基于异步检查点机制的细粒度流分区负载均衡方法

**朱和一，张展，陈思远，潘江浩，万丁**

（哈尔滨工业大学 计算机科学与技术学院 黑龙江省 哈尔滨市 150001）

**摘要：**在流计算系统中，数据倾斜通常会导致负载不均衡进而增加算子处理数据的延时，严重影响系统本身的运行效率。处理数据倾斜的方法是提出更好的分组策略，同时需要状态迁移机制的支持。更改分组策略的机制不合适可能会导致流计算系统的一致性要求得不到保证。而且按照一次性迁移的流程实现的状态迁移机制会带来非常高的延时或者需要暂停流计算系统，显著地降低流计算系统可用性。因此我们基于异步检查点机制，提出流分区负载均衡方法，可以在保持分布式一致性的前提下变更分组策略，保持流计算系统的语义并且处理数据倾斜问题；提出运行时的细粒度状态迁移方法，可以减少迁移对系统延迟带来的影响。我们在Apache Flink平台上实现了细粒度迁移的负载均衡模型，并对系统参数进行了分析，实验表明我们提出的处在均衡模型可以在数据倾斜的情况下很好的均衡负载，而且划分后的细粒度迁移造成的延时高峰与传统的一次性迁移相比有明显减少。

**关键词：**流计算；数据倾斜；运行时重配置；状态迁移；状态划分；

**中图分类号：**TP302.1 **文献标志码：**A **文章编号：**1671-6841（2020）00-0000-00

**DOI：**

# 引言

分布式流计算系统不断从数据源获取数据，并在亚秒级的时间内给出运行结果。通常这些流计算系统可以长时间保持运行以处理数据，数据在统计学分布中表现为集中在某些区域，则称之为数据倾斜，这往往会降低系统响应速度和吞吐量。根据数据源产生数据的速率及数据的统计学分布调整流计算系统的负载可以提高系统的运行效率。

流计算系统中，数据在算子间的传输遵守特定的规则，即分组策略，这是决定算子负载是否均衡的主要因素。改变负载即更改分组策略，通常以高频数据为切入点，改变数据的分发规则，将更多数据发送给负载低的算子，可以实现负载均衡。常见的分组策略有shuffle grouping、partial key grouping、all grouping、direct grouping、fields grouping[1]等等，其中shuffle grouping不具备数据分类统计的能力；direct grouping和fields grouping是对每种数据指定发送位置，数据范围较大时不适用；partial key grouping的分组效果比较好，也可以应对数据倾斜问题，但是它额外的资源进行数据重放和数据去重。

现今常用的流计算系统（Flink[2]，Storm[3]，Dhalion[4]等）中算子处理数据时会生成一部分信息存在本地线程，称之为状态。调整流计算系统的负载需要重新调整状态的分布以满足分布式一致性的要求，即需要状态迁移。状态迁移通常有两种实现方式：离线式和运行时。离线式状态迁移（Flink，Dhalion，SEEP[5]）需要暂停或结束流计算系统，然后调整负载并重新分配状态，再继续运行或重启系统；运行时状态迁移（ChronoStream[6]，Gloss[7]）则可以在系统运行的同时调整负载并迁移状态，不需要暂停工作。这些状态迁移的实现方式均为一次性迁移完成所有状态，消耗了大量系统资源的同时也造成了很高的延时。

我们提出了一种运行时调整负载的分组策略以及迁移状态模型，提出负载计算方式，由负载的不均衡程度触发负载调整，并追踪高频数据，根据高频数据在算子中的分布生成新的分组策略，并在流计算系统中广播。上游算子收到新的分组策略后替换原有的分组策略，下游算子依照新的分组策略迁移状态，保证负载调整后依然满足分布式一致性的要求，保持流计算系统语义的同时处理数据倾斜问题。

我们设计实现了划分模型将状态迁移划分为多次细粒度迁移过程，测量上一次迁移花费的时间，作为反馈信息为下一次迁移的状态大小提供参考，调整划分次数权衡单次迁移用时和有效迁移效率，借助检查点机制以最小的延时进行运行时状态迁移，减少高并发的资源占用来提高系统性能。我们的实验表明我们提出的运行时负载均衡模型可以使负载稳定在设定的不均衡度阈值之下，并且将迁移造成的高延时分解到多次细粒度迁移中，大幅缓解了迁移造成的不可用问题。

# 相关工作

分组策略是流计算领域研究的热门方向[8,9,10]，这些策略大多数是为了处理数据倾斜问题而提出的，他们以高频数据为分析对象，重新为这些高频数据分配算子，从而实现调整负载。分组策略可以分为拆分键和不拆分键两大类。拆分键[11]是指某个键可能被分配给若干个算子，这些算子之后需要再加入一层聚合算子对拆分的数据重新聚合，这会占用额外的系统资源，也会让流计算系统更加复杂。不拆分键的分组策略则要求某个键只能被分配给一个特定的算子，这样不需要引入额外的聚合算子，但是在数据发生倾斜的情况下，由于严重的负载不均衡，算子的工作效率会大打折扣，及时调整负载可以减轻倾斜的影响[6]。我们采用不拆分键的分组策略，因此需要探测数据倾斜是否发生，并生成新的分组策略以调整负载，还需要高效的状态迁移机制提供支持。

前文介绍的分组策略往往需要状态迁移提供支持，但是没有对状态迁移过程进行详细的研究和说明，只是使用迁移代价来引导新分组策略的构建，因此我们需要一种高效低影响的状态迁移机制支持分组策略的调整。

