arXiv:2301.12030v1 [cs.DB] 27 Jan 2023

TiLT：一种以时间为中心的流查询优化与并行化方法

Anand Jayarajan
anandj@cs.toronto.edu 多伦多大学、Vector Institute 加拿大

Wei Zhao*
xiaoteemo.zhao@mail.utoronto.ca
多伦多大学 加拿大

Yudi Sun*
y
udi@cs.toronto.edu
多伦多大学 加拿大

Gennady Pekhimenko
pekhimenko@cs.toronto.edu 多伦多大学、Vector Institute 加拿大

摘要

流处理引擎（SPEs）广泛用于在无界的按时间排序的数据流上执行大规模流分析。现代流分析应用表现出多样化的计算特征，并对延迟和吞吐量提出严格要求。多年来，研究人员一直致力于构建硬件高效流处理引擎（SPEs），支持多种查询优化、并行化与执行策略，以满足大规模流分析应用的性能要求。然而，在本文中，我们观察到，由于当前流处理引擎固有的多项设计局限，这些策略往往无法很好地泛化到许多真实世界流式分析应用。我们进一步指出，这些局限源于现代流处理引擎所采用的基本设计选择和查询表示模型存在不足。为应对这些挑战，我们首先提出*TiLT*，一种新颖的中间表示（IR），提供高度表现力的时间查询语言，适用于有效的查询优化与并行化策略。随后，我们为 TiLT 构建了编译器后端，将此类优化应用于流查询，并生成硬件高效代码，以在多核流查询执行中实现高性能。我们证明，在八个真实世界流式分析应用中，与最先进的流处理引擎（例如 Trill）相比，TiLT 的吞吐量最高可提升 326×（20.49× 平均提升）。TiLT 源代码可在 https://github.com/ampersand-projects/tilt.git获取。

CCS 概念

• **信息系统** → **数据流处理**；非关系型引擎查询语言； 流管理；• **计算方法** → **并行编程语言**； • **软件及其工程** → 并行编程语言；**领域专用语言<style>**。

*作者对本研究做出了同等贡献。

特此免费授予以个人或课堂使用为目的制作本作品部分或全部内容的数字或纸质副本的许可，但前提是不得出于营利或商业利益制作或分发副本，且副本必须包含本声明以及首页上的完整引文。本作品中第三方组件的版权必须得到尊重。如需用于其他用途，请联系版权所有者/作者。*ASPLOS '23，2023*年*3*月*25*日至*29*日，加拿大不列颠哥伦比亚省温哥华©2023年版权归版权所有者/作者所有。ACM ISBN 978-1-4503-9916-6/23/03。
https://doi.org/10.1145/3575693.3575704

关键词

流数据分析、时序查询处理、中间表示、编译器

ACM 参考格式：
Anand Jayarajan、Wei Zhao、Yudi Sun 和 Gennady Pekhimenko。2023年。TiLT：一种以时间为中心的流查询优化与并行化方法。载于第*28*届 *ACM* 编程语言与操作系统架构支持国际会议论文集，第 *2* 卷（*ASPLOS '23*），*2023*年*3*月*25*日至*29*日，加拿大不列颠哥伦比亚省温哥华。ACM，美国纽约州纽约市，16页。https://doi.org/10.1145/3575693.3575704

1 引言

流处理应用广泛应用于各行各业，用于对无界、有序且高频的事件流执行实时分析和离线分析。例如，社交媒体公司（如 Twitter、Meta）使用点击流分析来投放广告 [26, 49]，银行机构分析购买趋势，以识别欺诈交易 [58]，投资公司则根据实时股价开展高频交易 [1]。近年来，流处理在农业等非传统领域中的应用也日益广泛 [51]，气候科学 [5]，能源／制造业 [42]，和医疗保健 [19]。这些应用涉及需要对数据流的时间维度进行细粒度控制的计算，例如计算一段时间内的移动平均值，或查找跨数据流事件之间的时间相关性。此外，流式计算通常运行时间较长，并以连续方式处理数据，同时还必须满足严格的延迟和吞吐量要求 [45]。

流处理引擎（SPEs）是为满足流分析应用日益增长的性能需求而设计的专用系统。现代流处理引擎 [4, 6, 8, 11, 14, 19, 24, 25, 47, 59]提供了熟悉的类 SQL 时序查询语言接口，用于编写复杂的流分析应用。许多流行的流处理引擎 [4, 6, 8, 52, 59] 被设计为横向扩展系统，通过在大型机器集群上扩展查询执行来满足流应用的性能要求。尽管横向扩展型流处理引擎得到了广泛采用，但其硬件利用率显著偏低，因此资源消耗非常高 [32,60]。这催生了若干纵向扩展流处理引擎 [11, 14, 19, 33, 34, 47]，其采用注重硬件的设计，以更好地利用现代多核机器。表 1 展示了在热门的雅虎

表1：YahooStreaming Benchmark 的吞吐量（百万事件/秒）[31]（在一台32核机器上）

横向扩展[8, 59]		纵向扩展 [11, 14, 34, 47]			
Spark	Flink	Trill	StreamBox	Grizzly	LightSaber
0.14	0.59	34.07	167.19	118.74	296.40

流式基准测试 [31] 中的性能比较结果。如图所示，在相同的硬件预算 [60]，下，纵向扩展流处理引擎能够实现 100 – 1000× 更高的吞吐量 [30]，这使其成为大规模流处理领域中比横向扩展型流处理引擎更具成本效益的替代方案。

纵向扩展流处理引擎面临三项关键挑战，必须解决这些挑战才能为流查询实现较高的单机性能。首先，要提高流查询的硬件利用率，就需要支持有效的优化策略，以提升查询执行的缓存利用率并消除冗余计算。其次，与批处理应用不同，由于连续查询执行模型的存在，流应用并不会天然暴露数据并行性。因此，流处理引擎需要支持复杂的并行化策略，以充分利用多核机器中的所有处理器核心。最后，由于流查询属于长时间运行的工作负载，因此需要低开销运行时来满足其延迟要求。尽管当前研究已经对这一问题给予了广泛关注，但由于以下原因，现有的纵向扩展流处理引擎设计在许多真实世界流应用中仍无法同时应对这三项挑战。

当前的纵向扩展流处理引擎 [11, 33, 34, 61]遵循流应用的传统数据流表示，其中查询被定义为由时间操作构成的有向无环图（DAG），查询执行则通过解释数据流图来完成。尽管这种设计在概念上简单且易于扩展，但事实证明，这种查询执行模型会在运行时引入显著的解释开销 [37, 60]。此外，基于解释执行的流处理引擎中的查询优化大多是基于启发式的图级转换，例如对 DAG 中的操作重新排序 [6]。应用此类优化要求流查询必须精确匹配某些预定义规则，因此通常只适用于许多流应用中的有限场景。最后，许多此类系统只能从分区数据流中提取有限的并行化机会。

为解决基于解释执行的流处理引擎效率低下的问题，近期工作提出了基于编译器的解决方案 [14, 24, 47, 48]。这些方法通过从高级查询描述中自动生成硬件高效代码，为查询执行提供低开销的运行时。最先进的基于编译器的流处理引擎还支持算子融合等低级查询优化，通过使用寄存器或缓存在算子之间传递数据来最大化数据局部性，并且即使在非分区数据流上也能自动实现查询执行的并行化。然而，当前基于编译器的流处理引擎所提出的优化、并行化和代码生成策略，主要针对仅执行有限操作集的应用（例如流聚合）；对于包含更复杂操作的查询（例如流间连接），这些策略难以很好地泛化。这大幅限制了当前基于编译器的流处理引擎支持众多真实世界流式分析应用的能力。

在本项工作中，为支持流处理在广泛应用领域中的日益普及，我们的目标是提供一个用于流查询的有效且可泛化的优化与并行化策略基础设施。我们观察到，流处理引擎有限的优化与并行化能力，源于现代流处理引擎所采用的查询表示模型的根本局限。在当前所谓的以事件为中心的模型下，流查询通过基本时序操作构建，每个操作都定义了对一系列离散的时间有序事件的转换。尽管由于数据流本质上是由事件构成的无界按时间排序序列，这种表示模型对于流查询而言十分自然，但我们认为，这种以事件为中心定义时间操作的方式，并未揭示流查询中对于有效查询优化与并行化所必需的重要时间语义信息。由于流查询在本质上具有时间属性，我们认为，时间操作也应遵循一种从根本上基于时间的表示模型。

基于这一观察，我们提出了一种名为*TiLT* 的新型中间表示（IR），它采用以时间为中心的模型来定义流查询。不同于传统的事件中心模型，TiLT 中间表示使用时间对象、归约函数和时间表达式等新构造，将时间操作定义为在明确定义的时间域上进行的函数式变换。借助这些简单的构造，TiLT 提供了一种高度富有表现力的编程范式，用于表示多种流应用。同时，以时间为中心的流查询定义带来了使用传统查询表示模型难以实现的优化机会。此外，TiLT 查询的无副作用函数式定义暴露了固有的数据并行性，可用于并行化任意流查询。最后，我们为 TiLT 构建了一个编译器后端，可自动将逻辑流查询定义转换为硬件高效的可执行代码，并在广泛的流应用上实现高多核性能。

为了评估 TiLT 在多种应用上提供高性能的能力，我们准备了一个基准测试套件，其中包含八个流处理应用，代表股票交易、信号处理、工业制造、银行机构和医疗保健等领域中的现实世界流分析用例。在这些应用上，与业界最先进的基于解释的流处理引擎 Trill [11]相比，TiLT 实现了 6 – 322×（平均为20.49×）的更高吞吐量。这一加速主要来自两个方面：（i）TiLT 中以时间为中心的查询表示模型所实现的有效查询优化和并行化；（ii）基于编译器的流处理引擎设计，消除了查询解释和托管语言开销等解释型流处理引擎中常见的低效问题。我们还展示了 TiLT 能够与专为高效流聚合而设计的基于编译器的流处理引擎实现具有竞争力的性能。例如，在 Yahoo 流式基准测试 [12], 上，与业界最先进的基于编译器的流处理引擎 LightSaber [47] 和 Grizzly [14], 相比，TiLT 分别实现了 1.5× 和 3.8× 的更高吞吐量。

总而言之，我们的贡献如下：

- 我们指出，当前流处理引擎所用的查询表示模型在支持有效的优化与并行化策略方面存在局限。为解决这些局限，我们提出了一种名为 TiLT 的新型中间表示。

我们表明，TiLT 能够支持通用的优化与并行化策略，而传统查询表示模型难以支持这些策略。

- 我们为 TiLT 构建了一个编译器，可优化并并行化任意流查询，并生成硬件高效代码，以实现出色的多核性能。
- 我们准备了一套新的代表性基准测试套件，涵盖八种真实世界流式分析应用，涉及信号处理、股票交易、工业制造、银行服务和医疗保健领域。在这些应用中，我们证明 TiLT 能够实现 6 – 326× (20.49×平均而言）相比最先进的流处理引擎具有更高的吞吐量。TiLT 目前已开源，可通过以下地址获取：https://github.com/ampersand-projects/tilt.git

