10 | MySQL为什么有时候会选错索引?

2018-12-05 林晓斌

10 | MySQL为什么有时候会选错索引?
朗读人：林晓斌 17′25″ | 15.97M

前面我们介绍过索引，你已经知道了在 MySQL 中一张表其实是可以支持多个索引的。但是，你写 SQL 语句的时候，并没有主动指定使用哪个索引。也就是说，使用哪个索引是由 MySQL 来确定的。

不知道你有没有碰到过这种情况，一条本来可以执行得很快的语句，却由于 MySQL 选错了索引，而导致执行速度变得很慢？

我们一起来看一个例子吧。

我们先建一个简单的表，表里有 a、b 两个字段，并分别建上索引：

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `a` (`a`),
  KEY `b` (`b`)
) ENGINE=InnoDB;
```

然后，我们往表 t 中插入 10 万行记录，取值按整数递增，即：(1,1,1)，(2,2,2)，(3,3,3) 直到 (100000,100000,100000)。

我是用存储过程来插入数据的，这里我贴出来方便你复现：

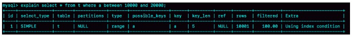
```
delimiter ;;
create procedure idata()
begin
  declare i int;
  set i=1;
  while(i<=100000)do
    insert into t values(i, i, i);
    set i=i+1;
  end while;
end;;
delimiter ;
call idata();
```

接下来，我们分析一条 SQL 语句：

```
mysql> select * from t where a between 10000 and 20000;
```

你一定会说，这个语句还用分析吗，很简单呀，a 上有索引，肯定是要使用索引 a 的。

你说得没错，图 1 显示的就是使用 explain 命令看到的这条语句的执行情况。



```
mysql> explain select * from t where a between 10000 and 20000;
+----+-------------+-------+------------+-------+---------------+------+---------+------+-------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key  | key_len | ref  | rows  | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+------+---------+------+-------+----------+-----------------------+
|  1 | SIMPLE      | t     | NULL       | range | a             | a    | 5       | NULL | 10001 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+------+---------+------+-------+----------+-----------------------+
```

图 1 使用 explain 命令查看语句执行情况

从图 1 看上去，这条查询语句的执行也确实符合预期，key 这个字段值是’a’，表示优化器选择了索引 a。

不过别急，这个案例不会这么简单。在我们已经准备好的包含了 10 万行数据的表上，我们再做如下操作。

session A	session B
start transaction with consistent snapshot;	
	delete from t; call idata();
commit;	
	explain select * from t where a between 10000 and 20000;

图 2 session A 和 session B 的执行流程

这里，session A 的操作你已经很熟悉了，它就是开启了一个事务。随后，session B 把数据都删除后，又调用了 idata 这个存储过程，插入了 10 万行数据。

这时候，session B 的查询语句 select * from t where a between 10000 and 20000 就不会再选择索引 a 了。我们可以通过慢查询日志（slow log）来查看一下具体的执行情况。

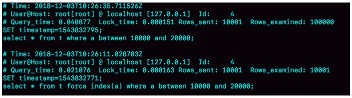
为了说明优化器选择的结果是否正确，我增加了一个对照，即：使用 force index(a) 来让优化器强制使用索引 a（这部分内容，我还会在这篇文章的后半部分中提到）。

下面的三条 SQL 语句，就是这个实验过程。

```
set sql_long_query=0;
select * from t where a between 10000 and 20000; /*Q1*/
select * from t force index(a) where a between 10000 and 20000;/*Q2*/
```

- 第一句，是将慢查询日志的阈值设置为 0，表示这个线程接下来的语句都会被记录入慢查询日志中；
- 第二句，Q1 是 session B 原来的查询；
- 第三句，Q2 是加了 force index(a) 来和 session B 原来的查询语句执行情况对比。

如图 3 所示是这三条 SQL 语句执行完成后的慢查询日志。



```
# Time: 2018-12-03T10:26:35.711526Z
# User@Host: root[root] @ localhost [127.0.0.1]  Id:     4
# Query_time: 0.040877  Lock_time: 0.000151 Rows_sent: 10001  Rows_examined: 100000
SET timestamp=1543832795;
select * from t where a between 10000 and 20000;

