44 | 答疑文章（三）：说一说这些好问题

**Join的写法**

在第35篇《join语句怎么优化》中，介绍join执行顺序，用的都是straight\_join两个问题

1. 如果left jion 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,还是只要把一个条件写到on里面，其他条件写到where部分？

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，往表中都插入了6条记录，其中表a和b中同时存在的数据有4行。

就是下面的这两种写法的区别

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\*/

首先，需说明的是，这两个left join语句的语义逻辑并不相同，先看执行结果

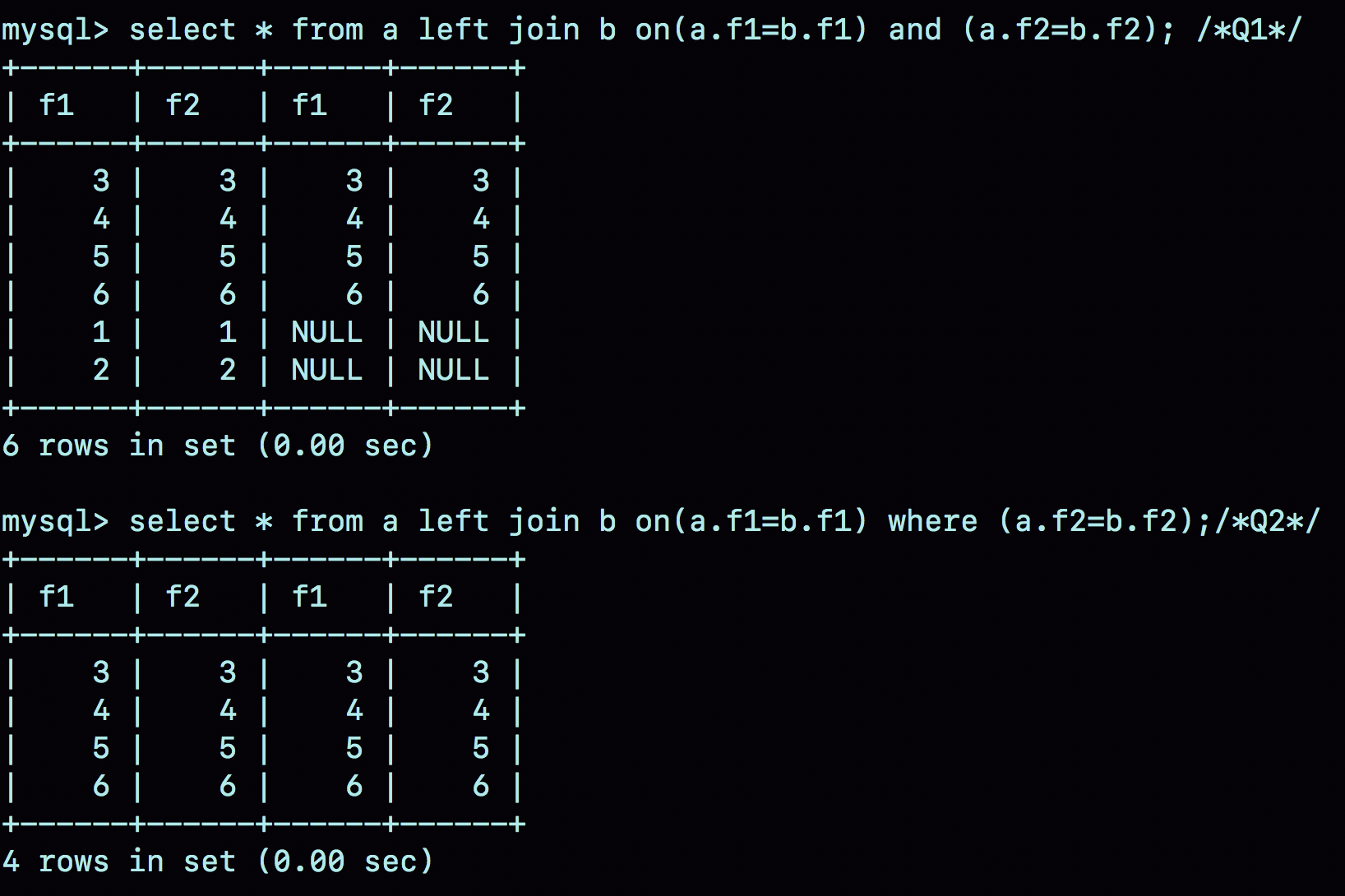


图 1 两个join的查询结果

