# MySQL 执行计划 explain 详解

**explain 命令是查看查询优化器如何决定执行查询的主要方法。**

**这个功能有局限性，并不总会说出真相，但它的输出是可以获取的最好信息，**

**值得花时间去了解，因为可以学习到查询是如何执行的。**

## SQL 优化准则

禁用 select *

使用 select count(*) 统计行数

尽量少运算

尽量避免全表扫描，如果可以，在过滤列建立索引

尽量避免在 where 子句对字段进行 null 判断

尽量避免在 where 子句使用!= 或者<>

尽量避免在 where 子句使用 or 连接

尽量避免对字段进行表达式计算

尽量避免对字段进行函数操作

尽量避免使用不是复合索引的前缀列进行过滤连接

尽量少排序，如果可以，建立索引

尽量少 join

尽量用 join 代替子查询

尽量避免在 where 子句中使用 in,not in 或者 having，使用 exists,not exists 代替

尽量避免两端模糊匹配 like %***%

尽量用 union all 代替 union

尽量早过滤

避免类型转换

尽量批量 insert

优先优化高并发 sql，而不是频率低的大 sql

尽可能对每一条 sql 进行 explain

尽可能从全局出发

**2、 执行计划的生成和查看**

2.1 执行计划的生成方法：explain select .....

生成的方法很简单在相应的 select 前面加 explain 即可

2.2 执行计划的查看

**Id：**包含一组数字，表示查询中执行 select 子句或操作表的顺序；

执行顺序从大到小执行；

当 id 值一样的时候，执行顺序由上往下；

**Select_type:**表示查询中每个 select 子句的类型（简单 OR 复杂），有以下几种

? SIMPLE：查询中不包含子查询或者 UNION

? PRIMARY：查询中若包含任何复杂的子部分，最外层查询则被标记为 PRIMARY

? SUBQUERY：在 SELECT 或 WHERE 列表中包含了子查询，该子查询被标记为 SUBQUERY

? DERIVED：在 FROM 列表中包含的子查询被标记为 DERIVED（衍生）

? 若第二个 SELECT 出现在 UNION 之后，则被标记为 UNION；

? 若 UNION 包含在 FROM 子句的子查询中，外层 SELECT 将被标记为：DERIVED

? 从 UNION 表获取结果的 SELECT 被标记为：UNION RESULT

**Type：**表示 MySQL 在表中找到所需行的方式，又称“访问类型”，常见有以下几种

? ALL：Full Table Scan， MySQL 将进行全表扫描；

? index：Full Index Scan，index 与 ALL 区别为 index 类型只遍历索引树；

? range：range Index Scan，对索引的扫描开始于某一点，返回匹配值域的行，常见于 between、<、>等的查询；

? ref：非唯一性索引扫描，返回匹配摸个单独值的所有行。常见于使用非唯一索引或唯一索引的非唯一前缀进行的查找；

? eq_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描

? const、system：当 MySQL 对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于 where 列表中，MySQL 就能将该查询转换为一个常量

? NULL：MySQL 在优化过程中分解语句，执行时甚至不用访问表或索引

**possible_keys：**指出 MySQL 能使用哪个索引在表中找到行，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用；

**key：**显示 MySQL 在查询中实际使用的索引，若没有使用索引，显示为 NULL。当查询中若使用了覆盖索引，则该索引仅出现在 key 列表中

**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**
**ref：表示上述表的连接匹配条件，即那些列或常量被用于查找索引列上的值；**
**rows：表示 MySQL 根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数；**
**Extra：包含不适合在其他列中显示但十分重要的额外信息；**

? Using where :表示 MySQL 服务器在存储引擎受到记录后进行“后过滤”（Post-filter），如果查询未能使用索引，Using where 的作用只是提醒我们 MySQL 将用 where 子句来过滤结果集

? Using temporary：表示 MySQL 需要使用临时表来存储结果集，常见于排序和分组查询；

? Using filesort：MySQL 中无法利用索引完成的排序操作称为“文件排序”；

2.3 mysql 执行计划的局限

EXPLAIN 不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况

EXPLAIN 不考虑各种 Cache

EXPLAIN 不能显示 MySQL 在执行查询时所作的优化工作

部分统计信息是估算的，并非精确值

EXPALIN 只能解释 SELECT 操作，其他操作要重写为 SELECT 后查看执行计划

### 3、 对于非 select 语句查看执行计划

在实际的工作中也经常需要查看一些诸如 update、delete 的执行计划，（mysql5.6 的版本已经支持直接查看）但是这时候并不能直接通过 explain 来进行查看，而需要通过改写语句进行查看执行计划；

