Stream Ancestor Function: A Mechanism for Fine-Grained Provenance in Stream Processing Systems

Watsawee Sansrimahachai
School of Electronics
and Computer Science
University of Southampton
Email: ws07r@ecs.soton.ac.uk

Mark J. Weal
School of Electronics
and Computer Science
University of Southampton
Email: mjw@ecs.soton.ac.uk

Luc Moreau
School of Electronics
and Computer Science
University of Southampton
Email: L.Moreau@ecs.soton.ac.uk

***Abstract*—Applications that require continuous processing of high-volume data streams have grown in prevalence and importance. These systems process streaming data in real-time and provide instantaneous response to support precise and ontime decisions. In such systems, it is difficult to know exactly how a particular result is generated or more particularly how to precisely trace stream events that caused a particular result. However, such information is extremely important for validating stream processing results. Therefore, it is crucial that stream processing systems have a mechanism for capturing and querying provenance information - the information pertaining to the process that produced result data - at the level of individual stream events, which we refer to as fine-grained provenance. In this paper, we propose a novel fine-grained provenance solution called Stream Ancestor Function - a reverse mapping function used to express precise dependencies between input and output stream elements. We demonstrate how to utilize stream ancestor functions by means of a stream provenance query and replay execution algorithm. Finally, we evaluate the stream ancestor function in terms of storage consumption for provenance collection and system throughput, demonstrating significant reductions in storage size and reasonable processing overheads.**

I. Introduction

The use of real-time data streams has played an important role in data-driven computational science. A data stream is a real-time, continuous, ordered sequence of data items which can be submitted from different kinds of data source (e.g. sensors) [11]. The size of data streams is usually unbounded and once each individual stream element has been processed it is eventually discarded or archived [1]. Because of the unique characteristics of data streams, the scientific community is adopting the data streaming technique for various kinds of applications that need instantaneous responses. Examples of stream-based applications include sensor network applications, real-time mapping systems and network monitoring systems.

Imagine that in a radioactivity leak incident in a nuclear submarine (such as [18]), an operator relies on a real-time mapping (geographic information system - GIS) application in order to manage and control the disaster. The information displayed on the GIS application is submitted by several sensors located near the scene of the incident in real-time. At some point during the incident, the operator found an anomaly in a number of results displayed in the GIS application. The operator anticipates that this problem has resulted from damage to, or malfunction of, sensors. The operator queries to find out which raw observations caused this anomaly and which sensors are responsible for sending the observations. If there is currently no provenance information - the information pertaining to the process that produced result data - available from the GIS application, the operator cannot easily determine why the unusual information is generated, which were the raw observations that led to the unusual information and which sensors contributed to that information being displayed in the GIS application.

This simple scenario illustrates the need for tracing individual results produced by existing stream processing systems. In such systems, a mechanism for tracking provenance at the level of individual stream elements, which we refer to as fine-grained provenance, is very important. The existence of such a functionality would allow users to be able to perform fault-diagnosis in the case of anomalies, to validate processing steps and to reproduce particular results in the case of stream imperfections. By understanding the process that led to each individual result produced by a stream system, users can have confidence in the data that is the output from the system.

To address fine-grained provenance tracking in stream processing systems, we propose a reverse mapping method called a *stream ancestor function* for each stream operation. The key concept of the stream ancestor function is that given the reference to a particular output element, it identifies the references to input elements involved in the production of the output. By utilizing an event key as an index for each individual stream element, all the input elements used in the generation of the output can be exactly identified.

To evaluate how precisely the stream ancestor function can identify individual stream elements involved in the production of a given output, a replay execution method is proposed. The key idea of the replay execution is that it utilizes provenance information stored in a provenance store in order to derive a particular output stream element. The support of the replay

execution would allow the stream systems to verify provenance query results and also to validate original results produced by the stream processing systems.

Applying the concept of the stream ancestor function to an actual stream processing system, however, may rise to a practical challenge. Because every intermediate stream element is required for the computing of stream ancestor functions, the persistence of high volume stream elements potentially results in a storage problem. To deal with this challenge, we present an enhanced solution for provenance collection that reduces the storage cost of the stream ancestor function model. With this solution, the implementation of our stream provenance model can offer a reasonable storage consumption.

This paper makes the following key contributions:

- It introduces a novel stream provenance model based on the key principle - the stream ancestor function which precisely captures dependency relationships for every individual stream element.
- It presents an enhanced solution for provenance collection that eliminates the requirement for storing every intermediate stream element.
- It identifies a set of primitive stream operations for which fine-grained provenance can be computed.
- It presents a novel fine-grained stream provenance query and replay execution algorithm.

The rest of this paper is organized as follows. Section 2 reviews previous work in the area of provenance and stream processing systems. Section 3 introduces a fine-grained provenance model for stream processing systems. Section 4 presents algorithms for provenance query and replay execution. Section 5 demonstrates the evaluation of stream ancestor functions in terms of storage consumption for provenance collection and the impact of provenance recording (system throughput). Finally, section 6 presents conclusions and further work.

II. Related work

Tan [20] classifies research studies on data provenance (sometimes called lineage) into two distinct approaches: in the lazy approach, provenance is generated on demand, by means of a query, only when requested [24], [7], [8], whereas in the eager approach information is propagated at runtime [4], [6]. We now discuss them in turn.

To address the data lineage problem, Woodruff and Stonebraker [24] proposed a technique called weak inversion which is used to regenerate input data items that produced a particular output. The drawback of this technique is that the answer returned by this function is not guaranteed to be perfectly accurate and the weak inversion need to be defined by a user who creates a new database. In [7], [8], a lineage tracing algorithm has been proposed. This lazy algorithm can generate provenance information through analyzing view definitions and query algebraic structures.

Buneman *et al.* [3] formalize the data provenance problem and draw a distinction between two types of provenance: "why-provenance" and "where-provenance". Why-provenance determines what tuples in the source database contributed to an output data item. Where-provenance, on the other hand, identifies locations in the source database from which the data item was extracted. Based on these types of provenance, an eager approach propagating annotation according to *propagation rules* [4] has been proposed to address where-provenance. The idea of annotation propagation is further extended by DBNotes [6]. In DBNotes, an extension of a fragment of SQL was introduced to allow users to specify how annotations should propagate. Green *et al.* [12] define *provenance semirings* as a formal way of understanding "how-provenance" which describes how the input data leads to the existence of the output data. Annotations in the form of variables are propagated so as to form polynomials with integer coefficients for the output tuples.

Based on literature in the context of database systems, our provenance solution aims to address a form of "why-provenance" for stream systems since it aims to identify a minimal set of input elements used in the production of a particular output element. The approach presented in this paper combines both eager and lazy techniques, eagerly propagating minimum information at runtime, and relying on queries to extract fine-grained provenance, lazily, on demand. Instead of using a simple or non-structural annotation, our provenance model uses a structural annotation (event key). We show that this type of annotation is more suitable for expressing dependencies between input and output stream elements.

In the context of scientific workflow, the data provenance problem has received substantial attention. Taverna [17] provides support for provenance tracking to allow scientists to understand how results from experiments were obtained. In Taverna, provenance information is collected by recording metadata information and intermediate results during workflow enactment. Another system, the COMAD provenance framework [2], is designed specifically to deal with collections of data. To trace provenance of scientific data products, output collections are embedded with a metadata annotation containing explicit data dependency information. Such annotations are used to describe the derivation of output objects computed in a scientific workflow.

