基于事件驱动架构的分布式流处理弹性资源分配策略研究

汤小春 张 克 赵 全 李战怀
(西北工业大学计算机学院 西安 710129)

摘 要 针对具有多个数据源以及多个输出的流处理应用,使用单个分布式数据流引擎开发时,不论在架构还是可扩展性方面都存在着不足,而基于事件驱动架构的分布式流处理技术是解决该问题的主要方式.但是,事件驱动架构应用于流处理时,往往面临着数据注入速率与数据处理速率不一致的矛盾,当流数据源的数量发生变化、数据值的分布发生波动时,会导致处理延迟加大或资源利用不充分.针对数据注入与数据处理不一致的问题,现有的弹性资源分配策略难以有效处理生产者和消费者之间的依赖关系,且资源分配效果欠佳.论文提出了一种基于强化学习的弹性资源分配方法,解决了具有依赖关系的流处理应用程序之间的数据波动带来的延迟或者资源利用不充分的问题.通过建立状态矩阵和命令矩阵,使得资源管理器能够感知上下游应用的状态变化,从而及时调整流处理应用的资源需求,保证了流处理应用执行过程的延迟要求,提高了系统的资源利用率.经过测试,基于强化学习的弹性资源分配与Spark动态资源分配方法相比,延迟能减少15%,资源利用率能提高20%以上,其吞吐量能够提高10%左右.

关键词 事件驱动;分布式流处理;弹性资源;强化学习;数据注入
中图法分类号 TP301 **DOI号** 10.11897/SP.J.1016.2023.00244

Elastic Resource Allocation of Distributed Streaming Process Based on Event Driven Architecture

TANG Xiao-Chun ZHANG Ke ZHAO Quan LI Zhan-Huai
(*School of Computer Science, Northwestern Polytechnical University, Xi'an* 710129)

Abstract With the development of cloud computing and big data, many data stream engines have appeared, such as Spark or Flink. When a single distributed data stream engine is used to develop a distributed stream processing application with multiple data sources and multiple query targets, there are some shortcomings not only in architecture, but also in scalability, which increases the difficulty for developers to develop large-scale applications. Therefore, distributed stream processing based on event-driven architecture has been adopted by more and more developers. It can perform well both in building simple or complex applications due to its low coupling and high scalability characteristic. An event-driven architecture based application consists of a series of single-purpose components responsible for asynchronously receiving and processing events. However, event-driven architecture faces the contradiction between the data injection rate and the data processing rate when the number of streaming data sources fluctuates and the distribution of data values changes, resulting in processing delays or insufficient resource utilization. Elastic resource allocation can solve the problems caused by data fluctuations by dynamically adjusting the resources allocated to the application based on changes in load pressure in the stream processing system, but the traditional

收稿日期:2021-11-16;在线发布日期:2022-09-10.本课题得到国家重点研发计划(2018YFB1003400)资助.**汤小春**,博士,副教授,主要研究方向为大数据处理、集群资源管理、图数据管理等.E-mail:tangxc@nwpu.edu.cn.**张 克**,硕士研究生,主要研究方向为大数据计算.**赵 全**,硕士研究生,主要研究方向为大数据计算.**李战怀**,博士,教授,主要研究领域为海量数据管理、大数据计算等.

strategy (e.g. back pressure mechanism or simple machine learning method) has shortcomings in event-driven architecture. It is difficult to effectively handle the dependencies between production and consumption, which is manifested as being unable to detect the fluctuations in the data generated by the producers in time to adjust the resources of consumers. Besides, it is difficult to grasp the timing of resource allocation, which will lead to the inability to provide resources in time and increase the delay of stream processing. This paper conducts an in-depth study and comparison of the existing elastic resource allocation strategies and analyzes the limitations of some existing strategies in detail. On this basis, we propose an elastic resource allocation method based on reinforcement learning, which uses a production/consumption model to solve the delay caused by data accumulation between dependent applications, or insufficient resource utilization caused by data shrinking. The reinforcement learning method can learn strategies to maximize benefits during interaction with the environment, which is suitable for practical stream processing application scenarios. By establishing a state matrix and a command matrix, the resource manager can perceive the status changes of upstream and downstream applications, so as to timely adjust the resource requirements of stream processing applications by expanding or reducing the corresponding number of executors, ensuring the low latency requirements of the stream processing job execution process, and improving the system resources utilization rate. We implement this strategy based on the Mesos resource management framework and validate the effectiveness of the strategy using applications that process real-world road video streams. After the evaluation of several of our applications, compared with exiting Spark dynamic resource allocation method, the reinforcement learning based flexible resource allocation method can reduce stream processing delay by up to 15%, increase resource utilization by more than 20%, and its throughput can be improved by more than 10%.

Keywords　event driven; distributed streaming process; elastic resource; reinforcement learning; data injection

1　引　言

随着分布式流处理(Distributed Streaming Process, DSP)[1-3]的数据源的不断扩大以及数据源与数据中心之间距离的增加,各种靠近数据源的预处理应用被大量使用[4].另外,随着用户查询需求的多样化,其数据处理逻辑也呈现不断复杂化的趋势[5].例如,在混合交通视频流检测中,数据源包含大量的摄像头,它们散布在城市的多个位置.为了减少网络传输,原始数据经过分辨率调整等操作后产生图片帧,再转换为带位置和时间标签的二进制数据发送到数据中心;数据中心在处理视频流数据时,可能面对两种不同的查询需求,一个需求是车牌识别,另外一个需求是人流量统计.在这样的场景中,视频流预处理、车牌识别以及行人流量计算等都具有各自特有的处理逻辑.如果使用单个 DSP 完成所有功能,可扩展性及性能方面都面临以下问题[6]:

(1)数据需要通过日志系统进行传递.由于数据源的预处理与数据处理引擎位于不同的位置,它们各自作为单独的应用存在,使用日志系统构成生产者和消费者模式.在数据源所在位置对原始数据进行清洗、过滤等预处理后,可以减少传入到流处理引擎的数据量.

(2)可扩展性差.在一些大规模流数据处理应用中,如果将所有的计算逻辑融合到单独的一个流处理引擎中,往往会造成系统的可扩展性困难[7].另外,当集中式的调度器无法及时响应各个操作符的消息时,会出现延迟的现象.

(3)失效恢复效率低下.为了实现确切一次性的消息处理,DSP 使用检测点技术来存储快照[8],当出现处理失败的情况时,重新加载快照数据并计算.如果将所有的计算逻辑都集成到单一的 DSP 中,快照的保存以及失效后重新计算的代价都非常巨大.

针对这些问题,一种基于事件驱动架构的分布式流处理[3,6,9-11]被广泛使用,它建立在分布式流处

理的平台上，将多个应用按照生产者和消费者模式通过日志系统[3]连接起来，形成事件驱动的应用. 日志系统作为数据通道，实现生产者和消费者之间的解耦. 生产者连续不断地产生事件流并发送到日志系统，消费者订阅事件并进行处理. 如图 1 所示，数据源被预处理程序处理后，其输出流被发送到日志记录系统. 各个流处理应用从日志记录系统获得数据并进行计算，计算结果要么输出到日志系统被其它流处理应用使用，要么输出到仪表盘等展示系统或者输出到存储系统.

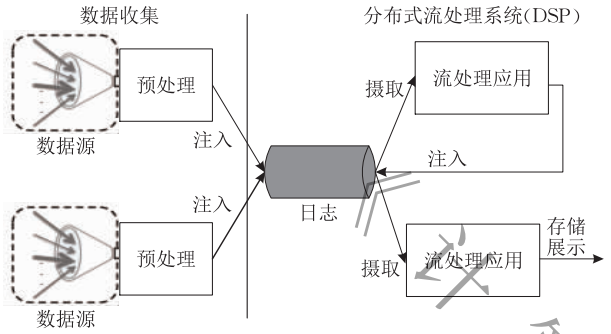


图 1 基于事件驱动结构的分布式流处理

基于事件驱动架构的流处理解决了多数据源和多处理输出的问题，但是，当数据源波动较大时往往会导致整个应用的处理延迟，甚至降低系统的资源使用率. 例如传感器等设备的低连接性或缺乏活动等导致数据源间歇性断开连接、数据源的数据分布特征导致输入数据动态变化以及流处理应用产生的数据值的偏差等，使得前一个应用产生的数据无法被后续的应用及时处理. 因此，基于事件驱动架构的分布式流处理应用必须能够以自适应方式弹性分配计算资源，实现数据注入与数据处理的一致性，达到高效使用计算资源的要求[11].

当作为消费者的流处理应用收到数据后，再根据注入数据的大小进行资源的动态调整，其调度时机会存在滞后，降低了系统的响应时间. 另外，生产者输入流数据的任何改变(如流数据的数量、数据注入速率或值的分布)都会影响后续消费者所需的资源量. 由于存在这两方面的因素，基于事件驱动架构的分布式流处理系统在弹性资源的分配方面，存在着以下的挑战：

(1) 数据注入和实时处理难以保持一致. 事件驱动的流应用采用微批次处理，流数据的产生者可能是一些边缘计算应用[4]，产生的数据输出到日志系统，每个微批次输出可能波动较大，消费者从日志系统获取数据后，由于资源不足造成无法及时处理该批次的数据，使得注入与处理不一致. 例如，当数据源的数量增加或减少，或者原始数据经过过滤等操作使得注入速度增强或者减弱时，消费者一侧无法感知生产者的状态变化，导致两者之间无法同步.

(2) 上下游 DSP 的状态难以感知. 事件驱动的流处理应用采用松耦合方式，往往由多个 DSP 组成，这些系统之间存在依赖关系. 不幸的是，由于不同的 DSP 处理逻辑不一样，无法采用单一的资源分配方式为整个流处理任务弹性分配资源. 也就是说无法对不同 DSP 的状态进行感知，因此它们就无法采用相同的策略来控制下游应用程序的实例数量，在多个流处理应用之间协调资源的弹性分配.

