Mysql内核查询成本计算实战

Optimizer Trace

有道云链接：http://note.youdao.com/noteshare?id=6c03ef8fa44c1d2d15e319e2f2ed5a6a&sub=3E5FB31F083F4AA69F55B65D395E8241

对于MySQL5.6之前的版本来说，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。我们可能有这样的疑问：“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？为什么MySQL一定要全文扫描，不用索引呢？”

在MySQL 5.6以及之后的版本中，MySQL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，这个功能的解释请参考第四期MySQL的《04-VIP-Mysql索引优化实战一》。

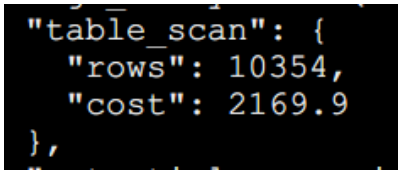
比如对于下面这个SQL语句：

```
SELECT * FROM order_exp WHERE order_no IN ('DD00_6S', 'DD00_9S',
'DD00_10S') AND  expire_time> '2021-03-22 18:28:28' AND expire_time<=
'2021-03-22 18:35:09' AND insert_time> expire_time AND order_note LIKE '%7
排1%' AND  order_status = 0;
```

我们执行如下的命令：

```
SET optimizer_trace="enabled=on";
SELECT * FROM order_exp WHERE order_no IN ('DD00_6S', 'DD00_9S',
'DD00_10S') AND  expire_time> '2021-03-22 18:28:28' AND expire_time<=
'2021-03-22 18:35:09' AND insert_time> expire_time AND order_note LIKE '%7
排1%' AND  order_status = 0;
SELECT * FROM information_schema.OPTIMIZER_TRACE\G
```

可以看见全表扫描的成本：2169.9



使用索引idx_order_no的成本为72.61：

```
"index": "idx_order_no",
"ranges": [
  "DD00_10S <= order_no <= DD00_10S",
  "DD00_6S <= order_no <= DD00_6S",
  "DD00_9S <= order_no <= DD00_9S"
],
"index_dives_for_eq_ranges": true,
"rowid_ordered": false,
"using_mrr": false,
"index_only": false,
"rows": 58,
"cost": 72.61,
"chosen": true
```

使用索引idx_expire_time的成本为47.81：

```
"index": "idx_expire_time",
"ranges": [
  "0x99a92d271c < expire_time <= 0x99a92d28c9"
],
"index_dives_for_eq_ranges": true,
"rowid_ordered": false,
"using_mrr": false,
"index_only": false,
"rows": 39,
"cost": 47.81,
"chosen": true
```

最终MySQL使用了idx_expire_time作为这个SQL查询过程中索引：

```
1  explain SELECT * FROM order_exp WHERE order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S') AND
   expire_time> '2021-03-22 18:28:28' AND expire_time<= '2021-03-22 18:35:09' AND insert_time>
   expire_time AND order_note LIKE '%7排1%' AND  order_status = 0;
```

结果 1 | 剖析 | 状态

	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	order_exp	(Null)	range	idx_order_no,idx_expire_time	idx_expire_time	5	(Null)	

因为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。

但是这些成本怎么来的呢？搞明白了这些成本的由来，就不会再有诸如“为什么MySQL一定要全文扫描，不用索引呢？为什么MySQL要用A索引不用B索引之类的疑问？”了，因为以上的答案都可以用成本分析来解答。所以接下来，我们就要深入MySQL的内核来看看这些成本是如何计算的。

什么是成本

MySQL执行一个查询可以有不同的执行方案，它会选择其中成本最低，或者说代价最低的那种方案去真正的执行查询。什么是执行成本呢？其实在MySQL中一条查询语句的执行成本是由下边这两个方面组成的：

I/O成本

我们的表经常使用的MyISAM、InnoDB存储引擎都是将数据和索引都存储到磁盘上的，当我们想查询表中的记录时，需要先把数据或者索引加载到内存中然后再操作。这个从磁盘到内存这个加载的过程损耗的时间称之为I/O成本。

CPU成本

读取以及检测记录是否满足对应的搜索条件、对结果集进行排序等这些操作损耗的时间称之为CPU成本。

对于InnoDB存储引擎来说，页是磁盘和内存之间交互的基本单位，MySQL规定读取一个页面花费的成本默认是1.0，读取以及检测一条记录是否符合搜索条件的成本默认是0.2。1.0、0.2这些数字称之为成本常数，这两个成本常数我们最常用到，当然还有其他的成本常数。

注意，不管读取记录时需不需要检测是否满足搜索条件，其成本都算是0.2。

单表查询的成本

MySQL查询成本计算实战

在一条单表查询语句真正执行之前，MySQL的查询优化器会找出执行该语句所有可能使用的方案，对比之后找出成本最低的方案，这个成本最低的方案就是所谓的执行计划，之后才会调用存储引擎提供的接口真正的执行查询，这个过程总结一下就是这样：

1、根据搜索条件，找出所有可能使用的索引

2、计算全表扫描的代价

3、计算使用不同索引执行查询的代价

4、对比各种执行方案的代价，找出成本最低的那一个

我们依然以上面的查询语句来分析：

```
SELECT * FROM order_exp WHERE order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S') AND  expire_time> '2021-03-22 18:28:28' AND expire_time<= '2021-03-22 18:35:09' AND insert_time> expire_time AND order_note LIKE '%7排1%' AND  order_status = 0;
```

我们一步一步分析一下。

1. 根据搜索条件，找出所有可能使用的索引

MySQL把一个查询中可能使用到的索引称之为possible keys。

我们分析一下上边查询中涉及到的几个搜索条件：

order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S') ，这个搜索条件可以使用二级索引idx_order_no。

expire_time> '2021-03-22 18:28:28' AND expire_time<= '2021-03-22 18:35:09'，这个搜索条件可以使用二级索引idx_expire_time。

insert_time> expire_time，这个搜索条件的索引列由于没有和常数比较，所以并不能使用到索引。

order_note LIKE '%hello%'，order_note即使有索引，但是通过LIKE操作符和以通配符开头的字符串做比较，不可以适用索引。

order_status = 0，由于该列上只有联合索引，而且不符合最左前缀原则，所以不会用到索引。

综上所述，上边的查询语句可能用到的索引，也就是possible keys只有idx_order_no,idx_expire_time。

```
mysql> explain SELECT * FROM order_exp WHERE order_no
 18:35:09' AND insert_time> expire_time AND order_not
*************************** 1. row ******************
           id: 1
  select_type: SIMPLE
        table: order_exp
   partitions: NULL
         type: range
possible_keys: idx_order_no,idx_expire_time
          key: idx_expire_time
      key_len: 5
          ref: NULL
         rows: 39
     filtered: 0.13
        Extra: Using index condition; Using where
1 row in set, 1 warning (0.00 sec)
```

2. 计算全表扫描的代价

对于InnoDB存储引擎来说，全表扫描的意思就是把聚簇索引中的记录都依次和给定的搜索条件做一下比较，把符合搜索条件的记录加入到结果集，所以需要将聚簇索引对应的页面加载到内存中，然后再检测记录是否符合搜索条件。由于查询成本=I/O成本+CPU成本，所以计算全表扫描的代价需要两个信息：

