硕士学位论文

基于异步检查点机制的细粒度流分区负载均衡方法研究

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

朱和一

哈尔滨工业大学

2020年6月

国内图书分类号：TP301.6 学校代码：10213

国际图书分类号：004.2 密级：公开

工学硕士学位论文

基于异步检查点机制的细粒度流分区负载均衡方法研究

|  |  |  |
| --- | --- | --- |
| 硕士研究生 | ： | 朱和一 |
| 导 师 | ： | 张展副教授 |
| 申请学位 | ： | 工学硕士 |
| 学科 | ： | 计算机科学与技术 |
| 所 在 单 位 | ： | 计算机科学与技术学院 |
| 答 辩 日 期 | ： | 2020年6月 |
| 授予学位单位 | ： | 哈尔滨工业大学 |
|  |  |  |

Classified Index: TP301.6

U.D.C: 004.2

Dissertation for the Master Degree in Engineering

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

|  |  |
| --- | --- |
| **Candidate：** | Zhu Heyi |
| **Supervisor：** | Associate Professor Zhang Zhan |
| **Academic Degree Applied for：** | Master of Engineering |
| **Speciality：** | Computer Science and Technology |
| **Affiliation：** | School of Computer Science and Technology |
| **Date of Defence：** | June, 2020 |
| **Degree-Conferring-Institution：** | Harbin Institute of Technology |

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，保证后续到达的数据可以均衡地分配给下游算子，同时需要状态迁移机制的支持，保证更改分组策略后，状态的存储位置可以对应新的分组策略。本文在Apache Flink计算平台上，提出了一种负载均衡模型，命名为S1负载均衡模型。

为实现该模型，本文提出了S1选择模型。定义负载计算方式以及负载不均衡代价和迁移代价的计算方式。本文需要在多次细粒度迁移中，选择收益最大（即代价最小）的状态优先迁移，剩余状态交由后续的细粒度迁移完成。S1选择模型提供了状态迁移的目的地和状态优先级排序队列，可以看作一种特殊的分组策略自动构建模型。

目前国内外关于流计算系统负载均衡的研究中，负载均衡策略使用的状态迁移机制都是按照一次性迁移的流程实现的，这种一次性状态迁移机制暂停流计算系统再对所有需要迁移的状态进行迁移，会带来非常高的延时，可以说是一种以显著地降低流计算系统可用性为代价的负载均衡方法。因此，本文提出细粒度状态迁移机制，将一次性迁移划分为多次细粒度迁移，从而降低单次迁移的代价，使迁移带来的延时不那么突兀，细粒度负载均衡策略基于此机制才得以实现。

为了实现细粒度状态迁移机制，本文还提出S1划分模型。本文发现一次细粒度迁移有固定的最小用时，因此本文收集上一次细粒度迁移的用时，从而推测迁移剩余状态用时，由剩余用时和最小用时可以计算出后续迁移最合适的次数以及单次迁移的状态大小。本文提出了两种划分策略：最大用时划分策略和自平衡划分策略。最大用时划分策略可以在用户指定的最大用时内完成一次细粒度迁移。自平衡划分策略的模型更加准确，而且不需要用户显式地给出最大用时这一要求，对细粒度迁移的适应性更强。

本文基于异步检查点机制，确定分组策略替换时机和状态迁移时机，可以在保持分布式一致性以及exactly-once语义的前提下变更分组策略以及迁移状态，从而优雅地处理数据倾斜问题。

本文在Apache Flink平台上实现了细粒度负载均衡模型，命名为S1负载均衡器，并对系统参数进行了分析，实验表明本文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。本文还对比了S1选择模型与其他分组策略生成模型的优劣以及自平衡划分策略和最大用时划分策略的差异。

**关键词：**负载均衡；分组策略；异步检查点机制；运行时状态迁移机制；细粒度状态迁移；

Abstract

Data skewness is a very common phenomenon in stream processing which is the main cause of load imbalance and may have side effect on the performance of stream processing system by increasing the latency of data processing. A more suitable partitioning strategy is needed to deal with data skewness by distributing the following data evenly to downstream operators. State migration mechanism is always needed to ensure that the position where state is hold in suits the new partitioning strategy. And inappropriate timing of partitioning strategy changes can cause consistency issues.

In the state-of-art research on load balance of stream processing system, the state migration mechanism used by load balance strategy is implemented according to the one-time migration process, which suspends the stream processing system and then transfers all the states that need to be migrated. Performing migration, which will bring very high latency, can be said to be a load balance method at the cost of significantly reducing the availability of stream processing system. Therefore, we propose a fine-grained load balance strategy, named S1 load balance model, to reduce the delay of one-time migration and complete load balance at a low cost.

We propose a fine-grained state migration mechanism, which divides a one-time migration into multiple fine-grained migrations, thereby reducing the cost of a single migration, and making the delay caused by the migration less abrupt. The fine-grained load balancing strategy is implemented based on this mechanism.

In order to support the fine-grained state migration mechanism, we propose the S1 selection model. We define the load calculation method and the calculation method of load imbalance cost and migration cost. Among the multiple fine-grained migrations, we need to choose the state with the most benefits (that is, the least cost) to migrate preferentially, and leave the rest to the subsequent fine-grained migration. The S1 selection model provides the destination of state transition and state priority ranking queue, which can be regarded as a special grouping strategy to automatically build the model.

In order to implement the fine-grained state migration mechanism, we propose the S1 partition model. We found that a fine-grained migration has a fixed minimum time, so we collect the time of the last fine-grained migration, so as to speculate the remaining state of the migration, from the remaining time and the minimum time we can calculate the most appropriate number of subsequent migrations and the state size of a single migration. This paper proposes two division strategies: maximum time division strategy and self-balancing division strategy. The maximum time division strategy can complete a fine-grained migration within the maximum time specified by the user. The model of the self-balancing division strategy is more accurate, and does not require the user to explicitly give the maximum time requirement, which is more adaptable to fine-grained migration.

Based on the asynchronous checkpoint mechanism, we determine the timing of the replacement of the grouping strategy and the timing of the state transition. We can change the grouping strategy and the state of migration while maintaining distributed consistency and exactly-once semantics, thereby elegantly handling the data tilt problem.

We implemented the fine-grained load balancing model on the Apache Flink platform, named S1 load balancer, and analyzed the system parameters. Experiments show that the proposed balance model can balance the load well under the condition of data tilt. And, the delay peak caused by the fine-grained migration after division is significantly reduced compared with the traditional one-time migration. We also compared the advantages and disadvantages of the S1 selection model with other grouping strategy generation models, as well as the differences between the self-balancing division strategy and the maximum time division strategy.

**Keywords:** load balance model; grouping strategy; asynchronous checkpoint mechanism; runtime state migration; fine-grained state migration;

目录

[摘要 I](#_Toc43374379)

[Abstract I](#_Toc43374380)

[第1章 绪论 1](#_Toc43374381)

[1.1 课题来源 1](#_Toc43374382)

[1.2 课题背景及研究的目的和意义 1](#_Toc43374383)

[1.3 国内外研究现状 2](#_Toc43374384)

[1.3.1 分组策略研究现状 3](#_Toc43374385)

[1.3.2 状态迁移机制研究现状 5](#_Toc43374386)

[1.3.3 现有研究现状的不足 6](#_Toc43374387)

[1.4 本文主要研究内容 7](#_Toc43374388)

[1.5 论文组织结构 8](#_Toc43374389)

[第2章 分组策略和状态迁移问题概述 10](#_Toc43374390)

[2.1 分布式流计算系统模型概述 10](#_Toc43374391)

[2.2 分组问题模型 11](#_Toc43374392)

[2.3 数据采样及迁移触发 12](#_Toc43374393)

[2.4 Flink现有机制介绍 12](#_Toc43374394)

[2.4.1 Flink 的运行时架构 12](#_Toc43374395)

[2.4.2 Apache Flink中的状态 14](#_Toc43374396)

[2.4.3 弹性伸缩有状态的流计算作业 15](#_Toc43374397)

[2.4.4 弹性伸缩时重新分配状态 16](#_Toc43374398)

[2.4.5 异步检查点机制 17](#_Toc43374399)

[2.5 运行时状态迁移 19](#_Toc43374400)

[2.6 S1负载均衡模型功能设计 20](#_Toc43374401)

[2.7 本章小结 21](#_Toc43374402)

[第3章 S1负载均衡模型 23](#_Toc43374403)

[3.1 S1选择模型与S1划分模型的协作 24](#_Toc43374404)

[3.2 用户定义函数分类及负载的计算 25](#_Toc43374405)

[3.2.1 一致性hash 26](#_Toc43374406)

[3.2.2 单数据负载 26](#_Toc43374407)

[3.2.3 算子负载的计算 27](#_Toc43374408)

[3.3 S1选择模型 28](#_Toc43374409)

[3.3.1 收益计算 28](#_Toc43374410)

[3.3.2 S1选择算法 30](#_Toc43374411)

[3.4 S1划分模型 32](#_Toc43374412)

[3.4.1 不划分策略 32](#_Toc43374413)

[3.4.2 最大用时划分策略 33](#_Toc43374414)

[3.4.3 自平衡划分策略 34](#_Toc43374415)

[3.5 本章小结 37](#_Toc43374416)

[第4章 模型的实现 38](#_Toc43374417)

[4.1 总体流程 38](#_Toc43374418)

[4.2 S1选择模块 39](#_Toc43374419)

[4.3 S1划分模块 40](#_Toc43374420)

[4.4 迁移协调器 41](#_Toc43374421)

[4.5 下游算子行为 42](#_Toc43374422)

[4.6 基于异步检查点传播指令 44](#_Toc43374423)

[4.7 本章小结 45](#_Toc43374424)

[第5章 实验及模型评价 46](#_Toc43374425)

[5.1 细粒度迁移与一次性迁移的延时对比 46](#_Toc43374426)

[5.1.1 无状态累积的用户定义函数对延时的影响 46](#_Toc43374427)

[5.1.2 有状态累积的用户定义函数对延时的影响 47](#_Toc43374428)

[5.2 细粒度迁移与一次性迁移对不同并行度的适应性 48](#_Toc43374429)

[5.3 S1选择模型与其他分组策略生成算法的对比 49](#_Toc43374430)

[5.4 最大用时划分策略和自平衡划分策略的分析 51](#_Toc43374431)

[5.5 本章小结 54](#_Toc43374432)

[结论 55](#_Toc43374433)

[参考文献 57](#_Toc43374434)

[攻读硕士学位期间发表的论文及其它成果 62](#_Toc43374435)

[哈尔滨工业大学学位论文原创性声明和使用权限 63](#_Toc43374436)

[致 谢 64](#_Toc43374437)

# 绪论

## 课题来源

课题来源于实验室项目：流计算平台的弹性伸缩和容错策略相关研究，项目立足于流计算领域，弹性伸缩的主要目标是保证状态一致性和exactly-once语义的条件下，研究细粒度局部在线重配置的机制和方法。在线重配置通常需要改变分组策略并调整算子状态的存储位置。本课题提出的策略和模型可以以低代价完成重配置，是该基金项目的重要组成部分。

## 课题背景及研究的目的和意义

随着5G时代的到来，互联网和物联网应用深度融合。在智慧城市、社交网络、移动数据和物联网、金融风控、推荐系统等诸多应用场景中，大规模的数据往往以数据流的方式高速且持续性的到达，数据的价值也会随着时间的流逝而快速降低，这就需要系统实时收集、处理和分析这些数据，进而能够以亚秒级的延迟实时交付数据的处理结果。对于这些计算实时性要求很高的场景，耗时的批处理系统，明显的难以满足需求，因此流式处理平台应运而生。

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。

目前需要状态迁移的负载均衡策略大多数没有考虑迁移过程对系统的影响，本文的目的则是实现高效的状态迁移机制，从而“优雅地”解决数据倾斜问题，以实现负载均衡。

本文提出的负载均衡模型可以在运行时调整分组策略以及迁移状态，本文提出负载计算方式，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。

本文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。本文的实验表明本文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。

本文在完成本文的大部分工作后，发现本文将状态迁移划分为多次细粒度迁移的想法与Java VM的g1垃圾回收器非常相似，这一垃圾回收器相比上一代回收器CMS来说，提供了最大停顿时间这一设定，在最大停顿时间内回收净收益最大的内存块，未能来得及回收的垃圾会在下一次垃圾回收时参与净收益计算，等待回收；这样垃圾回收的超长停顿时间被化解为多次短暂的停顿时间。本文将一次性状态迁移优化为细粒度状态迁移的思想与g1收集器不谋而合，因此将本文的负载均衡机制命名为**S1负载均衡模型**，重要的子模型也以S1命名。

## 国内外研究现状

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。

对于分布式流计算系统来说，基础的分组方法有两种，分别为随机分组和键分组。其中随机分组是将每个数据都等概率的随机分配到并行算子实例上，从而使得每个实例所处理的数据数量基本相同。随机分组通常用于无状态的算子。另外一种分组方法是键分组，是将数据中的特定字段的集合作为键并将具有相同键的数据分配给同一个实例。键分组通常用于有状态的算子。但是当数据倾斜较大时，键分组将高频键的数据分到同一实例，将导致下游算子实例的负载严重不均衡，从而对系统性能造成较大的影响。由于有状态算子相较于无状态算子的处理更加复杂，所以当前分布式流计算系统分组策略的研究对象主要是有状态算子基于键的分组问题。本课题讨论的所有分组策略也都是基于键的分组策略。

国内外目前研究的分组策略大多是从键分组的角度切入，对输入数据进行分组，在检测到负载不均衡时，由自动构建算法为一部分键重新分配下游算子，从而生成新的分组策略以均衡负载。本文的S1选择模型是一种特殊的分组策略，其工作过程中也需要为一部分状态重新分配下游算子，并对外提供满足新分配方式的分组策略。因此本文需要调查分组策略的国内外研究现状，从而优化S1选择模型。

### 分组策略研究现状

基于键的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]。本文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。

#### 不拆分键的分组策略研究现状

键分组是常用的不拆分键的分组策略，这种分组策略过于简洁，在数据倾斜的情况下会发生负载不均衡，且无法自我调整修正。国内外学者在此基础上提出了许多改进的分组策略。

Gedik[17]提出了基于多目标优化的READJ分组策略。READJ使用状态迁移重新调整负载，以适应倾斜数据；使用优化算法减少迁移代价；在运行时对下游算子并行度动态调整减少迁移代价；

Rivetti等人[18]提出了一种负载感知的分组策略DKG（Distribution-aware Key Grouping）。通过取样输入数据的分布情况，统计数据出现频率，用多线程的调度算法对数据进行分组。对于Zipfian分布[19]的倾斜数据流，该策略时能够维持下游算子负载均衡。

Caneill等人[20]依据局部性原理提出了一种基于物理位置的分配策略。该策略通过Metis划分库[21]分析上下游算子数据之间的相关性，将相关性较大的数据的键分配给处在同一物理节点上的上下游算子，减少跨机器的网络开销从而提高系统处理速度，降低处理延时

以上不拆分键的分组算法虽然能够在如状态转移、算子迁移等一些方面提高系统性能，并且能够在一定程度下如数据流键值分布倾斜较小时保证系统的负载均衡性。但随着数据流键值分布倾斜的增大，拆分键的分组算法将会导致严重的负载不均衡。

#### 拆分键的分组策略研究现状

由于不拆分键的分组策略在严重数据倾斜的情况下表现不够好，相关研究开始更多的关注拆分键的分组策略。基于键值拆分的分组策略可将同一键的数据分配到多个实例上处理，分组策略没有了一对一的局限性，每个输入数据的目标实例可选择性更大，更有利于负载的均衡[25][26][27]。

