软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(1):430−454 [doi: 10.13328/j.cnki.jos.006917]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

支持实时流计算应用的关键技术研究进展*

徐志榛[1,2], 徐 辰[1,2,3], 丁光耀[1,2], 陈梓浩[1,2], 周傲英[1,2]

[1](华东师范大学 数据科学与工程学院, 上海 200062)

[2](上海市大数据管理系统工程研究中心, 上海 200062)

[3](广西可信软件重点实验室 (桂林电子科技大学), 广西 桂林 541004)

通信作者: 徐辰, E-mail: cxu@dase.ecnu.edu.cn

摘 要: 信息系统在进行知识的挖掘和管理时, 需要处理各种形式的数据, 流数据便是其中之一. 流数据具有数据规模大、产生速度快且蕴含的知识具有较强时效性等特点, 因而发展支持实时处理应用的流计算技术对于信息系统的知识管理十分重要. 流计算系统可以追溯到 29 世纪 90 年代, 至今已经经历了长足的发展. 然而, 当前多样化的知识管理需求和新一代的硬件架构为流计算系统带来了全新的挑战和机遇, 催生出了一系列流计算领域的技术研究. 首先介绍流计算系统的基本需求以及发展脉络, 再按照编程接口、执行计划、资源调度和故障容错 4 个层次分别分析流计算系统领域的相关技术; 最后, 展望流计算技术在未来可能的研究方向和发展趋势.

关键词: 实时处理; 流计算; 数据处理系统

中图法分类号: TP311

中文引用格式: 徐志榛, 徐辰, 丁光耀, 陈梓浩, 周傲英. 支持实时流计算应用的关键技术研究进展. 软件学报, 2024, 35(1): 430–454. http://www.jos.org.cn/1000-9825/6917.htm

英文引用格式: Xu ZZ, Xu C, Ding GY, Chen ZH, Zhou AY. Research Progress on Key Technologies Towards Real-time Stream Processing Applications. Ruan Jian Xue Bao/Journal of Software, 2024, 35(1): 430–454 (in Chinese). http://www.jos.org.cn/1000-9825/6917.htm

Research Progress on Key Technologies Towards Real-time Stream Processing Applications

XU Zhi-Zhen[1,2], XU Chen[1,2,3], DING Guang-Yao[1,2], CHEN Zi-Hao[1,2], ZHOU Ao-Ying[1,2]

[1](School of Data Science and Engineering, East China Normal University, Shanghai 200062, China)

[2](Shanghai Engineering Research Center of Big Data Management, Shanghai 200062, China)

[3](Guangxi Key Laboratory of Trusted Software (Guilin University of Electronic Technology), Guilin 541004, China)

Abstract: In order to perform knowledge mining and management, information systems need to process various forms of data, including stream data. Stream data have the characteristics of large data scale, fast generation speed, and strong timeliness of the knowledge contained in them. Therefore, it is very important for knowledge management of information systems to develop stream processing technology that supports real-time stream processing applications. Stream processing systems (SPSs) can be traced back to the 1990s, and they have undergone significant development since then. However, current diverse knowledge management needs and the new generation of hardware architectures have brought new challenges and opportunities for SPSs, and a series of technical research on stream processing ensues. This study introduces the basic requirements and development history of SPSs and then analyzes relevant technologies in the SPS field in terms of four aspects: programming interface, execution plan, resource scheduling, and fault tolerance. Finally, this study predicts the research directions and development trends of stream processing technology in the future.

Key words: real-time processing; stream processing; data processing system

---

* 基金项目: 国家自然科学基金 (61902128); 广西可信软件重点实验室研究课题
收稿时间: 2022-08-15; 修改时间: 2022-10-05, 2022-12-05; 采用时间: 2023-02-06; jos 在线出版时间: 2023-07-12
CNKI 网络首发时间: 2023-07-13

流数据是指由多个数据源持续生成的数据, 其单条数据的大小一般较小, 但可能同时产生大量数据. 典型的流数据包括传感器数据、日志文件、网购数据、行为记录数据等. 由于流数据通常包含宝贵的知识, 开发者通常希望编写程序以挖掘并处理流数据, 从中获取需要的信息[1,2]. 然而, 流数据包含的知识具有较强的时效性, 如果不能对流数据进行及时处理, 这些知识的价值可能会严重折损[3]. 例如, 在金融交易中产生的流数据包含的最新成交价格, 及时获取这些价格的波动有助于自动交易软件做出正确的决策[4]. 然而, 如果信息系统不能及时处理流数据, 那么其获取到的成交价格就会过时, 导致这些知识无法帮助自动交易软件做出正确决策, 造成知识的价值贬值. 将流计算系统应用于流数据的处理可以以较低延迟处理源源不断产生的数据, 并从中及时获取有价值的信息.

流计算系统漫长的发展过程衍生出多种多样的流计算技术, 其发展历史及其对应的关键技术如图 1 所示. 最早的流计算系统可以追溯到 20 世纪 90 年代. 通过对数据库执行引擎和数据库查询语言的拓展, Tapestry[5]成为第 1 个支持流数据处理的系统. 此后, 诸如 GigaScope[6]、Aurora[7]、Borealis 等系统都通过拓展数据库执行引擎或数据库查询语言的方式实现了对流计算的支持. 上述系统可以看作第 1 代流计算系统, 其主要技术特点为实现了处理流数据所必需的功能, 如窗口聚合、时间语义等. 第 2 代流计算系统源于 MapReduce[8]后出现的大数据处理系统发展浪潮. 这期间出现了诸如 Storm[9]和 Spark Streaming[10]等流计算系统. 这些系统借鉴 MapReduce 中并行化处理思想, 通过横向拓展, 即向流计算系统中引入更多节点, 提升流计算系统的处理能力以应对规模不断增长的流数据[11]. 截至今天, 新一代流计算系统的发展出现了两个不同的方向. 第 1 类是以 Flink[12]为代表的, 采用基于 JVM 的执行引擎, 拥有良好可拓展性的流计算系统. 这类系统在保持了可拓展性的基础上改进了编程模型以支持更加丰富的语义表达, 使开发者可以灵活处理乱序数据等问题. 同时, 这类系统设计或优化了故障容错机制, 从而提升了系统在分布式流计算场景中的容错性能. 由于具有完善的功能支持和丰富的第三方库, 这一类系统在如今的学术界和工业界得到了广泛的应用. 另一类是以 StreamBox[13]为代表的高性能流计算系统. 一些研究人员注意到 JVM 不能有效利用硬件资源, 限制了可拓展流计算系统的性能, 因而采用 C++等语言实现了一类高性能流计算系统. 这类系统通过对硬件资源的充分利用, 大幅提升流计算系统的处理能力, 从而在单机情况下取得了远超 Flink、Storm 等系统的性能.

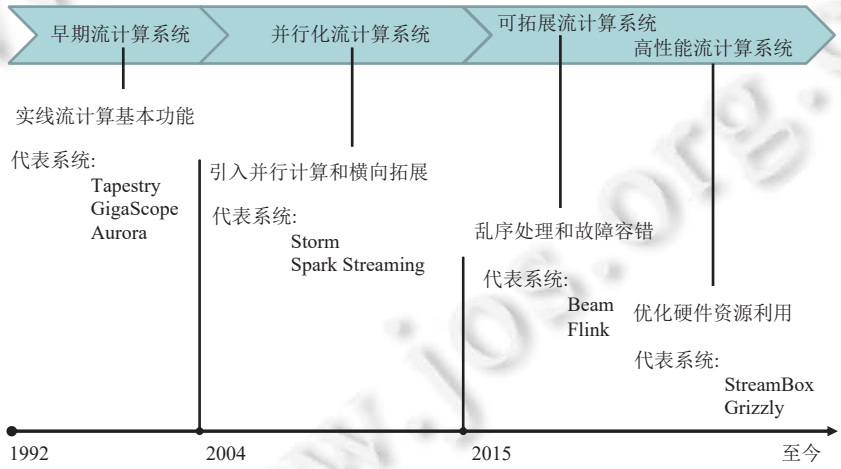

图 1 流计算关键技术发展历史

本文第 1 节从一个典型的流计算应用——广告投放应用开始, 举例说明流计算应用从流数据中获取有价值的信息的过程, 并分析在此过程中面临的挑战, 并据此将流计算系统中的应用技术分层. 在第 2–5 节中, 本文将关注现有的研究工作, 分别从编程语义、执行计划、资源调度和故障容错 4 个层面具体介绍流计算系统. 本文在第 6 节中讨论流计算系统的发展方向和未来趋势, 并在第 7 节总结全文.

1 流计算技术概述

1.1 研究挑战

流计算系统的开发需要考虑各个方面的设计问题, 例如系统架构、开发接口、数据组织和底层硬件等. 尽管这些问题本身相对独立, 它们的共同目的都是为了解决流计算带来的挑战. 为了解决流计算应用工作流程中存在的挑战, 研究人员和开发人员针对流计算相关的各类技术进行了改进和创新, 并将之应用到流计算系统中. 这些技术不仅使得流计算应用能够适应于流数据的处理方式, 还能够提升流计算应用的性能. 因此, 本节将从流计算应用的工作流程出发, 以流计算应用中常见的广告投放应用为例, 简要介绍流计算应用的需求和挑战.

在编程接口方面, 如何利用编程接口降低管理流数据的复杂程度. 如图 2 所示, 开发者需要利用流计算系统提供的编程接口表达他们的应用逻辑. 应用逻辑中必须包含对流数据独特性质的处理方法. 例如, 广告投放应用需要能够生成并识别广告点击事件的发生时间, 才能够对其进行正确处理; 在对广告点击事件进行聚合的过程中, 需要定义窗口才能对持续产生的数据进行聚合; 由多个终端产生的点击事件进入系统的顺序可能会和它们的产生顺序不同, 流计算应用需要识别这些乱序数据的处理进度来完成窗口聚合操作. 虽然开发者可以在他们的应用逻辑中加入对以上功能的支持, 但这会导致开发流计算应用的复杂程度增加. 另一方面, 这种方式增加了流数据的管理和应用逻辑代码之间的耦合, 加大了故障或错误的可能性, 损害了系统的稳健性. 因此, 流计算系统需要在编程接口中提供流数据的管理功能, 避免开发者在应用逻辑中进行流数据管理, 减少其带来的编程复杂度和可能的故障.

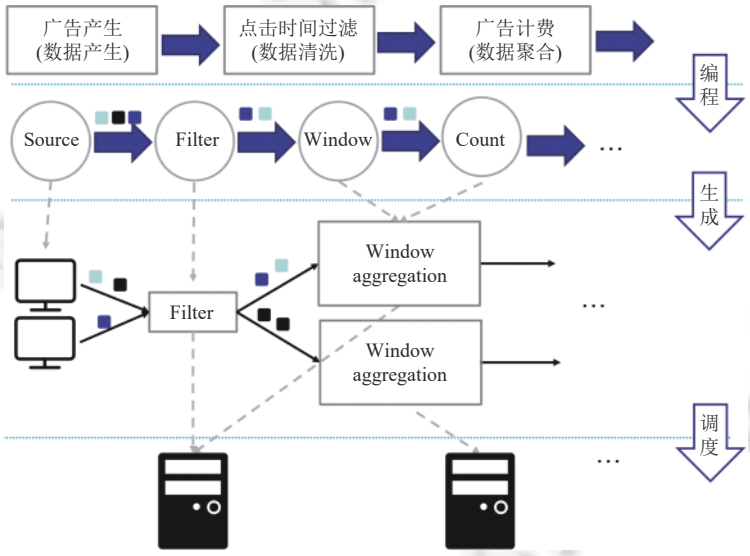

图 2 流计算广告投放应用

在执行计划方面, 如何生成适用于流计算场景的高效执行计划. 如图 2 所示, 在开发者将他们编写的流计算程序提交之后, 流计算系统负责根据只包含处理逻辑的程序生成执行计划, 从而使得流计算应用可以运行在各种硬件环境中. 在静态的计划生成过程中, 流计算系统需要从候选执行计划中选择效率最优的执行计划, 由于执行计划的可选空间巨大, 这一过程通常十分困难. 另一方面, 当流计算应用的输入数据发生动态变化时, 最优的执行计划也可能随之改变. 因此, 流计算系统需要自适应调整执行计划, 来保证在面对不同的输入数据时流计算应用始终保持高效.

在资源调度方面, 如何提供资源节省和延迟敏感的资源调度. 在生成执行计划之后, 流计算系统需要为执行计划中的任务分配资源, 以保证流计算任务的正常运行. 相比于传统资源调度框架, 流计算应用的特性产生了新的资源调度需求. 一方面, 由于流计算应用中各个算子的计算负载各不相同, 因此也产生了不同的资源需求. 传统的以

作业为单位的调度框架会产生大量的资源浪费, 因此流计算系统的资源调度应更精确地分配资源, 从而降低资源浪费. 另一方面, 流计算应用相较批处理应用而言通常对于延迟更为敏感, 而传统的以提高吞吐量为目标的策略不仅无法通过资源调度降低延迟, 还有可能在资源调度过程中增加延迟, 导致违反流计算应用的延迟约束. 因此流计算系统应当实现延迟敏感的资源调度, 从而使资源调度能够降低流计算应用的延迟.

在故障容错方面, 如何在保证正常运行性能的情况下加速故障恢复. 由于软件崩溃或硬件故障等原因, 流计算系统在运行过程中可能遭遇故障, 这些故障在分布式或异构环境中往往更为频繁. 流计算系统需要提供故障容错机制在遭遇故障后恢复流计算应用的运行. 然而, 实现适用于流计算系统的故障容错机制需要面对两点挑战: 第一, 流计算系统对延迟更为敏感, 因此需要快速地对遭遇故障的流计算任务进行恢复. 第二, 流计算系统存在有状态算子, 因此需要正确恢复状态至故障发生前, 否则可能会导致计算结果错误. 为了加速流计算系统的故障恢复过程, 通常需要在系统正常运行时采用容错技术. 但这些容错技术在系统正常运行时产生了额外开销, 降低了系统的正常运行性能. 因此, 流计算系统应当在减少对正常运行性能影响的前提下, 加速故障恢复过程, 并保证状态的一致性.

1.2 相关技术分类

为了解决上述流计算应用在各个方面带来的挑战, 从业者和研究人员提出了一系列关键技术. 因此, 依据前述流计算应用面对主要挑战的类别, 本节将流计算系统的相关技术分层归纳为了 4 部分, 即编程接口、执行计划、资源调度和故障容错, 分类情况如图 3 所示.

