"

# Explain 详解（下）

标签： MySQL 是怎样运行的

---

## 执行计划输出中各列详解

本章紧接着上一节的内容，继续唠叨EXPLAIN语句输出的各个列的意思。

### 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提供的额外信息有好几十个，我们就不一个一个介绍了（都介绍了感觉我们的文章就跟文档差不多了～），所以我们只挑一些平时常见的或者比较重要的额外信息介绍给大家哈。

- No tables used

  当查询语句的没有FROM子句时将会提示该额外信息，比如：

```
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 (0.00 sec)
```

- Impossible WHERE

  查询语句的WHERE子句永远为FALSE时将会提示该额外信息，比方说：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE 1 != 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 | Impossible WHERE |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------------+
1 row in set, 1 warning (0.01 sec)
```

- 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，比方说：

```
mysql> EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'abcdefg';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------------------+
| 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 matching min/max row |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------------------+
1 row in set, 1 warning (0.00 sec)
```

- 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_key1而不需要回表操作：

```
mysql> EXPLAIN SELECT key1 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 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
1 row in 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 key1 > 'z' AND key1 LIKE '%a';
```

  其中的key1 > 'z'可以使用到索引，但是key1 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：

  - 先根据key1 > 'z'这个条件，从二级索引idx_key1中获取到对应的二级索引记录。
  - 根据上一步骤得到的二级索引记录中的主键值进行回表，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key1 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key1列，所以设计MySQL的大叔把上边的步骤改进了一下：

  - 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。
  - 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key1 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。
  - 对于满足key1 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，所以上述修改虽然只改进了一点点，但是可以省去好多回表操作的成本。设计MySQL的大叔们把他们的这个改进称之为索引条件下推（英文名：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 key1 > 'z' AND key1 LIKE '%b';
  +----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
  | id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
  +----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
  |  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  266 |   100.00 | Using index condition |
  +----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
  1 row in set, 1 warning (0.01 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。比如下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE common_field = '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 | 9688 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.01 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列中也会提示上述额外信息。比如下边这个查询虽然使用idx_key1索引执行查询，但是搜索条件中除了包含key1的搜索条件key1 = 'a'，还有包含common_field的搜索条件，所以Extra列会显示Using where的提示：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' AND common_field = '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 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
```

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

- 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)
```

可以在对s2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s2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s2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.common_field = s2.common_field条件，因为s1是驱动表，s2是被驱动表，所以在访问s2表时，s1.common_field的值已经确定下来了，所以实际上查询s2表的条件就是s2.common_field = 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：

```
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 |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------------------+
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，这意味着必定是驱动表的记录在被驱动表中找不到匹配ON子句条件的记录才会把该驱动表的记录加入到最终的结果集，所以对于某条驱动表中的记录来说，如果能在被驱动表中找到1条符合ON子句条件的记录，那么该驱动表的记录就不会被加入到最终的结果集，也就是说我们没有必要到被驱动表中找到全部符合ON子句条件的记录，这样可以稍微节省一点性能。

> 小贴士：右（外）连接可以被转换为左（外）连接，所以就不提右（外）连接的情况了。

- 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。比如这个查询的执行计划：

```
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 w
+----+-------------+-------+------------+-------------+-----------------------+-----------------------+---------+------+------+----------+-----------------------------------------------------
1 row in set, 1 warning (0.01 sec)
```

其中Extra列就显示了Using intersect(idx_key3,idx_key1)，表明MySQL即将使用idx_key3和idx_key1这两个索引进行Intersect索引合并的方式执行查询。

> 小贴士：剩下两种类型的索引合并的Extra列信息就不一一举例子了，自己写个查询瞅瞅呗～

- 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，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 LIMIT 0;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------+
| 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 | Zero limit |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------+
1 row in set, 1 warning (0.00 sec)
```

- 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)
```

这个查询语句可以利用idx_key1索引直接取出key1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，设计MySQL的大叔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（英文名：filesort）。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，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 ORDER BY common_field 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 | 9688 |   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，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)
```

再比如：

