Extraction and classification of phases in schema evolution histories

A Thesis

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

by

Maria Zerva

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
WITH SPECIALIZATION
IN SOFTWARE

University of Ioannina

January 2018

Examining Committee:

- **Panos Vassiliadis**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina (Supervisor)
- **Aristidis Likas**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Apostolos Zarras**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina

DEDICATION

To my beloved grandfather.

ACKNOWLEDGMENTS

To my family, that raised me and believed in me all these years, my friends that supported all my efforts and finally my supervisor Panos Vassiliadis, for the huge inspiration and support in both my undergraduate and postgraduate studies. Thank you all very much.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Maria Zerva. MSc in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, January 2018.

Extraction and classification of phases in schema evolution histories

Advisor: Panos Vassiliadis, Associate Professor.

Software projects that are built on top of relational databases evolve over time just like any other software project. Bugs occur and user requirements change and in order to keep the users satisfied and provide consistent services, software projects have to adapt to the new requirements. The information capacity of a software project also needs to be aligned with these requirements resulting in the need to evolve the database schema along with the software. Schema evolution affects the surrounding applications in both a syntactic and semantic manner, thus making its understanding a topic of significance.

Our fundamental research question is of investigative nature: *are there phases in the lives of relational schemata*? To support our study towards answering the above question, we have used 6 free open source software projects that include relational databases, whose evolution we have tracked and for which, we have identified the changes that took place in each committed version. Based on these data, this Thesis is structured along two parts: the first part is of explorative nature, and studies the collected data to *manually* extract phases and patterns in the tables' lives, whereas the second part, proposes an *automated method* to algorithmically extract these phases.

The first part of this Thesis addresses the following question: *when are tables, attributes and foreign keys born and evicted in the life of a schema*? Based on the information on table and attribute births, deaths and updates, along with the timeline of schema size, we have manually derived phases in the life of our 6 database schemata. Our characterizations are based on the demonstrated *growth* (increase of information capacity) or *maintenance* (containing deletions and updates in order to improve the quality of the schema). The most interesting finding in our study is that, with a single exception, the history of a database schema comes in two *mega-phases*: (a) a "hot" *expansion mega-phase at the start of its life* demonstrating growth of information

capacity, along with the necessary maintenance, and, (b) a "cooling" *housekeeping mega-phase at its middle and later life* where either maintenance actions or stillness dominate the update activity. We call this phenomenon *progressive cooling of the schema heartbeat*. Several observations support this finding.

The second part of the Thesis addresses the following question: *given the history and heartbeat of a schema, can we automatically extract phases in its evolution*? Our algorithmic method includes four steps. The first step of our method, involves the characterization of the releases in terms of the two aforementioned change families, growth and maintenance. Based on these characterizations, the second step of the method splits the timeline of the schema's life in phases, by applying a hierarchical agglomerative clustering, that clusters together consecutive releases. In the third step of our method, we use several measures of clustering quality, such as Silhouette, Cohesion and Separation to characterize the discriminating quality of each of the derived clusterings. Finally, the fourth step of the method classifies clusters, i.e. phases in the life of a schema, in terms of their nature, on the basis of a taxonomy of change profiles (e.g., *Minor Activity, Restructuring, Intense Evolution,* among others).

The phase extraction and classification method introduced in the second part of this Thesis was evaluated with respect to clustering oriented measures and quality measures based on our golden standard. The findings of this evaluation show that our method performs fairly, having a small error rate and the solutions it produces are of significant quality.

ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ ΣΤΑ ΕΛΛΗΝΙΚΑ

Μαρία Ζέρβα, ΜΔΕ στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιανουάριος 2018.

Εξαγωγή και κατηγοριοποίηση φάσεων στην ιστορία της εξέλιξης σχημάτων βάσεων δεδομένων

Επιβλέπων: Παναγιώτης Βασιλειάδης, Αναπληρωτής Καθηγητής.

Το λογισμικό που είναι σχεδιασμένο βασισμένο σε μία σχεσιακή βάση εξελίσσεται με την πάροδο του χρόνου όπως οποιοδήποτε άλλο λογισμικό. Συχνά προκύπτουν λάθη ή οι χρήστες ανακαλύπτουν πώς θα ήθελαν επιπλέον λειτουργικότητες από το λογισμικό και για να είναι οι χρήστες ικανοποιημένοι και να παρέχονται συνεπείς υπηρεσίες, το λογισμικό θα πρέπει να προσαρμόζεται στις νέες απαιτήσεις. Η χωρητικότητα της πληροφορίας θα πρέπει να συμβαδίζει με τις νέες απαιτήσεις και γι' αυτό το λόγο η ανάγκη για εξέλιξη του σχήματος βάσης σε συγχρονισμό με την εξέλιξη του λογισμικού είναι ζήτημα μεγάλης σημασίας. Η εξέλιξη του σχήματος επηρεάζει σημαντικά τις εφαρμογές που βασίζονται σε αυτό και σε επίπεδο συντακτικό αλλά και σε επίπεδο σημασιολογικό. Συνεπώς, η κατανόηση της εξέλιξης του σχήματος χρήζει μεγάλης προσοχής.

Το βασικό ερώτημα που θέλουμε να απαντήσουμε στην παρούσα εργασία είναι διερευνητικής φύσεως: *υπάρχουν φάσεις στη ζωή των σχεσιακών βάσεων;* Για την διεκπεραίωση της έρευνας με στόχο την απάντηση του εν λόγω ερωτήματος, χρησιμοποιήσαμε έξι συστήματα λογισμικού ανοικτού κώδικα τα οποία εμπεριέχουν σχεσιακές βάσεις δεδομένων. Παρακολουθήσαμε την εξέλιξη αυτών των βάσεων και εντοπίσαμε τις αλλαγές που έλαβαν χώρα σε κάθε δημόσια καταχωρημένη έκδοση του λογισμικού. Με βάση αυτά τα δεδομένα, η παρούσα εργασία είναι δομημένη σε δύο μέρη: το πρώτο μέρος είναι διερευνητικής φύσεως και μελετά τα συλλεγμένα δεδομένα με στόχο την χειροκίνητη εξαγωγή φάσεων και προτύπων συμπεριφοράς στις ζωές των πινάκων, ενώ το δεύτερο μέρος, προτείνει μία αυτοματοποιημένη μέθοδο για την εξαγωγή των φάσεων με τη χρήση ενός αλγορίθμου που προτείνουμε.

Το πρώτο μέρος της εργασίας ασχολείται με την ακόλουθη ερώτηση: *πότε γεννιούνται και πεθαίνουν οι πίνακες, τα πεδία και τα ξένα κλειδιά στη ζωή ενός σχήματος;* Βασισμένοι στις πληροφορίες για τις γεννήσεις, θανάτους και ενημερώσεις πινάκων και πεδίων, μαζί με την εξέλιξη του μεγέθους του σχήματος, εξάγαμε χειροκίνητα φάσεις της ζωής των έξι συνόλων δεδομένων. Οι χαρακτηρισμοί μας βασίζονται στην ύπαρξη ανάπτυξης (αύξησης της χωρητικότητας της πληροφορίας) ή συντήρησης (περιεκτικότητας σε διαγραφές και ενημερώσεις με σκοπό τη βελτίωση του σχήματος της βάσης). Το πιο ενδιαφέρον εύρημα της έρευνάς μας είναι, ότι με μία μεμονωμένη εξαίρεση, η ιστορία του σχήματος της βάσης αποτελείται από δύο υπέρ-φάσεις: (α) *μία επεκτατική υπερ-φάση στην αρχή της ζωής του σχήματος*, η οποία επιδεικνύει αύξηση στη χωρητικότητα της πληροφορίας, και, (β) *μία υπέρ-φάση συντήρησης στη μέση ή στο τέλος της ζωής του σχήματος*, όπου κυριαρχούν είτε εργασίες συντηρήσης είτε ηρεμία στην δραστηριότητα ενημέρωσης. Ονομάζουμε αυτό το φαινόμενο *βαθμιαία ψύξη του παλμού του σχήματος.* Πολλές παρατηρήσεις υποστηρίζουν το εν λόγω εύρημα.

Το δεύτερο μέρος της εργασίας ασχολείται με την ακόλουθη ερώτηση: *δοθείσης της ιστορίας και του παλμού ενός σχήματος, μπορούμε να εξάγουμε φάσεις της εξέλιξής του με αυτοματοποιημένο τρόπο?* Η αλγοριθμική μας μέθοδος αποτελείται από τέσσερα βήματα. Το πρώτο μέρος της μεθόδου μας περιλαμβάνει το χαρακτηρισμό των δημοσίων releases του λογισμικού σε σχέση με τις δύο προαναφερθείσες οικογένειες αλλαγών, *ανάπτυξη* και *συντήρηση*. Βασισμένοι σε αυτούς τους χαρακτηρισμούς, το δεύτερο βήμα της μεθόδου σπάει το χρονοδιάγραμμα της ζωής του σχήματος σε φάσεις, εφαρμόζοντας μία ιεραρχική αθροιστική μέθοδο συσταδοποίησης, η οποία συσπειρώνει διαδοχικές releases. Στο τρίτο βήμα της μεθόδου μας, χρησιμοποιούμε μετρικές της ποιότητας συσταδοποίδησης, όπως Silhouette, Cohesion και Separation για να χαρακτηρίσουμε την ποιότητα κάθε πιθανής συσταδοποίησης. Το τέταρτο και τελικό βήμα της μεθόδου, κατατάσσει τις συστάδες, δηλαδή τις φάσεις της ζωής του σχήματος, σε σχέση με τη φύση τους, με βάση την ταξινόμηση των προφίλ αλλαγών. (δηλαδή Ασήμαντη Δραστηριότητα, Αναδιάρθρωση, Έντονη Εξέλιξη, μεταξύ άλλων).

Η μέθοδος εξαγωγής και χαρακτηρισμού φάσεων που παρουσιάστηκε στο δεύτερο μέρος της εργασίας, αξιολογήθηκε με μετρικές βασισμένες στην ποιότητα της συσταδοποίησης, αλλά και στην αναμενόμενη εξαγωγή φάσεων. Τα ευρήματα της αξιολόγησης έδειξαν ότι η μέθοδος μας έχει ικανοποιητική απόδοση, έχοντας ένα μικρό ποσοστό λαθών και οι λύσεις που παράγει είναι σημαντικά ποιοτικές.

CHAPTER 1.

INTRODUCTION

1.1 Scope

It is well-known that software projects evolve as the time passes. It is common for software developers to face the need to modify a project, because the specifications changed, an error was found, or simply because they want to add more functionalities to an existing project. Just like every software project evolves over time, so do data intensive software projects that are built on top of relational databases.

When a database has an application built around it, it needs to follow the evolution of the software in order to be consistent and fully functional. The information capacity needs to be aligned with the user requirements. When the schema evolution is not in sync with the software evolution there is a high risk of errors, as both the syntactic correctness and the semantic validity of all the surrounding applications can be significantly affected. In the former case, due the syntactic incorrectness the queries as well as their host code crash. In the latter case, the applications can suffer from loss of information, or even incorrect answers, risking the possibilities of producing results that may be misleading and inconsistent.

To understand and study schema evolution is of great importance, because exploring patterns that apply for databases can help us predict future changes. These predictions can be great help both for (a) the database design part, as we can design schemata that minimize the impact on the surrounding

software, and, (b) for the software development, as we can locate parts that need more attention by predicting future maintaining problems. It is well known that the majority of a project's resources is spend in maintenance and any knowledge that can help make this difficult procedure easier by taking the right precautions from the start is very important.

Our fundamental research question is of investigative nature and asks: "***are there phases in the lives of relational schemata?***"

To support our study towards answering the above question, we have basically used 6 free open source software projects that included relational databases. We have tracked the change history of these schemata from their public repositories, and collected information that concerns the public releases of the projects' schemata as well as their heartbeat. The *heartbeat* of change is a vector with information about addition and deletions in tables and attributes and type or key participation updates for each release.

The rest of the study is mainly based on this information and concerns two parts: the first part is of explorative nature, and studies the collected data to *manually* extract phases and patterns in the tables' lives, whereas the second part, proposed an *automated method* to algorithmically extract these phases.

The first part of this Thesis addresses the following question: "***When are tables, attributes and foreign keys born, updated and evicted in the life of a schema?***"

For each data set, and for each release, we have measured (a) the births and deaths of tables, (b) the injection of attributes to existing tables and the ejection of attributes from tables that continue to exist after the ejection and (c) the update of attribute data types and keys. We combined this information with the timeline of schema size, as it evolves over the different releases. Based on all this data, we manually derive phases in the life of our 6 database schemata. Our characterizations are based on the demonstrated *growth* (increase of information capacity) or *maintenance* (containing deletions and updates in order to improve the quality of the schema). Along with them, we highlight *spikes*, single releases of high change intensity, which are idiosyncratic characteristic of how schemata seem to evolve.

- The most interesting finding in our study is that, with the single exception of Typo3, the history of a database schema comes in two *mega-phases*: (a) a "hot" *expansion mega-phase at the start of its life* demonstrating growth of information capacity, along with the necessary maintenance and (b) a "cooling" *housekeeping mega-phase at*

its middle and later life where either maintenance actions or stillness dominate the update activity. We call this phenomenon ***progressive cooling of the schema heartbeat***.

- In the same spirit, we can also observe that *Growth* mainly takes place at the beginning of a schema's life and can be either the main focus of the developers' activity, or combined with maintenance. *Maintenance*, at the same time, takes place in all stages of the schema's life and is frequently combined with phases of minor activity, either preceding or following it.

- A third testimony of progressive cooling, comes with the observation that minor (or even zero) activity periods frequently take up long periods in time, especially at the end of the schema history

Apart from this main topic of research, we have also studied side problems. The first problem studied has to do with zombie tables, i.e., tables that are deleted at some point and later re-instated in the schema. Interestingly, after that, we observe that the majority of zombie tables tend to survive. A second question studied has to do with the period of a table's life during which injections and ejections of attributes take place. As a demonstration of the progressive cooling phenomenon, we observe that injections and ejections of attributes mostly happen at the start or mid of a table's life and rarely in the end. Moreover, we have studied how foreign keys evolve during schema evolution. Typically, the individual changes of the foreign keys are small in volume. Yet, one can observe two modes operandi for the overall treatment of foreign keys by developers, specifically, (a) foreign keys are treated as integral parts of the schema and they get born and evicted along with their tables, or (b) frequently, foreign keys are treated as second-class add-ons with small table participation in foreign keys and ad hoc foreign key births and deaths. We have witnessed the extreme case of total removal of foreign keys in two CMSs.

The second part of the Thesis addresses the following question: ***"given the history and heartbeat of a schema, can we automatically extract phases in its evolution?"***

Our algorithmic method of *Phase Extraction and Classification* is structured along four steps. We start with the heartbeat of a schema summarized at the release level and containing the sum of births and deaths of tables and attributes as well as any data type or key alterations. The first step of our method, involves the *characterization of the releases in terms of two change families: growth and maintenance*. Each release is characterized with respect to

the extent that the information capacity of the schema grows (growth) and to the extent that the internal quality of the schema is maintained (maintenance). A quantitative assessment of the amount of change per family is also produced. Based on these characterizations, the second step of the method *splits the timeline of the schema's life in phases*, by applying a hierarchical agglomerative clustering, that clusters together consecutive releases. As the split is done in a hierarchical fashion, it is clear that the quality of each of the produced clusterings has to be assessed in order to be able to pick the best, or the set of top-k such clusterings. In the third step of our method, *we use several measures of clustering quality, such as Silhouette, Cohesion and Separation to characterize the discriminating quality of each of the derived clusterings*. The fourth step of the method *classifies clusters, i.e. phases in the life of a schema, in terms of their nature*. We provide a taxonomy of change profiles, based on the intensity of the growth and maintenance that a phase demonstrates. Then, each phase can be characterized with respect to our taxonomy. The classification is based on producing the histogram of the different intensities per family {*zero*, *low*, *medium*, *high*} × {*growth*, *maintenance*}. In order to classify a phase, we apply a set of rules, that annotate the phase with a label from our taxonomy (e.g., *Minor Activity, Restructuring, Intense Evolution,* among others).

We have assessed our method over our test bed data sets. The results show that the top solutions produced by our algorithm are fairly similar to those of our golden standard, having an average misclassified release value smaller than 2. Considering the fact that we did not use any heuristics and the quality of our solutions, the method of phase extraction and classification introduced in this thesis performs well for its purpose.

To facilitate this effort, we also designed a tool that allows the collection of information, the extraction of phases and the computation and visualization of the results.

1.2 Roadmap

This thesis is structured as follows. In chapter 2, we present related work that focuses on schema evolution. In chapter 3, we analyze the first part of this study that concerns the nature of births and deaths that occur in the evolution of a database schema. Chapter 4 describes the method of phase extraction and classification we propose. Chapter 5 concludes this thesis by summarizing our findings and presents potential future work.

CHAPTER 2.

RELATED WORK

In this Chapter, we present the state of the art in the related literature on the topic of this Thesis. First we present case studies previously published in the field of schema evolution and then we compare our work to the aforementioned studies.

2.1 Case studies of Schema Evolution

[Sjob93] was the first to publish the findings of a study concerning the changes of a database schema over time and how those changes affect any software built around it. The author presents a method for measuring the modifications of database schemata and their consequences by using a thesaurus tool. The main findings of this study are:

- Additions in databases are the most common change
- Deletions are also quite common
- Renamings do not happen that much
- The automation of the handling of problems related to deletion is feasible

- The automation of the handling of problems related to additions is not that easy, as there are too many changes that need to be done by hand

In 2008, C. Curino, H.J. Moon, L. Tanca and C. Zaniolo[CMTZ08] published a study aboutMediawiki, the content management system (CMS) of Wikipedia. The problem of evolving a database schema in web information systems can be very difficult as it has a large group of contributors. The authors provide a conceptual representation for complex schema changes and software tools that help automating the analysis process. Curino et al. came to the conclusion that there is great need for better developing methods and tools to achieve schema changes with the minimum loss. The main findings of this study are:

- In Mediawiki only a small percentage (order of 20%) of the queries, that were constructed depending on the old database schema, are still valid after schema evolution.
- The tables and attributes are divided into two categories:
    - Tables of small duration, which is a result of their recent creation
    - Tables of long duration, as the cores of the schema tend to be stable throughout the whole history of the database

In 2009, Dien-Yen Lin and IulianNeamtiu [LiNe09] published the findings of a study of the co-evolution of applications and databases. The authors shed light to the well-known problem that the separation of the evolution of software and its corresponding data can lead to collateral damage. This damage may include information loss, system failure or decreasing efficiency. The main findings of this study include the following:

- The applications used to be designed based on a stable schema, something that nowadays does not happen
- The biggest problem when a system depends on a database is that it usually is considered that the version of the software is the same as the version of the database schema, while in reality the two versions are not in sync
- If the evolution of the software is not in sync with the evolution of the database schema, this will lead on an untrustworthy system

In [WuNe11], an effort to understand how dynamic updating solutions can support changes to embedded database schemas is presented. The authors

automatically extracted the schemas from software projects and also automatically computed how these schemas evolve as the applications evolve. During this study, the tool SCVD was developed, that takes the source code of all the releases, extracts their schemas, compares them and presents the results in a way that is easy to understand. The main findings include:

- Frequently, after an update has been made, the queries refer to the old schema, so we have loss of information and runtime errors
- Embedded databases have significantly less changes than large, enterprise-class databases and significantly more deletion occurrences
- Database schemas tend to change more in the initial stages of the application and progressively become stable over time

In 2012, G. Papastefanatos, P. Vassiliadis, A. Simitsis and Y. Vassiliou [PVSV12] published the findings of their study about the impact that evolution has on ETL ecosystems. The authors presented graph-theoretic metrics that help predict the effort of the ETL workflow and techniques that assess the quality of their design in terms of evolution. The software tool Hecateus was developed, that allows the monitoring of the evolution in database related environments. Since 60% of the resources of a data warehouse project is spent on maintenance, the significance of creating systems that are easy to maintain is quite obvious. The main findings of this study include:

- The size of the schema and the complexity of its parts are factors that make the system vulnerable to changes
- A good design pattern is one with tables of a small schema with few attributes, because the more the attributes the more the levels of vulnerability
- If these metrics are applied by the ETL designers the maintaining process will become significantly easier

D. Qiu, B. Li and Z. Su in [QiLS13] study the co-evolution of database schemas and code in ten open-source database applications. One of the main research questions of this study concerned the frequency and extent of database schema evolution. By examining the occurrences of schema changes the authors concluded that schemas evolve frequently. The authors also studied the nature of database schema evolution and to do so they categorized the atomic schema changes. They found that the main high-level schema changes are transformations, structure refactorings and data quality

refactorings, while architectural refactorings took place relatively infrequently. The low-level most frequent atomic change types were *add table*, *add column* and *change column datatype*. The authors also observed that referential integrity constraints and procedures are rarely used in practice, while additions and changes were the most common cases of schema evolution. Finally, an additional research question that the authors answer, investigates how the code co-changes with a schema change. The corresponding findings show that more than 70% of all valid DB revisions contained effective co-change information, schema changes impact code greatly and the changes that show more significant impact on application code are transformations and structure refactorings.

I.Skoulis, P. Vassiliadis and A. V. Zarras in 2014 [SkVZ14] published their findings of an open-source relational database evolution study. The authors performed a thorough study on the evolution of database schemas of publicly available projects and used Lehman's Laws of software evolution as a guideline for the schema evolution. Even though Lehman's laws cannot be entirely matched to databases, their significance is of the same importance and they are very helpful in monitoring schema evolution. The main findings of this study are:

- Schemas evolve in bursts, in grouped periods of evolving activities and not in a continuous process, which means that the first law of Lehman can be partly applied to databases
- The second law of increasing complexity seems to be quite applicable to databases too
- The third law of self-regulation also applies for databases, except for the fact that changes do not follow a smooth evolution pattern, but the presence of feedback is obvious
- Evolution of databases, even in phases, is not stable so the law of conservation of organizational stability does not apply to databases
- Even though the conservation of familiarity is significant, it does not guarantee incremental growth, so this law is possible but not confirmed
- The law of continuing growth, when it is adapted to the particularities of database schemas, applies to databases
- The law of declining quality is uncertain

- The law of feedback system applies, as the evolution of a schema obeys the behavior of such a mechanism

P. Vassiliadis, A. V. Zarras and I. Skoulis in 2015 [SkVZ15] performed a study that focuses on the behavior of tables during the evolution of a database schema. The authors studied whether table characteristics, like number of attributes or time of birth can be related to chances of deletion, amount of changes etc. The main findings of this study include:

- Thin tables with small schema size have unspecified life duration
- Wide tables with larger schema size tend to live longer
- The tables that are removed are mostly newborns, that get deleted quickly, with few or no updates
- Tables of medium or big duration do not get deleted that often
- The rest of the tables live a quite calm life

P. Vassiliadis, A. V. Zarras and I.Skoulis in [VaZa17] published their findings on the categories of tables that evolve as the schema evolves and the nature of this evolution. The authors' findings relate to the relevance of table properties to evolution related ones. More specifically they categorized the tables according to their survival or death in three main categories. The first one is "wide survivors", that includes tables with large size that tend to live longer and survive. This behavior was introduced by the authors as the Γ pattern, that concerns the relation of schema size with duration. The second category is "entry level removals", that includes newly born tables, quickly removed, or/and with no or few updates. The last category is "old timers", which includes the tables with long duration that rarely are deleted. The relation of duration and birth can be described by the "Empty triangle" pattern, which means that there are very few cases of tables not born from the start that do not survive or have a long duration. As far as the tables that are prone to updates are concerned, the authors observed two different patterns. The first one is called the "inverse Γ" pattern and states that the duration of the lifetime of a table is not proportional to the amount of updates that the table endures. The second one, the "Comet" pattern was revealed due to the correlation of schema size at the birth of a table with its update profile and states that most small-sized tables have few to none changes, some medium-sized tables have many changes and wide tables have medium amount of updates. Finally, the main finding of this study was the gravitation to rigidity in database schemas, which is a tendency to avoid change and evolution.

P.Vassiliadis et al., in [VKZZ17], published the findings of a schema evolution study focused on foreign key evolution, in the broader context of schema evolution for relational databases. The authors explored the nature of growth and heartbeat of foreign keys and designed a software tool that represents, visualizes and measures the foreign key evolution. The findings of this study show that foreign keys are in some cases an integral part of the system, at least in the cases of scientific nature projects, where they use to be born and removed along with their tables. The authors also observed cases of projects with foreign keys that seem to be unwanted and removed mostly not in the same time with their corresponding tables. The datasets collected for this study contained two cases of Content Management Systems and in both of these cases the foreign keys were completely removed from them in the last known to the authors' version of the schema. This behavior seems to be a result of difficulty of managing technical issues with foreign keys. Finally the authors observed that changes in foreign keys are not so common and when they do exist, they are mostly small in volume.

2.2 Comparison to the state of the art

Despite the achievements of the previous efforts, currently, we have no structured and well-founded knowledge of integration and organization of the heartbeat of a database schema in a principled way. To the best of our knowledge no other study has explored thoroughly the nature of births and deaths of tables and attributes in the life of a relational database schema. Furthermore, the organization of a schema's life into phases that are based on the changes the schema undergoes has not been studied in the literature.

CHAPTER 3.

BIRTH AND DEATH IN SCHEMA EVOLUTION

In this chapter, we present the first part of this study that explores the births and deaths that occur in a database schema. In the first section we describe the experimental setup in which this study has taken place. The second section explores the nature of births, deaths and updates that occur in a schema's life. In the third section we present some special topics and more specifically) the results that concern the tables that get deleted sometime in the evolution and later on they get reborn, b) the time period in a table's life when most attributes get injected or ejected and finally, c) a study of the commit histories of six open source projects that contain foreign keys and explores their nature and behavior. The last section presents the conclusions of this part of the study.

3.1 Experimental Setup

In this study twelve different open source projects were used. These datasets are: Atlas, Biosql, Castor, Coppermine, Egee, Ensembl, Mediawiki, Opencart, Phpbb, Slashcode, Typo3 and Zabbix. Three of the databases, Atlas, Castor and Egee are hosted by CERN, the European Organization for Nuclear Research based on Geneva. Biosql and Ensembl contain medical information, while all the others are part of a Content Management System (CMS). Each part of the study uses the datasets that have the needed features. For example, of all the datasets Castor, Ensembl, Opencart, Slashcode and Phpbb were the only ones that had over 10 tables that get deleted and reborn, so the respective section describes the behavior of these datasets. The last part of this chapter that studies the foreign keys takes place for the six datasets that contain foreign keys. When it is feasible, in terms of the nature of the study, we take all datasets into consideration (for example attribute injections and ejections in time).Several basic statistics for each dataset are shown in Figure 1.

All the statistics and numbers we use concerning releases/commits, we got by the differences in the schemas between every two consecutive commits/releases.

Dataset	#tables@start	#tables@end	#attributes@start	#attributes@end	#commits	#releases
Atlas	56	73	709	858	84	*N/A*
Biosql	21	28	74	129	46	12
Castor	62	74	632	838	194	*N/A*
Coppermine	8	22	87	169	117	*N/A*
Egee	6	10	34	71	17	*N/A*
Ensembl	17	75	75	486	528	122
Mediawiki	17	50	100	318	322	112
Opencart	46	114	292	731	164	27
Phpbb	61	65	611	565	133	45
Slashcode	42	87	259	610	399	*N/A*
Typo3	10	23	122	414	97	52
Zabbix	15	48	81	306	160	*N/A*

Figure 1 Table with statistics for all datasets.

3.2 Births, Deaths and Updates

In this section we explore the nature of births, deaths and updates that occur during the evolution of the schema of six open source software projects. In this specific part of the study we use the information that concerns the public releases of the projects' schemata collected in [Papp17].

To explore the nature of the evolution we study the heartbeat of change of each dataset. The heartbeat of change is a vector with information about addition and deletions in tables and attributes and type or key participation updates for each release. In the following charts we visualize the heartbeat via a combination of barchart, linechart and scatterchart that depicts the amount of births, deaths and alternations in schema size that a schema's releases undergo through time.

Figures 3-14 show the heartbeat of change in terms of table birth and death, attribute injection and ejection, type and key participation updates and schema size in number of tables and attributes for each one of the studied datasets. The x-axis of the figures represents the name of the schema's release or the date of the last commit of the corresponding release.

Terminology. Studying these figures we try to observe the phases that each dataset undergoes. We characterize these phases with the following terminology.

- A phase of *growth* shows increase in the schema size and its heartbeat mainly concerns additions to information capacity.
- A phase of *maintenance* contains deletions and updates and does not really increase information capacity.
- We characterize a phase as a *minor activity* phase when there is zero or low activity and there are no significant increases or decreases in schema size in tables and attributes.
- The combination of *growth and maintenance* concerns the demonstration of both kinds of activity in a phase, and sometimes it comes with a renaming or restructuring manner.
- A phase of *intense evolution* is actually a *growth + maintenance* phase with very high volume of change in number of attributes.
- Finally, it should be noted that when we refer to the notion *spike* we mean a phase that contains mostly one single release of very high volume of change, which could be considered a phase by itself.

As mentioned above, the main purpose of this study is to explore the nature of evolution of the studied schemata. We proceed in detailing the observations around the evolution of each data set.

Biosql. As we can see from Figures 3 and 4 Biosql, in the 10 years that we have available for study, seems to have a very quiet and almost non-existent evolution. Studying the changes Biosql undergoes in terms of table birth and death, attribute injections and ejections and attribute type or key participation updates, we assume that the history of Biosql could be divided in three phases. There is the first phase of *maintenance*, where the administrators deleted tables and injected attributes and altered others. Then there is the second phase that could be considered as a single spike where there was clear effort from the administrator's/developer's point where we have all kinds of changes in a single release and in significant volume, thus we have a phase of *intense evolution*. Finally, in the last phase, which is almost ten years nothing happens except from some attribute type update and the phase is characterized as *minor activity*.

Ensembl. For Ensembl, this is clearly not the case. Ensembl, in the 17 years of evolution that we have available, is clearly a more vivid and intense dataset in terms of evolution. As we can see from Figures 5 and 6, taking into consideration all the kinds of changes depicted in the aforementioned charts, we assume that a possible segmentation of Ensembl's history in phases could consist of eight discrete phases. The first phase is clearly a case *of growth with maintenance*, as it shows high growth effort both in tables and attributes, while it also contains a few deletions and a significant amount of updates. The second phase shows *intense evolution* with significant additions and deletions. Then, we notice a short period of *minor* to none *activity*, which is followed by an *intense evolution* phase with attribute deletions and updates and high growth in tables and attributes. The fifth phase is considered as a medium *growth* period with some deletions of *maintenance*, while the sixth phase only shows *minor activity*. Then, we notice a period of *maintenance* with medium volume with some deletions and updates and finally the last phase shows quiet behavior with very few updates and is quite long in time and is characterized as *minor activity*.

Mediawiki. The next dataset, Mediawiki, has an evolution of 13 years and is also quite intense with no significant periods of calmness. As seen in Figures 7 and 8, we assume that Mediawiki's history could be divided in seven phases. First we have a period of *growth*, which is quite common in the start of projects, both in terms of tables and attributes. The second phase is a period of *maintenance* with a significant amount of both additions and deletions. Then, we notice a long period of slow *growth* with some spikes of updates. The fourth phase shows intense *maintenance* with a lot of additions and deletions depicted by the spikes in the aforementioned figures. After this intense

maintaining period, there is a period of *minor activity* where nothing significant occurs and right after that, a phase of *maintenance* appears. Finally, there is a period of *minor activity*, with very few changes in the schema.

