1 RangeScanSolution - 如何最高效的扫描表数据

by heylichen@qq.com

有时我们需要全表扫描。为便于讨论，以同步数据库数据到搜索引擎（以 es 为例不限于具体某个引擎）为应用背景，这又分 2 种：

- 不做任何过滤的全表扫描，例如首次将全部数据同步到 es。或者索引 shcema 字段调整，需要全量同步数据。
- 有一定过滤条件的，例如按修改时间，同步昨天 0 点到今天 0 点之间的数据到 es，用于补偿这段时间可能发生的错漏。这一般使用二级索引来保证效率。

本笔记的示例表如下

```
CREATE TABLE `shop_item`
(
    `id`          bigint unsigned  NOT NULL AUTO_INCREMENT,
    `item_name`   varchar(30)      NOT NULL DEFAULT '',
    `shop_id`     bigint unsigned  NOT NULL DEFAULT '0',
    `is_del`      tinyint unsigned NOT NULL DEFAULT '0',
    `create_time` datetime         NOT NULL DEFAULT CURRENT_TIMESTAMP,
    `update_time` datetime         NOT NULL DEFAULT CURRENT_TIMESTAMP
                                            ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    KEY `idx_update_time` (`update_time`),
    KEY `idx_shop_id_del` (`shop_id`, `is_del`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
COMMENT ='table of items selling at all shops.'
```

数据按 shop 分布如下

```
+-------+--------+
|shop_id|count(*)|
+-------+--------+
|1001   |60000   |
|10002  |60010   |
|10003  |60010   |
|10004  |60010   |
|10005  |60010   |
|10006  |60010   |
|10007  |60010   |
|10008  |60010   |
|10009  |60010   |
|10010  |60010   |
+-------+--------+
```

为简化场景，我们先限制 select 字段只有主键 id, 有了主键可以再发起一次 sql 查询整行数据。

1.1 扫描全表

idea: 直接按主键扫描即可。

1.1.1 单线程扫描

一个表的数据量可以很大，因此不能用分页 limit 语句去实现，可能会有深度分页的性能问题。

一个避免深度分页的重要思路就是，使用类似 cursor 的概念来完成分批遍历, 正如 High Performance MySql 4th Edition, 第 8 章，224 页所说:

The problem with LIMIT and OFFSET is really the OFFSET, which represents rows the server is generating and throwing away. If you use a sort of cursor to remember the position of the last row you fetched, you can generate the next set of rows by starting from that position instead of using an OFFSET.

```
-- SQL id_scan_start
select id
from shop_item
order by id asc
limit 500;
```

```
-- explain of SQL id_scan_start
id             1
select_type    SIMPLE
table          shop_item
partitions     null
type           index
possible_keys  null
key            PRIMARY
key_len        8
ref            null
rows           500
filtered       100
Extra          Using index
```

```
-- SQL id_scan_keep
select id
from shop_item
where id > 500
order by id asc
limit 500;
```

```
-- explain of SQL id_scan_keep
id             1
select_type    SIMPLE
table          shop_item
partitions     null
type           range
possible_keys  PRIMARY,idx_update_time
               ,idx_shop_id_del
key            PRIMARY
key_len        8
ref            null
rows           329074
filtered       100
Extra          Using where; Using index
```

第一次使用 `SQL id_scan_start`，按 id 升序扫描主键索引。结果集里最后一行的 id, 作为 cursor，后续都是用`SQL id_scan_keep` 来查询，把 cursor 值作为 SQL 查询参数传入。这样可以保证直到扫描完整个表，每次查询的效率都是很高的。

上面 sql 都使用了 primary index, 并且是 covering index，因为 select 只有一列为 id。不论表的数据量多大，这样把整个表扫完，不会出现慢查询。

全表扫描的算法伪代码为

```
public class ScanParam {
  private BigInteger previousId;
  private final int pageSize;

  public ScanParam(int pageSize) {
    this.pageSize = pageSize;
  }
  //getters and setters
}
```

```
// the scan full table algorithm
int pageSize = 500;
ScanParam param = new ScanParam(pageSize);
while (true) {
  // ShopItem is entity class of shop_item
  // iterate(param) query database,
  //         use SQL id_scan_start if previousId is null,
  //         use SQL id_scan_keep if previousId is not null
  List<ShopItem> details = iterate(param);
  if(details.isEmpty()){
    break;
  }

  businessProcess(details);
  BigInteger previousId = details.get(details.size()-1).getId();
  so.setPreviousId(previousId);
  // not a full page, no need to issue another sql query
  if (details.size()<pageSize){
    break;
  }
}
```

