FluxSieve：统一流式处理与分析数据平面，实现可扩展的云可观测性

Adriano Vogel Dynatrace Research 林茨，奥地利 adriano.vogel@dynatrace.com

Sören Henning Dynatrace Research 林茨，奥地利 soeren.henning@dynatrace.com

Otmar Ertl Dynatrace Research 林茨，奥地利 otmar.ertl@dynatrace.com

arXiv:2603.04937v1 [cs.DB] 5 Mar 2026

摘要

尽管查询优化、索引技术和数据存储领域取得了诸多进展，现代数据平台在高并发和计算密集型查询的情况下，仍难以提供稳定可靠的查询性能。这一挑战在处理高容量、高速率数据记录的大规模可观测性平台中尤为突出。例如，在查询时反复执行代价高昂的过滤查询，会给分析数据平面带来巨大的计算开销和存储开销。

本文提出 FluxSieve，一种统一架构，通过将轻量级流内预计算与过滤层直接嵌入数据摄取路径，使传统的拉取式查询处理与推式流处理相互协调。这避免了在专用流处理框架中运行查询所带来的复杂性与运维负担。具体而言，本工作（i）提出一种基础架构，通过流内过滤与记录丰富来统一流式数据平面与分析数据平面；（ii）设计一种可扩展的多模式匹配机制，支持过滤规则的并发评估与即时更新，且每条记录的开销极低；（iii）展示如何将这种摄取时处理与两个开源分析系统集成——以 Apache Pinot 作为实时在线分析处理（RTOLAP）引擎，以 DuckDB 作为嵌入式分析型数据库；（iv）对我们的方法进行全面的实验评估。

我们在不同系统、查询类型和性能指标上的评估表明，查询性能可获得高达数量级的提升，而额外存储可忽略不计，计算开销也极低。这些结果表明，将稳定且高成本的过滤操作从分析平面转移到流式数据平面，是大规模数据分析的一种有效设计模式，能够实现简单而高效的架构。所提出的方法尤其与从事工业级分析平台的从业者，以及数据库和流处理系统设计者密切相关。

1 引言

大规模数据处理系统（包括传统数据库、数据仓库和现代数据湖仓）的查询性能优化，是数据管理中的一项根本挑战。数十年来，人们在查询计划优化、索引策略、分区方案和物化视图方面的努力，显著提升了性能 [29]。然而，对于系统日志、应用程序追踪和事件流等海量半结构化及非结构化数据，要始终保持低延迟仍然困难重重。难点源于多个因素：需要扫描或筛选的数据规模、分析查询的复杂性、存储效率与查询速度之间的权衡，以及现代架构的分布式特性。

即使列式存储格式、索引、自适应查询执行和基于成本的优化器不断发展，日志数据仍因其极高的摄取速率而格外难以处理。要在 PB 级日志数据集上持续保持亚秒级查询响应时间，仍不断将系统设计推向极限，也促使人们探索更高效的数据处理新方法 [1]。

传统数据库和数据处理架构通常采用以自身为中心的拉取式查询模型。用户和应用程序反复轮询集中式存储库，按需获取当前状态，并假设数据相对静态且查询延迟可以接受。尽管这种方法已被证明适用于许多事务型和分析型工作负载，但它并不适合实时分析、事件驱动架构等现代高速度环境。拉取式查询需要在每次请求时遍历索引并执行查询计划，消耗计算资源，而且往往会重新读取自上次轮询以来未发生变化的数据。此外，这种方式要求数据库承担所有查询负载，使数据库与客户端紧密耦合，从而限制可扩展性并增加延迟。因此，拉取式系统存在重复轮询、资源消耗增加以及洞察延迟等问题。

流处理通过引入基于推送的查询来解决这些局限性，从而实现实时响应并消除重复轮询。在这种模式下，数据持续流经处理流水线，结果会立即推送给消费者。然而，纯流处理也带来了自身的挑战。首先，设计和运维流处理系统通常需要复杂状态管理，并谨慎处理容错和可扩展性问题。其次，许多流处理引擎缺乏易于访问的持久状态，这使即席分析和历史数据访问更加困难，限制了它们对结合流式处理与批处理的工作负载的适用性 [6]。第三，传统流处理框架（如 Apache Flink、Apache Spark 结构化流和 Kafka Streams）通常需要投入大量工程工作，才能开发、部署和维护生产系统。最后，运行独立的流处理器可能会增加基础设施成本和运维开销 [22]。

为平衡这些权衡，近期的研究和工业系统探索了混合架构，将流式处理的推送机制与传统的数据库拉取查询相结合 [5]。这些架构旨在兼顾流式处理的即时性与按需查询的灵活性，同时支持连续型和交互式工作负载 [8]。这种混合模式在大规模可观测性系统等领域尤其具有吸引力，因为这些领域需要实时洞察与历史分析并存。

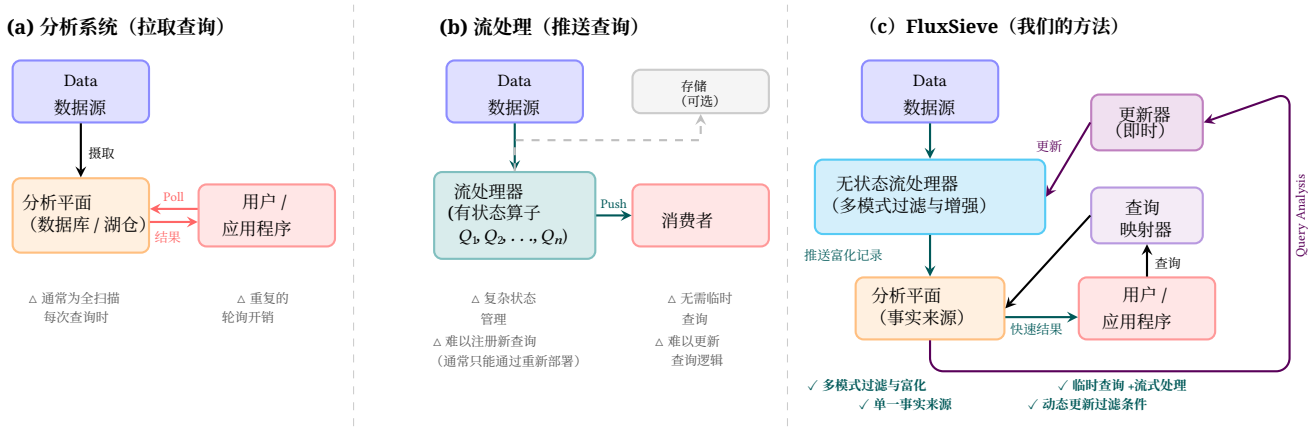


图 1：查询处理范式：(a) 传统拉取式查询；(b) 流处理；(c) FluxSieve 通过执行流内多模式过滤与增强实现统一，同时保留分析平面作为事实来源。

本研究通过一种高效的流内预计算与过滤机制，推进了这一研究方向。该机制直接作用于数据摄取管道，从而统一流处理与数据库范式。我们提出的 FluxSieve 将轻量级、无状态的流处理能力嵌入数据源与分析处理架构之间的数据路径中。因此，记录在到达分析存储层之前即可在流内进行过滤和丰富，并以优化模式持久化数据，以提升数据检索、剪枝及下游查询性能。图 1 概述了我们的方法与其他范式的对比。

我们方法的一个关键要素是高效的多模式匹配，它能够针对输入记录评估多个过滤条件。通过在摄取时执行模式匹配和增强，我们将计算工作前移，从分析平面中移除了开销高昂的过滤操作。这种流内过滤与增强具有双重作用：它降低了存储和索引维护成本，并预先计算派生过滤属性，否则这些属性需要在查询时进行开销高昂的计算。因此，分析平面中的查询可以在经过精炼和增强、针对分析访问模式进行优化的数据集上运行，从而显著提升性能和整体系统效率。我们的统一架构保留了流式思维的优势，同时避免了传统流处理框架的复杂性和开销，将开销高昂且频繁执行的过滤查询移出数据库。[1] 本文的贡献如下：

- 我们提出 FluxSieve，作为一种基础架构，用于统一分析型与流式数据平面。
- 我们为流处理系统设计了一种基于多模式匹配的高效流内过滤方法。
- 我们提出了一种在流处理器中即时更新过滤条件的方法。
- 我们展示了如何将 FluxSieve 与两个开源分析处理系统集成：ApachePinot [11] 作为实时在线分析处理（RTOLAP）系统，以及 DuckDB [19]作为嵌入式分析型数据库。
- 我们对该方法在多种系统、查询和指标上的表现进行了全面的实验评估。

本文结构如下。第 2 节介绍背景与研究动机。第 3 节详细阐述我们的方法。第 4 节讨论评估方法，第 5 节和第 6 节展示实验结果。第 7 节概述最新研究进展。最后，第 8 节总结全文。

2 背景与动机

本节介绍我们的动机用例（第 2.1 节），并在第 2.2 节进一步阐述现有挑战。

2.1 真实世界的用例与动机

现代云原生系统会产生异构的数据流，包括可观测性信号 **日志**、分布式**追踪**（由跨度组成）、高分辨率**指标**、执行**剖析数据**以及**事件** [9]。这些数据为两类主要的分析查询提供支持：