(3) 触发弹性资源分配的时机难以把握. 事件驱动的流处理采用生产者/消费者模式，下游的 DSP 无法根据上游的输出缓冲区来进行调度. 当下游流处理引擎发现存在大量未运行的任务或者大量的计算资源空闲时，系统才触发弹性资源的分配，但这样往往会造成资源分配的延迟，不能在下游作业启动前就重新配置计算资源.

无论从系统结构还是弹性资源分配策略来看，现有的分布式流处理应用的弹性资源分配机制都无法满足基于事件驱动架构的分布式流处理应用. 现有流处理弹性资源分配的系统结构只针对单个应用，无法跨越多个流处理应用，也无法在这些流处理应用之间进行全局的协调. 另外，现有流处理弹性资源分配的策略主要采用阈值、预测模型以及机器学习等[12-26]，它们根据节点或者操作符的性能指标动态增加操作符的并行性或者迁移操作符. 由于其仅仅对某个操作符进行扩展或者缩小，没有考虑操作符之间的数据流依赖关系，一个操作符的扩展可能导致后续多个操作符的重新配置，往往会引起大量扩展开销.

总之，这些弹性资源分配策略只针对单个 DSP 进行，很少关注数据源处理状态以及数据注入的变化，即消费者不监视和管理生产者的状态和输出流. 例如，生产者不断产生数据并将数据进行分片，然后发送到日志系统，如 kafka 等，消费者获取数据并根据分片产生大量的任务. 由于缺乏生产者和消费者之间的数据传输特征，也不对这些数据源进行监控，因此，这样的弹性资源方式存在两个问题，一是调度时机滞后，二是不能充分利用数据产生者的特征.

本论文采用基于 DataFlow 模型[27]的流处理计

算框架 Spark Structured Streaming[28]来构建事件驱动的分布式流处理应用，旨在根据流数据生产者的变化来动态调整计算资源，即根据数据源的数量变动、数据注入率和数据值的分布的重大波动而动态地增加或者减少计算资源，确保数据注入和数据处理的一致性，实现高吞吐量和低延迟的要求.

论文以 Mesos 资源管理框架[29]为基础，设计了一种以使用状态矩阵和命令矩阵为基础的弹性资源管理框架，可以满足不同位置的多数据流源以及多个 DSP 集成场合的统一弹性资源分配；将数据流的注入速度和资源使用量统一起来，形成系统的状态，将每个微批次作业的延迟当作激励因素，通过使用强化学习的方法，充分利用历史数据，实现了一个多流数据源的动态自适应的资源调度中间件，达到了弹性资源分配要求，保证了低延迟要求及提高了资源使用率. 论文的主要工作点是：

(1) 设计了一个满足数据注入和数据处理统一的弹性资源管理中间件；(2) 提出了状态矩阵和命令矩阵，使得 DSP 之间能够相互感知数据注入特征；(3) 通过强化学习方法，为流处理应用提供动态资源分配.

在接下来的章节中，第 2 节描述相关技术；第 3 节给出系统的总体结构；第 4 节给出基于强化学习的弹性资源分配算法；第 5 节结合 Mesos 资源管理和 Spark 流处理框架，采用强化学习方法实现弹性资源分配；第 6 节给出系统的性能评价.

2　相关技术

分布式流处理引擎分为原生模式以及微批模式两种，在弹性资源分配方面，前者更注重有向无环图(Directed Acyclic Graph，DAG)的部署及优化，其延迟较低；而后者更注重以微批次为单位计算，动态分配计算资源，具有更好的吞吐量[30].

2.1　原生处理模式的弹性资源分配策略

原生处理模式的弹性资源分配策略要么采取静态优化，要么采取动态优化. 另外，还涉及缩小/扩大资源的决策以及操作符迁移带来的重新配置的成本. 文献[31-36]采用阈值方式，当计算节点或操作符资源利用率超过阈值时，调整操作符的并行实例数量. 文献[31]采用单个静态定义的阈值来控制计算节点之间的负载不平衡. 文献[32-33]采用多个静态定义的阈值，允许自定义系统中每个计算节点的行为. 文献[34-35]采用动态配置的阈值提高了系统的适应性. 文献[36]提出了一种考虑重新配置成本的策略：首先通过模型估计操作符重新分配引起的延迟峰值，然后采用背包问题优化操作符的部署位置. 文献[18-19,37]使用更复杂的集中策略来确定弹性资源分配决策，依赖于全局模型知识进行优化. 文献[37]采用控制理论，提出了一种基于主动控制的策略，依据有限的未来时间范围来选择重新配置资源. 文献[18]采用排队论，通过排队理论来预测延迟并强制执行延迟约束. 文献[19]采用模糊逻辑，提出了一种两级自适应解决方案：在短时间尺度上，使用控制理论方法来处理负载不平衡，而在长时间尺度上，使用模糊逻辑来决策操作符的扩大或缩小. 此外，文献[20]提出了一个逐步分析的框架，该框架考虑了计算资源和应用程序处理能力，并有选择地评估了几种可能的并行配置的效率.

以上基于阈值的弹性策略代表了弹性资源分配经典和最简单的方法，但是，它们要求用户知道如何设置相应的阈值. 为了让用户免于定义阈值，一些诸如机器学习的自动决策方法被提出，它们允许用户专注于目标，而不是如何获得阈值. 文献[21]采用模糊逻辑，提出了一个自动缩放控制器，该控制器在运行时使用强化学习(Reinforcement Learning，RL)修改模糊规则. 文献[22]观察到 RL 方法可能会受到维度问题的影响，因为查找表必须为每个可能的状态-动作对存储一个单独的值. 为了克服这些问题，他们将 RL 与基于队列的系统模型相结合，该模型计算初始部署决策并驱动探索行动. 出于减少状态空间的相同目标，文献[23]将经典强化学习算法与模糊推理系统相结合，而文献[24]提出了并行强化学习方法.

文献[25]为 IBM Streams 提出了一种同时利用管道并行性和数据并行性的解决方案. 根据包含的运算符是否可以复制，将链状数据流图分割成区域. 对于可并行化的区域，分别在拆分和合并操作符之前和之后创建复制管道.

文献[26]考虑到底层硬件对于有效利用现代硬件的重要性，认为现有的系统没有多核处理器和高速网络，对 SPE 通用架构进行了设计更改，以在现代硬件上进行扩展.

总之，这些方式主要应用于原生流处理模式，不太适合微批方式的流处理模式，同时也无法跨越多个流处理引擎进行实时优化，它不太适合于面向事

件驱动的分布式流处理应用.

2.2 基于微批处理模式的流处理弹性资源分配

实现 DSP 弹性资源分配的关键是确定其自适应决策的方法.一个较优的策略目标是可以及时分配足够的资源,并不造成浪费.大多数方法是以计算资源为基准而实施尽力而为的策略[12,26].

Spark 流处理采用微批模式来实现流数据处理,提供了动态资源分配和背压机制[12-13],背压机制只能限制数据注入速度,无法满足实时性要求,动态资源分配根据等待任务的数量来增加或者回收资源,其资源分配的指标是在任务执行前确定的,是一种事前决策的方法.这提供了一种思路,通过背压机制,动态调整流处理应用上下文程序的状态,可以解决数据注入与输出的矛盾,使得应用内部达到平衡状态,同时调整 DSP 整体的资源分配,以达到和外部数据注入的平衡.文献[14]通过动态调整微批的大小来满足流处理作业的延迟要求.文献[15]考虑资源的使用来弹性增加或减少计算节点的数量.这些处理方式的最大缺点是阈值的设置比较困难,在数据流注入速度波动的情况下,弹性资源分配的效果不是太好.文献[16]提出了一种基于强化学习的自适应资源分配方法,但是它是在确定资源的情况下,自适应决策并行运行的作业之间占用的资源比例,前后作业之间的资源比例以及并行运行的作业的数量,所以说它无法保证数据注入速度变化时流处理应用的实时性.对于资源管理器来说,可以将外部的数据注入作为强化学习的输入参数,自适应决策 DSP 整体的资源分配,以达到和外部数据注入速率的平衡.文献[17]计算在一定间隔时间内到达的数据量以及资源使用量之间的关系,将数据的到达量控制在可处理的范围,从而避免数据处理间隔超过流数据的处理间隔,它的目的在于数据处理的顺畅,而不是弹性资源分配,但提供了评估数据注入速率的变化与资源量调整之间的方法.

文献[38-39]通过强化学习方法提出了不同的弹性数据流调度策略,该方法通过使用机器学习算法和集群的性能指标来自动缩放 DSP 的资源.但是这些方法针对的是单个应用程序上的弹性资源分配,无法扩展到不同的应用程序之间的弹性资源分配.而本论文则要求在不同的应用程序之间实施弹性资源分配,可以跨越不同的流处理引擎.

文献[40]的目的在于提高吞吐量,它通过将到达的流处理作业的 DAG 图转换为嵌套图的方式,使用强化学习方法来计算不同的嵌套图之间资源分配比例,从而保证系统的吞吐量,它不太关注数据注入速度的变化,只关注作业 DAG 图的特征,因此不适合数据注入速度变化的场合.

文献[41]针对同一个流数据处理应用中上下游操作符之间预测的不准确而导致的波动问题进行解决.提出了负载关联模型来提高预测精度,使用双向控制机制来决定资源的大小.它没有一个全局机制来解决多个具有关联关系的流处理引擎之间的数据注入与数据处理不匹配的问题.

以上的弹性资源分配策略,关注点都只在于单个 DSP 内部的应用程序的控制和调度,无法感知其上游生产者 DSP 状态的变化,因此不适合于多个 DSP 应用之间的控制和调度,即不适应事件驱动的流处理的弹性资源分配.

可以看到,此前的许多工作已经采取基于机器学习模型的资源分配算法来提高分布式流处理系统的性能或资源利用率.然而,考虑到流数据的动态性,这些传统的模型方法需要考虑到完整的系统状态,并进行高复杂度的计算.在此基础上的一些进一步研究使用深度学习模型作为一种数据驱动的方法,通过可用的训练输入和输出来降低计算复杂度.但使用神经网络首先需要用模拟的训练集数据对神经网络进行离线训练,然而系统面临的情况复杂多变,通常很难获得完全适用的数据集或最优解决方案,并且训练过程通常很耗时.

