第17章、optimizer trace 表的神奇功效

标签： MySQL 是怎样运行的

对于MySQL 5.6以及之前的版本来说，查询优化器就像是一个黑盒子一样，你只能通过EXPLAIN语句查看到最后优化器决定使用的执行计划，却无法知道它为什么做这个决策。这对于一部分喜欢刨根问底的小伙伴来说简直是灾难： “我就觉得使用其他的执行方案比EXPLAIN输出的这种方案强，凭什么优化器做的决定和我想的不一样呢？”

在MySQL 5.6以及之后的版本中，设计MySQL的大叔贴心的为这部分小伙伴提出了一个optimizer trace的功能，这个功能可以让我们方便的查看优化器生成执行计划的整个过程，这个功能的开启与关闭由系统变量optimizer_trace决定，我们看一下：

```
mysql> SHOW VARIABLES LIKE 'optimizer_trace';
+-----------------+--------------------------+
| Variable_name   | Value                    |
+-----------------+--------------------------+
| optimizer_trace | enabled=off,one_line=off |
+-----------------+--------------------------+
1 row in set (0.02 sec)
```

可以看到enabled值为off，表明这个功能默认是关闭的。

小贴士： one_line的值是控制输出格式的，如果为on那么所有输出都将在一行中展示，不适合人阅读，所以我们就保持其默认值为off吧。

如果想打开这个功能，必须首先把enabled的值改为on，就像这样：

```
mysql> SET optimizer_trace="enabled=on";
Query OK, 0 rows affected (0.00 sec)
```

然后我们就可以输入我们想要查看优化过程的查询语句，当该查询语句执行完成后，就可以到information_schema数据库下的OPTIMIZER_TRACE表中查看完整的优化过程。这个OPTIMIZER_TRACE表有4个列，分别是：

- QUERY：表示我们的查询语句。
- TRACE：表示优化过程的JSON格式文本。
- MISSING_BYTES_BEYOND_MAX_MEM_SIZE：由于优化过程可能会输出很多，如果超过某个限制时，多余的文本将不会被显示，这个字段展示了被忽略的文本字节数。
- INSUFFICIENT_PRIVILEGES：表示是否没有权限查看优化过程，默认值是0，只有某些特殊情况下才会是1，我们暂时不关心这个字段的值。

完整的使用optimizer trace功能的步骤总结如下：

```
# 1. 打开optimizer trace功能 (默认情况下它是关闭的):
SET optimizer_trace="enabled=on";

# 2. 这里输入你自己的查询语句
SELECT ...;

# 3. 从OPTIMIZER_TRACE表中查看上一个查询的优化过程
SELECT * FROM information_schema.OPTIMIZER_TRACE;

# 4. 可能你还要观察其他语句执行的优化过程，重复上边的第2、3步
...

# 5. 当你停止查看语句的优化过程时，把optimizer trace功能关闭
SET optimizer_trace="enabled=off";
```

现在我们有一个搜索条件比较多的查询语句，它的执行计划如下：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE
    ->     key1 > 'z' AND
    ->     key2 < 1000000 AND
    ->     key3 IN ('a', 'b', 'c') AND
    ->     common_field = 'abc';
+----+-------------+-------+------------+-------+---------------------------+----------+---------+------+------+----------+-----------------------------------+
| id | select_type | table | partitions | type  | possible_keys             | key      | key_len | ref  | rows | filtered | Extra                             |
+----+-------------+-------+------------+-------+---------------------------+----------+---------+------+------+----------+-----------------------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key2,idx_key1,idx_key3 | idx_key2 | 5       | NULL |   12 |     0.42 | Using index condition; Using where |
+----+-------------+-------+------------+-------+---------------------------+----------+---------+------+------+----------+-----------------------------------+
1 row in set, 1 warning (0.00 sec)
```

可以看到该查询可能使用到的索引有3个，那么为什么优化器最终选择了idx_key2而不选择其他的索引或者直接全表扫描呢？这时候就可以通过otpimzer trace功能来查看优化器的具体工作过程：

```
SET optimizer_trace="enabled=on";

SELECT * FROM s1 WHERE
    key1 > 'z' AND
    key2 < 1000000 AND
    key3 IN ('a', 'b', 'c') AND
    common_field = 'abc';

SELECT * FROM information_schema.OPTIMIZER_TRACE\G
```

我们直接看一下通过查询OPTIMIZER_TRACE表得到的输出（我使用#后跟随注释的形式为大家解释了优化过程中的一些比较重要的点，大家重点关注一下）：

```
*************************** 1. row ***************************
# 分析的查询语句是什么
QUERY: SELECT * FROM s1 WHERE
    key1 > 'z' AND
    key2 < 1000000 AND
    key3 IN ('a', 'b', 'c') AND
    common_field = 'abc'

