LPStream: Fine-grained Lazy Provenance for Stream Processing

MASAYA YAMADA*, University of Tsukuba, Japan and National Institute of Advanced Industrial Science and Technology, Japan
HIROYUKI KITAGAWA, University of Tsukuba, Japan and National Institute of Advanced Industrial Science and Technology, Japan
SALMAN AHMED SHAIKH, National Institute of Advanced Industrial Science and Technology, Japan
TOSHIYUKI AMAGASA, University of Tsukuba, Japan
AKIYOSHI MATONO, National Institute of Advanced Industrial Science and Technology, Japan

Stream processing enables real-time data analysis. Recent stream processing engines (SPEs) execute stream processing in a distributed manner for real-time analysis of massive amounts of data produced by IoT devices and sensors. It has been widely adopted in various applications that support critical decision making. To explain the results of stream processing, ensuring provenance is indispensable. Provenance clarifies the relationship between input data and output data in the processing. With provenance, we can understand what input data contributed to the output. Existing frameworks for providing provenance for stream processing generate provenance or additional information to construct provenance at runtime. However, these approaches impose substantial overhead in ordinary stream processing. In this paper, we propose a new framework, named LPStream, for fine-grained lazy provenance. LPStream is the first framework to support lazy provenance for stream processing. In the ordinary execution mode, LPStream executes stream processing with checkpointing but without provenance generation. If provenance is necessary for some target output tuples, it replays the processing from an appropriate checkpoint and generates the provenance for the target tuple. We explain the design and implementation of LPStream and evaluate its performance by comparing LPStream with stream processing without provenance and with eager provenance. The experimental results demonstrate the effectiveness of our proposal.

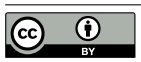
CCS Concepts: • **Information systems** → **Data provenance**; **Stream management**.

Additional Key Words and Phrases: Lazy Provenance, Stream processing, Checkpointing, Apache Flink

ACM Reference Format:
Masaya Yamada, Hiroyuki Kitagawa, Salman Ahmed Shaikh, Toshiyuki Amagasa, and Akiyoshi Matono. 2025. LPStream: Fine-grained Lazy Provenance for Stream Processing. *Proc. ACM Manag. Data* 3, 4 (SIGMOD), Article 254 (September 2025), 25 pages. https://doi.org/10.1145/3749172

*He is currently affiliated with LY Corporation.

Authors' Contact Information: Masaya Yamada, University of Tsukuba, Tsukuba, Ibaraki, Japan and National Institute of Advanced Industrial Science and Technology, Koto, Tokyo, Japan, yamada@kde.cs.tsukuba.ac.jp; Hiroyuki Kitagawa, University of Tsukuba, Tsukuba, Ibaraki, Japan and National Institute of Advanced Industrial Science and Technology, Koto, Tokyo, Japan, kitagawa@cs.tsukuba.ac.jp; Salman Ahmed Shaikh, National Institute of Advanced Industrial Science and Technology, Koto, Tokyo, Japan, shaikh.salman@aist.go.jp; Toshiyuki Amagasa, University of Tsukuba, Tsukuba, Ibaraki, Japan, amagasa@cs.tsukuba.ac.jp; Akiyoshi Matono, National Institute of Advanced Industrial Science and Technology, Koto, Tokyo, Japan, a.matono@aist.go.jp.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2836-6573/2025/9-ART254
https://doi.org/10.1145/3749172

1 Introduction

Stream processing is a modern data processing paradigm that supports real-time data analysis. It is usually executed on stream processing engines (SPEs). Today, a huge number of IoT devices and sensors are continuously producing massive amounts of data as data streams, and many of the applications demand their real-time processing. Apache Flink [2], Spark Streaming [6], Structured Streaming [7], and Kafka Streams [5] are well-known modern SPEs, and they support distributed execution of stream processing. These frameworks enable scalable and high-performance (i.e., low-latency, high-throughput) processing even when the workloads are extremely heavy. SPEs have been adopted in various applications to support online monitoring and critical decision making (e.g., anomaly detection and countermeasures). In such applications, obtaining real-time query results is essential but often insufficient. When alarms or unpredicted results are produced, additional information is necessary to investigate the situation to determine, for example, the origin of the results and how they are computed.

Data provenance (*provenance*) refers to information explaining the data generation process, and it has been widely researched in several data engineering fields [10, 11, 16, 21]. In particular, fine-grained provenance clarifies the relationship between output data items and their input items [28, 38, 43]; it is also called *data lineage*. Fine-grained provenance provides clues for understanding why each output item is produced through processing and is beneficial for elucidating the details of the situations that cause the obtained query results.

For database queries, numerous provenance frameworks have been researched [13–15, 18, 28, 40, 43]. Fine-grained provenance in database queries is usually generated at the tuple level. The existing approaches can be categorized into one of two types — *eager* or *lazy* — depending on the timing of provenance generation. Eager provenance refers to frameworks in which provenance is generated during the query processing [13, 18, 28, 43]. Therefore, provenance is ready for all query result tuples when they are generated. However, provenance computation usually imposes substantial overhead on query processing. Lazy provenance refers to provenance computed independently of the query processing [14, 15, 40]. In lazy provenance, query processing is conducted in an ordinary manner. After the query processing, once target tuples in the query results are specified, provenance for the target tuples is lazily generated. This process imposes little overhead on ordinary query processing. However, sophisticated provenance generation schemes are often required to realize lazy provenance. As discussed, eager and lazy provenance schemes have distinctive characteristics, which enables users to choose the most suitable scheme depending on the specific features of the intended application.

Some frameworks ([9, 17, 19, 20, 25, 29, 35–37, 41]) target provenance for stream processing. Among existing proposals, GeneaLog [25] is the state-of-the-art fine-grained provenance framework for modern SPEs and is based on eager provenance. We were unable to identify any proposals related to lazy provenance for stream processing. According to our analysis presented in Section 6, GeneaLog can degrade the processing performance by 93% (throughput) and 37% (latency) in the worst case. In addition, the size of all output tuples is drastically increased by the attached provenance data, which imposes overhead on application systems to process the output tuples. These overheads can demand additional computational resources to satisfy certain performance requirements. If provenance is required for most of the output tuples, the overheads may be acceptable. Otherwise, the extra overhead and cost are not tolerable. In such cases, lazy provenance will be the better choice.

In this paper, we propose a fine-grained lazy provenance framework, named LPStream, for stream processing. This work is the first proposal for fine-grained lazy provenance for streams. The conceptual architecture is shown in Figure 1. To support lazy provenance, the framework

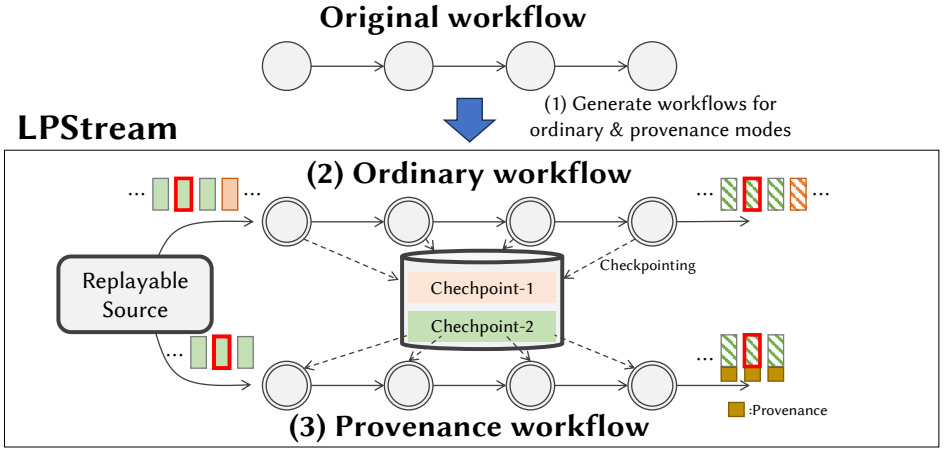


Fig. 1. Conceptual Architecture of LPStream.

uses *replayable stream sources* and *checkpointing*, which modern SPEs generally provide for their fault tolerance. LPStream consists of two modes: *ordinary* (no provenance) *mode* and *provenance mode*. Given the original workflow implemented by analysts, LPStream generates workflows for both modes ((1) in Figure 1). In the ordinary mode ((2) in Figure 1), LPStream executes stream processing workflow in an almost ordinary manner with checkpointing (*ordinary workflow*). When users require provenance of some output tuples (highlighted by a red square), they direct LPStream to enter provenance mode. In the provenance mode, LPStream starts another background job (*provenance workflow*) to generate provenance of the specified tuples in addition to the ordinary stream workflow processing ((3) in Figure 1). The newly started job replays the original workflow with provenance from an appropriate checkpoint stored during the ordinary mode. This provenance workflow generates provenance of the specified output tuples in a lazy manner.

Even with a replayable source and a checkpointing mechanism in place, generating provenance by replaying the workflow is not straightforward. Several technical challenges arise. (1) Ordinary and provenance workflows should be automatically generated from the original workflow. However, complex semantics and context information involved in workflows may make this non-trivial. (2) To start the provenance workflow from a checkpoint of the ordinary workflow, it is mandatory to use the same data classes/types in both workflows for the snapshot compatibility. However, since the two workflows perform different processing, this poses a challenge. (3) Stream processing inherently involves non-determinism in its semantics (e.g., the order of tuples after shuffling). Under such potentially non-deterministic processing, generating a tuple that correctly contains the provenance of a specified tuple from the replayed provenance workflow is challenging. (4) The execution of the provenance workflow requires identifying the correct checkpoint for replay. Flink does not natively provide a means to identify the checkpoint corresponding to a given output. This work addresses these challenges. In summary, our work makes the following contributions:

- **Proposal of lazy provenance framework**: We propose a novel framework (*LPStream*) that derives lazy provenance for streams. LPStream is based on the aforementioned key ideas, and it generates provenance only when target tuples that need provenance are specified. Because provenance is generated in a background job, it significantly reduces the performance overhead of

ordinary stream workflow processing compared to eager provenance. We present the architecture of LPStream and explain how LPStream generates lazy provenance.
- **Implementation of LPStream over Flink and GeneaLog**: To demonstrate the feasibility of the proposed framework, we have implemented a prototype system of LPStream over Flink and GeneaLog. It consists of approximately 10,000 lines of Java program code. The code for LPStream is available [31].
- **Experiments with diverse datasets, queries, and metrics**: We performed intensive experiments to evaluate the performance of LPStream using the prototype system. The experiments consist of 16 workflows with different properties (i.e., different operators, window sizes, and input tuple sizes) using four benchmark and real datasets and three synthetic datasets. We measured three metrics in the experiments (i.e., sustainable throughput, latency, and duration for provenance generation). The experimental results demonstrate that LPStream imposes much smaller overhead for ordinary workflow execution compared with the eager provenance framework.

