Ariadne: Managing Fine-Grained Provenance on Data Streams

Boris Glavic
Illinois Institute of Technology
Chicago, IL
bglavic@iit.edu

Kyumars Sheykh Esmaili
Nanyang Technological University
Singapore
kyumarss@ntu.edu.sg

Peter M. Fischer
University of Freiburg
Germany
peter.fischer@cs.uni-freiburg.de

Nesime Tatbul
Intel Labs and MIT
Cambridge, MA
tatbul@csail.mit.edu

ABSTRACT

Managing fine-grained provenance is a critical requirement for data stream management systems (DSMS), not only to address complex applications that require diagnostic capabilities and assurance, but also for providing advanced functionality such as revision processing or query debugging. This paper introduces a novel approach that uses operator instrumentation, i.e., 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 run-time overhead and avoid unnecessary provenance retrieval. This includes computing a concise superset of the provenance to allow lazily replaying a query network 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*

Keywords

Data Streams, Provenance, Annotation, Experiments

1. INTRODUCTION

Stream processing has recently been gaining traction in a new class of applications that require diagnostic capabilities, assurance, and human observation [3, 14]. In these applications, there is a common need to provide "fine-grained provenance" information (i.e., at the same level as in database provenance [10]), to trace an output event back to the input events contributing to its existence.

Example. In monitoring and control of manufacturing systems, sensors are attached along a supply chain. Sensor readings are processed by a DSMS in order to detect critical situations such as machine overheating. 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. Figure 1 shows a simplified example of a continuous query that detects overheating. Two sensors feed timestamped temperature readings to the query. Each sensor stream is filtered to remove massive outliers (i.e., temperature t above 350°C). The stream is aggregated by averaging the temperature over a sliding window of 3 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 2 time units. In this example, the user would want to understand which sensor readings caused an "overheating" alarm event, i.e., determine the tuples that belong to the *fine-grained* provenance of this event.

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 [11] and relational databases [10]. 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 can 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 [22, 24]) and certain operator definitions such as windowing on system time result in outputs that are not determined solely by the inputs. Conventional provenance management techniques (e.g., query rewrite [13]) and naive solutions (e.g., tak-

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

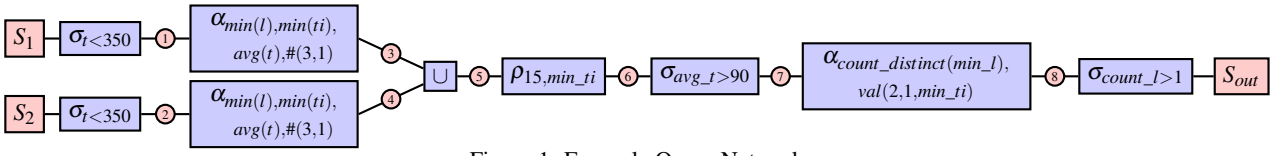


Figure 1: Example Query Network

ing advantage of cheap, fast storage by dumping all inputs and inferring provenance from the complete stream data) are not sufficient to address all of the challenges outlined above.

Contributions and Outline. In this paper, 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 [10]. 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*). This approach can also be used to compute the provenance of a part of the query network by only instrumenting a subset of the operators. Previous annotation propagation approaches for fine-grained stream provenance [12] are restricted to one-step provenance, i.e., annotating output tuples from an operator with their provenance from the operator's input. Our approach is more general and flexible; provenance can also be propagated through several operators or even a complete query network. By lifting this restriction we are able to overcome many of the shortcomings of these approaches including large storage overhead (tracking provenance through a path in the query network requires storage of all streams on the path) and expensive retrieval (queries over provenance require recursive tracing using the single-step provenance).

We represent provenance as sets of tuple identifiers during provenance generation. Querying provenance is supported by reconstructing complete input tuples from the identifier sets using a new operator called *p-join*. This is achieved by temporarily storing input stream tuples for the reconstruction.[1] A number of optimizations enable us to decouple provenance management from query processing: The *Replay-Lazy* optimization reduces the run-time overhead of provenance computation by propagating a concise superset of the provenance and lazily replaying a query network to reconstruct its provenance. The *Lazy-Retrieval* method avoids reconstructing provenance for retrieval if parts of the provenance will not be needed by the query. Furthermore, we devise a number of compression schemes to reduce the computation cost. We have implemented our approach in Ariadne, a provenance-aware DSMS that is based on the Borealis prototype [1]. More specifically, this paper 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 generating provenance for networks and subnetworks without the need to materialize data at each operator.
- We propose a number of optimization techniques that allow decoupling provenance computation from query processing through the application of lazy generation and retrieval techniques and improve computation performance by compression.

[1]This is in contrast to one-step approaches that also require the storage of intermediate streams.

Method	Applicable to	Runtime Overhead	Retrieval Overhead
Inversion	Invertible	None	High
Query Rewrite	Deterministic	High	-
Operator Instrumentation	All	Low	-

Figure 2: Comparison of Provenance Generation Alternatives

- We present Ariadne, the first DSMS prototype providing support for fine-grained multi-step 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.

The rest of this paper 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 model underlying our approach in Section 3. Building upon this model, we present its implementation in the Ariadne prototype in Section 4. We cover optimizations of our basic approach 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 special provenance-aware operators. 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.

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 tradeoffs. *Inversion* (e.g., Woodruff et al. [27]) 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 constructed from an output tuple. Inversion has very limited applicability to DSMSs, because no real inverse exists for most non-trivial operators. *Query Rewrite*, established in relational systems such as Perm [13], DBNotes [8], or Orchestra [19], generates provenance by rewriting a query network Q into a network that generates the provenance of Q in addition to the original network outputs. This usually requires changes to the structure of the query network. For example, as explained in [13], a provenance-generating copy of a subnetwork has to be added to and joined with the original subnetwork to support aggregates. This leads to significant additional run-time overhead and incorrect provenance for non-deterministic operators.

In summary, we believe that Operator Instrumentation is the best approach for generating provenance in DSMSs, because it is applicable to a large class of queries while maintaining low overhead in

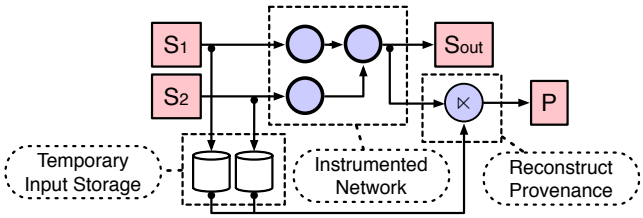


Figure 3: Reduced-Eager Operator Instrumentation

terms of provenance computation and retrieval. Our experimental results in Section 6 verify our hypothesis.