聚簇索引占用的页面数

该表中的记录数

这两个信息从哪来呢？MySQL为每个表维护了一系列的统计信息，关于这些统计信息是如何收集起来的我们放在后边再说，现在看看怎么查看这些统计信息。

MySQL给我们提供了SHOW TABLE STATUS语句来查看表的统计信息，如果要看指定的某个表的统计信息，在该语句后加对应的LIKE语句就好了，比方说我们要查看order_exp这个表的统计信息可以这么写：

```
SHOW TABLE STATUS LIKE 'order_exp'\G
```

```
mysql> SHOW TABLE STATUS LIKE 'order_exp'\G
*************************** 1. row *********
           Name: order_exp
         Engine: InnoDB
        Version: 10
     Row_format: Dynamic
           Rows: 10354
 Avg_row_length: 153
    Data_length: 1589248
Max_data_length: 0
   Index_length: 1130496
      Data_free: 4194304
 Auto_increment: 10819
    Create_time: 2021-03-31 15:39:43
    Update_time: NULL
     Check_time: NULL
      Collation: utf8_general_ci
       Checksum: NULL
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

出现了很多统计选项，但我们目前只需要两个：

Rows

本选项表示表中的记录条数。对于使用MyISAM存储引擎的表来说，该值是准确的，对于使用InnoDB存储引擎的表来说，该值是一个估计值。从查询结果我们也可以看出来，由于我们的order_exp表是使用InnoDB存储引擎的，所以虽然实际上表中有10567条记录，但是SHOW TABLE STATUS显示的Rows值只有10354条记录。

Data_length

本选项表示表占用的存储空间字节数。使用MyISAM存储引擎的表来说，该值就是数据文件的大小，对于使用InnoDB存储引擎的表来说，该值就相当于聚簇索引占用的存储空间大小，也就是说可以这样计算该值的大小：

Data_length = 聚簇索引的页面数量 x 每个页面的大小

我们的order_exp使用默认16KB的页面大小，而上边查询结果显示Data_length的值是1589248，所以我们可以反向来推导出聚簇索引的页面数量：

聚簇索引的页面数量 = 1589248 ÷ 16 ÷ 1024 = 97

我们现在已经得到了聚簇索引占用的页面数量以及该表记录数的估计值，所以就可以计算全表扫描成本了。

现在可以看一下全表扫描成本的计算过程：

I/O成本

97 x 1.0 + 1.1 = 98.1

97指的是聚簇索引占用的页面数，1.0指的是加载一个页面的IO成本常数，后边的1.1是一个微调值。

TIPS：MySQL在真实计算成本时会进行一些微调，这些微调的值是直接硬编码到代码里的，没有注释而且这些微调的值十分的小，并不影响我们大方向的分析。

CPU成本

10354x 0.2 + 1.0 = 2071.8

10354指的是统计数据中表的记录数，对于InnoDB存储引擎来说是一个估计值，0.2指的是访问一条记录所需的CPU成本常数，后边的1.0是一个微调值。

总成本：

98.1 + 2071.8= 2169.9

综上所述，对于order_exp的全表扫描所需的总成本就是2169.9。

TIPS：我们前边说过表中的记录其实都存储在聚簇索引对应B+树的叶子节点中，所以只要我们通过根节点获得了最左边的叶子节点，就可以沿着叶子节点组成的双向链表把所有记录都查看一遍。

也就是说全表扫描这个过程其实有的B+树非叶子节点是不需要访问的，但是MySQL在计算全表扫描成本时直接使用聚簇索引占用的页面数作为计算I/O成本的依据，是不区分非叶子节点和叶子节点的。

3. 计算使用不同索引执行查询的代价

从第1步分析我们得到，上述查询可能使用到idx_order_no，idx_expire_time这两个索引，我们需要分别分析单独使用这些索引执行查询的成本，最后还要分析是否可能使用到索引合并（什么是索引合并，我们后面的课程会讲到）。这里需要提一点的是，MySQL查询优化器先分析使用唯一二级索引的成本，再分析使用普通索引的成本，我们这里两个索引都是普通索引，先算哪个都可以。我们先分析idx_expire_time的成本，然后再看使用idx_order_no的成本。

使用idx_expire_time执行查询的成本分析

idx_expire_time对应的搜索条件是：expire_time> '2021-03-22 18:28:28' AND expire_time<= '2021-03-22 18:35:09'，也就是说对应的范围区间就是：('2021-03-22 18:28:28', '2021-03-22 18:35:09')。

使用idx_expire_time搜索会使用用二级索引 + 回表方式的查询，MySQL计算这种查询的成本依赖两个方面的数据：

1、范围区间数量

不论某个范围区间的二级索引到底占用了多少页面，查询优化器认为读取索引的一个范围区间的I/O成本和读取一个页面是相同的。本例中使用idx_expire_time的范围区间只有一个，所以相当于访问这个范围区间的二级索引付出的I/O成本就是：1 x 1.0 = 1.0

2、需要回表的记录数

优化器需要计算二级索引的某个范围区间到底包含多少条记录，对于本例来说就是要计算idx_expire_time在('2021-03-22 18:28:28', '2021-03-22 18:35:09')这个范围区间中包含多少二级索引记录，计算过程是这样的：

步骤1：先根据expire_time> '2021-03-22 18:28:28'这个条件访问一下idx_expire_time对应的B+树索引，找到满足expire_time> '2021-03-22 18:28:28'这个条件的第一条记录，我们把这条记录称之为区间最左记录。我们前头说过在B+数树中定位一条记录的过程是很快的，是常数级别的，所以这个过程的性能消耗是可以忽略不计的。

步骤2：然后再根据expire_time<= '2021-03-22 18:35:09'这个条件继续从idx_expire_time对应的B+树索引中找出最后一条满足这个条件的记录，我们把这条记录称之为区间最右记录，这个过程的性能消耗也可以忽略不计的。

步骤3：如果区间最左记录和区间最右记录相隔不太远（在MySQL 5.7这个版本里，只要相隔不大于10个页面即可），那就可以精确统计出满足expire_time> '2021-03-22

18:28:28’ AND expire_time<= ‘2021-03-22 18:35:09’条件的二级索引记录条数。否则只沿着区间最左记录向右读10个页面，计算平均每个页面中包含多少记录，然后用这个平均值乘以区间最左记录和区间最右记录之间的页面数量就可以了。那么问题又来了，怎么估计区间最左记录和区间最右记录之间有多少个页面呢？解决这个问题还得回到B+树索引的结构中来。

我们假设区间最左记录在页b中，区间最右记录在页c中，那么我们想计算区间最左记录和区间最右记录之间的页面数量就相当于计算页b和页c之间有多少页面，而它们父节点中记录的每一条目录项记录都对应一个数据页，所以计算页b和页c之间有多少页面就相当于计算它们父节点（也就是页a）中对应的目录项记录之间隔着几条记录。在一个页面中统计两条记录之间有几条记录的成本就很小了。

不过还有问题，如果页b和页c之间的页面实在太多，以至于页b和页c对应的目录项记录都不在一个父页面中怎么办？既然是树，那就继续递归，之前我们说过一个B+树有4层高已经很了不得了，所以这个统计过程也不是很耗费性能。

知道了如何统计二级索引某个范围区间的记录数之后，就需要回到现实问题中来，MySQL根据上述算法测得idx_expire_time在区间(’2021-03-22 18:28:28’ ,’2021-03-22 18:35:09’)之间大约有39条记录。

```
explain SELECT * FROM order_exp WHERE expire_time> ’2021-03-22 18:28:28’ AND expire_time<= ’2021-03-22 18:35:09’;
```

```
mysql> explain SELECT * FROM order_exp WHERE expire_time> '2021-03-22 18:28:28' AND expire_time<= '2021-03-22 18:
+----+-------------+-----------+------------+-------+----------------+----------------+---------+------+------+
| id | select_type | table     | partitions | type  | possible_keys  | key            | key_len | ref  | rows |
+----+-------------+-----------+------------+-------+----------------+----------------+---------+------+------+
|  1 | SIMPLE      | order_exp | NULL       | range | idx_expire_time | idx_expire_time | 5       | NULL |   39 |
+----+-------------+-----------+------------+-------+----------------+----------------+---------+------+------+
1 row in set, 1 warning (0.00 sec)
```

读取这39条二级索引记录需要付出的CPU成本就是：

39 x 0.2 + 0.01 = 7.81

其中39是需要读取的二级索引记录条数，0.2是读取一条记录成本常数，0.01是微调。

在通过二级索引获取到记录之后，还需要干两件事儿：

1、根据这些记录里的主键值到聚簇索引中做回表操作

MySQL评估回表操作的I/O成本依旧很简单粗暴，他们认为每次回表操作都相当于访问一个页面，也就是说二级索引范围区间有多少记录，就需要进行多少次回表操作，也就是需要进行多少次页面I/O。我们上边统计了使用idx_expire_time二级索引执行查询时，预计有39 条二级索引记录需要进行回表操作，所以回表操作带来的I/O成本就是：

39 x 1.0 = 39 .0

其中39 是预计的二级索引记录数，1.0是一个页面的I/O成本常数。

2、回表操作后得到的完整用户记录，然后再检测其他搜索条件是否成立

回表操作的本质就是通过二级索引记录的主键值到聚簇索引中找到完整的用户记录，然后再检测除expire_time> '2021-03-22 18:28:28' AND expire_time< '2021-03-22 18:35:09'这个搜索条件以外的搜索条件是否成立。

因为我们通过范围区间获取到二级索引记录共39 条，也就对应着聚簇索引中39 条完整的用户记录，读取并检测这些完整的用户记录是否符合其余的搜索条件的CPU成本如下：

39 x 0.2 =7.8

其中39 是待检测记录的条数，0.2是检测一条记录是否符合给定的搜索条件的成本常数。

所以本例中使用idx_expire_time执行查询的成本就如下所示：

I/O成本：

1.0 + 39 x 1.0 = 40 .0（范围区间的数量 + 预估的二级索引记录条数）

CPU成本：

39 x 0.2 + 0.01 + 39 x 0.2 = 15.61 （读取二级索引记录的成本 + 读取并检测回表后聚簇索引记录的成本）

综上所述，使用idx_expire_time执行查询的总成本就是：

40 .0 + 15.61 = 55.61

使用idx_order_no执行查询的成本分析

idx_order_no对应的搜索条件是：order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S')，也就是说相当于3个单点区间。

与使用idx_expire_time的情况类似，我们也需要计算使用idx_order_no时需要访问的范围区间数量以及需要回表的记录数，计算过程与上面类似，我们不详列所有计算步骤和说明了。

范围区间数量

使用idx_order_no执行查询时很显然有3个单点区间，所以访问这3个范围区间的二级索引付出的I/O成本就是：

3 x 1.0 = 3.0

需要回表的记录数

由于使用idx_expire_time时有3个单点区间，所以每个单点区间都需要查找一遍对应的二级索引记录数，三个单点区间总共需要回表的记录数是58。

explain SELECT * FROM order_exp WHERE order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S');

```
mysql> explain SELECT * FROM order_exp WHERE order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S');
+----+-------------+-----------+------------+-------+---------------+--------------+---------+------+------+
| id | select_type | table     | partitions | type  | possible_keys | key          | key_len | ref  | rows |
+----+-------------+-----------+------------+-------+---------------+--------------+---------+------+------+
|  1 | SIMPLE      | order_exp | NULL       | range | idx_order_no  | idx_order_no | 152     | NULL |   58 |
+----+-------------+-----------+------------+-------+---------------+--------------+---------+------+------+
1 row in set, 1 warning (0.00 sec)
```

读取这些二级索引记录的CPU成本就是：58 x 0.2 + 0.01 = 11.61

得到总共需要回表的记录数之后，就要考虑：

根据这些记录里的主键值到聚簇索引中做回表操作，所需的I/O成本就是：58 x 1.0 = 58.0

回表操作后得到的完整用户记录，然后再比较其他搜索条件是否成立

此步骤对应的CPU成本就是：

58 x 0.2 = 11.6

所以本例中使用idx_order_no执行查询的成本就如下所示：

I/O成本：

3.0 + 58 x 1.0 = 61.0（范围区间的数量 + 预估的二级索引记录条数）

CPU成本：

58 x 0.2 + 58 x 0.2 + 0.01 = 23.21 （读取二级索引记录的成本 + 读取并检测回表后聚簇索引记录的成本）

综上所述，使用idx_order_no执行查询的总成本就是：

61.0 + 23.21 = 84.21

是否有可能使用索引合并（Index Merge）

本例中SQL语句不满足索引合并的条件，所以并不会使用索引合并。而且MySQL查询优化器计算索引合并成本的算法也比较麻烦，我们不去了解。

4. 对比各种方案，找出成本最低的那一个

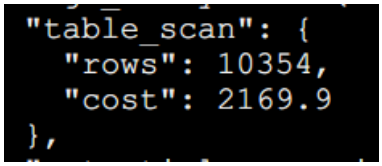
下边把执行本例中的查询的各种可执行方案以及它们对应的成本列出来：

全表扫描的成本：2169.9

使用idx_expire_time的成本：55.61

使用idx_order_no的成本：84.21

很显然，使用idx_expire_time的成本最低，所以当然选择idx_expire_time来执行查询。来和Tracer中的比较一下：