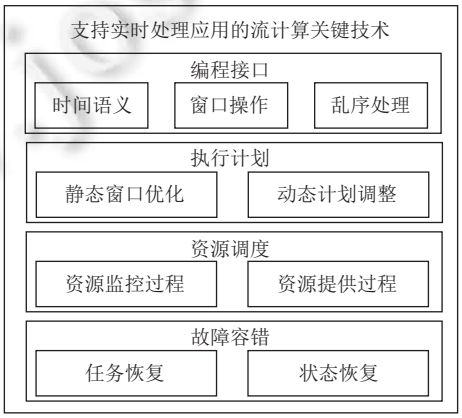

图 3 流计算应用相关技术分类

1.2.1 编程接口

流计算系统在 3 个方面提供了适用于流数据处理的编程接口. 首先, 流计算系统在编程接口中提供了时间语义的支持. 早期的时间语义接口通常将时间作为某条数据的一个属性, 在数据产生或进入系统时进行标注. 目前主流的时间语义接口则通过划分事件时间、摄入时间和处理时间, 允许开发者在不同情境下使用多样化的时间语义. 其次, 流计算系统在编程接口中提供了窗口聚合的支持. 早期的流计算系统用系统内置的窗口代替用户手动实现的窗口, 减少了应用窗口操作的复杂度. 目前的主流系统则丰富了支持的窗口种类, 允许开发者利用编程接口表达丰富的窗口语义. 最后, 流计算系统在编程接口中提供了乱序处理的支持. 早期部分系统采用缓冲区将乱序数据转化为有序数据. 目前更多系统采用基于标点的乱序处理方法, 增加了开发者处理乱序数据时的灵活性.

1.2.2 执行计划

流计算系统在生成和调整执行计划的过程中, 主要采用了两个方面的技术进行优化. 一方面, 在输入数据特征基本保持静态的情况下, 流计算应用的执行计划可以保持不变. 此时流计算系统需要对执行计划中计算开销较大

的窗口聚合操作进行优化. 流计算系统应用预聚合、近似计算、溢写内存数据等技术, 优化了窗口聚合操作的内存占用和触发延迟. 另一方面, 在输入数据特征动态改变的情况下, 流计算系统需要自适应调整执行计划来保证执行效率. 在这种情况下, 流计算系统需要应用动态数据分区来应对输入数据键分布的变化, 并通过动态计划更新来应对输入数据其他统计特征的变化.

1.2.3 资源调度

为了实现资源节省和延迟敏感的资源调度, 流计算系统可以从两方面优化了资源调度过程. 1) 改进资源监控过程. 流计算系统用细粒度的资源监控方式可以更精确地识别流计算系统中的资源不足或浪费, 而延迟敏感的资源监控方式可以更及时地识别流计算系统中资源不足带来的延迟上升, 因而改进资源提供过程可以更准确并及时地触发资源调度过程. 2) 改进资源提供过程. 对于增加节点的资源提供方法, 流计算系统通过为指定任务分配节点的方式减少资源浪费, 并通过预先加入节点池的方式减少提供过程的延迟. 为了更精确和更及时的资源提供, 流计算系统还可以采用向节点内增加资源或调整线程优先级的方式进行资源提供.

1.2.4 故障容错

故障容错技术主要包含任务恢复和状态备份恢复两部分. 任务恢复主要考虑运行时容错和故障时恢复两个方面, 可选的技术包括主动容错、被动容错和部分被动容错等, 其中被动容错对正常运行时性能影响最小, 主动容错则达到了最快的恢复时间, 部分被动容错则中和了二者的优缺点. 状态备份恢复则分为检查点的备份粒度和检查点的备份位置两方面. 相比于局部独立的检查点, 全局统一的检查点可以提升一致性保证, 但牺牲了备份和恢复的速度. 检查点可以选择集群外部或集群内部; 前者有着更好的一致性保障, 而后者因为避免了跨集群网络通讯开销而实现了更优的性能.

2 编程语义

开发者通常通过流计算系统提供的编程接口编写流计算应用程序. 由于流数据独特的特征, 流计算系统需要提供一些特殊的编程语义, 以方便开发者对流数据进行处理. 流数据为流计算系统在编程语义方面带来了 3 点挑战: 1) 流数据的产生和处理时间和其含义密切相关. 因此, 流计算系统需要提供时间语义的支持, 以供开发者对产生或处理时间不同的流数据进行多样化的操作. 2) 流数据具有无界性, 即流数据始终处于持续变更、连续追加的状态, 因此其数据的范围没有边界, 开发者因而无法直接对其应用静态数据的聚合操作. 流计算系统需要提供窗口语义, 使得开发者可以根据条件对部分流数据进行处理. 3) 流数据在生产环境中通常是乱序的, 即其产生的顺序和到达流计算系统的顺序并不完全相同. 这使得流计算系统难以确认数据的完整性, 为窗口聚合等操作带来了额外的困难. 因此流计算系统应当提供完整性语义, 方便开发者确认所需的数据是否已经完全到达, 从而做出相应的响应. 本节将从时间语义、窗口操作和乱序处理 3 个方面, 介绍流计算系统如何提供编程语义, 以供开发者开发针对流数据的高效应用.

2.1 时间语义

由于流数据是在一段时间内连续产生的, 因此其对应的时间往往具有重要意义; 同时流计算系统也需要依据当前的系统时间来对流数据进行合适的处理. 因此, 支持时间语义是流计算系统的基本功能之一.

早期的流计算系统通过在流数据中加入时间戳字段时间语义的相关处理. 例如, Tapestry[5]作为一个拓展关系型数据库来实现的早期流计算系统, 通过为所有表加入了一列时间戳属性来实现了对于时间语义的支持. Tapestry 在流数据进入系统时按照系统时间为其生成时间戳, 因此这个时间戳反映的是系统获取到该条数据的时间, 即摄入时间; 同时, Tapestry 允许用户通过调用操作系统接口的方式获取当前的处理时间. Aurora[7]同样为每条流数据增加了一个时间戳字段, 并在系统获取到数据时为其分配当前的时间戳; 更进一步, Aurora 将流数据的时间戳向流计算系统下游传递, 使得流计算系统所有算子都可以正确获取该条数据进入系统的时间. GigaScope[6]注意到了流计算任务中数据生成时间, 即事件时间的重要性, 因而在系统中提供了对于事件时间的支持. 向流数据中加入时间戳的解决方案可以在一定程度上满足流计算应用关于时间语义的需求. 但由于时间戳的生成和处理仍依赖

于开发者的手动管理, 这种方式增加了开发者的开发成本. 同时, 这种方式也增加了开发过程中人为失误对系统造成的影响, 降低了流计算系统的稳健性.

现代的流计算系统基本沿袭了这些时间语义的支持. Google 提出的 Dataflow 流计算处理模型[14], 进一步规范了事件时间和处理时间的使用, 其中事件时间被用于决定数据的处理方式, 而处理时间则被用于决定计算的触发时机. 当下得到广泛运用的流计算系统 Flink 则明确区分了 3 种时间语义, 即事件时间、摄入时间和处理时间. 除了事件时间在输入系统前就需要被定义外, Flink 系统会自动为每一条数据分配摄入时间和处理时间, 开发者可以灵活地运用这些时间语义以表达其需要的计算逻辑.

2.2 窗口操作

由于流数据是持续生成并进入流计算系统中的, 因而流数据具有一个重要的特性: 无界性. 也就是说, 理论上流数据是没有边界的, 会源源不断地进入流计算系统. 所以流计算系统无法对流数据直接应用静态数据的聚合操作, 这是由于流计算系统无法等到所有数据都进入系统后再进行聚合. 因此, 流计算系统需要一种语义对满足某些条件的数据进行聚合操作, 即窗口语义. 流计算系统提供的窗口语义允许开发者将流数据按照其事件时间、处理时间等特征分配到不同的窗口中, 并对同一个窗口内的流数据应用聚合操作; 窗口语义是流计算系统表达处理无界数据方式的关键语义, 也是流计算系统内最重要和最频繁的语义之一.

早期的基于关系型数据库实现的流计算系统[5]并未显式地提供窗口语义. 例如在 Tapestry[5]中 , 用户需要通过在 SQL 语句中加入关于时间的筛选条件来实现类似窗口的语义. 一方面, 这样的方式将实现窗口语义的工作交给了开发者, 从而加大了开发者编写程序的复杂度, 增加了潜在的错误和故障风险, 降低了应用的稳健型; 另一方面, 流计算系统无法区分用户在 SQL 语句中加入的时间筛选条件, 从而使系统丧失了为窗口操作进行特殊优化的可能性, 降低了流计算应用的性能 (例如, 流计算系统在触发窗口计算时需要扫描整张表以筛选出时间符合条件的数据).

为解决开发者手动实现窗口语义带来的问题, 一些流计算系统开始提供由流计算系统实现的窗口操作接口. 例如, Aurora[7]提供了窗口聚合的接口, 开发者可以借助该接口, 以指定窗口间隔和窗口长度, 对流数据应用自定义的聚合函数. 斯坦福大学的 Arasu 等人在其实现的流计算系统 STREAM[15]及流计算语言 CQL[16]中进一步拓展了窗口聚合的语义. 除去基于时间的窗口, CQL 还支持基于元组的窗口和基于值的窗口; CQL 允许用户可以通过拓展窗口定义来使用时间戳、元组序号或其他值来划分窗口, 使用户可以表达其丰富的流计算语义. 虽然上述工作通过提供内置的窗口操作简化了开发者实现窗口的过程, 并有助于流计算系统对其的优化, 但却牺牲了编程语义的丰富灵活. 一方面, 上述工作对窗口类型的支持不足, 尚未支持会话窗口等语义. 另一方面, 上述工作对非对齐窗口的支持不足. 由于一个窗口的长度和间隔是确定的, 具有不同键的数据必须先被分配同一个窗口中, 再按键进行处理. 然而实际场景中, 具有不同键的数据对窗口起始时间、长度和间隔等可能存在不同需求, 上述的流计算系统内置的窗口操作并不能为这种场景提供支持.

目前流行的流计算系统针对上述窗口语义的问题, 主要进行了两点改进. 以现代流计算系统广为采用的 Dataflow 编程模型[14]为例: 首先, Dataflow 编程模型全面定义了流计算系统中可能用到的基础窗口. 依据数据时间与窗口划分的关系, Dataflow 模型将流计算系统中的窗口操作分为了 3 种类型: 与数据时间戳无关的窗口, 即固定窗口; 与本条数据时间戳相关的窗口, 即滑动窗口; 与本条以及前一条数据的时间戳有关的窗口, 即会话窗口. 会话窗口的引入进一步提升了流计算系统中窗口语义表达的丰富性. 其次, 通过分配-合并的方法, Dataflow 模型很好地支持了非对齐的窗口. 图 4 是一个非对齐窗口中的典型例子: 会话窗口, 其过期时间为 2 个单位. 该窗口中有两个键 a 和 b, 需要分别进行聚合, 会话窗口. 流计算系统首先将每条数据都分配到一个过期时间为 2 个单位的窗口中. 之后, 开发者指定一个合并逻辑对这些窗口进行合并. 由于需要对两个键分别聚合, 因此只对时间有重叠且键相同的窗口进行合并. 最后的结果是 a1、a2 被合并到一个窗口中, 而 b1 和 a3 则在单独的窗口中, 实现了非对齐的会话窗口的处理. 通过这种分配-合并的方法, Dataflow 模型允许开发者按照需求开发非对齐的窗口, 提高了窗口编程的灵活程度和表达范围.

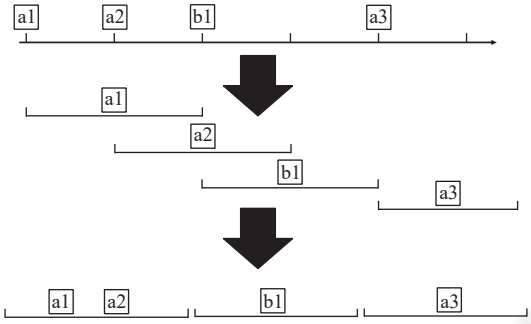

图 4 非对齐窗口的处理

2.3 乱序处理

流数据在进入流计算系统前和进入流计算系统后往往需要进行网络传输, 这为流数据带来了传输延迟和丢失风险. 因此, 流计算系统通常需要处理数据乱序到达的情况. 也就是说, 流数据产生的顺序可能与其到达流计算系统时不同, 其到达流计算系统的顺序也有可能与其被某个算子处理的顺序不同. 乱序到达为流计算系统带来的主要挑战是为开发者编写应用程序带来额外的困难. 具体来说, 由于在输入数据乱序条件下无法预知应用运行时的情况, 如流数据到达的顺序、窗口包含的数据是否全部到达等, 开发者需要在编写流计算应用逻辑时手动处理乱序输入. 这不仅加大了流计算应用的开发复杂度, 也会使应用逻辑和乱序处理逻辑高度耦合, 不利于代码的复用. 因此, 流计算系统需要为开发者提供一套支持乱序处理的编程接口, 使得开发者在不大量改动应用逻辑的前提下就可以对乱序的输入数据进行处理.

早期的流计算系统主要使用基于缓冲区的有序化技术来处理乱序数据, 其中的代表就是 STREAM 系统[15,16]. STREAM 的编程接口要求开发者指定一个用于数据有序化的缓冲区大小. 在数据进入系统前, 会先进入这个指定大小的缓冲区并进行排序, 经过排序后的数据视为有序数据, 由流计算系统处理. Aurora[7]也采用了相似的缓冲区技术, 但开发者可以为不同的算子指定不同的缓冲区策略, 例如是否使用缓冲区和缓冲区的大小等. 在采用了缓冲区技术的流计算系统中, 开发者只需要指定缓冲区的大小, 这使得开发者不必调整应用的处理逻辑就可以对乱序数据进行处理, 降低了开发支持乱序处理的流计算应用的复杂程度. 但是, 基于缓冲区的有序化技术存在两个不可避免的问题. 1) 开发者难以准确设置缓冲区的大小. 缓冲区过小会导致乱序数据得不到有效处理, 而过大缓冲区会带来更大延迟, 因而开发者需要反复调试以找到合适的缓冲区大小. 考虑到流数据的性质不可预知且频繁变化, 确定最优的缓冲区大小极为困难. 2) 缓冲区无法在不影响下游算子的情况下进行乱序处理, 使得开发者难以为不同算子配置不同的乱序处理方法. 例如, 一个流计算应用中的窗口操作算子需要输入数据保持有序性; 但该窗口算子的下游可能存在一个日志记录的算子, 该算子不要求输入数据的有序性, 但需要实时输出日志记录. 采用缓冲区方法的流计算系统在满足了窗口操作算子的数据有序性要求的同时, 将会为日志记录算子带来不可避免的延迟, 无法满足开发者兼顾二者特性的需求.