# Time: 2018-12-03T10:26:11.028703Z
# User@Host: root[root] @ localhost [127.0.0.1]  Id:     4
# Query_time: 0.021076  Lock_time: 0.000163 Rows_sent: 10001  Rows_examined: 10001
SET timestamp=1543832771;
select * from t force index(a) where a between 10000 and 20000;
```

图 3 slow log 结果

可以看到，Q1 扫描了 10 万行，显然是走了全表扫描，执行时间是 40 毫秒。Q2 扫描了 10001 行，执行了 21 毫秒。也就是说，我们在没有使用 force index 的时候，MySQL 用错了索引，导致了更长的执行时间。

这个例子对应的是我们平常不断地删除历史数据和新增数据的场景。这时，MySQL 竟然会选错索引，是不是有点奇怪呢？今天，我们就从这个奇怪的结果说起吧。

优化器的逻辑

在第一篇文章中，我们就提到过，选择索引是优化器的工作。

而优化器选择索引的目的，是找到一个最优的执行方案，并用最小的代价去执行语句。在数据库里面，扫描行数是影响执行代价的因素之一。扫描的行数越少，意味着访问磁盘数据的次数越少，消耗的 CPU 资源越少。

当然，扫描行数并不是唯一的判断标准，优化器还会结合是否使用临时表、是否排序等因素进行综合判断。

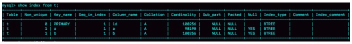
我们这个简单的查询语句并没有涉及到临时表和排序，所以 MySQL 选错索引肯定是在判断扫描行数的时候出问题了。

那么，问题就是：**扫描行数是怎么判断的？**

MySQL 在真正开始执行语句之前，并不能精确地知道满足这个条件的记录有多少条，而只能根据统计信息来估算记录数。

这个统计信息就是索引的“区分度”。显然，一个索引上不同的值越多，这个索引的区分度就越好。而一个索引上不同的值的个数，我们称之为“基数”（cardinality）。也就是说，这个基数越大，索引的区分度越好。

我们可以使用 show index 方法，看到一个索引的基数。如图 4 所示，就是表 t 的 show index 的结果 。虽然这个表的每一行的三个字段值都是一样的，但是在统计信息中，这三个索引的基数值并不同，而且其实都不准确。



```
mysql> show index from t;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| t     |          0 | PRIMARY  |            1 | id          | A         |      100256 |     NULL | NULL   |      | BTREE      |         |               |
| t     |          1 | a        |            1 | a           | A         |       98190 |     NULL | NULL   | YES  | BTREE      |         |               |
| t     |          1 | b        |            1 | b           | A         |      100256 |     NULL | NULL   | YES  | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
```

图 4 表 t 的 show index 结果

那么，**MySQL 是怎样得到索引的基数的呢？** 这里，我给你简单介绍一下 MySQL 采样统计的方法。

为什么要采样统计呢？因为把整张表取出来一行行统计，虽然可以得到精确的结果，但是代价太高了，所以只能选择“采样统计”。

采样统计的时候，InnoDB 默认会选择 N 个数据页，统计这些页面上的不同值，得到一个平均值，然后乘以这个索引的页面数，就得到了这个索引的基数。

而数据表是会持续更新的，索引统计信息也不会固定不变。所以，当变更的数据行数超过 1/M 的时候，会自动触发重新做一次索引统计。

在 MySQL 中，有两种存储索引统计的方式，可以通过设置参数 innodb_stats_persistent 的值来选择：

- 设置为 on 的时候，表示统计信息会持久化存储。这时，默认的 N 是 20，M 是 10。
- 设置为 off 的时候，表示统计信息只存储在内存中。这时，默认的 N 是 8，M 是 16。

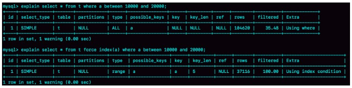
由于是采样统计，所以不管 N 是 20 还是 8，这个基数都是很容易不准的。

但，这还不是全部。

你可以从图 4 中看到，这次的索引统计值（cardinality 列）虽然不够精确，但大体上还是差不多的，选错索引一定还有别的原因。

其实索引统计只是一个输入，对于一个具体的语句来说，优化器还要判断，执行这个语句本身要扫描多少行。

接下来，我们再一起看看优化器预估的，这两个语句的扫描行数是多少。



```
mysql> explain select * from t where a between 10000 and 20000;
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------+
|  1 | SIMPLE      | t     | NULL       | ALL  | a             | NULL | NULL    | NULL | 104620 |    35.48 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain select * from t force index(a) where a between 10000 and 20000;
+----+-------------+-------+------------+-------+---------------+------+---------+------+-------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key  | key_len | ref  | rows  | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+------+---------+------+-------+----------+-----------------------+
|  1 | SIMPLE      | t     | NULL       | range | a             | a    | 5       | NULL | 37116 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+------+---------+------+-------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

图 5 意外的 explain 结果

rows 这个字段表示的是预计扫描行数。

其中，Q1 的结果还是符合预期的，rows 的值是 104620；但是 Q2 的 rows 值是 37116，偏差就大了。而图 1 中我们用 explain 命令看到的 rows 是只有 10001 行，是这个偏差误导了优化器的判断。