2 背景

流分析应用通常以连续方式处理无界的按时间排序的事件序列。表2展示了八种具有代表性的真实世界流式分析应用，涉及股票市场交易、信号处理、医疗保健、制造业和银行服务等领域。这些应用以高速处理数据，并对延迟和吞吐量有严格要求。例如，高频交易应用要求延迟低于一秒 [15, 28, 44]。此外，这些应用执行的计算通常需要对数据流的时间维度进行细粒度控制，例如采用不同的窗口策略来分析数据流中不断变化的趋势 [18, 41, 46]。

许多现代流处理引擎 [8, 11, 14, 19, 47, 59] 提供带有时间扩展的类 SQL 查询语言，用于编写此类复杂的流应用。这些语言提供了一套简单却极具表现力的基本时序操作词汇，其中每种操作都定义了对一个或多个数据流的转换。图1展示了流查询中常用的四种基本时序操作，以及它们对事件流执行的相应转换。不失一般性，数据流中的每个事件均由一个载荷值和一个有效时间区间表示。图中所示的 选择和筛选操作1a 和1b 分别遵循投影和选择操作的关系型 SQL 语义。这两种操作都会执行逐事件转换：前者修改每个事件的载荷字段，后者则根据用户在载荷上定义的谓词，有条件地过滤掉事件。图中所示的时态连接操作1c 将两个数据流连接成单个输出流。该连接操作的输出流包含与输入流中事件严格重叠区域相对应的事件。最后，数据流上的聚合操作通常在时间限定窗口上执行，该窗口由窗口大小和步长定义。例如，在一个 求和聚合操作中，若其定义于 窗口(大小, 步长)，则会计算每个 *size*秒[1] 窗口的总和，这些窗口之间相隔步长秒。图1d 展示了窗口大小为10 、步长为5的滑动窗口聚合。由于所有这些操作都被定义为对事件执行的转换，我们称其遵循时态算子定义的事件中心模型。

这些简单的原语操作可以组合起来，构造更复杂的流查询。图2a 展示了一个使用图1所示基本时序操作编写的流查询示例。该查询是股市趋势分析应用的简化版本 [18]，在表2中进行了描述。该查询通过计算某只股票的股价在10 和20 秒区间内的两个移动平均值来分析其价格趋势，并且每秒计算一次。首先使用两个窗口-求和 操作，然后使用选择操作，将求和结果除以各自对应的窗口大小。之后，使用时间连接操作，计算同期的10秒平均值与20秒平均值之间的差值。最后，筛选操作仅保留平均值差为正的事件。最终输出流中事件的有效区间对应于股价呈上升趋势的时间段。

定义好时间查询后，流处理引擎便可在内部处理这些查询的执行。许多广受欢迎的流处理引擎 [3, 4, 6, 8, 52, 59]主要通过将查询执行分布到大型机器集群中，来满足查询的性能需求。尽管这种方法广受欢迎，但先前的研究 [11, 19, 40, 60]表明，此类横向扩展型流处理引擎会大量消耗资源，其性能往往比相应的手工调优实现 2 – 3 慢几个数量级，因而造成计算和能源资源的严重浪费。这一发现促使人们开发了多种纵向扩展流处理引擎 [11, 14, 19, 24, 33, 34, 47, 61]，它们采用注重硬件的设计，以最大限度地提升流查询执行在单机上的性能。最先进的纵向扩展流处理引擎已证明，配备数百个处理器核心和数 GB 内存带宽的现代多核机器，足以满足大规模流处理工作负载的性能需求，并且可以成为成本更高的大型多机集群的高性价比替代方案 [30]。

3 现代流处理引擎的设计与局限

纵向扩展流处理引擎主要通过三种方式来实现较高的单机性能。首先，用户定义的查询会经过多轮优化，以削减冗余计算并提高数据局部性，从而提升查询执行期间的硬件利用率。其次，流处理引擎会采用不同的并行化策略，以利用多核机器中可用的并行处理器核心。最后，流处理引擎会努力提供低开销运行时，以满足长时间运行的流应用对延迟和吞吐量的要求。尽管受到了广泛关注，我们观察到，先前流处理引擎提出的查询优化、并行化和执行策略由于若干根本性的设计局限，在真实世界流式分析应用上难以良好泛化，具体问题如下所述。**查询优化机会有限**。当前流处理引擎采用的优化策略适用范围有限，通常无法覆盖广泛的真实世界流式分析应用。例如，大多数流处理引擎 [6, 11, 34]采用基于启发式的查询优化策略，而这类策略仅限于对查询中的单个操作进行替换或重新排序等基本图

[1]为便于讨论，我们使用秒作为时间单位。不过，本文定义同样适用于其他时间单位。

表 2：真实世界流式分析应用

分析应用	描述	查询运算符	数据集
基于趋势的交易 [18]	股价的移动平均趋势	平均值 (2)、连接、筛选	纽约证券交易所 [38]
相对强弱指数 [46]	股价动量指标	移位、连接、平均值（2）	
归一化 [57]	使用 Z 分数归一化事件值	平均值、标准差、连接	合成数据（随机生成的浮点值以 1000 赫兹频率）
信号插补 [54]	替换缺失信号值	平均值、移位、连接	
重采样 [55]	更改信号频率	选择、连接、移位、Chop	
Pan-Tomkins 算法 [39]	检测心电图中的 QRS 波群	自定义聚合(3)、选择、平均值	MIMIC-III 波形数据 [21]
振动分析 [41]	使用峰度监测机器振动，均方根和峰值因子指标	最大值、平均值(2)、连接 (2)，自定义聚合	轴承振动数据 [17]
欺诈检测 [58]	信用卡欺诈检测	平均值、标准差、移位、连接	Kaggle 信用卡数据 [22]

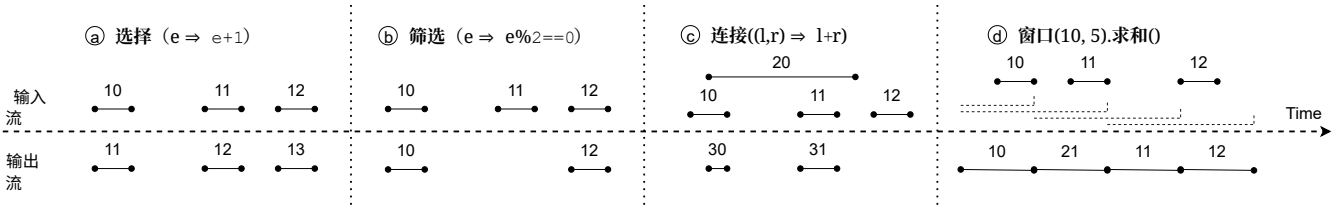


图 1： 常见的时间操作：(a) 选择，(b) 筛选，(c) 时间连接，以及 (d) 窗口求和

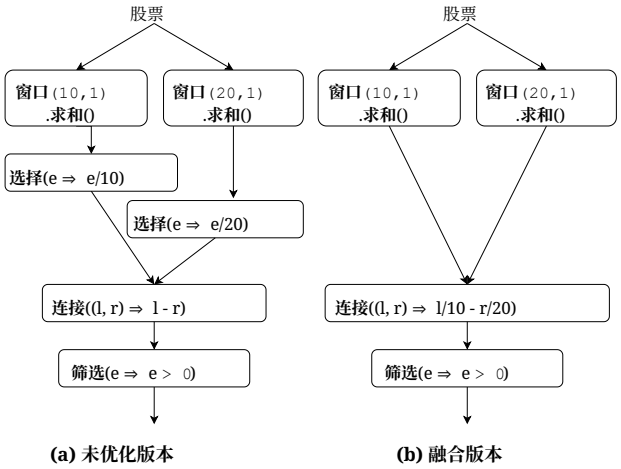


图2：股票价格趋势分析查询

转换，例如替换或重新排序查询中的单个操作 [6, 11]。例如，谓词下推 [6]是一种常见的优化方式，其中筛选操作（筛选）会被移至更靠近数据源的位置，以减少查询中其余操作需要处理的事件数量。然而，只有当筛选操作的谓词基于输入流中的事件定义时，这种优化才适用。例如，在图2a 所示的筛选操作依赖于其父级连接操作生成的结果，因此无法对示例查询应用谓词下推。

某些高级流处理引擎 [14, 47] 支持更复杂的底层优化，例如通过将多个算子合并为一个算子。例如，选择股票分析查询中的选择算子可以轻松地与连接算子融合，如图2b所示。这样可以避免算子之间不必要的数据移动，并使中间结果尽可能长时间地保留在寄存器或缓存中。遗憾的是，当前流处理引擎中实现的融合规则只能融合算子，直到遇到所谓的软流水线阻断算子 [37, 60] 。软流水线阻断算子是指需要先对输出事件进行部分物化，后续算子才能开始处理的算子。例如，在图2b中，窗口求和和连接算子都是软流水线阻断算子。将这些算子融合在一起并非易事，当前流处理引擎中的优化器无法处理此类情况。这极大地限制了融合优化在许多真实世界流应用中的适用性，因为这些应用的查询中通常包含多个流水线阻断算子。例如，表2 展示了各应用查询中使用的时间操作，每个查询都包含 2 – 6 个流水线阻断算子。

有限的查询并行化能力。 与批处理应用不同，流查询不会天然展现数据并行性，因为许多时间操作都存在顺序数据依赖（例如滑动窗口聚合）。因此，从流查询中提取数据并行性通常颇具挑战，许多流处理引擎依赖用户提供分区数据流，以实现查询执行的并行化。例如，趋势分析查询（图 2a）可以通过在对应不同股票的数据流上执行而轻松实现并行化。然而，这种方法所能提供的并行度受数据流中可用的唯一分区数量限制 [47]。此外，在医疗保健 [19]和制造业 [23], 中使用的某些流分析应用里，即使数据流的单个分区也可能包含以高达 1 – 40 KHz 的速率生成的事件。在此类情况下，并行化流查询执行需要更复杂的策略。

先前的研究 [9, 24, 47, 50] 提出了无需分区数据流即可自动从流查询中提取数据并行性的解决方案。然而，这些工作一直仅专注于基于窗口的聚合操作。例如，图 1c 中的滑动窗口求和可以通过先在 5秒翻转窗口上计算部分和[2]，然后将两个连续的部分和相加来实现并行化。由于翻转窗口彼此不重叠，因此可以按照 5秒窗口边界对数据流进行分区，并并行处理每个窗口。部分和的加法也可以通过并行归约 [47]实现并行化。然而，将这些策略扩展到任意流查询的并行化通常并非易事。例如，在示例查询中，由于查询包含多个滑动窗口和一个时间连接操作，因此难以确定股票价格流中的分区边界。我们观察到，当前纵向扩展流处理引擎中的查询并行化方法 [14, 47] 无法处理此类场景。

查询执行期间的运行时开销很高。当前流处理引擎采用的查询执行模型无法为广泛的实际应用提供较低的运行时开销。大多数流处理引擎 [8, 11, 33, 34, 59, 61] 采用基于解释的查询执行模型，也称为迭代器模型 [13, 29]。在该模型中，逻辑查询描述通过将查询中的每个时态算子映射到具体实现，被转换为数据流图。每个物理算子负责逐个处理事件或以微批次处理事件，并通过消息队列将生成的输出事件传递给图中的下一个算子。尽管这种设计得到了广泛采用，但已有研究 [14, 19, 37, 40, 47, 60] 表明，与相应的手工调优实现相比，解释型流处理引擎的性能通常慢了 1 – 2 几个数量级。这种低效率主要可归因于数据流图中算子之间的数据传输成本 [37, 60], 、对算子融合等有效优化策略支持不足 [60], ，以及由于固定大小微批处理而无法保持端到端数据局部性 [19]。

