分布式任务调度内的 MySQL 分页查询优化

> 作者：vivo 互联网数据库团队- Qiu Xinbo

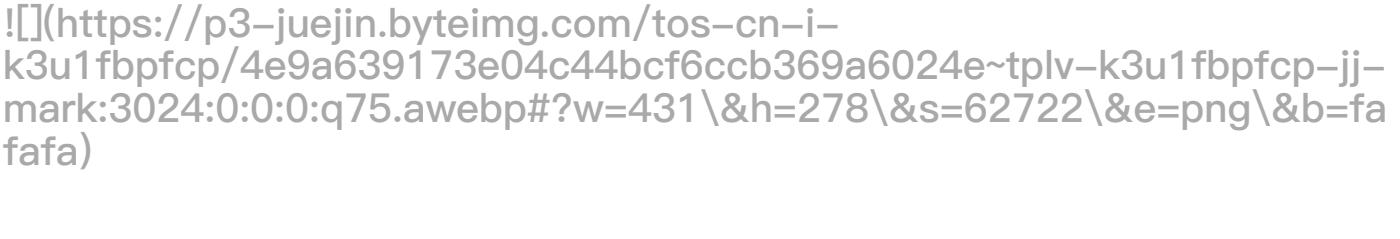
本文主要通过图示介绍了用主键进行分片查询的过程，介绍了主键分页查询存在SQL性能问题，如何去创建高效的索引去优化主键分页查询的SQL性能问题。

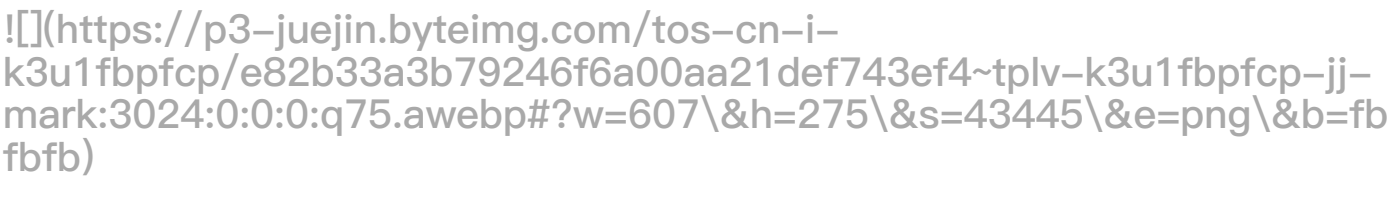
对于数据分布不均如何发现，提供了一些SQL查询案例来进行参考，对MySQL Index Condition Pushdown优化算法做了一些简单介绍。

