rapids - 香港科技大学 - 比赛攻略

本文分为四个章节，初赛赛题相关的分析在第一章节前半部分给出。本文的主体部分主要对复赛题目进行了剖析：赛题分析(第一章后半部分)，核心思路(第二章)，关键代码(第三章)。第四章为rapids团队对第三届阿里中间件挑战赛比赛的总结以及心得体会。

1. 赛题背景分析及理解

1.1 初赛概览

1.1.1 赛题要求

在单机环境下，实现带有持久化的消息队列引擎，由第一个生产进程产生消息队列，并在最终调用`Producer`的`flush()`接口时候，保证安全地持久化到磁盘；之后由另一个消费进程进行消息队列的消费。消息会对应到一个命名空间，例如`Topic0`，`Queue0`这样的字符串。对于同一个`Producer`产生的命名空间的消息需要保证顺序性，也就是说由`(Producer_j, Name_i)`唯一标识了一个队列，这个队列中的消息在消费的时候需要保证顺序性。

题目要求实现四个`Producer`有关的接口:`createBytesMessageToTopic(topic, body)`,`createBytesMessageToQueue(queue, body)`,`send(message)`,`flush()`和两个`Consumer`有关的接口`attachQueue(queue, topics)`,`poll()`。并且，如果不同`Consumer`绑定到了同一个命名空间，这些`Consumer`看到是对应命名空间的视图，需要全部消费完命名空间中的消息，并且大家消费的内容是相同的。题目中，产生出的消息总量为40,000,000条。

1.1.2 赛题分析

题目中，产生出的消息总量为40,000,000条。如果在文件中不经过压缩，存所有信息所需的磁盘空间是4GB左右。但是测试环境的IO速度很不理想，因此选手需要充分利用linux 的 pagecache来解决IO瓶颈。

根据 https://lonesysadmin.net/2013/12/22/better-linux-disk-caching-performance-vm-dirty_ratio/ 和 https://stackoverflow.com/questions/27900221/difference-between-vm-dirty-ratio-and-vm-dirty-background-ratio ，我们可以知道，在linux中，pagecache sync相关有两个主要参数`vm.dirty_background_ratio`(默认为10%)，`vm.dirty_ratio`(默认为20%)，当dirty page的大小达到了总物理内存大小的10%时，linux操作系统会进行刷盘但不阻塞`fwrite`系统调用的写线程，但若dirty page的大小达到了物理内存大小的20%的时候，写线程就会被阻塞。

基于此，我们可以想到的优化策略是减小最终消息队列序列化到文件的大小，这个可以通过压缩算法达到，例如jdk中带有的`GZIPOutputStream`，`DeflaterOutputStream`,和其他的压缩方法实现`SnappyFramedOutputStream`,`LZ4BlockOutputStream`。其他的选手也有通过统计消息的特征进行非通用的存储设计来减小最终文件的大小，利用的是相同的思路。

1.1.3 赛题要点

然后针对本题目，文件的存储方式对于最终的成绩影响并不大，我们选择的存储方式是：每个名字一个文件夹，例如`Topic0`一个文件夹；在文件夹下面，每个文件是每个线程的名字(因为一个线程绑定一个`Producer`)，例如`Thread-0`。这样的存储设计较为简单，也不需要频繁加锁，读写锁的使用当且仅当文件夹创建的时候；并且所有文件夹创建之后(命名空间确定之后)就不会有读者-写者冲突。`(Producer_j, Name_i)`唯一标识的队列对应了一个唯一的文件，该文件访问不需要加锁。

当使用压缩算法时候，我们会发现jdk中带有的`GZIPOutputStream`，`DeflaterOutputStream`的压缩速率不太理想，因此最终我们选择了`SnappyFramedOutputStream`这个广泛应用的压缩算法，做一个压缩率和CPU占用的权衡。

对于读取的反序列化，我们做了一个实现上的优化，只要进行一次线性访问就可以获取出对应的属性，而不是通过正则表达式的`split`进行三次线性扫描。具体实现可见 https://github.com/CheYulin/OpenMessageShaping/blob/master/src/main/java/io/openmessaging/demo/DefaultBytesMessage.java 中`public String toString()`和`static public DefaultBytesMessage valueOf(String myString)`这两个序列化和反序列化的方法。

其他选手还考虑了使用一些 hard coding的trick来进一步减小磁盘IO,因为理论上分析`4G * 0.2 = 0.8G`，只要能够使得最终的输出达到`0.8G`的大小的话，磁盘IO对于写线程来说总是异步的，因为写线程只是进行了内存的拷贝。我们最终没有选择进行hard coding，需要写磁盘的大小为`2.4G`，最终的成绩为: 生产耗时`73.952s`， 消费耗时`35.995s`， TPS为`363811`， 对应代码的版本为 https://github.com/CheYulin/OpenMessageShaping/tree/296f94e3b63f5d286d421a4ae180dd3af63be3b1 。

1.1.4 源代码

详细代码可见 https://github.com/CheYulin/OpenMessageShaping/tree/master/src/main/java/io/openmessaging/demo 下的文件。

1.2 复赛概览

1.2.1 赛题要求

进行数据库(从空开始)的主从增量同步，输入为10G顺序append的日志文件(只能单线程顺序读取1次，模拟真实流式处理场景)。日志文件中包含insert key(带有全部属性)/delete key/update property(单个)/update key这四种类型的数据变化操作(通过I/D/U以及其他日志中的字段标识)。其中，10G数据在server端，最后的查询结果需要落盘在client端，server和client是两台配置相同的16逻辑cpu核数的阿里云虚拟机。

1.2.2 赛题分析

赛题数据为单表的日志，并且有字段的长度有确定的范围，选手可以基于此进行优化。赛题最后的符合要求range在(1,000,000, 8,000,000)，最后产生符合的条数占比约为1/7，byte[]大小大约38MB,因此网络带宽不会成为系统瓶颈。比赛测试环境为16个logical CPUs，因此需要解决并发的一些问题，充分利用并行，来获取极致的性能。所有输入文件在ramfs中，顺序单线程访问所需理论时间2.5s，因此需要选手能够比较好地通过计算流水线，overlap处理，计算和IO(通过ramfs)。

1.2.3 赛题要点

文件读取通过mmap，减少内核态和用户态拷贝。建立好的流水线，处理好重放计算时候需要保证的顺序性，并且需要充分将IO和计算overlap在一起。并行eval产生byte[]，网络传输和落盘采用Zero-Copy，以减少内核态和用户态拷贝的开销。

1.2.4 数据库重放要点

数据库重放时候如果利用数据update key不会带来原来属性的特征，可以使用数组表示符合范围的数据库记录，否则可以通过hashmap加hashset进行表示。使用通用策略的时候，重放计算的内存占用为数据库的大小；使用针对题目专用策略时候，重放计算的内存占用为数组大小。数据库重放时候，若采用通用策略，则重放计算瓶颈在于内存占用和hashmap的index probing计算。