鉴于上述问题,强化学习方法可以作为实时动态决策问题(如动态资源分配)的一种可行的解决方案,它可以放宽对系统全局模型和训练数据的依赖,对长期运行的系统生成几乎最优的决策,而不是只优化当前的收益.

综合以上方法,对于具有依赖关系的多个 DSP,本文提出通过获得上游 DSP 应用中的输出流,为下游 DSP 的应用提前估计出需要的计算资源,达到数据注入与数据处理的一致性,不但满足实时性的要求,还提高了资源使用率.

3 系统模型

3.1 事件驱动的分布式流处理模型

事件驱动的分布式流处理应用是一类建立在流处理平台上的应用,能够从一个或多个事件流提取数据,并根据到来的事件触发计算、状态更新或其它外部动作.图 2 是一个基于事件驱动的分布式流处理模型.该模型包括流数据、应用和查询三个部分.

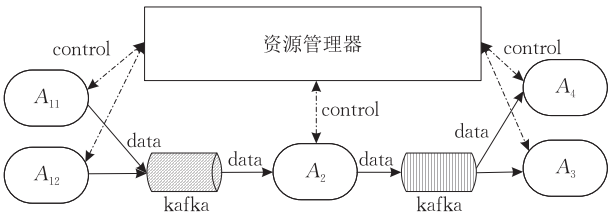


图 2　事件驱动的分布式流处理模型

3.1.1　流数据

流数据 e 是一个由无限元组构成的序列集合 $e=\{\tau\},\tau\in e$. 元组 τ 由逻辑时间戳 Δt、键 key 以及值 $data$ 组成，可以表示为 $\tau=(\Delta t,key,data)$. 当在流中创建元组时，时间戳 $\Delta t\in N^+$ 由运算符的逻辑时钟单调递增地分配.

图 2 中的带箭头的实线代表流数据，其包含的元组根据其时间戳进行排序，填充斜线的圆柱代表分布式日志系统，例如 kafka 集群，元组被缓存在日志系统中.

3.1.2　应用

应用用来处理流数据中的元组，例如一个微服务或者有状态的流处理操作. 对于应用 a，假设存在 n 个输入流组成的集合 I_a，a 从 I_a 中获得元组，处理这些元组并输出到一个或者多个新的数据流 D_a. 应用的计算过程表示为 f_a，它定义了应用对数据流的处理过程，即 $f_a:(I_a,w,\theta_a)\rightarrow(D_a,w,\theta_a)$. 对于每一次函数 f_a 的调用，应用接收一个有限个元组的集合 $I_a[w]$，其中 w 是处理间隔. 处理完成后，应用更新位置并进入下一个处理间隔. 在每一个处理间隔，应用访问状态 θ_a，并在函数执行后更新状态. 对于一个无状态的应用，$\theta_a=\phi$.

图 2 中的圆角矩形代表应用，A_{11} 和 A_{12} 是代表对原始流数据进行处理的应用. 例如一个基于 kafka 的流处理应用，对数据源进行简单的清洗和过滤等处理后，采用一定的分片规则将数据分片，然后写入分布式日志系统 kafka. A_2 作为一个单独的分布式流处理引擎，例如使用 Spark Structured Streaming 创建的应用，从 kafka 集群中订阅事件，根据处理间隔的设置来启动数据转换，并将结果数据写入 kafka 集群. A_3 作为另外的一个分布式流处理引擎，它从 kafka 集群中获得 A_2 的输出数据，然后进行分析处理，其处理结果可以直接输出到 A_3 和 A_4，A_3 可以是一个实时的仪表板，A_4 可以是一个类似于数据仓库的系统.

3.1.3　查询

一个查询 G 可以抽象为一个有向无环图 $G=(A,E)$，其中 A 代表应用的集合，E 代表流数据的集合. 一个流数据 $e\in E$ 是两个应用之间的有向边，表示为 $e_{ij}=(a_i,a_j)$，其中 $\{a_i,a_j\}\in A$. 对于每一个查询，都包含一个数据源 src 和结果汇 snk. 如果存在一个从应用 a_i 到应用 a_j 的流数据，那么称 a_i 为 a_j 的上游应用，表示为 $a_i\in up(a_j)$，称 a_j 为 a_i 的下游应用，表示为 $a_j\in down(a_i)$. 其中 $up(a)$ 为 a 的全部上游应用的集合，$down(a)$ 为 a 的全部下游应用的集合. 例如，对于图 2 中的 A_2 来说，A_{11} 和 A_{12} 是其上游应用，此时 A_{11} 和 A_2 之间以及 A_{12} 和 A_2 之间就存在依赖关系.

一个查询 G 启动执行后，G 中的应用 a 会产生 n 个并发的操作符[42]实例 $a^1,a^2,\cdots,a^n$. n 代表了应用 a 的资源使用量，它与集群的资源分配直接相关.

图 2 中直角矩形代表资源管理器，带箭头的虚线代表应用与资源管理器的交互. 一个查询在数据处理过程中，其包含的应用都向资源管理器发布自己的状态，资源管理器根据这些状态信息进行分析后，给出资源配置计划，然后将这些计划下发给各个流处理引擎，动态调整计算资源.

3.2　数据注入与数据处理一致性

影响数据注入与数据处理一致性的关键因素是资源调度器将多少资源分配于某个应用. 但是，整个查询的全局状态以及上下游应用的状态都会影响资源分配决策的正确性. 假设两个应用都拥有足够的资源，那么前一个应用产生的数据就可以很快被后一个应用消费，从而得到一个低延迟和低缓存的效果. 在一个流数据处理中，如果某个下游应用占有的资源太少就可能导致上游应用产生的数据不能及时处理，反之，它的下游应用则会因为没有数据而导致资源浪费.

上下游应用之间存在依赖关系，保证具有依赖关系的应用资源分配恰当的关键因素是：有效地表达数据大小的波动和应用当前的资源状态. 监视这些信息，并通过网络传输到全局资源管理器中，以期得到准确的系统状态. 另外，系统状态信息的准确性与资源分配的代价之间需要权衡，既要获得精确的数据注入的变化，又要防止带来的其它开销，这是弹性资源管理的基本要求.

对于一个查询，假设 W 是微批次处理间隔的集合，$W_i=\{(W_i^j)\}_{j\in N^+}$. 每个处理间隔 W_i 关联着一个应用 a_i，它由多个迭代 W_i^j 构成的序列组成. 每个迭代 W_i^j 被定义为一个时间区间 w，应用 a_i 会获得在此区间内收集的测量值. 这些测量是在预定的时间戳

$m_i=\{m_i^1,m_i^2,\cdots,m_i^n\}_{n\in N^+}$ 收集的. 对于每一个应用 a_i,在每个时间戳 m_i^d 收集了在间隔$[m_i^{d-1},m_i^d]$内产生或者处理的元组数,其中 $d\in\{1,2,\cdots,n\}$.

上游应用产生数据,下游应用消费数据,上下游应用之间存在一个流. 设 D_i是应用 a_i产生的数据的集合,它是无界的,$D_{i,d}$代表应用 a_i在时间区间 w_d 内产生的数据的集合. 图 3 中包含一个生产者应用 a_i和一个消费者应用a_j,a_i和a_j之间存在一个流 e_{ij}. a_i在检测点 m_0,m_1和 m_2产生数据,将这些数据累积可以得到在处理间隔 w_0内 a_i产生数据为 $D_{i,0}$. 在 t_0 时刻 a_j开始处理数据,设 a_j的注入数据大小为 $I_{j,1}$,理论上应当满足 $I_{j,1}=D_{i,0}$,系统根据注入数据为 a_j 分配必要的资源配额,保证 a_j的处理无延迟.

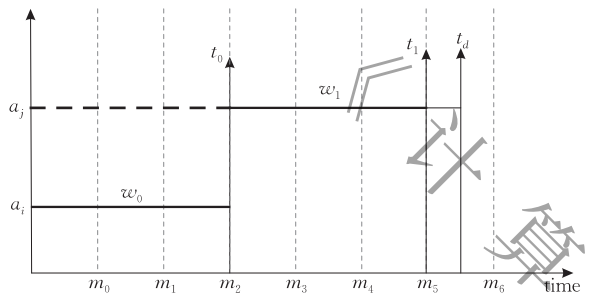


图 3 数据注入与处理一致性模型

在一个处理间隔 w 内,a_j的上游应用产生数据,a_j消费数据. 在一个处理间隔期间,a_j应当消费 $up(a_j)$产生的全部数据.

对于生产者应用 a_i,实际运行时可能存在 n 个相同的实例,即 a_i^k,$k=1,2,\cdots,n$. 假设在一个处理间隔 w 内,每个实例产生的数据数量为 D_i^k,则 a_i^k产生数据的速度记为 $q_i^k=D_i^k/w$,a_i产生数据的速度记为 $q_i=\sum_{k=1}^{n}D_i^k/w$. 那么,下游应用 a_j在下一个处理间隔期间,其需要消费生产者产生的数据,其数据处理速度可以表示为 $p_j=\sum_{x\in up(a_j)}q_x$.

数据注入速度应当与数据处理速度具有一致性. 对于应用 a_i与 a_j,如果 $q_i=p_j$,那么数据注入速度与数据处理速度就是一致的. 也就是说,对于任何一个流 e_{ij},如果应用 a_i在当前处理间隔产生的数据 $D_{i,d}$与下一个处理间隔 a_j 消费的数据 $I_{j,d+1}$相等的话,应用 a_i与 a_j之间的数据注入速度与数据处理速度是一致的.

如图 3 所示,在处理间隔触发时刻 t_0,消费者应用 a_j需要处理的数据规模为 $w_0\cdot q_i$,即生产者应用 a_i 产生的数据大小. 在处理间隔触发时刻 t_1,a_j处理数据的大小为 $w_1\cdot p_j$,如果 $w_0\cdot q_i=w_1\cdot p_j$,说明数据注入与数据处理是一致的.

