青石路

唯有读书和健身不可辜负



# 神奇的 SQL 之 联表细节 → MySQL JOIN 的执行过程（二）

## 开心一刻

一头母牛在吃草，突然一头公牛从远处狂奔而来说：“快跑啊！！楼主来了！”

母牛说：“楼主来了关我屁事啊？”

公牛急忙说：“楼主吹牛逼呀！”

母牛大惊，拔腿就跑，边跑边问：“你是公牛你怕什么啊？”

公牛无奈道：“现在的楼主不仅吹牛逼，还扯蛋！”

然后小牛也在跟着跑，公牛和母牛问：“儿子你跑什么呢？”

小牛说：“楼主还扯犊子啊”


## 前情回顾

神奇的 SQL 之 联表细节 → MySQL JOIN 的执行过程（一）中，我们讲到了 JOIN 的部分内容，像：驱动表、JOIN 大致流程等。什么，还没看？赶紧去看呀，啊？ 你都知道呀，那你走吧


走就走，你把欠的内容还上我就走；我欠什么了？ 我欠，我欠... 我好像是欠点东西

1、BKA（Batched Key Access）

2、ON 和 WHERE

请各位坐好，我要开始表演了


## 环境准备

数据库： `MySQL 5.7.1`

存储引擎： `InnoDB`

建表和初始化数据

View Code

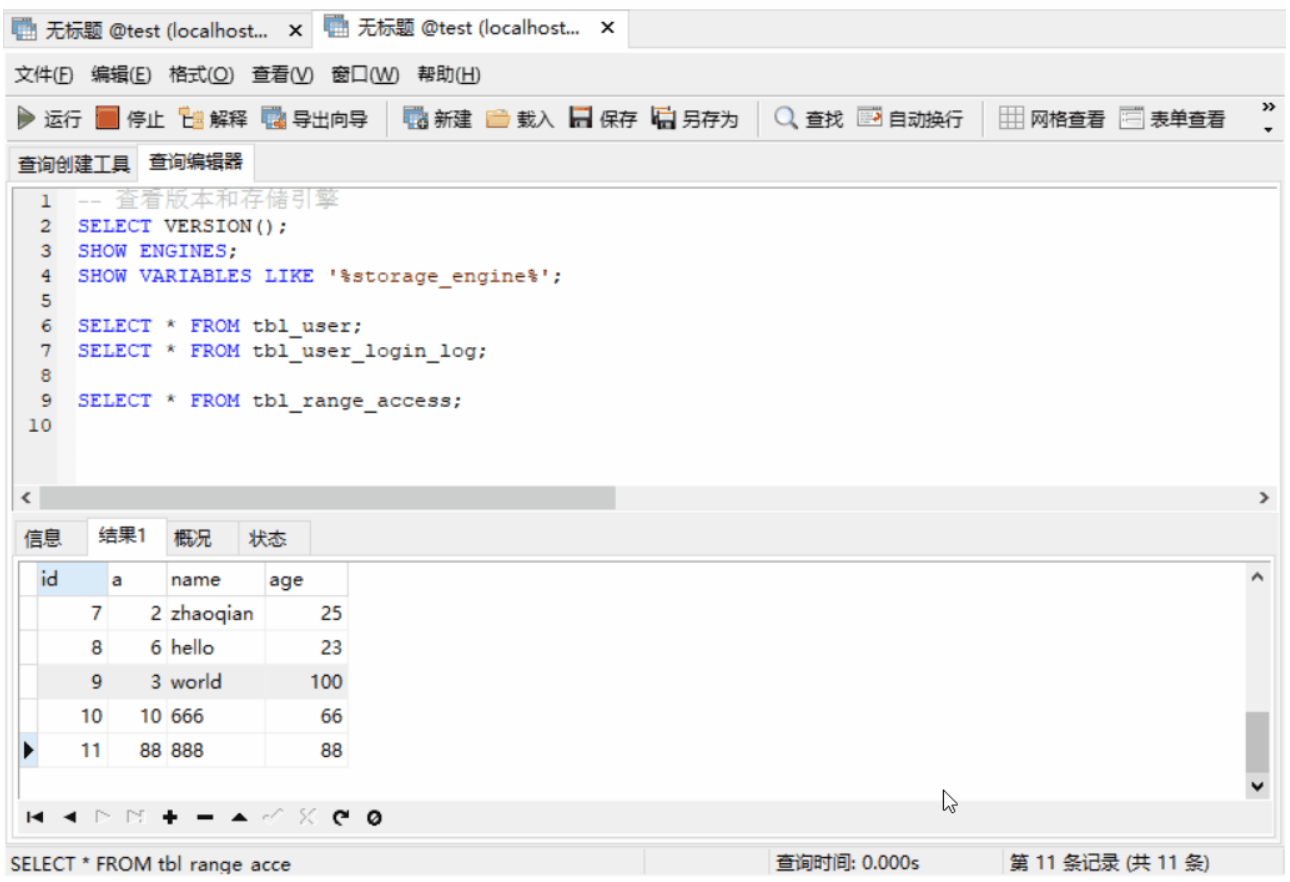



表 `tbl_range_access` 的数据要多一点，像上面示例只有 11 条记录，那么即使 a 字段上有索引， `SELECT * FROM tbl_range_access WHERE a BETWEEN 4 AND 9;` 也不会走索引，执行计划如下

```
7 EXPLAIN SELECT * FROM tbl_range_access WHERE a BETWEEN 4 AND 9;
```

信息 结果1 概况 状态

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | tbl_range | (Null) | ALL | i_a | (Null) | (Null) | (Null) | 11 | 54.55 | Using where |

数据太少，优化器觉得走索引，然后回表查询数据，还不如直接走聚簇索引全表查询来的快，所以没有选择走索引 `i_a`

