Science of Computer Programming 97 (2015) 113–121

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Understanding database schema evolution: A case study

Anthony Cleve [a,*], Maxime Gobert [a], Loup Meurice [a], Jerome Maes [a], Jens Weber [b]

[a] University of Namur, Belgium
[b] University of Victoria, Canada

HIGHLIGHTS

- We present a tool-supported method to analyze the history of a database schema.
- The method makes use of mining software repositories (MSR) techniques.
- We report on the application of the method to a large-scale case study.

ARTICLE INFO

Article history:
Received 9 October 2013
Accepted 5 November 2013
Available online 22 November 2013

Keywords:
Database understanding
Schema evolution
Software repository mining

ABSTRACT

Database reverse engineering (DRE) has traditionally been carried out by considering three main information sources: (1) the database schema, (2) the stored data, and (3) the application programs. Not all of these information sources are always available, or of sufficient quality to inform the DRE process. For example, getting access to real-world data is often extremely problematic for information systems that maintain private data. In recent years, the analysis of the evolution history of software programs have gained an increasing role in reverse engineering in general, but comparatively little such research has been carried out in the context of database reverse engineering. The goal of this paper is to contribute to narrowing this gap and exploring the use of the database evolution history as an additional information source to aid database schema reverse engineering. We present a tool-supported method for analyzing the evolution history of legacy databases, and we report on a large-scale case study of reverse engineering a complex information system and curate it as a benchmark for future research efforts within the community.

Working in Paul Klint's group at CWI has been my very first professional experience. This was certainly the best possible way to start my research career. Paul is one of the most inspiring persons I have ever met. His ability to share his enthusiasm with his colleagues and students, and to show them the way through his own achievements, is truly outstanding. Now that I have, in turn, the privilege to supervise students, I make sure to regularly ask them the question Paul used to ask me: "So, are you still making progress?" Their most recent answer is summarized in this paper. Happy Birthday, Paul! – Anthony.

1. Introduction

Understanding the evolution history of a complex software system can significantly aid and inform current and future development initiatives of that system. Software repositories such as version management systems and issue trackers provide excellent opportunities for historical analyses of system evolution. Most research work in this area has concentrated on program code, design and architecture. Fewer studies have focused on database systems and schemas. This is an unfortunate gap as databases are often at the heart of many of today's information systems. Understanding the database schema –

* Corresponding author.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.025

which captures domain-specific concepts, data structures and integrity constraints – often constitutes a prerequisite to understanding the evolution of such systems.

In this paper, we report on our experiences made in the context of a real-world project with the objective of evolving a complex medical information system to fit new requirements. Specifically, (1) we present the tool-supported approach we developed to better understand the evolution history of the system's database, (2) we identify research challenges in the context of studying the evolution of data-intensive systems, and (3) we curate a rich and complex case study that can be used to explore these challenges (and others) by the software evolution research community.

The remainder of this paper is structured as follows. The next section introduces the main subject system studied in this paper (OSCAR) and the general context of our software evolution project. Section 3 describes the approach we have followed to study the evolution of the OSCAR database. In Section 4, we briefly present the tool suite that supports our approach. The results obtained when analyzing the OSCAR's history are summarized in Section 5 and discussed in Section 6. A related work discussion is given in Section 7 and Section 8 provides concluding remarks.

2. Context: The OSCAR system

OSCAR (*Open Source Clinical Application Resource*) is full-featured Electronic Medical Record (EMR) software system for primary care clinics. It has been under development since 2001 and is widely used in hundreds of clinics across Canada. As an open source project, OSCAR has a broad and active community of users and developers. The Department of Family Practice at McMaster University, which has managed OSCAR development efforts from inception to 2012, has recently transferred oversight of ongoing development to a newly formed not-for-profit company called OSCAR-EMR. This move was motivated by a new regulatory requirement to undergo ISO certification (*ISO 13485 Medical devices – Quality management systems*).

OSCAR architecture. OSCAR has a Web application architecture following the classical 3-tier paradigm. It employs a Java-based technology stack, making use of Java Server Pages (JSP), Enterprise Java Beans (J2EE) and several frameworks such as Spring, Struts and Hibernate. The source code comprises approximately two million lines of code with a rough distribution of 600 kLOC for the application logic, 1200 kLOC for the presentation layer and 100 kLOC for the persistence layer. OSCAR uses MySQL as the relational database engine and a combination of different ways to access it, including Hibernate object-relational middleware, Java Persistence Architecture (JPA) and dynamic SQL (via JDBC). The reason for this combination of technologies is the constant and ongoing evolution history of the product, which originated from JDBC, via Hibernate to JPA.

Oscar database. The OSCAR database schema has over 440 tables and many thousands of attributes. At the time of conducting our study, the database schema of the OSCAR distribution did not contain any information on relationships between tables (foreign keys) and no documentation was available about the schema. We later learned that the missing relationships were due to the evolution history of OSCAR, which has been using the older MyISAM database engine provided by MySQL that does not support foreign keys. A port to the newer InnoDB engine is underway, which will eventually allow foreign keys to be defined explicitly.

