。执行计划的 filtered 列就代表查询优化器预测在这 5286 条记录中，有多少条记录满足其余的搜索条件，也就是 order_note = 'a'这个条件的百分比。此处 filtered 列的值是 10.0，说明查询优化器预测在 5286 条记录中有 10.00%的记录满足 order_note = '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:

EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.order_no = s2.order_no WHERE s1.order_note > '你好，李焕英';

```
mysql>  EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.order_no = s2.order_no WHERE s1.order_note > '你好，李焕英';
+----+-------------+-------+------------+------+---------------+--------------+---------+---------------------+-------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len | ref                 | rows  | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+--------------+---------+---------------------+-------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_order_no  | NULL         | NULL    | NULL                | 10573 |    33.33 | Using where |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_order_no  | idx_order_no | 152     | mysqladv.s1.order_no |     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 s 1 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 10573，filtered 列为 33.33 ，这意味着驱动表 s1 的扇出值就是 10573 x 33.33 % = 3524.3，这说明还要对被驱动表执行大约 3524 次查询。

Extra

顾名思义，Extra 列是用来说明一些额外信息的，我们可以通过这些额外信息来更准确的理解 MySQL 到底将如何执行给定的查询语句。MySQL 提供的额外信息很多，几十个，无法一一介绍，挑一些平时常见的或者比较重要的额外信息讲讲，同时因为数据的关系，很难实际演示出来，所以本小节的相关内容不会提供全部实际的 SQL 语句和结果画面。

No tables used

当查询语句的没有 FROM 子句时将会提示该额外信息。

Impossible WHERE

查询语句的 WHERE 子句永远为 FALSE 时将会提示该额外信息。

No matching min/max row

当查询列表处有 MIN 或者 MAX 聚集函数，但是并没有符合 WHERE 子句中的搜索条件的记录时，将会提示该额外信息。

Using index

当我们的查询列表以及搜索条件中只包含属于某个索引的列，也就是在可以**使用索引覆盖的情况**下，在 Extra 列将会提示该额外信息。比方说下边这个查询中只需要用到 idx_order_no 而不需要回表操作：

EXPLAIN SELECT expire_time FROM s1 WHERE insert_time = '2021-03-22 18:36:47';

