SAND2016-10022C

# That Is One BIGINT: A Novel Relational Database Data Versioning Framework

Christopher Frazier
Sandia National Laboratories
Post Office Box 5800
Albuquerque, NM 87185-1188
Phone (505) 284-7922
email: crfrazi@sandia.gov

Asael Sorensen
Sandia National Laboratories
Post Office Box 5800
Albuquerque, NM 87185-1188
Phone (505) 844-5249
email: ahsoren@sandia.gov

***Abstract*—Tracking the revisions to a database table over time is useful for many applications, such as when prior analyses need to be vetted or replicated. A novel system is introduced to track the revisions in a database with minimal modifications to the existing structure. Specifically, every modification to a table is captured as a new row, with the use of a single integer field to capture the revision metadata. Using standard database queries, the state of the data at any time in the past can be extracted. The system also allows for parallel branches of data modifications to be tracked. Details of using the system, its consequences, and an evaluation of its performance are also presented.**

## I. Introduction

When storing data in database tables, the transactional changes to that data over time can often be of interest. Such revision tracking can be used not only to track how and when the data was modified or updated, but also to recreate the state of the data at a specific point in the past. This latter point can be particularly useful for recreating modelling results or examining previous analyses. Additionally, revision tracking the data can allow for hypothetical or "what if" scenarios to be developed within the same data store without losing access to the source data state.

In this paper, a framework for maintaining the transaction history of data stored in a relational database is presented. The motivations for this work were to provide a system which satisfied the following needs:

- A means of tracking the changes made to rows of data in specific tables, including additions, deletions, and updates. This requirement does not include schema changes. The revision tracking should include information about who made the change and when it happened.
- The ability to extract the state of the data in a tracked table at a specific time in the past.
- The ability to create revision data branches which isolate the transactional changes between branches. That is, changes made in a branch will only be visible within that branch, allowing multiple independent lines of data modification to be maintained in parallel.
- The avoidance of reliance on a specific database program's functionality for the implementation. In other words, the solution should be (roughly) portable across differing database vendor products.
- A "lightweight" solution was preferred. While this goal is not specific, the general intention was to avoid having the underlying database schema heavily influenced by the inclusion of revision tracking. Additionally, if special database requirements or controls were needed for the solution, then the end user should be provided with a simple or transparent interface for interacting with the database.
- A reasonable performance penalty for the inclusion of data revision tracking.

## II. Prior Work

Before developing the data revision tracking framework, a number of existing technologies were investigated to determine if a solution satisfying the above requirements already existed. Data change tracking has been implemented in a variety of contexts, both within the database program and on top of it. For example, Microsoft SQL Server has change tracking functionality which maintains a history of changes to a table, and allows a user to capture the state of the data at a given point in time [1]. However, this solution does not provide branching support, and requires the use of SQL Server.

An example of change tracking using only SQL is the revision tracking system built into the MediaWiki database which uses a separate revision table referenced by change tracked tables [2]. However the MediaWiki database does not incorporate branching and, being specifically geared towards MediaWiki applications, is not a generic solution but rather an implementation example. Another SQL-based change tracking example described in [3] uses a table to track every row entry change using triggers. While it is extremely lightweight, the method allows for neither recreation of data states nor branching.

An interesting implementation of data versioning involves the use of long transactions, which may span days or even years [4]. Each unique data state of interest is separated into

its own transaction, which can be extracted, modified, or replicated as long as it stays open. Branching functionality could be provided by such a formulation (where each branch is stored in its own long transaction), though revision tracking within a branch would still need to be implemented.[1]

Temporal databases are an extension of relational databases which have special support for tables holding data involving time. At a very high level, a temporal table includes columns which indicate over which time range a row of data is valid [5].[2] The support for transactional temporal information allows the extraction of historical data states in a generalized framework that is built into the database system. Although branching is not a basic concept in temporal databases, the native support of temporal information makes them interesting for investigation as a base upon which branching could be added.

While the concept of temporal databases has been known and researched for decades (*e.g.* [8]), only recently has it begun to find more widespread availability. The SQL:2011 standard codified the functionality and extensions for temporal tables [9]. However, not all database vendors have implemented this part of the standard, and those that have done so in only very recent versions of the software (Microsoft SQL Server 2016 [10]) or through third-party extensions (PostgreSQL [11]). Additionally, a temporal table may have limitations compared with standard tables [10], which may impact usability. There has also been some discussion in the literature of improving the performance of temporal database performance through the use of special indexing, which are not commonly available in most databases [12], [13]. Finally, temporal databases include support for functionality which is beyond the scope of what is required here (such as interacting with ranges of data) which indicates the possibility of a simpler solution.

Other research involves how to capture the changes to data that are made over time. A complex management system which runs on top of a standard database is presented in [14], but it is much more extensive than what is needed here as it includes features such as interacting time-ranges. Tracking data changes through comparisons of state is a related problem, but the applications are not well aligned with the current problem as the focus is on data consistency, merging changes across separate data sources, or capturing knowledge from the actual changes over time [15]–[18]. Some research has been done into how to extract previous data states, including discussions of the scope of persistance [19] and the tradeoffs between space and performance when storing temporal data [20], but this research is aimed at developing new data structures and algorithms within a database management system, rather than working with existing systems as-is.