目前业界最常见的状态迁移机制是“结束-迁移-重启”这一类型（Spark Streaming [12]，Apache Flink [2]），在需要迁移的时候先结束流计算系统的运行，再迁移状态，最后重新开启流计算系统。这类机制往往有检查点的支持，重启流计算系统后需要从最近的检查点开始运行。这一过程花费的时间往往是几十秒，而发生数据倾斜通常表示此时有大规模数据需要处理，使用这类型迁移机制造成的这段系统不可用时间是让人无法接受的。“暂停-迁移-继续”是对这一机制的进一步优化（Flux [13]，SEEP [5], IBM Streams [14], Chi [15]），需要迁移状态的算子暂停工作，进行迁移后再继续处理数据。暂停可能导致上游算子的数据积压过多，所以这类机制需要流计算平台本身的支持，部分暂停可能导致后续算子在对齐数据的时候需要缓存大量的数据，造成额外的资源消耗。“备份拓扑”则是保持流计算系统处在运行状态，迁移开始时启动对整个拓扑图或部分拓扑图的备份[6]，保持主拓扑和备份拓扑同时运行直到迁移完成，这类机制需要系统额外分配一倍或几倍的资源，还需要上游算子多发数据，下游算子对数据去重。我们提出了一种“部分暂停-迁移-继续”类型的迁移机制，并用迁移划分模型将迁移过程划分为多次细粒度迁移过程，减少暂停的持续时间，使得每次细粒度迁移对系统的影响降到最低。

# 分组策略构建模型和迁移划分模型

流计算系统中，各算子是由上下游的关系组织在一起的，算子间传输的数据由一个键和若干个值组成。上游算子把某一条数据传递给哪个下游算子（）是由这条数据的键（）以及分组策略（）共同决定的，表示为。键在一段时间内的数据源中出现的频率表示为，在各个键的出现频率基本相同即数据分布均匀的情况下，一致性hash分组策略可以把键均匀地分配给下游算子，保证各算子得到的数据量相差无几，只有在某些键的出现频率较高的情况下，才有可能引起下游算子负载不均衡。

针对数据倾斜的情况，我们使用分组策略构建模型生成新的分组策略，为这些高频键重新分配下游算子，一致性hash算法本身难以为小部分值重新绑定位置，而我们需要频繁改变分组策略，所以我们在一致性hash基础上提出两级路由策略，由动态路由和静态路由组成，动态路由专用于分配高频键，根据数据倾斜情况由迁移划分策略动态调整；静态路由则是一致性hash本身，均匀分配非高频键。如图1，分组策略生成模型先生成备选的分组策略，并利用下游算子的负载分布信息，计算出备选分组策略带来的均衡开销以及迁移开销，从中选择开销最小的策略作为新分组策略。下游算子的负载分布信息也用来作为触发迁移的参考值。

生成新分组策略后需要迁移状态来满足新分组策略的要求以及分布式一致性的要求，前文已经提到，目前的状态迁移会让系统在很长一段时间内处于不可用状态，因此我们提出状态迁移划分模型将一次迁移分解为多次细粒度迁移，减少单次迁移的不可用状态持续时间。迁移模型在算子发起迁移时开始统计本次细粒度迁移所用时间，以此作为迁移划分模型的参数。



**图1** 模型各功能间关系

下面分别介绍分组策略构建模型和细粒度迁移的划分模型。

## 分组策略构建模型

建立两级路由可以提高迁移过程中分组算法的更新效率。系统检测到数据倾斜后，根据新的高频键值，会生成新的分组策略，新旧两种分组策略在下游算子键值分布上存在较多差异。我们建立两级路由策略，动态路由优先分配差异键值，非差异键值按照旧的分组策略继续在静态路由中选择下游算子。

对于上游算子的输出端，构建两级路由分组算法模型。两级路由分组算法表示为。其中为动态路由，保存高频键值的路由表，直接将高频键值指派给下游算子；为非高频的普通键值的分组方法，我们使用一致性hash方式来为这些键值分配算子。对于某个待分配的键值，模型若在中检测到这一键值，则按照的规则分配，若没能检测到，则按照的规则分配。

### 一致性hash

采用一致性hash的优势是：1 确保非高频键值在下游算子中的分配是均匀的，例如，下游有N个算子的情况下，每个算子会大约分到总键值数量1/N的键值；2 对于下游可分配算子数量的改变，一致性hash可以用最小的迁移开销来适应（合并hash环中的两个节点即可）。该部分内容的时间复杂度为，桶优化技术[16]可以让hash环的查找复杂度降低到。

### 负载的计算

对于下游算子我们构建其负载评价模型，用来计算算子的负载大小以及各个算子负载的均衡程度。

#### 数据采样及迁移触发

流计算系统中的输入数据以键值对形式存在，若高频键值的频率过高则认为倾斜发生。解决数据倾斜问题，往往需要对近期的数据倾斜程度进行评估，即采样近期一段时间的数据，统计其中的高频数据。Space saving是常用的高频数据统计方法，从固定时刻开始采样数据，提取其中出现频率大于或接近阈值的作为高频数据。其采样开始时刻是固定的，所以方法无法很好的反应近期的倾斜情况，也有滑动窗口版本的space saving方法实现，但是其延时和系统资源开销过大。所以我们采用多个滚动窗口叠加的方法，模拟细粒度实时采样，从而满足对近期高频键值的频繁查询。最近一次滚动窗口中所有数据的键取值范围表示为。

高频键值的变化并不一定表示此时需要迁移，因为下游算子所获得的高频数据可能也是分布均匀的，迁移触发取决于下游算子的负载不均衡程度（均衡开销）是否大于触发阈值。

#### 内存型负载

内存型负载是算子在本地存储的状态，取决于用户定义函数，定义函数，表示这一复杂度。例如，算子用时间窗口保存所有输入数据，则；算子若使用word count这类计数函数，则。对于某个特定的算子，表示这个算子的内存负载，有：