Nasir等人[28]开创了拆分键的分组策略的先河，提出PKG（Partial Key Grouping）分组策略。“the power of two choice”是经过理论分析和实验验证的重要结论[29][30]，PKG策略以此结论为出发点，使用两个hash函数为每个数据分别计算hash值，对应两个候选下游算子，将数据发送到已分配数据较少的下游算子中。这与排队论[49]中“两个服务共同读取一个队列效率高于两个服务分别读取两个队列”的结论非常相似，增加可选出口数量可以显著提高系统整体运行速度。

PKG算法还有两个改进版本，分别是D-Choices和W-Choices，也是由Nasir等人提出。他们发现在数据倾斜非常严重的情况下，两个候选下游算子无法有效缓解数据倾斜带来的压力，此时需要增加候选下游算子数量。W-Choices中，候选下游算子数量与总下游算子数量相等，即所有下游算子均可被分配；D-Choices则是提出了一种估算方法来计算合适的候选下游算子数量。其实验结果表明在一些特定数据集上，这两种改进版本的分组策略比原始的PKG效果更好，但是分配更多的下游算子就意味着下游算子需要更大内存来维护状态。

Chen F.等人[34]提出了名为NAG网络感知分组算法。NAG算法对每个上游算子分别进行分析，依据下游算子与上游算子的网络距离为每个下游算子实例生成相应的优先级和代价，网络距离越大代价越大，优先级也越低。NAG算法运行在已有分组策略基础上，若已有分配策略指定的算子实例代价已经达到阈值，则从优先级最高的算子实例开始搜索，找到代价尚未超过阈值的算子。若所有算子代价都已经超过阈值，依据网络距离和算子的负载重新设定阈值和代价。NAG算法更适用于部署在机架间的流计算系统，算子间网络差异越大，NAG算法的优势越明显，越能增加吞吐量减低延迟。

Pacaci等人[35]提出了一种需要感知数据分布的贪心分组算法。为了缓解数据倾斜对系统负载不均衡的影响，该算法采用了拆分高频键的方法，与朴素的拆分方法不同，该算法权衡了下游算子维护状态的内存开销，因为拆分后的高频键需要多个下游算子接受并存储相应的数据，这比不拆分键的下游算子要多申请一些内存空间。该算法将数据拆分并分配给下游算子时，会对这一过程造成的内存开销和负载均衡进行加权，根据加权结果选择最合适的下游算子作为目标。该算法使用尽量少的额外内存来进行负载均衡。

以上的大部分拆分键的分组策略在面对倾斜数据流时，仍然能够保持系统负载的均衡。但以上策略在多候选实例的选择上多使用随机的方法，将键的拆分数设置的偏大，从而增大了下游算子维护状态内存开销。因此选择合适的键值拆分数对于减少系统内存开销具有十分重要的意义。另一方面，以上策略没有考虑到数据流分布随时间变化以及系统异构性对分组策略的影响。

### 状态迁移机制研究现状

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，因此本文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。目前SEEP和Chi的实现是一次性迁移，局部暂停流计算系统后，在一段时间内将生成新的分组策略为状态重新分配位置，并让下游算子之间自发互动迁移（SEEP）或借助中间协调器（Chi）完成迁移。这两种迁移机制的实现都借助了流计算系统的检查点机制，以检查点信号触发状态的迁入迁出。

Shukla[18]等人提出了两种状态迁移机制，并在Apache Storm平台做了实现。Storm平台本身的状态迁移需要关闭全部算子，迁移完状态后再重启拓扑。Shukla等人的第一种状态迁移机制是先暂停源算子的数据输出，然后让下游算子继续处理数据直到流计算系统中所有数据都被处理完，即数据排空后，对状态做检查点，然后迁移状态，这种做法的一个优点是不需要考虑考虑数据重放的问题，因为这一过程没有重复处理数据。他们第二个机制是通过优化检查点过程来减少暂停时间。

Fernandez[19]等人提出的弹性伸缩流计算系统中，他们使用虚拟机池来快速扩容。他们的状态迁移机制也是“暂停-迁移-继续”，但是，他们为了提高容错性需要频繁地做检查点，这会给系统带来非常多的额外延时开销，他们后来提出了一种重复使用检查点的优化策略，该策略可以减少做检查点的时间，从而减少状态迁移机制的延时开销。

“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。

本文提出了一种“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。

### 现有研究现状的不足

通过对国内外研究现状总结出当前研究的不足如下：

（1）当前国内外研究中，拆分键的分组策略研究较多，而不拆分键的分组策略的研究则相对较少。不拆分键的分组策略虽然能够保持较小的下游算子维护状态的内存开销，但在面对倾斜数据流时无法较好的均衡负载[37]。拆分键的分组策略在对键进行拆分时多使用随机性方法如哈希函数等，从而为了减少随机性对算法性能的影响，会将键值拆分数设置偏大甚至等于下游算子实例个数，造成下游算子维护状态的内存开下较大。拆分键的分组策略还需要在下游算子之后加入一组聚合算子将拆分后的键重新聚合在一起以便正确统计数据。

（2）当前国内外研究中，对于倾斜数据的检测和负载均衡触发时机没有明确的结论，大部分分组策略从流计算平台运行开始统计倾斜数据，在需要负载均衡以及状态迁移时，使用的并非近期高频数据，而是从系统运行之初到负载均衡发起时刻的高频数据；发起负载均衡的时机大多是人为指定的，而不是由分组策略或倾斜检测模块智能确定。

（3）当前国内外研究中，对状态迁移的具体机制少有详细的介绍，只是分组策略用到了状态迁移，在重新分配状态时考虑了迁移的代价，其实验内容大多是在对比新旧两个分组策略在负载均衡上的优劣以及生成新分组策略的时间，但是没有考虑从旧分组策略变换到新分组策略这一过程的代价，例如系统暂停时间，CPU和内存的额外开销。大多数流计算平台不支持运行时部分暂停的功能，有状态的拓扑在运行时更改分组策略无法保证exactly-once等语义。

## 本文主要研究内容

通过对流计算平台分组策略国内外研究现状的分析和总结，本文着手于不拆分键的分组策略，采用近期高频数据检测和运行时状态迁移等技术优化该策略在数据倾斜时的表现。

本文的具体研究内容如下：

（1）基于异步检查点机制提出S1负载均衡模型。S1负载均衡模型由S1选择模型和S1划分模型两个主要部分和一些辅助部分组成。在检测到负载不均衡时，触发S1选择模型，为高频数据重新分配新的下游算子，并将高频数据按照迁移后的收益排序，在S1划分模型选取本次迁移的状态数量后，生成新的分组策略交付辅助模块，辅助模块将新分组策略绑定在触发检查点的信息上。异步检查点机制天生具有保持分布式一致性和exactly-once语义的特点，因此在算子接收到检查点后，做出相应的动作（例如替换分组策略，迁入迁出状态）也不会破坏一致性和语义。

（2）研究用于负载均衡的S1选择模型。拆分键的分组策略在实际应用中，需要在下游算子之后增加一组聚合算子，重新整合拆分的键，以确保查询结果正确，这会显著增加数据处理延时；下游算子维护状态的内存开销也比不拆分键的策略大。所以本文选择不拆分键的分组策略，对于其处理数据倾斜能力较差的缺点，本文提出高频数据检测模型和S1选择模型。高频数据检测模型用来检测一段时间内的高频数据，S1选择模型主要对高频数据进行选择, 通过改变高频数据的分组位置达到均衡负载的目的。本文提出负载的计算方式，负载不均衡程度的衡量方式以及迁移代价的计算方式，基于这些量化方式，对于某个高频数据，S1选择模型可以选择出这个高频数据分配到哪个下游算子为系统负载均衡带来的收益最大，以及将状态迁移到这个算子需要的代价，以净收益为基准对高频数据进行排序，得到高频数据迁移的优先级队列。本文提出的模型可以在运行时实时调整，以应对短期内的高频数据爆发。

（3）研究化解系统暂停延时高峰的S1划分模型。当前国内外研究中，使用状态迁移的分组策略大多只考虑了减少迁移的数据量，对迁移是否需要暂停全部流计算系统和暂停时间长短没有详细研究，对迁移的实现机制也没有足够说明。本文基于Apache Flink实现了运行时状态迁移机制，考虑到迁移需要暂停流计算系统的运行会造成较大延时，本文提出S1划分模型，将S1选择模型得到的优先级队列划分为多个细粒度迁移队列，将迁移造成的延时分散于各次细粒度迁移中，避免出现单次过大的延时。S1划分模型选择出本次需要迁移的状态数量交付S1选择模型，由S1选择模型生成与本次细粒度迁移相对于的分组策略，交付辅助模块。本文提出的模型可以在运行时实时调整，以应对短期内的高频数据爆发。

## 论文组织结构

第一章，绪论。先介绍课题的来源和课题的研究背景及意义，然后分析国内外研究现状及不足，然后概述本文研究内容，最后介绍本文组织结构。

第二章，S1负载均衡模型。先对流计算系统和分组问题分别建立数学模型，然后研究了数据采样和迁移触发的方法，对Apache Flink平台的状态迁移机制进行分析并指出其中不足，然后介绍了异步检查点机制，最后总结前文内容，简单阐述S1负载均衡模型的设计需求。

第三章，S1负载均衡模型。先介绍了S1负载均衡模型的设计和工作原理，解释了它是如何完成负载均衡的。然后介绍S1负责均衡模型的两个重要组成部分：S1选择模型和S1划分模型，说明两个模型之间的协作关系，然后为流计算系统的各种用户定义函数研究负载计算方式，之后介绍S1选择模型如何为高频键选择下游算子，以及S1划分模型的两种划分策略。

第四章，模型的实现。先介绍了本文所实现的项目整体工作流程，如何对重要模块进行介绍，指出实现的关键点。

第五章，实验及模型评价。实验部分先将细粒度迁移和一次性迁移做了对比，然后展示了细粒度迁移在不同并行度下的适应性，之后对比了S1选择模型和其他分组策略的优劣，最后分析了最大用时划分策略和自平衡划分策略的差异。

# 分组策略和状态迁移问题概述

为实现负载均衡，本文需要为流计算系统更新针对高频数据的分组策略，并同步迁移下游算子存储的状态以满足分布式一致性的要求。S1负载均衡模型是本文提出的用于在流计算系统中运行时均衡负载的模型，该模型由S1选择模型和S1划分模型以及一些辅助模块组成。S1选择模型和S1划分模型的最终结果表现为生成一种分组策略，该分组策略在流计算系统中广播，算子依据这该分组策略以及相应的指令进行相应的操作（如替换分组策略、迁移状态等）；S1负载均衡模型依赖的底层基础是状态迁移机制。

因此本章先介绍本文涉及的重要流计算系统概念，并结合常用的流计算系统，建立通用的系统模型；进而对分组策略问题进行数学描述，同时也为后文介绍S1选择模型提供基础技术支持；并对实验平台Apache Flink的状态迁移情况进行简单介绍，并阐述本文实现的状态迁移机制；最后介绍S1负载均衡模型的整体工作流程。

本章主要主要目的是问题建模和S1负载均衡模型的机制介绍，阅读本章只需要知道S1选择模型和S1划分模型会提供一个用于负载均衡的新分组策略即可，关于这两个模型的详细内容在下一章介绍。

## 分布式流计算系统模型概述

分布式流计算系统由算子和数据通道组成，算子从数据通道接收到其他算子发送的数据，在本地进行计算，得出的结果由数据通道发送给其他算子。算子的本地计算是一些列预定义的计算逻辑，例如聚合、过滤、链接等，也可以是用户自己定义的处理逻辑，例如单词计数。图2-1是“求拍卖行每个商品竞拍价格的最大值”的分布式流计算系统拓扑图，第一列算子功能是从数据源读取数据并随机发送给下游算子；第二列算子功能是对数据预处理，将数据中的拍卖品ID和竞拍价格提取出来，组成新的数据(拍卖品ID，竞拍价格)并发送给下游算子；第三列算子为每个收到的数据建立一条本地记录，用以保存该拍卖品ID有史以来的最大竞拍价格，这些本地记录称作算子的状态。对于有状态的算子，其上游算子的分组策略需要考虑下游算子存储状态的内存开销，以及更改分组策略前后的数据一致性。



**图2-1** 求拍卖行每个商品竞拍价格的最大值

流计算系统的拓扑图可以抽象成一个有向图，算子抽象为图中的点，数据流抽象为图中的边。有向图是否有环取决于流计算平台的具体实现，Apache Flink平台不允许环的出现，但Apache Storm可以定义环形拓扑，本文实现基于Apache Flink平台，所以后文不考虑环形拓扑的情况。

## 分组策略问题模型概述

表示上游算子和下游算子之间的一条由数据组成的数据流通道[38][39][40]，对于数据流中的任意数据，本文用表示这条数据的身份识别信息，即这条数据的键。数据通常是一个由多个域组成的元组，键通常是这个数据元组中的某个域，用以区分元组所属类别。图2-1中，拍卖品ID即为数据的键，竞拍价格等信息则是数据的值。对于某个特定的流计算系统，键的所以可能取值范围即取值空间定义为，由此本文可以得出。对于数据流中的每个可能的键，本文用表示这个键在数据流中出现的频率，其中为开始统计频率的时刻，为结束统计频率的时刻，后文用表示为流计算系统运行开始时刻，为当前时刻的情况下，键的出现频率。

本文定义分组函数来表示分组策略，分组函数将根据数据的键将数据映射到下游算子，下游算子数量为个，分别由整数表示。由此可以对键被分配到的下游算子定义分组策略的函数式描述形式。在弹性可伸缩流计算系统中，可以改变分组策略以及增加或减少下游算子的数量来实现负载均衡，即调整上述式子中的函数和常数来改变算子所在机器的可用资源量。调整后的分组策略表示为，通常流计算系统运行过程中可能需要多次调整，而只需要关注单次调整之前和调整之后的变换过程即可。

## 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[41][42][43]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。所以本文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K。

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。

## Flink现有机制介绍

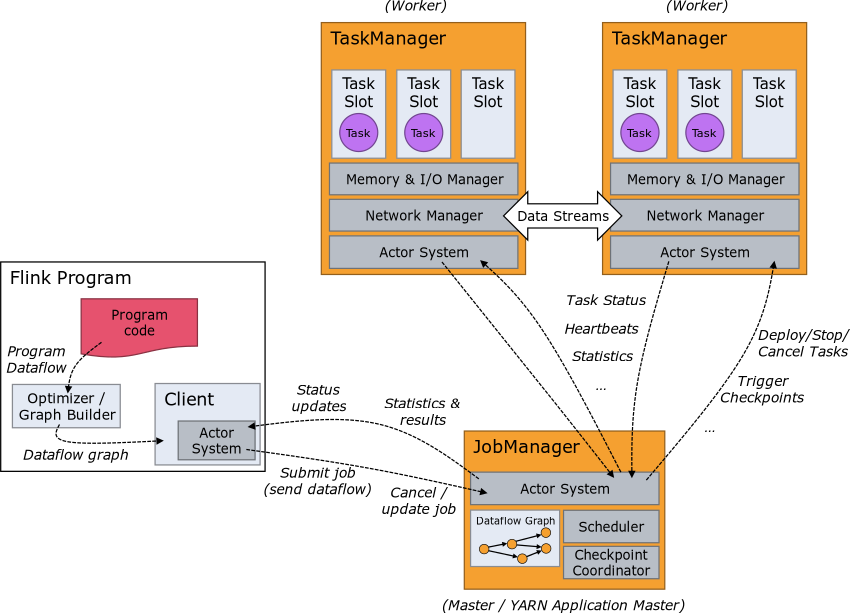
本节介绍Apache Flink中原生的状态迁移机制，以及本文受此机制启发实现的运行时状态迁移机制。