（1）周期性仪表板与警报。运维人员持续重新计算服务级指标，例如请求延迟百分位数、错误率时间序列和资源利用率。（2）定期取证分析与容量分析。事件后的调查、威胁狩猎、回归问题分类或长期容量规划，往往需要对数周的原始遥测数据进行全扫描、对自由文本消息进行复杂模式匹配，或对日志、追踪和指标进行多属性连接。这些查询计算量大且对时效性要求高，但也具有很强的重复性，例如在不同事件中定期执行类似的过滤器。

本研究聚焦于**日志**工作负载，尤其是筛选率极高的工作负载，即结果所含记录远少于输入数据的情况。这些场景最能凸显穷举扫描的成本，因此也最能说明改进的必要性。尽管我们主要关注日志，但所提出的流内过滤与增强机制具有通用性：它可以预先计算开销高、

[1]本文重点探讨如何将过滤移至流式数据平面，不过，也可以在流内预先计算适当的聚合，以减轻分析查询的负载。

重复性强的过滤器，适用于多种可观测性数据类型，包括日志、跨度和指标记录。

高严重性事件往往需要“海底捞针”式查询：在数百万条日志行中，只有极少数记录能够匹配。这类工作负载极高的选择率和紧迫性，暴露了拉取查询分析和朴素流式过滤器的性能上限。在秒到分钟的运行时间范围内执行此类“海底捞针”式查询颇具挑战，因为分析引擎必须扫描数十亿甚至数万亿字节的半结构化文本，并评估数千个表达式。此外，许多可观测性查询会使用固定过滤器，定期在不同时间窗口内执行。

2.2 现状与挑战

生产环境中的探索性日志分析仍主要依赖拉取查询：分析人员或自动化仪表板发起查询，回溯性地在庞大的不可变日志语料库上执行。这种方式没有考虑到新记录（例如日志）持续且快速地到达，而这些记录通常与查询服务相关。此外，尽管最先进的系统通过倒排索引、范围索引和全文索引缓解了暴力扫描的问题，但仍存在若干瓶颈。

2.2.1 拉取式查询分析的局限性。

(a) **扫描开销**。即使使用现代全文搜索（FTS）索引，高选择性的查询通常也需要读取并解压每个可能包含匹配项的候选数据段。因此产生的 I/O 和 CPU 周期会主导查询执行时间，限制整体查询性能和系统可扩展性。(b) **组合谓词爆炸**。运维手册经常会使用数百或数千个字符串匹配、正则表达式或模糊匹配。在传统索引结构中编码这类复合条件，要么不可行（会导致索引膨胀），要么需要进行后置筛选，从而抵消索引带来的性能优势。(c) **全文搜索索引的局限性**。全文搜索索引是日志字符串搜索的先进技术；然而，对于海量数据中的高选择性过滤查询场景，它们存在固有局限。无论是在构建还是查询执行期间，这类索引都会消耗大量 CPU，带来显著的存储开销，而且随着数据增长至 TB 级，查询性能也会下降。

2.2.2 流处理的优势及其缺陷。一种自然的应对策略是将复杂筛选前移至流处理器，在摄取时评估记录谓词。尽管这一做法在概念上颇具吸引力，但也带来了自身的挑战：

- **运维碎片化**。与采用统一的分析技术栈相比，维护一个独立的、有状态的流处理集群（例如 Flink、Kafka Streams）会增加部署、监控和故障处理的复杂性 [23]。
- **性能与效率**。在流处理中，高效的模式匹配和过滤仍是一项重大挑战 [17]。随着规则集不断扩展，CPU 和内存需求会非线性增长，因此需要大幅增加资源配置，推高运维成本。在典型的流处理架构中，例如 ShuffleBench [9], 所展示的架构，聚合是在独立的有状态流算子（处理器）中执行的。除了造成运维碎片化，这种分离还增加了针对新查询更新聚合逻辑的开销，尤其是与在分析平面上执行的传统拉取式查询相比更是如此。此外，尽管容错能力近期取得了进展 [23], , 这些机制仍会带来性能损失。
- **规则集敏捷性**。更新数百条过滤表达式通常需要更改代码、重新编译和重新部署，导致新数据的应用延迟，而此时快速迭代往往至关重要。

为克服上述局限与挑战，我们提出了一种统一方法，融合流式处理与分析系统的优势：在流动过程中进行复杂模式匹配和语义增强，同时使增强后的记录立即可在分析数据平面中查询。第 3 节将对此方法进行介绍。

3 FLUXSIEVE 方法

大规模数据处理系统——从传统关系型数据库到现代数据湖仓——在海量数据集上执行重复且密集的查询时，往往会遇到性能瓶颈。为应对这一挑战，我们提出 FluxSieve，在将数据摄取到分析平面之前，引入**流内预过滤与增强机制**。该方法旨在通过有选择地处理和丰富记录，再将其送入分析存储层，从而降低查询延迟并优化资源利用率。

3.1 核心思想

我们通过将有针对性的高价值计算从分析平面移至摄取路径，将数据库由内向外翻转，同时保留分析平面作为权威的事实来源。流内处理会执行无状态预过滤和轻量级增强，并根据观察到的工作负载进行调整，将原始信号转换为适合高效查询执行的记录。随后，分析平面会解析这些增强信息（例如标记、紧凑结构、部分聚合），以加速查询执行，同时不牺牲正确性或完整性，也不会造成重复存储数据。这种统一方案旨在保持清晰的逻辑：

- **权威性：** 分析平面仍是存储、模式和结果的权威来源。增强属于提示或加速手段，而非正确性的替代品。
- **性能：** 对重复或成本高昂的操作（过滤）进行流内预计算，从而缩小候选集，并降低查询时的 CPU 和 I/O 开销。
- **适应性：查询分析器**（分析平面）会检测开销大且频繁的查询，并通过**更新器组件** 注册条件，以便进行流内编译和多模式匹配（例如 Hyperscan [24]）。

该方法运行于数据源与分析平面之间，利用流处理器执行以下关键操作：

- **预过滤：** 匹配与查询相关的记录。
- **富化：** 为相关记录添加轻量、节省存储空间的元数据，以编码查询相关性。这些富化字段旨在最大限度地减少存储开销，同时在列式编码方案（例如游程编码）下保持高度可压缩，因此特别适用于面向列的存储系统（例如分析型数据库、湖仓）。

3.2 架构

该架构由以下模块组成：

(1) 数据源接口： 从异构来源（例如事件流、日志聚合器）采集原始记录。**(2) 流处理器：** 利用高效的多模式匹配实时执行过滤和增强逻辑，为记录添加预计算查询结果。**(3) 更新器组件：** 使过滤条件和增强规则与不断演变的查询模式保持同步，并在需要时触发模式匹配引擎重编译。**(4) 分析平面：** 存储和查询执行层，利用增强后的元数据高效执行查询。它内嵌 查询分析器 ，用于分析查询和执行计划，同时将开销较大的/频繁出现的过滤条件转移至流内处理。**(5) 查询映射器：** 将传入查询转换为经过优化的内部查询，以利用预计算字段。查询映射器使分析平面能够绕过开销高昂的全表扫描和谓词求值。

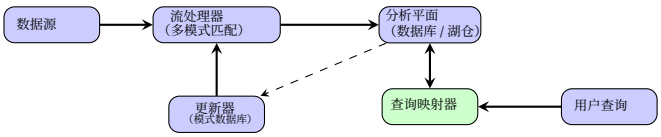


图 2：我们方法的架构。

3.3 流内多模式匹配

流处理器的一项关键要求，是能够针对每条传入记录，以极低的延迟开销同时评估许多过滤条件。朴素地遍历数百或数千个过滤谓词，会引入高得难以接受的单条记录处理成本，从而削弱流内处理的优势。

因此，流处理器利用**多模式匹配算法**来实现大规模高效匹配。我们使用**Hyperscan** [24],[2] 实现了多模式匹配；它是一个高性能正则表达式匹配库（最初由英特尔开发），可将多个模式编译为经过优化的有限自动机。Hyperscan 在对输入数据的一次遍历中评估所有已注册模式，能够以可预测且有界的单条记录延迟匹配数千个模式，因此非常适合流处理工作负载。

匹配引擎的选择可以根据工作负载特征进行调整：正则密集型工作负载受益于 Hyperscan 的编译自动机，而以关键词为中心的过滤则可能更适合 Aho-Corasick算法 [2] 或基于哈希的方法。在这项工作中，我们选择使用 Hyperscan 作为多字符串匹配引擎，因为它是一种现代方案，具有适当的算法效率，并支持在云原生流处理器上进行高效实现。与使用 Hyperscan 多模式引擎处理单个模式的方案不同，[7], 我们充分利用其能力，注册多个模式，以高效过滤流式记录。

[2]Java 绑定可在 https://github.com/gliwka/hyperscan-java 获取

3.4 面向即时过滤条件更新的动态自适应

尽管流处理执行中面向并行度与弹性的自适应已取得显著进展 [21],，但仍缺乏能够支持流处理器逻辑即时自适应、且无需代价高昂的应用重新部署的方法。在本工作中，我们关注对持续对输入记录执行过滤条件的流处理器进行即时更新。本节描述我们的方法及其与分布式流处理框架的集成。