在每个处理间隔,如果上游应用产生的数据能够被下游应用及时处理,那么数据就不会在系统中堆积,系统可以稳定的运行. 但是,当消费者应用无法在处理间隔内处理完全部的数据时,数据就会产生堆积,在图 3 中,如果 a_j在时刻 t_d才处理完全部数据,则其数据堆积量为$(t_d-t_1)\cdot q_i$,说明 a_j分配到的资源太少. 反之,如果 a_j在 t_1时刻前就处理完全部数据,则是向 a_j分配的资源太多.

资源管理器需要对查询中的所有应用的数据产生进行监视,并及时触发对应用的资源分配,即对应用获得的资源进行动态地弹性扩展或者收缩.

3.3 基于事件驱动架构的流处理弹性资源分配模型

在基于事件驱动的流处理应用弹性资源分配的过程中,需要根据数据注入速度的变化对应用的并行实例的数量进行增加或者减少,实例的数量代表资源的使用量. 资源管理器通过监测器得到上游应用的数据产生速度,然后结合下游应用的当前状态(实例数量),将这两类状态信息发送到资源管理器的决策机构,即智能代理(Agent). 当智能代理接收到数据注入速度以及应用的状态后,对资源是否进行扩展或者缩小进行决策,其目的是使应用达到最好的服务水平,即应用的延迟时间尽量接近数据流的处理间隔. 当应用的处理时间超过一个处理间隔时,说明资源太少,数据会堆积,延迟增大. 当应用的处理时间大大低于一个处理间隔时,虽然延迟满足要求,但是资源使用量太大. 应用执行的结果信息(延迟)也需要反馈给智能代理,它为智能代理的下一次决策提供参考依据.

流处理的查询执行特定操作(例如,过滤、聚合、合并)或更复杂的操作(例如 POS 标记等),而流数据是无界的数据序列. 图 4 给出了一个事件驱动的流处理弹性资源分配模型,图中的查询包含两个应用,一个生产者(Producer)和一个消费者(Consumer). Producer 产生数据,Consumer 消费数据,Resource Management 是资源管理器,用于向应用分配计算资源. 流处理查询表示为 $G_{dsp}=(A_{dsp},E_{dsp})$,其中 A_{dsp}代表 DSP 中的应用以及输入源和输出汇,E_{dsp}代表应用之间的流. 对于应用 $a_i\in A_{dsp}$,设 r_i代表 a_i所需要的计算资源,包括 CPU 核的数量、内存使用量、I/O 带宽等,L_i代表 a_i的处理延迟. $e_{ij}\in E_{dsp}$代表从应用 a_i到 a_j之间的流,q_i代表应用 a_i产生数据的速度,p_i代表应用 a_i处理数据的速度.

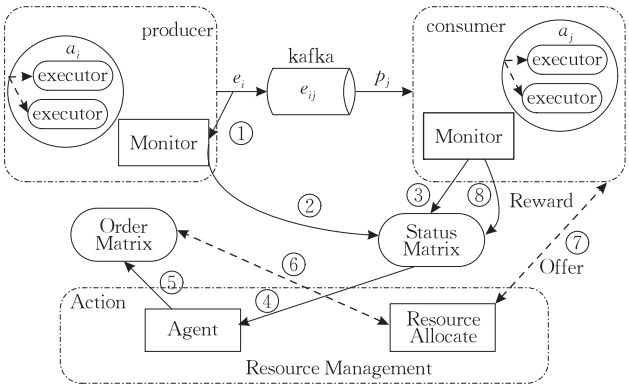


图 4　基于 RL 的应用程序资源分配模型

Producer 应用 a_i 产生流数据(图 4 中的①),其监控器(Monitor)定时向状态矩阵汇报应用 a_i 发送到 kafka 的流数据 e_{ij} 的信息,并存储到状态矩阵(Status Matrix)中(图 4 中的②). Consumer 的监控器定时汇报其资源使用信息,并存储到状态矩阵中(图 4 中的③). 智能代理从状态矩阵中获得信息(图 4中的④),经过决策,启动一个动作(如图 4 中的⑤),并写入命令矩阵(Order Matrix);资源管理器的资源分配模块从命令矩阵中获得最新的行动信息后(如图 4 中的⑥),预留计算资源,然后向流数据处理引擎发送资源的增加或者减少命令(如图 4 中的⑦);Consumer 使用最新的资源量,对当前微批次的数据进行处理,并将本次处理的延迟情况反馈到状态矩阵,再由智能代理进行新一次的决策.

3.4　状态矩阵

状态矩阵包含了系统中的所有可用资源,包括计算节点的硬件资源,以及应用和流数据. 对于任何一个应用 a,其状态可以表示为 $state(a,r,e)$,其中 r 代表资源,e 代表应用流数据的集合.

r 还可以表示为 $r(cpu,mem,io,net,gpu,gm)$,cpu 代表 CPU 使用量,mem 代表内存使用量,io 代表 I/O 带宽,net 代表网络带宽,gpu 代表 GPU 设备数量,gm 代表显存使用量.

e 表示为 $e=\{e_{11},e_{21},\cdots,e_{km}\}$,$|e|$ 表示应用的流数据的数量,其中 $k=|e|$,e_{km} 代表应用 k 输出的流数据,该流被应用 m 使用,m 可以是另外一个应用. q_i 表示第 i 个流数据的产生速度,p_i 表示第 i 个流数据的注入速度,其中 $1<i\leq k$.

对于图 4 中应用,其状态矩阵如表 1 所示. 流数据 e_{ij} 表示应用 a_i 产生数据,应用 a_j 消费数据. 对于应用 a_i,它产生数据的速度为 q_i,应用 a_j 的数据注入速度为 p_j,通过流 e_{ij},a_i 和 a_j 之间传递流数据,即 a_i 最新的产生数据速度为 a_j 最新的数据注入速度.

表 1　状态矩阵

应用	r	e_{ij}		e_{jk}	
		p	q	p	q
a_i	…		q_i		
a_j	2	p_j			q_j
a_k	…			p_k	

应用的状态资源信息由监控器提供,监控器定时采集数据并更新状态矩阵. 对于任何一个流数据,只要上游应用的输出速度与下游应用的注入速度不相等,就代表着应用的状态发生变化. 智能代理定期检查这种变化,一旦存在波动,就开始一次决策.

3.5　命令矩阵

命令矩阵中包含了需要为应用预留资源的指示,是由资源管理器分配给某个应用的资源信息. 与状态矩阵类似,每个应用对应着一个资源预留请求. 对于任何一个应用 a,其资源预留请求可以表示为 $action(a,r,v)$,其中 r 代表预留资源量,v 代表要执行的动作. r 表示为 $r(cpu,mem,io,net,gpu,gm)$,cpu 代表 CPU 使用量,mem 代表内存使用量,io 代表 I/O 带宽,net 代表网络带宽,gpu 代表 GPU 设备数量,gm 代表显存使用量. v 表示为增加预留量还是减少使用量,“+”表示预留资源数量,“−”代表减少的资源数量.

表 2 中的“+”代表系统需要为应用 a_j 预留资源,预留的资源数量如该行中的 r 列对应的数值所示. 对应用 a_i 来说,正常情况下 q_i 与 p_j 应当相同,智能代理不进行任何决策. 一旦 q_i 超过 p_j,并且达到一个界限,那么就必须为应用 a_j 增加计算资源,即 $v=+$;如果 q_i 小于 p_j,并且达到一个界限,那么就需要为应用 a_j 减少计算资源,即 $v=-$. 减少或增加资源的数量由智能代理进行决策.

表 2　命令矩阵

应用程序	r	v
a_i	…	…
a_j	1	+
a_k	…	…

3.6　资源管理器与应用之间的通信

在传统的分布式系统中,最常用的是客户服务器模式,即客户端发送请求到服务器,然后等待服务器的回复,客户端和服务器紧密耦合在一起. 对于共享的集群环境,各个 DSP 的操作符实例不会固定在某个计算节点,因此,最好的通信方式是使用发布/订阅模型. 在这种模型中,每个应用作为一个发布者发送状态信息,并且负责更新这些信息,任何与流数据处理相关的其它应用订阅这些消息. 每个应用只

需要订阅自己需要的那些信息，而不用关心这些消息的发布者是哪个应用. 发布与订阅采用异步的方式，发布者不需要等待接受者接收到消息后才进行后面的操作. 但是，作为集群系统，其基础设施要提供一种非常可靠的通信基础设置.

在基于事件驱动架构的分布式流处理应用中，我们希望能降低资源管理器、生产者、消费者、智能代理之间的控制消息的通信数量，因此使用发布/订阅模型，当资源状态信息和资源命令信息发生变化的时候，才进行信息的发布. 另外，采用发布/订阅模型可以满足系统的可扩展性，即随时可以添加新的生产者或者消费者进入流数据处理中，而系统的结构不做任何的变更.

图 5 给出了基于事件驱动架构的流处理应用的发布/订阅通信模型，该模型与流处理应用框架完全融合. 如前面章节所述，各个应用(DSP)的状态信息是由各个 DSP 发布的，存储在状态矩阵中. 每个 DSP 向状态矩阵中添加自己的状态信息，并且是该信息的拥有者，同时负责对自己的状态信息的更新. 资源管理器只关心状态矩阵中的信息，订阅这些数据. 同时，命令矩阵中的信息由智能代理负责发布和更新. 智能代理获得状态矩阵的数据后，根据数据注入速度的变化情况以及各个 DSP 已经使用的资源情况，为应用预留计算资源，同时向命令矩阵发送最新的资源分配请求. 资源管理器订阅命令矩阵，根据命令矩阵的信息来控制 DSP 资源使用量.

