08 Join语句可以这样优化

更新时间：2019-08-13 14:08:27

> 世上无难事,只要肯登攀。
>
> ——毛泽东

我们在使用数据库查询数据时，有时一张表并不能满足我们的需求，很多时候都涉及到多张表的连接查询。今天，我们就一起研究关联查询的一些优化技巧。在说关联查询优化之前，我们先看下跟关联查询有关的几个算法：

为了方便理解，首先创建测试表并写入测试数据，语句如下：

```
CREATE DATABASE muke; /* 创建测试使用的database，名为muke */
use muke; /* 使用muke这个database */
drop table if exists t1; /* 如果表t1存在则删除表t1 */
CREATE TABLE `t1` ( /* 创建表t1 */
`id` int(11) NOT NULL auto_increment,
`a` int(11) DEFAULT NULL,
`b` int(11) DEFAULT NULL,
`create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '记录创建时间',
`update_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
COMMENT '记录更新时间',
PRIMARY KEY (`id`),
KEY `idx_a` (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

drop procedure if exists insert_t1; /* 如果存在存储过程insert_t1，则删除 */

delimiter ;;
create procedure insert_t1() /* 创建存储过程insert_t1 */
begin
declare i int; /* 声明变量i */
set i=1; /* 设置i的初始值为1 */
while(i<=10000)do /* 对满足i<=10000的值进行while循环 */
insert into t1(a,b) values(i, i); /* 写入表t1中a、b两个字段，值都为i当前的值 */
set i=i+1; /* 将i加1 */
end while;
end;;
delimiter ; /* 创建批量写入10000条数据到表t1的存储过程insert_t1 */
call insert_t1(); /* 运行存储过程insert_t1 */
drop table if exists t2; /* 如果表t2存在则删除表t2 */
create table t2 like t1; /* 创建表t2，表结构与t1一致 */
insert into t2 select * from t1 limit 100; /* 将表t1的前100行数据导入到t2 */
```

1 关联查询的算法

MySQL 使用以下两种嵌套循环算法或它们的变体在表之间执行连接（参考 《MySQL 5.7 Reference Manual》8.2.1.6 Nested-Loop Join Algorithms）：

- Nested-Loop Join 算法
- Block Nested-Loop Join 算法

另外还有一种算法 Batched Key Access，其实算对 Nested-Loop Join 算法的一种优化。

下面我们就看下这些算法的设计思想：

1.1 Nested-Loop Join 算法

一个简单的 Nested-Loop Join(NLJ) 算法一次一行循环地从第一张表（称为驱动表）中读取行，在这行数据中取到关联字段，根据关联字段在另一张表（被驱动表）里取出满足条件的行，然后取出两张表的结果合集。

我们试想一下，如果在被驱动表中这个关联字段没有索引，那么每次取出驱动表的关联字段在被驱动表查找对应的数据时，都会对被驱动表做一次全表扫描，成本是非常高的（比如驱动表数据量是 m，被驱动表数据量是 n，则扫描行数为 m * n ）。

好在 MySQL 在关联字段有索引时，才会使用 NLJ，如果没索引，就会使用 Block Nested-Loop Join，等下会细说这个算法。我们先来看下在有索引情况的情况下，使用 Nested-Loop Join 的场景（称为：Index Nested-Loop Join）。