既然数据太少，我们就多造点数据，运行 data-init 下的 `RangeAccessTest.java` 中的 `batchAddData` 方法就好，轻轻松松 10W 到手！ 此时执行计划如下

```
7 EXPLAIN SELECT * FROM tbl_range_access WHERE a BETWEEN 4 AND 9;
```

信息 结果1 概况 状态

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | tbl_range | (Null) | range | i_a | i_a | 4 | (Null) | 45 | 100 | Using index condition |

## MRR

讲 BKA 之前了，我们不得不先看下 MRR，它是 BKA 的重要支柱

全称 `Multi-Range Read` ，是对多行 IO 查询进行优化的一种策略，详情可看 MySQL 的 mrr-optimization 或者 MariaDB 的 Multi Range Read Optimization（MySQL 和 MariaDB 是什么关系?



quartz(3)
更多

随笔档案

2023年11月(2)
2023年10月(2)
2023年9月(1)
2023年7月(1)
2023年6月(1)
2023年4月(2)
2023年3月(1)
2023年1月(1)
2022年12月(1)
2022年11月(2)
2022年9月(2)
2022年8月(1)
2022年7月(2)
2022年6月(1)
2022年5月(2)
更多

阅读排行榜

1. nginx实现请求的负载均...
2. Maven pom.xml中的元...
3. spring jdbcTemplate ...
4. 利用maven/eclipse搭...
5. spring-session实现分...

推荐排行榜

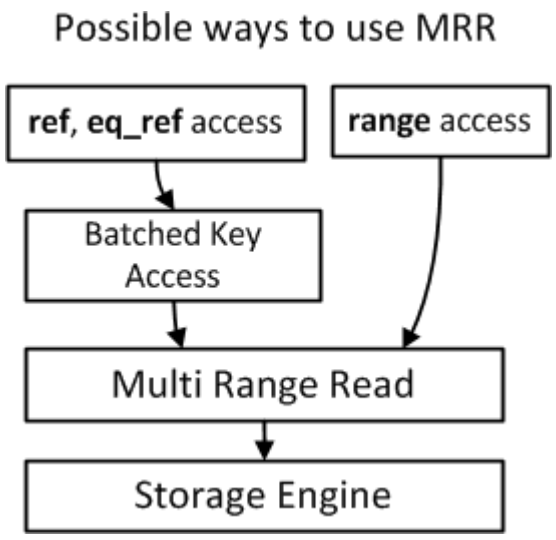
1. 杂谈篇之我是怎么读源码的，授人以渔(72)
2. 神奇的 SQL 之温柔的陷阱 → 三值逻辑 与 NULL ！(61)
3. 神奇的 SQL 之性能优化 → 让 SQL 飞起来(57)
4. 神奇的 SQL 之 HAVING → 容易被轻视的主角(53)
5. 神奇的 SQL 之 联表细节 → MySQL JOIN 的执行过程（一）(50)

