Scalable Data Citation in Dynamic, Large Databases: Model and Reference Implementation

Stefan Pröll
SBA Research
Vienna, Austria
sproell@sba-research.org

Andreas Rauber
Technical University of Vienna
Vienna, Austria
rauber@ifs.tuwien.ac.at

***Abstract*—Uniquely and precisely identifying and citing arbitrary subsets of data is essential in many settings, e.g. to facilitate experiment validation and data re-use in meta-studies. Current approaches relying on pointers to entire data collections or on explicit copies of data do not scale. We propose a novel approach relying on persistent, timestamped, adapted queries to versioned and timestamped data sources. Result set hashes are used for validation correctness on later re-execution. The proposed method works both for static as well as dynamically growing or changing data. Alternative implementation styles for relational databases are presented and evaluated with regard to performance issues and impact on existing applications while aiming at minimal to no additional effort requirements for data users. The approach is validated in an infrastructure monitoring domain relying on sensor data networks.**

I. Introduction and Motivation

In many settings it is essential to be able to precisely and preferably automatically identify specific subsets of data that, e.g. was used in a given study, served as the basis for training a specific model, or served as input to subsequent processing. When any of these computations should be repeated for verification purposes in the future, providing access to the exact subset of data is a core requirement. Data citation is essential in various settings in academia, research and the business realm. Not only is giving credit an established standard, it is also economically reasonable to reuse and share data [1]. Therefore data citation also provides additional value by providing mechanisms for verifying and validating experiments. In many areas understanding how a result was achieved is required, especially when it comes to critical IT infrastructure. Not only do publications and supplementary data need to be citable, but also the attribution of the creation of specific subsets is a demand.

Many data citation approaches exist that allow to reference data sets. So far data sets are usually referenced as one unit having a textual description attached that serves as metadata source. These descriptions of such subsets can often not be generated automatically and they are often not machine interpretable. As a result these descriptions are often not precise enough to be useful. Also the amounts of data are growing increasingly in size and complexity, therefore referencing whole data sets as one unit becomes impractical. Hence this approach does not scale with increasing data set sizes. Also records evolve over time. They can be updated or deleted, which is hardly reflected in subsets and their citations. This is often not suitable for many applications, because live data that is still evolving while previous versions are still available, needs to be identifiable as well. We need data citation solutions that can be acted upon automatically, i.e. machine readable data citation tools that scale with growing amounts of data.

In this paper we present a mechanism for making evolving data citable. Our mechanism allows the definition of subsets and their citation by assigning persistent identifiers to modified database queries, which define the subset of data to be used. It ensures reproducible queries without having to store complete result sets externally, hence the approach is scalable even for large data sets. Our approach supports long term preservation by keeping data sets accessible.

The remainder of this paper is organized as follows. In Section II we give an overview of the current state of the art. In Section III we describe how existing relational databases can be rendered citable. Section IV demonstrates a use case based on sensor network data followed by conclusions in Section V.

II. Related Work

The reproducibility of scientific experiments is often impeded by missing sample data [2]. Data disappears when the link between its reference and the actual data breaks, i.e. standard URLs are used for denoting the location of the data. Because of the link rot phenomenon [3], URLs are not suitable for long term reference. Therefore persistent identifiers have been introduced that guarantee the availability of the link target by maintaining the relationship between resource and link by central authorities. Many different standards exist [4]. Most of these concepts origin from the scientific or bibliographical domain and are mainly used to provide persistent identifiers for arbitrary digital objects. Although there is need for the reference of subsets of data [5] in the scientific community, hardly any methods exist that allow referencing arbitrary subsets without their definition in

advance. So far, many data sets are cited as complete entities with natural language text sometimes being used to describe the kind of subset used. In order to overcome the limitations of entire data set citation, some approaches propose to assign PIDs to individual data items, or even individual attributes. While this supports very fine-granular citations, the solution does not scale. Furthermore many data sets continue to grow and can be updated. Therefore versioning is an integral part of any data citation framework.

Our approach relies on concepts from temporal databases, which have been around for a long time [6], [7] and are still a topic of current research [8]. In this paper we want to bridge the gap between digital preservation (temporal) data and data citation.