The rest of this paper is organized as follows. We overview provenance frameworks for traditional database queries and stream processing in Section 2. Section 3 introduces data model, checkpointing, and provenance for stream processing as the preliminary. Section 4 describes core ideas to lazily generate provenance by replaying from the checkpoint. We explain our implementation of LPStream in Section 5. Section 6 explains experimental settings and the results of our intensive experiments. We conclude this paper in Section 7.

2 Related Work

Provenance has been researched in numerous domains [21], including databases [10, 11], big data processing [1, 23], scientific workflows [16], and stream processing [19, 25]. We focus on database query and stream processing domains and briefly review existing approaches.

2.1 Provenance for Database Systems

Many frameworks have been proposed for database queries [10, 11, 21], and they focus on fine-grained eager or lazy provenance [13–15, 18, 28, 40, 43]. [13, 18, 28, 43] target eager provenance. [13, 43] assign identifiers to all input tuples/values and query results inherit them. In [18], output data piggybacks input data to maintain provenance when a query is executed. Smoke [28] records relationships between input and output data at each operator in the query execution. Provenance of an output tuple is obtained by traversing the relationships backward.

[14, 15, 40] target lazy provenance. [15] focused on relational queries and proposed tracing queries to identify input tuples that contribute to output tuples. Tracing queries enable us to derive lazy provenance only for the target tuples after query execution. [40] proposed *augmented lineage* to enrich provenance for analyses including artificial intelligence and machine learning (AI/ML) processing. Augmented lineage integrates information about why the outputs are generated by the AI/ML models with the conventional lineage that presents input tuples contributing to the outputs. It formulated an algorithm to derive augmented lineage in a lazy manner extending the tracing queries in [15]. [14] enables lazy provenance in more general data transformations. Their algorithm covers data transformations beyond relational queries and proposes a general inversion procedure to identify source data on the basis of the properties of operators in the workflow (e.g., dependency on input/output items and schema mapping).

2.2 Provenance for Streaming Processing

Eager provenance for stream processing has been researched [9, 17, 19, 20, 25, 35, 36, 41]. [35, 36] target coarse-grained provenance for stream processing. [17] supports fine-grained provenance by

recording dependencies between input data and output data in each execution unit (e.g., operator) during the ordinary query execution. Ariadne [19] assigns an identifier to each tuple and appends identifiers of the contributing tuples to output tuples as annotations; the annotations are directly used to derive provenance of the output tuples. [9] also uses similar annotations for provenance.

GeneaLog [25] employs pointers to the contributing tuples, which makes additional data concise and a fixed size. It is the state-of-the-art framework for eager provenance, and our framework refers to GeneaLog as a base component. GeneaLog deploys several objects and methods dedicated to eager provenance over original workflows. Therefore, additional fragments of program code need to be included before the original program code runs on GeneaLog. In the *Native* approach, users need to inspect and directly modify the original program codes to embed additional processing for provenance in user-defined data objects and operators. By contrast, the *Wrapping* approach provides wrapper functions to extend the original methods to generate eager provenance[1]. Applying the wrapper functions is an easier way to modify the original program code when compared with the Native approach. However, the program code obtained by the Wrapping approach generally has room for optimization, such as with respect to the number of method invocations. Therefore, the Native approach provides more chances to obtain program codes that run more efficiently. We describe additional details of GeneaLog in Subsection 3.3. The authors of GeneaLog proposed Ananke [26] and Erebus [27], which target different types of provenance. Ananke identifies output tuples to which specified input tuples contributed, known as forward provenance. Erebus is the first framework targeting why-not provenance in the streaming context.

Twins [20] reduces overhead for ordinary processing of GeneaLog by activating/deactivating provenance generation depending on user-defined rules. Once Twins produces output tuples without provenance, it cannot reconstruct their provenance even if users discover unexpected results after processing. In addition, it assumes that window processing is the only stateful operator, so it cannot handle general stateful operations (e.g., rolling aggregation). s2p [41] provides online and offline provenances. The former is fine-grained eager provenance generated for all output data. The latter is generated after the online provenance processing for output tuples that the user specifies. Offline provenance provides more detailed information (e.g., intermediate data, state values, and their transitions at each operator) in addition to fine-grained provenance. Snapshots of checkpoints in the online provenance processing are used to initiate the offline provenance processing. The basic scheme of s2p is eager provenance, and some technical details, such as how to include state information in provenance, are unclear.

A number of proposed approaches [19, 22, 29, 37] store additional data for provenance during the ordinary processing and use the stored data to generate provenance later. [22] proposed the inference approach to derive provenance using timestamps attached to tuples after processing. In [37], developers should define dependency rules to derive input data from output data for each operator, and provenance is generated by applying the rules. [29] proposed stream ancestor functions that work like the above dependency rules for the normal operators; however, in their frameworks, all intermediate query tuples or keys (identifiers) need to be stored, which is not feasible or imposes substantial overhead. Ariadne [19] also provides Replay-Lazy and Lazy-Retrieval modes. In both modes, Ariadne eagerly computes coarse provenance for all output data in the ordinary processing. The coarse provenance designates the range of identifiers of input tuples that contain real provenance tuples. Fine-grained provenance is derived by replaying input tuples within the target range.

[1]Original Genealog [25] does not include wrapping approach, but it is provided as a part of Ananke [26], which is another framework by the same authors. In Ananke, Native and Wrapping approach are called non-transparent and transparent implementation, respectively.

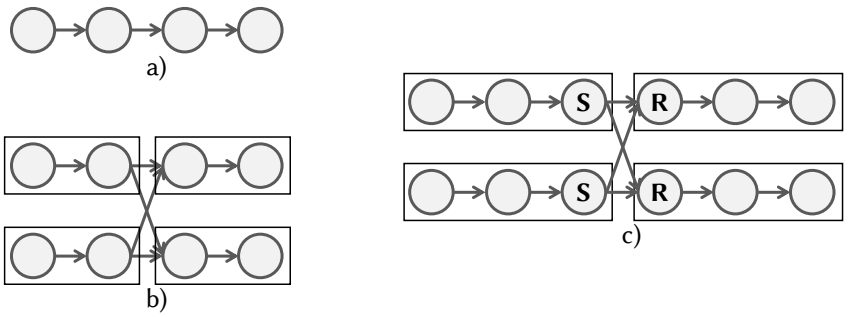


Fig. 2. Workflow representation: a) Logical representaiton, b) physical representation, and c) the Send(S)/Receive(R) operators.

The proposed LPStream enables fine-grained lazy provenance with substantially lower overhead in the ordinary stream processing. It only requires periodic checkpointing, without the additional overhead imposed in the existing approaches.

3 Data Model, Checkpointing, and Data Provenance

3.1 Data Model

In this section, we define a streaming data model employed in this paper. The model follows Flink data model. *Data stream (Stream)* S is a sequence of tuples. *Tuple* t consists of its timestamp ts and attribute values a_i ($t = \langle ts, a_0, \ldots, a_m \rangle$). In this paper, we consider the timestamp as an *event time*, which represents the time when the event occurred in the real world; $t.ts$ and $t.a_i$ denote the timestamp and an attribute value of the tuple t, respectively.

Workflow W is defined as a directed acyclic graph (DAG) structure whose nodes represent operators and whose edges are dataflow channels between operators. There are two ways to represent a workflow. *Logical representation* shows operators and channels in the workflow at the logical level (Figure 2.a). This enables us to understand what analysis the workflow will perform. In the execution of workflows, parallelism is employed and each operator is executed by multiple operator instances. *Physical representation* shows workflows at the physical level and expands each logical operator to operator instances distributed on the physical cluster (Figure 2.b). This representation explains how operators in the workflow are deployed on physical nodes and how the workflow runs on them. As shown in Figure 2.b, two types of channels — one-to-one and shuffling channels — appear at the physical level. In Figure 2, each rectangle represents a *task*, which is a sequence of operator instances connected by one-to-one channels. A task corresponds to a thread and executes operator instance(s) on SPEs.

We consider four types of operators: 1) Source/Sink, 2) Transformer, 3) Send/Receive, and 4) WatermarkGen. The first three follow the existing work on provenance over streams [25], and we explicitly include WatermarkGen for event time based processing. A *Source* operator is placed at the beginning of the workflow. It ingests a data stream into a workflow from external sources (e.g., networks, Apache Kafka [4], or files). *Sink* is placed at the end of a workflow. It sends an output stream to external systems (e.g., user programs or Apache Kafka).

Transformer receives one or two input stream(s) and produces one output stream. Each output tuple has its own *contributing tuples*. We consider the four transformation operators.

Filter: Given a tuple t_i, Filter sends it to the succeeding operator if the tuple meets filtering condition c. Otherwise, the tuple is discarded. Filter simply sends or discards the input tuple; the

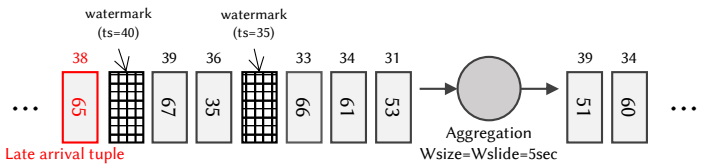


Fig. 3. Late-arrival tuple.

output tuple therefore has the same timestamp as the input. In Filter, the contributing tuple of the output is t_i.

Map: Given a tuple t_i, Map applies user-defined function (UDF) f_m for the tuple and produces one or more tuples $t_o^j, 1 \leq j \leq n$. It assigns the same timestamp to output tuples as the input ($t_o^j.ts = t_i.ts$). In Map, the contributing tuple of t_o^j is t_i.

Aggregation: Given a tuple t_i, Aggregation dispatches it to *windows* depending on its key and timestamp. Because each window has window size $Wsize$ and window slide $Wslide$, tuples are sent to windows whose time intervals contain the tuple's timestamp. When a window is triggered, tuples in the window are aggregated by user-defined aggregate function f_a. Aggregation assigns the end time of the window to its output. For example, an output is produced from a window of size 50 whose time interval is [100,150], the output timestamp becomes 150. In Aggregation, all aggregated tuples are contributing tuples of the output.