PASOA [13], [14] investigates the concept of provenance and built an infrastructure for recording and reasoning over provenance in service-oriented architectures. It is mainly designed for supporting interactions between loosely-coupled services. By recording assertions comprising interaction messages and causal relationships between messages, provenance of output data products can be captured. Our provenance solution for streams extends the PASOA provenance mechanism. In our model, a stream operation is treated as a "grey box" [9] and provenance is collected based on dependencies between input and output elements of the operation. However, because PASOA need to store all dependencies and intermediate data objects, the amount of information recorded can potentially cause a storage problem when dealing with high volume data streams. Therefore, one of requirements for our provenance solution is to find out an enhanced technique that

can address this storage problem. Furthermore, our approach is also compatible with the community-based Open Provenance Model [16], though it proposes a compact representation of its was-derived-from edges.

A few research efforts have been made in exploring provenance techniques for data streams. Vijayakumar *et al.* [21], [22] propose a provenance model and architecture to track provenance in stream filtering systems. Their study focuses on a coarse-grained provenance method that identifies dependencies between streams or sets of stream elements as the smallest unit for which provenance is collected. However, the method does not offer a level of granularity for capturing provenance that is detailed enough to identify dependencies among individual stream elements. The *Time-Value-Centric* model [23], [15] (TVC) is a finer-grained provenance solution that provides the ability to express data dependencies for individual stream elements. In this model, dependencies between input and output stream elements are described in terms of three primitive invariants: time, value and sequence. This model assumes that all elements of all data streams are persisted. By composing these primitives, provenance of individual stream elements can be explained. Nevertheless, this model still has some limitations, since it can fail to identify precisely input stream elements that are used in the production of an output. Another limitation of this model is pertaining to storage consumption for provenance collection. Because all intermediate stream elements need to be stored for computing the provenance of an output element, the persistence of high volume stream events potentially results in a storage burden problem. Therefore, to address these limitations, we introduce a finer-grained provenance solution that precisely captures the provenance of every individual stream element without requiring every intermediate stream elements to be stored.

So, to sum up, the approach that is presented in this paper improves over the state-of-the-art in multiple ways. It defines a fine-grained notion of provenance for streams similar to why-provenance, which can explain the presence of individual elements in streams. In doing so, it identifies a class of stream operations for which such fine-grained provenance can be determined. It blurs the distinction between lazy and eager approaches, since it propagates structured information at runtime, which is exploited for retrieving provenance by queries, on demand. Furthermore, it reduces the storage requirement compared to a related stream provenance approach.

III. Fine-grained stream provenance model

Our requirement for the provenance support in stream processing systems is to track provenance information at the level of individual stream events so that data dependencies for each individual stream event in a particular processing step can be examined. To address the requirement, we introduce a reverse mapping method called a *stream ancestor function* for each stream transformation (stream operation). The purpose of the stream ancestor function is to explicitly express dependency relationships between input and output elements of a stream operation. We first present primitive

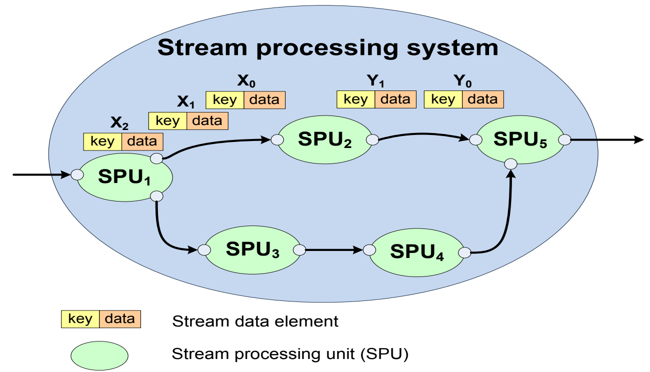


Fig. 1. Infrastructure of a stream processing system

stream operations to demonstrate how the output element for each stream operation is produced in terms of input elements. Based on these operations, the specification of a stream ancestor function for each stream operation is described. It is important to note that all stream operations are defined by using Event Processing Language (EPL) (sometime called StreamSQL [19])- an SQL-like language extended to handle event streams. We use EPL provided by Esper [10] - an open-source stream processing engine - to illustrate how continuous queries is formulated in stream operations and how provenance assertions are computed. The definition of stream ancestor functions is presented by using standard Structured Query Language (SQL). By using SQL and EPL which is a variant of the SQL language, we believe that a concise and clear definition of stream ancestor functions can be explained.

A. Basic assumptions for stream provenance model

To design a fine-grained provenance model, we make the following assumptions describing stream systems and a stream model that our provenance model is intended to support.

- A stream processing system is represented as a set of interconnected nodes, with each node representing a stream operation or a stream processing unit (SPU). Input stream events flow through a directed graph of stream processing operations and finally, streams of output events are presented to applications that subscribe to receive results.
- Each data stream consists of a sequence of time ordered stream events and each individual stream event is composed of an event key and a content of stream event (data). An event key - a unique reference of an individual stream event - contains a timestamp, a sequence number, a stream identifier and a delay time for processing.
- Streams are implicitly timestamped. In this kind of stream timestamps, stream events that first enters a stream system from data sources are timestamped on entry to a stream system. Timestamps are derived from a stream system time, and are typically based on the order in which stream events arrive at the stream system.

From the definitions of a data stream, a timestamp and a sequence number serve to define a temporal order and sequential order among stream events in a stream. Because

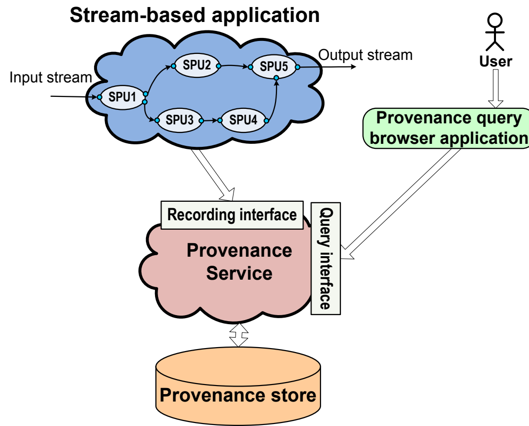


Fig. 2. Provenance architecture for stream processing systems

a processing delay time for each stream event is generally different, it is necessary to include a delay time in an event key. The delay time is used as a variable for computing time dependencies between stream events. The stream processing infrastructure based on our assumptions is shown in Figure 1.

To overview our stream provenance system, we present a provenance architecture for stream processing systems in Figure 2. A provenance service plays a central role in this architecture. The provenance service consisting of a recording service and a query service is responsible for receiving provenance assertions from a stream system. Provenance assertions are typically recorded as a stream. In the case that replay execution is required, a provenance store - a central storage component that offers a long-term persistent storage - is used to stored streams of provenance assertions. The provenance of stream processing results can be retrieved by performing provenance query through a query interface.

B. Primitive stream processing operations

In this section, primitive stream operations for expressing stream processing requirements are presented. These stream operations include windowed operations that operate on sets of consecutive events from a stream at a time and operations that operate on a single event at a time. These operations are recognized as common stream operations developed in several stream projects [11], [5].

- **Map** *Map(F,sid):* A map operation is a stream operation that operate on a single stream element at a time. The operation applies an input function (F) to the content of every element in a stream.
- **Filter** *Filter(P,sid):* A filter operation screens events in a stream for those that satisfy an input predicate (P).
- **Sliding time window** *TW(w,sid):* A time window is a data window where the extent of the window is defined in terms of time interval. At any point in time, the time window generates an output event from the most recent input events over a given time period (w).
- **Length window** *LW(l,sid):* A length window is a data window where the extent of the window is defined in terms of the number of events. At any point in time, a length window covers the most recent N events (size of window - l) of a stream.
- **Time-window join** *JoinTW(w1,w2,sid):* A time-window join is a binary operation that pairs stream events from two input streams. Stream events from two time-based windows are combined and output events are produced according to a join condition. The required parameters consists of duration of time windows (w1,w2).
- **Length-window join** *JoinLW(l1,l2,sid):* Similar to the time-window join, a length-window join is a binary operation that join pairs stream events from two input streams. The difference between these two operation is that the length-window join operates on stream events from two tuple-based windows. The required parameters consists of size of length windows (l1,l2).

