

## 44 | 答疑文章（三）：说一说这些好问题

2019-02-22 林晓斌

MySQL实战45讲

[进入课程 >](#)



讲述：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時候，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

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

```
1 select * from a left join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q1*/
2 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：

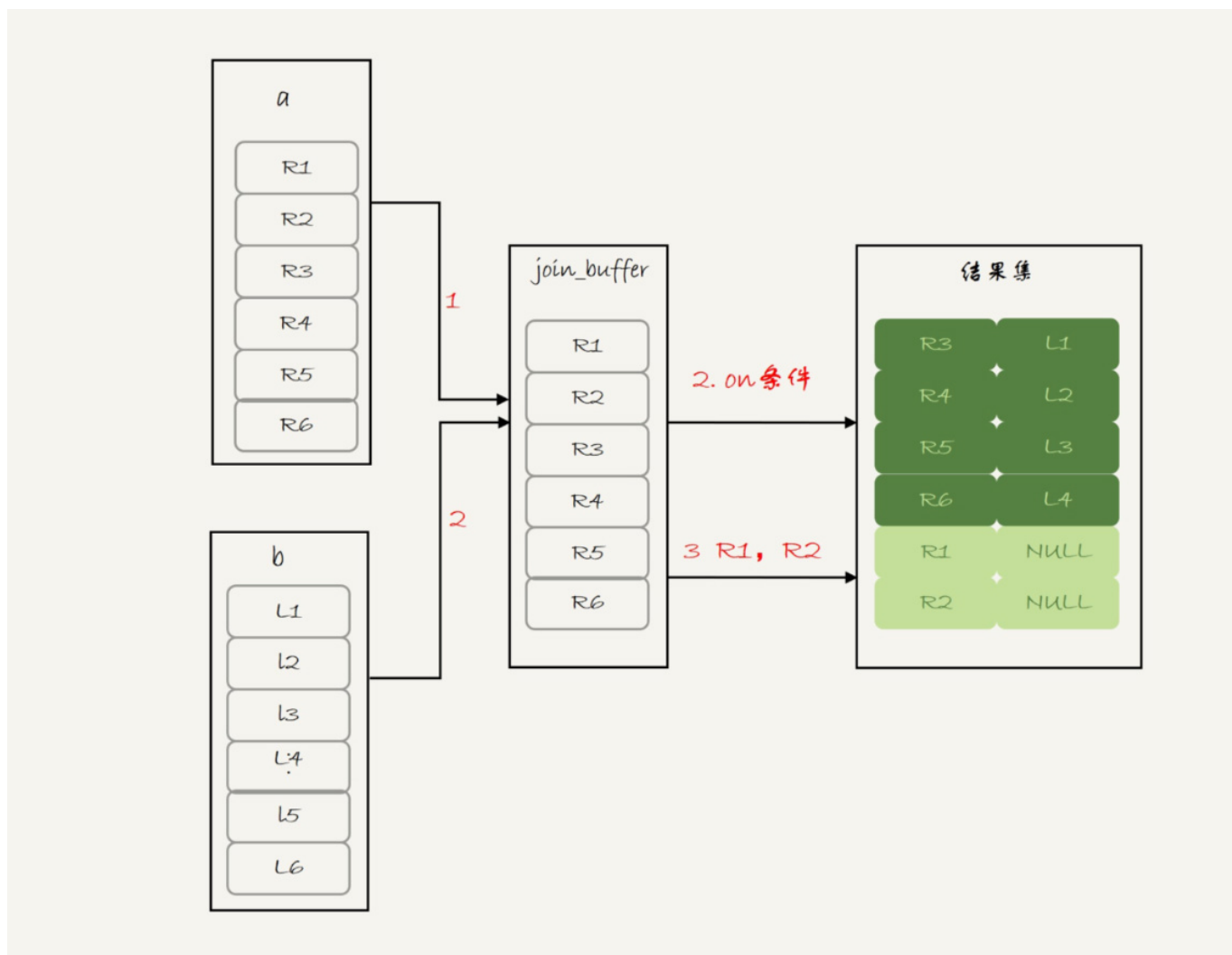


图 3 left join -BNL 算法

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 explain 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

```
1 select * from a join b on(a.f1=b.f1) and (a.f2=b.f2); /*Q3*/
2 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`)) |
+-----+-----+-----+
1 row in set (0.00 sec)
```

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

