

34 | 到底可不可以使用join？

2019-01-30 林晓斌

MySQL实战45讲

[进入课程 >](#)



讲述：林晓斌

时长 16:39 大小 15.25M




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。

 复制代码

```
1 CREATE TABLE `t2` (  
2   `id` int(11) NOT NULL,  
3   `a` int(11) DEFAULT NULL,
```

```
4  `b` int(11) DEFAULT NULL,  
5  PRIMARY KEY (`id`),  
6  KEY `a` (`a`)  
7 ) ENGINE=InnoDB;  
8  
9 drop procedure idata;  
10 delimiter ;;  
11 create procedure idata()  
12 begin  
13     declare i int;  
14     set i=1;  
15     while(i<=1000)do  
16         insert into t2 values(i, i, i);  
17         set i=i+1;  
18     end while;  
19 end;;  
20 delimiter ;  
21 call idata();  
22  
23 create table t1 like t2;  
24 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：

 复制代码

```
1 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：

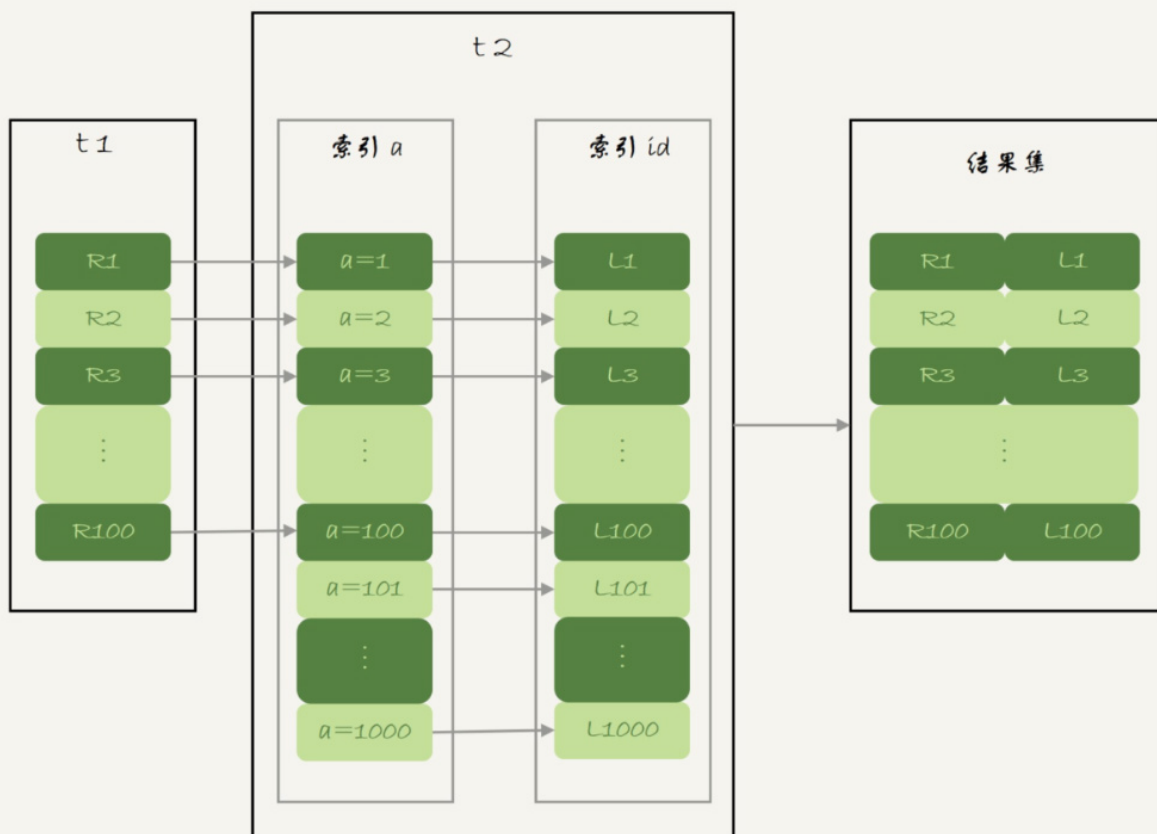


图 2 Index Nested-Loop Join 算法的执行流程

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：**如何选择驱动表？**

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：

 复制代码

```
1 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：

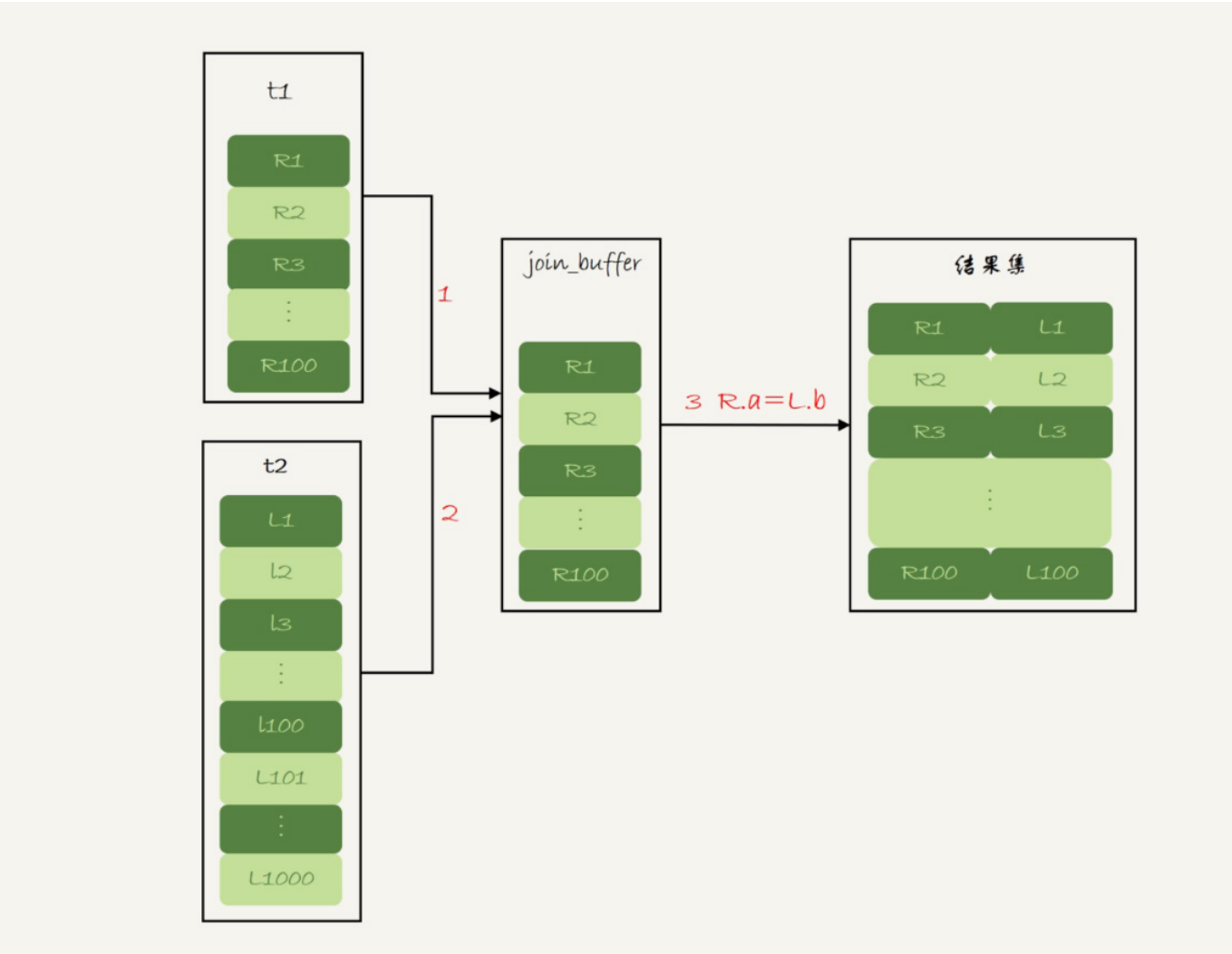


图 3 Block Nested-Loop Join 算法的执行流程

对应地，这条 SQL 语句的 explain 结果如下所示：

```
mysql> 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 $M \times N$ 。

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 $t1$ 的所有数据的话，策略很简单，就是分段放。我把 `join_buffer_size` 改成 1200，再执行：

