A Redundancy-Free Approach to Schema Evolution

Claudiomar Desanti[1], Marcos Bernardelli[1], Márcio Fuckner[1], Raquel Kolitski Stasiu[12]

[1] Pontifícia Universidade Católica do Paraná, Brazil
[2] Universidade Tecnológica Federal do Paraná, Brazil
{c.desanti, m.bernardelli, marcio.fuckner, raquel.stasiu}@pucpr.br, raquel@utfpr.edu.br

Abstract. Changes in relational database schemas are part of the continuous improvement process of information systems. In general, iterative and incremental approaches for software development produce frequent releases in order to improve the operational environment and also attend to business or legal requirements. As an effect, such dynamic produces significant amounts of work to database administrators when applying those changes in production. The risk also increases as cumulative changes need to be manually executed in the environment. Having those issues in mind we propose an approach to automate the schema evolution, using a redundancy-free algorithm to merge cumulative changes, reducing downtimes and improving the software availability. When executed in a production environment, the proof-of-concept demonstrated a significant reduction of the amount of time to process the update, resulting in the same schema as applied by known tools.

Categories and Subject Descriptors: H.2.1 [**Database Management**]: Logical Design—*Schema and subschema*

Keywords: Diff Algorithms, Evolving Database Systems, Schema Evolution

1. INTRODUCTION

Scope changes are part of the continuous improvement process of information systems. Several software development teams use iterative and incremental development approaches in order to accelerate the requirements gathering process, in contrast to traditional waterfall approaches. The agile approach leads to frequent changes of software artifacts and requires an effective change management process. As a consequence, those changes affect database structures and their current data [Batini et al. 2009] [Roddick 2009]. Also, several companies acquire off-the-shelf solutions, which aim at attending different configurations and customer needs. It leads companies to update their environment with small, but frequent releases generated from their suppliers. Risks and downtime rates increase when cumulative and interdependent scripts are applied. As a result, customers decrease the frequency of updates, adopting cumulative "big bang" update strategies, which demands complex, interdependent and error-prone tasks.

According to [Fowler 2002], refactoring is a quality improvement process to rewrite source-code without affecting the resulting product. Changes could improve readability, which has impact on future maintenance. Moreover, architectural changes could allow software modularization and improve testability. [Ambler and Sadalage 2006] have adapted Fowler's concept to fit within the database reality. Such changes aim at attending an indirect business requirement or improve the integration between the software and the database. These improvements could generate new database objects such as indexes, columns, relationships and constraints.

Schema evolution is the ability for a database schema to evolve without the loss of existing information. Due to its great significance and practical importance, many research efforts have been made to propose approaches for schema evolution. The work of [Papastefanatos et al. 2008] proposes a

graph-based framework for capturing changes in database schemas and generating annotation based on a proposed extension to the SQL language, specifically tailored for the management of evolution. Using a similar approach, the work of [Curino et al. 2009] presents a web-based framework, which uses several semantic techniques for query rewriting. Techniques for mapping the differences between models can also be used as a key element for the schema evolution problem, even if they address different contexts such as ontologies and object-oriented models. For example, [Fagin et al. 2011] presents two fundamental operators on schema mappings, namely composition and inversion to address the mapping adaptation problem in the context of schema evolution. [Carvalho et al. 2013] proposes an evolutionary approach to find complex matches between schemas of semantically related data repositories using only the data instances. The work of [dos Santos Mello et al. 2002] describes a unification method for heterogeneous XML schemata.

Differently from other works, this approach proposes a schema evolution mechanism that minimizes possible redundancies that exists in a set of schema updates. A schema update in this particular scenario could be interpreted as a group of SQL sentences, arranged in a labeled version. The proposed mechanism allows the user to apply the set of updates in only one step. For the sake of clarity, the solution could be decomposed in two main processes: (i) Generation of a delta version with the difference between two schemas and (ii) merging a set of delta versions. In (i) the inputs come from the development team, whereas in (ii), the tasks are conducted by database administrators who want to refresh their environments with the last stable version of the software. The results show that our approach reduced the amount of time to process the update, resulting in the same target schema as applied by known tools.

2. A REDUNDANCY-FREE APPROACH TO SCHEMA EVOLUTION

A method was developed to address the problem of finding differences between two schemas using a diff algorithm. This type of algorithm aims at finding the shortest distance between two scripts, which can be represented by t_i and t_{i-1}, where t represents the schema and i is its version number. The result is a delta version, represented by d, which has directives to allow the construction of one of the t versions using the previous or a subsequent version [Cobena et al. 2001]. In this particular case, both scripts and deltas are sets of SQL commands. The generated prototype was developed to recognize PostgreSQL objects [PostgreSQL Global Development Group 2013]. However, a clear separation between core and interface layers demands few changes to adapt the prototype to work with other relational databases. The next section describes the process to obtain the delta version.