```
mysql> EXPLAIN SELECT common_field, COUNT(*) AS amount FROM s1 GROUP BY 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 | Using temporary; Using filesort |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+---------------------------------+
1 row in set, 1 warning (0.00 sec)
```

不知道大家注意到没有，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 common_field, COUNT(*) AS amount FROM s1 GROUP BY common_field ORDER BY common_field;
```

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，就像这样：

```
mysql> EXPLAIN SELECT common_field, COUNT(*) AS amount FROM s1 GROUP BY common_field ORDER BY 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 | Using temporary |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
1 row in set, 1 warning (0.00 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。

另外，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：

```
mysql> EXPLAIN SELECT key1, COUNT(*) AS amount FROM s1 GROUP BY key1;
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9688 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
1 row in 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_key1索引就可以搞定了，不再需要临时表了。

- Start temporary, End temporary

我们前边唠叨子查询的时候说过，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)
```

## Json格式的执行计划

我们上边介绍的EXPLAIN语句输出中缺少了一个衡量执行计划好坏的重要属性 —— 成本。不过设计MySQL的大叔贴心的为我们提供了一种查看某个执行计划花费的成本的方式：

- 在EXPLAIN单词和真正的查询语句中间加上FORMAT=JSON。

这样我们就可以得到一个json格式的执行计划，里边儿包含该计划花费的成本，比如这样：

```
mysql> EXPLAIN FORMAT=JSON SELECT * FROM s1 INNER JOIN s2 ON s1.key1 = s2.key2 WHERE s1.common_field = 'a'\G
*************************** 1. row ***************************

EXPLAIN: {
  "query_block": {
    "select_id": 1,      # 整个查询语句只有1个SELECT关键字，该关键字对应的id号为1
    "cost_info": {
      "query_cost": "3197.16"    # 整个查询的执行成本预计为3197.16
    },
    "nested_loop": [     # 几个表之间采用嵌套循环连接算法执行

    # 以下是参与嵌套循环连接算法的各个表的信息
      {
        "table": {
          "table_name": "s1",    # s1表是驱动表
          "access_type": "ALL",      # 访问方法为ALL，意味着使用全表扫描访问
          "possible_keys": [     # 可能使用的索引
            "idx_key1"
          ],
          "rows_examined_per_scan": 9688,    # 查询一次s1表大致需要扫描9688条记录
          "rows_produced_per_join": 968,     # 驱动表s1的扇出是968
          "filtered": "10.00",  # condition filtering代表的百分比
          "cost_info": {
            "read_cost": "1840.84",      # 稍后解释
            "eval_cost": "193.76",       # 稍后解释
            "prefix_cost": "2034.60",    # 单次查询s1表总共的成本
            "data_read_per_join": "1M"  # 读取的数据量
          },
          "used_columns": [      # 执行查询中涉及到的列
            "id",
            "key1",
            "key2",
            "key3",
            "key_part1",
            "key_part2",
            "key_part3",
            "common_field"
          ],

          # 对s1表访问时针对单表查询的条件
          "attached_condition": "((`xiaohaizi`.`s1`.`common_field` = 'a') and (`xiaohaizi`.`s1`.`key1` is not null))"
        }
      },
      {
        "table": {
          "table_name": "s2",    # s2表是被驱动表
          "access_type": "ref",      # 访问方法为ref，意味着使用索引等值匹配的方式访问
          "possible_keys": [     # 可能使用的索引
```

```
            "idx_key2"
          ],
          "key": "idx_key2",     # 实际使用的索引
          "used_key_parts": [    # 使用到的索引列
            "key2"
          ],
          "key_length": "5",     # key_len
          "ref": [       # 与key2列进行等值匹配的对象
            "xiaohaizi.s1.key1"
          ],
          "rows_examined_per_scan": 1,  # 查询一次s2表大致需要扫描1条记录
          "rows_produced_per_join": 968,     # 被驱动表s2的扇出是968（由于后边没有多余的表进行连接，所以这个值也没啥用）
          "filtered": "100.00",      # condition filtering代表的百分比