1.2.5 源代码

版本	链接
trick统计信息获取版本	https://github.com/CheYulin/IncrementalSyncShaping/tree/specialStatistics
trick版本	https://github.com/CheYulin/IncrementalSyncShaping
通用版本	https://github.com/CheYulin/IncrementalSyncShaping/tree/generalImpl8.9s

2. 核心思路

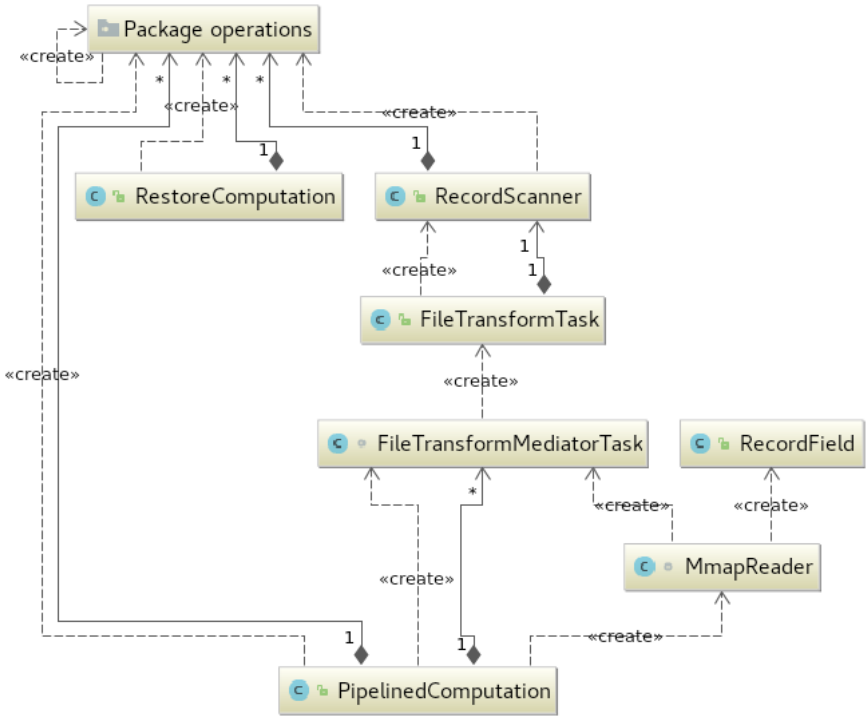
2.1 基本思路

本题主要涉及到重放算法的设计和Server-Client网络传输落盘的设计。

- 重放算法分为两个阶段，第一个阶段：单线程顺序读取十个文件，重放出数据库中最后时候符合主键在查询范围内的记录；第二个阶段：遍历第一阶段的记录数组，针对每一条记录，插入主键为key并且`byte[]`为value的`ConcurrentSkipListMap`中， 为之后产生出对应的文件作准备。
- Server段在程序启动时候，开启一个线程监听Client连接请求；在最后执行完第二阶段计算时候，遍历有序的`ConcurrentSkipListMap`， 产生出结果文件对应的`byte[]`，并使用 java nio 的`transferFrom`方式直接发送到Client并通过Client Direct Memory进行落盘。

2.2 第一阶段流水线的设计

整个重放算法有关的类都放在`server2`文件夹下， 其中的类关系如下图所示(通过jetbrains intellij生成)。



图中有四种不同的actor，这些actors的交互构成了完整的第一阶段计算的流水线：

2.2.1 actor 1: MmapReader(主线程)

职责：负责顺序读取十个文件，按64MB为单位读取，若文件尾部不满64M就读取相应的大小, 读取之后对应的`MappedByteBuffer`会传入一个大小为1的`BlockingQueue<FileTransformMediatorTask>`, 来让Mediator进行消费。因为阻塞队列的大小为1， 所以内存中最多只有三份`MappedByteBuffer`(分别于主线程/Mediator线程/BlockingQueue中)，总大小至多为192MB。

在获取下一块文件Chunk的时候，该Reader会判断是否已经初始化了关于单表的Meta信息。详细代码可见:
(其中RecordField类的类静态变量将用来记录这些Meta信息)。

2.2.2 actor 2: Mediator(单个Mediator线程)

职责：负责轮询`BlockingQueue<FileTransformMediatorTask>`来获取任务， 一个任务中包含一个`MappedByteBuffer`和对应的Chunk大小。

在收到任务后，Mediator负责分配， 保证每个Tokenizer and Parser处理的都是完整的块， 也就是说， 开始的index在`|mysql...`的`|`上， 结束的index在`\n`的后一个上。在这一步中， 需要由Mediator维护好Chunk中末尾'\n'之后的bytes， 这也是Mediator最关键的工作之一。

关于任务分配，Mediator通过sumbit的方式向Tokenizer and Parser对应线程池提交任务， 并获取`Future<?>`传入下一个FileTransFormTask， 因为重放计算要求保证顺序， 一个任务做完后放入计算队列之前需要等上一个任务结束， 以保证顺序重放的正确性。一开始`Future<?>`的类静态对象被初始化为`isDone = true`。这一步的依赖至关重要， 保证了Log重放时候的顺序性，并且在最后做完tokenizer和parser任务后再放入taskQueue的设计最大程度利用了CPU。Mediator在解耦和简化并行计算模型方面发挥了重要作用。也是这个简洁的流水线设计中必不可少的一环节。任务分配相关的核心代码在关键代码中(其中关键点在于start, end index的计算和prevRemainingBytes的维护以及prevFuture的维护)。

下面关键代码中相应的的`submitIfPossible(FileTransformTask fileTransformTask)`方法中`serverPCGlobalStatus[globalIndex]`是根据统计出来的有用的Chunk，这是看其他队伍实现了5s以内的版本，不得以想到的，因为理论分析只有这样作或者利用其他数据的特征才可能实现5s以内的版本。这个一个取巧之处。这个取巧之处才可能帮助很多队伍创造出5s以内的时间，因为这样做的话，极限就是 ***1s的评测程序开销 + 2.5s mmap load 10G文件开销 (完全和计算overlap,计算包括了tokenize, parse, restore) + 0.5s(并行eval，网络传输和落盘) = 4s***。

2.2.3 actor 3: Tokenizer and Parser for LogOperation(线程数为16的线程池)

