Automated Representation, Quality Assessment, Visualization, and Adaptation to Change for Data Intensive Ecosystems

A Dissertation

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

by

Petros Manousis

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
February, 2019

Advisory Committee:

- **Panos Vassiliadis**, Assoc. Professor, Department of Computer Science and engineering, University of Ioannina (Advisor)
- **Evaggelia Pitoura**, Professor, Department of Computer science and engineering, University of Ioannina
- **Apostolos Zarras**, Assoc. Professor, Department of Computer Science and engineering, University of Ioannina

Examining Committee:

- **Panos Vassiliadis**, Assoc. Professor, Department of Computer Science and engineering, University of Ioannina
- **Evaggelia Pitoura**, Professor, Department of computer Science and engineering, University of Ioannina
- **Apostolos Zarras**, Assoc. Professor, Department of Computer Science and engineering, University of Ioannina
- **Nikos Mamoulis**, Assoc. Professor, Department of Computer Science and engineering, University of Ioannina
- **Diomidis Spinellis**, Professor, Department of Management Science and Technology, Athens University of Economics and Business
- **Spiros Skiadopoulos**, Professor, Department of Informatics and Telecommunications, University of Peloponnese
- **Alkis Simitsis**, Research Director, Research Center "Athena" – Research & Innovation Information Technologies

Dedication

This book is dedicated to my friends and family.

Acknowledgements

First and foremost, I would like to thank my advisor Panos Vassiliadis for giving me the opportunity to broaden my knowledge and pursue this “trip” of research. His guidance was helpful not only in research field but in life too! Thank you very much Panos.

I would also like to thank George Papastefanatos who spent hours helping me with his insightful and kind comments during my research.

A special thank goes to Apostolos Zarras, Nikos Mamoulis and Aristidis Likas who were always there to help and encourage me.

I am also grateful to have been surrounded by colleagues such as Efthimia Kontogiannopoulou, Dimitrios Gkesoulis, Maria Zerva, Athanasios Pappas, Ioannis Skoulis, Konstantinos Semertzidis, Nikolaos Papanikos, Spiros N. Agathos and George Z. Zachos that helped in various ways when there were ups and downs during the years of my research. Thank you all.

Finally, I would like to express my gratitude to my family, for their support all of these years of my academic studies.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Glossary

Data-intensive ecosystems: are conglomerations of databases surrounded by applications that depend on them for their operation. The main characteristic of a data-intensive ecosystem is the co-existence of (a) a central repository of data, typically in the form of a relational database, and (b) a set of software applications that require access to the central database, typically via queries to its views and relations. Data-intensive ecosystems differ from the typical information systems in the sense that the management of the database profoundly takes its surrounding applications into account.

Embedded query:An embedded query is a progressively constructed query via a sequence of source code statements that is modified according to user choices (e.g. in a GUI form).

Architecture Graph: a map of the source code modules to the database schema modules of a Data Intensive Ecosystems. This map is a directed graph where the source code modules represent the queries, and the database schema modules represent the views and tables. All those modules are the nodes of the graph. The edges of the graph represent the data provision of one module to another.

Abstract

Petros Manousis, Ph.D., Department of Computer Science and Engineering, University of Ioannina, Greece, February, 2019.
Automated Representation, Quality Assessment, Visualization, and Adaptation to Change for Data Intensive Ecosystems.
Advisor: Panos Vassiliadis, Associate Professor.

Software evolution is the most demanding part of software development, since the 60% of the resources (time, money, etc.) of a company is consumed in order to evolve its software so as to meet the new requirements of its users. Database-related software also needs to evolve, but since a database is a software part that many other parts rely on it, the difficulty of evolving a database-related project increases because changing a small part of the database schema could result in failures in many different parts of the application's code. Additionally, the problem becomes even more difficult due to the scarcity of tools that help the developers evolve the database in sync with the software. In this research, we introduce principles, constructs, algorithms and metrics that help the database administrators and the software developers write easier to comprehend, adapt and maintain database related code. To achieve all that, we introduce a mapping between the software code of the projects that are related to a database, with the database schema. Initially, we locate the database queries in the software. Since the queries are embedded in a hosting language (PHP, C++, Java, etc.), we produce every possible variant of a query, based on the branches and loops of the hosting language providing abstract representations for each of the queries. Then, we come up with a language-independent representation of the queries and based on it, we translate the queries to concrete ones in more than one query languages (e.g. SQL, MongoDb). Second, to describe how well constructed a data-intensive ecosystem is, we propose a metric that describes the quality of the written query in the code of the project regarding the database schema of the project. Our experiments reveal that our metric is in sync with the Lehman's low of evolution. Whenever it is possible to propose a better way to use the database schema, we propose the introduction of views along with the rewriting of queries over them, so as to achieve better metric values and less developer effort for future schema evolution maintenance. Third, we employ a rigorous formal modelling of host code queries, tables and views in the form of a graph, called *Architecture Graph*, to facilitate diverse tasks related to the management of a Data Intensive Ecosystems. The first task facilitated by *Architecture Graph* is the identification of the impact that a schema change can have. We introduce algorithms to (a) identify and (b) adapt (via the appropriate rewritings) the impacted queries. We employ a regulation of the propagation of changes in the *Architecture Graph* via appropriate policies. Finally, the *Architecture Graph* can be exploited for visualizing the Data Intensive Ecosystems. We explore alternative ways of visualization, following [1], and we extend them via proposing a "what-if" visualization method, and we conduct a user study for all the visualizing algorithms.

Εκτεταμενη Περιληψη

Πέτρος Μανούσης, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος, 2019.
Αυτοματοποιημένη Αναπαράσταση, Αξιολόγηση Ποιότητας, Οπτικοποίηση και Προσαρμογή στις Αλλαγές για Οικοσυστήματα Βάσεων Δεδομένων.
Επιβλέπων: Παναγιώτης Βασιλειάδης, Αναπληρωτής Καθηγητής.

Η εξέλιξη των εφαρμογών είναι ένα από τα πιό απαιτητικά ζητήματα στην ανάπτυξη λογισμικού, μια και οι περισσότεροι πόροι των εταιριών (κοντά 60% των χρημάτων, του χρόνου και άλλων πηγών) δαπανούνται για την εξέλιξη του λογισμικού ώστε αυτό να συνεχίζει να ικανοποιεί της εξελισσόμενες ανάγκες των χρηστών του. Το ίδιο συμβαίνει και με τις εφαρμογές που σχετίζονται με βάσης δεδομένων, όπου η δυσκολία της εξέλιξης είναι ακόμη μεγαλύτερη διότι πολλά τμήματα λογισμικού χρησιμοποιούν της βάσης δεδομένων για να αποθηκεύουν, να ενημερώνουν και να ρωτούν για δεδομένα. Αυτοί λοιπόν οι λόγοι, δύναται να προκαλέσουν την κατάρρευση του λογισμικού σε μια ενδεχόμενη αλλαγή στο σχήμα της βάσης δεδομένων μιας εφαρμογής. Επιπλέον το γεγονός ότι δεν υπάρχουν πολλά εργαλεία που να μπορούν να κάνουν ταυτόχρονα εξέλιξη και στα δύο κομμάτια (κώδικας εφαρμογής και βάση δεδομένων) αυτών των εφαρμογών, κάνει το έργο της εξέλιξης των εφαρμογών που σχετίζονται με βάσεις δεδομένων ακόμη δυσκολότερο. Στη διατριβή αυτή μελετούμε εφαρμογές που σχετίζονται με βάσεις δεδομένων και παρουσιάζουμε μια σειρά αλγορίθμων και μετρικών που μπορούν να βοηθήσουν τους προγραμματιστές αυτού του είδους των εφαρμογών καθώς επίσης και τους διαχειριστές των βάσεων δεδομένων, ώστε να δημιουργήσουν πιο κατανοητό, πιο εύκολο να αναπτυχθεί και να συντηρηθεί κώδικα. Για να το επιτύχουμε αυτό, χρειαζόμαστε μια διασύνδεση μεταξύ της βάσης δεδομένων και των εφαρμογών που τη χρησιμοποιούν.΄Εχοντας αυτή τη διασύνδεση, μπορούμε να μετρήσουμε το πόσο καλός είναι ο κώδικας μιας εφαρμογής σε σχέση με τη βάση δεδομένων, να εξελίξουμε

τη βάση αλλά και τον κώδικα της εφαρμογής κ.ά. Το αρχικό ζήτημα που θα ασχοληθούμε είναι η εύρεση των ερωτήσεων που χρησιμοποιούν τη βάση δεδομένων στον πηγαίο κώδικα. Δοθέντος ότι οι ερωτήσεις γράφονται σε μια προγραμματιστική γλώσσα υποδοχής, που έχει διακλαδώσεις και επαναλήψεις, προτείνουμε ένα τρόπο ώστε να έχουμε κάθε διαφορετική εκδοχή της συγκεκριμένης ερώτησης, σε μια πιο αφαιρετική αναπαράσταση που στη συνέχεια μπορούμε να την εξάγουμε σε πολλές συγκεκριμένες μορφές γλωσσών ερωτήσεων. Στη συνέχεια, προτείνουμε μια μετρική που περιγράφει το πόσο καλά είναι δομημένος ο κώδικας σε σχέση με το σχήμα της βάσης δεδομένων και εξετάζουμε το κατά πόσον η μετρική αυτή εξελίσσεται με βάση τις αλλαγές που συμβαίνουν στον πηγαίο κώδικα. Επιπλέον προτείνουμε, εφόσον είναι δυνατό, αλλαγές στη δομή των ερωτήσεων και της βάσης δεδομένων ώστε να επιτυγχάνεται καλύτερη χρήση του σχήματος της βάσης από τον πηγαίο κώδικα. Έπειτα, εξετάζουμε τη δομημένη εξέλιξη εφαρμογών που σχετίζονται με βάσεις δεδομένων. Για το λόγο αυτό προτείνουμε μια δομή συσχέτισης των δύο κομματιών και χρησιμοποιούμε κανόνες για κάθε πιθανή αλλαγή που μπορεί να συμβεί στα κομμάτια αυτής της δομής συσχέτισης. Κλείνοντας, χρησιμοποιούμε αλγορίθμους οπτικοποίησης ώστε να δούμε τα αποτελέσματα των αλλαγών μας στο σύνολο αλλά και στο επιμέρους κομμάτι που αφορά τις αλλαγές ενός προγράμματος και της συσχετιζόμενης βάσης δεδομένων και διερευνούμε κατά πόσο αυτές οι μορφές οπτικοποίησης είναι αρεστές μέσω μιας μελέτης χρηστών.

Chapter 1

Introduction

According to Darwin's Origin of Species, it is not the most intellectual of the species that survives; it is not the strongest that survives; but the species that survives is the one that is able best to adapt and adjust to the changing environment in which it finds itself.[1]

In the struggle for survival, the fittest win out at the expense of their rivals because they succeed in adapting themselves best to their environment.[2]

The two previous statements –that are mistakenly attributed to Charles Darwin– describe that in real life, the species that manage to survive are the ones that evolve. What actually happens is that the faster the species evolve to handle new situations that try to eliminate their existence and / or occupy their supplies, the likelier that those species will survive. Likewise to the species evolution, software companies need to evolve their products too in order to retain, or even better grow, their position in the market.

Software *needs* to change so as to accommodate the new requirements their users have. As Meir M. Lehman introduced in 1970's, the complexity of the software that

[1] Megginson, 'Lessons from Europe for American Business', Southwestern Social Science Quarterly (1963) 44(1): 3-13, at p. 4.

[2] The Living Clocks (1971) by Ritchie R. Ward.

evolves without maintenance increases. This increased complexity makes it more difficult for the developers to add newer features in the next round. The addition of new features and their integration in the software is a re-occurring procedure. Therefore, if the software changes without any maintenance, there comes a point that no more features can be added because of the high complexity. The developers, then, need to restructure their code, or, in even worse cases, they need to rewrite the software nearly from scratch.

Likewise to the software, the database schema changes to facilitate the new features the users want. So, if it is difficult to evolve the software, even though there exist tools to help the developers, consider the difficulty on the evolution of data-intensive ecosystems, where there is nearly no tool to help!

Evolution of software and data is a fundamental aspect of their life-cycle. In the case of data management, evolution concerns changes in the contents of a database and, most importantly, in its schema. Database evolution can concern (a) changes in the operational environment of the database, (b) changes in the content of the databases as time passes by, and (c) changes in the internal structure, or *schema*, of the database. *Schema evolution*, itself, can be addressed at (a) the conceptual level, where the understanding of the problem domain and its representation via an ER schema evolves, (b) the logical level, where the main constructs of the database structure evolve (for example, relations and views in the relational area, classes in the object-oriented database area, or (XML) elements in the XML/semi-structured area), and, (c) the physical level, involving data placement and partitioning, indexing, compression, archiving etc. Likewise to the schema evolution, the software (that is represented via the queries that use a database in our search field) might also change in order to satisfy the changing user requirements, producing (a) syntactical, or (b) semantic failures. This work focuses on the evolution of database related software, providing ways to help both software developers and database administrators.

1.1 Objectives of this Thesis

The objective of this work is to help software developers write code that will be easily understandable and maintainable. To achieve this, one may have to work in conjunction with the software developers as well as the database administrators of a project, since a change in any of the involved parts could become an issue for the other part.

The main thread behind this work is the need to identify and represent how the host applications relate to the underlying databases via a rigorous “map”. To this end, we introduce such a map, in the form of a graph, which we call *Architecture Graph* that captures how the software part of a Data Intensive Ecosystems connects to the database part. We first need to solve the problem of constructing the map, given the source code of the Data Intensive Ecosystems.

In this research we propose a map that connects the software part of the data-intensive ecosystems to the database part. Having that map, we can have metrics that are related to both data and software, which metrics could then help us evaluate the quality, understandability, and maintainability of the project we are interested in. Additionally, when this is applicable, we also suggest restructures in both parts (software and database) to achieve higher software and database schema quality in the project we examine. Those suggestions fully describe what the new structure of what the code, and the database are going to be after the imminent change.

To achieve our goal we have split our research in a number of sub-problems:

- Initially, we locate the embedded queries in the software we want to examine (this could be just a small program or a number of programs, webpages, etc.) and represent them in a language-independent way.

- Then, we evaluate the quality of the source code, with respect its relationship to the schema of the database, based on metrics fulfilling a set of principles and requirements that we propose. Moreover, we provide a methodology to rewrite parts of the software code and database schema in an automatic way, when we locate parts of the software that are not “well” written.

- We introduce a fully automated method to perform “what-if” analysis for the evolution of the schema, by identifying the affected queries and automatically suggesting reparations to them, in the event of a change to the underlying schema.

- Finally, we visualize the results of our work in a way without visual clutter that will help anyone understand how the code is connected to the database schema and what / how any parts of the database or software has changed.

1.2 Contributions

Query Extraction

Regarding the query extraction problem that we examine, *what we want to achieve is to correctly identify the embedded queries within the source code of an information system and represent them in a generic, language independent way*. This is a significant aid to developers and administrators, as it can facilitate the visualization of a map of the information system, the identification of areas affected by schema evolution, code migration, and the planning of the joint maintenance of code and data. In this line of research, we provide a solution to the problem of identifying the location and semantics of embedded queries with a generic, *language-independent* method that identifies the embedded queries of a data-intensive ecosystem, *regardless of the programming style* and the host language, and represents them in a *universal*, also language-independent manner that facilitates the aforementioned maintenance, evolution and migration tasks with minimal user effort and significant effectiveness.

This is because the state of the art provides solutions that help developers identify where the database queries of their software rely, but since the queries change during runtime due to the user selections that happen during user interaction. Additionally these solutions work with only one programming language and we failed to identify any solution that works with all the query programming styles we encountered. The programming styles are: (a) the embedded SQL string statements style, (b) the ORM query style (where classes are generated and interact with only one table for inserting, updating, deleting and querying data), and (c) the object based query style (where an API is used to create a query object and the functions called upon it, perform the insertions, filters, joins etc.).

On our research on the other hand, we propose a 4 step method where we initially discard all the database unrelated code, then we create an abstract representation of the code that interacts with the query which we call Query Variants Graph (QVG). A QVG is a graph where the branch and loop statements of the host language are consumed. Then traversing the QVGs we create every possible query that might exist during runtime, which we call QVG paths. Next, we use the QVG paths to create an abstract representation of a query that we call Abstract Query Representation (AQR). The nodes of the AQR are part of en extensible pallet that now contains data transformation and filtering operators. This provides a way to represent a query of every existing query language in an independent way. Finally, we reconstruct the AQR representation to a concrete query environment such as SQL and MongoDB.

In the experiments we conducted, we examined two projects, one written in a scripting language (PHP) and another written in a procedural language (C++). On the projects we encountered a mixed style of queries, where the traditional embedded SQL string representations, and the object based (with the help of an API) were present. We were unable to locate any other query that existed in the project, besides the ones that our methods located, and we managed to reconstruct at least the 80% of those queries in their original form.

A Metric to Assess the Coupling of Software to the Database

Moving on, we propose a metric to assess the Coupling of Software to the Database. Software metrics evaluate how well-designed, understandable and maintainable a software system is. Regarding the software projects with access to data sources, the state of the art demonstrates an observable gap in providing metrics to describe the quality of the connection between the source code of the software, the data sources queries, and the data sources schema.

To address this shortcoming, in this line of research, we introduce, in a principled manner, the *fundamental ideas*, *properties* and *constraints* for objectively evaluating a *well-designed Data Intensive Information Systemsproject*, and we propose a *data-to-software coupling metric* based on this foundation. We use the proposed framework and metric over a set of projects to assess their applicability and effectiveness. Finally, we propose software and database schema changes, when applicable, to achieve higher metric values.

We have applied our metric to the evolution of a project and we observe that our metric is in sync with the Lehman's Lows of evolution, since when new features are inserted in the project, our metrics value decreases, while when maintenance steps occur in the project, our metrics value increases. Next, since the majority of the projects we tried to examine had no changes in their database schema, or only small ones, we propose an algorithm that suggests schema changes so as to increase the metric value, producing an easier to comprehend and maintain software. Using this algorithm, we additionally increased the developers gain, since he would have less parts of code to examine in order to adapt his code for a new feature. The minimum developer gain was 14% while the average was above 50%, meaning that the developer would need to check only half of the files / queries.

Regulation of Schema Evolution with Policies

Having identified and extracted the queries of a data-intensive ecosystem and their location in the source code, we can finally present the “skeleton” of the data-intensive ecosystem as a graph (called *Architecture Graph*), uniformly covering relations, views and queries as nodes and their internal structure and interdependencies as the edges of the graph. The *Architecture Graph* is useful for many tasks and the first that we address is the “what-if” analysis for the evolution of the database part of a data-intensive ecosystem, in order to identify all the parts of the ecosystem that are affected by a potential change in the database schema. Additionally we want to see how will the ecosystem look like once the change has been performed, while, at the same time, retaining the ability to regulate the flow of events.

To do so, we provide a simple language to annotate the modules of the graph with policies for their response to evolutionary events in order to regulate the flow of events and their impact by (i) vetoing (“blocking”) the change in parts that the developers want to retain unaffected and (ii) allowing (“propagating”) the change in parts that we need to adapt to the new schema.

Our method for the automatic adaptation of ecosystems is based on three algorithms that automatically (i) assess the impact of a change, (ii) compute the need of different variants of an ecosystem’s components, depending on policy conflicts, and (iii) rewrite the modules to adapt to the change. We theoretically prove the coverage of the language, as well as the termination, consistency and confluence of our algorithms, and experimentally verify our methods‘ effectiveness and efficiency.

Data-Intensive Ecosystem Visualization

Finally, since we used a graph to depict the data-intensive ecosystems we examined, we need a proper way to visually demonstrate the structure of the data-intensive ecosystems. In particular, in the context of this thesis, we have explored the evolution of projects based on visual representation methods introduced in [1], and we extended this work with a “what-if” analysis visualization of the affected nodes with reduced visual clutter. Finally, we evaluated of all the methods presented in [1] and our “what-if” visualization with a user study.

1.3 Structure

The text is organized in 6 chapters. In Chapter 2 we present the current state of the art on the evolution of database related software for each one of our sub-problems described in Section 1.1.

Then, in Chapter 3 we focus on our findings on query extraction from the source code of the data intensive ecosystems, where we present the different forms of queries that we found on the projects we examined, the universal way we proposed in order to represent every query and some experiments that describe that our method can be used in order to migrate a software query from one querying language to another (e.g. from SQL to MongoDb).

Following, in Chapter 4 we present a metric for identifying whether a database related software code is well defined over a database schema or not, as well as an algorithm that using query rewrites through views, helps us achieve higher metric measurement values.

After that, in Chapter 5 we present the benefits of our methodology (presented in Chapter 4) on the evolution of a real case software and we describe how the software can smoothly operate despite the database schema changes that occur, using policies on evolution events.

Finally, in Chapter 6 we present a visualization approach of database related projects that depicts the parts of the data intensive ecosystem in a way that any project related person (database administrator, software developer, software architect) or not, will easily understand how the code is constructed, and when any change occurs which parts of the ecosystem are affected.

Chapter 2

Related work

2.1 Introduction

In this chapter we discuss the state of the art works that are related to the evolution of the source code and the database schema. In the first section we discuss the about the evolution of database-related software in general, and following, we discuss the state of the art for the query extraction problem, followed by works that are related to metrics of software and databases. Then, we discuss works that are related to rewrites of source code or database schema and finally, we explore works that are related to the visualization of graphs, that we use in our model, so as to produce easier to comprehend representations.

2.2 Database Evolution

Evolution of software and data is a fundamental aspect of their lifecycle. In the case of data management, evolution concerns changes in the contents of a database and, most importantly, in its schema. Database evolution can concern (a) changes in the operational environment of the database, (b) changes in the content of the databases as time passes by, and (c) changes in the internal structure, or *schema*, of the database. *Schema evolution*, itself, can be addressed at (a) the conceptual level, where the understanding of the problem domain and its representation via an ER schema evolves, (b) the logical level, where the main constructs of the database structure evolve (for example, relations and views in the relational area, classes in the object-oriented database area, or (XML) elements in the XML/semi-structured area), and, (c) the physical level, involving data placement and partitioning, indexing, compression, archiving etc.

In this section, we will focus on the evolution of the logical schema of relational data and also extend our survey to the special case of data warehouse evolution. For the rest, we refer the interested reader to the following very interesting surveys. First, it is worth visiting a survey by Roddick [2], which appeared 20 years ago and summarizes the state of the art of the time in the areas of schema versioning and evolution, with emphasis to the modeling, architectural and query language issues related to the support of evolving schemata in database systems. Second, 16 years later, a comprehensive survey by Hartung, Terwilliger and Rahm [3] appeared, in which the authors classify the related tools and research efforts in the following subareas: (a) the management of the evolution of relational database schemata, (b) the evolution of collections of XML documents, and (c) the evolution of ontologies. In the web site `http://dbs.uni-leipzig.de/en/publications` one may also find a comprehensive list of publications in the broader area of schema and data evolution.

2.2.1 Empirical Studies on Database Evolution

In this section, we survey empirical studies in the area of database evolution. These studies monitor the history of changes and report on statistical properties and recurring phenomena. In our coverage we will follow a chronological order, which also allows us to put the studies in the context of their time.

Statistical profiling of database evolution via real world studies

Studies during the 90's. The first account of a sizable empirical study, by Sjoberg [4], discusses the evolution of the database schema of a health management system over a period of 18 months, monitored by a tool specifically constructed for this purpose. A single database schema was examined, and, interestingly, the monitored system was accompanied by a metadata dictionary that allowed to trace how the queries of the applications surrounding the database relate to the tables and attributes of the evolving database. Specific numbers for the evolution of the system, during this period of 18 months, include:

- There was a 139% increase of the number of tables and a 274% increase of the number of attributes (including affected attributes due to table evolution), too.
- All (100%) the tables were affected by the evolution process.
- Additions were more than deletions, by an 28% tables and a 42% for attributes.
- An insignificant percentage of alterations involved renaming of relations or merge/split of tables.
- Changes in the type of fields (i.e., data type, not null, unique constraints) proved to be equal to additions (31 both) and somehow less than deletions (48) for a period of 12 months, during which this kind of changes were studied.
- On average, each relation addition resulted in 19 changes in the code of the application software. At the same time, a relation deletion produced 59.5 changes in the application code. The respective numbers for attributes were 2 changes for attribute additions and 3.25 changes for attribute deletions, respectively.
- The evolution process was characterized by an inflating period (during construction) where almost all changes were additions, and a subsequent period where additions and deletions where balanced.

Revival in late 00's. In terms of empirical studies, and to the best of our knowledge, no developments took place for the next 15 years. This can be easily attributed to the fact that the research community would find it very hard to obtain access to monitor database schemata for an in-depth and long study. The proliferation of free and open-source software changed this situation. So, in the last few years, there are more empirical studies in the area that report on how schemata of databases related to open source software have evolved.

The first of these studies came fifteen years later after the study of Sjoberg. The authors of [5] made an analysis on the database back-end of MediaWiki, the software that powers Wikipedia. The study conducted over the versions of four years, and came with several important findings. The study reports an increase of 100% in the number of tables and a 142% in the number of attributes. Furthermore, 41.5% of the attributes of the original database were removed from the database schema, and 25.1% of the attributes were renamed respectively. The major reasons for these alterations were (a) the improvement of performance, which in many cases induces partitioning of existing tables, creation of materialized views, etc., (b) the addition of new features which induces the enrichment of the data model with new entities, and (c) the growing need for preservation of database content history. A very interesting observation is that around 45% of changes do not affect the information capacity of the schema, but they are rather index adjustments, documentation, etc. A statistical study of change breakdown revealed that attribute addition is the most common alteration, with 39% of changes, attribute deletion follows with 26%, attribute rename was up to the 16% and table creation involved a 9% of the entire set of recorded changes. The rest of the percentages were insignificant.

Special mention should be made to this line of research [6], as the people involved in this line of research should be credited for providing a large collection of links[1] for open source projects that include database support. Also, it is worth mentioning here that the effort is related to PRISM (later re-engineered to PRISM++ [7]), a change management tool, that provides a language of Schema Modification Operations (SMO) (that model the creation, renaming and deletion of tables and attributes, and their merging and partitioning) to express schema changes (see Section 2.2.3 for details).

Shortly after, two studies from the Univ. of Riverside appear. In [8], Lin and Neamtiu study two aspects of database evolution and their effect to surrounding applications. The first part of the study concerns the impact that schema evolution has on the surrounding applications. The authors work with two cases, specifically the evolution of Mozilla, between 2005 and 2009 and the evolution of the Monotone version control system between 2003 and 2009, both of which use a database to store necessary information for their correct operation. The authors document and exem-

[1] http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Benchmark_Extension

plify how the developers of the two systems address the issue of schema evolution between different versions of their products. The authors also discuss the impact of erroneous database evolution, even though there exists software that is responsible for the migration of the system's modules to the new database schema. One very interesting finding is that although the applications can include a check on whether the database schema is synchronized to the appropriate version of the application code, this check is not omnipresent; thus, there exist cases where the application can operate on a different schema than the one of the underlying database, resulting in crashes or data loss. At the same time, the authors have measured the breakdown of changes during the period that they have studied. The second part of the study concerns DBMS evolution (attention: *DBMS*, not database) from the viewpoint of file storage. The authors study SQLite, MySQL and Postgres on how different releases come with different file formats and how usable old formats can be under a new release of the DBMS. Also, the authors discuss how the migration of stored databases should be performed whenever the DBMS is upgraded, due to the non-compatibility of the file formats of the different releases.

In a similar vein, in [9], Wu and Neamtiu considered 4 case studies of embedded databases (i.e., databases tightly coupled with corresponding applications that rely on them) and studied the different kinds of changes that occurred in these cases. Specifically, the authors study the evolution of Firefox between 2004 and 2008, Monotone (a version management system) between 2003 and 2010, BiblioteQ (a catalog management suite) between 2008 and 2010 and Vienna (an RSS newsreader) between 2005 and 2010. Comparing their results to previous works, the authors see the same percentages concerning the expansion of the database, but a larger number of table and column deletions. This is attributed to the nature of the databases, as the databases that are studied by Wu and Neamtiu are embedded within applications, rather than largely used databases as in the case of the previous studies. Moreover, the authors performed a respective frequency and timing analysis, which showed that the database schemata tend to stabilize over time, as the evolution activity calms down over time. There is more change activity for the schemata at the beginning of their history, whereas the schemata seem to converge to a relatively fixed structure at later versions.

A large scale study in 2013. In [10], Qiu, Li and Su report on their study of the evolution of 10 databases, supporting open source projects. The authors collected the source files of the applications via their SVN repositories and isolated the changes to the logical schema of each database (i.e., they ignored changes involving comments, syntax correction, DBMS-related changes, and several others). The remaining changes are characterized by the authors as valid DB revisions. The authors report that they have avoided the automatic extraction of changes, as the automatic extraction misses changes like table split or merge, or renaming and have performed manual checks for all the valid DB revisions for all the datasets. The study covers 24 types of change including the additions and deletions of tables, attributes, views, keys, foreign keys, triggers, indexes, stored procedures, default value and not null constraints, as well as the renaming of tables, attributes and the change of data types and default values. We summarize the main findings of the study in four categories.

Temporal and Locality Focus. Change is focused both (a) with respect to time and (b) with respect to the tables that change. Concerning timing, a very important finding is that 7 out of 10 databases reached 60% of their schema size within 20% of their early lifetime. *Change is frequent in the early stages of the databases, with inflationary characteristics; then, the schema evolution process calms down.* Schema changes are also focused with respect to the tables that change: 40% of tables do not undergo any change at all, and 60%-90% of changes pertain to 20% of the tables (in other words, 80% of the tables live quiet lives). The most frequently modified tables attract 80% of the changes.

Change breakdown. The breakdown of changes revealed the following catholic patterns: (a) insertions are more than updates which are more than deletions and (b) table additions, column additions and data type changes are the most frequent types of change.

Schema and Application Co-evolution. To assess how applications and databases co-evolve, the authors have randomly sampled 10% of the valid database revisions and manually analyzed co-evolution. The most important findings of the study are as follows:

- First, the authors characterized the co-change of applications in four categories and assessed the breakdown of changes per category. In 16.22% of occasions, the code change was in a previous/subsequent version than the one where the database schema change occurred; 50.67% of application adaptation changes took place in the same revision with the database change, 21.62% of database changes were not followed by code adaptation and 11.49% of code changes were unrelated to the database evolution.

- A second result says that each atomic change at the schema level is estimated to result in 10 – 100 lines of application code been updated. At the same time, a valid database revision results in 100 – 1000 lines of application code being updated.

A final note: Early in the analysis of results, the authors claim that change is frequent in schema evolution of the studied datasets. Although we do not dispute the numbers of the study, we disagree with this interpretation: change caries a lot between different cases (e.g., coppermine comes with 8.3 changes and 14.2 atomic changes per year contrasted to 65.5 changes and 299.3 atomic changes per year at Prestashop). We would argue that change can be arbitrary depending on the case; in fact, each database seems to present its own change profile.

Recent advances in uncovering patterns in the evolution of databases

A recent line of research that includes [11, 12, 13], reveals patterns and regularities in the evolution of database schemata. At a glance, all these efforts analyze the evolution of the database schemata of 8 open source case studies. For each case study, the authors identified the changes that have been performed in subsequent schema versions and re-constructed the overall evolution history of the schema, based on Hecate, an automated change tracking tool developed by the authors for this purpose. The number of versions that have been considered for the different schemata ranged from 84 to 528, giving a quite rich data set for further analysis. Then, in [11] the authors perform a macroscopic study on the evolution of database schemata. Specifically, in this study the authors detect patterns and regularities that concern the way that the database schema grows over time, the complexity of the schema, the maintenance actions that take place and so on. To detect these patterns they resort to the properties that are described in Lehnman's laws of software evolution [14]. In [12], extend their baseline work in [11] with further results and findings revealed by the study, as long as detailed discussions concerning the relevance of the Lehman's laws in the case of databases, and the metrics that have been employed. On the other hand, in [13] the authors perform a microscopic study that delves into the details of the life of tables, including the tables' birth, death, and the updates that occur in between. This study reveals patterns, regularities and relations concerning the aforementioned aspects.

The life of a database schema. In the early 70's, Lehman and his colleagues initiated their study on the evolution of software systems [15] and continued to refine and extend it for more than 40 years [14]. Lehman's laws introduce the properties that govern the evolution of *E-type systems*, i.e., software systems that solve a problem, or address an application in the real world [14]. For a detailed historical survey of the evolution of Lehman's laws the interested reader can refer to [16]. The essence of Lehman's laws is that *the evolution of an E-type system is a controlled process that follows the behavior of a feedback-based mechanism*. In particular, the evolution is driven by *positive feedback* that reflects the need to adapt to the changing environment, by *adding functionalities* to the evolving system. The growth of the system is constrained by *negative feedback* that reflects the need to perform *maintenance activities*, so as to prevent the deterioration of the system's quality.

In more detail, as discussed in [11, 12] the laws can be organized in three groups that concern different aspects of the overall software evolution process. The first group of laws discusses the existence of the feedback mechanism that constrains the uncontrolled evolution of software. The second group focuses on the properties of the growth part of the system, i.e., the part of the evolution mechanism that accounts for positive feedback. Finally, the third group of laws discusses the properties of perfective maintenance that constrains the uncontrolled growth, i.e., the part of the evolution mechanism that accounts for negative feedback. The major patterns and regularities revealed in [11, 12] from the investigation of each group of laws are summarized below:

- *Feedback mechanism for schema evolution*: Overall, the authors found that schema evolution demonstrates the behavior of a stable, feedback-regulated system, as the need for expanding its information capacity to address user needs is controlled via perfective maintenance that retains quality; this antagonism restrains unordered expansion and brings stability. Positive feedback is manifested as expansion of the number of relations and attributes over time. At the same time, there is negative feedback too, manifested as house-cleaning of the schema for redundant attributes or restructuring to enhance schema quality. In [11, 12] the authors further observed that the inverse square models [17] for the prediction of size expansion hold for all the schemata that have been studied.

- *Growth of schema size due to positive feedback*: The size of the schema expands over time, albeit with versions of perfective maintenance due to the negative feedback. The expansion is mainly characterized by three patterns/phases, (i) abrupt change (positive and negative), (ii) smooth growth, and, (iii) calmness (meaning large periods of no change, or very small changes). The schema's growth mainly occurs with spikes oscillating between zero and non-zero values. Also, the changes are typically small, following a Zipfian distribution of occurrences, with high frequencies in deltas that involved small values of change, close to zero.

- *Schema maintenance due to negative feedback*: As stated in [12] the overall view of the authors is that due to the criticality of the database layer in the overall information system, maintenance is done with care. This is mainly reflected by the decrease of the schema size as well as the decrease in the activity rate and growth with age. Moreover, the authors observed that age results in a reduction of the complexity to the database schema. The interpretation of this observation is that perfective maintenance seems to do a really good job and complexity drops with age. Also, they authors point out that in the case of schema evolution, activity is typically less frequent with age.

The life of a table - microscopic viewpoint. In [13], the authors investigated in detail the relations between table schema size, duration and updates. The main findings of this study are summarized below:

- From a general perspective, early stages of the database life are more "active" in terms of births, deaths and updates, whereas, later, growth is still there, but deletions and updates become more concentrated and focused.

- The life and death of tables is governed by the *Gamma* pattern, which says that large-schema tables typically survive. Moreover, short-sized tables (with less than 10 attributes) are characterized by short durations. The deletions of these "narrow" tables typically take place early in the lifetime of the project either due to deletion or due to renaming (which is equivalent from the point of view of the applications: they crash in both cases).

- Concerning the amount of updates, most tables live quiet lives with few updates. The main reason is the dependency magnet phenomenon, i.e., table updates induce large impact on the surrounding dependent software.