Opencart. For Opencart we have less than 4 years of releases available and it is one of the quietest of our datasets. As it is depicted in Figures 9 and 10, Opencart's history can be divided in 4 phases. The first phase is considered the first release alone, which contains a huge amount of births deaths and updates and it is the only one with this kind of volume in the whole life of Opencart and it is characterized as *maintenance*. The second phase is a *minor activity* phase and it is quite short period, where almost nothing happens and right after that we have a *maintenance* phase with a quite significant amount of additions and some deletions. Finally, the last phase is of *minor activity* and is very long in terms of time and number of releases and it consists of only minor updates.

Phpbb. The next dataset, Phpbb, in the 11 years of releases available to us, seems to be a calm dataset with a significant amount of spikes. In Figures 11 and 12, we see that its history can be divided in five phases. The releases of the first phase are considered as *minor activity* and have zero activity and in the second phase we have a spike with all kinds of changes and the phase is characterized as *growth* with *maintenance*. The third period has a *maintenance* nature and contains two intense spikes of additions and deletions and quite a few spikes of updates. In the fourth *minor activity* phase we notice only a few changes, it is a three year period of calmness and right after that, we have a period of *maintenance* with significant amount of all possible updates.

Typo3. Our last dataset, Typo3, has an evolution of circa ten years. As seen in Figures 13 and 14, we assume a segmentation of its history that consists of six phases. As usual, the first phase contains releases with *growth* nature in tables and attributes and releases with a lot of updates. The second phase is a *minor activity* phase and it is a quite long period of calmness. Then, we notice a short *growth* period and right after that a period of *minor activity* again. The fifth phase consists of releases with intense *maintenance* behavior with a lot of deletions and a few updates. Finally, we notice a period of *intense evolution* with a lot of additions and deletions both in terms of tables and attributes and also a few updates.

	BioSQL	Ensembl	Mediawiki	Opencart	Phpbb	Typo3
1	Maintenance	Growth+ Maintenance	**Growth**	Maintenance (spike)	Minor Activity	**Growth**
2	**Intense Evolution (spike)**	**Intense Evolution**	Maintenance	Minor Activity	Growth+ Maintenance (spike)	Minor Activity
3	Minor Activity	Minor Activity	**Growth**	Maintenance (spike)	Maintenance	**Growth**
4		**Intense Evolution**	Maintenance	Minor Activity	Minor Activity	Minor Activity
5		Growth+ Maintenance	Minor Activity		Maintenance	Maintenance
6		Minor Activity	Maintenance			**Intense Evolution**
7		Maintenance	Minor Activity			
8		Minor Activity				

Figure 2 Table of the phases of all datasets.

As discussed above and seen in Figure 2, common patterns are noticeable and seem to occur for most of the datasets. Those patterns are:

- *Growth*(colored green in Figure 2) mainly happens in the start of a dataset's life for most of the cases (with the single exception of Ensembl, having a growth phase in its mid-life).Growth can come wither solo (as the main content of update activity – colored as solid green background in Figure 2) or it can also be accompanied by *maintenance.*
- *Maintenance* (denoted as text in blue font in Figure 2) can be found in all possible stages of schema's life, in practically all data sets.
- *Maintenance* followed or preceded by *minor activity* (the combination coming as mid-intensity blue background in Figure 2) seems to be the most common pattern, whose occurrences are overwhelmingly located

in the end of a schema's life and accompanied with low or zero growth in information capacity.

- Practically, with the single exception of Typo3, we can safely argue that the history of a database schema comes in two *mega-phases*: (a) a "hot" *expansion mega-phase at the start of its life* demonstrating growth of information capacity, along with the necessary maintenance and (b) a "cooling" *housekeeping mega-phase at its middle and later life* where either maintenance actions or stillness dominate the update activity.
- For the majority of the datasets, minor (or even zero) activity periods frequently take up long periods in time, especially at the end of their history
- A few datasets though, have *intense evolution* with changes of significant volume

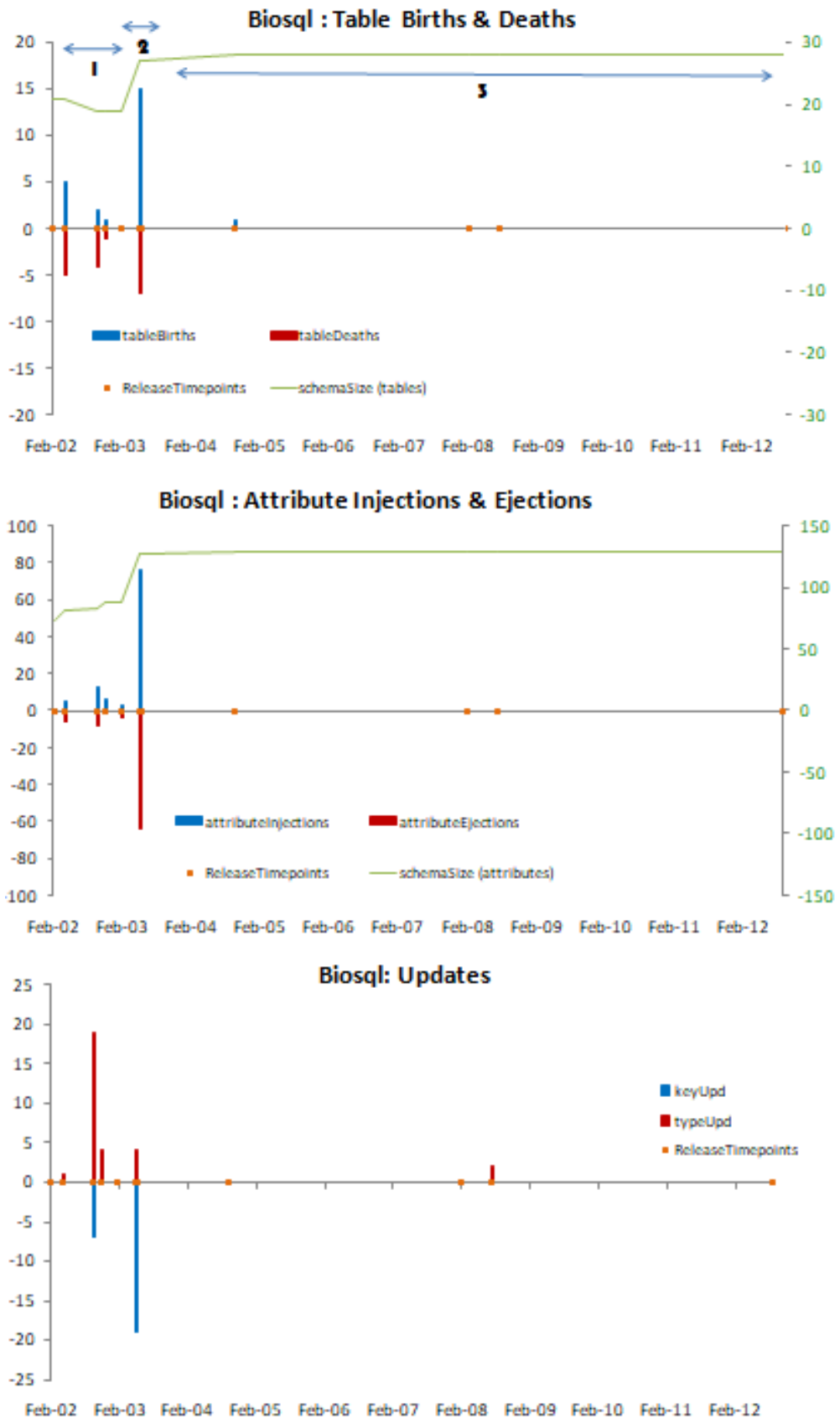


Figure 3 Biosql :Heartbeat of change in time.

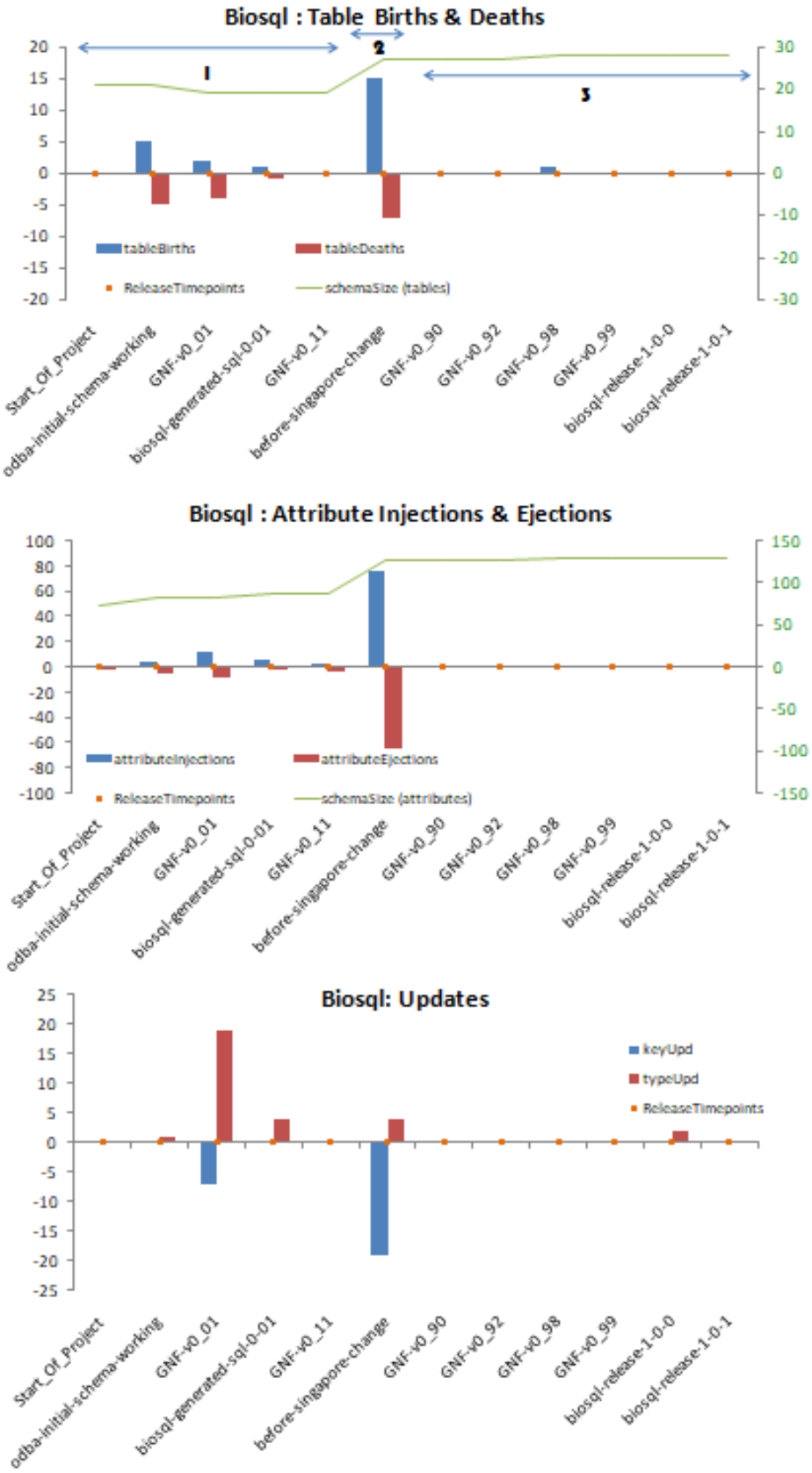


Figure 4 Biosql : Heartbeat of change per release.

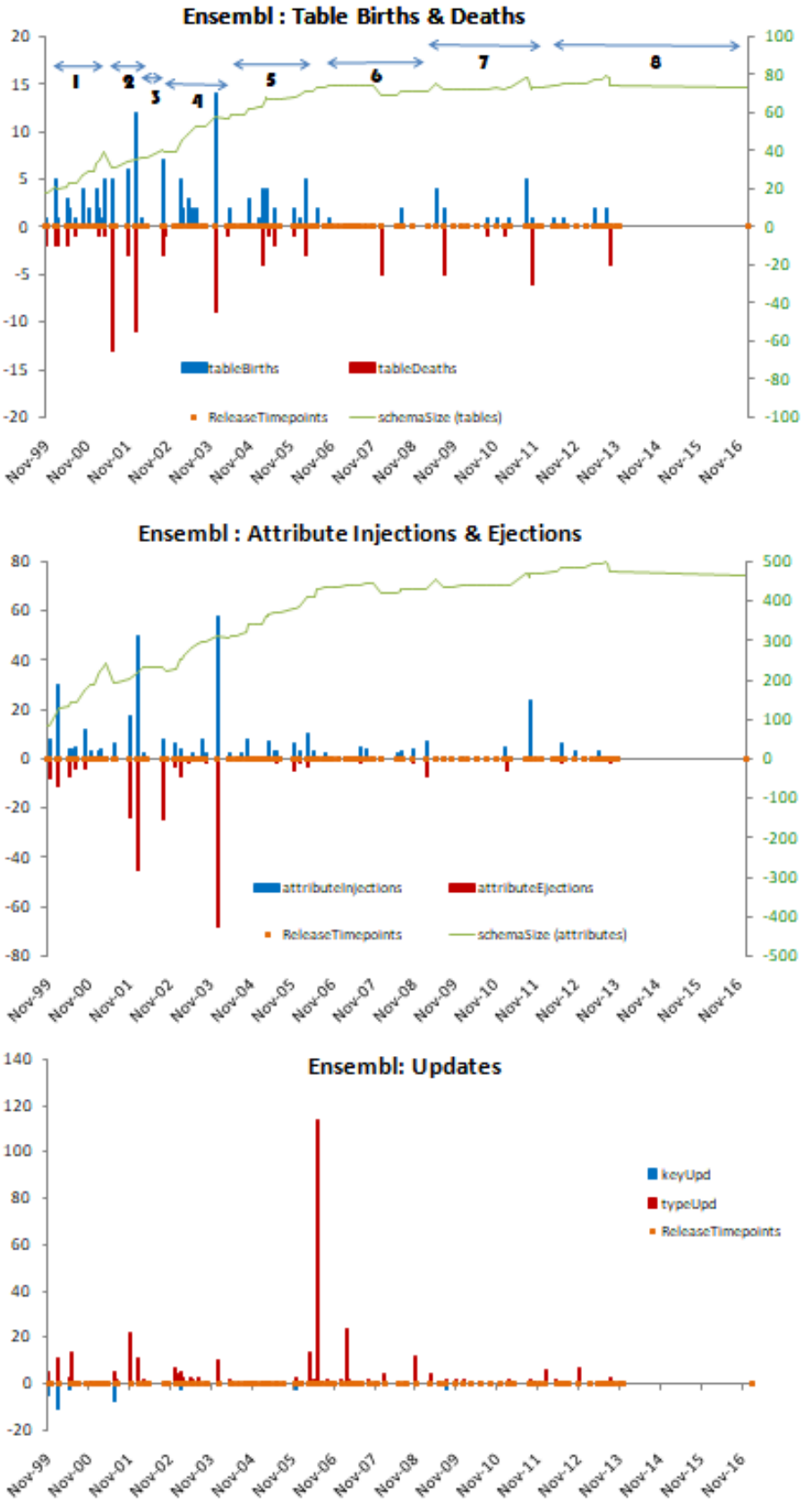


Figure 5 Ensembl : Heartbeat of change in time.

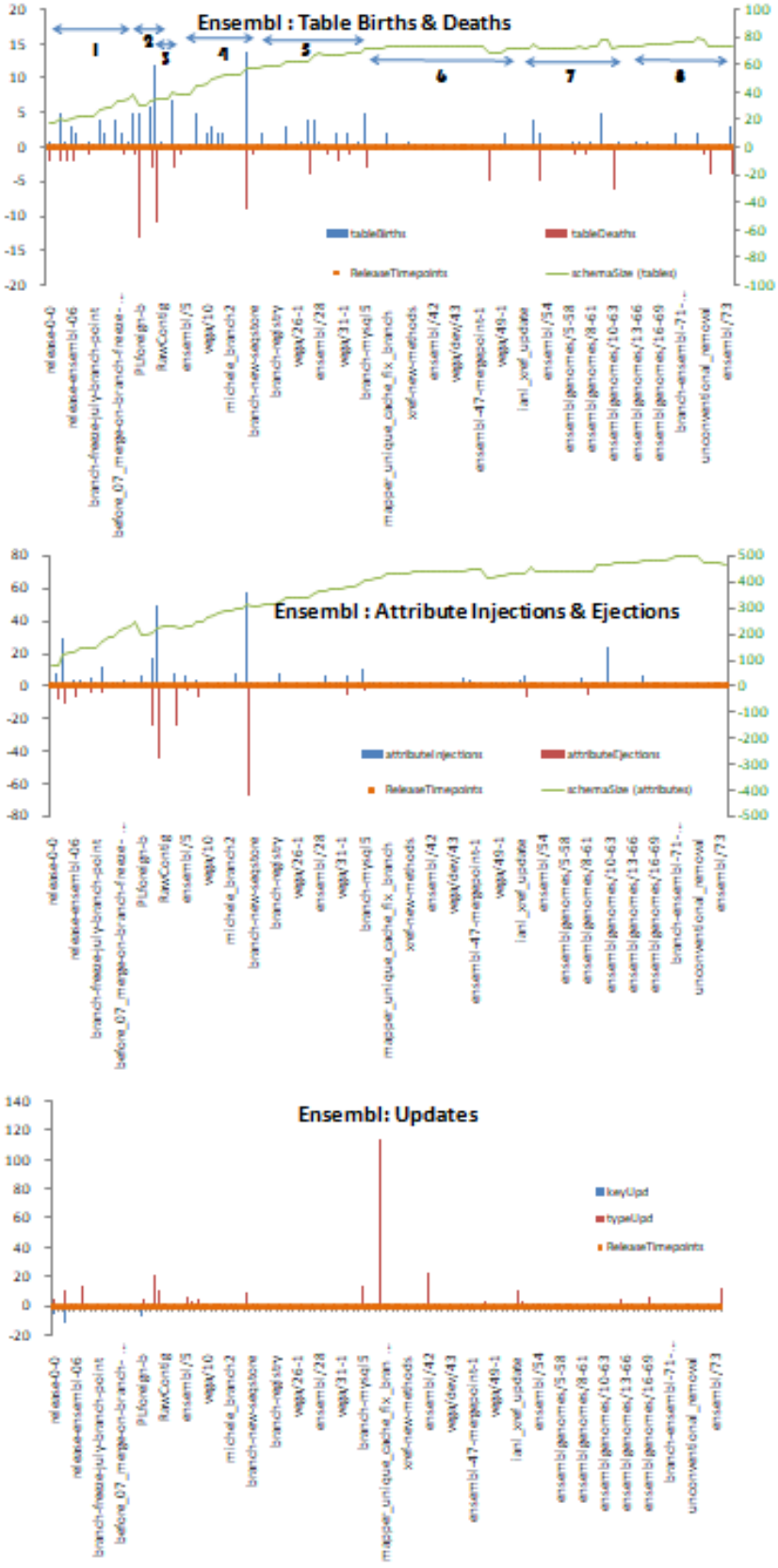


Figure 6 Ensembl : Heartbeat of change per release.

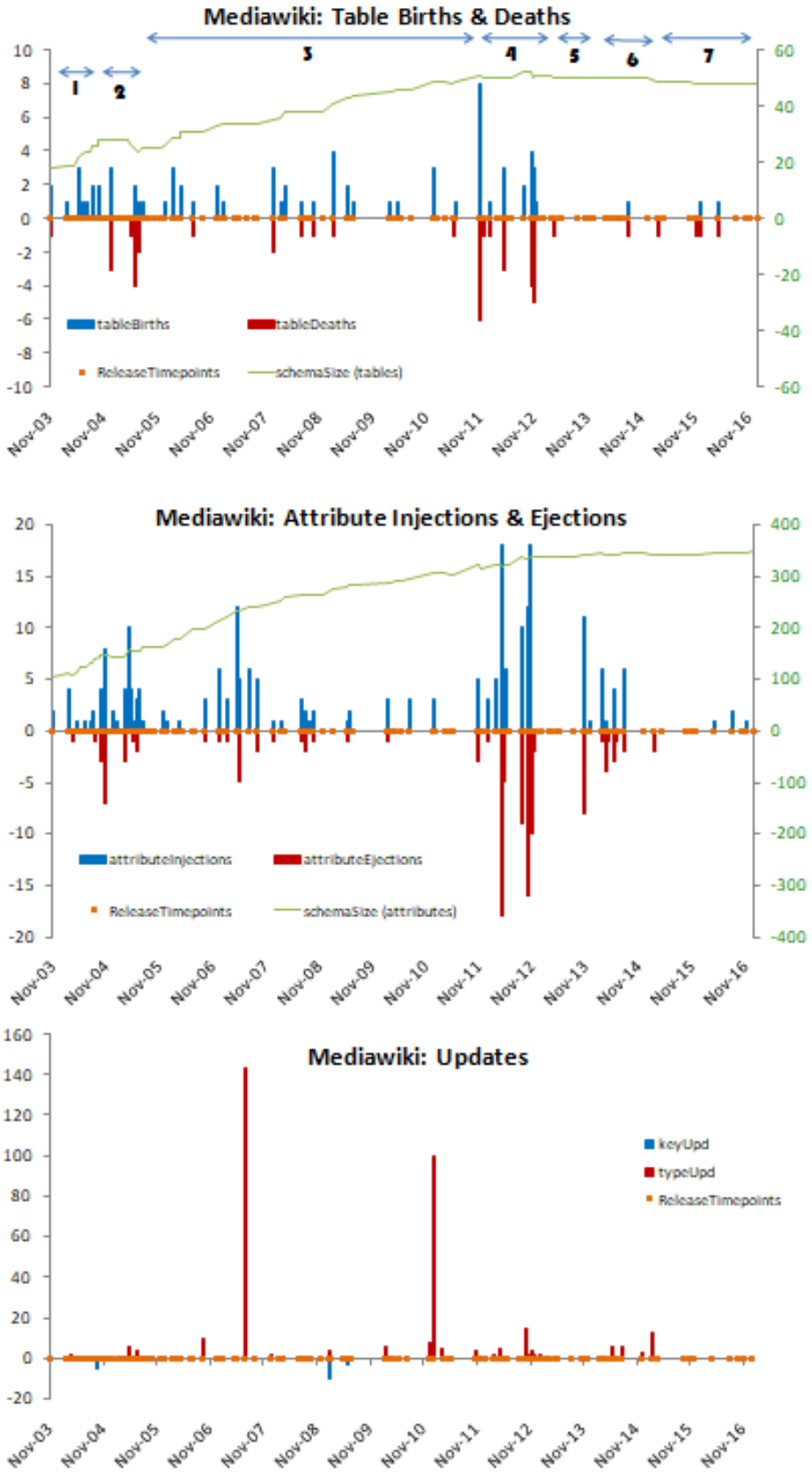


Figure 7 Mediawiki : Heartbeat of change in time.

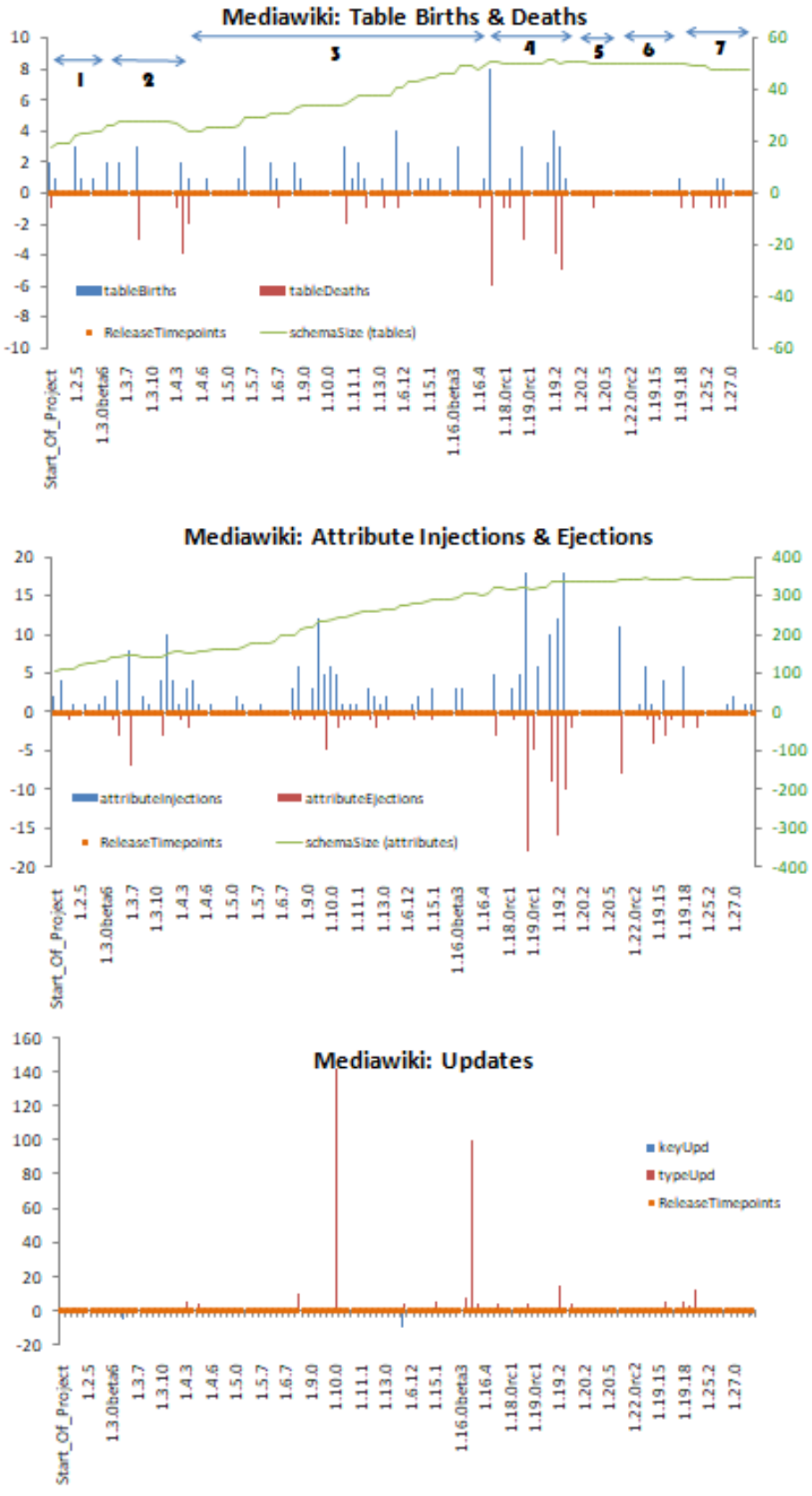


Figure 8 Mediawiki : Heartbeat of change per release.

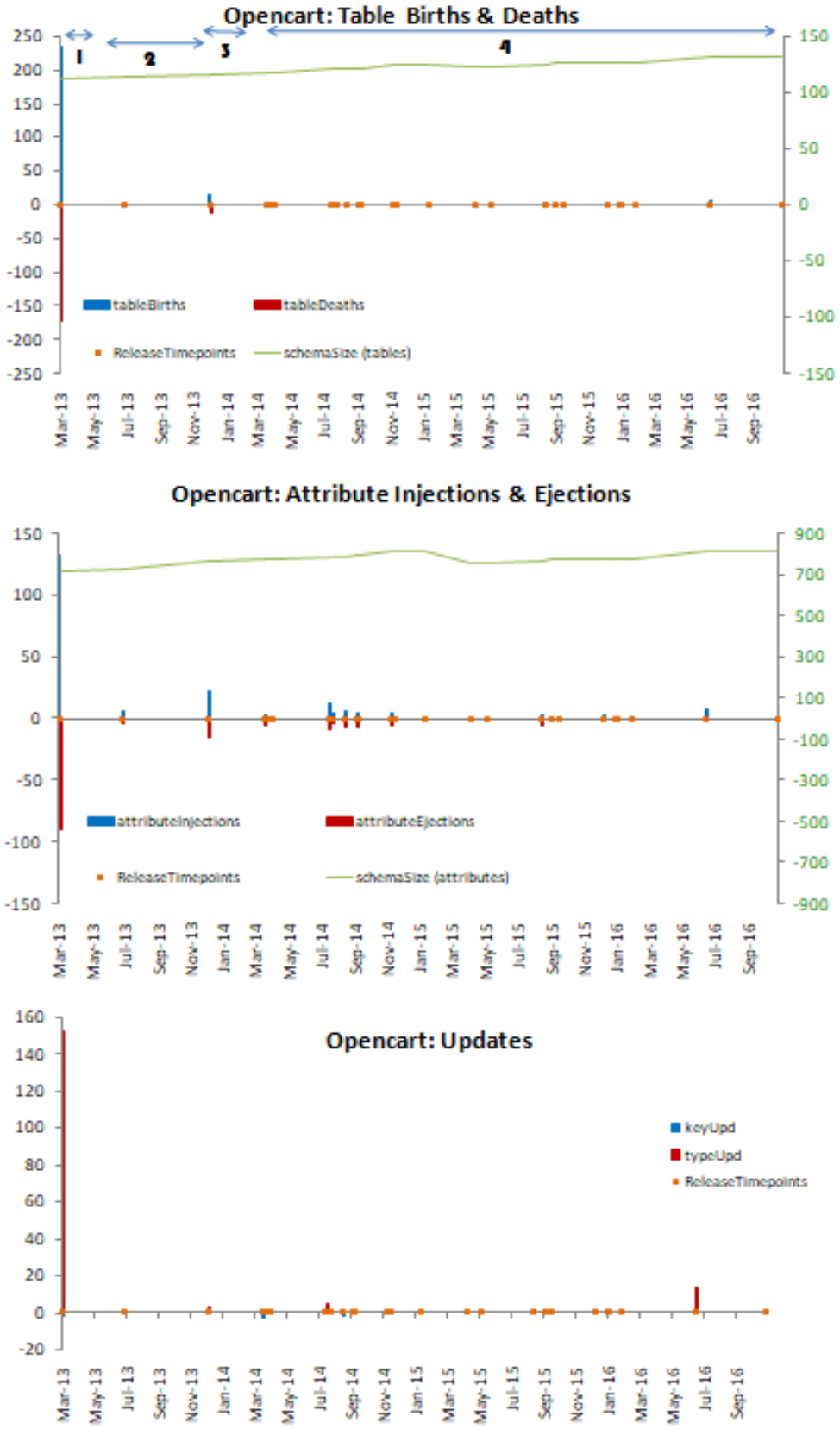


Figure 9 Opencart : Heartbeat of change in time.