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. That is, the structure of the original query network is kept as is (operators are simply replaced with their instrumented counterparts). Thus, most issues caused by non-determinism are dealt with in a rather natural way, since the execution of the original query network is traced[2]. Provenance can be traced for a single operator (as supported by previous approaches [12]) or for a complete subnetwork. Furthermore, we can trace provenance for a subnetwork by instrumenting only operators in that subnetwork. 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

Reduced-Eager: Figure 3 shows an example 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 the provenance annotations as described above, the set of identifiers for the tuples in the provenance of an output. Provenance is reconstructed for retrieval from the 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. 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).

[2]The overhead introduced by provenance generation may affect temporal conditions (e.g., windows based on system time), but such systems are sensitive to system events in general.

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

Figure 4: Trade-offs for Eager vs. Lazy

Replay-Lazy: Instead of generating provenance *eagerly* while the query network is running, we would like to be able to generate provenance *lazily* in order to decouple provenance generation from the execution of the query network. Since DSMSs are expected to deal with high rates and low latency requirements, eager provenance computation may incur significant runtime overhead to the critical data processing path. Decoupling most of the provenance computation from query processing enables us to reduce the runtime overhead on the query network and outsource provenance generation to a separate machine and thus improve performance for both normal query processing and provenance computation. For deterministic networks, we can realize lazy generation by replaying relevant inputs through a instrumented copy of the network. We call this approach *Replay-Lazy*. *Replay-Lazy* has to propagate minimal bookkeeping information during query execution to be able to determine which inputs are relevant and, thus, reduce the amount of data that is stored and replayed. We record for each output tuple the parts of the input which are needed for the replay to be executed correctly (by annotating the tuple), which turns out to be a concise superset (constant size) of the actual provenance. *Replay-Lazy* reduces the runtime overhead by just computing this minimal type of provenance, but incurs a higher retrieval cost due to the replay and is only applicable to deterministic networks.

3. PROVENANCE PROPAGATION BY OPERATOR INSTRUMENTATION

Based on the stream data and query model of Borealis [1], we now informally introduce our stream provenance model and discuss how to instrument queries to annotate their outputs with provenance information. A formal treatment can be found in [15].

3.1 Data and Query Model

We model a stream $S = \ll t_1, \ldots \gg$ as a possibly infinite sequence of tuples. A tuple $t = (TID; a_1, \ldots)$ is an ordered list of attribute values (here a_i denotes a value) plus a tuple-identifier (TID) that uniquely identifies the tuple within stream scope denoted as *stream-id:tuple-id*. 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 [1].

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.

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.

Aggregation: An aggregation operator $\alpha_{agg,\omega}(S)$ groups its input S into windows using the window function ω and computes the aggregation functions (agg) over each window generated by ω. For example, the count-based window function $\#(c,s)$ groups a consecutive input tuple sequence (length c) into a window and slides by

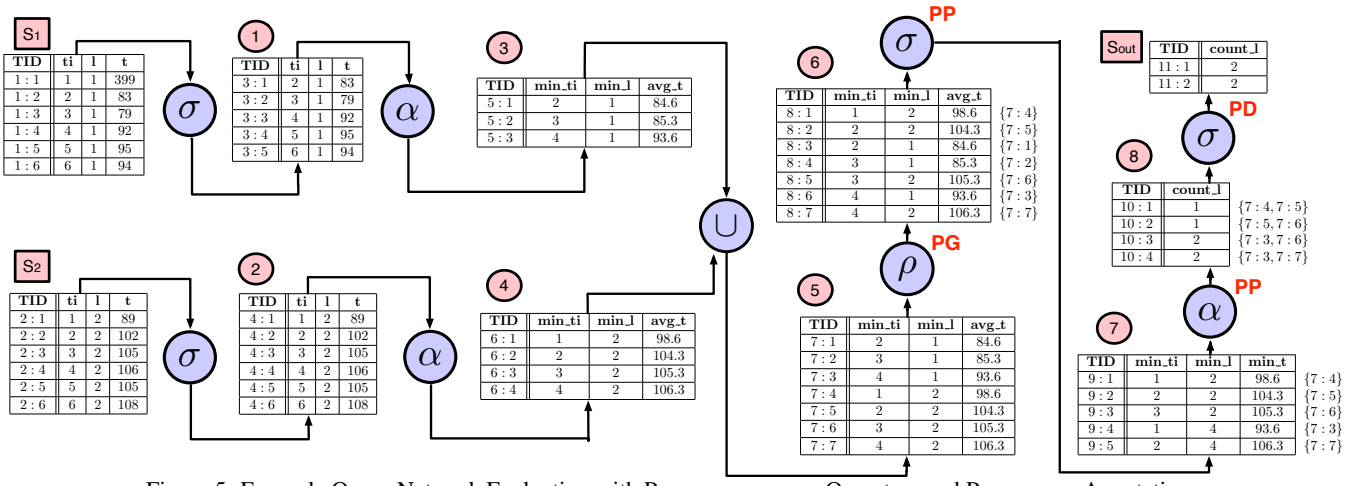


Figure 5: Example Query Network Evaluation with Provenance-aware Operators and Provenance Annotations

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.

Join: A join operator $\bowtie_{c,j\omega}(S_1,S_2)$ joins two input streams S_1 and S_2 by applying the join window function $j\omega$ to S_1 and S_2. A join window function models the buffering behavior of stream joins. All tuples from both input streams that are in the join's buffer at a point in time are grouped into the same join window. For each join window w, the join operator outputs all pairs of tuples from S_1 and S_2 that belong to w and fulfill the join condition c.

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.

B-Sort: A b-sort operator $\rho_{s,a}(S)$ with slack s and an order-on attribute a applies bounded-pass sort with buffer size $s+1$ on its input, and produces an output that is approximately sorted on a.

