Supporting exploratory analytics on repository-extracted schema histories by integrating external contextual information

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Athanasios Pappas

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
WITH SPECIALIZATION
IN SOFTWARE

University of Ioannina
June 2017

DEDICATION

To my family.

ACKNOWLEDGMENTS

To anyone and everyone that has mentally, and otherwise helped me in succeeding this huge task. First and foremost, I would like to thank my supervisor, Prof. Panos Vassiliadis for his guidance throughout my graduate studies. Secondly, I would like to express my gratitude to my family, for their support all of these years

Thank you all for making this thesis possible.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Athanasios Pappas. MSc in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece. June 2017.

Supporting exploratory analytics on repository-extracted schema histories by integrating external contextual information.

Advisor: Panos Vassiliadis, Associate Professor.

Data-intensive software systems evolve over time and, as part of this evolution process, so does the schema of any database which is included as an integral part of them. Version control systems store the version histories of open source software projects and the information extraction from these histories can be useful for gaining insights about their evolution. Alongside with the software evolution, new information is posted in different external systems improving in this way the software development experience for example. In this thesis, we combine all the various, heterogeneous, dissimilar sources of information for the history of a schema in one reference model which represents all the aspects of repository-based information. Then, we use the defined reference model to create a system that supports both an interactive and a traditional way to exploratory analytics using the integrated contextual information about the schema histories. Beyond that, we use the same meta-model in order to group the entire lifetime of a database into phases, to which we refer to the term release, and perform a study on how these phases are related to changes affecting the schema of the database. Based on our findings, we can argue that change is mostly absent or kept in small numbers in contrast with few releases collecting a large percentage of the changes.

Εκτεταμενη Περιληψη στα Ελληνικα

Αθανάσιος Παππάς. ΜΔΕ στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2017.

Αναλυτική πλοήγηση σε δεδομένα αποθετηρίου για την εξέλιξη του σχήματος μιας βάσης δεδομένων δια της ενοποίησης εξωτερικών πηγών πληροφορίας

Επιβλέπων: Παναγιώτης Βασιλειάδης, Αναπληρωτής Καθηγητής.

Όπως το λογισμικό ανοικτού κώδικα έτσι και οι βάσεις δεδομένων, οι οποίες αποτελούν αναπόσπαστο κομμάτι του λογισμικού, εξελίσσονται με την πάροδο του χρόνου. Τα δημόσια αποθετήρια κώδικα είναι συστήματα που αποθηκεύουν τις αλλαγές που έχει υποστεί ένα λογισμικό κατά την διάρκεια της ανάπτυξής του. Η εξαγωγή των αλλαγών αυτών από τα αποθέτηρια είναι χρήσιμη για την μελέτη και την κατανόηση τόσο της εξέλιξης του λογισμικού όσο και της εξέλιξης των σχημάτων βάσεων δεδομένων. Παράλληλα με την εξέλιξη του λογισμικού, νέες πληροφορίες δημιουργούνται σε διαφορετικά εξωτερικά συστήματα τα οποία χρησιμοποιούνται για την βελτίωση της ανάπτυξης του λογισμικού. Παραδείγματα τέτοιων συστημάτων μπορεί να είναι τα συστήματα διαχείρισης των προβλημάτων που προκύπτουν στο λογισμικό ή το σύστημα με το οποίο επικοινωνούν οι προγραμματιστές του λογισμικού. Στόχος της συγκεκριμένης μεταπτυχιακής εργασίας είναι η μελέτη όλων αυτών των ετερογενών πηγών πληροφορίας με στόχο την ενοποίηση τους σε ένα μοντέλο το οποίο θα διευκολύνει την μελέτη της εξέλιξης των σχημάτων βάσεων δεδομένων. Πιο συγκεκριμένα, ορίζουμε ένα μοντέλο αναφοράς που περιέχει κάθε πτυχή των ετερογενών αυτών πηγών και κατασκευάζουμε ένα σύστημα το οποίο χρησιμοποιεί το συγκεκριμένο μοντέλο και παρέχει έναν διαδραστικό τρόπο μελέτης της εξέλιξης του σχήματος βάσεων.

Επιπλέον, χρησιμοποιώντας το μοντέλο αναφοράς ομαδοποιούμε την ιστορία μιας βάσης δεδομένων σε φάσεις με βάση τις πληροφορίες που έχουν εξαχθεί από το αποθετήριο κώδικα και μελετούμε πως οι αλλαγές στο σχήμα της βάσης

οργανώνονται σε φάσεις και πώς αυτές σχετίζονται με την ιστορία της εξέλιξης του σχήματος της βάσης δεδομένων. Με βάση τις μετρήσεις μας, μπορούμε να υποστηρίξουμε ότι ένα μεγάλο μέρος της ιστορίας των σχημάτων βάσεων δεδομένων χαρακτηρίζεται από την απουσία αλλαγών. Αντίθετα, ένα μικρό ποσοστό των φάσεων συγκεντρώνει το μεγαλύτερο ποσοστών αλλαγών σε όλη την ιστορία της εξέλιξης του σχήματος βάσεων δεδομένων.

CHAPTER 1.

INTRODUCTION

1.1 Scope

Every software project faces the need to evolve for various reasons. Firstly, new requirements may arise due to the fact that some of the specifications of a large project are not easy to be defined in detail from the start of the project. In addition, bugs may appear or alternations of the requirements that they were not foreseen are likely to occur and need to be addressed by making changes to the source code. Any database which is integral part of software also needs to evolve along with it. Much like any other module that is part of a project, database schemata defining the structure of the database that support the software project, evolve too. The evolving of a database schema to adapt to the new changes is called *schema evolution.*

The last years, the rise of *Free* and *Open Source Software* (FOSS) is rapid due to the fact that distributed version control systems like git[1] support multiple

[1] https://git-scm.com/

repositories and cheap branching capabilities that makes easier the involvement of more developers in a software project. The increase on the amount of developers in open source projects in combination with the software and schema evolution has impact on the amount of data that exist in open source software repositories. This amount of data becomes large and makes the tracking of the development process harder. Beyond that, most of the open source software projects use *external systems* (issue tracking systems, forums, build systems and project management systems) for archiving useful information that improves the development experience. Most of these external systems contain useful *contextual information* which is gathered from the content that is generated from the developers involved in the project or the users that use the software. The most important problem that arises from the usage of all these different sources of information is the difficulty for someone to connect the different pieces of information from those sources together in order to understand the evolution. Therefore, there is the need of a system, or a method that unifies all the data from the different information sources and helps the analysis of open source software repositories.

Until today, the previous studies on the topic of schema evolution (described in detail in Chapter 2) of open source projects were focused on exploring *what* was affected based only in the descriptive statistics that can be extracted from the version history of the database schema. To our knowledge, none of the previous studies combined the information which is archived in the repository and the data from the external systems with the version history of the database schema to enrich the details on *why.* Software development is a human-intensive activity and for this reason, enriching the study of schema and software evolution with additional information is important because we can gain useful insights on how software and database schemas evolve.

In this thesis, we focus on the study of integrating external contextual information in order to support exploratory analytics on the version histories of database schemas. Specifically, one of our main research questions is: ***can we combine the various, heterogeneous, dissimilar sources of information for the history of a schema in one integrated, all-encompassing representation?*** To answer this question, we study in depth the characteristics of open source software repositories and what we can extract from those as well as the types of external systems that are used for improving the development experience. Then, our next step is to create a reference model in order to solve the aforementioned data integration problem. This reference model will help us combine the heterogeneous sources of information into one internal representation that connects everything together.

The second research question we answer is: ***given the entire version history of a schema can we group individual changes in phases?*** The entire lifetime of a database schema can be separated into phases where each of them contains a series of events. Organizing events in phases can be very helpful to understand the general evolution of the system. Focusing on development phases of a project may provide insights regarding the development goals for the specific time period. In order to group the series of events we use information gathered on *releases* where a release is the sum of the stages of development for a piece of software. Then, we discuss statistical findings that we observed on releases and how releases are related to schema changes. Section 3 discusses in detail how the releases are used to group individual changes and Chapter 6 presents the discussion on the statistical findings.

Finally, the last research question is: ***how can statistical observations from schema evolution be used for characterization of individual releases?*** Based on the aggregate measures we gathered on releases for six open source software projects, we follow a rule-based classification technique that utilizes rules to characterize the nature of a release's activity and the intensity of the activity (low, medium, high). Then, we study these characterizations and present our observations.

Based on the above, the contributions of this thesis can be summarized as follows:

- We provide a guide on what we can extract from a typical open source software repository and how we can extract the data from all these dissimilar sources of information.
- We provide a reference meta-model that puts together all the different sources of information into one representation that can be used to analyze the version history of a database schema.
- We create a system which contains a database that is loaded with the information gathered from different sources of information for the life and evolution of a software project, with an emphasis to the life of the database schema and provides an interactive way for the inspection of schema histories.
- We relate releases to individual commits in terms of aggregate values and study these aggregate measures providing useful statistic observations on the schema evolution.
- We provide a principled method for classifying releases based on descriptive statistics extracted from the evolution history of a schema.

1.2 Roadmap

The structure of this thesis is as follows. In Chapter 2, we present related work on change classification and on schema evolution. In Chapter 3, we describe in detail the data that can be extracted from an open source software repository including the different external sources which are usually used in the software development process. In Chapter 4, we define our reference model which combines all the different sources of information into one representation. Chapter 5 describes the architecture of the system we created and also the interactive way that provides in order to explore the entire life of database schemata. In Chapter 6, we present off-line analytics and findings on the data gathered for releases. Finally, Chapter 7 summarizes our findings on schema evolution and presents open issues for future research.

CHAPTER 2.

RELATED WORK

There is a very long list of work in the area of Mining Software Repositories. The flagship conference of the area, MSR, is annually held in conjunction with the major conference of the Software Engineering discipline (ICSE). We have focused only on works that pertain to charting the evolution of a software project. Moreover, we also present results in the area of schema evolution.

2.1 Change Classification

Kagdi et al. [KaMS07] apply a heuristic-based approach that uses sequential-pattern mining to the commits in software repositories for uncovering highly frequent co-changing sets of artifacts (e.g., source code and documentation). Firstly, the authors present three heuristics for grouping related change-sets formed from version history metadata found in software repositories (i.e., developer, time, and changed files). These heuristics can be considered similar to the fixed; and sliding window techniques. Secondly the authors import the changes into a sequential-pattern mining tool, namely sqminer, which they have developed and that is based on the Sequential Pattern Discovery Algorithm (SPADE). To evaluate their approach to recovering traceability links they use the open-source system KDE. The evaluation methodology is to first mine a portion of the version history for traceability patterns (training-set). Next the authors mine a later part of the version history (called the evaluation-set) and check if the results generated from the training-set can accurately predict changes that occur in the evaluation-set. Metrics: Precision, Recall, Coverage.

Kim et al [KiWZ08] present a technique called *change classification* for predicting bugs in file-level software changes. A key insight behind this work is viewing bug prediction in changes as kind of a classification problem assigning each change to one of 2 classes: clean changes and bug changes. Features are extracted from the revision history of 12 projects and each of them is used to train a Support Vector Machine (SVM). Features include all terms (variables, method calls, operators, constants and comment text) in the complete source code, the lines modified in each change (delta), and change metadata such as author and change time. Complexity metrics, if available, are computed at this step. Once a classifier has been trained, new changes can be fed to the classifier which determines if a new change is clean or buggy.

The classification performance is evaluated using the 10-fold cross-validation method and the computation of the standard classification evaluation measures, including accuracy, recall, precision, and F-value.

Hindle et al [HGGH09] (see also [HiGH08]) introduce a method for classifying large commits. Large commits are those in which a large number of files (say thirty or more), are modified and submitted to the Source Control System (SCS) at the same time. The authors show that large source control system (SCS) commit messages often contain enough information to determine the type of change occurring in the commit.

The authors gather the version histories from a set of long-lived projects and they manually classified 1% of the largest commits based on the number of files changed. The authors use the following features to train the classifiers: (1) Word Distribution, (2) Author, (3) Module and File Types. In this work 7 Machine Learners are used: J48, NaiveBayes, SMO, KStar, IBk, JRip, ZeroR and 5 metrics to evaluate each learner.

Finally the authors conclude that commit messages provide enough information to reliably classify large commits. Each learner indicates that there is some consistent terminology internal and external to projects that can be used to classify commits by their maintenance task. The author's identity may be significant for predicting the purpose of a change, which suggests that some authors may take on a role or take responsibility for a certain aspect of maintenance in a project.

Zimmermann et al. [ZWDZ04] apply data mining techniques to obtain association rules from version histories, detect coupling between fine-grained program entities such as functions or variables and thoroughly evaluate the ability to predict future or missing changes. The authors use their ROSE server which collects the version history from CVS archives and then, run the Apriori Algorithm to compute association rules which refer to a set of changed entities. The use of the Apriori Algorithm is to compute all rules beforehand, and then search the rule set for a given situation. For their evaluation, they analyzed the archives of eight large open-source projects. Metrics: Precision, Recall, Likelihood, Closure, Granularity

Herzig and Zeller [HeZe13] try to find out the impact of tangled code changes and use a multi-predictor approach to untangle these changes. A tangled change is eminent when a developer is assigned multiple tasks (let's say A, B, and C) all with a separate purpose (for example A is a bug fix, B is a feature request, and C is a refactoring or code cleanup). Once all tasks are completed, the developer commits her changes to the source code management system (SCM), such that her changes to be visible to other developers and integrated into the product. However, when committing changes, developers frequently group separate changes into a single commit, resulting in a tangled change. In this work the main assumption is that untangling changes can be seen as a code change classification problem. The authors conduct an exploratory study on 5 open-source projects and manually classify more than 7000 individual change sets and check whether they address multiple (tangled) issue reports. In addition, the authors show that 73% of all change sets have 2 of individual tasks compiled into a tangle. Finally, the authors observe that tangled change sets have impact on bug counting models.

Kevic and Fritz [KeFr14], given (a) the vocabulary used for describing change tasks and (b) the one used for identifiers in the source code as two separate languages, introduce an approach for creating a dictionary that maps the different vocabularies using information from change sets and interaction histories stored with previously completed tasks. The dictionary to map natural language to source code language is built by mining previously resolved change tasks from task repositories and the source code associated to these change tasks. The summary and description are preprocessed, resulting in a list of distinct terms. For each term of a change task, it then creates or updates the mapping in the dictionary from the term to all code elements in the change set or task context of a change task. Each mapping between a term in natural language (denoted as NL) and the terms in source code language (denoted as SCL) has a weight that is one at first. To identify the best translation of a change task into SCL, a weight is calculated for each NL term in the change task based on tf/idf and used as a multiplier to update the weights of the mapping. This approach creates an approximate mapping between the terms of the natural language used for change tasks (NL) and the source code language (SCL). Finally the authors evaluate their approach gathering information from four open source projects.

Howard et al in [HGPS13] present an approach to automatically mine word pairs that are semantically similar in the software domain. The authors claim that semantic similarity is useful when analyzing maintenance requests and tools for finding differences between versions of software, especially when the change between two versions involves renaming to a synonymous word. The authors based on the simple observation that a leading comment sentence and a method name are expected to express the same action, they present a sequence of automated steps to map the main action verb from the leading comment of each method to the main action verb of its method signature. In addition, the authors demonstrate all the challenges in designing a system that uses this simple observation that leading descriptive comments and the documented method name should describe the same action. The results show that their miner has 87% accuracy in identifying descriptive comments and 94% accuracy in extracting the correct main action word from a descriptive comment.

2.2 Schema Evolution

One of the first studies on schema evolution was done by D. Sjoberg [Sjob93]. The author created a measuring tool called "thesaurus" which was built to monitor the evolution of a large, industrial database application – a health management system. This system was observed in a period of 18 months and during these months was found that there was 139% increase in the number of relations, as well as 274% increase in the number of fields. In addition, the results showed that, 35% more additions than deletions took place and every relation was changed. Finally, the author noted that the results confirm that change management tools are needed.

Several years later, a new study was published [CMTZ08]. In this study, the authors investigate MediaWiki, the back-end system that powers the well-known Wikipedia. The authors gather 171 different versions over 4 years and 7 months and show that there is a 100% increase in schema size and 142% increase in the number of fields. In addition, they observe that on average, each table and field lasts 60.4% and 56.8% of the total database history respectively. Finally, the authors conclude that there are serious indications that the database schema evolution has impact in application which uses it and support the contention that there is the need for better support in schema evolution.

In [DoBZ13], the authors make an empirical analysis of the co-evolution of database schemas and code in ten popular large open-source database applications. In this work, the authors study how frequently and extensively database schemas evolve, how it evolves and how much application code has to co-changed with a schema change. Specifically, the authors state that the results provide solid evidence that schemas evolve frequently and that schemas increase in size. In addition, the authors note that there are 3 main high-level schema change categories: Transformations, Structure Refactoring and Architectural Refactoring and that schema changes impact code greatly.

One more study [SkVZ14] in larger scale was published in 2014. In this study the authors inspect if the Lehman laws [LMR+97] for the software evolution hold in the database schema evolution. Specifically, the authors collect and study 8 open-source systems using their open source SQL diff tool, Hecate. The results show that there are periods where there is an increase in the schema size, mostly in the beginning or after large decreases in the size, but there are also periods of stability. In addition, the authors observe that the database maintenance exists in all datasets and conclude that the Lehman laws hold in open source database systems.

Until today, the previous studies on the topic of schema evolution of open source projects were focused on exploring what was affected based only in the descriptive statistics that can be extracted from the version history of the database schema. To our knowledge, none of the previous studies combined the information which is archived in the repository and the data from the external systems with the version history of the database schema to enrich the details on why. Software development is a human-intensive activity and for this reason, enriching the study of schema and software evolution with additional information is important because we can gain useful insights on how software and database schemas evolve.

CHAPTER 3.

SOURCES OF INFORMATION FOSS PROJECTS

The last years, the rise of *Free* and *Open Source Software (FOSS)* is rapid due to the fact that distributed version control systems like *Git* support multiple repositories and cheap branching capabilities that makes easier the involvement of more developers in a software project. The large amount of publicly available software repositories attracted the interest of the research community that focus on both qualitative and quantitative studies. In this chapter, we discuss on the availability of data for *FOSS* projects at the web. First, we start with a discussion on the information that can be extracted from the web concerning *Free* and *Open Source Software* projects. Then, we move to our experimental setup where we present the six dataset we used in this thesis along with the retrieval process we followed to extract them from the publicly available repositories.

3.1 What can we extract from the web concerning a Free and Open Source Software (FOSS) project?

The process of extracting useful information from open source software repositories as well as from the external information sources they use has become easier in recent years with the rise of web-based code hosting services like *Github*[2]. However, the mining of publicly available data may have potential risks of misinterpretation. For this reason, there is the need for deep understanding of the characteristics of publicly available data for choosing the appropriate software repositories regarding the specific research goals. In [KGBS14], the authors document the results of an empirical study that aims at understanding the characteristics of the repositories on *Github*, where recommendations to research are provided on how to use the data available from *Github*.

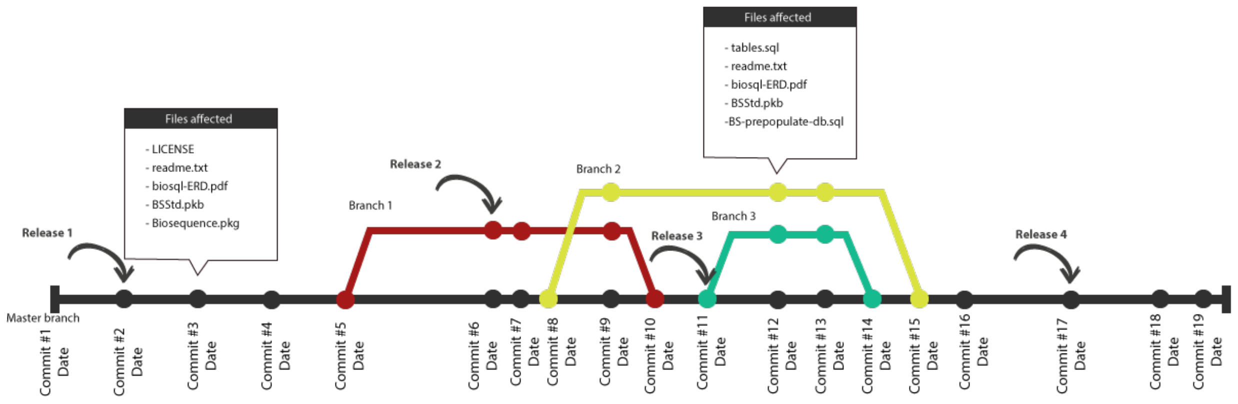


Figure 1 Life story of a software repository

2 https://github.com/

Before we go further on discussing the information that can be extracted from an open source software repository, we need to see an example of a software repository. Figure 1 shows an example of the structure of a *Git* repository. *Git* version control system is software repository with the goal of facilitating, composing and saving snapshots of a project and then working with and comparing those snapshots. Every repository has, by default, a master branch and maybe one or more, optional development branches that may keep different stages of the source code. The different branches can be cloned from the master branch and also can be merged to master. Every branch has a list of commits and every commit consists of changes which are made to one or more files in the repository. Moreover, every commit can contain a tag that provides useful insights regarding the specific commit. These tags may refer to the starting point of a release as we will see later.

Commit is an individual change to a file, or set of files, with a unique *ID* that keeps a record of *what* changes were made *when* and by *whom*.

Branch in *Git* is simply a lightweight movable pointer to one *commit*. The default branch name in *Git* is *master*. This pointer points to the last commit and every time a new commit is made, it moves forward automatically.

Version control systems like *Git* provide a rich set of features for the development process but most of the times active software projects do not conduct all their software development in *Git* as mentioned in [KGBS14]. There is a wide variety of external systems in the web that provide solutions to different development tasks. For example, there are systems for issue tracking, code review, build capabilities, project management or even forums for the development community of a project. Based on the above, we present a list of all the different types of information that can be extracted from an open source software project.

<u>List of branches:</u> constitutes to different development versions of a software project's file structure.

<u>List of commits</u>**:** when someone makes a list of changes to files in a branch and he wants to record these changes he needs to create a snapshot which is created using *Git* commit command. Individual changes can be grouped to separate commits where each of them contains a text describing the commit, the commit date and also the author of the commit.

Using the information about the files that changed in the same commit we can compute *co-changed files* for this specific commit. Therefore, using the term *co-changed files,* we refer to two files that change in the same commit.

Difference between two commits**:** we can also extract the difference between two different versions of the same file. This way we can detect all the new changes which have been introduced by a new commit.

Using the information from the difference between two different versions of the same file we can also detect *source comments* added, removed or changed in the source code of the file that appears in a newest commit. Examining the *source comments* that are changed from a version to another, someone may be in position of spotting useful information about the purpose of the changes.

List of releases: using *Git* commands we can also extract the releases for a specific repository. Like most of version control systems, *Git* has the ability to tag specific points in history as being important. Typically, people use this feature to mark release points. We remind the reader that a software release is the process of launching a new version of the software publicly available.

List of bug/issues: *Git* does not support issue / bug tracking by its own but most of the projects are using external systems for this job such as *Redmine*[3], *Bugzilla*[4], *JIRA*[5] and *Github*. Issue tracking systems are useful for organizing different kinds of issues like bugs and tasks in a software development process.

List of builds: most of the projects also use an external service for testing. Using these services someone can try to build a specific project from a *Git* repository

[3] http://www.redmine.org/

[4] https://www.bugzilla.org/

[5] https://www.atlassian.com/software/jira

and check if the build was successful or not. There are various services in the web for this kind of job, such as *TravisCI*[6] and *Coveralls*[7].

Code Review Systems: these types of systems provide a systematic examination of the source code of a project. A code review system can be used in a software development process for finding mistakes, vulnerabilities or bugs before the official release is published. Examples of these systems are *Gerrit*[8], *Codacy*[9] and *Crucible*[10].

Developer Social aspect: services like *Github* have integrated social features for developers. Specifically, developers are able to *follow* other developers and to *watch* projects of their choice. Using this information, it is easy for someone to extract the relation between the developers as well as their interests.

3.2 Proof of Concept: experimental setup

The purpose of the first part was to list all the different sources which are usually used in a development lifecycle and can be extracted from open source software repositories. In this section, we start with the description of the datasets which are used in this thesis and then we move on presenting the methodology for retrieving the data from the selected repositories. We divided our retrieval process in two parts: (a) retrieval of the contents of *Github-located* information and (b) retrieval using external sources *API*.

[6] https://travis-ci.org/

[7] https://coveralls.io/

[8] https://www.gerritcodereview.com/

[9] https://www.codacy.com/

[10] https://www.atlassian.com/software/crucible

3.2.1 Datasets

In this study we have collected data from six open source-projects. BioSQL[11] is a generic relational model covering sequences, features, sequence and feature annotation, a reference taxonomy, and ontologies from various sources such as GenBank or Swissport. Ensembl[12] is a joint scientific project between the European Bioinformatics Institute (EBI) and the Wellcome Trust Sanger Institute (WTSI). The goal of Ensembl is to automatically annotate the three billion base pairs of sequences of the genome, integrate this annotation with other available biological data and make all this publicly available via the web. MediaWiki[13] was first introduced in early 2002 by the Wikimedia Foundation along with Wikipedia, and hosts Wikipedia's content since then. OpenCart[14] is an open source eCommerce platform. PhpBB[15] is a free flat-bulletin board software solution. TYPO3[16] is a free and open source web content management framework.

3.2.2 Retrieval of the contents of Github-located information

We collected our data during February 2017. For all the projects we focused on the master branch and on commits where the file holding the database schema appears. Then, from each of those commits we extracted:

1. Text describing the commit
2. Author of the commit

[11] http://www.biosql.org/wiki/Main Page

[12] http://atlas.web.cern.ch/Atlas/Collaboration/

[13] https://www.mediawiki.org/wiki/MediaWiki

[14] http://www.opencart.com

[15] https://www.phpbb.com

[16] http://typo3.org/

3. Date of the commit

4. Contents of the SQL file containing the definition of the database schema

We saved the different versions of the file containing the database schema in separate files, one file per version, using the UNIX timestamp of the commit as filename.

In addition, we got the difference between every two consecutive commits in history and extracted the comments added in the new version of SQL source code.

We also retrieved, from every commit, the names of the files which are modified together with the database schema.

Finally, for all the projects, the *Git* tags referring to the whole history were gathered.

3.2.3 Retrieval using external sources API

This part of the retrieval was challenging because we had to manually search every repository to identify the external systems that are used by the project. After this process, we came with a list of external systems for every repository. Some of these systems such as *Github*, *JIRA* and *TravisCI* have an *API* which can be used to retrieve the data from them and some of them such as *Redmine* and *JIRA* provide a graphical interface from which someone can manually download the tasks / bugs.

After our research on the 6 datasets we observed that three issue / bug tracking systems are used: *Github*, *Redmine* and *JIRA*. Specifically, BioSQL and Typo3 use *Redmine,* Opencart uses *Github* and PhpBB use *JIRA* for issue tracking. In order to retrieve the data from *Github* project we used its API, for gathering the data from *Redmine* we used its graphical interface and for JIRA we used both its API and graphical interface. It is worth mentioning that we gathered issues for the whole history of the projects.

In addition we used TravisCI API to retrieve the build information for the whole history of the projects that use external system for testing.

Table 1 presents the datasets along with the external systems that they use. However, the full list of sources for each dataset is shown in Appendix 1.

