Efficient Stream Provenance via Operator Instrumentation

BORIS GLAVIC, Illinois Institute of Technology
KYUMARS SHEYKH ESMAILI, Technicolor R&I Lab, Paris
PETER M. FISCHER, University of Freiburg
NESIME TATBUL, Intel Labs and Massachusetts Institute of Technology

Managing fine-grained provenance is a critical requirement for data stream management systems (DSMS), not only for addressing complex applications that require diagnostic capabilities and assurance, but also for providing advanced functionality, such as revision processing or query debugging. This article introduces a novel approach that uses operator instrumentation, that is, modifying the behavior of operators, to generate and propagate fine-grained provenance through several operators of a query network. In addition to applying this technique to compute provenance eagerly during query execution, we also study how to decouple provenance computation from query processing to reduce runtime overhead and avoid unnecessary provenance retrieval. Our proposals include computing a concise superset of the provenance (to allow lazily replaying a query and reconstruct its provenance) as well as lazy retrieval (to avoid unnecessary reconstruction of provenance). We develop stream-specific compression methods to reduce the computational and storage overhead of provenance generation and retrieval. Ariadne, our provenance-aware extension of the Borealis DSMS implements these techniques. Our experiments confirm that Ariadne manages provenance with minor overhead and clearly outperforms query rewrite, the current state of the art.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Data streams, provenance, annotation, experiments

ACM Reference Format:
Boris Glavic, Kyumars Sheykh Esmaili, Peter M. Fischer, and Nesime Tatbul. 2014. Efficient stream provenance via operator instrumentation. ACM Trans. Internet Technol. 14, 1, Article 7 (July 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2633689

1. INTRODUCTION

Detecting events on streaming data is a common task in areas like environmental monitoring, smart manufacturing, compliance and security checking, as well as social media [Alvanaki et al. 2012]. Similar to other applications on data streams, event detection requires diagnostic capabilities and support for human observation [Ali et al. 2009; Glavic et al. 2011]. These requirements lead to the common need to provide *fine-grained provenance* information (i.e., at the same level as in database provenance [Cheney et al. 2009]), to trace an output event back to the input events contributing to its existence. There is a significant overlap in concepts, methods, and implementations between event detection on streaming data and generic stream processing. Thus, it is often possible to rely on the same foundations for provenance. In this article, we propose efficient

Authors' addresses: B. Glavic (corresponding author), Computer Science Department, Illinois Institute of Technology, Chicago, IL; email: bglavic@iit.edu; K. Sheykh Esmaili, Technicolor R&I Lab, Paris, France; P. M. Fischer, Computer Science Institute, University of Freiburg, Germany; N. Tatbul, Intel Labs and MIT, Cambridge, MA.

DOI: http://dx.doi.org/10.1145/2633689

fine-grained stream provenance management techniques that are generally applicable to streaming applications, including real-time event detection.

1.1. Motivating Applications

We now provide a short overview on applications and technical means for streaming event detection.

Trend and Story Detection in Social Media. Social media like Twitter or Facebook, that have reached a significant coverage of population, act as social sensors [Sakaki et al. 2010], providing insights into emerging and ongoing events that have not yet been noticed by traditional media. Given the ever-increasing volume of interactions on such social media (500 million Twitter messages per day[1]) and the required short response times for event detection, scalable streaming approaches are required. The community is addressing these challenges with a large number of algorithms and systems (e.g., EnBlogue [Alvanaki et al. 2012]). Assessing correctness, reliability, and trustworthiness of such detected events is of the utmost concern for the general public, news media as well as decision makers, such as politicians. Therefore, an ability to trace events back to their influencing factors is of great benefit. Since many of these approaches utilize common stream processing techniques, provenance can easily be mapped onto generic stream provenance techniques.

Sequence Pattern Matching. Many types of complex, higher-level events can be modeled as a sequence of lower-level events, additionally correlated on their values or times. Examples for such sequence-based event detection are fraud detection or financial market analysis [Lerner and Shasha 2003]. Models and implementations for such complex event processing systems (aka CEP) found great interest in both academia and industry. We observe two common implementation approaches which both provide expressiveness and performance: (1) In systems like ZStream [Mei and Madden 2009], the sequencing operations can be expressed using standard streaming operators. (2) In systems like SASE [Agrawal et al. 2008], the correlations are expressed as an automaton. Since these CEP systems work on sensitive applications, deal with high volumes and rates, and express complex correlations, there is also significant need for tracing, assurance, and explanation. Provenance for CEP systems based on standard streaming operators can directly be expressed with the Ariadne approach. Adaptations of our models can also be used for automata-based approaches. However, we would have to adapt provenance computation to this type of execution model.

Environmental Monitoring and Sensor Data Management. Sensor systems have become small and cheap enough to be routinely deployed in many environment monitoring scenarios and in process monitoring. The means to transfer and process data from these sensors are also affordable and reliable enough to permit continuous monitoring. Sensor readings are processed by a DSMS in order to detect critical situations, such as quality deviations, overheating of equipment, or fires. These detected events are then used for automatic corrections as well as for notifying human supervisors. Human supervisors need to understand why and how such events were triggered to be able to assess their relevance and react appropriately. In addition, these observations may be used to compute higher-level indicators and compliance with service-level agreements.

As a concrete example from this domain, Figure 1 shows a simplified continuous query that detects overheating. Two sensors feed time-stamped temperature readings to the query. Each sensor stream is filtered to remove outliers (i.e., temperature t above 350°C). The stream is aggregated by averaging the temperature over a sliding

[1] `https://blog.twitter.com/2013/new-tweets-per-second-record-and-how`.

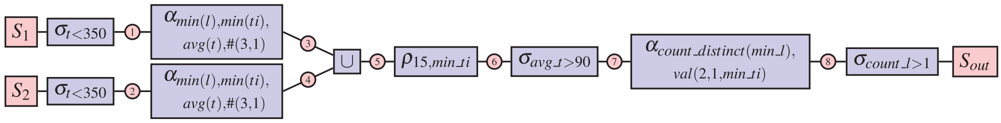

S_1
$\sigma_{t<350}$
1
$\alpha_{min(l),min(ti),avg(t),\#(3,1)}$
S_2
$\sigma_{t<350}$
2
$\alpha_{min(l),min(ti),avg(t),\#(3,1)}$
3
4
$\cup$
5
ρ_{15,min_ti}
6
$\sigma_{avg\_t>90}$
7
$\alpha_{count\_distinct(min\_l),val(2,1,min\_ti)}$
8
$\sigma_{count\_l>1}$
S_{out}

Fig. 1. Running example.

window of three temperature readings to further reduce the impact of sudden spikes. These data cleaning steps are applied to each sensor stream individually. Afterwards, readings from multiple sensors are combined for cross-validation (i.e., a union followed by a sort operator to globally order on time). The final aggregation and selection ensure that a fire alert will only be raised if at least three different sensors show average temperatures above 90°C within two time units. In this example, the user would want to understand which sensor readings caused an "overheating" event, that is, determine the tuples that belong to the *fine-grained* provenance of this event. We will use this example as our running example throughout the article.

1.2. Challenges and Opportunities

Tracking provenance to explore the reasons that led to a given query result has proven to be an important functionality in many domains, such as scientific workflow systems [Davidson et al. 2007] and relational databases [Cheney et al. 2009]. However, providing fine-grained provenance support over data streams introduces a number of unique challenges that are not well addressed by traditional provenance management techniques.

—*Online and Infinite Data Arrival.* Data streams could potentially be infinite; therefore, no global view on all items is possible. As a result, traditional methods that reconstruct provenance from the query and input data on request are not applicable.
—*Ordered Data Model.* In contrast to relational data, data streams are typically modeled as ordered sequences. This ordering can be exploited to provide optimized representations of provenance.
—*Window-Based Processing.* In DSMSs, operators like *aggregation* and *join* are typically processed by grouping tuples from a stream into windows. Stream provenance must deal with windowing behavior in order to trace the outputs of such operators back to their sources correctly and efficiently. The prevalence of aggregations leads to enormous amounts of provenance per result.
—*Low-Latency Results.* Performance requirements in most streaming applications are strict; in particular, low latency should be maintained. Provenance generation has to be efficient enough to not violate the application's latency constraints.
—*Non-determinism.* Mechanisms for coping with high input rates (e.g., load shedding [Reiss and Hellerstein 2005; Tatbul et al. 2003]) and certain operator definitions, such as windowing on system time, result in outputs that are not determined solely by the inputs. Provenance tracking should be able to cope with these types of nondeterminism that are specific to stream processing.

Conventional provenance techniques (e.g., query rewrite [Glavic and Alonso 2009]) and naive solutions (e.g., taking advantage of fast storage by dumping all inputs and inferring provenance from the complete stream data) are not sufficient to address all of the challenges previously outlined.

1.3. Contributions and Outline

In this article, we propose a novel propagation-based approach for provenance generation, called *operator instrumentation*. We use a simple definition of fine-grained provenance that is similar to *lineage* in relational databases [Cheney et al. 2009]. Our approach annotates regular data tuples with their provenance while they are being processed by a network of streaming operators. Propagation of these provenance annotations is realized by replacing the operators of the query network with operators that create and propagate annotations in addition to producing regular data tuples (we refer to this transformation as operator instrumentation). Previously, De Pauw et al. [2010] proposed an annotation propagation approach for tracking fine-grained provenance to be used in visual debugging of stream processing applications. Since the main focus is on debugging, only single-step provenance is computed, and multistep provenance is generated offline if requested. Our approach is more general and flexible, as it allows for (a) both eager or lazy provenance generation and (b) direct provenance propagation for partial as well as complete query networks. We represent provenance as sets of tuple identifiers during provenance generation. A number of optimizations enable us to decouple provenance management (generation and retrieval) from query processing. Our work makes the following contributions.

—We introduce a novel provenance generation technique for DSMS based on annotating and propagating provenance information through operator instrumentation, which allows for generating provenance for networks and subnetworks without the need to materialize data at each operator.
—We propose optimizations that decouple provenance computation from query processing.
—We present Ariadne, the first DSMS providing support for fine-grained multistep provenance.
—We provide an experimental evaluation of the proposed techniques using Ariadne. The results demonstrate that providing fine-grained provenance via optimized operator instrumentation has minor overhead and clearly outperforms query rewrite, the current state of the art.

This article is an extension of our conference paper [Glavic et al. 2013], adding (i) an analysis of streaming event detection cases and their provenance requirements, (ii) a formalization of our provenance model, (iii) a description of provenance propagation through query rewrite, and (iv) new experimental results. We refer interested readers to the conference paper for a more thorough description of the implementation and optimizations.