EXAMPLE 1. *Figure 5 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). Temperature outliers are filtered out. The results 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 5, 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 O 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 O form a proper query network. For instance, assume that t is a tuple from stream 5 in the network shown in Figure 5. $P(q,t,\{1,2\})$ 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. Formally, our work is based on a declarative definition of provenance, which is used to determine the provenance behavior for each of the operators. Intuitively, the provenance definition for all operators 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 *BSort*, since only a single tuple is contributing to t. For example, tuple 9:1 in the network shown in Figure 5 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.

We use the concept of provenance sets to 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. *For instance, consider the PAS $P(6,\{5\})$ for the output of the b-sort operator according to its input shown in Figure 5 (provenance sets are shown to the right of the tuples). Each output tuple t of the b-sort is annotated with a singleton set containing the corresponding tuple from the input of the b-sort, e.g., tuple 8:1 is derived from tuple 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 tuple is computed using information from a window containing two input tuples with one tuple overlap between the individual provenance sets. For example, tuple 10:2*

Algorithm 1 InstrumentNetwork Algorithm

```
1:  procedure INSTRUMENTNETWORK(q, O, I)
2:      mixed ← ∅
3:      for all o ∈ q do                          ▷ Find operators with mixed usage
4:          if ∃S, S' ∈ input(o) : S ∈ I ∧ S' ∉ I then
5:              mixed ← mixed ∪ input(o)
6:      for all S ∈ (mixed ∩ I) do                ▷ Add projection wrappers
7:          S ← Π_schema(S)(S)
8:      for all o ∈ q do                          ▷ Replace operators
9:          if ∃S ∈ I : HASPATH(S, o) ∧ HASPATH(o, O) then
10:             if ∃S' ∈ input(o) : S' ∈ I then
11:                 o ← PG(o)
12:             else
13:                 o ← PP(o)
14:     for all o ∈ q do                          ▷ Drop annotations
15:         if O ∈ input(o) then
16:             o ← PD(o)
```

is derived from a window containing tuples 7:5 and 7:6, and tuple 10:3 is derived from a window containing tuples 7:6 and 7:3.

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 annotating counterparts. Three types of *instrumented operators* are used in this approach:

Provenance Generator (PG): The provenance generator version $PG(o)$ of an operator o computes the PAS for all output streams of the operator according to its input streams. The purpose of a PG is to generate a PAS from input streams without annotations. 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): This version of operator generates the PASs for its outputs from PASs of its inputs. 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 of o will output $P(O,I)$, i.e., the output will be annotated with provenance sets of O according to I. Intuitively, a PP generates annotated output streams by modifying the annotations of its input streams according to the provenance behavior of the operator.

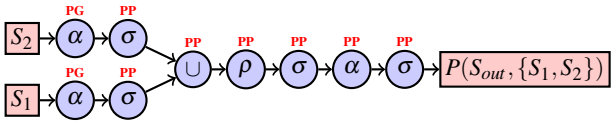
Provenance Dropper (PD): The provenance dropper version $PD(o)$ of an operator o removes annotations from the input before applying operator o. Provenance droppers are used to remove annotations from streams in networks with partial provenance generation.

For selection, the *PG* operator generates an annotated output stream where the provenance set of each output tuple t contains the tuple itself, and the *PP* operator outputs the input tuples with unmodified provenance sets (for tuples that fulfill the selection condition). Projection, union, and b-sort behave in the same way by creating singleton provenance sets (*PG*) or passing on provenance sets from the input (*PP*). The *PG* operator for aggregation annotates each output tuple t with a provenance set that consists of all identifiers for tuples in the input window that generated t, and the *PP* operator annotates each output tuple t with the union of the provenance sets of all tuples in the window that generated t. The *PG* version of join annotates each output t with a set consisting of the two tuples that were joined to produce tuple t. The *PP* version of this operator unions the provenance sets of the join partners.

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 *InstrumentNetwork* (Algorithm 1) to instrument a network q to compute a PAS $P(O,I)$. First we normalize the network to make sure 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 and not in I, because the annotation propagation behaviour of these operators is neither correctly modelled 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 in 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]. The algorithm then iterates through all operators in the query network and replaces each operator that reads solely from streams in I (case 1) 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.

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 its connected to. Thus, additional PAS are generated for free by our approach. We use $P(q)$ (called *provenance generating network* or PGN) 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 straightforward extension of Algorithm 1 to sets of output streams.

(a) Query Network with Full Propagation



(b) Annotation Propagation for Parts of the Network

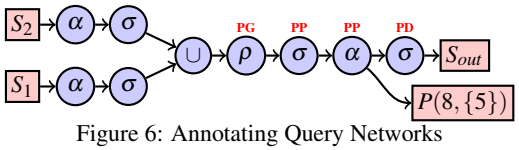


Figure 6: Annotating Query Networks

EXAMPLE 3. *Two provenance generating versions of the example network are shown in Figure 6 (the operator parameters are omitted to simplify the representation). Figure 6(a) shows $P(q)$, i.e., 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 left-most filter 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 6(b) generates the PAS $P(8, \{5\})$ (An example execution was shown in Figure 5). The output stream of the right-most 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*

[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.

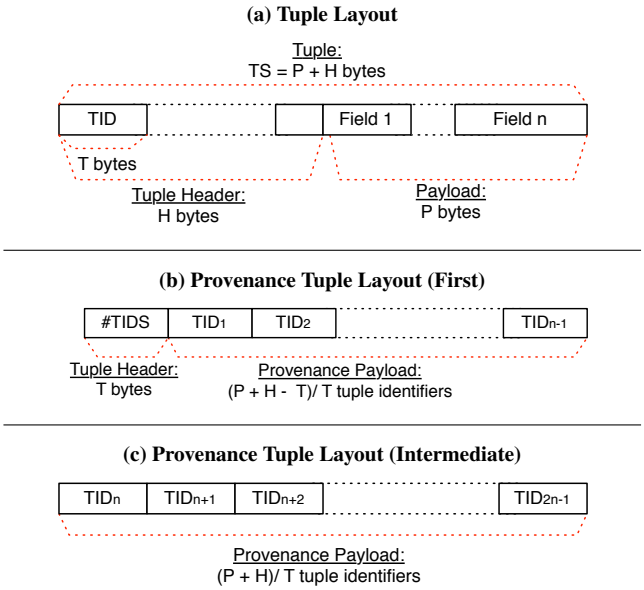


Figure 7: Physical Tuple Layout

annotations before applying the selection. This is necessary to produce the output stream S_{out} without annotations.