在一个生产数据库的慢查询日志发现有条语句如下：

```
Count: 13 Time=73.44s (954s) Lock=0.00s (0s) Rows=0.0 (0), ipos[ipos]@2hosts
update ipos_zdjhd m,ipos_zdjhdtj tj set
m.qr=N,m.qrrq='S',m.qrr='S',tj.qr=N,tj.qrrq='S'
where m.ydjh='S' and tj.djbh='S'
```

可以改写如下：

```
Explain Select m.qr , m.qrr , tj.qr , tj.qrrq from ipos_zdjhd m,ipos_zdjhdtj tj where
m.ydjh='17233' and tj.djbh='48632';
```

马上可以发现 ipos_zdjhd 表进行了全表扫描，而 ipos_zdjhd 表有 1076971 行的数据，所以整个 update 的操作肯定是一个很慢的过程，经过和开发人员沟通后，在 ipos_zdjhd 表增加相应的索引便让整个过程提升了 **500 倍。**

小结：执行计划加上慢查询日志组成了 mysql 调优过程的一组调优利器，当数据库稳定过后参数的调优是很少的一部分，80%以上的调优都会是 SQL 调优。

## 调用 EXPLAIN

在 select 之前添加 explain，mysql 会在查询上设置一个标记，当执行查询计划时，这个标记会使其返回关于执行计划中每一步的信息，而不是执行它。
它会返回一行或多行信息，显示出执行计划中的每一部分和执行次序。

这是一个简单的 explain 效果：

```
mysql> EXPLAIN SELECT 1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: No tables used
1 row in set (0.00 sec)
```

在查询中每个表在输出只有一行，如果查询是两个表的联接，那么输出中将有两行。
别名表单算为一个表，因此，如果把一个表与自己联接，输出中也会有两行。
“表”的意义在这里相当广，可以是一个子查询，一个 union 结果等。

同时 explain 有两个变种
EXPLAIN EXTENDED 会告诉服务器“逆向编译” 执行计划为一个 select 语句。可以通过紧接其后运行 show warnings 看到这个生成的语句，这个语句直接来自执行计划，而不是原 SQL 语句，到这点上已经变成一个数据结构。
大部分场景下，它都与原语句不相同，你可以检测查询偶花旗到底是如何转化语句的。
EXPLAIN EXTENDED 在 mysql 5.0 以上版本中可用，在 5.1 中增加了一个 filtered

列。

EXPLAIN PARTITIONS 会显示查询将访问的分区，如果查询是基于分区表的话。在 mysql 5.1 以上的版本中会存在。

EXPLAIN 限制：

- explain 根本不会告诉你触发器、存储过程或 UDF 会如何影响查询
- 不支持存储过程，尽管可以手动抽取查询并单独地对其进行 explain 操作
- 它并不会告诉你 mysql 在执行计划中所做的特定优化
- 它并不会显示关于查询的执行计划的所有信息
- 它并不区分具有相同名字的事物，例如，它对内存排列和临时文件都使用“filesort”，并且对于磁盘上和内存中的临时表都显示“Using temporary”
- 可能会产生误导，比如，它会对一个有着很小 limit 的查询显示全索引扫描（mysql 5.1 的 explain 关于检查的行数会显示更精准的信息，但早期版本并不考虑 limit）

## 重写非 SELECT 查询

mysql explain 只能解释 select 查询，并不会对存储程序调用和 insert、update、delete 或其他语句做解释。然而，你可以重写某些非 select 查询以利用 explain。为了达到这个目的，只需要将该语句转化成一个等价的访问所有相同列的 select,任何体积的列都必须在 select 列表，关联子句，或者 where 子句中。

假如，你想重写下面的 update 语句使其可以利用 explain

```
1. UPDATE sakila.actor

2. INNER JOIN sakila.film_actor USING (actor_id)
3. SET actor.last_update=film_actor.last_update;
```

下面的 explain 语句并不等价于上面的 update,因为它并不要求服务器

从任何一个表上获取 last_update 列

```
mysql> EXPLAIN SELECT film_actor.actor_id
    -> FROM sakila.actor
    -> INNER JOIN sakila.film_actor USING(actor_id)\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: index
possible_keys: PRIMARY
          key: idx_actor_last_name
      key_len: 137
          ref: NULL
         rows: 200
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 13
        Extra: Using index
2 rows in set (0.00 sec)
```

