Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report Nr. 771

Systems Group, Department of Computer Science, ETH Zurich

Ariadne: Managing Fine-Grained Provenance on Data Streams

by

Boris Glavic
Kyumars Sheykh Esmaili
Peter M. Fischer
Nesime Tatbul

September 2012

inf | Informatik
Computer Science

Abstract

Managing fine-grained provenance is a critical requirement for data stream management systems (DSMS) to be able to address complex applications that require diagnostic capabilities and assurance as well as serving as a supporting technology for other tasks such as revision processing. In this paper, based on an example use case, we motivate the need for fine-grained provenance in stream processing and analyze its requirements. Inspired by these requirements, we investigate different techniques to generate and retrieve stream provenance, and propose a new technique that is based on operator instrumentation. Ariadne, our provenance-aware DSMS implements this technique on top of the Borealis system. We propose new optimization techniques to reduce the computational overhead of provenance generation and retrieval. Our experiments confirm that by applying these optimizations, Ariadne can provide fine-grained provenance with acceptable overhead.

Ariadne: Managing Fine-Grained Provenance on Data Streams

Boris Glavic, Kyumars Sheykh Esmaili, Peter Fischer, Nesime Tatbul

October 21, 2012

1 Introduction

Stream processing has recently been gaining traction in a new class of applications that require diagnostic capabilities and assurance [15]. In these applications, there is a common need to provide "fine-grained provenance" information (as provided by traditional database provenance semantics [11]), in order to trace an output event back to the input events contributing to its existence. For example, in sensor-based manufacturing control systems, upon receiving alerts that signal critical situations like overheating, human supervisors would like to understand why/how these alerts were triggered as to assess their relevance and react accordingly.

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 data management, workflow systems [13], and relational databases [11]. 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. First, data streams involve transient data that is possibly unbounded, which prohibits having a full view over all data items. Second, streaming data is typically ordered (e.g., by time), requiring a provenance model that incorporates order. Third, continuous queries over streams typically make extensive use of windowed aggregation which may lead to enormous amount of provenance data that should be managed efficiently. Fourth, streaming applications are time-critical, requiring a light-weight mechanism in order to maintain low latency for data as well as its provenance. Last but not least, streaming queries can behave non-deterministically (e.g., due to approximations or uncertain inputs). In combination, these challenges restrict the applicability of some well-known provenance management techniques (e.g., query rewrite [14]) and naive solutions (e.g., taking advantage of cheap, fast storage by dumping all inputs and inferring provenance from the complete stream data).

In the area of data stream management systems (DSMS), there has been little work beyond coarse-grained source provenance [27], for reasons of potentially high overhead and lack of application demand. In this paper, we argue that in fact, many stream processing applications require fine-grained provenance, and present Ariadne, a provenance-enabled DSMS that can provide this functionality with low overhead.

In this work, we first carefully analyze the application requirements and the main challenges to meet. In addition, we explore the design space with all its tradeoffs in terms of how fine-grained stream provenance can be generated, represented, and retrieved. We observe that the traditional provenance generation techniques based on query inversion or rewrite have limited applicability and high expected overhead.

We then propose a new, propagation-based approach that is based on operator instrumentation. In this approach, regular data tuples are annotated with their provenance while they are being processed by a query network of streaming operators. Propagation of provenance annotations is realized by replacing the operators of the query network with operators that create and propagate provenance annotations in addition to producing regular data tuples (we refer to this transformation as instrumentation). We have implemented our approach in Ariadne, a provenance-aware DSMS that is based on the Borealis prototype [1]. We have proposed a number of optimizations for provenance compression and on-demand retrieval, which significantly lower the overhead in the system. Our experiments study the fundamental tradeoffs across different techniques from the design space, show the benefits of our optimizations, and analyze sensitivity of performance against a number of factors including query complexity, window properties, and operator selectivity. The results demonstrate that providing fine-grained provenance via operator instrumentation is not only feasible, but is also optimizable and has tolerable overhead. While our implementation just picks one specific DSMS and one provenance management architecture, our overall analysis, provenance model, generation approaches, and optimizations are applicable to a much wider range of systems and implementations.

This paper thus makes the following contributions:

- It identifies the requirements and challenges of managing fine-grained provenance on data streams.
- It explores the design space for providing this capability in a DSMS.
- It introduces a new provenance generation technique for stream processing systems based on annotating and propagating provenance information through operator instrumentation.
- It proposes a number of optimization techniques for compressing and efficiently retrieving stream provenance.
- It presents the implementation of Ariadne, the first DSMS prototype providing support for fine-grained provenance.
- It provides an experimental evaluation of the proposed techniques on Ariadne.

The rest of this paper is organized as follows. First, we motivate the need for and identify the challenges of fine-grained stream provenance by means of an example use case in Section 2. Taking the identified challenges into account, Section 3 discusses the design space for integrating provenance generation and retrieval to a DSMS. We introduce the stream model and provenance semantics underlying our approach in Section 4. Building upon this model, we present its implementation in the Ariadne prototype in Section 6, and present a number of optimizations over our basic approach in Section 7. We discuss the results of our experimental study in Section 8, summarize the related work in Section 9, and finally conclude in Section 10.

2 Motivation and Challenges

Event stream processing has recently been gaining traction in applications that not only need to deal with large amounts of observed data (a traditional strong point of event processing), but also require control loops to react on this data. Additional requirements of such applications include human observation, assurance and recording of the application state, and query debugging [4]. Common to these new use cases of stream processing is the need to trace an output data item generated by the DSMS back to the

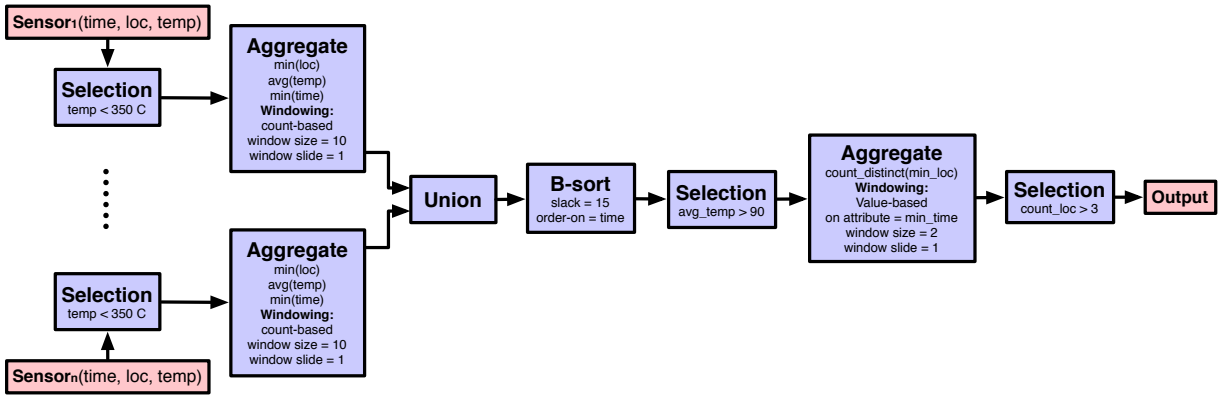


Figure 1: Running Example: Overheating Query

input data that contributed to its existence, i.e., tracking data stream provenance. We motivated this need for a diverse spectrum of use cases and identified general challenges in a recent position paper [15]. In this work, we focus on developing an efficient approach to extend a DSMS with provenance functionality.

2.1 Running Example

In monitoring and control of manufacturing systems, sensors are attached to machines and to key points along a supply chain. Sensor readings are processed by a DSMS in order to detect critical situations such as machine overheating, fire, standstill, and low inventory. These detected events are then used for automatic corrections as well as for notifying human supervisors.

Example 1 *Figure 1 shows a continuous query that is used to detect overheating for a simplified version of such a scenario. The input streams of the query are produced by a number of sensors measuring temperature. Each sensor stream is filtered to remove massive outliers (i.e., temperature above 350°C). The filtered stream is aggregated by averaging the temperature over a sliding window of 10 temperature readings in order 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 maintain global ordering on time across all sensors). 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 an interval of 10 time units.*

To react on events in such a scenario, a human supervisor needs to understand why/how these events were triggered to be able to assess their relevance. For example, if the DSMS outputs an overheating alarm event, the user would want to understand which sensor readings caused the event, i.e., belong to the *fine-grained* provenance of the event. For the example use case, provenance is requested in an ad-hoc manner for events of interest. Provenance should be provided efficiently for all alarms raised by the system to enable interactive drilldown if this information is requested by the user. Furthermore, the user should be able to define policies to assert his interest in provenance under certain circumstances, such as for overheating alarms which are based on unusually high temperature readings. That is the system should support queries over generated provenance information to, e.g., filter out parts of the provenance that are irrelevant for the user.

2.2 Challenges

In many stream-based use cases including the one above, data is essentially transient, rapid, time-ordered, and possibly unbounded, while queries can behave non-deterministically due to approximate processing or uncertain inputs. These characteristics lead not only to stream-oriented data representations and processing models, but also more strict constraints on performance, both of which greatly affect how provenance should be managed. A major challenge is to find a solution that balances the amount of data needed to correctly represent provenance, while guaranteeing efficient generation and scalable retrieval. More specifically, provenance management on data streams must deal with the following list of challenges:

Online and Infinite Data Arrival: Data streams can potentially be infinite. This means that we usually do not have a full view on all items of a stream (i.e., some items may have not appeared yet). Moreover, it may be impractical or even impossible, to preserve all items for later processing. This limits the applicability of methods that reconstruct provenance from the query and input data on request.

Ordered Data Model: In contrast to the set or bag model of relational databases, data streams are typically modeled as ordered sequences, requiring a provenance model that incorporates order. This ordering, however, can be exploited in providing optimized representations of provenance.

Window-based Processing: Operators like *aggregation* and *join* are typically processed in DSMSs by grouping tuples from a stream into windows and computing the result over the content of each window. Windowing is usually implemented using stateful operators. Stream provenance must deal with windowing behavior in order to trace the outputs of such stateful operators back to their sources correctly and efficiently. The statefulness of operators restricts the applicability of certain provenance techniques (as will be explained in Section 3) and the prevalence of aggregations in queries leads to enormous amounts of provenance. However, windowing also enables new types of optimizations for provenance generation (such optimizations are presented in Section 7 and their behavior is analyzed in the experimental evaluation in Section 8).

Low-latency Results: Streaming applications have strict performance requirements where low latency should be maintained, even under very high data arrival rates. For example, in ad-hoc human inspection queries, the events of interest are relevant only for short periods of time and, thus, need to be available in a timely fashion. Provenance generation has to be light-weight enough not to violate the application's latency constraints for data and provenance.

Non-determinism: Mechanisms applied by DSMSs to cope with issues like high input rates (e.g., load shedding or approximations [23, 26]), unpredictable input source behavior (e.g., delays or disorder [25]), and certain operator definitions (e.g., windowing on system time, merging input streams based on arrival order, or operators with timeout parameters [2]), can lead to non-deterministic behavior. We call a query network *deterministic* if the output produced by the network solely depends on its input, i.e., the results are *reproducible*. For instance, a network that uses windowing based on system time is not deterministic, because two executions of the network over the same input may produce different results. Non-determinism restricts the applicability of some approaches developed for database provenance. For instance, query rewrite approaches used for database provenance are often not applicable to streaming queries, since most of these techniques require reproducibility of query results to deal with operations such as aggregation [12, 14] (this will discussed further in Section 3.1.2).

In this paper, we address these challenges, by providing a novel, fine-grained, propagation-based prove-

Method	Applicable to	Storage Overhead	Runtime Overhead	Retrieval Overhead
How to generate?				
Inverse	Invertible	-	None	High
Query Rewrite	Deterministic	-	High	-
Operator Instrumentation	All	-	Low	-
When to generate?				
Eager	All	-	Generation (high)	-
Reduced-Eager	All	Temp. Input Storage (low)	Generation (low) + Input Storage (low)	Reconstruct (low)
Replay-Lazy	Deterministic	Temp. Input Storage (high)	Input Storage (high)	Replay (high)

Figure 2: Comparison between Methods for Provenance Generation

nance management approach which exerts small overhead in the system through a number of optimizations.

3 Provenance Design Space

We now discuss alternative ways of how to design or extend a DSMS to support the requirements of use cases like the one presented in Section 2.1. We present approaches for generating and representing provenance, discuss when to generate provenance (*eagerly* during query execution or *lazily* on request), and discuss the tradeoffs of these methods with respect to *query non-determinism*, *latency*, and *provenance retrieval frequency* (the fraction of the output for which provenance is required). All options and relevant evaluation criteria are outlined in Figure 2.

3.1 Provenance Generation

We consider three approaches for provenance generation: (1) computing inverses, (2) rewriting the query network to propagate provenance annotations using the existing operators of the DSMS, and (3) instrumenting the operators of the query network to propagate provenance information. *Inverse* (e.g., Woodruff et al. [30]) and propagation through *Query Rewrite* (e.g., Glavic et al. [14]) are well-known techniques studied in the database and workflow provenance literature. *Operator Instrumentation* also applies propagation of provenance information, but in contrast to query rewrite, modifies the operators of the system to enable propagation. So far, we are not aware of any system that actually implements *Operator Instrumentation*.

3.1.1 Inversion

Inversion generates provenance by applying the inverse (in the mathematical sense) of an operator to an output to generate its provenance. Examples of invertible operators are join (without projection) and selection, because for these operators the inputs can be constructed from an output tuple. We call a network *invertible* if it only consists of invertible operators. In general, every invertible network is deterministic, but the opposite does not necessarily hold.

For most non-trivial operators, no inverse in the strict mathematical sense exists and, thus, additional information is required to compute the inverse. Even if the complete input data of the operator is present, it might still not be possible to invert it and we may have to resort to re-execution to generate its provenance.

For example, determining which input tuples have contributed to an output tuple from a value-based window is not possible if we do not expose the boundary conditions of a window instance in the output tuple. For such operations, inversion degenerates to propagation (by either rewrite or operator instrumentation).

3.1.2 Propagation by Query Rewrite

Similar to relational provenance systems such as *Perm* [14], *DBNotes* [9], or *Orchestra* [19], we can generate provenance for DSMS by rewriting a query network q into a network that generates the provenance of q in addition to the original network outputs. However, for this approach to be applicable, the query language of the DSMS has to be powerful enough to express the provenance computation for an arbitrary network expressed in this language.

Many query rewrite techniques require changes to the structure of the query network. For instance, we cannot propagate provenance information though operators like aggregation directly without changing the results of this operator (see [14] for a discussion on the topic). To generate the provenance for such operators, a provenance-generating copy of the sub-network that generates the operator's input has to be added to the network and joined with the original sub-network, leading to a tight coupling of query execution and provenance computation. This results in query networks with large number of joins between streams with imbalanced arrival rates. For instance, the provenance of an aggregation operator is usually much larger than the operators output. Unless the DSMS is applying sophisticated load balancing and scheduling techniques, the imbalance in the arrival rates of join inputs will lead to unbounded growth of join buffers and, thus, poor throughput and latency. Furthermore, two copies of a sub-network may produce different results if the sub-network is non-deterministic (e.g., using a random number generator function or windows based on system time). In this case, the rewritten network will fail to attach the correct provenance to an original result.

3.1.3 Propagation by Operator Instrumentation

The key idea behind the 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 on its inputs. We refer to this modification of an operator's behavior as *Operator Instrumentation*. Since these annotations can be processed in line with the regular data, the original query plan can be kept as is. Thus, most issues caused by non-determinism are dealt with in a natural way, since the execution of the original query network is traced. The only exception is that the overhead introduced by provenance generation may affect temporal conditions, e.g., the content of a window based on system time may change. A drawback of *Operator Instrumentation* is the need to extend all operators. Since operator instrumentation generates provenance without additional operators to the query network, its runtime and latency overhead is usually lower than for rewrite. Choosing the right provenance representation and compression can further mitigate this overhead.

3.2 Eager and Lazy Provenance Generation

Provenance can either be generated *eagerly* by producing complete provenance while the query network is running, or *lazily* by postponing the provenance generation (or parts thereof) to when it is requested.

We now discuss the advantages and disadvantages implied by eager and lazy approaches for provenance generation. In its extreme form, lazy generation would not produce any provenance during the execution of the query network, i.e., the query network is executed with no runtime or latency overhead. However, this is only possible for networks that are completely invertible, and as the previous section shows, such networks are rare in practice. For non-invertible deterministic networks, we can realize lazy generation by temporarily storing input tuples and replaying these inputs though a copy of the network modified for provenance generation (e.g., using query rewrite techniques). We call this approach *Replay-Lazy*. The naive implementation of Replay-Lazy requires the storage and replay of the entire input that was processed before the generation of the output for which provenance is computed, which can be prohibitively expensive due to the long-running nature of streaming queries. Thus, for this approach to be practical we would need to record additional information to be able to reduce the amount of data that is stored and replayed. For example, we could store for each output which parts of the input are needed for the replay to be executed correctly (e.g., store a superset of the provenance). This idea leads into the direction of eagerly generating a limited type of provenance information (e.g., representing provenance as tuple identifiers) to reduce the runtime overhead at the cost of spending additional time to reconstruct the full provenance from the limited representation for retrieval. We refer to this approach as *Reduced-Eager*.