4. IMPLEMENTATION

In this secion, we present the implementation of the our approach in the prototype system *Ariadne*. Given the overall architecture (outlined in Section 2) and the provenance propagation model (Section 3), three aspects are now of particular 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

The physical representation of provenance annotations and mechanism for passing them between operators is a crucial design decision, because it strongly influences the run-time overhead of provenance generation and how we implement annotating operators. Since these annotations consist of TID sets, they can be of variable size. However, Borealis uses fixed-length tuples that consist of a fixed length header storing information such as TID and arrival time, and a payload which stores the binary data of the tuple's attributes values. The schema of a stream, which is not stored in the stream tuples, is used to interpret the payload of tuples.

We considered three alternatives to pass these variable-size TID-Sets between annotating operators: (1) Modify the queuing mechanism to deal with variable-length tuples, (2) propagate TID-Sets through channels other than Borealis queues, or (3) split large TID-Sets into fixed-length chunks which are then streamed over standard queues. We chose the third approach, as it is least intrusive and retains the performance benefits of fixed-size tuples.

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. Figures 7(b) and 7(c) show the physical layout of such tuples. The first tuple (Figure 7(b) 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. This header is used by downstream operators to determine how many provenance tuples have to be dequeued. Given that the size of a TID in Borealis is 8 bytes (actually *sizeof (signed 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. This is a trivial extension of our approach and left for future work.

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 three steps: (1) retrieving 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. When reading the information from the tuples in the input streams in *PG*-mode, the TIDs from the input tuples are extracted as provenance. In PP-mode we read the provenance sets attached by previous operations. For *PD* we simply discard the retrieved TID set. We factored out these commonalities into a so-called *provenance wrapper*. The provenance wrapper also implements additional common functionality such as buffering and merging of TID-Sets. Hooking into the *dequeue()* and *enqueue()* methods of Borealis, retrieval and serialization can be added trivially. Using the provenance wrapper, the operator-specific part of the provenance computation can thus be expressed with a small amount of code. For Selection, Projection, Union the provenance of a result tuple consists of the provenance of a single input tuple which can be directly determined. B-Sort and Join are slightly more complicated, requiring some lightweight bookkeeping to keep track of the contributing tuples. 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. Nonetheless, the amount of code needed for aggregation was just around 200 LOC, a fairly small change.

EXAMPLE 4. *Figure 8 shows the provenance computation for the annotating network from Figure 6(b). Recall that this network generates $P(q, 8, \{5\})$. Provenance headers are highlighted in brown and TIDs in a provenance tuple are highlighted in red. We use unrealistically small tuple sizes for presentation purposes. For instance, the aggregation operator uses the provenance wrapper to merge the TID-Sets from all tuples in a window and output them as the TID-Set for the result tuple produced for this window. For instance, the tuple 10:1 is generated from a window containing tuples 9:1 and 9:2. The merged TID-Set for these tuples ($\{7{:}4, 7{:}5\}$) is appended to the output tuple queue after tuple 10:1.*

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. We use a Borealis feature called Connection Point for input tuple storage and introduce the p-join operator for transforming TID-Sets into a queryable format.

Input Storage at PG operators: Connection points (CP), introduced by Ryvkina et al. [23] for revision processing in Borealis, provide temporary storage for tuples that pass though a queue. Besides other strategies, CPs support a time-out based strategy for removing old tuples from storage. We set this timeout according to

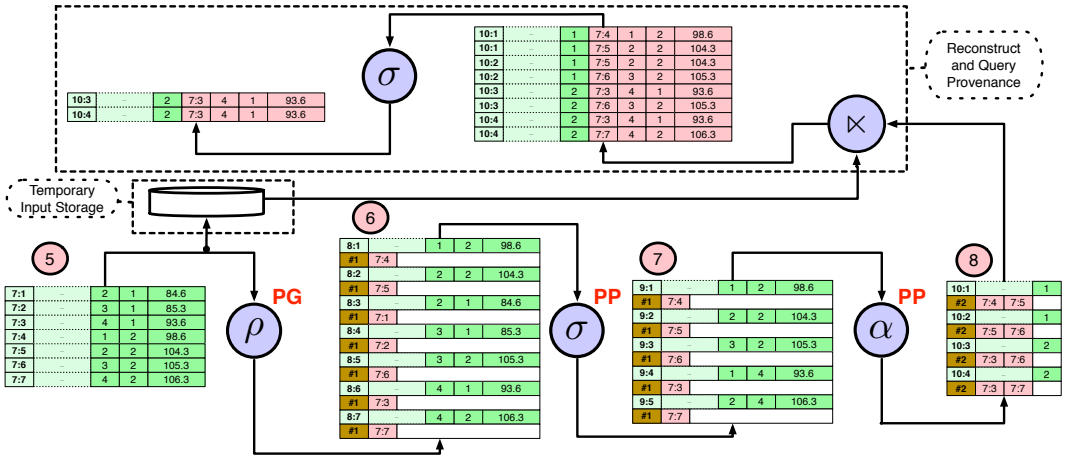


Figure 8: Provenance-enabled Query Network with Retrieval

the provenance retrieval pattern of the application, which typically span several seconds to minutes. Using the provenance itself for more directed expiration as well as utilizing write-optimized, possibly distributed storage technologies are interesting avenues for future work. 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, i.e., the streams that are inputs of provenance generators.

P-join: Similar to the approach in [13], we have chosen to represent provenance to the consumer using the non-extended 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. One tuple from the provenance is attached to each of these duplicates. Thus, we effectively list the provenance as a sequence of regular tuples which enables the user to express complex queries over the relationship between data and its provenance using existing Borealis operators. 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 and, thus, outputs tuples with tuples from their provenance. P-join uses a fast hash-based look-up from a CP (using the TID as the key) to determine the tuples to join with an input tuple instead of using a regular join with an input stream.

