# Inferring Fine-Grained Data Provenance in Stream Data Processing: Reduced Storage Cost, High Accuracy

Mohammad Rezwanul Huq, Andreas Wombacher, and Peter M.G. Apers

University of Twente, 7500 AE Enschede, The Netherlands
{m.r.huq,a.wombacher,p.m.g.apers}@utwente.nl

**Abstract.** Fine-grained data provenance ensures reproducibility of results in decision making, process control and e-science applications. However, maintaining this provenance is challenging in stream data processing because of its massive storage consumption, especially with large overlapping sliding windows. In this paper, we propose an approach to infer fine-grained data provenance by using a temporal data model and coarse-grained data provenance of the processing. The approach has been evaluated on a real dataset and the result shows that our proposed inferring method provides provenance information as accurate as explicit fine-grained provenance at reduced storage consumption.

## 1 Introduction

Stream data processing often deals with massive amount of sensor data in e-science, decision making and process control applications. In these kind of applications, it is important to identify the origin of processed data. This enables a user in case of a wrong prediction or a wrong decision to understand the reason of the misbehavior through investigating the transformation process which produces the unintended result.

Reproducibility as discussed in this paper means the ability to regenerate data items, i.e. for every process $P$ executed on an input dataset $I$ at time $t$ resulting in output dataset $O$, the re-execution of process $P$ at any later point in time $t'$ (with $t' > t$) on the same input dataset $I$ will generate exactly the same output dataset $O$. Generally, reproducibility requires metadata describing the transformation process, usually known as provenance data.

In [1], data provenance is defined as derivation history of data starting from its original sources. Data provenance can be defined either at tuple-level or at relation-level known as fine-grained and coarse-grained data provenance respectively [2]. Fine-grained data provenance can achieve reproducible results because for every output data tuple, it documents the used set of input data tuples and the transformation process itself. Coarse-grained data provenance provides similar information on process or view level. In case of updates and delayed arrival of tuples, coarse-grained data provenance cannot guarantee reproducibility.

Applying the concept of fine-grained data provenance to stream data processing introduces new challenges. In stream data processing, a transformation

A. Hameurlain et al. (Eds.): DEXA 2011, Part II, LNCS 6861, pp. 118–127, 2011.

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 provenance data size might be bigger than the actual sensor data size. Since provenance data is 'just' metadata, this approach seems to be too expensive.

In [3], we initially propose the basic idea of achieving fine-grained data provenance using a temporal data model. In this paper, we extend and complete our work to infer fine-grained data provenance using a temporal data model and coarse-grained data provenance. Adding a temporal attribute (e.g. timestamp) to each data item allows us to retrieve the overall database state at any point in time. Then, using coarse-grained provenance of the transformation, we can reconstruct the window which was used for the original processing and thus ensuring reproducibility. Due to the plethora of possible processing operations, a classification of operations is provided indicating the classes applicable to the proposed approach. In general, the approach is directly applicable if the processing of any window produces always the same number of output tuples. Eventually, we evaluate our proposed technique based on storage and accuracy using a real dataset.

This paper is structured as follows. In Section 2, we provide a detailed description of our motivating application with an example workflow. In Section 3, we discuss existing work on both stream processing and data provenance briefly. In Section 4, we explain our approach and associated requirements followed by the discussion on few issues in Section 5. Next, we present the evaluation of our approach in Section 6. Finally, we conclude with hints of future research.

## 2 Motivating Scenario

RECORD[1] is one of the projects in the context of the Swiss Experiment, which is a platform to enable real-time environmental experiments. One objective of the RECORD project is to study how river restoration affects water quality, both in the river itself and in groundwater. Several sensors have been deployed to monitor river restoration effects. Some of them measure electric conductivity of water. Increasing conductivity indicates the higher level of salt in water. We are interested to control the operation of the drinking water well by facilitating the available online sensor data.

