Version control features for versioning-required documents within relational databases

Luka Hrgarek, Tatjana Welzer, Aida Kamišalić
Faculty of Electrical Engineering and Computer Science, University of Maribor
Maribor, Slovenia
{luka.hrgarek | tatjana.welzer | aida.kamisalic} @um.si

Abstract – **Version control systems, such as Git, successfully manage tracing changes within files. Advantages of the Git system are cryptographic authentication of history, branching capabilities, distributed development, etc. Even though those systems are sophisticated considering the file versioning they are limited regarding the support for other data structures. Often we are faced with the challenge of versioning relational data structures since the file structure is not always adequate storing structure. Database management systems do not offer explicit versioning of target data structures. Some aspects of version control might be used implicitly. In this paper, we will explore the possibilities of supporting version control features for versioning-required document files within with relational databases.**

I. Introduction

The majority of creative people start a project without having a back-up strategy [1]. Since digital data is fleeting and can be easily lost, the loss of important data without having a backup solution can be painful and in some cases financially burdening. Furthermore, for certain types of data, it is important for the user to have access to versions of data before certain changes. Version control, also known as source control or revision control is the process of managing multiple versions of a piece of information [2].

Many people perform version control by hand: they save every modification of a file under a new name that contains a number, each one higher than the number of the preceding version [2]. The use of a manual approach can be time consuming and is much less error-tolerant, as users can often mistakenly rename a file using a wrong name or delete it. Various tools have been created for the automation of this task aiming to reach more efficient version management.

Versioning tools mostly work within the file system, taking into account the hierarchical structure of directories and files. However, data or files that we work with and their versions we want to store are often not just independent units, but are linked in a relational data structure. Relational databases, which are traditionally used in the case of relational data structures, have limited support for data versioning. In this paper, we will consider the options of integrating version control systems with relational databases in order to take advantage of the best features of both.

The structure of the paper is as follows. In Subsection II-A, version control systems are presented. Different solutions for versioning within relational databases is detailed in Subsection II-B. The main contribution of the paper is presented in Section III with the proposed solution detailed. Section IV provides a final remarks and future research directions.

II. Related works

A. Version control

We can define a version control system as a system that tracks incremental versions (or revisions) of files and, in some cases, directories over time [3]. It must provide three important capabilities: *reversibility*, *concurrency*, and *annotation* [4]. Beside tracking incremental versions of files, many version control systems offer support for branching (see Figure 1). Branching can be defined as tracking parallel changes to the same codebase (file) [5].

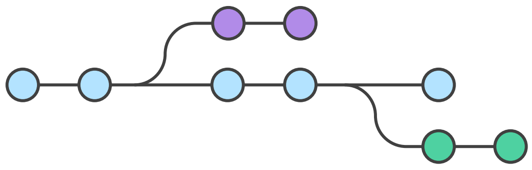

Fig. 1. Git branches [6]

We can divide the history of version control into three generations (see Table I). From the first generation, where concurrency was handled only with locks, version control has progressed to separation of merges and commits [7].

TABLE I
Three Generations of Version control [7]

Generation	Networking	Operations	Concurrency	Examples
First	None	One file at a time	Locks	RCS, SCCS
Second	Centralized	Multi-file	Merge before commit	CVS, SourceSafe, Subversion
Third	Distributed	Changesets	Commit before merge	Bazaar, Git, Mercurial

One of the oldest version control systems is Concurrent Versions System (CVS) [8], developed by Dick Grune in 1986 [1], [9]. Its primary focus was on solving concurrency problems in projects with multiple people working together on a set of files. It is based, as well as most of the later solutions, on the concept of abstract data types – repositories. CVS was developed in the form of UNIX shell scripts and uses RCS programs as version control primitives. Despite the fact that CVS is the oldest, many newer systems are compared to it [9].

Subversion (SVN), introduced around 2001, has quickly gained popularity. The most significant differences in comparison with CVS were atomically committed changes [1] and better support for branches.

One of the most widely used version control tool today is Git, invented by Linus Torvalds to support the development of the Linux Kernel [1].

B. Versioning in relational databases

There are several views on the versioning challenge within relational databases and approaches to address them. Most approaches are based on a specific method of modeling a database rather than technical changes to the database management system itself.

In data warehousing *Slowly Changing Dimensions* (SCD) are not changed on regular basis but change slowly and unpredictably, while some changes can cause referential integrity problems [10], [11]. Kimball *et al.* [12] define set of methods to manage SCD which are known as *Kimball Types*. If we take into account the similarity between the SCD problem and the data versioning problem, we can conclude that Kimball's Types could also be used for versioning in relational databases.

When using Kimball's Type 1 (*Overwrite the Value*) we merely overwrite the old attribute value in the dimension row, replacing it with the current value (see Table II). Despite relatively simple implementation, it is obvious that the major issue of using such a data storage method does not support preservation of previous versions of individual data.

