

34 | 到底可不可以使用join?

2019-01-30 林晓斌



在实际生产中，关于**join**语句使用的问题，一般会集中在以下两类：

1. 我们DBA不让使用**join**，使用**join**有什么问题呢？
2. 如果有两个大小不同的表做**join**，应该用哪个表做驱动表呢？

今天这篇文章，我就先跟你说说**join**语句到底是怎么执行的，然后再来回答这两个问题。

为了便于量化分析，我还是创建两个表t1和t2来和你说明。

```
CREATE TABLE `t2` (
  `id` int(11) NOT NULL,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `a` (`a`)
) ENGINE=InnoDB;
```

```
drop procedure idata;
delimiter ;;
create procedure idata()
begin
  declare i int;
  set i=1;
  while(i<=1000)do
    insert into t2 values(i, i, i);
    set i=i+1;
  end while;
end;;
delimiter ;
call idata();

create table t1 like t2;
insert into t1 (select * from t2 where id<=100)
```

可以看到，这两个表都有一个主键索引**id**和一个索引**a**，字段**b**上无索引。存储过程**idata()**往表**t2**里插入了1000行数据，在表**t1**里插入的是100行数据。

Index Nested-Loop Join

我们来看一下这个语句：

```
select * from t1 straight_join t2 on (t1.a=t2.a);
```

如果直接使用**join**语句，MySQL优化器可能会选择表**t1**或**t2**作为驱动表，这样会影响我们分析SQL语句的执行过程。所以，为了便于分析执行过程中的性能问题，我改用**straight_join**让MySQL使用固定的连接方式执行查询，这样优化器只会按照我们指定的方式去**join**。在这个语句