Based on this motivating scenario, we present a simplified workflow, that will also be used for evaluation. Fig. 1 shows the workflow based on the RECORD project. There are three sensors, known as: Sensor#1, Sensor#2 and Sensor#3. They are deployed in different locations in a known region of the river which is divided into a grid with $3 \times 3$ cells. These sensors send data tuples, containing *sensor id*, *(x,y) coordinates*, *timestamp* and *electric conductivity*, to source processing element named $PE_1$, $PE_2$ and $PE_3$ which outputs data tuples in a *view* $V_1$, $V_2$ and $V_3$ respectively. These views are the input for a *Union* processing

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

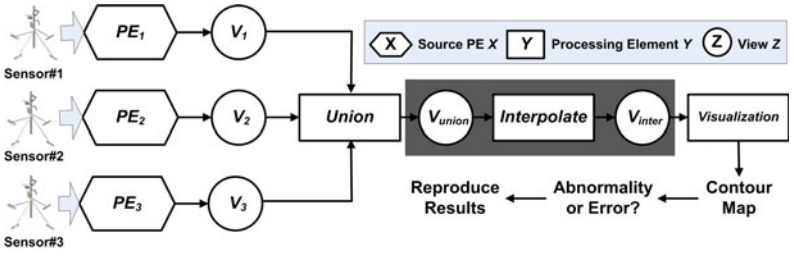


**Fig. 1.** Workflow based on RECORD scenario

element which produces a view $V_{union}$ as output. This view acts as an input to the processing element *Interpolate*. The task of *Interpolate* is to calculate the interpolated values for all the 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 *Visualization* processing element to produce a contour map of electric conductivity. If the map shows any abnormality, researchers may want to reproduce results to validate the previous outcome. The dark-shaded part of the workflow in Fig. 1 is considered to evaluate our proposed approach.

## 3 Related Work

Stream data processing engines reported in [4], [5], [6]. These techniques proposed optimization for storage space consumed by sensor data. However, neither of these systems maintain provenance data and cannot achieve reproducible results.

Existing work in data provenance addresses both fine and coarse-grained data provenance. In [7], 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, both of these techniques incur extra storage overhead to maintain fine-grained data provenance.

In sensornet republishing [9], 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. In [10], 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. Their approach has less storage consumption than explicit fine-grained provenance in case of sliding overlapping windows. However, these methods still maintain fine-grained data provenance explicitly.

# 4 Proposed Solution

## 4.1 Provenance Inference Algorithm

The first phase of our provenance inference algorithm is to document coarse-grained provenance of the transformation which is an one-time action, performed during the setup of a processing element. The algorithms for the next two phases are given here along with an example. To explain these algorithms, 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* of *source view* is 2 time units and the window holds 3 tuples. The *processing element* will be executed after arrival of every 2 tuples. $t_1$, $t_2$ and so on are different points in time and $t_1$ is the starting time.

**Document Coarse-grained Provenance:** The stored provenance information is quite similar to *process provenance* reported in [11]. Inspired from this, we keep the following information of a processing element specification based on [12] and the classification introduced in Section 4.2 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; 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. The value can be either *tuple* or *time*.
- Trigger predicate: specifies when a *processing element* will be triggered for execution. If *trigger type* is *tuple* and the value of *trigger predicate* is 10, it means that the processing element will be executed after the arrival of every 10th tuple.

**Algorithm 1:** Retrieve Data & Reconstruct Processing Window Algorithm

**Input**: A tuple $T$ produced by processing element $PE$, for which fine-grained provenance needs to be found

**Output**: Set of input tuples $I_j^{P_w}$ for each source $j$ which form processing window $P_w$ to produce $T$

```
1 TransactionTime ← getTransactionTime(PE, T);
2 noOfSources ← getNoOfSources(PE);
3 for j ← 1 to noOfSources do
4 |   sourceView ← getSourceName(PE, j);
5 |   wType ← getWindowType(sourceView);
6 |   wPredicate ← getWindowPredicate(sourceView);
7 |   I_j^{P_w} ← getLastNTuples(sourceView, TransactionTime, wType, wPredicate);
8 end
```