```
"index": "idx_order_no",
"ranges": [
  "DD00_10S <= order_no <= DD00_10S",
  "DD00_6S <= order_no <= DD00_6S",
  "DD00_9S <= order_no <= DD00_9S"
],
"index_dives_for_eq_ranges": true,
"rowid_ordered": false,
"using_mrr": false,
"index_only": false,
"rows": 58,
"cost": 72.61,
"chosen": true
```

```
"index": "idx_expire_time",
"ranges": [
  "0x99a92d271c < expire_time <= 0x99a92d28c9"
],
"index_dives_for_eq_ranges": true,
"rowid_ordered": false,
"using_mrr": false,
"index_only": false,
"rows": 39,
"cost": 47.81,
"chosen": true
```

嗯？除了全表扫描，其他的怎么好像有点对不上呢？

请注意：

1、在MySQL的实际计算中，在和全文扫描比较成本时，使用索引的成本会去除读取并检测回表后聚簇索引记录的成本，也就是说，我们通过MySQL看到的成本将会是：idx_expire_time为47.81(55.61-7.8)，idx_order_no为72.61(84.21-11.6)。但是MySQL比较完成本后，会再计算一次使用索引的成本，此时就会加上前面去除的成本，也就是我们计算出来的值。

```
"best_access_path": {
  "considered_access_paths": [
    {
      "rows_to_scan": 39,
      "access_type": "range",
      "range_details": {
        "used_index": "idx_expire_time"
      },
      "resulting_rows": 39,
      "cost": 55.61,
      "chosen": true
    }
  ]
},
```

2、MySQL的源码中对成本的计算实际要更复杂，但是基本思想和算法是没错的。

基于索引统计数据的成本

index dive

有时候使用索引执行查询时会有许多单点区间，比如使用IN语句就很容易产生非常多的单点区间，比如下边这个查询（下边查询语句中的...表示还有很多参数）：

```
SELECT * FROM order_exp WHERE order_no IN ('aa1', 'aa2', 'aa3', ... , 'zzz');
```

很显然，这个查询可能使用到的索引就是idx_order_no，由于这个索引并不是唯一二级索引，所以并不能确定一个单点区间对应的二级索引记录的条数有多少，需要我们去计算。就是先获取索引对应的B+树的区间最左记录和区间最右记录，然后再计算这两条记录之间有多少记录（记录条数少的时候可以做到精确计算，多的时候只能估算）。MySQL把这种通过直接访问索引对应的B+树来计算某个范围区间对应的索引记录条数的方式称之为index dive。

有零星几个单点区间的话，使用index dive的方式去计算这些单点区间对应的记录数也不是什么问题，如果IN语句里20000个参数怎么办？

这就意味着MySQL的查询优化器为了计算这些单点区间对应的索引记录条数，要进行20000次index dive操作，这性能损耗就很大，搞不好计算这些单点区间对应的索引记录条数的成本比直接全表扫描的成本都大了。MySQL考虑到了这种情况，所以提供了一个系统变量eq_range_index_dive_limit，我们看一下在MySQL 5.7.21中这个系统变量的默认值：

```
show variables like '%dive%';
```

```
mysql> show variables like '%dive%';
+------------------------------+-------+
| Variable_name                | Value |
+------------------------------+-------+
| eq_range_index_dive_limit | 200   |
+------------------------------+-------+
1 row in set (0.00 sec)
```

也就是说如果我们的IN语句中的参数个数小于200个的话，将使用index dive的方式计算各个单点区间对应的记录条数，如果大于或等于200个的话，可就不能使用index dive了，要使用所谓的索引统计数据来进行估算。怎么个估算法？

像会为每个表维护一份统计数据一样，MySQL也会为表中的每一个索引维护一份统计数据，查看某个表中索引的统计数据可以使用SHOW INDEX FROM 表名的语法，比如我们查看一下order_exp的各个索引的统计数据可以这么写：

```
show index from order_exp;
```

```
mysql> show index from order_exp;
+-----------+------------+------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+
| Table     | Non_unique | Key_name         | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment |
| Index_comment |
+-----------+------------+------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+
| order_exp |          0 | PRIMARY          |            1 | id          | A         |        9460 |     NULL | NULL   |      | BTREE      |         |
|               |
| order_exp |          0 | u_idx_day_status |            1 | insert_time | A         |         915 |     NULL | NULL   |      | BTREE      |         |
|               |
| order_exp |          0 | u_idx_day_status |            2 | order_status | A         |         915 |     NULL | NULL   |      | BTREE      |         |
|               |
| order_exp |          0 | u_idx_day_status |            3 | expire_time | A         |        9774 |     NULL | NULL   |      | BTREE      |         |
|               |
| order_exp |          1 | idx_order_no     |            1 | order_no    | A         |        9432 |     NULL | NULL   |      | BTREE      |         |
|               |
| order_exp |          1 | idx_expire_time  |            1 | expire_time | A         |        9802 |     NULL | NULL   |      | BTREE      |         |
|               |
+-----------+------------+------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+
+---------------+
6 rows in set (0.00 sec)
```

属性名　描述

Table　索引所属表的名称。

Non_unique　　索引列的值是否是唯一的，聚簇索引和唯一二级索引的该列值为0，普通二级索引该列值为1。

Key_name　　索引的名称。

Seq_in_index　　索引列在索引中的位置，从1开始计数。比如对于联合索引u_idx_day_status，来说，`insert_time`, `order_status`, `expire_time`对应的位置分别是1、2、3。

Column_name　　索引列的名称。

Collation　　索引列中的值是按照何种排序方式存放的，值为A时代表升序存放，为NULL时代表降序存放。

Cardinality　　索引列中不重复值的数量。后边我们会重点看这个属性的。

Sub_part　　对于存储字符串或者字节串的列来说，有时候我们只想对这些串的前n个字符或字节建立索引，这个属性表示的就是那个n值。如果对完整的列建立索引的话，该属性的值就是NULL。

Packed　索引列如何被压缩，NULL值表示未被压缩。这个属性我们暂时不了解，可以先忽略掉。

Null　　该索引列是否允许存储NULL值。

Index_type　　使用索引的类型，我们最常见的就是BTREE，其实也就是B+树索引。

Comment 索引列注释信息。

Index_comment　索引注释信息。

Cardinality属性，Cardinality直译过来就是基数的意思，表示索引列中不重复值的个数。比如对于一个一万行记录的表来说，某个索引列的Cardinality属性是10000，那意味着该列中没有重复的值，如果Cardinality属性是1的话，就意味着该列的值全部是重复的。不过需要注意的是，对于InnoDB存储引擎来说，使用SHOW INDEX语句展示出来的某个索引列的Cardinality属性是一个估计值，并不是精确的。

前边说道，当IN语句中的参数个数大于或等于系统变量eq_range_index_dive_limit的值的话，就不会使用index dive的方式计算各个单点区间对应的索引记录条数，而是使用索引统计数据，这里所指的索引统计数据指的是这两个值：

使用SHOW TABLE STATUS展示出的Rows值，也就是一个表中有多少条记录。

使用SHOW INDEX语句展示出的Cardinality属性。

结合上一个Rows统计数据，我们可以针对索引列，计算出平均一个值重复多少次。

一个值的重复次数 ≈ Rows ÷ Cardinality

以order_exp表的idx_order_no索引为例，它的Rows值是10354，它对应的Cardinality值是10225，所以我们可以计算order_no列平均单个值的重复次数就是：

10354÷ 10225≈ 1.0126（条）

此时再看上边那条查询语句：

SELECT * FROM order_exp WHERE order_no IN ('aa1', 'aa2', 'aa3', ... , 'zzz');

假设IN语句中有20000个参数的话，就直接使用统计数据来估算这些参数需要单点区间对应的记录条数了，每个参数大约对应1.012条记录，所以总共需要回表的记录数就是：

20000 x1.0126= 20252

使用统计数据来计算单点区间对应的索引记录条数比index dive的方式简单，但是它的致命弱点就是：不精确！。使用统计数据算出来的查询成本与实际所需的成本可能相差非常大。

大家需要注意一下，在MySQL 5.7.3以及之前的版本中，eq_range_index_dive_limit的默认值为10，之后的版本默认值为200。所以如果大家采用的是5.7.3以及之前的版本的话，很容易采用索引统计数据而不是index dive的方式来计算查询成本。当你的查询中使用到了IN查询，但是却实际没有用到索引，就应该考虑一下是不是由于eq_range_index_dive_limit 值太小导致的。

EXPLAIN输出成本

前面我们已经对MySQL查询优化器如何计算成本有了比较深刻的了解。如何通过EXPLAIN语句查看成本呢？MySQL已经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，里边包含该计划花费的成本，比如这样：