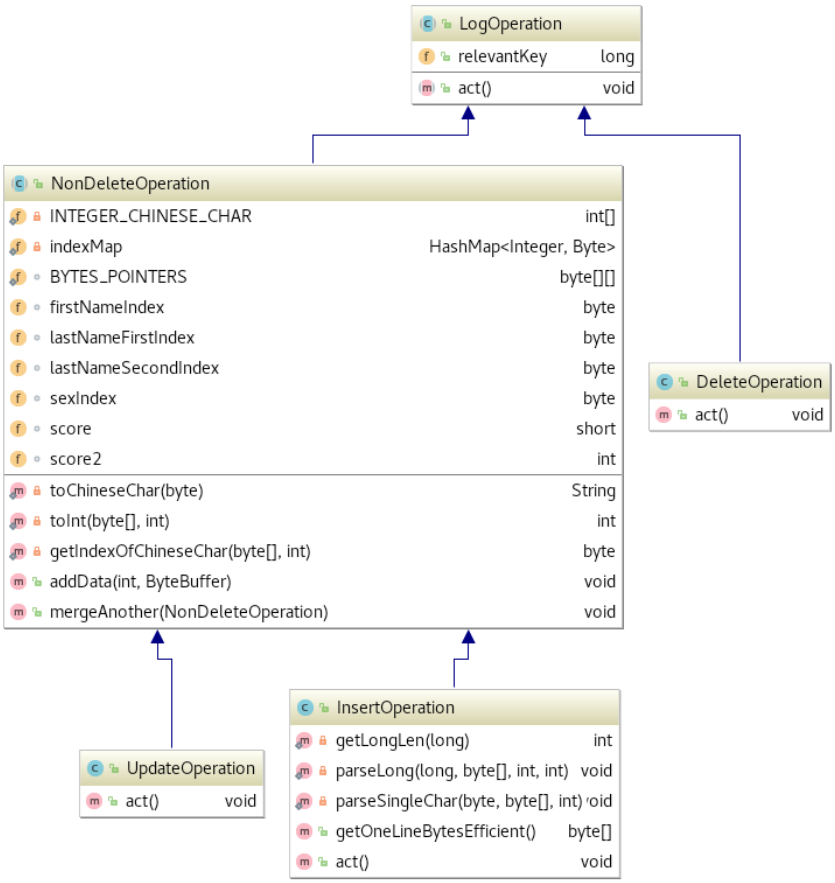
职责: 这个逻辑在`FileTransformTask`中，负责对分配到某区间ByteBuffer里面的bytes进行解析，产生出用于重放的LogOperation对象来。 其中主要涉及到主键的解析，类型的解析和必要时LogOperation对象的创建。每个`FileTransformTask`对应一个唯一的`RecordScanner`， `RecordScanner`中封装了解析LogOperation对象的内容。中间设计到了利用表Meta信息减少访问bytes的优化，例如Delete操作的所有field都可以跳过，这也是比较容易发现的一个优化点。

2.2.4 actor 4: Restore Computation Worker(单个重放计算线程)

职责：负责轮询获取任务进行计算， 当遇到大小为0的数组时候退出。重放计算线程轮询和退出的方式与Mediator类似，这里就不再给出。

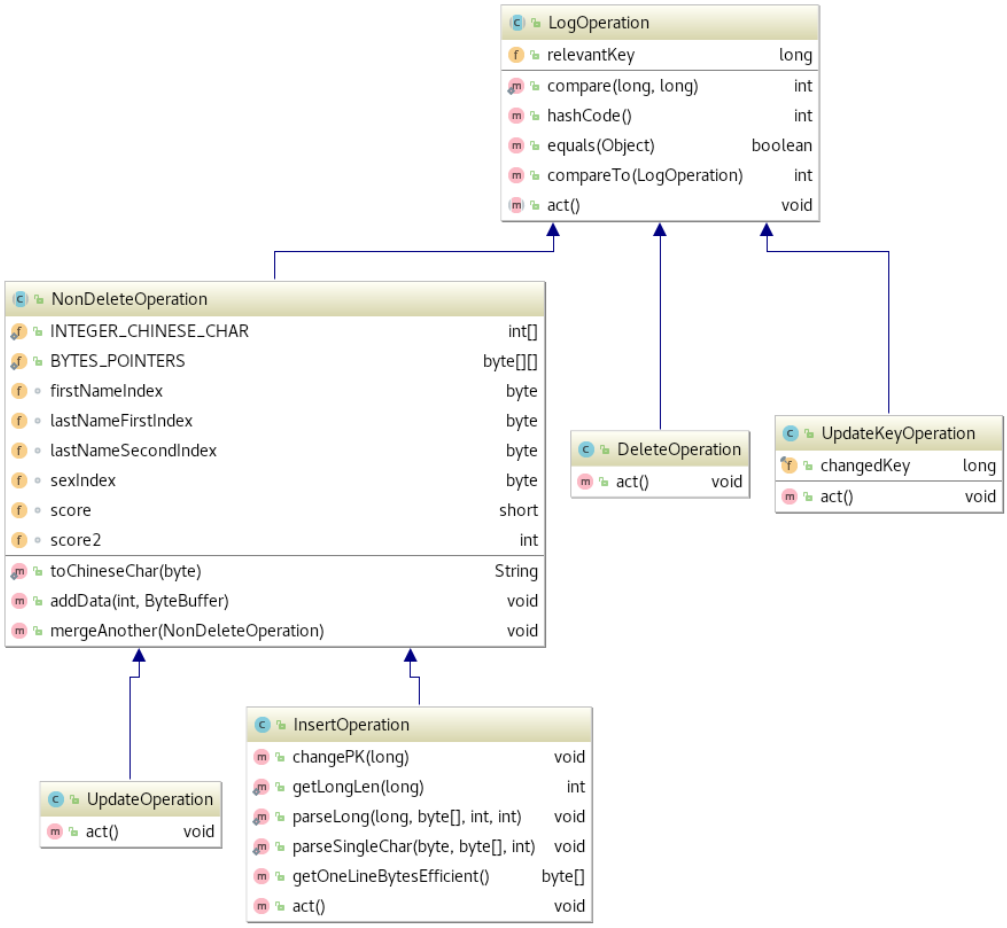
为了取巧使用array代替hashmap我们不得不发现一个重要的规律：***主键变更并不会带来原来主键的属性***，比如主键从1->3，那么主键1原来的属性一定会被全update或者3不在range范围中，那么update key的操作就可以简单变成两个操作，一个delete之前主键，另一个insert新的主键。 这样才使得我们只要keep在范围内的主键相关记录，比如只有1000000到8000000的key对应记录有用，不会出现2^63的key有用，所以才可以使用array。

取巧版本中，三种不同的LogOperation(`InsertOperation`, `DeleteOperation`, `UpdateOperation`)在重放过程中被进行了处理。LogOperation的相关类继承关系如下图所示((通过jetbrains intellij生成)):



如果不取巧，我们也参考Trove Hashmap实现了一个 efficient的 hashmap，另外通过另一个hashset来记录range范围内的记录有哪些，这个实现并且在其它地方都不取巧，我们可以获得8.9s的成绩。

在不取巧版本中，四种不同的LogOperation(`InsertOperation` , `DeleteOperation` , `UpdateKeyOperation` , `UpdateOperation`)在重放过程中被进行了处理。LogOperation的相关类继承关系如下图所示((通过jetbrains intellij生成)):