OSCAR software repositories. The OSCAR community utilizes a range of software repositories and tools, including a feature request and bug tracking system (provided by Sourceforge), a source code submission and review system (Gerrit Code Review), a git-based configuration management system, a community Wiki (based on Plone) and three active mailing lists (one for developers and two for users of different levels of technical expertise).

The need to understand the database schema. The OSCAR database has grown organically over many years and knowledge about its internal structure is distributed among pockets of developers who have been contributing to specific functions of the system (e.g., prescription writer, representation of lab results etc.). Our need to understand the OSCAR database schema originated from our involvement in a project with the goal to develop software for a primary care research network (PCRN). The purpose of the PCRN is to integrate health information kept in primary care EMR software in order to make them accessible to medical research and data mining. An important step in developing the PCRN software is to create "*export conduits*" for transferring health data from the EMR into a research database for subsequent query processing. Due to its popularity (second largest market share in British Columbia) and openness, OSCAR has been chosen as one of the first EMR products to interface with the emerging PCRN.

While designing early versions of the PCRN data migration adapter for OSCAR, we found that we were running into questions pertaining to the database schema. Of course, as could be expected for any heavily evolved, real-world system, some of them had to do with the fact that the database schema lacked documentation. Moreover, the schema did not contain any declared relationships (foreign keys). Other questions were of a more semantic and puzzling nature. For example, when attempting to design the function to export data on patient immunization records, we found two seemingly unrelated schema structures covering the same semantic issue. One schema structure revolved around tables entitled "*immunizations*" and "*configimmunization*" while the other schema structure revolved around tables entitled "*preventions*" and "*preventionsext*". During our project we found that taking into consideration the evolution history of the database schema was helpful in answering questions like these. (We found out that the "prevention" structured superseded the "immunization" structure but has still been retained in order to deal with legacy data.) This motivated us to investigate more formally OSCAR's evolution history and develop methods and tools to help with this investigation.

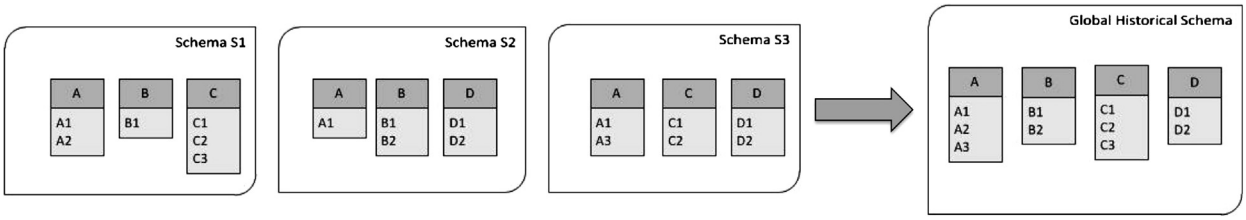

Fig. 1. Schema evolution example (left) and corresponding historical schema (right).

3. Approach

Because of the above observations and questions, we found that recovering a precise knowledge of the evolution history of the OSCAR database schema was an important prerequisite for gaining an understanding of the OSCAR database. Such historical knowledge could indeed help us to identify the most *stable* tables of the OSCAR schema – that we could start with when migrating data from the transactional OSCAR database to the non-relational research database that was selected as platform for the PCRN software system. It could also allow the discovery of potentially *dead* schema fragments – that are not used anymore but are kept for backward compatibility and medico-legal reasons.

The main question then becomes: *How to extract, represent and exploit the history of a database schema*? While we address this question in the context of a specific information system (OSCAR), the approach we present in this paper is fully generic, and therefore easily reusable for analyzing other data-intensive systems. It consists of extracting and comparing the successive versions of the database schema from the versioning system, in order to produce the so-called *global historical schema*. The latter is a visual and browsable representation of the database schema evolution over time. It contains all database schema objects (i.e., tables, columns and constraints) that have existed in the history of the system. Those schema objects are annotated with meta-information about their lifetime, which in turn serve as a basis for the visualization of the schema and its further analysis. This historical schema can be queried in order to derive valuable information about the evolution of the database, potentially raising other interesting system-specific questions to investigate.

The global process that we follow to build the historical database schema of a system consists of several steps:

1. *SQL code extraction*: We first extract all the SQL files corresponding to each system version, by exploiting the versioning system.[1]
2. *Schema extraction*: We extract the logical schema corresponding to each SQL file obtained so far, by means of a dedicated SQL parser.
3. *Schema comparison*: We compare the successive logical schemas while incrementally building the resulting historical schema.
4. *Visualization & exploitation*: The historical schema can then be visualized and further analyzed, depending of the project-specific needs.

Let us now further specify the *Schema comparison* step. The left-hand side of Fig. 1 gives an example evolution of a database schema, involving three successive schema versions. Schema S_1 is the oldest one and schema S_3 is the most recent one. We can see that between version 1 and version 2 column A_2 has been deleted, column B_2 has been created as well as table D and its columns. Moreover the entire table C has been dropped. In version 3, table B has disappeared, table D has been left unchanged, and table C has *re-appeared*. Indeed, it used to exist in version 1, it had been removed in version 2 and it is now back in version 3. We will refer to that phenomenon by saying that a schema object may have *several lives*.

