**浅谈JOIN在SQL语句中的查询优化**

SQL中的JOIN语句用于将数据库中的两个或多个表组合起来。由“连接”生成的集合，可以被保存为表，或者当成表来使用。JOIN语句的含义是把两张表的属性通过它们的值组合在一起。程序员在应用中经常用JOIN谓词表示要得到“连接”后的集合。但是在实际生产中，数据库管理员常常不让使用join语句，因为join语句的使用不当，通常会导致效率及其低下。然而，在一些场景下，对join语句的恰当使用也能起到事倍功半的效果。为了更加得心应手的运用join语句，笔者在此以mysql数据库为例对join在sql语句中的一些查询优化进行简单介绍。

为了实现对join语句的优化，必然要对join语句的执行机理有所了解。为了便于量化分析，下面采用实例对join语句的执行流程进行介绍。首先建立两个表table\_one和table\_two。表结构如图1所示：

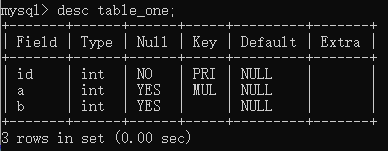


图1 表结构

两表结构一致，有主键id、字段a和字段b，皆为int类型，且在字段a上建有索引。然后在表table\_one中依次插入1000条数据：id、a、b的值都相同，由1至1000，在表table\_two中插入table\_one中的前100条数据。接下来我们对各种join语句的执行情况进行分析。在后文的SQL语句中，为了便于分析执行过程中的性能问题，排除MySQL优化器的影响，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。

select \* from table\_two straight\_join table\_one on (table\_one.a=table\_two.a);

通过explain命令，我们可以得到下面的分析图：

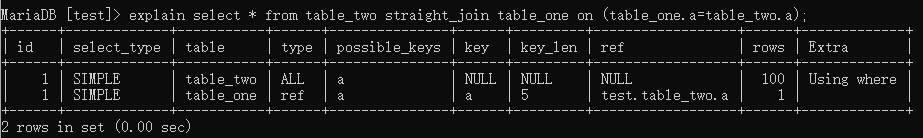


图2 使用索引字段join的 explain结果

可以看到，在这条语句里，驱动表为table\_two，被驱动表为table\_one，因为被驱动表table\_one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：从表table\_two中读入一行数据R，从数据行R中，取出a字段到表table\_one里去查找，取出表table\_one中满足条件的行，跟R组成一行，作为结果集的一部分。重复执行以上步骤，直到表table\_two的末尾循环结束。在形式上，这个过程就跟我们写程序时的嵌套查询类似，并且在查找满足的条件时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。在NLJ的流程里：对驱动表table\_two做了全表扫描，这个过程需要扫描100行；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。所以，整个执行流程，总扫描行数是200。

那么这个场景是否适合使用join语句呢？假设不使用join，那我们就只能用单表查询。首先执行select \* from table\_two，查出表table\_two的所有数据。循环遍历这些数据，从每一行R取出字段a的值，执行select \* from table\_one where a=R.a，把返回的结果和R构成结果集的一行。在文件disadvantageWithoutJoin.cpp中笔者采用C++语言实现上述与join相比繁琐很多的流程，具体实现详见代码文件。

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，还要自己拼接SQL语句和结果。显然，此处使用join效果更佳。

对这个场景进行泛化，假设被驱动表的行数是M，驱动表的行数是N。在这个join语句执行过程中，每在被驱动表查一行数据，要先搜索索引a，再搜索主键索引。被驱动表是走树搜索，每次搜索一棵树近似复杂度是以2为底的M的对数，记为，所以在被驱动表上查一行的时间复杂度是 2\*。而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\*。

可以看出，N对扫描行数的影响更大，因此为了优化join的查询效率，应该选用小表作为驱动表。

上述是对于可以使用被驱动表的索引的情况的分析，下面对于被驱动表不能使用索引的情况继续用例子进行介绍。

select \* from table\_two straight\_join table\_one on (table\_two.a=table\_one.b);