呃，这么说吧，他们是一个爹的儿子）。简单点来说，MRR 是优化器将随机 IO 转化为顺序 IO 以降低查询过程中 IO 开销的一种手段

> 什么是读盘与落盘（IO）
>
> 当前绝大多少情况下，MySQL 的数据是存在机械硬盘（SATA 盘）上的，极少数情况下是存在固态硬盘（SSD）
>
> 为什么顺序读盘比随机读盘快
>
> 这不是绝对的，多数情况下是这样的；至于为什么，这涉及到机械硬盘的硬件知识了，包括其组织结构，以及磁盘
> MariaDB 中说明了如下 3 个原因
> 1、Rotating drives do not have to move the head back and forth
> 2、One can take advantage of IO-prefetching done at various levels
> 3、Each disk page will be read exactly once, which means we won't rely on disk cache (o

## 使用场景

不是任何情况下 MySQL 都会使用 MRR 的，只是在某些情况下会用 MRR 来进行优化




摘自 Multi Range Read Optimization

MySQL 中的 NDB 也会用到 MRR，一般而言，我们无需关注，我们只关注上图中的情况就行了

理论之后来点案例，完美！

## range access

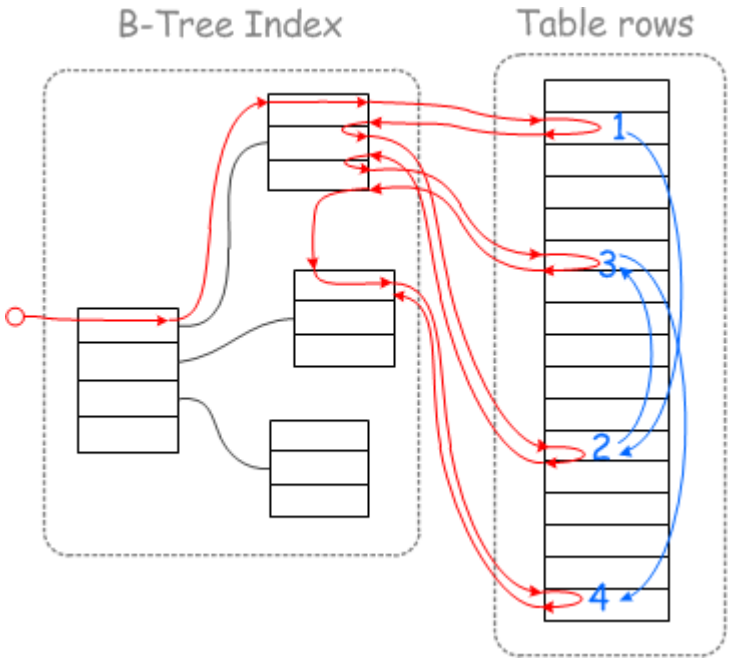
表 `tbl_range_access` 的 a 字段上我们已经建了索引 `i_a` ，我们来个范围查询，看下执行计划 `EXPLAIN SELECT * FROM tbl_range_access WHERE a BETWEEN 4 AND 9;` 如下

```
7  EXPLAIN SELECT * FROM tbl_range_access WHERE a BETWEEN 4 AND 9;
```

信息 | 结果1 | 概况 | 状态

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | tbl_range | (Null) | range | i_a | i_a | 4 | (Null) | 45 | 100 | Using index condition |

