MySQL InnoDB 查询优化实现分析

网易杭研-何登成

1 目的

分析 mysql+innodb 如何实现查询优化？实现查询优化，存储引擎需要做哪些方面的配合？

2 测试准备

mysql

select version(); 5.1.49-debug-log

innodb

--------表定义---------

```
+-------+------------------------------
| Table | Create Table
+-------+------------------------------
| nkeys | CREATE TABLE `nkeys` (
  `c1` int(11) NOT NULL,
  `c2` int(11) DEFAULT NULL,
  `c3` int(11) DEFAULT NULL,
  `c4` int(11) DEFAULT NULL,
  `c5` int(11) DEFAULT NULL,
  PRIMARY KEY (`c1`),
  UNIQUE KEY `c2` (`c2`),
  UNIQUE KEY `c3` (`c3`),
  UNIQUE KEY `c4` (`c4`),
  KEY `nkey1` (`c3`,`c5`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk |
+-------+------------------------------
```

----数据----

```
insert into nkeys values (1,1,1,1,1);
insert into nkeys values (2,2,2,2,2);
insert into nkeys values (3,3,3,3,3);
insert into nkeys values (4,4,4,4,4);
insert into nkeys values (5,5,5,5,5);
```

3 单表查询

3.1 单表 range 查询

1) select * from nkeys where c3 > 3; 不能进行索引覆盖扫描 index range scan
2) select c3 from nkeys where c3 > 3; 可以进行索引覆盖扫描 index only range scan

调用流程：

msyql_select -> JOIN::optimize -> make_join_statistics ->

0. sql_select.cc::get_quick_record_count -> opt_range.cc::SQL_SELECT::test_quick_select –>ha_innobase::scan_time ->get_key_scans_params ->check_quick_select–>Opt_range.cc::check_quick_keys ->ha_innobase::records_in_range-> get_index_only_read_time -> ha_innobase::read_time ->get_best_ror_intersect ->get_best_covering_ror_intersect ->

   a) **ha_innobase::scan_time** 函数，给出全表扫描 read_time
      i. scan_time = (double) records / TIME_FOR_COMPARE + 1;
         1. mysql 层面，返回一个 record 需要的时间(CPU 时间)
         2. TIME_FOR_COMPARE = 5
      ii. return (double) (prebuilt->table->stat_clustered_index_size(聚簇索引叶页面数);
         1. innodb 层面，全表扫描时间，用读取的 page 数计算(IO 时间)
         2. 由于 innodb 是索引组织表，用不到 page 的预读，因此一次读取一个 page
      iii. **table_read_time = ha_innobase::scan_time() + scan_time + 1;**
         1. 全表扫描总时间 = innodb 读取数据块时间 + mysql 比较记录时间 + 1
         2. 测试中：table_read_time = 4.3000000000000007

   b) **check_quick_select** 函数，判断索引扫描的代价

   c) **ha_innobase::records_in_range** 函数，判断给定 range 的索引扫描，将返回多少记录
      i. 给定 range 的 min_key，max_key，根据 min_key，max_key 构造查询条件，分别进行 btr_cur_search_to_nth_level
      ii. 传入的 level 是 0，search 到叶页面
      iii. 根据返回的两个页面的关系，计算 range 中的数据量
      iv. 详细的 records_in_range 函数实现，请见 3.1.1 章节。

   d) **get_index_only_read_time** 函数，当前 scan 为 index only scan，调用此函数计算 read_time
      i. cpu_cost = (double) found_records / TIME_FOR_COMPARE;
         1. range 中的记录数，除以比较时间(CPU 时间)
      ii. get_index_only_read_time，mysql 上层提供函数，用于计算 index only scan 的

代价

1. keys_per_block = (table->file->stats.block_size/2)(a) / (table->key_info[keynr].key_length + table->file->ref_length + 1)(b)
   a) (a) 估计索引页面的利用率为 1/2
   b) (b) 索引中，每个索引占用的空间；keynr 为索引的编号，哪个索引？
   c) 测试中：keys_per_block = 911
2. io_time = (double) (records + keys_per_block - 1) / keys_per_block;
   a) 需要进行多少次 index 叶页面的 IO (index only scan，不需要回表)
   b) 测试中：io_time = 1.0021953896816684 (records = 3)
3. **index_only_read_time = cpu_cost + io_time + 0.01 = 1.6121953896816683**
   a) index_only_read_time < table_read_time
   b) 测试中：index only scan 要好于 table scan，针对第二条语句
   c) **对于语句(2)，mysql 选择索引覆盖扫描**
      i. **索引 c1，而非 nkey1，虽然 nkey1 索引也可以做覆盖性扫描，但是 nkey1 的 key_length 要大于 c1，导致 io_time 略微大于 c1。**

e) **ha_innobase::read_time** 函数，非 index only scan 时，调用此函数计算 read_time
   i. cpu_cost = (double) found_records / TIME_FOR_COMPARE;
      1. range 中的记录数，除以比较时间(CPU 时间)
   ii. ha_innobase::read_time Innodb 层面读取的页面，IO 时间
      1. 聚簇索引
         a) rows <= 2 时 return rows
         b) return (ranges + rows)/total_rows * 全表扫描 IO 时间
      2. 二级索引
         a) return rows2double(ranges+rows)
         b) IO 时间为 ranges 个数+range 中的行数 = 二级索引定位代价 + 回聚簇代价(由于无法进行 index only scan)
   iii. index_read_time = cpu_cost + io_cost + 0.01;
      1. 测试中：index_read_time = 4.60999999999999994，大于全表扫描时间

f) 比较所有 e 步骤计算出来的 index_read_time 与 a 步骤计算出来的 table_read_time 之间的大小关系，确定当前 scan 是选择全表扫描，还是索引扫描
   i. **对于语句(2)，mysql 选择索引覆盖扫描**
   ii. **对于语句 1)，mysql 最终选择全表扫描**

g) get_best_ror_intersect 函数，是一个优化路径。
   i. ROR = *Rowid Ordered Retrieval* key scan，索引扫描得到的记录，其 rowid 是排序的，极大降低了回表开销，在此重新评估此索引扫描的代价(ror 类似于 Oracle 中的 cluster index factor，索引聚簇因子，不过没有 Oracle 统计的那么详尽)

1. optimize_keyuse ->
   a) 针对 join 查询
2. choose_plan ->
   a) 执行计划选择主函数，读取分析用户定义属性
3. greedy_search ->
   a) 从 join_tables 中逐个选取最优的表，加入当前已选择的 pplan

```
@code
procedure greedy_search
input: remaining_tables
output: pplan;
{
  pplan = <>;
  do {
    (t, a) = best_extension(pplan, remaining_tables);
    pplan = concat(pplan, (t, a));
    remaining_tables = remaining_tables - t;
  } while (remaining_tables != {})
  return pplan;
}
```