- The relation between table duration and amount of updates is governed by the *inverse Gamma* pattern, which states that updates are not proportional to longevity, but rather, few top-changer tables attract most of the updates.
  - Top-changer tables live long, frequently they are created in the first version of the database and they can have large number of updates (both in absolute terms and as a normalized measure over their duration).
  - Interestingly top-changer tables, they are not necessarily the larger ones, but typically medium sized.

2.2.2 State of Practice

In this section, we discuss how the commercial database management systems handle schema changes. The systems that we survey are: (a) Oracle, (b) DB2 of IBM, and, (c) SQL Server of Microsoft. Another part of this research is dedicated to the open sourced or academic tools that are dealing with the schema changes. Some of those tools are: (a) Django, (b) South, and, (c) Hecate.

Commercial tools

Oracle - Change Management Pack (CMP). Oracle Change Management Pack ([18]) is part of Oracle Enterprise Manager. CMP enables the management and deployment of schema changes from development to production environments, as well as the identification of unplanned schema changes that potentially cause application errors.

CMP features the following concepts:

- Change plans: A change plan is an object that serves as a container for change requests.
- Baselines: A baseline is a group of database object definitions captured by the Create Baseline application at a particular point in time.
- Comparisons: A comparison identifies the differences found by the Oracle Change Management Pack in two sets of database object definitions that you have specified in the Compare Database Objects application.

The *Create Baseline* application enables users in creating database schema descriptions in a CMP format or plain SQL DDL files. These descriptions are used to compare, or make changes to other schemata.

The *Compare Database Objects* application allows DBA users to compare different "database" versions. This way, in case of an application error produced by a non-tested schema change applied in the database, the DBA can produce all changes a-posterior and find the cause of the application failure.

The *Synchronization Wizard* of CMP supports the user in modifying an item *target* to match another item *source*. The *Synchronization Wizard* needs a comparison of the *target* and *source* items, so it works after the *Compare Database Objects* application. The *Synchronization Wizard* orders the "transformation" steps, in order to produce the *target* item. This is, for example, to make sure that the foreign keys will be applied after the primary keys. Besides that, the *Synchronization Wizard* can delete items. This happens, when there is no *source* item. Moreover, if there is no *target* item, the *Synchronization Wizard* initially creates and then synchronizes a *new target* item with the *source* one. Finally, using the *Synchronization Wizard*, the user may keep or undo the changes made to a *target* item.

Another module that works similar to the *Synchronization Wizard* is the *DB Propagate* application of CMP, which allows the user to select one or more object definitions and reproduce them in one or or more *target* schemata.

Two other applications of CMP are: *DB Quick Change*, and, *DB Alter*. The *DB Quick Change* application helps the user in making *one* change to a *single* database item. The *DB Alter* application helps the user in making one or more changes to one, or more database items (in comparison to the *Synchronization Wizard*, here there is no need of any preceding comparison).

Finally, the *Plan Editor* of CMP lets the user perform a single change plan on one or more databases, that he may keep or undo. The *Plan Editor* can perform a wider variety of changes, compared to those that *Synchronization Wizard*, *DB Alter*, *DB Quick Change*, and *DB Propagate* can perform. The *Plan Editor* allows the creation of a change plan that serves as a container for change requests (directives, scoped directives, exemplars, and modified exemplars), generates scripts for those change requests and executes them on one or more databases.

IBM - DB2. IBM DB2 provides a mechanism that checks the type of the schema changes [19] that the users want to perform in system-period temporal tables. A system-period temporal table is a table that maintains historical versions of its rows. A system-period temporal table uses columns that capture the begin and end times when the data in a row is valid and preserve historical versions of each table row whenever updates or deletes occur. In this way, queries have access to both current data, i.e., data with a valid current time, as well data from the past. Finally, DB2 offers the DB2 Object Comparison Tool [20]. It is used for identifying structural differences between two or more DB2 catalogs, DDL, or version files (even between objects with different names). Moreover, it is able to generate a list of changes in order to transform the *target* comparator into a new schema, described by the *source* comparator. Finally, it is capable to undo changes that were performed and committed in a version file, so as to restore it to a given previous version.

Temporal tables prohibit changes that result in loss of data. These changes can be produced by commands like DROP COLUMN, ADD COLUMN, and, ALTER COLUMN. All the changes, applicable to temporal tables, can be propagated back to the history of the schema, with only two exceptions, the renaming of a table and the renaming of an index.

Microsoft - SQL Server. Change management support for Microsoft SQL Server comes with the SQL Server Management Studio [21] (SSMS). SSMS allows the user to browse, select, and manage any of the database objects (e.g., create a new database, alter an existing database schema, etc.) as well as visually examine and analyze query plans and optimize the database performance. SSMS provides data import

export capabilities, as well as data generation features, so that users can perform validation tests on queries. Regarding the evolution point of view, it is capable of comparing two different database instances and returning their structural differences. The tool may also provide information on DDL operations that occurred, through the reports of schema changes. An example of such a report from `https://www.mssqltips.com/sqlservertip/4057/capture-sql-server-schema-changes-using-the-default-trace/` is displayed in Table 2.1.

Another set of tools that Microsoft offers for the validation of SQL code is the SQL Server Data Tools [22] (SSDT). SSDT follows a project-based approach for the database schema and SQL source code that is embedded in the applications. A developer can use SSDT to locally check and debug SQL code (by using breakpoints in his SQL code).

database name	start time	login name	user name	application name	ddl operation	object
msdb	2015-08-27 14:08:40.460	sa	sa	SSMS - Query	CREATE	DDL_History
TestDB	2015-08-26 11:32:19.703	sa	sa	SSMS	ALTER	SampleData

Table 2.1: SSMS Report

Open source or academic tools

Django. Django [23] uses the model-view-controller idea for the database schema manipulation. In Django, the user defines classes that represent the columns of an RDBMS's table. The classes are mapped to relational tables and created in the database.

Regarding evolution, Django uses an automatic way to identify which columns were added or deleted from the tables between two versions of code and migrate these changes to the database schema. Django identifies the changes in the attributes of a class and then produces the appropriate SQL code that performs the changes to the underlying database schema.

South. South [24] is a tool operating on top of Django, identifying the changes in the Django's models and providing automatic migrations to match the changes. South supports five database backends (PostgreSQL, MySQL, SQLite, Microsoft SQL Server, and, Oracle), while Django officially supports four (PostgreSQL, MySQL, SQLite, and, Oracle). South also supports another five backends(SAP SQL Anywhere, IBM DB2, Microsoft SQL Server, Firebird, and, ODBC) through unofficial third party database connectors.

In South, one can express dependencies of table versions so as to have the correct execution order of migration steps and void inconsistencies. For example, in a case where a foreign key references a column that is not yet a key, this kind of problem can be identified and avoided.

The Autodetector part of South can extend the migrations that Django offers. Specifically, South can automatically identify the following schema modifications: model creation and deletion (create/drop a table), field changes (type change of columns) and unique changes, while Django can only identify the addition or deletion of columns.

Hecate. Hecate [25] is a tool that parses the DDL files of a project and compares the database schemata between versions. Hecate also exports the transitions between two versions, describing the additions and deletions that occurred between the versions (renames are treated as deletions followed by additions). Hecate also provides measures such as size and growth of the schema versions.

Hecataeus. Hecataeus [26] is a what-if analysis tool that facilitates the visualization and impact analysis of data-intensive software ecosystems. As these ecosystems include software modules that encompass queries accessing an underlying database, the tool represents the database schema along with its dependent views and queries as a uniform directed graph. The tool visualizes the entire ecosystem in a single representation and allows zooming in and out its parts. Most related to the topic of this survey, the tool enables the user to create hypothetical evolution events and examine their impact over the overall graph. Hecataeus does not simply flood the event over the underlying graph; it also allows users as to define "veto" rules that block the further propagation of an evolutionary event (e.g., because a developer is adamant in keeping the exact structure of a table employed by one of her applications). Hecataeus also rewrites the graph, after the application of the event so that both the syntactical and the semantic correctness of the affected queries and views are retained.

2.2.3 Techniques for managing database and view evolution

Impact Assessment of Database Evolution

In this section, we discuss the impact of changes in a database schema to the applications that are related to that schema. Given a set of scripts, the methods proposed in this part of the literature identify how database and software modules are affected by changes that occur at the database level. Techniques for query rewriting are also discussed. Closely to this topic is the topic of view adaptation: how must the definition (and the extent, in case of materialization) of a view adapt whenever the schema of its underlying relations changes?

Early Attempts towards facilitating Impact Assessment. Maule, Emmerich and Rosenblum [27] propose a technique for the identification of the impact of relational database schema changes upon object-oriented applications. In order to avoid a high computational cost, the proposed technique uses slicing, so as to reduce the size of the program that is needed to be analyzed. At a first step, the authors use a prototype slicing implementation that helps them identify the database *queries* of the program. Then, with a data-flow analysis algorithm, the authors estimate the possible runtime values for the parameters of the query. Finally, the authors use an impact assessment tool, Crocopat, coming with a reasoning language (RML) to describe the impacts of a potential change to the stored data of the previous step. Depending on the type of change, a different RML program is run, and this eventually isolates the *lines of code of the program* that are related to the queries affected by the change.

The authors evaluated their approach on a C# CMS project of 127000 lines of code, and a primary database schema of up to 101 tables, with 615 columns and 568 stored procedures. The experiments showed that the method needed about 2 minutes for each execution, where they found that there were no false negatives. On the other hand, there were false positives in the results, meaning that the tool was able to find all the lines of code that were affected, leaving none out, but also falsely reported that some lines of code would be affected, whilst this was not really happening.

Architecture Graphs. Papastefanatos et al. [28, 29, 30] introduced the idea of dealing with *both* the database and the application code in uniform way. The results of this line of research are grouped in the areas of (a) modeling, (b) change impact analysis, and (c) metrics for data intensive ecosystems (data intensive ecosystems are conglomerations of data repositories and the applications that depend on them for their operations). This line of work has been facilitated by the Hecataeus tool (see [28], [31]).

Concerning the modeling and the impact analysis parts, in [28], the authors proposed the use of the *Architecture Graph* for the modeling of data intensive ecosystems. The *Architecture Graph* is a directed graph where the nodes represent the entities of the ecosystem (relations, attributes, conditions, queries, views, group by clauses, etc), while the edges represent the relationships of these entities (schema relationships, operand relationships, map-select relationships, from relationships, where relationships, group by relationships, etc). In the same paper, the authors proposed an algorithm for the propagation of the changes of one entity to other related entities, using a *status* indicator of whether the imminent change is accepted, blocked or if the user of the tool should be asked.

In [29], the authors proposed an extension for the SQL query language, that introduced policies for the changes in the database schema. The users could define in the declaration of their database schema whether a change should be accepted, blocked or if the user should be prompted. In this work, the policies were defined over: (a) the database schema universally, (b) the *high level modules* (relations, views and queries) of the database schema, and, (c) the remaining entities of the database, such as attributes, constraints and conditions.

Regarding the metrics part, a first attempt to the problem was made by Papastefanatos et al, on ways to predict the maintenance effort and the assessment of the design of ETL flows of data warehouses under the prism of evolution in [30]. In [32], the same authors used a real world evolution scenario, which used the evolution of the Greek public sector's data warehouse maintaining information for farming and agricultural statistics. The experimental analysis of the authors is based in a six-month monitoring of seven real-world ETL scenarios that process the data of the statistical surveys. The Architecture Graph of the system was used as a provider of graph metrics. The findings of the study indicate that schema size and module complexity are important factors for the vulnerability of an ETL flow to changes.

In a later work [33], Manousis et al., redefine the model of the *Architecture Graph*. The paper extends the previous model by requiring the *high level modules* of the graph to include *input* and *output* schemata, in order to obtain an isolation layer that leads to the simplification of the policy language. The method is based on the annotation of modules with *policies* that regulate the propagation of events in the Architecture Graph; thus, a module can either block a change or adapt to it, depending on its policy. The method for impact assessment includes three steps that: (a) assess the impact of a change, (b) identify policy conflicts from different modules on the same change event, and (c) rewrite the modules to adapt to the change. It is noteworthy that simply flooding the evolution event over the *Architecture Graph* in order to assess the impact and perform rewrittings, is simply not enough, as different nodes can react with controversial policies to the same event. Thus, the three stages are necessary, with the middle one determining conflicts and a "cloning" method, for affect paths on the graph, in order to service conflicting requirements, whenever possible.

In Figure 2.1, we depict a situation that exemplifies the above. In the Architecture Graph that is displayed in the left part of Fig. 2.1, a change happens in view V_0 and affects the view V_1, which, in turn, affects the two queries Q_1 and Q_2 of the example. The first query (Q_1) accepts the change, whereas the second one (Q_2) blocks it. This means that Q_2 wants to retain its semantics and be defined over the old versions of

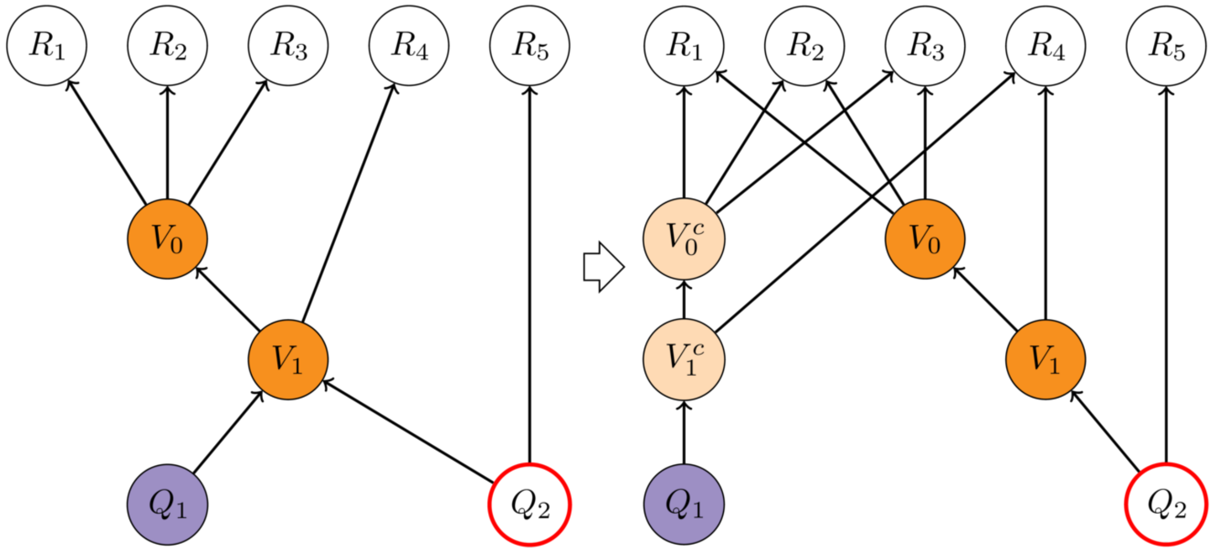


Figure 2.1: A example of a rewrite process when the policies of Q_1 and Q_2 queries are conflicting [34].

the views of the Architecture Graph. Therefore, the query that accepted the change will get a new path, composed of ”cloned”, modified versions of the involved views that abide by the change (depicted in light color in the left part of the figure and annotated with a superscript c), whereas the original views and their path towards Q_2 retain their previous definition (i.e., they decline the change).

Schema Modification Operators. In this section, we review a work that produces – when it is possible– valid query rewritings of old queries over a new database schema, as if the evolution step of the database schema never happened. This way, the results that the user receives, after the execution of the rewritten query, are semantically correct.

An approach that supports the ecosystem idea, to a certain extent, is [35]. In this approach, the authors propose a method that rewrites queries whenever one of their underlying relations changes with the goal of retaining the same query result as if the evolution event never happened, using *Schema Modification Operators* (SMOs). The Schema Modification Operators that PRISM/PRISM++ tool uses are:

- `CREATE TABLE` $R(a, b, c)$
- `DROP TABLE` R
- `RENAME TABLE` R `INTO` T
- `COPY TABLE` R `INTO` T

- MERGE TABLE R, S INTO T
- PARTITION TABLE R INTO S WITH condition, T
- DECOMPOSE TABLE R INTO $S(a, b)$ $T(a, c)$
- JOIN TABLE R, S INTO T WHERE condition
- ADD COLUMN d [AS constant | function(a, b, c)] INTO R
- DROP COLUMN r FROM R
- RENAME COLUMN b IN R TO d

The R, S, and T variables represent relations. The a, b, c, d, and r variables represent attributes. The constant variable stands for a fixed value, while the function is used in ADD COLUMN in order to express simple tasks as data type and semantic conversions are. Besides the schema modification operators, PRISM/PRISM++ uses the integrity constraints modification operators ICMO and policies (which will be described later on) for this kind of rewrites. The ICMOs are:

- ALTER TABLE R ADD PRIMARY KEY $pk1(a, b)$ <policy>
- ALTER TABLE R ADD FOREIGN KEY $fk1(c, d)$ REFERENCES $T(a, b)$ <policy>
- ALTER TABLE R ADD VALUE CONSTRAINT $vc1(c, d)$ AS $R.e$="0" <policy>
- ALTER TABLE R DROP PRIMARY KEY $pk1$
- ALTER TABLE R DROP FOREIGN KEY $fk1$
- ALTER TABLE R DROP VALUE CONSTRAINT $vc1$

The R and T variables represent relations. The a, b, c, d, and e variables represents attributes. The $pk1$ represents the primary key of the preceding relation. The $fk1$ represents the foreign key of a relation. Finally, the $vc1$ represents a value constraint. The ICMOs have, also, a <policy> placeholder, where the policy can be one of the following:

1. CHECK, where the PRISM/PRISM++ tool verifies that the current database satisfies the constraint, otherwise the ICMO is rolled back,

2. `ENFORCE`, where the tool removes all the data that violate the constraint, and,

3. `IGNORE`, where the tool ignores if there exist tuples that violate the constraint or not, but informs the user about this.

When the `ENFORCE` policy is used and tuples have to be removed, the tool creates a new database schema and inserts all the violating tuples in order to help the DBA carry out inconsistency resolution actions.

Regarding the rewrite process of queries through SMOs, the Chase & Backchase algorithm uses as input the SMOs and a query that is to be rewritten. The algorithm rewrites the query through an inversion step of the SMO's (for example, the inversion of a JOIN is the DECOMPOSITION), in order to retain the query's results unchanged, independently of the underlying schema. This way, the resulting tuples of the query will be the same as if the database schema never changed. The rewrite process of queries through ICMOs is done with the help of policies.

So, the steps that describe the algorithm of the rewriting that the authors proposed, are:

1. Get the SMOs from the DBA

2. Inverse the SMOs, in order to guarantee –if it is possible– the semantic correctness of the new query

3. Rewrite the query and validate its output.

The authors also describe a rewrite process of updates statement queries ("UPDATE table SET...") through SMOs and ICMOs, based in the ideas described in the previous paragraph. If the rewrite is through SMOs, the UpdateRewrite algorithm tries to invert the evolution step, while if the rewrite is through ICMOs, the policies ask the tool to check the tuples of the database and either guarantee or inform the user about the contents of the database.

To improve their rewrite time the authors try to minimize the input of the Chase & Backchase algorithm, by removing from the input all the mappings and constraints that are not related with the evolution step. Moreover, the proposed method uses only the version of the relation in which the query was written, leaving all the previous modifications out, as they are unrelated to the query. This is the backchase optimizer technique that the authors proposed, which produced bigger execution times in the chase and backchase phase of higher connected schemata because of the foreign keys that lead to higher input in chase phase, in the experiments that were conducted. In order to achieve even better execution time, the authors propose the use of a caching technique, since from the observations they made on their datasets, they noticed that there is a number of common query/update templates, which is parametrized and reused multiple times. These patterns are:

Join pattern type 1. In this pattern, a new table is created to host joined data from the desired column of two or more tables and migrates the data from the old tables to the new one.

Join pattern type 2. In this pattern, the data of a column are moved from the source table to the destination table.

Decompose pattern. In this pattern, a table is decomposed to two new tables. In order to be correct, both tables should have the key of the table.

Partition pattern. In this pattern, a part of the data of a table is moved into a new table and deleted from the original one.

Merge pattern. In this pattern, all the tuples of a table are moved into another table.

Copy pattern. In this pattern, an existing table is cloned.

The authors validated the PRISM/PRISM++ tool using the Ensembl project, including 412 schema versions, and the Mediawiki project, which is part of the Wikipedia project and had 323 schema versions. The authors used 120 SQL statements (queries and updates) from those two projects, tested them against SMO and ICMO operators and their tool found a correct rewriting, whenever one existed.

In a later work [7], the authors provide an extended description of the tool that performs the rewrites of the queries (PRISM/PRISM++) and its capabilities. Moreover, the authors introduce two other tools of which the first one collects and provides statistics on database schema changes and the other derives equivalent sequences of (SMOs) from the migration scripts that were used for the schema changes.

Summary

In Table 2.2 we summarize what are the problems that the selected works are dealing with. For example the first two works are dealing with the impact analysis problem, which is to identify which parts of the code is affected by a change, and the

Works	Problem	Input	Output	Method
[27]	Impact of DB schema changes to C# OO apps	DB schema and source code; an imminent change	The lines of code that are affected by the DB schema change	Slicing technique to identify the DB related lines of C# code, and estimation of values so as to further slice the C# code.
[28, 29]	Propagation of DB schema changes	DB schema and application's queries abstracted as *Architecture Graph*; policies for of the nodes; an imminent change	The *Architecture Graph* and a set of affected nodes.	Language for node annotation. Propagation of a change, based on the node's *policy* for the change.
[33]	Rewrite of DB schema and applications	DB schema and application's queries abstracted as *Architecture Graph*; policies for of the nodes; an imminent change	New DB and queries that acquired the change	Rewrite via cloning the queries that want to acquire the change and leave intact the ones that block the change.
[35, 7]	Hiding the DB schema change via rewriting	SMOs and ICMOs of the modification, and queries that use the modified table/view	Rewritten queries that "hide" the DB schema change	The 1 hop away queries are rewritten as if the schema change never happened, using the *Chase & Backchase* algorithm

Table 2.2: Summary table for Section 2.2.3

other two works are dealing with the rewriting of the code in order to obtain or hide the schema changes.

Views: rewriting views in the context of evolution

A view is a query expression, stored in the database dictionary, which can be queried again, just as if it was a regular relation of the database. A view, thus, retains a dual nature: on the one hand, it is inherently a query expression; yet, on the other hand, it can also be treated as a relation. A *virtual view* operates as a macro: whenever used in a query expression, the query processor incorporates its definition in the query expression and the query is executed afterwards. *Materialized views* are a special category of views, that persistently store the results of the query in a persistent table of the DBMS. In this section, we survey research efforts that handle two problems. First, we start with the problem of materialized view redefinition: the expression defining the view is altered and the stored contents of the view have to be adjusted to fit the new definition (ideally, without having to fully recompute the contents of the view from scratch). Second, we survey efforts pertaining to how views should be adapted when the schema of their defining tables evolves.

In [36], Mohania deals with the problem of maintaining the extent of a materialized view that is under redefinition, by proposing methods that try to avoid the full re-computation of the view. The author uses *expression trees*, which are binary trees, the leaf nodes represent base relations that are used for defining the view, while the rest of the nodes contain binary relational algebraic operators. Unary operators such as selection and projection are associated with the edges of the tree. In a nutshell, the author proposes that making use of these expression trees, it is easy to find common subexpressions between the new and old view statements and thus, if applicable, make use of the old view to get the desired results of the redefined view, without recomputing the new definition. Due to its structure, the tree allows to avoid interfering with the result of the view computation: (a) the height of the trees is no more than two levels, and, (b) a change is either a change to a unary operator associated with the edge of the tree, or a change to a binary node. This way, when the change is made at the root node, then the expression corresponding to the right hand child in the tree has to be evaluated only, while when the change is made at level d=1, the view re-computation becomes a view maintenance problem. Finally, when the change is made at any other node, it is only the intermediate results of the nodes that have to be maintained.

Gupta, Mumick, Rao and Ross [37] provide a technique that redefines a materialized view and adapts its extent, as a sequence of primitive local changes in the view definition, in order to avoid a full re-computation. Moreover, on more complex adaptations –when multiple simultaneous changes occur on a view– the local changes are pipelined in order to avoid intermediate creations of results of the materialized view. The following changes are supported as primitive local changes to view definitions:

1. Addition or deletion of an attribute in the `SELECT` clause.
2. Addition, deletion, or modification of a predicate in the `WHERE` clause (with and without aggregation).
3. Addition or deletion of a join operand (in the `FROM` clause), with associated equijoin predicates and attributes in the `SELECT` clause.
4. Addition or deletion of an attribute from the `GROUP BY` list.
5. Addition or deletion of an aggregate function to a `GROUP BY` view.
6. Addition, deletion or modification of a predicate in the `HAVING` clause. Addition of the first predicate of deletion of the last predicate corresponds to addition and deletion of the `HAVING` clause itself.
7. Addition of deletion of an operand to the `UNION` and `EXCEPT` operators.
8. Addition or deletion of the `DISTINCT` operator.

Concerning the problem of adapting a view definition to changes in the relations that define it, Nica, Lee and Rundensteiner [38] propose a method that makes legal rewritings of views affected by changes. The authors primarily deal with the case of relation deletion which (under their point of view) is the most difficult change of a database schema, since the addition of a relation, the addition of an attribute, the rename of a relation and the rename of an attribute can be handled in a straightforward way (the attribute deletion, according to the authors, is a simplified version of the relation deletion). To attain this goal one should find valid replacements for the affected components of the existing view, so, in order to achieve that, the authors of [38] keep a Meta-Knowledge Base on the join constraints of the database schema. This Meta-Knowledge Base (MKB) is modeled as a hyper-graph that keeps meta-information about attributes and their join equivalence attributes on other tables. The proposed algorithm, has as input the following: (a) a change in a relation,

(b) MKB entities, and, (c) new MKB entities. Assuming that valid replacements exist, the system can automatically rewrite the view via a number of joins and provide the same output as if there was no deletion. The main steps of the algorithm are: (a) find all entities that are affected for Old MKB to became New MKB, (b) mark these entities and for each one of them find a replacement from Old MKB, using join equivalences, and, (c) rewrite the view over these replacements. Interestingly, the authors accompany their method with a language called E-SQL that annotates parts of a view (exported attributes, underlying relations and filters) with respect to two characteristics: (a) their dispensability (i.e., if the part can be removed from the view definition completely) and (b) their replaceability with an another equivalent part.

Work	Problem	Input	Output	Method
[36]	Maintenance of redefined materialized views	Definition and redefinition of a materialized view	Recomputed content of the redefined view	Use of expression trees that identify common subexpressions between the input and output of their method, thus helping to avoid the full recomputation of a materialized view
[37]	Maintenance of redefined materialized views	Definition and redefinition of a materialized view	Recomputed content of the redefined view	The redefinition takes place as a sequence of primitive local changes (in complex adaptations this sequence is pipelined to avoid temporal results).
[38]	View adaptation on column deletion	Hypergraph that contains the join constraints of the DB schema	Valid replacement of column that is to be deleted	Search in the hypergraph (named MKB) for a replacement of the column that is to be deleted, and replace that column in the view with the replacement

Table 2.3: Summary table for Section 2.2.3

2.2.4 Techniques for managing data warehouse evolution

A research area where the problem of evolution has been investigated for many years is the area of data warehouses. In this section, we concentrate on works related on evolution of both schema and data modifications in the context of data warehouses, and we review methods and tools that help on the adaptation of those changes. We also refer the reader to two excellent surveys on the issue, specifically, [39] and [40].

Data warehouses and views. At the beginning of data warehousing, people tended to believe that data warehouses were collections of materialized views, defined over sources. In this case, evolution is mostly an issue of adapting the views definitions whenever sources change.

Bellahsene, in two articles, [41] and [42], proposed a language extension to annotate views with a HIDE clause that works oppositely to SELECT (i.e., the idea is to project all attributes except for the hidden ones and an ADD ATTRIBUTE clause to equip views with attributes not present in the sources (e.g., timestamps or calculations). Then, in the presence of an event that changes the schema of a data warehouse source (specifically, the events covered are attribute/relation addition and deletion), the methods proposed by the author for the adaptation of the warehouse handle the view rematerialization problems i.e., how to recompute the materialized extent via SQL commands. The author also proposes a cost model to estimate the cost of alternative options.

In [43], the author proposes an approach on data warehouse evolution based on a meta-model, that provides complementary metadata that track the history of changes (in detail, changes that are related to data warehouse views) and provide a set of consistency rules to enforce when a quality factor (actual measurement of a quality value) has to be re-evaluated.

Evolution of multidimensional models. Multidimensional models are tailored to treat the data warehouse as a collection of *cubes* and *dimensions*. Cubes represent clean, undisputed facts that are to be loaded from the sources, cleaned and transformed, and eventually queried by the client application, Cubes are defined over unambiguous, consolidated dimensions that uniquely and commonly define the context of the facts. Dimensions comprise *levels*, which form a hierarchy of degrees of detail according to which we can perform the grouping of facts. For example, the Time dimension can include the levels (1) Day, that can be rolled up to either (2a) Week or (2b)

Month, both of which can be rolled up to level (3) Year. Each level comes with a domain of values that belong to it. The values of different levels are interrelated via rollup functions (e.g., 1/1/2015 can be rolled up to value 1/2015 at the Month level). As levels construct a hierarchy that typically takes the form of a lattice, evolution is mainly concerned with changing (i) the nodes of the lattice, or (ii) their relationship, or (iii) the values of the levels and their interrelationship.

The authors of [44] present a formal framework, based on a formal conceptual description of an evolution algebra, to describe evolutions of multi-dimensional schemata and their effects on the schema and on the instances. In [44], the authors propose a methodology that supports an automatic adaptation of the multi-dimensional schema and instances, independently of a given implementation. The main objectives of the proposed framework are: (i) the automatic adaptation of instances, (ii) the support for atomic and complex operations, (iii) the definition of semantics of evolution operations, (iv) the notification mechanism for upcoming changes, (v) the concurrent operation and atomicity of evolution operations, (vi) the set of strategies for the scheduling of effects and (vii) the support of the design and maintenance cycle.

The authors provide a minimal set of atomic evolution operations, which they use in order to present more complex operations. These operations are: (i) insert level, (ii) delete level, (iii) insert attribute, (iv) delete attribute, (v) connect attribute to dimension level, (vi) disconnect attribute from dimension level, (vii) connect attribute to fact, (viii) disconnect attribute from fact, (ix) insert classification relationship, (x) delete classification relationship, (xi) insert fact, (xii) delete fact, (xiii) insert dimension into fact, and, finally, (xiv) delete dimension.

In [45], the authors suggest a set of primitive dimension update operators that address the problems of: (i) adding a dimension level, above (generalize) or below (specialize) an existing level, (ii) deleting a level, (iii) adding or deleting a value from a level (add/delete instance), or (iv) adding (relate) or removing edges between parallel levels (unrelate). In [45], the authors also suggest another set of complex operators, that intend to capture common sequences of changes in instances of a dimension and encapsulate them in a single operation. The set of those operators consists of: (i) reclassify (used, for example, when new regions are assigned to salespersons as a result of marketing decisions of a company), (ii) split (used, for example, when a region is divided into more regions and more salespersons must be assigned to those regions due to the population density), (iii) merge (the opposite of split), and, (iv) update (used, for example, when a brand name for a set of items changes but the corporation as well as the set of products related to the brand remain unchanged).

The mappings that the authors propose, for the transitions from the multidimensional to the relational model, support both the de-normalized and normalized relational representations. In the de-normalized approach, the idea is to build a single table containing all the roll-ups in the dimension while in the normalized approach, the idea is to build a table for each direct roll-up in the dimension.

Finally, in the experiments that the authors conducted, they found that the structural update operators in the de-normalized representation are more expensive. The instance update operators in the normalized representation are more expensive because of the joins that have to be performed, whilst both representations are equally good for the operators that compute the net effect of updates.

In a later work, the authors of [46] suggest a set of operators which encapsulate common sequences of primitive dimension updates and define two mappings from the multidimensional to the relational model, suggesting a solution on the problem of multidimensional database adaptation.

The effects of evolution to alternative relational logical designs is explored in [47]. Specifically, the authors explore the impact of changes to both star and snowflake schemata. The changes covered include (i) the addition of deletion of attributes to levels, (ii) the addition/deletion of dimension levels, (iii) the addition/deletion of measures, and (iv) the addition/deletion of dimensions into fact tables. A notable, albeit expected, result is that comparison of the effect of changes to the two alternative structures, reveals that the simplest one, star schema, is more immune to change than the more complicated one.

Multiversion querying over data warehouses Once the research community had obtained a basic understanding of how multidimensional schemata can be restructured, the next question that followed was: “what if we keep track of the history of all the versions of a data warehouse schema as it evolves?” Then, we can ask queries that span several versions having different structure, also known as *multiversion queries*. The essence of multi-version queries involves transforming the data of previous versions (that obey a previous structure) to the current version of the structure of the data warehouse, in order to allow their uniform querying with the current data.

In this section, we discuss the adaptation of multiversion data warehouses [48], the use of data mining techniques in order to detect structural changes in data warehouses [49, 50, 51], and, the use of graph representations (directed graphs) [52], in order to achieve correct cross version queries.

Eder and Koncilia [51] propose a multidimensional data model that allows the registration of temporal versions of dimension data in data warehouses. Mappings are provided to transfer data between different temporal versions of instances of dimensions. This way, the system can answer correctly queries that span in periods where dimension data have changed. The paper makes no assumption on dimension levels, so when referring to a dimension, the paper implies a flat structure with a single domain. The mappings are described as transformation matrices. Each matrix is a mapping of data from version $\mathtt{V}_i$ to version $\mathtt{V}_{i+1}$ for a dimension $\mathtt{D}$. Assume, for example a 2-dimensional cube, including dimensions A and B with domains $\{a_1, a_2\}$ and $\{b_1, b_2\}$ respectively. Assume that in a subsequent version: (i) a_1 is split to a_1^1 and a_1^2 and (ii) b_1 and b_2 are merged into a single value b. Then, there is a transformation matrix for dimension A, with one row for each old value $\{a_1, a_2\}$ and one column for each new value $\{a_1^1, a_1^2, a_2\}$ expressing how the previous values relate to the new ones. For example, one might say that a_1^1 takes 30% of a_1 and a_1^2 takes the other 70%. The respective matrix is there for dimension B. Then, by multiplying any cube with A and B as dimensions with the respective transformation matrices, we can transform an old cube defined over $\{a_1, a_2\} \times \{b_1, b_2\}$ to a new cube defined over $\{a_1^1, a_1^2, a_2\} \times \{b\}$.

So at the end, the resulting factual cube maps the data of the previous version to the dimension values of the current version; this way, both the current and the previous version can be presented uniformly to the user.

Eder, Koncilia and Mitsche [49] propose the use of data mining techniques for the detection of structural changes in data warehouses, in order to achieve correct results in multi-period data analysis OLAP queries. Making use of three basic operations (INSERT, UPDATE and DELETE), the authors are able to represent more complex operations such as: SPLIT, MERGE, CHANGE, MOVE, NEW-MEMBER, and DELETE-MEMBER. The authors propose several data mining techniques that detect which is the schema attribute that changed. In the experiments that were conducted, the authors observed that the quality of the results of the different methods depends on the quality and the volatility of the original data.

The same authors continue their previous work on data mining techniques for detection of changes in OLAP queries in [50]. Since their previous approach was incapable of detecting some variety of changes, the authors propose data mining techniques in form of multidimensional outlier detection to discover unexpected deviations in the fact data, which suggests that changes occurred in dimension data. By fixing a dimension member they get a simple two-dimensional matrix where the one axis is the excluded dimension member. From that matrix, a simple deviation matrix with relative differences is computed. In this deviation matrix, the results are normalized to get the probability of a structural change that might have occurred. The authors propose the 10% as a probability threshold for the change to have occurred. From the conducted experiments, the authors found that this method analyzes the data in more detail and gives a better quality of the detected structural changes.

