GeneaLog: Fine-Grained Data Streaming Provenance at the Edge

GeneaLog: Fine-Grained Data Streaming Provenance at the Edge

Dimitris Palyvos-Giannas
Chalmers University of Technology
Gothenburg, Sweden
palyvos@chalmers.se

Vincenzo Gulisano
Chalmers University of Technology
Gothenburg, Sweden
vinmas@chalmers.se

Marina Papatriantafilou
Chalmers University of Technology
Gothenburg, Sweden
ptrianta@chalmers.se

ABSTRACT

Fine-grained data provenance in data streaming allows linking each result tuple back to the source data that contributed to it, something beneficial for many applications (e.g., to find the conditions triggering a security- or safety-related alert). Further, when data transmission or storage has to be minimized, as in edge computing and cyber-physical systems, it can help in identifying the source data to be prioritized.

The memory and processing costs of fine-grained data provenance, possibly afforded by high-end servers, can be prohibitive for the resource-constrained devices deployed in edge computing and cyber-physical systems. Motivated by this challenge, we present GeneaLog, a novel fine-grained data provenance technique for data streaming applications. Leveraging the logical dependencies of the data, GeneaLog takes advantage of cross-layer properties of the software stack and incurs a minimal, constant size per-tuple overhead. Furthermore, it allows for a modular and efficient algorithmic implementation using only standard data streaming operators. This is particularly useful for distributed streaming applications since the provenance processing can be executed at separate nodes, orthogonal to the data processing. We evaluate an implementation of GeneaLog using vehicular and smart grid applications, confirming it efficiently captures fine-grained provenance data with minimal overhead.

CCS CONCEPTS

• **Information systems → Data provenance**; **Online analytical processing engines**;

KEYWORDS

Fine-grained data provenance, Edge architectures, Data streaming

ACM Reference Format:
Dimitris Palyvos-Giannas, Vincenzo Gulisano, and Marina Papatriantafilou. 2018. GeneaLog: Fine-Grained Data Streaming Provenance at the Edge. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3274808.3274826

1 INTRODUCTION

Data streaming is a winning paradigm for applications that need to process data through continuous queries, at appropriate levels of edge, fog and cloud architectures in digitalized systems. Besides possibilities for low latency, another critical purpose of stream processing is to *distill* information into *events*, reducing the amount of data to be maintained. When the events produced by a streaming application refer to unusual or critical situations, it is then desirable to *keep the source data* to trace, understand and deal with the cause of the problems, to replay the query or to develop learning structures for future situations and related purposes [3, 14]. This is enabled by *fine-grained data provenance*, which allows linking back each output (e.g., an alert in the presence of an accident [5]) with the source data that leads to it (e.g., the position reports of the cars involved). As also discussed in [16], this is essential for industry 4.0, smart cities, vehicular networks, and other cyber-physical systems' applications. In the remainder of the text, we use the terms fine-grained data provenance or simply data provenance interchangeably.

In state-of-the-art solutions, data provenance is achieved through operator instrumentation that enriches the tuples with provenance meta-data annotations [16]. These variable-length annotations are then used to trace back the source tuples contributing to each output event. For this to work, all source data must be stored temporarily, later discarding those tuples that did not contribute to an output event. Although several optimizations have been discussed for such an approach (e.g., provenance compression), the disadvantages of variable-length annotation-based techniques can become problematic and introduce prohibitive storage overheads for applications maintaining large states [36].

Challenges. Fine-grained data provenance is an intrinsically heavy operation that bounds the performance of a given application to the efficiency with which the data provenance information of the latter is maintained. Our goal is to minimize the provenance overhead, both for time-performance aspects (e.g., throughput and latency) as well as for memory requirements (e.g., temporal storage). This is all the more important in edge computing, due to the limited resources of the employed devices, e.g., when the dozens of gigabytes of data sensed every day by a modern vehicle [10] cannot be stored (or transmitted to a dedicated storage unit) until each piece of information is distinguished into contributing or non-contributing to an event.

Contribution. We propose GeneaLog, a new technique and framework for data provenance in deterministic streaming applications. GeneaLog provides several major novelties:

- It relies on small, fixed-size annotations that work for all standard data streaming operators, reducing the per-tuple memory overhead incurred for provenance.
- It leverages the memory management of the process to distinguish source tuples that contribute to the application output from the ones that do not, without requiring temporary storage of all source data.

Middleware '18, December 10–14, 2018, Rennes, France

- It further allows for a modular and efficient algorithmic implementation using only standard data streaming operators. This is particularly useful for distributed streaming applications since the provenance processing can be (i) executed at separate independent nodes, orthogonally to the data processing, and (ii) parallelized using existing techniques available for standard streaming operators.

We show the correctness of GeneaLog and evaluate it in a challenging context, namely an edge-processing environment, with applications for monitoring of unusual or critical situations such as accidents and anomalies, with a variety of data rates and operators in their queries.

We provide a fully implemented prototype of GeneaLog on top of the Liebre Stream Processing Engine (SPE): a lightweight SPE for edge-computing [27]. As we show in our evaluation, GeneaLog overcomes state-of-the-art techniques making fine-grained data provenance a reality for streaming application in edge computing and cyber-physical systems.

The rest of the paper is organized as follows. We introduce preliminary concepts in § 2. We provide a formal problem definition in § 3 and present GeneaLog's approach in § 4-6, evaluating it in § 7. We discuss related work in § 8 and conclude in § 9.

2 PRELIMINARIES

Streams and operators are the basic building blocks of a data streaming continuous query. A stream is an unbounded sequence of tuples sharing the same schema composed by attributes $\langle ts, a_1, \ldots, a_n \rangle$ (we refer to attributes ts and a_i of tuple t as $t.ts$ and $t.a_i$, respectively). Attribute $t.ts$ represents the time at which the tuple has been created. In a query, *source tuples* are delivered by *Sources*, analyzed by a Directed Acyclic Graph (DAG) of *operators* which can also produce new tuples (as described later in this section) and, eventually, delivered as *sink tuples* to *Sinks*.

When the tuples of each source stream are fed to the operators of a query in timestamp order (either because Sources deliver timestamp-sorted streams as in [6, 18, 26] or by leveraging sorting techniques such as [25]) and each operator produces timestamp-sorted output streams (merging in timestamp order its input tuples if the latter are delivered by multiple input streams, as discussed in [18–20, 35]) a query's execution is deterministic. In a nutshell, this is given by the fact that each processing step depends on the notion of time carried by the tuples themselves (attribute ts) and is affected neither by the latency incurred in transmitting tuples from an operator to another operator nor by the interleaving of tuples to an operator with multiple input streams. For the edge-related monitoring applications motivating our work (§ 1), determinism is crucial to identify the source data contributing to each output event unambiguously. For this reason, we assume in the following that the queries for which data provenance is provided run deterministically. We refer to [18–20] for a detailed discussion on determinism.

Queries are run by being deployed at one or multiple *SPE instances*. Existing SPEs use different naming conventions for such instances (e.g., *Worker* for Apache Storm [32] and *Task Manager* for Apache Flink [8]). Nonetheless, each SPE instance represents a single *process* in which threads share memory but maintain the tuples being processed in thread-local data structures, using queues to communicate with other threads. As typical in modern applications, we assume that the memory allocated to objects maintained by each process is freed when such objects are no longer accessible (directly or indirectly) by the processes' threads (either by garbage collection techniques or other memory reclamation techniques such as hazard pointers [28, 33]). When a query is run by multiple SPE instances, the latter can be located at the same physical node or distinct ones.

The standard operators provided by SPEs can be distinguished into stateless and stateful. Stateless operators process input tuples on a one-by-one basis. The standard *stateless operators* provided by SPEs such as [8, 16, 18, 32] are:

Map which produces one or more output tuples for each input tuple by selecting one or more of the input tuples' attributes, optionally applying functions to them.

Filter which is used to decide whether a certain tuple should be forwarded or discarded based on a condition.

Multiplex which copies input tuples to multiple out streams.

Union which merges multiple input streams into a single output stream. Since we assume operators enforce determinism, the Union operator merges timestamp-sorted input streams into a timestamp-sorted output stream, as discussed in [18, 20].