The historical schema derived from the above schema evolution example is depicted at the right-hand side of Fig. 1. The historical schema is a global representation of all previous versions of a database schema, since it contains *all* objects that have ever existed in the entire schema history.

The historical schema is annotated with a list of couples $(date(S_i), committer(S_i))$ that provides, for each successive schema version S_i, the commit date ($date(S_i)$) and the id of the committer ($committer(S_i)$). Each object of the historical schema (table or column) is annotated with the following meta-attributes:

- *isDead*: true if the object is not present in the latest (current) version of the schema, false otherwise.
- *creationDate*: the date the oldest schema version where the object appears, i.e., the date of creation of the object.
- *lastAppearanceDate*: The date of the most recent schema where the object appears.
- *listOfPresence*: the list of schema version dates where the object is present.
- *listOfDeletion*: the list of schema version dates where the object has been deleted.

[1] A Git repository in the case of OSCAR.

The historical schema derivation algorithm is based on a pairwise comparison of all those schema versions. The algorithm starts from an empty historical schema, and then iterates on all the schemas in chronological order, while comparing the current schema S_i with the current historical schema S_H. The comparison is made by iterating on each schema object (table or column) of both schemas. Several situations may occur for a given schema object:

1. o belongs to S_i but does not belong to S_H. This means that o has been created in version i. We therefore add it to S_H and sets its date of creation to $date(S_i)$.
2. o belongs to S_i and (now) belongs to S_H. In this case, we update its *listOfPresence* and *lastAppearanceDate* and we set its *isDead* attribute to *false*.
3. o belongs to S_H but does not belong to S_i. If its meta-attribute *isDead* is *false*, o was present in S_{i-1} but was deleted in S_i. We then update the *listOfDeletion* attribute and set its *isDead* attribute to *true*.

4. Tool support

We have developed three main tools for supporting our approach: a schema extractor, a historical schema derivator and a history visualizer. Those tools are implemented as Java plugins of DB-MAIN [1]. DB-MAIN is a generic database engineering tool with integrated support for database design, reverse engineering, re-engineering, integration, maintenance and evolution.

The schema extractor allows to extract all the successive database schema versions from the project's repository (SVN and GitHub are currently supported). The output consists of a set of DB-MAIN schemas, each annotated with the commit date and a committer id.

The historical schema derivator takes as input the set of extracted schemas and produces the corresponding global historical schema. Our initial (naive) implementation of the algorithm was not sufficiently scalable to analyze long histories of large database schemas in satisfying time. We therefore implemented a multi-threaded version of the algorithm. In this version, an independent thread is used to analyze each distinct table of the schema, and is responsible for iterating through all the schema versions in order to derive the historical information of that table. All parallel threads share a common resource, namely the historical schema, that they can all update when they discover information about their respective tables.

The main program thread iterates over all tables of all successive schema versions. Each time the main thread encounters a table t that does not correspond to a running or terminated thread, it starts a new thread for t. As we will see below, this multi-thread implementation allowed us to significantly improve the efficiency of our historical schema derivation tool.

The current version of the tool is able to identify 16 distinct types of database schema changes: adding/dropping a table; adding/dropping a column; adding/dropping a primary identifier; adding/dropping a foreign key; adding/dropping an index; adding/dropping/updating a default column value; changing the type of a column; making a mandatory column optional (i.e., *nullable*) and conversely.

The visualizer provides the user with a visual and browsable representation of the database schema evolution over time. It takes the historical schema as input and allows, among others, to (1) compare two arbitrary schema versions, (2) extract the database schema at a given date, (3) extract the complete history of a particular schema object (column/table), (4) extract various statistics about the evolution of the database schema, (5) analyze the involvement of each developer in that evolution.

5. Results

We analyzed the history of the OSCAR database schema during a period of almost ten years (22/07/2003–27/06/2013). During this period, a total of 670[2] different schema versions can be found in the project's GitHub repository. The earliest schema version analyzed (22/07/2003) includes 88 tables, while the latest schema version considered (27/06/2013) comprises 445 tables. When applied to this dataset, our single-thread schema comparison implementation takes more than three hours to derive the historical schema, while the multi-threaded version completes the process in 40 minutes.

Once the historical schema was derived, we applied a procedure that would colorize each historical schema object, depending on its age and its liveness. Fig. 2 (left) shows a colorized version of the historical schema of OSCAR, that has been automatically derived by our tool, and that can be browsed and queried using DB-MAIN. All schema objects depicted in green constitute the tables and columns that are still present in the latest schema version of OSCAR. All red schema objects have been deleted. The color shade corresponds to the age of the objects. A dark red schema object is a table or a column that has been deleted a long time ago. A light red object is an object has recently been removed from the schema. An object depicted in green corresponds to a column or a table that is still present in the latest schema version. The darker the green, the older the corresponding table or column is, and vice versa. A schema object colored in orange is a deleted object that had several lives. Fig. 2 (right) zooms on a particular table of the global historical schema of OSCAR, namely the *integratorconsent* table. The table itself is green, which means that it belongs to the latest schema version of OSCAR.