1.1.2 多线程扫描

对于整个同步数据过程，一般处理的速度瓶颈不会在扫表阶段。但如果我们想进一步加速扫描，可以考虑多线程扫描。idea，生产者消费者模式，单线程生产者，多线程消费者，之间通过 BlockingQueue 传递信息。

- 生产者：按 pkId 升序将 pkId 分为若干区间，一个区间有例如 10 万个 pkId。将 pkId 区间发送到 BlockingQueue(blocking put).
- 消费者：多个消费者从 BlockingQueue 取 id 区间，就扫描这个 pkId 区间的数据并处理。

生产逻辑

查 id 区间 sql(为表述方便命名为 splitIdRange)

```
-- SQL id_range_start
select min(id) as min_id,
       max(id) as max_id
from (select id
      from shop_item
      order by id asc
      limit 100000) a;

-- say max(id) = 100050
```

```
-- explain of id_range_start
id             1          2
select_type    PRIMARY    DERIVED
table          <derived2> shop_item
partitions     null       null
type           ALL        index
possible_keys  null       null
key            null       PRIMARY
key_len        null       8
ref            null       null
rows           100000     100000
filtered       100        100
Extra          null       Using index
```

使用了 primary index, 并且是 covering index(extra 中出现 using index). 性能符合预期。

```
-- SQL id_range_keep_scan
-- 100050 is max(id) from
--  SQL id_range_start result
select min(id) as min_id,
       max(id) as max_id
from (select id
      from shop_item
      where id > 100050
```

```
      order by id asc
      limit 100000,1) a;
```

```
-- explain of id_range_keep_scan
id             1          2
select_type    PRIMARY    DERIVED
table          <derived2> shop_item
partitions     null       null
type           ALL        range
possible_keys  null       PRIMARY,idx_update_time,
                          idx_shop_id_del
key            null       idx_update_time
key_len        null       13
ref            null       null
rows           100001     218760
filtered       100        100
Extra          null       Using where;
                          Using index for skip scan;
                          Using filesort
```

使用了idx_update_time，不是想用的索引。extra 中出现"Using index for skip scan"，表明使用了MySQL 8 Skip Scan Range Access Method. 但这种优化后的执行计划效率不如直接使用主键索引。

测试了一下，此执行计划耗时 261ms。加上 index hint 使用主键索引后，sql 执行耗时 56ms。执行计划如下，使用了 primary index, covering index, 符合预期。

```
-- SQL id_range_keep_scan_use_pk
-- 100050 is max(id) from
--  SQL id_range_start result
select min(id) as min_id,
       max(id) as max_id
from (select id
      from shop_item
           use index (PRIMARY)
      where id > 100050
      order by id asc
      limit 100000) a;
```

```
-- explain of id_range_keep_scan_use_pk
id             1          2
select_type    PRIMARY    DERIVED
table          <derived2> shop_item
partitions     null       null
type           ALL        range
possible_keys  null       PRIMARY
key            null       PRIMARY
key_len        null       8
ref            null       null
rows           100000     328173
filtered       100        100
Extra          null       Using where; Using index
```

pk 区间大小一般 10 万即可，太大可能会有慢查询。生产全部 pkId 区间伪代码

```
//IdRange 类， 只有2个属性,min, max

List<IdRange> idRanges= new ArrayList<>();
while(true) {
    //splitIdRange query database
    //  use SQL id_range_start if param.previousId is null
    //  use SQL id_range_keep_scan_use_pk if param.previousId is not null
    //param has a single field: previousId
  IdRange idRange=splitIdRange(param);
    if(idRange==null){
      break;
    }
    idRanges.add(idRange);
    param.setPreviousId=idRange.max;
}

//process
```

```
for(IdRange idRange: idRanges){
  putToQueue(idRange); //blocking put, 避免生产与消费速度差异太大
}
putToQueue(POSON_PILL);//告知停止流程
```