Some years later, Golfarelli et al. [52] propose a representation of data warehouse schemata as graphs. The proposed graph represents a data warehouse schema, in which the nodes are: (i) the fact tables of the data warehouse, and (ii) the attributes of fact tables (including properties and measures), while the edges represent simple functional dependencies defined over the nodes of the schema. The authors also define an algebra of schema graph modifications that are used to create new schema versions and discuss of how cross-version queries can be answered with the help of augmented data warehouse schemata. The authors finally show how a history of versions for data warehouse schemata is managed.

Since the authors' approach is based on a graph, the schema modification algebra uses four simple schema modification operations (M): (i) $\texttt{Add}_F$ that adds an arc involving existing attributes, (ii) $\texttt{Del}_F$ that deletes an existing arc, (iii) $\texttt{Add}_A$ that adds a new attribute –directly connected by an arc to its fact node– and (iv) $\texttt{Del}_A$ that deletes an existing attribute. Besides those simple operators, the authors define the $\texttt{New}(S, M)$ operator that describes the creation of a new schema, based on the existing schema S when a simple schema modification M is applied.

The authors introduce augmented schemata to serve multiversion queries. Each previous version of the data warehouse schema is accompanied by an augmented schema whose purpose is to translate the old data under the old schema to the current version of the schema. To this end, the augmented schema keeps track of every new attribute (say A), or new functional dependency (say f). In order to translate the old data to the new version of the schema, the system might have to: (i) estimate values for A, (ii) disaggregate or aggregate measure values depending on the change of granularity, (iii) compute values for A, (iv) add values for A, or, (v) check if f holds.

The set of versions of the schemata is described by a triple (S, S^{AUG}, t), where S is a version, S^{AUG} is the related augmented schema and t is the start of the validity interval of S. This way, the history of the versions of the data warehouse can be described as a sequence of changes over changes, starting from the initial schema of the history: $H = S_0, S_0^{AUG}, t_0$. Since every previous version is accompanied by an augmented schema that transforms it to the current one, it is possible to pose a query that spans different versions and translate the data of the previous versions to a representation obeying the current schema, as explained above.

Practically around the same time, Wrembel and Bebel [48] deal both with cross-version querying and with the problems that appear when changes take place at the external data sources (EDS) of a data warehouse. Those problems can be related to a multi-version data warehouse which is composed of a sequence of persistent versions that describe the schema and data for a given period of time. The authors approach has a meta-data model with structures that support: (i) the monitoring of the external data sources on content and structural changes, (ii) the automated generation of processes monitoring external data sources, (iii) the adaptation of a data warehouse version to a set of discovered external changes, (iv) the description of the structure of every data warehouse version and (v) the querying of multiple data warehouse versions (cross version querying), and (vi) the presentation of the output as an integrated result.

The schema change operations that the authors support are: (i) the addition of a new attribute to a dimension level table, (ii) the removal of an attribute from a dimension level table, (iii) the creation of a new fact table, (iv) the association of a fact table with a dimension table, (v) the renaming of a table, and three more operations that are applicable to snowflake schemata, (vi) the creation of a new dimension level table with a given structure, (vii) the inclusion of a parent dimension level table into its child dimension level table, and, (viii) the creation of a parent dimension level table based on its child level table.

The instance change operations that the authors have worked on, are: (i) the insertion of a new level instance into a given level, (ii) the deletion of a level instance, (iii) the change of the association of a child level instance to another parent level instance, (iv) the merge of several instances of a given level into one instance of the same level, and (v) the split of a given level instance into multiple instances of the same level.

In order to query multiple versions, the authors' method is based on a simple and elegant idea: the original query is split to a set of single version queries. Then, for each single version query, the system does a best-effort approach: if, for example, attributes are missing from the previous version, the system omits them from the single version query; the system exploits the available metadata for renames; it can even, ignore a version, if the query is a group by query and the grouping is impossible. If possible, the collected results are integrated under the intersection of attributes common to all versions (if this is the case of the query); otherwise, they are presented as a set of results, each with its own metadata.

Regarding the detection of changes in external data sources, the authors propose a method that uses wrappers (software modules responsible for data model transformations). Each wrapper is connected to a monitor (software that detects predefined events at external data sources). When an event is detected, a set of actions is generated and stored in *data warehouse update register* in order to be applied to the next data warehouse version when the data warehouse administrator calls the warehouse refresher. The events are divided into two categories: (i) structure events (which describe a modification in the schema of the data warehouse) and (ii) data events (which describe a modification in the contents of a data warehouse). For each event, an administrator defines a set of actions to be performed in a particular data warehouse version. The actions are divided in two categories: (i) messages (which represent actions that cannot be automatically applied to a data warehouse version) and (ii) operations (for events whose outcomes can be automatically applied to a data warehouse version). Both categories of actions do not create a new data warehouse version automatically but require either the administrator to apply them *all* in an action definition of an explicitly selected version, or the actions are logged in a special structure for manual application of the ones the administrator wants to apply.

Works	Problem	Input	Output	Method
[51]	Data transfer between versions of a Data Warehouse	Data Warehouse over time	Queries that are correct over the time span changes of a Data Warehouse	Transformation matrices that are mappings between the different versions of the Data Warehouse
[49, 50]	Data transfer between versions of a Data Warehouse	Data Warehouse over time	Queries that are correct over the time span changes of a Data Warehouse	Data mining techniques that identify Data Warehouse schema changes and dimension changes, using a normalized matrix
[52]	Correct queries over the changes of a Data Warehouse	Data Warehouse over time	Queries that are correct over the time span changes of a Data Warehouse	Graphs with a simple algebra that describes changes and augmented schemata to “find” the values of columns that didn’t exist
[48]	Cross version queries and changes of external data sources	Data Warehouse over time, with their data provider	Queries that are correct over the time span changes of a Data Warehouse	Decompose a query to queries that are correct at each schema version. For the EDS use of wrappers that “signal” software monitors to perform rules set by the administrator on what to do for the specific change

Table 2.4: Summary table for multidimensional model evolution

2.3 Query Extraction

The query extraction problem covers the problem of reverse engineering and reconstructing the queries that are embedded in an application written in a host language (like Java, PHP, etc.). In this section, we review the state of the art on the problem of the query extraction. The presented methods are mostly used for the error checking/testing of database-related projects, for examining the impact of database schema changes to the database-related applications, or for identifying the bottlenecks of the applications' code.

Several previous works identify database accesses by extracting dynamically constructed SQL queries (JDBC-based database accesses) from the bytecode of a Java program. Those works deal with the problem of error checking or fault diagnosis, query testing prior to execution, as well as, the impact analysis for database schema changes problem.

Regarding the string concatenation query construction for error detection there is a work of Christensen et al. [53], which proposes a static string analysis technique that uses as input a Java program. This program is translated into a flow graph, and analysed to generate a finite state automaton. The authors evaluate their approach on Java classes with at most 4 kLOC. In the same problem (typographical and syntactical errors because of queries that are constructed through string concatenation), there is also another method of Gould et al., presented in [54, 55]. The proposed method is based on an data-flow analysis and includes the following steps. First, the Java source code is sliced to identify the SQL related parts and produce a finite state automaton. Then, from those string parts and the automaton, via a DFS traversal, all the variations of a query are formed. Those variations are finally tested for type (using the DB schema description) and syntax errors.

The method described Annamaa et al. in [56] presents a way of testing the DB queries before their execution. The proposed method identifies the SQL queries embedded into Java source code via searching for functions that the user indicated. Using that as input, the proposed method constructs Abstract Syntax Trees. Those ASTs produce statements of a grammar that is later on used by a lexical analyser and syntactic parser for finding if any errors exist.

Van den Brink et al. present a quality assessment approach for SQL statements embedded in PL/SQL, COBOL and Visual Basic code in [57]. The initial phase of their method consists in extracting the SQL statements from the source code using control and data-flow analysis techniques. They evaluate their method on COBOL programs with at most 4 kLOC.

Ngo and Tan [58] make use of symbolic execution to extract database interaction points from web applications. Their research consists of a case study of PHP applications with sizes ranging 2 to 584 kLOC. Their method is able to extract about 80% of those interactions.

The method presented in [27] by Maule et al. uses a k-CFA algorithm and a software dependence graph to identify the impact of relational database schema changes upon object-oriented applications. The method identifies the parts of the source code that are affected by a change in the DB schema. The proposed method has as first step the slicing of the C# source code of a project to identify SQL related lines of code. In the next step, data-flow analysis is performed, to compute a possible set of run-time values that may occur at a given point of the running program. Finally, the parts of the source code that will be affected, when a database schema change occurs are given as output.

In another line of research, Cleve et al. perform a research on query extraction using the traces of the queries that were executed in the DBMS during the execution of a program. In [59], the authors try to identify the file name and the line number at that file where a query was executed. The authors, using the traces, try to identify the specific location in the source code of a project that a query was executed. The main goal of this work is to identify which parts of the code are responsible for database downtime when a query is executed on the server. To do so, the authors propose a tree based query representation of the queries that were found in the source code, and use a "joker" node in the parts of the query that there are host language variables. Then, they use the traces of the DBMS that contain the "problematic" queries, to represent those queries to trees too. In the next step, the authors perform a comparison of the trees and each line of the source code that matches is presented to the source code developers.

	Host languages	Query Type	#Variants
Christensen et al. [53]	Java	String-based	
Gould et al. [54, 55]	Java	String-based	
Annamaa et al. [56]	Java	String-based	ASTs
Van den Brink et al. [57]	PL/SQL, COBOL, V. Basic	String-based	
Ngo and Tan [58]	PHP	String-based	
Maule et al. [27]	C#	String-based	
Cleve et al. [59]	Java	String-based	Partial

Table 2.5: A structured overview of the state of the art

2.4 Software Metrics

The Software Metrics are a way to describe when the software is easily understandable, and maintainable. In this section, we survey the related work that pertains to the Software metric issue. We discuss works related to coupling and cohesion metrics in software, we move on to works that examine metrics in other areas such as web services and data warehouses.

The well-known LCOM metrics (Lack of Cohesion of Methods) for measuring the cohesion of object-oriented software was proposed in [60] which describes all the fundamental ideas that are related to cohesion and coupling measurements. The LCOM metrics are generally based on bipartite graphs where the first group contains the subroutines of the examined class and the second group contains the variables. The general idea of LCOM is that using that graph, LCOM measures the number of pairs of subroutines that have none common variable minus the number of pairs that have common variables.

Coupling is usually contrasted with cohesion, where high cohesion means low coupling and vice versa. In coupling metric one examines whether a class is dependent on another class, thus a change on the first would probably result a change on the second class too.

In [61] the authors performed a study where they examined C++ object-oriented projects to assess the metrics that were introduced in [60], and they concluded that those metrics could predict dificult to maintain code during the early phases of the projects, compared to other metrics that could be collected only at later phases of the projects' lifecycle. The interested reader may refer to [62] and [63] for two detailed surveys of the cohesion metrics that have been proposed since [60].

In another line of research [64], the authors examine a number of coupling and cohesion metrics and their correlation on the quality of three open source programs written in Java programming language. The authors via their evaluation concluded that cohesion and coupling are inversely proportional and that high cohesive classes probably need decoupling.

Another method of measuring code quality is via using Function Points [65]. Function Points are used as the measurement unit of the software development in business logic. This way, one may use Function Points to express that a great amount of money will be needed for a new project or for the maintenance of a project since the amount of Function Points is high.

Metrics in other areas: Besides the object oriented software measurements, there have been approaches on other areas, such as measuring the cohesion in web services ([66], [67], [68]). The first efforts for measuring cohesion in services have been made in [69]. Finally, another interesting work that concerns the cohesion of services is presented in [70]. Finally, in [71] the authors propose a catalogue of 13 “bad smells” in data-intensive ecosystems, and evaluate the 9 of them using a survey. Those smells are of one of the following categories: either schema issues (e.g. *god table*), or query *e.g. misused null*, or data (*e.g. intermingled data types*) related. To detect those smells, the authors had to extract the query statements that were embedded in the source code of the projects they examined using regular expressions.

2.5 Query rewriting

The query rewriting problem covers the problem of smoothly evolving from a database schema to a new schema, without having syntactic or semantic problems. Most works of this area are related to query rewriting using materialized views.

The latest algorithm that is related to query rewriting using views is Minicon, presented in [72]. The authors based their work on Bucket Algorithm ([73]). In Bucket Algorithm, the new query (*NQ*) is subdivided to subgoals. Any view that produces a result for a subgoal is inserted into the subgoal’s bucket. Finally, the Bucket Algorithm combines every possible combination of views in the buckets to produce the answer of *NQ* query. Minicon Algorithm uses the same technique, but instead of inserting every possible view of a subgoal in its bucket, it checks and uses only the views that can also be used to solve the *NQ* query, thus having a Cartesian product to search of smaller size.

Besides Bucket and Minicon algorithms, there exists another work ([74]) that is called Inverse-Rules Algorithm. In Inverse-Rules Algorithm, the authors propose a way of inverting the view definitions. This way, the authors can write a new query as if they use the view sources instead of the views. The results produced by the inversion step of the algorithm can be reused, compared to the previous two algorithms which produce only a valid writing of a new query in each algorithm run, but since the results are more general, one may have as result view sources that are irrelevant to the new query.

2.6 Visualization of Data Intensive Ecosystems

The visualization of data intensive ecosystems problem covers the problem of representing an ecosystem in a way that provides as much information as possible, without disturbing the viewer of the representation with visual clutter. Related work concerns aspects of software visualization, visualization fundamentals, recent advances in graph drawing and visualization for analysis tools/methods.

Software visualization. Diagrammatic representation and software visualization support the work of the developers in many ways. A survey over 400 Microsoft employees [75] has distilled 3 main reasons where *a diagrammatic representation is essential for the developers' tasks*: (a) code understanding, (b) code design and refactoring, and (c) ad-hoc meetings. In the field of software engineering there are plenty of methods for visualizing object-oriented systems; we constrain the discussion in classic and recent advances only. Tree-maps [76] is a classic method, extensively used –see, for example, `http://www.makelinux.net/kernel_map` for the visual representation of the Linux kernel. Treemaps allow the recursive splitting of the screen's area in nested rectangles to represent the hierarchical structure of a data set (e.g., a directory with source code files). SeeSoft [77] is another classic method, relying on the idea of file "thumbnails" (where each line of code is represented by a small line or some pixels on the screen and different colors corresponding to different statistics of the code, e.g., age, developer). Recently, the Code Canvas system [75] has been based on the idea of having a master "map" of the information system, that serves as a starting point for the developers to interactively browse the code via alternative views of the master map, customized to their needs. Code Bubbles [78] is a similar effort providing a reference canvas where the users can incrementally build a personalized map according to the needs of the current session. Despite the plethora of approaches in software engineering (see [79] for a recent survey), to the best of our knowledge, there are no similar efforts in the field of data-intensive information systems engineering.

Diagramming fundamentals. The fundamental concepts that govern user perception of visually demonstrated information have been investigated by the Gestalt school of psychology founded in 1912 and can be summarized as follows [80]:(i) *proximity* (objects close to each other tend to be perceived as similar), (ii) *similarity* (objects of the same shape, color, orientation and size are perceived similar by individuals), (iii) *connectedness* (to express semantic relationship among visually connected objects), (iv) *closure* (the eye tends to create perceptions of closed space, even if they do not exist – best served when the depicted objects tend to create a "border" around similar objects along with blobs of whitespace), (v) *continuity* (as the eye tends to perceive as related objects that are aligned together), (vi) *symmetry* (as a means to emphasize non-typical behavior or emphasis when symmetry is broken by an object), (vii) *contrast*, achieved in terms of chromatic, size or shape choices, (viii) *proportion* (where an object placed in a area of the visualization is scaled according to its semantic significance, as the difference in proportion creates a visual attraction to the eye). We also take into consideration best practices [81] closely related to the above Gestalt principles like (ix) *clutter avoidance* (i.e., the avoidance of noise on the diagram via uninterrupted areas of white-space that act as separators of the groups of objects), (x) *isolation*, to promote emphasis for an object in sharp antithesis to the continuity of the vast majority of the "regular" objects, (xi) *visual hierarchy*, to denote a semantic hierarchy in the depicted objects, (xii) *focal points* to guide *visual flow* (i.e., objects that intentionally stand out in the representation and whose sequence guides the eye in the visual flow of exploring the diagram). In [82] the author proposes a model for the nested visualization design and validation that is based in four layers. The four levels are: (a) characterize the tasks and data in the vocabulary of the problem domain, (b) abstract into operations and data types, (c) design visual encoding and interaction techniques, and (d) create algorithms to execute these techniques efficiently. Having this model, the author presents a number of examples, and how those examples pertain to the proposed model.

Graph drawing. In terms of drawing techniques, the research that mostly pertains to our method involves circular graph drawing. This is due to the increased ability of circular methods to clearly demonstrate natural group structures – clusters – within the overall graph. In [83] the authors propose a technique for producing circular drawings, using fixed angles on biconnected graphs with the goal of minimizing edge crossings. The method places (a) edges towards the circumference of the embedding circle and (b) the neighbors of a node as close as possible to the node. [84] proposes a technique that uses fixed angles to place disks on the circumference of a circle. The disks are either touching the circumference, or in case their size is greater than the angle that is predefined for them, they are moved further from the circle, till they fit in the predefined angle. In [85] the authors propose a visualization technique using circular layouts. In this method, the nodes of the graph are divided into two groups, the "anchor nodes" that are arranged on the circumference of a circle in fixed angles, and the "free nodes" that are positioned inside the circle. The final position of the "free nodes" depends on how similar they are to the nodes of the circumference. For example, if we have 3 "anchor nodes" and a "free node" that depends equally on each one of the three, then this node will be placed in the center of the circle. In [86] the authors propose a similar to [85] method for drawing bipartite graphs in circular layouts. In this proposal, the "free nodes" are positioned in the circular disk in relation to the adjacent anchor maps. A simulated annealing algorithm provides the final graph arrangement via the iterative computation of a cost for misplacing free nodes with respect to anchor nodes. In [87] the authors present a number of circular visualization methods, such as pie chats, star plots, spirals, etc. The majority of those methods, does not contain any dependency information between the nodes, except the connected ring pattern, which contains cross edges between the nodes that are placed in the circumference of a circle, which provides much visual noise. A method to reduce the visual clutter is by bundling the edges.

Visualization for analyzing. Since there is a majority of data available (Big Data), there is a need of tools to properly retrieve those data, and visualize them. DIVE [88] is data-agnostic framework for big data analysis, which uses pipelines in order to retrieve the data from the sources, and via an object oriented approach to locate the data that the user wants to see. DIVE is extensible through script language (DIVE mostly is .NET oriented) and includes some pipelines for 2D and 3D representations.

Another approach [89] is the DOSA tool where the authors let the user have an overview of all the available data and the connections between them, and they provide an info-graphic style of visualization. The user of DOSA selects specific areas that he wants to see as a group (e.g. New York and California, in a geographical dataset) and he has the detailed view in one area of his screen and the info-graphic style in another area. A work that is related to visualization and analysis of changes that occur in the graph that is depicted is GraphDiaries [90]. This tool, highlights with color the nodes that are to be deleted, or inserted to the examined graph at a specific moment. The tool also provides three layout methods: (a) fixed (globally optimized) for all the examined steps, (b) locally optimized layout per step, and (c) any possible combination of the two previous. From the user study the authors performed, they observed that the users focus on different changes of the graph, depending on what layout was used.

2.7 Comparison to the state of the art

Query Extraction. Concerning the topic of query extraction, the state of the art has been able to provide a solution for the problem of query extraction when one hosting language is used (if there is any solution that can be used in more than one hosting language, we were not able to find any experiment describing that). Moreover, the state of the art works only with string based queries, leaving out the ones that use an API to get constructed (object-based queries). At the same time, there are issues that remain unsolved. To the best of our knowledge, there is no principled method to extract queries of different programming styles in a single, reference language-independent query representation. To this end, in Chapter 3, we address this shortcoming by proposing a method that takes the source code of data-intensive ecosystem, and produces the abstract representation of all the variants of the embedded queries in a form that is possible to be further exploited by subsequent tasks, such as creating the Architecture Graph of the ecosystem or migrating to another language.

Metrics. Regarding the topic of metrics, the state of the art provides solutions concerning the metrics of software, which in most cases is object oriented software. The provided solutions describe which principles a software metric should have. Additionally, the metrics are capable to express the different costs of maintenance in an object oriented software application. Moreover, there are works that examine metrics in other areas, such as services (e.g. web services). Regarding the software-database metrics area, we were not able to locate any principles that such a metric should retain. To this end, in Chapter 4, we describe the principles of what a software-database related metric should have, and we provide such a metric that is in sync with the Lehman's Laws of evolution. Additionally, we introduce views and query rewrites in order to achieve better metric values, which provides lower maintenance effort for the developers.

Query rewriting. Concerning the query rewriting for evolution issue, the state of the art is mostly concerned about the rewriting of materialized views. That is because the rewriting of materialized views is both time and resources consuming. To achieve that, the state of the art proposes solutions that reverses the schema evolution in order to keep intact the software part of a data-intensive ecosystem. In Chapter 5 we introduce the mapping of the source code to the database schema, using the *Architecture Graph*. The *Architecture Grpah* can be used in many tasks, such as the program comprehension, impact analysis, documentation etc. In this chapter, we focus on the impact analysis of a change, using policies over the events on the nodes of the *Architecture Graph*. This way, we smoothly adapt the database to its new schema, in sync with the surrounding software.

Visualization. The work presented in [1] contains a set of algorithms that reduce the visual clutter that exists when we visualize the *Architecture Graph* of a data-intensive ecosystem. This way the *Architecture Graph* can highlight the correlation of the code to the database. In extension to [1] we propose a "what-if" visualization algorithm that was missing from the state of the art, and, additionally, we perform a user study for the new algorithm, and the ones described in [1], regarding the user satisfaction, and the code understanding that the visualization methods provide.

Chapter 3

Query Extraction

3.1 Introduction

To operate properly, data-intensive applications rely on *embedded queries*, that are programmatically constructed (typically, in progressive, incremental fashion) to facilitate the retrieval of data from the underlying databases. *Identifying the location and semantics of these queries and making them available to developers is very important*. In a most common scenario, database schema migration, refactoring and evolution require the appropriate visualization and inspection of data-related code, spread across multiple modules and files, for evaluating the impact of the schema change to the overall software ecosystem. As another example, when an administrator wants to modify a part of the database, it is imperative that the developers of the surrounding applications are informed on the change and have the means to identify the parts of the code that are going to be affected by that change [27],[33]. In Chapter 6 we demonstrate that the availability of this information can facilitate the visualization of the internal code architecture in terms of application-to-database interdependencies [1] and the management of the impacts of evolution [33].

Yet, obtaining these queries is an extremely painful process. An embedded query is, typically, progressively constructed via a sequence of source code statements that modify the query internals according to user choices. In the past, the most popular way to perform this task was via *string-based* embedded queries (Listing 3.1 top). String-based queries were authored in SQL and parts of the query clauses were added or modified according to the context via `if` statements.

However, programming practice has departed from the traditional string-based construction of embedded queries and, *developers now employ certain reusable host language facilities (e.g., a specific API provided by the host language), to programmatically construct and execute the respective queries*. We call this way of query construction *object-based* as queries are formed as objects of the host language that are further manipulated by functions of an API that is responsible for the integration with the database. See Listing 3.1 for the construction of such a query; the query is represented by an object, under the variable `$query` and further modified by the host PHP code via calls to the methods of a database-related API.

The state of the art methods and tools on query extraction do not support a general, easily understood and language-independent method for the identification of embedded queries, especially when it comes to object-based ones (see Chapter 2). The current methods and tools work only in specific environments (e.g., Java, or C#) via translating the object-based queries to string-based ones, or examine only the queries that are most likely to be generated by the execution flow of the source code [53, 27].

To address these shortcomings, we propose a language-independent *method that is principled, customizable and is able to (a) identify the embedded queries of a data-intensive ecosystem, regardless of the programming style and the host language, as well as by finding all their variations due to branching or looping statements, and at the same time, (b) represent them in a universal, language - independent manner that can later facilitate migration or reconstruction, with (c) minimal user effort and significant effectiveness.*

Our method consists of four parts, depicted in Fig. 3.2. As discussed in Section 3.2, we start with source code files as input. Initially, we decompose the input files to their

```
$result = db_query('SELECT source, alias FROM {url_alias} WHERE source in (:system) AND
    language = :language_none ORDER BY pid asc;', $args);
```

```
/* Modified example of object-based embedded query. */
function _profile_get_fields($category,$register=FALSE) {
// Modification: addition of if statements.
 if (isEmpty($category)) {
  if(loggingIsEnabled) {
   log('Error: you did not provide any category.');
  }
  return;
 }
 else {
  $query = db_select('profile_field');
  if ($register) {
   $query->condition('register',1);
  }
  else {
   $query->condition('category',db_like($category),'LIKE');
  }
  while (!user_access('administer users')) {
  // Some comments
   $query->condition('visibility',PROFILE_HIDDEN,'<>');
  }
  return $query
   ->fields('profile_field')
   ->orderBy('category','ASC')
   ->orderBy('weight','ASC')
   ->execute();
 }
}
```

Figure 3.1: Embedded queries of Drupal-7.39; string (top) and object based (bottom)

structural parts (functions/methods) and we keep only these parts of the code that host queries. In the context of our language-independent approach, we uniformly will hereafter refer to functions/methods/procedures/routines as *Callable Units*. In general, a Callable Unit is: "a sequence of program instructions that perform a specific task,

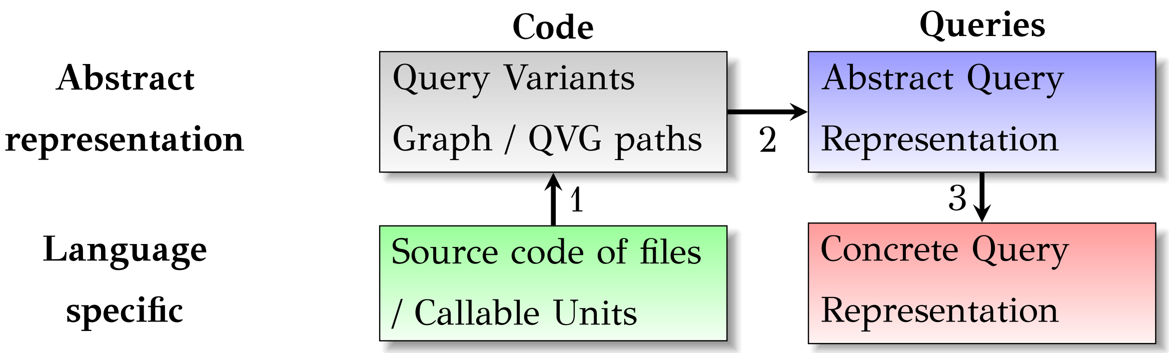

Code
Queries
Abstract representation
Language specific
Query Variants Graph / QVG paths
Abstract Query Representation
Source code of files / Callable Units
Concrete Query Representation
1
2
3

Figure 3.2: The steps of our method

packaged as a unit"[1]. For those Callable Units we create an abstract representation of their code that we call *Query Variants Graph (QVG)*. A QVG is a tree-like graph representation of a function/method that uses the database, where we consume the branch and loop statements of the host language. Due to the existence of branch and loop statements in the code, our next task is to traverse the Query Variants Graph and find every possible variation of a query that could occur at runtime. The result is a set of *QVG paths*, i.e., path traversals from the root of the QVG till one of its leafs. Observe that our representation has crossed the border of language specific details and is now language-independent, depending only on premises like Callable Units, and branch and loop statements that are practically universal. Our next step is the extraction of queries from the QVG paths and their representation into a generic, language-independent model. To represent queries in our model, we introduce an extensible pallet of *Abstract Data Manipulation Operators* with fundamental data transformation and filtering operators. This facilitates a universal representation of queries, independently of the source language (thus the need for extensibility). So, in Section 3.3 we present how queries are represented as combinations of these operators, via a model of representation which we call *Abstract Query Representation* (AQR). An AQR is a directed acyclic graph with nodes that describe the database-related parts of the code and its purpose is to formally represent the queries. Finally, we can exploit the Abstract Query Representation for various purposes, by converting the abstract representation to a specific, target language, a facility useful both for the understandability of the queries and for different kinds of migrations – e.g., either between database engines (e.g. from MySQL to Oracle) or to completely different environments, like MongoDB. This part is discussed in Section 3.4.

[1] https://en.wikipedia.org/wiki/Subroutine

Algorithm 3.1: Method overview using developer's input. For each of the Algorithms (3.3, 3.4, and 3.5) we mention the parts of the developer's input that is needed.

```
Input: Project's folder $pf$, Files to exclude $excl$, Beginners $begins$, API
        functions $apiF$, Language specifics $lspecs$
Output: The Abstract Query Representations of the project.
$paths = \emptyset$;
$files2search = Recursively\ search\ the\ project's\ folder$;
foreach $file\ f : files2search$ do
    if $f \in excl$ then
        $files2search- = f$;                    ▷ Remove files w/o queries
    end
end
$CallableUnits$ = split file into Callable Units;
foreach $CallableUnit \in CallableUnits$ do
    QVGs += create a Query Variants Graph (QVG) per Callable Unit;
end
foreach $QVG \in QVGs$ do
    $paths+ =$ to transform each QVG to a set of QVG paths;
end
foreach $path \in paths$ do
    $AQRs+ =$ extract the embedded queries into an Abstract Query
      Representation (AQR);
end
foreach $AQR \in AQRs$ do
    export $AQR$ to a target language;
end
```

In Figure 3.2 we present a macroscopic overview of our method and we can see that the parts of our method are related with the "concrete" and "abstract" representations of the source code (left) and database queries (right). The arrows describe the input and output of each step. Algorithm 3.1 describes those steps.

Concluding, with this method:

- we provide a customizable way of extracting queries from a specific source project,
- we provide a language-independent abstract representation of database-related source code,
- done with minimal effort and significant effectiveness.

We have tested our method with systems built in different source languages (PHP and C++) and achieve very high numbers of recall and correctness with quite low user effort.

In Section 3.2 we will see how we locate the functions/methods that contain database queries in their source code, and how we decompose those methods to Query Variants Graphs. Then, how we construct Abstract Query Representations from the Query Variants Graphs, so as to move from abstract code to abstract queries. Next in Section 3.3, we will see a universal representation of the queries which is used to create concrete queries for specific target environments (which is discussed in Section 3.4). Additionally, in Section 3.5 we present a cross layer method for the first two steps of our method. Finally, in Section 3.6 we present our experimental evaluation.

3.2 Source Code to Query Variants Graph

In this section, we address the problem of identifying all the variants of the queries that exist in the source code of a given information system. To do so, we initially abstract the input of this step, which is the source code of the information system, to a Query Variants Graph that removes the language-specific control statements such as branch and loop statements of the host language. Next, we generate every possible query variant via traversing the QVG paths. Thus, the result of this step is a set of QVG paths for every Callable Unit of the information system that generates a query.

3.2.1 Query Variants Graph Construction

In this subsection, we address the problem of abstracting the source code from the branch and loop statements that exist in the host language via an abstract representation (Query Variants Graph).

Our input is a *Project* that consists of a set of *Files* which may include functions or methods (depending on the host language). Both functions and methods are elementary building blocks of software, and, therefore we need to represent them in our analysis. Specifically, starting from a set of files that constitute the source code of an information system, our first step is to identify the database-related files and skip everything else. Then, we decompose these files to their Callable Units and we perform a second layer filtering keeping only the database-related Callable Units. The Callable Units shown at Listing 3.1 are database-related ones, since they are querying two database tables (url_alias and $profile\_field$ respectively). The Callable Units typically consist of branch and loop statements that interact with the objects of the Callable Units, and the query objects also get modified, therefore we also represent the branch and loop statements in our analysis too.

Extraction of Callable Units The first intermediate step towards abstracting the source code in language-independent format is the extraction of Callable Units. We initially check whether a file contains any database-related code statement either checking for query-related statements through string-based pattern matching or for query-related object initializations. If there is no such statement, we skip the file. Otherwise, we split it to its Callable Units. To end up working only with query-embedding Callable Units, significantly reducing the amount of work and resources needed to be invested in the subsequent steps, we perform a second layer filtering, at *Callable Units* level, since in the first filtering we only excluded the *files* that had no database connection. We do that in a second step because the Callable Unitsdecomposition step is time consuming so we avoid it for every file and we perform it only when necessary. The second layer filtering is done as the first layer filtering: via checking whether any of the code statements of the Callable Unit is database-related or not. Algorithm 3.2 formally describes those steps.

The price to pay To extract the appropriate information from the source files, we need to perform simple extractions from the source code. This requires (a) *physical level information* like the location of the source code and the parts of it that are to be ignored (e.g., binary files), (b) *query-related information* denoting the terms signifying a query, and, (c) *language specific information*.

Concerning the query-related information, as already mentioned, we discern between two categories of hosting. In the first case, where queries are handled as strings, we need to know the API functions that use that string, so as to perform slicing in

Algorithm 3.2: Callable Unit Extraction: extraction of database-related Callable Units of a project

```
Input: A set of source files, $F$, of the project;
Output: The database-related Callable Units ($U$);
Variables: The set of Callable Units of f ($f^{CU}$);
The set of Callable Units of the project, $U = \emptyset$;
forall files $f \in F$ do
    if $f$ does not call DB-related functions then
        continue;                    ▷ early prune for time measurement reasons
    end
    $f^{CU}$ is created by splitting f to Callable Units via the grammar's tokenizer;
    forall Callable Units: $cu \in f^{CU}$ do
        if $cu$ calls any DB-related function then
            $U \cup cu$;                 ▷ only DB-related Callable Units
        end
    end
end
```

order to find the query strings (in our example of Listing 3.1 the function contains the complete query string). In the second case, where queries are handled as objects and their definition is manipulated via a dedicated API for query construction, we need to know the API functions that construct an object-based query.

The way we do this is by splitting the original project to Callable Units on the basis of a formally specified grammar that requires the user to enter *once per language*: (i) how the comments start and end (both single-line and multiple-line comments), (ii) how the string values are described in the host language (e.g., in C++ this is done by using the character: '"'), (iii) if there are characters that "escape" the string value markers (eg. in C++ the character: '\'), (iv) finally how to treat the branch and loop statements of the host language. In this grammar, we treat nearly all loop statements similarly to branch statements. Remember that we are doing *static* analysis to dig out the query semantics. As loops are typically populating filters with values produced at runtime, we only need to handle the contents of the loop once, to identify the used expression along with the usage of an artificial set-valued pseudo-constant without practically misrepresenting the query's semantics.

Having the previously stated input, we can treat each source code file with the grammar that is described in Listing 1. We treat nearly all loop statements as branch statements with only one exception. That is the `do…while` loop that is executed at least once, therefore, when we encounter such a loop, we just add its statements to our representation (QVG).

Query Variants Graphs Having explained our meta-model for the input and the method for the extraction of its Callable Units, we now move on to describe the abstract representation of the code, which is the first step in Fig. 3.2.

Table 3.1 contains the host language block types that are used in this meta-model as input, with a description of what they represent in the host language. A block is used to describe the scope validity of the statements of the host language. Note that each block may contain internal blocks, e.g. loop blocks inside other loop blocks for a two dimensional array iteration. Therefore we decompose them to their parts. Also, since there can be code statements before and after an internal block, we split the code of the Callable Unitinto two parts: one before and another after the block we examine. Those blocks are used to create the nodes of the Query Variants Graph.

Our fundamental structure for abstracting a Callable Unit is the Query Variants Graph. A Query Variants Graph is a graph (almost a tree) with nodes the blocks of the source code, without branch and loop statements. The edges correspond to the control flow of the code (aka they "consume" the branch and loop statements). A formal definition of the Query Variants Graph is described in Definition 3.1.

