Parallel Computing 89 (2019) 102552

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

# GeneaLog: Fine-grained data streaming provenance in cyber-physical systems

Dimitris Palyvos-Giannas*, Vincenzo Gulisano, Marina Papatriantafilou

*Department of Computer Science and Engineering Chalmers University of Technology 412 96 Göteborg, Sweden*

ARTICLE INFO

*Article history:*
Received 2 November 2018
Revised 20 June 2019
Accepted 6 September 2019
Available online 14 September 2019

*Keywords:*
Cyber-physical systems
Online analytical processing engines
Fine-grained data provenance
Stream processing

ABSTRACT

Streaming applications continuously process data to deliver streams of up-to-date results. Their growing adoption for data analysis in many distributed systems is motivated by their performance (in terms of processing throughput and latency) and their support for easy-to-program distributed and parallel analysis. When streaming applications are designed to detect unusual or critical events (e.g., security- or safety-related), it can be beneficial to maintain the associated source data for further analysis. This can be achieved by fine-grained data provenance, which links each detected event back to the source data that contributed to it, allowing to distinguish and isolate the source data that generated such unusual or critical events.

Fine-grained data provenance can be especially useful in cyber-physical systems, such as vehicular networks and smart grids. By enabling the extraction of valuable information from raw sensor data, it could, for instance, reduce data transmission and storage requirements. Since cyber-physical systems can have heterogeneous multi-core architectures, ranging from inexpensive single-board computers to high-end servers, there is a demand for efficient provenance techniques that can take advantage of such parallel architectures with minimal overhead. 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 (instrumented) data streaming operators. This is particularly useful to distribute the provenance overheads to operators that can be run in parallel, thus leveraging multi-core architectures. We evaluate two implementations of GeneaLog, one based on Apache Flink, a widely-adopted state-of-the-art Stream Processing Engine, and one based on Liebre, an edge-tailored lightweight Stream Processing Engine. We test them both on vehicular and smart grid applications with single-board embedded devices and a high-end server, also studying how GeneaLog affects their scalability and confirming that it efficiently captures fine-grained provenance data with minimal overhead.



## 1. Introduction

The data streaming processing paradigm is commonly leveraged in applications that continuously process sensor data and deliver streams of up-to-date results. A critical goal of stream processing is to *distill* information into *events*, reducing the amount of data to be maintained. For example, in a streaming application that performs Complex Event Processing (CEP) [1], the detection of a user-defined pattern (e.g., “Smoke and High Temperature”) could lead to an output event (e.g., “Fire”). When the events produced by a streaming application refer to unusual or critical situations, it can then be desirable to *keep the source data* to understand the cause of the problems, replay the query or develop learning structures for future situations [2,3]. This is enabled by *fine-grained data provenance* (or simply provenance, in the remainder), which allows linking back each output (e.g., an alert in the presence of an accident [4]) with the source data that leads to it (e.g., the position reports of the cars involved).

In state-of-the-art solutions, provenance is achieved through operator instrumentation that enriches the tuples with provenance meta-data annotations [5]. 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

* Corresponding author.
*E-mail addresses:* palyvos@chalmers.se (D. Palyvos-Giannas), vinmas@chalmers.se (V. Gulisano), ptrianta@chalmers.se (M. Papatriantafilou).

https://doi.org/10.1016/j.parco.2019.102552
0167-8191/

for such an approach (e.g., provenance compression), the disadvantages of variable-length annotation-based techniques can result in prohibitive storage overheads for applications maintaining large states [6].

*Challenges.* Cyber-physical systems (CPSs) such as vehicular networks and smart grids are characterized by the large volumes of data sensed in them (e.g., dozens of gigabytes of data sensed every day by a modern vehicle [7]) and by the multi-core heterogeneous devices deployed in them. These devices can range from embedded computers with limited computational power, memory, and bandwidth (e.g., a vehicle's onboard computer) to high-end servers found in utilities' data centers. Provenance is an intrinsically heavy operation that bounds the performance of a given application to the efficiency with which provenance information is maintained for that application. Thus, our goal is to minimize the provenance overhead, both for time-performance aspects (e.g., throughput and latency) and memory requirements (e.g., temporal storage), making provenance affordable for the whole spectrum of devices found in modern CPSs, from embedded devices to data center servers.

*Contribution.* We propose GeneaLog, a new technique and framework for 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.
- For parallel and distributed deployments, it further allows for a modular and efficient algorithmic implementation by leveraging the instrumented operators and, optionally, by enriching queries with additional standard 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 operators.

We show the correctness of GeneaLog and evaluate it 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 implement prototypes of GeneaLog for the Liebre SPE, a lightweight SPE tailored for edge-computing [8], as well as for Apache Flink [9], a widely adopted modern SPE. We evaluate these prototypes in the different types of devices (and thus computational power levels) that can be found in modern CPSs. We present performance results observed in embedded devices as well as in high-end servers, studying the provenance overheads, comparing GeneaLog with state-of-the-art streaming provenance techniques and also studying GeneaLog's overheads on the scalability of streaming queries. As we show in our evaluation, GeneaLog overcomes state-of-the-art techniques making provenance affordable for streaming application in CPSs.

The paper is organized as follows. We introduce preliminary concepts in Section 2. We provide a formal problem definition in Section 3 and present GeneaLog's approach in Section 4–8, evaluating it in Section 9. We discuss related work in Section 10 and conclude in Section 11.

## 2. Preliminaries

A streaming continuous query consists of streams and operators. A stream is an unbounded sequence of tuples sharing a schema of 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 tuple creation time. 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 in this section) and, eventually, delivered as *sink tuples* to *Sinks*.

Deterministic execution is supported for a query when (i) the tuples of each source stream are fed to the operators in timestamp order (either because Sources deliver timestamp-sorted streams as in [10–12] or by leveraging sorting techniques such as [13]) and (ii) each operator produces timestamp-sorted output streams (merging its input tuples in timestamp order if they are delivered by multiple input streams, as discussed in [11,14–16]). Determinism implies that each processing step depends on the time attribute (*ts*) carried by the tuples themselves and is affected neither by the tuple transmission latency from one operator to another nor by the interleaving of tuples to an operator with multiple input streams. For the monitoring applications motivating our work (Section 1), determinism is crucial to identify the source data contributing to each output event unambiguously. For this reason, we assume that the queries for which provenance is provided run deterministically. We refer to [11,14,15] for more discussion on determinism.

Queries are deployed at one or multiple *nodes* and can be split at one or multiple *SPE instances* within each node. Existing SPEs use different naming conventions for such instances (e.g., *Worker* for Apache Storm [17] and *Task Manager* for Apache Flink [9]). Furthermore, SPEs such as Apache Flink define an additional intra-process unit named *task* [9]. Each task (we use this naming convention in the remainder) exists within a single *process*. Threads within the same task use shared memory to exchange tuples with each other while they use queues to communicate with threads from different tasks. The memory allocated to the objects of each task is freed when such objects are no longer accessible by the task's threads. This removal of (directly or indirectly) inaccessible objects is done either by garbage collection or other memory reclamation techniques. Without loss of generality, notice that based on our naming convention, one task is equivalent to a process for SPEs that do not define intra-process tasks.

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 [5,9,11,17] 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 one out stream. Since we assume operators enforce determinism, the Union merges timestamp-sorted input streams into a timestamp-sorted out stream, as discussed in [11,15].

Differently from stateless operators, stateful operators output tuples that depend on multiple input tuples. The standard *stateful operators* provided by SPEs such as [5,9,11,17] 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 a left ($L$) and a right ($R$) input stream, and produces a tuple combining (and possibly altering) the attributes of tuples $t_L \in L$ and $t_R \in R$ for each pair $< t_L, t_R >$ satisfying a given predicate and not being far apart more than a given window size WS (i.e., $|t_L.ts - t_R.ts| \leq \text{WS}$).