3.3 Provenance Representation and Retrieval

We now discuss how to query provenance and represent this information *externally* to a user and *internally* for efficient operations. External representations should be informative enough to be easily interpretable and queryable. Complete input tuples seem to be a better fit for *external* provenance representation than, e.g., sets of tuple identifiers, because the latter are quite meaningless to a human. *Internally*, we may prefer to use a more compact representation of provenance to increase performance and save storage space, and only expand it to the external representation if needed (this corresponds to the Reduced-Eager approach described above).

Two options for querying provenance have been established in related work: Develop a new data model and query language for provenance [9, 20], or represent provenance using the original data model and query it using the original query language [14]. When developing a new query language for provenance it can be custom fitted for provenance querying. However, the development effort will be larger than for adding new features to an existing query language and the support for query non-provenance data will probably very limited.

3.4 Summary of Tradeoffs

Figure 2 shows the tradeoffs implied by when and how to compute provenance in a DSMS. Pure *Inverse* is only applicable for a restricted set of operators and queries, thus restricting its use to a small set of special cases or as an additional optimization. *Query Rewrite* is restricted to deterministic networks and carries a high runtime and latency overhead, since it needs to introduce additional sub-networks which need be tightly coordinated with the regular query execution. As a result *Query Rewrite* should typically only be used if the price of extending operators and possibly also their cost models for *Operator Instrumentation* cannot be paid. The advantage of *Operator Instrumentation* over *Query Rewrite* will be confirmed in Section 8.

Eager computation is applicable to all query network types and does not require storing any input tuples, but propagating full tuples through all operators is very costly and prohibits any further optimizations, making it only suitable when storage is limited and computational cost is secondary. *Reduced-Eager* needs to store input tuples and therefore pays a price in terms of storage and computational cost for storing and reconstructing these tuples. This cost is however offset by a significant reduction in provenance generation cost (witnessed by both decreased runtime and latency), since compressed representations can be used. *Replay-Lazy* further reduces the runtime overhead by just computing some minimal provenance, but incurs higher retrieval cost due to the replay and is only applicable to deterministic networks. *Reduced-Eager* is beneficial if provenance is requested often (high retrieval frequency). *Replay-Lazy* is preferable for low retrieval frequency. We study this tradeoff in Section 8.6.

Various architectures to support DSMS provenance can be designed within this design space, ranging from storage-centric approaches which compute provenance on demand to computation-centric approaches which output provenance continuously. The basis for a provenance-enabled streaming system can be a conventional DSMS, fast DBMS, or a hybrid system, coupled with a scalable storage system. Yet, as this design space analysis shows, no solution can exist without substantial extensions on the underlying system in order to deal with the specifics of data stream systems outlined in Section 2.

For Ariadne, the system presented in this paper, we settled on an existing DSMS, as it provides the highest fidelity in terms of typical DSMS semantics. We chose *Reduced-Eager Operator Instrumentation* as our "workhorse", given its general applicability and moderate overall cost. *Replay-Lazy* is used as an optimization for workloads with low retrieval frequency. In spite of its limitations, we still chose to also develop a *Rewrite* approach to be able to verify our claim about its inferior performance characteristics experimentally (see Section 5).

4 Provenance Annotation Propagation by Operator Instrumentation

In this section, we introduce a provenance model for streams and then discuss how to extend queries to annotate their outputs with provenance information according to this model. We first define a stream data and query model. The choice of operators included in the query model is based on the operators implemented in Ariadne, which correspond to the operators in Borealis [1]. Since these operators include the most commonly used streaming operators such as windowed aggregation and joins, we can easily adapt our model for other DSMS.

A formal provenance model requires a formal underpinning in terms of a data model and query language, e.g., like the relational algebra in databases. Since there is no generally accepted formal model for data streams, we first establish the necessary definitions for the data model (Section 4.1) and operator semantics (Section 4.2), using a recursive prefix-based model. We have tried to keep this model as generic and minimal as possible, so that it will apply to many existing DSMS. Even if not all operations of a DSMS map directly to the formalism, the provenance model itself can be easily adapted

4.1 Data Model

We model streams as (possible infinite) sequences of so-called *stream items*. Each stream stores items of a specific type. In our model we use three item types: *tuples*, *windows*, and *join-windows*. Tuples are lists of attribute values that conform to a given schema (attribute name and domain pairs). Windows are ordered sequences of tuples and are used to define stream operators that compute output items based on subsequences of their input stream. Similar, join-windows, storing two windows from different streams, are used in the definition of the join operator.

Definition 1 (Stream Item and Stream) *A **stream item** i is of a type $Type \in \{T, W, JW\}$ that defines its structure: A stream item of type T (a **tuple**) is an element $t = [tid, a_1, \ldots, a_n]$ from $\mathscr{T} \times D_1 \times \ldots \times D_n$ for a list of domains $D_1, \ldots, D_n$ and a set $\mathscr{T}$ of tuple identifiers. Let $\mathscr{T}(t)$ denote the identifier of tuple t which is required to be unique. We reserve the attribute name TID for the attribute that stores the $\mathscr{T}(t)$ value of t. A stream item of type W (a **window**) is a finite sequence of tuples denoted as $w = \ll t_1, \ldots, t_n \gg$. A stream item of type JW (a **join-window**) is a tuple $jw = [w_1, w_2]$ where w_1 and w_2 are windows. A **stream** S of type $Type$ is a, (possibly infinite) sequence of stream items of type $Type$ denoted by $\ll i_1, \ldots \gg_{Type}$ ($Type$ is omitted if clear from the context). We use $S[i]$ to denote the i^{th} element of stream S, and S_{Type} to denote that stream S is of type $Type$.*

For example, $S = \ll\ll t_1, t_2 \gg, \ll t_2, t_3 \gg\gg_W$ with $t_1 = [\mathscr{T}_1, 5]$, $t_2 = [\mathscr{T}_2, 7]$, and $t_3 = [\mathscr{T}_3, 12]$ is a stream of type W containing two windows; each of them containing two tuples. Fig. 3 summarizes the notations we use in the stream algebra and provenance definitions (some will be introduced later).

4.2 Stream Algebra

We now present an algebraic formalization of stream operators. A stream operator produces one or more tuple output streams from one or more tuple input streams. In the definitions we use some auxiliary functions presented in the following. Applying the head function H to a stream S returns the first item in the stream ($H(S) = S[1]$). The result of applying the tail function T to a stream S_{Type} is the original stream with the first element removed ($T(S) = \ll S[2], S[3], \ldots \gg_{Type}$). Both head and tail are also defined to return resp. remove m stream items (e.g., $H(S, m) = \ll S[1], \ldots, S[m] \gg_{Type}$). The concatenation of a stream item i and a sequence S or of two sequences S_1 and S_2 is defined as: $i \mid\mid S = \ll i, S[1], \ldots \gg_{Type}$ and $S_1 \mid\rightarrow S_2 = \ll S_1[1], \ldots, S_1[l(S_1)], S_2[1], \ldots \gg_{Type}$ where $l(S)$ denotes the number of items in sequence S. For a tuple t, $t.\mathscr{N}$ is the tuple without its identifier.

We first present the definitions for *selection* and *projection* that directly operate on input tuples without grouping them into windows. Afterwards, we present two auxiliary classes of operators, called *windowing* and *join windowing*, that are used in the definition of *aggregation* and *join* presented in the following. In the operator definitions we use new to denote a function that generates new TID values for the output of an operator. To not loose generality we only require that new is deterministic (it generates the same values for the same input).

Selection: A selection $\sigma_c(I)$ on condition c filters out tuples from a stream that do not fulfill the condition c.

$$\sigma_c(I) = \begin{cases} [new, H(I).\mathscr{N}] \mid\mid \sigma_c(T(I)) & \text{if } H(I) \models c \\ \sigma_c(T(I)) & \text{else} \end{cases}$$

S_{Type}	Stream S is of type $Type \in \{T, W, JW\}$
$[tid, a_1, \ldots, a_n]$	Item of type T (tuple)
$\ll t_1, \ldots, t_n \gg$	Item of type W (window) with $t_i \in T$
$[w_1, w_2]$	Item of type JW with $w_1, w_2 \in W$
$H(S)$	First stream item of sequence S
$H(S,n)$	Sequence containing the first n items of sequence S
$T(S)$	Sequence S with first element removed
$T(S,n)$	Sequence S with first n elements removed
$i \mid\mid S$	Sequence S with item i added at the beginning
$S_1 \mid\rightarrow S_2$	Concatenation of sequences S_1 and S_2
TID	Name of the attribute for tuple identifiers
new	Function that generates new tuple identifiers for the output of an operator
$t.A$	Project tuple t on expressions A
$t.\mathscr{N}$	Project tuple t on its data (remove TID and/or provenance attribute)
$t.\mathscr{P}$	Project tuple t on the provenance set
$q[O]$	Output stream O from query network q
$S \cap . \mathscr{M}$	Remove elements from stream S that are not in M
$S \uparrow . \mathscr{M}$	Remove elements from stream S that follow the last element from M
q_O	Sub-network of network q that contains only nodes that influence output stream O

Figure 3: Notations

Projection: A projection $\pi_A(I)$ on a list of projection expressions A (attributes and application of functions) projects each input tuple on the expressions from A. In the definition $t.A$ denotes the projection of a tuple t on A.

$$\pi_A(I) = [new, H(I).A] \mid\mid \pi_A(T(I))$$

Windowing: A window operator is a function $\omega : I_T \rightarrow O_W$. I.e., groups tuples from a tuple stream into windows. As examples for a window operators we present *count-based* windowing and *value-based* windowing. The count-based window operator $\mathscr{C}(c,s)$ groups c (called *count*) tuples from the input into a window and skips s (called the *slide*) tuples before opening a new window:

$$\mathscr{C}(c,s)(I) = \ll H(I,c) \gg \mid\mid \mathscr{C}(c,s)(T(I,s))$$

The value-based window operator $\mathscr{V}(x,r,s)$ groups all tuples into a window that have an x attribute value that is smaller than the x attribute value of the first item in the window plus the parameter r (called *range*). Windows are advanced by s:

$$\mathscr{V}(x,r,s)(I) = \sigma_{x \leq H(I).x+r}(I) \mid\mid \mathscr{V}(x,r,s)(\sigma_{x \geq H(I).x+s}(I))$$

Join Windowing: A join-windowing operator is a function $j\omega : I_T \times I'_T \rightarrow O_{jw}$ that groups inputs from two tuple streams into join-windows. For a join-window $jw = [w_1, w_2]$ we denote the access to window w_i by $jw.w_i$. For example, *value-based* join-windowing ($jv(x_1, x_2, r)$) groups each tuple t from the left stream with all tuples from the right stream that have an x_2 attribute value between $t.x_1$ and $t.x_1 + r$.

$$\begin{aligned} jv(x_1,x_2,r)(I,I') =& [\ll H(I) \gg, \sigma_C(I')] \\ & \mid\mid jv(x_1,x_2,r)(T(I),I') \\ C =& x_2 \geq H(I).x_1 \wedge x_2 \leq H(I).x_1 + r \end{aligned}$$

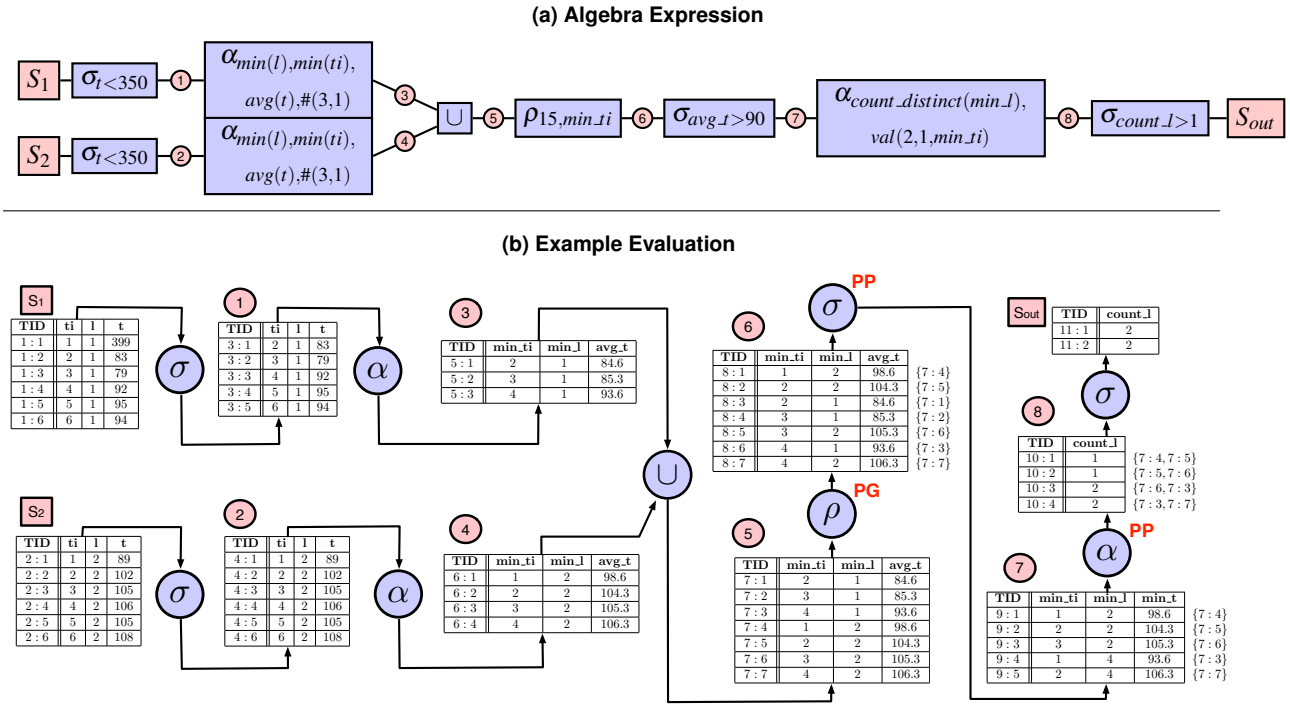


Figure 4: Example Query Network with Data

Aggregation: An aggregation $\alpha_{agg,\omega}(I)$ partitions its input into windows using the window function ω and computes the aggregation functions from $agg = (agg_1(a_1),\ldots,agg_n(a_n))$ over each window generated by ω. Each 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 aggreation function $agg(a_i)$ to a window w as $agg(a_i,w)$.

$$\begin{aligned}\alpha_{agg,\omega}(I) &= a(\omega(I))\\ a(I) &= agg(H(I)) \mid\mid a(T(I))\\ agg(w) &= [new, agg_1(a_1,w),\ldots,agg_n(a_n,w)]\end{aligned}$$

Join: The join operator $\bowtie_{c,j\omega}(I_T,I'_T)$ joins two input tuple streams I and I' by applying the join windowing operator $j\omega$ to I and I', and for each generated join-window jw outputs each combination of a single tuple from $jw.w_1$ with a tuple from $jw.w_2$ that fulfills the join condition C.

$$\begin{aligned}\bowtie_{C,j\omega}(I,I') &= join(H(j\omega(I,I'))) \mid\!\rightarrow join(T(j\omega(I,I')))\\ join(jw) &= joinl([\ll H(jw.w_1) \gg, jw.w_2])\\ &\quad \mid\!\rightarrow joinl([T(jw.w_1), jw.w_2])\\ joinl(jw) &= \begin{cases} X \mid\mid Y & \text{if } [H(jw.w_1),H(jw.w_2)] \models C\\ Y & \text{else}\end{cases}\\ X &= [new, H(jw.w_1).\mathcal{N}, H(jw.w_2).\mathcal{N}]\\ Y &= joinl([jw.w_1, T(jw.w_2)])\end{aligned}$$

Union: A union operator $\cup(I,I')$ merges tuples from two input streams I and I' with the same schema into a single stream based on their arrival order. Union is unique in that the result of the union depends

the run-time behaviour of the system. We refrain from trying to fully model this behaviour, but instead just assume the existence of a function $Order(S_1, S_2)$ that sorts both streams S_1 and S_2 according to arrival order. This function models the non-deterministic behaviour of the union operator.

$$\cup(I, I') = [new, H(Order(I, I'))] \;||\; \pi_A(T(Order(I, I')))$$

B-Sort: A b-sort operator $\rho_{s,a}(S)$ with slack s and an order-on attribute a applies bounded-pass bubble sort with buffer size $s+1$ on its input, and thereby, produces an output that is approximately sorted on a. Let $pos(S, a, i)$ denote a function that returns the i^{th} tuple from stream S ordered on attribute a. Using this

$$\rho_{s,a}(S) = BSort(S, a, s, 0)$$
$$BSort(S, a, s, i) = pos(H(S, s + (i-1)), a, i) \;||\; BSort(S, a, s, i+1)$$

Example 2 *An algebraic query network for the overheating query with two sensors (Figure 1) is shown in Figure 4(a), with streams inside the graph labeled 1 to 8 for easier referencing. Figure 4(b) shows an execution of this network for a given input. Note that we choose smaller window sizes than in the original network to simplify the example. Both input streams (S_1 and S_2) have the same schema with attributes time (ti), location (l), and temperature (t). The query network uses selection ($\sigma_{t<350}$) to filter out temperature outliers and groups each filtered input stream into windows of three tuples advancing the window by one tuple ($\#(3,1)$). 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 ($\alpha_{min(l),min(ti),avg(t)}$). The aggregated streams are merged into one stream ($\cup$) and sorted on time (ρ_{15,min_ti}). We then filter out tuples with temperature values below the overheating threshold ($\sigma_{avg\_t>90}$). The result is grouped into windows with a maximal time difference of two time units ($val(2,1,min\_ti)$) and the number of distinct location values is computed for each window ($count\_distinct(min\_l)$). Tuples with less than two distinct locations are filtered out in the last step ($\sigma_{count\_l>1}$). For instance, in the example execution shown in Figure 4b, 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).*

4.3 Declarative Provenance Semantics

We now present the contribution semantics (definition of provenance) applied by *Ariadne* that models the provenance of an output tuple t of a query q as a provenance set, the set of tuples from the input stream(s) of q that were used to derive t. The declarative definition of our contribution semantics captures assumptions about provenance that one would intuitively expect to hold. (i) The provenance of a tuple should produce this tuple and nothing else. (ii) Provenance should not include tuples that did not contribute to the output. These intuitions are captured by stating conditions over the result of evaluating a query network over subsets of its input streams. E.g., by removing all tuples from the input that do not belong to the provenance set of an output tuple.