Definition 3.1. Query Variants Graph - a directed rooted graph $QVG(V, E, r)$, where V is the set of nodes of the graph corresponding to elements of a Callable Unit, E, the set of directed edges connecting elements of the Callable Unit together, and r belongs to V is the root node, with the following properties:

1. The root of the graph corresponds to the entire Callable Unit CU.

2. Sibling nodes have the following properties:

    - they share the same code both among them and also with their parent, both before and after the branching/looping statement of their parent

    - each sibling replaces the branching/looping block (including the branch/loop statement) of their parent with exactly one alternative execution block

```
<Callable Unit> ::= <function type> <function name> '(' <parameters definition list> ')' '{' <
    block contents> '}'
<function type> ::= <chars> | e
<function name> ::= <chars>
<parameters definition list> ::= <parameter definition> | <parameters definition list> ',' <
    parameter definition>
<parameter definition> ::= <chars> | <chars> <chars> | e
<block contents> ::= <statement> | <block contents> ';' <statement>
<statement> ::= <string> | <for statement> | <foreach statement> | <while statement> | <do
    while statement> | <switch statement> | <if statement> | <function call> | e
<string> ::= <start string symbol><string content><end string symbol>
<for statement> ::= if ( <condition> ) <block>
<foreach statement> ::= if ( <condition> ) <block>
<while statement> ::= if ( <condition> ) <block>
<do while statement> :: <block>
<switch statement> ::= switch ( variable ) <case blocks>
<case blocks> ::= case <block> break | <case blocks> default <block> break | e
<if statement> ::= if '(' <condition> ')' <block> [ else <block> ]
<block> ::= <start block symbol> <block contents> <end block symbol>
<function call> ::= <function name> ( <function parameters> )
<function parameters> ::= <parameter> | <function parameters> <parameter>
<parameter> ::= <chars> | e
<chars> ::= <char> | <char><alnums>
<char> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" |
    "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "
    d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s
    " | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<alnums> ::= <alnum> | <alnum><alnums>
<alnum> ::= <char> | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | e
```

Listing 1: Host language grammar example for C++ programming language

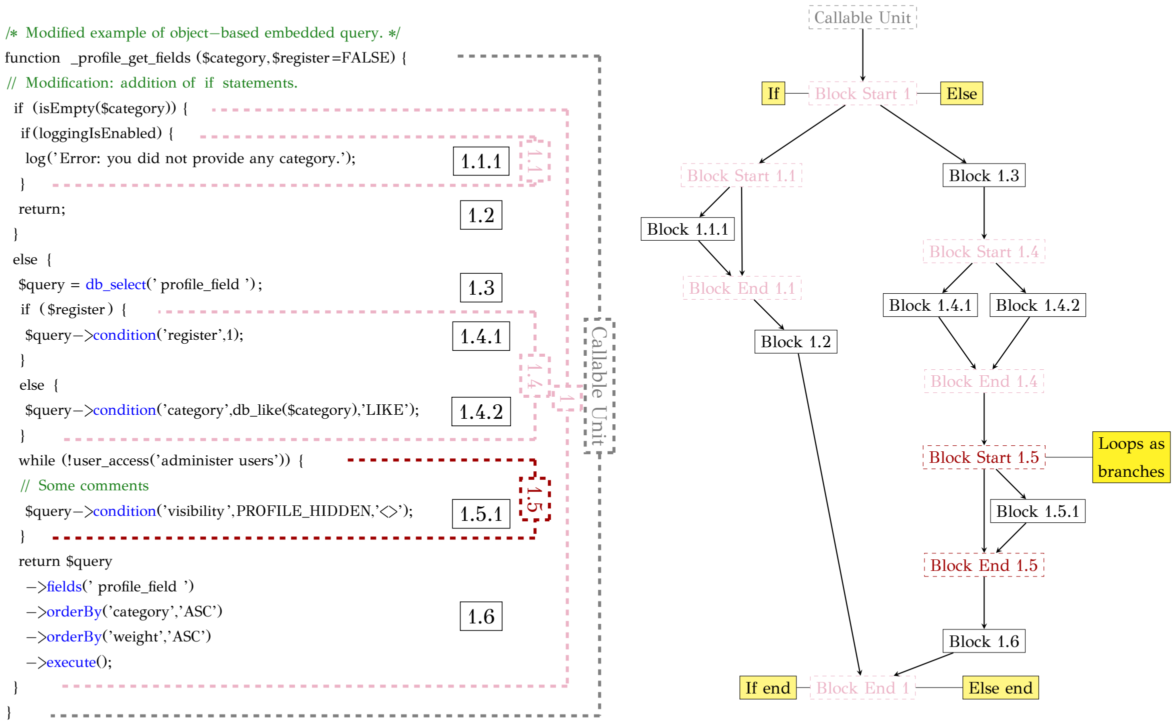
	Name	Description
	Loop	Loop blocks contain a group of statements that is repeated while a condition is met.
	Branch	Branch blocks contain statements that get executed after a condition is examined and is found true.
	QVGNode	Nodes that are used for the creation of the Query Variants Graph, and specifically are one of the following types: QVGNModule, QVGNStatements, QVGNStart, or QVGNEnd.
QVGNodes	QVGNModule	QVGNModule represents the beginning of a Callable Unit.
	QVGNStatements	QVGNStatements component contain the code statements that are to be executed the one after the other, without any branch or loop statement between them.
	QVGNStart	QVGNStart represents the beginning of a Branch or Loop block. As already mentioned, the Branch and Loop blocks lead to different Statement blocks. Even the simpler “if” statement, leads to two Statement blocks, one that contains the code that is to be executed when the condition is true, and another that does not contain that extra code.
	QVGNEnd	QVGNEnd represents where a Branch or Loop block terminates.

Table 3.1: Block types of host language, with their descriptions and components of Query Variants Graph

- for every alternative branching/looping block there is exactly one sibling node.

Algorithm 3.3 serves the creation of the Query Variants Graph tree. A Callable Unit is decomposed to its blocks, starting with the first branch or loop block. The code of that block is split to its components and each one of them becomes a “sibling” node of the QVG. After that, the remaining code is checked again for branch/loop blocks, and, of course, the “siblings” are checked for branch/loop blocks too.

In Fig. 3.3 we have the example of the Query Variants Graph for the object-based query presented in Listing 3.1. On the left side we have the source code denoted with the blocks of the code. On the right we have the graph representation of the Query Variants Graph.



```
/* Modified example of object−based embedded query. */
function  _profile_get_fields ($category,$register=FALSE) {
// Modification: addition of if statements.
 if (isEmpty($category)) {
  if(loggingIsEnabled) {
   log('Error: you did not provide any category.');
  }
  return;
 }
 else {
  $query = db_select(' profile_field ');
  if ($register) {
   $query−>condition('register',1);
  }
  else {
   $query−>condition('category',db_like($category),'LIKE');
  }
  while (!user_access('administer users')) {
  // Some comments
   $query−>condition('visibility',PROFILE_HIDDEN,'<>');
  }
  return $query
   −>fields(' profile_field ')
   −>orderBy('category','ASC')
   −>orderBy('weight','ASC')
   −>execute();
 }
}
```

(a) Object-based query, annotated with black horizontal labels on sequential blocks, and dashed vertical on loop and branch blocks.

(b) Query Variants Graph of modified reference example.

Figure 3.3: The reference example of Listing 3.1 (down) in two representations: a) text and b) graph.

Next, we describe the class diagram for the classes that get involved in the Callable Unit extraction as well as the Query Variants Graph construction. To represent a Callable Unit as a graph, we need to use nodes. These nodes represent the blocks of Table 3.1. In order to identify where an internal block begins and ends, we used the *Start* and *End* components of Table 3.1. Regarding the implementation aspect, all the nodes implement the abstract class $QVGNode$. The root of the QVG graph is the $QVGNModule$ class.

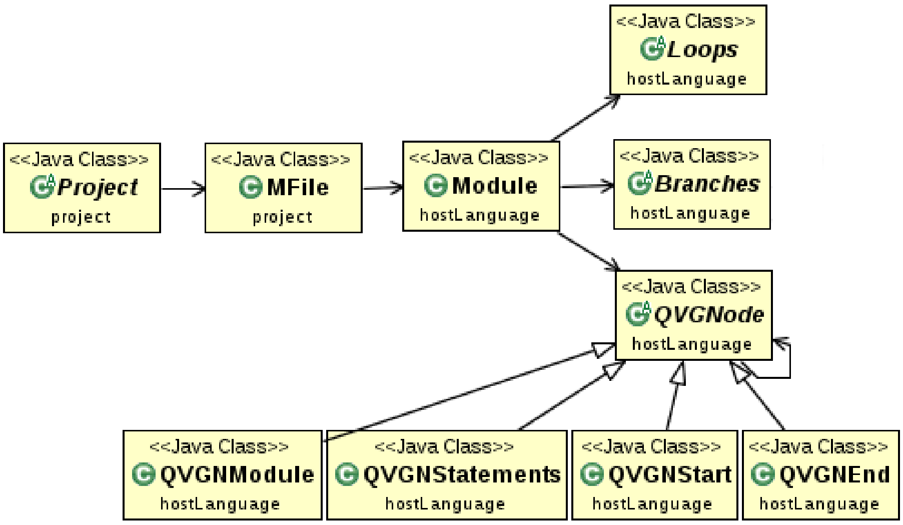


Figure 3.4: Host language class diagram: Loops are treated as branches. QVGNode is used to create the graph representation of Query Variants Graph as described in Definition 3.1

3.2.2 Query Variants Graph Path Identification

In this subsection we address the problem of identifying the different variants of a query that may occur during the execution of the code. This is done via a DFS-like algorithmic approach, where we traverse every Query Variants Graph path, regardless of whether the path contains database-related code or not. Algorithm 3.4 formally describes how we identify the variants of a query.

We perform a top-down traversal of the graph and we keep all code statements encountered from the root to each visited node, in a variable, called QP in our algorithm.

Initially, we start from the root node of the QVG (*CU.Block*), with an empty list of query variants (named $queryVariants$) and an empty "up to now" string statement (named $codeUpToNow$). For each node that we visit, we append in QP the code statements of the visited node. Then, we check if the visited node has any children nodes. If the node has no children nodes and QP is not empty, then we have finished with a traversal and we add the contents of QP to the $queryVariants$ list. The contents of QP are the code statements from the *CU.Block* node up to a "leaf" node of QVG. If the node we visited has children nodes, then for each one of them we recursively call the *TraversePaths* procedure, giving as starting node the child node that we want to visit, as "up to now" string statements the QP variable and as list, the $queryVariants$

list of paths.

The difference of *TraversePaths* procedure to the well known DFS algorithm is that we do not mark the nodes we visit. This is because we may encounter a node in more than one traversals, coming from different ancestor nodes, thus having different $codeUpToNow$ value, which is the information that is kept in a node. So, marking a node as visited would produce wrong results. Observe the Query Variants Graph of Figure 3.3b: the bottom *Block 4* node is used in four different traversals, marking it as visited after the first traversal would result in ignoring its statements in the remaining three traversals.

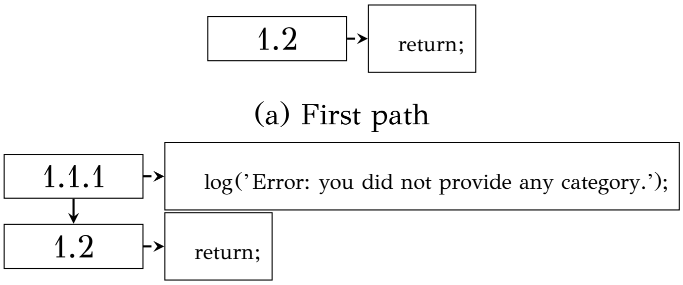
Coming back to our reference example, we can see that the Query Variants Graph of Figure 3.3b provides six different traversals. The traversals we are interested in are the ones that contain non dashed boxes, since the dashed nodes are used for auxiliary purposes, and they do not contain any project related statements as their attributes. The six different traversals of Algorithm 3.4 for the Query Variants Graph of $\_profile\_get\_fields$ Callable Unit are displayed in Figures 3.5a, 3.5b, 3.6a, 3.6b, 3.6c and 3.6d. Each figure contains the traversal as graph (with its contents on the right side) that are used in the abstract representation described later in Section 3.3.

The first one uses only the *1.2* node, and it is described by Figure 3.5a.

The second one, uses –in addition to *1.2* node– the *1.1.1* node, and is described by Figure 3.5b.

An observant reader may notice that the previous two traversals of Figure 3.5 do not contain any kind of database interaction. The statements of the nodes participating in those traversals are for log purposes and termination of $\_profile\_get\_fields$ Callable Unit. Therefore, since we are interested only in the database-related Callable Units, those traversals are considered as false positive results in our research. To avoid having such results, our method initially included a check point of the node's statements whether the node contains database-related code or not. What we noticed in our efficiency evaluation regarding the time measurement is that this kind of check at *TraversePaths* method, slowed down our method up to 4 times compared to the presented approach of non checking! Therefore, we add those traversals to our list, and we deal with them later on, in Section 3.3.

Returning to our example, we move on to the third traversal of Algorithm 3.4. This traversal uses the *1.4.1* node and does not use the *1.5.1* node. The *1.3* and *1.6* nodes are also used in this traversal. The code depicted in Fig. 3.6a describes the



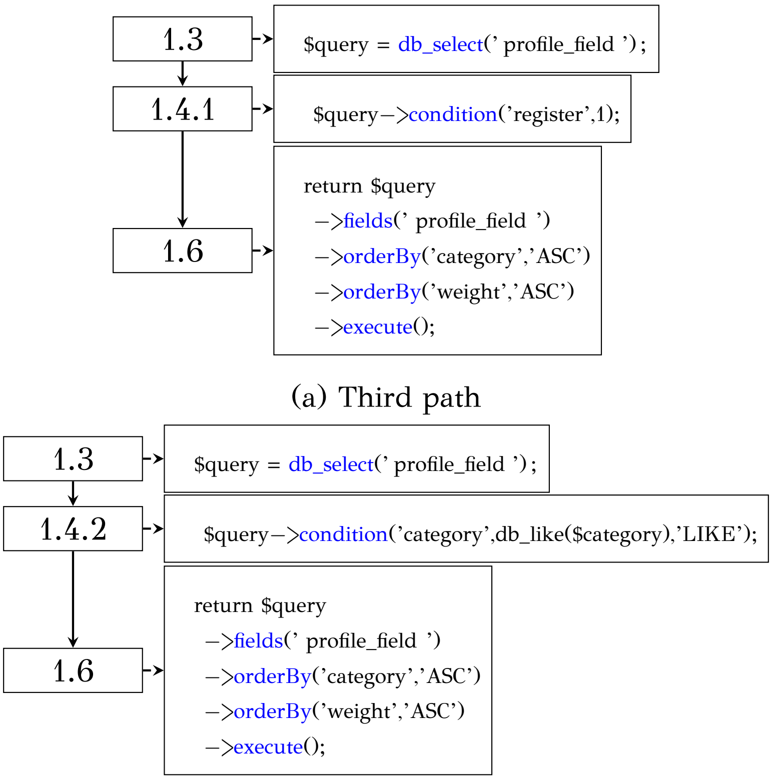
```
return;
```

(a) First path

```
log('Error: you did not provide any category.');
return;
```

(b) Second path

Figure 3.5: Execution paths of our object-based reference example of Listing 3.1 that are not database-related. The paths are described by "next" arrows (non dashed arrows) that start from the first non dashed node and move to the final one. The dashed nodes do not provide any project related statements for our execution path representation, therefore we omit them from the path representation. The dashed arrows are there to describe the content relationship between the nodes and the project source code statements.

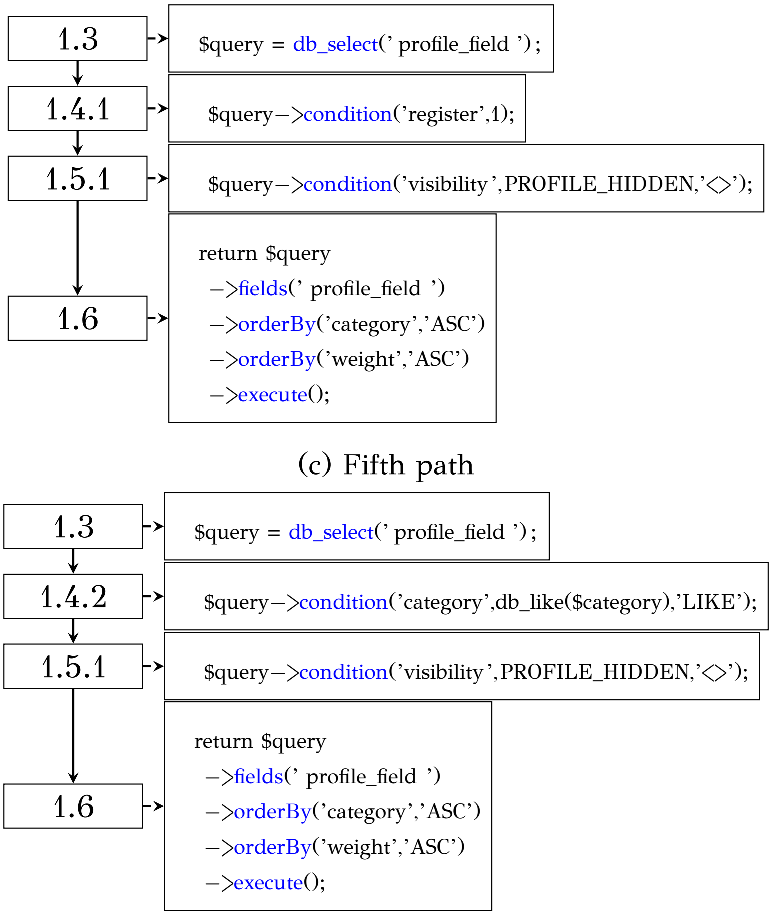


```
$query = db_select(' profile_field ');
$query−>condition('register',1);
return $query
  −>fields(' profile_field ')
  −>orderBy('category','ASC')
  −>orderBy('weight','ASC')
  −>execute();
```

(a) Third path

```
$query = db_select(' profile_field ');
$query−>condition('category',db_like($category),'LIKE');
return $query
  −>fields(' profile_field ')
  −>orderBy('category','ASC')
  −>orderBy('weight','ASC')
  −>execute();
```

(b) Fourth path



```
$query = db_select(' profile_field ');
```

```
$query->condition('register',1);
```

```
$query->condition('visibility',PROFILE_HIDDEN,'<>');
```

```
return $query
  ->fields(' profile_field ')
  ->orderBy('category','ASC')
  ->orderBy('weight','ASC')
  ->execute();
```

(c) Fifth path

```
$query = db_select(' profile_field ');
```

```
$query->condition('category',db_like($category),'LIKE');
```

```
$query->condition('visibility',PROFILE_HIDDEN,'<>');
```

```
return $query
  ->fields(' profile_field ')
  ->orderBy('category','ASC')
  ->orderBy('weight','ASC')
  ->execute();
```

(d) Sixth path

Figure 3.6: Execution paths of our object-based reference example of Listing 3.1 that are database-related.

database-related code that is to be executed from this traversal during runtime.

The fourth traversal differs to the previous one only in one place: instead of *1.4.2* node, this traversal uses the *1.4.2* node. The code of the fourth path is depicted in Fig. 3.6b.

The next two traversals of Query Variants Graph of $\_profile\_get\_fields$ Callable Unit are based on the previous two traversals, but now they both additionally use the *1.5.1* node that was previously excluded from the traversals. Their codes are depicted in Fig. 3.6c and Fig. 3.6d, respectively. The previous four traversals of Figure 3.6 are all database-related and provide four different variants for the database query of $\_profile\_get\_fields$ Callable Unit.

Concluding, the Algorithm 3.4 added a new variant to the list of query variants ($queryVariants$) for the examined Callable Unit ($\_profile\_get\_fields$), for each one of the previous six traversals of Figures 3.5 and 3.6.

The next step in our method is to use those variants, and to create a universal representation for each one of the database-related query variants. This way, we can compare the queries and find duplicates, we can also notice queries that are frequently used in different files/Callable Units, and finally we can use that universal representation to translate the query variants to more than one target languages (e.g. such as SQL for a specific DBMS, or MongoDB).

Algorithm 3.3: Creation of Query Variants Graph

```
Input: A Callable Unit ($CU$)
Output: The root node for the Query Variants Graph of $CU$ Callable Unit's
        source code (along with the rest of the tree that is constructed).
$Block = new\ node$;
$Block =$CreateGraph($CU$, $Block$);
Procedure CreateGraph($CU$, $Parent$)
    $Block$ = new node;
    $branches$ = code of the first branch/loop block;
    if $branches \neq \emptyset$ then
        if $branches \neq contain$ final $alternative$ then
            $branches$ += empty branch statement;
        end
        $BlockStart$ = new node;
        $preceding$ = code before the start of first branch block;
        if $preceding \neq \emptyset$ then
            $Block = preceding$;
            link $BlockStart$ to $Block$;
            link $Block$ to $Parent$;
        else
            link $BlockStart$ to $Parent$;
        end
        $BlockEnd$ = new node;
        foreach $sibling \in branches$ do
            link $BlockEnd$ to CreateGraph($sibling$, $BlockStart$);
            remove examined code;
        end
        return CreateGraph($M$, $BlockEnd$);              ▷ Code after 1st branch
    else
        $Block = all\ CU\ code$;                          ▷ Block without branch/loop
        link $Block$ to $Parent$;
        return $Block$;
    end
```

Algorithm 3.4: Creation of QVG paths for a Callable Unit CU

```
Input: A Callable Unit ($CU$)
Output: The database-related QVG paths of a Callable Unit ($queryVariants$).
Variables: $queryVariants = \emptyset$, $codeUpToNow = \emptyset$;
TraversePaths($CU.Block$, $codeUpToNow$, $queryVariants$);
Procedure TraversePaths($v$, $codeUpToNow$, $queryVariants$)
    $QP = codeUpToNow + statements\ of\ v$;
    if $v$ has no children then
        if $QP \neq \emptyset$ then
            $queryVariants + = QP$;
        end
    else
        forall $w : children\ of\ v$ do
            TraversePaths($w$, $QP$, $queryVariants$);
        end
    end
```

3.3 From QVG Paths to Abstract Query Representations

In this section, we introduce a universal way to represent the query variants that we obtained from the QVG traversals. Moreover, since this is an abstract query representation, it should be able to describe any database query, despite of how it was created (object-based or string-based queries).

To represent queries, we use an extensible pallet of Abstract Data Manipulation Operators ($ADMO$) that represent the different parts of a query. Our operators cover the relational algebra, therefore we are able to represent queries embedded in relational database management systems. The operators are given in Table 3.2. The Abstract Data Manipulation Operator pallet is extensible; new operators can be added to cover cases of non relational databases.

So, to represent a query, we need: (i) to tokenize the string query to query parts (which is a straight forward procedure), when the string-based creation method is used, (ii) to identify the parts of the source code that pertain to the specified query object variable (i.e. in Fig. 3.3a the $\$query$ object variable), when the object-based creation method is used. For both cases we use the Abstract Data Manipulation Operators to represent the queries.

Algorithm 3.5 formally describes the AQR construction from QVG paths. For the string-based constructed queries, the mapping of the SQL parts to the AQR nodes is a straightforward procedure. Using as reference example of Listing 3.1 we tokenize the first parameter of *db_query* function (which is our input) to the parts that are between the keywords of SQL query language (the capitalized words with *blue* color). Then, we add Projector operators for each of the values that follow the SELECT keyword as a parameter to each node. We add a Source operator for the value that follows the FROM keyword, with its parameter (*url_alias* in our example). We add comparator operators (with their parameters) for the values that follow the WHERE, and AND keywrods. Finally, we add an ordering operator (with its parameters) for the value that follows the ORDER BY keyword. Table 3.2 describes all possible keyword - ADMO combinations for the SQL queries.

In Definition 3.2, we have a formal way to describe our Abstract Query Representation.

Definition 3.2. Abstract Query Representation (AQR) - An abstract query representation $AGR = (V, E)$ is a directed acyclic graph whose nodes, V, are Abstract

Source	Describes a provider of information in a query	A table in SQL.
Projector	Describes an output attribute	The SELECT attributes in SQL.
Comparator	Describes a filter that the output of the query should fulfil	The conditions of the WHERE clause in SQL.
Grouper	Used for summarizing of the output (used for grouping the incoming data in groups, each group identified by a unique combination of grouper values)	The attributes of the GROUP BY clause in SQL.
Ordering	Used for sorting of the output	The attributes of the ORDER BY clause in SQL.
Limiter	Used for restricting the size of the output	The TOP/LIMIT clauses of an SQL query
Aggregator	Used for applying an aggregate function to a input attributes	The MIN, MAX, COUNT, SUM, AVG functions in SQL

Table 3.2: Abstract Data Manipulation Operator with a description of the part of a query that they represent

Data Manipulation Operators that describe a part of the query. An edge $e \in E$ from a node v_i to a node v_j specifies that the execution of the statement represented by v_i precedes the execution of the statement represented by v_j. The set of nodes $V = \mathit{Start} \cup Nodes \cup \mathit{End}$, is a union that comprises the following nodes:

- A node Start that specifies the beginning of a query variant q.
- A set of nodes $Nodes$ that represent Abstract Data Manipulation Operators which serve for generating the different parts of the query variant q. Each one of the nodes is an Abstract Data Manipulation Operator (ADMO) as described in Table 3.2.
- A node End that serves for concluding the generation of q.

Observe that since a string-based query might be modified in the source code, we may need to perform slicing (forward slicing, as mentioned in [91]) to find out

Algorithm 3.5: Transforming a QVG path to its AQR representations

```
Input: A QVG path of a Callable Unit ($P$), a mapping ($M$) of the API
       functions to ADMOs
Output: The Abstract Query Representation of $P$.
Add Start node for AQR ;
foreach $QVGNode\ N \in P.nodes$ do
    $functionsOfNode$ = split contents of $N$ to its functions;
    foreach $F \in functionsOfNode$ do
        $FAMDOs = M(F)$;                ▷ Find the ADMO nodes for function $F$
        foreach $fadmo \in FAMDOs$ do
            Set function's $F$ parameters to $fadmo$'s ADMO parameters;
            Add $fadmo$ to AQR ;
        end
    end
end
Add End node for Abstract Query Representation ;
```

whether our query was modified or not (in our example it is not happening). In our approach, we perform slicing only on the code of the Callable Unit that we examine. Inter slicing techniques that use dependency graphs to identify the parts of the queries that are constructed in other Callable Units that use the one that we examine (e.g., see [92]) have not proved to be necessary in our experiments; of course, they are a clear extension for future work.

In the case of object-based constructed queries, we need some additional input in order to construct the AQR out of the variants we obtained from Algorithm 3.4. We initially retrieve the contents of the variants and we decompose the statements of those variants to the API functions of the project we examine, as we need to map the functions of the project's API to the Abstract Data Manipulation Operators of Table 3.2. This is work performed exactly once, and it is project-related (since each project has it's own API). In Section 3.6 we discuss the developer's effort for this task.

In Listing 2 we can see how the source code statements of the object-based Query Variants Graph paths and string-based queries are transformed to Abstract Query Representations.

```
<AQR> ::= <object based> | <string based>
<object based> ::= <function call list>
<function call list> ::= <function call> |
   <function call list> delimiter <function call>
<function call> ::= <function name> ( <function parameters> )
<function name> ::= mapping of function to <ADMO nodes list>
<function parameters list> ::= <function parameter> |
   <function parameters list> delimiter <function parameter>
<function parameter> ::= <parameter>
<ADMO nodes list> ::= <ADMO node> | <ADMO nodes list> <ADMO node>
<ADMO node> ::= <Source> | <Projector> | <Comparator> | <Grouper> | <Ordering> | <Limiter> | <
    Aggregator>
<Source> ::= Source with a mapping of parameter to <ADMO parameter>
<Projector> ::= Projector with a mapping of parameter to <ADMO parameter>
<Comparator> ::= Comparator with a mapping of parameters to <par1><operand><par2>
<Grouper> ::= Grouper with a mapping of parameter to <ADMO parameter>
<Ordering> ::= Ordering with a mapping of parameter to <ADMO parameter> [ <ordinance> ]
<Limiter> ::= Limiter with a mapping of parameter to <ADMO parameter>
<Aggregator> ::= Aggregator <aggr function> with a mapping of parameter to <ADMO parameter>
<operand> ::= < | > | <> | != | = | <= | >=
<par1> ::= <par>
<par2> ::= <par>
<par> ::= <ADMO parameter> | <Aggregator>
<ordinance> ::= ASC | DESC
<aggr function> ::= "MIN" | "MAX" | "COUNT" | "SUM" | "AVERAGE"
<ADMO parameter> ::= <parameter>
<parameter> ::= <chars>
<chars> ::= <char> | <char><alnums>
<char> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" |
    "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "
    d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s
    " | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<alnums> ::= <alnum> | <alnum><alnums>
<alnum> ::= <char> | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | e
<string based> ::= <SELECT> <FROM>
   [ <WHERE> <GROUP BY> <ORDER BY> <HAVING> <LIMITend> ] |
   <SELECTwithLimitbegin> <FROM>
   [ <WHERE> <GROUP BY> <ORDER BY> <HAVING> ]
<SELECT> ::= " SELECT " <SELECT params>
<SELECT params> ::= <SELECT param> | <SELECT params><SELECT param>","
```

```
<SELECT param> ::= <Projector>
<FROM> ::= " FROM " <FROM parms>
<FROM params> ::= <FROM param> | <FROM params><FROM param>","
<FROM param> ::= <Source>
<WHERE> ::= " WHERE " <WHERE parms>
<WHERE params> ::= <WHERE param> | <WHERE params><CONNECTOR><WHERE param>
<CONNECTOR> ::= " AND " | " OR "
<WHERE param> ::= <Comparator>
<GROUP BY> ::= " GROUP BY " <GROUP BY params>
<GROUP BY params> ::= <GROUP BY param> |
   <GROUP BY params><GROUP BY param>","
<GROUP BY param> ::= <Grouper>
<ORDER BY> ::= " ORDER BY " <ORDER BY params>
<ORDER BY params> ::= <ORDER BY param> |
   <ORDER BY params><ORDER BY param>","
<ORDER BY param> ::= <Ordering>
<LIMITend> ::= " LIMIT " <LIMIT param>
<LIMIT param> ::= <Limiter>
<LIMITbegin> ::= " SELECT TOP(" <LIMIT param> ")" <SELECTwithLimitbegin>
<LIMIT param> ::= <Limiter>
<SELECTwithLimitbegin> ::= <SELECT params>
<HAVING> ::= " HAVING " <HAVING params>
<HAVING params> ::= <HAVING param> | <HAVING params><HAVING param>","
<HAVING param> ::= <Aggregator>
```

Listing 2: Abstract Query Representation grammar (with *italics* in the parts that need user input

Finally, in Figure 3.7 we see the creation of an AQR that comes from the first traversal of $\_profile\_get\_fields$ Callable Unit (which comes from an object-based Query Variants Graph path of Fig. 3.6a that was presented in Section 3.2.2). The project's API functions are translated to Abstract Data Manipulation Operators.

The AQR representation allows us to compare queries on the similarity of their structure. That is useful because from the Query Variants Graph traversals we might obtain query variants (in the Callable Units that we examine) with identical structure (albeit, possibly with different values). Since we consider all query variants as valid for our research, we need to identify the duplicate representations of such query variants here (these duplicates are due to branch/loop blocks in the source code of a Callable Unit that are unrelated to the query-object). Therefore, we use the Abstract

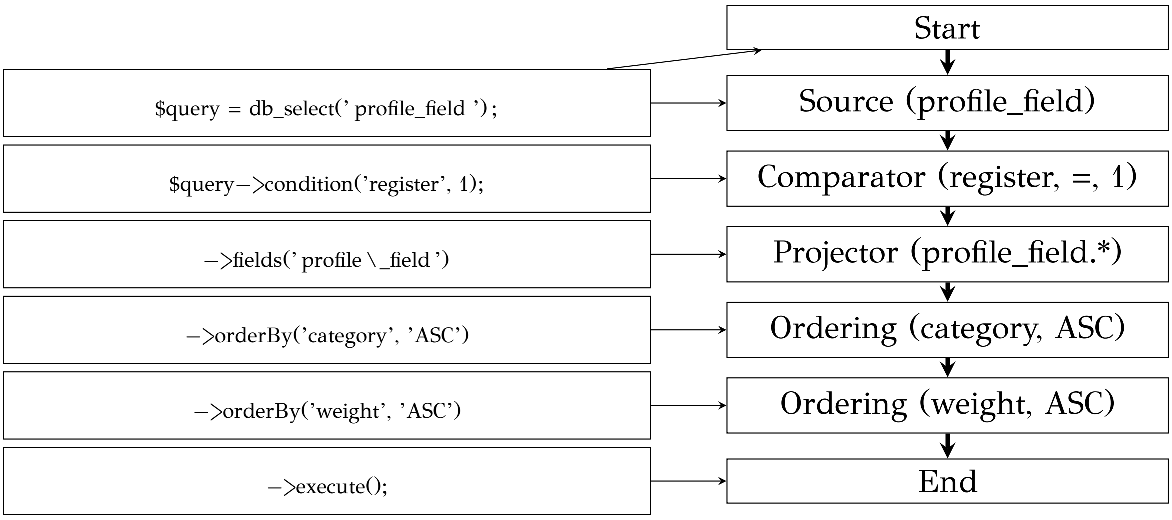
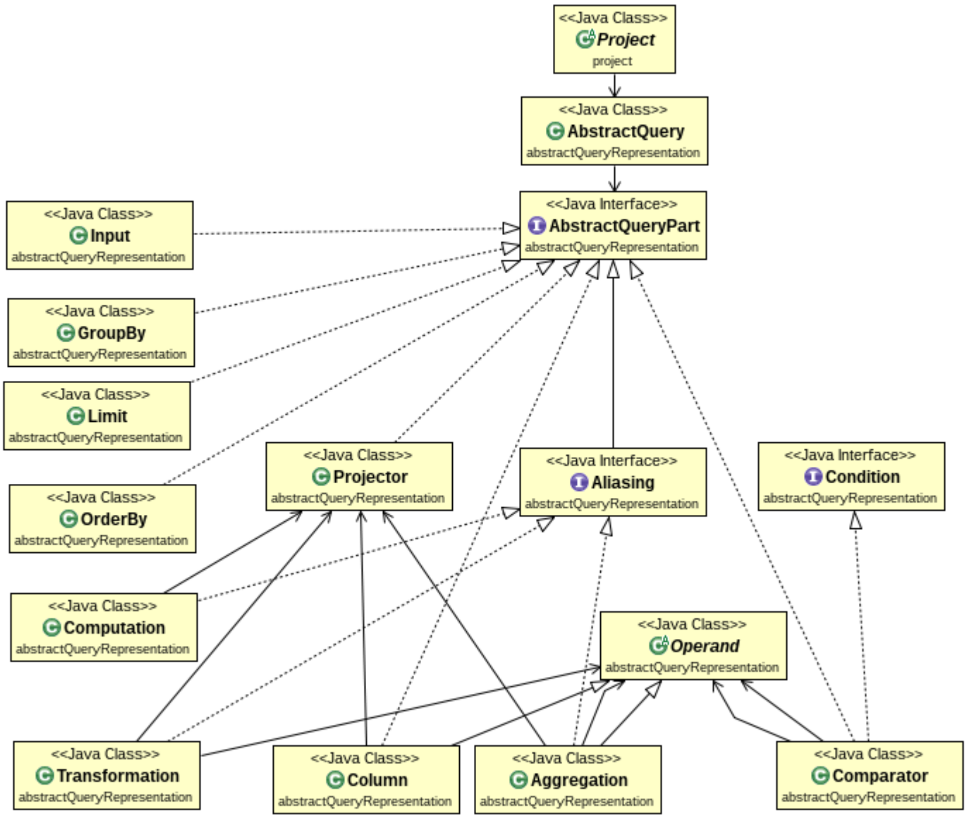

$query = db_select(' profile_field ');
$query->condition('register', 1);
->fields(' profile _field ')
->orderBy('category', 'ASC')
->orderBy('weight', 'ASC')
->execute();
Start
Source (profile_field)
Comparator (register, =, 1)
Projector (profile_field.*)
Ordering (category, ASC)
Ordering (weight, ASC)
End

Figure 3.7: Abstract query representation of the path presented in Fig. 3.6a. On the left we have the source code that constracts the query and on the right we have the AQR nodes with their parameters.