里，**t1**是驱动表，**t2**是被驱动表。

现在，我们来看一下这条语句的**explain**结果。

```
mysql> explain select * from t1 straight_join t2 on (t1.a=t2.a);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | t1    | NULL      | ALL  | a           | NULL | NULL    | NULL | 100  | 100.00   | Using where |
| 1 | SIMPLE     | t2    | NULL      | ref  | a           | a    | 5       | test.t1.a | 1    | 100.00   | NULL        |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图1 使用索引字段join的 explain结果

可以看到，在这条语句里，被驱动表**t2**的字段**a**上有索引，**join**过程用上了这个索引，因此这个语句的执行流程是这样的：

1. 从表**t1**中读入一行数据 **R**;
2. 从数据行**R**中，取出**a**字段到表**t2**里去查找;
3. 取出表**t2**中满足条件的行，跟**R**组成一行，作为结果集的一部分;
4. 重复执行步骤1到3，直到表**t1**的末尾循环结束。

这个过程是先遍历表**t1**，然后根据从表**t1**中取出的每行数据中的**a**值，去表**t2**中查找满足条件的记录。在形式上，这个过程就跟我们写程序时的嵌套查询类似，并且可以用上被驱动表的索引，所以我们称之为“**Index Nested-Loop Join**”，简称**NLJ**。

它对应的流程图如下所示：

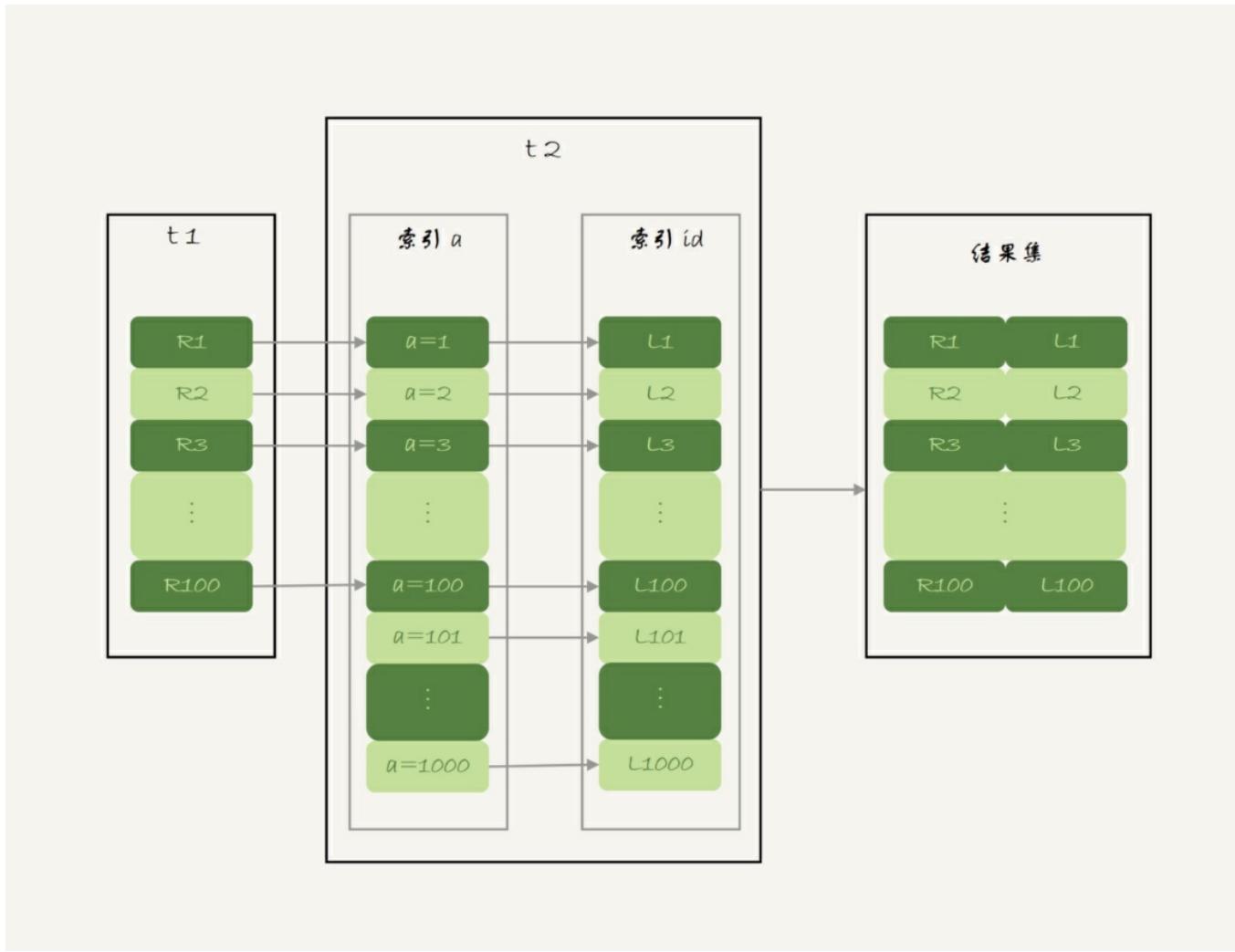


图2 Index Nested-Loop Join算法的执行流程

在这个流程里：

1. 对驱动表t1做了全表扫描，这个过程需要扫描100行；
2. 而对于每一行R，根据a字段去表t2查找，走的是树搜索过程。由于我们构造的数据都是一一对应的，因此每次的搜索过程都只扫描一行，也是总共扫描100行；
3. 所以，整个执行流程，总扫描行数是200。

现在我们知道了这个过程，再试着回答一下文章开头的两个问题。

先看第一个问题：能不能使用join？

假设不使用join，那我们就只能用单表查询。我们看看上面这条语句的需求，用单表查询怎么实现。

1. 执行select * from t1，查出表t1的所有数据，这里有100行；
2. 循环遍历这100行数据：

- 从每一行**R**取出字段**a**的值\$R.a;
- 执行select * from t2 where a=\$R.a;
- 把返回的结果和**R**构成结果集的一行。

可以看到，在这个查询过程，也是扫描了200行，但是总共执行了101条语句，比直接join多了100次交互。除此之外，客户端还要自己拼接SQL语句和结果。

显然，这么做还不如直接join好。

我们再来看看第二个问题：怎么选择驱动表？

在这个join语句执行过程中，驱动表是走全表扫描，而被驱动表是走树搜索。

假设被驱动表的行数是M。每次在被驱动表查一行数据，要先搜索索引a，再搜索主键索引。每次搜索一棵树近似复杂度是以2为底的M的对数，记为 $\log_2 M$ ，所以在被驱动表上查一行的时间复杂度是 $2 * \log_2 M$ 。

假设驱动表的行数是N，执行过程就要扫描驱动表N行，然后对于每一行，到被驱动表上匹配一次。

因此整个执行过程，近似复杂度是 $N + N * 2 * \log_2 M$ 。

显然，N对扫描行数的影响更大，因此应该让小表来做驱动表。

如果你没觉得这个影响有那么“显然”，可以这么理解：N扩大1000倍的话，扫描行数就会扩大1000倍；而M扩大1000倍，扫描行数扩大不到10倍。

到这里小结一下，通过上面的分析我们得到了两个结论：

1. 使用join语句，性能比强行拆成多个单表执行SQL语句的性能要好；
2. 如果使用join语句的话，需要让小表做驱动表。

但是，你需要注意，这个结论的前提是“可以使用被驱动表的索引”。

接下来，我们再看看被驱动表用不上索引的情况。

Simple Nested-Loop Join

现在，我们把SQL语句改成这样：