Join: Given two input streams, Join dispatches incoming tuples to windows depending on their keys and timestamps. Like Aggregation, Join sends tuples to windows whose time intervals contain their timestamps. When a window is triggered, Join applies join function f_j for all tuple pairs in the window. Join assigns the end time of the window to its output. In Join, the two joined tuples are contributing tuples of the output.

Note that transformer may have its state, update it in the operation, and produce output tuple(s) referencing it. Aggregation and Join maintain built-in states for window-based processing. Users may define additional states to address their data processing requirements.

When stream processing is executed in a distributed environment, tuples flowing in the workflow should be transferred between different tasks. *Send* is placed at the end of a task. It serializes tuples and sends them to the succeeding task. *Receive* is placed at the beginning of the succeeding task; it deserializes tuples sent from the Send operator and ingests them into the task. A sample workflow after the addition of Send/Receive is shown in Figure 2.c.

WatermarkGen inserts *watermarks* into a stream. Watermark ws has a timestamp ($ws.ts$), and it determines the progress of time inside the system. Watermark ws suggests that the current time inside the system is $ws.ts$ and that no new tuple whose timestamp is smaller than or equal to $ws.ts$ is expected after the watermark. In general, stream processing, especially window-based processing such as Aggregation and Join, uses watermarks to decide which windows should be triggered.

Let us consider the example shown in Figure 3. In this example, the Aggregation operator computes a 5s average over an input stream and timestamps of input tuples are shown above them. Usually, incoming tuples have increasing timestamps because subsequent events are notified later. However, some exceptions can occur because of communication delays and other factors. In fact, the timestamps of the tuples "66" and "65" are smaller than their preceding tuples. Watermarks explicitly indicate the progress of time. The watermark with timestamp 35 notifies that the current time has reached 35. Therefore, window processing that should be triggered before timestamp 35 should be invoked upon the arrival of this watermark. This example assumes that the next window processing covers the time interval $[30, 35]$. Therefore, when the watermark with timestamp 35

reaches the Aggregation operator, it calculates the aggregation. Note that the tuple "65" with timestamp 38 arrives after the watermark with timestamp 40. Therefore, during processing, the tuple "65" is ignored as a *late-arrival tuple*.

WatermarkGen is placed at the beginning of the processing, monitors timestamps of tuples in an input stream, and composes and inserts watermarks into the stream. WatermarkGen decides when it inserts watermarks and what timestamp is assigned to each watermark.

3.2 Checkpointing

Checkpointing provides fault tolerance for stream workflow execution on SPEs. It usually takes snapshots of all states in the workflow periodically and stores them in a file system, HDFS [3], or RocksDB [30]. Each invocation of checkpointing has a *checkpoint ID*. During checkpointing, *checkpoint barriers* flow from Sources through the workflow. When an operator receives relevant checkpoint barriers, it takes a snapshot of its state and forwards barriers downstream. When the barriers reach the Sinks, the checkpointing is completed. Each operator can be notified of the completion of checkpointing with its ID. When a failure occurs, SPEs restart the workflow from the latest checkpoint. With checkpointing, SPEs can restart the workflow from a specific consistent execution point.

3.3 Data Provenance

Given a target output tuple of a workflow, its provenance is intuitively the set of input tuples that contribute to the derivation of the target. Provenance is defined more formally as follows.

Definition 3.1 (Provenance wrt. an operator)**.** For an output tuple of an operator, its *provenance with respect to the operator* is the set of contributing tuples defined in Subsection 3.1. The provenance of a set of output tuples wrt. an operator is the union of their provenance.

Definition 3.2 (Provenance)**.** A workflow consists of multiple operators. The *provenance* of an output tuple of the workflow is the set of input tuples, transitively obtained by applying Definition 3.1 to each operator from Sink to Source.

We call input tuples contained in the provenance *source tuples*.

Example 3.1. Figure 4 shows a workflow for analyzing a temperature data stream produced by sensors on machines in a factory. There are multiple machines in the factory, and each machine has several sensors that periodically monitor the machine's current temperature and power consumption. The workflow computes the average temperature for the machines every ten seconds. The Source operator ingests raw input sensor reading objects into the workflow. The Map operator parses each input object and transforms it into a tuple. Note that attribute "Type" indicates whether the tuple is for temperature (Type = 0) or for power consumption (Type = 1), and attribute "Ts" represents the event timestamp, which represents the time of each sensor reading at second-level granularity. The Filter operator filters out tuples for power consumption data. The Aggregation operator then computes the average temperature value for each machine every ten seconds using all the sensor readings within the recent 10s window. The Sink operator sends the produced stream to succeeding applications. In this example, shaded tuples in Figure 4 contribute to the output tuple $\langle 0, 570.4, 1010\rangle$. Therefore, the set of tuples at Source ($\{\langle 0, 0, 0, 252.8, \tilde{}, 1002\rangle, \langle 0, 0, 0, 888.0, \tilde{}, 1007\rangle\}$) constitutes the provenance of the output tuple $\langle 0, 570.4, 1010\rangle$.

Generating provenance means finding source tuples, which identifies the relationship between outputs and inputs. Through the provenance, we can understand why each output was derived from the workflow. Our proposed framework uses an eager provenance framework for stream processing as its component.

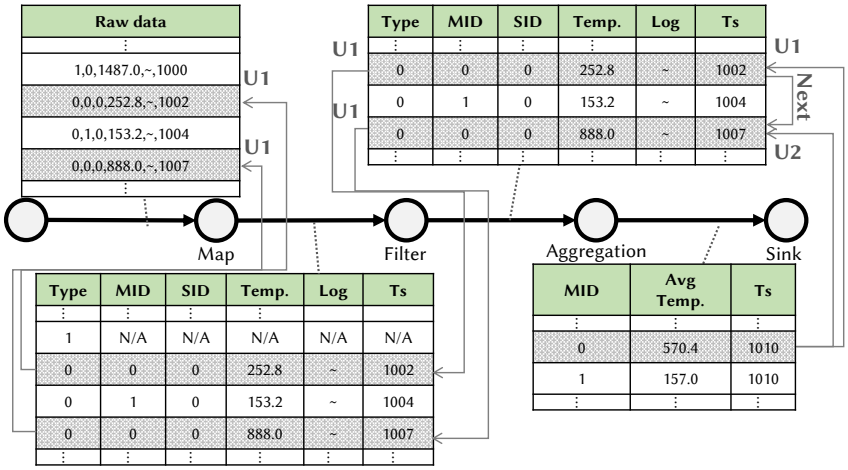


Fig. 4. Example workflow. Type, MID, and SID denote the data type (temperature data or power consumption data), MachineID, and SensorID, respectively.

GeneaLog [25] is the state-of-the-art eager provenance framework. We overview its design consisting of four key ideas: provenance metadata, operator instrumentation, contribution graph traversal, and provenance derivation in a multiple-task environment.

Provenance metadata: Each tuple in the workflow is extended with the four types of provenance metadata: $Type$, $U1$, $U2$, and $Next$. $Type$ represents the operator where the tuple was generated. $U1$, $U2$, and $Next$ are pointers to identify input tuples that contribute to the tuple.

Operator instrumentation: Some transformer operators in the workflow are instrumented to set provenance metadata.

- **Filter**: Filter just passes an input tuple to a succeeding operator; therefore, there is no instrumentation.
- **Map**: Given an input tuple t_i, Map produces new tuples $t_o^j (1 \leq j \leq n)$ with UDF f_m. Before returning them, it sets $Type$ to "Map" and $U1$ to the input tuple ($t_o^j.Type$ = "Map" and $t_o^j.U1 = t_i$).
- **Aggregation**: When a window fires, Aggregation produces an output tuple t_o with aggregate function f_a from $t_i^j (1 \leq j \leq n)$. It sets $Type$ to "Aggregation" and the three pointers: $t_o.U2 = t_i^n, t_o.U1 = t_i^1, t_i^j.Next = t_i^{j+1} (1 \leq j \leq n-1)$.
- **Join**: Given two tuples (t_i^1 and t_i^2), Join produces an output t_o with join function f_j. Join sets Type to "Join" and sets $U1$ and $U2$ pointers: $t_o.U1 = t_i^1$ and $t_o.U2 = t_i^2$.

Contribution graph traversal: After the instrumentation, output tuples of the workflow have pointers to their relevant input tuples to the last operator. The input tuples also have pointers to the input tuples to the previous operator. These recursively structured pointers construct a graph structure (Figure 4), and the graph is called a *contribution graph*. The provenance of a target tuple can be obtained by traversing the contribution graph from the target tuple to the source tuples.

Let us derive the provenance of the tuple $\langle 0, 570.4, 1010 \rangle$ in Figure 4. The tuple is the result of the Aggregation operator, and its input tuples can be obtained through $U1$, $U2$, and $Next$ pointers ($\langle 0, 0, 0, 252.8, \tilde{}, 1002 \rangle$, $\langle 0, 0, 0, 888.0, \tilde{}, 1007 \rangle$). Through $U1$ pointers of these tuples, we can reach source tuples ({"0, 0, 0, 252.8, ˜, 1002", "0, 0, 0, 888.0, ˜, 1007"}). These tuples are provenance of $\langle 0, 570.4, 1010 \rangle$.

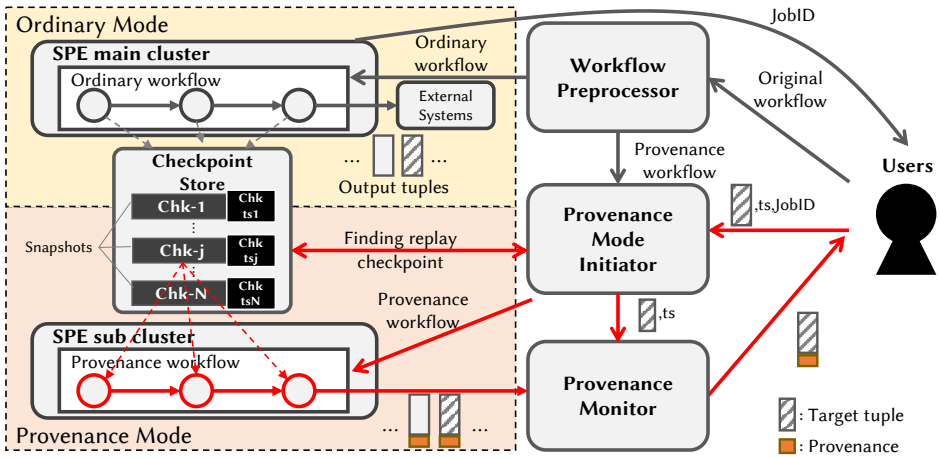


Fig. 5. Framework architecture.