2.3 第二阶段Eval以及后续传输落盘

- 第二个阶段的计算：遍历第一阶段的记录数组，针对每一条记录，插入主键为key并且`byte[]`为value的`ConcurrentSkipListMap`中， 为之后产生出对应的文件作准备。这个过程是并行进行的，并且我们实现了更efficient的eval，详细可见`3.5 第二阶段 并行Eval`中对`public byte[] getOneLineBytesEfficient()`的描述。
- 在最后执行完第二阶段计算时候，遍历有序的`ConcurrentSkipListMap`， 产生出结果文件对应的`byte[]`，并使用java nio 的`transferFrom`方式直接发送到Client并通过Client Direct Memory进行落盘。

2.4 整体设计的工程价值分析

2.4.1 工程背景的契合

本题潜在的工程需求为：canal产生流式数据，接入本题实现的模块进行范围内的数据重放，可能会在server端重放 ***多个表多个范围，备份到多个client端***，来实现真正的实时数据同步。本题中，从ramfs中读取对应的数据就模拟了canal在线生成实时数据的场景；因为canal产生的数据是有严格顺序的流式数据，所以本复赛题目才规定了只能 ***单线程顺序读取文件*** 。

我们的第一阶段流水线设计中`actor 1: mmap reader`是符合了 ***单线程顺序读取文件*** 的要求的。我们以`64MB`为chunk size，读取了文件中的每一个chunk，末尾不足`64MB`以末尾所剩余大小来处理。并且在我们流水线的设计中，至多占用`64MB * (1 + 1 + 1 ) = 192MB`的 direct memory(mmap reader 1份， blocking queue中1份，mediator 正消费中 1份)，严格控制了direct memory的使用，并且可配置。当需要针对 ***多个表多个范围*** 重放时候，只要配置 chunk size就可以保证actor1 到 actor2 的流水线正常工作。

若要进一步支持 ***多个表多个范围*** 的重放功能，需要再基于我们的通用8.9s版本, 针对每个表设计对应的`LogOperation`相关类，以及修改`RecordScanner`(主要是基于表中固定字段值范围，进行无用byte跳跃的优化) 和`LogOperation`中关于单表有关的优化代码(主要是固定的字段的扁平化表示的修改，例如firstNameIndex修改为对应字段的index)。在完成`LogOperation`相关类，以及修改`RecordScanner`修改后，需要进一步修改`RestoreComputation`，为每一个表和范围创建一个单独的对象。最后流水线中数据流动的逻辑需要进行相应的修改，例如：增加blocking queue，以及block queue中放入任务逻辑的修改。但是我们设计中模块的许多基本逻辑是可以复用的。

2.4.2 健壮性 - 不同数据集/DML操作/pk的不同范围输入

我们的通用8.9s版本可以在不同数据集/DML操作/pk的不同范围输入条件下保证正确性。

在不同数据集/DML操作/pk的不同范围输入条件下保证正确性, 是因为我们的通用版本设计中使用了：hashmap(key为long, value为LogOperation)来记录数据库(含有垃圾，因为不remove,但包含数据库中当前所有信息和垃圾)，hahset(element为LogOperation)记录range范围内记录。

这两个`RestoreComputation`类静态变量如下所示，分别叫做`recordMap`(追溯database)和`inRangeRecordSet`(记录range范围内的记录)：

```
public static YcheHashMap recordMap = new YcheHashMap(24 * 1024 * 1024);
public static THashSet<LogOperation> inRangeRecordSet = new THashSet<>(4 * 1024 * 1024);
```

重放线程的逻辑就是顺序遍历取到的任务中每条LogOperation采取相应的行为。其中包含有`DeleteOperation`，`InsertOperation`， `UpdateOperation` 和`UpdateKeyOperation`这四种类型的操作。

```
static void compute(LogOperation[] logOperations) {
    for (LogOperation logOperation : logOperations) {
        logOperation.act();
    }
}
```

- 对DeleteOperation 操作：仅仅对`inRangeRecordSet`进行更新，若删除的LogOperation在range范围内就把其从`inRangeRecordSet`中删除。 不对`recordMap`操作的原因，是我们在保证正确性前提下，可以让`recordMap`中含有垃圾信息(被删除的记录)，这么做可以减少remove时候probing的cost以及去除状态数组的内存开销。

```
@Override
public void act() {
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }
}
```

- 对InsertOperation 操作： 对`inRangeRecordSet`进行更新，若insert的LogOperation在range范围内就把其加入到`inRangeRecordSet`中。对`recordMap`进行更新，直接插入。

```
@Override
public void act(){
    recordMap.put(this); //1
    if (PipelinedComputation.isKeyInRange(relevantKey)) {
        inRangeRecordSet.add(this);
    }
}
```

- 对UpdateOperation 操作: 仅仅对`recordMap`中记录, 因为普通属性变更不会影响主键。首先probing获取`insertOperation`，然后进行`insertOperation.mergeAnother(this)`落实这次属性的变更。

```
@Override
public void act(){
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    insertOperation.mergeAnother(this); //3
};
```

- 对UpdateKeyOperation 操作： UpdateKeyOperation相同于进行了一次DeleteOperation和一次InsertOperation，但是有一个注意点就是InsertOperation中需要带来有原来被删除的Operation对应记录中的所有属性。例如：UpdateKeyOperation进行了1->3的主键变更，那么我们需要首先获取所有1的属性，然后删除1对应记录，然后插入3并且3对应记录包含之前1的所有属性。下面代码中, 先从`recordMap`中取出UpdateKey之前的key对应记录(`InsertOperation insertOperation = (InsertOperation) recordMap.get(this);`)，然后判断如果这个变更之前的key在range内，把其从`inRangeRecordSet`删除。删除后，通过`insertOperation.changePK(this.changedKey);`得出新key对应的对象，然后插入到`recordMap`中；最后判断变更后的key若在range内，把其加入到`inRangeRecordSet`。

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }

    insertOperation.changePK(this.changedKey); //4
    recordMap.put(insertOperation); //5

