An On-the-fly Provenance Tracking Mechanism for Stream Processing Systems

Watsawee Sansrimahachai
School of Science and Technology
University of the Thai
Chamber of Commerce
Email: watsawee_san@utcc.ac.th

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

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

***Abstract*—Applications that operate over streaming data with high-volume and real-time processing requirements are becoming increasingly important. These applications process streaming data in real-time and deliver instantaneous responses to support precise and on-time decisions. In such systems, traceability - the ability to verify and investigate the source of a particular output - in real-time is extremely important. This ability allows raw streaming data to be checked and processing steps to be verified and validated in timely manner. Therefore, it is crucial that stream systems have a mechanism for dynamically tracking provenance - the process that produced result data - at execution time, which we refer to as on-the-fly stream provenance tracking. In this paper, we propose a novel on-the-fly provenance tracking mechanism that enables provenance queries to be performed dynamically without requiring provenance assertions to be stored persistently. We demonstrate how our provenance mechanism works by means of an on-the-fly provenance tracking algorithm. The experimental evaluation shows that our provenance solution does not have a significant effect on the normal processing of stream systems given a 7% overhead. Moreover, our provenance solution offers low-latency processing (0.3 ms per additional component) with reasonable memory consumption.**

I. Introduction

Major changes in daily life have been caused by recent advancements in micro-sensor and wireless communication technologies. The functionality and usability of sensor technologies enables several kinds of sensors to be deployed in a wide variety of environments. The price of these devices is becoming cheaper and sensors are increasingly considered as commodity products that anyone can afford to buy. This will lead to a significant increase of wide range environment monitoring and control applications that operate over streaming data with high-volume and real-time processing requirements.

There are a number of significant requirements that these kinds of systems need to satisfy [12]. The first is that a stream system needs to process data streams in real-time to support a precise and on time decision. The second requirement is that a stream system must be able to process stream events on-the-fly without any requirement to store them. With these requirements, traceability - the ability to verify and investigate the source of a particular output stream element - is extremely important. Stream systems that do not provide provenance information - the information pertaining to the process that led to result data - can suffer from problems of traceability.

Imagine that in a radioactivity leak incident in a nuclear submarine, emergency services operators rely on a real-time mapping (GIS) application to manage the disaster. The information displayed on the GIS application is submitted by sensors located near the scene of the incident. The sensor measurements are also forwarded to an early warning system where predictions of a possible 'dirty bomb" event are made in real-time. Because extreme weather conditions, some sensors were damaged and they continuously submit their faulty measurements into the GIS system. At some point, an operator has received a report indicating that there is an explosion in the nuclear reactor. The operator questions why this explosion was not automatically detected and why the level of radioactive material shown on the map display was not classified as being potentially dangerous radioactive intensity levels. If the stream systems have support for a provenance functionality that can be operated dynamically in real-time, this would allow the operator to validate processing results in a timely manner, trace back incorrect information to its origin and also verify sequence of processing steps used to produce those results.

To address such a provenance challenge in stream systems, we introduce a novel on-the-fly provenance tracking mechanism. The key concept is that we exploit a provenance service as a stream component. Provenance assertions – assertions pertaining to provenance recorded by each stream operation for individual stream elements – are processed dynamically without any requirement to store them persistently. We extend the persistent provenance mechanism presented in our previous work [11] by introducing the idea of property propagation. By utilizing a new version of stream ancestor functions - reverse mapping functions used to express dependencies between input and output stream elements, properties can be propagated and provenance query results are produced as a stream.

This paper makes the following key contributions:

- It presents a novel on-the-fly provenance tracking mechanism for stream processing systems.
- It defines a set of stream ancestor functions designed to work with the on-the-fly provenance mechanism.
- It proposes a novel on-the-fly provenance tracking algorithm for stream processing systems.
- It presents the performance characteristics of our provenance solution for streams.