To be able to state such conditions we define two types of reduced input streams: Intersection of a stream

I with a set $\mathscr{M}$ is denoted by $I \cap \mathscr{M}$ and defined as:

$$I \cap \mathscr{M} = \begin{cases} H(I) \mid\mid T(I) \cap \mathscr{M} & \text{if } H(I) \in \mathscr{M} \\ T(I) \cap \mathscr{M} & \text{else} \end{cases}$$

The prefix $I \uparrow \mathscr{M}$ of a stream I according to a set $\mathscr{M}$ contains all tuples from the stream until the last (in order of the stream) tuple from $\mathscr{M}$:

$$I \uparrow M = \begin{cases} H(I) \mid\mid T(I) \uparrow M & \text{if } \exists t \in M : p_I(H(I)) \leq p_I(t) \\ T(I) \uparrow M & \text{else} \end{cases}$$

Prefix and intersection of a list of streams $\mathbb{I} = (I_1, \ldots, I_n)$ with a set are defined as the list generated by applying prefix respective intersection to each stream in the list:

$$\mathbb{I} \cap \mathscr{M} = (I_1 \cap \mathscr{M}, \ldots, I_n \cap \mathscr{M})$$
$$\mathbb{I} \uparrow \mathscr{M} = (I_1 \uparrow \mathscr{M}, \ldots, I_n \uparrow \mathscr{M})$$

For an output stream O of a query network q we define q_O as the query network that contains all nodes that are reachable from O if we reverse the edges in q. I.e. the sub-network that contains only streams and operators that may influence the evaluation of O. Furthermore, $I \subseteq I'$ denotes that all items from I are contained in I'. Having defined prefix, intersection, and q_O we now present our declarative provenance definition.

Definition 2 (Provenance Sets) *The Provenance set $\mathscr{P}(q, \mathbb{I}, t)$ of a result tuple t from an output stream O of a stream algebra expression q over a list of input streams $\mathbb{I} = (I_1, \ldots, I_n)$ is the minimal subset of all tuples from $\mathbb{I}$ ($Set(\mathbb{I})$) that fulfills the following conditions:*

$$q(\mathbb{I} \cap \mathscr{P}(q, \mathbb{I}, t))[O] = \ll x \gg with\, t.\mathscr{N} = x.\mathscr{N} \quad (1)$$
$$q(\mathbb{I} \uparrow P(q, \mathbb{I}, t))[O] = \ll \ldots, t \gg \quad (2)$$
$$\forall \omega \in q_O : \omega(\mathbb{I} \cap P(q, \mathbb{I}, t)) \subseteq \omega(\mathbb{I}) \quad (3)$$

The conditions of Def. 2 capture the intuitive assumptions presented beforehand. Condition 1 guarantees that the provenance of t is sufficient for producing t and only produces t. This is done by evaluating q over the provenance, checking that only a single tuple x with the same attribute values as t is returned (the TID of x may be different from the one of t, because the operators use new). The second assumption of $\mathscr{P}$ to be minimal is expressed by Conditions 2 and 3. Conditions 2 checks that $\mathscr{P}(q, \mathbb{I}, t)$ is the provenance of tuple t and not of some other output tuple with the same attribute values at a different position in the stream. This is achieved by applying the query to input streams prefixes up to the last tuple in the provenance, and checking that the last output tuple in the output stream O is t (With the same $\mathscr{T}$ value, since TID assignment is same for replays). Condition 3 plays the same role for operators with windowing. It requires that replaying the provenance does only produce windows that are produced by the original evaluation of q. The interested reader can verify that conditions 2 and 3 are necessary on the following example:

$$\alpha_{sum(a),\mathscr{C}(2,1)}(\ll [\mathscr{T}_1, 5], [\mathscr{T}_2, 5], [\mathscr{T}_3, 5], [\mathscr{T}_4, 5] \gg)$$
$$= \ll [\mathscr{T}_5, 10], [\mathscr{T}_6, 10], [\mathscr{T}_7, 10] \gg$$

Based on Def. 2 we define the *PEO* of q, a stream that contains original output tuples of q and their provenance.

Definition 3 (Provenance Enhanced Output Stream (PEO)) *For a query q over inputs $\mathbb{I}$ and an output stream O of q, the PEO $P(q,\mathbb{I},O)$ is a stream with schema $[\mathscr{O},\mathscr{P}:Set(\mathbb{I})]$ defined as:*

$$P(q,\mathbb{I},O) = [H(O),\mathscr{P}(op,\mathbb{I},H(O))] \,||\, P(q,\mathbb{I},T(O))$$

Example 3 *For instance, consider the PAS $P(6,\{5\})$ for the output of the b-sort operator according to its input shown in Figure 4(b) (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. This is a typical pattern for annotated outputs of operators using sliding windows. For example, tuple 10:2 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.*

4.4 Operators and Networks with Annotation Propagation

We now discuss how to translate a query network q into a network that generates the PAS for a subset of the streams in q by replacing operators in the network with operators that annotate their outputs with provenance information. For this, we introduce two new types of *provenance annotating operators* for each operator in our algebra:

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 for an operator from input streams without annotations. For each output stream S of the operator o the provenance generator $PG(o)$ creates $P(S,input(o))$ where $input(o)$ are the input streams of operator o.

Provenance Propagator (PP): This type 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 $P(S,\mathscr{I})$. The PP version of o will output $P(O,\mathscr{I})$, i.e., the output will be annotated with provenance sets of O according to $\mathscr{I}$. Intuitively, a PP generates annotated output streams by modifying the annotations of its input streams according to the provenance behavior of the operator.

In the definition of the PG and PP operators we use $\mathscr{N}$ to refer to all attributes of a tuple except for TID and $\mathscr{P}$.

Definition 4 (PG and PP) *The PG and PP stream algebra operators are defined as presented in Figure 5 and Figure 6 respectively.*

We briefly explain two operators and their PG and PP semantics here: For selection, the PG provenance of a tuple t is the tuple itself, shown as $\{H(I)\}$, in PP mode the provenance of the selected tuple,$H(P).\mathscr{P}$. For aggregation, the provenance of t is the set of all tuples in the window that generated t (in PG), respectively the union of their provenance (in PP).

Networks with Annotation Propagation: Through capturing provenance generation with two operator types, one for initial provenance generation (PG) and one for provenance propagation (PP), we have provided the necessary means to generate provenance for a complete (or parts of a) query network by

Provenance Generators

$$\sigma_c^{PG}(I) = \begin{cases} [new, H(I).\mathscr{N}, \{H(I)\}] \mid\mid \sigma_c^{PG}(T(I)) & \text{if } H(I) \models c \\ \sigma_c^{PG}(T(I)) & \text{else} \end{cases}$$

$$\pi_A^{PG}(I) = [new, H(I).A, \{H(I)\}] \mid\mid \pi_A^{PG}(T(I))$$

$$\alpha_{agg,\omega}^{PG}(I) = pa(\omega(I))$$
$$pa(I) = [new, agg_1(a_1, H(I)), \ldots, agg_n(a_n, H(I)), Set(H(I))] \mid\mid pa(T(I)$$

$$\bowtie_{c,j\omega}^{PG}(I,I') = pj(H(j\omega(I,I'))) \mid\rightarrow pj(T(j\omega(I,I')))$$
$$pj(jw) = pjl([\ll H(jw.w_1) \gg, jw.w_2]) \mid\rightarrow pj([T(jw.w_1), jw.w_2])$$
$$pjl(jw) = \begin{cases} [new, H(jw.w_1).\mathscr{N}, H(jw.w_2).\mathscr{N}, \\ \{H(jw.w_1), H(jw.w_2)\}] \mid\mid pjl([jw.w_1, T(jw.w_2)]) & \text{if } [H(jw.w_1), H(jw.w_2)] \models c \\ \\ pjl([jw.w_1, T(jw.w_2)]) & \text{else} \end{cases}$$

Figure 5: Provenance Generator Operator Types

Provenance Propagators

$$\sigma_c^{PP}(P) = \begin{cases} [new, H(P).\mathscr{N}, H(P).\mathscr{P}] \mid\mid \sigma_c^{PP}(T(P)) & \text{if } H(P).\mathscr{N} \models c \\ \sigma_c^{PP}(T(P)) & \text{else} \end{cases}$$

$$\pi_A^{PP}(I) = [new, H(I).A, H(I).\mathscr{P}] \mid\mid \pi_A^{PP}(T(I))$$

$$\alpha_{agg,\omega}^{PP}(I) = pa(\omega(I))$$
$$pa(I) = [new, agg_1(a_1, H(I)), \ldots, agg_n(a_n, H(I)), \bigcup_{i \in H(I)} i.\mathscr{P}] \mid\mid pa(T(I))$$

$$\bowtie_{c,j\omega}^{PP}(I,I') = pj(H(j\omega(I,I'))) \mid\rightarrow pj(T(j\omega(I,I')))$$
$$pj(jw) = pjl([\ll H(jw.w_1) \gg, jw.w_2]) \mid\rightarrow pj([T(jw.w_1), jw.w_2])$$
$$pjl(jw) = \begin{cases} [new, H(jw.w_1).\mathscr{N}, H(jw.w_2).\mathscr{N}, \\ H(jw.w_1).\mathscr{P} \cup H(jw.w_2).\mathscr{P}] \mid\mid pjl([jw.w_1, T(jw.w_2)]) & \text{if } [H(jw.w_1), H(jw.w_2)] \models c \\ \\ pjl([jw.w_1, T(jw.w_2)]) & \text{else} \end{cases}$$

Figure 6: Provenance Propagator Operator Types

deciding which operators are replaced by their annotating versions. That is, we can compute any valid PAS for that network. To create a PAS $P(O, \mathscr{I})$ for a network q we replace operators in the network with annotating operators using the *TransformNetwork* algorithm shown as Algorithm 1. Given a query network q, an stream O and a set of streams $\mathscr{I}$, the algorithms modifies q so that it generates $P(O, \mathscr{I})$. First we prepare the network to deal with operators that read from both streams in and not in $\mathscr{I}$. These operators are problematic, because they have to process both streams with and without provenance annotations and, thus, are neither PG nor PP operators. Our solution is to wrap each stream S in $\mathscr{I}$ that is connected to such an operator in a PG projection on all attributes in the schema of S. This projection does not change the results of the network, but guarantees that we can use solely PG and

Algorithm 1 TransformNetwork Algorithm

```
1:  procedure TRANSFORMNETWORK(q, O, 𝒥)
2:      mixed ← ∅
3:      for all o ∈ q do                                  ▷ Find operators with mixed usage
4:          if ∃S, S′ ∈ input(o) : S ∈ 𝒥 ∧ S′ ∉ 𝒥 then
5:              mixed ← mixed ∪ input(o)
6:          end if
7:      end for
8:      for all S ∈ (mixed ∩ 𝒥) do                        ▷ Add projection wrappers
9:          S ← Π_schema(S)(S)
10:     end for
11:     for all o ∈ q do                                  ▷ Replace operators
12:         if ∃S ∈ 𝒥 : HASPATH(S, o) ∧ HASPATH(o, O) then
13:             if ∃S′ ∈ input(o) : S′ ∈ 𝒥 then
14:                 o ← PG(o)
15:             else
16:                 o ← PP(o)
17:             end if
18:         end if
19:     end for
20: end procedure
```

PP operators to generate a PAS [1]. Afterwards, the algorithm iterates through all operators in the query network and replaces each operator that reads solely from streams in $\mathscr{I}$ with its PG version, and all remaining operators on paths between streams in $\mathscr{I}$ and O are replaced with their PP versions.

A side effect of the incremental one-operator-at-a-time provenance generation approach we use to modify a query network to generate a PAS $P(O, \mathscr{I})$ is that each PP operator in the modified network generates one or more PAS according to the subset of $\mathscr{I}$ its connected to. That means, additional PAS are generated for free by our approach. In our terminology we refer to PP operators as *p-sinks* (i.e., we can consume provenance from the output of such an operator) and PG operators as *p-hooks* (i.e., the inputs of these operator are the reference points for provenance generated by the annotating network). In the following, we use $P(q)$ (called *provenance generating network* or PNG) to denote a network that generates the PAS for all output streams of network q according to all input streams of q.

Example 4 *Two provenance generating versions of the example network are shown in Figure 7 (the operator parameters are omitted to simplify the representation). Figure 7(a) shows $P(q)$, i.e., the annotating version of q that generates the PAS $P(q, S_{out}, \{S_1, S_2\})$ for output stream S_{out} according to the 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 on in Figure 7(b) generates the PAS $P(q, 8, \{5\})$. Recall that a possible computation with data for this network is shown in Figure 4(b). The output stream of the right-most aggregation is annotated with provenance sets containing tuples of the b-sort operator's input stream.*

[1] Adding additional 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.

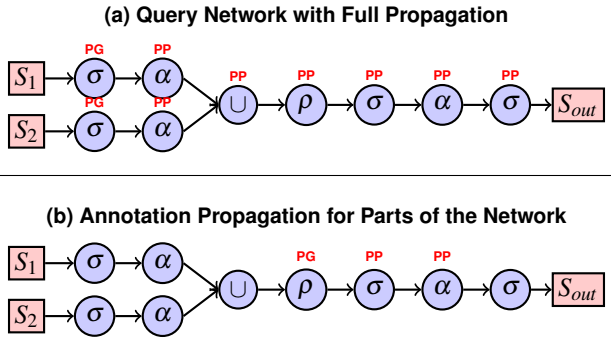


Figure 7: Annotating Query Networks

5 Provenance Propagation by Query Rewrite

We give a short overview of how the *Query Rewrite* approach is implemented in Ariadne. We followed the approach used in the Perm project [14] to compute database provenance. Let S_1 and S_2 be input streams of operators and S_{out} denote an operator output stream. Given a PAS $P(q, O, \mathscr{I})$ for a query network q, we have to transform q into a network that computes the PAS $P(O, \mathscr{I})$ using solely the standard operators of the DSMS. Since annotations are not part of the original data and query model, we have to fix a representation of annotations using regular data streams. Here, we use the same representation as used by the p-join and expand operators. We extend each data tuple t with additional attributes to be able to pair it with tuples from its provenance. For an input stream I, let $P(I)$ denote a stream following this representation that pairs every tuple t from I with itself. In a first step, we wrap every stream $S \in \mathscr{I}$ with an projection that duplicates the attributes of S, i.e. $S \in \mathscr{I} : S \rightarrow P(S) = \Pi_{P(S)}(S)$. Afterwards, we recursively apply the rewrite rules shown as graph patterns in Figure 8 [2] to all operators on paths between streams in $\mathscr{I}$ and O. Each of these rewrite rules transforms an operator into a new subnetwork that behaves like the PP-version of this operator, except that it produces a regular data stream encoding of an annotated stream instead of an annotated stream.

Figure 9 shows an example network with two aggregation operators and the rewritten variant of this network. The aggregations are rewritten by joining their outputs with PAS for their inputs. Note that this rewrite is only possible for certain types of window operators where we can express a join condition that guarantees that each tuple from a window only joins with the aggregated output produced for this window. For instance, for a value-based window function $\mathscr{V}(c, s, a)$, we add two additional aggregation functions to compute the minimum and maximum values in 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 Implementation

6.1 Overview

We now present the implementation of the approach developed above in our prototype system *Ariadne*. While annotating networks give us the means to correctly describe the generation of provenance anno-

[2] Note that we abuse the PAS notation to also denote the PAS representation as a data stream.