```
select * from t1 straight_join t2 on (t1.a=t2.b);
```

由于表t2的字段b上没有索引，因此再用图2的执行流程时，每次到t2去匹配的时候，就要做一次

全表扫描。

你可以先设想一下这个问题，继续使用图2的算法，是不是可以得到正确的结果呢？如果只看结果的话，这个算法是正确的，而且这个算法也有一个名字，叫做“**Simple Nested-Loop Join**”。

但是，这样算来，这个**SQL**请求就要扫描表t2多达100次，总共扫描 $100 \times 1000 = 10$ 万行。

这还只是两个小表，如果t1和t2都是10万行的表（当然了，这也还是属于小表的范围），就要扫描100亿行，这个算法看上去太“笨重”了。

当然，MySQL也没有使用这个**Simple Nested-Loop Join**算法，而是使用了另一个叫作“**Block Nested-Loop Join**”的算法，简称BNL。

Block Nested-Loop Join

这时候，被驱动表上没有可用的索引，算法的流程是这样的：

1. 把表t1的数据读入线程内存join_buffer中，由于我们这个语句中写的是**select ***，因此是把整个表t1放入了内存；
2. 扫描表t2，把表t2中的每一行取出来，跟join_buffer中的数据做对比，满足join条件的，作为结果集的一部分返回。

这个过程的流程图如下：

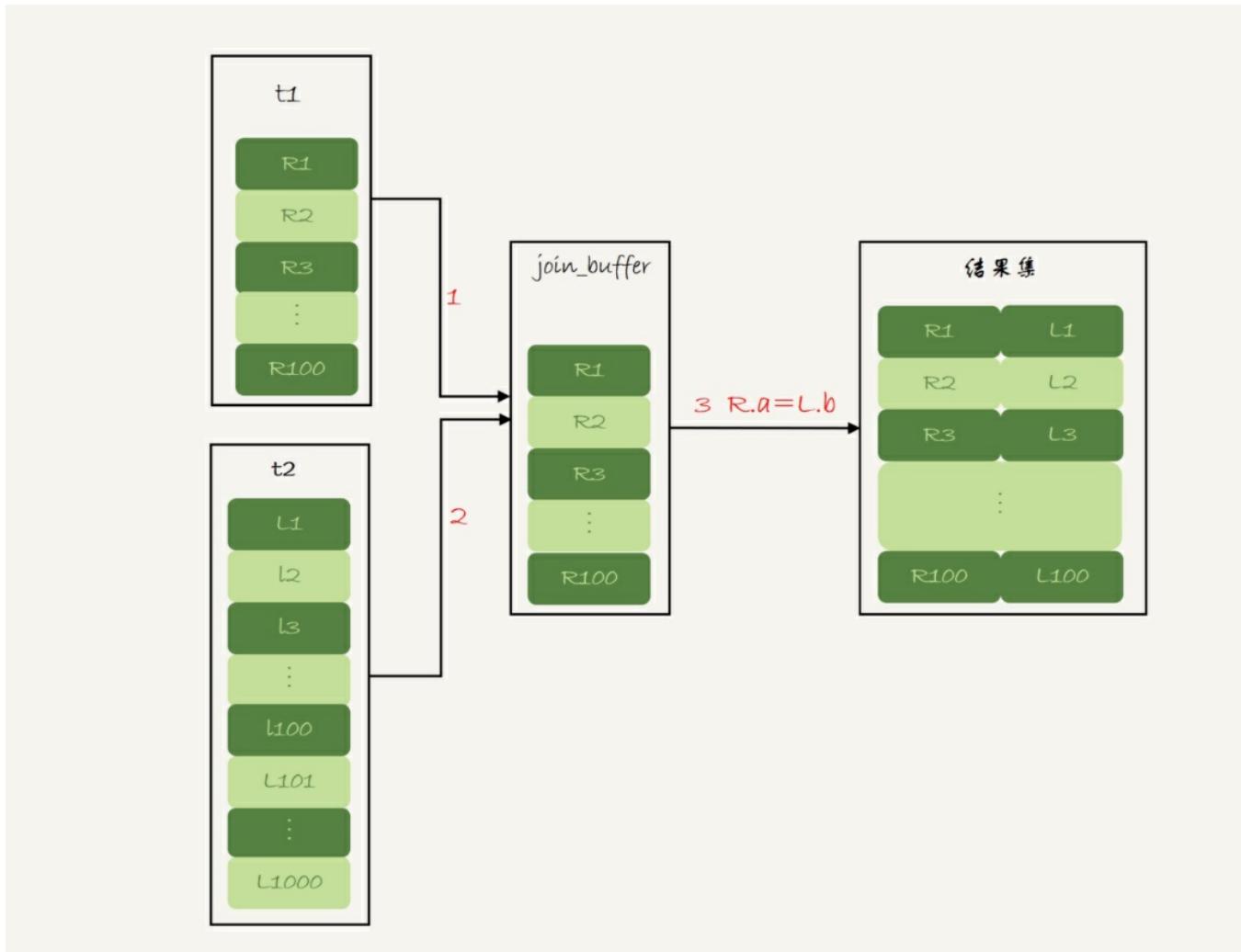


图3 Block Nested-Loop Join 算法的执行流程

对应地，这条SQL语句的explain结果如下所示：