III. Making Data Citable

Relational database management systems (RDBMS) are implemented in many scenarios and drive data storage, access and analysis. Therefore we will focus on this database model for our dynamic data citation approach. Enhancing data citation capabilities and enabling long term access requires a specific database design, which will be introduced in the following sections.

In [9] we provided an initial draft of a generic model for citing data sets. The model is based on timestamped SELECT-queries which are used in order to retrieve the relevant data. These SELECT-statements can be used for data citation as long as versioned data is available and all data manipulation language (DML) statements are provided with the timestamp of their execution. Data is never deleted (except in the case of e.g. legal requirements for deletion, which need to be documented) and the previous state of a row within a data set is maintained during the whole life cycle. Hence the state of the data set that was valid at the execution time can be reconstructed. The generation of citable result sets is query centric, hence not the data itself needs to be explicitly stored, but the query statements are preserved. Our model supports the citation of both static and dynamic data that gets updated and changed. This provides a very flexible citation solution that enables the provision of references in a timely manner. Basic requirements for a model of dynamic data citation from a database perspective are:

1) Unique subsets can be expressed (i.e. a primary key is available)
2) Dynamic data can be handled (inserts, updates and deletes are recorded)
3) Nothing is actually deleted from the system (except legal requirements enforce actual deletion. This would require to mark affected queries as non-reproducible.)

As a result, the evolution of data is traceable, citations can be managed in a scalable and automated way and the model does not require vendor specific features. A further detail that is crucial for the acceptance of dynamic data citation is transparency and the effort required for implementing the model for a given data source, both from the point of view of the operator as well as from a user's perspective. We propose three different approaches for storing the required metadata, all described in more detail, comparing their advantages and disadvantages in Section III-A. The implementation of this model should be as non-invasive with regard to the existing applications interacting with the DB at the cost of slightly more than doubling storage requirements.

The following steps describe the actions necessary for creating and retrieving a citable data set:

1) Record timestamps for all INSERT, UPDATE and DELETE statements
2) Maintain the history of all inserts, updates and deletes with the original values
3) Store all (adapted) SELECT-queries identifying subsets of data that need to be persistent and citable
4) Create a hash of the result set
5) Assign a PID to this query which serves as identification for the citation record

We thus need to first ensure that the data is stored in a timestamped and versioned manner (c.f. Subsection III-A). Then, each query that needs to be persistently stored is expanded with its execution timestamp and modified to ensure unique sorting (c.f. Subsection III-B) and assigned a PID (c.f. Subsection III-C). Hashes over the result set are computed to allow verification of correctness of the returned data at a later point in time (c.f. Subsection III-D)

The last two steps are the actual retrieval of the historical data set and its verification:

6) Re-execute the (adapted) stored query and retrieve the original data
7) Verify the correctness of the data retrieved using the hash key computed on the original data set

After the database schema has been enhanced to be compliant with the model, inserts, updates and deletes within databases can be executed fully transparent. Queries that lead to result sets that need to be persistently stored for later re-use can be stored and are assigned a PID for future identification. Whenever a data set needs to be reconstructed in the future, the query with the timestamps and the versioned data is sufficient to retrieve the exact same data set.

A. Adapting the Database Tables

In some settings, the database may already utilize timestamps to record when new data was being added. Also, versioning of data, recording updates and deletes, is already quite common in many big data applications, especially as in analytical settings these are frequently dominated by inserts of new data rather than corrections or deletions to existing data. In such cases, usually no specific steps are necessary on the database layer. Otherwise, we need to turn the database into a timestamped and versioned database.

There exist many approaches for handling temporal data [10]. The evolution of data needs to be traced by capturing DML statements. The implementation scenarios rely on additional columns for the current version, user information and the event type as well as a timestamp of the last event. Storing such additional metadata for each table introduces significant overhead that needs to be handled besides the original records. We introduce three approaches for handling the metadata required for data citation: (1) integrated, (2) hybrid and (3) separated.