 复制代码

```
1 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：

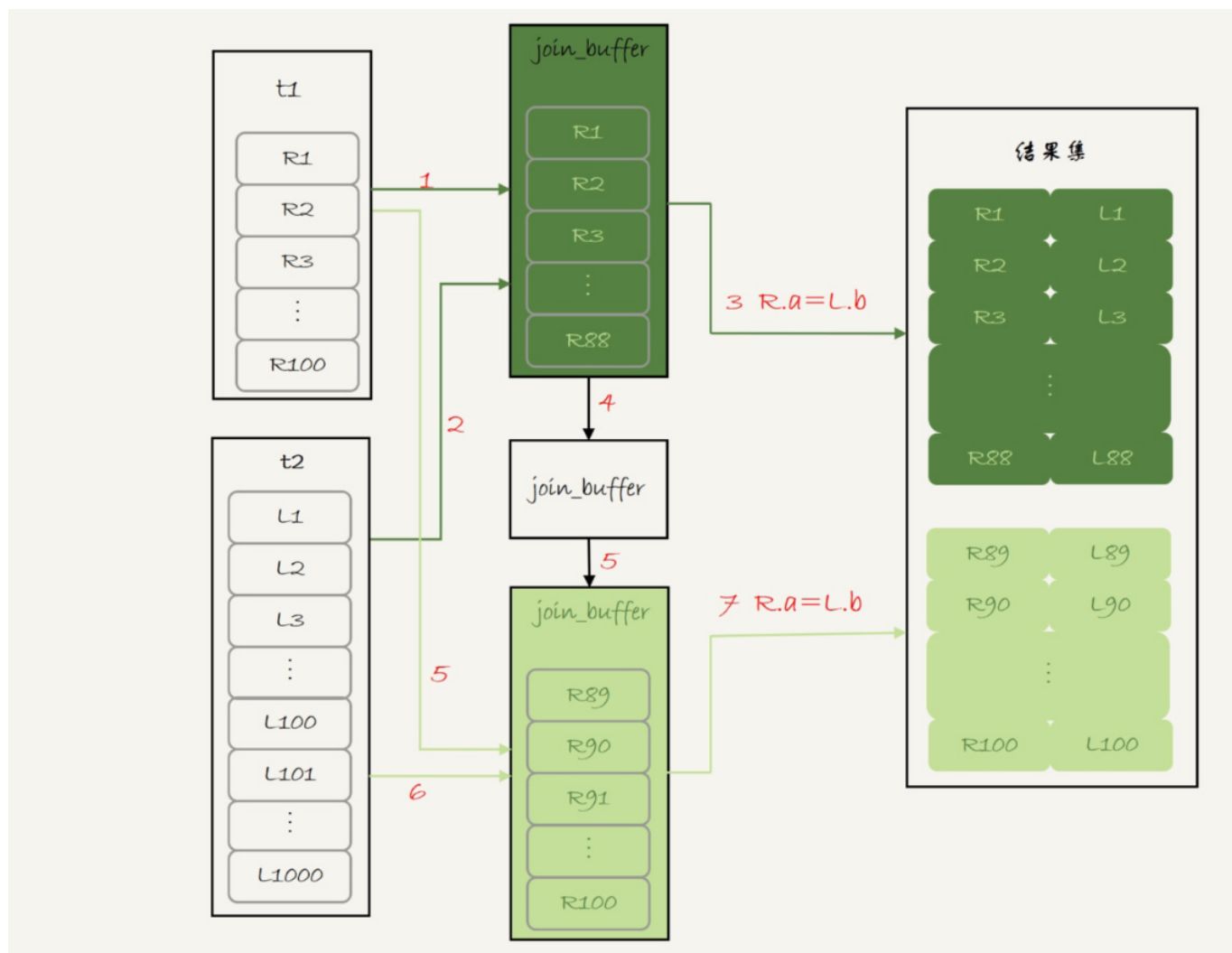


图 5 Block Nested-Loop Join -- 两段

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 t1 被分成了两次放入 join_buffer 中，导致表 t2 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 $N + \lambda * N * M$ 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：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：


 复制代码

```
1 select * from t1 straight_join t2 on (t1.b=t2.b) where t2.id<=50;
2 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：