```
mysql> explain select * from t1 straight_join t2 on (t1.a=t2.b);
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | t1   | NULL      | ALL  | a             | NULL | NULL    | NULL | 100  | 100.00  | NULL
| 1 | SIMPLE     | t2   | NULL      | ALL  | NULL          | NULL | NULL    | NULL | 1000 | 10.00   | Using where; Using join buffer (Block Nested Loop)
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

图4 不使用索引字段join的 explain结果

可以看到，在这个过程中，对表**t1**和**t2**都做了一次全表扫描，因此总的扫描行数是1100。由于**join_buffer**是以无序数组的方式组织的，因此对表**t2**中的每一行，都要做100次判断，总共需要在内存中做的判断次数是： $100 \times 1000 = 10$ 万次。

前面我们说过，如果使用**Simple Nested-Loop Join**算法进行查询，扫描行数也是10万行。因此，从时间复杂度上来说，这两个算法是一样的。但是，**Block Nested-Loop Join**算法的这10万次判断是内存操作，速度上会快很多，性能也更好。

接下来，我们来看一下，在这种情况下，应该选择哪个表做驱动表。

假设小表的行数是**N**，大表的行数是**M**，那么在这个算法里：

1. 两个表都做一次全表扫描，所以总的扫描行数是M+N；
2. 内存中的判断次数是M*N。

可以看到，调换这两个算式中的M和N没差别，因此这时候选择大表还是小表做驱动表，执行耗时是一样的。

然后，你可能马上就会问了，这个例子里表t1才100行，要是表t1是一个大表，join_buffer放不下怎么办呢？

join_buffer的大小是由参数join_buffer_size设定的，默认值是256k。如果放不下表t1的所有数据话，策略很简单，就是分段放。我把join_buffer_size改成1200，再执行：

```
select * from t1 straight_join t2 on (t1.a=t2.b);
```

执行过程就变成了：

1. 扫描表t1，顺序读取数据行放入join_buffer中，放完第88行join_buffer满了，继续第2步；
2. 扫描表t2，把t2中的每一行取出来，跟join_buffer中的数据做对比，满足join条件的，作为结果集的一部分返回；
3. 清空join_buffer；
4. 继续扫描表t1，顺序读取最后的12行数据放入join_buffer中，继续执行第2步。

