ABSTRACT
Data streaming enables online monitoring of large and continuous event streams in Cyber-Physical Systems (CPSs). In such scenarios, fine-grained backward provenance tools can connect streaming query results to the source data producing them, allowing analysts to study the dependency/causality of CPS events. While CPS monitoring commonly produces many events, backward provenance does not help prioritize event inspection since it does not specify if an event’s provenance could still contribute to future results.

To cover this gap, we introduce Ananke, a framework to extend any fine-grained backward provenance tool and deliver a live bipartite graph of fine-grained forward provenance. With Ananke, analysts can prioritize the analysis of provenance data based on whether such data is still potentially being processed by the monitoring queries. We prove our solution is correct, discuss multiple implementations, including one leveraging streaming APIs for parallel analysis, and show Ananke results in small overheads, close to those of existing tools for fine-grained backward provenance.

1 INTRODUCTION
Distributed, large, heterogeneous Cyber-Physical Systems (CPSs) like Smart Grids or Vehicular Networks [22] rely on online analysis applications to monitor device data. In this context, the data streaming paradigm [36] and the DataFlow model [2] enable the inspection of large volumes of continuous data to identify specific patterns [16, 28]. Streaming applications fit CPS’s requirements due to the high-throughput, low-latency, scalable analysis enabled by Stream Processing Engines (SPEs) [1, 4, 5, 8, 31] and their correctness guarantees, which are critical for sensitive analysis. Figure 1a shows two streaming applications, or queries, monitoring a vehicular network to spot cars visiting a specific area (Q₁) or speeding (Q₂). Vehicle reports (timestamp, id, position), or tuples, arrive every 5 minutes; \( t' \) is the \( i \)-th tuple from car \( j \), \( a' \) the \( i \)-th alert from query \( j \). This scenario is our running use-case throughout the paper.

Motivating challenge. CPSs need continuous monitoring for emerging threats or dangerous events [32, 37], which may result in many alerts that analysts are then left to prioritize [15, 34]. For streaming-based analysis, provenance techniques [19, 30], which connect results to their contributing data are practical ways to inspect data dependencies, since breakpoint-based inspection is not fit for live queries that run in a distributed manner and cannot be paused [14]. Existing provenance tools for traditional databases target backward tracing, to find which source tuples contribute to a result [7, 11, 12, 14, 19] (Figure 1b) and forward tracing, to find which results originate from a source tuple [7, 12]; however, streaming-based tools only exist for backward-provenance [19, 30].

The need for live, streaming, forward provenance is multifold: (1) while backward tracing can give assurance on the trustworthiness of end-results [11], forward tracing allows to identify all results linked to specific inputs [12], e.g. to mark all results linked to a privacy-sensitive datapoint (e.g. a picture of a pedestrian, in the context of Vehicular Networks) before such results are analyzed further; (2) live maintenance of the provenance graph avoids data duplication and allows to start the analysis of provenance data safely (e.g., once all sensitive results that could be connected to the
Contribution. Motivated by the open issues, our key question is: “Can we enrich data streaming frameworks that deliver backward provenance to efficiently provide live, duplicate-free, fine-grained, forward provenance for arbitrarily complex sets of queries?”

We answer affirmatively with our contributions:

- We formulate the concrete goals and evaluation metrics of solutions for live, duplicate-free, fine-grained, forward provenance.
- We implement a general framework, Ananke1, able to ingest backward provenance and deliver an evolving bipartite graph of live, duplicate-free, fine-grained, forward provenance (or simply live provenance) for arbitrary sets of queries. Ananke delivers each result and source tuple contributing to one or more results exactly once, distinguishing source data that could still contribute to more results from expired source data that cannot.
- Ananke’s key idea builds on our insights on forward provenance w.r.t. the backward provenance problem and defines a simple yet efficient approach, enabling specialized-operator-based implementations, as well as modular ones that utilize native operators of the underlying SPE. We design and prove the correctness of two streaming-based algorithmic implementations: one targeting to optimize the labeling of the expired source data as fast as possible, and one that shows how the general SPEs’ parallel APIs are sufficient to parallelize Ananke’s algorithm, and thus sustain higher loads of provenance data.
- We conduct a thorough evaluation of our Ananke implementation on top of Apache Flink [8], with real-world use cases and data, and also match with previous experiments and an implementation that delivers live forward provenance by relying on tools external to the SPE, for a fair comparison of Ananke’s overheads.

The implementations used in our evaluation are open-sourced at [18] for reproducibility. Figure 1c shows Ananke’s live provenance assuming both queries have processed all tuples up to time 8:21. Each source and sink tuple appear exactly once in the bipartite graphs. Some tuples are labeled by a green check-mark, indicating that they are expired and will not be connected to future results.

Organization: §2 covers preliminary data streaming and provenance concepts. §3 provides the definitions we use and also includes a formal problem formulation. §4–§5 cover our contribution, later evaluated in §6. We discuss related work in §7 and conclude in §8.

2 PRELIMINARIES

2.1 Data Streaming Basics

Like Apache Flink [8] (or simply Flink), Ananke builds on the DataFlow model [2]. Streams are unbounded sequences of tuples. Tuples have two attributes: the metadata \( \mu \) and the payload \( \phi \), an array of sub-attributes. The metadata \( \mu \) carries the timestamp \( t \) and possibly further sub-attributes. To refer to a sub-attribute of \( \mu \), e.g., \( \tau \), we use the notation \( t.\tau \). We reference \( \phi \)'s \( i \)-th sub-attribute as \( t.\phi[i] \) (omitting \( t \) when it is clear from the context). In combined notation, a stream tuple is written as \( \langle t.\mu, t.\phi \rangle \) (e.g., events measured by a sensor or reported by other applications).

Streaming queries (or simply queries) are composed of Sources, operators and Sinks. A Source forwards a stream of source tuples (e.g., events measured by a sensor or reported by other applications). Each source stream can be fed to one or more operators, the basic units manipulating tuples. Operators, connected in a Directed Acyclic Graph (DAG), process input tuples and produce output tuples; eventually, sink tuples are delivered to Sinks, which deliver results to end-users or other applications. In our model, we assume each tuple is immutable. Tuples are created by Sources and operators. The latter can also forward or discard tuples.

As source tuples correspond to events, \( \tau \) is set by the Source to when that event took place, the event time. Operators set \( \tau \) of each output tuple according to their semantics, while \( \phi \) is set by user-defined functions. Event time is not continuous but progresses in discrete increments defined by the SPE (e.g., milliseconds). We denote the smallest such increment of an SPE by \( \delta \). All major SPEs [3–5, 8] support user-defined operators but also provide native ones: Map, Filter, Aggregate and Join. Since we make use of such native operators, we provide in the following their formal description for self-containment. However, Ananke provides live provenance without imposing any restriction on the operators of the query. Figure 2 illustrates the native operators, the Source, and the Sink. We begin with stateless operators, which process tuples one-by-one.

A Filter (F) relies on a user-defined filtering condition \( C \) to either forward an input tuple, when \( C \) holds, or discard it otherwise.

A Map (M) uses a user-defined function \( F_M \) (to transform an input tuple into \( m \geq 1 \) output tuples) and \( S \), the schema of the output tuple payloads. It copies the \( \tau \) of each input into the outputs.