    if (PipelinedComputation.isKeyInRange(insertOperation.relevantKey)) {
        inRangeRecordSet.add(insertOperation);
    }
}
```

2.4.3 健壮性 - pk外其他field不同范围输入/不同表结构

对于pk外其他field不同范围输入和不同表结构的支持需要修改对应的`NonDeleteOperation`和`RecordScanner`这两个类。详细可见下一章节中`3.3 第一阶段 actor3: Transformer(Tokenizer and Parser, 16线程的线程池)`的描述。

2.5 版本演进历史

访问对应版本可以直接点链接或者通过 https://github.com/CheYulin/IncrementalSyncShaping/tree/ 加上版本号；例如版本号为 90b9f3a9253b877ed57641bffe1d719038ffb7f4 ， 则访问链接 https://github.com/CheYulin/IncrementalSyncShaping/tree/90b9f3a9253b877ed57641bffe1d719038ffb7f4 。

日期	成绩	版本说明	commit版本号以及链接
06/19	59.268s	放弃拷贝文件倒着读思路，正着重放第一次正确版本，属性存储使用`byte[][]`	90b9f3a9253b877ed57641bffe1d719038ffb7f4
06/19	46.163s	使用`InsertOperation`直接存储对应记录中的属性信息，减少内存拷贝和额外的小对象创建	dae437872ab9534268dc30d578f6c882f54427d7
06/19	43.433s	优化`RecordScanner`中`getNextLong()`和针对单表schema不变特征添加`skipFieldForInsert(int index)`	9d29966ddc5c9d93cbcbb0332054c6e39dcfd9cc
06/21	39.265s	使用更健壮的`BlockingQueue`协调生产者给消费者(重放计算线程)任务，重放计算线程使用轮询来获取任务	1ac36f81ae7a080b84cde5d6447d83c679758f09
06/22	20.155s	针对单表中字段长度范围固定进行`RecordScanner`以及`InsertOperation`的优化	e8a6389c1fa81eb397339248b2f21fe289cc68c6
06/23	14.456s	重放计算使用`gnu trove hashmap`	f5811f02f46256ffbe026f08faff9b8b2b343814
06/24	10.867s	优化网络传输和落盘，实现Zero-Copy	2576b8b54dd0853d5a430a1955c5b334824cfef6

日期	成绩	版本说明	commit版本号以及链接
06/25	9.228s	优化if-else分支，使用多态，使用`gnu trove hashset`	e631f2652238ca6861adbe906f1227c7b22ba2fa
06/26	8.979s	机器状态比较正常时候(上午)	版本同上
06/27	7.906s	使用UpdateKey的trick，主键变更不会带来之前属性	2a956e523d366bacfbe3a3303ffa6e75dbfb1241
06/28	6.670s	UpdateKey trick，使用数组代替hashmap和hashset，优化并行eval	36bb463ffbcb86a4eaf303489beb6239938350ed
06/29	4.819s	使用终极trick，计算时候跳过不需要的文件chunk，并且不使用logger，但是还是在`MmapReader`读文件时候调用了`load`，保证了不违反规则	b31990db8e3b3bc3c5722fec6e2068904ce71dd8

3. 关键代码

3.1 第一阶段 actor1: MmapReader(主线程)

```
// 1st work
private void fetchNextMmapChunk() throws IOException {
    int currChunkLength = nextIndex != maxIndex ? CHUNK_SIZE : lastChunkLength;

    MappedByteBuffer mappedByteBuffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, nextIn
dex * CHUNK_SIZE, currChunkLength);
    mappedByteBuffer.load();
    if (!RecordField.isInit()) {
        new RecordField(mappedByteBuffer).initFieldIndexMap();
    }

    try {
        mediatorTasks.put(new FileTransformMediatorTask(mappedByteBuffer, currChunkLength));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

3.2 第一阶段 actor2: Mediator(单个Mediator线程)

- 轮询的逻辑如下代码所示：

```
mediatorPool.execute(new Runnable() {
    @Override
    public void run() {
        while (true) {
            try {
                FileTransformMediatorTask fileTransformMediatorTask = mediatorTasks.take();
                if (fileTransformMediatorTask.isFinished)
                    break;
                fileTransformMediatorTask.transform();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
});
```

- 在最后读取完所有文件块的时候， 主线程会发送一个任务，通知 Mediator 可以结束了。该逻辑如下所示：

```
try {
    mediatorTasks.put(new FileTransformMediatorTask());
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

- Mediator核心的协调调度代码如下：

```
private static Future<?> prevFuture = new Future<Object>() {
    @Override
    public boolean cancel(boolean mayInterruptIfRunning) {
        return false;
    }

    @Override
    public boolean isCancelled() {
        return false;
    }

    @Override
    public boolean isDone() {
        return true;
    }

    @Override
    public Object get() throws InterruptedException, ExecutionException {
        return null;
    }

    @Override
    public Object get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionExce
ption, TimeoutException {
        return null;
    }
};

private void submitIfPossible(FileTransformTask fileTransformTask) {
//        if (localPCGlobalStatus[globalIndex] == 1) {
    if (serverPCGlobalStatus[globalIndex] == 1) {
        prevFuture = fileTransformPool.submit(fileTransformTask);
        prevFutureQueue.add(prevFuture);
    }
    globalIndex++;
    int avgTask = currChunkLength / WORK_NUM;

    // index pair
    int start;
    int end = preparePrevBytes();

    // 1st: first worker
    start = end;
    end = computeEnd(avgTask - 1);
    FileTransformTask fileTransformTask;
    if (prevRemainingBytes.limit() > 0) {
        ByteBuffer tmp = ByteBuffer.allocate(prevRemainingBytes.limit());
        tmp.put(prevRemainingBytes);
        fileTransformTask = new FileTransformTask(mappedByteBuffer, start, end, tmp, prevFutu
re);
    } else {
        fileTransformTask = new FileTransformTask(mappedByteBuffer, start, end, prevFuture);
    }

    submitIfPossible(fileTransformTask);

    // 2nd: subsequent workers
    for (int i = 1; i < WORK_NUM; i++) {
        start = end;
        int smallChunkLastIndex = i < WORK_NUM - 1 ? avgTask * (i + 1) - 1 : currChunkLength
 - 1;
        end = computeEnd(smallChunkLastIndex);
```

```
            fileTransformTask = new FileTransformTask(mappedByteBuffer, start, end, prevFuture);

            submitIfPossible(fileTransformTask);
        }

        // current tail, reuse and then put
        prevRemainingBytes.clear();
        for (int i = end; i < currChunkLength; i++) {
            prevRemainingBytes.put(mappedByteBuffer.get(i));
        }
    }
```

3.3 第一阶段 actor3: Transformer(Tokenizer and Parser, 16线程的线程池)

这一块需要有基础的操作基本类型`NonDeleteOperation`和`RecordScanner`这两个类，在使用中有如下两个注意点:

3.3.1 NonDeleteOperation通用化注意点

由于实际生产环境中，数据库的schema基本是不会变化的，所以我们针对比赛数据对应的单表结构进行了存储的设计。

- 字符串类型数据针对每个字符index存储，数字类型直接存储

在`NonDeleteOperation`中，数据通过index存储了下来。详细信息如下所示，如果schema改变用户可以依据对应table的meta信息来改变存储结构。

```
byte firstNameIndex = -1;
byte lastNameFirstIndex = -1;
byte lastNameSecondIndex = -1;
byte sexIndex = -1;
short score = -1;
int score2 = -1;
```

我们团队实现了index和真实char 之间的转换。通过`String toChineseChar(byte index)`可以把对应index转化成为具体的`char`，通过`byte getIndexOfChineseChar(byte[] data, int offset)`可以把具体的byte[]转化成为index。现实生活中字符个数是有限的，所以我们才想到了该index的方式，而且关系型数据库中一般会specify对应字段最长长度，扩展代码时候可以类似 lastName进行处理，存多个index，如果index的字符比较多的话，可以直接考虑存储char(2 byte)。要扩展到实际中进行使用的话，需要修改`INTEGER_CHINESE_CHAR`为所有可能的char，或者直接考虑存储char(2 byte)(需要修改`byte getIndexOfChineseChar(byte[] data, int offset)`为直接返回`char`)。

- `NonDeleteOperation`中index转换相关核心代码如下：