执行流程图也就变成这样：

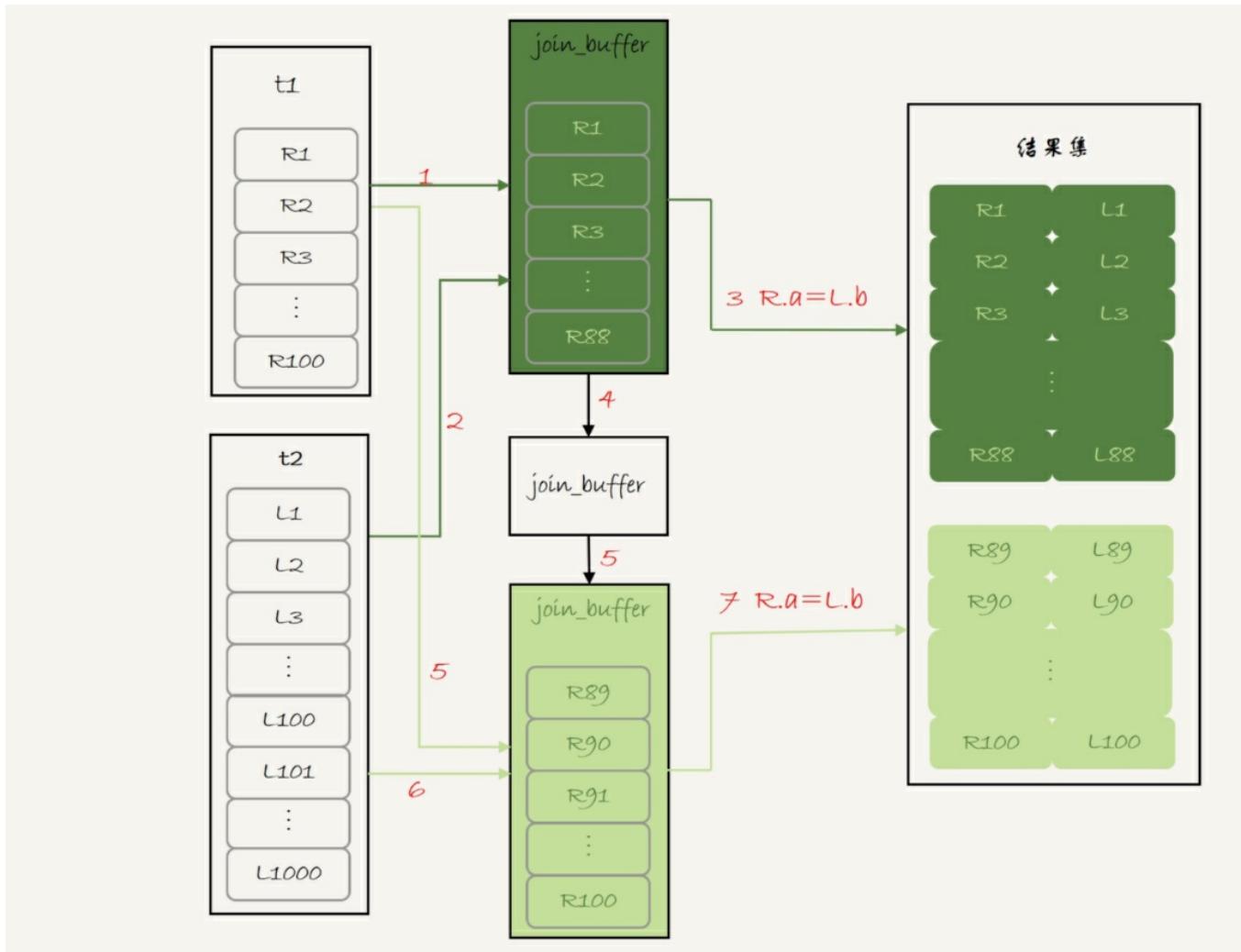


图5 Block Nested-Loop Join – 两段

图中的步骤4和5，表示清空join_buffer再复用。

这个流程才体现出了这个算法名字中“**Block**”的由来，表示“分块去join”。

可以看到，这时候由于表 t_1 被分成了两次放入join_buffer中，导致表 t_2 会被扫描两次。虽然分成两次放入join_buffer，但是判断等值条件的次数还是不变的，依然是 $(88+12)*1000=10$ 万次。

我们再来看下，在这种情况下驱动表的选择问题。

假设，驱动表的数据行数是 N ，需要分 K 段才能完成算法流程，被驱动表的数据行数是 M 。

注意，这里的 K 不是常数， N 越大 K 就会越大，因此把 K 表示为 $\lambda * N$ ，显然 λ 的取值范围是 $(0,1)$ 。

所以，在这个算法的执行过程中：

1. 扫描行数是 $N + \lambda * N * M$;
2. 内存判断 $N * M$ 次。

显然，内存判断次数是不受选择哪个表作为驱动表影响的。而考虑到扫描行数，在 M 和 N 大小确

定的情况下， N 小一些，整个算式的结果会更小。

所以结论是，应该让小表当驱动表。

当然，你会发现，在 $N+\lambda \cdot N \cdot M$ 这个式子里， λ 才是影响扫描行数的关键因素，这个值越小越好。

刚刚我们说了 N 越大，分段数 K 越大。那么， N 固定的时候，什么参数会影响 K 的大小呢？（也就是 λ 的大小）答案是`join_buffer_size`。`join_buffer_size`越大，一次可以放入的行越多，分成的段数也就越少，对被驱动表的全表扫描次数就越少。

这就是为什么，你可能会看到一些建议告诉你，如果你的`join`语句很慢，就把`join_buffer_size`改大。

理解了MySQL执行`join`的两种算法，现在我们再来试着回答文章开头的两个问题。

第一个问题是：能不能使用`join`语句？

1. 如果可以使用**Index Nested-Loop Join**算法，也就是说可以用上被驱动表上的索引，其实是没有问题的；
2. 如果使用**Block Nested-Loop Join**算法，扫描行数就会过多。尤其是在大表上的`join`操作，这样可能要扫描被驱动表很多次，会占用大量的系统资源。所以这种`join`尽量不要用。

所以你在判断要不要使用`join`语句时，就是看`explain`结果里面，`Extra`字段里面有没有出现“**Block Nested Loop**”字样。

第二个问题是：如果要使用`join`，应该选择大表做驱动表还是选择小表做驱动表？

1. 如果是**Index Nested-Loop Join**算法，应该选择小表做驱动表；
2. 如果是**Block Nested-Loop Join**算法：
 - 在`join_buffer_size`足够大的时候，是一样的；
 - 在`join_buffer_size`不够大的时候（这种情况更常见），应该选择小表做驱动表。

所以，这个问题的结论就是，总是应该使用小表做驱动表。

当然了，这里我需要说明下，什么叫作“小表”。

我们前面的例子是没有加条件的。如果我在语句的`where`条件加上`t2.id<=50`这个限定条件，再来看下这两条语句：

```
select * from t1 straight_join t2 on (t1.b=t2.b) where t2.id<=50;  
select * from t2 straight_join t1 on (t1.b=t2.b) where t2.id<=50;
```

