Baltic J. Modern Computing, Vol. 4 (2016), No. 4, pp. 978–993
http://dx.doi.org/10.22364/bjmc.2016.4.4.23

Data Version Control for Relational Databases: Small and Start-up Business Perspective

Girts STRAZDINS

Faculty of Engineering and Natural Sciences,
NTNU, Norwegian University of Science and Technology,
Trondheim, Norway

gist@ntnu.no

Abstract. Data-related businesses is an emerging trend in the recent decade. However, the availability and amount of information make it difficult to ensure quality in terms of data fusion and version control. Completely automated data aggregation systems fail to provide reliable and consistent data. In this paper we summarise existing knowledge on relation databases and augment it with description of business requirements for data version control. We propose an architecture that addresses the requirements, and discuss possible future work to improve and evaluate the approach.

Keywords: data version control; relational databases; architecture

1 Problem Statement

In the age of information we see more and more businesses in the field of information processing and analysis. Large amounts of data is gathered, aggregated and fused from many different sources. The challenges are changing: from unavailable to unreliable information. It is not a problem to get information any more. The question is - how accurate is the data?

On one hand, considering the vast amounts of information, it is clear, that wholly manual gathering is not possible. On the other hand, completely automated gathering by the machines is also challenging: different sources may report incomplete, inaccurate and sometimes contradictory information. The businesses face a challenge: how to control quality of large amounts of data gathered from diverse sources with various credibility?

Quality is a wide term with many possible interpretations. Therefore we clarify what quality means in the context of this paper:

1. **Consistency**. This is a fundamental principle for every database - all the data must always be in a consistent state. It may seem intuitive in the case where all the data is generated internally by the system itself (such as user- and administrator-created content). However, ensuring consistency is not trivial when the data is imported from several external systems, especially, if we consider external user-generated content (scraping public websites without a centralized application programming interface (API)).
2. **Credibility**. Even if the fused data is consistent, it may be wrong. E.g., one mistyped zero in an external data source. Our system should have a filtering and review phase, preferably with automatic constraint check that does not allow wrong values to be added into the system.
3. **Accountability**. For every value and statement in our data set we should be able to answer the question "Where does it come from?"
4. **Version history**. We should be able to see the whole history of data in case of multiple updates. We should also be able to return (*roll back*) to any of the earlier stable versions in case if a mistake is discovered in the current version.

If a business can not ensure any of these quality attributes the value of their data product is questionable. We argue that a data version control (DVC) is needed to ensure quality in the aspects mentioned above.

Relational databases using SQL language interface are the dominant form of data storage despite recent trend of No-SQL databases (Solid IT Gmbh, 2015). However, there is no formalized approach to data version control for relational databases. As a result, businesses are creating custom solutions and re-inventing the wheel. While it might be feasible for large corporations, the situation is different with small companies. Ability to reuse existing knowledge and solutions is especially important for start-ups where every drop of efficiency is critical.

Currently, information on existing DVC solutions is limited to programmer posts in software engineer forums, such as StackExchange. Although there might be solutions available in large database and data warehouse packages, such as the product lines from Oracle, it is not clear exactly what they include. Understanding the available packages can take days of browsing and phone calls in the best case, and require visiting specific courses and seminars in the worst case. Using powerful tools without proper understanding can be dangerous in terms of security and performance. The start-up developers are left in confusion: is it worth investing time in exploring complex business packages which will require extensive maintenance, education and significant budget? Or should we develop our own, custom solution from scratch? Both perspectives seem irrational considering the time and money constraints of small companies.

In this paper we propose a formalized approach to data version control. We combine existing knowledge on program source code version control systems and temporal databases to form a layer of version control (VC) for gathered data. Before publishing all information goes through approval system to ensure database consistency. The paper starts with a review of related work in Section 2 and our main contribution consists of three parts:

1. We describe system requirements from the business perspective in the form of *user stories*. Section 3 identifies necessary operations for each user role.

2. We describe the architecture and proposed technical implementation of the system (Section 4) that is compatible with any relational database supporting SQL. Each user requirement is matched with description on how it is satisfied.
3. We discuss the limitations, requirements and automation possibilities in Section 5.

2 Related Work

Although existing work does not solve directly our problem, significant amount of related work can be found that forms the basis for our research on this topic.

