MySQL学习笔记（Day028：join算法/锁_1）

MySQL学习

一. JOIN算法

1.1. JOIN 语法

```
mysql> select * from t4;
+---+------+
| a | b    |
+---+------+
| 1 |    1 |
| 2 |   11 |
| 3 |   12 |
| 5 |   50 |
+---+------+
4 rows in set (0.00 sec)

mysql> select * from t5;
+------+------+
| a    | b    |
+------+------+
|    2 |    2 |
+------+------+
1 row in set (0.00 sec)

--
-- 语法一
--
mysql> select * from t4, t5 where t4.a=t5.a;
+---+------+------+------+
| a | b    | a    | b    |
+---+------+------+------+
| 2 |   11 |    2 |    2 |
+---+------+------+------+
1 row in set (0.00 sec)

mysql> explain select * from t4, t5 where t4.a=t5.a;
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref            | rows | filtered | Extra       |
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
|  1 | SIMPLE      | t5    | NULL       | ALL    | NULL          | NULL    | NULL    | NULL           |    1 |   100.00 | Using where |
|  1 | SIMPLE      | t4    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | burn_test.t5.a |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

--
-- 语法二
--
mysql> select * from t4 inner join t5 on t4.a=t5.a;
+---+------+------+------+
| a | b    | a    | b    |
+---+------+------+------+
| 2 |   11 |    2 |    2 |
+---+------+------+------+
1 row in set (0.00 sec)

mysql> explain select * from t4 inner join t5 on t4.a=t5.a;
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref            | rows | filtered | Extra       |
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
|  1 | SIMPLE      | t5    | NULL       | ALL    | NULL          | NULL    | NULL    | NULL           |    1 |   100.00 | Using where |
|  1 | SIMPLE      | t4    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | burn_test.t5.a |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

--
-- 语法三
--
mysql> select * from t4 join t5 on t4.a=t5.a;
+---+------+------+------+
| a | b    | a    | b    |
+---+------+------+------+
| 2 |   11 |    2 |    2 |
+---+------+------+------+
1 row in set (0.00 sec)

mysql> explain select * from t4 join t5 on t4.a=t5.a;
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref            | rows | filtered | Extra       |
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
|  1 | SIMPLE      | t5    | NULL       | ALL    | NULL          | NULL    | NULL    | NULL           |    1 |   100.00 | Using where |
|  1 | SIMPLE      | t4    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | burn_test.t5.a |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+--------+---------------+---------+---------+----------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

> 通过上述的 `EXPLAIN` 可以得知，三种 `JOIN` 语法在执行 性能 和 效果 上都是 一样 的。

1.2. JOIN算法

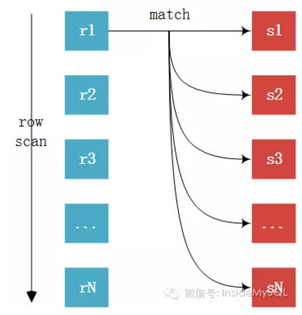
- **nsted_loop join**
  1. `simple nested-loop join`
  2. `index nested-loop join`
  3. `block nested-loop join`

1.2.1. simple nested loop join

`simple nested_loog join` 算法可以理解成 两个for循环 ，外循环走一次，内循环走N次（N=外循环的次数）

其算法伪代码如下：

```
For each row r in R do   # 扫描R表
  Foreach row s in S do  #  扫描S表
    If r and s satisfy the join condition  # 如果r和s满足join条件
      Then output the tuple   # 那就输出结果集
```

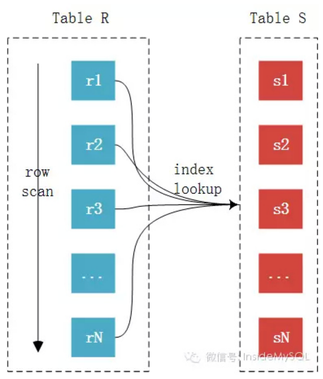


1. `R` 表，该表只扫描了一次；
2. `S` 表，该表扫面了 `count(R)` 次；
3. 该方法相当于是一个笛卡尔积，实际上数据库 不会使用 该算法；

1.2.2. index nested loop join

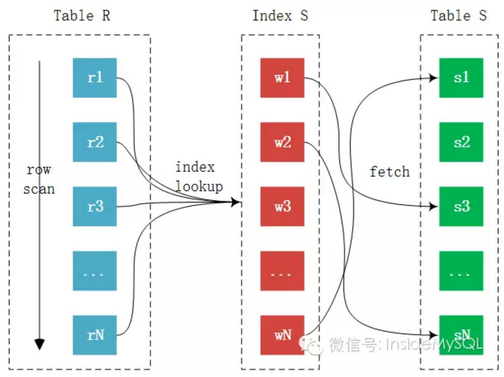
`index nested_loop join` 算法是将 外表扫描一次 ，内表不会直接去扫描，而是查找 内表 相应的 索引 的方式，和外表的记录进行匹配