注意，为了让两条语句的被驱动表都用不上索引，所以join字段都使用了没有索引的字段**b**。

但如果是用第二个语句的话，**join_buffer**只需要放入**t2**的前50行，显然是更好的。所以这里，“**t2**的前50行”是那个相对小的表，也就是“小表”。

我们再来看另外一组例子：

```
select t1.b,t2.* from t1 straight_join t2 on (t1.b=t2.b) where t2.id<=100;  
select t1.b,t2.* from t2 straight_join t1 on (t1.b=t2.b) where t2.id<=100;
```

这个例子里，表**t1**和**t2**都是只有100行参加join。但是，这两条语句每次查询放入**join_buffer**中的数据是不一样的：

- 表**t1**只查字段**b**，因此如果把**t1**放到**join_buffer**中，则**join_buffer**中只需要放入**b**的值；
- 表**t2**需要查所有的字段，因此如果把表**t2**放到**join_buffer**中的话，就需要放入三个字段**id**、**a**和**b**。

这里，我们应该选择表**t1**作为驱动表。也就是说在这个例子里，“只需要一列参与join的表**t1**”是那个相对小的表。

所以，更准确地说，在决定哪个表做驱动表的时候，应该是两个表按照各自的条件过滤，过滤完成之后，计算参与join的各个字段的总数据量，数据量小的那个表，就是“小表”，应该作为驱动表。

小结

今天，我和你介绍了MySQL执行join语句的两种可能算法，这两种算法是由能否使用被驱动表的索引决定的。而能否用上被驱动表的索引，对join语句的性能影响很大。

通过对Index Nested-Loop Join和Block Nested-Loop Join两个算法执行过程的分析，我们也得到了文章开头两个问题的答案：

1. 如果可以使用被驱动表的索引，join语句还是有其优势的；
2. 不能使用被驱动表的索引，只能使用Block Nested-Loop Join算法，这样的语句就尽量不要使用；
3. 在使用join的时候，应该让小表做驱动表。

最后，又到了今天的问题时间。

我们在上文说到，使用Block Nested-Loop Join算法，可能会因为**join_buffer**不够大，需要对被

驱动表做多次全表扫描。

我的问题是，如果被驱动表是一个大表，并且是一个冷数据表，除了查询过程中可能会导致IO压力大以外，你觉得对这个MySQL服务还有什么更严重的影响吗？（这个问题需要结合上一篇文章的知识点）

你可以把你的结论和分析写在留言区，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下的问题是，如果客户端由于压力过大，迟迟不能接收数据，会对服务端造成什么严重的影响。

这个问题的核心是，造成了“长事务”。

至于长事务的影响，就要结合我们前面文章中提到的锁、MVCC的知识点了。

- 如果前面的语句有更新，意味着它们在占用着行锁，会导致别的语句更新被锁住；
- 当然读的事务也有问题，就是会导致undo log不能被回收，导致回滚段空间膨胀。

评论区留言点赞板：

@老杨同志 提到了更新之间会互相等锁的问题。同一个事务，更新之后要尽快提交，不要做没必要的查询，尤其是不要执行需要返回大量数据的查询；

@长杰 同学提到了undo表空间变大，db服务堵塞，服务端磁盘空间不足的例子。

The image is a promotional graphic for a MySQL course. It features a portrait of the instructor, Ding Qi, a man with short dark hair and glasses, wearing a black button-down shirt, standing with his arms crossed. To the left of the portrait, the course title "MySQL 实战 45 讲" is displayed in large, bold, dark font. Below the title, a subtitle reads "从原理到实战，丁奇带你搞懂 MySQL". In the bottom left corner, the author's name "林晓斌" is written in a large, stylized font, with smaller text below it identifying him as "网名丁奇" and "前阿里资深技术专家". The overall background is a light gray gradient.

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。

精选留言



没时间了ngu

0

join这种用的多的，看完还是有很大收获的。像之前讲的锁之类，感觉好抽象，老是记不住，唉。

2019-01-30

| 作者回复

嗯嗯，因为其实每个同学的只是背景不一样。

这45讲里，每个同学都能从部分文章感觉到有收获，我觉得也很好了！

不过 锁其实用得也多的。。

我以前负责业务库的时候，被开发同学问最多的问题之一就是，为啥死锁了^_^

2019-01-30



抽离の』

8

早上听老师一节课感觉获益匪浅

2019-01-30

| 作者回复

好早呀！

2019-01-30



信信

6

老师好，回答本期问题：如果驱动表分段，那么被驱动表就被多次读，而被驱动表又是大表，循环读取的间隔肯定得超1秒，这就会导致上篇文章提到的：“数据页在LRU_old的存在时间超过1秒，就会移到young区”。最终结果就是把大部分热点数据都淘汰了，导致“Buffer pool hit rate”命中率极低，其他请求需要读磁盘，因此系统响应变慢，大部分请求阻塞。