Query Representation, and see if there are any AQRs with the exact same operators, in the exact same order, carrying the exact same ADMO parameters in the same position of the Callable Unit. Since we need only one of those queries, we eliminate the AQR duplicates. This is a rather simple task, since a simple walk over the Abstract Data Manipulation Operators of the Abstract Query Representation can provide us the information needed for the comparison.

3.4 From Abstract Query Representations to Concrete Query Representations

The Abstract Query Representation would be of small use, if we could not translate the AQRs to concrete queries for a specific query environment, so, the next step of our method is to be able to transform the model representation of AQR to a text-based representation of a concrete query environment. In Figure 3.8 we depict the classes that are responsible for the creation of the Abstract Query Representation and how the AQR is connected to the project class. Each query consists of a number of query parts. Regarding the output of a query, we use the Projector class that is used by the Computation, Transformation, Column, and Aggregation classes. That is because a query might output the data that unmodified, or transformed (using simple expressions, or combined with aggregation functions etc.). Another issue is that one might also set a new name for an output attribute of a query via an alias identifier, therefore some of the classes that manipulate the data also need to implement the *Aliasing* interface. The query environments on which we have up to now performed this model-to-text transformation are SQL (Section 3.4.1) and MongoDB (Section 3.4.2).



<<Java Class>>
Project
project
<<Java Class>>
AbstractQuery
abstractQueryRepresentation
<<Java Interface>>
AbstractQueryPart
abstractQueryRepresentation
<<Java Class>>
Input
abstractQueryRepresentation
<<Java Class>>
GroupBy
abstractQueryRepresentation
<<Java Class>>
Limit
abstractQueryRepresentation
<<Java Class>>
OrderBy
abstractQueryRepresentation
<<Java Class>>
Projector
abstractQueryRepresentation
<<Java Interface>>
Aliasing
abstractQueryRepresentation
<<Java Interface>>
Condition
abstractQueryRepresentation
<<Java Class>>
Computation
abstractQueryRepresentation
<<Java Class>>
Operand
abstractQueryRepresentation
<<Java Class>>
Transformation
abstractQueryRepresentation
<<Java Class>>
Column
abstractQueryRepresentation
<<Java Class>>
Aggregation
abstractQueryRepresentation
<<Java Class>>
Comparator
abstractQueryRepresentation

Figure 3.8: Class diagram of classes that are related with the Abstract Query Representation and their connection to the database-related project we examine. Since the pallet is extensible, one may add other classes that only need to implement the AbstractQueryPart interface.

3.4.1 From AQR to SQL

In this part, we are going to describe the way we translate the AQRs to SQL queries.

Initially, we gather the nodes of the AQR in groups. Those groups are:

- The "projection" group: for the Projector and Aggregator Abstract Data Manipulation Operators, described in Table 3.2

```
SELECT <the contents of the projection group, separated with ','> or '*' if no such node exists
FROM <the contents of the input group, separated with ','>
{WHERE <the contents of the filter group, separated with ' AND '> or nothing if no such node
    exists}
{GROUP BY <the contents of the group by group, separated with ','> or nothing if no such node
    exists}
{HAVING <the contents of the having group, separated with ','> or nothing if no such node
    exists}
{ORDER BY <the contents of the order by group, separated with ','> or nothing if no such node
    exists}
{LIMIT <the contents of the limit group, separated with ','> or nothing if no such node exists}
;
```

Listing 3: SQL description

- The "input" group: for the Source Abstract Data Manipulation Operators, described in Table 3.2

- The "group by" group: for the Grouper Abstract Data Manipulation Operators, described in Table 3.2

- The "order by" group: for the Ordering Abstract Data Manipulation Operators, described in Table 3.2

- The "limit" group: for the Limiter Abstract Data Manipulation Operators, described in Table 3.2

- The "having" group: for the Comparator Abstract Data Manipulation Operators, described in Table 3.2, when the Comparator performs a comparison of an Aggregation function of SQL (MIN, MAX, COUNT, SUM, AVG)

- The "filter" group: for the Comparator Abstract Data Manipulation Operators, described in Table 3.2 (the remaining ones).

Those groups are used so as to export the Abstract Query Representation to a file, using the description that is depicted in Listing 3. The groups that are embraced with '{' and '}' characters are optional for the creation of the SQL queries, therefore if there are no Abstract Data Manipulation Operator nodes in these groups of AQR, these groups are omitted from the output.

3.4.2 From AQR to MongoDB

Likewise, we gather the nodes to groups in MongoDB export too. Those groups are:

- The "projection" group: for the Projector Abstract Data Manipulation Operators, described in Table 3.2
- The "input" group: for the Source Abstract Data Manipulation Operators, described in Table 3.2
- The "group by" group: for the Grouper Abstract Data Manipulation Operators, described in Table 3.2
- The "order by" group: for the Ordering Abstract Data Manipulation Operators, described in Table 3.2
- The "limit" group: for the Limiter Abstract Data Manipulation Operators, described in Table 3.2
- The "filter" group: for the Comparator Abstract Data Manipulation Operators, described in Table 3.2
- The "having" group: for the Comparator Abstract Data Manipulation Operators, described in Table 3.2, that are using one of the following functions: *sum*, *avg*, *min*, and, *max*
- The "aggregation" group: for the Aggregator Abstract Data Manipulation Operators, described in Table 3.2

As previously, those groups are used to export the Abstract Query Representation to a file. In MongoDB there are two ways (using `find` and `aggregate` functions) of asking a query, of which one (the `aggregate`) is subdivided to two versions. Therefore we need three descriptions on how to ask MongoDB queries. Those ways are presented in: Listing 4, Listing 5, and Listing 6. The final export depends on the following:

1. If the *input* group has more than one items, then it can not be exported to MongoDB.
2. If all the *group by*, *order by*, *limit*, and, *having* groups are empty, then the output form that is used is described in Listing 4.

3. If the *group by* group of nodes is not empty, then we append all the nodes of the *aggregate* group to the *group by* group, and the output form that is used is described in Listing 5.

4. If the *group by* group of nodes is empty, then the output form that is used is described in Listing 6.

As previously stated, the groups that are embraced with '{' and '}' characters are optional. Although one may observe that there could be empty queries, there is no such case since there is at least one group of Abstract Query Representation that contains some Abstract Data Manipulation Operator nodes.

```
db.<input value>.find(
{ <the contents of filter group> },
{ <the contents of projection and aggregation groups> }
)
```

Listing 4: MongoDB form description when none of the GROUPER, ORDERING, LIMITER or COMPARATOR (using any of the SUM, AVG, MIN, and, MAX functions) nodes is used

```
db.<input value>.aggregate( [
{ $match: <the contents of filter and having groups> }, // if exists
{ $group: { _id: <the contents of group by group> } },
{ $sort: <the contents of order by group> }, // if exists
{ $limit: <the contents of the limit group> } // if exists
] )
```

Listing 5: MongoDB form description when the GROUPER nodes exist in the AQR representation

```
db.<input value>.aggregate( [
{ <the contents of aggregation group> }, // if exists
{ $match: <the contents of filter and having groups> }, // if exists
{ $sort: <the contents of order by group> }, // if exists
{ $limit: <the contents of the limit group> } // if exists
] )
```

Listing 6: MongoDB form description when at least one of the ORDERING, LIMITER or COMPARATOR (using any of the SUM, AVG, MIN, and, MAX functions) nodes is used

A formal description of Abstract Query Representation expression to MongoDB expression is described in Algorithm 3.6.

In cases where a query can not get exported from AQR to MongoDB, we produce a message stating the Callable Unitand the file which contain the examined query that we failed to export. This mechanism exists in SQL too, but it is of no use in our evaluation since both the projects we examined have their queries written in SQL query language.

Algorithm 3.6: Abstract Query Representation to MongoDB representation

```
Input: An Abstract Query Representation ($AQR$) of a path of a function.
Output: MongoDb query.
$projection = \emptyset$, $input = \emptyset$, $groupby = \emptyset$, $orderby = \emptyset$;
$limit = \emptyset$, $filter = \emptyset$, $having = \emptyset$, $aggregation = \emptyset$;    ▷ Initialize node groups
foreach $Node\ N \in AQR$ do
    switch $N.type$ do
        case ”Comparator” do
            if Any of N parameters contain Aggregate function then
                Add $N$ to $having$;
            else
                Add $N$ to $filter$;
            end
        end
        case ... do
            Add $N$ to its node group;
        end
    end
end
if $input.size > 1$ then
    return(”Error”);
end
if $groupby == \emptyset$ AND $orderby == \emptyset$ AND $limit == \emptyset$ AND $having == \emptyset$ then
    Use Listing 4 output;
end
if $groupby! = \emptyset$ then
    $groupby+ = aggregate$;
    Use Listing 5 output;
end
if $groupby == \emptyset$ then
    Use Listing 6 output;
end
```

3.5 Cross-layer method: from source code to execution paths

As the reader may have already noticed, the creation of the graph is a procedure where for each block of code you need to keep its beginning and end. One difficult part of the graph creation procedure is to make all the needed connections between the code blocks, in order to produce every possible execution path.

In this algorithmic approach we will construct the *output* of the Query Variants Graph (aka the execution paths) without the need of the Query Variants Graph. After all, what we needed from the Query Variants Graph were the execution paths that describe the different variants of a query located at a Callable Unit's source code. Moreover, this algorithmic approach (Algorithm 3.7) is easily parallelizable, thus it produces a slightly faster result, and due to "pruning" that we perform in the database unrelated parts, we also need less main memory too (in Section 3.6 we will see more).

As previously, the Callable Unit that we examine is checked for branch and loop blocks (as already mentioned, we treat –nearly all– the loop blocks as if they were branch blocks). In this approach, the *first* occurrence of those code blocks is split into the parts that construct it. For each one of those construction parts, we prepend the sequential code blocks that existed till the appearance of that *first* block and we append the code that follows that branch/loop block. This way, we produce a new set of Callable Units. In the next step, we take this new set of Callable Units and for each one of those Callable Units we perform the same procedure: we search for the *first* branch/loop block and we split it to its construction code parts, pre-pending the code that existed and appending the code that follows. At the end of the examined code (when we have no more branch/loop blocks), we have a set of Callable Units that is an execution path of the initial Callable Unit.

Both algorithms (Algorithm 3.4 and Algorithm 3.7) can be memory and time demanding if there exist many branch and/or loop statements in the code of a Callable Unit. In order to speed up our procedure and to reduce memory consumption (since we want to find only the query-related Callable Units), in Algorithm 3.7 we prune as soon as possible the Callable Units that are not database-related. This is done by checking the variant of the source code right away if it is database-related. Moreover, we check the code that follows a branch statement to see if it has any database-related functions. If none such function exists in the remaining source code it means that we already have all the database-related code that we want, so we stop examining the source code of the Callable Unit for further branch and loop statements.

In Algorithm 3.7 we have a formal definition of our algorithm, where we use the *CreateQueryVariants* procedure which is responsible:

- for the recognition of DB related Callable Units, and,
- for the identification of the first branch statement of a Callable Unit in order to create as many different "children" Callable Units as the conditions of this first branch (the "children" Callable Units are equivalent to the "sibling" nodes of the Query Variants Graph).

The steps of the creation of the "variant" Callable Units contain host language dependent parts which can be summarized in the following:

1. find the first branch/loop statement keyword,
2. find all different conditions of this branch statement (as already mentioned, loop statements are treated as branch statements with two different conditions: either not run at all, or run until a condition is met –exception is the `do…while` loop that runs at least once),
3. for each of those conditions create a different Callable Unit, if and only if the Callable Unit contains database-related text (*vertical prune*),
4. in each of those Callable Units, prepend the code that existed before the branch statement,
5. in each of those Callable Units, append the code that exists after the branch statement, if and only if it is database-related (*horizontal prune*).

This way, a new set of Callable Units is created, based on the original one. The benefit of this approach is that we can prune non database-related paths as soon as possible, saving both execution time, from the unnecessary traversals of Query Variants Graph paths, and main memory consumption from Query Variants Graph representation. Moreover, due to the parallelization of this method we gain some extra time as we will see in Section 3.6.

Returning to our reference example, Figure 3.5 (which the Algorithm 3.4 was unable to identify) are pruned. To be more precise: using Algorithm 3.7 the Fig. 3.5b execution path would *never* exist, since we already knew that it was a child of a path

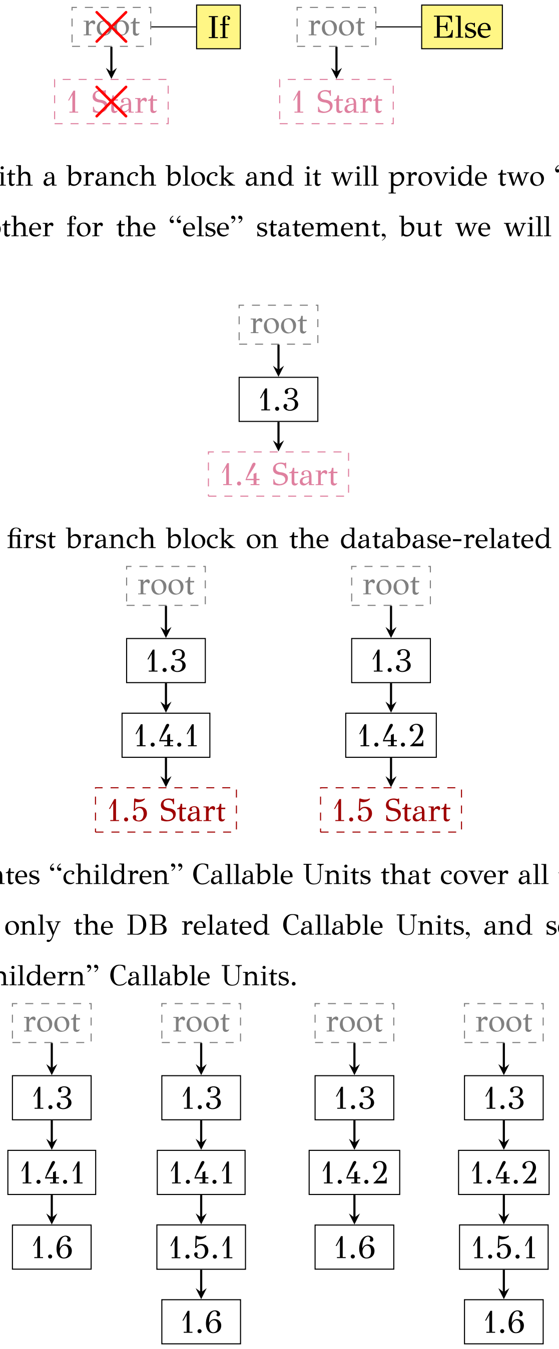
Algorithm 3.7: Creation of execution paths of a Callable Unit

```
Input: A Callable Unit ($M$).
Output: The database-related execution paths of a Callable Unit.
The set of query variants of the project, $QV = M$;
forall Callable Units $m \in QV$ do
    $execPlans(m)$ = CreateQueryVariants($m$);
    $QV \cup execplans(m)$;
end
Procedure CreateQueryVariants($M$)
    $execPlans = \emptyset$;
    if $M$ contains database-related code and not checked then
        if $M$ contains branch statements then
            $variants$ = create variants of $M$'s first branch;
            forall $var \in variants$ do
                if $var$ contains db-related code after the first branch then
                    $execPlans$ += $var$;          ▷ ``else'' is Horizontal prune
                end
            end
        else
            mark $M$ as checked;
            $execPlans$ = $M$;                     ▷ DB related without branches
        end
    end
    return $execPlans$;
```

that does not contain any database-related code (as a sub-graph of the Fig. 3.5a execution path, which we checked for database-related code and we did not find any such function). Moreover, Algorithm 3.7 prunes the code statements that follow after a query end, when there is no other query-related function to follow (e.g. for a new query). Therefore, we speed up our execution time, since we do not have to deal with irrelevant to database source code statements, and additionally we save the main memory needed for the QVG representation.

In Figure 3.9 we describe the steps of Algorithm 3.7 using our reference example

(Fig. 3.3a).



(a) Callable Unit starts with a branch block and it will provide two “children” Callable Units, one for the “if” and another for the “else” statement, but we will keep only the database-related one.

(b) We search for the first branch block on the database-related “child” Callable Unit.

(c) The Callable Unit creates “children” Callable Units that cover all the possible values of the branch block and keeps only the DB related Callable Units, and searches for the next first branch block at those “childern” Callable Units.

(d) Loop blocks are treated as branch blocks, so we create two new “children” Callable Units that: the first contains the loop block statements, whilst the second does not contain the loop block statements. When no more branch and loop statements exist, the paths are complete.

Figure 3.9: Steps of Algorithm 3.7 with the resulting query-related execution paths

3.6 Evaluation

We have evaluated our method using two ecosystems written in *different programming languages*. The first ecosystem we used is the Clementine[2] music player project, which is written in C++ and it stores the information of the tracks of the music library of its users in a database. The second ecosystem is Drupal[3], which is the most popular CMS on sites with heavy traffic[4]. Drupal is written in PHP and it stores the contents of the web pages it manages in a database. Table 3.3 contains more details, such as the number of lines of code, the number of files, and the number of subfolders of the projects we used for our evaluation.

Project	Code lines	Files	Folders	Variant queries	Fixed queries	Total
Clementine	210053	3072	159	10	14	24
Drupal	325421	1096	137	10	84	94

Table 3.3: Projects' descriptions and queries distribution per project

For parsing purposes, in both projects we changed the source code and we transformed the "(condition) **?** (true) : (false)" Branch statements to "**if** (condition) { (true) } **else** { (false) }" statements.

We assess our method for its efficiency, effectiveness and user effort. First, we measure the necessary time and memory (since we produce every possible execution path) for the extraction and reconstruction to take place. Next, our main research goal is to evaluate the extent to which our method actually detects the queries of an ecosystem's applications and correctly reconstructs them. Moreover, we measured the code writing effort of the developer who wants to use our tool.

All experiments were conducted on a typical PC with an Intel Core i5-4570S CPU clocked at 2.90GHz. During time and memory measurements, we used either 1 or all 4 available cores. The main memory of the PC rises to 32 GB (clocked at 1333Mhz).

Time-Memory consumption We have measured the time and the memory needed to recover all the queries for each of the involved data sets. Each experiment was conducted three times and we report the average values for each metric. Our results are

[2] `https://www.clementine-player.org/`

[3] `http://ftp.drupal.org/files/projects/drupal-7.39.tar.gz`

[4] See `http://w3techs.com/technologies/market/content_management`

depicted in Tables 3.4 and 3.5 respectively. Based on these observations, we can say that for both case studies the time and memory required to the query extraction is reasonable. Due to the fact that we do not perform any pruning in the Principled method, we receive a memory error (heap space error) during the execution of the Drupal project, therefore the time measurement is set to infinity. That error is due to a database-related Callable Unit that contains an excessive number of branch and loop statements causing the heap space error for the recursive Algorithm 3.4. In order to have a valid measurement in our tables, we modified our ecosystem excluding the file that was causing that problem.

	Principled		Cross-layer	
Project	1 thread	4 threads	1 thread	4 threads
Clementine	150	127.3	150	127
Drupal	∞	∞	136	89.6
Drupal modified	154.3	92.3	133.6	86.6

Table 3.4: Time measurements (in seconds) for each project, in single and multiple thread combinations

	Principled		Cross-layer	
Project	1 thread	4 threads	1 thread	4 threads
Clementine	1.67	2.586	1.326	2.586
Drupal	∞	∞	1.69	2.13
Drupal modified	2.89	2.73	1.52	1.713

Table 3.5: Max memory needed (measured in GB) for each project, in single and multiple thread combinations

Drilling in to our results: in the Clementine project that was able to run for both algorithms described earlier, we observe that there is no time or max memory difference between the two methods. This is because the Clementine project had only a small number of queries (24), coming from 6 source code files. In Drupal project on the other hand, we notice that there is a drop on both time and main memory measurements. Actually in the modified Drupal project we notice that the main memory drops to nearly half values! That is because due to our pruning technique which keeps only the database-related stuff. As we have seen in Table 3.3 there were not many queries that had variants. If there were existing, the Cross-layered method would have better times for the time measurement too, since the *CreateQueryVariants* procedure is parallelized.

Effectiveness We need to verify the extent to which our method retrieves and correctly reconstructs queries from the application scripts of the ecosystem. The performance measures for this kind of assessment are recall and correctness. *Recall* is defined as the fraction of the retrieved queries of each file over the actually existing ones. *Correctness* is defined as the fraction of the correctly reconstructed queries over the retrieved ones. A correct reconstruction of a query involves (a) retrieving all its structural parts and (b) assembling them correctly, in order to result in a correct and complete query. Table 3.3 depicts the distribution of queries that were either single path (fixed) queries or produced due to branch and loop statements of the host language (variant queries).

Recall To assess recall, we need to *manually* verify the percentage of queries that our method extracts with respect to the queries that actually exist in the code. Due to the vastness of the task, we have sampled the 10% of the database-related files. This is a standard practice in the software engineer community whenever the size of a project is too large for full manual inspection. We manually inspected the code of the evaluated files and we were unable to find any other query, besides the ones that our tool reported. In the functions that were repeating a query in one or more places in their source code, we reported only one occurrence of the query, since there was no variation. If a query changes, then we report the "new" query (the modified one) as well. Our manual inspection was further supported by automated searches in the source code. For Clementine, we decided to focus on a single table of the database. Then, we can search for all occurrences of the table's name. For Drupal, we took advantage of the fact that there are specific functions for querying the database, as prescribed by its manual (both for string-based and for object-based queries): `https://api.drupal.org/api/drupal/includes!database!database.inc/function/`.

Correctness

Regarding the correctness of our method, we examined the sample files on whether the queries that were translated to SQL query environment were correct or not. The correctness for the Drupal project is 95.6% and for the Clementine project is 79.1%. To explain what we considered as a correct query we created the following taxonomy

	Query class	Drupal-7.39	Clementine 1.2.3
Valid:	*all* parts fixed	28/94 (*29.7%*)	5/24 (*20.8%*)
Valid:	variable *values*	61/94 (*64.9%*)	14/24 (*58.3%*)
	overall	89/94 (**95.6%**)	19/24 (**79.1%**)
Invalid:	variable *structure*	05/94 (*04.4%*)	05/24 (*20.9%*)

Table 3.6: Breakdown of generated queries per query class.

of query classes:

1. *Fixed structure*: This class has the queries that can be translated to one of our concrete query environments and run without issues.
   (a) *All parts fixed*: queries that have no variable at all
   (b) *Variable values in "filtering"*: queries that contain a variable that gets its value at execution time but does not intervene with the query structure. In most cases this is a variable that is the second part of a comparison. In our reference example of Fig. 3.1, Line 16 contains the $\$category$ variable which can be replaced by a value, producing a valid query.
2. *Variable structure*: in this class we have variables that alter the query structure. This means that the data providers are unknown to us, so in order to produce a valid query we needed to know in advance the values of the parameters that were given to the calls of those Callable Units.

Table 3.6 contains the number of queries that belong to each classification for each one of our case studies, which consequently provide the correctness measurement for our method. Observe that the internal breakdown for the different categories (rows in the table) is quite different for the two cases. However, *we do achieve 100% correctness and recall for the two first categories*. For the last category, we fail to produce an abstract representation due to the fact, that many times the variable structure refers to a variable table in the FROM clause that is assigned at runtime. A flexible handling of such occurrences (with variable tables involved) is part of future work.

User effort

As previously stated at Section 3.3, there is a preprocessing step that is needed in order to translate the projects API database-related functions to Abstract Data Manipulation

Operators. In Table 3.7 we describe the user's effort for the two projects that we examined. The effort is measured in the number of functions that needed translation from the project's API, and in the lines of code that were written for the translation of those API functions to Abstract Data Manipulation Operator. Additionally, we present the number of functions and lines of code that were needed to use the code of a host language (C++ and PHP respectively) and functions and lines of code that were needed to translate a string based query to a set of Abstract Data Manipulation Operators (which is a fixed code for all projects).

Project	API	Host language	String based
	func./LOC	func./LOC	func./LOC
Clementine (C++)	4/59	9/341	7/195
Drupal (PHP)	11/251	9/347	〃

Table 3.7: User effort (Number of functions to translate / Lines Of Code)

3.7 Conclusion

As we have already seen in Chapter 2, there have been a number of works that tried to extract the embedded in the host code queries, but the problem was still open, since the queries got modified from the host language branch and loop statements. Additionally, most of the state of the art works were only capable to produce results when they examined only one host language, or they were just using the database query logs that were available in the database server that responded to the queries. In this chapter, we have set the properties on how to extract the embedded queries of a project in a *host language independent way*. In the projects that we examined we noticed that the developers were using two different programming styles to query a database. With the algorithms that we proposed, we have extracted queries in a *programming style independent way*.

Using the Query Variants Graph representation we were able to create *every possible database query that might occur during the execution time of the software*. Then, we proposed a way to represent those "text based" queries, using the Abstract Query Representation with the extensible palette of Abstract Data Manipulation Operators that are capable to describe not only the operators of the relational algebra but more.

Finally, we export the Abstract Query Representation to *more than one concrete query language environments* (in this work we described how an AQR query can become an SQL or MongoDB query).

In the experiments we conducted, we were able to fully locate the queries in the projects we examined, and we fully or partially recover an 80% of those queries, with small user effort.

Having found the queries of a project, we move on to describe the fundamentals of a well constructed query based on the database schema of an ecosystem, and a metric that will help us evaluate the quality of a project's queries, so as to evaluate the understandability and maintainability of the project's database related code.

Chapter 4

A Metric to Assess the Coupling of Software to the Database

4.1 Introduction

There is a great need of software companies and organizations to be able to adapt to the new requirements of their users. To do so, they need to have a well-written and easy to maintain and understand code. Using quality metrics is a way to identify whether a program is easily understandable and maintainable [61, 60, 64].

A vast set of metrics that describe the quality of (mostly object oriented) software exist in the literature. Those metrics show –early enough– how easy it is for developers to modify their code when needed (for example in order to avoid a full rewrite of a project), or how easy a new programmer could get into the source code and extend or fix it.

A metric that has been widely used in object oriented programming is coupling in order to describe the quality of software.

The coupling metric is used to describe the connection of a software module to another software module. For example, when a class $classA$ has a *high* coupling to class $classB$, it means that when a change occurs in $classB$, then there is a high probability that $classA$ should change too.

Our objects of study are Data Intensive Information Systems ($DIIS$) software. $DIIS$ are applications that use data sources in order to save, retrieve, or query-and-aggregate data that come from various sources (e.g., the entire data processing support of a company or an organization, content management systems like Twitter streams, personal music play lists, invoices of a company, etc.). $DIIS$ projects have their source code organized in a software module hierarchy and a data source hierarchy.

The software hierarchy –briefly– is: the packages, the classes, and the subroutines of the classes (in non object oriented software these are the folders, the files and the subroutines respectively). The data source hierarchy is: the data entities, the attributes of the tables and the data views. In relational databases –since this is where we focus via our examples and experiments– the data entities are represented by the database tables, the attributes are represented by the columns of the tables and the data views are represented by the views of the database schema.

In Data Intensive Information Systems applications, the software uses the underlying database schema to provide information to the users of the application. The issue is that since there is a variety of ways to use the database schema, for example i) via using Object-relational mapping (ORM) between the database tables and the software, or ii) via using Application Programming Interfaces (API) that construct the parts of the queries, or iii) via embedded query statements in the code, there is no clear way to identify whether the software is well-written or not. The question that rises is: *given the database schema and the code that contains the queries that act as a bridge between them, is it possible to assess whether this bridge is constructed in a way that minimizes the unnecessary coupling and increases the maintainability of the code with respect to the database schema?* Our goal is to come up with the means to evaluate the quality of the relationship between the source code of the application and the underlying database schema, and to propose a way to increase that quality, when applicable. Therefore, following the software engineering theory, we propose a metric for $DIIS$ applications since the state of the art does not provide adequate answers.

During our research we found that using one metric is adequate enough to provide information on whether the source code is not well-written, although more metrics might be even more helpful. Additionally, we provide a number of principles that all metrics (proposed of future ones) should follow that are Data Intensive Information Systems related.

With this work:

- we describe what the fundamental properties of a good database related software is,
- we propose a metric that will identify badly constructed database schemata and software as soon as possible,
- when it is possible to rewrite part of the code and the database schema, we propose how this should be done to obtain better quality metrics

.

Therefore, we propose a metric that is scalable regarding the levels of abstraction of the software. We start via checking the proposed metric (Data-Software Coupling Quality) between the subroutines that contain database queries, and we move on to higher layers where those subroutines are used. To evaluate the proposed metric, we use the results of Chapter 3 and we examine our metric in conjunction to the evolution of a project, as well as the evolution of Data-Software Coupling Qualitymetric, after the code rewrites.

4.2 Evaluating Data-Software Coupling Quality

Our goal is to develop metrics for Data Intensive Information Systems that could provide information on how tightly or loosely coupled is the source code of such a project to its database schema. Providing such a metric would benefit the developers in maintaining their code and would provide a way to identify whether the changes the developers suggest and apply to the source code help on the project's evolution and sustainability macroscopically. When the metrics are good, it is expected to have code that is easier to maintain, test and debug compared to a project with bad metric measurements. Additionally, a project with good measurements is easier for new developers to understand it, thus the developers can move faster from the training on the job part to the production part of their job.

Before we move on to propose such metrics, we need to define what a Data Intensive Information Systems project is and how we represent it on our proposal. As already mentioned, Data Intensive Information Systems are applications that use databases to work with data that either want to store, update, or query. Since there exist two different parts (the database and the software one) that work together to achieve whatever the developers need, we have to represent both parts in a uniform way.

Having a closer look on the database related part, we have the tables with their columns that describe where those data are stored, and the views that query those tables in order to provide easier access on the data to the developers. On the other hand, we have database queries that are used in the software part and either (rarely) use the views of the database, or directly use the tables, in order to aggregate, retrieve, store, modify or even remove data. Since the software part is frequently related to the user preferences (via a GUI or a terminal input etc.) the queries are generally not strictly fixed but parameterized. Therefore we need to represent the Data-Software metrics in a way that will let us identify not only on a low level but on different levels of a hierarchy the connection of the software to the database.

Figure 4.1 briefly depicts the Data Intensive Information Systems hierarchy that we mentioned. On the left side of we depict the software related part and on the right side we depict the database related part.

To further explain the hierarchy we discuss the software level, typically, where we have packages that are sub-folders that contain the source code files of each package. Those files, contain subroutines that may interact with the Database via database queries. Those subroutines might be organized in classes (when we refer to object-oriented software) or not, when we have procedural or other programming language styles.

Regarding the Database part, it consists of Tables that contain columns that the users use in order to store and update data to them, or to delete and retrieve data from them. Additionally the users might use aggregate functions (such as *SUM*, *AVERAGE*, *MIN*, *MAX*, *COUNT*, in order to acquire aggregated values from all or from specific parts of the stored data.

In the literature exist works ([93], [94]) that examined a great number of Data In-

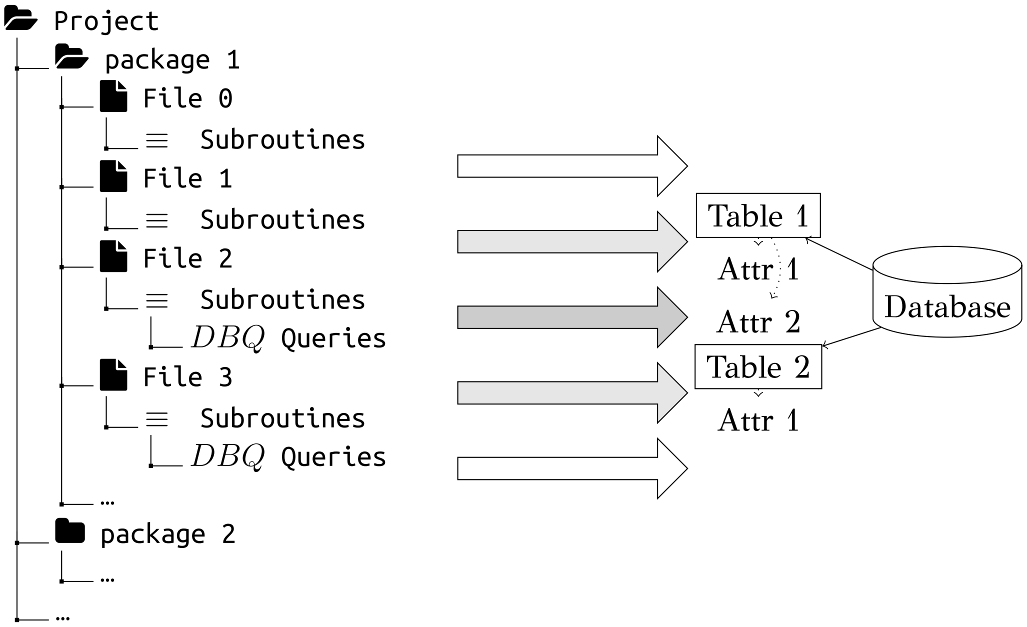


Figure 4.1: Typical Data Intensive Information Systems project organization

tensive Information Systems projects of a specific programming language (for example Java) and describe how those projects use the tools provided by the programming language to write queries. During our research we have encountered three different styles, regarding the software side, that a developer may use in Data Intensive Information Systems to create a database query in the subroutines that pertain to the project. These styles are programming language independent and they are described in the following list:

- A technique called Object-Relational Mapping (`https://en.wikipedia.org/wiki/Object-relational_mapping`) is when functions perform the Create, Read, Update and Delete (CRUD) task of data for each table of the database. Those functions let the developers work with main memory objects and whenever they want to perform a task such as to update an object to the database, they simply call the appropriate Update function (in many cases called "save"). Those functions might be user defined or generated by software such as Hibernate (`http://hibernate.org/`). Listing 7 describes the way the ORM querying method works.
- Via using custom made APIs[1] that create classes which are responsible for the database communication. Those classes need as input from the subroutines that use them the specific tables that are going to use. The API technique is more

[1] `https://en.wikipedia.org/wiki/Application_programming_interface`

```
public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      factory = new Configuration().configure().buildSessionFactory();
      ManageEmployee ME = new ManageEmployee();
      Integer empID1 = ME.addEmployee("Zara", "Ali", 1000); // Add employee
      Integer empID2 = ME.addEmployee("Daisy", "Das", 5000); // records in
      Integer empID3 = ME.addEmployee("John", "Paul", 10000); // database
      ME.updateEmployeeSalary(empID1, 5000); // Update salary
      ME.deleteEmployee(empID2); // Remove employee
   }

   public Integer addEmployee(String fname, String lname, int salary){
      Session session = factory.openSession();
      Transaction tx = session.beginTransaction();
      Integer employeeID = null;
      Employee employee = new Employee(fname, lname, salary);
      employeeID = (Integer) session.save(employee);
      tx.commit(); session.close();
      return employeeID;
   }

   public void updateEmployeeSalary(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = session.beginTransaction();
      Employee employee = (Employee)session.get(Employee.class, EmployeeID);
      employee.setSalary( salary );
      session.update(employee);
      tx.commit(); session.close();
   }

   public void deleteEmployee(Integer EmployeeID){
      Session session = factory.openSession();
      Transaction tx = session.beginTransaction();
      Employee employee = (Employee)session.get(Employee.class, EmployeeID);
      session.delete(employee);
      tx.commit(); session.close();
   }
}
```

Listing 7: ORM example usage (Java programming language)

```
function _profile_get_fields($category,$register=FALSE) {
 $query = db_select('profile_field');
 if ($register) {
  $query->condition('register',1);
 }
 while (!user_access('administer users')) {
  $query->condition('visibility',PROFILE_HIDDEN,'<>');
 }
 return $query->fields('profile_field')
   ->orderBy('category','ASC')
   ->orderBy('weight','ASC')
   ->execute();
}
```

Listing 8: Queries that use an API to retrieve the database data (PHP programming language)

versatile compared to ORM, since it may add functions that join many tables to get the desired results etc. Listing 8 describes a query using the API method. The semantics of the methods are as follows:

db_select: creates a query for the database table that is given as parameter

condition: creates a filter for the query (when two parameters follow, the filter is for equality '=', otherwise the third parameter describes what is the condition)

fields: describes which attributes are the output of the query (if it is the name of the queried table, then as output we have all the attributes of the table

orderBy: performs a sorting on the output, based on the second parameter (ascending, descending)

The order of the invocation of the API methods sometimes has an impact on the query itself: in the example of Listing 8 the ordering is first by 'category' and then by 'weight'.