# 优化的具体过程
TRACE: {
  "steps": [
    {
      "join_preparation": {     # prepare阶段
        "select#": 1,
        "steps": [
          {
            "IN_uses_bisection": true
          },
          {
            "expanded_query": "/* select#1 */ select `s1`.`id` AS `id`,`s1`.`key1` AS `key1`,`s1`.`key2` AS `key2`,`s1`.`key3` AS `key3`,`s1`.`key_part1` AS `key
 AS `key_part2`,`s1`.`key_part3` AS `key_part3`,`s1`.`common_field` AS `common_field` from `s1` where ((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.
and (`s1`.`common_field` = 'abc'))"
          }
        ] /* steps */
      } /* join_preparation */
    },
    {
      "join_optimization": {    # optimize阶段
        "select#": 1,
        "steps": [
          {
            "condition_processing": {   # 处理搜索条件
              "condition": "WHERE",
              # 原始搜索条件
              "original_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))",
              "steps": [
                {
                  # 等值传递转换
                  "transformation": "equality_propagation",
                  "resulting_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
                },
                {
                  # 常量传递转换
                  "transformation": "constant_propagation",
                  "resulting_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
                },
                {
                  # 去除没用的条件
```

```
                "transformation": "trivial_condition_removal",
                "resulting_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
              }
            ] /* steps */
          } /* condition_processing */
        },
        {
          # 替换虚拟生成列
          "substitute_generated_columns": {
          } /* substitute_generated_columns */
        },
        {
          # 表的依赖信息
          "table_dependencies": [
            {
              "table": "`s1`",
              "row_may_be_null": false,
              "map_bit": 0,
              "depends_on_map_bits": [
              ] /* depends_on_map_bits */
            }
          ] /* table_dependencies */
        },
        {
          "ref_optimizer_key_uses": [
          ] /* ref_optimizer_key_uses */
        },
        {

          # 预估不同单表访问方法的访问成本
          "rows_estimation": [
            {
              "table": "`s1`",
              "range_analysis": {
                "table_scan": {   # 全表扫描的行数以及成本
                  "rows": 9688,
                  "cost": 2036.7
                } /* table_scan */,

                # 分析可能使用的索引
                "potential_range_indexes": [
                  {
                    "index": "PRIMARY",   # 主键不可用
                    "usable": false,
                    "cause": "not_applicable"
                  },
                  {
                    "index": "idx_key2",  # idx_key2可能被使用
                    "usable": true,
                    "key_parts": [
                      "key2"
                    ] /* key_parts */
                  },
                  {
                    "index": "idx_key1",  # idx_key1可能被使用
                    "usable": true,
                    "key_parts": [
                      "key1",
                      "id"
                    ] /* key_parts */
                  },
                  {
```

```
            "index": "idx_key3",  # idx_key3可能被使用
            "usable": true,
            "key_parts": [
              "key3",
              "id"
            ] /* key_parts */
          },
          {
            "index": "idx_key_part",  # idx_keypart不可用
            "usable": false,
            "cause": "not_applicable"
          }
        ] /* potential_range_indexes */,
        "setup_range_conditions": [
        ] /* setup_range_conditions */,
        "group_index_range": {
          "chosen": false,
          "cause": "not_group_by_or_distinct"
        } /* group_index_range */,

        # 分析各种可能使用的索引的成本
        "analyzing_range_alternatives": {
          "range_scan_alternatives": [
            {
              # 使用idx_key2的成本分析
              "index": "idx_key2",
              # 使用idx_key2的范围区间
              "ranges": [
                "NULL < key2 < 1000000"
              ] /* ranges */,
              "index_dives_for_eq_ranges": true,   # 是否使用index dive
              "rowid_ordered": false,     # 使用该索引获取的记录是否按照主键排序
              "using_mrr": false,     # 是否使用mrr
              "index_only": false,    # 是否是索引覆盖访问
              "rows": 12,     # 使用该索引获取的记录条数
              "cost": 15.41,  # 使用该索引的成本
              "chosen": true  # 是否选择该索引
            },
            {
              # 使用idx_key1的成本分析
              "index": "idx_key1",
              # 使用idx_key1的范围区间
              "ranges": [
                "z < key1"
              ] /* ranges */,
              "index_dives_for_eq_ranges": true,   # 同上
              "rowid_ordered": false,   # 同上
              "using_mrr": false,   # 同上
              "index_only": false,   # 同上
              "rows": 266,   # 同上
              "cost": 320.21,   # 同上
              "chosen": false,   # 同上
              "cause": "cost"   # 因为成本太大所以不选择该索引
            },
            {
              # 使用idx_key3的成本分析
              "index": "idx_key3",
              # 使用idx_key3的范围区间
              "ranges": [
                "a <= key3 <= a",
                "b <= key3 <= b",
                "c <= key3 <= c"
```