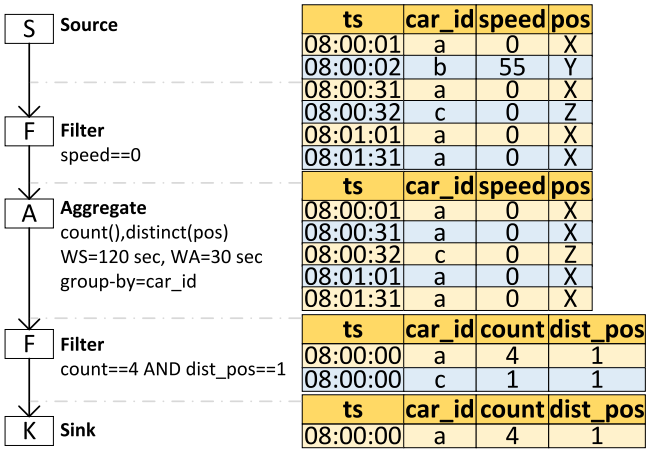



**Fig. 1.** Sample continuous query.

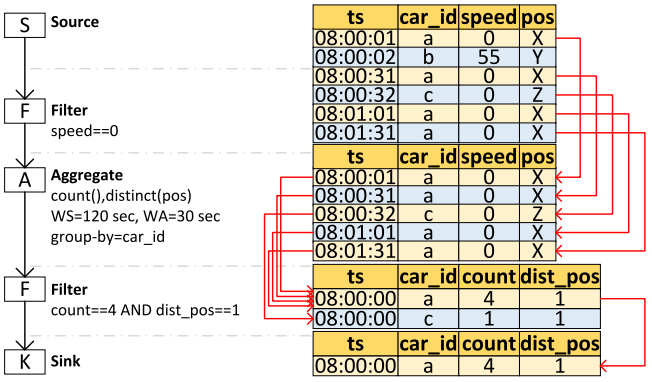



**Fig. 2.** Sample query with arrows for the sink tuple's contribution graph.

It should be noted that, once deployed at one task, each operator is not necessarily mapped to a dedicated thread. 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 Carbone et al. [9], 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 by combining a Multiplex and several Filter operators. We clarify this to highlight that, by discussing the standard (rather than ad-hoc) operators of an SPE, our contribution holds when their semantics 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.
*Send*. and *Receive* operators, which can be used to transmit and receive tuples between two distinct tasks (potentially deployed at distinct processes and/or nodes).

Fig. 1 presents a sample query that detects broken-down cars on highways (based on the Linear Road benchmark [4], further discussed in Section 9). The source tuples are position reports, emitted by each car every 30 s, 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 s, 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 provenance.

**Definition 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 a tuple from 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 $t_i$.

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 Fig. 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 be easily verified by retracing the process, as shown in Fig. 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.

In the remainder, when discussing deployments to multiple tasks (and possibly processes and nodes) we use the terms *explicit inter-task provenance* and *implicit inter-task provenance* to refer to solutions which maintain provenance by enriching the query with additional operators or by relying exclusively on its user-defined operators, as we further discuss in Section 6–8. For ease of exposition, we frame our discussion to the case of a single query deployed to one or multiple tasks. Multiple queries processing the same input data can be considered as a single query fed by the same set of Sources and, furthermore, it is straightforward to extend the discussion to scenarios in which more queries that process different input data are defined.

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

## 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 single-task deployments in Section 5 and multi-task deployments in Section 6–8.

The definition of the *contributes* relation (Definition 1), implies a *contribution graph* that connects each source tuple to the sink tuple(s) it contributes to. Fig. 2 shows the contribution graph for the example of Fig. 1. An approach for maintaining such a contribution graph, proposed by Glavic et al. [5], 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 Section 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 (Section 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 noted that no values are defined for operators that forward (instead of creating) tuples (e.g., the Filter operator), as we further elaborate in Section 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, contributing to it, and (ii) to traverse the contribution-graph of each sink tuple, back to the source tuples contributing to it. We detail in the following how these meta-attributes are set for the standard operators (Section 4.1) and how the contribution graph is traversed (Section 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 Section 2, to maintain the contribution graphs connecting source tuples to sink tuples. Similarly to [5], 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 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:

- *Source* operators create 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 $N$.
- *Map* operators create one or more output tuples for each input tuple they process. With GeneaLog's instrumented Map, each output tuple $t_O$ points to the input tuple $t_I$ contributing to it with meta-attribute $U_1$, hence $t_O.U_1 = t_I$, as also shown in Fig. 3. Additionally, attribute $T$ is set to *MAP*.
- *Multiplex* operators create a copy of each input tuple they process for each one of their 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 Fig. 3. Additionally, attribute $T$ is set to *MULTIPLEX*.
- *Join* operators, as discussed in Section 2, produce output tuples $t_O$ that have exactly two contributing input tuples $t_R$ and $t_S$. Without loss of generality, assuming $T_R.ts \geq T_S.ts$, GeneaLog's instrumented Join operator sets $t_O.T$ to *JOIN*, $t_O.U_1 = T_R$ and $t_O.U_2 = T_S$, as shown in Fig. 3.
- *Aggregate* operators produce output tuples to which multiple input tuples contribute to. Based on Definition 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 Fig. 3. This instrumentation works both for overlapping and non-overlapping windows. While we do not assume bounded windows, provenance for unbounded windows will also be unbounded (depend on an unbounded number of source tuples), eventually exceeding the system's memory capacity (because some source tuples will need to be kept indefinitely).
- *Filter and Union* operators do not produce new tuples, but rather forward existing ones across the query's streams. Hence, no instrumentation is defined for them.
- *Send and Receive* operators are used to send tuples across distinct tasks (possibly running at different processes or 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 create new memory objects when serializing / deserializing tuples across tasks, optionally transmitting these tuples 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 task, tuples produced at other tasks.

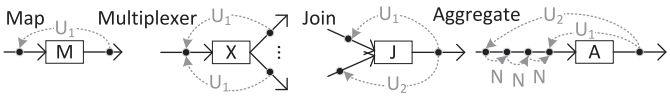



**Fig. 3.** Representation of meta-attributes (pointers) $U_1$, $U_2$ and $N$ set by the instrumented Map, Aggregate and Join operators. For simplicity, we show only the meta-attributes set by each operator, thus ignoring dangling pointers.

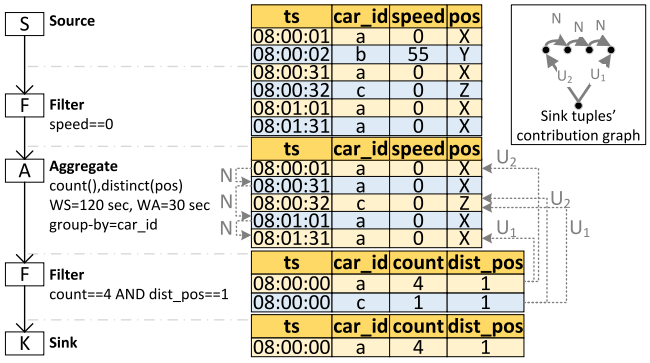



**Fig. 4.** Sample query and execution from Fig. 1 showing meta-attributes $U_1$, $U_2$ and $N$ as set by GeneaLog's instrumented operators.

Fig. 4 shows how the meta-attributes $U_1$, $U_2$ and $N$ are set for the sample query and execution presented in Fig. 1 by GeneaLog's

```
Set findProvenance(root):
  Set provenance
  Set visited
  Queue q
  q.addLast(root)
  while (!q.isEmpty()):
   Tuple t = q.removeFirst()
   switch (t.type):
    case SOURCE or REMOTE:
     result.add(t)
    case MAP or MULTIPLEX:
     enqueueIfNotVisited(t.U1, q, visited)
    case JOIN:
     enqueueIfNotVisited(t.U1, q, visited)
     enqueueIfNotVisited(t.U2, q, visited)
     break;
    case AGGREGATE:
     enqueueIfNotVisited(t.U2, q, visited)
     Tuple temp = t.U2.N;
     while (temp != null && temp != t.U1)
      enqueueIfNotVisited(temp, q, visited)
      temp = temp.N;
     enqueueIfNotVisited(t.U1, q, visited)
  return result;

void enqueueIfNotVisited(tuple, queue, visited):
  if (!visited.contains(tuple)):
   visited.add(t)
   queue.addLast(t)
```

**Listing 1.** Contribution graph traversal.

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 Section 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 tasks (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 (implementing a breadth-first search of the contribution graph of a tuple).