- Finally, the classic way of embedding SQL queries on the source code of the subroutines is the one that is found in the majority of the Data Intensive Information Systems. Listing 9 depicts an example of embedded SQL technique.

```
$result = db_query('SELECT source, alias FROM {url_alias} WHERE source in (:system)
 AND language = :language_none ORDER BY pid asc;', $args);
```

Listing 9: Queries that use embedded SQL to retrieve the database data (PHP programming language with embedded SQL)

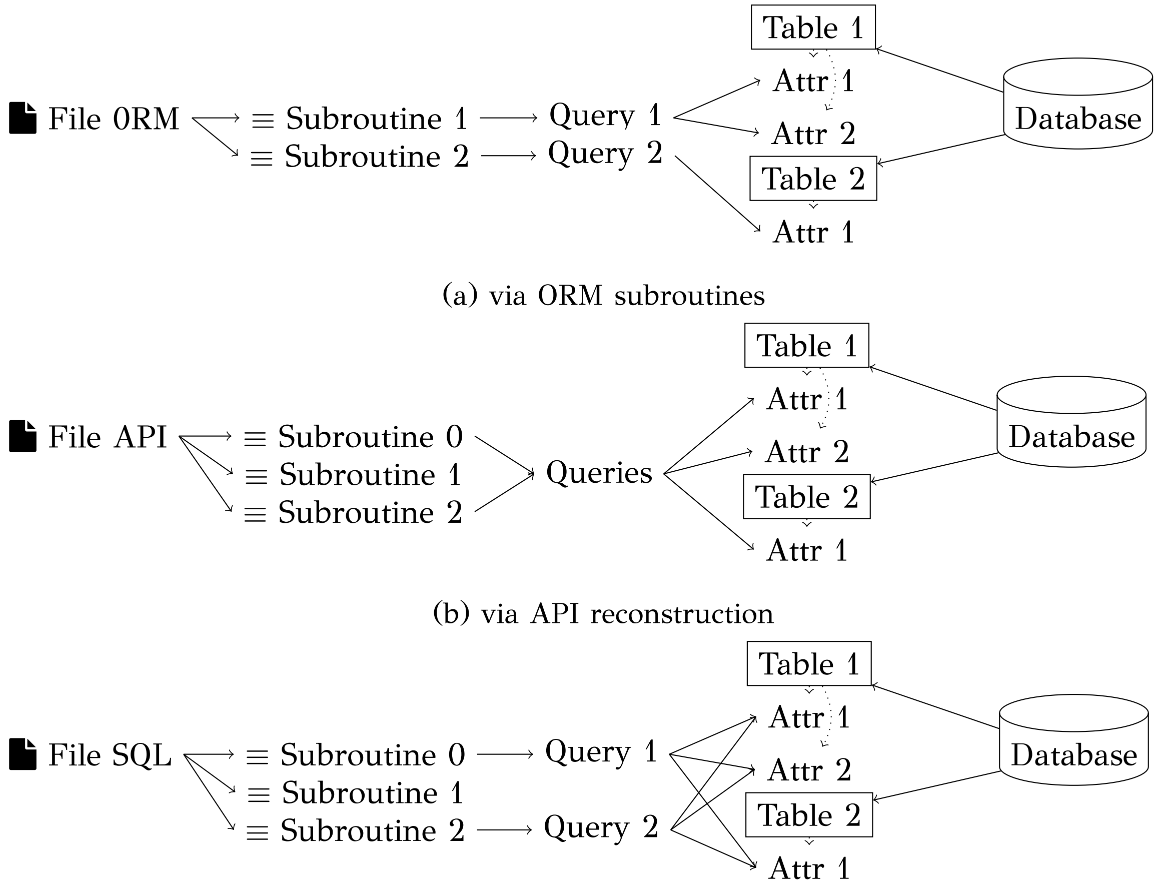
Regarding the querying techniques, an interesting finding is that we have encountered projects where the developers used more than one techniques to express a database query. Via examining the evolution of the code of those projects, we found that the developers are moving from the traditional way of writing embedded SQL to more agile ways, such as APIs.

Summarizing, in Fig. 4.2 we depict the three ways a database query may be formed. In the first part we can see that a set of subroutines used in a class that interact with the database each one for a single table (*ORM* technique). In the middle part we can see that all queries are executed using an API. Finally, in the last part we depict the oldest way that someone may use to query a database, which is via using embedded SQL statements inside the subroutines of the source code files.

Although the middle and last part (API and Embedded SQL) look different, using the method described in Chapter 3, we can represent both ways uniformly with the Abstract Query Representation (AQR). Via AQR, we are able to model Data Intensive Information Systems on the grands of a graph-based model that we use for the metric we propose.

4.2.1 Using Abstract Query Representation for API and Embedded SQL techniques

In our model, we treat both the *API* and the *embedded SQL* ways of query creation uniformly. As we already have seen in the formal definition of AQRs in Chapter 3, AQR is a way to describe each query as a sequence of operators that follow each other, and those operators contain information about the specific parts of the query. The Abstract Query Representation of a query for both API and Embedded SQL querying techniques is depicted in Fig. 4.3. The upper part depicts the API way of querying while the lower part depicts the Embedded SQL technique. What we focus on, for our purpose, is the rightmost part that contains the operators with their information. The rightmost part is the same since both ways create the same database query, fetching



(a) via ORM subroutines

(b) via API reconstruction

(c) Embeded SQL queries

Figure 4.2: Extension of graph for *DIS*

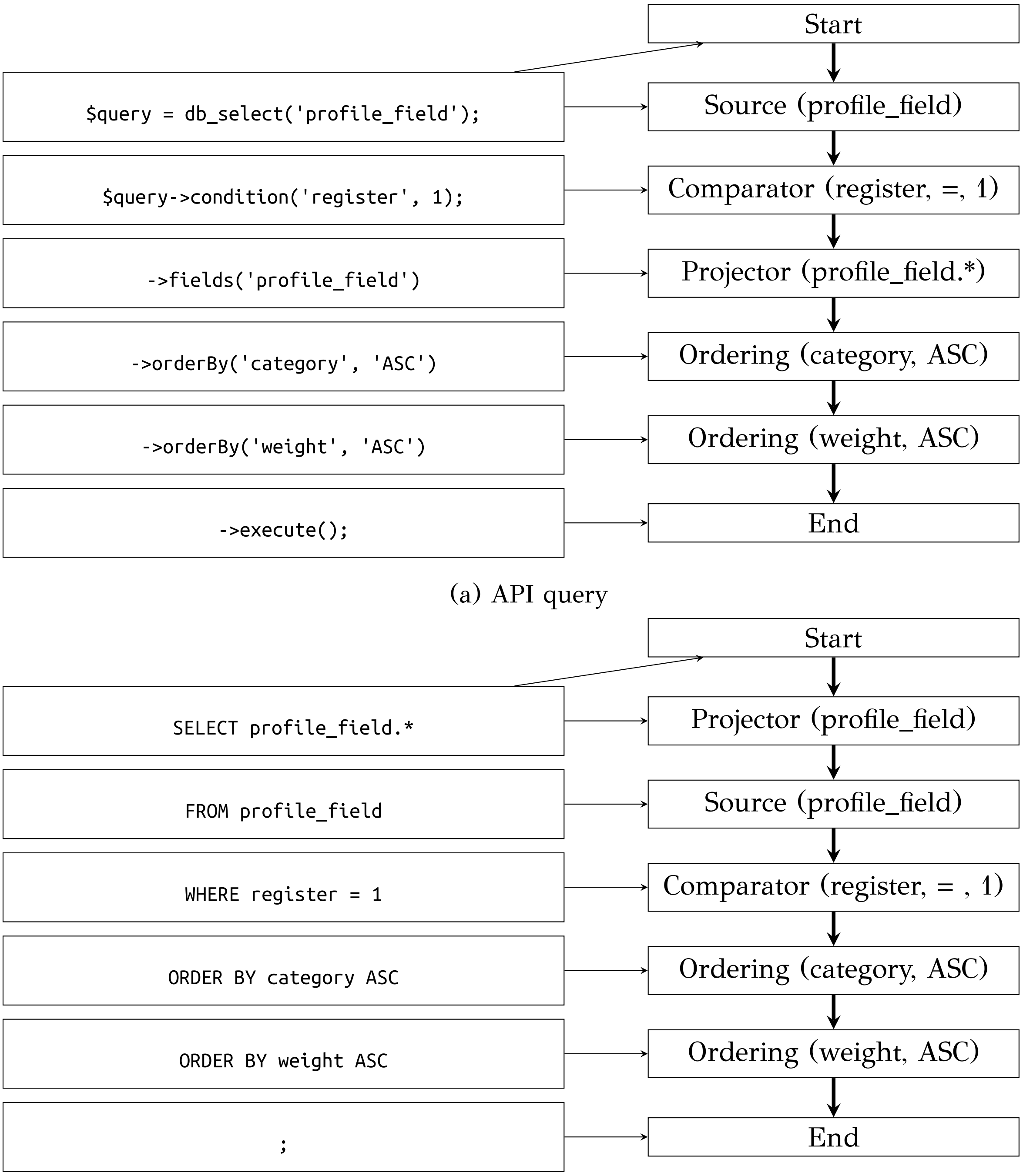
sorted per "category" and "weight" all the columns of "profile_field" table that have "register" equal to 1.

Having explained the way that we can represent the API and Embedded SQL querying methods to a uniform way, we move on to describe how we can represent a Data Intensive Information Systems project via a graph-based model.

4.2.2 Formal (graph-based, uniform) model of Software & Data

We model the Data Intensive Information Systems as graphs. The nodes of the graph represent the hierarchies of software and database parts of *DIIS* projects. The nodes related to the software hierarchy represent:

- the database queries as these are formed in the source code
- the subroutines that contain the database queries



```
$query = db_select('profile_field');
$query->condition('register', 1);
->fields('profile_field')
->orderBy('category', 'ASC')
->orderBy('weight', 'ASC')
->execute();
```

(a) API query

```
SELECT profile_field.*
FROM profile_field
WHERE register = 1
ORDER BY category ASC
ORDER BY weight ASC
;
```

(b) Embedded SQL query

Figure 4.3: Abstract Query Representation of API and Embedded SQL query

- the files that contain the subroutines
- the folders that contain the files
- the project that contains the folders.

As above, the database hierarchy related nodes represent:

- the columns (attributes) of a database table
- the table that contains those columns
- the database that contains the tables.

As one may observe, there is a "contains" relation between each line on those hierarchies. Additionally, there is another relationship that describes that parts of the graph are "used as parts" (components) of other parts. Given the "contains" hierarchy we can compute induced graphs at different abstraction levels. As we will demonstrate in the sequel, our proposed metric is multigranular and can be computed for any induced graph at arbitrary levels of abstraction. This way, we can, for example, describe the metric of a query of the source code, as well as the metric of the package that contains the file that contains that query.

The edges of our graph are used to describe relationships between the nodes. As already mentioned, we have a "contains" relationship already described in the hierarchies of our nodes. Those hierarchies are also related via the queries of the source code. These two relationships describe the edges we have in our graph:

- the "contains" relationship, as previously defined
- the "uses as part" relationship, as previously described.

We define a Data Intensive Information Systems as a graph $G(V, E)$, whose components we are going to detail right away. We will start by presenting the nodes and then move on to present the edges of the graph. Regarding the nodes of the graph have a hierarchy $\mathcal{H} = (\mathcal{T},\ \mathcal{L})$ (a strict partial order of node types), and in our case it is defined over a set of node types $\mathcal{T}$ (where $\mathcal{T} = \{\mathcal{T}_1 \ldots \mathcal{T}_n\}$) and an extension of $\mathcal{T}_{low} < \mathcal{T}_{high}$ is interpreted as $\mathcal{T}_{low}$ is contained in $\mathcal{T}_{high}$.

In the case of software we assume the hierarchy $\mathcal{T}_{sw} = \{project,\ package,\ script,\ embedded\ query\}$ and in the case of relational database we assume the hierarchy $\mathcal{T}_{db} = \{database,\ relation,\ column\}$. Our definitions are generic and extensible with respect to the particular node types, however, what we care about is the existence of a structured hierarchy of nodes.

Since our hierarchy is a strict partial ordered set, the following properties of a structural partial order hold for the members of $\mathcal{H}$:

- $\mathcal{T} \nless \mathcal{T}$

- $\mathcal{T}_{low} < \mathcal{T}_{high} \Rightarrow \mathcal{T}_{high} \not< \mathcal{T}_{low}$
- $\mathcal{T}_a < \mathcal{T}_b \wedge \mathcal{T}_b < \mathcal{T}_c \Rightarrow \mathcal{T}_a < \mathcal{T}_c$

Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ of a Data Intensive Information Systems we discriminate the following kinds of nodes and edges:

- $\mathcal{V}_{db}$ the set of nodes pertaining to the database
- $\mathcal{V}_{sw}$ the set of nodes pertaining to the software applications
- $\mathcal{E}_{cont}$ the set of edges denoting "is contained within" relationships
- $\mathcal{E}_{prov}$ the set of edges denoting "part of" relationships.

Regarding $\mathcal{G}$, $\mathcal{V}$ is the union of the $\mathcal{V}_{db}$ and $\mathcal{V}_{sw}$, and $\mathcal{E}$ is the union of the $\mathcal{E}_{cont}$ and $\mathcal{E}_{prov}$.

Then, a hierarchy $\mathcal{H}_{db}(\mathcal{T}_{db}, \mathcal{L})$ practically splits the nodes of $\mathcal{V}_{db}$ into equivalence classes. Given the set of nodes $\mathcal{V}_{db}$ each type of $\mathcal{T}_{db}$ has an active domain $dom(\mathcal{T}_{db}|\mathcal{V}_{db})$ which is the subset of the nodes of the graph $\mathcal{G}$ (hereafter *Architecture Graph*) that pertain to this type. Then the edges of $\mathcal{E}_{cont}$ materialize the relationship $\mathcal{L}_{db}$. In other words if $\mathcal{T}_{low} < \mathcal{T}_{high}$ then $\forall \mathcal{V}_{low} \in dom(\mathcal{T}_{low}|\mathcal{V}_{db}) \exists \mathcal{V}_{high} \in (\mathcal{T}_{high}|\mathcal{V}_{db})$ such that $\exists \mathcal{E}(\mathcal{V}_{low}, \mathcal{V}_{high}) \in \mathcal{E}_{cont}$. The respective property holds for $\mathcal{H}_{sw} \wedge \mathcal{E}_{cont}$.

4.2.3 Describing a well designed Data Intensive Information Systems

Having explained all the theoretical background that we use to represent the Data Intensive Information Systems projects, we move on to provide the fundamental properties that the *DIIS* metrics should have. Briand et al in [62] –where the object oriented metrics were originally introduced– proposed a set of properties for the software metrics that if none is met then the proposed metric is not well defined. Those are: i) non-negativity and normalization, ii) null value and maximum value, iii) monotonicity, and iv) cohesive modules.. In a similar vein to that work, we propose an additional (to object oriented) set of requirements for the Data Intensive Information Systems that describe whether a software that is related to a database schema is easily understandable and maintainable. Additionally, those requirements

- **P1**. A Data Intensive Information Systems project should try to *minimize the number of subroutines intervening with a database table*, since this would produce too much effort to resolve inconsistencies incurred by schema changes.
- **P2**. A Data Intensive Information Systems project should try *to minimize the number of relations accessed by a subroutine*, to minimize the probability of having to be maintained due to schema evolution actions.
- **P3**. A Data Intensive Information Systems project should not employ multiple (nested) subroutines to construct a database query, to avoid having variety of source code parts that need to get examined in the event of maintenance.
- **P4**. A Data Intensive Information Systems project should be able to adapt easily the database schema to changes that occur to data provider changes (persistent storage of JSON streams is a common example).

Figure 4.4: Data Intensive Information Systems understandability and maintenance requirements.

request the database to be able to adapt to software changes. Those requirements are presented in Fig. 4.4.

Moving on, we present the Data-Software Coupling Quality metric that follows the requirements of Fig 4.4.

4.2.4 Data-Software Coupling Quality

In this Section, we introduce the Data-Software Coupling Quality metric of Data Intensive Information Systems projects. Before we propose any metric, we discuss how the requirements of Fig. 4.4 can be translated in to metric requirements regarding the coupling metric and then, how those are applicable to our graph-based representation.

The list of Fig. 4.5 describes how each one of the requirements of list of Fig. 4.4 becomes a Data-Software Coupling Quality metric related requirement, alongside with the properties of each requirement. We additionally mention any requirements we need Data-Software Coupling Quality metric to follow.

- The first requirement (*not many subroutines intervening with a relation*) can be translated in to having the best possible value when there is only one subroutine using a database table.
- The second requirement (*a subroutine not querying many relations*) can be translated in to having the best possible value when there is only one database table used, and the value should worsen when there are more tables used.
- The third requirement (*not many levels of dependency to create a query*) can be translated into having a stable or worsening value of the metric when we merge the ancestors. That is a case where there exist nested queries at a subroutine to perform a query.
- The fourth requirement (*adapt database schema to providers changes*) can be translated in to the same metric requirement as the first requirement of this list.
- Finally, a metric should also be multigranular for different abstraction ground, both for software and data.

Figure 4.5: Translation of Data Intensive Information Systems requirements to coupling metric requirements.

Having explained the coupling metric requirements, we move on to discuss how these requirements can be used in the graph-based model we proposed in the beginning of this section. The list of Fig. 4.6 describes how each one of the requirements of list of Fig. 4.5 becomes a graph related requirement and is therefore applicable to our graph-based representation.

Next, we propose a coupling metric that we name Data-Software Coupling Quality and we explain how this metric meets the requirements we earlier described.

Assume a bipartite graph $G(R,\ S,\ E)$ with R representing a set of database constructs (e.g. tables, sub-schemata, etc.), S representing a set of software modules and E representing a set of directed edges, where an edge $(s,\ r)$ starts from a software

- *The minimization of the number of subroutines intervening with a relation*, can be translated to having each Relation ideally with only one incoming edge, since, the minimum amount Data-Software Coupling Quality occurs when there is only one data provider used.

- *The minimization of the number of relations queried by a subroutine* can be translated to ideally having only one outgoing edge at each subroutine of a projects file (that is complementary to the previous one, since there could be a subroutine that uses 3 relations that no one else uses). Likewise to the previous bullet, having a minimum amount of Data-To-Software Coupling value on a higher level of abstraction (for example a project file, instead of a query) occurs when there is only one data provider for the file.

- *Avoiding multiple (nested) subroutines to construct a database query* can be translated in to needing only one node to create a query (no subroutines at all).

- *Adapting easily the database schema to providers changes* can be translated to the same graph requirement as the first requirement.

- Extra to Fig. 4.4: *Multigranularity for different abstraction levels* is done via recursively rolling up from queries to subroutines, from subroutines to files, from files to packages, and from packages to project, as we show in Fig.4.1

Figure 4.6: Translation of Data-Software Coupling Quality metric requirements to *Architecture Graph* properties.

construct s and targets a database construct r. Assume that each module $s \in S$ has an out degree $\delta^o_\tau(s) \in \mathbb{N} \wedge 0 \leq \delta^o_\tau(s) \leq \tau$.

Definition 4.2.1. *The Data-Software Coupling Quality $C^{DB}(s|\tau)$ of a software construct s to a set of database constructs τ is defined as*

$$C^{DB}(s|\tau) = \begin{cases} \dfrac{1 - \log_{\frac{1}{|\tau|}}\left(\dfrac{\delta^o_\tau(s)}{|\tau|}\right)}{|s|} & \tau > 1 \wedge 0 < \delta^o_\tau(s) \\ 0 & \tau = 1 \\ undefined & \delta^o_\tau(s) = 0 \end{cases} \tag{4.1}$$

Properties of equation 4.1

1. $C^{DB}(s|\tau)$ is monotone.

2. $C^{DB}(s|\tau) \in [0 \ldots 1]$

3. $C^{DB}(s|\tau) = undefined$, if there is no dependency of s over the database ($\delta^o_\tau(s) = 0$).

4. $C^{DB}(s|\tau) = 1$, if there is only one dependency of s over the database tables ($\delta^o_\tau(s) = 1$).

5. $C^{DB}(s|\tau) = 0$, if the s depends on (has an edge to) all the nodes of τ when $\tau > 1$.

6. when $\delta^o_\tau(s) = 1$ due to *only* one edge (s, r), with $r \in \tau$, then $C^{DB}(s|\tau)$ has the maximum possible value.

Proofs of Properties of equation 4.1

1. *Proof.* We know that any logarithm $log_\beta x$ is monotone.

   - If $\beta > 1$ then the logarithm is monotonically increasing.
   - If $0 < \beta < 1$ then the logarithm is monotonically decreasing.

   In our case, when $\tau = 1$, the equation is always 0. When the $\delta^o_\tau(s) = 0$ the equation is undefined. The remaining case is where $\beta = \frac{1}{|\tau|}$. Therefore, since our β is always < 1 the $\log_{\frac{1}{|\tau|}} x$ is a monotonically decreasing function. Thus, if

$$\begin{gathered} x < y \implies \\ \log_{\frac{1}{|\tau|}}(x) > \log_{\frac{1}{|\tau|}}(y) \implies \\ 1 - \log_{\frac{1}{|\tau|}}(x) < 1 - \log_{\frac{1}{|\tau|}}(y) \implies \\ C^{DB}(x|\tau) < C^{DB}(y|\tau) \end{gathered}$$

   Therefore, $C^{DB}(\delta^o_\tau(s)|\tau)$ is a monotone function. □

2. *Proof.* If $\delta^o_\tau(s) = 0$ the equation 4.1 is undefined. Then, since $\delta^o_\tau(s) \in [1 \ldots \tau]$ –because $\delta^o_\tau(s) \in \mathbf{N}$– the minimum value for $C^{DB}(\delta^o_\tau(s)|\tau)$ is $C^{DB}(1|\tau)$ which (using the equation 4.1) is 1, and the maximum value is $C^{DB}(\tau|\tau)$ which is 0, when $\tau > 1$. Therefore we make sure that $C^{DB}(s|\tau) \in [0 \ldots 1]$ □

3. *Proof.* Obvious, using the equation 4.1 with $\delta_{\tau}^{o}(s) = 0$. □

4. *Proof.* Obvious, using the equation 4.1 with $\delta_{\tau}^{o}(s) = 1$. □

5. *Proof.* Obvious, using the equation 4.1 with $\delta_{\tau}^{o}(s) = \tau$. □

6. *Proof.* Obvious, since $C^{DB}(s|\tau)$ is monotonous. □

Proof of well defined metric of equation 4.1

We have already proved that Data-Software Coupling Quality metric complies with the additional requirements that we have set for the Data Intensive Information Systems projects. Finally, we need to prove that the proposed metric also meets (at least some of) the Briand et al. properties.

- The *non-negativity and normalization* property is met, since we can not have negative numbers because the values of equation $C^{DB}(s|\tau) \in [0 \ldots 1]$, as we proved in proof 2.

- The *null value and maximum value* property is met, since when we have 0 dependency then we have 0 coupling as we proved earlier in proofs 3 and 4, and also our maximum value is 1, as we proved in proofs 2 and 5.

- The *monotonicity* property is met, because we use a monotonous function in our equation, and we have proved that in proof 1.

- The *cohesive modules* property is not met. Take for example a database with two tables (T_a and T_b) and two queries in different subroutines (Q_a and Q_b), where both queries use both tables. The sum of equation 4.1 is 0 + 0 = 0. The equation 4.1 for the merged modules is 0. Therefore, the sum of the un-merged couplings is not grater than the coupling of the merged modules.

Why not use a simpler equation?

An observant reader would suggest that a simpler equation would perform as good as equation 4.1, without the need of logarithms. For example a simple equation would be to divide the out degree $\delta_{\tau}^{o}(s)$ with the total number of tables available τ in the database schema. Equation 4.2 describes such a metric.

$$C^{DB}(s|\tau) = \frac{\delta_{\tau}^{o}(s)}{|\tau|} \tag{4.2}$$

This simpler equation (4.2) is actually the base of our approach (equation 4.1). The benefits of our approach are that we can easily locate the subroutine queries that use only one relation, which is a good query as stated in the requirements of Fig. 4.4. That is because the simple approach would provide a different value on each database schema (or version of a database schema), depending on the size of the database.

For example: a database schema started with 38 relations but eventually, since the users required more features, it reached 57 relations. The simple approach for a query of a subroutine that only uses one relation, would provide for as a metric value a fraction of $\frac{1}{|38|} = 0.0263157894737$. In case this subroutine's query did not change, the value for the new version would be $\frac{1}{|57|} = 0.0175438596491$. In both cases the query is considered as a good one, based on the requirements described in Fig. 4.4. Now, using the equation 4.1, the result would be in both cases an easily to locate number: “1”.

Nevertheless, the idea of Data-Software Coupling Quality metric is to count the number of providers a query has. The more the providers, the less good the value. This is actually a common sense, since when a developer wants to separate a number of tuples that meet his criteria, he has to write complex comparisons probably containing all the providers included. Moreover, in cases where aliases are used, the query is besides longer, also harder to understand, since one has to search what the alias represents.

4.2.5 AQR in our model and metrics

As already mentioned, we used the AQR to represent the queries of API and Embedded SQL methods. Having this representation, we use it on our graph-based model to link the AQR operator nodes to the database table nodes. Fig. 4.7 depicts how the Abstract Query Representation is used in our graph-based model.

Regarding the metrics of Data Intensive Information Systems, the AQR provides an advantage on measuring Data-Software Coupling Quality metric. Via traversing the Fig. 4.7 horizontally we have an indication of the query's coupling. For example, via counting the number of *Source* operators we can count the coupling of Queries to Tables –which is our Data-Software Coupling Quality metric).

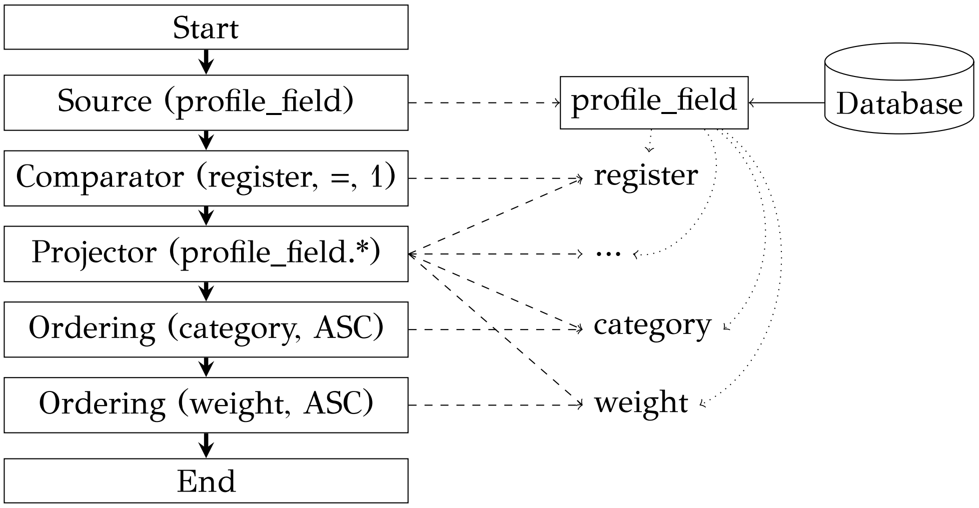


Figure 4.7: Abstract Query Representation of API created query extended to depict the database connections of the operators

4.3 Data-Software Coupling Quality Experiments

As already mentioned, our metric is measuring a value that is common sense that when it is bad, the queries are harder to follow, maintain, etc. Nevertheless, to evaluate our metric, we used the first and last versions of Drupal 4.x.x project that had a number of changes in its database schema, compared to other projects that we examined and had only minor (or no changes at all). In the earlier versions, the developers of the examined project used the Embedded SQL technique to query tables, while on the later ones, they started moving on to the API technique. The versions that we examine are in between, containing both styles: the first version contains only Embedded SQL technique queries while the last contains both Embedded and API technique queries. Table 4.1 contains information on the database related folders and files of each project, of the Drupal versions that we examined, and also depicts that the database related software increased both in number of folders and files (from 2 database related folders to 3 and from 28 files to 40). Additionally, Table 4.1 contains the number of database queries of each version, which also increased.

Regarding the research question, we examine two:

- *does the Data-Software Coupling Quality metric indicate which files change when we examine its rolled up per file value in a folder?*
- *does the Data-Software Coupling Quality metric follow the Lehman's Lows [17] knowing that a set of software maintenance steps occurred in the life of the project?*

Project information				Query related		
Version	Lines of code	Files	Folders	Folders	Files	Queries #
Drupal 4.1.0	21.198	107	13	2	28	240
Drupal 4.2.0	23.499	95	13	2	28	247
Drupal 4.3.1	24.678	102	13	2	29	260
Drupal 4.4.3	25.911	115	13	3	32	263
Drupal 4.5.8	35.312	132	12	3	35	284
Drupal 4.6.11	37.992	135	14	3	36	332
Drupal 4.7.11	47.936	172	14	3	40	358

Table 4.1: Evaluation Projects

For answering the questions, we check the rolled up metric value of the project. Regarding the first question, we examine the changes of the value of our metric in the files that are common in both minor and major versions. What we would ideally want to encounter is that the files that have small values in the minor version, will be affected by the changes of the database schema and increase their metric values in the major version. This would mean that the changes performed were meant for the files that were not in such a good quality condition. Moving on to the next question, we check whether the metric value of each version of the project increased or decreased, having in mind that:(i) the project adapted to new user requirements but eventually (ii) there occurred maintenance steps in the development (at latest versions). What we expect (based on Lehman's Lows of evolution) is that the average value will drop due to adaptation, but eventually rise due to maintenance.

Finally, as we mentioned in Section 4.2.1, the Abstract Query Representation could be used to measure the Data-Software Coupling Quality of the projects, but we have not performed any experiments using the AQR representation.

Before we move on, in Table 4.2 we present the changes that occurred between the versions we examine, grouped by type of change (addition, deletion, rename, type change, primary key change, others) per relation.

Relation	Change Type	Involved attributes
accesslog	addition	aid, path, sid, timer, title
	deletion	nid
	type change	url
blocks	addition	pages, theme, throttle, visibility
	deletion	path
	type change	delta, region
book	addition	vid
	deletion	format, log
	primary key change	nid removed from primary key
boxes	addition	format
	deletion	type
bundle	deletion	complete deletion of relation
cache	addition	created
comments	addition	format, homepage, mail, name, thread
	deletion	link
directory	deletion	complete deletion of relation
feed	deletion	complete deletion of relation
forum	addition	tid, vid
	deletion	icon, shadow
	primary key change	nid removed from primary key
item	deletion	complete deletion of relation
locales	deletion	complete deletion of relation
moderation_filters	deletion	complete deletion of relation
moderation_roles	deletion	complete deletion of relation
moderation_votes	deletion	complete deletion of relation
node	addition	sticky
	deletion	attrs, body, revisions, score, static, teaser
	type change	type
page	deletion	complete deletion of relation
poll	deletion	voters
search_index	addition	fromsid, fromtype, score, sid
	deletion	count, lno
site	deletion	complete deletion of relation
statistics	rename	relation renamed to node_counter
system	addition	bootstrap, schema_version, throttle, weight
term_hierarchy	primary key change	parent and tid removed from primary key
term_node	primary key change	nid and tid removed from primary key
users	addition	access, created, login, picture
	deletion	homepage, hostname, rating, rid, session, sid, timestamp
	type change	language
variable	type change	name
vocabulary	addition	help, module, tags
	deletion	types
watchdog	addition	link, referer, severity
	type change	location

Table 4.2: Changes of database schema between Drupal 4.1.0 and 4.7.11

4.3.1 Research question 1: Does Data-Software Coupling Quality metric indicate which files change, using the rolled up per file value?

In each project, we measured the Data-Software Coupling Quality metric for each query, and we followed the changes that occurred in its value at the source code files contained in `module` folder. Additionally, we measured the number of queries in the files since they also changed -either increased or decreased. Notice that only two files of `module` folder have the same number of queries: `blog.module` and `tracker.module`, where the Data-Software Coupling Quality metric increased for both. Combining those metrics, we measure the average value of Data-Software Coupling Quality metric for each source code file, and we search whether it increased or not. If the majority of the project's files have an increased Data-Software Coupling Quality metric value at the major version, then we shall have an indication that our metric depicts the files that contain queries that are to change.

	Drupal 4.1.0			Drupal 4.7.11		
File	# Q	Perf. coupl.	AVG coupl.	# Q	Perf. coupl.	AVG coupl.
blog.module	6	2	0.87296	6	3	0.88085
book.module	15	4	0.86026	19	3	0.85566
comment.module	31	18	0.92009	25	15	0.93142
forum.module	16	4	0.75887	13	6	0.87683
locale.module	3	3	1	6	4	0.94285
node.module	7	4	0.91833	18	16	0.97537
poll.module	5	3	0.90148	7	4	0.91219
statistics.module	22	20	0.97760	12	7	0.91185
taxonomy.module	24	10	0.91266	23	7	0.87637
tracker.module	4	0	0.80944	4	1	0.98034
user.module	36	31	0.97353	29	24	0.97044
watchdog.module	2	0	0.80945	3	1	0.88570

Table 4.3: Files of `module` folder affected by the changes of Table 4.2

Using bold we depict the files that their metric increased and italics the files that their metric decreased. Apparently there were 7 out of the 12 files that their Data-Software Coupling Quality metric increased and 5 that their metric decreased. Because of that, we can not clearly state that the metric value performed as we expected. Nevertheless, we encountered some files that had a great increase on their values (more than 0.1) while the decrease was at max 0.06.

4.3.2 Research question 2: Does Data-Software Coupling Quality metric follow the Lehman's Lows of evolution, when a set of software maintenance steps occurred in the projects life?

Moving on, we further roll up of our metric. Figure 4.8 depicts the average Data-Software Coupling Quality metric value per project's folder. We can clearly state that the folder with the lowest value is the one that the changes of Table 4.3 pertain to. Those changes managed to increase the rolled up average value of the metric of the specified folder (`modules`), even-though the number of the queries involved was quite high (and furthermore increased), as Table 4.4 describes.

Queries of folder	Drupal 4.1.0	Drupal 4.7.11
./includes	8	40
./modules	218	294
./database	0	24
.	13	1

Table 4.4: Rolled up metric values

Moving on, we examine the overall roll up value of our metric, Table 4.5 depicts the values of perfect coupling queries and the average value of the projects, using all the queries present in the project's subfolders.

