流处理引擎中算子的语义重叠

Vincenzo Gulisano 查尔姆斯理工大学 瑞典哥德堡
vincenzo.gulisano@chalmers.se

Alessandro Margara 米兰理工大学 意大利米兰
alessandro.margara@polimi.it

Marina Papatriantafilou 查尔姆斯理工大学 瑞典哥德堡
ptrianta@chalmers.se

arXiv:2303.00793v1 [cs.DC] 1 Mar 2023

摘要

流处理广泛应用于物联网到云的领域，用于从连续数据流中提取信息。流应用通常运行在专用的流处理引擎（SPE）中，这些引擎采用 DataFlow 模型，将此类应用定义为由算子构成的图；算子逐步转换数据，生成所需结果。由于算子可以独立部署和执行，DataFlow 模型支持并行性和分布式处理，从而使流应用具备可扩展性。

如今，流处理引擎种类繁多，各自拥有一套算子。在这种背景下，理解流处理引擎内部及彼此之间算子语义的重叠情况，以及哪些流处理引擎能够支持给定的应用，并非易事。我们通过形式化论证来解决这一问题：流处理引擎中的通用算子均可表示为单个精简聚合算子的组合，由此表明，任何能够运行此类算子组合的框架，都能运行针对最先进的流处理引擎定义的应用。该聚合算子仅依赖 DataFlow 模型的核心概念，例如按键进行数据分区和基于时间的窗口，并且对于其分析的每个窗口，最多只能输出一个值。除了形式化论证之外，我们还通过实证评估仅依赖此类算子的流处理引擎与提供特定算子实现的流处理引擎相比表现如何，并研究放宽每个窗口最多输出一个值这一约束后，更具表达能力的聚合算子对性能的影响。

这种共同基础的存在不仅意味着算子在流处理引擎内部及不同流处理引擎之间的可移植性，还为其他数据处理框架支持流应用定义了一组简洁的要求。

工件可用性：
源代码、数据和/或其他工件已在 https://github.com/vincenzo-gulisano/aggspes 上提供。

1 引言

流处理用于近实时地分析数据流，并在许多公司的数据处理栈中发挥着核心作用，使其能够基于最新数据进行知情决策。

专用软件组件，即流处理引擎（SPE），旨在应对定义流处理任务并以高效、可扩展的方式执行这些任务所带来的复杂性。为此，如今几乎所有分布式流处理引擎都基于 DataFlow 编程与执行模型 [1]。该模型将流式计算定义为由算子构成的图，其中每个算子都会将输入流转换为输出流，并将其传递给下游算子。这种方法简化了并行性和分布式处理，因为不同的算子可以部署在一台或多台机器上。同一算子的多个实例也可以并行运行，各自分析输入流的一部分。

尽管所有流处理引擎都基于相同的 DataFlow 基础，但每种流处理引擎都有自己的（庞大）算子库。要理解算子语义在同一流处理引擎内部及不同流处理引擎之间的重叠情况，从而确定哪些流处理引擎能够支持给定的流式应用，并非易事。揭示算子之间的语义重叠，并定义一组简洁的核心算子，使所有流处理引擎都能共享，并可用于实现其他算子的语义，从理论和实践角度来看都能带来显著益处：

(1) 在单个流处理引擎内及不同流处理引擎之间，这为多语言数据管道打开了大门，使各个算子或算子集合能够部署在最适合用户需求的流处理引擎中。(2) 开发者可以将流处理任务化简为等价任务，并采用最小化的算子集合，从而简化对应用正确性的验证与推理。(3) 通过支持这样一组核心算子，原本并非为流处理引擎而设计的大数据框架也能够支持基于Dataflow的应用。(4) 开发者可以将最小化的算子集合作为参考，用于定义、记录和验证新算子的语义。

本文表明，只需一个所有流处理引擎都实现的精简算子，以及 DataFlow 模型的几个基本概念（即事件时间、水印、按键数据分区和基本的非嵌套循环），就能将当前先进流处理引擎中的所有通用算子表示为该算子多个实例的组合。具体而言，我们仅考虑一种聚合算子：它分析基于时间的窗口，并在每次窗口滑动时最多输出一个结果。

我们为所提出的论断提供形式化证明，并通过实证评估，考察仅依赖聚合算子且仅包含上述基本 Dataflow构造的流处理引擎的性能。作为贡献的一部分，我们还表明，即便将流处理引擎的能力略微扩展到我们所确定的最小集合之外，也能显著提升性能。

总而言之，本文通过以下方面为支持复杂数据流水线中的流处理研究作出了贡献：

(1) 从形式上证明，当前先进流处理引擎中的所有通用算子，都可以表示为多个单一、极简聚合算子实例的组合，而该聚合算子仅依赖于 DataFlow 模型的基本概念；(2) 评估仅采用这种极简聚合算子的参考流处理引擎的性能；(3) 评估放宽构成这一核心算子集的聚合算子的某些能力假设所带来的性能提升。

组织结构：第 2 节介绍流处理和 DataFlow 模型的背景概念；第 3 节定义研究问题，并介绍我们采用的总体方法；第 4 节证明通用流处理算子等价于极简聚合算子的组合；第 5 节讨论对本研究的假设进行放宽以及可能的扩展；第 6 节分析仅采用聚合算子的流处理引擎的性能；第 7 节讨论与本研究相关的研究工作；第 8 节给出结论。

2 预备知识

2.1 流处理基础

根据 DataFlow，[1], a 流 S是一个无界的元组序列。每个元组都定义为属性值对列表 $\tau{:}v_0$，$a_1{:}v_1$，……，$a_n{:}v_n$。流是同质的，这意味着同一流中的每个元组 $t\,S$都具有相同的属性列表。我们将此列表称为 t或 S的类型，并用 $T(t)$表示与 $t \in S$关联的类型，或用 $T(S)$表示 S中任意元组的类型。我们假设元组的类型中始终包含一个特殊的时间戳属性（τ）。我们用 $t[i]$表示 t在索引 i处的属性，索引从 0 开始（即 $t[0] = t.\tau$）；而 $t[i{:}j]$表示从 i到 j（含首尾）的属性列表。给定一个属性值对列表 L，L表示携带该列表的元组。给定两个属性值对列表 L_1和 L_2，$L_1 ⌢ L_2$表示它们的串接。

流处理查询（或简称查询）由入口、算子和出口组成。入口会将入口元组（例如传感器或其他应用报告的事件）前向传递给算子；算子是处理元组的基本单元。在有向图中相连的算子会处理并前向传递或生成元组；最终，出口元组会送入出口，再由出口将结果交付给最终用户或其他应用。正如我们将在 § 2.2 中进一步说明的，同一算子的多个副本可以部署在同一图中，每个副本分析给定流的一部分，例如，在键分组并行性下，由共享同一键的元组组成的部分。

由于入口元组 t对应一个事件，$t.\tau$就是入口设定的事件时间，用于指明事件发生的时刻。算子会根据其语义，为每个输出元组设置 $t_o.\tau$ t_o，具体说明见后文。其他属性由用户定义的函数设置。事件时间以相对于给定纪元的时间单位表示，并按流处理引擎特定的离散 δ增量推进（例如毫秒 [2]）。

如 § 1 所述，我们将证明，一种名为 Aggregate 的极简算子（通常由流处理引擎 [2–4], 提供）以及 DataFlow 模型的若干核心概念，足以实现其他常见流处理算子的语义：过滤、映射、扁平映射和连接。下面将对这些算子进行介绍，以确保内容完整自洽。

这些算子的第一个区别在于其分析是无状态还是有状态。扁平映射、过滤和映射均为无状态算子，因为它们不会维护任何随所处理元组而变化的状态：

扁平映射 $S_O = FM(S_I, f_{FM})$ 是一种通用算子，它接收单个流 S_I和一个函数 f_{FM}作为输入，并生成单个流 S_O。具体而言，它会对每个输入元组$t_i \in S_I$调用函数 f_{FM}：该函数可以生成零个、一个或多个输出元组，并将其追加到输出流S_O中。请注意，f_{FM}可能会改变输入元组的类型。另外请注意，f_{FM}不会设置输出元组 t_o中的 τ，因为后者由 FM设置，因此，对于由 $t_i \in S_I$生成的 t_o，$t_i.\tau = t_o.\tau$。

过滤 $S_O = F(S_I, f_C)$ 会在 $f_C(t_i)$ 成立时，将每个元组 $t_i \in S_I$转发到 S_O。请注意，$T(S_I) = T(S_O)$，并且对于在处理 t_i时转发的元组 t_o，$t_i = t_o$。

映射 $S_O = M(S_I, f_M)$ 对每个元组 $t_i \in S_I$ 根据 $f_M(t_i)$ 将该元组 t_o 前向传递至 S_O。与 FM 一样，f_M 不会为输出元组 t_o 设置 τ，因为后者由 M 设置，因此对于由 $t_i \in S_I$ 产生的 t_o，$t_i.\tau = t_o.\tau$。

有状态算子根据依赖于一个或多个元组的状态生成结果。这种状态可以通过多种方式定义。在本文中，我们主要研究定义在由元组构成的有界分组上的有状态算子，这些分组称为基于时间的窗口（或简称窗口），通常由流处理引擎 [2–4] 提供：我们考虑窗口上的聚合和连接。在 § 5 中，我们将详细说明如何将分析扩展到更一般的状态定义。

为简洁起见，我们使用记号 Γ(窗口步长，时间窗 S_I，f_K，L) 表示由以下参数指定的窗口：

窗口步长（WA）、大小（WS） 定义了时期 $[\ell WA\, \ell WA + WS)$，其中 $\ell \in \mathbb{N}$，由 Γ 覆盖。我们将这样的一个时期称为窗口实例 γ。如果$WA < WS$，连续的窗口实例会重叠，Γ称为滑动窗口，一个元组可以落入多个窗口实例中。如果$WA = WS$，Γ称为滚动窗口，每个元组恰好落入一个窗口实例中。我们用 $t \in \gamma$ 表示 t 落入 γ。

输入流 S_I是馈送给 Γ 的输入流。

按键属性 f_K指定从 $T(S_I)$的属性中选取哪个子集（可能为空），以便为共享同一键的元组维护不同的窗口实例。请注意，f_K会影响维护 Γ 的算子的并行化方式（见 § 2.2）。

允许迟到时间 L用于判定落入 γ、但在维护 Γ 的算子为 γ 生成结果后才被该算子接收的元组 t，是否仍应加入 γ，这可能会产生新的输出元组或更新现有输出元组[1]。

我们将落入窗口实例 γ的元组集合称为 $\gamma.\zeta$，并将 $\gamma.\zeta$中的单个元组称为：若 $\gamma.\zeta$以列表形式维护，则 $\gamma.\zeta[0]$为索引 0 处的元组。我们将 γ左边界（含）的事件时间称为 $\gamma.l$。γ的右边界（不含）计算为 $\gamma.l$+时间窗。与相关研究中的常见做法一样 [2, 3, 5], ，当创建与窗口实例 γ相关联的输出元组 t_o时，将

$t_o.\tau$ $\gamma.l +$ $-\delta$ γ 设为时间窗。由于右边界不包含在内：

观察 1. 对于由某个窗口实例生成的任意输出元组 t_o，若输入元组 t_i落入该窗口实例，则 $t_o.\tau \geq t_i.\tau$。

我们考虑以下有状态算子：

聚合 $S_O = A(\Gamma(WA,$ 时间窗 $S_I,$ $f_K,$ $L)$ $f_O)$定义函数 $f_O(\gamma)$，用于根据 γ计算输出 t_o的值（τ除外）；如果 f_O不返回 $\emptyset$，则前向t_o。请注意，如前所述，$t_o.\tau$由 A根据运行 f_O所在的窗口实例 γ进行设置。

连接 $S_O = J(\Gamma(WA,$ 时间窗 $S_{I_1},$ $f_K^1,$ $L)$ $\Gamma(WA,$ 时间窗 $S_{I_2},$ $f_K^2,$ $L)$, $f_P)$定义 f_P，用于根据WA和 时间窗匹配分别落入对齐窗口γ_1和γ_2中的元组对 $t_1 \in S_{I_1}$和 $t_2 \in S_{I_2}$，从而使 $f_K^1(t_1) = f_K^2(t_2)$。如果 $f_P(t_1, t_2)$成立，则 J前向 $\gamma_1.l$+时间窗$-\delta ⌢ t_1 ⌢ t_2$。