此时没有用到 MRR，执行此查询时，磁盘 IO 访问模式将遵循下图中的红线



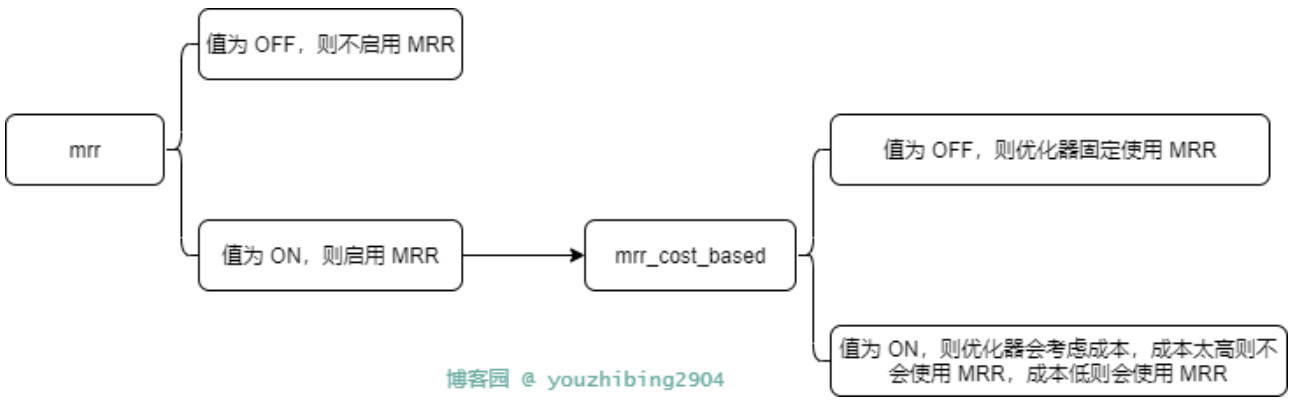

因为是 `SELECT *` ，所以通过索引 `i_a` 先找到主键 ID，然后通过主键 ID 回表（从聚簇索引）查询完整记录；a 在索引 `i_a` 中是有序的，但不保证主键在 `i_a` 中也是有序的（关于 MySQL 的索引，推荐大家去看：MySQL的索引），这就导致回表的过程是随机 IO

12

为什么 MySQL 没有采用 MRR 来保证回表的过程是顺序 IO 呢？mrr-optimization 中有这么一段话

```
Two optimizer_switch system variable flags provide an interface to the use of MRR optimization
```

`mrr` 和 `mrr_cost_based` 的默认值是 `on` ；我简单画个图，大家就明白这两个开关的作用了




上面的示例之所以没使用 MRR，是优化器觉得使用 MRR 反而提升了成本，还不如不使用

我们强制优化器使用 MRR：

```
-- 查看所有开关及其默认值
SELECT @@optimizer_switch;

-- mrr_cost_based设置成off,强制优化器使用 mrr
SET optimizer_switch='mrr_cost_based=off';
```

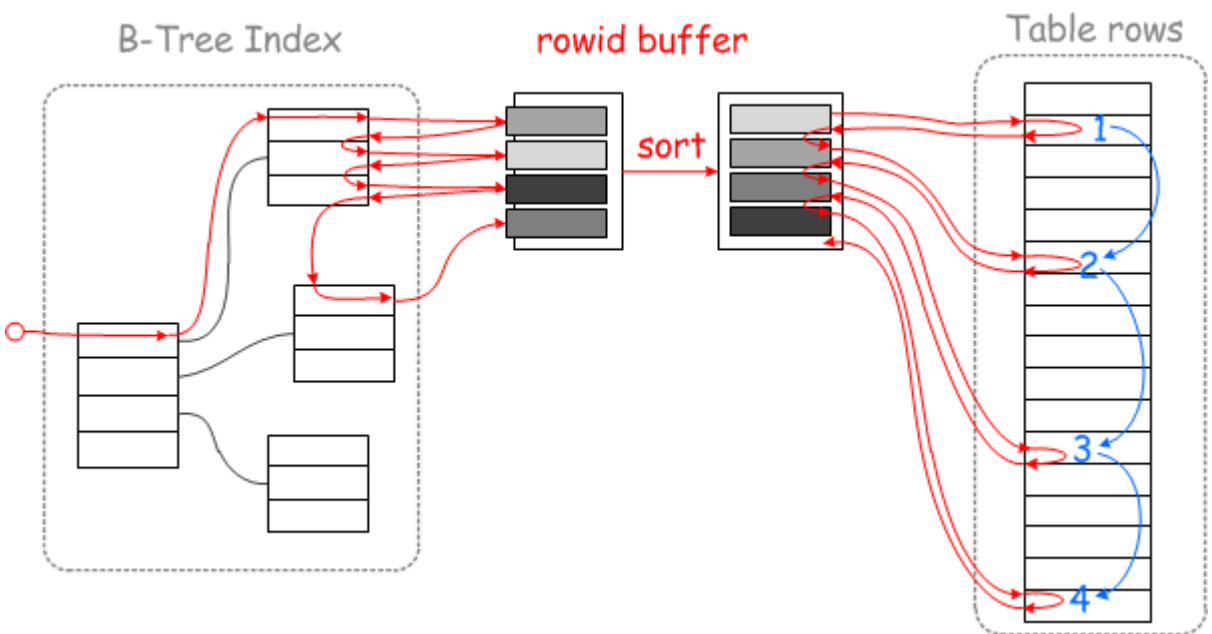
我们再来看看执行计划是什么样的

```
7  EXPLAIN SELECT * FROM tbl_range_access WHERE a BETWEEN 4 AND 9;
```