Dataset	**Repository URL**	**Releases**	**Issues**	**Builds**
Biosql	https://github.com/biosql/biosql/	Git	Redmine	-
Ensembl	https://github.com/ensembl/ensembl/	Git	-	Travis CI
Mediawiki	https://github.com/wikimedia/mediawiki	Git	-	Travis CI
Opencart	https://github.com/opencart/opencart/	Git	Github	-
PhpBB	https://github.com/phpbb/phpbb/	Git	JIRA	Travis CI
Typo3	https://github.com/typo3/typo3.cms/	Git	Redmine	Travis CI

Table 1 Repositories with their urls and the external systems that they use

3.2.4 Data preprocess

We created an ETL workflow to transform and load the data into a SQLite database. For the transformations we used Pentaho[17] data integration tool.

Connect Builds with commits

The linking between build and commits is very easy because *TravisCI* provides the commit id for every build. Every build has a unique identifier across the project which refers to a commit. We observed that from a total of 1573 commits that belong to six different projects, only 20 commits have information about a build, while the total number of builds which are referring to the six datasets are 50297. Therefore, we conclude that when a change in the schema of the database takes place developers do not build the project. This is reasonable because in most cases, changes must be done in the source code of the project that uses the database first and then test and build the project.

Linking bugs with commits

Links between bugs and commits are not easy to be found. Bird et al. [BBAD09] showed that linked bug fixes can sometimes be found, amongst commits in a code repository by pattern-matching but all the bug-fixing commits cannot be identified without extensive, costly and post-hoc effort. In addition, Bachmann et al [BBRD10] found that the bug tracking systems may be biased because not all bugs are reported through those systems but also some bugs are reported in mailing lists for example.

In order to link bugs with commits we made the following simple assumption: for every bug which has three different dates (*date created*, *date updated* and *date closed*) we keep two pointers for each of those dates. These two pointers point to the next and the previous chronologically commits respectively. An example of this method is shown in Figure 2. A bug which

[17] http://www.pentaho.com/

was created chronologically between *commit*$_1$ and *commit*$_2$ will have *prev_created* pointer set to the *id* of *commit*$_1$ and *next_created* pointer set to the *id* of commit$_2$. The pointers for *update* and *close* event are set in a similar manner.

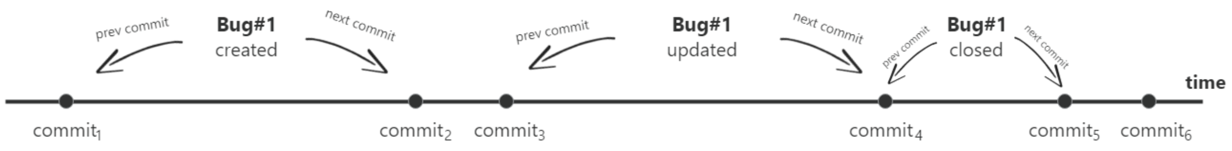


Figure 2 Graphical representation of the method used for linking a bug with the corresponding commits.

Linking releases with commits

Every release in *Git* and every commit have both dates. We link every commit to the previous and the next release using the date information. Specifically, we assign as previous release to every commit, the release with the most recent date which is smaller than the date of the commit and as next release, the oldest release which its date is more recent than the date of the commit.

3.2.5 Changes to Database Schema - Hecate

In order to retrieve the changes in the schema of the database, the files containing the database schema were processed in sequential pairs from Hecate[18], to give us in an automated way the differences between two subsequent commits. Hecate is a tool which detects changes at the attribute level and changes at the relation level. Attributes are marked as altered if they exist in both versions and their type or participation in their table's primary key changed. Tables are marked as altered if they exist in both versions and their contents have changed (attributes inserted/deleted/altered).

[18] https://github.com/DAINTINESS-Group/Hecate

In Table 2 some representative stats for each dataset, based on the retrieved data, are shown. The stats for *#Branches*, *#Releases*, *#Issues* and *#Builds* refer to the whole project and the stats for *#Commits* and *#Developers* refer only to the commits and developers that affect the schema of the database.

Dataset	**#Branches**	**#Commits**	**#Releases**	**Start/end date**	**#Issues**	**#Builds**	**#Developers**
Biosql	6	47	18	2002/01/28 – 2017/02/03	13	0	6
Ensembl	100	527	247	1999-10-10 - 2017/02/03	0	1164	53
Mediawiki	42	411	295	2003-04-14 - 2017/02/03	0	16430	80
Opencart	5	412	30	2009-02-11 - 2017/02/03	14087	0	69
Phpbb	13	230	134	2002-07-16 - 2017/02/03	17388	19344	23
Typo3	26	98	422	2003-10-03 - 2017/02/03	1184	13359	39

Table 2 Stats for each dataset from the retrieved data

CHAPTER 4.

A REFERENCE MODEL FOR THE BIOGRAPHY OF A SCHEMA

In this chapter we define a reference model that can be used to host the available information for FOSS projects as described in detail in Chapter 3. First, we analyze why we need a reference model and then we describe in detail every aspect of the model.

We define our reference model in order to support the problem of mining the software evolution. The reference model will help us answer one of our initial research questions regarding the generation of a biography which highlights the reasons of important actions out of the heterogeneous various, dissimilar sources of info.

In order to include all the needed functionality, we define four major levels for our reference model: (a) software evolution level, (b) schema evolution level, (c) explanation of why / motivational level and (d) purpose / summary level.

4.1 Software Evolution Level

In this level we define two sets of concepts. The first one is used to represent the structure of the software project in each version of the system's history. The second one is used to represent the changes that affect the project structure in each version. The relation between the two inner layers of our model is shown in Figure 3.

4.1.1 Project Structure Level

This level contains the basic elements of a software project. The concepts that compose this level are useful for the representation of the project's structure combined with the version history of a project.

Repository: this concept represents the central element in the software project and it contains all the files of the project. It consists of branches which are different development versions of the project's source code which is usually cloned from the master branch. In addition, a repository can have different versions which are created when a new commit takes place.

Repository version: a snapshot of the repository's structure. When a commit takes place, it may introduce a change in the repository's folder structure. Thus, this element consists of a tree of modules for every different snapshot of the repository.

Tree of modules: an entity that represents the structure of the repository in a hierarchical manner. Specifically, a tree of modules holds the hierarchy of the system's modules and how they are connected. The hierarchical representation of different modules of the repository can be a graph, a tree, or any other structure.

Module: a part of the repository that implements a particular functionality. Every module can contain a list of submodules which are in fact modules appeared lower in the hierarchy. A module can be a whole folder or a single file and it can provide a list of module parts.

Module part: a module part can be a small part of a module that provides a single functionality. An example of a module part is a method or attribute if we are talking about an object-oriented class or a table if we are talking about database schema.

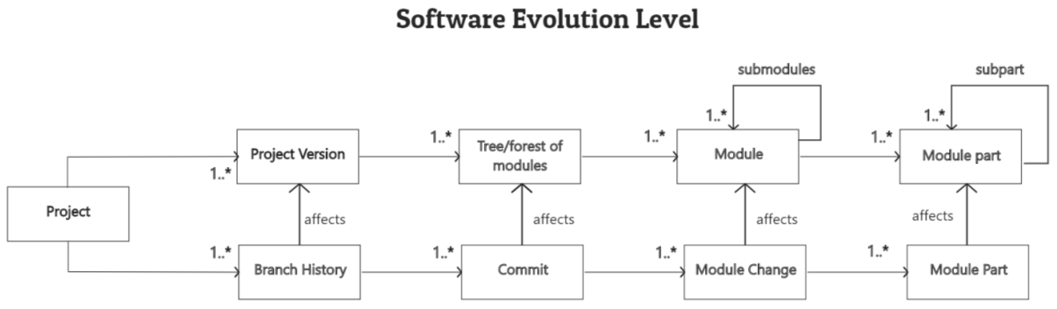


Figure 3 Software evolution level

4.1.2 Software Evolution Level

The concepts of this level are defined for modeling the evolution of a project and its components. The evolution of a software project is characterized by (a) different development branches and (b) a series of change events that affect one or more module and module parts of the system. For this reason, we define the following four basic concepts that represent the key functionality of this level.

Branch History: constitutes a version history of a development branch. Every branch history contains a list of commits where each of them may introduce new change events.

Commit: corresponds to a single commit in the version history of a repository. It contains a message written by a user and a timestamp. Every commit can affect more than one file and can also change the repository's folder structure. In addition the same commit can be in more than one branch. Therefore, based on the above, a commit consists of a list of module changes.

Module Change: corresponds to a change that affects a repository's module. This change can affect a whole folder or a single file. Every module change contains a list of module part changes.

Module Part Change: changes at a single file which is affected from a commit. These changes can be lines of code added, modified, deleted or any other type of change one can think of.

4.2 Schema Evolution Level

A database is frequently an integral part of a FOSS software project, so as the software evolves so does the schema of the database. During the project's lifetime the schema of the database is changing in a manner similar to every other module on the version history of the project. This level is a subset of software evolution level because schema evolution constitutes only a part of a project's history. In this level we define two sets of concepts. The first one is used to represent the structure of the database schema in each version of the system's history. The second one is used to represent the changes affected the database schema in each version. The relation between the two inner layers of our model is shown in Figure 4.

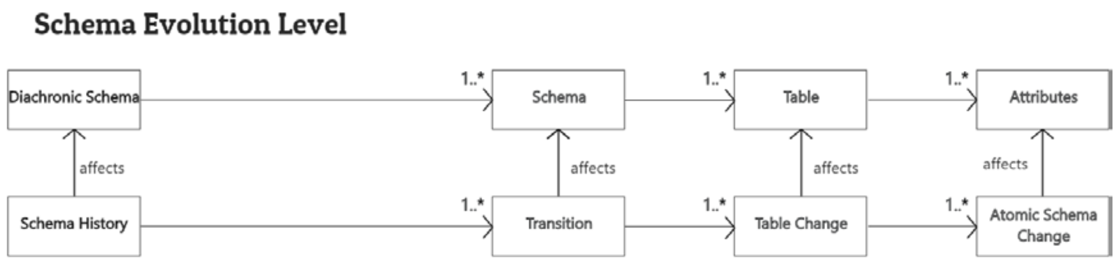


Figure 4 Schema evolution level

4.2.1 Database structure

The concepts which are part of this level represent the structure of a database schema in the evolution history of a system. This level is a subset of the *project structure* level and it consists of four basic concepts which are connected to the concepts of the project structure. More specifically, *diachronic schema* corresponds to *repository version, schema* corresponds to *tree of modules, table* corresponds to *module* and *attributes* correspond to *module part.*

Diachronic schema: contains all the versions of a database schema for a specific development branch.

Schema: corresponds to a database schema of the system. More specifically, this concept contains a list of tables as well as their attributes and their constraints.

Table: a single database table which contains the information which is connected to a table such as a list of attributes and a list of constraints.

Attribute: corresponds to a single database field and holds information for its type, name and constraints.

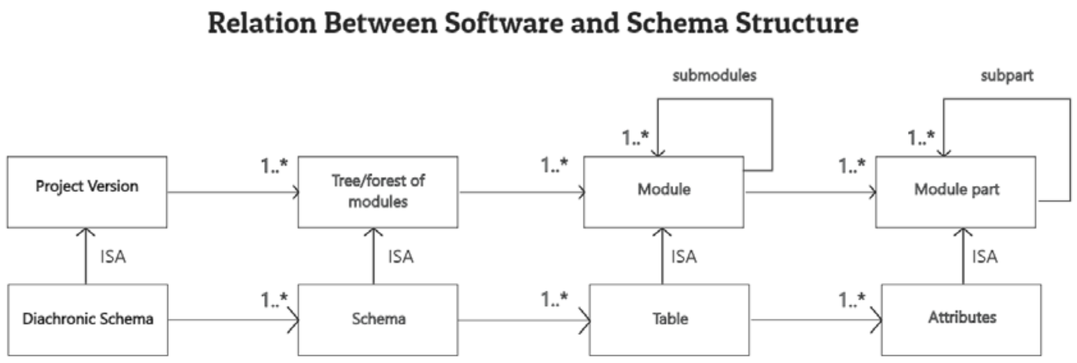


Figure 5 Relation between software structure and database schema structure levels

4.2.2 Schema evolution

The concepts of this level are used to model the evolution history of a database schema. The history of a database schema is composed of *module changes* where in this level, we refer to them as *transitions*. Every module change contains a list of *module parts changes* where in this level we refer to them as *table changes*. In addition, every table change contains a list of change events (we refer to them as *atomic change events*) which corresponds to changes that affect *module parts* placed lower in the hierarchy. To clarify the above sentences better, we present four basic concepts on this level which model the evolution of a database schema.

Schema history: contains a list of grouped change events which affect the schema of the database. When we refer to *transition*, we refer to one group of those change events.

Transition: contains the whole information about a transition from a database schema version v_i to a database schema version v_j, where $i < j$. Every transition may contain information about which tables are affected in the specific transition. In addition, every transition can have additional metrics and statistics.

Table change: contains the name of the table whose changes are kept, and a list of these changes named *Atomic Change Events*.

Atomic change event: any individual change that affects the database schema. An *atomic change event* describes a change that took place in a database table. Specifically, we distinguish the types of changes in the following categories:

- Attribute insertion at pre-existing table
- Attribute insertion at new table
- Attribute deletion at table without deleting the whole table
- Attribute deletion at table deletion
- Table insertion
- Table deletion
- Attribute type change
- Primary key change

Summarizing the above, a *transition* from schema evolution level can be mapped to a *commit.* Hence, we can use all the metadata from a commit to enrich the information of every *transition*. In addition, *schema history* is also a subset of *branch history*.

In this point it's worth mentioning that commits and therefore transitions are subset of the total commits of the project.

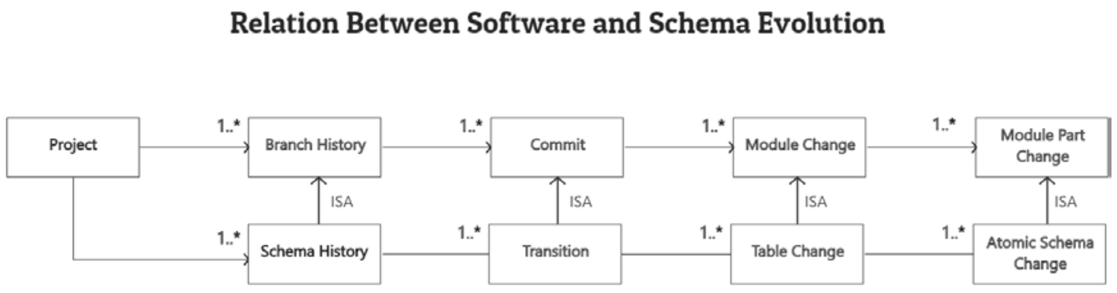


Figure 6 Relation between software evolution and database schema evolution levels

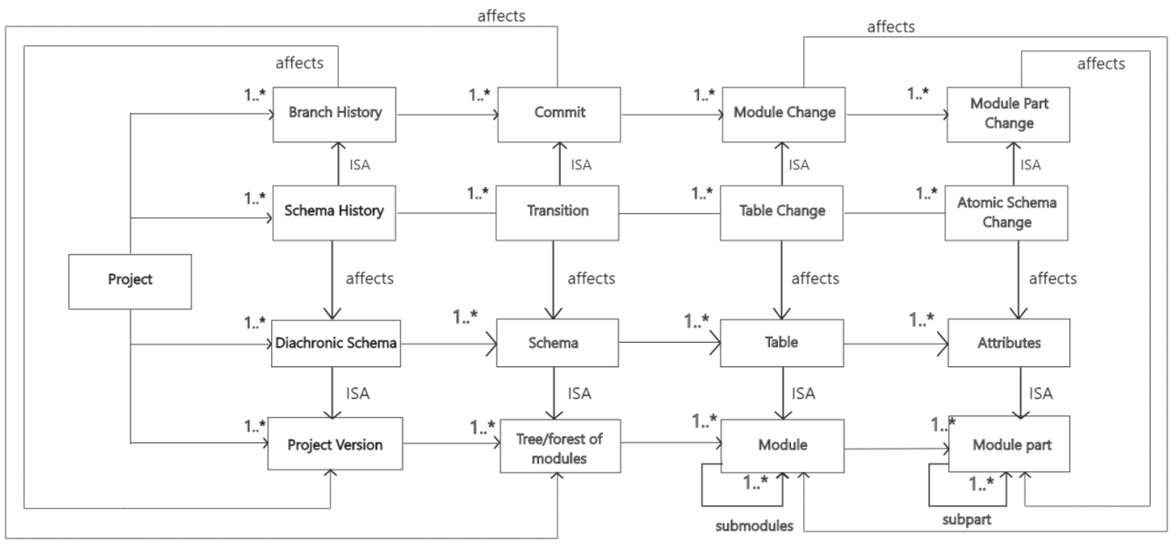


Figure 7 Relation between different concepts from schema evolution level concepts and software evolution level

4.3 Explanation of why / Motivational level

Every module change in a software development project is the result of a decision taken from a group of people involved in the project. Therefore, every module change in the version history of a project is characterized by a *reason* and maybe a *motivation*. In order to capture the decisions, the motivations and the reasons of change events as well as the people involved in those, we define two levels of concepts. The *contextual level* contains concepts which are required by every version control system and every version history of a software project accommodates. Specifically, the concepts of *contextual level* represent the explanation of *why* a change has taken place, *when* and by *whom*. The *external systems level* contains concepts which are optional in the development process but they are often used to improve the development experience. In essence, the concepts that belong to *external systems level* form a basic model for the representation of external systems used in a development environment such as issue tracking systems or project management systems.

4.3.1 Contextual level

This level contains the elements that create the *WWW* (*When, Who, Why*) three-dimensional space. Moreover, this level is directly connected with *project structure level*, the s*oftware evolution level* and s*chema evolution level*. This is because the elements which are part of this level contain useful metadata in order to explain the nature of the detected changes.

Timestamp: this element defines the time where an event took place. *Timestamps* can be measured in human time or in a sequential version id. This concept is responsible for modeling the *When* dimension.

Commit text: text describing the reason of the commit. Every commit text is written by a contributor and it is used to model the *Why* dimension.

Contributor: an individual user who is affiliated with specific events in a repository. For example, a contributor may be affiliated with a commit, a change event or an issue. This concept is used for modelling the *Who* dimension.

4.3.2 External systems level

This level is optional and it consists of the simplest possible model for the representation of any external system which is used in the development life cycle. We define three abstract concepts on this level.

External system: corresponds to any external system that is used to improve the experience of the software development process. Examples of such systems are (a) issue tracking systems, (b) code review systems, (c) build systems, (d) project management systems and more. In addition, every external system contains a list of *postings* which are created by the people involved in the software development project. Therefore, these systems can be connected to the *WWW* three-dimensional space.

Examples of such external systems are shown in the following list:

- *Issue system:* a system that tracks all the reported issues for a repository. This concept contains a list of issues that are registered to the system by different users.
- *Code Review System:* a system which helps the examination of a project's source code through the process of software development. This concept provides the *Why*, the *When* and *Who* and therefore can be connected to the *WWW* three-dimensional space.

- *Build System:* a system which provides the functionality of building and testing a project.

Posting: an element in the external system. A posting may refer to a specific development branch or even a specific module on a specific development branch. It contains a list of *post entities*.

- *Issue:* An element in issue tracker in a bug system.
- *Build:* contains useful information about a build which run after the commit. This information can the status of the build (if the build was successful or not) as well as time started, the duration of build etc.

Post entity: corresponds to an element which contains all the information about a single external event. An external event can be one reported issue, one build or any other external event one can think of. Examples of post entities are shown in the following list:

- *Issue entity:* an issue in general contains a text description, a timestamp, a status (open, closed) and a category (task, feature, bug and more). Depending on the level of detail, someone can also use additional information such as assignees, labels, due date, priority, percentage done and more. The basic elements of this concept provide the *Why*, the *When* and the *Who* and depending on the level of details someone can discover more insights into projects issues and how these arise from the project's evolution.
- *Build entity:* contains information such as the status of the build (if the build was successful or not) as well as time started, the duration of build etc. This concept provides the *Why* and then *When*.

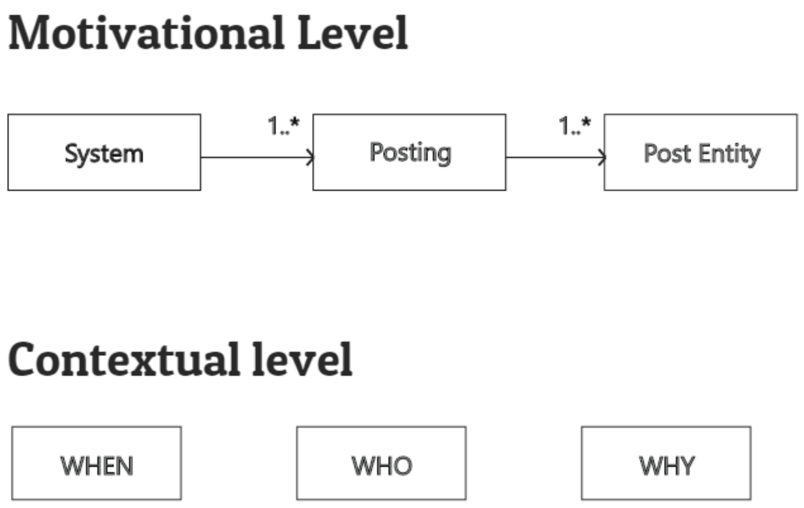


Figure 8 Explanation of why / Motivational level

4.4 Purpose / Summary Level

The concepts that compose this level are useful for generating the biography of a schema but before we define these concepts, we must first determine what characterizes a great schema biography. An interesting schema biography must be characterized by the following properties:

- The entire lifetime of a database schema must be separated into phases where each of them contains a series of events. Organizing events in phases can be very helpful to understand the general evolution of the system. Focusing on development phases of a project may provide insights regarding the development goals for the specific time period.
- A biography must have periods of highlighted events, where something interesting or worth mentioning has happened. Some change events in a specific period are more interesting than others and focusing on the events that matter can help us get rid of any kind of noise.
- A biography must have some visuals that help us understand the reasons of the evolution and maybe some metrics and statistics that prove those reasons.

Therefore, based on the above preferred properties we define the concepts of this level.

Schema Biography: constitutes the basic structure which in fact links all the different concepts of this level. This concept consists of phases and phase highlights.

Phase: a distinct time period in the evolution history of a module. A phase consists of a sequential list of project's versions. In this thesis, we consider release equivalent to phase but in general case a phase provides better abstraction than a release.

> *Release:* a distinct time period that contains a list of commits. The start of a release refers to the date where the release is first introduced in the system and the end of the release refers to the date where the next chronologically release tag is introduced.

Phase highlights: a series of change events which is a subset of a distinct phase. These events are chosen instead of others based on a scoring function which calculates the importance of each event.

Transition summary: contains a list of *transitions* which are combined based on a summary generation algorithm.

Transition highlights: a series of *transitions* which are marked as important based on a scoring function.

Metrics and statistics: any kind of metric and statistic that can be calculated in order to explain the reason of a detected change.

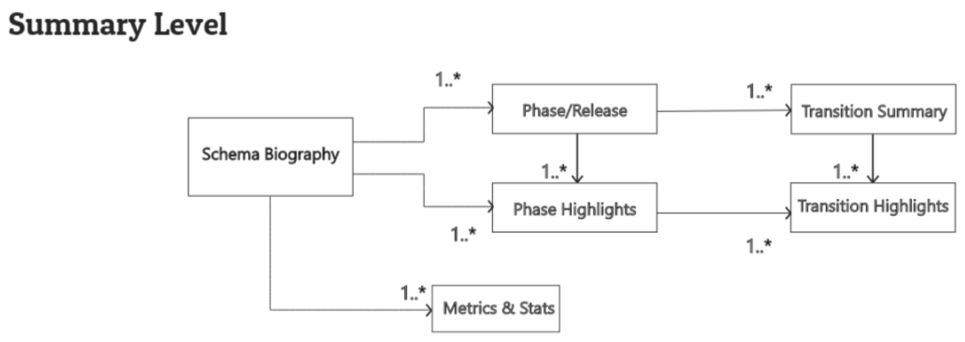


Figure 9 Summary level

Based on the above, the meta-model used in this thesis is presented in Figure 10.

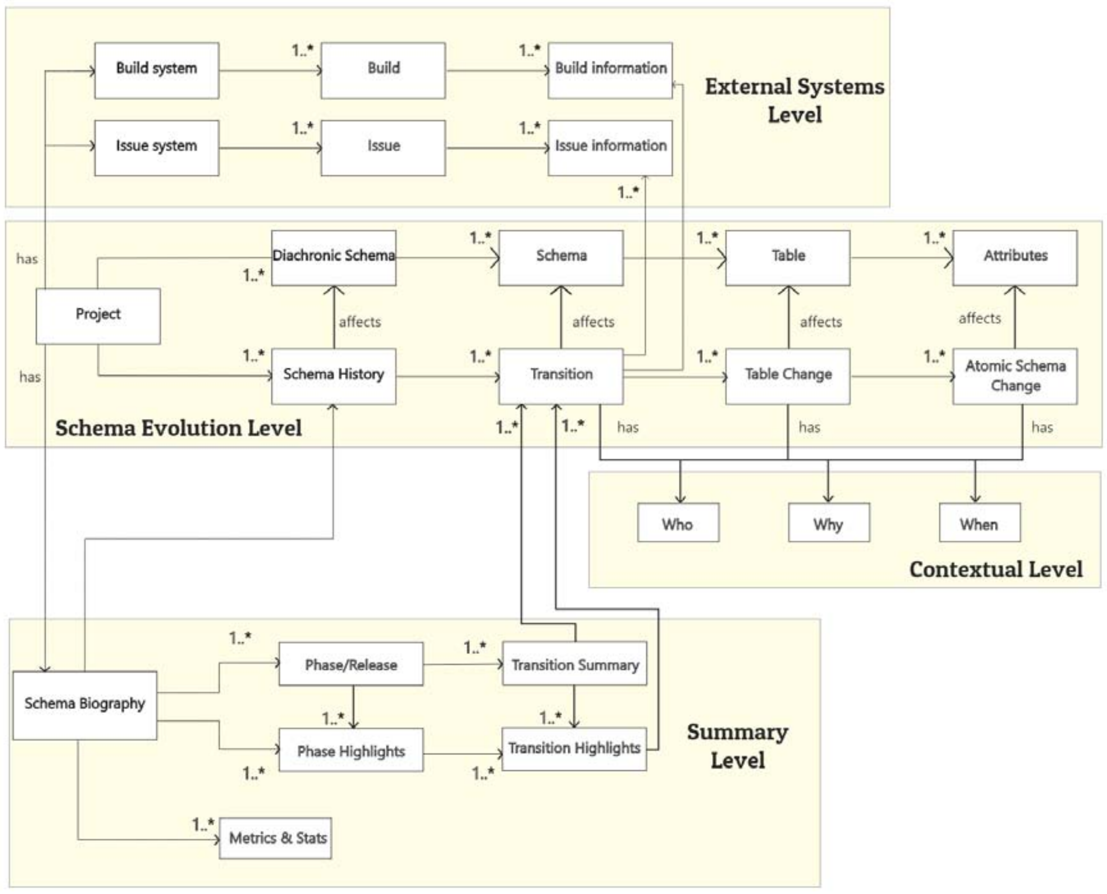


Figure 10 Model used in this study

CHAPTER 5.