分析平面中的监控模块可通过工作负载分析识别感兴趣的查询 ，例如检测频繁执行的查询、反复出现的过滤模式以及成本高昂的查询片段。过滤条件和增强模式会通过更新器组件传播至流处理器。这一反馈回路确保流处理逻辑始终与查询需求保持一致。

更新器组件维护一个**模式匹配引擎**，其中封装了所有生效的过滤条件及其关联的增强规则。当识别出新的感兴趣查询，或现有模式不再适用时，系统会执行以下步骤来更新匹配引擎：

(1) 增量计算：更新器计算当前模式集与目标模式集之间的差异，识别新增、修改和移除项。**(2) 模式匹配引擎重编译：**对于 Hyperscan 等引擎，会将模式匹配引擎重新编译为经过优化的自动机。尽管这一步编译计算开销较大，但它会异步执行，不会阻塞正在进行的流处理。**(3) 即时规则更新：** 新的模式匹配引擎就绪后，会将其切换至活动流处理器。在途记录会继续使用先前的匹配引擎进行处理，直至切换完成，从而确保切换期间不会错误地过滤任何记录。**(4) 一致性传播：**查询映射器收到模式已更新的通知后，在查询时即可正确解读具有更新后富化字段的新摄取记录。

这种架构使过滤逻辑能够**持续演进**，而不会造成服务中断，使系统能够适应不断变化的查询模式、支持新的分析用例，并淘汰过时的过滤器，同时避免流处理停机。

3.4.1 架构概述。 模式更新机制遵循一种分布式协调架构，旨在最大限度地降低延迟、确保一致性，并避免干扰正在进行的流处理。图 3 展示了完整的更新生命周期。

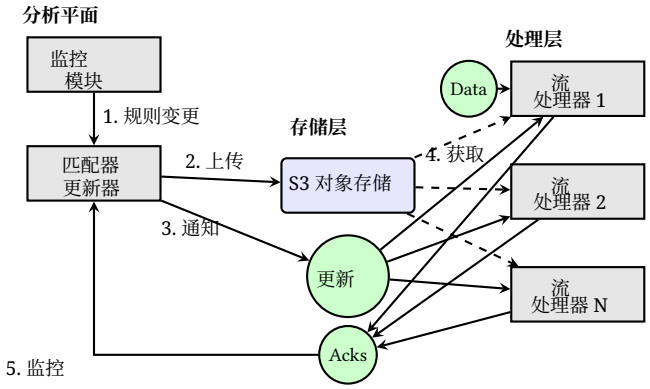


图 3：模式匹配引擎更新：(1) 检测变化，(2) 更新器编译并上传，(3) 发送通知，(4) 处理器获取匹配引擎，(5) 确认。

该架构由四个主要组件构成 nts:

- **过滤规则管理接口：**接收来自分析平面监控模块的模式更新。此接口预计会根据工作负载遥测、查询模式分析和自主学习独立运行，以检测频繁且开销高昂的查询。

- **匹配器更新器：**负责模式匹配引擎编译、版本控制和分发的中央协调器。收到更新后的条件后，它会将模式编译为经过优化的匹配自动机（例如 Hyperscan 匹配引擎），分配版本标签，并协调在所有流处理实例中的部署。

- **对象存储层（S3）：**[3] 编译后的模式匹配引擎会被序列化并存储在分布式对象存储中。这一设计解决了两个关键问题：(1) 大型编译匹配引擎（可能达数十至数百兆字节）会给 Kafka 主题带来过多开销；(2) 对象存储能够以较低成本提供高可用分发，并内置版本控制。

- **流式应用程序（匹配器）：**每个分布式流 m 处理器实例都会维护对当前活动模式匹配引擎的本地引用。实例通过 Kafka 收到更新通知后，会异步从对象存储获取新编译的匹配引擎，确认下载的文件与预期版本一致，并使用存储的校验和（例如与对象关联的哈希值）验证其完整性，然后执行热切换。

这种架构将资源密集型编译与分发（去中心化且带宽高效）解耦，从而支持可扩展的部署，并以较低开销更新匹配引擎。

3.4.2 模式匹配引擎更新。更新流程如下：

（1）监控模块检测到变化，或由来自湖仓或数据库的查询分析器触发，并生成更新后的过滤条件集。

[3]需要注意的是，亚马逊 S3 是我们的方法所支持的一种对象存储技术示例。不过，我们的方法也可以在任何具备类似功能特性的其他对象存储中实现。

（2）匹配器更新器将过滤条件编译为带版本的模式匹配引擎，并使用唯一的版本标识符将其上传到 S3。（3）包含版本标签和 S3 对象引用的通知消息会发布到专用的 Kafka 主题。（4）所有流式应用程序实例都会接收更新通知，从 S3 获取已编译的模式匹配引擎，并做好激活准备。（5）获取并验证后，每个实例都会将新的模式匹配引擎切换到其活动处理流水线中。（6）通过 Kafka 发回确认消息，以确认集群内各实例已成功激活匹配引擎。此步骤为可选。

3.4.3 Kafka Streams 上的实现。 模式更新机制通过 Kafka Streams 应用程序实现，采用双拓扑设计：一个拓扑处理带有活动模式匹配的主数据流，另一个拓扑则通过专用控制平面处理模式匹配引擎更新。

通信协议。 更新协调协议利用 Kafka 的精确一次语义和消费者组协调机制，确保分布式实例间的模式一致激活。匹配器更新器可以订阅确认主题，以监控发布进度，并检测是否有实例未能在可配置的超时时间内激活新的匹配引擎。

对象存储分发。 系统并未将编译后的模式匹配引擎直接嵌入 Kafka 消息，而是采用**基于引用的分发模型**。这种方法有以下几项优势：

(1) **带宽效率：**Kafka 消息保持轻量（通常小于 1 KB），避免了大型消息有效载荷导致的吞吐量下降。用于数千个模式的编译版 Hyperscan 模式匹配引擎可能超过 100 MB，如果以内联方式传输，将使 Kafka 代理不堪重负。(2) **成本优化：** 对于大型二进制对象，S3 存储成本远低于 Kafka 保留成本。此外，S3 存储和生命周期策略支持自动归档较旧的模式版本。(3) **缓存：** 流处理器可以利用 HTTP 缓存标头或类似服务，降低地理位置分散的部署中的检索延迟。(4) **版本控制：**S3 对象版本控制可确保每个模式匹配引擎版本保持不可变且可访问，从而支持回滚能力和审计跟踪。

Kafka Streams 集成。 更新处理逻辑被实现为 Kafka Streams 拓扑中的一个独立处理器，独立于主数据处理管道运行。该处理器订阅 matcher-updates 主题，并触发更新工作流，而不会阻塞记录处理。该集成利用 Kafka Streams 的状态存储，维护有关当前活动和待处理模式版本的元数据，以跟踪：

- 当前活动模式版本
- 待更新版本（如果更新正在进行）

• 用于审计和监控的激活时间戳

数据处理拓扑通过共享的线程安全引用访问当前活动的模式匹配引擎。当有新的模式匹配引擎时，该引用会更新，以确保后续记录使用最新的模式进行处理。

4 评估方法

本节介绍评估所提出方法的方法。就评估方法而言，在适用且合适的情况下，我们采用了良好实践，例如来自 [10, 18] 3 的实践。我们将我们的方法（第 节）与基线系统进行了比较。我们将处理所测试查询的最合适方法视为基线；例如，在 Apache Pinot 中，基线是全文搜索（FTS）索引，因为查询会在字符串字段中搜索术语。比较涵盖以下内容：

• 基线 – 摄取的事件（Pinot）或写入 Parquet 的事件；
• FluxSieve（我们的方法）通过流内过滤和增强实现 – 使用相同的工作负载，但通过流处理器对记录进行过滤和增强，然后将其流入 Pinot 或 Parquet。

4.1 指标

以下是我们评估中考虑的主要指标：

• 查询性能：在不同系统中，对增强数据集与非增强数据集执行具有代表性的分析查询。• 摄取吞吐量。• CPU资源使用量。• 存储大小：段/文件大小。

查询性能评估以多次运行的中位数作为主要指标，并按照良好实践，通过反复对测量值进行重采样并重新计算中位数，使用自助法报告95%置信区间 [10, 18]。

用于评估的基础查询工作负载如下：

• 查询1：筛选字符串字段，搜索一个不匹配的词项。

• 查询2：筛选字符串字段，搜索极少满足匹配条件的内容。

• 查询 3：筛选字符串字段中符合某个词项条件的记录，并计数匹配项的数量（聚合）。• 查询 4：多字段搜索。搜索两个字符串字段的内容中都包含任意词项的记录。

4.2 查询执行

在我们的实验方法中，我们明确区分冷 和热 查询执行。**冷运行** 指底层系统无法从驻留在内存（随机存取存储器，RAM）中的数据、缓存数据、已编译的查询计划或其他预热效果中获益的执行。

为近似这种行为，我们会在冷运行前执行缓存清理操作，包括：(i) 清除操作系统页缓存（第 5 节）；以及 (ii) 重新部署用于承载待测分析系统的基于 Kubernetes 的基础设施（第 6 节）。这些步骤旨在移除之前执行所保留的数据和元数据，确保冷运行能够模拟大数据场景下的性能：在该场景中，所需的数据和查询计划必须从持久化存储中加载，因为预计它们无法装入内存。