信息 | 结果1 | 概况 | 状态

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | tbl_range | (Null) | range | i_a | i_a | 4 | (Null) | 45 | 100 | Using index condition; Using MRR |

此时用到 MRR，执行此查询时，磁盘 IO 访问模式将遵循下图中的红线




此时回表查询的主键是有序的，会采用顺序 IO 来读取数据，从而提高查询效率

```
MySQL 中有个 rowids_buffer，用来缓存从索引 i_a 中查询到的数据记录（包含字段 a 和主键 ID），缓存满了或
```

是不是感觉 MRR 有点像二级索引与主键的 JOIN 操作，有这感觉就对了，后面的 BKA 也就好理解了

## BKA

BKA 全称是：`Batched Key Access` ，是对 INL 优化后的一种联表算法，类似与 BNL 对 SNL 的优化，但又有些不同，具体我们往下看

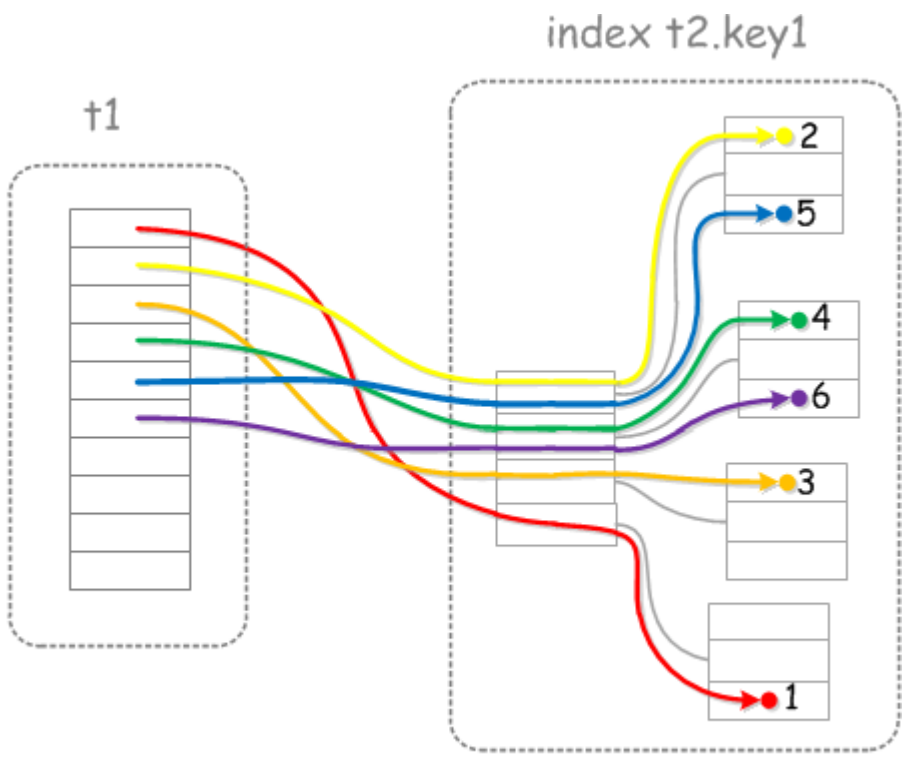
先在表 `tbl_user` 新增一个索引 `ALTER TABLE tbl_user ADD index i_aaa(user_name);` ，此时查看执行计划 `EXPLAIN SELECT * FROM tbl_user_login_log tl LEFT JOIN tbl_user tu ON tl.user_name = tu.user_name;` 如下图

```
11  EXPLAIN SELECT * FROM tbl_user_login_log tl LEFT JOIN tbl_user tu ON tl.user_name = tu.user_name;
```