```
1 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題。

我们在文中说到，虽然 BNL 算法和 Simple Nested Loop Join 算法都是要判断  $M \times N$  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 select 部分加一个聚合函数，比如：

 复制代码

```
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


```
1 create table t(id int auto_increment primary key);
2 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

```
1 SET INSERT_ID=2;  
2 语句 B;  
3 SET INSERT_ID=1;  
4 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

```
1 CREATE TABLE `t` (  
2   `id` int(11) NOT NULL AUTO_INCREMENT,  
3   `ftime` datetime NOT NULL,  
4   `c` int(11) DEFAULT NULL,  
5   PRIMARY KEY (`ftime`,`id`)  
6 ) ENGINE=MyISAM DEFAULT CHARSET=latin1  
7 PARTITION BY RANGE (YEAR(ftime))  
8 (PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = MyISAM,  
9  PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = MyISAM,  
10 PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = MyISAM,  
11 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

```
1 CREATE TABLE `t` (  
2   `id` int(11) NOT NULL AUTO_INCREMENT,  
3   `ftime` datetime NOT NULL,
```

```
4  `c` int(11) DEFAULT NULL,  
5  PRIMARY KEY (`ftime`,`id`),  
6  KEY `id` (`id`)  
7 ) ENGINE=InnoDB DEFAULT CHARSET=latin1  
8 PARTITION BY RANGE (YEAR(ftime))  
9 (PARTITION p_2017 VALUES LESS THAN (2017) ENGINE = InnoDB,  
10 PARTITION p_2018 VALUES LESS THAN (2018) ENGINE = InnoDB,  
11 PARTITION p_2019 VALUES LESS THAN (2019) ENGINE = InnoDB,  
12 PARTITION p_others VALUES LESS THAN MAXVALUE ENGINE = InnoDB);
```