Differently from stateless operators, stateful operators output tuples that depend on multiple input tuples. The standard *stateful operators* provided by SPEs such as [8, 16, 18, 32] are:

Aggregate which maintains a *sliding time-based window* of size WS and advance WA of the most recent input tuples and aggregates them (e.g., with functions such as max, min or sum) possibly defining one or more group-by attributes (from the input tuples' schema) to aggregate together only tuples sharing the same value for these attributes.

Join which defines one left input stream (L) and one right input stream (R), and produces an output tuple combining and/or altering the attributes of tuples $t_L \in L$ and $t_R \in R$ for each pair of tuples $\langle t_L, t_R \rangle$ satisfying a given predicate while not being far apart more than a given window size WS (i.e., $|t_L.ts - t_R.ts| \leq \text{WS}$).

It should be noted that once deployed at one or more SPE instances, these operators are not necessarily mapped to dedicated threads. E.g., when a query defines three consecutive Filter operators, their conditions can be checked at the same time by a single thread *chaining* the operators, as done by [8], rather than by three dedicated threads whose per-tuple communication costs could be higher than the processing ones. Similarly, the semantics of different operators could be combined, for instance defining a routing operator that forwards input tuples to one or more output streams based on a set of conditions (i.e., by combining a Multiplex and several Filter operators). We clarify this to highlight that, by discussing the standard operators provided by an SPE rather than ad-hoc ones, our contribution holds even when the semantics of such standard operators are combined.

In the remainder, together with stateless and stateful operators, we assume that queries can include one or more:

Source creating the source tuples fed to the query.

Sink receiving the sink tuples produced by the query.

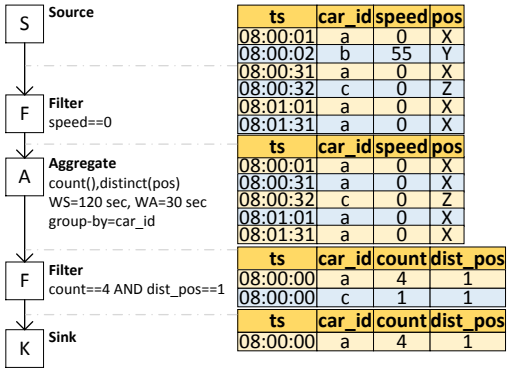

Figure 1: Sample continuous query.

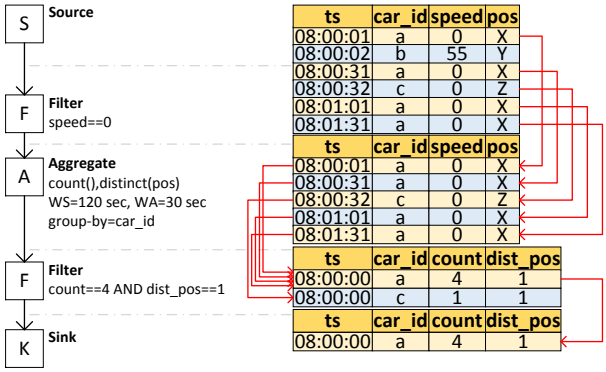

Figure 2: Sample query from Figure 1 with additional red arrows showing the contribution graph of the sink tuple.

Send and **Receive** operators, which can be used to transmit and receive tuples between two distinct processes (potentially deployed at distinct nodes).

Figure 1 presents a sample query that detects broken-down cars on highways (based on the Linear Road benchmark [5], further discussed in § 7). The source tuples are position reports, emitted by each car every 30 seconds, carrying information about its speed and position. A car is considered stopped if at least four consecutive position reports report zero speed and the same position. Three operators compose the query. First, a Filter forwards only the position reports that have zero speed. Subsequently, an Aggregate operator aggregates the position reports of each car individually over a time window of size and advance of 120 and 30 seconds, respectively. If four tuples in a window have the same position, the output tuple produced by the Aggregate is forwarded by a Filter.

3 PROBLEM DEFINITION

We first define the contributes-relation between source and sink tuples, setting the basis of fine-grained data provenance.

Definition 3.1. We say that input tuple t_{IN} to an *operator* OP *contributes* to an output tuple t_{OUT} of OP, if:

i OP is a Filter, Union or Receive and $t_{OUT} = t_{IN}$
ii OP is a Map, Send or a Multiplex and t_{OUT} is created upon the processing of t_{IN}[1]
iii OP is an Aggregate and if t_{IN} is in the window of tuples whose aggregation results in the creation of t_{OUT}
iv OP is a Join and t_{IN} is one of the tuples in a pair of tuples within time distance WS for which the Join predicate holds.

This relation is transitive and hence generalized in the context of a query as follows: a tuple t_{SOURCE} of a source stream *contributes* to a tuple t_{SINK} of a sink stream, if in the DAG of operators in the query there is a directed path of topologically-sorted operators $OP_1, \ldots, OP_i, \ldots, OP_k$, starting with a Source and ending with a Sink, and a sequence of tuples $t_0 = t_{SOURCE}, \ldots, t_{i-1}, t_i, \ldots, t_k = t_{SINK}$, such that for all $i = 1, \ldots, k$, t_{i-1} and t_i are input and output tuples of OP_i, respectively, and t_{i-1} directly *contributes* to t_i.

[1]The difference between type (i) and type (ii) operators in this context is that the former forward tuples, while the latter create new ones (even if sometimes they are identical).

A solution to fine-grained data provenance must, for each sink tuple t_{SINK} generated by a query, associate all the source tuples that contribute to t_{SINK}. In the sample query of Figure 1, the source tuples contributing to the sink tuple (08:00:00, a, 4, 1) are the tuples (08:00:01, a, 0, X), (08:00:31, a, 0, X), (08:01:01, a, 0, X) and (08:01:31, a, 0, X). This can easily be verified by retracing the process, as shown in Figure 2.

Besides correctness and efficiency (in time and space), the additional requirements to be fulfilled for fine-grained data provenance to be feasible in edge streaming applications are:

C1 A fixed-size per-tuple overhead when metadata is used to maintain provenance information.
C2 Avoid maintaining (in memory or storage) all source tuples to later differentiate them between contributing and non contributing to sink tuples.
C3 The possibility of implementing the provenance semantics by employing standard streaming operators, being thus able to leverage existing distribution and parallelization techniques for both the analysis run by a given streaming application and the analysis run to provide fine-grained data provenance for the latter.

For ease of exposition, we frame our discussion to a case in which a single query is being deployed to one or multiple SPE instances. It is nonetheless trivial to extend the discussion to scenarios in which more queries are defined.

4 LINKING SINK AND SOURCE TUPLES

In this section, we discuss GeneaLog's central idea, which allows it to maintain information about source tuples contributing to sink tuples with fixed-size per-tuple metadata. We describe in detail how provenance is provided for centralized and distributed deployments in § 5 and § 6, respectively.

The definition of the *contributes* relation (Def. 3.1), implies a *contribution graph* that connects each source tuple to the sink tuple(s) it contributes to. Figure 2 shows the contribution graph for the example of Figure 1. An approach for maintaining such a contribution graph, proposed by [16], is to assign a unique id to each tuple and to enrich each tuple with *meta-data* that carries over the ids of the source tuples contributing to it. In this way, contribution graphs

can be thought of as trees rooted at sink tuples with one leaf per contributing source tuple. This approach, nevertheless, conflicts with the motivating challenges discussed in § 3: (i) the list of ids carried by each tuple can grow arbitrarily and (ii) all source tuples need to be maintained (in memory or disk) until they are distinguished into contributing or not by inspecting the ids carried by sink tuples.

As we show in the following, GeneaLog's approach can solve both shortcomings, thus addressing challenge C1 (§ 3). For GeneaLog, the additional meta-data of each tuple consists of four *meta-attributes*: *Type* (T), *Upstream*$_1$ (U_1), *Upstream*$_2$ (U_2) and *Next* (N). For tuple t, the meta-attribute $t.T$ specifies which operator creates the tuple. The value of $t.T$ can be *SOURCE*, *MAP*, *MULTIPLEX*, *JOIN*, *AGGREGATE* and *REMOTE*. It should be noticed that no values are defined for operators that forward (instead of creating) tuples (e.g., the Filter operator), as we further elaborate in § 4.1. Meta-attributes $t.U_1$, $t.U_2$ and $t.N$ are memory pointers that can be used to access other tuples maintained by the streaming application. In a nutshell, they are used to (i) link each output tuple produced by an operator to the input tuples, processed by such operator, that contribute to it, and (ii) to traverse the resulting contribution-graph of each sink tuple, back to the source tuples contributing to it. We describe in detail in the following how these meta-attributes are set for the standard operators (§ 4.1) and how the contribution graph is traversed (§ 4.2).