The first approach (1) requires to extend all original tables `t` by the temporal metadata and expand the primary key by the versions column. All operations that alter data need to be reflected in the original table. This method is intrusive, because other tables and applications need to be adapted to this alteration. Other queries need to refer to the latest version of a record only and ignore the older versions, also INSERT-statements need to be timestamped. Requiring all queries to incorporate the new primary keys and versioned data is challenging. This modification can have serious effects on the performance, as other queries require to sort-out outdated data. Nevertheless this method can be used in scenarios where deletions and updates hardly occur, e.g. sensor data that produce time series. The same is true for log-data which are optimized for write rather than read operations. This approach will usually require a major redesign of the database, as well as the interfaces of programs accessing it. As an advantage, the retrieval of the earlier data state then requires a simple query `q'` that selects the latest version per record and omits deleted records. From a storage perspective this approach only produces low overhead as records only append timing and versioning data.

A hybrid approach (2) moves records into a history table `h` whenever a record gets updated or deleted, but inserts are only recorded with a timestamp in the history table. This approach facilitates a history table `h` having the same structure as the original table `t`, but includes columns for data citation metadata. The original table `t` always reflects the latest versions as if no versioning and timestamping was enabled, whereas the history table records the evolution of the data. If a record has been inserted but not updated or deleted, no additional metadata or versioning is required. The advantage of this approach is again a minimal demand for storage, especially when data is hardly updated but lots of inserts occur. But more importantly, the approach is non-intrusive, as table `t` remains unchanged. A disadvantage is a more complex query structure for retrieving historical data, because the system needs to check whether there exist updates or not and then retrieve the records either from `t` or from `h`. Yet, in settings where the re-execution of historic queries is a low-frequency event, this might be the preferred solution.

The full history approach (3) also uses a history table as described before, but this approach also copies records that have been inserted into table `t` immediately over into the history table `h` and marks them as `inserted`. Deleted records are again only marked as deleted in the history table `h` and removed from the original table `t`. An advantage of this approach is that table `t` remains unchanged again and that it facilitates a simple query structure as no joins are required. All requests for data citation are handled by the history table `h` whereas the original tables is not involved in the data citation process. This also reduces the size of the indices of the original table. Hence the performance for general SQL statements is not influenced as the history values are only considered when they are required for data citation. The drawback of this approach is its huge increase in storage size by keeping virtually a historicizing copy of the entire data. Many RDBMS such as PostgreSQL or Oracle support timestamped and versioned data off the shelf, hence such an approach might be implemented with standard tools. Table I shows a comparison of the three table designs [11].

Table I
COMPARISON OF TABLE DESIGNS

	Intrusiveness	Storage demand	Query complexity
Integrated	high	low	low
Hybrid	low	medium	medium
Separated	low	high	low

The granularity of this approach is on a record level, hence every change on the data is reflected. In our model, the timestamp contains the explicit date at which the data has been changed with the respective updates being executed as atomic transactions.

B. Ensuring Unique Sorts

A crucial aspect of result sets is their sorting. The results returned by a database query are commonly fed into subsequent processes in their order of appearance in the result set. In situations where the order of processing has some effect on the outcome (e.g. in machine learning processes), we need to ensure consistent sorting of the result sets is provided. Hence, the order in which records are returned from the database needs to be maintained upon a query's re-execution. This is challenging as relational databases are inherently set based. According to the SQL99 standard[1], if no ORDER BY clause is specified, the sorting is implementation specific. Even if an ORDER BY clause is provided, the exact ordering may not be defined if there are several rows having the same values for the specified sorting criteria.

For this reason queries need to have a sorting order specified in order to be compliant with our model. As a basic mechanism, we only consider standard sorting mechanisms, i.e. using ORDER BY clauses. If no such sorting behavior is determined, a standard sorting based on the primary key

[1]http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

columns of the involved tables is appended to the query. This covers only basic SQL SELECT statements, but is sufficient for the most use cases requiring data citation. Additionally means to mitigate non-determinism and randomness are required when preparing and re-executing processes at a later point in time for validation purposes [12]. Randomized functions or relative time specifications are still a problem when they are used in stored procedures or SELECT-statements. So far, there exists no standard solution that can mitigate the effects of randomness and non-determinism.

