44 | 答疑文章（三）：说一说这些好问题

2019-02-22 林晓斌

朗读：林晓斌

时长 16:20 大小 14.98M

这是我们专栏的最后一篇答疑文章，今天我们来说说一些好问题。

在我看来，能够帮我们扩展一个逻辑的边界的问题，就是好问题。因为通过解决这样的问题，能够加深我们对这个逻辑的理解，或者帮我们关联到另外一个知识点，进而可以帮助我们建立起自己的知识网络。

在工作中会问好问题，是一个很重要的能力。

经过这段时间的学习，从评论区的问题我可以感觉出来，紧跟课程学习的同学，对 SQL 语句执行性能的感觉越来越好了，提出的问题也越来越细致和精准了。

接下来，我们就一起看看同学们在评论区提到的这些好问题。在和你一起分析这些问题的时候，我会指出它们具体是在哪篇文章出现的。同时，在回答这些问题的过程中，我会假

设你已经掌握了这篇文章涉及的知识。当然，如果你印象模糊了，也可以跳回文章再复习一次。

join 的写法

在第 35 篇文章《join 语句怎么优化？》中，我在介绍 join 执行顺序的时候，用的都是 straight_join。@郭健 同学在文后提出了两个问题：

1. 如果用 left join 的话，左边的表一定是驱动表吗？
2. 如果两个表的 join 包含多个条件的等值匹配，是都要写到 on 里面呢，还是只把一个条件写到 on 里面，其他条件写到 where 部分？

为了同时回答这两个问题，我来构造两个表 a 和 b：

复制代码

```
create table a(f1 int, f2 int, index(f1))engine=innodb;
create table b(f1 int, f2 int)engine=innodb;
insert into a values(1,1),(2,2),(3,3),(4,4),(5,5),(6,6);
insert into b values(3,3),(4,4),(5,5),(6,6),(7,7),(8,8);
```

表 a 和 b 都有两个字段 f1 和 f2，不同的是表 a 的字段 f1 上有索引。然后，我往两个表中都插入了 6 条记录，其中在表 a 和 b 中同时存在的数据有 4 行。

@郭健 同学提到的第二个问题，其实就是下面这两种写法的区别：

复制代码

```
select * from a left join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q1*/
select * from a left join b on(a.f1=b.f1) where (a.f2=b.f2);/*Q2*/
```

我把这两条语句分别记为 Q1 和 Q2。

首先，需要说明的是，这两个 left join 语句的语义逻辑并不相同。我们先来看一下它们的执行结果。

```
mysql> select * from a left join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q1*/
+------+------+------+------+
| f1   | f2   | f1   | f2   |
+------+------+------+------+
|    3 |    3 |    3 |    3 |
|    4 |    4 |    4 |    4 |
|    5 |    5 |    5 |    5 |
|    6 |    6 |    6 |    6 |
|    1 |    1 | NULL | NULL |
|    2 |    2 | NULL | NULL |
+------+------+------+------+
6 rows in set (0.00 sec)

mysql> select * from a left join b on(a.f1=b.f1) where (a.f2=b.f2);/*Q2*/
+------+------+------+------+
| f1   | f2   | f1   | f2   |
+------+------+------+------+
|    3 |    3 |    3 |    3 |
|    4 |    4 |    4 |    4 |
|    5 |    5 |    5 |    5 |
|    6 |    6 |    6 |    6 |
+------+------+------+------+
4 rows in set (0.00 sec)
```

图 1 两个 join 的查询结果

可以看到：

- 语句 Q1 返回的数据集是 6 行，表 a 中即使没有满足匹配条件的记录，查询结果中也会返回一行，并将表 b 的各个字段值填成 NULL。
- 语句 Q2 返回的是 4 行。从逻辑上可以这么理解，最后的两行，由于表 b 中没有匹配的字段，结果集里面 b.f2 的值是空，不满足 where 部分的条件判断，因此不能作为结果集的一部分。

接下来，我们看看实际执行这两条语句时，MySQL 是怎么做的。

我们先一起看看语句 Q1 的 explain 结果：

```
mysql> explain select * from a left join b on(a.f1=b.f1) and (a.f1=1);/*Q1*/
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | a     | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    6 |   100.00 | NULL                                                |
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    6 |   100.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
```