A final issue which arises from the use of existing technologies for data revision tracking is how to manage the data state for simultaneous users. While some technologies (such as many temporal databases) can efficiently manage the data state at an individual query or transaction level, other solutions require the extraction or replication of the full valid data state needed for each connection [4], [13], which may cause performance issues if different data states need to be accessed often and/or simultaneously.

## III. Capturing Data State

In order to meet the previously enumerated criteria, a new technique was developed to track the revisions in the data state over time. This technique uses a single 64-bit integer column as a bitfield to hold all of the information necessary to capture the state of the data at any point in (transaction) time. Careful ordering of the information in the bitfield allows for efficient extraction of the data states using standard SQL.

The underlying structure of the revision tracking is as follows: Data branches, which are independently revision tracked data states, are uniquely identified by an integer, called the branch id. Data modifications are tracked to the second resolution, though there is flexibility to operate at courser or finer time granularities. An individual user is associated with a unique identification integer, called the user id. Both the user and the branch may have additional information about them stored in separate tables using the branch/user id as the primary key.

Every revision tracked table includes a single 64-bit integer (`BIGINT`) column to store the revision track information; throughout this paper, this column will be referred to as `RS` (an abbreviation of revision stamp). The generic form of the bitfield is shown in Figure 1. Because unsigned 64-bit integers are not available in some databases, a signed integer is assumed and the most significant bit (the sign bit) is kept at zero (a possible modification of this is discussed later).[3] After that, most significant bits hold the branch id, and the next most significant bits hold the time of the modification encoded as an integer. The remaining bits are used to hold the user id, a flag indicating a deletion, and any other information as needed.

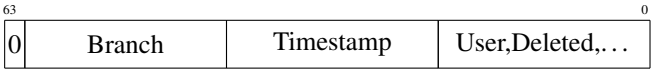



Figure 1. Generic bitfield layout for data revision tracking.

Whenever a table modification occurs, a new row is added to represent the updated data state. A new or modified row is just the row data with the appropriate `RS` value. A deleted row will be added as the most recent row data along with a `RS` value with the "deleted" bit set to 1.[4] Thus, in a revision tracked table, a `UPDATE` or `DELETE` action will never occur.[5]

[1] In theory, full revision tracking could be provided if separate transactions were created every modification. It is doubtful that such a system would be manageable, though.

[2] Temporal data may exist across both the data and transaction dimensions, respectively called valid (stated) time and transaction (logged) time, the latter of which is relevant to the problem addressed in this paper [5]–[7].

[3] Using the sign bit is possible and does not invalidate the methodology discussed here, but it is preferable to avoid it because it causes complications with the required binary and arithmetic operations used to extract a data state.

[4] It is not necessary for a deleted element to have its last data state replicated in the row marking its deletion. However, doing so preserves the state of the element when it was deleted. Furthermore, *some* data would need to be included in the row, and if the table includes columns that cannot hold `NULL` values, then what data to put in the row would need to be determined. Finally, before marking the element as deleted, a sanity check of its last state should be made to confirm that the element exists, which means that the state will be readily available.

[5] Another way to look at this is a Create-Read-Update-Delete (CRUD) application has been reframed as CR.

In doing this, all distinct transactional changes are retained in the table.

The process for encoding time as an integer is not restricted to any specific function or methodology, but the resulting timestamp must obey the following properties:

- If two timestamps, $T_A$ and $T_B$, represent different times with respect to the chosen temporal resolution, then $T_A \neq T_B$.
- If one timestamp, $T_A$, represents a time that occurred before another, $T_B$, then $T_A < T_B$.

Note there is not a requirement that timestamps representing the same time be equal, only that they satisfy the distinctness and ordering properties. As mentioned above, any temporal resolution and date-time span (the minimum and maximum recorded transaction time) can be used provided its range can fit in the (bit) space allocated to the timestamp value. For the examples discussed here, so-called "Unix time" is used, which is the number of seconds that have elapsed since midnight January 1, 1970 in Coordinated Universal Time (UTC). This timestamp provides one second resolution, and can be calculated easily in all major database systems.

Because the coordination of clocks across systems can be difficult, it is recommended that a single entity (system or service) be responsible for providing the timestamps, so that the transaction times are consistently recorded. Additionally, changes to the data spanning multiple cells, rows, or tables that need to be considered atomic (so that the data state is not invalid) must be registered with timestamps representing the same transaction time.

The actual bit layout for the application that was developed to use this framework is presented in Figure 2. With this layout, up to 16,383 branches and 1,023 unique users may be tracked. The timestamp fits into a standard 32-bit singed integer, and has a maximum transaction time occurring during January 19, 2038. The seven least significant bits are reserved for possible future needs, which may include expansion of the branch or user id space and encoding a version number of the bitfield.

