浅析MySQL代价模型：告别盲目使用EXPLAIN，提前预知索引优化策略|京东零售技术团队

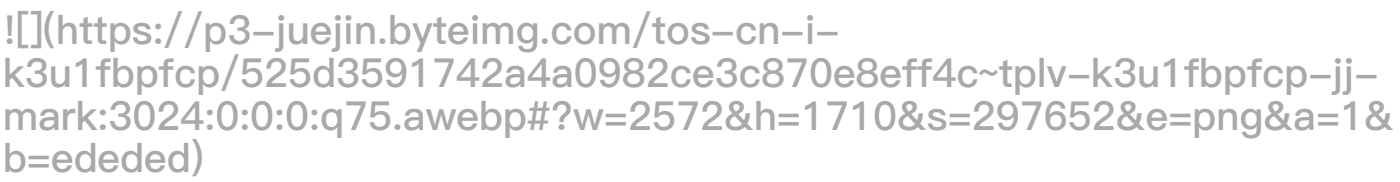
背景

在 MySQL 中，当我们为表创建了一个或多个索引后，通常需要在索引定义完成后，根据具体的数据情况执行 EXPLAIN 命令，才能观察到数据库实际使用哪个索引、是否使用索引。这使得我们在添加新索引之前，无法提前预知数据库是否能使用期望的索引。更为糟糕的是，有时甚至在添加新的索引后，数据库在某些查询中会使用它，而在其他查询中则不会使用，这种情况下，我们无法确定索引是否发挥了预期的作用，让人感到非常苦恼。这种情况基本上意味着 MySQL 并没有为我们选择最优的索引，而我们不得不在茫茫数据中摸索，试图找到问题的症结所在。我们可能会尝试调整索引，甚至删除索引，然后重新添加，希望 MySQL 能从中找到最优的索引选择。然而，这样的过程既耗时又费力，而且往往收效甚微。

如果在添加索引之前，我们能够预知索引的使用情况，那么对于表设计将大有裨益。我们可以在设计表结构时，更加明确地知道应该选择哪些索引，如何优化索引，以提高查询效率。我们不再需要依赖盲目尝试和猜测，而是可以基于实际的数据和查询情况，做出更加明智的决策。因此，对于 MySQL 用户来说，能够预知索引走势的需求非常迫切。我们希望能有一种方法，能够让我们在添加索引之前，就清楚地了解 MySQL 将如何使用索引，以便我们能够更好地优化表结构，提高查询效率。这将极大地减轻我们的工作负担，提高我们的工作效率，让我们能够更加专注于业务逻辑的处理，而不是在索引的海洋中挣扎。

为了解决这个问题，我们可以深入研究 MySQL 的索引选择机制。实际上，这个机制的核心就是代价模型，它通过一个公式来决定索引的选择策略。相对于 MySQL 其他复杂的概念，代价模型实现起来要简单得多。熟悉代价模型之后，我们可以预先了解 MySQL 在执行查询时会如何选择索引，从而更有效地进行索引优化。在接下来的文章中，我将结合近期进行索引优化的具体案例，来详细解释如何运用代价模型来优化索引。