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 an on-the-fly provenance tracking mechanism for streams. Section 4 presents an on-the-fly provenance tracking algorithm. Section 5 demonstrates the experimental evaluation in terms of the impact of provenance recording, the memory consumption and the time latency. Finally, Section 6 presents conclusions.

II. Related work

Considerable research efforts have been made by the database community to address the provenance problem. Woodruff and Stonebraker [15] proposed a technique called weak inversion used to regenerate input data items that produced an output. The drawback is that the answer returned by this function is not guaranteed to be perfectly accurate and in several cases it is not possible to define inversion functions. Buneman *et al.* [1] 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, a technique that propagates annotation according to *propagation rules* [2] has been proposed.

In the context of scientific work flow systems, Taverna [10] provides support for provenance tracking to allow scientists to understand how results from experiments were obtained. Provenance information is collected by recording metadata and intermediate results during workflow enactment. Another system, PASOA [6], [7], built an infrastructure for reasoning over provenance in service-oriented architectures. By recording assertions comprising interaction messages and causal relationships between messages, provenance of data products can be captured. Our provenance solution extends the PASOA provenance mechanism. In our solution, a stream operation is treated as a "grey box" [4] and provenance is collected based on input-output dependencies 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 burden problem.

In the area of distributed stream processing, Vijayakumar *et al.* [14] propose a coarse-grained provenance model that identifies dependencies between streams or sets of stream elements as the smallest unit for which provenance is collected. However, the level of granularity for capturing provenance in this model is not detailed enough to identify dependencies among individual stream elements. The *Time-Value-Centric* model [9] (TVC) is a finer-grained provenance solution that provides the ability to express dependencies for individual stream elements by utilizing input-output dependencies described in terms of three primitive invariants: time, value and sequence. Nevertheless, this model still has some limitations, since it can fail to identify precisely input elements used in the production of an output. Another limitation is that because all intermediate stream elements need to be stored for querying, this potentially leads to a storage problem when dealing with high volume data streams. Another similar model [8] has the same limitation as TVC as well.

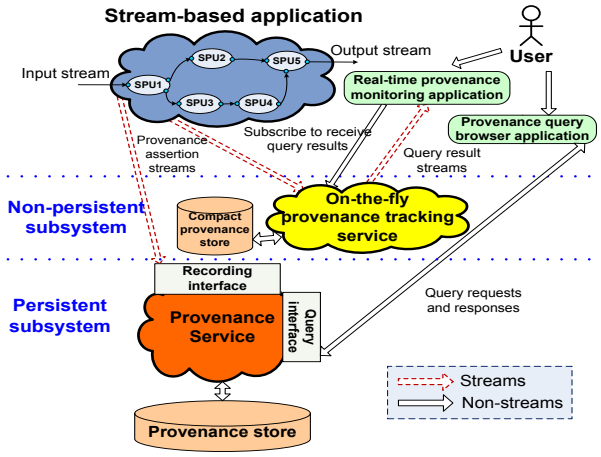

Fig. 1. The provenance architecture for streams

To sum up, the approach 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 stream elements. It identifies a set of stream operations for which on-the-fly provenance tracking can be computed. It applies eager approach [13], since it propagates provenance-related information at runtime to obtain provenance tracking results in real-time. Furthermore, it eliminate the storage problem caused by storing provenance information.

III. On-the-fly provenance tracking

To design an on-the-fly provenance tracking mechanism, we make the following assumptions describing stream processing systems that our provenance mechanism is intended to support.

- A stream processing system is represented as a set of interconnected nodes, with each node representing a stream operation.
- Each data stream consists of a sequence of time ordered stream events and each individual stream event is composed of an event key - a unique reference of an individual event - and a content of stream event (data).
- Each individual stream element is associated with one or more properties. A property in our context is a piece of information describing a quality, characteristic or attribute that belongs to an individual stream element.