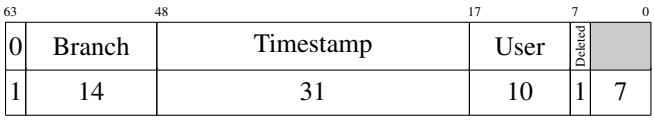



Figure 2. Actual bitfield layout used in prototype system. The second row gives the width of each field in bits.

Placing the branch and timestamp in the bitfield as described allows for relatively easy extraction of data states primarily through the use of ordering, grouping, and filtering operations. For a given table, ordering the rows by the `RS` field will result in all of the data (including changes) for a given branch to be contiguous. A result of this is that a maximum and minimum `RS` value for a given branch can be calculated, and all rows within that range will belong only to that branch. Thus, isolating a given branch requires only an ordering statement (on one column) and a `WHERE` clause with two predicates involving integer comparisons.

Within the data for a given branch, the rows can be further filtered so that they only include those entries which occurred before a given time. This is done by calculating the timestamp for this cutoff, and then calculating the `RS` value from it and the branch id (the remaining bits are set to zero). Selecting only those rows which are less than this value provides the data history of the table for that branch up to the time represented by the timestamp.

Within the time-restricted branch data, rows can be grouped together by the id column (or columns) of the table data. (Typically this unique id would be considered the primary key for the table, but, as discussed below, the inclusion of revision tracking no longer allows this.) Within a group with the same id, if the rows are sorted by the `RS` field in descending order, then they represent the full transaction history of that data element, with the more recent transactions listed higher in the result. The first row of the result table (for the grouped data) is the state of the data within the branch at the selected maximum transaction time.

Finally, any row in the data state whose `RS` field has the "deleted" flag set to 1 is filtered out (using bit-wise arithmetic). Thus, using standard query operations the data state at any transaction time can be extracted. Furthermore, because this occurs entirely within a query, multiple simultaneous requests for different data states - across both branch and transaction time dimensions - are inherently supported.

## IV. Implementation Considerations

### A. Keys

Maintaining the entire transactional history of a given data item in a single table requires the use of the `RS` field in any primary key for that table. That is, if a given item has a unique identifier $I$ (which may be composite), then for every modification to that data item a new row will be added to the table with the same identifier. So, by itself, $I$ cannot be used as a primary key; instead, a composite key of `(i,RS)` must be used to uniquely identify each row.

However, whenever a unique data state is extracted, each row in a given table will have a unique identifier (in $I$), so, in the context of an individual data state, it does form a valid key. Building an index on the identifier field (excluding `RS`) will provide the lookup benefits of a key to queries on a data state, as they will be inherited by subtables representing the data state. Because `RS` cutoffs are used to isolate data branches, having an explicit index on `RS` will also improve query lookups.

Unfortunately, building table relationships is complicated by the inclusion of transaction history. This is actually a problem that applies to temporal data in general, as the linkages between data rows may change depending on what time the data state is queried on [5]. For example, say that one row in a table ($R_A$) has a foreign key relationship to a row in another table ($R_B$) at a given time $t$, and that the data in $R_A$ is changed between $t$ and a later time $t'$. For the data state at time $t$, the foreign key relationship between the rows is valid, but at $t'$ it is not, as the row that was modified should replace the original. This issue is depicted in Figure 3.

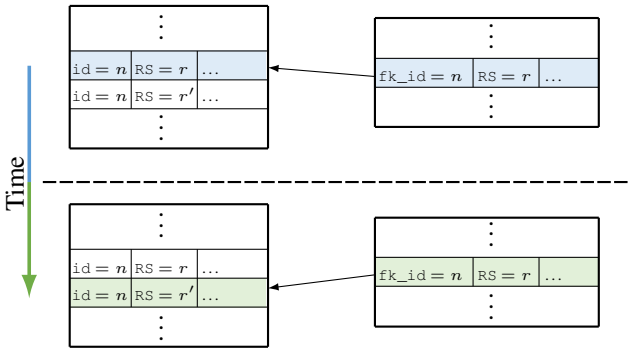



Figure 3. Illustration of foreign key relationship changing over transaction time. The dotted line separates the data state at two distinct (transaction) times, with the color fill indicating which rows are included in the data state at that time. The table on the left is a foreign key to the table on the right, and the row with `id` $= n$ has been modified during the time between the states. Although the foreign key is *relationally* the same at the two times, the actual rows forming the relationship in the database have changed.

