35|join语句怎么优化

BNL算法在大表join的时候性能就差多了，比较次数等于两个表参与join行数的乘积，很消耗CPU资源

创建表t1，t2来展开今天的话题

create table t1(id int primary key, a int, b int, index(a));

create table t2 like t1;

drop procedure idata;

delimiter ;;

create procedure idata()

begin

declare i int;

set i=1;

while(i<=1000)do

insert into t1 values(i, 1001-i, i);

set i=i+1;

end while;

set i=1;

while(i<=1000000)do

insert into t2 values(i, i, i);

set i=i+1;

end while;

end;;

delimiter ;

call idata();

为了便于量化说明，表t1,插入1000行数据，每一行的a=1001-id的值，也就是说，表t1字段a是逆序的，同时，表t2插入100万行数据

**Mutil-Range Read优化**

这个优化的主要目的是尽量使用顺序读盘。

执行下面的语句

Select \* from t1 where a >=1 and a <= 100

主键索引是一颗B+树，在这颗树上，每次只能根据一个主键id查到一行数据，因此，回表肯定是一行行搜索主键索引的，基本流程如图1

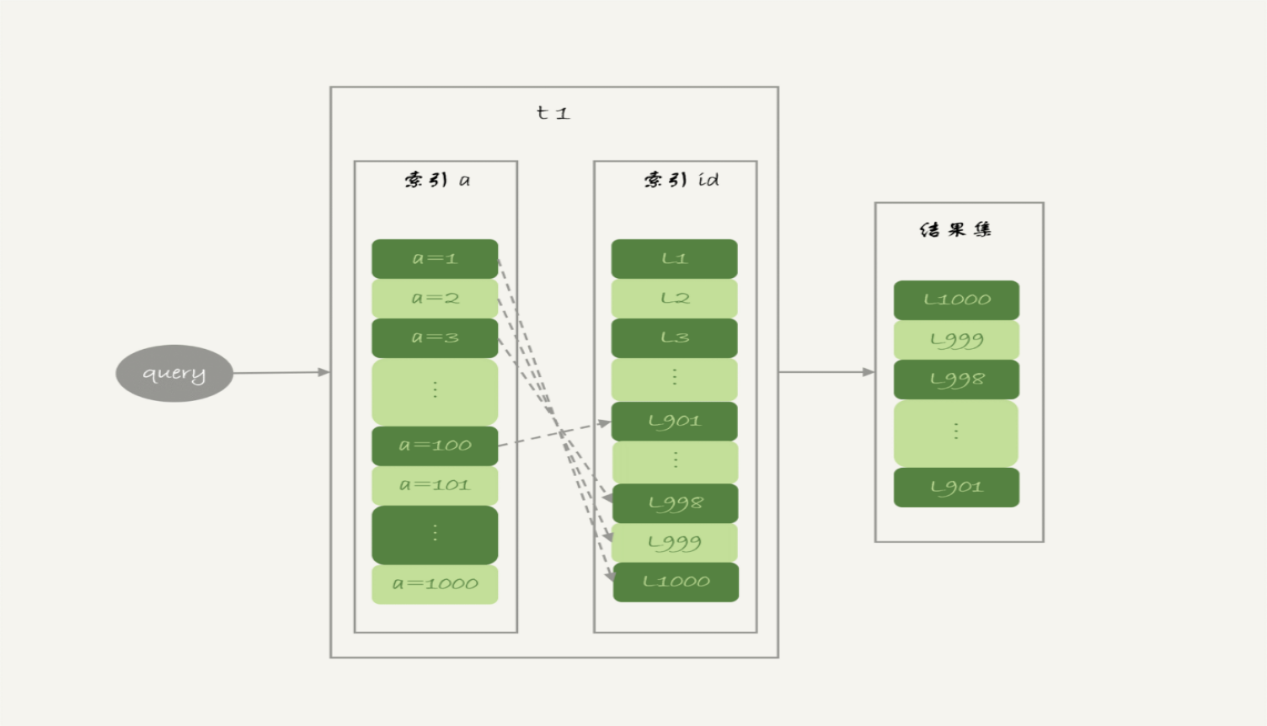


图 1 基本回表流程

如果随着a的值递增顺序查找的话，id的值就变成了随机的，那么就会出现随机访问，性能相对较差，虽然“按行查”这个机制不能改，但是调整查询的顺序，还是能够加速的

**因为大多数的数据都是按照主键递增顺序插入得到的，所以人为，如果按照主键的递增顺序查找的话，对磁盘的读比较接近顺序读，能够提升读性能。**

这是MRR优化的设计思路，此时，语句的执行流程如下

1. 根据索引a,定位到满足条件的记录，将id值放入read\_rnd\_buffer中
2. 将read\_rnd\_buffer中的id进行递增排序
3. 排序后的id数组，依次到主键id索引查找记录，并作为结果返回

这里，read\_rnd\_buffer的大小是有read\_rnd\_buffer\_size参数控制的，如果步骤1中，read\_rnd\_buffer放满了，就会先执行步骤2和3,然后清空read\_rnd\_buffer,之后继续找索引，并继续循环。

另外需要说明的是，如果想要稳定使用MRR优化，需要设置set optimizer\_switch=”mrr\_cost\_bsed=off”(官方文档的说法，是现在的优化器策略，判断消耗的时候，会更倾向于不使用MRR,把mrr\_cost\_based设置为off,就是固定使用MRR了)