相比之下，**热运行** 是指数据库引擎、操作系统及周边基础设施提供的系统缓存机制得以保留的执行。对于热运行，我们不会在各次执行之间清除缓存，因此缓冲池、操作系统页缓存和计划缓存等组件可以保留先前运行产生的数据和已编译查询计划。由于冷运行和热运行的计时特征可能存在显著差异，我们分别收集并报告这两种模式下的性能指标。这种区分使我们能够同时捕捉系统的初始开销表现（数据无法装入内存时，即冷运行）和稳态性能（热运行）。

4.3 摄取和查询的数据

在实验中，我们重点关注针对数据集的查询工作负载，这些数据集的规模从 500 万条记录到 4000 万条记录不等。具体规模会因设置和环境而异（例如，不同的集群配置或云部署），但始终处于此范围内。我们认为，这一数据量能够代表现实中的运营数据集，同时也在实验代表性与云成本效益之间保持了合理平衡。对于更大的数据量，我们预计观察到的性能趋势仍然成立，因为超出这一范围进行扩展不会从根本上改变底层系统行为和瓶颈。

每条记录都有一个代表可观测性日志的逻辑模式：一个时间戳字段，其数值对应事件时间；一个状态字段；一个事件类型字段；以及 2 到 5 个额外的字符串内容字段，每个字段包含 60 个单词。内容字段的数量取决于场景。我们至少使用两个内容字段来支持多字段查询（查询 4），确保评估涵盖跨多个属性的现实搜索与筛选模式。第 6 节的场景中引入了额外的内容字段，以便在冷数据上有效运行查询；每个查询都针对不同字段运行，以避免使用缓存数据，并减少所需的重新部署次数。这种设计使我们能够在现实且可扩展的部署中，对系统在冷查询和热查询执行条件下的行为进行压力测试和评估。

5 流式数据湖评估

本节展示了我们的方法集成到 DuckDB [19], 后的评估结果。DuckDB 是一款嵌入式分析型数据库，专为高性能分析型工作负载而优化。考虑到 DuckDB 的高效性，将我们的方法与 DuckDB 进行比较，可以检验流内增强是否有利于嵌入式分析场景。该场景模拟了流式数据湖：流处理器近乎实时地处理记录并将其写入文件，使摄取的记录能够以低延迟进行查询。这一实验设置使我们能够衡量该方法的关键方面，包括：(I) 通过对新增属性进行过滤器下推所实现的扫描量减少；(II) 相较于未增强文件的查询性能提升；以及 (III) 相对于基线，该方法可能引入的开销。

5.1 实现与设置

图4展示了两条端到端流水线，它们共享同一数据流：*Kafka* → 流处理器 → *Parquet* → *DuckDB*。**基线（上方泳道）**是一个流处理应用，每秒从 Kafka 主题消费 10000 条日志事件，将有效载荷反序列化后写入 Parquet 文件。生成的文件仅包含原生日志属性（时间戳、状态、content1、content2 等）。DuckDB 随后查询这些 Parquet 文件，并利用向量化、投影和谓词下推。我们选择优化的全表扫描作为 DuckDB 的基线，因为它原生不支持全文搜索（FTS）索引。在我们的实验场景中，当前的全文搜索索引扩展[4] 性能低于优化的全表扫描。

我们的方法（下方处理通道）是一种增强型处理器，部署在同一实例上，以相同的记录数和速率为输入，针对字段执行一组 1 000 个模式匹配条件。[5] 流处理器会输出一个额外的列 matched_rule_ids INT[]，其中包含稀疏数组，用于标识匹配项。据我们所知，由于 DuckDB 会将缺失列视为 NULL，且支持高效的嵌套类型，因此这种数组表示方式是向本地分析工作负载提供匹配元数据最合适且最节省存储空间的方式。

保持 Kafka、Parquet 格式、压缩设置和 DuckDB 查询引擎不变，可以隔离增强策略的影响。通过比较基线与采用过滤和基于数组的增强的 FluxSieve，我们可以量化预先计算匹配信息对 Parquet 文件大小和查询性能提升的影响。

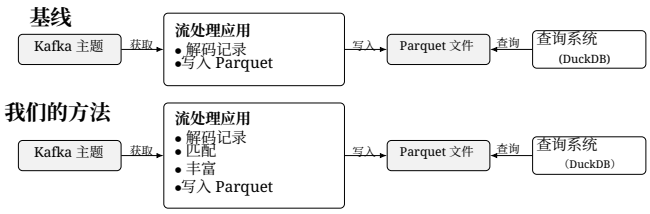


图4：部署对比。

两个流处理器运行在配置相同的虚拟机上，每台虚拟机配备 1 个 vCPU 和 4 GiB 内存，在生成 1000 万个事件（10 分钟的流量）的同时，保持 10000 个事件/秒的摄入速率。对于查询时基准测试，我们复制生成的 Parquet 文件，并在两种硬件配置下评估每种工作负载：(i) 与摄取时相同的单核配置，以及 (ii) 一个 4 vCPU / 16 GiB 内存配置反映了面向分析师的真实场景，在这些场景中，临时查询必须快速扫描大量日志。这一双配置设计旨在衡量增强如何与 DuckDB 的查询内并行相互作用，以及预物化的已匹配_规则_ID 的收益在有更多 CPU 资源可用时是否仍然存在——甚至被放大。

[4]https://duckdb.org/docs/stable/core_extensions/full_text_search [5]流处理器中运行的匹配引擎会过滤两个文本字段。

5.2 开销分析

图5将仅解码记录并将其写入 Parquet 的基线 DuckDB 摄取流水线，与我们在写入 Parquet 之前额外执行匹配和增强的扩展流水线进行了比较（图4）。两种设置均在同一单核实例上运行，输入速率同为每秒 10000 条记录，并包含 30 秒的预热阶段。两种方法的吞吐量曲线几乎无法区分，表明我们方法中额外的匹配和增强逻辑不会对 Kafka 分区引入任何明显的延迟或背压，并保持了相当的摄取吞吐量。

CPU 使用率衡量的是我们方法的额外处理开销。在数据输入开始之前，我们的方法平均使用 17.73% 的 CPU，而基线为 14.71%，即比基线高出 20.5% 的 CPU 使用率。由于我们的方法会在启动时编译匹配引擎，因此产生这样的开销是预期之中的。在稳态处理期间，基线的平均 CPU 使用率为 53.18%，而我们的方法为 59.19%，相应地，基线的 CPU 消耗低 11.3%。这一差异主要反映了在流内筛选并丰富记录所需的额外工作。[6]

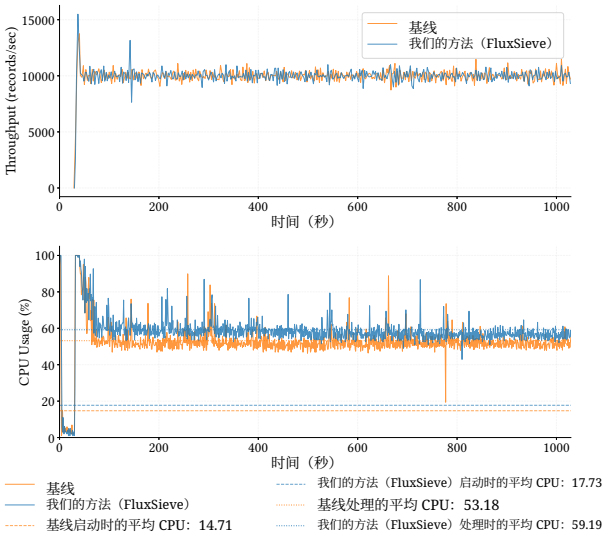


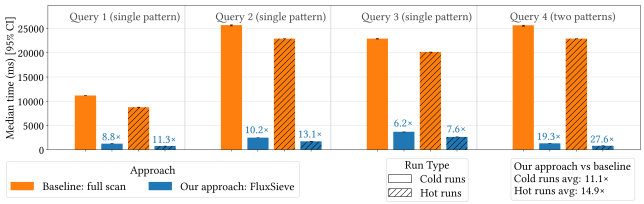
图 5：开销分析。

总体而言，这些结果表明，我们的方法能够在维持与更简单的基线相当的摄取吞吐量的同时，仅适度增加 CPU 使用量。[7] 鉴于额外的 CPU 开销换来了经过预筛选和丰富的数据，从而简化并加速了下游查询，因此对于重视查询性能和效率的部署而言，这种开销是可以接受的。

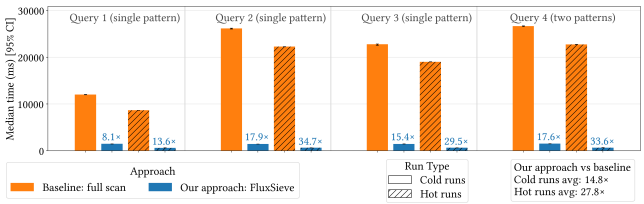
5.3 查询性能

实验使用包含 1000 万条日志记录的数据集作为代表性工作负载。每条记录包含第 4.3 节中描述的结构化字段和两个文本字段。此设置旨在

[6]为减少变异性，实验中将 CPU 频率固定为 2.5 GHz。[7]我们还测量了使用 Zstandard（zstd）压缩的 Parquet 文件的存储大小，结果显示，除个位数精度外，大小均无差异。