```
explain format=json SELECT * FROM order_exp WHERE order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S') AND  expire_time> '2021-03-22 18:28:28' AND expire_time<= '2021-03-22 18:35:09' AND insert_time> expire_time AND order_note LIKE '%7排1%' AND  order_status = 0\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": "55.61" # 整个查询的执行成本预计为55.61
    },
    "table": {
      "table_name": "order_exp",
      "access_type": "range",
      "possible_keys": [
```

```
        "idx_order_no",
        "idx_expire_time"
      ],
      "key": "idx_expire_time",
      "used_key_parts": [
        "expire_time"
      ],
      "key_length": "5",
      "rows_examined_per_scan": 39,
      "rows_produced_per_join": 0,
      "filtered": "0.13",
      "index_condition": "((`mysqladv`.`order_exp`.`expire_time` > '2021-03-22
18:28:28') and (`mysqladv`.`order_exp`.`expire_time` <= '2021-03-22 18:35:09'))",
      "cost_info": {
        "read_cost": "55.60",
        "eval_cost": "0.01",
        "prefix_cost": "55.61",    #单独查询表的成本，也就是：read_cost + eval_cost
        "data_read_per_join": "24"  #和连接查询相关的数据量，单位字节，这里无用
      },
      "used_columns": [
        "id",
        "order_no",
        "order_note",
        "insert_time",
        "expire_duration",
        "expire_time",
        "order_status"
      ],
      "attached_condition": "((`mysqladv`.`order_exp`.`order_status` = 0) and
(`mysqladv`.`order_exp`.`order_no` in ('DD00_6S','DD00_9S','DD00_10S')) and
(`mysqladv`.`order_exp`.`insert_time` > `mysqladv`.`order_exp`.`expire_time`) and
(`mysqladv`.`order_exp`.`order_note` like '%7排1%'))"
    }
  }
}
1 row in set, 1 warning (0.00 sec)
```


连接查询的成本

连接查询的概念

下面的课程中可能牵涉连接查询中诸如“驱动表”之类的概念请参考第四期MySQL的《5、Mysql索引优化实战二》。

Condition filtering介绍

连接查询至少是要有两个表的，课程的讲述中可能使用order_exp表的派生表s1、s2和order_exp2。

我们前边说过，MySQL中连接查询采用的是嵌套循环连接算法，驱动表会被访问一次，被驱动表可能会被访问多次，所以对于两表连接查询来说，它的查询成本由下边两个部分构成：

单次查询驱动表的成本

多次查询被驱动表的成本（具体查询多少次取决于对驱动表查询的结果集中有多少条记录）

对驱动表进行查询后得到的记录条数称之为驱动表的**扇出**（英文名：fanout）。很显然驱动表的扇出值越小，对被驱动表的查询次数也就越少，连接查询的总成本也就越低。

当查询优化器想计算整个连接查询所使用的成本时，就需要计算出驱动表的扇出值，有的时候扇出值的计算是很容易的，比如下边这两个查询：

查询一：

```
SELECT * FROM order_exp AS s1 INNER JOIN order_exp2 AS s2;
```

假设使用s1表作为驱动表，很显然对驱动表的单表查询只能使用全表扫描的方式执行，驱动表的扇出值也很明确，那就是驱动表中有多少记录，扇出值就是多少。统计数据中s1表的记录行数是10573，也就是说优化器就直接会把10573当作s1表的扇出值。

查询二：

```
SELECT * FROM order_exp AS s1 INNER JOIN order_exp2 AS s2
WHERE s1.expire_time> '2021-03-22 18:28:28' AND s1.expire_time<= '2021-03-22 18:35:09';
```

仍然假设s1表是驱动表的话，很显然对驱动表的单表查询可以使用idx_expire_time索引执行查询。此时范围区间('2021-03-22 18:28:28', '2021-03-22 18:35:09')中有多少条记录，那么扇出值就是多少。

但是有的时候扇出值的计算就变得很棘手，比方说下边几个查询：

查询三：

```
SELECT * FROM order_exp AS s1 INNER JOIN order_exp2 AS s2 WHERE s1.order_note > 'xyz';
```

本查询和查询一类似，只不过对于驱动表s1多了一个order_note > 'xyz'的搜索条件。查询优化器又不会真正的去执行查询，所以它只能猜这10573记录里有多少条记录满足order_note > 'xyz'条件。

查询四：

SELECT * FROM order_exp AS s1 INNER JOIN order_exp2 AS s2 WHERE s1.expire_time> '2021-03-22 18:28:28' AND s1.expire_time<= '2021-03-22 18:35:09' AND s1.order_note > 'xyz';

本查询和查询二类似，只不过对于驱动表s1也多了一个order_note > 'xyz'的搜索条件。不过因为本查询可以使用idx_expire_time索引，所以只需要从符合二级索引范围区间的记录中猜有多少条记录符合order_note > 'xyz'条件，也就是只需要猜在39条记录中有多少符合order_note > 'xyz'条件。

```
mysql> explain SELECT * FROM order_exp AS s1  WHERE s1.expire_time> '2021-03-22 18:28:28' AND s1.expire_time<= '
+----+-------------+-------+------------+-------+----------------+----------------+---------+------+------+---
| id | select_type | table | partitions | type  | possible_keys  | key            | key_len | ref  | rows | fi
+----+-------------+-------+------------+-------+----------------+----------------+---------+------+------+---
|  1 | SIMPLE      | s1    | NULL       | range | idx_expire_time | idx_expire_time | 5       | NULL |   39 |
+----+-------------+-------+------------+-------+----------------+----------------+---------+------+------+---
```

查询五：

SELECT * FROM order_exp AS s1 INNER JOIN order_exp2 AS s2 WHERE s1.expire_time> '2021-03-22 18:28:28' AND s1.expire_time<= '2021-03-22 18:35:09' AND s1.order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S') AND s1.order_note > 'xyz';

本查询和查询四类似，不过在驱动表s1选取idx_expire_time索引执行查询后，优化器需要从符合二级索引范围区间的记录中猜有多少条记录符合下边两个条件：

order_no IN ('DD00_6S', 'DD00_9S', 'DD00_10S')

order_note > 'xyz'

也就是优化器需要猜在39条记录中有多少符合上述两个条件的。

说了这么多，其实就是想表达在这两种情况下计算驱动表扇出值时需要靠猜：

如果使用的是全表扫描的方式执行的单表查询，那么计算驱动表扇出时需要猜满足搜索条件的记录到底有多少条。

如果使用的是索引执行的单表扫描，那么计算驱动表扇出的时候需要猜满足除使用到对应索引的搜索条件外的其他搜索条件的记录有多少条。

MySQL把这个猜的过程称之为**condition filtering**。当然，这个过程可能会使用到索引，也可能使用到统计数据，也可能就是MySQL单纯的瞎猜，整个评估过程非常复杂，所以我们不去细讲。

在MySQL 5.7之前的版本中，查询优化器在计算驱动表扇出时，如果是使用全表扫描的话，就直接使用表中记录的数量作为扇出值，如果使用索引的话，就直接使用满足范围条件的索引记录条数作为扇出值。

在MySQL 5.7中，MySQL引入了这个condition filtering的功能，就是还要猜一猜剩余的那些搜索条件能把驱动表中的记录再过滤多少条，其实本质上就是为了让成本估算更精确。 我们所说的纯粹瞎猜其实是很不严谨的，MySQL称之为启发式规则。

两表连接的成本分析

连接查询的成本计算公式是这样的：

连接查询总成本 = 单次访问驱动表的成本 + 驱动表扇出数 x 单次访问被驱动表的成本

对于左（外）连接和右（外）连接查询来说，它们的驱动表是固定的，所以想要得到最优的查询方案只需要分别为驱动表和被驱动表选择成本最低的访问方法。

可是对于内连接来说，驱动表和被驱动表的位置是可以互换的，所以需要考虑两个方面的问题：

不同的表作为驱动表最终的查询成本可能是不同的，也就是需要考虑最优的表连接顺序。然后分别为驱动表和被驱动表选择成本最低的访问方法。

很显然，计算内连接查询成本的方式更麻烦一些，下边我们就以内连接为例来看看如何计算出最优的连接查询方案。当然在某些情况下，左（外）连接和右（外）连接查询在某些特殊情况下可以被优化为内连接查询。

我们来看看内连接，比如对于下边这个查询来说：

```
SELECT * FROM order_exp AS s1 INNER JOIN order_exp2 AS s2 ON s1.order_no=
s2.order_note  WHERE s1.expire_time> '2021-03-22 18:28:28' AND
s1.expire_time<= '2021-03-22 18:35:09' AND s2.expire_time> '2021-03-22
18:35:09' AND s2.expire_time<= '2021-03-22 18:35:59';
```

可以选择的连接顺序有两种：

s1连接s2，也就是s1作为驱动表，s2作为被驱动表。

s2连接s1，也就是s2作为驱动表，s1作为被驱动表。

查询优化器需要分别考虑这两种情况下的最优查询成本，然后选取那个成本更低的连接顺序以及该连接顺序下各个表的最优访问方法作为最终的查询计划。我们定性的分析一下，不像分析单表查询那样定量的分析了：

使用s1作为驱动表的情况

分析对于驱动表的成本最低的执行方案，首先看一下涉及s1表单表的搜索条件有哪些：

```
s1.expire_time> '2021-03-22 18:28:28' AND s1.expire_time<= '2021-03-22
18:35:09'
```

所以这个查询可能使用到idx_expire_time索引，从全表扫描和使用idx_expire_time这两个方案中选出成本最低的那个，很显然使用idx_expire_time执行查询的成本更低些。

然后分析对于被驱动表的成本最低的执行方案，此时涉及被驱动表s2的搜索条件就是：

1、s2.order_note = 常数（这是因为对驱动表s1结果集中的每一条记录，都需要进行一次被驱动表s2的访问，此时那些涉及两表的条件现在相当于只涉及被驱动表s2了。）