为了解决解释型流处理引擎的低效问题，近期研究提出了基于编译器的解决方案 [14, 24, 47],，它们能够使用定制的代码生成技术，根据高级查询描述生成紧凑高效的机器码。尽管基于编译器的流处理引擎已被证明能够实现最先进的单机性能，但我们发现，这些流处理引擎采用的查询语言限制严格，表达能力有限。据我们所知，先前的基于编译器的流处理引擎仅针对执行窗口聚合的查询进行设计。由于流查询通常需要使用更加复杂且多样的时间操作，当前基于编译器的流处理引擎支持现实世界流分析应用的能力受到显著限制。此外，这些流处理引擎采用的代码生成技术主要是作为源到源转换器设计的，并且严重依赖模板展开器。众所周知，这类编译器设计灵活性很差，维护起来也极其困难 [10, 43]。

基于这些观察，我们得出结论：当前的流处理引擎仅有限地利用了.....方面的优化与并行化机会流查询的优化与并行化机会有限，这是由若干固有的设计局限所致。这使得这类流处理引擎无法为许多现实世界中的流分析应用提供高性能流处理。本研究旨在实现硬件高效的流处理，同时不牺牲可编程性和通用性，以支持当今流分析工作负载多样化的计算需求。

[2]当步长与窗口大小相同时，翻转窗口是滑动窗口的一种特殊情况。

4 TiLT：一种以时间为中心的方法

要解决上述设计局限，并为多种流分析应用提供（i）有效的查询优化、（ii）并行化以及（iii）执行策略，就需要从根本上重新思考应如何设计流处理引擎（SPE）。在本文中，我们认为，有限的查询优化能力以及缺乏自动并行化支持，源于 SPE 所采用的事件中心时间查询表示模型。由于数据流被表示为事件序列，因此将时间算子定义为作用于事件的转换是很自然的。然而，我们观察到，这种以事件为中心的时间操作定义，无法充分表达有效查询优化与并行化所需的时间查询时间语义（详见第 5 节）。基于这一观察，我们从这一既有设计原则进行了根本转变，并提出了一种基于编译器的新型 SPE 设计，该设计遵循流查询的时间中心模型。

不同于传统的事件中心模型，时间中心模型采用更细粒度的时间操作表示方式：借助一种名为*TiLT*的新型中间表示（IR），将时间操作定义为在明确定义的时间域上执行的函数变换。TiLT 中间表示是一种高度表达性的函数式语言，并扩展支持时间操作，相较于传统查询表示模型具有多项优势。首先，TiLT 中间表示的函数式定义揭示了内在的数据并行性，使 TiLT 能够并行化任意流查询。其次，我们证明，细粒度的时间中心时间操作定义使 TiLT 能够通过简单的中间表示变换来支持有效的查询优化策略。最后，我们为 TiLT 构建了一个编译器后端，可以自动将时间中心的中间表示查询定义转换为硬件高效的可执行代码。

图3 展示了流式应用在 TiLT 中的生命周期。在第一阶段，TiLT 将用户编写的流查询转换为 TiLT 中间表示形式（第 4.1 节）。之后，编译器通过称为边界解析的步骤推断并行化查询执行所需的边界条件（第 5.1 节）。随后，TiLT 查询进入优化阶段（第 5.2 节）。最后，在代码生成步骤中，优化后的查询被降级为 LLVM IR，随后转换为用于并行查询执行的可执行代码（第6 节）。

4.1 TiLT 中间表示

与常规函数式语言类似，TiLT 支持整数、浮点数、数组、结构体、字典等数据类型，也支持算术/逻辑运算、条件运算、变量以及外部函数/库调用等表达式。在此基础上，我们引入了三种新构造，即 时序对象、

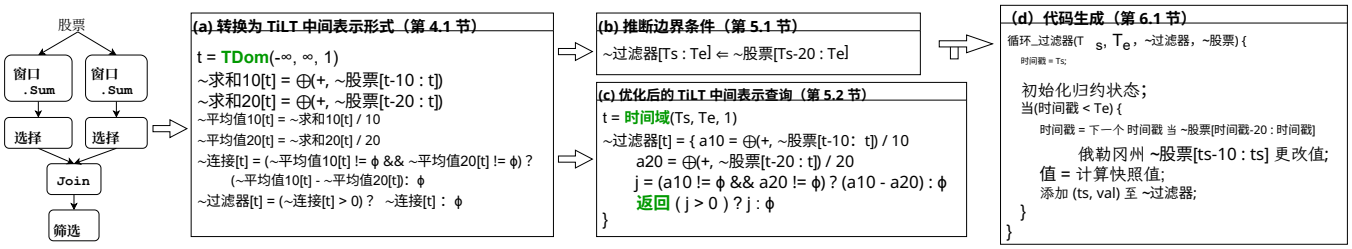


图 3：TiLT 编译流水线

归约函数和时间表达式，使 TiLT 能够定义多种时间操作。首先，时间对象类似于常规标量对象，不同之处在于它的值随时间变化，并覆盖一条无限长的时间线。传统方式将数据流表示为由离散事件组成的序列，每个事件包含时间区间和负载字段；与此不同，TiLT 使用单个时间对象来建模数据流，该对象根据当时生效的事件，在不同时间点取不同的值。其次，TiLT 使用归约函数将时间对象不断变化的值归约为单个标量值。在 TiLT 中引入归约函数，是为了对数据流执行聚合操作。最后，时间表达式是在定义明确的时间域上，对一个或多个时间对象进行的函数变换。时间表达式是 TiLT 中流式应用的基本构建模块。

时间对象： 我们使用 ~ 符号来区分时间对象和标量对象。例如，设 $\sim stock$ 为一个时间对象，它对应于股票价格事件的数据流 e_i，其中价格值为 ρ_i，有效区间为（t_i^s， t_i^e]。在任意时间点 T，都可以使用索引运算符（[]）检索 $\sim stock$ 的值，其定义如下。

$$\sim stock[T] = \begin{cases} \rho_i & \text{if} \quad \exists e_i \mid T \in (t_i^s, t_i^e] \\ \phi & \text{otherwise} \end{cases} \tag{1}$$

时间对象 $\sim stock$ 在时间 T 的值，是在 T 处处于活动状态的股票价格事件的负载值[3]。当时间 T 处没有任何活动事件时，时间对象取一个称为空值的空值 ϕ。值 ϕ 具有特殊属性：对其执行任何算术运算都会始终得到 ϕ。此外，TiLT 还允许通过向索引传入一个时间区间，从现有时间对象定义派生时态对象。例如，时间点 t_s 和 t_e 之间的股价值可以写成派生时态对象 $\sim win = \sim stock[t_s : t_e]$。随后，$\sim win$ 在任意时间点 T 的值定义如下。

$$\sim win[T] = \begin{cases} \sim stock[T] & \text{if} \quad T \in (t_s, t_e] \\ \phi & \text{otherwise} \end{cases} \tag{2}$$

归约函数： 归约函数记作 $\oplus(f, \sim I)$，是一种特殊表达式，用于根据归约操作 f 将时间对象 $\sim I$ 归约为标量值。TiLT 默认支持多种基本聚合操作，例如求和 (+)、乘积 (∗)、最大值 (>) 和最小值 (<)。例如，对时间对象进行归约 $\sim win$ 使用求和 (+) 可以写作 $\oplus(+, \sim win)$，其定义如下：

$$\oplus(+, \sim win) = \sum \{\sim win[t] \quad \forall t | \sim win[t] \neq \phi\} \tag{3}$$

平均值等其他聚合操作可以通过组合内置归约函数来实现。此外，TiLT 还允许用户定义自定义归约操作（详见第 6 节）。

时间表达式： TiLT 将流查询表示为一系列时间表达式，每个表达式都定义一个输出时间对象，该对象是在某个时间域上对一个或多个输入时间对象进行函数变换的结果。时间域 *TDom(start, end,precision)* 具有开始时间和结束时间，表示时间表达式所定义的区间；同时还具有时间精度，用于指明所得时间对象的值在该时间域内发生变化的频率。

$$\begin{aligned} t &= TDom(T_s, T_e, p) \\ \sim O[t] &= Fn(\sim I_1, \ldots, \sim I_n, C_1, \ldots, C_m) \end{aligned}$$

上方展示了时间表达式的一般语法。上述表达式定义了输出时间对象 $\sim O$，它是在时间域 t 上对 n 个输入时间对象 $\sim I_1$、……、$\sim I_n$ 和 m 个常量 C_1、……、C_m 进行函数变换（Fn）的结果。其中，t 定义在时间区间（T_s, T_e] 内，时间精度为 p。因此，在任意时刻 $T \in$（T_s, T_e] 且为 p 的倍数），$\sim O[T]$ 的值等于表达式 Fn（$\sim I_1$, I_2 … $\sim I_n$ T „）在时刻 的返回值。

4.2 TiLT 查询

图 4 展示了与图 1 中所述操作相对应的时间表达式。在上面的示例中，时间域 t 定义在 $-\infty$ 和 $+\infty$ 之间，精度为 1 秒。这意味着，使用 t 的时间表达式会在无限时间域内每隔一秒定义输出时间对象的一个值。第一个时间表达式等价于图 1a 中的选择操作。该表达式定义了一个函数变换，将时间对象 $\sim m$ 转换为 $\sim select$，作用于时间域 t；并且在任意时间点，$\sim select$ 的值定义为比同一时间点 $\sim m$ 的值大 1。类似地，第二个表达式等价于 Where 操作，只从 $\sim m$ 中过滤出偶数值。在此示例中，任意时间点的 $\sim where$ 值会根据条件选择：如果 $\sim m$ 在该时间点具有奇数值，则选择 ϕ。第三个表达式对应于时态连接操作，其结构与以下内容非常相似：

[3]为简单起见，在本节中，我们假设数据流中不存在区间重叠的事件，因此在任意给定时刻，至多只有一个事件处于活动状态。关于如何处理包含重叠事件的数据流，请参见第 6 节。

Where 操作，但它是由两个输入时间对象 $\sim m$ 和 $\sim n$导出的二元表达式。该表达式通过检查 $\sim m$ 和$\sim n$ 中的事件在给定时间是否都具有非空值，来识别严格重叠区间。如果是，则该表达式返回这些值的求和；否则返回 ϕ。最后，通过对从 $\sim m$ 中导出的每个 10 秒窗口应用 *Sum*归约函数，可以定义步长为 5 秒的 10 秒滑动窗口*Window-Sum*操作，该操作作用于时间域 $t\prime$，精度为 5。

```
t = TDom(-∞, ∞, 1)
~select[t] = ~m[t] + 1
~where[t] = (~m[t]%2 == 0) ? ~m[t] : ϕ
~join[t] = (~m[t] != ϕ && ~n[t] != ϕ) ?
      (~m[t] + ~n[t]) : ϕ

t" = TDom(-∞, ∞, 5)
~wsum[t"] = ⊕(+, ~m[t"-10 : t"])
```