[2] We consider one schema version per commit.

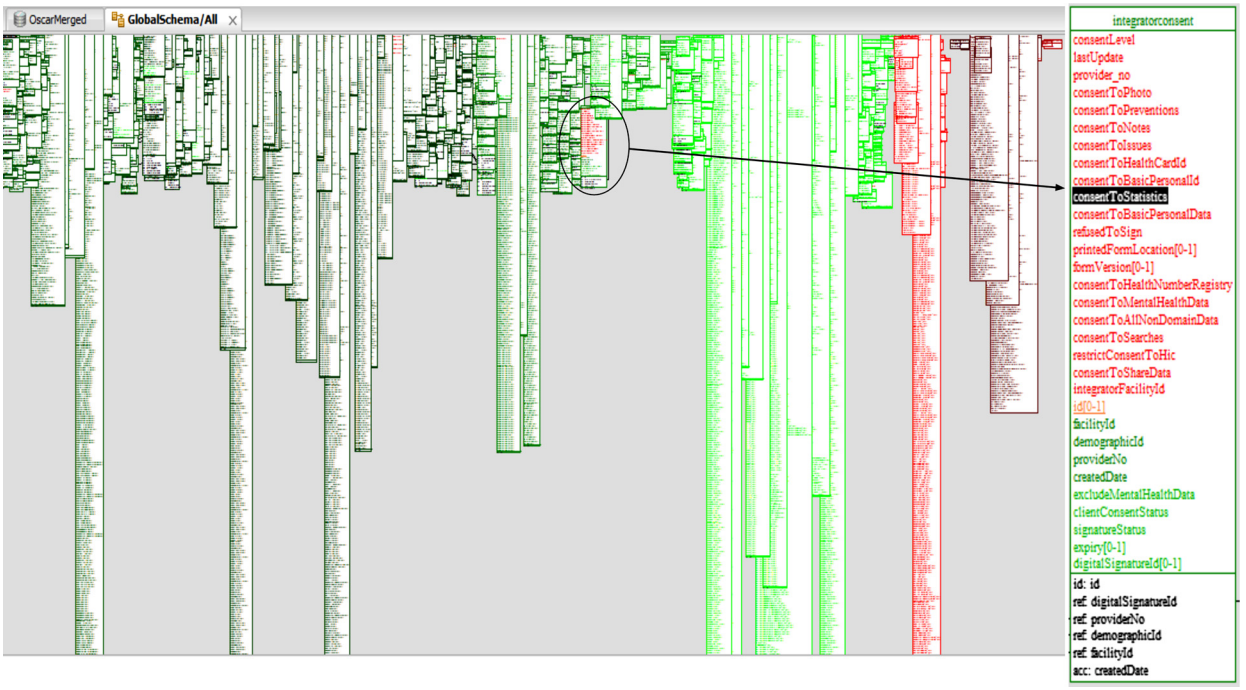

Fig. 2. The OSCAR historical schema as it can be viewed in DB-MAIN (left), and a zoom on a particular table (right).

It includes 21 columns that are in red, meaning that they have been deleted. The table also contains one deleted column that had several lives, as well as 9 columns that still belong to the latest schema version. Among those 9 *active* columns, 4 are present in the table since its creation, while the 5 other ones have been added more recently. When selecting a given schema table or column of the historical schema, the user may inspect its associated meta-attributes (creation date, number of versions, etc.) via the *property box* of DB-MAIN.

We also provide the user with a historical schema querying tool, allowing the extraction of interesting statistics regarding the evolution of the schema of interest. Some of those statistics for the OSCAR database are given below. In addition, the history of each single schema object (table or column) is also available via our tool. This historical information includes the *date*(s) of all creation, update and deletion operations, and for each of them, the id of the corresponding developer.

Fig. 3(A) depicts the evolution of the number of tables in the OSCAR schema. We can observe that this number keeps increasing. Indeed, we found out that in our case study, developers are very reluctant in removing tables in order to achieve backward compatibility and avoid the important impact of a schema refactoring on the data and the application programs. From the same figure, we can also easily identify those schema versions that could be considered as “major releases”, i.e., those versions where an important number of tables have been added and/or deleted. Fig. 3(B) represents the evolution of the total number of columns in the OSCAR schema, that has grown from 2443 to 13 364 columns in circa ten years. Fortunately, this number follows a similar trend as the evolution of tables, keeping the average number of columns per table quite stable over time (around 25).

Fig. 3(C) provides some finer-grained information about the creation and deletion of tables. One can easily notice that OSCAR tables are rarely removed. The evolution of the schema consists (most of the time) of adding tables, while not replacing or splitting them up. The total number of deleted tables is around 30, and we can again quickly identify the major release time periods. The observation is similar for the ratio of created and deleted columns, as shown in Fig. 3(D). The number of column creations is, indeed, often greater than the number of column deletions. During the last releases, however, the number of columns more instable: 964 columns were created in version 452, 954 columns were deleted in version 453. The explanation is the following: at release 452, huge tables (each including hundreds of columns) have been added to the database, and the peak that we observe originates from the deletion of some of those tables. During our test period, a total of 3872 columns were removed, while 14 793 columns were created.