C. Assigning PIDs to Queries

It is essential to detect queries that have already been captured by the system in order to avoid the multiple assignment of new PIDs to identical queries. Furthermore the result sets of queries have to be analyzed to verify the correctness. We rely on hashes to determine identity and correctness of result sets. There are two options for computing the hash key for a query result. The simplest solution requires computing the hash key over the entire result set. This allows straightforward comparison to see whether two result sets are identical. Yet as result sets are potentially huge, hash computation can be quite expensive. A more efficient alternative requires the existence of a unique identifier for each row in the result set. In this case we can compute the hash key over the unique key and the table columns (headings) to be able to ensure that two result sets are identical.

If a new query or an altered result set was detected, then a PID can be assigned. Any of the mechanisms introduced in Section II is suitable for assigning PIDs for the later identification of queries. A further possibility is to generate a PID by creating a hash from the query string and an appended timestamp. In our scenario, only queries that are either new or return an altered result set will be assigned a new PID.

D. Retrieving a Citable Subset

The query store contains the modified query `q`, a timestamp and the hash value `HR(q)` of the result set that `q` returned at that point of time. Whenever the subset that was delivered by `q` is referenced at a later point in time, the SQL query statement obtaining the previously stored data set denoted `q'` is executed. At the retrieval of the subset, query `q'` needs to point to the subset of the history table presenting only those values that have been valid during the time of the original query execution. By using the version information and the temporal metadata for every record, the appropriate subset delivering the appropriate data can easily be constructed. The exact query structure again depends on the implementation design of the temporal data in the tables.

IV. Use Case: Automated Report Generation for Critical Infrastructure

We present a use case for monitoring the performance and stability of large concrete dams which are used for hydroelectric power generation. More than 30 different sensor types gather data about the structures such as rainfall, water levels, temperature, humidity and many other factors. Complexity, amount, coverage and data collection frequency vary considerably, as the sources can be either collected manually or in an automated fashion. All sensor data is collected in a central database. A Web portal allows researchers, scientific and maintenance staff to generate reports of the data sources that aggregate the measurement data. This data is then used for generating monitoring reports including tables, graphs and all details required by the users to get informed on the status of the structure during the selected period.

For liability reasons, it is important to identify the exact subset that was used for the creation of the monitoring report. Only if the data that was used during report generation can be identified unambiguously and mapped to a specific report generation request, the automated generation of the monitoring reports is reproducible and understandable in the future.

Enabling data citation for current and future data can be achieved by copying the table structure of the original table and appending the versioning information (*version*, *last_event_type*, *last_event*) as new columns to the history table. The *last_event_type* refers to the three operations INSERT, UPDATE and DELETE. The column *last_event* stores the timestamp of the event and the *version* column stores an incremented version number for each event that alters data. Note that this more than doubles the storage requirements. Yet, given the size of the database and the effort that would be required to re-engineer the multitude of applications that would be affected by a change to the original DBMS, the full history solution (3) turns out to be the most cost-efficient.

A. Preparing the Database Schema

We need to adapt the way data gets inserted into the data collection in order to enable to trace inserts, updates and deletes into the history table. In our example we use three simple stored procedures: `sp_insert_value`, `sp_update_value` and `sp_delete_value`. All three stored procedures encapsulate the data insertion of the metadata in the history table by using transactions. This ensures that no changes can occur in between e.g. the actual deletion of a record in the original table and the metadata insertion into the history table. A more traditional approach to automatically react on updates would be the use of database triggers. We refrained from using triggers because they are highly vendor specific and provide a very different feature set that can not be used on other systems.

B. Query Re-Writing: Replanting Branches

For retrieving the results of the original query q it is sufficient to replace references to the original table with a generic query pattern addressing earlier versions of the data. The sample query from Listing 1 retrieves the average value aggregated by parameters for a given instrument.

Listing 1. Example Query

```
SELECT parameter, AVG(value)
FROM instrument
GROUP BY parameter
ORDER BY parameter ASC;
```

As the data in the database is still evolving, updates and deletes occur to the original data. For obtaining the original result set at a later point in time, the query q′ has to be rewritten in order to include only data as it was valid during the time of the execution of q.