图 4：选择、筛选、连接和窗口求和 操作的时间表达式

值得注意的是，尽管这些时间操作的传统定义在语义上看似不同，TiLT 对相同操作的定义在结构上却非常相似。这表明，TiLT 能够找到一组最小的抽象，用于表示范围广泛的时间操作。我们认为，将时间对象、归约函数和时间表达式等构造融入强大的函数式编程范式，使 TiLT 成为一种表达能力很强的查询表示模型，适用于现代流应用。

5 TiLT：优化与并行化

TiLT 中的编译流水线首先将流查询转换为 TiLT 中间表示（IR）形式，如图 3a所示。此处，示例查询通过将输入数据流定义为时间对象（$\sim stock$），并将每个操作映射到无限时间域上定义的相应时间表达式，从而降低为 TiLT 中间表示。转换后，TiLT 首先通过一个称为边界解析（第5.1节）的步骤，解析 TiLT 表达式的边界条件，从查询中提取数据并行性。然后，TiLT 通过简单的中间表示变换对查询执行若干优化遍。全面探索 TiLT 中间表示上的优化机会超出了本文的范围。相反，我们关注一种收益最高的优化：算子融合。流查询通常具有较高的数据局部性，因此可从利用这一特性的算子融合优化中显著获益，从而提高寄存器/缓存利用率 [19, 60]。在第5.2节中，我们将展示 TiLT 如何克服当前流处理引擎在跨流水线阻断算子执行算子融合时的局限性。

5.1 边界解析

TiLT 中时间查询以时间为中心的定义，精准捕捉了整个时间域内时序对象之间的数据依赖。例如，图 $\sim join$ 中时间 T 时的值仅依赖于图 4 中 $\sim m$ 和 $\sim n$ 在相同时间点的值。这意味着，$\sim join$ 在两个不同时间点 T_1 和 T_2 的值彼此独立，可以并行计算。这些数据依赖信息可以扩展到整个 TiLT 中间表示查询。例如，图 $\sim filter[T]$ 中3a中 $\sim stock$的数据依赖，可以通过沿血缘关系一直追溯到输入时序对象 $\sim filter$来确定。在此示例中，计算 T在 $\sim stock$处的值，仅依赖于时间区间$(T-10,\ T]$和$(T-20,\ T]$之间的数据。我们将其称为时序对象的时间血缘。TiLT 利用这些时间血缘信息，通过称为边界解析的步骤，从任意时间查询中提取数据并行性。

在边界解析步骤中，TiLT 通过推断时间域上的边界条件，将定义在无限时间域上的初始查询转换为有界域。例如，根据图 3a 中查询的时间血缘关系，$\sim filter$在任意区间 $(T_s, T_e]$ 上的值仅取决于 $(T_s-20, T_e]$ 中区间 $\sim stock$内的值。推断出时间边界条件后，TiLT 通过将 t设置为 $TDom(T_s, T_e, 1)$，重新定义时间查询的时间域为符号区间 $(T_s, T_e]$（图 3b）。TiLT 使用该边界条件对数据流进行分区，以便并行执行查询（详见第 6.2 节）。

5.2 算子融合

一旦查询以 TiLT 中间表示表达式的形式定义，通过简单的中间表示变换执行融合优化就十分直接。在 TiLT 查询中应用算子融合，只需将定义在同一时间域上的两个连续时间表达式合并为一个表达式即可。例如，以下是对时间表达式 $\sim avg10$、$\sim avg20$ 和 $\sim join$ 应用融合规则后的结果表达式。

```
~join[t] = { a10 = ~sum10[t]/10
             a20 = ~sum20[t]/20
             return (a10 != ϕ && a20 != ϕ) ?
       (a10 - a20) : ϕ }
```

融合由 $\sim avg10$、$\sim avg20$ 和 $\sim join$ 定义的表达式，只需替换每一处 $\sim avg10[t]$

并且 $\sim avg20[t]$ 在 $\sim join$ 使用 $\sim sum10[t]/10$ 以及 $\sim sum20[t]/20$ 如上所示。此转换等价于当前流处理引擎所支持的融合优化过程（如图2b所示）。然而，与传统流处理引擎不同的是，相同的 IR 转换可以应用于查询中的所有表达式，包括流水线阻断算子（例如 $\sim join$、$\sim sum10$ $\sim sum20$ ）。TiLT 会重复应用此转换，将趋势分析查询中的所有时间表达式融合为如下所示的单个表达式。

```
t = TDom(Ts, Te, 1)
~filter[t] = { s10 = ⊕( ~stock[t-10:t] )/10
               s20 = ⊕( ~stock[t-20:t] )/20
               j = (a10 != ϕ && a20 != ϕ) ? (a10-a20) : ϕ
               return (j != 0) ? j : ϕ }
```

与当前的流处理引擎相比，TiLT 得益于以下

两个关键原因，支持更加全面且可泛化的查询优化策略。首先，流式查询的图级表示通常粒度过粗。因此，在这一层级上往往无法表达原语操作的融合版本。另一方面，TiLT 提供了更加灵活的查询表示，支持如上所示的细粒度转换。其次，事件中心模型通常不会向优化器公开每个时态算子之后如何处理事件的区间。例如，连接的输出事件区间只有在运行时才能确定。相比之下，TiLT IR 表达式中的时间中心定义明确编码了跨时间域的转换，使 TiLT 能够执行更加复杂的优化。我们认为，TiLT IR 为时间查询带来了多种优化机会，而这些优化在传统查询表示模型上难以实现；我们计划在未来的工作中对其进行探索。

6 TiLT：编译与执行

尽管 TiLT 中间表示中的细粒度计算定义更适合进行有效的流查询优化，但它也带来了大量冗余。例如，与图 *4* 所示 Select 操作相对应的时间表达式，在时间域中每一秒都会为时间对象 $\sim select$ 定义一个值。尽管这种细粒度定义为中间表示操作提供了很大的灵活性，但它也会引入冗余计算，因为输入时间对象 $\sim m$ 的值不一定每秒都会发生变化。因此，直接将 TiLT 查询转换为可执行代码可能会非常低效。下面将介绍 TiLT 编译器如何消除这种冗余，并生成与 TiLT 中间表示查询相对应的硬件高效可执行代码。

6.1 代码生成

TiLT 编译器使用 C++ 和 LLVM JIT 编译器基础设施 [27]编写。在代码生成阶段，编译器将 TiLT 中间表示降级为 LLVM IR。由于 TiLT 中间表示本质上是一种函数式语言，因此适用于编译器中遵循的标准代码生成实践 [2, 35]。下面，我们将解释 TiLT 中三个新引入构造的代码生成策略。

6.1.1 时间对象。 根据方程中的形式化定义，时 1，时序对象在每个时间点都有一个值。然而，将相同的定义用于物理实现并不切实际。TiLT 转而使用一种称为快照缓冲区（SSBuf）的数据结构，仅存储时间对象值的变化。快照缓冲区是存储在数组中的有序快照序列，其中每个快照都存储发生变化时的时间戳（ts）和值（val）。

图 5 展示了一个以快照缓冲区形式存储的事件流示例。该缓冲区中的第一个快照取值为 null（ϕ），时间戳为 5，因为在此之前事件流中没有处于活动状态的事件。当第一个事件结束时（在 10），会添加第二个快照，其取值为该事件负载的值（a）。类似地，在数据流中每个后续事件开始和结束时，都会向缓冲区添加一个新的快照。当数据流包含有效区间重叠的事件时，单个快照可以

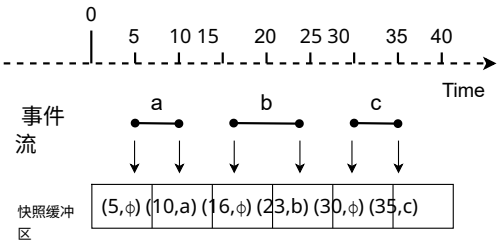


图 5：作为快照缓冲区的事件流

包含多个值。在这种情况下，TiLT 使用列表/映射来存储快照的值。

6.1.2 归约函数。 我们原生支持多种常见的归约函数，例如求和、乘积、最小值和最大值。此外，TiLT 还支持用户定义的归约函数。内置函数和用户定义函数均采用一种通用模板实现，与其他流处理引擎 [11, 47]所采用的模板类似。该模板包含四个 λ函数，用于在时间对象的每个快照到来时增量更新状态。(i) *Init* 函数返回归约操作的初始状态（例如，求和的初始值为 0）。(ii) *Acc* 函数将单个快照累积到状态中（例如，求和时进行加法运算）。(iii) *Result* 函数根据增量状态返回归约结果（例如，求和时返回该状态）。(iv) 对于可逆归约函数，可以提供可选的 *Deacc*函数，对状态应用聚合函数的逆运算（例如，求和时进行减法运算）。这一简单模板使 TiLT 能够支持高效的聚合实现，例如逐出时减法 [16]。

6.1.3 时间表达式。最后，将时间表达式综合为循环，这些循环遍历输入快照缓冲区并更新输出快照缓冲区。图 3d 展示了示例查询的综合循环。循环边界（T_s， T_e）以及循环计数器（ts）的递增量由时间域边界和精度确定。循环体执行时间表达式定义的计算。循环的一次迭代会计算输出缓冲区 $\sim filter$ 在时间戳 ts处的快照值（val）。

然而，如上所述，根据时间域精度（$ts = ts + 1$）来朴素地设置循环计数器的增量可能会非常低效，因为这会引入冗余迭代。相反，TiLT 利用了时间表达式函数定义中的一个不变量来避免冗余迭代，即只有当输入发生变化时，时间表达式的输出值才会发生变化。基于这一不变量，TiLT 编译器生成一个用于递增循环计数器的表达式，该表达式计算 ts的下一个值，在该值处，至少有一个 $\sim stock[ts-10:\ ts]$和 $\sim stock[ts-20:\ ts]$ 改变了外层快照。循环综合完成后，生成的循环会被封装在一个可调用函数中，并将符号化循环边界参数化为参数（图 3d ）。这样，TiLT 就可以在输出快照缓冲区上的任意时间区间内执行查询。

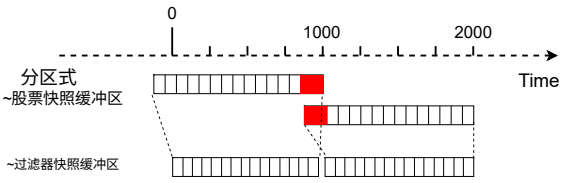


图 6：并行查询执行

6.2 查询执行

TiLT 将输入数据流划分为快照缓冲区，并使用相互独立的工作线程并行处理这些缓冲区，从而执行生成代码。数据流根据已解析的边界条件（图 3b）和用户定义的区间大小进行划分。例如，图 6 展示了图 3c 中示例查询的分区快照缓冲区，其区间大小为 1000 秒。在此示例中，计算 $\sim filter$区间内的输出快照，需要读取输入流中位于区间 $(0, 1000]$ 内的快照；同样，位于 $(-20,\ 1000]$之间的输出快照，需要处理位于 $(1000, 2000]$ 之间的输入快照，依此类推。尽管如上所述划分快照缓冲区会因重复快照而增加一些冗余（图 6中的阴影区域），但以这种方式提取数据并行性，可以使用无同步并行工作线程执行连续流查询。