```
2、s2.expire_time> '2021-03-22 18:35:09' AND s2.expire_time<= '2021-03-22
18:35:59'
```

很显然，第一个条件由于order_note没有用到索引，所以并没有什么用，此时访问s2表时可用的方案也是全表扫描和使用idx_expire_time两种，假设使用idx_expire_time的成本更小。

所以此时使用s1作为驱动表时的总成本就是（暂时不考虑使用join buffer对成本的影响）：

使用idx_expire_time访问s1的成本 + s1的扇出 × 使用idx_expire_time访问s2的成本

使用s2作为驱动表的情况

分析对于驱动表的成本最低的执行方案

首先看一下涉及s2表单表的搜索条件有哪些：

s2.expire_time> ’2021-03-22 18:35:09’ AND s2.expire_time<= ’2021-03-22 18:35:59’

所以这个查询可能使用到idx_expire_time索引，从全表扫描和使用idx_expire_time这两个方案中选出成本最低的那个，假设使用idx_expire_time执行查询的成本更低些。

然后分析对于被驱动表的成本最低的执行方案

此时涉及被驱动表s1的搜索条件就是：

1、s1.order_no = 常数

2、s1.expire_time> ’2021-03-22 18:28:28’ AND s1.expire_time<= ’2021-03-22 18:35:09’

这时就很有趣了，使用idx_order_no可以进行ref方式的访问，使用idx_expire_time可以使用range方式的访问。

那么优化器需要从全表扫描、使用idx_order_no、使用idx_expire_time这几个方案里选出一个成本最低的方案。

这里有个问题，因为idx_expire_time的范围区间是确定的，怎么计算使用idx_expire_time的成本我们上边已经说过了，可是在没有真正执行查询前，s1.order_no = 常数中的常数值我们是不知道的，怎么衡量使用idx_order_no执行查询的成本呢？其实很简单，直接使用我们前面说过的索引统计数据就好了（就是索引列平均一个值重复多少次）。一般情况下，ref的访问方式要比range成本更低，这里假设使用idx_order_no进行对s1的访问。

所以此时使用s2作为驱动表时的总成本就是：

使用idx_expire_time访问s2的成本 + s2的扇出 × 使用idx_order_no访问s1的成本

最后优化器会比较这两种方式的最优访问成本，选取那个成本更低的连接顺序去真正的执行查询。从上边的计算过程也可以看出来，一般来讲，连接查询成本占大头的其实是驱动表扇出数 x 单次访问被驱动表的成本，所以我们的优化重点其实是下边这两个部分：

尽量减少驱动表的扇出

对被驱动表的访问成本尽量低

这一点对于我们实际书写连接查询语句时十分有用，我们需要尽量在被驱动表的连接列上建立索引，这样就可以使用ref访问方法来降低访问被驱动表的成本了。如果可以，被驱动表的连接列最好是该表的主键或者唯一二级索引列，这样就可以把访问被驱动表的成本降到更低了。

连接查询EXPLAIN输出连接成本

连接查询在输出成本时和单表查询稍有不同，如下：

```
explain format=json SELECT * FROM order_exp AS s1 INNER JOIN order_exp2 AS s2 ON s1.order_no= s2.order_note  WHERE s1.expire_time> '2021-03-22 18:28:28' AND s1.expire_time<= '2021-03-22 18:35:09' AND s2.expire_time> '2021-03-22 18:35:09' AND s2.expire_time<= '2021-03-22 18:35:59'\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": "840.51" # 整个查询的执行成本
    },
    "nested_loop": [    # 几个表之间采用嵌套循环连接算法执行
      {
        "table": {
          "table_name": "s2",    # s2表是驱动表
          "access_type": "range",   # 访问方法为range
          "possible_keys": [
            "idx_expire_time"
          ],
          "key": "idx_expire_time",
          "used_key_parts": [
            "expire_time"
          ],
          "key_length": "5",
          "rows_examined_per_scan": 321, # 查询s2表大致需要扫描321条记录
          "rows_produced_per_join": 321, # 驱动表s2的扇出是321
```

```
        "filtered": "100.00",    # condition filtering代表的百分比
        "index_condition": "((`mysqladv`.`s2`.`expire_time` > '2021-03-22
18:35:09') and (`mysqladv`.`s2`.`expire_time` <= '2021-03-22 18:35:59'))",
        "cost_info": {
          "read_cost": "386.21",
          "eval_cost": "64.20",
          "prefix_cost": "450.41", # 查询s1表总共的成本，read_cost + eval_cost
          "data_read_per_join": "152K" # 读取的数据量
        },
        "used_columns": [
          "id",
          "order_no",
          "order_note",
          "insert_time",
          "expire_duration",
          "expire_time",
          "order_status"
        ]
      }
    },
    {
      "table": {
        "table_name": "s1",  # s1表是被驱动表
        "access_type": "ref",
        "possible_keys": [
          "idx_order_no",
          "idx_expire_time"
        ],
        "key": "idx_order_no",
        "used_key_parts": [
          "order_no"
        ],
        "key_length": "152",
```

```
          "ref": [
            "mysqladv.s2.order_note"
          ],
          "rows_examined_per_scan": 1, # 查询一次s1表大致需要扫描1条记录
          "rows_produced_per_join": 16, # 被驱动表s2的扇出是16（由于没有多余的表进行连接，所以这个值无用）
          "filtered": "4.94", # condition filtering代表的百分比
          "index_condition": "(`mysqladv`.`s1`.`order_no` = `mysqladv`.`s2`.`order_note`)",
          "cost_info": {
            "read_cost": "325.08",
            "eval_cost": "3.21",
            "prefix_cost": "840.51", # 单次查询s2、多次查询s1表总共的成本
            "data_read_per_join": "7K"
          },
          "used_columns": [
            "id",
            "order_no",
            "order_note",
            "insert_time",
            "expire_duration",
            "expire_time",
            "order_status"
          ],
          "attached_condition": "((`mysqladv`.`s1`.`expire_time` > '2021-03-22 18:28:28') and (`mysqladv`.`s1`.`expire_time` <= '2021-03-22 18:35:09'))"
        }
      }
    ]
  }
}
1 row in set, 1 warning (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，s2表的"cost_info"中prefix_cost就是单独查询s2表的成本。

对于s1表的"cost_info"中，由于s1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，而s1表中的prefix_cost的值代表的是整个连接查询预计的成本。

看完了上面的执行计划的输出，可能大家有疑惑，驱动表S2的查询成本为450.41，总查询成本为840.51，也就是说对被驱动表S1的查询成本也就是390左右，看起来用S1做驱动表好像更省一点。真的这样吗？我们把SQL语句改造一下，将INNER JOIN 替换为STRAIGHT_JOIN：

```
explain format=json SELECT * FROM order_exp AS s1 STRAIGHT_JOIN order_exp2 AS s2 ON s1.order_no= s2.order_note  WHERE s1.expire_time> '2021-03-22 18:28:28' AND s1.expire_time<= '2021-03-22 18:35:09' AND s2.expire_time> '2021-03-22 18:35:09' AND s2.expire_time<= '2021-03-22 18:35:59'\G
```

大家可以自行看看所需要的成本是多少，并自行解释一下原因。

多表连接的成本分析

首先要考虑一下多表连接时可能产生出多少种连接顺序：

对于两表连接，比如表A和表B连接

只有 AB、BA这两种连接顺序。其实相当于2 × 1 = 2种连接顺序。

对于三表连接，比如表A、表B、表C进行连接

有ABC、ACB、BAC、BCA、CAB、CBA这么6种连接顺序。其实相当于3 × 2 × 1 = 6种连接顺序。

对于四表连接的话，则会有4 × 3 × 2 × 1 = 24种连接顺序。

对于n表连接的话，则有 n × (n-1) × (n-2) × ••• × 1种连接顺序，就是n的阶乘种连接顺序，也就是n!。

有n个表进行连接，MySQL查询优化器要每一种连接顺序的成本都计算一遍么？那就有n!种连接顺序。其实真的是要都算一遍，不过MySQL用了很多办法减少计算非常多种连接顺序的成本的方法：

提前结束某种顺序的成本评估

MySQL在计算各种链接顺序的成本之前，会维护一个全局的变量，这个变量表示当前最小的连接查询成本。如果在分析某个连接顺序的成本时，该成本已经超过当前最小的连接查询成本，那就压根儿不对该连接顺序继续往下分析了。比方说A、B、C三个表进行连接，已经得到连接顺序ABC是当前的最小连接成本，比方说10.0，在计算连接顺序BCA时，发现B和C的连接成本就已经大于10.0时，就不再继续往后分析BCA这个连接顺序的成本了。

系统变量optimizer_search_depth

为了防止无穷无尽的分析各种连接顺序的成本，MySQL提出了optimizer_search_depth系统变量，如果连接表的个数小于该值，那么就继续穷举分析每一种连接顺序的成本，否则只对与optimizer_search_depth值相同数量的表进行穷举分析。很显然，该值越大，成本分析的越精确，越容易得到好的执行计划，但是消耗的时间也就越长，否则得到不是很好的执行计划，但可以省掉很多分析连接成本的时间。

根据某些规则压根儿就不考虑某些连接顺序

即使是有上边两条规则的限制，但是分析多个表不同连接顺序成本花费的时间还是会很长，所以MySQL干脆提出了一些所谓的启发式规则（就是根据以往经验指定的一些规则），凡是不满足这些规则的连接顺序压根儿就不分析，这样可以极大的减少需要分析的连接顺序的数量，但是也可能造成错失最优的执行计划。他们提供了一个系统变量optimizer_prune_level来控制到底是不是用这些启发式规则。