Table 1 provides statistics about the different types of schema changes applied to the OSCAR schema during the studied period. In those statistics, the *add column* operation corresponds to the creation of a new column in an *existing* table. Creating a table including n columns only counts as one *add table* operation, not as n *add column* operations.

Fig. 3(E) shows the classification of the OSCAR tables according to two dimensions: their creation schema version (X axis) and their size, expressed in number of columns (Y axis). We can notice that the large tables (over 200 columns) are created throughout the whole life of the system. This means that those tables are not a reflection of early design problems but are still being generated and used now. We investigated further this unexpected observation and found that the large tables

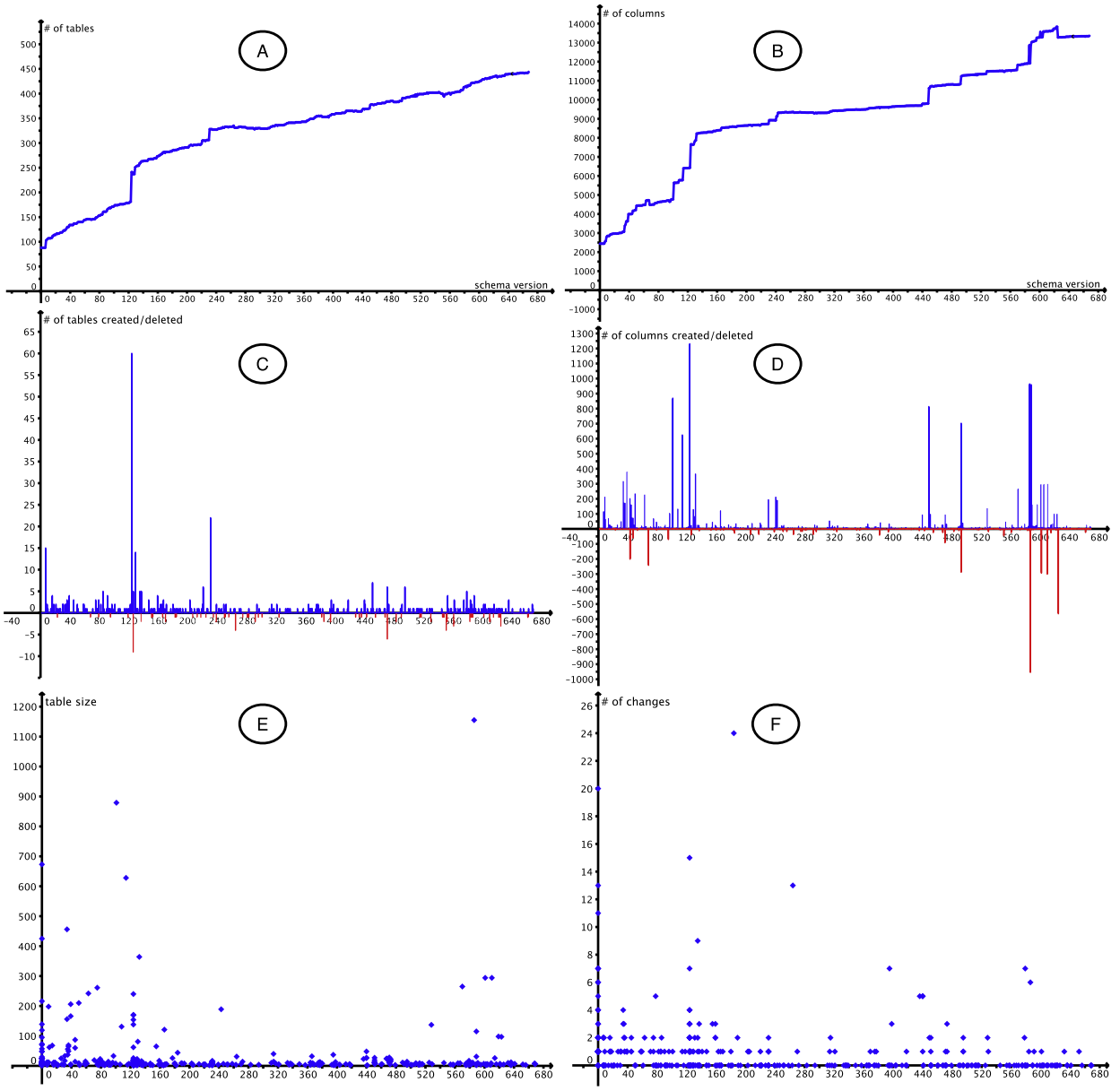

Fig. 3. Extracted information from the OSCAR historical schema: (A) evolution of the number of tables; (B) evolution of the number of columns; (C) creation/deletion of tables; (D) creation/deletion of columns; (E) table creation version VS table size; (F) table creation version VS number of changes.