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

**Definition 2.** Tuple $t'$ is an *originating tuple* of $t$ if $t'$ is returned by the provenance method in Listing 1 as contributing to $t$.

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

## 5. Intra-task provenance

In this section, we examine GeneaLog's behavior when all the operators of a query are deployed in a single task. More specifically, we show how GeneaLog addresses challenges C2 and C3 presented in Section 3 (i.e., avoiding maintaining all source tuples and allowing for provenance analysis to be implemented with standard operators), given the instrumented operators and the contribution graph traversal approach discussed in Section 4.

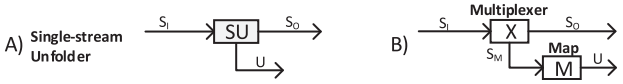



**Fig. 5.** SU operator **(A)** and its implementation using standard operators **(B)**.

Regarding *challenge C2*, GeneaLog can access the necessary information without maintaining or storing all source tuples. It does this by utilizing memory pointers and assuming that the memory used by objects like tuples is freed when such objects are no longer accessible (directly or indirectly) by the task's threads (Section 2). The memory of these tuples will be referenced by GeneaLog (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.

Thus, GeneaLog's meta-attributes $U_1$, $U_2$ and $N$ have a dual role. First, they represent the edges of the contribution graph, connecting each source tuple to the sink tuple(s) it contributes to (as described in Section 4). Second, by being actual memory references, these meta-attributes prevent source tuples from being reclaimed as long as they are (potentially) part of a sink tuple.

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

**Definition 3.** Stream $U$ is the *unfolded stream* obtained from stream $S$ if each tuple $t \in S$ is replaced by its originating tuples (see Definition 2) combined with $t$'s attributes.

In the following, we consider the existence of a *single-stream unfolder* operator (whose semantics are described in Definition 4) and show that provenance can be achieved Theorem 1, by enriching the query with such an operator. Next, in Section 5.1, we show that the semantics of such a streaming operator can indeed be achieved utilizing the instrumented operators described in Section 3, thus addressing challenge C3. Moreover, we prepare the ground for the explanation of GeneaLog's inter-task provenance, later provided in Section 6–8.

**Definition 4.** The SU (*single-stream unfolder*) operator (Fig. 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 1.** *A query in which an additional SU operator is added before each Sink (with $S_O$ feeding the Sink) provides provenance through U.*

**Proof.** Since all query's operators are deployed within the same task, 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 provenance. □

### 5.1. SU implementation using standard operators

Fig. 5B shows how the semantics of the SU operator can be defined utilizing the instrumented operators provided by GeneaLog. As discussed in Section 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 implementations can assign these operators to the same thread using chaining (e.g., as in [9]) or by implementing their semantics in a single user-defined operator.

As shown in Fig. 5B, 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 those of each originating source tuple of $t_{SINK}$.

## 6. From intra-task to inter-task provenance

In multi-task deployments, it is not enough to rely solely on the memory management of a task. Maintaining references from sink to source tuples through the use of pointers is no longer sufficient since pointers assigned to tuples that are later forwarded across tasks and dereferenced would be lost.

To satisfy challenge C3 (Section 3), we can enrich a query with additional (instrumented) standard operators[2] that are responsible for sharing the SOURCE tuples of a task with the latter's downstream tasks. We refer to this as *explicit* inter-task provenance, discussed in Section 7. Alternatively, we can distribute the actions related to sharing SOURCE tuples among tasks to the query's operators (i.e., without adding extra operators). We refer to this as *implicit* inter-task provenance, discussed in Section 8.

As we discuss in the following, while both approaches can leverage existing distribution and parallelization techniques for standard streaming operators, the former can result in a large number of additional operators (depending on the query and how the latter is deployed). On the other hand, the latter does not result in additional operators for the query but incurs an additional overhead for some of them. Our discussion is complemented by the empirical evaluation presented in Section 9.

## 7. Explicit inter-task provenance

Here we extend the provenance technique discussed in Section 5 to setups where the operators of a query are deployed to multiple tasks with explicit inter-task provenance. We introduce some notation and terms and then present our method and argue about correctness, similarly to Section 5. First, we show that provenance can be achieved by considering the existence of an additional (*multi-stream unfolder*) streaming operator, whose semantics are described in Definition 8. Next, we show that the semantics of this streaming operator can be implemented by composing standard operators described in Section 3. In this way, besides proving correctness, we also explain how GeneaLog meets challenge C3 (Section 3) for inter-task explicit data provenance.

We refer to the $n$ tasks in which a query is deployed as $\mathbb{V} = V_1, \ldots, V_n$. We say that $V_i$ is a *source* task if the operators deployed inside it are fed by Sources also deployed inside it, but not from other tasks (i.e., if no Receive operators are deployed at $V_i$). We say $V_i$ is a *sink* task if Sinks are deployed in it, and $V_i$ does not contain Send operators. Finally, we say $V_i$ is an *intermediate* task if it is neither a source nor a sink task. The *ordering value* of $V_i$ is defined as the longest path in the graph of tasks from a source task to $V_i$. Let $\mathbb{V}_q$ denote the set of tasks having ordering value $q$. For inter-task explicit 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[3]. We use the terms *delivering stream*, *unfolded delivering stream* and *complete unfolded delivering stream* as:

[2] Aside from the SU operators preceding the Sink operators since, as discussed in Section 5, their output stream $U$ can be consumed by the user (e.g., writing tuples to disk or transmitting them through the network) as discussed in Section 9.

[3] For instance, composed by the unique id of the Source or operator producing the tuple and a sequential counter, as done by Glavic et al. [5].

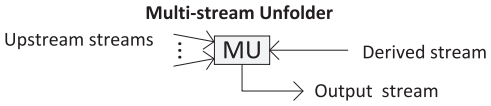



**Fig. 6.** Representation of the input and output streams defined by the MU operator (Definition 8).

**Definition 5.** Stream $S$ is a *delivering stream* if it feeds a Sink or is produced by a Send operator.

**Definition 6.** 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 (Definition 2) concatenated with $t$'s attributes. For each tuple $t'$ in $U$, we refer to the originating tuple's attributes ID and ts of tuple $t'$ as $t'.ts_O$ and $t'.ID_O$.

**Definition 7.** 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 Fig. 6.

**Definition 8.** The MU (*multi-stream unfolder*) operator defines multiple unfolded delivering streams (see Definition 7) 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.

Finally, assuming that the SU operator presented in Section 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 Fig. 5), we can state the following theorem, that builds on complementing queries with SU and MU operators.

