Adaptive Inference of Fine-grained Data Provenance to Achieve High Accuracy at Lower Storage Costs

Mohammad Rezwanul Huq, Andreas Wombacher, Peter M.G. Apers
Computer Science Dept.
University of Twente
7500AE, Enschede, The Netherlands
Email: {m.r.huq, a.wombacher, p.m.g.apers}@utwente.nl

***Abstract*—In stream data processing, data arrives continuously and is processed by decision making, process control and e-science applications. To control and monitor these applications, reproducibility of result is a vital requirement. However, it requires massive amount of storage space to store fine-grained provenance data especially for those transformations with overlapping sliding windows. In this paper, we propose techniques which can significantly reduce storage costs and can achieve high accuracy. Our evaluation shows that adaptive inference technique can achieve almost 100% accurate provenance information for a given dataset at lower storage costs than the other techniques. Moreover, we present a guideline about the usage of different provenance collection techniques described in this paper based on the transformation operation and stream characteristics.**

***Keywords*-Fine-grained data provenance, Stream Data, Inference, Storage.**

I. Introduction

Stream data processing involves a large number of sensors and a massive amount of sensor data. Applications take decisions as well as control operations using these sensor readings. In case of any wrong decisions, it is important to have reproducibility to validate the previous outcome. Reproducibility refers to the ability of producing the same output after having applied the same transformation process on the same set of input dataset, irrespective of the process execution time. To be able to reproduce results, we need to store provenance data, a kind of metadata relevant to the transformation process and associated input and output dataset.

Data provenance refers to the derivation history of data from its original sources [1]. It can be defined either at the tuple-level or at the relation-level [2] also known as fine-grained and coarse-grained data provenance respectively. Fine-grained data provenance can achieve reproducibility because it documents the used set of input tuples for each output tuple and the transformation process as well. On the other hand, coarse-grained data provenance cannot achieve reproducibility because of the updates and delayed arrival of tuples. However, maintaining fine-grained data provenance in stream data processing is challenging. In stream data processing, a transformation process is continuously executed on a subset of the data stream known as a window. Executing a transformation process on a window requires to document fine-grained provenance data for this processing step to enable reproducibility. If a window is large and subsequent windows overlap significantly, then the ratio of storage space consumed by provenance data to actual sensor data becomes too high. Since provenance data is 'just' metadata and less often used by the end users, this approach seems to be infeasible and too expensive.

In [3], we report our initial idea of achieving fine-grained data provenance using a temporal data model. In that paper, we theoretically explain the application of the temporal data model to achieve the database state at a given point in time. Then, we introduce fine-grained provenance inference algorithm combining temporal data model and coarse-grained data provenance in [4]. This *inference* approach provides accurate provenance information considering both processing delay of the operation and sampling time of input tuples.

Processing delay or δ refers to the amount of time required by the system to execute the transformation process on the current window. The other parameter, sampling time or λ refers to the amount of time between the arrival of the current tuple to the subsequent one. Our proposed provenance inference algorithm reported in [4] provides 100% accurate provenance if the maximum processing delay is less than the minimum sampling time which can be expressed as $max\ (\delta) < min\ (\lambda)$, i.e. no new tuple arrives before the processing is completed.

However, due to the system workload and irregular arrival pattern of input data tuples, both δ and λ vary which may lead to infer inaccurate provenance information by our algorithm. To increase the accuracy of inferred provenance in these situations, we propose an *adaptive inference* algorithm that can achieve almost 100% accurate provenance information at a storage cost equal to the *inference* approach [4] in this paper. The *adaptive inference* algorithm adjusts the size of the window so that the exclusion of contributing tuples and the inclusion of non-contributing tuples can be avoided. Furthermore, in this paper, we also present the *dynamic mode switching* approach which decides at run-time either to store fine-grained provenance data explicitly or to infer the provenance information using *adaptive inference* based on the variation of δ and λ.

The rest of the paper is organized as follows. Section II discusses our motivating scenario. In Section III, we briefly explain our provenance inference algorithm, reported in [4], followed by the discussion of problems which result into inaccuracy in Section IV. In Section V and VI, we propose the two new, extended approaches. Section VII shows our

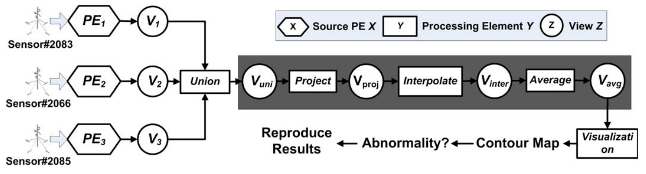

Fig. 1. Example workflow

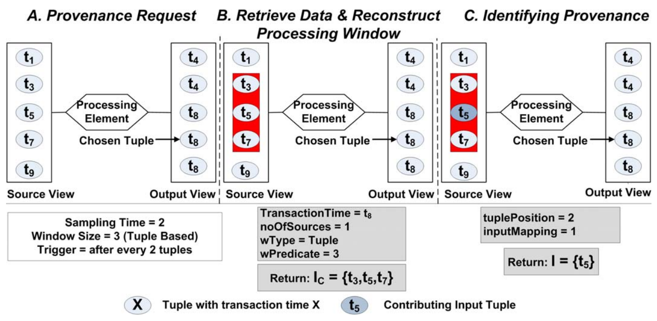

Fig. 2. Retrieval, Reconstruction & Inference of Provenance Algorithm

experimental results followed by a brief discussion on related work in Section VIII. Finally, we conclude with hints of future work.