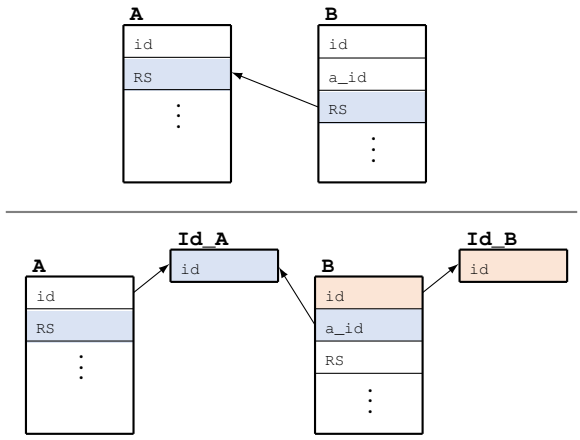



Figure 4. Illustration of the use of identifier tables in revision tracked tables; the colors represent columns with foreign key relationships. The upper part of the figure shows the actual foreign key relationship between tables `A` and `B` in a given data state. The lower part shows the relationship when using an identifier table (`Id_B`) which removes the transaction time dependency from the relationship. An identifier table has also been added for `B` to express the restriction that its identifier (`id`) be unique within a given data state.

While it is possible to forego referential database checks and drop foreign key relationships between revision tracked tables altogether, this is not ideal. Though it will decrease the impact on the database design, such a solution would discard any relational information from the database and would allow unchecked inclusion of invalid data into the database.

A better solution is to create a new table for each revision tracked table to hold only the foreign key identifier column(s). This new identifier table will have a foreign key relationship with the original table. All foreign key relationships with the original table will now point to the identifier table, thus expressing the basic structural relationship as well as providing some referential integrity (see Figure 4). This identifier table has the additional benefit of providing a straightforward (and standard) way to get new unique identifiers when new items are added to the revision tracked data.

Using the identifier table will provide only a subset of the referential integrity typically associated with foreign keys. Specifically, referential constraints will not work correctly with respect to the underlying data states, since the foreign key references are indirect. For example, referring to Figure 4, if the original relationship has a `CASCADE` constraint, then deleting a referenced row from `A` should delete the referencing rows from `B` with matching foreign key.[6] However, when using an identifier table the actual relationship between `A` and `B` at a given point in transaction time is not known to the underlying database engine, and thus the `CASCADE` constraint cannot be maintained. If the data using this revision tracking system requires such propagation or referential constraints, then custom triggers will need to be added to enforce them.

### B. Interacting With Revision Tracked Data

Querying revision tracked data requires putting together all of the steps outlined in Section III into a single query, which will return a table which captures the data state for the specified branch and transaction time. A prototype of such a query, using T-SQL syntax, is presented in Figure 5. If the list of columns (`<COLUMNS>`) includes everything except `RS`, then the returned table will represent the data as it existed at the specified transaction time as if there was no revision tracking. For a specific table, query can be pre-built with the table and column names, leaving the transaction time and branch as parameters to be specified. Though such a parameterized query cannot be used as a view, it can be implemented as a table-valued function or (less ideally) a stored procedure, for convenient access. The benefit of using a table-valued function (if supported by the database) is that the function can used directly in other queries.

```
SELECT
  <IDS>,<COLUMNS>
FROM (
  SELECT
    RANK() OVER (
      PARTITION BY SiteID ORDER BY <IDS>,RS DESC
    ) AS ROW_NUM,
    128 & RS AS DELETED,
    <IDS>,<COLUMNS>
  FROM <TABLE>
  WHERE
    RS >= <BRANCH>*562949953421312 AND
    RS <= <BRANCH>*562949953421312 +
            <TIMESTAMP>*262144 + 262143
) T
WHERE
  ROW_NUM=1 AND
  DELETED=0
```

Figure 5. Prototype (T-SQL) query for extracting the data state of table `<TABLE>` for branch `<BRANCH>` at transaction time `<TIMESTAMP>`. `<IDs>` is a comma-separated list of the unique id columns, and `<COLUMNS>` is a comma-separated list of the remaining columns. The hardcoded numbers are powers of 2 used as bitshifts and bitmasks and, for simplicity, required casts to `BIGINT` have been removed.

Because this revision tracking system requires that operations modifying rows are transformed into row additions, these operations must be controlled. One method for this is to create custom triggers for `UPDATE` and `DELETE` procedures for each revision tracked table. A trigger for `INSERT` procedures might be needed as well, to enforce correct unique id assignment.

[6] Delete here means inserting a new row with a `RS` value with the "deleted" flag set.

An alternative to triggers, which was used for the example implementation, is to build an API for interactions with the database, and to build the required logic into the system serving the API. While such a system will not provide the protection that triggers do for direct database interactions, it can be preferable if access to the database can be restricted to the API. Because such a system is not restricted to using a query language, it can be easier to program the required functionality and there is greater flexibility in exposing the system to a user. If querying functionality is also implemented, then this may be an alternative to table-valued functions or stored procedures for convenient access to the table data states.