Figure 4.9 depicts the average value of the Data-Software Coupling Quality metric, over all the queries of the project. The Drupal project had been expanding for quite some time until major maintenance effort took place. Observe the behaviour of the metric at the points in which the developers started their code maintenance (4.6.11 (`https://www.drupal.org/project/drupal/releases?api_version%5B%5D=80`), where the developers mention: "Latest and greatest of the 4.6 releases. Also probably the last.

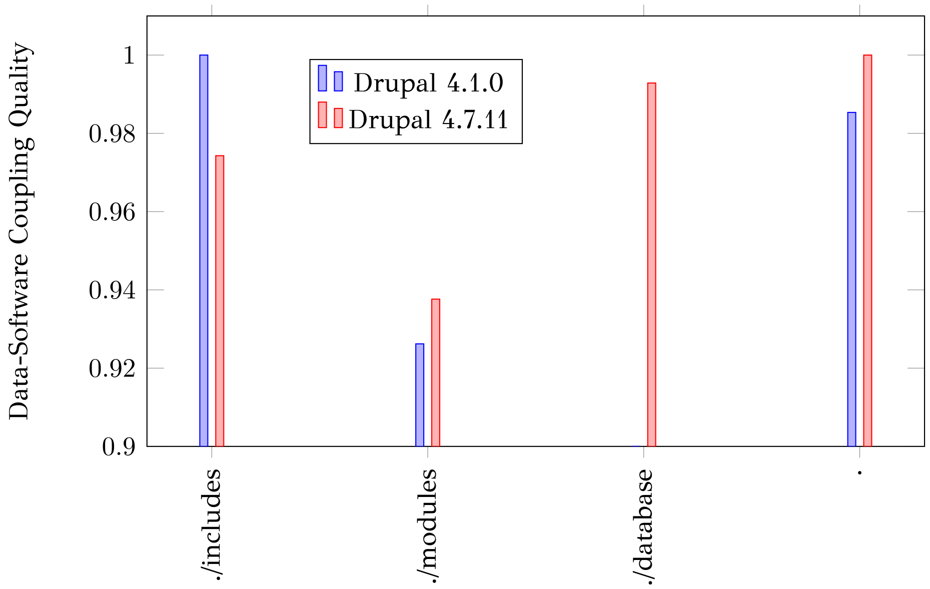


Figure 4.8: Average Data-Software Coupling Quality metric of each folder.

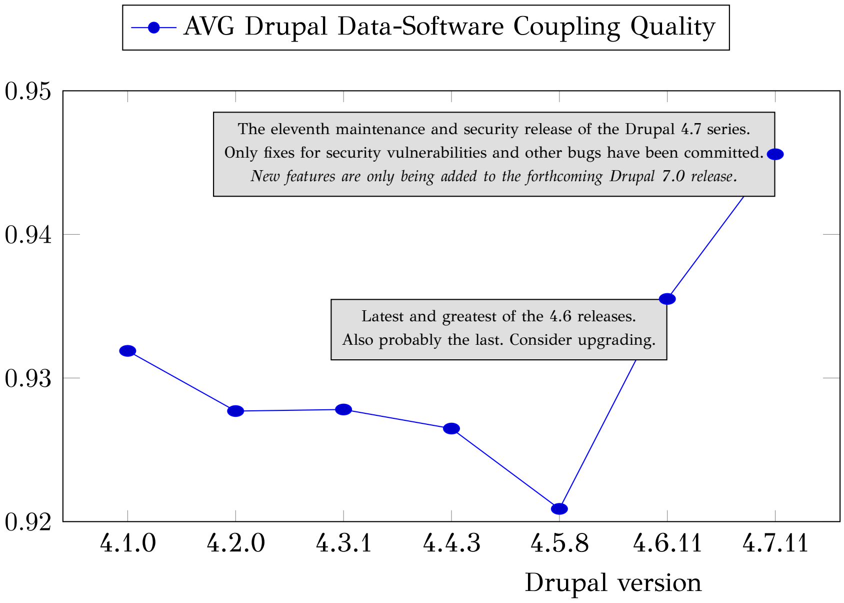


Figure 4.9: Average Data-Software Coupling Quality metric of each folder.

Version	#Q	Perf. coupl.	AVG coupl.
Drupal 4.1.0	239	163	0.93189
Drupal 4.2.0	246	168	0.927699186992
Drupal 4.3.1	259	244	0.927806949807
Drupal 4.4.3	262	247	0.926480916031
Drupal 4.5.8	283	267	0.920886925795
Drupal 4.6.11	331	228	0.93550755287
Drupal 4.7.11	359	256	0.94558

Table 4.5: Rolled up metric values

Consider upgrading." and 4.7.11 (`https://www.drupal.org/project/drupal/releases?api_version%5B%5D=79`), where the developers mention: "The eleventh maintenance and security release of the Drupal 4.7 series. Only fixes for security vulnerabilities and other bugs have been committed. *New features are only being added to the forthcoming Drupal 7.0 release.*"). Lehman's Law holds here, and in sync with it, our metric increases too.

4.4 Query Rewriting

In this section we propose an algorithm, that given a project and the queries that use that project, proposes database schema changes so as to produce easier to comprehend, maintain and debug queries.

The main idea is that the existing queries can be rewritten using views to perform the same tasks. The views do not have to be materialized as we have seen in Chapter 2, but simple view definitions, used for writing simpler, more reliable, and easier to understand code regarding the software developer point of view. Our algorithm starts by separating the queries by the number of the providers they use into different "buckets". Then, we sort those "buckets" on the number of the providers of each "bucket". After that, we traverse the "buckets" and rewrite the queries contained in the "bucket". To do so, we use any existing views and we also use the views that we created during the rewrite of a previous query (of the same or not "bucket"). This way, a view produced for a query of two providers, might be used in the rewrite of another query of two providers or it could even be a partial solution for a query that uses three or more providers etc. The final step is when we reach the "bucket" with the queries with the most number of providers. This technique is based on an observation we made in [1] where the projects are depicted in clustered graphs in circular layouts. In [1] we observed that the more "complex" queries are placed in the outer part of the circles clusters but the providers that were used were only a small part of heavily used providers.

The proposed algorithm (Alg. 4.1) is using the following steps:

1. Grouping the queries based on the number of inputs the have.

2. Starting with group of queries with number of providers > 1, we examine whether those queries can be rewritten with any existing view. This check is performed via inspecting the providers of the query and the providers of the view and we have a "positive" when and only when the two sets match. If any of the two sets (query providers and view providers) differs (not only having different providers, but also if the size of the providers is different), then we conclude that the query cannot be rewritten with the examined view.

3. When we find such a view, we rewrite the query using that view and we add the query to our output.

4. If no, the query is rewritten, using a simple view we create. The simple view performs just one task: it joins the inputs of the query, without using any filter besides the one given on the join equation, and provides all the available output via a simple to understand renaming: ⟨TABLE⟩_⟨COLUMN⟩. This way, the developers will only have to read the beginning of the output attribute to understand what they will use. Using the same logic, the name of the new view is the concatenation of the joined tables: ⟨TABLE⟩_⟨TABLE⟩.... Then, using the new provided view, the query is rewritten and we add it to the output of the queries. Apparently the query uses only one view for its task, therefore, the Data-Software Coupling Quality value will be a perfect "1".

This technique reuses the views that exist from previous execution steps. This way, when we examine the "clusters" that were introduced in [1] and further discussed in Chapter 6 (depicted in Fig. 4.10) we provide the smallest number of views for the given queries in a stratified way.

Algorithm 4.1: Rewrite of queries with views

```
Input: A set of existing views ($V$), and a set of queries ($Q$) with some
       coupling value.
Output: A final set of views ($V^f$), and a final set of rewritten queries ($Q^f$)
        with a better Data-Software Coupling Quality value.
$Map\langle Integer, List\langle Query\rangle\rangle\ mapJQ = \emptyset$ ;      ▷ Queries grouped by joins num.
$Q^f = \emptyset$ ;
foreach $q \in Q$ do
    $mpJQ[CountJoinsNumber(q)] += q$ ;
end
$mpJQ.sort()$ ;                    ▷ Start with least number of joins and grow.
$V^f = V$ ;
foreach $qs \in mapJQ$ do
    foreach $q \in qs$ do
        $View\ viewtoUse = CanBeRewritten(q, V^f)$;
        if $viewToUse \neq \emptyset$ then
            $Q^f += Rewrite(q, viewToUse)$ ;      ▷ Rewrite with existing view.
        else
            $viewToUse = CreateSimpleViewForQuery(q)$ ;          ▷ New view.
            $V^f += viewToUse$ ;
            $Q^f += Rewrite(q, viewToUse)$ ;          ▷ Rewrite using new view.
        end
    end
end
Procedure $CanBeRewritten(Query\ q, Views\ V)$
    foreach $v \in V$ do
        if $v.input == q.input$ then
            return $v$ ;             ▷ Providers of $q$ **exactly match** those of $v$.
        end
    end
    return $\emptyset$ ;
```

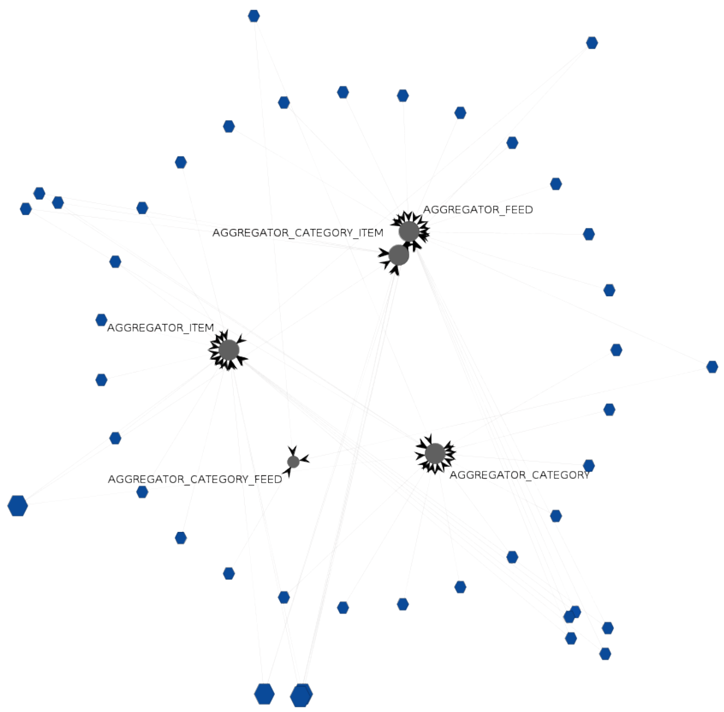


Figure 4.10: A "cluster" of queries using the same input providers. Blue nodes represent queries and the outermost cirlce is of queries that use more than one providers. Gray nodes are the providers, each one anotated with their name.

4.5 Query Rewriting Experiments

In this section we discuss the complexity that our method causes to the database schema. Via complexity we refer to the number of the additional views that are needed in order to achieve the best possible Data-Software Coupling Quality metric using the method described in Section 4.4.

Table 4.6 describes the characteristics (Relations, Views, Queries, Queries that use more than one Relations) of each of the Data Intensive Information Systems projects that we examined. The differences of the database schemata on those projects were minor (e.g. a rename of a table column, or a movement of a column from table A to table B). This is something expected, since the database schema is the part of the software that every other part relies on in order to gain access to data. Therefore, even a minor change of the schema could result into faults at the software. Those faults might be during the execution time, where the software might halt, or even worse, the faults could be semantic ones, meaning that the code runs without halting issues, but the results of the queries are faulty, which are harder to locate and fix. An example of such an error is the following: a provider table has the classes that the students passed (having >= 5 grade), and the developer wants the average of each student. When for some purpose (eg. for space issues) the provider table is dropped, the developer has to use another one that contains grades, which is the one that contains all the classes (passed or not). The developer has to adapt his code so as to provide the correct results (via crossing out the results that have grade < 5).

Project	Relations	Views	Queries	Queries to rewrite
Drupal 4.1.0	38	0	240	76
Drupal 4.2.0	38	0	247	83
Drupal 4.3.1	40	0	260	90
Drupal 4.4.3	40	0	263	92
Drupal 4.5.8	52	0	284	71
Drupal 4.6.11	55	0	332	107
Drupal 4.7.11	57	0	358	101
OpenCart	115	0	651	122
ZenCart	106	0	150	7

Table 4.6: Data Intensive Information Systems projects

Project	Views	Rewrite failures	Project metric increase	Developer gain
Drupal 4.1.0	28	1	6.7%	62%
Drupal 4.2.0	26	1	7.1%	67%
Drupal 4.3.1	29	1	7.2%	67%
Drupal 4.4.3	29	1	7.2%	67%
Drupal 4.5.8	26	11	4.2%	48%
Drupal 4.6.11	37	3	6%	63%
Drupal 4.7.11	40	2	8%	58%
OpenCart	53	4	3.2%	53%
Zencart	4	2	0.6%	14%

Table 4.7: Data Intensive Information Systems project measurements. Due to parsing issues we were unable to rewrite all the queries. The corresponding column depicts the number of queries we failed to rewrite. The The developer gain column describes in percentage how many queries the developer avoids to examine due to views existence.

In Table 4.7 we depict the number of the views we added to the database schema, using Algorithm 4.1. Due to limitations in our query parsing software, there were occasions where the produced view, or the rewritten query was unable to be parsed. We were able to catch those exceptions and we stopped our algorithm as soon as we encountered such an exception, leaving the query in its original state. Table 4.7 depicts the number of the queries that we failed to rewrite and remained intact. Additionally, Table 4.7 depicts the gain of a developer who has to adapt his queries when a schema change occurs. The definition of the developer gain is described in equation 4.3.

Definition 4.5.1. *The Developer Gain $DG(q|v)$ of a rewritten Data Intensive Information Systems is defined as*

$$\begin{aligned} DG(q|v) &= 1 - (CostWithView/CostWithoutViews) \rightarrow \\ &= 1 - ((FailedRewrites + Views)/QueriesNeedingRewrite) \end{aligned} \tag{4.3}$$

Without the views the developer would have to check all the queries that use a relation, when the relation changes. Using Algorithm 4.1 we observe that the developer would always need a smaller amount of effort since Alg. 4.1 creates new views only if the existing ones cannot be used to rewrite the queries that have non perfect Data-Software Coupling Quality metric value. Having the views, a developer / database administrator examines a significantly smaller number of code parts that might produce errors due to schema changes. The developer gain measurement (query checking and rewriting) is at worst case 14%, while at best case it is 67%. A 50% developer gain measurement describes that each one of the views of a Data Intensive Information Systems project is used by two queries, while a 67% developer gain measurement describes that each one of the views of this Data Intensive Information Systems project is used by three queries more or less! In most cases, the examined projects provided a developer gain measurement above 50%. The worst case scenario would be having always failure to rewrites (where the $Views$ would be 0, and $FailedRewrites$ equal to $QueriesNeedingRewrite$) which would give a DG equal to 0, or when each one of the $QueriesNeedingRewrite$ would produce a unique view (never used by any other query), which would produce as many $Views$ as $QueriesNeedingRewrite$ so the DG would also be 0.

Returning to the schema / graph of Fig. 4.1, we conclude that using Alg. 4.1 we managed to minimize the edges that come from software part (left of figure), since we have only 1 edge for each query, via adding edges internally to the database part (right of figure), via adding intermediate representations of providers.

4.6 Conclusions

The problem of metrics in areas other than object oriented software is still open. In this chapter, we have just done the first steps to describe such a metric for the data-intensive ecosystems.

More particularly, we described the *fundamental properties* of what a well structured database related code is, and we proposed a *scalable metric that identifies badly constructed database related software*. In the experiments we conducted, we observed that *our metric follows the software evolution maintenance steps of the code* (when the developers maintain their code, our metric increases, when the developers insert new features without maintenance, our metric decreases).

Additionally, since the database schema does not change easily because it is a part of the ecosystem that many components rely on, we proposed a rewriting algorithm for the software parts (queries) and the database schema (view) of a project. The algorithm changes the database schema via adding –only when it is absolutely necessary– views to the schema joining more than one relations, and then rewrites the queries that depended on those relations, to depend on the newly created views. As we observed in the experiments we conducted, *the proposed algorithm provided better Data-Software Coupling Quality metric measurements and we additionally managed to increase the developer's evolution gain*, since there exist less points a developer has to check when a schema change occurs.

As next step, we check whether the rewriting solution (of view creations and query rewrites) can help the developers and the database administrators regulate an evolution step in a data-intensive ecosystem, and provide a smooth transaction from one database schema to another, avoiding either syntactic or semantic inconsistencies that a schema change could produce.

Chapter 5

Regulation of Schema Evolution with Policies

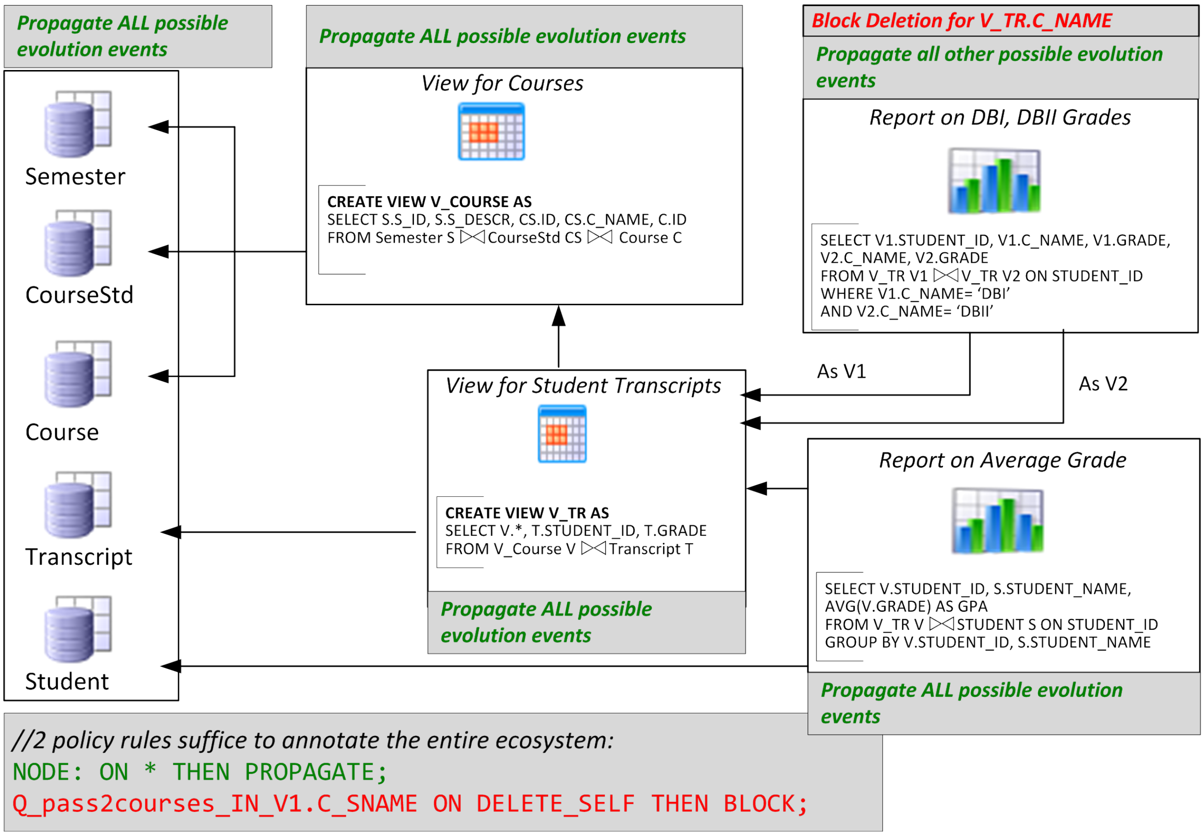
5.1 Introduction

A data-intensive ecosystem is a conglomeration of software modules that includes (a) at least one central database (typically, but not obligatorily, relational), and, (b) a set of software applications that require access to the central database via queries embedded in their code. The distinctive feature of data-intensive ecosystems is the cohesive management of databases and applications – plainly put, the management of the database has to profoundly take its surrounding applications into account (and vice versa). In this chapter, we deal with the problem of facilitating the evolution of an ecosystem without impacting the smooth operation or the semantic consistency of its components.

To operate smoothly, an ecosystem must withstand change gracefully. Software maintenance comprises 60% of the resources spent on building and operating a software system [95] and thus, it is of utmost importance for a system's life-cycle. In this context, the management of changes in a data-centric ecosystem is an important problem. In this chapter, we extend the state of the art concerning several research questions in the area of managing the evolution of data-intensive ecosystems.

What does evolution of data-intensive ecosystems comprise? We start by example – here are a few examples of possible changes:

- A certain attribute of the schema of a view is about to be deleted, as the administrator wants to simplify the definition of the view
- A new attribute is added to a relation, because new content is available
- The WHERE clause of a view is modified with an extra condition, to account for a new definition of the view's contents


Propagate ALL possible evolution events
Semester
CourseStd
Course
Transcript
Student
Propagate ALL possible evolution events
View for Courses
Block Deletion for V_TR.C_NAME
Propagate all other possible evolution events
Report on DBI, DBII Grades
As V1
As V2
View for Student Transcripts
Propagate ALL possible evolution events
Report on Average Grade
Propagate ALL possible evolution events

```
CREATE VIEW V_COURSE AS
SELECT S.S_ID, S.S_DESCR, CS.ID, CS.C_NAME, C.ID
FROM Semester S ⋈CourseStd CS ⋈ Course C
```

```
SELECT V1.STUDENT_ID, V1.C_NAME, V1.GRADE,
V2.C_NAME, V2.GRADE
FROM V_TR V1 ⋈V_TR V2 ON STUDENT_ID
WHERE V1.C_NAME= 'DBI'
AND V2.C_NAME= 'DBII'
```

```
CREATE VIEW V_TR AS
SELECT V.*, T.STUDENT_ID, T.GRADE
FROM V_Course V ⋈Transcript T
```

```
SELECT V.STUDENT_ID, S.STUDENT_NAME,
AVG(V.GRADE) AS GPA
FROM V_TR V ⋈STUDENT S ON STUDENT_ID
GROUP BY V.STUDENT_ID, S.STUDENT_NAME
```

```
//2 policy rules suffice to annotate the entire ecosystem:
NODE: ON * THEN PROPAGATE;
Q_pass2courses_IN_V1.C_SNAME ON DELETE_SELF THEN BLOCK;
```

Figure 5.1: An exemplary University-DB Ecosystem, annotated with policies.

Taking the aforementioned examples at a more abstract level, we claim that *evolution is of significance if it affects the syntactic correctness*, *the semantic validity*, *the* *operational effectiveness, or the administrative overhead of a data-intensive ecosystem*. The most disordering (and also visible) type of impact is *syntactic impact*: in this case, a change might drive parts of the ecosystem to be syntactically inconsistent and thus fail. A deleted attribute might cause applications using it to crash. In this case, the applications' developers have to take care of the change: pinpoint its impact in their code, assess the necessity for the existence of this information in the applications and modify their applications accordingly. If things go wrong, this might even require negotiations with the DBAs to restore the deleted attribute. Second, applications can be affected *semantically*. If a new attribute is added to a relation it is possible that it contains important information that applications should be exploiting (and thus, have to be synchronized to the new contents of the relation). If the semantics of a view change, then the data delivered at the view's clients are different than the ones delivered before: in this case, the developers of the affected queries and applications would have to be notified and decide on whether the queries have to adapt to the new semantics of the view, or, they would have to retain the old semantics (again leading to the problem of compensating the change performed by the DBAs). A third type of impact (that falls outside the scope of this chapter) involves the effect of a change to the performance and administrations of the ecosystem. Dropping an index may result in a large number of queries running unacceptably slow or moving a table may result in making less space for other tables to perform their insertions.

In all these occasions, we observe that a change performed by the DBA team can have several side-effects both for the team itself, the developers of applications of the ecosystem and the end-users. *The problem in all the aforementioned events is that the change is performed before assessing its impact over the ecosystem*. Therefore, addressing the impact assessment problem in advance of a potential change can be really valuable.

How can we assess the impact of a change in a data intensive ecosystem? Is it possible to regulate change in a data-intensive ecosystem? In this chapter, we improve the state of the art with concrete results for the problem of impact assessment. We follow the model of *Architecture Graphs* [28, 31] that capture all the inter-dependencies between the constructs of databases and the application queries via a graph. The graph models constraints, attributes, relations, views and queries along with their internal structure as the nodes of the graph. The edges of the graph denote dependency for data provision (e.g., between a view and a relation that populates it with data), part of relationships (e.g., between a relation and its attributes) and semantic relationships (e.g., the construction of the WHERE clause of a query as a tree of expressions). This way, the Architecture Graph models all the components of a data-intensive ecosystem in a uniform way. One of the main utilities of the Architecture Graph is that it facilitates impact assessment for potential changes in the ecosystem: whenever a potential change is tested over the Architecture Graph, the graph allows us to identify the area of impact by recursively following edges between affected nodes. Practically speaking, each node has to assume a status concerning its reaction to an event that we test; once a status is assumed, subsequent nodes of the graph have to be notified too.

At the same time, we are not helpless in managing potential changes in the core of the ecosystem. If an application developer is really adamant on retaining the structure and semantics of a database view, is it possible that this requirement is incorporated in the Architecture Graph, to prevent possible modifications? As previous research [29, 28] has demonstrated, it is possible to regulate the flow of events by annotating the modules of the Architecture Graph with policies, i.e., rules for handling events. Specifically, we can annotate a module (i.e., relation, view or query) with a policy for each possible event that it can withstand, in one of two possible modes: (a) *block*, to veto the event and demand that the module retains its previous structure and semantics, or, (b) *propagate*, to allow the event and make the module adapt with an updated internal structure. Once the adaptation is complete, the module is also responsible for igniting the recursive notification of all the affected software modules in the graph.

To make the discussion a little more concrete, we present an evolving data-intensive ecosystem in Figure 5.1. On the left, we depict a small part of a university database with three relations and two views, one for the information around courses and another for the information concerning student transcripts. On the right, we isolate two queries that the developer has embedded in his applications, one concerning the statistics around the database course and the other reporting on the average grade of each student. If we were to delete attribute C_NAME, the ecosystem would be affected in two ways : (a) *syntactically*, as both the view V_TR and the query on the database course would crash, and, (b) *semantically*, as the latter query would no longer be able to work with the same selection condition on the course name. Similarly, if an attribute is added to a relation, we would like to inform dependent modules (views or queries) for the availability of this new information. Observe the two policy rules at the bottom of the figure. The first one dictates that every node of the graph adapts to any evolutionary event that appears in the future. The rule uses two shorthands: the term `NODE` refers to all the nodes of the graph and the term $*$ refers to any potential event that arrives. The second rule overrides the first global policy by stating that the report on the upper right has a veto over the deletion of one of the attributes exported by the view on student transcripts (V_TR). In Figure 5.1, we have used a lightly shaded box to show how these rules are assigned to each module.

Once the impact of a change has been assessed, is it possible to see how the ecosystem will look like if the change is eventually performed? Even with the presence of policies, it is possible that a potential modification in the database affects several queries and views that are willing to accept it and adapt to the new structure or semantics of the database. The problem becomes more complicated whenever a change ignites different reactions – e.g., some of the affected queries are willing to adapt whereas others assume a vetoing status. Then, the question that has to be answered is "what will the new structure and semantics of all the affected modules look like?". As we will show, the answer to the question is not straightforward and unfortunately, the state of the art in ecosystem adaptation is not sufficient to address it. Specifically, although previous work in ecosystem adaptation has provided us with techniques for view adaptation [38], [37], [96], the existing works do not allow the definition of policies for the adaptation of the ecosystem modules. At the same time, our own previous work [28] has proposed algorithms for impact assessment with explicit policy annotation but without the mechanisms to perform the rewriting of the ecosystem. Overall, to the best of our knowledge, there is no method that allows both the impact assessment and the rewriting of the ecosystem's modules along with correctness guarantees.

To address this shortcoming, the core result of this chapter is the provision of algorithms that identify which parts of the ecosystem are affected by a potential change and perform the rewriting of affected modules to adapt to it, while fulfilling all the constraints imposed by the -possibly conflicting- policies of all affected modules. Specifically, our method works in the following three steps:

1. Impact assessment. Given a potential event, a status determination algorithm makes sure that the nodes of the ecosystem are assigned a status concerning (a) whether they are affected by the event or not and (b) what their reaction to the event is (block or propagate).

2. Conflict resolution and calculation of variants. Assume a view used by two queries is altered. Assume also that the first query vetoes the change and requires the structure and semantics of the old view to remain, whereas the second concedes to the change and states it will adapt to the new structure and semantics of the view. *The co-existence of blocker and adapter data consumers of an affected module signifies the need to retain both the old and the new version of the module, whenever this is possible*. To this end, we introduce an algorithm that checks the affected parts of the graph in order to highlight affected nodes with whether they will adapt to a new version or retain both their old and new variants.

3. Module Rewriting. Once the status and number of variants has been determined for the modules of the graph, we need to implement the rewritings. This is heavily dependent upon the nature of the event (obviously, a query adapts differently to the removal of an attribute and differently to the addition of an attribute, let alone changes in semantics). Our algorithm visits affected modules sequentially and performs the appropriate restructuring of nodes and edges.

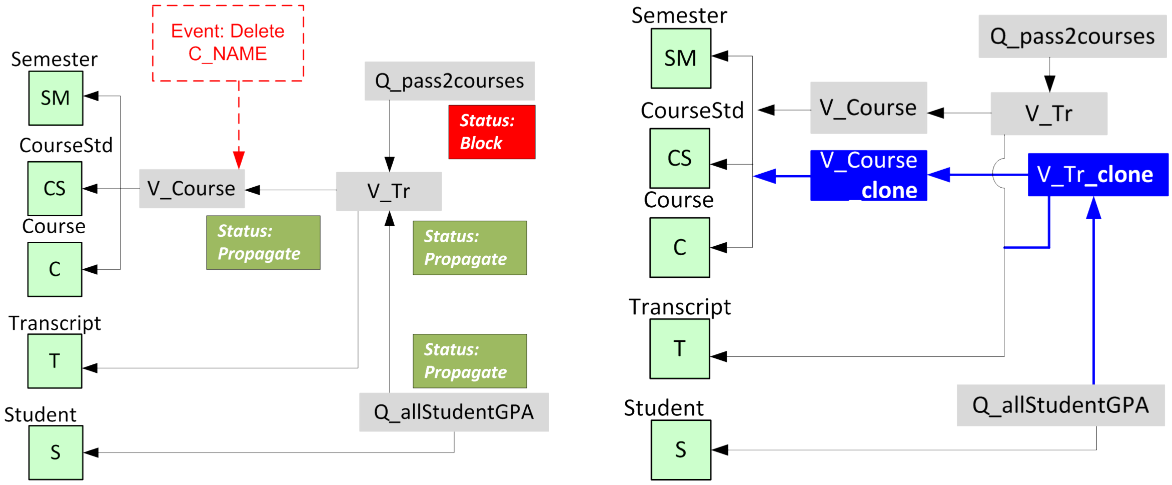


Figure 5.2: Impact analysis (left) and ecosystem rewriting (right) for an event on our exemplary ecosystem

Coming back to our motivating example, let's see what happens when the DBA of the ecosystem tries to delete attribute C_NAME from the intermediate view V_COURSE. As instructed by its policy, the view "agrees" to adapt to the event and adopts a *Propagate* status. Then, it notifies its consumer V_TR which also agrees and pushes the event to its consumers, specifically, Q_pass2courses which vetoes the event and assumes a *Block* status and Q_allStudentGPA which is actually unaffected by the event after it self-examines whether it is affected. The rest of the modules of the graph, and specifically, the source relations, have a status NO_STATUS as the propagation of the event has never reached them. The depiction of the status determination part is shown in the left part of Figure 5.2. Then, our method detects a conflict, as the view V_TR decides to adapt to the event in contrast to the veto from the application developer of the query Q_pass2courses. Once this conflict is detected, a cloning mechanism is initiated to satisfy both requirements. The result is depicted in the right part of Figure 5.2. The query Q_pass2courses retains the old definition of both views (i.e., the entire backwards path till the node initiating the event), whereas the two views are cloned and these clones (depicted in darker colors in the figure) are adapted to satisfy the requirement set by the DBA.

We have implemented our method in a *what-if analysis* tool, Hecataeus[1] where all stakeholders can pre-assess the impact of possible modifications before actually performing them, in a way that is loosely coupled to the ecosystem's components. We have assessed our method (Sec. 6.4) over ecosystems of increasing size and complexity and also varied the policy assignments in order to assess the method's scale up with size and the effect of the policy annotation to the method's usefulness. Our first experimental goal involved assessing the effectiveness of our method, i.e., the benefits introduced by our method concerning the effort performed by the application developers and administrators of the ecosystem. The results indicate that in the absence of our system, the typical developer would have to perform at least 21% of routine, useless checks to views and queries that are not related to the event at all; on average, the number of useless checks is located in the area of 90%-97%. A second observation has to do with the amount of rewriting: in all occasions, there have been several modules that had to be rewritten. Although the average numbers are not particularly high, ranging from 2 to 13 modules depending on the experimental setup, the maximum numbers are quite high and, in any case, the automation of the work, equips the involved stakeholders with correctness guarantees that would otherwise be non-existent. Another significant observation has to do with the conciseness of the policy annotation rules. The number of policy rules is practically equivalent to the number of the exceptions to the default policies (resulting in just a handful of rules in our experiments). In terms of efficiency, all the experiments show a completion of

[1] http://www.cs.uoi.gr/~pvassil/projects/hecataeus/

the tested changes as small fractions of a second. At the same time, the chosen policy significantly affects the spreading of the impact of a change over the ecosystem: a policy with early containment of the event (by blocking it inside the database) can be an order of magnitude faster than a policy that blocks changes at the queries only. At the same time, the graph size is linearly related to the time needed to complete the impact analysis and rewriting task. Overall, our experimentation with ecosystems of different policies and sizes indicates that our method offers significant effort gains for the maintenance team of the ecosystem and, at the same time, is executed fast and scales gracefully.

5.2 Formal Background

To assess the impact of a potential change over the data centric ecosystem, we construct a graph of modules (relations, queries and views) where data consuming nodes are linked with edges to their providers. Whenever an event is applied over a module, the module has to assess the impact of the event and notify its consumers. This recursive process allows us to assess the impact of the event over the entire ecosystem. Naturally, to facilitate this process, we need to establish a formal model for the main constituents of the problem and its solution. So, before proceeding with the algorithmic parts of the adaptation process, in this Section, we present the formal background for the modeling of Architecture Graphs, along with the space of possible events and policy annotations. First, we present how relations, views and queries construct the Architecture Graph of the ecosystem. Then, we move on to present the space of possible events that can be applied to the nodes of the graph, either directly by the user (initiating the entire process of assessing the impact of an event) or transitively, as modules affected by the event notify other modules that depend on them for the change. Moreover, in order to regulate the propagation of events over the graph, we present the language for policy annotations, along with its semantics and the rules for policy overriding.