The definitions of primitive stream operations are described in Table I. In this context, a stream id (sid) for every operation is the ID of an output stream. Note that the filter, time-window join and length-window join operations use a sequence number (sn) as an internal variable which is generated according to a number of output events. Furthermore, every stream operation presented in the table takes a stream event - Event(key(t,n,s,d),data) - as an input.

C. Stream ancestor functions

We can express dependencies between input and output events of stream operations by means of a *stream ancestor function* (SAF) that is defined for each stream operation. The key idea of stream ancestor function is that for a given reference to an output element, it identifies which references to input events involved in the production of that output. The stream ancestor function does not work directly with individual stream elements, but instead it operates on a representation of each individual stream element - provenance assertion - that is recorded by each stream operation. We assume that every provenance assertion contains an event key. The event key plays an important role in the mapping process of the stream ancestor function. It serves as a unique reference for identifying each provenance assertion of an individual stream element in a stream. By composing stream ancestor functions for all stream operations in a stream system, all the elements of the intermediate streams (represented by particular provenance assertions) involved in the processing of a particular output, which we refer to as the complete provenance of a stream processing result, can be exactly identified.

The concept of stream ancestor function is illustrated in Figure 3(a). This figure shows how the stream ancestor function is used to express dependencies between input and output stream elements. In this example, we can determine the input events involved in the processing of the output event Y_0 by passing the provenance assertion $PA(Y_0)$ to the stream ancestor function defined explicitly for SPU2. The stream ancestor function returns the provenance assertions $PA(X_0)$ and $PA(X_1)$ which represent the stream events X_0 and X_1 belonging to the input stream of SPU2.

TABLE I
THE DEFINITIONS OF PRIMITIVE STREAM PROCESSING OPERATIONS

	Stream operations *(input: Event(key(t,n,s,d),data))*
$Map(F, sid)$	$(*\, fn\ :\ ('a \rightarrow' b)\ *\ INT\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'b\ EVENT\ list\ *)$ $insert\ \ into\ \ resultEvent(timestamp, seqNo, streamID, delay, eventData)$ $select\ \ current_timestamp, n, \mathbf{sid}, (current_timestamp - t), \mathbf{F}(data)\ \ from\ \ Event$
$Filter(P, sid)$	$(*\, fn\ :\ ('a \rightarrow bool)\ *\ INT\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ *)$ $insert\ \ into\ \ resultEvent(timestamp, seqNo, streamID, delay, eventData)$ $select\ \ current_timestamp, sn, \mathbf{sid}, (current_timestamp - t), data\ \ from\ \ Event\ \ where\ \ \mathbf{P}(data)$
$TW(w, sid)$	$(*\, fn\ :\ TIME\ *\ INT\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ *)$ $insert\ \ into\ \ resultEvent(timestamp, seqNo, streamID, delay, eventData)$ $select\ \ current_timestamp, n, \mathbf{sid}, (current_timestamp - t), list(data)\ \ from\ \ Event.win : time(\mathbf{w})$
$LW(l, sid)$	$(*\, fn\ :\ INT\ *\ INT\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ *)$ $insert\ \ into\ \ resultEvent(timestamp, seqNo, streamID, delay, eventData)$ $select\ \ current_timestamp, n, \mathbf{sid}, (current_timestamp - t), list(data)\ from\ \ Event.win : length(\mathbf{l})$
$JoinTW(w1, w2, sid)$	$(*\, fn\ :\ TIME\ *\ TIME\ *\ INT\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ *)$ $insert\ \ into\ \ resultEvent(timestamp, seqNo, streamID, delay, eventData)$ $select\ \ current_timestamp, sn, \mathbf{sid}, (current_timestamp - MaxTime(e1.t, e2.t)), J(e1.data, e2.data)$ $from\ \ Event1.win : time(\mathbf{w1})\ \ as\ \ e1, Event2.win : time(\mathbf{w2})\ \ as\ \ e2$
$JoinLW(l1, l2, sid)$	$(*\, fn\ :\ INT\ *\ INT\ *\ INT\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ \rightarrow\ 'a\ EVENT\ list\ *)$ $insert\ \ into\ \ resultEvent(timestamp, seqNo, streamID, delay, eventData)$ $select\ \ current_timestamp, sn, \mathbf{sid}, (current_timestamp - MaxTime(e1.t, e2.t)), J(e1.data, e2.data)$ $from\ \ Event1.win : length(\mathbf{l1})\ \ as\ \ e1, Event2.win : length(\mathbf{l2})\ \ as\ \ e2$

TABLE II
THE DEFINITIONS OF STREAM ANCESTOR FUNCTIONS

	Stream ancestor functions *(input: key(t,n,s,d))*
$Map_A(sid)$	$(*\, fn\ :\ INT\ \rightarrow\ KEY\ \rightarrow\ KEY\ list\ *)$ $select\ \ timestamp, seqNo, streamID, delay$ $from\ \ assertions$ $where\ \ streamID\ \ =\ \ \mathbf{sid}\ \ and\ \ (timestamp = t - d)$
$Filter_A(sid)$	$(*\, fn\ :\ INT\ \rightarrow\ KEY\ \rightarrow\ KEY\ list\ *)$ $select\ \ timestamp, seqNo, streamID, delay$ $from\ \ assertions$ $where\ \ streamID\ \ =\ \ \mathbf{sid}\ \ and\ \ (timestamp = t - d)$
$TW_A(w, sid)$	$(*\, fn\ :\ TIME\ *\ INT\ \rightarrow\ KEY\ \rightarrow\ KEY\ list\ *)$ $select\ \ timestamp, seqNo, streamID, delay$ $from\ \ assertions$ $where\ \ streamID\ \ =\ \ \mathbf{sid}\ \ and\ \ (timestamp\ \ \geq\ \ t - d - \mathbf{w})\ \ and\ \ (timestamp\ \ \leq\ \ t - d)$
$LW_A(l, sid)$	$(*\, fn\ :\ INT\ *\ INT\ \rightarrow\ KEY\ \rightarrow\ KEY\ list\ *)$ $select\ \ timestamp, seqNo, streamID, delay$ $from\ \ assertions$ $where\ \ streamID\ \ =\ \ \mathbf{sid}\ \ and\ \ (seqNo\ \ \geq\ \ (n - \mathbf{l}) + 1)\ \ and\ \ (seqNo\ \ \leq\ \ n)$
$JoinTW_A(w1, w2, sid1, sid2)$	$(*\, fn\ :\ TIME\ *\ TIME\ *\ INT\ *\ INT\ \rightarrow\ KEY\ \rightarrow\ KEY\ list\ *)$ $select\ \ timestamp, seqNo, streamID, delay$ $from\ \ assertions\ \ where\ \ streamID = \mathbf{sid1}\ \ and\ \ (timestamp \geq t - d - \mathbf{w1})\ \ and\ \ (timestamp \leq t - d)$ $union$ $select\ \ timestamp, seqNo, streamID, delay$ $from\ \ assertions\ \ where\ \ streamID = \mathbf{sid2}\ \ and\ \ (timestamp \geq t - d - \mathbf{w2})\ \ and\ \ (timestamp \leq t - d)$
$JoinLW_A(l1, l2, sid1, sid2)$	$(*\, fn\ :\ INT\ *\ INT\ *\ INT\ *\ INT\ \rightarrow\ KEY\ \rightarrow\ KEY\ list\ *)$ $select\ \ timestamp, seqNo, streamID, delay$ $from\ \ assertions\ \ where\ \ streamID = \mathbf{sid1}\ \ and\ \ (timestamp \leq t - d)$ $order\ \ by\ \ timestamp\ \ desc\ \ fetch\ \ first\ \ \mathbf{l1}\ \ rows\ \ only$ $union$ $select\ \ timestamp, seqNo, streamID, delay$ $from\ \ assertions\ \ where\ \ streamID = \mathbf{sid2}\ \ and\ \ (timestamp \leq t - d)$ $order\ \ by\ \ timestamp\ \ desc\ \ fetch\ \ first\ \ \mathbf{l2}\ \ rows\ \ only$