1. 语句Q1返回的数据集是6行，表a中即使没有满足条件的的记录，查询结果中也会返回一行，并将表b的各个字段值填成NULL
2. 语句Q2返回的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

看下Q1的explain结果

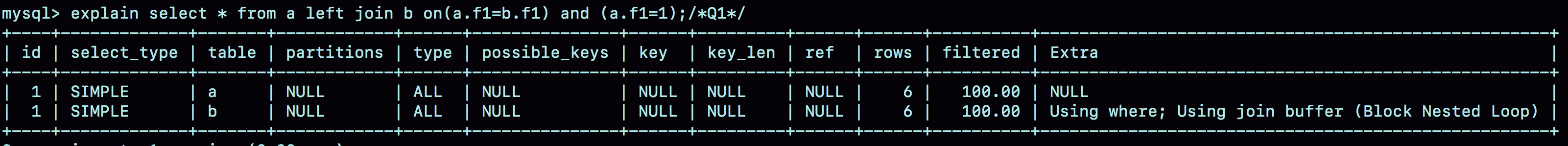


图2 Q1的expalin结果

结果符合预期

·驱动表示表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，执行流程是这样的

1. 把表a的内容读入join\_buffer中，因为select \* ,所以字段f1和f2都被放入join\_buffer
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.f1=1））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。