### Flink 的运行时架构

Flink的运行时架构如图2-2所示，主要由两个重要部件组成：Job Manager和Task Manager。其中Job Manager负责管理作业，类似于其它分布式计算系统的master角色；Task Manager负责具体的作业执行，类似于其他分布式计算系统的slave角色。

（1）Job Manager

Job Manager的主要作用时作业调度和检查点协调，可以协调整个分布式系统的运行。如图2-3所示，流计算任务的分配是由Job Manager中的调度器（Scheduler）模块将任务调度到Task Manager的任务槽（Task Slot）中。检查点的触发和故障恢复由检查点协调器（Checkpoint Coordinator）负责，该模块定期检查当前状态是否满足做检查点的要求。Flink中可以配置多个Job Manager，同一时间只有一个Job Manager处在正常运行状态，其他Job Manager在正常运行的Job Manager故障后发挥作用，保证系统高可用性。



**图2-2** Flink运行时架构

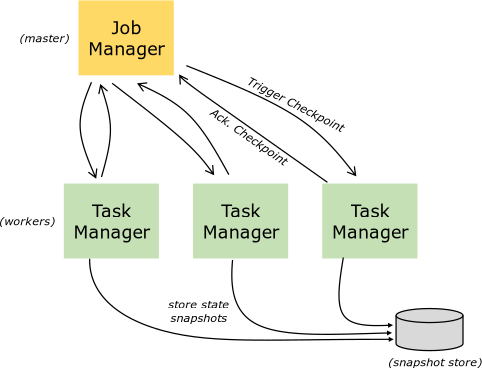
（2）Task Manager

Task Manager是一组执行作业的线程集合。实际上一个Task Manager就是一个Java VM进程，其中多个作业线程可以并行运行。Flink实际上没有使用Java VM本身的内存管理模型，而是自己开发了一种高效的块内存模型，该模型可以在将内存分为块状，这些块分为临时块和持久块两类，临时块可以用于记录流计算系统运行时产生的临时状态，而持久块则可以用于记录需要长期保存的状态。

任务分配时，Flink会按照数据局部性原理，将相关的作业尽量放置在同一个Task Manager，这样只需要通过线程切换即可通信，不需要额外的网络开销。局部性的数据在Flink中用上下文环境来表示，默认情况下，数据相关的作业会共享任务槽，使用相同的上下文环境。Flink中实现共享槽位的方法主要是Task Chaining机制，Flink会将可以合并的算子合并在一起放入相同的Task Manager，这样这些算子就可以共用上下文环境。

（3）Client

客户端的主要作用是提交作业。用户编辑的Flink作业程序经过拓扑优化等步骤生成数据流图，由客户端交给Job Manager。同时客户端也可以从Job Manager获取一些统计信息来了解作业运行状态。



**图2-3** Flink Task Manager和Job Manager之间的关系

### Apache Flink中的状态

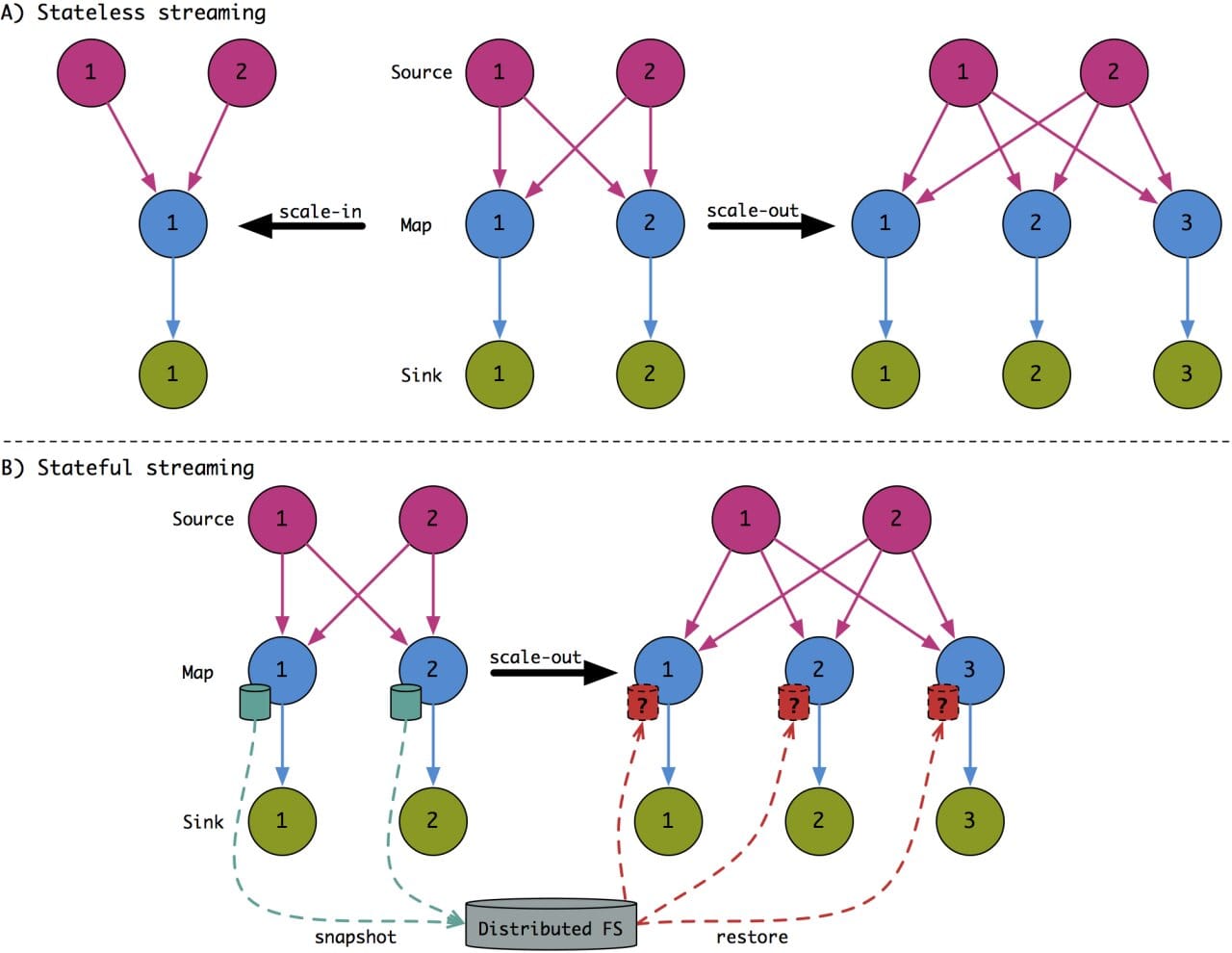
Apache Flink是一个大规模并行分布式系统，允许大规模进行有状态的流计算。为了实现弹性可伸缩，Flink作业在逻辑上分解为算子拓扑图，并且每个算子的执行在物理上分解为多个并行算子实例。从概念上讲，Flink中的每个并行算子实例都是一个独立的任务，可以在不共享任何网络的群集中在其自己的计算机上进行调度。

为了在此前提下实现高吞吐量和低延迟，必须最小化任务之间的网络通信。在Flink中，用于流处理的网络通信仅发生在算子拓扑图中的逻辑边缘（垂直），因此流数据可以从上游算子传输到下游算子,算子的并行实例之间（水平）没有通信。Flink利用数据局部性（本地性）的原则，避免了算子并行实例之间的网络通信，同时数据局部性原则也会影响状态的存储和访问方式。

为了实现数据的局部性，Flink中的所有状态数据始终绑定到运行相应并行算子实例的任务，并共同位于运行该任务的同一台计算机上。通过这种设计，任务的所有状态数据都是本地的，任务之间不需要网络通信即可进行状态访问。

### 弹性伸缩有状态的流计算作业

在无状态流中更改并行度（即更改为算子执行工作的并行子任务的数量）非常容易，如图2-4所示，仅需要启动或停止无状态算子的并行实例，并使其与上游和下游算子断开连接/断开连接。另一方面，改变由状态算子的并行度要涉及得多，因为还必须以一致有意义的方式重新分配先前的算子状态。在Flink的无共享架构中，所有状态对于运行拥有的并行算子实例的任务来说是本地的，并且在作业运行时并行算子实例之间没有通信。



**图2-4** Flink中有状态算子和无状态算子的重配置

Flink中已经存在一种机制，该机制允许在任务之间以一致的方式交换算子状态，并且保证exactly-once的语义，这就是Flink的检查点。检查点barrier会随着事件流从源流向接收器向下游流动，每当算子实例接收到屏障时，算子实例都会立即将其当前状态做快照并存储到分布式存储系统，例如HDFS。还原时，作业的新任务可以再次从分布式存储系统中获取状态数据。

可以在检查点上搭载状态作业的重新缩放，如图2-4所示。首先，触发检查点并将其发送到分布式存储系统。接下来以更改的并行度重新启动作业，并且可以从分布式存储访问所有先前状态的一致快照。虽然这解决了在计算机之间重新分配一致状态的问题，但仍然存在一个问题：如果再次将先前map\_1和map\_2的状态分配给新map\_1和map\_2，这会让map\_3处于空状态，根据工作状态和具体语义的不同，这种幼稚的方法可能导致从低效到错误的结果。

所以先前状态与新的并行算子实例之间没有明确的1：1关系的情况下，需要对存储的状态进行迁移。

### 弹性伸缩时重新分配状态

Flink中的状态可以分类两类，operator-state和keyed-state。其中，operator-state是算子运行时所必须保存的一些元信息，例如从kafka消息队列读取数据的算子会保存消费信息的offset；而keyed-state是处理数据过程中产生的并需要长期保存的 状态，例如拍卖行例子中求最高竞拍价格时，算子对每个键保存一个整型数据表示拍卖品的最高价格。Operator-state与算子处理数据的过程关系不大，本文主要考虑keyed-state，本文所述状态也均指keyed-state。

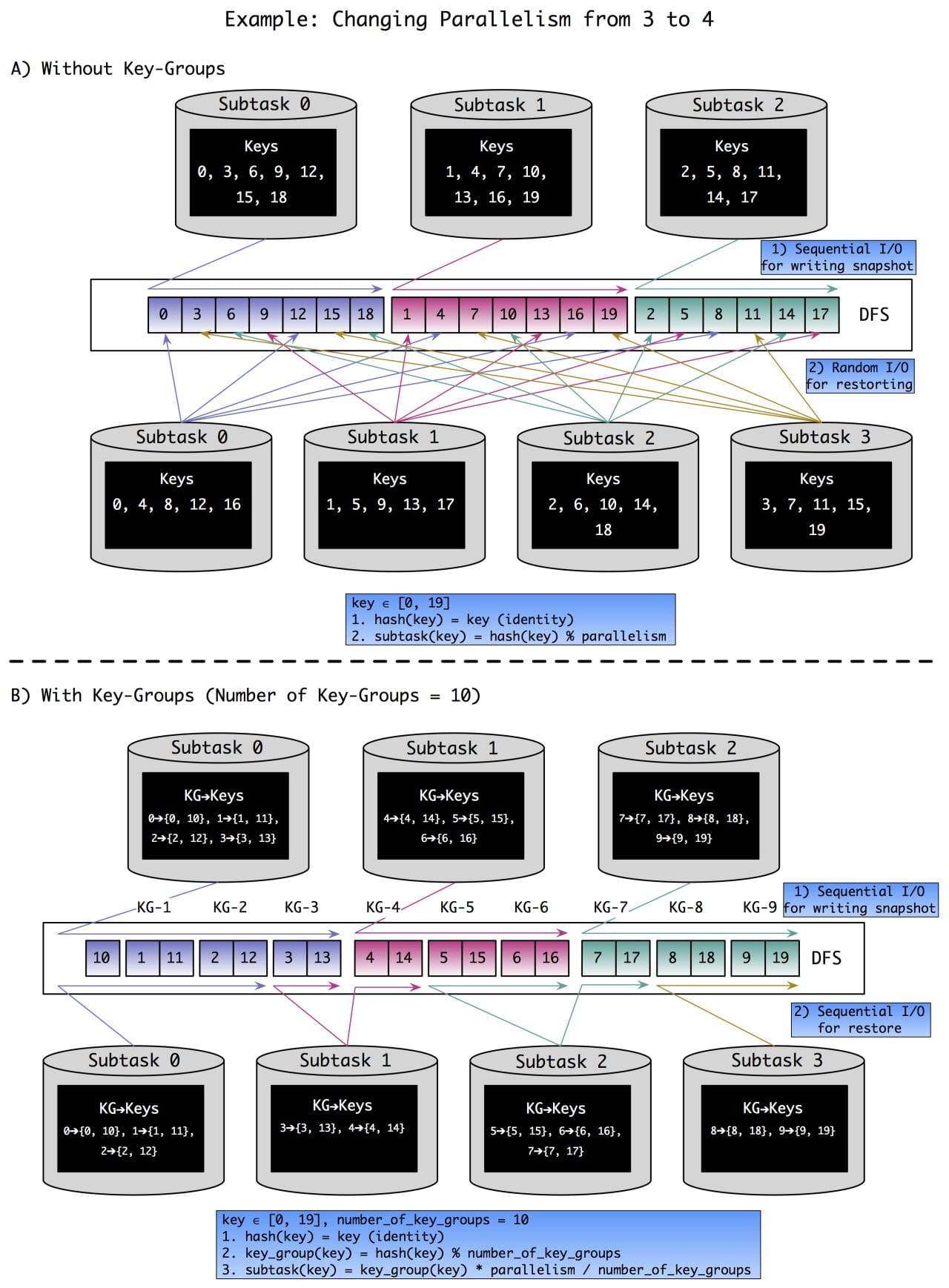
状态是来源于Flink中的keyBy()操作创建的数据流。 keyBy()操作指定如何从每个事件中提取键，对应上文模型中的；并且确保具有相同键的所有事件始终由相同的并行算子实例处理，对应上文中的。从键到算子的映射是通过键上的哈希分区确定地计算得到的。在进行重新缩放时，很容易就可以确定如何在并行算子实例之间正确分割和重新分配状态。重新缩放后，必须将每个键的状态分配给现在负责该键的算子实例，该实例由键流的哈希分区确定。

解决了状态映射的问题，现在继续考虑状态迁移的方法。

经典的方法如图2-5 A所示，是从所有子任务中的检查点读取所有先前的子任务状态，并过滤出每个子任务的匹配键。尽管此方法可以从顺序读取模式中受益，但每个子任务都可能读取很大一部分不相关的状态数据，并且分布式文件系统会收到大量并行读取请求。

另一种方法是建立一个索引，以跟踪检查点中每个键的状态位置。通过这种方法，所有子任务都可以有选择性地定位和读取匹配的键，可以避免读取无关的数据。但是这种方法有两个主要缺点。所有键的物化索引（即键到读取偏移的映射）可能会变得非常大。此外，这种方法还会引入大量的随机I/O，因为在分布式文件系统中为单个键寻找数据的性能非常差。

Flink的方法介于这两种极端之间。Flink通过引入键组作为状态分配的基本单位，必须在开始作业之前确定键组的数量，并且不能更改键组的数量。由于单个键组是状态分配的基本单位，因此这也意味着单个键组的数量是并行度的上限。简而言之，键组提供了一种弹性伸缩灵活性（通过设置并行度的上限）与索引和还原状态所涉及的最大开销之间进行权衡的方法。



