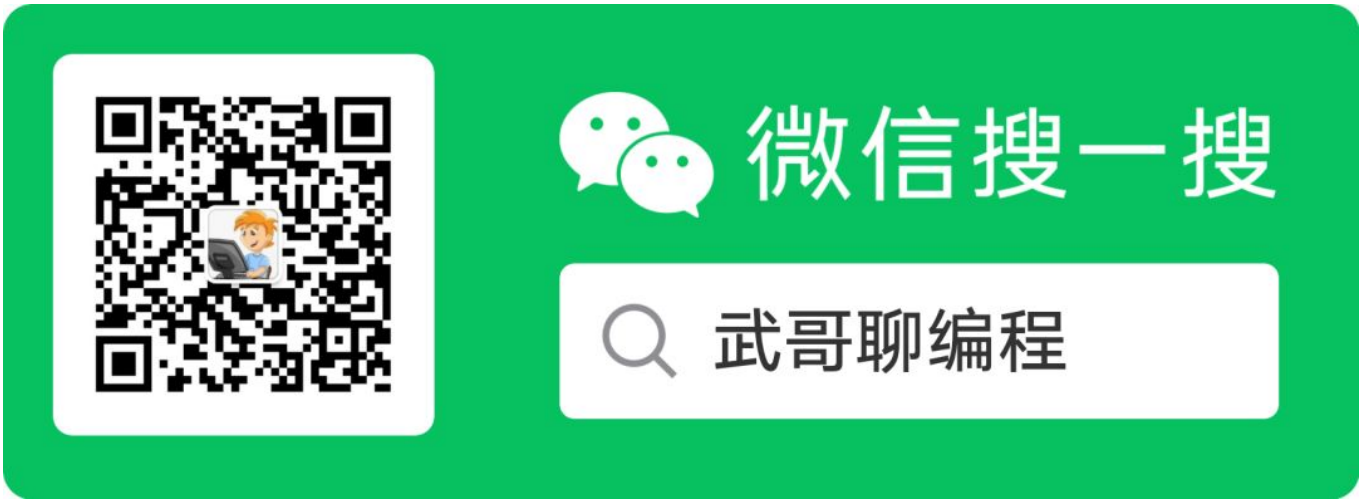
MySQL执行计划Explain详解

六点半起床 武哥聊编程 今天



本文来源：http://8rr.co/S4B7

大家都知道，mysql在执行查询的时候会进行查询优化。简单来讲就是执行的时候先基于成本和规则优化生成执行计划，然后再按照执行计划执行查询。本文主要介绍 EXPLAIN 各输出项的含义，从而帮助大家更好的进行**sql性能优化**！

> 本文主要内容是根据掘金小册《从根儿上理解 MySQL》整理而来。如想详细了解，建议购买掘金小册阅读。

我们可以在查询语句前面加上 EXPLAIN 关键字来查看这个查询的执行计划。例如

```
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)
```

可以看到，执行计划包含很多输出列，我们先简单过一下各列的大致作用，后面再进行详细讲解。

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

前置相关知识点

为了详细了解执行计划各列含义，我们先得了解以下相关知识点。

不相关子查询

如果*子查询*可以单独运行出结果，而不依赖于外层查询，我们把这个子查询称之为*不相关子查询*。

相关子查询

如果子查询的执行需要依赖于外层查询的值，我们就把这个子查询称之为*相关子查询*。

子查询物化

不直接将不相关子查询的结果集当作外层查询的参数，而是将该结果集写入一个临时表(物化表)里。例如：

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

假设子查询物化表的名称为 `materialized_table` ，该物化表存储的子查询结果集的列为 `m_val` 。子查询物化之后可以将表 **`s1`** 和子查询物化表 **`materialized_table`** 进行内连接操作，然后获取对应的查询结果。