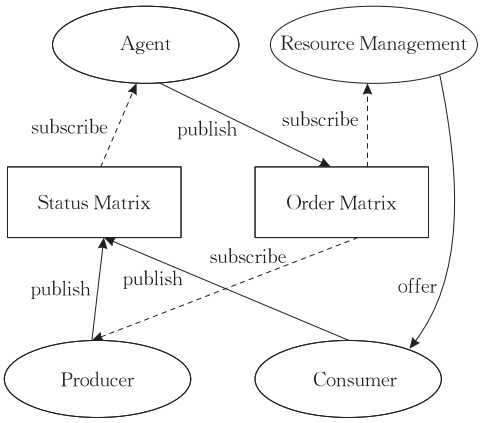


图 5 事件驱动的流处理应用的通信模型

发布/订阅机制仅仅用在状态矩阵和命令矩阵的存取部分，其它部分的通信数据传输，例如 DSP 自身的调度等不使用这种机制.

4 基于强化学习的弹性资源分配算法

针对注入速度与数据处理不匹配的问题，采用强化学习的方法来进行动态调节，即通过比较已完成的流数据处理的测量性能(延迟和吞吐量)来确定有效的配置. 由于 Spark Structured Streaming 具有周期性的运行属性和特征，通过针对特定工作负载类型的自适应调整配置，它能够不断改进资源分配解决方案. 当使用强化学习方法时，通过定义状态空间 S 和动作集 B，系统不断地进行参数调整.

4.1 问题描述

由于决策是周期性进行的，考虑一个固定长度的时间间隔 w 为系统的处理间隔. 在第 i 个处理间隔，设其时间范围为 $[\Delta t_i, \Delta t_{i+1}]$. 在该处理间隔，假设应用具有 h_i 个执行器，数据在第 $i-1$ 个处理间隔期间的数据产生速率为 λ_i. 在第个 i 处理间隔开始的时刻，得到数据注入速度为 λ_i，智能代理决策出一个动作 b，它决定了应用程序继续使用当前的资源配置还是重新进行资源的配置. 因此，我们把数据注入和数据处理的匹配问题就转化为一个马尔科夫过程(Markov Decision Process，MDP). 一个 MDP 被表述为一个 5 元组 $\langle S, B, p, c, \gamma \rangle$，$S$ 表示一个有限的状态集合，$B(s)$ 表示在状态 s 时有限的动作集合，$p(s'|s,b)$ 表示从状态 s 经过动作 b 后转换到状态 s' 的概率，$c(s,b)$ 代表在状态 s 经过动作 b 后，立即得到的代价，$\gamma \in [0,1]$ 代表计算将来的代价时的折扣系数.

在系统中，我们将 MDP 的状态表示为执行器的数量(容器的数量)h_i 和数据注入速度 λ_i 组成的元组，即 $s_i = \langle h_i, \lambda_i \rangle$. 为了进行分析，系统考虑一个离散的状态空间，即 $\lambda_i \in \{0, \lambda^-, 2\lambda^-, \cdots\}$，其中 λ^- 是一个合适的量子，例如每秒钟达到的元组个数. 执行器的数量决定了资源数量的大小. 为了加快查找速度，我们将这些参数的搜索空间缩小到一个合理的范围(例如 $h_i \in [h_{\min}, h_{\max}]$).

对于每一个状态 s，将动作调整的粒度设置为 1，动作矩阵 $\boldsymbol{B}$ 表示为 $[+1, 0, -1]$. 当 $h_i = h_{\min}$ 时，动作集合 $\boldsymbol{B}(s) = \{+1, 0\}$，即保证有最小数量的执行器存在，当 $h_i = h_{\max}$ 时，动作集合 $\boldsymbol{B}(s) = \{-1, 0\}$，即不能超过最大数量的执行器. 由于自动缩放决策和元组到达速率的变化，适配器会发生转换，使用 $p(s'|s,b)$ 来表示. 即

$$p(s'|s,b) = P[s_{i+1} = (h', \lambda') | s_i = (h, \lambda), b_i = b] = \begin{cases} P[\lambda_{i+1} = \lambda' | \lambda_i = \lambda], & h' = h + b \\ 0, & \text{其他} \end{cases} \tag{1}$$

很容易认识到，由于数据注入速度的变化，通过

智能代理给定的一个动作，使执行器的数量 h 变化为 $h'=h+b$.

在资源共享配置问题中，希望得到的配置可以提高整个系统的性能(例如，延迟和吞吐量). 对于每一个状态，我们设计一个收益函数，通过收益来衡量性能. 收益是用新产生的配置进行流处理得到的汇总性能的反馈，即当前吞吐量与参考吞吐量的比值加上违反延迟 SLA 时可能的惩罚.

$$reward=\frac{thrpt}{thrpt_{ref}}-penalty,$$
$$penalty=\begin{cases}0, & latency\leq SLA\\ \dfrac{latency}{SLA}, & 其他\end{cases} \tag{2}$$

吞吐量参考值 $thrpt_{ref}$ 是通过应用当前的默认顺序调度策略可实现的最大流处理速率. SLA 等于处理间隔，因此可以使应用保持与处理间隔相同的延迟. 以上约束保证了系统的延迟不会持续增加，并且可以保持稳定. 较低的收益表示资源浪费或作业执行中数据注入的速度和任务处理的速度不匹配，在调整参数时应避免这两种情况的出现.

4.2　搜索算法的实现

系统采用 MDP 转换来确定最优方案，其解决方案旨在最大化每个状态的累积收益. 等效于找到近似于其实际值的 $Q(s,b)$的估计值. 基于经验的解决方案中的基本 RL 算法称为时间差方法，每次收集样本时更新 $Q(s,b)$：

$$Q(s_{t+1},b_{t+1})=Q(s_t,b_t)+\theta\times[R_{t+1}+\gamma\times Q(s_{t+1},b_{t+1})-Q(s_t,b_t)] \tag{3}$$

其中，θ 是学习率，γ 是折扣系数. Q 值通常被存放在查找表中，通过将新值写入表中的相应条目来更新. 以任何初始策略为起始，适配器将根据在每个步骤中感知到的反馈逐渐完善策略.

算法 1.　RL 过程.

S1. 初始化 Q,V 函数

S2. LOOP

S3.　选择动作 b_i(根据当前的 Q 估计)

S4.　观察下一个状态 s_{i+1}以及引入的代价 c_i

S5.　更新 Q,V 函数

S6. END LOOP

算法 1 中的 S1 是初始化，设置为 0；S3 是适配器在每一个迭代步选择动作 b_i；S4 是观察产生的效益 c_i以及下一个状态 s_{i+1}；S5 是根据我们的期望更新 Q,V 函数. 基于 RL 的决策过程与现有的方式基本一致，这里不详细说明.

5　系统实现

5.1　分布式数据流弹性资源分配的实现

论文使用的编程框架为 Spark2.4，资源管理工具为 Mesos1.8. 对 Spark 的 JobScheduler、JobGenerator 和 StreamingListener 三个类进行了修改，可以定时向资源管理器汇报最新的状态，添加了一个新的组件来获取 kafka 中 Topic 的最新数据的数量. 在作业执行期间，jobAnalyzer 使用 StatsReportListener 和 SparkJobInfo 收集每个已完成作业的状态. StreamingListener 定期报告端到端的延迟.

在 Mesos 中，增加了状态矩阵类 *SM*. 另外增加一个强化学习模块 QLearning 以及命令矩阵类 *OM*. 当 Spark 作业的最新状态到达后，*SM* 被更新. QLearning 定期检查变换情况，然后实时调度策略，并将调度结果发送到 *OM* 上. Mesos 中的 Allocated 类定期检查 *OM*，一旦出现新的请求，则预留一定的资源，然后向 Spark 的 SchedulerDriver 发送 ResourceOffer 或者 stopExecutor 消息，以便 Spark 引擎启动新的 Executor 或者结束某个 Executor.

图 6 给出了弹性资源分配实现过程. Spark Engine 定时汇报输出数据流和资源的使用情况到状态矩阵 Status Matrix(图 6 中的步骤①)；QLearning 模块定时检查状态矩阵中数据流的注入速度变化(图 6 中的步骤②)，一旦发现数据注入速度变化，就启动算法 1 中的决策功能，同时将决策的动作 action 写入命令矩阵 Order Matrix(图 6 中的步骤③)；Mesos 资源管理模块定期检查 Order Matrix(图 6 中的步骤④)，一旦发现存在新的命令，就开始为作业预留资源，同时向 Spark Engine 发送资源提供清单 ResourceOffer(图 6 中的步骤⑤)；Spark Engine 为作业启动新的 Executor 或者停止 Executor，同时使用新的资源对流数据进行计算并得到延迟，然后将延迟转化为收益 Reward 发送给 QLearning 模块

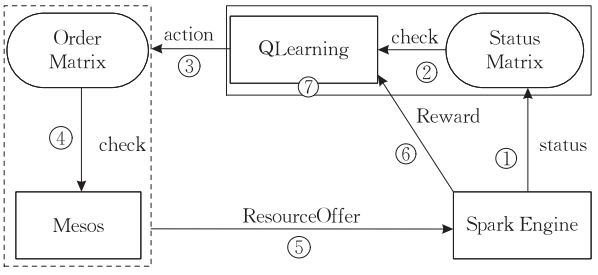


图 6　弹性资源分配实现

(图 6 中的步骤⑥);QLearning 模块根据收益值,更新其决策表 QTable(图 6 中的步骤⑦).

5.2 状态矩阵与命令矩阵的更新

针对某个应用的弹性资源分配,必然涉及到一个上游应用,因为它决定了数据注入速度的收集问题,也涉及当前应用资源弹性扩展或者收缩请求和资源管理器对资源的预留.算法 2 给出了更新数据注入速度监视的过程,算法 3 给出了应用的资源分配过程.

算法 2. 数据注入速度监视过程.

输入:数据产生速度 λ_{new},λ_{old}

输出:*SM* 的更新

过程:

S1. 获取数据产生速度 λ_{new},λ_{old}

S2. FOR (){

S3. $\lambda_{new}=getdata()$

S4. IF($|\lambda_{new}-\lambda_{old}|<\lambda^{-}$)

S5. CONTINUE;

S6. *publish* λ_{new} TO *SM*

S7. }

算法 2 中,S3 定期获得输出数据流的变化,S4 检查数据流变化的大小,当变化范围超过一定的阈值后,S6 将数据发送到状态矩阵.

算法 3. 应用的资源分配过程.