To overview our stream provenance system, we present a provenance architecture for streams in Figure 1. In this paper we focus on the non-persistent subsystem. An on-the-fly provenance tracking service plays a central role in this subsystem. It is designed to be utilized as a stream component that executes continuous queries over continuous data streams. During execution time, provenance assertions are recorded as a stream by each stream operation for each individual stream element. The provenance assertions are not generated at a single

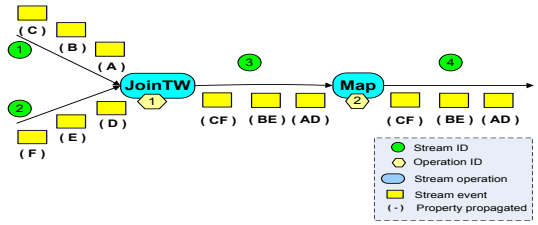

Fig. 2. An example of property propagation in a stream processing system

time, but instead their generation is interleaved continuously with execution. As provenance assertions are being received by the on-the-fly provenance tracking service, provenance queries are performed continuously over the streams of provenance assertions. Note that the generation of on-the-fly provenance queries is based on configuration parameters and stream topology information specified during system registration time. Such information is stored in a compact provenance store - a compact version of the provenance store.

A. Fundamental concept of on-the-fly provenance tracking

We extend the concept of *Stream Ancestor Function* (SAF) in our previous work [11] by adding stream-specific techniques designed specifically for addressing on-the-fly provenance tracking. Our key concept is inspired by the idea of property propagation. We assume that each individual stream element contains a provenance-related property in its accumulator - a field used to accumulate property computing results. For each stream operation, properties are computed and propagated automatically from input streams to output streams based on input-output dependencies between stream elements. Each intermediate result produced by the processing of provenance-related properties is temporarily stored in the accumulator field of intermediate stream elements. The processing of property propagation is performed continuously until reaching the final stream operation of a stream processing flow. Figure 2 illustrates an example of our property propagation approach in which properties are propagated through a stream processing flow. However, we do not try to propagate properties within a stream system layer because this would require the modification of the internal processing of stream operations, but instead properties are computed and propagated through the use of provenance assertions inside a provenance service.

An example process of our on-the-fly provenance tracking inside the provenance service is shown in Figure 3. The execution begins with the provenance service receiving streams of provenance assertions (AS) generated by stream operations in a stream system. Each assertion is detected by an assertion separation unit (ASU) - a component used to detect provenance assertions for a particular stream and direct them to a SAF that they are associated with. After that each individual assertion is computed by its associated SAF. The SAF utilized in our on-the-fly provenance tracking is the new version of the original SAF - called a property stream ancestor function (PSAF). It receives stream elements from both an assertion stream (AS) and result streams (RS) - the output from the

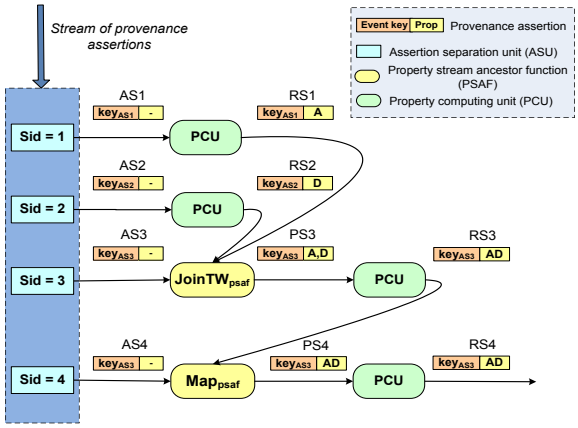

Fig. 3. On-the-fly provenance tracking inside the provenance service

previous step of property propagation - as an input and then it produces an output element belonging to a property stream (PS). Not only is the PSAF used to identify the ancestors of a particular provenance assertion, it is also used to extract properties from the ancestor assertions (elements from RS). The output generated from the PSAF is fed into a PCU to compute property propagation. Once the property propagation is processed, an output provenance assertion containing a new property is produced as an element of a result stream (RS).