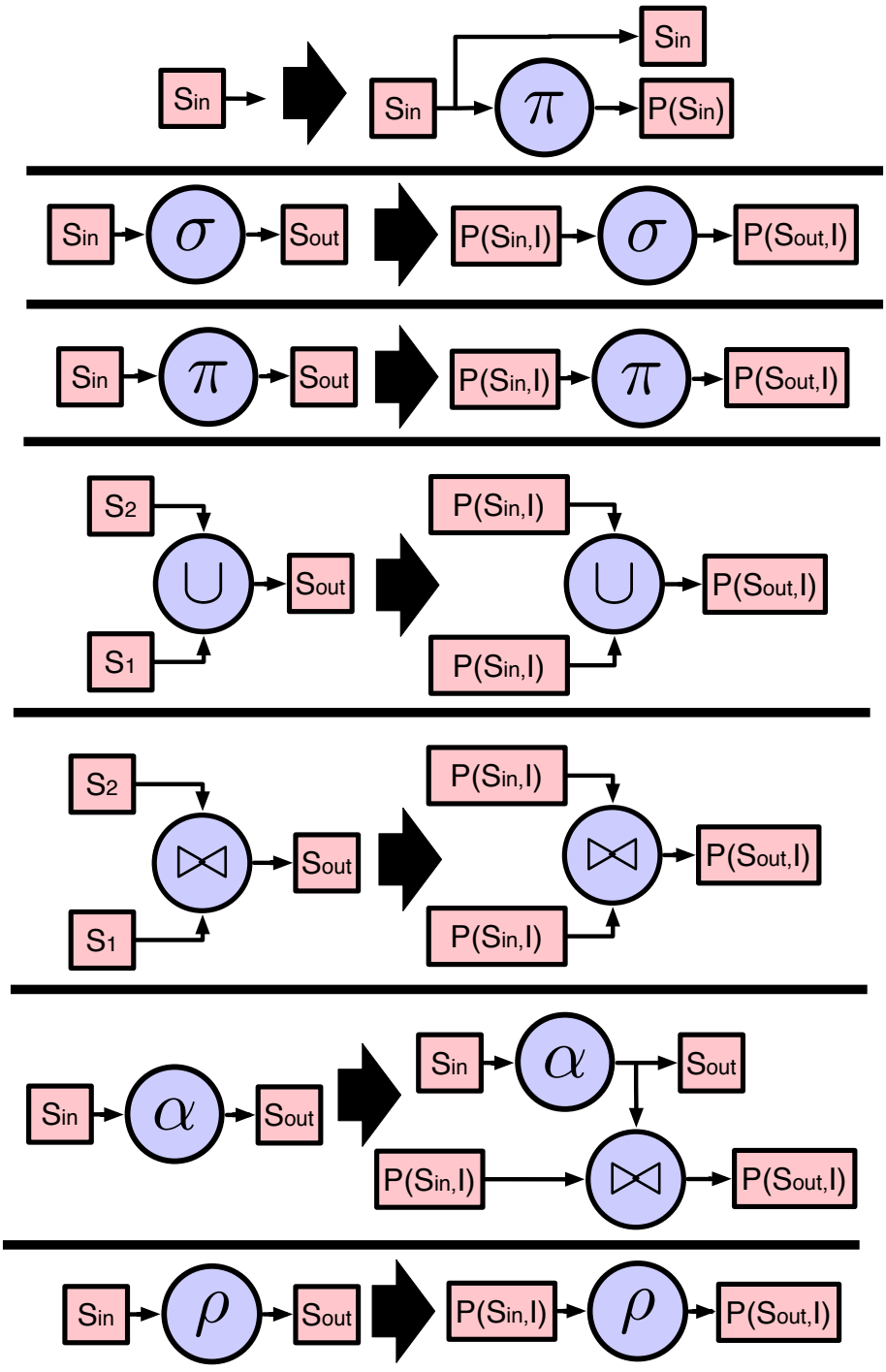


Figure 8: Query Rewrite Rules

tated streams for a given query network, several problems had to be overcome in the implementation to achieve efficient computation and retrieval of provenance.

We choose to implement the annotating operators introduced in the last section using *Reduced-Eager Operator Instrumentation*, because this approach can be applied to all types of query networks and is

expected to result in a lower overhead than pure eager approaches. For operator instrumentation, we extend each operator's implementation by introducing two new operational modes that implement the PG and PP versions of that operator. A query network q is set up for provenance computation by setting the operational modes as presented in Section 4.4, thus providing the flexibility to (re-)configure provenance computation for parts of the network.

We implement *Reduced-Eager* by letting the annotating operators generate provenance as sets of tuple identifiers (*TID-Set*) instead of sets of complete input tuples. To be able to restore the input tuples that correspond to the tuple identifiers in the provenance for retrieval, we (1) temporarily preserve the input tuples of *p-hooks* and (2) introduce two new operators called *expand* and *p-join*. The expand operator turns a tuple annotated with a TID-Set representing its provenance into duplicates of this tuple each paired with a single TID. The output tuples are regular data tuples and, thus, can be further processed using the standard operators of the system. The second operator, p-join, restores complete input tuples for provenance retrieval by joining a PAS with preserved input tuples from one of its p-hooks. For a tuple t annotated with a TID-Set, p-join retrieves the preserved input tuples that correspond to the TIDs in the TID-Set and outputs each combination of t with one of the retrieved input tuples. We store input tuples as long as they may be required for provenance retrieval using a Borealis feature called *Connection Points* [24]. Our current approach uses time-based purging of input tuples (standard purging strategy of connection points). This solution is sufficient for most use-cases, but can be replaced with more elaborate strategies in the future. Our variant of the *Reduced-Eager* approach provides both efficiency and flexibility for provenance operations. Using TID-Sets avoids the cost of propagating full tuples and enables further optimizations which we present in Section 7. Using the original data model to represent provenance information by duplicating the data tuples is simpler and more general than an extended data model and enables the user to express complex queries over the relationship between data and its provenance. Developing specialized operators that evaluate queries directly over the TID-Set representation of provenance is an interesting avenue for future work.

6.2 Internal Representation and Serialization of Provenance

We now discuss how Ariadne represents provenance internally and how it serializes and passes provenance information between operators. Borealis uses queues to pass fixed-length tuples as uninterpreted chunks of memory between operators in a query network. The physical layout of a tuple is shown in Figure 10(a). A tuple consists of a fixed length header (H bytes) storing information such as TID (T bytes)

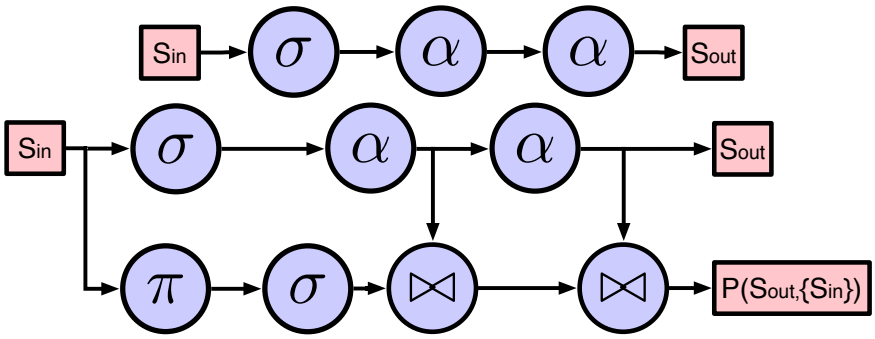


Figure 9: Query Rewrite Example

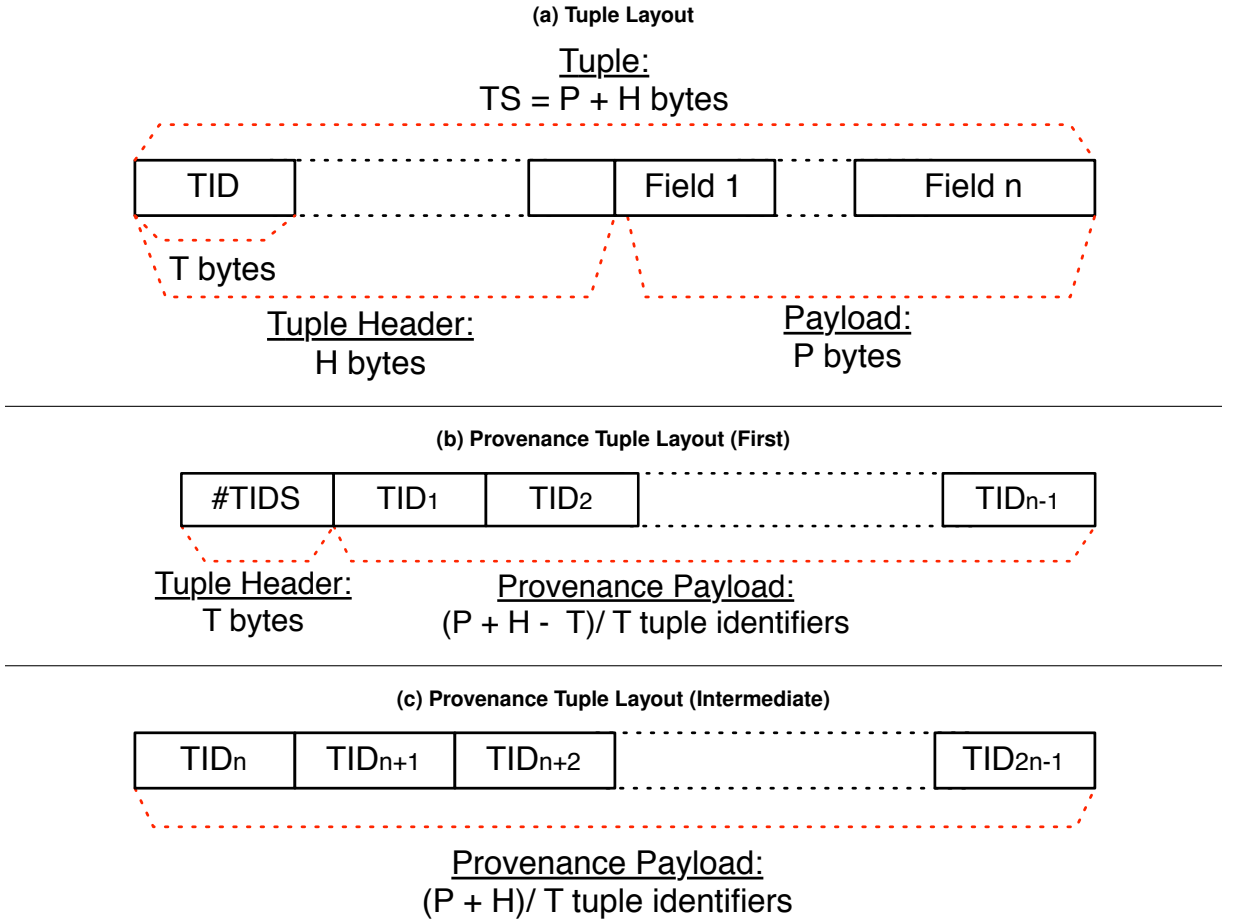


Figure 10: Physical Tuple Layout

and arrival time, and a payload which stores the raw byte data of the tuple's attribute values (P bytes with P depending on the tuple's schema). No schema information is stored in the tuple itself. The schema of the stream the tuple belongs to is needed to interpret the payload information.

Example 5 *The execution of part of the running example network (from the b-sort to the right-most aggregation) is shown on the top of Figure 11(a). For each tuple, the header is highlighted in light green and the payload in dark green. The size of an element in a tuple shown in the figure represents the number of bytes occupied by this element. Note that the sizes assumed here do not correspond one to one to the sizes used in the actual implementation, but were chosen for presentational purposes. For instance, the size of a tuple header used by Borealis is larger than shown in the figure.*

We considered three alternatives to pass the 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 Borealis queues. We chose the third approach, because it is less intrusive than changing all code that depends on fixed-length or introducing a new information passing mechanism. Furthermore, we benefit from several optimizations in the engine that rely on tuples of fixed-size.

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 10(b) and 10(c) show the physical layout of such tuples. The first tuple (Figure 10(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 down-stream operators to determine how many provenance tuples have to be dequeued. Given that the size of a TID in Borealis is 8 bytes[3], we are saving around an order of magnitude of space (and number of tuples propagated) compared to using full tuples even for streams with a very small tuple payload. The actual savings depend on the number and type of payload data fields of the stream and the underlying architecture: Recall that H is the size of a standard Borealis tuple header, T the size of a TID value, and P the size of the payload data for tuples in a queue. Thus, the size TS of a complete tuple in the queue would be $P+H$. To serialize a TID-Set, we can store $\lfloor TS/T \rfloor$ TIDs in a tuple of size TS with exception of the first tuple in the serialization which stores $\lfloor (TS-T)/T \rfloor$ TIDs and the provenance header. Despite these savings, the TID-Set representation can still cause significant overhead due to the large amount of provenance certain operations (e.g. aggregations on large windows) can create. We investigate compression methods to further reduce this overhead in Section 7.

6.3 Provenance Annotating Operator Modes

As outlined in the beginning of this section, we extend the existing Borealis operators with new operational modes to realize the provenance generating (PG) and provenance propagating (PP) operators introduced in Section 4.4. Operators in both PG- and PP-mode need to be able to serialize a provenance set as outlined in Section 6.2. An operator in PG-mode has to buffer TIDs from its input stream and use these TIDs to generate provenance sets for its output. PP-mode requires the ability to de-serialize and/or buffer provenance sets from the operators input and to merge provenance sets.

We factored out the common functionality such as buffering, serialization and de-serialization, and merging of TID-Sets to limit the changes to the original operator implementations. This common part, which we refer to as *provenance wrapper*, provides the following functionalities:

- Encapsulate reading from input queues: The provenance wrapper encapsulates access to the *dequeue* method of Borealis tuple queues. This method is used by operators to read tuples from their input queues. The original implementation of this method removes a tuple from the beginning of a queue and returns it to the caller. If the input queue is a PAS then the provenance wrapper also dequeues all provenance tuples following the data tuple that was read, deserializes the TID-Set stored in these provenance tuples if necessary, and buffers the TID-Set for output provenance generation. From the operators point of view, the wrapped method behaves in the same way as the regular tuple queue dequeue method.
- Encapsulate writing to output queues: The *enqueue* method of a tuple queue is used by operators to append their outputs to the end of a tuple queue. In addition to sending a data tuple, the provenance wrapper constructs the tuple's provenance from the currently buffered input provenance and serializes it over the output queue. Afterwards, obsolete input provenance information is discarded from the buffer.

Whenever the Borealis framework signals to an operator the arrival of new tuples in one of its input

[3]To be precise, the size the C++ compiler allocates for the *signed long* data type.

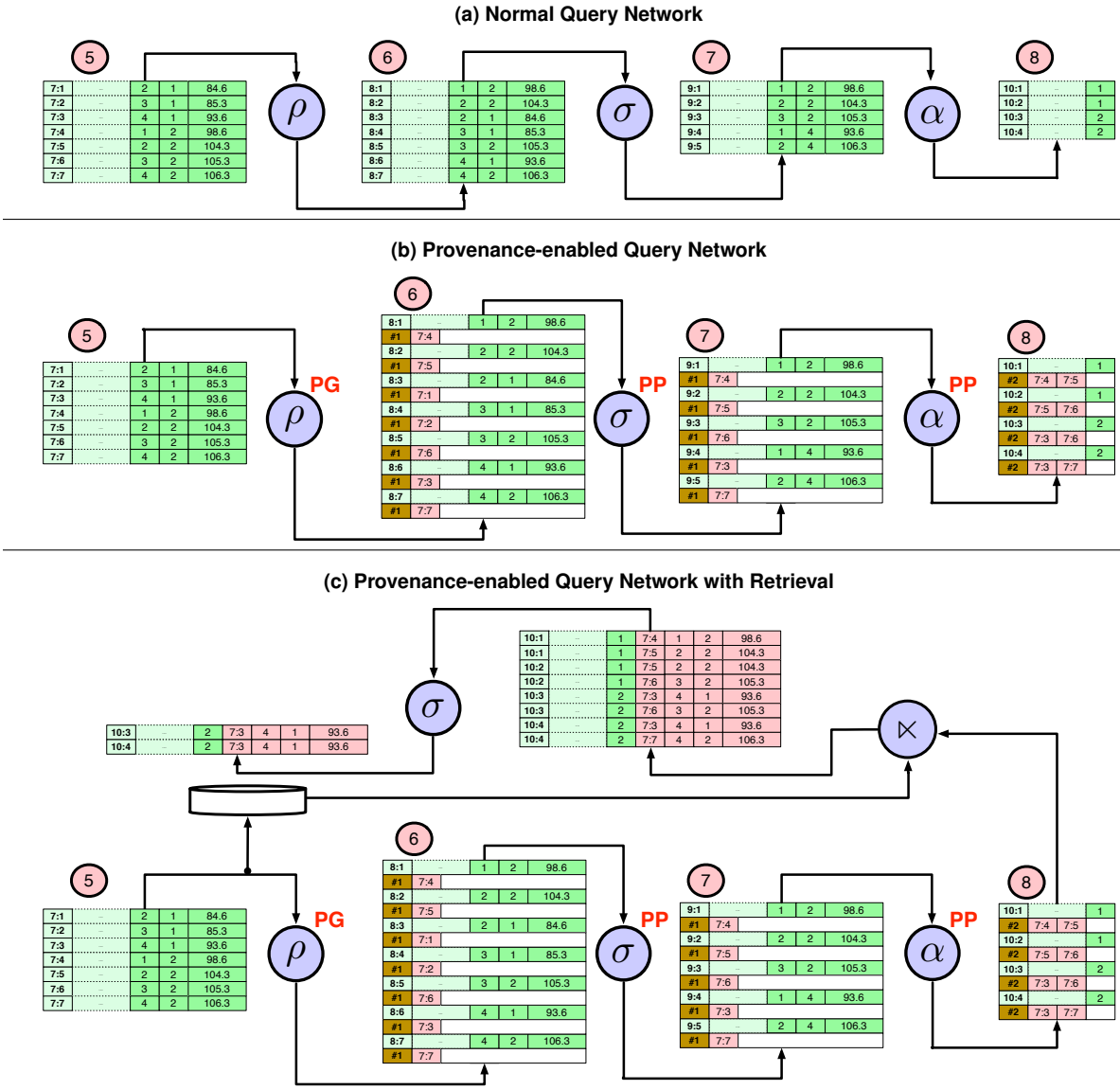


Figure 11: Example for Provenance Computation

queues (Borealis uses a push model), the operator retrieves these tuples by calling the provenance wrapper's *dequeue()* method. This causes the wrapper to update its buffer by adding the provenance of newly arrived data tuple(s) (or their TIDs if the operator runs in PG-mode). Afterwards, the operator performs its regular computations on the data tuples, which may produce output data tuples. These tuples are pushed towards the output queue using the wrapper's *enqueue()* method which appends the output data and combined buffered provenance information to the output queue. Using the provenance wrapper, the annotating versions of most operators can be implemented in a few lines of code. An exception is aggregation that requires some additional bookkeeping to record the relationship between windows and input tuples.