Differently from stateless operators, stateful ones run their analysis on windows, delimited groups of tuples maintained by the operators. Time windows are defined by their size \( WS \) (the length of the window), advance \( WA \) (the time difference between the left boundaries of consecutive windows), and offset \( WO \) (the alignment of windows relative to a reference time; in Flink, this is the Unix epoch). For example, a window having \( WS, WA, WO \) set to 60, 10 and 5 minutes, respectively, will cover periods [00:05,01:05), [00:15,01:15), etc. Consecutive periods covered by a window can overlap when \( WA < WS \). Lastly, the left and right boundaries of a window are inclusive and exclusive, respectively. We say a tuple \( t \) falls in a window \( [A,B) \) if \( A \leq t.\tau < B \). As windows can overlap, a tuple can fall into one or more windows.

We now present stateful operators in more detail.

An Aggregate (A) is defined by: (1) \( WS, WA, WO \): the window size, advance, and offset, (2) \( KB \): an optional key-by function to

---

1In Greek mythology, Ananke personifies inevitability, compulsion and necessity.
maintain separate (yet aligned) windows for different key-by values, 
(3) \(F_J\): a function to aggregate the tuples falling in one window 
into the \(\phi\) attribute of the output tuple created for such window, 
(4) \(S\): the output tuple’s payload schema. 

When an output tuple is created for a window (and a key-by 
value is being defined), we assume its timestamp is set to such 
window’s right boundary [3, 5, 8]. 

A Join \((J)\) matches tuples from two input streams, \(r\) and \(s\). It 
keeps two windows, one for \(r\) and one for \(s\) tuples, which share 
the same values for parameters \(W S, WA, W O\): 
the window size, advance, and offset. Each pair of \(r\) 
and \(s\) tuples sharing a common key are matched for every pair 
of windows covering the same event-time period they fall in. 
The Join operator relies on the following parameters: (1) \(W S, WA, W O\): 
the window size, advance, and offset, (2) \(KB\): a key-by function 
to maintain separate (yet aligned) pairs of windows for different 
key-by values, (3) \(P\): a predicate for pairs of tuples from the 
two input streams, (4) \(F_J\): a function to create the \(\phi\) attribute of 
the output tuple, for each pair of input tuples for which 
\(P\) holds, and (5) \(S\): the schema of the output tuple’s payload \(\phi\). Similarly, 
to the Aggregate, when an output tuple is created by a Join, its 
sub-attribute \(r\) is set to the right boundary of the window. 

In the remainder, we (1) differentiate between stateless and state-
ful only if necessary, we (2) assume \(W O = 0\) unless otherwise stated, 
and (3) assume a stream can be multiplexed to many operators. 

Figure 3 presents Figure 1’s queries. Source \(S\) emits tuples of schema 
\((\tau, [V_{id}, x, y])\) (timestamp, vehicle ID, \(x\)- and \(y\)-coordinates). In \(Q_1\), Filter \(F_1\) forwards tuples within region \(R\). Aggregate \(A_1\) 
counts each car’s reports, and Filter \(F_2\) forwards to Sink \(K_1\) only 
tuples with a count higher than 2 (as tuples arrive every 5 minutes, 
the count can only be 1, 2 or 3). In \(Q_2\), Aggregate \(A_2\) emits the mean 
speed of each car within the last 15 minutes. Filter \(F_3\) forwards the 
tuples whose mean speed\(^{2}\) exceeds 110km/h to Sink \(K_2\). 

2.2 Watermarks and Correctness Guarantees 

Because of asynchronous, parallel, and distributed execution, state-
ful operators processing tuples from multiple streams can receive 
such tuples out-of-order. Hence, receiving a tuple with a timestamp 
greater than some window’s right boundary does not imply that 
tuples received later could not still contribute to said window. 

To ensure result correctness for out-of-order streaming processing 
[25], Aggregate and Join rely on watermarks to make such 
distinction, as suggested by pioneer as well as state-of-the-art 
SPEs [8, 25]; the definition is paraphrased here: 

**Definition 3.2.** The watermark \(W_i^o\) of operator \(O_i\) at a point in 
wall-clock time \(o\) is the earliest event time a tuple to be processed by 
\(O_i\) can have from time \(o\) on (i.e., \(t_i, t_i \geq W_i^o\), \(\forall t_i \text{ processed from } o\)). 

Watermarks are created periodically by Sources and propagate 
as special tuples through the DAG\(^4\). Upon receiving a watermark, 
an operator stores the watermark’s time, updates its watermark 
to the minimum of the latest watermarks received from each of 

\(^2\)In Figure 1, given the grid’s cell size, cars’ mean speed is 120km/h if covering four 
cells in three consecutive tuples. 
\(^3\)Notice that from here on, we only differentiate between wall-clock time (or simply time) 
and event time if such distinction is not clear from the context. 
\(^4\)Notice that this assumption about in-band watermarks is not a constraint. Different 
watermarking schemes that could also be adopted are discussed in e.g., [25, 26]. 

Figure 3: The DAG of \(Q_1\), \(Q_2\) from Figure 1. Source tuples contain 
report’s timestamp in \(\mu\), and the vehicle ID \(V_{id}\) and position \(x, y\) in \(\phi\). Source tuples are forwarded to both queries. 
Tuples arriving at the Sinks correspond to alerts in Figure 1. 

3 DEFINITIONS AND GOALS 

This section includes the definitions we use to present and prove 
our contribution’s correctness, and a formal statement of our goals. 

**Definition 3.1.** We say a tuple \(t\) contributes directly to another 
tuple \(t^*\) if an operator produces \(t^*\) based on the processing of \(t\) and 
write: \(t \rightarrow t^*\). We then say \(t\) contributes to \(t^*\) and use the notation 
\(t^* \rightarrow t\) if \(t \rightarrow t' \rightarrow t'' \rightarrow \ldots \rightarrow t^*\). Thus, if \(t \rightarrow t^*\), then \(t \rightarrow t^*\). 

From the above, a source tuple \(t_0\) from Source \(S\) contributes to a 
sink tuple \(t_k\) received by Sink \(K\), if there is a directed, topologically-
sorted path \(S, O_1, \ldots, O_i, \ldots, O_k, K\) and a sequence of tuples \(t_0, \ldots, t_{i-1}, t_i, \ldots, t_k = t_K\), s.t. \(\forall i = 1, \ldots, k, \) where \(t_{i-1}\) and \(t_i\) are 
input and output tuples of \(O_i\), and \(t_{i-1} \rightarrow t_i\) for all \(i = 1, \ldots, k\). 

**Definition 3.2.** At time \(o\), tuple \(t\) is active if it can still contribute 
directly to a tuple produced by an operator. Otherwise, \(t\) is inactive. 
At time \(o\), tuple \(t\) is alive if it is active or if there is at least one active 
tuple \(t^*\) such that \(t \rightarrow t^*\). Otherwise, \(t\) is expired. 