为了解决以上两点问题, 流计算系统广泛采用了基于标点的乱序处理方式. 基于标点的乱序处理方式通过系统生成的标点来指示流数据到达的进度. 开发者可以在应用中识别这些标点, 并根据其中的信息和应用需求作出相应的处理. 其中应用的最为广泛的是由 Google 提出的 Dataflow 模型[14]. Dataflow 模型通过水位线来标识数据的最晚到达期限, 它标识着流计算系统认为事件时间在某个时刻以前的输入数据已经全部到达. 用户因而可以利用水位线对流计算任务中的无序输入数据进行处理. 例如, 用户既可以在延迟需求较高的场景下选择提前触发窗口聚合计算, 来降低流计算任务的延迟, 也可以等到某个时刻的水位线抵达时再触发窗口聚合计算, 以最大化结果的准确性; 或是多次触发窗口计算, 在实时输出的情况下, 保证了最终结果的准确性. 并且在一个算子触发计算以前, 数据就可以向下游流动, 避免了下游算子受上游算子乱序处理的影响产生以基础. 除此之外, 由于 Dataflow 模型的水位线只是系统对输入数据抵达情况的推测, 因此可能出现迟到数据违反水位线约束的情况出现. Dataflow 模型允许开发者单独配置对迟到数据的处理方式, 通过撤回结果、修正结果或丢弃迟到数据的方式处理这种情

况, 使得开发者拥有了灵活处理乱序输入的能力.

2.4 小 结

本节介绍了流计算系统中的编程接口, 根据流计算系统对编程语义的需求将其分为 3 类: 时间语义、窗口操作和无序处理. 表 1 比较了不同实现方式在易用性、表达力、高效性等方面的优劣.

表 1 编程语义技术总结

类型	实现方式	易用性	表达力	高效性
时间语义	用户定义	×	√	—
	系统定义	√	√	—
窗口语义	手动实现	×	√	√
	系统提供	√	×	×
	分配-合并	√	√	√
乱序处理	基于缓冲区	√	×	×
	基于标点	√	√	√

流计算系统中时间语义的演变方向是由单一的、用户定义的发展为多样化的、系统提供的时间语义. 这种变化为流计算应用的开发者带来了两个好处: 1) 开发者拥有了更丰富的时间语义接口, 从而可以在时间语义方面实现更复杂的处理逻辑, 以应对更多样的应用场景; 2) 开发者可以使用由系统提供的时间语义编程接口, 由系统代替开发者实现时间语义的相关功能和处理逻辑, 从而降低了流计算应用在时间语义方面的开发难度, 提高了流计算系统编程接口的易用性和流计算应用的稳健性.

流计算系统中窗口操作编程接口的主要目的则是实现易用性、表达力和高效性的平衡. 早期的流计算系统通常需要开发者自己定义并实现窗口, 这种方式要求开发者实现复杂繁琐的窗口逻辑, 降低了流计算编程接口的易用性. 后来的流计算系统提供了由系统实现的窗口操作接口, 降低了开发者的编程难度. 但是, 直接使用系统内置的窗口也可能会限制开发者在编程程序时的表达能力, 降低流计算应用的性能. 针对这些问题, Dataflow 模型提出了可组合的窗口操作: 通过将数据分配给指定窗口, 再进行合并的方式, Dataflow 模型允许用户自由选择各种语义和实现方式的窗口, 并通过组合这些窗口实现表达力丰富并具备良好性能的窗口语义.

而在无序处理方面, 流计算系统的主要目标则是为用户提供一个易用且灵活的乱序处理机制, 以满足各种应用的需求. 最早的流计算系统只支持处理有序数据, 开发者需要手动对乱序情况做出处理, 增加了开发流计算应用的复杂程度. 因而一些流计算系统采取了缓冲区来应对无序数据. 但缓冲区不仅需要开发者确定合适的缓冲区大小来以提升性能, 同时也无法满足流计算系统中不同应用、不同算子对乱序数据不同的处理要求. 基于标点的乱序处理方法则解决了这两个问题. 开发者可以根据需求为不同的应用或算子设定不同的乱序处理方式, 从而降低性能开销, 并实现对乱序数据的灵活处理.

3 执行计划

开发者提交到流计算系统中的应用程序通常只包括流计算的逻辑语义, 并不包括具体的运行细节. 由于流计算应用通常运行在各种复杂的底层硬件环境中, 包括单台个人计算机、分布式集群、高性能计算设备或云环境等. 因此, 流计算系统需要根据开发者提交的程序, 结合系统运行的具体硬件情况, 确定适宜的算子间的拓扑结构、算子的并行度、算子的实现细节等信息, 以生成具体的执行计划. 一个应用程序往往对应很多不同的执行计划, 而如果流计算系统采用了低效的执行计划, 就会导致资源利用不足, 流计算应用性能降低等情况. 因此, 流计算系统需要在众多的可能中生成一个效率较高的执行计划, 从而提高流计算应用的性能.

用于批处理的大规模数据处理系统也需要为应用程序生成执行计划. 例如, SparkSQL[17]可以根据用户提交的查询确定物理算子, 从而生成一个适宜于分布式环境中运行的执行计划. 但是由于流数据和流计算应用的特殊性, 流计算系统在生成执行计划时通常还面临着以下问题:

1) 窗口操作是流计算中最重要的语义之一: 一方面, 窗口操作在流计算应用中十分普遍, 流计算应用中许多从

时间中提取信息的语义都需要用到窗口操作; 另一方面, 窗口操作需要在触发时处理大量数据, 带来了较大的计算负载和显著的延迟. 这为流计算系统在如何提供窗口操作的具体实现上带来了挑战, 因为简单的窗口实现可能会导致内存占用上升、服务延迟增加和计算冗余等问题. 流计算系统需要针对不同的窗口语义和数据特征实现高效的窗口操作, 从而提升流计算应用的整体性能.

2) 由于流数据实时生成、实时到达和实时处理的特点, 流计算系统无法预知整体或未来一段时间内输入数据的特征. 而流计算系统通常面临高度变化的输入数据, 为流计算系统的执行计划生成带来了挑战. 这主要体现在两个方面: 首先, 流计算系统面临的输入数据的分布是不断变化的, 不论是键的分布还是时间戳的分布, 都可能会导致流计算系统在分区时产生数据倾斜, 从而造成同一个算子的多个并行实例之间的计算负载不均, 导致流计算系统资源利用率降低和整体性能的下降. 另一方面, 流计算面临的输入数据特征也处于高度变化之中, 导致流计算应用中某些算子的统计信息发生变化; 例如输入数据值的分布改变, 可能造成流计算应用中算子的选择率大大变化, 为执行计划的代价估计造成了严重的困难. 因此, 流计算系统应当生成可以应对动态负载的执行计划, 以在运行过程中根据负载和硬件相关情况对执行计划进行自适应调整.

3.1 窗口聚合优化

窗口操作是流计算应用中较为常见的计算操作. 朴素的窗口聚合实现通常会产生大量的内存开销和计算负载, 这是因为窗口操作需要 1) 记录所有属于该窗口的到达数据, 将它们保存在内存中, 并 2) 按照编程语义在可以触发窗口计算之后访问这些数据进行聚合运算.

在第 1 步中流计算系统需要将分配到窗口的原始数据都保存在内存中, 带来了额外的内存开销. 这个额外的内存开销与流数据的输入速率成正比; 此外, 因为流计算系统需要等待更长的时间确认窗口可以触发计算, 所以数据的无序到达也会加剧对内存的消耗. 综上, 当流计算系统面临高速输入的无序数据时, 可能会面临内存不足的情况, 从而造成窗口操作延迟或失效. 流计算系统需要对窗口操作的数据保存方式进行优化, 降低内存开销, 以达到更高的吞吐量.

在第 2 步中, 由于聚合运算在确认窗口可以触发计算 (即可以输出) 时才开始计算, 不能有效利用等待窗口数据到达的时间进行运算, 造成了计算负载在时间上的不均匀性. 这样的结果就是在所有数据到达窗口后仍需要较长的计算时间, 为流计算应用中的窗口操作带来了显著的延迟. 因此, 流计算系统需要针对窗口操作的计算时机进行优化, 避免窗口触发后进行的密集计算为流计算应用带来的显著延迟.

3.1.1 降低内存开销

流计算系统可以采取两种方式来降低窗口计算过程中的内存开销: 一种是减少需要保存的元组数量; 另一种是利用磁盘来缓解内存容量不足的压力. 接下来本文将分别介绍这两类技术.

第 1 种技术预聚合是窗口实现中一种常见的优化技术[18], 该技术提前对属于该窗口的部分元组进行聚合, 形成部分聚合结果; 之后再在部分聚合结果的基础上计算最终聚合结果. 图 5 是应用了预聚合技术的一个示例, 该窗口接收输入的流数据并对他们进行求和操作. 最下面一行的方块代表窗口处理的原始数据, 方块内是该条数据的值. 应用了预聚合技术之后, 在接收到前 3 条数据时, 流计算系统就可以进行预聚合计算, 得出前 3 条数据的和为 12. 由于计算最终聚合结果只需要将 3 次预聚合结果相加, 因此在得到第 1 个预聚合结果后, 该窗口就可以不再保留前 3 条数据, 从而减少了需要保存的元组数量, 有效降低了窗口聚合过程的内存开销.

另一种可以用来优化流计算系统窗口操作的内存占用的方式是将部分数据溢写到磁盘中. 流计算系统通过将部分数据保存在磁盘上, 从而减少了对内存的消耗. 然而, 向磁盘存取数据的过程通常显著慢于向内存存取数据的过程, 这可能为流计算系统带来不可忽视的延迟. 因此, 流计算系统在应用溢写磁盘技术时, 需要精心设计一套数据存取策略来规定向磁盘溢写数据的方式, 从而最小化读写磁盘为流计算应用带来的性能恶化. Photon[19]是一个被设计用于大规模数据处理的流计算系统; Photon 为了支持长时间的窗口 join 操作, 引入了磁盘来存储需要被暂时保存的元组. Photon 观察到了大部分数据都是在到达后短时间内被访问的现象, 因而设计了一个层级的数据存储结构: 到达的数据先被保存到内存中, 其中大部分将会在短时间内被访问并丢弃; 部分未被访问的数据会被溢写到磁盘上以降低内存开销, 同时系统在内存中建立了一个索引以快速访问这部分数据. Railgun[20]为了实现大规模

窗口低延迟的准确计算, 设计了一个名为事件仓库的数据结构来保存窗口中的元组. Railgun 的事件仓库通过将窗口中的大多数元组溢写到磁盘上来减少对内存的占用, 并应用了两个优化技术来提高系统访问磁盘上元组的效率. 首先, Railgun 在向磁盘溢写数据前, 会先将时间戳相近的多条元组合并为一个块, 通过读写块代替读写单条数据, 降低了磁盘访问 IO 次数, 减轻了磁盘访问代价对窗口性能带来的影响. 其次, Railgun 基于窗口中的数据通常是按照时间戳顺序被消耗的观察, 将多个块有序排列, 并在读取时利用操作系统的缓存机制预先读取相邻的块. 由于这些块在时间戳上的相近, 他们很可能在接下来被流计算系统从缓存中读取, 从而大大减少了流计算系统等待磁盘 IO 带来的性能恶化.

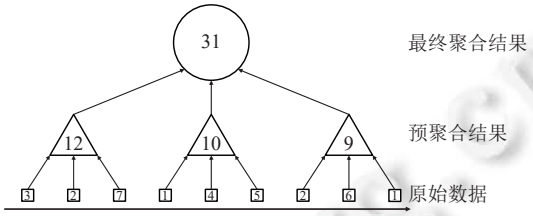

图 5 预聚合示意图

3.1.2 降低触发延迟

在流计算系统中有两种优化技术可用于降低窗口的触发延迟. 首先, 预聚合技术可以在窗口数据未完全到达时, 利用这部分不完全的数据计算部分聚合结果, 再基于部分聚合结果计算最终聚合结果, 即在窗口到期之前分摊一部分计算量以减少窗口到期后窗口计算带来的延迟. 同样以图 5 中的窗口求和为例; 如果不应用预聚合技术, 该窗口需要等到最后一条数据到达后才开始计算, 即在窗口触发后需要进行 8 次求和操作. 但在应用了预聚合技术后, 在窗口触发前, 前 6 条数据已经通过预聚合计算得到了两个预聚合结果; 因此窗口触发时只需要进行 5 次求和操作, 有效降低了窗口的触发延迟. WID[21]是一个应用了预聚合技术的窗口实现的典型例子: WID 并不保存所有的原始数据, 而是为每个窗口维护一个部分聚合结果. 当数据到达时, WID 会根据聚合函数的语义逻辑更新其对应的一个或多个聚合结果; 在窗口计算时, WID 会直接输出该聚合结果, 而非利用所有原始数据开始从头计算. 通过将聚合操作的计算从窗口结束后分摊到窗口中, WID 相比于保存所有原始数据再进行计算的方法在执行延迟上取得了较大的改善: 相较于使用朴素缓冲区的窗口实现方式, WID 最多降低了约 70% 的触发延迟.

另一种用于降低流计算系统中窗口触发延迟的技术是近似计算. 由于流计算应用通常对实时计算结果的精确度存在容忍空间, 因此很多情况下窗口计算只需要处理部分数据就可以满足流计算应用的精度要求. 因此, 流计算系统可以在保证窗口计算结果的精度在应用可接受范围内的情况下, 跳过处理某些暂时未到达的数据的过程, 从而提前输出窗口聚合的结果, 达到减少窗口触发延迟的目的. 流计算系统很早就存在减载技术, 即跳过处理一部分元组以提高流计算系统的性能. Tatbul 等人在 2003 年的工作 [22]就是一个例子: 他们详细研究了如何在合适的时机触发减载以保证系统的性能, 以及如何最小化丢弃部分元组对结果带来的影响. 虽然这些减载技术可以有效提高系统性能, 但其对窗口聚合结果精度影响是未知的; 因此流计算系统需要一类可以保证精度的近似计算方法. AQ-K-slack[23]是一个基于缓冲区的近似计算解决方案. 传统的缓冲区技术使用固定的缓冲区大小, 用户需要配置合适的缓冲区大小才能实现计算精度和触发延迟的平衡, 这为用户使用缓冲区技术提出了严峻的挑战. AQ-K-slack 则将计算精度代替缓冲区大小提供给用户进行配置, 并通过计算达到该精度所需要的窗口覆盖率, 即处理的数据占窗口中所有数据的百分比. 根据窗口覆盖率, AQ-K-slack 确定一个合适的缓冲区大小, 在保证足够多的数据得到处理后, 丢弃那些多余的迟到数据, 从而优化窗口触发延迟. SPEAr[24]则提供了基于水位线的近似计算解决方案. SPEAr 会在窗口结束之前计算一个采用部分元组计算得到的近似结果, 并估计近似结果的准确率. 在水位线到达时, SPEAr 会验证窗口聚合结果的准确率是否满负责户需求. 如果准确率符合用户的需求, 则直接输出该近似结果, 否则使用全部数据重新计算准确的结果. 实验结果证实, 在窗口包含的元组较多 (大于 5 000 个) 时, SPEAr 对窗口的触发延迟有着显著改善. 在不影响计算结果精度的情况下, SPEAr 可以减少一个数量级以上的窗口处理时间.