```
For each row r in R do  # 扫描R表
 lookup s in S index    # 查询S表的索引（固定3~4次IO，B+树高度）
   if found s == r      # 如果 r匹配了索引s
      Then output the tuple    # 输出结果集
```



1. S表上有索引
2. 扫描R表，将R表的记录和S表中的索引进行匹配
3. R表上可以没有索引
4. 优化器倾向使用 记录数少 的表作为外表（又称驱动表）

如果数据量大，index nested loop join的成本也是高的，尤其是在二级索引的情况下，需要大量的回表操作

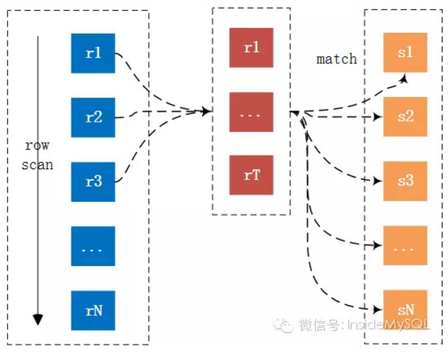


1.2.3. block nested loop join

`block nested loop join` 将外表中的 需要join匹配的列 （不是完整的记录）暂时保存在一块内存（ `join buffer` ）中，让后将这块内存中的数据和内表进行匹配（内表只扫描一次）

`block nested loop join` 可被用于联接的是ALL，index，range的类型

```
For each tuple r in R do
  store used columns as p from R in join buffer  # 将部分或者全部R的记录保存到 join buffer中，记为p
  For each tuple s in S do
    If p and s satisfy the join condition # p 与 s满足join条件
      Then output the tuple    # 输出为结果集
```



`block nested loop join` 与 `simple nested loop join` 相比，多了一个 `join buffer`

```
mysql> show variables like "%join%buffer%";
+------------------+-----------+
| Variable_name    | Value     |
+------------------+-----------+
| join_buffer_size | 134217728 | -- 128M，默认是256K
+------------------+-----------+
1 row in set (0.00 sec)
```

> `join buffer` 用的不是Buffer Pool中的内存，而是 线程级别 的内存。
> 可以通过explain查看执行计划，并通过 join条件字段 的大小，来预估 join_buffer_size 的大小。

注意：
增大**`join_buffer_size`** **进行优化的前提是** 没有使用**`index`** **，如果使用了index，根本** 不会 **使用block nested join算法**

1.2.4. block join与simple join比较

- **外表A：a b c**
- **内表B：1 2 3 4**

```
                Table_A : a   b   c
                          +
 Simple join              |
                          +---+---+---+
                          v   v   v   v  # a b c 分别来访问一次 1 2 3 4
                Table_B : 1   2   3   4

+-----------------------------------------------------------+

                Table_A : a   b   c
                   +
                   |
                   |         +--------------+
                   +------>  |   a   b   c  |  join buffer
    Block join               +---+---+---+--+
                                 |   |   |
                             +---<---<---+  # （a b c） 组团 访问 1 2 3 4
                             |
                             v
                Table_B : 1   2   3   4
```

join算法	外表A扫描次数	内表B的扫描次数	比较次数
simple	1	3	12
block	1	1	12

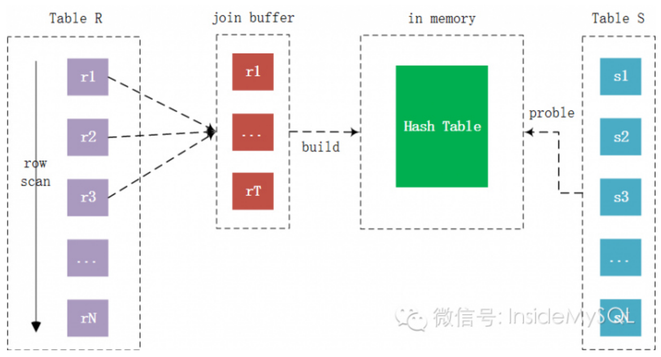
> 这里作为演示，将A表的数据都放进了join buffer中，如果join buffer中一次性 放不下 A表的数据，那 B表 还是要被 扫描多次 ；
>
> 假设A表有 1000W 条数据，join buffer能存放 10W 条数据，那B表需要被扫描 100 次

1.2.5. 几种join算法的开销对比

开销统计	SNLJ	INLJ	BNLJ
外表扫描次数：O	1	1	1
内表扫描次数：I	R	0	R*used_column_size/ join_buffer_size + 1
读取记录数：R	R + S*R	R + Smatch	R + S*I
Join比较次数：M	S*R	R * IndexHeight	S*R
回表读取记录次数：F	0	Smatch (if possible)	0

1.2.6. MariaDB中的hash join算法

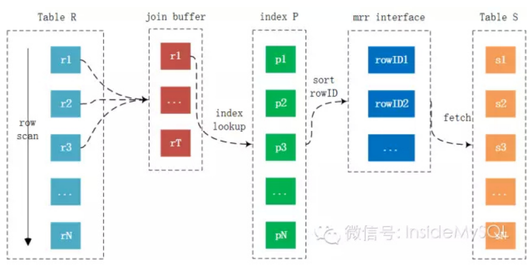
MySQL 目前 不支持 hash join



由上图可知， `MariaDB` 中的 `hash join` 算法是在 `block join` 基础之上，根据join buffer中的对象创建哈希表，内表通过哈希算法进行查找，减少内外表扫描的次数，提升join的性能

> MariaDB中的hash join问题是，优化器默认 不会 去选择hash join算法
> `set join_cache_level = 4;`
> `set optimizer_switch='join_cache_hashed=on';`
> 设置两个变量后，MariaDB将 强制 使用hash算法，无法智能判断

1.2.7. BKA join（batched key access join）



> 通过上图可以知道，在index join的基础上，增加MRR的功能，先对索引进行排序，然后批量获取聚集索引中的记录
> 由于使用了MRR的功能，所以默认该join算法也是不会被使用到的
> `set optimizer_switch='mrr_cost_based=off';`