```
                        ] /* ranges */,
                        "index_dives_for_eq_ranges": true,   # 同上
                        "rowid_ordered": false,   # 同上
                        "using_mrr": false,   # 同上
                        "index_only": false,   # 同上
                        "rows": 21,   # 同上
                        "cost": 28.21,   # 同上
                        "chosen": false,   # 同上
                        "cause": "cost"   # 同上
                      }
                    ] /* range_scan_alternatives */,

                    # 分析使用索引合并的成本
                    "analyzing_roworder_intersect": {
                      "usable": false,
                      "cause": "too_few_roworder_scans"
                    } /* analyzing_roworder_intersect */
                  } /* analyzing_range_alternatives */,

                  # 对于上述单表查询s1最优的访问方法
                  "chosen_range_access_summary": {
                    "range_access_plan": {
                      "type": "range_scan",
                      "index": "idx_key2",
                      "rows": 12,
                      "ranges": [
                        "NULL < key2 < 1000000"
                      ] /* ranges */
                    } /* range_access_plan */,
                    "rows_for_plan": 12,
                    "cost_for_plan": 15.41,
                    "chosen": true
                  } /* chosen_range_access_summary */
                } /* range_analysis */
              }
            ] /* rows_estimation */
          },
          {

            # 分析各种可能的执行计划
            #（对多表查询这可能有很多种不同的方案，单表查询的方案上边已经分析过了，直接选取idx_key2就好）
            "considered_execution_plans": [
              {
                "plan_prefix": [
                ] /* plan_prefix */,
                "table": "`s1`",
                "best_access_path": {
                  "considered_access_paths": [
                    {
                      "rows_to_scan": 12,
                      "access_type": "range",
                      "range_details": {
                        "used_index": "idx_key2"
                      } /* range_details */,
                      "resulting_rows": 12,
                      "cost": 17.81,
                      "chosen": true
                    }
                  ] /* considered_access_paths */
                } /* best_access_path */,
                "condition_filtering_pct": 100,
                "rows_for_plan": 12,
```

```
              "cost_for_plan": 17.81,
              "chosen": true
            }
          ] /* considered_execution_plans */
        },
        {
          # 尝试给查询添加一些其他的查询条件
          "attaching_conditions_to_tables": {
            "original_condition": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))",
            "attached_conditions_computation": [
            ] /* attached_conditions_computation */,
            "attached_conditions_summary": [
              {
                "table": "`s1`",
                "attached": "((`s1`.`key1` > 'z') and (`s1`.`key2` < 1000000) and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
              }
            ] /* attached_conditions_summary */
          } /* attaching_conditions_to_tables */
        },
        {
          # 再稍稍的改进一下执行计划
          "refine_plan": [
            {
              "table": "`s1`",
              "pushed_index_condition": "(`s1`.`key2` < 1000000)",
              "table_condition_attached": "((`s1`.`key1` > 'z') and (`s1`.`key3` in ('a','b','c')) and (`s1`.`common_field` = 'abc'))"
            }
          ] /* refine_plan */
        }
      ] /* steps */
    } /* join_optimization */
  },
  {
    "join_execution": {    # execute阶段
      "select#": 1,
      "steps": [
      ] /* steps */
    } /* join_execution */
  }
] /* steps */
}

# 因优化过程文本太多而丢弃的文本字节大小，值为0时表示并没有丢弃
MISSING_BYTES_BEYOND_MAX_MEM_SIZE: 0

# 权限字段
INSUFFICIENT_PRIVILEGES: 0

1 row in set (0.00 sec)
```

大家看到这个输出的第一感觉就是这文本也太多了点儿吧，其实这只是优化器执行过程中的一小部分，设计MySQL的大叔可能会在之后的版本中添加更多的优化过程信息。不过杂乱之中其实还是蛮有规律的，优化过程大致分为了三个阶段：

- prepare阶段
- optimize阶段
- execute阶段

我们所说的基于成本的优化主要集中在optimize阶段，对于单表查询来说，我们主要关注optimize阶段的"rows_estimation"这个过程，这个过程深入分析了对单表查询的各种执行方案的成本；对于多表连接查询来说，我们更多需要关注"considered_execution_plans"这个过程，这个过程里会写明各种不同的连接方式所对应的成本。反正优化器最终会选择成本最低的那种方案来作为最终的执行计划，也就是我们使用EXPLAIN语句所展现出的那种方案。

如果有小伙伴对使用EXPLAIN语句展示出的对某个查询的执行计划很不理解，大家可以尝试使用optimizer trace功能来详细了解每一种执行方案对应的成本，相信这个功能能让大家更深入的了解MySQL查询优化器。