2019-01-30

| 作者回复

』

2019-01-30



老杨同志

3

对被驱动表进行全表扫描，会把冷数据的page加入到buffer pool，并且block nested-loop要扫描多次，两次扫描的时间可能会超过1秒，使lru的那个优化失效，把热点数据从buffer pool中淘汰掉，影响正常业务的查询效率

2019-01-30

| 作者回复

漂亮！

2019-01-30





壮火虫

1 0

年底了有一种想跳槽的冲动 身在武汉的我想出去看看 可一想到自身的能力和学历 又不敢去了
苦恼...

2019-01-30

作者回复

今年这情况还是要先克制一下^_^

先把内功练起来！

2019-01-30



清风浊酒

1 2

老师您好，`left join` 和 `right join` 会固定驱动表吗？

2019-01-30

作者回复

不会强制，但是由于语义的关系，大概率上是按照语句上写的关系去驱动，效率是比较高的

2019-01-30



柚子

1 2

`join`在热点表操作中，`join`查询是一次给两张表同时加锁吧，会不会增大锁冲突的几率？

业务中肯定要使用被驱动表的索引，通常我们是先在驱动表查出结果集，然后再通过`in`被驱动表索引字段，分两步查询，这样是否比直接`join`委托点？

2019-01-30

作者回复

`join`也是普通查询，都不需要加锁哦，参考下MVCC那篇；

就是我们文中说的，“分两步查询，先查驱动表，然后查多个`in`”，如果可以用上被驱动表的索引，我觉得可以用上Index Nested-Loop Join算法，其实效果是跟拆开写类似的

2019-01-30



郝攀刚

1 1

业务逻辑关系，一个SQL中`left join`7，8个表。这我该怎么优化。每次看到这些脑壳就大！

2019-01-30

作者回复

[]

`Explain`下，没用用`index nested-loop`的全要优化

2019-01-31



Zzz

1 1

林老师，我没想清楚为什么会进入`young`区域。假设大表`t`大小是 M 页>`old`区域 N 页，由于`Block Nested-Loop Join`需要对`t`进行 k 次全表扫描。第一次扫描时， $1 \sim N$ 页依次被放入`old`区域，访问 $N+1$ 页时淘汰1页，放入 $N+1$ 页，以此类推，第一次扫描结束后`old`区域存放的是 $M-N+1 \sim M$ 页。第二次扫描开始，访问1页，淘汰 $M-N+1$ 页，放入1页。可以把 M 页想象成一个环， N 页想象成在这个环上滑动的窗口，由于 $M > N$ ，不管是哪次扫描，需要访问的页都不会在滑动窗口上，所以不会存在“被访问的时候数据页在 LRU 链表中存在的时间超过了 1 秒”而被放入`young`的情况

。我能想到的会被放入young区域的情况是，在当次扫描中，由于一页上有~~多~~行数据，需要对该页访问多次，超过了1s，不管这种情况就和t大小没关系了，而是由于page size太大，而一行数据太少。

2019-01-30

作者回复

你说得对，分两类情况，

小于bp 3/8的情况会跑到young，

大于3/8的会影响young部分的更新

2019-01-30



700

1

老师，您好。看完文章后有如下问题请教：

1) 文章内容「可以看到，在这个查询过程，也是扫描了 200 行，但是总共执行了 101 条语句，比直接 join 多了 100 次交互。除此之外，客户端还要自己拼接 SQL 语句和结果。」这个有没有啥方法来仅通过1次交互就将这101条语句发到服务端执行？

2) 文章内容「每次搜索一棵树近似复杂度是以 2 为底的 M 的对数，记为 $\log_2 M$ ，所以在被驱动表上查一行的时间复杂度是 $2 * \log_2 M$ 。」