II. Motivating Scenario

RECORD[1] is one of the projects in the context of the Swiss Experiment[2], which is a platform to enable real-time environmental experiments. One of the main objectives of the RECORD project is to study how river restoration affects the purity of groundwater and the ecosystem.

Several sensors have been deployed to monitor river restoration effects. Some of them measure electric conductivity of water which is a measure of the number of ions in the water. Increasing conductivity indicates higher level of salt in the water. We are interested to control the operation of a nearby drinking water well by facilitating the available online sensor data.

Fig. 1 shows the workflow. There are three sensors, known as: Sensor#2083, Sensor#2066 and Sensor#2085. They are deployed in different geographic locations in a known region of the river. The whole region is represented by a grid with 3×3 cells. For each sensor, there is a corresponding source processing element named PE_1, PE_2 and PE_3 which provide data tuples in a *view* V_1, V_2 and V_3 respectively. These views are the input for the *Union* processing element which produces a view V_{uni} as output. For each data tuple, *TransactionTime* is added which indicates the point in time when the tuple inserts to the database. Next, the view V_{uni} is fed to the processing element *Project* which selects only a few attributes from each tuple and generates the view V_{proj}. This view acts as an input to the processing element *Interpolate*. The task of *Interpolate* is to calculate the interpolated values for the different cells of the grid using the values sent by the three sensors and store the interpolated values in the view V_{inter}. Next, V_{inter} is used by the *Average* processing element which calculates the average electric conductivity over the region at a particular point in time. The view V_{avg} holds these data tuples and later *Visualization* processing element facilitates this view V_{avg}, to produce a contour map of electric conductivity. Later, they may produce a contour map of electric conductivity for that region. If the map shows any abnormality, researchers may want to reproduce results to validate the previous outcome. We consider the shaded processing elements in Fig. 1 to evaluate the proposed solutions later in this paper.

[1]http://www.swiss-experiment.ch/index.php/Record:Home
[2]http://www.swiss-experiment.ch/

III. Provenance Inference Technique [4]

A. The Algorithm

At first, we document coarse-grained provenance of the transformation which is a one-time action, and performed during the setup of a processing element. To explain the remaining phases, we consider a simple workflow where a *processing element* takes one *source view* as input and produces one *output view*. Moreover, we assume that, *sampling time* is 2 time units and the window holds 3 tuples. The *processing element* will be executed after arrival of every 2 tuples.

1) Document Coarse-grained Provenance: The stored provenance information is quite similar to *process provenance* reported in [5]. Inspired from this, we keep the following information of a processing element specification based on [6] as coarse-grained data provenance.

- Number of sources: indicates the total number of source views.
- Source names: a set of source view names.
- Window types: a set of window types; one element for each source. The value can be either *tuple* or *time*.
- Window predicates: a set of window predicates; one element for each source. The value actually represents the size of the window.
- Trigger type: specifies how the *processing element* will be triggered for execution (e.g. *tuple* or *time* based)
- Trigger predicate: specifies when a *processing element* will be triggered for execution.

2) Retrieve Data & Reconstruct Processing Window: This phase will be only executed if the provenance information is requested for a particular output tuple T generated by a processing element PE and it returns the set of candidate tuples which reconstruct the processing window, denoted as I_C. The tuple T is referred here as *chosen tuple* for which provenance information is requested (see Fig. 2.A).

We apply a temporal data model on streaming sensor data to retrieve appropriate data tuples based on a given timestamp. The temporal attributes are: i) **valid time** represents the point in time a tuple was created by a sensor and ii) **transaction time** is the point in time a tuple is inserted into a database. While *valid time* is anyway maintained in sensor data, *transaction time* attribute requires extra storage space.

Fig. 2.B shows this retrieval phase. The *transaction time* of the chosen tuple is t_8 which is the *reference point* to reconstruct the processing window. Since window size is 3 tuples, we retrieve the last 3 tuples having *transaction time* $< t_8$ from the source view. The retrieved tuples reconstruct the processing window which is shown by the tuples surrounded by a dark shaded rectangle in Fig. 2.B. This set of candidate tuples denoted by I_C is used in the final phase of our inference algorithm.

3) Identifying Provenance: The last phase of our proposed approach associates the chosen output tuple with the set of contributing input tuples. This mapping is done by facilitating the output and input tuples order in their respective view. At first, we determine the chosen tuple's *tuplePosition*. In our example, we have 3 tuples having *TransactionTime* equal to chosen tuple's *TransactionTime* which is t_8. Since the chosen tuple's position is 2nd among them, *tuplePosition* = 2. Now, using same approach we also determine each input tuple's *tuplePosition* which are included in the reconstructed window. Since, the value of *tuplePosition* is 2, we choose 2nd tuple in descending order of tuple appearance from the reconstructed window. The tuple with *TransactionTime* t_5 actually contributes to produce the chosen tuple from output view and is represented as a shaded tuple within the dark shaded rectangle in Fig. 2.C.

B. Requirements

The provenance inference algorithm has some requirements to be satisfied. Most of the requirements are already introduced to process streaming data in literature. In [7], authors propose to use transaction time on incoming stream data. Ensuring temporal ordering of data tuples is one of the main requirements in stream data processing. It also ensures *monotonicity in tuple ordering* property in both source and output views. This property ensures that input tuples producing output tuples in the same order of their appearance and this order is also preserved in the output view. **Classification of operations** is an additional requirement for the proposed approach.