到这里，可能你的第一个疑问不是为什么不准，而是优化器为什么放着扫描 37000 行的执行计划不用，却选择了扫描行数是 100000 的执行计划呢？

这是因为，如果使用索引 a，每次从索引 a 上拿到一个值，都要回到主键索引上查出整行数据，这个代价优化器也要算进去的。

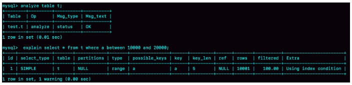
而如果选择扫描 10 万行，是直接在主键索引上扫描的，没有额外的代价。

优化器会估算这两个选择的代价，从结果看来，优化器认为直接扫描主键索引更快。当然，从执行时间看来，这个选择并不是最优的。

使用普通索引需要把回表的代价算进去，在图 1 执行 explain 的时候，也考虑了这个策略的代价，但图 1 的选择是对的。也就是说，这个策略并没有问题。

所以冤有头债有主，MySQL 选错索引，这件事儿还得归咎到没能准确地判断出扫描行数。至于为什么会得到错误的扫描行数，这个原因就作为课后问题，留给你去分析了。

既然是统计信息不对，那就修正。analyze table t 命令，可以用来重新统计索引信息。我们来看一下执行效果。



```
mysql> analyze table t;
+--------+---------+----------+----------+
| Table  | Op      | Msg_type | Msg_text |
+--------+---------+----------+----------+
| test.t | analyze | status   | OK       |
+--------+---------+----------+----------+
1 row in set (0.01 sec)

mysql>  explain select * from t where a between 10000 and 20000;
+----+-------------+-------+------------+-------+---------------+-----+---------+------+-------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key | key_len | ref  | rows  | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+-----+---------+------+-------+----------+-----------------------+
|  1 | SIMPLE      | t     | NULL       | range | a             | a   | 5       | NULL | 10001 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+-----+---------+------+-------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

图 6 执行 analyze table t 命令恢复的 explain 结果

这回对了。

所以在实践中，如果你发现 explain 的结果预估的 rows 值跟实际情况差距比较大，可以采用这个方法来处理。

其实，如果只是索引统计不准确，通过 analyze 命令可以解决很多问题，但是前面我们说了，优化器可不止是看扫描行数。

依然是基于这个表 t，我们看看另外一个语句：

```
mysql> select * from t where (a between 1 and 1000)  and (b between 50000 and 100000) order by b li
```

从条件上看，这个查询没有符合条件的记录，因此会返回空集合。

在开始执行这条语句之前，你可以先设想一下，如果你来选择索引，会选择哪一个呢？

为了便于分析，我们先来看一下 a、b 这两个索引的结构图。

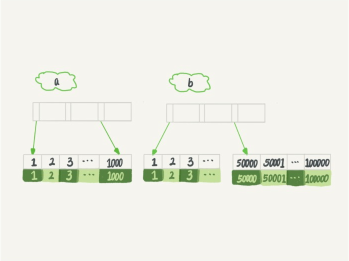


图 7 a、b 索引的结构图

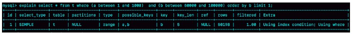
如果使用索引 a 进行查询，那么就是扫描索引 a 的前 1000 个值，然后取到对应的 id，再到主键索引上去查出每一行，然后根据字段 b 来过滤。显然这样需要扫描 1000 行。

如果使用索引 b 进行查询，那么就是扫描索引 b 的最后 50001 个值，与上面的执行过程相同，也是需要回到主键索引上取值再判断，所以需要扫描 50001 行。

所以你一定会想，如果使用索引 a 的话，执行速度明显会快很多。那么，下面我们就来看看到底是不是这么一回事儿。

图 8 是执行 explain 的结果。

```
mysql> explain select * from t where (a between 1 and 1000) and (b between 50000 and 100000) order
```



```
mysql> explain select * from t where (a between 1 and 1000)  and (b between 50000 and 100000) order by b limit 1;
+----+-------------+-------+------------+-------+---------------+-----+---------+------+-------+----------+------------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key | key_len | ref  | rows  | filtered | Extra                              |
+----+-------------+-------+------------+-------+---------------+-----+---------+------+-------+----------+------------------------------------+
|  1 | SIMPLE      | t     | NULL       | range | a,b           | b   | 5       | NULL | 50198 |     1.00 | Using index condition; Using where |
+----+-------------+-------+------------+-------+---------------+-----+---------+------+-------+----------+------------------------------------+
```

图 8 使用 explain 方法查看执行计划 2

可以看到，返回结果中 key 字段显示，这次优化器选择了索引 b，而 rows 字段显示需要扫描的行数是 50198。

从这个结果中，你可以得到两个结论：

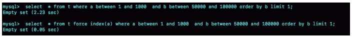
1. 扫描行数的估计值依然不准确；
2. 这个例子里 MySQL 又选错了索引。

索引选择异常和处理

其实大多数时候优化器都能找到正确的索引，但偶尔你还是会碰到我们上面举例的这两种情况：原本可以执行得很快的 SQL 语句，执行速度却比你预期的慢很多，你应该怎么办呢？

一种方法是，像我们第一个例子一样，采用 force index 强行选择一个索引。 MySQL 会根据词法解析的结果分析出可能可以使用的索引作为候选项，然后在候选列表中依次判断每个索引需要扫描多少行。如果 force index 指定的索引在候选索引列表中，就直接选择这个索引，不再评估其他索引的执行代价。

我们来看看第二个例子。刚开始分析时，我们认为选择索引 a 会更好。现在，我们就来看看执行效果：