4. best_extension_by_limited_search ->
   a) 从 join_tables 的 remain_tables 中选择一个 table 加入 pplan，目标使得整体 pplan 的开销最小
5. best_access_path ->
   a) 若为单表，计算单表的全表扫描代价。
   b) 若为多表，计算当前选择表的扫描代价。
6. make_join_readinfo -> pick_table_access_method -> tab->index = find_shortest_key(table, & table->covering_keys) -> tab->read_first_record = join_read_first -> tab->type = JT_NEXT ->
   a) 索引覆盖扫描路径优化。若当前为全表扫描，同时存在一个或多个可以进行索引覆盖扫描的查询，那么优先选择索引覆盖扫描。
      i. 原理：针对 Innodb 引擎，索引覆盖扫描一定要优于全表扫描
      ii.
   b) 对于单表扫描，步骤 0 确定是否可以选择索引。步骤 5 返回全表扫描开销。步骤 6 主要处理 index coverage scan 的部分优化。
   c) 在函数 find_shortest_key 中，选择合适的索引，for index coverage scan。
      i. 索引必须包含 scan 键值？
      ii. 索引列的 key_length 最小？

3.1.1 records_in_range 函数分析

records_in_range -> btr_estimage_n_rows_in_range ->
    tuple1 = min value in range scan，range scan 的范围起始值
btr_cur_search_to_nth_level(index, tuple1, &cursor) ->
    tuple2 = max value in range scan，range scan 的范围终止值
btr_cur_search_to_nth_level(index, tuple2, &cursor) ->

根据起始值与终止值，做两次 search path，确定 index path，存储在 cursor 中我们有了起始值与终止值的两个 path，起始值与终止值所对应的索引叶节点如何根据两个叶节点计算叶节点范围内的数据量(records in range)，想法如下：

1. 计算出两个叶节点间，包含多少个索引页，记为 n (n leaf pages in range)
2. 计算索引页平均包含多少个索引项，记为 r (records per leaf page)
3. 那么，records in range = n * r

Innodb 采用相同的计算方法，innodb 计算 n，r 的算法如下:

计算 n，采用自顶向下的方式计算。

(1) 根页面只有一个，可以计算出根页面内，两个 range 间相差多少索引项 n1

(2) 第二层页面的 n2 = n1 * r2(第二层页面平均记录数)

(3) 索引叶节点一层，nn = records in range = nn-1 * rn(叶节点评价记录数)

计算 r，采用水平采样的方式计算。

r 的算法相对简单，但是却会产生 IO，其核心思想是，对于索引的每一层，从 tuple1 确定的路径页面开始，沿着页面中的水平链表，向后遍历页面，遍历结束的条件是：

1) 遇到了 tuple2 确定的页面;

2) 或者读取了 N_PAGES_READ_LIMIT 个页面(10 个)。

假设读取了 N_PAGES_READ_LIMIT 个页面，一共有 M 个索引记录，那么 r = M / N_PAGES_READ_LIMIT.

records_in_range 函数总结分析：

- 总的来说，records_in_range 函数是一个较为费时的函数，两次 search path 开销，索引层数次的计算索引每一层的 records in range 开销。如果表数据量较大，索引层数较多，同时查询的 range 也较大，那么 records_in_range 函数会产生多次 IO，甚至是物理 IO，这个是难以接受的。当然，如果查询的 range 较小，两次 search path 得到相同的路径，那么 records in range perevery level 的开销可以忽略不计。

- 解决 records_in_range 函数开销的办法之一，就是在 Innodb 内部收集更为详尽的统计信息，包括 unique key counts，以及 histogram(柱状图)。通过统计信息的计算来近似估计 records_in_range，从而避免两次 search path 以及 records in range per every level 的开销。

3.1.2 best_access_path 函数(单表)

在前面，我们分析到，best_access_path 函数针对单表，仅仅计算全表扫描的开销。其实这个说法不是完全正确。在部分情况下，best_access_path 函数针对单表 range 查询，也会计算索引开销。也就是说函数中，s->keyuse != NULL。
那么 s->keyuse 参数是何时设置的呢？

通过跟踪两个不同的查询，可以找出设置规律。

sql 1：

```
select c3 from nkeys where c3 <= 520;
```

sql 2(参考文献[2])：

```
select * from TB_IMGOUT_Picture WHERE (2601629 > ID) AND (UserId = 129329421) AND (DeleteState = 0) ORDER BY ID DESC LIMIT 100;
```

通过测试发现，执行 sql 1 时，s->keyuse == NULL；而执行 sql2 时，s->keyuse != NULL。

s->keyuse 设置流程：
sql_select.cc::make_join_statistics -> update_ref_and_keys ->
sql_select.cc::add_key_fields ->

重点在于函数 sql_select.cc::add_key_fields：
此函数会被递归调用，遍历当前 sql 中的所有查询条件，为每一个查询条件，寻找合适的索引。简单来说，也就是设置每个表的 s->keyuse，在 best_access_path 函数中可以估算这些索引的代价。
那么哪些查询条件，才会使用索引呢？
sql 1 的查询条件是 <=
sql 2 的查询条件是 >==

结果是 sql 2 的 s->keyuse 被设置。就我测试跟踪的代码看来，若是等值条件，则设置 s->keyuse，否则不设置。代码如下：

针对 sql 1 与 sql 2 的第一个查询条件(2601629 > ID)，函数 add_key_fields 判断如下：

```
case Item_func::OPTIMIZE_OP: {
    bool equal_func=(cond_func->functype() == Item_func::EQ_FUNC ||
             cond_func->functype() == Item_func::EQUAL_FUNC);
    add_key_equal_fields(... ,equal_func,...);
        add_key_field(..., equal_func, ...);
            if (!eq_func)
                return;
}
```

cond_func->functype() = LE_FUNC(GT_FUNC)，因此 equal_func = false，导致 add_key_field 直接返回，因此 s->keyuse 不设置

