

## 第 53 期：EXPLAIN 中最直观的 rows

原创 杨涛涛 爱可生开源社区 2025年03月20日 18:00 上海

作者：杨涛涛，爱可生技术专家。

爱可生开源社区出品，原创内容未经授权不得随意使用，转载请联系小编并注明来源。



MySQL 和大多数关系型数据库一样，SQL 语句执行计划的输出栏都有一行 **rows**，代表优化器执行这条 SQL 所需算子扫描的记录数，是优化器根据表和索引的统计信息数据评估出来的结果。

### 如何根据 rows 值的大小判断 SQL 性能？

对于大多数场景来讲，可以直接凭借 **rows** 值的大小来判断 SQL 语句性能的高低，但也不能一概而论。

本篇就通过几个简单的示例，来列举三种 **rows** 值判断的情况。

- rows 值小，性能高
- rows 值小，性能不一定
- 不适合看 rows 值

### rows 值小，性能高

第一种情况就是同一条 SQL，只是用到索引不同，rows 值越小，SQL 性能越高。

示例 SQL：

```
select * from t1 where r1=2 and r2=2
```

如果不考虑真实业务逻辑，单从写法上来讲，这条 SQL 已经无法优化，因为已经足够简单。优化策略可简单的定义为过滤字段是否匹配索引、匹配的索引是否足够好的问题。比如可能有如下四种索引被用到：

`idx_r1(r1) / idx_r2(r2) / idx_u1(r1,r2) / idx_u2(r2,r1)`

对于以上几个索引，MySQL 可以根据统计信息、数据物理分布、成本模型等选择使用以上四个索引中任意一个，或者直接使用 INDEX MERGE 算法来选择合适的索引组合。

这种情况下，要看哪种索引对这条 SQL 最高效，除了之前介绍过的查看索引本身的数据外，还可以从执行计划的 rows 值直接来判断。

我们使用 `force index` 来指定优化器强制匹配不同的索引，来看这四个索引对应不同执行计划的 rows 值。

```
localhost:ytt>desc select * from t1 force index (idx_r1) where r1 = 2 and r2 = 2\G
***** 1. row *****

      id: 1
select_type: SIMPLE
      table: t1
partitions: NULL
      type: ref
possible_keys: idx_r1
         key: idx_r1
      key_len: 5
         ref: const
        rows: 18638
   filtered: 0.10
      Extra: Using where
1 row in set, 1 warning (0.00 sec)

localhost:ytt>desc select * from t1 force index (idx_r2) where r1 = 2 and r2 = 2\G
***** 1. row *****

...

      rows: 102
   filtered: 11.11
      Extra: Using where
1 row in set, 1 warning (0.00 sec)

localhost:ytt>desc select * from t1 force index (idx_u1) where r1 = 2 and r2 = 2\G
***** 1. row *****

...

      rows: 12
   filtered: 100.00
      Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

```
localhost:ytt>desc select * from t1 force index (idx_u2) where r1 = 2 and r2 = 2\G
***** 1. row *****
...
      rows: 12
    filtered: 100.00
      Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

根据查询计划可知（索引：rows 值）：

- idx\_r1: 18638
- idx\_r2: 102
- idx\_u1: 12
- idx\_u2: 12

很明显，在此场景下走联合索引 `idx_u1/idx_u2` 扫描记录数最小，效率最高。

## rows 值小，性能不一定

有些情况下，不能简单通过 `rows` 值作为判断 SQL 是否高效执行的标准。

示例 SQL：

```
select * from t1 where r1<5
```

这条 SQL 也很简单，就是对 `r1` 进行一个范围过滤完后取结果。依照之前文章里讲的，对于这样的查询，有时候不走索引反而效率更高，虽然单从走索引扫描的 `rows` 值一定会更小。来看下两条不同的执行计划：

```
localhost:ytt>desc select * from t1 where r1 <5 \G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: t1
    partitions: NULL
      type: ALL
possible_keys: idx_r1,idx_u1
      key: NULL
     key_len: NULL
        ref: NULL
       rows: 101745
    filtered: 50.00
      Extra: Using where
```

```

1 row in set, 1 warning (0.01 sec)

localhost:ytt>desc select * from t1 force index (idx_r1) where r1 <5 \G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: t1
   partitions: NULL
         type: range
possible_keys: idx_r1
          key: idx_r1
        key_len: 5
         ref: NULL
        rows: 50872
   filtered: 100.00
      Extra: Using index condition
1 row in set, 1 warning (0.00 sec)

```

根据查询计划可知：

- 执行计划 1 的 rows 值：101745
- 执行计划 2 的 rows 值：50872

如果仅从 rows 值来判断，那第二个执行计划更优，但事实并非如此。

MySQL 自主选择了第一个执行计划（全表扫描）。其实就是优化器基于一定的数据基础评估，走全表扫的成本要比走索引后再来回表来的更优化。

为了继续验证我们的判断，查看 **EXPLAIN ANALYZE** 结果：

```

localhost:ytt>desc analyze select * from t1 where r1 <5 \G
***** 1. row *****

EXPLAIN: -> Filter: (t1.r1 < 5) (cost=10262.75 rows=50872) (actual time=0.044..104.673 r
      -> Table scan on t1 (cost=10262.75 rows=101745) (actual time=0.041..90.898 rows=1018

1 row in set (0.12 sec)

localhost:ytt>desc analyze select * from t1 force index (idx_r1) where r1 <5 \G
***** 1. row *****

EXPLAIN: -> Index range scan on t1 using idx_r1, with index condition: (t1.r1 < 5) (cost

1 row in set (0.16 sec)