1) Classification of Operations: In our streaming data processing platform, various types of SQL operations (e.g. *select*, *project*, *aggregate functions*, *cartesian product*, *union*) and generic functors (e.g. *interpolate*, *extrapolate*) are considered as *operations* which can be implemented inside a *processing element*. Each of these operations takes a number of input tuples and maps them to a set of output tuples. Depending on the ratio of mapping from input to output tuples we can classify these processing elements (PEs) into two major categories: Constant and Variable mapping operations.

Constant mapping operations are PEs which maintain a fixed ratio of mapping from input to output tuples. As for example: project, aggregate functions, interpolation, Cartesian product, and union. *Variable mapping* operations are PEs which don't maintain a fixed ratio of mapping from input to output tuples. As for instance: select and join. Currently, our inference algorithm is directly applicable to constant mapping operations.

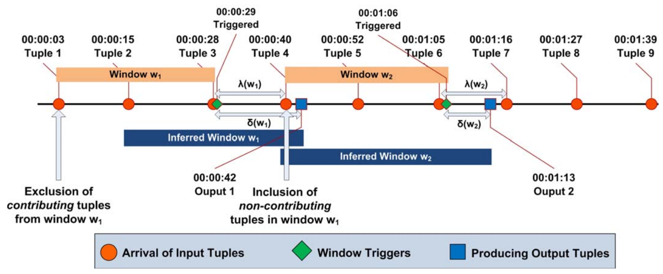

Fig. 3. Chance of inaccuracy in tuple-based windows

Each of these operations has properties like *Input tuple mapping* which specifies the number of input tuples per source contributed to produce exactly one output tuple and *Output tuple mapping* which refers to the number of output tuples produced from exactly one input tuple per source. Moreover, there are operations where all sources (e.g. join) or a specific source (e.g. union) can contribute at once. These information should be also documented in the coarse-grained data provenance of the transformation.

IV. Inaccuracy Problem

A. Tuple-based Windows

In case of tuple-based windows, our inference algorithm may infer inaccurate provenance information if a new input tuple arrives, i.e. outside the current processing window, before finishing the execution. To explain this further, at first, let us assume the following variables:

- W be the set of processing windows. $W = \{w_i \mid w_i \epsilon W\}$ where $i = 1, 2, ..., n$
- t^i_j be the j^{th} input tuple of window w_i where $j = 1, 2, ..., m$
- $\tilde{t^i_j}$ be the j^{th} input tuple outside window w_i.
- $\lambda(w_i)$ be the amount of time between first tuple outside window w_i and last tuple of w_i. $\lambda(w_i) = getTransactionTime(\tilde{t^i_1}) - getTransactionTime(t^i_m)$
- $\delta(w_i)$ be the amount of the time required to process window w_i, also known as *processing delay*.

In Fig. 3, the window size is 3 tuples and the window is triggered after the arrival of every 3^{rd} tuple. The window w_1, containing tuple 1, 2 and 3, triggers just after the arrival of tuple 3 at 00:00:29. This small time gap between the arrival of the last tuple in the window and of the window triggering is represented by ϵ which is 1 second in this case. The output of window w_1 is produced at 00:00:42 referred to by a square. Therefore, the processing delay $\delta(w_1)$ is 13 seconds. Meanwhile, a new input tuple (Tuple 4) arrives at 00:00:40 before the processing of the window w_1 is finished. Thus, the value of $\lambda(w_1)$ is 12 seconds. Since we use the output tuples' *TransactionTime* as the *reference point* in our inference algorithm, our algorithm will infer window w_1 containing the latest 3 tuples where the *TransactionTime* of input tuples is less than the *reference point* 00:00:42. Therefore, our inferred window w_1 contains tuple 2, 3 and 4 which is wrong compared

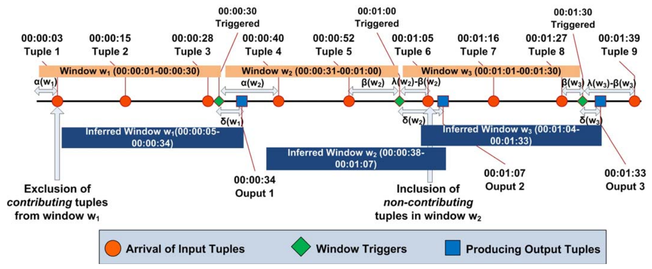

Fig. 4. Chance of inaccuracy in time-based windows

to the actual window w_1. This wrong reconstruction of the window w_1 introduces inaccuracy in our inference algorithm. On the contrary, in case of the next window w_2, no new input tuple arrives till the processing is finished. Both original and reconstructed window contain exactly the same set of tuples. From this example, we can define one of the failures of our inference algorithm that may introduce inaccuracy. It is defined as follows.

Failure 1: If a new input tuple arrives before the processing of the window w_i is finished, our inference algorithm fails to reconstruct w_i correctly due to the inclusion of the newly arrived non-contributing tuple. If the following condition holds for tuple-based windows, we have a failure.

$$\lambda(w_i) < \delta(w_i) + \epsilon$$
$$\lambda(w_i) < \delta(w_i) \qquad [\ \epsilon << \delta\ ] \tag{4.1.1}$$

B. Time-based Windows

Inaccuracy can also occur in case of time-based windows:

- Exclusion of a contributing tuple from the lower end of the window.
- Inclusion of a non-contributing tuple at the upper end of the window.