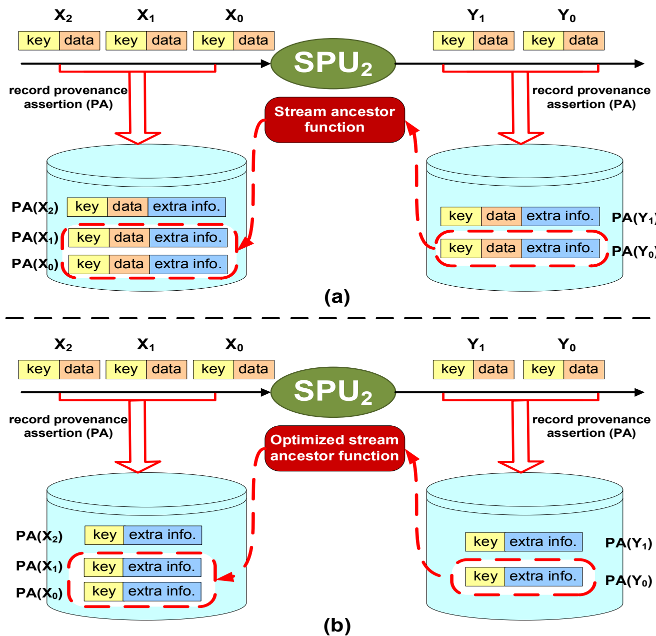


Fig. 3. Unoptimized (a) vs optimized stream ancestor function (b)

To extend the concept of stream ancestor functions, we introduce an enhanced solution called an *optimized stream ancestor function*. The aim of this optimized function is to reduce storage consumption of the original function by recording only necessary information. For this solution, only contents of stream events that act as first input to a stream processing system are stored. Every intermediate stream event is recorded only its key. Considering the fact that we use provenance assertions as representations of individual stream elements; therefore, the generation of the provenance assertions for every intermediate stream event does not include the contents of stream events, except for the first-input events - stream events that enter a stream system from data sources (e.g. sensors) and are first processed by stream operations. In addition, when the content of each individual event is required, we can obtain the content by replaying the execution of stream operations. With the concept of optimized stream ancestor functions, the amount of storage consumed for provenance collection is reduced and thus this can address the storage burden problem. Figure 3(b) illustrates the concept of optimized stream ancestor functions.

As illustrated in Figure 3(b), for each provenance assertion that represents individual event, only an event key is contained. The content of each intermediate event is discarded because it can be obtained later by replaying the stream execution if required. In this example the optimized stream ancestor function takes the provenance assertion $PA(Y_0)$ representing the output event Y_0 as an input and returns the provenance assertions $PA(X_0)$ and $PA(X_1)$ which represent the input events X_0 and X_1 that are involved in the production of Y_0.

The definitions of stream ancestor functions are presented in table II. Because of the reason of space, we can only present the optimized stream ancestor functions. Furthermore, because of the simplicity and conciseness of the specifications, the stream ancestor functions that will be presented are used event keys as inputs and outputs of the functions instead of provenance assertions. Every stream ancestor function takes an event key - $key(t, n, s, d)$ - as an input and returns a set of output event keys. The important parameter used in all stream ancestor functions is sid (stream identifier). In this context, sid for every ancestor function is the ID of an input stream.

According to the definition, stream ancestor functions for windowed operations (TW, LW, JoinTW and JoinLW) utilize parameters a size of data window and a delay time for each stream element in order to define the extent of a past data window which a particular output element is generated from. For example, the definition of $TW_A(w, sid)$ indicates that an output element containing an event key(*key(t,n,s,d)*) (represented by a provenance assertion) is generated from a past data window where the interval of the window is between *t - d - w* (lower bound) and *t - d* (upper bound). Aside from windowed operations are stream operations that operate on a single element at a time such as Map and Filter operations. Stream ancestor functions for these operations straightforwardly utilize a timestamp and a delay time for each output element to identify ancestor elements (provenance assertions) belonging to an input stream. For example, the definition of $Map_A(sid)$ indicates that an output element containing an event key (*key(t,n,s,d)*) is generated from an input element that was timestamped with a value *t - d*.

IV. Provenance queries and replay execution

A. Provenance queries

We now describe how to utilize stream ancestor functions to address the fine-grained provenance query. In this context, the processing flow of a stream processing system is represented as a set of interconnected nodes (stream operations). By composing all nodes, the output of the stream system can be retrieved. To trace back a particular output event, it is necessary to compose stream ancestor functions for stream operations as well. For each stream ancestor function, we can identify input events (representations of input events) involved in the processing of a particular output event. By composing all stream ancestor functions in a stream system, the complete provenance of an individual stream element can be captured.

Figure 4 illustrates the pseudo-code for the fine-grained provenance query. The important concept of this provenance query algorithm is that for a given event key of output stream element, the provenance query algorithm dynamically composes stream ancestor functions for all stream operations in the processing flow of a stream processing system together (like traversing a graph in reverse order on a node by node basis) in order to resolve data dependencies among intermediate stream elements. This provenance query algorithm consists of two internal functions: *retrieveAncestors* and *composeSAF* functions. The *retrieveAncestors* is the entry-point function that takes a list of event keys and a list of targeted stream IDs (stream IDs used to terminate the query) as input parameters and returns a set of event keys which is a query result.

```
1: /* fn: KEY list * INT list → KEY list */
2: Function retrieveAncestors(
3:   keyList :KEY list,
4:   sidList :INT list) {
5:     resultList = [] :KEY list
6:     composeSAF(keyList,sidList,resultList);
7:     return resultList;
8: }
```

```
1: Function composeSAF(
2:   keyList :KEY list,
3:   sidList :INT list,
4:   resultList :KEY list)  {
5: bufferList = [] :KEY list
6: for each  element (key) ∈  keyList do
7:    /*check whether sid of the element is the terminated
      sid or not*/
8:    if key.sid ∈ sidList then
9:      resultList.add(key);
10:   else
11:     /*get a stream ancestor function by using sid*/
12:     saf = getSAF(key.sid);
13:     /*execute the stream ancestor function on the
       element*/
14:     outputList = saf.execute(key);
15:     bufferList.add(outputList);
16:   end if
17: end for
18: if bufferList ≠ empty then
19:   /*recursive call if there are elements in
      bufferList*/
20:   composeSAF(bufferList,sidList,resultList);
21: end if
22: }
```

Fig. 4. Algorithm for a fine-grained provenance query

The other function, *composeSAF*, is the recursive function that contains the business logic of the provenance query. The process of the function consists of first receiving parameters passed by the retrieveAncestors function. Then, it iterates over a list of event keys ($keyList$) in order to execute stream ancestor functions on every input event key. For each event key, it will be processed by its associated stream ancestor function. Finally, if there are output event keys in the data buffer (it means some intermediate event keys still waiting to be processed), the $composeSAF$ function will be called recursively until no event keys in the data buffer.