Example 6 *Consider the provenance computation for the annotating network from Figure 7(b) shown*

in Figure 11(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. For instance, each data tuple in the output of the b-sort is followed by a provenance tuple storing the single TID (#1) in the provenance of the data tuple (compare Figure 4(b)). 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 data tuple 10:1. Note that in this example we assumed that the tuples in stream 8 are large enough to store four TIDs (or three TIDs and a provenance header) In reality, we can store many more TIDs in a single tuple.

6.4 Input Storage and Retrieval

As mentioned before, we apply a *Reduced-Eager* approach which requires preservation of input tuples at p-hooks 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 two new operators (expand and p-join) for transforming TID-Sets into a queryable format.

Input Storage at P-hooks: Connection points (CP) were introduced by Ryvkina et al. [24] as a supporting technology for revision processing in Borealis. A connection point acts as temporary storage for the tuples that pass though the queue the CP is attached to. CPs support different strategies for removing old tuples from storage. For example, one strategy is to preserve tuples for a given time interval. We use CPs to store and look-up input tuples from p-hooks. For a query network q instrumented to compute a PAS $P(O, \mathscr{I})$ we add a connection point to each stream in $\mathscr{I}$, i.e., the streams that are inputs of provenance generators (p-hooks). This guarantees that we have the necessary information available to restore complete input tuples from the TIDs in a TID-Set. Exploring more sophisticated (and distributed) storage technologies is an interesting avenue for future work.

Expand: An *expand* operator $\varepsilon(S)$ transforms each input tuple t of PAS S followed by its serialized TID-Set into duplicates of t with a single TID from the set attached to each duplicate (stored in an additional attribute called *tid*). A provenance wrapper is used to read all provenance tuples for an input tuple. The output of an expand can, for example, be joined with input streams attached to p-hooks to combine tuples with full tuples from their provenance.

P-join: In addition we developed a new operator called p-join. A p-join $\ltimes(S, CP)$ joins a TID-carrying stream S with a connection point CP and, thus, outputs tuples extended with complete input tuples from their provenance. The output of such a join is used to represent provenance as regular tuples to a user and can be queried using the operators of the DSMS. P-joins provide a more efficient way to combine tuples with tuples from their provenance than expand plus a regular join, because (1) a p-join can use a fast hash-based TID look-up from a CP to determine which tuples to join with an input tuple based on their TID-Set instead of using a regular join with an input stream and (2) it avoids the materialization of one intermediate queue (the output of the expand operator). However, p-join requires the CP and p-join to be one the same node. Thus, expand may be useful if the query network is split over several processing nodes.

Example 7 *The running example network with retrieval is shown in Figure 11(c). Recall that this network was instrumented to generated $P(q, 8, \{5\})$. Hence, a CP (the cylinder) is used to preserve tuples of 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. For example, consider tuple 10:1 from stream 8*. The TID-Set for this tuple contains two TIDs 7:4 and 7:5 stored in one provenance tuple. After reading tuple 10:1, the p-join operator reads and de-serializes the provenance tuple using a provenance wrapper. For each TID in the set the corresponding input tuple* t *is retrieved from the connection point and the concatenation of tuples 10:1 and* t *is outputted. The stream produced by the p-join can then 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 11(c). The representation returned by the p-join operator is expressive enough to be used in complex queries that access both data and its provenance. For instance, the user may proceed by limiting the investigation to alarms that were caused by at least* n *slightly overheated sensors using an aggregation and filter on top of the example query.*

6.5 Implementing Query Rewrite

As mention in Section 3.4, we also implemented eager propagation through query rewrite to be able to compare the performance of this technique with our approach. In Section 5 we gave a brief overview of the rewrite approach. The major difference to Perm is that a separate rewrite rule for aggregation is needed for each type of windowing operator. An example of a rewritten network is shown in Section 8.1, Figure 16(b).

7 Optimizations

By its very definition, fine-grained provenance potentially references a significantly larger set of data than the result generated by normal query processing. This additional data can cause significant cost in computation and storage, as observed in many database or workflow provenance systems. Providing provenance for data stream systems further aggravates this problem, mainly for two reasons:

1. Typical DSMS workloads rely heavily on aggregation to reduce downstream workload, reducing many input tuples to few output tuples. When requesting provenance for such aggregation queries, these savings are partially negated, as all input tuples from a window are part of the provenance for the output tuple produced for this window.
2. Stream processing systems treat data as transient, computing results on the fly, and discard data as soon as possible to keep up with high input data rates. Thus, also the provenance generation has to be performed on the fly, possibly wasting significant resources if this provenance is never requested.

We address these problems by developing concise representations of provenance (to reduce the overhead of provenance generation and number of queue operations to transfer provenance) as well as enabling on-demand provenance computation and retrieval using lazy approaches (to avoid unnecessary provenance generation).

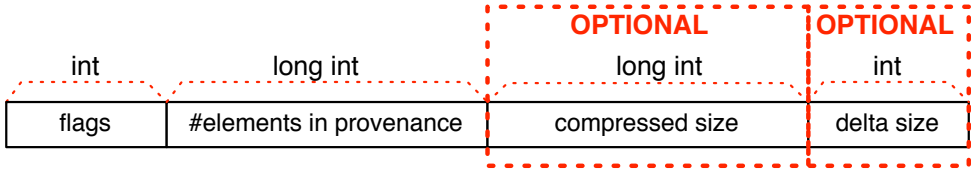
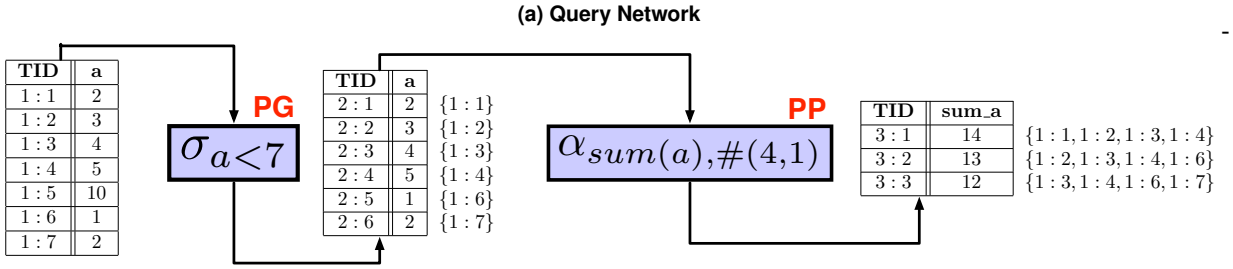


Figure 12: Provenance Tuple Header



(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 13: Compression Techniques Example

7.1 Provenance Compression

We now study a number of methods for efficient TID-Set compression, ranging from generic data compression to methods which exploit data model and operator characteristics. An important consideration is the balance between the cost needed to perform compression/decompression operations and the savings achieved. In particular, choosing compression methods that do not require decompression for most operators is important to avoid the cost of repeatedly compressing and decompressing provenance at each operator. As we will see, the proposed compression methods work best under certain workload characteristics, necessitating an adaptive combination of them to provide effective processing for an arbitrary and changing workload. Thus, a provenance wrapper may read TID-Sets from its input that are represented and compressed in different ways. We extended the provenance header structure sent

with the first tuple of a serialized TID-Set to record which compression was applied to the TID-Set. The structure of this header is shown in Figure 12. A header consists of an *int* storing flags (e.g., which compression method is used), the number of elements in the provenance, and two optional parts that are used by specific compression methods and will be explained later.

Example 8 *Figure 13(a) shows an example query network that filters tuples with a-attribute values above seven, groups the result into windows of four tuples, and computes the sum of attribute a for each window. Figure 13(b) shows the provenance in the PAS generated by the aggregation operator using different compression techniques. We leave out the stream identifier part of the TIDs to simplify the representation. For example, the standard TID-Set representation is shown as no compression. Each provenance tuple has a header storing the value of the flags for this tuple (set to "S" for single TIDs) and the number of TIDs in the TID-Set.*

7.1.1 Interval Encoding

Interval encoding exploits the observation that the provenance of a window consists of the provenance of the tuples in the window which form contiguous sub-sequences of the input sequence. Instead of storing each element of a TID-Set individually, this method encodes a TID-Set as a list of intervals spanning continuous sequences of TID values in the set. An interval can be represented by its start and end points. Hence, a single interval occupies twice the space of a single TID. For interval encoding the flags in the provenance header are set to I and the second field of the header is used to store the number of intervals in the provenance. For example, consider the interval encoding for the example shown in Figure 13(b). The provenance of tuple 3:1 is represented as a single interval $[1,4]$, because the TID-Set is a single contiguous sequence of TIDs (1 to 4). Interval encoding does not guarantee a smaller representation than no compression. For instance, the provenance of tuples 3:2 and 3:3 consists of two sequences of length two and, thus, requires the same amount of storage with interval encoding and no compression.

Under interval encoding, operators in PG-mode group contiguous TIDs from the inputs into intervals; operators in PP-mode try to merge intervals they read from their input. The merging can be efficiently done using existing well-known interval merging techniques [17]. Interval encoding is most advantageous for queries involving aggregations over a long sequence of contiguous TIDs. Such a sequence can be represented as a single interval. In the worst-case there are no contiguous sequences of TIDs in the input and, thus, each interval stores only a single TID resulting in twice the storage of no compression (e.g., only projection and filter operators).

7.1.2 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). Delta encoding is similar to incremental backup techniques. Repeatedly we send a tuple with full provenance followed by several tuples with their provenance encoded as a delta to the last full provenance that was sent. While this approach has a potentially higher space overhead, we can restore a TID-Set from its delta representation in a single step without the need to apply and

Parameter	Compression Method	Description
α	Interval	Minimum factor of size reduction with respect to simple TID-Set representation
β	Delta	Minimum overlap factor between delta and last full provenance
γ	Delta	Maximum number of deltas in sequence
δ	Delta	Minimum size (number of tuples) of a TID-Set
ε	LZ77	Minimum size in bytes for compression

Figure 14: Compression Thresholds

store a long chain of deltas.

Example 9 *To illustrate the mechanism and benefits of delta encoding, consider how delta encoding handles the example from Figure 13(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 aggregation is sent without applying delta encoding (*0 *overlap). The TID-Set of tuple 10:2 shares three TIDs with the TID-Set of the previous tuple. It is encoded as a delta storing the overlap (*3*) and the single additional TID (*6*). The provenance of the third tuple has two TIDs overlap with the provenance of tuple 3:1. Thus, the delta representation of this TID-Set is 2:*$\{6,7\}$.

Delta encoding requires additional bookkeeping at the enqueue part of a provenance wrapper. We need to store the last complete TID-Set that was sent and the number of deltas applied to it. Operators in PP-mode which read from a stream containing delta encoded provenance may need to reconstruct TID-Sets from the delta representation. For example, an aggregation in PP-mode has to reconstruct the input TID-Sets to be able to merge the provenance for a window of tuples. For projection we can simply pass on delta compressed provenance without the need for reconstruction. The same applies for selection except that selection may filter out a tuple with a complete TID-Set which would leave the deltas following this TID-Set orphaned. We currently handle this situation by reconstructing the first delta and adapting the following deltas.

7.1.3 Dictionary Compression

Given their potential size, TID-Sets and also collections of intervals can be compressed using standard compression techniques. We used LZ77 as it deals with flexible input sizes (not a block-based compression) and provides a good tradeoff between speed and effectiveness, but other methods are certainly possible as well. Compression is only activated if the size of a TID-Set or interval collection exceeds a fixed threshold to avoid paying the price of compression for a small number of values, e.g., if a TID-Set can be serialized as a single provenance tuple there is no potential gain in compressing the provenance. If dictionary compression is applied, this is indicated in the flags of the provenance header sent in the first tuple of the set. In addition the provenance header stores the compressed size of the TID-Sets (see Figure 12). This type of compression can reduce the load on the queues significantly for large TID-Sets at the cost of additional processing to compress and decompress the TID-Sets. Since generic compression methods do not take advantage of the specifics of the data model and operator semantics, it performs worse for provenance in DSMS than interval or delta encoding. As experiments demonstrate, significant space savings are possible, but the computational overhead is typically offsetting these savings.

Algorithm 2 Adaptive Compression Algorithm

```
1: fullP ← ∅                                                    ▷ Static Initialization
2: numDelta ← 0
3: tupleSize ← |S|

4: procedure CompressProv(TID-Set P, OutputStream O)
5:     numTuples ← ⌈tupleSize/byteSize(P)⌉
6:     if fullP = ∅ ∨ numDelta > γ ∨ numTuples < δ
7:         ∨ overlap(P, fullP) < β then                         ▷ Send full provenance?
8:         I ← interval(P)
9:         if byteSize(I)/byteSize(P) < α then
10:            P ← I
11:        end if
12:        numDelta ← 0
13:        fullP ← P
14:    else                                                     ▷ Send delta
15:        if isInterval(fullP) then
16:            P ← interval(P)
17:        end if
18:        numDelta ← numDelta + 1
19:        P ← delta(fullP, P)
20:    end if
21:    if byteSize(P) > ε then                                  ▷ Apply LZ77 compression?
22:        P ← LZ77(P)
23:    end if
24:    O.enqueue(P)
25: end procedure
```

7.1.4 Adaptive Combination of Compression Techniques

We now discuss how to combine the compression techniques presented in this section by using a set of heuristic rules that determine when to apply which type of compression. Interval encoding, dictionary compression, and delta encoding can be freely combined. For instance, we could apply delta encoding to intervals and apply LZ77 compression to the resulting deltas. We use the algorithm `CompressProv` (Algorithm 2) to decide how to compress an incoming TID-Set. The algorithm uses several threshold parameters shown in Figure 14. The general approach is to avoid compression if it would not result in reasonable space savings or would lead to long sequences of deltas. The algorithm buffers the last complete (non-delta) TID-Set that was sent (*fullP*) and how many deltas have been sent after sending the last full TID-Set (*numDelta*).

For an input TID-Set P we first determine if it should be sent fully or as a delta. The TID-Set P is sent completely if at least one of the following conditions holds: (1) We need less than threshold δ number of tuples to serialize P. This condition guarantees that we do not pay the overhead for delta encoding if the TID-Set is small. For example, if P can be stored in a single tuple, then there is no need to apply delta compression. (2) Less then threshold γ number of deltas have been sent after sending the last full TID-Set. This condition limits the number of deltas that depend on a full TID-Set. (3) The overlap between

P and the last full TID-Set is at least of factor β. That is the delta representation of P is significantly smaller than P. If P is sent completely, then the algorithm determines if compressing it using interval encoding would save space. If the interval encoded version of P is smaller than the original P by a factor α, then P is compressed using interval encoding. Afterwards, we update *fullP* and reset *numDelta*. If P is sent as a delta then it is interval encoded only if *fullP* is also interval encoded. Dictionary compression is considered for both delta encoded and full TID-Sets. If the compressed version of P generated by the first part of the algorithm occupies more than threshold ε bytes, then P is compressed using LZ77 compression. The final result of this process is then serialized to the output stream O.

7.2 On-Demand Provenance Operations

In this section we introduce two types of optimizations that increase performance by avoiding provenance generation or reconstruction where possible. Our *Reduced-Eager* approach for provenance generation reduces the runtime overhead in comparison with eager at the cost of tuple reconstruction when retrieving provenance. For use cases where the result of the reconstruction is further processed by the query network (queries over provenance information) it is possible to partially avoid the cost of reconstruction by filtering out parts of the provenance that will not be used by the query (as long as we can determine this upfront). We call this approach *Lazy Retrieval*. As a possible optimization for low retrieval frequencies we also introduce a variant of *Replay-Lazy*.