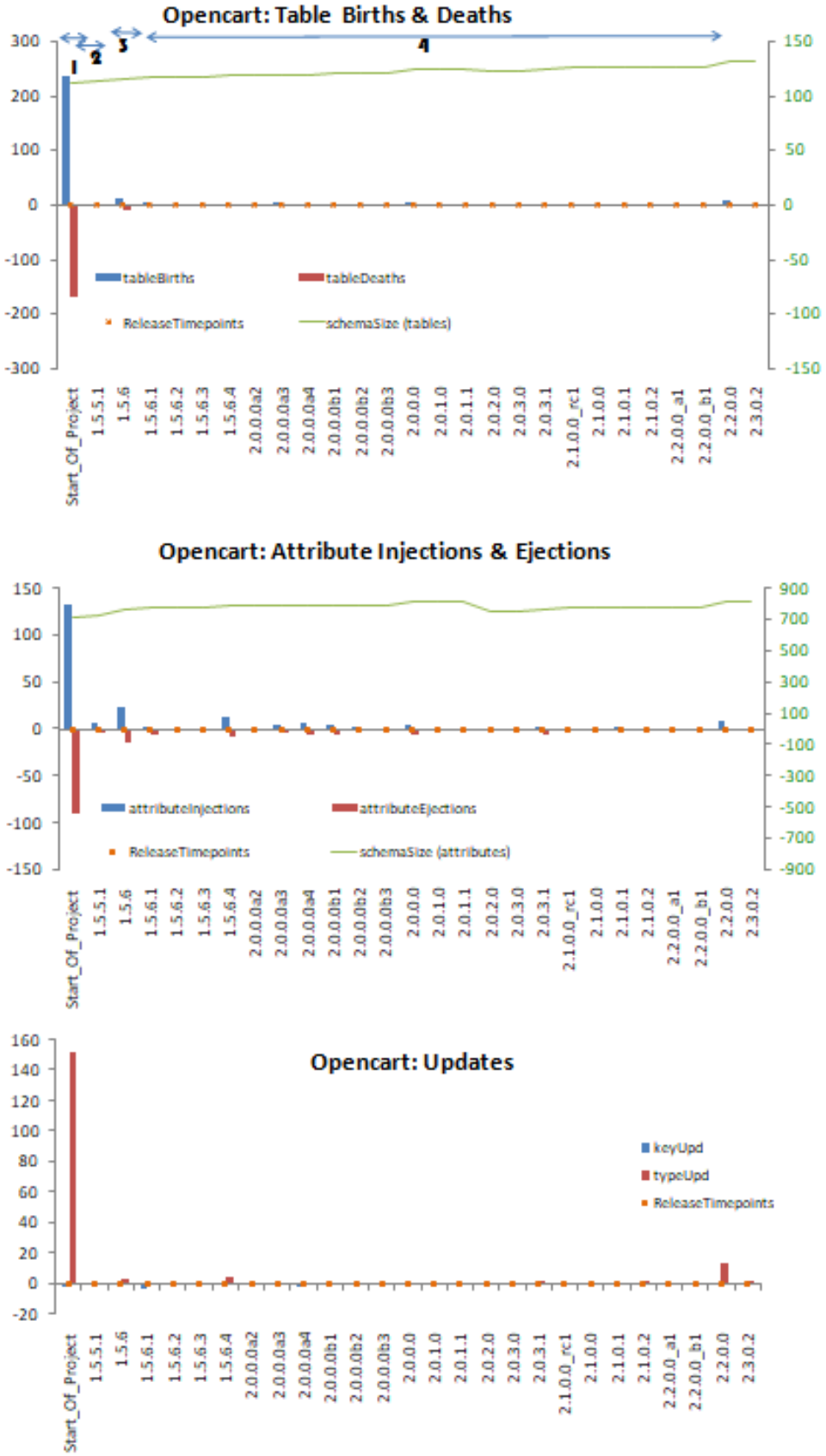


Figure 10 Opencart : Heartbeat of change per release.

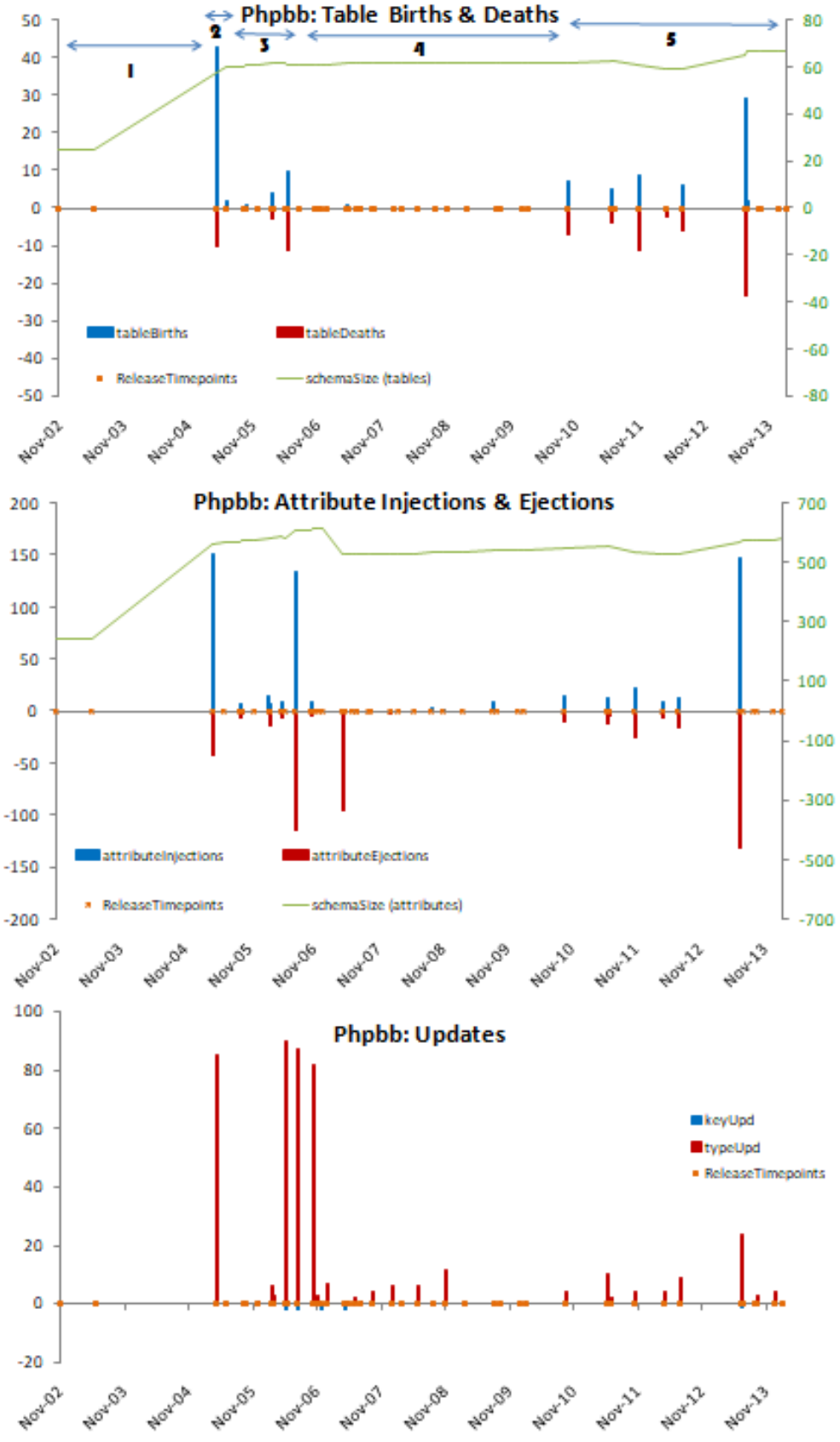


Figure 11 Phpbb : Heartbeat of change in time.

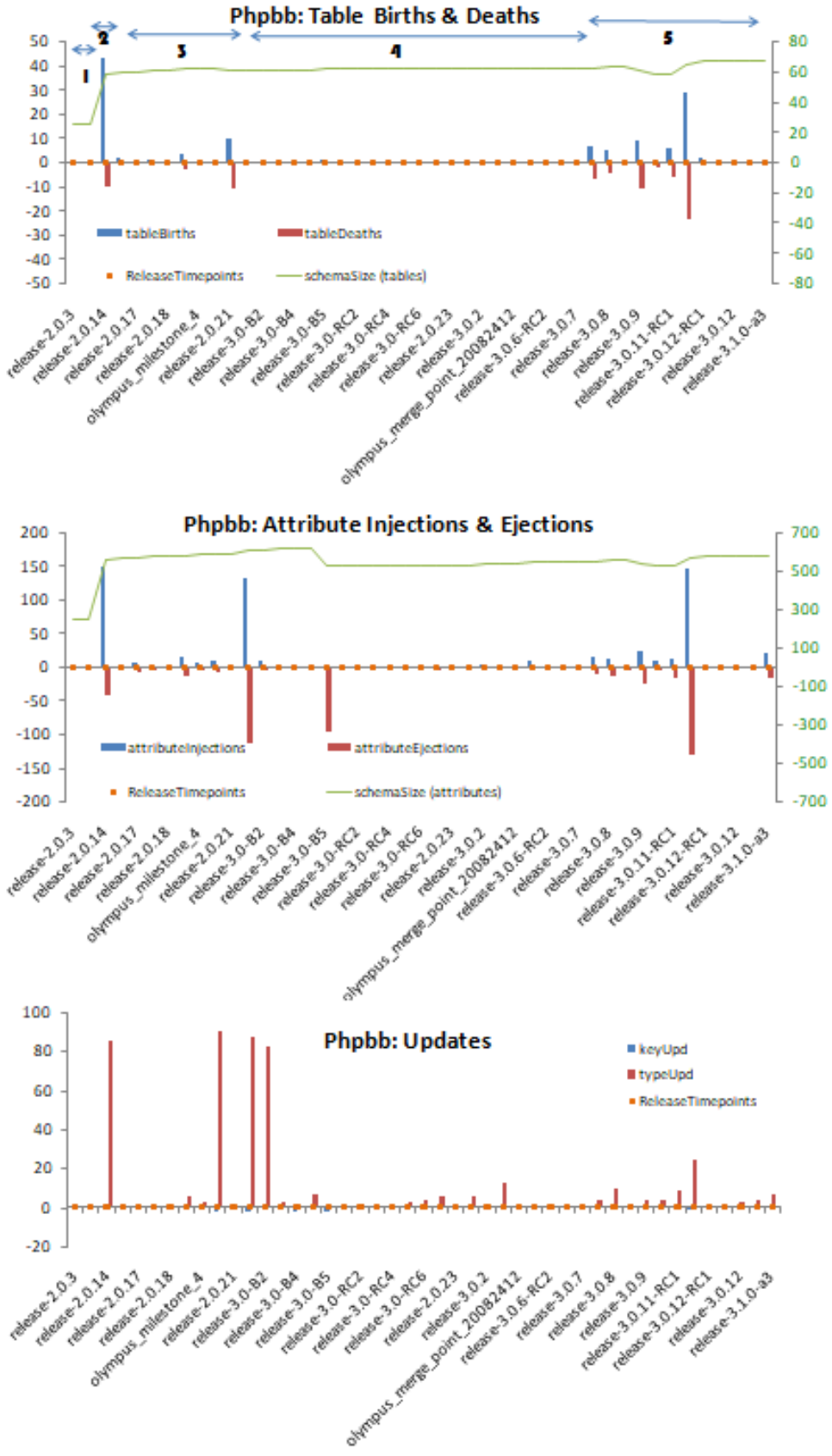


Figure 12 Phpbb : Heartbeat of change per release.

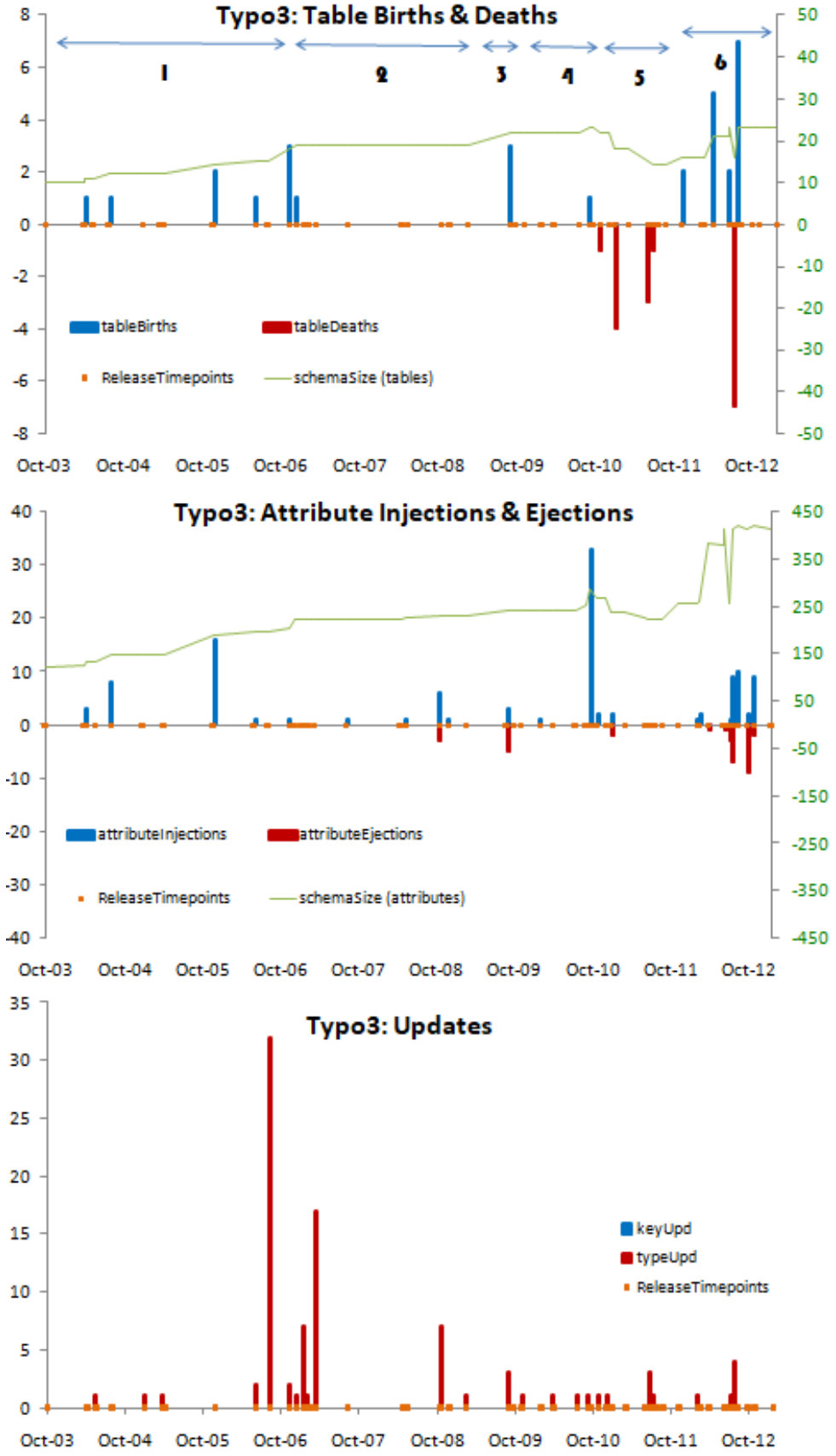


Figure 13 Typo3 : Heartbeat of change in time.

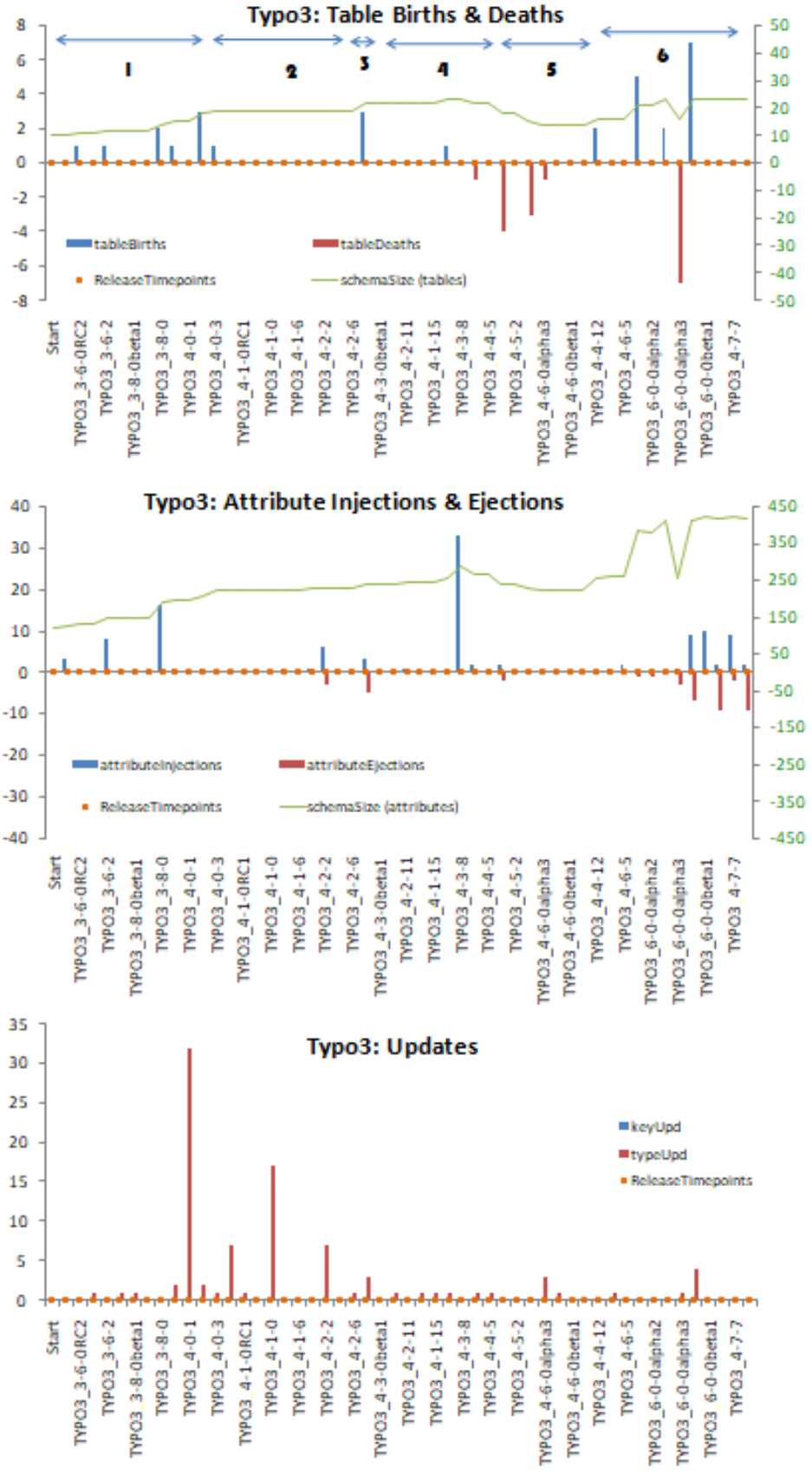


Figure 14 Typo3 : Heartbeat of change per release.

3.3 Special Topics

In this section, we explore three special topics in the context of studying the occurrence of birth and death in evolution of database schemata of open source software projects. Specifically, the first subsection presents the results of our study considering *zombie* tables that are originally deleted and later reborn in a different version. In the second subsection, we try to understand how injections and ejections of attributes are spread in the different periods of a table's life. Finally, in the last subsection, we present a study of the commit histories of six open source projects that contain foreign keys and study their nature and behavior.

3.3.1 Zombie tables : death and rebirth

This part of the study focuses on the existence and behavior of *zombie* tables in the evolution of a database schema. A *zombie* table is defined as the table that at some version of the schema gets deleted and later on another version gets recreated.

A *zombie* table is considered as a part of a group, when it is born, gets deleted, or both at the exact same versions as one or more other *zombie* tables. This notion is important, as it helps us realize with more ease if there exists a mass deletion and re-addition as a part of maintenance procedure, or if the deletion and re-addition is an individual and more explicit occurrence.

In order to explore the existence and behavior of *zombie* tables there was a need for information like specific periods of life for each table, death duration, survival until the last known version and many more. This information was retrieved using the output file of Hecate containing information about the life and size of each table, as input. From the twelve initial datasets, four had no *zombie* tables, three had below six and five had over ten *zombie* tables. This study took place for the five abovementioned datasets, each of them having more than ten *zombies*.

As we can see in Figure 15, over 25% of three datasets' tables were *zombies* (one even had 70% *zombie* tables) and the other two contained a little less than 20% *zombies*.

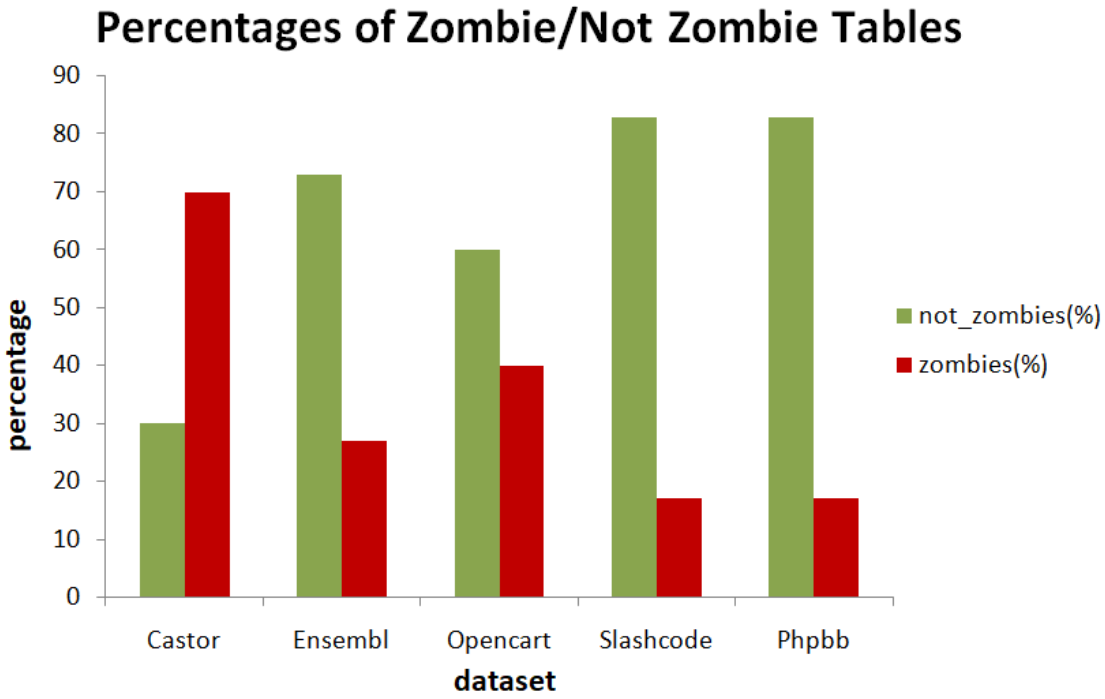


Figure 15 Percentages of *zombie*/not *zombie* tables for each dataset.

Apart from the existence of *zombie* tables, the survival of these tables until the last known schema version was studied. Figure 16 shows the percentages of dead and survivor *zombies*. *The findings concerning the survival of the zombie tables, show that the majority of zombies finally survive, which is an indication of deletions due to maintenance or perfective procedures.* This is the case for four out of five datasets, with Opencart having slightly more dead *zombies* than survivors.

As previously mentioned, a *zombie* table can be a part of a group depending on the versions it died or was born. Figure 17 shows the percentages of *zombie* tables that belong (or do not belong) to groups, for each dataset. For most of the datasets, four out of five, the vast majority of *zombie* tables are part of groups. This probably means that, their deletions and re-additions were part of a larger, coordinated maintenance activity. Though phpbb has equal number of *zombie* tables in groups and *zombie* tables not in groups, due to its small amount of *zombies* (12), phpbb is probably not quite representative.

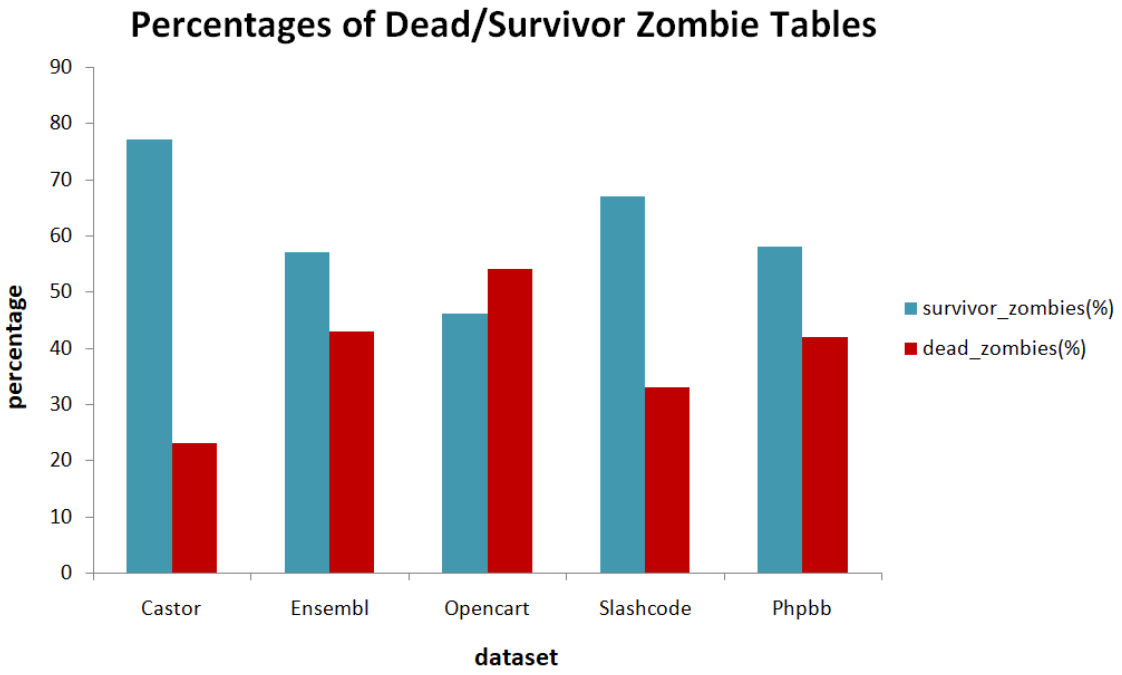


Figure 16 Percentages of dead/survivor *zombie* tables for each dataset.

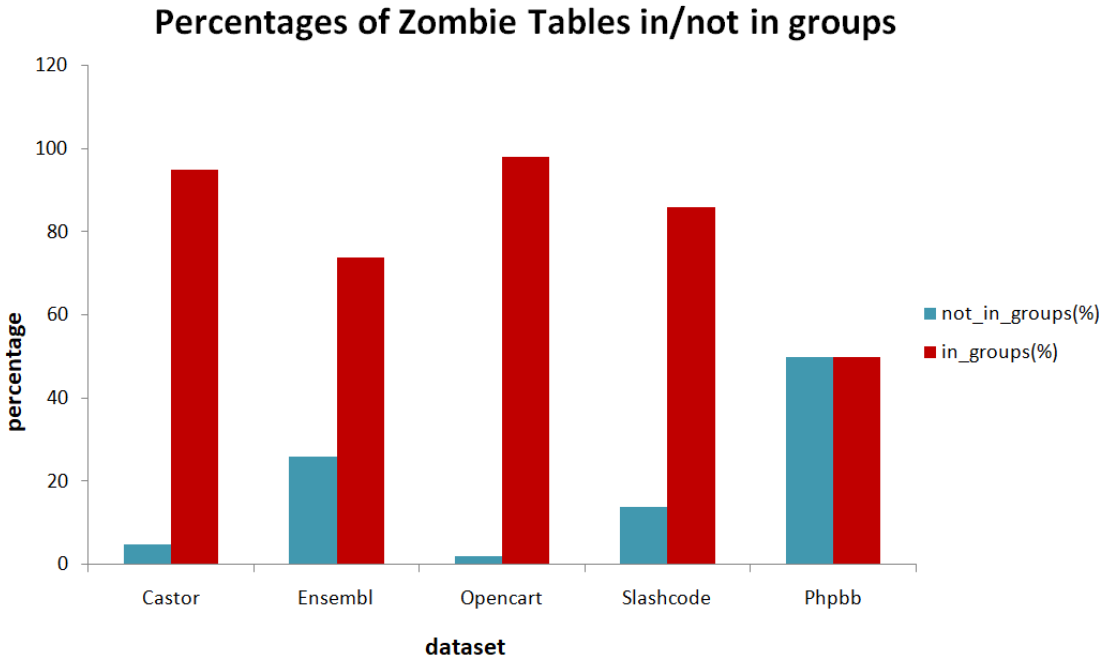


Figure 17 Percentages of *zombies* in groups/not in groups for each dataset.

Figures 18 and 19 show general statistics of *zombie* tables in absolute numbers and percentages respectively.

Dataset	#tables	#zombie_tables	#survivor_zombies	#in_groups
Castor	91	64	49	61
Ensembl	155	42	24	31
Opencart	236	95	44	93
Slashcode	126	21	14	18
Phpbb	70	12	7	6

Figure 18 General statistics of *zombie* tables in absolute numbers.

Dataset	zombies(%)	survivor_zombies(%)	in_groups(%)
Castor	70	77	95
Ensembl	27	57	74
Opencart	40	46	98
Slashcode	17	67	86
Phpbb	17	58	50

Figure 19 General Statistics of *zombie* tables in percentages.

Figures 20 contains statistics of death duration in terms of number of schema versions in which the tables were dead in absolute numbers and percentages respectively.

Dataset	avg_death_duration	dead_avg_death_duration	survivor_avg_death_duration
Castor	3	3,6	3,26
Ensembl	15	24,83	7,25
Opencart	3	1,23	5
Slashcode	9	22,4	2
Phpbb	16	13,8	17,4

Figure 20 Death duration statistics of *zombie* tables in absolute numbers.

Figure 21 shows statistics of occurrences of birth and death of ”zombie” tables in groups. The number of *birth groups* represents the occurrences of “zombie” tables being reborn in the same schema version, while *death groups* the occurrences of “zombie” tables being removed in the same schema version. Finally *both groups* are occurrences of “zombie” tables being both reborn and

removed in the same schema versions. These statistics imply large coordinated activities and not intentional targeted table removals.

Dataset	#birth_groups	#death_groups	#both_groups
Castor	7	7	9
Ensembl	28	18	28
Opencart	5	6	7
Slashcode	19	7	19
Phpbb	10	9	10

Figure 21 Group statistics of *zombie* tables in absolute numbers.

3.3.2 Period of attribute injections and ejections

This part of the study focuses on the injections and ejections of attributes in a table and more specifically, the period of the corresponding table's lifetime, that these actions took place. We treat that the lifetime of a table as a list that contains the schema versions, in which the specific table exists, in ascending chronological order. This ordered set of versions is equally divided in three parts. In the rest of the study when we refer to an attribute as *added@start* or *deleted@start*, we mean that this injection or ejection took place in the first of the three parts of the table's lifetime. *Added/Deleted@mid* and *added/deleted@end* mean injection or ejection at the second and third part of a table's lifetime respectively.

It should be noted that we do not focus on the attributes that are born along with their table or die with it, but we focus on the attributes that are added later on the table's life. The same applies for attribute deletions.

To explore the attribute injections and find when these actions take place, we retrieved the number of attributes *added@start*, *mid* or *end* of each table by using information about the attribute additions per table and per version retrieved by Hecate. As we can see in Figure 22 the injections of attributes took place mainly at the *start* or *mid* of a table's life for the majority of the datasets. This means that any attribute injections that may occur are rarely during the end of a table's life. There are datasets that have these injections divided almost equally into the three parts of the lifetime though. Those three datasets are Slashcode, Typo3 and Zabbix.

As far as the attribute ejections are concerned, we used a similar procedure as before with the information about attribute deletions per table and per version retrieved with Hecate. As we can see from Figure 23 the ejections of attributes also happen mostly at the *start* or *mid* of a table's lifetime for the majority of the datasets. Similarly to injections, Slashcode, Typo3 and Zabbix have deviant behaviors and the ejections there happen mainly at the end of a table's lifetime.