For instance, if a Map produces a tuple \(t_2\) upon ingesting tuple 
\(t_1\), the latter is inactive, but remains alive as long as \(t_2\) is being 
processed downstream (or as long as \(t_2\) itself is alive). Note that all 
sink tuples are expired by definition. Using the above, we define 
the live provenance to be delivered by \(Ananke:\)
Definition 3.3. At time $\omega$ in the execution of a set of queries $Q$, being $\mathbb{K}_Q$ the set of $Q$’s Sinks, the live, duplicate-free, bipartite graph forward provenance $P(Q, \omega)$ consists of (1) a set of vertices $V$, containing exactly one vertex for each sink tuple forwarded to $\mathbb{K}_Q$, and exactly one vertex for each source tuple contributing to any sink tuple forwarded to $\mathbb{K}_Q$, (2) a set of edges $E$, each edge connecting a sink tuple with its contributing source tuples, and (3) a set of “expired” labels $L$, one for each vertex in $V$ if the tuple it refers to is expired.

Notice that if, at time $\omega$, a vertex in $V$ is alive, then more edges connecting such vertex to other vertices could be found later in the execution of $Q$. In Figure 1c, which shows $P(Q, 8:21)$, new edges could connect the vertices of tuples not marked as expired to vertices referring to sink tuples that have not yet been produced.

Given the preceding definitions, we now formulate our goals and the necessary requirements to reach these (using prefix R- for the latter). For brevity, we refer to vertices, referring to a source tuple or a sink tuple as source vertices and sink vertices, respectively, and use the expression expired for tuples and vertices interchangeably.

Problem formulation. Being $\mathbb{B}_Q$ a set of streams delivering $\mathbb{K}_Q$’s sink tuples with backward provenance via attribute $\mu$, the goal is to continuously deliver $P(Q, \omega)$’s $V$, $E$, and $L$, for increasing values of $\omega$, as one or multiple streams, to meet the following requirements:

- **R-V** Each vertex referring to a source or a sink tuple is delivered exactly once, by a tuple $(\tau_V, [ID_S, t_S])$ or $(\tau_V, [ID_K, t_K])$, respectively. $ID_i$ is a unique ID for the vertex associated to $t_i$; 
- **R-E** each edge between vertices $ID_S$ and $ID_K$ is delivered exactly once by a tuple $(\tau_E, [ID_S, ID_K])$, with $\tau_E$ greater than or equal to the timestamp $\tau_V$ of the connected vertices; and
- **R-L** an “expired” label is delivered once for each vertex $ID_i$ by a tuple $(\tau_L, [ID_i])$, with $\tau_L \geq \tau_E$, for each edge $E$ adjacent to $ID_i$.

For the queries in Figure 1a, Figure 4 shows $P(Q, 8:16)$ and the tuples delivered to update it to $P(Q, 8:22)$.

While referring to a set of queries $Q$ and a set of Sinks $\mathbb{K}_Q$ for generality, our problem formulation is justified even for a single query with exactly one Source and Sink, because subsequent sink tuples can have overlapping sets of source tuples (as in the example of Figure 1), and each such source tuple still needs to be delivered exactly once and later marked as expired.

Performance metrics. For provenance to be practical in real-world applications, its performance overheads need to be small. The efficiency of our solution is evaluated through its overhead on the following metrics:

- **Throughput**, number of tuples a query ingests per unit of time.
- **Processing Latency**, the delay in the production of a sink tuple after all its contributing source tuples have arrived at the query.
- **CPU utilization**, the percentage of the total CPU time a query utilizes, across all available processors (0-100%).
- **Memory consumption**, the amount of RAM a query utilizes.

We also introduce the provenance latency metric, to quantify the event time it takes for $P(Q, \omega)$’s components to become available:

- For tuple $t_i$’s vertex $V$, it is computed as $\tau_V - t_i$. 
- For an edge $E$, it is computed as $\tau_E - \max(\tau_V^S, \tau_V^K)$, where $\tau_V^S$ and $\tau_V^K$ are the timestamps of the vertices connected by the edge. 
- For the label $L$ of some vertex, it is computed as $\tau_L - \tau_V$.

Implementation requirements. We want Ananke to be a streaming-based extension of (Q) that delivers the edges, vertices, and labels of $P(Q, \omega)$ through its output stream(s). A solution can rely on user-defined or native operators (§2). Being a streaming-based extension of Q, the latter’s watermarks are propagated to Ananke operators, too. For generality, we assume a user-defined operator needs to support two methods (not invoked concurrently by the SPE): on_tuple($t$), invoked upon reception of tuple $t$, and on_watermark($W$), invoked when the watermark $W$ is updated.

4 DISCERNING ALIVE AND EXPIRED TUPLES

Live provenance needs to discern alive from expired tuples. Even if sink tuples cannot contribute directly to other tuples, source tuples can be inactive but alive. As we show, alive and expired tuples can be separated using static query attributes and the Sink watermarks.

Figure 5 illustrates how a source tuple’s contribution “ripples” through event time for a query composed of Aggregates $A_1$ and $A_2$, Map $M$, and Filter $F$. A source tuple, $t_S$, falls into two windows of $A_1$, which emit two outputs that pass through $M$ and contribute to three separate windows in $A_2$. Two of the three output tuples of $A_2$ get dropped by $F$, and a single tuple $t_{K}$ arrives at $K$. When that happens, it is unknown whether $t_S$ will contribute to more sink tuples - for example, it is uncertain if $F$ will drop the next output of $A_2$. The figure also explores $t_S$’s hypothetical maximal contributions: We ignore exact window placements and examine the extreme case, tuples always falling at the beginning or the end of windows. We shade all event times that can contain (direct or
indirect) contributions of $t_S$. As shown, the growth of the shaded region only depends on the window size of stateful operators. Tuple $t_S$ can contribute to any tuple inside the shaded region. Thus, if $K$’s watermark falls after that region ($W_2$ in the example), then $t_S$ is surely expired. $U_K$ denotes the width of the shaded region. With this intuition, we proceed with proving the following theorem:

**Theorem 4.1.** Given $\mathbb{K}_Q$ (Definition 3.3), it is possible to statically compute constants $U_K$, one for each Sink $K \in \mathbb{K}_Q$ so that: If a source tuple $t_S$ has contributed to a sink tuple $t_K$ that arrives at $K$ at time $\omega$ or later, then:

$$t_S: \tau \geq W^\omega_K - U_K. \quad (4.1)$$

Inversely, if $t_S: \tau < \min_K(W^\omega_K - U_K)$, then $t_S$ is expired and cannot contribute to any sink tuple fed to $\mathbb{K}_Q$ at time $\omega$ or later.

We move bottom-up towards the proof. First, we study pairs of chained operators, each with one downstream peer in Lemma 4.1. In Lemma 4.2, we focus on longer operator chains before examining arbitrary paths between sets of operators in Corollary 4.1 and finally concluding with the proof of Theorem 4.1. Here, we use the term operator in a broad sense, also to refer to Sources and Sinks.

**Lemma 4.1.** For any operator $O_1$ with downstream operator $O_{i+1}$, and a tuple $t_{i+1}$ arriving at $O_{i+1}$ at time $\omega$ or later, it holds that if an input tuple $t_i$ of $O_i$ contributed to $t_{i+1}$, then $t_i: \tau \geq W^\omega_{i+1} - W_{S_1}$.