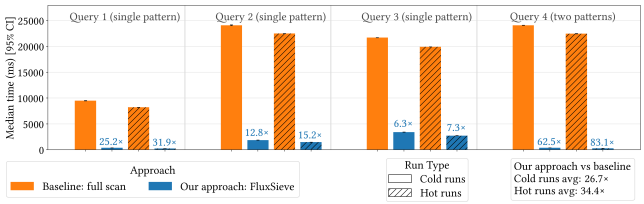


(a) 返回记录

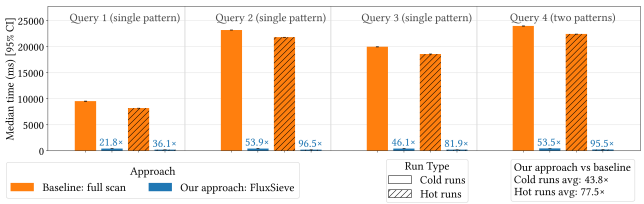


(b) 计数记录

图 6：并行度为 1： ≈5k 个文件，每个文件 ≈2k 条记录。



(a) 返回记录



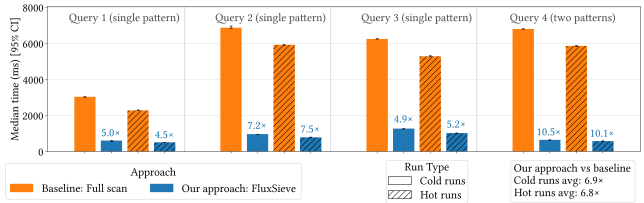
(b) 计数记录

图 7：并行度为 1： ≈1k 个文件，每个文件包含 ≈10k 条记录。

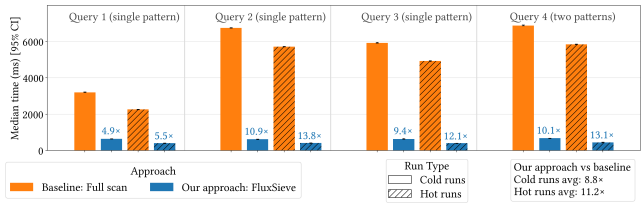
模拟真实的可观测性或日志数据，其中结构化属性和自由文本字段的组合会被反复查询。我们考察对 DuckDB 等分析引擎至关重要的两个维度：(1) 1000 万条记录在物理上如何分布到 Parquet 文件中（许多小文件与较少的大文件），以及 (2) 可用的查询内并行度有多大（单核与四核执行）。对于每种组合，我们评估两种常见的查询模式：(i) 返回所有匹配记录的查询（复制场景），以及 (ii) 计数匹配记录的查询（聚合场景）。这些场景十分重要，因为文件布局和并行度是数据湖部署中的核心配置选项，而返回结果与计数则代表两种主要的使用模式（探索性检索与分析性汇总）。

1 个 *CPU* 核心， *≈5k* 个文件， *≈2k* 条记录/文件（图 *6*）。在只有一个核心且 Parquet 文件数量众多、每个文件较小的情况下，DuckDB 需要打开并扫描大量文件，才能覆盖这 1000 万条记录。在需要复制记录以返回所有匹配项的复制场景（图 6a）中，由于引擎既要定位相关文件片段，又要物化匹配行，因此每次查询都会产生明显开销。在计数场景（图 6b）中，执行通常更快且更稳定，因为 DuckDB 可以在流内进行聚合，而无需返回完整行，从而减少数据传输和结果物化。总体而言，这些结果表明，在单核环境下，查询延迟对文件管理开销较为敏感；即使逻辑工作负载相同，简单聚合（计数）也显著比返回完整记录更省资源。

1 个 *CPU* 核心， *≈1k* 个文件， *≈10*每个文件 *k* 条记录（图 7）。将相同的 1000 万条记录整合到数量更少、体积更大的 Parquet 文件中，可以减少元数据和文件打开开销。在复制场景（图 7a）中，与图 6 的结果相比，查询时间有所缩短，反映出顺序扫描效率更高，且文件句柄更少。在计数场景（图 7b）中，性能提升更加明显，因为 DuckDB 可以按文件处理更大的批次，并快速聚合结果。这表明，即使不使用并行度，文件压缩（缓解小文件问题）也有益处。



(a) 返回记录

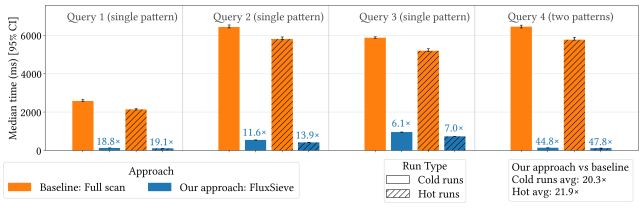


(b) 计数记录

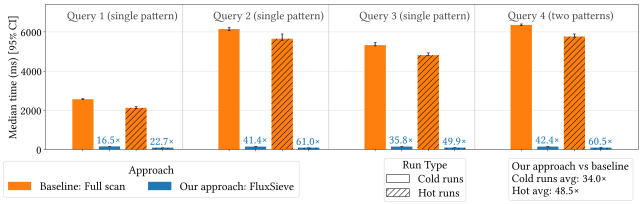
图 8：并行度为 4： ≈5k 个文件，每个文件包含 ≈2k 条记录。

4 个 *CPU* 核心， *≈5k* 个文件， *≈2*每个文件 *k* 条记录（图 *8*）。DuckDB 可以利用四个核心/线程并行扫描这些文件。在复制场景（图 8a）中，并行度弥补了部分小文件开销：与 1 个核心 / 5k 个文件的配置相比，查询时间中位数有所缩短，但受每个文件的开销和协调成本限制，收益有限。在计数场景（图 8b）中，并行扫描与低成本聚合相结合，进一步提升了性能；但扩展性呈次线性，因为引擎必须协调大量文件并合并部分聚合。

4 个 *CPU*核心， *≈1k* 个文件， *≈10*每个文件 *k* 条记录（图 *9*）。这是一个最佳场景，拥有四个核心且只有约 1k 个 Parquet 文件，DuckDB 可以为每个工作线程分配更大的数据块，并摊销文件打开开销。在复制场景中（图 9a），查询延迟大幅下降，表明可用资源得到了近乎理想的利用。



(a) 返回记录



(b) 计数记录

图 9：并行度为 4： ≈1k 个文件，每个文件 ≈10k 条记录。

CPU 资源。对于计数 场景，结果甚至更好（图 9b）：查询速度显著提升，体现了并行I/O与轻量聚合的高效结合。

研究发现总结。 综合所有结果，可以得出四项主要观察。首先，**我们的方法非常高效，在 CPU受限设置下实现了最高的加速比**：全扫描基线的 CPU 消耗显著更高，因此从增加核心中获益更多；相比之下，我们的流内过滤方法本身已经很高效，只需较少的硬件即可实现较低的查询延迟。例如，在单核心配置下，我们的流内过滤方法相较于基线取得了巨大收益：计数查询和复制查询的加速比通常超过 30×，并且可能超过 60×。这些结果表明，在 CPU 资源有限时，避免全扫描并将过滤下推至摄取路径能够带来显著优势。其次，**并行度有所帮助，但受布局限制**：增加 CPU 核心数可以提升性能，尤其是在文件数量适中的情况下；然而，当存在数千个小文件时，每文件开销会限制增加核心所带来的收益。最后，**计数记录的速度更快**：尽管计数查询涉及聚合，但它们无需物化和传输完整的结果行，因此比返回所有匹配记录的查询更快。

6 分析系统评估

实时在线分析处理（RTOLAP）系统旨在摄取高速记录流的同时，最大限度地降低查询延迟。Apache Pinot [11] 便是这种设计的典范，能够持续处理每秒数百万个事件，并服务大量并发查询。由于我们的流式富化层专注于加速下游分析，因此我们通过将该方法集成到 Apache Pinot 的部署中来衡量潜在改进。Pinot 的 broker-server 架构能够提供精确的单次查询计时，从而支持在毫秒粒度上对富化流水线与基线流水线进行统计稳健的比较。

衡量多模式匹配和记录富化对分析数据平面存储的影响至关重要。Pinot 的列式段采用字典编码、游程编码和位压缩编码来压缩列值，在不牺牲扫描效率的情况下实现紧凑的磁盘占用。将评估重点放在查询执行时间（以毫秒报告）上，使测量维度与 Pinot 的核心优势保持一致：分析查询执行时间短，且列式表示具有较高的存储效率。

6.1 与 Apache Pinot 集成

应用日志被写入 Kafka 分区主题，其中各分区被分配给基线 Apache Pinot 的服务器进行数据摄取。Pinot 的 Kafka连接器直接将每个分区摄取到 REALTIME表中，其模式与原始记录字段保持一致。相关索引配置包括：(1) 默认 （称为全表扫描）：采用字典编码的前向索引；(2)默认 + 文本 （称为文本索引）：在相同配置的基础上，为文本字段增加 Pinot 按列建立的全文搜索索引，*json_context*），可加速正则表达式和关键词谓词。在我们的实验中，全表扫描 配置的查询性能始终比文本索引慢数个数量级。因此，我们在实验分析中省略全表扫描 ，因为全文索引提供了更具参考价值的基线。