4.1 GeneaLog's instrumented operators

Here we show how the meta-attributes T, U_1, U_2 and N are used by the instrumented Sources and operators listed in § 2, to maintain the contribution graphs connecting source tuples to sink tuples. Similarly to [16], we rely on *instrumented* operators and Sources that, besides running the analysis defined by their semantics can (i) access and modify the meta-data used for data provenance and (ii) use such metadata to create tuples that can be then forwarded to other operators in the query. The details of each instrumented operator are given here:

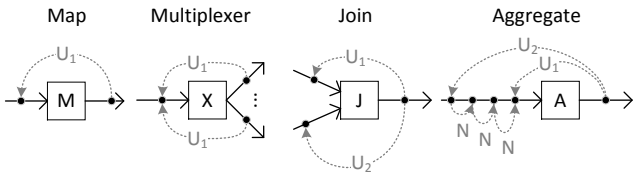

Figure 3: Visual representation of how the meta-attributes (pointers) U_1, U_2 and N are set by the instrumented Map, Aggregate and Join operators. For simplicity, we show for each operator only the meta-attributes that are set by it, thus ignoring dangling pointers.

Source. The Source of a query creates tuples that do not depend on other tuples. For this reason, GeneaLog's instrumented Source sets the meta-attribute T to *SOURCE* but does not set pointers U_1, U_2 and N.

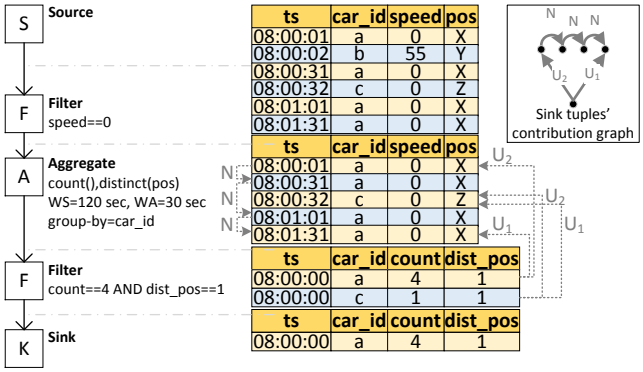

Figure 4: Sample query and execution from Figure 1 showing meta-attributes U_1, U_2 and N as set by GeneaLog's instrumented operators.

Map. The Map operator creates one or more output tuples for each input tuple it processes. With GeneaLog's instrumented Map, each output tuple t_O points to the input tuple t_I that contributes to it using the meta-attribute U_1, hence $t_O.U_1 = t_I$, as also shown in Figure 3. Additionally, attribute T is set to *MAP*.

Multiplex. The Multiplex operator creates a copy of each input tuple it processes for each one of its output streams. With GeneaLog's instrumented Multiplex, each output tuple t_O points to the input tuple t_I that contributes to it using the meta-attribute U_1, hence $t_O.U_1 = t_I$, as also shown in Figure 3. Additionally, attribute T is set to *MULTIPLEX*.

Join. As discussed in § 2, each output tuple t_O produced by a Join operator has exactly two contributing input tuples t_R and t_S. Without loss of generality, assuming T_R is more recent than T_S, GeneaLog's instrumented Join operator sets $t_O.T$ to *JOIN* as well as $t_O.U_1 = T_R$ and $t_O.U_2 = T_S$, as shown in Figure 3.

Aggregate. For the Aggregate, multiple input tuples can contribute to one output tuple. Based on Definition 3.1, all the input tuples that are part of the same window (and possibly group-by value) contribute to the output tuple produced when the window is full. If $t_1, \ldots, t_n$ are the input tuples that contribute to the output tuple t_O (being t_1 the earliest tuple), GeneaLog's instrumented Aggregate operator sets $t_O.U_1 = t_n$ and $t_O.U_2 = t_1$. Moreover, if $n > 1$, it also sets $t_i.N = t_{i+1}$ for $i = 1, \ldots, n-1$. Finally, it sets $t_O.T$ to *AGGREGATE*, as shown in Figure 3.

Filter and Union. Since Filter and Union operators do not produce new tuples, but rather forward existing ones across the query's streams, no instrumentation is defined for them.

Send and Receive. These operators are used to send tuples across distinct processes (possibly running at different physical nodes). From a semantics perspective, they do not create new tuples but rather forward existing ones across the streams of a query. From an implementation perspective, they indeed create new memory objects when sharing tuples across processes (optionally transmitting them through the network). The Send operator is instrumented so that the meta-attribute T is set to *REMOTE* if the latter is not

SOURCE, which allows distinguishing, locally in each process, tuples produced by operators deployed at other processes.

Figure 4 shows how the meta-attributes U_1, U_2 and N are set for the sample query and execution presented in Figure 1 by GeneaLog's instrumented operators. Moreover, it also shows the resulting contribution graph of each sink tuple.

4.2 Traversal of the contribution graph

Once the meta-attributes defined by GeneaLog are added and set according to the description of § 4.1, the contribution graph of each sink tuple can be traversed back to its contributing tuples, be them source tuples (when attribute T is set to *SOURCE*) or tuples produced by operators deployed at other instances (when attribute T is set to *REMOTE*).

It should be noticed that the meta-attribute U_1 (for tuples of type *MAP* and *MULTIPLEX*) together with U_2 (for tuples of type *JOIN*) allow traversing all their contributing tuples. For tuples of type *AGGREGATE*, on the other hand, the input tuples contributing to a given output tuple can be traversed using the meta-attribute N starting from the contributing tuple pointed by U_2 and ending at the contributing tuple pointed by U_1 (inclusive). Listing 1 presents the traversal algorithm, which implements a breadth-first search of the contribution graph of a tuple.

```
1  Set findProvenance(root):
2    Set provenance
3    Set visited
4    Queue q
5    q.addLast(root)
6    while (!q.isEmpty()):
7     Tuple t = q.removeFirst()
8     switch (t.type):
9      case SOURCE or REMOTE:
10      result.add(t)
11     case MAP or MULTIPLEX:
12      enqueueIfNotVisited(t.U1, q, visited)
13     case JOIN:
14      enqueueIfNotVisited(t.U1, q, visited)
15      enqueueIfNotVisited(t.U2, q, visited)
16      break;
17     case AGGREGATE:
18      enqueueIfNotVisited(t.U2, q, visited)
19      Tuple temp = t.U2.N;
20      while (temp != null && temp != t.U1)
21       enqueueIfNotVisited(temp, q, visited)
22       temp = temp.N;
23      enqueueIfNotVisited(t.U1, q, visited)
24   return result;
25
26 void enqueueIfNotVisited(tuple, queue, visited):
27   if (!visited.contains(tuple)):
28    visited.add(t)
29    queue.addLast(t)
```

Listing 1: Contribution graph traversal

To facilitate the presentation in the remainder, we introduce the term *originating tuple* in the following definition.

Definition 4.1. Tuple t' is an *originating tuple* of t if t' is returned by the provenance method in Listing 1 as a tuple contributing to t.

Using this definition, before proceeding in more detail with explanations about how provenance is guaranteed while meeting the challenges discussed in § 3, let us observe that when a query is entirely deployed within one process, all the originating tuples of a sink tuple are of type *SOURCE*; on the other hand, they can also be of type *REMOTE* when multiple SPE instances run the operators of the query.

5 INTRA-PROCESS DATA PROVENANCE

Given the instrumented operators and the contribution graph traversal approach discussed in § 4, we show here how challenges C2 and C3 presented in § 3 (i.e., to avoid maintaining all source tuples and to allow for provenance analysis to be implemented utilizing standard operators) are addressed by GeneaLog, when all the operators of a query are deployed within the process (i.e., the same SPE instance).

With respect to *challenge C2*, it can be noticed that by using memory pointers and by assuming, as discussed in § 2, that the memory used by objects such as tuples is freed when such objects are no longer accessible (directly or indirectly) by the processes' threads, GeneaLog can access all the information it needs, without maintaining or storing source tuples; this is so since the memory used by them will be referenced (and thus not reclaimed) as long as a reference to the sink tuple they contribute to exists. On the other hand, as soon as a tuple no longer contributes to any output or sink tuple, it will be dereferenced, and its memory can be reclaimed.