The remainder of this article is organized as follows: Section 2 gives an overview of our approach for adding provenance generation and retrieval to a DSMS. We introduce the stream, provenance, and annotation models underlying our approach as well as the instrumentation mechanism in Section 3. Building upon this model, we present its implementation in the Ariadne prototype in Section 4 and optimizations in Section 5. We present experimental results in Section 6, discuss related work in Section 7, and conclude in Section 8.

2. OVERVIEW OF OUR APPROACH

We generate and propagate provenance annotations by replacing query operators with provenance-aware operators (we call this *operator instrumentation*). Our approach can be used to compute either the provenance of a whole query network or just parts of the network. Provenance is modeled as a set of tuples from the input streams that are sufficient to produce a result tuple. Output tuples are annotated with sets of tuple identifiers representing their provenance.

Method	Applicable to	Runtime Overhead	Retrieval Overhead
Inversion	Invertible	None	High
Query Rewrite	Deterministic	High	None
Operator Instr.	All	Low	Medium

Fig. 2. Provenance generation methods.

2.1. Why Operator Instrumentation?

There are two well-known provenance generation techniques in the literature that we considered as alternatives to operator instrumentation for generating DSMS provenance: (1) computing inverses and (2) rewriting the query network to propagate provenance annotations using the existing operators of the DSMS. Figure 2 shows a summary of the trade-offs. *Inversion* (e.g., [Woodruff and Stonebraker 1997]) generates provenance by applying the inverse (in the mathematical sense) of an operator. For example, a join (without projection) is invertible, because the inputs can be reconstructed from an output tuple. Inversion has very limited applicability to DSMSs, because no real inverse exists for most nontrivial operators. *Query rewrite*, established in relational systems such as Perm [Glavic and Alonso 2009], DBNotes [Bhagwat et al. 2004], or Orchestra [Ives et al. 2008], generates provenance by rewriting a query network Q into a network that generates the provenance of Q in addition to the original network outputs. *Query rewrite* leads to significant additional runtime overhead and incorrect provenance for nondeterministic operators. Rewrite techniques have to duplicate parts of the query network to compute the provenance of operators, such as windowed aggregation (see Section 6.1). Assume a subnetwork q that contains nondeterministic operators is duplicated as q'. Networks q and q' may produce different results, leading to missing or incorrect provenance.

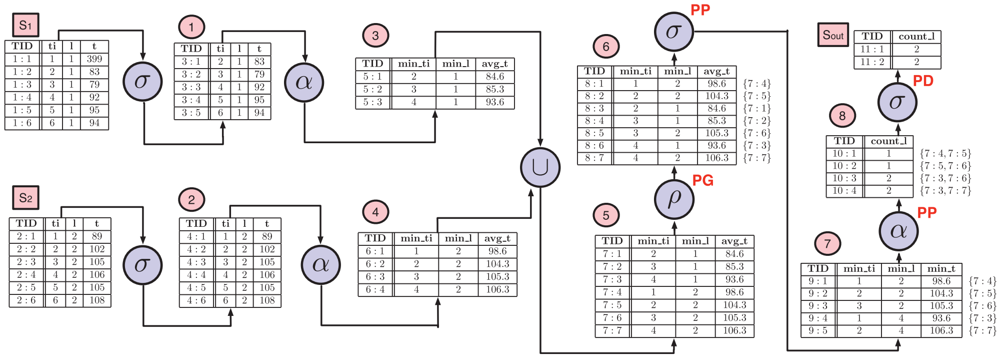
In summary, we believe that operator instrumentation is the best approach, as it is applicable to a large class of queries while maintaining low overhead for provenance computation and retrieval.

2.2. The Operator Instrumentation Approach

The key idea behind our operator instrumentation approach is to extend each operator implementation so that the operator is able to annotate its output with provenance information based on provenance annotations of its inputs. Under operator instrumentation, provenance annotations are processed in line with the regular data. In other words, the structure of the original query network is kept intact, because operators are simply replaced with their instrumented counterparts. Provenance can be traced for a single operator at a time, through a whole subnetwork, or for a complete network by instrumenting only some or all operators of the network.

For example, consider an execution of the network from Figure 1 shown in Figure 3. Here we have instrumented some operators, that is, the ones marked with *PG* or *PP* (will be explained later), to compute the provenance of the last aggregation according to the input of the b-sort operator. By propagating annotations (shown as sets on the right of each tuple), we have annotated each output tuple of the aggregation with the identifiers of tuples from stream 5 that are in its provenance.

Most issues caused by nondeterminism are dealt with in a natural way if operator instrumentation is applied, because the execution of the original query network is traced. However, the overhead introduced by provenance generation may affect temporal conditions (e.g., system time windows). In general, this effect cannot be avoided when modifying time-sensitive operations, because the output of such operations may be affected by any modification introducing computational overhead. The only way to avoid this is to execute the system in a fully simulated environment which is not



S1

TID	ti	l	t
1 : 1	1	1	399
1 : 2	2	1	83
1 : 3	3	1	79
1 : 4	4	1	92
1 : 5	5	1	95
1 : 6	6	1	94

1

TID	ti	l	t
3 : 1	2	1	83
3 : 2	3	1	79
3 : 3	4	1	92
3 : 4	5	1	95
3 : 5	6	1	94

3

TID	min_ti	min_l	avg_t
5 : 1	2	1	84.6
5 : 2	3	1	85.3
5 : 3	4	1	93.6

S2

TID	ti	l	t
2 : 1	1	2	89
2 : 2	2	2	102
2 : 3	3	2	105
2 : 4	4	2	106
2 : 5	5	2	105
2 : 6	6	2	108

2

TID	ti	l	t
4 : 1	1	2	89
4 : 2	2	2	102
4 : 3	3	2	105
4 : 4	4	2	106
4 : 5	5	2	105
4 : 6	6	2	108

4

TID	min_ti	min_l	avg_t
6 : 1	1	2	98.6
6 : 2	2	2	104.3
6 : 3	3	2	105.3
6 : 4	4	2	106.3

5

TID	min_ti	min_l	avg_t
7 : 1	2	1	84.6
7 : 2	3	1	85.3
7 : 3	4	1	93.6
7 : 4	1	2	98.6
7 : 5	2	2	104.3
7 : 6	3	2	105.3
7 : 7	4	2	106.3

6

TID	min_ti	min_l	avg_t	
8 : 1	1	2	98.6	{7 : 4}
8 : 2	2	2	104.3	{7 : 5}
8 : 3	2	1	84.6	{7 : 1}
8 : 4	3	1	85.3	{7 : 2}
8 : 5	3	2	105.3	{7 : 6}
8 : 6	4	1	93.6	{7 : 3}
8 : 7	4	2	106.3	{7 : 7}

7

TID	min_ti	min_l	min_t	
9 : 1	1	2	98.6	{7 : 4}
9 : 2	2	2	104.3	{7 : 5}
9 : 3	3	2	105.3	{7 : 6}
9 : 4	1	4	93.6	{7 : 3}
9 : 5	2	4	106.3	{7 : 7}

8

TID	count_l	
10 : 1	1	{7 : 4, 7 : 5}
10 : 2	1	{7 : 5, 7 : 6}
10 : 3	2	{7 : 3, 7 : 6}
10 : 4	2	{7 : 3, 7 : 7}

Sout

TID	count_l
11 : 1	2
11 : 2	2

Fig. 3. Query network evaluation with provenance-aware operators and provenance annotations.

Method	Applicable to	Runtime Overhead	Retrieval Overhead
Reduced-Eager	All	Full Generation (high)	Reconstruct (low)
Replay-Lazy	Deterministic	Minimal Generation (low)	Replay (high)

Fig. 4. Trade-offs for eager vs. lazy.

practical. In contrast to the query rewrite and inversion alternatives, operator instrumentation does not modify the structure of the query network. Thus, while the result of nondeterministic operations, such as a random number generator, may be affected by the overhead introduced by provenance computation, the provenance of such an operation will still correctly describe the origin of a tuple under the given result. As explained in Section 2.1, this is not necessarily the case for query rewrite. The only drawback of operator instrumentation is the need to extend all operators. However, as we will demonstrate in Section 4.2, this extension can be implemented with reasonable effort.

With operator instrumentation, provenance can be generated either *eagerly* during query execution (our default approach) or *lazily* upon request. We support both types of generation, because their performance characteristics in terms of storage, runtime, and retrieval overhead are different (see Figure 4). This enables the user to trade runtime overhead on the original query network for storage cost and runtime overhead when retrieving provenance. However, for lack of space we will mostly focus on the eager method in the remainder of this article.

Reduced-Eager. Figure 5 shows an example for how we instrument a network for *eager* provenance generation. We temporarily store the input tuples for the instrumented parts of the network (e.g., for input streams S_1 and S_2, since we want provenance for the entire query network). The tuples in the output stream of the instrumented network carry provenance annotations as just described, that is, each output is annotated with the set of identifiers of the tuples in its provenance. Using tuple identifiers as an internal provenance representation reduces the size of provenance annotations in comparison to full input tuples. However, these identifiers are meaningless to a user and thus have to be replaced with actual tuples before returning the provenance to the user. We *reconstruct* provenance for retrieval from the identifier annotations using a new operator called *p-join* ($\ltimes$). For each output tuple t, this operator retrieves all input tuples in the provenance using the set of identifiers from the provenance annotation and outputs all combinations of t with a tuple from its provenance. Each of these combinations is emitted as a single tuple to stream P. For example, consider the first output tuple $t = (1)$ of the final aggregation operator in Figure 3. The provenance annotation of this tuple contains two tuple identifiers, 7:4 and 7:5 . If we feed this annotated stream

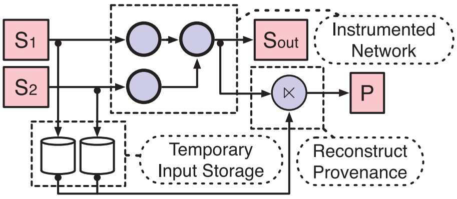


Fig. 5. Reduced-eager operator instrumentation.

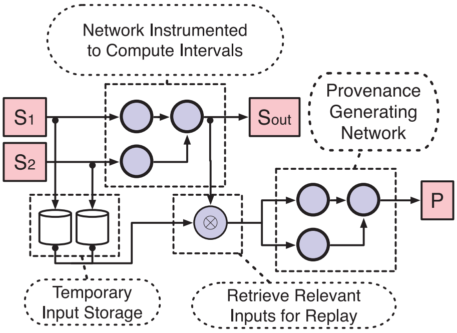

Network Instrumented to Compute Intervals
S1
S2
Sout
Provenance Generating Network
P
Temporary Input Storage
Retrieve Relevant Inputs for Replay

Fig. 6. Replay-lazy.

into a p-join operator, the operator would look up the tuples from the cached input stream 5 using their identifiers. For tuple t, the p-join would generate two result tuples (1, 1, 2) and (1, 2, 2) by combining t with the tuples (1, 2) and (2, 2) from stream 5.