3.2 自适应执行计划

实时到达是流数据的一个重要特征, 也是流计算系统在生成执行计划时面临的主要挑战之一. 对于批处理系统而言, 系统通常通过随机采样等方法确定输入数据的一些统计特征, 如数据规模、稀疏度、键值分布等, 并基于这些统计特征为某个作业选择一个最优的执行计划. 但流数据的高度动态和实时到达的特征却使得在流计算系统中直接应用以上方法变得困难, 这主要有两方面的原因: 首先, 流计算系统无法获取在整个执行过程中输入数据的分布, 因此流计算系统很难确定一个最优的分区方法; 这会在执行过程中造成数据分区的不均衡, 进而导致各个任务间处理负载的不均衡, 造成系统资源浪费, 损害系统的性能. 另一方面, 由于流数据的统计特征处于高度变化之中, 各个时刻流数据的统计特征存在较大差异, 而基于这些统计特征生成的最优执行计划也有所不同. 因此, 对于同一个流计算应用来说, 最优的执行计划会不断变化, 而静态的执行计划无法在整个执行流程中都保持高效.

3.2.1 动态分区方式

流数据的实时性对流计算系统提出的第 1 个挑战在于, 输入数据的键分布可能会产生大的变化. 流计算系统通常会为编程逻辑中的一个算子生成多个并行实例, 并按照输入数据的键将数据分区到其中一个并行实例进行处理. 因此, 如果数据分区不均匀, 很容易造成各个实例的处理负载不均衡, 形成单点瓶颈, 进而降低应用的整体性能. 批处理系统通常会在开始运行前对输入数据进行抽样分析, 获取关于键分布的统计信息, 运用这些统计信息对数据进行分区, 来保证各个分区之间负载均衡. 但流计算系统无法在开始运行前获取关于数据的信息, 于是无法基于抽样统计来决定数据分区. 进一步来说, 由于流数据的分布通常会随时间不断变化, 因而无法找到一个固定的分区方式能使得整个运行过程中数据分区都保持均衡. 因此, 流计算系统在生成执行计划时, 需要采用动态的分区函数, 根据流计算应用运行的情况不断调整分区方式, 以保证在整个运行过程中流计算应用的分区方式的高效.

根据是否要根据数据的键对数据进行分区, 流计算系统中应用的分区技术大体可以分为 3 类: 1) 随机分区, 即不考虑流数据的键, 按照其到达顺序随机分配给一个计算实例进行处理. 随机分区最大程度地保证了分区的均匀, 但由于拥有同一个键的多条数据可能会被分配到不同的计算实例中, 因此在各个计算实例处理完毕后, 还需要一个额外的聚合过程汇总多个计算实例输出的结果. 2) 按键分区, 即根据流数据的键进行分区, 通过哈希映射等方式将拥有同一个键的数据分到同一个分区中; 由于按键分区保证了相同键的数据被同一个实例处理, 因此无需额外的聚合过程. 3) 部分按键分区, 即结合以上两种方法: 基于按键分区的方式以减少不必要的聚合过程, 在负载不均衡的情况下将部分数据划分到其他分区来减少分区不均对性能的影响. 在分区策略设置恰当地情况下, 部分按键分区可以结合随机分区和按键分区的优点, 最小化分区不均带来的性能影响和聚合过程的额外开销的和, 进而提高分区的最终性能.

由于向各个计算实例发送的数据数量几乎相等, 随机分区技术可以最大程度地保证分区的均匀性, 且不受输入数据键分布变化的影响. 虽然在理想状况下随机分区可以完全消除各个计算实例之间的负载不均衡, 但这是建立在算子处理每一条数据的计算量相同的假设下. 实际情况中, 不同的数据可能需要大小不同的计算量. 针对这个问题, Rivetti 等人提出了 OSG (online shuffle grouping)[25]技术, 通过估计每条数据的处理时间, 来实现基于处理代价的实时随机分区; 在不同数据处理代价有所差异的情况下, 这种基于处理代价的随机分区相较于基于数据数量的随机分区消除了各计算实例之间的负载不均衡, 进而提高了流计算系统的整体性能. 虽然随机分区可以保证分区的均匀性、降低分区过程的响应延迟, 但其应用场景存在一定的限制. 对于流计算应用中常见的有状态计算, 即各个算子都维护一个和键相关的状态, 随机分区技术不可避免需要大量额外的聚合过程以汇总各个实例的计算结果, 对流计算系统的性能造成负面影响. 因此, 随机分区技术通常只能被应用到无状态算子中.

为避免随机分区给有状态计算带来大量的聚合开销, 流计算系统通常会采用按键分区的分区方式. 但按键分区可能会造成流计算系统分区的倾斜, 导致各个计算实例之间的负载不均衡, 损害流计算系统的整体性能. 因此按键分区技术需要优化分区方式, 避免数据倾斜情况的出现. 按键分区最简单的实现方法是使用一个固定的哈希函数, 计算数据键的哈希值, 并将数据映射到其中一个分区中. 基于固定哈希函数的按键分区方式因为其实现简单、使用方便, 被广泛应用在各类流计算系统中. 但由于哈希函数对输入数据键分布的不了解, 且无法在运行过程中进行动态调整, 因此给予固定哈希函数的按键分区方式很容易造成数据倾斜. 为解决区数据倾斜的问题, 一些流计算

系统采用了自适应重新平衡的技术改进按键分区. Shah 等人[26]通过在流计算系统中引入一个名为 Flux 的算子来实现自适应重新平衡. Flux 算子跟踪统计当前数据分区的情况, 并在分区严重失衡的情况下进行分区策略的调整. Balkesen 等人[27]采用了类似的方法, 在使用哈希函数的同时, 跟踪统计输入数据键的频率, 根据频率调整哈希函数以保证负载均衡. 自适应重新平衡虽然可以解决按键分区的倾斜问题, 但也带来了另一个重大开销: 在有状态计算中, 算子通常需要为每个键维护一个状态; 如果在运行过程中要将某个键对应的数据从一个分区调整至另一个分区, 就必须同时将与这个键相关联的状态也迁移到另一个分区对应的算子. 这个过程中不仅造成了额外的网络通讯开销, 迁移完成前该算子还需要暂停处理, 带来了流计算服务的暂时中断和延迟上升. 要因为重新平衡带来的状态迁移开销, Gedik 等人[28]的工作中引入了一致哈希技术来减少需要迁移的状态数量; 为了避免一致哈希在分区数量不足时存在的负载倾斜问题, 他们设计了一种混合哈希技术, 使用一致哈希来处理数据量较大的键, 而使用统一哈希函数来处理数据量较小的键. 混合哈希技术结合了一致哈希和固定哈希的优点, 降低了分区的负载不均和调整分区时的迁移代价, 提高了分区过程的整体性能.

部分按键分区可以看作是按键分区的一个变种. 相较于按键分区要求属于同一个键的所有数据都被划分到一个分区之中, 部分按键分区可以将部分键拆分到多个分区中, 分别由不同的计算实例进行处理, 减少了由于少部分热点键带来的数据倾斜. Nasir 等人[29]针对并行度为 2 的算子的数据分区问题提出了部分按键分区的技术. 他们的方法针对数据量较多的热键进行了键拆分, 即将这些键对应的数据拆分为两部分, 并分别由两个算子处理. 之后他们进一步将这项技术拓展到了算子并行度大于 2 的多个分区场景[30]. Katsipoulakis 等人[31]详细评估了部分按键分区的不均衡代价和聚合代价, 并基于代价模型提出了一种新的分区算法, 实现了更好的分区效果, 以及更小的延迟和内存开销. 他们的实验证实, 相比于朴素的分区方法, 部分按键分区在不平衡的输入负载上表现出了对延迟的显著优化. 部分案件分区减少了 60% 以上的平均延迟以及 80% 以上的尾延迟. 部分按键分区技术降低了数据倾斜造成的系统负载不均衡, 但同时也引入一定的聚合开销. 因此在实际应用中需要精心权衡这两者的代价才能实现流计算应用的性能最大化.

3.2.2 动态计划调整

不断变化的输入数据为流计算系统带来的另一个挑战在于, 流计算系统很难在应用开始运行前就生成一份在整个应用生命周期中都保持高效的执行计划. 为了应对输入数据特征变化对于执行计划性能的影响, 流计算系统需要依据数据的实时变化动态调整执行计划, 以改善流计算应用的性能.

然而, 流计算系统采用的连续执行模式为执行计划的动态调整带来了困难. 为了在连续执行的流计算应用中进行执行计划的优化, 许多研究工作聚焦于如何在流计算应用运行过程中进行动态计划调整. 依据在动态计划调整过程中, 是否存在多个版本的执行计划同时运行, 可以将动态计划调整的工作分为单版本计划调整和多版本计划调整.

单版本计划调整是指在流计算系统的执行计划动态调整期间, 系统始终只维护并运行一个版本的执行计划. 一个直观的单版本计划调整的实现方式是重新启动: 先中止流计算服务[32]. 这种方式不仅实现简单, 还可以保证流计算应用运行结果的一致性. 然而, 流计算系统的动态计划调整通常只涉及小部分算子, 将整个流计算服务重启会造成资源的浪费和流计算服务的长时间中止, 严重损害流计算应用的性能, 也不利于流计算系统根据输入数据的实时变化频繁调整执行计划. 为了降低计划调整对流计算应用性能的影响, 后续的研究工作更多采用了算子级别的计划调整代替整体重启. SPADE[33]使用了代码生成技术产生调整后的执行计划. 同时, SPADE 将多个算子合并为一个处理单元, 以处理单元为最小单位进行执行计划的调整. SPADE 利用以上两点改进加快了动态计划调整的过程, 但关于 SPADE 的工作中并未讨论如何在计划调整过程中保证结果的一致性. Ding 等人在 SQO[34]中实现了一种名为多模态算子的机制, 即为每个算子提供多种实现方式; 系统可以在运行过程中, 根据优化执行计划的需求, 独立地调整每一个算子实例的实现方式, 从而实现动态的计划调整. 这种方式有效降低了动态计划调整对系统造成的性能影响, 但独立调整单个实例的实现方式可能造成流计算应用结果的不一致. 为了解决这个问题, SQO 要求每个实例和固定的数据分区一一对应, 从而提供结果的一致性保障. Grizzly[35]采用代码生成技术产生新的执行计划, 并且使用线程来控制执行计划的切换. 每个线程都可以独立调整执行计划, 并使用线程间的通讯机制来保

证结果的一致性. 得益于线程切换机制的高效性, Grizzly 的动态计划调整过程产生的开销微不足道, 这允许 Grizzly 可以在运行过程中频繁调整执行计划. 但采用线程控制计划切换的机制在可拓展性方面存在限制, 难以应用在基于大规模分布式集群的流计算系统中. Chi[36]采用了一种基于标点的计划调整方法, 将计划调整的信号嵌入流计算系统的数据流中. 接收到信号的算子先等待标点到达, 以确保该历元的数据已完全被处理; 这之后算子就可以进行计划的调整. 当信号在系统中完整流动后, 所有算子都完成了计划的调整. 这种基于信号流动进行计划调整的方式在不造成服务中断的情况下, 保证了处理结果的一致性, 同时具有良好的可拓展性. 虽然基于信号的计划调整可以在不重启服务的情况下保证结果的一致性, 但由于未被调整的算子仍需等待历元进行同步, 造成了不必要的性能损失. 因此近年来一些工作致力于优化计划调整过程中同步带来的开销. 例如, Meces[37]通过一个全局的协调器来更新在计划调整中受影响的部分算子, 这使得其他算子可以继续运行, 不必等待历元或其他同步方式, 从而降低了计划调整产生的开销. 单版本计划调整避免了因引入多个同时运行的执行计划, 不必造成对计算资源和内存空间的多倍占用, 适合被应用于计算和存储资源较为紧张的场景. 但单版本计划调整可能造成服务的暂时中断, 因此采用单版本计划调整技术的同时需要采取方案降低服务中断的时间, 并同时保证应用处理结果的一致性.

另外一些流计算系统采用了多版本计划调整来实现动态计划调整. 这些系统会在流计算应用运行过程中同时维护并运行多个版本的执行计划. 在执行计划动态调整的过程中, 系统会先在执行计划的一个或多个副本中应用这些调整; 当调整完成并经过性能分析证实有效性后, 系统会用副本替换原始的执行计划, 从而完成执行计划的动态调整. 在采用多版本计划调整时, 流计算系统需要额外的资源来运行执行计划的副本, 从而带来一定的资源开销, 但多版本计划调整也具有一定优点. 首先, 多版本计划可以有效降低计划调整过程带来的延迟, 这是因为执行计划调整可以简单地通过重定向输入数据到新的计划来完成. 其次, 由于多个版本的执行计划都会产生输出结果, 流计算系统可以通过合并这些结果来解决计划调整带来的一致性问题. 最后, 流计算系统可以在不影响原始计划正常运行的情况下, 在副本计划上进行性能测试, 避免不当的计划调整损害流计算应用性能. Heinz 等人[38]在他们的系统中采用了多版本执行计划调整, 以进行优化测试和性能分析. 他们将流计算系统分为主系统和第 2 系统两部分, 主系统负责执行流计算应用程序, 第 2 系统维护一个逻辑和主系统一样的执行计划副本. 第 2 系统监控主系统的运行情况, 并在执行计划副本上应用调整、分析性能, 并在合适的时机替换主系统的执行计划来完成计划调整. Turbine[39]提出了一个大规模分布式流计算系统中的执行计划更新机制. 其采用多版本计划来进行调整: Turbine 在计划更新时首先将发生变化的算子提交到一个执行计划的副本中, 再在合适的时机用副本中新的执行计划替换原始执行计划, 并在此时进行状态同步和更新等操作. 多版本计划调整为 Turbine 带来了两个好处: 首先, Turbine 可以将多个算子的更新过程合并, 降低调整过程带来的延迟; 其次, Turbine 可以在替换原始执行计划过程中进行状态同步和更新, 避免了计划更新带来的一致性问题. 然而, 随着近年流计算系统执行计划对于空间 (如大规模哈希表) 和计算 (如深度学习模型) 需求的不断增长, 多版本计划调整会产生更多的内存需求和计算开销. 流计算系统亦可能因为内存限制而无法满足同时维护多个执行计划版本. 这限制了多版本计划调整的应用场景和可拓展性.

3.2.3 小 结

本节综述了流计算系统在生成和优化执行计划时面临的挑战及其对应的解决方案. 流数据的特殊性为流计算系统生成及优化执行计划主要带来了两方面的挑战: 第一, 由于流数据的无界性, 流计算系统需要大量应用窗口聚合对流数据进行处理, 而窗口聚合的实现方式将严重影响流计算应用的性能. 因此流计算系统需要优化执行计划中的窗口聚合实现方式. 第二, 由于流数据的实时性, 因此一成不变的执行计划无法在整个应用周期内保持高效. 流计算系统因而需要在运行过程中根据输入数据的变化自适应地对执行计划进行调整.