Authors of (Wang and Strong, 1996) Collected data quality attributes important for consumers using two surveys. In the first survey a list of 179 quality attributes were collected. Subjects of the first survey were 25 real data users working in the industry, and 112 MSc students studying in a field closely related to data analysis. In the second survey the collected attributes were prioritized. The subjects here were 1500 alumni (355 viable responses were received) of a large M.B.A. program in a U.S. University. As a result of the survey the authors compiled a 20 most important quality attribute list according to consumer opinion. Although the research is relatively old (from 1996), we believe that the core data quality values remain the same.

Our approach is aligned with the previous research. The four quality attributes discussed in our paper are directly related to five attributes in the list which form the top twelve:

1. ”Believability” represents the same meaning as ”Credibility” (1st place in the list according to (Wang and Strong, 1996)).
2. ”Accuracy” relates to both ”Credibility” and ”Consistency” in our terminology (4th place).
3. ”Timeliness” relates to both ”Version history” and ”Accountability” (9th place).
4. ”Traceability” relates to both ”Accountability” and ”Version history” (11th place).
5. ”Reputation” is covered by ”Accountability” (12th place).

The rest of the important data quality attributes relate more to the business operation (such as ”Value-added” and ”Interpretability”). Some aspects also relate to user interface (such as ”Representational consistency” and ”Concise”). These attributes are out of scope of this paper, as they do not relate to the technical implementation of the database.

We use the concept of *data object* as the basic unit of information in the database. An object represents a general unit according to the business logic. For simplicity we can assume that one row in a table is one object, although the same principles apply for complex objects spanning over several tables using joins. In a more general case we can say that an object is composed of several cells in one or more tables. If two tables have a one-to-many (1:M) relationship, then we say that the foreign key column belongs to the *parent* object on the ”one” side of the relationship. I.e., the 1:M relationship represents a composition and references to the sub-objects are also considered part of the parent object. In the case of many-to-many relationship (M:N) we can say that the relationship itself is considered a separate fact and is therefore a separate object. I.e., the rows in the junction table are separate data objects.

The concept of *temporal data*, i.e. data with validity time boundaries, has a long history (Snodgrass, 1990). In many business cases is important to specify validity period and hold previous versions of data. In 1990s Snodgrass introduced the concept of bi-temporal data (Snodgrass, 1992). Each object has two temporal dimensions: physical availability and logical validity. The bi-temporal validity is applicable also to our case (Snodgrass, 1999). We separate technical data state (and thus validity) in the VC layer from the logical state that depends on the business use of the data. In early 1990s Snodgrass proposed extensions to SQL language to include data temporality. Although it was not included as a general standard, several commercial products adopted this concept. Microsoft SQL server has a part called "Transact-SQL" (Microsoft Corporation, 2015). Among other things it includes "Change Data Capture" and "Change Tracking" modules, which track object update events for selected tables. Oracle has support for Temporal validity: specific SQL-constructs, including "AS OF PERIOD" (Verrier and Jeunot, 2015).

One seemingly similar problem addressed in previous work is version control of database schema (i.e., table structure). Roddick surveys issues of database schema versioning (Roddick, 1995). Tools such as RedGate SQL Source Control (Redgate, 2015) and gitSQL (gitSQL, 2015) can be used in this case. Rossini et al work on the same issue yet in another domain: version control for models in model driven engineering (Rossini et al., 2010). However, it is a different challenge. There is a significant difference between version control of data structure and actual data in terms of update frequency and data volume. This approach could be used to synchronize data for two instances of the same database: *draft* and *stable*. However, it would work only if there is no user-generated content in the stable instance, as it could be overwritten by the merging operation.

We use the *history table* concept from data warehouse techniques (Inmon, 1996). Kimball was the first one who introduced Transactional tables and the approach of keeping data history by storing transactions (events) - what happened to the data, not the data snapshots themselves (Kimball, 1996). We store object states in the history, in contrast to storing update events as it is used by systems supporting transactional tables. Audit tools such as Apex SQL Audit (Apex SQL LLC, 2015) and McAfee MySQL Audit Plugin (McAfee Inc, 2015) focus on recording executed transactions, including changes to both table structure and data. Oracle Audit allows to log operations performed with data, including executed queries (Jeloka et al., 2012). There are also tools for history table update automation, such as Hibernate ORM Envers (Red Hat, 2015) for the Java language and EntityAudit Doctrine Extension (Badura, 2015) for the PHP language. However, these extensions only create historical (i.e. replaced) version of the objects, without maintaining any object states. They could be used for VC layer, yet modifications are necessary.