TABLE II
SCD TYPE 1

ID	Department_name	FK_supervisor
14	Human Resources	255

ID	Department_name	FK_supervisor
14	Human Resources	367

Kimball's Type 2 (*Add a Dimension Row*) approach creates additional row for each change (see Table III). By using Type 2, all versions of data remain preserved and unchanged. Nevertheless, there is an issue in grouping all versions of a particular data, since each version has its own key, and the versions are not related to each other. In addition, the temporal dimension of each version of the data is missing.

TABLE III
SCD TYPE 2

ID	Department_name	FK_supervisor
14	Human Resources	255
18	Human Resources	367

Type 3 (*Add a Dimension Column*) adds an additional column to the table, which contains the previous value of the specified data (see Table IV). Similar to Type 2, all versions of data remain preserved, however, the temporal dimension is missing.

TABLE IV
SCD TYPE 3

ID	Department_ name	FK_ supervisor_ prior	FK_ supervisor
14	Human Resources	NULL	255
18	Human Resources	255	367

In addition to three basic types, Kimball *et al.* [12] define also Types 4, 5, 6 and 7. Type 4 adds an additional history table, where the original table keeps the current data, the history table contains all changes with the changes timestamped. The remaining types combine previous ones with addition of start and end dates as well as boolean flags which mark current and previous versions.

Change data capture (CDC) design patterns [13] use similar approach while adding timestamps or version numbers to table rows.

A related solution can also be found in the use of bitemporal databases [14] which distinguish between two dimensions of query tuples: *validity time* and *transaction time*. Bitemporal databases offer easier integration since such system already acts as one component, however, bitemporal data tends to increase in size as various versions of data accumulate over time.

Also, some DBMS offer the database snapshots functionality which captures the current state of the entire database and saves it to a separate location. This paper focuses on a specific case of versioning the document files that are simultaneously associated with the relational data structure.

III. PROPOSED SOLUTION

In larger systems, we often need to track versions only of some data, usually documents. At the same time, that data is part of a larger, usually relational, structure. Using version control system for entire data structure is not always possible since the majority of those systems work with the hierarchical structure of directories and files.

One option is to use Kimball Types, which can practically provide a solution to the versioning problem. The problem that may arise is mainly reflected in duplication of rows (when using Types > 1) and the active intervention in the database structure (when using Types > 2).

Duplication of rows is problematic mainly considering a querying process, since using Type 2 [12] do not provide a reliable attribute that would define information which row is current and which is "historical". If we assume that the primary key is a strictly auto-increment, we can claim that the current line is the one that has the highest value of the primary key (see Figure 2).

In tables containing rows from different categories despite a guaranteed auto-increment of the primary key, we can

```
SELECT FK\_supervisor
FROM Department
WHERE ID = (SELECT MAX(ID) FROM
    ↪ Department);
```

Fig. 2. SQL Query: trying to select newest row

not know which is the most recent value of an individual attribute in a particular category. We can only provide this if there is an unchangeable attribute (a [foreign] key) by which categories can be grouped. If we intend to group by an attribute that does not have the properties of a key, it may result in our query becoming irrelevant when the value of that attribute changes (in our example, *Department_name*).

Using Type 3 [12] adds a certain advantage in terms of identifying the newest row, compared to Type 2 [12], although this query becomes more complex to a certain extent. The easier task is to find out which row is the oldest – the one that has the *prior* attribute value set to `NULL`. Finding the newest row is a bit more complicated, since we have to find to which row none of the other rows are linked by using the *prior* attribute – in this problem, we can quickly see recursion, which is definitely wasteful from the optimizational point of view. The grouping problem remains unchanged, only the possibility of grouping into "chains" appears by using the current and *prior* keys.

Using history table in Type 4 [12] solves most of the aforementioned problems: the complexity of queries is slightly reduced and there are no more problems finding the current row. The new issue that arises is increased complexity of inserting new rows – next to normal data entry it is required to properly handle the previous data. This can be solved either programmatically, either by using a trigger in the database.

Higher types use start timestamps, end timestamps and *current* flags, which to some extent simplify queries since we can eliminate recursion. However, the problem of adapting the structure of the database remains, as we lose the "clean" database model.

Given that relational databases well resolve the problem of storing data with relational data structure, but they were not designed for versioning document data (for which the Kimball's Types can provide a sufficient rather than the ideal solution), and on the other hand, the version control systems are good solution for the problem of data versioning, but are not able to effectively store data with a relational data structure, for the specific problem of storing document data that are part of a relational data structure, we can propose the following solution.

Assume that we have a diversified data model that does not require versioning for most of its tables, but there is a table that has an attribute in which we store (structured or non-structured) document material. To store document material, we use a version control system, Git in our example, and one attribute for file referencing in our corresponding table within relational database.

The table in relational database (see example provided in Table V) retains its original structure, the only supplement thereto is a column that contains a reference to versioned file in version control system.