B. Primitive stream processing operations

We now present the primitive stream operations that on-the-fly provenance tracking can be computed. These operations are recognized as common operations developed in several stream projects [5], [3]. Note that all stream operations are defined by using Event Processing Language (EPL) - an SQL-like language extended to handle event streams. We use EPL provided by Esper stream engine to illustrate how continuous queries are formulated in stream operations and how provenance-related properties can be computed and propagated.

- **Map** *Map(F,sid):* A map operation is a stream operation that 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 a 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 is defined in terms of the number of events. At any point in time, a length window covers the most recent l events 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 ($w1$,$w2$) are combined and output events are produced according to a join condition.

TABLE I
THE DEFINITIONS OF PRIMITIVE STREAM PROCESSING OPERATIONS

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

TABLE II
THE DEFINITIONS OF PROPERTY STREAM ANCESTOR FUNCTIONS (PSAFS)

	Property stream ancestor functions *(input: AS/RS - Assertion(key(t,n,s,d),prop))*
$Map_{psaf}()$	$(* \ fn \ : \ 'a \ ASSERTION \ list \ \rightarrow \ 'b \ ASSERTION \ \rightarrow \ 'a \ ASSERTION \ *)$ $insert \ into \ \texttt{PS}(timestamp, seqNo, streamID, delay, property)$ $select \ t, n, s, d, PExtract.compute((select \ * \ from \ \texttt{RS}), t \ - \ d)$ $from \ \texttt{AS}$
$Filter_{psaf}()$	$(* \ fn \ : \ 'a \ ASSERTION \ list \ \rightarrow \ 'b \ ASSERTION \ \rightarrow \ 'a \ ASSERTION \ *)$ $insert \ into \ \texttt{PS}(timestamp, seqNo, streamID, delay, property)$ $select \ t, n, s, d, PExtract.compute((select \ * \ from \ \texttt{RS}), t \ - \ d)$ $from \ \texttt{AS}$
$TW_{psaf}(w)$	$(* \ fn \ : TIME \ * \ 'a \ ASSERTION \ list \ \rightarrow \ 'b \ ASSERTION \ \rightarrow \ 'a \ ASSERTION \ *)$ $insert \ into \ \texttt{PS}(timestamp, seqNo, streamID, delay, property)$ $select \ t, n, s, d, PExtract.compute((select \ * \ from \ \texttt{RS}), t \ - \ d, t \ - \ d \ - \ \texttt{w})$ $from \ \texttt{AS}$
$LW_{psaf}(l)$	$(* \ fn \ : \ int \ * \ 'a \ ASSERTION \ list \ \rightarrow \ 'b \ ASSERTION \ \rightarrow \ 'a \ ASSERTION \ *)$ $insert \ into \ \texttt{PS}(timestamp, seqNo, streamID, delay, property)$ $select \ t, n, s, d, PExtract.compute((select \ * \ from \ \texttt{RS}), n, n \ - \ \texttt{l} \ + \ 1)$ $from \ \texttt{AS}$
$JoinTW_{psaf}(w1, w2)$	$(* \ fn \ : \ TIME \ * \ TIME \ \rightarrow \ 'a \ ASSERTION \ list \ \rightarrow \ 'a \ ASSERTION \ list$ $\rightarrow \ 'b \ ASSERTION \ \rightarrow \ 'a \ ASSERTION \ *)$ $insert \ into \ \texttt{PS}(timestamp, seqNo, streamID, delay, property)$ $select \ t, n, s, d$ $\quad PExtract.compute((select \ * \ from \ \texttt{RS1}), t \ - \ d, t \ - \ d \ - \ \texttt{w1}) \ \|\|$ $\quad PExtract.compute((select \ * \ from \ \texttt{RS2}), t \ - \ d, t \ - \ d \ - \ \texttt{w2})$ $from \ \texttt{AS}$
$JoinLW_{psaf}(l1, l2)$	$(* \ fn \ : \ int \ * \ int \ \rightarrow \ 'a \ ASSERTION \ list \ \rightarrow \ 'a \ ASSERTION \ list$ $\rightarrow \ 'b \ ASSERTION \ \rightarrow \ 'a \ ASSERTION \ *)$ $insert \ into \ \texttt{PS}(timestamp, seqNo, streamID, delay, property)$ $select \ t, n, s, d,$ $\quad PExtract.compute((select \ * \ from \ \texttt{RS1}), \texttt{l1}, t \ - \ d) \ \|\|$ $\quad PExtract.compute((select \ * \ from \ \texttt{RS2}), \texttt{l2}, t \ - \ d)$ $from \ \texttt{AS}$