采用流内过滤和增强的 *FluxSieve*。 它在 Kafka 与 Pinot 之间加入了一个流处理器（第 3.3 节和第 3.4 节）。对于每条日志记录，该处理器使用多模式匹配应用 1 000 条布尔过滤规则。系统会输出增强后的 记录，在基线模式的基础上增加 1 000 个布尔列 rule_1……rule_1000。[8] 每次匹配恰好有一个字段被设置为 TRUE；其余字段默认为 FALSE。增强后的流被发布到另一个 Kafka 主题[9]，Pinot 随后将记录摄取到一个类似于默认 +文本 的表中（包含带全文搜索索引的大型文本列）。[10]

我们为每个需要筛选的记录文本字段分别部署一个模式匹配引擎实例（总共五个），所有实例均运行在同一个流处理器中。每个实例负责评估相应的字段和过滤规则。匹配任一规则的记录都会通过该标识符进行增强。该设计在保持 Pinot 索引策略不变的同时，隔离了流内过滤/增强的开销。

6.2 设置

Apache Pinot 部署在由 Amazon EBS gp3 卷提供支持的 Kubernetes集群上，卷采用 ext4文件系统格式化。Pinot 集群采用标准的控制器–代理–服务器架构：一个控制器、一个代理和四个服务器实例，以提供合理的查询并行性。Pinot 控制器和代理 Pod，以及每个 Pinot 服务器 Pod，均调度在 m6i.xlarge实例上。每个 Pinot 服务器 Pod 都运行在专用实例上，而其他 Pod 则共用一个实例。[11]

[8]我们将额外的布尔列具体化，以简化分析。在生产环境中可以采用更高效的表示方式，例如将匹配项存储在稀疏列表中。[9]额外的 Kafka 主题用于提供一种可靠的方式来评估我们的方法。通过使用连接器将记录直接从流处理器传输到 Apache Pinot，可以避免使用这一额外主题。[10]需要注意的是，针对我们方法所用表执行的查询，查询的是经过增强的字段。不过，我们仍然在该方法中创建全文搜索索引，以确保公平比较，因为强度较低的查询仍然可以直接查询全文搜索索引。[11]有关 Apache Pinot 架构和部署的深入介绍，我们建议读者参阅原系统论文 [11] 及其文档 [16]。

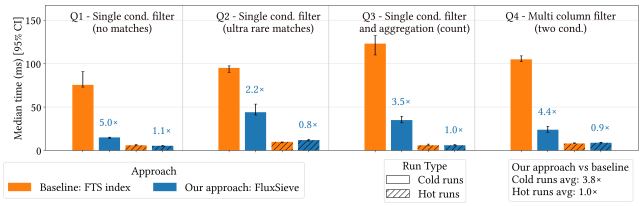


图 10：500 万条记录。

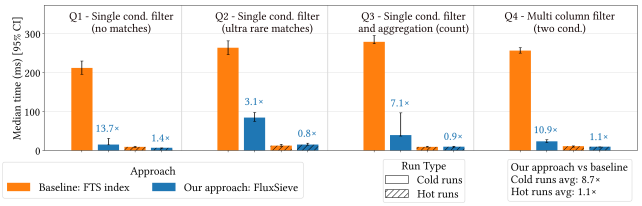


图11：1000万条记录。

在数据摄取方面，我们使用两个通过 Strimzi 部署在 Kubernetes 上的 Apache Kafka 代理，每个代理都运行在 m6i.2xlarge实例上。输入数据写入包含 40 个分区的 Kafka 主题，从而支持并行摄取到 Pinot 服务器，并满足实时分析工作负载的高吞吐量需求。

6.3 查询性能

在此实验场景中，第 6.3.1 节介绍了超高选择性查询场景，其中匹配记录极为罕见，这与本文的背景和研究动机相符（见第 2 节）。此外，我们还在第二种场景中评估查询性能，即高选择性查询（第 6.3.2 节）。在该场景中，匹配项不那么罕见，而且一旦出现，其数量比第 6.3.1 节中的高一个数量级。选择这些选择率不同的场景，是为了评估在记录匹配情况各异时，性能趋势是否相似。此外，对于执行时需要因环境重新部署而产生基础设施开销的冷查询，我们将每次查询执行重复 10 次。至于热查询，由于无需重新部署，我们将每条查询重复执行 40 次。

6.3.1 超高选择性查询。

500 万条记录（图 *10*）。对于包含 500 万（5M）条记录的数据集，不同评估方法的查询执行时间中位数存在明显差异：文本索引搜索与我们的流内过滤技术。文本索引搜索能够显著提升查询性能，但对于超高选择性查询，仍会产生明显开销。相比之下，我们的流内过滤方法在几乎所有查询中，无论是针对冷数据还是热数据，都能实现最低的执行时间中位数。我们的方法与文本索引基线之间的差距，在冷数据运行场景中尤为明显，这主要归功于我们的方法能够进行数据剪枝，从而避免 I/O 瓶颈。这表明，即使数据集规模适中，利用流内过滤也能在超高选择性条件下带来切实的性能提升。

*1000*万条记录（图*11*）。在1000万（10M）条记录下，由于数据集更大，各方法的绝对执行时间更高，但方法之间的相对排序保持相似。无论在冷数据还是热数据上，文本索引基线在几乎所有查询中都始终慢于我们的流内过滤方法。虽然从500万到1000万条记录的查询时间增长是可观察到的，但我们方法的增长幅度小于文本索引基线，表明随着数据量增加，我们的方法具有更好的可扩展性和加速比。

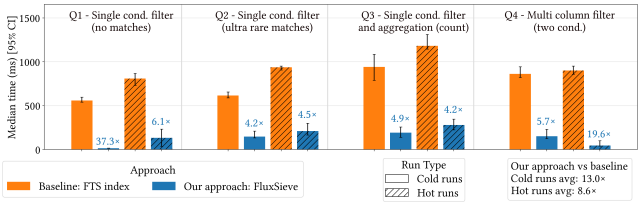


图12：2000万条记录。

*2000*万条记录（图*12*）。在2000万（20M）条记录下，两种主要方法之间的差距变得更加明显。文本索引查询的中位数执行时间大幅增加，而我们的流内过滤方法在整体上始终保持更低的查询时间，冷运行和热运行均是如此。在这一规模下，我们的方法在所有查询中仍明显优于文本索引基线，这凸显出在超高选择性条件下，随着数据集增长，流内过滤的优势会进一步放大。冷数据运行与热数据运行之间的关系也表明，该方法对缓存状态具有稳健性，这在查询可能频繁命中冷数据的部署场景中尤为重要。这些图还表明，无论数据来自冷缓存还是热缓存，流内过滤的性能优势都得以保持，从而进一步说明收益来自执行策略，而非仅来自缓存效应。

4000 万条记录（图*13*）。当扩展到4000万（40M）条记录时，在较小规模下观察到的趋势依然存在，并且变得更加明显。文本索引基线显示出明显更高的中位数执行时间，这反映了在此规模下索引结构与查找的成本。我们的FluxSieve方法在冷数据和热数据上，对所有查询都实现了显著更低的执行时间。两条曲线的平行走势表明，在这一超高选择率场景中，随着数据集的增长，我们的方法保持并扩大了其优势。这表明FluxSieve能够应对大规模部署，同时保持可预测且较低的查询执行时间。

整体加速比与文本索引搜索的对比（图*14*）。图14中的汇总视图总结了我们的流内过滤方法相对于文本索引基线在所有数据集大小和查询类型上的加速比。对于所有查询变体（纯过滤、过滤加聚合和多重过滤），我们的方法在冷数据和热数据上均实现了性能提升。加速比通常大于1，并且执行速度常常达到数倍。随着数据集大小从500万条记录增加到4000 万条记录，

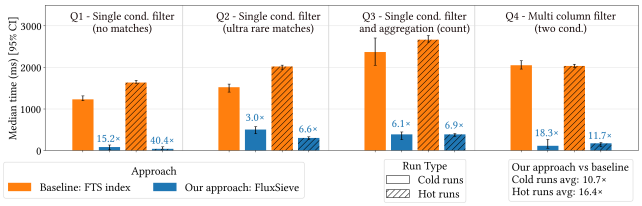


图13：4000 万条记录。

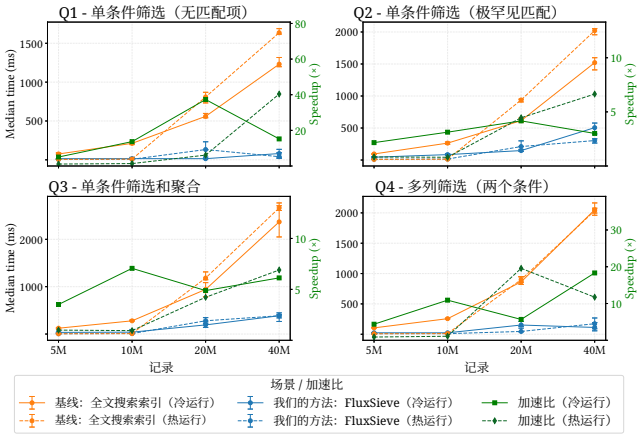


图14：超高选择性：总体比较。

加速比往往会增加，尤其是在冷数据运行中，这表明文本索引基线的相对成本比流内过滤增长得更快。这些结果证实，我们的方法系统地优于文本索引搜索，并且这些优势会随数据规模和查询复杂度而扩大。