图 2 Q1 的 explain 结果

可以看到，这个结果符合我们的预期：

- 驱动表是表 a，被驱动表是表 b；
- 由于表 b 的 f1 字段上没有索引，所以使用的是 Block Nexted Loop Join（简称 BNL）算法。

看到 BNL 算法，你就应该知道这条语句的执行流程其实是这样的：

1. 把表 a 的内容读入 join_buffer 中。因为是 select * ，所以字段 f1 和 f2 都被放入 join_buffer 了。
2. 顺序扫描表 b，对于每一行数据，判断 join 条件（也就是 a.f1=b.f1 and a.f2=b.f2) 是否满足，满足条件的记录, 作为结果集的一行返回。如果语句中有 where 子句，需要先判断 where 部分满足条件后，再返回。
3. 表 b 扫描完成后，对于没有被匹配的表 a 的行（在这个例子中就是 (1,1)、(2,2) 这两行），把剩余字段补上 NULL，再放入结果集中。

对应的流程图如下：

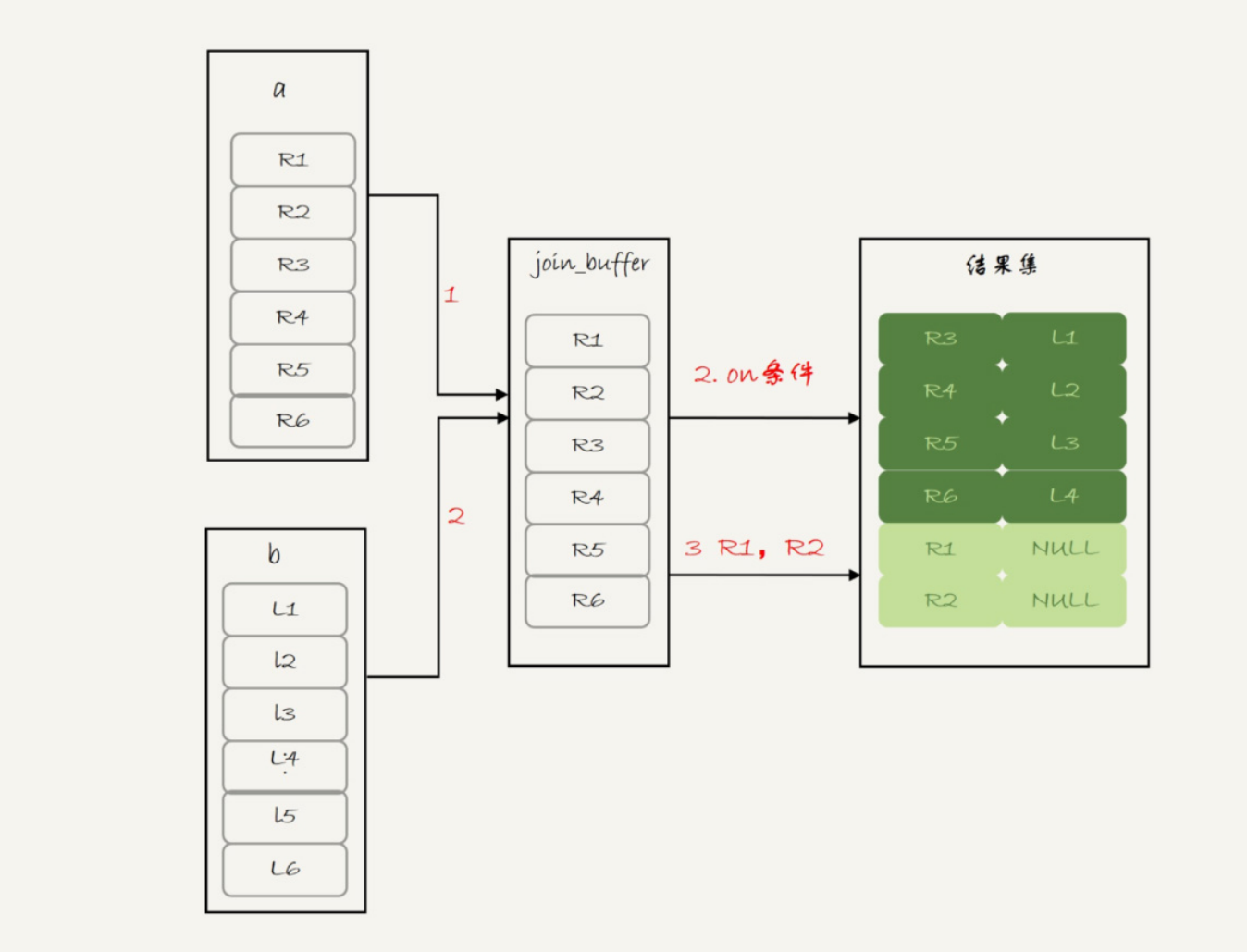

图 3 left join -BNL 算法

可以看到，这条语句确实是以表 a 为驱动表，而且从执行效果看，也和使用 straight_join 是一样的。

你可能会想，语句 Q2 的查询结果里面少了最后两行数据，是不是就是把上面流程中的步骤 3 去掉呢？我们还是先看一下语句 Q2 的 expain 结果吧。

```
mysql> explain select * from a left join b on(a.f1=b.f1) where (a.f2=b.f2); /*Q2*/
+----+-------------+-------+------------+------+---------------+------+---------+-----------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref       | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+-----------+------+----------+-------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL | NULL    | NULL      |    6 |   100.00 | Using where |
|  1 | SIMPLE      | a     | NULL       | ref  | f1            | f1   | 5       | test.b.f1 |    1 |    16.67 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+-----------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

图 4 Q2 的 explain 结果

这里先和你说一句题外话，专栏马上就结束了，我也和你一起根据 explain 结果“脑补”了很多次一条语句的执行流程了，所以我希望你已经具备了这个能力。今天，我们再一起分析一次 SQL 语句的 explain 结果。