- **Length-window join** *JoinLW(l1,l2,sid):* Similar to the time-window join, a length-window join pairs events from two input streams. The difference between these two operations is the length-window join operates on stream events from two tuple-based windows ($l1$,$l2$).

The definitions of the primitive stream operations are described in Table I. Note that, a stream id (sid) for every operation is the ID of an output stream.

C. Property stream ancestor functions

In this section, "Property stream ancestor functions" (PSAFs) used for on-the-fly provenance tracking are described. As presented in Table II, every PSAF utilizes internal values of its associated stream operation as input parameters at registration time. For example, the PSAF for a time-window (TW_{psaf}) utilizes a duration of the time window (w) as a parameter. Each PSAF takes a provenance assertion (AS) and the elements of a result stream (RS) - a stream generated from the previous property propagation step - as an input and generates an element of a property stream (PS) containing all provenance-related properties needed for further processing.

In addition, the $PExtract$ function is used by each PSAF to extract properties from elements in a result stream (RS). In the PSAFs for windowed operations (e.g. TW and LW), the $PExtract$ function utilize the size of a data window and a delay time for processing to define the extent of a past data window at the time that the stream operation produced the output. This extent of a past data window is then used to identify the ancestor provenance assertions (the elements in RS) and also to extract provenance-related properties from them. For example, the definition of the PSAF for a sliding time-window (TW_{psaf}) indicates that the $PExtract$ function extracts properties from elements in RS by creating the extent of a past time-window where the interval of the window is between *t - d* (upper bound) and *t - d - w* (lower bound).

IV. An on-the-fly provenance tracking algorithm

We now demonstrate how the on-the-fly provenance mechanism works by means of an on-the-fly provenance tracking algorithm. The main concept of the algorithm is that it utilizes stream topology information and operation parameters to automatically create the internal processing of on-the-fly provenance queries. The algorithm is presented in Figure 4.

As shown in Figure 4, the algorithm consists of two main functions: $OTFpquery$ and $composePSAF$. The $OTFpquery$ is the entry-point function that takes a list of provenance assertions ($paList$), a list of stream IDs ($sidList$ - used to terminate the query execution) as input parameters and returns provenance tracking results ($resultList$). The other function - $composePSAF$ - is the recursive function containing the business logic of the algorithm. The function begins by receiving parameters passed by the $OTFpquery$. For each assertion (pa), it is first processed by an assertion separation unit (ASU) to create an extra field for storing properties. Then, in the case that input assertions belong to the first-input stream, they are executed by a property computing

```
1: /* ASSERTION list * INT list -> ASSERTION list */
2: Function OTFpquery(
3:   paList: ASSERTION list,
4:   sidList: INT list){
5:     resultList = []:ASSERTION list;
6:     bufferList = []:ASSERTION list;
7:     composePSAF(paList,sidList,bufferList,resultList);
8:     return resultList;
9: }
```

```
1: Function composePSAF(
2:   paList: ASSERTION list,
3:   sidList: INT list,
4:   bufferList: ASSERTION list,
5:   resultList: ASSERTION list){
6: elm = paList.removeElement();
7: pa = ASU.execute(elm);
8: if pa.sid is FirstInputStream then
9:   pa'' = PCU.execute(pa);
10:   bufferList.add(pa'');
11: else
12:   psaf = getPSAF(pa.sid);
13:   pa' = executePSAF(bufferList,psaf,pa);
14:   Dequeue(bufferList,psaf,pa);
15:   pa'' = PCU.execute(pa');
16:   bufferList.add(pa'');
17: end if
18: if pa.sid ∈ sidList then
19:   resultList.add(pa'');
20: end if
21: if paList ≠ empty then
22:   composePSAF(paList,sidList,bufferList,resultList);
23: end if
24: }
```