因为表table\_one的字段b上并没有建立索引，因此在对table\_one进行匹配时并不能采用树搜索，而是进行全表扫描。

同样运用explain命令进行分析，得到如下的分析图：

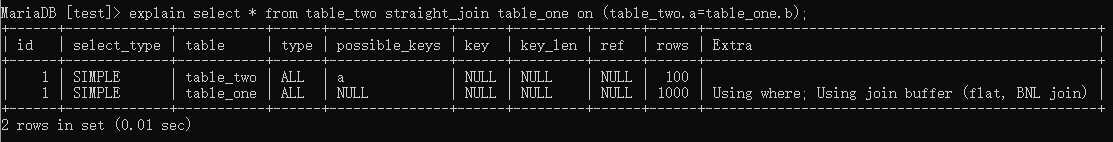


图3 不使用索引字段join的 explain结果

Mysql在这个场景下使用了 “Block Nested-Loop Join”的算法，简称BNL。BNL算法的流程为：首先把表table\_two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table\_two放入了内存，接着扫描表table\_one，把表table\_one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。在这个过程中，对表table\_one和table\_two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table\_one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\*1000=10万次。

同样再对这个场景泛化，假设被驱动表的行数是M，驱动表的行数是N。执行过程中，两个表都做一次全表扫描，扫描行数是M+N，而在内存中的判断次数是M\*N。看似这时候选择大表还是小表做驱动表，执行耗时是一样的。然而我们需要考虑到join\_buffer的大小对执行效率的影响，当join\_buffer放不下表table\_two的所有数据时，mysql采取的策略是分段放。假设需要分K段才能完成算法流程，并设K=λ\*N。此时在执行过程中，扫描行数变为N+λ\*N\*M，内存判断次数依旧是N\*M次。考虑扫描行数，在M和N大小确定的情况下，N小一些，结果会更小，因此应该让小表当驱动表。此外，当λ越小时，结果也越小，因此增大join\_buffer的大小也能提高join语句的执行效率。

上面我们对join语句执行过程中采用的两种算法进行了介绍， NLJ算法的效果不错，比通过应用层拆分成多个语句然后再拼接查询结果更方便，而且性能也不会差。但是，BNL算法在大表join的时候性能不佳，比较次数等于两个表参与join的行数的乘积，效率低下。不过，在了解了这两个算法的机理后，我们可以对它们继续优化。

为了便于分析，同样创建表t1、t2，t1、t2的结构与上文中的表结构一致。

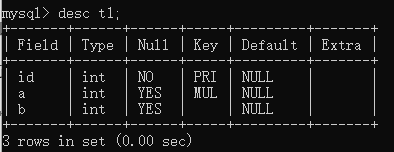


图4 表结构

在表t1里插入了1000行数据，每一行的a=1001-id，即表t1中字段a是逆序的。同时，我在表t2中插入了100万行顺序数据。

MySQL在5.6版本后引入了Batched Key Acess(BKA)算法，而这它就是对NLJ算法的优化。通过下面的参数设置语句：

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

可以开启mysql内置的对BKA优化算法的支持。BKA算法的优化依赖于Multi-Range Read(MRR)优化，因此下面首先举例对MRR算法进行介绍。

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

MRR优化的主要目的是尽量使用顺序读盘。根据InnoDB的索引结构，在普通索引a上查到主键id的值后，再根据一个个主键id的值到主键索引上去查整行数据。如果随着a的值递增顺序查询的话，id的值则不是顺序的，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，但是可以通过调整查询的顺序来对执行速度进行加速。因为大多数的数据都是按照主键递增顺序插入得到的，所以我们可以认为，如果按照主键的递增顺序进行查询的话，对磁盘的读取比较接近顺序读取，能够提升读性能。这就是MRR优化的设计思路。此时，语句的执行流程如下：首先根据索引a，定位到满足条件的记录，将id值放入read\_rnd\_buffer中。然后将read\_rnd\_buffer中的id进行递增排序。对排序后的id数组，依次到主键id索引中查记录，并作为结果返回。