下面两幅图就是使用了MRR优化后执行流程和explain结果

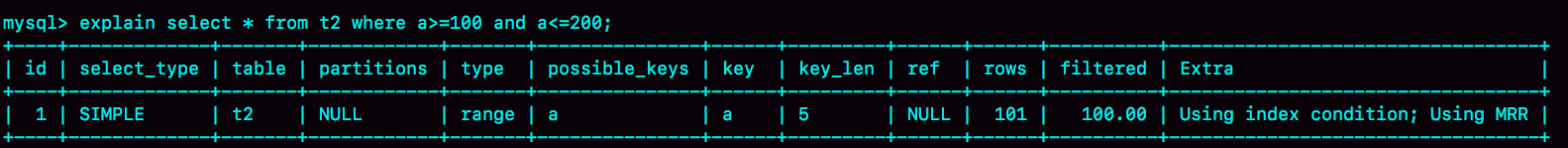


图2 执行流程的explain 结果

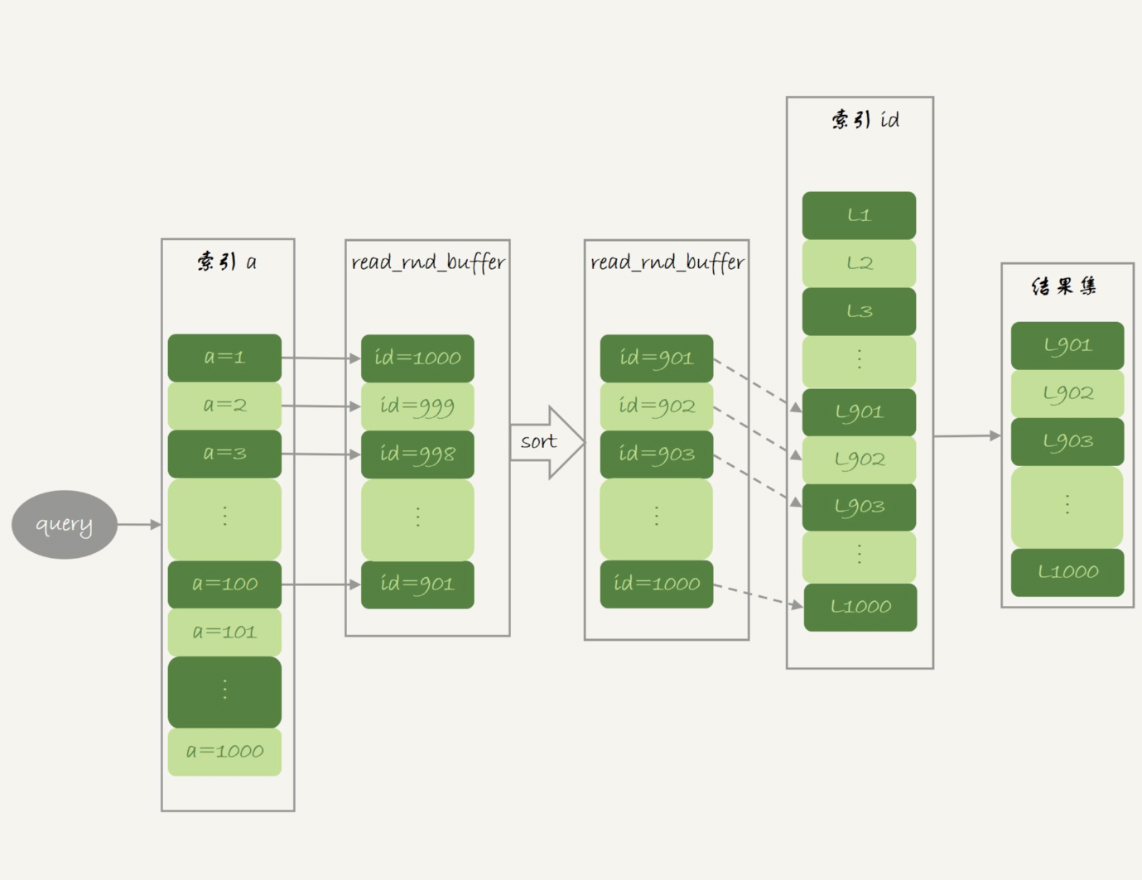


图3 MRR 执行流程

从图2中的explain结果中，可以看到Extra字段多了Using MRR,表示用上了MRR优化，而且，由于在read\_rnd\_buffer中按照id做了排序，所以最后得到的结果集也是按照主键id递增顺序的，也就是与图1结果集中行的顺序相反。