因为 MySQL 在关联字段有索引时，才会使用 NLJ，因此本节后面的内容所用到的 NLJ 都表示 Index Nested-Loop Join。

如下例：

```
select * from t1 inner join t2 on t1.a = t2.a;      /* sql1 */
```

> Tips：表 t1 和表 t2 中的 a 字段都有索引。

怎么确定这条 SQL 使用的是 NLJ 算法？

我们先来看下 sql1 的执行计划：

```
mysql> explain select * from t1 inner join t2 on t1.a = t2.a;
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key   | key_len | ref       | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
|  1 | SIMPLE      | t2    | NULL       | ALL  | idx_a         | NULL  | NULL    | NULL      |  100 |   100.00 | Using where |
|  1 | SIMPLE      | t1    | NULL       | ref  | idx_a         | idx_a | 5       | muke.t2.a |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

从执行计划中可以看到这些信息：

- 驱动表是 t2，被驱动表是 t1。原因是：explain 分析 join 语句时，在第一行的就是驱动表；选择 t2 做驱动表的原因：如果没固定连接方式（比如没加 straight_join）优化器会优先选择小表做驱动表。**所以使用 inner join 时，前面的表并不一定就是驱动表。**
- 使用了 NLJ。原因是：一般 join 语句中，如果执行计划 Extra 中未出现 Using join buffer （***）；则表示使用的 join 算法是 NLJ。

sql1 的大致流程如下：

1. 从表 t2 中读取一行数据；
2. 从第 1 步的数据中，取出关联字段 a，到表 t1 中查找；
3. 取出表 t1 中满足条件的行，跟 t2 中获取到的结果合并，作为结果返回给客户端；
4. 重复上面 3 步。

在这个过程中会读取 t2 表的所有数据，因此这里扫描了 100 行，然后遍历这 100 行数据中字段 a 的值，根据 t2 表中 a 的值索引扫描 t1 表中的对应行，这里也扫描了 100 行。因此整个过程扫描了 200 行。

在前面，我们有说到：如果被驱动表的关联字段没索引，就会使用 Block Nested-Loop Join(简称：BNL)，为什么会选择使用 BNL 算法而不继续使用 Nested-Loop Join呢？下面就一起分析下：

1.2 Block Nested-Loop Join 算法

Block Nested-Loop Join(BNL) 算法的思想是：把驱动表的数据读入到 join_buffer 中，然后扫描被驱动表，把被驱动表每一行取出来跟 join_buffer 中的数据做对比，如果满足 join 条件，则返回结果给客户端。

我们一起看看下面这条 SQL 语句：

```
select * from t1 inner join t2 on t1.b = t2.b;      /* sql2 */
```

> Tips：表 t1 和表 t2 中的 b 字段都没有索引

看下执行计划：

```
mysql> explain select * from t1 inner join t2 on t1.b = t2.b;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | t2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  100 |   100.00 | NULL                                               |
|  1 | SIMPLE      | t1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9963 |    10.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

在 Extra 发现 Using join buffer (Block Nested Loop)，这个就说明该关联查询使用的是 BNL 算法。

我们再看下 sql2 的执行流程：

1. 把 t2 的所有数据放入到 join_buffer 中
2. 把表 t1 中每一行取出来，跟 join_buffer 中的数据做对比
3. 返回满足 join 条件的数据

在这个过程中，对表 t1 和 t2 都做了一次全表扫描，因此扫描的总行数为10000(表 t1 的数据总量) + 100(表 t2 的数据总量) = 10100。并且 join_buffer 里的数据是无序的，因此对表 t1 中的每一行，都要做 100 次判断，所以内存中的判断次数是 100 * 10000= 100 万次。

下面我们来回答上面提出的一个问题：

如果被驱动表的关联字段没索引，为什么会选择使用 BNL 算法而不继续使用 Nested-Loop Join 呢？

在被驱动表的关联字段没索引的情况下，比如 sql2：

如果使用 Nested-Loop Join，那么扫描行数为 100 * 10000 = 100万次，这个是磁盘扫描。

如果使用 BNL，那么磁盘扫描是 100 + 10000=10100 次，在内存中判断 100 * 10000 = 100万次。

显然后者磁盘扫描的次数少很多，因此是更优的选择。因此对于 MySQL 的关联查询，如果被驱动表的关联字段没索引，会使用 BNL 算法。

1.3 Batched Key Access 算法

在学了 NLJ 和 BNL 算法后，你是否有个疑问，如果把 NLJ 与 BNL 两种算法的一些优秀的思想结合，是否可行呢？

比如 NLJ 的关键思想是：被驱动表的关联字段有索引。

而 BNL 的关键思想是：把驱动表的数据批量提交一部分放到 join_buffer 中。

从 MySQL 5.6 开始，确实出现了这种集 NLJ 和 BNL 两种算法优点于一体的新算法：Batched Key Access(BKA)。

其原理是：

1. 将驱动表中相关列放入 join_buffer 中
2. 批量将关联字段的值发送到 Multi-Range Read(MRR) 接口
3. MRR 通过接收到的值，根据其对应的主键 ID 进行排序，然后再进行数据的读取和操作
4. 返回结果给客户端