B. Replay execution

Similar to the provenance query, our replay execution method applies the idea of function composition. The fundamental concept of the replay method is that it utilizes provenance assertions, configuration parameters of a processing flow and stream operation parameters which are stored in a provenance store in order to derive an output stream element. We assume that the processing flow of a stream processing system is represented as a set of interconnected nodes and stream events flow through a directed graph of stream processing operations. Therefore, we can derive a particular output stream event in the processing flow by composing stream operations involved in the production of that output element.

Figure 5 illustrates the pseudo-code for the replay execution. In our replay execution algorithm, a hash map (*HashMap*) ,which consists of a stream ID as a key and a list of stream events as its associated value, is mainly used to store intermediate results. The replay execution algorithm consists of two

```
1: /* fn: HashMap<INT,Event list> * INT list →
   HashMap<INT,Event list> */
2: Function replayExecution(
3:   inputMap :HashMap,
4:   sidList :INT list) {
5:     resultMap = new HashMap();
6:     composeStreamOp(inputMap,sidList,resultMap);
7:     return resultMap;
8: }
```

```
1: Function composeStreamOp(
2:   inputMap :HashMap,
3:   sidList :INT list,
4:   resultMap :HashMap) {
5: bufferMap = new HashMap();
6: for each  key  element (sid_in) ∈  inputMap do
7:   tempMap = new HashMap();
8:   tempMap.put(sid_in,inputMap.get(sid_in));
9:   opID = getOperationID(sid_in);
10:   /*obtain all required parameters*/
11:   paramMap = getParameters(opID);
12:   istreamList = getInputStream(opID);
13:
14:   /*if the operation has more than one input stream*/
15:   notFoundFlag = false;
16:   if istreamList.size()  >  1   then
17:     for each  element (istr) ∈  istreamList do
18:       if istr ∈ inputMap.key() then
19:         tempMap.put(istr,inputMap.get(istr));
20:       else
21:         bufferMap.put(sid_in,inputMap.get(istr));
22:         notFoundFlag = true;
23:       end if
24:     end for
25:   end if
26:   if notFoundFlag = true then
27:     continue; /*skips the current iteration*/
28:   end if
29:
30:   /*get a stream operation by using operation ID*/
31:   streamOp = getStreamOperation(opID);
32:   /*execute the stream operation*/
33:   returnMap = streamOp.execute(tempMap,paramMap);
34:
35:   for each  key  element (sid_out) ∈  returnMap do
36:     /*check if sid_out is the terminated sid*/
37:     if sid_out ∈ sidList   then
38:       resultMap.put(sid_out,returnMap.get(sid_out));
39:     else
40:       bufferMap.put(sid_out,returnMap.get(sid_out));
41:     end if
42:   end for
43: end for
44: if bufferMap ≠ empty  then
45:   /*recursive call if there are elements in bufferMap*/
46:   composeStreamOp(bufferMap,sidList,resultMap);
47: end if
48: }
```

Fig. 5. Algorithm for replay execution

internal functions: *replayExecution* and *composeStreamOp*. The *replayExecution* function is the entry-point function that takes a hash map containing input stream events and a list of stream IDs used to terminate the replay execution as input parameters and returns a result hash map which contains output events produced by the replay execution process. Like the provenance query algorithm, the *composeStreamOp*, is the recursive function that contains the business logic of the replay execution method. The process of the function begins by receiving parameters passed by the $replayExecution$ function. Then, it iterates over a list of input stream IDs (sid_in) in order to execute stream operations on every input stream and every input event. For each input stream, all parameters

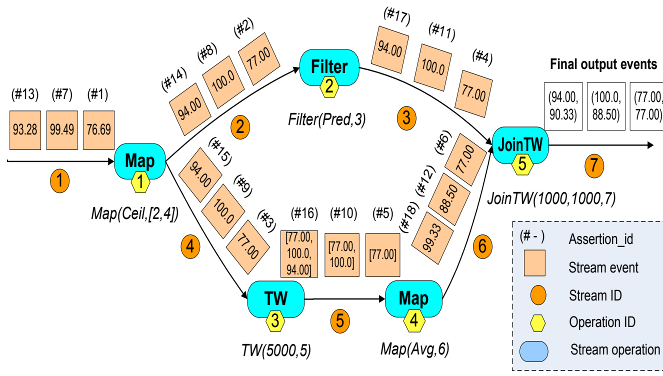


Fig. 6. The processing flow of a stream processing system

required for the processing of its associated stream operation are collected (using $getParameters$ function). If the stream operation has more than one input streams, the other input stream of the operation is obtained. Then, the stream operation is executed by using all events of the input streams and the required parameters. Finally, if there are streams in the data buffer, the $composeStreamOp$ function will be called recursively until no intermediate streams in the data buffer.

C. A case study for fine-grained provenance queries

To demonstrate that our provenance solution is expressive enough for precisely tracking provenance at the level of individual stream events, we use a simple synthetic processing flow of a stream system as an example. A provenance query constructed by composing stream ancestor functions is presented to capture the provenance of the synthetic processing flow. Then, the outputs of the provenance query are used as an input for our replay execution method in order to demonstrate how to validate provenance query results.

The processing flow of a stream processing system is presented in Figure 6. It is constructed by composing stream operations including *Map*, *Filter*, *Sliding time window (TW)* and *Time window join (JoinTW)*. For each operation, input and output streams are labelled with unique IDs (stream identifiers). Each stream operation is assigned a unique operation ID as well. Figure 7 presents the assertions table which is used to store a set of provenance assertions recorded during the execution of the synthetic processing flow. The assertions table consists of the following fields: an assertion identifier (assertion id), a set of fields representing an event key of a stream element(timestamp, seqno, stream id and delay), event content and an assertor (operation ID of a stream operation that records provenance assertions). In this example, three input stream events are fed into the system. In addition, we also provide an extra field for event content that are discarded during the generation of provenance assertions.

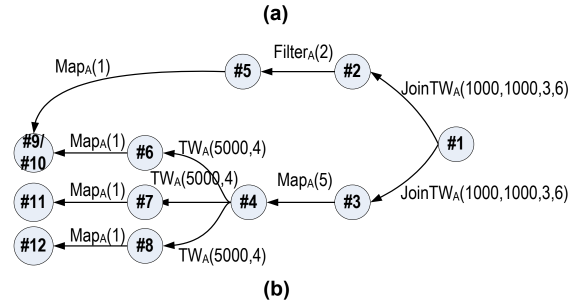
To capture the provenance of the processing flow, stream ancestor functions including Map_A, $Filter_A$, TW_A and $JoinTW_A$ are composed. The provenance assertions of all intermediate streams are required to be stored in a provenance store to support provenance queries. The provenance query

assertion_id	timestamp	seqno	stream_id	delay	event_content	assertor	content discarded
1	1279398105675	1	1	0	76.69	1	
2	1279398105684	1	2	9		3	77.00
3	1279398105684	1	4	9		2	77.00
4	1279398105693	1	3	9		5	77.00
5	1279398105701	1	5	17		4	[77.00]
6	1279398105702	1	6	1		5	77.00
7	1279398107678	2	1	0	99.49	1	
8	1279398107678	2	2	0		3	100.00
9	1279398107678	2	4	0		2	100.00
10	1279398107680	2	5	2		4	[77.00,100.00]
11	1279398107681	2	3	3		5	100.00
12	1279398107682	2	6	2		5	88.50
13	1279398109678	3	1	0	93.28	1	
14	1279398109678	3	2	0		3	94.00
15	1279398109678	3	4	0		2	94.00
16	1279398109680	3	5	2		4	[77.00,100.00,94.00]
17	1279398109681	3	3	3		5	94.00
18	1279398109682	3	6	2		5	90.33