MySQL代价模型浅析



MySQL数据库主要由4层组成：

1.连接层：客户端和连接服务，主要完成一些类似于连接处理、授权管理、以及相关的安全方案。

2.服务层：主要完成大多数的核心服务功能，如SQL接口，并完成缓存的查询，SQL的分析和优化以及内部函数的执行。

3.引擎层：负责MySQL中数据的存储和提取，服务器通过AP1与存储引擎进行通信。

4.存储层：将数据存储文件系统上，并完成与存储引擎的交互。

索引策略选择在SQL优化器进行的

SQL 优化器会分析所有可能的执行计划，选择成本最低的执行，这种优化器称之为：CBO（Cost-based Optimizer，基于成本的优化器）。

Cost = Server Cost + Engine Cost = CPU Cost + IO Cost

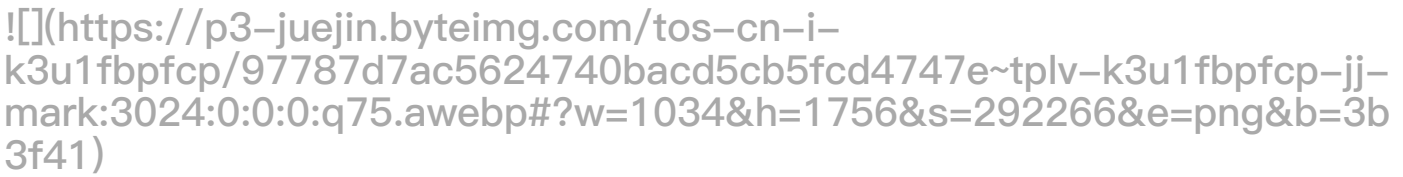
其中，CPU Cost 表示计算的开销，比如索引键值的比较、记录值的比较、结

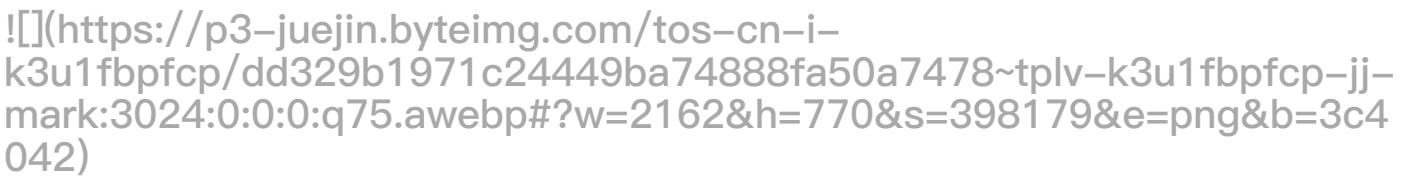
果集的排序 ..... 这些操作都在 Server 层完成；

IO Cost 表示引擎层 IO 的开销，MySQL 可以通过区分一张表的数据是否在内存中，分别计算读取内存 IO 开销以及读取磁盘 IO 的开销。

源码简读

MySQL的数据源代码采用了5.7.22版本，后续的代价计算公式将基于此版本进行参考。





opt_costconstants.cc【代价模型——计算所需代价计算系数】

```
/*
```

```
  在Server_cost_constants类中定义为静态常量变量的成本常量的值。如果服务器管理员没有在server_cost表中添加新值，则将使用这些默认成本常数值。
  5.7版本开始可用从数据库加载常量值，该版本前使用代码中写的常量值
*/

// 计算符合条件的行的代价，行数越多，此项代价越大
const double Server_cost_constants::ROW_EVALUATE_COST= 0.2;

// 键比较的代价，例如排序
const double Server_cost_constants::KEY_COMPARE_COST= 0.1;

/*
   内存临时表的创建代价
   通过基准测试，创建Memory临时表的成本与向表中写入10行的成本一样高。
*/
const double
Server_cost_constants::MEMORY_TEMPTABLE_CREATE_COST= 2.0;

// 内存临时表的行代价
const double
Server_cost_constants::MEMORY_TEMPTABLE_ROW_COST= 0.2;

/*
  内部myisam或innodb临时表的创建代价
  创建MyISAM表的速度是创建Memory表的20倍。
*/
const double
Server_cost_constants::DISK_TEMPTABLE_CREATE_COST= 40.0;

/*
  内部myisam或innodb临时表的行代价
  当行数大于1000时，按顺序生成MyISAM行比生成Memory行慢2倍。然而，没有非常大的表的基准，因此保守地将此系数设置为慢5倍（即成本为1.0）。
*/
const double Server_cost_constants::DISK_TEMPTABLE_ROW_COST=
1.0;




/*
  在SE_cost_constants类中定义为静态常量变量的成本常量的值。如果服务器管理员没有在engine_cost表中添加新值，则将使用这些默认成本常数值。
*/

// 从主内存缓冲池读取块的成本
```

```
const double SE_cost_constants::MEMORY_BLOCK_READ_COST= 1.0;

// 从IO设备（磁盘）读取块的成本
const double SE_cost_constants::IO_BLOCK_READ_COST= 1.0;
```

opt_costmodel.cc【代价模型——部分涉及方法】

```
double Cost_model_table::page_read_cost(double pages) const
{
  DBUG_ASSERT(m_initialized);
  DBUG_ASSERT(pages >= 0.0);

  // 估算聚集索引内存中页面数占其所有页面数的比率
  const double in_mem= m_table->file->table_in_memory_estimate();

  const double pages_in_mem= pages * in_mem;
  const double pages_on_disk= pages - pages_in_mem;
  DBUG_ASSERT(pages_on_disk >= 0.0);

  const double cost= buffer_block_read_cost(pages_in_mem) +
    io_block_read_cost(pages_on_disk);

  return cost;
}

double Cost_model_table::page_read_cost_index(uint index, double
pages) const
{
  DBUG_ASSERT(m_initialized);
  DBUG_ASSERT(pages >= 0.0);

  double in_mem= m_table->file->index_in_memory_estimate(index);

  const double pages_in_mem= pages * in_mem;
  const double pages_on_disk= pages - pages_in_mem;

  const double cost= buffer_block_read_cost(pages_in_mem) +
    io_block_read_cost(pages_on_disk);

  return cost;
```