In a non-scientific article Fuller describes a database design approach he calls *Point-in-Time Architecture* (PTA) (Fuller, 2007). Although he reuses many concepts from Snodgrass, Fuller describes several interesting points. He combines saving of historical records with data audit. Fuller argues that for databases we typically do not need the ability to roll-back to an older version, we just want to be able to see how it was. PTA requirements:

- Deletes are forbidden.
- Updates are forbidden for all data columns.
- Inserts act as both additions and replacements of rows. For each insert we want to know who and when inserted it.

A PTA system stores both: replaced old data and information on who and when updated it. We use this approach in our solution. Differences between the PTA and our proposed approach:

- PTA uses a single table for all the versions of the object. We store the object history in a separate table for improved performance and scalability. According terminology used by Johnston and Weiss, we use *history tables* as opposed to *version tables* used in PTA (Johnston and Weis, 2010).
- We split the two temporal dimensions. In PTA the business logic is not separated (fields DateEffective and DateEnd), whereas in our approach the technical state of objects is maintained at the VC layer and the business logic layer is allowed to maintain other validity constraints depending on application requirements. We do not restrict the business cases.
- In PTA every update triggers a new version of the object. In contrast, we release data updates in batches.
- The issue of cascading primary key (PK) updates is solved in our approach by introducing two dimensions of primary keys: technical PK, unique across all versions of all objects in a particular table, and a logical business-PK, uniquely identifying an object from the business perspective. The logical PK does not change over the lifetime of an object.

Johnston discusses the term *time-state tables* to keep *life history* of objects (Johnston, 2014). In addition, a concept of *event* is used. An event is a process that changes state of objects. Objects show up, change and disappear. The same object may re-appear again. Events, on the other hand, happen just once. Object is described by it's state. Transition between states occurs during an event. The sequence of objects states during it's lifetime is called *life history*. Atomic sequences of events are called transactions. To save the life history we can keep a list of either object's states, or transactions (including those creating new objects).

We use the concept of time-state tables to keep object life history in our approach:

- Past - historically replaced or discarded draft versions.
- Present - the actual data visible to the users.
- Future - draft data to be approved and released.

As the survey shows - we build on the existing knowledge, yet no previous work provides solution for the problem at hand - version control layer for data to ensure validation of unreliable data.

In the following section we describe more in detail the requirements from business and user perspective.

3 Business Requirements

In this section we describe the business requirements of the DVC layer. We identify the roles and necessary operations of different system users. We include a user role called *quality control manager (QC manager)*. Although the main focus in our paper is data version control, the role of this person is expected to go beyond the operations described in this paper. The *quality* in the title represents the person's responsibility to provide high-quality data in all the four aspects discussed above.

Although the requirements can differ significantly from system to system, here we try to summarize general requirements that can ensure the four necessary data quality attributes: Credibility, Consistency, Accountability and Version history. The original specification was created in a specific project, during discussions with business manager. Afterwards the requirements were made more generic to be applicable to a wide information system range. However, we acknowledge that a user study should be conducted to find out priorities seen by other manager, and evaluate whether the presented requirement set is representative.

Although the data version control described in this section involves manual approval, it will be a semi-automated approach, not completely manual. First, data is approved in bulks/batches. The QC managers will skim through the updates and are expected to notice very suspicious cases. The system should be able to give warnings and mark updates which are outside some given constraints. For example, if the expected value is in the range 1-100, and the newly fetched value is 1000, perhaps it is a typing error (one zero too much) by someone entering data in the external source system. As the system matures, it is expected that the suggesting system improves in accuracy and can handle more and more updates automatically, without human intervention. In general case it is hard to define automated quality check systems. That would be possible for some specific domains. One approach that could be taken: supervised machine learning, similar to what Google is doing for number recognition in their Street view module of Google Maps (Perez, 2012).

3.1 User Roles

A database with data version control requires the following roles:

- **Customers** use the system for their business needs.
- **Data editors** enter new data into the database, using semi-automated methods. Crawlers and robots using Application Programming Interfaces (APIs) represent a specific editor sub-group.
- **QC managers** review and approve or discard data change drafts.
- **Administrators** can edit data with more privileges than QC managers, including user administration.
- **Root administrators** have all the administrator rights and direct access to read, write and change any record and any structure in the DB.

