第13章、InnoDB 统计数据是如何收集的

标签： MySQL 是怎样运行的

我们前边唠叨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？本章将聚焦于InnoDB存储引擎的统计数据收集策略，看完本章大家就会明白为啥前边老说InnoDB的统计信息是不精确的估计值了（言下之意就是我们不打算介绍MyISAM存储引擎统计数据的收集和存储方式，有想了解的同学自己个儿看看文档哈）。

两种不同的统计数据存储方式

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。

设计MySQL的大叔们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。

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：

```
mysql> SHOW TABLES FROM mysql LIKE 'innodb%';
+---------------------------+
| Tables_in_mysql (innodb%) |
+---------------------------+
| innodb_index_stats        |
| innodb_table_stats        |
+---------------------------+
2 rows in set (0.01 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。

我们下边的任务就是看一下这两个表里边都有什么以及表里的数据是如何生成的。

innodb_table_stats

直接看一下这个innodb_table_stats表中的各个列都是干嘛的：

字段名	描述
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

注意这个表的主键是(database_name,table_name)，也就是innodb_table_stats表的每条记录代表着一个表的统计信息。我们直接看一下这个表里的内容：

```
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 | 2018-07-10 23:51:36 |      0 |                    1 |                        0 |
| sys           | sys_config    | 2018-07-10 23:51:38 |      5 |                    1 |                        0 |
| xiaohaizi     | single_table  | 2018-12-10 17:03:13 |   9693 |                   97 |                      175 |
+---------------+---------------+---------------------+--------+----------------------+--------------------------+
3 rows in set (0.01 sec)
```

可以看到我们熟悉的single_table表的统计信息就对应着mysql.innodb_table_stats的第三条记录。几个重要统计信息项的值如下：

- n_rows的值是9693，表明single_table表中大约有9693条记录，注意这个数据是估计值。
- clustered_index_size的值是97，表明single_table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175，表明single_table表的其他索引一共占用175个页面，这个值是也是一个估计值。