**Proof of Lemma 4.1.** From the way stateful operators set their output timestamps ($\S 2$), we obtain $t_i: \tau \geq t_{i+1}: \tau - W_{S_i}$. Furthermore, from Definition 2.1 we get $t_{i+1}: \tau \geq W^\omega_{i+1}$. Thus, Lemma 4.1 follows immediately and it also holds for a stateless $O_i$ by setting $W_{S_1} = 0$. \hfill $\Box$

We now expand the proof to chains of operators.

**Lemma 4.2.** For any chain of operators $O_1 \ldots O_n$, their downstream operator $O_{n+1}$, and a tuple $t_{n+1}$ that arrives at $O_{n+1}$ at time $\omega$ or later, it holds that if an input tuple $t_i$ of $O_i$ contributed to $t_{n+1}$, then $t_i: \tau \geq W^\omega_{n+1} - \sum_{j=1}^{n} W_{S_j}$.

**Proof of Lemma 4.2.** We begin by showing that:

$$\forall t_1, t_{n+1}. \quad t_1 \implies t_{n+1} \implies t_i: \tau \geq t_{n+1}: \tau - \sum_{j=1}^{n} W_{S_j}. \quad (4.2)$$

Let us denote as $t_i$ tuples arriving at operator $O_i$, with $t_i \implies t_{i+1}$ for $i \in [1, n + 1]$. From the proof of Lemma 4.1, we know that $t_1: \tau \geq t_{2}: \tau - W_{S_1}$. Plugging in this relation again into the first term on the right-hand side of the inequality, we obtain $t_1: \tau \geq t_2: \tau - W_{S_1} \geq t_3: \tau - W_{S_2} - W_{S_1}$. Performing this step $n$ times yields Equation 4.2. Given Definition 2.1, $t_{n+1}: \tau \geq W^\omega_{n+1}$; hence:

$$\forall t_1, t_{n+1}. \quad t_1 \implies t_{n+1} \implies t_i: \tau \geq t_{n+1}: \tau - \sum_{i=1}^{n} W_{S_i} \geq W^\omega_{n+1} - \sum_{i=1}^{n} W_{S_i}. \quad (4.3)$$

5 ALGORITHMIC IMPLEMENTATION

As mentioned in §2, SPEs provide native operators and support user-defined ones. Here we show how Ananke’s goals can be met by a user-defined operator (ANK-1, §5.1) or by composing native operators (ANK-N, §5.2). While ANK-1 targets the prompt final labeling of $F(Q, \omega)$’s vertices, ANK-N shows how the APIs for parallel execution commonly provided by SPEs are sufficient to parallelize Ananke’s algorithm. We study their trade-offs in §6.

Both in ANK-1 and ANK-N, the ID of $t_S$’s source vertex is based on $t_S$’s attributes. Hence, source tuples with equal attributes refer to the same source vertex. As each sink tuple represents a unique event, it results in a sink vertex with a unique ID. Since each sink tuple can carry each source tuple at most once in its provenance, edges are also unique. We discuss in §5.3 how to extend Ananke to other ID policies. In the following, we make use of Remark 4.1 for distinguishing alive from expired tuples. As mentioned in §3, the set of streams $\mathbb{B}_Q$, delivering backward provenance to Ananke, forwards the required watermarks. For both implementations, we show how they meet the requirements for vertices (R-V), edges (R-E), and labels (R-L) from §3.

5.1 ANK-1: Single User-defined Operator

As introduced in §3, user-defined operators must support two methods: on_tuple and on_watermark. Algorithm 1 covers such methods for ANK-1: methods unique_id() and get_id() respectively generate a unique ID and compute the ID of $t$ based on its attributes.

**Claim 5.1.** A user-defined operator fed $\mathbb{B}_Q$ can correctly deliver $F(Q, \omega)$ with Algorithm 1’s on_tuple and on_watermark methods.

**Proof.** Upon reception of sink tuple $t_K$ from $\mathbb{B}_Q$, ANK-1 emits exactly once the corresponding (1) sink and (2) source vertex, only if its ID was not stored in the timestamp-sorted set $T$ (i.e., if the corresponding source vertex was not forwarded before), thus meeting requirement (R-V); (3) edges, and (4) sink vertex label (L1-11). Set $T$ represents ANK-1’s “memory” about forwarded source vertices. The emitted tuples carry as timestamp the current watermark value,
Algorithm 1: ANK-1 algorithmic implementation

Data: Set T of pairs (u, ID), ordered on u, and watermark W

1. Method on_tuple(tk)
   2. IDK = unique_id()
   3. emit((W, [IDK, tk]));
   4. sourceTuples ← get_provenance(tk);
   5. for i: sourceTuples do
      6. IDti = get_id(ti);
      7. if (u, IDti) ∈ T then
         8. emit((W, [IDti, tk]));
         9. T ← T ∪ {(t, IDti)};
      10. emit((W, [IDti, tk]));
      11. emit((W, [IDti]));

Method on_watermark(W)

12. for (u, IDu) ∈ T do
13.   if u = W - U then
14.      return;
15.   emit((W, [IDu]));
16.   T ← T \ (u, IDu).

5.2 ANK-N: Native Operator Composition

We now present ANK-N, based on native operators. First, we study the case of U > 0. For ease of exposition, we initially rely on two auxiliary stateful operators, Delay (D) and Forward Once (FO), that help meet the requirements, and later show how D’s and FO’s semantics can be satisfied by native operators. Finally, we cover the case U = 0, where all Q’s operators are stateless.

A Delay (D) operator produces, for each input tuple t with a unique payload φ, an output tuple t’ as a copy of t with t’ = delay(t, τ = delay(τ, u) + 2) · U, τ’ = φ, and U < t’ - τ ≤ 2U.

A Forward Once (FO) guarantees that, whether one or more tuples are fed to it sharing the same ID sub-attribute, only the earliest such tuple is output, with its payload unchanged but its timestamp delayed by delay(τ) as in D. After such tuple is output, FO produces nothing for the subsequent input tuples with that ID until a period of U has passed in the input stream. As identical source tuples that appear at different times in BQ are not spaced apart further than U (Remark 4.1) and have the same ID, FO will output unique source tuples exactly once.

ANK-N overview: Using D, FO and native operators, we construct the DAG of Figure 6 to meet the requirements from §3, with BQ as input. First, we outline the main idea. Let us consider sink tuple tk from BQ, and follow its path through the DAG.

Upon processing tk, Mn produces a sink vertex that carries a copy of tk, a unique IDK and the character “K” as sub-attributes of its payload. Mn also produces, for all source tuples in tk’s provenance, a source vertex with character “S” (carrying an ID based on the source tuple attributes) as well as an edge. Each edge carries the character “E” and the IDs of the source and sink tuples that it connects. In the proof of the following claim, we continue tracing the paths of “K”, “E” and “S” tuples and show that all requirements for delivering a live provenance graph are met.

Claim 5.2. The DAG in Figure 6, using the D and FO operator, as well as native ones with the mapping function Mn defined in Algorithm 2, once fed BQ, correctly delivers F(Q, ω).