（1）

其中为分组函数对的分组结果。

#### 计算型负载

计算型负载是算子的数据处理逻辑复杂度，定义函数，表示这一复杂度。例如，算子用时间窗口保存所有输入数据，每个新数据都需要与窗口中的所有数据进行比较，则。对于某个特定的算子，表示这个算子的内存负载，有：

（2）其中为分组函数对的分组结果。

#### 均衡开销（负载不均衡程度，Balance cost）

定义均衡开销，即向下游算子发送数据的不均衡程度。

（3）

其中type为负载类型，可以是计算类型(c)或存储类型(s)，从前文中提到的两种负载中选择类型。

总均衡开销定义如下：

（4）

#### 迁移开销（Migration cost）

迁移开销表示如下，由两部分组成，第一部分表示原先在动态路由、现在不在动态路由中的这些键值，若一致性hash的结果与原先的动态路由不符，则需要迁移来满足一致性hash；第二部分表示原先和现在都在动态路由中，但是由两种算法给出的路由路径不一样的键值，需要迁移来满足新的动态路由。表示出现在原先或现在的动态路由中的键值。

所以现在的动态路由的迁移开销表示为，意为迁移的状态在总状态中所占的比例。

（5）

（6）

（7）

### 分组算法的构建

用来表示新的分组算法，静态路由保持不变以避免大规模迁移，重新生成动态路由，迁移量初始值为，，运行算法1，算法的主要思想为：从最高频键开始，为这些高频键分配下游算子，每次分配都满足在有部分已分配高频算子的情况下，此次分配的位置是最优的。

| **Algorithm 1** |
| --- |
| for each do //枚举按频率排序的待分配键值 |
|  |
|  |
|  |
| for each do //N为下游算子数量，尝试将当前键值分给下游算子 |
|  |
|  |
| if then //当前键值分给下游算子的代价更小 |
| //更新目标和代价 |
| //更新迁移量 |
|  |

## 细粒度迁移的划分模型

细粒度迁移是将单次迁移分解为多次细粒度迁移，可以减少单次迁移过程带来的延迟开销。每次小规模迁移少数个键值的状态，并将这些键值加入动态路由，当两级路由所形成的路由规则与新分组策略的路由规则相同时，即替换为新分组策略，此时这一组细粒度迁移完成。由于动态路由的存在，静态路由不需要频繁改动，只需要在迁移完成后做一次改动，这样不需要多次构建分组算法，提高整体运行效率。

我们根据单次小规模迁移固有用时，单位数据量的迁移用时，推测朴素方法的迁移用时，为划分次数提供依据，进而计算剩余的划分次数。由于网络状况随时变化，我们则根据过往若干次的单位数据量的迁移用时，推测下一次迁移的单位数据量的迁移用时，进而更精准地计算划分次数

假设一次细粒度迁移不做任何状态的迁移，只进行其他必要的步骤，如对齐操作，则称这一次迁移为空迁移。一次空迁移的时间是固定的，随环境因素波动，定义为一次细粒度迁移的固有延时，表示为，其中，是检查点游标的对齐时间，取决于流平台的任务类型、拓扑图、输入流等，通常是固定值；是系统环境因素带来的延时，与网络状况等因素有关，而且会随时变动，我们认为其期望为。

### 迁移优先级

细粒度迁移中，状态的迁移有先后顺序由优先级决定。在所有待迁移的状态中，计算每个状态迁移后使得均衡开销减少的量，并以此为关键字排序，选取其中使得开销减少最多的状态优先迁移，从而保证前几次细粒度迁移可以最大程度上减少均衡开销，以实现对数据倾斜问题的快速响应功能。

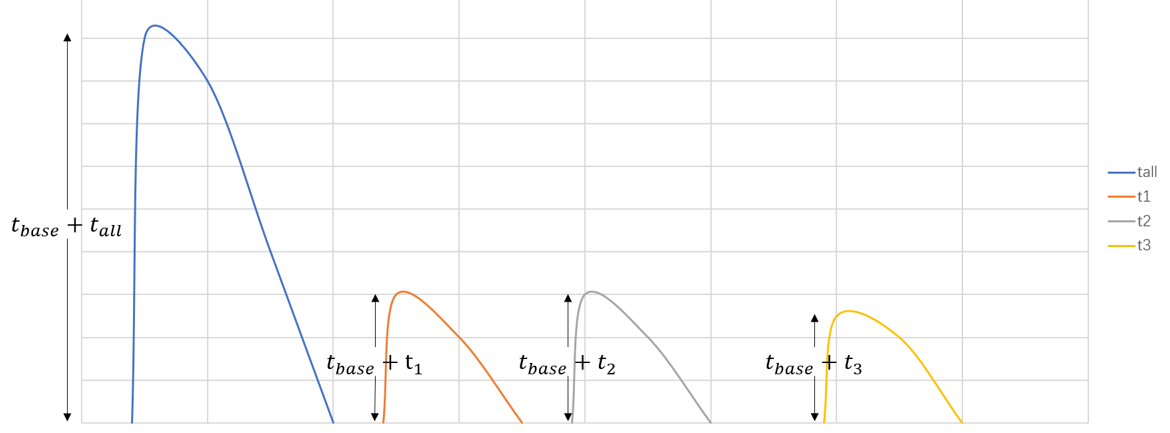
### 单次迁移量

构建迁移划分模型。假设不进行划分，对系统进行足够频繁的查询，将延时-时间图的延时峰面积定义为延时代价

（8）

其中是每单位长度的延时对延时代价的贡献，在系统运行前由实验测得，是固有延时的延时代价，满足 ，是延时峰的左边缘高度。

第次细粒度迁移延时峰的左边缘高度形式上定义为，其中为固有延时，为迁移所有状态用时，其中是此次迁移的状态大小，是迁移单位大小的状态的用时。

**图2** 迁移划分的原理

如图2，将左侧第一条曲线划分为右侧若干条曲线，即进行k-1次迁移划分将划分为k份。实际运行时我们无法事先知道 ，所以我们采用试错的方式，在迁移过程中不断逼近真正的划分次数。具体做法如下：先迁移一定数量（由参数初始迁移因子决定）的状态来确定，进而依据状态大小的推测计算，再计算出合适的划分次数，其中是划分成份后每次迁移的延时代价，计算方式与前文所述相同。由于系统情况在随时变化，所以后续每次迁移过后，测量计算得到最新的，重新计算迁移剩余状态所需的切分次数。