可以看到，这条语句是以表 b 为驱动表的。而如果一条 join 语句的 Extra 字段什么都没写的话，就表示使用的是 Index Nested-Loop Join（简称 NLJ）算法。

因此，语句 Q2 的执行流程是这样的：顺序扫描表 b，每一行用 b.f1 到表 a 中去查，匹配到记录后判断 a.f2=b.f2 是否满足，满足条件的话就作为结果集的一部分返回。

那么，**为什么语句 Q1 和 Q2 这两个查询的执行流程会差距这么大呢？**其实，这是因为优化器基于 Q2 这个查询的语义做了优化。

为了理解这个问题，我需要再和你交代一个背景知识点：在 MySQL 里，NULL 跟任何值执行等值判断和不等值判断的结果，都是 NULL。这里包括， select NULL = NULL 的结果，也是返回 NULL。

因此，语句 Q2 里面 where a.f2=b.f2 就表示，查询结果里面不会包含 b.f2 是 NULL 的行，这样这个 left join 的语义就是“找到这两个表里面，f1、f2 对应相同的行。对于表 a 中存在，而表 b 中匹配不到的行，就放弃”。

这样，这条语句虽然用的是 left join，但是语义跟 join 是一致的。

因此，优化器就把这条语句的 left join 改写成了 join，然后因为表 a 的 f1 上有索引，就把表 b 作为驱动表，这样就可以用上 NLJ 算法。在执行 explain 之后，你再执行 show warnings，就能看到这个改写的结果，如图 5 所示。

```
mysql> show warnings;
+-------+------+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| Level | Code | Message                                                                                                                                                                                                                    |
+-------+------+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| Note  | 1003 | /* select#1 */ select `test`.`a`.`f1` AS `f1`,`test`.`a`.`f2` AS `f2`,`test`.`b`.`f1` AS `f1`,`test`.`b`.`f2` AS `f2` from `test`.`a` join `test`.`b` where ((`test`.`a`.`f1` = `test`.`b`.`f1`) and (`test`.`a`.`f2` = `test`.`b`.`f2`)) |
+-------+------+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