**图2-5** Key-Groups提升状态迁移的效率

本文将键组分配给子任务作为其处理数据的范围。迁移状态时的读取不仅在每个键组中是连续的，而且可以跨越多个键组。另一个好处是键组到子任务分配的元数据非常小，只需要保存键组的边界即可。在图2-5 B中使用10个键组从并行度3缩放到4，引入键组可以大大改善朴素方法的访问模式。

### 异步检查点机制

在实时性方面，流计算平台比传统的批处理计算平台有很大优势，一般在秒级别甚至毫秒级别的延迟上就可以得到分析的结果。然而，流式计算也具有一个较大的缺点，即难以保证强一致性。 在分布式流计算系统中，根据数据流中的数据被算子有效处理的次数，可以将一致性级别划分为以下三类：至多一次、至少一次、恰好一次[47]至多一次表示一个数据将被一个算子“有效处理”0 次或者 1 次，即可能存在数据丢失。至少一次表示每个数据会被算子“有效处理”1 次或者多次，即可能存在数据重 复计算。恰好一次表示每个数据仅被相应算子“有效处理”一次。此处的“有效处理”指的是对该算子的状态仅产生一次影响，因此，讨论无状态算子的一致性级别没有太大意义。在海量数据处理的前提下，为了保障处理的吞吐量，流计算中难以使用类似于事务处理的强一致性方案。常见的流计算平台为了保证了数据不会丢失，都会提供较弱的一致性保证，即至少一次语义。为了避免故障的发生对一致性的影响，流平台必须借助外部稳定存储的数据重放的能力。Apache Storm[48]中通过Source算子对所有发射的数据进行备份，并利用Acker线程在下游算子中追踪数据的处理情况，从而保证了恰好一次的语义。Storm Trident通过微批处理的形式，以牺牲实时性为代价，提供了恰好一次的语义保证。然而，Flink基于异步Barrier快照算法，使其在容错的同时，可以有效保证恰好一次的语义。

Flink引入了Barrier的概念来实现一致的异步检查点。在Flink中，JobManager定期把Barrier加入到数据流中，作为提醒算子进行状态保存的特殊信号，它穿插在流数据之间，随着数据流在各级算子之间流动。在后续的介绍中可以看出， Barrier并不会中断整体数据流的处理，因此不会对系统性能造成较大影响。对于每一个Barrier，都具有唯一的编号，记为Barrier i。任意一个Barrier i 将源源不断的流数据一分为二，使得每个算子在处理Barrier i 时必然已经处理了Barrier i前的所有数据，并且还未处理Barrier i之后的数据。图 2-6以流计算拓扑中间某一层算子为例，描述了算子3在包含多个上游算子的情况下，处理Barrier n的操作流程。

如图2-6（a）所示，当算子3从上游的某个算子接收到Barrier消息Barrier n 后，算子3即进入检查点的准备工作——对齐。由图 2-6（b）可以看出，在Barrier对齐阶段，算子3暂停处理来自算子1的数据，并将其缓存，与此同时继续处理算子2发射过来的数据。当算子3接收到来自算子2的Barrier n消息时，算子3将自身的状态保存至可靠稳定的状态后台中，并将Barrier n继续向其下游算子传递。图2-6（d）展示了算子3在完成检查点操作后先处理此前缓存队列中的数据，恢复处理来自所有输入流的数据。由此可以看出，一个算子仅当接收到来自其上游所有算子的相同Barrier消息时，才会真正执行状态存储的操作。



**图2-6** Flink中异步检查点的Barrier机制

在Flink中，为了避免算子在存储状态快照的同时停止数据的处理，甚至可以让状态快照的存储异步地在后台进行。当算子接收到所有输入流的检查点Barrier之后，算子生成一个新的状态对象，并异步地将其写入状态后台。与此同时，算子可以继续处理输入的数据，进一步修改状态而不会对该过程造成影响。

## 运行时状态迁移

在第一章的国内外研究现状中本文介绍了状态迁移机制的三种类型，Apache Flink使用的是“结束-迁移-重启”这一机制，而本文为了在运行时进行状态迁移，则需要实现“暂停-迁移-继续”这一类型的迁移机制。不选择“备份拓扑”机制是因为备份拓扑机制本身需要大量冗余算子和冗余状态，这与本文  
优雅地实现负载均衡这一目标相矛盾。本文实现的运行时状态迁移机制主要解决了两个问题，一是运行时确保分布式一致性以及语义的情况下发生迁移，二是具体如何迁移。

为了保持分布式一致性以及exactly-once语义同时让迁移发生在系统运行时，本文需要在迁移状态时暂停正在迁移状态的算子，迁移完成后再继续处理数据，这样就不需要结束流计算系统再迁移。本文重写了算子处理逻辑，并为算子接口增加onBarrier()方法，在算子接收到检查点指令时，额外调用onBarrier()方法。同时本文根据算子种类的不同以及上下游的组织关系，实现了多种算子分别作为上下游时的onBarrier()方法。onBarrier()方法的具体实现是阻塞式的，这样就可以实现暂停算子的目的。本文只需要暂停发生状态迁移的算子（称为下游算子），其他算子仍然可以正常工作，局部暂停以对系统最小的影响实现负载均衡，轻量而高效。

具体地，本文实现了迁移协调器辅助迁移，迁移状态的算子需要与迁移协调器建立通信，协同完成状态迁移。算子依据新分组策略将不属于本地的状态发送到迁移协调器，并从迁移协调器获取其他算子发往本地的状态。一次性迁移和细粒度迁移的迁移机制实现稍有区别，由于一次性迁移需要在仅有的一次暂停机会中完成所有状态的迁移，所以需要先与迁移协调器建立通信连接的算子等待后续建立通信链接的算子，当所有算子都建立通信连接后，才可以开始迁移，也只有这样才可以保证迁移完成后，所有状态都在新分组策略指定的位置，即保证分布式一致性。细粒度迁移则可以不受这一约束的影响，当算子与迁移协调器建立通信连接后，只需要发送不属于本地的状态，并根据新分组策略查看是否有状态会发往本地，若这些状态已经在迁移服务器中准备完毕，则拉取状态完成本次细粒度迁移，若这些状态尚未准备完毕即不在迁移服务器中，则建立本地监听线程，等待这些状态准备完毕后，迁移协调器将状态推送至本地监听线程，由监听线程接收状态并与本地状态合并。

## S1负载均衡模型功能设计

在前文描述的基础上，本节简单介绍S1负载均衡模型。该模型基于异步检查点机制协同工作，实现细粒度负载均衡。需要S1负载均衡模型实现的功能如图2-7所示。

图2-7展示了负载均衡模型需要完成的工作（该模型应具备的功能）。首先负载均衡模型要与负责检测数据倾斜的源算子交互，在必要的时候获取高频数据相关信息。S1负载均衡模型需要获取下游算子（算子3和算子4）的负载不均衡程度相关信息，从而判断是否应当触发状态迁移。



**图2-7** S1负载均衡模型的功能

触发状态迁移后，S1负载均衡模型会把新分组策略和状态迁移的相关指令及数据绑定到检查点，加入源算子的数据流中，与正常的barrier一样在流计算系统中传播。上游算子barrier后将分组策略替换为barrier中携带的分组策略，新的分组策略可以保证将后续到达的数据均匀地发送到下游算子，只要数据倾斜情况不变。

下游算子的负载不均衡程度就不会因为数据分发不均而增加。下游算子接收到barrier后，与S1负载均衡模型建立链接，将今后不属于本地的状态迁出，并拉去其他并行实例发给本算子的状态。细粒度迁移中，各个下游算子进行该过程是相互独立的，不需要同步，只需要访问以此迁移协调器即可。而作为对比实验的一次性迁移实现时，该过程必须在访问迁移协调器时同步一次，保证所有算子都可以获得自身需要的状态。

与一次性迁移不同，细粒度迁移会多次调用上述过程，每次调用迁移一部分状态，等待下次带有指令的检查点到来后再进行后续迁移。

## 本章小结

本章主要为分布式流计算系统和分组问题建立数学模型；解释了模型用到的数据采样和迁移触发机制的工作原理；对开发平台Apache Flink的状态迁移机制做了详细介绍，并指出了其中不足；最后，基于异步检查点机制，简单阐述了本文的S1负载均衡模型需要实现的功能。

# S1负载均衡模型

在前文描述的基础上，本节介绍S1负载均衡模型。该模型需要辅助模块负责倾斜检测/迁移触发/指令发布等工作，迁移协调器来实现运行时状态迁移的功能，S1选择模型和S1划分模型计算出最佳的新分组策略。三部分基于异步检查点机制协同工作，实现细粒度负载均衡。S1负载均衡模型的结构图及工作流程如图3-1所示。



**图3-1** S1负载均衡模型结构图

（1）①检测数据倾斜；②检测下游算子不均衡程度；③发出高频数据分布统计指令；④指令绑定检查点加入源算子；

（2）①请求新分组策略；②计算收益与代价并生成新分组策略；③提供新分组策略；

（3）①发出负载均衡指令；②指令绑定检查点并加入源算子；③上游算子（算子1和算子2）更新分组策略；④下游算子（算子3和算子4）迁移状态；⑤收集本次迁移的延时信息；

辅助模块的迁移触发功能会在每次检查点到来时，从代表下游算子的算子3和算子4获取当前负载不均衡程度作为迁移作为是否触发迁移的依据，不均衡程度大于触发阈值时触发迁移。源算子负责检测数据倾斜，提取数据流中的高频键。在触发迁移后，辅助模块的倾斜检测功能会访问源算子获取当前时间段的高频键和频率，将高频键相关信息发送到S1选择模型/S1划分模型，同时生成指令要求下游算子统计自身的高频键分布并将指令加入源算子使指令在流计算系统中广播，下游算子接收到指令后，会将自身含有的高频键数量以及占用内存空间大小等信息发送到S1选择模型/S1划分模型。S1选择模型会统计到访的下游算子数量，当所有下游算子都访问过后，使用从下游算子获取的信息生成可以均衡负载的新分组策略，交付给辅助模块的指令发布部分。

此后，辅助模块的指令发布部分会将新分组策略和更新分组策略及迁移状态的指令加入后续检查点中，上游算子（算子1和算子2）接收到指令后，会用新的分组策略替换原有的分组策略，以此实现分则策略的更新；下游算子（算子3和算子4）在接收到指令后，会与迁移协调器建立数据通信，根据新的分组策略将今后不应由自身存储的状态迁出到迁移协调器并标记应当发往的下游算子序号，同时从迁移协调器拉取其他算子传给该算子的状态，以此实现状态迁移。本次状态迁移仅完成了适应新分组策略完成一次状态迁移后，迁移协调器会将这段时间统计的状态迁移用时信息发送给S1选择模型/S1划分模型，生成新的分组策略交付给辅助模块的指令发布部分。

由此开始下一次循环，直到S1选择模型/S1划分模型将所有状不再提供新分组策略，辅助模块则停止工作，至此是一套完整的细粒度状态迁移过程。

## S1选择模型与S1划分模型的协作

S1选择模型和S1划分模型是S1负载均衡模型的重要组成部分，二者协同生成新分组策略用于负载均衡。S1选择模型的主要作用是为高频状态选择一个合适的下游算子，因此命名为S1选择模型。S1选择模型会按照净收益大小对结果排序。S1划分模型的主要作用是决定一次细粒度迁移状态的数量和大小，其工作流程是对S1选择模型的结果进行划分，所以命名为S1划分模型。

针对数据倾斜的情况，本文使用S1选择模型与S1划分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，而本文需要频繁改变分组策略，所以在一致性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3-2，S1选择模型利用下游算子的负载分布信息，计算出负载均衡收益以及迁移代价，按照净收益大小排序得到状态队列，交由S1划分模型从中选择一部分作为本次细粒度迁移的状态，生成动态路由交给S1选择模型，S1选择模型将动态路由组装到新分组策略，交给辅助模块的指令发布部分。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，因此本文提出S1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。迁移协调器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3-2** S1选择模型和S1划分模型

下面先介绍模型中用到的负载计算方法，再分别介绍S1选择模型的算法和S1划分模型。

## 用户定义函数分类及负载的计算

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（target）是由这条数据的键（k）以及分组策略（p）共同决定的，表示为p(k)→target。键k在一段时间内的数据源中出现的频率表示为f(k)，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。

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。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，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[18]可以让hash环的查找复杂度降低到。

在流计算系统中，每个算子的处理逻辑可以形式化表示为，为算子的处理函数，表示该算子的输入数据，表示该算子处理后发送给下游的输出数据。算子的处理逻辑可以由系统本身提供的函数，也可以由用户定义。由用户定义的算子逻辑成为用户定义函数。系统本身提供的算子逻辑函数有聚合、过滤、合并等，用户也可以完全按照自己的想法实现算子处理逻辑。

为了科学地计算算子的负载，本文需要先为算子处理一条数据造成的负载建立模型，即单数据负载，进而计算算子累积处理数据的负载，即算子的负载。

### 单个状态的CPU/存储负载计算

算子处理数据的负载分为计算型负载和存储型负载，计算型负载消耗CPU资源，负载越大CPU消耗越大，该算子的延时也越大；存储型负责消耗内存/外存资源，存储负载越大内存/外存开销越大，检索存储信息的延迟也会越大。

#### 计算型负载

算子数据处理逻辑的时间复杂度是算子处理一条数据的计算型负载，表示为，通常情况下，算子对数据的处理是常数时间复杂度的，即，其中为任意一条输入数据。对于时间窗口算子，若算子接收到数据后需要对时间窗口内的所有其他数据进行枚举访问，那么这种算子的计算负载是。

#### 存储型负载

算子保存的状态造成的负载是算子处理一条数据的存储型负载，表示为，不论是系统本身提供的算子逻辑函数还是用户自己实现的算子处理逻辑，存储负载都可以分为两大类，**有状态用户定义函数**和**无状态用户定义函数**。

其中，无状态用户定义函数是指不需要在算子本地存储信息的处理逻辑，例如图2-1第二列的算子，其功能是提取读入数据中有用的信息，组和成新的数据并发送给第三列算子，本身并不需要保存任何关于数据的信息。计算方式如下：

有状态用户定义函数可以分为两大类，一类是**无状态累积的用户定义函数**，一类是**有状态累积的用户定义函数**。其中，无状态累积的用户定义函数会在本地保存状态，状态的大小（状态占用的内存或外存）只取决于输入数据中不相同的键的个数，与重复出现的键的数量无关。例如图2-1中的第三列算子，其功能是保存所收到的拍卖品ID对应的竞拍价格最大值，对于某个特定的键（拍卖品ID），当有更高价格出现时，本文只需要用更高的价格替换原先最高价格即可，所以重复出现的键只会改变状态的数值，不会增加状态占用的内存空间，算子需要为每一个不相同的键开辟一块空间存储状态，即算子的状态大小则是由不相同的键的数量决定。

有状态累积的用户定义函数不仅会在本地保存状态，还会在收到重复出现的键时申请新的内存/外存空间保存新数据的必要信息，例如，要计算每个拍卖品的历史价格中位数，那就需要保存每个拍卖品的所有历史价格作为一个列表，在收到新的竞拍数据后，更新相应的拍卖品价格列表，以获取新的中位数。有状态累积的用户定义函数的状态大小取决于收到拥有相同键的数据数量，以及相同键的数据聚合算法。

其中，是关于x的多项式，在上述求中位数例子中，。

### 算子所有状态的CPU/存储负载计算

前文介绍了如何计算单个数据对算子造成的负载，本节研究一段时间内算子的累积负载。对于下游算子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。

#### 存储型负载

存储型负载是算子在本地存储的状态，取决于用户定义函数的存储复杂度，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，则；算子若使用word count这类计数函数，则。对于某个特定的算子，表示这个算子的存储型负载，有：