```
-- 方法一
mysql> SET optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';  -- 在session中打开BKA特性

mysql> explain SELECT * FROM part, lineitem WHERE l_partkey=p_partkey AND p_retailprice>3000\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: part
   partitions: NULL
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 195802
     filtered: 33.33
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: lineitem
   partitions: NULL
         type: ref
possible_keys: i_l_suppkey_partkey,i_l_partkey
          key: i_l_partkey
      key_len: 5
          ref: dbt3.part.p_partkey
         rows: 27
     filtered: 100.00
        Extra: Using join buffer (Batched Key Access)  -- 使用了BKA
2 rows in set, 1 warning (0.00 sec)

-- 方法二

mysql> SET optimizer_switch='mrr=on,mrr_cost_based=on,batched_key_access=off'; -- 在session中关闭BKA特性

mysql> explain SELECT /*+ BKA(lineitem)*/ * FROM part, lineitem WHERE l_partkey=p_partkey AND p_retailprice>2050\G  -- 使用/*+ BKA(tablename)*/ 的语法，强制使用BKA特性
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: part
   partitions: NULL
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 195802
     filtered: 33.33
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: lineitem
   partitions: NULL
         type: ref
possible_keys: i_l_suppkey_partkey,i_l_partkey
          key: i_l_partkey
      key_len: 5
          ref: dbt3.part.p_partkey
         rows: 27
     filtered: 100.00
        Extra: Using join buffer (Batched Key Access)  -- 使用了BKA
2 rows in set, 1 warning (0.00 sec)
```

二. 锁（一）

2.1. 锁的介绍

- **什么是锁**
  - 对共享资源进行并发访问
  - 提供数据的完整性和一致性
- **每个数据库的锁的实现完全不同**
  - MyISAM表锁
  - InnoDB行锁（与Oracle的行锁不同）
  - MSSQL 行级锁 with 锁升级
- **latch**
  - mutex
  - rw-lock
- **锁的区别**

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过waits-for graph、time out等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（latch leveling）保证无死锁的情况发生
存在于	Lock Manager的哈希表中	每个数据结构的对象中

> `latch` 是针对程序内部的资源（*比如：全局变量*）的锁的定义，而这里的 `lock` 针对的是数据库的 事物
>
> `lock` 由 `latch` 来保证和实现

2.2. 锁的查看

```
mysql> show engine innodb mutex;  -- 主要给内核开发人员给予帮助
+--------+-----------------------------+-----------+
| Type   | Name                        | Status    |
+--------+-----------------------------+-----------+
| InnoDB | rwlock: dict0dict.cc:1184   | waits=2   |
| InnoDB | rwlock: log0log.cc:785      | waits=21  |
| InnoDB | sum rwlock: buf0buf.cc:1379 | waits=138 |
+--------+-----------------------------+-----------+
3 rows in set (0.01 sec)
```

2.3. 锁的类型

1. S 行级共享锁
2. X 行级排它锁锁
3. IS
4. IX
5. AI 自增锁

三. MRR补充

```
mysql> show variables like "%optimizer_switch%"\G
*************************** 1. row ***************************
Variable_name: optimizer_switch
        Value: index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,engine_condition_pushdown=on,index_condition_pushdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_access=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,duplicateweedout=on,subquery_materialization_cost_based=on,use_in
dex_extensions=on,condition_fanout_filter=on,derived_merge=on
1 row in set (0.00 sec)

mysql> explain select  * from employees where hire_date >= '1991-01-01'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
   partitions: NULL
         type: ALL
possible_keys: idx_date
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 298124
     filtered: 50.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)

mysql> explain select /*+ MRR(employees)*/ * from employees where hire_date >= '1991-01-01'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
   partitions: NULL
         type: range
possible_keys: idx_date
          key: idx_date
      key_len: 3
          ref: NULL
         rows: 149062
     filtered: 100.00
        Extra: Using index condition; Using MRR
1 row in set, 1 warning (0.00 sec)
```

> 如果强制开启MRR，那在某些SQL语句下，性能可能会变差；因为 `MRR需要排序` ，假如排序的时间超过直接执行的时间，那性能就会降低。
>
> optimizer_switch 可以是全局的，也可以是会话级的。