对上面类似的sql语句，在开启MRR和关闭MRR的情况下分别执行，通过show profiles命令查看执行时间如下图：

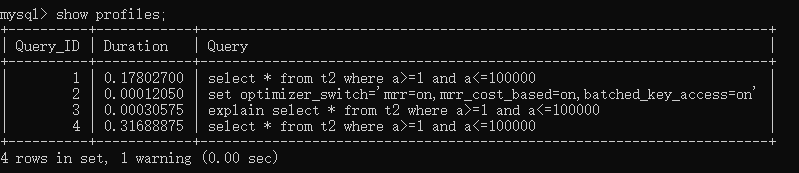


图5 开启MRR和关闭MRR执行时间

可以看到，在未开启MRR的情况下，执行时间接近于开启MRR情况下的两倍，开启MRR对效率的提升明显。MRR能够提升性能的核心在于，例子中的查询语句在索引a上做的是一个范围查询，可以得到足够多的主键id。这样通过排序以后，再去主键索引查数据，充分体现出了“顺序性”的优势。

在NLJ算法中，从驱动表t1，一行行地依次取出a的值，再到被驱动表t2去做join。即对于表t2来说，每次都只是匹配一个值。MRR的优势完全没有用上。BKA算法对于NLJ算法的优化之处就在于利用了MRR的顺序优势，它把表t1的数据取出来一部分，先放到临时内存join\_buffer中，然后一起传给表t2，从而利用到MRR的顺序优势。

说完了NLJ算法的优化，我们接下来介绍BNL算法的优化。

一些情况下，我们可以直接在被驱动表上建立索引，这时就可以直接转成BKA算法了。但是，有时候你确实会碰到一些不适合在被驱动表上建索引的情况。比如下面这个例子：

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

虽然表t2中有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，那么为这个语句在表t2的字段b上创建一个索引就很浪费了。

但是，不建立索引，使用BNL算法来join的话，这个语句的执行流程则是：把表t1的所有字段取出来，存入join\_buffer中。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, 再判断其他条件，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。上述流程中判断等值条件的次数是1000\*100万=10亿次，这个判断的工作量很大。

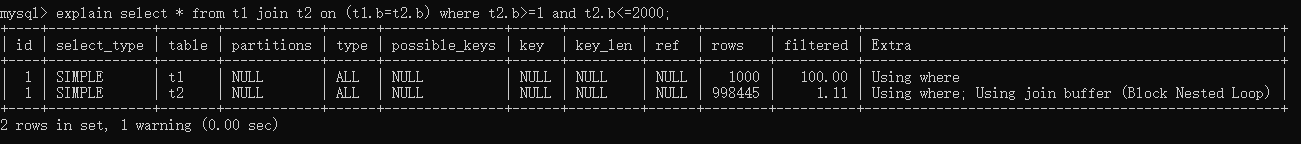


图6 BNL算法explain结果



图7 执行时长

可以看到，explain结果里Extra字段显示使用了BNL算法。在我的测试环境里，这条语句需要执行长达57.5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，如何解决这个问题呢？临时表是一个解决方案。使用临时表的大致思路是：把表t2中满足条件的数据放在临时表tmp\_t中，为了让join使用BKA算法，给临时表tmp\_t的字段b加上索引，然后再让表t1和tmp\_t做join操作。对应的SQL语句如下：

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);

执行耗时如下图所示：



图8 构造临时表耗时

可以看到，整个过程3个语句执行时间的总和还不到0.6秒，相比于前面的57.5秒，性能得到了大幅提升。这个过程的流程中，执行insert语句构造temp\_t表并插入数据，对表t2做了全表扫描，这里扫描行数是100万。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，这个优化效果还是很明显的。

实际上，上面的解决方案并不是最佳方案。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，那么就不再需要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就快多了。然而在本次实验中所使用的MySQL版本为5.7，其还未支持哈希join。而在MySQL 8.0.18 版本增加了新的特性hash join，能使效率进一步提升。

总体来看，不论是在原表上加索引，还是用有索引的临时表，我们的思路都是让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。

综上，如果可以使用被驱动表的索引，join语句还是有其优势的。对于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。在使用join的时候，应该让小表做驱动表，来提高效率。BKA优化是MySQL已经内置支持的，建议默认使用。而BNL算法效率低，尽量转成BKA算法。优化的方向就是给被驱动表的关联字段加上索引，基于临时表的改进方案，对于能够提前过滤出小数据的join语句来说，效果不错。而hash join理论上效果还要好于临时表的方案。