**Theorem 2.** *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 $Q$'s operators and $Q_E$'s extra MU 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 $MU_i$,*
- *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 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 a task $V_j$ 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 $\text{MU}_i$ operator from a task $V_j$ 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 tasks in $\mathbb{V}_0$, which are all of type *SOURCE* and can be found in the upstream streams of $\text{MU}_i$. By induction, any stream $O_i$ produced by an $\text{MU}_i$ operator from a task $V_j$ 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 tasks in $\mathbb{V}_m | m < l$, which are all of type *SOURCE* and

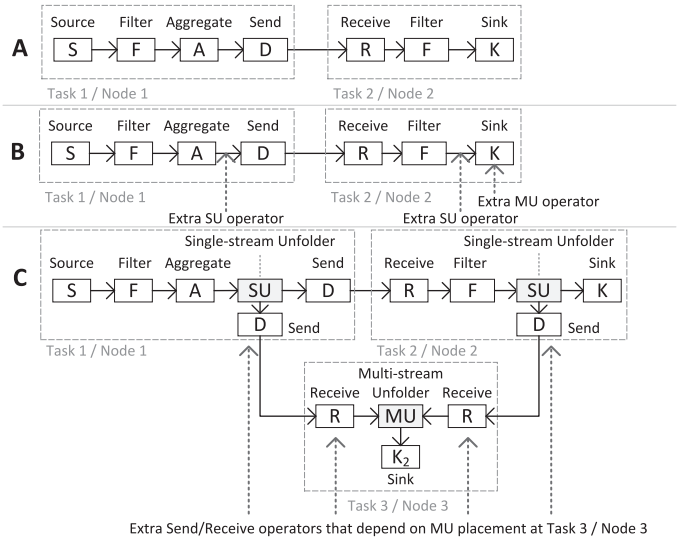



**Fig. 7.** Distributed deployment of the sample query (Fig. 1) showing how additional SU and MU operators are added for explicit provenance depending on the query's and the extra MU operator deployment decisions.

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 producing a sequence of tuples in which each sink tuple is combined with all the source tuples contributing to it, thus providing provenance. □

As stated in Theorem 2, the deployment of additional Send / Receive operators depends on how $Q$'s operators and how $Q_E$'s extra MU operators are deployed. The reason why $Q_E$'s extra SU operators do not lead to additional Send / Receive operators is because SU operators need to be placed in the same task as their subsequent Send or Sink operators, in order to traverse their tuples' contribution graphs. On the other hand, MU operators can be deployed at any of the tasks defined for $Q$ as well as at other dedicated tasks, as shown in the following step-by-step example, in Fig. 7-A. That figure presents how the query from Fig. 1, deployed at two tasks (each at a different node), is enriched for inter-task explicit provenance. As shown in Fig. 7-B, two additional SU operators are required (one for the Send and one for the Sink operators) and, since the Sink belongs to $\mathbb{V}_1$, one MU operator is also required to deliver its provenance stream. In the example, we assume the provenance is maintained at a dedicated third task (deployed at a separate third node) with the extra MU operator and an additional Sink deployed as shown in Fig. 7-C. Notice that two additional pairs of Send / Receive operators are deployed in this case.

For simplicity, the figures shown in the remainder for multi-task deployments 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.

### 7.1. MU implementation using standard operators

The key operator needed to implement the semantics of the MU operator (Definition 8) is a Join. Considering the task ordering values as defined in this section, the Join in MU at each ordering value level processes unfolded streams of the previous and the current tasks level, to extract useful information to be fed to downstream tasks, in order to generate the source-and-sink tuple pairs belonging to the contributes relation (Definition 1).

Fig. 8 shows how the MU operator can be constructed, utilizing the standard operators described in Section 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. 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 task 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.

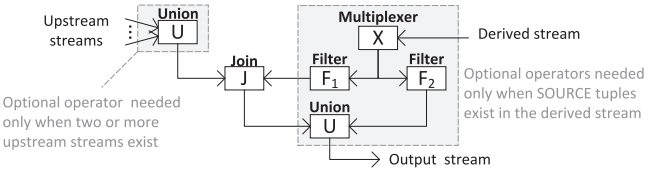



**Fig. 8.** Implementation of the MU operator's semantics using the standard operators defined in Section 2.

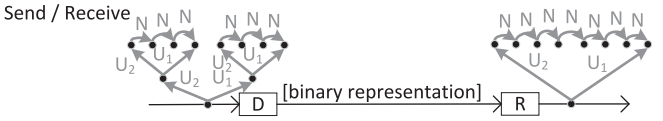



**Fig. 9.** Representation of meta-attributes (pointers) $U_1$, $U_2$ and $N$ set by the instrumented Send and Receive operators for implicit inter-task provenance (for a sample tuple and its contribution graph).

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.

## 8. Implicit inter-task provenance

As anticipated in Section 6, implicit inter-task provenance relies solely on the query's operators to perform the additional operations required to share SOURCE tuples among tasks (differently from the explicit inter-task provenance discussed in Section 7).

As mentioned in Section 7, extra SU and MU operators (and potentially extra Send and Receive operators too) are required for each Send operator defined in the query for which provenance is maintained. When the query's operators are distributed to many tasks (to the extreme case of dedicated per-operator tasks) this can result in a large number of additional operators for explicit inter-task provenance. At the same time, leveraging explicit inter-task provenance could also be difficult when the placement of such extra operators are based on decisions taken by an SPE that cannot be (easily) modified by the programmer.

For implicit inter-task provenance, the traversal of the contribution graphs by the SU operators before each Send operator can be integrated into the Send operator itself. Thus, each REMOTE tuple is *serialized* locally, at the Send operator's task, together with the SOURCE tuples contributing to it and later *deserialized* at the task where the corresponding Receive operator is deployed. As shown in Fig. 9, the modified Send and Receive operators can maintain the contribution graph of REMOTE tuples by proper serialization / deserialization and reduce it to its SOURCE tuples, using at the Re-

```
case SOURCE:
 result.add(t)
case AGGREGATE or REMOTE:
 enqueueIfNotVisited(t.U2, q, visited)
 Tuple temp = t.U2.N;
 while (temp != null && temp != t.U1)
  enqueueIfNotVisited(temp, q, visited)
  temp = temp.N;
 enqueueIfNotVisited(t.U1, q, visited)
```

**Listing 2.** Modified cases of the contribution graph traversal for implicit inter-task provenance.

ceive task meta-attributed $U_1$ and $U_2$ to point to the earliest and latest contributing SOURCE tuples, respectively, and the SOURCE tuples' $N$ meta-attributes when more than one SOURCE tuple is found in the graph. The tuples deserialized by the Receive operator resemble the ones produced by the Aggregate operator in terms of how they use the $U_1$, $U_2$ and $N$ meta-attributes. Based on this observation, the modified graph traversal algorithm can be obtained from Listing 1 by moving the REMOTE case to the AGGREGATE one, as shown in Listing 2.

With the updated Send and Receive operators, and the modified graph traversal in Listing 2 we can have the theorem below.

**Theorem 3.** *A query deployed to multiple tasks and leveraging implicit inter-task provenance in which an additional SU operator is added before each Sink (with $S_O$ feeding the Sink) provides provenance through U.*

**Proof.** Any Receive operator deployed at a task $V_j$ in $\mathbb{V}_1$ will append to its incoming REMOTE tuples all the SOURCE tuples contributing to the latter. Locally in task $V_j$, the contribution graph traversal will then find all SOURCE tuples from $\mathbb{V}_0$ and $V_j$. By induction, any Receive operator deployed at a task $V_j$ in $\mathbb{V}_l$ will append to its incoming REMOTE tuples all the SOURCE tuples contributing to the latter. Locally in task $V_j$, the contribution graph traversal will then find all SOURCE tuples from tasks in $\mathbb{V}_m | m < l$, up to all sink tasks and SU operators preceding their Sink operators. □

## 9. Evaluation

In this section, we evaluate GeneaLog's performance. We first introduce the hardware and software setup and then present the different use cases. The evaluation on embedded devices resembling those at the edge of CPSs takes into account both single- and multi-node deployments. For both, we evaluate *Liebre* [8] (for implicit inter-task communication) and Apache Flink (for explicit inter-task communication) and present the performance when no provenance is maintained and when GeneaLog's provenance capture is active. Furthermore, we compare GeneaLog's performance in Liebre with an implementation of Ariadne, the current state-of-the-art in eager streaming provenance [5] since both rely on explicit inter-task provenance. For all experiments, we provide a detailed analysis of the cost of traversing the contribution graph. To study GeneaLog's scalability in server-like architecture with a high number of cores, we also examine query performance for increasing parallelism degrees. In that case, we rely on Apache Flink since it transparently manages query parallelization.