针对 sql 2 的第二个查询条件(UserId = 129329421)，函数 add_key_fields 判断如下：

```
case Item_func::OPTIMIZE_EQUAL:
    add_key_field( ..., TRUE, ...);
        // 此处增加possible_keys
        // 根据field->table->keys_in_use_for_query，与当前表上的index做一次比较，
        // 就得出possible_keys = 6 = 2^1 + 2^2，  1号与2号索引可以用，分别是
        // IDX_UID_PHOTOID，IDX_UID_ID。都是以UserId字段打头的索引。
        possible_keys.intersect(field->table->keys_in_use_for_query);
        stat[0].keys.merge(possible_keys);

        if (!eq_func)
            return;
        (*key_fields)->field    =    field;
        (*key_fields)->eq_func  =    eq_func;
        (*key_fields)->val      =    *value;
```

```
        (*key_fields)->level    =    and_level;
    }
```
equal_func 直接设置为 TRUE，因此 UserId 对应的索引将会被添加到 s->keyuse 中。

为什么单表 range 查询，在 range query optimization 之后，还需要调用 best_access_path 函数进行二次执行计划的选择？

我的理解是，这是针对 等值 查询条件的一种二次优化。mysql range query optimization 更倾向于选择聚簇索引扫描(二级索引回表代价过大)，而对于 等值 查询，我们可以在 best_access_path 函数中再次计算使用二级索引的代价。若此时计算的代价小于 range query optimization 选择的执行计划，则替换新的执行计划。

mysql range query optimization 选择的执行计划，如果不是全表扫描，那么一定要优于全表扫描的代价。

3.1.3 单表 range 查询总结

- 每一条执行路径，其 cost = CPU cost + IO cost
- 单表 range scan 的执行计划选择，记住几个关键数据的计算即可
  - table read time，以上步骤 0)下面的 a)步骤
  - index only read time，以上步骤 0)下面的 d)步骤
  - index read time，以上步骤 0)下面的 e)步骤
- 为了计算以上关键数据，innodb 存储引擎提供了 ha_innobase::records_in_keys 方法，该方法根据 scan 的 min_key，max_key 进行两次索引上的 search path(一直读取到叶页面)，可能产生 IO，然后根据两个叶页面，进行 records_in_range 的预估。
- 根据 where 查询条件，mysql 首先会选择哪些索引可能适合做查询，这些潜在的索引，就是 explain 中的 possible_keys 一列中所列出来的索引。而 possible_keys 的选择，就是由 where 条件中涉及到的列打头的所有索引。附录一证实了这个估计。
- 由于 mysql 会对每一个 possible_keys 调用 records_in_range 进行预估，同时该函数需要两次 search path，多次索引页面水平扫描，可能产生两次或以上的 IO，因此我的建议是，同一个键值打头的索引，越少越好，在 mysql 中
- explain 不会真正执行 scan，而是只会执行上面的流程，然后将预估出来的各值返回给用户
- mysql 的查询优化，用的是 CBO，而非 RBO。
- mysql 的查询优化，对于 unique scan，会特殊处理，不会这么复杂的路径，因此 unique scan 的查询优化效率很高。
- 根据索引覆盖扫描与全表扫描的理论代价可得，索引覆盖扫描一定要优于全表扫描。innodb 的全表扫描也就是聚簇索引的覆盖扫描，由于聚簇索引包含记录的所有属性，因此其单行记录的长度一定大于二级索引，
  - 二级索引的页面数一定小于聚簇索引页面数➔二级索引的代价一定小于聚簇索引➔若有索引覆盖扫描，哪怕覆盖扫描的估算代价大于全表扫描，最终还是会补偿选择索引覆盖扫描，在以上测试的第 6 步骤可得。
  - 附录二证实了此分析。

3.2 单表 unique 查询

select * from nkeys where c3 = 3;

调用流程：

mysql_execute_command -> handle_select -> mysql_select -> JOIN_optimize -> make_join_statistics ->

if ((table->key_info[key].flags & (HA_NOSAME | HA_END_SPACE_KEY)) == HA_NOSAME) ->

1. 若当前索引为 unique 索引，同时

if (const_ref == eq_part) ->

2. 指定的等值条件与当前索引的 unique key 一致

s->type = JT_CONST ->

3. 当前表上的查询，返回的数据量是一个常量

join_read_const_table -> … -> row0sel.c::row_search_for_mysql ->

4. 调用底层函数，做一次 unique scan，主要功能是保证 explain 的正确性。有这个必要吗？看下面的 explain 截图，就可以发现此函数的功能：

if (s->type == JT_SYSTEM || s->type == JT_CONST) ->

s->found_records = s->records = s->read_time = 1; s->worst_seeks = 1.0;

5. JT_CONST 情况下，查询一定只返回 1 条数据，不需要调用 ha_innobase::records_in_range 函数进行判断。

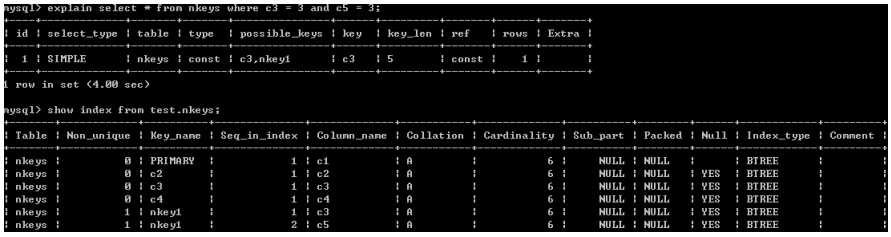
if (join->const_tables != join_tables)

choose_plan();

6. 若当前不全是 TL_CONST 查询，还需要调用 choose_plan 函数，判断非 const 表的最优执行路径。单表情况下，只需要计算全表扫描代价；多表 join 条件下，需要计算各非 const 表的 join 顺序以及单表的最优路径。TL_CONST 情况下，不需要调用 choose_plan(单表查询情况下)。

3.2.1 单表 Unique 查询总结

1. mysql 查询优化，对于 unique 查询做了路径优化，并不需要调用底层提供的 records_in_range 函数判断查询的代价；同时也不需要调用 choose_plan 函数，确定此查询全表扫描代价(单表情况下)。
2. 只要指定了 unique key 上的等值查询，那么无论后面还有多少其他查询条件，mysql 查询优化一定会选择 unique key 索引，做 unique 查询，如下：



查询优化仍旧选择 c3 unique 索引，而非选择 nkey1 索引，虽然 nkey1 上有 c3，c5 两列。

4 多表查询

4.1 多表简单 join

select * from nkeys, aaa where nkeys.c3 = aaa.a3 and aaa.a2 = 2;
具体 nkeys，aaa 表的表定义，在附录一：aaa 表；附录四：nkeys 表中给出。

调用主流程：

1. mysql_select ->JOIN::optimize -> make_join_statistics ->SQL_SELECT::test_quick_select ->get_key_scans_params ->