如表 2 所示, 流计算系统中窗口聚合优化的优化目标主要分为两类: 减少内存占用和降低触发延迟. 将内存中部分数据溢写到磁盘可以利用磁盘的大量空间来保存数据, 从而减少内存占用, 但磁盘的读写操作可能会降低流计算应用的性能. 另一种减少窗口聚合内存占用的思路是减少需要保存的数据量, 这可以通过近似计算或预聚合实现. 前者可以在不保存全部原始数据的基础上计算结果, 而后者则通过用预聚合结果代替原始数据降低了需要保存的数据量. 此外, 预聚合和近似计算可以被用于降低窗口聚合的触发延迟. 预聚合通过将窗口触发时的部分聚

合运算提前到窗口触发前, 降低了窗口的触发延迟. 近似计算则可以不必等待所有数据到达就可以触发窗口计算, 从而达到了降低窗口延迟的目的.

表 2 窗口聚合优化技术总结

优化技术	内存占用		触发延迟	
	减少元组数量	增加可用空间	计算数量减少	触发时间提前
溢写内存数据	×	√	×	×
预聚合	√	×	√	×
近似计算	√	×	√	√

输入数据的动态变化给流计算系统带来了两方面问题. 首先, 流数据键的分布变化可能为流计算系统带来负载不均的问题, 因此流计算系统需要采用动态分区技术以在运行过程中实时调整. 其次, 流数据统计特征的变化可能导致执行计划效率下降. 流计算系统因而需要在执行过程中对执行计划进行动态更新, 来满足流计算应用的性能需求.

为了在输入数据键分布变化的情况下实现负载均衡, 流计算系统采用了 3 类分区技术, 如表 3 和表 4 所示. 随机分区可以实现最大程度的负载均衡, 但随之而来的聚合过程会产生大量网络和计算开销, 降低系统性能. 按键分区避免了聚合过程, 并且可以通过动态调整分区的方式改善负载均衡. 部分按键分区根据键的特性, 如访问频率等, 将不同的键分别处理, 既降低了需要进行聚合的数据量, 又近似到达了随机分区的负载均衡水平.

表 3 动态分区技术总结

分区技术	负载均衡	聚合开销
随机分区	√	×
按键分区	×	√
部分按键分区	√	√

表 4 动态计划技术总结

计划技术	资源占用	更新延迟	一致性保证
单版本	√	—	×
多版本	×	√	√

除了需要动态调整数据分区机制外, 流计算系统还需要进行执行计划的动态更新. 执行计划的动态更新方式主要分为单版本计划更新和多版本计划更新. 其中, 单版本计划更新由于不需要额外资源在多数系统中被采用. 这些工作同时也研究了如何在保证结果一致性的情况下降低计划更新的延迟. 相应的, 多版本计划更新需要额外的资源来维护执行计划的副本, 但多版本的计划更新方式在版本管理和一致性保证上具有优势, 但其造成了更多计算和内存开销, 使其应用场景和可拓展性受到一定限制.

4 资源调度

在流计算系统中, 低效的硬件资源调度可能会导致资源浪费、系统性能降低等问题, 现代流计算系统常常运行在复杂的底层硬件架构 (如异构硬件、分布式环境、云计算) 上, 这进一步加剧了资源调度的困难程度, 因此如何高效地调度资源就成为流计算系统中的一个重要挑战. 相较于数据库系统和批处理系统, 流计算系统的资源调度有两点不同.

1) 流计算系统广泛采用了连续处理模型, 即执行计划中的每个算子会生成一个或多个并行任务, 每个任务都长期运行, 等待数据输入并执行特定的处理逻辑. 该模型使得流计算作业中的任务因处理逻辑不同而具有不同的计算负载, 每个任务都可能成为作业的性能瓶颈. 在这种情况下, 以作业为单位进行资源调度会造成资源的浪费. 因此流计算系统需要设计调度策略来降低资源调度过程中的资源浪费, 从而减少资源的使用量.

2) 流计算系统对于延迟有着很高的要求. 不当的资源调度策略很可能会恶化流计算系统的服务延迟, 使之不能满足应用的需求. 因此, 流计算系统的资源调度需要将调度对延迟的影响纳入考虑, 从而避免资源调度造成的延迟恶化, 实现流计算系统的低延迟处理.

流计算系统的资源调度通常分为两个过程: 首先, 流计算系统通过监控某些指标判断系统的资源使用情况, 如

果系统处于资源过剩或不足的情况, 则进行资源调度. 之后, 流计算系统根据特定的策略申请并分配资源. 因而, 为了在流计算系统中实现低资源占用和低延迟的资源调度机制, 需要从两方面着手. 第一, 如何监控流计算系统中资源使用情况, 以便在合适的时机触发资源调度. 更加精确的资源监控实现方式可以提升流计算系统中资源调度的效果, 减少资源浪费; 但同时也可能因为过多的性能分析为流计算系统的正常运行带来性能损耗, 造成延迟上升. 第二, 如何向流计算系统中的任务提供资源. 流计算系统可以通过不同的方式向任务提供资源, 如增加节点、分配内存池或调整进程优先级等, 这些资源提供的实现方式会影响资源调度过程的延迟和效果. 本节将从流计算系统实现资源监控和资源提供的过程出发, 介绍资源调度的相关工作技术, 以及它们如何帮助流计算系统在资源调度过程中减少资源浪费并降低服务延迟.

4.1 资源监控

监测系统平均资源使用率的传统方法在批处理等系统中被广泛使用, 然而该方法不利于在流计算系统中减少资源浪费. 这是由于流计算系统的各个算子的资源使用情况存在较大的差异, 利用系统平均资源使用率无法获悉各个任务的资源使用细节. 因此, 为了能使调度过程减少资源浪费, 流计算系统需要在资源调度中使用更细粒度的资源监测方式, 进而更全面且及时地发现系统资源伸缩需求.

第 1 种在流计算系统中广泛采用的是基于节点的硬件资源使用率的资源监测方式. 一个典型的例子是 StreamCloud[40], 其利用这种资源监测方式实现了一个基于子查询的资源调度机制. 具体来说, StreamCloud 将流计算应用拆分为多个子查询, 每个子查询可能包含一个或多个算子. StreamCloud 将不同的子查询分别部署在不同的节点上, 并通过各个节点的资源使用率来监测这些子查询的资源使用状况. 这种方式利用了节点资源使用率来监测流计算系统算子的资源使用情况, 最小化资源监测的复杂度和资源监测对流计算系统性能的影响, 是一种简单直观但有效的资源监测方式. 然而, 这种方式也具备明显的缺点. 由于流计算系统需要在应用开始执行前就确定子查询的划分, 并且无法在执行过程中更改划分方式, 因此子查询的划分方式将严重影响资源监测的效果. 更多地, 这种方式无法精确为每个算子分配资源. 流计算系统中另一种资源监测方式是算子级别的监测方法, 这种方法解决了基于硬件资源利用率的资源监测方式在灵活性和精确性上的不足. DS2[41]是一个采用算子级别资源监控进而实现精确实时的流计算资源伸缩的调度控制器, 其有效解决了流计算系统中因资源分配不当所带来的背压、延迟上升和资源浪费等问题. DS2 基于算子运行时间通常与其计算负载正相关的现象, 通过监测每一个算子近期的平均运行时间来估计其需要的计算资源. 这种监测算子运行时间的资源监测方法能够精确及时地检测不同算子资源使用情况, 有效改善了之前方法在灵活性和精确度上的不足; 但是, 频繁监测统计算子的执行时间为流计算系统带来了不可忽视的额外开销, 并最终影响流计算系统的整体性能. Cameo[42]中实现的算子级别资源监控方式采用对算子的资源使用情况进行建模的方式缓解了这一影响: 通过分析一段时间内的应用运行情况, 确定算子的计算负载与输入数据特性, 如输入速率、选择率等的关系, 建立一个算子的资源使用模型. 在应用的长时间运行过程中, 只需要利用建立的模型就可以大致确定算子的资源消耗情况, 无需频繁统计算子的运行情况. Cameo 实现的基于资源模型的资源监控方式在几乎不影响监控精确程度的情况下, 大大减少了资源监控对于流计算系统性能的影响.

另一方面, 为了利用资源调度过程来有效降低流计算系统较为关注的服务延迟, 流计算系统在实现资源监控等过程中还考虑了延迟约束. 传统资源调度框架的目标通常是提高系统吞吐量或提高资源利用率, 其通常不会在监控中考虑延迟约束. 这导致资源调度过程虽然提高了资源利用率和系统吞吐量, 但并未改善延迟的恶化, 甚至反而对延迟有负面影响. 流计算系统的资源监控方式可以从两个方面改善传统监控方式对延迟考虑不足的情况. 首先, 针对传统资源监测方式对延迟恶化不敏感的现象, 流计算系统的资源监控方式可以将端到端服务延迟纳入监控范围. 基于这种做法, 流计算系统可以将延迟恶化作为触发资源调度的条件, 从而对资源不足所导致的应用延迟恶化作出及时响应. Lohrmann 等人[43]设计一个考虑延迟约束的流计算资源调度框架. 为了提供流计算延迟保证, 该资源调度框架会在流计算应用运行过程中持续监测端到端延迟, 并在延迟违反约束的情况下触发资源调度. Dhalion[44]作为一个模块化的流计算系统资源调度框架, 其允许用户依照需求使用各种指标来监测流计算系统的运行状态. 其中, Dhalion 可以被配置为根据应用的端到端延迟来触发资源调度, 从而达到在流计算应用延迟恶化时及时进行资源调度的目的. 另一方面, 资源调度通常会导致流计算应用服务的暂时中止, 导致流计算应用延迟上

升. 因此流计算系统可以在资源监控过程考虑资源调度带来的延迟恶化代价, 即延迟惩罚, 权衡比较资源调度带来的短期延时上升和维持现状导致的长期性能恶化对应用性能的影响程度, 从而判断是否进行资源调度. Heinze 等人[45]将流计算系统执行资源调度的延迟惩罚纳入考虑之中. 他们的工作通过建立模型的方式估计流计算系统资源调度的延迟惩罚, 并基于代价模型设计了一套调度策略. 实验显示, 该策略能够在提高资源利用率、保证系统吞吐量的同时, 大幅降低流计算系统中因资源调度引起的延迟约束违反, 从而降低流计算应用的整体延迟.

4.2 资源提供

在实现资源提供机制的过程中, 为了减少调度过程中的资源浪费, 流计算系统需要实现细粒度的资源提供方式. 传统的资源管理框架通常向流计算系统提供更多的节点 (虚拟机) 来完成资源伸缩. 上文提到的 StreamCloud[40]就利用了这种机制来实现流计算系统的资源提供. StreamCloud 将流计算应用的任务划分为多组 (多个子查询), 并针对子查询改进了资源提供的方法: 虽然 StreamCloud 也是采用增减虚拟机的方式进行资源管理, 但相比于为系统内所有任务平均地增加资源, StreamCloud 会优先为资源需求较大的子查询调度资源. 由于单个子查询包含的算子都运行在相同的节点上, 因此 StreamCloud 很容易使用增减节点的方式为子查询进行资源调度. 但增减节点的资源提供粒度仍受限于子查询的定义, 无法为某个子任务调度资源.

第 2 种资源提供方式是计算资源分配, 其代表 Elasticutor[46]是一个用于流计算系统资源调度的弹性伸缩框架. Elasticutor 通过为需要计算资源的算子分配更多的处理器核心来实现资源提供. 相比于节点级别的资源提供方式, 分配处理器核心可以实现更细粒度的资源调度, 进而避免了过度提供资源带来的资源浪费. 类似的, 一些工作通过调整不同算子的执行线程优先级来实现资源调度, 如 DS2[41]和 Cameo[42]. 相比于 Elasticutor 需要框架开发者和应用开发者配置处理器核心的调度策略, 通过调整线程优先级的方式进行资源调度可以有效利用操作系统的资源调度机制, 实现更为简单, 同时也提高了系统的稳健性. 相较于通过增减节点的资源提供方式, 计算资源分配支持为单个子任务进行精确的资源调度. 但受限于机器或集群的计算资源上限, 计算资源分配存在可拓展性的限制.

另一方面, 为了满足应用的延迟需求, 流计算系统也广泛聚焦于降低资源提供过程带来的延迟. 第 1 种优化延迟的方法是加速资源申请和释放的过程. 增减节点的资源提供方式可以通过预先启动的方式加快资源提供过程. 例如, StreamCloud[40]和 SEEP[47]为了加快资源提供的过程, 提前维护了一组资源池, 并在资源池中的空闲节点上提前部署了资源调度框架, 但并不执行任何任务. 当这些空闲节点需要被加入流计算系统中时, 流计算系统可以直接在节点上部署计算任务, 从而避免了启动节点和流计算系统带来的延迟. 而计算资源分配的资源提供方式[46]同样有利于加速资源提供过程, 因为其避免了申请和释放资源带来的延迟. 通过这种资源提供方式, Elasticutor[46]实现了流计算系统中实时的低延迟资源调度. 第 2 种优化延迟的方法是减少资源调整的次数. 流计算系统在资源调度过程中可能需要多次调整资源以使得应用达到理想运行状态. 反复多次的资源调整会导致资源提供过程带来额外延迟, 因此流计算系统可以减少需要的资源调整次数, 以降低资源调度带来的延迟. DS2[41]是一个致力于实现流计算系统中资源精确提供的资源调度框架. 通过准确测定流计算应用中各个算子需要的资源数量, DS2 只需要一次资源提供过程就可以为流计算应用准确地调度资源. 相对于传统资源调度方式所需要的反复调整过程, DS2 通过减少资源调整次数实现了降低资源提供过程延迟的目的. 更多地, 由于流计算系统通常运行多个查询, 而每个算子对实时性的要求并不相同. 在资源调度中优先满足那些即将输出的算子可以避免它们违反延迟约束, 从而改进流计算系统的延迟. Cameo[42]的资源提供方式是通过为算子分配工作线程来实现的. 由于这种基于线程的资源提供方式可以很容易在各算子之间转移资源, Cameo 实现了考虑实时性的优先级调度. Cameo 估算每个算子需要的处理时间和资源, 并优先为那些可能违反延迟约束的算子调度资源, 尽可能降低了系统违反延迟约束的次数. Meces[48]则考虑了同一个算子的多个并行实例之间的调度优先级; Meces 基于流计算系统中常见的数据倾斜现象, 优先调度进行热点数据处理的算子, 避免了大量数据在调度过程中阻塞, 降低了资源调度带来的延迟.