*Hardware and software setup.* To evaluate provenance on embedded devices similar to those deployed in modern cyber-physical systems, we use 3 Odroid-XU4 [18] (or simply Odroid in the remainder), equipped with a Samsung Exynos5422 Cortex-A15 2 Ghz and Cortex-A7 Octa core CPUs and with 2 GB of memory. The Odroids are running Ubuntu 16.04.4 LTS and Java HotSpot Client VM 1.8.0_161-b12. The devices are connected to a 100 Mbps

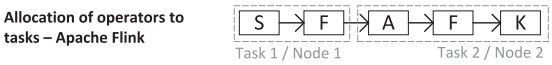



**Fig. 10.** Allocation of Query $Q_1$ operators to task / nodes for the Apache Flink SPE.

switch. The scalability evaluation is performed on a server with two 2.10 GHz Intel(R) Xeon(R) E5-2695 processors and 64 GB memory. Each processor has 18 cores with 32 KB of L1, 256 KB of L2 and 45 MB of L3 cache. The server runs Ubuntu 16.04 and Java HotSpot 1.8.0_171-b11.

To evaluate GeneaLog, we consider the *throughput* (the average number of tuples per second that a 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 stored on disk and the total size of the provenance is negligible compared to that of the source data (ranging from 0.003%–0.5% of the latter). 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 and results are averaged over at least five runs and presented with 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 [19], discontinued since 2008. As discussed in Section 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.

*Use cases.* We test GeneaLog with two queries from the vehicular network domain (using the Linear Road benchmark) and two queries implementing real use cases from a real-world Smart Grid infrastructure. These queries use all the standard operators presented in Section 2. The different queries are chosen to determine the overhead incurred by GeneaLog for different amounts of information (e.g., contribution graph size) needed to maintain provenance information. For all queries, we chose different allocations of operators to tasks and nodes to explore a broader set of deployments, as shown in Figs. 10–13. In the Liebre SPE, each task corresponds to a process and is deployed in a dedicated node (for multi-node deployments).

- **Q1.** Detecting broken-down cars (Linear Road benchmark). The first use case is the one presented in Section 2 (Fig. 1), based on the Linear Road benchmark [4], an established standard to study SPE performance. It simulates vehicular traffic on linear expressways, composed of predefined *segments*. As discussed in Section 2, position reports are forwarded every 30 s by the cars traveling in the highway and carry the attributes[4] $< ts, car\_id, speed, position >$. 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) – Fig. 11. This query extends Q1 to detect accidents. 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

[4] 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).

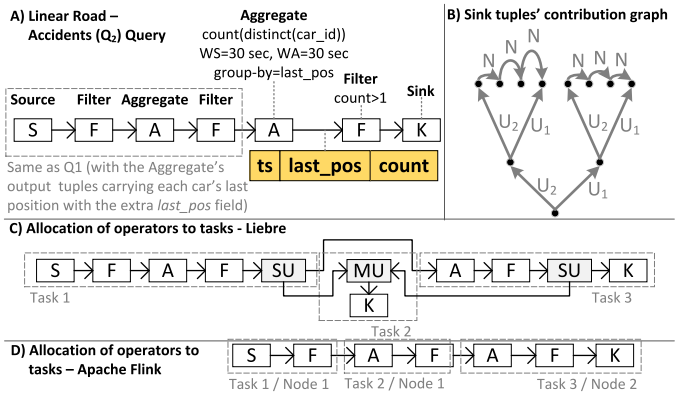



**Fig. 11.** **(A)** Query $Q_2$. **(B)** Sink tuples' contribution graph, with 8 input tuples. **(C,D)** Allocation of operators to task / nodes.

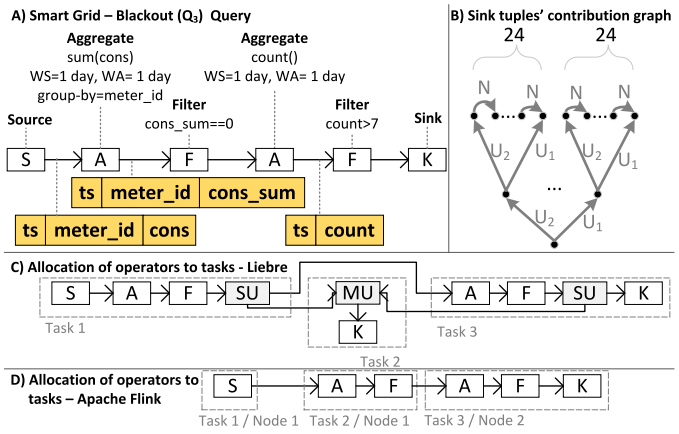



**Fig. 12.** **(A)** Query $Q_3$. **(B)** Sink tuples' contribution graph, with 192 input tuples on average. **(C,D)** Allocation of operators to task / nodes.

operators as Q1[5] plus an additional Aggregate and Filter operator. The former aggregates using the position as the group-by and a window of size and advance of 30 s. The resulting tuple carries the number of stopped vehicles observed for each position in the same time window. The Filter operator forwards only tuples carrying a counter value equal to or greater than 2. As for provenance, 8 source tuples contribute to each sink tuple.

**Q3.** Long-term blackout detection (Smart Grid) – Fig. 12. This query aims to detect blackouts in Smart Grid systems. Source tuples are measurements forwarded by smart meters every hour, with the schema $< ts, meter\_id, consumption >$. The 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 of one day. If there are more than seven meters which report zero consumption for a whole day, an alert is raised. In this case, 192 source tuples contribute to each sink tuple on average.

**Q4.** Anomaly detection (Smart Grid) – Fig. 13. This query aims at detecting faulty meters that show suspiciously high consumption in correspondence with the beginning of a new day (i.e., reporting such value at midnight). Such behavior indicates meters are compensating for missing consumption

[5] With the Aggregate operator producing tuples with an extra `last_pos` field carrying the last position reported by each car.

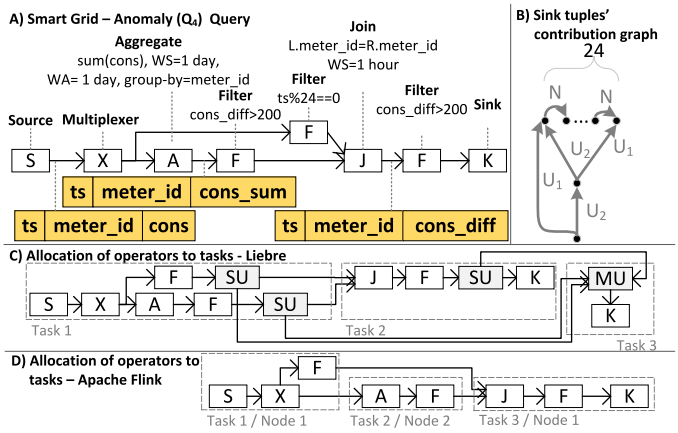
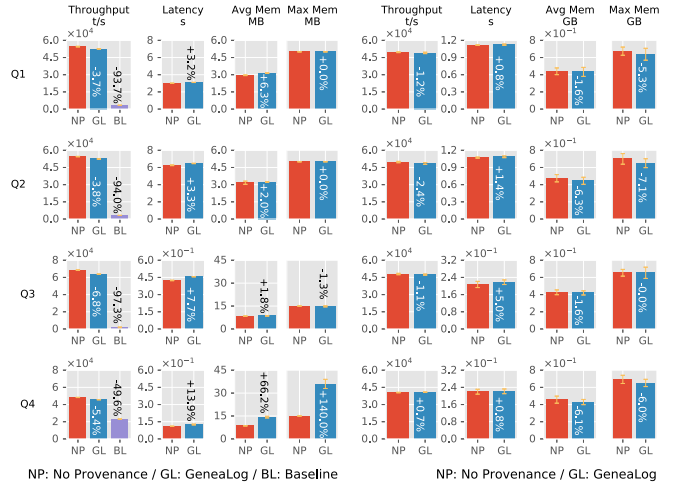



**Fig. 13.** **(A)** Query $Q_4$. **(B)** Sink tuples' contribution graph, with 24 input tuples. **(C,D)** Allocation of operators to task / nodes.