这个差别非常重要。例如，输出结果显示 mysql 将使用覆盖索引，但是，当检索并更新 last_updated 列时，就无法使用覆盖索引了，下面这种改写法就更接近原来的语句：

```
mysql> EXPLAIN SELECT film_actor.last_update, actor.last_update
    -> FROM sakila.actor
    -> INNER JOIN sakila.film_actor USING (actor_id)\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 200
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 13
        Extra:
2 rows in set (0.00 sec)
```

像这样重写查询并不非常科学，但对帮助理解查询是怎么做的已经足够好了。

（MySQL 5.6 将允许解释非 SELECT 查询）

显示查询计划时，对于写查询并没有“等价”的读查询，理解这一点非常重要。一个 SELECT 查询只需要找到数据的一份副本并返回。而任何修改数据的查询必须在所有索引上查找并修改其所有副本，这常常比看起来等价的 SELECT 查询的消耗要高得多。

## EXPLAIN 中的列

将在下一部分展示 explain 结果中每一列的意义。

输出中的行以 mysql 实际执行的查询部分的顺序出现，而这个顺序不总是与其在原始 SQL 中的一致。

### 【id 列】

这一列总是包含一个编号，识别 select 所属的行，如果在语句当中没有子查询或联合，那么只会有唯一的 select，于是每一行在这个列中都将显示一个 1，否则，内层的 select 语句一般会顺序编号，对应于其在原始语句中的位置。

mysql 将 select 查询氛围简单和复杂类型，复杂类型可分为三大类：简单子查询，所谓的派生表（在 from 子句中的子查询），以及 union 查询。

下面是一个简单的子查询：

```
mysql> EXPLAIN SELECT (SELECT 1 FROM sakila.actor LIMIT 1) FROM sakila.film;
+----+-------------+-------+-------+---------------+---------------------+---------+------+------+-------------+
| id | select_type | table | type  | possible_keys | key                 | key_len | ref  | rows | Extra       |
+----+-------------+-------+-------+---------------+---------------------+---------+------+------+-------------+
|  1 | PRIMARY     | film  | index | NULL          | idx_fk_language_id  | 1       | NULL | 1022 | Using index |
|  2 | SUBQUERY    | actor | index | NULL          | idx_actor_last_name | 137     | NULL |  200 | Using index |
+----+-------------+-------+-------+---------------+---------------------+---------+------+------+-------------+
2 rows in set (0.00 sec)
```

from 子句中的子查询和联合给 id 列增加了更多的复杂性。

下面是一个 from 子句中的基本子查询：

```
mysql> EXPLAIN SELECT film_id FROM (SELECT film_id FROM sakila.film) AS der;
+----+-------------+------------+-------+---------------+-------------------+---------+------+------+-------------+
| id | select_type | table      | type  | possible_keys | key               | key_len | ref  | rows | Extra       |
+----+-------------+------------+-------+---------------+-------------------+---------+------+------+-------------+
|  1 | PRIMARY     | <derived2> | ALL   | NULL          | NULL              | NULL    | NULL | 1000 |             |
|  2 | DERIVED     | film       | index | NULL          | idx_fk_language_id | 1       | NULL | 1022 | Using index |
+----+-------------+------------+-------+---------------+-------------------+---------+------+------+-------------+
2 rows in set (0.00 sec)
```

如你所知，这个查询执行时有一个匿名的临时表，mysql 内部通过别名（der）在外层查询中引用这个临时表，在更复杂的查询中可以看到 ref 列。

最后下面是一个 union 查询：

```
mysql> EXPLAIN SELECT 1 UNION ALL SELECT 1;
+----+--------------+------------+------+---------------+------+---------+------+------+----------------+
| id | select_type  | table      | type | possible_keys | key  | key_len | ref  | rows | Extra          |
+----+--------------+------------+------+---------------+------+---------+------+------+----------------+
|  1 | PRIMARY      | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL | No tables used |
|  2 | UNION        | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL | No tables used |
| NULL | UNION RESULT | <union1,2> | ALL  | NULL          | NULL | NULL    | NULL | NULL |                |
+----+--------------+------------+------+---------------+------+---------+------+------+----------------+
3 rows in set (0.00 sec)
```