输入:*OM*

输出:*ResourceOffer*

过程:

S1. 初始化参数

S2. 从 *OM* 中得到资源需求 *order*

S3. IF(*Offer*>*order*)

S4. SEND *ResourceOffer* TO operator

S5. ELSE

S6. FAILURE

算法 3 中,当命令矩阵 *OM* 发生变化时,S2 从资源管理器获得最新的资源分配请求 *order*. S3 用来检查系统当前的可用资源 *Offer* 数量是否大于请求的资源数量,如果满足,则执行 S4,向应用程序提供资源列表;否则执行 S6,发送错误信息,表明系统资源不足.

6 系统评价

6.1 评价环境

系统在一个集群上进行实验,集群中包含 16 台 NF5468M5 服务器,作为计算节点,1 台中科曙光服务器 620/420,作为调度节点.每个服务器节点包含 2 颗 Xeon2.1 处理器,每个处理器包含 8 个核,128 GB DDR4 内存,2 块 RTX2080TI GPU 卡,10GB 显存.集群包含 1 台 AS2150G2 磁盘阵列.服务器操作系统为 Ubuntu18.04,CUDA 版本为 10.1.105,采用 C++11 作为编程语言,Mesos 的基础版本为 1.8.这些计算机被组织在 4 个机架内,每个机架包含 4 台计算机.机架内的计算机通过机架交换机连接,各个机架交换机通过级联方式与汇聚交换机连接.

6.2 评价用例

以混合视频流数据处理作为一个查询,将查询分解为目标检测、车牌识别以及行人密度检测三个应用,各个应用之间通过 kafka 来缓存中间结果,其处理流程如图 7 所示.目标检测应用通过网络读取多路视频流,按照设置的处理间隔,将视频流分割成视频流片段,并将片段分解成数据帧.然后使用 YoloV3 模型检测出图片中的目标,并按照类别分别存储在 kafka 中.车牌识别应用使用 HyerLPR 模型,按照 topic 从 kafka 中读取目标图片,进行车牌号的识别.行人统计采用同样的方式,从 kafka 中读取目标图片,根据时间处理间隔和目标出现的次数来计算一段时间内行人的数量.

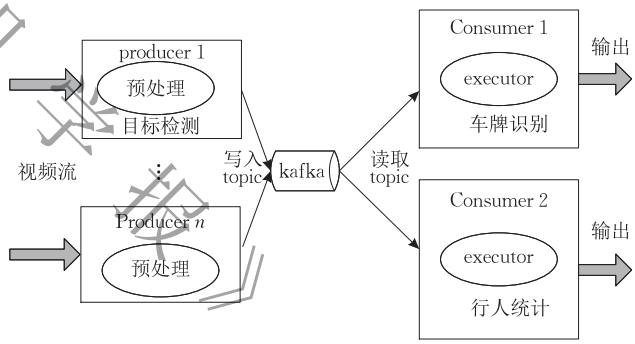


图 7 综合交通视频流的实时检测

6.3 数据注入速度波动的评价

考虑 6.2 节中的视频流检测的数据集①,该数据集包含一段时间内的整个交通视频数据.视频流中的每一个车辆,都要报告其车牌号.车辆的出现情况在每个时间段都有明显的特征(早晚是高峰、晚上数量少),因此需要一个弹性的资源分配功能来负责实时分析生成的数据的系统.

我们将处理间隔设置为 $w=30$ s.为了产生输入速率值 λ_i 的序列,我们在一分钟的间隔内汇总数据集中的事件.对于每一张图片,我们设置处理能力为 $\mu=11.7$,即每秒能够识别的检测目标数量.为了使数据注入速度离散化,我们设置 $\lambda^{-}=5,10,15$ 三种

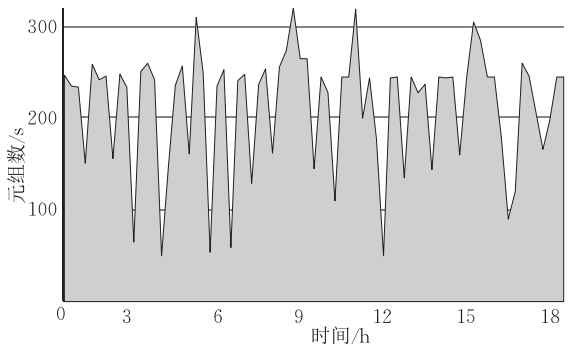
① https://gitlab.com/Dinggy/traffic_video_dataset

情况，λ^- 的单位是目标数量/每秒. 使用较大的 λ^- 值来离散化输入速率，减少了系统状态的数量，从而简化了学习过程. 但是，较大的 λ^- 会使应用程序的资源分配精度降低，从而可能导致较差的策略. 对于每一次学习，其代价由资源分配的代价 c_{res} 和延迟大小的代价 c_{sla} 组成，c_{res} 与 c_{sla} 的和为 1，代价 $cost=x\times c_{res}+y\times c_{sla}$，当智能代理给出的动作是不变化时，$x=0$，当 c_{sla} 在指定的区间时，$y=0$. 表 3 给出了实验时的参数设置.

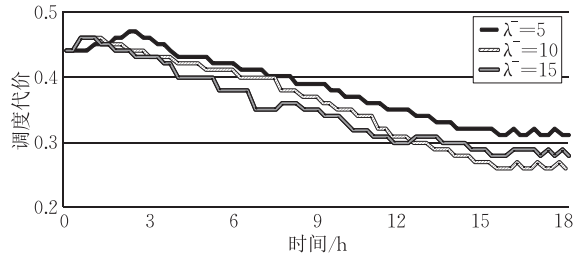
表 3　强化学习参数

编号	参数名	参数值
1	惩罚系数	$\gamma=0.68$
2	学习率	$\theta=0.4$
3	处理间隔	$w=30$ s
4	数据注入速率	$\lambda^-=5,10,15$

图 8(a)给出了数据流注入速度变化，其最大值为每秒 320 个元组，最小值为每秒 50 个元组. 图 8(b)给出了数据注入与强化学习代价的关系图，对于不同的注入速率，为了调整状态的数量，我们使用了三种不同的数据注入速率离散值，即 5、10 和 15. 离散值越小，状态越多，调度中的计算代价大，反之，状态数量少，调度速度快. 从图 8(b)可以看出，最初一段时间，系统需要学习，三种情况都存在较大的延迟，因此，离散值较大的调度方法其代价较小；在后期，系统得到了较好的学习参数后，延迟误差变小，状态数量多时花费的调度开销较大，因此离散值小的代价较大. 但是，由于离散值大时，延迟误差波动较大，所以离散值为 15 的调度代价反而高于离散值为 10 的调度代价. 所以，选择合适的离散值时可以获得了更低的平均成本.



(a) 数据注入速度变化



(b) 注入速度离散值与代价关系

图　8

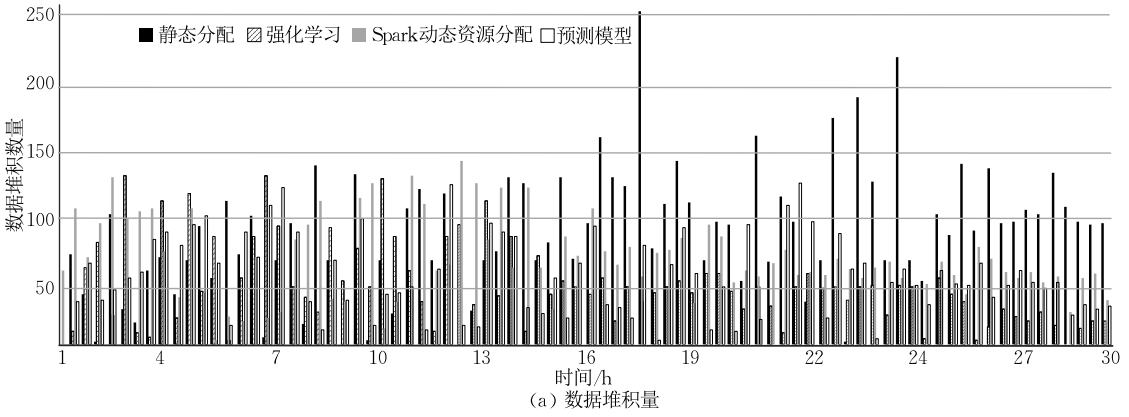
6.4　对比评价

相比于其他的动态资源分配，强化学习方法可以在程序运行期间调整分配策略，不需要专门的训练数据进行训练，更适应流处理系统的实时状况. 针对 Spark 引擎，我们分别使用采取事前策略的静态资源分配方法、预测模型方法，采取事后策略的 Spark 动态资源分配方法、以及采取事中策略的基于强化学习的资源分配方法，对比这几种资源分配方法在数据堆积、资源使用率、延迟时间以及吞吐量方面的变化，其结果如图 9 所示. 其中预测模型根据历史数据预测即将达到的数据. 为了简化计算，我们使用 $t-2$，$t-1$，t 时刻的数据来预测，即 $x=\alpha x_{t-2}+\beta x_{t-1}+\gamma x_t$，其中 $\alpha+\beta+\gamma=1$. 以静态调度时资源多少与数据大小为基准，再根据预测数据等比例计算出资源扩大或者收缩的多少.