# 模型的实现

我们的负载均衡模型需要分组策略的自动化构建模块，迁移协调模块，迁移划分模块，检测数据倾斜以及触发迁移的模块，对各模块分发指令的控制模块。我们在Apache Flink开源流计算平台上实现了这些模块。本章节先介绍各模块之间的协作总体流程，再介绍一些复杂模块的实现方案。

## 总体流程

我们各模块的总体工作流程如图3。具体流程如下：

1) ①检测倾斜，②下游算子均衡开销大于阈值时触发迁移

2) ①以较小的均衡开销和迁移开销为代价，构建新分组算法，②建立迁移动态路由

3) 确定迁移划分次数和优先级

4) ①发出迁移指令，②迁移指令加入源算子，③将本次细粒度迁移的键加入迁移动态路由，④对齐游标后迁移状态，⑤收集本次细粒度迁移的延时数据，重新计算剩余状态需要的划分次数

5）重复执行4），直至迁移完毕

6）完成迁移并替换新的分组算法，同时取消迁移动态路由



**图3** 总体流程图

## 分组策略的自动化构建模块

构建分组策略需要的信息有：高频数据的统计信息、下游算子的高频键分布信息和流计算系统的拓扑。倾斜检测模块给出高频数据及其频率，该模块将这些信息发给下游算子，下游算子则根据这些信息将自身所包含的高频键发送给本模块，流计算系统的拓扑在系统开始运行时就可以获得。

本模块根据这些信息，构建新的分组策略，并生成新的迁移划分模块，将新分组策略交付于迁移划分模块。

## 迁移划分模块

迁移划分模块收集迁移用时作为反馈数据，对下次迁移的状态数量进行实时调整，从而保证划分次数是最合适的。本模块对外提供动态路由，暴露“下一个动态路由”接口，分组策略自动化构建模块调用这一接口获取下一个动态路由，从而构建新分组策略，并发送给下游算子。

## 迁移协调模块

迁移协调模块主要作用是下游算子迁移数据时的中转地。下游算子与本模块建立联系后，将不属于自身的状态发送至本模块，同时从本模块获取由其他算子发出的属于自身的状态，接收到状态后与本地状态进行合并，从而实现状态迁移。

一次性迁移需要在所有算子都发送完状态后才可以断开连接（如果接受状态的速度足够快），而细粒度迁移由于要迁移若干次，并不需要等待其他算子的工作，完全可以并行地运行，这正是并行度的意义。

## 基于异步检查点传播指令

我们需要在流计算系统中广播很多信息，例如更换分组策略的指令、分组策略的自动化构建模块生成的新分组策略、高频数据的统计信息、下游算子的高频键分布信息及其收集指令以及不均衡程度的测量信息、下游算子与迁移协调模块的交互信息等等。这些信息可主要有参数和指令两部分组成。

使用参数服务器对分布式系统的个节点分发参数和指令是一种常见的分布式协调方案，但是流计算系统中各算子接收到参数和指令时已经处理的数据数量可能不一致，即处理进度不一致，采用这种同时接收参数和指令的方案在流计算系统中会引起一致性问题。因此我们利用Apache Flink本身的异步检查点机制，将需要广播的参数和指令绑定在检查点barrier上，当算子收到检查点barrier后，会根据其中的指令来进行相应的操作（例如联系迁移协调服务器），并把barrier传递给下游算子。异步检查点机制中，同一并行度的算子实例接收到检查点barrier的时刻略有差异，但是在数据流处理的角度来看，这些算子是在同一数据后接收到了barrier，即在数据流方向是同步的。我们利用这一点实现的对算子信息的查询、对算子功能的更新等操作，在数据流方向是同步的，即这些操作是满足分布式一致性的。

我们复写了算子的处理逻辑，为算子增加了一个功能方法，在算子检查点对齐操作后，调用算子的功能方法。实现用户定义函数时，可以根据上下游的不同重写这一功能方法，从而实现算子的分组策略更新逻辑、迁移逻辑、以及倾斜检测和数据测量等逻辑。

# 实验及模型评价

我们实验部署在阿里云的6个虚拟机上， Intel ® Xeon ® Platinum 8269CY @ 2.5GHz CPU，32GB内存，100Mbit/s带宽，操作系统是CentOS 7。