```
mysql>  select  * from t where a between 1 and 1000  and b between 50000 and 100000 order by b limit 1;
Empty set (2.23 sec)

mysql>  select  * from t force index(a) where a between 1 and 1000  and b between 50000 and 100000 order by b limit 1;
Empty set (0.05 sec)
```

图 9 使用不同索引的语句执行耗时

可以看到，原本语句需要执行 2.23 秒，而当你使用 force index(a) 的时候，只用了 0.05 秒，比优

化器的选择快了 40 多倍。

也就是说，优化器没有选择正确的索引，force index 起到了“矫正”的作用。

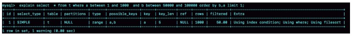
不过很多程序员不喜欢使用 force index，一来这么写不优美，二来如果索引改了名字，这个语句也得改，显得很麻烦。而且如果以后迁移到别的数据库的话，这个语法还可能会不兼容。

但其实使用 force index 最主要的问题还是变更的及时性。因为选错索引的情况还是比较少出现的，所以开发的时候通常不会先写上 force index。而是等到线上出现问题的时候，你才会再去修改 SQL 语句、加上 force index。但是修改之后还要测试和发布，对于生产系统来说，这个过程不够敏捷。

所以，数据库的问题最好还是在数据库内部来解决。那么，在数据库里面该怎样解决呢？

既然优化器放弃了使用索引 a，说明 a 还不够合适，所以**第二种方法就是，我们可以考虑修改语句，引导 MySQL 使用我们期望的索引。**比如，在这个例子里，显然把“order by b limit 1” 改成 “order by b,a limit 1” ，语义的逻辑是相同的。

我们来看看改之后的效果：



```
mysql>  explain select  * from t where a between 1 and 1000  and b between 50000 and 100000 order by b,a limit 1;
+----+-------------+-------+------------+-------+---------------+-----+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+-------+---------------+-----+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | t     | NULL       | range | a,b           | a   | 5       | NULL | 1000 |    50.00 | Using index condition; Using where; Using filesort |
+----+-------------+-------+------------+-------+---------------+-----+---------+------+------+----------+----------------------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

图 10 order by b,a limit 1 执行结果

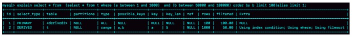
之前优化器选择使用索引 b，是因为它认为使用索引 b 可以避免排序（b 本身是索引，已经是有序的了，如果选择索引 b 的话，不需要再做排序，只需要遍历），所以即使扫描行数多，也判定为代价更小。

现在 order by b,a 这种写法，要求按照 b,a 排序，就意味着使用这两个索引都需要排序。因此，扫描行数成了影响决策的主要条件，于是此时优化器选了只需要扫描 1000 行的索引 a。

当然，这种修改并不是通用的优化手段，只是刚好在这个语句里面有 limit 1，因此如果有满足条件的记录， order by b limit 1 和 order by b,a limit 1 都会返回 b 是最小的那一行，逻辑上一致，才可以这么做。

如果你觉得修改语义这件事儿不太好，这里还有一种改法，图 11 是执行效果。

```
mysql> select * from  (select * from t where (a between 1 and 1000)  and (b between 50000 and 10000
```



```
mysql> explain select * from  (select * from t where (a between 1 and 1000)  and (b between 50000 and 100000) order by b limit 100)alias limit 1;
+----+-------------+------------+------------+-------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table      | partitions | type  | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+------------+------------+-------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | PRIMARY     | <derived2> | NULL       | ALL   | NULL          | NULL | NULL    | NULL |  100 |   100.00 | NULL                                               |
|  2 | DERIVED     | t          | NULL       | range | a,b           | a    | 5       | NULL | 1000 |    50.00 | Using index condition; Using where; Using filesort |
+----+-------------+------------+------------+-------+---------------+------+---------+------+------+----------+----------------------------------------------------+
```

图 11 改写 SQL 的 explain

在这个例子里，我们用 limit 100 让优化器意识到，使用 b 索引代价是很高的。其实是我们根据数据特征诱导了一下优化器，也不具备通用性。

第三种方法是，在有些场景下，我们可以新建一个更合适的索引，来提供给优化器做选择，或删掉误用的索引。

不过，在这个例子中，我没有找到通过新增索引来改变优化器行为的方法。这种情况其实比较少，尤其是经过 DBA 索引优化过的库，再碰到这个 bug，找到一个更合适的索引一般比较难。

如果我说还有一个方法是删掉索引 b，你可能会觉得好笑。但实际上我碰到过两次这样的例子，最终是 DBA 跟业务开发沟通后，发现这个优化器错误选择的索引其实根本没有必要存在，于是就删掉了这个索引，优化器也就重新选择到了正确的索引。

小结

今天我们一起聊了聊索引统计的更新机制，并提到了优化器存在选错索引的可能性。

对于由于索引统计信息不准确导致的问题，你可以用 analyze table 来解决。

而对于其他优化器误判的情况，你可以在应用端用 force index 来强行指定索引，也可以通过修改语句来引导优化器，还可以通过增加或者删除索引来绕过这个问题。

你可能会说，今天这篇文章后面的几个例子，怎么都没有展开说明其原理。我要告诉你的是，今天的话题，我们面对的是 MySQL 的 bug，每一个展开都必须深入到一行行代码去量化，实在不是我们在这里应该做的事情。

所以，我把我用过的解决方法跟你分享，希望你在碰到类似情况的时候，能够有一些思路。

你平时在处理 MySQL 优化器 bug 的时候有什么别的方法，也发到评论区分享一下吧。

最后，我给你留下一个思考题。前面我们在构造第一个例子的过程中，通过 session A 的配合，让 session B 删除数据后又重新插入了一遍数据，然后就发现 explain 结果中，rows 字段从 10001 变成 37000 多。

而如果没有 session A 的配合，只是单独执行 delete from t 、call idata()、explain 这三句话，会看到 rows 字段其实还是 10000 左右。你可以自己验证一下这个结果。