在下文中，我们使用以下定义：

定义 1. 我们称元组 t'是 t 的后继，或者称 t'属于 t 的后继有限集，并记作 $t' \in succ(t)$，如果 t'是输出

[1]我们将在介绍第 2.3 节中的正确性条件后，于第 2.4 节讨论 L。

由算子 O 在处理 t 或其后继时产生。我们也使用记法 $succ(T)$，其中 T 是由 O 处理的元组集合。

2.2 无共享并行

如 § 2.1 所述，流处理引擎通过部署同一算子的多个副本来并行执行算子，从而最大化处理吞吐量和/或最小化处理延迟。

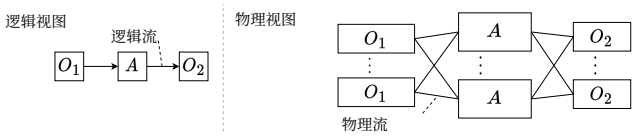

图 1：算子及其上游（O_1）和下游（O_2）对等节点的逻辑/物理视图。

为此，流处理引擎允许用户将算子定义为逻辑算子，之后再将其转换为物理实例（图 1）。由于无状态算子逐个处理元组，因此可以将输入数据打乱后发送给同一逻辑算子的多个实例，也可以按轮询方式分发。对于逻辑有状态算子，流处理引擎则运行一个或多个物理实例，利用按键并行，将数据拆分后发送给这些实例，以确保具有相同 f_K 值的元组（对于 A）或相同 f_K^1，f_K^2 值的元组（对于 J）由同一实例正确处理（关于正确性的更多内容，请参见第 2.3 节）。连接一对逻辑算子（或入口/出口）的每条逻辑流都会转换为一条或多条物理流。

2.3 正确性条件

部署并运行 F、M、FM、A 和 J 算子时，用户期望流处理引擎正确地执行这些算子的语义。由于 F、M 和 FM 逐个处理元组，且没有不断变化的状态，因此只要每个元组恰好处理一次，就能轻松确保语义正确。不过，A 和 J 则需要更加谨慎，才能正确地执行其语义。暂且不考虑迟到数据的存在（见 § 2.4），其正确执行可定义如下。

定义 2. 根据 A/J 的规范，如果来自 A/J 输入流的 A/J 的任意子集输入元组能够共同产生一个输出元组，那么这些元组确实会被一并处理，并产生该输出元组（若有），则称 A/J 的执行是正确的。

对于 A，定义 2 意味着，所有落入窗口实例 γ 的输入元组都应由 f_O 一并处理。对于 J，这意味着如果 $\gamma_1.l=\gamma_2.l$ 和 $f_K^1(t_1)=f_K^2(t_2)$，那么 $t_1\in\gamma_1$ 和 $t_2\in\gamma_2$ 中的任意一对元组都会由 f_P 处理。如 [6] 中所述，正确执行 A/J 的基础是持续维护其水位线：

定义 3. 在墙上时钟时间[2]ω，A/J的水位线 W_A^ω/W_J^ω是元组 t_i自时间 ω起馈送至 A/J时可能具有的最早事件时间（即，自 ω 起处理的 $t^i.\tau\geq W_A^\omega/W_J^\omega,\forall t_i$）。

在文献中，[2, 6], 水位线通常采用这样一种假设来维护：入口会定期以特殊元组的形式输出水位线。它们用于通知从入口的视角来看事件时间如何推进：

[2]下文中，只有在上下文无法明确区分时，我们才会区分墙上时钟时间（或简称时间）与事件时间。

算子将它们用作屏障，以便（1）即使没有显式元组也能取得进展；（2）对从时间戳无序的流中接收的元组重新排序。收到水位线后，A/J会存储该水位线的时间，将 W_A^ω/W_J^ω更新为由每个输入流最新水位线构成的集合中的最小值，并传播 W_A^ω/W_J^ω。

收到一个递增的水位线 W_A^ω后，A可以输出右边界不大于 W_A^ω的所有窗口实例的结果（即，在任何 $\gamma|\gamma.l+\text{时间窗}\leq W_A^\omega$ 上调用 f_O），因为不会再有元组落入这些窗口。根据观察 1，这意味着 A 也可以在输出此类结果后，将 W_A^ω转发给其下游对等算子。同样，J算子可以利用水位线 W_J^ω，安全地丢弃无法再产生任何结果的窗口实例。具体而言，J算子可以安全地丢弃分别定义在 S_{I_1}和 S_{I_2}上的任意一对窗口实例 γ_1,γ_2，前提是满足 $\gamma_1.l=\gamma_2.l$且 $\gamma_1.l+\text{时间窗}\leq W_J^\omega$。根据观察 1，$W_J^\omega$的值每次更新时都可以转发。

2.4 处理迟到数据

如 § 2.1 所述，Γ定义了 L，以处理迟到数据。若在时间 ω，O 处理t时满足 $t.\tau<W_O^\omega$，则元组 t对于算子 O而言是迟到数据。根据 Dataflow 模型，如果$\gamma.l+\text{时间窗}\leq W_O^\omega+L$在$\omega$，则$t$会被处理并添加到$\gamma$，且可能产生一个输出元组（可能表示对先前输出元组的更新）。在这种情况下，窗口实例 γ的清除会延迟 L，以考虑可能出现的迟到数据。请注意，通过设置 $L=0$，O将不会处理任何迟到数据。不过，如果 $L>0$和水位线按照 § 2.3 中所述由 O 转发，那么 O产生的结果可能会成为 O下游对等算子的迟到数据。为简洁起见，若 $L=0$，我们省略 L 的 Γ。下文还将使用以下定义：

定义 4。我们称元组 t 是在 O的水位线增长至 W_O时触发的输出元组，并记作 $t\in trig(W_O)$，如果 t由 O在 O的水位线增长至 W_O时生成并转发。

3 问题定义与方法

本研究有两个主要目标。首先，证明 F、M、FM和 J算子的语义可以通过组合 A 算子来实现（见 § 2.1）。其次，根据吞吐量和延迟等常见指标 [7–10], ，比较 F、M、FM和 J的专用实现（即流处理引擎提供的实现），以及基于 A 算子组合的基于聚合实现；具体讨论见 § 6。

虽然本文主要着眼于说明 § 2.1 中描述的基本 A算子足以实现 F、M、FM和 J算子，但我们也在 § 5.1 中考虑了 A的一种语义更丰富的变体，并说明它如何简化此类 F、M、FM和 J的定义。此外，虽然本文主要关注使用基于时间的窗口的 F、M、FM和 J 算子，但我们在 § 5.2 中表明，A 算子的组合也能够为其他流处理算子维护任意状态，这是我们计划在未来工作中探索的方向。

在我们的工作中，我们考虑满足以下条件的流处理引擎：

P1 类型相同的物理流可以输入同一个 A 算子。如果这些物理流属于逻辑流S_{I_1}、S_{I_2}……，我们用{S_{I_1}、S_{I_2}……}表示输入到 A 和 $T(S_{I_1})$的逻辑流，表示此类流共享的类型。

P2 物理/逻辑流可以输入一个或多个 A 算子，并按相同顺序传递相同的元组/水位线。
P3 一个 A 算子可以定义一个循环，迭代处理自身的输出流。A 转发的水位线不会通过这样的循环再次输入到 A 中（因为它是由 A 自身产生的）。

关于 **P1**，请注意，像 Apache Flink 这样的流处理引擎 [2] 会自动合并来自同一算子的并行实例的物理流（见 § 2.2），但对于来自不同逻辑流的物理流，则需要调用 union 方法。由于此类调用可以在查询编译时自动处理，因此我们不区分这两种情况。

关于 **P3**，请注意，通过循环从 A 输入到 A 的输出 t_o 始终是迟到数据，因为只有当 $W_A^\omega > t_o.\tau$ 时，t_o 才会输出（见 § 2.3）；并且处理此类 t_o 后产生的任何输出，对于 A 的下游对等算子而言也都属于迟到数据。有关更深入的讨论，请参阅附录 A；关于流式查询中循环复杂性的更多细节，请参阅 [11]。

我们将范围限定为处理通过循环馈送给 A 的迟到数据，因此假设基于聚合的 J 无需处理迟到数据（即，$L=0$ 对于 J），并假设满足以下条件：

C1 每个流 S 都会定期发送水印，且连续水印之间的最大事件时间间隔为 D。如果第一个元组 $t^0 \in S$ 早于第一个水印 W^0，则 $W^0 - t^0.\tau \leq D$，**C2** 运行 A 的流处理引擎不会仅因任何 t 被视为迟到数据，就阻止其通过循环馈送至 A 后进行处理，并且**C3** 运行 A 的流处理引擎会将 A 的所有输出和水印前向传递给 A 的下游对等算子，从而不会有输出成为迟到数据。

关于**C1**，请注意，如果向某个 A 算子馈送满足**C1** 的入口流，则 A 的输出也存在一个 D 间隔。进一步地，如果某个基于聚合的 F、M、FM 或 J 在接收 S 馈送后生成一个流，并且可以根据**C1** 为该流定义 D，则**C1** 也适用于此类 F、M、FM 或 J 以及下游 F、M、FM 或 J 对等算子。此外，请注意，这里以简化形式给出**C2** 和**C3**，以便于说明。关于基于 A 的水印的更详细表述，请参阅§ 4.4。在§ 4.5中，我们提供了在所使用的流处理引擎原生不支持**C2** 和**C3** 时用于强制实施这两项条件的算法；而在§ 5.1中，我们讨论了无需**P3**、**C1**、**C2**和**C3**的模型放宽方案。

为便于表述，且不失一般性，我们将讨论聚焦于逻辑算子和流（见第 2.2 节）。请注意，同一逻辑算子的并行实例接收相同的水印，因此可以按照下文所述执行分析、生成结果并转发水印。最后，为便于说明，所有引理的证明均见附录 B。

4 使用 A 算子确保通用算子的语义

本节将说明，通过组合 A 算子，可以实现 F、M、FM 和 J 算子的语义。首先要指出，FM 算子能够在处理输入元组 t 时生成任意数量的元组，其语义涵盖了 F 算子的语义（后者在处理 t 时生成 0 或 1 个元组），也涵盖了 M 算子的语义（后者在处理 t 时恰好生成 1 个元组）。因此，证明可以通过组合 A 算子来实现 FM 的语义，就表明 A 的语义也足以实现 F 和 M 的语义。至于 J 和 FM，考虑到每个窗口实例只生成 0 或 1 个元组的最小 A，一个固有难题是，J 和 FM 可能会生成多个分别来自单个元组或窗口实例 γ 的元组。正如我们将要说明的，解决这一问题的关键思路是将这项功能分解为两个步骤：先将多个元组的内容封装在一个元组中，再将这些内容展开为各个独立的元组。为此，我们引入两个辅助算子：嵌入（E）和展开（X），分别用于将多个元组合并为一个，以及将它们逐个展开并输出。E 和 X 利用了 Dataflow 模型的基本构件，即数据分区和循环。

基于这些思路，本节首先概述 E 和 X，并在第 4.1 节讨论它们如何支持 FM 和 J 语义；随后说明，在第 4.2 节中，如何通过一个 A 来确保封装 FM 的 E 的语义；在第 4.3 节中，如何通过组合 A 个{s}来确保封装 J 的 E 的语义；以及在第 4.4 节中，如何通过组合 A 个{s}来确保 X 算子的语义。此外，本文还提出了算法，以满足不原生支持这些要求的流处理引擎的**C2** 和 **C3**。