TABLE V
TABLE WITH REFERENCE TO GIT REPOSITORY

ID	**Department_ name**	**FK_ supervisor**	**Manifesto**
`14`	`Human Resources`	`255`	`m/hr.xml`
`19`	`Finances`	`345`	`m/f.xml`

In the version control system, we prepare a system of directories and files (which need to be versioned) that structurally corresponds as closely as possible to the tables in the relational database or it is logically organized in any other form allowing easy access (see Figure III).

```
m/
- hr.xml
- f.xml
- ...
d/
- ...
⋮
```

Fig. 3. File and folder structure in Git repository

Using the file path (we can make a HTTP request: `GET /projects/:id/repository/files/:file_path`) allows us access to all metadata about the file including size, encoding, reference to previous commit, etc. We can access the content itself by adding `/raw` to the file path.

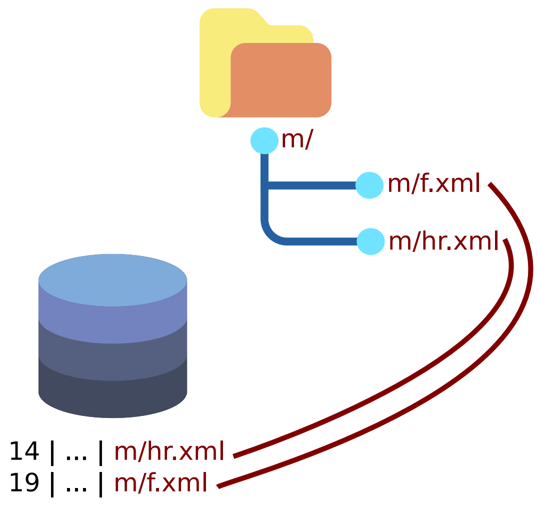

Fig. 4. Using file paths in database for referencing associated files in Git repository

The usage of this method of storing data is possible without any additional software, but can quickly lead to inconsistent maintenance of links between the relational database and Git repository. For this reason, it is advisable to develop and deploy special software that would take care of the synchronization between the two storage systems mentioned above.

IV. Conclusion

The problem of data versioning is not rare. Moreover, it is often present within file systems. Therefore, in most cases, developers are the most common users of version control systems. Nevertheless, developers are not the only users of version control systems. In use cases we can find examples, where it is necessary to control versions of data or documents which form a part of a larger, in most cases relational, data structure.

Versioning such data without the use of a relational database is nearly impossible, as storing relational data inside file structure would be very difficult to implement. It is difficult to avoid using relational databases and at the same time the versioning method should be implemented wisely. We have reviewed some of the most common methods of introducing data versioning to a relational database. All these methods, to a certain extent, solve the data versioning problem, while at the same time they contain some shortcomings, such as intervention with the original database structure, increased complexity of queries and in the case of large documents, load to the entire database.

For this reason, we proposed an approach in which we would maintain the use of a relational database, while at the same time the data or documents that we want to version would be excluded from the database and linked to it by a location reference. It is important to emphasize that the proposed solution is intended for a specific scenario for versioning the document files that are simultaneously associated with the relational data structure. In case of using the proposed approach, final query is divided into two parts: the SQL query in its original form and the file query that uses the file path from the SQL query result. Therefore, we distribute the responsibility between the database and the versioning system, since each one performs the work for which it was originally intended. Further research options include an overview of implementation possibilities of the aforementioned software solutions and study the effectiveness of the proposed approach. The aspect of scalability and performance can be crucial in many systems, therefore we can considered it as an independent subject of future research.

V. Acknowledgments

The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0057).

References

[1] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

[2] B. O'Sullivan, *Mercurial: The Definitive Guide: The Definitive Guide.* " O'Reilly Media, Inc.", 2009.

[3] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version Control with Subversion: Next Generation Open Source Version Control.* " O'Reilly Media, Inc.", 2008.

[4] E. Raymond, "Understanding version-control systems (draft)."

[5] S. Phillips, J. Sillito, and R. Walker, "Branching and merging: an investigation into current version control practices," in *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 9–15, ACM, 2011.

[6] Atlassian, "Git branch | atlassian git tutorial."

[7] E. Sink, *Version control by example*, vol. 20011. Pyrenean Gold Press Champaign, IL, 2011.

[8] D. Grune *et al.*, *Concurrent versions systems, a method for independent cooperation.* VU Amsterdam. Subfaculteit Wiskunde en Informatica, 1986.

[9] P. Louridas, "Version control," *IEEE Software*, vol. 23, no. 1, pp. 104–107, 2006.

[10] R. Kimball, "Slowly changing dimensions," *Information Management*, vol. 18, no. 9, p. 29, 2008.

[11] M. Wancerz and P. Wancerz, "History management of data: slowly changing dimensions," *Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska*, 2013.

[12] R. Kimball and M. Ross, *The data warehouse toolkit: The definitive guide to dimensional modeling.* John Wiley & Sons, 2013.

[13] W. D. Norcott, M. Brey, J. Galanes, P. Bingham, and R. Guzman, "Method and apparatus for change data capture in a database system," Feb. 14 2006. US Patent 6,999,977.

[14] A. Kumar, V. J. Tsotras, and C. Faloutsos, "Designing access methods for bitemporal databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 1, pp. 1–20, 1998.