Proof. We first prove that for each source tuple ts, its vertex, edges, and label are delivered correctly:

(1) Ensuring ts’s vertex is created once. ts can appear multiple times, as provenance of multiple sink tuples. Based on Theorem 4.1, after contributing to sink tuple tK (timestamped tK), tk cannot contribute to later sink tuples timestamped ≥ tK + U. Thus, no pair of source tuples with the same ID can be farther away than U.

As source vertices (marked with “S”) are forwarded to FO, which is defined to output each source vertex with a given ID exactly once with ts = delay(tK), (R-V) is met for source vertices.

(2) Ordering ts’s edges behind the vertex. For every ts in the provenance of tk, Mn produces the connecting edge “E”. For these edges to come after ts’s vertex, they are forwarded by F1 to D1, which outputs copies of each edge, with tK = delay(tK), meeting (R-E).

(3) Producing ts’s label correctly. The latest edge E involving ts could be produced by Mn at event time tK + U = e (for e > 0), according to Remark 4.1. As E’ will be delayed to tK + U = e the source label tuple must be delayed beyond tK to meet (R-L). From FO, the source vertex (which has been delayed already) is multiplexed to D2, delayed again, and mapped by M2 to a label tuple with timestamp tK + delay(delay(tK)) = delay(tK) + 2U which is strictly greater than tK, meeting (R-L) for source tuples.

Thus, the components involving ts are meeting the requirements.

We now focus on sink tuple tk’s vertices, edges, and labels:

(1) Ensuring tk’s vertex is created once. F2 forwards the single instance of tk’s vertex (timestamp tK), meeting (R-V) for sink tuples.

(2) Ordering tk’s edges behind the vertex. As explained in (2) above, edges involving tk are delayed to tK = delay(tK) and (R-E) is met.

(3) Producing tk’s label correctly. The label for tk’s vertex must not

Algorithm 2: Map function ℋMn of Mn

1. def out-ℋMn(tk):
2.   IDK = unique_id();
3.   out.add([‘K’, tk, IDK]);
4.   for ts in get_provenance(tk) do
5.     IDti = get_id(ti);
6.     out.add([‘S’, ts, IDti]);
7.     out.add([‘E’, IDti, IDK]);
8.   return out;
have a lower timestamp than any edge connected to \( t_K \)'s vertex, and these edges are delayed. As the sink vertex "K" is multiplexed from \( F_2 \) to \( F_2 \) and mapped to a label by \( M_L \), the resulting label has timestamp \( t_{K,L} = \text{delay}(t_K) = t_E \), meeting (R-L) for sink tuples.

Thus, the components involving \( t_K \) also meet the requirements.

Lastly, we now show how Aggregate \( A \) emits vertices, edges, and labels in order. Each tuple, timestamped \( t \), falls in one window \( [t, t + \delta] \) of \( A \), and a copy of each tuple is produced by \( A \) when \( A \) receives a watermark > \( t + \delta \). As Aggregates emit results in timestamp-order (§2), this effectively sorts \( A \)'s outputs. Thus, ANK-N correctly delivers live provenance, meeting the requirements in §3. □

We now construct \( D \) and \( FO \) using native operators:

**Delay.** An Aggregate \( A \), Filter \( F \) and Map \( M \) (as in Figure 7) can enforce this operator’s semantics. \( A \) and \( F \) create the delay, while \( M \) restores the input tuple’s payload, creating a delayed copy of it. As \( A \)'s window size is twice as big as its window advance and \( KB : \phi \), any input tuple with a unique payload falls into two windows and contributes directly to two output tuples of \( A \), with timestamps spaced \( U \) apart. As each output tuple \( t' \) of \( A \) carries the timestamp of its corresponding input tuple \( (t_{\text{orig}}) \), the delay induced on the input tuples can be computed as \( t' = t - t \). \( F \) ensures that this delay is greater than \( U \), which is always the case for exactly one of the two tuples produced by \( A \) for each input tuple - the other is delayed at most \( U \) and thus discarded. In the extreme case, an input tuple \( t \) can be delayed by \( 2U \) (the window size), namely if \( t \) coincides with a window’s left boundary. The window advance dictates that output tuples produced from \( t \) have timestamps spaced \( U \) apart. Thus, there will also be a tuple delayed by \( U \) produced by \( A \) - however, this tuple will be discarded by \( F \). This earlier output tuple, in all other cases where \( t \) does not coincide with a window’s left boundary, will be delayed even less, and thus also discarded. From this discussion, it is also apparent that the delay for two input tuples \( t_1, t_2 \) is identical (as both \( t_1 \) and \( t_2 \) are equally distanced from the left boundaries of the windows they fall in).

**Forward Once.** \( FO \) ensures that from a group of tuples \( t_1, \ldots, t_n \) with increasing timestamps, common key \( K \) and \( t_n, t - t_1, t < U \), exactly one tuple \( t_{FO} \) with payload \( t_n, \phi \) is produced. This tuple is delayed from \( t_1 \) by at least \( U \) and at most \( 2U \). Figure 8 shows how \( FO \) can be constructed using two Aggregates \( A_1 \) and \( A_2 \) and a Join \( J \), to satisfy the required semantics. \( J \)'s predicate is defined as:

\[
\text{FPredicate}(r, s) = ((r.\tau \leq s.\tau) \land (r.\phi[2] \leq s.\phi[2])) \lor (5.1) (s.\tau \leq r.\tau) \land (s.\phi[2] \leq r.\phi[2])),
\]

where \( \phi[2] \) is the count emitted by the Aggregate operators.

The group of input tuples to \( FO \) are multiplexed to both \( A_1 \) and \( A_2 \). Since \( t_n, t - t_1, t < U \), and the windows of \( A_2 \) are offset by \( U \), \( t_1, \ldots, t_n \) will land in either (1) two windows of one Aggregate and

**Figure 7: Composition of the D operator.**

**Figure 8: Composition of the FO operator.**

one window of the other, or (2) in exactly one window in both Aggregates. Figure 9 exemplifies how \( FO \) achieves the required exactly-once forwarding for both cases. \( A_1 \) and \( A_2 \) produce output tuples that carry the common payload of the input tuples and the count \( c \) of input tuples per window. Their window alignment guarantees that a tuple produced by \( A_1 \) or \( A_2 \) lands in exactly two of \( J \)'s windows. Let us now explain the two different cases in detail:

- If \( t_1, \ldots, t_n \) fall in one \( A_1 \) and \( A_2 \) window, output tuples \( r \) and \( s \) are then fed to \( J \). Since \( r.\tau < s.\tau \) and \( r.\phi[2] = s.\phi[2] \), predicate 5.1 holds. When \( J \) emits \( t_{FO} \), it holds that \( t_1, t + U < t_{FO}, t \leq t_1 + 2U \).

- If \( t'_1, \ldots, t'_n \) fall in two \( A_1 \) windows and one \( A_2 \) window, output tuples \( r'_1, r'_2, s' \) are fed to \( J \). Then, two windows of \( J \) have one tuple each from \( A_1 \) and \( A_2 \); however, predicate 5.1 holds only for the earlier of the two windows, in which \( r \) has lower timestamp and lower count \( c \). Also in this case, \( t'_1, t + U < t'_2 \). \( t' \leq t'_1 + 2U \).