根据查询条件，aaa 表指定了查询条件：aaa.a2 = 2，对于指定查询条件的表，可以通过第 3 章节单表查询的流程找到针对查询条件的最优路径。当然，若 aaa.a2 列为 unique 列，那么就通过第 3 章节单表 unique 查询来判断。

2. if (join->const_tables != join_tables) ->

此处判断 join 查询中，指定了 unique key 查询条件的表的数量，与 join 中表的数量是否一致？所有指定了 unique key 查询条件的表，执行计划已经确定，必定是走 unique key 索引，因此不需要进行查询优化，代价评估。此处不一致，因此需要调用下面的 choose_plan 函数，确定整个 join 操作的最优路径。整个 join 操作的最优路径，包含两层含义：(一) join 操作，涉及到的表的 join 顺序是最优的；(二) 每张表的执行路径同样也是最优的；最终的目标，是保证选择出整个 join 代价最小的路径。

3. choose_plan ->

4.1 optimize_straight_join -> best_access_path -> return

straight_join，join 顺序确定，只需要确定每个 table 的最优路径。如果是 straight_join，也就是说不对查询给定的 table 顺序进行调整，而仅仅是根据 table 的顺序，为每一个 table 找到最优的执行路径，选择 4.1 的代码路径。此路径相对简单，顺序遍历 join 结构中的每一张表，找到表的 best_access_path，估算代价。一般情况下，不会选择此方案。

4.2 greedy_search ->

greedy_search 函数的功能，是用贪心法找到局部最优的 join 执行计划，与 optimize_straight_join 函数不同，greedy_search 函数会改变表的 join 顺序。

关于 greedy_search 函数的详细说明，可见 sql_select.cc::greedy_search 函数功能说明。简单来说，给定剩余 join 表集合，每次调用 best_extension_by_limited_search 函数，从剩余表集合中选择一个代价最小的表，加入到当前执行计划中来，直到剩余 join 表集合耗尽为止。

给定 N 张表的 join，需要经过 O(N!)次(N 的阶层，card(5) = 120)的计算，才能找到最优的执行计划，cpu 开销较大。Mysql 的优化措施为，设置 search_depth，控制最优路径的查找限制，search_depth 参数在下面提到的 best_extension_by_limited_search 函数中使用。若 N <= search_depth，则一定能找到最优计划；否则只能找到局部最优计划
greedy_search 函数的伪代码如下：

```
@code
    procedure greedy_search
    input: remaining_tables
    output: pplan;
    {
      pplan = <>; // 当前已经选择的表的最优执行计划
      do {
        (t, a) = best_extension(pplan, remaining_tables);
        pplan = concat(pplan, (t, a));
        remaining_tables = remaining_tables - t;
      } while (remaining_tables != {})
      return pplan;
    }
```

5. best_extension_by_limited_search ->

greedy_search 函数传入的 search_depth 参数，在 best_extension_by_limited_search 函数中使用。

best_extension_by_limited_search 函数是一个深度优先的递归调用。调用的深度即为 search_depth。

best_extension_by_limited_search 函数的原理，就是通过深度优先的递归调用，找到一个表，两个表，…，search_depth 个表的最优 partial plan 的 cost。如下图所示：

pplan[0]: t0, cost0
pplan[1]: t1, cost1
pplan[2]: t2, cost2
pplan[3]: t3, cost3
…
pplan[search_depth]:

只有第一个和最终选出的最优计划(partial or full)，是完全可用的执行计划。中途的 pplan 中的各节点，不一定能构成一个完整的计划，因为随时都可能在第 n 个步骤断掉。

若 search_depth >= N(number of remaining tables)，则能够完成所有路径的遍历，找到对于 remaining tables 的一个最优执行计划。返回 greedy_search 的 QEP，已经是针对于 remaining tables 则最优执行计划，greedy_search 函数直接将此计划返回即可。

若 search_depth < N，那么 best_extension_by_limited_search 函数不能在 remaining tables 中找到最优的执行计划，而只能找到一个局部最优的计划。此时，函数返回 greedy_search，greedy_search 会提取出当前局部最优计划的第一张表，作为下一个已经确定的 join 表。于此同时，greedy_search 函数会将选中的 table 从 remaining tables 中删除，然后再次调用 best_extension_by_limited_search 函数，直到 N(number of remaining tables) <= search_depth，余下的所有 table，一次 best_extension_by_limited_search 函数调用，就能够找到最优的执行计划。

极端情况下，search_depth = 1，则 best_extension_by_limited_search 函数退化为宽

度优先的策略，每次从 remaining tables 中提取能够与已经选出的最后一个 table 做 join，并且扫描代价最小的 table，然后立即返回 greedy_search 函数，greedy_search 函数提取此表作为当前 partial QEP 的最后一张表。

best_extension_by_limited_search 函数的伪代码实现，可参见附录三。

根据以上的分析，可以总结出 greedy_search+ best_extension_by_limited_search 两个函数的复杂度。

若 N <= search_depth，则复杂度为 O(N!)，best_extension_by_limited_search 函数通过深度优先递归遍历，找到针对于 N tables 的最优执行路径。

若 N > search_depth，则复杂度为 O(N*N^search_depth/search_depth)，greedy_search 不能找到 join 的最优执行路径。

6. best_access_path ->

对于指定的表 s，找到其最优的执行路径。可选的路径，必须是 join 查询中，可以完成 nestloop join 的路径，对于 4.1 章节给定的查询，aaa 表，可选的路径有两条：aaa.a2 索引(针对于 aaa.a2 = 2)；aaa.a3 索引(针对于 aaa.a3 = nkeys.c3 条件)。Best_access_path 函数，其对每条路径的代价算法较为复杂，目前暂时不准备详尽分析其过程。

4.2 best_access_path 函数分析

同样是使用 4.1 章节中的测试语句：

select * from nkeys, aaa where nkeys.c3 = aaa.a3 and aaa.a2 = 2;

4.2.1 总流程分析

根据 4.1 章节的分析，join 涉及到两张表，同时 search_depth 参数设置为 62. N < search_depth，复杂度为 O(N!) = 2! = 2(任何一张表，都可以做驱动表)，同时，查询优化能够找到最优的执行计划。

以上 join，调用 best_access_path 的流程如下：

第一次：

best_access_path(table = aaa, key = 9, a2 索引) -> best_access_path(table = nkeys, key = 2, c3 索引) -> 得到(aaa, nkeys) join 顺序的执行计划，记为 P1。代价为 CP1.

第二次：

best_access_path(table = nkeys, key = 2, c3 索引) -> 得到一个 partial 执行计划 PP2，代价为 CPP2.