INTERACTIVE ANALYSIS OF SCHEMA HISTORIES

After creating our meta-model we need to provide a tool which visualizes and helps the user to use the defined meta-model in order to interactively navigate and explore the story of the version history of database schemata. For this reason, we created a web-based application that provides the user a variety of tools to explore the version histories of schemata.

5.1 Technologies used

In order to create the web application that provides the interactive story-telling, we use state-of-the-art frameworks and techniques that help us keep the application maintainable and extensible. The main programming languages that are used are HTML, CSS and Typescript. HTML is the standard markup language for web pages and can be visually enhanced using CSS. Typescript is a typed superset of Javascript that compiles to plain Javascript. Typescript was developed and maintained by Microsoft[19] and adds optional static typing and class-based object-oriented programming to the language. We chose to use Typescript because we think that the feature of using an object-oriented style makes the source code clean, maintainable and extensible. Beyond these three programming languages, we also used some auxiliary frameworks that improve the development experience. The most important frameworks that are used for this system are presented below.

5.1.1 Back-end

Node.js[20] is an open source, cross-platform JavaScript runtime environment. Node.js through its runtime environment provides the ability to execute Javascript code on the server-side. In addition, it uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Also, Node.js provides a package ecosystem which is one of the largest ecosystems of open source libraries in the world. Some of the basic reasons that Node.js was chosen as the lead framework for the back-end are (a) the fact that Node.js unifies the web application development around a single programming language (which is Typescript), (b) the event-driven architecture which is capable of asynchronous I/O operations that aim to optimize the scalability of the application and (c) the large ecosystem of open source packages and extensive documentation that can be used.

SQLite[21] is a library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. We used SQLite to store all the information that we gathered from the version history of the projects which is described in Chapter 3. The basic reason that SQLite was chosen is

[19] https://www.typescriptlang.org/

[20] https://nodejs.org/en/

[21] https://www.sqlite.org/

because it is very light and portable as it does not have a separate server process.

5.1.2 Middle-end

Express [22] is a minimal and flexible Node.js web application framework which provides a set of features for web applications. Our goal was to create a web application which is easy to be built and run across different machines without the need of setting up external HTTP server like Apache[23]. For this reason, we used Express and Node.js to create the HTTP server and to handle the routing and the middleware level of the application. In general, the routing level is used to identify the resource which is requested from an HTTP request and for invoking the middleware's functions that can either (a) execute any code, (b) make any changes to the request, (c) call the next middleware in the stack, or, (d) end the request-response cycle. The reason that we chose express is because it is the standard server framework for Node.js.

5.1.3 Front-end

AngularJS[24] is a Javascript-based open-source front-end web application framework. AngularJS is usually used for developing single-page applications and provides a framework for client-side, model-view-controller (MVC) architectures. We used AngularJS in order to create our front-end part of the application, which provides all the necessary functionality for interaction with the user.

[22] https://expressjs.com/

[23] https://www.apache.org/

[24] https://angular.io/

Bootstrap[25] is one of the most popular HTML, CSS and JS framework for developing responsive applications. It contains design templates and graphical elements for HTML and CSS such as buttons, navigations and forms. All these design templates are reusable and are designed to work nice to all screen sizes and devices. Therefore, we use Bootstrap for creating the front-end layout of the application.

Concerning the visualization of the different types of charts, we used **D3.js**[26]. D3.js is a Javascript library that allows the binding of arbitrary data to a Document Object Model (DOM), and supports data-driven transformations to the document. We use this library in order to create interactive SVG charts that provide useful insights on the evolution of database schemata.

5.2 Architecture

The whole application is built using the model-view-controller (MVC) architectural pattern. MVC architecture divides a given application into three interconnected parts in order to separate internal representation of the information allowing the efficient code reuse. Specifically, the project structure is separated into three large modules. The first module (*models module*) is responsible for implementing the concepts of the reference model, which is presented in Chapter 4, along with the database controllers that feed those concepts with data from the database. The second module (*controllers module*) is responsible for handling the HTTP requests making the resources of the application available to the client. Finally, the third module (*views modules*) is responsible for the interaction with the user.

[25] http://getbootstrap.com/

[26] https://d3js.org/

5.2.1 *Models* Module

The business logic of the application is implemented in this module. This module is composed of three individual parts: (a) databases and database handlers, (b) data structures that hold the meta-model which is described in detail in Chapter 4, and, (c) data enrichment modules.

Databases and database handlers. In this part, the files holding the database of the system are located. The SQLite database of the system is loaded with the pre-processed data that was gathered with the method described in Chapter 3. Moreover, in this part of the system, a database controller is implemented for every concept of the meta-model. Every database handler is responsible for retrieving the data from the database, populate the data structures with the data and return the result.

Data structures implementing the meta-model: This part contains the concepts that are defined in each level of the meta-model as a data structures.

Data enrichment modules: This subpart of the system includes different modules that are used in order to enrich the raw data that was gathered with useful metrics and statistics. This sub-system, provides modules for automatic text generation based on the descriptive statistics for each release and for each commit. Moreover, in this part of the system, rule-based techniques are implemented for the characterization of commits and releases based on the change events that take place in any of them. More details for release and commit characterization are given in Chapter 6.

5.2.2 Controllers Module

This module is responsible for handling all the HTTP requests. Moreover, this module provides a RESTful web service using the HTTP protocol that provides access to application's resources. REST stands for Representational State Transfer and it is a web-standards-based architecture[27]. Therefore, this module is the intermediate that connects the back-end which holds all the

[27] https://en.wikipedia.org/wiki/Representational_state_transfer

available information with the front-end which presents the data to the user using an interactive way.

5.2.3 Views Module

This module is responsible for handling the human and computer interaction. In fact, views module implements the front-end of the application that is built using AngularJS, Bootstrap and D3.js. It is responsible for (a) retrieving the necessary information using the RESTful API and (b) presenting the data to the user in an interactive way. A detailed description about the features of the views module is given in Section 5.3.

The general representation of how the modules of the system are connected together and also how the system's database is populated with the gathered data is shown Figure 11. Specifically, the publicly online data are retrieved using the methodology described in Chapter 3. Then, the data are pre-processed and transformed to match the schema of the system's database, where they are finally stored. After, that additional summaries and metrics are calculated and stored in the database. The data which are stored in the database are available to different kind of clients (web applications, desktop applications, HTTP request that are made from a browser) through the REST API. In addition, different clients can use the REST API to enrich the information which is stored in the database. For example, clients can make changes to automatically generated text that describes a version.

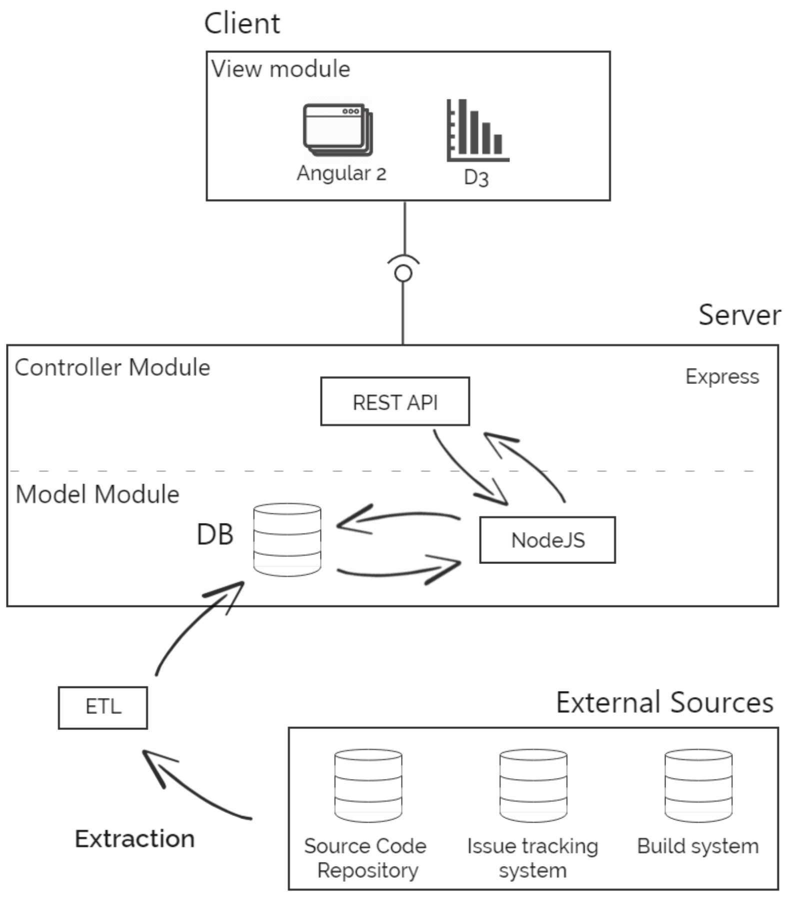


Figure 11 A general representation of how different parts of the system are connected together.

5.3 Interactive analysis

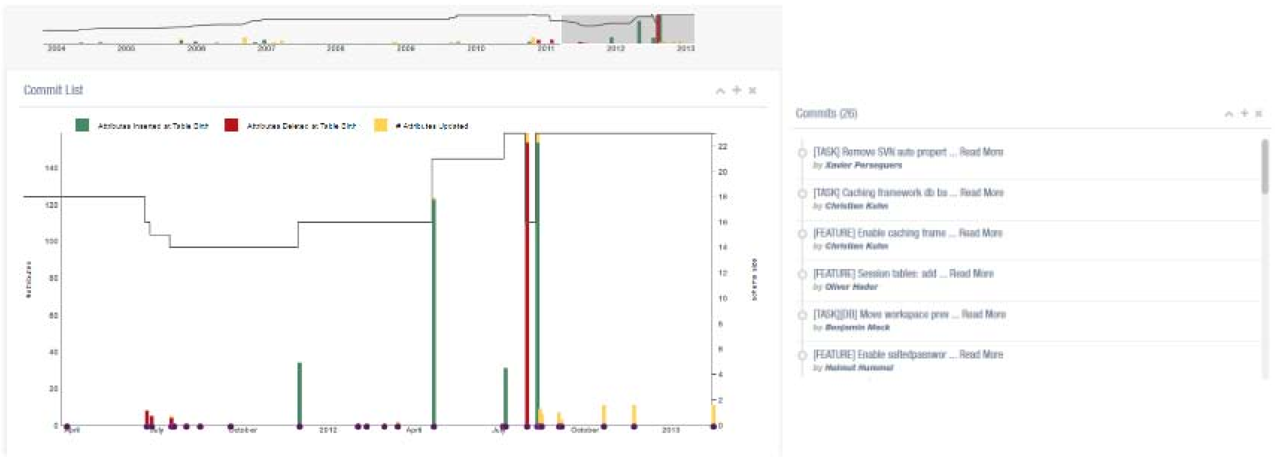
In this thesis, we decide to focus on the information gathered on releases. We treat the version history of database schema as a collection of releases, as published by the developers, with a release to contain a list of commits. Therefore, we enrich the raw data with contextual data gathered for releases, and, we create aggregate measures for each release. Then, based on the releases, we create the front-end application in which we use the Visual Information Seeking Mantra: Overview first, zoom and filter, then details-on-demand which is presented in [Shne96]. Based on this principle, we define three different levels of detail: (a) the *summary level*, (b) the *release level* and (c) the *commit level*.

5.3.1 Levels of detail

Summary level. This level of details contains the subset of releases in the schema history that include at least one commit to the schema definition file of the database of a project. Releases that have not touched the schema are omitted because they do not have statistics for our investigation and they only provide unnecessary noise. In this level some useful overall statistics are presented such as the top co-changed files along with SQL file and developers ranked based on the number commits that they made. In this level, there is the basic change breakdown and schema size chart for the releases using a stacked bar chart in combination with a line chart and a scatter plot, an example of which is shown in Figure 14. Someone could argue with the choice of using stacked bar charts because are useless for comparing series magnitudes in one or another certain category but in our case we do not use them for that reason. We decided to use a stacked bar chart because it provides the ability to simultaneously compare totals and notice sharp changes at the release level. Moreover, a stacked bar chart enables a better understanding of the big picture, which is what we are looking for in this level of detail without much focus on details such as small changes.

In addition, in this level, the user can filter out unnecessary releases and focus only on a specific time period which contains a subset of the total releases of the project. Beyond that filtering, there is the ability to drill down to the lower level named r*elease level* for a specific time period or for a specific release.

Release level. This level presents all the commits that took place inside the selected time period or in the selected release. In this level, the basic change breakdown and schema size chart (similar to the one that is presented in upper level) appears, but this time information about commits is presented on it. Also, there is a list of commits along with the date of the specific commit and the author of the commit. Figure 12 depicts an example of this level. In addition, depending on the number of releases which are selected, an automatically generated text is shown. This generated text provides a summary and the highlights of a release based on the descriptive statistics and characterizations which are explained in detail in Chapter 6. An example of a text summary for a release is shown in Figure 13.


Commit List
Attributes Inserted at Table Birth
Attributes Deleted at Table Birth
# Attributes Updated
Commits (26)

[TASK] Remove SVN auto propert ... Read More
by Xavier Perseguers

[TASK] Caching framework db ba ... Read More
by Christian Kuhn

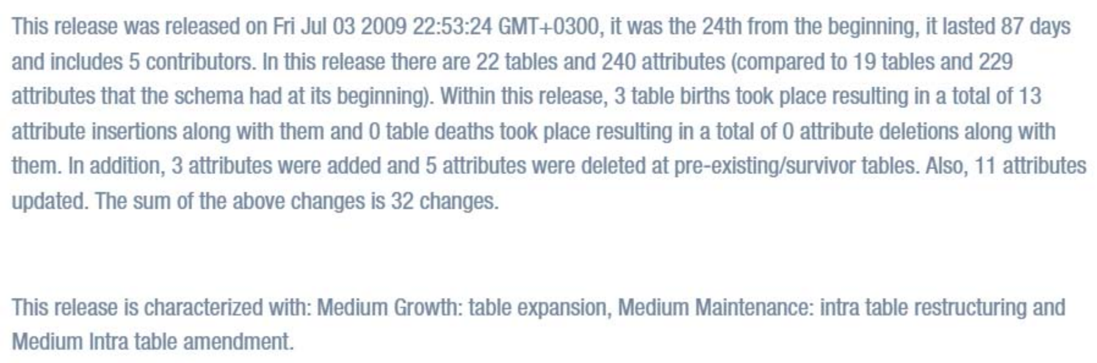
[FEATURE] Enable caching frame ... Read More
by Christian Kuhn

[FEATURE] Session tables: add ... Read More
by Oliver Hader

[TASK][DB] Move workspace prev ... Read More
by Benjamin Mack

[FEATURE] Enable saltedpasswor ... Read More
by Helmut Hummel

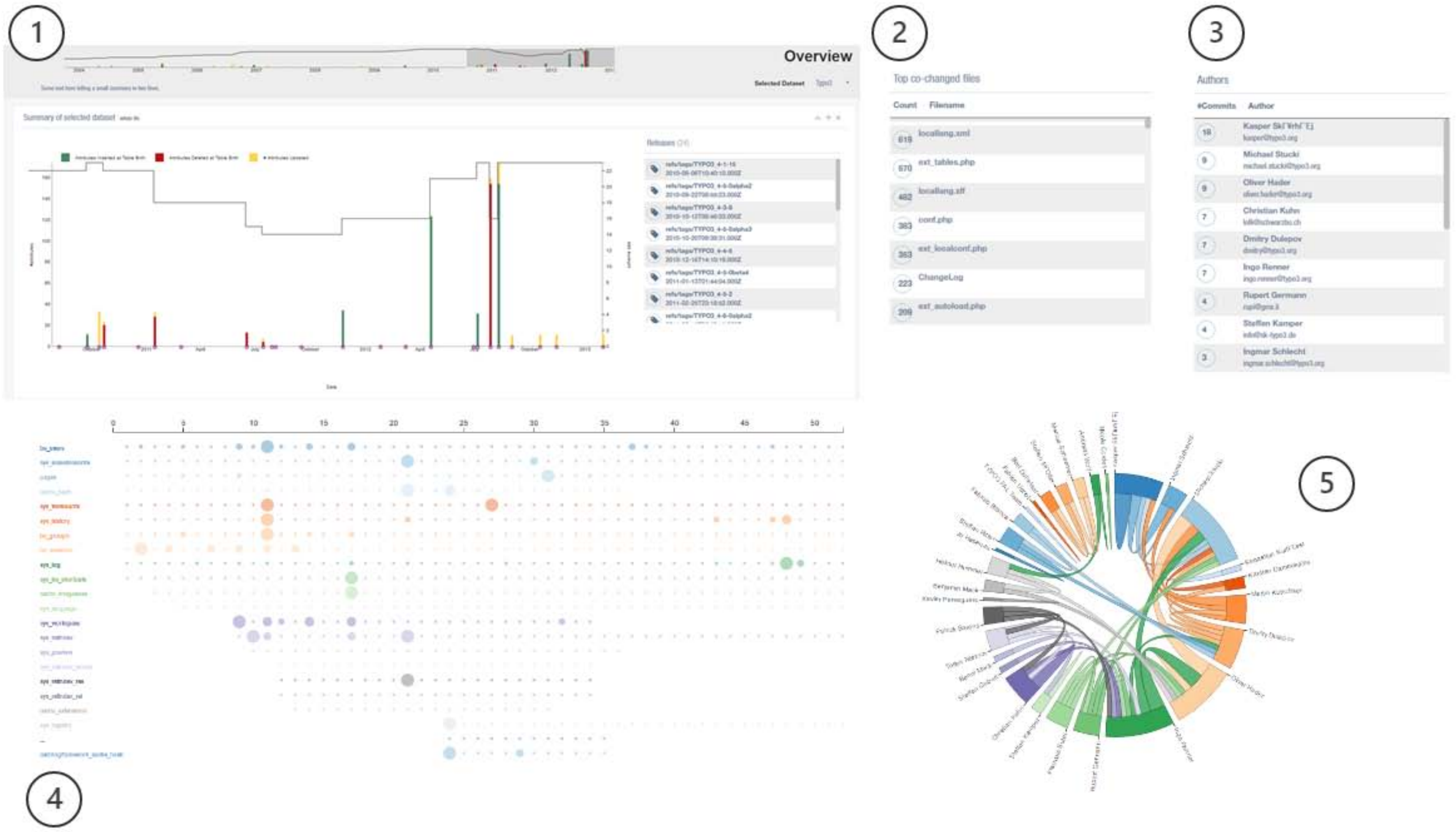
Figure 12 Graphical elements that are part of the release level



This release was released on Fri Jul 03 2009 22:53:24 GMT+0300, it was the 24th from the beginning, it lasted 87 days and includes 5 contributors. In this release there are 22 tables and 240 attributes (compared to 19 tables and 229 attributes that the schema had at its beginning). Within this release, 3 table births took place resulting in a total of 13 attribute insertions along with them and 0 table deaths took place resulting in a total of 0 attribute deletions along with them. In addition, 3 attributes were added and 5 attributes were deleted at pre-existing/survivor tables. Also, 11 attributes updated. The sum of the above changes is 32 changes.

This release is characterized with: Medium Growth: table expansion, Medium Maintenance: intra table restructuring and Medium Intra table amendment.

Figure 13 Automatically generated text for describing a release



Top co-changed files

Count	Filename
619	locallang.xml
570	ext_tables.php
482	locallang.xlf
363	conf.php
363	ext_localconf.php
223	ChangeLog
209	ext_autoload.php

Authors

#Commits	Author
18	Kasper SkI˚rhI˚Ej kasper@typo3.org
9	Michael Stucki michael.stucki@typo3.org
9	Oliver Hader oliver.hader@typo3.org
7	Christian Kuhn lolli@schwarzbu.ch
7	Dmitry Dulepov dmitry@typo3.org
7	Ingo Renner ingo.renner@typo3.org
4	Rupert Germann rupi@gmx.li
4	Steffen Kamper info@sk-typo3.de
3	Ingmar Schlecht ingmar.schlecht@typo3.org

Figure 14 Graphical elements that are part of the summary level. (1) The basic change breakdown and schema size chart which presents information about releases in a specific time period, (2) top co-changed files along with the schema definition file, (3) developers sorted by the number of commits they pushed on the repository, (4) table lives across the whole lifetime of the database schema, (5) chord diagram which presents the relations between developers (developers that make a commit in the same release)

Commit level. This level constitutes the most detailed level of the system. It presents the detailed information about a specific commit. This information includes:

- An automatically generated summary which presents the highlights of the commit.
- Some descriptive statistics like the number of tables which are added to the schema, the number of tables which are deleted from the schema and information about the changes on attributes.
- An analytical list of all the tables affected in the specific commit along with the details regarding the changes on the table attributes.
- Issues that have been reported immediately before and immediately after the specific commit.

Figure 15 presents the different graphical elements of this level.

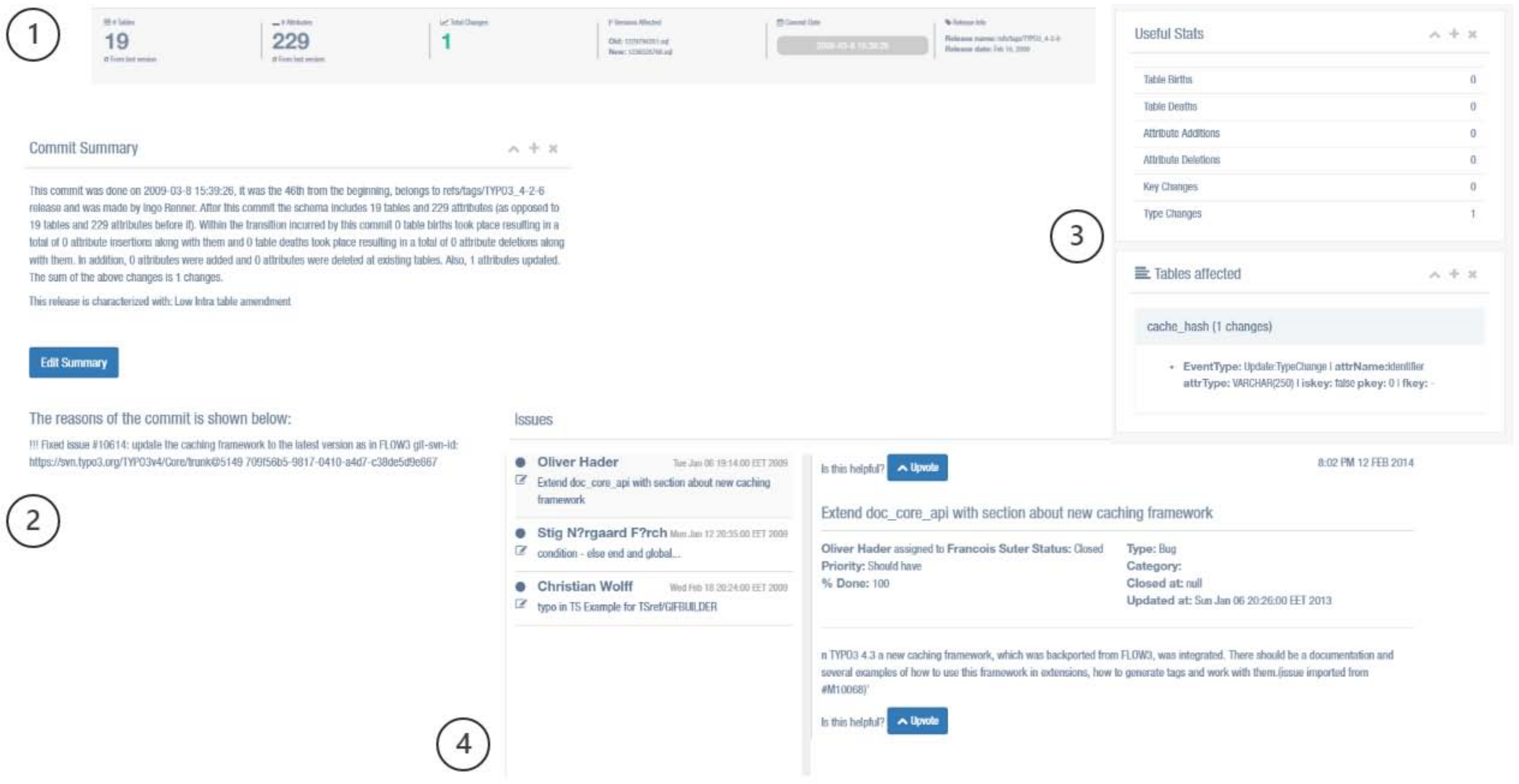
5.3.2 A common scenario on using the system

In this section we present a common scenario for exploring the history of a database schema. When the user selects a project to examine, the page containing the higher level of detail (*summary level*) is populated with the useful information that was described in the previous section.

Step 1: Zoom and filter on releases. As we mentioned earlier, the highest level of detail displays a list which contains all the releases for a specific schema history. This amount of information may be large for software projects with a long lifetime. For this reason, the user can focus only in a specific time period using the filter option. In this way, the user can explore the releases inside a time period. In addition, the user is able to drill down to a specific release or to a specific time period which may contain more than one release. In our example, we selected to examine all the releases that are published in the last three years of the Typo3 project and specifically between 2011 - 2013. The zoom option directs the user to middle level of detail (*release summary*) where useful information for the commits that took place in the selected releases are shown. An example of the filter option for the summary level is shown in Figure 16.

Step 2: Zoom and filter on commits. The release level displays all the commits that took place inside the selected releases. In our example, we selected to examine commits that took place inside releases that were published between 2011 and 2013. In this level, the user is able to focus on specific commits inside a selected time period using the filter tool in a manner similar to the filter tool for release filtering. For this scenario, we decided to focus on commits that took place in the last year of the project (from Feb 2012 to Feb 2013). Figure 17 shows the information about the commits for this time period using a chart similar to the one for releases and a list containing the author and the date of each commit. In this level we can zoom in a commit and examine all the details for this commit. In our scenario we zoom to a commit with title *"[!!!][BUGFIX] *_user table password field is to short".* After choosing a commit, the user is directed to the lowest level of detail (*commit level*) where the changes that are introduced in the selected commit are presented in detail along with an automatically generated text describing the highlights of the commit.

Step 3: Exploring the details of a commit. At the commit level, the detailed information for a selected commit is presented. In our case the details for the commit with title *"[!!!][BUGFIX] *_user table password field is to short"* are displayed. Figure 18 shows the information for the selected commit. More specifically, (1) and (3) present some statistics about the change events that took place along with the date of the commit and the release to which it belongs. In (2) the automatically generated summary for the selected commit is displayed along with the detailed message from the commit. The affected tables along with their affected attributes are shown in detail in (4). Finally, (5) presents all the issues that were reported in the issue tracking system immediately before and immediately after the commit has taken place.



Commit Summary

This commit was done on 2009-03-8 15:39:26, it was the 46th from the beginning, belongs to refs/tags/TYPO3_4-2-6 release and was made by Ingo Renner. After this commit the schema includes 19 tables and 229 attributes (as opposed to 19 tables and 229 attributes before it). Within the transition incurred by this commit 0 table births took place resulting in a total of 0 attribute insertions along with them and 0 table deaths took place resulting in a total of 0 attribute deletions along with them. In addition, 0 attributes were added and 0 attributes were deleted at existing tables. Also, 1 attributes updated. The sum of the above changes is 1 changes.