We call this approach *reduced-eager*, because we are eagerly propagating a reduced form of provenance (the tuple identifier sets) during query execution and lazily reconstructing provenance independent of the execution of the original network. In comparison with using sets of full tuples as annotations, this approach pays a price for storing and reconstructing tuples. However, because compressed representations can be used, this cost is offset by a significant reduction in provenance generation cost (in terms of both runtime and latency). Since *reconstruction* is separate from generation, we can often avoid reconstructing complete provenance tuples during provenance retrieval (e.g., if the user only requests provenance for some results (query over provenance)).

Replay-Lazy. Instead of generating provenance eagerly while the query network is running, we are also able to generate provenance lazily in order to decouple provenance generation from the query network execution. Since DSMSs have to deal with high input rates and low latency requirements, the runtime overhead to the critical data processing path incurred by eager provenance computation may be too high for some applications. Decoupling provenance computation from query processing enables us to reduce the runtime overhead on the query network and outsource provenance generation to a separate machine. This improves performance for both query processing and provenance computation.

For deterministic networks, we can realize lazy generation, as shown in Figure 6. We instrument the network in a similar fashion as for reduced eager by temporarily storing input streams and propagating annotations. However, instead of using sets of tuple identifiers, we annotate each tuple with intervals of tuple identifiers (one per input stream) represented as the minimal and maximal identifiers of each interval. These intervals are of constant size, can be computed efficiently, and are guaranteed to be supersets of the actual provenance. To generate the actual provenance for a tuple t, we retrieve all input tuples contained in the intervals for t and replay them through a copy of the original query network instrumented for provenance generation (e.g., using the reduced eager instrumentation previously discussed). We call this approach *replay-lazy*. For example, assume that we want to compute provenance for the output of the aggregation operator producing stream 3 according to input stream S_1 from Figure 3. The first output tuple 5:1 of the aggregation is generated based on a window containing the first three tuples of stream 1. These tuples are produced from tuples 1:2, 1:3, and 1:4 of stream 1. For *replay-lazy*, we annotate tuple 5:1 with the interval [1:2, 1:4] covering its provenance. To compute the provenance of t, we retrieve all tuples in this interval from the temporary storage of stream 1 and replay them through a provenance generating copy of the subnetwork containing streams 1 to 3. This will reproduce t annotated with its correct provenance. *Replay-lazy* is only applicable to deterministic networks. Using constant size intervals instead of full provenance annotations reduces the runtime overhead, but results in higher retrieval costs due to the replay.

3. PROVENANCE PROPAGATION BY OPERATOR INSTRUMENTATION

Based on the stream data and query model of Borealis [Abadi et al. 2005], we now introduce our stream provenance model and discuss how to instrument queries to annotate their outputs with provenance information. A thorough formal treatment can be found in Glavic et al. [2012]. We first introduce our data and query model and then introduce our definition of provenance. Next, focusing on a set of core streaming operators, we define instrumented operator versions that produce streams where each tuple is annotated with its provenance and show that they produce provenance according to our definition. Finally, we elaborate on how to extend our model for generic operators.

3.1. Data and Query Model

We model a stream $S =< i_1, i_2, \ldots >$ as a possibly infinite sequence of stream items: A stream consists of a fixed type of stream items; either tuples, windows, or join-windows. A tuple $t = [a_1, \ldots, a_n]$ is an ordered list of attribute values (here each a_i denotes a value). We assume the existence of tuple-identifiers (*TID*) in the form of *stream-id:tuple-id* that uniquely identify tuples within a query network. We use $\tau(t)$ to denote the identifier of tuple t. A window $w =< t_1, \ldots, t_n >$ is a sequence of tuples, and a join-window $[t, w]$ is a pair of a tuple and a window. These types of stream items will be used later in the definition of aggregation and join. For a stream S, we use $S[i]$ to denote the ith stream item in S and $S[i, j]$ to denote the stream containing the ith up to the jth stream item of S. We use $H(S)$ (the head) as a shortcut for $S[0]$ and $T(S)$ (the tail) as a shortcut for $S[1, \infty]$. We use $S_1 \mid\mid S_2$ to denote the concatenation of two sequences (or an item and a sequence).

A *query network* is a directed acyclic graph (DAG) in which nodes and edges represent streaming operators and input/output streams, respectively. Each stream operator in a query network takes one or more streams as input and produces one or more streams as output. The query algebra we use here covers all the streaming operators from Abadi et al. [2005]. Each operator is defined recursively using the following notation.

Selection. A selection operator $\sigma_C(S)$ with predicate C filters out tuples from an input stream S that do not satisfy the predicate C.

$$\sigma_C(S) = \begin{cases} H(S) \mid\mid \sigma_C(T(S)), & \textit{if } H(S) \models C, \\ \sigma_C(T(S)), & \textit{else}. \end{cases}$$

Projection. A projection operator $\pi_A(S)$ with a list of projection expressions A (e.g., attributes, function applications) projects each input tuple from stream S on the expressions from A.

$$\pi_A(S) = H(S).A \mid\mid \pi_A(T(S)).$$

Aggregation. An aggregation operator $\alpha_{agg,\omega}(S)$ groups its input S into windows using the window function ω and computes the aggregation functions from list $agg = (agg_1(a_1), \ldots, agg_n(a_n))$ over each window generated by ω. An aggregation function $agg_i(a_i)$ computes a single attribute value from all values of attribute a_i in a window w. We denote the application of an aggregation function agg to a window w as $agg(w)$.

$$\begin{aligned} \alpha_{agg,\omega}(S) &= a(\omega(S)), \\ a(S) &= [agg_1(H(S)), \ldots, agg_n(H(S))] \mid\mid a(T(S)). \end{aligned}$$

As an example for a typical window operator, consider the count-based window function $\#(c, s)$ that groups a consecutive input tuple sequence (length c) into a window and slides by a number of tuples s before opening the next window. The value-based window function $val(c, s, a)$ groups a consecutive sequence of tuples into a window if their values in attribute a differ less than c from the attribute value of the first tuple in the window. The slide s determines how far to slide on a before opening the next window. Note that value-based windows subsume the concept of time-based windows by using a time attribute as the attribute for determining the window boundaries.

$$\begin{aligned} \#(c, s)(S) &=< S[0, c] >\mid\mid \#(c, s)(S[s, \infty]), \\ val(c, s, a)(S) &=< \sigma_{a \leq H(S).a+c}(S) >\mid\mid val(c, s, a)(\sigma_{a > H(S).a+s}(S)). \end{aligned}$$

Join. A join operator $\bowtie_{C,\phi} (S_1, S_2)$ joins two streams S_1 and S_2 by applying the join window function ϕ to S_1 and S_2. A join window function models the buffering behavior of stream joins. For each tuple t from the left input stream, a join window for t contains all tuples from the right input stream that were in the buffer during the time tuple t was in the buffer. For each join window $j = [t, w]$, the join operator outputs all pairs of tuples $[t, t']$ where $t' \in w$, and the join condition C is fulfilled. The following definition of join first groups the input into join windows ($\phi(S_1, S_2)$) and then uses WINJOIN to iterate over all combinations of tuple t with a tuple from window w for each join window $j = [t, w]$.

$$\begin{aligned} \bowtie_{C,\phi} (S_1, S_2) &= \textsc{join}(\phi(S_1, S_2)), \\ \textsc{join}(S) &= \textsc{winjoin}(H(S)), \mid\mid \textsc{join}(T(S)), \\ \textsc{winjoin}(j) &= \begin{cases} [j.t, H(j.w)] \mid\mid \textsc{winjoin}([j.t, T(j.w)]), & \textit{if } [j.t, H(j.w)] \models C, \\ \textsc{winjoin}([j.t, T(j.w)]), & \textit{else}. \end{cases} \end{aligned}$$

As an example for join-windowing, consider *value-based* join-windowing ($jval(a_1, a_2, s)$) that groups each tuple t from the left stream with all tuples from the right stream that have an a_2 attribute value between $t.a_1$ and $t.a_1 + s$.

$$\begin{aligned} jval(a_1, a_2, s)(S_1, S_2) &= [H(S_1), < \sigma_C(S_2) >] \mid\mid jval(a_1, a_2, s)(T(S_1), S_2), \\ C &= a_2 \geq H(S_1).a_1 \wedge a_2 \leq H(S_1).a_1 + s. \end{aligned}$$

Union. A union operator $\cup(S_1, S_2)$ merges tuples from two input streams S_1 and S_2 into a single stream based on their arrival order. The arrival order of tuples may depend on the input data and operator scheduling policies of the DSMS. There is no clean way to model such behavior in an abstract and deterministic operator model. We solve this problem by encapsulating the nondeterministic arrival order in a function O that maps a tuple to its arrival time stamp (at the union operator). Using this function, union is defined as

$$\cup(S_1, S_2) = \begin{cases} H(S_1) \mid\mid \cup(T(S_1), S_2), & \textit{if } O(H(S_1)) < O(H(S_2)), \\ H(S_2) \mid\mid \cup(S_1, T(S_2)), & \textit{else}. \end{cases}$$

B-Sort. A b-sort operator $\rho_{s,a}(S)$ with slack s and an order-on attribute a applies a bounded-pass sort with buffer size $s + 1$ on its input, that is, once the buffer is full, the sort emits the smallest tuple in sort order from the buffer for every new arriving tuple. Thus, it produces an output that is approximately sorted on a. Let $\textsc{sort}(S, a)$ denote a function that sorts a sequence S on attribute a.

$$\rho_{s,a}(S) = \textsc{bsort}(S, a, s, 0),$$
$$\textsc{bsort}(S, a, s, i) = \textsc{sort}(S[0, s + i], a)[i] \mid\mid \textsc{bsort}(S, a, s, i + 1).$$

Example 1. Figure 3 shows an execution of the network introduced in Figure 1 for a given input. For now, ignore the annotations on operators and tuples in streams 6 to 8. Both input streams (S_1 and S_2) have the same schema with attributes time (ti), location (l), and temperature (t). The leftmost selections drop temperature outliers. The results of this step are grouped into windows of three tuples using slide one. For each window, we compute the minimum of time (to assign each aggregated tuple a new time value) and location (the location is fixed for one stream, thus, the minimum of the location is the same as the input location), and average temperature. The aggregated streams are merged into one stream ($\cup$) and sorted on time. We then filter out tuples with temperature values below the overheating threshold and compute the number of distinct locations over windows of two time units. Tuples with fewer than two distinct locations are filtered out in the last step. For instance, in the example execution shown in Figure 3, the upper-left selection filters out the outlier tuple 1:1 $(1, 1, 399)$. The following aggregation groups the first three result tuples into a window and outputs the average temperature (84.6), minimum time (2), and location (1).