由于 CP1 < CPP2，因此执行计划 P1 要由于 partial 执行计划 PP2，P1 为最优执行计划，直接退出即可。为何 P1 的代价 CP1 要小于 PP2 的代价 CPP2，下一章节会给出详细的分析。

4.2.2 代价估计分析

首先，best_access_path 函数，需要计算的代价包括：

3. records_read: 选择此路径，需要读取多少记录。
4. read_time: 选择此路径，需要多少读取时间。

其次，best_extension_by_limited_search 函数，如何计算一个已选择的路径的代价：

- current_record_count = record_count * join->positions[idx].records_read;
  当前路径读取的记录数，是所有路径中的表返回的记录数的乘积。
- current_read_time = read_time + join->positions[idx].read_time;
  当前路径读取需要的时间，是所有路径中的表的读取时间的总和。
- 路径总代价 = current_read_time + (current_record_count / TIME_FOR_COMPARE)
  路径总代价，为底层 IO 时间，加上上层 CPU 时间，这与前面的代价模型一致
- 若 路径总代价 < join->best_read，同时当前路径已经是完整路径，则认为当前路径是目前最优的。
- 在测试的 join 语句中，P1 的代价 CP1 = 2.1990000000000003；PP2 的代价 CPP2 = 1.0 + (6/5) = 2.2000000000000000，因此 CPP2 > CP1，确定 P1 为最优执行计划。

4.2.3 best_access_path 函数流程

接下来，我们详细分析这两个代价在 best_access_path 函数中是如何计算？

best_access_path 主流程：

```
if (s->keyuse)
```

1. 若当前表 s 可以进行索引扫描，则首先判断各索引扫描的代价；若不能进行索引扫描，则直接判断全表扫描的代价。以下将重点分析索引扫描代价计算模型。索引扫描代价计算，主要分这么几种情况：

```
if (found_part == PREV_BITS(uint, keyinfo->key_parts) && !ref_or_null_part)
```

2. **情况 1**：指定查询条件覆盖索引全部列，索引所有的列，都给定了查询条件(没有给定 is null 条件。例如：aaa.a2 = 2；2 为常量条件，同时索引只有 a2 一个字段，满足情况 1 的约束。)

   ```
   if (!found_ref)
   ```

   a) 情况 1.1：指定查询条件均为常量条件，没有与其他表之间的 join 条件(aaa.a2 = 2，2 为常量条件，可走此路径)。
   针对情况 1.1，可参考 best_access_path 函数中的#ReuseRangeEstimateForRef-1#说明。简单来说，就是可以使用单表 range 查询优化中给出的 records 估计值。

   ```
   records = (double) table->quick_rows[key];
   ```

   其中，quick_rows[key]在单表 range 查询优化中赋值。

   b) 情况 1.2：指定的查询条件覆盖索引中的所有列。但是包含了到其他表的引用，不全是常量查询条件，与情况 1.1 有所不同。例如：nkeys.c3 = aaa.a3，对于表 nkeys，其有一个 c3 索引，正好只包含 c3 列，索引列全覆盖；同时查询条件引用到了 aaa 表的 a3 列，非常量条件，不属于情况 1.1，而属于情况 1.2。
   针对情况 1.2，可参考 best_access_path 函数中的#ResueRangeEstimateForRef-2#说明。
   由于情况 1.2 包含了到其他表的引用，因此 range 查询优化可能包含部分条件(常量

条件部分)，或者是未经过 range 查询优化(所有的条件均为引用条件，无常量条件)。

i. 首先，判断存储引擎层面是否收集、设置 rec_per_key 参数。若设置，则

```
records = keyinfo->rec_per_key[keyinfo->keypars - 1]
```

包含所有索引键值的 rec_per_key 的取值。例如 idx1[c1,c2,c3]，那么此处的是 rec_per_key[c1,c2,c3]。

ii. 其次，若 rec_per_key 未收集。那么则根据全表数据量来预估。

```
records = s->records / rec * …
```

一个复杂的计算公式，其中 s->records 为全表数据量；rec 为 unique records 的估计值。

iii. 最后，做一次微调。若有常量查询条件，那么 range 查询优化可能已经针对常量条件部分做了索引的预估。若 i，ii 步骤计算的 records 值 > range 查询优化中计算出来的部分 keys 的 records，则，调整 records 取值.

```
records = table->quick_rows[key];
```

若 table->quick_rows[key]小于 records，则调整 records 的取值。

c) 根据情况 1.1，1.2 计算出来的 records 取值，统计将 s 表通过当前索引加入 partial 执行计划之后，partial 执行计划的 records 开销。

```
partial_plan_records = record_count * records;
```

其中，records_count 为当前 partial plan 的开销，records 为当前表 s 选择索引的开销，根据情况 1.1 或情况 1.2 计算得来。

3. **情况 2**：查询条件未覆盖索引中的所有列，仅仅只包含部分索引列，或者是包含了索引所有列，但是部分列上的条件为 is null 判断(例如：索引 idx(c1,c2,c3)，给定的查询条件为 c1 = 1 and c2 = 2，没有指定 c3 上的查询条件)。

```
if ( table->quick_keys.is_set(key) && !found_ref &&          // C1
   table->quick_key_parts[key] == max_key_part &&            // C2
   table->quick_n_ranges[key] == 1+test(ref_or_null_part) )  // C3
```

a) 情况 2.1：情况 2.1 与情况 1.1 类似。情况 2.1 处理的也是全指定 const 条件的 index scan，与情况 1.1 不同之处在于，所有这些 const 条件，没有覆盖索引全部列。条件 C1 为 range 查询优化选择了此索引同时查询条件中没有引用条件；条件 C2，C3 的原理，可参考 best_access_path 函数中的#ReuseRangeEstimateForRef-3#说明。

```
tmp = records = (double) table->quick_rows[key];
```

在情况 2.1 下，records 的计算，可以直接才有 range 查询优化中计算出来的取值。

b) 情况 2.2：情况 2.2 与情况 1.2 类似。指定的查询条件中，包括到其他表的引用。此时 range 查询优化计算出的 records 是放大的(没有考虑引用条件的过滤，因为引用条件在真正执行之前，无法获取其实际取值)。

i. 首先，判断存储引擎层面是否收集、设置 rec_per_key 参数。若收集，则将 records 设置为 max_key_part 的 rec_per_key 取值。

```
records = keyinfo->rec_per_key[max_key_part - 1];
```