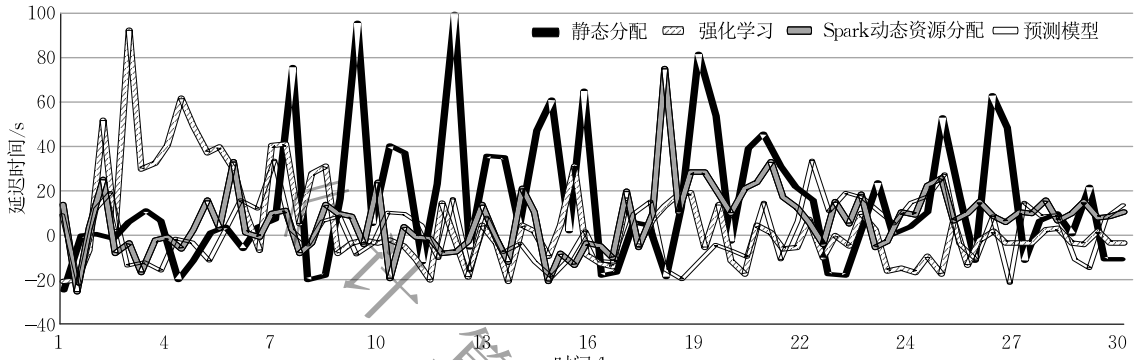
图 9(a)表示了数据在生产者和消费者之间的堆积情况，其横坐标代表时间，纵坐标代表数据堆积的大小，即元组的数量. 从图中可以看出，在整个运行期间，静态资源分配方法的数据堆积一直处于波动作态，数据注入小，则堆积小，数据注入大，则堆积大. Spark 动态资源分配方法中，数据堆积量一直处于较低的位置，变化幅度小于静态资源分配，其根本原因在于动态调整资源，系统能够根据数据大小来增加处理能力，所以波动幅度小. 基于预测模型的资源分配中，数据堆积的变化幅度与 Spark 动态调度相当，部分时刻较好，部分时刻较差，与预测的精度有关. 对于强化学习方法，在 20 h 以前，其数据堆积量的波动非常大，20 h 以后，其波动幅度非常小，低于动态资源分配方法，其原因在于，开始处于学习阶段，算法在不断地修正参数；一定时间后，参数达到最优，系统就能够根据数据注入速度快速地分配计算资源，使得数据堆积减少.

图 9(b)是对延迟时间的评价，其横坐标代表时间，纵坐标代表延迟大小，正数代表超过一个时间间隔的时间，而负数代表负数代表没有延迟，应用程序在低于一个时间间隔内处理了本次注入的数据. 从图中可以看出，在整个运行期间，静态资源分配方法由于不调整计算资源，所以其延迟一直忽高忽低.

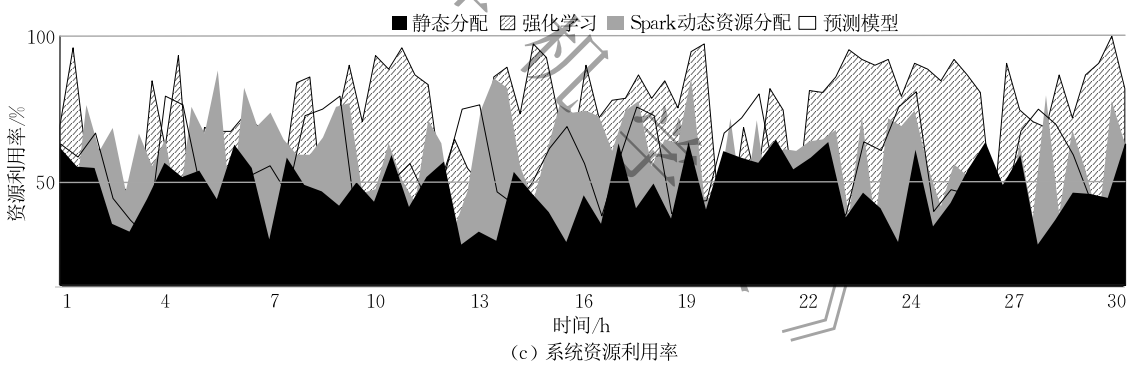
Spark 动态资源分配方法能够进行动态调整资源，所以其延迟较小. 基于预测模型的资源分配中，延迟变化幅度比 Spark 动态资源分配方法稍微小一点，其原因是通过历史数据的叠加，堆积的数据进行



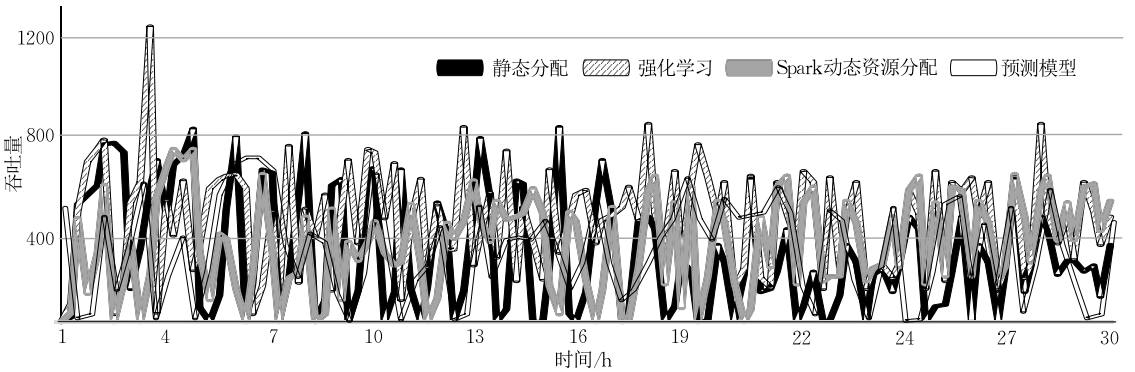
(a) 数据堆积量



(b) 调度延迟时间



(c) 系统资源利用率



(d) 系统吞吐量

图 9 不同资源分配方法对比图

了一定的平均.对于强化学习方法,在 20 h 以前,延迟不稳定,20 h 以后,延迟逐渐稳定,并且低于动态资源分配方法,其原因在于,动态资源分配方法是反应式调整,而强化学习在执行前调整,所以效果较好.通过计算延迟时间的偏差,求得延迟时间的平均值,基于强化学习的平均值为 12.26,基于 Spark 的动态资源分配方法的平均为 15.46,Spark 动态资源分配来说,其延迟平均值较少 14.67%.

图 9(c)、(d)分别是资源使用率和吞吐量的对比，从图中可以看出，强化学习到达一定阶段后，其吞吐率和资源使用率都比较好，通过实验计算，强化学习的平均资源利用率为 73.63%，吞吐量为 382.71，Spark 动态资源分配的平均资源利用率为 60.29%，吞吐量为 336.5. 相对 Spark 动态资源分配来说，其平均资源利用率提高 22%，吞吐量提高 13.7%. 其原因是强化学习能够安排合理的计算资源来处理波动的数据，所以资源使用率和吞吐量较好.

6.5　数据源增加或者减少对延迟的影响

对于数据源的突然增加或者减少，其数据注入变化就非常大，其必然导致资源的不足或者资源的浪费. 图 10 中的正数代表超过一个时间间隔的时间，而负数代表没有延迟，应用程序在低于一个时间间隔内处理了本次注入的数据. 在间隔 5 突然减少 4 个数据源，在间隔 17 突然减少 2 个数据源，在间隔 20 突然增加 4 个数据源. 图 10 给出了 Spark 动态资源分配、基于预测模型的资源分配以及基于强化学习的资源分配对数据源突然变化的延迟情况. 从执行结果看，数据源增加/减少幅度较大时，Spark 动态资源分配和基于预测模型的资源分配需要多个间隔才能够达到注入速度与处理的一致性，即达到稳定状态. 而基于强化学习的资源分配系统一般在 2 个间隔就可以达到稳定状态.

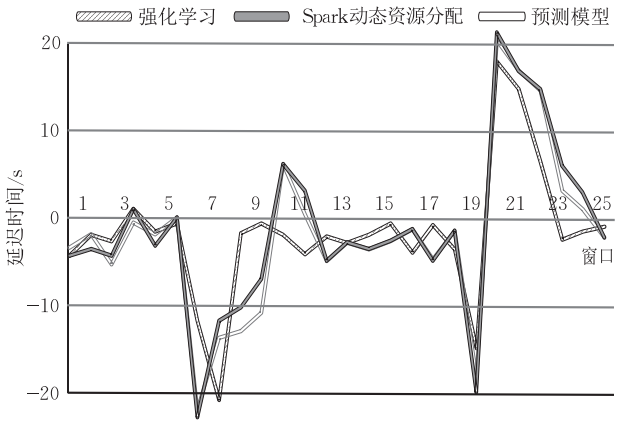


图 10　数据源增加/减少时的延迟

6.6　系统压力测试

系统的框架是集中式方式，生产者和消费者应用不断向资源管理器发送消息更新状态矩阵，QLearning 模块据此进行决策并更新命令矩阵，这些变化信息都需要资源管理器来处理. 当集群规模扩大，多个生产者与多个消费者同时运行时，就会存在大量的状态变化消息. 此时，如果某个生产者的状态发生变化，而资源管理器由于忙无法及时处理，就会导致消费者无法确定增加或减少资源的数量，从而引起消费者任务的延迟. 因此，资源管理是调度的瓶颈，需要对资源管理进行压力测试. 目前，Spark 设置最小资源限制为 1 个 GPU，而现有的测试环境为 32 个 GPU，所以最多运行 32 个 Spark 处理引擎，这样的话，资源管理器能够快速处理. 为此，论文启动了大量模拟应用，模拟生产者和消费者不断向资源管理器发送状态变化，资源管理器收到状态变化后，使用强化学习方法给出新的资源分配策略. 模拟应用按照一定的心跳间隔(1s)向资源管理发送不同的消息数据.

在资源管理器的压力测试过程中，通过资源管理器的处理效率来体现系统的调度性能. 资源管理器的处理效率定义为 $h=\min(m/n)$，其中 n 为集群资源管理器在心跳时间间隔收到的状态变化事件的数量，m 为资源管理器在心跳时间间隔内处理的节点资源更新事件数量.

经过多次测试，当生产者和消费者程序大约为 2050 个时，在一个心跳间隔内，资源管理器能够处理完全部的状态变化消息；当生产者和消费者程序超过 2050 个时，在一个时心跳隔内资源管理器无法处理完全部请求，即 $h<1$，说明部分应用的资源分配出现延迟. 因此，在当前实验条件下系统的最佳处理能力为同时运行 2050 个应用.

7　结　论

本文研究了基于事件驱动架构的分布式流处理应用的弹性资源分配问题. 与传统的分布式流处理的弹性资源分配策略相比，我们分析了多相互依赖的 DSP 应用之间的数据流关系，并使用强化学习算法进行弹性资源分配. 数值评估表明，与文献中经常采用的算法相比，我们所提出的弹性资源分配策略能够使分布式流处理应用的状态得到更快的收敛，系统整体性能也能得到较好的提升.