3.2. Provenance Model and Annotated Streams

We use a simple provenance model that defines the provenance of a tuple t in a stream S of a query network q as a set of tuples from input (or intermediate) streams of the network. We use $P(q, t, I)$ to denote the *provenance set of a tuple* t from one of the streams of network q with respect to inputs from streams in a set I. For instance, if t is a tuple in stream 3 of the example network shown in Figure 1, then $P(q, t, \{S_1\})$ denotes the set of tuples from input stream S_1 that contributed to t. We omit I if we compute the provenance according to the input streams of the query network.

Note that we require I to be chosen such that the paths between streams in I and S (the stream of t) form a proper query network. For instance, assume that t is a tuple from stream 5 in the network shown in Figure 3. $P(q, t, \{1, 2\})$ denotes the set of tuples from streams 1 and 2 that contributed to t. $P(q, t, \{2\})$ would be undefined, because only one of the inputs of the union is included. For the remainder of this section we will limit the discussion to query networks with a single output stream O and provenance of tuples in that output stream. The concepts introduced in this section extend naturally to networks with multiple output streams.

Formally, our work is based on a declarative characterization of provenance, which is used to determine whether the provenance generated by instrumented networks introduced in the next section captures intuitive properties of provenance. To simplify the exposition, we will limit the discussion to computing the provenance of an output stream according to the input streams of the query network. This discussion naturally extends to provenance computations for a partial query network. The declarative characterization of provenance captures two intuitive properties. (1) Sufficiency: the provenance of tuple t is sufficient for producing t, that is, if we evaluate the query network over minimal prefixes of the input streams (a prefix of length n is a subsequence $S[0, n]$ of a stream S, see Section 3.1) that include the provenance, then the result will contain t. (2) *Distinguishability*: if t is the nth occurrence of a tuple with the same values in a stream O, then replaying a minimal prefix including the provenance will produce at least n duplicates of t. This guarantees that we actually capture the provenance of t and not of another tuple from the same stream with the same values. To be able to formalize these intuitions, we need to introduce some preliminary concepts first. The minimal prefix $S \uparrow \mathcal{M}$ of an input stream S of a query network q according to a set of tuples $\mathcal{M}$ is the shortest prefix of S that includes all items of $\mathcal{M}$, and if q is executed over this prefix, only windows that also exist in the original execution of q are produced. The requirement that only original windows are produced ensures that, for example, the buffer of a b-sort operator is filled so that the necessary tuples to produce an output can be emitted. We use the same notation for sets of streams I, that is, $I \uparrow \mathcal{M}$. For example, the minimum prefix 7 ↑ {9:2 , 9:4 } of tuples 9:2 and 9:4 from stream 7 in the running example (Figure 3) contains tuples 9:1 to 9:4.

Definition 3.1 (*Provenance Set*). Let $\textsc{Count}(S, t)$ denote the number of tuples in stream S that are exact copies of t, that is, that differ from t only in their tuple identifier. Similarly, $\textsc{DupPos}(S, t)$ denotes the number of duplicates of t that are in the smallest prefix of stream S that includes t. The provenance set $P(q, t, I)$ of a tuple t from a stream S of a query network q is a subset of tuples from I that fulfills the following conditions.

—Tuple t is in the result of executing q over the minimal prefix of the provenance of t.

$$t \in q(I \uparrow P(q, t, I)).$$

—Replaying minimal prefixes of streams in I that include the provenance produces the correct number of copies of t before producing t.

$$\textsc{Count}(q(I \uparrow P(q, t, I)), t) \geq \textsc{DupPos}(S, t).$$

A possible way to define the provenance for our query operators based on this characterization is as follows: for *selection* and *projection*, the provenance of t consists of the provenance of the corresponding input tuple. The same is true for *union* and *b-sort*, since only a single tuple is contributing to t. For example, tuple 9:1 in the network shown in Figure 3 was generated by the selection from tuple 8:1 . Thus, the provenance set of this tuple is {7:4 }, the same as the provenance set of tuple 8:1 . For *join*, the union of the provenance sets of the join partners generating t constitutes the provenance. Finally, the provenance set for t in the result of an *aggregation* is the union of the provenance sets for all tuples from the window used to compute t.

Based on the concept of provenance sets, we define streams of tuples that are annotated with their provenance sets. For a query network q, the *provenance annotated stream* (PAS) $P(q, O, I)$ for a stream O according to a set of streams I is a copy of stream O, where each tuple t is annotated with its corresponding provenance set $P(q, t, I)$. In the following, we will omit the query parameter q from provenance sets and PAS if it is clear from the context.

Example 2. Consider the PAS $P(6, \{5\})$ for the output of the b-sort operator according to its input shown in Figure 3 (provenance sets are shown on the right of tuples). Each output t of the b-sort is annotated with a singleton set containing the corresponding tuple from the b-sort's input, for example, tuple 8:1 is derived from 7:4 . Now consider the PAS for the output of the last aggregation in the query according to the input of the b-sort ($P(8, \{5\})$). Each output is computed using information from a window containing two tuples with one tuple overlap between the individual provenance sets. For example, tuple 10:2 is derived from a window with provenance {7:5} and {7:6}, and tuple 10:3 is derived from a window with provenance {7:3, 7:6}. The set $P(q, 10{:}3, \{5\}) = \{7{:}3, 7{:}6\}$ fulfills the two conditions of Definition 3.1. The minimal prefix of stream 5 containing $P(q, 10{:}3, \{5\})$ that does not produce new windows is $<7{:}1, \ldots, 7{:}6>$. Replaying this prefix through the query network produces tuples 10:1 to 10:3. Thus, the first condition, 10:3 being in the result of replaying the prefix, is fulfilled. The second condition is also fulfilled, because 10:3 is the first tuple with $count\_1 = 2$ in both the original result and the result of replaying the prefix.

3.3. Instrumenting Operators and Networks for Annotation Propagation

We now discuss how to instrument a query network q to generate the PAS for a subset of the streams in q by replacing all or a subset of the operators with their annotating counterparts. We introduce three types of *instrumented operators* that handle streams annotated with provenance information. Afterwards, we present annotating versions of stream algebra operators.

Provenance Generator (PG). The provenance generator version $PG(o)$ of an operator o annotates its outputs with provenance according to its inputs. The purpose of a PG operator is to generate a PAS from input streams without provenance annotations. In an instrumented network, we will attach a provenance generator to each stream in I. For each output stream S of the operator o, $PG(o)$ creates $P(S, input(o))$, where $input(o)$ are the input streams of operator o.

Provenance Propagator (PP). The provenance propagator version of each operator consumes annotated input streams and produces annotated output streams by combining provenance from its inputs based on the semantics of the operator. For simplicity, let us explain the concept for an operator o with a single output O and a single input PAS $P(S, I)$. The PP version $PP(o)$ of o will output $P(O, I)$, that is, the output will be annotated with provenance sets of O according to I. This is achieved by modifying the annotations in the input streams according to the provenance behavior of the operator o. We use the PP version of operators in an instrumented network $P(q, O, I)$ to propagate provenance along paths between streams in I and stream O.

Provenance Dropper (PD). The provenance dropper version $PD(o)$ of an operator o removes annotations from the input before applying operator o. Provenance droppers are used to remove annotations from streams in networks with partial provenance generation.

Definition 3.2 (*Operator Versions*). Figure 7 shows the PG and PP versions of aggregation (the annotating versions of the remaining operators can be found in [Glavic et al. 2012]). We represent the provenance annotations as an additional attribute P that stores a set of tuple identifiers. For example, $t.P$ returns the provenance annotation of a tuple t. For convenience, we have marked the annotation part in bold red. Recall that $\tau(t)$ denotes the identifier of tuple t.

PG operators create a TID set from the TIDs of all input tuples that contribute to a tuple. All PP operator versions union the provenance annotations from the inputs that contribute to a tuple t. For example, the PG version of selection generates an annotated output stream, where the provenance set of each output tuple t contains the corresponding input tuple, and the PP version outputs the input tuples with unmodified provenance sets (for tuples that fulfill the selection condition). The PG operator for aggregation annotates each output tuple t with a provenance set $P(w)$ that consists of all identifiers for tuples in the input window w that generated t, and

Provenance Generators

$$\alpha^{PG}_{agg,\omega}(S) = a(\omega(S))$$
$$a(S) = [agg_1(H(S)),\ldots,agg_n(H(S)),\mathbf{P(H(S))}] \mid\mid a(T(S))$$
$$\mathbf{P(w)} = \{\tau(\mathbf{t}) \mid \mathbf{t} \in \mathbf{w}\}$$

Provenance Propagators

$$\alpha^{PP}_{agg,\omega}(S) = a(\omega(S))$$
$$a(S) = [agg_1(H(S)),\ldots,agg_n(H(S)),\ \bigcup_{\mathbf{t}\in\mathbf{H(S)}} \mathbf{t.P}] \mid\mid a(T(S))$$

Fig. 7. Provenance generator and propagator versions of the aggregation operator.

ALGORITHM 1: InstrumentNetwork Algorithm

```
procedure InstrumentNetwork($q$, $O$, $I$)
    $mixed \leftarrow \emptyset$
    for all $o \in q$ do                                  ▷ Find operators with mixed usage
        if $\exists S, S' \in input(o) : S \in I \wedge S' \notin I$ then
            $mixed \leftarrow mixed \cup input(o)$
    for all $S \in (mixed \cap I)$ do                     ▷ Add projection wrappers
        $S \leftarrow \Pi_{schema(S)}(S)$
    for all $o \in q$ do                                  ▷ Replace operators
        if $\exists S \in I$ : hasPath($S, o$) $\wedge$ hasPath($o$, $O$) then
            if $\exists S' \in input(o) : S' \in I$ then
                $o \leftarrow PG(o)$
            else
                $o \leftarrow PP(o)$
    for all $o \in q$ do                                  ▷ Drop annotations
        if $O \in input(o)$ then
            $o \leftarrow PD(o)$
```

the *PP* operator annotates each output tuple t with the union of the provenance sets of all tuples in the window that generated t.[2]

Networks with Annotation Propagation. Using the PG and PP versions of operators, we have the necessary means to generate provenance for a complete (or parts of a) query network by replacing all (or some) operators with their annotating counterparts. PD versions of operators are used to remove provenance annotations from streams that are further processed by the network. We use Algorithm 1 to instrument a network q to compute a PAS $P(O, I)$. We first normalize the network to ensure that the inputs to every operator are either (1) only streams from I or (2) contain no streams from I. This step is necessary to avoid having operators that read from both streams in O, and not in I, because the annotation propagation behavior of these operators is neither correctly modeled by their PG nor PP version. We wrap each stream S in I that is connected to such an operator in a projection on all attributes of the schema of S. This does not change the results of the network, but guarantees that we can use solely PG and PP operators to generate a PAS.[3] We then iterate over all operators in the query network and replace each operator that reads solely from streams in I with its PG version, and all remaining operators on paths between streams in I and O are replaced with their PP versions. Finally, all non-instrumented operators reading from O are replaced by their PD version. This step is necessary to guarantee that non-instrumented operators are not reading from annotated streams.