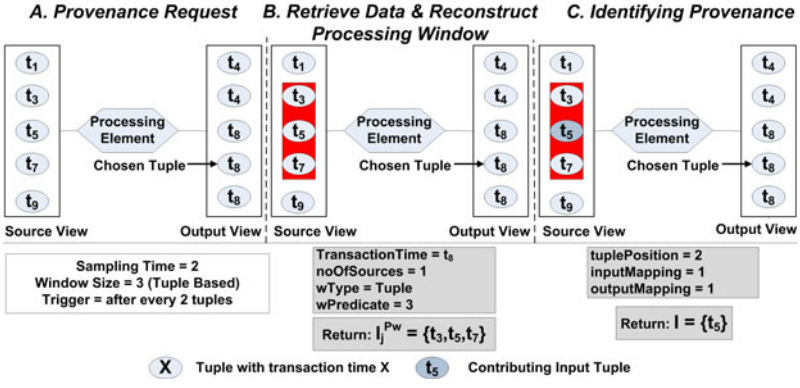


**Fig. 2.** Retrieval, Reconstruction and Inference phases of Provenance Algorithm

**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 $PE$. 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.

The method of retrieving data and reconstructing processing window is given in Algorithm 1. The *transaction time* of the chosen tuple and number of participating sources are retrieved in line 1 and 2. Then, for each participating *source* view, we retrieve it's *name*, *window type* and *window predicate* in line 4-6. Then, we retrieve the set of the input tuples which form the processing window based on the chosen tuple's *transaction time* in line 7. If window type is *tuple*, we retrieve last $n$ tuples added to the source view before the *TransactionTime* where $n$ is the window predicate or window size. On the contrary, if window type is *time*, we retrieve tuples having *transaction time* ranging within $[TransactionTime - wPredicate, TransactionTime)$. The retrieved tuples reconstruct the processing window which is shown by the tuples surrounded by a dark shaded rectangle in Fig. 2.B.

**Identifying Provenance:** The last phase associates the chosen output tuple with the set of contributing input tuples based on the reconstructed window in the previous phase. This mapping is done by facilitating the output and input tuples order in their respective view. Fig. 2.C shows that the chosen tuple in the output view maps to the 2nd tuple in the reconstructed window (shaded rectangle in source view). To compute the tuple position and infer provenance, some requirements must be satisfied which are discussed next.

### 4.2 Requirements

Our provenance inference algorithm has some requirements to be satisfied. Most of the requirements are already introduced to process streaming data in existing literature. In [9], authors propose to use transaction time on incoming stream data as **explicit timestamps**. Ensuring **temporal ordering** of data tuples is one of the main requirements in stream data processing. 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.

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.

*Constant Mapping Operations* are PEs which have a fixed ratio of mapping from input to output tuples per window, i.e. $1:1$, $n:1$, $n:m$. As for example: project, aggregates, interpolation, cartesian product, and union. *Variable Mapping Operations* are PEs which have not a fixed ratio of mapping from input to output tuples per window, e.g. select and join. Currently, our inference algorithm can be applied directly to constant mapping operations. Each of these operations has property 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 coarse-grained data provenance.

### 4.3 Details on Identifying Provenance Phase

Algorithm 2 describes the approach we take to identify the correct provenance. First, we retrieve our stored coarse-grained provenance data in line 2-5. For operations where only one input tuple contributes to the output tuple (line 6), we have to identify the relevant contributing tuple. In case there are multiple sources used but only one source is contributing (line 7), a single tuple is contributing. Based on the temporal ordering, the knowledge of the nested processing of multiple sources, the contributing source and the output tuple mapping, the position of the tuple in the input view which contributed to the output tuple can be calculated (line 9). The tuple is then selected from the contributing input source in line 10.

If there is one source or there are multiple sources equally contributing to the output tuple, the position of the contributing tuple per source has to be determined (line 13). The underlying calculation is again based on the knowledge of the nested processing of multiple sources, the contributing source and the output tuple mapping, the position of the tuple in the input view $j$. In line 14 the tuple is selected based on the derived position from the set of input tuples.