Often, the data's state, rather than its transaction history, is important to an application and thus the information held in the `RS` field will not be directly needed. However, there can be situations where the capturing changes in the data over time may be important (such as data state differencing), and in these cases, the `RS` provides useful information. The fixed structure of the `RS` field allows the information in the field to be quickly extracted and presented in a useful format. For example, the prototype query (using T-SQL) in Figure 6 will return a table listing the transactional history (starting with the most recent change) of a single data element in a given branch, including information about who may the changes and when they happened.

```
SELECT
  <COLUMNS>,
  RS / 562949953421312 AS Branch,
  DATEADD(S,
          (RS & 562949953159168)/262144,
          '1970-01-01'
  ) AS TransactionTime,
  (RS & 261888) / 256 AS UserID,
  (RS & 128) / 128 AS Deleted
FROM <TABLE>
WHERE
  RS >= <BRANCH>*562949953421312 AND
  RS < (<BRANCH> + 1)*562949953421312 AND
  <ID_COLUMN> = <ID>
```

Figure 6. Prototype (T-SQL) query for capturing the transactional history of a data element with unique id `<ID>` in table `<TABLE>` for branch `<BRANCH>`. `<COLUMNS>` is a comma-separated list of the table's columns. The transaction time will be represented in UTC time. The hardcoded numbers are bitmasks and bitshifts and, for simplicity, required casts to `BIGINT` have been removed.

### C. Data Maintenance

Because this system of revision tracking will add a full row for every modification, the size of a database can grow much larger than the size of any one of its actual data states. For this reason, it might be useful or necessary to remove rows from revision tracked tables as part of an overall maintenance strategy. Such culling of records can have major implications for the transactional information represented in the database, so great care must be taken if performing such tasks. As is the case even with non-revision tracked databases, deleting records should only happen when it is certain that they will not be needed at any future time. Note that one of the motivations for revision tracking was to have the data state used for a particular analysis available for reference, and removal of data may make this impossible.

If an entire branch used for analysis is deleted, then obviously the data state cannot be extracted, but if only some records are removed, then a possibly worse problem of an incorrect data state may arise. Specifically, if a row that represented a particular element in a data state is removed and then the state recreated, the *previous* transaction involving the data element will be used for the data state. Not only is the data state incorrect, it may very well be implausible or inconsistent, and tracing such an error would very likely be difficult if not impossible. Furthermore, deleting an entire section of data based on the transaction times (*e.g.* delete all rows with a timestamp less than $t$) can also create problems, as the actual transaction which is valid - that is, most recent - for a given timestamp may have occurred well before that timestamp and would be incorrectly removed.

Because corrupting the proper data states is difficult to avoid, it is suggested that any deletions occur over an entire branch. If such a strategy is needed for branch management as opposed to reducing the database size, an alternative strategy may be useful: If, across all the data in a branch, the `RS` field's most significant bit is set to 1, then the `RS` values will become negative and will no longer be part of the "mainline" branch space. While this will provide no direct performance benefits, it may be convenient if queries across the branch space are ever performed (such as inter-branch summaries). Using this method can thus be viewed as an archiving operation, where the branch history is retained, but the direct availability of the branch is removed.

Another concern with tracking data revisions is what should happen when the database schema is modified. Specifically, if a revision tracked table's definition is modified, how should those modifications be reflected in the data history? When confronting schema changes with non-revision tracked data, the primary concern is how to apply the changes through the existing data in a manner that is consistent and meaningful. While this remains an issue with this revision tracking system, a more fundamental one also arises: If a table schema is modified, then the data state returned by the system for a transaction time *before* the schema modification will not accurately represent the data state at that transaction time. That is, the change to the schema will propagate backwards throughout the entire history of the table. Unfortunately, with this system there is no straightforward way to deal with this, outside of creating custom queries which (if possible) will revert the modifications for transaction times prior to the change. Though some modifications (such as adding a column) may be relatively harmless, it is emphasized that schema modifications can have much deeper implications when the data is revision tracked, and that care must be taken when applying them.

## V. Performance Evaluation

Utilizing this revision tracking framework will clearly have some performance implications. When inserting or updating data, a process (either external or via triggers) will need to occur to transform the query into an appropriate row insert reflecting the change. This process adds time to the operation, and will involve checking the most recent state of the modified data element (to ensure data integrity) as well as calculating a new value for the `RS` column. A more subtle issue is the impact on transactional consistency when the system is under load. The longer processing time will increase the chance that

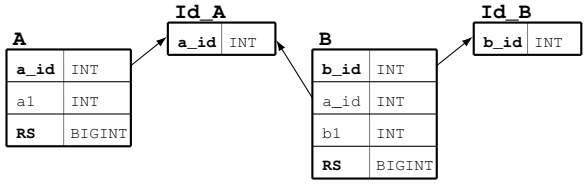



Figure 7. Schema diagram for database used to evaluate revision tracking read performance. Bold fields form the primary keys for the table, and arrows indicate foreign key relationships.

two data modifications will collide, especially if the temporal resolution of the tracking is on the order of (or slower than) the rate of the updates.

A larger concern is that the introduction of time into the system means that transactional consistency alone cannot be relied on to ensure temporal consistency in the data. That is, it is possible that two updates on the same element may create a revision history which is insensible. For example, two different modifications on the same element may be registered at the same time by different users, or, more pernicious, a delete operation may be saved *before* a modification on the same element. These issues can be mitigated through the use of a more complex system which performs failsafe checks before modifications are saved. Such a system will be difficult to correctly implement (and integrate) and will have further performance implications (*e.g.*, see [21]). It is noted that the consistency problems only arise for existing data elements, since new data element consistency (which involves ensuring unique identifiers) can be controlled through the use of the identifier tables discussed previously.