7 评估

基准测试：为了评估 TiLT，我们准备了两组基准测试，以及一组具有代表性的实际流式应用。（i）时间操作：该基准测试包含图 1 所示的四种常用基本时序操作。（ii）雅虎流式基准测试（YSB）[12]：一种流行的流式基准测试，包含 选择、筛选和翻转窗口计数操作。（iii）实际应用：其中包括表 2 所示的八种真实世界流式分析应用。[4] 表 2 还介绍了评估中使用的公共数据集。有关基准测试查询的更多详情，请参见附录 A。**指标：**与以往工作一致 [11, 34, 47, 59]，，我们将数据处理吞吐量，即每秒处理的事件数，作为性能评估的主要比较指标。此外，我们还报告延迟受限吞吐量，以评估不同流处理引擎在较宽延迟范围内的性能。除非另有说明，性能数据均使用包含 160 百万个事件的数据集测得。所有报告的数据均为每项实验 5 次运行的平均测量值。所有测量值的标准差均低于 2%。

基线： 在时间操作基准测试中，我们将 TiLT 的吞吐量与四种最先进的纵向扩展流查询处理引擎（流处理引擎）进行比较：StreamBox [34], 微软 Trill [11], LightSaber [47], 和 Grizzly [14]。StreamBox 和 Microsoft Trill 都是基于解释执行的流处理引擎。StreamBox 使用 C++ 编写，并采用流水线并行来实现流查询的并行化。Trill 是一个使用 C# 编写的流处理引擎，旨在支持多样化的流分析应用。LightSaber 和 Grizzly 都是针对聚合操作进行优化的基于编译器的流处理引擎。

实验设置： 所有实验均在 AWS EC2 m5.8xlarge 上进行，该实例配备 32 个核心（支持超线程）、2.5 GHz 和 128 GB 动态随机存取存储器。对于可扩展性实验，我们还使用了 AWS EC2 m5zn.3xlarge，该实例配备 12 个核心、4.5 GHz 和 48 GB 动态随机存取存储器。为进行公平的性能比较，我们排除磁盘和网络访问所耗费的时间，仅在将整个输入数据集加载到内存后，测量查询执行的计算性能。

7.1 时间操作吞吐量

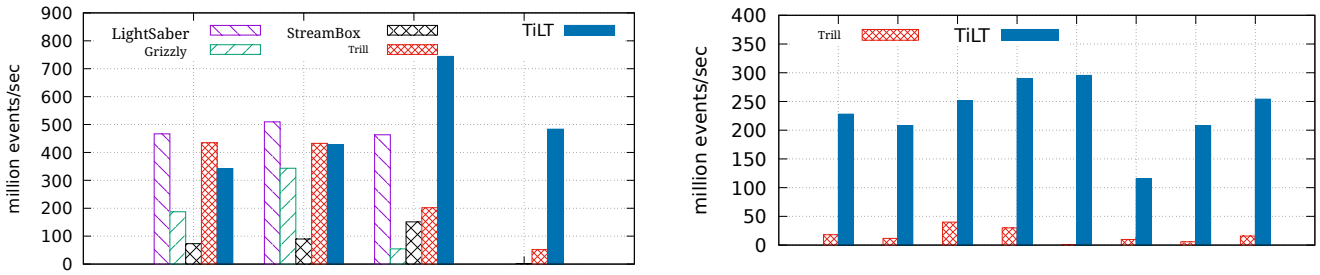
我们使用包含 选择、筛选、窗口求和和连接 的合成数据集来测量 TiLT 在时间操作上的性能，该数据集包含 1.6 亿个事件，并使用 16 个工作线程。图 7a 展示了 TiLT 与 StreamBox、Trill、Grizzly 和 LightSaber 的处理吞吐量比较。对于 选择 和 筛选等简单的逐事件操作，TiLT 的性能与其他流处理引擎相近（介于 0.69−1.44×）。对于 窗口求和等更复杂的操作，TiLT 的性能分别比 Trill 和 StreamBox 高出 6.64× 和 18.30×。这表明，TiLT 生成的代码能够显著优于基于解释执行的流处理引擎中的手写操作。此外，TiLT 的性能也分别比针对基于窗口的聚合进行优化的两个基于编译器的流处理引擎 Grizzly 和 LightSaber 高出 7.44× 和 1.87×。我们观察到，Grizzly 的高开销源于流处理引擎为执行并行聚合而采用的昂贵原子状态更新。另一方面，LightSaber 使用了复杂的数据结构，例如并行聚合树；我们观察到，对于现代流分析应用中常见的细粒度窗口聚合，这些结构的效率并不高。

最后，我们评估时间连接的性能。Grizzly 和 LightSaber 均不支持连接操作，因此我们仅将 TiLT 的性能与 StreamBox 和 Trill 进行比较。我们观察到，TiLT 的性能比 StreamBox 高出 321.94×，比 Trill 高出 13.87×。StreamBox 中的连接操作效率很低，因为它使用 $O(n^2)$ 算法来查找重叠事件。Trill 和 TiLT 都遵循事件的按序处理，因此只需进行 $O(n)$ 次比较即可执行连接。然而，Trill 的实现使用开销较高的并发哈希映射来维护算子状态，而时间中心模型使 TiLT 能够为连接生成更高效的无状态代码。这些结果表明，TiLT 能够为常用的时间操作生成高效代码，其性能可以显著优于基于解释器和基于编译器的流处理引擎。

7.2 可扩展性

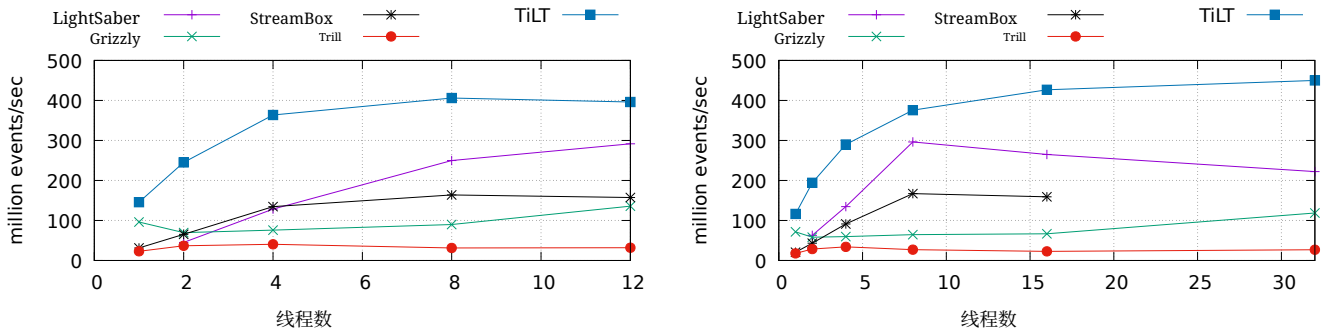
我们使用 YahooStreaming Benchmark 评估 TiLT 在多核心上扩展流查询的能力 [12]。我们通过增加工作线程数量，在一台 12 核机器和一台 32核机器上执行该查询，并将吞吐量与 Trill、Stream- Box、Grizzly 和 LightSaber 进行比较。图8a 和图8b 展示了

[4]我们基于这样一个认识来准备这些应用：用于评估流处理引擎的常见基准测试（如 YSB）只能代表现实世界中的流式分析用例中的一小部分。



（a）基本时序操作的吞吐量　　（b）真实世界流应用的吞吐量

图7：性能　　TiLT 在时间操作和真实世界流式应用基准测试上的比较　　ks



（a）12核机器上的吞吐量　　（b）32核机器上的吞吐量

图 8：Yahoo 流式基准测试（YSB）上的多核可扩展性

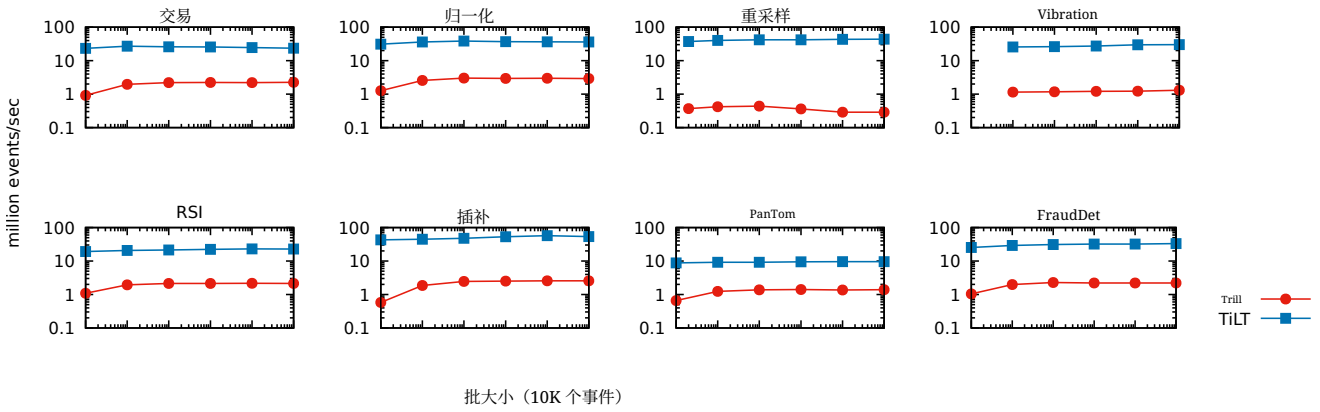


图 9：Trill 和 TiLT 的延迟受限吞吐量

分别在 12 核和 32核机器上测得的吞吐量。Trill 仅支持对分区数据流进行并行执行，因而表现出最差的可扩展性。我们还观察到有限的 Grizzly 的可扩展性。我们认为，这源于用于在线程之间进行同步的并发数据结构和原子状态。StreamBox 和 LightSaber 都可以扩展到 8 个并行线程，运行于

32核机器上。LightSaber 在多核环境下达到了 291*M* 事件/秒的峰值性能，并且在 12核机器和 32核机器上分别达到 296*M* 事件/秒。TiLT 始终优于所有流处理引擎，并在 12核机器上达到 406 百万事件/秒的峰值性能，在 32核机器上达到 450 百万事件/秒。TiLT 的卓越性能源于第 6.2 节所述的无同步数据并行查询执行策略。TiLT 在 12核机器上可实现至 4 个线程、在 32核机器上可实现至 8 个线程的近线性扩展。此后，可扩展性收益开始减小，因为查询执行正从受计算限制转变为受内存限制。这表明，TiLT 能够有效地实现查询执行的并行化，同时获得 1.52 – 13.20× 高于当前最先进流处理引擎的峰值性能。

7.3 真实世界应用的性能

我们评估 TiLT 在多大程度上能够满足真实流处理工作负载的性能要求，并将其与最先进的流处理引擎进行比较。为此，我们评估了 TiLT 在表 2所列八种真实世界流分析应用上的吞吐量。这些应用需要对数据流执行复杂的时间变换，因此编写相应的流查询需要一种表达能力很强的时间语言。在所有基线系统中，我们发现只有 Trill 提供的查询语言能够支持全部八种应用。因此，我们将 TiLT 在这些应用上的性能与 Trill 进行比较。