To explain these two cases, let us assume some variables in addition to the variables described in Section IV-A.

- $\alpha(w_i)$ be the amount of the time between the window w_i starts and arrival of first tuple in w_i.
- $\beta(w_i)$ be the amount of the time between the arrival of last tuple in w_i and the window w_i triggers.

Fig. 4 shows the situation in a particular time-based window of 30 seconds and triggering after every 30 seconds. In Fig. 4, the window w_1 contains tuples 1, 2 and 3. The first tuple, tuple 1, arrives at 00:00:03 and thus, $\alpha(w_1) = 2$ seconds. This window is triggered at 00:00:30 and the output tuples are produced at 00:00:34. Therefore, $\delta(w_1) = 4$ seconds. Since we use the output tuples' *TransactionTime* as the *reference point* in our inference algorithm, our inferred window for w_1 contains the input tuples having *TransactionTime* within 00:00:05 to 00:00:34. Therefore, it contains tuples 2 and 3 only which is not the same as the actual window w_1 and thus our inference algorithm provides inaccurate provenance information. This failure can be defined as follows.

Failure 2: Exclusion of a contributing tuple from the lower end of the window w_i may occur if the processing delay $\delta(w_i)$ is longer than the difference between the first input tuple in w_i and the time at which w_i starts. If the following condition holds, we have a failure.

$$\alpha(w_i) < \delta(w_i) \tag{4.2.1}$$

The next window w_2 contains the tuples 4 and 5. Here, $\alpha(w_2) = 10$ and $\beta(w_2) = 8$. Meanwhile, a new input tuple (Tuple 6) arrives at 00:01:05 before the processing is finished which makes the value of $\lambda(w_2) = 13$ seconds and $\delta(w_2) = 7$ seconds. The inferred window for w_2 holds tuples 4, 5 and 6 which is different to the actual window of w_2. This failure can be defined as follows.

Failure 3: Inclusion of a newly arrived non-contributing input tuple may occur due to arrival of the new input tuple before the processing of the window w_i is finished. If the following holds, our inference algorithm provides inaccurate provenance information.

$$\lambda(w_i) - \beta(w_i) < \delta(w_i) \tag{4.2.2}$$

The last window, w_3, shown in Fig. 4 starts at 00:01:01 and ends at 00:01:30. In this case: $\alpha(w_i) \nleq \delta(w_i)$ and $\lambda(w_i) - \beta(w_i) \nleq \delta(w_i)$ holds. So, none of the failures occur in window w_3.

V. Proposed Dynamic Mode Switching

To increase the accuracy of the provenance inference, we propose a *dynamic mode switching* approach. This technique switches the provenance collection mode from *inferred* to *explicit* at run-time, if it detects any possibilities of reconstructing the window incorrectly using our inference algorithm. After switching from *inferred* to *explicit* mode, it collects provenance information in a traditional way only for that particular processing window w_i and then it switches back again to the *inferred* mode before processing the next window w_{i+1}.

To detect any anomalies at the time of window reconstruction, *dynamic mode switching* uses the various failure conditions given in Section IV. However, the value of $\lambda(w_i)$ cannot be computed unless a new input tuple arrives before this switching takes place. Therefore, we use the minimum value computed over $\lambda(w_1)$ to $\lambda(w_{i-1})$ instead of $\lambda(w_i)$. Since the other parameters $\alpha(w_i)$, $\beta(w_i)$ and $\delta(w_i)$ associated with the failure conditions can be computed for that particular window w_i, we need not to replace them. Therefore, *dynamic mode switching* method decides to switch from *inferred* to *explicit* mode if any of the given conditions hold:

- For tuple-based windows: $min\ [\lambda(w_1)..\lambda(w_{i-1})] < \delta(w_i)$
- For time-based windows: $\alpha(w_i) < \delta(w_i)$ or $min\ [\lambda(w_1)..\lambda(w_{i-1})] - \beta(w_i) < \delta(w_i)$

However, using the minimum value computed over previous $\lambda(w_j)$ (where $j < i$) will switch the mode from *inferred* to *explicit* many times. It will incur more storage overhead and may exceed the storage cost of *explicit* approach. To reduce the storage overhead, it is possible to compute the minimum

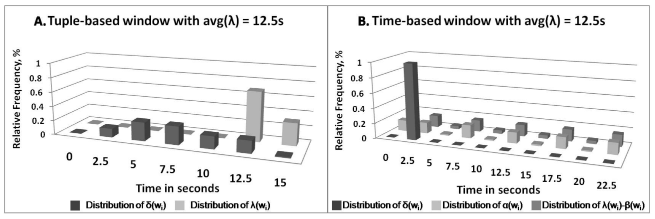

Fig. 5. Distribution of different variables in Tuple and Time-based windows

value over the previous $\lambda(w_j)$ confined within a window. The larger the size of the window defined over $\lambda(w_j)$, the more precise the provenance information is.

The main advantage of this method is to achieve 100% accurate provenance. However, this method incurs extra storage costs because of maintaining explicit provenance data for some cases where our inference algorithm cannot reconstruct the window correctly. Furthermore, this method requires extra processing costs since the mode switching decision is made for each processing window w_i. If a particular processing element has a very high trigger rate, we would advise to avoid using this method.