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，表示这个算子的计算型负载，有：

其中为分组函数对的分组结果。

## S1选择模型

### 收益计算

在S1负载均衡模型中，S1选择器需要对所有高频键分别尝试，分配到每个下游算子，对于每个尝试的下游算子，当前已分配的键可以构成一个临时动态路由（虽然最后不会使用这个动态路由）。所以，对于每个键，都可以得到一系列临时动态路由，对这些临时动态路由分别计算收益和代价，选择净收益最高的临时动态路由中对这个键的分配位置作为这个键的最终分配位置。

#### 可容忍负载比例

本文定义下游算子的负载比例为

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

假设可以容忍的负载最大比例为，即

所以可以得到算子负载与平均负载的差值的上限：

其中，代表下游算子的负载平均值。当只有一个算子负载较大，其他算子负载都特别小且相等的时候，假设小负载算子的负载是，则大负载算子的负载是，大负载算子的负载与平均值的差值可以达到上限，即，又有，可得：

因此对于可以容忍的负载最大比例，用表示可以容忍的下游算子负载差值上限。

#### 负载不均衡代价及均衡收益（Balance profit）

定义均衡开销，即向下游算子发送数据的不均衡程度。通过前文的定义，可以计算出每个算子的负载，由此将负载超出容忍限度的比例定义为负载不均衡代价：

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。为可容忍的负载差值上限，

总负载不均衡代价定义如下：

由于S1负载均衡模型是中，有新旧两个分组策略即两个动态路由，以及S1选择模型中有一系列临时动态路由，所以本文需要比较动态路由的优劣，即使用新动态路由对负载不均衡程度的缓解定义为使用该动态路由/新分组策略的收益：

其中表示从旧动态路由/分组策略转换为新动态路由/分组策略后，在负载均衡方面的收益。

#### 迁移代价（Migration cost）

迁移代价表示如下，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额外迁移比例表示为，意为额外迁移的状态在总状态中所占的比例。

理想情况下，需要迁移的高频状态加入动态路由后，一直维持在动态路由中，这样一来，这些状态只需要迁移一次，即动态路由替换的最少迁移量，表示为，由在新/旧动态路由中至少出现一次的状态组成。现实情况下，这些状态可能不会一直持续高频，或者新/旧动态路由为这些状态分配的位置有变化，那么就需要额外再对这些状态进行迁移，即这些状态造成了额外的迁移代价，表示为，由前文所述的两部分组成。因此，评价动态路由优劣时，需要为满足动态路由要求而额外迁移的状态所占比例，作为迁移代价。

### S1选择算法

前文针对新/旧动态路由计算了负载不均衡代价和迁移代价，定义采用新动态路由后的净收益为，关于净收益的计算，本文提出三种计算方式：

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：从未分配的高频键集合中，选择收益最大的高频键及其分配目的地，加入新动态路由，

| **Algorithm 1:** ) |
| --- |
| Param: // 原先的动态路由 |
| Param: // 待分配的高频键集合 |
| Let |
| while has elements do // 还有剩余高频键未分配 |
| // 找到收益最大的状态 |
|  |
|  |
| return |

算法2是算法1中用来查找当前状况下收益最大的键机器分配位置的算法。该算法的主要思想是：枚举剩余待分配的高频键，尝试将这一键分配给每个下游算子并计算收益，选取收益最大的算子作为这个键的最优分配算子；在所有键中找到按照最优分配算子分配收益最大的键，作为一次选择的结果。

算法1的结果是高频状态以收益为排序关键字组成的优先队列，包含的数据元素的结构与分组策略中包含的元素结构相同。因此算法1中将队列作为参数传入算法2表示新动态路由的参数位置。

至此，S1选择模型的选择算法完成了所有工作，但是并没有给出新动态路由，而是给出了完整的迁移队列，S1选择模型将队列交给S1划分模型，由S1划分模型处理队列并生成新动态路由，之后S1选择模型将新动态路由交给辅助模块在流计算系统中广播。

| **Algorithm 2:** |
| --- |
| Param: // 原先的动态路由 |
| Param: // 正在生成的新动态路由 |
| Param: // 待分配的键 |
| Let |
| for each do |
|  |
|  |
|  |
| 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净收益更大 |
| //更新目标和净收益 |
| If then |
|  |
|  |
|  |
| return |

## S1划分模型

S1划分模型将队列划分成多个子队列，依次返回由单个子队列组成的新动态路由。将一次性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。

### 不划分策略

不划分策略是S1划分模型的退化版本，适用于一次性迁移的要求。使用此策略意味着放弃细粒度迁移的优势，模型将队列中的全部高频键都加入新动态路由，返回给S1选择模型。这样整个S1负载均衡模型就退化成了只生成一次新分组策略并进行一次性迁移的朴素负载均衡模型。这一划分方式可以帮助本文与其他分组策略进行对比，或者作为实验的对照组用来对比细粒度迁移与一次性迁移（即划分策略与不划分策略）的优劣。

### 最大用时划分策略

最大用时划分策略是用于S1划分模型的正常版本。最大用时划分策略与Java VM的g1垃圾收集器相似，可以由用户指定最大迁移用时作为最大用时。划分策略根据以往迁移用时的经验，为每次细粒度迁移选择一些状态，保证每次细粒度迁移在最大迁移用时内完成。

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；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，认为其期望为。第次细粒度迁移的用时定义为，其中是迁移状态需要的时间，其数值与这次迁移的状态大小相关，最大用时划分策略在为每次细粒度迁移划分状态时，需要估计迁移这些状态花费的时间，即。

本文认为迁移状态花费的时间与状态的大小成正相关，因此每次进行细粒度迁移都记录下用时和迁移的状态大小，在划分策略工作时，根据这些以往的状态大小和用时记录，使用最小二乘法计算出迁移花费时间和迁移状态大小之间的函数关系，即。根据这一函数关系在预估迁移用时小于最大迁移用时的前提下，从队列的剩余备选状态中尽量多地选择状态生成本次细粒度迁移的新动态路由。

由于使用的分组策略是不拆分键的分组策略，因此本文在选取状态时不能分割队列中的单个状态。所以采用类似于背包问题的动态规划方法从队列中选择尽量多的状态，过程如算法3所示。

最大用时划分策略使用算法3可以得出，在用户给定最大迁移用时的情况下，如何选择状态可以让这次细粒度迁移的收益最大。

| **Algorithm 3:** *)* |
| --- |
| Param: // 待分配的状态队列 |
| Param: // 最大迁移用时 |
| Let |
| Let |
| for do //搜索队列中每个状态 |
| for do |
|  |
| if then |
|  |
|  |
| continue |
|  |
|  |
| if then |
|  |
|  |
| else |
|  |
| Let |
| return |

### 自平衡划分策略

自平衡划分策略是对最大用时划分策略的优化，更适于S1划分模型。大多数情况下用户无法为迁移最大用时设定合适的值，因为使用流计算系统的用户大多数是把流计算系统当作服务器中的一项服务，设定最大用时可能会给用户造成困扰。另一方面，如果最大用时设置不当，还有可能导致一些较大状态一直无法迁移。

因此本文又提出了自平衡划分策略，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，自动平衡每次细粒度迁移的状态量大小。由于网络状况随时变化，本文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

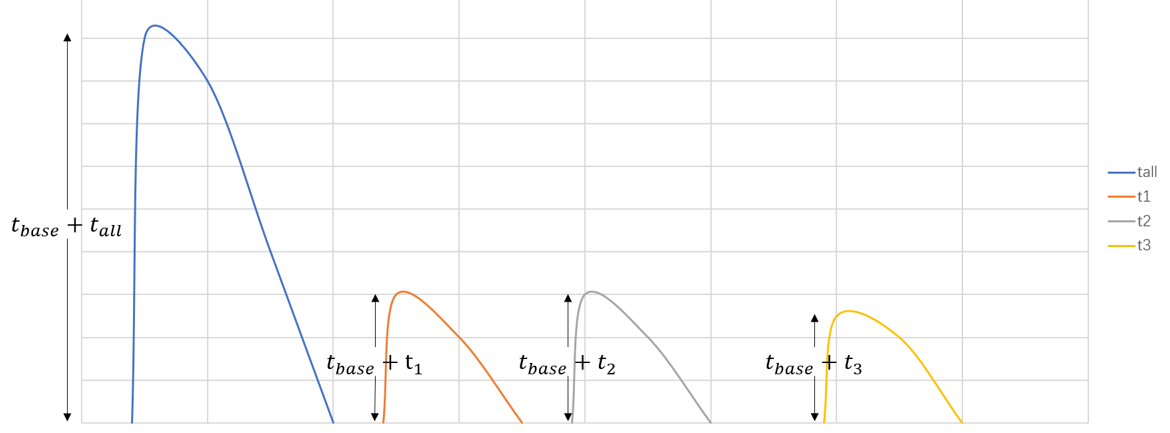
假设一次细粒度迁移不做任何状态的迁移，只进行其他必要的步骤，如对齐操作，则称这一次迁移为空迁移。一次空迁移的时间是固定的，随环境因素波动，定义为一次细粒度迁移的固有延时，表示为，其中，是检查点游标的对齐时间，取决于流平台的任务类型、拓扑图、输入流等，通常是固定值；是系统环境因素带来的延时，与网络状况等因素有关，而且会随时变动，认为其期望为。

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

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

其中，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3-3** 迁移划分的原理

如图3-3，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。实际运行时无法事先知道 ，所以本文采用试错的方式，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。表达式是关于参数的凸函数，对于整数必然存在最小值。

确定划分次数后就可以求出每次细粒度迁移的状态大小，其中是剩余带迁移状态大小的总和。然后本文使用算法4从队列中选出不超过状态组作为新动态路由。

| **Algorithm 4:** *)* |
| --- |
| Param: // 待分配的状态队列 |
| Param: // 最大迁移状态大小 |
| Let |
| Let |
| for do |
| for do |
| if then |
|  |
|  |
| continue |
|  |
|  |
| if then |
|  |
|  |
| else |
|  |
| Let |
| return |

自平衡划分策略中，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（不是确定次数，划分次数本身由倾斜程度，要迁移的数据量，λ等因素共同决定。）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。事实上，最合适的取值影响因素与最大用时划分策略中的最大用时选取因素相似，例如机群的网络状况，输入数据的倾斜幅度等。但是通常取值范围是1~10的正整数，且呈现一定规律，比最大迁移用时更容易选取。第五章实验部分详细探讨了值的选取问题。

## 本章小结

本章主要研究S1选择模型和S1划分模型。先阐述了两个模型的协同工作原理；然后常见流计算平台的用户定义函数进行数学建模，方便计算算子负载；之后介绍了S1选择模型的收益计算方式和选择算法；最后介绍了S1划分模型的两种划分策略。

# 模型的实现

本文的负载均衡模型需要S1选择模块，迁移协调器，S1划分模块，检测数据倾斜以及触发迁移的辅助模块，对各模块分发指令的辅助模块。本文在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

总体流程图如图3-1所示，辅助模块的迁移触发功能会在每次检查点到来时，接受来自源算子的socket请求，根据上次检查点统计的负载不均衡程度判断本次是否应该触发迁移，不均衡程度大于触发阈值时触发迁移，并生成指令要求下游算子统计自身的高频键分布，将指令加入源算子使指令在流计算系统中广播。源算子负责检测数据倾斜，提取数据流中的高频键。在触发迁移后，源算子接收到指令后将高频键相关信息发送到辅助模块，辅助模块的倾斜检测功能会接受来自源算子的socket请求，获取当前时间段的高频键和频率，由辅助模块将高频键数据转发给S1划分模块。

下游算子接收到指令后，会将自身含有的高频键数量以及占用内存空间大小等信息通过socket请求发送到S1选择模型。S1选择模型会统计到访的下游算子数量，当所有下游算子都访问过后，使用从下游算子获取的信息，依据算法1和算法2从候选高频数据中选择净收益较大的高频数据，并将净收益作为优先级排序生成候选状态队列。此时新建S1划分模型实例，将状态队列交给S1划分模型，S1划分模型根据以往经验和状态优先级，选取第一批迁移的状态，生成动态路由并交付给S1选择模型，S1选择模型进一步将动态路由组装到新分组策略交付给辅助模块的指令发布部分。

此后，辅助模块的指令发布部分会将新分组策略和更新分组策略及迁移状态的指令加入后续检查点中，上游算子接收到指令后开始阻塞，用新的分组策略替换原有的分组策略，以此实现分则策略的更新，更换完成后解除阻塞；下游算子（算子3和算子4）在接收到指令后开始阻塞，与迁移协调器建立socket通信，根据新的分组策略将今后不应由自身存储的状态迁出到迁移协调器并标记应当发往的下游算子序号，同时从迁移协调器拉取其他算子传给该算子的状态，以此实现状态迁移。完成一次状态迁移后，迁移协调器会将这段时间统计的状态迁移用时信息发送给S1划分模块，S1划分模块用最大用时划分策略或自平衡划分策略计算剩余状态应当划分的次数，并生成新的动态路由交付给分组策略自动化生成模块。S1选择模型进一步将动态路由组装到新分组策略交付给辅助模块的指令发布部分。由此开始下一次循环，直到S1划分模型将所有状态划分完成，不再提动态路由，S1选择模型也会对辅助模块表现为不在提供新分组策略，辅助模块则停止工作，至此是一套完整的细粒度状态迁移过程。

## S1选择模块

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，由动态路由和静态路由组成；并生成新的S1划分模块，将新分组策略交付于S1划分模块。

分组策略的声明如下：

**public** **class** **MyPF**<K> **implements** Serializable, Partitioner<K> {

**private** MyConsistentHash<K> Hb;

**private** HashMap<K, Integer> Ha;

}

分组策略的对一个数据的分组过程如下：

**public** int partition(K key, int n) {

int tar=ha(key);

**if** (tar!=-1) **return** tar;

**return** hb.hash(key);

}

本节实现S1选择模块时，为S1选择模块增加了三个状态，使其像状态机一样工作。Idle状态表示目前没有迁移发生，系统准备就绪；metric表示需要统计下游算子状态分布情况，该状态由idle触发迁移后到达；migrating表示正在迁移，是metric的下一个状态，在metric状态的下一个检查点到来时，由辅助模块设置该状态：

**private** int state=0; *// 0:idle, 1:metric, 2:migrating*

提供新分组策略的接口设计成与迭代器相似的接口，辅助模块通过hasNext和next方法获得新分组策略。

S1选择模块的主要部分是运行S1选择算法，将状态按照净收益排序生成状态队列，该算法调用了三个函数来计算均衡收益、迁移代价和净收益：

**public** void selectAlgo();

**private** float balancePenalty(ArrayList<Float> operatorLoad, K d, int l, int cnt);

**private** float migrationPenalty(float m, float f, int l, int h, float mCeil);

**private** float computeNetProfit(float a, float r);

在下游算子提供分布数据时，S1选择模块会统计到访的下游算子数量：

**synchronized** boolean addMetric(int index, List<K> arr) {

metric.get(index).addAll(arr);

metricCnt++;

**return** metricCnt == parallelism;

}

到访算子数量达到并行度后由辅助模块调用selectAlgo来生成状态队列。

## S1划分模块

S1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S1选择模块提供动态路由，暴露hasNext和next接口，S1选择模块调用该接口获取下一个动态路由，从而构建新分组策略，并发送给分发指令模块。下面的代码是S1划分模块的接口定义，三种划分策略都是实现了这个接口。

**public** **interface** **HyperRouteProvider**<K> {

void addKey(K key, int oriPos, int newPos, float frequency);

void prepare(HashMap<K, Integer> finalHyperRoute);

boolean hasNextHyperRoute();