[2] We assume no knowledge about the semantics of aggregation functions. The approach can easily be extended to support more concise provenance for functions, such as min/max, where the output only depends on some tuples in the window.

[3] Adding operator types to the algebra that deal with a mix of annotated and non-annotated streams does not pose a significant challenge. However, for simplicity we refrain from using this approach.

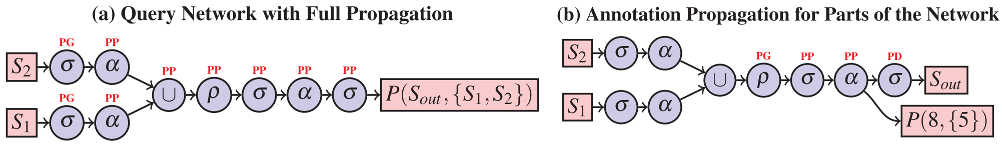

(a) Query Network with Full Propagation **(b) Annotation Propagation for Parts of the Network**

Fig. 8. Annotating query networks.

A query network instrumented to compute a PAS $P(O, I)$ generates additional PAS as a side effect. Each PP operator in the modified network generates one or more PAS (one for each of its outputs) according to the subset of I it is connected to. Thus, additional PAS are generated for free by our approach. We use $P(q)$ (called *provenance generating network*) to denote a network that generates the PAS for all output streams of network q according to all input streams of q. Such a network is generated using a straight-forward extension of Algorithm 1 to sets of output streams.

Example 3. Two provenance generating versions of the example network are shown in Figure 8 (the operator parameters are omitted to simplify the representation). Figure 8(a) shows $P(q)$, that is, the annotating version of q that generates the PAS $P(S_{out}, \{S_1, S_2\})$ for output stream S_{out} according to all input streams (S_1 and S_2). The leftmost selection operators in the network are only attached to input streams and thus are replaced by their PG versions. All other operators in the network are replaced by PP operators. The query network shown in Figure 8(b) generates the PAS $P(8, \{5\})$ (An example execution was shown in Figure 3). The output stream of the rightmost aggregation is annotated with provenance sets containing tuples of the b-sort operator's input stream. The right-most selection is replaced with its PD version to drop provenance annotations before applying the selection. This is necessary to produce the output stream S_{out} without annotations.

Having defined the annotating versions of each operator, it remains to show that the provenance produced by these operators complies with our definition of stream provenance (Definition 3.1).

THEOREM 1. *A network instrumented to compute $P(q, O, I)$ using Algorithm* 1 *annotates each tuple in O with provenance $P(q, t, I)$ that fulfills the conditions of Definition* 3.1.

PROOF. Sketch: The proof is by induction over the structure of a query network using the semantics of each operator. Given that we have established that correct provenance is computed by $P(q', O', I')$ for a subnetwork q' with n operators, we have to prove that the same holds for an extension of q' with an additional operator o. This operator may either be applied to the output O' or be placed on a path between one of the streams in I' and O'. As an example, consider the case where we add an additional selection over O'. We then have to show that all the conditions of Definitions 3.1 are fulfilled for each annotation generated by the new selection. We make use of the induction assumption that in O' each tuple is correctly annotated with its provenance. Thus, for example, the first condition will trivially hold, because each output tuple t of the selection will be annotated with the provenance of the corresponding input tuple t'. Replaying the minimal prefix including the provenance will produce t' which in turn will cause t to be in the result of the selection. □

3.4. Extending the Ariadne Model for Generic Operators

The Ariadne model can be easily extended to other common streaming operators. For systems that implement an operator-based query language, we can apply the approach used for the Borealis algebra. For systems that support more generic operators, for example, user-defined stream operators written in a generic programming language, we can apply techniques similar to Cui et al. [2000]. This approach classifies operators according to their input behavior and has introduced generic provenance tracing procedures for each class. Operators that work on infinite input but treat each data item individually (one-to-one mapping) can be handled like selection or projection. While these classes cover most of the common built-in expressions and user-defined functions, one challenging case remains: operators that perform their own state management over infinite sequences (e.g. advanced window operators, pattern matching, exponential decay, or punctuations). In such a case, explicit modeling of provenance on the basis of the formalism and the operator semantics is needed. The only way to avoid that would be to statically or dynamically analyze the dataflow of the code implementing the operator which is either too expensive (dynamic code analysis) or too imprecise (static code analysis). Typically, however, such complex operators are system-defined and not provided by users, so the implementation needs to be done once by the platform designers, not platform users. Understanding all aspects of such complex operators is part of future research, but given our previous work on such operators (SECRET [Botan et al. 2010], pattern matching [Fischer et al. 2010]), we do not foresee any fundamental problems to model and implement provenance.

4. IMPLEMENTATION

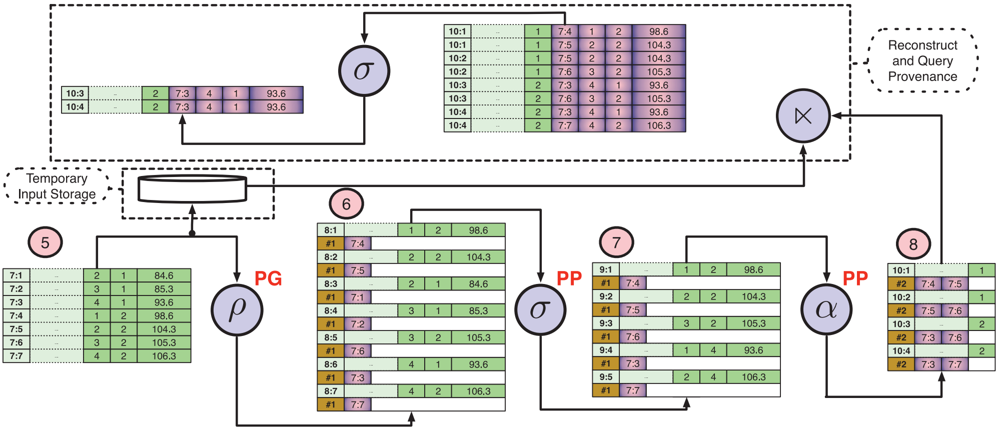
In this section, we present the implementation of the Ariadne prototype. Given the overall architecture (outlined in Section 2) and the provenance propagation model (Section 3), three aspects are now of interest: (1) representation of provenance annotations during the computation, (2) implementation of PG and PP operators, and (3) storing and retrieving the input tuples for *reduced-eager*.

4.1. Provenance Representation as Annotations

The physical representation of provenance annotations and mechanism for passing them between operators is a crucial design decision, because it strongly influences the runtime overhead and implementation of operators. It is important to note that while Borealis uses fixed-length tuples, the provenance annotations consist of TID sets of variable size. To transfer provenance between operators we can either (i) split TID sets into fix-length chunks and stream these chunks over standard Borealis queues, (ii) implement a new type of information passing between operators, or (iii) modify the queuing mechanisms to support variable-length tuples. We chose to split large TID sets into fixed-length chunks, as it is the least intrusive alternative (large parts of the code rely on fix-length tuples) and retains the performance benefits of fixed-size tuples (e.g., less indirection when accessing tuple values). We serialize the provenance (TID set) for a tuple t into a list of tuples that are emitted directly after t. Each of these tuples stores multiple TIDs from the set. The first tuple in the serialization of a TID set has a small header (same size as a TID) that stores the number of TIDs in the set. Given that the size of a TID in Borealis is 8 bytes (actually `sizeof(long)`), we are saving at least an order of magnitude of space (and tuples propagated) compared to using full tuples. We adapted the TID assignment policy to generate globally unique TIDs that are assigned as contiguous numbers according to the arrival order at the input streams. If stream-based tuple lookup becomes necessary, we could reserve several bits of a TID for storing the stream ID.

4.2. Provenance Annotating Operator Modes

We extend the existing Borealis operators with new operational modes to implement PG, PP, and PD operators. Operators in both PG- and PP-mode need to perform



10:3	...	2	7:3	4	1	93.6
10:4	...	2	7:3	4	1	93.6

10:1	...	1	7:4	1	2	98.6
10:1	...	1	7:5	2	2	104.3
10:2	...	1	7:5	2	2	104.3
10:2	...	1	7:6	3	2	105.3
10:3	...	2	7:3	4	1	93.6
10:3	...	2	7:6	3	2	105.3
10:4	...	2	7:3	4	1	93.6
10:4	...	2	7:7	4	2	106.3

7:1	...	2	1	84.6
7:2	...	3	1	85.3
7:3	...	4	1	93.6
7:4	...	1	2	98.6
7:5	...	2	2	104.3
7:6	...	3	2	105.3
7:7	...	4	2	106.3

8:1	...	1	2	98.6
#1	7:4			
8:2	...	2	2	104.3
#1	7:5			
8:3	...	2	1	84.6
#1	7:1			
8:4	...	3	1	85.3
#1	7:2			
8:5	...	3	2	105.3
#1	7:6			
8:6	...	4	1	93.6
#1	7:3			
8:7	...	4	2	106.3
#1	7:7			

9:1	...	1	2	98.6
#1	7:4			
9:2	...	2	2	104.3
#1	7:5			
9:3	...	3	2	105.3
#1	7:6			
9:4	...	1	4	93.6
#1	7:3			
9:5	...	2	4	106.3
#1	7:7			

10:1	...	1
#2	7:4 7:5	
10:2	...	1
#2	7:5 7:6	
10:3	...	2
#2	7:3 7:6	
10:4	...	2
#2	7:3 7:7	

Fig. 9. Provenance-enabled query network with retrieval.

three steps: (1) retrieve existing provenance-related information from the input tuples, (2) compute the provenance, and (3) serialize provenance annotations along with data tuples. These steps have a lot of commonalities: Serialization (step 3) is the same for all operators. Retrieval (step 1) differs only slightly for PG and PP modes, but is again the same for all operators. We factored out these commonalities into a so-called *provenance wrapper*. The operator-specific parts of the provenance-wrapper are fairly small and straightforward for most operators. The most complicated case is aggregation, in particular with overlapping windows: each output tuple may depend on several input tuples, and each input tuple may contribute to several output tuples. This requires fairly elaborate state management, including merging and sharing TID sets.