图 5 Q2 的改写结果

这个例子说明，即使我们在 SQL 语句中写成 left join，执行过程还是有可能不是从左到右连接的。也就是说，**使用 left join 时，左边的表不一定是驱动表。**

这样看来，**如果需要 left join 的语义，就不能把被驱动表的字段放在 where 条件里面做等值判断或不等值判断，必须都写在 on 里面。**那如果是 join 语句呢？

这时候，我们再看看这两条语句：

复制代码

```
select * from a join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q3*/
select * from a join b on(a.f1=b.f1) where (a.f2=b.f2);/*Q4*/
```

我们再使用一次看 explain 和 show warnings 的方法，看看优化器是怎么做的。

```
mysql> explain select * from a join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q3*/
+----+-------------+-------+------------+------+---------------+------+---------+----------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref      | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+----------+------+----------+-------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL | NULL    | NULL     |    6 |   100.00 | Using where |
|  1 | SIMPLE      | a     | NULL       | ref  | f1            | f1   | 5       | test.b.f1 |    1 |    16.67 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+----------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql> show warnings;
+-------+------+---------------------------------------------------------------
| Level | Code | Message
                                                                              |
+-------+------+---------------------------------------------------------------
| Note  | 1003 | /* select#1 */ select `test`.`a`.`f1` AS `f1`,`test`.`a`.`f2` AS `f2`,`test`.`b`.`f1` AS `f1`,`test`.`b`.`f2` AS `f2` from `test`.`a` join `test`.`b` where
 ((`test`.`a`.`f2` = `test`.`b`.`f2`) and (`test`.`a`.`f1` = `test`.`b`.`f1`)) |
+-------+------+---------------------------------------------------------------
1 row in set (0.00 sec)

mysql> explain select * from a join b on(a.f1=b.f1) where (a.f2=b.f2);/*Q4*/
+----+-------------+-------+------------+------+---------------+------+---------+----------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref      | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+----------+------+----------+-------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL | NULL    | NULL     |    6 |   100.00 | Using where |
|  1 | SIMPLE      | a     | NULL       | ref  | f1            | f1   | 5       | test.b.f1 |    1 |    16.67 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+----------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql> show warnings;
+-------+------+---------------------------------------------------------------
| Level | Code | Message
                                                                              |
+-------+------+---------------------------------------------------------------
| Note  | 1003 | /* select#1 */ select `test`.`a`.`f1` AS `f1`,`test`.`a`.`f2` AS `f2`,`test`.`b`.`f1` AS `f1`,`test`.`b`.`f2` AS `f2` from `test`.`a` join `test`.`b` where
 ((`test`.`a`.`f1` = `test`.`b`.`f1`) and (`test`.`a`.`f2` = `test`.`b`.`f2`)) |
+-------+------+---------------------------------------------------------------
```

图 6 join 语句改写

可以看到，这两条语句都被改写成：

复制代码

```
select * from a join b where (a.f1=b.f1) and (a.f2=b.f2);
```

执行计划自然也是一模一样的。

也就是说，在这种情况下，join 将判断条件是否全部放在 on 部分就没有区别了。

Simple Nested Loop Join 的性能问题

我们知道，join 语句使用不同的算法，对语句的性能影响会很大。在第 34 篇文章《到底可不可以使用 join？》的评论区中，@书策稠浊 和 @朝夕心 两位同学提了一个很不错的问题。

我们在文中说到，虽然 BNL 算法和 Simple Nested Loop Join 算法都是要判断 M*N 次（M 和 N 分别是 join 的两个表的行数），但是 Simple Nested Loop Join 算法的每轮判断都要走全表扫描，因此性能上 BNL 算法执行起来会快很多。

为了便于说明，我还是先为你简单描述一下这两个算法。

BNL 算法的执行逻辑是：

1. 首先，将驱动表的数据全部读入内存 join_buffer 中，这里 join_buffer 是无序数组；
2. 然后，顺序遍历被驱动表的所有行，每一行数据都跟 join_buffer 中的数据进行匹配，匹配成功则作为结果集的一部分返回。