对应流程图如下

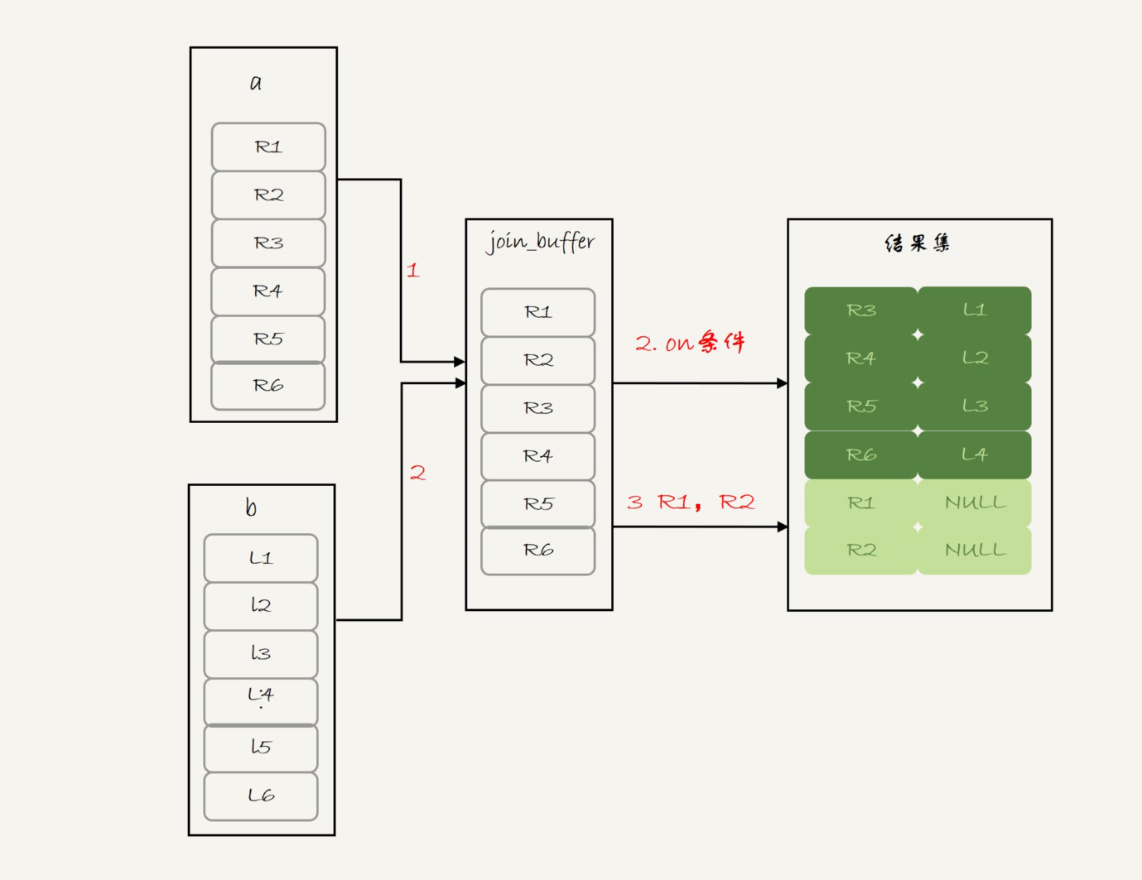


图3 left join -BNL算法

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。

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？看下去Q2的explain 结果

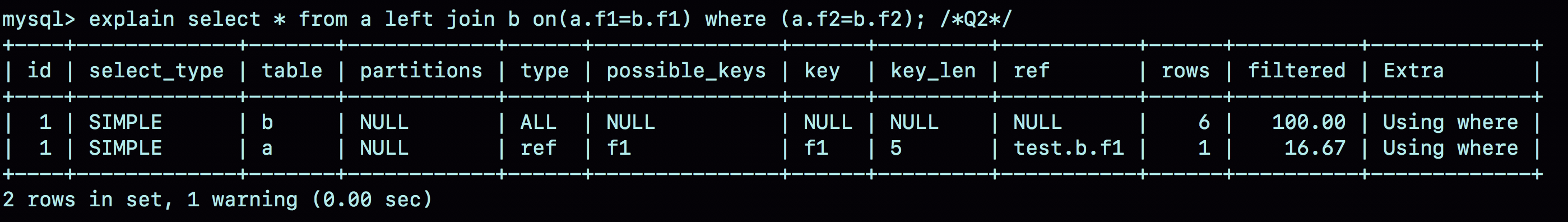


图 4 Q2的explain结果

这条语句是以表b为驱动表的。而如果一条join语句的Extra字段什么都没有写，就表示使用的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。

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

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匹配不到的行，就放弃”

这样，这条语句虽然用的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，这样就可以用上了NLJ算法，在执行explain之后，在执行show warnings，就能看到这个改写的效果，如图5 所示