Depending on the business logic, one user may play several roles. Yet it is important to identify all the roles and necessary operations for each of them.

The Root administrator role is outside the scope of QC layer. Here we identify it only for the sake of not forgetting that each database always has at least one such *super user*.

3.2 Stored Data Categories

All the logical data is separated into two categories: verified and non-verified. Decision on which data to verify depends on the business logic. For example, the managers can decide that all data gathered by automated crawlers and data imports must be validated, while all user-generated content can be show as-is. In general, it depends on who is responsible for validity of data.

In addition, the database will also store third kind of information: *meta data*. This includes historical versions, drafts and other information necessary to maintain the VC layer, but not part of the business data itself.

All the verified data requires:

1. Quality control before publishing.
2. Source of origin to ensure reliability.
3. Audit and historical versions (visible only internally, not for the customers).

While the first two features are necessary for the customers, the third one, data audit, is necessary internally to improve business processes. During system deployment we see questions such as ”Who updated this record? When did it happen? What was the reason?” quite often. Data audit helps to answer these and improve performance of the business.

3.3 General Version Control Requirements

At any point of time there is exactly one *latest version* for each object registered in the system. In addition, there might be previous versions stored. Each version has exactly one of the following object states associated, see Figure 1:

- Draft new data inserted by the editors.
- Approved data acknowledged by the QC managers but not yet published by the update system.
- Published data visible to the customers.
- Archived previously published data replaced by a more recent version. Not visible by customers any longer.
- Discarded data marked by QC managers as inappropriate. Has never been published.

We define the following general requirements for the DVC system:

1. All the published data has a *timestamp* of publish date. It is auto-generated by the data update sub-system, and can be manually set by administrators for each data version.
2. Upon approval the status of the according data changes from draft to approved.

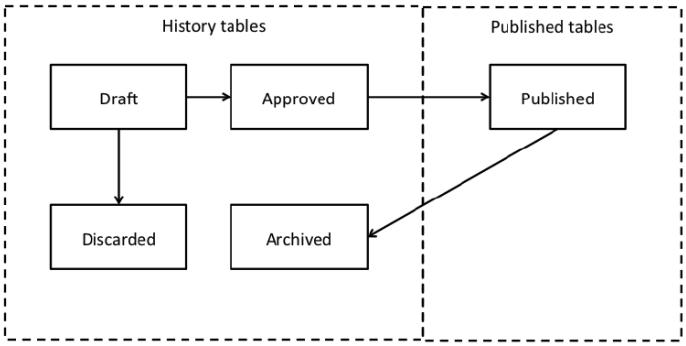

Fig. 1. Data state change diagram. New changes by editors come in as drafts. These get approved by QC managers and published by a periodic data update sub-system. Whenever an object is replaced by a new (published) version, the old one is marked as archived.

3. Upon arrival of a new data in the published state, the old record of the according data (if there is one) is automatically marked as archived.
4. The approved data gets published by a data update sub-system periodically.
5. Each update generates a new data version.
6. The publishing should happen in reasonable time periods (such as, daily updates). The period depends on business needs.
7. All the internal data (not published) should be stored separately. The amount of internal data should not slow down the published data access performance. However, the solutions should allow to store published and non-published data in the same table if the business requires it.
8. The primary keys (PKs) of all records should be kept intact during all updates to preserve consistency for the users. For example, if the user has saved a URL of the system web interface containing object PK, the URL should work also when the company has a new data version.
9. No data is ever deleted from the database. It is archived.

3.4 Data Source Requirements

All system users should be able to see the source for all verified data. Several fields in one object can have separate data sources. Therefore the users should be able to track origin of information for every data field separately.

The minimum information necessary for each data source:

- Name of the source
- Web URL
- Believed trustworthiness

The believed trustworthiness is important, as the business might decide to combine sources with small and very accurate data set with huge data sources with possible errors. It is important to inform the users how trustworthy the data is. And it allows

each user to specify minimum desired reliability. Note that we mention Web URL as a necessary attribute for each data source. However, the data sources are not limited to Web APIs only. Data can come from various databases, FTP server, and other types of communication. The Web URL is only meant as a unique descriptor of a web site which gives more information about the data source. It can be a web site of the data provider, or an URL to a PDF file containing documentation of the source.