To facilitate the explanation of how GeneaLog addresses *challenge C3*, we first introduce an auxiliary term:

Definition 5.1. Stream U is the *unfolded stream* obtained from stream S if each tuple $t \in S$ is replaced by its originating tuples (see Def. 4.1) combined with t's attributes.

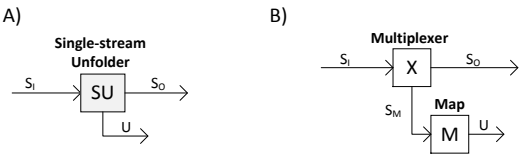

Figure 5: SU operator (A) and implementation of its semantics using the standard operators (B).

In the following, by considering the existence of a *single-stream unfolder* operator, whose semantics are described in Definition 5.2, we show that fine-grained data provenance can be achieved (Theorem 5.3), by enriching the query with such an operator. Next, in § 5.1, we show that the semantics of such a streaming operator can indeed be achieved utilizing the instrumented standard operators described in § 3, thus addressing challenge C3. Moreover, we prepare the ground for the explanation of GeneaLog's inter-process data provenance, later provided in § 6.

Definition 5.2. The SU (*single-stream unfolder*) operator (Figure 5A) has one input stream S_I and produces two output streams: the first one (S_O) is an exact copy of S_I and the second one (U) is the unfolded stream of S_I.

THEOREM 5.3. *A query in which an additional SU operator is added before each Sink (with S_O feeding the Sink) provides fine-grained data provenance through U.*

Proof. Since all the operators of the query are deployed within the same process, the unfolded stream U of each SU operator, for the Sink to which the SU operator is connected, contains originating tuples of type *SOURCE* only, and thus delivers a stream in which each sink tuple is combined with all the source tuples contributing to it, thus providing fine-grained data provenance. □

5.1 SU implementation using standard operators

Figure 5B shows how the semantics of the SU operator can be defined utilizing the instrumented operators provided by GeneaLog. As discussed in § 2, we stress that it is not necessary to map each of these operators to a dedicated thread (communicating with other threads and operators through shared queues). Efficient implementation can assign these operators to the same thread using chaining (e.g., as possible in Apache Flink [8]) or by implementing their semantics in a single user-defined operator.

As shown in Figure 5B, first a Multiplex operator can be used to duplicate the tuples of the input stream S_I and forward them to streams S_O (which will deliver S_I's tuples to the following Sink) and S_M. Then, a Map operator can be used, to unfold S_M to U by applying the `findProvenance` function (Listing 1) and produce, for each sink tuple t_{SINK}, a tuple carrying t_{SINK}'s attributes and the attributes of each originating source tuple of t_{SINK}.

6 INTER-PROCESS DATA PROVENANCE

In this section, we extend the provenance technique discussed in § 5 to setups where the operators of a query are deployed to multiple SPE instances.

A first observation we can make in this case is that relying solely on the process memory management to keep references to source tuples that contribute to sink tuples with the help of pointers is not sufficient since pointers assigned to tuples that are later forwarded across processes and dereferenced would be lost. Before explaining how GeneaLog's approach provides inter-process provenance, we first introduce some further notation and terms. We argue about correctness along with the presentation of our method, following the methodology of the previous section. I.e., first, by considering the existence of an additional (*multi-stream unfolder*) streaming operator, whose semantics are described in Definition 6.4, we show that fine-grained data provenance can be achieved. Next, we show that the semantics of this streaming operator can be implemented by composing standard operators described in § 3. In this way, besides proving correctness, we explain how GeneaLog meets challenge C3 (§ 3) in inter-process data provenance as well.

We refer to the n SPE instances in which a query is deployed as $\mathbb{V} = V_1, \ldots, V_n$. We say that V_i is a *source SPE instance* if the operators deployed inside it are fed by Sources also deployed inside it, but not from other SPE instances (i.e., if no Receive operators are deployed at V_i). We say V_i is a *sink SPE instance* if Sinks are deployed in it, and V_i does not contain Send operators. Finally, we say V_i is an *intermediate SPE instance* if it is neither a source nor a sink SPE instance. The *ordering value* of V_i is defined as the longest path in the graph of SPE instances from a source SPE instance to V_i. Let $\mathbb{V}_q$ denote the set of SPE instances having ordering value q.

For inter-processing provenance, we also assume tuples' constant-size meta-data is enriched by one additional meta-attribute *ID*, which is a unique id for each tuple[2]. We also use the terms *delivering stream*, *unfolded delivering stream* and *complete unfolded delivering stream*, defined here:

Definition 6.1. Stream S is a *delivering stream* if it feeds a Sink or is produced by a Send operator.

Definition 6.2. Stream U is the *unfolded delivering stream* obtained from the delivering stream S if each tuple $t \in S$ is replaced by its originating tuples (see Def. 4.1) concatenated with t's attributes. For each tuple t' in U, we refer to the originating tuple's attributes ID and ts that tuple t' carries as $t'.ts_O$ and $t'.ID_O$.

Definition 6.3. Stream U is a *completely unfolded delivering stream* if all its tuples are of type *SOURCE*.

The additional operator, called *multi-stream unfolder* (MU), is defined below and presented in Figure 6.

Definition 6.4. The MU (*multi-stream unfolder*) operator defines multiple unfolded delivering streams (see Def. 6) as input streams: (i) one *derived* and (ii) an arbitrary number of *upstream* unfolded delivering streams. Additionally, it defines one output stream. Each tuple t in the *derived* stream is forwarded to the output stream if it is of type *SOURCE*. Alternatively, it is replaced by the sequence of tuples $t_1, \ldots, t_i, \ldots, t_n$ found in any upstream stream for which $t_i.\text{ID} = t.\text{ID}_O$; this sequence is then forwarded to the output stream.

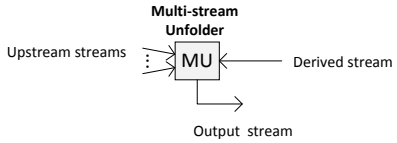

Figure 6: Representation of the input and output streams defined by the MU operator (Def. 6.4)

Finally, assuming that the SU operator presented in § 5 can be used to produce *unfolded delivering stream* (the additional setting of attributes *ts* and *ID* can be done by the Map operator in Figure 5), we can state the following theorem, that builds on complementing queries with SU and MU operators.

Theorem 6.5. *Given a query Q, let us define a query Q_E that is composed by the same operators defined by Q, plus (i) one SU operator preceding each Send operator and each Sink, (ii) one MU operator for each Send operator $\notin \mathbb{V}_0$ and for each Sink $\notin \mathbb{V}_0$ and (iii) any additional number of Send / Receive operator pairs according to how Q_E's extra operators are deployed.*

Let us connect these SU and MU operators so that:

- *each stream S_i feeding a Send operator or a Sink is unfolded into U_i by the corresponding SU_i,*
- *each stream $U_i \in \mathbb{V}_j | j > 0$ is fed as the derived stream to the corresponding MU_i,*

[2]For instance, composed by the unique id of the Source or operator producing the tuple and a sequential counter, as done by [16].

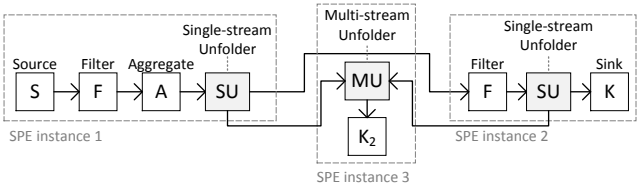

Figure 7: Distributed deployment of the sample query (Figure 1) with SU and MU operators for data provenance and an additional Sink K_2 to persist provenance data.

- *each stream U_i produced by an SU_i that precedes a Send operator is fed as an upstream stream to the MU operator of the instance to which S_i is delivered.*

Then, Q_E provides fine-grained data provenance for each Sink K in Q through: (i) the U_i stream produced by the SU_i preceding K (if $K \in \mathbb{V}_0$) or (ii) the stream O_k generated by the MU_k associated to K.