一、背景介绍

最近在线上环境发现了一条执行较慢的分页查询，高并发执行，产生了大量的慢查询日志，CPU使用率逐步升高。

通过观察它的执行时间，发现该SQL查询时快时慢，执行时间并不稳定，以至于在高并发执行场景时，数据库来不及响应，数据库服务变慢。





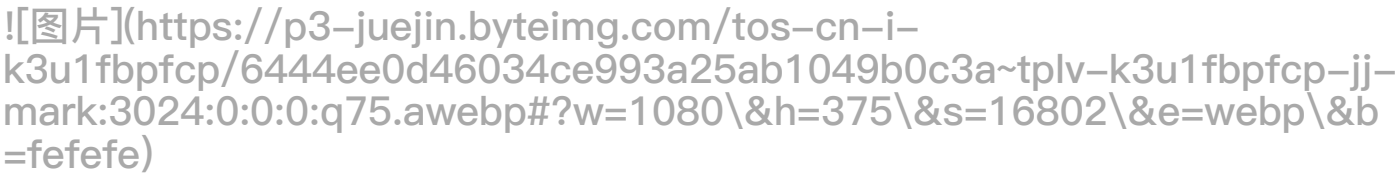
二、分析定位

2.1 定位 SQL 执行变慢的原因

通过数据库管理平台查看SQL执行信息发现，SQL解析行数（扫描行数）和SQL执行时间都很不稳定，执行时长和解析行数（扫描行数）是成正比的。

这个也能解释的通为什么SQL执行时长变了，因为扫描行数变多了，SQL执行时间成比例增长。

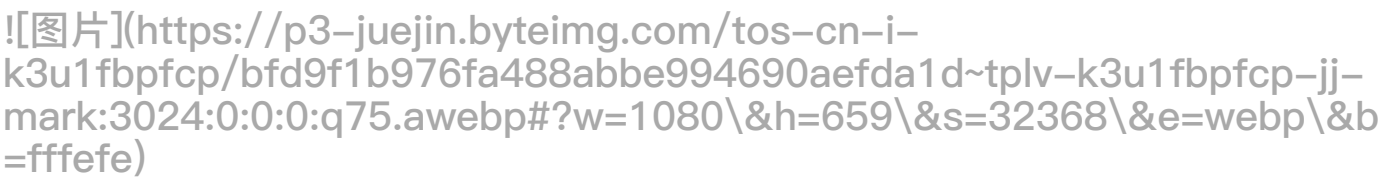
```
-- SQL全文
select
    id,
    uuid,
    name,
    user_type,
    is_deleted,
    modify_date
from
    test_user
where
    is_deleted=0
    and user_type=0
    and id > 10000
    and id % 10 = 9
order by
    id  limit 500;
```



2.2 了解 SQL 的业务背景

通过与研发沟通发现，该SQL原来是串行执行，单个线程在跑，后来觉得比较慢，改为分布式任务并行执行，通过id取模0-9，调度10个线程，每个线程处理1个分区，这样就有10个并发相当于把数据做了切片，并发查询并发处理，由此带来数据库端的并发升高。从技术角度上看，提高数据处理速度，给数据做切片，改单线程为并发处理，并没有任何问题，反而是一种比较好的优化方案

，但是高并发执行的SQL都是要有一个前提，SQL执行效率要特别高，否则会导致数据库端物理机资源耗尽，数据库服务来不及响应。



2.3 定位 SQL 扫描行数变化的原因

2.3.1 慢 SQL 及表结构信息

```
-- 为了方便理解和说明，新建一个test_user表，造了一些模拟数据，将SQL做了一些简化，不影响整体的分析效果

-- SQL全文
select
    id,
    uuid,
    name,
    user_type,
    is_deleted,
    modify_date
from
    test_user
where
    is_deleted=0
    and user_type=0
    and id > 10000
    and id % 10 = 9
order by
    id  limit 500;


-- 表信息
 CREATE TABLE `test_user` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `uuid` varchar(64) NOT NULL COMMENT '用户ID',
  `name` varchar(20) DEFAULT '' COMMENT '用户名',
  `user_type` tinyint(4) NOT NULL DEFAULT '0',
  `is_deleted` tinyint(4) NOT NULL DEFAULT '0',
```

```
  `modify_date` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP
ON UPDATE CURRENT_TIMESTAMP COMMENT '修改时间',
  `create_date` datetime DEFAULT CURRENT_TIMESTAMP COMMENT
'创建时间',
  PRIMARY KEY (`id`),
  UNIQUE KEY `uniq_uuid` (`uuid`),
  KEY `idx_modifydate` (`modify_date`)
) ENGINE=InnoDB AUTO_INCREMENT=7986024 DEFAULT
CHARSET=utf8mb4
```

2.3.2 查看 SQL 执行计划

通过查看SQL执行计划，发现执行计划走主键索引扫描，以下是SQL执行计划的关键信息解读：

* type=range　　范围扫描
* key = primary 使用主键索引
* rows = 877w　预估的扫描行数
* filter = 1.00　　百分比，满足过滤条件返回的行数　= rows * filter

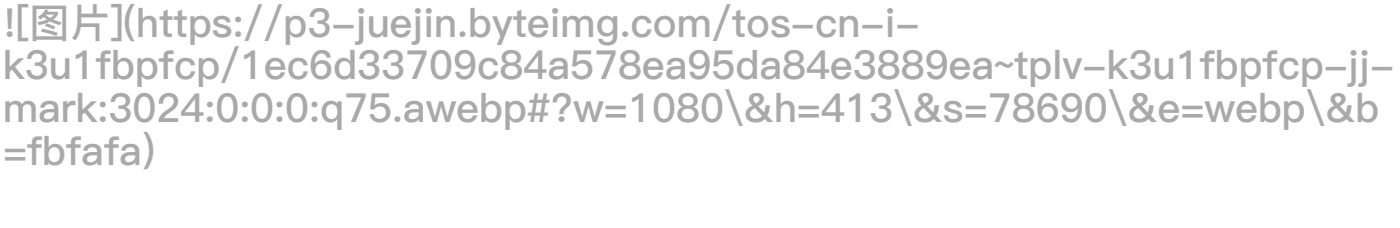
```
mysql> explain  select
    ->     id,
    ->     uuid,
    ->     name,
    ->     user_type,
    ->     is_deleted,
    ->     modify_date
    -> from
    ->     test_user
    -> where
    ->     is_deleted=0
    ->     and user_type=9
    ->     and id > 10000
    ->     and id % 10 = 9
    -> order by
    ->     id  limit 500;
+----+-------------+-----------+------------+-------+----
-----------+---------+---------+------+---------+--------
--+-------------+
| id | select_type | table     | partitions | type  | possible_keys | key     |
key_len | ref  | rows    | filtered | Extra       |
```

```
+----+-------------+-----------+------------+-------+---
----------+---------+---------+------+---------+-------
--+-------------+
|  1 | SIMPLE      | test_user | NULL       | range | PRIMARY       | PRIMARY
| 8       | NULL | 8775507 |     1.00 | Using where |
+----+-------------+-----------+------------+-------+---
----------+---------+---------+------+---------+-------
--+-------------+
1 row in set, 1 warning (0.00 sec)