4.1 嵌入 (E) 和展开 (X) 算子

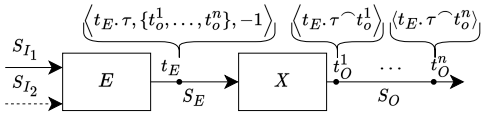

图 2：嵌入 (E) 和展开 (X) 的组合，可用于强制实现 FM 和 J 语义。

图 2 概述了我们如何组合 E 和 X 算子，以强制实现 FM/J 算子的语义。对于 FM，E 会输出一个元组 t_E，其中嵌入了处理 t_i 所得的所有结果；处理输入元组 t_i 时，还会输出值 -1，这是一个用于终止的特殊值。对于 J，E 会在 t_E 中嵌入来自两个对齐窗口实例的所有匹配对（见 § 2.1）。后续的 X 算子会展开 t_E，并输出包含在 t_E 中的所有元组 $\{t_o^1,\ldots,t_o^n\}$。在内部，它使用一个循环，反复从 t_E 中提取元组，直到遇到终止值 -1。更形式化地说：

- E 接收一个输入流 S_{I_1} 和一个可选的输入流 S_{I_2}，后者的类型可能不同[3]（即，$T(S_{I_1})$ 和 $T(S_{I_2})$ 可以不同）。
- E 生成一个输出流 S_E。一个元组 $t_E \in S_E$ 包含其时间戳 $t_E.\tau$、一个元组集合 $\{t_o^1,\ldots,t_o^n\}$，以及值 -1。正如我们在 § 4.4 中所述，值 -1 用于标识作为 E 输出的 t_E。
- X 生成一个输出流 S_O。收到元组 t_E 后，它会输出 $\langle t_E.\tau \frown t_o^1\rangle$，...，$\langle t_E.\tau \frown t_o^n\rangle$。

对于一个 E 和 X 对（见图 2），要实现 FM 的语义，E 应封装 f_{FM}，并将 $f_{FM}(t)$ 产生的元组输出到 t_E 中，以便 $t_E.\tau = t.\tau$ 之后能将它们解封装。更具体地说：

声明 1. E 和 X 实现 $S_O = FM(S_{I_1}, f_{FM})$ 的语义，当且仅当对于输入到 E 的每个 $t \in S_{I_1}$，都存在一个 t_E，使得 $t_E.\tau = t.\tau$，并且 $f_{FM}(t)$ 被携带在 t_E 的第二个属性中（即 $t_E[1]$，见 *§ 2.1*）。

为了让 E 和 X（见图 2）实现 J 的语义，E 应封装 J 的参数，并将一对对齐窗口 γ_1 和 γ_2 中的所有匹配元组嵌入带有 $t_E.\tau = \gamma_1.l + WS - \delta$ 的 t_E 中，以便之后由 X 解包这些元组。具体而言：

[3] S_{I_2} 是可选的，因为 FM 只有一个逻辑输入流（§ 2.1）。

声明 2. 如果对于每一对元组 $t_1 \in S_{I_1}$ 和 $t_2 \in S_{I_2}$，满足 $f_K^1(t_1) = f_K^2(t_2)$ 且 $f_P(t_1, t_2)$；并且对于每一对窗口 γ_1 和 γ_2，满足 $t_1 \in \gamma_1$，$t_2 \in \gamma_2$ 且 $\gamma_1.l = \gamma_2.l$，则 E会生成一个输出元组 $t_E = \langle \gamma_1.l + WS - \delta \frown \{t_o^1, ..., t_o^n\} \frown -1 \rangle$，该元组在 $t_E[1]$中携带 $t_1 \frown t_2$，此时 E 和X 实现了$S_O{=}J$（$\Gamma(WA,$ 时间窗 $S_{I_1}, f_K^1)$ $\Gamma(WA,$ 时间窗 $S_{I_2}, f_K^2)$ f_P）的语义。

在下文中，我们用 $E_{FM}(S_{I_1}, f_{FM})$ 表示封装 FM 语义的 E，用 $E_J(WA,\ WS\, S_{I_1},\ S_{I_2},\ f_K^1,\ f_K^2,\ f_P)$ 表示封装 J 语义的 E，并用 $X(S_E)$ 表示 X。

4.2 使用 A 强制执行 E_{FM} 的语义

定理 1。 声明 *1* 中描述的 $S_E = E_{FM}(S_{I_1}, f_{FM})$ 语义可以通过代码清单 *1* 中的 A 算子来强制执行。

代码清单 1： 实现 E_{FM} 语义的 A 算子

$S_E = E_{FM}(S_I, f_{FM}) = A(\Gamma(\delta, \delta, S_I, T(S_I))\ f_O)$，其中：

```
1 函数 f_O(γ)
2     T ← {} // 创建空集 T
3     对 t′ ∈ γ.ζ 执行 // 对每个 t′ ∈ γ
4         T ← T ∪ f_FM(t′) // 将 f_FM(t′) 添加到 T
5     若 T ≠ {} 则 // 一个或多个元组来自 f_FM
6         返回 T⌢−1
7     否则// 没有来自 f_FM
8         返回 {}
```

在证明定理 1 之前，我们先介绍以下引理。

引理 1. 如果 A定义了一个带有 $WA = \delta$和 $WS = \delta$的滚动窗口 Γ，那么任意 t都恰好落入一个窗口实例 $\gamma | \gamma.l = t.\tau$。此外，如果 $t_i \in \gamma$，且在调用 $f_O(\gamma)$时产生了 t_o，那么 $t_o.\tau = t_i.\tau$。

根据引理 1，我们现在可以证明定理 1。

证明。 （定理 1）每个元组t都恰好落入一个窗口实例，因为 Γ 是一个滚动窗口。如果窗口实例 γ包含多个元组，那么 γ中的所有元组都相同，因为 A使用完整的属性列表 T（S_{I_1}）进行键分组。由于γ中的所有元组都相同，每个元组 $t_\prime \in \gamma$输入 f_{FM}后都会产生相同的元组集合，而 T包含所有这些集合中的元组；如果 f_{FM}没有返回任何元组，则它为空。如果产生了一个输出 t_o，那么 γ的 $t_o.\tau$ 与 $\gamma.\zeta$ 中包含的相同（引理 1）。因此， A 实现了 E_{FM} 的语义。□

4.3 使用 A强制执行 E_J的语义

定理 2. $S_E = E_J(WA, WS\, S_{I_1}, S_{I_2}, f_K^1, f_K^2, f_P)$ 的语义可以通过组合如图 *3* 和代码清单 *2* 所示的 As 来实现。

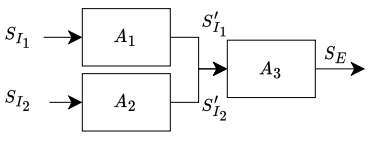

图 3：实现 J语义的 A 算子图。

代码清单 2： 实现 E_J语义的 A 算子。

$S'_{I_1} = A(\Gamma(\delta, \delta, S_{I_1}, T(S_{I_1}))\ f_O)$，其中： // A_1

```
1  函数 f_O(γ)
2      T ← {} // 创建空集 T
3      对于 t′ ∈ γ.ζ 执行 // 对每个 t′ ∈ γ
4          T ← T ∪ t′ // 将 t′ 添加到 T
5      返回 T⌢{}
```

$S'_{I_2} = A(\Gamma(\delta, \delta, S_{I_2}, T(S_{I_2}))\ f_O)$，其中： // A_2

```
6  函数 f_O(γ)
7      T ← {} // 创建空集 T
8      对于 t′ ∈ γ.ζ 执行 // 对每个 t′ ∈ γ
9          T ← T ∪ t′ // 将 t′ 添加到 T
10     返回 {}⌢T
```

$S_E = A(\Gamma(WA,$时间窗$, \{S'_{I_1}, S'_{I_2}\}\ f_{\prime K})\ f_O)$，其中： // A_3

```
11 函数 f′_K(t)
12     如果 t[2] = {} 那么 // t 来自 S_I1
13         返回 f_K^1(T[1][0]) // f_K^1 t 中第一个的
14     否则// t 来自 S_I2
15         返回 f_K^2(T[2][0]) // f_K^2 在 T[2]中的第一个 t
16 函数 f_O(γ)
17     win1 ← {} // 为来自 S_I1 的元组创建列表
18     win2 ← {} // 为来自 S_I2 的元组创建列表
19     T ← {} // 创建输出元组列表
20     for t ∈ γ.ζ do
21         if t[2] = {} then // t 来自 S_I1
22             for t′ ∈ t[1] do // 对 t′ 中的所有元素 t
23                 for t″ ∈ win2 do // 对 t″ 中的所有元素 win2
24                     如果 f_P(t′, t″) 那么// 连接 t′ 和 t″
25                         T ← T ∪ t′⌢t″
26                 win1 ← win1 ∪ t′// 存储 t′
27         否则// t 来自 S_I2对于 t″ ∈ t[2] 执行// 对于所
           有 t″ 在 t
28             对于 t′ ∈ win1 执行// 对于所有 t′在 win1
29                 如果 f_P(t′, t″) 那么// 连接 t′ 和 t″
30                     T ← T ∪ t′⌢t″
31             win2 ← win2 ∪ t″// 存储 t″
32     如果 T ≠ {} 那么 // 有结果可返回
33         返回 T⌢−1
34     否则// 没有结果可返回
35         返回 {}
36
```

为了便于理解和证明定理 2，我们可以注意到，在代码清单 2 中， A_1和 A_2输出流具有相同的类型，由时间戳 τ和两个集合组成。对于 A_1，第一个集合包含来自S_{I_1}的元组，第二个集合为空。与之对称，对于A_2，第一个集合为空，第二个集合包含来自 S_{I_2} 的元组（代码清单 2，第 1–10 行）。根据 **P1**（见 § 3），二者都可以透明地馈送给 A_3。 A_3负责匹配来自 S_{I_1}和 S_{I_2}的元组，它定义了一个 $f_{\prime K}$函数，根据t来自 S_{I_1}还是 S_{I_2}，在 t上运行 f_K^1 或 f_K^2（代码清单 2，第 11–15 行）；也就是说，它根据输入元组所属的原始流计算正确的键。由于A_1将所有输入元组的属性用作键分组，因此如果向输出元组 t_o所携带的集合中添加了多个输入元组，那么 t_o所携带的所有元组都相同。对 A_2也有类似的观察。因此，尽管 $f_{\prime K}$会对输入元组 t在 $t[1]$或 $t[2]$中携带的第一个元组运行 f_K^1或 f_K^2，A_3的 $f_{\prime K}$仍会根据元组的键分组值，将元组一致地分配到其所属的窗口实例。

A_3的 f_O会遍历所有键相同的元组（List.2,L20-32）。对于每个这样的元组 $t_\prime$，它会执行笛卡尔积

，将 $t_\prime$ 携带的 S_{I_1} 或 S_{I_2} 中的元组，与此前从另一条流中遍历到的元组进行匹配（List.2,L22-25,L28-31），并存储 $t_\prime$，以供 $t_\prime$ 与 $\gamma.\zeta$ 中在 $t_\prime$ 之后遍历到的其他元组进行比较（List.2,L26,L32）。所有匹配的元组对都会存储在 T 中。最后，如果 T 不为空，A_3 会前向传递 T 和 -1（List.2,L33-36）。

证明。（定理 2）反证：若 E_J 实现不正确，则存在一对元组 t_1 和 t_2，分别落入窗口 γ_1 和 γ_2，使得 $f_P(t_1, t_2)$ 成立，且 $\gamma_1.l = \gamma_2.l$，但 $t_1 \frown t_2$ 并未由 A_3 的 f_O 添加到 T 中。根据 List.2,L16-36 中的函数 f_O，只有在 t_1 或 t_2 未分别落入 γ_1 或 γ_2，或者其中一个或两个元组起初就没有输入到 A_2 中时，这两种情况才可能成立，这与初始假设矛盾。□

4.4 使用 A 强制执行 X 的语义

在第 3 节中，我们结合 A 循环遍历其输出元组（**P3**）的可能性，介绍了 **C1**、**C2** 和 **C3**。由于 X 依赖于 **P3**，本节将首先以更精确的方式重新表述 **C2** 和 **C3**（为便于阐述，前文采用了简化的表述）。如果所用的流处理引擎不原生支持 **C2** 和 **C3**，读者可参阅 § 4.5，了解用于强制执行它们的算法。