Provenance derivation in a multiple-task environment: Pointers are used to locate source tuples. In a multiple-task environment, tuples are transferred among them and may be processed on different processes. In this case, we cannot use pointers to maintain input/output relationships. To resolve this problem, GeneaLog uses two approaches: an explicit approach and an implicit approach. The latter is simpler and better suited to distributed stream processing environments. For this reason, we focus on the implicit approach. In the implicit approach, Send and Receive operators are instrumented as follows:

- **Send**: Before sending a tuple t, Send generates its provenance $p_t = \{s_1, \ldots, s_n\}$, where $s_1, \ldots, s_n$ are source tuples of t, by traversing the contribution graph at this point. It then serializes the tuple and provenance together and sends them to the succeeding operator.
- **Receive**: Receiving data from Send, the Receive operator deserializes the data to a tuple t and its provenance $\{s_1, \ldots, s_n\}$. First, Receive sets t.Type to "Remote". It then sets $t.U1 = s_1$, $t.U2 = s_n$, and $s_i.Next = s_{i+1} (1 \leq i \leq n-1)$. Finally, Receive sends the tuple t to the succeeding operator.

Note that the Send and Receive operators are embedded in data deserialization and serialization on Flink, respectively.

4 Proposed Framework

4.1 Architecture

This section describes LPStream architecture. Figure 5 shows the architecture of LPStream. To derive lazy provenance, LPStream provides two execution modes: *ordinary (no provenance) mode* and *provenance mode*. LPStream first receives an original workflow from users. Then, Workflow Preprocessor translates it into two workflows: *ordinary workflow* and *provenance workflow*. The execution of the former processes streams in an almost ordinary manner without provenance in an SPE cluster, whereas the execution of the latter processes the same workflow with provenance in another SPE cluster. The detail of Workflow Preprocessor is explained in Subsection 4.3.

LPStream starts the ordinary execution mode running the ordinary workflow as a foreground job in the main cluster. At the same time, SPE main cluster returns JobID of the job to users. In the ordinary mode, checkpointing is performed periodically, and the checkpoint data including

the state snapshots are stored in Checkpoint Store with the *chk timestamp*. The assignment of chk timestamp is explained in Subsections 4.3 and 4.4.

The users monitor output tuples of the ordinary workflow. If they want to get the provenance of a tuple, called *target tuple*[2], and they direct LPStream to move to the provenance mode as follows.

(1) The user decides a target tuple for which provenance is required and then passes a) the tuple (attribute values), b) its timestamp, and c) JobID to Provenance Mode Initiator.
(2) Provenance Mode Initiator searches for the *replay checkpoint* in Checkpoint Store. The replay checkpoint is the checkpoint from which the provenance workflow starts to reproduce the target tuple with provenance. The procedure to decide the replay checkpoint will be explained in Subsection 4.5.
(3) Provenance Mode Initiator sets up the execution environment for the provenance workflow and restores operator states as those at the replay checkpoint. It then starts Provenance Monitor, which monitors output streams with provenance from the provenance workflow. Then, Provenance Mode Initiator starts the provenance workflow as a background job which replays the ordinary workflow. Note that the main cluster keeps running the ordinary workflow during the provenance mode.
(4) Provenance Monitor monitors the output stream from the provenance workflow. The target tuple is identified by its timestamp and attribute values. When it finds the target tuple, it extracts the tuple and its provenance and returns them to the user. Finally, Provenance Monitor stops the execution of the provenance workflow[3], and the system returns to the ordinary mode.

4.2 Assumptions

We make two assumptions for input streams and original workflows. The first assumption is basically for simplicity of explanations. We assume that each Source operator in the workflow is directly followed by Map and WatermarkGen. The Source ingests a stream, the Map parses it into a sequence of tuples, and the WatermarkGen periodically generates watermarks. The Filter may appear between the Map and WatermarkGen. This operator sequence can cover many real workflows, and it poses no practical constraint.

Next, we explain the second assumption, consisting of four sub-assumptions. To obtain the provenance of the target tuple by replaying the workflow, the target tuple must be found in the provenance workflow with its provenance. However, stream processing is known to involve several factors that make the workflow non-deterministic in the sense that the same workflow can generate different output even when the input streams are the same. One important factor of them is shuffling channels. The order of tuples output from the shuffling channel is generally non-deterministic. Then, stateful operation may produce different output attribute values depending on the order of input tuples. Computing a cumulative sum is an example of the operation. We call such attributes as *order-sensitive attributes* and others as *order-insensitive attributes*. The sub-assumptions are as follows:

(1) Input streams are ingested from a replayable system.
(2) There is no late-arrival tuple.
(3) Each output tuple of the workflow is identified by its *order-insensitive attributes* and timestamp.
(4) Each operator produces output tuples solely on the basis of the input tuple(s) and its state.

[2]For simplicity of explanation, we assume that one tuple in the output streams is specified as a target tuple throughout the paper. However, the proposed framework can easily deal with cases where multiple tuples are specified as target tuples.
[3]The execution environment of the provenance workflow is maintained in the ordinary mode, to prepare for the next initiation of the provenance mode.

These sub-assumptions are not so restrictive in many stream processing contexts. (1) is often a basic requirement for recovery from failures and is satisfied by deploying message-passing systems such as Kafka for data feed[4]. (2) can be satisfied if the timestamps (event times) of tuples in input streams ingested by Source operators are monotonically increasing. However, even if this condition is not met, it can still hold if watermarks are inserted in a manner that prevents the generation of late-arrival tuples[5]. The sub-assumption (3) means that each output tuple is identified by a combination of some order-insensitive attributes (e.g., sensor id, object id) and timestamp, which usually holds in common stream applications. (4) means that each operator generates its output not referencing resources such as external databases, clocks, or generators. These sub-assumptions ensure that the provenance workflow reproduces tuples which exactly correspond to the output tuples in the ordinary workflow[6].

4.3 Workflow Preprocessor

Workflow Preprocessor generates two modified versions (programs) of the workflow from a given original workflow: ordinary workflow and provenance workflow. We explain necessity of the two workflows and their generation. The provenance workflow is obviously needed since it has to execute the original workflow with eager provenance. The ordinary workflow is also necessary for two reasons. The first reason involves using the same types/classes in both the ordinary and provenance workflows. Recent eager provenance systems, such as Ariadne [19] and GeneaLog [25], require original data tuples to be extended with additional fields and original operators to be modified. As a result, stream tuples and operators must use data types/classes specifically designed for provenance generation. In contrast, tuples and operators in the original workflow have no such extensions, meaning that the provenance workflow uses different data types/classes from those in the original workflow. If types/classes are different, snapshots of states in the original workflow and the provenance workflow are inconsistent. Therefore, the provenance workflow cannot restart from snapshots taken by checkpointing the original workflow. The ordinary workflow processes streams as the original workflow but its internal types/classes are compatible with the provenance workflow. It makes any checkpoint taken for the ordinary workflow available for the provenance workflow. The second reason is to assign chk timestamp to each checkpoint of the ordinary workflow. For this purpose, the preprocessing inserts a new operator ChkTsAssigner into the ordinary workflows just before WatermarkGen. The detail is explained in Subsection 4.4. Therefore, LPStream generates the ordinary workflow as well.

4.4 Chk Timestamp Assignment with ChkTsAssigner

This section describes ChkTsAssigner in the ordinary workflow. It assigns chk timestamp to each checkpoint. The chk timestamp ts of a checkpoint represents the upper bound of the timestamps of tuples that have influenced the snapshot of operator states captured by the checkpoint. Specifically, the snapshot of the checkpoint with ts reflects information obtained by tuples up to ts. In addition, restarting from the checkpoint with ts will replay all input tuples with timestamps greater than ts. To assign such a timestamp, the ChkTsAssigner in the ordinary workflow has a state (LatestTs) and updates the state with the current tuple timestamp. ChkTsAssigner can be notified of the completion of each checkpoint with its checkpoint ID (ChkID) by SPEs. Once ChkTsAssigner

[4]Instead of Kafka, file logs can be used as long as the tuples in the file are ordered by timestamps and the Source is properly implemented to manage offsets, ensuring that streams are replayed from the correct file position.
[5]We could relax this sub-assumption by introducing an alternative sub-assumption that the set of late-arrival tuples during the replay is guaranteed to be identical to that of the ordinary processing. It can be realized by implementing a new source that reproduces the same set of late-arriving tuples during replay, depending on the properties of the intended application.
[6]They have same values except for order-sensitive attributes.

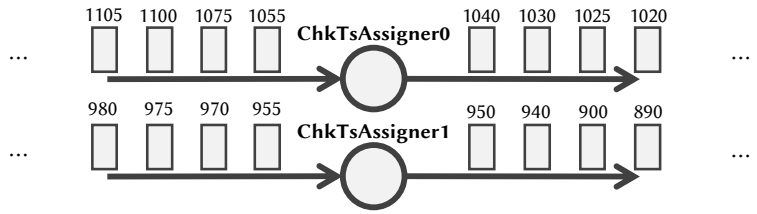


Fig. 6. Timestamp assignment for a checkpoint.

receives the notification, it sends the current LatestTS value to Checkpoint Store in the format: "$JobID, ChkID, OpInsID, LatestTs$", where $JobID$ and $OpInsID$ identify the ordinary workflow job and an operator instance of ChkTsAssigner, respectively. The maximum value among the LatestTS values for the checkpoint decides its chk timestamp.

Example 4.1. Let us consider the example shown in Figure 6. In this example, input streams are processed in parallelism 2 and two ChkTsAssigner operator instances are deployed. Small rectangles and numbers represent tuples and their timestamps, respectively. The JobID of the workflow is j. Outgoing tuples have already been processed by ChkTsAssigner and incoming tuples have not yet been processed. If the notification of checkpoint with ID i arrives at this moment, two operator instances store "$j, i, 0, 1040$" and "$j, i, 1, 950$" into Checkpoint Store. In this case, the chk timestamp of checkpoint i becomes 1040.

4.5 Decision of Replay Checkpoint

To reproduce the target tuple, the provenance workflow should be replayed from the appropriate replay checkpoint. This section describes how to decide the replay checkpoint. Given a target tuple with timestamp ts_o, the replay checkpoint with ID i should meet the following two conditions:

(1) The provenance workflow, restored from the checkpoint i, should produce the target tuple with its correct provenance.
(2) The checkpoint i should be the latest one that satisfies the above condition.