(a) (b)

**Fig. 14.** Performance of single-node/single-task deployments in Liebre (a) and single-node/multi-task deployments in Apache Flink (b).

about the previous day. The source tuples have the same schema as in Q3 and are forwarded every hour. The source stream is split into two identical streams. The first is sent to an Aggregate similar to the one in Q3, grouping by meter and calculating the daily consumption. The second 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. Another Filter produces an alert if the consumption difference is higher than a specified threshold. 24 source tuples contribute to each sink tuple.

### 9.1. Embedded devices, single-node deployment

We discuss in the following the results observed when deploying the use cases (with and without provenance enabled) on a single Odroid device, as presented in Fig. 14. Each row of the graph refers to one query, and each column is a performance metric. For explicit provenance, we also provide a comparison with the baseline. As discussed later in this section, the performance degradation for the baseline is so severe that we failed to record useful data for latency and memory consumption, thus forcing us to dis-

play only the throughput values. Note that, in this deployment, the query is executed as a single process of the SPE. In the case of the Liebre SPE, this translates to a one processing task and thus only intra-task provenance computation. On the contrary, Apache Flink splits the query into multiple sub-tasks resulting in both intra- and inter-task provenance computation even in this single-node deployment.

*Liebre (single-node/single-task deployment).* As seen in Fig. 14a, GeneaLog's intra-task provenance incurs minimal performance overhead. In both Linear Road queries (Q1, Q2), the throughput and latency overhead of GeneaLog's fine-grained provenance tracking is less than 4% while the memory consumption is almost identical. The blackout detection query (Q3) experiences slightly higher performance degradation, about 8%, when GeneaLog is enabled. The situation is similar in the anomaly detection query (Q4) where the throughput drops by 5.4% and the latency increases by 13.9% (+15 ms). For this query, 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 the smart grid 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. This can be explained by considering that Q4 has a Join which, in order to ensure determinism, cannot process its input immediately but needs to buffer tuples from each stream, maintaining them until it can match them with the all relevant ones from the other stream [15], delaying garbage collection and leading to an increase in the maximum amount of memory allocated by the JVM. The fact that the Join operator provided by Liebre does not include optimizations (e.g., KeyBy for EquiJoins) exacerbates this issue. Note that the actual memory consumed is generally so small that in Q2 the maximum memory usage drops slightly (this is likely because of the different behavior of Java's garbage collector).

For the baseline (BL), in all queries except Q4, the average throughput is an order of magnitude lower than GeneaLog, with the bottleneck being the high memory usage. Further, we observe that BL's throughput decreases as the experiment progresses, reaching values close to zero and indicating that the system is overloaded. Consequently, we do not report BL's data for latency since it was not possible to get an accurate representation based on the limited number of sink tuples produced. Moreover, the memory consumption of BL is always more than one order of magnitude higher (approximately) than both NP and BL. For this reason, and in order to clearly compare NP and BL, we do not show BL's memory consumption either.

*Apache Flink (single-node/multi-task deployment).* Fig. 14b demonstrates that GeneaLog with multiple tasks in a single node results in a very small drop in performance, usually 1% – 2%. Q3 has slightly higher decrease in average latency of 5.3% but also a higher variance between experiment repetitions. We could not observe any notable changes in the average and maximum memory consumption when GeneaLog is enabled. Furthermore, in some cases the memory consumption is lower with GeneaLog. Like in the case of Liebre, this is likely because the memory access patterns and the behavior of the garbage collector change when GeneaLog is active.

### 9.2. Embedded devices, multi-node deployment

We now discuss the results observed when deploying the use cases on multiple Odroid devices. The deployment of the two Linear Road queries (Q1 and Q2) is shown in Figs. 7,10 and 11, while Smart Grid (Q3 and Q4) deployments are shown in Figs. 12 and 13.

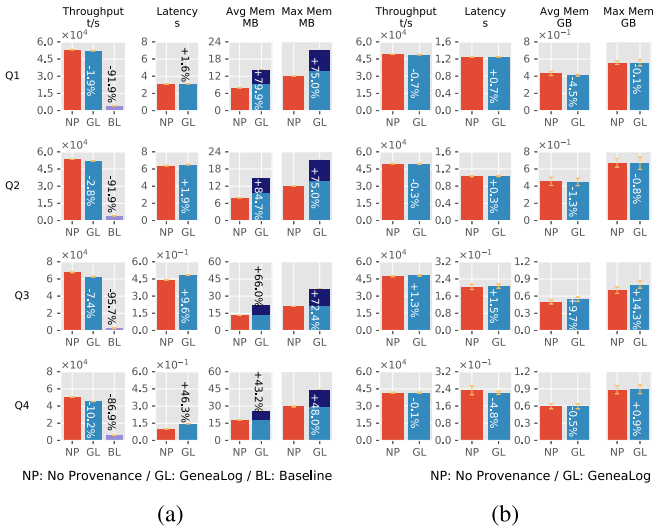



**Fig. 15.** Performance of multi-node/multi-task deployments for Liebre (a) and Apache Flink (b).

*Liebre (explicit provenance).* The results of the explicit provenance experiments can be seen in Fig. 15a. Two Odroids process the data and one more records the final provenance stream. The memory consumption is measured as the sum of the consumption of each process. While this consumption remains almost identical in the processing nodes, the total is always higher due to the additional node. The contribution of the additional node is the darker-colored part at the top of the bars and as seen in the figure, most of the increase in memory consumption is due to that effect. In Q1 and Q2, the impact of GeneaLog's provenance in throughput and latency is less than 3% while the memory consumption on the two processing nodes is almost identical. Similarly to the single-node deployment, GeneaLog's overhead is higher in Q3 and Q4, with throughput dropping by 7.4% and 10.2% and latency increasing by 9.6% (+42 ms) and 46.3% (+46 ms) respectively. The effect of the large window and the relatively high output event rate is even more prevalent in the multi-node deployment since a larger number of tuples need to be serialized and transmitted over the network to the third node, negatively impacting the performance metrics.

Here, the behavior of BL is on par with the single-node deployment. Not only does the memory once again become a bottleneck but also 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 experiments' duration). Because of this severe performance degradation of the baseline, we failed to record any accurate latency or memory consumption data for BL, and thus they are omitted from the figure. GeneaLog overcomes this problem by transmitting the actual provenance data between nodes, not the entire source stream (Section 7).

*Apache Flink (implicit provenance).* Fig. 15b presents the impact of provenance computation in multi-node deployments with Apache Flink. In correspondence with the single-node deployment, the overhead induced by GeneaLog is negligible, staying in the order of 1% – 2% in the majority of cases. The memory consumption also remains mostly unchanged, especially taking the high variance of its values between executions.

*Graph traversal overhead.* To better understand the performance characteristics of GeneaLog, we evaluate the cost of traversing the contribution graph for every sink tuple produced, using either ex-

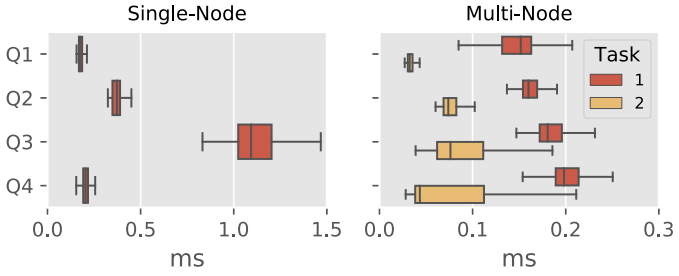



**Fig. 16.** Time required to traverse the contribution graph for each output tuple using explicit provenance. Note the different x-axes between the figures.