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

```

将子查询转换为semi-join

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

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

我们可以把这个查询理解成：对于 **`s1`** 表中的某条记录，如果我们能在 **`s2`** 表（准确的说是执行完 **`WHERE s2.key3 = 'a'`** 之后的结果集）中找到一条或多条符合 **`s2.common_field=s1.key1`** 的记录，那么该条 **`s1`** 表的记录就会被加入到最终的结果集。这个过程其实和把 `s1` 和 `s2` 两个表连接起来的效果很像：

```
SELECT s1.* FROM s1 INNER JOIN s2 ON s1.key1 = s2.common_field WHERE
s2.key3 = 'a';
```

这么做唯一的问题在于，对于 s1 表的某条记录来说，如果 s2 表中有多条记录满足 s1.key1 = s2.common_field 这个条件，那么该记录会被多次加入最终的结果集，因此二者不能认为是完全等价的，因此就有了 semi-join (半连接)。将 s1 表和 s2 表进行半连接的意思就是：对于 s1 表的某条记录来说，我们只关心在 s2 表中是否存在与之匹配的记录，而不关心具体有多少条记录与之匹配，最终的结果集中只保留 s1 表的记录。当然 semi-join 是mysql内部机制，无法直接用在sql语句中。

semi-join实现机制

Table pullout （子查询中的表上拉）

当子查询的查询列表处只有主键或者唯一索引列时，可以直接把子查询中的表上拉到外层查询的 FROM 子句中，并把子查询中的搜索条件合并到外层查询的搜索条件中，比如这个：

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

```

由于 key2 列是 s2 表的唯一二级索引列，所以我们可以直接把 s2 表上拉到外层查询的 FROM 子句中，并且把子查询中的搜索条件合并到外层查询的搜索条件中，实际上就是直接将子查询优化为连接查询，上拉之后的查询就是这样的：

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

DuplicateWeedout execution strategy （重复值消除）

比如下面这个查询语句：

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

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

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

```

这样在执行连接查询的过程中，每当某条s1表中的记录要加入结果集时，就首先把这条记录的id值加入到这个临时表里。这种使用临时表消除 `semi-join` 结果集中的重复值的方式称之为 `DuplicateWeedout` 。

LooseScan execution strategy （松散扫描）

比如下面这个查询语句：

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

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

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

FirstMatch execution strategy （首次匹配）

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

执行计划详解

为了详细解释执行计划各列含义，先建2张示例表 s1 和 s2 ，它们的表结构完全一样。

```
CREATE TABLE s1 (
    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;

```

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，因此**EXPLAIN语句输出的每条记录都对应着某个单表的访问方法**。其中的 `table` 列代表的就是该表的表名。比如：

```
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)
```

上面的查询只涉及单表查询，因此 `EXPLAIN` 只输出了一条记录。 `table` 列的值是 `s1` ，表示该条记录描述了**对 `s1` 表的访问方法**。

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， `EXPLAIN` 只输出了两条记录。 `table` 列的值是 `s1` 和 `s2` ，分别表示了**对 `s1` 表和 `s2` 表的访问方法**。

id

大家都知道，查询语句中一般都会包含一个或多个 `select` 关键字。可以简单认为，**查询语句每出现一个 `select` 关键字，执行计划中就会有一个对应的id值**。比如下边这个查询中只有一个 `SELECT` 关键字：

```
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` 往往是对多张表进行查询的，所以在执行计划中就会有多条记录，但是它们的 `id` 都是一样的。其中，出现在前边的表是驱动表，出现在后边的表是被驱动表。

```
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)
```

对于子查询来说，就可能包含多个 **select** 关键字，每个 **select** 关键字都会对应一个唯一的**id**值。

```
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)
```

但是还有一点需要注意：**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**。此时执行计划的id值就是一样的了。

对于包含 `union` 关键字的查询来说，除了每个 `select` 关键字对应一个id值，还会包含一个 `id` 值为 `NULL` 的记录。这条记录主要用来表示将两次查询的结果集进行去重的(`union all` 因为不需要去重，所以没有这条记录)。

```
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)
```

select_type

我们已经知道，每一个 `select` 关键字都代表一次小查询，而 `select_type` 属性就是用来描述当前这个小查询的含义的。 `select_type` 属性含义(直接用官网英文表示)如下：