EXAMPLE 5. *The running example network with retrieval is shown in Figure 8. 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. The PAS generated by the aggregation operator is used as the input for a p-join with the single CP in the network. The stream produced by the p-join can be shown to a user or be used as input for further processing. Assume the user expected the system to output less alarms and suspects that the threshold for overheating should be raised. To test this assumption she can investigate which alarms (output tuples) have temperature readings (input tuples) in their provenance that are slightly above the threshold (e.g., below 100 degree). This query can be implemented by applying a filter ($avg\_t < 100 \wedge count\_l > 1$) on the output of the p-join as shown in Figure 8.*

5. OPTIMIZATIONS

Reduced-Eager is a solid solution for provenance computation. However, certain challenges in stream processing call for additional optimizations: (1) Typical DSMS workloads rely heavily on windowed aggregation. Such workloads produce large amounts of provenance per result. (2) Stream processing systems treat data as transient and discard data as soon as possible to keep up with high input data rates. Computing provenance on the fly to deal with the transient nature of streams increases run-time and latency. We address these challenges by developing compressed provenance representations (to reduce the overhead) as well as lazy provenance computation and retrieval techniques (to decouple query execution from provenance generation).

5.1 Provenance Compression

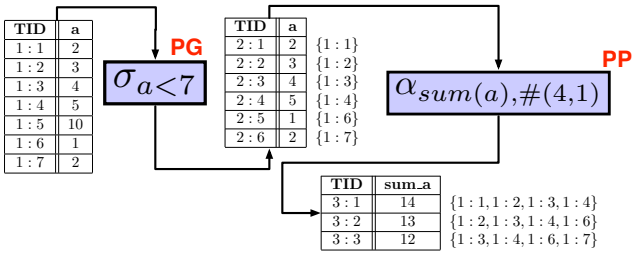
The methods we developed for TID-set compression range between generic data compression to methods which exploit data model and operator characteristics. We mainly target aggregation and focus on techniques that enable provenance computations at operators without having to decompress. Since each presented compression method has its sweet spot, we adaptively combine them.

Interval encoding exploits the fact that the provenance of a window is the union of the provenance of the tuples in the window. These tuples form contiguous sub-sequences of the input sequence. This method encodes a TID-Set as a list of intervals spanning continuous sequences in the set. For example, consider the interval encoding (Figure 9(b)) for the example network shown in Figure 9(a). The provenance of tuple 3:1 is represented as a single interval $[1,4]$, because the TID-Set forms a single contiguous sequence of TIDs (1 to 4). Interval encoding is most advantageous for queries involving aggregations over long sequences of contiguous TIDs, but introduces overhead if such sequences do not occur - both the start and end TID of an interval need to be stored.

Delta Encoding: Delta encoding utilizes the fact that windows with small slide values overlap to a large extent. Therefore, the TID-Set of a tuple may be encoded more efficiently by representing it as some delta to the TID-Set of one of its predecessors (by encoding which TIDs at the start of the previous set are left out and which TIDs are appended to the end). We repeatedly send a tuple with uncompressed provenance followed by several tuples with their provenance encoded as a delta to the last uncompressed provenance that was sent. While this "restart" approach has a higher space overhead than encoding a TID-Set as a delta to the last delta, we can restore a TID-Set from its delta representation in a single step without the need to apply a long chain of deltas to the last uncompressed provenance.

EXAMPLE 6. *Consider how delta encoding handles the example from Figure 9(b). The provenance header of a delta compressed tuple contains an additional field storing the amount of overlap (number of TIDs) between the delta and the last complete TID-Set that was sent. The TID-Set of the first output tuple of the aggrega-*

(a) Query Network



(b) Compressed Provenance

Technique	Data	Provenance	Physical Representation
No compression	3:1 (14)	$\{1,2,3,4\}$	S 4 1 2 3 4 ...
	3:2 (13)	$\{2,3,4,6\}$	S 4 2 3 4 6 ...
	3:3 (12)	$\{3,4,6,7\}$	S 4 3 4 6 7 ...
Interval	3:1 (14)	$\{[1,4]\}$	I 1 1 4 ...
	3:2 (13)	$\{[2,4],[6,6]\}$	I 2 2 4 6 6 ...
	3:3 (12)	$\{[3,4],[6,7]\}$	I 2 3 4 6 7 ...
Delta	3:1 (14)	$0{:}\{1,2,3,4\}$	D 4 0 1 2 3 4 ...
	3:2 (13)	$3{:}\{6\}$	D 1 3 6 ...
	3:3 (12)	$2{:}\{6,7\}$	D 2 2 6 7 ...
Covering Interval	3:1 (14)	$[1,4]$	3:1 ... 14 1 4
	3:2 (13)	$[2,6]$	3:2 ... 13 2 6
	3:3 (12)	$[3,7]$	3:3 ... 12 3 7

Figure 9: Compression Techniques Example

tion is sent completely. The TID-Set of tuple 3:2 (3:3) shares three (two) TIDs with last full TID-Set (tuple 3:1). The provenance of these tuples is encoded as deltas storing the overlap (3 respective 2) and the additional TIDs ($\{6\}$ respective $\{6,7\}$).

For Delta encoding, we have to cache the last complete TID-Set that was sent and the number of deltas applied to it. Operators in PP-mode may have to reconstruct TID-Sets from the delta representation. For example, an aggregation in PP-mode needs to do so to merge the provenance for a window of tuples. In contrast, Projection can simply pass on delta compressed provenance. The same applies for selection except that selection may filter out a tuple with a complete TID-Set leaving the following deltas orphaned. We handle this situation by reconstructing the full TID-Set for the first orphaned tuple and adapting the following deltas.

Dictionary Compression: If the size of a TID-Set exceeds a threshold we can use dictionary compression techniques (we use LZ77) to compress it. This type of compression can reduce the cost of forwarding significantly for large TID-Sets at the cost of additional processing to compress and decompress TID-Sets.

Adaptive Combination of Compression Techniques: Our prototype combines the presented compression techniques using a set of heuristic rules that determine when to apply which type of compression. Generally speaking, we first choose whether to use intervals or a TID-Set, then apply delta-encoding on-top if the overlap between consecutive TID-Sets is high, and finally apply dictionary compression if the result size still exceeds a threshold.

5.2 Lazy Generation and Retrieval

We now introduce two optimizations that decouple query processing and provenance operations to save computation cost.

Replay-Lazy: The *Replay-Lazy* method introduced in Section 2.2 computes provenance by replaying parts of the input through a provenance generating network. *Replay-Lazy* can be advantageous for several reasons: (1) the cost of provenance generation is only paid if provenance is actually needed, (2) the overhead on regular query processing is minimal, enabling provenance for time-critical applications, and (3) provenance computation is mostly decoupled from query execution. Thus, provenance generation can be performed later or on different resources, e.g. a distributed system.

