Facilitating Fine Grained Data Provenance using Temporal Data Model

Mohammad R. Huq
University of Twente
Enschede, The Netherlands.
m.r.huq@utwente.nl

Andreas Wombacher
University of Twente
Enschede, The Netherlands.
a.wombacher@utwente.nl

Peter M. G. Apers
University of Twente
Enschede, The Netherlands.
p.m.g.apers@utwente.nl

ABSTRACT

E-science applications use fine grained data provenance to maintain the reproducibility of scientific results, i.e., for each processed data tuple, the source data used to process the tuple as well as the used approach is documented. Since most of the e-science applications perform on-line processing of sensor data using overlapping time windows, the overhead of maintaining fine grained data provenance is huge especially in longer data processing chains. This is because data items are used by many time windows. In this paper, we propose an approach to reduce storage costs for achieving fine grained data provenance by maintaining data provenance on the relation level instead on the tuple level and make the content of the used database reproducible. The approach has prototypically been implemented for streaming and manually sampled data.

Keywords

E-science applications, Sensor data, Fine grained data provenance, Temporal data model

1. INTRODUCTION

Sensors have become very common in our day-to-day lives and are used in many applications. Sensor data are acquired and processed to higher level events used in applications for decision making and process control. Events are often processed continuously in a streaming fashion to facilitate ongoing processes. In many applications it is important that the origin of processed data can be explained to understand the semantics of the event and to reproduce events. Data provenance documents the origin of data by explicating the relation of input data, algorithm, and processed data. Thus, data provenance can be used to derive event semantics. Data provenance can be defined on data relation level called coarse grained data provenance or on data tuple level called fine grained data provenance [4].

Provenance is applied on different kinds of sensor data: Streaming data is continuously produced data, while manually sampled data is a small set of data produced at a particular point in time [1, 6]. While streaming data is never updated, sampled data might be updated.

In case of fine grained data provenance, storage cost is linear with the number of sensors and processed data. Stream processing is often based on sliding windows resulting in a single data tuple contributing to many processed data. As a consequence, fine grained or tuple-based data provenance has to refer to a single tuple multiple times depending on the overlap of two subsequent sliding windows. Thus, the storage costs for tuple-based data provenance with many overlapping data in a sliding window can result in a multitude of provenance data related to the actual sensor data. The aim of this research is to provide tuple-based data provenance functionality with reduced storage costs to keep data provenance in streaming scenarios manageable.

Though the volume of manually sampled data is much less than streaming data, manually sampled data can be updated. If a piece of data is updated or deleted from the database, relation-based data provenance cannot extract the original data again. Tuple-based data provenance can solve this problem. But once there will be an update, new provenance data should be also preserved. Moreover, manually sampled data are often combined with streaming data and processed together to achieve more meaningful results. Thus this combination will end up with high volume of provenance data to be maintained compared to the actual sensor data. Therefore, low-cost tuple-based data provenance functionality should be realized in an environment where both streaming and manually sampled data are handled together.

Our proposed approach provides tuple-based data provenance with reduced storage costs by maintaining relation-based data provenance and using a temporal data model. We add timestamps to each tuple which allows us to retrieve a particular database state based on a given timestamp. Then using coarse grained provenance data, we can figure out the original tuples participated in a query to produce output tuples. The additional storage costs of these temporal attributes along with the cost of relation-based data provenance together will not exceed the storage costs for tuple-based data provenance. Furthermore, we develop a prototype combining streaming and manually sampled data to realize our approach.

This paper is structured as follows. In Section 2, we discuss related work. Next, we provide a detailed description of our motivating scenario followed by the problem description in Section 4. In Section 5, we provide the structure of our temporal data model followed by the implementation

VLDB '10, September 13-17, 2010, Singapore

that demonstrates the viability of our approach in Section 6. Finally, we conclude with the hints of some future work.

2. RELATED WORK

Recently issues pertaining to data provenance are getting more attention from researchers. In [3], authors have described a data model to compute provenance on both relations and tuples level. In this data model, the location of any piece of data can be uniquely described by a path. This paper shows case studies for traditional data but it does not address how to handle streaming data and associated overlapping windows. In [10], authors have provided a data model for provenance repository which is based on relational database. Their approach maintains relation-based data provenance whereas our approach provides fine grained data provenance.