实验分为两个部分，分别验证了程序正确性、讨论了参数的选取。

## 正确性验证实验

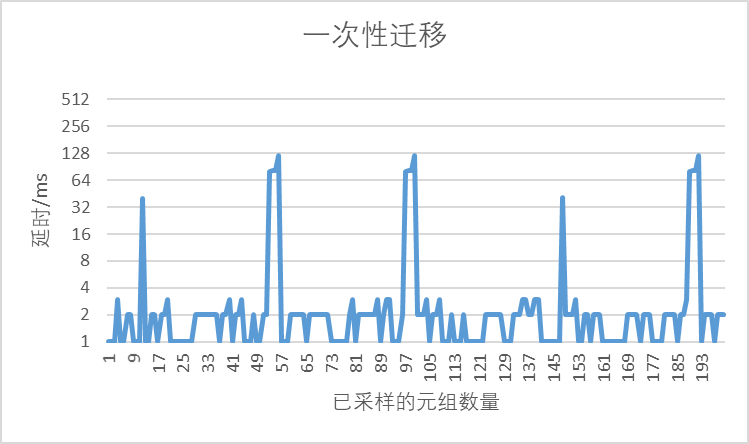
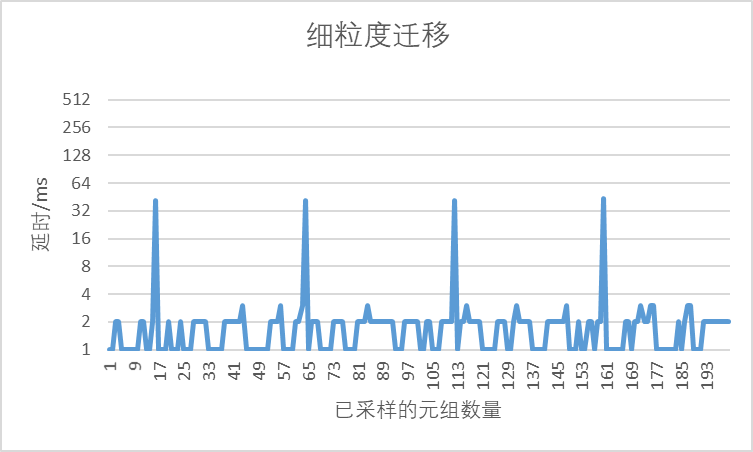
在Apache Flink中，算子的处理逻辑称为用户定义函数。根据用户定义函数的类型，正确性的验证需要考虑两种场景：

1. 使用无状态累积的用户定义函数，算子在本地存储的状态大小只取决于数据取值范围，与频率无关，因此高频数据的出现只对下游算子造成计算型负载不均衡，对状态迁移却是毫无影响，即高频与低频用在迁移上的时间是一样的。
2. 使用有状态累积的用户定义函数，算子在本地存储的状态大小主要由高频数据的出现次数决定，高频数据出现的越多，状态累积的越多，迁移时所花费的时间也就越多。

正确性验证实验使用的输入数据是拍卖行买家的竞拍数据，倾斜程度随时间变化非常剧烈。系统运行10分钟后开始使用容量为10000个数据的滚动窗口采样，获取窗口中延时的平均值作图。

### 无状态累积的用户定义函数对延时的影响

求每位买家竞拍价格的最大值作为用户定义函数，算子只需要保存价格最大值作为状态，所以该用户定义函数是无状态累积的。

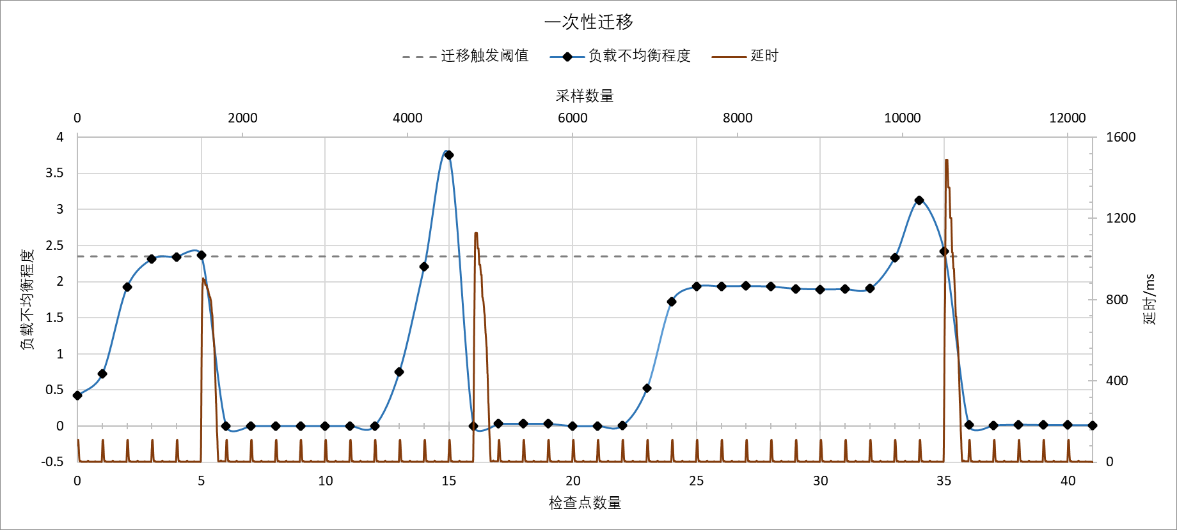
**图4** 使用无状态累积的用户定义函数进行延时对比

图4为流平台使用无状态累积的用户定义函数，在一次性迁移和细粒度迁移两种工作模式下的延时表现。

延时出现峰值的时刻表明系统进行了检查点，由于没有状态累积，迁移本身占用的时间很少，但是细粒度迁移因为准备工作需要额外消耗一些时间，导致延时的峰值比一次性迁移略微高了一些。