Note that the first condition requests the correctness of the provenance as well as reproduction of the target tuple. As explained in Subsection 3.3, provenance of an output tuple consists of all input tuples that contributed to the output. Therefore, to ensure the correct provenance of the target tuple, we must replay all input tuples that may have contributed to the target tuple. When the target tuple has timestamp ts_o, the smallest timestamp of an input tuple that may have contributed to this target tuple can be decided on the basis of how each operator determines the timestamp of each output tuple. If the workflow contains the window-based Aggregation or Join, all input tuples inside the window whose timestamps are less than or equal to ts_o may contribute to the output. If the workflow contains no window-based operators, only input tuples with timestamp ts_o contribute to the output tuple with ts_o. On the basis of these observations, we select the replay checkpoint as follows.

Replay Checkpoint Decision: Let the summation of window sizes in the workflow be $Wsize$[7]*. If the workflow does not include any window,* $Wsize$ *is* 0*. Then, for a target tuple with timestamp* t_o*, select the replay checkpoint* i *with the maximum chk timestamp* ts_i *satisfying* $ts_i < ts_o - Wsize$*.*

Lemma. The checkpoint chosen by Replay Checkpoint Decision satisfies both the above required conditions.

[7]If the workflow has multiple paths, we calculate the sum of window sizes along each path and take the maximum.

Proof. Here, we prove that the checkpoint determined by the proposed scheme satisfies the two required conditions for a replay checkpoint described in this section.

Condition (1): We prove that all input tuples contributing to the derivation of an output tuple with timestamp t_o are replayed from the checkpoint. Let us consider an output tuple produced by an operator. Then, timestamps of the contributing input tuples must be within the interval $[t_o - ws, t_o]$, where ws is the window size ($ws = 0$ for Filter and Map), according to properties of operators in Subsection 3.1. Therefore, $t_o - ws$ is the smallest timestamp that contributing input tuples can have. Thus, a checkpoint with a timestamp less than $t_o - ws$ ensures that all contributing input tuples are replayed. In general, the output tuple is produced by sequences of multiple operators along paths. The same discussion can be applied to each operator along the paths. Therefore, $t_o - Wsize$ represents the possibly smallest timestamp of a contributing tuple. Therefore, the checkpoint selected by the proposed scheme guarantees that the target tuple is produced with its correct provenance.

Condition (2): By definition, checkpoint timestamps must be monotonically increasing. Since the proposed scheme selects the checkpoint with maximum timestamp satisfying $ts_i < t_o - Wsize$, the selected checkpoint is guaranteed to be the latest under it. □

Ananke [26] used a similar approach based on timestamps and window sizes. However, it focuses on forward provenance and proposes a method to determine whether an input tuple will contribute to the derivation of future output tuples. In contrast, our approach takes the opposite direction: it identifies the range of input tuples that contribute to a given output tuple based on its timestamp and the window sizes.

4.6 Resource Deployment for Provenance Workflow

LPStream requires additional hardware resources to run the provenance workflow for provenance generation. LPStream offers users two scenarios for resource deployment. Note that, in both cases, the snapshot of the replay checkpoint is shared between both workflows through the Checkpoint Store provided by LPStream.

Provisioning an additional cluster on demand: Before users send the original workflow to LPStream, they first provision an additional cluster for the provenance workflow and provide its information (e.g., the IP address and port of the new Flink cluster) along with the workflow. Given a target tuple, the Provenance Mode Initiator starts the provenance workflow from the replay checkpoint on the specified cluster. Although the workflow finishes by generating its provenance, the additional cluster remains active for future target tuples until the intended application stops.

Allocating additional resources in the main cluster: In this scenario, users do not need to deploy resources in advance. Given a target tuple, the Provenance Mode Initiator triggers the provenance workflow with the replay checkpoint using available resources in the main cluster that are not utilized by the ordinary workflow. In fact, Apache Flink can start additional jobs on the same cluster by assigning unused resources. Although the provenance workflow finishes upon completion of the provenance generation, the available resources remain ready for future provenance generation. Note that users must ensure that sufficient resources are available in the main cluster when the provenance workflow starts.

LPStream can generate correct provenance even if the parallelism of the provenance workflow is reduced compared to the ordinary workflow, as long as the assumptions in Subsection 4.2 are satisfied. Therefore, while LPStream does require additional hardware resources, resource allocation can be adjusted according to the user's application needs.

5 Implementation

We developed the prototype of LPStream [31] using GeneaLog and Flink. As explained in Subsection 3.3, GeneaLog is the state-of-the-art eager provenance framework for stream processing, and we deployed it to execute provenance workflows in the provenance mode of LPStream. This section describes our implementation, highlighting the following three points: i) operators, ii) ChkTsAssigner operator, and iii) workflow preprocessing.

5.1 Operators

In this paper, we have introduced four types of operators:

Source/Sink: In our implementation, streams are ingested from Kafka. Flink provides users with connectors to read/send streams from/to Kafka. We employ $fromSource()$ and $sinkTo()$ to invoke them in Flink with Kafka connectors.

Transformer: We implemented Filter, Map, Aggregation, and Join operators with FilterFunction, MapFunction, AggregateFunction, and JoinFunction, respectively. Each operator are implemented in the Flink program as follows: Filter $\rightarrow filter()$, Map $\rightarrow map()$, Aggregation $\rightarrow keyBy().window().aggregate()$, and Join $\rightarrow join().where().equalTo().window().with()$.

Send/Receive: As mentioned in Subsection 3.3, Send/Receive operators are implemented as a Flink serializer/deserializer, respectively.

WatermarkGen: $assignTimestampsAndWatermarks()$ (denoted as $watermark()$) and WatermarkStrategy are employed for WatermarkGen. We assume that original workflows should implement the following two functions in WatermarkStrategy: $createTimestampAssigner()$ and $createWatermarkGenerator()$. The first function specifies which attribute represents event time in an input tuple, and the second function defines how the watermarks are inserted with respect to event times of incoming tuples.

5.2 ChkTsAssigner Operator

As mentioned in Subsection 4.4, ChkTsAssigner sends the latest timestamp to Checkpoint Store when it is notified of the completion of a checkpoint to assign the chk timestamp. We developed the operator as $assignChkTs()$, which implements RichMapFunction in the Flink program and has a state to keep the latest timestamp. Note that the original workflows contain WatermarkStrategy, which has $createTimestampAssigner()$. ChkTsAssigner reuses the function to extract timestamps from tuples. It also implements the *CheckpointListener* interface on Flink, which enables it to catch the notification when a checkpoint is completed.

5.3 Workflow Preprocessing

This section introduces how original workflows for Flink are preprocessed to work on LPStream. This preprocessing is based on a wrapping approach in GeneaLog. The preprocessor processes a given original workflow and generates a preprocessed workflow as an ordinary one or a provenance one depending on the switching argument. For this purpose, we extended the original GeneaLog wrappers. The extension contains the implementation of a new mode for the ordinary workflow, in which computation for provenance generation is skipped. The provenance workflow runs on GeneaLog and generates eager provenance while processing streams. The ordinary workflow runs on Flink, like original workflows without provenance, but their types/classes are consistent with provenance workflows so that their checkpoint snapshots can be used to start provenance workflows. In addition, the ordinary workflow assigns chk timestamp to each checkpoint. Figure 7.a shows an example fragment of an original Flink workflow.

```
1.  final StreamExecutionEnvironment env = ...;
2.  DataStream<O1> s1 = env.fromSource(...);
3.  DataStream<O2> s2 = s1.map(...);
4.  DataStream<O3> s3 = s2.assignTimestampsAndWatermarks(...);
5.  DataStream<O4> s4 = s3.filter(...);
6.  DataStream<O5> s5 = s4.keyBy(...).window(...).aggregate(...);
7.  s5.toSink(...);
```

a)

```
1.  final StreamExecutionEnvironment env = ...;
2.  ExperimentSettings settings = ExperimentSettings.newInstance(...);
3.  final LPOpWrapperStrategy LP = settings.lpOpWrapperStrategy().apply(...);
4.  FlinkSerializerActivator.LPSTREAM.activate(env, settings);
5.  DataStream<O1> s1 = env.fromSource(...).uid(...);
6.  DataStream<LP<O2>> s2 = s1.map(LP.initMap(...)).uid(...).map(LP.map(...)).uid(...);
7.  If ordinarymode:
8.     DataStream<LP<O2>> s2_ext = s2.map(LP.assignChkTs(...)).uid(...);
9.  Else:
10.    DataStream<LP<O2>> s2_ext = s2;
11. DataStream<LP<O3>> s3 = s2_ext.assignTimestampsAndWatermarks(LP.assignTimestampsAndWatermarks(...)).uid(...);
12. DataStream<LP<O4>> s4 = s3.filter(LP.filter(...)).uid(...);
13. DataStream<LP<O5>> s5 = s4.keyBy(LP.keyBy(...)).window(...).aggregate(LP.aggregate(...)).uid(...);
14. s5.toSink(LP.sink(...));
```

b)

Fig. 7. a) Original workflow, b) Preprocessed workflow.

Algorithm 1 Workflow Preprocessing

1: Add preambles after the declaration of StreamExecutionEnvirionment.
2: Insert $initMap()$ between $fromSource()$ and the subsequent $map()$.
3: **for** operator in the workflow **do**
4: **if** operator is $map()$, $filter()$, $keyBy()$, $aggregate()$, $where()$, $equalTo()$, $with()$, or $watermark()$ **then**
5: Wrap it using the corresponding wrapper functions.
6: **if** operator is $watermark()$ **then**
7: Insert $assignChkTs()$ for ordinary mode prior to this.
8: **if** operator is $sinkTo()$ **then**
9: Replace the Sink with our Sink and remove the definition of the user-defined Sink.
10: **if** operator is none of $keyBy()$, $window()$, $where()$, $equalTo()$, or $join()$ **then**
11: Insert $uid()$ after it.
12: **for** variable declaration of DataStream<T> **do**
13: **if** the variable is not initialized by the output of $fromSource()$ **then**
14: Wrap the data type T as LP<T>.

Algorithm 1 outlines the preprocessing procedure. Figure 7.b shows a fragment of the workflow preprocessed from the original fragment in Figure 7.a. Let us consider the preprocessing of the original workflow. Line 1 of the algorithm (denoted as Line 1A) simply adds preambles after the declaration of StreamExecutionEnvironment (lines 2–4 in Figure 7.b, denoted as lines 2-4b). These preambles define variables that wrap the original program. Line 2A inserts $initMap()$ into the original workflow (line 6.b). Lines 3–11A apply rewriting rules to each operator in the workflow. If

Table 1. Workflow summary (So: Source, W: WatermarkGen, F: Filter, M: Map, J: Join, A: Aggregate, Si: Sink). LR and SynA are real-time workflows, and the others are window workflows. Wsize, IS, and PS denote the window size, the input size, and provenance size in GeneaLog, respectively. Note that SynC corresponds to the example workflow in Figure 4.