}

```

handler.cc【代价模型——部分涉及方法】

```
// 聚集索引扫描IO代价计算公式
Cost_estimate handler::read_cost(uint index, double ranges, double rows)
{

  DBUG_ASSERT(ranges >= 0.0);
  DBUG_ASSERT(rows >= 0.0);

  const double io_cost= read_time(index, static_cast<uint>(ranges),
                                  static_cast<ha_rows>(rows)) *
                        table->cost_model()->page_read_cost(1.0);
  Cost_estimate cost;
  cost.add_io(io_cost);
  return cost;
}

// 表全量扫描代价相关计算（IO-cost）
Cost_estimate handler::table_scan_cost()
{
  const double io_cost= scan_time() * table->cost_model()-
>page_read_cost(1.0);
  Cost_estimate cost;
  cost.add_io(io_cost);
  return cost;
}

// 覆盖索引扫描代价相关计算
Cost_estimate handler::index_scan_cost(uint index, double ranges,
double rows)
{
  DBUG_ASSERT(ranges >= 0.0);
  DBUG_ASSERT(rows >= 0.0);

  const double io_cost= index_only_read_time(index, rows) *
    table->cost_model()->page_read_cost_index(index, 1.0);
  Cost_estimate cost;
  cost.add_io(io_cost);
```
```

```
  return cost;
}


/**
  估算在指定 keynr索引进行覆盖扫描（不需要回表），扫描 records条记录
，需要读取的索引页面数

  @param keynr    Index number
  @param records  Estimated number of records to be retrieved
  @return
    Estimated cost of 'index only' scan
*/

double handler::index_only_read_time(uint keynr, double records)
{
  double read_time;
  uint keys_per_block= (stats.block_size/2/
                        (table_share->key_info[keynr].key_length + ref_length) +
                        1);
  read_time=((double) (records + keys_per_block-1) /
             (double) keys_per_block);
  return read_time;
}
```

sql_planner.cc【用于ref访问类型索引费用计算】

```
        double tmp_fanout= 0.0;
        if (table->quick_keys.is_set(key) && !table_deps &&          //(C1)
            table->quick_key_parts[key] == cur_used_keyparts &&
//(C2)
            table->quick_n_ranges[key] == 1+MY_TEST(ref_or_null_part))
//(C3)
        {
          tmp_fanout= cur_fanout= (double) table->quick_rows[key];
        }
        else
        {
          // Check if we have statistic about the distribution
```

```
        if (keyinfo->has_records_per_key(cur_used_keyparts - 1))
        {
          cur_fanout= keyinfo->records_per_key(cur_used_keyparts - 1);

          if (!table_deps && table->quick_keys.is_set(key) &&      // (1)
              table->quick_key_parts[key] > cur_used_keyparts)     // (2)
              {
                trace_access_idx.add("chosen", false)
                    .add_alnum("cause", "range_uses_more_keyparts");
                is_dodgy= true;
                continue;
              }

          tmp_fanout= cur_fanout;
        }
        else
        {

          rec_per_key_t rec_per_key;
          if (keyinfo->has_records_per_key(
                keyinfo->user_defined_key_parts - 1))
            rec_per_key=
              keyinfo->records_per_key(keyinfo->user_defined_key_parts -
1);
          else
            rec_per_key=
              rec_per_key_t(tab->records()) / distinct_keys_est + 1;

          if (tab->records() == 0)
            tmp_fanout= 0.0;
          else if (rec_per_key / tab->records() >= 0.01)
            tmp_fanout= rec_per_key;
          else
          {
            const double a= tab->records() * 0.01;
            if (keyinfo->user_defined_key_parts > 1)
              tmp_fanout=
                (cur_used_keyparts * (rec_per_key - a) +
                 a * keyinfo->user_defined_key_parts - rec_per_key) /
                (keyinfo->user_defined_key_parts - 1);
            else
              tmp_fanout= a;
            set_if_bigger(tmp_fanout, 1.0);
          }
          cur_fanout= (ulong) tmp_fanout;
        }

        if (ref_or_null_part)
```

```
        {
          // We need to do two key searches to find key
          tmp_fanout*= 2.0;
          cur_fanout*= 2.0;
        }

        if (table->quick_keys.is_set(key) &&
            table->quick_key_parts[key] <= cur_used_keyparts &&
            const_part &
            ((key_part_map)1 << table->quick_key_parts[key]) &&
            table->quick_n_ranges[key] == 1 + MY_TEST(ref_or_null_part
&
                                                      const_part) &&
            cur_fanout > (double) table->quick_rows[key])
        {
          tmp_fanout= cur_fanout= (double) table->quick_rows[key];
        }
      }


......

......

        // Limit the number of matched rows
        const double tmp_fanout=
          min(cur_fanout, (double) thd->variables.max_seeks_for_key);
        if (table->covering_keys.is_set(key)
            || (table->file->index_flags(key, 0, 0) &
HA_CLUSTERED_INDEX))
        {
          // We can use only index tree
          const Cost_estimate index_read_cost=
            table->file->index_scan_cost(key, 1, tmp_fanout);
          cur_read_cost= prefix_rowcount * index_read_cost.total_cost();
        }
        else if (key == table->s->primary_key &&
                 table->file->primary_key_is_clustered())
        {
          const Cost_estimate table_read_cost=
            table->file->read_cost(key, 1, tmp_fanout);
          cur_read_cost= prefix_rowcount * table_read_cost.total_cost();
        }
        else
          cur_read_cost= prefix_rowcount *
            min(table->cost_model()->page_read_cost(tmp_fanout),
                tab->worst_seeks);
```

```

handler.cc【用于range访问类型索引费用计算】