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

名称	描述
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)

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，比如常见的单表查询和连接查询等。

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)
```

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 。

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 。

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)
```

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)
```

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` 。

```
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)
```

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)
```

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)
```

type

上面提到过，执行计划的一条记录就代表了对一张表的访问方法，其中的 `type` 列就是用描述访问方法的。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` 。

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` 。

const

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` 。

```
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` 。

```
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)
```

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` 。

```
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)
```

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` 。

```
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，但是某些场景下也可能使用索引合并，此时的 `type` 就是 `index_merge` 。

```
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)
```

索引合并

一般情况下，执行一个查询最多只会用到一个索引。但是在特殊情况下也可能会使用多个二级索引，使用这种方式执行的查询称为 `index_merge` 。具体的索引合并算法有下边三种。

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

```
SELECT * FROM single_table WHERE key1 = 'a' AND key3 = 'b';
```

- Union合并 我们在写查询语句时经常想把既符合某个搜索条件的记录取出来，也把符合另外的某个搜索条件的记录取出来，我们说这些不同的搜索条件之间是OR关系。比如：

```
SELECT * FROM single_table WHERE key1 = 'a' OR key3 = 'b'
```

`Intersection` 是交集的意思，这适用于使用不同索引的搜索条件之间使用 `AND` 连接起来的情况； `Union` 是并集的意思，适用于使用不同索引的搜索条件之间使用 `OR` 连接起来的情况。

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

```
SELECT * FROM single_table WHERE key1 < 'a' OR key3 > 'z'
```

我们把上述这种先按照二级索引记录的主键值进行排序，之后按照 Union 索引合并方式执行的方式称之为 Sort-Union 索引合并，很显然，这种 Sort-Union 索引合并比单纯的 Union 索引合并多了一步对二级索引记录的主键值排序的过程。

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)
```

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)
```

index

需要扫描全部的索引记录时，该表的访问方法就是 `index` 。

```
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)
```

ALL

全表扫描

possible_keys和key

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)
```

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；对于指定字符集是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。

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具体信息，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)
```

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)
```

filtered

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，因为这直接影响了驱动表的扇出值。在 `rows` 样的情况下， `filtered` 越大，扇出值越小，效率可能也越高。比如：

```
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。

Extra

`Extra` 是用来说明一些额信息的，从而帮助我们更加准确的理解查询。下面我们挑几个比较常见的进行介绍。

No tables used

当查询语句中没有 `from` 字句时会出现 `No tables used` 。

Impossible WHERE

当查询语句中的 `where` 字句永远为 `false` 时会出现 `Impossible WHERE` 。

No matching min/max row

当查询列表有 `min()` 或者 `max()` 聚集函数，但是没有匹配到对应的记录时会出现 `No matching min/max row` 。

Using index

当使用 `索引覆盖` 的时候，会出现 `Using index` 。

Using index condition

如果查询的执行过程中使用了**索引条件下推(Index Condition Pushdown)**，就会出现 `Using index condition` 。例如：

```
SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%a';

```

1、先根据 `key1 > 'z'` 这个条件，定位到二级索引 `idx_key1` 中对应的二级索引记录。
2、先不回表，而是检测是否满足 `key1 LIKE '%a'` 条件，最后再将满足条件的二级索引记录回表。

Using where

当使用全表扫描执行查询时，如果查询语句包含 `where` 条件，就会出现 `Using where` 。
当使用索引访问执行查询时，如果 `where` 字句包含非索引列字段，也会出现 `Using where` 。

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，也就是我们所讲的*基于块的嵌套循环算法*，比如下边这个查询语句：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.common_field =
s2.common_field;
+----+-------------+-------+------------+------+---------------+------
+---------+------+------+----------+---------------------------------
-----------------+
| 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 |    10.00 | Using where; Using join buffer
(Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------
+---------+------+------+----------+---------------------------------
-----------------+
2 rows in set, 1 warning (0.03 sec)
```