Thus, in both cases, the input is deduplicated and delayed, and the timestamp of the output tuple \( t_{FO} \) is given by \( \text{delay}(t_1/t'_1) \).

**Corner case.** If all operators in \( Q \) are stateless, then \( U = 0 \). This corner case is not covered by the above implementation of ANK-N, as \( D \) and \( FO \) cannot have \( WS = 0 \). If \( U = 0 \), each sink tuple and its single provenance source tuple have the same timestamp; thus, source tuples are immediately expired once event time passes beyond their timestamp. Each sink tuple will be contributed to by a single source tuple; however, the latter could contribute to several sink tuples. One approach for \( U = 0 \) is to replace \( D \) and \( FO \) with identity Maps. The final Aggregate \( A \) will then deduplicate source vertices (and source labels, which could now be duplicated as well), as they share the same payload and fall into the same window.

\( ^6 \) The case in which \( t_1, \ldots, t_n \) fall in one window for both \( A_1 \) and \( A_2 \) but \( A_1 \)'s window ends later than \( A_2 \)'s one, and the case in which \( t'_1, \ldots, t'_n \) fall in two \( A_1 \) windows and one \( A_1 \) window are given by “swapping” \( A_1 \) and \( A_2 \).
5.3 Extensions

Ananke associates each sink tuple with a dedicated sink vertex. Hence, our implementations do not need to deduplicate sink vertices, edges, or sink vertex labels. Modifying Ananke to allow distinct sink tuples to refer to the same sink vertex and perform that deduplication is nonetheless trivial, as it simply requires to store the sink vertex IDs which have already been forwarded (ANK-1) or to replace the D operators of Figure 6 with FO operators (ANK-N).

6 EVALUATION

We study Ananke’s performance relative to the state-of-the-art framework GeneaLog [30]. In §6.1, we compare the performance of queries (1) without provenance, (2) with GeneaLog’s backward provenance, and (3) with Ananke’s live, forward provenance (ANK-1 and ANK-N, cf. §5). In §6.2, we evaluate the provenance latency for the same use-cases. In §6.3, we compare ANK-1 and ANK-N in-depth, studying their performance for various configurations. Finally, in §6.4 we highlight Ananke’s strengths in comparison to ad-hoc implementations relying on tools external to the SPE.

Ananke Implementation. Ananke is implemented in Java in Flink [8]. It instruments the queries without altering the SPE and uses GeneaLog for backward provenance. We extended GeneaLog to handle tuples arriving at windows of stateful operators out of timestamp order (e.g., when there is parallelism). Moreover, GeneaLog requires operator and tuple objects to inherit provenance-specific code. This non-transparent (or optimized) implementation can introduce a non-negligible development and maintenance overhead, as implementations need to be altered tying the query implementation to the provenance framework. Here we introduce an alternative transparent implementation (denoted by suffix /T), which is based on encapsulation: The query is decoupled from the provenance capture, which can be enabled through an automated process.

Evaluation Setup. To account for the broad range of modern CPSs’ devices, we use (1) Odroid-XU4 [21] devices (or simply Odroid), mounting Samsung Exynos5422 Cortex-A15 2Ghz and Cortex-A7 Octa core CPUs, 2 GB RAM, running Ubuntu 18.04.2, OpenJDK 1.8.0_252, and Flink 1.10.0 (pinned to the four big cores); and (2) a single-socket Intel Xeon-Phi server with 72 1.5GHz cores with 4-way hyper-threading, 32KB L1 and 1MB L2 caches, 102 GB RAM, running CentOS 7.4, OpenJDK 1.8.0_161, and Flink 1.10.0. The execution environment is made explicit in each experiment.

We study the average throughput, latency, CPU and memory utilization (§3). For real-world use-cases (§6.1), we also study the provenance latency. These experiments are repeated at least ten times and are at least ten minutes long. Results are presented as averages with 95% confidence intervals between repetitions. Unless otherwise stated, the parallelism of all operators is set to one. We evaluate the scalability of ANK-N separately in §6.3.

6.1 Comparison With the State-of-the-art

To compare with GeneaLog, we study four queries from the domain of CPSs [30], targeting smart highways and smart grids, and four from smart vehicular systems. The latter are real-world examples from the automotive industry, with broader provenance characteristics, more operators, and larger data volumes. To show Ananke’s support for multiple Sinks, we run queries from the same overhead and lower performance. We study both techniques and let the user choose between flexibility and performance.

Table 1: Query configurations explored in the evaluation.

<table>
<thead>
<tr>
<th>NP</th>
<th>GL</th>
<th>ANK-1</th>
<th>ANK-1/T</th>
<th>ANK-N</th>
<th>ANK-N/T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provenance</td>
<td>-</td>
<td>Backward</td>
<td>Live</td>
<td>Live</td>
<td>Live</td>
</tr>
<tr>
<td>Native Ops</td>
<td>-</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Transparent</td>
<td>-</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Figure 12: Real-world example queries: a) Vehicle Tracking queries. \( \tau, \text{lat, lon} \) give the timestamp, GPS latitude and GPS longitude of the car with ID \( \text{car_ID} \). b) Object Annotation queries. \( v_L, v_R \) are the camera images from the left- and right-facing camera modules; \( l \) is the LiDAR point cloud. \( \text{obj} \) is a concretely-bounded object within an image or a point cloud.

**Linear Road.** We run two queries from the Linear Road Benchmark [6] on an Odroid. The first query detects broken-down vehicles through consecutive reports of zero speed and constant position. The second detects accidents from cars stopped at the same position. Both queries receive reports from vehicles every 30 seconds, and contain Aggregates and Filters (we refer the reader to [30] for more details). Each sink tuple depends on 4 source tuples in the first query and 8 in the second. Figure 10 shows the query performance without provenance (NP), with GeneaLog (GL), and Ananke with the user-defined operator (ANK-1, ANK-1/T) or the native operators (ANK-N, ANK-N/T). The text in Ananke’s bars shows the percentage difference from GL. The performance impact of both GL and Ananke is small. GL results in about a 3% performance drop for rate and latency and Ananke causes a further drop of 2% for ANK-1 and ANK-N and up to 5.6% for the transparent variants (/T).

**Smart Grid.** Figure 11 shows the performance of two queries from the smart grid domain, run on an Odroid. The first reports long-term blackouts by identifying meters with zero consumption for 24 hours. The second detects anomalies, through meters that report abnormal consumption at midnight as compensation for the previous day. Both queries receive hourly power measurements and use Aggregates and Filters; the second also has a Join (we refer the reader to [30] for more details). On average, a sink tuple depends on 192 source tuples in the first query and 24 in the second. The provenance overhead is higher here since the queries have larger aggregation windows and higher volumes of provenance data. GL results in a 9% rate drop and 3% latency increase, while ANK-1 (/T) causes a further 2.7% (14.1%) drop in the rate and a jump of 2.6% (5.2%) in the CPU. For ANK-N (/T), the rate drops by 7.2% (16.8%) compared to GL and the latency rises by at most 2.5%, ANK-N’s higher number of operators causes a jump in the CPU, around 14%.