VI. Proposed Adaptive Inference

A. Adaptivity in Tuple-based Windows

We can estimate the accuracy of our *adaptive inference* algorithm based on the failure conditions given in Section IV. Suppose, W^E be the set of processing windows where Failure 1 occurs. Thus, $W^E := \{\ w_i \ \epsilon\ W |\ \lambda(w_i) < \delta(w_i)\ \}$. Using probability theory, we can compute the estimated accuracy.

$$\begin{aligned} \textit{Estimated Accuracy} &= 1 - P(w_i \ \epsilon\ W^E) \\ &= 1 - \sum_{\delta(w_i)} \ [P(\lambda(w_i) < \delta(w_i) \mid \delta(w_i)) \\ &\quad \times\ P(\delta(w_i))] \qquad (6.1.1) \end{aligned}$$

In a tuple-based window w_i, inaccuracy can occur if a new input tuple arrives before the processing on the current window is finished (see Eq. 4.1.1). In this case, our reconstructed window may include one non-contributing tuple at the upper end and exclude one contributing tuple from the lower end of the window since the window size is based on the number of tuples. To overcome this problem, we have to adjust the point in time known as *reference point* beyond which we will not consider to include any more tuples in the window w_i. Currently, this *reference point* is the *transaction time* of the chosen tuple. The time gap between the current and new *reference point* is known as *offset* and it depends on $\delta(w_i)$. Since $\forall_i : \delta(w_i) \geq 0$, *offset* $= \min\ \{\delta(w_i)_{i=1..n}\}$. Therefore,

$$\begin{aligned} \textit{New Reference Point} &= \textit{Current Reference Point} - \textit{offset} \\ &= \textit{TransactionTime of the chosen tuple} \\ &\quad - \min\ \{\delta(w_i)_{i=1..n}\} \qquad (6.1.2) \end{aligned}$$

Fig. 5.A shows the histogram of the distribution of $\delta(w_i)$ and $\lambda(w_i)$ for the experiment explained in Section IV-A with $avg\{\lambda(w_i)_{i=1..n}\} = 12.5$ seconds (see Fig. 3) . The black and light-grey shaded bar shows the relative frequency for $\delta(w_i)$ and $\lambda(w_i)$ respectively based on the range defined in x-axis in seconds. We can see little overlapping between $\delta(w_i)$ and $\lambda(w_i)$ within 10-12.5 seconds range. Intuitively, it suggests that there are only a few windows where Failure 1 occurs.

Our proposed *adaptive inference* algorithm adjusts the *reference point* to infer accurate provenance in presence of this failure. From Fig. 5.A, we see that $\delta(w_i) = 2.5$ seconds. Considering the window w_1 in Fig. 3, the *TransactionTime of the chosen tuple* is $00:00:42$. Therefore, according to Eq. 6.1.2,

$$\begin{aligned} \textit{New Reference Point} &= 00:00:42 - 00:00:02.5 \\ &= 00:00:39.5 \end{aligned}$$

So, our *adaptive inference* algorithm considers only those tuples having *TransactionTime* less than $00:00:39.5$ and retrieves the last 3 tuples based on their *TransactionTime*. Now, the adapted inferred window w_1 contains tuples 1, 2 and 3 which is exactly same as original window w_1. Therefore, *adaptive inference* algorithm can significantly resolve the failures.

B. Adaptivity in Time-based Windows

Estimating the accuracy of our inference algorithm for time-based windows is possible in the following way. If W^E be the set of processing windows where Failure 2 and Failure 3 occurs then, $W^E := \{\ w_i \ \epsilon\ W |\ \lambda(w_i) < \delta(w_i)$ or $\lambda(w_i) - \beta(w_i) < \delta(w_i)\}$. Therefore, using probability theory,

$$\begin{aligned} \textit{Estimated Accuracy} &= 1 - P(w_i \ \epsilon\ W^E) \\ &= 1 - \sum_{\delta(w_i)} [P(\alpha(w_i) < \delta(w_i) | \delta(w_i)) \\ &\quad \times P(\delta(w_i))] \\ &\quad - \sum_{\delta(w_i)} [P(\lambda(w_i) - \beta(w_i) < \delta(w_i) | \delta(w_i)) \\ &\quad \times P(\delta(w_i))] \qquad (6.2.1) \end{aligned}$$

In time-based windows, inaccuracy can occur by either including a non contributing tuple at the upper end of the window or excluding a contributing tuple from the lower end of the window. Our proposed adaptivity technique adapts the window size so that we can avoid these situations. Both lower and upper ends of the window based on Eq. 4.2.1 and 4.2.2 respectively will be adjusted. The lower end of the window is calculated by this formula: *LowerEnd = TransactionTime - windowSize - offset* $_{lower}$ where *offset* $_{lower}$ must satisfy the following:

- *offset* $_{lower} \geq a$ and *offset* $_{lower} \leq avg\{\lambda(w_i)_{i=1..n}\} + b$ where $a = \max\{\delta(w_i) - \alpha(w_i)_{i=1..n}\}$ and $b = \min\{\delta(w_i) - \alpha(w_i)_{i=1..n}\}$. The value of *offset* $_{lower}$ should lie within this range and the value should maximize the accuracy. To determine the actual value, we should iterate over this given range and choose the value which gives the maximum accuracy.