Relation-based data provenance cannot reproduce results. The work described in [12] proposed a data and collection model using timestamp based approach to collect provenance information when there is any change in sampling rate and accuracy of the stream. Though it saves a lot of disk space, it cannot address update of sampled data. We use multiple timestamps to identify the validity of a particular (updated) tuple. In [5] authors have presented an algorithm for lineage tracing in a data warehouse environment. They have provided data provenance on tuple level. Their algorithm only works for traditional data. It cannot address the issue of database state change due to update.

For databases that change over time, compact versioning is essential to recover data referenced by the provenance of data derived from an earlier version of the database [2]. Since one version is an extension of the previous version, this versioning technique incurs space overhead. Our proposed approach does not store any versions physically in the disk instead we attach timestamps to each tuple to retrieve any database state based on a given timestamp.

In e-science applications, supporting reproducibility of research results are necessary. In [8], authors outline the structure of a provenance-aware storage where provenance data will be treated as the first class data. For recording and querying provenance data Tupelo2 project [7] has been initiated. This project is aimed at creating a metadata management system which stores annotation triples (subject-predicate-object) in several kinds of databases, including normal relational databases. Tupelo2 cannot address issue with update operation.

Recently, a complete DBMS, LIVE [9] can store base and derived relations with simple versioning capabilities where each tuple includes a start and end version number. LIVE also preserves the lineage of derived data items. Since LIVE uses different version number associated with each relation, we cannot retrieve the overall database state given a single version number. Our approach of using timestamp instead of version number overcomes this drawback. Most significantly, in our approach, we need not maintain any tuple-based provenance data instead we store relation-based provenance data along with temporal data model which is more cost effective than LIVE.

3. SCENARIO

Figure 1 depicts a Bluetooth localization scenario that has been set up in our SensorDataLab [17]. The location of a user is determined by acquiring the signal strength of a Bluetooth device carried by the user and the known location of the system acquiring the signal strength measurement. Linksys NSLU2 devices are used for acquiring signal strength measurements of all Bluetooth devices. On these systems, a Bluetooth discovery application is installed which continuously checks for handheld devices and reports detected devices via a UDP packet to the data processing system. A packet contains the person's device MAC address, identification number of the discovery systems, signal strength, and the timestamp (see figure 6). The NSLU2 systems are represented as EWI 1148, EWI 1149, EWI 1150.

In addition, the deployment location of NSLU2 device as well as the mapping of MAC address of a handheld device to an actual person at a specific point in time is documented and made available as manually sampled data. The data processing allows to correlate streaming and sampled data and provides a query interface to access the data on-line and off-line.

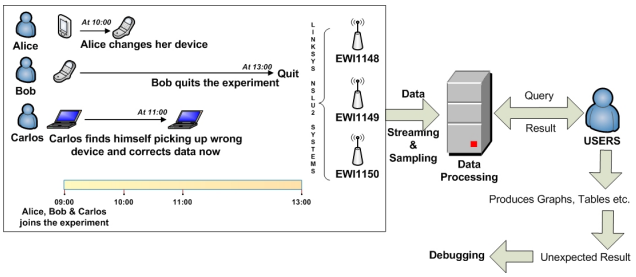

Figure 1: Bluetooth localization scenario in SensorDataLab

In our scenario Alice, Bob, and Carlos are three users, joining the experiment on 2010-03-03 at 9:00. Each of them using a mobile device. It turns out that the mobile device of Alice has not been charged over night and is running out of power. Therefore, at 10:00 she has to exchange the mobile device with another one. Bob has to attend a lecture in the afternoon and therefore is leaving the experiment at 13:00. At around 11:00 Carlos finds out that he picked up a new, unused mobile device instead which had been assigned to him. Since Carlos likes this mobile device better, the device had been permanently assigned to Carlos.

The supervisor of Alice, Bob and Carlos uses the data for publishing a paper about a new localization approach tested in this experiment. From the available data, she evaluates her approach and creates graphs to document the results. If the outcome is unexpected, she may want to debug the results. Thus, this scenario exhibits the properties of an e-science scenario, since she must be able to reproduce the evaluation results and graphs later. The requirement of reproducible results corresponds to tuple-based provenance in the scenario as described above because it documents how each tuple has been created. For the rest of this paper, we will explain our approach to achieve fine grained data provenance based on this scenario.