4.3 小 结

本节简述了流计算系统进行资源调度的两个过程, 即资源监控和资源提供, 并在此基础上介绍了流计算系统在这两个过程中的相关技术, 研究它们如何满足流计算系统减少资源浪费和降低服务延迟的调度目标, 并讨论了这些技术在不同环境中的可拓展性. 资源调度技术的总结如表 5 所示.

表 5 资源调度技术总结

资源调度技术	实现方式	减少资源浪费	降低服务延迟	可拓展性
资源监控	资源利用率	√	×	—
	算子运行时间	√	√	—
	端到端延迟	×	√	—
资源提供	增加节点	×	×	√
	增加节点资源	√	√	√
	调整线程优先级	√	√	×

在资源监控过程中, 监控资源利用率是一种简单有效的办法. 为了实现更细粒度的调度以减少资源浪费, 流计算系统中通常采用分别监测单个节点的资源利用率代替系统的平均资源利用率. 但是, 只监测资源利用率无法得知当前资源配置情况下流计算系统的延迟情况, 也就无法通过资源调度解决对延迟约束的违反. 因此, 一些流计算系统采用监测端到端延迟的技术来保证流计算应用的低延迟. 算子运行时间可以精确反映流计算应用中各个任务对资源的需求, 有效减少资源浪费并降低延迟; 但频繁监测算子运行时间可能带来额外性能开销.

在资源提供过程中, 增加节点的方法具有最好的可拓展性, 但是提供过程延迟较长, 且容易造成资源浪费. 增加节点内资源的办法可以实现更灵活的资源调度, 降低资源的浪费. 相比于增加节点, 增加节点资源的方法速度更快, 带来的延迟更小. 然而增加节点资源的方法依赖于硬件资源的架构, 在可拓展性上存在一定限制. 最后, 调整线程优先级的方法可以实现最快速的资源提供, 但几乎不具备任何可拓展性.

5 故障容错

对于运行在复杂的分布式集群中的大规模数据处理系统来说, 遭遇故障是十分常见的. 这一点对于流计算系统来说也是一样. 在流计算应用的长期运行中, 流计算系统可能遭遇软件错误、节点故障、网络通信中断等各类故障, 这些故障会造成流计算服务暂时或永久中断. 因此, 流计算系统需要设计有效的故障容错机制, 以便在遭遇故障时快速地恢复正常运行, 减少故障对于流计算服务的影响. 故障容错机制主要分为两个部分: 首先, 流计算系统需要对受到故障影响的流计算任务进行恢复. 流计算系统需要重启任务、重启节点或将任务迁移至其他节点, 以保证流计算任务可以继续运行. 其次, 流计算系统需要对有状态算子的状态进行恢复. 流计算系统需要利用检查点或日志等技术, 将流计算任务的状态恢复到故障发生之前, 从而保证流计算应用结果的正确性.

批处理系统已经形成了一套较为成熟的基于检查点的故障容错机制, 并在 MapReduce、Spark 等大规模处理系统中得到了广泛的应用. 但是这种故障容错机制在流计算系统中遭遇了新的挑战. 首先, 相比批处理系统, 流计算系统对于延迟有着更加严格的要求. 批处理系统的恢复过程通常不会对总运行时间产生显著影响, 但对于流计算系统而言, 任务恢复会造成流计算服务的暂时中止, 产生不可忽视的延迟, 使得流计算系统的延迟严重恶化. 因此, 流计算系统需要根据其延迟要求, 设计故障容错机制, 降低任务恢复过程的延迟. 其次, 批处理系统通常使用基于检查点的备份恢复机制来保证故障恢复之后的一致性. 但由于流数据的无界性, 在流计算系统中直接应用批处理系统中的检查点机制来备份和恢复状态必须在每次处理数据后都进行检查点备份, 导致正常运行时的大量备份开销. 因此, 流计算系统需要针对流数据的特性设计状态备份恢复机制, 在不造成正常运行时过多额外开销的情况下, 保证故障后状态能够正常恢复, 使得流计算应用可以继续正常运行.

5.1 任务恢复

流计算系统在故障容错过程中要面对的第 1 个挑战是如何设计任务恢复技术. 任务恢复技术主要需要考虑两方面的问题. 第一, 在系统正常运行, 即未发生故障期间, 任务恢复技术应产生较少的资源开销, 从而减少资源浪费, 提高系统的资源利用率. 第二, 在系统发生故障时, 流计算系统需要尽快完成任务恢复, 降低流计算服务中止运行的时间.

大规模数据处理系统中应用于任务恢复的故障容错技术主要分为两种: 被动容错和主动容错. 被动容错是指在系统正常运行时不进行任何操作, 在检测到系统遭遇故障后再进行处理的容错技术. 典型的处理方式包括重启

系统、重启节点和将任务迁移至新节点等. 早期的流计算系统在进行被动容错时通常指定一个尚未启动的节点作为备用节点. 流计算系统会定期将检查点备份至该分布式文件系统或该节点内. 当流计算系统在主节点上遭遇故障时, 系统会启动该备用节点, 并利用先前备份的检查点恢复处理. SGuard[49]实现了这样一种被动容错的技术. 通过定期将检查点备份至分布式文件系统中, SGuard 可以在正常运行过程中几乎不产生任何的额外的内存开销. 然而在遭遇故障后, SGuard 需要启动一个新的节点并读取检查点以继续处理. 这种重启故障节点或启动其他节点的被动容错技术在节点恢复过程中产生较大的延迟, 不利于降低流计算服务的中止时长. 为解决这一问题, 一些流计算系统将受故障影响的任务迁移至其他节点, 以避免等待节点启动而造成的延迟. 例如, SEEP[47]将一个算子的多个并行实例互相作为故障容错的备份. 在故障发生时, SEEP 不会立即中断流计算应用, 等待节点和软件的故障恢复, 而是通过调整上游分区, 将发生故障的任务需要处理的数据, 转移到其他并行实例中, 从而降低了恢复延迟, 避免了流计算服务的长时间中断. ChronoStream[50]也采用了和 SEEP 类似的容错技术, 但 ChronoStream 设计了一种计算实例的放置机制. 该机制将同一个算子的多个并行实例尽量分布在不同的节点上, 以最大化系统遭遇节点故障后的可用程度. 将任务迁移至其他节点可以有效降低节点恢复过程的延迟, 但迁移的目标节点需要运行更多的流计算任务. 这为目标节点带来了更大的计算和内存开销, 可能造成流计算系统长期运行过程中的性能下降. 被动容错的优点在于在系统正常运行时不会为引入额外的资源开销. 但由于被动容错技术在故障发生之后才开始工作, 因此其故障恢复速度通常较慢. 这可能引起流计算服务的长时间中断, 在对响应速度具有严格要求的流计算场景中难以适用.

相比于被动容错技术在故障发生后才开始工作, 主动容错技术则是在流计算系统正常运行时就为故障做准备. 主动容错技术在系统正常运行时为每个任务启动一个副本任务, 副本任务始终处于待机状态, 准备随时接管主任务的工作. 当主任务遭遇故障时, 只需要令副本任务成为新的主任务即可完成任务恢复. 由于主动容错技术在故障恢复阶段只需要让副本任务接替主任务进行处理, 因此主动容错技术在恢复过程中具有低延迟的优点. 但是, 相比于被动容错技术, 由于主动容错技术需要启动一个待机的副本任务, 产生了一定的资源开销, 因此可能会减少主任务可用的资源并降低系统的运行效率. Borealis[51]实现了一个基于副本的主动容错技术. 在主任务所在节点遭遇故障后, Borealis 可以迅速将处理工作移交给副本任务, 从而实现快速的任务恢复. 然而, 由于需要为所有任务启动一个副本任务, 这些副本任务在系统正常运行时产生了显著的内存和计算资源开销, 不利于系统性能的提升. 为了减少主动容错技术为在系统正常运行时产生的额外资源开销, 一些工作对流计算系统中的主动容错技术进行了优化. Shah 等人[52]也采用了类似于 Borealis 主动容错技术. 有所不同的是, Shah 等人对恢复过程进行了优化: 他们允许流计算系统在任务恢复期间继续运行. 也就是说, 对于其他不受故障影响的任务来说, 它们可以继续进行处理而不必等待流计算系统的故障完全恢复, 这有助于进一步降低任务恢复为流计算应用带来的延迟. 主动容错技术虽然可以实现在故障发生后进行快速的任务恢复, 但其在正常运行期间需要额外的资源, 这种额外资源开销为流计算系统造成的性能损害常常不可忽视. 为了降低主动容错技术在正常运行期间的额外资源开销, 一些工作提出了部分主动容错技术来减少需要进行故障容错的流计算任务数量. IBM 在他们的流计算系统 Stream 中实现了一种部分主动容错技术[53]. IBM Stream 为开发者提供了一组用于主动容错的编程接口, 使开发者可以在开发应用时指定需要进行主动容错的流计算任务. 流计算系统可以根据开发者的指令对应用中的关键任务创建副本, 从而保证这些任务在故障发生后能进行快速恢复, 同时又避免了对系统中的所有任务都创建一个活跃的副本, 造成正常运行期间过多的资源浪费. Su 等人在他们的工作中提出了一种容错机制[54], 即通过定义算子保真度来衡量每个算子输出结果对其他算子的影响. 算子保真度较高的算子在遭遇故障时会对流计算应用的整体运行造成较大影响. 因此流计算系统为执行这些算子的任务建立副本, 利用主动容错技术降低这些算子的故障时恢复时间, 并对其他任务应用被动容错技术来减少正常运行时资源消耗.

5.2 状态恢复

流计算系统在对无状态算子的故障容错过程中需要考虑如何降低恢复延迟. 但对于流计算应用中广泛存在的有状态算子来说, 故障容错中还存在另一个问题, 即如何在系统正常运行时对状态进行备份, 并在故障发生后正确恢复状态. 在系统正常运行时, 状态恢复的目标是减少状态备份过程对系统正常运行性能的影响. 在系统发生故障

后, 状态恢复的目标包括两点. 一方面是降低完成状态恢复所需的时间, 另一方面是提供一致性保证, 即保证系统内所有状态都恢复到故障前的一致状态.

在数据管理系统中, 有两类基础的故障容错技术, 分别是基于检查点和基于重放的故障容错技术. 由于流计算任务连续处理的特性, 任务内的状态处于持续变化之中, 因而流计算系统很难单独使用其中一种技术. 单独使用基于检查点的技术在正常运行时, 要求流计算任务每一次状态改变都进行检查点的写入, 其性能开销对于流计算系统而言过大. 单独使用基于重放的技术则要求流计算任务在遭遇故障时重放所有输入数据, 即重新处理所有流数据, 带来不可接受的恢复延迟. 因而, 流计算系统通常将二者结合使用: 在正常运行时, 每隔一段时间为系统内的状态生成检查点; 对于那些尚未被写入检查点的状态更改, 则将原始数据暂时保存. 遭遇故障后, 流计算系统先读取最新的检查点, 并重放并再次处理被保存的原始数据, 以达到恢复状态的目的. 保存用于重放的数据通常由数据源或流计算系统的输入算子负责, 而流计算系统则负责定期生成检查点. 在生成检查点时, 流计算系统主要考虑两方面问题: 第一, 该以何种粒度生成流计算应用的检查点. 第二, 该在何种位置对流计算应用的检查点进行备份.

由于流计算应用中的状态是和特定的算子关联的, 一个简单的检查点生成思路是在正常运行时为每个算子独立地生成含有状态的检查点. 在恢复时先将每个算子的状态恢复至其最新的检查点, 再通过让进度落后的算子追赶的方式来达到全局状态一致. 这种局部的检查点生成方式在流计算系统中应用广泛. 例如, MillWheel[55]对每条状态记录单独备份. 更具体地说, MillWheel 将状态保存为键值对的形式, 并在每一次状态更新后都进行备份. StreamScope[56]以单个流计算任务为单位, 将一个流计算任务内的所有有状态算子的状态组合并生成一个检查点. 值得注意的是, MillWheel 和 StreamScope 在状态每次更新后都备份了检查点, 这样做的好处是故障发生时各个任务可以直接读取最新的状态检查点进行恢复. 但这种频繁的生成检查点的操作对流计算系统正常运行时性能产生了严重影响, 不适用于输入数据规模较大、处理延迟要求较高的场景. 利用了主动容错机制的系统常常通过让副本任务和主任务处理相同内容的方式, 使得二者的状态保持一致, 从而实现状态备份[51,53]. 副本任务进行状态备份对主任务的运行不会产生任何影响, 并且可以始终保持副本任务中的状态与主任务保持一致, 有利于在故障发生时迅速进行状态恢复. 然而, 由于副本任务必须和主任务进行相同的处理, 这种状态备份机制产生了额外的计算开销, 对系统总体的资源利用率造成了严重的影响. SEEP[47]中的一个任务每隔一段时间独立地生成状态检查点. 在此基础上, SEEP 通过暂时保存该任务的输入数据来应对两次检查点间隔中发生故障的情况. 在恢复过程中, 每个任务独立地读取状态检查点并进行恢复, 再读取并重新处理那些更改未被检查点记录的数据. 通过这种方法, SEEP 可以将所有任务都恢复到故障发生前的状态. ChronoStream[50]采用了类似的, 即让每个任务独立进行检查点备份和恢复的机制. 与 SEEP 有所不同的是, ChronoStream 假定每个算子的输入输出顺序严格保持一致. 在这样的基础上, ChronoStream 中的算子只需要用数据的编号记录处理进度, 并从头重放这些数据. 避免了像 SEEP 一样需要为每个算子保存一部分尚未被应用到检查点中的数据. 然而, Silvestre 等人[57]指出, 上述方法只有在算子满足确定性时才能保证状态恢复的一致性. 如果算子处理逻辑中包含随机数或调用外部函数等情况, 上述方法恢复的状态就无法保持全局一致. 这会导致系统必须重新处理所有数据, 或者忍受部分错误的结果. 为解决这一问题, 他们在 Clonos 中将不满足确定性的调用记录为日志, 将存在不确定性的调用转为确定的日志, 从而保证每次读取得到相同的结果, 但这种方式带来了更大的备份和还原开销.