注意第三个额外的行，union 的结果总是放在一个匿名临时表中，之后 mysql 将结果读取到临时表外，临时表并不在原 SQL 中出现，因此它的 id 列为 null。

与之前的例子相比（演示子查询的那个 from 子句中），从这个查询产生的临时表在结果中出现在最后一行，而不是第一行。

到目前为止这些都非常直截了当，但这三类语句的混合则会使输出变得非常复杂，我们稍后就会看到。

【select_type 列】

这一列显示了对应行是简单还是复杂的 select（如果是后者，那么是三

种复杂类型中的哪一种）。simple 值意味着查询不包括子查询和 union，如果查询有任何负责的子部分，则最外层部分标记为 primary，其他部分标记如下：

SUBQUERY

包含在 select 列表中的子查询中的 select（换句话说，不在 from 子句中）标记为 SUBQUERY

DERIVED

DERIVED 值用来表示包含在 FROM 子句的子查询中的 select，mysql 会递归执行并将结果放到一个临时表中。服务器内部称其“派生表”，因为该临时表是从子查询中派生来的。

UNION

在 UNION 中的第二个和随后的 select 被标记为 unoin，第一个 select 被标记就好像它以部分外查询来执行。这就是之前的例子中在 union 中的第一个 select 显示为 primary 的原因。如果 union 被 from 子句中的子查询包含，那么它的第一个 select 会被标记为 derived。

UNION RESULT

用来从 union 的匿名临时表检索结果的 select 被标记为 UNION RESULT。

除了这些值，SUBQUERY 和 UNION 还可以被标记为 DEPENDENT 何 UNCACHEABLE。

DEPENDENT 意味着 select 依赖于外层查询中发现的数据。

UNCACHEABLE 意味着 select 中的某些特性阻止结果被缓存于一个 Item_cache 中。

（Item_cache 未被文档记载，它与查询缓存不是一回事，尽管它可以被一些相同类型的构件否定，例如 RAND()函数。）

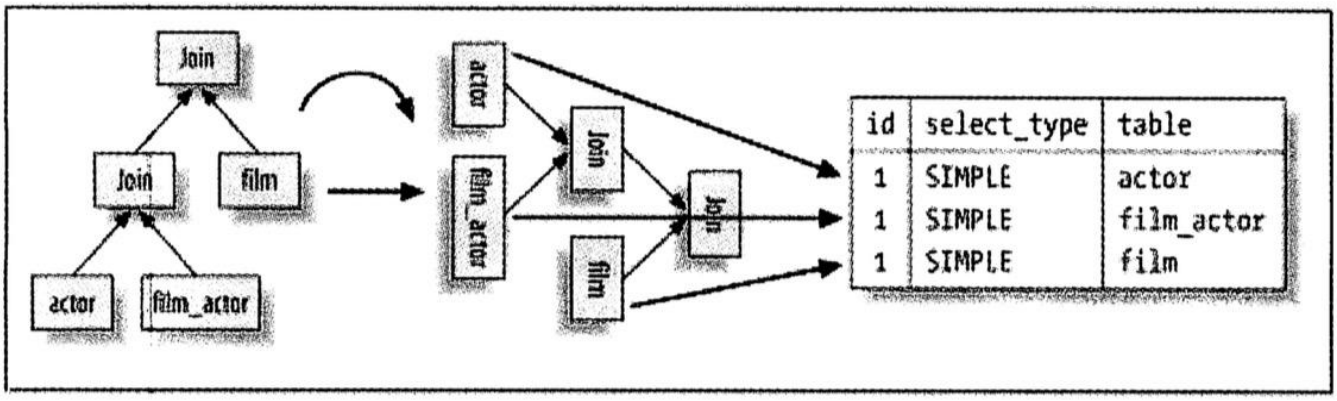
【table 列】

这一列显示了对应行正在访问哪个表，在通常情况下，它相当明了：它就是哪个表，或是该表的列明（如果 SQL 中定义了别名）。

可以在这一列中从上往下观察 mysql 的关联优化器为查询选择的关联顺序，例如，可以看到在下面的查询中 mysql 选择的关联顺序不同于语句中所指定的顺序：

```
mysql> EXPLAIN SELECT film.film_id
    -> FROM sakila.film
    -> INNER JOIN sakila.film_actor USING(film_id)
    -> INNER JOIN sakila.actor USING(actor_id);
+----+-------------+------------+--------+-----------------------
| id | select_type | table      | type   | possible_keys
+----+-------------+------------+--------+-----------------------
|  1 | SIMPLE      | actor      | index  | PRIMARY
|  1 | SIMPLE      | film_actor | ref    | PRIMARY,idx_fk_film_id
|  1 | SIMPLE      | film       | eq_ref | PRIMARY
+----+-------------+------------+--------+-----------------------
3 rows in set (0.01 sec)
```