Replay-Lazy is only feasible for query networks consisting of deterministic and monotone operators. Furthermore there is one critical concept required to make replay feasible: With no additional information, the whole input of the query network (i.e. stream prefix up to this point) has to be replayed through a PGN until the output of interest is produced. This can be avoided if we can computate which parts have to replayed while executing the query. We can prove that for all monotone and deterministic operators, replaying all tuples from the interval spanned by the smallest and largest TID in the provenance of an output tuple (we refer to them as the *covering interval* of a TID-Set) is sufficient. The proof of this property requires induction over the structure of a query network. We sketch the base case here. Consider a consecutive subsequence S of an operator's output stream where the operator reads from the inputs of the network. We have to show that replaying all tuples in the covering interval of S produces S. For example, *Selection* and *projection* generate a single output from a single input tuple solely based on the values of this tuple. Thus, applying them on the covering interval for S will produce S. For *aggregation* we need to guarantee that we open and close windows at the same positions when replaying the covering interval. With regarding to opening, the semantics of Borealis windows yield a opening at the beginning of the stream. Since we take the original window opening as the start of the covering interval, we get exactly the same (first) window opening. For closing, the arguments are analogous. By inductively applying these arguments we can show that replaying the covering interval for a result is sufficient to reproduce this result.

Replay-Lazy in Ariadne is based on these observations. The network is instrumented in the same way as for *Reduced-Eager*, except that we annotate each tuple with its covering interval. Given that these intervals require constant storage space, we can piggyback them on data tuples instead of sending the possibly unbounded TID set in additional tuples. This significantly reduces the amount of data to be propagated, reducing the processing cost on the tuple queues. Furthermore, covering intervals can be generated very efficiently during operator execution. The most complex case is aggregation where we compute the covering interval for a result tuple as the minimum and maximum TID values in the covering intervals for the tuples in the window.

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. Since this computation does not handle covering intervals one-by-one, but in a streaming fashion, we will encounter some issues with overlapping covering intervals and gaps between covering intervals. The input to a window operator is not necessarily a consecutive subsequence of the input, but a concatenation of subsequences that may have overlap or gaps. Thus, the operator may produce different windows when run over a concatenation of covering intervals. We address this problem by (1) replaying the overlapping parts of covering intervals only once and (2) forcing operators to drop state if there is a gap between consecutive intervals. The c-join operator sends a control tuple after the last tuple from each covering interval. This control tuple instructs downstream operators to drop their internal state (e.g., open windows) and flush their buffers (b-sort).

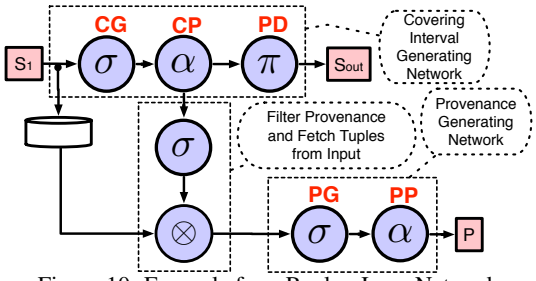


Figure 10: Example for a Replay-Lazy Network

EXAMPLE 7. *Consider the covering intervals shown in Figure 9(b). The TID-Set for tuple 3:1 is covered by the interval* $[1,4]$. *This covering interval is stored in two additional fields at the end of the data tuple 3:1. For this tuple the covering interval is the same as the interval encoding of the TID-Set. In general, this is obviously not the case. For example, the covering interval for tuple 3:2 contains TID* 5 *that is not in the provenance of 3:2. Figure 10 shows how to instrument the query network from Figure 9(a) for Replay-Lazy. The operators in the original part of the network are set to produce covering intervals (we refer to the covering interval version of PG and PP-mode as CG and CP-mode). The output of this part of the network is then routed through a selection to filter out parts of the provenance according to the user's preferences. Afterwards, we use a c-join to fetch all tuples with TIDs of the covering interval from the connection point and route these tuples through a provenance generating copy of the query network to produce provenance for tuples of interest.*

Lazy Retrieval: Our provenance generation approaches (both *reduced-eager* and *replay-lazy*) reduce the runtime cost of provenance generation by shifting computational cost to tuple reconstruction when retrieving provenance. If interactive retrieval is used, we only need to reconstruct provenance for tuples when explicitly requested. If the reconstruction result is further processed by a query network (queries over provenance information), we have the opportunity to avoid the cost of reconstruction through a p-join operator if we can determine that parts of the provenance are not needed to answer the retrieval part of the query. To this end we try to push selections that are applied during retrieval of provenance through the reconstruction (p-join) if the selection condition does not access attributes from the provenance, i.e., we use the following algebraic equivalence:[4] $\sigma_c(S \ltimes CP) \equiv \sigma_c(S) \ltimes CP$.

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.

Figure 11 shows the query network (called *Basic* network) used in most experiments in its original (a), rewritten (b) and instrumented (c) version. Details of the rewrite can be found in [15]. 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).

[4]For conjunctive selection conditions, we can split the condition and push conjuncts that only reference attributes from $schema(S)$ through the p-join.

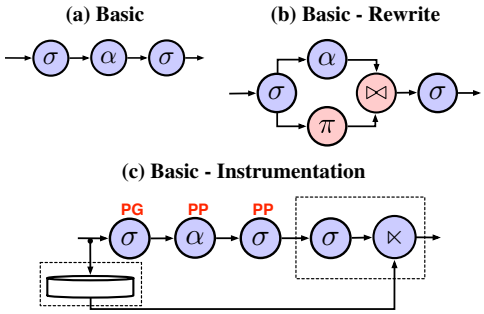


Figure 11: Experiment Queries

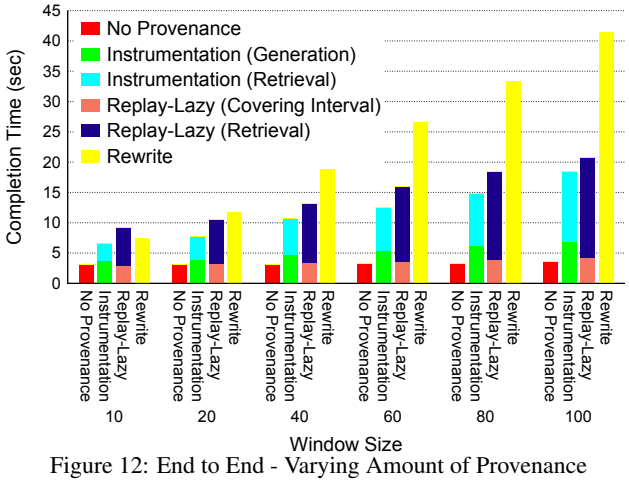


Figure 12: End to End - Varying Amount of Provenance

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 2.26 Ghz quad-core CPUs, 24GB RAM, running Ubuntu Linux 10.04 64 bit. Both the client (load generator) and the 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 10 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.