信息 | 结果1 | 概况 | 状态

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | tl | (Null) | ALL | (Null) | (Null) | (Null) | (Null) | 16 | 100 | (Null) |
| 1 | SIMPLE | tu | (Null) | ref | i_aaa | i_aaa | 152 | test.tl.u | 1 | 100 | (Null) |

此时的联表算法就是 INL，因为表 `tbl_user_login_log` 的 `user_name` 是无索引的，那么从表 `tbl_user_login_log` 取出的 `user_name` 的值就是无序的，再去关联 `tbl_user` ，就会随机匹配索引 `i_aaa` ，类似下图




Regular Nested Loops Join will hit the index at random

是不是有点类似于前面讲过的回表随机 IO？

BKA 功能默认是关闭的（ `batched_key_access=off` ），开启它

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

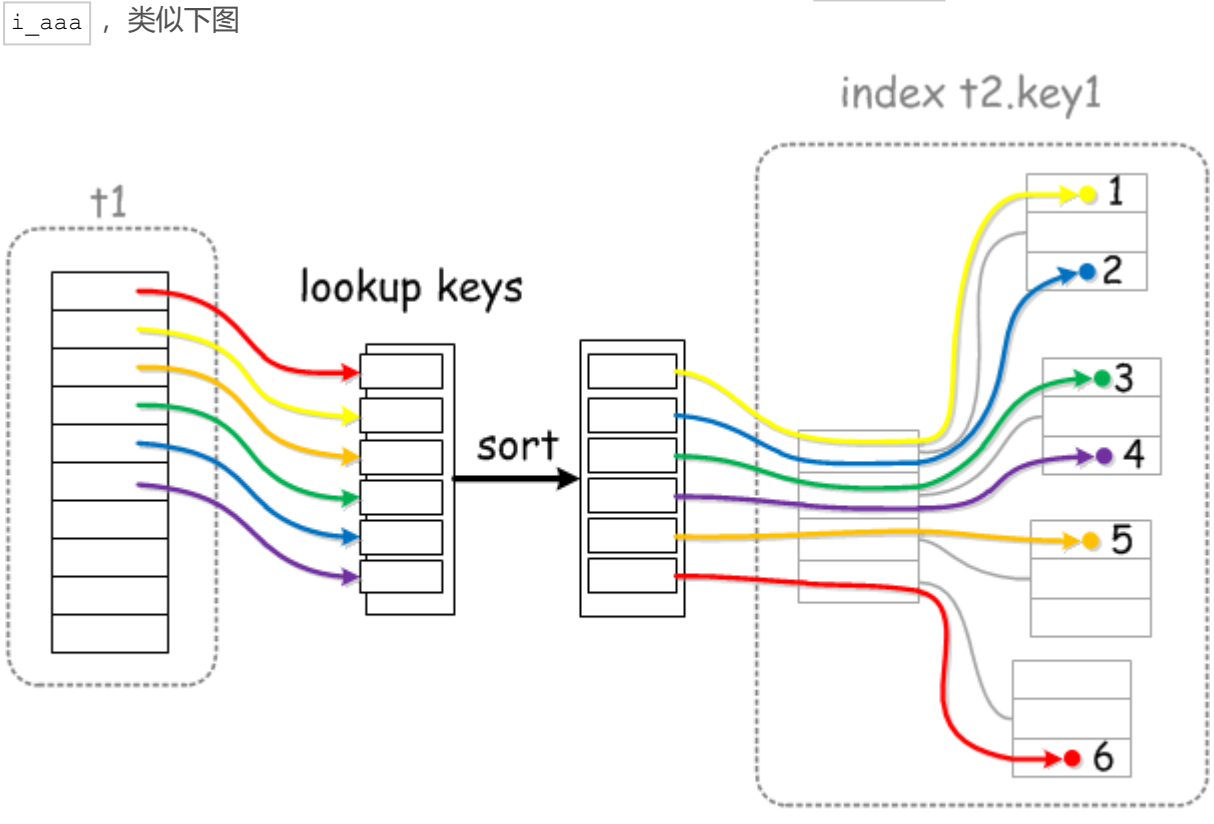
我们再来看执行计划

```
9  SET optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on';
10
11 EXPLAIN SELECT * FROM tbl_user_login_log tl LEFT JOIN tbl_user tu ON tl.user_name = tu.user_name;
```