> 这里补充下 **MRR** 相关知识：
>
> 当表很大并且没有存储在缓存中时，使用辅助索引上的范围扫描读取行可能导致对表有很多随机访问。
>
> 而 Multi-Range Read 优化的设计思路是：查询辅助索引时，对查询结果先按照主键进行排序，并按照主键排序后的顺序，进行顺序查找，从而减少随机访问磁盘的次数。
>
> 使用 MRR 时，explain 输出的 Extra 列显示的是 Using MRR。
>
> optimizer_switch 中 mrr_cost_based 参数的值会影响 MRR。
>
> 如果 mrr_cost_based=on，表示优化器尝试在使用和不使用 MRR 之间进行基于成本的选择。
>
> 如果 mrr_cost_based=off，表示一直使用 MRR。
>
> 更多 MRR 信息请参考官方手册：https://dev.mysql.com/doc/refman/5.7/en/mrr-optimization.html。

下面尝试开启 BKA ：

```
set optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
```

这里对上面几个参数做下解释：

- mrr=on 开启 mrr
- mrr_cost_based=off 不需要优化器基于成本考虑使用还是不使用 MRR，也就是一直使用 MRR
- batched_key_access=on 开启 BKA

然后再看 sql1 的执行计划：

```
explain select * from t1 inner join t2 on t1.a = t2.a;
```

```
mysql> explain select * from t1 inner join t2 on t1.a = t2.a;
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key   | key_len | ref       | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | t2    | NULL       | ALL  | idx_a         | NULL  | NULL    | NULL      |  100 |   100.00 | Using where                                        |
|  1 | SIMPLE      | t1    | NULL       | ref  | idx_a         | idx_a | 5       | muke.t2.a |    1 |   100.00 | Using join buffer (Batched Key Access)             |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

在 Extra 字段中发现有 Using join buffer (Batched Key Access)，表示确实变成了 BKA 算法。

2 优化关联查询

通过上面的知识点，我们知道了关联查询的一些算法，下面一起来讨论下关联查询的优化：

2.1 关联字段添加索引

通过上面的内容，我们知道了 BNL、NLJ 和 BKA 的原理，因此让 BNL变成 NLJ 或者 BKA，可以提高 join 的效率。我们来看下面的例子

我们构造出两个算法对于的例子：

Block Nested-Loop Join 的例子：

```
select * from t1 join t2 on t1.b= t2.b;
```

需要 0.08 秒。

Index Nested-Loop Join 的例子：

```
select * from t1 join t2 on t1.a= t2.a;
```

只需要 0.01 秒。

再对比一下两条 SQL 的执行计划：

```
mysql> explain select * from t1 join t2 on t1.b= t2.b;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | t2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  100 |   100.00 | NULL                                               |
|  1 | SIMPLE      | t1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9963 |    10.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)

mysql> explain select * from t1 join t2 on t1.a= t2.a;
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key   | key_len | ref       | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
|  1 | SIMPLE      | t2    | NULL       | ALL  | idx_a         | NULL  | NULL    | NULL      |  100 |   100.00 | Using where |
|  1 | SIMPLE      | t1    | NULL       | ref  | idx_a         | idx_a | 5       | muke.t2.a |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

前者扫描的行数是 100 和 9963。

后者扫描的行数是 100 和 1。

对比执行时间和执行计划，再结合在本节开始讲解的两种算法的执行流程，很明显 Index Nested-Loop Join 效率更高。

> Tips:因为在写入数据时使用的是 insert into t1(a,b) values(i, i); 因此 a、b 两个字段的值是相等的，因此这个实验的两条 SQL 虽然使用的是不同关联字段，但是实际相当于同一个字段添加索引前后的状态。

因此建议在被驱动表的关联字段上添加索引，让 BNL变成 NLJ 或者 BKA，可明显优化关联查询。

2.2 小表做驱动表

前面说到，Index Nested-Loop Join 算法会读取驱动表的所有数据，首先扫描的行数是驱动表的总行数（假设为 n），然后遍历这 n 行数据中关联字段的值，根据驱动表中关联字段的值索引扫描被驱动表中的对应行，这里又会扫描 n 行，因此整个过程扫描了 2n 行。当使用 Index Nested-Loop Join 算法时，扫描行数跟驱动表的数据量成正比。所以在写 SQL 时，如果确定被关联字段有索引的情况下，建议用小表做驱动表。