HashMap<K, Integer> nextHyperRoute();

}

使用不划分策略时，本模块会直接返回一个完整的动态路由，这使得迁移过程和一次性迁移完全一样。

使用最大用时划分策略时，本模块根据以往用时的经验数据计算各个状态迁移用时，从中选择小于最大用时且收益最大的状态加入动态路由。

使用自平衡划分策略时，本模块将总状态大小除以初始迁移因子作为第一次尝试迁移的状态大小参考值；同时根据以往经验计算这次迁移最合适的状态大小；若二者相差过大，自平衡划分策略会使用用户指定的初始迁移因子作为第一次尝试的大小，否则会使用经验计算的迁移量作为第一次迁移的状态大小。此后每次迁移都会根据之前的经验重新计算合适的划分次数。

## 迁移协调器

迁移协调器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。

迁移协调器实现时采用了阻塞IO（BIO）的设计模式，如图4-2所示。因为下游算子数量不多，通常流计算平台的并行度为10~20，这种规模的socket完全可以由BIO+线程池处理。NIO等异步IO会增加编码复杂度，而且不适合全程长时间传输数据。

迁移协调器结构如图4-2所示。迁移协调器为每个下游算子建立一个阻塞队列。每个socket都是用两个线程分别负责读写，负责读socket的线程接收到状态后把状态发送到目的地线程负责的阻塞队列，负责写socket的线程在下游算子对应的阻塞队列有数据时，从中获取数据并写入自身负责的socket。



**图4-2** 迁移协调器的结构图

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。

## 下游算子行为

下游算子接收到barrier后调用downStreamOnBarrier方法：

**private** void downStreamOnBarrier(Barrier barrier)

如果检查点中带有向S1选择模块提供状态分布信息的指令，则与S1先择模块建立链接，发送分布信息；若带有迁移指令，则与迁移协调器建立链接，迁出迁入状态，如图4-3所示。



**图4-3** 下游算子行为

合并状态是因为拉取迁移协调器的状态后，下游算子可能早已开始运行，已经处理了一些高频数据，产生了状态，所以需要将拉取的状态与本地状态合并，需要用户提供合并函数，合并函数需要实现Combiner接口，Combiner声明了处理一个数据对本地状态的更新方法addOne和合并两个已有状态的方法addAll：

**public** **interface** **Combiner**<T> {

T addOne(T t1, T t2);

T addAll(T t1, T t2);

}

## 基于异步检查点传播指令

本文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、S1选择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器的交互信息等等。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。因此本文利用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。

barrier上绑定的信息中，pf是分组策略，cmd是命令，ha存储的是动态路由序列化后的数据，代码如下：

**public** MyPF<K> pf;

**public** String cmd;

**public** Byte[] ha;

本文实现了多种命令，定义在一个公用的文件中，这些命令的作用分别是要求下游算子发送状态分布信息、要求下游算子进行细粒度迁移、要求下游算子进行一次性迁移、要求上游算子更新分组策略、要求上游算子发送统计信息、要求源算子发送高频键统计信息：

**public** **static** String downStreamMetric = "downStreamMetric";

**public** **static** String downStreamSplitMigration = "downStreamSplitMigration";

**public** **static** String downStreamOnceMigration = "downStreamOnceMigration";

**public** **static** String upStreamFetch = "upStreamFetch";

**public** **static** String upStreamMetric = "upStreamMetric";

**public** **static** String sourceHotKey = "sourceHotKey";

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。本文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。

本文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。

## 本章小结

本章先阐述了S1负载均衡器的总体工作流程；然后分模块介绍各模块的实现原理和工作过程中的重要细节以及重要的实现细节；最后介绍了异步检查点传播指令的实现原理和实现细节。

# 实验及模型评价

本文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，分别对比了细粒度迁移和一次性迁移的延时、观察了细粒度迁移和一次性迁移对不同并行度的适应性。

## 细粒度迁移与一次性迁移的延时对比

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，正确性的验证考虑两种常见的情况：

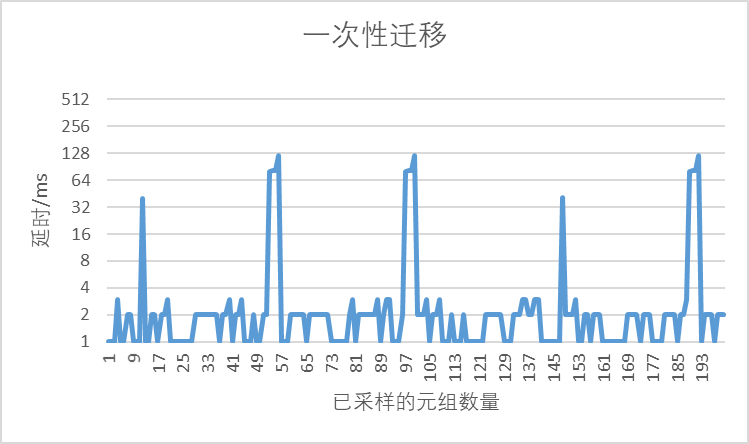
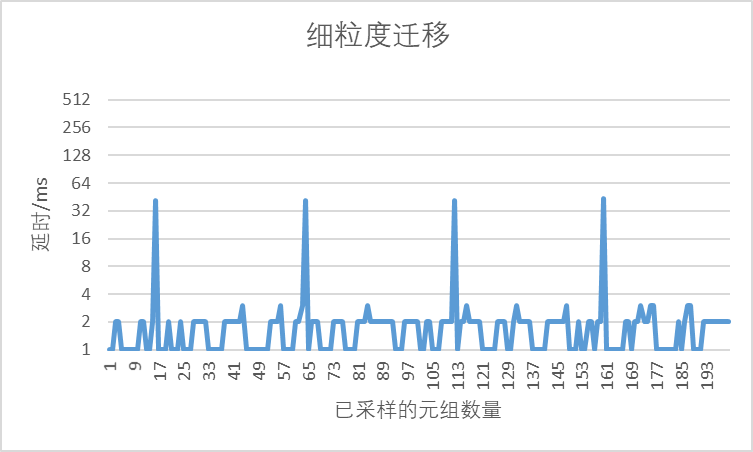
1）使用无状态累积的用户定义函数，算子在本地存储的状态大小只取决于数据取值范围，与频率无关，因此高频数据的出现只对下游算子造成计算型负载不均衡，对状态迁移却是毫无影响，即高频与低频用在迁移上的时间是一样的。对应的单数据负载表达式为，。

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。对应的单数据负载表达式为，。

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。

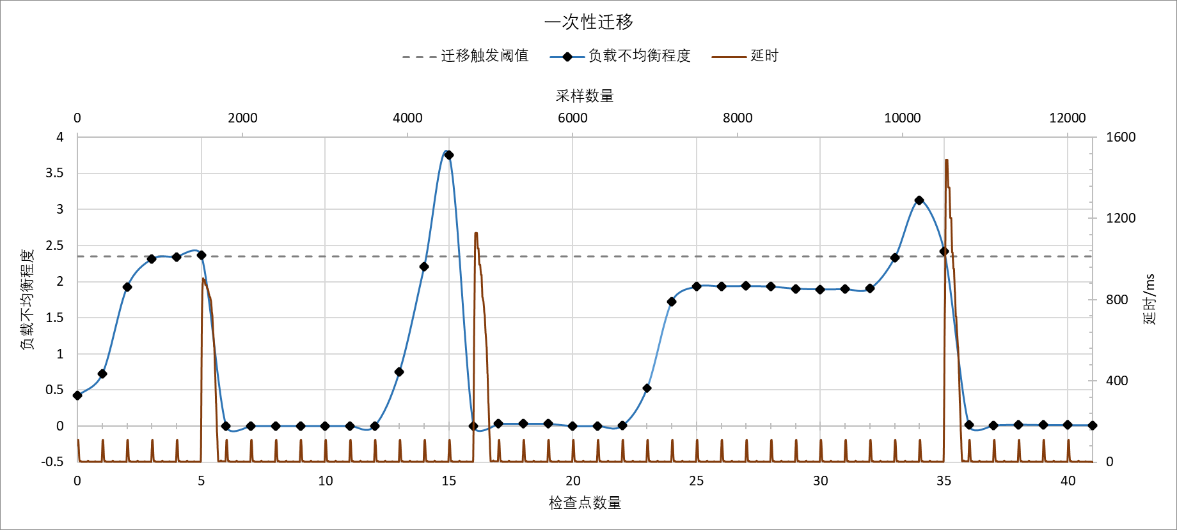
**图5-1** 使用无状态累积的用户定义函数进行延时对比

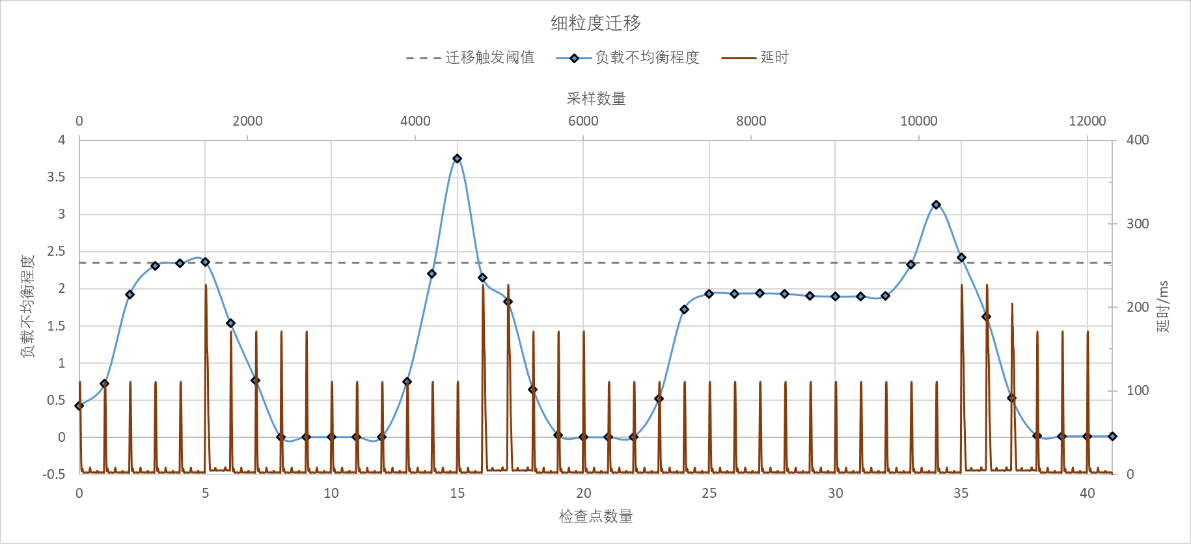
图5-1为流平台使用无状态累积的用户定义函数，在一次性迁移和细粒度迁移两种工作模式下的延时表现。

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，因此细粒度迁移和一次性迁移相比优势并不明显。

### 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5-2** 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5-2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。

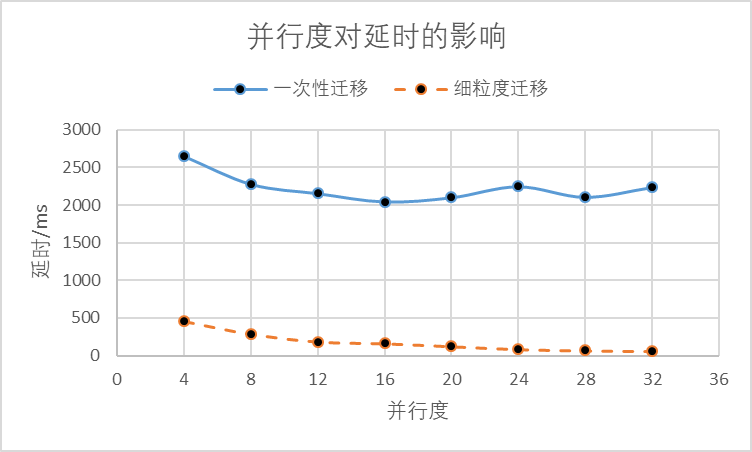
此拍卖行的买家竞拍数据随时间变化较为剧烈，在短时间内会形成多次倾斜，图5-2中负载不均衡程度的三个高峰表示出现了三次高频的卖家数据，导致下游负载不均衡。而本系统可以根据设定的迁移触发阈值在运行时迁移下游算子的状态，及时的重分配可以避免过大的不均衡情况，从而让下游算子运行在相对健康的情况下。图中三次高峰达到阈值2.35后，系统触发了细粒度迁移，逐渐将负载的不均衡程度减少到0.5以下的可接受范围内。一次性迁移直接将负载不均衡程度降到最低；细粒度迁移分两次将负载不均衡程度降低到迁移触发阈值以下，然后继续迁移进一步降低不均衡程度。

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5-2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。

## 细粒度迁移与一次性迁移对不同并行度的适应性

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5-3** 并行度对延时的影响

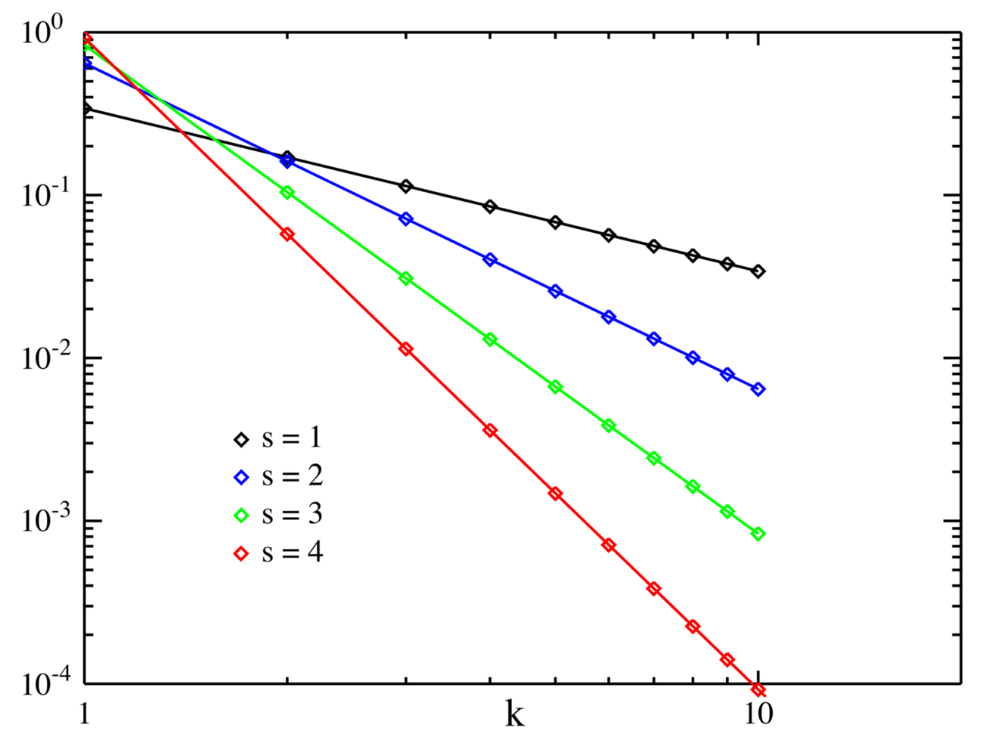
增大并行度可以减少每个下游算子存储的状态大小，减少状态传输用时。但是一次性迁移需要下游算子同时连接迁移模块（迁移服务器）同步传输数据，因此增加并行度，迁移模块需要同步的下游算子也随之增加。各算子进度及运行状态的不同，导致访问迁移模块的时刻也略有差异，并行度的增加导致等待同步的时间增加，进而增加数据处理的延时，所以图5-3中增加并行度不能有效减少一次性迁移造成的延时峰值。