On the other hand, *UpperEnd = TransactionTime - offset* $_{upper}$ where *offset*$_{upper}$ must satisfy the following conditions.

- *offset* $_{upper} \geq c$ and *offset* $_{upper} \leq avg\{\lambda(w_i)_{i=1..n}\} + d$ where $c = \max\{(\delta(w_i) - (\lambda(w_i) - \beta(w_i)))_{i=1..n}\}$ and $d = \min\{(\delta(w_i) - (\lambda(w_i) - \beta(w_i)))_{i=1..n}\}$. Similar to the *offset* $_{lower}$, the value of *offset* $_{upper}$ should lie within the aforesaid range and we should choose the value which gives the maximum accuracy.

Fig. 5.B shows the relative frequency distribution of $\delta(w_i)$, $\alpha(w_i)$ and $\lambda(w_i) - \beta(w_i)$ in black, light-grey and dark-grey shaded bars respectively for the experiment described in Section IV-B with $avg\{\lambda(w_i)_{i=1..n}\} = 12.5$ seconds (see Fig. 4). In Fig. 5.B, we see that there are some cases where $\alpha(w_i) < \delta(w_i)$, i.e. failure 2 occurs. There is also little overlapping between $\delta(w_i)$ and $\lambda(w_i) - \beta(w_i)$ within 0-2.5 seconds range which suggests that failure 3 can also occur a few times.

Adaptive inference algorithm adjusts the lower and upper end of the window to provide accurate provenance in case of these failures. From Fig. 5.B, $a = 2.5$ and $b = 0$ seconds. Therefore, *offset* $_{lower} \geq 2.5$ and ≤ 12.5 seconds. Similarly, $c = 2.5$ and $d = 0$ seconds. Thus, *offset* $_{upper} \geq 2.5$ and ≤ 12.5 seconds. Considering the window w_2 in Fig. 4, $00:00:24.5 \leq LowerEnd \leq 00:00:34.5$ and $00:00:54.5 \leq UpperEnd \leq 00:01:04.5$. Now, *adaptive inference* technique adjusts the window w_2 that starts at $00:00:34.5$ and ends at $00:00:54.5$ and contains tuples 4 and 5 which is exactly same as the original window w_2 (see Fig. 4).

VII. Evaluation

A. Evaluating Criteria & Datasets

The consumption of storage space for fine-grained data provenance is our main evaluation criteria. Existing approaches [8], [7], [9] record fine-grained data provenance explicitly in varying manners. Since these implementations are not available, our proposed approach is compared with an implementation of a fine-grained data provenance documentation running in parallel with the proposed approach on the Sensor Data Web[3] platform. Moreover, we improve this explicit fine-grained provenance collection system using the concept of *basic factorization* [9]. *Basic factorization* technique minimizes the provenance storage requirement by removing the common provenance records and storing only one copy of the provenance record. We also compare our approach with this *improved explicit* method.

Finally, we need to check whether all approaches produce the same provenance information. In this case the traditional fine-grained provenance information is used as a ground truth and it is compared with the fine-grained provenance information inferred by *dynamic mode switching* and *adaptive inference* techniques.

For evaluation, a real dataset[4] measuring electric conductivity of the water, collected by the RECORD project is used.

[3]http://sourceforge.net/projects/sensordataweb/

[4]http://data.permasense.ch/topology.html\#topology

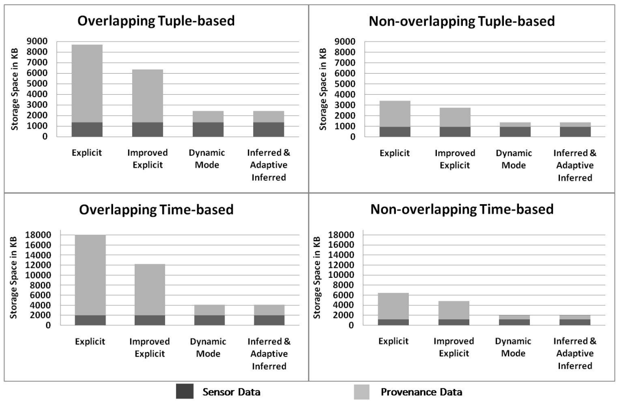

Fig. 6. Storage space requirements in different cases

The workflow operating on this dataset has been discussed in Section II. The experiments are performed on a PostgreSQL 8.4 database and the Sensor Data Web platform. The input dataset contains 3000 tuples requiring 720kB storage space.

B. Storage Requirement

We measure the storage overhead to maintain fine-grained data provenance for the *Interpolation* processing element based on our motivating scenario (see Section II) with various window specifications. In the tuple-based non-overlapping case, each window contains three tuples and the interpolation is executed for every 3rd tuple. This results in about $3000 \div 3 \times 9 = 9000$ output tuples which require about 220kB. In the tuple-based overlapping case, the window contains 3 elements and the operation is executed for every tuple. This results in about $3000 \times 9 = 27000$ output tuples which require about 650kB. For time-based windows, we consider windows having size 15 seconds and trigger rate is once in every 5 and 15 seconds for overlapping and non-overlapping case respectively.