3.5 Necessary Operations

Here we identify the operations and data access necessary for each user role.

3.5.1 Customer should be able to:

1. See all published data.
2. Modify the customer-generated data that belongs to them.

3.5.2 Data editors should be able to:

1. See limited part of published data that is necessary for their job.
2. Enter new updates in the form of draft data.
3. Delete (archive) data that they have created.
4. See their log: the drafts that they have created and according status: approved/discarded.

The limitation on data visibility depends on business needs. The information available to customers might have a high value and secrecy. Meanwhile, the data editors might be employed at external companies, not necessarily highly educated and fully loyal to the business. It is important to not expose data which might have significant monetary or legal consequences. Customers and data editors are user roles where access limitation is important.

3.5.3 QC managers should be able to:

1. Have all the data editor privileges.
2. See all current draft data.
3. See the source of draft data which user or API module has added it.
4. See version history for all data.
5. See the difference between the old (published) and new (draft) data in a convenient way.
6. Approve or discard any draft data object-by-object or field-by-field, as well as a convenient interface for batch approval.
7. Revert back to any of the previous data versions for selected objects.
8. Delete any object (archive it).

The QC managers are limited number of loyal employees of the business. Therefore they should be given access to all the information so that they see the big picture and can approve and discard data drafts appropriately.

3.5.4 Administrators should be able to:

1. Have all the data editor and QC manager privileges.
2. Control the period of data updates.
3. Manually force a data update.

3.6 Business case example: Decision Support Systems

This section describes potential use cases where data version control would be applicable. We mark the challenges that such systems face. The term *decision support systems* (DSS) describe a wide range of information systems with the goal to collect data from several sources and summarize it for the user in a way that makes it easy to make decision: compare, find the best (cheapest, fastest, etc). Example: consumer product price comparison web pages such as `https://www.salidzini.lv/` in Latvia and `https://www.prisguide.no/` in Norway (accessed 2016-07-04).

Such systems are processing data from several external sources and providing a summarized price comparison. Some data may be collected using official supplier API, some of it may be collected using web crawling techniques. The gathered data may contain errors due to several reasons:

- Wrong or outdated data on the supplier side.
- Format of data changed on the crawled website.
- Unpredicted values received (out of range, floating point instead of integer)
- Suspiciously contradicting values received from several suppliers for the same item. E.g. price EUR 100 and EUR 1000 for the same item. Possibly a typing error, or a dishonest marketing from one retailer.

Although in all these cases the source of the errors is in the external data supplier, the users will see the DSS as unreliable. These systems must be really accurate and reliable to be usable. Therefore, a verification system is required to filter out at least the extreme cases of external errors and publish only reliable data on the DSS website. The approach proposed in this paper can serve as a verification layer.

4 Implementation Proposal

In this section we describe proposed implementation architecture. It satisfied all the user requirements, and is generic - it can be implemented in any relational database management system.

For every type of verified objects, we store two tables: published data table and history table. The former stores information shown to the customers (published versions of data) while the latter stores all the non-published object versions, including drafts, approved, discarded and archived versions. The same applies for tables storing many-to-many (M:N) relations. If at least one of the tables in the many-to-many relation requires verification, the rows in the junction table storing the relation are also considered verifiable.

The implementation of DVC can happen in two ways. Either the whole system is designed with DVC from the beginning, or a transition is made from the original database to a versioned database. The steps in the case of transition:

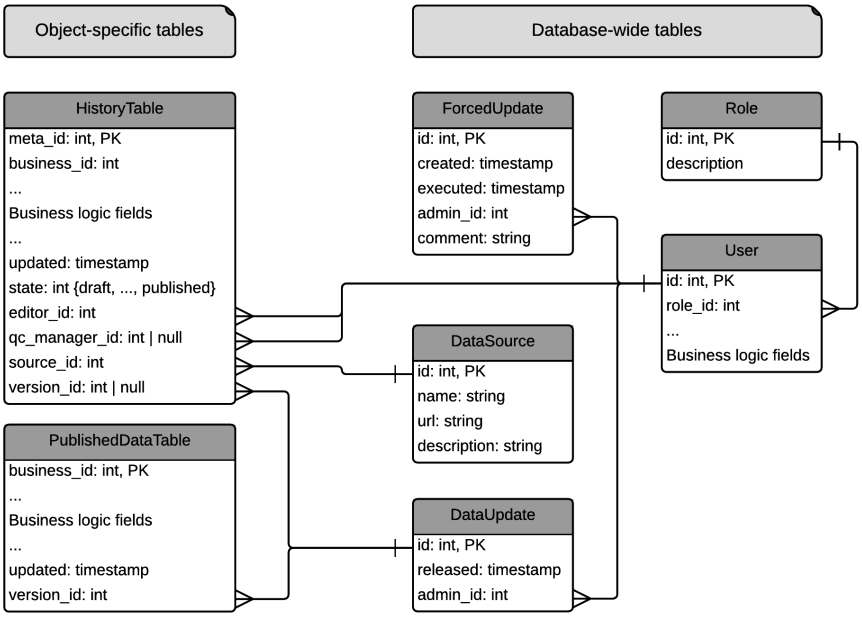

Fig. 2. Tables used for the QC infrastructure to store verified objects.

1. Decide which part of the data needs VC with verification.
2. For each verified (published) table do the following (details described below):
   (a) Create history table for each verified table.
   (b) Add the necessary meta-data fields.
3. Create database-wide meta tables.

In the rest of the article we refer to *initial* tables without any versioning as *business logic tables*, because these represent the real meaningful information to the users. We say that these tables consist of *business logic fields*.

Each object (row in a business logic table) is expected to have a unique identifier, called *business PK*, which would be there even if the proposed version control would not be used. It is considered good practice to include primary keys for all tables. They could be compound keys, consisting from multiple fields, but it should always be there. And the PK should not change over time.

In addition to business logic fields the published tables need the following meta-data fields, see Figure 2:

- updated: a timestamp of the last update for this object.
- references to DataUpdate table (can be seen as the version number) and DataSource.

The history tables include the following fields:

- All the fields from the corresponding business logic table.
- meta_id: primary key field, unique in the scope of history table. Each version of the same logical object will have the same business PK while having different meta_id.
- updated: a timestamp storing the moment of creation of this particular object version.
- state: the state of the object, one of the following: {draft, approved, discarded, published, archived}.
- references to DataUpdate table, DataSource and users who edited and approved this version.

The table names are different for every object, here the names HistoryTable and PublishedDataTable are used for demonstration purpose. In reality the database will have tables such as Sales, Sales_history, Order, Order_history, etc.

In the published data tables objects are identified by there business PK, in the history tables: by their meta_id.

Published data and history tables are per-object. I.e, each business logic table in the database requires one table for published data and one for history. The other tables are database-wide and contain meta-data related to version control:

- Role - stores all roles in the system, should hold only pre-defined objects.
- User - stores system users, including all roles listed above. The only requirement is to have ID and reference to roles. The other fields are business logic specific.
- DataUpdate - a milestone in data versioning, specifies a regular data release.
- ForcedUpdate - stores extra-ordinary data updates forced by administrators.
- DataSource - store information on the origin of data.

In the following subsection we describe how the implementation meets the listed user requirements.

4.1 Matching business requirements

Customer requirements are satisfied without any additional effort - they interact only with the published tables as they would without the VC layer. In all cases we assume that meta_id field has the *auto-increment* type and a unique meta_id is automatic for all entries in history tables.

4.1.1 Creating new updates in the form of draft data If it is an update (instead of an insert), find the according old record in published and history table - the one with the newest update timestamp. If no previous record found, create one. Update it with the new information and store it in the history table: updated = currentTime; editor_id = thisUser; source_id = userSpecifiedSource; state = draft; qc_manager_id = NULL.

4.1.2 Review all updates for editors Each editor can see all the data that has their ID in the editor_id field. The version history can be generated by ordering by the updated field.

4.1.3 QC Managers can see all current drafts QC managers can see all the drafts by selecting all the records r from the history tables with the following characteristics: state = draft, and no record r2 exists in history having r2.businessPK = r.businessPK and r.2state in {approved, discarded} and r2.updated $\geqslant$ r.updated.

4.1.4 Identify changes between the old published record and new draft update Any published data has also corresponding record in the history it was a draft and was approved before it was published. Therefore the changes can be selected by comparing records from the history tables: the last approved and the last draft records with the same ID.