Fig. 4. On-the-fly provenance tracking algorithm

unit (PCU) to extract properties. After that, each assertion will be processed by its associated PSAF and PCU. Every output of property computing (pa'') is inserted to a data buffer ($bufferList$). Finally, the $composePSAF$ function will be called recursively until no assertions remain in the $paList$.

V. Experimental evaluation

In this section, we evaluate the implementation of our on-the-fly provenance tracking mechanism. Our evaluation is conducted across three different aspects, including the provenance recording impact, the memory consumption for a provenance service and the time latency for on-the-fly provenance tracking. For all experiments, the experimental setup was as follows: Our provenance service and a stream system were hosted on a Linux PC with 1.60GHz Intel Xeon Quad Core CPU and 4 GB memory. All our application components were implemented in Java. We use JMS as our messaging infrastructure with ActiveMQ 5 as the implementation choice.

A. Provenance recording impact

In this evaluation, system throughput - the number of stream events processed over a given interval of time - is used as our performance indicator. In each set of experiments, we first measured the throughput of a stream system that does not record provenance information. Then, we measured the throughput of a stream system that records provenance information for different modes of the provenance service including 1) Just receive provenance assertions, 2) Store provenance assertions and 3) Perform on-the-fly provenance queries. In addition, because different stream-based applications have different time delays for processing depending on the complexity

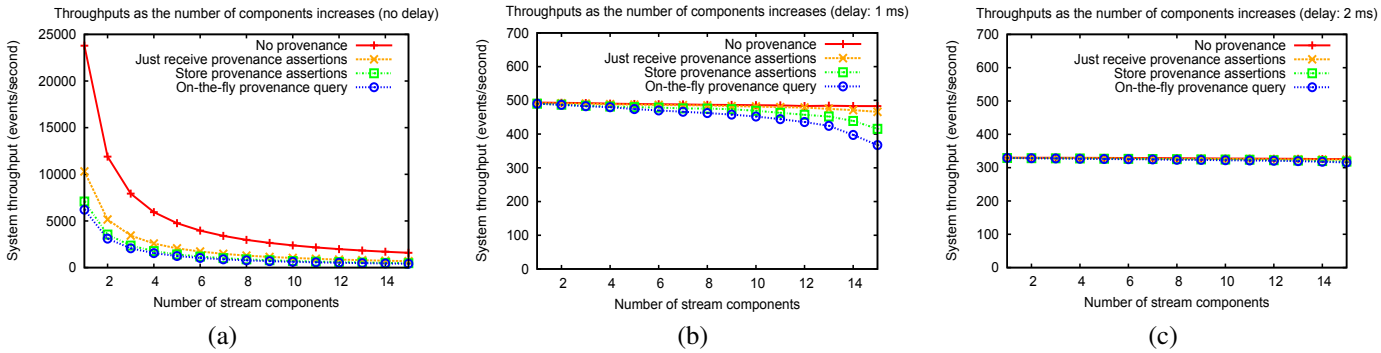

Fig. 5. System throughputs of the implementation of our stream provenance system as time delays for stream processing increase