```

结果很明显，走全表扫无论成本和最终时间都比走索引有优势。

## 不适合看 rows 值

前两个情况都是基于单表检索，我们再来看下多表联接的例子。

示例 SQL：

```
select a.* from t1 a join t2 b using(f0,f1)
```

这条 SQL 没有过滤条件，仅仅是两表内联，而且表 `t1` 有 10W 行记录，表 `t2` 只有 5W 行记录。正常情况，应该走基于主键的 NLJ 算法，表 `t2` 驱动表 `t1`。

来看下执行计划：

```
localhost:ytt>desc select a.* from t1 a join t2 b using(f0,f1)\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: a
  partitions: NULL
        type: ALL
    ...
      rows: 101745
  filtered: 100.00
    Extra: NULL
***** 2. row *****
      id: 1
  select_type: SIMPLE
        table: b
  partitions: NULL
        type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 8
         ref: ytt.a.f0,ytt.a.f1
        rows: 1
  filtered: 100.00
    Extra: Using index
2 rows in set, 1 warning (0.00 sec)
```

从执行计划结果来看，表 `t1` 被放在驱动表的位置，rows 值显示需要扫 10W 行记录（全表扫描）；表 `t2` 随后作为被驱动表来检索（走主键），对于表 `t2` 的效率很高。

这个结果和我们的认知刚好相反（表 `t2` 的扫描行数仅仅是针对 NLJ 算法的内表来讲，每次扫描的行数，而不是整体扫描的行数），并且两表 JOIN 的顺序不对，我们强制手动收集统计信息再次进行优化：

```
localhost:ytt>analyze table t1,t2;

+-----+-----+-----+-----+
| Table | Op      | Msg_type | Msg_text |
+-----+-----+-----+-----+
| ytt.t1 | analyze | status   | OK       |
| ytt.t2 | analyze | status   | OK       |
+-----+-----+-----+-----+

2 rows in set (0.43 sec)
```

收集完后，再次查看执行计划：

```
localhost:ytt>desc select a.* from t1 a join t2 b using(f0,f1)\G

***** 1. row *****

      id: 1
select_type: SIMPLE
      table: b
  partitions: NULL
        type: index
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 8
         ref: NULL
        rows: 48339
   filtered: 100.00
      Extra: Using index

***** 2. row *****

      id: 1
select_type: SIMPLE
      table: a
  partitions: NULL
        type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 8
         ref: ytt.b.f0,ytt.b.f1
        rows: 1
   filtered: 100.00
      Extra: NULL

2 rows in set, 1 warning (0.00 sec)
```

两表执行顺序做了置换，并且总体的 rows 值都变小；表 `t2` 为驱动表，rows 值接近 5W，表 `t1` 做为被驱动表进行内部判断。

## 总结

在不同的情况下，执行计划 **rows** 值展示出来的信息有不同的参考价值，并不能直接作为 SQL 高效与否的判断标准。


MySQL 的 SQL 到底是走何种执行计划，与执行计划成本模型、表统计信息、索引统计信息、表的数据分布等都有关系，不能仅凭执行计划 rows 值的大小来判断，需要这些因素来综合决定一个最优的执行计划。

## 往期内容

数据类型 | 大对象字段 | 列非空与自增 | 外键 | 字符集1 | 字符集2 | 字符集3 | 字符集4 | 表空间 | 压缩表1 | 压缩表2 | 表统计 | 页合并 | B+树 | 索引结构 | 主键设计 | 哈希表1 | 哈希表2 | 前缀索引 | 函数索引 | 组合索引1 | 组合索引2 | 多值索引 | 索引基数 | 索引下推 | 全文索引1 | 全文索引2 | 全文索引3 | 全文索引4 | 索引数量 | 索引设计 | 表标准化设计 | 表冗余设计 | 垂直拆分 | 水平分表 | 原生表分区 | 时间分区 | 分区案例 | 哈希分区 | 多列分区 | 多表关联分区 | 无主键分区 | SQL 优化思路 | 执行计划1 | 执行计划2 | 执行计划3 | 执行计划4 | 执行计划5 | 执行计划6 | 执行计划 7

✨ Github : <https://github.com/actiontech/sqlc>  
📖 文档 : <https://actiontech.github.io/sqlc-docs/>  
🌐 官网 : <https://opensource.actionsky.com/sqlc/>



 微信群：请添加小助手加入 ActionOpenSource

 商业支持：<https://www.actionsky.com/sql>

SQL 优化 16    MySQL 231    EXPLAIN 9    执行计划 14    索引调优 8

SQL 优化 · 目录

上一篇

第 52 期：根据 EXPLAIN EXTRA 栏提示进行优化（四）

下一篇

第 54 期：使用 JSON 格式的执行计划优化 SQL

[阅读原文](#)