7.2.1 Lazy Retrieval

Using *Reduced-Eager* we separate the computation of provenance (TID-Sets) from the generation of a representation that is useful for retrieval. We therefore have the opportunity to avoid the cost of reconstruction through a p-join or expand operator if we can determine that parts of the provenance are not needed to answer a retrieval query. To this end we try to push selections that are applied during retrieval of provenance through the reconstruction (p-join) based on the algebraic equivalence presented below. For a p-join between a PAS S and connection point CP, and a selection with condition c the following holds iff c only accesses attributes from $schema(S)$[4]: $\sigma_c(S \ltimes CP) \equiv \sigma_c(S) \ltimes CP$

7.2.2 Replay-Lazy

Recall from Section 3.2 that a *Replay-Lazy* approach computes provenance by replaying parts of the input through a provenance generating network. *Replay-Lazy* can be advantageous if provenance retrieval is infrequent, because it almost completely avoids runtime overhead (and significantly reduces latency) for the original query network. However, unless additional information is kept, the whole input of the query network has to be replayed through a provenance generating network until the output of interest is produced. This can be avoided if we determine, during query execution, which parts of the input have to be replayed. For the deterministic operators of our algebra it can be shown that it is sufficient to replay all input tuples from the interval spanned by the smallest and largest TID in the provenance of an output tuple (we refer to this interval as the *covering interval* of an TID-Set). *Replay-Lazy* in Ariadne is based on this observation.

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

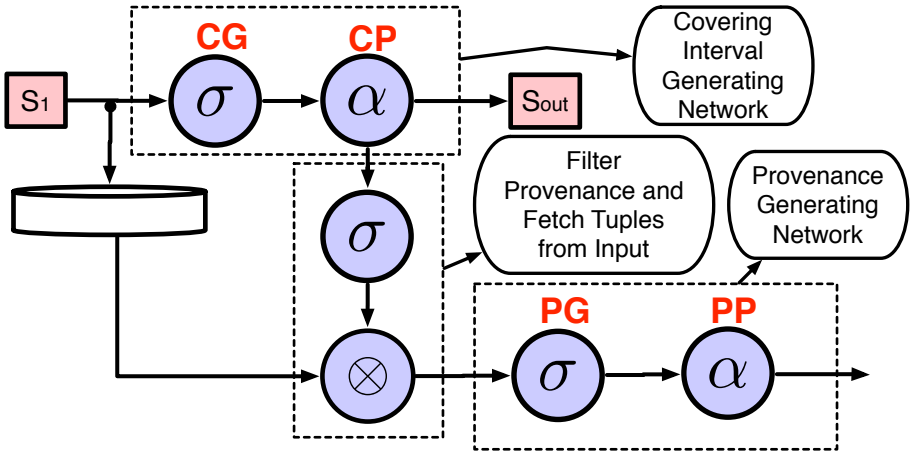


Figure 15: Example for a Replay-Lazy Network

The network is instrumented in the same way as for *Reduced-Eager* with the exception that we annotate each output tuple with the minimal interval that covers all TIDs in the tuple's provenance. Given that these intervals need constant size, we do not need to use separate tuples for provenance serialization, but can piggyback the intervals on data tuples, thereby significantly reducing load on tuple queues. Covering intervals can be generated very efficiently during operator execution. For example, to combine the covering intervals for an aggregation output we simply have to find the minimum and maximum TID values in the intervals attached to the tuples in the window.

To generate the provenance from a stream with covering intervals we introduce an new join operator (*c-join*). 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 outputting these tuples. For Replay-Lazy these tuples are fed into a copy of the query network that is instrumented for provenance generation. Similar to *Reduced-Eager* we can apply selection on data attributes beforehand to reduce the load on the provenance generating network. The c-join operator sends a control tuple after sending the last tuple of a covering interval. This control tuple instructs downstream operators to drop their internal state (e.g., open windows) and flush their buffers (b-sort). This is necessary to produce correct results and reduce the number of spurious tuples in the output.

For example, the tuples in the covering interval for a tuple t may also belong to the covering intervals of other tuples. Thus, replaying these tuples may produce other tuples in addition to t. As an example for why dropping internal operator state after each covering interval is necessary consider an aggregation with count-based windowing of size 3 and slide 2. Assume the tuples from the covering intervals $[1,3]$ and $[5,7]$ have been chosen for replay. Replaying these tuples through the aggregation causes the following windows to be created: $<1,2,3>$ and $<3,5,6>$ while the original computation generated windows $<1,2,3>$, $<3,4,5>$, and $<5,6,7>$. Note that we can avoid dropping the operator state if the covering intervals of consecutive tuples overlap. In this case the c-join does not send the control tuple and sends tuples for reoccurring TIDs only once. For instance, if the covering intervals in the example above would have been $[1,3]$ and $[3,5]$, the c-join would output tuples 1 to 5.

Replay-Lazy has the advantage that provenance generation is almost completely separated from normal query processing. Thus, to scale out, the provenance generating part of the network can be executed on a different Borealis node keeping the performance impact on normal processing at a minimum.

Example 10 *Consider the covering intervals shown in Figure 13(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 15 shows how to instrument the query network from Figure 13(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 that are in the covering interval of an input tuple from the connection point and route these tuples through a provenance generating copy of the query network to produce provenance for tuples of interest.*

8 Experiments

The goal of our experimental evaluation is to investigate the overhead of provenance management with *Ariadne*, compare with competing approaches (*Rewrite*), and study the benefits of optimizations proposed in Section 7.

8.1 Overview

Following the discussion of approaches in Section 3, we evaluate the following methods for *how* to generate provenance: *Operator Instrumentation* as the expected best option with query *Rewrite* as the current state-of-the-art. We do not consider *Inversion*, since it is only applicable to a very limited class of query networks and provenance retrieval scenarios. For *Operator Instrumentation*, we compare the *Reduced-Eager* vs. *Replay-Lazy* options for generating provenance. *Lazy Retrieval* will be studied as an optimization for both rewrite and instrumentation. We also evaluate the impact of compression on the provenance generation (as outlined in Section 7.1)

8.1.1 Query Networks

We use the following query networks in our experiments: The **Basic** network is modeled after a linear prefix of our running example (Figure 4(a)). As shown in Figure 16(a), it consists of a selection, followed by an aggregate using count-based windowing and then another selection. This query covers the most critical operator for provenance management (aggregation) and is simple enough to study individual cost drivers. As is turns out, this query is also one of the best (non-trivial) cases for *Rewrite* (Figure 16(b)). For convenience, we also show the rewritten (Figure 16(b)), instrumented (Figure 16(c)), and Replay-Lazy version of this query (Figure 16(d)). In experiments that focus on the cost of provenance generation, we leave out parts of these networks that implement retrieval (the dashed boxes). The **Nested** network (Figure 16(e)) is a variation of the *Basic* network with additional aggregations. In the experiment using this network we increase the number of aggregations to exponentially increasing the amount of provenance per result tuple. The **Complex**, real-life network, which represents the complete running example introduced in Figure 4a. We use this network to study how the results for the *Basic* and *Nested* networks translate to a larger set of operators and a more complex query graph.

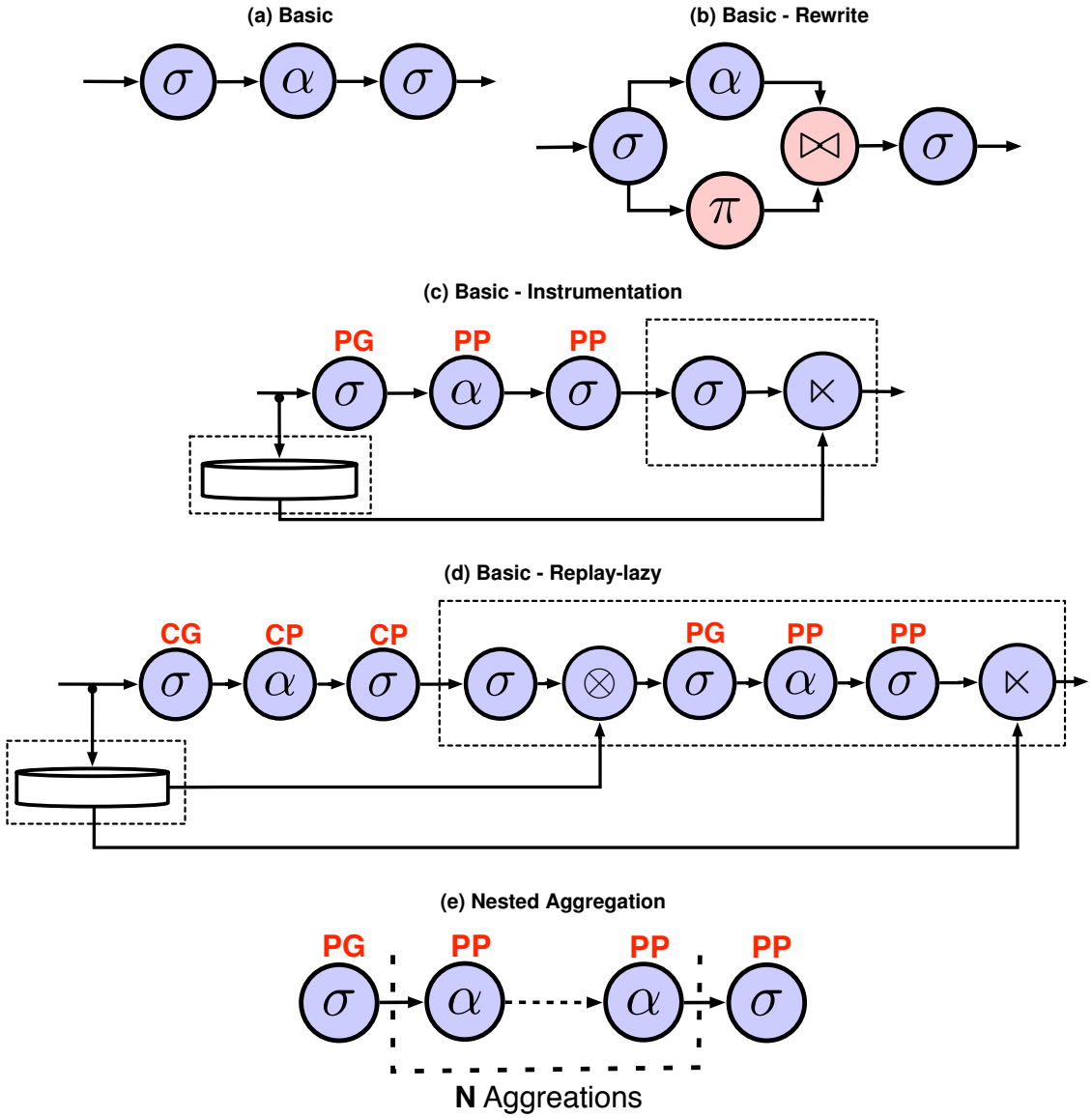


Figure 16: Experiment Queries

8.1.2 Setup and Methodology

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 workloads sent by the clients were generated beforehand. We consider two approaches for pushing the inputs from the client to the server:

Maximum Load: The client sends its input in very large batches (100K tuples). The server starts processing once a batch has fully arrived, i.e., the entire data set is available at once for processing. We used this setup, because preliminary experiments indicated that the Borealis code that is responsible for retrieving inputs from the client is a major performance bottleneck and, thus, covered up the overhead

introduced by provenance generation. Furthermore, maximum load allows us to identify the worst-case overhead introduced by provenance operations.

Varying Load: Running the query network with different load to determine the minimum overhead that always has to be paid. Load variations are achieved by varying batch sizes and delays between batches.

Since the overhead of unused provenance code turned out to be negligible, we used *Ariadne* also for experiments without provenance generation. Each experiment was repeated 10 times to minimize the impact of random effects. We show the standard deviation where possible in the graphs. The following measures were captured in the experiments:

Completion Time, measuring the time (and thus CPU cost) required to fully process a workload at *maximum load*. Completion time is the difference between the arrival timestamp of the first input tuple and the leaving timestamp of the last output tuple.

Tuple Latency, measuring the delay that each tuple incurs while being processed. Latency is the difference between the departure time of a tuple and the arrival time of the last tuple that contributed to it.

Queue Memory, measuring the amount of data accumulated in tuple queues at a point in time (i.e., data that has been sent by one operator but has not yet been processed by downstream operators).

Provenance Structure Memory, measuring the total additional state of all provenance-enabled operators. We added a thread for memory measurements that records the sizes of all queues and provenance structures (number of bytes) every 100 milliseconds.

8.2 Fundamental Tradeoffs

In the first set of experiments, we study the cost of different methods for provenance generation and retrieval. We use the *Basic* query network with *Maximum Load* for all methods (plus no provenance generation, as a reference point), but do not consider provenance compression yet (i.e., we use *Single*). The result for each method is broken down into the time to generate provenance (*Generation*, by removing the operators in the dashed box) and the time to reconstruct the generated provenance for retrieval (*Retrieval*). These details are not provided for rewrite, because rewrite is eager (there is only one phase).

8.2.1 End to End Cost

Experiment 1 (shown in Figure 17) compares the end-to-end cost when *changing the amount of provenance* 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 to 12. Even for this relatively benign workload, *Rewrite* shows much worse scaling than unoptimized *Instrumentation* with full *Retrieval*: while roughly on par for WS=10, it uses twice as much time on WS=100. A significant amount of cost when using *Instrumentation* comes from Retrieval: around 40 percent at WS=10, and around 65 percent at WS=100. This cost is roughly linear to the amount of provenance produced. The overhead of provenance gener-

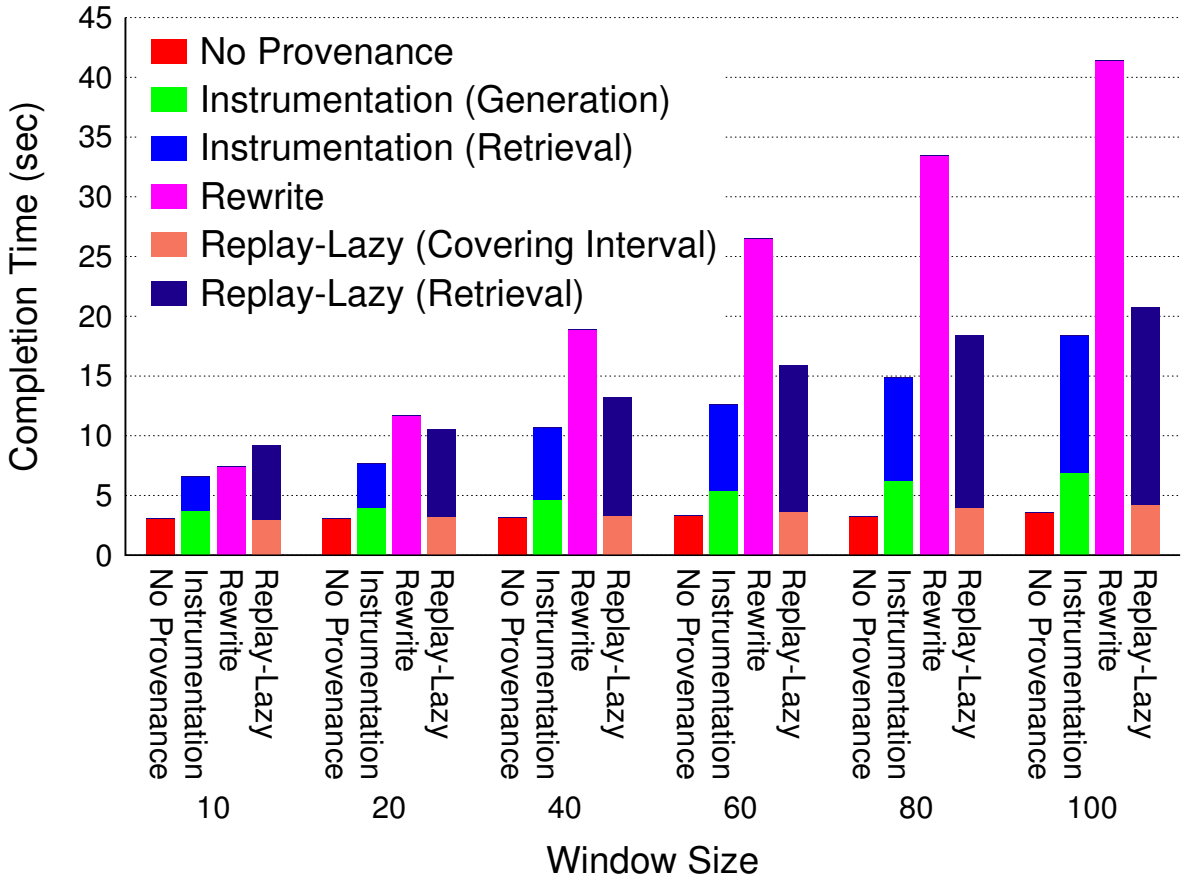


Figure 17: End to End - Varying Amount of Provenance

ation through Instrumentation is between 20 (WS=10) and 113 (WS=100) percent. For *Replay-Lazy* the overhead on the original query network (generation of *covering intervals*) can be further reduced to 3 (WS=10) and 16 (WS=100) percent respectively. The price to pay for this reduction of overhead is the additional cost of provenance *Retrieval*, where the cost is very similar to the combination of *Instrumentation* Generation and Retrieval, as this method is now applied on all covering intervals to compute the actual provenance.