### 有状态累积的用户定义函数对延时的影响

求每位买家竞拍价格的中位数作为用户定义函数，算子需要保存每位买家以往所有竞拍价格作为状态，并从中选择中位数，该用户定义函数是有状态累积的。



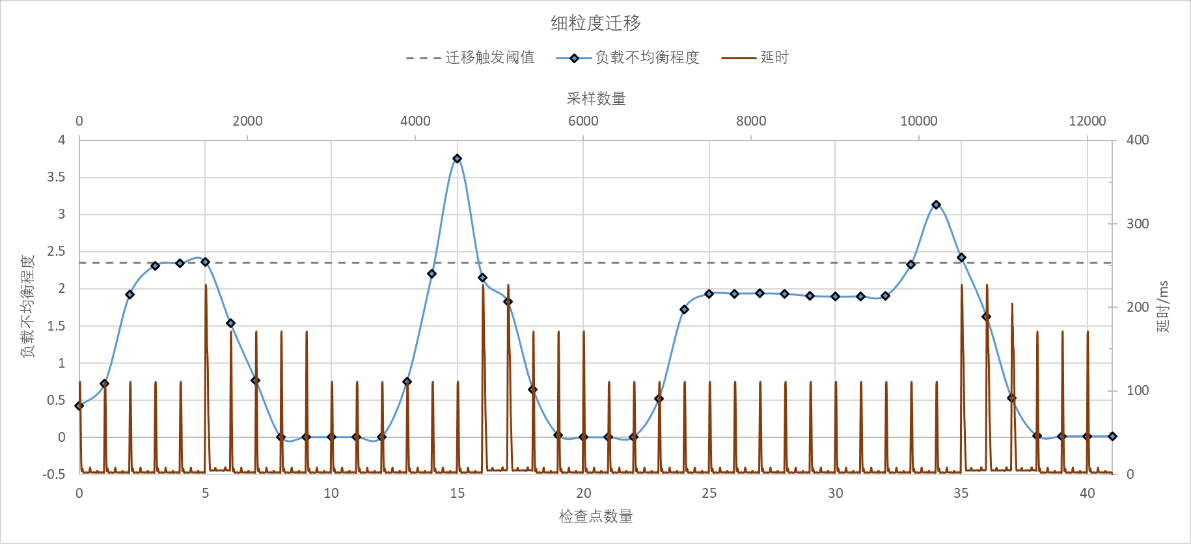


图5 使用有状态累积的用户定义函数进行正确性验证

源算子在负载不均衡程度大于阈值时触发迁移，本次实验的阈值设定为2.35，由图5的“迁移触发阈值”直线表示；散点图“负载不均衡程度”表示下游算子的负载不均衡程度随时间（检查点数量）的变化情况；折线图“延时”表示采样数据流经系统的时间。

此拍卖行的买家竞拍数据随时间变化较为剧烈，在短时间内会形成多次倾斜，图5中负载不均衡程度的三个高峰表示出现了三次高频的卖家数据，导致下游负载不均衡。而本系统可以根据设定的迁移触发阈值在运行时迁移下游算子的状态，及时的重分配可以避免过大的不均衡情况，从而让下游算子运行在相对健康的情况下。图中三次高峰达到阈值2.35后，系统触发了细粒度迁移，逐渐将负载的不均衡程度减少到0.5以下的可接受范围内。一次性迁移直接将负载不均衡程度降到最低；细粒度迁移分两次将负载不均衡程度降低到迁移触发阈值以下，然后继续迁移进一步降低不均衡程度。

一次性迁移出现了明显的延时峰，细粒度迁移的延时多次涨到220ms，却没有明显的延时峰。为了确保语义正确，一次性迁移需要下游算子同时链接迁移服务器，全部完成迁移后才可继续运行，所以延时会明显增加；细粒度迁移则不需要下游算子同时链接服务器，而是通过异步解耦合的方式，将需要迁出的状态存储在迁移服务器，同时读取需要迁入的状态，如此循环多次从而完成状态迁移。

发生严重的负载不均衡时，获得高频数据的算子CPU会超负荷运行，大量I/O操作读写状态也会增加算子的处理延时；发生轻度负载不均衡时，算子各部件均未满载，所以算子的工作效率与负载均衡时相差无几。因此负载不均衡程度低于迁移触发阈值时，可以认为负载是均衡的；高于迁移触发阈值则认为负载不均衡，图5中一到两次细粒度迁移即可将负载不均衡程度降至迁移触发阈值之下，使负载均衡，剩余状态分若干次迁移，保持负载均衡的同时不会引起高延时；而一次性迁移将不均衡程度降至迁移触发阈值之下后，继续花费宝贵的时间迁移剩余状态直至不均衡程度降至最低，以延时为代价达到负载均衡。

## 参数的选择

影响系统效果的参数主要有并行度和初次迁移量。并行度决定了单个下游算子所处理的数据量及负载，负载的分布情况对迁移有重要影响。初始迁移量用来计算合理的迁移划分次数，过小会导致测量不准确，从而导致计算有误，过大则会让系统的性能类似于一次性迁移。