**MRR能够提升性能的核心**在于，这条查询语句在索引a上做的是一个范围查找（也就是说，这是一个多值查询），可以得到足够多的主键id,这样通过排序后，在去主键索引查数据，才能体现出“顺序性”的优势。

**Batched Key Access**

理解了MRR,MySQL5.6引入了Batched Key Access(BKA)算法，BKA,其实是对NLJ算法的优化

回忆NLJ的流程

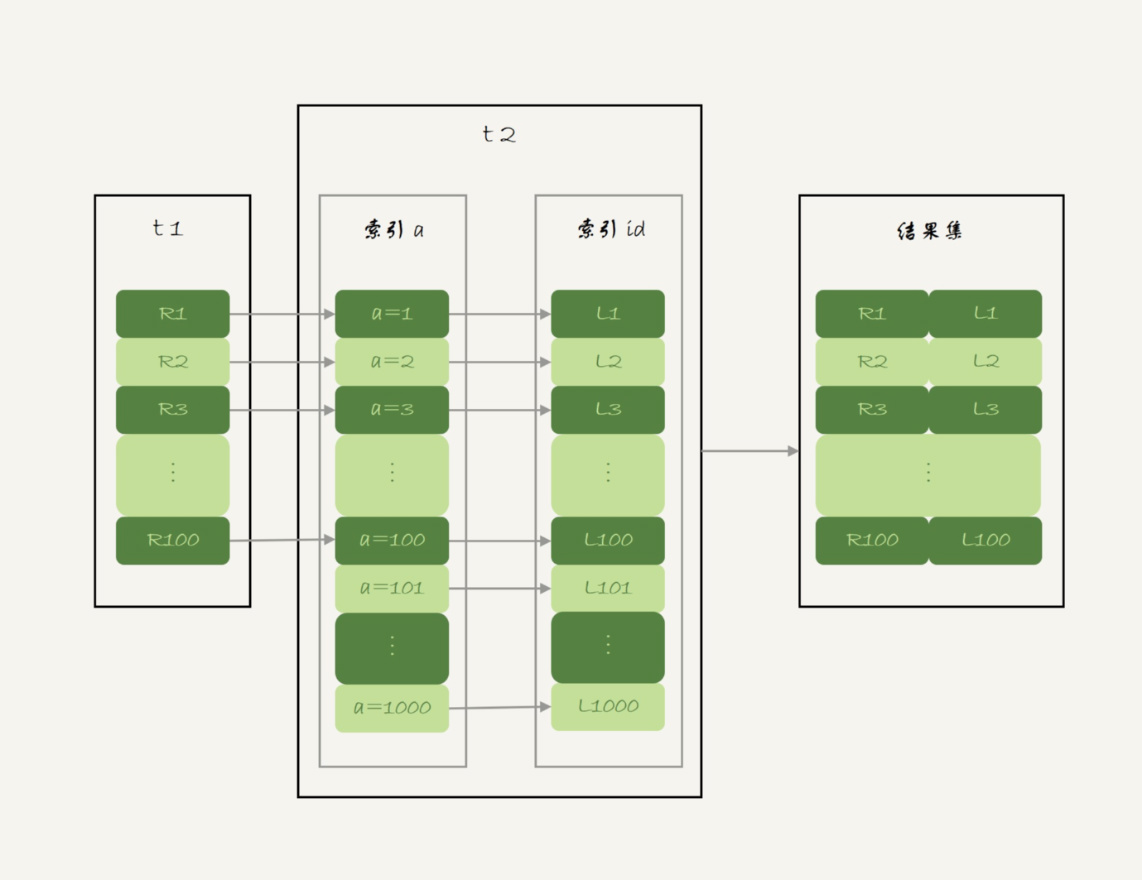


图 4 Index Nested-Loop Join 流程图

NLJ算法执行的逻辑是：从驱动表t1,一行行的取出a的值，在到驱动表t2去做join,对于t2表来说，每次都是匹配一个值，此时，MRR的优势就用不上

怎么才能一次性的多传些值给表t2呢？方法是，从表t1理一次性的多拿些行出来，一起传给表t2.

既然如此，把表t1的数据取出一部分，先放到一个临时内存，这个临时内存不是别人，就是join\_buffer.