 复制代码

```
1 select t1.b,t2.* from t1 straight_join t2 on (t1.b=t2.b) where t2.id<=100;
2 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，“只需要一行参与 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。

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 33 | 我查这么多数据，会不会把数据库内存打爆？

下一篇 35 | join语句怎么优化？

精选留言 (68)

写留言



没时间了ng... 置顶

2019-01-30

👍 1

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。

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 ^_^



信信

2019-01-30

👍 21

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。

展开 ▾

作者回复: 👍



抽离の

2019-01-30

👍 10

早上听老师一节课感觉获益匪浅

展开 ▾

作者回复: 好早呀💖



老杨同志

2019-01-30

👍 8

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

展开 ▾

作者回复: 漂亮👍



斜面镜子 ...

2019-01-31

👍 7

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拿不到锁堵慢连接池；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...
展开 ∨

作者回复: 很赞
之前知识点的也都加进来啦



萤火虫

2019-01-30

👍 5

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...

作者回复: 今年这情况还是要先克制一下^_^
先把内功练起来😁



Zzz

2019-01-30

👍 4

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~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~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...
展开 ∨

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



清风浊酒

2019-01-30

👍 4

老师您好，left join 和 right join 会固定驱动表吗？

展开 ∨

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



柚子

2019-01-30

👍 4

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?

展开 ∨

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



朝夕心

2019-02-19

👍 2

文中解释NLJ和BNL时间复杂度相同, 都是 $M*N$ 。但是对于BNL性能好于NLJ的原因只是提到:“BNL的判断是在内存中操作, 速度上会快很多, 性能也更好”。请问老师? 这句话的言外之意是: NLJ的判断不是在内存中操作吗? 不将数据加载到内存, CPU如何进行判断呢?

展开 ∨

作者回复: 这个我在答疑文章中展开哈, 其实还是“内存数据是从哪里来的”的问题。

我们这里说的是BNL比Simple nested loop 快哈



泡泡爱dota

2019-01-31

👍 2

`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

展开 ∨

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”



郝攀刚...

2019-01-30

👍 2

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!

作者回复: 😊

Explain下, 没用用index nested-loop 的全表优化



思考特~

2019-03-03

👍 1

老师, 这边想请教一个困扰很久的问题, 用mysql经常会制定这么一个规则, 不允许多表join。从实际情况看, 几乎不太可能遵守这个规则, 有个交易的业务场景涉及 用户表 300W、订单表 900W、支付表 900W, 每次需要查一个用户下面的订单信息可能就有点慢了, 但是还能接受, 如果是查询一个团体的订单信息, 这个量就非常可观了, 查询有时候根本返回不了结果。根本无法避免多表Join, 所以想问问老师, 在这种需要多表Join业务...
展开

作者回复: 我的建议就是用好NLJ和BKA算法😊



思考特~

2019-02-19

👍 1

老师, 下面的sql

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

使用BNL进行连接, t1表的记录存放到join buffer中去, t2表是要做全表扫描的, 它是将所有记录取出来, 这个记录应该也很大, 那么存放到什么地方?

展开

作者回复: 没有存, 就是按顺序读一行, 然后跟join buffer中对比以后这个内存就复用了, 又读下一行



403

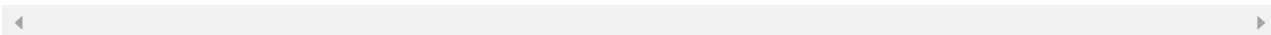
2019-02-09

👍 1

用那个作为驱动表，mysql会自己优化么？

展开 ▾

作者回复: 会的



剃刀吗啡

2019-01-31

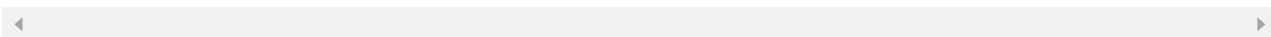
👍 1

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。

展开 ▾

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



13P

2019-01-30

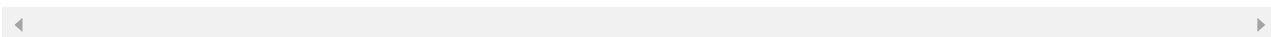
👍 1

老师您好：

您最后那个例子如果关联列是有索引的呢？就是使用Index Nested-Loop Join时，如何对比相对来说哪个是大表，哪个是小表，是对比两个表通过where条件筛选后的行数，哪个行数多哪个就是大表了？

展开 ▾

作者回复: 对的



700

2019-01-30

👍 1

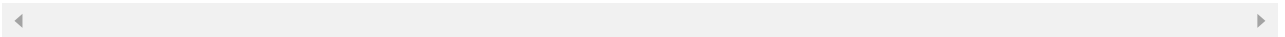
老师，您好。看完文章后有如下问题请教：

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？...

展开 ▾

- 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



amazon101...

2019-01-30

👍 1

这个专栏受益匪浅，老师再搞个内核源码专栏：)

展开 ▾



undefined

2019-01-30

👍 1

老师有几个问题：

1. select * 和 select 所有字段会有性能差异吗
2. 我们的业务有很多查询需要 join 很多个表，附带有动态的查询条件，几百万数据的表，查起来很慢，分多次查询聚合又要费很大功夫，现在的办法是给用到的没有索引的列加索引，像这种 SQL 该怎么优化呢...

展开 ▾

- 作者回复: 1. 没有，但是如果业务真的需要，还是建议写 “select 所有字段”
2. 下一篇会说哈，“现在的办法是给用到的没有索引的列加索引” 这个方法是可以的