mysql 的执行计划总是左侧深度优先树，如果把这个计划放倒，就能按顺序读出叶子节点，它们直接对应于 explain 中的行，之前的查询计划看起来如下图所示：




**派生表和联合**

当 from 子句中有子查询或有 union 时，table 列会变得复杂很多，这些场景下，确实没有一个“表”可以参考到，因为 mysql 创建的匿名临时表仅在查询执行过程中存在。

当在 from 子句中有子查询时，table 列是<derivedN>的形式，其中 N 是子查询的 id。这总是“向前引用”——换言之，N 指向 explain 输出中后面的一行。

当有 union 时，union result 的 table 列包含一个参与 union 的 id 列表。这总是“向后引用”，因为 union result 出现在 union 中所有参与行之后，如果在列表中有超过 20 个 id，table 列卡诺被截断以防止太长，此时不可能看到所有的值。幸运的是，仍然可以推测包括哪些行，因为你可以看到第一行的 id，在这一行和 union result 之间出现的一切都会以某种方式被包含。

**一个复杂 select 类型的例子**

下面是一个无意义的查询，我们这里把它用作某种复杂 select 类型的紧凑示例

```
mysql> EXPLAIN
    -> SELECT actor_id,
    -> (SELECT 1 FROM film_actor WHERE film_actor.actor_id = der_1.actor_id LIMIT 1)
    -> FROM ( SELECT actor_id FROM actor LIMIT 5 ) AS der_1
    -> UNION ALL
    -> SELECT film_id,
    -> (SELECT @var1 FROM rental LIMIT 1)
    -> FROM ( SELECT film_id, ( SELECT 1 FROM store LIMIT 1 ) FROM film LIMIT 5) AS der_2;
```

limit 子句只是为了方便起见，以防你打算不以 explain 方式执行来看结果。

以下是 explain 的部分结果：

| id | select_type | table |
|---|---|---|
| 1 | PRIMARY | <derived3> |
| 3 | DERIVED | actor |
| 2 | DEPENDENT SUBQUERY | film_actor |
| 4 | UNION | <derived6> |
| 6 | DERIVED | film |
| 7 | SUBQUERY | store |
| 5 | UNCACHEABLE SUBQUERY | rental |
| NULL | UNION RESULT | <union1,4> |

我们特意让每个查询部分访问不同的表，以便可以弄清楚问题所在，但仍然难以解决，从上面开始看起：

第 1 行向前引用了 der_1，这个查询被标记为<derived3>，在原 SQL 中是第 2 行，想了解输出中哪些行引用了<derived3>中的 select 语句，往下看。

第 2 行，它的 id 是 3，因为它是查询中第 3 个 select 的一部分，归为 derived 类型是因为它嵌套在 from 子句中的子查询内部，在原 sql 中为第 4 行。

第 3 行的 id 为 2，在原 sql 中为第 3 行，注意，它在具有更高 id 的行的后面，暗示后面再执行，这是合理的。它被归为 DEPENDENT SUBQUERY，意味着其结果依赖于外层查询（亦即某个相关子查询）。本例中的外查询从第 2 行开始，从 der_1 中检索数据的 select。

第 4 行被归为 union，意味着它是 union 中的第 2 个或之后的 select，

它的表为<derived6>，意味着是从子句 from 的子查询中检索数据并附加到 union 的临时表。像之前一样，要找到显示这个子查询的查询计划的 explain 行，往下看。

第 5 行是在原 sql 中的第 8 行的 der_2 子查询，explain 称其为<derived6>。

第 6 行是<derived6>的 select 列表中的一个普通子查询,它的 id 为 7，这非常重要……

……因为它比 5 大，而 5 是第 7 行的 id。为什么重要？因为它显示了<derived6>子查询的边界。当 explain 输出 select 类型为 derived 的一行时，表示一个“嵌套范围”开始。如果后续行的 id 更小（本例中，5 小于 6），意味着嵌套范围已经被关闭。这就让我们知道第 7 行是从<derived6>中检索数据的 select 列表中的部分——例如，第 4 行的 select 列表的第一部分（原 sql 中的第 7 行）。这个例子相当容易理解，不需要知道嵌套范围的意义和规则，当然有时候并不是这么容易。关于输出中的这一行另外一个要注意的是，因为有用户变量，它被列为 UNCACHEABLE SUBQUERY。