这个复杂度的计算难理解，为什么是这么计算？

假设 $M = 256$ ，则搜索树的复杂度为8？

3) 文章内容「因此整个执行过程，近似复杂度是 $N + N * 2 * \log_2 M$ 。」

驱动表的复杂度直接记为 N？

4) 文中提到索引扫描需扫1行数据，全表扫描需扫1000行数据。这是由统计信息决定的？

提前感谢老师！

2019-01-30

作者回复

1. 用 in，但是不建议语句太长

2. 看一下前面我们介绍索引的文章哈

3. 因为是在叶子索引上直接顺序扫描，是一个大致值哈

4. 不是呀，因为表t2是1000行哦

2019-01-30



Ryoma

1

前提：冷数据表 & 大表

buffer pool 中的old区会被持续刷新，并且基本没有升级到young区的可能性。

一定程度上会降低hit rate

2019-01-30



403

0

用那个作为驱动表，mysql会自己优化么？

2019-02-09

| 作者回复

会的

2019-02-10



陈华应

0

老师，放完88行就满了，88是怎么计算得来的呢？

2019-02-02

| 作者回复

这个是实际跑出来的效果

如果说计算的话，每一行固定长度，你用1024除一下

2019-02-02



库淘淘

0

```
set optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
create index idx_c on t2(c);
create index idx_a_c on t1(a,c);
create index idx_b_c on t3(b,c);
mysql> explain select * from t2
-> straight_join t1 on(t1.a=t2.a)
-> straight_join t3 on(t2.b=t3.b)
-> where t1.c> 800 and t2.c>=600 and t3.c>=500;
+-----+-----+-----+-----+-----+-----+-----+
+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | NULL | range | idx_c | idx_c | 5 | NULL | 401 | 100.00 | Using index condition
; Using where; Using MRR |
| 1 | SIMPLE | t1 | NULL | ref | idx_a_c | idx_a_c | 5 | test.t2.a | 1 | 33.33 | Using index condition;
Using join buffer (Batched Key Access) |
| 1 | SIMPLE | t3 | NULL | ref | idx_b_c | idx_b_c | 5 | test.t2.b | 1 | 33.33 | Using index condition;
Using join buffer (Batched Key Access) |
+-----+-----+-----+-----+-----+-----+-----+
```

3 rows in set, 1 warning (0.00 sec)

以自己理解考虑如下，有问题请老师能够指出

- 1.根据查询因是select * 肯定回表的，其中在表t2创建索引idx_c,为了能够使用ICP,MRR，如果c字段重复率高或取值行数多，可以考虑不建索引
- 2.已t2 作为驱动表，一方面考虑其他两表都有关联,t2表放入join buffer后关联t1后，再关联t2 得

出结果 再各回t2,t3表取出 得到结果集（之前理解都是t1和t2join得结果集再与t3join，本次理解太确定）

3.t2、t3表建立联合查询目的能够使用ICP

2019-02-01

作者回复

“2.已t2作为驱动表，一方面考虑其他两表都有关联,t2表放入join buffer后关联t1后，再关联t2得出结果 再各回t2,t3表取出 得到结果集”

即使是用t1做驱动表，也是可能可以都用上BKA的哈

新春快乐~

2019-02-04



郭健

0

老师，太棒了！！终于讲join了！！！作为一个实际开发人员，索引了解是必须得，单表索引有所掌握，始终对join没法理解，这节课对我的帮助是最大的。谢谢老师

2019-02-01

作者回复

0

2019-02-01



辣椒

0

我是开发，但是看了老师的专栏，对怎么写数据库应用更有心得了

2019-01-31

作者回复

0，如果有有趣的经验也放到这里跟大家分享哦

2019-02-01



泡泡爱dota

0

explain select * from t1 straight_join t2 on (t1.a=t2.a) where t1.a < 50;

老师，这条sql为什么t1.a的索引没有用上，t1还是走全表

2019-01-31

作者回复

如果数据量不够多，并且满足a<50的行，占比比较高的话，优化器有可能会认为“还要回表，还不如直接扫主键id”

2019-01-31



剃刀吗啡

0

我们某个业务使用infobright这种列式存储，字段没用索引。我在想这种引擎在join的时候是否也会遵守类似的规则？但列式存储并不是按行扫描，所以有点困惑。

2019-01-31

作者回复

是的，只是获取数据的时候，不会去读整行。

但是没有索引就也只能用BNL，可以explain看看

2019-01-31



一大只

0

老师，我想问下，如果使用的是Index Nested-Loop Join，是不是就不会使用join_buffer了？直接将循环结果放到net_buffer_length中，边读边发哈？

2019-01-31

| 作者回复

是的，Index Nested-Loop Join没有用到join buffer

不过35篇马上会介绍到一个优化，把join buffer用上，晚上关注下哦

2019-01-31



斜面镜子 Bill

0

因为 join_buffer 不够大，需要对被驱动表做多次全表扫描，也就造成了“长事务”。除了老师上节课提到的导致undo log 不能被回收，导致回滚段空间膨胀问题，还会出现：1. 长期占用DML锁，引发DDL拿不到锁堵塞连接池；2. SQL执行socket_timeout超时后业务接口重复发起，导致实例IO负载上升出现雪崩；3. 实例异常后，DBA kill SQL因繁杂的回滚执行时间过长，不能快速恢复可用；4. 如果业务采用select *作为结果集返回，极大可能出现网络拥堵，整体拖慢服务端的处理；5. 冷数据污染buffer pool，block nested-loop多次扫描，其中间隔很有可能超过1s，从而污染到lru头部，影响整体的查询体验。

2019-01-31

| 作者回复

很赞

之前知识点的也都加进来啦

2019-01-31