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。

## S1选择模型与其他分组策略生成算法的对比

为了充分挖掘S1选择模型与其他分组策略生成模型之间的差别，本文使用满足Zipfian分布的生成数据对这些模型进行测试，观察它们在不同倾斜程度不同用户定义函数情况下的负载均衡表现。

Zipf法则是统计数学归纳出的经验法则，常用来模拟许多物理学或社会学的数据，按照这一法则生成的数据称为Zipfian分布数据。



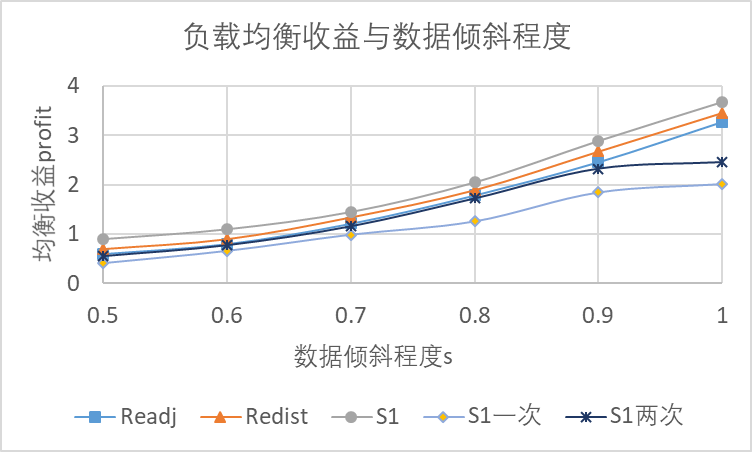
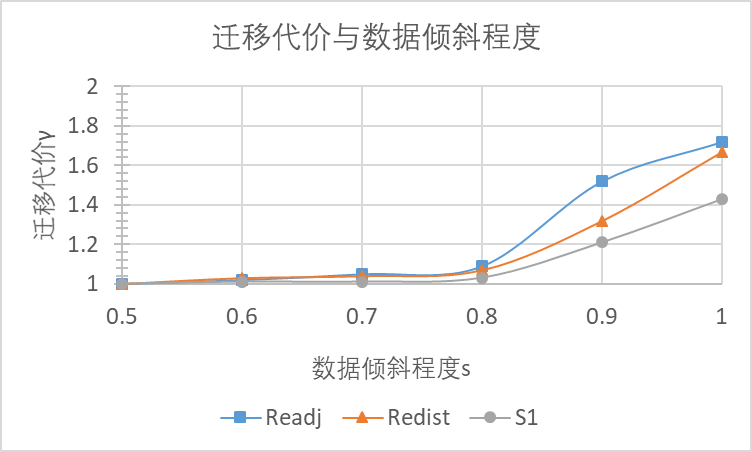
**图5-4** Zipfian分布

如图5-4所示，Zipf分布是一种离散数据分布，图中的线条只是表示数据分布概率走向，不代表数据是连续的。图中共有10个数据表示在x轴，y轴则表示这些数据出现的概率的对数。可见这些数据出现概率差别非常大，x轴排第一的数据（）比第二高两倍有余。Zipfian概率分布公式如下：

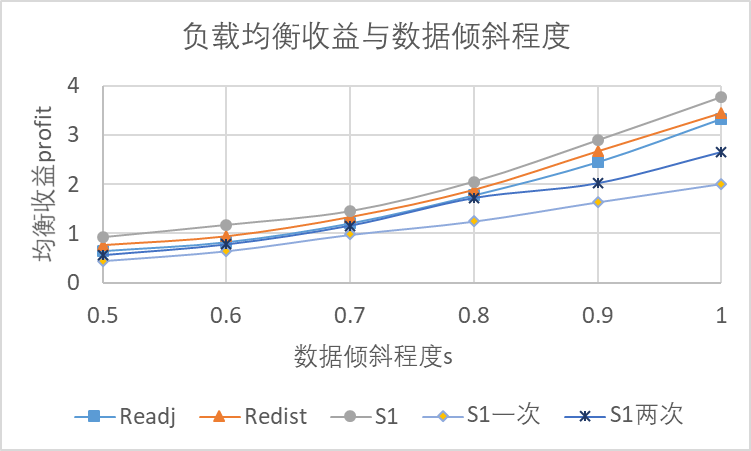
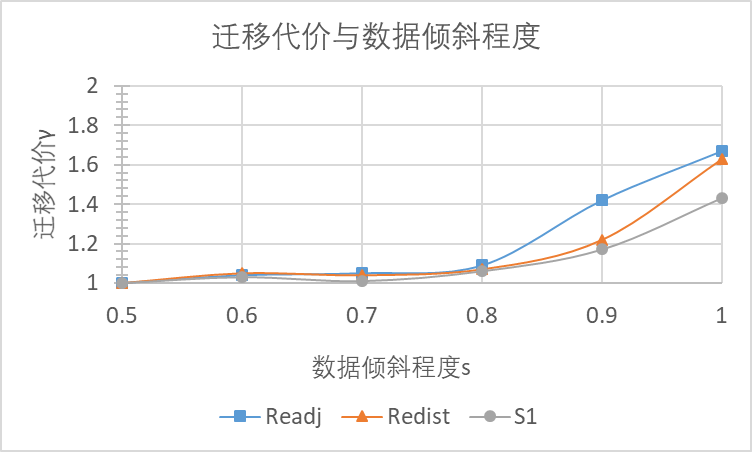
其中，是排名第的数据，是反应倾斜程度的参数，是参与统计的不同元素数量。通常设置来模拟常见的流计算系统数据倾斜情况。

用Zipf分布生成数据来测试S1选择模型和其他分组策略生成模型非常合适，这种分布天然提供了数据倾斜，同时这种分布来源于现实数据的统计学经验总结，所以具有一定真实性。

本文将S1选择模型与Readj和Redist分组策略生成模型的迁移代价和均衡收益做对比，得出图5-5结果。



1）用户定义函数类型为，



2）用户定义函数类型为，

**图5-5** 迁移代价和均衡收益受数据倾斜程度的影响

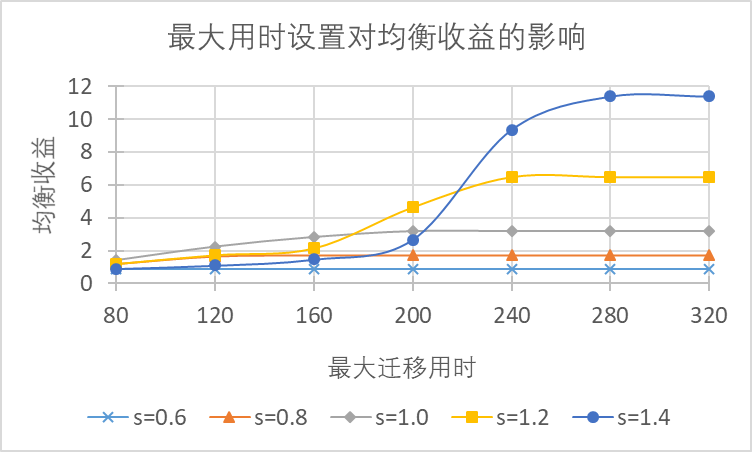
对于单数据计算型负载为常数级，存储型负载为线性的用户定义函数，图5-5 1）展示了这种负载的迁移代价和均衡收益在不同数据倾斜程度下与其他分组策略生成模型进行了对比，可以从图中观察到，S1选择模型相比其他分组策略生成模型（Readj、Redist）更有优势，在倾斜程度较小时，三者迁移代价均比较低；随着倾斜程度加大，Readj和Redist的迁移代价上涨较快，而S1选择模型的迁移代价涨幅较小。

S1选择模型在负载均衡收益方面优势更加明显。图中S1负载均衡模型的均衡收益始终大于Readj和Redist的均衡收益，更为重要的是，图中展示了一次S1细粒度迁移和两次S1细粒度迁移后，负载均衡收益与Readj和Redist以及S1本身的差别情况，可以发现，一次S1细粒度迁移的负载均衡收益涨幅是最大的，两次S1细粒度迁移的负载均衡收益即可逼近Readj模型的负载均衡收益，这得益于本文提出的S1选择模型和S1划分模型，这两个模型协同工作，选择收益最大且迁移代价最小的状态进行迁移，所以进行两次S1细粒度迁移的结果就已经非常优秀了。

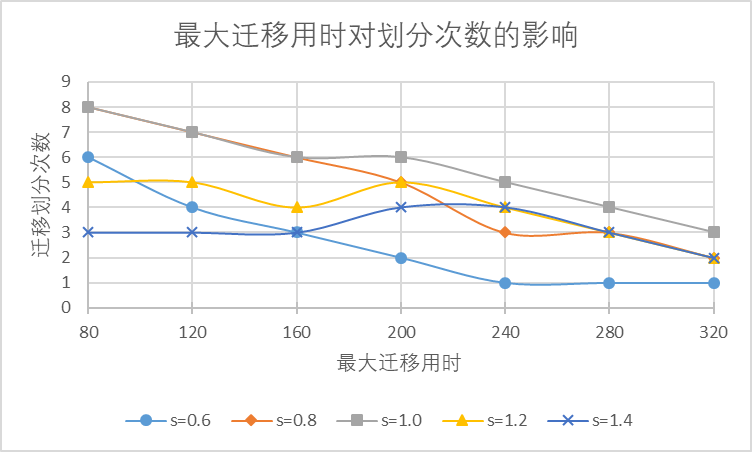
用户定义函数类型为，，即计算型负载为线性且存储型负载也是线性的情况下，图中结果表明，S1选择模型相比其他分组策略生成模型（Readj、Redist）更有优势，在倾斜程度较小时，三者迁移代价均比较低；随着倾斜程度加大，Readj和Redist的迁移代价上涨较快，而S1选择模型的迁移代价涨幅较小。一次S1细粒度迁移和两次S1细粒度迁移的负载均衡优势在这种用户定义函数下依然保持良好。

## 最大用时划分策略和自平衡划分策略的分析

最大用时策略需要用户给定最大用时这一参数，该策略可以保证在迁移时间小于最大用时的情况下迁移的状态收益最大。但是这种划分策略的效果受最大用时这一参数影响较大，如果用户对数据倾斜状况不够了解或者用户经验不足而设置不当，可能导致细粒度迁移表现较差，而且系统网络状况也在随时变化，固定的迁移用时显得不够灵活。因此本文做实验测试了最大迁移用时对均衡收益和划分次数的影响。



1）最大迁移用时对均衡收益的影响



2）最大迁移用时对迁移划分次数的影响

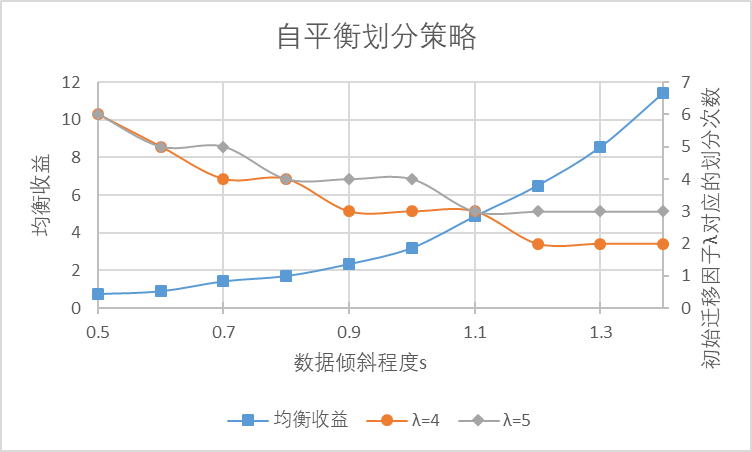
**图5-6** 最大迁移用时对均衡收益和划分次数的影响

如图5-6 1）所以，时，数据倾斜幅度较小，最大迁移用时设置为80ms或者320ms都可以正常迁移数据，均衡收益是相同的；当时，最大迁移用时设置为80ms已经不足以完成迁移，因为倾斜幅度变大导致有些状态迁移的时间大于，所以设置为80ms的均衡收益无法达到设置为320ms的均衡收益；当时，数据倾斜幅度已经非常大了，有些状态非常大，迁移这些状态需要很多时间，只有将设置为280ms或320ms才可以正常完成迁移。

最大迁移用时的设置同样会影响划分次数，如图5-6 2）所示，划分次数随着最大迁移用时的增加而减少。将设置过大会导致划分次数只有一到两次，这就无法体现细粒度迁移比一次性迁移的优势。设置过小会导致迁移次数较多，完成细粒度迁移的周期太长，而且对于倾斜幅度过大的数据，例如，太小的因为无法迁移某些状态而表现为划分次数比较少。

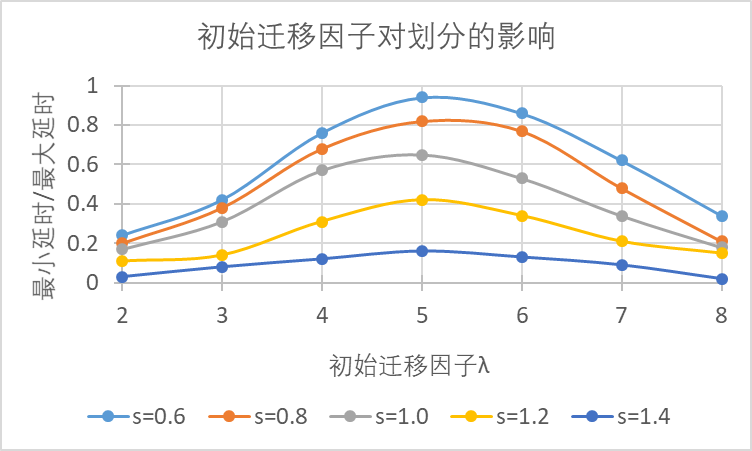
自平衡划分模型对迁移用时建立了代价模型，以此模型自动确定每次迁移的状态大小，不需要用户显式地指定最大迁移用时，避免了用户因经验不足或对数据倾斜情况不够了解导致的迁移效果较差的情况。本文对自平衡划分策略在不同数据倾斜幅度下做了比较。

图5-7表明，使用自平衡划分策略可以使均衡收益随数据倾斜程度正确增加，也就是自平衡划分策略会保证迁移的完整性。自平衡划分策略通常将初始迁移因子设置为4或5，通过影响第一次迁移状态的大小影响延时测量结果，从而影响了后续的划分次数，但是后续的划分次数是由之前若干次迁移的延时测量结果共同决定的。



**图5-7** 自平衡划分策略在不同数据倾斜程度下的表现

图5-7中可以看出初始迁移因子较小的情况下划分次数相对较少，但是无法直观的观察到初始迁移因子对细粒度迁移的有害/有利影响。所以继续实验，观察初始迁移因子和这组细粒度迁移中最小延时与最大延时的比例之间的关系，延时的比例大小可以反应划分是否均匀，



**图5-8** 初始迁移因子对划分的影响

观察图5-8，可以发现初始迁移因子设置在4~6比较合适，这时的最小延时/最大延时比例更加接近1，这意味着划分结果比较均匀，各次细粒度迁移的开销基本一样。当初始迁移因子设置过小时，因为第一次尝试迁移了太多状态，导致后续可迁移的状态太少，即最小延时/最大延时比例接近0；当初始迁移因子过大时，由于第一次尝试迁移的状态太少，导致后续迁移堆积了较多状态，而且第一次尝试迁移的状态太少也会导致测量不准确，为后续划分造成较大误差。