C2 W_A^ω 仅更新一次 W_A^ω 并不能阻止由 A 生成并经由循环流 S 馈送给 A 的任何 t_0 因被视为迟到数据而得到处理。也就是说，如果 $\forall t_o \in S$，那么在 ω 时，t_o 仍将由 A 处理；并且若其在 A 时属于满足 $t_o \in \gamma$ 的窗口实例 γ，则有 $\gamma.l$+时间窗 $\leq W_A^\omega + L$，且

C3 A 在 $succ(trig(W_A^\omega))$ 之后将 W_A^ω 馈送给下游对等算子。

有了这些更精确的表述，我们可以给出以下结论：

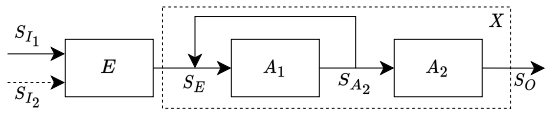

图 4：用于 X 算子的 A 算子的组合。

定理 3. 一个 $S_O = X(S_I)$ 的语义可以通过组合算子 A_1 和 A_2 来实现，如图 *4* 和代码清单 *3* 所示，其中 ***C1*** 对 S_E 成立，***C2*** 和 ***C3*** 对 A_1 及 $L \geq D$ 成立。

为便于理解和证明定理 3，我们可以注意到，在代码清单 3 中，A_1 具有一个大小为 δ 且允许迟到时间为 L 的滚动窗口（参见 § 2.3），并且根据 **P2**，S_{A_2} 同时馈送给 A_1 和 A_2。如果一个输入 t 来自 E，因而将 -1 作为最后一个值（代码清单 3，L3），那么 A_1 会转发一个元组 t_o，其中携带了 t 所携带的所有输出元组以及值 0，表示 A_2 将要生成的下一个元组位于索引 0。反之，如果 t 来自 A_1 自身，A_1 会转发一个 t_o，其中最后一个属性上的计数器加 1（代码清单 3，L9）。请注意，如果两个相同的元组从 E 馈送给 A_1，它们最终会落入同一个窗口实例 γ，因为 A_1 的 f_K 会选取所有元组的属性。因此，这些重复元组所携带的所有 t_o 元组都会被添加到由 γ 生成的输出中。考虑到 A_1 如何处理重复元组，我们还可以给出以下引理。

引理 2. A_1 不会产生重复项。

代码清单 3： A 实现 X 语义的算子。

$S_{A_2} = A(\Gamma(\delta,\ \delta,\ \{S_E,\ S_{A_2}\}\ T(S_E)\ L)\ f_O)$，其中：// A_1

```
1  函数 f_O(γ)
2  |  t ← γ.ζ[0]
3  如果 t[2] = −1 那么 // 如果 t 来自 E
4  |  |  T ← {} // 创建空集
5  |  |  对于 t′ ∈ γ.ζ 执行
6  |  |  |  T ← T ∪ t′[1] // 用输出填充 T
7  |  |  返回 T⌢0 // 前向 T索引 0
8  |  否则如果 t[2] < |t[1]| 那么 // 如果 t 应继续循环
9  |  |  返回 t[1]⌢(t[2]+1) // 增加索引
10 |  否则 // 如果 t 循环已完成
11 |  |  返回 ∅
```

$S_O = A(\Gamma(\delta,\ \delta,\ S_{A_2},\ T(S_{A_2}))\ f_O)$，其中：// A_2

```
12 函数 f_O(γ)
13 |  t ← γ.ζ[0]
14 |  返回 t[1][t[2]] // 给定索引处的前向元组
```

对于输入元组 t 第二个属性中包含的元素，A_2 会前向传递索引由 t 的第三个属性指定的元素（列表 3，L14）。由于输入到 A_1 的每个元组都携带一个由 t_o 元组组成的有限集，因此从 A_1 输出的元组 t 也携带一个由 t_o 元组组成的有限集。因此，$t[2]$ 处不断增长的索引最终会大于 $|t[1]|$，从而防止 t 在 A_1 中无限次循环，并避免其被无限次输入到 A_2。

证明。（定理 3）反证法；将 $t^l = \left\langle \tau^l \frown T \frown -1 \right\rangle$ 与 $T = \{..., t_o^j, ...\}$ 输入到 A_1，但 A_2 没有生成 t_o^j。

如果 A_2 没有生成 t_o^j，那么 A_2 没有接收到 $t^* = \left\langle \tau^l \frown T \frown j \right\rangle$**(1)**，$A_2$ 接收到了 t^*，但它作为迟到数据未被处理 **(2)**；或者 A_2 接收到了 t^*，但 t^* 不在 $\gamma.\zeta[0]$ 中，因为 $\gamma.\zeta[0]$ 是 A_2 的 f_O 所考虑的唯一元组（列表 3，L12-14）**(3)**。

(1) 意味着序列 $\tau \frown T \frown -1, ..., \tau \frown T \frown j-1$ 并未完整地交付给 A_1。假设 $\tau \frown T \frown -1$ 未被交付，会与初始假设相矛盾。如果处理了 $\tau \frown T \frown -1$，由于 **C2** 和 **C1**，且序列中的所有元组具有相同的时间戳，**(1)** 会导致矛盾。

(2) 意味着 t^l 被馈送给了 A_1。我们将 t^l 在 A_1 时落入的窗口实例称为 γ_l。如果 t^* 由 A_1 生成，那么 A_1 接收到了水位线 W^m，使得 $W^m \geq t^*.\tau + \delta (\gamma_l.\tau = t^*.\tau$，参见引理 1)。令 W^m 为大于或等于 $t^*.\tau + \delta$ 的最早水位线，即 $W^{m-1} < t^*.\tau + \delta$。**C3** 确保 W^{m-1} 是馈送给 A_2 的最新水位线，因为 $t^* \in succ(trig(W^m))$。然而，要成为 A_2 的迟到数据，必须满足 $t^*.\tau < W^{m-1}$，这会导致矛盾。

(3) 意味着存在相同的元组被馈送至 A_2，因为 A_2 将所有属性值用作键分组。因此，**(3)** 与引理 2 矛盾。□

4.5 处理 X 的 A_1 的水位线

对循环图支持有限的流处理引擎可能无法原生支持 **C2/C3**，因此，X 的 A_1 不会基于 t_o 是迟到数据这一情况，在输出元组 t_o 上循环，也不会将迟到数据转发至 A_2。因此，我们引入两种算法来强制实现 **C2/C3**，并定义流处理引擎处理流 S_E 和 S_{A_2} 的元组 t 或水位线 W 时执行的四种（非并发）操作（见图 4）：

- $processT(t)$，在 t 被馈送至 S_E/S_{A_2} 时运行
- $processW(W)$，当 W 被输入到 S_E/S_{A_2} 时运行，

- $forwardT(t)$运行，将 t 提供给由 S_E/S_{A_2}提供输入的算子；
- $forwardW(W)$运行，将W提供给由 S_E/S_{A_2}提供输入的算子。

这些算法使用以下数据结构：

- *Queue*（FIFO），具有相关方法 *enq*和 *deq*，并使用记法 $Q[i]$来指代 Q的索引 i处的元素，以及
- *TreeMap*，一种支持按其键及关联值进行有序遍历的映射。

我们使用记法 $m[k]$来指代 *TreeMap* m中键 k对应的值。方法 *firstKey*返回第一个键的值（不移除 k及其值），而方法 *remove*(k)会移除键 k以及与 k关联的值。

清单 4 展示了 *processT*和 *processW*的实现，以便为 S_E强制实施 **C2**。S_E维护四个变量：B（列表 4，L1），即 S_E可前向的最高水位线的上界；*succ*Γ（列表 4，L2），即一个 *TreeMap*，用于维护在 A_1处每个窗口实例 γ的左边界所关联的后继元组数量；*pendingW*（列表 4，L3），即 S_E可前向的水位线 *Queue*；以及 L（列表 4，L4），即 A_1的允许迟到时间。

收到元组 t(List.4,L5)后，S_E前向 t(List.4,L6)，并继续更新 *succ*Γ：将左边界为 $t.\tau$（见引理 1）的窗口对应条目增加 t所携带的后继数量（List.4,L7），前提是 t来自 A_1之前的 E算子（即，如果 $t[2]=-1$）；如果 t是来自 E的先前元组的后继，则减少 $t.\tau$处的条目（List.4,L10）。如果 *succ*Γ中的某个条目减至 0，则从 *succ*Γ中移除该条目（List.4,L11-12）。随后，S_E继续更新 B：如果 *succ*Γ中的所有条目均已清除，则将其设为 ∞；否则，将其设为 S_E尚未收到的最早后继的左窗口边界时间戳值加上 L（List.4,L13-16）。最后，S_E尝试前向 *pendingW*中小于或等于 B的最新水位线，并丢弃其他更早的水位线（List.4,L17-21）。

收到水位线W（清单 4，第 L22 行）后，S_E会立即转发W（如果 $W \le B$），或将其存储在 *pendingW*中，以便在随后调用 *processT*（清单 4，第 L23–26 行）时尝试转发W。

根据清单 4，我们注意到。

引理 3。如果***C1***对 S_E成立，则清单 *4* 会对 A_1强制实施***C2***。

在讨论如何对 S_E强制实施**C2**之后，我们现在基于清单 5 讨论如何对 S_{A_2}强制实施**C3**。

S_{A_2}收到元组 t后立即转发该元组，并继续更新变量 *succ*Γ（清单 5，第 L4–10 行）。由于 S_{A_2}只能看到由 S_E转发给 A_1的元组 $t_\prime$的后继，S_{A_2}会增加 *succ*Γ中元组 t对应的条目，使其变为 $|t[1]|-1$（因为 t本身就是源自 $t[1]$的后继之一）。更新 *succ*Γ后，如果 *succ*Γ为空，S_{A_2}就会将 $t.\tau$作为水位线转发；否则，如果某个值大于 S_{A_2}最近转发的水位线，S_{A_2}会将尚未被 S_{A_2}看到的某个后继的窗口 -1最早左边界的时间戳作为水位线转发（清单 5，第 L11–16 行）。

注意，由于$T=succ(t)$，$T[0]$会在水位线 $W|W \ge T[0].\tau$之前被 S_{A_2}接收（因为W是 A_1生成 $T[0]$所必需的，而W会在 $T[0]$之后被馈送给 S_{A_2}，见第 2.3 节），与此同时，T 中的所有其他元组都会在 W 之后被 S_{A_2} 接收。因此，所有导致 *succ*Γ中的某个条目增加的元组，都先于那些使 *succ*Γ中的某个条目减少的元组；并且，如果 *succ*Γ中最早的条目在处理 $T[0]$时减少为 0，那么 T中其他元组的处理只能使 *succ*Γ中较晚的条目减少。

清单 4： 为 Stream S_E 强制实施 **C2**

```
1  B=∞ // 的上限          即 S_E 可以前向
2  succΓ// TreeMap 的 γ's 右边界-#后继
3  pendingW // Queue 的W
4  L// A_1的允许迟到时间
5  函数 processT(t)
6      forwardT(t) // 前向 t
7      如果 t[2]=−1 则
8          succΓ[t.τ]←succΓ[t.τ]+|t[1]|
             // 跟踪 γ 的后继，其左边界为 t.τ
9      else
10         succΓ[t.τ]←succΓ[t.τ]−1
             // 减少 γ 的后继，其左边界为 t.τ
11         如果 succΓ[t.τ]=0 则 // 已获取所有 t.τ 的后继
12             succΓ.remove(t.τ)
13     如果 |succΓ|>0 则 // 更新 B
14         B←succΓ.fistKey()+L
15     else
16         B←∞
17     nextW←−1// 前向最新的 W|W      ≤B
18     当 |pendingW|>0∧pendingW[0]≤B 执行
19         nextW←pendingW.deq()
20     如果 nextW≠−1那么
21         forwardW(nextW)
22 函数 processW(W)
23     if W≤B 那么// 如果 W 在 B 内，发送W
24         forwardW(W)
25     否则// 否则，存储 W
26         pendingW.enq(W)
```