Simple Nested Loop Join 算法的执行逻辑是：顺序取出驱动表中的每一行数据，到被驱动表去做全表扫描匹配，匹配成功则作为结果集的一部分返回。

这两位同学的疑问是，Simple Nested Loop Join 算法，其实也是把数据读到内存里，然后按照匹配条件进行判断，为什么性能差距会这么大呢？

解释这个问题，需要用到 MySQL 中索引结构和 Buffer Pool 的相关知识点：

1. 在对被驱动表做全表扫描的时候，如果数据没有在 Buffer Pool 中，就需要等待这部分数据从磁盘读入；
   从磁盘读入数据到内存中，会影响正常业务的 Buffer Pool 命中率，而且这个算法天然会对被驱动表的数据做多次访问，更容易将这些数据页放到 Buffer Pool 的头部（请参考第 35 篇文章中的相关内容）；
2. 即使被驱动表数据都在内存中，每次查找“下一个记录的操作”，都是类似指针操作。而 join_buffer 中是数组，遍历的成本更低。

所以说，BNL 算法的性能会更好。

distinct 和 group by 的性能

在第 37 篇文章《什么时候会使用内部临时表？》中，@老杨同志 提了一个好问题：如果只需要去重，不需要执行聚合函数，distinct 和 group by 哪种效率高一些呢？

我来展开一下他的问题：如果表 t 的字段 a 上没有索引，那么下面这两条语句：

复制代码

```
1 select a from t group by a order by null;
2 select distinct a from t;
```

的性能是不是相同的?

首先需要说明的是，这种 group by 的写法，并不是 SQL 标准的写法。标准的 group by 语句，是需要在 select 部分加一个聚合函数，比如：

```
1 select a,count(*) from t group by a order by null;
```

这条语句的逻辑是：按照字段 a 分组，计算每组的 a 出现的次数。在这个结果里，由于做的是聚合计算，相同的 a 只出现一次。

> 备注：这里你可以顺便复习一下第 37 篇文章中关于 group by 的相关内容。

没有了 count(*) 以后，也就是不再需要执行“计算总数”的逻辑时，第一条语句的逻辑就变成是：按照字段 a 做分组，相同的 a 的值只返回一行。而这就是 distinct 的语义，所以不需要执行聚合函数时，distinct 和 group by 这两条语句的语义和执行流程是相同的，因此执行性能也相同。

这两条语句的执行流程是下面这样的。

1. 创建一个临时表，临时表有一个字段 a，并且在这个字段 a 上创建一个唯一索引；
2. 遍历表 t，依次取数据插入临时表中：
   - 如果发现唯一键冲突，就跳过；
   - 否则插入成功；
3. 遍历完成后，将临时表作为结果集返回给客户端。

备库自增主键问题

除了性能问题，大家对细节的追问也很到位。在第 39 篇文章《自增主键为什么不是连续的？》评论区，@帽子掉了 同学问到：在 binlog_format=statement 时，语句 A 先获取 id=1，然后语句 B 获取 id=2；接着语句 B 提交，写 binlog，然后语句 A 再写 binlog。

这时候，如果 binlog 重放，是不是会发生语句 B 的 id 为 1，而语句 A 的 id 为 2 的不一致情况呢?

首先，这个问题默认了“自增 id 的生成顺序，和 binlog 的写入顺序可能是不同的”，这个理解是正确的。

其次，这个问题限定在 statement 格式下，也是对的。因为 row 格式的 binlog 就没有这个问题了，Write row event 里面直接写了每一行的所有字段的值。

而至于为什么不会发生不一致的情况，我们来看一下下面的这个例子。

复制代码

```
create table t(id int auto_increment primary key);
insert into t values(null);
```

```
BEGIN
/*!*/;
# at 486
# at 518
#190219 18:42:49 server id 1  end_log_pos 518 CRC32 0x6364946b  Intvar
SET INSERT_ID=1/*!*/;
#190219 18:42:49 server id 1  end_log_pos 618 CRC32 0xb6277773  Query   thread_id=4     exec_time=0     error_code=0
SET TIMESTAMP=1550572969/*!*/;
insert into t values(null)
```

图 7 insert 语句的 binlog

可以看到，在 insert 语句之前，还有一句 SET INSERT_ID=1。这条命令的意思是，这个线程里下一次需要用到自增值的时候，不论当前表的自增值是多少，固定用 1 这个值。