首先，我们测量了 Trill 和 TiLT 在这些应用上使用 16 个工作线程时的吞吐量。如图 7b所示，在所有应用上，TiLT 的性能都优于 Trill，领先 6.29 – 326.30×。这表明 TiLT 能够在多种流分析应用中提供更出色的性能。在信号重采样基准测试中，TiLT 获得了最大的加速，吞吐量比 Trill 高 326.30× 。该查询需要一种名为 Chop 的非标准时间操作，而我们发现 Trill 对该操作的算子实现效率不高。尽管需要执行这种非标准操作，TiLT 仍能生成高效的实现，最终显著提升了性能。

此外，我们还使用合成数据集，在实际应用中测量 TiLT 相较于 Trill 的延迟受限吞吐量。Trill 经过优化，可在较宽的延迟范围内提供高吞吐量。如图 9所示，我们通过设置批次/快照缓冲区大小，使其包含介于 10 与 1M 个事件。我们观察到，TiLT 在整个延迟范围内始终提供更高的吞吐量，而 Trill 则因查询执行开销较高，在较小批大小下出现 18 – 227× 性能下降。这表明 TiLT 提供的运行时环境开销极低，并能够在较宽的延迟范围内提供高性能。

7.4 查询优化的有效性

我们通过测量示例查询（图 3）在应用第 5.2 节所述的中间表示变换前后的单线程执行时间，分析 TiLT 中融合优化的效果。我们将结果与图 2a 和 2b 所示的 Trill 未优化和优化查询版本进行比较。在图 10 中，我们报告了这些查询版本各自的加速比

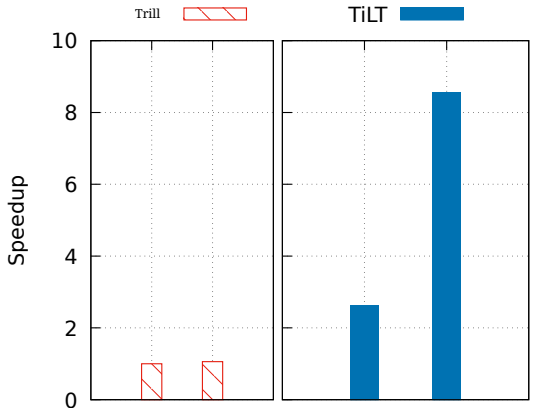


图 10：Trill 和 TiLT 中查询优化的性能细分（以 Trill 为基准归一化）

其结果以未优化 Trill 查询的吞吐量为基准进行了归一化。如图所示，在 Trill 中应用算子融合仅带来了 1.06× 的微小加速。这凸显了当前纵向扩展流处理引擎可用的优化机会有限。相比之下，在 TiLT 中，即使是未优化的查询版本，性能也优于优化后的 Trill 查询 2.61×。TiLT 中未应用任何优化（例如算子融合）的查询，采用了与解释型流处理引擎类似的查询执行模型。因此，此处观察到的加速主要归因于避免了 Trill 基于托管语言（C#）实现所带来的常见开销。这表明，TiLT 能够为单个算子生成高效代码，其性能优于解释型流处理引擎中的手写实现。此外，应用算子融合优化后，加速比还能进一步提升至 8.55× 。这一加速得益于最大限度地提高缓存利用率，并在查询执行过程中立即复用生成的中间结果。这项敏感性研究表明，TiLT 的性能优势既来自采用基于编译器的方法尽可能降低运行时开销，也来自利用以时间为中心的查询表示模型实现有效的查询优化。

8 相关工作

业界常用的许多流处理引擎（SPE）（例如 Apache Spark [59], Flink [8], Storm [52], Beam [4]）都是基于这样的假设设计为横向扩展系统的：单台机器无法满足现代流分析应用的性能需求。然而，近期关于纵向扩展流处理引擎的研究 [11, 14, 24, 33, 34, 47, 61] 表明，在单台多核机器上运行的设计良好的系统通常能够满足大规模流处理的性能需求 [30]。遗憾的是，我们观察到，许多先进的纵向扩展流处理引擎为了实现高单机性能，牺牲了语言的表达能力。在本文中，我们认为，要满足现代流分析

应用的需求，在可编程性与性能之间取得良好平衡非常重要。下面，我们将从这两个方面比较先进的纵向扩展流处理引擎。

MicrosoftTrill [11]针对多种流式分析应用进行了优化，并提供了一种表达能力很强的时态查询语言，支持丰富的时间操作和细粒度窗口划分。Trill 使用高效的算子实现和列式数据表示，以最大限度地提高缓存利用率；研究表明，其 10 – 100× 单机性能高于横向扩展型流处理引擎 [36]。然而，Trill 采用基于解释执行的查询执行模型，因此会产生显著的运行时开销 [19, 37, 60]。相比之下，TiLT 提供了类似的查询语言，但采用基于编译器的方法生成高效的硬件代码，其性能超过 Trill， 20× 平均值（第 7.3 节）。其他基于解释执行的流处理引擎，例如 StreamBox [7],StreamBox-HBM [33], BriskStream [61] ，旨在通过高效地并行化流查询，在多核机器上实现高性能。然而，这些流处理引擎仅提供低级 API 来编写流应用，对时态查询语言的支持有限。此外，我们表明，TiLT 中的无同步数据并行查询执行比这些系统具有更好的可扩展性（第 7.2 节）。

SABER [24], LightSaber [47], Scabbard [48], 和 Grizzly [14] 是近期基于编译器的流处理引擎中的典型代表。然而，正如我们在第 3 节所述，这些系统仅针对高效执行基于窗口的聚合进行了优化，并不支持时间连接等重要且常见的时间操作。因此，这些系统无法支持许多真实世界流式分析应用，例如表 2 中所示的应用。另一方面，TiLT 借助 TiLT 中间表示中表达能力很强的时间构造，支持范围广泛的流分析应用。我们还展示了，与这些流处理引擎相比，TiLT 能够生成更高效的代码并实现更优的性能（第 7 节）。此外，这些流处理引擎中的代码生成器被设计为模板展开器，因此更难维护和扩展 [43]。相比之下，TiLT 通过提出定义明确的中间表示，采用了更加系统化的方法；我们认为，这对于推动该领域的进一步研究十分重要。

9 结论

本文强调了当前流处理引擎（SPEs）的局限性，以及它们无法满足当今各类流分析应用的性能需求。为解决这些局限性，我们设计了 TiLT——一种面向流式应用的新型时间查询表示模型。TiLT 提供了丰富的编程接口，可支持广泛的流分析应用，同时实现高效的查询优化和并行化策略，而这些策略在传统 SPEs 上往往更难实现。我们还构建了一个编译器后端，可根据 TiLT 查询表示生成硬件高效代码。我们证明，在八个具有多样计算特征的真实世界流式分析应用上，TiLT 的性能最高可达到当前最先进 SPEs（例如 Trill）的 326×（20.49× 平均值）倍。TiLT 源代码可在 https://github.com/ampersand-projects/tilt.git 获取。

10 数据可用性声明

本文的制品已通过 Zenodo 发布 [20]。

致谢

我们首先感谢指导人和匿名审稿人提出的宝贵反馈与意见。我们还要感谢 EcoSystem 实验室的成员，尤其是 Kevin Song、JasperZhu、XinLi 和 ChristinaGiannoula，感谢他们为本文提供富有洞察力的意见和建设性反馈。本项目部分获得了加拿大创新基金会 JELF 基金、NSERC 发现基金、AWS 机器学习研究奖和 Facebook 教师研究奖的支持。

真实世界流应用

我们准备了一个基准测试套件，其中包含八个流分析应用，应用领域包括股票交易、信号处理、工业制造、金融机构和医疗保健。我们准备这些应用的依据是：用于评估流处理引擎（SPE）的常用基准测试查询，例如雅虎流式基准测试（YSB）[12]和 Nexmark [14] 仅代表了现实世界中的流式分析用例中较为狭窄的一部分。表 2简要介绍了基准测试套件中包含的流应用，以及评估所使用的相应公共数据集。下文将详细介绍这些应用。我们还以制品形式发布了这些查询在 Trill 和 TiLT 中的实现。

股票交易查询： 投资服务广泛使用流应用来分析股票市场趋势，以辅助购买决策。这些应用持续对高频股票价格数据流运行统计算法。在我们的基准测试套件中，我们纳入了两种常用的交易算法：(i) 基于趋势的交易；(ii) 基于相对强弱指数的交易。基于趋势的交易算法 [18] 会计算每只股票价格随时间变化的短期和长期移动平均线（例如，20分钟和50分钟），并在短期平均值高于长期平均值时识别上升趋势，反之则识别下降趋势。第二种交易算法使用相对强弱指数（RSI） [46] 作为动量指标，而不是使用移动平均线。RSI 是一种指标，根据股票或市场在 14 天交易周期内的收盘价，描绘其当前及历史强弱状况。这两种算法应用广泛，也常与更复杂的交易算法结合使用。

数据清洗/准备： 流处理应用处理的实时数据通常格式错误、已损坏或杂乱无章。因此，数据分析师通常需要在分析原始数据流之前进行数据清洗和预处理。例如，从不同来源收集的事件，其数值尺度往往差异很大。数据归一化是一种常用方法，可将不同尺度的事件值转换到名义上统一的尺度。我们纳入了一个基于标准分数的归一化查询，该查询会在每个 10 秒滚动窗口中计算事件载荷值的均值（μ）和标准差（σ）。窗口中每个事件的值通过计算（$X - \mu$）/σ进行归一化。

其次，医疗保健行业会使用信号处理操作来清理和准备从患者身上采集的生理信号，例如心电图和脑电图 [19]。这些信号以固定频率采集，通常为 10^{-4} Hz 至 10_3 Hz。我们在基准测试套件中纳入了两种常用的信号处理操作：（i）信号插补；（ii）信号重采样。信号插补操作用于填补信号流中缺失的事件。简单的插补方法包括用常数（例如零）或上一个有效事件的值替代缺失信号值。我们纳入了一个信号插补查询，用对应 10 秒滚动窗口中事件值的平均值替代缺失信号值。信号重采样操作用于将信号流从一种频率转换为另一种频率。我们使用线性插值[55]算法对信号流进行频率转换。

最后，PanTomkins算法 [39] 通常用于检测心电图信号中的 QRS 波群。QRS 波群代表心室除极，是心电图信号中可见的主要尖峰，可用于测量患者的心率。

制造业： 制造业中用于监测大型机械设备运行状况的工业传感器。其中一种应用场景是监测滚珠轴承的振动信号，以预测其故障率。这些传感器通常以非常高的频率生成数据流（频率最高可达 40 KHz） [23]，而标准的振动分析算法需要对这些数据流计算复杂的聚合函数。我们纳入了一个振动分析查询，用于在 100 毫秒滚动窗口上对数据流计算三个基于窗口的聚合函数，即峰度 [41], 、均方根 [56], 和峰值因子 [53] 。

金融机构： 金融和银行机构需要通过分析客户的实时金融交易数据来识别欺诈活动。一种基本的欺诈检测策略是针对异常交易金额制定规则。我们的基准测试套件中的欺诈检测查询，会在 10 天的滑动窗口内计算每位用户交易购买金额的移动平均值（μ）和标准差（σ），并将大额交易的阈值计算为 $\mu + 3 * \sigma$。最后，应用筛选操作，选择超过大额交易阈值的交易，并将其标记为潜在欺诈交易。