不过按照《阿里最新Java编程规范泰山版》中《(二) 索引规约》中的说法：

2. 【强制】超过三个表禁止 join。需要 join 的字段，数据类型保持绝对一致；多表关联查询时，保证被关联的字段需要有索引。
   说明：即使双表 join 也要注意表索引、SQL 性能。

当出现超过三个表的join时，就应该考虑改写SQL语句了，因为从我们上面的多表关联成本分析可以知道，就算是不考虑多表关联时需要查询的巨大记录条数，就算是几个表的关联成本计算也是个很耗费时间的过程。

调节成本常数

我们前边已经介绍了两个成本常数：

读取一个页面花费的成本默认是1.0

检测一条记录是否符合搜索条件的成本默认是0.2

其实除了这两个成本常数，MySQL还支持很多，它们被存储到了MySQL数据库的两个表中：

```
SHOW TABLES FROM mysql LIKE '%cost%';
```

```
mysql> SHOW TABLES FROM mysql LIKE '%cost%';
+--------------------------+
| Tables_in_mysql (%cost%) |
+--------------------------+
| engine_cost              |
| server_cost              |
+--------------------------+
2 rows in set (0.00 sec)
```

因为一条语句的执行其实是分为两层的：server层、存储引擎层。

在server层进行连接管理、查询缓存、语法解析、查询优化等操作，在存储引擎层执行具体的数据存取操作。也就是说一条语句在server层中执行的成本是和它操作的表使用的存储引擎是没关系的，所以关于这些操作对应的成本常数就存储在了server_cost表中，而依赖于存储引擎的一些操作对应的成本常数就存储在了engine_cost表中。

mysql.server_cost表

server_cost表中在server层进行的一些操作对应的成本常数，具体内容如下：

SELECT * FROM mysql.server_cost;

```
mysql> SELECT * FROM mysql.server_cost;
+------------------------------+------------+---------------------+---------+
| cost_name                    | cost_value | last_update         | comment |
+------------------------------+------------+---------------------+---------+
| disk_temptable_create_cost   |       NULL | 2021-03-19 16:19:15 | NULL    |
| disk_temptable_row_cost      |       NULL | 2021-03-19 16:19:15 | NULL    |
| key_compare_cost             |       NULL | 2021-03-19 16:19:15 | NULL    |
| memory_temptable_create_cost |       NULL | 2021-03-19 16:19:15 | NULL    |
| memory_temptable_row_cost    |       NULL | 2021-03-19 16:19:15 | NULL    |
| row_evaluate_cost            |       NULL | 2021-03-19 16:19:15 | NULL    |
+------------------------------+------------+---------------------+---------+
6 rows in set (0.00 sec)
```

我们先看一下server_cost各个列都分别是什么意思：

cost_name

表示成本常数的名称。

cost_value

表示成本常数对应的值。如果该列的值为NULL的话，意味着对应的成本常数会采用默认值。

last_update

表示最后更新记录的时间。

comment

注释。

从server_cost中的内容可以看出来，目前在server层的一些操作对应的成本常数有以下几种：

disk_temptable_create_cost 默认值40.0 创建基于磁盘的临时表的成本，如果增大这个值的话会让优化器尽量少的创建基于磁盘的临时表。

disk_temptable_row_cost 默认值1.0 向基于磁盘的临时表写入或读取一条记录的成本，如果增大这个值的话会让优化器尽量少的创建基于磁盘的临时表。

key_compare_cost 0.1 两条记录做比较操作的成本，多用在排序操作上，如果增大这个值的话会提升filesort的成本，让优化器可能更倾向于使用索引完成排序而不是filesort。

memory_temptable_create_cost 默认值2.0 创建基于内存的临时表的成本，如果增大这个值的话会让优化器尽量少的创建基于内存的临时表。

memory_temptable_row_cost 默认值0.2 向基于内存的临时表写入或读取一条记录的成本，如果增大这个值的话会让优化器尽量少的创建基于内存的临时表。

row_evaluate_cost 默认值0.2 这个就是我们之前一直使用的检测一条记录是否符合搜索条件的成本，增大这个值可能让优化器更倾向于使用索引而不是直接全表扫描。

MySQL在执行诸如DISTINCT查询、分组查询、Union查询以及某些特殊条件下的排序查询都可能在内部先创建一个临时表，使用这个临时表来辅助完成查询（比如对于DISTINCT查询可以建一个带有UNIQUE索引的临时表，直接把需要去重的记录插入到这个临时表中，插入完成之后的记录就是结果集了）。在数据量大的情况下可能创建基于磁盘的临时表，也就是为该临时表使用MyISAM、InnoDB等存储引擎，在数据量不大时可能创建基于内存的临时表，也就是使用Memory存储引擎。大家可以看到，创建临时表和对这个临时表进行写入和读取的操作代价还是很高的就行了。

这些成本常数在server_cost中的初始值都是NULL，意味着优化器会使用它们的默认值来计算某个操作的成本，如果我们想修改某个成本常数的值的话，需要做两个步骤：

对我们感兴趣的成本常数做update更新操作，然后使用下边语句即可：

```
FLUSH OPTIMIZER_COSTS;
```

当然，在你修改完某个成本常数后想把它们再改回默认值的话，可以直接把cost_value的值设置为NULL，再使用FLUSH OPTIMIZER_COSTS语句让系统重新加载。

mysql.engine_cost表

engine_cost表表中在存储引擎层进行的一些操作对应的成本常数，具体内容如下：

```
SELECT * FROM mysql.engine_cost;
```

```
mysql> SELECT * FROM mysql.engine_cost;
+-------------+-------------+------------------------+------------+---------------------+---------+
| engine_name | device_type | cost_name              | cost_value | last_update         | comment |
+-------------+-------------+------------------------+------------+---------------------+---------+
| default     |           0 | io_block_read_cost     |       NULL | 2021-03-19 16:19:15 | NULL    |
| default     |           0 | memory_block_read_cost |       NULL | 2021-03-19 16:19:15 | NULL    |
+-------------+-------------+------------------------+------------+---------------------+---------+
2 rows in set (0.00 sec)
```

与server_cost相比，engine_cost多了两个列：

engine_name列

指成本常数适用的存储引擎名称。如果该值为default，意味着对应的成本常数适用于所有的存储引擎。

device_type列

指存储引擎使用的设备类型，这主要是为了区分常规的机械硬盘和固态硬盘，不过在MySQL 5.7.X这个版本中并没有对机械硬盘的成本和固态硬盘的成本作区分，所以该值默认是0。

我们从engine_cost表中的内容可以看出来，目前支持的存储引擎成本常数只有两个：

io_block_read_cost　　默认值1.0　　从磁盘上读取一个块对应的成本。请注意我使用的是块，而不是页这个词。对于InnoDB存储引擎来说，一个页就是一个块，不过对于MyISAM存储引擎来说，默认是以4096字节作为一个块的。增大这个值会加重I/O成本，可能让优化器更倾向于选择使用索引执行查询而不是执行全表扫描。

memory_block_read_cost　默认值1.0　　与上一个参数类似，只不过衡量的是从内存中读取一个块对应的成本。

怎么从内存中和从磁盘上读取一个块的默认成本是一样的？这主要是因为在MySQL目前的实现中，并不能准确预测某个查询需要访问的块中有哪些块已经加载到内存中，有哪些块还停留在磁盘上，所以MySQL简单的认为不管这个块有没有加载到内存中，使用的成本都是1.0。

与更新server_cost表中的记录一样，我们也可以通过更新engine_cost表中的记录来更改关于存储引擎的成本常数，做法一样。

InnoDB中的统计数据

我们前边学习查询成本的时候经常用到一些统计数据，比如通过SHOW TABLE STATUS可以看到关于表的统计数据，通过SHOW INDEX可以看到关于索引的统计数据，那么这些统计数据是怎么来的呢？它们是以什么方式收集的呢？

统计数据存储方式

InnoDB提供了两种存储统计数据的方式：

永久性的统计数据，这种统计数据存储在磁盘上，也就是服务器重启之后这些统计数据还在。

非永久性的统计数据，这种统计数据存储在内存中，当服务器关闭时这些这些统计数据就都被清除掉了，等到服务器重启之后，在某些适当的场景下才会重新收集这些统计数据。

MySQL给我们提供了系统变量innodb_stats_persistent来控制到底采用哪种方式去存储统计数据。在MySQL 5.6.6之前，innodb_stats_persistent的值默认是OFF，也就是说InnoDB的统计数据默认是存储到内存的，之后的版本中innodb_stats_persistent的值默认是ON，也就是统计数据默认被存储到磁盘中。

SHOW VARIABLES LIKE 'innodb_stats_persistent';

```
mysql> show variables like 'innodb_stats_persistent';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| innodb_stats_persistent | ON    |
+--------------------------+-------+
1 row in set (0.01 sec)
```

不过最近的MySQL版本都基本不用基于内存的非永久性统计数据了，所以我们也就不深入研究。

不过InnoDB默认是以表为单位来收集和存储统计数据的，也就是说我们可以把某些表的统计数据（以及该表的索引统计数据）存储在磁盘上，把另一些表的统计数据存储在内存中。怎么做到的呢？我们可以在创建和修改表的时候通过指定STATS_PERSISTENT属性来指明该表的统计数据存储方式：

CREATE TABLE 表名 (...) Engine=InnoDB, STATS_PERSISTENT = (1|0);

ALTER TABLE 表名 Engine=InnoDB, STATS_PERSISTENT = (1|0);

当STATS_PERSISTENT=1时，表明我们想把该表的统计数据永久的存储到磁盘上，当STATS_PERSISTENT=0时，表明我们想把该表的统计数据临时的存储到内存中。如果我们在创建表时未指定STATS_PERSISTENT属性，那默认采用系统变量innodb_stats_persistent的值作为该属性的值。