**Algorithm 2:** Identifying Provenance Algorithm

**Input**: Set of input tuples $I_j^{P_w}$ for each source $j$ which form processing window $P_w$ to produce $T$
**Output**: Set of input tuples $I$ which contribute to produce $T$

```
1  I = ∅;
2  inputMapping ← getInputMapping(PE);
3  outputMapping ← getOutputMapping(PE);
4  contributingSource ← getContributingSource(PE, T);
5  noOfSources ← getNoOfSources(PE);
6  if inputMapping = 1 then          /* only one input tuple contributes */
7      if noOfSources > 1 ∧ contributingSource = Specific then
8          parent ← getParent(PE, T);
9          tuplePosition ← getPosition(PE, T, parent, outputMapping);
10         I ← selectTuple(I^{Pw}_{parent}, tuplePosition);
11     else
12         for j ← 1 to noOfSources do
13             tuplePosition ← getPosition(PE, T, j, outputMapping);
14             I ← selectTuple(I^{Pw}_j, tuplePosition) ∪ I;
15         end
16     end
17 else                                /* all input tuples contribute */
18     for j ← 1 to noOfSources do
19         I ← I^{Pw}_j ∪ I;
20     end
21
```

In cases where all input tuples contribute to the output tuple independent of the number of input sources, all tuples accessible of all sources (line 18) are selected. Thus, the set of contributing tuples is the union of all sets of input tuples per source (line 19).

## 5 Discussion

The proposed approach can infer provenance for *constant mapping* operations. However, *variable mapping* operations have not any fixed mapping ratio from input to output tuples. Therefore, the approach cannot be applied directly to these operations. One possible solution might be to transform these operations into *constant mapping operations* by introducing *NULL* tuples in the output. Suppose, for a *select* operation, the input tuple which does not satisfy the selection criteria will produce a *NULL* tuple in the output view, i.e. a tuple with a *transaction time* attribute and the remaining attributes are *NULL* values. We will give an estimation of storage overhead incurred by this approach in future.

Our inference algorithm provides 100% accurate provenance information under the assumption that the system is almost infinitely fast, i.e. no processing

delay. However, in a typical system due to other working load, it is highly probable that a new input tuple arrives before the processing is finished and our inference algorithm may reconstruct an erroneous processing window inferring inaccurate provenance. In future, we will address this limitation.

## 6 Evaluation

### 6.1 Evaluating Criteria and Datasets

The consumption of storage space for fine-grained data provenance is our main evaluation criteria. Existing approaches [8], [9], [10] 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[2] platform.

To implement the explicit fine-grained data provenance, we create one *provenance view* for each output view. This provenance view documents *output tuple ID*, *source tuple ID* and *source view* for each tuple in the output view. We also assign another attribute named as *tuple ID* which is auto incremental and primary key of the *provenance view*.

To check whether both approaches produce the same provenance information, explicit fine-grained provenance information is used as a ground truth and it is compared with the fine-grained provenance inferred by our proposed approach, i.e. the accuracy of the proposed approach.

For evaluation, a real dataset[3] measuring electric conductivity of the water, collected by the RECORD project is used . The experiments (see Section 2) are performed on a PostgreSQL 8.4 database and the Sensor Data Web platform. The input dataset contains 3000 tuples requiring 720kB storage space which is collected during last half of November 2010.

### 6.2 Storage Consumption

In this experiment, we measure the storage overhead to maintain fine-grained data provenance for the *Interpolation* processing element based on our motivating scenario (see Section 2) with *overlapping* and *non-overlapping* windows. In the non-overlapping case, each window contains three tuples and the operation is executed for every third arriving tuple. This results in about $3000 \div 3 \times 9 = 9000$ output tuples since the interpolation operation is executed for every third input tuple and it produces 9 output tuples at a time, requires about 220kB space. In the overlapping case, the window contains 3 tuples and the operation is executed for every tuple. This results in about $3000 \times 9 = 27000$ output tuples which require about 650kB. The sum of the storage costs for input and output tuples, named as sensor data, is depicted in Fig. 3 as dark gray boxes, while the provenance data storage costs is depicted as light gray boxes.

[2] `http://sourceforge.net/projects/sensordataweb/`
[3] `http://data.permasense.ch/topology.html#topology`