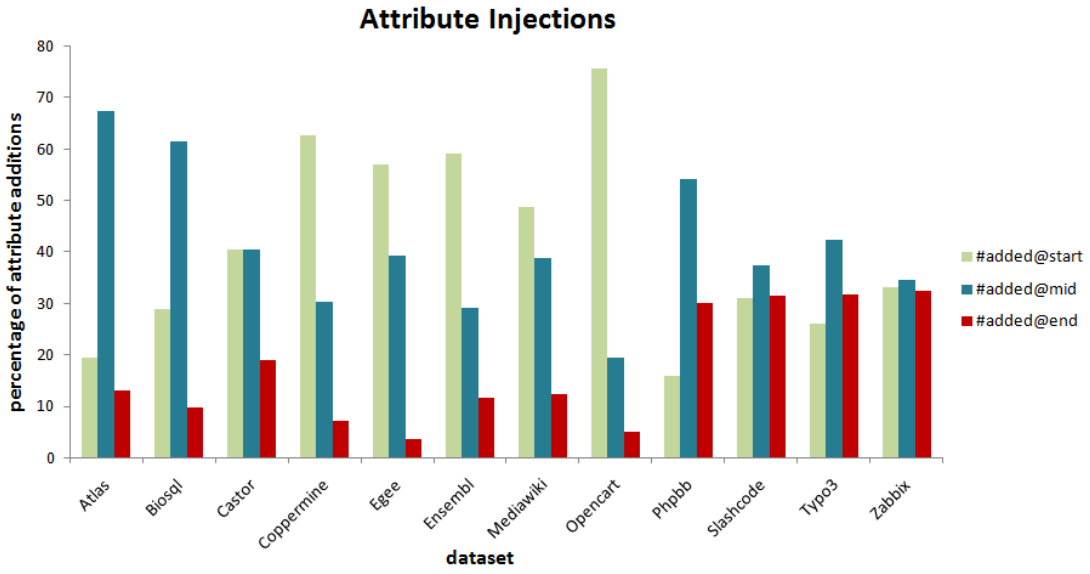


Figure 22 Percentages of attribute injections to existing tables.

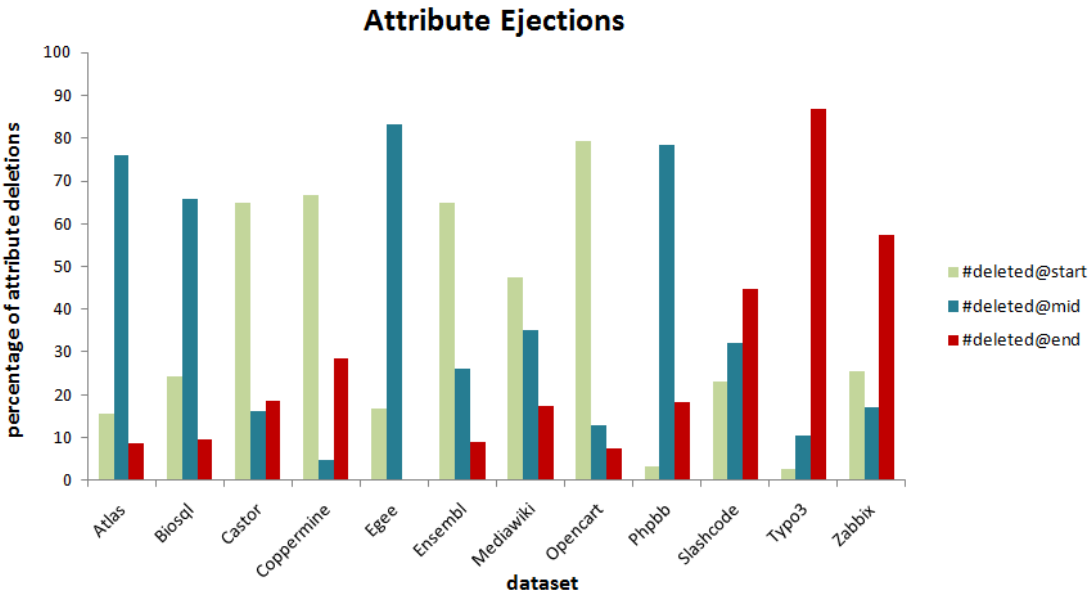


Figure 23 Percentages of attribute ejections from tables.

Figures 24 and 25 contain the statistics of attribute ejections and injections for all twelve datasets in absolute numbers and percentages respectively. It should be noted that the absolute numbers are the aggregations of the corresponding numbers of all the tables of each dataset.

Dataset	#added@start	#added@mid	#added@end	#deleted@start	#deleted@mid	#deleted@end
Atlas	30	104	20	18	88	10
Biosql	30	64	10	20	54	8
Castor	62	62	29	52	13	15
Coppermine	27	13	3	14	1	6
Egee	16	11	1	2	10	0
Ensembl	319	157	63	272	110	38
Mediawiki	63	50	16	19	14	7
Opencart	106	27	7	73	12	7
Phpbb	19	65	36	6	142	33
Typo3	29	47	35	1	4	33
Zabbix	48	50	47	12	8	27
Slashcode	134	162	136	59	82	114

Figure 24 Statistics of attribute injections and ejections in absolute numbers.

Dataset	#added@start	#added@mid	#added@end	#deleted@start	#deleted@mid	#deleted@end
Atlas	19,5	67,5	13	15,5	75,9	8,6
Biosql	28,9	61,5	9,6	24,4	65,9	9,7
Castor	40,5	40,5	19	65	16,25	18,75
Coppermine	62,8	30,2	7	66,7	4,8	28,5
Egee	57,1	39,3	3,6	16,7	83,3	0
Ensembl	59,2	29,1	11,7	64,8	26,2	9
Mediawiki	48,8	38,8	12,4	47,5	35	17,5
Opencart	75,7	19,3	5	79,4	13	7,6
Phpbb	15,8	54,2	30	3,3	78,5	18,2
Typo3	26,1	42,3	31,6	2,6	10,5	86,9
Zabbix	33,1	34,5	32,4	25,5	17	57,5
Slashcode	31	37,5	31,5	23,1	32,2	44,7

Figure 25 Statistics of attribute injections and ejections in percentages.

3.3.3 Foreign Key birth and death

This part of the study focuses on the way that foreign keys are added or deleted during the evolution of a schema. At first we analyze the behavior of foreign keys for each dataset and in the end we present all the statistics gathered.

Terminology
The additions and deletions of the foreign keys were categorized into ”born with table”, ”explicit addition”, ”died with table” or ”explicit deletion”. An addition of a foreign key is considered as ”born with table”, when either the source or the target table is born along with the foreign key, while an ”explicit addition” happens, when a foreign key is added to existing tables. Respectively, in the case of deletions, a deletion of a foreign key is considered as ”died with table”, when either the source or the target table is removed along with the foreign key, while an ”explicit deletion” happens when neither of the source or target table gets deleted and only the foreign key is removed.

This study took place for six different datasets. Three of the databases, Atlas, Castor and Egee are hosted by CERN, the European Organization for Nuclear Research based on Geneva. Two databases, Slashcode and Zabbix are content management systems (CMS), while the last database, Biosql, stores genomic data.

Atlas
Atlas has the highest number of foreign keys among the studied datasets. Over the 85 versions of Atlas' schema there were numerous events of both additions and deletions of foreign keys. In version 1177518923, an entire ”neighborhood” of 8 tables was deleted, along with their foreign keys and in version 1215283813, an entire ”neighborhood” of 6 tables was added with foreign keys among them. Also, in versions 1207729000 and 1217322513, both additions and deletions of foreign keys of previous versions, were reversed. In Figure 26 we can see the changes that happened over the versions in terms of foreign key additions and deletions in Atlas' schema. The horizontal axis represents the schema's version id, while the vertical axis represents the number of foreign keys, additions, or deletions of foreign keys respectively.

During the evolution of Atlas' schema, the additions of foreign keys took place mostly along with a table addition and not to existing tables (for over 90% of the time). The deletions of the foreign keys took also place along with a table change (here the deletion of the table) for the majority (almost 70% of the time). Figure 27 depicts the way that foreign keys are added or removed. The horizontal axis represents the name of the schema's version, while the vertical axis represents the number of the corresponding foreign key change.

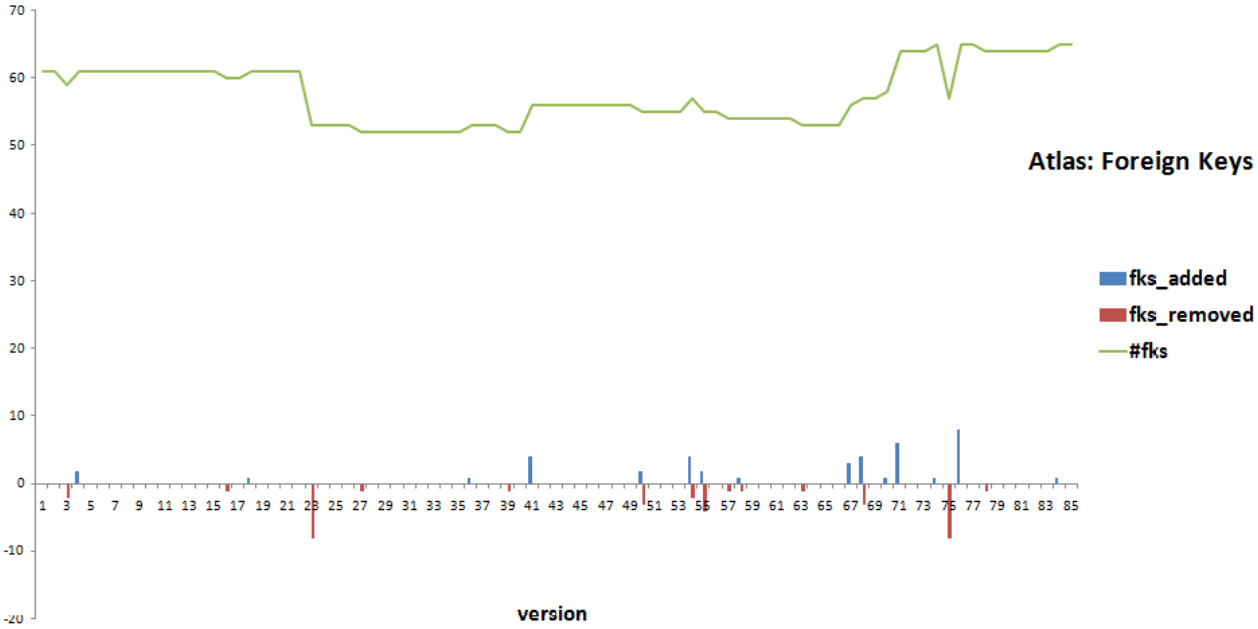


Figure 26 Foreign key changes in the 85 versions of Atlas.

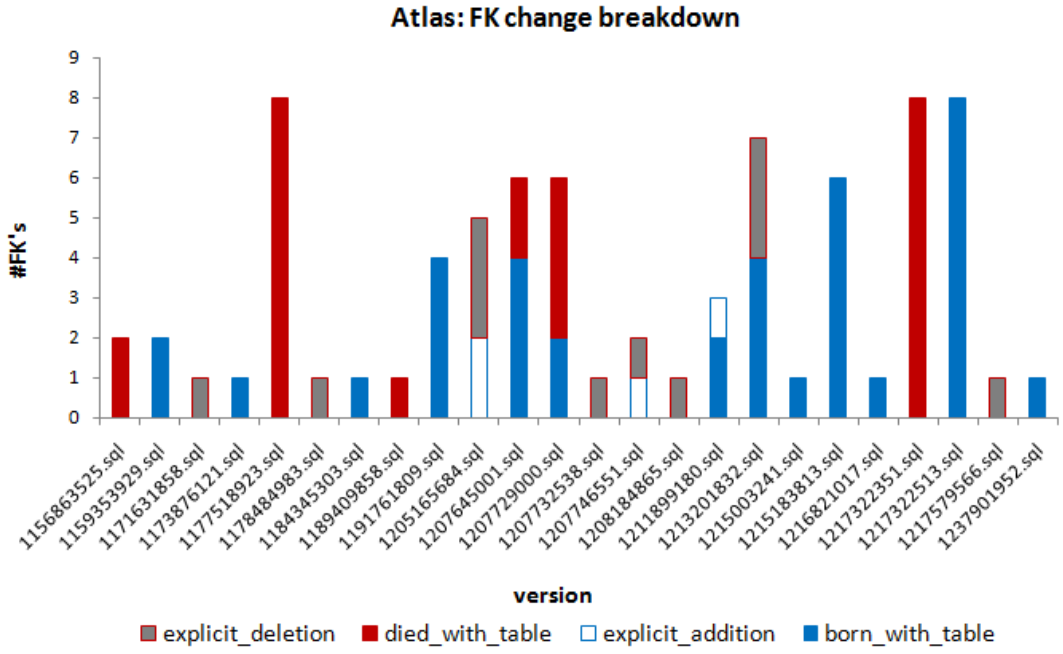


Figure 27 Foreign key type of changes in Atlas.

Biosql

Over the 47 versions of Biosql's schema, there were 127 additions and deletions of foreign keys. Biosql's evolution contained 4 renaming cases in

versions 1031817528, 1045618809, 1047465289 and 1047466335 respectively. A renaming case is considered as the action of deleting a table with its foreign keys and recreating it in the same version, to change the table's or table's attribute's name. Figure 28 shows the foreign key additions and deletions that took place in the evolution of Biosql's schema. The horizontal axis represents the schema's version id, while the vertical axis represents the number of foreign keys, additions, or deletions of foreign keys respectively.

The additions of foreign keys in Biosql follow the same behavior as Atlas. Almost 90% of the time, the additions happened along with the addition of a table. The deletions of foreign keys, also for over 90% of the time, happened when a table was deleted and not as an explicit deletion of the foreign key. In Figure 29, we can see the type of foreign key changes that took place in the evolution of Biosql. The horizontal axis represents the name of the schema's version, while the vertical axis represents the number of the corresponding foreign key change.

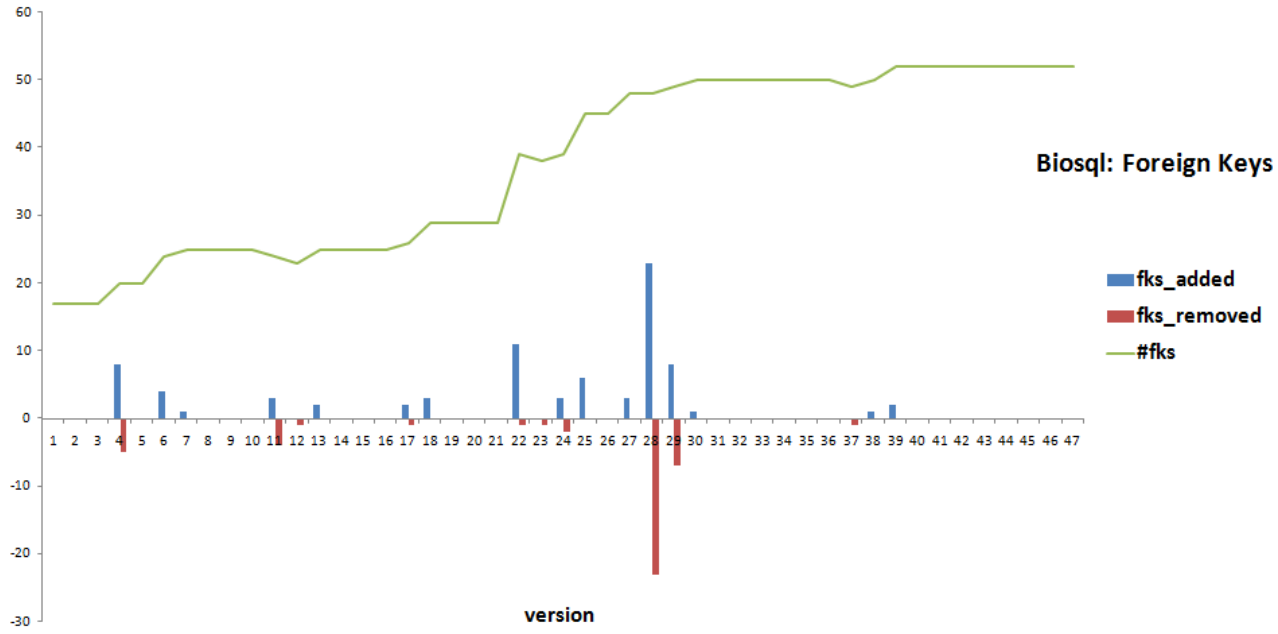


Figure 28 Foreign key changes in the 47 versions of Biosql.

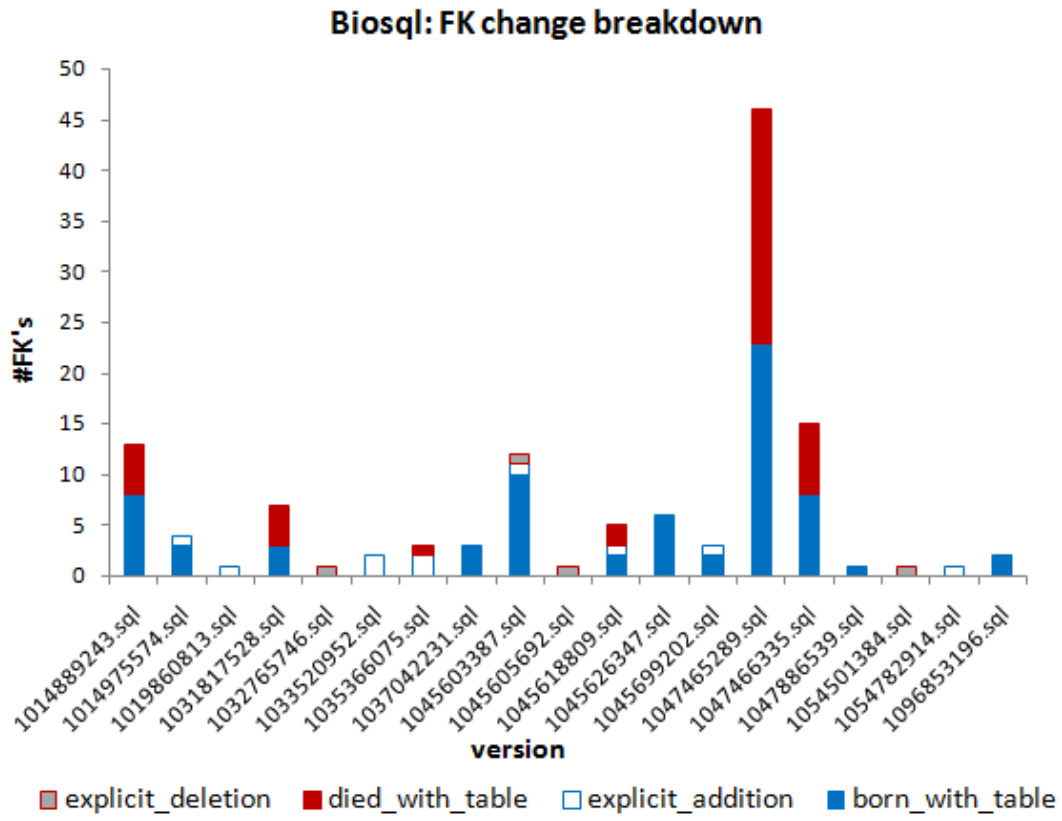


Figure 29 Foreign key type of changes in Biosql.

Castor

For the study of Castor, 194 versions of its schema were available. During those 194 versions though, there were very few changes, under 10 additions and under 10 deletions of foreign keys, while in the level of tables the changes were significantly more. This probably happened, because the dataset itself had very few foreign keys to begin with. Among these few changes, in version rev 1.051 a foreign key is added, then this action is reversed in version rev 1.103 and then the same foreign key is re-added in version rev 1.104. Figure 30 shows the additions and deletions of foreign keys that happened during the 194 versions of Castor's schema. The horizontal axis represents the schema's version id, while the vertical axis represents the number of foreign keys, additions, or deletions of foreign keys respectively.

Because of the small number of foreign key changes that happened during the evolution of Castor, the percentages of the causes of additions and deletions are not very similar to the previous ones. Most of the additions here

happened to existing tables, while the deletions are equally divided into removal of foreign key along with the table deletion, and removal of foreign key with no table deletion. Figure 31 depicts the type of foreign key changes that took place in Castor. The horizontal axis represents the name of the schema's version, while the vertical axis represents the number of the corresponding foreign key change.

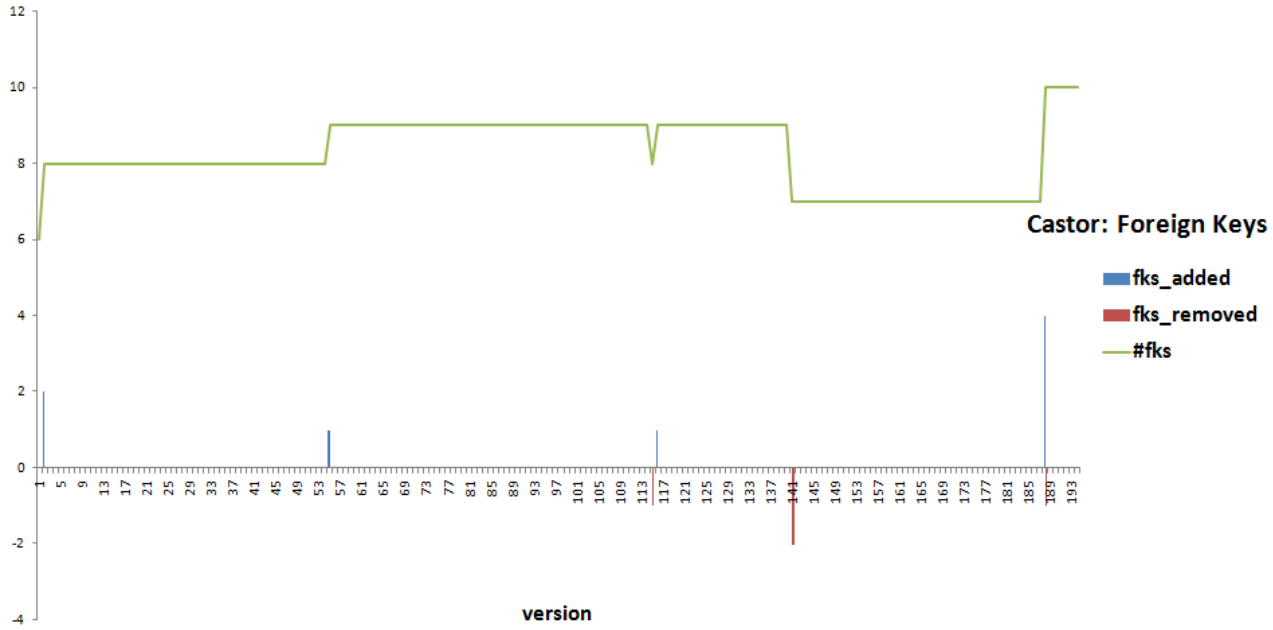


Figure 30 Foreign key changes in the 194 versions of Castor.

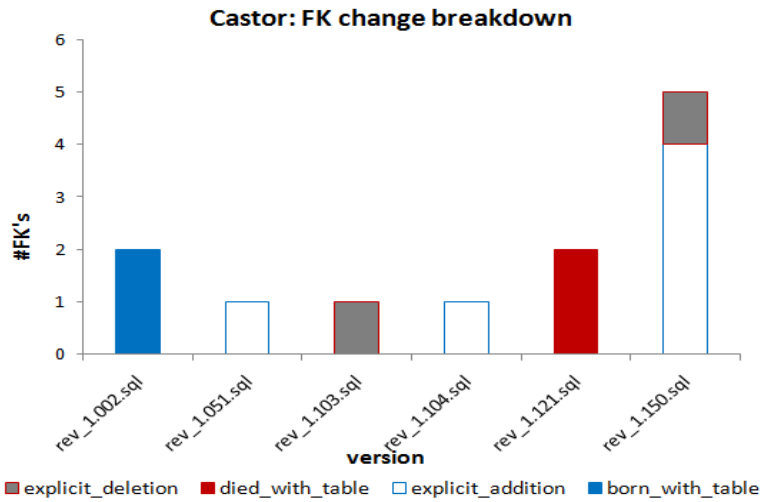


Figure 31 Foreign key type of changes in Castor.

Egee

During the study of Egee's evolution, there were only 17 versions of its schema available. This database has very few foreign keys, very few changes happened and none of those was quite interesting. In Figure 32 we can see the foreign key changes that took place during the evolution of Egee. The horizontal axis represents the schema's version id, while the vertical axis represents the number of foreign keys, additions, or deletions of foreign keys respectively.

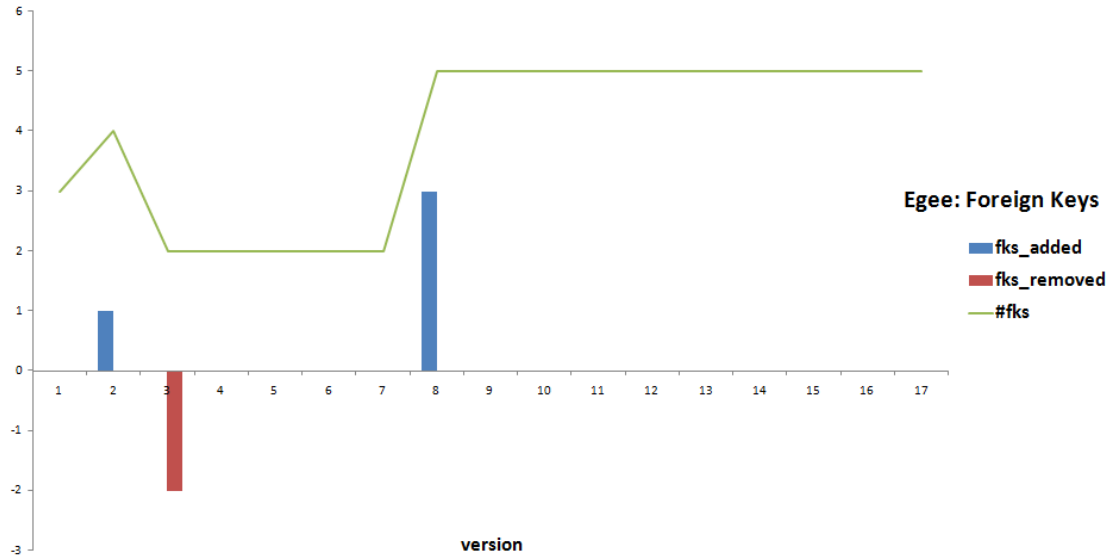


Figure 32 Foreign key changes in the 17 versions of Egee.

Similarly with Castor, Egee's percentages of additions' and deletions' causes cannot be representative, because of the small amount of changes that took place. Most of the additions happened with a table addition, while all of the deletions happened along with the table deletion. Figure 33 depicts the type of foreign key changes that took place in Egee. The horizontal axis represents the name of the schema's version, while the vertical axis represents the number of the corresponding foreign key change.

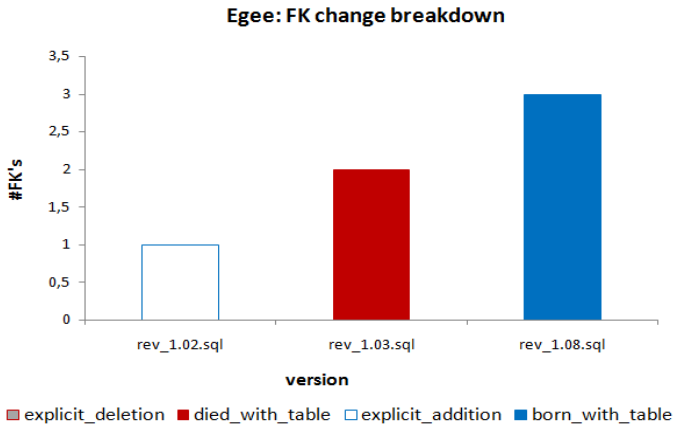


Figure 33 Foreign key type of changes in Egee.

Slashcode

Slashcode's schema evolution consists of 399 different versions. Slashcode has a lot of foreign key changes, both additions and deletions and a lot of interesting cases. There were two cases where the actions were reversed in the next versions and also a renaming case. The most interesting part of Slashcode's evolution though, is that after a certain version all its foreign keys are gradually deleted. In the first occurrence of massive foreign key removals (at version rev 1.120), 23 foreign keys were deleted. This mass removal took place due to a problem with the compatibility of the attribute types that the foreign keys referred to. The Data Definition file contains an explanatory comment for this removal:

"Commented-out foreign keys are ones which currently cannot be used because they refer to a primary key which is NOT NULL AUTO INCREMENT and the child's key either has a default value which would be invalid for an aut increment field, typically NOT NULL DEFAULT '0'. Or, in some cases, the primary key is e.g. VARCHAR(20) NOT NULL and the child's key will be VARCHAR(20). The possibility of NULLs negates the ability to add a foreign key. ⇐ That's my current theory, but it doesn't explain why discussions.topic SMALLINT UNSIGNED NOT NULL DEFAULT '0' is able to be foreign-keyed to topics.tid SMALLINT UNSIGNED NOT NULL AUTO NCREMENT".

In the second deletion (at version rev 1.151), 12 foreign keys were removed, because some tables changed their storage engine to Innodb from Myisam. There was also an explanatory comment inside the corresponding sql file:

"Stories is now InnoDB and these other tables are still MyISAM, so no foreign keys between them.".

The rest of the deletions happened because the foreign keys caused too many problems to the system that could not debugged, resulting in the decision to leave the schema without any foreign keys. We have retrieved several comments for these removals. At version re 1.174, where 3 foreign keys were deleted the following comment was found:

"This doesn't work, makes createStory die. These don't work, should check why..."

At version's rev 1.189 file the comments mention:

"This doesn't work, since in the install pollquestions is populated before users, alphabetically"

Finally, at version rev 1.201 the following comment was found:

"This doesn't work, since discussion may be 0."

At the end of this process, the schema is left with zero foreign keys. Interestingly enough, the schema also contained no foreign keys at its start.

Figure 34 shows the foreign key changes that took place during the evolution of Slashcode. The horizontal axis represents the schema's version id, while the vertical axis represents the number of foreign keys, additions, or deletions of foreign keys respectively.

As it was mentioned above, Slashcode had a huge amount of deletions, caused by system problems and in those deletions the foreign keys were deleted explicitly, without any of the tables being removed. Slashcode has a behavior different than all the other studied datasets. It is the dataset, that the additions and the deletions happened mostly explicitly. Figure 35 depicts the type of foreign key changes that took place. The horizontal axis represents the name of the schema's version, while the vertical axis represents the number of the corresponding foreign key change.