Fig. 7. The assertions table

index	assertion_id	timestamp	seqno	stream_id	delay	trace_status
1	-	1279398109685	3	7	3	0
2	17	1279398109681	3	3	3	0
3	18	1279398109682	3	6	2	0
4	16	1279398109680	3	5	2	0
5	14	1279398109678	3	2	0	0
6	15	1279398109678	3	4	0	0
7	9	1279398107678	2	4	0	0
8	3	1279398105684	1	4	9	0
9	13	1279398109678	3	1	0	2
10	13	1279398109678	3	1	0	2
11	7	1279398107678	2	1	0	2
12	1	1279398105675	1	1	0	2

(a)



(b)

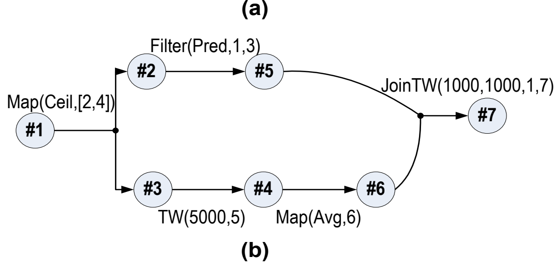
Fig. 8. The trace table (a) and the provenance graph (b)

algorithm (described in Figure 4) is applied to compose all stream ancestor functions dynamically. In Figure 8(a), the trace table, the internal table used to temporarily store intermediate results produced during the execution of the provenance query, is presented. In the trace table, the top row (index:#1) represents the event key of the provenance assertion which is the input of the provenance query, and the last four rows (index:#9-#12) represent the query results. In Figure 8(b), a provenance graph related to the trace table is presented. Each node labeled with an index number represents individual records in the trace table. By traversing the provenance graph in reverse, we can exactly identify that the events from the stream 1 (index:#9-#12) are the source events used in the production of the output event from the stream 7 (index:#1).

To demonstrate stream reproduction and validate provenance query results, we apply the replay execution algorithm (described in Figure 5) in order to compose all stream operations dynamically. Figure 9(a) shows the replay table -

index	assertion_id	timestamp	seqno	stream_id	delay	event_content	trace_status
1	13	1279398109678	3	1	0	93.28	0
2	14	1279398109678	3	2	0	94	0
3	15	1279398109678	3	4	0	94	0
4	16	1279398109680	3	5	2	[77,100,94]	0
5	17	1279398109681	3	3	3	94	0
6	18	1279398109682	3	6	2	90.33	0
7	-	1279398109685	3	7	3	(90.33,94)	2

(a)



(b)

Fig. 9. The replay table (a) and the processing graph (b)

the internal table used by the replay execution algorithm for processing the stream replay execution. In the replay table the top row (index:#1) represents the provenance assertion which is the input of the replay execution, and the last row (index:#7) represents the replay result. We also present a processing graph related to the replay table in Figure 9(b) to describe dependencies between each intermediate replay result. We start validating the query result of the previous provenance query by passing the query result (index:#1) to the replay execution algorithm. Then, the intermediate results produced during the processing of replay execution are stored in the replay table. Finally, we can derive the output event (index:#7). By comparing the output derived from this replay execution and the assertion input to the previous query, we can demonstrate the precision of our provenance query solution.

V. Experimental evaluation

In this section, we evaluate the implementation of our provenance solution - the fine-grained provenance model for streams. For all experiments, the experimental setup was as follows: Our provenance service and a stream processing system were hosted on a Linux PC with 1.60GHz Intel Xeon Quad Core CPU and 4 GB memory. To store provenance information, our implementation used MySQL 5 as a database backend and MyISAM is used as our storage engine. All our application components were implemented in Java. In addition, we use JMS as our messaging infrastructure with ActiveMQ 5 as the implementation choice.

A. Storage overheads for provenance collection

We now evaluate our fine-grained provenance model in terms of storage consumption for provenance collection. We compare storage space consumed by the implementation of our provenance model applying our storage reduction technique (optimized SAF) to another implementation that do not employ the reduction technique (unoptimized SAF). The synthetic processing flow used is a linear stream processing flow where stream components (stream operations) are chained together

TABLE III
Mathematical Symbols for storage formulas

Symbol	Definition
$SC_{SAF-unopt}$	Storage cost of the unoptimized ancestor function
$SC_{SAF-opt}$	Storage cost of the optimized ancestor function
$MC_{SAF-unopt}$	Marginal cost of the unoptimized ancestor function
$MC_{SAF-opt}$	Marginal cost of the optimized ancestor function
SS	The percentage of storage saved
k	Size of an event key
e	Size of an event's content
m	No. of messages fed to a stream processing system
c	No. of stream processing components

and each component takes input events from a previous component. In each set of experiments, we first measured the original storage cost of provenance recording (without applying the storage reduction technique). This experiment aims to demonstrate how much storage space the system requires to store every intermediate stream element. Then, we measured the storage cost resulting from the system that applies our reduction technique. By analyzing the storage measurements collected from these experiments, we can indicate the amount of storage saved when applying our reduction technique.

To understand the variation of storage overheads incurred by the system, the number of stream components used in the experiments was increased from 2 up to 15. Two message payload sizes, 100 Bytes and 1 Kbytes, were considered to demonstrate the storage overheads for different message sizes. In addition, for each test, the number of stream events fed to the stream system is 100,000 stream events.

The storage cost for provenance collection in our provenance solution can be explained in terms of some straightforward mathematical formulas. Table III lists the mathematical symbols used in our storage formulas. For the unoptimized stream ancestor function approach (SAF-unopt) which stores every intermediate stream elements, we can derive the storage cost from the following equation:

$$SC_{SAF-unopt} = ((k + e) * m) * c$$

For the optimized stream ancestor function approach (SAF-opt), only contents of stream elements that act as the first input to a stream processing system are recorded. The content of each intermediate event is discarded and only its event key is stored. Hence, the storage cost for provenance collection can be calculated as follows:

$$SC_{SAF-opt} = ((k + e) * m) + ((k * m) * (c - 1))$$

By utilizing the previous storage formulas, we can derive the storage saving rate (SS) from the following equation:

$$SS = \left(\frac{SC_{SAF-unopt} - SC_{SAF-opt}}{SC_{SAF-unopt}} \right) * 100$$

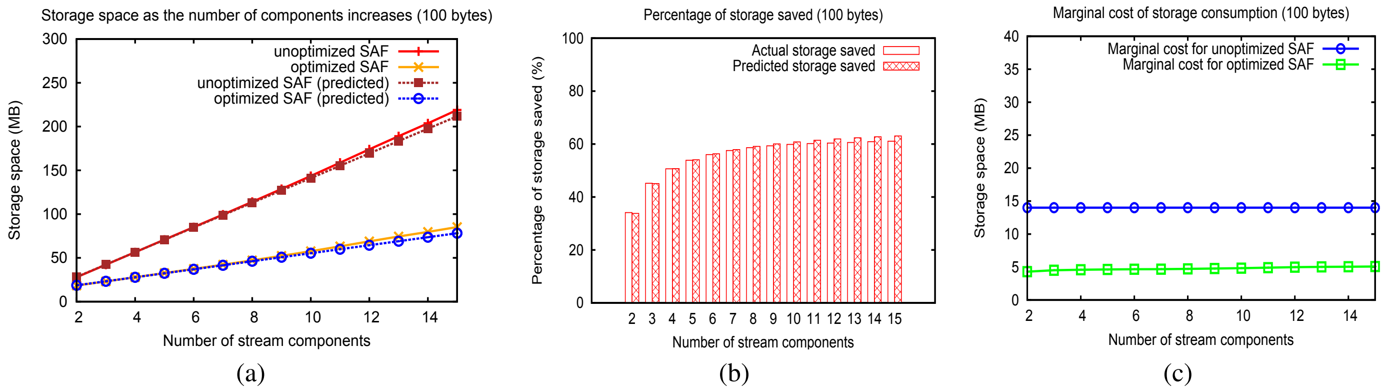


Fig. 10. Provenance storage cost for 100 bytes stream events as the number of stream components increases