我们来看下以 t2 为驱动表的 SQL：

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

这里使用 straight_join 可以固定连接方式，让前面的表为驱动表。

再看下以 t1 为驱动表的 SQL：

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

我们对比下两条 SQL 的执行计划：

```
mysql> explain select * from t2 straight_join t1 on t2.a = t1.a;
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key   | key_len | ref       | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
|  1 | SIMPLE      | t2    | NULL       | ALL  | idx_a         | NULL  | NULL    | NULL      |  100 |   100.00 | Using where |
|  1 | SIMPLE      | t1    | NULL       | ref  | idx_a         | idx_a | 5       | muke.t2.a |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

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  | idx_a         | NULL  | NULL    | NULL      | 9963 |   100.00 | Using where |
|  1 | SIMPLE      | t2    | NULL       | ref  | idx_a         | idx_a | 5       | muke.t1.a |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

明显前者扫描的行数少（注意关注 explain 结果的 rows 列），所以**建议小表驱动大表**。

2.3 临时表

多数情况我们可以通过在被驱动表的关联字段上加索引来让 join 使用 NLJ 或者 BKA，但有时因为某条关联查询只是临时查一次，如果再去添加索引可能会浪费资源，那么有什么办法优化呢？

这里提供一种创建临时表的方法。

我们一起测试下：

比如下面这条关联查询：

```
select * from t1 join t2 on t1.b= t2.b;
```

我们看下执行计划：

```
mysql> explain select * from t1 join t2 on t1.b= t2.b;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | t2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL |  100 |   100.00 | NULL                                               |
|  1 | SIMPLE      | t1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9963 |    10.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

由于表 t1 和表 t2 的字段 b都没索引，因此使用的是效率比较低的 BNL 算法。

现在用临时表的方法对这条 SQL 进行优化：

首先创建临时表 t1_tmp，表结构与表 t1 一致，只是在关联字段 b 上添加了索引。

```
CREATE TEMPORARY TABLE `t1_tmp` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  `create_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '记录创建时间',
  `update_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '记录更新时间',
  PRIMARY KEY (`id`),
  KEY `idx_a` (`a`),
  KEY `idx_b` (b)
) ENGINE=InnoDB ;
```

把 t1 表中的数据写入临时表 t1_tmp 中：

```
insert into t1_tmp select * from t1;
```

执行 join 语句：

```
select * from t1_tmp join t2 on t1_tmp.b= t2.b;
```

我们再看下执行计划：

```
mysql> explain select * from t1_tmp join t2 on t1_tmp.b= t2.b;
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key    | key_len | ref       | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
|  1 | SIMPLE      | t2     | NULL       | ALL  | NULL          | NULL   | NULL    | NULL      |  100 |   100.00 | Using where |
|  1 | SIMPLE      | t1_tmp | NULL       | ref  | idx_b         | idx_b | 5       | muke.t2.b |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+------+---------------+-------+---------+-----------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

Extra 没出现 “Block Nested Loop”，说明使用的是 Index Nested-Loop Join，并且扫描行数也大大降低了。

所以当遇到 BNL 的 join 语句，如果不方便在关联字段上添加索引，不妨尝试创建临时表，然后在临时表中的关联字段上添加索引，然后通过临时表来做关联查询。

3 总结

本节首先讲到了 NLJ 、BNL、和 BKA 这几种 join 算法的原理，然后通过认识这些算法，从而引申出 join 语句的一些优化技巧，比如关联字段添加索引、小表做驱动表和创建临时表等方法。

4 问题

哪种情况下，小表做驱动表跟大表做驱动表的执行效率是一样的？

5 参考资料

《MySQL 5.7 Reference Manual》

8.2.1.6 Nested-Loop Join Algorithms:https://dev.mysql.com/doc/refman/5.7/en/nested-loop-joins.html

8.2.1.10 Multi-Range Read Optimization:https://dev.mysql.com/doc/refman/5.7/en/mrr-optimization.html

8.2.1.11 Block Nested-Loop and Batched Key Access Joins:https://dev.mysql.com/doc/refman/5.7/en/bnl-bka-optimization.html

}

07 换种思路写分页查询

09 为何count(*)这么慢?