```
public static int[] INTEGER_CHINESE_CHAR = {14989440, 14989441, 14989443, 14989449, 14989450,
 14989465, 14989712, 14989721, 14989725, 14989964, 14989972, 14989996, 14990010, 14990230, 14
991005, 14991023, 14991242, 15041963, 15041965, 15041970, 15042203, 15042712, 15042714, 15043
227, 15043249, 15043969, 15044497, 15044749, 15044763, 15044788, 15045032, 15047579,
15048590, 15049897, 15050163, 15050917, 15052185, 15056301, 15056528, 15106476, 15108240, 151
08241, 15111567, 15112334, 15112623, 15112630, 15113614, 15113640, 15113879, 15114163, 151184
81, 15118751, 15175307, 15176860, 15176880, 15176882, 15176887, 15178634, 15182731, 15240841,
 15249306, 15250869, 15303353, 15303569, 15307441, 15307693, 15308725, 15308974, 15309192, 15
309736, 15310233, 15313551, 15313816, 15317902};
private static HashMap<Integer, Byte> indexMap = new HashMap<>();
public static byte[][] BYTES_POINTERS = new byte[INTEGER_CHINESE_CHAR.length][];

static {
    for (byte i = 0; i < INTEGER_CHINESE_CHAR.length; i++) {
        indexMap.put(INTEGER_CHINESE_CHAR[i], i);
        BYTES_POINTERS[i] = InsertOperation.toChineseChar(i).getBytes();
    }
}

public static String toChineseChar(byte index) {
    int intC = INTEGER_CHINESE_CHAR[index];
    byte[] tmpBytes = new byte[3];
    tmpBytes[0] = (byte) (intC >>> 16);
    tmpBytes[1] = (byte) (intC >>> 8);
    tmpBytes[2] = (byte) (intC >>> 0);
    return new String(tmpBytes);
}

private static byte getIndexOfChineseChar(byte[] data, int offset) {
    int intC = toInt(data, offset);
    return indexMap.get(intC);
}
```

- `NonDeleteOperation`中属性变更相关核心代码如下：

其中`addData(int index, ByteBuffer byteBuffer)`在针对某个表，创建`InsertOperation`和`UpdateOperation`对象时候使用，`mergeAnother(NonDeleteOperation nonDeleteOperation)`在属性update落实的时候使用：

```
public void addData(int index, ByteBuffer byteBuffer) {
    switch (index) {
        case 0:
            firstNameIndex = getIndexOfChineseChar(byteBuffer.array(), 0);
            break;
        case 1:
            lastNameFirstIndex = getIndexOfChineseChar(byteBuffer.array(), 0);
            if (byteBuffer.limit() == 6)
                lastNameSecondIndex = getIndexOfChineseChar(byteBuffer.array(), 3);
            break;
        case 2:
            sexIndex = getIndexOfChineseChar(byteBuffer.array(), 0);
            break;
        case 3:
            short result = 0;
            for (int i = 0; i < byteBuffer.limit(); i++)
                result = (short) ((10 * result) + (byteBuffer.get(i) - '0'));
            score = result;
            break;
        case 4:
            int resultInt = 0;
            for (int i = 0; i < byteBuffer.limit(); i++)
                resultInt = ((10 * resultInt) + (byteBuffer.get(i) - '0'));
            score2 = resultInt;
            break;
        default:
            if (Server.logger != null)
                Server.logger.info("add data error");
            System.err.println("add data error");
    }
}

public void mergeAnother(NonDeleteOperation nonDeleteOperation) {
    if (nonDeleteOperation.score != -1) {
        this.score = nonDeleteOperation.score;
        return;
    }
    if (nonDeleteOperation.score2 != -1) {
        this.score2 = nonDeleteOperation.score2;
        return;
    }
    if (nonDeleteOperation.firstNameIndex != -1) {
        this.firstNameIndex = nonDeleteOperation.firstNameIndex;
        return;
    }
    if (nonDeleteOperation.lastNameFirstIndex != -1) {
        this.lastNameFirstIndex = nonDeleteOperation.lastNameFirstIndex;
        this.lastNameSecondIndex = nonDeleteOperation.lastNameSecondIndex;
        return;
    }
    if (nonDeleteOperation.sexIndex != -1) {
        this.sexIndex = nonDeleteOperation.sexIndex;
    }
}
```

3.3.2 RecordScanner通用化注意点

当前的RecordScanner利用了当前表中数据字段长度范围和fieldName的特点，在实际使用中需要修改RecordScanner得以适应其他表结构。

- 原本就不需要修改的函数如下:

```
private void getNextBytesIntoTmp() {
    nextIndex++;

    tmpBuffer.clear();
    byte myByte;
    while ((myByte = mappedByteBuffer.get(nextIndex)) != FILED_SPLITTER) {
        tmpBuffer.put(myByte);
        nextIndex++;
    }
    tmpBuffer.flip();
}

private long getNextLong() {
    nextIndex++;

    byte tmpByte;
    long result = 0L;
    while ((tmpByte = mappedByteBuffer.get(nextIndex)) != FILED_SPLITTER) {
        nextIndex++;
        result = (10 * result) + (tmpByte - '0');
    }
    return result;
}

private long getNextLongForUpdate() {
    primaryKeyDigitNum = 0;
    nextIndex++;

    byte tmpByte;
    long result = 0L;
    while ((tmpByte = mappedByteBuffer.get(nextIndex)) != FILED_SPLITTER) {
        nextIndex++;
        primaryKeyDigitNum++;
        result = (10 * result) + (tmpByte - '0');
    }
    return result;
}
```

- 由于我们使用了`RecordField`类记录一些表中的meta信息，其中不需要修改的函数如下:

```
private void skipKey() {
    nextIndex += RecordField.KEY_LEN + 3;
}

private void skipNull() {
    nextIndex += 5;
}

private void skipFieldForInsert(int index) {
    nextIndex += fieldSkipLen[index];
}
```

- 需要修改的函数主要有`skipHeader()`，`skipField(int index)`，`int skipFieldName()`:

`skipHeader()`主要skip了四个信息段，分别为msql-binlog-id(长度不固定)，因此我们先skip了一个最小的长度，也就是`20`，然后我们尝试找到`|`这个特殊的分割符，接着skip了3个信息段分别为timestamp/schema string/table string，这三个信息段的长度是固定的因此我们跳过了`34`。在通用化的过程中需要修改这一部分的逻辑以适应真实场景而不是单表日志。

`skipField(int index)`主要根据所读表中对应的字段长度范围skip掉无用的值，比如在delete操作时候，所有的属性值都是没有用的。在通用化的过程中需要根据对应表的字段长度范围修改对应的skip大小。

`int skipFieldName()`主要根据所读表中包含的字段field name的这个信息来跳过field name，直接找对应value，比如对于`first_name:2:0|NULL|郑|`我们只要能确定这个表的属性是`first_name`就可以跳过不需要的字节，直接取对应的value。