          # s2表使用索引进行查询的搜索条件
          "index_condition": "(`xiaohaizi`.`s1`.`key1` = `xiaohaizi`.`s2`.`key2`)",
          "cost_info": {
            "read_cost": "968.80",       # 稍后解释
            "eval_cost": "193.76",       # 稍后解释
            "prefix_cost": "3197.16",    # 单次查询s1、多次查询s2表总共的成本
            "data_read_per_join": "1M"  # 读取的数据量
          },
          "used_columns": [      # 执行查询中涉及到的列
            "id",
            "key1",
            "key2",
            "key3",
            "key_part1",
            "key_part2",
            "key_part3",
            "common_field"
          ]
        }
      }
    ]
  }
}
1 row in set, 2 warnings (0.00 sec)
```

我们使用#后边跟随注释的形式为大家解释了EXPLAIN FORMAT=JSON语句的输出内容，但是大家可能有疑问"cost_info"里边的成本看着怪怪的，它们是怎么计算出来的？先看s1表的"cost_info"部分：

```
"cost_info": {
    "read_cost": "1840.84",
    "eval_cost": "193.76",
    "prefix_cost": "2034.60",
    "data_read_per_join": "1M"
}
```

- read_cost是由下边这两部分组成的：
  - IO成本
  - 检测rows × (1 - filter)条记录的CPU成本

  小贴士： rows和filter都是我们前边介绍执行计划的输出列，在JSON格式的执行计划中，rows相当于rows_examined_per_scan，filtered名称不变。

- eval_cost是这样计算的：

  检测 rows × filter条记录的成本。

- prefix_cost就是单独查询s1表的成本，也就是：

  read_cost + eval_cost

- data_read_per_join表示在此次查询中需要读取的数据量，我们就不多唠叨这个了。

  小贴士： 大家其实没必要关注MySQL为啥使用这么古怪的方式计算出read_cost和eval_cost，关注prefix_cost是查询s1表的成本就好了。

对于s2表的"cost_info"部分是这样的：

```
"cost_info": {
    "read_cost": "968.80",
    "eval_cost": "193.76",
    "prefix_cost": "3197.16",
    "data_read_per_join": "1M"
}
```

由于s2表是被驱动表，所以可能被读取多次，这里的read_cost和eval_cost是访问多次s2表后累加起来的值，大家主要关注里边儿的prefix_cost的值代表的是整个连接查询预计的成本，也就是单次查询s1表和多次查询s2表后的成本的和，也就是：

```
968.80 + 193.76 + 2034.60 = 3197.16
```

## Extented EXPLAIN

最后，设计MySQL的大叔还为我们留了个彩蛋，在我们使用EXPLAIN语句查看了某个查询的执行计划后，紧接着还可以使用SHOW WARNINGS语句查看与这个查询的执行计划有关的一些扩展信息，比如这样：

```
mysql> EXPLAIN SELECT s1.key1, s2.key1 FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 WHERE s2.common_field IS NOT NULL;
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref               | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key1      | NULL     | NULL    | NULL              | 9954 |    90.00 | Using where |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s2.key1 |    1 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `xiaohaizi`.`s1`.`key1` AS `key1`,`xiaohaizi`.`s2`.`key1` AS `key1` from `xiaohaizi`.`s1` join `xiaohaizi`.`s2` where ((`xiaohaizi`.`s1`.`key1` = `:
1 row in set (0.00 sec)
```

大家可以看到SHOW WARNINGS展示出来的信息有三个字段，分别是Level、Code、Message。我们最常见的就是Code为1003的信息，当Code值为1003时，Message字段展示的信息类似于查询优化器将我们的查询语句重写后的语句。比如我们上边的查询本来是一个左（外）连接查询，但是有一个s2.common_field IS NOT NULL的条件，着就会导致查询优化器把左（外）连接查询优化为内连接查询，从SHOW WARNINGS的Message字段也可以看出来，原本的LEFT JOIN已经变成了JOIN。

但是大家一定要注意，我们说Message字段展示的信息类似于查询优化器将我们的查询语句重写后的语句，并不是等价于，也就是说Message字段展示的信息并不是标准的查询语句，在很多情况下并不能直接拿到黑框框中运行，它只能作为帮助我们理解查MySQL将如何执行查询语句的一个参考依据而已。