```
EXPLAIN SELECT expire_time FROM s1 WHERE insert_time = '2021-03-22 18:36:47';
+-----------+-------+-----------+------+-------------------------------+-----------------+---------+-------+------+----------+-------------+
| select_type | table | partitions | type | possible_keys                 | key             | key_len | ref   | rows | filtered | Extra       |
+-----------+-------+-----------+------+-------------------------------+-----------------+---------+-------+------+----------+-------------+
| SIMPLE      | s1    | NULL       | ref  | u_idx_day_status,idx_insert_time | u_idx_day_status | 5       | const |   14 |   100.00 | Using index |
+-----------+-------+-----------+------+-------------------------------+-----------------+---------+-------+------+----------+-------------+
1 set, 1 warning (0.00 sec)
```

Using index condition

有些搜索条件中虽然出现了索引列，但却不能使用到索引，比如下边这个查询：

SELECT * FROM s1 WHERE order_no > 'z' AND order_no LIKE '%a';

其中的 order_no> 'z'可以使用到索引，但是 order_no LIKE '%a'却无法使用到索引，在以前版本的 MySQL 中，是按照下边步骤来执行这个查询的：

1、先根据 order_no> 'z'这个条件，从二级索引 idx_order_no 中获取到对应的二级索引记录。

2、根据上一步骤得到的二级索引记录中的主键值进行回表(因为是 select *)，找到完整的用户记录再检测该记录是否符合 key1 LIKE '%a'这个条件，将符合条件的记录加入到最后的结果集。

但是虽然 order_no LIKE '%a'不能组成范围区间参与 range 访问方法的执行，但这个条件毕竟只涉及到了 order_no 列，MySQL 把上边的步骤改进了一下。

索引条件下推

1、先根据 order_no> 'z'这个条件，定位到二级索引 idx_order_no 中对应的二级索引记录。

2、对于指定的二级索引记录，先不着急回表，而是先检测一下该记录是否满足 order_no LIKE '%a'这个条件，如果这个条件不满足，则该二级索引记录压根儿就没必要回表。

3、对于满足 order_no LIKE '%a'这个条件的二级索引记录执行回表操作。

我们说回表操作其实是一个随机 IO，比较耗时，所以上述修改可以省去很多回表操作的成本。这个改进称之为索引条件下推（英文名：ICP ，Index Condition Pushdown）。

如果在查询语句的执行过程中将要使用索引条件下推这个特性，在 Extra 列中将会显示 Using index condition，比如这样：

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE order_no > 'z' AND order_no LIKE '%a';
+----+-------------+-------+-----------+-------+---------------+-------------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key         | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+-----------+-------+---------------+-------------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_order_no  | idx_order_no | 152     | NULL |    1 |   100.00 | Using index condition |
+----+-------------+-------+-----------+-------+---------------+-------------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

Using where

当我们使用全表扫描来执行对某个表的查询，并且该语句的 WHERE 子句中有针对该表的搜索条件时，在 Extra 列中会提示上述额外信息。

EXPLAIN SELECT * FROM s1 WHERE order_note = 'a';

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE order_note = 'a';
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 10573 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

当使用索引访问来执行对某个表的查询，并且该语句的 WHERE 子句中有除了该索引包含的列之外的其他搜索条件时，在 Extra 列中也会提示上述信息。

比如下边这个查询虽然使用 idx_order_no 索引执行查询，但是搜索条件中除了包含 order_no 的搜索条件 order_no = 'a'，还有包含 order_note 的搜索条件，此时需要回表检索记录然后进行条件判断，所以 Extra 列会显示 Using where 的提示：

EXPLAIN SELECT * FROM s1 WHERE order_no = 'a' AND order_note = 'a';

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE order_no = 'a' AND order_note = 'a';
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_order_no  | idx_order_no | 152     | const |    1 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+--------------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

但是大家注意：出现了 Using where，只是表示在 server 层根据 where 条件进行了过滤，和是否全表扫描或读取了索引文件没有关系，网上有不少文章把 Using where 和是否读取索引进行关联，是不正确的，也有文章把 Using where 和回表进行了关联，这也是不对的。按照 MySQL 官方的说明：

https://dev.mysql.com/doc/refman/5.7/en/explain-output.html#explain_extra

> `Using where` (JSON property: `attached_condition`)
>
> A `WHERE` clause is used to restrict which rows to match against the next table or send to the client.

意思是：Extra 列中出现了 Using where 代表 WHERE 子句用于限制要与下一个表匹配或发送给客户端的行。

很明显，Using where 只是表示 MySQL 使用 where 子句中的条件对记录进行了过滤。

Using join buffer (Block Nested Loop)

在连接查询执行过程中，当被驱动表不能有效的利用索引加快访问速度，MySQL 一般会为其分配一块名叫 join buffer 的内存块来加快查询速度：

EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.order_note = s2.order_note;

```
mysql>  EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.order_note = s2.order_note;
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 10315 |   100.00 | NULL                                               |
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 10573 |    10.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

我们看到，在对 s1 表的执行计划的 Extra 列显示了两个提示：

Using join buffer (Block Nested Loop)：这是因为对表 s1 的访问不能有效利用索引，只好退而求其次，使用 join buffer 来减少对 s1 表的访问次数，从而提高性能。

Using where：可以看到查询语句中有一个 s1.order_note = s2.order_note 条件，因为 s2 是驱动表，s1 是被驱动表，所以在访问 s1 表时，s1.order_note 的值已经确定下来了，所以实际上查询 s1 表的条件就是 s1.order_note = 一个常数，所以提示了 Using where 额外信息。

Not exists

当我们使用左 (外) 连接时，如果 WHERE 子句中包含要求被驱动表的某个列等于 NULL 值的搜索条件，而且那个列又是不允许存储 NULL 值的，那么在该表的执行计划的 Extra 列