```

2.3.3 图示 SQL 执行过程

通过简单的图示，描述下SQL扫描过程，由于是通过主键索引遍历，避免了额外的排序行为，从最小id开始取到最大id。

```
mysql> select min(id),max(id) from test_user;
+---------+----------+
| min(id) | max(id)  |
+---------+----------+
|       3 | 17889149 |
+---------+----------+
1 row in set (0.00 sec)

```



2.3.4 计算数据分布

从SQL过滤条件看只有is_deleted、user_type、id这三个，能预估到is_deleted和user_type区分度不高，通过SQL查看下数据的分布。

```
mysql> select is_deleted,user_type,count(*) from test_user group by
is_deleted,user_type order by count(*) desc limit 1,10;
```

```
+------------+-----------+----------+
| is_deleted | user_type | count(*) |
+------------+-----------+----------+
|          1 |         1 |  4473019 |
|          1 |         0 |  4471648 |
|          0 |         0 |  4470140 |
|          0 |         2 |      999 |
+------------+-----------+----------+
4 rows in set (4.81 sec)
-- 从数据分布来看user_type等于2的数据较少，只有999条，其他相对比较均匀
```

数据分布验证测试

将上述4种结果（is_deleted和user_type）分别通过SQL查看最近1000条满足条件的数据的id区间，验证数据的分布。

* is_deleted=1、user_type=1
* is_deleted=1、user_type=0
* is_deleted=0、user_type=0

```
-- 最近1000条is_deleted=1、user_type=1的数据记录分布在id 6-3876,大约扫描3871条数据,能返回500条满足条件的值,数据分布均匀.
mysql> select max(id),min(id) from( select id from test_user where is_deleted=1 and user_type=1 order by id  limit 1000) a;
+---------+---------+
| max(id) | min(id) |
+---------+---------+
|    3876 |       6 |
+---------+---------+
1 row in set (0.00 sec)

-- 最近1000条is_deleted=1、user_type=0的数据记录分布在id 3-4019,大约扫描4016条数据,能返回500条满足条件的值,数据分布均匀.
mysql> select max(id),min(id) from( select id from test_user where is_deleted=1 and user_type=0 order by id  limit 1000) a;
+---------+---------+
| max(id) | min(id) |
+---------+---------+
|    4019 |       3 |
+---------+---------+
```

1 row in set (0.00 sec)

-- 最近1000条is_deleted=0、user_type=0的数据记录分布在id 5-4020,大约扫描4015条数据,能返回500条满足条件的值,数据分布均匀.
mysql> select max(id),min(id) from(select id from test_user where is_deleted=0 and user_type=0 order by id limit 1000) a;
+---------+---------+
| max(id) | min(id) |
+---------+---------+
| 4025 | 5 |
+---------+---------+
1 row in set (0.00 sec)

```



**is\_deleted=0、user\_type=2**

```
-- 最近1000条is_deleted=0、user_type=2的数据记录分布在id 17890648-17891147，是比较紧凑的，但是由于id比较大，整体排在较后的位置。
-- 如果按照主键遍历，需要遍历完前面的1700w条不符合条件数据，才能遍历到满足条件的数据。
mysql> select max(id),min(id) from(select id from test_user where is_deleted=0 and user_type=2 order by id limit 1000) a;
+----------+----------+
| max(id) | min(id) |
+----------+----------+
| 17891147 | 17890149 |
+----------+----------+
1 row in set (0.00 sec)

```



### 2.3.5 实际执行测试
```