Proof. Any unfolded delivering stream U_i from an SPE instance V_j in $\mathbb{V}_0$ is complete since, for each tuple $t \in U_i$, t's contributing tuples are provided by the Sources deployed at V_j. Similarly, any stream O_i produced by an MU_i operator from an instance V_j in $\mathbb{V}_1$ is also complete since, for each tuple $t \in O_i$, t's contributing tuples can only be provided by Sources deployed at V_j or tuples forwarded by instances in $\mathbb{V}_0$, which are all of type *SOURCE* and can be found in the upstream streams of MU_i. By induction, any stream O_i produced by an MU_i operator from an instance V_j in $\mathbb{V}_l$ is also complete since, for each tuple $t \in O_i$, t's contributing tuples can only be provided by Sources deployed at V_j or tuples forwarded by the output streams of MU operators from instances in $\mathbb{V}_m | m < l$, which are all of type *SOURCE* and can be found in the upstream streams of MU_i. Delivering streams connected to Sinks $in\mathbb{V}_j | j > 1$ are thus unfolded by an SU operator and then fed as derived streams to MU operators that produce a sequence of tuples in which each sink tuple is combined with all the source tuples contributing to it, thus providing fine-grained data provenance. □

Figure 7 shows how SU and MU operators are added to the query presented in Figure 1 when its operators are deployed at three distinct SPE instances. For simplicity, Figure 7 and the figures shown in the remainder for query deployed at multiple processes do not show explicitly Send and Receive operators. In the following, we show how the MU operator can be implemented using GeneaLog's instrumented operators.

6.1 MU implementation using standard operators

The key operator needed to implement the semantics of the MU operator (Def. 6.4) is a Join. Considering the SPE ordering values as defined in this section, the Join in MU at a particular at each ordering value level processes unfolded streams of the previous and the current level of SPE instances, to extract useful information to be fed to downstream SPEs, in order to generate the source-and-sink tuple pairs belonging to the contributes relation (Def. 3.1). Figure 8 shows how the MU operator can be constructed, utilizing the standard operators described in § 2. For ease of explanation, let us first assume that the MU operator is fed by exactly one upstream stream S_D and by a single derived stream S_T, that does not contain any *SOURCE* tuples. In order to produce the output unfolded stream, the core semantics of the MU operator can be implemented using a Join operator, the predicate of which is used to match a tuple $t_D \in S_D$ and $t_T \in S_T$ if $t_D.ID == t_T.ID_O$ and whose window size is set to the sum of the window sizes of the stateful operators deployed at the instance producing S_T; the latter guarantees no tuple t_D is purged by the Join before a tuple t_T for which the Join's predicate hold has been processed.

As shown in the figure, if the MU operator needs to deal with multiple delivering unfolded upstream-streams, then a Union operator can be used to merge their tuples deterministically into one stream. If, moreover, tuples of type *SOURCE* can be found in S_T, a set of operators including one Multiplex operator, two Filter operators, and an additional Union operator can be used: as shown in the figure, the Multiplex operator is used to feed each tuple delivered by S_T into the two Filter operators F_1 and F_2. F_1 forwards tuples that are not of type *SOURCE* to the Join operator while F_2 forwards tuples of type *SOURCE* to the Union operator, which eventually merges them with the tuples produced by the Join operator into its output stream.

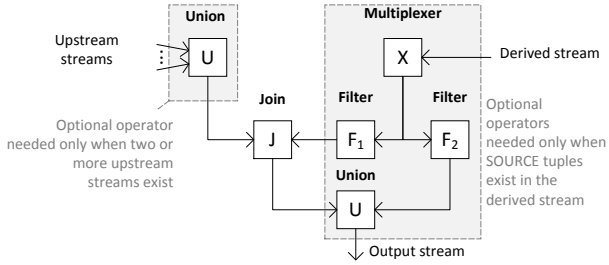

Figure 8: Implementation of the MU operator's semantics using the standard operators defined in § 2.

7 EVALUATION

In this section, we evaluate GeneaLog's performance. We first introduce the hardware and software setup. Subsequently, we present the different use cases we take into account. We then discuss the performance observed for these use cases when no fine-grained data provenance is maintained and when GeneaLog provenance capture is active, for deployments to a single process (to evaluate GeneaLog's intra-process technique, § 5) and to multiple processes run by different physical nodes (to evaluate GeneaLog's inter-process technique, § 6). For both deployments, we also compare our algorithm with an implementation of Ariadne, the current state-of-the-art in eager streaming data provenance [16] that is as general as ours and is targeted for use in stream processing systems. Finally, we provide a detailed analysis of the cost of traversing the contribution graph in each process in the intra-process and inter-process scenarios.

Hardware and software setup. For the hardware to resemble the embedded devices deployed in modern cyber-physical systems, the experiments are conducted on a network of 3 Odroid-XU4 [29] (or simply Odroid in the remainder), equipped with a Samsung

Exynos5422 Cortex-A15 2Ghz and Cortex-A7 Octa core CPUs and with 2 GB of memory. The Odroids are running a variant of Ubuntu 16.04.4 LTS and are using Java HotSpot(TM) Client VM 1.8.0_161-b12. They are connected to a 100Mbps switch. The experiments are performed using the lightweight Liebre SPE [27].

To evaluate GeneaLog, we consider the *throughput* (the average number of tuples per second that query can process), the *latency* (the average time interleaving the production of each sink tuple and the reception of the latest source tuple contributing to it), the *memory footprint* (the average and maximum size of memory used by the process running a given query) and the traversal time of the contribution graph of each sink tuple.

The provenance information of each sink tuple is calculated using the algorithm in Listing 1 and stored on disk. Note that in our experiments the total size of the provenance information is negligible compared to that of the source data (ranging from 0.003% to 0.5% of the latter). Although beyond the scope of our evaluation, each sink tuple's provenance information could also be forwarded to the end user (rather than stored) given its negligible impact on the overall network traffic. Experiments are at least six minutes long. Statistics are taken after a warm-up phase and before the cool down phase. Results are averaged over five runs and present the 95% confidence interval.

To compare with the state-of-the-art technique of Ariadne, we opted for a new implementation since the published one is based on the Borealis SPE [1], discontinued since 2008. As discussed in § 4, this technique, which we refer to as the baseline *BL*, annotates intermediate tuples with variable-length provenance metadata. In order to retrieve the actual provenance result, source streams are temporarily maintained and later joined with the annotated output streams.

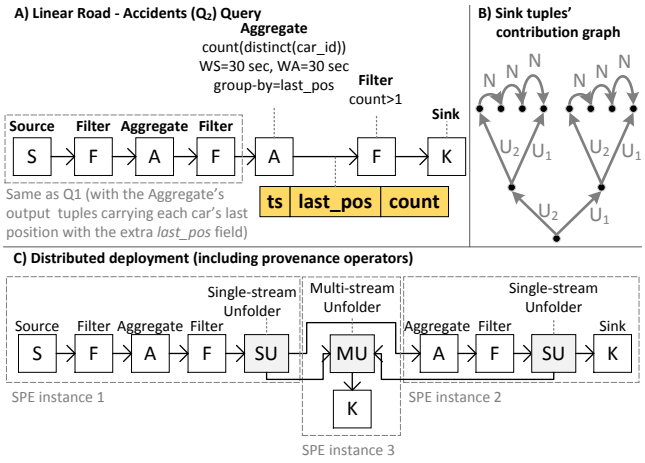

Figure 9: A) Query Q_2. B) Sink tuples' contribution graph, with 8 input tuples. C) Distributed deployment.

Use cases. We test GeneaLog with two queries from the vehicular network domain (using the Linear Road benchmark [5]) and two queries implementing real use case from a real-world Smart Grid infrastructure. As we discuss in the following, all the standard operators presented in § 2 are used by these queries. Also, the different queries are chosen to observe the overhead incurred by GeneaLog for different amounts of information (e.g., contribution graph size) needed to maintain provenance information.