4.1.5 Approve or discard a draft To approve a draft:

1. Create a copy of the last approved record with the same business PK
2. Update it with all the approved fields from the draft.
3. Set state=approved; qc_manager_id=thisUser; updated=thisTime.
4. Store the record in the history table.

We assume that the user interface allows to choose which fields to approve. For example, a table where each draft is represented by a row and each column represents a particular field. A checkbox in each column would allow to choose whether to approve the changes to the particular field.

To discard a particular draft, create a copy of the draft and store it in the history table with the following modifications: state=discarded; qc_manager_id=thisUser; updated=thisTime.

4.1.6 Revert back to an earlier version of an object Create a copy of the desired version and store it in the history table with the following modifications: state=approved; qc_manager_id=thisUser; updated=thisTime.

4.1.7 Delete an object - archive it

1. Create a copy of the last approved version with the same business PK.
2. Set state=archived; qc_manager_id=thisUser; updated=thisTime.
3. Store the new record in the history table.
4. Delete the published record (this step happens during batch data update cycle, see below).

4.1.8 Batch release of approved data updates and deleted records

1. Find time of the last batch data update, let us call it lastUpdate
2. Create and store a new DataUpdate record with released=thisTime (or a specific timestamp, specified by administrators in the ForcedUpdate table).
3. For each record rh in the history table with state=approved and updated $>$ lastUpdate:

   (a) Find the record rp in published table with the same businessPK (if there is one) or insert a new record rp in the published table.
   (b) Update rp with new data fields and set rp.version_id=thisUpdate.id.
4. For each record rh in the history table with state=archived and updated $>$ lastUpdate:
   (a) Delete all records rp in the published table with rp.businessPK = rh.businessPK.

It is important to execute all these statements as an atomic transaction. If one statement fails, the whole transaction is rolled back and error is reported. Otherwise, some data updates might get never executed resulting in an inconsistent database state.

4.1.9 Source information on the field level Reference to the source is included only in the history tables, not published data. For each published object there is at least one approved history version containing the most recent source. Different fields of the same object can be updated in separate versions. For example, an initial data version might contain company name and address from one source. The second version might have URL and phone number from a different source. The procedure how to find the source for all the fields of a particular object:

1. Select the published record as r1.
2. Mark all r1 fields as fromSource = unknown
3. Find the most recent record r2 in history table having r2.businessPK = r1.businessPK, r2.source $\neq$ r1.source, r2.updated $<$ r1.updated
4. For all the fields where r1.field $\neq$ r2.field and r1.field.fromSource = unknown set: r1.field.fromSource = r2.source
5. Assign the value of r2 to r1 (r1=r2) and repeat steps 3-4 until either all fields have fromSource $\neq$ unknown or no more sources can be found in the history.

5 Discussion and Conclusion

We discuss the requirements of data version control for data aggregation and analysis systems using SQL databases. We have proposed a generic solution that matches the requirements. However, it is only the first step towards high-quality databases. Future work is required to evaluate the performance and feasibility, and to improve the approach. Is it more efficient in terms of performance to have history table in a separate database? These answers should be found in empirical evaluations.

One visible drawback of our approach - the database size grows at least three times faster. Any record has to go through at least 3 phases: draft, approved, published. However, this information is essential for audit and reverting to older versions. We argue that there are three aspects mitigating the issue of storage space:

1. History logging can be defined based on business needs. As explained in the requirement section, the whole database can be separated into two parts: the tables (objects) requiring version control and the rest with only one version for every object. There can be cases with significant amounts of new data arriving fast. Special cases: user-generated content and sensor data in large networks. In both cases the

data (user posts and sensor values) is created once and does not change over time. The system could mark the corresponding objects as ”non-versioned”, and therefore avoiding the extra storage space needed.
2. Historical versions can be purged, if saving the storage space is more important than keeping history for some objects. E.g., future research could be done on systems that allow to define rules for automatic historical version deletion. In the proposed approach the whole history is stored as the latest version plus incremental changes growing towards the past. By recalculating the changes, all or specific historical versions can be deleted.
3. Cloud services are increasingly popular. Storage space is decreasing in price and the value of history could exceed the storage space price. We argue that proper version and quality control systems can increase the relative value of data. Therefore, it could be (although it is hard to measure) that the overall value-per-byte is increased even when the absolute storage space requirement is growing.