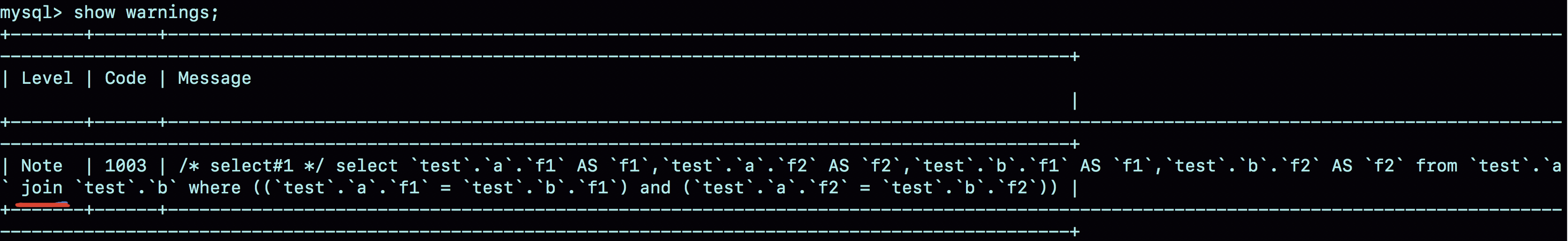


图 5 Q2的改写效果

即使在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**

**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**，如果是join语句呢？

此时，看着两条语句

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\*/

使用explain和show warnings看看优化器是怎么做的

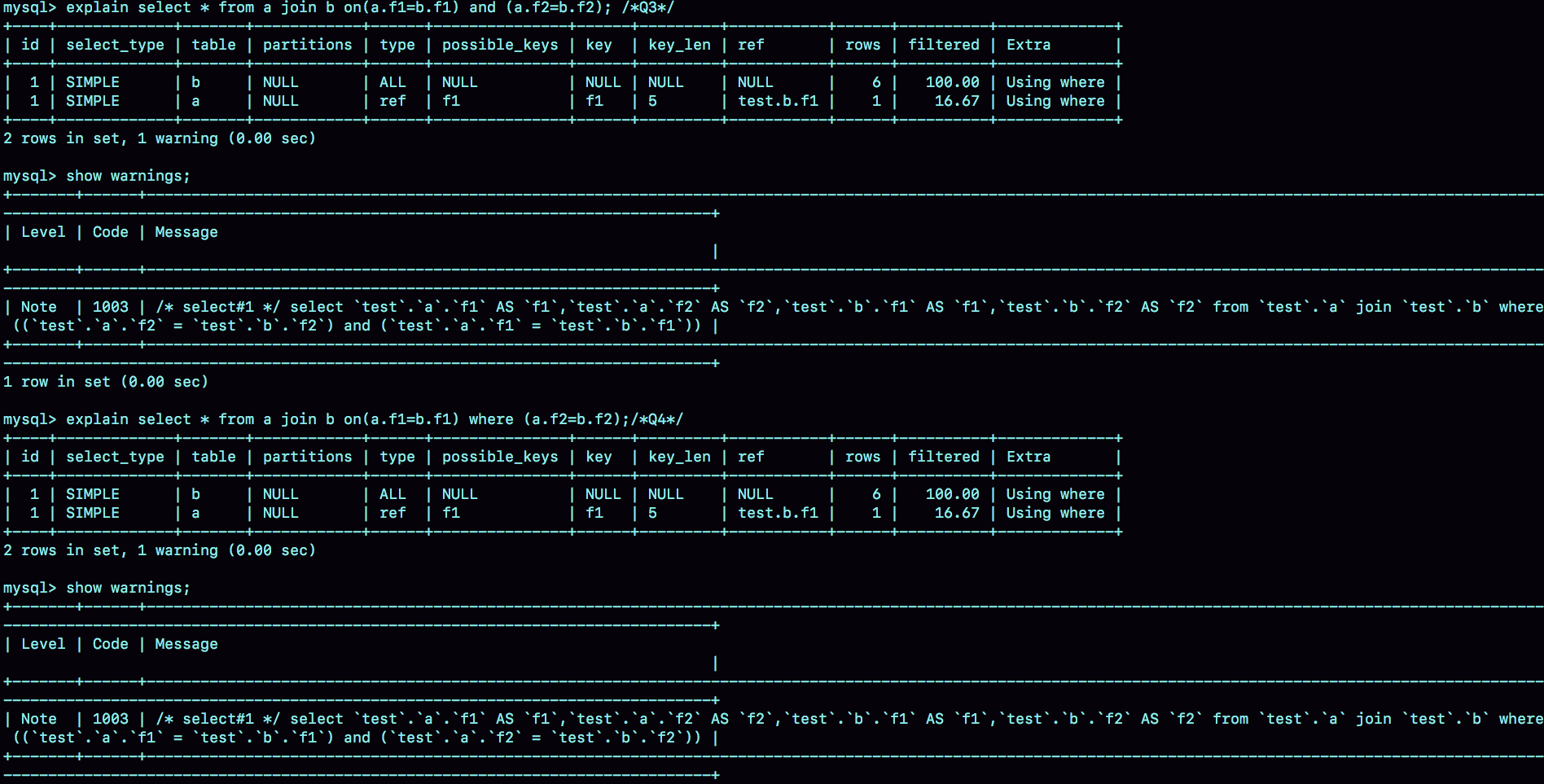


图 6 join语句改写

可以看到，这两条语句都被改些成

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**

join语句使用不同的算法，对语句的性能影响会很大，第34篇 评论区的问题。

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分别是两个表的行数），但是Simple Nested Loop Join算法的每轮判断都需要走全表扫描，因此性能上BNL算法执行起来会更快

**BNL算法的执行逻辑**

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。

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时，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，更容易将这些数据也放到Buffer Pool的头部（参考第35篇）

1. 即使被驱动表数据都在内存中，每次查找“下一个记录的操纵”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算法的性能更好。

distinct和group by的性能

在第37篇评论区一个好问题：如果只需要去重，不需要时执行聚合函数，distinct和group by那种效率高一些？

如果表t的字段a上没有索引，那么下面这两条语句

select a from t group by a order by null;

select distinct a from t;

的性能是不是相同？

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

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，在这个结果里面，由于做的是聚合计算相同的a只能出现一次

备注：顺便复习第37篇关于group by的相关内容