```
handler::multi_range_read_info_const(uint keyno, RANGE_SEQ_IF *seq,
                                     void *seq_init_param, uint n_ranges_arg,
                                     uint *bufsz, uint *flags,
                                     Cost_estimate *cost)
{
  KEY_MULTI_RANGE range;
  range_seq_t seq_it;
  ha_rows rows, total_rows= 0;
  uint n_ranges=0;
  THD *thd= current_thd;

  /* Default MRR implementation doesn't need buffer */
  *bufsz= 0;

  DBUG_EXECUTE_IF("bug13822652_2", thd->killed=
THD::KILL_QUERY;);

  seq_it= seq->init(seq_init_param, n_ranges, *flags);
  while (!seq->next(seq_it, &range))
  {
    if (unlikely(thd->killed != 0))
      return HA_POS_ERROR;

    n_ranges++;
    key_range *min_endp, *max_endp;
    if (range.range_flag & GEOM_FLAG)
    {
      min_endp= &range.start_key;
      max_endp= NULL;
    }
    else
    {
      min_endp= range.start_key.length? &range.start_key : NULL;
      max_endp= range.end_key.length? &range.end_key : NULL;
    }

    int keyparts_used= 0;
```
```

```
    if ((range.range_flag & UNIQUE_RANGE) &&                        // 1)
        !(range.range_flag & NULL_RANGE))
      rows= 1; /* there can be at most one row */
    else if ((range.range_flag & EQ_RANGE) &&                       // 2a)
             (range.range_flag & USE_INDEX_STATISTICS) &&           // 2b)
             (keyparts_used= my_count_bits(range.start_key.keypart_map))
&&
             table->
               key_info[keyno].has_records_per_key(keyparts_used-1) && //
2c)
             !(range.range_flag & NULL_RANGE))
    {
      rows= static_cast<ha_rows>(
        table->key_info[keyno].records_per_key(keyparts_used - 1));
    }
    else
    {
      DBUG_EXECUTE_IF("crash_records_in_range", DBUG_SUICIDE(););
      DBUG_ASSERT(min_endp || max_endp);
      if (HA_POS_ERROR == (rows= this->records_in_range(keyno,
min_endp,
                                                        max_endp)))
      {
        /* Can't scan one range => can't do MRR scan at all */
        total_rows= HA_POS_ERROR;
        break;
      }
    }
    total_rows += rows;
  }

  if (total_rows != HA_POS_ERROR)
  {
    const Cost_model_table *const cost_model= table->cost_model();

    /* The following calculation is the same as in multi_range_read_info():
*/
    *flags|= HA_MRR_USE_DEFAULT_IMPL;
    *flags|= HA_MRR_SUPPORT_SORTED;

    DBUG_ASSERT(cost->is_zero());
    if (*flags & HA_MRR_INDEX_ONLY)
      *cost= index_scan_cost(keyno, static_cast<double>(n_ranges),
                             static_cast<double>(total_rows));
    else
      *cost= read_cost(keyno, static_cast<double>(n_ranges),
                       static_cast<double>(total_rows));
    cost->add_cpu(cost_model->row_evaluate_cost(
```

```
        static_cast<double>(total_rows)) + 0.01);
    }
    return total_rows;
}
```