This release is characterized with: Low Intra table amendment

The reasons of the commit is shown below:

!!! Fixed issue #10614: update the caching framework to the latest version as in FLOW3 git-svn-id: https://svn.typo3.org/TYPO3v4/Core/trunk@5149 709f56b5-9817-0410-a4d7-c38de5d9e867

Useful Stats

Table Births	0
Table Deaths	0
Attribute Additions	0
Attribute Deletions	0
Key Changes	0
Type Changes	1

Tables affected

cache_hash (1 changes)

- **EventType:** Update:TypeChange | **attrName:**identifier **attrType:** VARCHAR(250) | **iskey:** false **pkey:** 0 | **fkey:** -

Issues

- **Oliver Hader** Tue Jan 06 19:14:00 EET 2009
  Extend doc_core_api with section about new caching framework
- **Stig N?rgaard F?rch** Mon Jan 12 20:35:00 EET 2009
  condition - else end and global...
- **Christian Wolff** Wed Feb 18 20:24:00 EET 2009
  typo in TS Example for TSref/GIFBUILDER

Is this helpful? Upvote 8:02 PM 12 FEB 2014

Extend doc_core_api with section about new caching framework

Oliver Hader assigned to **Francois Suter** **Status:** Closed
Priority: Should have
% Done: 100

Type: Bug
Category:
Closed at: null
Updated at: Sun Jan 06 20:26:00 EET 2013

n TYPO3 4.3 a new caching framework, which was backported from FLOW3, was integrated. There should be a documentation and several examples of how to use this framework in extensions, how to generate tags and work with them.(issue imported from #M10068)'

Is this helpful? Upvote

Figure 15 Graphical elements that are part of the commit level. (1) presents highlights and useful information for the specific commit, (2) automatically generated text summary for a specific commit, (3) useful statistics regarding the different types of changes, (4) issues that was reported immediately before and immediately after the commit.

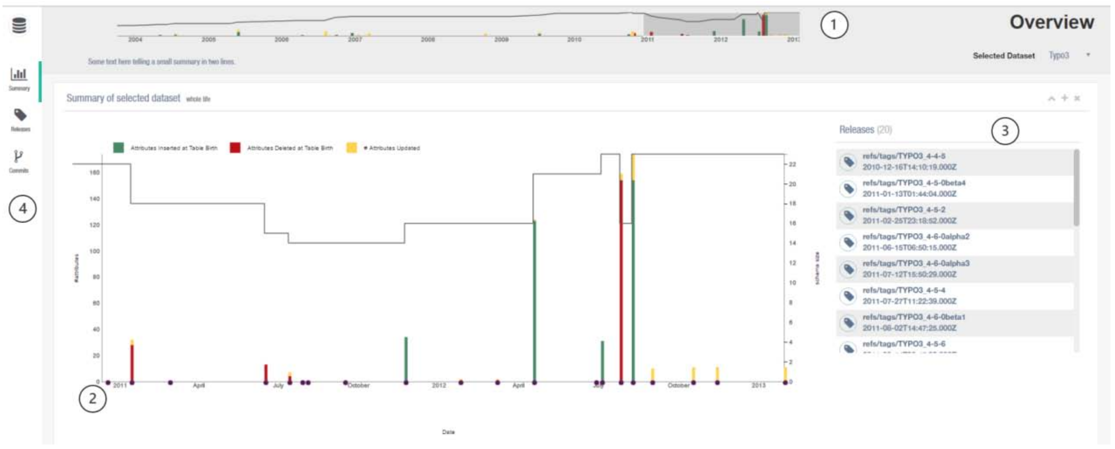

Overview
Selected Dataset Typo3
Summary of selected dataset
Releases (20)
Attributes Inserted at Table Birth
Attributes Deleted at Table Birth
# Attributes Updated
1
2
3
4

Figure 16 Filtering option in summary level. (1) tool for filtering down in a specific period, (2) change breakdown chart displaying information for the releases in the selected time period and (3) name and date for each release inside the time period, (4) menu for choosing one of the three different levels of detail

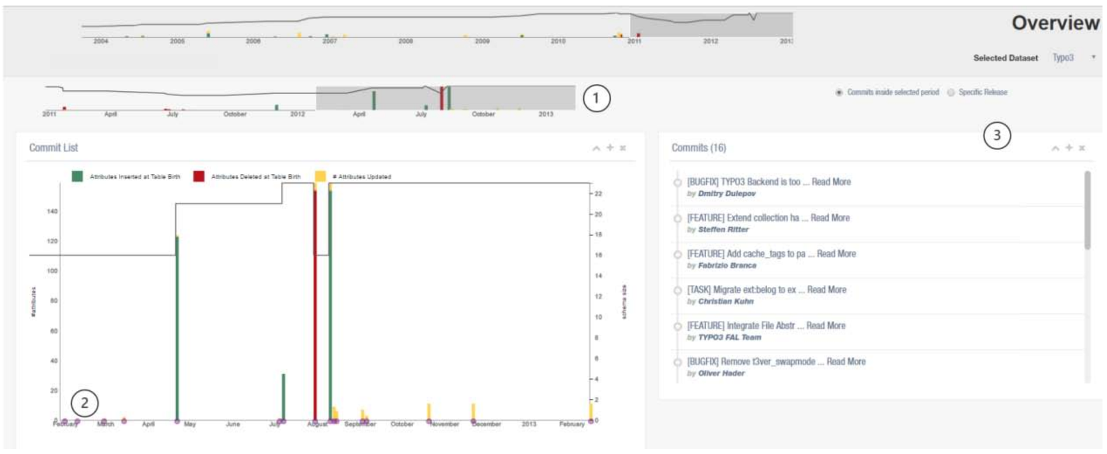
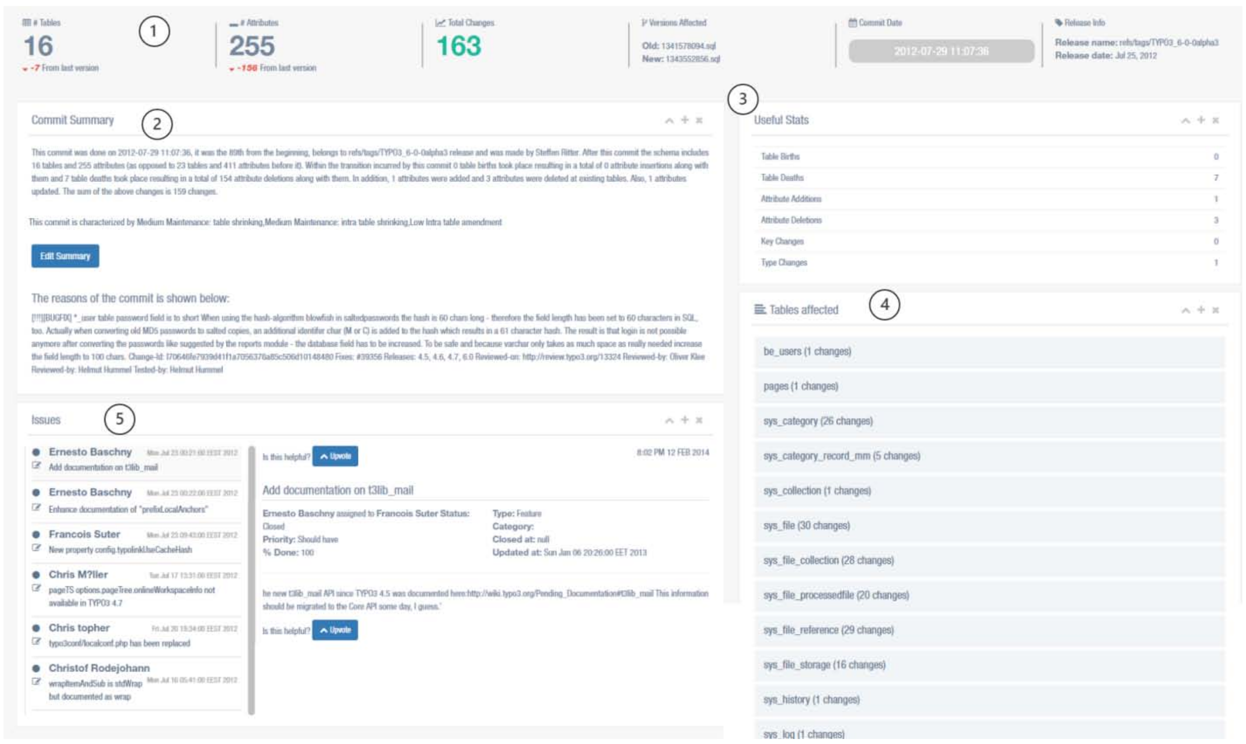


Figure 17 Filtering option in release level. (1) tool for filtering down in a specific period, (2) change breakdown chart displaying information for the commits in the selected time period and (3) for each commit we present the first 30 characters from the commit text and the author of the commit.



Tables
16
-7 From last version

Attributes
255
-156 From last version

Total Changes
163

Versions Affected
Old: 1341578094.sql
New: 1343552856.sql

Commit Date
2012-07-29 11:07:36

Release Info
Release name: refs/tags/TYPO3_6-0-0alpha3
Release date: Jul 25, 2012

Commit Summary

This commit was done on 2012-07-29 11:07:36, it was the 89th from the beginning, belongs to refs/tags/TYPO3_6-0-0alpha3 release and was made by Steffen Ritter. After this commit the schema includes 16 tables and 255 attributes (as opposed to 23 tables and 411 attributes before it). Within the transition incurred by this commit 0 table births took place resulting in a total of 0 attribute insertions along with them and 7 table deaths took place resulting in a total of 154 attribute deletions along with them. In addition, 1 attributes were added and 3 attributes were deleted at existing tables. Also, 1 attributes updated. The sum of the above changes is 159 changes.

This commit is characterized by Medium Maintenance: table shrinking,Medium Maintenance: intra table shrinking,Low Intra table amendment

Edit Summary

The reasons of the commit is shown below:

[!!!][BUGFIX] *_user table password field is to short When using the hash-algorithm blowfish in saltedpasswords the hash is 60 chars long - therefore the field length has been set to 60 characters in SQL, too. Actually when converting old MD5 passwords to salted copies, an additional identifer char (M or C) is added to the hash which results in a 61 character hash. The result is that login is not possible anymore after converting the passwords like suggested by the reports module - the database field has to be increased. To be safe and because varchar only takes as much space as really needed increase the field length to 100 chars. Change-Id: I70646fe7939d41f1a7056378a85c506d10148480 Fixes: #39356 Releases: 4.5, 4.6, 4.7, 6.0 Reviewed-on: http://review.typo3.org/13324 Reviewed-by: Oliver Klee Reviewed-by: Helmut Hummel Tested-by: Helmut Hummel

Useful Stats

Table Births	0
Table Deaths	7
Attribute Additions	1
Attribute Deletions	3
Key Changes	0
Type Changes	1

Tables affected

be_users (1 changes)
pages (1 changes)
sys_category (26 changes)
sys_category_record_mm (5 changes)
sys_collection (1 changes)
sys_file (30 changes)
sys_file_collection (28 changes)
sys_file_processedfile (20 changes)
sys_file_reference (29 changes)
sys_file_storage (16 changes)
sys_history (1 changes)
sys_log (1 changes)

Issues

Ernesto Baschny Mon Jul 23 00:21:00 EEST 2012
Add documentation on t3lib_mail

Ernesto Baschny Mon Jul 23 00:22:00 EEST 2012
Enhance documentation of "prefixLocalAnchors"

Francois Suter Mon Jul 23 09:43:00 EEST 2012
New property config.typolinkUseCacheHash

Chris M?ller Tue Jul 17 13:31:00 EEST 2012
pageTS options.pageTree.onlineWorkspaceInfo not available in TYPO3 4.7

Chris topher Fri Jul 20 15:34:00 EEST 2012
typo3conf/localconf.php has been replaced

Christof Rodejohann Mon Jul 16 05:41:00 EEST 2012
wrapItemAndSub is stdWrap but documented as wrap

Is this helpful? Upvote 8:02 PM 12 FEB 2014

Add documentation on t3lib_mail

Ernesto Baschny assigned to **Francois Suter** **Status:** Closed
Priority: Should have
% Done: 100

Type: Feature
Category:
Closed at: null
Updated at: Sun Jan 06 20:26:00 EET 2013

he new t3lib_mail API since TYPO3 4.5 was documented here:http://wiki.typo3.org/Pending_Documentation#t3lib_mail This information should be migrated to the Core API some day, I guess.'

Is this helpful? Upvote

Figure 18 Details for a selected commit in commit level

CHAPTER 6.

WHAT DO THE DATA ON RELEASES TELL US WITH RESPECT TO SCHEMA EVOLUTION?

Apart from facilitating on-line interactive analysis, our integrated data model allows the traditional, batch extraction of knowledge from the collected data. As an example of this, in this chapter, we present how we can use contextual information such as the grouping of individual commits to releases to extract statistical findings that would otherwise be unattainable via the simple history of commits alone. Previous studies on the same data [SkVZ14, VaZS15, SkVZ15, VaZS17] studied evolution on the granularity of individual commits, as the information on releases had not been gathered yet. In this chapter, we exploit the data that we have gathered towards answering several research questions. First, we start with some descriptive statistics on releases in Section 6.1. Then, in Section 6.2 we continue our study on releases presenting progress reports using the aggregate changes on releases. After that, we discuss the schema size and heartbeat per release and finally we move on to present a rule-based classification technique for characterizing the nature of changes that affect the schema of the database. In Section 6.3 we present the aforementioned technique and we discuss our findings.

6.1 An aggregate overview of release data for schema evolution

We will start our deliberations, by observing how releases are related to change in terms of aggregate values per data set. The goal of this chapter is to answer one of our initial research questions: ***"given the entire version history of a schema, can we group individual changes in phases in order to provide statistic observations on the schema evolution?"***. In order to answer the above research question, we relate the information gathered on releases to individual commits in terms of aggregate values and study these aggregate measures.

6.1.1 Terminology

Some terminology is appropriate first. We remind the reader that unless explicitly mentioned, ***in all our deliberations, we work with the subset of releases in the schema history that include at least one commit to the schema definition file of the database of a project***. Releases that have not touched the schema are omitted from our investigation.

The *number of commits* of a release is the number of commits touching the schema definition file that are pushed in the master branch for each release. The *number of contributors* is measured with similar semantics.

Schema size. When we refer to *schema size* of a *version*, without other characterizations, we refer to the number of *tables* present at this version (equivalently, we use the term *schema size in terms of tables*). It is possible that we refer to *schema size in terms of attributes*, in which case, we count the number of attributes in all the tables of the version.

Whenever we refer to *releases*, rather than versions, the *schema size of a release* is the number of tables at the end of the release (i.e., the schema size of the last version of the release).

A reference to *schema growth* concerns the difference between the schema size of two (typically subsequent) versions (i.e., new–old).

Change Events. We discriminate births, deaths and updates as follows.

Births: for births, we count (a) the number of tables being added to the schema, (b) the number of attributes being born along with these newborn tables (which we call *table-born attributes*), and, (c) attributes added to tables that pre-existed (which we call *attributes injected*). Table-born attributes do not include the ones in the original, starting version of the schema.

Deaths: for deaths, we count (a) the number of tables being removed from the schema, (b) the number of attributes being deleted along with these deleted tables (which we call *table-gone attributes*), and, (c) attributes removed from tables that continue to exist (which we call *attributes ejected*).

> We collectively refer to the union of injected and ejected attributes as **jected attributes*.

Updates: with the collective term *attributes updated* we refer to the union of attributes that undergo a data type change with the attributes that participate in a primary key change.

> The collective term *intra-table updates* (in attributes) refers to the sum of the measures of **jected attributes* and *attributes updated*, i.e., it measures all the activity of change *within* a table, excluding its birth and possible death.
>
> The sum of intra table updates, table-born, and table-gone attributes is the *total volume of change* (in attributes) of a table.

Durations. We handle several types of durations in our data. The *release date* is the *Unix time* of a release, whereas the *human release date* is the date of release in *human time*. The *duration* of the release is the difference of the dates between the first and the last commit in the same release in days. The *real duration* of a release is the duration (again in days) between its own start and the start of its subsequent release. The latter is very useful for the case of releases with just a single commit or releases having all their commits at the same day. The *commit gap* is the time gap (in days) between the last commit of a release and the first commit of its subsequent release. The *release gap* is the time gap (in days) between two releases (say *i* and *i*+3), between which there exists one or more releases (say *i+1* and *i*+2) that did not include a commit of the schema definition file.

6.1.2 Breakdown of change

Figure 19 presents an overview of the different metrics of the lifetime of our six studied data sets. Clearly schemata grow over time, both in terms of tables and in terms of attributes (the Pearson correlation between schema size in terms of tables and attributes is 0.8 for the start of the lifetimes and 0.87 for the end of the lifetimes of the data sets).

Figure 20 presents the breakdown of events in percentages. To use a unique scale of measurements, we have used affected attributes as the unit of change. So the births and deaths of tables are covered by the respective events to their attributes. Then, we compute the percentage of each category of events over the total volume of change of the data set (expressed in number of affected attributes).

Table-born attributes range between 23% - 48% over the total number volume of change in attributes, with an average of 34% over all datasets and are the most common type in 4 out of 6 data sets and second in the other 2. The percentage of injected attributes ranges between 6% - 27% with an average of 17% over all data sets. The percentage of table-gone attributes ranges between 14% - 35% of events (with an average of 21%) whereas the percentage of ejected attributes ranges between 4% - 21% of events with an average of 12% over all the datasets. Finally, data type updates range between 5% - 28%with an average of 14% and key change ranges between 0% (in half the data sets) and 7% with an average of 2%.

	Time as …		*Schema size …*				*Births…*			*Deaths…*			*#Attr's with…*	
	Duration (human time)	Releases w. DDL	Tables at start	Tables at end	Attributes at start	Attributes at end	Tables	Table-born attr.	Injected attr.	Tables	Table-gone attr.	Ejected attr.	Data type update	Key change
Biosql	14 years, 11 months, 9 days	12	21	28	74	129	24	88	104	17	55	82	30	26
Ensembl	17 years, 8 months, 10 days	122	19	73	82	464	144	729	375	90	461	261	365	38
Mediawiki	13 years, 9 months, 1 days	112	17	48	100	348	75	356	232	44	218	122	361	22
Opencart	7 years, 10 months, 18 days	27	48	131	297	815	276	1652	215	193	1198	151	184	7
PhpBB	11 years, 5 months, 15 days	45	25	67	247	584	119	694	602	77	458	501	472	9
Typo3	13 years, 4 months, 0 days	52	10	23	122	414	29	438	115	16	219	42	92	0

Figure 19 Aggregate measures of change for the entire life of the six data sets that we study

	Time as …		*Schema size …*				*Total vol.*	*Births…*		*Deaths…*		*Updates…*	
	Duration (human time)	Releases w. DDL	Tables at start	Tables at end	Attributes at start	Attributes at end	*of change (attr.)*	Table-born attr.	Injected attr.	Table-gone attr.	Ejected attr.	Data type update	Key change
Biosql	14 years, 11 months, 9 days	12	21	28	74	129	*385*	23%	27%	14%	21%	8%	7%
Ensembl	17 years, 8 months, 10 days	122	19	73	82	464	*2229*	33%	17%	21%	12%	16%	2%
Mediawiki	13 years, 9 months, 1 days	112	17	48	100	348	*1311*	27%	18%	17%	9%	28%	2%
Opencart	7 years, 10 months, 18 days	27	48	131	297	815	*3407*	48%	6%	35%	4%	5%	0%
PhpBB	11 years, 5 months, 15 days	45	25	67	247	584	*2736*	25%	22%	17%	18%	17%	0%
Typo3	13 years, 4 months, 0 days	52	10	23	122	414	*906*	48%	13%	24%	5%	10%	0%
							Value range:	*23% - 48%*	*6% - 27%*	*14% - 35%*	*4% - 21%*	*5% - 28%*	*0% - 7%*

Figure 20 Total volume of change and its breakdown (in attributes) for the entire life of the data sets that we study

Overall, we can summarize our findings as follows:

- *Schemata typically grow via the addition of new tables and their table-born attributes, rather than with addition of new attributes to existing tables.* On average, 1 out of 3 attribute events involves an attribute being born with a new table and 1 out of 6 events involves an attribute being injected in an existing table. There are exceptions to this rule (here: Biosql with an inclination towards injections and PhPBB with practically a balanced mixture of table-born and injected attributes.) There are also different profiles of the intensity of the inclination to table-born attributes: some data sets rely heavily to new tables (OpenCart and Typo3) whereas others give a mild inclination to this trend (Mediawiki and Ensembl). In addition, in 4 out of 6 the ratio between table-born and attribute injected ranges 1.5 – 8.

- Schema cleanup, via the removal of tables and attributes (which encompasses renames too) follows the same pattern with schema expansion. On average, 1 in 5 events involved an attribute being removed along with its containing table and 1 in 8 events involves an attribute being removed from its table, with its table continuing to exist. *Attributes are mostly removed along with their containing tables, albeit with exceptions and differences in the balance of the different categories.* Again, Biosql is an exception to the rule and phpBB demonstrates a balanced mixture of table-gone and ejected attributes. Two data sets favor strongly table-gone attributes (again, Opencart and Typo3), signifying a development profile of working with the tables and two other data sets, Mediawiki and Ensembl demonstrate a mild inclination towards table-gone attributes.

- *Attribute updates comprise the smallest group of the attribute activity and are primarily of a data type change nature.* On average, 1 in 7 events involves an attribute being updated for its data type and 1 in 50 events (frequently: none) involves an attribute being involved in an updated primary key. Typically, updates are more often than deletions, but way fewer than attribute additions (with the exception of Mediawiki that demonstrates an excess of attribute data type updates).

The extent to which renames are involved (both at the table and at the attribute level) is an area of future research. We insist that our method is fully automated, and thus, the identification of renames (that would either require a manual intervention, or, results that are not 100% certain) is a future task and out of the context of this study.

6.1.3 Aggregate measures of activity and growth

There are some very interesting statistics concerning the amount of growth and update activity over the different releases.

Table growth. Starting with the amount of change of the schema size, we study the distribution of values of schema change: for each release we measure the growth or shrinking of the schema size (in tables), collectively referred to as *schema growth,* and we count how many releases pertain to each value. Figure 21 depicts our findings graphically.

Clearly, *lack of any growth (positively or negatively) dominates the evolution of schemata.* We have already seen this phenomenon when we studied the evolution on the basis of individual commits, and, not unexpectedly, we see the same behavior here. Very few releases express any change in schema size, and this is a typical pattern in all data sets, with the percentage of releases of zero growth ranging between 58% and 78%, and an average of 68% over all data sets. *In other words, in 2 out of 3 releases, the schema size typically remains the same*. Also, as the strong correlation of attribute and table growth has been demonstrated [SkVa13], we do not delve into the study of attribute growth.

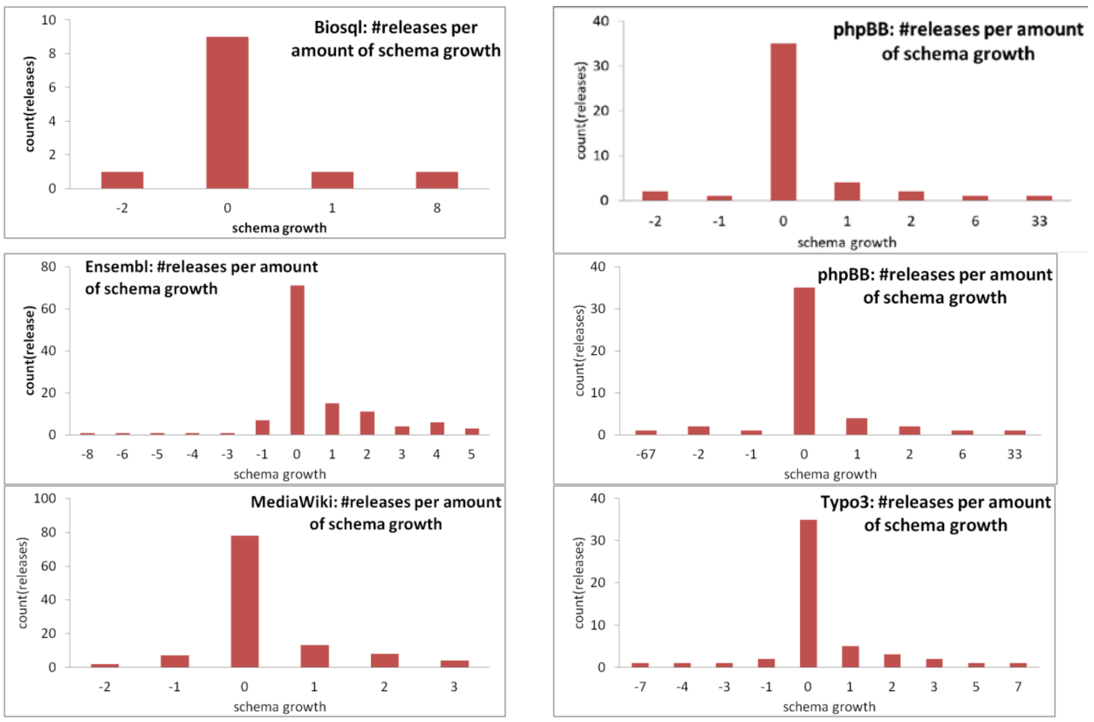


Figure 21 Release breakdown per schema size growth: for every value of schema growth (in tables), we measure how many releases have demonstrated this value

Attribute *jections. In Figure 22 we present the breakdown of values for attribute injections to pre-existing table and attribute ejections from tables that are not deleted. It is straightforward to see that (a) *in more than 40% of releases*, (actually, between 41% and 54% of all releases, with an average of 46% over all data sets) *there is no occurrence of such actions* ***(i.e., a value of zero at the horizontal axis).*** *In all the data sets, however, we can observe the existence of releases with a large amount of such restructurings*. Remember that injections involve 17% of all events and ejections 12% of all events on average.