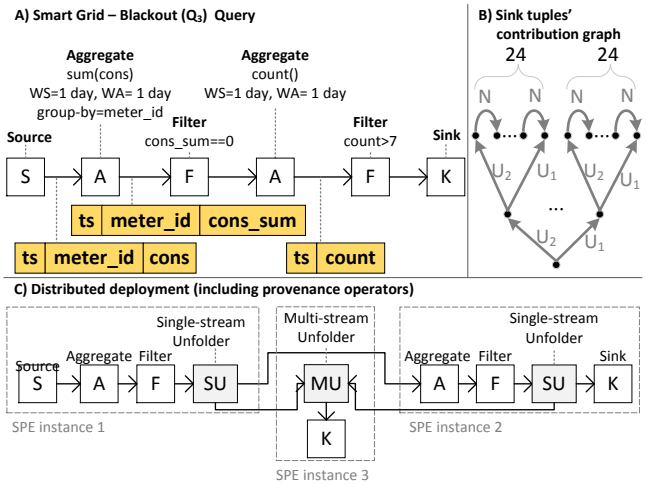

Figure 10: A) Query Q_3. B) Sink tuples' contribution graph, with 192 input tuples on average. C) Distributed deployment.

(Q1) - Detecting broken-down cars (Linear Road benchmark). The first use case we test is the one presented in § 2 (Figure 1), based on the Linear Road benchmark [5], an established standard to study SPEs' performance. It simulates vehicular traffic on a number of linear expressways, each composed of predefined *segments*. This is a representative example where stream processing in fog/edge architectures can result in extra benefits compared to processing in the cloud, as discussed in [11]. The generated data simulates the traffic of one highway since its volume is adequate for our evaluation.

As discussed in § 2, position reports are forwarded every 30 seconds by the cars traveling in the highway and carry the attributes[3] ⟨ts, car_id, speed, position⟩. A car is stopped if at least four consecutive position reports from the same car report zero speed and the same position.

(Q2) - Detecting accidents (Linear Road benchmark) - Figure 9. This second query extends Q1 to detect accidents. In the Linear Road benchmark, an accident is detected if at least two broken-down cars are found in the same position at the same time. This query defines the same operators as Q1[4] plus an additional Aggregate and an additional Filter operator. The former aggregates using the position as the group-by and a window of size and advance of 30 seconds. The resulting tuple carries the number of stopped vehicles observed for each position in the same time window. Subsequently, the Filter operator forwards only the tuples carrying a counter that

[3]Some unrelated attributes have been omitted to preserve clarity. We also use a single position attribute for ease of exposition (in the benchmark, positions are given by several attributes).
[4]With the Aggregate operator producing tuples with an extra `last_pos` field carrying the last position reported by each car.

is equal to or greater than 2. As for provenance, 8 source tuples contribute to each sink tuple.

(Q3) - Long-term blackout detection (Smart Grid) - Figure 10. This query aims at detecting blackouts in Smart Grid systems. Source tuples are measurements forwarded by smart meters every hour, with schema ⟨ts, meter_id, consumption⟩. Source data is grouped by meter, and the consumption is summed throughout each day by an Aggregate operator. A Filter forwards tuples with zero consumption to a second Aggregate, which counts them with a window of size and advance equal to one day. If there are more than seven meters which reported zero consumption for a whole day, then an alert is raised by the system. In this case, 192 source tuples contribute to each sink tuple on average.

(Q4) - Anomaly detection (Smart Grid) - Figure 11. This query aims at detecting faulty meters that show a suspiciously high consumption in correspondence with the beginning of a new day (i.e., reporting such value at midnight). Such behavior usually indicates meters are compensating for missing reported consumption about the previous day. The source tuples have the same schema as in Q3 and once again are forwarded every hour. The source stream is broken into two identical streams. The first one is sent to an Aggregate that is similar to the one in Q3, grouping by meter and calculating the daily consumption. The second stream is forwarded to a Filter which allows only the measurements done at midnight to pass through. The results of the Filter and the Aggregate with the same meter_id are joined using a window of one hour, and the consumption of the output is set as the absolute difference between the two inputs. Finally, another Filter produces an alert if the consumption difference is higher than a specified threshold. 24 source tuples contribute to each sink tuple.

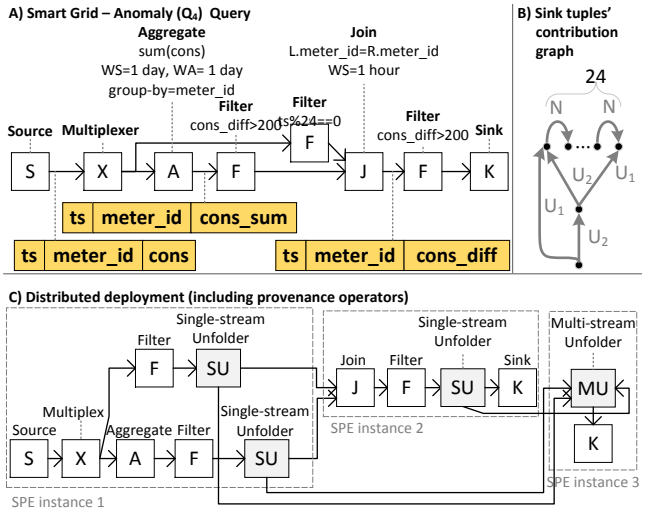

Figure 11: A) Query Q_4. B) Sink tuples' contribution graph, with 24 input tuples. C) Distributed deployment.

Intra-process performance. The results of the experimental evaluation of intra-process provenance are presented in Figure 12. Each

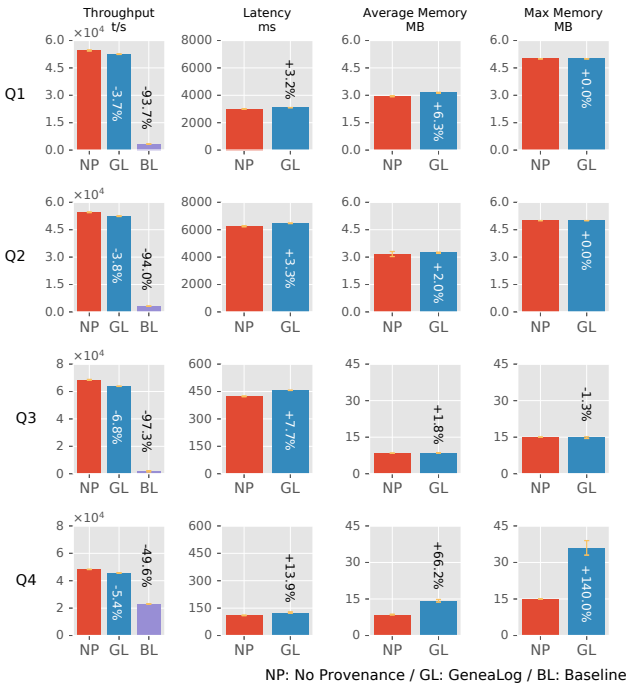

Figure 12: Evaluation of the impact of provenance computation for the intra-process provenance, when the use cases are deployed at a single process.

row of the graph refers to one query, and each column is a performance metric. As discussed later in this section, the performance degradation in the case of the baseline is so severe that we failed to record useful data for latency and memory consumption, thus forcing us to display only the throughput values.

In both the Linear Road Benchmark queries (Q1 and Q2), the throughput and latency overhead of GeneaLog's fine-grained provenance is less than 4%. The reduction in throughput is around 2000 t/s in both cases, and there is an increase in latency by 97 ms and 210 ms respectively. The change in the average memory consumption is less than 500 KB.

Results are similar for the Smart Grid queries. The blackout detection query (Q3) has a decrease of about 7% or 4680 t/s in throughput, and an increase of approximately 33 ms in latency when GeneaLog is enabled. Moreover, the actual memory consumed in general is so small that in the second query the maximum memory usage drops slightly (we believe this is because of the different behavior of Java's garbage collector). In the anomaly detection query (Q4) the throughput drops by 2590 t/s and the latency increases by about 15 ms. This is the case for which we observe the higher increase in memory consumption, 66%. However, the actual memory consumed remains very low, less than 2% of the total available memory of the device. The overhead in these two queries is slightly higher, which is expected since (i) they produce a more substantial number of events, and thus a higher volume of provenance data (almost two orders of magnitude more than the Linear Road Benchmark queries) and (ii) their windows are larger and thus contain more

tuples. The effect of windowing in memory consumption is more pronounced in Q4 since it uses two consecutive windowed operations and tuples that are candidates for provenance need to spend more time in memory until they can be safely garbage collected.