这个 SET INSERT_ID 语句是固定跟在 insert 语句之前的，比如 @帽子掉了同学提到的场景，主库上语句 A 的 id 是 1，语句 B 的 id 是 2，但是写入 binlog 的顺序先 B 后 A，那么 binlog 就变成：

复制代码

```
SET INSERT_ID=2;
语句 B;
SET INSERT_ID=1;
语句 A;
```

你看，在备库上语句 B 用到的 INSERT_ID 依然是 2，跟主库相同。

因此，即使两个 INSERT 语句在主备库的执行顺序不同，自增主键字段的值也不会不一致。

小结

今天这篇答疑文章，我选了 4 个好问题和你分享，并做了分析。在我看来，能够提出好问题，首先表示这些同学理解了我们文章的内容，进而又做了深入思考。有你们在认真的阅读和思考，对我来说是鼓励，也是动力。

说实话，短短的三篇答疑文章无法全部展开同学们在评论区留下的高质量问题，之后有的同学还会二刷，也会有新的同学加入，大家想到新的问题就请给我留言吧，我会继续关注评论区，和你在评论区交流。

老规矩，答疑文章也是要有课后思考题的。

在第 8 篇文章的评论区， @XD 同学提到一个问题：他查看了一下 innodb_trx，发现这个事务的 trx_id 是一个很大的数（281479535353408），而且似乎在同一个 session 中启动的会话得到的 trx_id 是保持不变的。当执行任何加写锁的语句后，trx_id 都会变成一个很小的数字（118378）。

你可以通过实验验证一下，然后分析看看，事务 id 的分配规则是什么，以及 MySQL 为什么要这么设计呢?

你可以把你的结论和分析写在留言区，我会在下一篇文章和你讨论这个问题。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

上期的问题是，怎么给分区表 t 创建自增主键。由于 MySQL 要求主键包含所有的分区字段，所以肯定是要创建联合主键的。

这时候就有两种可选：一种是 (ftime, id)，另一种是 (id, ftime)。

如果从利用率上来看，应该使用 (ftime, id) 这种模式。因为用 ftime 做分区 key，说明大多数语句都是包含 ftime 的，使用这种模式，可以利用前缀索引的规则，减少一个索引。

这时的建表语句是：

复制代码

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `ftime` datetime NOT NULL,
  `c` int(11) DEFAULT NULL,
  PRIMARY KEY (`ftime`,`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1
PARTITION BY RANGE (YEAR(ftime))
(PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = MyISAM,
 PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = MyISAM,
 PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = MyISAM,
 PARTITION p_others VALUES LESS THAN MAXVALUE ENGINE = MyISAM);
```

当然，我的建议是你要尽量使用 InnoDB 引擎。InnoDB 表要求至少有一个索引，以自增字段作为第一个字段，所以需要加一个 id 的单独索引。

复制代码

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `ftime` datetime NOT NULL,
  `c` int(11) DEFAULT NULL,
  PRIMARY KEY (`ftime`,`id`),
  KEY `id` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
PARTITION BY RANGE (YEAR(ftime))
(PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = InnoDB,
 PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = InnoDB,
 PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = InnoDB,
 PARTITION p_others VALUES LESS THAN MAXVALUE ENGINE = InnoDB);
```

当然把字段反过来，创建成：

复制代码

```
  PRIMARY KEY (`id`,`ftime`),
  KEY `id` (`ftime`)
```

也是可以的。

评论区留言点赞板：

> @夹心面包 、@郭江伟 同学提到了最后一种方案。

> @aliang 同学提了一个好问题，关于 open_files_limit 和 innodb_open_files 的关系，我在回复中做了说明，大家可以看一下。

> @万勇 提了一个好问题，实际上对于现在官方的版本，将字段加在中间还是最后，在性能上是没差别的。但是，我建议大家养成习惯（如果你是 DBA 就帮业务开发同学养成习惯），将字段加在最后面，因为这样还是比较方便操作的。这个问题，我也在评论的答复中做了说明，你可以看一下。

上一篇 43 | 要不要使用分区表?

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。