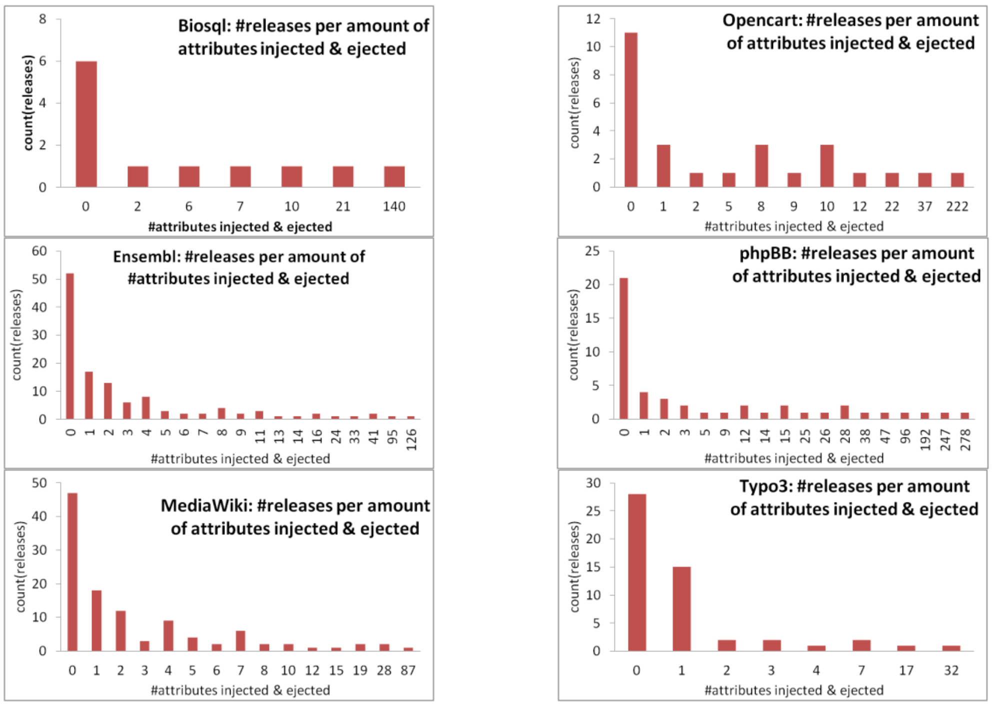


Figure 22 Release breakdown per amount of attributes injected or ejected: we add the amount of attributes injected to existing tables and ejected from tables that survive this change and measure we measure how many releases have demonstrated this value

For each release we compute the percentage of *jections it contains over the total number of *jections. Then, we sort the releases over this percentage. To show that *few releases contain a large part of the *jections*, in Figure 23, we present the cumulative percentage of *jections, over the total number of *jections, for the top ranked releases. We can see that *the releases in the top-5 positions amass between 51% - 99% of *jections* (Figure 23). In two cases, Mediawiki and Ensembl, with a long number of releases that touch more than 5 attributes, the percentage of the releases in the top-5 positions slightly surpasses 50% (which is already too much). In the rest of the data sets, it rests between 78% -99%. Interestingly, the profiles of the different data sets differ in their progress, however in all but one of the data sets, 50% is reached in the top-3 releases. The point made here is that *there exist releases of mass maintenance in terms of attribute injections and ejections, and few of them (indicatively, the top-5) take up between 51% and 99% of *jections.*

					*Cumulative pct of *jections for the top-5 releases, with rank:*				
	#Releases	total attr change	attr *jected	*jections over total	1	2	3	4	5
Biosql	12	385	186	48%	75%	87%	92%	96%	99%
Ensembl	122	2229	636	29%	20%	35%	48%	53%	57%
Mediawiki	112	1311	405	31%	21%	35%	45%	48%	51%
Opencart	27	3407	366	11%	61%	71%	77%	80%	88%
PhpBB	45	2736	1103	40%	25%	48%	65%	74%	78%
Typo3	52	906	92	10%	35%	53%	68%	73%	79%
				Range:	*20% -75%*	*35% -87%*	*45%-92%*	*48%-96%*	*51%-99%*

Figure 23 Cumulative percent of *jections for the releases in the top-5 positions with respect to *jections (dark red for the high values at start and end, and for the steps higher than 10%; blue for low values at start and end).

Note than the actual values for the top-5 events can be seen in the values of the x-axis in Figure 22. For the rare occasions where more than one releases tie at the same value, we count all of them (so, we have the top -5 ranks and not top-5 releases).

Attribute updates. ***Attribute updates are more frequent than one would expect, but typically less in numbers than injections, and slightly higher than ejections*** (see Figure 20). With a small supremacy of injection, the three categories keep close in volumes with only a couple of exceptions (too few data type updates in Biosql and too many of them in Mediawiki, compared to the two other categories).

The breakdown of occurrences for the updated attributes follows the same pattern with ejections and injections with (a) most releases carrying zero such events, and (b) specific releases of mass maintenance in the attributes of several tables of the schema.

	#Releases	total attr change	attr updated	updates over total	*Cumulative pct of updates for the top-5 releases, with rank:* 1	2	3	4	5
Biosql	12	385	56	15%	46%	88%	95%	98%	100%
Ensembl	122	2229	403	18%	28%	34%	45%	52%	55%
Mediawiki	112	1311	389	30%	37%	62%	66%	70%	73%
Opencart	27	3407	191	6%	81%	88%	91%	95%	98%
PhpBB	45	2736	192	7%	10%	20%	28%	39%	44%
Typo3	52	906	92	10%	35%	53%	68%	73%	79%
				Range:	*10%-81%*	*20%-88%*	*28%-95%*	*39%-98%*	*44%-100%*

Figure 24 Cumulative percentage of updates for the top-5 releases (dark red for the high values at start and at the end, and for the steps higher than 10%; blue for low values at start and end).

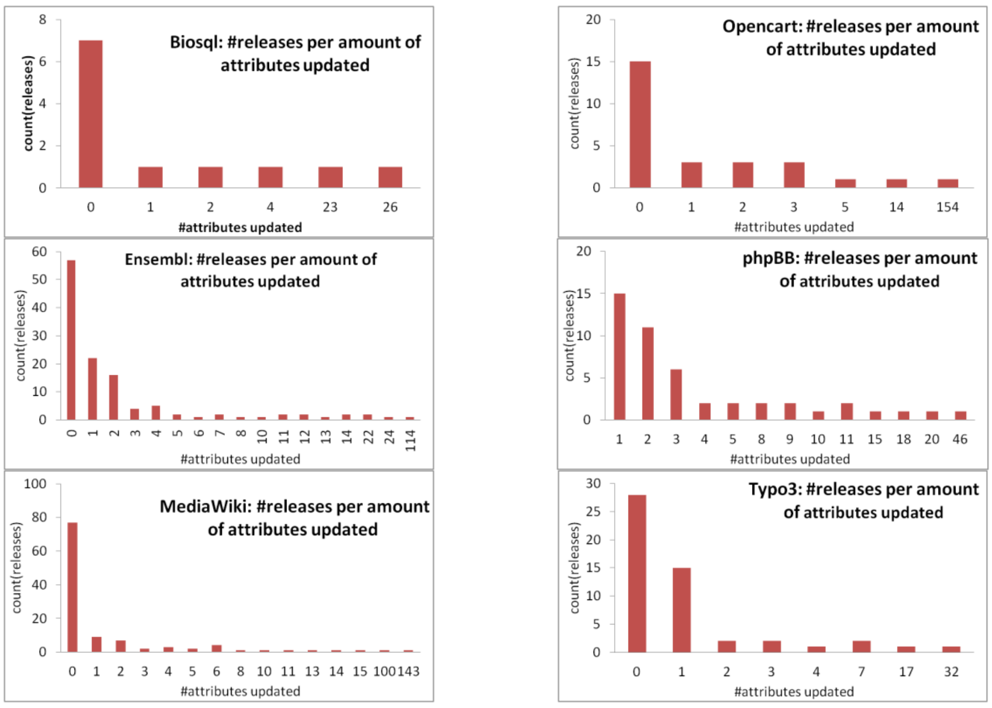


Figure 25 Release breakdown per amount of attributes updated: we add the amount of attributes with a data type change or participating at a key change and measure we measure how many releases have demonstrated this value

To clearly show the role of top releases with respect to updates we compute the percentage of attributes updated over the total number of attributes updated. Then, we sort the releases over this percentage and we present the cumulative percentage for the top ranked releases on Figure 24. Concerning the role of the releases in the top-5 ranks with respect to updates, we see a similar effect as in the case of *jections, albeit with less intensity. Interestingly, all data sets except for Opencart, start with a low or medium sized percentage for the top-1 release, but quickly build up the percentages; by the time we reach to 5th highest release in terms of updates, only one data set is below 50%.

Releases with zero change dominate. An overall observation, vividly reported on Figure 26, is that *absence of change is omnipresent for all types of changes*. Releases with zero schema growth range between 58% - 78%, with an average of 68%: 2 out of 3 releases mark zero schema growth. The respective range for attribute injections and ejections ranges between 41% and 54%, whereas for attributes updated it demonstrates a much broader range, between 33% - 69%. On average attribute *jections and updates occur only in half the releases.

Overall, we can argue that ***change is mostly absent or kept in small numbers, with more than 40% of the releases carrying zero change in at least one of the categories, few releases collecting a large percentage of the changes, and, a large number of releases carrying very small amounts of updates each.***

		Releases with zero …		
	#releases	schema growth	*jected attributes	attributes updated
Biosql	12	75%	50%	58%
Ensembl	122	58%	43%	47%
Mediawiki	112	70%	42%	69%
Opencart	27	63%	41%	56%
PhpBB	45	78%	47%	33%
Typo3	52	67%	54%	54%
	Range:	58%-78%	41% - 54%	33% - 69%
	Average:	68%	46%	53%

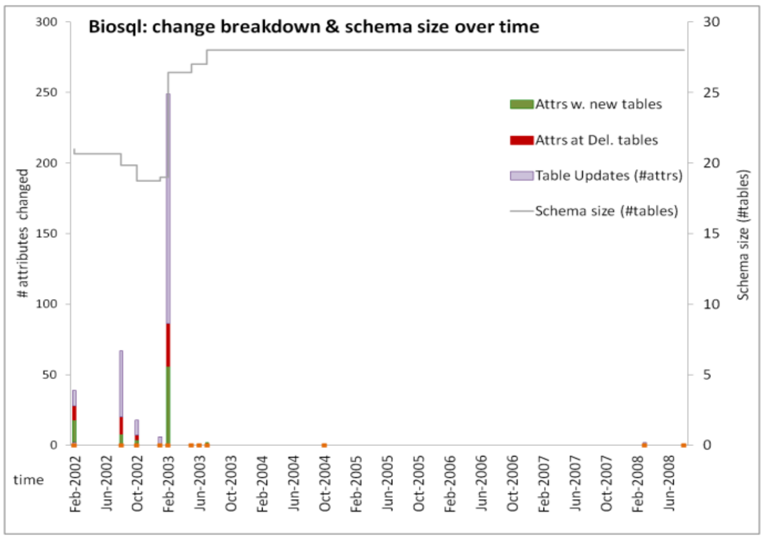
Figure 26 Percentage of releases with zero change in different categories of change

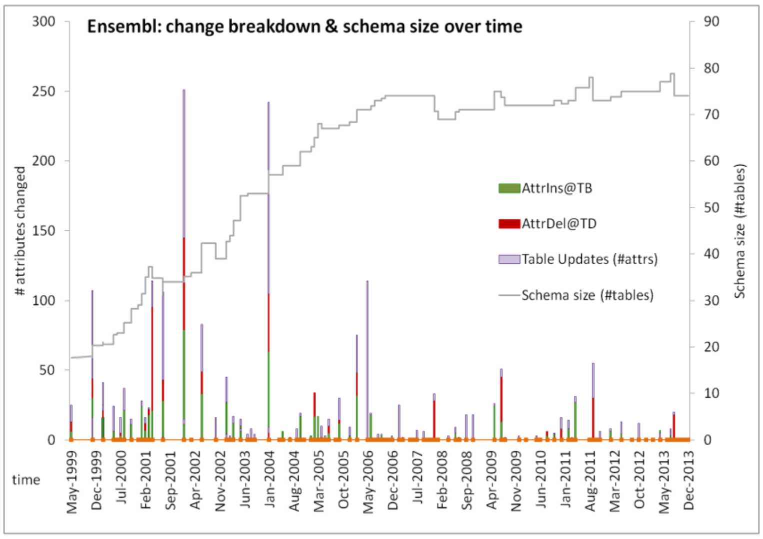
6.2 Schema size, heartbeat and progress of change over time

The research goal of this chapter is to answer the following research question: *"how frequently and extensively do database schemata evolve in terms of releases? "*. In order to answer the above question, we study the breakdown of changes in the version history of schema histories and the progress of individual changes over time.

Starting with schema size and heartbeat, Figure 27 and Figure 28 depict the combined evolution of schema size and heartbeat for the data sets that we study.

- The horizontal axes represent human time. Every small orange square over the horizontal axis signifies the date of a release (one can observe that there exist releases without any change, marked only by their particular small orange square).
- The left vertical axis measures change in terms of number of attributes involved. The stacked bars within the chart indicate the attributes born with new tables, deleted along with removed tables and the intra-table updates (attributes *jected and updated, together).
- The right vertical axes measures schema size. The thin grey line that traverses the chart of each data set relates to the right vertical axes and measures schema size in tables (again, for completeness, let us mention that measuring in attributes presents a very similar line, so we omit this line to avoid cluttering the diagram).





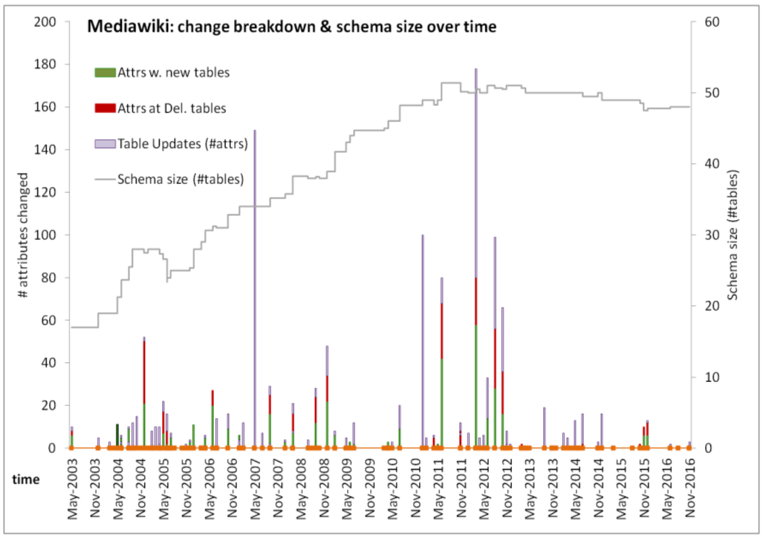
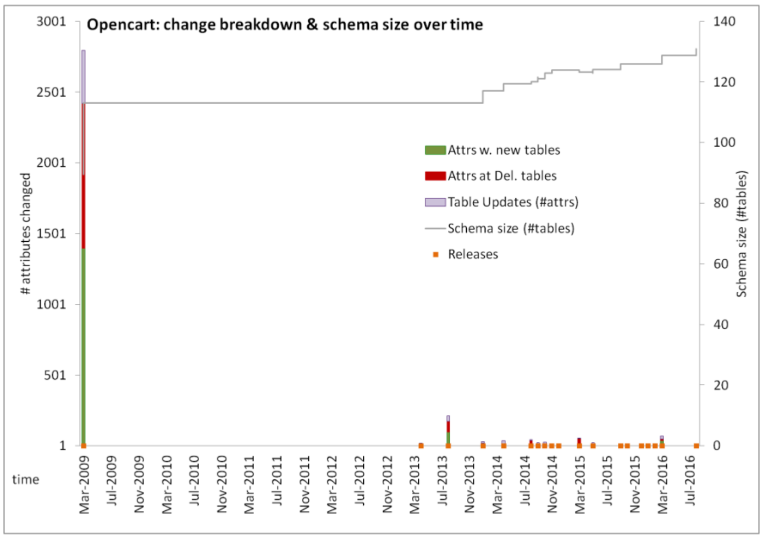
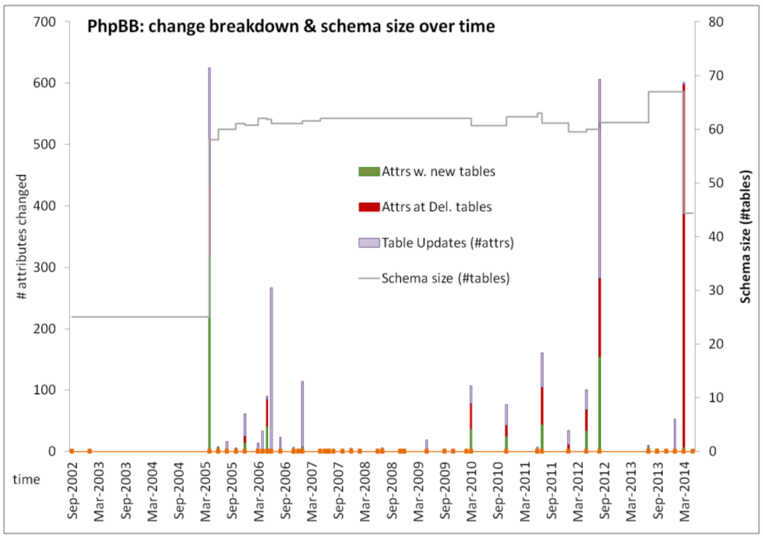


Figure 27 Schema size and heartbeat evolving over time for Biosql, Ensembl, and MediaWiki





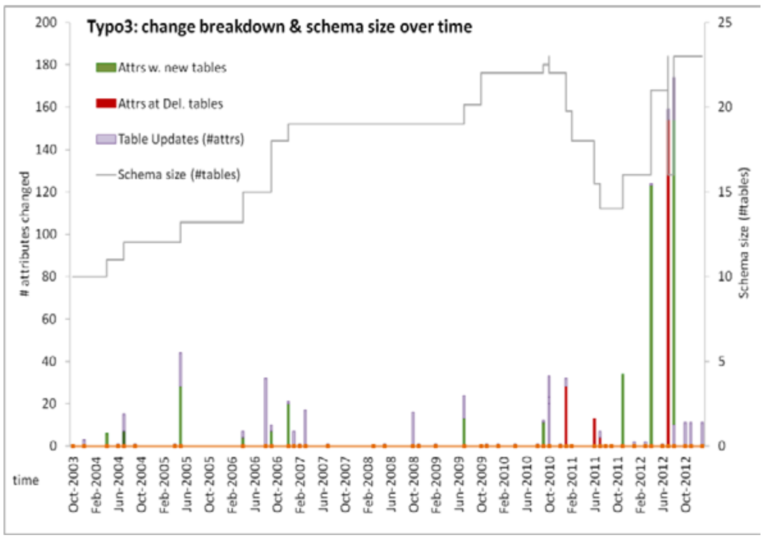


Figure 28 Schema size and heartbeat evolving over time for Opencart, phpBB, and Typo3

After studying the schema size and the heartbeat, we examine the progress of each software project using the different types of change events that can take place in the schema of the database. Specifically, we compute the cumulative progress for (a) *table-born attributes* and *table-gone attributes*, (b) **jected attributes* and (c) *attributes updated*.

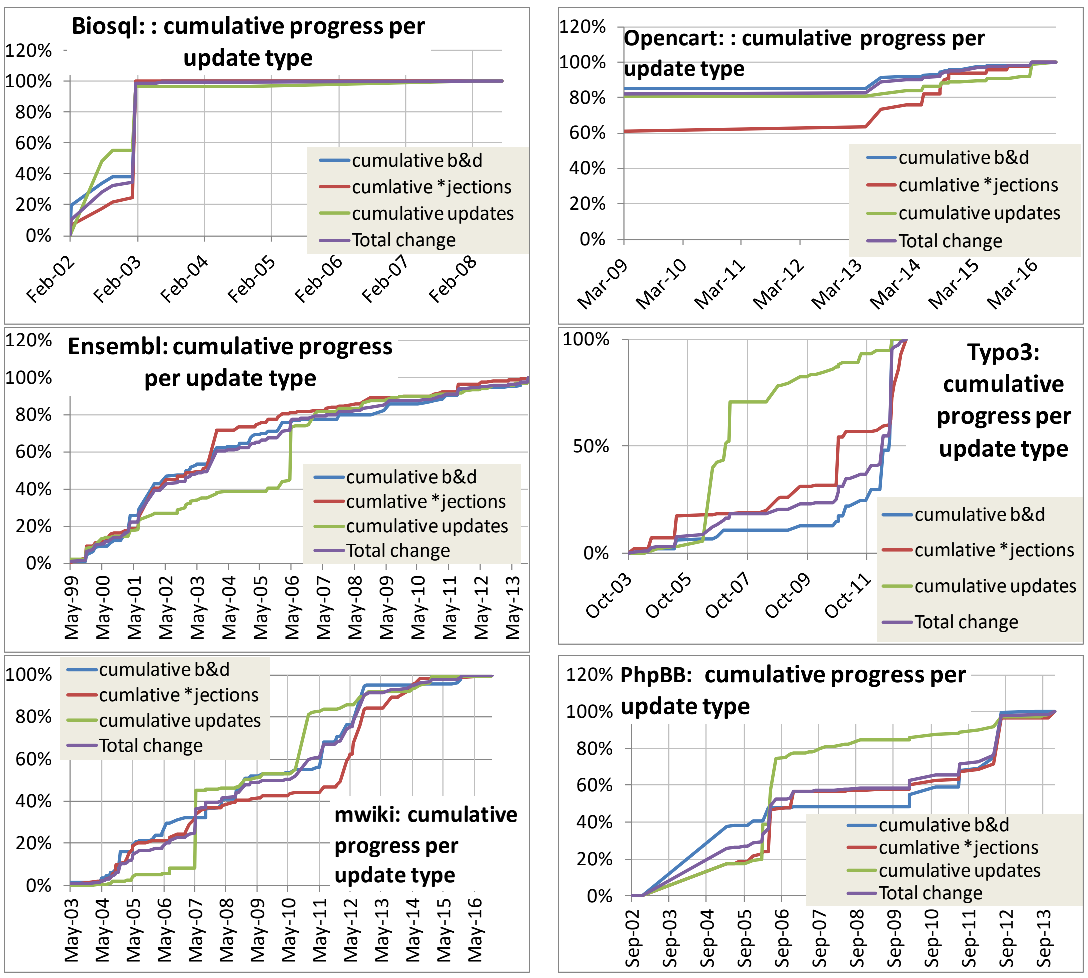


Figure 29 Cumulative progress per update types including all datasets.

Figure 29 presents the cumulative progress for all different update types including all datasets. We observe that in the BioSQL dataset all progresses are pretty much in synchronization. In addition, the progress is completed within the 1st year. Concerning Ensembl, in the 5 first years 60% of the progress is completed and in the 7 first years 80% of the progress is completed. It is worth mentioning the spike of attribute updates in the 7th year of the dataset and after that year, the progress is slower. We observe that all the types of updates are in synchronization except for the spike of updates. With the exception of *jections that show an abrupt rise in 2012, and another abrupt rise of updates in 2007, the overall rate of change in Mediawiki seems stable. Beyond that, a small slowdown towards the end (at the middle of 2014) is detected where all categories have reached the 95%. Opencart, in the 1st year starts with 60% *jections and 80% everything else. Over the next 4 years nothing happens to the schema and the 2nd release appears in 2013. After that, there is small progress on the project. Regarding Typo3, except for early updates, nothing really happens till late in 2012. Specifically, in Nov. 2011 the births and deaths gather 30%, *jections gather 57%, updates gather 93% and the percent of total updates is 41%. Most of the changes take place afterwards, practically within 2012. Finally, PhpBB starts linearly for the first 3.5 years; then over the next 4 years nothing really happens and then goes slowly up. All update types except attribute updates have an abrupt rise (from ~75% to ~100%) in 2012 because attribute updates had risen already with an abrupt rise in 2006.

Concerning the calmness periods for each dataset, we observe that the calmness period in Biosql constitutes to all of its life after the first year. Similarly, the calmness period in Opencart starts after the first release and ends at the end of the project's lifetime. For Typo3 the calmness period lasts for about six years starting in 2004 and ending in 2010. Similarly, the calmness period of PhpBB starts in 2004, lasts for about six years and ends in 2010. The calmness period for Mediawiki lasts for about three years, from 2008 to 2011. Finally, Ensembl does not seem to have any calmness periods. The progress seems to continuously raising during the entire lifetime of the project.

Overall, ***the progress of individual update types is in synchronization in all datasets, except the spikes of updates in Typo3 and in Phpbb. Moreover, the evolution patterns are different across the datasets.***

- Ensembl and Mediawiki, in the first years of their project's life, have a steady progress until the last three years where the progress is slower.

- Opencart and Biosql contain abrupt rise in the progress and after that the progress is almost frozen. The difference between the two of them is that Opencart starts with 60% *jections and 80% for the other categories in the first release and Biosql starts with small percentages in the first release but the progress is completed within the first year. Opencart is the only dataset that starts with progress percentages way larger than zero. This is because the first stable version of the software was in progress for 4 years (1504 days) and contains 152 commits.

- Both Typo3 and Phpbb have a large calmness period in the middle of their life. The difference between the two of them is that Typo3 gathers a small percentage of the progress before its calmness period instead of

Phpbb which gathers more than 50% of its progress before its calmness period. For this reason, in Typo3 the progress has an abrupt rise in the last year of its development in contrast with Phpbb where the progress is lower after its calmness period.

6.3 Classifying Releases

Given the multitude of information on the behavior of the size of the schema per release, as well as on the way the changes have occurred (in terms of table and attribute births and deaths, attributes being injected and ejected and attributes being updated in terms of data types and participation to key constraints), the next question to ask is *"can we characterize the nature of a release by inspecting these characteristics?"*

As we mentioned in previous sections, organizing and studying the commits of a database schema organized in releases is important because this way we can obtain insights regarding the development goals for specific time periods.

In this study we focus on both commit and release classification. From here on we refer to any of those as versions. We follow a rule-based classification technique that utilizes rules to characterize (a) the nature of the version's activity (e.g whether the schema is expanded with new tables or the existing tables are maintained internally) and (b) the intensity of the activity (low, medium, high). Practically, we divide the multidimensional space created by the different version metrics (eg. schema size and growth, amount of updates within tables, amount of attributes added to existing tables etc) into regions with different semantics. Of course, the problem is then reduced to deciding the range of each such region. To this end, we define two different *discretization* techniques, *maintenance discretization* and *volume of change discretization*. The first one will provide the discretization of releases into maintenance categories and the second one will discretize each of these categories into three further categories based on the amount of changes that take place in each release.

6.3.1 Maintenance categories discretization

In order to classify the commits or releases that affect the schema of the database into maintenance categories, we first need to inspect the complex behavior of schema evolution. Therefore, based on the change events that can take place in a single commit or release, we consider three major categories of structural modification that will help us classify a commit or a release respectively.

- *Table activity*: this category includes two types of change events that affect the schema from the scope of (a) table births and (b) table deaths. We remind the reader that for table births we count the number of tables added to the schema and for table deaths we count the number of tables removed from the schema.
- *Intra table growth*: this category contains change events that affect the number of table attributes. These change events include (a) the number of attributes injected (attributes added to tables that pre-existed) and (b) the number of attributes ejected (attributes removed from tables that continue to exist).
- *Intra table updates*: this category includes change events concerning (a) attributes that undergo a data type change and (b) attributes that participate in a primary key change.

Based on the above structural modification categories, we apply at most one label for each category.

6.3.2 Zero logical change

If a version does not have any label after the examination of the three categories then we apply to the specific version the maintenance label *"Zero Logical Change"*. This means that the changes that took place in the specific version did not affect the logical schema of the database. Changes that do not affect the logical schema of the database can be changes on indexes or in comments.

6.3.3 Table activity

We define the sum of table births and table deaths as follows:

$$\text{table\_change} = \#\text{table\_births} + \#\text{table\_deaths}$$

In order to apply a selected label for this category we discriminate the following cases:

<u>Case 1.</u> $table\_change = 0$

This means that no tables were added or removed from the schema on a specific version. In this case there is no table activity, so we do not apply any label.

Case 2. $table\_change > 0$

This means that there is table activity that affects the schema of the database and we need to address all the different cases. Practically, we identify the following possible values (i.e the domain) for table activity:

i. *Growth: table expansion*, referring to the situation where the developers are significantly expanding the schema with new tables

ii. *Maintenance: table shrinking,* where developers are intentionally performing cleanup, perfective maintenance by removing unnecessary tables

iii. *Maintenance: table restructuring*, where there is a mixed activity of additions and deletions (typically encountered in renames and restructurings)

Before we examine all the cases we define some useful equations that will be used later. We use a user defined threshold $t \in [0, 1)$ that will help us classify versions, to define two quantities, the table birth percentage and the table death percentage, defined as follows:

$$tb_p = \frac{\#table\_births}{\text{table\_change}}$$

$$td_p = \frac{\#table\_deaths}{\text{table\_change}}$$

For our characterizations we use t = 0.3

We have the following cases here:

Case 2.1. $tb_p - td_p > t$

This case means that the percentage of tables added to the schema is larger than the percentage of tables deleted from the schema by the threshold t. Therefore, we apply the label *Growth: table expansion* to the specific version.