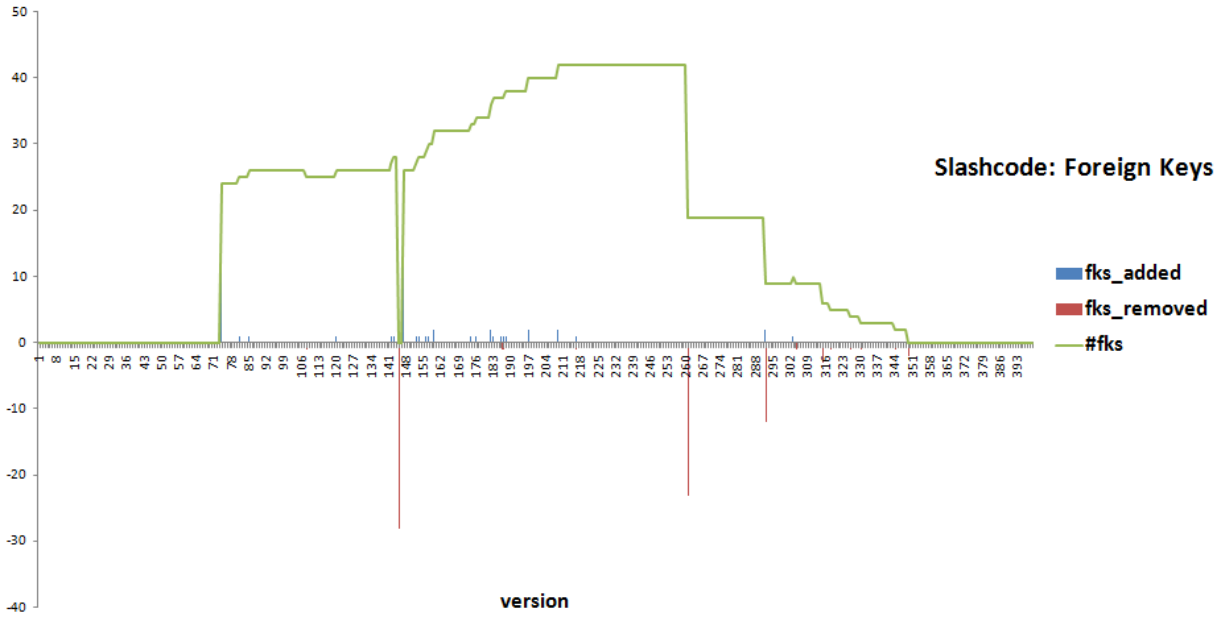


Figure 34 Foreign key changes in the 399 versions of Slashcode.

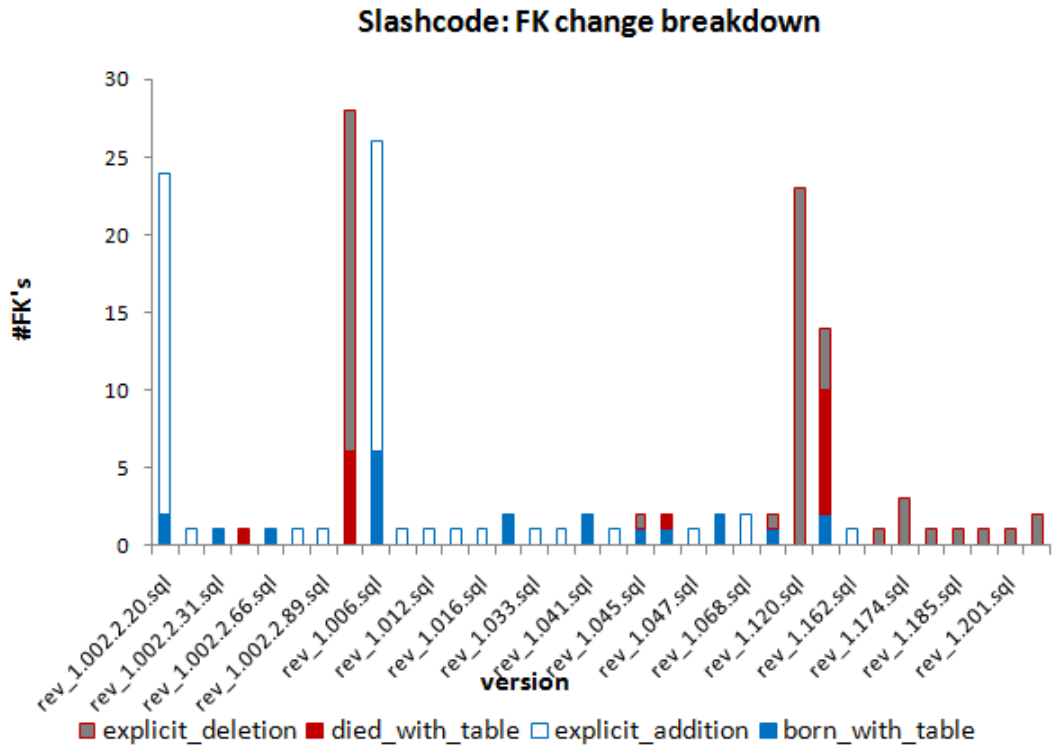


Figure 35 Foreign key type of changes in Slashcode.

Zabbix

During the study of Zabbix, 160 versions of its schema were available. Like Slashcode, Zabbix is also a CMS and has a lot of changes to its foreign key state and a lot of interesting cases. There were two cases of renaming, in versions 1.041 and 1.135, but the most interesting case of all, is that in version 1.151 all but two of the foreign keys are commented out of the schema. We could not find an explanation as to why this removal took place inside the mysql Data Definition files. Figure 36 depicts the foreign key changes that took place in Zabbix. The horizontal axis represents the schema's version id, while the vertical axis represents the number of foreign keys, additions, or deletions of foreign keys respectively.

The addition of foreign keys in Zabbix, follows the behavior of most of the datasets. Circa 85% of the additions of foreign keys happened along with a table addition. Because of the abrupt removal of foreign keys that was described above most of the deletions took place explicitly, without any table removal. In figure 37 we can see the types of foreign key changes that took place in Zabbix. The horizontal axis represents the name of the schema's version, while the vertical axis represents the number of the corresponding foreign key change.

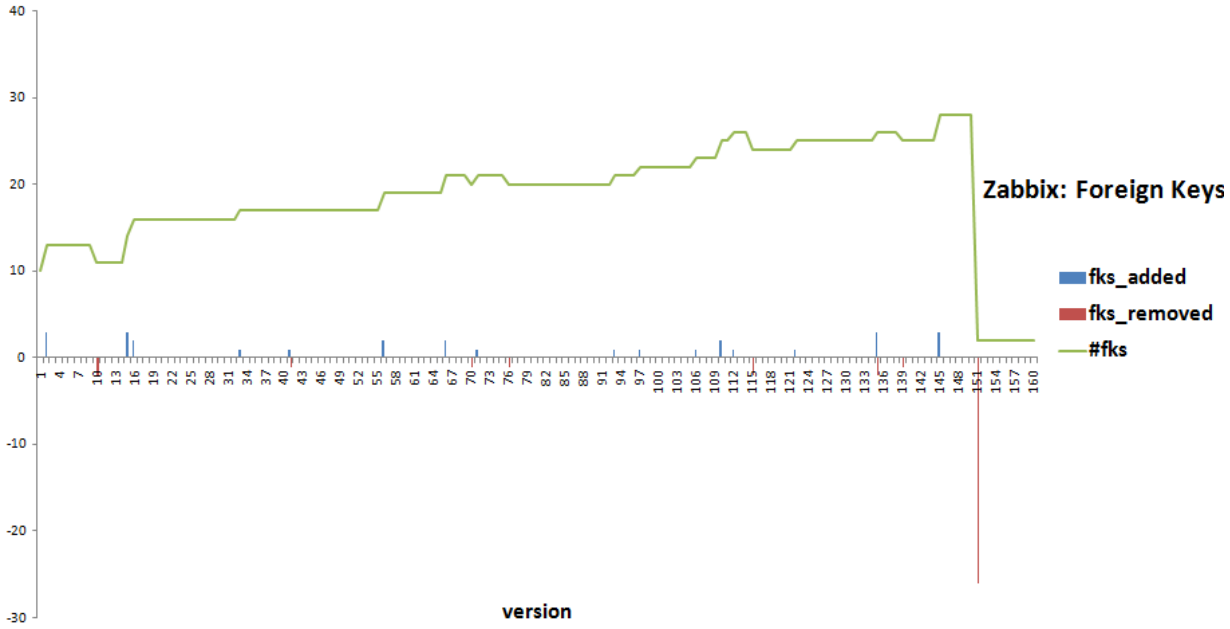


Figure 36 Foreign key changes in the 160 versions of Zabbix.

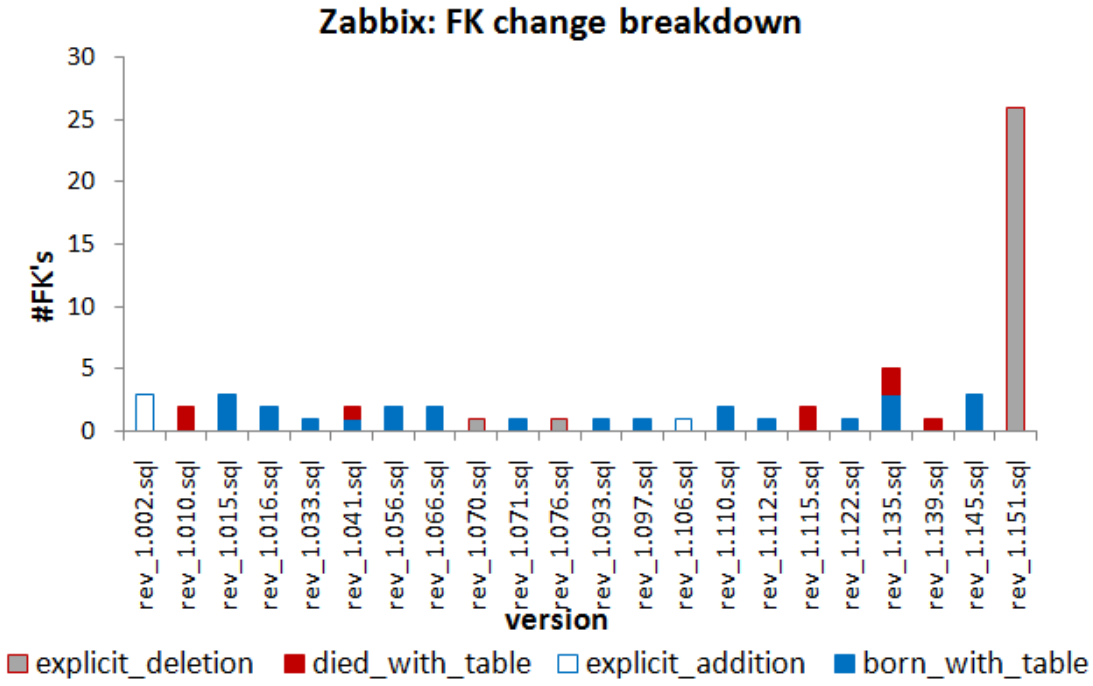


Figure 37 Foreign key type of changes in Zabbix.

The main findings of this study are summarized as follows. Foreign keys are sometimes treated as an integral part of the system, mainly in scientific projects. In those cases the foreign keys are born and evicted along with table birth and eviction. There are cases though, that foreign keys seem to be a second-class add-on, where the foreign keys seem to be removed mostly not along with their table. The two cases of CMSs, that were available to us, show a disinclination towards having foreign keys as part of the schema. Both of these cases ended up, to the best of our knowledge, with no foreign keys in their schema. These removals seem to be a result of difficulty of managing technical issues with foreign keys. In the studied data sets, the mere existence of foreign keys is too scarce. The foreign key changes found in this study are mainly small in volume.

Figure 38 shows the total numbers and percentages of foreign key statistics for all the studied databases.

Diachronic Graph

Dataset	TablesDG	FK'sDG	FKs@start	FKs@end
Atlas	88	88	61	65
Biosql	45	79	17	52
Castor	91	13	6	10
Egee	12	6	3	5
Slashcode	126	47	0	0
Zabbix	58	38	10	2

	#FKs_added				
Dataset	Total	Born w/ source table	Born w/ target table	Born w/ both tables	Explicit Addition
Atlas	41	26	2	9	4
Biosql	81	43	23	5	10
Castor	8	1	0	1	6
Egee	4	1	0	2	1
Slashcode	77	21	0	0	56
Zabbix	28	17	4	3	4

Dataset	(%)Born w/ source table	(%)Born w/ target table	(%)Born w/ both tables	(%)Explicit Addition
Atlas	63%	5%	22%	10%
Biosql	53%	29%	6%	12%
Castor	12,50%	0%	12,50%	75%
Egee	25%	0%	50%	25%
Slashcode	27%	0%	0%	73%
Zabbix	61%	14%	11%	14%

	#FKs_removed				
Dataset	Total	Died w/ source table	Died w/ target table	Died w/ both tables	Explicit Deletion
Atlas	37	15	1	9	12
Biosql	46	20	18	4	4
Castor	4	0	0	2	2
Egee	2	2	0	0	0
Slashcode	77	9	3	4	61
Zabbix	36	2	5	1	28

Dataset	(%)Died w/ source table	(%)Died w/ target table	(%)Died w/ both tables	(%)Explicit Deletion
Atlas	41%	3%	24%	32%
Biosql	43%	39%	9%	9%
Castor	0%	0%	50%	50%
Egee	100%	0%	0%	0%
Slashcode	12%	4%	5%	79%
Zabbix	5%	14%	3%	78%

Figure 38 Tables with foreign key statistics for all the studied datasets.

3.4 Conclusions

The purpose of this part of the study was to explore the nature of births and deaths in schema evolution and to search for possible representative segmentations of the schema that can characterize the nature of change of their evolution. The conclusions we reached during this study are as follows.

- The most interesting finding in our study is that, with the single exception of Typo3, the history of a database schema comes in two *mega-phases*: (a) a "hot" *expansion mega-phase at the start of its life* demonstrating growth of information capacity, along with the necessary maintenance and (b) a "cooling" *housekeeping mega-phase at its middle and later life* where either maintenance actions or stillness dominate the update activity.
- *Growth* in schemata is mainly located in the start of their life, either alone or accompanied by *maintenance*
- *Maintenance* can be found in all the possible stages of a schema's life
- *Maintenance* is frequently followed or preceded by *minor activity* periods with occurrences of this combination overwhelmingly found towards the end of the schema's life
- For the majority of the datasets, minor (or even zero) activity periods frequently take up long periods in time, especially at the end of their history
- A few datasets though, have *intense evolution* with changes of significant volume

To study in detail all the changes that schemata undergo during their evolution and extract phases that are representative and can fairly be characterized by their nature is a manual, and potentially exhaustive and difficult procedure. Naturally, then, the question arises is: *Is there a way we can fully automate the segmentation of the history of a schema in phases that represent the essence of the changes that it undergoes in a meaningful manner*? In the next chapter, we propose such a method that aims to fully automate the phase extraction and classification of a schema's life.

The study of the special topics left us with a few auxiliary conclusions that are as follows.

- The majority of zombie tables tend to survive

- Injections and ejections of attributes mostly happen at the start or mid of a table's life and rarely in the end
- Foreign keys, come in two fashions (a)foreign keys are treated as integral parts of the schema and they get born and evicted along with their tables, mostly in scientific projects, and (b) foreign keys are treated as second-class add-ons, that get removed not along with their table, especially in CMSs
- Foreign key changes for the studied datasets are small in volume

CHAPTER 4.

PHASE EXTRACTION & CLASSIFICATION

In this chapter, we present the second part of this thesis that aims in extracting patterns and motifs that apply to database evolution in order to generate a model of evolution.

To do so we apply an algorithmic procedure that uses the history of schema releases and their characteristics as input and gives a set of phases, with each phase labeled with respect to its evolution profile as output. Each subsection of this chapter analyzes a step of this procedure.

The first step of our *Phase Extraction & Classification* method is the characterization of the releases in terms of a concise vocabulary of characterizations for the nature of the maintenance process that took place during the release. We introduce the notion of *change family* and restrict the vocabulary of change families to two members: (a) growth, meaning that during the release under consideration, the aim of the maintainers was to augment the information capacity of the schema and (b) maintenance, meaning that the maintainers' aim was to improve the internal quality of the

existing schema structures rather than augment its information capacity. Apart from deriving a characterization for each release with respect to the aforementioned change families, we also measure the intensity of change for that family, too. The change family of a release is computed by the taking into consideration all types of change that took place during the release (table and attribute additions and deletions, data type changes, etc.), all measured in attributes as units. More details about this step of our method can be found in subsection 4.1.

The second step iteratively groups consecutive releases in clusters via hierarchical cluster analysis using an Agglomerative algorithm we implemented. Each such cluster of consecutive releases is considered as a *phase* of the schema's evolution. More details about this step can be found in 4.2.

The third step aims in evaluating the clustering procedure of the previous step. Since we are employing a hierarchical agglomerative clustering algorithm, the algorithm produces a sequence of solutions, i.e., segmentation of the history of the schema in phases, that starts from the most detailed solution with each release as a different cluster and ends with all the lifetime of the schema being considered one, single phase. Assuming we want to fully automate the segmentation of the schema history in phases, which of the produced solutions is the "best"? To address this problem, we need to evaluate the quality of the clustering and select the best –or the top-*k*- solution in terms of clustering quality. For each iteration of the clustering algorithm, we use a set of methods that evaluate the consistency within our clusters. Using these methods we pick the optimal clustering sets as our sets of phases. This procedure will be described in more detail in subsection 4.3.

The fourth and final step classifies each phase of the schema's evolution. Practically, the question answered in this step is: "assume any segmentation of the history in phases; can we characterize each phase with respect to the contents and aim of its updates?" The goal is different from the one of the first step, because we characterize the evolution of an entire phase and not each release separately. To achieve the final characterization, we calculate two histograms based on the change families' metrics of each phase's releases and compute the winner of each histogram. Then, based on the two histogram winners we use a set of rules based, and produce a classification that is representative of each phase's nature of evolution. This process is analyzed in subsection 4.4.

The entire process is presented as an algorithm in Figure 39.

Algorithm : Phase Extraction and Classification

Input: A list $R=\{r_1,\ldots,r_n\}$ of n releases and for each r_i a list $A=\{$ *#table_born, #table_gone, #injected, #ejected, #data_type_updated, #key_updated*$\}$
Output: a set $P=\{p_1,\ldots,p_m\}$ of m classified phases

1. foreach r_i:
   characterize it and compute its change family *metrics*
2. given the pre-computed change family *metrics* execute hierarchical Agglomerative clustering for R
   foreach merging step of the Agglomerative algorithm
   compute clustering evaluation metrics
3. Evaluate each merging step of Agglomerative based on the pre-computed clustering evaluation metrics and choose the *top clustering set P*
4. Compute histograms for P and classify it based on the histogram winners
5. Return P

Figure 39 Phase Extraction and Classification Algorithm

4.1 Release Characterization

In order to study the releases we reuse information about their database schema gathered by [Papp17]. This information, which is used as the input for our method consists of a list of releases for each dataset and for each release a vector that contains information about the following measurements:

- attribute births
- attribute deaths
- attribute injections
- attribute ejections
- attribute data type updates
- attribute key participation updates

Our goal is to translate this numbers into a characterization that represents the evolution profile of each release and a corresponding metric that will be used for the following step's clustering procedure.

We aim to use this input data to characterize each release in terms of (a) the nature of the changes performed and (b) the intensity of the evolution activity. To achieve this, we proceed as follows:

1. First, we organize the aforementioned input in three categories depending on whether they modify (a) the information capacity of the schema (in terms of new or deleted tables), (b) the information capacity of tables (in terms of new or deleted attributes inside tables), or, (c) the typing constraints of the attributes themselves. We discuss this labeling in Sections 4.1.1 and 4.1.2.

2. Second, we use this labeling to label the releases in more coarse grained categories, which we call *change families* and we also measure the extent that the evolution effort within each release aimed to provide *Growth of information capacity of the schema*, or *Maintenance of the structure of the database*. We discuss this labeling and measuring in Section 4.1.3. In Section 4.1.4, we also discuss how we quantize the intensity measurement in a small vocabulary of values for each change family that will also allow us to provide classifications later.

The final outcome of the entire process is two two-dimensional vectors for each release. The first vector contains the intensity label of the two change families *Growth* and *Maintenance* , while the second one represents the metrics of the abovementioned families.

4.1.1 Activity Characterization

The activity characterization of a release works at three levels. The first one is the *inter table change level* that concerns table births and deaths and is measured by the number of attributes that are born with or die with their tables (we use attributes as the unit of measurement for uniformity with the other categories). The second level refers to *intra table change* concerning the number of attribute injections (attributes that are added after the birth of their table) and ejections (attributes removed while their table survives). Finally, the last activity level is *amendment* that measures updates to the attribute data type and key participation of attributes. For all the levels, the activity is measured by the number of attributes involved in the respective type of change.

It should be noted that from now on attributes born with their table will be referred as *table_born*, attributes removed with the death of their table as

table_gone, while attributes that are added after the birth of their table or removed while their table survives will be referred to as *injected* and *ejected* respectively.

The following cases are counted in number of attributes and we use a threshold set to 0.3.The purpose of this threshold was to avoid cases where we have both births and deaths in the same release that indicate a renaming or restructuring case, rather than expansion or shrinking.

Inter-Table Change

- **Characterization: Table Expansion**

  Condition:

  #table_born - #table_gone> *threshold*∗(#table_born + #table_gone)

  Metric (in number of attributes):

  #table_born - #table_gone

- **Characterization: Table Shrinking**

  Condition:

  #table_gone - #table_born> *threshold*∗(#table_born + #table_gone)

  Metric (in number of attributes):

  #table_gone - #table_born

- **Characterization: Table Restructuring**

  Condition:

  Both table expansion and table shrinking checks fail

  Metric (in number of attributes):

  |#table_born - #table_gone|

Intra-Table Change

- **Characterization: Intra Table Expansion**

  Condition:

  #injected - #ejected > *threshold*∗(#injected + #ejected)

Metric (in number of attributes):

#injected - #ejected

- **Characterization: Intra Table Shrinking**

  Condition:

  #ejected - #injected > *threshold*∗(#injected + #ejected)

  Metric (in number of attributes):

  #ejected - #injected

- **Characterization: Intra Table Restructuring**

  Condition:

  Both intra table expansion and intra table shrinking checks fail

  Metric (in number of attributes):

  |#injected - #ejected|

***A*mendments**

- **Characterization: Intra Table Amendment**

  Condition:

  #data_type_updated + #key_updated>0

  Metric (in number of attributes):

  #data_type_updated + #key_updated

Each release can have at most one activity characterization per level. When a release does not have any of the activity characterizations above, its activity characterization is *No Change***.**

4.1.2 Intensity of Activity Characterization

Each characterization (except for *No Change*)is defined by its metric and is given an intensity characterization according to it. We assume three intensity levels: *Low*, *Medium* and *High*. We also assume that each characterization has a

list, containing the values that the releases have for its corresponding metric, in ascending order.

A release's activity characterization intensity is characterized as

- Low :

  If its value for the characterization's metric is lower than the value of the same metric of the release indexed in the 80% of the corresponding list.

- Medium:

  If its value for the characterization's metric is equal or higher than the value of the same metric of the release indexed in the 80% and lower than the 95% of the corresponding list.

- High:

  If its value for the characterization's metric is equal or higher than the value of the same metric of the release indexed in the 95% of the corresponding list.

4.1.3 Change Families

The first levels of characterizations provide a fairly detailed labeling of releases, with a vocabulary that is quite voluminous for being able to automatically derive classifications in the sequel. Thus, we have resorted to reducing this labeling vocabulary by grouping activity characterizations to two *change families*. The first family refers to the cases of inter or intra table expansion, where clearly the nature of the changes of the schema were related to its *Growth,* since the number of attribute additions in those cases is significant. The second family refers to inter or intra table shrinking, restructuring or intra table amendment, which indicate a *Maintenance* nature.

Maintenance

- Case:

  Table Shrinking or Table Restructuring

  Metric (in number of attributes):

  #table_born + #table_gone

- Case:

  Intra-Table Shrinking or Intra-Table Restructuring

  Metric (in number of attributes):

  #injected + #ejected

- Case:

  Intra-Table Amendment

  Metric (in number of attributes):

  #data_type_updated + #key_updated

Growth

- Case:

  Table Expansion

  Metric (in number of attributes):

  #table_born - #table_gone

- Case:

  Intra-Table Expansion

  Metric (in number of attributes):

  #injected - #ejected

We assume that a release is characterized with respect to both the *Growth* and *Maintenance* families. The value of the family metric of a release is the aggregation of the metrics of the activity characterizations it has (analyzed in 4.1.1).

4.1.4 Intensity of Change Family

Each release's growth and maintenance can be labeled with its intensity level. So, each of the two families can also be labeled as *Zero, Low, Medium* or *High*. Of course, the problem is how to compute the thresholds that differentiate subsequent intensity levels – e.g., at which amount of change do we stop labeling activity as *Low* and start labeling it *Medium*?

Our approach to the problem is based on the distribution of the values for the entire list of releases, for each of the two families. Take for example, the *Growth* family. We take all the values of the *Growth* family for all the releases of the data set and sort them in ascending order. Then, we cut this series in two places, one for the border between *Low* and *Medium* and another for the border between *Medium* and *High*. Figure 40 depicts the intensity levels for both Growth and Maintenance based on the index of the sorted release list for all the datasets. Observe the universality of the pattern of the lines; thus, we can uniformly apply the same rules for all data sets. Specifically, a release's family intensity is characterized by the following rules:

- Zero:

  The value for the family's metric is zero.

- Low:

  The value for the family's metric is lower than the value of the same metric of the release indexed in the 80% of the corresponding list.

- Medium:

  The value for the family's metric is equal or higher than the value of the same metric of the release indexed in the 80% and lower than the 95% of the corresponding list.

- High:

  The value for the family's metric is equal or higher than the value of the same metric of the release indexed in the 95% of the corresponding list.

We normalize the Growth and Maintenance metric values with respect to the max value of each metric in order to use these normalized values in the next step of the release clustering. Figure 41 depicts via scatter charts the values of normalized Growth and Maintenance metrics for all datasets. For the largest datasets we provide zoomed-in areas of these scatter charts.

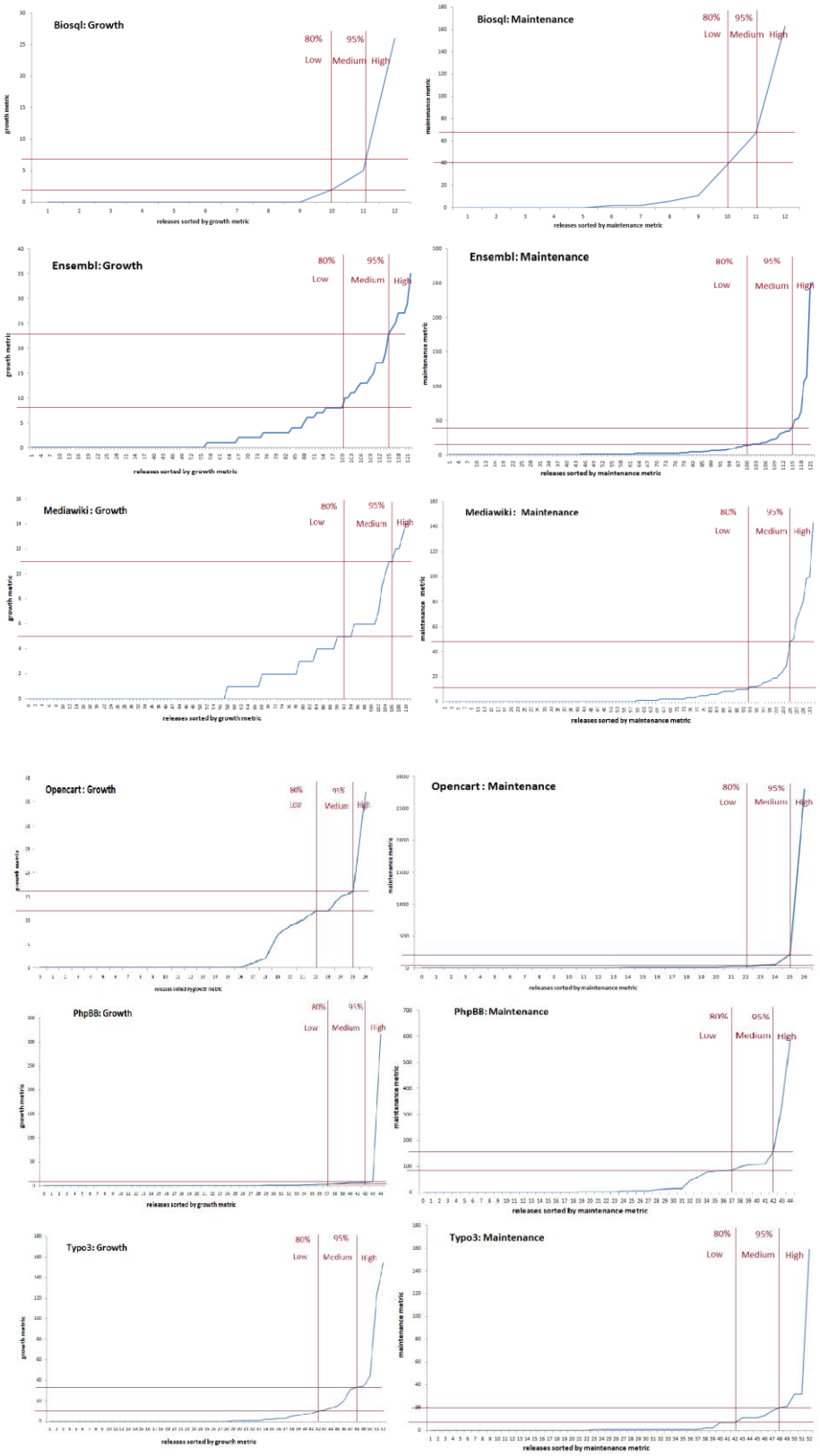


Figure 40 Intensity Levels of Change Families

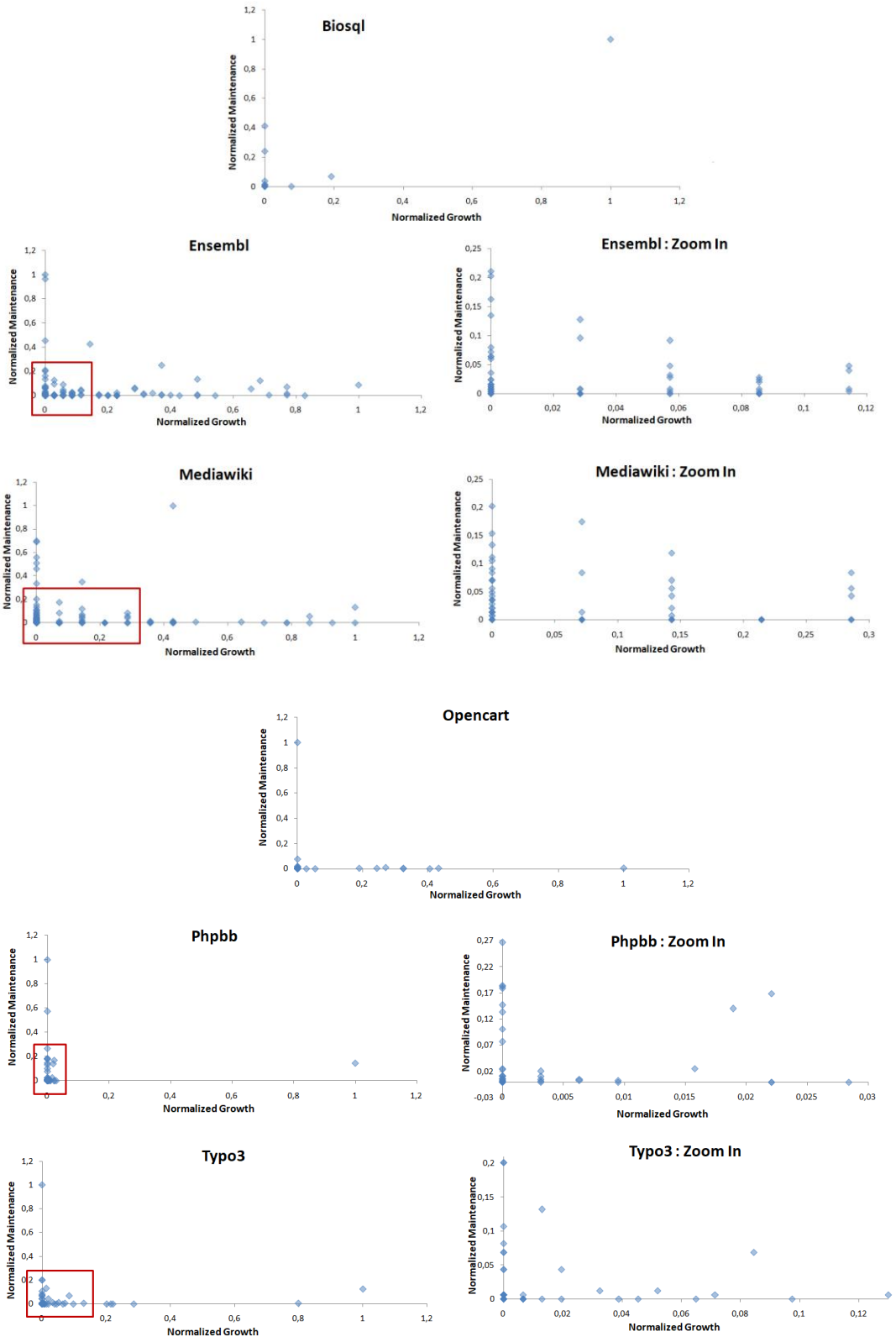


Figure 41 Normalized Growth and Maintenance scatter charts for all datasets

The characterization rules can be summarized as shown in Figure 42.