### 并行度对延时的影响

固定输入数据调整下游算子的并行度，测量迁移延时峰的最大值作图。

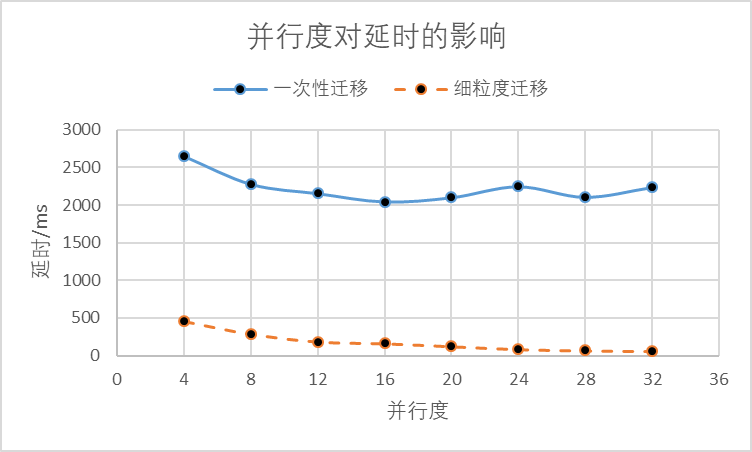


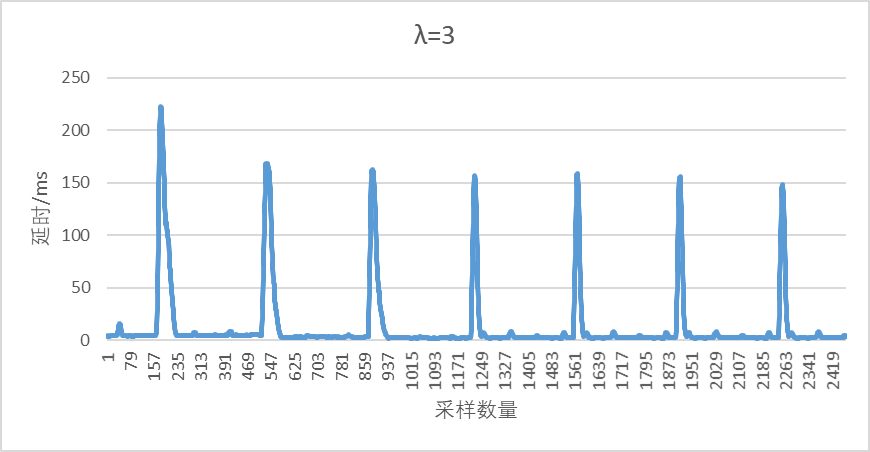
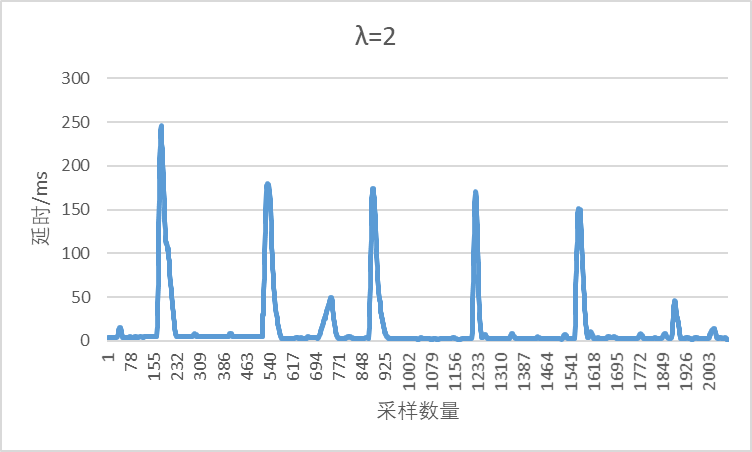
图6 并行度对延时的影响

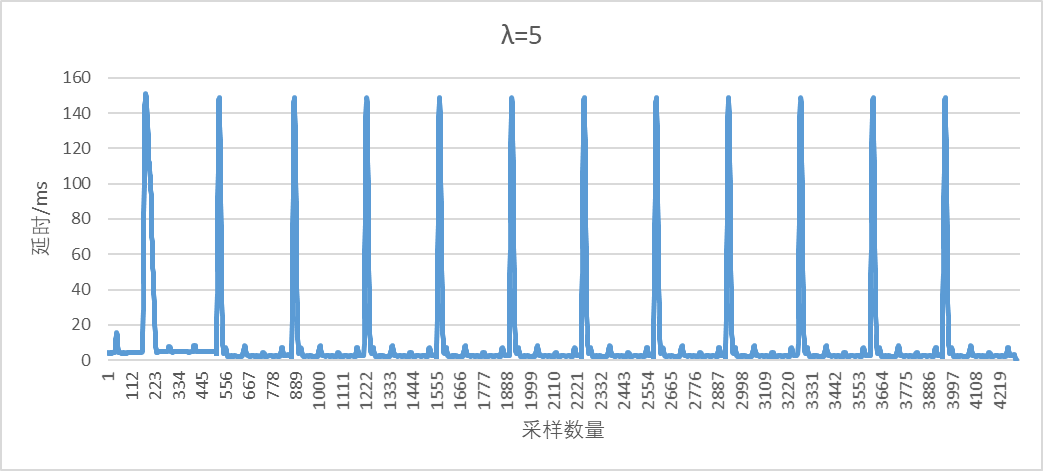
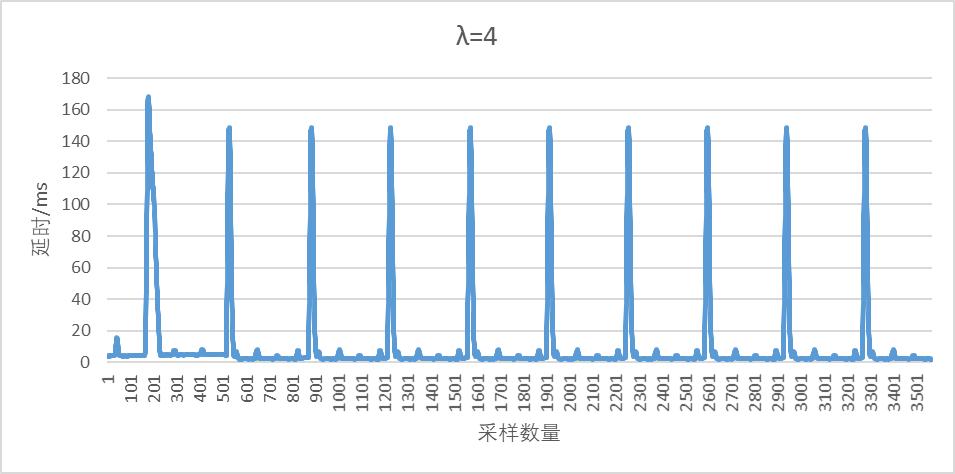
增大并行度可以减少每个下游算子存储的状态大小，减少状态传输用时。但是一次性迁移需要下游算子同时连接迁移模块（迁移服务器）同步传输数据，因此增加并行度，迁移模块需要同步的下游算子也随之增加。各算子进度及运行状态的不同，导致访问迁移模块的时刻也略有差异，并行度的增加导致等待同步的时间增加，进而增加数据处理的延时，所以图6中增加并行度不能有效减少一次性迁移造成的延时峰值。