Case 2.2. $td_p - tb_p > t$

This case means that the percentage of tables removed from the schema is larger than the percentage of tables added to the schema by the threshold

t. Therefore, we apply the label *Maintenance: table shrinking* to the specific version.

<u>Case 2.3.</u> $|tb_p - td_p| \leq t$

In this case we have maintenance in terms of restructuring in the specific version. Therefore, we apply the label *Maintenance: table restructuring* to the specific version.

Finally, we can summarize the discretization process for this category in the following formula.

$$TA\_label = \begin{cases} - , & tb_p + td_p = 0 \\ \text{Growth: table expansion,} & tb_p - td_p > t \\ \text{Maintenance: table shrinking,} & td_p - tb_p > t \\ \text{Maintenance: restructuring,} & otherwise \end{cases}$$

6.3.4 Intra table growth

In order to apply a selected label for this category we follow a similar methodology with the methodology of *table activity* category. Again, we need to classify a version's activity into one of four classes:

i. *Growth: intra table expansion*, when there is a significant amount of attributes added (*injected*) to existing tables.

ii. *Maintenance: intra table shrinking* where developers are performing cleanup, perfective maintenance by removing (*ejecting*) attributes from surviving tables.

iii. *Maintenance: intra table restructuring* where there is a mixed activity of attributes injected and attributes ejected.

iv. No changes were made to the schema of the database.

We define the sum of attributes injected and attributes ejected as follows:

$$intra\_table\_change = \#attributes\_injected + \#attributes\_ejected$$

Therefore, we discriminate the following cases:

<u>Case 1.</u> $intra\_table\_change = 0$

This means that there are not injected and ejected attributes in the schema on a specific release. In this case we do not apply any label on the release.

<u>Case 2.</u> $intra\_table\_change > 0$.

We define a threshold $t \in [0, 1)$ and the percentage of attributes injected and attributes ejected as follows:

$$ai_p = \frac{\#attributes\ injected}{intra\_table\_change}$$

$$ae_p = \frac{\#attributes\ ejected}{intra\_table\_change}$$

For our characterizations we use t = 0.3

We have the following cases here:

<u>Case 2.1</u>. $ai_p - ae_p > t$

This case means that the percentage of attributes injected is larger than the percentage of attributes ejected by the threshold t. Therefore, we apply the label *Growth: intra table expansion* to the specific version.

<u>Case 2.2.</u> $ae_p - ai_p > t$

This case means that the percentage of attributes ejected is larger than the percentage of attributes injected at the threshold t. Therefore, we apply the label *Maintenance: intra table shrinking* to the specific version.

<u>Case 2.3.</u> $|ai_p - ae_p| \leq t$

In this case, we consider that we have intra table maintenance in the specific version. Thus, we apply the label *Maintenance: intra table restructuring* to the specific version.

Finally, we can summarize the discretization process for this category in the following formula.

$$ITG\_label = \begin{cases} - \quad , & ai_p + ae_p = 0 \\ Growth\text{: } intra\ table\ expansion\ , & ai_p - ae_p > t \\ Maintenance\text{: } intra\ table\ shrinking, & ae_p - ai_p > t \\ Maintenance\text{: } intra\ table\ restructuring, & otherwise \end{cases}$$

6.3.5 Attribute updates

In this category the things are simple. If *#type_updates* + *#key_updates* > *0* then we apply the *Maintenance: intra table amendment* label. Otherwise we do not apply any label.

6.3.6 Volume of change discretization

One problem that arises from the above labels is that they do not measure the volume of change. Therefore, we need to define taxonomies for measuring the volume of those changes. In this thesis, we consider three different levels (*low, medium* and *high*) that measure the extent of the structural modifications in the schema of the database.

This discretization process is based on descriptive statistics of two structural modification types: *intra table updates (*includes *attribute updates* and *intra table growth)* and *table activity.* At this point, it is useful to mention that the discretization process is similar for both releases and commits. The two processes are explained below.

(a) Releases

In order to be able to define when an update category has *low, medium* or *high* impact on the database schema i.e, in order to define the range of each intensity value, we inspect the distribution of changes across the major modification types (*intra table updates* and *table activity).* More specifically, for each dataset we create two different plots, one for each modification type. In Appendix 2 we present, for all datasets and for each modification type (*intra table update* and *table activity*), two different kinds of plots. We refer to

$intra\_table\_change + \#attributes\ updated$ as <u>*number of total intra changes*</u> for the *intra table update* category and to $table\_change$ as <u>*number of total births and deaths*</u> for the *table activity* category.

$$\#total\ intra\ changes = \#intra\_table\_change + \#attributes\ updated$$

$$table\_change = \#table\_births + \#table\_deaths$$

Moreover, the first kind of plot refers to the discretization of commits based on total amount of updates and the second kind refers to the discretization of releases based on total amount of updates. A subset of those plots is presented in Figure 30 and Figure 31 and refers to the discretization of intra table updates for releases. Specifically:

- The horizontal axis represents the releases which are sorted in asceding order based on the volume of change (<u>*number of total intra changes*</u> or <u>*number of total births and deaths*</u>).
- The left vertical axis represents the number of <u>*number of total intra-table changes*</u> or the <u>*number of total births and deaths*</u>.
- The vertical red lines represent the thresholds for the discretization into three categories. These thresholds are set to a percentage of the release population based on the observation on the volume change distribution.

Although all plots look strikingly similar, the exact thresholds differ. Therefore, based on the exact numbers we extracted Table 3 which presents the ranges for each discretization category for the two different modification types including all datasets.

	Total table births & deaths			Total Intra table changes		
	Low *(lower 80%)*	Medium *(80% - 95%)*	High *(higher 90%)*	Low *(lower 80%)*	Medium *(80% - 95%)*	High *(higher 95%)*
Biosql	< 3	4-10	> 10	< 11	12 - 47	> 47
Ensembl	< 4	4-7	> 7	< 8	9-25	> 25
Mediawiki	< 3	3-5	> 5	< 7	8-19	> 19
Opencart	< 4	4-9	> 9	< 12	12-27	> 27
Phpbb	< 3	3-20	> 20	< 36	36-107	> 107
Typo3	< 2	2-4	> 4	< 9	9-17	>17
#Releases:	*308*	*40*	*22*	*296*	*53*	*21*

Table 3 Discretization thresholds for releases, including the two different modification types.

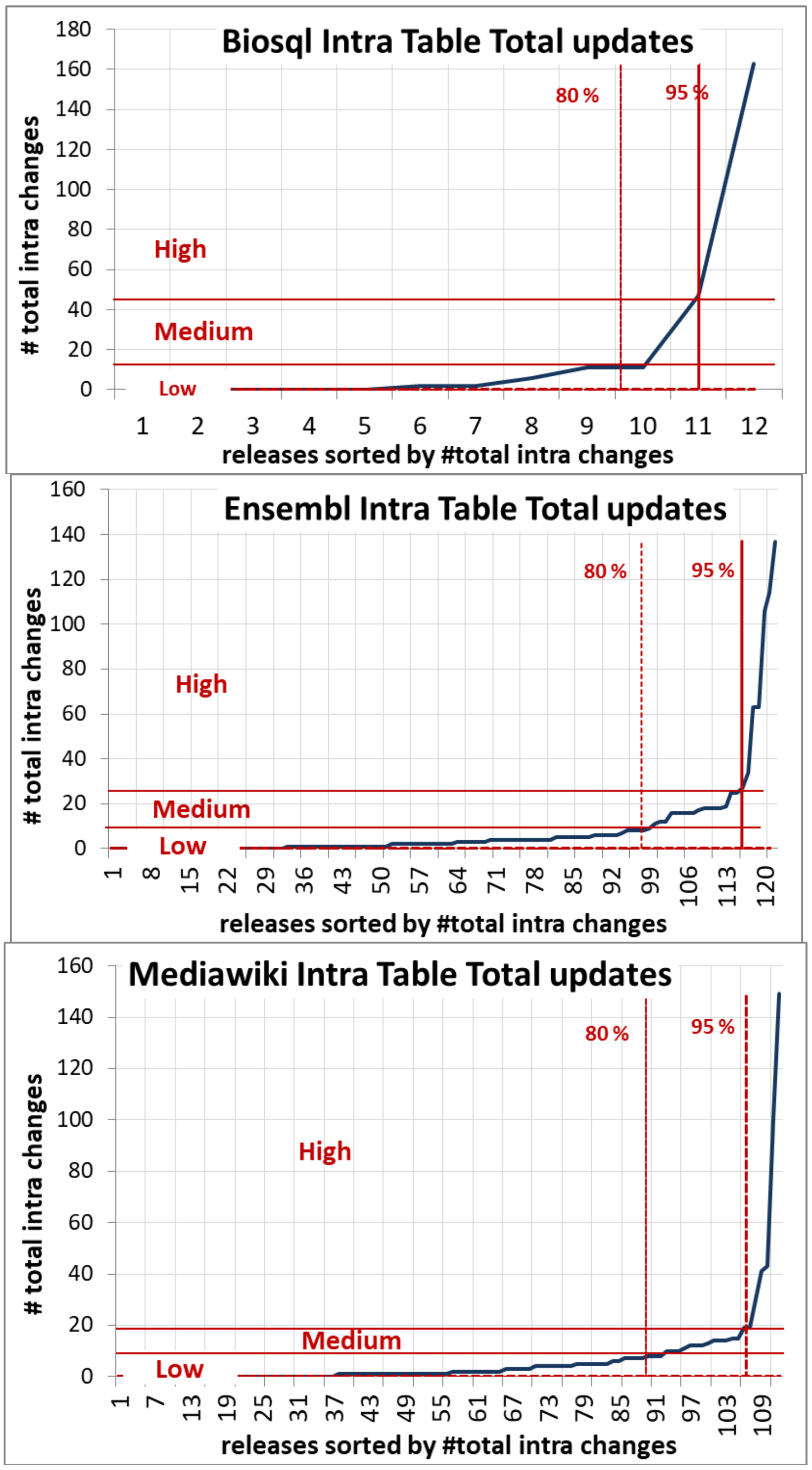


Figure 30 Distribution of total intra table updates for releases for Biosql, Ensembl and Mediawiki.

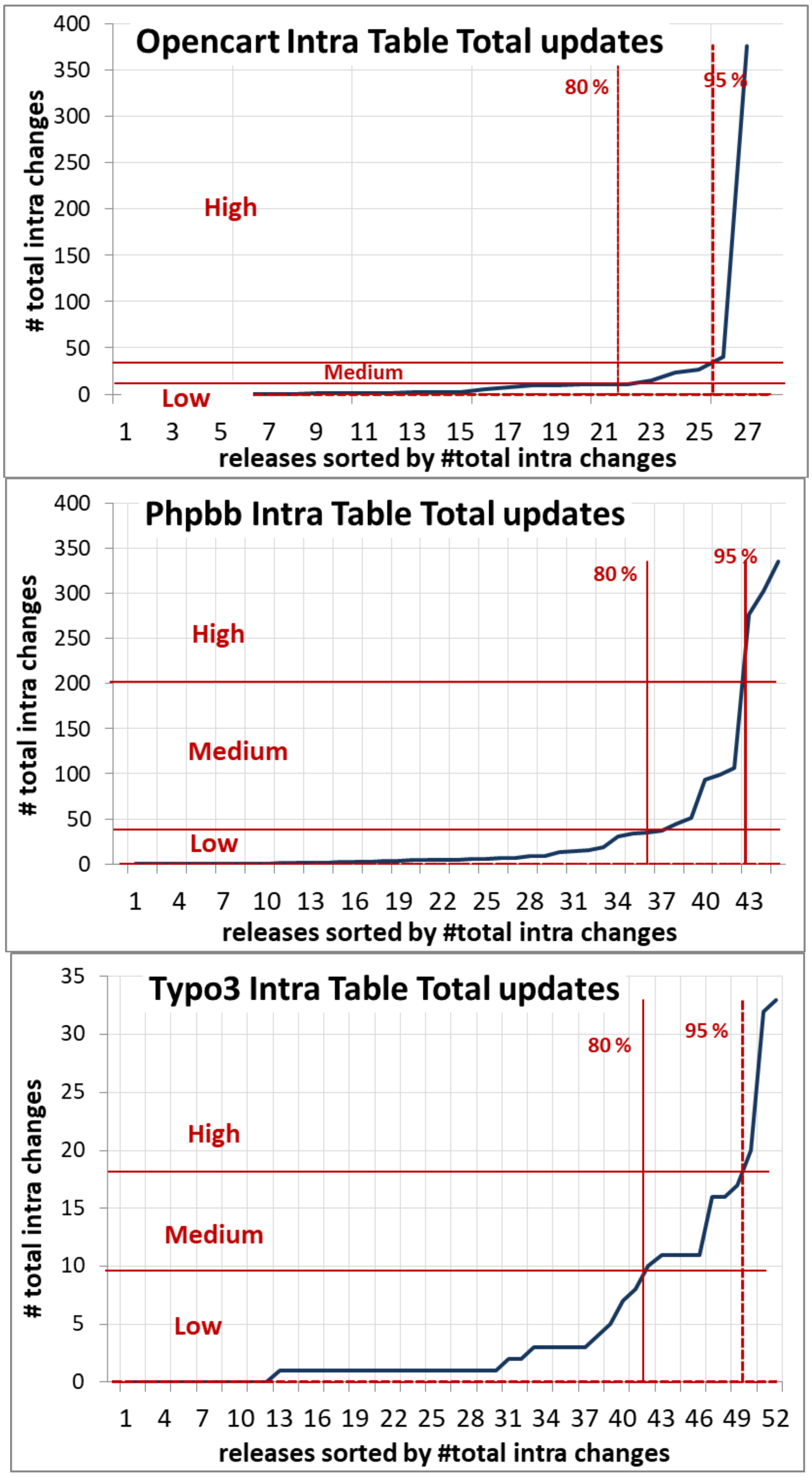


Figure 31 Distribution of total intra table updates for releases for Opencart, Phpbb and Typo3.

(b) Commits

Concerning commits, we use the same process of discretization using again two major modification types (*intra table updates* and *table activity)* and creating similar plots which are presented in Appendix 2.

Although all plots look strikingly similar, the exact thresholds differ. Therefore, based on these plots we extracted Table 4 which presents the ranges for each discretization category for the two different modification types and including all datasets.

	Total table births & deaths			Intra table total updates		
	Low *(lower 80%)*	Medium *(80% - 95%)*	High *(higher 95%)*	Low *(lower 80%)*	Medium *(80% - 95%)*	High *(higher 95%)*
Biosql	< 2	2-5	> 5	< 5	5-18	> 18
Ensembl	< 2	2	> 2	< 3	3-6	> 6
Mediawiki	< 2	2	> 2	< 3	3-6	> 6
Opencart	< 1	1-2	> 2	< 2	2-5	> 5
Phpbb	< 2	2-4	> 4	< 7	7-24	> 24
Typo3	< 2	2	> 2	< 4	4-11	> 11
#Commits	*1536*	*122*	*62*	*1433*	*210*	*77*

Table 4 Discretization thresholds for commits, including the two different modification types.

6.3.7 Summarizing the labeling possibilities

In this section we summarize our rule-based classification technique as follows. We consider three different types of modification changes for each version: (a) *table activity*, (b) *intra table growth* and (c) *intra table updates*. A version can have more than one type of modification change because developers can make different kind of changes in each version. For example, a developer can add new tables in the schema and remove attributes from tables that continue to exist in the same time. Therefore, a version can have more than one label assigned to it. More specifically, at most one label for each type of modification change can be assigned to a version. For this reason, the number of labels that can be assigned to each version ranges between 1 and 3.

Each of these three categories consists of different cases that handle the nature of the activity.

<u>*Table activity.*</u> A version is assigned (a) *Growth: table expansion*, (b) *Maintenance: table shrinking,* or, (c) *Maintenance: table restructuring* label, based on the nature of table activity. If there is not table activity, then no label is assigned to a specific version.

<u>*Intra table growth.*</u> A version is assigned (a) *Growth: intra table expansion*, (b) *Maintenance: intra table shrinking,* or, (c) *Maintenance: intra table restructuring* label, based on the nature of **jected* attributes. If there are not **jected* attributes, then no label is assigned to a specific version.

<u>*Attribute updates.*</u> A version is assigned *Maintenance: intra table amendment* label in the case where there are attributes that undergo a data type or attributes that participate in a primary key change. If there are not updated attributes, then no label is assigned to a specific version.

Based on the above, we discriminate each of the aforementioned labels into two parts that can be summarized as follows: *<meta-label:> <activity-nature>*. For example, in the above labels *Growth* and *Maintenance* at the beginning of the label are considered as *meta-labels.* Beyond that, one additional label that measures the *intensity of the activity* (*low, medium, high*) must be assigned at the start of each of those labels.

Overall, we can summarize the format of the labels as follows:

- <u>*Absence of any kind of activity.*</u> *Zero Logical Change* label is assigned to the specific version.
- <u>*Existence of any kind of activity.*</u> In this case, we used the two aforementioned discretization techniques to assign the suitable labels to each version. The format of these labels can be summarized as follows:

  <intensity> <meta-label>:<activity-nature>

  where *<intensity>* can be *low, medium* or *high, <meta-label>* can be either *Growth* or *Maintenance* and *<activity-nature>* can take all the possible characterizations based on type of change from each category.

6.3.8 Overall stats

One of our initial research goals on version classification was to answer the following question: ***what are the most used modification types in the evolution history of a database schema?***

For this reason, the first thing we measure is the percentage of each label in the entire life of the schema for all datasets. We separate our study in two different parts: (a) commits and (b) releases.

(a) Commits

For this part of the investigation, we measure the percentage of commits where each label appears and we present the results in Table 7 for all datasets. Beyond that, we sort the labels based on the arithmetic mean over all datasets and we show the top-5 ranked labels and bottom-5 ranked labels at Table 5 and Table 6 respectively.

We observe that the top-5 ranked labels based on the arithmetic mean of releases is (1) *Zero Logical Change* which ranges between 17% - 57% with an average of 36% over all datasets, (2) *Low maintenance: intra table amendment* which ranges between 6% - 33% with an average of 20%, (3) *Low Growth: intra table expansion* which ranges between 7% - 26% with an average of 16%, (4) *Low Growth: table expansion* which ranges between 0% - 11% with an average of 6% and (5) *Medium Growth: table expansion* 2% - 10% with an average of 6%.

It is worth mentioning that in all datasets except PhpBB the *zero logical change* label exists in more than 1 out of 4 commits. In addition, we observe that ***all but one of the top-5 ranked labels gather low activity. Beyond that, we can observe in Table 6 that the majority of bottom-5 labels gather high activity. Based on these observations, we conclude that the changes introduced by commits are quite often of zero (in 4 out 6 data sets the largest category) or low volume while high volume of activity are really infrequent.***

To forestall the criticism that this should be expected based on the definition of high and low, we refer to the reader to Figure 30, Figure 31and the figures of the Appendix where it is evident that the definition was adapted to the scarcity of intense updates and the predominance of zero updates and not vice-versa.

Characterizations	Mean	Standard deviation	Max	Min
Zero Logical Change	36%	14%	57%	17%
Low Maintenance: intra table amendment	20%	11%	33%	6%
Low Growth: intra table expansion	16%	8%	26%	7%
Low Growth: table expansion	6%	5%	11%	0%
Medium Growth: table expansion	6%	3%	10%	2%

Table 5 Top-5 ranked characterizations of commits based on the average of percentages over all datasets.

Characterizations	Mean	Standard deviation	Max	Min
High Maintenance: table restructuring	0%	1%	1%	0%
High Maintenance: intra table shrinking	1%	2%	4%	0%
High Growth: table expansion	1%	1%	2%	0%
High Maintenance: intra table restructuring	1%	1%	2%	0%
Medium Maintenance: intra table shrinking	2%	2%	4%	0%

Table 6 Bottom-5 ranked characterizations of commits based on the average percentages over all datasets

Characterization	Biosql	Ensembl	Mwiki	Ocart	PhpBB	Typo	*Mean*	*Mean w/o outliers*
Zero Logical Change	37%	44%	40%	57%	17%	24%	37%	36%
Low Growth: intra table expansion	26%	10%	13%	7%	25%	18%	17%	17%
Low Growth: table expansion	11%	0%	11%	0%	5%	10%	6%	7%
Low Maintenance: intra table amendment	20%	22%	6%	8%	33%	30%	20%	20%
Low Maintenance: intra table restructuring	7%	2%	0%	0%	10%	0%	3%	2%
Low Maintenance: intra table shrinking	7%	4%	5%	2%	10%	3%	5%	5%
Low Maintenance: table shrinking	7%	0%	4%	0%	3%	4%	3%	3%
Medium Growth: intra table expansion	0%	3%	0%	4%	3%	7%	3%	3%
Medium Growth: table expansion	4%	10%	2%	8%	6%	3%	6%	5%
Medium Maintenance: intra table amendment	4%	4%	0%	2%	4%	2%	3%	3%
Medium Maintenance: intra table restructuring	13%	1%	0%	6%	3%	3%	4%	3%
Medium Maintenance: intra table shrinking	0%	1%	0%	2%	3%	4%	2%	2%
Medium Maintenance: table restructuring	11%	2%	0%	1%	2%	0%	3%	1%
Medium Maintenance: table shrinking	0%	4%	2%	4%	6%	3%	3%	3%
High Growth: intra table expansion	0%	1%	9%	1%	1%	1%	2%	1%
High Growth: table expansion	0%	2%	1%	2%	1%	2%	1%	2%
High Maintenance: intra table amendment	2%	1%	10%	2%	2%	2%	3%	2%
High Maintenance: intra table restructuring	2%	2%	2%	1%	2%	0%	2%	2%
High Maintenance: intra table shrinking	0%	0%	4%	0%	0%	0%	1%	0%
High Maintenance: table restructuring	0%	1%	0%	1%	0%	0%	0%	0%

Table 7 Overall percentages for commit characterization

(b) Releases

Our study on releases is similar to the study on the commits. We first measure the extent of each characterization over all datasets by measuring the percentage of releases that a specific label appears over the entire life of each dataset. Again, a release can be characterized by more than one label, depending on the nature and heterogeneity of its commits. The information presented in Table 8 clearly shows that the evolution of schemata is dominated by periods of no logical activity. This means that other types of changes take place, such as index maintenance or updates in the comments of the source code.

Another interesting fact, as we already mentioned in previous section, is ***the large extent of attribute updates*** (attributes that undergo a data type change or participate in a key change). If someone examines Table 10 she will notice the large percentage ranges on *Low Maintenance: intra table amendment* and *Medium Maintenance: intra table amendment. Low Maintenance: intra table amendment* appears to all datasets and ranges between 25% - 58% with an average of 39% and *Medium Maintenance: intra table amendment* also appears in all datasets and ranges between 2% - 17%. In addition, ***it is worth mentioning that the label Low Maintenance: intra table amendment is the most used label on releases over all datasets as presented in Table 8.***

Moreover, the *Maintenance: intra table shrinking* label appears in smaller percentages than *Maintenance: table shrinking* label and both of them appear less frequent with respect to the other labels. ***This means that table deletions are more frequent than the attribute deletions from surviving tables but they appear in small numbers.*** In addition, the presence of *intra table restructuring* is significant and it ranges between 4% - 25% in *Low* label with an average of 14%, between 2% - 8% in *Medium* label with an average of 5% and between 0% - 8% in *High* label with an average of 4%. These percentages may hide attribute renames and must be examined in the future.

Based on the above observations we can conclude that the attribute updates take place more often than someone would expect with two same labels with different intensity belonging in the top-5 of labels used. In addition, the presence of Zero Logical Change label is very frequent in the releases as well. One more conclusion that arises by looking Table 6 and Table 9 is that in both commits and releases table shrinking label gathers very small percentages. This means that shrinking is less frequent than expansion.

Characterizations	Mean	Standard deviation	Max	Min
Low Maintenance: intra table amendment	39%	12%	58%	25%
Low Growth: intra table expansion	22%	10%	34%	8%
Zero Logical Change	22%	7%	33%	15%
Low Maintenance: intra table restructuring	15%	9%	25%	4%
Low Growth: table expansion	14%	6%	22%	8%

Table 8 Top-5 ranked labels of releases based on the average of percentages over all datasets.

Characterizations	Mean	Standard deviation	Max	Min
High Maintenance: intra table shrink	0%	0%	1%	0%
High Growth: intra table expansion	1%	1%	2%	0%
Medium Maintenance: intra table shrink	1%	2%	4%	0%
High Maintenance: intra table amendment	2%	2%	4%	0%
Medium Growth: intra table expansion	2%	3%	8%	0%

Table 9 Botom-5 ranked labels of releases based on the average of percentages over all datasets.

Characterizations	Biosql	Ensembl	Mwiki	Ocart	Phpbb	Typo	*Mean*	*Mean without outliers*
Zero Logical Change	33%	17%	23%	26%	18%	15%	22%	21%
Low Growth: intra table expansion	8%	32%	34%	15%	24%	21%	22%	23%
Low Growth: table expansion	8%	20%	17%	22%	9%	10%	14%	14%
Low Maintenance: intra table amendment	25%	43%	26%	37%	58%	42%	39%	37%
Low Maintenance: intra table restructuring	25%	7%	9%	22%	20%	4%	15%	15%
Low Maintenance: intra table shrink	0%	6%	7%	7%	0%	6%	4%	5%
Low Maintenance: table restructuring	8%	2%	6%	0%	0%	0%	3%	2%
Low Maintenance: table shrinking	0%	5%	5%	4%	2%	4%	3%	4%
Medium Growth: intra table expansion	0%	3%	2%	0%	0%	8%	2%	1%
Medium Growth: table expansion	0%	8%	4%	7%	0%	10%	5%	5%
Medium Maintenance: intra table amendment	17%	9%	4%	4%	9%	2%	8%	7%
Medium Maintenance: intra table restructuring	8%	5%	5%	7%	4%	2%	5%	5%
Medium Maintenance: intra table shrink	0%	0%	0%	0%	2%	4%	1%	1%
Medium Maintenance: table restructuring	8%	4%	5%	7%	13%	0%	6%	6%
Medium Maintenance: table shrinking	8%	4%	2%	0%	0%	6%	3%	3%
High Growth: intra table expansion	0%	1%	1%	0%	2%	2%	1%	1%
High Growth: table expansion	8%	0%	0%	0%	2%	4%	2%	2%
High Maintenance: intra table amendment	0%	1%	3%	4%	0%	2%	2%	2%
High Maintenance: intra table restructuring	8%	2%	1%	7%	4%	0%	4%	4%
High Maintenance: intra table shrink	0%	1%	0%	0%	0%	0%	0%	0%

Table 10 Overall percentages for release characterization

6.3.9 The extent of tangled changes

In this section we are going to answer some of our initial research goals but before we go further we need to define some terminology.

A *tangled change* corresponds to a set of unrelated modification types that take place in a specific version. *We consider that a version contains tangled changes if it is characterized with more than one label (i.e., a triplet <intensity><meta-label><activity-nature>).* In fact, this means that in the specific version different types of modifications changes are grouped together into a single version. For example, assume that a developer may have added new tables in the schema of the database and removed some attributes from tables that continue to exist in the same version. These two different types of changes belong to different modification categories and we consider the union of them as a *tangled change*.