6.1 Fundamental Tradeoffs

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 12) compares the end-to-end cost when *changing the amount of provenance*

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

Figure 13: Varying Aggregations: Completion Time (Sec)

that is being produced per result tuple. 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 (*NoProvenance*), while the effort spent for replaying shows the same behavior as the total cost of *Instrumentation*. Finally, the results (Figure 13) 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 3300% (two aggregations). At three aggregations, the execution exhausts the available memory.

6.2 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 shown before, 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 Prefilter *Selectivity* (TID contiguity). These experiments use the *Basic* network.

Window Size: Figure 14(a) shows *Completion Time* for varying WS from 50 to 2000. A front filter selectivity of 25% ensures that there are very few contiguous TID sequences, limiting the potential of *Interval Compression*. As expected, completion time is higher for larger window sizes, but compression mitigates this effect: While the completion time overhead for *Single* increases from 70% to 530%, compression reduces it to 50% and 140%, respectively. Covering Intervals further reduces it to 14% and 70%. The cost savings are even more pronounced for queue sizes and memory consumption, where the overhead is reduced to a small, almost constant factor. For space reasons we omit these graphs.

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 14(b)). 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 Contiguity 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 14(c)). 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 increasing selectivity as more and more contiguous TID ranges are created. We therefore see no further increase in cost for selectivities over 75%.

6.3 Influence of Network Load on Latency

In reality, a query network is rarely run at maximum load, Thus, other 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 15, 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, *Optimized* reduces this overhead to around 60%, while *Covering Intervals* is the cheapest with around 20% overhead.

6.4 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 16 shows, the overhead of *Reduced-Eager* instrumentation without compression (*Single*) is higher than

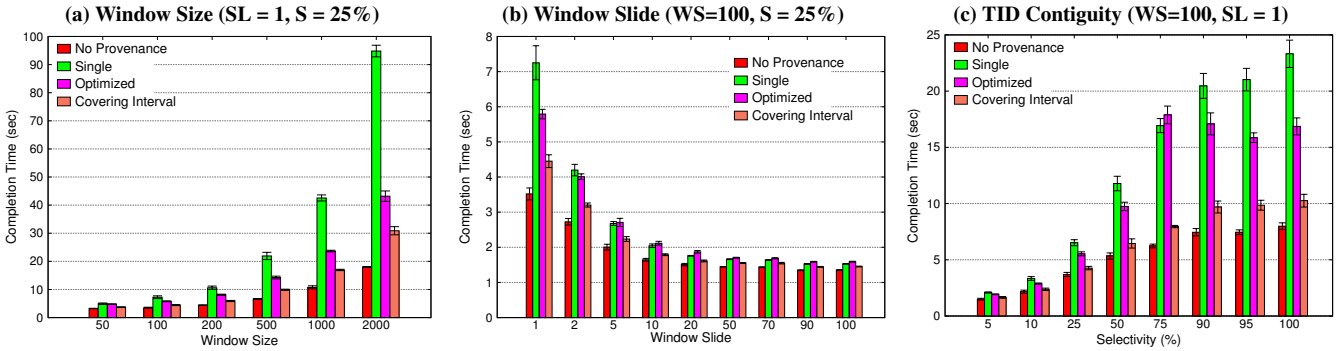


Figure 14: Impact on Completion Time

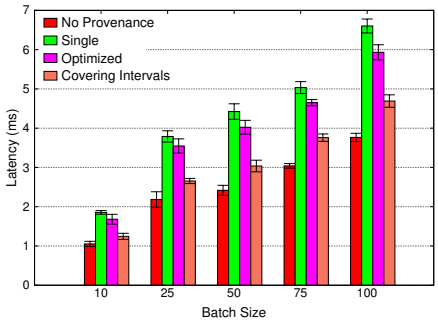


Figure 15: Latency For Varying Load

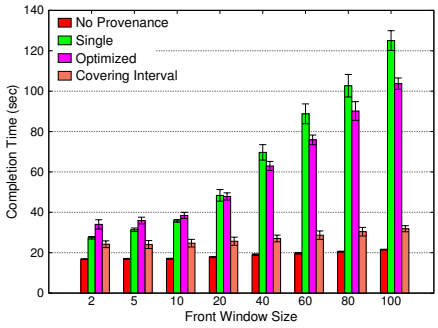


Figure 16: Complex Network

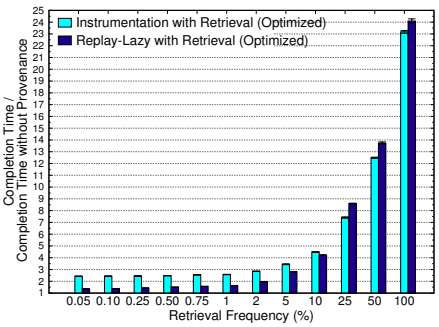


Figure 17: Varying Retrieval Frequency

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.

6.5 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 17 (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.

6.6 Summary

Our experiments demonstrate the feasibility of fine-grained end-to-end provenance in data stream systems and the benefits of our approach. *Operator Instrumentation* clearly outperforms *Rewrite*, since provenance generation is more efficient. Furthermore, Reduced-Eager allows us to separate generation and retrieval. Replay-Lazy based on covering intervals further reduces the overhead on the "normal" query network and enables us to scale-out. 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 previous work on provenance management in workflow systems, databases, and stream processing systems.