重要字段信息说明：

* Query_time：SQL执行时间
* Rows_examined：SQL扫描行数
* Rows_sent：SQL返回行数

```
# Query_time: 0.012232  Lock_time: 0.000076 Rows_sent: 500
Rows_examined: 19507

SET timestamp=1695711685;

select id,uuid,name,user_type,is_deleted,modify_date from test_user
where is_deleted=1 and user_type=1 and id > 0 and id % 10 = 9 order by
id  limit 500;
```

```
# Query_time: 0.009549  Lock_time: 0.000074 Rows_sent: 500
Rows_examined: 20537

SET timestamp=1695711745;

select id,uuid,name,user_type,is_deleted,modify_date from test_user
where is_deleted=1 and user_type=0 and id > 0 and id % 10 = 9 order by
id limit 500;
```

```
# Query_time: 0.009835  Lock_time: 0.000081 Rows_sent: 500
Rows_examined: 21037

SET timestamp=1695711779;

select id,uuid,name,user_type,is_deleted,modify_date from test_user
where is_deleted=0 and user_type=0 and id > 0 and id % 10 = 9 order by
id limit 500;
```

（这边大家可能会有疑惑，为什么扫描行数要比预估的多一些，其实也正常，我们在做预估时并没有把取模的过滤条件加上，所以必然会多扫描）

```
# Query_time: 6.981938  Lock_time: 0.000076 Rows_sent: 100 Rows_examined: 17890145

SET timestamp=1695711818;

select id,uuid,name,user_type,is_deleted,modify_date from test_user where is_deleted=0 and user_type=2 and id > 0 and id % 10 = 9 order by id limit 500;
```

2.3.6 自此能得到结论

因为is_deleted和user_type数据分布不均匀并且数据区分度不高，执行计划走主键顺序扫描， 在查询is_deleted=0 and user_type=2 特定场景的时，因为走主键索引顺序遍历，满足user_type=2 的id比较靠后，需要先扫描完成前面1700w条数据后，才能找到满足user_type=2的数据，SQL扫描行数变多，SQL执行时间变长。

三、优化方案

3.1 优化方案确定

当前SQL执行计划以主键进行顺序遍历，是一个范围扫描，有点像在一片很大的居民区按照序号挨家挨户寻找一些特定的人一样，比较简单也比较低效。

既然查询是以**is_deleted**和**user_type**为主要的过滤条件，查询特定的人群信息，可以考虑直接在这两列上添加索引，记录特定人群信息的位置，根据位置直接去定向寻找。

虽然**is_deleted**和**user_type**字段区分度很低，但是成为有序结构，能避免这条SQL大量的读取不符合条件的数据的行为，添加索引的收益远大于索引带来负面影响。

最终的添加的索引：

```
alter table test_user add index
idx_isdeleted_usertype_id(is_deleted,user_type,id);
```

添加该索引的考虑：遵循ESR原则（等值在前，排序在中间，范围在最后），既能高效扫描到对应的数据,还能避免id的排序，extra内显示使用了Using index condition。

```
mysql>  explain select id,uuid,name,user_type,is_deleted,modify_date
from test_user where is_deleted=0 and user_type=2 and id > 0 and id %
10 = 9 order by id limit 500;
+----+-------------+-----------+------------+-------+----
-------------------------------+-------------------------
--+---------+------+------+----------+--------------------
----+
| id | select_type | table     | partitions | type  | possible_keys
| key                           | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-----------+------------+-------+----
-------------------------------+-------------------------
--+---------+------+------+----------+--------------------
----+
|  1 | SIMPLE      | test_user | NULL       | range |
PRIMARY,idx_isdeleted_usertype_id | idx_isdeleted_usertype_id | 10      |
NULL |  999 |   100.00 | Using index condition |
+----+-------------+-----------+------------+-------+----
-------------------------------+-------------------------
--+---------+------+------+----------+--------------------
----+
1 row in set, 1 warning (0.00 sec)
```

3.2 优化效果对比

优化前：

```
# Query_time: 6.981938  Lock_time: 0.000076 Rows_sent: 100
Rows_examined: 17890145
SET timestamp=1695711818;
select id,uuid,name,user_type,is_deleted,modify_date from test_user
where is_deleted=0 and user_type=2 and id > 0 and id % 10 = 9 order by
id limit 500;
```