Table 1
Distribution of the OSCAR schema changes during the considered time period.

add table	443	add identifier	48	add index	125	change default value	23
drop table	86	drop identifier	21	drop index	32	change column type	342
add column	2091	add foreign key	3	add default value	1509	set nullable column	32
drop column	703	drop foreign key	12	drop default value	47	set non-nullable column	27

were related to a user-programmable extension mechanism of OSCAR, which provides “power-users” with tools to add so-called *forms* to OSCAR for capturing specialized clinical data input.

Another interesting property, particularly in the context of database migration, is the *stability* of the tables. A table that has been created a long time ago, and that was not subject to frequent modifications can be considered *stable*. In Fig. 3(F) we characterize each table with respect to the number of times it has been modified since its creation, and we relate this information to its creation version. We can see that the database schema is globally stable, most of the tables having less

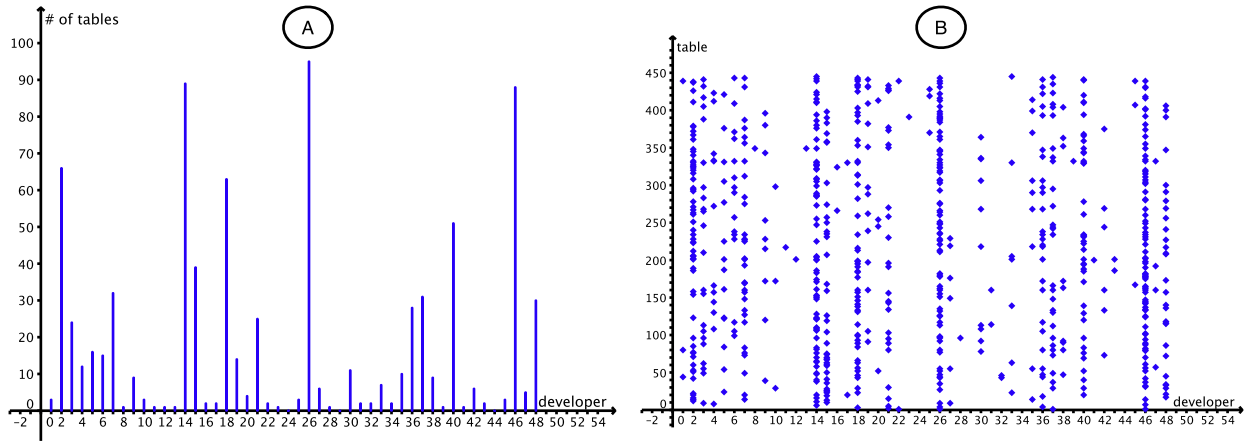

Fig. 4. Information about the developers involved in the evolution of the OSCAR database schema. (A) number of tables impacted by each developer; (B) table vs developer matrix.

than 4 modifications. As expected, it is mainly the oldest tables that have the higher number of changes, but there are a few exceptions.

Fig. 4 relates the OSCAR developers with the evolution of the schema. Fig. 4(A) shows, for each developer, the number of distinct tables in the evolution of which she has been involved in (by creating, updating or deleting the table). We observe that the few most active schema committers have hardly touched 20% of the OSCAR tables. Fig. 4(B) provides a set of points (x, y) meaning that developer x has been involved in the evolution of table y. Such information is useful, for instance, to identify the *experts* of a given table, or the creator of a given schema object. We can also formulate the assumption, to be confirmed in the next steps of our research, that the set of tables modified in a short period of time by the same developer are somehow related to each other. This can potentially constitute additional insights in the context of implicit foreign key detection.

6. Discussion

Analyzing the evolution history of the OSCAR database schema has helped us in understanding the current schema structure and informed our process of developing a software for migrating OSCAR data to the PCRN research data base. We were surprised to see that schema structures are rarely deleted even if they are semantically replaced by others, for example as in the case of the “preventions” table structure replacing the “immunizations” structure (see above). In these kinds of situations, our tool-based method allowed us to easily determine which schema structure superseded which other schema structure. There are potentially multiple explanations for keeping these legacy schema structures. The most likely explanation is domain-specific, namely that they are kept for medico-legal reasons: a patient record (electronic or otherwise) is a legal document that can only be amended but not deleted or arbitrarily modified. Keeping superseded database schema structures is the easiest way of accommodating older patient data that uses these outdated fields. Indeed, in case of the preventions tables, the OSCAR preventions user dialog has at the bottom a link entitled “old immunizations”, which allows the user to access the legacy data.

To some degree, the creation of new schema structures to semantically replace existing ones without removing them is similar to the well-known process of “cloning” in program code. Therefore, the preventions tables and the immunization tables could be considered *database clones*. However, subtle differences exist to the concept of program clones. For example, program clones are usually still made up of functional code (as opposed to dead code), while the superseded database clone may indeed be considered “dead schema” from the point of view of at least a newer installation of the information system software, i.e., an installation that does not have to deal with legacy data that uses the superseded database structure.