```
private void skipHeader() {
    nextIndex += 20;
    while ((mappedByteBuffer.get(nextIndex)) != FILED_SPLITTER) {
        nextIndex++;
    }
    nextIndex += 34;
}

private void skipField(int index) {
    switch (index) {
        case 0:
            nextIndex += 4;
            break;
        case 1:
            nextIndex += 4;
            if (mappedByteBuffer.get(nextIndex) != FILED_SPLITTER)
                nextIndex += 3;
            break;
        case 2:
            nextIndex += 4;
            break;
        default:
            nextIndex += 3;
            while (mappedByteBuffer.get(nextIndex) != FILED_SPLITTER) {
                nextIndex++;
            }
    }
}

private int skipFieldName() {
    // stop at '|'
    if (mappedByteBuffer.get(nextIndex + 1) == 'f') {
        nextIndex += 15;
        return 0;
    } else if (mappedByteBuffer.get(nextIndex + 1) == 'l') {
        nextIndex += 14;
        return 1;
    } else {
        if (mappedByteBuffer.get(nextIndex + 2) == 'e') {
            nextIndex += 8;
            return 2;
        } else if (mappedByteBuffer.get(nextIndex + 6) == ':') {
            nextIndex += 10;
            return 3;
        } else {
            nextIndex += 11;
            return 4;
        }
    }
}
```

3.4 第一阶段 actor4: Computation Worker(单个计算线程)

3.4.1 基于数组的实现

重放中，为了更memory-efficient，我们使用数组来模拟Hashmap表示对应的数据库，下标对应key, 引用对应value， 基于Range固定并且在int表示范围内

```
public static LogOperation[] ycheArr = new LogOperation[8 * 1024 * 1024];
```

重放线程的逻辑就是顺序遍历取到的任务中每条LogOperation采取相应的行为。

```
static void compute(LogOperation[] logOperations) {
    for (LogOperation logOperation : logOperations) {
        logOperation.act();
    }
}
```

- DeleteOperation的操作， 从数据库中删除记录

```
@Override
public void act() {
    ycheArr[(int) (this.relevantKey)] = null;
}
```

- InsertionOperation的操作， 数据库中插入新的记录

```
@Override
public void act() {
    ycheArr[(int) (this.relevantKey)] = this;
}
```

- UpdateOperation的操作， 从数据库中取出对应的记录，并进行属性更新

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) RestoreComputation.ycheArr[(int) (thi
s.relevantKey)]; //2
    if(insertOperation==null){
        insertOperation=new InsertOperation(this.relevantKey);
        RestoreComputation.ycheArr[(int) this.relevantKey]=insertOperation;
    }
    insertOperation.mergeAnother(this); //3
}
```

3.4.2 基于hashmap和hashset的实现

在不同数据集/DML操作/pk的不同范围输入条件下保证正确性, 是因为我们的通用版本设计中使用了：hashmap(key为long, value为LogOperation)来记录数据库(含有垃圾，因为不remove,但包含数据库中当前所有信息和垃圾)，hahset(element为LogOperation)记录range范围内记录。

这两个 `RestoreComputation` 类静态变量如下所示，分别叫做 `recordMap` (追溯database)和 `inRangeRecordSet` (记录range范围内的记录)：

```
public static YcheHashMap recordMap = new YcheHashMap(24 * 1024 * 1024);
public static THashSet<LogOperation> inRangeRecordSet = new THashSet<>(4 * 1024 * 1024);
```

- 对DeleteOperation 操作：仅仅对 `inRangeRecordSet` 进行更新，若删除的LogOperation在range范围内就把其从 `inRangeRecordSet` 中删除。 不对 `recordMap` 操作的原因，是我们在保证正确性前提下，可以让 `recordMap` 中含有

垃圾信息(被删除的记录)，这么做可以减少remove时候probing的cost以及去除状态数组的内存开销。

```
@Override
public void act() {
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }
}
```

- 对InsertOperation 操作： 对 `inRangeRecordSet` 进行更新，若insert的LogOperation在range范围内就把其加入到 `inRangeRecordSet` 中。对 `recordMap` 进行更新，直接插入。

```
@Override
public void act(){
    recordMap.put(this); //1
    if (PipelinedComputation.isKeyInRange(relevantKey)) {
        inRangeRecordSet.add(this);
    }
}
```

- 对UpdateOperation 操作: 仅仅对 `recordMap` 中记录, 因为普通属性变更不会影响主键。首先probing获取 `insertOperation` ，然后进行 `insertOperation.mergeAnother(this)` 落实这次属性的变更。

```
@Override
public void act(){
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    insertOperation.mergeAnother(this); //3
};
```

- 对UpdateKeyOperation 操作： UpdateKeyOperation相同于进行了一次DeleteOperation和一次InsertOperation，但是有一个注意点就是InsertOperation中需要带来有原来被删除的Operation对应记录中的所有属性。例如：UpdateKeyOperation进行了1->3的主键变更，那么我们需要首先获取所有1的属性，然后删除1对应记录，然后插入3并且3对应记录包含之前1的所有属性。下面代码中, 先从 `recordMap` 中取出UpdateKey之前的key对应记录 (`InsertOperation insertOperation = (InsertOperation) recordMap.get(this);`)，然后判断如果这个变更之前的key在range内，把其从 `inRangeRecordSet` 删除。删除后，通过 `insertOperation.changePK(this.changedKey);` 得出新key对应的对象，然后插入到 `recordMap` 中；最后判断变更后的key若在range内，把其加入到 `inRangeRecordSet` 。

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this); //2
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }

    insertOperation.changePK(this.changedKey); //4
    recordMap.put(insertOperation); //5

    if (PipelinedComputation.isKeyInRange(insertOperation.relevantKey)) {
        inRangeRecordSet.add(insertOperation);
    }
}
```

3.5 第二阶段 并行Eval

- 第二阶段的`byte[]` Evaluation是完全并行的，详细过程抽象出下面的代码, 其中`finalResultMap`为类型`public static final ConcurrentMap<Long, byte[]>`, 获取到的中间结果可以进一步被进行遍历生成最后有序的输出到文件的`byte[]`：

```
private static class EvalTask implements Runnable {
    int start;
    int end;
    LogOperation[] logOperations;

    EvalTask(int start, int end, LogOperation[] logOperations) {
        this.start = start;
        this.end = end;
        this.logOperations = logOperations;
    }

    @Override
    public void run() {
        for (int i = start; i < end; i++) {
            InsertOperation insertOperation = (InsertOperation) logOperations[i];
            if (insertOperation != null)
                finalResultMap.put(insertOperation.relevantKey, insertOperation.getOneLineByt
esEfficient());
        }
    }
}