最后一行 union result，它代表从 union 的临时表中读取行的阶段。你可以从这行开始反过来向后，如果你愿意的话，它返回 id 是 1 和 4 的行结果，它们分别引用了<derived3>和<derived6>

如你所见，这些复杂的 select 类型的组合会使 explain 的输出相当难懂，理解规则会使其简单些，但仍然需要多时间。

阅读 explain 的输出经常需要在列表中跳来跳去，例如，再查看第 1 行输出，仅仅盯着看，是无法知道它是 union 的一部分的，只有看到最后 1 行你才会明白过来。

【type 列】

mysql 用户手册上说这一列显示了“关联类型”，但我们认为更准确的说法是访问类型——换言之就是 mysql 决定如何查找表中的行。下面是最重要的访问方法，依次从最差到最优：

ALL：

这就是所谓的全表扫描，意味着 mysql 必须扫描整张表，从头到尾，去找到需要的行。（有个例外，例如在查询里使用了 limit，或者在 extra 列中显示“Using distinct/not exists”。

index：

这个跟全表扫描一样，只是 mysql 扫描表时按索引次序而不是行，它的主要优点是避免了排序；最大缺点是要承担按索引次序读取整个表的开销。这通常意味着若是按随机次序访问行，开销将非常大。

如果在 extra 列中看到“Using index”，说明 mysql 正在使用索引覆盖，它只扫描索引的数据，而不是按索引次序的每一行，它比按索引次序全表扫描的开销要少很多。

range：

范围扫描就是一个有限制的索引扫描，它开始与索引里的某一点，返回匹配这个值域的行，这比全索引扫描要好一点，因为它用不着遍历全部索引，显而易见的扫描是带有 between 或在 where 子句里带有>的查询。

当 mysql 使用索引去查找一系列值时，例如 in()和 or 列表，也会显示为范围扫描，然而，这两者其实是相当不同的访问类型，在性能上有重要的差异。

此类扫描的开销跟索引类型的相当。

ref：

这是一种索引访问（有时也叫做索引查找），它返回所有匹配某个单个值的行，然而，它可能会找到多个符合条件的行，因此，它是查找和扫描的混合体，此类索引访问只有当使用非唯一性索引或者唯一性索引的非唯一性前缀时才会发生。把它叫做 ref 是因为索引要跟某个参考值相比较。这个参考值或者是一个常数，或者是来自多表查询前一个表里的结果值。

ref_or_null 是 ref 之上的一个变体，它意味着 mysql 必须在初次查找的结果里进行第二次查找以找出 null 条目。

eq_ref：

使用这种索引查找，mysql 知道最多只返回一条符合条件的记录，

这种访问方法可以在 mysql 使用主键或者唯一性索引查找时看到，它会将它们与某个参考值做比较。mysql 对于这类访问类型的优化做得非常好，因为它知道无需估计匹配行的范围或在找到匹配行后再继续查找。

const，system：

当mysql能对查询的某部分进行优化并将其转换成一个常量时，他就会使用这些访问类型，举例来说，如果你通过将某一行的主键放入 where 子句里的方式来选取此行的主键，mysql 就能把这个查询转换为一个常量，然后就可以高效地将表从联接执行中移除。

null：

这种访问方式意味着 mysql 能在优化阶段分解查询语句，在执行阶段甚至用不着再访问表或者索引。例如，从一个索引列里选取最小值可以通过单独查找索引来完成，不需要在执行时访问表。

【possible_keys 列】

这一列显示了查询可以使用哪些索引，这是基于查询访问的列和使用的比较操作符来判断的。这个列表是在优化过程的早期创建的，因此有些罗列出来的索引可能对于后续优化过程是没用的。

【key 列】

这一列显示了 mysql 决定采用哪个索引来优化对该表的访问。如果该索

引没有出现在 possible_keys 列中，那么 mysql 选用它是处于另外的原因——例如，它可能选择了一个覆盖索引，哪怕没有 where 子句。

换句话说，possible_keys 揭示了哪一个索引能有助于高效地进行查找，而 key 显示的是优化采用哪一个索引可以最小化查询成本。下面是一个例子：

```
mysql> EXPLAIN SELECT actor_id, film_id FROM film_actor\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: index
possible_keys: NULL
          key: idx_fk_film_id
      key_len: 2
          ref: NULL
         rows: 6315
        Extra: Using index
1 row in set (0.00 sec)
```