Fig. 6 shows the comparison among *explicit*, *improved explicit*, *dynamic mode switching*, *inference* and *adaptive inference* techniques. The sum of the storage costs for input and output tuples is depicted as dark gray boxes named as sensor data, while the provenance data storage costs is depicted as light gray boxes. In all cases, the *explicit* method takes maximum storage to maintain provenance data. The *improved explicit* technique takes less storage to store provenance data than *explicit* technique especially in case of overlapping windows. However, maintaining provenance data requires 2 times more space than actual sensor data in this method. The *inference* technique [4] and our proposed *adaptive inference* method always have same storage cost for any window specification. Moreover, in this experiment, the *dynamic mode switching* approach also requires storage space equal to *adaptive inference* because it never switched the mode from *inferred* to *explicit*. For these techniques, the ratio of provenance data to sensor data is $1:1$ in the worst case which is the lowest among all the other techniques.

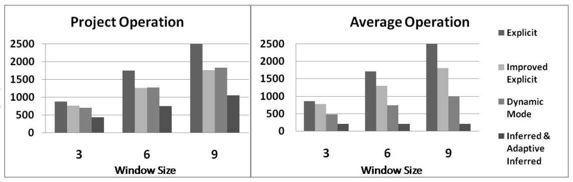

Fig. 7. Comparison of Provenance Data storage requirement

However, *dynamic mode switching* has processing overhead due to the computation of switching decision and it may require extra storage when any of the failure conditions has occurred. On the other hand, *adaptive inference* approach has no extra processing cost and is only dependent on the volume of sensor data which makes the *adaptive inference* approach more storage-friendly than the others.

Additional tests confirm the results. Fig. 7 depicts the comparison among different methods for project and average operations with varying window sizes. In all these cases, the window triggers after arrival of every new tuple. In case of project operation, the *dynamic mode switching* requires more storage than the *inference* and *adaptive inference* technique due to the mode switching in almost 30% window executions. Since the *adaptive inference* technique only depends on the size of the sensor data, it takes less than half the storage compared to the explicit method.

The average operation produces the same number of output tuples in all cases which is 3000. Therefore, our *adaptive inference* method always takes the same amount of storage space because of the equal size of sensor data. However, the explicit method depends on the window size and the overlapping between windows. For the average operation, our proposed *adaptive inference* method outperforms all the other methods and takes at least 4 times less space than the *explicit* method. Please be noted that this ratio depends on the chosen window size and trigger specification and if the window size is larger and there is a big overlap between windows, the *adaptive inference* approach performs even better.

C. Accuracy

Lastly, we discuss about the accuracy of our proposed techniques. For the experiments described in Section VII-B, we get 100% accurate inferred provenance for all different methods. Therefore, we carry out another experiment with *project* processing element having different parameters. Fig. 8 shows the comparison of accuracy achieved among different methods proposed in this paper. The *estimated* method computes the estimated accuracy using equations 6.1.1 and 6.2.1 in Section VI. This method gives the approximate lower bound of the accuracy that can be achieved using *inference* and *adaptive inference* algorithm. The *inference* method provides the result after using inference algorithm reported in [4]. It achieves more than 90% accuracy in all experiments. The small inaccuracy is introduced due to the longer and variable delays. To increase accuracy, we propose two methods. Firstly, the *dynamic mode switching* achieves 100% accuracy since it stores provenance information explicitly if any of the failure conditions occurs. Secondly, the *adaptive inference* method also increases the accuracy explained in Section VI-A and VI-B. Our proposed *adaptive inference* approach achieves almost 100% accurate provenance information even in the presence of longer and variable delays.

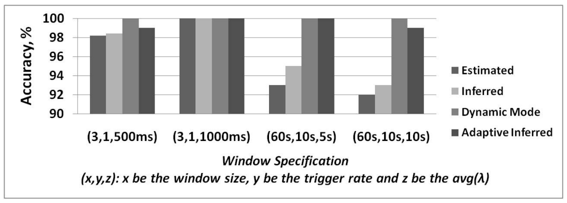

Fig. 8. Comparison of Accuracy among different techniques in various cases

TABLE I
VARIABLES USED FOR ANALYSIS

Number of input tuples: n_i	Size of a provenance tuple: S_p
Window size in tuples: w	Size of a pointer tuple to refer actual provenance: p
Trigger rate in tuples: t	Number of distinct provenance tuples: n_1 ($n_1 \leq n_p$)
Input tuple mapping: x	Overhead to add TransactionTime: q
Output tuple mapping: y	Prob. of wrong window formulation: $P(w_i \epsilon W^E)$
Number of output tuples: n_o	
$n_o = (y \times n_i \times w) \div (t \times x)$	
Number of provenance tuples: $n_p = x \times n_o$	

D. Discussion

Table I summarizes different variables we use in Table II for quantitative analysis. The first two methods: *explicit* and *improved explicit* provides 100% accurate provenance information. However, these techniques are not well-suited in case of sliding windows with large overlapping between them and processes with high trigger rate due to their excessive storage space requirement. The *inference* algorithm [4] achieves around 90% accuracy at less than half the storage space compared to the *explicit* method. It suits better for the applications where approximate results are accepted. Our proposed *adaptive inference* technique provides almost 100% accuracy with the consumption of storage space equal to the *inference* method. This approach performs better than the *inference* method where 'exact result' is a requirement and also transformations are taking place very frequently. The other proposed method, *dynamic mode switching* also provides 100% accurate provenance but at a higher storage costs than *adaptive inference*. Moreover, it has a processing overhead to switch between modes. That is why, this approach is well-suited for the applications with less trigger rate. Table II can be used as a guideline so that a user can choose a specific technique based on the transformation processes, distribution of δ, λ and other parameters. For transformations with high trigger rate, large overlapping between windows and to achieve exact reproducibility, *adaptive inference* approach outperforms all the other methods.