就会提示 Not exists 额外信息，比如这样：

EXPLAIN SELECT * FROM s1 LEFT JOIN s2 ON s1.order_no = s2.order_no WHERE s2.id IS NULL;

```
mysql>  EXPLAIN SELECT * FROM s1 LEFT JOIN s2 ON s1.order_no = s2.order_no WHERE s2.id IS NULL;
+----+-------------+-------+------------+------+---------------+--------------+---------+---------------------+-------+----------+--------------------------+
| id | select_type | table | partitions | type | possible_keys | key          | key_len | ref                 | rows  | filtered | Extra                    |
+----+-------------+-------+------------+------+---------------+--------------+---------+---------------------+-------+----------+--------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL         | NULL    | NULL                | 10573 |   100.00 | NULL                     |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_order_no  | idx_order_no | 152     | mysqladv.s1.order_no |     1 |    10.00 | Using where; Not exists |
+----+-------------+-------+------------+------+---------------+--------------+---------+---------------------+-------+----------+--------------------------+
2 rows in set, 1 warning (0.00 sec)
```

上述查询中 s1 表是驱动表，s2 表是被驱动表，s2.id 列是主键而且不允许存储 NULL 值的，而 WHERE 子句中又包含 s2.id IS NULL 的搜索条件。

Using intersect(...)、Using union(...)和 Using sort_union(...)

如果执行计划的 Extra 列出现了 Using intersect(...)提示，说明准备使用 Intersect 索引合并的方式执行查询，括号中的...表示需要进行索引合并的索引名称；如果出现了 Using union(...)提示，说明准备使用 Union 索引合并的方式执行查询；出现了 Using sort_union(...)提示，说明准备使用 Sort-Union 索引合并的方式执行查询。什么是索引合并，我们后面会单独讲。

Zero limit

当我们的 LIMIT 子句的参数为 0 时，表示压根儿不打算从表中读出任何记录，将会提示该额外信息。

Using filesort

有一些情况下对结果集中的记录进行排序是可以使用到索引的，比如下边这个查询：

EXPLAIN SELECT * FROM s1 ORDER BY order_no LIMIT 10;

```
mysql>  EXPLAIN SELECT * FROM s1 ORDER BY order_no LIMIT 10;
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key          | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | idx_order_no | 152     | NULL |   10 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+--------------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

这个查询语句可以利用 idx_order_no 索引直接取出 order_no 列的 10 条记录，然后再进行回表操作就好了。但是很多情况下排序操作无法使用到索引，只能在内存中（记录较少的时候）或者磁盘中（记录较多的时候）进行排序，MySQL 把这种在内存中或者磁盘上进行排序的方式统称为文件排序。如果某个查询需要使用文件排序的方式执行查询，就会在执行计划的 Extra 列中显示 Using filesort 提示：

EXPLAIN SELECT * FROM s1 ORDER BY order_note LIMIT 10;

```
mysql>  EXPLAIN SELECT * FROM s1 ORDER BY order_note LIMIT 10;
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 10573 |   100.00 | Using filesort |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+----------------+
1 row in set, 1 warning (0.00 sec)
```

需要注意的是，如果查询中需要使用 filesort 的方式进行排序的记录非常多，那么这个过程是很耗费性能的，我们最好想办法将使用文件排序的执行方式改为使用索引进行排序。

Using temporary

在许多查询的执行过程中，MySQL 可能会借助临时表来完成一些功能，比如去重、排

序之类的，比如我们在执行许多包含 DISTINCT、GROUP BY、UNION 等子句的查询过程中，如果不能有效利用索引来完成查询，MySQL 很有可能寻求通过建立内部的临时表来执行查询。如果查询中使用到了内部的临时表，在执行计划的 Extra 列将会显示 Using temporary 提示：

EXPLAIN SELECT DISTINCT order_note FROM s1;

```
mysql>  EXPLAIN SELECT DISTINCT order_note 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 | 10573 |   100.00 | Using temporary |
+----+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-----------------+
1 row in set, 1 warning (0.00 sec)
```

再比如：

EXPLAIN SELECT order_note, COUNT(*) AS amount FROM s1 GROUP BY order_note;