【key_len 列】

该列显示了 mysql 在索引里使用的字节数，如果 mysql 正在使用的只是索引里的某些列，那么就可以用这个值来算出具体是哪些列，要记住，mysql 5.5 及之前的版本只能使用索引的最左前缀，举例来说，film_actor 的主键是两个 smallint 列，并且每个 smallint 列是两字节，那么索引中的每项是 4 字节，以下就是一个查询的示例：

```
mysql> EXPLAIN SELECT actor_id, film_id FROM film_actor WHERE actor_id=4 ;
+----+-------------+------------+------+---------------+---------+---------+
| id | select_type | table      | type | possible_keys | key     | key_len |
+----+-------------+------------+------+---------------+---------+---------+
|  1 | SIMPLE      | film_actor | ref  | PRIMARY       | PRIMARY | 2       |
+----+-------------+------------+------+---------------+---------+---------+
1 row in set (0.00 sec)
```

基于结果中的 key_len 列，可以推断出查询使用唯一的首列——actor_id 列，来执行索引查找，当我们计算列的使用情况时，务必把字符列中的字符集页考虑进去。

```
mysql> CREATE TABLE t(
    -> a char(3) NOT NULL,
    -> b int(11) NOT NULL,
    -> c char(1) NOT NULL,
    -> PRIMARY KEY (a,b,c)
    -> ) ENGINE=MyISAM DEFAULT CHARSET=utf8;
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> INSERT INTO t(a,b,c)
    -> SELECT DISTINCT LEFT(TABLE_SCHEMA,3), ORD(TABLE_NAME),
    -> LEFT(COLUMN_NAME,1)
    -> FROM INFORMATION_SCHEMA.COLUMNS;
Query OK, 333 rows affected (0.07 sec)
Records: 333  Duplicates: 0  Warnings: 0
```

查看执行计划：

```
mysql> EXPLAIN SELECT a FROM t WHERE a='sak' AND b = 112;
+----+-------------+-------+------+---------------+---------+---------+
| id | select_type | table | type | possible_keys | key     | key_len |
+----+-------------+-------+------+---------------+---------+---------+
|  1 | SIMPLE      | t     | ref  | PRIMARY       | PRIMARY | 13      |
+----+-------------+-------+------+---------------+---------+---------+
1 row in set (0.00 sec)
```

这个查询中平均长度为 13 字节，即为 a 列和 b 列的总长度，a 列是 3

个字符，utf8 下每一个最多为 3 字节，而 b 列是一个 4 字节整型。

mysql 并不总是显示一个索引真正使用了多少，例如，如果对一个前缀模式匹配执行 like 查询，它会显示列的完全宽度正在被使用。

key_len 列显示了在索引字段中可能的最大长度，而不是表中数据使用的实际字节数，在前面例子中 mysql 总是显示 13 字节，即使 a 列恰巧只包含一个字符长度。换言之，key_len 通过查找表的定义而被计算出，而不是表中的数据。

【ref 列】

这一列显示了之前的表在 key 列记录的索引中查找值所用的列或常量，下面是一个展示关联条件和别名组合的例子，注意，ref 列反映了在查询文本中 film 表是如何以 f 为别名的：

```
mysql> EXPLAIN
    -> SELECT STRAIGHT_JOIN f.film_id
    -> FROM film AS f
    -> INNER JOIN film_actor AS fa
    -> ON f.film_id=fa.film_id AND fa.actor_id=1
    -> INNER JOIN actor AS a USING(actor_id);
```

| table | type | possible_keys | key | key_len | ref |
|---|---|---|---|---|---|
| f | index | PRIMARY | idx_fk_language_id | 1 | NULL |
| fa | eq_ref | PRIMARY,idx_fk_film_id | PRIMARY | 4 | const,sakila.f.film_id |
| a | const | PRIMARY | PRIMARY | 2 | const |

【rows 列】

这一列是 mysql 估计为了找到所需的行而要读取的行数。这个数字是内

嵌循环关联计划里的 循环数目，也就是说它不是 mysql 认为它最终要从表里读取出来的行数，而是 mysql 为了找到符合查询的每一点上标准的那些行而必须读取的行的平均数。(这个标准包括 sql 里给定的条件，以及来自联接次序上前一个表的当前列。)