消费逻辑

伪代码

```
    // multiple consumer thread, only one consumer will receiver the
    // POSON_PILL, and mark the globalStopFlag. Then all other consumers
    // will exit loop if detect globalStopFlag true.
    // this globalStopFlag variable is declared outside any consumer instance,
    // as a shared state.
    volatile boolean globalStopFlag = false;

    // inside cosumer
    while(true) {
        IdRange idRangeParam=blockQueue.poll(2,TimeUnit.SECONDS);
        if(POSON_PILL.equals(idRangeParam)){
            //got poison pill, mark and exit loop
            globalStopFlag=true;
            break;
        }
        if(globalStopFlag){
            // other consumers marked, exit loop
            break;
        }
        if(idRangeParam==null){
            // may have more msgs
            continue;
        }
        //根据id range查ids, 处理逻辑
        process(idRangeParam);
    }
```

其中 process 逻辑类似于单线程扫描逻辑，遍历这个 id 范围的所有 ids，再进行业务处理逻辑。这里贴一下 sql(mybatis 语法)，不再展开了。

```
select id from shop_item
where id>=#{idRange.min} AND id<#{idRange.max}
<if previousId!=null>
    AND id > #{previousId}
</if>
order by id asc
limit 1000;
```

生产消费 id 逻辑优化

上面提出生产者查询出 idRange, 不断的提供给消费者。消费者收到 idRange 消息，再不断扫描这一段 idRange 内的数据行。这里生产者查询出 idRange 的 SQL, `SQL id_range_start` 和 `SQL id_range_keep_scan` 因为"limit 100000,1"，效率不是特别好。查出 idRange 后，消费者还要根据 idRange 再次查询才能得到具体的 id 列表，多了一次查询。

生产者端可以不生产 idRange，而是直接用"单线程扫描" 小节中的 SQL 直接查询出 id 列表，将 id 列表作为消息发送给多个消费者。消费者收到消息，直接根据 id 列表处理业务逻辑。这样就没有低效率的 SQL 和无谓的查询了。

为何之前首先想到的是生产出 IdRange 作为消息传递和消费的单位？可能是想消费者一次收到的消息代表的任务量要足够大，线程之间交互频率会低一些。例如 10 万个 id，足够消费者处理一段时间了。但仔细想想，必要性有待商榷。

1.2 指定范围的表数据扫描

例如按修改时间update_time，同步昨天 0 点到今天 0 点之间的数据到 es，用于补偿这段时间可能发生的错漏。或者按一批shop_id, 同步某些店铺的全部数据。

以按修改时间 (update_time) 范围扫描为例。此方案假定有update_time 的二级索引，且二级索引只有 1 个字段: update_time，不是组合索引。对于有多个字段的组合索引，也可以支持，但为简化表述，这里先不考虑。

我们有入参

```
{
  updateTimeRange":{
    "min":"#{min}",
    "max":"#{max}"
  }
}
```

下面先说一些工作中实际遇到的，对此场景的错误解决案例。

1.2.1 谬误 1：未走预期的索引

猜测一下开发者的设计思路。因为过去一段时间变化的 item 数量不可控，可能很多，因此决定按shop_id 来处理。每个shop_id 扫描变化的 items, 这里又担心一个shop_id 对应的过去 24 小时变化的数量太多，又把 24 小时分隔为 24 个时间段，遍历处理。总体流程如下

```
//按时间段和shop_id扫描方案
{
 1 调接口，查出系统中所有shop_ids.
 2 过去24小时的时间段，按小时分割为24个时间段 update_time_hour_ranges
 3 for each shop_id in shop_ids
       for update_time_range_1_hour in update_time_hour_ranges
            scanAndProcess(shop_id, update_time_range_1_hour)

}
```

scanAndProcess 就是扫描 1 个小时范围内，一个shop_id 的所有 items，并做业务处理。下面看一下对一个shop_id, 一个小时时间段扫描表数据的 SQL.

```
-- SQL shop_id_update_time_start
select min(id)
from shop_item
where shop_id = 10011
  AND update_time
    between '2023-02-14 23:00:00'
    AND '2023-02-15 00:00:00';

-- min(id)=60136
```

```
 -- explain of shop_id_update_time_start
id            1
select_type   SIMPLE
table         shop_item
partitions    null
type          ref
possible_keys idx_update_time,idx_shop_id_del
key           idx_shop_id_del
key_len       8
ref           const
rows          109244
filtered      50
Extra         Using where
```