Figure 1 depicts a minimal example of the query from Listing 1 that aggregates the average values of the instruments of a structure grouped by the parameter type. The figure also shows the modification of the operator tree and demonstrates the retrieval of the data for a simple query q. The query pattern q′ is generic and can be used for all history data tables that have been adapted to the new schema we proposed. The query q′ selects the latest version of a row that was valid at the time of the execution of q and thus reflects the state of the table at the earlier point in time. Thus the query pattern q′ can be inserted into the operator tree constructed by the original query and replace every occurrence of the original table.

The sorting order is crucial for maintaining the same sequence of records in the result set and thereby producing identical result set hashes (RH(q)) for queries. If query q includes an ORDER BY-clause it needs to be asserted that the primary keys of the involved tables are used as sorting criteria. If no ORDER BY-clause is present, the system needs to append this as part of the statement in order to maintain the sorting behavior. Several queries used for the report generation use a SELECT * - statement pattern and perform a selection on the result set. Hence the columns describing temporal metadata need to be excluded by the query.

The query presented here is a minimal example for demonstrating the feasibility of the approach. There are numerous optimizations possible that we will tackle in our future work. In this example, we did not encapsulate the query inside a view, as not all database systems allow to use other fundamental database system concepts such as indexes on views, rendering the process described here inefficient. The same is true for temporary tables as there are too many differences between the available RDBMS vendors.

C. Setting up the Query Store

We add a query table that contains the following columns:

- A PID

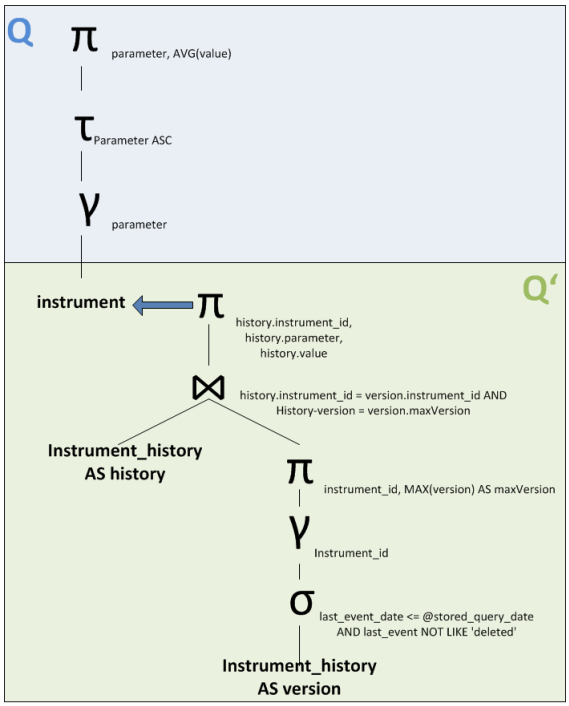

Figure 1. Replanting the Historical Branch

- A timestamp
- The original query statement q as entered by the user
- The modified original query to ensure unique sorting as processed by the system q_s. The results of this query have been returned to the user.
- The modified query as adapted by the system to enable retrieval on the timestamped and versioned tables, with adaptations performed to ensure unique sorting. This version will be used for citation and re-creation of the data set at a later point in time
- A hash key on the result set (to facilitatve verification of correctness)
- Additional metadata following recommendations, e.g. from DataCite[2]

The queries that result in the generation of the monitoring report can be logged automatically and inserted into the query store.

V. Conclusions and Outlook

The model for data citation we proposed in this work can be used to persistently cite arbitrary subsets in relational databases. We presented a series of steps that can be applied on relational databases for rendering them citable. We described how deterministic query results can be established by maintaining the sorting order of the result sets. We provided a basic database schema for versioning and introduced fundamental operations required to handle

[2]http://schema.datacite.org

evolving data. Being based upon temporal database aspects and unambiguous result presentation by adding stable sorting criteria, citing only the queries persistently is sufficient for our model. It guarantees not only consistent result sets across time, but also consistent result lists even in case of none or ambiguous result set sortings in the initial query, and also in the case of migration to a different DBMS. A mechanism that checks whether a query was already issued and that can detect if two subsequently executed queries return different result sets, ensures that only new queries will be assigned a new PID to avoid ambiguity.