Another important insight created by our schema evolution analysis method was a better understanding of the role of the many large-scale tables that contain hundreds or even thousands of attributes. The information content of these tables overlaps significantly with other parts of the transactional database. Therefore, our initial hypothesis (before considering the evolution history) was that these tables were relicts of early database designs. The evolution analysis, however, refuted this hypothesis and indicated that these tables are indeed being generated throughout the system lifetime. Further investigation showed that they were connected to an extension mechanism. We were therefore able to exclude them for the first phase of our database migration project, which simplified our task significantly.

Methods for analyzing the evolution history of database schemas could be helped significantly by having access to actual production data. Unfortunately, this is virtually impossible due to the sensitive nature of the data in case of the OSCAR system. One way to get around this problem is to transform the production data in a set of unrealized data, which still retains certain properties that are important to understand the schema evolution. For this purpose, we have written a program to securely hash encrypt OSCAR patient data prior to export. Special values such as `null` and timestamps would

remain unchanged during this transformation. We have sought and received ethics approval for using our unrealization process to transform real patient data for the purposes of understanding software evolution. We are currently working with a clinic in Vancouver on executing this process. The resulting data set will provide us with further data to help mine for "dead schema" and "cloned" schema structures.

7. Related work

Previous database understanding techniques take as input one or several of the following information sources:

- the *database schema* [2–4]. Spotting similarities in names, value domains and representative patterns may help identify hidden constructs such as foreign keys;
- the *data* [5–8]. Mining the database contents can be used in two ways. Firstly, to discover implicit properties, such as functional dependencies and foreign keys. Secondly, to check hypothetic constructs that have been suggested by other means;
- the *programs* [9–12]. Basic dataflow analysis techniques can already bring valuable information about table and column structures and meaningful names. More sophisticated techniques such as program slicing can be used to identify complex constraint checking or implicit foreign keys. Dynamic program analysis techniques have also been proposed to do the same job in presence of highly dynamic database manipulation code [13].

It is important to note that none of those three sources of information is sufficient, yet they can all contribute to a better knowledge of the hidden components and properties of a database schema. In the present work, we argue that considering historical information as additional, complementary input has the potential to further enrich the knowledge that can be recovered.

Since the seminal work by Manny Lehman on the laws of software evolution [14,15], many researchers have conducted empirical software evolution research, working on a wide variety of topics such as analyzing the co-evolution of test and production code [16], analyzing code cloning [17], predicting bugs from history [18,19], and many more.

While the literature on database schema evolution is very large [20], very few authors have proposed approaches to systematically *observe* how developers cope with database evolution in practice. Existing work in this domain [21,22] analyzed the evolution of rather *small* database schemas. Curino et al. [21] present a study of the structural evolution of the Wikipedia database, with the aim to extract both a micro-classification and a macro-classification of schema changes. In this study, a period of 4 years has been considered, corresponding to 171 successive versions of the Wikipedia database schema. The latter is rather limited in size: It includes from 17 to 34 tables depending on the schema version considered. The total number of columns in the schema does not exceed 250, whatever the version. Lin et al. [22] study the co-evolution of database schemas and application programs in two open-source applications, namely Mozilla and Monotone. The number of schema changes reported is very limited. In Mozilla, 20 table creations and 4 table deletions are reported in a period of 4 years. During 6 years of Monotone schema evolution, only 9 tables were added while 8 tables were deleted.

In this paper, that extends our recent previous results [23], we report on the history analysis of a very large database schema that is several orders of magnitude larger than the schemas analyzed in all previous similar studies. We introduce the concept of global historical schema, and propose a scalable tool allowing to derive such a schema from the versioning system, in a browsable and queriable format. The colored visualization of the global historical schema is analogous to the approach of visualizing program change sets at the architectural level with UML class diagrams [24]. A myriad of other inspiring software visualization tools have been proposed to support the understanding of large-scale software systems evolution, including CodeCity [25], the Evolution Radar [26] and ExtraVis [27].

8. Conclusions

This paper reports on our experiences made in the context of large-scale project aiming at evolving OSCAR, a large and complex medical information system. We present a tool-supported process that allowed us to analyze the evolution history of the OSCAR database over a period of circa ten years. The method is based on the automated derivation of a global historical schema, that includes all the schema objects involved in the entire lifetime of the database, each annotated with historical and temporal information. The identification is schema changes is currently limited to 16 distinct operations. More sophisticated analysis techniques have to be developed in order to identify such changes as renaming a column, splitting a table, or merging two tables into one single table. Since our approach only considers successive schema versions as input, such *refactoring* operations are currently seen as the combination of deletion and creation operations.

While this work only makes a first humble step towards the understanding of large database evolution histories, it also opens several important new research and collaboration perspectives for the entire software evolution research community. First, because the OSCAR system constitutes by itself, considering its size and complexity, a very rich case study that could be used as a basis for validating other reverse engineering techniques and identifying novel research questions in the field. Second, because considering the link between the evolution of the database and the evolution of all the other software artifacts remains a largely unexplored yet important research domain. Such well-known notions as code ownership, cloning,

late propagation, bad smells, bug linkage, bug assignment, etc. do also make much sense in the context of database/program co-evolution.

Acknowledgements

This work has been supported by the F.R.S.-FNRS, in the context of the DISSE research project.