6.3.2 高选择性：包含计数聚合的扩展场景。图15展示了在高选择性场景下，我们的流内过滤方法与文本索引搜索的性能对比；在此场景中，匹配记录仍然相对稀少，但比超高选择性场景中的匹配记录明显更多。除了原有的四种查询类型外，该图还展示了查询1、2和4的扩展场景：系统不仅过滤记录，还会计算匹配项的计数（即简单聚合）。这些扩展查询表示如下：

- 带计数的查询*1*：过滤 +聚合（计数），
- 带计数的查询*2*：过滤 +聚合（计数），• 带计数的查询*4*：两个过滤器 + 聚合（计数）。

这些查询对应的场景中，不需要完整记录，只需匹配项数量的摘要。引入这类查询很重要，因为许多现实应用（例如仪表板、警报）可以针对经过筛选的数据子集进行聚合，而不必获取全部原始记录。此外，聚合会改变执行特征：当只需要计数时，引擎可以避免物化和传输所有匹配的记录，从而减少内存和网络

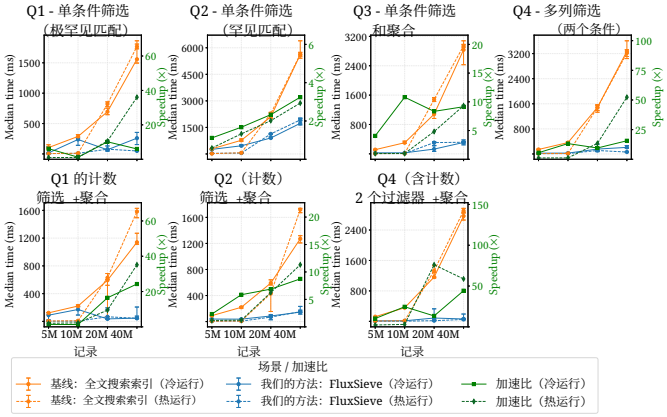


图15：高选择性：总体比较。

压力。分别在启用和不启用计数的情况下评估查询 1、2 和 4，可以衡量 FluxSieve 的性能优势是否能在不同查询语义下持续保持。

与高选择性仅过滤查询的比较。在查询1、2和4等查询中，结果表明，“带计数”变体的性能表现与纯过滤版本相似。对于每种数据集大小，FluxSieve 方法始终快于文本索引基线；“带计数”查询的加速比曲线有时甚至高于仅过滤查询的曲线。这表明，添加聚合不会削弱流内过滤的优势。

从加速比来看，冷数据和热数据运行均表明，我们的方法在所有查询变体中都能保持加速效果，其中更复杂的查询（例如查询 4 和带计数的查询 4）通常表现出最高的加速比。这说明，随着查询的选择性提高和/或复杂度增加（包含多个过滤器和聚合），即使请求的操作是计数而非完整记录检索，基于文本索引的基线的相对成本增长速度也快于流内过滤。

研究发现总结。 我们的方法在不同选择率、数据量和查询类型下均表现出稳定的性能提升：[12]

- 尽管在选择性较低、聚合不那么极端的查询中，绝对执行时间有所增加，但我们的方法 FluxSieve 仍能带来显著的加速比。这证实了 FluxSieve 的优势并不局限于超高选择性场景。
- 在超高选择性场景下观察到的数据集大小扩展趋势（从 500 万条记录到 4000 万条记录）同样适用于高选择性场景。随着数据量增长，FluxSieve 的扩展性优于基线全文搜索索引。这表明 FluxSieve 具有良好的可扩展性。
- 对于热数据查询，当数据集无法装入主内存时（例如，超过2000万条记录），我们的方法可显著提升性能。对于冷数据查询，

[12]这些性能提升几乎不会带来额外的存储成本。在使用 Apache Pinot 的默认压缩和编码配置进行的实验中，我们的方法与基线之间的数据大小差异始终低于 2%。

我们的方法在所有场景下都能带来显著的加速比。冷、热运行之间的对比凸显了 FluxSieve 的优势，以及数据剪枝等优化所带来的效果。

7 相关工作

鉴于流数据库作为持续摄取数据、对实时数据流提供低延迟查询并无缝支持历史数据查询的系统而兴起， [5], 我们的方法可被视为一种经过改进的流数据库，它统一了流式处理与分析数据平面。FluxSieve 尤其侧重于优化结合多模式匹配的复杂过滤，并将其与分析数据平面相统一。关键相关技术包括物化视图 [14] 和增量视图维护（*IVM*） [4, 15]。物化视图预先计算并存储查询结果，而增量视图维护则在数据变化时对其进行增量更新。这些机制可以提升查询性能，但会在复杂性、存储占用以及对动态记录数据模式的适应方面引入开销 [26]。

Winter 等人 [27] 针对流数据提出了一种混合视图维护策略，将工作拆分到插入时与查询时。他们的系统既不会在插入时完全物化流元组，也不会将所有处理推迟到查询时，而是仅将每个元组处理到第一个物化点，并在请求结果时完成剩余的查询处理。该“拆分维护”方法在 Umbra 编译型查询引擎中实现，其插入吞吐量可与专用流处理器相媲美，同时保持较低的查询延迟。该方法在分析型工作负载上的表现优于 Apache Flink 和传统增量视图维护方法，同时与数据库紧密集成，以结合实时数据与历史数据。

Zhang 等人 [30] 聚焦于阿里云 AnalyticDB 的实时数据处理引擎，旨在解决 Lambda ETL架构和增量视图维护的局限性。他们的系统将增量视图维护与 Zero-ETL 相结合，以支持高吞吐量的流式ETL并保持数据一致性，从而无需使用流处理引擎。

许多研究与我们的方法互为补充，重点在于加速数据库中的模式匹配 [20, 25]。Körber 等人[13] 提出了一种索引方法，用于筛选事件存储中的查询，以执行数据预过滤。然而，他们的设计在可扩展性方面仍有限，因为它依赖顺序模式匹配而非多模式匹配，且不支持跨多个分区进行处理。此外，流式处理平面与数据库平面之间的集成程度有限。相比之下，据我们所知，我们的方案提供了更一体化的架构设计和处理模型，将流内多模式过滤与分析系统紧密结合。

8 结语

本文探讨了如何在大规模可观测性平台中，通过将昂贵的重复查询移出分析平面并纳入摄取管道来协调推式流处理与拉式分析查询。我们提出的 FluxSieve 并非仅在查询时依赖表扫描和重量级文本索引，而是嵌入高效的流内预计算层，同时仍将分析平面作为事实来源。

关键发现与启示。跨不同系统：

- 查询性能最高可提升数个数量级。
- 摄取吞吐量的开销并不高，CPU 使用开销也在可控范围内，与我们的方法带来的收益相比，这种权衡是值得的。
- 存储开销可以忽略不计。
- 这些收益适用于不同的查询类型、分析系统、配置、数据量和部署模式。

这些结果表明，对于查询密集且通常具有超高选择性的可观测性工作负载，将过滤逻辑转移到流式数据平面是一种强有力的设计模式。它减少了查询时所需的工作量，并使分析引擎能够处理以最适合检索的形态存储的数据。这些发现指向了传统全文搜索（FTS）索引之外的可行方案：昂贵且重复的查询可以依靠流内预计算，而非使用 FTS 索引；对于不太密集的查询或临时查询，则可以采用更轻量的索引策略，或在适当情况下进行全表扫描。这种分层方法能够高效分配系统资源，仅在重量级索引确实能带来收益的场景中使用，而不是将其统一应用于所有查询模式。

更广泛地说，我们的研究结果表明，“将数据库彻底反转” [12] 对于选定的高选择性查询而言，不仅在概念上具有吸引力，而且在与现代多模式匹配引擎和列式存储相结合时，在实证上也是有效的。此外，所提出的方法并不局限于可观测性：它可以更普遍地应用于其他具有重复分析查询、并能从统一架构中受益的领域，例如通用重复查询 [28] 和更新型连续查询 [3]。

局限性。 这项工作有意保持具体：**(I)** 我们针对**高选择性过滤条件**。**(II)** 我们的评估使用具有受控模式和查询模式的**日志类数据集** 。尽管它们被设计为具有代表性，但其他场景可能会呈现额外的复杂性。**(III)** 我们聚焦于**基于模式的过滤和聚合**。其他流内计算（例如窗口连接）可能会呈现不同的权衡。这些局限性并非根本性的，但它们界定了当前结果应被解释的范围。

未来工作。 我们的方法开启了几个颇具前景的方向：（I）**超越过滤：**许多工作负载都能从流内聚合中受益，研究预计算与灵活性之间的适当平衡是下一步的重要工作。（II）更深入地**集成开放表格式**：将我们的方法与开放格式集成，可以进一步提升其泛化能力。（III）**更新的模式匹配引擎**：BLARE [31] 等新方法或许能进一步降低流内过滤的 CPU 成本。