调用 *processW*时，如果 S_{A_2}为空，则W会立即转发；否则，如果某个尚未被 S_{A_2}看到的后继窗口 -1的最早左边界的时间戳，大于 S_{A_2}最近转发的水位线的时间戳，则 S_{A_2}会立即转发该时间戳（列表 5，第 L17–23 行）。

根据清单 5，我们注意到：

引理 4. 如果***C1***和***C2***分别对 S_E和 A_1成立，那么清单 *5* 会对 A_1强制实施***C3***。

5 放宽与扩展

在进入评估之前，我们将在此讨论，放宽 A 仅从窗口实例 γ 中产生零个或一个元组这一要求（见 § 2.1），如何有助于强制实施 *FM*和 J语义；我们还将展示证据，说明 A的语义如何在流处理引擎及其他框架内部和彼此之间，与除 F、M、*FM* 和 J之外的其他算子发生重叠（这是本文的主要关注点）。

5.1 语义丰富的 er *A*s

如第 4 节所述，我们引入了 E和 X算子，以弥补 A约束：每个窗口实例只能产生零个或一个元组 γ。不过，Flink 等流处理引擎允许用户从 γ产生任意数量的输出元组。既然已经证明 A足以实施 F、M、*FM*和 J，我们便可得出结论：这种语义更丰富的 A（我们称之为 A^+）也能实施 F、M、*FM* 和 J语义。出于篇幅考虑，我们不作正式讨论；只需指出，基于 A^+的 E无需使用 X，因为它能够立即前向传递原本会由 E嵌入 $t_E[1]$中的元组（见第 4.1 节）。请注意，如果不需要 X算子，则可以取消 **C1**、**P3**、**C2**和**C3**。关于 A和 A^+的性能实证比较，请参见第 6 节。

清单 5： 为 Stream S_A 强制执行 C3 $_2$

```
succΓ// γ 的 TreeMap 右边界-后继
lastW // 最后W 由 S_A前向转发 2
函数 processT(t)
    forwardT(t) // 前向 t
    如果 t[2]=−1 那么
        succΓ[t.τ] ←succΓ[t.τ]+|t[1]|−1 // 跟踪
            左边界为 t.τ 的 γ 后继，不包括其自身 t 。
    else
        succΓ[t.τ] ←succΓ[t.τ]−1
            // 减少左边界为 t.τ 的 γ 后继
        如果 succΓ[t.τ]=0 那么 // 已获取所有后继： t.τ
            succΓ.remove(t.τ)
    如果 |succΓ|=0 则// 前向 t.τ作为水位线
        forwardW(t.τ)
        lastW ←t.τ
    否则如果 succΓ.firstKey()−1>lastW
     则// 基于最早的待处理后继生成前向水位线。
        forwardW(succΓ.firstKey()−1)
        lastW ←succΓ.firstKey()−1
函数 processW (W)
    如果 |succΓ|=0 那么 // 所有待处理后继均已清除
        forwardW(W)
        lastW ←W
    否则如果 succΓ.firstKey()−1>lastW
     那么// 基于最早的待处理后继设置前向水位线。
        forwardW(succΓ.firstKey()−1)
        lastW ←succΓ.firstKey()−1
```

5.2 使用 A强制执行自定义语义

虽然通用算子（如 F、M、FM和J）是我们关注的重点，但也可以提出这样一个问题：A 是否也能实施其他算子的语义，包括用户定义或自定义的算子。尽管高效地组合 A 可能需要针对具体情况逐一讨论，但我们认为这个问题的答案是肯定的，因为正如我们接下来将展示的，组合 A可用于维护超出基于时间的窗口所能维护范围的状态。

接下来，我们定义一个具有无界状态（就事件时间而言）的通用 O算子，使得输入到O中的每个元组 t都只被处理一次，并根据之前处理的所有元组更新状态。需要注意的是，O的状态在其涵盖的事件时间范围内是无界的，但用户应对其进行定义，使其空间复杂度不会不断增长。正如我们将展示的那样，可以依靠一个带有滑动窗口的 A 以及一个循环，将某个窗口实例 γ_l的状态传递给 γ_{l+1}的状态，从而强制执行 O的语义。随后，用户可以根据一个 状态元组 定义O的状态，该元组会在连续窗口之间传递，同时考虑到 O的输入元组和 O的状态元组可能具有不同的类型。更具体地说，我们将$S_O = O(f_c, f_a, f_m, f_o, P, f_k, S_I)$定义为一个有状态算子，使得：

函数 f_c根据输入元组 t_i创建状态元组 t_s,**函数 f_a**将 t_i 添加到现有状态元组 t_s,**函数 f_m**合并两个状态元组 t_{s_1} 和 t_{s_2},**函数 f_o**根据 t_s生成输出元组,**输出周期 P**是调用 f_o 的周期，**键分组函数 f_k** 是一个键分组函数（与 A 相同），并且**输入流 S_I**是输入流，

并陈述以下引理：

引理 5. $S_O = O(f_c, f_a, f_m, f_o, P, f_k, S_I)$ 的语义可通过组合多个 A 来强制实现，如图 *5* 和代码清单*6*所示，其中 ***C1***对 S_{FM} 成立，***C2*** 和***C3***对 A_1 成立，而 A_1 的 $L > D$ 成立。

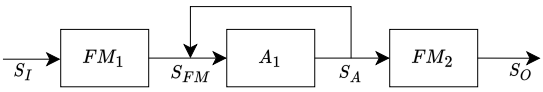

图 5：实现 O 的语义的 A 算子图。

代码清单6： 实现 O 的语义的 A 算子。

```
S_FM = FM(S_I, f_FM)，其中： // FM_1
1 函数 f_FM(t)
2     返回 {t}⌢{}
S_A = A(Γ(P, P+δ, S_FM, f_k, L) f_o)，其中： // A_1
3 函数 f_o(γ)
4     first_t_s ←True
5     对于 t ∈ γ.ζ 执行 // 对每个 t ∈ γ
6         如果 t.τ ≠ γ.l+P−δ 那么 // t 仅在 γ中结束
7             如果 t[1] ≠ 0 那么 // t 来自 FM_1
8                 如果 first_t_s 那么// 未创建状态
9                     t_s ←f_c(t[1])
10                    first_t_s ←False
11                否则// 已创建状态
12                    t_s ←f_a(t_s, t[1])
13            否则// t 来自 A_1
14                如果 first_t_s 那么// 未创建状态
15                    t_s ←t[2]
16                    first_t_s ←False
17                否则// 已创建状态
18                    t_s ←f_m(t_s, t[2])
19    返回 {}⌢{t_s}
S_O = FM（S_A, f_FM），其中： // FM_2
20 函数 f_FM(t)21 返回
f_o(t[2])
```

请注意，由于A_1的窗口实例会在到期时进行处理，因此可以按类型对每个实例中的元组进行排序，以控制调用 f_c、f_a 和 f_m 的顺序。

6 评估

我们的工作表明，通用流处理引擎算子可以表示为 A算子的组合。因此，可以构建一个仅采用 A 算子的流处理引擎，并将其作为参考，用于验证专用算子的语义，即特定流处理引擎提供的算子。本节通过比较通用算子的基于聚合实现与专用实现，考察这种参考流处理引擎的性能。此比较并非旨在确定最佳实现，因为预期专用实现的性能更好。相反，我们考察了多种算子、实现特性、工作负载、选择率和元组处理成本，并表明基于聚合的算子的性能往往可与专用算子相媲美。因此，依赖基于聚合算子的参考流处理引擎完全可以用于测试和验证，也可能在某些场景中投入实际使用。为构建这样的参考流处理引擎，我们采用

Flink。我们考察 Flink 的 A 和 $A+$（见 § 5.1）算子。我们使用后者来说明语义更丰富的 A 算子如何缩小专用实现与基于聚合实现之间的差距。我们还比较了专用实现和基于聚合实现，选择了能够代表部署在云边光谱两端的分析应用，从而考虑到底层硬件计算能力的差异。

下文将在第 6.1 节介绍我们的设置，并在第 6.2 节讨论结果。我们将专用实现称为 D，将基于聚合的实现称为 A 和 $A+$。

6.1 设置

数据集。我们使用了（1）[12]提供的 4300 万条维基百科原子编辑中的一部分，以及（2）在工业环境中使用测距传感器采集的 565 组二维扫描结果 [13, 14]。对于（1），元组的属性包括：τ（long），在元组前向传递时设置；*orig*（String），原始条目；*change*（String），新增文本；以及 *updated*（String），修改后的条目。对于（2），元组的属性包括：τ（long），在元组前向传递时设置；*id*（int），元组标识符；以及 *dist*（double 数组），传感器测得的距离值。

实验。我们比较 D、A、$A+$ 对 FM 和 J 算子的实现。表1列出了用于评估各种实现性能的实验。对于每项实验，我们都会说明涉及的算子、其选择率、计算成本以及用于执行实验的硬件设备。

硬件。我们使用 (1) 一台配备 Intel Xeon E5-2637 v4@ 3.50GHz（4 核、8 线程）和 64 GB RAM 的服务器处理维基百科编辑；以及 (2) Odroid-XU 4 [15]（或简称 *Odroid*）边缘设备 [16, 17],，其搭载 Samsung Exynos5422 Cortex-A15 2GHz 和 Cortex-A7 八核 CPU，并配备 2 GB RAM，用于处理二维扫描结果。

软件。我们使用 Apache Flink 1.15.2 [2] 和 Ubuntu 18.04。由于 Flink 中存在一个可能导致循环死锁的未解决问题，[18], 我们定义了自己的循环处理机制，以满足**C2** 和 **C3**。

方法。性能以最大可持续吞吐量和延迟来衡量。FM的吞吐量以元组/秒（t/s）为单位衡量，而 J的吞吐量以比较次数/秒（c/s）为单位衡量。对于 FM和 J，延迟均指在产生 t_o的算子处理完所有共同促成 t_o产生的输入元组（一个或多个）之后，生成一个输出元组 t_o 所需的时间。我们将最大可持续

吞吐量是指流处理引擎能够持续10分钟、且延迟超过15秒不超过3次时所能承受的注入速率（此时我们称该实验为成功）。我们针对不同的注入速率分别运行每次10分钟的实验。每次执行的第一个预热分钟和最后一个冷却分钟不计入结果。对于吞吐量和注入速率，我们报告每秒平均值。对于延迟，我们报告每秒值的第99百分位数。结果取三次重复实验的平均值。

我们研究不同的选择率和每元组处理开销。FM的选择率是每个输入元组的平均输出元组数。J的选择率是每次比较产生的平均元组数。对于 FM，我们考虑 FM 的选择率小于 1，以模拟 F；并考虑 FM 的选择率等于 1，以模拟 M。由于 J的比较次数与输入速率 [19], 呈二次关系，因此我们参照相关工作 [9, 19]，考虑小于 1 的选择率。表1概述了下文讨论的所有实验。

6.2 结果

现在，我们介绍评估结果，先考察FM算子，再考察 J算子。

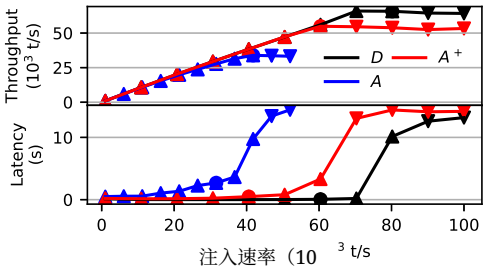

图6：AHF——吞吐量/延迟与注入速率的关系

FM 算子。我们首先展示一个特定实验及其一次重复的结果，并指出其他实验也观察到了类似的行为。图6展示了在AHF实验中（见表1），随着注入速率（t/s）增加，吞吐量（上图）和延迟（下图）如何变化。向上的标记表示成功的实验，向下的标记表示不成功的实验。