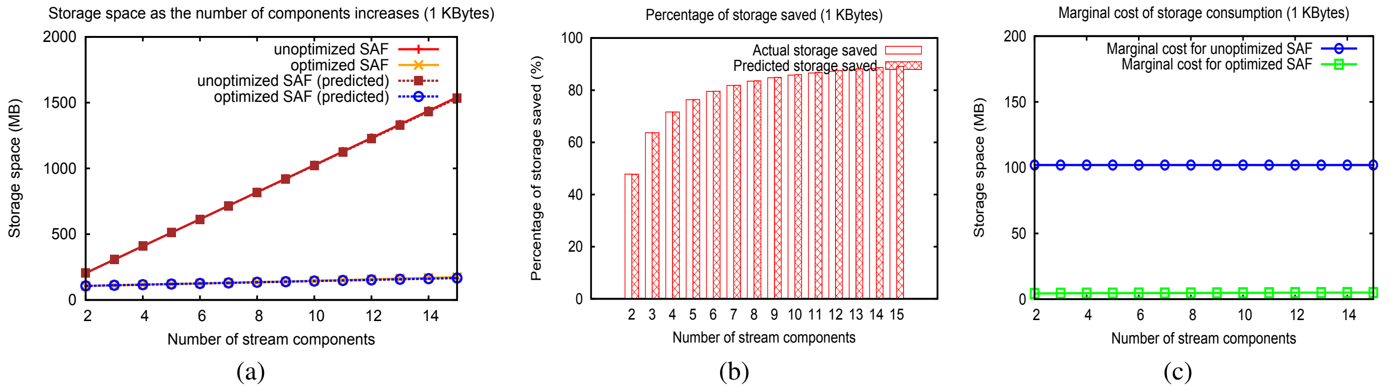


Fig. 11. Provenance storage cost for 1 KBytes stream events as the number of stream components increases

We can further utilize the storage measurement information to predict the marginal cost of storage consumption for provenance collection. In our context, we consider the marginal cost as the amount of storage space required for adding an additional component to a stream system. For the unoptimized stream ancestor function, we can derive the marginal cost of storage consumption from the following equation:

$$MC_{SAF-unopt} = (k + e) * m$$

For the optimized stream ancestor function, the marginal cost of storage consumption can be calculated as follows:

$$MC_{SAF-opt} = (k * m)$$

Figure 10(a) and 11(a) show the storage cost needed to store provenance information for different provenance storage approaches and different payload sizes (message sizes). We show the storage cost observed for the synthetic processing flow applying our storage reduction technique (optimized SAF) and the other that does not employ the storage reduction technique (unoptimized SAF). The predicted storage cost of both approaches derived from our storage formulas are also presented. In the both figures, the storage cost of unoptimized SAF grows significantly because both event keys and event contents for every intermediate events need to be stored in a provenance store. On the other hand, compared to the optimized SAF which applies the storage reduction technique, the amount of storage consumed by the stream system is just slightly increased. This is because many event contents for every intermediate event are discarded.

The storage saving rates shown in Figure 10(b) and 11(b) indicate that our storage reduction technique (optimized SAF) is extremely effective when dealing with large message sizes. For instance, at 10 stream components, the percentage of storage space we can save from discarding event contents of intermediate stream elements with 1 Kbytes payloads is almost 85 percent. It is much higher than the percentage of storage saved for 100 bytes payloads stream events at the same number of stream components which is about 60 percent. This finding shows that the bigger the message size that a stream processing system exploits, the greater the storage overheads can be saved by our storage reduction technique.

Furthermore, the marginal costs of storage consumption in Figure 10(c) and 11(c) indicates that our storage reduction solution can economize the storage cost for provenance collection when a stream processing system is scaled up. In the both figures, the marginal cost for unoptimized SAF and that for optimized SAF remain stable when the number of stream components increases. However, the fixed rate of the marginal cost for optimized SAF is considerably less than that for unoptimized SAF. Compared to unoptimized SAF, the

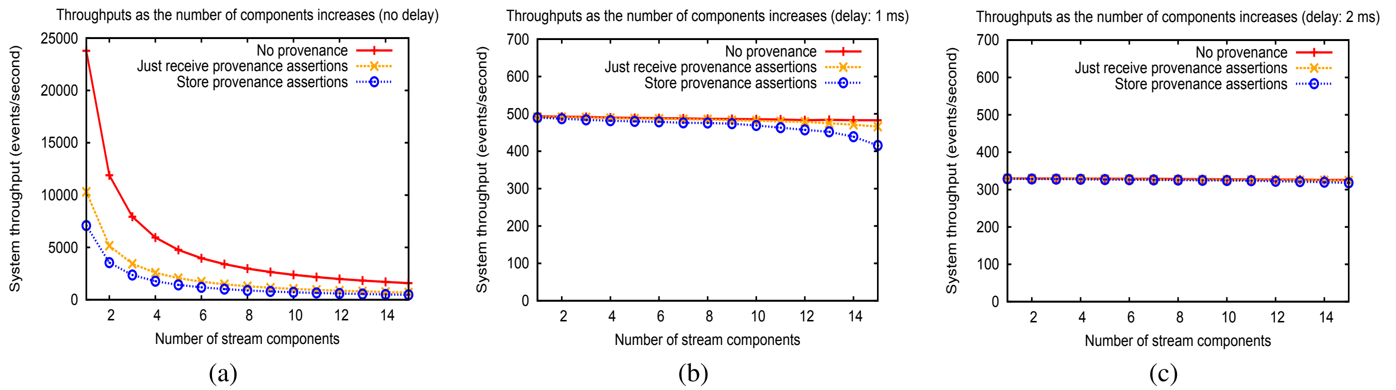


Fig. 12. System throughputs of the implementation of our stream provenance system as time delays for stream processing increase

percentage of the marginal costs reduced by the optimized SAF approach for 100 bytes payloads stream events is around 65 percent and that for 1 Kbytes payloads stream events is about 95 percent. These results show the substantial reduction in storage consumption when stream components increase. The results do not only present the fixed rate of storage costs but also indicate that the marginal costs for the optimized SAF does not depend on the size of stream event. Therefore, with the considerably smaller and fixed marginal costs, practical storage cost control can be greatly maintained by application developers when a stream system needs to be scaled up.

B. Provenance recording impact

The purpose of this evaluation is to observe the impact of provenance recording on a stream processing system in a controlled environment. In this evaluation, system throughput - the number of messages (stream events) processed by a stream system over a given interval of time - is used as our performance indicator. Similar to the storage experiments, the synthetic processing flow used is a linear processing flow.

In each set of experiments, we first measured the system throughput of the implementation of a stream processing system that does not record provenance information. This experiment aims to demonstrate how much is the system throughput that we can expect under normal processing. Then, we measured the system throughput of a stream processing system that records provenance information for different provenance processing modes of the provenance service including Just receive provenance assertions (without any provenance processing by the provenance service) and Store provenance assertions into a provenance store. This experiment aims to present the effect of provenance recording on the system throughput of a stream processing system.