4. PROBLEM DESCRIPTION

4.1 Fine grained Data Provenance

In our scenario (see Section 3), each second a lot of streaming data is arriving from different sensor nodes. Moreover

manually sampled data including metadata are also stored to help the overall data processing job. We have 8 NSLU2 devices installed in our lab. Whenever users are roaming around the lab, each second a UDP packet is sent containing the user's device MAC address, detection timestamp along with other parameters.

Now, consider a time-triggered query to compute a person's location based on the readings of the last 30 seconds. This translates in a continuous query having window size of 30 seconds. At each second, 8 different tuples/packets will be sent by those NSLU2 devices. After each second, the window shifts forward by a second. Therefore, at a particular moment we have 240 data tuples which should be processed to compute that time-triggered query. For each data tuple, we associate provenance data using a pointer to the tuple represented as *bigint* field. Moreover, we need two more pointers to point to the activity and the resulting output tuple assuming there is only one output tuple. In total, we need to preserve 242 pointers in order to have the tuple-based data provenance. In MySQL[15], the *bigint* field consumes 8 bytes. The output will be current location of a particular person which is nothing but a co-ordinate in form of (x,y) and consumes 8 bytes in total. Therefore, the ratio of the provenance data to processed data is 242:1 per processed data tuple in this scenario. In other words, only 4 gigabytes of a 1 terabyte disk will be used to store the sensor data and the rest of the space will be consumed by provenance data. Moreover, provenance data is a type of indirection used to identify the original data which has no significant meaning to users. Therefore, buying additional storage space seems to be an expensive solution to this problem.

Based on the example described above in this section, relation-based data provenance needs to preserve only three provenance data. One for the set of input data, another for the query and the rest is maintained for the output. For relation-based data provenance, the ratio of provenance data to actual desired sensor data is 3:1 per processed query and it is independent of overlapping window size between two subsequent windows and number of tuples per second. Therefore, from the storage point of view relation-based data provenance is more efficient than tuple-based data provenance.

4.2 Reproducible Results

Reproducibility of results can be achieved by having different versions of a database - a new version after every change of the database. Traditionally, versioning is implemented by replicating the complete database before applying the change. An alternative way is to document changes in the database according to time. Timestamps can be used as a global version number. Using timestamp, we can provide a particular state of the database without storing versions physically. In this paper, we achieve reproducibility by using timestamps as version numbers and requiring a *consistency* property on the database which ensures for a query on a particular database state in the past to have the same result set regardless of the query execution time. The definition of *consistency* is given below.

Definition 1. If a particular query is executed on the same database state by same/different users at different points in time, users are expecting to have the same result sets each time under the assumption that the query processing is not hindered by any means of network volatility.

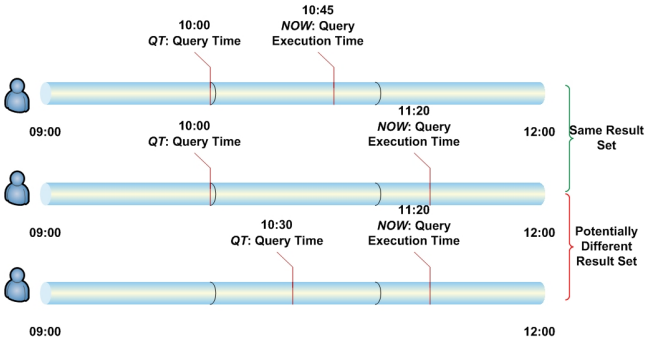

Figure 2: Query Time and Query Execution Time

In the definition, the term *same result set* refers to the same set of tuples extracted from the same set of relations from participating nodes for the same query executed at different points in time. Figure 2 pictorially represents definition 1. Assume that a user wants to know the location of Alice on a particular point in time which can be termed as *query time,* QT is represented as query Q at QT. In the upper pair of timelines, a user submits the query Q at QT=10:00. Regardless of their query execution time which is represented as NOW, the outcome should be the same since they queried on the same database state that is available on 10:00. On the other hand, QT is different for the lower two timelines. The middle timeline shows that the user submits query Q at QT=10:00 and the last timeline depicts that the user submits Q at QT=10:30. Though these two queries are executed at the same point in time, the result set may be potentially different if there is a change on Alice's handheld device. In other words, if there is a database state change we may get potentially different results for a particular query. The consistency property is also applicable for continuous queries. Reconstructing the window having same set of tuples and trigger condition will ideally produce the same result set each time irrespective of the *query execution time*. This is how definition 1 differentiates between *query time* and *query execution time* which allows users to have the same set of data retrieved as a result of the query depending on the point in time for which they want the query result, but irrespective of the time when the query has been executed. It fulfills the requirement of retrieving historic data as well as provides a consistent view of the database.