**Vehicle Tracking queries.** This use-case is based on Figure 1. It uses the GeoLife dataset, composed of 18670 GPS traces of various vehicles over 4 years around Beijing [41]. We employ 10046 traces of cars driving a full day each to simulate a large fleet driving simultaneously. Figure 12a shows the queries, fed tuples carrying the car ID, timestamp, and latitude/longitude. \( Q_1 \) calculates the immediate and average speed of the last two minutes per car, and forwards to \( K_1 \) tuples with average speed \( \bar{v} > 70 \text{km/h} \). \( Q_2 \) forwards

Figure 13: Performance - Vehicular Tracking queries.

Figure 14: Performance - Object Annotation queries.
events with more than 9 coordinates within area $A$, a region around Yuyuantan Park in Beijing, to $K$. This experiment is performed on an Odroid receiving the input data via 100MBit/s Ethernet. Sink tuples of $Q_1$ depend on around 30-160 tuples and sink tuples of $Q_2$ depend on 25-250 tuples. As shown in Figure 13, the performance of Ananke follows the same trend as before. The impact on rate and latency is small for ANK-1 (/T), at 2-4.5% more than GL and higher for ANK-N (/T), up to 18.3%. The larger provenance graphs of $Q_1$ and $Q_2$ cause Ananke to have higher resource requirements, with the memory utilization almost doubling in some cases and the CPU utilization jumping by up to 57% in the worst case (ANK-N).

Object Annotation queries. These queries enrich an in-vehicle computer vision system based on LiDAR and two cameras. We use the Argoverse Tracking dataset [10], with 113 segments of 15-30s continuous sensor recordings of urban driving, plus 3D annotations of surrounding objects. The two queries, shown in Figure 12b, receive a stream of tuples carrying sets of annotations $O_{\text{LiDAR}}, O_{\text{cam,L}}, O_{\text{cam,R}}$, of objects found by the vehicle’s LiDAR and a left- and right-facing camera. These sets contain objects labeled with the type (e.g., “pedestrian”), 2D position, and a unique object ID. $M_1$ reproduces all objects found by the LiDAR, while $F_1$ forwards only bicycles found in front of the vehicle. $A$ and $F_2$ then forward a tuple to $K_1$ if a specific bike was in front of the vehicle for more than 11 frames during a 6s window. In $Q_2$, only tuples referring to pedestrians are forwarded as two streams to a Join. If, during $2s$, a certain pedestrian is found by both cameras, the pedestrian has crossed, and a tuple is forwarded to $K_2$. To simulate powerful, specialized vehicular hardware, this experiment was performed on the Xeon-Phi server. On average, $Q_1$’s sink tuples depend on 15-50 source tuples whereas $Q_2$’s ones depend on 2. The tuples of these queries are much bigger than all previous use-cases, in the order of kilobytes instead of bytes. As evident by the performance of NP in Figure 14, these queries are much more demanding. For example, GL drops 21% in rate, mostly due to the large volume of provenance data transferred between the SPE tasks. Ananke has a small effect on the rate, causing a further drop of 3.9-6.6%. Latency is affected more, increasing by about 25% for ANK-1(/T) and 48% for ANK-N(/T), while remaining at small absolute values. Memory consumption does not change significantly compared to GL. The CPU grows for ANK-N(/T), similar to previous use-cases.

### 6.2 Provenance Latency

We now study the provenance latency (§3) for ANK-1 and ANK-N for the Linear Road (LR), Smart Grid (SG), Vehicular Tracking (VT), and Object Annotation (OA) queries. The results are shown in Figure 15 in multiples of $U$ (see §5), for vertices (Sink/Source), edges (Edge), and labels (Sink-L/Source-L) of vertices. As shown, ANK-1 generally has a lower provenance latency than ANK-N. The main difference is that, while ANK-1 can output Source almost immediately (and then store its ID to not output it again), ANK-N delays the production of Source by at least $U$ to avoid duplicates. This delay propagates to Sink-L (see §5.2). Also, ANK-1 can immediately emit Edge and Sink-L without any delay (see Algorithm 1). Both variants have low Sink latency as those are safe to emit immediately upon arrival of a sink tuple. The variance is close to zero, as the frequency of watermark updates, the only execution-dependent feature of provenance latency, does not change between repetitions.

### 6.3 ANK-1 vs. ANK-N Trade-offs

Here, we compare ANK-1 and ANK-N for different data characteristics and query configurations. The experiments, run on the Xeon-Phi server, use synthetic queries in which Sources feed Ananke (non-transparent) with pre-populated provenance graphs.

Figure 16 shows the performance for different provenance sizes ($x$-axis) and overlaps (bar colors). The former is the number of source tuples each sink tuple depends on, the latter is the percentage of shared provenance between subsequent sink tuples. ANK-1 has better performance and lower resource requirements than ANK-N, due to the simpler algorithm and single-task deployment. Both ANK-1’s and ANK-N’s performance drops as the provenance size increases, as more data is maintained and transferred between tasks. Larger provenance overlaps result in slightly better performance since fewer source vertices and labels are emitted.

Figure 17 studies ANK-N’s ability to take advantage of the scalability features of the SPE. The $x$-axis is the number of queries feeding data to Ananke, and the bars refer to different parallelism values of Ananke. The provenance size is 50, and the overlap 25%. As shown (in log scale), ANK-N outperforms ANK-1 for parallelism 4 or higher since ANK-1 does not support parallel execution. ANK-N’s resource consumption increases with parallelism, making it better suited for use-cases with more available resources. For both ANK-1 and ANK-N, larger provenance overlaps result in slightly better performance since fewer source vertices and labels are emitted.

---

**Figure 15: Provenance Latency in multiples of $U$.**

**Figure 16: Performance of the synthetic query for different overlaps and provenance sizes.**
ANK-1 and ANK-N, a higher number of queries results in a drop in performance, caused by the increased data flow.

6.4 Comparison with On-demand Techniques

ANK-1 and ANK-N are not the only ways to achieve the goals of §3. Here, we compare Ananke with ad-hoc alternatives relying on existing systems (external to the SPE) to produce the provenance graph on-demand. These alternatives can seem appealing due to their additional features, e.g. persistent storage of backward provenance. Thus, a comparison with them is crucial to understand the properties of the fully-streaming approach provided by Ananke.

We study alternatives based on database systems with varying performance and safety guarantees. The first, SQL-P, relies on an established relational database (PostgreSQL [33]), the second, SQL-I, on a fast, self-contained relational database running in-memory (SQLite [35]), and the third, NoSQL, on a non-relational database (MongoDB [27])

The SQL implementations adhere strictly to the goals of §3, whereas NoSQL follows a best-effort principle, without strict ordering guarantees (due to the concurrent accesses by several threads). In contrast with Ananke, the alternatives produce the (streaming) provenance graph on-demand. A thread polls the database periodically and performs the data transforms. We evaluate an aggressive (suffix /A) polling strategy (as frequently as possible), and a relaxed (suffix /R) one (polling every second).

We compare ANK-1 with the above alternatives for two real-world experiments (Smart Grid and Vehicle Annotation queries), as well as for a synthetic one. In addition to previous performance metrics, we also study the delivery latency of the provenance graph, to assess the benefits and drawbacks of different polling strategies. This metric expresses the delay (in wall-clock time) between a graph component (vertex, edge, “expired” label) being ready to be delivered and actually being delivered. We do not study memory consumption, as the usage of external systems with different storage mechanisms creates a non-uniform measurement environment.