To understand the impact of provenance recording when a stream processing system is scaled up, the number of stream components (stream operations) used in the experiments was increased from 2 up to 15. In addition, because different stream-based applications have different time delays for processing depending on how fast stream operations can be executed or more particularly the complexity of stream computation performed, therefore three different time delays for stream processing - no delay, 1ms and 2ms - were considered as significant parameters in the experiments. With the introduction of various time delays for processing, we can evaluate the effectiveness of our provenance system when dealing with stream systems at different levels of complexity.

Figure 12(a) displays the system throughputs of our implementations with no time delays in processing, as a number of stream components increases. The figure shows a significant drop-off in the system throughput of the implementation that does not record provenance information from the maximum throughput (around 23,000 messages/second). Similar but significant lower trends in system throughput were observed for the other implementations that record provenance information. At the same number of stream components, the throughput decreases more than 50 percent for all implementations compared to the case of “no provenance” implementation. We determined that this degradation is due to the introduction of provenance recording functionality which doubles the number of data streams maintained by the message broker software.

Figure 12(b) and 12(c) demonstrate the system throughputs of our implementations that increase time delays for processing. The time delays are increased from no delay to 1ms and 2ms respectively. In Figure 12(b), all system throughputs significantly drop from that in “no processing delay” experiment (shown in Figure 12(a)) as expected due to the introduction of time delay 1ms. The overall trends of the system throughputs for our implementations using different provenance processing modes has been changed by the introduction of time delay as well. The system throughput of store provenance assertions implementation gradually decreases when a number of stream components increases. This degradation of the system throughputs is considerably smaller compared to the reduction of the system throughputs in “no processing delay” experiment. In addition, as shown in Figure 12(c), the trends of the system throughputs for all our implementations are almost flat and there are almost no significant difference between the system throughputs of “no provenance” implementation and that of store provenance assertions implementation. The more the time delay for processing increases the more the processing overheads for provenance recording can be reduced. As a result, we conclude that the processing overheads caused by

our provenance solution can be greatly reduced when time delay of a stream processing system is large.

Furthermore, considering the percentage of the processing overheads incurred by our provenance solution, the average processing overheads for the 'no processing delay" experiment are excessively high - about 70 percent for store provenance assertions approach. On the other hand, when the time delays for processing are introduced (1ms and 2ms), the average processing overheads are significantly reduced to be less than 10 percent for our provenance recording approach. Therefore, with the experimental results, we can establish that the impact of provenance recording is relatively small or more particularly it generally does not have a significant effect on the normal processing of stream systems. In addition, our provenance solution is more suitable for stream-based applications that process slightly low-rate data streams (e.g. greater than 1ms per event used in our experiments) due to the fact that the impact of provenance recording is minimal.

VI. Conclusion and Future Work

In this paper, we presented a novel fine-grained provenance solution - the stream ancestor function - that enables stream processing systems to precisely capture dependency relationships for every individual stream element. We have also demonstrated that, by utilizing stream ancestor functions, fine-grained provenance query and stream reproduction functionality for stream processing systems can be facilitated. To deal with a practical storage issue, we introduced an enhanced solution called the optimized stream ancestor function which can significantly reduce the amount of storage consumed for provenance collection and eliminate the requirement for storing every intermediate stream element. The experimental evaluation demonstrated that our stream provenance solution enables the fine-grained provenance problem in stream processing systems to be addressed with reasonable storage consumption and acceptable processing overheads.

Although we have proposed the fine-grained provenance solution for streams that offers reasonable overheads, there are practical challenges related to the unique characteristic of streams we plan to address. Our ongoing work is to design stream-specific provenance queries that can be performed on-the-fly over streams of provenance assertions. These queries should exploit our stream ancestor functions and compose them dynamically without requiring additional storage space. With this enhanced solution, we believe that our provenance solution can offer very low processing and storage overheads for provenance collection in stream processing systems.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, Madison, Wisconsin, 2002, pp. 1–16.

[2] S. Bowers, T. M. McPhillips, and B. Ludascher, "Provenance in collection-oriented scientific workflows," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 519–529, 2008.

[3] P. Buneman, S. Khanna, and W.-C. Tan, "Why and where: A characterization of data provenance," in *the 8th International Conference on Database Theory*, ser. LNCS 1973, 2001, pp. 316–330.

[4] ——, "On propagation of deletions and annotations through views," in *the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. Symposium on Principles of Database Systems, Madison, Wisconsin, 2002, pp. 150 – 158.

[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams a new class of data management applications," in *the 28th international conference on Very Large Data Bases*. Hong Kong, China: VLDB Endowment, 2002, pp. 215–226.

[6] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "Dbnotes: A post-it system for relational databases based on provenance," in *the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2005, pp. 942–944.

[7] Y. Cui and J. Widom, "Tracing the lineage of view data in a warehousing environment," *ACM Transactions on Database Systems*, vol. 25, no. 2, pp. 179–227, 2000.

[8] ——, "Lineage tracing for general data warehouse transformations," *The VLDB Journal*, vol. 12, no. 1, p. 4158, 2003.

[9] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludaescher, T. M. McPhillips, S. Bowers, M. K. Anand, and J. Freire, "Provenance in scientific workflow systems," *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 44–50, 2007.

[10] Esper, "Espertech: Esper reference documentation," Technical Report, Version 2.0.0, retrieved March, 2008. [Online]. Available: http://esper.codehaus.org/

[11] L. Golab and M. T. Ozsu, "Issues in data stream management," *ACM SIGMOD Record*, vol. 32, no. 2, pp. 5–14, 2003.

[12] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. Symposium on Principles of Database Systems, 2007, pp. 31–40.

[13] P. Groth, S. Miles, and L. Moreau, "A Model of Process Documentation to Determine Provenance in Mash-ups," *Transactions on Internet Technology (TOIT)*, vol. 9, no. 1, pp. 1–31, 2009.

[14] P. Groth and L. Moreau, "Recording process documentation for provenance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 9, pp. 1246 –1259, 2009.

[15] A. Misra, M. Blount, A. Kementsietsidis, D. Sow, and M. Wang, "Advances and challenges for scalable data provenance in stream processing systems," in *Proceedings of Second International Provenance and Annotation Workshop (IPAW'08)*, ser. LNCS 5272. Salt Lake City, Utha, USA: Springer, 2008, pp. 253–265.

[16] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche, "The open provenance model — core specification (v1.1)," *Future Generation Computer Systems*, Jul. 2010.

[17] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[18] SouthamptonCityCouncil, "Port of southampton off-site reactor emergency plan," SotonSafe report, Version 4, 2006.

[19] M. Stonebraker, U. etintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[20] W.-C. Tan, "Provenance in databases: Past, current, and future," *IEEE Data Engineering Bulletin*, vol. 30, no. 4, pp. 3–12, 2007.

[21] N. Vijayakumar and B. Plale, "Towards low overhead provenance tracking in near real-time stream filtering," in *International Provenance and Annotation Workshop (IPAW'06)*, ser. LNCS 4145. Chicago, Illinois: Springer, 2006, pp. 46–54.

[22] ——, "Tracking stream provenance in complex event processing systems for workflow-driven computing," *Second Int'l Workshop on Event-driven Architecture, Processing, and Systems (EDA-PS'07), in conjunction with VLDB'07*, 2007.

[23] 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 *International Conference On Mobile Systems, Applications And Services*, San Juan, Puerto Rico, 2007, pp. 95–100.

[24] A. Woodruff and M. Stonebraker, "Supporting fine-grained data lineage in a database visualization environment," in *the 13th International Conference on Data Engineering (ICDE)*, 1997, pp. 91–102.