由于局部独立检查点可能带来的一致性问题, 一些流计算系统考虑采用全局统一的方式生成检查点. 顾名思义, 流计算系统在生成检查点时要为系统中所有任务生成一份状态一致的检查点. 最直观的全局统一检查点生成方法是暂时中止流数据进入系统, 并等待流计算系统将系统内的剩余数据全部处理完毕. 这时流计算系统内所有任务的状态均保持一致, 可以生成全局检查点. 但该方法会频繁终止流计算系统的正常运行, 这是不可接受的. 现今为流计算系统所采用的全局统一检查点生成方法通常基于 Chandy-Lamport 分布式快照算法. 其中的代表就是 Flink 所采用的状态备份技术[58]. Flink 在数据源生成一些分隔符, 将流计算系统内的数据分为不同的历元 (epoch), 不同的历元之间的数据不可以越过分隔符. 每当流计算任务接收到一个分隔符时, 代表这一历元的所有数据都已经处理完毕, 该任务即开始生成检查点. 将所有任务生成的这一历元数据的检查点汇总, 就得到了一个全局统一的检查点. 这种方法有效解决了难以生成全局一致的状态检查点的情况, 为全局统一检查点在流计算系统中的应用

提供了基础. 由于在生成检查点的过程中, 流计算系统保证了各算子的状态是一致的, 因而在恢复过程中, 所有任务都可以直接恢复到一致的状态, 不需要再进行处理进度的同步. 但是, 相比于局部独立检查点, 全局统一检查点存在两个劣势: 首先, 在状态备份过程中, 由于每次备份都需要生成一个包含全局所有状态的检查点, 因此读写开销相比局部检查点更大. 其次, 在状态恢复过程中, 由于需要保证全局一致性, 因此所有的算子, 包括未受到故障影响的算子, 都需要读取检查点并回滚状态. 这加大了恢复过程中需要恢复的状态数量, 并且使得未受到故障影响的算子进行不必要的重复处理. 因此生成全局统一检查点的间隔可以减少生成检查点过程中的性能开销, 但在遭遇故障时会加剧回滚后重新计算的性能损失. 平衡这两部分性能对于全局一致检查点技术的应用十分重要. Jayasekara 等人[59]研究了全局一致检查点生成间隔对于流计算系统性能的影响, 并提出了一种优化生成间隔的方法用以改善流计算系统的性能. 他们的实验结果显示, 优化延迟间隔后的检查点技术不仅没有影响故障后恢复的性能, 还大大改善了正常运行时的性能. 相比于默认的检查点技术, 优化后的检查点技术在正常运行时提高了 10%–40% 的吞吐量, 并降低了一半以上的端到端延迟.

流计算系统在生成检查点时需要考虑的第 2 个问题是应该在什么位置备份检查点. 出于最大程度地保护检查点不受故障影响的想法, 大多数流计算系统采用了将检查点保存在外部节点上. 一些流计算系统采用了运行在外部节点上的数据库: MillWheel[55]将状态保存在了一个外部的键值数据库中, 而 S-Store[60]则将状态保存至一个关系型数据库中. 由于外部数据库自身具有容错机制, 因此可以最大程度地保证状态不在故障中丢失. 但受限于数据库的存储结构, 流计算系统很难在其中保存体积较大的检查点. 其次, 外部数据库的读写速度瓶颈也会限制流计算系统正常运行和故障恢复时的性能. 另一种选择则是将数据保存在分布式文件系统中. 自从 Kwon 等人[49]在 2008 年提出用分布式文件系统保存流计算状态的检查点以来, 分布式文件系统成为许多流计算系统保存检查点的一个重要位置. 诸如 Flink、Kafka Streaming[61]等流计算系统都采用分布式文件系统来备份状态. 分布式文件系统的一大优势在于其容量大, 且大文件的读写性能较好, 相较于数据库更适合用于保存流计算系统生成的状态. 同时, 分布式文件系统自身具有的容错机制也可以应对外部节点发生的一些故障. 外部数据库和外部分布式文件系统都为流计算系统的状态检查点提供了很好的容错保障. 但与此同时, 流计算系统在正常运行时和故障恢复时都需要大量读写外部节点, 带来了大量的跨节点、跨机架、跨集群的网络通讯开销, 严重限制了流计算系统运行和恢复时的性能. 因此, 一些流计算系统采用流计算运行的内部节点保存状态, 从而加速写入和读取检查点的过程. 利用内部节点保存检查点的第 1 种方式是将当前节点的状态检查点保存在其他节点上. 例如, 在 SEEP[47]中. 每个任务负责保存下游任务生成的检查点及发往该任务的数据. 在故障发生后, 重启的新任务可以直接从上游读取状态和数据, 实现快速的状态恢复. ChronoStream[50]则将状态保存在兄弟节点, 即运行同一逻辑算子的并行子任务的节点中. 在故障发生后, 该逻辑算子的其他子任务可以通过修改路由表的方式接管故障任务的处理工作, 并在他们的节点上快速读取并恢复状态. 另一种集群内的检查点保存方式是将检查点保存在当前节点的文件系统中. 这种方法起源于 Samza[62]观察到流计算系统中发生的大部分故障是软件故障而非硬件故障, 因此将检查点保存在独立于流计算系统的本机文件系统中在多数情况就可以应对故障. 读写本地文件系统避免了前述工作中需要和其他节点或集群进行通讯带来了大量网络开销.

5.3 小 结

本节介绍流计算系统中用于故障容错方面的技术. 如表 6 所示, 流计算系统的故障容错过程可以分为两部分. 首先, 流计算系统需要对那些遭遇故障的流计算任务进行恢复, 使得它们可以继续处理输入数据, 从而保证流计算应用的继续运行. 其次, 流计算系统需要恢复有状态算子在故障中丢失的状态, 使得算子状态恢复到故障前, 从而保证流计算应用输出结果的正确性.

在任务恢复过程中可选的容错技术有 3 种. 其中, 被动容错在故障发生后才开始工作, 对系统的正常运行性能几乎没有影响, 但恢复时间较长, 提高了故障给流计算系统的带来的延迟. 主动容错通过建立运行的副本的方式降低了恢复延迟, 但在系统正常运行时产生了额外资源占用, 影响系统性能. 部分被容容错结合二者的优点, 对关键任务应用主动容错, 在降低正常运行时资源占用的情况下降低了恢复延迟.

状态恢复要解决的两个问题是状态检查点的生成粒度和备份位置. 局部独立地生成检查点有利于提高备份和

恢复过程的延迟, 但可能带来状态不一致性的问题. 全局统一的检查点可以弥补状态一致性的缺陷, 但代价是降低了备份和恢复过程的性能. 流计算系统中的大多数故障容错技术将检查点备份在集群外部, 来保证故障发生时检查点不受影响. 但这种做法需要频繁进行跨集群网络通讯, 降低了备份和恢复过程的性能. 因此一些流计算系统采用了在集群内进行检查点备份的技术, 牺牲了一定的一致性保证换取了更优的备份和恢复性能.

表 6 故障容错技术总结

容错内容	容错技术		正常运行性能	恢复过程	
				恢复延迟	一致性保证
任务恢复	被动容错		√	×	—
	主动容错		×	√	—
	部分被动容错		√	√	—
状态恢复	检查点粒度	局部独立	√	√	×
		全局统一	×	×	√
	检查点位置	集群内部	√	√	×
		集群外部	×	×	√

6 研究展望与未来趋势

流计算系统经过长期的研究和发展, 已经产生了丰富的技术和成熟的系统, 并且在数据规模和实时性激增的当下和未来都有着广泛的应用场景. 本节将结合前文的技术总结, 简要讨论流流计算系统未来可能的发展方向.

1) 实现可拓展性和性能优化的统一. 目前主要存在两类流计算系统, 第 1 类以 Storm、Flink 为代表, 运行在 JVM 中, 因而具备很好的可拓展性. 另一类以 StreamBox、Grizzly 为代表, 专注于在单机上实现更高性能. 前者虽然可以通过增加节点来提升处理能力, 但未能充分利用硬件, 致使系统性能远低于后者[63]; 而后者则面临难以横向拓展的问题. 因此, 未来的流计算系统可以结合二者的优势, 在保证可拓展性的前提下充分利用硬件资源, 实现可拓展性和性能优化的统一.

2) 实现执行计划和计算资源的自动调整. 目前流计算系统中存在众多计划调整和资源调度的技术, 可以在一定程度上应对流数据的动态变化. 然而, 目前的自动调整技术仍然依赖于开发者的人工干预, 不仅加大了开发和维护的难度, 也限制了流计算系统应对大幅数据流变化的能力. 因此, 未来的流计算系统应能够在不增加用户配置复杂度的情况下, 借鉴数据库参数自动调优技术, 实现执行计划和计算资源的自动配置, 以应对生产环境中流数据可能存在的大幅变化.

3) 实现对异构硬件和新硬件的充分利用. 目前大部分流计算系统主要利用 CPU 和内存进行计算和存储. 然而, 流数据规模的进一步加大可能对依赖于 CPU 和内存的计算存储能力提出挑战, 而新硬件的出现也为进一步提升流计算系统性能提供了可能. 目前存在一些利用 GPU[64]或高速缓存[65]优化流计算系统性能的技术, 但其适用场景都较为局限. 未来的流计算系统应能综合利用各种计算通讯和存储的新硬件, 如 FPGA、RDMA、非易失性内存, 全面优化流计算系统的计算性能、通讯开销以及状态容错等.

4) 实现高效的批流融合计算. 批流融合技术指使用一个系统同时进行批数据和流数据处理的技术. 目前的批流融合技术主要分为编程接口的统一, 如 Apache Beam, 以及执行引擎的统一, 如 Spark Streaming、Flink. 但这两类技术都存在计算效率不高的问题. 对于前者, 由于系统将统一的编程语言翻译成不同的底层代码在不同的计算引擎上执行, 因而浪费了针对系统的特性进行优化的机会. 对于后者, 基于批处理执行引擎的微批处理会为流计算引入额外延迟, 而基于流计算执行引擎的连续处理则在批处理中损害了吞吐量. 因此, 未来的流计算系统应对批处理和流计算进行联合优化, 从而提供高效的批流融合计算服务.

7 总 结

本文首先介绍了流数据的基本特征和流计算系统的发展历史, 从而引出了目前流计算系统的设计和优化中的

存在的挑战. 之后, 本文根据流计算系统的结构将相关技术分为 4 类, 分别介绍和分析了这些技术, 以及它们是如何解决前述挑战的. 最后, 本文结合上述分析, 展望了流计算系统未来的研究方向和发展趋势.

References:

[1] Cui B, Gao J, Tong YX, Xu JQ, Zhang DX, Zou L. Progress and trend in novel data management system. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 164–193 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5646.htm [doi: 10.13328/j.cnki.jos.005646]

[2] Li S, Huang YZ, Chen HY. Review of big data stream computing system study. Journal of Information Engineering University, 2016, 17(1): 88–92 (in Chinese with English abstract). [doi: 10.3969/j.issn.1671-0673.2016.01.017]

[3] Qi KY, Zhao ZF, Fang J, Ma Q. Real-time processing for high speed data stream over large scale data. Chinese Journal of Computers, 2012, 35(3): 477–490 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2012.00477]

[4] Wu F, Lu ZQ, Zhao WY. Application of real-time flow computation in insurance decision system. Computer and Digital Engineering, 2020, 48(6): 1324–1327 (in Chinese with English abstract). [doi: 10.3969/j.issn.1672-9722.2020.06.012]

[5] Terry D, Goldberg D, Nichols D, Oki B. Continuous queries over append-only databases. ACM SIGMOD Record, 1992, 21(2): 321–330. [doi: 10.1145/141484.130333]

[6] Cranor C, Johnson T, Spataschek O, Shkapenyuk V. GigaScope: A stream database for network applications. In: Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data. San Diego: Association for Computing Machinery, 2003. 647–651. [doi: 10.1145/872757.872838]

[7] Abadi DJ, Carney D, Çetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S. Aurora: A new model and architecture for data stream management. The VLDB Journal, 2003, 12(2): 120–139. [doi: 10.1007/s00778-003-0095-z]

[8] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1): 107–113. [doi: 10.1145/1327452.1327492]

[9] Toshniwal A, Taneja S, Shukla A, Ramasamy K, Patel JM, Kulkarni S, Jackson J, Gade K, Fu MS, Donham J, Bhagat N, Mittal S, Ryaboy D. Storm@Twitter. In: Proc. of the 2014 ACM SIGMOD Int'l Conf. on Management of Data. Snowbird: Association for Computing Machinery, 2014.147–156. [doi: 10.1145/2588555.2595641]

[10] Zaharia M, Das T, Li HY, Hunter T, Shenker S, Stoica I. Discretized streams: A fault-tolerant model for scalable stream processing. Technical Report UCB/EECS-2012-259, Berkeley: University of California, 2012.

[11] Sun DW, Zhang GY, Zheng WM. Big data stream computing: Technologies and instances. Ruan Jian Xue Bao/Journal of Software, 2014, 25(4): 839–862 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4558.htm [doi: 10.13328/j.cnki.jos.004558]

[12] Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache Flink™: Stream and batch processing in a single engine. IEEE Data Engineering Bulletin, 2015, 38(4): 28–38.

[13] Miao HY, Park H, Jeon M, Pekhimenko G, McKinley KS, Lin FX. StreamBox: Modern stream processing on a multicore machine. In: Proc. of the 2017 USENIX Annual Technical Conf. Santa Clara: USENIX Association, 2017. 617–629.

[14] Akidau T, Bradshaw R, Chambers C, Chernyak S, Fernández-Moctezuma RJ, Lax R, McVeety S, Mills D, Perry F, Schmidt E, Whittle S. The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proc. of the VLDB Endowment, 2015, 8(12): 1792–1803. [doi: 10.14778/2824032.2824076]

[15] Arasu A, Babcock B, Babu S, Datar M, Ito K, Nishizawa I, Rosenstein J, Widom J. STREAM: The stanford stream data manager (demonstration description). In: Proc. of the 2003 ACM SIGMOD Int'l Conf. on Management of Data. San Diego: Association for Computing Machinery, 2003. 665. [doi: 10.1145/872757.872854]

[16] Arasu A, Babu S, Widom J. The CQL continuous query language: Semantic foundations and query execution. The VLDB Journal, 2006, 15(2): 121–142. [doi: 10.1007/s00778-004-0147-z]

[17] Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng XR, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M. Spark SQL: Relational data processing in spark. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: Association for Computing Machinery, 2015. 1383–1394. [doi: 10.1145/2723372.2742797]

[18] Traub J, Grulich PM, Cuéllar AR, Breß S, Katsifodimos A, Rabl T, Markl V. Scotty: General and efficient open-source window aggregation for stream processing systems. ACM Trans. on Database Systems, 2021, 46(1): 1. [doi: 10.1145/3433675]

[19] Ananthanarayanan R, Basker V, Das S, Gupta A, Jiang HF, Qiu TB, Reznichenko A, Ryabkov D, Singh M, Venkataraman S. Photon: Fault-tolerant and scalable joining of continuous data streams. In: Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2013. 577–588. [doi: 10.1145/2463676.2465272]

[20] Gomes AS, Oliveirinha J, Cardoso P, Bizarro P. Railgun: Managing large streaming windows under mad requirements. Proc. of the VLDB Endowment, 2021, 14(12): 3069–3082. [doi: 10.14778/3476311.3476384]

[21] Li J, Maier D, Tufte K, Papadimos V, Tucker PA. Semantics and evaluation techniques for window aggregates in data streams. In: Proc. of the 2005 ACM SIGMOD Int'l Conf. on Management of Data. Baltimore: Association for Computing Machinery, 2005. 311–322. [doi: 10.1145/1066157.1066193]