这是什么原因呢？也请你分析一下吧。

你可以把你的分析结论写在留言区里，我会在下一篇文章的末尾和你讨论这个问题。感谢你的收听，也欢迎你把这篇文章分享给更多的朋友一起阅读。

上期问题时间

我在上一篇文章最后留给你的问题是，如果某次写入使用了 change buffer 机制，之后主机异常重启，是否会丢失 change buffer 和数据。

这个问题的答案是不会丢失，留言区的很多同学都回答对了。虽然是只更新内存，但是在事务提交的时候，我们把 change buffer 的操作也记录到 redo log 里了，所以崩溃恢复的时候，change buffer 也能找回来。

在评论区有同学问到，merge 的过程是否会把数据直接写回磁盘，这是个好问题。这里，我再为你分析一下。

merge 的执行流程是这样的：

1. 从磁盘读入数据页到内存（老版本的数据页）；
2. 从 change buffer 里找出这个数据页的 change buffer 记录 (可能有多个），依次应用，得到新版数据页；
3. 写 redo log。这个 redo log 包含了数据的变更和 change buffer 的变更。

到这里 merge 过程就结束了。这时候，数据页和内存中 change buffer 对应的磁盘位置都还没有修改，属于脏页，之后各自刷回自己的物理数据，就是另外一个过程了。

评论区留言点赞板：

> @某、人 把 02 篇的 redo log 更新细节和 change buffer 的更新串了起来；
> @Ivan 回复了其他同学的问题，并联系到 Checkpoint 机制；
> @约书亚 问到了 merge 和 redolog 的关系。

极客时间
MySQL 实战 45 讲
从原理到实战，丁奇带你搞懂 MySQL
林晓斌
网名丁奇
前阿里资深技术专家

©版权归极客邦科技所有，未经许可不得转载

上一篇 09 | 普通索引和唯一索引，应该怎么选择？

写留言

精选留言

bowenz 0

在5.7.21 percona 版本实验，未出现案例1的情况 。

```
dev02> select @@global.tx_isolation,@@tx_isolation,version(),"session A";
+-----------------------+-----------------+---------------+-----------+
| @@global.tx_isolation | @@tx_isolation | version() | session A |
+-----------------------+-----------------+---------------+-----------+
| REPEATABLE-READ | REPEATABLE-READ | 5.7.21-20-log | session A |
+-----------------------+-----------------+---------------+-----------+
dev02> start transaction with consistent snapshot;
Query OK, 0 rows affected (0.00 sec)
dev02> commit;
Query OK, 0 rows affected (0.00 sec)
dev02> select now() ;
+---------------------+
| now() |
+---------------------+
| 2018-12-04 22:03:48 |
+---------------------+
1 row in set (0.00 sec)
dev02> select @@global.tx_isolation,@@tx_isolation,version(),"session B";
+-----------------------+-----------------+---------------+-----------+
| @@global.tx_isolation | @@tx_isolation | version() | session B |
+-----------------------+-----------------+---------------+-----------+
| REPEATABLE-READ | REPEATABLE-READ | 5.7.21-20-log | session B |
+-----------------------+-----------------+---------------+-----------+
1 row in set, 2 warnings (0.00 sec)

dev02> delete from t;
Query OK, 100000 rows affected (0.51 sec)

dev02> call idata();
Query OK, 1 row affected (2 min 38.34 sec)

dev02> select now();
+---------------------+
| now() |
+---------------------+
| 2018-12-04 22:03:58 |
+---------------------+
1 row in set (0.00 sec)

dev02> explain select * from t where a between 10000 and 20000;
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
| 1 | SIMPLE | t | NULL | range | a | a | 5 | NULL | 10001 | 100.00 | Using index condition |
```

2018-12-05

作者回复

Session A提交早了… 从上到下按照时间顺序执行哈

2018-12-05

斜面镜子 Bill 3

问题的思考：
我理解 session A 开启的事务对 session B的delete操作后的索引数据的统计时效产生了影响，因为需要保证事务A的重复读，在数据页没有实际删除，而索引的统计选择了N个数据页，这部分数据页不收到前台事务的影响，所以整体统计值会变大，直接影响了索引选择的准确性；

2018-12-05

作者回复

👍

2018-12-05

Leon📷 1

公司测试机器IO性能太差，插十万条要27分钟，做这个文章的实验要1个小时以上

2018-12-05

高枕 1

老师，您文中提到：“从 change buffer 里找出这个数据页的 change buffer 记录 (可能有多个），依次应用，得到新版数据页；

写 redo log。这个 redo log 包含了数据的变更和 change buffer 的变更。”
您的意思是，这些新版的数据页后的数据，没有直接写回到磁盘（数据页），而是写进了redo log?
写回数据页和更新系统表空间对应的change buffer是另外的一个过程?
写进

2018-12-05

作者回复

1. 是的

2018-12-05

高枕 1

老师，您文中最后写道：“这时候，数据页和内存中 change buffer 对应的磁盘位置都还没有修改，属于脏页，之后各自刷回自己的物理数据，就是另外一个过程了。”
能再给解释下change buffer和数据页对应的磁盘位置是哪里吗？
我理解change buffer 对应的磁盘位置是系统表空间，数据页所在的磁盘位置是表空间的位置，marge并没有把数据直接写道这个表空间的位置吗？