当然把字段反过来，创建成：

 复制代码

```
1  PRIMARY KEY (`id`,`ftime`),  
2  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。

# 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



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 43 | 要不要使用分区表？

下一篇 45 | 自增id用完怎么办？

## 精选留言 (26)

写留言



永恒记忆

2019-02-25

2

老师，看评论包括您的回复说 “left join 后加上 where 的话，肯定会被优化器优化成 join where 的形式，那是否下次写 left join ..where 的时候，不如直接写成 join .. where”，这个也是分情况的吧比如还是文章中的2张表，select \* from a left join b on(a.f1=b.f1) where (a.f2=2);/\*Q5\*/和select \* from a join b on(a.f1=b.f1) where (a.f2=2);/\*Q6\*/ 这个left join和join的语意和返回结果都不一样，怎么能直接写成join呢...

展开

作者回复: 嗯 我的意思是，如果where条件里面，用到了b.f2的判断，干脆就直接写成join，不需要left join了



如果业务逻辑需要left join，就要把条件都放到on里面

业务逻辑正确性还是优先的



宝玉

2019-02-25

2

老师，BNL算法，如果where条件中有驱动表的过滤条件，也不会在join时候全部载入内存吧？

作者回复: 对，驱动表先过滤，然后进join buffer



白永伟

2019-02-22

2

老师，关于备库自增id我有一个问题。既然binlog不管是statement模式还是row模式，里面的insert语句跟着的自增id都是固定的。那假如发生主备切换，备库变成主库后，客户端往新主库里插入数据时，自增id的起始值是多少，有没有可能跟已有的记录id冲突？尤其是备库还没有处理完同步过来的binlog就开始接受客户端请求时。如果要求备库必须处理完binlog才能接受客户端请求，那么怎么保证主备切换的过程中，不影响用户使用。谢谢。

展开

作者回复: “自增id的起始值是多少，有没有可能跟已有的记录id冲突？”

如果没有主备延迟就不会出现；

“尤其是备库还没有处理完同步过来的binlog就开始接受客户端请求时。”，对，这种情况就会。

“如果要求备库必须处理完binlog才能接受客户端请求，那么怎么保证主备切换的过程中，不影响用户使用” 一般都是有这个要求的。要尽量减少影响，就是控制主备延迟。



还一棵树

2019-02-26

1

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

文章中的流程是写错了？还是我理解的有问题

1、如果是a表数据放入join buffer，根据b的每一条记录去判断是否在a中 如果在则保留记录

这个更像是b left join a。 而不是a left join b...

展开 ∨

作者回复:

“如果按照这个流程，比如a里面有2行重复的数据，如果拿b的数据在a中判断，存在则保留，那结果集只有一条数据，”

不会呀，你看它是这样的：

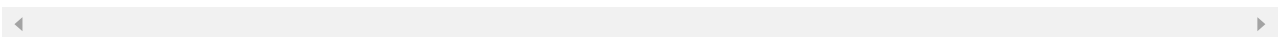
假设join buffer中有两个行1

然后被驱动表取出一个1，

跟join buffer中第一个1比较，发现满足条件，放到结果集；

跟join buffer中第二个1比较，发现满足条件，放到结果集；

是得到两行的



千木

2019-02-23

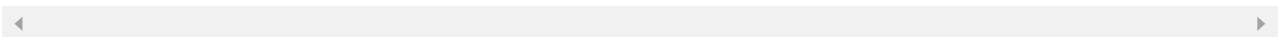
👍 1

老师您好，join使用join\_buffer和内存区别那个问题的第一点解释我还是有些纳闷，你说由于从磁盘拿数据到内存里面会导致等等的性能问题我能够理解，但是说即使使用nbl算法也会涉及到从磁盘拿数据到内存吧，所以这点导致两种算法执行差异貌似不太合理，您觉得呢？

展开 ∨

作者回复: BNL算法拿的数据是确定的只会拿一次（遍历一遍）

而simple nested loop join是会遍历多次的



400磅

2019-05-17

👍

老师你好，我想咨询一下我今天遇到的诡异的 Table " doesn't exist 问题

问题描述：

在2019年5月16日下午我在我司开发环境的数据库 (192.168.1.81:3306) 上创建了一张表...  
展开 ▾



**大宝**

2019-04-06



老师你好，看了你回复 @彼得.林 时说，“在线服务最好不要让索引树超过4层”；那么有没有什么好的方法或者工具可以查看当前表的层数吗？

展开 ▾

作者回复: innblock 可以了解下 😊



**Invictus\_C...**

2019-03-26



自增id还有一个bug。就是replace into的时候，假如唯一索引有冲突，主库传过来的binlog里是update，但是在主库操作的时候是先insert，这样主从的自增id那个值就会不一样。这时候主从切换也会有冲突的问题。不过这种情况发生的概率不大，除非一直replace或者刚出现一个replace数据库就主从切换了。

展开 ▾



**不惑ing**

2019-03-23



您好，我看留言区@龙文 问了个问题

1.rc隔离级别下对唯一键的insert也会加next-key lock吗？

您的回答是：会

我的问题：我记得前面哪篇文章里提到rc隔离级别是没有gap lock的么，我也网上用百...

展开 ▾



**彼得.林**

2019-03-16



不知道老师还会不会回复问题：

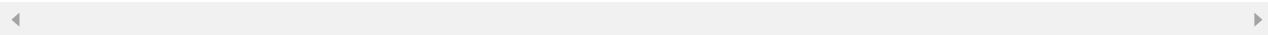
b+树索引结构的层次和表数据量的关系是怎么样的？也就是说15万的数据量是三层结构？达到多少数据量是四层次？

一般在线服务中一个表的数据量一般多大合适？

展开 ▾

作者回复: 你可以这么理解, N层放不下的时候, 就增加一层来放。

这个行为是由页分裂触发的  
在线服务最好不要让索引树超过4层



Chris

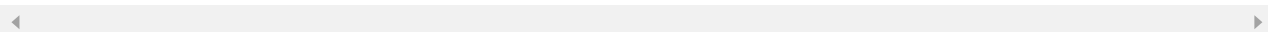
2019-03-15



这两天在线上遇到一个比较诡异的事情, 突然有几分钟连不上MySQL, 通过error日志和监控的processlist显示, MySQL把很多链接都kill掉了, 但处于sleep状态和show status的语句没有kill, 看监控的资源使用情况不是很高, 只是innodb rows read指标特别高, 现在完全是没头绪了

展开 ▾

作者回复: 看看是不是有什么外部工具在工作



长杰

2019-03-02

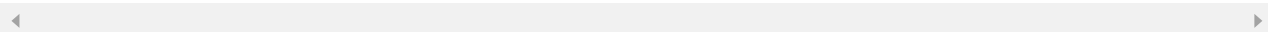