The original target application for the revision tracking system involved a limited number of users and infrequent updates to existing data (relative to the one second revision tracking resolution). The larger concern was the cost of the system to data reads, specifically the performance of the system when extracting the data state at a given transaction time. To evaluate this, a simple database was set up (see Figure 7) and two queries were executed and timed to quantify the performance of the system. One query just captured the entire data state of `A`, and the other (left) joined `A` to `B`. Both used the query prototype in Figure 5 with a specific branch and transaction time. To get a baseline performance for comparison, the state of the data in the `A` and `B` tables for the specific branch and transaction time was loaded into "fixed" versions of the tables (which did not include the `RS` column). Queries equivalent to those described were then run on these fixed tables.

The evaluation database tables were loaded with randomized data that varied in the following dimensions:

- The number of data elements (unique items) stored.
- The total number of data modifications. This is equivalent to the total number of rows in the tables.
- The total number of data branches.

Table I enumerates the various combinations of these dimensions against which the queries were evaluated. To

Table I. ENUMERATION OF TEST DATABASE DIMENSIONS AND TABLE STATISTICS.

| | | Undeleted Elements[a] | | Total Table Size[b] | |
|---|---|---|---|---|---|
| Elements | Branches | A | B | A | B |
| 100 | 10 | 100 | 100 | 100,000 | 91,213 |
| 100 | 10 | 100 | 100 | 1,000,000 | 954,303 |
| 100 | 100 | 100 | 100 | 100,000 | 96,505 |
| 100 | 100 | 100 | 100 | 1,000,000 | 954,653 |
| 100 | 100 | 97 | 100 | 10,000,000 | 4,709,283 |
| 100 | 1,000 | 80 | 81 | 100,000 | 99,676 |
| 100 | 1,000 | 100 | 100 | 1,000,000 | 932,966 |
| 100 | 1,000 | 100 | 100 | 10,000,000 | 5,265,964 |
| 1,000 | 1 | 992 | 995 | 100,000 | 99,581 |
| 1,000 | 10 | 990 | 989 | 1,000,000 | 950,819 |
| 1,000 | 100 | 999 | 1,000 | 1,000,000 | 953,870 |
| 1,000 | 1,000 | 779 | 786 | 1,000,000 | 953,668 |
| 10,000 | 1 | 9,950 | 9,956 | 1,000,000 | 996,748 |
| 10,000 | 10 | 9,988 | 9,990 | 1,000,000 | 995,280 |
| 10,000 | 100 | 9,992 | 9,995 | 10,000,000 | 9,508,104 |
| 100,000 | 1 | 99,494 | 99,501 | 10,000,000 | 9,947,960 |
| 100,000 | 10 | 99,907 | 99,917 | 10,000,000 | 9,948,994 |

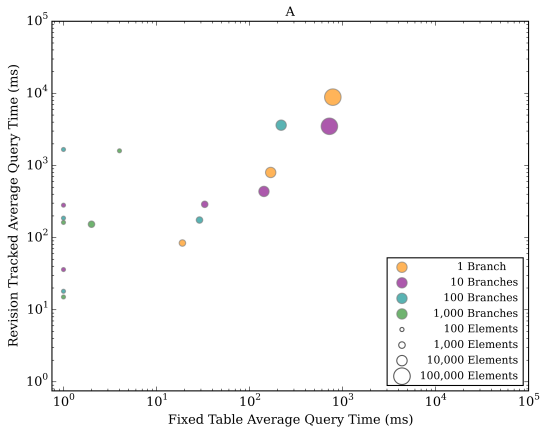
[a]These columns are the number of rows in the queries on `A` and `B JOIN A`, respectively.
[b]This is equivalent to the number of modifications to the data elements.

generate the data, a program was run which randomly added, modified, or deleted elements, until the desired sizes were achieved. Because of the randomization and the deletions, the final number of elements and table sizes did not always equal round numbers. Because query timings can vary from one execution to the next, the queries were repeated and the evaluation times averaged.

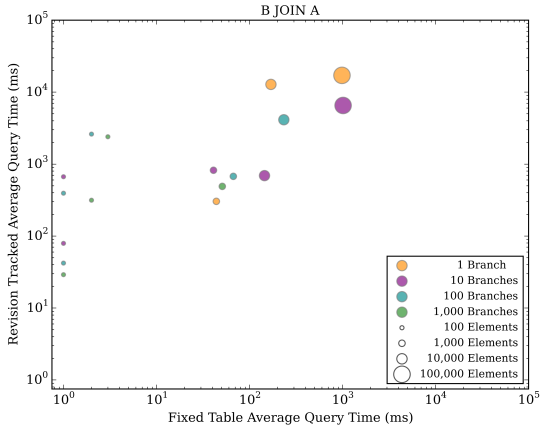
Scatter plots comparing the queries for revision tracked and fixed tables are shown in Figure 8. The number of branches does not appear to have any correlation with the running time of the queries, which makes sense given that the branch is used purely to restrict the range of `RS`, independent of the number of branches in the table.