基于磁盘的永久性统计数据

当我们选择把某个表以及该表索引的统计数据存放到磁盘上时，实际上是把这些统计数据存储到了两个表里：

SHOW TABLES FROM mysql LIKE 'innodb%';

```
mysql> SHOW TABLES FROM mysql LIKE 'innodb%';
+---------------------------+
| Tables_in_mysql (innodb%) |
+---------------------------+
| innodb_index_stats        |
| innodb_table_stats        |
+---------------------------+
2 rows in set (0.00 sec)
```

可以看到，这两个表都位于mysql系统数据库下边，其中：

innodb_table_stats存储了关于表的统计数据，每一条记录对应着一个表的统计数据。

innodb_index_stats存储了关于索引的统计数据，每一条记录对应着一个索引的一个统计项的统计数据。

innodb_table_stats

直接看一下这个innodb_table_stats表中的各个列都是干嘛的：

```
mysql> desc mysql.innodb_table_stats;
+-------------------------+------------
| Field                   | Type
+-------------------------+------------
| database_name           | varchar(64)
| table_name              | varchar(199
| last_update             | timestamp
| n_rows                  | bigint(20)
| clustered_index_size    | bigint(20)
| sum_of_other_index_sizes | bigint(20)
+-------------------------+------------
6 rows in set (0.00 sec)
```

database_name　数据库名

table_name　表名

last_update　本条记录最后更新时间

n_rows　表中记录的条数

clustered_index_size　表的聚簇索引占用的页面数量

sum_of_other_index_sizes　表的其他索引占用的页面数量

我们直接看一下这个表里的内容：

SELECT * FROM mysql.innodb_table_stats;

```
mysql> SELECT * FROM mysql.innodb_table_stats;
+---------------+---------------+---------------------+--------+----------------------+-------------------------+
| database_name | table_name    | last_update         | n_rows | clustered_index_size | sum_of_other_index_sizes |
+---------------+---------------+---------------------+--------+----------------------+-------------------------+
| mysql         | gtid_executed | 2021-03-19 16:19:15 |      0 |                    1 |                       0 |
| mysqladv      | e1            | 2021-03-31 16:34:58 |      3 |                    1 |                       0 |
| mysqladv      | e2            | 2021-03-31 16:35:08 |      3 |                    1 |                       0 |
| mysqladv      | order_exp     | 2021-03-31 15:31:28 |  10248 |                   97 |                      81 |
| mysqladv      | s1            | 2021-03-26 21:42:50 |  10573 |                  161 |                      50 |
| mysqladv      | s2            | 2021-03-26 18:11:20 |  10315 |                   97 |                      69 |
| mysqladv      | score         | 2021-03-31 16:50:57 |      4 |                    1 |                       0 |
| mysqladv      | student       | 2021-03-31 16:48:12 |      4 |                    1 |                       0 |
| mysqladv      | teacher       | 2021-03-30 20:11:12 |      5 |                    1 |                       1 |
| mysqladv      | test          | 2021-03-22 14:31:58 |      1 |                    1 |                       0 |
| mysqladv      | undo_test     | 2021-03-29 21:53:16 |      0 |                    1 |                       1 |
| sys           | sys_config    | 2021-03-19 16:19:16 |      6 |                    1 |                       0 |
+---------------+---------------+---------------------+--------+----------------------+-------------------------+
12 rows in set (0.00 sec)
```

几个重要统计信息项的值如下：

n_rows的值是10350，表明order_exp表中大约有10350条记录，注意这个数据是估计值。

clustered_index_size的值是97，表明order_exp表的聚簇索引占用97个页面，这个值是也是一个估计值。

sum_of_other_index_sizes的值是81，表明order_exp表的其他索引一共占用81个页面，这个值是也是一个估计值。

n_rows统计项的收集

InnoDB统计一个表中有多少行记录是这样的：

按照一定算法（并不是纯粹随机的）选取几个叶子节点页面，计算每个页面中主键值记录数量，然后计算平均一个页面中主键值的记录数量乘以全部叶子节点的数量就算是该表的n_rows值。

可以看出来这个n_rows值精确与否取决于统计时采样的页面数量，MySQL用名为innodb_stats_persistent_sample_pages的系统变量来控制使用永久性的统计数据时，计算统计数据时采样的页面数量。该值设置的越大，统计出的n_rows值越精确，但是统计耗时也就最久；该值设置的越小，统计出的n_rows值越不精确，但是统计耗时特别少。所以在实际使用是需要我们去权衡利弊，该系统变量的默认值是20。

InnoDB默认是以表为单位来收集和存储统计数据的，我们也可以单独设置某个表的采样页面的数量，设置方式就是在创建或修改表的时候通过指定STATS_SAMPLE_PAGES属性来指明该表的统计数据存储方式：

CREATE TABLE 表名 (...) Engine=InnoDB, STATS_SAMPLE_PAGES = 具体的采样页面数量;

ALTER TABLE 表名 Engine=InnoDB, STATS_SAMPLE_PAGES = 具体的采样页面数量;

如果我们在创建表的语句中并没有指定STATS_SAMPLE_PAGES属性的话，将默认使用系统变量innodb_stats_persistent_sample_pages的值作为该属性的值。

clustered_index_size和sum_of_other_index_sizes统计项的收集牵涉到很具体的InnoDB表空间的知识和存储页面数据的细节，我们就不深入讲解了。

innodb_index_stats

直接看一下这个innodb_index_stats表中的各个列都是干嘛的：

desc mysql.innodb_index_stats;

```
mysql> desc mysql.innodb_index_stats;
+------------------+---------------------+------+-----+-------------------+-----------------------------+
| Field            | Type                | Null | Key | Default           | Extra                       |
+------------------+---------------------+------+-----+-------------------+-----------------------------+
| database_name    | varchar(64)         | NO   | PRI | NULL              |                             |
| table_name       | varchar(199)        | NO   | PRI | NULL              |                             |
| index_name       | varchar(64)         | NO   | PRI | NULL              |                             |
| last_update      | timestamp           | NO   |     | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
| stat_name        | varchar(64)         | NO   | PRI | NULL              |                             |
| stat_value       | bigint(20) unsigned | NO   |     | NULL              |                             |
| sample_size      | bigint(20) unsigned | YES  |     | NULL              |                             |
| stat_description | varchar(1024)       | NO   |     | NULL              |                             |
+------------------+---------------------+------+-----+-------------------+-----------------------------+
8 rows in set (0.00 sec)
```

字段名　描述

database_name　　数据库名

table_name　　表名

index_name　　索引名

last_update　　本条记录最后更新时间

stat_name　　统计项的名称

stat_value　　对应的统计项的值

sample_size　　为生成统计数据而采样的页面数量

stat_description　　对应的统计项的描述

innodb_index_stats表的每条记录代表着一个索引的一个统计项。可能这会大家有些懵逼这个统计项到底指什么，别着急，我们直接看一下关于order_exp表的索引统计数据都有些什么：

mysql> SELECT * FROM mysql.innodb_index_stats WHERE table_name = 'order_exp';

```
mysql> SELECT * FROM mysql.innodb_index_stats WHERE table_name = 'order_exp';
+---------------+------------+-----------------+---------------------+--------------+------------+-------------+---------------------------------+
| database_name | table_name | index_name      | last_update         | stat_name    | stat_value | sample_size | stat_description                |
+---------------+------------+-----------------+---------------------+--------------+------------+-------------+---------------------------------+
| mysqladv      | order_exp  | PRIMARY         | 2021-03-25 23:32:02 | n_diff_pfx01 |       9460 |          20 | id                              |
| mysqladv      | order_exp  | PRIMARY         | 2021-03-25 23:32:02 | n_leaf_pages |         79 |        NULL | Number of leaf pages in the index |
| mysqladv      | order_exp  | PRIMARY         | 2021-03-25 23:32:02 | size         |         97 |        NULL | Number of pages in the index    |
| mysqladv      | order_exp  | idx_expire_time | 2021-03-31 15:31:28 | n_diff_pfx01 |       9802 |          13 | expire_time                     |
| mysqladv      | order_exp  | idx_expire_time | 2021-03-31 15:31:28 | n_diff_pfx02 |      10562 |          13 | expire_time,id                  |
| mysqladv      | order_exp  | idx_expire_time | 2021-03-31 15:31:28 | n_leaf_pages |         13 |        NULL | Number of leaf pages in the index |
| mysqladv      | order_exp  | idx_expire_time | 2021-03-31 15:31:28 | size         |         14 |        NULL | Number of pages in the index    |
| mysqladv      | order_exp  | idx_order_no    | 2021-03-25 23:32:02 | n_diff_pfx01 |       9432 |          24 | order_no                        |
| mysqladv      | order_exp  | idx_order_no    | 2021-03-25 23:32:02 | n_diff_pfx02 |       9774 |          24 | order_no,id                     |
| mysqladv      | order_exp  | idx_order_no    | 2021-03-25 23:32:02 | n_leaf_pages |         24 |        NULL | Number of leaf pages in the index |
| mysqladv      | order_exp  | idx_order_no    | 2021-03-25 23:32:02 | size         |         25 |        NULL | Number of pages in the index    |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | n_diff_pfx01 |        915 |          27 | insert_time                     |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | n_diff_pfx02 |        915 |          27 | insert_time,order_status        |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | n_diff_pfx03 |       9774 |          27 | insert_time,order_status,expire_time |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | n_leaf_pages |         27 |        NULL | Number of leaf pages in the index |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | size         |         28 |        NULL | Number of pages in the index    |
+---------------+------------+-----------------+---------------------+--------------+------------+-------------+---------------------------------+
16 rows in set (0.00 sec)
```