验证公式

创建验证需要的表

```
CREATE TABLE `store_goods_center`
(
    `id`           bigint(20)  NOT NULL AUTO_INCREMENT COMMENT '主键id',
    `sku_id`       bigint(20)  NOT NULL COMMENT '商品skuid',
    `station_no`   varchar(20) NOT NULL COMMENT '门店编号',
    `org_code`     bigint(20)  NOT NULL COMMENT '商家编号',
    `extend_field` text COMMENT '扩展字段',
    `version`      int(11)          DEFAULT '0' COMMENT '版本号',
    `create_time`  datetime         DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
    `create_pin`   varchar(50)      DEFAULT '' COMMENT '创建人',
    `update_time`  datetime         DEFAULT CURRENT_TIMESTAMP COMMENT '更新时间',
    `update_pin`   varchar(50)      DEFAULT '' COMMENT '更新人',
    `yn`           tinyint(4)       DEFAULT '0' COMMENT '删除标示  0:正常 1： 删除',
    `ts`           timestamp   NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '时间戳',
    PRIMARY KEY (`id`),
    UNIQUE KEY `uniq_storegoods` (`station_no`, `sku_id`) USING BTREE,
    KEY `idx_storegoods_org` (`org_code`, `sku_id`, `station_no`),
    KEY `idx_sku_id` (`sku_id`),
    KEY `idx_station_no_and_id` (`station_no`, `id`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4 COMMENT ='门店商品关系表';
```

通过存储过程初始化测试数据

```
DELIMITER //
CREATE PROCEDURE callback()
BEGIN
    DECLARE num INT;
    SET num = 1;
    WHILE
        num <= 100000 DO
        INSERT INTO store_goods_center(sku_id, station_no, org_code)
VALUES (num + 10000000, floor(50+rand()*(100-50+1)), num);
        SET num = num + 1;
    END WHILE;
END;
```

执行存储过程生成数据

```
CALL callback();
```

1.全表扫描计算代价公式

计算过程：

```
// 不同引擎计算方式有所区别
// innodb引擎实现handler.h
// 预估记录数：ha_innobase::info_low
// 页数量：ha_innobase::scan_time【数据总大小(字节) / 页大小】

// 查询全表数据大小（7880704）
SHOW TABLE STATUS LIKE 'store_goods_center';
// 查询数据库页大小（默认：16384）
```

SHOW VARIABLES LIKE 'innodb_page_size';

// 全表扫描计算代价
// 页数量
page = 数据总大小(字节) / 页大小 = 7880704 / 16384 = 481;
// 预估范围行数（总数据条数：10万，预估数据条数：99827，有一定误差）
records = 99827;

// 计算总代价
// 481 * 1 中的系数1 代表从主内存缓冲池读取块的成本
（SE_cost_constants::IO_BLOCK_READ_COST= 1.0）
// 99827 * 0.2 中的系数0.2 代表计算符合条件的行的代价
（ROW_EVALUATE_COST= 0.2）
cost = IO-cost + CPU-cost = (481 * 1) + (99827 * 0.2) = 481 + 19965.4 = 20446.4

```

验证结果：

```
explain format = json
select * from store_goods_center;

"cost_info": {"query_cost": "20446.40"}

```

总结公式：

```
全表扫描代价 = 数据总大小 / 16384 + 预估范围行数 * 0.2

```
```

2.覆盖索引扫描计算代价公式

计算过程：

```
// 查询全表数据大小（7880704）
SHOW TABLE STATUS LIKE 'store_goods_center';
// 查询数据库页大小（默认：16384）
SHOW VARIABLES LIKE 'innodb_page_size';

// 预估范围行数（总数据条数：1999，预估数据条数：1999，有一定误差）
1999;
records = 1999

// keys_per_block计算
// block_size是文件的block大小，mysql默认为16K；
// key_len是索引的键长度；
// ref_len是主键索引的长度；
keys_per_block = (stats.block_size / 2 / (table_share->key_info[keynr].key_length + ref_length) + 1);
// table_share->key_info[keynr].key_length 为联合索引，分别是station_no和sku_id
// station_no 为varchar(20)且为utf8mb4，长度 = 20 * 4 + 2 (可变长度需要加2) = 82
// sku_id bigint类型，长度为8
// 主键索引为bigint类型，长度为8
keys_per_block = 16384 / 2 / (82 + 8 + 8) + 1 ≈ 84

// 计算总代价
read_time = ((double) (records + keys_per_block - 1) / (double) keys_per_block);
read_time = (1999 + 84 - 1) / 84 = 24.78;

// 计算总代价
// 24.78 * 1 中的系数1 代表从主内存缓冲池读取块的成本（SE_cost_constants::IO_BLOCK_READ_COST= 1.0）
// 1999 * 0.2 中的系数0.2 代表计算符合条件的行的代价（ROW_EVALUATE_COST= 0.2）
cost = IO-cost + CPU-cost = (24.78 * 1) + (1999 * 0.2) = 24.78 + 399.8 = 424.58
```

```

验证结果：