参考文献

[1] Daniel Abadi、Anastasia Ailamaki、David Andersen、Peter Bailis、Magdalena Balazinska、Philip Bernstein、Peter Boncz、Surajit Chaudhuri、Alvin Cheung、AnHai Doan 等。2020。《西雅图数据库研究报告》。*ACM Sigmod Record* 48, 4 (2020), 44–53. [2] Alfred V Aho 和 Margaret J Corasick。1975。《高效字符串匹配：书目检索的利器》。*Commun. ACM* 18, 6 (1975), 333–340. [3] Angela Bonifati 和 Riccardo Tommasini。2024。《（现代）数据系统中的连续查询概述》。载于*Companion of the2024 International Con- ference on Management of Data*（智利圣地亚哥）。美国计算机协会，纽约，纽约州，美国，605–612。https://doi.org/10.1145/3626246.3654679

[4] Mihai Budiu、Tej Chajed、Frank McSherry、Leonid Ryzhyk 和 Val Tannen。2024。《DBSP：流上的增量计算及其在数据库中的应用》。*ACM SIGMOD Record* 53, 1 (2024), 87–95. [5] Hubert Dulay 和 Ralph Matthias Debusmann。2025。流数据库。O'Reilly Media, Inc.，加利福尼亚州塞瓦斯托波尔。https://www.oreilly.com/library/view/ streaming-databases/9781098154820/ [6] Marios Fragkoulis、Paris Carbone、Vasiliki Kalavri 和 Asterios Katsifodimos。2024。《流处理系统演进综述》。*The VLDBJournal* 33, 2 (2024), 507–541. https://doi.org/10.1007/s00778-023-00819-8[7] Dimitrios Giouroukis、Dwi PA Nugroho、Varun Pandey、Steffen Zeuch 和 Volker Markl。2025。《利用数据处理单元协同处理分析近网络硬件加速》。*Proceedingsof the VLDB Endowment* 18, 13 (2025), 5689–5702.[8] Fabio Grandi、Federica Mandreoli、Riccardo Martoglia 和 Wilma Penzo。2022。《在时态数据库环境中释放流数据查询的潜力：一种关系代数方法》。*Information Systems* 103 (2022), 101872. https: //doi.org/ 10 1016 2021 101872 . /j.is. . Sören Henning、Adriano Vogel、[9] Michael Leichtfried、Otmar Ertl 和 Rick Rabiser。2024。《ShuffleBench：使用分布式流处理框架进行大规模数据洗牌操作的基准测试》。载于 *ACM/SPEC*国际性能工程会议。ACM，纽约，纽约州，美国，2–13。https: //doi.org/ 10 1145 3629526 3645036 . / . Torsten Hoefler 和 Roberto Belli。2015。[10]《并行计算系统的科学基准测试：报告性能结果时告知大众的十二种方法》。载于 *Proceedings of the international conference for high performance com- puting, networking, storage and analysis*。ACM，奥斯汀，得克萨斯州，美国，1–12。[11] Jean-François Im、Kishore Gopalakrishna、Subbu Subramaniam、Mayank Shri- vastava、Adwait Tumbde、Xiaotian Jiang、Jennifer Dai、Seunghyun Lee、Neha Pawar、Jialiang Li 等。2018。《Pinot：为5.3亿用户提供实时在线分析处理》。载于 *Proceedings of the 2018 International Conference on Management of Data*。ACM，休斯敦，得克萨斯州，美国，583–594。[12] Martin Kleppmann。2017。设计数据密集型应用：可靠、可扩展且可维护系统背后的核心理念。O'Reilly Media, Inc.，加利福尼亚州塞瓦斯托波尔。616页。[13] Michael Körber、Nikolaus Glombiewski 和 Bernhard Seeger。2021。《事件存储中的索引加速模式匹配》。载于*Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*。ACM，线上会议，中国，1023 1036 10 1145 3448016 3457245 – 。https://doi.org/ 。/ 。[14] Imene Mami 和 Zohra Bellahsene。。《视图选择方法综述》。*ACM Sigmod Record* 41, 1 (2012), 20–29. [15] Dan Olteanu。2024。《增量视图维护的新进展》。载于第 *43*届数据库系统原理研讨会伴随会议（智利圣地亚哥 AA）*(PODS '24)*。美国计算机协会，纽约，纽约州，美国， 8 17 10 1145 3635138 3654763 – 。[16] https://doi.org/ 。/ 。Apache Pinot。。《Apache Pinot——官方网站》。https://pinot.apache.org/。访问日期：2026年3月2日。[17] Steven Purtzel 和 Matthias Weidlich。2025。《SuSe：流上正则表达式子序列聚合的摘要选择》。*2025*年数据管理国际会议论文集3, 3，文章 222（2025年6月），27 10 1145 3725359 页。https://doi.org/ 。/ Mark Raasveldt、[18] Pedro Holanda、Tim Gubner 和 Hannes Mühleisen。2018。《公平基准测试并非易事：数据库性能测试中的常见陷阱》测试。载于数据库系统测试研讨会论文集。ACM，休斯顿，德克萨斯州，美国，1–6。[19] Mark Raasveldt 和 Hannes Mühleisen。2019。DuckDB：一种可嵌入式分析型数据库。载于*2019* 年国际数据管理会议论文集。ACM，阿姆斯特丹，荷兰，1981–1984。[20] Julian Reichinger、Thomas Krismayer 和 Jan Rellermeyer。2024。COPR——高效、大规模的日志存储与检索。arXiv:2402.18355 [cs.IR] https: // arxiv.org/abs/ 2402 18355 。Adriano Vogel、Dalvan Griebler、[21] Marco Danelutto 和 Luiz Gustavo Fernandes。2022。并行流处理的自适应：系统综述。*Concurrency and Computation: Practice and Experience* 34, 6 (2022)，e6759。[22] Adriano Vogel、Sören Henning、Otmar Ertl 和 Rick Rabiser。2023。分布式流处理系统性能的系统映射研究。载于*Euromicro* 软件工程与高级应用会议。IEEE，都拉斯，阿尔巴尼亚， 293 300 10 1109 60479 2023 00052 – 。[23] https://doi.org/ 。 /SEAA . 。Adriano Vogel、Sören Henning、Esteban Perez-Wohlfeil、Otmar Ertl 和 Rick Rabiser。2024。流处理框架故障恢复的全面基准测试分析。载于第 *18* 届 *ACM* 分布式与事件驱动系统国际会议论文集（法国维勒班）（*DEBS '24*）。ACM，纽约，纽约州，美国，171–182。https://doi.org/10.1145/3629104.3666040[24] Xiang Wang、Yang Hong、Harry Chang、KyoungSoo Park、Geoff Langdale、Jiayu Hu 和 Heqing Zhu。2019。Hyperscan：面向现代中央处理器的高速多模式正则表达式匹配器。载于第 *16* 届 *USENIX* 网络系统设计与实现研讨会论文集（*NSDI '19*）。USENIX 协会，波士顿，马萨诸塞州，美国，631–648。https://www.usenix.org/system/files/nsdi19-wang-xiang.pdf [25] Ziheng Wang、Junyu Wei、Alex Aiken、Guangyan Zhang、Jacob O Tørring、Rain Jiang、Chenyu Jiang 和 Wei Xu。2025。LogCloud：在对象存储上快速搜索压缩日志。*Proceedings of the VLDB Endowment* 18, 8 (2025)，2362–2370。[26] Wolfram Wingerath、Felix Gessert 和 Norbert Ritter。2020。InvaliDB：在拉取式数据库之上实现可扩展的推送式实时查询。载于*2020* 年 *IEEE* 第 *36* 届数据工程国际会议（*ICDE*）。IEEE，皮斯卡特维，新泽西州，美国，1874–1877。[27] Christian Winter、Tobias Schmidt、Thomas Neumann 和 Alfons Kemper。2020。各让一步：连续视图的拆分维护。*Proceedings of the VLDB Endowment* 13, 11 (2020)，2620–2633。https://doi.org/10.14778/3407790. 3407849[28] Feng Yu、Wen-Chi Hou、Cheng Luo、Qiang Zhu 和 Dunren Che。2011。数据库中重复查询的连接选择率重新估算。载于第 *22* 届数据库与专家系统应用国际会议论文集（法国图卢兹）（*DEXA'11*）。施普林格，柏林，420–427。[29] Matei Zaharia、Ali Ghodsi、Reynold Xin 和 Michael Armbrust。2021。湖仓：统一数据仓库与高级分析的新一代开放平台。载于第 *11* 届创新数据系统研究会议，*CIDR 2021*，线上会议，*2021* 年 *1* 月 *11–15* 日，线上论文集。www.cidrdb.org，线上会议，1–8。[30] Fangyuan Zhang、Mengqi Wu、Chunlei Xu、Yunong Bao、Jiyu Qiao、Yingli Zhou、Hua Fan、Caihua Yin、Wenchao Zhou 和 Feifei Li。2025。流式视图：面向阿里云现代实时数据仓库的高效数据处理引擎。*Proceedingsof the VLDB Endowment* 18, 12 (2025)，5153–5165。[31] Ling Zhang、Shaleen Deep、Avrilia Floratou、Anja Gruenheid、Jignesh M Patel 和 Yiwen Zhu。2023。利用正则表达式查询中的结构。*ACM* 数据管理论文集1, 2 (2023)，1–28。