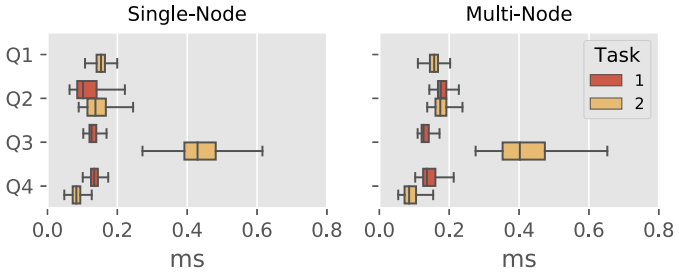



**Fig. 17.** Time required to traverse the contribution graph for each output tuple using implicit provenance. Note that Q1 is deployed in two tasks, only one of which traverses the contribution graph.

plicit (Listing 1) or implicit (Listing 2) provenance. Figs. 16 and 17 show the results for both single- and multi-node deployments of Liebre (explicit provenance) and Apache Flink (implicit provenance), respectively. Note that in Liebre, each process corresponds to a single task and as a result, only one task is executed in single-node deployments. In the vast majority of cases, the traversal requires less than 0.5 ms on average. Even in Q3, which has the largest provenance graph with hundreds of tuples, the average traversal time reaches a maximum of 1.6 ms, which is negligible considering the low frequency of alerts in streaming monitoring applications.

In multi-task deployments with explicit provenance, the second node has lower traversal times than the upstream one. This is because the contribution graph in these experiments is split into multiple SPE instances, reducing the amount of work on each processing node. The situation is reversed in the case of implicit provenance. For the latter, the downstream task has to perform additional work since it is responsible for combining all previous provenance data into a new provenance record. This additional work causes downstream tasks to have slightly higher graph traversal times in the case of implicit provenance.

### 9.3. High-end server, scalability study

Figs. 18–21 present the throughput and latency of Q1-Q4 when they are deployed in Apache Flink with varying degrees of parallelism. We measure the performance of each query with and without GeneaLog's implicit provenance. Each line shows the value of a performance metric over time for a given parallelism level while the shaded regions represent the 95% confidence interval. We increase the parallelism values until the performance of the original query (without provenance) starts degrading or when the query requires more than the available physical threads. As seen in Figs. 18 and 19, for the linear road queries, the performance of GeneaLog is nearly identical to the original query. The stopped vehicle detection query (Q1) shows signs of overload (increasing latency over time) as parallelism increases, which is to be expected since it only has one aggregate and thus it becomes mostly IO-bound for

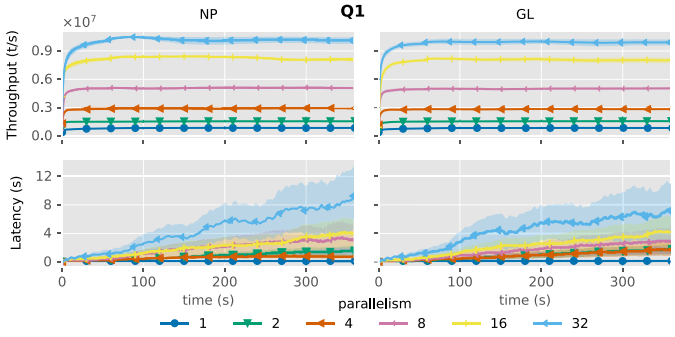



**Fig. 18.** Scalability study for the stopped vehicles query.

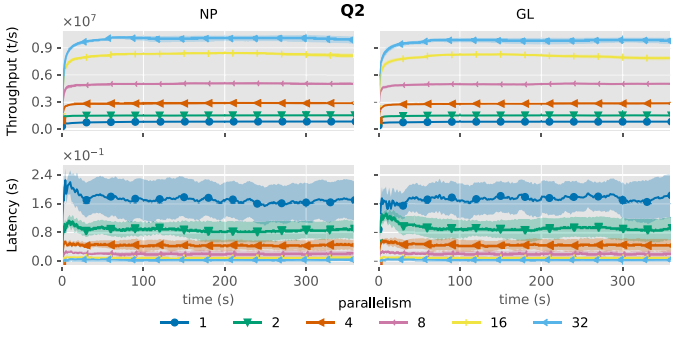



**Fig. 19.** Scalability study for the accident detection query.

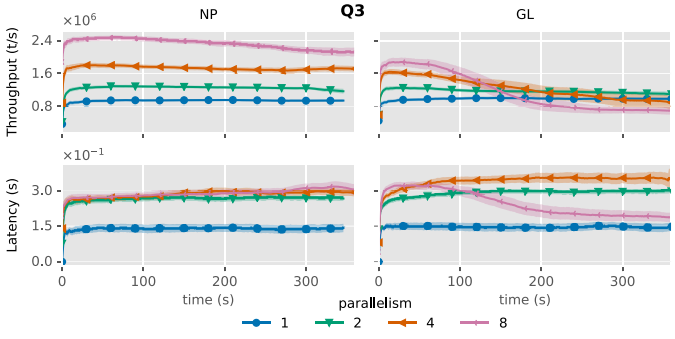



**Fig. 20.** Scalability study for the blackout detection query.

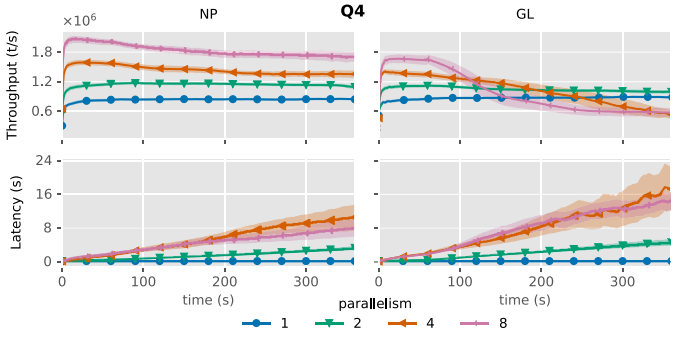



**Fig. 21.** Scalability study for the anomaly detection query.

higher parallelism degrees. Figs. 20 and 21 indicate that the smart grid queries (Q3 and Q4) do not benefit from higher parallelism. The anomaly detection query (Q4) starts having increasing latency from parallelism of 2 and degrades fast for further increases in parallelism. The situation is less pronounced in the blackout detection query (Q3), but it still exists, with throughput dropping

dramatically from parallelism 8 and garbage collection times (not shown in the graph) increasing even from parallelism 4. In these two cases, GeneaLog's performance drops faster than the query without provenance, but this is expected since the additional work of provenance computation pushes the already overloaded system over the edge earlier. Notice though that, as long as the system is not severely overloaded, the throughput and latency observed when provenance is captured are similar to those observed when no provenance is maintained.

*Summary of evaluation results.* We show GeneaLog's overhead in real-world use cases is minimal. In the majority of deployments, queries' performance was reduced by less than 10% whereas the memory remained largely unchanged. Even in more demanding queries for provenance, GeneaLog incurred little overhead even in resource-constrained embedded devices, with its main performance bottleneck being the CPU. In comparison, other provenance approaches led to the degradation of the query's QoS metrics and proved inadequate for our use cases. For parallel deployments on high end-servers, GeneaLog could provide throughput and latency figures close to those of a not-saturated system that was not capturing provenance information. The results show that GeneaLog can indeed be used in CPSs with devices of varying computational capabilities.

## 10. Related work

Although many different types of provenance exist [20], the most related to our work are *data provenance* and *workflow provenance*. In the field of databases [21], the former tracks individual data items to find from *where* a result comes from, *how* such result is produced (which operations) and *why*. The latter, on the other hand, tracks the scientific, business and data analytics workflows [22,23]. Provenance is especially challenging in big data analytics [24] since many traditional techniques need access to the whole dataset and are thus inapplicable.

In streaming applications, early work has been discussed in [25], proposing a low-latency technique for coarse-grained provenance information about the dependencies between different streams instead of individual tuples. In [6], Wang et al. deem annotation based techniques inadequate and propose a model-based provenance technique for medical systems. Apart from the need to define explicit provenance rules on each operator, their implementation requires all intermediate streams and is thus unsuitable for modern SPEs. A different approach, aimed at minimizing provenance's storage requirements, is followed in [26] 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 standard operators (cf. Section 2). The use of fine-grained data provenance in debugging streaming applications is described in [3]. Ariadne [5] is, to the best of our knowledge, the closest approach in the literature. Nonetheless, it stores all input data and uses variable-length annotations, potentially leading to degradation figures similar to the ones presented in our evaluation.