Activity	Condition	Metric	Family	Type of Change
Table Expansion	*#table_born-#table_gone>0.3(#table_born+#table_gone)*	*#table_born-#table_gone*	Growth	Inter-Table
Table Shrinking	*#table_gone-#table_born>0.3(#table_born+#table_gone)*	*#table_born+#table_gone*	Maintenance	Inter-Table
Table Restructuring	*else*	*#table_born+#table_gone*	Maintenance	Inter-Table
Intra Table Expansion	*#injected-#ejected>0.3(#injected+#ejected)*	*#injected-#ejected*	Growth	Intra-Table
Intra Table Shrinking	*#ejected-#injected>0.3(#injected+#ejected)*	*#injected+#ejected*	Maintenance	Intra-Table
Intra Table Restructuring	*else*	*#injected+#ejected*	Maintenance	Intra-Table
Intra Table Amendment	*#data_type_updated+#key_updated>0*	*#data_type_updated+#key_updated*	Maintenance	Amendment

Figure 42 Release Characterization Rules

4.2 Release Clustering

The second step of our algorithm performs hierarchical agglomerative clustering using the vectors produced by the previous step. The main goal of this step is to merge similar consecutive releases to clusters in order to create candidate phases and calculate evaluation measures for each candidate segmentation of the schema, which are needed for the next step of our method.

We implemented a Hierarchical Agglomerative Clustering algorithm that takes as input the releases of the schema and their Change Family metrics' values [4.1.3]. As it is known, an agglomerative clustering algorithm starts with each point (here a release) as a different cluster and, progressively, at each step, merges the two closest clusters until only one cluster (or k clusters if accordingly set) remains. In our setting, the proximity matrix of the Agglomerative algorithm is updated in each step using the *average-linkage* distance method.

To calculate the *distance* that two releases have with each other we use the Maintenance and Growth metrics described in 4.1.3. For each release we normalize the values of the maintenance and growth metric with respect to the maximum value of the corresponding metric. We define the distance between two releases as the *Euclidean distance of the normalized growth and maintenance values*.

Notation: we denote the normalized maintenance score of a release R_i with M_i^*, the normalized growth score with G_i^* and the distance of two releases via δ.

$$\delta(R_1, R_2) = \sqrt{(M_1^* - M_2^*)^2 + (G_1^* - G_2^*)^2}$$

One of the distinctive characteristics of our method is the need to customize the agglomerative clustering algorithm to our setting. Our intention is to cluster together *adjacent* releases, so as to construct contiguous phases. Thus, there is no need to attempt assessing the distance (and thus, to possibly merge) phases that are not adjacent. Thus, *our implementation only merges clusters/releases that are consecutive in time.*

The entire process is presented as an algorithm in Figure 43.

Algorithm : Candidate Phase Extraction via Agglomerative Hierarchical Clustering

---

Input: A list $R=\{r_1,...,r_n\}$ of n releases and for each r_i a *normalized_growth* value and a *normalized_maintenance* value

Output: a set $C=\{c_1,...,c_m\}$ of m cluster sets (each one being a merging step of the algorithm) and for each c_i the *silhouette*, *cohesion* and *separation* values

1. Start with each release r_i as a separate cluster
2. For each pair r_i, r_j , where r_j is adjacent to r_i, compute the Euclidean distance of their *normalized_growth* value and the *normalized_maintenance* value
    a. Merge the two closest adjacent releases
    b. Compute the *silhouette, cohesion* and *separation* values of the current cluster set c_i and add it to C
3. Return C

Figure 43 Algorithm : Candidate Phase Extraction via Agglomerative Clustering

4.3 Clustering Evaluation

After the clustering procedure of each dataset is done, we need to evaluate each step of the agglomerative algorithm in order to decide which clustering step is the best and should be used as a guide to extract phases of the schema's life. In order to decide which solution should be considered as the best one, we need to ensure the quality of the solution both based on clustering evaluating measures and the similarity with our ground truth. Both types of measures used will be described in this subsection.

Silhouette. The first of these methods is *Silhouette*[Rous87], a method of interpretation and validation of consistency within clusters of data. This method shows how well a point (here a release) is matched with its cluster. It uses the average dissimilarity that a certain point has with every point in the same cluster and the average dissimilarity it has with every point in every cluster it is not a part of. The original *Silhouette* type for a data point p_i is as follows:

$$s(p_i) = \frac{b(p_i) - a(p_i)}{max\{b(p_i), a(p_i)\}}, \text{ with}$$

$b(p_i)$ the lowest average dissimilarity of p_i to any other clusters to which it does not belong to, and

$a(p_i)$ the average dissimilarity of p_i to the rest of the points within the cluster to which p_i currently belongs to

The lowest value ***Silhouette*** can have is -1, while the highest is 1. In order to say that we have a well clustered dataset we want a *Silhouette* value that is very close to 1. Typically, Silhouette values in the range of 0.2 to 0.5 are considered to be fair.

Because our clustering implementation only considers merging consecutive clusters in time, for our purposes we modified the original *Silhouette* type so that the lowest average dissimilarity (b(i)) is computed only for adjacent clusters.

Assume a cluster set **C** = {C_1, …,C_n} which is a possible clustering of the history of a schema in phases. Then, we define the *adjacency clustering silhouette value*, denoted as s^{adj} for an arbitrary release R_i is defined as

$$s^{adj}(R_i) = \frac{b^{adj}(R_i) - a(R_i)}{max\{b^{adj}(R_i), a(R_i)\}}, \text{ with}$$

$b^{adj}(R_i)$ the lowest average dissimilarity of R_i to the clusters adjacent to the one R_i currently belongs to, and

$a(R_i)$ the average dissimilarity of R_i to the rest of the releases in the cluster to which R_i currently belongs to

Finally, the *adjacency clustering silhouette value* for a cluster set is the average value of the respective adjacency clustering silhouette values of all the clusters belonging to the cluster set.

$$s^{adj}(\mathbf{C}) = avg\{s^{adj}(C_1), \ldots, s^{adj}(C_n)\}$$

Figure 44 shows the *Silhouette* value of each merging step of the Agglomerative algorithm, for every dataset.

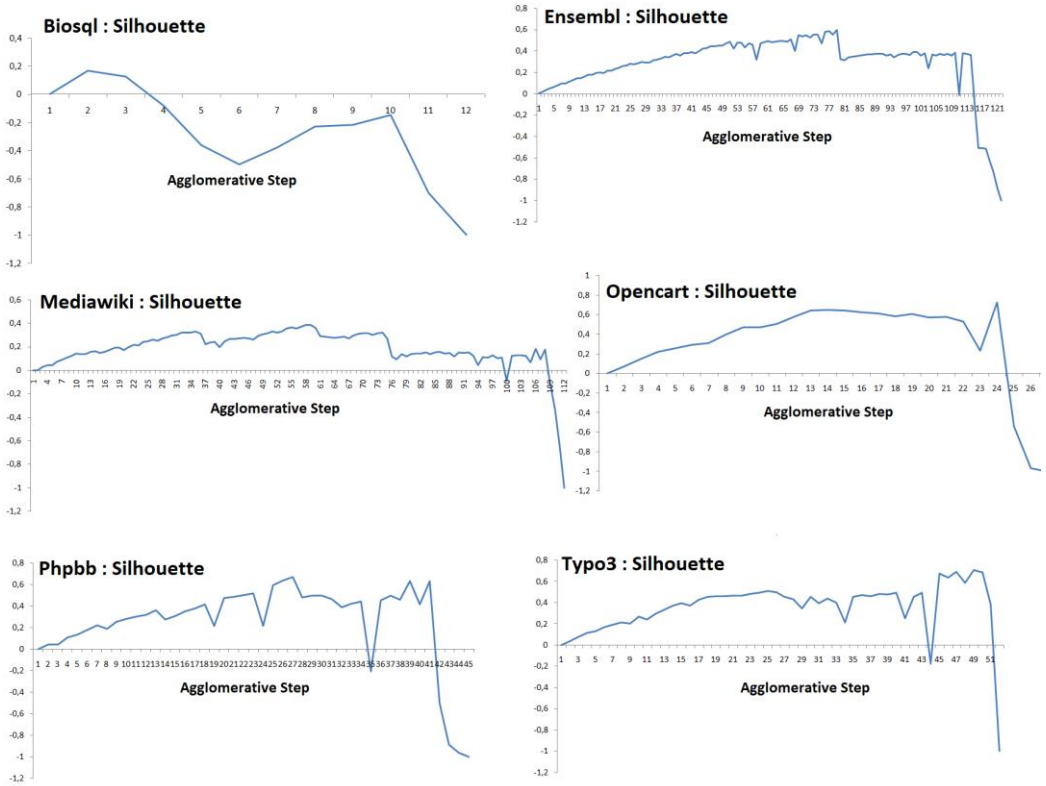


Figure 44 Silhouette values for each merging step of the Agglomerative algorithm and Silhouette distances for every transition of merging steps.

Cohesion and Separation. The second set of evaluation methods we use includes the combination of two metrics, namely *cohesion* and *separation*. Cohesion and separation in [TaSK05] are defined as the following two formulae:

$$Cohesion(C_i) = \sum_{\substack{x \epsilon Ci \\ y \epsilon Ci}} proximity(x, y)$$

$$separation(C_i, C_j) = \sum_{\substack{x \epsilon Ci \\ y \epsilon Cj}} proximity(x, y)$$

We decided to intervene in the definition of the two measures. The reason lies in the fact that our clustering implementation is a modification of the original Agglomerative algorithm (it only considers consecutive clusters for merging), so we modified the two formulae in an appropriate manner with respect to out implementation. We define the *cohesion* value of a cluster as the opposite

of the average distance that every release has with all releases in the same cluster.

We assume a schema history including n releases and a cluster set $\mathbf{C} = \{c_1, \ldots, c_m\}$ containing m possible segmentations of the schema history. We also define $\mathbf{dist}(r_i, r_j)$ the distance that two of those releases have with each other. Assume a release r_i belonging to cluster c. We define the *ClusterFitness* of r_i as follows:

$$ClusterFitness(r_i) = -\ \text{avg}\{\ \mathbf{dist}(r_i, r_j)\ \},\ r_i \neq r_j \text{ and } r_i, r_j \in c$$

We assume a cluster set $\mathbf{C}$ over $\{r_1, \ldots, r_n\}$ with n releases. The *cohesion* value of $\mathbf{C}$ is defined as follows:

$$Cohesion(\mathbf{C}) = \text{avg}\{\ ClusterFitness(r_i)\ \},\ \text{foreach } r_i \in \mathbf{C}$$

The lowest value cohesion can have is the opposite of the largest distance there is for each dataset, while the highest is 0, where all releases in the same cluster have 0 distance. The higher the cohesion value, the better the clustering step, as we want each point to have the smallest distance possible with all the other points in the same cluster.

We define the *separation* of a cluster as the average distance that every release has with all releases in adjacent clusters.

We define a cluster set $\mathbf{C} = \{C_1, \ldots, C_n\}$ with n clusters, a cluster $C_i = \{r_1, \ldots, r_m\}$ with m releases and $\mathbf{dist}(r_i, r_j)$ the distance that two of those releases have with each other. Each release r_i has a *separation* value as follows:

$$Separation(r_i) = \text{avg}\{\ \mathbf{dist}(r_i, r_j)\ \},\ r_i \neq r_j\ , r_i \in C_i\ , r_j \in C_j \text{ with } C_i \neq C_j \text{ and } C_i \text{ and } C_j \text{ are adjacent}$$

We assume a cluster set $\mathbf{C} = \{r_1, \ldots, r_n\}$ with n releases. The *separation* value of $\mathbf{C}$ is defined as follows:

$$Separation(\mathbf{C}) = \text{avg}\{\ Separation(r_i)\ \},\ \text{foreach } r_i \in \mathbf{C}$$

The lowest value separation can have is0, while the highest is the largest distance there is for each dataset. The higher the separation value, the better the clustering step, as we want each point of each cluster to have the highest distance possible with all points from other clusters.

We normalize the average separation and cohesion values of each cluster set with respect to the min and max corresponding values as follows:

NormalizedCohesion:

$$normalizedCohesion(C_i) = \frac{cohesion(C_i) - \min Cohesion}{\max Cohesion - \min Cohesion}$$

NormalizedSeparation:

$$normalizedSeparation(C_i) = \frac{separation(C_i) - \min Separation}{\max Separation - \min Separation}$$

We can combine these two values in a single metric *Normalized Clustering Quality (NCQ)*

$$NCQ(C_i) = normalizedSeparation(C_i) + normalizedCohesion(C_i)$$

Figure 45 depicts the normalized cohesion and normalized separation values, while Figure 46 depicts the sum of the two normalized metrics. Both figures show the corresponding values for each Agglomerative step and for each dataset.

In order to extract the best phases, we considered all the possible evaluation metrics presented in figures 44, 45 and 46. The Silhouette metric for the majority of the datasets has its highest values for cluster sets with quite high number of clusters in the orders of half the number of releases (practically, average cluster size is about 2 in these solutions). Having empirically evaluated the phases of each dataset in Section 3.2, we consider such solutions not desired.

When considering the sum of Normalized Cohesion and Normalized Separation, however (Figure 46), we observe that there is a critical area of approximately 10 or less clusters (practically the area of solutions at the rightmost part of the respective plots in Figure 46) where (a) the combination of the two measures gives a satisfactory compromise, (b) the number of clusters is close to the golden standard of Section 3.2, (c) the sum is maximized, and (d) the value of the measure reaches fairly higher values than the rest of the plot, for several points in this critical area.

Our analysis is thus, restricted to this critical area of potential solutions; as we shall see, the most compatible solution with respect to the golden standard is also found in it. We have also observed that this area has the second best (or even best for Typo3) silhouette values.

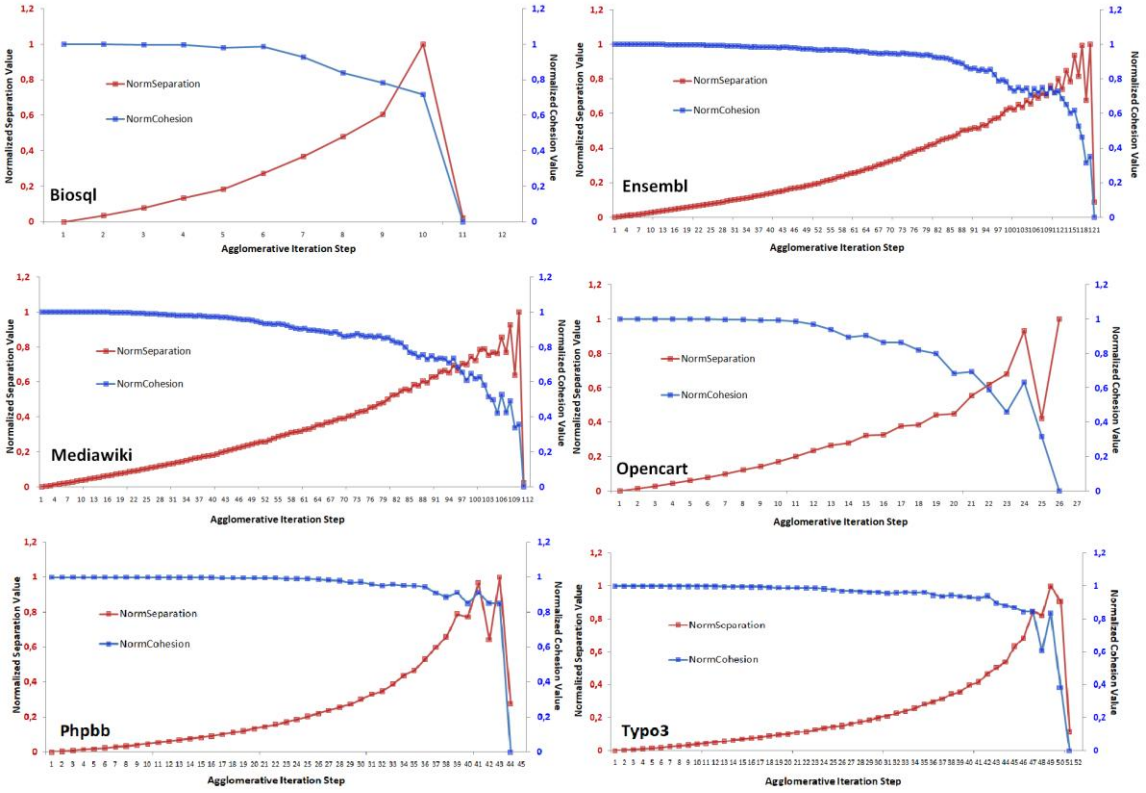


Figure 45 Normalized Cohesion-Separation values for each Agglomerative iteration step for all datasets

A final evaluation procedure we considered was to compare the differences (in phases) of our golden standard with the phase extraction of the top points of *NCQ* (seen in Figure 46). To do so, we calculated a new measure, *Lag*, that is defined as follows. We assume a set **H** = {r_1, …,r_n} with n releases. We label each release with a schema history including the phase to which it belongs and denote this labeling as $phase(r_i) \in N$. Each $phase(r_i)$ is a number with the phase r_i belongs to, e.g., if $phase(r_3)$ =1, this means that r_3 belongs to phase 1. Assume now two different segmentations of the history, for example a manual golden standard $T=\{phase^T(r_1),\ldots, phase^T(t_n)\}$ and an algorithmically produced segmentation $A=\{phase^A(r_1),\ldots, phase^A(t_n)\}$. Then the *Lag* between the two clusterings is the average difference of phase id, over all releases, for the two labeling. *Lag* is defined as follows:

$$Lag(A,T) = \frac{\sum_{i=1}^{n} abs(phase^A(r_i) - phase^T(r_i))}{n}$$

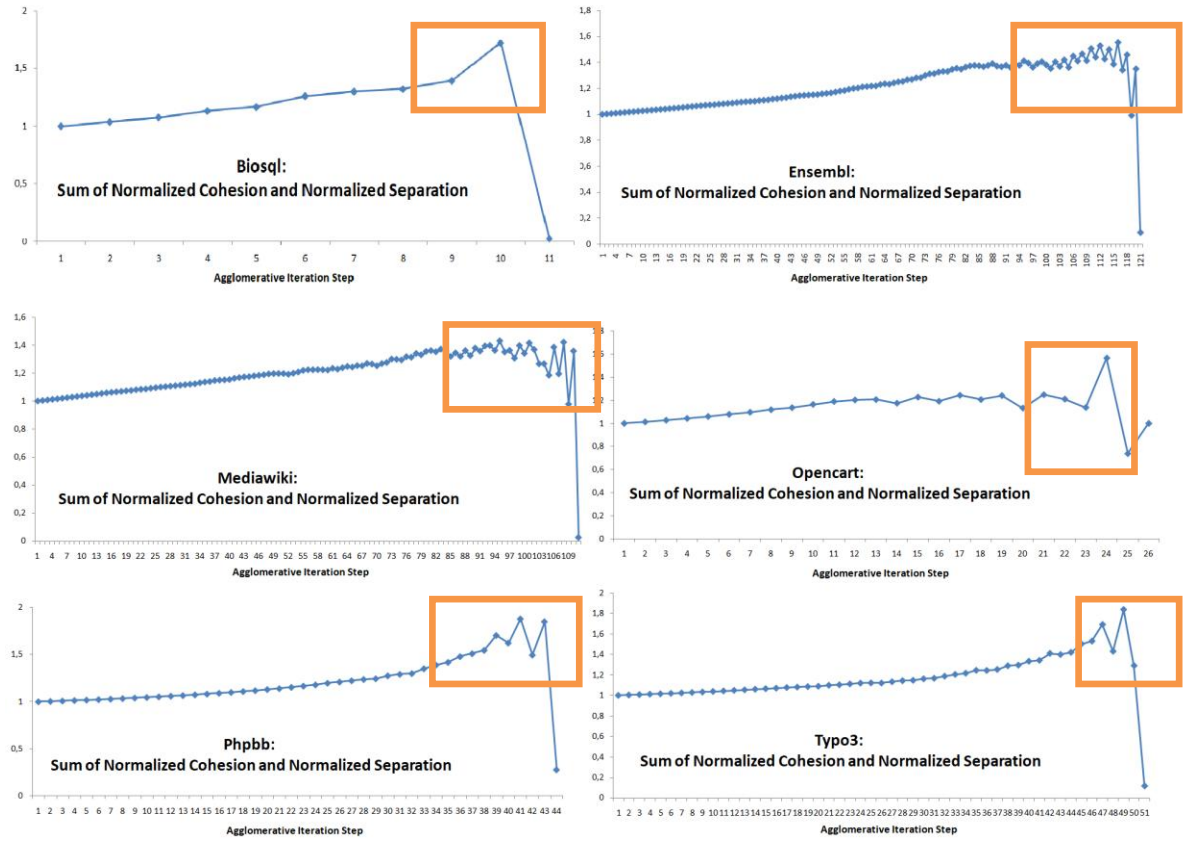


Figure 46 Sum of Normalized Cohesion and Normalized Separation values for each Agglomerative iteration step for all datasets. The critical area of candidate final clusterings is depicted in the orange box for each data set.

Figure 47 depicts the top candidate algorithmically produced solution per dataset. We examine the top solutions based on NCQ, find the *Lag* value of misclassified releases per solution and rank them with respect to both *Lag* and *NCQ*.

	Biosql (#clusters@goldenStandard=3)				Ensembl (#clusters@goldenStandard=8)				Mediawiki (#clusters@goldenStandard=7)			
	#clusters	Lag	Rank wrt Lag	Rank wrt NCQ	#clusters	Lag	Rank wrt Lag	Rank wrt NCQ	#clusters	Lag	Rank wrt Lag	Rank wrt NCQ
Solution 1	3	0	1	1	7	1,17	1	1	5	0,93	1	2
Solution 2					9	1,2	2	4	12	3,75	3	3
Solution 3					11	1,55	3	2				
Solution 4					13	2,13	4	3				

	Opencart (#clusters@goldenStandard=4)				Phpbb (#clusters@goldenStandard=5)				Typo3 (#clusters@goldenStandard=6)			
	#clusters	Lag	Rank wrt Lag	Rank wrt NCQ	#clusters	Lag	Rank wrt Lag	Rank wrt NCQ	#clusters	Lag	Rank wrt Lag	Rank wrt NCQ
Solution 1	4	1,7	2	1	3	1,02	3	2	4	2,15	3	1
Solution 2	7	0,96	1	2	5	0,78	2	1	6	1,83	2	2
Solution 3	11	2,56	3	3	7	0,71	1	3	7	1,33	1	3
Solution 4												

Figure 47 Top candidate algorithmically produced solutions per dataset

We observe that the value of misclassified releases for the top *NCQ* solutions mainly ranges from 0 to 2. This means that our algorithm performs well, as the top solutions of *NCQ* that our method produces, are very similar with those of Section's 3.2 golden standard. We additionally see that, the solution with the minimum *Lag* value, for every single one of the studied datasets, lies within the top-3 solutions with respect to *NCQ*, thus making *NCQ* the best possible measure for evaluating the solutions.

The details of the algorithmically produced best solutions, along with the classification of their phases, are presented in 4.5. Before that, though, in Section 4.4, we present the method that produces these classifications.

4.4 Phase Classification

Once the third step of clustering evaluation and phase extraction is completed and we have the winner solution for each dataset, we now must classify each release with respect to its evolution profile. The release characterizations we produced during the first step of our method are not enough, as we want the characterization of a phase to be representative of the evolution activity as a group and not based on each release's individual evolution. For this reason, we continue with the next step of phase classification. The procedure of phase classification consists of two steps.

The first step is the computation of histograms for each phase of the schema's life. We compute one histogram for the intensity of *Growth* and another for the intensity of Maintenance, for each phase (i.e., cluster of contiguous releases) of a candidate segmentation of the history of a schema. The second step is a rule-based classification method based on these histograms and ultimately labeling a phase with respect to the nature and intensity of the changes it performs.

4.4.1 Histogram Computation

Each phase is a set of consecutive releases of a schema's life. Each release as mentioned above has been labeled with change families and the intensity of these families.

A release has a measure for the Growth family, and another for the Maintenance family. This means that each release is considered as a two

dimensional (Growth, Maintenance) vector. Based on this measure, each of these families is labeled with a *Zero*, *Low*, *Medium*, or *High* intensity.

For each phase of the schema, which is ultimately a list of adjacent releases, we compute the histogram for Growth and Maintenance. For each of these two dimensions we count how many releases had a *Zero*, *Low*, *Medium*, and *High* value, thus producing four buckets for the histogram. Then, the histogram of the corresponding phase for the corresponding change family is computed as the percentage of releases in the phase that have the corresponding dimension value.

Figure 48 shows an example of the Agglomerative clustering of Mediawiki with 5 clusters. Each color represents a cluster. The x axis represents the date of the latest commit of each release; the positive numbers of the y axis show the normalized growth and the negative numbers the normalized maintenance. It should be noted that the chart of Figure 48 has an offset of 0.05 for both the normalized Growth and Maintenance values in order to be able to visually depict the *Zero* values in a distinctive way. Figure 49 shows the growth and maintenance histograms computed for each cluster shown in Figure 48.

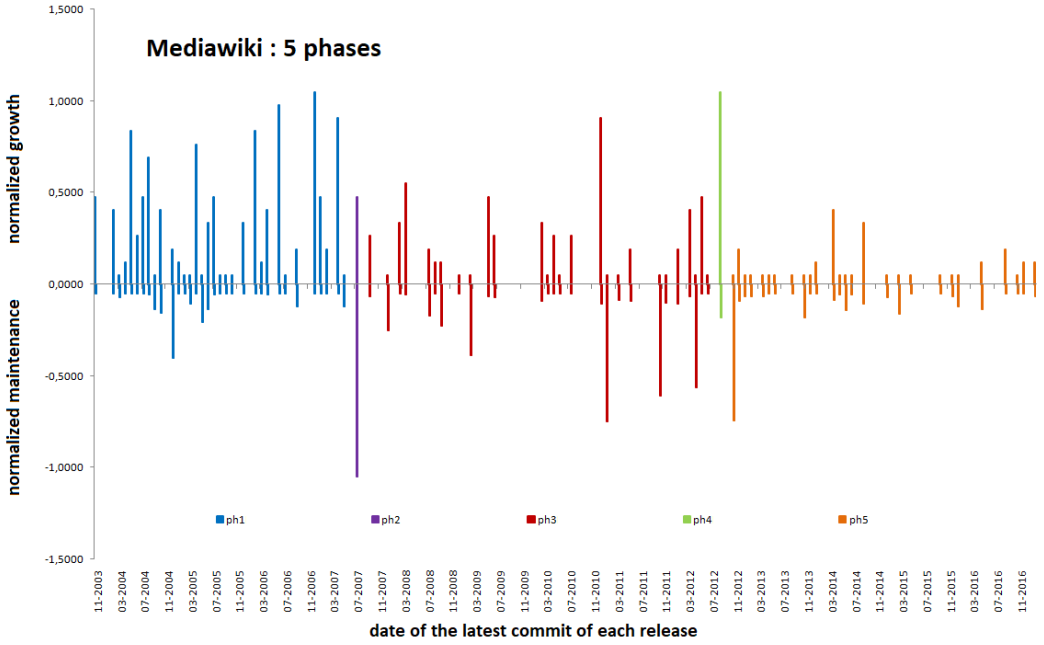


Figure 48 Agglomerative clustering example of Mediawiki with 4 clusters

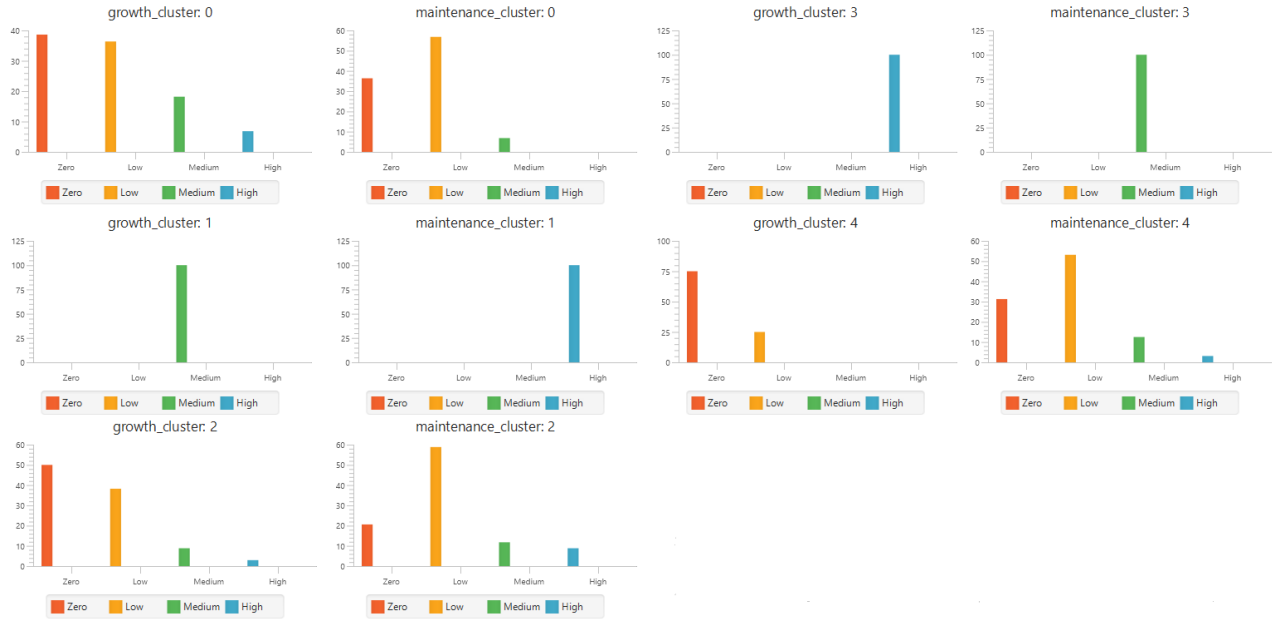


Figure 49 Growth and Maintenance histograms of the clusters of Mediawiki shown in Figure 48

4.4.2 Classification

After having computed the *Growth* and *Maintenance* histograms of the corresponding phases, the remaining step is their classification. For the classification process we follow a rule-based procedure with respect to the possible values of the two dimensions of the winner release. As already mentioned each phase consists of a list of adjacent releases and has two histograms, one for each of the two change families.