The query times for the revision tracked tables tend to take roughly an order of magnitude longer than the fixed table queries, which indicates that scaling issues would have impacts for queries involving millions of data elements or higher. For queries involving fewer elements, the query times, while slower than with fixed tables, are reasonable given the fact that a data state is being dynamically extracted. Note, however, that for a specific number of data elements, there is much more variation in the query times for the revision tracked tables than for fixed tables. This is especially evident for the tables involving about 100 elements, where the fixed tables have nearly instant query times but the revision tracked tables shown execution times spanning multiple orders of magnitude.

The primary determiner of query performance is actually the number of data modifications in the table, or, more specifically, the number of rows in the subtable for a given branch. It is over this subtable in the prototype query (Figure 5) that the rows are partitioned and the most recent modification selected. The equivalent measurement for the fixed tables is the size of the entire (fixed) table, which is just the number of elements. Figure 9 plots the query times against the size of the branch or fixed table, and a similar relationship is seen with respect to the query times. That is, as the table size influences a standard query, so too does the entire branch size for revision tracked

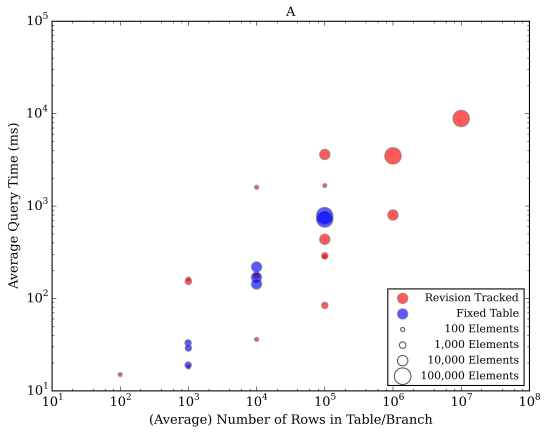



(a)

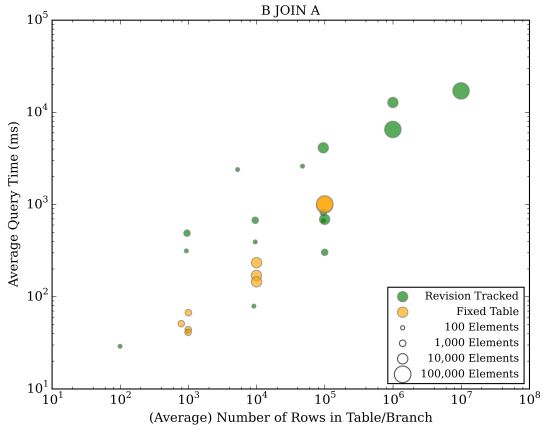



(b)

Figure 8. Comparison of the average query times for fixed *vs.* revision tracked tables.




(a)




(b)

Figure 9. Comparison of the table/branch size *vs.* the average query times for fixed *vs.* revision tracked tables. The $x$-axis for the fixed table is the number of rows in (a) `A` and (b) `B`. The $x$-axis for the revision tracked table is the number of rows in the branch queried; this value is roughly equal to the total number of rows in `A` or `B` divided by the number of branches.

tables. Note that the use of a join does not appear to incur much of an additional penalty above that shown by the size of the branch subtable.

## VI. Conclusion

The framework described in this paper provides a relatively simple way to add revision tracking to databases. The tracking is performed per table, so that it can be excluded from tables which do not require it. For each table that is tracked, the schema changes include one additional 64-bit integer column and an new unique identifier table comprising a single integer column. Additionally, indexes should be built on the unique identifiers and `RS` field to attain good performance and expected behaviors with the data state subtables. With this structure, the data state at any previous transaction time can be extracted using standard query operations. Modifying the data requires the fairly straightforward mapping from data transformations to new row inserts. Additionally, multiple branches of modifications can be maintained in parallel, and the revision tracked tables retain every modification which was made to the data, including what time it occurred and by which user.

Despite its relative simplicity, a number of potential issues must be taken into consideration when using the framework. First, to be user-friendly, a system must be developed to manage access to the database. This system will control data edits, as well as provide an interface through which the data states can be accessed. Such a system may involve database-based programming, such as triggers, stored procedures, and/or table-valued functions. However, developing an external application and controlling access through an API can provide more flexibility, although such a solution will be less "lightweight" than a database-based one.