在所有实现中，吞吐量最初都会随注入速率线性增长，但达到最大可持续吞吐量后便趋于平稳，同时延迟也会相应增加；越接近算子的最大可持续吞吐量，延迟上升越快。D 的最大吞吐量高于 A/A^+，尽管三者仍处于同一数量级，其中 A 的性能下降约 50%，A^+ 则下降 16%。在可持续注入速率下，我们观察到 A 的延迟增长速度快于 A^+，而 A^+ 的延迟增长速度又快于 D。这是因为 D 无需水位线来触发结果生成（因为 FM 是无状态算子），而 A 和 A^+ 则需要，因此它们的延迟取决于入口转发水位线的周期性。此外，对于 A，延迟增加也是由于

为满足 **C2** 和 **C3** 而产生的水印前向延迟（参见 § 4.4）。与 A 相比，A^+ 的吞吐量更高、延迟更低，这也归因于 A^+ 不依赖于 X算子（参见 § 5.1），因此无需循环。

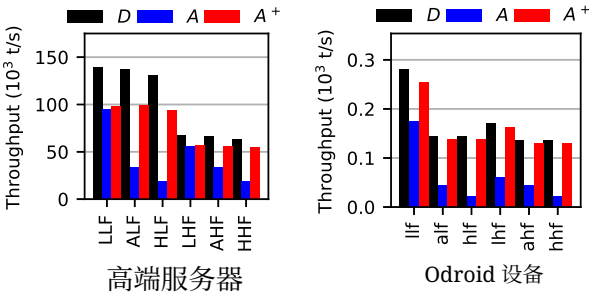

图 7：FM——所有实验的平均吞吐量。

表1：实验列表。

高端服务器（大写标识符）					Odroid 设备（小写标识符）				
ID	**选择率**	**Cost**	**Op.**	**备注**	**ID**	**选择率**	**Cost**	**Op.**	**备注**
LLF	低（$\sim 5e^{-3}$）	Low	*FM*	查找 *orig*中的最高频词。前向，如果长度 ＞10个字符	llf	低（0.2）	Low	*FM*	将坐标从极坐标转换为笛卡尔坐标。如果平均距离为 ＞3 米，则前向
ALF	平均值（1）	Low	*FM*	查找 *orig* 中的最高频词并前向it	alf	平均值（1）	Low	*FM*	将坐标从极坐标转换为笛卡尔坐标
HLF	高（3）	Low	*FM*	查找 *orig*中的前 3 个高频词。前向为单独的元组	hlf	高（3）	Low	*FM*	将坐标从极坐标转换为笛卡尔坐标，并拆分/前向处理为3个部分
LHF	低（$\sim 3e^{-4}$）	High	*FM*	在 *orig*、*change*中查找最高频词，并 *update*。如果它们的长度都为 ＞10个字符，则将它们放入单个元组中进行前向处理	lhf	低（~ 0.7）	High	*FM*	将坐标从极坐标转换为笛卡尔坐标从参考点开始。若平均距离为 ＞3米，则向前
AHF	平均值（1）	High	*FM*	在 *orig*、*change*中查找最高频词，并且 *update*。将它们放入单个元组中向前传递	ahf	平均值（1）	High	*FM*	将坐标从极坐标转换为笛卡尔坐标从参考点
HHF	高（~ 2.3）	High	*FM*	查找 *orig*、*change*和中的前 3 个高频词 *update*。以前向方式作为独立三元组	hhf	高（3）	High	*FM*	将坐标从极坐标转换为笛卡尔坐标以参考点为起点，并拆分/前向分为 3 部分
LLJ	低（$\sim 1e^{-4}$）	Low	*J*	匹配两个不同的（不区分大小写）*orig*，前提是它们具有相同长度且有 $\|orig\| > 210$ 个字符。键分组 *change*中的单词数量，*WA*=1秒和时间窗=3秒	llj	低（$\sim 8e^{-5}$）	Low	*J*	如果 *dist*中的差异总和为＜0.5米，则匹配两个不同的扫描结果。*WA*=0.5秒和时间窗=1秒
ALJ	平均值（$\sim 1e^{-3}$）	Low	*J*	与LLJ相同，但 $\|orig\| > 150$	alj	平均值（$\sim 8e^{-4}$）	Low	*J*	与 LLJ 相同，但差异总和为 ＜0.6m
HLJ	高（$\sim 3e^{-3}$）	Low	*J*	与 LLJ 相同，但 $\|orig\| > 100$	hlj	高（$\sim 5e^{-3}$）	Low	*J*	与 LLJ 相同，但差异总和为 ＜0.7m
LHJ	低（$\sim 1e^{-4}$）	High	*J*	与 LLJ 相同，但时间窗=10秒	lhj	低（$\sim 6e^{-5}$）	High	*J*	与 llj 相同，但 时间窗=2秒
AHJ	平均值（$\sim 1e^{-3}$）	High	*J*	与 LLJ 相同，但 $\|orig\| > 150$ 和 时间窗=10秒	ahj	平均值（$\sim 7e^{-4}$）	High	*J*	与 LLJ 相同，但差异总和为 ＜0.6m，且时间窗时间窗=2s
HHJ	高（$\sim 3e^{-3}$）	High	*J*	与 LLJ 相同，但为 $\|orig\| > 100$，且时间窗 时间窗=10s	hhj	高（$\sim 3e^{-3}$）	High	*J*	与 LLJ 相同，但差异总和为 ＜0.7m，且时间窗时间窗=2s

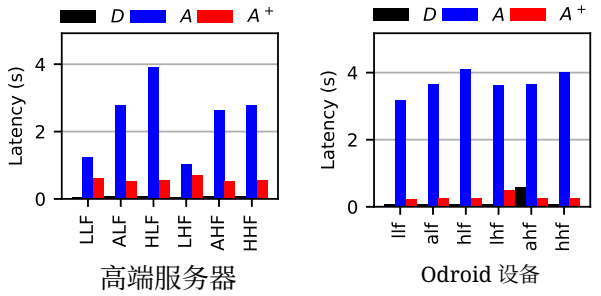

图 8: *FM* **- 所有实验的平均延迟。**

图 7 和图 8 对所有实验中的吞吐量和延迟进行了比较。对于 *D*，吞吐量（图 7）的数值不会受到选择率的显著影响，但会受到每元组计算成本的影响。不过，*A* 并非如此：当 *FM* 的选择率接近 0 时，这两个

A 算子 *X*（见 § 4.4）仅处理少量元组。因此，尽管定义 E_{FM} 的 *A* 效率不如 *D*，但 E_{FM} 和 *D* 的性能处于同一数量级（例如，LLF 中 *A* 的吞吐量低 34%）。当输入到 *X* 的速率增加时，吞吐量会下降，例如在 HLF 中降至原来的 90%。*X* 对 *A* 性能影响的进一步证据来自 A^+ 的性能：其选择率不会影响吞吐量（与 *D* 相同），因此能获得更高的最大可持续吞吐量；在基于 Odroid 的实验中，其性能与 *D* 差异可以忽略不计。

对于延迟指标，我们可以注意到，正如预期的那样，*A* 和 A^+ 的延迟始终高于 *D*。然而，我们注意到，当在 Odroid 等边缘设备上运行分析时，*A* 和 *D* 之间的延迟差异会变小。我们还注意到，随着选择率的提高，*A* 的延迟持续增加。这可以归因于元组在 *X* 算子中的循环。*D* 也呈现出延迟逐渐增加的趋势，并且

在 Odroid 上执行时也是如此。然而，这一趋势在高端服务器上更加明显。在后者上，*D* 的延迟比 *A* 和 A^+ 低几个数量级。这与我们的预期一致，即 *D* 的评估基于单个无状态算子。最后请注意，即使在高端服务器上，A^+ 的延迟仍保持在亚秒级范围内。对于许多现实世界中的流应用而言，这一延迟是可以接受的。

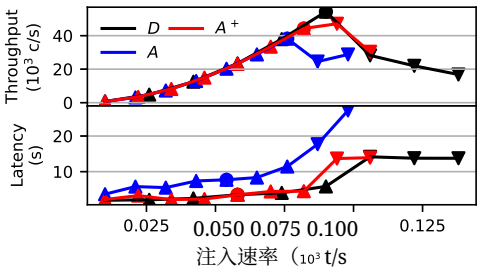

图9：ahj - 吞吐量/延迟与注入速率的关系。

J 算子。图 9 展示了实验 ahj 中 *J* 算子的吞吐量和延迟。在这种情况下，*D* 和 A^+ 实现的表现相近，而 *A* 的延迟会随着速率提高而增长得更快。对于 *A* 和 A^+，给定窗口 γ 的所有比较会在 γ 的右边界先于 *A* 的水位线到达时一次性完成；而对于 *D*，比较则随着元组到达而进行。由于 *A* 还需要对 *X* 进行后续展开，因此窗口过期时需要执行更多工作，进而导致吞吐量降低、延迟升高。

图10和图11包含所有实验的吞吐量和延迟比较。其趋势与 *FM* 类似，但绝对值方面存在细微差异。特别是，A^+ 和 *J* 算子表现出的差异都可以忽略不计。在某些情况下，我们还注意到 *A* 和 *D* 之间存在细微差异，尤其是在 *A* 和 *D* 运行

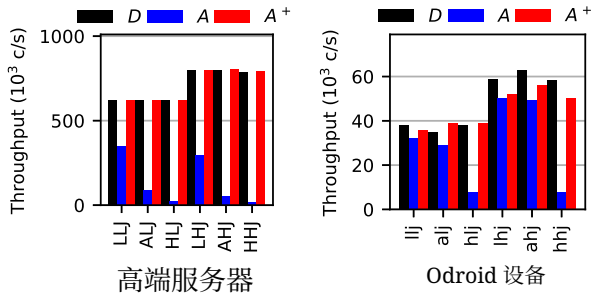

图 10： *J*- 所有实验的平均吞吐量。

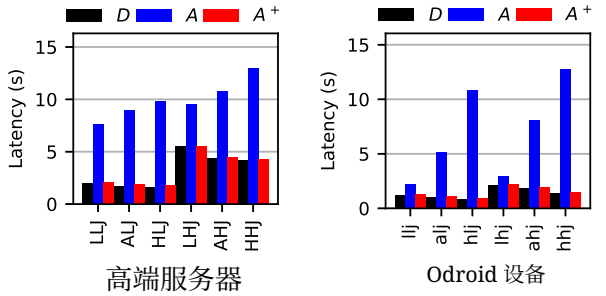

图 11： *J*- 所有实验的平均延迟。

于 Odroid 设备上时，例如 llj 实验。在所有实验中，我们还观察到，延迟增长趋势主要出现在 *A* 算子中，而在本例中，A^+ 和 *D* 在所有实验中的延迟均具有可比性。

总结。 我们比较了两个现实应用中多项实验的吞吐量和延迟性能，并采用了能够代表现代信息物理系统中计算能力两端的硬件。

正如预期，专用算子的表现更好。然而，基于最简 *A* 算子或语义更丰富的 A^+ 算子的实现往往也能取得相当的结果，尤其是在对 *J* 进行有状态分析时，因为 *D* 和 A/A^+都需要依赖水印来推进分析并生成结果。我们认为，鉴于使用 A^+ 而非 *A*使性能差距大幅缩小，采用语义更丰富的 *A*（例如，不仅能在窗口实例到期时计算结果，还能生成中间结果）或许可以进一步缩小这一差距；我们计划在未来的工作中探索这一点。

7 相关工作

此处考虑的 Dataflow 模型 [1] 是现代分布式流处理引擎（流处理引擎）中的事实标准。

DataFlow模型的核心思想或许早已体现在开创性的 MapReduce [20] 数据处理系统中。21世纪初，MapReduce 展示了这种编程抽象的优势：开发人员只需定义各个算子所执行的计算，而系统运行时则负责处理与并发与分布式执行相关的所有复杂问题，例如部署、调度、通信、并发控制和容错。同一时期，Aurora [21] 等早期流处理引擎已经基于 DataFlow编程与执行模型构建。

如今，Dataflow 已广泛应用于分布式数据处理系统 [22],，例如 Apache Spark [23],、Apache Flink [24],、ApacheStorm [25]和 Heron [26],、Kafka Stream [27], 以及 Samza [28], 等。它还为一些分布式数据库系统中的分析查询提供执行支持，例如 F1 [29]。关系型/SQL流处理 [30],、RDF流处理、流推理平台 [31], 乃至 ActiveSheets [32]等电子表格框架中，都包含了基于时间的窗口等核心概念；这些概念是我们所需的极简聚合算子（见 § 2.1）的基础。