没有了count(\*)以后，也就是不需要执行“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。二者就是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

1. 创建一个临时表，临时表有个字段a,并且在这个字段a上创建一个唯一索引
2. 遍历表t，依次去数据插入临时表中；

·如果发现唯一键冲突，就跳过

·否则插入成功

1. 遍历完成后，降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，在第39篇评论区，问题：在binlog\_format=statement时，语句A先获取id=1，然后语句B取id=2;接着语句B提交，写binlog，然后语句A再写binlog，此时，如果binlog重发，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statemnet格式下，也是对的，因为row格式的binlog就没有这个问题，Write row event里面直接写了每一行的所有字段的值。

至于为什么不会发生不一致的情况，看卡下面的例子

create table t(id int auto\_increment primary key);

insert into t values(null);

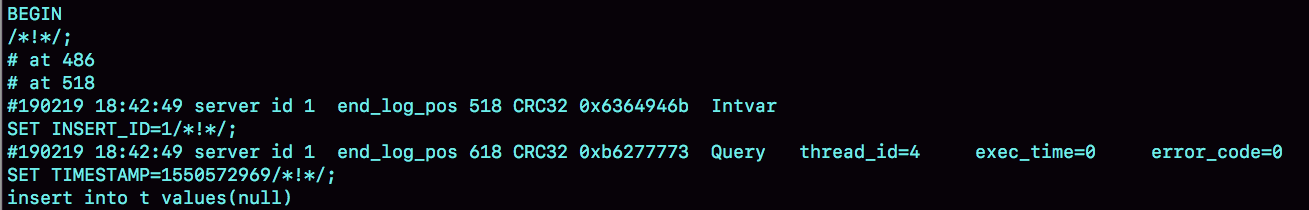


图 7 insert语句的binlog

可以看到，在insert与之前，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的自增值时，不论当前的自增值是多少，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，比如问题的提出，主库上的语句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就变成先B后A,那么binlog就变成

SET INSERT\_ID=2;

语句B

SET INSERT\_ID=1

语句A

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语句在朱备库的执行顺序不同，自增主键字段的值也不会不一致

**思考题**

在第8篇评论区，提到问题；查看innodb\_trx,发现这个事务的trx\_id是一个很大的数（281479535353408）,而且似乎在用一个session中启动的会话得到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）

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？