n_rows统计项的收集

为啥老强调n_rows这个统计项的值是估计值呢？现在就来揭晓答案。InnoDB统计一个表中有多少行记录的套路是这样的：

- 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。

  > 小贴士：真实的计算过程比这个稍微复杂一些，不过大致上就是这样的啦~

  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，设计MySQL的大叔很贴心的为我们准备了一个名为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。

  我们前边说过，不过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

统计这两个数据需要大量用到我们之前唠叨的InnoDB表空间的知识，如果大家压根儿没有看那一章，那下边的计算过程大家还是不要看了（看也看不懂）；如果看过了，那大家就会发现InnoDB表空间的知识真是有用啊啊啊！！！

这两个统计项的收集过程如下：

- 从数据字典里找到表的各个索引对应的根页面位置。

  系统表SYS_INDEXES里存储了各个索引对应的根页面信息。

- 从根页面的Page Header里找到叶子节点段和非叶子节点段对应的Segment Header。

  在每个索引的根页面的Page Header部分都有两个字段：

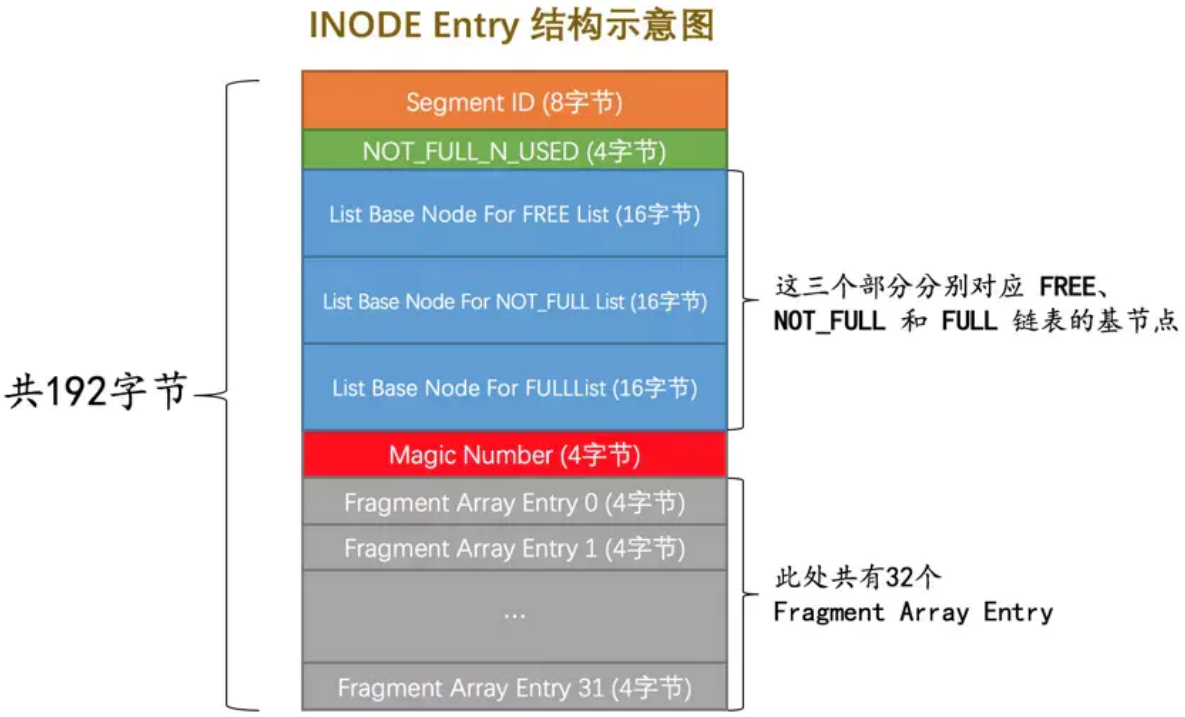
  - PAGE_BTR_SEG_LEAF：表示B+树叶子段的Segment Header信息。
  - PAGE_BTR_SEG_TOP：表示B+树非叶子段的Segment Header信息。

- 从叶子节点段和非叶子节点段的Segment Header中找到这两个段对应的INODE Entry结构。

这个是Segment Header结构：

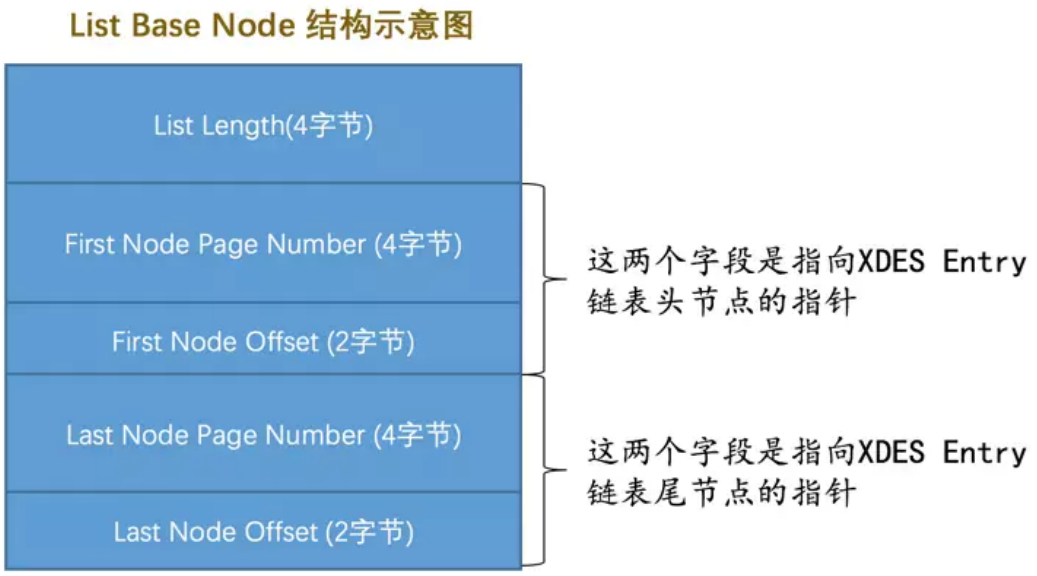
- 从对应的INODE Entry结构中可以找到该段对应所有零散的页面地址以及FREE、NOT_FULL、FULL链表的基节点。

  这个是INODE Entry结构：



- 直接统计零散的页面有多少个，然后从那三个链表的List Length字段中读出该段占用的区的大小，每个区占用64个页，所以就可以统计出整个段占用的页面。

  这个是链表基节点的示意图：



- 分别计算聚簇索引的叶子结点段和非叶子节点段占用的页面数，它们的和就是`clustered_index_size`的值，按照同样的套路把其余索引占用的页面数都算出来，加起来之后就是`sum_of_other_index_sizes`的值。

这里需要大家注意一个问题，我们说一个段的数据在非常多时（超过32个页面），会以区为单位来申请空间，这里头的问题是以区为单位申请空间中有一些页可能并没有使用，但是在统计`clustered_index_size`和`sum_of_other_index_sizes`时都把它们算进去了，所以说聚簇索引和其他的索引占用的页面数可能比这两个值要小一些。

innodb_index_stats

直接看一下这个`innodb_index_stats`表中的各个列都是干嘛的：

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

注意这个表的主键是`(database_name,table_name,index_name,stat_name)`，其中的`stat_name`是指统计项的名称，也就是说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`single_table`表的索引统计数据都有些什么：