综上所述，在最大迁移用时设置合理的情况下，最大用时划分策略和自平衡划分策略表现相似。但是最大迁移用时的设置取决于数据倾斜程度和分布式环境的网络状况，本身难以预估，需要多次尝试才可以确定比较合理的值，所以自平衡划分策略对不同分布式环境和不同倾斜数据的适应性更好。

## 本章小结

本章先用实验展示了细粒度迁移相比一次性迁移的优势，即S1负载均衡模型的正确性验证，实验表明细粒度迁移不会造成非常大的延时，并且可以在一到两次细粒度迁移后将负载不均衡程度降低至触发阈值以下，本文使用了拍卖行竞拍数据完成实验，该数据集倾斜情况有多次变动，S1负载均衡模型可以应对所有变动并在保持分布式一致性的前提下均衡负载，与预期相符。

本章然后说明了细粒度迁移在并行度方面的适应性；

之后对比了S1选择模型和其他分组策略生成模型的优劣；最后探讨了最大用时划分策略和自平衡划分策略的特点和适用性。

结论

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，保证后续到达的数据可以均衡地分配给下游算子，同时需要状态迁移机制的支持，保证更改分组策略后，状态的存储位置可以对应新的分组策略。本文研究流计算平台数据特征及延时产生原因，提出S1负载均衡模型，减少一次性迁移的延时，以低代价完成负载均衡。本文的主要研究成果包括：

1）本文为S1负载均衡模型提出细粒度状态迁移机制，将一次性迁移划分为多次细粒度迁移，从而降低单次迁移的代价，使迁移带来的延时不那么突兀，细粒度负载均衡策略基于此机制才得以实现。基于异步检查点机制，确定分组策略替换时机和状态迁移时机，可以在保持分布式一致性以及exactly-once语义的前提下变更分组策略以及迁移状态，从而优雅地处理数据倾斜问题。

2）为了实现细粒度状态迁移机制，本文提出S1选择模型。定义负载计算方式以及负载不均衡代价和迁移代价的计算方式。本文需要在多次细粒度迁移中，选择收益最大（即代价最小）的状态优先迁移，剩余状态交由后续的细粒度迁移完成。S1选择模型提供了状态迁移的目的地和状态优先级排序队列，可以看作一种特殊的分组策略自动构建模型。本文为S1选择模型提出了两级路由分组策略，将高频数据和低频数据分别分组，在尽量保证大多数低频数据不变的情况下为高频数据重新分配算子，而且动态路由更适合多次调整。

3）为了实现细粒度状态迁移机制，本文提出S1划分模型。本文发现一次细粒度迁移有固定的最小用时，因此收集上一次细粒度迁移的用时，从而推测迁移剩余状态用时，由剩余用时和最小用时可以计算出后续迁移最合适的次数以及单次迁移的状态大小。本文提出了两种划分策略：最大用时划分策略和自平衡划分策略。最大用时划分策略可以在用户指定的最大用时内完成一次细粒度迁移。自平衡划分策略的模型更加准确，而且不需要用户显式地给出最大用时这一要求，对细粒度迁移的适应性更强

本文在Apache Flink平台上实现了细粒度负载均衡模型，命名为S1负载均衡器，并对系统参数进行了分析，实验表明本文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。本文还将S1选择模型与其他分组策略生成模型进行对比，发现S1选择模型在在均衡收益和迁移代价方面均比其他模型有优势，而且本文还发现一次细粒度迁移和两次细粒度迁移在均衡收益和迁移代价方面已经有很大提升，这说明S1选择模型的收益计算非常合理，一次或两次迁移的结果已经接近其他模型的完成全部迁移的结果。本文还对比了自平衡划分策略和最大用时划分策略的差异，最大用时划分策略的最大用时参数选取需要一定经验以及对数据集的了解，而自平衡划分策略则可以自动选择每次迁移的状态量，对使用者更加友好。

综上所述，本文通过对分布式流计算系统分组策略的研究和对细粒度状态迁移机制的研究，针对流计算系统常见的数据倾斜问题，提出了S1负载均衡模型在运行时自动均衡负载。本文还存在着一些不足之处，例如当检查点间隔设置过大并且数据倾斜情况变化频繁的时候，基于检查点发生的细粒度迁移响应过于缓慢，不能及时处理倾斜问题。未来的研究方向可以在检测到数据倾斜后减小检查点间隔从而尽快响应数据倾斜；或者让负载均衡指令从检查点中脱离出来，增加具有相似机制的其他数据，保证一致性的前提下异步异步负载均衡。

参考文献

1. H. Chen, F. Zhang and H. Jin. Popularity-aware differentiated distributed stream processing on skewed streams[J]. IEEE 25th International Conference on Network Protocols (ICNP), 2017, 10(12):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[J]. Data Engineering, 2015, 38(4): 23-34.

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[C]. Data Engineering, 2015, 42(12): 81-128.

A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron[C]. 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[C]. In Proceedings of the 2013 ACMSIGMOD international conference on Management of Data, 2013: 725–736.

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

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

1. Buğra Gedik. 2014. Partitioning functions for stateful data parallelism in stream processing[J]. 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[C]. Database Systems for Advanced Applications: 22nd International Conference, 2017: 623-635.
3. 何忠政. 分布式实时系统任务容错调度优化算法研究[D]. 2016.
4. J. Fang, R. Zhang, T. Z. J. Fu, Z. Zhang, A. Zhou and X. Zhou. Distributed Stream Rebalance for Stateful Operator Under Workload Variance[C]. IEEE Transactions on Parallel and Distributed Systems, 2018: 2223-2240.

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

1. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale[J]. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013: 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[C]. In ICDE, 2002: 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[C]. 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[C]. Compute Netw, 1999. 31(11–16): 1203– 1213.
6. A. Shukla and Y. Simmhan. Toward Reliable and Rapid Elasticity for Streaming Dataflows on Clouds[C]. IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018: 1096–1106.
7. Raul Castro Fernandez, Maeo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management[C]. In Proceedings of the 2013 ACM SIGMOD, 2013: 725–736.
8. Gedik B. Partitioning functions for stateful data parallelism in stream processing[J]. The VLDB Journal—The International Journal on Very Large Data Bases, 2014, 23(4): 517-539.
9. Rivetti N, Querzoni L, Anceaume E, et al. Efficient key grouping for near-optimal load balancing in stream processing systems[C]. Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. ACM, 2015: 80-91.
10. 刘智亮.面向流数据处理的动态自适应检查点机制研究[D].吉林大学,2017.
11. Graham R L. Bounds on Multiprocessing Timing Anomalies[J]. SIAM Journal on Applied Mathematics, 1969, 17(2):416-429.
12. Egghe L. Zipfian and Lotkaian continuous concentration theory[J]. Journal of the Association for Information Science & Technology, 2014, 56(9):935-945.
13. Caneill M, El Rheddane A, Leroy V, et al. Locality-aware routing in stateful streaming applications[C]. Proceedings of the 17th International Middleware Conference. ACM, 2016: 4.
14. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs[J]. SIAM Journal on scientific Computing, 1998, 20(1): 359-392.
15. Guo Q, Zhou Y. Stateful Load Balancing for Parallel Stream Processing[C]. European Conference on Parallel Processing. Springer, Cham, 2017: 80-93.
16. Zhang X, Chen H, Hu F. Back Propagation Grouping: Load Balancing at Global Scale When Sources Are Skewed[C]. 2017 IEEE International Conference on Services Computing (SCC). IEEE, 2017: 426-433.
17. McAuley J J, Leskovec J. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews[C]. Proceedings of the 22nd international conference on World Wide Web. ACM, 2013: 897-908.
18. Katsipoulakis, Nikos R., Alexandros Labrinidis, and Panos K. Chrysanthis. A holistic view of stream partitioning costs[J]. Proceedings of the VLDB Endowment, 2017, 10(11): 1286-1297.
19. Fang J, Chao P, Zhang R, et al. Integrating workload balancing and fault tolerance in distributed stream processing system[J]. World Wide Web, 2019: 1-26.
20. 蒋丰景, 陈玥琪. 复杂网络中节点重要度的一个评估指标[J]. 西安工程大学 学报, 2014(1):140-142.
21. Nasir M A U, Morales G D F, Garcia-Soriano D, et al. The power of both choices: Practical load balancing for distributed stream processing engines[C]. 2015 IEEE 31st International Conference on Data Engineering. IEEE, 2015: 137-148.
22. Mitzenmacher M. The power of two choices in randomized load balancing[J]. IEEE Transactions on Parallel and Distributed Systems, 2001, 12(10): 1094-1104.
23. Byers J W, Considine J, Mitzenmacher M. Geometric generalizations of the power of two choices[C]. Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2004: 54-63.
24. Chen H, Zhang F, Jin H. Popularity-aware differentiated distributed stream processing on skewed streams[C]. 2017 IEEE 25th International Conference on Network Protocols (ICNP). IEEE, 2017: 1-10.
25. Chen F, Wu S, Jin H. Network-Aware Grouping in Distributed Stream Processing Systems[C]. International Conference on Algorithms and Architectures for Parallel Processing. Springer, Cham, 2018: 3-18.
26. Nasir M A U, Morales G D F, Kourtellis N, et al. When two choices are not enough: Balancing at scale in distributed stream processing[C]. 2016 IEEE 32nd International Conference on Data Engineering (ICDE). IEEE, 2016: 589-600.
27. Fan L, Cao P, Almeida J, et al. Summary cache: a scalable wide-area Web cache sharing protocol[J]. IEEE/ACM Transactions on Networking, 2000, 8(3):281-293.
28. Pacaci A, Özsu M T. Distribution-Aware Stream Partitioning for Distributed Stream Processing Systems[C]. Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. ACM, 2018: 6.
29. 王蕾.基于复杂网络的软件关键节点和关键路径挖掘方法研究[D]. 燕山大学, 2016.
30. Metwally A, Agrawal D, El Abbadi A. Efficient computation of frequent and top-k elements in data streams[C]. International Conference on Database Theory. Springer, Berlin, Heidelberg, 2005: 398-412.
31. Carter J L, Wegman M N. Universal classes of hash functions[J]. Journal of computer and system sciences, 1979, 18(2): 143-154.
32. Karger D, Sherman A, Berkheimer A, et al. Web caching with consistent hashing[J]. Computer Networks, 1999, 31(11-16): 1203-1213.
33. Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web[C]. STOC. 1997, 97: 654-663.
34. Arasu A, Manku G S. Approximate counts and quantiles over sliding windows[C]. Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2004: 286-296.
35. Golab L, DeHaan D, Demaine E D, et al. Identifying frequent items in sliding windows over on-line packet streams[C]. Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement. ACM, 2003: 173-178.
36. Stoica I, Morris R, Karger D, et al. Chord: A scalable peer-to-peer lookup service for internet applications[J]. ACM SIGCOMM Computer Communication Review, 2001, 31(4): 149-160.
37. McAuley J J, Leskovec J. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews[C]. Proceedings of the 22nd international conference on World Wide Web. ACM, 2013: 897-908.
38. 梁毅, 侯颖, 陈诚, et al. 面向大数据流式计算的任务管理技术综述[J]. 计算机工程与科学, 2017, 39(2) : 215-226.
39. 唐应辉. 排队论：基础与分析技术[M] 排队论:基础与分析技术. 2006.
40. Hirzel M, Soulé R, Schneider S, et al. A catalog of stream processing optimizations[J]. ACM Computing Surveys (CSUR), 2014, 46(4): 46.
41. Shuang X, Pei W. Identifying important nodes by adaptive LeaderRank[J]. Physica A Statistical Mechanics & Its Applications, 2017, 469:654-664.

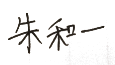
攻读硕士学位期间发表的论文及其它成果

1. 杨李杨, 陈思远, 张展, 朱和一, 左德承. 基于分布式流处理系统的时间感知分组算法[C]. 2019全国容错计算学术会议. 2019.
2. 朱和一，张展，陈思远，潘江浩，万丁. 基于异步检查点机制的细粒度流分区负载均衡方法[C]. 2020 中国计算机学会测试学术会议. 2020.

哈尔滨工业大学学位论文原创性声明和使用权限

学位论文原创性声明

本人郑重声明：此处所提交的学位论文《基于异步检查点机制的细粒度流分区负载均衡方法研究》，是本人在导师指导下，在哈尔滨工业大学攻读学位期间独立进行研究工作所取得的成果，且学位论文中除已标注引用文献的部分外不包含他人完成或已发表的研究成果。对本学位论文的研究工作做出重要贡献的个人和集体，均已在文中以明确方式注明。



作者签名： 日期： 2020 年 6 月 21 日

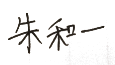
学位论文使用权限

学位论文是研究生在哈尔滨工业大学攻读学位期间完成的成果，知识产权归属哈尔滨工业大学。学位论文的使用权限如下：

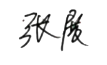
（1）学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文，并向国家图书馆报送学位论文；（2）学校可以将学位论文部分或全部内容编入有关数据库进行检索和提供相应阅览服务；（3）研究生毕业后发表与此学位论文研究成果相关的学术论文和其他成果时，应征得导师同意，且第一署名单位为哈尔滨工业大学。

保密论文在保密期内遵守有关保密规定，解密后适用于此使用权限规定。

本人知悉学位论文的使用权限，并将遵守有关规定。



作者签名： 日期： 2020 年 6 月 21 日

导师签名： 日期： 2020 年 6 月 21 日

致 谢

两年的硕士生涯已经在不知不觉间接近尾声，回想当时刚进入实验室的场景依旧历历在目。同时，硕士生活的结束也代表着我学生时代的结束，代表着我在哈工大六年学习生涯的结束，代表着我在哈尔滨生活的结束。大学生活虽然有着许多的烦恼，但也获得了很大的收获，包括学习上、生活上以及思想上等各个方面。我很荣幸能够在哈工大学习和生活，我将牢记哈工大的精神继续前行。

首先感谢我的导师张展教授。张老师无论在学习上还是生活上都给予我许多无私的帮助。张老师虽然教学和管理工作繁忙，却依然关注于我们学习情况和课题的进展，并悉心进行指导。张老师对我的谆谆教诲，我将一直铭记在心。张老师严谨的治学态度、求真务实的精神将一直激励着我，对我产生深远的影响。张老师对于我在学习、生活、工作等各个方面都给予了我很大的帮助。当我遇到困难时，张老师每次都会支持我并帮助我指明方向并细心的教导。张老师工作上尽职尽责，将每位学生和实验室的各种事务作为自己的责任扛在肩上，并且细心、用心的完成每一项工作。回顾两年的硕士生活，我在此由衷的感谢张老师。

感谢杨孝宗老师、刘宏伟老师、吴智博老师、董剑老师、温东新老师、舒燕君老师和罗丹彦老师，在学习和课题上对我进行了耐心的教导和帮助。

感谢陈思远师兄和杨李杨师兄。杨李杨师兄在学习、生活、工作上给我许多帮助。我们多次一起工作和研究，陈思远师兄以其博学的知识和敏捷的思维总能解决许多实际中遇到的问题，并提出新的见解，是我遇到困难时的可靠灯塔。陈思远师兄的科研态度、高尚的品质值得我去学习和敬佩。

感谢周鹏师兄、李文浩师兄和陈新昊师兄对我的指导和帮助。感谢庆骁、安宜豪、徐彬、庞洁、孙日辉等同学，我们一直在实验室陪伴度过每一天，一起生活、学习和工作，他们给了我许多帮助和鼓励。

最后感谢我的父母。感谢我的父母将我养育成人，支持我的学业。无论我遇到什么困难和挫折，总是他们在支持我、鼓励我，帮助我度过那些烦恼的岁月。我将脚踏实地，立身行道，才能报答我的父母，不辜负父母的期望。