Example 4. Figure 9 shows the provenance computation for the annotating network from Figure 8(b). Recall that this network generates $P(q, 8, \{5\})$. Provenance headers are prefixed with # and TIDs in a provenance tuple are highlighted with shaded background. For instance, the aggregation operator uses the provenance wrapper to merge the TID sets from all tuples in a window and emit them as the TID set for the result tuple produced for this window.

4.3. Input Storage and Retrieval

As mentioned before, we apply a *reduced-eager* approach which requires preservation of input tuples at PG operators to be able to reconstruct fully-fledged provenance from TID sets for retrieval.

Input Storage at PG Operators. We utilize connection points (CP) [Ryvkina et al. 2006] to provide temporary storage for tuples that pass through a queue. If a query network q is instrumented to compute a PAS $P(O, I)$, then we add a connection point to each stream in I, that is, the streams that are inputs of provenance generators. We rely on a timeout (or tuple count)-based strategy for removing old tuples from storage, adapted to the retrieval pattern of the application.

If expiry is desired (i.e., the user is not storing the inputs anyways for other purposes), we can use the covering intervals in many cases as means to immediately prune noncovered input. The requirement for such pruning is that bounds exists on the arrival order of covering intervals (e.g., later intervals cannot extend further into the past than already pruned areas). Based on this observation, a static analysis of the query network can be used to determine safe count-based expiration settings for many queries. As a counterexample, consider the b-sort operator. This operator may keep a tuple in its buffer for an arbitrarily long interval of time. For this type of operator, the provenance of an output may depend on an arbitrarily "old" tuple from the input streams. In this case, we either have to keep the whole input to guarantee correct provenance, accept that the provenance is not complete if we set a time-out, or rely on application knowledge to determine when the pruning needs to be performed. In many scenarios, such a requirement is intuitively understood by developers and users, and the relevant knowledge is readily available. Using the provenance itself for more data-directed dynamic expiration as well as utilizing write-optimized, possibly distributed storage technologies are interesting avenues for future work. For example, we could dynamically inspect provenance to learn over time what are "safe" boundaries for pruning.

P-Join. Similar to the approach in Glavic and Alonso [2009], we have chosen to represent provenance to the consumer using Borealis' data model. For each result tuple t with a provenance annotation set P, we create as many duplicates of t as there are entries in P, and attach one tuple from the provenance set to each of these duplicates. This functionality is implemented as a new operator called *p-join*. A p-join $\ltimes(S, CP)$ joins an annotated stream S with a connection point CP to output tuples joined with tuples from their provenance.

Example 5. The relevant part of the running example network with retrieval is shown in Figure 9. Recall that this network was instrumented to generate $P(q, 8, \{5\})$. Hence, a CP (the cylinder) is used to preserve tuples from stream 5 for provenance retrieval.

5. OPTIMIZATIONS

Certain stream processing challenges call for additional optimizations beyond *reduced-eager*. (1) Windowed aggregation produces large amounts of provenance. (2) Computing provenance on the fly to deal with the transient nature of streams increases runtime and latency. We address these challenges through compressed provenance representations (reduces overhead) and lazy provenance computation and retrieval techniques (decouples query execution from provenance generation).

Provenance Compression. The methods we developed for TID set compression range between generic data compression to methods which exploit data model and operator characteristics. *Interval encoding* compresses contiguous sub-sequences of TIDs by replacing them with intervals (e.g., a TID set $\{1, 2, 3, 4, 6, 7, 8\}$ would be encoded as $[1, 4], [6, 8]$, reducing its size from 7 to 4). *Delta encoding* exploits overlap between the provenance of tuples by encoding the provenance of a tuple as a delta over the provenance of a previous tuple. For example, consider the provenance of two consecutive tuples: $\{1, 3, 5, 7, 9\}$ and $\{3, 5, 7, 9, 11\}$. The second provenance can be encoded as a delta "Skip the first element of the previous provenance and append $\{11\}$". This type of encoding is very effective for sliding windows. Generic *dictionary compression* (we use LZ77) is used if the size of a TID set exceeds a threshold. Our prototype combines the presented compression techniques using a set of heuristic rules. Generally speaking, we first choose whether to use intervals or a TID set, then apply delta-encoding ontop if the overlap between consecutive TID sets is high, and finally apply dictionary compression if the result size still exceeds a threshold.

Replay-Lazy. The *replay-lazy* method introduced in Section 2.2 computes provenance by replaying parts of the input through a provenance generating network, providing several benefits: (1) the cost of provenance generation is only paid if provenance is actually needed, (2) the overhead on regular query processing is minimal, enabling

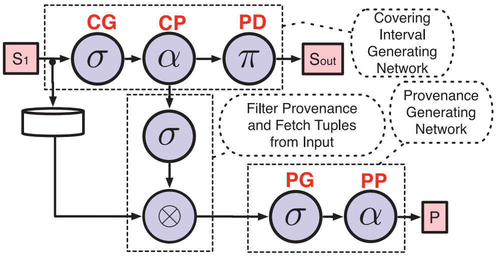

CG
CP
PD
S_1
σ
α
π
S_{out}
Covering Interval Generating Network
Provenance Generating Network
Filter Provenance and Fetch Tuples from Input
σ
$\otimes$
PG
PP
σ
α
P

Fig. 10. Example replay-lazy network.

provenance for time-critical applications, and (3) provenance computation is mostly decoupled from query execution. *Replay-Lazy* is only applicable to query networks consisting of deterministic and monotone operators. In order to avoid having to replay a complete prefix of a stream, we compute which parts have to be replayed during query execution. Specifically, these are all tuples from the interval spanned by the smallest and largest TID in the provenance of an output tuple (we refer to this set of tuples as the *covering interval* of a TID set). The network is instrumented in the same way as for *reduced-eager* (see Figure 10), except that we annotate each tuple with its covering interval (*CG* and *CP* are *PG* and *PP* operators that annotate with covering intervals). These intervals require constant space, thus reducing the overhead of propagation significantly. Furthermore, generating and maintaining them is rather cheap. In order to access the tuples belonging to a covering interval, we introduce a new join operator: A *c-join* $\otimes(S, CP)$ between a stream S and a connection point CP processes each tuple t from S by fetching all tuples included in the covering interval of t from the connection point and emitting these tuples. These tuples are then fed into a copy of the query network that is instrumented for provenance generation.

Lazy Retrieval. Both *reduced-eager* and *replay-lazy* reduce the runtime overhead of provenance generation at the cost of additional computation for tuple reconstruction during provenance retrieval. If interactive retrieval is used, we only need to reconstruct provenance for tuples when explicitly requested. If the reconstructed provenance is used as an input to a query over provenance, then we have the opportunity to avoid the cost of reconstruction through a p-join operator by determining upfront which parts of the provenance are not needed in the retrieval part of the query.

6. EXPERIMENTS

The goal of our experimental evaluation is to investigate the overhead of provenance management with Ariadne, compare with competing approaches (*rewrite*), investigate the impact of varying the provenance generation and retrieval methods (eager vs. lazy), and study the effectiveness of the optimizations proposed in Section 5.

6.1. Provenance Propagation by Query Rewrite

To be able to compare against *query rewrite*, we have implemented this technique following the approach pioneered in the Perm project [Glavic and Alonso 2009]. More specifically, let S_1 and S_2 be input streams of operators and S_{out} denote an operator output stream. Given a PAS $P(O, I)$ for a query network q, we have to transform q into a network that computes the PAS $P(O, I)$ using solely the standard operators of the DSMS. The rewrite process is straightforward for most operators. Figure 11 shows rewrite rules for selection and aggregation and an example query and its rewritten counterpart (bottom). Aggregations are rewritten by joining their outputs with PAS for their inputs. Note that this rewrite is only possible for window operators where we can

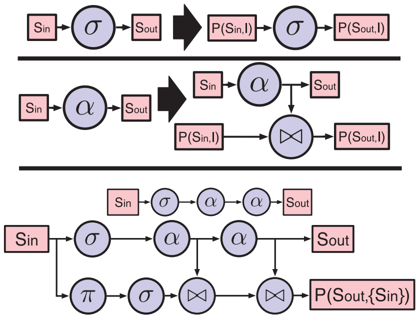


Fig. 11. Query rewrite rules and example.

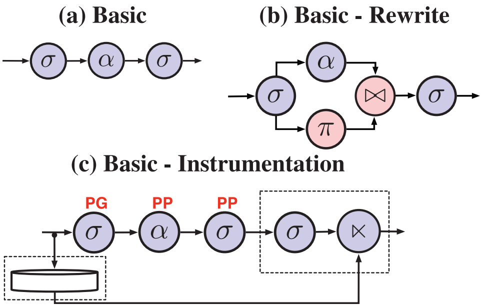


Fig. 12. Experiment queries.

express a join condition which guarantees that each tuple from a certain window only joins with the aggregated output produced for that particular window. For instance, for a value-based window function $val(c, s, a)$, we add two additional aggregation functions to compute the minimum and maximum values of attribute a for the window. These values are used in the join condition as follows: $min(a) \geq a \wedge a \leq max(a)$.

6.2. Setup

Figure 12 shows the query network (called *Basic*) used in most experiments in its original (a), rewritten (b), and instrumented (c) versions. The replay-lazy version closely resembles Figure 10. This query covers the most critical operator for provenance management (aggregation) and is simple enough to study individual cost drivers. In experiments that focus on the cost of provenance generation, we leave out parts of these networks that implement retrieval (the dashed boxes).

Setup and Methodology. Since the overhead of unused provenance code turned out to be negligible, we used Ariadne also for experiments without provenance generation. All experiments were run on a system with four Intel Xeon L5520 2.26Ghz quadcore CPUs, 24GB RAM, running Ubuntu Linux 10.04 64 bit. Client (load generator) and server are placed on the same machine. The input data consists of tuples with a small number of numeric columns (in total around 40 bytes), to make the overhead of provenance more visible. The values of these columns are uniformly distributed. All input data is generated beforehand. Each experiment was repeated ten times to minimize the impact of random effects. We show the standard deviation where possible in the graphs. Our study focuses on the time overhead introduced by adding provenance management to continuous queries, as this is the most discriminative factor between competing approaches. We are interested in two cost measures: (1) *computational cost*, which we determine by sending a large input batch of 100K tuples over the network at maximum load and measuring the *completion time*; (2) *tuple latency* determined by running the network with sufficient available computational capacity.

6.3. Fundamental Trade-Offs

In the first set of experiments, we study the computational overhead of managing provenance (split into generation and retrieval) using the *basic* query with maximum load. We show results for our *reduced-eager* and *replay-lazy* approaches without provenance compression (called *single* from now on), and compare them with the cost of the network with *no provenance* as well as *rewrite*.

End-to-End Cost. The first experiment (shown in Figure 13) compares the end-to-end cost when changing the amount of provenance that is being produced per result tuple.