As far as the baseline is concerned, in all the queries except for Q4, its average throughput is an order of magnitude lower than GeneaLog. The bottleneck is given by BL's high usage of memory. Further, we observed that the throughput measured for BL keeps decreasing as the experiment progresses, reaching values close to zero, thus indicating that the processes are overloaded. Due to this, we do not report BL's data related to memory consumption (since this is always more than one order of magnitude higher, approximately) nor to latency (since it was not possible to get an accurate representation of the latency based on the limited number of sink tuples produced by the queries).

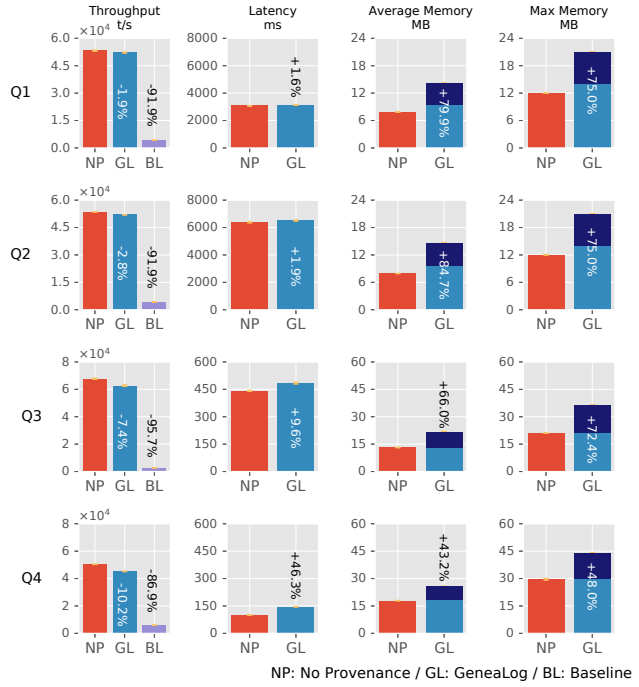

Figure 13: Evaluation of the impact of provenance computation for inter-process provenance, when the use cases are deployed at distinct processes and physical nodes.

Inter-process performance. Figure 13 presents the results of the same experiments for the inter-process case. In all distributed deployments, we use two Odroids for data processing and one more for recording the final provenance stream. Note that the memory consumption is measured as the sum of the consumption of each process. While the memory consumption in the processing nodes remains almost identical, the total memory consumption shown in the graphs is always higher due to the additional node. The memory consumption of the additional node is the darker-colored part at the top of the bars. As shown in the figures, the difference in memory consumption is mainly given by the additional node's memory consumption.

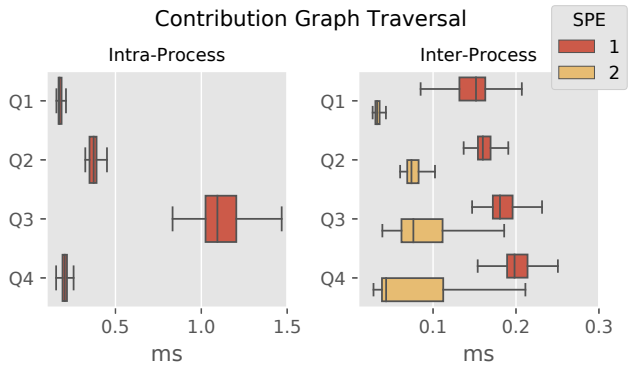

Figure 14: Time required to traverse the contribution graph for each output tuple.

The deployment of the two Linear Road queries (Q1 and Q2) is shown in Figures 7 and 9. The impact of GeneaLog's provenance in throughput and latency is less than 3%. More specifically, in the stopped vehicles detection query (Q1), there is a decrease in throughput of 1032 t/s and an increase in latency of 50 ms whereas these numbers for the accident detection (Q2) are 1483 t/s and 124 ms respectively. The memory consumption on the two nodes that process the actual data is almost identical, but as mentioned above, the cost of adding the additional node for provenance computation is reflected by the higher memory requirements.

The Smart Grid queries are deployed as in Figures 10 and 11. For the blackout detection query (Q3) there is a drop of 7.4% in throughput (-5000 t/s) and an increase of approximately 9.6% (+42 ms) in latency when GeneaLog is active. For the anomaly detection query (Q4) the impact is higher, around 5100 t/s (-10.2%) drop in throughput and 46 ms (+46.3%) increase in latency. The effect of the large window and the relatively high output event rate is even more prevalent in the distributed case because a larger number of tuples need to be serialized over the network to the third "provenance" node, negatively impacting throughput and latency as a result.

The behavior of BL, in this case, is on par with the intra-process experiments. Not only does the memory once again become a bottleneck but also in these experiments the network communication overhead incurred by serializing and transporting all the source streams through the network is so high that the system produces very little or no provenance data (even when increasing the duration of the experiments). As discussed in § 6, GeneaLog overcomes this problem since it only transmits the actual provenance data between nodes instead of the entire source stream.

Graph Traversal Overhead. To achieve further insight into the performance characteristics of our technique, we evaluated the cost of traversing the contribution graph (Listing 1) for every sink tuple produced. Figure 14 shows the results for both the intra-process and the inter-process case. For the former, in the majority of the queries, the traversal requires on average less than 0.4 ms. Even in Q3 which has the largest provenance graph with hundreds of tuples, the average overhead is approximately 1.6 ms, which is negligible considering the low frequency of alerts in streaming monitoring applications. For the latter, the average traversal time in all queries

in SPE instances 1 and 2 is less than 0.3 ms. This lower overhead is expected since the contribution graph in these experiments is split into multiple SPE instances, reducing the amount of work on each processing node. The slightly higher overhead in the first SPE instances is also expected since the contribution graphs are larger in these nodes which are closer to the sources.

Summary of evaluation results. As we showed in the different experiments, the overhead introduced by the GeneaLog framework in real-world use cases is minimal. In the majority of cases, the performance metrics of the queries were reduced by less than 15% whereas the memory usage remained less than 25MB. Even in queries which were more demanding for provenance, it continued to incur little overhead while running on resource-constrained devices, with the main performance bottleneck being the CPU. On the contrary, previously known provenance approaches proposed in the literature led to degradation of the QoS metrics of the query and proved inadequate for our use cases. This result indicates that GeneaLog can indeed be used in edge and fog environments with inexpensive, low-energy devices.

8 RELATED WORK

Although many different types of provenance exist [17], the most related to our work are *data provenance* and *workflow provenance*. In the field of databases[37], data provenance tracks individual data items to find from *where* the data of a result comes from (which attributes), *how* such result is produced (which operations) and *why* (i.e., the lineage of the result). It requires a high degree of instrumentation and strict semantics, which are usually defined in a database system [9] [21]. Alternatively, *why-not* provenance tries to explain why some expected results were missing from the output [7]. Workflow provenance is more general and refers to tracking the scientific, business and data analytics workflows [15] [13]. Provenance is especially challenging in the field of big data analytics [12] since many traditional techniques require access to the whole dataset and are thus inapplicable. Several provenance toolkits have been proposed for popular analytics frameworks such as Lipstick and Inspector Gadget [30] [4] for Pig [31] and Titian [24] for Apache Spark [38].

When focusing on streaming applications, early work on provenance has been discussed in [34], proposing a low-latency technique for generating coarse-grained provenance information about the dependencies between different streams instead of individual tuples. In [36], Wang et al. deem annotation based techniques inadequate and propose a model-based provenance technique for provenance in medical stream processing systems. Apart from the need to define explicit provenance rules on each operator their implementation requires all intermediate streams and thus is not well-suited for modern stream processing systems. A different approach, aimed at minimizing the storage requirements of provenance is followed in [22] where the processing time along with other run-time characteristics are utilized to generate the provenance information. However, this technique is not applicable to all the standard operators presented in § 2. The use of fine-grained data provenance in debugging stream processing applications is described in [14]. Provenance, in this case, is constructed using automatically generated inter-operator data-flows and user-defined intra-operator relations. Due to the performance impact and huge data volume of provenance-related data, the authors suggest using time-based and topology-based *execution slices*. Ariadne [16] is, to the best of our knowledge, the closest approach to GeneaLog in the literature. While also relying on operator instrumentation, it nonetheless requires the storing of all input data and uses variable-length annotations, potentially leading to degradation figures similar to the ones presented in our evaluation section.