2.1 FINDING THE DELTA VERSION BETWEEN TWO SCHEMAS

A database schema is a hierarchical structure of objects. Hence, a tree could be used as its representation. Figure 1 presents an illustrative and non-exhaustive tree that fits to the natural taxonomy of a schema. To generate the delta d, we transform schemas t_i and t_{i-1} into trees and use them as an input to the diff algorithm. To discover the differences, the algorithm needs to identify which node was changed (e.g., attribute, table or index). To provide such functionalities the algorithm traverses both trees from the upper nodes to the leaf, matching each one with their corresponding node in the opposite tree.

In order to avoid visiting all nodes during the comparison, the MD5 (*Message-Digest Algorithm 5*) [1] hash function is executed for each node, using the associated SQL sentence as an input parameter. Figure 2 shows two trees with their calculated hash values. When traversing the tree (a), the algorithm ignores the node 1 because both hashes from trees (a) and (b) are the same. By the other hand, the same example shows $node2$ with different hash values, which justifies further verification.

[1] A complete specification of the MD5 algorithm can be found at *http://tools.ietf.org/html/rfc1321*

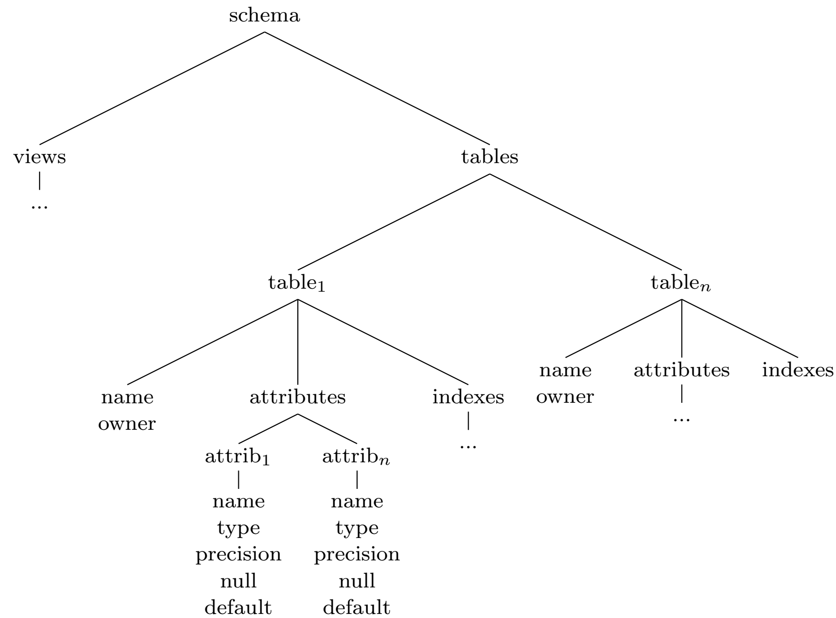

Fig. 1. A tree representing the schema

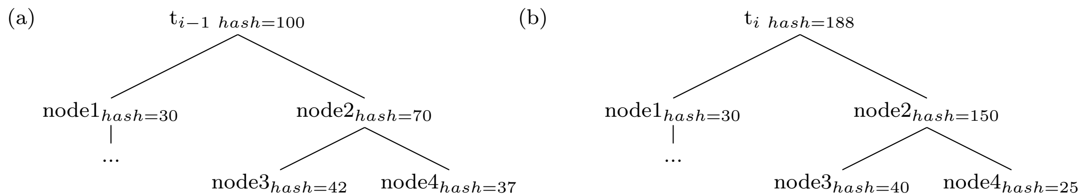

Fig. 2. A tree representing the schema

The algorithm evaluates each node from the versions t_i and t_{i-1} using three possible patterns: *drop* operations indicating that one or more objects in t_i were removed from t_{i-1}, *create* operations indicating that one or more new objects in t_i were identified, and *modify* operations selecting objects belonging to both trees, with different values on at least one common attribute in the node.

At the end of the comparison process, a delta version is generated in XML format (see Figure 3). A root node called *version* contains the attribute that indicates the version number. The subsequent nodes represent structural changes by object type. The implemented prototype recognizes structural changes in tables, views, sequences and indexes. Each one has specific metadata, providing sufficient information to change the schema. There is a common attribute called *op*, which identifies the type of difference (*drop*, *create* or *modify*). One special node called *scripts* can be used to allow administrators to insert user-defined scripts, commonly used for conversion process purposes.