Monothematic change. A version that does contain tangled changes consists only of one modification type that takes place in the specific version. *We consider that a version contains monothematic changes if it characterized with only one label.*

We examine the extent of tangled changes because we observed that developers commit more than one changes of different kind in the same commit. Tangled changes can threaten any analysis of the corresponding history and make the classification harder. We separate our examination on the extent of tangled changes in (a) commits and (b) in releases.

(a) Commits

In Table 11 we present the number of tangled changes along with the corresponding percentages for commits.

<u>*The extent of tangled changes.*</u> The extent of tangled changes does not seem to follow a specific pattern. Tangled changes range between 7% - 35% with an average of 18% over all datasets. We conclude that the extent of tangled changes depends on the development style of each project. With the exception of Biosql and PhpBB, we can say that for the rest of the projects, tangled commits constitute a small minority.

	#Commits	*#Tangled Changes*	*#Monothematic Changes*	*% Tangled Commits*	*%Monothematic Changes*
Biosql	46	16	30	35%	***65%***
Ensembl	526	56	470	11%	***89%***
Mediawiki	410	28	382	7%	***93%***
Opencart	411	37	374	9%	***91%***
Phpbb	229	74	155	32%	***68%***
Typo	97	12	85	12%	***88%***

Table 11 Number of tangled and monothematic changes along with their percentages per dataset.

(b) Releases

For releases we measure (a) the extent of releases that contain tangled changes and (b) the extent of releases that contain commits with tangled changes.

The extent of tangled changes. The extent of tangled changes does not seem to follow a specific pattern. Releases with tangled changes range between 29% - 51% with an average of 41% over all datasets. These percentages are way larger than the respective on commits. Of course, this is expected because inside a release the existence of commits with different change categories is very possible. The worth mentioning fact here is that on average 41% of releases contain more than one development goals.

The extent of releases that contain commits with tangled changes. As we have already mentioned, a release contains a list of commits. The aggregate measures from commits that belong to the same release can provide more than one modification types resulting to a tangled change for a specific release. For this reason, we also measure the percentage of releases that contain at least one commit with tangled changes. The percentage of releases that contain commits with tangled changes ranges between 16% - 41% over all datasets. We cannot identify any pattern on these percentages. Maybe the extent of commits that contain tangled changes inside releases depends on the development style of each project.

	#Releases	*#Releases with tangled changes*	*# Releases with tangled commits*	*#Releases with monothematic changes*	*%Releases with monothematic changes*	*% Releases with tangled changes*	*% Releases with tangled commits*
Biosql	12	4	4	8	***67%***	33%	33%
Ensembl	122	57	39	65	***53%***	47%	32%
Mediawiki	112	44	18	68	***61%***	39%	16%
Opencart	27	13	11	14	***52%***	48%	41%
Phpbb	45	23	19	22	***49%***	51%	40%
Typo3	52	15	10	37	***71%***	29%	19%

Table 12 Number of releases that contain tangled and unique changes along with their percentages per dataset.

With the minor exception of Phpbb, it is encouraging to see that the (sometimes vast) majority of releases. These percentages on monothematic changes of releases are larger than one would expect.

6.3.10 The extent of unique label

A *unique label* corresponds to a characterization or to a set of different characterizations that appear only once in a single version of the entire lifetime of the database schema. For example, let us consider that a version v_i is characterized with the labels l_1 and l_2 (again, a label is triplet <*intensity*><*meta-label*><*activity-nature*>). If this set of labels $<l_1,l_2>$ does not appear on any of the other releases, then we consider it as *unique label*.

This examination is similar to the one made for the extent of tangled changes in the previous section. Again, we separate our examination on the extent of unique labels in (a) commits and (b) in releases.

(a) Commits

In Table 12 we present the number of unique labels over the entire schema lifetime along with the corresponding percentages for all datasets.

The extent of unique labels. Unique labels range below 26% with only one exception (Biosql gathers 41%). It is possible that unique labels hide interesting insights because they may contain useful information and the changes that appear there may be prone to bugs. This would require, however, a dedicated study that fall outside the scope of this thesis; and thus, we list it as a topic of future work.

	#Commits	*#Unique Labels*	*%Unique Labels*
Biosql	46	19	41%
Ensembl	526	51	10%
Mediawiki	410	38	9%
Opencart	411	43	10%
Phpbb	229	59	26%
Typo	97	25	26%

Table 13 Number of unique changes along with their percentages per dataset

(b) Releases

The extent of unique labels. Releases with unique labels range between 35% - 67%. These large percentages spring up from the fact that the extent of tanged changes on releases is large. Tangled changes on a release means that the specific release is characterized with more than one different labels. Therefore, if there are releases with more than one label there are a lot of combinations that may be unique in the entire lifetime of a schema.

The extent of releases that contain commits with unique labels. We measure the extent of releases that contain commits with unique labels for the same reason we measured the percentage of releases that contain at least one commit with tangled changes before. Table 14 presents the number of releases that contain commits with unique labels along with their percentages. The percentage of releases that contain at least one commit with unique label ranges between 40% and 59% with an average of 48% over all datasets. This means that almost the half of the releases contain commits with unique labels.

	#Releases	*# Releases with unique labels*	*# Releases with unique commits*	*% Releases with unique labels*	*% Releases with unique commits*
Biosql	12	8	7	67%	58%
Ensembl	122	52	57	43%	47%
Mediawiki	112	39	39	35%	35%
Opencart	27	17	16	63%	59%
Phpbb	45	19	24	42%	51%
Typo3	52	26	21	50%	40%

Table 14 Number of releases that contain unique labels and number of release that contain commits with unique changes along with their percentages per dataset.

CHAPTER 7.

CONCLUSION AND FUTURE WORK

In this final chapter, we will first start with a summary of our findings and answer on our initial research questions and then we will discuss issues of future work.

7.1 Conclusions

The goal of this thesis was to combine all the various, heterogeneous, dissimilar sources of information for the history of a schema in one reference model which represents all the aspects of repository-based information. To achieve this goal, we created a reference model that combines all these different sources of information in one representation. Then, we used the defined reference model to create a system that supports both an interactive and a traditional way to exploratory analytics using the integrated contextual information about the schema histories. Beyond that, we used the same meta-model in order to group the entire lifetime of a database into phases, to which we refer to the term release, and performed a study on how these phases are related to changes affecting the schema of the database.

Moreover, given the multitude of information on the behavior of the size of the schema per release, as well as the way the changes have occurred we presented a rule-based technique that characterizes the nature of a release by inspecting these characteristics. Then within the same context, we measured the extent of each characterization over the whole history of database schemata. Based on our findings, we can argue that change is mostly absent or kept in small numbers in contrast with few releases collecting a large percentage of the changes.

7.2 Future work

In this thesis, we make a first step towards understanding the schema evolution using the external contextual information that exists in open source code repositories. In this section we present a list of interesting research extensions that need to be examined.

One of the open issues is the automation of the extraction and transformation process of data from the public repositories. In this thesis the process of extracting, transforming and loading the data is semi-automatic. Specifically, the extraction process needs manually inspection of the open source software repository to detect the location of the file with the schema definition as well as the location of different external systems. An automated process would save the researcher enough time and would also make the examination of software repositories easier.

In this thesis, we do not study how the bugs are related to commits. In another line of future work, a more sophisticated way to link bugs with commits is needed. The methodology that we propose in this thesis is naive because it requires a dedicated study that fall outside the scope of this thesis. Linking bugs with commits can provide useful insights on the reasons and motivations of a change event. Moreover, in the same context one question that arises is whether the message from commit text or the message from bug report is informative for understanding what has changed and the reason of the change.

In this thesis we used the metrics of each version in order to characterize it, without taking into consideration the text message from the commits. One question that arises from this is whether the text message from commits will improve the classification of releases or commits. Therefore, a future study could apply text mining in the commit messages to classify each commit. In the same context, sequence pattern mining algorithms can be applied on the characterizations of versions in order to discover frequent sequential pattern of modification changes. In addition, using the characterizations for each version someone can apply phasic analysis by merging sequential versions together to create more general phases from these characterizations.

Related literature studies, beyond others, the social aspect of developers that are involved in a software project and how they affect the development process. Software development is human-intensive and the study of people that are involved in the development process can help the research community to understand better the evolution process.

As we mentioned before, we handle several types of durations for releases in our data. Based on this information one resulting question is whether there is a relation between duration of a release and the nature of the changes that take place inside a release. Useful insights may arise from the study on durations of releases in the evolution of schemata and we believe that it is worth studying.

In another line of future work, a clear target of research involves detecting renames, both at the table and the attribute level. We insist that our method is fully automated, and thus, the identification of renames (that would either require a manual intervention, or, results that are not 100% certain) is a future task and out of the context of this study. A rename involves both an ejection and an injection of a renamed table or attribute, respectively.

BIBLIOGRAPHY

[BBAD09] C. Bird, A. Bachmann, E. Aune1, J. Duffy, A. Bernstein, V. Filkov, P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. Proceeding ESEC/FSE '09. pp 121-130, Amsterdam, The Netherlands, August 2009

[BBRD10] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, A. Bernstein1. The missing links: bugs and bug-fix commits. SIGSOFT FSE 2010. pp 97-106, Santa Fe, New Mexico, USA — November 2010

[CMTZ08] Carlo A. Curino, Hyun J. Moon, Letizia Tanca, Carlo Zaniolo. Schema Evolution In Wikipedia toward a Web Information System Benchmark. International Conference on Enterprise Information Systems (ICEIS '08), pp. 323-332, 2008

[DoBZ13] Q. Dong, L.Bixin, S. Zhendong. An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications. Proceeding ESEC/FSE 2013, pp. 125-135

[GoSp12] G. Gousios, D. Spinellis. GHTorrent: Github's Data from a Firehose. Proceeding MSR '12 Proceedings of the 9th IEEE Working Conference on Mining Software repositories, pp. 12-2, Zurich, Switzerland, 2012

[GVSZ14] G. Gousios, B. Vasilescu, A. Serebrenik, A. Zaidman. Lean GHTorrent: GitHub Data on Demand. MSR 2014, pp. 384-387, Hyderabad, India, 2014

[HeZe13] K. Herzig, A. Zeller. The impact of tangled code changes. MSR '13 Proceedings of the 10th Working Conference on Mining Software Repositories

[HGGH09] A. Hindle, D.M. German, M.W. Godfrey, R.C. Holt. Automatic Classification of Large Changes into Maintenance Categories. Program Comprehension, 2009 ICPC 09. IEEE 17th International Conference, pp.30-39, May 2009

[HGPS13] M.J Howard, S. Gupta, L. Pollock, K. Vijay-Shanker. Automatically Mining Software-Based, Semantically-Similar Words from Comment-Code Mappings. MSR 2013, San Francisco, CA, USA

[HiGH08] A. Hindle, D.M. German, R.C. Holt. What Do Large Commits Tell Us? A taxonomical study of large commits. MSR 2008, pp. 99-108, Leipzig, Germany, May 10-11, 2008

[KaMS07] H. Kagdi, J.I. Maletic, B.Sharif. Mining Software Repositories for Traceability Links. 15th IEEE International Conference on Program Comprehension, 2007. ICPC '07.

[KeFr14] K. Kevic, T. Fritz. A Dictionary to Translate Change Tasks to Source Code. MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 320-323, Hyderabad, India, May 31 - June 07 2014

[KGBS14] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M.German, D. Damian. The promises and perils of mining Github. Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92 – 101, May 2014, Hyderbad, India

[KiWZ08] S. Kim, E. James Whitehead Jr., Yi Zhang. Classifying Software Changes: Clean or Buggy?. IEEE Transactions on software engineering, vol. 34, no. 2, March/April 2008

[LMRW97] Lehman, M.M., Ramil, J.F, Wernick, P., Perry, D.E., Turski, W.M. Metrics and laws of software evolution - the nineties view. Fourth International, Software Metrics Symposium. pp. 20 - 32 November 1997

[RNBN15] B. Ray, M. Nagappan, C. Bird, N. Nagappan, T. Zimmerman. The uniqueness of changes: characteristics and applications. Proceedings of the 12th Working Conference on Mining Software Repositories. pp 34-44, May 2015, Florence, Italy

[Shne96] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. Proceedings of the 1996 IEEE Symposium on Visual Languages. pp.336-343, September 3-6, 1996, Boulder, Colorado, USA

[SiMG12] V. S.Sinha, S. Mani, M. Gupta. MINCE: MINing ChangE History of Android Project. MSR 2012, Zurich, Switzerland, 2012

[Sjøb91] Dag Sjøberg. Quantifying Schema Evolution. Information and Software Technology, Vol. 35, No. 1, pp. 35-44, January 1993

[SkVa13] I. Skoulis, P. Vassiliadis. Analysis of Schema Evolution for Databases in Open-Source Software. MSc, Department of Computer Science and Engineering. September 2013, Ioannina, Greece

[SkVZ14] I. Skoulis, P. Vassiliadis, A. Zarras. Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution?. 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014). 16-20 June 2014, Thessaloniki, Hellas

[SkVZ15] I. Skoulis, P. Vassiliadis, A. Zarras. Growing up with stability: How open-source relational databases evolve. Information Systems, Volume 53, pp 363 -

385 October - November 2015

[VaZa17] P. Vassiliadis, A. Zarras. Survival in Schema Evolution: Putting the Lives of Survivor and Dead Tables in Counterpoint. 29th International Conference on Advanced Information Systems Engineering (CAiSE 2017), 12-16 June 2017, Essen, Germany

[VaZS15] I. Skoulis, P. Vassiliadis, A. Zarras. How is Life for a Table in an Evolving Relational Schema? Birth, Death & Everything in Between. 34th International Conference on Conceptual Modeling (ER 2015). 19-22 October 2015, Stockholm, Sweden.

[ZWDZ04] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller. Mining Version Histories to Guide Software Changes. Proceedings of the 26th International Conference on Software Engineering, pp 563-572, May 23 - 28, 2004

APPENDICES

Appendix 1. Dataset urls and sources

In this section we describe in detail (a) the locations of the different sources from where we extract useful information, (b) the specific date and (c) the way we gathered it. All this information is represented in a table for each dataset but before we show the tables we define some taxonomy for the types of different sources:

- *SQL History*: SQL file which holds all the information about the schema of the database.
- *Commit*: file that contains commit text and information regarding the developer who made the commit.
- *Releases*: Github page that contains all the information for the releases as a free text.
- *Source Comments*: newly added comments from source code inside SQL files.
- *Issues*: source which keeps information for reported issues. These issues can be tasks or bugs and they contain free text which may contains information for database schema modifications and additional changes.
- *Builds:* contains information about the builds of a project.
- *Changes Spec*: a single file in the repository which contains free text with information about modifications and additional changes.
- *CodeReview*: an external system which helps the examination of a project's source code through the process of software development.
- *History:* a file which contains useful information about the history of a project in a free text format.

- Upgrades: file which provides an overview of the upgrade process.
- *Mailing List:* a forum or website which lets users add threads and communicate. In general, it is a system which helps the development group to communicate for the development process.
- *ChangeLogs:* contains information about the builds of a project
- *Git Message guidelines:* a file which contains suggestions about the format of commits in the version control system.

BioSQL

Repository location: https://github.com/biosql/biosql/blob/master/sql/biosqldb-mysql.sql

Source	Where?	Where?	When?	How processed?
SQL History	Github	https://github.com/biosql/biosql/blob/master/sql/biosqldb-mysql.sql	2017/03/02	Using git commands
Commit	Github	https://github.com/biosql/biosql/blob/master/sql/biosqldb-mysql.sql	2017/03/02	Using git commands
Releases	Github	https://github.com/biosql/biosql.github.io/blob/master/wiki/releases.md	2017/03/02	Using git commands
Source Comments	Github SQL files	https://github.com/biosql/biosql/blob/master/sql/biosqldb-mysql.sql	2017/03/02	Using git commands
Issues	Redmine	https://redmine.open-bio.org/projects/biosql/issues	2017/03/02	Manually exported a csv output.

Ensembl

Repository location: https://github.com/Ensembl/ensembl

Source	Where?	Where?	When?	How processed?
SQL History	Github	https://github.com/ensembl/ensembl/commits/c74bc67fc6aca5e864d7bb072373dc3251bf01b1/sql/table.sql	2017/02/03	Using git commands
Commit	Github	https://github.com/ensembl/ensembl/commits/c74bc67fc6aca5e864d7bb072373dc3251bf01b1/sql/table.sql	2017/02/03	Using git commands
Releases	Github	https://github.com/ensembl/ensembl/releases	2017/02/03	Using git commands
Source Comments	Github SQL files	https://github.com/ensembl/ensembl/commits/c74bc67fc6aca5e864d7bb072373dc3251bf01b1/sql/table.sql	2017/02/03	Using git commands
Builds	-Travis CI -Coveralls	- https://travis-ci.org/ensembl/ensembl/builds - https://coveralls.io/github/ensembl/ensembl	2017/02/03	Using TravisCI API
Changes Spec	Github	https://github.com/ensembl/ensembl/blob/release/86/docs/ensembl_changes_spec.txt	2017/02/09	Manually download

Mediawiki

Repository location: https://github.com/wikimedia/mediawiki

Source	Where?	Where?	When?	How processed?
SQL History	Github	https://github.com/wikimedia/mediawiki/commits/master/maintenance/tables.sql	2017/03/02	Using git commands
Commit	Github	https://github.com/wikimedia/mediawiki/commits/master/maintenance/tables.sql	2017/03/02	Using git commands
Releases	Github	https://github.com/wikimedia/mediawiki/releases	2017/03/02	Using git commands
Source Comments	Github SQL files	https://github.com/wikimedia/mediawiki/commits/master/maintenance/tables.sql	2017/03/02	Using git commands
Builds	Travis CI	https://travis-ci.org/wikimedia/mediawiki/builds	2017/03/02	Using TravisCI API
Issues	Phabricator	https://phabricator.wikimedia.org/	-	-
CodeReview	Gerrit	https://gerrit.wikimedia.org/r/#/q/status:open	-	Not retrieved
History	Github	https://github.com/wikimedia/mediawiki/blob/master/HISTORY	2017/02/09	Manually download.
Upgrades	Github	https://github.com/wikimedia/mediawiki/blob/master/UPGRADE	2017/02/09	Manually download.
Mailing List	-	https://lists.wikimedia.org/pipermail/mediawiki-l/	-	Not retrieved

Opencart

Repository location: https://github.com/opencart/opencart

Source	Where?	Where?	When?	How processed?
SQL History	Github	https://github.com/opencart/opencart/commits/master/upload/install/opencart.sql	2017/03/02	Using git commands
Commit	Github	https://github.com/opencart/opencart/commits/master/upload/install/opencart.sql	2017/03/02	Using git commands
Releases	Github	https://github.com/opencart/opencart/releases	2017/03/02	Using git commands
Source Comments	Github SQL files	https://github.com/opencart/opencart/commits/master/upload/install/opencart.sql	2017/03/02	Using git commands
ChangeLogs	Github	https://github.com/opencart/opencart/blob/master/CHANGELOG_AUTO.md https://github.com/opencart/opencart/blob/master/changelog.md	2017/02/09	Manually download
Issues	Github	https://github.com/opencart/opencart/issues	2017/03/02	Using GitHub API.

phpBB

Repository location: https://github.com/opencart/opencart

Source	Where?	Where?	When?	How processed?
SQL History	Github	https://github.com/phpbb/phpbb/commits/3.1.x/phpBB/install/schemas/oracle_schema.sql	2017/02/03	Using git commands.
Commit	Github	https://github.com/phpbb/phpbb/commits/3.1.x/phpBB/install/schemas/oracle_schema.sql	2017/02/03	Using git commands.
Releases	Github	https://github.com/phpbb/phpbb/releases	2017/02/03	Using git commands.
Source Comments	Github SQL files	https://github.com/phpbb/phpbb/commits/3.1.x/phpBB/install/schemas/oracle_schema.sql	2017/02/03	Using git commands.
ChangeLogs	Github	https://github.com/phpbb/phpbb/blob/master/phpBB/docs/CHANGELOG.html	2017/02/09	Manually download.
Issues	Github	https://tracker.phpbb.com/browse/PHPBB3-15078?filter=-4	2017/02/03	Manually download.
Builds	Travis CI	https://travis-ci.org/phpbb/phpbb/builds	2017/02/03	Using TravisCI API.

Version 2.0: https://github.com/phpbb/phpbb/blob/2.0.x/phpBB/install/schemas/mysql_schema.sql

Version 3.0: https://github.com/phpbb/phpbb/blob/3.0.x/phpBB/install/schemas/mysql_41_schema.sql

Typo3

Repository location: https://github.com/opencart/opencart

Source	Where?	Where?	When?	How processed?
SQL History	Github	https://github.com/TYPO3/TYPO3.CMS/blob/TYPO3_6-0/t3lib/stddb/tables.sql	2017/02/03	Using git commands.
Commit	Github	https://github.com/TYPO3/TYPO3.CMS/blob/TYPO3_6-0/t3lib/stddb/tables.sql	2017/02/03	Using git commands.
Releases	Github	https://github.com/TYPO3/TYPO3.CMS/releases	2017/02/03	Using git commands.
Source Comments	Github SQL files	https://github.com/TYPO3/TYPO3.CMS/blob/TYPO3_6-0/t3lib/stddb/tables.sql	2017/02/03	Using git commands.
ChangeLogs	Github	https://docs.typo3.org/typo3cms/extensions/core/	-	-
CodeReview		https://review.typo3.org/#/q/status:open	-	Not retrieved
Git Message guidelines		https://docs.typo3.org/typo3cms/ContributionWorkflowGuide/GitSetup/CommitMessageFormat.html	-	-
Issues	Github	https://forge.typo3.org/projects/team-docteam/issues		Manually exported a csv output.
Builds	Travis CI	https://travis-ci.org/TYPO3/TYPO3.CMS/builds/11237834	2017/02/03	Using TravisCI API.

Appendix 2. Discretization based on the intensity the activity.

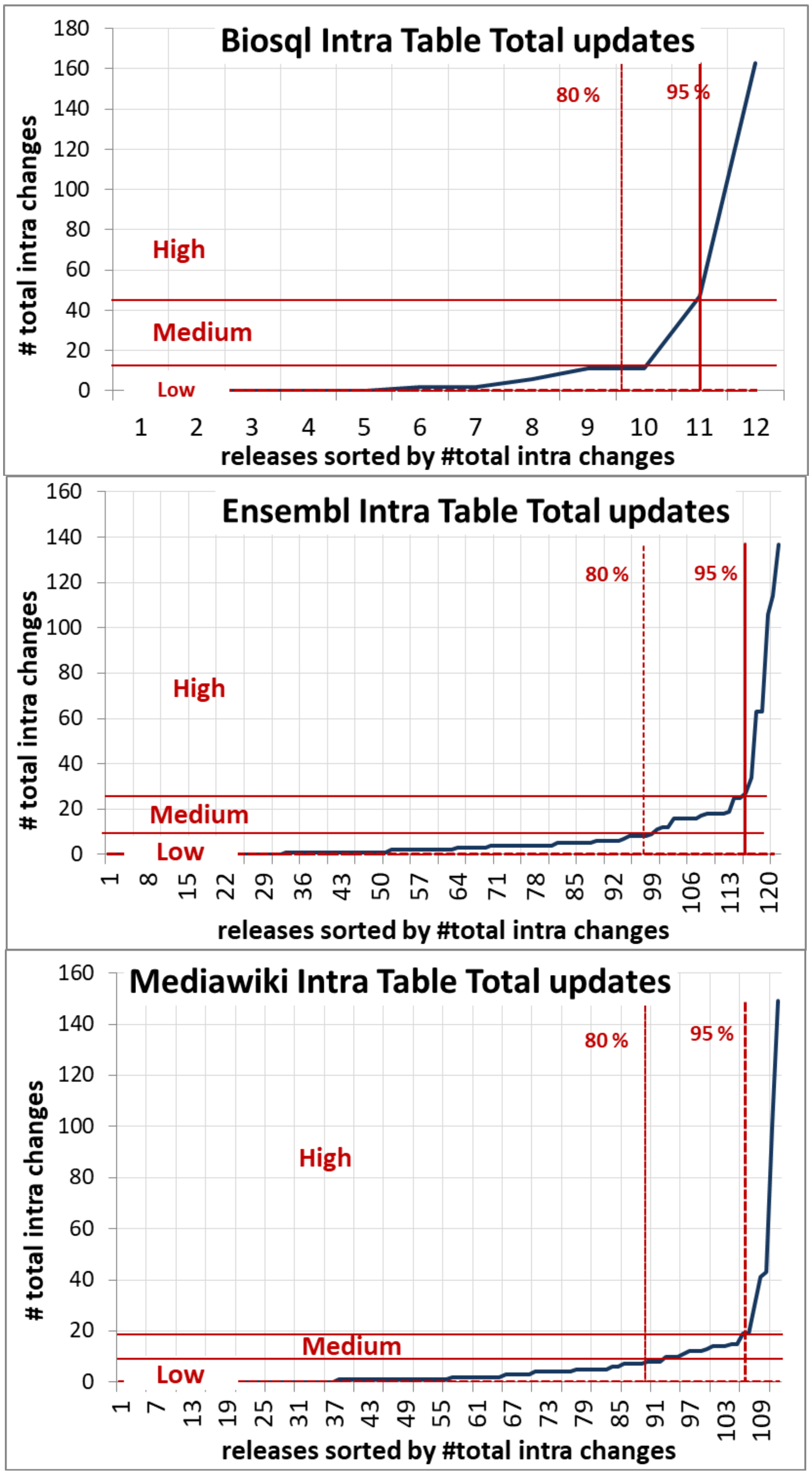


Figure 32 Distribution of total intra table updates for releases for Biosql, Ensembl and Mediawiki.