// used by master thread
static void parallelEvalAndSend(ExecutorService evalThreadPool) {
    LogOperation[] insertOperations = ycheArr;
    int lowerBound = (int) PipelinedComputation.pkLowerBound;
    int upperBound = (int) PipelinedComputation.pkUpperBound;
    int avgTask = (upperBound - lowerBound) / EVAL_WORKER_NUM;
    for (int i = lowerBound; i < upperBound; i += avgTask) {
        evalThreadPool.execute(new EvalTask(i, Math.min(i + avgTask, upperBound), insertOpera
tions));
    }
}
```

- 其中`insertOperation.getOneLineBytesEfficient()`是一个优化点，如果使用StringBuilder实现会比较慢，我们的实现如下，避免使用StringBuild和调用append。
  在下面的代码中我们的实现主要使用了直接的`byte[]`的操作和自己写的转换`parseLong`和`parseSingleChar`，可以从原来基于StringBuild实现的 500ms cost减到 250ms。

```
private static int getLongLen(long pk) {
    int noOfDigit = 1;
    while ((pk = pk / 10) != 0)
        ++noOfDigit;
    return noOfDigit;
}

private static void parseLong(long pk, byte[] byteArr, int offset, int noDigits) {
    long leftLong = pk;
    for (int i = 0; i < noDigits; i++) {
        byteArr[offset + noDigits - i - 1] = (byte) (leftLong % 10 + '0');
        leftLong /= 10;
    }
}

private static void parseSingleChar(byte index, byte[] byteArr, int offset) {
    System.arraycopy(NonDeleteOperation.BYTES_POINTERS[index], 0, byteArr, offset, 3);
}

public byte[] getOneLineBytesEfficient() {
    byte[] tmpBytes = new byte[48];
    int nextOffset = 0;
    // 1st: pk
    int pkDigits = getLongLen(relevantKey);
    parseLong(relevantKey, tmpBytes, nextOffset, pkDigits);
    nextOffset += pkDigits;
    tmpBytes[nextOffset] = '\t';
    nextOffset += 1;

    // 2nd: first name
    parseSingleChar(firstNameIndex, tmpBytes, nextOffset);
    nextOffset += 3;
    tmpBytes[nextOffset] = '\t';
    nextOffset += 1;

    // 3rd: second name
    parseSingleChar(lastNameFirstIndex, tmpBytes, nextOffset);
    nextOffset += 3;
    if (lastNameSecondIndex != -1) {
        parseSingleChar(lastNameSecondIndex, tmpBytes, nextOffset);
        nextOffset += 3;
    }
    tmpBytes[nextOffset] = '\t';
    nextOffset += 1;

    // 4th: sex
    parseSingleChar(sexIndex, tmpBytes, nextOffset);
    nextOffset += 3;
    tmpBytes[nextOffset] = '\t';
    nextOffset += 1;

    // 5th score
    pkDigits = getLongLen(score);
    parseLong(score, tmpBytes, nextOffset, pkDigits);
    nextOffset += pkDigits;
    tmpBytes[nextOffset] = '\t';
    nextOffset += 1;

    // 6th score2
    if (score2 != -1) {
        pkDigits = getLongLen(score2);
        parseLong(score2, tmpBytes, nextOffset, pkDigits);
```

```
            nextOffset += pkDigits;
            tmpBytes[nextOffset] = '\t';
            nextOffset += 1;
        }
        tmpBytes[nextOffset - 1] = '\n';

        byte[] retBytes = new byte[nextOffset];
        System.arraycopy(tmpBytes, 0, retBytes, 0, nextOffset);
        return retBytes;
    }
```

- 第二阶段的后续处理可见代码， 生成出最后会落盘至文件的`byte[]`， 交给Server进行发送

```
public static void putThingsIntoByteBuffer(ByteBuffer byteBuffer) {
    for (byte[] bytes : finalResultMap.values()) {
        byteBuffer.put(bytes);
    }
}
```

3.6 第二阶段后 网络传输和落盘(Zero-Copy)

充分利用Direct Memory的特性，去除内核态和用户态拷贝。

- Client Side, API usage

```
FileChannel fileChannel = new RandomAccessFile(Constants.RESULT_HOME + File.separator + Const
ants.RESULT_FILE_NAME, "rw").getChannel();
nativeClient.start(fileChannel);
```

- 底层的实现, 使用了`outputFile.transferFrom(clientChannel, 0, chunkSize);`， 直接从网络的clientChannel Zero-Copy到对应文件落盘， 不拷贝到用户态空间

```
public void start(FileChannel outputFile){
    if(outputFile == null){
        return;
    }
    try {
        clientChannel.write(ByteBuffer.wrap("A".getBytes()));
        int chunkSize = recvChunkSize();

        int recvCount = 0;
        ByteBuffer recvBuff = ByteBuffer.allocate(chunkSize);
        while (recvCount < chunkSize){
            recvCount += clientChannel.read(recvBuff);
        }
        String[] args = new ArgumentsPayloadBuilder(new String(recvBuff.array(), 0, chunkSiz
e)).args;

        chunkSize = recvChunkSize();

        outputFile.transferFrom(clientChannel, 0, chunkSize);

        clientChannel.finishConnect();
        clientChannel.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

4. 比赛经验总结和感想

4.1 技术总结

经过初赛，我们明白了在rocketmq设计中pagecache的重要性，只有把文件读写的size压缩到pagecache对应大小才可以取得比较好的性能，学习到了一些快速的压缩算法，例如snappy和lz4。

复赛中，我们明白了内存访问的pattern的重要性，通过 `byte[][]` 的方式访问会比较慢，而通过扁平化的存储会比较访存友好。在实际应用中，需要针对具体的数据库表字段进行相应的优化。并且，我们认识到了在java中，默认的 `extends Object` 会带来将近8byte的开销，并且jvm会进行8byte内存对齐，在设计小对象的时候要格外小心。

复赛中，另一个考察点是计算和IO的overlap，我们需要设计出来合适的并行计算流水线，解决一些相关的同步和并发控制问题，来取得比较好的性能。另外hashmap和hashset的具体内存layout对于性能的影响也非常大，java自带的基于拉链和转化拉链为红黑树的组织形式开销会比较大，hashmap的probing方式对于性能影响也很大。我们最终选择了从trove hashmap开地址的方式进行修改，利用了trove hashmap中基于knuth书本上的probing方式，可以取得不错的性能。对应不作任何取巧的通用版本也可以在线上跑到8.9s的成绩。

4.2 感想

在之后的开发中，也需要注意操作系统和语言底层vm实现相关的内容，才能取得比较理想的性能。

我们通过阿里中间件的博客学习到了许多新的知识，之后需要多多关注业界的技术热点，从而自己提高工程能力，并且也可能从中找出数据库领域值得研究并且未被研究透的问题。