Related in spirit to GeneaLog's distributed provenance approach, data streaming fault tolerance techniques are designed to maintain information about the data sent from a node A to a node B by forwarding copies of A's output tuples to *replicas* [6] or by buffering output tuples at A (the ones sent since the last backup of B for passive-standby [2] or all the ones contributing to B's state in upstream-backup [23]). To the best of our knowledge, existing solutions are not designed to purge the additional information A maintains per tuple as soon as such tuple can no longer contribute to a sink tuple but rather when such tuple is received by all replicas, safely persisted in a backup or acknowledged as processed by B. That is, existing solutions do not aim at minimizing the additional information they maintain based on whether the latter contributes to the application's results. As we mention in § 9, it would be valuable to study the joint benefits of techniques for reliable stream processing and provenance.

9 CONCLUSIONS AND FUTURE WORK

We presented GeneaLog, a method for streaming applications' fine-grained data provenance, and its algorithmic implementations for intra-process and inter-process deployments. GeneaLog advances the state-of-the-art by defining a technique in which performance overheads are minimized. This is crucial for streaming applications running edge analysis in modern cyber-physical systems. Under the hood, this is achieved by leveraging a small, fixed-size set of meta-attributes for each tuple processed by a streaming application (in contrast to existing solutions that rely on an arbitrary number of meta-attributes) and by using processes' memory reclamation techniques to discard tuples as soon as they do not contribute to other tuples in the streaming application.

We provide a fully implemented prototype and show it incurs small throughput and latency overheads and, in the worst case, a memory footprint of some tens of megabytes, in contrast with state-of-the-art techniques' overheads that are at least one order of magnitude greater and rapidly exhaust the memory of the devices running the analysis.

We believe that the results we present can be further investigated (i) to define operator-specific optimizations for window-based analysis and reduce overheads when only some (rather than all) of the tuples of a window contribute to a sink tuple, (ii) to adapt GeneaLog's technique to SPEs with ad-hoc memory management and (iii) to leverage GeneaLog in fault tolerance approaches that rely on upstream peers' buffering and minimize the number of tuples the latter maintain (in order to replay them in case of failure).

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions. The work was supported by the Swedish Foundation for Strategic Research, proj. "Future factories in the cloud (FiC)" grant nr. GMT14-0032, by the Chalmers Energy AoA framework proj. INDEED and STAMINA and by the Swedish Research Council (Vetenskapsrådet) proj. "HARE: Self-deploying and Adaptive Data Streaming Analytics in Fog Architectures" grant nr. 2016-03800.

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The Design of the Borealis Stream Processing Engine.. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Vol. 5. Asilomar, CA, USA, 277–289.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[3] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. 2009. Microsoft CEP Server and Online Behavioral Targeting. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1558–1561. https://doi.org/10.14778/1687553.1687590

[4] Yael Amsterdamer, Susan B Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. 2011. Putting lipstick on pig: Enabling database-style workflow provenance. *Proceedings of the VLDB Endowment* 5, 4 (2011), 346–357.

[5] 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, Toronto, Canada, 480–491. http://dl.acm.org/citation.cfm?id=1316689.1316732

[6] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. 2005. Fault-tolerance in the Borealis Distributed Stream Processing System. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1066157.1066160

[7] Nicole Bidoit, Melanie Herschel, and Aikaterini Tzompanaki. 2015. Efficient Computation of Polynomial Explanations of Why-Not Questions. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15)*. ACM, New York, NY, USA, 713–722. https://doi.org/10.1145/2806416.2806426

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[9] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2007. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2007), 379–474. https://doi.org/10.1561/1900000006

[10] Riccardo Coppola and Maurizio Morisio. 2016. Connected car: technologies, issues, future trends. *ACM Computing Surveys (CSUR)* 49, 3 (2016), 46.

[11] Stefania Costache, Vincenzo Gulisano, and Marina Papatriantafilou. 2016. Understanding the data-processing challenges in Intelligent Vehicular Systems. In *Intelligent Vehicles Symposium (IV), 2016 IEEE*. IEEE, Gothenburg, Sweden, 611–618. https://doi.org/10.1109/IVS.2016.7535450

[12] Alfredo Cuzzocrea. 2015. Provenance research issues and challenges in the big data era. *Proceedings - International Computer Software and Applications Conference* 3 (2015), 684–686. https://doi.org/10.1109/COMPSAC.2015.345

[13] 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)*. ACM, New York, NY, USA, 1345–1350. https://doi.org/10.1145/1376616.1376772

[14] 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*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–35.

[15] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. 2008. Provenance for computational tasks: A survey. *Computing in Science and Engineering* 10, 3 (2008), 11–21. https://doi.org/10.1109/MCSE.2008.79

[16] Boris Glavic, Kyumars Sheykh Esmaili, Peter M Fischer, and Nesime Tatbul. 2014. Efficient stream provenance via operator instrumentation. *ACM Transactions on Internet Technology (TOIT)* 14, 1 (2014), 7.

[17] Paul Groth and Luc Moreau. 2013. PROV-Overview. An Overview of the PROV Family of Documents. (April 2013). https://eprints.soton.ac.uk/356854/

[18] Vincenzo Gulisano. 2012. *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine.* Ph.D. Dissertation. Universidad Politécnica de Madrid.

[19] Vincenzo Gulisano, Yiannis Nikolakopoulos, Daniel Cederman, Marina Papatriantafilou, and Philippas Tsigas. 2017. Efficient Data Streaming Multiway Aggregation Through Concurrent Algorithmic Designs and New Abstract Data Types. *ACM Trans. Parallel Comput.* 4, 2 (Oct. 2017), 11:1–11:28.

[20] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2016. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Transactions on Big Data* (2016), 1–1. https://doi.org/10.1109/TBDATA.2016.2624274

[21] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *VLDB Journal* 26, 6 (2017), 881–906. https://doi.org/10.1007/s00778-017-0486-1

[22] Mohammad Rezwanul Huq, Andreas Wombacher, and Peter M.G. Apers. 2011. *Adaptive Inference of Fine-grained Data Provenance to Achieve High Accuracy at Lower Storage Costs.* IEEE Computer Society, USA, 202–209. https://doi.org/10.1109/eScience.2011.36 eemcs-eprint-21400.

[23] J. . Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. 2005. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, Tokyo, Japan, 779–790. https://doi.org/10.1109/ICDE.2005.72

[24] 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. *Proceedings of the VLDB Endowment* 9, 3 (2015), 216–227.

[25] Yuanzhen Ji, Hongjin Zhou, Zbigniew Jerzak, Anisoara Nica, Gregor Hackenbroich, and Christof Fetzer. 2015. Quality-Driven Continuous Query Execution over Out-of-Order Data Streams. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 889–894. https://doi.org/10.1145/2723372.2735371

[26] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. 2016. THEMIS: Fairness in Federated Stream Processing Under Overload. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 541–553. https://doi.org/10.1145/2882903.2882943

[27] liebre 2017. Liebre SPE. https://github.com/vincenzo-gulisano/Liebre.

[28] M. M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

[29] Odroid-XU4 2016. Odroid-XU4. http://www.hardkernel.com.

[30] Christopher Olston and Benjamin Reed. 2011. Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 1221–1224. https://doi.org/10.1145/1989323.1989459

[31] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 1099–1110. https://doi.org/10.1145/1376616.1376726

[32] storm 2017. Apache Storm. http://storm.apache.org/.

[33] Håkan Sundell and Philippas Tsigas. 2005. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. *J. Parallel and Distrib. Comput.* 65, 5 (2005), 609–627.

[34] Nithya N. Vijayakumar and Beth Plale. 2006. Towards Low Overhead Provenance Tracking in Near Real-Time Stream Filtering. In *Provenance and Annotation of Data*, Luc Moreau and Ian Foster (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 46–54.

[35] Ivan Walulya, Dimitris Palyvos-Giannas, Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. 2018. Viper: A module for communication-layer determinism and scaling in low-latency stream processing. *Future Generation Computer Systems* 88 (2018), 297–308.

[36] 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)*. ACM, New York, NY, USA, 95–100. https://doi.org/10.1145/1248054.1248082

[37] Y. Richard Wang and Stuart E. Madnick. 1990. A Polygen Model for Heterogeneous Database Systems: The Source Tagging Perspective. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB '90)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 519–538. http://dl.acm.org/citation.cfm?id=645916.758355

[38] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.