4.3 Data Classes

In our scenario, we have both streaming and manually sampled data. Streaming data is automatically acquired from sensors which is not the case for manually sampled data. The volume of streaming data is much larger than the volume of sampling data. Manually sampled data or metadata are associated with streaming data which make them important to preserve into the database. Sometimes, there is a large time delay between a fact becomes valid in the real world and that fact is inserted into the database. This delay is known as *propagation delay*. Since human intervention is needed to enter sampled data in database, it may have longer propagation delay than streaming data. Moreover, data processing is also challenged by update of

sampling data. Streaming data, on the other hand, never updates but due to its high volume, it is a challenging task to provide tuple-based data provenance in streaming scenarios. Next sections will discuss these problems associated with streaming and sampling data in detail.

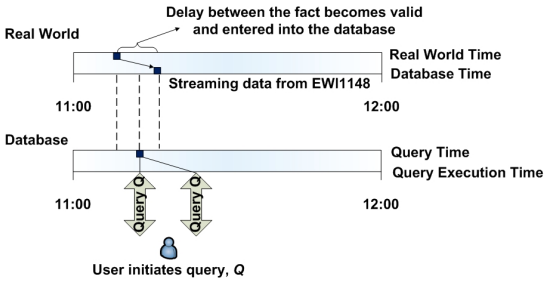

Figure 3: Propagation delay of streaming data

4.4 Propagation Delay in Streaming Data

In Figure 3, we have one continuous timeline and two different worlds: real world and database world. The upper block of timeline shows that the data tuples generated from EWI 1148 has delay between the time they got valid and entered into the database due to the propagation delay. This delay can affect the data processing steps and eventually the outcome. In the lower block of the processing timeline, we see a user initiates a query Q with same *query time* on tuples generated by EWI 1148 in two different points of *query execution time*. When the query Q is executed for the first time, as tuples are yet to enter in the database the outcome contains no result which is unexpected. The set of data arrives later after the first query execution and influence the outcome of the next execution of the same query Q. Maintaining relation-based provenance in the aforementioned scenario, cannot extract original data to reproduce results.

4.5 Updates in Sampled Data

Sampling data may be updated or modified over time. Figure 4 shows an example where user Alice changes her handheld device at 10:00 (see section3). After changing the device, the related data on Alice's new handheld device (e.g. device MAC) is entered into the database and overwrites the previous data. This operation indicates a state change for the overall database. Now, if a query Q on Alice's handheld device is executed on two different points in time (different query execution times), we will get two different results because of the state change of the database. Therefore, this update operation in the database causes to have inconsistent results and thus creates inconsistency in the database.

5. TEMPORAL DATA MODEL

One of the major challenges to preserve our consistency property 1 is to allow query execution on the same database state. That's why, we use a temporal data model to avoid storing of all the previous versions physically. Using a temporal model, we can retrieve any particular state of the database based on a given timestamp since each data tuple will be associated with temporal attributes. Our proposed data model is actually inspired by the bi-temporal data model [11] using the following temporal attributes:

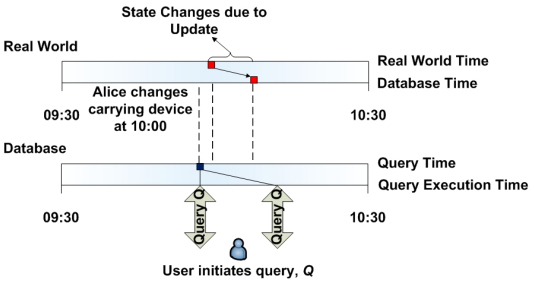

Figure 4: Update of sampled data

- **valid time** representing the point in time a sample has been taken or a measurement has been sensed.
- **transaction time from** is the point in time the tuple has been inserted in the database.
- **transaction time to** representing the point in time the tuple is marked as deleted without physically deleting it.