上面 SQL 走了idx_shop_id_del 索引，where 条件中的update_time 条件需要回表才能过滤。性能不稳定，如果一个shop_id 的数据量不大还好，很大的话必将产生慢查询。

开发者做过全表扫描同步数据的开发，可能是沿用"单线程扫描" 的思路，再通过 id 按升序，遍历完剩下的数据。

```
-- SQL shop_id_update_time_keep
-- 60136是
-- SQL shop_id_update_time_start
-- 查出的min(id)
select id
from shop_item
where shop_id = 10011
  AND update_time
    between '2023-02-14 23:00:00'
    AND '2023-02-15 00:00:00'
  AND id > 60136
order by id asc
limit 0, 500;
```

```
 -- explain of shop_id_update_time_keep
id            1
select_type   SIMPLE
table         shop_item
partitions    null
type          range
possible_keys PRIMARY,idx_update_time,
              idx_shop_id_del
key           PRIMARY
key_len       8
ref           null
rows          328173
filtered      8.32
Extra         Using where
```

开发者可能是想用 idx_shop_id_del 索引，但结果是走了 primry index。取决于数据的分布情况，可能要扫描很多行才能找到足够数量的结果集，效率很低。在测试表中实测，执行耗时 225ms，速度比较慢.

where 过滤条件中，shop_id 和 update_time 分别属于不同的二级索引，所以是无法完全利用索引的。而且这 2 个字段作为过滤条件，排序却是 id，是无法通过扫描索引来实现排序的。

可以使用 index hint 使得 optimizer 使用idx_shop_id_del 索引。实测耗时 130ms, 速度提升了不少，但也不是很快。从执行计划可以看到，过滤使用了 index condition pushdown, 可以使用 idx_shop_id_del 索引过滤shop_id 和 id 的过滤条件，但update_time 过滤需要回表后才能过滤，所以 filtered 不高。另外 id 排序是无法通过扫描该索引来实现，因此 extra 出现了"Using filesort".

再者，使用idx_shop_id_del 索引通过 ICP 过滤，也是有效率风险的。如果一个shop_id 对应的数据量极端大，例如百万，千万级数据量，那么扫描回表就需要涉及到大量的数据行，此时执行耗时就不可控了。

SQL 和执行计划如下。

```
-- SQL shop_id_update_time_hint_keep
select id
from shop_item
     use index(idx_shop_id_del)
where shop_id = 10011
  AND update_time
    between '2023-02-14 23:00:00'
    AND '2023-02-15 00:00:00'
  AND id > 60136
order by id asc
limit 0, 500;
```

```
 -- explain of shop_id_update_time_hint_keep
id            1
select_type   SIMPLE
table         shop_item
partitions    null
type          ref
possible_keys idx_shop_id_del
key           idx_shop_id_del
key_len       8
ref           const
rows          109244
filtered      3.7
Extra         Using index condition;
              Using where; Using filesort
```

总结: 对组合索引的最左匹配原则，对于索引在过滤和排序中是如何起作用的没有深入理解，盲目沿用主键扫描的思路，导致了这样的方案和 SQL.

1.2.2 谬误 2 逻辑错误导致扫描数据不全

回头思考一下"谬误 1：未走预期的索引中的按时间段和shop_id 扫描方案", 其实合理使用索引，本不需要那么复杂的按门店和小时分割查询范围。原本的需求是扫描出一段时间内变化的 items, 能分批的扫描完这批数据即可。

使用idx_update_time 索引，既可以用来过滤，也用它来实现排序，避免 filesort, 效率是可以保证很高的。

```
-- SQL update_time_scan_incomlpete
select id
from shop_item
where update_time
          between '2023-02-14 00:00:00'
          AND '2023-02-15 00:00:00'
order by update_time asc, id asc
limit 0, 500;
```

```
 -- explain of update_time_scan_incomlpete
id            1
select_type   SIMPLE
table         shop_item
partitions    null
type          range
possible_keys idx_update_time
key           idx_update_time
key_len       5
ref           null
rows          328173
filtered      100
Extra         Using where; Using index
```