信息 | 结果1 | 概况 | 状态

| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SIMPLE | tl | (Null) | ALL | (Null) | (Null) | (Null) | (Null) | 16 | 100 | (Null) |
| 1 | SIMPLE | tu | (Null) | ref | i_aaa | i_aaa | 152 | test.tl.user_na | 1 | 100 | Using join buffer (Batched Key Access) |

从 `tbl_user_login_log` 查询到的 `user_name` 的值先放到 join buffer，当 join buffer 满了或者数据查完了，再对 join buffer 里面的值进行排序，然后再去关联 `tbl_user` ，此时就会顺序匹配索引 `i_aaa` ，类似下图




With *Batched Nested Loops Join* and *Key-Ordered retrieval*, index lookups are done in one "sweep"

12

如果需要回表，那么 MySQL 会按之前讲到过的回表流程再优化一次

## 默认值的思考

MRR 相关的 3 个开关的默认值是这样的 `mrr=on,mrr_cost_based=on,batched_key_access=off`

`mrr=on` 表示 mrr 功能是开启的，开启并不代表一定会使用，但不开启则一定享受不到 mrr 带来的优化；

`mrr_cost_based=on` 表示优化器会基于成本考虑来决定是否使用 mrr，使用 mrr 反而使成本变高，那为什么使用 mrr ？只有 mrr 确实是带来了效率上的提升，那么使用它才有意义，但是成本的计算又是优化器来完成的，而且是一个比较复杂的过程，一定能保证优化器的成本计算是准确的吗？100%准确肯定不敢保证，但经过这么多年的沉淀，绝大多数情况下，优化器的成本计算是准确的，所以 `mrr_cost_based` 建议就采用默认值 `on` ，由优化器来决定是否采用 mrr

`batched_key_access=off` 表示默认不启用 BKA，说实话，我没太理解这么做的意图；既然是否使用 mrr 交由优化器来决定了，为什么不把是否使用 BKA 也交由优化器来决定？我能猜到的可能原因之一是 基本用不到 ，为什么这么说？ 我们回想下 BKA 会在什么情况下使用： 驱动表在关联的字段上无索引，而被驱动表在关联的字段上有索引 ，而如果驱动表在关联的字段上有索引了，还有必要进行缓存、排序、再关联被驱动表吗 ？ 很显然不必了，因为索引的字段本来就是有序的了；而实际应用中，关联的字段，不管是驱动表还是被驱动表，往往是同时存在索引的，而不是一个存在索引而另一个不存在索引。这只是我个人的猜想，望知道的大神能解惑下，小弟不胜感激！

## 总结

1、mrr 带来的性能上的提升就是将随机 IO 优化成 顺序 IO，从而提高查询效率

2、mrr 的使用场景比较有限， `range access` 和基于 `req、eq_ref access` 的 BKA，至于其他不适用的场景，我们可以结合 mrr 的特性分析出原因

3、mrr 相关的 3 个开关的默认值不建议改动，这可是 MySQL 这么多年的经验总结

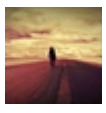
有人可能会这样说了，既然这 3 个开关不推荐改，那看与不看这篇博文没什么区别，额...，你好像说的对

4、关于 ON 和 WHERE，我只能说真的抱歉了，又要往后拖了，实在是不行，你，你.....，你来打我呀


分类: SQL，MySQL

标签: JOIN，MRR，BKA





青石路
粉丝 - 912 关注 - 7
会员号：467
+加关注

« 上一篇： 神奇的 SQL 之 联表细节 → MySQL JOIN 的执行过程（一）
» 下一篇： 记一次线上问题 → 事务去哪了

posted @ 2019-12-30 09:14 青石路 阅读(4460) 评论(18) 编辑 收藏 举报