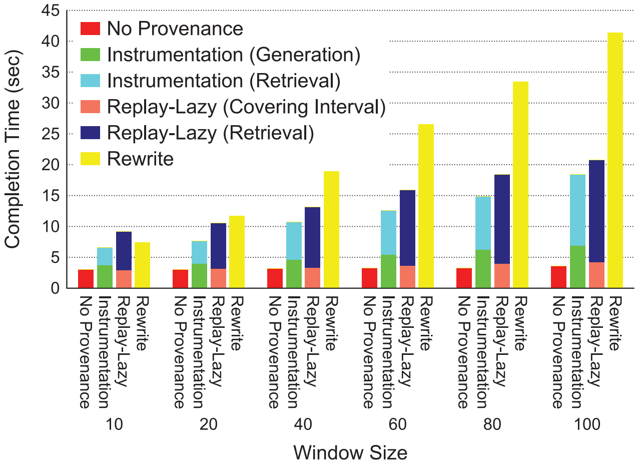


Fig. 13. End to end - vary provenance amount.

Method		Number of Aggregations			
		1	2	3	4
No Provenance		3.1	3.9	4.8	5.7
Instrumentation	Generation	3.9	7.4	14.7	48.6
	Retrieval	3.0	12.9	103.0	2047.0
Replay-Lazy	Covering Interval	3.1	4.4	5.2	6.3
	Retrieval	5.2	14.7	91.1	2224.0
Rewrite		7.2	625.0	crash	crash

Fig. 14. Aggregations: completion time (sec).

This is achieved by changing the *window size* (WS) of the aggregation operator from 10 to 100 tuples (while keeping a constant *slide* SL = 1 and selectivity 25% for the first selection in the network). Provenance is retrieved for all result tuples. The results demonstrate that the general overhead of provenance management is moderate for all methods: an order of magnitude more provenance tuples than data tuples (WS =10) roughly doubles the cost, two orders of magnitude (WS =100) lead to an increase by a factor 5 (*instrumentation*) to 12 (*rewrite*). Analyzing the individual methods, we see that the cost of *instrumentation* is strongly influenced by retrieval: around 40% at WS =10, and around 65% at WS =100. This cost is roughly linear to the amount of provenance produced. The overhead of provenance generation through instrumentation is between 20% (WS =10) and 113% (WS =100). Using *replay-lazy*, the overhead on the original query network (generation of *covering intervals*) is further reduced to 3% (WS =10) and 16% (WS =100), respectively. The price to pay for this reduction is the additional cost of provenance *replay*, where the cost is similar to the combination of *instrumentation* generation and retrieval, as this method is now applied on all covering intervals to compute the actual provenance. Even for this benign workload, *rewrite* shows much worse scaling than *instrumentation* with full *retrieval*: while roughly on par for WS =10, it requires twice as much time for WS =100.

Nested Aggregations. We now increase the number of aggregations to exponentially increase the amount of provenance per result tuple. We start off with the *Basic* network (WS =10 and SL = 1) and gradually add more aggregation operators. The increase of cost for *instrumentation* is (slightly) sublinear in the provenance size. Most of the overhead can be attributed to *retrieval*, while provenance generation increases moderately due to the TID set representation. The overhead of generating *covering intervals* for *replay-lazy* is around 10% over the baseline (*no provenance*), while the effort spent for

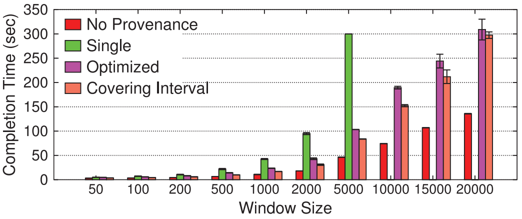


Fig. 15. Window size (SL=1, S=25%).

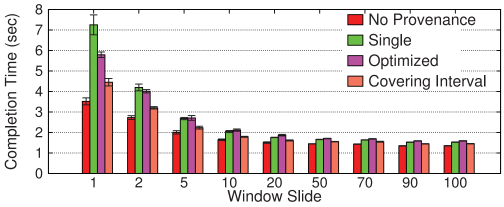


Fig. 16. Window slide (WS =100, S=25%).

replaying shows the same behavior as the total cost of *instrumentation*. Finally, the results (Figure 14) indicate that *rewrite* does not scale in the number of aggregations, as demonstrated by an increase in overhead in comparison to *instrumentation* from 20% (one aggregation) to 3,300% (two aggregations). At three aggregations, the execution exhausts the available memory.

6.4. Cost of Provenance Generation

We now focus on window-based aggregation, since it is not used in traditional, non-streaming workloads and produces large amounts of provenance. In addition to the methods previously shown, we enable the adaptive compression technique (denoted as *optimized*). Furthermore, we will no longer consider the *rewrite* method (its drawbacks are obvious) and *retrieval* cost (as it is linear with respect to the provenance size). We study the impact of *window size* (provenance amount per result), *window overlap* (commonality in provenance), and pre-selection *selectivity* (TID contiguity). These experiments use the *basic* network.

Window Size. Figure 15 shows *completion time* for varying WS from 50 to 20000. A front selection selectivity of 25% ensures that there are very few contiguous TID sequences, limiting the potential of *interval compression*. Furthermore, the overlap introduced by the small slide and large windows is detrimental for covering intervals, since a significant amount of interval merging needs to be performed. Completion time is higher for larger window sizes, but compression mitigates this effect: the completion time overhead for *single* grows significantly, starting from 75 % at WS 100 and reaching around 550% at WS 5,000. After this point, the overhead became so high that the system did not stay stable. Despite the challenging workload, adaptive compression reduces the overhead to 50% and 130%, respectively. Covering intervals further reduce the overhead, albeit with diminishing returns at larger window sizes due to ever-increasing number of intervals to merge. The amount of memory needed to maintain provenance information in the window operator follows a similar pattern: *single* uses a naive approach that keeps provenance for every output window separately, utilizing 35 KB at WS 100 and 163 MB at WS 5,000. Adaptive compression uses an improved approach that shares provenance information whenever possible, reducing the cost to 114 KB at WS 5,000 and 384 KB at WS 20,000. These values come very close to the space needed for the TIDs of all open windows, showing that provenance management is not a bottleneck when scaling the workload. Covering intervals do not need any additional space beyond their extended headers, since intervals are merged as soon as possible. Likewise, the amount of data transferred between operators increases sharply when using *single*, from 39 MB (WS 100) to 440 MB (WS 5,000) against a baseline of 15 MB. Adaptive compression, on the other hand, sees a moderate increase: 22 MB at WS 100 and 40 MB at WS 20,000. For covering intervals, there is only limited overhead, less than an additional MB regardless of the window size.

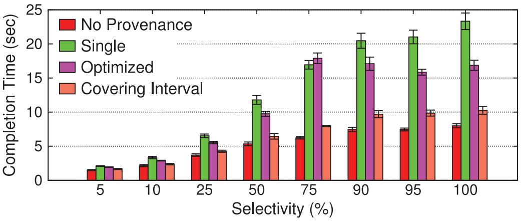


Fig. 17. TID contiguity (WS =100, SL =1).

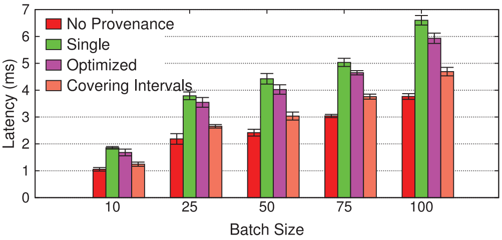


Fig. 18. Latency.

We then altered our workload in three ways to study the consequences of extremely large windows: (i) instead of sending a batch of 100K tuples and measuring the completion time (which would limit the window size), we sent a large number of consecutive batches of size 10K tuples, (ii) we set the slide size to be one tenth of the window size, and (iii) we increased the selectivity of the front filter to 50%, to allow formation of a reliable number of very large windows. We also skipped *single*, since it is clearly not competitive for very large windows. With this setup, we could scale up our measurements to windows containing 500K tuples while observing only moderate overhead: The memory needed to maintain provenance state in the window operator was less than 9 MB at WS 500K for adaptive compression. The computational overhead leveled off at large window sizes, staying at around 100% for adaptive compression and 60% for covering intervals.

Window Slide. Reducing the overlap between windows (increasing SL from 1 to 100, WS =100) decreases the overall cost, since far fewer result tuples need to be generated (Figure 16). The logarithmic decline can be explained by the fact that the low load makes the impact of provenance generation negligible for slides bigger than 10. Large slide values result in small overlap between open windows. Hence, they demonstrate the worst-case scenario for the adaptive compression, because maintaining the complex data structures of these techniques does not pay off anymore. Yet, compression performs only slightly worse than the single approach.

TID. Besides the specific window parameters such as WS or SL, the performance for window-based aggregates is also influenced by upstream operators affecting the distribution of TID values. We investigate these factors by varying the selectivity of the first selection operator in the *basic* network between 5% and 100% (Figure 17). Without TID compression, the *completion time* is linear to selectivity, because the number of generated output tuples also grows linearly and generation is not affected by TID distribution. Interval compression used by *optimized* becomes more efficient when selectivity is increased, as more and more contiguous TID ranges are created. We therefore see no further increase in cost for selectivities over 75%.

6.5. Influence of Network Load on Latency

In reality, a query network is rarely run at maximum load. Thus, performance metrics such as *latency* play an important role. We run the *basic* network (*generation* and *retrieval*, WS =100, SL =1, S =25%) and vary the load by changing the size of the batches being sent from the client between 10 and 100 tuples while keeping the frequency of sending batches fixed. Smaller batches are avoided, because they result in very unpredictable performance. For sizes larger than 100 the slowest method (*single*) would not be able to always process input instantly. As shown in Figure 18, provenance generation does indeed increase the latency, but this increase is very moderate and stays at the same ratio over an increasing load. *Single* results in about 75% additional latency,

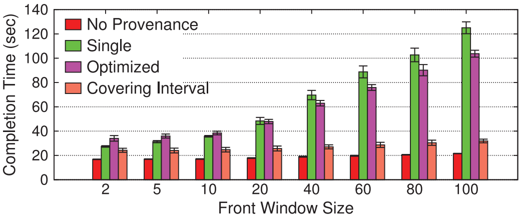


Fig. 19. Complex network.

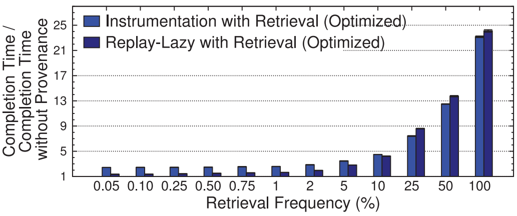


Fig. 20. Retrieval frequency.

optimized reduces this overhead to around 60%, while *covering intervals* is the cheapest with around 20% overhead.