8.2.2 Nested Aggregates

In this experiment, we investigate the cost of nested aggregations, because for the *Nested* network the provenance size grows exponentially with the number of nested aggregations. We start off with the *Basic* network (WS=10 and SL=1) and gradually add more aggregation operators with same settings to this network, creating the *Nested* network shown in Figure 16(e). The results (Figure 18) 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 percent (two aggregations). At three aggregations, the execution is no longer possible. The increase of cost for *Instrumentation* is (slightly)

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
Rewrite		7.2	625.0	crash	crash
Replay-Lazy	Cov. Inter.	3.1	4.4	5.2	6.3
	Retrieval	5.2	14.7	91.1	2224.0

Figure 18: Varying Number of Aggregation Operators: Completion Time in Seconds

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. Finally, the overhead of generating *Covering Intervals* for *Replay-Lazy* is around 10 percent over the baseline (*NoProvenance*), while the effort spent for replaying shows the same behavior as the total cost of *Instrumentation*.

8.3 Cost of Provenance Generation

We now further study the cost factors and optimizations for operators and their combinations. We focus on window-based aggregation, since it is not used in traditional, non-streaming workloads and produces large amounts of provenance.

8.3.1 Window Size

Since window size determines the amount of provenance per data tuple, it is important for studying the effectiveness of our compression methods. We use the *Basic* network, keep SL at 1 and the selectivity of the selection in front (S) at 25 percent. In addition to the methods shown before, we demonstrate the impact of provenance compression by enabling the adaptive technique (indicated by the label *Optimized* in all following graphs). 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).

Figure 19 shows *Completion Time*, *Queue Memory* and *Provenance Structure Memory* for varying WS from 50 to 2000. As expected, increasing the window size leads to increasing the cost, but compression can mitigate this increase significantly: While the Single TID representation sees an increase in completion cost from 70 % to 530 % overhead when increasing the window size, compression reduces this overhead to 50 % and 140 %, respectively. Covering Intervals further reduces this overhead to 14% and 70 %. Normalized on the amount of provenance generated, an increase by a factor of 40 (50 to 2000) leads to cost increase by a factor of 19 for *Single*, but only to a factor of 9 for *Optimized* and 8 for *Covering Intervals*. The cost savings of compression and covering intervals are even more pronounced on the queue and memory side, where the overhead introduced by *Single* is pushed down to a small, almost constant factor. It should be noted that the high 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.

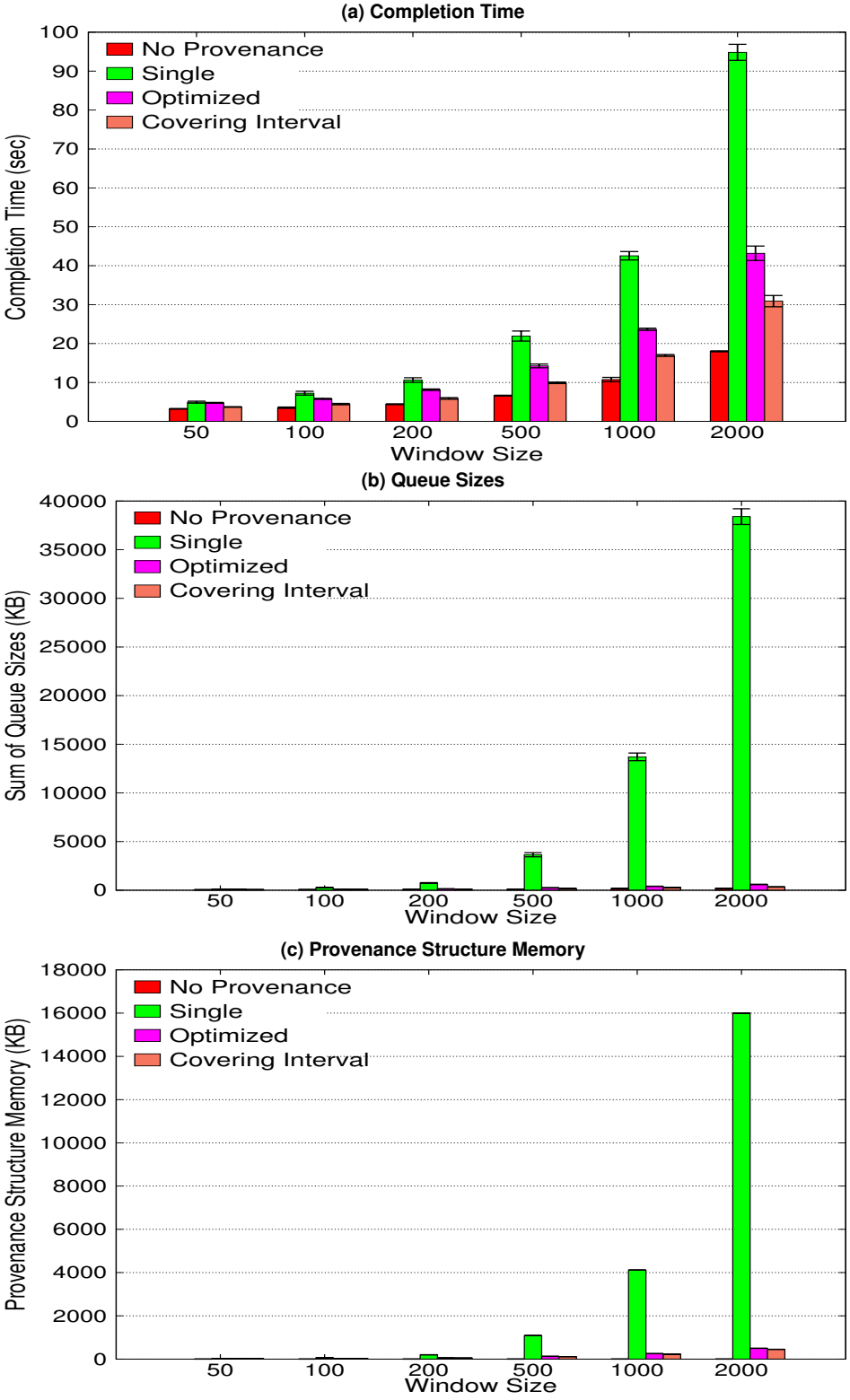


Figure 19: Varying Window Size (S = 25%, SL = 1)

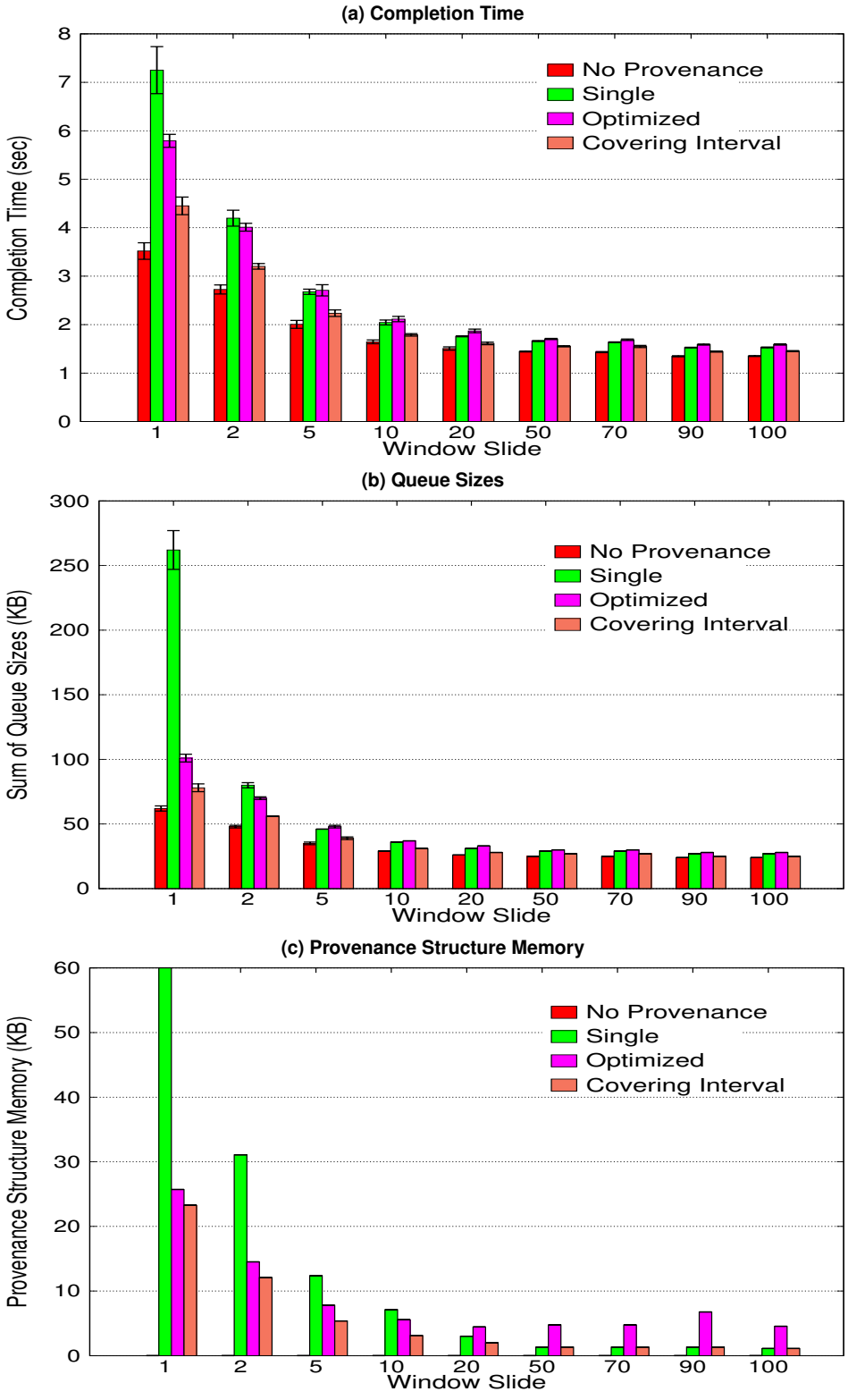


Figure 20: Varying Window Overlap (S = 25%, WS = 100)

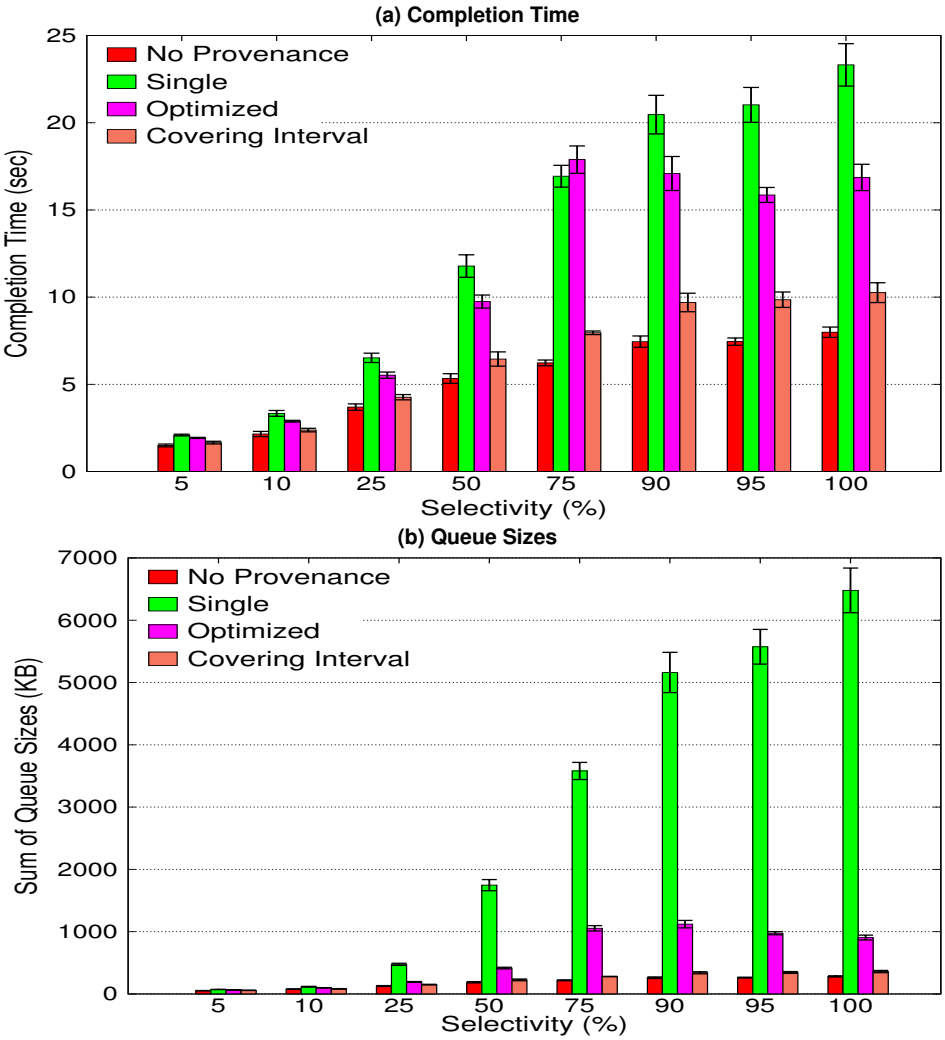


Figure 21: Varying Selectivity (WS = 100, SL = 1)

8.3.2 Window Overlap

Reducing the overlap between windows by increasing SL from 1 to 100 (with WS=100) decreases the overall cost consistently, since far fewer result tuples need to be generated (Figure 20). The logarithmic fashion of these three graphs can be explained by the fact that for a slide bigger 10, the system is not stressed anymore and therefore the impact of provenance generation is not noticeable. The memory consumed by provenance structures for Single and Interval Encoding drops linearly as the number of open windows decreases for bigger slides. Big slides 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 back anymore. As a result, these methods perform (slightly) worse than the naive approach (Single).

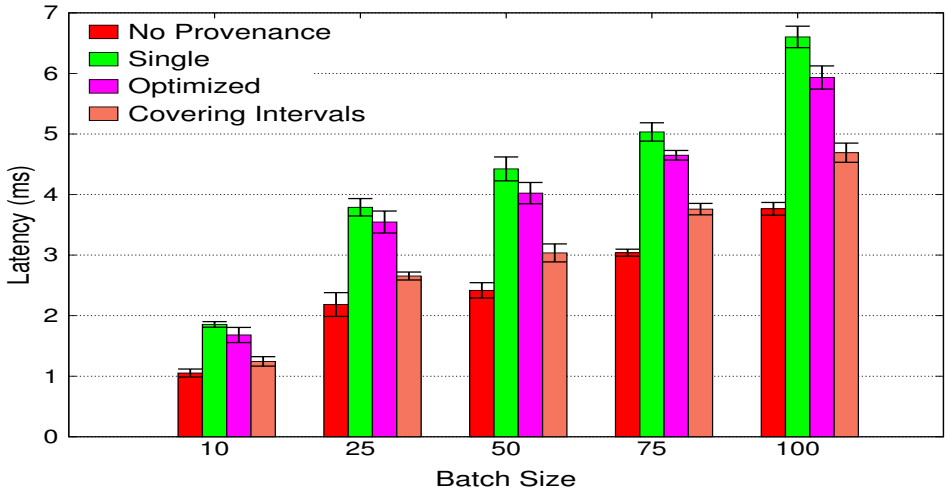


Figure 22: Latency Under Varying Network Load

8.3.3 Selectivity

Besides the specific window parameters such as WS oder SL, the workload on 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 percent while keeping WS at 100 and SL at 1 (Figure 21).

For *NoProvenance*, *Single* as well as *Covering Intervals*, *Completion Time* is linear in 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 after around 75 percent selectivity. In terms of *Queue Memory*, the naive approach (Single TIDs) results in bigger queues for higher selectivity values while other methods reach their maximums at medium selectivity values (between 50% and 75%), where the load level is relatively high but TID contiguities are not yet sufficiently big. We do not show a graph for *Provenance Structure Memory*, because this parameter depends mainly on WS and SL.

8.4 Influence of Network Load on Latency

So far, we have studied the cost of provenance on a network with *Maximum Load*, giving results on the overhead for computation and storage, and on worst-case behavior. In reality, a query network is rarely run at maximum load, and other performance metrics such as *Latency* play an important role. We take 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. Smaller batches introduced too much noise into the measurement, for larger sizes the slowest method (*Single*) would not be able to always process input instantly, thus skewing the latency measurement by additional wait time. As Figure 22 shows, 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* are the cheapest with around 20 % overhead.