Workflow Name	Summary	Operators	Wsize	Avg. IS	Avg. PS
LR	Project an input stream to extract car type, id, speed, direction, and position attributes.	So,W, M,Si	NA	64	1
SynA(10) SynA(100) SynA(400)	Select tuples representing temperature measurements.	So,W,F, M,Si	NA	26 116 416	1
Nexmark Nexmark2	Join auction stream and bid stream with auction ID.	So,W,F, M,J,Si	20ms 1s	339	2
NYC NYC2	Compute the number of rides and the average distance for each taxi vendor for long-distance rides.	So,W,F, M,A,Si	2s 15s	102	8 44
YSB YSB2	Compute the number of Web views for each campaign.	So,W,F, M,A,Si	1s 10s	309	33 333
SynB(10) SynB(100) SynB(400)	Join temperature and power consumptions streams for each machine.	So,W,F, M,J,Si	1s	26 116 416	2
SynC(10) SynC(100) SynC(400)	Compute the average temperature for each machine.	So,W,F, M,A,Si	10s	26 116 416	990

the operator is one of the specified operators in line 4A, the rule wraps the operator with a wrapper function; for example, *map*() in line 3 in Figure 7.a is rewritten as *map*(*LP.map*()) in line 6b. If the operator is *watermark*(), the rule in lines 6–7A inserts *assignChkTs*() before it for ordinary mode (lines 7–8b). If the operator is *sinkTo*(), the rule in lines 8–9A replaces it with our Sink and removes the user-defined sink definition, if it exists (line 14b). The rule in lines 10–11A inserts *uid*() after specified operators. Assigning operator IDs is necessary to properly restore operator states when the job is restarted from a snapshot. Finally, for each variable declaration of DataStream<T>, lines 12–14A wrap the data type T as LP<T>, unless the variable is initialized by the output of *fromSource*() (lines 6, 8, 10, 11, 12, and 13b). In this way, the original workflow is translated into a modified program, in which the ordinary and provenance workflows are switched based on the input argument.

6 Experiments

This section explains our experimental evaluation. We used the following six nodes dedicated to the experiments: 2 Flink nodes (1 Job Manager + 1 Task Manager), 2 Kafka nodes (1 Zookeeper + 1 Broker), 1 Redis node, and 1 data source node. Flink executes workflows with parallelism 10. For the performance evaluation purpose, ordinary workflow and provenance workflow are executed on the same cluster, and they are run independently. This will facilitate a fair performance comparison. Each topic on Kafka has 10 partitions. The Redis node is used to store chk timestamps sent from

ChkTsAssigners. The data source node executes a Java program that sends tuples of a stream to Kafka, and it emulates stream data sources (e.g., sensors). It can control the shipping rate of tuples, which results in different ingestion rates into Flink. This functionality is required to measure the throughput. The data source node attaches an simulated event time to each input tuple on the basis of the input rate before feeding it to Kafka. Flink is set to take checkpointing every 5s as default. Each node has 10 CPU cores and 128 GB memory.

6.1 Experimental setting

We explain datasets, workflows, and the experimental setting.

Datasets: We use the following seven datasets: 1) Linear Road dataset (LR_{ds}) [8], 2) Nexmark dataset ($Nexmark_{ds}$) [32], 3) NewYork City Taxi dataset (NYC_{ds}) [24], 4) Yahoo Streaming Benchmarks dataset (YSB_{ds}) [12], 5) Synthetic(10), 6) Synthetic(100), and 7) Synthetic(400). LR_{ds}, $Nexmark_{ds}$, NYC_{ds}, and YSB_{ds} are benchmark or real datasets for streaming processing. LR_{ds} simulates positions and speeds of cars on an expressway. We assign an additional attribute to LR_{ds}, which is necessary to satisfy assumptions in Subsection 4.2. $Nexmark_{ds}$ represents auctions, bids, etc. on an online auction system. NYC_{ds} is a real dataset collected from taxies in New York City, and each tuple represents a taxi trip. YSB_{ds} simulates advertisements and user behaviors on the Web. We also prepared our own three datasets, Synthetic(i), which simulate temperature and power consumption measurements of machines in a factory; the datasets differ in their payload sizes, and (i) represents the size in bytes of each dataset. Keys are uniformly distributed in Synthetic(i).

Workflows: Each of the LR_{ds}, NYC_{ds}, and YSB_{ds} datasets has a previously used stream workflow for evaluation ([25, 26], [42], and [39], respectively). An additional workflow is deployed from Nexmark benchmark Q20 to include a join operation. We use 16 workflows in total: LR (for LR_{ds}), Nexmark and Nexmark2 (for $Nexmark_{ds}$), NYC and NYC2 (for NYC_{ds}), YSB and YSB2 (for YSB_{ds}), and SynA(i), SynB(i), and SynC(i) (for Synthetic(i)). Each workflow is summarized in Table 1, which shows the average input data size ([bytes]) and the average provenance size ([tuples]), which is the number of source tuples in the provenance. We show the provenance size when each workflow is executed in 10,000 tuples/s. LR and SynA(i) contain no window (real-time workflows), whereas the others do (window workflows). Nexmark/Nexmark2 differ only in their window sizes; the other properties of the operators are the same. The same remark applies to NYC/NYC2 and YSB/YSB2. The original LR workflow detects car accidents. In our throughput evaluation, we run each workflow with different input rates. Because the accident detection rate heavily depends on the input rate, we omitted that part for fair comparison in our experiment. We extended the original workflow [42] for NYC and NYC2 to consider multiple key values. The join with the external database in the original YSB workflow was preprocessed over the input stream because it is outside the scope of our framework. Since Flink SQL is used to define the original Nexmark workflow, we rewrite the workflow with DataStream API and window join which is also used for the other experimental workflows.

Comparison: We compare the performance of four approaches: *Baseline*, *GeneaLog*, *Ordinary*, and *Provenance*. *Baseline* corresponds to the original workflow running on Flink. *GeneaLog* is the state-of-the-art eager provenance approach. We implemented the workflow for GeneaLog using the Native approach, where additional metadata and processing codes for provenance generation are directly embedded in the original workflow. Because the original GeneaLog does not support checkpointing, we extended GeneaLog to enable it. *Ordinary* and *Provenance* correspond to the ordinary and provenance workflows of LPStream, respectively. *Provenance* employs the wrappers to add additional metadata and processing to the original workflow, unlike *GeneaLog*. Therefore, the performance of *Provenance* may deteriorate compared to that of *GeneaLog*. Although a comparison

with the coarse provenance computation in Ariadne would be interesting, a fair evaluation is difficult since Ariadne operates exclusively on Borealis.

6.2 Metrics

We use the following metrics for performance evaluation: sustainable throughput, latency, and duration for provenance generation.

Sustainable throughput: We define *sustainable throughput* for a workflow as the maximum input rate in which Flink can continue to stably execute the workflow [33, 34]. If the following three conditions are met, we regard Flink execution as stable:

(1) Latency is not continuously increasing.
(2) The CPU usage is below the threshold.
(3) The number of input tuples awaiting processing is not increasing.

The method used to measure latency is explained later; intuitively, however, latency refers to the time duration from when an input tuple is ingested into the workflow to when the corresponding output tuple is produced. In the throughput experiment, we executed each workflow in 12 min and monitored whether the three conditions were satisfied during the measurement. For the first condition, we computed the start latency (lat_s) and end latency (lat_e). First, we exclude tuples produced in the first 25% and last 10% of the execution since their latencies can fluctuate. Then, the start latency is the median of the latency measurements for tuples in the first 10% of the remaining tuples, and the end latency (lat_e) is the median of the latency measurements for tuples produced in the last 10%. If the lat_e/lat_s ratio is smaller than 3, we consider the first condition met. For the second condition, we checked whether the average CPU usage during the execution did not exceed 80%, where the threshold is based on [33, 34]. For the third condition, we monitored whether the measured throughput fell below 0.9x the input rate of the data source node.

We evaluated whether each case runs stably on Flink by gradually increasing the input rate of the source node. Every trial checks whether the measured output rate from Source is $\geq$ 90% of the input rate of the data source node. This ensures that streams are indeed being ingested almost at the specified rate. When the execution became unstable, we reported the input rate immediately before the breakdown as the sustainable throughput. We basically increased the input rate in increments of 10,000 tuples/s. We first increased the input rate in increments of 100,000 tuples/s to narrow down the range of sustainable throughput. In some cases, data ingestion into Kafka was the bottleneck of the workflow. Except for such cases, we evaluate the sustainable throughput by increasing the input rate in smaller increments of 10,000 tuples/s. We show the highest sustainable throughput in three trials for each case.

Latency: We define *latency* as the time from when an input tuple arrives at the Source operator to when the corresponding output tuple reaches the Sink operator. For approaches deriving provenance (i.e., *GeneaLog* and *Provenance*), the time for traversing contribution graph at Sink is also included. In the Aggregation and Join operators, several input tuples contribute to an output tuple. If the workflow includes such operators, the latency for an output tuple is the time from when the latest contributing input tuple arrives at Source to when the corresponding output tuple reaches Sink. For latency evaluation, the input rate was set to 10,000 tuples/s, which is smaller than the sustainable throughputs for all of the workflows. We calculated the latency values of output tuples produced in 12 min runs for each case. Similarly to the case of sustainable throughput, tuples in the first 25% and last 10% intervals of the run were excluded from the analysis.

Duration for provenance generation: This metric is the time from when Provenance Mode Initiator starts the execution of the provenance mode to when Provenance Monitor identifies the provenance of the target tuple. In the experiment, the output stream of the provenance workflow

Table 2. Sustainable throughput [10^5 tuples/sec]. Upper: Real-time workflows, Lower: Window workflows. Nex. is abbreviation of Nexmark.