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：

```
mysql> EXPLAIN SELECT * FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 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              | 9688 |   100.00 | NULL
|
|  1 | SIMPLE      | s2    | NULL       | ref   | idx_key1      |
idx_key1 | 303     | xiaohaizi.s1.key1 |    1 |    10.00 | Using where;
Not exists |
+----+-------------+-------+------------+-
```

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

如果使用了*索引合并*执行查询，则会出现 `Using intersect(...)` 或者 `Using union(...)` 或者 `Using sort_union(...)` 。比如：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' AND 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_key3,idx_key1 | 303,303 | NULL |    1 |
100.00 | Using intersect(idx_key3,idx_key1); Using where |
+----+-------------+-------+------------+-------------+----------------
---+------------------+---------+------+------+----------+------------
-----------------------------------+
1 row in set, 1 warning (0.01 sec)
```

Zero limit

当 `limit` 子句参数为0时，就会出现 `Zero limit` 。

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：

```
mysql> EXPLAIN SELECT * FROM s1 ORDER BY key1 LIMIT 10;
+----+-------------+-------+------------+-------+---------------+------
----+---------+------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key
| key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+------
----+---------+------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          |
idx_key1 | 303     | NULL |   10 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+------
----+---------+------+------+----------+-------+
1 row in set, 1 warning (0.03 sec)
```

但是更多情况下，排序操作无法使用到索引，而是只能使用文件排序(`filesort`)。如果排序使用了 `filesort` ，那么在 `Extra` 列就会出现 `Using filesort` 。

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，比方说这样：

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

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。

Start temporary, End temporary

查询优化器会优先尝试将IN子查询转换成 `semi-join` ，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，就是这样：

```
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)
```

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，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key3 IN (SELECT key1 FROM s2
WHERE key1 > 'z');
+----+-------------+-------+------------+-------+---------------+------
----+---------+-------------------+------+----------+-----------------
-------------------+
| id | select_type | table | partitions | type  | possible_keys | key
| key_len | ref               | rows | filtered | Extra
|
+----+-------------+-------+------------+-------+---------------+------
----+---------+-------------------+------+----------+-----------------
-------------------+
|  1 | SIMPLE      | s2    | NULL       | range | idx_key1      |
idx_key1 | 303     | NULL              |  270 |   100.00 | Using where;
Using index; LooseScan |
|  1 | SIMPLE      | s1    | NULL       | ref   | idx_key3      |
idx_key3 | 303     | xiaohaizi.s2.key1 |    1 |   100.00 | NULL
|
+----+-------------+-------+------------+-------+---------------+------
----+---------+-------------------+------+----------+-----------------
-------------------+
2 rows in set, 1 warning (0.01 sec)
```

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，比如这样：

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

```
+----+-------------+-------+------------+------+-------------------+---
-------+---------+------------------+------+----------+--------------
--------------+
| id | select_type | table | partitions | type | possible_keys     |
key      | key_len | ref              | rows | filtered | Extra
|
+----+-------------+-------+------------+------+-------------------+---
-------+---------+------------------+------+----------+--------------
--------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_key3          |
NULL     | NULL    | NULL             | 9688 |   100.00 | Using where
|
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1,idx_key3 |
idx_key3 | 303     | xiaohaizi.s1.key3 |    1 |     4.87 | Using where;
FirstMatch(s1) |
+----+-------------+-------+------------+------+-------------------+---
-------+---------+------------------+------+----------+--------------
--------------+
2 rows in set, 2 warnings (0.00 sec)
```


最后免费给大家分享**50个Java项目实战资料**，涵盖入门、进阶各个阶段学习内容，可以说非常全面了。大部分视频还附带源码，学起来还不费劲！

附上截图。（下面有下载方式）。