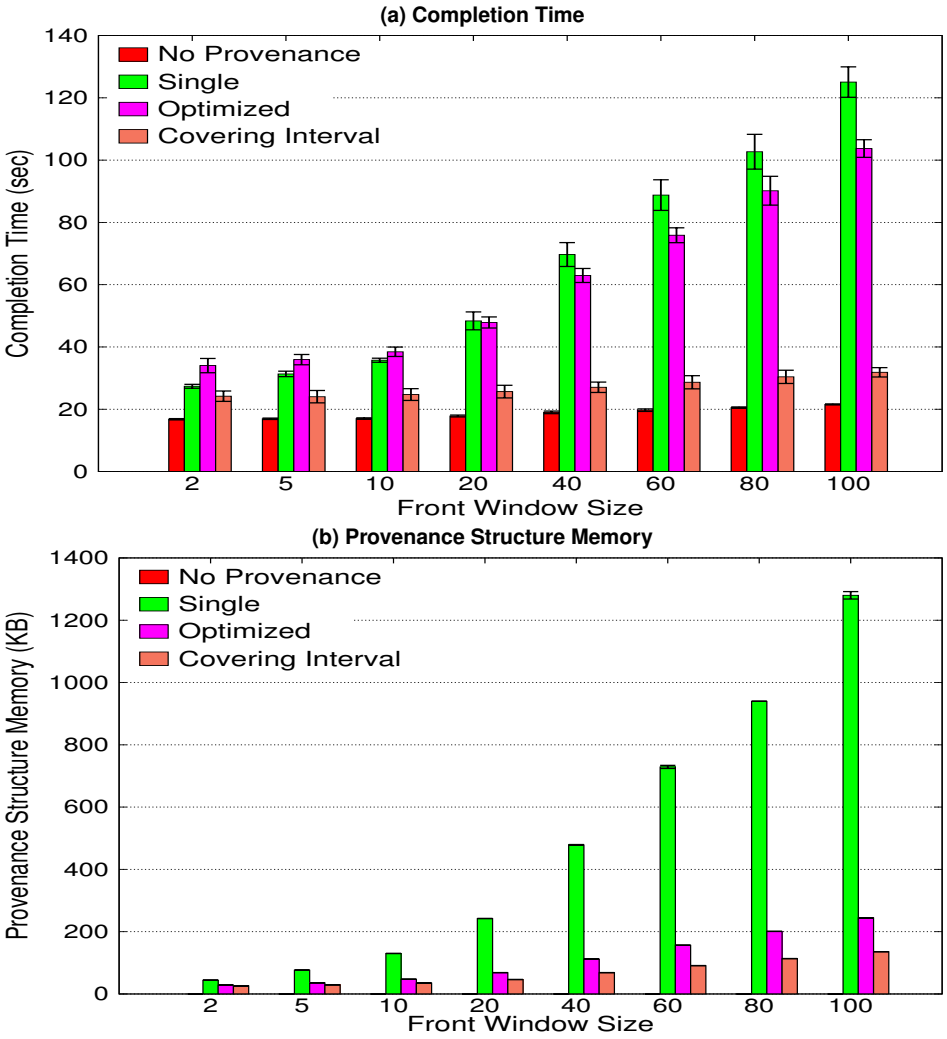


Figure 23: Complex Network: Varying Front Window Size

8.5 Complex Query Networks

We now investigate whether our understanding of the cost of individual operators translates to complex query networks using the *Complex* query network (see Section 8.1.1). This network consists of multiple paths and includes a broad selection of operators. 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 23 shows, the overhead of *Reduced-Eager* instrumentation without compression (*Single TIDs*) is higher than in previous experiments: around 60 percent at WS=2, and around 480 percent at WS=100. Our *Optimized* method using adaptive compression shows its benefits: while more expensive for very small WS values (100 percent overhead at WS=2), it becomes more effective once the workload grows. At WS=100, the overhead is at 380 percent. *Covering Intervals* is again very effective at 40 percent overhead, furthermore it is not affected by the increasing amount of provenance. Memory measurements support these observations, since the additional provenance does not increase the cost significantly when using compression or covering intervals.

8.6 Varying Retrieval Frequency

For many real-world scenarios, provenance is not retrieved on the entire result stream. In this experiment we therefore study the effect of retrieval frequency (as 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 measure completion time while varying the rate of retrieval from 0.05 to 100 percent (by inserting an additional selection before reconstruction). The results shown in Figure 24 (overhead with respect to completion time of *No Provenance*) demonstrates that both the eager and Replay-Lazy methods benefit from low retrieval frequencies, but for different reasons: *Reduced-Eager instrumentation* just saves the cost of reconstruction, but still performs full generation. For low retrieval frequencies (less than 1%) the cost of retrieval is insignificant. Thus, the overall cost for *Reduced-Eager* is dominated by the cost of provenance generation which is independent of the retrieval frequency. As a result we observe an approximately constant overhead of *Reduced-Eager* for retrieval frequencies below 2%. 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 very few replay requests at low retrieval rates, but in turn pays a higher overhead for higher retrieval rates. If the retrieval frequency is less than 10% then Replay-Lazy is the better choice for the given workload, e.g., for 0.05% retrieval frequency the relative overhead for Replay-Lazy is only 35% in contrast to 142% overhead for eager. This experiment confirms our assumption that significant savings are possible using *Replay-Lazy* unless provenance is required for most of the results.

8.7 Summary

Our experiments demonstrate the feasibility of fine-grained 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, enabling on-demand operations and distribution. 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.

9 Related Work

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

Workflow Systems. Davidson et al. [13] present a survey of provenance in workflow systems. Most of the approaches for workflow provenance handle tasks in workflows as black-boxes, and therefore, consider all outputs of a task to depend on all of its inputs. Such a provenance model is not a good match for the kind of use cases we are considering. More recently, Anand et al. have proposed a new workflow provenance model that allows explicit declarations of fine-grained data dependencies [7]. Amsterdamer et al. [5] apply a fine-grained database provenance model to workflows expressed in Pig Latin. In stream provenance, such declarations are not necessary, as we can directly exploit well-defined semantics of

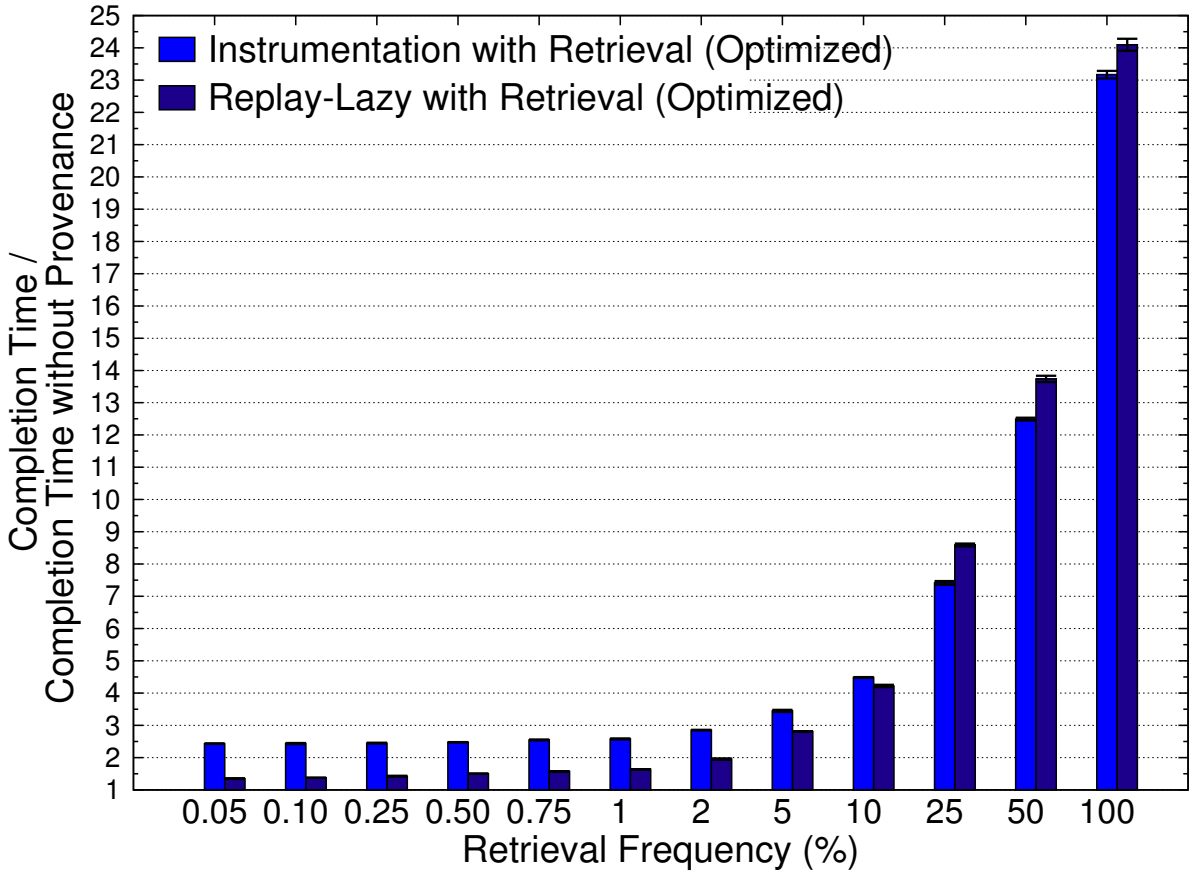


Figure 24: Relative Completion Time Overhead for Varying Retrieval Frequency

the streaming operators. Ariadne's provenance compression techniques are related to previous work on efficient provenance storage in workflow systems. More specifically, our delta encoding technique is similar to the sub-sequence compression technique of Anand et al. [7], and can be seen as a special case of node factorization approach of Chapman et al. [10]. The transient and incremental nature of streaming settings has led us to use compression for optimizing provenance generation, while previous work uses it for optimizing space and retrieval. Furthermore, our compression techniques cannot rely on global knowledge about all provenance in the system, while how we do compression at an operator may affect the load on its downstream operators (e.g., where a decompression might be needed). Therefore, our compression techniques have been designed to have small memory requirements, and efficient encoding and decoding mechanisms.

Database Systems. Several notions of provenance have been developed for databases [11] which are supported by different systems (e.g., Trio [8], DBNotes [9], and Perm [14]). In our work, we use a simple notion of provenance that represents the provenance of a tuple as a set of input tuples. This is similar to the *lineage provenance semantics* used in relational databases [11]. In principle, our Reduced-Eager operator instrumentation techniques can be extended to support more informative database provenance semantics, such as provenance polynomials [16] or graph-based models [3]. However, a major advan-

tage of some of these models is that they are invariant under query equivalence (i.e., equivalent queries have equivalent provenance), or they generalize other data model extensions (e.g., provenance polynomials generalize bag semantics). Another line of related work along this direction tries to reduce the size of provenance information, again relying on query equivalence in the relational model [6, 22]. Since languages for streaming queries have different semantics and, thus, different equivalence rules hold for these languages, it is an open question whether these existing provenance models or minimization techniques would apply – an interesting problem to look at in the future.

Stream Processing Systems. There is only a handful of related work on provenance management in stream processing systems. The need for low-overhead provenance collection for scientific stream processing has been addressed by Vijayakumar [28, 27]. However, this work focuses on coarse-grained provenance, which is not suitable for our use cases. Wang et al. have proposed a rule-based provenance model for sensor streams which captures operator states and time intervals that led to the generation of an output tuple [29, 21]. These rules have to be manually defined for each operation. Furthermore, it is unclear if they are powerful enough to deal with complex query networks efficiently. More recently, Huq et al. have proposed to achieve fine-grained stream provenance by augmenting coarse-grained provenance with timestamp-based data versioning, focusing specifically on query result reproducibility at reduced provenance metadata storage cost [18]. This work generates provenance using inversion based on static information about query operators and does not support common streaming operators such as joins and selections. Finally, the general provenance management solution of Ariadne can also be used as a supporting technology for other specialized tasks in stream processing systems such as revision processing [24] and query debugging [4].

10 Conclusions

In this paper, we present a prototype system for generation and retrieval of fine-grained provenance for data stream processing. We introduce a model for provenance generation of a query network (or parts thereof) based on annotated streams and two new classes of annotating operators (provenance generators and provenance propagators). Using our *Ariadne* prototype, we show how stream provenance can be implemented in a typical DSMS. Our experimental evaluation demonstrates that fine-grained provenance can be generated efficiently using Reduced-Eager Operator Instrumentation and provenance compression. Note that although Borealis was used as a proof-of-concept platform, our techniques are general enough to be easily adapted to other DSMSs as well.

There are many interesting avenues for future work. First, we would like to study provenance retrieval patterns to exploit additional knowledge for storage decisions and in optimizing computations. Second, we want to further investigate possible architectures, starting from a distribution of our current system and the integration of scalable, distributed storage systems. Third, we want to extend our provenance semantics to model how the order of input tuples and ordering imposed by operators influences the order and existence of output tuples and enable querying of this information.

References

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

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

[4] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona1, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu1, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Ying Li, V. Di Nicola, X. Wang, David Maier, S. Grell, O. Nano, and I. Santos. Microsoft CEP Server and Online Behavioral Targeting (Demonstration). In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, 2009.

[5] Y. Amsterdamer, S.B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *Proceedings of the VLDB Endowment*, 5(4):346–357, 2011.

[6] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On Provenance Minimization. In *Proc. of the Symposium on Principles of Database Systems (PODS)*, pages 141–152, 2011.

[7] Manish Kumar Anand, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Efficient Provenance Storage over Nested Data Collections. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, pages 993–1006, 2009.

[8] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with Uncertainty and Lineage. In *Proc. of the 32th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 953–964, 2006.

[9] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An Annotation Management System for Relational Databases. *VLDB Journal*, 14(4):373–396, 2005.

[10] Adriane Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient Provenance Storage. In *Proc. of the ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 993–1006, 2008.

[11] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[12] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Trans. on Database Systems (TODS)*, 25(2):179–227, 2000.

[13] Susan B. Davidson, Sarah Cohen-Boulakia, Anat Eyal, Bertram Ludäscher, Timothy McPhillips, Shawn Bowers, and Juliana Freire. Provenance in Scientific Workflow Systems. *IEEE Data Engineering Bulletin*, 32(4):44–50, 2007.

[14] Boris Glavic and Gustavo Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *Proc. of the IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 174–185, 2009.

[15] Boris Glavic, Kyumars Sheykh Esmaili, Peter M. Fischer, and Nesime Tatbul. The Case for Fine-Grained Stream Provenance. In *Proc. of the BTW Workshop on Data Streams and Event Processing (DSEP)*, pages 58–61, 2011.

[16] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance Semirings. In *Proc. of the Symposium on Principles of Database Systems (PODS)*, pages 31–40, 2007.

[17] Sun-Yuan Hsieh. The Interval-merging Problem. *Information Science*, 177(2):519–524, 2007.

[18] M.R. Huq, A. Wombacher, and P.M.G. Apers. Facilitating Fine Grained Data Provenance using Temporal Data Model. In *Proc. of the 7th Intl. Workshop on Data Management for Sensor Networks (DMSN)*, pages 8–13, 2010.

[19] Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The ORCHESTRA Collaborative Data Sharing System. *ACM SIGMOD Record*, 37(2):26–32, 2008.

[20] G. Karvounarakis, Z.G. Ives, and V. Tannen. Querying data provenance. In *Proc. of the 37th ACM Intl. Conf. on Management of Data (SIGMOD)*, pages 951–962, 2010.

[21] A. Misra, M. Blount, A. Kementsietsidis, D. Sow, and M. Wang. Advances and Challenges for Scalable Provenance in Stream Processing Systems. In *Proc. of the 2nd Intl. Provenance and Annotation Workshop (IPAW)*, pages 253–265, 2008.

[22] Dan Olteanu and Jakub Závodný. On Factorisation of Provenance Polynomials. In *Proc. of the 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2011.

[23] F. Reiss and J.M. Hellerstein. Data triage: An adaptive Architecture for Load Shedding in TelegraphCQ. In *Proc. of the 25th IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 155–156, 2005.

[24] E. Ryvkina, A. Maskey, M. Cherniack, and S. Zdonik. Revision Processing in a Stream Processing Engine: A High-Level Design. In *Proc. of the 22th IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 141–143, 2006.

[25] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *Proc. of the 23th Symposium on Principles of Database Systems (PODS)*, pages 263–274. ACM, 2004.

[26] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 309–320, 2003.

[27] Nithya Vijayakumar and Beth Plale. Tracking Stream Provenance in Complex Event Processing Systems for Workflow-Driven Computing. In *Proc. of the 2nd Intl. Workshop on Event-driven Architecture, Processing, and Systems (EDA-PS)*, 2007.

[28] N.N. Vijayakumar and B. Plale. Towards Low Overhead Provenance Tracking in Near Real-time Stream Filtering. In *Proc. of the 1th Intl. Provenance and Annotation Workshop (IPAW)*, pages 46–54, 2006.

[29] M. Wang, M. Blount, J. Davis, A. Misra, and D. Sow. A Time-and-Value Centric Provenance Model and Architecture for Medical Event Streams. In *Proc. of the 1st Intl. Workshop on Systems and Networking Support for Healthcare and Assisted Living Environments (ACM HealthNet)*, pages 95–100, 2007.

[30] Allison Woodruff and Michael Stonebraker. Supporting Fine-grained Data Lineage in a Database Visualization Environment. In *Proc. of the 30th Intl. Conf. on Data Engineering (ICDE)*, pages 91–102, 1997.