据我们所知，尚无先前研究探讨基于 Dataflow 的流处理引擎中算子之间的关系，并像本文这样寻求一组用于表达查询的核心算子。不过，还有一些相关研究领域或许能为未来研究提供启发。

例如， [33] 中的工作为流应用定义了一种演算，并展示了如何将 CQL [34] 等高级语言移植到该演算中。不过请注意， [33] 将语义封装在彼此正交且不透明的函数中，而我们则明确依赖基本的 Dataflow构造，来展示流处理引擎内部及彼此之间的语义重叠。

一些研究聚焦于流处理引擎各个方面的语义。例如，SECRET模型提供了一组最小参数，用于精确定义窗口 [35],，从而识别不同系统之间可能存在的差异 [36]。Akidau等人 [37] 研究了水印的语义以及两种系统实现之间的差异。Gévay等人 [38] 综述并分析了Dataflow系统中处理迭代的不同方法。其他研究则关注基于Dataflow的流处理引擎之上的高层抽象，例如通过关系型数据处理抽象，利用数据流与关系表之间的二元性 [39]。Fernandez等人 [40]提出了一种面向对象编程模型，可自动转换为Dataflow图。还有一些研究探讨如何扩展基于Dataflow的流处理引擎的表达能力。Naiad [41] 提供了一种扩展的编程模型，通过显式的向量时间戳来控制流元素如何遍历嵌套循环。CIEL [42]提供了动态实例化新算子的原语，从而允许在运行时定义计算图，例如根据数据值进行定义。还有多种提案旨在为Dataflow 模型添加某种形式的可变状态。TSpoon [43] 提供了查询和访问算子状态的原语，并允许开发者指定计算图中需要以事务语义访问和更新的部分。相应地，S-Store [44] 在关系数据库内核中实现了一个流处理引擎，从而为有状态算子的内部状态提供事务语义。

流处理还存在其他编程模型 [45]。特别是，复杂事件识别（CER） [46]将流元素视为事件的发生，并旨在识别此类事件的模式。有趣的是，CER 运算符的语义已得到详细研究 [47]: 本文旨在为 Dataflow 模型提供类似的贡献。在 CER 的语境下，运算符之间的等价性已被用于将查询重写为一种等价形式，从而实现优化执行 [48]。我们认为，更深入地理解算子之间的语义关系

或许能为 Dataflow 模型带来类似的策略，补充现有的优化策略 [49],，并有望利用更高层次的查询定义 [50]。

8 结论

我们的工作旨在找出一组基于 DataFlow 模型、为所有流处理引擎所共有的核心算子，这组算子足以保证常见流处理管道的语义。我们的形式化分析表明，单个精简的聚合算子足以保证 DataFlow 模型中定义的所有常见无状态和有状态算子的语义。这一发现意味着，一个流处理引擎中的流应用可以轻松移植到其他流处理引擎或大数据框架，只要它们能够支持这样一个精简的聚合算子。此外，我们还发现，在各种真实场景中，聚合算子与其他流处理算子之间的性能差异可能很小。

我们还发现，聚合算子可用于定义超出常见基于窗口的算子范围的有状态算子。我们计划在未来的工作中进一步探讨这一观察结果，并研究基于聚合算子组合的流应用在分布式/并行部署中的性能变化。

带循环的 A A 水印

我们在此详细说明关于水位线和循环元组处理的假设 **C2** 和**C3**（见第 3 节），并讨论如果某个流处理引擎根据 DataFlow 模型维护 A'的水位线，却不针对由循环向 A馈送数据采取额外操作，可能会发生什么。

假设 A为键 k维护一个窗口实例 γ，覆盖范围为 [10:00−11:00）。要使 γ产生一个输出元组 t_o，W_A^{ω}必须增长到 11:00 或更晚。如果携带时间 12:00 的 W^j是触发生成 t_o的水位线，而 t_o再次被馈送至 A，则根据 $t_o.\tau$，t_o将构成迟到数据，即 11:00-δ和 W_A^{ω}，即 12:00（见第 2.1 节）。一般而言，鉴于**C1**，$t_o \in trig$（W^i）可以携带 $[W^j-D, W^j$中的时间戳，将 A的允许迟到时间设置为$L \geq D$，可能会防止 t_o未被 A处理。

不过，这样的预防措施还不够。如果不对 A 交错处理其输入元组/水位线（包括来自循环的元组）的顺序作出额外假设，就无法排除这种情况：在接收到 t_o时，A处理来自其上游对等算子的水位线，导致 W_A^{ω}增长到任意值。如果 $L=D$和 $D=1$小时，并且在接收到 t_o时，W_A^{ω}增长到例如 12:05，那么t_o仍不会由 A处理。

最后还要指出，即使在 A 处任意交错输入元组/水位线并未导致经由 A的循环传递的 t_o无法得到处理，将 W^i从 A 前向传递给其有状态的下游对等算子（参见§ 2.3），也要求后者通过其允许迟到时间来处理迟到数据（参见§ 2.4）。

B 引理的证明

证明。（引理 1）t所贡献的窗口实例的左边界 $\gamma.l$，对于一个滚动窗口，可以计算为 $\lfloor t.\tau/WA \rfloor WA$。如果 $WA=\delta$，则 $\lfloor t.\tau/WA \rfloor = t.\tau/WA$，因此 $\gamma.l = t.\tau$。此外，$t_o.\tau$被设置为 $\gamma.l+WS-\delta=t_i.\tau$。□

证明。（引理 2）通过反证法，假设 t_o^l和 t_o^m由 A_1 产生，并且满足 $t_o^l = t_o^m$。注意，A_1 为窗口实例 γ 产生的任何元组，都与落入 γ 中的元组具有相同的时间戳（见引理 1）；同时注意，当 $t_o[2] \neq -1$ 时，A_1 不会改变$t_o[1]$（列表 3，第 8—9 行）。t_o^l和 t_o^m的存在意味着存在由 A_1 产生的 $t_o^{l'}$和 $t_o^{m'}$，使得 $t_o^{l'}[0{:}1] = t_o^l[0{:}1]$，$t_o^{l'}[2]=0$ $t_o^{m'}[0\ 1] = t_o^m[0\ 1]$，$t_o^{m'}[2]=0$ 3，3 7 、.....以及（列表 L - ）。假设$t_o^{l'}$和 $t_o^{m'}$由 A_1处理来自 S_E的两个相同元组 t_i^l和t_i^m产生，这将导致矛盾，因为如果 $t_i^l = t_i^m$，那么二者会落入同一个窗口实例 γ，而 γ只会产生一个输出元组。假设 $t_o^{l'}$和 $t_o^{m'}$ 由 A_1 处理来自 $t_i^{m_1}$和$t_i^{m_2}$等的两个或更多输入元组 $t_i^{l_1}$、$t_i^{l_2}$等产生，并且满足 $t_i^{l_j} \neq t_i^{m_j}$、$\forall l_j$、m_j，同样会导致矛盾，因为 $t_i^{l_j} \neq t_i^{m_j}$意味着串接$t_i^{l_1}[1]^\frown t_i^{l_2}[1]^\frown ...$ 和 $t_i^{m_1}[1]^\frown t_i^{m_2}[1]^\frown ...$ 也不同，从而意味着$t_o^{l'} \neq t_o^{m'}$。□

证明。**（引理 3）**水位线W始终会在满足条件 $W \leq B$（列表 4、L17-21、L23-24）时被前向转发。

如果 $B=\infty$，那么W是在任何元组 t之前接收到的水位线，因为 B在 ∞时初始化，或者 $succ\Gamma$ 为空。在前一种情况下，S_{A_2} 中没有仍待处理的待处理元组。在后一种情况下，来自 E的任何元组 t在调用 $processT(t)$时使 $succ\Gamma[t.\tau]$ 增加，随后都接收到了所有 $t\prime \in succ(t)$；这些元组在相关调用 $processT(t\prime)$时使 $succ\Gamma[t\prime.\tau]$ 降至 0。因此，A_1尚无待处理的元组。

如果 $B \neq \infty$，那么W与来自 A_1 且仍待由 A_1 处理的最早元组 $t\prime$之间的距离小于或等于 L。当 $t\prime$在 W之后于 A_1 被接收时，条件 $t\prime.\tau+\delta > W_{A_1}^{\omega}+L$将得到满足（注意，根据引理 1$t\prime$落入一个窗口实例，其左边界等于 $t\prime.\tau$），因为这意味着 $W_{A_1}^{\omega} < t\prime.\tau+\delta+L$，并且由 S_E馈送至 A_1的最新水位线W满足 $W \leq t\prime.\tau+L$。□

证明。**（引理 4）**如果W在接收时即被转发（列表.5，L19），那么 $|succ\Gamma|=0$。因此，$\forall t \in succ(trig(W\prime))$，从而$W\prime < W$，在$W$之前被馈送至 A_2。$forwardW$的所有其他调用都会向 A_2馈送一个等于（列表.5，L12）或小于（列表.5，L15，L22）$succ\Gamma$中某个条目的时间戳 τ的水位线，且该条目前面没有计数大于 0 的其他条目。因此，只有在任何 $t \in succ(trig(W\prime))$之后，$\tau$才会作为水位线馈送至 A_2，从而$W\prime \leq \tau$。□

证明。**（引理 5）**在图 5 中，无状态的 FM_1为传递给 A_1的元组定义了一组通用属性，无论这些元组来自 S_I还是 S_A。随后由 A_1执行核心功能。根据A_1定义的Γ，只有两个连续的窗口实例 $\gamma_l = [lP, lP+P+\delta)$ 和 γ_{l+1} [$l+1$ P $l+1$ $P+P+\delta$ [$l+1$ P $l+1$ $P+\delta$ ( ),( )) 会在 ( ),( )) 上重叠。尽管如此，落在后一个区间内的元组只有在 (List.6,L6) 中才会被处理。因此，A_1 会对每个元组恰好处理一次。我们还注意到，一旦传递给 A_1的第一个元组 t得到处理，就会调用 f_c (List.6,L9)创建一个状态元组 t_s。设 γ_l为处理 t的第一个窗口。由此产生的 t_s将具有 $t_s.\tau=(l+1)P$，随后会在窗口实例 γ_{l+1}中处理，因为 **C1**、**C2**和**C3** 可防止 t_s因迟到而无法得到处理。通过迭代，t_s会将基于所有已处理元组（而不只是某个单独窗口实例内的元组）的状态，从 γ_l传递到所有后续窗口。最后，FM_2会对周期性为 P的状态元组运行 f_o，因为 P是 A_1的WA，从而保证 O的语义。 □