根据表的统计信息和索引的选用情况，这个估算可能很不精确，在 mysql5.0 及更早的版本里，它也反映不出 limit 子句，举例来说，下面这个查询不会真的检查 1057 行。

```
mysql> EXPLAIN SELECT * FROM film LIMIT 1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1057
        Extra:
1 row in set (0.00 sec)
```

通过把所有 rows 列的值相乘，可以粗略地估算出整个查询会检查的行数，例如，以下这个查询大约会检查 2600 行。

```
mysql>  EXPLAIN SELECT f.film_id
    ->  FROM film AS f
    ->  INNER JOIN film_actor AS fa USING(film_id)
    ->  INNER JOIN actor AS a USING(actor_id);
```

```
+------+
| rows |
+------+
|  200 |
|   13 |
|    1 |
+------+
```

要记住这个数字是mysql认为它要检查的行数，而不是结果集里的行数，同时也要认识到有很多优化手段，例如关联缓冲区和缓存，无法影响到行数的显示，mysql 可能不必真的读所有它估计到的行，它也不知道任何关于操作系统或硬件缓存的信息。

【Extra 列】

这一列包含的是不适合在其他列显示的额外信息。mysql 用户手册里记录了大多数可以在这里出现的值。

常见的最重要的值如下。

“Using index”

此值表示 mysql 将使用覆盖索引，以避免访问表。不要把覆盖索引和 index 访问类型弄混了。

“Using where”

这意味着 mysql 服务器将在存储引擎检索行后再进行过滤，许

多 where 条件里涉及索引中的列，当（并且如果）它读取索引时，就能被存储引擎检验，因此不是所有带 where 子句的查询都会显示“Using where”。有时“Using where”的出现就是一个暗示：查询可受益于不同的索引。

“Using temporary”

这意味着 mysql 在对查询结果排序时会使用一个临时表。

“Using filesort”

这意味着 mysql 会对结果使用一个外部索引排序，而不是按索引次序从表里读取行。mysql 有两种文件排序算法，这两种排序方式都可以在内存或者磁盘上完成，explain 不会告诉你 mysql 将使用哪一种文件排序，也不会告诉你排序会在内存里还是磁盘上完成。

“Range checked for each record(index map: N)”

这个意味着没有好用的索引，新的索引将在联接的每一行上重新估算，N 是显示在 possible_keys 列中索引的位图，并且是冗余的。

**树形格式的输出**

mysql 用户往往更希望把 explain 的输出格式化成一棵树，更加精确地展示执行计划。实际上，explain 查看执行计划的方式确实有点笨拙，树状结构也不适合表格化的输出，当 extra 列里有大量的值时，缺点更明显，使用 union 也是这样，union 跟 mysql 能做的其他类型的

联接不太一样，它不太适合 explain。

如果对 explain 的规则和特性有充分了解，使用树形结构的执行计划也是可行的。但是这有点枯燥，最好还是留给自动化的工具处理，Percona Toolkit 包含了 pt-visual-explain，它就是这样一个工具。

## MySQL 5.6 中的改进

mysql5.6 中将包含一个对 explain 的重要改进：能对类似 update、insert 等的查询进行解释，尽管可以将 dml 语句转化为等价的“select”查询并 explain，但结果并不会完全反映语句是如何执行的，因而这仍然非常有帮助。在开发使用类似 Percona Toolkit 中的 pt-upgrade 时曾尝试使用过那个技术，我们不止一次发现，在将查询转化为 select 时，优化器并不能按我们预期的代码路径执行。因而 explain 一个查询而不需要转化为 select，对我们理解执行过程中到底发生什么，是非常有帮助的。

mysql5.6 还将包括对查询优化和执行引擎的一系列修改，允许匿名的临时表尽可能晚地被具体化，而不总是在优化和执行使用到此临时表的部分查询时创建并填充它们，这将允许 mysql 可以直接解释带子查询的查询语句，而不需要先实际地执行子查询。

最后，mysql5.6 将通过在服务器中增加优化跟踪功能的方式改进优化器

的相关部分，浙江允许用户查看优化器坐出的抉择，以及输入（例如，索引的基数）和抉择的原因。这非常有帮助，不仅仅对理解服务器选择的执行计划如此，对为什么选择这个计划也如此。