	Base.	Gen.	Ord.	Prov.	Ord#.
LR	10.8	8.3 (-23)	9.9 (-8)	7.3 (-32)	10.0 (-7)
SynA(10)	14.0	10.7 (-24)	13.2 (-6)	9.2 (-34)	13.1 (-6)
SynA(100)	10.2	8.0 (-22)	9.6 (-6)	6.8 (-33)	10.0 (-2)
SynA(400)	≥5	≥4(-20)	≥5(0)	3.8 (-24)	NA
Nex.	3.1	1.7 (-45)	2.3 (-26)	1.4 (-55)	2.3 (-26)
Nex.2	2.9	1.6 (-45)	2.2 (-24)	1.3 (-55)	2.3 (-21)
NYC	≥13	8.9 (-32)	10.5(-19)	7.1 (-45)	10.7(-18)
NYC2	≥13	2.9 (-78)	10.6(-18)	2.7 (-79)	10.8(-17)
YSB	≥6	5.3 (-12)	≥6 (0)	4.1 (-32)	NA
YSB2	≥6	1.6 (-73)	≥6 (0)	1.4 (-77)	NA
SynB(10)	1.3	0.9 (-31)	1.2 (-8)	0.8 (-38)	1.2 (-8)
SynB(100)	1.0	0.7 (-30)	1.0 (0)	0.6 (-40)	1.1 (+10)
SynB(400)	0.6	0.4 (-33)	0.6 (0)	0.4 (-33)	0.8 (+33)
SynC(10)	16.1	2.2 (-86)	7.1 (-56)	1.4 (-91)	7.1 (-56)
SynC(100)	9.0	1.7 (-81)	5.4 (-40)	1.2 (-87)	5.5 (-39)
SynC(400)	4.1	0.3 (-93)	3.1 (-24)	0.3 (-93)	2.8 (-32)

is first sent to Kafka. Provenance Monitor continuously reads each output tuple from Kafka, and checks whether it matches the target tuple with its timestamp and attribute values. When a match is found, the tuple is written on a file to return it to users. This marks the moment when the target tuple is identified by the Provenance Monitor. The objective of the metric is to evaluate the computation time required to generate lazy provenance in the provenance mode of LPStream. For each workflow, we executed the workflow in the ordinary mode for 12 min and randomly selected 20 target tuples from the produced tuples. We then measured the time required to obtain the provenance of each target tuple and derived the average time as the result. This metrics was evaluated with both real-time and windowed workflows. For the performance evaluation purpose, we stop ordinary mode once 20 target tuples are selected. Provenance workflow for provenance generation is executed independently of ordinary workflow. The checkpoint interval can affect the duration of provenance generation. To evaluate the effect of different checkpoint intervals, we tested three cases (5s, 30s, and 60s).

6.3 Evaluation

6.3.1 Sustainable throughput. Table 2 shows the sustainable throughput for all cases. Each value represents the measured maximum input rate in which Flink runs stably for the case. As mentioned before, in some cases, data ingestion into Kafka was the bottleneck of the workflow. Such cases are shown by "≥X", which means that X [tuples/s] is the maximum input rate evaluated in this experiment but the workflow may run stably with greater throughput. For *GeneaLog*, *Ordinary*, and *Provenance*, we also show the decreasing ratio in brackets compared with *Baseline*. We will explain the additional column Ordinary# later.

The overall trend is that *Baseline* shows the highest sustainable throughput, and *Ordinary*, *GeneaLog*, and *Provenance* follow, in that order (*Baseline* ≥ *Ordinary* > *GeneaLog* ≥ *Provenance*). This finding suggests that eager provenance generation strongly affects the sustainable throughput. There

are two considerable causes for the overhead on *Ordinary* compared with *Baseline*: One is wrapping data objects and operators relevant to provenance generation, and the other is the additional operator ChkTsAssigner. To analyze this, we evaluated the sustainable throughput of *Ordinary#*, a variation of *Ordinary*, in which ChkTsAssigner is removed. As shown in Table 2, the sustainable throughput of *Ordinary#* is almost comparable to that of *Ordinary*[8] in all workflows, suggesting that the effect of ChkTsAssigner is negligible. This implies that the main cause of the degradation is the wrapping of data objects and operators. This will also degrade *Provenance* compared to *GeneaLog*. The observation that *Ordinary* > *GeneaLog* suggests that additional computational cost caused by wrapping data objects and operators without provenance is much smaller than the cost for eager provenance generation. The sustainable throughput of *Ordinary* is the same as that of *Baseline* for five workflows and 75% or more of *Baseline* for eight workflows. This result suggests that *Ordinary*, which executes the ordinary workflow in our framework, ensures comparable throughput for most of the workflows. In addition, *Provenance*, which replays the workflow and generates lazy provenance, shows the same throughput as *GeneaLog* for two workflows (SynB(400) and SynC(400)) and the 20% or less degradation for ten workflows (LR, SynA(10), SynA(100), SynA(400), Nexmark, Nexmark2, NYC2, YSB2, SynB(10), and SynB(100)). This indicates that the provenance workflow in LPStream performs comparably to the original GeneaLog. The only exception is SynC. In SynC, throughput degradation of *Ordinary* was more than in the other cases. The aggregate operator in SynC needs to deal with much more incoming tuples per unit time compared with other workflows. We suspect that the increase in function calls of the wrapped aggregate operator is the main reason for the degradation.

Next, we consider the effect of workflow properties on throughput. When the window size increases, resulting in the increased provenance size (the number of input tuples in provenance) (i.e., NYC vs. NYC2 and YSB vs. YSB2), the sustainable throughput of *GeneaLog* and *Provenance* worsens compared with *Baseline* and *Ordinary*. The reason will be the increased computation cost for traversal of the contribution graph due to the increased provenance size. SynA, SynB, and SynC show how the tuple size affects the sustainable throughput. When the tuple size increases, the sustainable throughput decreases in all the approaches. For example, the sustainable throughputs of *Baseline* are 14.0×10^5 in SynA(10), 10.2×10^5 in SynA(100), and $\geq 5 \times 10^5$ in SynA(400). Note that the decreasing ratio is roughly the same for all of the approaches. For example, *GeneaLog* shows throughputs of 10.7×10^5, 8.0×10^5, and $\geq 4 \times 10^5$ in SynA(10), SynA(100), and SynA(400), respectively. This result suggests that the overhead caused by increasing input tuple size is almost the same for all of the approaches.

These results demonstrate that our lazy approach (*Ordinary*) substantially reduces the overhead for the original workflow execution compared with the eager approach. They further demonstrate the efficacy of our lazy provenance approach.

6.3.2 Latency. Figure 8 shows the results of latency experiments. The X and Y axes represent the four approaches and latency, respectively. We used boxplots to show the results, which represent the median, first quartile (Q1), third quartile (Q3), the lowest datum above $Q1 - 1.5IQR$, and the highest datum below $Q3 + 1.5IQR$ ($IQR = Q3 - Q1$). Because latency values for output tuples tend to fluctuate substantially, boxplots are more suitable for showing the range of latencies. Flier values are not plotted in the figure. Focusing on the median latency, *GeneaLog* and *Provenance* are larger than *Baseline* and *Ordinary* in nine workflows. In addition, the latencies of *Ordinary* are comparable to those of *Baseline* in most cases. The latencies of *GeneaLog* and *Provenance* are likewise comparable.

[8]The Kafka ingestion is the bottleneck in *Ordinary* for workflows with NA, where comparison using Ordinary# does not make sense.

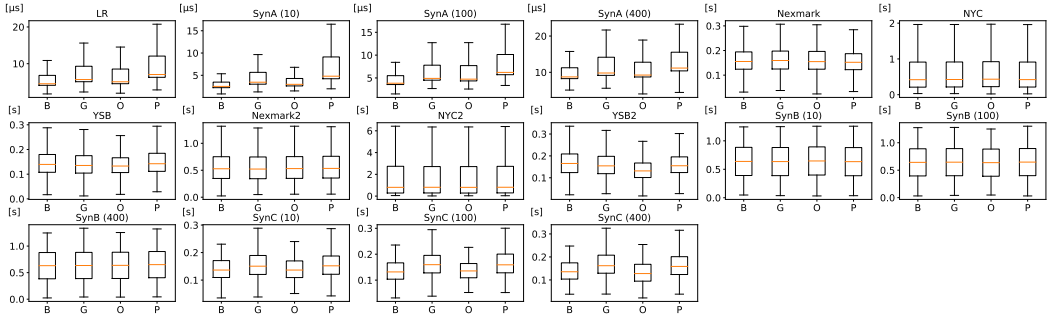


Fig. 8. Latency (B: Baseline, G: GeneaLog, O: Ordinary, P: Provenance).

Table 3. Overall latency (OL), traversal time (TT), and traversal ratio (TR) [%]. Upper: Real-time workflows, Lower: Window workflows. G: GeneaLog, P: Provenance.

	G.OL	G.TT	G.TR	P.OL	P.TT	P.TR
LR	5.7μs	683ns	12	7.0μs	1.3μs	19
SynA(10)	3.5μs	680ns	20	4.8μs	1.3μs	27
SynA(100)	4.9μs	804ns	16	6.2μs	1.4μs	22
SynA(400)	9.8μs	787ns	8	11.2μs	1.3μs	12
Nexmark	159ms	8.8μs	6e-3	152ms	5.8μs	4e-3
Nexmark2	522ms	2.4μs	5e-4	534ms	2.4μs	5e-4
NYC	425ms	7.8μs	2e-3	422ms	8.9μs	2e-3
NYC2	810ms	34.0μs	4e-3	813ms	35.3μs	4e-3
YSB	135ms	78.7μs	0.06	142ms	81.6μs	0.06
YSB2	154ms	792μs	0.5	155ms	815μs	0.5
SynB(10)	635ms	3.0μs	5e-4	634ms	2.0μs	3e-4
SynB(100)	646ms	3.1μs	5e-4	648ms	2.0μs	3e-4
SynB(400)	634ms	4.5μs	7e-4	651ms	2.0μs	3e-4
SynC(10)	151ms	2.5ms	2	152ms	2.3ms	2
SynC(100)	160ms	2.9ms	2	159ms	2.2ms	1
SynC(400)	162ms	4.0ms	2	159ms	2.2ms	1

Note that the latencies of *GeneaLog* and *Provenance* are always larger than those of *Baseline* and *Ordinary* in real-time workflows (LR and SynA). However, no substantial difference is observed in some window workflows. LR and SynA do not contain expensive operators like Join and Aggregation. In fact, their baseline latency is on the order of several microseconds. Therefore, the provenance generation strongly affects their latency. Table 3 shows the overall latency (median), traversal time (median), and its ratio in the overall latency for *GeneaLog* and *Provenance*. As we mentioned, in LR and SynA, the ratio is relatively large (8–20% in *GeneaLog* and 12–27% in *Provenance*). However, window workflows include dominant operators such as Join and Aggregation. In window workflows, the latency caused by such operators occupies most of the overall latency (almost 100%), which is on the order of milliseconds. As shown in Table 3, in such workflows, the overhead for the traversal is almost negligible.