优化后：

```
# Query_time: 0.000884  Lock_time: 0.000091 Rows_sent: 100
Rows_examined: 100
SET timestamp=1695714485;
select id,uuid,name,user_type,is_deleted,modify_date from test_user
where is_deleted=0 and user_type=2 and id > 0 and id % 10 = 9 order by
id limit 500;
```

* 优化提升：

扫描行数从1700w条**降低为100条**，查询时间从6.98s **降低为
 0.8ms**。

3.3 图示的优化后的SQL执行过程

1. 通过idx_isdeleted_usertype_id索引的有序性，进行二分查找，快速定位到满足is_deleted和user_type、id条件主键信息。
2. 通过主键信息回表读取完整的数据。
3. 返回数据给客户端服务。

3.4 ICP特性(Index Condition Pushdown)

补充下执行计划内extra列体现Using index condition优化。

* 索引条件下推 (ICP) 是针对 MySQL 使用索引从表中检索行的情况的优化。
* 如果没有 ICP，存储引擎会遍历索引以定位基表中的行，并将它们返回给 MySQL server，由 MySQL server评估行的 WHERE 条件。
* 在启用 ICP 的情况下，如果 WHERE 条件的一部分可以通过仅使用索引中的列来评估，MySQL server会将这部分 WHERE 条件下推到存储引擎。
* 然后存储引擎通过使用索引条目来评估推送的索引条件，并且只有在满足这一条件时才从表中读取行。
* ICP可以减少存储引擎必须访问基表的次数和MySQL server必须访问存储引擎的次数。

ICP优化的使用和局限性路:

ICP优化在数据库优化器内默认是开启的，ICP优化适用性取决于以下条件:

* icp 对于使用rang、ref、eq_ref 和ref_or_null访问模式去检索全表数据行时候。
* icp 只适用于innodb、myisam引擎的表，包括分区的InnoDB和MyISAM表。
* icp只会使用二级索引，减少完整行记录的读取和减少I/O操作 对于聚集索引，完整行记录已经被读入innodb buffer中，using icp不能减少I/O操作。
* icp不支持使用创建在虚拟列上的二级索引，innodb引擎支持在虚拟列上创建二级索引。
* 引用子查询的条件无法下推。
* 引用存储函数的条件无法下推。存储引擎无法调用存储的函数。
* Triggered conditions cannot be pushed down。

```
-- 测试下相同的SQL执行在开启ICP优化和关闭ICP优化,执行时间和扫描行数的对比.
```

```
-- 关闭ICP,SQL执行扫描行数是5043行,执行时间为8.03ms.
SET optimizer_switch='index_condition_pushdown=off';
# Query_time: 0.008031  Lock_time: 0.000085 Rows_sent: 500
Rows_examined: 5043
select id,uuid,name,user_type,is_deleted,modify_date from test_user
where is_deleted=0 and user_type=0  and id > 10000 and id % 10 = 9
order by id limit 500;

-- 开启ICP,SQL执行扫描行数仅为500行,执行时间为2.72ms.
SET optimizer_switch='index_condition_pushdown=on';
# Query_time: 0.002724  Lock_time: 0.000082 Rows_sent: 500
Rows_examined: 500
select id,uuid, name, user_type,is_deleted,modify_date from test_user
where is_deleted=0 and user_type=0 and id > 10000 and id % 10 = 9
order by id limit 500;
```

结论：本次测试，开启ICP优化，SQL执行时扫描的行数仅为未开启时的1/10，执行时间提升约2-3倍。

四、总结

1. 将SQL查询从串行改为高并发执行，需要评估下SQL查询效率是否足够高，评估的标准：SQL扫描行数/SQL返回行数 结果越大说明存在很多低效的数据扫描，执行效率不高。
2. 分页查询通过主键遍历是顺序遍历，从最小id到最大id，当存在其它过滤条件时，需要再次判断数据是否满足这些过滤条件，扫描的行数会随着增长。
3. 区分度较低的字段并非不适合创建索引，仔细评估查询的场景，建立特定的组合索引，触发MySQL icp优化，对查询性能会有很大提升。

参考文章

Index Condition Pushdown介绍：

* [Index Condition Pushdown](http://cxyroad.com/"https://mariadb.com/kb/en/index-condition-pushdown/")
* [Index Condition Pushdown Optimization](http://cxyroad.com/"https://dev.mysql.com/doc/refman/5.7/en/index-condition-pushdown-optimization.html")

原文链接: https://juejin.cn/post/7372100124637134858