```
mysql> SELECT * FROM mysql.innodb_index_stats WHERE table_name = 'single_table';
+--------------+--------------+--------------+---------------------+--------------+------------+-------------+---------------------------------+
| database_name | table_name   | index_name   | last_update         | stat_name    | stat_value | sample_size | stat_description                |
+--------------+--------------+--------------+---------------------+--------------+------------+-------------+---------------------------------+
| xiaohaizi     | single_table | PRIMARY      | 2018-12-14 14:24:46 | n_diff_pfx01 |       9693 |          20 | id                              |
| xiaohaizi     | single_table | PRIMARY      | 2018-12-14 14:24:46 | n_leaf_pages |         91 |        NULL | Number of leaf pages in the index |
| xiaohaizi     | single_table | PRIMARY      | 2018-12-14 14:24:46 | size         |         97 |        NULL | Number of pages in the index    |
| xiaohaizi     | single_table | idx_key1     | 2018-12-14 14:24:46 | n_diff_pfx01 |        968 |          28 | key1                            |
| xiaohaizi     | single_table | idx_key1     | 2018-12-14 14:24:46 | n_diff_pfx02 |      10000 |          28 | key1,id                         |
| xiaohaizi     | single_table | idx_key1     | 2018-12-14 14:24:46 | n_leaf_pages |         28 |        NULL | Number of leaf pages in the index |
| xiaohaizi     | single_table | idx_key1     | 2018-12-14 14:24:46 | size         |         29 |        NULL | Number of pages in the index    |
| xiaohaizi     | single_table | idx_key2     | 2018-12-14 14:24:46 | n_diff_pfx01 |      10000 |          16 | key2                            |
| xiaohaizi     | single_table | idx_key2     | 2018-12-14 14:24:46 | n_leaf_pages |         16 |        NULL | Number of leaf pages in the index |
| xiaohaizi     | single_table | idx_key2     | 2018-12-14 14:24:46 | size         |         17 |        NULL | Number of pages in the index    |
```