致　谢　诚挚感谢评阅老师对论文提出的改进意见！

参　考　文　献

[1] Isah H, Abughofa T, Mahfuz S, et al. A survey of distributed data stream processing frameworks. IEEE Access, 2019, 7: 154300-154316

[2] Noghabi S A, Paramasivam K, Pan Y, et al. Samza: Stateful scalable stream processing at LinkedIn. Proceedings of the VLDB Endowment, 2017, 10(12): 1634-1645

[3] Apache K. A high-throughput distributed messaging system. http://kafka.apache.org/design.html, 2013

[4] de Assuncao M D, da Silva Veith A, Buyya R. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. Journal of Network and Computer Applications, 2018, 103: 1-17

[5] Nardelli M, Russo G R, Cardellini V, et al. A multi-level elasticity framework for distributed data stream processing// Proceedings of the European Conference on Parallel Processing. Cham, Swiss: Springer, 2018: 53-64

[6] Carbone P, Fragkoulis M, Kalavri V, et al. Beyond analytics: The evolution of stream processing systems//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Portland, USA, 2020: 2651-2658

[7] Carbone P, Ewen S, Fóra G, et al. State management in apache Flink: Consistent stateful distributed stream processing. Proceedings of the VLDB Endowment, 2017, 10(12): 1718-1729

[8] Castro Fernandez R, Migliavacca M, Kalyvianaki E, et al. Integrating scale out and fault tolerance in stream processing using operator state management//Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. New York, USA, 2013: 725-736

[9] Carbone P, Katsifodimos A, Ewen S, et al. Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2015, 36(4): 25-32

[10] Botan I, Cho Y, Derakhshan R, et al. A demonstration of the MaxStream federated stream processing system// Proceedings of the 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). Long Beach, USA, 2010: 1093-1096

[11] Liu X, Buyya R. Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions. ACM Computing Surveys (CSUR), 2020, 53(3): 1-41

[12] Gedik B, Schneider S, Hirzel M, et al. Elastic scaling for data stream processing. IEEE Transactions on Parallel and Distributed Systems, 2013, 25(6): 1447-1463

[13] Cardellini V, Presti F L, Nardelli M, et al. Decentralized self-adaptation for elastic data stream processing. Future Generation Computer Systems, 2018, 87: 171-185

[14] Das T, Zhong Y, Stoica I, et al. Adaptive stream processing using dynamic batch sizing//Proceedings of the ACM Symposium on Cloud Computing. Seattle, USA, 2014: 1-13

[15] De Matteis T, Mencagli G. Proactive elasticity and energy awareness in data stream processing. Journal of Systems and Software, 2017, 127: 302-319

[16] Cardellini V, Lo Presti F, Nardelli M, et al. Optimal operator deployment and replication for elastic distributed data stream processing. Concurrency and Computation: Practice and Experience, 2018, 30(9): e4334

[17] Cheng D, Chen Y, Zhou X, et al. Adaptive scheduling of parallel jobs in spark streaming//Proceedings of the IEEE INFOCOM 2017-IEEE Conference on Computer Communications. Atlanta, USA, 2017: 1-9

[18] Heinze T, Roediger L, Meister A, et al. Online parameter optimization for elastic data stream processing//Proceedings of the 6th ACM Symposium on Cloud Computing. Kohala Coast, USA, 2015: 276-287

[19] De Matteis T, Mencagli G. Elastic scaling for distributed latency-sensitive data stream operators//Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). St. Petersburg, Russia, 2017: 61-68

[20] Lohrmann B, Janacik P, Kao O. Elastic stream processing with latency guarantees//Proceedings of the IEEE 35th International Conference on Distributed Computing Systems (ICDCS). Columbus, USA, 2015: 399-410

[21] Mencagli G, Torquati M, Danelutto M. Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams. Future Generation Computer Systems, 2018, 79: 862-877

[22] Liu X, Dastjerdi A V, Calheiros R N, et al. A stepwise auto-profiling method for performance optimization of streaming applications. ACM Transactions on Autonomous and Adaptive Systems (TAAS), 2017, 12(4): 1-33

[23] Jamshidi P, Pahl C, Mendonça N C. Managing uncertainty in autonomic cloud elasticity controllers. IEEE Cloud Computing, 2016, 3(3): 50-60

[24] Tesauro G, Jong N K, Das R, et al. On the use of hybrid reinforcement learning for autonomic resource allocation. Cluster Computing, 2007, 10(3): 287-299

[25] Arabnejad H, Pahl C, Jamshidi P, et al. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling// Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). Madrid, Spain, 2017: 64-73

[26] Barrett E, Howley E, Duggan J. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. Concurrency and Computation: Practice and Experience, 2013, 25(12): 1656-1674

[27] Gedik B, Özsema H G, Öztürk Ö. Pipelined fission for stream programs with dynamic selectivity and partitioned state. Journal of Parallel and Distributed Computing, 2016, 96: 106-120

[28] Zeuch S, Monte B D, Karimov J, et al. Analyzing efficient stream processing on modern hardware. Proceedings of the VLDB Endowment, 2019, 12(5): 516-530

[29] Akidau T, Bradshaw R, Chambers C, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proceedings of the VLDB Endowment, 2015, 8 (12): 1792-1803

[30] Armbrust M, Das T, Torres J, et al. Structured streaming: A declarative api for real-time applications in apache spark// Proceedings of the 2018 International Conference on Management of Data. Houston, USA, 2018: 601-613

[31] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for fine-grained resourcesharing in the data center// Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation. Berkeley, USA, 2011: 295-308

[32] Röger H, Mayer R. A comprehensive survey on parallelization and elasticity in stream processing. ACM Computing Surveys (CSUR), 2019, 52(2): 1-37

[33] Gulisano V, Jimenez-Peris R, Patino-Martinez M, et al. StreamCloud: An elastic and scalable data streaming system. IEEE Transactions on Parallel and Distributed Systems, 2012, 23(12): 2351-2365

[34] Castro Fernandez R, Migliavacca M, Kalyvianaki E, et al. Integrating scale out and fault tolerance in stream processing using operator state management//Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. New York, USA, 2013: 725-736

[35] Cardellini V, Nardelli M, Luzi D. Elastic stateful stream processing in storm//Proceedings of the 2016 International Conference on High Performance Computing & Simulation (HPCS). Innsbruck, Austria, 2016: 583-590

[36] Heinze T, Pappalardo V, Jerzak Z, et al. Auto-scaling techniques for elastic data stream processing//Proceedings of the IEEE 30th International Conference on Data Engineering Workshops. Chicago, USA, 2014: 296-302

[37] Lombardi F, Aniello L, Bonomi S, et al. Elastic symbiotic scaling of operators and resources in stream processing systems. IEEE Transactions on Parallel and Distributed Systems, 2017, 29(3): 572-585

[38] Russo Russo G, Cardellini V, Presti F L. Reinforcement learning based policies for elastic stream processing on heterogeneous resources//Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems. Darmstadt, Germany, 2019: 31-42

[39] Russo Russo G, Nardelli M, Cardellini V, et al. Multi-level elasticity for wide-area data streaming systems: A reinforcement learning approach. Algorithms, 2018, 11(9): 134

[40] Ni X, Li J, Yu M, et al. Generalizable resource allocation in stream processing via deep reinforcement learning. Proceedings of the AAAI Conference on Artificial Intelligence, 2020, 34(1): 857-864

[41] Li Li-Na, Wei Xiao-Hui, Li Xiang, Wang Xing-Wang. Burstiness-aware elastic resource allocation in stream data processing. Chinese Journal of Computers, 2018, 41(10): 2193-2208(in Chinese)
(李丽娜，魏晓辉，李翔，王兴旺. 流数据处理中负载突发感知的弹性资源分配. 计算机学报，2018，41(10)：2193-2208)

[42] Zaharia M, Das T, Li H, et al. Discretized streams: Fault-tolerant streaming computation at scale//Proceedings of the 24th ACM Symposium on Operating Systems Principles. Pennsylvania, USA, 2013: 423-438

TANG Xiao-Chun, Ph. D., associate professor. His research interests include big data computing, graph data management and cluster resource management.

ZHANG Ke, M. S. candidate. His research interest is big data computing.

ZHAO Quan, M. S. candidate. His research interest is big data computing.

LI Zhan-Huai, Ph. D., professor. His research interests include ocean base management and big data computing.

Background

Data stream processing systems were born nearly a decade ago, due to the need for low-latency processing of large volumes of highly dynamic, time-sensitive continuous streams of data from sensor-based monitoring devices and alike. Some application that processes stream of events does not just perform trivial record-at-a-time transformations needs to be ability to store and access intermediate data. But "Store-and-Pull" model of traditional data processing systems was simply not suitable for the high performance needs of the such applications, where data was much more dynamic than queries and had to be processed on the fly. Consequently, this change of roles between data and queries led to an upside down architecture, and thus, to Stream Processing Engines (SPEs). For SPEs to realize their full potential and to see stronger adoption, first and foremost, they require critical support for integration across streaming data sources and SPE-SPE.

In order to meet the above data processing requirements, we distributed data stream processing applications based on event-driven architecture that ingest event streams and process the events with application-specific business logic. Depending on the business logic, an application can trigger actions such as sending an alert or an email or write events to an outgoing event stream to be consumed by another applications.

Applications based on event-driven architecture are an evolution of micro-services. One application emits its output to an event log and another application consumes the events the other application emitted. The event log decouples senders and receivers and provides asynchronous, non-blocking event transfer. In order to assure the consistence of processing procedure between producers and consumers, we establishing a state matrix and a command matrix to monitor the status changes of upstream and downstream applications. By this status, an elastic resource allocation method based on reinforcement learning is designed to control the computing resources of all applications. So, it is not only improving the system resources utilization rate, but also reducing the delay of data processing times.

This work is supported by the National Key Research & Development Program of China (Project No. 2018YFB1003400).