These temporal attributes allow users to initiate queries on a database mentioning a specific timestamp. Next, we discuss the way of executing some most common database operations based on our data model. Since streaming data never changes, only *insert* operation is applicable for streaming data.

- **Insert**: A tuple is added in the database for the very first time with specified *valid time* and *transaction time from* being the current point in time (e.g. Alice, Bob and Carlos join the experiment). The value of *transaction time to* is set to '0:00:00'.
- **Update**: It addresses the situation whenever a user would like to rectify wrong data given earlier (e.g. Carlos uses one device but another device was registered for him). *Transaction time to* of the existing tuple is set to $NOW - 1$ and a new tuple is added to the database with same *valid time* as the existing tuple. *Transaction time from* is set to NOW and *transaction time to* is set to '0:00:00'. The difference from *change of data* operation is that here the *valid time* of existing and new tuples are same.
- **Delete**: It refers to the incident that causes damage or complete removal of a particular entity from the scenario (e.g. at 13:00 Bob is not participating in the experiment anymore). In the tuple describing the participation of Bob in the experiment is updated by setting the value of *transaction time to* to NOW-1.

One of the principle requirements of our data model is to preserve all the past data in order to execute queries on a given database state so that we can maintain consistency according to the given definition 1. We are not going to delete or modify any existing tuples in the database rather we will insert new tuples with different *valid* and *transaction times*. Therefore, all these database operations need to be handled in a different manner than a traditional database does.

6. IMPLEMENTATION

6.1 Prototype

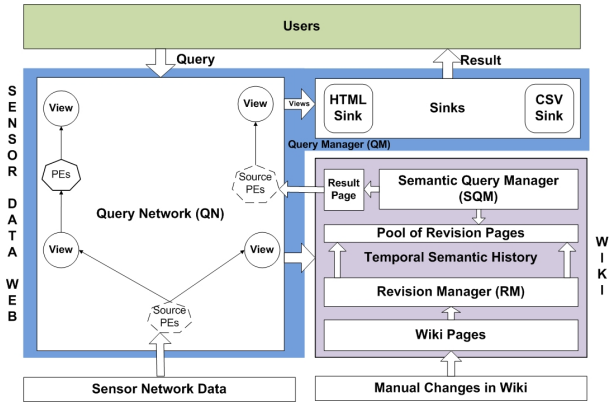

Figure 5: Architecture of prototype

We build a prototype to validate our approach of achieving fine grained data provenance for both streaming and sampled data which ensures to reproduce query results. We make use of Sensor Data Web [18] for gathering, processing and publishing sensor data. We perform some modifications on the existing java code so that we can realize and execute our proposed approach.

Figure 5 shows the basic building block of the platform. In sensor data web platform, *Query Manager* (QM) is responsible for collecting streaming data from sources like GSN [13] and sampling data from the wiki via a Sparql end point. Within the query manager a query network is generated, consisting of several processing elements (PEs). Some of them are source PEs which can communicate and receive streaming data directly from nodes in the sensor network or pull information from external sources. Every PE presents output as a view. A view is not considered as the final outcome since it can be input for another PE. We modify the structure of original views to ensure that the *transaction time* for each data tuple is now included into the views. Users can request results in a preferred format like as a HTML page or a CSV document. This request is handled by *sinks* which return results to users in requested format via specific sink (e.g. HTML sink, CSV sink). We add one extra parameter *query time* in each sink structure so that each of these sinks now can return results based on the given *timestamp*.

Sensor data web can also interact with external sources to pull sampled data according to a given query. In order to manage sampled data, we use MediaWiki [14] as our basic platform. One of the main reasons for choosing MediaWiki is to collaborate with different metadata and sampled data owners. As wiki is well known for it's community based use, using wiki as the repository of sampling data would be an easy way to collect those data. On the top of MediaWiki, we use semantic mediawiki extension [16] on top of which we build our own semantic wiki extension known as Temporal Semantic History [19].