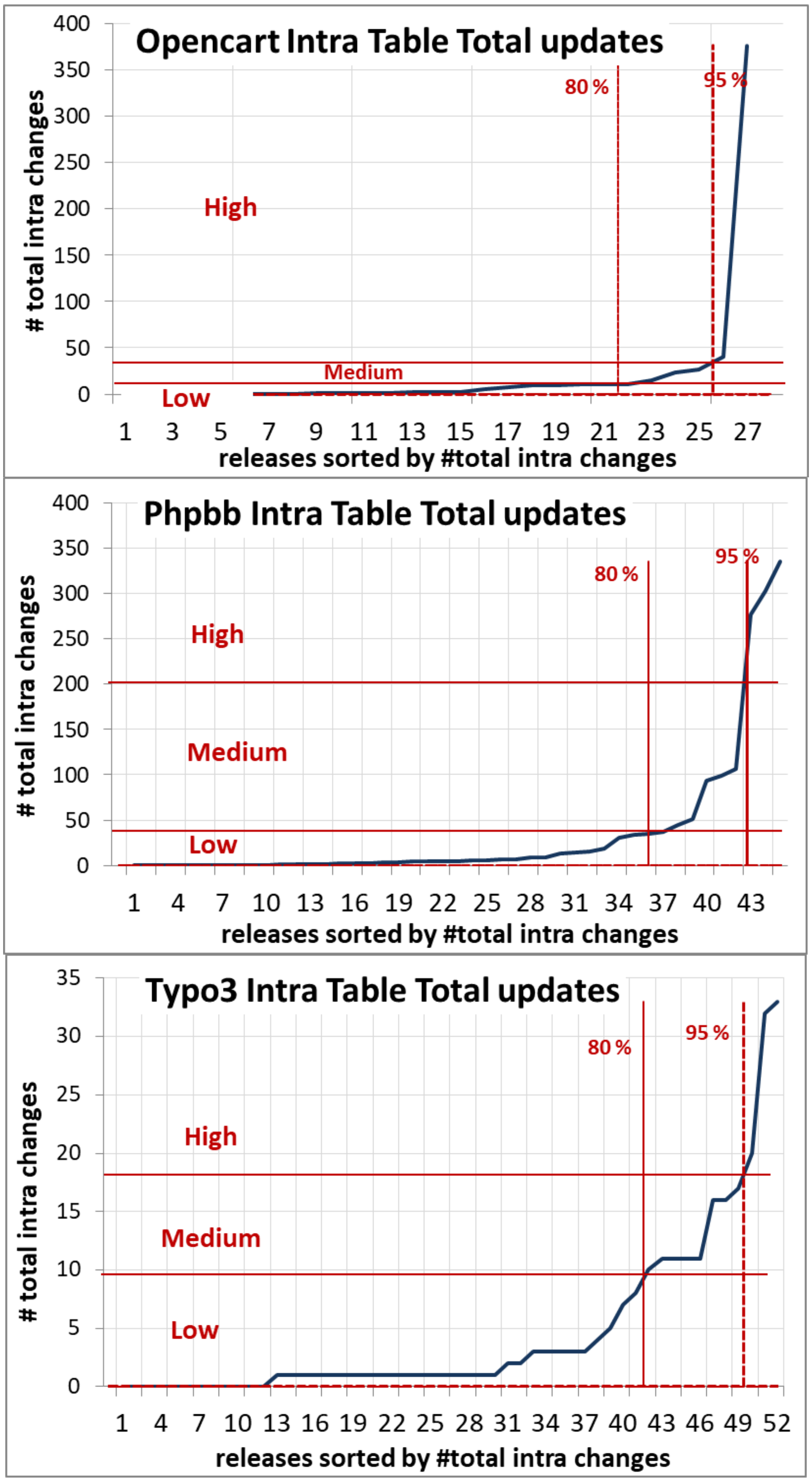


Figure 33 Distribution of total intra table updates for releases for Opencart, Phpbb and Typo3.

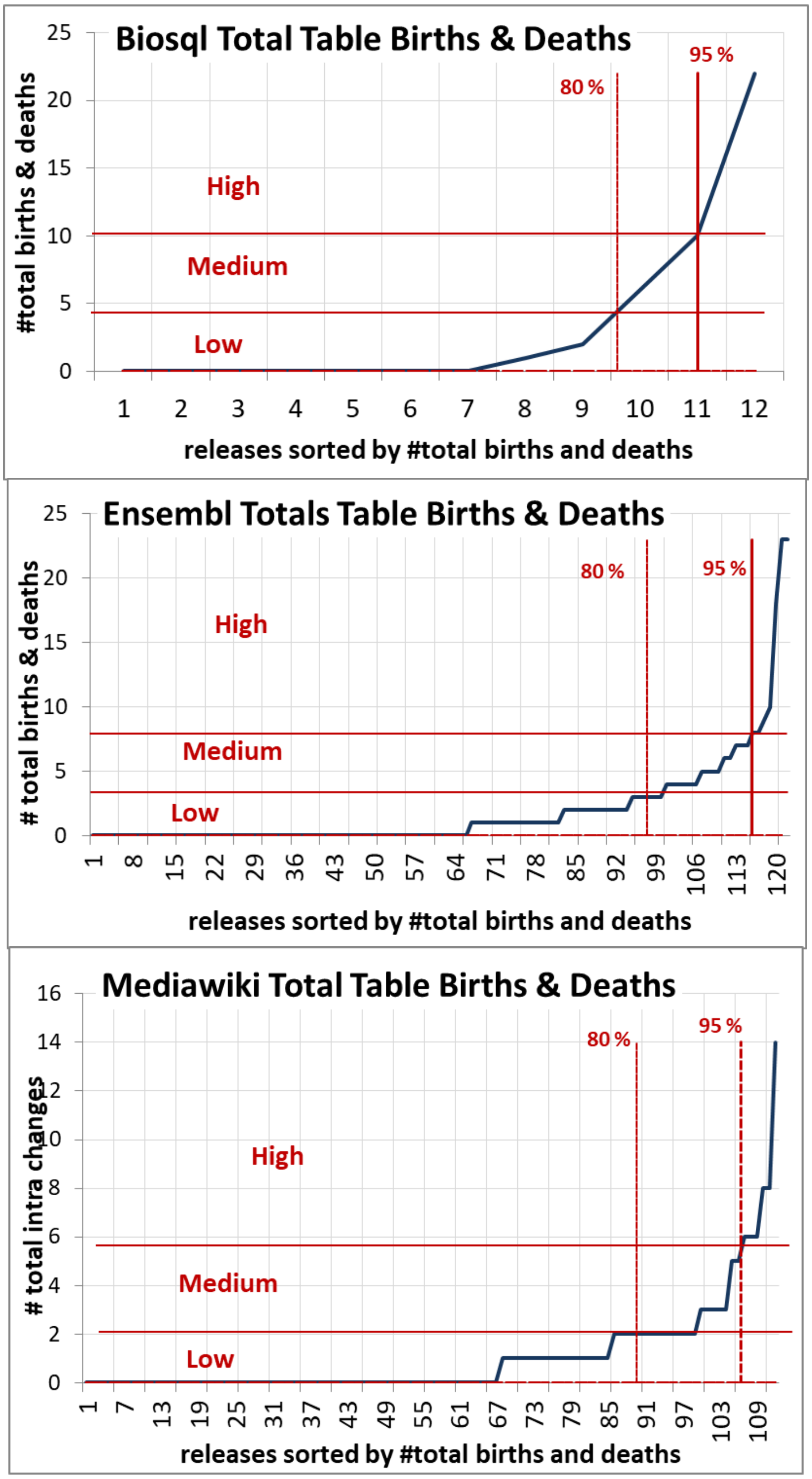


Figure 34 Distribution of total table births and deaths for releases for Biosql, Mediawiki and Ensembl.

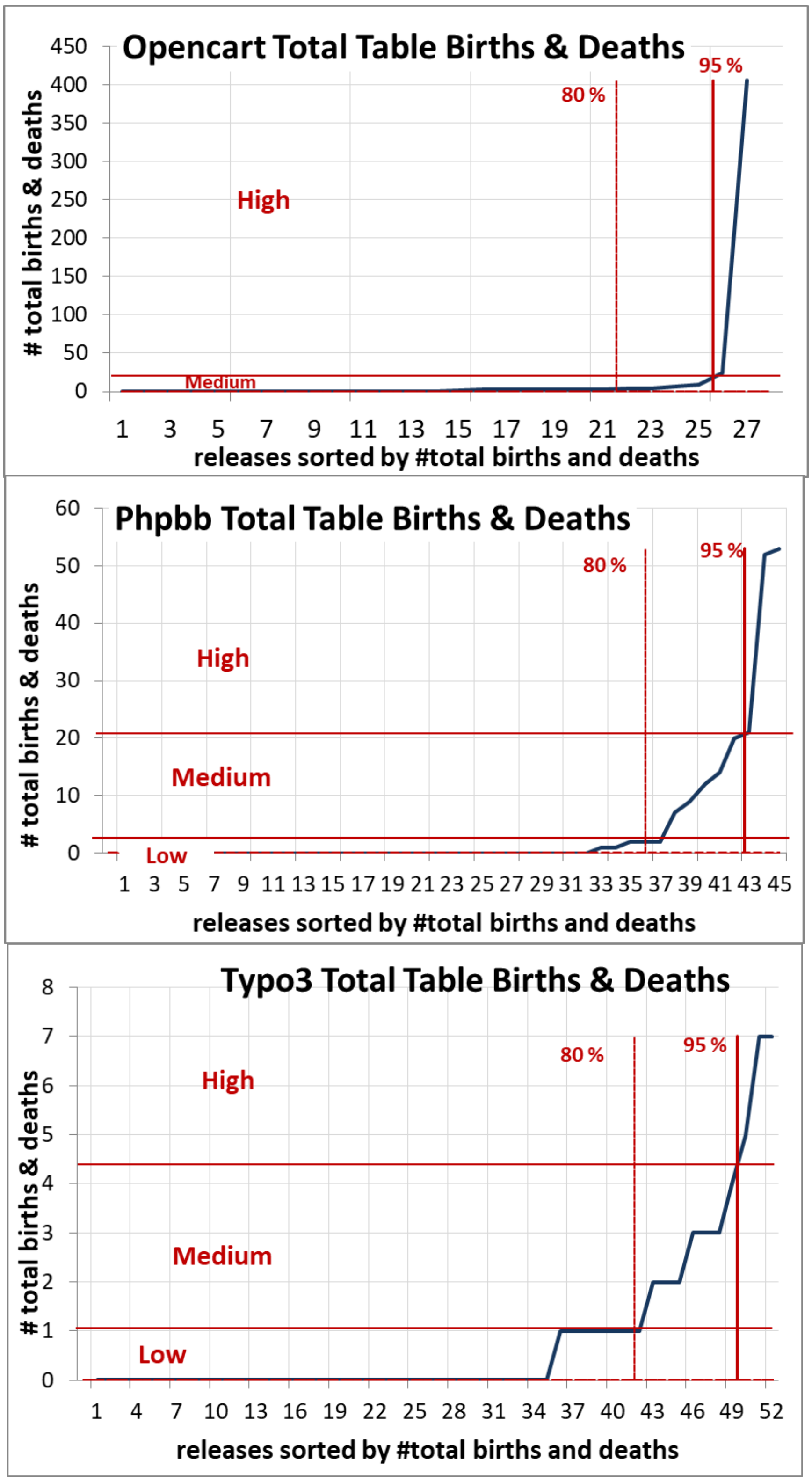
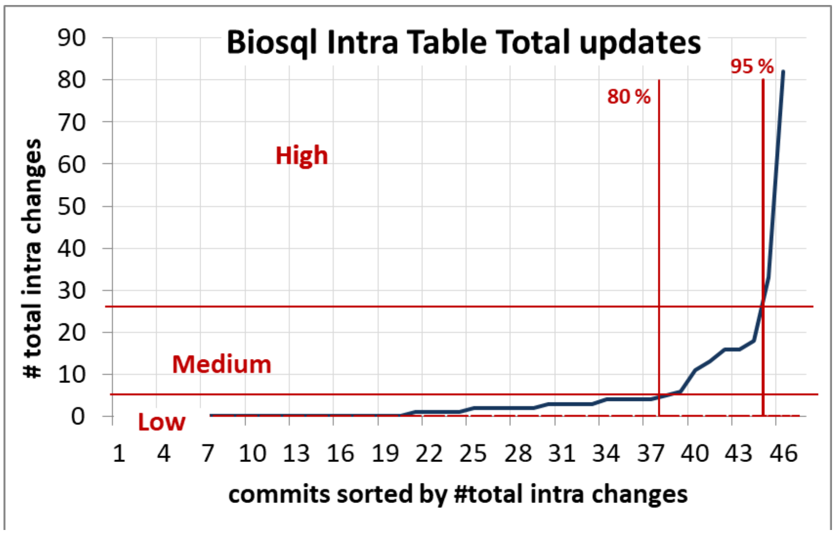
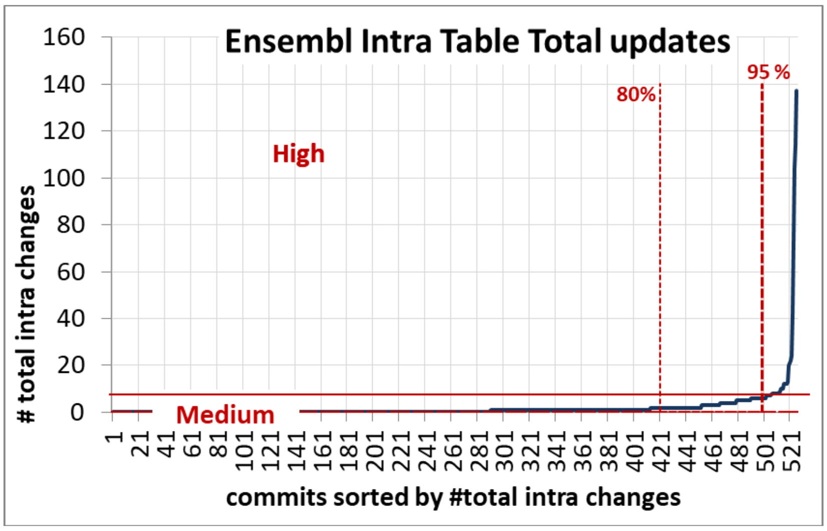


Figure 35 Distribution of total table births and deaths for releases for Opencart, Phpbb and Typo3.





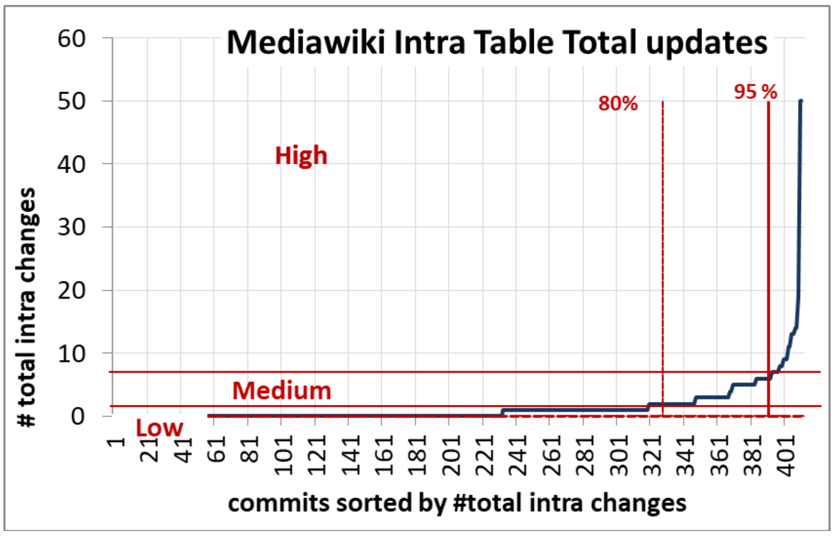
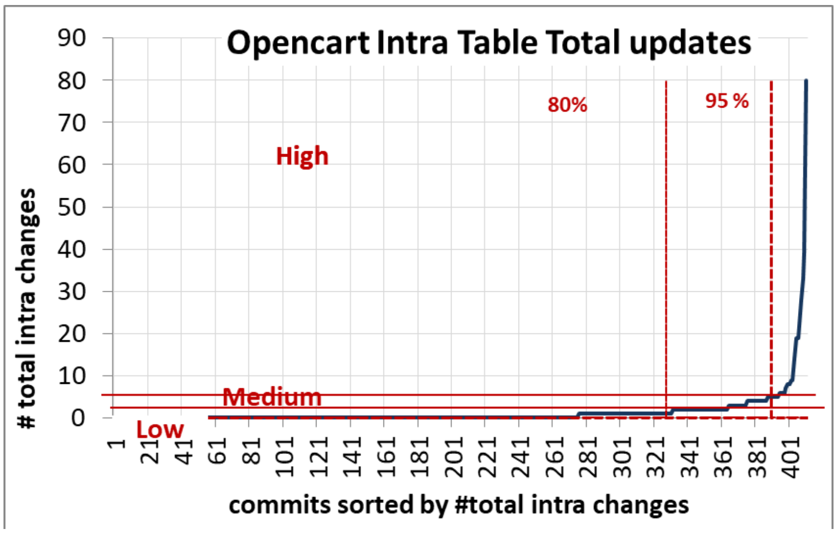
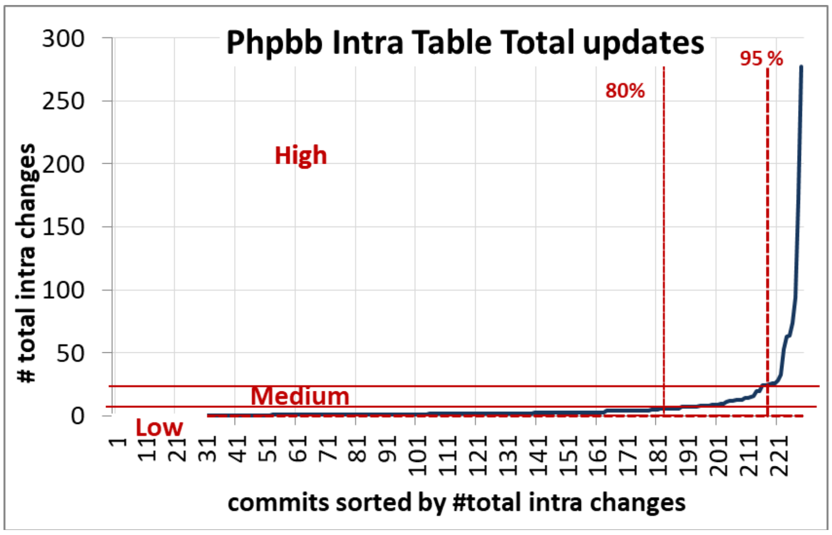


Figure 36 Distribution of intra table total updates for commits for Biosql, Ensembl and Mediawiki.





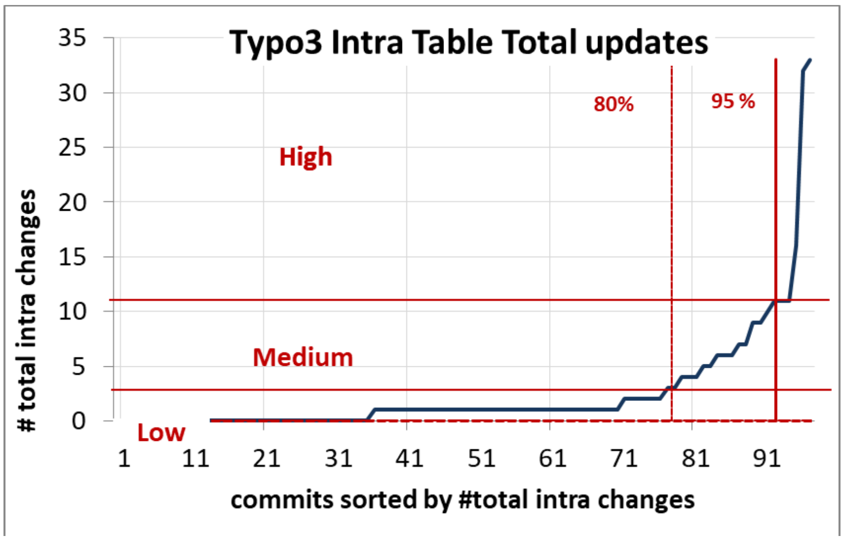
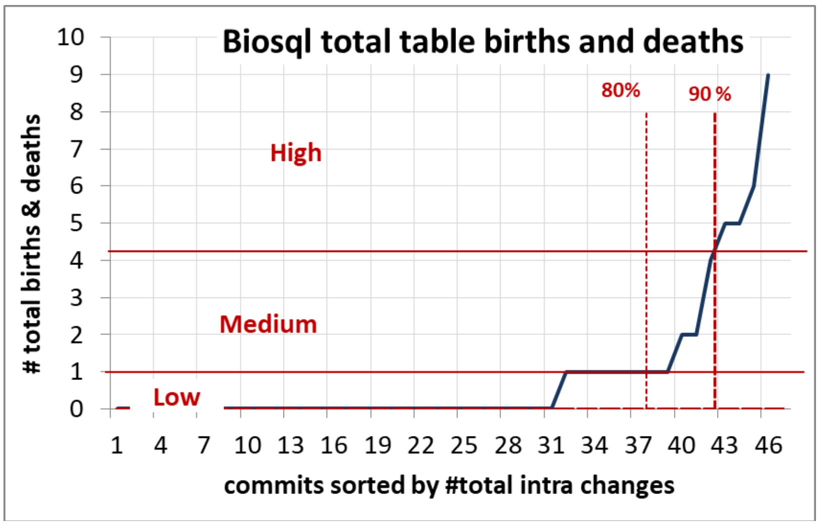
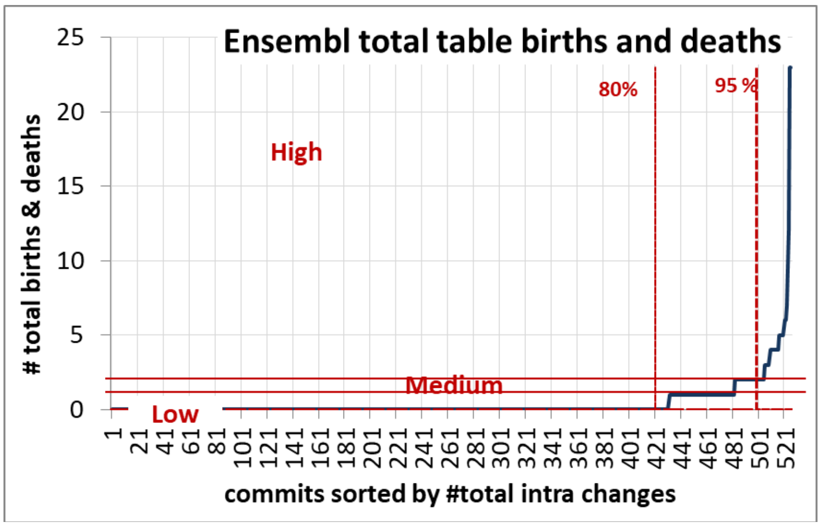


Figure 37 Distribution of intra table total updates for commits for Opencart, Phpbb and Typo3.





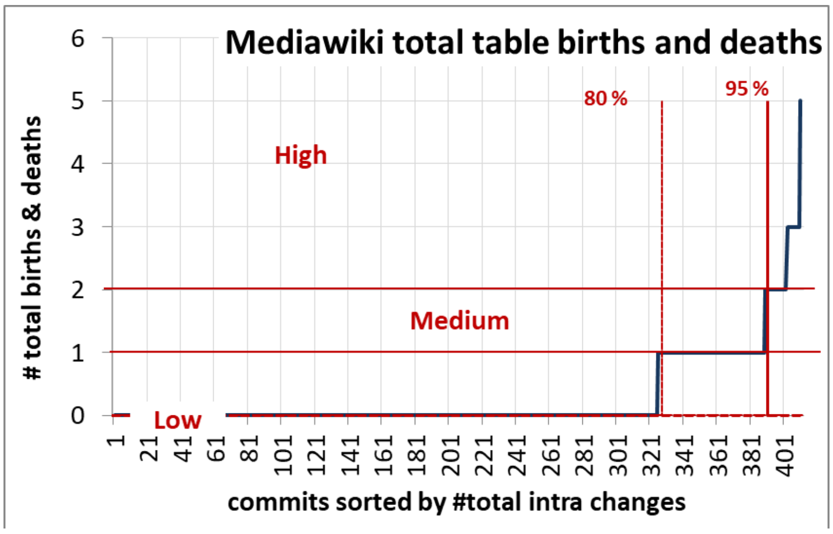
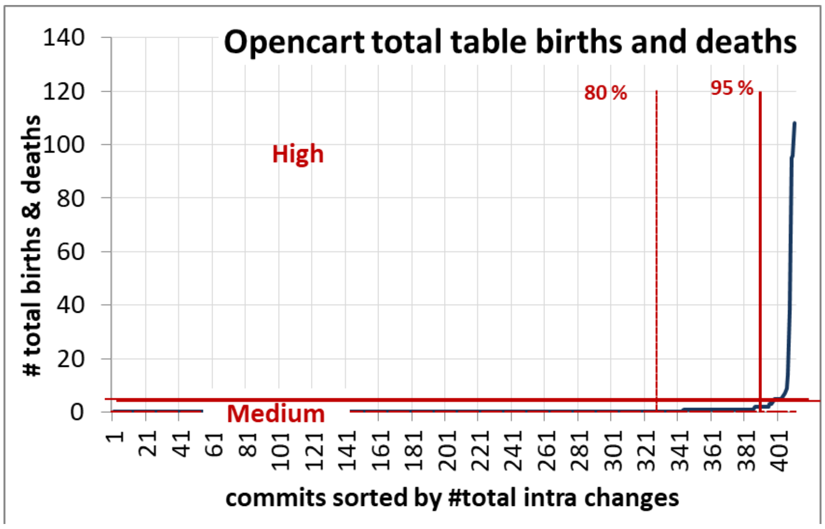
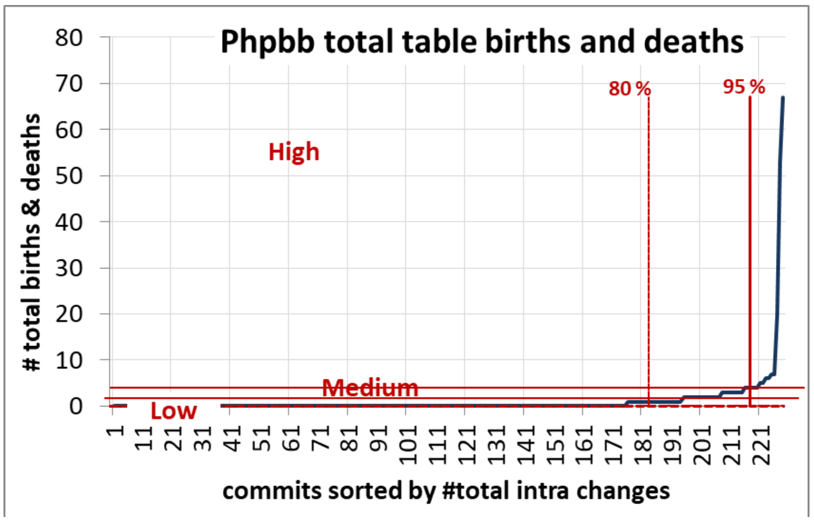


Figure 38 Distribution of total table births and deaths for commits for Biosql, Ensembl and Mediawiki.





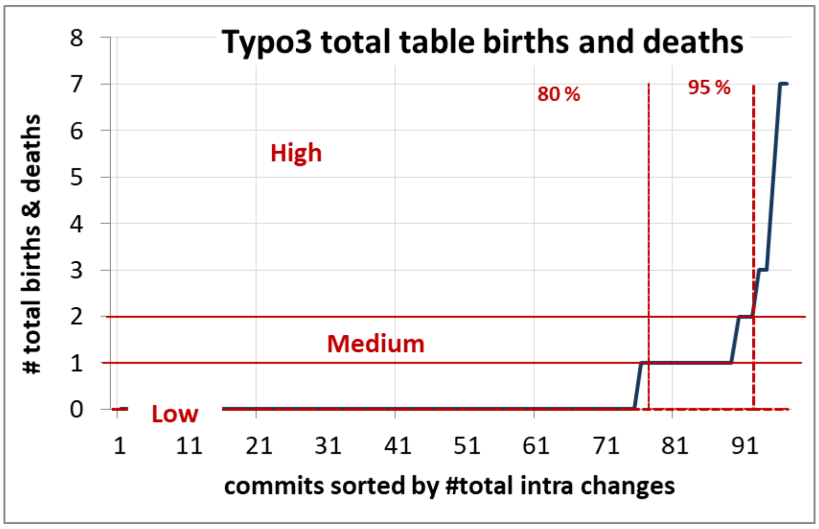


Figure 39 Distribution of total table births and deaths for commits for Opencart, Phpbb and Typo3.

SHORT CV

Athanasios Pappas was born in Ioannina in 1992. He received his BSc degree from the Department of Computer Science & Engineering of University of Ioannina at 2015. In 2015 he became an MSc student in the same institution under the supervision of Panos Vassiliadis. As a member of the DAINTINESS group, his academic interests include Software Engineering and Relational Database Evolution.