Another issue regarding the storage space - it is important to understand that the version control presented in this article is not meant as a backup system. It is purely to maintain the four necessary data quality attributes described in Section 1. Yet the proposed DVC system stores all the meta-data inside SQL. Therefore all the conventional SQL backup systems can be used as before.

Another point to be considered - modern databases have large numbers of tables. Maintaining published and history tables (i.e., synchronizing structure) is prone to errors. Ideally the VC layer should be invisible to the developers. They should use Object-relational mapping (ORM) approach to work with regular classes and objects. There are tools such as Doctrine and Hibernate supporting ORM, and both of them have extensions for data audit (Badura, 2015; Red Hat, 2015). Integration of the VC layer in ORM is one of the future directions. The VC layer provides wide range of opportunities for experimentation on how to improve published data quality.

Acknowledgment

The author would like to thank Sasan Mameghani and Johan Jøsok for valuable input and contribution to specification of business requirements for data version control.

References

Apex SQL LLC (2015). Apex SQL Audit. `http://www.apexsql.com/sql_tools_audit.aspx`. [Online; Accessed 2015-07-22].

Badura, D. (2015). EntityAudit Extension for Doctrine2. `https://github.com/simplethings/EntityAudit`. [Online; Accessed 2015-07-22].

Fuller, A. (2007). Database design: A point in time architecture. `https://www.simple-talk.com/sql/database-administration/database-design-a-point-in-time-architecture/`. [Online; Accessed 2015-07-22].

gitSQL (2015). Source control for SQL. http://www.gitsql.net/. [Online; Accessed 2015-07-22].

Inmon, W. H. (1996). The data warehouse and data mining. *Communications of the ACM*, 39(11):49–50.

Jeloka, S., Gosselin, D., and Smith, R. (2012). *Oracle Database Security Guide*. Oracle CorporationS, 10g release 2 (10.2) edition.

Johnston, T. (2014). *Bitemporal Data: Theory and Practice*. Newnes.

Johnston, T. and Weis, R. (2010). *Managing Time in Relational Databases*. Morgan Kaufmann.

Kimball, R. (1996). *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. Wiley.

McAfee Inc (2015). MySQL Audit Plugin. https://github.com/mcafee/mysql-audit. [Online; Accessed 2015-07-22].

Microsoft Corporation (2015). Tracking Data Changes. https://technet.microsoft.com/en-us/library/bb933994(v=sql.105).aspx. [Online; Accessed 2015-07-22].

Perez, S. (2012). Google Now Using ReCAPTCHA To Decode Street View Addresses. TechCrunch. https://techcrunch.com/2012/03/29/google-now-using-recaptcha-to-decode-street-view-addresses/. [Online; Accessed 2015-07-22].

Red Hat (2015). Hibernate ORM Envers. http://hibernate.org/orm/envers/. [Online; Accessed 2015-07-22].

Redgate (2015). SQL Source Control. http://www.red-gate.com/products/sql-development/sql-source-control/. [Online; Accessed 2015-07-22].

Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393.

Rossini, A., Rutle, A., Lamo, Y., and Wolter, U. (2010). A formalisation of the copy-modify-merge approach to version control in MDE. *The Journal of Logic and Algebraic Programming*, 79(7):636 – 658. The 20th Nordic Workshop on Programming Theory (NWPT 2008).

Snodgrass, R. (1990). Temporal databases status and research directions. *ACM SIGMOD Record*, 19(4):83–89.

Snodgrass, R. T. (1992). Temporal databases. In *Theories and methods of spatio-temporal reasoning in geographic space*, pages 22–64. Springer.

Snodgrass, R. T. (1999). *Developing Time-oriented Database Applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Solid IT Gmbh (2015). Db-engines ranking. http://db-engines.com/en/ranking. [Online; Accessed 2015-07-22].

Verrier, J.-F. and Jeunot, D. (2015). Oracle: Implementing Temporal Validity. http://www.oracle.com/webfolder/technetwork/tutorials/obe/db/12c/r1/ilm/temporal/temporal.html. [Online; Accessed 2015-07-22].

Wang, R. Y. and Strong, D. M. (1996). Beyond accuracy: What data quality means to data consumers. *Journal of management information systems*, 12(4):5–33.

Received December 11, 2015 , revised July 18, 2016, accepted July 28, 2016