```
explain format = json
select station_no from store_goods_center where station_no = '53';

"cost_info": {"query_cost": "424.58"}
```

总结公式：

```
keys_per_block = 8192 / 索引长度 + 1
覆盖索引扫描代价 = (records + keys_per_block – 1) / keys_per_block + 预估范围行数 * 0.2

公式简化（去除影响较小的复杂计算）
覆盖索引扫描代价 = (records * 涉及索引长度) / 8192 + 预估范围行数 * 0.2
```

3.ref索引扫描计算代价公式

计算过程：

```
// cardinality = 49（基数，即有多少个不同key统计。）
SHOW TABLE STATUS LIKE 'store_goods_center';

// 页数量
page = 数据总大小(字节) / 页大小 = 7880704 / 16384 = 481;

// 计算代价最低索引(sql_planner.cc 中find_best_ref函数)
// IO COST最坏不会超过全表扫描IO消耗的3倍(或者总记录数除以10)
// 其中s->found_records表示表上的记录数，s->read_time在innodb层表示
```
```

page数
// s-> worst_seeks = min((double) s -> found_records / 10, (double) s -> read_time * 3);
// cur_read_cost= prefix_rowcount * min(table->cost_model() -> page_read_cost(tmp_fanout), tab -> worst_seeks);

// 预估范围行数（总数据条数：10万，预估数据条数：99827，有一定误差）

total_records = 99827;
// 预估范围行数（总数据条数：1999，预估数据条数：1999，有一定误差）
1999;
records = 1999

// 计算总代价
// 1999 * 0.2 中的系数0.2 代表计算符合条件的行的代价
（ROW_EVALUATE_COST= 0.2）
// s-> worst_seeks = min((double) s -> found_records / 10, (double) s -> read_time * 3) -> min(99827 / 10, 481 * 3) = 481 * 3
// min(table->cost_model() -> page_read_cost(tmp_fanout), tab -> worst_seeks) -> min(page_read_cost(1999), 481 * 3) = 481 * 3
cost = IO-cost + CPU-cost = 481 * 3 + (1999 * 0.2) = 1443 + 399.8 = 1842.80

```

验证结果：

```
explain format = json
select * from store_goods_center where station_no = '53';

"cost_info": {"query_cost": "1842.80"}

```

总结公式：

```
下面3个公式，取值最低的
1.(数据总大小 / 16384) * 3 + 预估范围行数 * 0.2
2.总记录数 / 10 + 预估范围行数 * 0.2
3.扫描出记录数 + 预估范围行数 * 0.2

```
```

4.range索引扫描计算代价公式

```

// 预估范围行数（总数据条数：1299，预估数据条数：1299，有一定误差）
1299;
records = 1299

// 计算代价最低索引(handler.cc 中 multi_range_read_info_const 函数)
// 计算总代价
// 1299 * 0.2 计算公式：cost_model->row_evaluate_cost(static_cast<double>(total_rows))
// + 0.01 计算公式：cost->add_cpu(cost_model->row_evaluate_cost(static_cast<double>(total_rows)) + 0.01);
// 1299 + 1 中的 +1 ： 单个扫描区间（ id > 35018 ）
// 1299 + 1 计算公式：*cost= read_cost(keyno, static_cast<double>(n_ranges), static_cast<double>(total_rows));
// (1299 * 0.2 + 0.01 + 1299) * 1 中的系数1 代表从主内存缓冲池读取块的成本（SE_cost_constants::IO_BLOCK_READ_COST= 1.0）
// 1299 * 0.2 中的系数0.2 代表计算符合条件的行的代价（ROW_EVALUATE_COST= 0.2）
cost = IO-cost + CPU-cost = ((1299 * 0.2 + 0.01 + 1299 + 1) * 1) + (1299 * 0.2) = 1559.81 + 259.8 = 1819.61

```

验证结果：

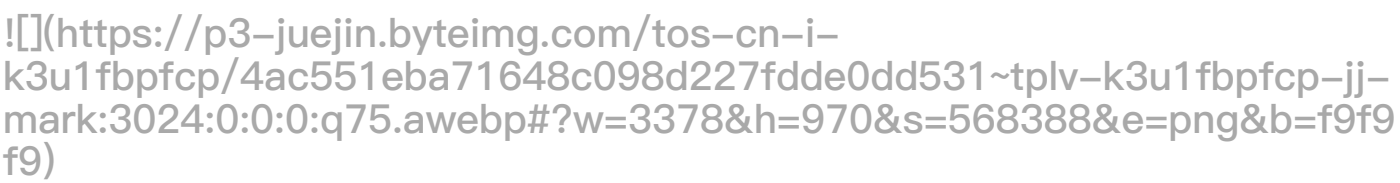
```
explain format = json
select * from store_goods_center where station_no = '53' and id > 35018;

"cost_info": {"query_cost": "1819.61"}
```

```

总结公式：