B 工件附录

B.1 摘要

我们提供了源代码和脚本，用于复现主论文中的可扩展性结果（第7.2节）以及实际应用性能（第 7.3节）。本附录介绍如何在合成数据集上生成类似于图8a、图 8b和图 7b 的图表。在合成数据集上测得的性能数据应能较为准确地估计真实数据集上的相应性能。

我们提供了 Docker 容器，用于为所有实验设置运行时环境，以支持可移植性。因此，该制品可在安装了 Docker 引擎的任何多核机器上运行。我们还使用 Linux 的 gnuplot 工具，根据收集到的性能数据生成图表。我们**建议**使用 Ubuntu 20.04 操作系统来运行制品中提供的脚本。

B.2 制品清单

• **算法：** 不适用。• **程序：** 表 2 中描述的基准测试。• **编译：** 以 Docker 化容器的形式提供。• **转换：** 无需转换工具。• 二进制文件： 包含源代码和脚本。• **数据集：** 随制品提供合成数据集。

• **运行时环境：** 提供用于创建运行时环境的 Docker 文件。

• **硬件：** 单个多核中央处理器。理想情况下，至少配备 16 个核心和 128 GB 内存。• **运行时状态：** 不受运行时状态影响。

• **执行：** 评估所有基准测试所需时间少于一小时。
• **指标：** 每秒处理的事件数（吞吐量）。
• **输出：** 与正文中的图8a、图8b和图7b相似的绘图。

• **实验：** 提供了 Bash 脚本和 Docker 文件，用于运行基准测试。结果中的数值差异可以忽略不计。• **大约需要多少磁盘空间？：** ~ 50 GB。

• **准备工作流大约需要多长时间？：** 不到1 小时即可设置好运行时环境。

• **完成实验大约需要多长时间？：** 不到30 分钟。

• **是否公开提供？：** 是• 30
代码许可（如公开提供）： LGPL- 。**数据许可（如公开提供）：** 不适用。
• **使用的工作流框架？：** 否。

• **存档链接：** https://doi.org/10.5281/zenodo.7493145

B.3 描述

B.3.1 如何访问。该制品可通过 GitHub 链接下载：https://github.com/ampersand-projects/streambench，也可通过 DOI 链接下载：https://doi.org/10.5281/zenodo.7493145 。

B.3.2 硬件依赖。TiLT 不需要任何特殊硬件。使用一台通用多核 CPU 即可运行该制品。我们建议使用至少配备 16 个核心和 128 GB 内存的机器。

B.3.3 软件依赖。该制品中的实验已配置为在 Docker 容器中运行。为复现结果，我们建议使用安装了 Docker 的 Ubuntu 20.04 机器。此外，我们使用 gnuplot 根据性能数据生成绘图/图表。

B.3.4 基准测试与基线。该制品中的实验使用雅虎流式基准测试（YSB） [12]进行第 7.2 节所述的可扩展性实验，并使用表 2 中展示的流应用来衡量第 7.3 节所述的实际应用性能。我们提供了用于构建和运行这些实验的脚本，涉及 Trill、 [11], StreamBox [34], Grizzly [14], LightSaber [47], 和 TiLT。我们对所有基准测试均使用查询处理吞吐量作为比较指标，即每秒处理的事件数量。

B.3.5 数据集。为方便起见，我们为所有实验提供了一个合成生成的数据集。在该数据集上产生的结果应与我们在论文正文中报告的

真实数据集上的结果具有可比性。

B.4 安装

我们提供 Docker 文件，为所有实验设置运行时环境。

(1) 按照以下说明安装 Docker：https://docs.docker.com/engine/install/ubuntu/。(2) 运行以下命令安装 gnuplot：`sudo apt-get install -y gnuplot`(3) 运行以下命令克隆 Git 仓库：`git c lonehttps://github.com/ampersand-projects/streambench.git--recursive`(4) 在克隆的仓库根目录下运行 setup.sh 脚本，以构建 Docker 镜像。

B.5 实验工作流

执行 run.sh 以运行所有实验并生成图表。

B.6 评估与预期结果

脚本执行完成后，应会生成类似于图 8a 和 7b 的图表。这些图表位于代码库的根目录下，文件名为 `ysb.pdf` 和 `e2e.pdf`。

首先，`ysb.pdf` 展示了 TiLT 与 Trill、StreamBox、Grizzly 和 LightSaber 在 Yahoo 流式基准测试（YSB）上的吞吐量比较，[12] 并行度范围为 1 到 16。与其他基线相比，TiLT 应始终实现更高的吞吐量和可扩展性。其次，`e2e.pdf` 展示了 TiLT 与 Trill 在表 2 所列真实世界流应用上的吞吐量比较，二者均采用固定的 8 线程并行度。平均而言，TiLT 应实现 ~ 10 – 100× 比 Trill 更高的吞吐量。

B.7 实验自定义

实际应用性能评估实验的并行度可通过在脚本 trill_bench/run.sh 和 tilt_bench/run.sh 中将 $THREADS 环境变量设为适当的线程数来调整。

B.8 方法论

投稿、评审与徽章评定方法：

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

参考文献

[1] Ajay Acharya 和 Nandini S. Sidnal。2016。基于复杂事件处理的高频交易。载于 *2016*年电气电子工程师学会第*23*届国际高性能计算会议研讨会 *(HiPCW)*。 39 42 10 1109 2016 014 – 。 [2] https://doi.org/ 。 /HiPCW. 。Alfred V. Aho、Monica S. Lam、Ravi Sethi 和 Jeffrey D. Ullman。。编译器：原理、技术与工具（第*2*版）。Addison-Wesley Longman Publishing Co., Inc.，美国。[3] Tyler Akidau、Alex Balikov、Kaya Bekiroglu、Slava Chernyak、Josh Haberman、Reuven Lax、Sam McVeety、Daniel Mills、Paul Nordstrom 和 Sam Whittle。2013。MillWheel：互联网规模的容错流处理。载于 超大型数据库。734–746。[4] Tyler Akidau、Robert Bradshaw、Craig Chambers、Slava Chernyak、Rafael J. Fernández-Moctezuma、Reuven Lax、Sam McVeety、Daniel Mills、Frances Perry, Eric Schmidt, and Sam Whittle. 2015.《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》。《VLDB基金会会刊》8 (2015), 1792–1803. [5] Iyad Aldasouqi and Jalal Atoum. 2011.《约旦河谷的流处理环境应用》。《国际计算机科学与安全期刊》(IJCSS) 5, 1 (2011), 1.[6] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion St oica, and Matei Zaharia. 2018.《结构化流处理：用于实时应用的声明式 API》。载于《2018年国际数据管理会议论文集》（美国德克萨斯州休斯敦）（SIGMOD '18）。美国计算机协会，纽约州纽约市，601–613. https://doi.org/10.1145/3183713.3190664[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. 2002.《数据流系统的模型与问题》。载于《第二十一届ACM SIGMOD-SIGACT-SIGART 数据库系统原理研讨会论文集》（威斯康星州麦迪逊）（PODS '02）。美国计算机协会，纽约州纽约市，1–16. https://doi.org/10.1145/543613.543615[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015.《Apache FlinkTM：在单一引擎中进行流处理与批处理》。《IEEE数据工程简报》38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf [9] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016.《Cutty：用户定义窗口的聚合共享》。载于《第25届ACM国际信息与知识管理会议论文集》（美国印第安纳州印第安纳波利斯）（CIKM '16）。美国计算机协会，纽约州纽约市， 1201 1210 10 1145 2983323 2983807 – 。https://doi.org/ . / . Donald D. [10] Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. 1981.《System R 的历史与评估》。《ACM通讯》24, 10 (1981年10月), 632–646. https://doi.org/ 10 1145 358769 358784 . / . Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert De-Line, Danyel Fisher, John Platt, James Terwilliger, John Wernsing, and Robert DeLIne. 2015.《Trill：用于多样化分析的高性能增量查询处理器》。《VLDB—超大型数据库》。 [11] h t t p s:// w w w. m i c r o s o f t. c o m/ en-us/research/publication/trill-a-high-perfor mance-incremental-query-processor-for-diverse-analytics/[12] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, and Paul Poulosky. 2016.《流式计算引擎基准测试：Storm、Flink 和 Spark Streaming》。载于《2016年IEEE国际并行与分布式处理研讨会》（IPDPSW）。1789–1792. https://doi.org/10.1109/IPDPSW.2016.138[13] G. Graefe and W.J. McKenna. 1993.《Volcano 优化器生成器：可扩展性与高效搜索》。载于《IEEE第九届国际数据工程会议论文集》。 209 218 10 1109 1993 344061 – 。https://doi.org/ . /ICDE. . Philipp M. [14] Grulich, Breß Sebastian, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, and Volker Markl. 2020.《Grizzly：通过自适应查询编译实现高效流处理》。载于《2020年ACM SIGMOD国际数据管理会议论文集》（美国俄勒冈州波特兰）（SIGMOD '20）。美国计算机协会，纽约州纽约市， 2487 2503 10 1145 3318464 3389739 – 。https://doi.org/ . / . Joel Hasbrouck and [15] Gideon Saar. .《低延迟交易》。《金融市场期刊》16, 4 (2013), 646–679. https://doi.org/10.1016/j.finmar.2013.05.003 高频交易。[16] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. 2017.《滑动窗口聚合算法：教程》。载于《第11届ACM国际分布式与事件驱动系统会议论文集》（西班牙巴塞罗那）（DEBS '17）。美国计算机协会，纽约州纽约市，11–14. https://doi.org/ 10 1145 3093742 3095107 . / . Natalie Huang, Huan; Baddour. .《时变转速条件下的轴承振动数据》。 [17] https://doi.org/ 2019 10 17632 43 2 . /v hmbwxpm. [18] Investopedia. n. d. .《算法交易基础：概念与示例》。 h t t p s:// w w w. investopedia.com/articles/active-trading/101014/basics-algorithmic-trading-concepts-and-examples.asp [19] Anand Jayarajan, Kimberly Hau, Andrew Goodwin, and Gennady Pekhimenko. 2021.《LifeStream：用于周期流的高性能流处理引擎》。载于《第26届ACM国际编程语言体系结构支持会议论文集》（虚拟会议，美国）（ASPLOS 2021）。美国计算机协会，纽约州纽约市，107–122. https://doi.org/ 10 1145 3445814 3446725 . / . Anand Jayarajan, Wei Zhao, and Yudi Sun. 2022.《TiLT：以时间为中心的流查询优化与并行化方法》。(2022). https://doi.org/10.5281/zenodo.7493145[21] Alistair E.W. Johnson, Tom J. Pollard, Lu Shen, Li-wei H. Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G. Mark. 2016.《MIMIC-III：一个可免费访问的重症监护数据库》。 [20]