当然，此时也需要微调，若 rec_per_key 的取值过于乐观，过大，甚至大于 range 查询优化中计算的部分 const 查询条件返回的 records，则重设 records 取值。

```
records = table->quick_rows[key];
```

ii. 若 rec_per_key 未收集。则根据以下算法，计算 records。

```
records = (x * (b-a) + a*c - b) / (c - 1)
b = records matched by whole key
a = records matched by first key part (1% of all records)
c = number of key parts in key
x = used key parts (1 <= x <= c)
```

iii. 对于 i，ii 步骤中计算出来的 records 进行微调。仍旧使用一直以来的调整策略。若计算出来的 records 取值过于乐观，甚至大于 range 查询优化针对部分 const 条件计算而来的 records 取值，则重设 records。

```
tmp = records = table->quick_rows[key];
```

4. 在情况 1，情况 2 确定了当前表 s 在给定索引 key 下的 records 计算，并得到 partial_plan_records 之后，判断当前 key 是否为最优选择；若是，则将当前表 s 的最优访问路径设置为 key；否则跳过 key，选择下一个表 s 的可选路径，继续。

```
if (partial_plan_records < best_time – records/ TIME_FOR_COMPARE) {
    best_time = tmp + records / TIME_FOR_COMPARE;
    best = tmp;
    best_records = records;
    best_key = start_key;
}
```

其中，best_time 为当前 partial plan 的最优路径(包括 s 表)；tmp = partial_plan_records。

5. 继续步骤 1，直到完成 s->keyuse 中所有的索引，根据步骤 4 的判断，获取当前表 s 上的最优索引。
6. 在步骤 5 完成，遍历所有可选索引之后，判断是否需要对全表扫描进行判断。mysql 查询优化，将全表扫描的权重放的较低，一般情况下，不会在 join 情况下，选择全表扫描做 nestloop join，否则性能一定较差。
7. 在完成以上 6 个步骤之后，best_access_path 函数结束。当前表 s，s 的最优路径 key，都已经得到，同时加入 s 表之后，当前 partial plan 的代价已经得到。在 best_extension_by_limited_search 函数中，根据 search_depth 参数的设置，会获取一个完整的 plan，然后将此 plan 与已有最优的 plan 进行比较，若更优则替换；否则放弃当前 plan，进行寻找下一个。

4.3 optimizer_search_depth 参数

以上提到的 greedy_search+best_extension_by_limited_search 函数，通过 search_depth 参数控制递归调用的深度。而 search_depth 参数，可通过 optimizer_search_depth 来设置。

一般而言，如果 optimizer_search_depth 设置过大，那么 join 时，获取最优执行计划的代价十分巨大。

optimizer_search_depth = join tables 的数量，一定能获得最优执行计划(根据 mysql 的代价估计模型)，但是计算代价大。

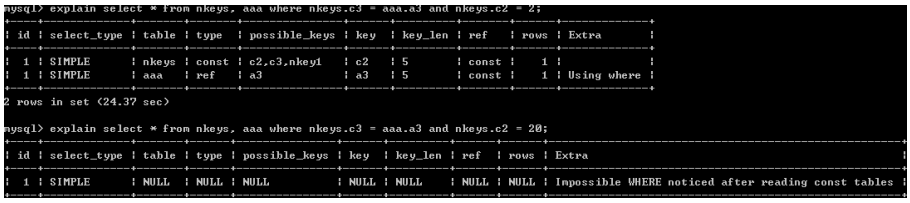
optimizer_search_depth < join tables 的数量，获取的执行计划，是局部最优，但是计算代价小。

optimizer_search_depth 参数，对于单表查询无意义。

http://dev.mysql.com/doc/refman/5.0/en/controlling-optimizer.html 中，有 mysql 对于此参数的说明，可以参考。

4.4 多表 join 查询总结

5. join 的查询优化，是一个复杂的过程。
6. mysql 在 join 的查询优化中，同样为指定 unique 查询的 sql 做了优化，优化方案与单表 unique 查询类似：若发现指定的 unique 无法找到匹配的记录，直接返回，而不产生真正的执行计划，如下图所示：



当 mysql 查询优化发现 nkeys.c2 = 20 无法匹配到记录，直接返回。并不会继续生成完整的执行计划。

7. 在 best_access_plan 函数中，对表 s 进行最优路径选择时，会充分利用 range 查询优化的结果。若无法利用 range 查询优化结果，还会使用的统计信息包括：

7.1. rec_per_key

每一个 key，包含多少记录。在存储引擎，info 函数中进行收集。

7.2. records_in_table

当前表上，有多少记录。在存储引擎，info 函数中进行收集。

8. 对于多表 join 查询，rec_per_key, records_in_table 两个统计信息相当重要，直接影响到最后的执行计划选择。因此在引擎实现中，要充分考虑这两个统计信息的收集算法。如果能够持久化这两个统计信息，就基本上能够保证 join 查询的执行计划稳定。
9. 下一章节，将分析 INNODB 如何收集 records_in_table，rec_per_key 这两个统计信息。

4.5 多表 join 查询优化-5.6 增强

5 统计信息

5.1 统计信息收集

show index from nkeys;

函数调用流程：

sql_parse.cc::mysql_execute_command(lex->sql_command == SQLCOM_SHOW_KEYS) ->

sql_show.cc::get_schema_stat_record ->

1. ha_innobase::info(HA_STATUS_VARIABLE | HA_STATUS_NO_LOCK | HA_STATUS_TIME)(info

函数，实现统计信息收集功能，宏定义说明了需要收集的统计信息的类型) ->

2. dict0dict.c::dict_update_statistics(ib_table)(HA_STATUS_TIME，指定此参数时进行统计信息的重新收集) ->
3. dict_update_statistics_low(统计信息收集主函数，遍历表上的所有 index，收集统计信息，设置到 dict_index 与 dict_table 结构之中) -> btr_estimate_number_of_different_key_vals ->
   a) dict_index
      i. 索引总页面数；叶节点页面数
      ii. 索引中不同键值个数，对于索引中的每一列，都需要计算。

```
for (i = 0; i < BTR_KEY_VAL_ESTIMATE_N_PAGES; i++ )
```

      1. 随机在索引中定位 BTR_KEY_VAL_ESTIMATE_N_PAGES = 8 次页面。一定是 8 次，无论实际页面是否小于 8.

```
next_rec = page_rec_get_next(rec);
cmp_rec_rec_with_match(rec, next_rec, &matched_fields);
```

      2. 读取页面中的索引项，与前一项进行对比。找到第一个不相同的列值后退出，matched_fields 为相同列的个数。

```
for (j = matched_fields + 1; j <= n_cols; j++)
    n_diff[j]++;
```