Join\_buffer在BNL算法里的作用，是暂存驱动表的数据，但是在NLJ算法里并没有用，刚好就可以复用join\_buffer到BKA中

如图5 ，是上面NLJ算法优化后的BKA算法的流程

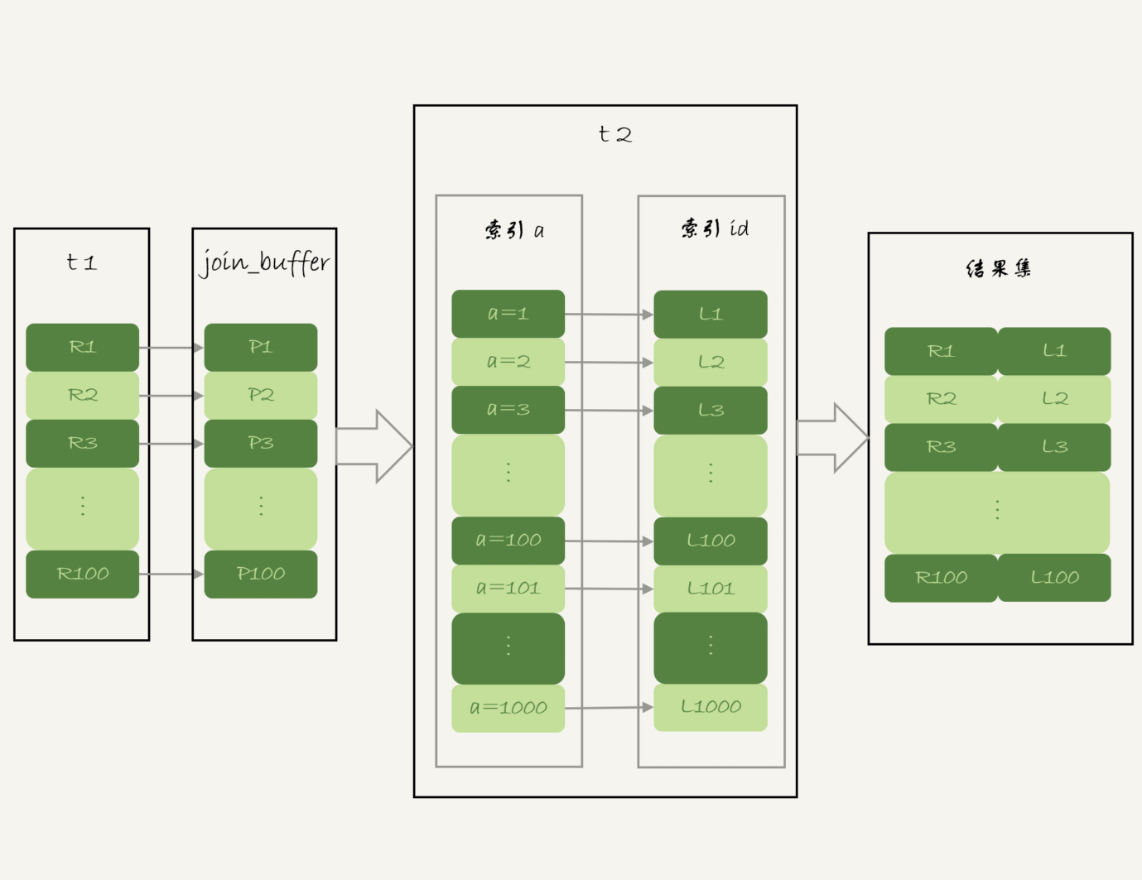


图 5 Batched Key Access流程

图中，在join\_buffer中放入的数据P1-P100,表示的是只会查询需要的字段，如果join\_buffer放不下P1-P100的所有数据，就会把这100行数据分成多段执行上图的流程。

那么，这个BKA算法到底怎么起作用？

如果使用BKA优化算法，需要在执行SQL语句之前，先设置

set optimizer\_switch='mrr=on,mrr\_cost\_based=off,batched\_key\_access=on';

其中，前两个参数的作用是要启用MRR,这么做的原因，BKA算法的优化要依赖于MRR

**BNL算法的性能问题**

说完了NLJ算法的优化，看BNL算法的优化

上篇文末，使用Block Nested-Loop Join(BNL)算法时，可能会对被驱动表做多次扫描，如果这个被驱动表是一个大的冷数据表，除了会导致IO压力大以外，还会对系统有什么影响呢？

在33篇中，InnoDB的LRU算法时提到，由于InnoDB对Buffer Pool的LRU算法做了优化，即：第一次从磁盘读入内存的数据页，会先放在old区域，如果1秒之后这个数据也不在被访问，就不会被移动到LRU链表头部，这样对Buffer Pool的命中率影响就不大

但是，如果一个是使用了BNL算法的join语句，多次扫描一个冷表，而且这个语句执行时间超过了1秒，就会在次扫描冷表的时候，把冷表的数据页移动到LRU链表头部。

如果这个冷表很大，就会出现另外一种情况：业务正常访问的数据页，没有机会进入young区域

由于优化机制的存在，一个正常访问的数据页，要进入young区域，需要隔1秒后再次被访问到。但是，由于join语句在循环读磁盘和淘汰内存页，进入old区域的数据页，很可能在1秒之内就被淘汰了，这样导致了这个MySQL实例的Buffer Pool在这段时间内，young区域的数据页没有被合理的淘汰。