A phase will be characterized according to the value of the winner of each histogram. The *Zero* and *Low* intensities dominate the histograms most of the times as the calmness seems to be the default state of the schema. Figure 50 depicts the detailed percentages of Growth and Maintenance intensity percentages for all 6 datasets. More specifically, the percentage of releases that have *Zero* or *Low* Growth for our datasets is between 83% and 89% and the percentage of releases that have *Zero* or *Low* Maintenance is between 82% and 87%. Thus, the importance of having *Zero* or *Low* intensity is small and the occurrences of *Medium* or *High* intensities have greater significance.

On the contrary, *High* intensities are very infrequent and are the most important ones. *High* Growth and *High* Maintenance percentages both range from 4% to 8%. *Medium* intensities, although not so rare as *High*, are also quite infrequent. *Medium* Growth percentages are between 7% and 12% for our datasets, and *Medium* Maintenance between 8% and 13%.

Keeping these facts in mind, higher intensities take precedence on lower ones. This does not mean that if we have a phase of several *Low* intensity releases and only one *High* intensity release the entire phase will be characterized by a single release. We consider the duration of a phase in number of releases. A set of high intensity releases of a phase will characterize it only if it is at least the one quarter of the phase's population. The rules of the winner selection for both histograms are as follows:

→ **If** *High%>threshold* **then** winner=*High*

→ **Else if** *Medium%>threshold* **then** winner=*Medium*

→ **Else if** *Low%>threshold* **then** winner=*Low*

→ **Else** winner=*Zero*

The threshold is empirically set to 0.25 to honor the one quarter of the population rule we discussed above. Different thresholds were tested and due to this empirical study we consider the 0.25 threshold as the best possible.

	Growth				
dataset	**%Zero**	**%Low**	**%Zero+%Low**	**%Medium**	**%High**
Biosql	75%	8%	**83%**	8%	8%
Ensembl	45,9%	37,7%	**83,6%**	11,5%	4,9%
Mediawiki	51,8%	33%	**84,8%**	10,7%	4,5%
Opencart	63%	25,9%	**88,9%**	7,4%	3,7%
Phpbb	64,4%	20%	**84,4%**	11,1%	4,4%
Typo3	53,8%	28,8%	**82,6%**	11,5%	5,8%

	Maintenance				
dataset	**%Zero**	**%Low**	**%Zero+%Low**	**%Medium**	**%High**
Biosql	33,3%	50%	**83,3%**	8,3%	8,3%
Ensembl	20,5%	62,3%	**82,8%**	12,3%	4,9%
Mediawiki	29,5%	55,4%	**84,9%**	10,7%	4,5%
Opencart	29,6%	55,6%	**85,2%**	11,1%	3,7%
Phpbb	17,8%	64,4%	**82,2%**	13,3%	4,4%
Typo3	19,2%	67,3%	**86,5%**	7,7%	5,8%

Figure 50 Growth/Maintenance intensity percentages

Based on the winner values of each dimension and each phase we define the following phase classification rules:

1. **Class:**

   Minor Activity (with growth/maintenance spike(s))

   Condition:

   {(*Low* or *Zero*) *Growth*, (*Low* or *Zero*) *Maintenance*}

2. **Class:**

   (*Medium* or *High*) *Maintenance*

   Condition:

   {(*Low* or *Zero*) *Growth*, (*Medium* or *High*) *Maintenance*}

3. **Class:**

   (*Medium* or *High*) *Growth*

   Condition:

   {(*Medium* or *High*) *Growth*, (*Low* or *Zero*) *Maintenance*}

4. **Class:**

   High Maintenance – Medium Growth

   Condition:

   {*Medium Growth*, *High Maintenance*}

5. **Class:**

   High Growth – Medium Maintenance

   Condition:

   {*High Growth*, *Medium Maintenance*}

6. **Class:**

   Restructuring

Condition:

{*Medium Growth, Medium Maintenance*}

7. **Class:**

   Intense Evolution

   Condition:

   {*High Growth, High Maintenance*}

Figure 51 depicts a mapping of the aforementioned phase classification rules.

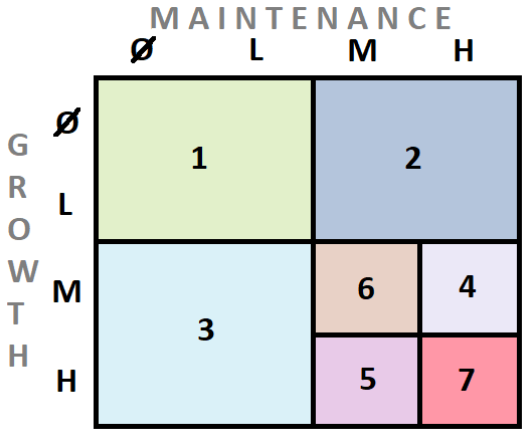


Figure 51 Mapping of Phase Characterization Rules

4.5 Top Phase Extractions

As described in 4.3 and seen in Figures 46, 47 we present the best phase extractions for the studied datasets along with their classification based on the rules of 4.4.2

All the Figures of this subsection show a table with the basic statistics of each phase extraction solution (*growth*, *maintenance* winners, percentages of each intensity, classification etc.) and each phase is colored the same color as it is shown in the chart below the statistics table. All charts' x-axis' points represent the date the last commit of each release was made. Y-axis' positive numbers represent the *normalized growth* value of each release and Y-axis' negative numbers represent the *normalized maintenance* values. It should be noted that each release point has an offset of 0.05.

Biosql. Biosql is a set with a small number of releases (12) and an easy dataset to perform phase extraction to. Our golden standard totally agrees with the top solution our method produced. There are zero misclassified releases and the phase extraction and classification of Biosql consists of 3 phases and is presented in Figure 52.

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,80	0,00	0,20	0,00	0,00	0,80	0,20	0,00	Zero	Low	Minor Activity with growth and maintenance spike(s)
1	0,00	0,00	0,00	1,00	0,00	0,00	0,00	1,00	High	High	Intense Evolution
2	0,83	0,17	0,00	0,00	0,67	0,33	0,00	0,00	Zero	Low	Minor Activity

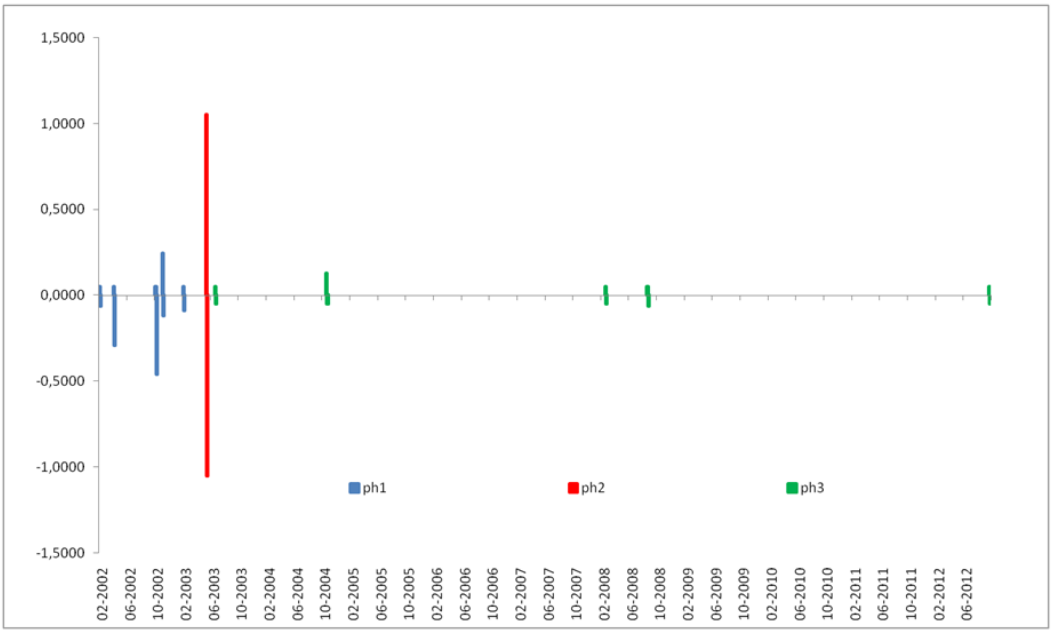


Figure 52 Phase Extraction and Classification of Biosql with 3 phases

For Biosql there was a phase of *Minor Activity* with some growth and maintenance spikes then a huge spike of *Intense Evolution* and then finally a long period of *Minor Activity*.

Ensembl. Ensembl is the dataset with the largest amount of releases and based on the best possible phase extraction of our method contains 7 phases and is presented in Figure 53. Other possible extractions based on our evaluation are presented in Figures 54, 55 and 56 with 9, 11 and 13 phases respectively.

Ensembl's first phase is classified as a period of *Medium Maintenance* followed by a spike of *High Growth – Medium Maintenance*. The third phase is classified as a *Medium Growth* period and then a spike of *High Maintenance* appears. Then we have a phase of *Medium Growth* again followed by a spike of *High*

Maintenance. Finally, again we have a last long phase of *Minor Activity* with several growth and maintenance spikes.

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,50	0,50	0,00	0,00	0,00	0,00	1,00	0,00	Low	Medium	Medium Maintenance
1	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
2	0,38	0,25	0,25	0,13	0,25	0,50	0,13	0,13	Medium	Low	Medium Growth
3	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
4	0,40	0,20	0,33	0,07	0,07	0,80	0,13	0,00	Medium	Low	Medium Growth
5	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
6	0,48	0,44	0,06	0,02	0,23	0,65	0,09	0,02	Low	Low	Minor Activity with growth and maintenance spike(s)

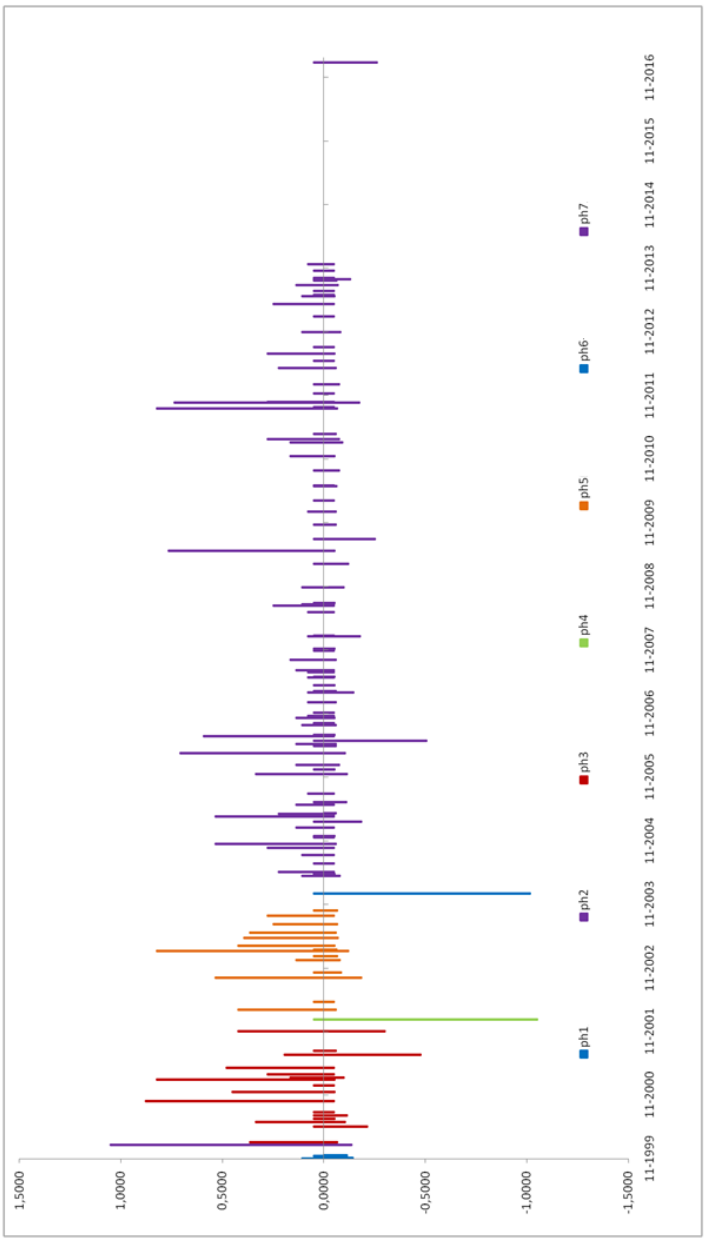
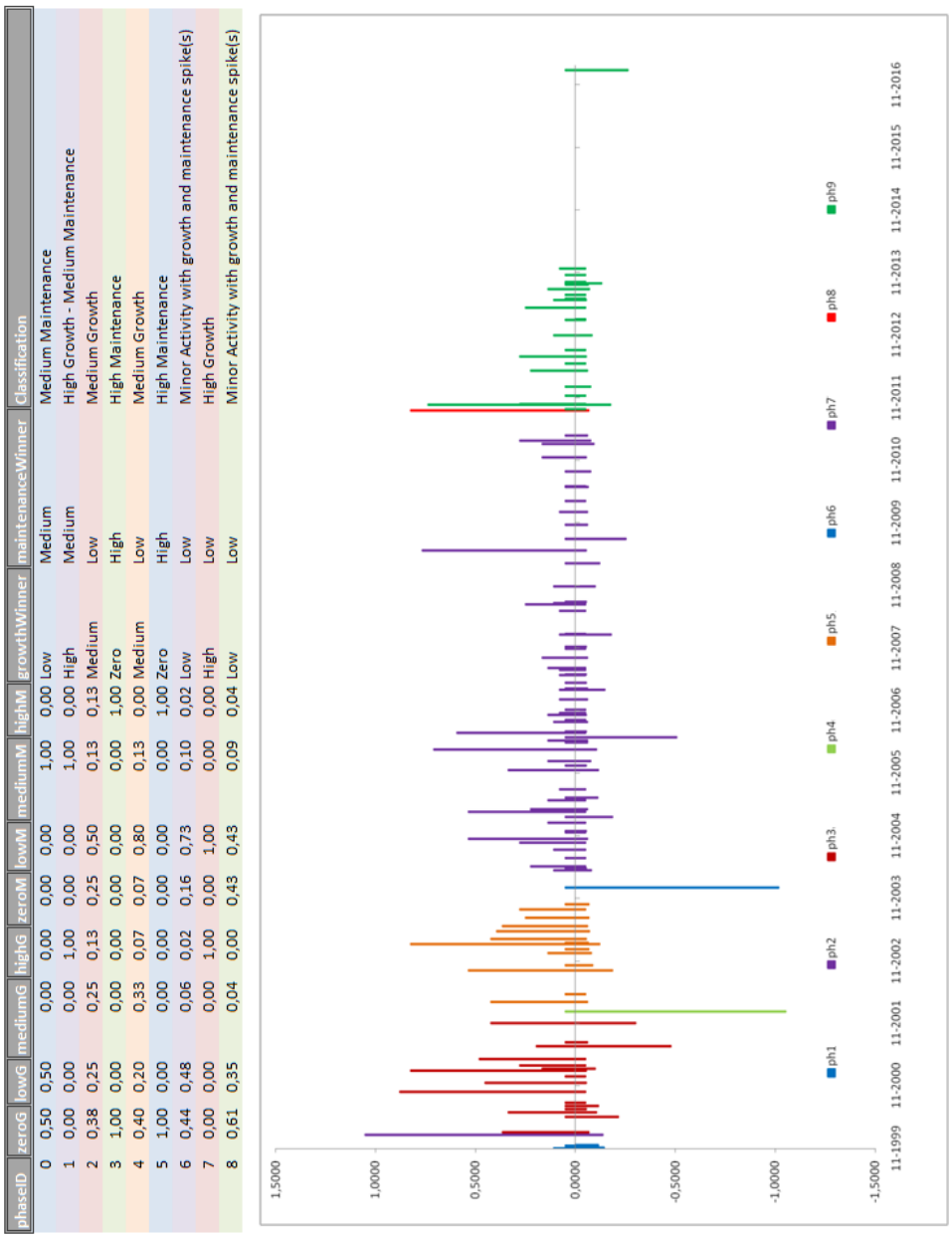


Figure 53 Phase Extraction and Classification of Ensembl with 7 phases



phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,50	0,50	0,00	0,00	0,00	0,00	1,00	0,00	Low	Medium	Medium Maintenance
1	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
2	0,38	0,25	0,25	0,13	0,25	0,50	0,13	0,13	Medium	Low	Medium Growth
3	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
4	0,40	0,20	0,33	0,07	0,07	0,80	0,13	0,00	Medium	Low	Medium Growth
5	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
6	0,44	0,48	0,06	0,02	0,16	0,73	0,10	0,02	Low	Low	Minor Activity with growth and maintenance spike(s)
7	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
8	0,61	0,35	0,04	0,00	0,43	0,43	0,09	0,04	Low	Low	Minor Activity with growth and maintenance spike(s)

Figure 54 Phase Extraction and Classification of Ensembl with 9 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,50	0,50	0,00	0,00	0,00	0,00	1,00	0,00	Low	Medium	Medium Maintenance
1	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
2	0,38	0,25	0,25	0,13	0,25	0,50	0,13	0,13	Medium	Low	Medium Growth
3	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
4	0,40	0,20	0,33	0,07	0,07	0,80	0,13	0,00	Medium	Low	Medium Growth
5	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
6	0,44	0,48	0,06	0,02	0,16	0,73	0,10	0,02	Low	Low	Minor Activity with growth and maintenance spike(s)
7	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
8	1,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00	Zero	Zero	Minor Activity
9	0,00	0,00	1,00	0,00	0,00	0,00	1,00	0,00	Medium	Medium	Restructuring
10	0,62	0,38	0,00	0,00	0,43	0,48	0,05	0,05	Low	Low	Minor Activity with maintenance spike(s)

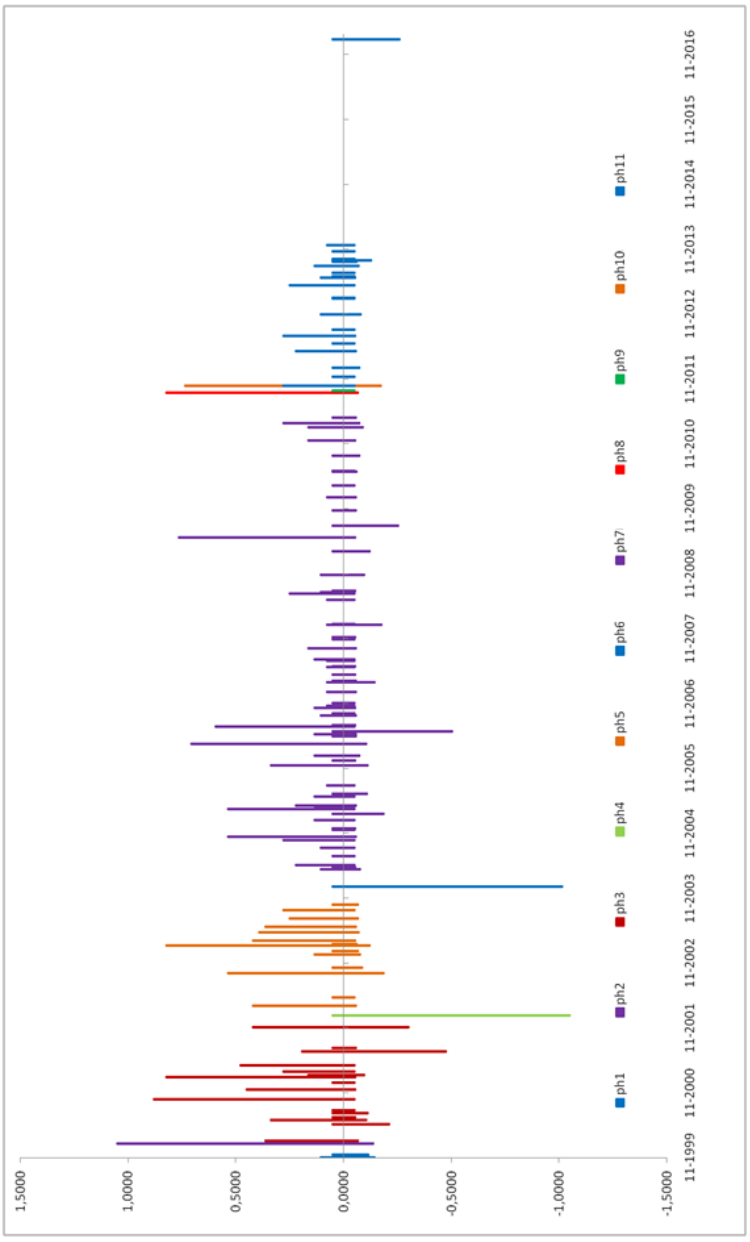


Figure 55 Phase Extraction and Classification of Ensembl with 11 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,50	0,50	0,00	0,00	0,00	0,00	1,00	0,00	Low	Medium	Medium Maintenance
1	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
2	0,38	0,25	0,25	0,13	0,25	0,50	0,13	0,13	Medium	Low	Medium Growth
3	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
4	0,40	0,20	0,33	0,07	0,07	0,80	0,13	0,00	Medium	Low	Medium Growth
5	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
6	0,40	0,52	0,08	0,00	0,16	0,72	0,10	0,02	Low	Low	Minor Activity with growth and maintenance spike(s)
7	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
8	0,64	0,36	0,00	0,00	0,18	0,73	0,09	0,00	Low	Low	Minor Activity with maintenance spike(s)
9	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
10	1,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00	Zero	Zero	Minor Activity
11	0,00	0,00	1,00	0,00	0,00	0,00	1,00	0,00	Medium	Medium	Restructuring
12	0,62	0,38	0,00	0,00	0,43	0,48	0,05	0,05	Low	Low	Minor Activity with maintenance spike(s)

Figure 56 Phase Extraction and Classification of Ensembl with 13 phases

Mediawiki. Mediawiki also consists of a large number of releases and the top solution our method produces consists of 5 phases and is presented in Figure 57. Another fair possible extraction is presented in Figure 58 and consists of 12 phases.

Mediawiki's first phase is a long period that is mostly *Minor Activity* but contains a lot of big growth and maintenance spikes. Then we have a phase (spike) of *Medium Growth – High Maintenance* followed by a long phase of *Minor Activity* that contains spikes of growth and maintenance. The fourth phase of Mediawiki is a spike of *High Growth – Medium Maintenance* followed by a long period of *Minor Activity* with some maintenance spikes.

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,39	0,36	0,18	0,07	0,36	0,57	0,07	0,00	Low	Low	Minor Activity with growth and maintenance spike(s)
1	0,00	0,00	1,00	0,00	0,00	0,00	0,00	1,00	Medium	High	Medium Growth - High Maintenance
2	0,50	0,38	0,09	0,03	0,21	0,59	0,12	0,09	Low	Low	Minor Activity with growth and maintenance spike(s)
3	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
4	0,75	0,25	0,00	0,00	0,31	0,53	0,13	0,03	Low	Low	Minor Activity with maintenance spike(s)

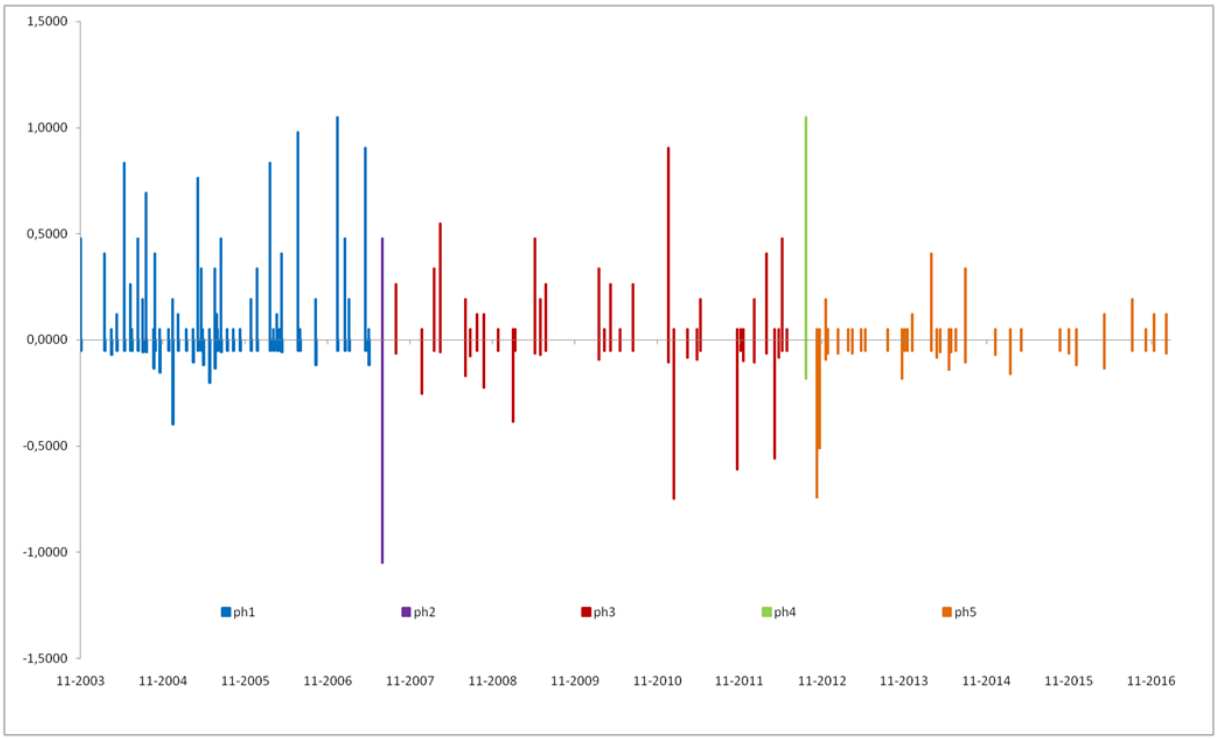


Figure 57 Phase Extraction and Classification of Mediawiki with 5 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,42	0,39	0,19	0,00	0,39	0,53	0,08	0,00	Low	Low	Minor Activity with growth and maintenance spike(s)
1	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
2	0,50	0,50	0,00	0,00	0,50	0,50	0,00	0,00	Low	Low	Minor Activity
3	0,00	0,00	0,00	1,00	1,00	0,00	0,00	0,00	High	Zero	High Growth
4	0,25	0,25	0,25	0,25	0,00	1,00	0,00	0,00	High	Low	High Growth
5	0,00	0,00	1,00	0,00	0,00	0,00	0,00	1,00	Medium	High	Medium Growth - High Maintenance
6	0,40	0,50	0,10	0,00	0,25	0,55	0,20	0,00	Low	Low	Minor Activity with growth and maintenance spike(s)
7	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
8	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
9	0,67	0,25	0,08	0,00	0,17	0,67	0,00	0,17	Low	Low	Minor Activity with growth and maintenance spike(s)
10	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
11	0,75	0,25	0,00	0,00	0,31	0,53	0,13	0,03	Low	Low	Minor Activity with maintenance spike(s)

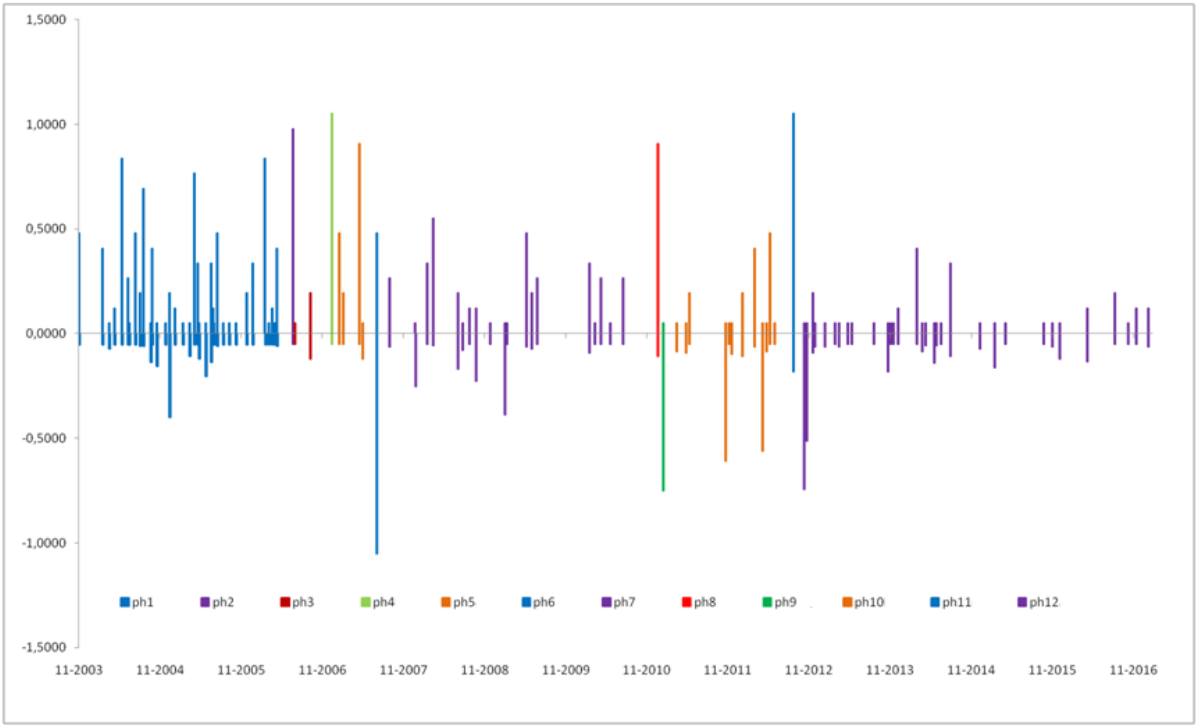


Figure 58 Phase extraction and Classification of Mediawiki with 12 phases

Opencart. Opencart is the second smallest dataset and the top solution produced by our method consists of 4 phases and is presented in Figure 59. Other fair possible phase extractions of Opencart are depicted in Figures 60 and 61 and consist of 7 and 11 phases respectively.

Opencart's first phase is a spike of *High Maintenance* and then it is followed by a long period of *Minor Activity* with some spikes of growth and maintenance nature. Then there is a phase of *High Growth* followed by a last *Minor Activity* phase.