参考文献

[1] T. Akidau、R. Bradshaw、C. Chambers、S. Chernyak、R. J. Fernández-Moctezuma、R. Lax、S. McVeety、D. Mills、F. Perry、E. Schmidt 和 S. Wittle，《Dataflow 模型：在大规模、无界、乱序数据处理中平衡正确性、延迟与成本的实用方法》，Proc. Endowment，第 8 卷，第 12 期，第 1792–1803 页，2015 年。[2]《Apache Flink》，https://flink.apache.org，访问日期：2023-1-27。[3]《Apache Storm》，http://storm.apache.org，访问日期：2019-3-1。[4] Liebre 流处理引擎，“https://github.com/vincenzo-gulisano/liebre”，2022 年，访问日期：2022-6-27。[5]《Apache Beam》，https://beam.apache.org/，访问日期：2020-11-12。[6] V. Gulisano、D. Palyvos-Giannas、B. Havers 和 M. Papatriantafilou，《事件时间顺序在数据流分析中的作用》，载于第 14 届 ACM 分布式与事件驱动系统国际会议论文集，2020 年，第 214–217 页。[7] V. Gulisano、Y. Nikolakopoulos、D. Cederman、M. Papatriantafilou 和 P. Tsigas，《通过并发算法设计和新型抽象数据类型实现高效的数据流多路聚合》，ACM 并行计算汇刊，第 4 卷，第 2 期，第 11:1–11:28 页，2017 年 10 月。[8] V. Gulisano、Y. Nikolakopoulos、M. Papatriantafilou 和 P. Tsigas，《ScaleJoin：一种确定性、互不相交并行且抗倾斜的流连接方法》，IEEE 大数据汇刊，第 7 卷，第 2 期，第 299–312 页，2021 年。[9] J. Teubner 和 R. Mueller，《足球运动员会如何进行流连接》，载于 2011 年 ACM SIGMOD 数据管理国际会议论文集，2011 年。[10] V. Gulisano、H. Najdataei、Y. Nikolakopoulos、A. V. Papadopoulos、M. Papatriantafilou 和 P. Tsigas，《Stretch：用于可扩展弹性流处理的虚拟无共享并行》，IEEE 并行与分布式系统汇刊，第 33 卷，第 12 期，第 4221–4238 页，2022 年。[11] A. Trofimov、N. Sokolov、N. Marshalkin、I. Kuralenok 和 B. Novikov，《分布式流数据流中的子流管理》，载于第 16 届 ACM 分布式与事件驱动系统国际会议论文集，2022 年，第 55–66 页。[12] M. Faruqui、E. Pavlick、I. Tenney 和 D. Das，《WikiAtomicEdits：用于语言与话语建模的维基百科编辑多语言语料库》，载于 EMNLP 论文集，2018 年。[13] I. S. Mohamed、A. Capitanelli、F. Mastrogiovanni、S. Rovetta 和 R. Zaccaria，《利用机器学习技术和二维距离数据检测、定位和跟踪托盘》，arXiv 预印本 arXiv:1803.11254，2018 年。[14] ——，《标准欧式托盘的二维激光测距仪扫描数据集》，Data in Brief，第 103837 页，2019 年。[15] HardKernel，《Odroid-XU4》，2020 年。[在线]。网址：http://www.hardkernel.com [16] X. Fu、T. Ghaffar、J. C. Davis 和 D. Lee，《Edgewise：面向边缘的更优流处理引擎》，载于 USENIX年度技术会议（ATC）19。美国华盛顿州：USENIX，2019 年 7 月，第 929–946 页。[17] D. Palyvos-Giannas、V. Gulisano 和 M. Papatriantafilou，《Haren：面向数据流应用的临时线程调度策略框架》，载于第 13 届 ACM 分布式与事件驱动系统国际会议论文集，DEBS '19 系列。德国达姆施塔特：ACM，2019 年，第 19–30 页。[18]《Apache Flink——重构流迭代流程控制》，https: // issues.apache.org/jira/browse/FLINK-2497?jql=project%20%3D%20FLINK%20AND%20text%20~%20%22loop%20deadlock%22，访问日期：2023-1-27。[19] V. Gulisano、A. V. Papadopoulos、Y. Nikolakopoulos、M. Papatriantafilou 和 P. Tsigas，《流连接的性能建模》，载于第 11 届 ACM 分布式与事件驱动系统国际会议论文集，2017 年，第 191–202 页。[20] J. Dean 和 S. Ghemawat，《MapReduce：大型集群上的简化数据处理》，ACM 通讯，第 51 卷，第 1 期，第 107–113 页，2008 年。[21] D. J. Abadi、D. Carney、U. Çetintemel、M. Cherniack、C. Convey、S. Lee、M. Stonebraker、N. Tatbul 和 S. Zdonik，《Aurora：一种新的数据流管理模型与架构》，《VLDB期刊》，第 12 卷，第 2 期，第 120–139 页，2003 年。[在线]。网址：https://doi.org/10.1007/s00778-003-0095-z [22] A. Margara、G. Cugola、N. Felicioni 和 S. Cilloni，《分布式数据密集型系统的模型与综述》，2022 年。 [在线]。网址：https://arxiv.org/abs/ 2203 10836。M. Zaharia、R. S. Xin、P. Wendell、T. Das、M. Armbrust[23]、A. Dave、X. Meng、J. Rosen、S. Venkataraman、M. J. Franklin、A. Ghodsi、J. Gonzalez、S. Shenker 和 I. Stoica，《Apache Spark：用于大数据处理的统一引擎》，ACM 通讯，第 59 卷，第 11 期，第 56–65 页，2016 年。[24] P. Carbone 、 A. Katsifodimos 、 S. Ewen 、 V. Markl 、 S. Haridi 和 K. Tzoumas，《Apache Flink：在单一引擎中实现流处理与批处理》，IEEE 数据工程通报，第 38 卷，第 4 期，第 28–38 页，2015 年。[25] A. Toshniwal、S. Taneja、A. Shukla、K. Ramasamy、J. M. Patel、S. Kulkarni、J. Jackson、K. Gade、M. Fu、J. Donham、N. Bhagat、S. Mittal 和 D. Ryaboy，《Storm@Twitter》，载于数据管理国际会议论文集，SIGMOD '14 系列。ACM，2014 年，第 147–156 页。[26] S. Kulkarni、N. Bhagat、M. Fu、V. Kedigehalli、C. Kellogg、S. Mittal、J. M. Patel、K. Ramasamy 和 S. Taneja，《Twitter Heron：大规模流处理》，载于数据管理国际会议论文集，SIGMOD '15 系列。ACM，2015 年，第 239–250 页。[27] B. Bejeck，《Kafka Streams 实战：使用 Kafka Streams API 构建实时应用和微服务》。Manning，2018 年。[28] S. A. Noghabi、K. Paramasivam、Y. Pan、N. Ramesh、J. Bringhurst、I. Gupta 和 R. H. Campbell，“Samza：LinkedIn 上的有状态可扩展流处理”，《VLDB基金会会刊》，第 10 卷，第 12 期，第 1634–1645 页，2017年。[29] J. Shute、R. Vingralek、B. Samwel、B. Handy、C. Whipkey、E. Rollins、M. Oancea、K. Littlefield、D. Menestrina、S. Ellner、J. Cieslewicz、I. Rae、T. Stancescu 和 H. Apte，“F1：可扩展的分布式 SQL 数据库”，《VLDB基金会会刊》，第 6 卷，第 11 期，第 1068–1079 页，2013年。[30] E. Begoli、T. Akidau、F. Hueske、J. Hyde、K. Knight 和 K. Knowles，“一统天下的 SQL——管理流与表的高效且符合语法习惯的方法”，载于《2019年国际数据管理会议论文集》，2019年，第 1757–1772 页。[31] A. Margara、J. Urbani、F. van Harmelen 和 H. Bal，“流式处理万维网：对动态数据进行推理”，《网络语义学期刊》，第 25 卷，第 24–44 页，2014年。[32] M. Hirzel、G. Baudart、A. Bonifati、E. Della Valle、S. Sakr 和 A. Akrivi Vlachou，“大数据时代的流处理语言”，ACM SIGMOD Record，第 47 卷，第 2 期，第 29–40 页，2018年。[33] R. Soulé、M. Hirzel、R. Grimm、B. Gedik、H. Andrade、V. Kumar 和 K.-L. Wu，“流处理语言的通用演算”，载于《编程语言与系统：第19届欧洲编程研讨会（ESOP 2010），作为欧洲软件理论与实践联合会议（ETAPS 2010）的一部分于2010年3月20日至28日在塞浦路斯帕福斯举行。第19卷论文集》。施普林格，2010年，第 507–528 页。[34] A. Arasu、S. Babu 和 J. Widom，“CQL 连续查询语言：语义基础与查询执行”，《VLDB期刊》，第 15 卷，第 121–142 页，2006年。[35] I. Botan、R. Derakhshan、N. Dindar、L. Haas、R. J. Miller 和 N. Tatbul，“Secret：一种用于分析流处理系统执行语义的模型”，《VLDB基金会会刊》，第 3 卷，第 1–2 期，第 232–243 页，2010年。 [在线]。链接：https://doi.org/ 10 14778 1920841 1920874。/。L. Affetti、R. Tommasini、A. Margara[36]、G. Cugola 和 E. Della Valle，“定义流处理引擎的执行语义”，《大数据期刊》，第 4 卷，第 1 期，第 1–24 页，2017年。[37] T. Akidau、E. Begoli、S. Chernyak、F. Hueske、K. Knight、K. Knowles、D. Mills 和 D. Sotolongo，“流处理系统中的水位线：Apache Flink 与 Google Cloud Dataflow 的语义和比较分析”，《VLDB基金会会刊》，第 3 期，2020年。[38] G. E. Gévay、J. Soto 和 V. Markl，“分布式数据流系统中的迭代处理”，《ACM计算调查》，第 54 卷，第 9 期，2021年。 [在线]。链接：https://doi.org/ 10 1145 3477602。/ M. J. Sax、G. Wang、M. Weidlich 和 J.-C. Freytag[39]，“流与表：同一枚硬币的两面”，载于《国际实时商业智能与分析研讨会论文集》，丛书 BIRTE '18。纽约，NY，美国：计算机协会，2018年。 [在线]。链接：https://doi.org/ 10 1145 3242153 3242155。/。R. C. Fernandez、M. Migliavacca、E. Kalyvianaki[40]、和 P. Pietzuch，“在命令式大数据处理中显式表示状态”，载于《USENIX年度技术会议论文集》，丛书 ATC'14。USENIX 协会，2014年，第 49–60 页。[41] D. G. Murray、F. McSherry、R. Isaacs、M. Isard、P. Barham 和 M. Abadi，“Naiad：及时数据流系统”，载于《操作系统原理研讨会论文集》，丛书 SOSP '13。ACM，2013年，第 439–455 页。[42] D. G. Murray、M. Schwarzkopf、C. Smowton、S. Smith、A. Madhavapeddy 和 S. Hand，“CIEL：用于分布式数据流计算的通用执行引擎”，载于《网络系统设计与实现会议论文集》，丛书 NSDI'11。USENIX 协会，2011年，第 113–126 页。[43] L. Affetti、A. Margara 和 G. Cugola，“TSpoon：流处理器上的事务”，《并行与分布式计算期刊》，第 140 卷，第 65–79 页，2020年。[44] U. Cetintemel、J. Du、T. Kraska、S. Madden、D. Maier、J. Meehan、A. Pavlo、M. Stonebraker、E. Sutherland、N. Tatbul 等，“S-Store：面向高数据速率应用的流式 NewSQL 系统”，《VLDB基金会会刊》，第 7 卷，第 13 期，第 1633–1636 页，2014年。[45] G. Cugola 和 A. Margara，“信息流处理：从数据流到复杂事件处理”，《ACM计算调查》，第 44 卷，第 3 期，2012年。 [在线]。链接：https://doi.org/ 10 1145 2187671 2187677。/。N. Giatrakos、E. Alevizos[46]、A. Artikis、A. Deligiannakis、和 M. Garofalakis，“大数据时代的复杂事件识别：综述”，《VLDB期刊》，第 29 卷，第 1 期，第 313–352 页，2019年7月。 [在线]。链接：https://doi.org/10.1007/s00778-019-00557-w [47] A. Grez、C. Riveros、M. Ugarte、和 S. Vansummeren，“复杂事件识别的形式化框架”，《ACM数据库系统汇刊》，第 46 卷，第 4 期，2021年12月。 [ ] 10 1145 3485463 在线。链接：https://doi.org/。/ N. P. Schultz-Møller、M. Migliavacca[48]、和 P. Pietzuch，“通过查询重写实现分布式复杂事件处理”，载于《第三届ACM分布式事件系统国际会议论文集》，丛书 DEBS '09。纽约，NY，美国：计算机协会，2009年。 [在线]。链接：https://doi.org/ 10 1145 1619258 1619264。/。M. Hirzel、R. Soulé、S. Schneider、B. Gedik[49]、和 R. Grimm，“流处理优化目录”，《ACM计算调查》，第 46 卷，第 4 期，第 1–34 页，2014年。[50] M. Guerriero、D. A. Tamburri、和 E. D. Nitto，“Streamgen：分布式流应用的模型驱动开发”，《ACM软件工程方法学汇刊》，第 30 卷，第 1 期，2021年。 [在线]。链接：https://doi.org/10.1145/3408895