References

[1] DB-MAIN, The DB-MAIN official website, http://www.db-main.be, 2011.
[2] S.B. Navathe, A.M. Awong, Abstracting relational and hierarchical data with a semantic data model, in: Proc. of the Sixth International Conference on Entity-Relationship Approach (ER'1987), North-Holland Publishing Co., 1988, pp. 305–333.
[3] V.M. Markowitz, J.A. Makowsky, Identifying extended entity-relationship object structures in relational schemas, IEEE Trans. Softw. Eng. 16 (8) (1990) 777–790.
[4] W.J. Premerlani, M.R. Blaha, An approach for reverse engineering of relational databases, Commun. ACM 37 (5) (1994) 42.
[5] R.H.L. Chiang, T.M. Barron, V.C. Storey, Reverse engineering of relational databases: extraction of an EER model from a relational database, Data Knowl. Eng. 12 (2) (1994) 107–142.
[6] S. Lopes, J.-M. Petit, F. Toumani, Discovering interesting inclusion dependencies: application to logical database tuning, Inf. Syst. 27 (1) (2002) 1–19.
[7] H. Yao, H.J. Hamilton, Mining functional dependencies from data, Data Min. Knowl. Discov. 16 (2) (2008) 197–219.
[8] N. Pannurat, N. Kerdprasop, K. Kerdprasop, Database reverse engineering based on association rule mining, CoRR arXiv:1004.3272.
[9] J.-M. Petit, J. Kouloumdjian, J.-F. Boulicaut, F. Toumani, Using queries to improve database reverse engineering, in: Proc. of the 13th International Conference on the Entity-Relationship Approach (ER'1994), Springer-Verlag, 1994, pp. 369–386.
[10] G.A. Di Lucca, A.R. Fasolino, U. de Carlini, Recovering class diagrams from data-intensive legacy systems, in: Proc. of the 16th IEEE International Conference on Software Maintenance (ICSM'2000), IEEE Computer Society, 2000, p. 52.
[11] J. Henrard, Program understanding in database reverse engineering, PhD thesis, University of Namur, 2003.
[12] A. Cleve, J. Henrard, J.-L. Hainaut, Data reverse engineering using system dependency graphs, in: Proc. of the 13th Working Conference on Reverse Engineering (WCRE'2006), IEEE Computer Society, Washington, DC, USA, 2006, pp. 157–166.
[13] A. Cleve, J.-R. Meurisse, J.-L. Hainaut, Database semantics recovery through analysis of dynamic SQL statements, J. Data Semant. 15 (2011) 130–157.
[14] M.M. Lehman, On understanding laws, evolution, and conservation in the large-program life cycle, J. Syst. Softw. 1 (1984) 213–221.
[15] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, W.M. Turski, Metrics and laws of software evolution – the nineties view, in: Proc. of the 4th Int'l Symp. on Software Metrics, METRICS '97, IEEE CS, 1997, p. 20.
[16] A. Zaidman, B.V. Rompaey, A. van Deursen, S. Demeyer, Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining, Empir. Softw. Eng. 16 (3) (2011) 325–364.
[17] N. Göde, R. Koschke, Frequency and risks of changes to clones, in: Proc. of ICSE '11, ACM, New York, NY, USA, 2011, pp. 311–320.
[18] P.J. Guo, T. Zimmermann, N. Nagappan, B. Murphy, Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows, in: Proc. of ICSE '10, ACM, New York, NY, USA, 2010, pp. 495–504.
[19] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empir. Softw. Eng. 17 (4–5) (2012) 531–577.
[20] E. Rahm, P.A. Bernstein, An online bibliography on schema evolution, SIGMOD Rec. 35 (4) (2006) 30–31.
[21] C. Curino, H.J. Moon, L. Tanca, C. Zaniolo, Schema evolution in wikipedia – toward a web information system benchmark, in: J. Cordeiro, J. Filipe (Eds.), ICEIS (1), 2008, pp. 323–332.
[22] D.-Y. Lin, I. Neamtiu, Collateral evolution of applications and databases, in: Proc. of IWPSE-EVOL'09, ACM, 2009, pp. 31–40.
[23] M. Gobert, J. Maes, A. Cleve, J. Weber, Understanding schema evolution as a basis for database reengineering, in: Proceeding of 29th IEEE International Conference on Software Maintenance (ICSM 2013), IEEE CS, 2013, pp. 472–475.
[24] A. McNair, D.M. German, J. Weber-Jahnke, Visualizing software architecture evolution using change-sets, in: Proc. of WCRE'07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 130–139.
[25] R. Wettel, M. Lanza, R. Robbes, Software systems as cities: a controlled experiment, in: ICSE, ACM, 2011, pp. 551–560.
[26] M. D'Ambros, M. Lanza, M. Lungu, Visualizing co-change information with the evolution radar, IEEE Trans. Softw. Eng. 35 (5) (2009) 720–735.
[27] B. Cornelissen, A. Zaidman, A. van Deursen, A controlled experiment for program comprehension through trace visualization, IEEE Trans. Softw. Eng. 37 (3) (2011) 341–355.