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
1	0,63	0,29	0,08	0,00	0,33	0,54	0,13	0,00	Low	Low	Minor Activity with growth and maintenance spike(s)
2	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
3	1,00	0,00	0,00	0,00	0,00	1,00	0,00	0,00	Zero	Low	Minor Activity

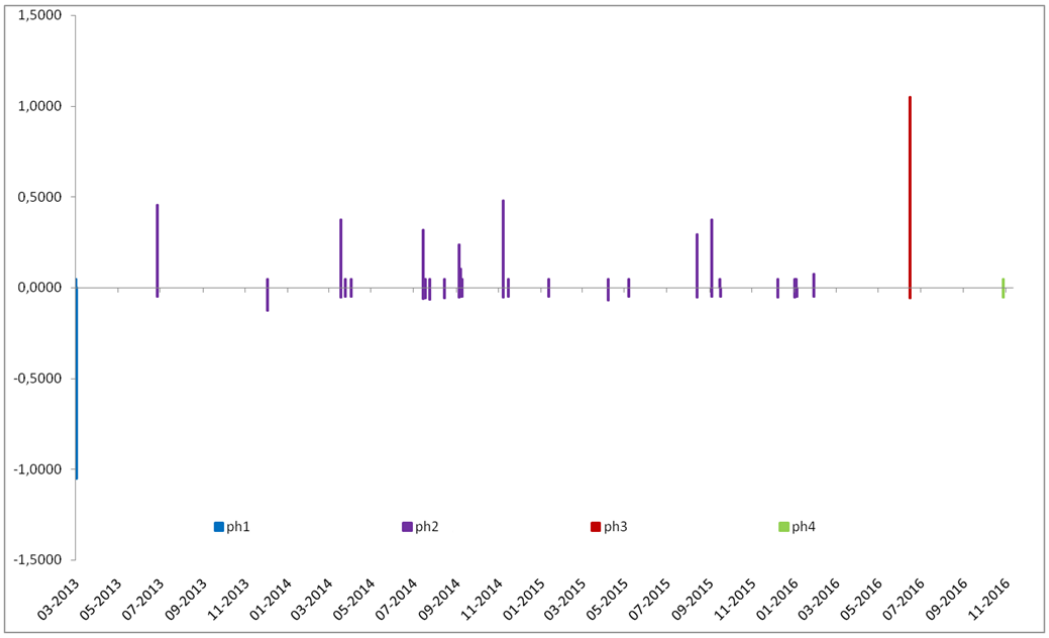


Figure 59 Phase Extraction and Classification of Opencart with 4 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
1	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00	Medium	Zero	Medium Growth
2	0,64	0,36	0,00	0,00	0,18	0,64	0,18	0,00	Low	Low	Minor Activity with maintenance spike(s)
3	0,00	0,00	1,00	0,00	0,00	1,00	0,00	0,00	Medium	Low	Medium Growth
4	0,73	0,27	0,00	0,00	0,45	0,45	0,09	0,00	Low	Low	Minor Activity with maintenance spike(s)
5	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
6	1,00	0,00	0,00	0,00	0,00	1,00	0,00	0,00	Zero	Low	Minor Activity

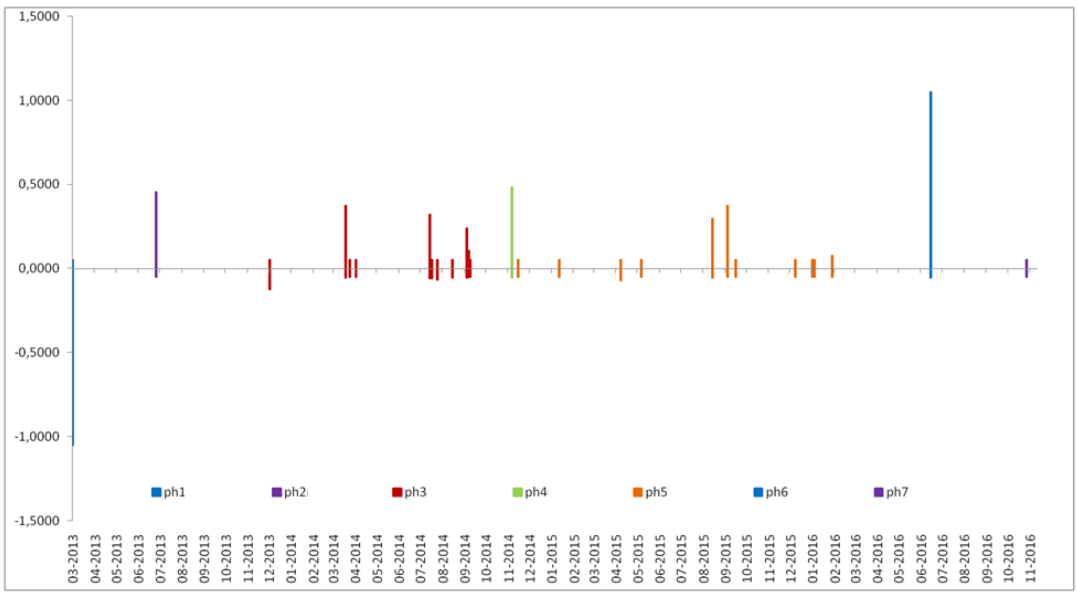


Figure 60 Phase Extraction and Classification of Opencart with 7 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	classification
0	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
1	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00	Medium	Zero	Medium Growth
2	1,00	0,00	0,00	0,00	0,00	0,00	1,00	0,00	Zero	Medium	Medium Maintenance
3	0,00	1,00	0,00	0,00	0,00	1,00	0,00	0,00	Low	Low	Minor Activity
4	0,67	0,33	0,00	0,00	0,22	0,67	0,11	0,00	Low	Low	Minor Activity with maintenance spike(s)
5	0,00	0,00	1,00	0,00	0,00	1,00	0,00	0,00	Medium	Low	Medium Growth
6	1,00	0,00	0,00	0,00	0,75	0,00	0,25	0,00	Zero	Medium	Medium Maintenance
7	0,00	1,00	0,00	0,00	0,00	1,00	0,00	0,00	Low	Low	Minor Activity
8	0,80	0,20	0,00	0,00	0,40	0,60	0,00	0,00	Zero	Low	Minor Activity
9	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
10	1,00	0,00	0,00	0,00	0,00	1,00	0,00	0,00	Zero	Low	Minor Activity

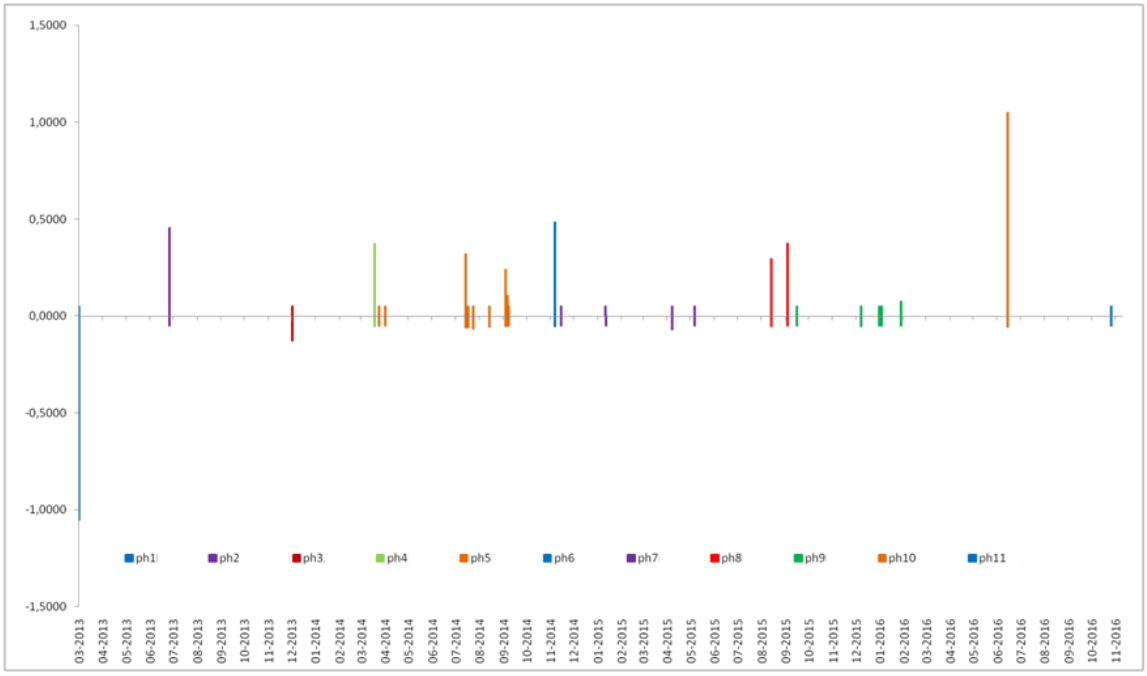


Figure 61 Phase Extraction and Classification of Opencart with 11 phases

Phpbb. Phpbb is one of the medium sized datasets in terms of releases and our top solution consists of 5 phases and is depicted in Figure 62. Figures 63 and 64 represent two alternative fair solutions of phase extraction and consist of 3 and 7 phases respectively.

Phpbb's first phase is quite long in time consists of only 2 releases and is classified as *Minor Activity*. The second phase is a spike of *High Growth* followed by a very long *Minor Activity* phase containing some spikes. The next phase is a spike of *High Maintenance* that is followed by a *Minor Activity* phase with small volume spikes.

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	1,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00	Zero	Zero	Minor Activity
1	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
2	0,61	0,25	0,11	0,03	0,17	0,64	0,17	0,03	Low	Low	Minor Activity with growth and maintenance spike(s)
3	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
4	0,80	0,00	0,20	0,00	0,00	1,00	0,00	0,00	Zero	Low	Minor Activity with growth spike(s)

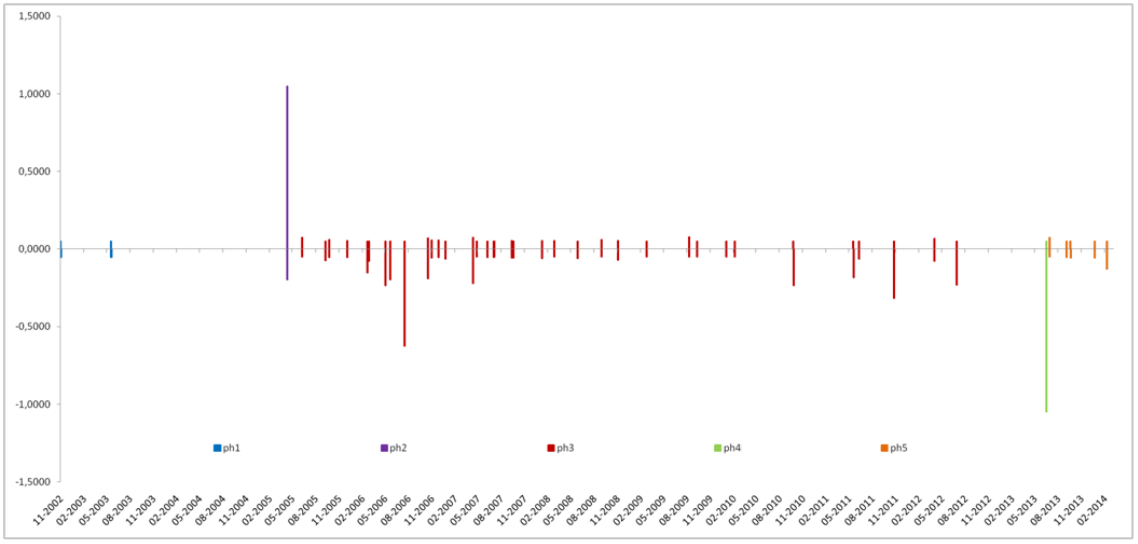


Figure 62 Phase Extraction and Classification of Phpbb with 5 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	1,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00	Zero	Zero	Minor Activity
1	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
2	0,64	0,21	0,12	0,02	0,14	0,67	0,14	0,05	Zero	Low	Minor Activity with growth and maintenance spike(s)

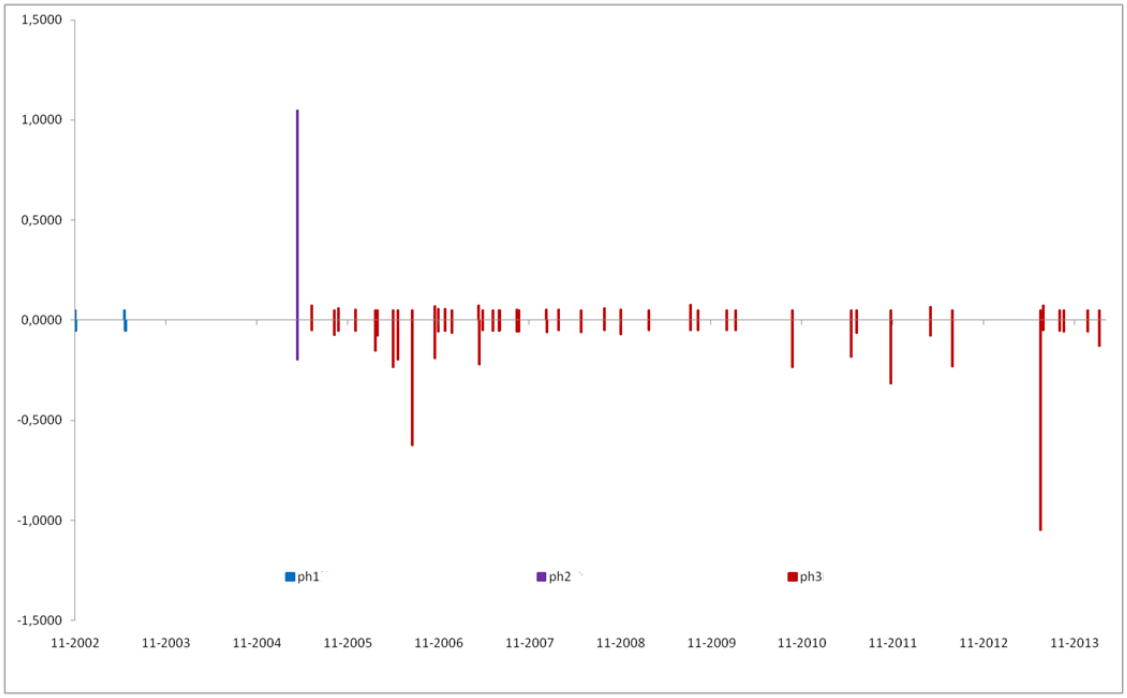


Figure 63 Phase Extraction and Classification of Phpbb with 3 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	1,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00	Zero	Zero	Minor Activity
1	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
2	0,63	0,25	0,13	0,00	0,00	0,75	0,25	0,00	Low	Medium	Medium Maintenance
3	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
4	0,59	0,26	0,11	0,04	0,22	0,63	0,15	0,00	Low	Low	Minor Activity with growth and maintenance spike(s)
5	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
6	0,80	0,00	0,20	0,00	0,00	1,00	0,00	0,00	Zero	Low	Minor Activity with growth spike(s)

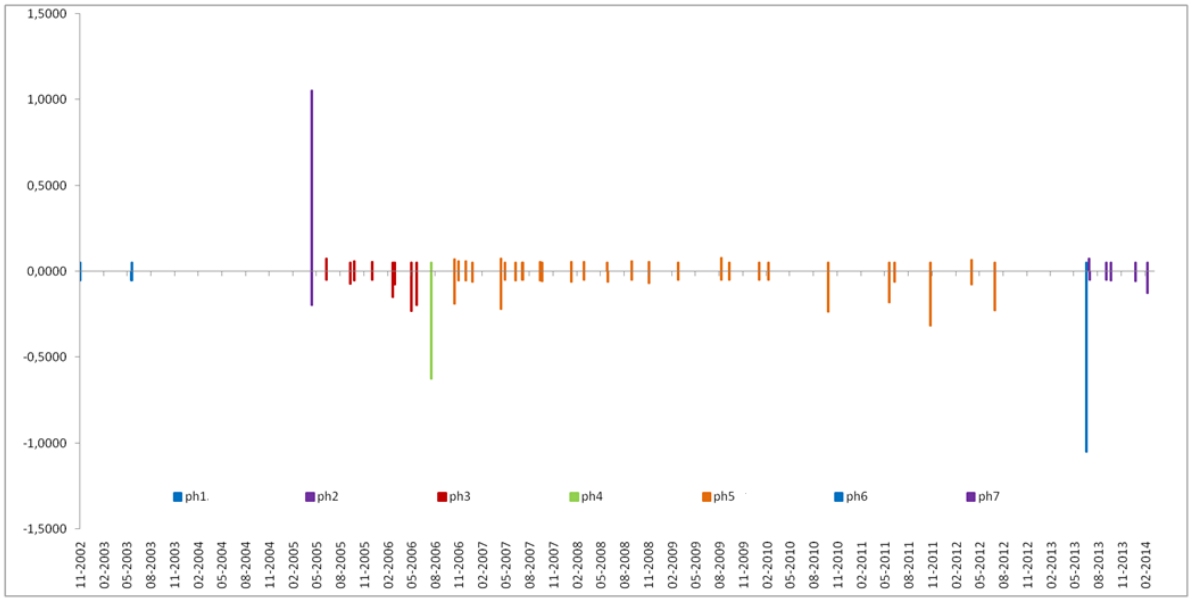


Figure 64 Phase Extraction and Classification of Phpbb with 7 phases

Typo3. Typo3 is also a medium sized dataset in terms of releases and the top algorithmically produced solution consists of 4 phases and is presented in Figure 65. In Figures 66 and 67 we can see additional fair alternatives of the phase extraction of Figure 65 consisting of 6 and 7 phases respectively.

Typo3's first phase is a very long period of *Minor Activity* with growth and maintenance spikes. The phase that follows is a spike of *High Maintenance* followed by a phase (spike) of *High Growth – Medium Maintenance*. Finally, the last phase of Typo3 is a period of calmness, thus classified as *Minor Activity*.

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,54	0,28	0,13	0,04	0,22	0,67	0,07	0,04	Low	Low	Minor Activity with growth and maintenance spike(s)
1	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
2	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
3	0,50	0,50	0,00	0,00	0,00	1,00	0,00	0,00	Low	Low	Minor Activity

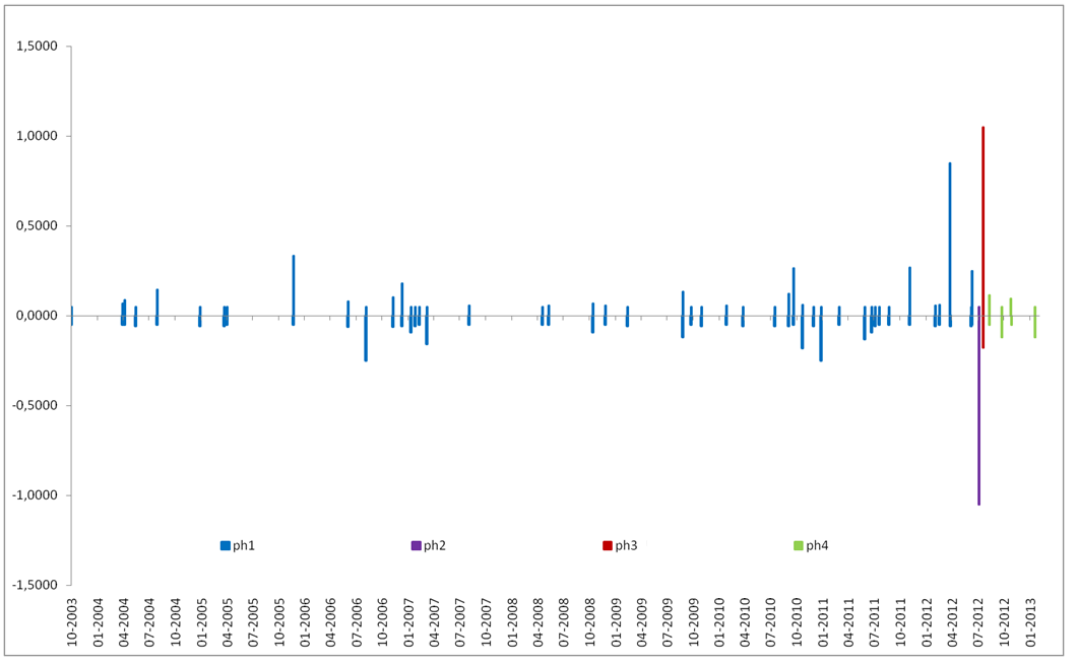


Figure 65 Phase Extraction and Classification of Typo3 with 4 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,56	0,30	0,12	0,02	0,23	0,65	0,07	0,05	Low	Low	Minor Activity with growth and maintenance spike(s)
1	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
2	0,50	0,00	0,50	0,00	0,00	1,00	0,00	0,00	Medium	Low	Medium Growth
3	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
4	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
5	0,50	0,50	0,00	0,00	0,00	1,00	0,00	0,00	Low	Low	Minor Activity

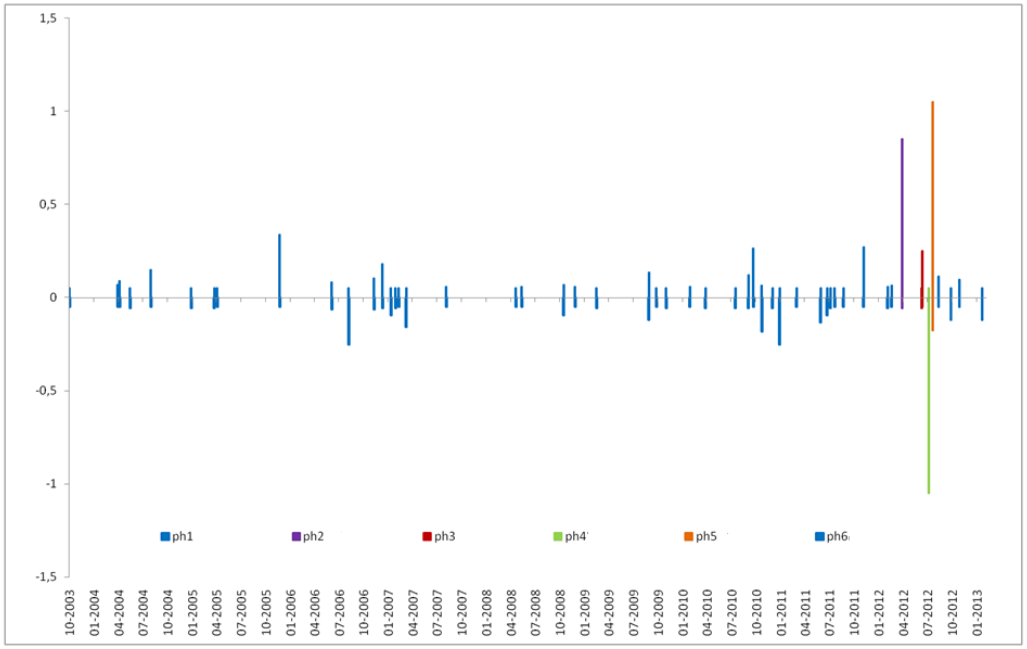


Figure 66 Phase Extraction and Classification of Typo3 with 6 phases

phaseID	zeroG	lowG	mediumG	highG	zeroM	lowM	mediumM	highM	growthWinner	maintenanceWinner	Classification
0	0,63	0,25	0,13	0,00	0,38	0,63	0,00	0,00	Low	Low	Minor Activity with growth spike(s)
1	0,54	0,31	0,11	0,03	0,20	0,66	0,09	0,06	Low	Low	Minor Activity with growth and maintenance spike(s)
2	0,00	0,00	0,00	1,00	0,00	1,00	0,00	0,00	High	Low	High Growth
3	0,50	0,00	0,50	0,00	0,00	1,00	0,00	0,00	Medium	Low	Medium Growth
4	1,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	Zero	High	High Maintenance
5	0,00	0,00	0,00	1,00	0,00	0,00	1,00	0,00	High	Medium	High Growth - Medium Maintenance
6	0,50	0,50	0,00	0,00	0,00	1,00	0,00	0,00	Low	Low	Minor Activity

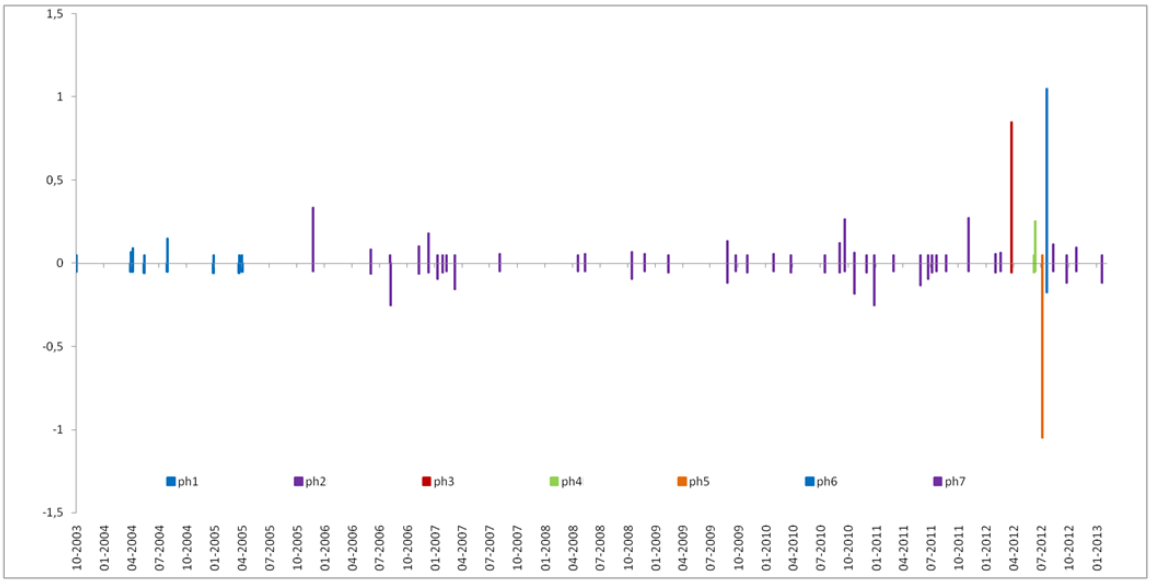


Figure 67 Phase Extraction and Classification of Typo3 with 7 phases

4.6 Conclusions

The goal of this part of the study was to introduce a new method that automatically extracts phases of a schema's life and classifies each one of them with respect to their evolution profile.

We evaluated our method via a set of measures that take into consideration both the quality of the clustering procedure based on the distance of the releases, but also the resemblance of the algorithmically produced top solutions with our golden standard. The top solutions of our *NCQ* metric (which we consider as winners) have a *Lag* value of misclassified releases that is typically less than 2. This means that our algorithm shows a fair performance with a small error rate and that for the purpose it serves, considering no heuristics were used, is a significantly decent method for automatically extracting and classifying schema releases.

Figure 68 depicts the phases of the top algorithmically produced solutions along with their classifications. We observe several similarities with Figure 2

of Section 3.2, where our golden standard was depicted and a similar amount of phases for each dataset. Furthermore, the cooling period in the end of each schema's life with a domination of *Maintenance - Minor Activity* ending phases is also present for this set of solutions. Additionally, *Maintenance* is still observed to be located anywhere in the history of a schema and finally, *Minor Activity* phases are still usually long periods of calmness.

The main difference that our algorithmically produced solution has with the golden standard is the relocation of *Growth* classified phases to slightly posterior phases than before. We address this issue to the fact that an Agglomerative implementation for the purposes of our problem, tends to isolate spikes as single phases and not merge them with their adjacent releases.

	BioSQL	Ensembl	Mediawiki	Opencart	Phpbb	Typo3
1	Minor Activity (w/ spikes)	Maintenance	Minor Activity (w/ spikes)	Maintenance (spike)	Minor Activity	Minor Activity (w/ spikes)
2	**Intense Evolution (spike)**	Growth+ Maintenance	Growth+ Maintenance	Minor Activity (w/ spikes)	**Growth**	Maintenance
3	Minor Activity	**Growth**	Minor Activity (w/ spikes)	**Growth (spike)**	Minor Activity (w/ spikes)	Growth+ Maintenance
4		Maintenance	Growth+ Maintenance	Minor Activity	Maintenance	Minor Activity
5		**Growth**	Minor Activity (w/ spikes)		Minor Activity (w/ spikes)	
6		Maintenance				
7		Minor Activity (w/ spikes)				

Figure 68 Phase Extraction and Classification of Top Algorithmically Produced Solutions

CHAPTER 5.

CONCLUSIONS 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.

5.1 Conclusions

The main goal of this thesis was to research the existence of phases in the lives of relational databases. To do so, we first studied the periods when tables, attributes and foreign keys are born, updated or evicted in a schema's life by studying the heartbeat of changes.

The most interesting finding in our study was that, with the single exception of one dataset, the history of a database schema comes in two *mega-phases*: (a) a "hot" *expansion mega-phase at the start of its life* demonstrating growth of information capacity, along with the necessary maintenance and (b) a "cooling" *housekeeping mega-phase at its middle and later life* where either maintenance actions or stillness dominate the update activity. We called this phenomenon *progressive cooling of the heartbeat*.

Some additional findings concerning the first part of this study showed that the majority of *zombie* tables tend to survive and injections and ejections of

attributes mostly happen at the start or mid of a table's life and rarely in the end.

As long as the foreign keys are concerned, we found that they come in two fashions (a) they are either treated as integral parts of the schema and get born and evicted along with their tables, mostly in scientific projects and (b) they are treated as second-class add-ons that get removed not along with their table, especially in CMSs.

The second part of this Thesis presented an automatic method for phase extraction and classification, given the history and the heartbeat of a schema, that consists of four main steps. The evaluation procedure we followed, showed that our method has a significantly decent performance in terms of misclassified releases, especially considering the fact that we did not use any heuristics.

5.2 Future work

More research can be done in the phase extraction and classification field. The clustering procedure could be implemented with another clustering algorithm, other than Agglomerative. In our implementation we used the Euclidean distance function, but one can try a different definition of distance between releases. Furthermore, the distance was calculated via our Growth – Maintenance metric values, but another direction would be to use the labels we produced with our release characterization step. Another possible modification would be to change the way we evaluate the clustering procedure by removing spikes considering them as noise.

A different direction of the one we followed during our phase extraction and classification algorithm, would be to treat the data as sets and not timeseries. In this direction one could cluster the points as sets and then label them with the cluster they belong to. This would result in a timeseries of clustering labels and one can treat this problem as a change detection problem.

BIBLIOGRAPHY

[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

[LiNe09] Dien-Yen Lin and Iulian Neamtiu. Collateral Evolution of Applications and Databases. In Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE), pages 31-40, 2009.

[Papp17] Athanasios Pappas. Supporting exploratory analytics on repository-extracted schema histories by integrating external contextual information. Master Thesis University of Ioannina, June 2017.

[PVSV12] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. Metrics for the Prediction of Evolution Impact in ETL Ecosystems: A Case Study. Journal on Data Semantics, 1(2):75-97, 2012.

[QiLS13] Dong Qiu, Bixin Li, and Zhendong Su. An Empirical Analysis of the Co-evolution of Schema and Code in Database Applications. In Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 125-135, 2013.

[Rous87] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. In Journal of Computational and Applied Mathematics, Volume 20, 1987, Pages 53-65

[Sjøb93] Dag Sjøberg. Quantifying Schema Evolution. Information and Software Technology, Vol. 35, No. 1, pp. 35-44, January 1993

[SkVZ14] Ioannis Skoulis, Panos Vassiliadis, Apostolos 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] Ioannis Skoulis, Panos Vassiliadis, Apostolos 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.

[TaSK05] Pang-Ning Tan, Michael Steinbach, Vipin Kumar. Introduction to Data Mining. Addison-Wesley 2005, ISBN 0-321-32136-7Section 8.5.2.

[VaZa17] Panos Vassiliadis, Apostolos 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.

[VKZZ17] Panos Vassiliadis, Michail-Romanos Kolozoff, Maria Zerva, Apostolos V. Zarras. Schema Evolution and Foreign Keys: Birth, Eviction, Change and Absence. 36th International Conference on Conceptual Modeling (ER 2017), pp. 106-119, Nov. 6th-9th, 2017, Valencia Spain.

[WuNe11] Shengfeng Wu and Iulian Neamtiu. Schema evolution analysis for embedded databases. In Proceedings of the 27th IEEE International Conference on Data Engineering Workshops (ICDEW), pages 151-156, 2011.

SHORT CV

Maria Zerva was born in Ioannina in 1992. She received her Bachelor's degree from the Department of Computer Science and Engineering of University of Ioannina in 2016. At the same year she became a MSc student in the same institution under the supervision of Associate Professor Panos Vassiliadis. Her main interests are software development and relational databases.