6.6. Complex Query Networks

We now investigate whether our understanding of the cost of individual operators translates to real-life query networks using the complete running example introduced in Figure 1. We use this network (called Complex) to study how our approach translates to a more complex query network with multiple paths and a broad selection of operators. This query does not lend itself easily to straightforward optimizations (limited TID contiguity) and stresses intermediate operators with large amounts of provenance. We vary the amount of provenance created by the network by varying the window size for the aggregations applied before the union operator ("front" windows). As Figure 19 shows, the overhead of *reduced-eager* instrumentation without compression (*single*) is higher than in previous experiments. The *optimized* method (adaptive compression) shows its benefits: while more expensive for very small WS values (100% overhead at WS =2), it becomes more effective for larger window sizes. *Covering intervals* is again very effective with 40% overhead independent of the increase in provenance. Memory measurements support these observations, since the additional provenance does not increase the cost significantly when using compression or covering intervals.

6.7. Varying Retrieval Frequency

Many real-world scenarios do not need provenance for the entire result stream. We therefore study the effect of retrieval frequency (as a simple form of partial provenance retrieval) on the trade-off between *reduced-eager* and *replay-lazy*. Using the *nested aggregation* network with four aggregations (WS =10 and SL=3) and 2 million input tuples, we vary the rate of retrieval from 0.05% to 100% (by inserting an additional selection before reconstruction). The results are shown in Figure 20 (overhead w.r.t. completion time of *no provenance*). For low retrieval frequencies (less than 1%), the cost of retrieval is insignificant. *Reduced-eager* generates provenance for all outputs, and thus, the overall cost is dominated by provenance generation. Computing covering intervals for *replay-lazy* results in a relative overhead of about 13% over the completion time for *no provenance* (which is constant in the retrieval frequency). *Replay-lazy* has to compute only few replay requests at low retrieval rates, but in turn pays a higher overhead for higher retrieval rates. *Replay-lazy* is the better choice for the given workload if the retrieval frequency is 10% or less.

Summary. Our experiments demonstrate the feasibility of fine-grained end-to-end provenance in DSMS. *Operator instrumentation* clearly outperforms *rewrite*. Furthermore, *reduced-eager* allows us to separate generation and retrieval. *Replay-lazy* based on covering intervals reduces the overhead on the "normal" query network and enables us to scaleout. The optimizations for provenance compression are effective in both small-scale, synthetic, as well as large-scale, real-life workloads.

7. RELATED WORK

Our work is related to provenance on workflow systems, databases, and stream processing systems.

Workflow Systems. Workflow provenance approaches that handle tasks such as black boxes are not suitable for managing stream provenance [Davidson et al. 2007]. More recently, finer-grained workflow provenance models have been proposed (e.g., allowing explicit declarations of data dependencies [Anand et al. 2009] or applying database provenance models to Pig Latin workflows [Amsterdamer et al. 2011a]). These systems only support nonstream processing models and require explicit declarations. Ariadne's compression techniques resemble efficient provenance storage and retrieval techniques in workflow systems (e.g., subsequence compression technique [Anand et al. 2009] or node factorization [Chapman et al. 2008]). However, due to the transient and incremental nature of streaming settings, we use compression mainly for optimizing provenance generation.

Database Systems. There are several different notions of database provenance [Cheney et al. 2009] supported by different systems (e.g., Trio [Benjelloun et al. 2006], DBNotes [Bhagwat et al. 2004], Perm [Glavic and Alonso 2009]). Like *lineage* in relational databases, Ariadne represents the provenance of an output tuple as a set of input tuples that contributed to its generation. In principle, our operator instrumentation techniques can be extended to support more informative provenance models similar to database provenance models, such as provenance polynomials [Green et al. 2007] and graph-based models [Acar et al. 2010]. However, it is unclear if their benefits in terms of equivalences will hold for streaming operators. Given the fundamental differences in the data and query models for streams, investigating whether these existing provenance models or minimization techniques [Amsterdamer et al. 2011b] can be adapted to stream provenance is promising future work.

Stream Processing Systems. There is only a handful of related work on managing stream provenance. Vijayakumar and Plale have proposed coarse-grained provenance collection techniques for low-overhead scientific stream processing [2006]. Wang et al. have proposed a rule-based provenance model for sensor streams, where the rules have to be manually defined for each operation [2007]. More recently, Huq et al. have proposed achieving fine-grained stream provenance by augmenting coarse-grained provenance with time-stamp-based data versioning [2011]. In his work, provenance generation is based on inversion, as opposed to Ariadne's propagation-based approach, hence it is more restricted. A common use case for stream provenance data is query debugging. Microsoft CEP server [Ali et al. 2009] exposes coarse-grained state of the system through snapshots and streams of manageability events. The visual debugger proposed in De Pauw et al. [2010] supports fine-grained provenance computation based on identifier annotation and operator instrumentation, where per-operator provenance is stored and multi-operator provenance is generated from it on an on-demand basis.

8. CONCLUSIONS

We present *Ariadne*, a system addressing the challenges of computing fine-grained provenance for data stream processing, which provides an important building block for provenance on event detection. *Reduced-eager operator instrumentation* provides a novel method for computing provenance for an infinite stream of data that adds only a moderate amount of latency and computational cost and correctly handles nondeterministic operators. *Replay-lazy* and *lazy-retrieval* provide additional optimizations for decoupling provenance computation from stream processing. The effectiveness of our techniques is successfully validated in the experimental evaluation over various performance parameters and workloads. Interesting avenues for future work include (i) studying provenance retrieval patterns to exploit additional knowledge for storage decisions and in optimizing computations, (ii) investigating distributed architectures and integration of our system with scalable distributed storage, and (iii) extending our provenance semantics to model the inherent order of streams.

REFERENCES

Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The design of the Borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*. 277–289.

Umut Acar, Peter Buneman, James Cheney, Jan van den Bussche, Natalia Kwasnikowska, and Stijn Vansummeren. 2010. A graph model of data and workflow provenance. In *Proceedings of the Workshop on the Theory and Practice of Provenance (TaPP)*.

Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 147–160.

Mohamed H. Ali, Ciprian Gerea, Balan Sethu Raman, Beysim Sezgin, Tiho Tarnavski, Tomer Verona, Ping Wang, Peter Zabback, A. Ananthanarayan, A. Kirilov, Ming Lu, Alex Raizman, Ramkumar Krishnan, Raman Schindlaur, Torsten Grabs, Sharm Bjeleticl Badrish Chandramonli, Jonathan Goldstein, Sudin Bhat, Ying Li, Vincenzo Di Nicola, Xinafara Wanq, David Maier, Ivo Santos, Olivier Nano, and Stephan Grell. 2009. Microsoft CEP server and online behavioral targeting. In *Proceeding of the International Conference on Very Large Data Bases (VLDB)*. 1558–1561.

Foteini Alvanaki, Sebastian Michel, Krithi Ramamanitham, and Gerald Weikum. 2012. See What's enBlogue: Real-time emergent topic identification in social media. In *Proceeding of the International Conference on Extending Database Technology (EDBT)*. 336–347.

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

Yael Amsterdamer, Daniel Deutch, Tova Milo, and Val Tannen. 2011b. On provenance minimization. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 1–36.

Manish Kumar Anand, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. 2009. Efficient provenance storage over nested data collections. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 958–969.

Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. 2006. ULDBs: Databases with uncertainty and lineage. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 953–964.

Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. 2004. An annotation management system for relational databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 900–911.

Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. 2010. SECRET: A model for analysis of the execution semantics of stream processing systems. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 232–243.

Adriane P. Chapman, Hosagrahar V. Jagadish, and Prakash Ramanan. 2008. Efficient provenance storage. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 993–1006.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in databases: Why, how, and where. *Found. Trends Datab.* 1, 4 (2009), 379–474.

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

Susan B. Davidson, Sarah Cohen Boulakia, Anat Eyal, Bertram Ludäscher, Timothy M. McPhillips, Shawn Bowers, Manish Kumar Anand, and Juliana Freire. 2007. Provenance in scientific workflow systems. *IEEE Data Eng. Bulle.* 32, 4 (2007), 44–50.

Wim De Pauw, Mihai Leţia, Buğra Gedik, Henrique Andrade, Andy Frenkiel, Michael Pfeifer, and Daby Sow. 2010. Visual debugging for stream processing applications. In *Proceedings of the International Conference on Runtime Verification (RV)*. 18–35.

Peter M. Fischer, Aayush Garg, and Kyumars Sheykh Esmaili. 2010. Extending XQuery with a pattern matching facility. In *Proceeding of the International XML Database Symposium (XSym)*. 48–57.

Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 174–185.

Boris Glavic, Kyumars Sheykh Esmaili, Peter M. Fischer, and Nesime Tatbul. 2011. The case for fine-grained stream provenance. In *Proceedings of the BTW Workshop on Data Streams and Event Processing (DSEP)*. 58–61.

Boris Glavic, Kyumars Sheykh Esmaili, Peter M. Fischer, and Nesime Tatbul. 2012. Ariadne: Managing fine-grained provenance on data streams. Technical Report 771. ETH Zurich.

Boris Glavic, Kyumars Sheykh Esmaili, Peter M. Fischer, and Nesime Tatbul. 2013. Ariadne: Managing fine-grained provenance on data streams. In *Proceedings of the International Conference on Distributed Event-Based Systems (DEBS)*. 39–50.

Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*. 31–40.

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. In *Proceedings of the IEEE International Conference on E-Science (e-Science)*. 202–209.

Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. 2008. The ORCHESTRA collaborative data sharing system. *SIGMOD Record* 37, 2 (2008), 26–32.

Alberto Lerner and Dennis Shasha. 2003. The virtues and challenges of ad hoc + streams querying in finance. *IEEE Data Eng. Bullet.* 26, 1 (2003), 49–56.

Yuan Mei and Samuel Madden. 2009. ZStream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 193–206.

Frederick Reiss and Joseph M. Hellerstein. 2005. Data triage: An adaptive architecture for load shedding in telegraphcq. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 155–156.

Esther Ryvkina, Anurag S. Maskey, Mitch Cherniack, and Stan Zdonik. 2006. Revision processing in a stream processing engine: A high-level design. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 141–141.

Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. 2010. Earthquake shakes twitter users: Real-time event detection by social sensors. In *Proceedings of the International World Wide Web Conferences (WWW)*. 851–860.

Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 309–320.

Nithya N. Vijayakumar and Beth Plale. 2006. Towards low overhead provenance tracking in near real-time stream filtering. In *Proceedings of the International Provenance and Annotation Workshop (IPAW)*. 46–54.

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 *Proceeding of the ACM HealthNet Workshop*. 95–100.

Allison Woodruff and Michael Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 91–102.

Received November 2013; revised March 2014; accepted April 2014