2.2 MERGING A SET OF DELTA VERSIONS

This section presents an approach to evaluate a set of delta versions, identify and remove their redundancies, and finally apply those changes in the target schema. The approach is organized as follows: (i) recovery of delta versions from the repository, (ii) merging the delta versions, removing redundancies, and applying the changes in the target schema. In (i) the current schema and the set of deltas are selected from one repository. Schemas should be located in a configuration control infrastructure with support to artifact versioning and tracking, which is out of the scope of this work.

```
<version id="3">
   <tables>
      <table id="invoice" operation="modify">
         <attributes>
            <attr id="number" op="create"><name>number</name><type>integer</type> ...
            <attr id="comments" op="modify"><name>remarks</name> <type>varchar</type> ...
         </atributes>
      </table>
   <tables>
   <scripts>...</scripts>
</version>
```

Fig. 3. Generated delta example

In (ii), the algorithm receives a list of delta scripts, identify and merge those changes in one redundancy-free script. Three types of changes could occur in one database object: *drop* operation, *create* operation, and *modify* operation. For illustration purposes, the following scenario with a delta script in the version v_1 can be considered. This delta script from version 1 creates a new field in the table *invoice* called *trackingNumber*. The field *trackingNumber* is an integer field that accepts null values and has a precision of 8 digits. In version v_2 the field is changed in order to forbid null values. Finally, in the version v_3, the precision is changed to support 10 digits instead of 8.

The Algorithm 1 was created in order to mimic the decisions typically made by one database administrator when merging two scripts. The function called $schemaMerge$ requires an input parameter called D, which has the set of deltas to merge. Lines from 12 to 20 are responsible for evaluate each sentence from the delta, using a rule-based algorithm detailed in Algorithm 2 called *mergeSentences*. The rules guide the merging process, deciding what is the most economic operation. Redundancies are removed as a consequence of a series of override operations in the underlying object. For example, taking two SQL sentences s_1 = *"create table foo (field1 integer)"* and s_2 = *"alter table foo add field2 char(1)"*. The first sentence ($s1$) is a *create operation*, while the second ($s2$) is a *modify operation*. According to the rule 3 described in the algorithm 2, the merging process will generate only one merged sentence S_r = *"create table foo (field1 integer, field2 char(1))"*.

Algorithm 1 Schema merging algorithm

Require: $D = d_1, d_2, ..., d_n$: delta bundle
1: **function schemaMerge**
2: $Map \leftarrow empty\ map$ **{Map of schema objects with their changes}**
3: $M \leftarrow empty\ set$ **{Set of merged changes}**
4: **for each delta** $d_i \in D$ **do**
5:    **for each change** $c \in d_i$ **do**
6:       **add** c **in** $Map[c.object\ name].values$
7:    **end for**
8: **end for**
9: **for each object name** $n_i \in Map.keys$ **do**
10:    $current\ change \leftarrow null$
11:    **for each change** $c \in Map[n_i].values$ **do**
12:       $current\ change \leftarrow mergeSentences(current\ change, c)$
13:    **end for**
14:    **if** $current\ change \neq null$ **then**
15:       $M.add(current\ change)$
16:    **end if**
17: **end for**
18: **return** M

3. EXPERIMENTAL RESULTS

This section presents the results of o the first experiment, which evaluates the performance and reliability of this approach. A set of releases of a real ERP called *Auditor*, implemented by the *Method Informatics* company were used as a input for the experiment, resulting in 20 schema-changing scripts

Algorithm 2 Sentence merging functions

```
1: function mergeSentences(current,new)
2: if current = null then
3:     return new
4: end if
5: if current.operation ≡ create and new.operation ≡ drop then
6:     return null
7: end if
8: if current.operation ≡ create and new.operation ≡ modify then
9:     return mergeScript(create, current.sentence, new.sentence)
10: end if
11: if current.operation ≡ drop and new.operation ≡ create then
12:     return new
13: end if
14: if current.operation ≡ modify and new.operation ≡ drop then
15:     return new
16: end if
17: if current.operation ≡ modify and new.operation ≡ modify then
18:     return mergeScript(modify, current.sentence, new.sentence)
19: end if
20:
21: function mergeScript(type,current,new)
22: if type ≡ create then
23:     return current + new
24: else
25:     return current ∪ new
26: end if
```

delivered from 2011 to 2012. The samples allowed the detection of different scenarios, from simple inclusion of objects to more complex tasks such as column merging and splitting. The scripts were applied in two databases namely A and B, with the same schema but varying in size (170mb and 410mb). Splitting and merging operations must be added in the delta script, specifically inside the tag called *scripts* as shown in Figure 3. Table I also shows relevant numbers about the experiment.