第一次按上面的 SQL 执行，然后取最后一行的 id, 作为 previousId(作为 cursor)，用下面的 SQL 循环查询直到没有结果，就完成了遍历。

```
-- SQL update_time_scan_keep_incomlpete
select id
from shop_item
where update_time
    between '2023-02-14 00:00:00'
    AND '2023-02-15 00:00:00'
  AND id > 58500
order by update_time asc, id asc
limit 0, 500;
```

```
-- explain of update_time_scan_incomlpete
id            1
select_type   SIMPLE
table         shop_item
partitions    null
type          range
possible_keys PRIMARY,idx_update_time
key           idx_update_time
key_len       13
ref           null
rows          328173
filtered      50
Extra         Using where; Using index
```

看执行计划，使用了idx_update_time 索引，where 条件过滤和排序都走了索引，而且是 coverint index, "完美 Perfect!"。

关于 order by 如何利用索引，避免 filesort，详见 High Perforamce MySql, 下面简略摘要一下:

MySQL can use the same index for both sorting and finding rows. If possible, it’s a good idea to design your indexes so that they’re useful for both tasks at once.

Ordering the results by the index works only when the index’s order is exactly the same as the ORDER BY clause and all columns are sorted in the same direction (ascending or descending). 排序字段与顺序要与用到的索引的字段和顺序一致，或者符合最左匹配。并且所有字段的排序方向要是一致的。组合索引实际就是把其包含的多个字段预先排好序，可以理解上面是很自然的要求。

One case where the ORDER BY clause doesn’t have to specify a leftmost prefix of the index is if there are constants for the leading columns. If the WHERE clause or a JOIN clause specifies constants for these columns, they can “fill the gaps” in the index. 例如，组合索引字段为 key a_b_c (a,b,c), 过滤和排序 sql 为 where a=’a’ order by b asc, c asc，或者 where a=’a’ AND b= ’b’ order by c asc.

此方案的问题

但此 sql 以及遍历逻辑的致命问题，是它从逻辑上是错的，无法保证扫描完整的指定时间段变更的数据。包括上一小节"谬误 1：未走预期的索引中的按时间段和shop_id 的方案也是有逻辑上的问题。

例如，指定时间段 2023-02-14 00:00:00　2023-02-15 00:00:00。其中在 2023-02-14 00:00:01 时间点，更新了 id=10, 2023-02-14 00:00:02, 更新 id=9. 每次 sql 分页大小为 1，即 1 次查 1 条出来。那么在第一次查出 id=10, 第二次再查

```
-- SQL update_time_scan_keep_incomlpete
select id
from shop_item
where update_time
    between '2023-02-14 00:00:00'
    AND '2023-02-15 00:00:00'
  AND id > 10
order by update_time asc, id asc
limit 0, 500;
```

查询结果为空，漏掉了 id=9。

错误的原因在于排序是按"update_time asc, id asc"，遍历时却只按 id 扫，扫描顺序和排序顺序不一致导致的。

此方案我经过思考，提前总结了它的问题所在，在第一次优化时避免了此问题。但后面还是在某次优化任务中犯了此方案的错误，比较容易迷惑人。

1.2.3　一个高效并正确的方案

case1

```
-- SQL update_time_scan_case1_start
select id,update_time
from shop_item
where update_time
    -- #{updateTime.min}
    between '2023-02-14 00:00:00'
    -- #{updateTime.max}
    AND '2023-02-15 00:00:00'
order by update_time asc, id asc
limit 0, 500;
```

```
 -- explain of update_time_scan_case1_start
id            1
select_type   SIMPLE
table         shop_item
partitions    null
type          range
possible_keys idx_update_time
key           idx_update_time
key_len       5
ref           null
rows          328173
filtered      100
Extra         Using where; Using index
```

过滤和排序走idx_update_time 索引，使用 covering index 查出 2 个字段 id,update_time.