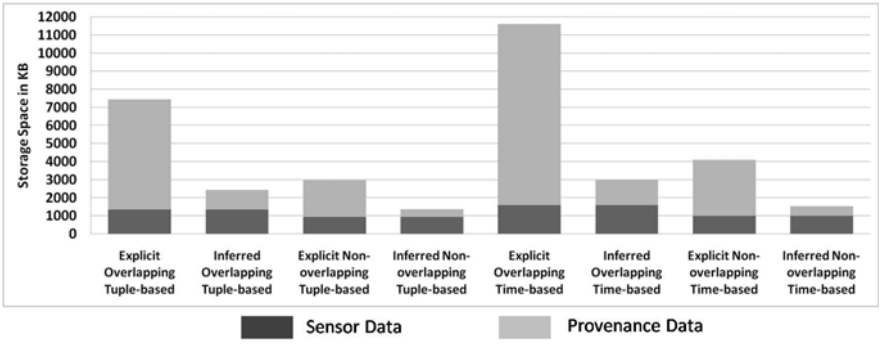


**Fig. 3.** Storage space consumed by Explicit and Inference method in different cases

From Fig. 3, we see that for *explicit* approach, the amount of required provenance information is more than twice the amount of actual sensor data in the best case (non-overlapping). On the contrary, the proposed *inference* approach requires less than half the storage space to store provenance data compared to the actual sensor data in non-overlapping cases and at least 25% less space in overlapping cases. As a whole, for *interpolation*, inferring provenance takes at least 4 times less storage space than the *explicit* approach. Therefore, our proposed approach clearly outperforms the explicit method. This is because the proposed approach adds only one timestamp attribute to each input and output tuple whereas the explicit approach adds the same provenance tuple several times because of overlapping sliding windows. Our proposed approach is not dataset dependent and also has window and trigger independent storage cost. The overhead ratio of provenance to sensor data depends on the payload of input tuples.

Additional tests confirm the results. We perform experiments for *project* and *average* operation with same dataset and different window size. In *project* operation, our method takes less than half storage space to maintain provenance data than the explicit method. For *average* operation, our proposed *inference* method 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. With the increasing window size and overlapping, our approach performs better.

### 6.3 Accuracy

To measure the accuracy, we consider provenance data tuples documented by explicit fine-grained data provenance as ground truth. Our experiment shows that the proposed *inference* method achieves 100% accurate provenance information. In our experiments, the processing time is much smaller than the minimum value of sampling time of data tuples, i.e. no new tuples arrive before finish processing, as discussed in Section 5). This is why, *inference* method is as accurate as *explicit* approach. These results are confirmed by all tests performed so far.

## 7 Conclusion and Future Work

Reproducibility is a requirement in e-science, decision making applications to analyze past data for tracing back problems in the data capture or data processing phase. In this paper, we propose a method of inferring fine-grained data provenance which reduces storage cost significantly compared to explicit technique and also provides accurate provenance information to ensure reproducibility. Our proposed approach is dataset independent and the larger the subsequent windows overlap, the more the storage reduction is. In future, we will address limitations in case of longer and variable delays for processing and sampling data tuples to ensure reproducibility at low storage cost.

## References

1. Simmhan, Y.L., Plale, B., Gannon, D.: A survey of data provenance in e-science. SIGMOD Rec. 34(3), 31–36 (2005)
2. Buneman, P., Tan, W.C.: Provenance in databases. In: SIGMOD: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 1171–1173. ACM, New York (2007)
3. Huq, M.R., Wombacher, A., Apers, P.M.G.: Facilitating fine grained data provenance using temporal data model. In: Proceedings of the 7th Workshop on Data Management for Sensor Networks (DMSN), pp. 8–13 (September 2010)
4. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 1–16. ACM, New York (2002)
5. Abadi, D., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal 12(2), 120–139 (2003)
6. Abadi, D., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. In: Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), Asilomar, CA, pp. 277–289 (2005)
7. Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. VLDB Journal 12(1), 41–58 (2003)
8. Sarma, A., Theobald, M., Widom, J.: LIVE: A Lineage-Supported Versioned DBMS. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 416–433. Springer, Heidelberg (2010)
9. Park, U., Heidemann, J.: Provenance in sensornet republishing. Provenance and Annotation of Data and Processes, 280–292 (2008)
10. Chapman, A., Jagadish, H., Ramanan, P.: Efficient provenance storage. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 993–1006. ACM, New York (2008)
11. Simmhan, Y.L., Plale, B., Gannon, D.: Karma2: Provenance management for data driven workflows. International Journal of Web Services Research 5, 1–23 (2008)
12. Wombacher, A.: Data workflow - a workflow model for continuous data processing. Technical Report TR-CTIT-10-12, Centre for Telematics and Information Technology University of Twente, Enschede (2010)