```
select * from a left join b on(a.f1=b.f1) where (a.f2=b.f2);
```

老师, 这个语句是否可以理解为:先按照on后面的条件关联, 获取结果集, 然后对结果集用where条件进行二次过滤?

展开 ▾

作者回复: 要看索引哈  
如果b上的索引只有f1 是的,  
如果b上的索引是(f1,f2), 就两个一起关联了  
【咱们文中说了, 这个语句会被转成普通join哦】



长杰

2019-03-02



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。

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...  
展开 ▾

作者回复: 不是，如果有where,并且where里面有用到b.f1或b.f2, 那就要求结果集里面没有这些null的行。

就是说where  $a.f2=b.f2$ 的意思是  
Where ( $a.f2$  is not null) and ( $b.f2$  is not null) and ( $a.f2 = b.f2$ )



梦康

2019-02-25



🙄留言的人太多，辛苦老实答疑了。虽然我的问题没能被翻牌子

作者回复: 不好意思，确实你的问题比较难一些

最近在做收尾的工作，后面一定会把问题都清理掉的哈。

你的问题质量高，是我喜欢回答的问题类型 😊



PYH

2019-02-24



你好 我想问一下mysql能实现oracle的拉链表么。如果能前提条件是什么？



龙文

2019-02-24



明白了 谢谢老师！

展开 ▾

作者回复: 🙏



滔滔

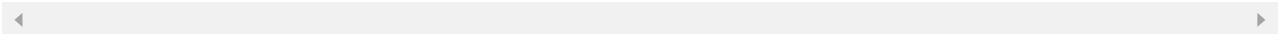
2019-02-24



老师您好，想请问下在innodb引擎rr隔离级别下，单独的一条update语句是不是默认就是一个事务(在执行update前不输入begin)，而单独的一条select语句是不是不会开启一个事务，哪怕是"当前读"也不会开启一个事务，更进一步，是不是对表的增删改操作默认都会开启一个事务？ 😊

展开 ∨

作者回复: 1. 单独一个update，会启动一个事务  
2. 单独一个select，也会启动一个事务  
3. innodb表，增删改查都会启动一个事务



发条橙子 ...

2019-02-23

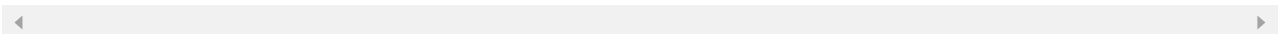


啧啧，原来我写的 left join 一直都不是标准的，每次后面都会加上 where，还一直以为左面是驱动表。既然实际上 left join 后加上 where 的话，肯定会被优化器优化成 join where 的形式，那是否下次写 left join ..where 的时候，不如直接写成 join .. where，省去优化器自己去优化，这样是不是稍稍快些

展开 ∨

作者回复: 是的

如果原来就有where，说明原来其实也不用left join 😊



龙文

2019-02-23



老师你好，我在第21讲求助了一个死锁问题，当时你回复说后面会解答，不过我浏览了下后续文章没找到解答，所以再次求助下。ps:用的阿里云的rds,提了工单没效果啊

作者回复: 有的，你看一下第40篇 “insert 唯一键冲突” 这一段

ps:我已经离开阿里云挺久的了 😊...

展开 ∨

作者回复: 1. 会  
2. 你这里  
session 1 成功加锁一个record lock ;



session 2执行的是一个select 语句，而且a=1 and b=1就只锁一行（a，b上有联合唯一索引），这里就是要申请一个记录行锁(but not gap waiting)。  
这里虽然没有加锁成功，但是已经加入了锁队列（只是这个锁是处于等待状态）  
---这时候队列里面有两个锁对象了

然后session 1 再insert失败的时候，就要加next-key lock，（注意这个锁对象跟第一个锁对象不同）。

然后死锁检测看到，2号锁在等1号锁；3号要等2号，而3和1又是同一个session，就认为是死锁了。



龙文

2019-02-23



老师你好，我在第21讲求助了一个死锁问题，当时你回复说后面会解答，不过我浏览了下后续文章没找到解答，所以再次求助下。ps:用的阿里云的rds,提了工单没效果啊

作者回复: 有的，你看一下第40篇 “insert 唯一键冲突” 这一段

ps:我已经离开阿里云挺久的了 😊