```
EXPLAIN SELECT order_note, COUNT(*) AS amount FROM s1 GROUP BY order_note;
+-----------+-------+------------+------+---------------+------+---------+------+-------+----------+-----------------------------+
elect_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra                       |
+-----------+-------+------------+------+---------------+------+---------+------+-------+----------+-----------------------------+
IMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 10573 |   100.00 | Using temporary; Using filesort |
+-----------+-------+------------+------+---------------+------+---------+------+-------+----------+-----------------------------+
 set, 1 warning (0.00 sec)
```

上述执行计划的 Extra 列不仅仅包含 Using temporary 提示，还包含 Using filesort 提示，可是我们的查询语句中明明没有写 ORDER BY 子句呀？这是因为 MySQL 会在包含 GROUP BY 子句的查询中默认添加上 ORDER BY 子句，也就是说上述查询其实和下边这个查询等价：

EXPLAIN SELECT order_note, COUNT(*) AS amount FROM s1 GROUP BY order_note order by order_note;

如果我们并不想为包含 GROUP BY 子句的查询进行排序，需要我们显式的写上 ORDER BY NULL：

EXPLAIN SELECT order_note, COUNT(*) AS amount FROM s1 GROUP BY order_note order by null;

```
EXPLAIN SELECT order_note, COUNT(*) AS amount FROM s1 GROUP BY order_note order by null;
+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-----------------+
select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows  | filtered | Extra           |
+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-----------------+
SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 10573 |   100.00 | Using temporary |
+-------------+-------+------------+------+---------------+------+---------+------+-------+----------+-----------------+
n set, 1 warning (0.01 sec)
```

这回执行计划中就没有 Using filesort 的提示了，也就意味着执行查询时可以省去对记录进行文件排序的成本了。

很明显，执行计划中出现 Using temporary 并不是一个好的征兆，因为建立与维护临时表要付出很大成本的，所以我们最好能使用索引来替代掉使用临时表，比方说下边这个包含 GROUP BY 子句的查询就不需要使用临时表：

EXPLAIN SELECT order_no, COUNT(*) AS amount FROM s1 GROUP BY order_no;

```
EXPLAIN SELECT order_no, COUNT(*) AS amount FROM s1 GROUP BY order_no;
+-----------+-------+------------+-------+---------------+--------------+---------+------+-------+----------+-------------+
elect_type | table | partitions | type  | possible_keys | key          | key_len | ref  | rows  | filtered | Extra       |
+-----------+-------+------------+-------+---------------+--------------+---------+------+-------+----------+-------------+
IMPLE      | s1    | NULL       | index | idx_order_no  | idx_order_no | 152     | NULL | 10573 |   100.00 | Using index |
+-----------+-------+------------+-------+---------------+--------------+---------+------+-------+----------+-------------+
 set, 1 warning (0.00 sec)
```

从 Extra 的 Using index 的提示里我们可以看出，上述查询只需要扫描 idx_order_no 索引就可以搞定了，不再需要临时表了。

总的来说，发现在执行计划里面有 using filesort 或者 Using temporary 的时候，特别需要注意，这往往存在着很大优化的余地，最好进行改进，变为使用 Using index 会更好。

Start temporary, End temporary

有子查询时，查询优化器会优先尝试将 IN 子查询转换成 semi-join(半连接优化技术，本质上是把子查询上拉到父查询中，与父查询的表做 join 操作)，而 semi-join 又有好多种执行策略，当执行策略为 DuplicateWeedout 时，也就是通过建立临时表来实现为外层查询中的记录进行去重操作时，驱动表查询执行计划的 Extra 列将显示 Start temporary 提示，被驱动表查询执行计划的 Extra 列将显示 End temporary 提示。

LooseScan

在将 In 子查询转为 semi-join 时，如果采用的是 LooseScan 执行策略，则在驱动表执行计划的 Extra 列就是显示 LooseScan 提示。

FirstMatch(tbl_name)

在将 In 子查询转为 semi-join 时，如果采用的是 FirstMatch 执行策略，则在被驱动表执行计划的 Extra 列就是显示 FirstMatch(tbl_name)提示。