```
range扫描代价 = 预估范围行数 * 1.4 + 0.01 + 范围数

公式简化（去除影响较小的复杂计算）
range扫描代价 = 预估范围行数 * 1.4
```

索引冲突案例
======

门店商品系统中主要存储门店与商品的关联信息，并为B端提供根据门店ID查询关联商品的功能。由于门店关联的商品数据量较大，需要分页查询关联商品数据。为避免深分页问题，我们选择基于上次最新主键进行查询（核心思想：通过主键索引，每次定位到ID所在位置，然后往后遍历N个数据。这样，无论数据量多少，查询性能都能保持稳定。我们将所有数据根据主键ID进行排序，然后分批次取出，将当前批次的最大ID作为下次查询的筛选条件）。

```
select 字段1，字段2 ... from store_goods_center where station_no = ‘门店id’ and id > 上次查询最大id order by id asc
```

为了确保门店与商品组合的唯一性，我们在MySQL表中为门店ID和商品ID添加了组合唯一索引【UNIQUE KEY uniq_storegoods (station_no, sku_id) USING BTREE】。由于该索引包含门店ID并且在联合索引的第一个位置，查询会使用该索引。但是，当分页查询命中该索引后，由于排序字段无法使用索引，产生了【Using filesort】，导致门店商品系统出现了一些慢查询。为了解决这个问题，我们对慢查询进行了优化，优化思路是创建一个新的索引，使该SQL可以使用索引的排序来规避【Using filesort】的负面影响，新添加的索引为【KEY idx_station_no_and_id (station_no, id)】。添加该索引后，效果立竿见影。

然而，我们发现仍然有慢查询产生，并且这些慢查询仍然使用uniq_storegoods索引，而不是idx_station_no_and_id索引。我们开始思考，为什么MySQL没有为我们的系统推荐使用最优的索引？是MySQL索引推荐
```

有问题，还是我们创建索引有问题？如何做才能让MySQL帮我们推荐我们认为最优的索引？

当然，我们也可以使用FORCE INDEX强行让MySQL走我们提前预设的索引，但是这种方式局限太大，后期索引维护成本变得很高，甚至可能使用该SQL的其他业务性能变低。为了突破整体优化的卡点状态，我们需要了解一下MySQL索引推荐底层逻辑，即MySQL代价模型。了解相应规则后，现阶段的问题将迎刃而解。



案例分析及优化
=======

在回顾刚才的问题时，我们发现问题源于原始索引产生了【Using filesort】，从而导致了慢查询的出现。为了解决这个问题，我们新增了一个索引，即【KEY idx_station_no_and_id (station_no, id)】，以替代原有的索引【UNIQUE KEY uniq_storegoods (station_no, sku_id)】。然而，尽管新增索引后大部分慢查询得到了解决，但仍有部分慢查询未能消除。进一步分析发现，这些慢查询是由于SQL没有使用我们期望的索引，而是使用了老索引，从而引发了【Using filesort】问题。在通过explain进行分析后，我们暂时还没有找到合适的解决方案。

问题：尽管我们新增了索引，并且大部分SQL已经能够使用新索引进行优化，但仍存在一些SQL没有使用新索引。

```
// 通过代价模型进行分析

// 使用上面的测试数据进行分析
// 新增索引后都没有走新索引
// 老索引，扫描行数：1999，代价计算值：1842.80，ref类型索引
// 新索引，扫描行数：1999，代价计算值：1850.46，range类型索引
select 字段1，字段2 ... from store_goods_center where station_no = ‘门店id’ and id > -1 order by id asc;

// 新增索引后走新索引
// 老索引，扫描行数：1999，代价计算值：1842.80，ref类型索引
// 新索引，扫描行数：1299，代价计算值：1819.61，range类型索引
select 字段1，字段2 ... from store_goods_center where station_no = ‘门店id’ and id > 35018 order by id asc;