```
| xiaohaizi     | single_table | idx_key3     | 2018-12-14 14:24:46 | n_diff_pfx01 |        799 |          31 | key3                                |
| xiaohaizi     | single_table | idx_key3     | 2018-12-14 14:24:46 | n_diff_pfx02 |      10000 |          31 | key3,id                             |
| xiaohaizi     | single_table | idx_key3     | 2018-12-14 14:24:46 | n_leaf_pages |         31 |        NULL | Number of leaf pages in the index   |
| xiaohaizi     | single_table | idx_key3     | 2018-12-14 14:24:46 | size         |         32 |        NULL | Number of pages in the index        |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_diff_pfx01 |       9673 |          64 | key_part1                           |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_diff_pfx02 |       9999 |          64 | key_part1,key_part2                 |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_diff_pfx03 |      10000 |          64 | key_part1,key_part2,key_part3       |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_diff_pfx04 |      10000 |          64 | key_part1,key_part2,key_part3,id    |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | n_leaf_pages |         64 |        NULL | Number of leaf pages in the index   |
| xiaohaizi     | single_table | idx_key_part | 2018-12-14 14:24:46 | size         |         97 |        NULL | Number of pages in the index        |
+---------------+--------------+--------------+---------------------+--------------+------------+-------------+-------------------------------------+
20 rows in set (0.03 sec)
```

这个结果有点儿多，正确查看这个结果的方式是这样的：

- 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_key_part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idx_key_part来说：
    - n_diff_pfx01表示的是统计key_part1这单单一个列不重复的值有多少。
    - n_diff_pfx02表示的是统计key_part1、key_part2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key_part1、key_part2、key_part3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_part1、key_part2、key_part3、id这四个列组合起来不重复的值有多少。

    小贴士： 这里需要注意的是，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_key1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。比如上边的idx_key1有n_diff_pfx01、n_diff_pfx02两个统计项，而idx_key2却只有n_diff_pfx01一个统计项。
- 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。

  小贴士： 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。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表里的统计数据是不是也应该跟着变一变了？当然要变了，不变的话MySQL查询优化器计算的成本可就差老鼻子远了。设计MySQL的大叔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。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single_table表的统计数据：

```
mysql> ANALYZE TABLE single_table;
+------------------------+---------+----------+----------+
| Table                  | Op      | Msg_type | Msg_text |
+------------------------+---------+----------+----------+
| xiaohaizi.single_table | analyze | status   | OK       |
+------------------------+---------+----------+----------+
1 row in set (0.08 sec)
```

需要注意的是，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，请不要没事儿就运行一下ANALYZE TABLE语句，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single_table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。

  ```
  UPDATE innodb_table_stats
      SET n_rows = 1
      WHERE table_name = 'single_table';
  ```

- 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 single_table;
  ```

之后我们使用SHOW TABLE STATUS语句查看表的统计数据时就看到Rows行变为了1。

基于内存的非永久性统计数据

当我们把系统变量innodb_stats_persistent的值设置为OFF时，之后创建的表的统计数据默认就都是非永久性的了，或者我们直接在创建表或修改表时设置STATS_PERSISTENT属性的值为0，那么该表的统计数据就是非永久性的了。

与永久性的统计数据不同，非永久性的统计数据采样的页面数量是由innodb_stats_transient_sample_pages控制的，这个系统变量的默认值是8。

另外，由于非永久性的统计数据经常更新，所以导致MySQL查询优化器计算查询成本的时候依赖的是经常变化的统计数据，也就会生成经常变化的执行计划，这个可能让大家有些懵逼。不过最近的MySQL版本都不咋用这种基于内存的非永久性统计数据了，所以我们也就不深入唠叨它了。

innodb_stats_method的使用

我们知道索引列不重复的值的数量这个统计数据对于MySQL查询优化器十分重要，因为通过它可以计算出在索引列中平均一个值重复多少行，它的应用场景主要有两个：

- 单表查询中单点区间太多，比方说这样：

  ```
  SELECT * FROM tbl_name WHERE key IN ('xx1', 'xx2', ..., 'xxn');
  ```

  当IN里的参数数量过多时，采用index dive的方式直接访问B+树索引去统计每个单点区间对应的记录的数量就太耗费性能了，所以直接依赖统计数据中的平均一个值重复多少行来计算单点区间对应的记录数量。

- 连接查询时，如果有涉及两个表的等值匹配连接条件，该连接条件对应的被驱动表中的列又拥有索引时，则可以使用ref访问方法来对被驱动表进行查询，比方说这样：

  ```
  SELECT * FROM t1 JOIN t2 ON t1.column = t2.key WHERE ...;
  ```