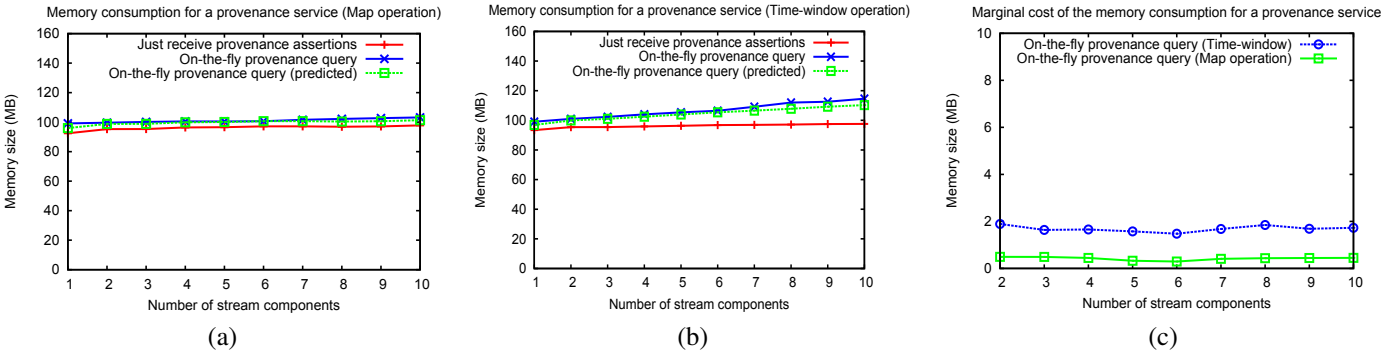

Fig. 6. Memory consumption of the implementation of an on-the-fly provenance service

of stream computation performed, therefore three different time delays for processing - no delay, 1ms and 2ms - were considered as significant parameters in the experiments.

Figure 5(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 5(b) and 5(c) demonstrate the system throughputs of our implementations that increase time delays for processing. In Figure 5(b), all system throughputs significantly drop from that in "no processing delay" experiment (shown in Figure 5(a)) as expected due to the introduction of time delay 1ms. The system throughput of all our implementations gradually decreases when a number of stream components increases. In addition, as shown in Figure 5(c), there are almost no significant difference between the throughputs of "no provenance" implementation and that of the on-the-fly provenance implementation. As a result, we conclude that the processing overheads caused by our provenance solution can be greatly reduced when the time delay of a stream system is large.

Furthermore, considering the percentage of the processing overheads incurred by our provenance solution, the average overheads for the 'no processing delay" experiment are excessively high - about 75% for the on-the-fly provenance approach. However, when the time delays for processing are introduced, the overheads are significantly reduced to be around 7%. 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.

B. Memory consumption for a provenance service

This evaluation aims to examine the effect of the on-the-fly provenance processing on the normal processing of a provenance service, with respect to memory consumption. In each set of experiments, we first measured the memory space used by the provenance service when provenance assertions are just collected but not executed. Then, we measured the memory space used by the provenance service that is operated on the on-the-fly provenance query mode. To investigate the change of the memory consumption, two different stream operations are considered: Map and Time-window operations.

Figures 6(a) and 6(b) present the memory space consumed for the map operations and the time window operations experiments respectively. The memory size for the on-the-fly provenance approach is slightly higher in Figure 6(a), and significantly higher in Figure 6(b) as the number of stream components increases, compared to that for the baseline (Just

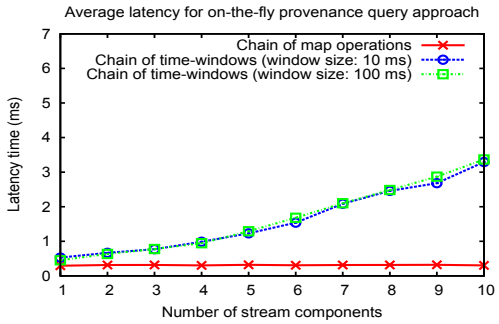

Fig. 7. Average time latency for on-the-fly provenance query approach

receive provenance assertions). We determine this increase of memory size is due to the processing of on-the-fly provenance tracking that needs to store assertions in the memory.