In order to mitigate the burden of data citation on a productive system, different replication strategies need to be investigated. A further question that needs to be tackled is the evolution of the database schemata itself. In changing environments such as research, database tables themselves are subject to changes, which need to be detected and handled by the data citation system. Data base schema versioning is a topic that has been tackled for many years [13], but still more research is required for the proposed solutions scale with increasing amounts of data. Query normalization was also not treated in this paper, addressing specifically the adaptation of the stored procedures to ensure identical result sets upon re-execution on the new schema. As stored procedures and user defined functions are highly implementation specific, no generic approach exists. So far we tested the use case only with standard SELECT-statements that did not involve other stored procedures. Obviously stored procedures that alter data would require adaptation as well.

In principle, however, the proposed model should be applicable in all settings where data can be timestamped, and where some form of access and query interface is being used to identify the subsets to be made persistently available as individual units. Specifically, we will investigate how to apply the proposed model to other types of data repositories, specifically data formats used specifically in the scientific domain such as HDF5 [3], as well as different types of databases from the NoSQL movement (document, key-value, column-oriented or graph databases). Further use case pilots will be implemented as part of a working group to be established under the umbrella of the Research Data Alliance (RDA)[4], to better understand the impact and limitations of the various design decisions.

ACKNOWLEDGMENT

Part of this work was supported by the projects APARSEN and TIMBUS, partially funded by the EU under the FP7 contracts 269977 and 269940.

[3]http://www.hdfgroup.org/HDF5/

[4]https://rd-alliance.org/working-groups/data-citation-wg.html

REFERENCES

[1] R. Darby, S. Lambert, B. Matthews, M. Wilson, K. Gitmans, S. Dallmeier-Tiessen, S. Mele, and J. Suhonen, "Enabling scientific data sharing and re-use," in *E-Science (e-Science), 2012 IEEE 8th International Conference on*, 2012, pp. 1–8.

[2] K. Belhajjame, M. Roos, E. Garcia-Cuesta, G. Klyne, J. Zhao, D. De Roure, C. Goble, J. M. Gomez-Perez, K. Hettne, and A. Garrido, "Why workflows break - understanding and combating decay in taverna workflows," in *Proceedings of the 2012 IEEE 8th International Conference on E-Science (e-Science)*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/eScience.2012.6404482

[3] S. Lyons, "Persistent identification of electronic documents and the future of footnotes," *Law Libr. J.*, vol. 97, p. 681, 2005.

[4] J. K. Hans-Werner Hilse, *Implementing Persistent Identifiers: Overview of concepts, guidelines and recommendations*. Consortium of European Research Libraries, London, 2006. [Online]. Available: http://www.cerl.org/publications/report_on_persistent_identifiers

[5] Y. Tian and P. Rhodes, "Partial replica selection for spatial datasets," in *E-Science (e-Science), 2012 IEEE 8th International Conference on*, 2012, pp. 1–10.

[6] G. Slivinskas, C. S. Jensen, and R. Snodgrass, "Query plans for conventional and temporal queries involving duplicates and ordering," in *Proceedings of IEEE ICDE*, 1999, pp. 547–558.

[7] R. T. Snodgrass, "Temporal databases," *IEEE Computer*, vol. 19, pp. 35–42, 1986.

[8] "Optimal splitters for temporal and multi-version databases," in *SIGMOD Conference*, W. Le, F. Li, Y. Tao, and R. Christensen, Eds. ACM, June 22-27 2013.

[9] S. Pröll and A. Rauber, "Citable by Design - A Model for Making Data in Dynamic Environments Citable," in *2nd International Conference on Data Management Technologies and Applications (DATA)*, Reykjavik, Iceland, July 29-31 2013.

[10] C. S. Jensen and R. Snodgrass, "Temporal data management," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 11, no. 1, pp. 36–44, 1999.

[11] T. Johnston and R. Weis, *Managing Time in Relational Databases: How to Design, Update and Query Temporal Data*, 1st ed. Morgan Kaufmann, 7 2010.

[12] M. Guttenbrunner and A. Rauber, "A measurement framework for evaluating emulators for digital preservation," *ACM Transactions on Information Systems (TOIS)*, vol. 30, no. 2, 3 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2180876

[13] J. F. Roddick, "A model for schema versioning in temporal database systems," *Australian Computer Science Communications*, vol. 18, pp. 446–452, 1996.