Scientific Data 3, 1 24 2016 160035 10 1038 2016 35 (May), . [22] [ ] https://doi.org/ . /sdata. . Kaggle. n. d. .《信用卡欺诈》。https://www.kaggle.com/code/samkirkiles/credit-card-fraud/data[23] A Khadersab and S Shivakumar. 2018.《旋转机械的振动分析技术及其对轴承故障的影响》。《Procedia Manufacturing》20 (2018), 247–252. https://doi.org/10.1016/j.promfg.2018.02.036 第二届材料、制造与设计工程国际会议（iCMMD2017），2017年12月11日至12日，MIT Aurangabad，马哈拉施特拉邦，印度。[24] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016.《SABER：面向异构架构的基于窗口的混合流处理》。载于《2016年国际数据管理会议论文集》（美国加利福尼亚州旧金山）（SIGMOD '16）。美国计算机协会，纽约州纽约市， 555 569 10 1145 2882903 2882906 – 。https://doi.org/ . / . Lingkun Kong [25] and Konstantinos Mamouras. .《StreamQL：用于处理流式时间序列的查询语言》。《ACM程序设计语言汇刊》4, OOPSLA, Article 183 (2020年11月), 32页。https://doi.org/10.1145/3428251[26] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015.《Twitter Heron：大规模流处理》。载于《2015年ACM SIGMOD国际数据管理会议论文集》（澳大利亚维多利亚州墨尔本）（SIGMOD '15）。美国计算机协会，纽约州纽约市， 239 250 10 1145 2723372 2742788 – 。https://doi.org/ . / . Chris [27] Lattner and Vikram Adve. .《LLVM：用于全生命周期程序分析与转换的编译框架》。载于《国际代码生成与优化研讨会：反馈导向与运行时优化论文集》（加利福尼亚州帕洛阿尔托）（CGO '04）。IEEE计算机学会，美国，75. [28] John W. Lockwood, Adwait Gupte, Nishit Mehta, Michaela Blott, Tom English, and Kees Vissers. 2012.《用于高频交易（HFT）的FPGA硬件低延迟库》。载于《2012年IEEE第20届年度高性能互连研讨会》。 9 16 10 1109 2012 15 – 。https://doi.org/ . / [29] HOTI. . Raymond A. Lorie. .《XRM——扩展型（N元）关系存储器》。研究报告 / G / IBM / 剑桥科学中心 G320-2096 (1974). [30] LSDS. [n. d.].《我们需要分布式流处理吗？》https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing [31] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. 2014.《Stream Bench：面向现代分布式流计算框架的基准测试》。载于《2014年第七届IEEE/ACM公用计算与云计算国际会议论文集》（UCC '14）。IEEE计算机学会，美国，69–78. https://doi.org/10.1109/UCC.2014.15[32] Frank McSherry, Michael Isard, and Derek G. Murray. 2015.《可扩展性！但代价是什么？》。载于《第15届操作系统热点问题研讨会》（HotOS XV）。USENIX协会，瑞士伊廷根修道院。https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry [33] Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2019.《StreamBox-HBM：基于高带宽混合内存的流分析》。载于《第二十四届体系结构支持编程语言和操作系统国际会议论文集》（美国罗得岛州普罗维登斯）（ASPLOS '19）。美国计算机协会，纽约州纽约市， 167 181 10 1145 3297858 3304031 – 。https://doi.org/ . / . [34] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017.《StreamBox：多核机器上的现代流处理》。载于《2017年USENIX年度技术会议》（USENIX ATC 17）。USENIX协会，加利福尼亚州圣克拉拉，617–629. https://www.usenix.org/conference/atc17/technical-sessions/presentation/miao [35] Steven S. Muchnick. 1998.《高级编译器设计与实现》。Morgan Kaufmann出版社，美国加利福尼亚州旧金山。[36] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013.《Naiad：及时数据流系统》。载于《第二十四届ACM操作系统原理研讨会论文集》（宾夕法尼亚州法明顿）（SOSP '13）。美国计算机协会，纽约州纽约市， 439 455 10 1145 2517349 2522738 – 。[37] https://doi.org/ . / . Thomas Neumann. .《为现代硬件高效编译高效查询计划》。《VLDB基金会会刊》4, 9 (2011年6月), 539–550. https://doi.org/10.14778/2002938.2002940[38] NYSE. [n. d.].纽约证券交易所。ftp://ftp.nyse.com/ [39] Jiapu Pan and Willis J. Tompkins. 1985.《实时QRS检测算法》。《IEEE生物医学工程汇刊》BME-32, 3 (1985), 230–236. https://doi.org/10.1109/TBME.1985.325532[40] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. 2018.《TerseCades：流处理中的高效数据压缩》。载于《2018年USENIX年度技术会议》（USENIX ATC 18）。USENIX协会，波士顿，MA，307–320. https://www.usenix.org/conference/atc18/presentation/pekhimenko [41] Robert B. Randall and Jérôme Antoni. 2011.《滚动轴承故障诊断——教程》。《机械系统与信号处理》25, 2 (2011), 485–520. https://doi.org/10.1016/j.ymssp.2010.07.017

[42] Md. Mamunur Rashid、Iqbal Gondal 和 Joarder Kamruzzaman。2015年。通过从机器振动数据中挖掘故障频率进行状态监测。载于2015年国际神经网络联合会议（IJCNN）。1–8。https: //doi.org/10.1109/IJCNN.2015.7280569[43] Amir Shaikhha、Yannis Klonatos、Lionel Parreaux、Lewis Brown、Mohammad Dashti 和 Christoph Koch。2016年。如何构建查询编译器。载于《2016年国际数据管理会议论文集》（美国加利福尼亚州旧金山）（SIGMOD '16）。美国计算机协会，美国纽约州纽约市， 1907 1922 10 1145 2882903 2915244 –。https://doi.org/ 。 / 。[44] Sasha Stoikov 和 Rolf Waeber。 。使用低延迟交易算法降低交易成本。《定量金融》16，9（2016），1445–1451。https://doi.org/10.1080/14697688.2016.1151926 arXiv:https://doi.org/ 10 1080 14697688 2016 1151926 。 / 。 。Michael Stonebraker、Ugur [45] Çetintemel 和 Stanley B. Zdonik。2005年。实时流处理的8项要求。《SIGMOD记录》34，4（2005），42–47。https://doi.org/ 10 1145 1107499 1107504 。 / 。Adrian [46] Ţăran-Moroşan。 。重新审视相对强弱指数。《非洲商业管理杂志》5，14（2011），5855–5862。[47] Georgios Theodorakis、Alexandros Koliousis、Peter Pietzuch 和 Holger Pirk。2020年。LightSaber：多核处理器上的高效窗口聚合。载于《2020年ACM SIGMOD国际数据管理会议论文集》（美国俄勒冈州波特兰）（SIGMOD '20）。美国计算机协会，美国纽约州纽约市， 2505 2521 10 1145 3318464 3389753 –。https://doi.org/ 。 / 。[48] Georgios Theodorakis、Fotios Kounelis、Peter Pietzuch 和 Holger Pirk。2021年。Scabbard：单节点容错流处理。《VLDB基金会会刊》15，2，2021，361–374，10，14778，3489496（10月），–。https://doi.org/ 。 / 。 3489515 [49] Ankit Toshniwal、Siddarth Taneja、Amit Shukla、Karthik Ramasamy、Jignesh M. Patel、Sanjeev Kulkarni、Jason Jackson、Krishna Gade、Maosong Fu、Jake Donham、Nikunj Bhagat、Sailesh Mittal 和 Dmitriy Ryaboy。2014年。Storm@twitter。载于《2014年ACM SIGMOD国际数据管理会议论文集》（美国犹他州雪鸟镇）（SIGMOD '14）。美国计算机协会，美国纽约州纽约市，147–156。https://doi.org/10.1145/2588555.2595641[50] Jonas Traub、Philipp Marian Grulich、Alejandro Rodriguez Cuellar、Sebastian Bress、Asterios Katsifodimos、Tilmann Rabl 和 Volker Markl。2018年。Scotty：面向乱序流处理的高效窗口聚合。载于2018年IEEE第34届国际数据工程会议（ICDE）。1300–1303。https: //doi.org/10.1109/ICDE.2018.00135[51] Deepak Vasisht、Zerina Kapetanovic、Jongho Won、Xinxin Jin、Ranveer Chandra、Sudipta Sinha、Ashish Kapoor、Madhusudhan Sudarshan 和 Sean Stratman。2017年。FarmBeats：面向数据驱动型农业的物联网平台。载于第14届USENIX网络系统设计与实现研讨会（NSDI 17）。USENIX协会，美国马萨诸塞州波士顿，515–529。https://www.usenix.org/conference/nsdi17/ technical-sessions/presentation/vasisht[52] Jan Sipke van der Veen、Bram van der Waaij、Elena Lazovik、Wilco Wijbrandi 和 Robert J. Meijer。2015年。动态扩展Apache Storm以分析流数据。载于《2015年IEEE首届大数据计算服务与应用国际会议论文集》（BIGDATASERVICE '15）。IEEE计算机学会，美国， 154 161 10 1109 2015 56 –。[53] https://doi.org/ 。 /BigDataService。 。维基百科。 。峰值因子——《自由百科全书》维基百科。http://en. wikipedia.org/w/index.php?title=Crest%20factor&oldid=1106197809。 [在线；访问日期：27-January-2023]。[54] 维基百科。2023年。插补（统计学）——《自由百科全书》维基百科。http://en.wikipedia.org/w/index.php?title=Imputation%20(statistics)&oldid=1118637345 [27 2023] 。在线；访问日期：-January- 。[55] 维基百科。2023年。线性插值——《自由百科全书》维基百科。http://en.wikipedia.org/w/index.php?title=Linear%20interpolation&oldid=1105095045 [27 2023] 。在线；访问日期：-January- 。[56] 维基百科。2023年。均方根——《自由百科全书》维基百科。http://en.wikipedia.org/w/index.php?title=Root%20mean%20square&oldid=1127312985 [27 2023] 。在线；访问日期：-January- 。[57] 维基百科。2023年。标准分数——《自由百科全书》维基百科。http://en. wikipedia.org/w/index.php?title=Standard%20score&oldid=1128321544。 [在线；访问日期：27-January-2023]。[58] wso2。 [不适用。]欺诈检测与预防：一种数据分析方法。https://wso2.com/whitepapers/fraud-detection-and-prevention-a-data-analytics-approach/#09[59] Matei Zaharia、Tathagata Das、Haoyuan Li、Timothy Hunter、Scott Shenker 和 Ion Stoica。2013年。离散化流：大规模容错流计算。载于《第24届ACM操作系统原理研讨会论文集》（宾夕法尼亚州法明顿）（SOSP '13）。美国计算机协会，美国纽约州纽约市，423–438。https://doi.org/10.1145/2517349.2522737[60] Steffen Zeuch、Bonaventura Del Monte、Jeyhun Karimov、Clemens Lutz、Manuel Renz、Jonas Traub、Sebastian Breß、Tilmann Rabl 和 Volker Markl。2019年。在现代硬件上分析高效流处理。《VLDB基金会会刊》12，5，2019，516–530，10，14778，3303753（1月），–。https://doi.org/ 。 / 。 3303753 [61] Shuhao Zhang、Jiong He、Amelie Chi Zhou 和 Bingsheng He。2019年。BriskStream：在共享内存多核架构上扩展数据流处理。载于

*2019*年国际数据管理会议论文集（荷兰阿姆斯特丹）（*SIGMOD '19*）。美国计算机协会，美国纽约州纽约市，705–722。https://doi.org/10.1145/3299869.3300067

收稿日期：2022年7月7日；录用日期：2022年9月22日