Our developed extension tracks and monitors the content of each page in wiki. Data are changed manually in a wiki page. Revision manager (RM) preserves the previous content after each revision done on a particular page according to *timestamp* in a new *revision page*. Each revision page keeps the value of (e.g. *valid time, transaction time from* and *transaction time to*) along with other data. The value for *transaction time from* and *transaction time to* are added into the wiki page by the system itself. These revision pages together form the pool of revision pages. When a query Q requests data from wiki, it is redirected via *query network* to this extension. *Semantic query manager* (SQM) chooses appropriate revision pages from the pool according to the user given timestamp in Q. Then the content of selected revision page is transferred and displayed in the result page. This data is provided as input to one of the source PEs which can handle sparql data. Then the data is further processed and result is sent to users.

timestamp	system_id	mac_address	rssi	valid time	transaction time
1267610610	ewi1148	00:15:F2:B6:C4:D5	75	2010-03-03 11:03:30	2010-03-03 11:03:36
1267610638	ewi1148	00:15:F2:B6:C4:D5	77	2010-03-03 11:03:58	2010-03-03 11:04:06
1267610669	ewi1148	00:1E:45:D8:C0:D1	78	2010-03-03 11:04:29	2010-03-03 11:04:36
1267610696	ewi1148	00:1E:45:D8:C0:D1	81	2010-03-03 11:04:56	2010-03-03 11:05:06
1267610731	ewi1148	00:15:F2:B6:C4:D5	73	2010-03-03 11:05:31	2010-03-03 11:05:35

Figure 6: A set of streaming data

6.2 Use Case

In the proposed data model, each data tuple is associated with temporal attributes irrespective of their sources and types. Figure 6 shows a set of streaming data produced by one of the NSLU2 systems, EWI 1148 in our scenario. The temporal attributes *valid time* and *transaction time* (as an abbreviation of *transaction time from*) are added for each tuple. *Transaction time to* is not needed for streaming data, since we consider *append-only* streaming data.

On the other hand, sampled data (e.g. manually sampled data, metadata) is stored in a semantic wiki. In a wiki, data is stored in form of SPO (Subject-Predicate-Object) triples. Figure 7 depicts that for each entity, a unique wiki page is created based on the *candidate key*. As for example, we have three different persons in our scenario: Alice, Bob and Carlos and a unique wiki page is created according to the person name. In triple store, for all triples, *subject* contains name of the page. Moreover, once a wiki page is modified, the page content before the revision is preserved in order to provide original data upon user requests. The name of these revision pages depends on the original page name and *transaction time from*. Moreover, '0:00:00' in *transaction time to* attribute is used as a pattern to indicate that the tuple is currently valid.

Sampling data may be updated and deleted over time. As discussed earlier, sampling data is organized in a semantic wiki which has different data organization technique. Among the different database operations, *update* is a more interesting operation for sampling data. In figure 7, Alice changed her device after a while which is indicated by tuple no. 4. As overwrite existing data causes problems to retrieve original data, we insert another tuple having different *valid* and *transaction time from*. Before inserting the new tuple, we update *transaction time to* of the previous tuple to NOW-1.

6.3 Discussion

Our approach achieves fine grained data provenance with reduced storage costs. Consider the set of streaming data in figure 6. If we perform any *select*, *project* or *join* operations on that dataset, we will get output tuples. Now,

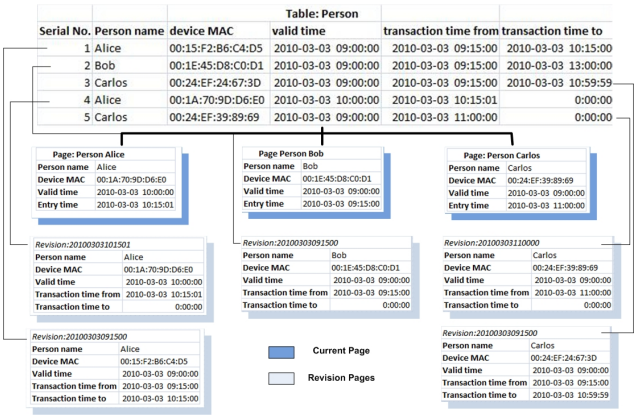

Figure 7: Organization of Sampled data in Wiki

based on a user given timestamp, we can retrieve original database state at that point in time. Then, we will use coarse grained provenance data to figure out the tuples from input dataset which participated in the query to produce output data tuples. This is how, we can achieve fine grained data provenance with reduced storage costs by maintaining coarse grained data provenance and applying temporal data model.