```

经过分析MySQL的代价模型，我们发现MySQL在选择使用哪个索引时，主要取决于扫描出的数据条数。具体来说，扫描出的数据条数越少，MySQL就越倾向于选择该索引（**由于MySQL的索引数据访问类型各异，计算公式也会有所不同。因此，在多个索引的扫描行数相近的情况下，所选索引可能与我们期望的索引有所不同**）。顺着这个思路排查，我们发现当id > -1时，无论是使用storeId + skuId还是storeId + id索引进行查询，扫描出的数据条数是相同的。这是因为这两种查询方式都是根据门店查询商品数据，且id值肯定大于1。因此，对于MySQL来说，由于这两种索引扫描出的数据条数相同，所以使用哪种索引效果相差不多。这就是为什么一部分查询走新索引，而另一部分查询走老索引的原因。然而，当查询条件为id > n时，storeId + id索引的优势便得以显现。因为它能够直接从索引中扫描并跳过id <= n的数据，而storeId + skuId索引却无法直接跳过这部分数据，因此真正扫描的数据条数storeId + skuId要大于storeId + id。因此，在查询条件为id > n时，MySQL更倾向于使用新索引。（**需要注意的是，示例给出的数据索引数据访问类型不同，一个是range索引类型，一个是ref索引类型。由于算法不同，即使某个索引的检索数据率略高于另一个索引，也可能导致系统将其推荐为最优索引**）

问题已经分析清楚，主要原因是存在多个索引，且根据索引代价计算公式的代价相近，导致难以抉择。因此，解决这个问题的方法不应该是同时定义两个会让MySQL”纠结”的索引选择。相反，应该将两个索引融合为一个索引。具体的解决方案是根据门店查询，将原来的主键id作为上次查询的最大id替换为skuId。在算法切换完成后，删除新的门店+主键id索引。然而，这种方式可能会引发另一个问题。由于底层排序算法发生了变化（由原来的主键id改为skuId），可能导致无法直接从底层服务切换。此时，应考虑从下游使用此接口服务的应用进行切换。需要注意的是，如果下游系统是单机分页迭代查询门店数据，那么下游系统可以直接进行切换。但如果这种分页查询动作同时交给多台应用服务器执行，切换过程将变得相当复杂，他们的切换成本与底层切换成本相同。但是，这个系统的对外服务属于这种情况，下游调用系统会有多台应用服务器协作分页迭代查询数据，为这次优化带来很大影响。

最终，让底层独立完成切换方式最为合适。在切换过程中，关键在于正确区分新老算法。老算法在迭代过程中不应切换至新算法。原系统对外服务提供的下次迭代用的id可用来进行区分。新算法在返回下次迭代用的id基础上增加一个常量值，例如10亿（加完后不能与原数据冲突，也可以将迭代id由整数转换成负数以区分新老算法）。因此，如果是第一次访问，直接使用新算法；如果不是第一次访问，需要根据下次迭代用的id具体规则来判断是否切换新老算法。

总结与后续规划

使用Explan执行计划存在无法提前预知索引选择的局限性。然而，只要熟悉MySQL底层代价模型的计算公式，我们就能预知索引的走向。借助代价模型，我们不仅可以分析索引冲突的原因，还可以在发生冲突之前进行预警。甚至在添加索引之前，我们也可以根据代价模型公式来排查潜在问题。此外，根据数据业务密度，我们还可以预估当前索引的合理性，以及是否可能出现全表扫描等情况。因此，深入研究MySQL代价模型对于优化索引管理具有关键意义。

未来我们的系统应用将结合MySQL代价模型进行集成，实现自动分析数据库和表的信息，以发现当前索引存在的问题，例如索引冲突或未使用索引导致的全表扫描。此外，该工具还可以针对尚未添加索引的表，根据数据情况提供合适的索引推荐。同时，该工具还能够预测当数据达到某种密度时，可能出现全表扫描的问题，从而帮助提前做好优化准备。

为了实现这些功能，我们将首先对MySQL代价模型进行深入研究，全面了解其计算公式和原理。这将有助于我们编写相应的算法，自动分析数据库和表的信息，找出潜在的索引问题。此外，我们还易用性和实用性，确保用户能够轻松地输入相关数据库和表的信息，并获取有关优化建议。

该工具的开发将有助于提高数据库性能，减少全表扫描的发生，降低系统资源消耗。同时，它还可以为数据库管理员和开发人员提供便利，使他们能够更加专注于其他核心业务。通过结合MySQL代价模型，我们相信这个工具将在优化索引管理方面发挥重要作用，为企业带来更高的效益。

参考资料

====

[github.com/mysql/mysql…](http://cxyroad.com/”https://github.com/mysql/mysql-server”)

作者：京东零售 王多友

来源：京东云开发者社区

原文链接: https://juejin.cn/post/7366899841455505418