      3. 根据 matched_fields，将所有之后的列对应的 n_diff 数组++，n_diff[j]++ (其中，n_cols 为当前索引中，能够确定 unique 键值的索引列数)

```
for (j = 0; j <= n_cols; j++)
    index->stat_n_diff_key_vals[j] =
        ((n_diff[j] *index->stat_n_leaf_pages +
        BTR_KEY_VAL_ESTIMATE_N_PAGES +
        total_external_size + not_emptry_flag) /
        (BTR_KEY_VAL_ESTIMATE_N_PAGES + total_external_size)
```

      4. 随机 8 个 page 扫描完毕，根据统计信息 n_diff，计算最终的 index 中，每一列不同取值的数量(注意，此处是每一列的不同取值的数量，而非 rec_per_key)。
   b) dict_table
      i. 表总行数。table->stat_n_rows = index->stat_n_diff_key_vals[dict_index_get_n_unique(index)]; 总行数，为表第一个索引，unique key 的个数。
      ii. 聚簇索引页面数
      iii. 二级索引页面数
      iv. table->stat_initialized = 1; 统计信息为重新收集的
      v. table->stat_modified_counter = 0; 信息收集之后，表上无 DML 操作发生，统计信息是准确的

4. 回到 info 函数，如果调用 info 函数时，指定了参数 HA_STATUS_CONST，那么 info 函数将会更新表中每一个索引，每一个索引对应的每一个列组合的 rec_per_key 取值。(例如：考虑 nkeys 表的 c1c2 索引，那么将会计算 rec_per_key(c1), rec_per_key(c1c2)的取值，而不会计算 rec_per_key(c2)，单独指定 c2 列无意义，并不会选择 c1c2 索引进行查询)。

   a) 计算 rec_per_key

```
for (I = 0; I < table->s->keys; i++)                    // 遍历索引索引
    for (j = 0; j < table->key_info[i].key_parts; j++)  // 遍历索引中的所有列
        rec_per_key = records / (index->stat_n_diff_key_vals[j+1]);
```

根据前面收集的不同键值数量，与表记录数量，计算最终，rec_per_key 的取值。

b) 调整 rec_per_key

```
rec_per_key = rec_per_key / 2;
```

由于 mysql 查询优化，更倾向于使用全表扫描，因此在此处做调整，将 rec_per_key 减半，减少索引 range 查询返回记录数，提高查询优化选择索引的概率。

5.2 统计信息更新

在以下过程中，INNODB 将会更新表的统计信息：

10. analyze table
    ha_innobase::analyze ->
    info(HA_STATUS_TIME | HA_STATUS_CONST | HA_STATUS_VARIABLE)
11. open table
    ha_innobase::open -> 第一次 open 表时需要调用，后续的 open，都不调用
    info(HA_STATUS_NO_LOCK | HA_STATUS_VARIABLE | HA_STATUS_CONST)
12. 查询优化前，此时不重新收集统计信息，只计算
    sql_select.cc::make_join_statistics ->
    info(HA_STATUS_VARIABLE | HA_STATUS_NO_LOCK);

5.3 统计信息收集总结

13. 在第一次 open table 时，会重新收集全表统计信息，因此第一次 open table 是较慢的过程。庆幸的是，Innodb 会将打开的表 cache 起来，因此并不会多次 open。
14. 结合 mysql 上层的查询优化可知，range 查询优化并不会使用 open table 过程中收集的统计信息。在此处看来，也无法使用，因为 mysql 收集的统计信息过于简单，只能回答 rec_per_key，无法回答 rec_in_range，rec_in_range 需要通过调用 records_in_range 函数计算。
15. records_in_table，rec_per_key 统计信息，在 mysql 多表 join 时使用。多表 join，ref 的属性，在语句执行前，并不能获得 key 取值，因此只能通过 rec_per_key 预估 join 的内表将会返回多少记录。

6 查询优化总结

16. 是否可以借鉴 oracle 中的执行计划，硬解析与软解析分离的实现思路？同样为 mysql 保留执行计划，降低查询优化的代价。从以上的分析看来，mysql 的查询优化，并不如想象中高效，同样是一个缓慢的过程。
17. 释放可以借鉴 oracle 中的统计信息持久化策略，持久化 rec_per_key，records_in_table 两类十分重要的统计信息，持久化统计信息，基本上能够保证 mysql join 操作的执行计

划的稳定性。

7 参考文献

[1] http://weevilgenius.net/2010/09/mysql-explain-reference/ MSYQL Explain – Reference

[2] Mysql 查询优化走错执行计划分析

[3] http://jorgenloland.blogspot.com/2012/04/improvements-for-many-table-joins-in.html Improvements for many-table joins in MySQL 5.6

[4] http://mysqloptimizerteam.blogspot.co.uk/ MySQL Optimizer Team blogs

[5] http://forge.mysql.com/wiki/MySQL_Internals_Optimizer MySQL Internals Optimizer

[6] http://forge.mysql.com/wiki/MySQL_Internals_Optimizer_tracing MySQL Internals Optimizer tracing

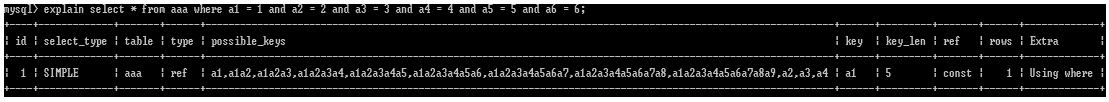
[7]

附录一

```
create table aaa (
   a1 int,
   a2 int,
   a3 int,
   a4 int,
   a5 int,
   a6 int,
   a7 int,
   a8 int,
   a9 int
) engine = innodb;

create index a1 on aaa (a1);
create index a1a2 on aaa (a1,a2);
create index a1a2a3 on aaa (a1,a2,a3);
create index a1a2a3a4 on aaa (a1,a2,a3,a4);
create index a1a2a3a4a5 on aaa (a1,a2,a3,a4,a5);
create index a1a2a3a4a5a6 on aaa (a1,a2,a3,a4,a5,a6);
create index a1a2a3a4a5a6a7 on aaa (a1,a2,a3,a4,a5,a6,a7);
create index a1a2a3a4a5a6a7a8 on aaa (a1,a2,a3,a4,a5,a6,a7,a8);
create index a1a2a3a4a5a6a7a8a9 on aaa (a1,a2,a3,a4,a5,a6,a7,a8a9);
create index a2 on aaa (a2);
create index a3 on aaa (a3);
create index a4 on aaa (a4);
```

explain select * from aaa where a1 = 1 and a2 = 2 and a3 = 3 and a4 = 4 and a5 = 5 and a6 = 6;



mysql> explain select * from aaa where a1 = 1 and a2 = 2 and a3 = 3 and a4 = 4 and a5 = 5 and a6 = 6;