In section 4.1, a comparison of consumption of storage space between fine grained and coarse grained provenance data has been given. In our prototype, for each tuple, we need at most three timestamp attributes which take at most 12 bytes storage space per tuple and it is independent of window size, size of the overlap of the windows, and number of tuples per second. In tuple-based provenance, each data tuple is associated with provenance data which is a pointer to the tuple itself and since a particular data tuple is participating in the query execution for several times depending on the overlap of subsequent sliding windows, the space consumed for provenance data is much larger than our proposed approach. If there is no overlap between subsequent sliding windows, our approach incurs extra disk space (8 bytes per tuple) as much as fine grained provenance does. Though our prototype requires more space than relation-based data provenance, it enables users to have reproducible results and overcomes drawbacks of relation-based data provenance.

We assume the *append-only* data stream processing engine in our scenario. There are some stream processing engines which act as *non append-only*. In those cases, our solution will handle stream data in a similar way of handling manually sampled data.

7. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an approach to achieve fine grained data provenance with low storage costs. To achieve our goal, we maintained relation-based data provenance along with timestamp-based logical versioning of the database. The proposed approach is mainly beneficial for streaming data, thus data processed on-line. However, the approach allows us also to combine and correlate streaming and sampled data. The proposed approach has been implemented for both of the streaming and sampled data which shows the viability of our approach.

In future, we would like to compare performance (i.e. storage cost, response time) of our approach to any existing techniques.

8. REFERENCES

[1] D. Brus and M. Knotters. Sampling design for compliance monitoring of surface water quality: A case study in a polder area. *Water Resources Research*, 44(11):95 – 102, 2008.

[2] P. Buneman, S. Khanna, and T. Wang-Chiew. Data provenance: Some basic issues. *Foundations of Software Technology and Theoretical Computer Science*, pages 87–93, 2000.

[3] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. *Database Theory - ICDT 2001*, pages 316–330.

[4] P. Buneman and T. Wang-Chiew. Provenance in databases. In *Proc. Intl. Conf. on Management of data*, pages 1171–1173, New York, NY, USA, 2007. ACM.

[5] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, vol. 12, pages 41–58.

[6] J. de Gruijter, D. Brus, M. Bierkens, and M. Knotters. *Sampling for natural resource monitoring*. Springer Verlag, 2006.

[7] J. Futrelle. Tupelo Server. Website. `http://tupeloproject.ncsa.uiuc.edu/`.

[8] J. Ledlie, C. Ng, D. A. Holland, K. kumar Muniswamy-reddy, U. Braun, and M. Seltzer. Provenance-aware sensor data storage. In *Workshop on Networking Meets Databases (NetDB)*, 2005.

[9] A. Sarma, M. Theobald, and J. Widom. LIVE: A Lineage-Supported Versioned DBMS. In *Proc. Intl. Conf. on Scientific and Statistical Database Management*, 2010.

[10] M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 603–620.

[11] C. K. University and C. Koncilia. A bi-temporal data warehouse model. In *Proc. Intl. Conf. on Advanced Information Systems Engineering*, pages 77–80, 2003.

[12] N. N. Vijayakumar and B. Plale. Towards low overhead provenance tracking in near real-time stream filtering. In *Provenance and Annotation of Data*, pages 46–54, 2006.

[13] Website. Global Sensor Network. `http://www.swiss-experiment.ch/index.php/GSN:Home`.

[14] Website. Mediawiki. `http://www.mediawiki.org/wiki/MediaWiki`.

[15] Website. MySQL. `http://www.mysql.com/`.

[16] Website. Semantic Mediawiki Extension. `http://www.mediawiki.org/wiki/Extension:Semantic_MediaWiki`.

[17] Website. Sensor Data Lab. `http://www.sensordatalab.org/wiki/index.php5/Loc:Home`.

[18] Website. Sensor Data Web. `https://sourceforge.net/projects/sensordataweb/`.

[19] Website. Temporal Semantic History. `http://www.sensordatalab.org/wiki/index.php5/Extensions:Temporal_Semantic_History`.