先查看index_name列，这个列说明该记录是哪个索引的统计信息，从结果中我们可以看出来，PRIMARY索引（也就是主键）占了3条记录，idx_expire_time索引占了6条记录。针对index_name列相同的记录，stat_name表示针对该索引的统计项名称，stat_value展示的是该索引在该统计项上的值，stat_description指的是来描述该统计项的含义的。我们来具体看一下一个索引都有哪些统计项：

n_leaf_pages：表示该索引的叶子节点占用多少页面。

size：表示该索引共占用多少页面。

n_diff_pfxNN：表示对应的索引列不重复的值有多少。其中的NN长得有点儿怪呀，啥意思呢？

其实NN可以被替换为01、02、03... 这样的数字。比如对于u_idx_day_status来说：

n_diff_pfx01表示的是统计insert_time这单单一个列不重复的值有多少。

n_diff_pfx02表示的是统计insert_time,order_status这两个列组合起来不重复的值有多少。

n_diff_pfx03表示的是统计insert_time,order_status,expire_time这三个列组合起来不重复的值有多少。

n_diff_pfx04表示的是统计key_pare1、key_pare2、expire_time、id这四个列组合起来不重复的值有多少。

对于普通的二级索引，并不能保证它的索引列值是唯一的，比如对于idx_order_no来说，key1列就可能有很多值重复的记录。此时只有在索引列上加上主键值才可以区分两条索引列值都一样的二级索引记录。

对于主键和唯一二级索引则没有这个问题，它们本身就可以保证索引列值的不重复，所以也不需要再统计一遍在索引列后加上主键值的不重复值有多少。比如u_idx_day_statu和idx_order_no。

在计算某些索引列中包含多少不重复值时，需要对一些叶子节点页面进行采样，sample_size列就表明了采样的页面数量是多少。

对于有多个列的联合索引来说，采样的页面数量是：

innodb_stats_persistent_sample_pages × 索引列的个数。

```
mysql> show variables like 'innodb_stats_transient_sample_pages';
+---------------------------------------+-------+
| Variable_name                         | Value |
+---------------------------------------+-------+
| innodb_stats_transient_sample_pages | 8     |
+---------------------------------------+-------+
1 row in set (0.00 sec)
```

当需要采样的页面数量大于该索引的叶子节点数量的话，就直接采用全表扫描来统计索引列的不重复值数量了。所以大家可以在查询结果中看到不同索引对应的size列的值可能是不同的。

定期更新统计数据

随着我们不断的对表进行增删改操作，表中的数据也一直在变化，innodb_table_stats和innodb_index_stats表里的统计数据也在变化。MySQL提供了如下两种更新统计数据的方式：

开启innodb_stats_auto_recalc。

系统变量innodb_stats_auto_recalc决定着服务器是否自动重新计算统计数据，它的默认值是ON，也就是该功能默认是开启的。

```
mysql> show variables like '%innodb_stats_auto_recalc%';
+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| innodb_stats_auto_recalc      | ON    |
+-------------------------------+-------+
```

每个表都维护了一个变量，该变量记录着对该表进行增删改的记录条数，如果发生变动的记录数量超过了表大小的10%，并且自动重新计算统计数据的功能是打开的，那么服务器会重新进行一次统计数据的计算，并且更新innodb_table_stats和innodb_index_stats表。不过自动重新计算统计数据的过程是异步发生的，也就是即使表中变动的记录数超过了10%，自动重新计算统计数据也不会立即发生，可能会延迟几秒才会进行计算。

再一次强调，InnoDB默认是以表为单位来收集和存储统计数据的，我们也可以单独为某个表设置是否自动重新计算统计数的属性，设置方式就是在创建或修改表的时候通过指定STATS_AUTO_RECALC属性来指明该表的统计数据存储方式：

CREATE TABLE 表名 (...) Engine=InnoDB, STATS_AUTO_RECALC = (1|0);

ALTER TABLE 表名 Engine=InnoDB, STATS_AUTO_RECALC = (1|0);

当STATS_AUTO_RECALC=1时，表明我们想让该表自动重新计算统计数据，当STATS_AUTO_RECALC=0时，表明不想让该表自动重新计算统计数据。如果我们在创建表时未指定STATS_AUTO_RECALC属性，那默认采用系统变量innodb_stats_auto_recalc的值作为该属性的值。

手动调用ANALYZE TABLE语句来更新统计信息

如果innodb_stats_auto_recalc系统变量的值为OFF的话，我们也可以手动调用ANALYZE TABLE语句来重新计算统计数据，比如我们可以这样更新关于order_exp表的统计数据：

mysql> ANALYZE TABLE order_exp;

```
mysql> SELECT * FROM mysql.innodb_index_stats WHERE table_name = 'order_exp';
+--------------+------------+-----------------+---------------------+--------------+------------+-------------+-----------------------------------------+
| database_name | table_name | index_name      | last_update         | stat_name    | stat_value | sample_size | stat_description                        |
+--------------+------------+-----------------+---------------------+--------------+------------+-------------+-----------------------------------------+
| mysqladv      | order_exp  | PRIMARY         | 2021-03-25 23:32:02 | n_diff_pfx01 |       9460 |          20 | id                                      |
| mysqladv      | order_exp  | PRIMARY         | 2021-03-25 23:32:02 | n_leaf_pages |         79 |        NULL | Number of leaf pages in the index       |
| mysqladv      | order_exp  | PRIMARY         | 2021-03-25 23:32:02 | size         |         97 |        NULL | Number of pages in the index            |
| mysqladv      | order_exp  | idx_expire_time | 2021-03-31 15:31:28 | n_diff_pfx01 |       9802 |          13 | expire_time                             |
| mysqladv      | order_exp  | idx_expire_time | 2021-03-31 15:31:28 | n_diff_pfx02 |      10562 |          13 | expire_time,id                          |
| mysqladv      | order_exp  | idx_expire_time | 2021-03-31 15:31:28 | n_leaf_pages |         13 |        NULL | Number of leaf pages in the index       |
| mysqladv      | order_exp  | idx_expire_time | 2021-03-31 15:31:28 | size         |         14 |        NULL | Number of pages in the index            |
| mysqladv      | order_exp  | idx_order_no    | 2021-03-25 23:32:02 | n_diff_pfx01 |       9432 |          24 | order_no                                |
| mysqladv      | order_exp  | idx_order_no    | 2021-03-25 23:32:02 | n_diff_pfx02 |       9774 |          24 | order_no,id                             |
| mysqladv      | order_exp  | idx_order_no    | 2021-03-25 23:32:02 | n_leaf_pages |         24 |        NULL | Number of leaf pages in the index       |
| mysqladv      | order_exp  | idx_order_no    | 2021-03-25 23:32:02 | size         |         25 |        NULL | Number of pages in the index            |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | n_diff_pfx01 |        915 |          27 | insert_time                             |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | n_diff_pfx02 |        915 |          27 | insert_time,order_status                |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | n_diff_pfx03 |       9774 |          27 | insert_time,order_status,expire_time    |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | n_leaf_pages |         27 |        NULL | Number of leaf pages in the index       |
| mysqladv      | order_exp  | u_idx_day_status | 2021-03-25 23:32:02 | size         |         28 |        NULL | Number of pages in the index            |
+--------------+------------+-----------------+---------------------+--------------+------------+-------------+-----------------------------------------+
16 rows in set (0.00 sec)

mysql> ANALYZE TABLE order_exp;
+-------------------+---------+----------+----------+
| Table             | Op      | Msg_type | Msg_text |
+-------------------+---------+----------+----------+
| mysqladv.order_exp | analyze | status   | OK       |
+-------------------+---------+----------+----------+
1 row in set (0.02 sec)

mysql> SELECT * FROM mysql.innodb_index_stats WHERE table_name = 'order_exp';
+--------------+------------+-----------------+---------------------+--------------+------------+-------------+-----------------------------------------+
| database_name | table_name | index_name      | last_update         | stat_name    | stat_value | sample_size | stat_description                        |
+--------------+------------+-----------------+---------------------+--------------+------------+-------------+-----------------------------------------+
| mysqladv      | order_exp  | PRIMARY         | 2021-04-01 00:05:51 | n_diff_pfx01 |      10345 |          20 | id                                      |
| mysqladv      | order_exp  | PRIMARY         | 2021-04-01 00:05:51 | n_leaf_pages |         86 |        NULL | Number of leaf pages in the index       |
| mysqladv      | order_exp  | PRIMARY         | 2021-04-01 00:05:51 | size         |         97 |        NULL | Number of pages in the index            |
| mysqladv      | order_exp  | idx_expire_time | 2021-04-01 00:05:51 | n_diff_pfx01 |       9802 |          13 | expire_time                             |
| mysqladv      | order_exp  | idx_expire_time | 2021-04-01 00:05:51 | n_diff_pfx02 |      10562 |          13 | expire_time,id                          |
```

ANALYZE TABLE语句会立即重新计算统计数据，也就是这个过程是同步的，在表中索引多或者采样页面特别多时这个过程可能会特别慢最好在业务不是很繁忙的时候再运行。

手动更新innodb_table_stats和innodb_index_stats表

其实innodb_table_stats和innodb_index_stats表就相当于一个普通的表一样，我们能对它们做增删改查操作。这也就意味着我们可以手动更新某个表或者索引的统计数据。

比如说我们想把order_exp表关于行数的统计数据更改一下可以这么做：

步骤一：更新innodb_table_stats表。

步骤二：让MySQL查询优化器重新加载我们更改过的数据。

更新完innodb_table_stats只是单纯的修改了一个表的数据，需要让MySQL查询优化器重新加载我们更改过的数据，运行下边的命令就可以了：

```
FLUSH TABLE order_exp;
```