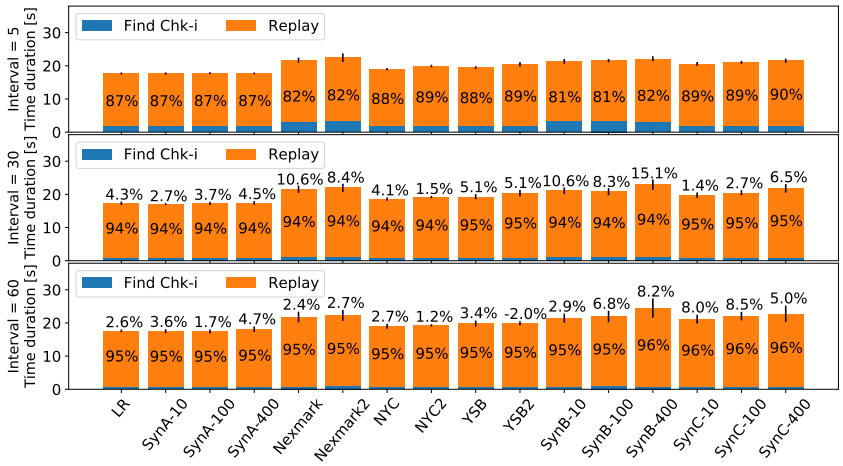


Fig. 9. Time duration for provenance generation.

In summary, if the workload of the original workflow is small, the effect of the eager approach on latency is relatively large, whereas the effect can otherwise be small or negligible. Notably, even if the effect on latency is small, the overall throughput tends to be strongly affected, as explained above.

6.3.3 Duration for Provenance Generation. The experimental results are presented in Figure 9. The top, middle, and bottom charts show results corresponding to the checkpoint intervals being set to 5s, 30s, and 60s, respectively. The height of each bar shows the average duration for provenance generation, and each bar also shows the standard deviation. Each bar is divided into two parts: the duration to identify the replay checkpoint for the target tuple (*Find Chk-i*) and the duration for executing the provenance workflow until the target provenance is obtained (*Replay*). The ratio of *Replay* to the overall duration is shown inside each bar. The middle and bottom figures also show the increasing rate of the overall duration in comparison with the cases of 5s and 30s, respectively. The experiments show that LPStream generates the lazy provenance of a target output tuple in a timeframe on the order of tens of seconds. This duration tends to increase as the window size and tuple size increase. The reason for the correlation between duration and window size is that the provenance workflow is restarted from an earlier checkpoint and more input tuples are replayed. The correlation between duration and tuple size is attributable to increased workload of the provenance workflow. When we increase the checkpoint interval, the overall duration increases. A longer interval means less frequent checkpointing, which makes each checkpoint cover a longer duration. Therefore, LPStream has to revert to a much earlier checkpoint time beyond the calculated chk timestamp, resulting in more input tuples being replayed in the provenance mode. Actually, the ratio of replay to the overall duration increases as the checkpoint interval becomes longer.

7 Conclusion

We propose LPStream as the first framework supporting lazy provenance for stream processing. LPStream uses checkpointing, which general SPEs provide for fault tolerance of the processing, and generates provenance by replaying from a checkpoint. We described the architecture of LPStream and its implementation and introduced experimental results with diverse datasets, queries, and metrics. Our experiments suggested that LPStream can decrease overhead for workflow execution

in terms of throughput and latency compared with the state-of-the-art eager provenance framework. In addition, we demonstrated that the computational cost to generate lazy provenance is reasonable. Future work will include evaluations using different SPEs and data sources and support different types of provenance (e.g., why-not provenance).

Acknowledgments

This paper is based on results obtained from the projects, "Research and Development Project of the Enhanced infrastructures for Post-5G Information and Communication Systems" (JPNP20017), commissioned by the New Energy and Industrial Technology Development Organization (NEDO), JSPS KAKENHI (JP24KJ0461, JP25K22810, JP23K28089, JP22K19802, JP23K24949), JST CREST (JPMJCR22M2), and AMED (JP21zf0127005).

References

[1] Sherif Akoush, Ripduman Sohan, and Andy Hopper. 2013. HadoopProv: Towards Provenance as a First Class Citizen in MapReduce. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*. USENIX Association, Lombard, IL.

[2] Apache. 2024. Flink. Retrieved June 15, 2024 from https://flink.apache.org/

[3] Apache. 2024. HDFS Architecture Guide. Retrieved June 1, 2024 from https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[4] Apache. 2024. Kafka. Retrieved June 1, 2024 from https://kafka.apache.org/

[5] Apache. 2024. KAFKA STREAMS. Retrieved June 15, 2024 from https://kafka.apache.org/documentation/streams/

[6] Apache. 2024. Spark Streaming Programming Guide. Retrieved June 15, 2024 from https://spark.apache.org/docs/latest/streaming-programming-guide.html

[7] Apache. 2024. Structured Streaming Programming Guide. Retrieved June 15, 2024 from https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[8] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB '04)*. VLDB Endowment, 480–491.

[9] Leilani Battle, Danyel Fisher, Robert DeLine, Mike Barnett, Badrish Chandramouli, and Jonathan Goldstein. 2016. Making Sense of Temporal Queries with Interactive Visualization. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. 5433–5443.

[10] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and Where: A Characterization of Data Provenance. In *Database Theory — ICDT 2001*. 316–330.

[11] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends® in Databases* 1, 4 (2009), 379–474.

[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. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1789–1792.

[13] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. 2005. DBNotes: a post-it system for relational databases based on provenance. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. 942–944.

[14] Y. Cui and J. Widom. 2003. Lineage tracing for general data warehouse transformations. *The VLDB Journal* 12, 1 (2003), 41–58.

[15] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* 25, 2 (2000), 179–227.

[16] Susan B. Davidson and Juliana Freire. 2008. Provenance and Scientific Workflows: Challenges and Opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 1345–1350.

[17] Wim De Pauw, Mihai LeŢia, Buğra Gedik, Henrique Andrade, Andy Frenkiel, Michael Pfeifer, and Daby Sow. 2010. Visual Debugging for Stream Processing Applications. In *Runtime Verification*. 18–35.

[18] B. Glavic and G. Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *2009 IEEE 25th International Conference on Data Engineering*. 174–185.

[19] Boris Glavic, Kyumars Sheykh Esmaili, Peter Michael Fischer, and Nesime Tatbul. 2013. Ariadne: Managing Fine-Grained Provenance on Data Streams. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. 39–50.

[20] Mikael Gordani Shahri, Andréas Erlandsson, Dimitris Palyvos-Giannas, and Vincenzo Gulisano. 2021. Poster: Twins, a Middleware for Adaptive Streaming Provenance at the Edge. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking (ICDCN '21)*. 235–236.
[21] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal* 26, 6 (2017), 881–906.
[22] Mohammad Rezwanul Huq, Andreas Wombacher, and Peter M. G. Apers. 2011. Inferring Fine-Grained Data Provenance in Stream Data Processing: Reduced Storage Cost, High Accuracy. In *Database and Expert Systems Applications*. 118–127.
[23] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: Data provenance support in spark. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 9. 216–227.
[24] City of New York. 2024. TLC Trip Record Data. Retrieved August 16, 2024 from https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[25] Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. 2019. GeneaLog: Fine-grained data streaming provenance in cyber-physical systems. *Parallel Comput.* 89 (2019).
[26] Dimitris Palyvos-Giannas, Bastian Havers, Marina Papatriantafilou, and Vincenzo Gulisano. 2020. Ananke: a streaming framework for live forward provenance. *Proc. VLDB Endow.* 14, 3 (2020), 391–403.
[27] Dimitris Palyvos-Giannas, Katerina Tzompanaki, Marina Papatriantafilou, and Vincenzo Gulisano. 2022. Erebus: explaining the outputs of data streaming queries. *Proc. VLDB Endow.* 16, 2 (oct 2022), 230–242. doi:10.14778/3565816.3565825
[28] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-Grained Lineage at Interactive Speed. *Proc. VLDB Endow.* 11, 6 (feb 2018), 719–732.
[29] Watsawee Sansrimahachai, Mark J. Weal, and Luc Moreau. 2012. Stream ancestor function: A mechanism for fine-grained provenance in stream processing systems. In *2012 Sixth International Conference on Research Challenges in Information Science (RCIS)*. 1–12.
[30] Meta Open Source. 2024. RocksDB. Retrieved June 1, 2024 from https://rocksdb.org/
[31] Open Source. 2024. LPStream Implementation. https://github.com/madamaya/lpstream
[32] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. Nexmark–a benchmark for queries over data streams (draft). *Technical report* (2008).
[33] Giselle van Dongen and Dirk Van den Poel. 2020. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1845–1858.
[34] Giselle Van Dongen and Dirk Van Den Poel. 2021. Influencing Factors in the Scalability of Distributed Stream Processing Jobs. *IEEE Access* 9 (2021), 109413–109431.
[35] Nithya Vijayakumar and Beth Plale. 2007. Tracking stream provenance in complex event processing systems for workflow-driven computing. In *EDA-PS Workshop*.
[36] Nithya N. Vijayakumar and Beth Plale. 2006. Towards Low Overhead Provenance Tracking in Near Real-Time Stream Filtering. In *Provenance and Annotation of Data*. 46–54.
[37] Min Wang, Marion Blount, John Davis, Archan Misra, and Daby Sow. 2007. A time-and-value centric provenance model and architecture for medical event streams. In *Proceedings of the 1st ACM SIGMOBILE International Workshop on Systems and Networking Support for Healthcare and Assisted Living Environments (HealthNet '07)*. 95–100.
[38] E. Wu, S. Madden, and M. Stonebraker. 2013. SubZero: A fine-grained lineage system for scientific databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 865–876.
[39] yahoo. 2022. streaming-benchmarks. Retrieved August 26, 2024 from https://github.com/yahoo/streaming-benchmarks
[40] Masaya Yamada, Hiroyuki Kitagawa, Toshiyuki Amagasa, and Akiyoshi Matono. 2023. Augmented lineage: traceability of data analysis including complex UDF processing. *The VLDB Journal* 32, 5 (2023), 963–983.
[41] Qian Ye and Minyan Lu. 2021. s2p: Provenance Research for Stream Processing System. *Applied Sciences* 11, 12 (2021).
[42] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.* 12, 5 (2019), 516–530.
[43] N. Zheng, A. Alawini, and Z. G. Ives. 2019. Fine-Grained Provenance for Matching & ETL. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 184–195.

Received January 2025; revised April 2025; accepted May 2025