Figures 18 and 19 present the performance of the Smart Grid and Object Annotation queries, respectively. The text inside the bars indicates the relative difference from ANK-1. The performance of the on-demand implementations ranges widely, with relaxed polling (/R) having better rate, latency, and CPU but more than one order of magnitude higher delivery latency. Aggressive polling (/A) lowers the delivery latency but severely degrades the other metrics, in most cases. SQL-I/A achieves the lowest delivery latency (up to 6.2x higher than ANK-1), whereas SQL-I/R has the least impact on the original queries (slightly better rate and latency than ANK-1, and at the price of 28x the delivery latency and 4.4x the CPU). SQL-I’s implementation closely resembles ANK-1, temporarily maintaining in-memory only unsent graph components, instead of persisting all backward provenance data like SQL-P and NoSQL. This similarity in SQL-I’s and ANK-1’s implementations explains their similarity in performance. NoSQL’s best-effort strategy results in a relatively low impact on the original queries but multiple orders of magnitude higher delivery latency than ANK-1, in most cases. SQL-P performs between SQL-I and NoSQL, with lower delivery latency than NoSQL but a much higher impact on the other metrics. This is expected, as SQL-P persists the backward provenance (unlike SQL-I), while also strictly adhering to the goals of §3, in contrast with NoSQL.

Both experiments indicate that ANK-1 significantly outperforms all studied alternatives in most or all metrics.
Table 2: Performance comparison between Ananke and on-demand implementations in the synthetic query.

<table>
<thead>
<tr>
<th></th>
<th>ANK-1</th>
<th>SQL-P</th>
<th>SQL-I</th>
<th>NoSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rate (t/s)</td>
<td>131.3</td>
<td>54.43</td>
<td>54.80</td>
<td>211.4</td>
</tr>
<tr>
<td>Latency (s)</td>
<td>1.03</td>
<td>2.86</td>
<td>2.82</td>
<td>0.62</td>
</tr>
<tr>
<td>Deliv. Latency (s)</td>
<td>0.07</td>
<td>2.97</td>
<td>2.98</td>
<td>25.10</td>
</tr>
<tr>
<td>CPU (%)</td>
<td>1.18</td>
<td>0.97</td>
<td>1.30</td>
<td>1.75</td>
</tr>
</tbody>
</table>

*The values for SQL-I and NoSQL do not reflect the steady-state performance, which is significantly lower. Refer to the text for more details.

Table 2 presents the aggregated results of the synthetic experiment, having a setup similar to Figure 16, with a provenance overlap of 0% and provenance sizes 10-100. The relative performance is comparable to the real-world experiments, with ANK-1 outperforming the alternatives overall. SQL-I seems to outperform ANK-1 in rate and latency but a detailed examination of the experimental data indicates that SQL-I can only sustain 40% of the measured rate without exhausting the memory of the system. As, in contrast to ANK-1, the on-demand alternatives lack a back-pressure mechanism, they can fetch backward provenance from the SPE faster than they can process it. This leads to a continuously-increasing provenance backlog (illustrated by the high delivery latencies). For an in-memory database like SQL-I, this is unsustainable. While backpressure could be added manually to the studied alternatives, this would be "reinventing the wheel" as such mechanisms are available out-of-the-box in Ananke, running inside the SPE.

Among the studied on-demand alternatives to Ananke, SQL-I performs best but is still disadvantaged by not being streaming-oriented. In contrast, ANK-1 has a much better sustained performance and does not have to maintain "raw" provenance. This enables the immediate forwarding of the forward provenance graph stream to a secondary ingesting system without keeping unnecessary state, saving space and computational resources.

Evaluation summary. Ananke has similar overheads to the state-of-the-art in backward streaming provenance while offering live, forward rather than simply backward provenance. Compared to [30], the best-performing implementations of Ananke incur less than 5% drop in the rate in all use-cases and less than 3% increase in latency in all but one use-case. The evaluation shows that Ananke is suitable in deployments of real-world applications requiring both efficient processing and live provenance capture. Alternative existing systems fall short in providing the graph timely and in a sustainable fashion suited to the data streaming paradigm.

7 RELATED WORK

Data provenance, extensively studied in databases [11, 13, 23], only recently started being focused in data streaming. Early such work [39] focuses on coarse-grained data stream dependencies. A finer-grained approach, in [40], produces time intervals which may contain provenance tuples. [24] focuses on minimizing the storage requirements of streaming provenance but produces approximate results and lacks support for some native operators (e.g., Join).

To the best of our knowledge, Ananke is the first to deliver live forward provenance. [14] presents a system for debugging streaming queries with integrated provenance capture and visualization. However, it focuses on one SPE [17] and slices of the execution (with the option to lazily create the complete query provenance). It requires users to declare relationships between input and output tuples and does not distinguish expired tuples. Likewise, StreamTrace [7], targeting the Trill SPE [9], assists the development and debugging of queries with data visualization, relying on provenance through instrumented operators and ad-hoc query rewriting. Ananke’s provenance graph allows creating similar visualizations for the end-to-end system provenance. GeneaLog is the state-of-the-art technique and framework for fine-grained provenance in streaming applications. It records and traces the provenance metadata while incurring a small, constant per-tuple overhead. In this work, we extend GeneaLog to support out-of-order data and use it as the provider of backward provenance for Ananke. Ariadne [19] is a similar framework for fine-grained streaming data provenance using instrumentation. However, it requires variable-length annotations and needs to temporarily store all alive source tuples (including those that do not contribute to any sink tuples) until they become expired. The authors hypothesize the use of static query analysis to discern alive and expired tuples, but without further details. As discussed in §3, Ananke can be adjusted for use with Ariadne or any streaming backward provenance framework.

8 CONCLUSIONS

We presented Ananke, a framework to extend streaming tools for backward-provenance and to deliver a live bipartite graph of fine-grained forward provenance. Ananke provides users with richer provenance information, not only specifying which source tuples contribute to which query results, but also whether each source tuple can potentially contribute to future results or not. This distinction can help analysts prioritize the inspection of the large volumes of events commonly observed when monitoring CPUs. We formally prove Ananke’s correctness and implement two variations (available in [18]) in Flink. Through our thorough evaluation, we show that Ananke incurs small overheads while being able to outperform alternatives relying on tools external to the SPE. Future work can address (1) finer-grained debugging and exploration by expanding Ananke’s live provenance to include intermediate tuples produced by query operators, also indicating whether such tuples could contribute to future results, and (2) exploring how Ananke’s theoretical foundations about alive/expired tuples can be used in fault-tolerant stream processing.

ACKNOWLEDGMENTS

Work supported by the Swedish Government Agency for Innovation Systems VINNOVA, proj. "AutoSPADA" grant nr. DNR 2019-05884 in the funding program FFI: Strategic Vehicle Research and Innovation, the Swedish Foundation for Strategic Research, proj. "FiC" grant nr. GMT14-0032, the Swedish Research Council (Vetenskapsrådet) proj. "HARE" grant nr. 2016-03800, and Chalmers Energy AoA framework proj. INDEED and STAMINA.