Moreover, as shown in Figures 6(c), the overall trend of the marginal cost of memory space consumed remains stable for both the map operation and the time-window experiments. The result shows that the marginal cost is relatively low compared to the total memory consumption. For example, the average marginal cost in the map operation experiment is slightly less than 0.5 MB and that in the time window experiment is about 1.8 MB. Therefore, we conclude that the memory consumption for a provenance service may vary based on the types of stream operations and the size of data windows used in a stream system, but the marginal cost of memory space consumed for our provenance solution is relatively low and reasonable.

C. Latency for on-the-fly provenance tracking

We now examine the runtime overheads for our provenance solution by measuring the time latency for on-the-fly provenance tracking. In this context, the latency is defined as a period of time difference between a stream system produces a result and a corresponding provenance-related property is computed. In this experiment, three different chains of stream operations are considered: map operations, time-windows of size 10ms and time-windows of size 100ms. The reason behind the use of different operations is because for some operations (e.g. time windows), more than one properties have to be computed. This potentially results in delayed query results.

Figure 7 displays the average time latency of the implementation of our on-the-fly provenance solution. In the figure, as the number of components increases, the latency for the chain of map operations remains stable. On the other hand, the time latency for both chains of time-window operations increases significantly. This can be explained that for windowed operations, internal data buffers of a stream engine are used to store intermediate provenance assertions during execution. This results in an increase of the average delay time in processing. From the latency graph, the average time latency increased per additional stream component is about 0.3 ms. Therefore, with a relatively low latency per additional component, we can establish that our provenance solution offers low-latency processing and our solution can provide provenance tracking results in real-time.

VI. Conclusion

In this paper, we presented a novel on-the-fly provenance tracking mechanism, which performs provenance queries dynamically over streams of provenance assertions without requiring the assertions to be stored persistently. We have discussed the important characteristics of the on-the-fly provenance tracking and also defined the specifications of property stream ancestor functions (PSAFs). To demonstrate how our provenance mechanism works in practice, we have presented an algorithm for on-the-fly provenance tracking. The experimental evaluation demonstrated that our provenance solution enables the *on-the-fly provenance tracking* problem in stream processing systems to be addressed with acceptable performance and reasonable overheads. A 7% overhead is observed as the impact of provenance recording on system performance. Moreover, our on-the-fly provenance approach offers low-latency processing (average latency: 0.3 ms per additional component) with reasonable memory consumption.

References

[1] P. Buneman, S. Khanna, and W. C. Tan, "Why and where: A characterization of data provenance," in *Proceedings of ICDT'01*, 2001, pp. 316–330.

[2] P. Buneman, S. Khanna, and W.-C. Tan, "On propagation of deletions and annotations through views," in *Proceedings of PODS'02*, 2002, pp. 150–158.

[3] D. Carney, U. Çetintemel, 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 *Proceedings of VLDB'02*, 2002, pp. 215–226.

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

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

[6] 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.

[7] 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.

[8] M. R. Huq, A. Wombacher, and P. M. G. Apers, "Facilitating fine grained data provenance using temporal data model," in *Proceedings of DMSN'10*, 2010, pp. 8–13.

[9] A. Misra, M. Blount, A. Kementsietsidis, D. Sow, and M. Wang, "Advances and challenges for scalable provenance in stream processing systems," in *Provenance and Annotation of Data and Processes*, J. Freire, D. Koop, and L. Moreau, Eds. Springer Berlin Heidelberg, 2008, pp. 253–265.

[10] 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.

[11] W. Sansrimahachai, M. Weal, and L. Moreau, "Stream Ancestor function: A mechanism for fine-grained provenance in stream processing systems," in *Proceedings of RCIS'12*, 2012, pp. 245–256.

[12] 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.

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

[14] N. Vijayakumar and B. Plale, "Towards low overhead provenance tracking in near real-time stream filtering," in *Provenance and Annotation of Data*, L. Moreau and I. Foster, Eds. Springer-Verlag, 2006, pp. 46–54.

[15] A. Woodruff and M. Stonebraker, "Supporting fine-grained data lineage in a database visualization environment," in *Proceedings of ICDE'97*, 1997, pp. 91–102.