Workflow Systems. Workflow provenance approaches that handle tasks as black-boxes where all outputs of a task are considered to depend on all of its inputs are not suitable for managing stream provenance [11]. More recently, finer-grained workflow provenance models have been proposed (e.g., allowing explicit declarations of data dependencies [6] or applying database provenance models to Pig Latin workflows [5]). These systems only support non-stream processing models. Furthermore, for stream provenance, data dependencies can be inferred from the well-defined operator semantics, without explicit declarations. Ariadne's compression techniques resemble efficient provenance storage and retrieval techniques in workflow systems (e.g., subsequence compression technique [6] or node factorization [9]). However, due to the transient and incremental nature of streaming settings, we use compression mainly for optimizing provenance generation. Furthermore, it is more critical for our techniques to be efficient in terms of memory usage and encoding/decoding overhead, since compression at an operator may also affect the load on its downstream operators (e.g., it may be necessary to decompress provenance).

Database Systems. There are several different notions of database provenance [10] supported by different systems (e.g., Trio [7], DBNotes [8], Perm [13]). Like the *lineage provenance semantics* in relational databases [10], Ariadne represents the provenance of an output tuple as a set of input tuples that contributed to its gener-

ation. In principle, our Reduced-Eager operator instrumentation techniques can be extended to support more informative provenance models similar to database provenance models such as provenance polynomials [16] and graph-based models [2]. A major advantage of some of these models is that they are invariant under query equivalence. However, it is unclear what equivalences 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 [4, 21] would work for stream provenance is promising future work.

Stream Processing Systems. There is only a handful of related work on managing stream provenance. Vijayakumar et al. have proposed coarse-grained provenance collection techniques for low-overhead scientific stream processing [25, 26]. Wang et al. have proposed a rule-based provenance model for sensor streams, where the rules have to be manually defined for each operation [20]. More recently, Huq et al. have proposed to achieve fine-grained stream provenance by augmenting coarse-grained provenance with time-stamp-based data versioning, focusing specifically on query result reproducibility at reduced provenance metadata storage cost [17] using provenance inference techniques [18]. In this work, provenance generation is based on inversion, as opposed to Ariadne's propagation-based approach. In contrast to *Ariadne*, the approach is only applicable to a small class of streaming operators and does not always guarantee correct provenance.

A common use case for stream provenance data is query debugging. Microsoft CEP server [3] exposes the state of the system through snapshots (containing runtime statistics) and streams of manageability events (e.g., query start, a query failure, stream overflow, etc.). This information can be used to track coarse-grained provenance. The visual debugger proposed in [12] supports fine-grained provenance computation based on identifier annotation and operator instrumentation. Our approach is more general in that we support multi-step provenance, can decouple provenance computation from regular query processing, and compress provenance. We expose provenance as regular stream data and, thus, queries over provenance can be expressed using standard streaming operators.

8. CONCLUSIONS

In this paper, we present *Ariadne*, a prototype system addressing the challenges of computing fine-grained provenance for data stream processing. *Reduced-Eager operator instrumentation* provides a novel method to compute provenance for an infinite stream of data that adds only a moderate amount of latency and computational cost and correctly handles non-deterministic operators. *Replay-Lazy* and *Lazy-Retrieval* provide additional optimizations to decouple provenance computation from stream processing, further reducing the impact on critical paths and saving cost when provenance is not needed. Query networks can also be partially instrumented, catering for use cases like stream debugging that do not always need end-to-end provenance. 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.

9. REFERENCES

[1] D. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] U. Acar et al. A Graph Model of Data and Workflow Provenance. In *USENIX TaPP Workshop*, 2010.

[3] M. H. Ali et al. Microsoft CEP Server and Online Behavioral Targeting (Demonstration). In *VLDB*, 2009.

[4] Y. Amsterdamer et al. On Provenance Minimization. In *PODS*, 2011.

[5] Y. Amsterdamer et al. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 5(4), 2011.

[6] M. K. Anand et al. Efficient Provenance Storage over Nested Data Collections. In *EDBT*, 2009.

[7] O. Benjelloun et al. ULDBs: Databases with Uncertainty and Lineage. In *VLDB*, 2006.

[8] D. Bhagwat et al. An Annotation Management System for Relational Databases. *VLDB Journal*, 14(4), 2005.

[9] A. Chapman et al. Efficient Provenance Storage. In *SIGMOD*, 2008.

[10] J. Cheney et al. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2009.

[11] S. B. Davidson et al. Provenance in Scientific Workflow Systems. *IEEE Data Engineering Bulletin*, 32(4), 2007.

[12] W. De Pauw et al. Visual debugging for stream processing applications. In *Proceedings of the International Conference on Runtime Verification*, pages 18–35, 2010.

[13] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE*, 2009.

[14] B. Glavic et al. The Case for Fine-Grained Stream Provenance. In *BTW DSEP Workshop*, 2011.

[15] B. Glavic et al. Ariadne: Managing fine-grained provenance on data streams. Technical Report 771, ETH Zurich, 2012. available at: `http://cs.iit.edu/~glavic/publications.html`.

[16] T. J. Green et al. Provenance Semirings. In *PODS*, 2007.

[17] M. Huq et al. Facilitating Fine-grained Data Provenance using Temporal Data Model. In *VLDB DMSN Workshop*, 2010.

[18] M. Huq et al. Adaptive Inference of Fine-grained Data Provenance to Achieve High Accuracy at Lower Storage Costs. In *e-Science*, 2011.

[19] Z. G. Ives et al. The ORCHESTRA Collaborative Data Sharing System. *ACM SIGMOD Record*, 37(2), 2008.

[20] A. Misra et al. Advances and Challenges for Scalable Provenance in Stream Processing Systems. In *IPAW Workshop*, 2008.

[21] D. Olteanu and J. Závodný. On Factorisation of Provenance Polynomials. In *USENIX TaPP Workshop*, 2011.

[22] F. Reiss and J. Hellerstein. Data Triage: An adaptive Architecture for Load Shedding in TelegraphCQ. In *ICDE*, 2005.

[23] E. Ryvkina et al. Revision Processing in a Stream Processing Engine: A High-Level Design. In *ICDE*, 2006.

[24] N. Tatbul et al. Load Shedding in a Data Stream Manager. In *VLDB*, 2003.

[25] N. Vijayakumar and B. Plale. Towards Low Overhead Provenance Tracking in Near Real-time Stream Filtering. In *IPAW Workshop*, 2006.

[26] N. Vijayakumar and B. Plale. Tracking Stream Provenance in Complex Event Processing Systems for Workflow-Driven Computing. In *VLDB EDA-PS Workshop*, 2007.

[27] A. Woodruff and M. Stonebraker. Supporting Fine-grained Data Lineage in a Database Visualization Environment. In *ICDE*, 1997.