[22] Tatbul N, Çetintemel U, Zdonik S, Cherniack M, Stonebraker M. Load shedding in a data stream manager. In: Proc. of the 29th Int'l Conf. on Very Large Data Bases. Berlin: VLDB Endowment, 2003. 309–320. [doi: 10.5555/1315451.1315479]

[23] Ji YZ, Zhou HJ, Jerzak Z, Nica A, Hackenbroich G, Fetzer C. Quality-driven processing of sliding window aggregates over out-of-order data streams. In: Proc. of the 9th ACM Int'l Conf. on Distributed Event-based Systems. Oslo: Association for Computing Machinery, 2015. 68–79. [doi: 10.1145/2675743.2771828]

[24] Katsipoulakis NR, Labrinidis A, Chrysanthis PK. SPEAr: Expediting stream processing with accuracy guarantees. In: Proc. of the 36th IEEE Int'l Conf. on Data Engineering. Dallas: IEEE, 2020.1105–1116. [doi: 10.1109/ICDE48307.2020.00100]

[25] Rivetti N, Anceaume E, Busnel Y, Querzoni L, Sericola B. Online scheduling for shuffle grouping in distributed stream processing systems. In: Proc. of the 17th Int'l Middleware Conf. Trento: Association for Computing Machinery, 2016. 11. [doi: 10.1145/2988336.2988347]

[26] Shah MA, Hellerstein JM, Chandrasekaran S, Franklin MJ. Flux: An adaptive partitioning operator for continuous query systems. In: Proc. of the 19th Int'l Conf. on Data Engineering. Bangalore: IEEE, 2003. 25–36. [doi: 10.1109/ICDE.2003.1260779]

[27] Balkesen C, Tatbul N, Özsu MT. Adaptive input admission and management for parallel stream processing. In: Proc. of the 7th ACM Int'l Conf. on Distributed Event-based Systems. Arlington: Association for Computing Machinery, 2013. 15–26. [doi: 10.1145/2488222.2488258]

[28] Gedik B. Partitioning functions for stateful data parallelism in stream processing. The VLDB Journal, 2014, 23(4): 517–539. [doi: 10.1007/s00778-013-0335-9]

[29] Nasir MAU, de Francisci Morales G, García-Soriano D, Kourtellis N, Serafini M. The power of both choices: Practical load balancing for distributed stream processing engines. In: Proc. of the 31st IEEE Int'l Conf. on Data Engineering. Seoul: IEEE, 2015. 137–148. [doi: 10.1109/ICDE.2015.7113279]

[30] Nasir MAU, de Francisci Morales G, Kourtellis N, Serafini M. When two choices are not enough: Balancing at scale in distributed stream processing. In: Proc. of the 32nd IEEE Int'l Conf. on Data Engineering. Helsinki: IEEE, 2016. 589–600. [doi: 10.1109/ICDE.2016.7498273]

[31] Katsipoulakis NR, Labrinidis A, Chrysanthis PK. A holistic view of stream partitioning costs. Proc. of the VLDB Endowment, 2017, 10(11): 1286–1297. [doi: 10.14778/3137628.3137639]

[32] Zhu YL, Rundensteiner EA, Heineman GT. Dynamic plan migration for continuous queries over data streams. In: Proc. of the 2004 ACM SIGMOD Int'l Conf. on Management of Data. Paris: Association for Computing Machinery, 2004. 431–442. [doi: 10.1145/1007568.1007617]

[33] Gedik B, Andrade H, Wu KL. A code generation approach to optimizing high-performance distributed data stream processing. In: Proc. of the 18th ACM Conf. on Information and Knowledge Management. Hong Kong: Association for Computing Machinery, 2009. 847–856. [doi: 10.1145/1645953.1646061]

[34] Ding LP, Works K, Rundensteiner EA. Semantic stream query optimization exploiting dynamic metadata. In: Proc. of the 27th IEEE Int'l Conf. on Data Engineering. Hannover: IEEE, 2011. 111–122. [doi: 10.1109/ICDE.2011.5767840]

[35] Grulich PM, Sebastian B, Zeuch S, Traub J, von Bleichert J, Chen ZX, Rabl T, Markl V. Grizzly: Efficient stream processing through adaptive query compilation. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: Association for Computing Machinery, 2020. 2487–2503. [doi: 10.1145/3318464.3389739]

[36] Mai L, Zeng K, Potharaju R, Xu L, Suh S, Venkataraman S, Costa P, Kim T, Muthukrishnan S, Kuppa V, Dhulipalla S, Rao S. Chi: A scalable and programmable control plane for distributed stream processing systems. Proc. of the VLDB Endowment, 2018, 11(10): 1303–1316. [doi: 10.14778/3231751.3231765]

[37] Mao YC, Huang Y, Tian RX, Wang X, Ma RTB. Trisk: Task-centric data stream reconfiguration. In: Proc. of the 2021 ACM Symp. on Cloud Computing. Seattle: Association for Computing Machinery, 2021. 214–228. [doi: 10.1145/3472883.3487010]

[38] Heinz C, Krämer J, Riemenschneider T, Seeger B. Toward simulation-based optimization in data stream management systems. In: Proc. of the 24th IEEE Int'l Conf. on Data Engineering. Cancun: IEEE, 2008. 1580–1583. [doi: 10.1109/ICDE.2008.4497626]

[39] Mei Y, Cheng LW, Talwar V, Levin MY, Jacques-Silva G, Simha N, Banerjee A, Smith B, Williamson T, Yilmaz S, Chen WT, Chen GJ. Turbine: Facebook's service management platform for stream processing. In: Proc. of the 36th IEEE Int'l Conf. on Data Engineering.

Dallas: IEEE, 2020. 1591–1602. [doi: 10.1109/ICDE48307.2020.00141]

[40] Gulisano V, Jiménez-Peris R, Patiño-Martínez M, Soriente C, Valduriez P. StreamCloud: An elastic and scalable data streaming system. IEEE Trans. on Parallel and Distributed Systems, 2012, 23(12): 2351–2365. [doi: 10.1109/TPDS.2012.24]

[41] Kalavri V, Liagouris J, Hoffmann M, Dimitrova D, Forshaw M, Roscoe T. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2018. 783–798. [doi: 10.5555/3291168.3291226]

[42] Xu L, Venkataraman S, Gupta I, Mai L, Potharaju R. Move fast and meet deadlines: Fine-grained real-time stream processing with cameo. In: Proc. of the 18th USENIX Symp. on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2021. 389–405.

[43] Lohrmann B, Janacik P, Kao O. Elastic stream processing with latency guarantees. In: Proc. of the 35th IEEE Int'l Conf. on Distributed Computing Systems. Columbus: IEEE, 2015. 399–410. [doi: 10.1109/ICDCS.2015.48]

[44] Floratou A, Agrawal A, Graham B, Rao S, Ramasamy K. Dhalion: Self-regulating stream processing in heron. Proc. of the VLDB Endowment, 2017, 10(12): 1825–1836. [doi: 10.14778/3137765.3137786]

[45] Heinze T, Jerzak Z, Hackenbroich G, Fetzer C. Latency-aware elastic scaling for distributed data stream processing systems. In: Proc. of the 8th ACM Int'l Conf. on Distributed Event-based Systems. Mumbai: Association for Computing Machinery, 2014. 13–22. [doi: 10.1145/2611286.2611294]

[46] Wang L, Fu TZJ, Ma RTB, Winslett M, Zhang ZJ. Elasticutor: Rapid elasticity for realtime stateful stream processing. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: Association for Computing Machinery, 2019. 573–588. [doi: 10.1145/3299869.3319868]

[47] Fernandez RC, Migliavacca M, Kalyvianaki E, Pietzuch P. Integrating scale out and fault tolerance in stream processing using operator state management. In: Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2013. 725–736. [doi: 10.1145/2463676.2465282]

[48] Gu R, Yin H, Zhong WC, Yuan CF, Huang YH. Meces: Latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems. In: Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 539–556.

[49] Kwon Y, Balazinska M, Greenberg A. Fault-tolerant stream processing using a distributed, replicated file system. Proc. of the VLDB Endowment, 2008, 1(1): 574–585. [doi: 10.14778/1453856.1453920]

[50] Wu YJ, Tan KL. ChronoStream: Elastic stateful stream computation in the cloud. In: Proc. of the 31st IEEE Int'l Conf. on Data Engineering. Seoul: IEEE, 2015. 723–734. [doi: 10.1109/ICDE.2015.7113328]

[51] Balazinska M, Balakrishnan H, Madden S, Stonebraker M. Fault-tolerance in the borealis distributed stream processing system. In: Proc. of the 2005 ACM SIGMOD Int'l Conf on Management of Data. Baltimore: Association for Computing Machinery, 2005. 13–24. [doi: 10.1145/1066157.1066160]

[52] Shah MA, Hellerstein JM, Brewer E. Highly available, fault-tolerant, parallel dataflows. In: Proc. of the 2004 ACM SIGMOD Int'l Conf. on Management of Data. Paris: Association for Computing Machinery, 2004. 827–838. [doi: 10.1145/1007568.1007662]

[53] Jacques-Silva G, Zheng F, Debrunner D, Wu KL, Dogaru V, Johnson E, Spicer M, Sariyüce AE. Consistent regions: Guaranteed tuple processing in IBM streams. Proc. of the VLDB Endowment, 2016, 9(13): 1341–1352. [doi: 10.14778/3007263.3007272]

[54] Su L, Zhou YL. Tolerating correlated failures in massively parallel stream processing engines. In: Proc. of the 32nd IEEE Int'l Conf. on Data Engineering. Helsinki: IEEE, 2016. 517–528. [doi: 10.1109/ICDE.2016.7498267]

[55] Akidau T, Balikov A, Bekiroğlu K, Chernyak S, Haberman J, Lax R, McVeety S, Mills D, Nordstrom P, Whittle S. MillWheel: Fault-tolerant stream processing at internet scale. Proc. of the VLDB Endowment, 2013, 6(11): 1033–1044. [doi: 10.14778/2536222.2536229]

[56] Lin W, Fan HC, Qian ZP, Xu JW, Yang S, Zhou JR, Zhou LD. STREAMSCOPE: Continuous reliable distributed processing of big data streams. In: Proc. of the 13th USENIX Conf. on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2016. 439–453. [doi: 10.5555/2930611.2930640]

[57] Silvestre PF, Fragkoulis M, Spinellis D, Katsifodimos A. Clonos: Consistent causal recovery for highly-available streaming dataflows. In: Proc. of the 2021 Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2021. 1637–1650. [doi: 10.1145/3448016.3457320]

[58] Carbone P, Ewen S, Fóra G, Haridi S, Richter S, Tzoumas K. State management in Apache Flink®: Consistent stateful distributed stream processing. Proc. of the VLDB Endowment, 2017, 10(12): 1718–1729. [doi: 10.14778/3137765.3137777]

[59] Jayasekara S, Karunasekera S, Harwood A. Optimizing checkpoint-based fault-tolerance in distributed stream processing systems: Theory to practice. Software: Practice and Experience, 2022, 52(1): 296–315. [doi: 10.1002/spe.3021]

[60] Meehan J, Tatbul N, Zdonik S, Aslantas C, Cetintemel U, Du J, Kraska T, Madden S, Maier D, Pavlo A, Stonebraker M, Tufte K, Wang H.

S-store: Streaming meets transaction processing. Proc. of the VLDB Endowment, 2015, 8(13): 2134–2145. [doi: 10.14778/2831360.2831367]

[61] Wang GZ, Chen L, Dikshit A, Gustafson J, Chen BY, Sax MJ, Roesler J, Blee-Goldman S, Cadonna B, Mehta A, Madan V, Rao J. Consistency and completeness: Rethinking distributed stream processing in Apache Kafka. In: Proc. of the 2021 Int'l Conf. on Management of Data. New York: Association for Computing Machinery, 2021. 2602–2613. [doi: 10.1145/3448016.3457556]

[62] Noghabi SA, Paramasivam K, Pan Y, Ramesh N, Bringhurst J, Gupta I, Campbell RH. Samza: Stateful scalable stream processing at LinkedIn. Proc. of the VLDB Endowment, 2017, 10(12): 1634–1645. [doi: 10.14778/3137765.3137770]

[63] Breß S, Köcher B, Funke H, Zeuch S, Rabl T, Markl V. Generating custom code for efficient query execution on heterogeneous processors. The VLDB Journal, 2018, 27(6): 797–822. [doi: 10.1007/s00778-018-0512-y]

[64] Koliousis A, Weidlich M, Fernandez RC, Wolf AL, Costa P, Pietzuch P. SABER: Window-based hybrid stream processing for heterogeneous architectures. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: Association for Computing Machinery, 2016. 555–569. [doi: 10.1145/2882903.2882906]

[65] Zhang SH, He J, Zhou AC, He BS. BriskStream: Scaling data stream processing on shared-memory multicore architectures. In: Proc. of the 2019 Int'l Conf. on Management of Data. Amsterdam: Association for Computing Machinery, 2019. 705–722. [doi: 10.1145/3299869.3300067]

附中文参考文献:

[1] 崔斌, 高军, 童咏昕, 许建秋, 张东祥, 邹磊. 新型数据管理系统研究进展与趋势. 软件学报, 2019, 30(1): 164–193. http://www.jos.org.cn/1000-9825/5646.htm [doi: 10.13328/j.cnki.jos.005646]

[2] 李圣, 黄永忠, 陈海勇. 大数据流式计算系统研究综述. 信息工程大学学报, 2016, 17(1): 88–92. [doi: 10.3969/j.issn.1671-0673.2016.01.017]

[3] 亓开元, 赵卓峰, 房俊, 马强. 针对高速数据流的大规模数据实时处理方法. 计算机学报, 2012, 35(3): 477–490. [doi: 10.3724/SP.J.1016.2012.00477]

[4] 吴锋, 陆智卿, 赵文洋. 实时流计算在保险决策系统中的应用. 计算机与数字工程, 2020, 48(6): 1324–1327. [doi: 10.3969/j.issn.1672-9722.2020.06.012]

[11] 孙大为, 张广艳, 郑纬民. 大数据流式计算: 关键技术及系统实例. 软件学报, 2014, 25(4): 839–862. http://www.jos.org.cn/1000-9825/4558.htm [doi: 10.13328/j.cnki.jos.004558]

徐志榛(1998－), 男, 硕士生, 主要研究领域为流计算系统, 深度推荐系统.

陈梓浩(1996－), 男, 博士生, 主要研究领域为分布式机器学习系统.

徐辰(1988－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为大规模数据处理系统, 分布式机器学习系统, 面向新硬件的数据管理技术.

周傲英(1965－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为数据库, 数据管理, 数据驱动的教授教育学, 教育科技、物流科技等基于数据的应用科技.

丁光耀(1996－), 男, 博士生, 主要研究领域为大规模数据处理系统, 机器学习系统.