也就是说，这两种情况都会影响Buffer Pool的正常运作

**大表join虽然对IO有影响，但是在语句执行结束后，对IO的影响也就结束了，但是，对Buffer Pool的影响就是持续的，需要依靠后续的查询请求慢慢恢复内存命中率**

为了减少这种影响，可以考虑增大join\_buffer\_size的值，减少对被驱动表的扫描次数

也就是说，BNL算法对系统的影响主要包括三个方面：

1. 可能会多次扫描被驱动表，占用磁盘IO资源
2. 判断join条件需要执行M\*N次对比（M、N分别是两张表的行数），如果是大表就会占用非常多的CPU资源
3. 可能会导致Buffer Pool的热数据被淘汰，影响内存命中率

执行语句之前，需要通过explain结果和理论分析，确认是否使用了BNL算法，如果确认优化器使用了BNL算法，就需要做优化，优化的常见做法是，给被驱动表的join字段加上索引，把BNL算法转换成BKA算法。

**BNL 转BKA**

一些情况下，可以直接在被驱动表上键索引，此时就可以直接转成BKA算法了。

但是有时，会碰到一些不适合在被驱动表上键索引的情况，比如下面的语句

select \* from t1 join t2 on (t1.b=t2.b) where t2.b>=1 and t2.b<=2000;

在表t2中插入了100万行数据，但是经过where条件过滤后，需要参与join的只有2000行数据，如果这条语句同时是一个低频的SQL语句，再为这个语句在表t2的字段b上创建一个索引就很浪费了

但是，如果使用了BNL算法来join的话，这个语句的执行流程是这样的

1. 把表t1的所有字段取出来，存入join\_buffer中，这个表只有1000行，join\_buffre\_size默认值是256k,可以完全存入
2. 扫描表t2,取出每一行数据跟join\_buffer中的数据进行对比

如果不满足t1.b=t2.b，则跳过

如果满足 t1.b=t2.b，在判断其他条件，也就是是否满足t2.b处于[1,2000]的条件

如果是，就作为结果集的一部分返回，否则跳过

在上一篇文中说过，对于表t2的每一行，判断join是否满足的时候，都需要遍历join\_buffer中的所有行，因此判断等值条件的次数是1000\*100万=10亿次，这个判断工作量大

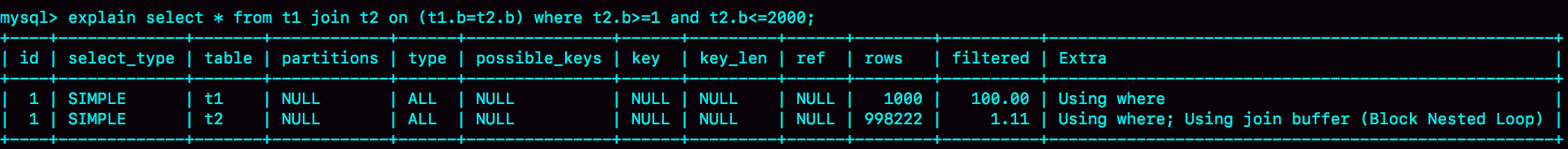


图 6 explain 结果

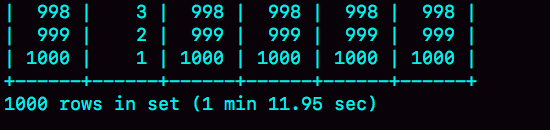


图 7 语句执行时间

explain结果是Extra字段显示使用了BNL算法，在测试环境，需要执行1分11秒。

在表t2的字段b上创建索引会浪费资源，但是不创建索引的话这个语句的等值条件要判断10亿次，工作量大，两全其美的办法

1. 把表t2中满足条件的数据放在临时表tmp\_t中
2. 为了让join使用BKA算法，给临时表tmp\_t的字段b加上索引
3. 让表t1和tmp\_t做join操作

此时，对应的SQL语句的写法如下

create temporary table temp\_t(id int primary key, a int, b int, index(b))engine=innodb;

insert into temp\_t select \* from t2 where b>=1 and b<=2000;

select \* from t1 join temp\_t on (t1.b=temp\_t.b);