Consistency, both in terms of the data states and the transactional history, can be difficult to maintain if the interactions with the database involve simultaneous edits of the same data elements. Additionally, the size of the data table will grow with the number of edits, which has a direct impact on the performance of read queries on the data. Because of this, the framework may not be suitable for applications with large numbers of data modifications, or if many users will be modifying the data simultaneously. If only a limited number of

previous data states are needed, then the performance impact of too many data modifications may be reduced by removing interim data edits (those between the transaction times of interest), though doing must be done with great care so as to not incorrectly capture the data state histories. Because reducing the number of saved edits has a direct impact on query times, developing robust strategies for such data cleaning would be a worthwhile area of future investigation.

Another area of potential research would be to look into alternative structures for the `RS` field. Specifically, the use of more columns to hold the information, while creating more complicated schema modifications, may provide performance benefits. Also, the issues with simultaneous edits may partly be addressed by splitting the `RS` field into two columns, with one including only the branch and edit time. This branch/time column would be used in the primary key for the table (along with the unique identifier), and thus would prevent two users from registering an edit at the same time.

Despite these concerns, for many applications this framework can provide a simple way to include revision tracking to a database. It is not tied to functionality specific to any database vendor or less common SQL standards. For situations where analyses or models are run on data which is updated over time, vetting or reproducing previous results is straightforward since the state of the data when the analysis/model run occurred is inherently retained. Additionally, different scenarios or hypotheses can be represented in the database through the use of branches, without the need to fully copy the database. Finally, the data modification history, which includes the user making the changes, helps create a "story" of the data which can provide context to the data states and their changes over times.

## REFERENCES


[1] Microsoft MSDN. "Track Data Changes (SQL Server)". Accessed November 5, 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb933994.aspx

[2] MediaWiki. "Manual:Database layout". Accessed November 5, 2015. [Online]. Available: https://www.mediawiki.org/wiki/Manual:Database_layout

[3] H.-J. Schoenig. "Tracking changes in PostgreSQL". Accessed November 5, 2015. [Online]. Available: http://www.cybertec.at/2013/12/tracking-changes-in-postgresql/

[4] R. Chatterjee, G. Arun, S. Agarwal, B. Speckhard, and R. Vasudevan, "Using applications of data versioning in database application development," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, May 2004, pp. 315–325.

[5] C. J. Date, H. Darwen, and N. Lorentzos, *Time and Relational Theory*, 2nd ed. Amsterdam: Elsevier (Morgan Kaufmann), 2014.

[6] C. S. Jensen and R. T. Snodgrass, "Semantics of time-varying information," *Inf. Syst.*, vol. 21, no. 4, pp. 311–352, Jun. 1996.

[7] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass, "On the semantics of “now” in databases," *ACM Trans. Database Syst.*, vol. 22, no. 2, pp. 171–214, Jun. 1997.

[8] P. Dadam, V. Y. Lum, and H.-D. Werner, "Integration of time versions into a relational database system," in *Proceedings of the 10th International Conference on Very Large Data Bases*, ser. VLDB '84. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1984, pp. 509–522.

[9] K. Kulkarni and J.-E. Michels, "Temporal features in sql:2011," *SIGMOD Rec.*, vol. 41, no. 3, pp. 34–43, Oct. 2012.

[10] Microsoft MSDN. "Temporal Tables". Accessed November 5, 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/dn935015.aspx

[11] arkhipov/temporal_tables (Github repository). Accessed November 5, 2015. [Online]. Available: https://github.com/arkhipov/temporal_tables

[12] D. Lomet and F. Nawab, "High performance temporal indexing on modern hardware," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, April 2015, pp. 1203–1214.

[13] M. Kvet, "Temporal data approach performance," in *Proceedings of International Conference CSSCC 2015*, March 2015, pp. 75–83.

[14] P. N. Hbler and N. Edelweiss, "Implementing a temporal database on top of aconventional database: Mapping of the data model and data definition management," in *Proc. 15th Brazilian Symposium on Databases (SBBD)*, 2000, pp. 259–272.

[15] S. Song and L. Chen, "Differential dependencies: Reasoning and discovery," *ACM Trans. Database Syst.*, vol. 36, no. 3, pp. 16:1–16:41, Aug. 2011.

[16] V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides, "High-level change detection in rdf(s) kbs," *ACM Trans. Database Syst.*, vol. 38, no. 1, pp. 1:1–1:42, Apr. 2013.

[17] G. Karvounarakis, T. J. Green, Z. G. Ives, and V. Tannen, "Collaborative data sharing via update exchange and provenance," *ACM Trans. Database Syst.*, vol. 38, no. 3, pp. 19:1–19:42, Sep. 2013.

[18] A. Shah, S. Ahsan, and A. Jaffer, "Temporal object-oriented system (tos) for modeling biological data," *Journal of American Science*, vol. 5, no. 3, pp. 63–73, 2009.

[19] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86–124, Feb. 1989.

[20] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan, "Archiving scientific data," *ACM Trans. Database Syst.*, vol. 29, no. 1, pp. 2–42, Mar. 2004.

[21] D. Lomet, A. Fekete, R. Wang, and P. Ward, "Multi-version concurrency via timestamp range conflict management," in *ICDE*. IEEE Computer Society, April 2012.