2018-12-05

作者回复

前面两句理解对应位置是对的哈，

最后一句问题：没有马上写

2018-12-05

沉浮 1

图十下面第二段
现在 limit b,a 这种写法，要求按照 b,a 排序，就意味着使用这两个索引都需要排序。
应该是order by b,a吧
另外有个问题请教林老师，根据经验大表增加索引的时候比较慢，这个是理解的，但是删除索引的时候能做到秒删，这个什么原理呢？

2018-12-05

作者回复

是，已经修改了，谢谢。

删除的时候是标记删除，所以很快。

建索引是要扫描数据和真正生成索引树，是会慢些

2018-12-05

Ying 0

现学现用 今天有个500万的表 分页查询特别慢。
select * from table where create_time and create_time>=时间戳 and create_time<=时间戳
and subtype='xx' and type='xx' and company_id =x order by create_time limited 90,30 ;
已经建立了组合索引 union_index包括字段 create_time subtype type company_id
但是 explain 发现竟然走了create_time 的索引
语句里加了一个use index(union_index) ，立马好了
真正的解决了客户的实际问题啊。 感谢老师

2018-12-05

作者回复

👍

而且发评论的时候还做了很细致地脱敏，赞

2018-12-05

monkay 0

merge的第三部写redo log，不是前面的更新语句已经写了redo log了吗？为什么merge时又写一次？

2018-12-05

作者回复

Merge的过程中要修改数据页，这次记数据页的

2018-12-05

Leon📷 0

老师，是不是新版mysql5.7已经修复了你说的这个bug，我的slow_log是这样的

```
SET timestamp=1543999785;
set long_query_time=0;
# Time: 2018-12-05T08:49:58.292037Z
# User@Host: root[root] @ localhost [] Id: 7
# Query_time: 0.017165 Lock_time: 0.000179 Rows_sent: 10001 Rows_examined: 10001
SET timestamp=1543999798;
select * from t where a between 10000 and 20000;
# Time: 2018-12-05T08:50:04.762129Z
# User@Host: root[root] @ localhost [] Id: 7
# Query_time: 0.014629 Lock_time: 0.000170 Rows_sent: 10001 Rows_examined: 10001
SET timestamp=1543999804;
select * from t force index(a) where a between 10000 and 20000;
```

2018-12-05

无眠 0

有些不太明白，图11下面描述的“我们用limit 100让优化器意识到，使用b索引代价是很高的”这句，limit 100怎么就代价高了？另外，优化器是如何分析limit子句的？对于limit 子句是在引擎层还是server层起作用的？望老师解惑

2018-12-05

kunsect 0

老师，我这边复现不了走全局扫描的案例。

```
mysql> select version();
+-----------+
| version() |
+-----------+
| 5.7.22 |
+-----------+
1 row in set (0.00 sec)

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

```
Session A:
mysql> start transaction with consistent snapshot;
Query OK, 0 rows affected (0.00 sec)

Session B:
mysql> delete from t;
Query OK, 100000 rows affected (0.52 sec)

Session B:
mysql> call idata();
Query OK, 1 row affected (1.91 sec)

Session A:
mysql> commit;
Query OK, 0 rows affected (0.00 sec)

Session B:
mysql> explain select * from t where a between 10000 and 20000;
```

```
+----+-------------+-------+------------+-------+---------------+------+---------+------+-
------+----------+-----------------------+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filter
ed | Extra |
+----+-------------+-------+------------+-------+---------------+------+---------+------+-
------+----------+-----------------------+
| 1 | SIMPLE | t | NULL | range | a | a | 5 | NULL | 10001 | 100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+------+---------+------+-
------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

2018-12-05

Jason

大神好，提前问一句，MySQL表分区方案靠谱么?每天产生上百万的大表，想按时间进行分区，不知道会不会踩坑，看很多人说不建议用。

2018-12-05

[illegible]

老师anlyze table或者optimize table都是会锁表的吧，线上可以直接执行吗?

2018-12-05

tchz

5.6.24，show variables like ‘tx_isolation’ 为REPEATABLE-READ，未复现，且explain的rows为10000

2018-12-05

lianjie1229

第一个选错索引的例子，在5.7.24版本上没复现出来

session A:

```
| transaction_isolation | REPEATABLE-READ |
13 rows in set (0.01 sec)
mysql> start transaction with consistent snapshot;
Query OK, 0 rows affected (0.00 sec)
mysql> select now();
+---------------------+
| now() |
+---------------------+
| 2018-12-04 22:20:03 |
+---------------------+
1 row in set (0.00 sec)

mysql> select now();
+---------------------+
| now() |
+---------------------+
| 2018-12-04 22:22:16 |
+---------------------+
1 row in set (0.00 sec)
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

session B:

```
mysql> select now();
+---------------------+
| now() |
+---------------------+
| 2018-12-04 22:20:20 |
+---------------------+
1 row in set (0.00 sec)
mysql>
mysql> delete from t;
Query OK, 100000 rows affected (0.66 sec)
mysql> call idata();
Query OK, 1 row affected (52.57 sec)
mysql> select now();
+---------------------+
| now() |
+---------------------+
| 2018-12-04 22:21:57 |
+---------------------+
1 row in set (0.01 sec)

mysql> explain select * from t where a between 10000 and 20000;
+----+-------------+-------+------------+-------+---------------+------+---------+------+-
------+----------+-----------------------+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filter
ed | Extra |
+----+-------------+-------+------------+-------+---------------+------+---------+------+-
------+----------+-----------------------+
| 1 | SIMPLE | t | NULL | range | a | a | 5 | NULL | 10001 | 100.00 | Using index condition |
```

2018-12-05

木偶人King

老师你好, 这个创建索引的时候,我看到这里经常用的是key, 而不是index. 这个有什么区别吗?

2018-12-05

作者回复

没区别

2018-12-05

bowenz

在5.7.21版本没有出现案例1的情况

留言好像上不了截图

2018-12-05

作者回复

发个微博或博客，然后贴地址。我也用的5.7.21...

2018-12-05

Laputa

老师，redo log 是实时写入磁盘的吗？是不是还有一层所谓的“redo log buffer”？

2018-12-05

作者回复

Redolog buffer是在事务执行过程中，先把要写入的内容在内存里存起来，在commit阶段，一次性写入redolog file

2018-12-05

Tony Du

对于redo log，后台会定期把redo log的内容刷到磁盘，然后对于change buffer，后台也会定期做merge操作（或者当访问数据页的时候做定期操作）。那当redo log刷到磁盘上后，读取磁盘上的数据页就是最新的了，则此时当读取数据页时的merge操作（merge操作 = 更新操作 + 老的数据页面）不就显得不不正确了?

换一个角度问，redo log定期刷到磁盘和change buffer定期merge操作（之后把脏页刷到磁盘），之间有什么联系？因为它们都会将最新的数据刷到磁盘上。如果有两条并行的线去刷最新

的数据到磁盘，则两者之间如何同步？

2018-12-05

作者回复

前面的过程有个理解不太对：

如果merge还没做，磁盘上page 2的数据是不会变的

2018-12-05

郭江伟 0

重复测试几次始终无法获取正确的统计信息，步骤如下：

```
mysql> show create table t;
| Table | Create Table |
| t | CREATE TABLE `t` (
`id` int(11) NOT NULL,
`a` int(11) DEFAULT NULL,
`b` int(11) DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `a` (`a`),
KEY `b` (`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 |
```

按照专栏脚本插入数据后：

```
mysql> select count(*) from t where a between 10001 and 20000;
| count(*) |
| 10000 |
```

执行计划rows理论值应该在10000附近，实际却是18900

```
mysql> explain select * from t where a between 10001 and 20000 ;
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filter
ed | Extra |
+----+------------+------+-----------+-----+-------------+-----+-------+-----+--
-----
| 1 | SIMPLE | t | NULL | ALL | a | NULL | NULL | NULL | 99722 | 18.95 | Using where |

mysql> explain select * from t force index(a) where a between 10001 and 20000 ;
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filter
ed | Extra |
+----+------------+------+-----------+------+-------------+-----+-------+-----+-
------
| 1 | SIMPLE | t | NULL | range | a | a | 5 | NULL | 18900 | 100.00 | Using index condition |
1 row in set, 1 warning (0.00 sec)

mysql> analyze table t;
| Table | Op | Msg_type | Msg_text |
| sysbench.t | analyze | status | OK |

mysql> explain select * from t where a between 10001 and 20000 ;
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filter
ed | Extra |
+----+------------+------+-----------+-----+-------------+-----+
| 1 | SIMPLE | t | NULL | ALL | a | NULL | NULL | NULL | 99722 | 18.95 | Using where |

mysql> show variables like '%stats_persistent%';
| innodb_stats_persistent | ON |
| innodb_stats_persistent_sample_pages | 20 |
```

执行 truncate table t;后重新插入数据依然是上面的情况

2018-12-05

作者回复

99722已经算是不错啦，挺接近的了...

2018-12-05

栋能 0

看到这篇文章我突然有了个新的MVCC疑惑，希望老师解惑下。谢谢。
我理解Delete操作是物理删除。如果有两个事务，事务A“读T表，再更新T表”的“更新T表”前事务B删除了T表，那更新还会成功吗？

2018-12-05

作者回复

删不掉，MDL锁住了

2018-12-05

kevin 0

老师你好。我用存储过程插入100000条数据特别慢，后来我set autocommit=0,每1000条才commit，这样就快了。我想不出来这是为什么，求解惑

2018-12-05

作者回复

Redo log 和 binlog刷盘次数少了，你把100000个事务变成了100个事务。

2018-12-05

KISS 0

老师你好，我把文章开头的实验按步骤做了。
执行了session A 和session B 的操作之后，MySQL并没有选错索引，用的还是`a`这个索引。
是因为MySQL版本问题吗？
我用的是5.7