图8 就是这个语句序列的执行结果

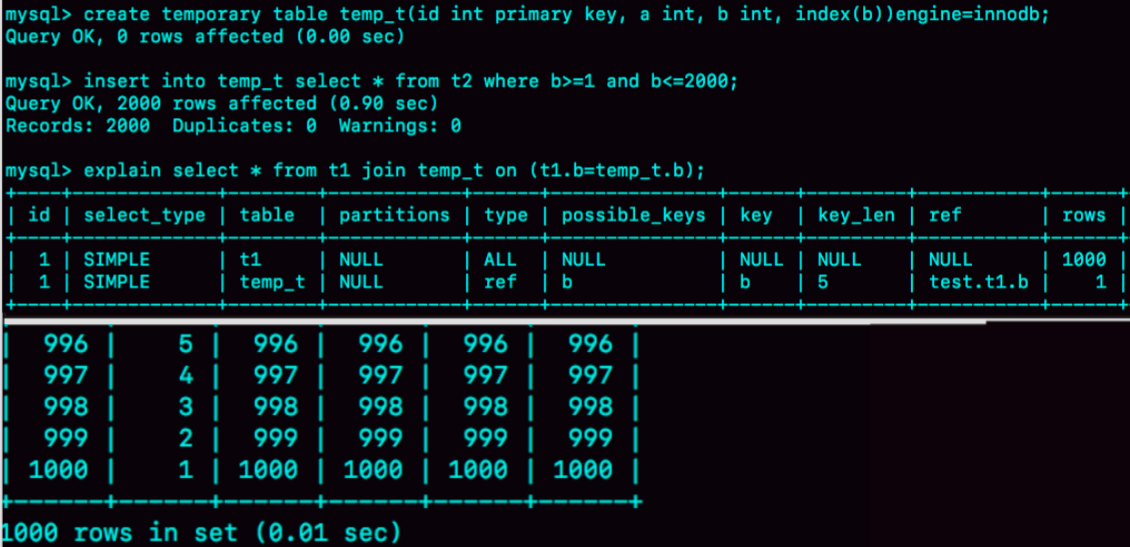


图 8 使用临时表的执行结果

这个过程3个SQL执行时间的总和不到1秒，相比于前面的1分11秒，性能得到了大幅提升，看下这个过程的消耗

1. 执行insert语句构造temp\_t表并插入数据的过程中，对表t2做了全表扫描，这里扫描行数是100万
2. 之后的join语句，扫描表t1,这里的扫描行数是1000，join比较过程中，做了1000次带索引的查询，相比于优化前的join语句需要做10亿次条件判断来说，这个优化效果还是明显的

总体来说，不论是在原表上加索引，还是用有索引的临时表，思路都是让join语句能够用上被驱动表上的索引，来触发BKA算法，提升查询性能。

扩展 -hash join

如果join\_buffer里面维护的不是一个无序数组，而是一个哈希表的话，那么就不是10亿次判断，而是100万次hash查找，这样的话，整条语句的执行速度够快了

MySQL的优化器和执行器一直被诟病的一个原因；不支持哈希join

1. select \* from t1；取表t1的全部1000行数据，在业务端存入一个hash结构
2. Select \* from t2 where b>=1 and b<=2000 ；获取表t2满足条件的2000行数据
3. 把这2000行数据，一行一行的取出到业务端，到hash结构的数据表中寻找匹配的数据，满足匹配条件的这行数据，作为结果集的一行。

思考题

现在有一个三个表join的需求，假设这个三个表的结果如下

CREATE TABLE `t1` (

`id` int(11) NOT NULL,

`a` int(11) DEFAULT NULL,

`b` int(11) DEFAULT NULL,

`c` int(11) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB;

create table t2 like t1;

create table t3 like t2;

insert into ... //初始化三张表的数据

为了得到最快的执行速度，如果设计表t1,t2,t3上的索引，来支持这个join 语句，会加那些索引？

如果希望用staight\_join来重写这个语句，配合索引，需要安排连接顺序，主要考虑因素是什么？