Provenance can be especially useful in Complex Event Processing (CEP) [1] where a streaming application is tasked with detecting patterns of events (e.g., traffic monitoring [27] or financial market analysis [28]). In cases where these events are critical, and they need to be explained or used to create learning structures, it might be needed to maintain the source events that led to their generation [5] CEP systems are usually implemented either using standard streaming operators (e.g., ZStream [29]) or automata (e.g., SASE [30]). GeneaLog's contribution can be used also by CEP systems relying on standard streaming operators. It would be interesting to extend GeneaLog to allow it to function with automata-based CEP systems.

Related to GeneaLog's distributed provenance approach, data streaming fault tolerance techniques maintain information about data sent from a node $A$ to a node $B$ by forwarding copies of $A$'s output tuples to *replicas* [10] or by buffering at $A$ the output tuples sent since the last backup of $B$ (for passive-standby [31]) or all that contribute to $B$'s state (for upstream-backup [32]). To the best of our knowledge, such solutions do not 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 by $B$. That is, they do not aim at minimizing the additional information they maintain based on whether the latter contributes to the application's results.

## 11. Conclusions and future work

We presented GeneaLog, a method for streaming applications' fine-grained data provenance, and its algorithmic implementations for intra- and inter-task 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 in modern CPSs. 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 prototypes on top of the Liebre and Flink SPEs, observing small throughput and latency overheads. In contrast, other state-of-the-art techniques result in at least one order of magnitude higher overheads and rapidly exhausted the available memory of the devices running the analysis.

Interesting directions for future work include optimizations for GeneaLog to reduce the transmission and serialization costs of provenance data as well as possible extensions to allow GeneaLog to work with CEP systems that rely on automata.

## Declaration of Competing Interest

We wish to confirm that there are no known conflict of interest associated with this publication.

## Acknowledgments

The work was supported by the Swedish Foundation for Strategic Research, proj. "FiC" grant nr. GMT14-0032, by the Chalmers Energy AoA framework proj. INDEED and STAMINA and by the Swedish Research Council (Vetenskapsrådet) proj. "HARE" grant no. 2016–03800.

## References

[1] H. Rger, R. Mayer, A Comprehensive Survey on Parallelization and Elasticity in Stream Processing, ACM Comput. Surv. 52(2019) 1–37.
[2] 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, Y. Li, V. Di Nicola, X. Wang, D. Maier, S. Grell, O. Nano, I. Santos, Microsoft CEP server and online behavioral targeting, Proc. VLDB Endow. 2(2009) 1558–1561.
[3] W. De Pauw, M. LeŢia, B. Gedik, H. Andrade, A. Frenkiel, M. Pfeifer, D. Sow, Visual debugging for stream processing applications, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, N. Tillmann (Eds.), Runtime Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 18–35.
[4] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts, 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, 2004, pp. 480–491.
[5] B. Glavic, K. S. Esmaili, P. M. Fischer, N. Tatbul, Efficient stream provenance via operator instrumentation, ACM Trans. Internet Technol. (TOIT) 14 (2014) 7.

[6] M. Wang, M. Blount, J. Davis, A. Misra, D. Sow, 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, 2007, pp. 95–100.
[7] R. Coppola, M. Morisio, Connected car: technologies, issues, future trends, ACM Comput. Surveys (CSUR) 49(2016) 46.
[8] Liebre, Liebre SPE, https://github.com/vincenzo-gulisano/Liebre, 2017.
[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, Bull. IEEE Comput. Soc. Techn. Committee Data Eng. 36 (2015).
[10] M. Balazinska, H. Balakrishnan, S. Madden, M. Stonebraker, Fault-tolerance in the borealis distributed stream processing system, in: Proceedings of the ACM SIGMOD International Conference on Management of Data SIGMOD, ACM, New York, 2005, pp. 13–24.
[11] V. Gulisano, StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine, Ph.D. thesis, Universidad Politécnica de Madrid, 2012.
[12] E. Kalyvianaki, M. Fiscato, T. Salonidis, P. Pietzuch, Themis: Fairness in federated stream processing under overload, in: Proceedings of the International Conference on Management of Data SIGMOD, ACM, New York, NY, USA, 2016, pp. 541–553.
[13] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, C. Fetzer, Quality-driven continuous query execution over out-of-order data streams, in: Proceedings of the ACM SIGMOD International Conference on Management of Data SIGMOD, ACM, New York, 2015, pp. 889–894.
[14] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, P. Tsigas, Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types, ACM Trans. Parallel Comput. 4(2017) 11:1–11:28.
[15] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join, IEEE Trans. Big Data(2016) 1–1.
[16] I. Walulya, D. Palyvos-Giannas, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, P. Tsigas, Viper: A module for communication-layer determinism and scaling in low-latency stream processing, Future Generat. Comput. Syst. 88 (2018) 297–308.
[17] Storm, Apache Storm, http://storm.apache.org/, 2017.
[18] Odroid-XU4, Odroid-XU4, http://www.hardkernel.com, 2016.
[19] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al., The design of the borealis stream processing engine., in: Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR), 5, Asilomar, CA, USA, pp. 277–289.
[20] P. Groth, L. Moreau, PROV-Overview. An Overview of the PROV Family of Documents, World Wide Web Consortium, 2013 https://eprints.soton.ac.uk/356854/.
[21] Y. R. Wang, S. E. Madnick, 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, 1990, pp. 519–538.
[22] J. Freire, D. Koop, E. Santos, C. T. Silva, Provenance for computational tasks: A survey, Comput. Sci. Eng. 10 (2008) 11–21.
[23] S. B. Davidson, J. Freire, Provenance and scientific workflows: Challenges and opportunities, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD '08, ACM, New York, NY, USA, 2008, pp. 1345–1350.
[24] A. Cuzzocrea, Provenance research issues and challenges in the big data era, Proceedings of the International Computer Software and Applications Conference 3 (2015) 684–686.
[25] N. N. Vijayakumar, B. Plale, Towards low overhead provenance tracking in near real-time stream filtering, in: L. Moreau, I. Foster (Eds.), Provenance and Annotation of Data, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 46–54.
[26] M. Huq, A. Wombacher, P. Apers, Adaptive Inference of Fine-grained Data Provenance to Achieve High Accuracy at Lower Storage Costs, IEEE Comput. Soc., USA, pp. 202–209. Eemcs-eprint-21400.
[27] R. Mayer, B. Koldehofe, K. Rothermel, Predictable Low-Latency Event Detection With Parallel Complex Event Processing, IEEE Int. Things J. 2 (2015) 274–286.
[28] A. Lerner, D. E. Shasha, The virtues and challenges of ad hoc + streams querying in finance, IEEE Data Eng. Bull. 26(2003) 49–56.
[29] Y. Mei, S. Madden, Zstream: A cost-based query processor for adaptively detecting composite events, in: Proceedings of the ACM SIGMOD International Conference on Management of Data SIGMOD, ACM, New York, NY, USA, 2009, pp. 193–206.
[30] J. Agrawal, Y. Diao, D. Gyllstrom, N. Immerman, Efficient pattern matching over event streams, in: Proceedings of the ACM SIGMOD International Conference on Management of Data SIGMOD, ACM, New York, NY, USA, 2008, pp. 147–160.
[31] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, Millwheel: fault-tolerant stream processing at internet scale, Proc. VLDB Endow. 6 (2013) 1033–1044.
[32] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, S. Zdonik, High-availability algorithms for distributed stream processing, in: Proceedings of the 21st International Conference on Data Engineering (ICDE'05), IEEE, Tokyo, Japan, pp. 779–790.