对于细粒度迁移而言，各算子访问迁移模块是异步解耦合的，不需要等待和其他算子同步，状态的迁入和迁出可以同时进行，极大地减少了算子访问迁移模块的时间，不需要迁移状态的算子可以直接断开链接继续处理数据。因此延时取样时大多数算子延时较低，只有需要迁移的算子才会造成高延时，所以图6中细粒度迁移的系统平均延时也比较低。

### 初始迁移量对延时和划分次数的影响

调整影响细粒度迁移初始迁移量的参数，可以改变第一次迁移时的尝试状态数量，从而影响后续细粒度迁移的划分次数和延时。





**图7** 初始迁因子对延时的影响

图7中可以看出，初始迁移因子主要影响细粒度迁移的划分次数（不是确定次数，划分次数本身由倾斜程度，要迁移的数据量，等因素共同决定），此次实验固定其他因素，只有初始迁移因子变化。图中可以发现，较小时（2或3），初始迁移量较大，增大后，可以使各个划分的峰值更加均衡，但是没有必要无限制增大，因为单次迁移的最小延时的固定的，只需要增大到合理的值，使得第一次迁移和后续的迁移所造成的延时峰相近即可。

# 结论

我们提出了两级路由分组策略构建方法，利用下游算子的负载计算开销，从备选分组策略中选择最优策略；基于异步检查点机制，可以在保持分布式一致性的前提下变更分组策略，保持流计算系统的语义并且处理数据倾斜问题；提出运行时的分组策略更新及细粒度状态迁移方法，可以减少迁移对系统延迟带来的影响。我们在Flink平台上实现了细粒度迁移的负载均衡模型，并对系统参数进行了分析，实验表明划分后的细粒度迁移造成的延时高峰与传统的一次性迁移相比有明显减少，我们的负载均衡模型可以保持系统运行的同时及时对数据倾斜做出响应，避免负载的过度不均衡，保证系统高可用的资源分配。

# 参考文献

1. H. Chen, F. Zhang and H. Jin. Popularity-aware differentiated distributed stream processing on skewed streams. IEEE 25th International Conference on Network Protocols (ICNP), Toronto, ON, 2017, pp. 1-10.
2. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. Data Engineering, 2015, 38(4).

Ding, Jianbing & Fu, Tom Z. J. & Ma, Richard & Winslett, Marianne & Yang, Yin & Zhang, Zhenjie & Chao, Hongyang. Optimal Operator State Migration for Elastic Data Stream Processing. 2015.

A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. PVLDB, 2017, 10(12):1825–1836.

R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In Proceedings of the 2013 ACMSIGMOD international conference on Management of Data, 2013, pages 725–736.

Y. Wu, K. Tan. Chronostream: Elastic stateful stream computation in the cloud. IEEE International Conference on Data Engineering (ICDE), 2015, pages 723–734.

S. Rajadurai, J. Bosboom, F. Wong, and S. Amarasinghe. Gloss: Seamless live reconfiguration and reoptimization of stream programs. ASPLOS, 2018, pages 98–112.

1. Buğra Gedik. 2014. Partitioning functions for stateful data parallelism in stream processing. The VLDB Journal 23, 2014, 517–539.
2. Wang, Xiaotong & Fang, Junhua & Li, Yuming & Zhang, Rong & Zhou, Aoying. Cost-Effective Data Partition for Distributed Stream Processing System. Database Systems for Advanced Applications: 22nd International Conference, 2017, 623-635.
3. J. Fang, R. Zhang, T. Z. J. Fu, Z. Zhang, A. Zhou and X. Zhou. Distributed Stream Rebalance for Stateful Operator Under Workload Variance. IEEE Transactions on Parallel and Distributed Systems, 2018, vol. 29, no. 10, pp. 2223-2240.

Nasir, Anis & Morales, Gianmarco & García-Soriano, David & Kourtellis, Nicolas & Serafini, Marco. Partial Key Grouping: Load-Balanced Partitioning of Distributed Streams. ArXiv, 2015, abs/1510.07623.

1. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pages 423–438.
2. M. A. Shah, M. A. Shah, S. Chandrasekaran, J. M. Hellerstein, J. M. Hellerstein, S. Ch, S. Ch, M. J. Franklin, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In ICDE, 2002, pages 25–36.
3. IBM Streams (accessed: November 2019). https://www. ibm.com/ch-en/marketplace/stream-computing.
4. L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. PVLDB , 2018, 11(10):1303–1316.
5. Karger, D.R., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., Yerushalmi, Y. Web caching with consistent hashing. Compute Netw, 1999. 31(11–16), 1203– 1213.

Fine-Grained Partitioning Strategy for Load Balancing Based on Asynchronous Checkpoint Mechanism

**Zhu Heyi, Zhang Zhan, Chen Siyuan, Pan Jianghao, Wan Ding**

(Department of Computer Science, Harbin Institute of Technology, Harbin 150001, China)

**Abstract:** Data skewness is a very common phenomenon in stream processing which is the main cause of load imbalance. The state-of-art approaches of handling skewness migrate all the state at once, which may lead to a very high latency peak. For high availability, we propose a fine-grained partitioning strategy based on asynchronous checkpoint mechanism to deal with skewness, minimize the latency caused by state migration as well as maintaining consistency of distributed computing. Our evaluation shows that our system can migrate the state by some steps at runtime with low latency compared with all at once migration method.

**Key Words:** stream processing; data skewness; state migration; state splitting