对于某一秒，可能有上百万的更新数据量。为了对这种场景优化，我们增加一种处理逻辑。

case2

下面 SQL 中的参数 updateTime，previousId 取 case1 查出来的结果最后一行数据的列。

```
-- SQL update_time_scan_case2
select id, update_time
from shop_item
-- #{updateTime}
where update_time = '2023-02-14 09:00:00'
-- #{previousId}
  AND id > 58500
order by update_time asc, id asc
```

```
limit 0, 500;


 -- explain of update_time_scan_case2
id            1
select_type   SIMPLE
table         shop_item
partitions    null
type          range
possible_keys PRIMARY,idx_update_time
key           idx_update_time
key_len       13
ref           null
rows          40050
filtered      100
Extra         Using where; Using index
```

case2 的 SQL 执行后，继续把结果集最后一行的相关列作为 updateTime，previousId 参数，循环用 SQL update_time_scan_case2 查询，直到结果集为空。

如果查询结果为空，说明这一秒更新的数据遍历完全。此时修改#{updateTime.min} 参数，切换到 case1 的逻辑，继续按 updateTime 遍历下去，直到遍历完所有时间范围的数据。

整个算法伪代码

```
public BusinessContext scanProcess(UpdateTimeScanParam p) {
   BusinessContext bc = new BusinessContext();
   while (true) {
     //empty results will be empty collection. No NPE can occur.
     List<ShopItem> items = shopItemsMapper.iterateByUpdateTimeRange(p);
     businessProcess(items, bc);

     //not enough rows to fill a full page, no need to issue another query
     if (items.size() < p.getLimit()) {
       break;
     }
     ShopItem previous = getPrevious(items);
     iterateAtUpdateTime(previous, p.getLimit(), bc);

     // update param for case 1
     p.getUpdateTimeRange().setMin(previous.getUpdateTime().
                                   format(DATE_TIME_FORMATTER));
   }
   return bc;
 }

 private void iterateAtUpdateTime(ShopItem previous, int limit, BusinessContext bc) {
   UpdateTimeScanParam localParam = new UpdateTimeScanParam();
   localParam.setLimit(limit);
   localParam.setUpdateTime(previous.getUpdateTime().format(DATE_TIME_FORMATTER));
   //must set previous id, or previous row would be retrieved again.
   localParam.setPreviousId(previous.getId());

   while (true) {
     List<ShopItem> items = shopItemsMapper.iterateAtUpdateTime(localParam);
     businessProcess(items, bc);

     //not enough rows to fill a full page, no need to issue another query
     if (items.size() < limit) {
       break;
```

```
      }
      previous = getPrevious(items);
      localParam.setPreviousId(previous.getId());
    }
  }

  private void businessProcess(List<ShopItem> items, BusinessContext bc) {
    bc.add(items.size());
  }

  private static <T> T getPrevious(List<T> items) {
    return items.get(items.size() - 1);
  }
```

查询参数为

```
  @Getter
  @Setter
  public class UpdateTimeScanParam implements Serializable, WithLimit {
    private static final long serialVersionUID = 1L;
    //for case 1
    private StringRange updateTimeRange;
    //for case 2
    private String updateTime;
    private BigInteger previousId;
    private Integer limit;
  }
```

相关代码见"https://github.com/heylichen/ScanTableLab". 此代码中的 DirectCodedRangeScan 就是算法的实现。当理清楚遍历逻辑后，用 MultiRangeScanIterable 封装了扫描逻辑，使用时不用每次都写 2 个循环。而且对于有任意列个数的组合索引，MultiRangeScanIterable 也能支持，不局限于单列的二级索引。当然，索引列数过多，会产生一些额外的 sql 查询消耗，不推荐太多列的组合索引。

所谓额外的 sql 查询，举例说明，case1 查询完，结果集数量大于 limit，说明有更多数据。那么将会发起一次 case2 查询。如果 case2 查询时没查到数据，那么这一次查询就是浪费的。建议 limit 可以设置稍大些，降低这种浪费的频率和占比。

为了简化场景，本笔记里的 SQL 都是 select 索引中的列。实际业务处理方法，可能需要再次根据主键查询到整行的字段。可否避免这一次额外的查询？详见笔记"Scan Index and fetch result in single SQL".