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	aaa	ref	a1,a1a2,a1a2a3,a1a2a3a4,a1a2a3a4a5,a1a2a3a4a5a6,a1a2a3a4a5a6a7,a1a2a3a4a5a6a7a8,a1a2a3a4a5a6a7a8a9,a2,a3,a4	a1	5	const	1	Using where

以上的查询，由于 a1 列打头的索引太多，导致 possible_keys 太多，一共有 9 个 a1+3 个其他= 12 个 possible_keys，每个 possible_keys 都需要做两次的 search_path to leaf page。
导致在查询优化的过程中，就有 24 次的 IO(假设叶节点不命中)，这个开销要远远大于查询本身的开销。
我的建议是，以同一个列打头的索引，越少越好

附录二

测试语句：
语句一：select c3, c5 from nkeys where c3 > 0;
语句二：select c3, c5 from nkeys where c3 > 0 for update;

源代码调整:
在 Opt_range.cc 文件的 get_key_scans_params 函数中，将索引覆盖扫描计算出的 found_read_time 增加，如下：

```
/*
          We can resolve this by only reading through this key.
          0.01 is added to avoid races between range and 'index' scan.
        */
    found_read_time= get_index_only_read_time(param,found_records,keynr) +
cpu_cost + 0.01;
    found_read_time += 1000.0;
```

保证索引覆盖扫描的 read_time 一定大于全表扫描，此时看 mysql 最终是否还是会选择索引覆盖扫描？

在 innodb 的 row0sel.c::row_search_for_mysql 中设置断点，最后发现，查询仍旧走的是 nkey1 索引(c3,c5)，虽然此索引计算出来的代价 1001.6 >>全表扫描代价 2.2.

为什么会如此，是因为 mysql 在代价估算完成之后，针对索引覆盖扫描，仍旧会再次优化，具体可见 3.1 单表 range 查询章节的第 6 步以及下面的详细分析：

```
// 若当前为全表扫描，JT_ALL，同时有一个或者多个索引可以进行索引覆盖扫描，则优先
// 选择索引覆盖扫描，进行查询
sql_select.cc::make_join_readinfo ->
    //find_shortest_key 函数，遍历表上的所有可以进行覆盖扫描的索引
    //找出其中键值(key_length)最小的索引，最为最优的覆盖扫描索引
    //并且以此索引，替换全表扫描
    tab->index = find_shortest_key(table, & table->covering_keys) ->
    // 进行索引覆盖扫描
    tab->read_first_record = join_read_first;
    tab->type = JT_NEXT;
```

不足分析：

针对语句二，select for update，mysql 同样做了 Coverage index scan 的优化，最后选择的仍旧是(c3, c5)的组合索引。但是，由于当前读需要回聚簇索引加事务锁，因此回表是必须的，不能做索引覆盖扫描。这个优化对于当前读来说，有可能适得其反。

附录三

best_extension_by_limited_search 函数实现伪代码：

```
@code
    procedure best_extension_by_limited_search(
      pplan in,             // in, partial plan of tables-joined-so-far
      pplan_cost,           // in, cost of pplan
      remaining_tables,     // in, set of tables not referenced in pplan
      best_plan_so_far,     // in/out, best plan found so far
      best_plan_so_far_cost,// in/out, cost of best_plan_so_far
      search_depth)         // in, maximum size of the plans being considered
    {
      for each table T from remaining_tables
      {
        // Calculate the cost of using table T as above
        cost = complex-series-of-calculations;

        // Add the cost to the cost so far.
        pplan_cost+= cost;

        if (pplan_cost >= best_plan_so_far_cost)
```

```
        // pplan_cost already too great, stop search
        continue;

      pplan= expand pplan by best_access_method;
      remaining_tables= remaining_tables - table T;
      if (remaining_tables is not an empty set
          and
          search_depth > 1)
      {
        best_extension_by_limited_search(pplan, pplan_cost,
                                         remaining_tables,
                                         best_plan_so_far,
                                         best_plan_so_far_cost,
                                         search_depth - 1);
      }
      else
      {
        best_plan_so_far_cost= pplan_cost;
        best_plan_so_far= pplan;
      }
    }
  }
  @endcode
```

附录四

nkeys 表定义：

```
mysql> show create table nkeys;
+-------+-------------------------------
---------------------------------+
| Table | Create Table
                                          |
+-------+-------------------------------
---------------------------------+
| nkeys | CREATE TABLE `nkeys` (
  `c1` int(11) NOT NULL,
  `c2` int(11) DEFAULT NULL,
  `c3` int(11) DEFAULT NULL,
  `c4` int(11) DEFAULT NULL,
  `c5` int(11) DEFAULT NULL,
  PRIMARY KEY (`c1`),
  UNIQUE KEY `c2` (`c2`),
  UNIQUE KEY `c3` (`c3`),
```

```
  UNIQUE KEY `c4` (`c4`),
  KEY `nkey1` (`c3`,`c5`),
  KEY `c1c2` (`c1`,`c2`),
  KEY `c1c2c3` (`c1`,`c2`,`c3`),
  KEY `c1c2c3c4` (`c1`,`c2`,`c3`,`c4`)
) ENGINE=InnoDB DEFAULT CHARSET=gbk
```

aaa 表定义：

```
create table aaa (
  a1 int,
  a2 int,
  a3 int,
  a4 int,
  a5 int,
  a6 int,
  a7 int,
  a8 int,
  a9 int
) engine = innodb;

create index a1 on aaa (a1);
create index a1a2 on aaa (a1,a2);
create index a1a2a3 on aaa (a1,a2,a3);
create index a1a2a3a4 on aaa (a1,a2,a3,a4);
create index a1a2a3a4a5 on aaa (a1,a2,a3,a4,a5);
create index a1a2a3a4a5a6 on aaa (a1,a2,a3,a4,a5,a6);
create index a1a2a3a4a5a6a7 on aaa (a1,a2,a3,a4,a5,a6,a7);
create index a1a2a3a4a5a6a7a8 on aaa (a1,a2,a3,a4,a5,a6,a7,a8);
create index a1a2a3a4a5a6a7a8a9 on aaa (a1,a2,a3,a4,a5,a6,a7,a8a9);
create index a2 on aaa (a2);
create index a3 on aaa (a3);
create index a4 on aaa (a4);
```