Table I. Experiment numbers

Information	#
# of tables at the baseline version	317
# of schema-changing scripts	20
# of new tables	38
# of modified tables	21
# of modifications (attribute insertions, deletes and updates)	239
# of tables at the 20th version	355

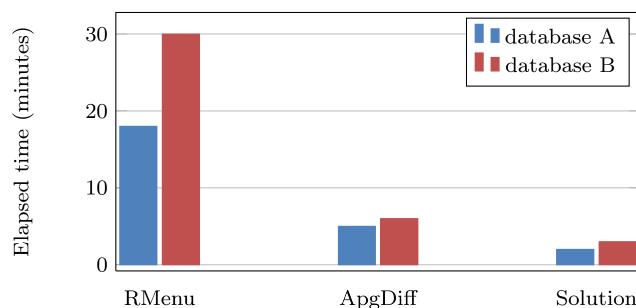

Fig. 4. Comparison of approaches (execution time).

Rmenu [Genexus 2013] and ApgDiff [StartNet 2013] were selected to make a preliminary comparison, with focus on response time and reliability. After the evolution, all schemas must be exactly the same. Both Rmenu and ApgDiff were selected because they are mature and multi-function tools, used in several environments. As it can be seen in Figure 4, the prototype was able to reduce the time spent

during the execution, since there is no extra work in order to process modifications that may be superposed. Compiled versions optimizes the DDL script generated to modify the database schema. For this optimization, all the detected redundancies were removed, reducing the number of DDL instructions and, consequently, decreasing the execution time.

4. CONCLUSIONS AND FUTURE WORK

This paper deals with the problem of database evolution maintenance using an approach to integrate the schema versioning during the database refactoring process, avoiding redundancies during the execution of several schema evolutions. The first proof-of-concept allows the automation and optimization of potential set of changes applied in one database schema. The applicability and efficiency of this approach has been tested using real world examples. A significant time reduction was observed in order to process the update, resulting on the same target schema as applied by known tools.

This work presents the initial results of experiments, where several improvements were detected. One of those improvements includes the evaluation of the approach on different databases as well as adding other perspectives to the benchmark, such as the readability of the generated scripts and identification of schema-changing patterns. There are three others directions of investigation: to identify the dependence objects in time of modification, to eliminate the redundancy during data update by changing the order of DDL script execution and to study techniques to identify the actual version of database schema.

REFERENCES

Ambler, S. W. and Sadalage, P. J. *Refactoring Databases: Evolutionary Database Design.* Addison-Wesley, 2006.

Batini, C., Cappiello, C., Francalanci, C., and Maurino, A. Methodologies for data quality assessment and improvement. *ACM Computing Surveys (CSUR)* 41 (3): 16:1–16:52, July, 2009.

Carvalho, M. G., Laender, A. H. F., Gonçalves, M. A., and Da Silva, A. S. An evolutionary approach to complex schema matching. *Information Systems* 38 (3): 302–316, May, 2013.

Cobena, G., Abiteboul, S., and Marian, A. Detecting changes in XML documents. In *Proceedings of the 2001 IEEE International Conference on Data Engineering.* ICDE '01. pp. 41–52, 2001.

Curino, C. A., Moon, H. J., Ham, M., and Zaniolo, C. The PRISM workbench: Database schema evolution without tears. In *Proceedings of the 2009 IEEE International Conference on Data Engineering.* ICDE '09. IEEE Computer Society, Washington, DC, USA, pp. 1523–1526, 2009.

dos Santos Mello, R., Castano, S., and Heuser, C. A. A method for the unification of XML schemata. *Information & Software Technology* 44 (4): 241–249, 2002.

Fagin, R., Kolaitis, P. G., Popa, L., and Tan, W. C. Schema mapping evolution through composition and inversion. In *Schema Matching and Mapping.* pp. 191–222, 2011.

Fowler, M. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 2002.

Genexus. *RMenu.* Genexus, 2013. Available at http://www.genexus.com.

Papastefanatos, G., Vassiliadis, P., Simitsis, A., Aggistalis, K., Pechlivani, F., and Vassiliou, Y. Language extensions for the automation of database schema evolution. In *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems.* pp. 74–81, 2008.

PostgreSQL Global Development Group. PostgreSQL, 2013. Available at http://www.postgresql.org.

Roddick, J. F. Schema evolution. In *Encyclopedia of Database Systems.* pp. 2479–2481, 2009.

StartNet. *Another PostgreSQL Diff Tool (apgdiff) free database schema diff tool.* StartNet, 2013. Available at http://www.apgdiff.com.