  在真正执行对t2表的查询前，t1.comumn的值是不确定的，所以我们也不能通过index dive的方式直接访问B+树索引去统计每个单点区间对应的记录的数量，所以也只能依赖统计数据中的平均一个值重复多少行来计算单点区间对应的记录数量。

在统计索引列不重复的值的数量时，有一个比较烦的问题就是索引列中出现NULL值怎么办，比方说某个索引列的内容是这样：

```
+------+
| col  |
+------+
```

```
|    1 |
|    2 |
| NULL |
| NULL |
+------+
```

此时计算这个col列中不重复的值的数量就有下边的分歧：

- 有的人认为NULL值代表一个未确定的值，所以设计MySQL的大叔才认为任何和NULL值做比较的表达式的值都为NULL，就是这样：

  ```
  mysql> SELECT 1 = NULL;
  +----------+
  | 1 = NULL |
  +----------+
  |     NULL |
  +----------+
  1 row in set (0.00 sec)

  mysql> SELECT 1 != NULL;
  +-----------+
  | 1 != NULL |
  +-----------+
  |      NULL |
  +-----------+
  1 row in set (0.00 sec)

  mysql> SELECT NULL = NULL;
  +-------------+
  | NULL = NULL |
  +-------------+
  |        NULL |
  +-------------+
  1 row in set (0.00 sec)

  mysql> SELECT NULL != NULL;
  +--------------+
  | NULL != NULL |
  +--------------+
  |         NULL |
  +--------------+
  1 row in set (0.00 sec)
  ```

  所以每一个NULL值都是独一无二的，也就是说统计索引列不重复的值的数量时，应该把NULL值当作一个独立的值，所以col列的不重复的值的数量就是：4（分别是1、2、NULL、NULL这四个值）。

- 有的人认为其实NULL值在业务上就是代表没有，所有的NULL值代表的意义是一样的，所以col列不重复的值的数量就是：3（分别是1、2、NULL这三个值）。

- 有的人认为这NULL完全没有意义嘛，所以在统计索引列不重复的值的数量时压根儿不能把它们算进来，所以col列不重复的值的数量就是：2（分别是1、2这两个值）。

设计MySQL的大叔蛮贴心的，他们提供了一个名为innodb_stats_method的系统变量，相当于在计算某个索引列不重复值的数量时如何对待NULL值这个锅甩给了用户，这个系统变量有三个候选值：

- nulls_equal：认为所有NULL值都是相等的。这个值也是innodb_stats_method的默认值。

  如果某个索引列中NULL值特别多的话，这种统计方式会让优化器认为某个列中平均一个值重复次数特别多，所以倾向于不使用索引进行访问。

- nulls_unequal：认为所有NULL值都是不相等的。

  如果某个索引列中NULL值特别多的话，这种统计方式会让优化器认为某个列中平均一个值重复次数特别少，所以倾向于使用索引进行访问。

- nulls_ignored：直接把NULL值忽略掉。

反正这个锅是甩给用户了，当你选定了innodb_stats_method值之后，优化器即使选择了不是最优的执行计划，那也跟设计MySQL的大叔们没关系了哈～ 当然对于用户的我们来说，最好不在索引列中存放NULL值才是正解。

总结

- InnoDB以表为单位来收集统计数据，这些统计数据可以是基于磁盘的永久性统计数据，也可以是基于内存的非永久性统计数据。
- innodb_stats_persistent控制着使用永久性统计数据还是非永久性统计数据；innodb_stats_persistent_sample_pages控制着永久性统计数据的采样页面数量；innodb_stats_transient_sample_pages控制着非永久性统计数据的采样页面数量；innodb_stats_auto_recalc控制着是否自动重新计算统计数据。
- 我们可以针对某个具体的表，在创建和修改表时通过指定STATS_PERSISTENT、STATS_AUTO_RECALC、STATS_SAMPLE_PAGES的值来控制相关统计数据属性。
- innodb_stats_method决定着在统计某个索引列不重复值的数量时如何对待NULL值。