TABLE II
SUMMARY OF STORAGE CONSUMPTION AND ACCURACY

Technique	Storage		Accuracy	
	Quantitative	Qualitative	Quantitative	Qualitative
Explicit Provenance	$n_p \times S_p$	most costly	100%	100% accurate
Improved Explicit Provenance	$n_1 \times S_p + n_p \times p$	less costly than explicit if window overlaps	100%	100% accurate
Inference Provenance	$n_i \times q + n_o \times q$	significant reduction	can be estimated by Eq. 6.1.1 & 6.2.1	less accurate around 90%
Adaptive Inference Provenance	$n_i \times q + n_o \times q$	same as inference	can be estimated by Eq. 6.1.1 & 6.2.1 with changed window bounds	more accurate than inference
Dynamic Mode Switching	$n_i \times q + n_o \times q + P \times n_p \times S_p$	more costly than inference if $P > 0$	100%	100% accurate

VIII. Related work

Several academic and scientific projects facilitate the execution of continuous queries and stream data processing, reported in [10], [11], [12]. All these techniques proposed optimization for storage space consumed by sensor data. However, none of these systems can achieve reproducible results in stream data processing.

In [13], authors have described a data model to compute provenance on both relation and tuple level. This data model follows a graph pattern and shows case studies for traditional data but it does not address how to handle streaming data and associated overlapping windows.

In [14], authors have presented an algorithm for lineage tracing in a data warehouse environment. They have provided data provenance on tuple level. LIVE [8] is an offshoot of this approach which supports streaming data. It is a complete DBMS which preserves explicitly the lineage of derived data items in form of boolean algebra. However, these techniques incur extra storage overhead to maintain fine-grained data provenance.

In sensornet republishing [7], the system documents the transformation of online sensor data to allow users to understand how processed results are derived and support to detect and correct anomalies. They used an annotation-based approach to represent data provenance explicitly. However, our proposed method does not store fine-grained provenance data explicitly.

In [9], authors proposed approaches to reduce the amount of storage required for provenance data. To minimize provenance storage, they remove common provenance records; only one copy is stored. Then, using an extra provenance pointer, data tuples can be associated with their appropriate provenance records. Their approach seems to have less storage consumption than traditional fine-grained provenance in case of sliding overlapping windows. However, their method still maintain fine-grained data provenance explicitly.

IX. Conclusion and Future Work

In this paper, we propose several methods to infer fine-grained data provenance with high accuracy at lower storage costs. Our proposed *adaptive inference* approach reduces storage cost significantly than the other approaches and also achieves almost 100% accuracy even if the processing takes longer and has high trigger rate. *Dynamic mode switching* is suitable for transformations that occur less frequently. Our proposed approaches are evaluated in different types of settings with multiple sources and variable delays. In future, we will try to infer provenance for multiple processing steps with high accuracy and reduced storage costs.

References

[1] Y. L. Simmhan, B. Plale, and D. Gannon, “A survey of data provenance in e-science,” *SIGMOD Rec.*, vol. 34, no. 3, pp. 31–36, 2005.

[2] P. Buneman and W. C. Tan, “Provenance in databases,” in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007, pp. 1171–1173.

[3] M. R. Huq, A. Wombacher, and P. M. G. Apers, “Facilitating fine grained data provenance using temporal data model,” in *Proceedings of the 7th Workshop on Data Management for Sensor Networks (DMSN)*, September 2010, pp. 8–13.

[4] M. R. Huq, A. Wombacher, and P. Apers, “Inferring fine-grained data provenance in stream data processing: Reduced storage cost, high accuracy,” in *International Conference on Database and Expert Systems Applications (DEXA 2011)*, Lecture Notes in Computer Science, August 2011, vol. 6861, pp. 118–127.

[5] Y. L. Simmhan, B. Plale, and D. Gannon, “Karma2: Provenance management for data driven workflows,” *International Journal of Web Services Research, Idea Group Publishing*, vol. 5, pp. 1–23, 2008.

[6] A. Wombacher, “Data workflow - a workflow model for continuous data processing,” Centre for Telematics and Information Technology, University of Twente, Technical Report TR-CTIT-10-12, 2010.

[7] U. Park and J. Heidemann, “Provenance in sensornet republishing,” *Provenance and Annotation of Data and Processes*, pp. 280–292, 2008.

[8] A. Sarma, M. Theobald, and J. Widom, “LIVE: A Lineage-Supported Versioned DBMS,” in *Proc. of International Conference on Scientific and Statistical Database Management (SSDBM)*, pp. 416–433, 2010.

[9] A. Chapman, H. Jagadish, and P. Ramanan, “Efficient provenance storage,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 993–1006.

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

[11] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, “Aurora: a new model and architecture for data stream management,” *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[12] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang *et al.*, “The design of the borealis stream processing engine,” in *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA*, 2005, pp. 277–289.

[13] P. Buneman, S. Khanna, and T. Wang-Chiew, “Why and where: A characterization of data provenance,” in *ICDT 2001*, pp. 316–330.

[14] Y. Cui and J. Widom, “Lineage tracing for general data warehouse transformations,” *VLDB Journal*, vol. 12, no. 1, pp. 41–58, May 2003.