```
mysql> select version();
+-------------------------+
| version() |
+-------------------------+
| 5.7.24-0ubuntu0.16.04.1
```

希望老师能解答一下：）

2018-12-05

作者回复

事务隔离级别是可重复读吗？

2018-12-05

萧若愚 0

请教个问题，如果一个表的字段名中包含表情符号比如 a😇，查询表的结果中字段名变为a? ，该表的字符集已设为 utf8_mb4。查看 information_schema 表中字段名也是a? ，发现 information_schema 表的默认字符集是 utf8。查询时若条件语句中用到该字段名时，用`a?`，`a😇`，以及 a😇 都可正确查询到结果，这是为什么？还有，如何查询出正确的字段名，即查询结果中为 a😇 而不是 a?。谢谢！

2018-12-05

作者回复

这个问题会在后面《乱码怎么办》中展开哈。

先简单回答一个，字段名的乱码是改不了了，返回结果的schema信息固定只能用utf8, utf8不认识表情符号...

2018-12-05

HuaMax 0

课后题我的理解是，session A在B之前启动，对于session B，A对数据的变更不可见，因此索引在统计时只能悲观的加上session A视图下的统计信息，并且B自己的事务内的存储过程插入了1

万行数据，由于没有提交，所以加上这1万行数据。这样算起来应该是2万，为啥是3万多，没想明白，望解惑

2018-12-05

天王 👍 0

请教一个问题：单表索引的分析相对简单一些。如果是多表关联查询，多个表的数据量很大，where条件里面有多个列，
怎么控制join和on的顺序，where条件里面字段的顺序，怎么建索引，让查询性能达到最优，有成型的规律吗，能否通过一些例子讲下？

2018-12-05

作者回复

Join的顺序MySQL 会帮你选的哈，

这个我得想下，放到篇文章展开

2018-12-05

看不穿 👍 0

老师，MySQL还有个不走索引的bug，感觉很诡异，生产上碰到多次了，就是update ... set ... where in select，走的索引不是我们期望的，必须转化为update ... join ... set ...。
参考文档：https://blog.csdn.net/defonds/article/details/46745143
以前在学oracle的时候，老师常说oracle的优化器是活的。可是到了mysql这，有时候只能呵呵了

2018-12-05

作者回复

嗯，MySQL 的优化器有时候就是不会做那些，我们觉得“本该做”的事情😓

2018-12-05

lianjie1229 👍 0

第一个选错索引的例子，在5.7.24版本上没复现出来，其他同学有复现出来的吗

2018-12-05

作者回复

Session A正常工作了吗？过程截图看下？

2018-12-05

dancer 👍 0

感谢老师的细致讲解，曾经面试过程中就遇到过类似的问索引区分度的问题~

2018-12-05

柚子 👍 0

我以前遇到过一次，字段a类型是int，where a= ‘1’ ,条件值是字符串类型，最后发现没有用到索引，改成数值1就用到了，只在那一台机子复现了，请问这是mysql有这样设计么，是有配置控制的么

2018-12-05

作者回复

是不是记反了… 应该是a 是varchar 类型，然后SQL是 a=1, 改成’ 1’ 就好了？
这个是跟类型转化规则有关，我们会在后面文章提到

2018-12-05

乐 👍 0

redolog在merge之前是不是写过一次，为什么merge后也会在写一次，是不是记录的东西不一样.is null会走索引么，之前碰到一个sql，使用explain的时候分析出is null是走索引的，结果实际运行的时候是扫描的全表

2018-12-05

作者回复

是的，第一次是记的”change buffer的改动”，第二次是记的“merge时数据页上的改动”

Explain和真正执行不同这种情况还挺少见的，可否贴下复现步骤😄

2018-12-05

Justin 👍 0

普通索引和mvcc有关系吗 会不会根据不同的tx id 有不同的结果呢？

2018-12-05

作者回复

会，普通索引也要满足事务隔离级别下的可见性规则

2018-12-05

天王 👍 0

请教几个问题 1 索引的基数 = 页面不同值的平均值数量*索引的页面数，什么是索引的页面数，索引是在页面上连续保存的？ 2 一个表有多个索引的情况，where条件里面有多个索引列的条件，每次都只能走一个索引吗，能都优化让走多个索引吗，不能同时走多个索引的原因是什么？

2018-12-05

作者回复

1. 一般不是，但是不影响这个逻辑哈
2. 有可能的, index_merge算法可以。就是你说的各自选出ID。这个过程需要画个图讲比较清楚，我放到答疑文章中。

2018-12-05

Justin 👍 0

删除行之后索引会立刻做出变化吗？是怎么变化的呢？

2018-12-05

作者回复

索引的内容（逻辑上）是马上变化的，否则我们再去查的时候就出错了。

但是物理上，可能会有Change buffer,
如果是这篇文章中说的统计信息，不会每次更新都重新统计，参考“1/M”那段

2018-12-05

一大只😴 👍 0

老师，联合索引的区分度咋看啊？

2018-12-05

钢铁加鲁鲁 👍 0

优化器选择索引的原则是什么呢？可以介绍下吗？

2018-12-05

钢铁加鲁鲁 👍 0

一楼的是不是用了联合查询，优化器会选择小表去驱动，如果如果排序字段是驱动表的字段会用索引的，不然只能指定驱动表了。可以执行explain看驱动表，迪哥0第一个就是驱动表

2018-12-05

不做码农 👍 0

遇到过这样一种情况，索引是建在一个日期类型的字段，范围查询时，要查的数量少的时候就走索引，数量大的时候就不走索引。

2018-12-05

作者回复

嗯嗯，这种不是 bug,这是优化器正常工作了😄

2018-12-05