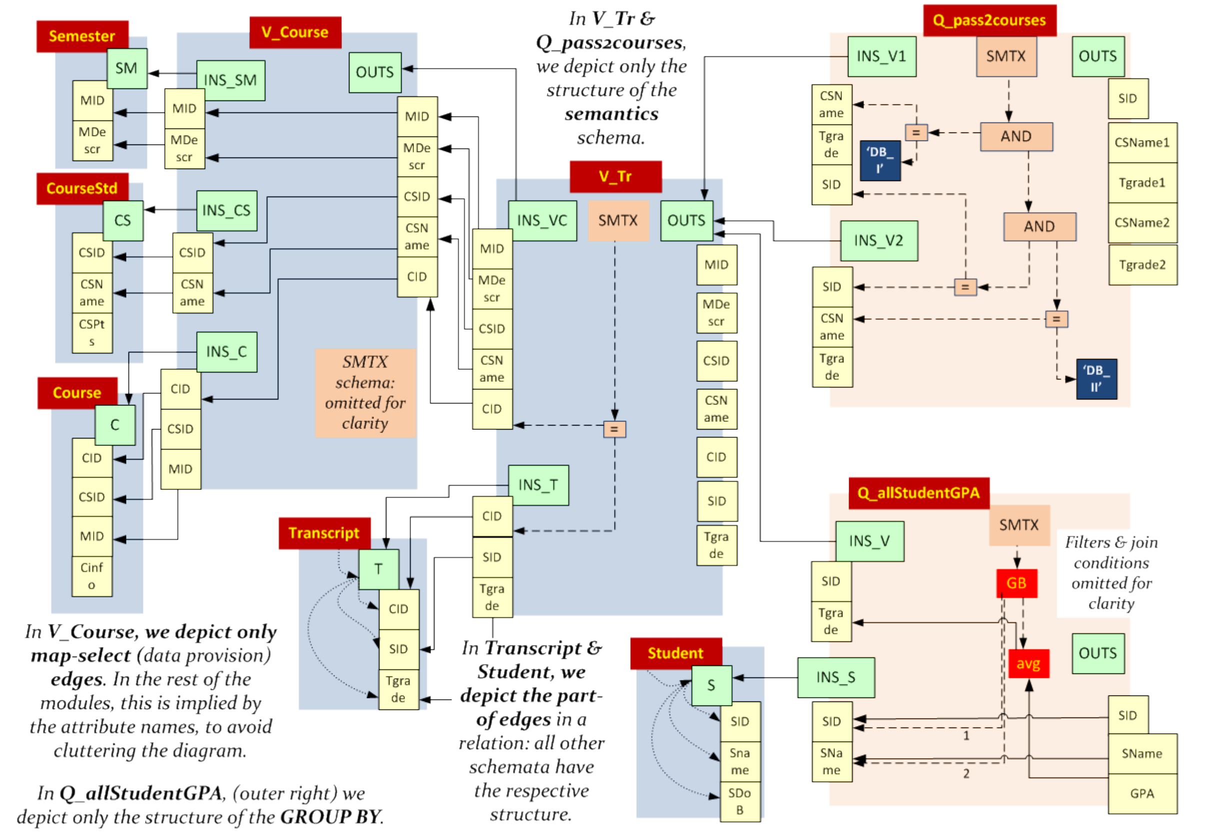
5.2.1 Architecture graph

Our modeling technique, following [97], represents all the aforementioned database constructs as a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, which we call the *Architecture Graph* of the ecosystem. For the reader who is not interested in all the formalities, the following quick summary along with Figures 5.2 and 5.3 should be sufficient to allow the understanding of our graph modeling.

- *Relations, views and queries (or else modules) come with a subgraph, that includes (a) a node for the module itself, (b) a set of input schemata for views and queries, used for linking these modules with their data providers, (b) an output schema for the data exported by the module and (d) a semantics schema for any filtering or restructuring taking place inside a view or a query (WHERE, GROUP BY, etc).*
- *Input and output schemata include their respective attributes; semantics schemata include a tree representing the logical expression of the WHERE clause and a list of groupers for the GROUP-BY clause, in case these exist in a query or view.*
- *Edges of the graph signify dependency on data provision: at the schema level, input and output schemata are linked with data dependency edges from data consumers towards data providers; the respective holds for attributes of the schemata too. Note that this mechanism applies both between modules (inter-module edges) and within the same module (intra-module edges). Semantic-related edges are also used for the constructs related to the semantics schema within views and queries.*

The reader who wants to skip the detailed description of the graph can jump to Section 5.2.2. If this is not the case, our deliberations proceed with a presentation of the components of the Architecture Graph as follows. We start with the high level constructs, such as relations and queries, which we call modules of the Architecture Graph, and then we move on to discuss their main properties. Fig. 5.3 visually represents the internals of the modules of Fig. 5.1. To avoid overcrowding the figure, we omit different parts of the structure in different modules; the figure is self-explanatory on this.

Modules. A module is a semantically high level construct of the ecosystem; specifically, the modules of the ecosystem are relations, views and queries. Every module defines a scope recursively: every module has one or more schemata in its scope (defined by part-of edges), with each schema including components (e.g., the attributes of a schema or the nodes of a semantics tree) linked to the schema also via part-of edges. In our model, all modules have a well defined scope, “fenced” by input and output schemata.



In ***V_Tr*** *&* ***Q_pass2courses****, we depict only the structure of the* ***semantics*** *schema.*

SMTX schema: omitted for clarity

In ***V_Course, we depict only map-select*** *(data provision)* ***edges****. In the rest of the modules, this is implied by the attribute names, to avoid cluttering the diagram.*

In ***Transcript*** *&* ***Student, we depict the part-of edges*** *in a relation: all other schemata have the respective structure.*

Filters & join conditions omitted for clarity

In ***Q_allStudentGPA****, (outer right) we depict only the structure of the* ***GROUP BY****.*

Figure 5.3: A subset of the graph structure for the University-DB Ecosystem.

Relations. Each relation R (A_1, A_2,..., A_n) in the database schema is represented as a directed graph, which comprises:

1. a node, R, representing the relation;

2. *an output schema node*, R_SCHEMA, representing the relation's output schema;

3. n attribute nodes $A_{i=1...n}$, one for each of the attributes and,

4. n+1 schema relationships $E_{i=1...(n+1)}$, directing from the schema node towards the attribute nodes, indicating that the attribute belongs to the relation's output schema and one directing from the relation node towards the output schema node indicating that the output schema belongs to the relation.

In our reference examples, we have the following relations, whose graphs are depicted in Fig. 5.3): relation $Semester$(MID, $MDescr$) standing for information on semesters, relation $CourseStd$($csid$, $csname$, $cspts$) standing for information on courses, relation $Course$(cid, $csid$, mid) standing for information on courses on semesters, relation $Student$(sid, $sname$) standing for information on students,and relation $Transcript$(cid, sid, $tgrade$) standing for information on the enrolment of students to courses and their grades.

Queries and Views. The graph representation of a Select - Project - Join - Group By (SPJG) query involves:

1. a new node representing the query, named *query node*,

2. a set of *input schemata nodes* (one for every table appearing in the FROM clause). Each input schema includes the set of attributes that participate in the syntax of the query (i.e., SELECT, WHERE clause, etc.)

3. an *output schema node* comprising the set of attributes present in the SELECT clause.

4. a *semantics* node as the root node for the sub-graph corresponding to the semantics of the query (specifically, the WHERE and GROUP-BY part), and,

5. *attribute nodes* belonging to the various input and output schemata of the query.

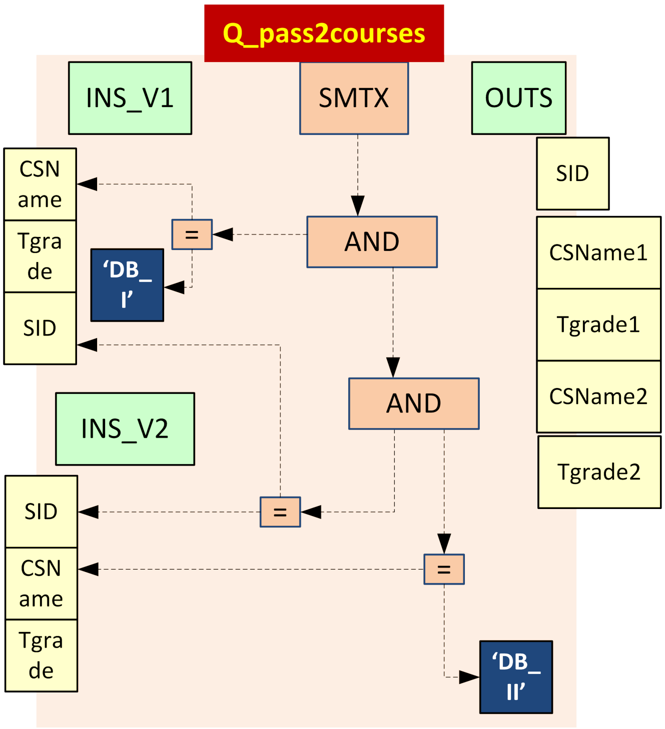
The query graph is therefore a directed graph connecting the query node with the high level schemata and semantics nodes. The query node contains an edge towards every schema it possesses. The schema nodes are connected to their attributes via part-of relationships. In order to explain the internal structure of a query, we structure our presentation of the query's graph in terms of its SQL parts: SELECT, FROM, WHERE, and GROUP BY, each of which is eventually mapped to a sub-graph.

Select part. Each query is assumed to own an output schema that comprises the attributes, either with their original or with alias names, appearing in the SELECT clause. In this context, the SELECT part of the query maps the respective attributes of the input schemata to the attributes of the query's output schema through *map-select* edges, directing from the output attributes towards the input schema attributes.

From part. The FROM clause of a query can be regarded as the relationship between the query and the relations (or views) involved in this query. Thus, the relations included in the FROM part are combined with the input schemata of the query node through *from* edges, directing from the nodes of the appropriate input schemata towards the output schema nodes of the relation/view nodes. The input schemata of the query comprise only the attributes of the respective relations that participate in any way in the query; the attributes of the input schemata are connected to the respective attributes of the provider relations or views via map-select relationships.

Where part. We assume that the WHERE clause of a query is in conjunctive normal form. Thus, we introduce a directed edge, called *where* edge, starting from the semantics node of a query towards an operator node corresponding to the conjunction of the highest level. Then, there is a tree of nodes hanging from this conjunction involving condition nodes (to be defined right away). The edges are operand relationships as mentioned above among binary comparators, boolean operators, input attributes and constants. In Fig. 5.4, we depict the graph of query $Q\_pass2courses$, which performs a self-join over view V_TR and presents a report of the students that enrolled in both DB_I and DB_II courses, and their grades. A tree, starting from the $SMTX$ node, describes the conditions of the selected tuples. Initially, we take the tuples where the name of the first course is equal to DB_I, then we filter them and take the ones that have the same SID for $V1$ and $V2$. Finally, we filter those results and take the ones having the name of the second course equal to DB_II.

We consider three classes of atomic conditions that are composed through the appropriate usage of an operator *op* belonging to the set of classic binary operators, *op* (e.g., $<$, $>$, $=$, $\leq$, $\geq$, $\neq$, IN, $EXISTS$, ANY): (i) Ω *op* constant, (ii) Ω *op* Ω', and (iii) Ω *op* Q where Ω, Ω' are attributes of the underlying relations and Q is a query. A



```
SELECT V1.SID, V1.CSNAME AS CSNAME1,
        V1.TGRADE AS TGRADE1,
        V2.CSNAME AS CSNAME2,
        V2.TGRADE AS TGRADE2
FROM V_TR V1, V_TR V2
WHERE V1.C_NAME= 'DBI'
        AND V1.SID = V2.SID
        AND V2.C_NAME= 'DBII'
```

Figure 5.4: The graph of the semantics schema for the Q_pass2courses query

condition node is used for the representation of the condition. Graphically, the node is tagged with the respective operator and it is connected to the operand nodes of the conjunct clause through the respective operand relationships, O. Composite conditions are easily constructed by tagging the condition node with a Boolean operator (e.g., AND or OR) and the respective edges to the conditions composing the composite condition.[2]

Group By part. The GROUP BY part is mapped in the graph via (i) a node GB, to capture the set of attributes acting as the aggregators and (ii) one node per aggregate function labeled with the name of the employed aggregate function; e.g., COUNT, SUM, MIN. For the aggregators, we use edges directing from the semantics node towards the GB node that are labeled *group-by*. The GB node is linked to the respective input attributes acting as aggregators with *group-by* edges, which are additionally tagged according to the order of the aggregators; we use an identifier i to represent the i-th aggregator. Moreover, for every aggregated attribute in the query's output

[2] Well-known constraints of database relations – i.e., primary/foreign key, unique, not null, and check constraints – can also be captured by this modeling technique. Foreign keys are subset relations of the source and the target attribute, check constraints are simple value-based conditions. Primary keys, which are unique-value constraints, are explicitly represented through a dedicated node tagged by their names and a single operand node.

schema, there exists a *map-select* edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective input attribute. In Fig. 5.5, we depict the graph of query $Q\_allStudentGPA$. In the left part, we have the edges that connect the output attributes with their providers in the input schemata. We have SID and $SName$ that are using as their providers the SID and $SName$ of $Semester$ relation, whilst the GPA is the $AVERAGE$ aggregate function of $TGrade$ coming from V_TR view. In the right part of the figure, we have the GB node, which is used to describe the "group by" clause of the query. The numbers on the edges depict the order of the groupers, meaning that first we group by SID and then with $SName$ columns. Additionally, in $MSTX$ node, we have a node that describes that in the resulting tuples of the query, the SID that comes from V_TR view and the SID that comes from $Semester$ relation should be equal to each other.

```
SELECT V.SID, S.SNAME,
          AVG(V.TGRADE) AS GPA
FROM V_TR V,  STUDENT S
WHERE V.SID = S.SID
GROUP BY V.SID, S.SNAME
```

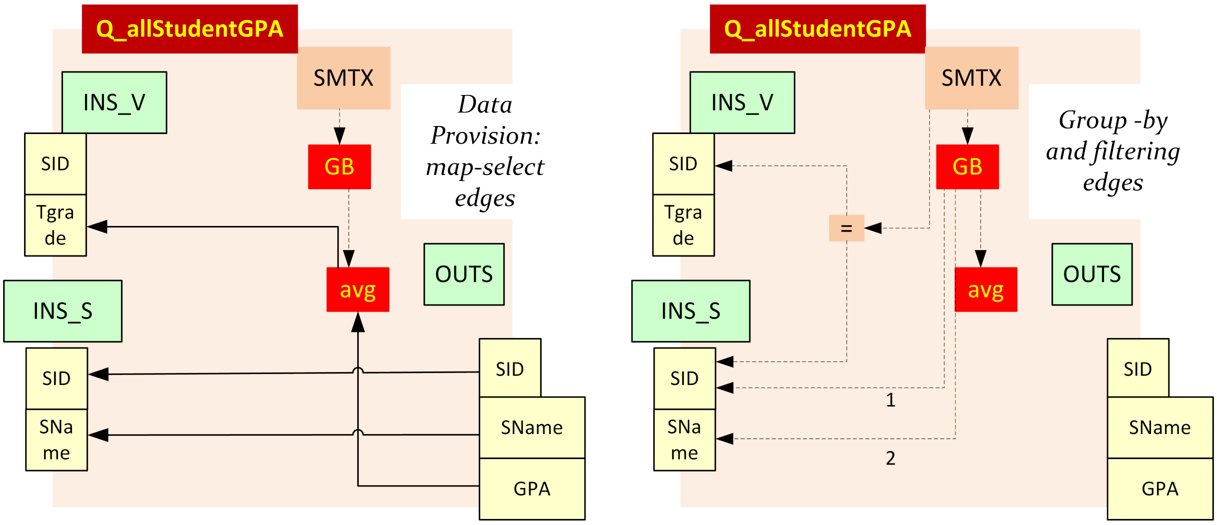


Figure 5.5: The graph of a group-by query. To avoid confusion, we depict the edges in two snapshots of the graph: provider edges (left) and filtering and grouping edges (right).

Views. Views are treated as queries; however the output schema of a view can be used as input by a subsequent view or query module.

Summary. A summary of the architecture graph is a zoomed-out variant of the graph at the schema level with provider edges only among schemata (instead of attributes too). Formally, a *summary graph* is a directed acyclic graph $\mathbf{G}_s$ = $(\mathbf{V}_s, \mathbf{E}_s)$, with $\mathbf{V}_s$ comprising the graph's module nodes (relations, views and queries) and $\mathbf{E}_s$ including an edge e(v,u) from a consumer module v to a provider module u if and only if there is an edge between an input schema of v and the output schema of u in the Architecture Graph. We can formally define different levels of zooming via summary graphs (i) at the schema level with input/output schemata, (ii) the module level as in Fig. 5.6.

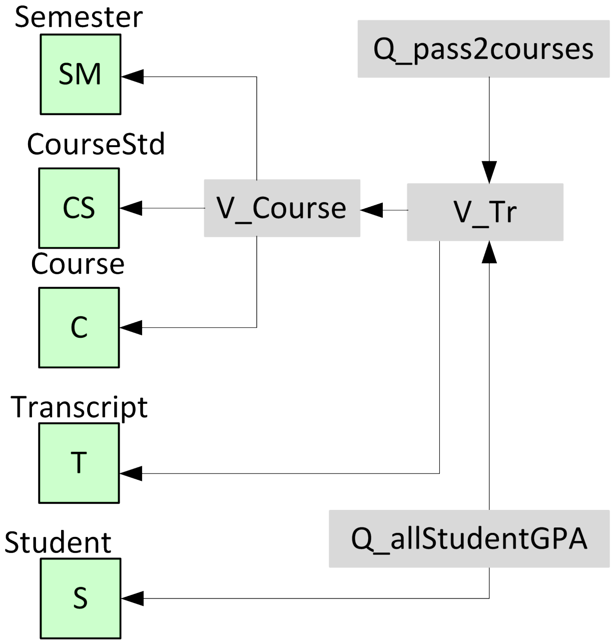


Figure 5.6: The Summary Graph of the University-DB Ecosystem.

5.2.2 Events

In this section we list the set of possible events that our method handles. We organize our discussion by classifying these events in three classes: (a) events pertaining to relations, (b) events pertaining to views or queries, and (c) events that occur as one module notifies another for the event it just received.

We can classify the impact of an event as *structural* whenever the exported schemata and their attributes are changed in terms of structure or naming. At the same time, the impact of an event is *semantic* whenever the internals of the semantics schema (i.e., the WHERE or the GROUP-BY clause of the respective SQL query) change.

Events that pertain to relations. The first class of events comprise changes on the schema of relations:

- *ADD_ATTRIBUTE*: in this case, a relation should obtain another column
- *DELETE_ATTRIBUTE*: in this case, a relation should drop a column
- *RENAME_ATTRIBUTE*: in this case, a relation should rename a column
- *DELETE_SELF*: in this case, a relation will be deleted
- *RENAME_SELF*: in this case, a relation will be called with a new name from now on.

Events that pertain to views and queries. The second class of events involve changes on the definitions of Views/Queries:

- *ADD_ATTRIBUTE*: in this case, a query or a view should have a new attribute (column, aggregate function or value) in its output
- *DELETE_ATTRIBUTE*: in this case, a query or a view should have less attributes in its output
- *RENAME_ATTRIBUTE*: in this case, an attribute is going to be called with a new name from now on
- *DELETE_SELF*: in this case, a view will be deleted (deleting queries is of no impact to the ecosystem anyway)
- *RENAME_SELF*: in this case, a view will be called with a new name from now on
- *ALTER_SEMANTICS*: in this case, a view is going to have another WHERE clause or another GROUP BY clause.

Events that pertain to the notification of a change between modules. As we will see in Section 5.3, whenever a module has decided on its reaction against the incoming events, it assumes a status and notifies subsequent modules. Thus, besides the aforementioned events, we need to support the following list of events that accrue from the flow of an event to the graph.

- *ADD_ATTRIBUTE_PROVIDER*: this event is generated by a module in order to inform its consumers that the module has added an attribute to its output schema.

- *DELETE_PROVIDER*: this event is generated by a module in order to inform its consumers that this module has deleted one or all its attributes.
- *RENAME_PROVIDER*: this event is generated by a module to inform its consumers that the module itself or one of the attributes that exist in output schema of the module want to change their name.
- *ALTER_SEMANTICS*: this event is generated by a module to inform its consumers that the semantics (as described previously: change of WHERE or/and GROUP BY clause) of a module have changed.

5.2.3 Policies

Our basic tool for the regulation of the propagation of an event's impact to the entire ecosystem is the ability to block further propagation at certain modules which veto the event. To achieve this, we employ *policies* that annotate the ecosystem's modules with predefined reactions to all possible incoming events they can receive. This way, whenever a node receives an event that concerns either itself or its constituents (e.g., the attributes of a schema), the node has already been instructed by the ecosystem's administrator on its reaction to the incoming event. The policy of a node for responding to an incoming event can be one of the following:

- PROPAGATE, which means that the node accepts the change and will adapt to the new reconfiguration of the ecosystem, or,
- BLOCK, which means that the node wants to retain the previous structure and semantics.

Requirements for policy annotation. We wish to provide a *language* that annotates nodes with policies and addresses the following usability requirements:

- *Completeness*: how can we be sure that we can define annotations for *all* the possible events that can arrive to a node, for all the nodes of the ecosystem?
- *Conciseness*: can we achieve this *easily* and *correctly* with respect to the user's intentions, without having the user going to great lengths of coding in order to annotate the ecosystem with policies?

Completeness. To achieve completeness, we need to be sure that we can provide an annotation for all the nodes of the graph and for all the events that each node can receive. To achieve this, we proceed in two steps: (a) we explicitly define the *node-event space*, i.e., the space of all valid combinations of nodes and incoming events, and (b) for each node-event combination, we define the respective *policy rule* that characterizes the reaction of the node to this event.

To implement the first of the aforementioned steps, we exhaustively enumerate all combinations of events and nodes (see Table 5.1). Observe, that Table 5.1 provides *a complete characterization of events that can arrive to a node organized per event type*. In Table 5.1, the rows (actually corresponding to the <receiver node> part of the above rule) are explained as follows:

1. [QUERY|VIEW].[OUT|IN].SELF standing for the node representing the output (input) schema of all queries (views)

2. [QUERY|VIEW].[OUT|IN].ATTRS standing for the nodes representing the attributes of the output (input) schema of all queries (views)

3. [QUERY|VIEW].SMTX.SELF standing for the root node of the semantics tree of all queries (views)

4. RELATION.OUT.[SELF|ATTRS] standing for the node representing the output schema of all relations (or its attributes)

Language for policies. Then, to implement the translation of the node-event space to policy rules, we need to provide a language that determines the policy for each event that appears to each node. The language that we introduce is used to *assign policies* to all the nodes of the ecosystem with guarantees for the complete coverage of *all* the graph's nodes along with syntax conciseness and customizability. In a nutshell, the main idea is the usage of rules of the form <receiver node [type]> : on <event> then <policy>, both at the default level –e.g.,

```
VIEW.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
```

and at the node-specific level (overriding defaults) –e.g.,

```
V_TR_OUT.SELF: on ADD_ATTRIBUTE then BLOCK;
```

			ADD		DELETE		RENAME		ALTER
				ATTR					
			ATTR	PROV	SELF	PROV	SELF	PROV	SMTX
QUERY	OUT	SELF	✓	✓	✓		✓		
		ATTRS			✓	✓	✓	✓	
	IN	SELF		✓		✓		✓	✓
		ATTRS				✓		✓	
	SMTX	SELF							✓
VIEW	OUT	SELF	✓	✓	✓		✓		
		ATTRS			✓	✓	✓	✓	
	IN	SELF		✓		✓		✓	✓
		ATTRS				✓		✓	
	SMTX	SELF							✓
RELATION	OUT	SELF	✓		✓		✓		
		ATTRS			✓		✓		

Table 5.1: The space of events that can be received by each node type

Before formally specifying the syntax of the policy language, we first discuss the issues of language conciseness and rule overriding.

Conciseness. The observant reader might wonder on the reasoning behind providing rules both at the node type and the node level. The reason is conciseness: we want to avoid annotating the graph in a node per node, event per event basis. To this end, we provide a language that comes with the simple semantics that unless otherwise specified (see the next paragraph), each node-event pair implements the respective *node type - event type* policy. The default, fixed list, comprising 33 rules that can be derived from the entries of Table 5.1 is depicted in Fig. 5.7. In Section 5.4, we provide a proof for the language completeness in Theorem 5.1.

Still, even so, the number of rules needed for completeness could be considered too large by some users. To this end, we provide some additional rules that simplify our policy language. These rules come as syntactic sugar to our language. Specifically, we introduce two syntactic sugar extensions as follows:

- the * notation for events allows the user to specify that a specific module type (i.e., all relations/views/queries of the ecosystem) of a specific node is annotated with the same policy for all the events that occur to it. In other words, the * notation signifies “for any incoming event”
- the NODE notation specifies that all nodes of the ecosystem, independently of their type, are annotated with the specified policy for the specified event (if, of course, the event pertains to the node).

Of course, the combination of the two syntactic shorthands is also allowed. Thus, we end up with the following list of syntactic sugar extensions:

<**moduleType**>: **ON * THEN** <**policy**>; This rule groups the events that a module type (RELATION, VIEW, QUERY) can receive and sets the policy for all these events to <policy>.

<**namedNode**>: **ON * THEN** <**policy**>; This rule finds the node that is specified by name <namedNode> and sets the policy for all these events to <policy>.

NODE: **ON** <**event**> **THEN** <**policy**>; This rule annotates all the nodes of the graph that can receive the specified event (named <event>) with the same policy, namely <policy>.

NODE: **ON * THEN** <**policy**>; This rule actually replaces the group of the 33 rules to one simple rule, saying that regardless of the event, the policy is uniformly set to <policy>.

Theorem 5.3 in Section 5.4 describes why these extra rules correctly cover up the needed events and correctly assign the policies to the nodes.

Customizability and Rule Overriding. Whereas our small list of generic, default rules can cover all possible combinations of events and node types, it is quite possible that we want to define a different reaction to the same event for different modules. For example, we might wish a certain view to block attribute addition, whereas we would allow another view to adapt to the same event. To facilitate this possibility we allow three *layers* of rules:

1. Layer 0: Rules that are applied to all the nodes of the *Architecture Graph* via the <NODE> notation.

2. Layer 1: General rules at the node type level, about *all* modules and their attributes.

3. Layer 2: Rules that apply to all the attributes of a *specific schema*.

4. Layer 3: Rules that apply to *specific attribute* nodes.

In our approach, the semantics of the layers of rules state that *each layer overrides the policy of its previous layers*. This way, if we have a default policy for all relations (layer 1) for a certain event, we can customize the behavior of a specific relation to be different than the default by defining a specific rule for it (layer 2). Theorem 5.2 in Section 5.4 proves that our overriding mechanism assigns the correct policy to each node. Within each of the layers, the following ordering is imposed:

1. First, the * notation is transformed to the appropriate list of rules.

2. Second, any more specific rules override the * notation with their designated policies.

Language Syntax. The language's syntax comprises rules that abide to the following structure:

<receiver> : on <event> then <policy>

where:

1. <receiver> can be any of the ecosystem's node types

2. <event> can be any of the events that can arrive to an instance of this node type, either because the user initiated this as the starting event, or due to the propagation of the event in the ecosystem

3. <policy> can be either PROPAGATE or BLOCK

The above list of possible rules covers the node type layer (Layer 0), but not the two others. To this end, we introduce two extra kinds of potential values for the <receiver> part of the rules of our language.

1. <NAMED SCHEMA NODE>.ATTRIBUTES standing for the nodes representing the attributes of the <named schema node> of the graph.

2. <NAMED NODE> standing for the <named node> node of the graph.

The first of the two extra rules refers to all the attributes of a specific schema (layer 2), and, the second one refers to individual nodes of the graph (layer 3).

Reference Example. Returning to our reference example, the following text represents a set of rules of how policy rules should be written in order to have all nodes of the graph propagating all possible events for all modules, except for V_TR view, in which only the CID attribute will propagate any of its incoming events. Fig. 5.8 covers the first set of completeness-ensuring rules mentioned previously.

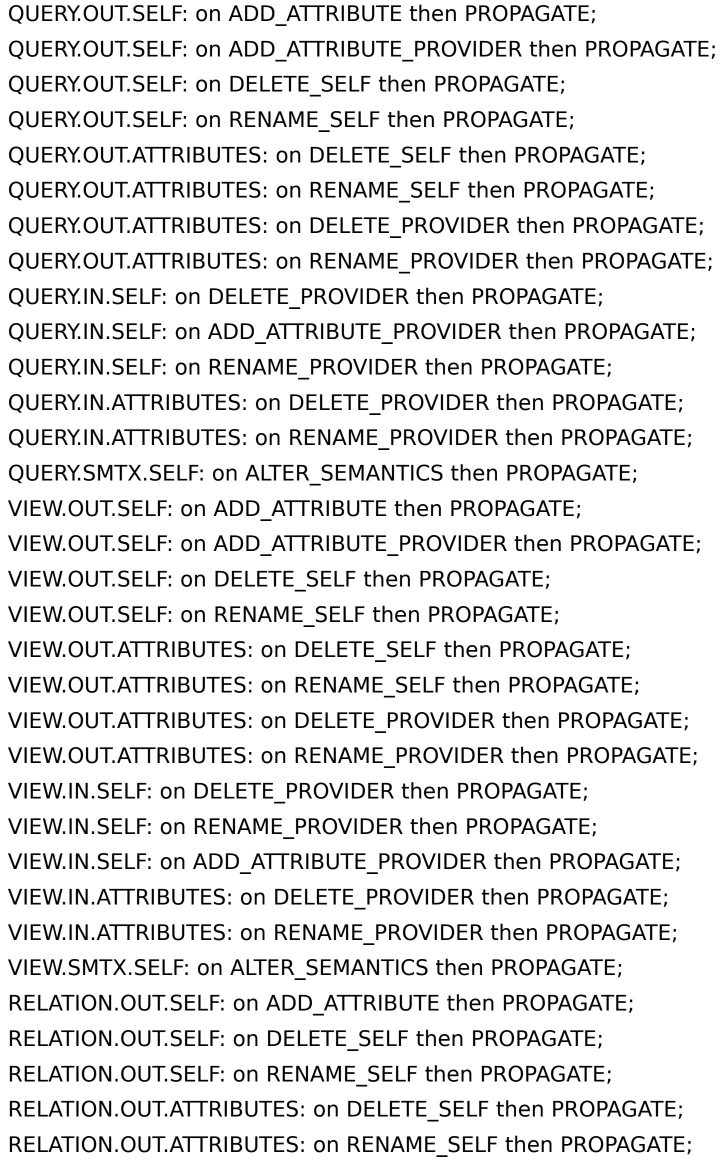
Assuming now that the user wanted for the view V_TR to have a BLOCK policy for all possible events, Fig. 5.9 describes the set or rules needed to be issued after the general rules of Fig. 5.8.

Finally, the user decided that there is an exception to the rules of Fig. 5.9, and the attribute CID of the output schema of the V_TR module should have *again* a different policy than its siblings (switching again to PROPAGATE instead of BLOCK that was set in the previous set of rules), for its deletion. This is achieved by the set of policies depicted in Fig. 5.10.

Using the additional rules that simplify our policy language, the same example could be written as Fig. 5.11 describes.

1. QUERY.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
2. QUERY.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
3. QUERY.OUT.SELF: on DELETE_SELF then <policy>;
4. QUERY.OUT.SELF: on RENAME_SELF then <policy>;
5. QUERY.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
6. QUERY.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;
7. QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
8. QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
9. QUERY.IN.SELF: on DELETE_PROVIDER then <policy>;
10. QUERY.IN.SELF: on RENAME_PROVIDER then <policy>;
11. QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
12. QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
13. QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
14. QUERY.SMTX.SELF: on ALTER_SEMANTICS then <policy>;
15. VIEW.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
16. VIEW.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
17. VIEW.OUT.SELF: on DELETE_SELF then <policy>;
18. VIEW.OUT.SELF: on RENAME_SELF then <policy>;
19. VIEW.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
20. VIEW.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;
21. VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
22. VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
23. VIEW.IN.SELF: on DELETE_PROVIDER then <policy>;
24. VIEW.IN.SELF: on RENAME_PROVIDER then <policy>;
25. VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
26. VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
27. VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
28. VIEW.SMTX.SELF: on ALTER_SEMANTICS then <policy>;
29. RELATION.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
30. RELATION.OUT.SELF: on DELETE_SELF then <policy>;
31. RELATION.OUT.SELF: on RENAME_SELF then <policy>;
32. RELATION.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
33. RELATION.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;

Figure 5.7: The 33 combinations of events and node types that provide complete graph coverage; *policy* can be either BLOCK or PROPAGATE



```
QUERY.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
QUERY.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
QUERY.OUT.SELF: on DELETE_SELF then PROPAGATE;
QUERY.OUT.SELF: on RENAME_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on DELETE_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on RENAME_PROVIDER then PROPAGATE;
QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
QUERY.SMTX.SELF: on ALTER_SEMANTICS then PROPAGATE;
VIEW.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
VIEW.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
VIEW.OUT.SELF: on DELETE_SELF then PROPAGATE;
VIEW.OUT.SELF: on RENAME_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on DELETE_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on RENAME_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
VIEW.SMTX.SELF: on ALTER_SEMANTICS then PROPAGATE;
RELATION.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
RELATION.OUT.SELF: on DELETE_SELF then PROPAGATE;
RELATION.OUT.SELF: on RENAME_SELF then PROPAGATE;
RELATION.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
RELATION.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
```

Figure 5.8: Application of default rules for our reference example

```
V_TR_OUT.SELF: on ADD_ATTRIBUTE then BLOCK;
V_TR_OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then BLOCK;
V_TR_OUT.SELF: on DELETE_SELF then BLOCK;
V_TR_OUT.SELF: on RENAME_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on DELETE_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on RENAME_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_OUT.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on DELETE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on RENAME_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on ADD_ATTRIBUTE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.SELF: on DELETE_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.SELF: on RENAME_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.SELF: on ADD_ATTRIBUTE_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_SMTX.SELF: on ALTER_SEMANTICS then BLOCK;
```

Figure 5.9: Overriding the default rules for a view in our reference example

```
V_TR_OUT.CID: on DELETE_SELF then PROPAGATE;
V_TR_OUT.CID: on DELETE_PROVIDER then PROPAGATE;
```

Figure 5.10: Overriding the default rules for an attribute in our reference example

```
NODE: on * then PROPAGATE;
V_TR: on * then BLOCK;
V_TR_OUT.CID: on DELETE_SELF then PROPAGATE;
V_TR_OUT.CID: on DELETE_PROVIDER then PROPAGATE;
```

Figure 5.11: Simplified policy language example

5.3 Impact Assessment and Adaptation of Ecosystems

The goal of our method is to assess the impact of a hypothetical event over an Architecture Graph annotated with policies and to adapt the graph to assume its new structure after the event has been propagated to all the affected modules. Before any event is tested, we topologically sort the modules of the architecture graph (always feasible as the summary graph is acyclic: relations have no cyclic dependencies and no query or view can have a cycle in their definition). This is performed once, in advance of any impact assessment. Then, in an on-line mode, we can perform what-if analysis for the impact of changes in two steps that involve: (i) the detection of the modules that are actually affected by the change and the identification of a status that characterizes their reaction to the event, and, (ii) the rewriting of the graph's modules to adapt to the applied change.

5.3.1 Topological sort

In order to make sure that the messages between modules are transferred in the right order from providers to consumers, we perform a topological sorting of the graph's modules prior to any other step. As Theorem 5.4 in Section 5.4 indicates, this is always feasible as the Architecture Graph does not contain cycles.

We follow a traditional approach to our topological sorting, which proceeds as follows: first we find the modules with zero incoming edges. These modules are removed from the examination set along with their outgoing edges, after being assigned a unique ID. This gives as a result a new set of modules with zero incoming edges. The algorithm stops when there are no more modules to visit. Relations have the smallest IDs, followed by views and queries.

Observe that topological sorting of the graph is necessary, as opposed to a simple flooding of messages with events over the graph, due to the existence of multiple paths from data providers to their consumers (e.g., observe in Fig. 5.3 how the query Q_pass2Courses is fed by view V_TR via two paths, as it performs a self-join). Also, the existence of policies (which we detail in Section 5.3.3) require a strict order for visiting the nodes of the graph. Apart from the termination of our algorithms, we also want to guarantee the following properties:

- *Confluence*: each module in the graph will assume the same status, independently from the order of processing the incoming messages.

Algorithm 5.1: Topological sort

```
Input: A summary of an architecture graph $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$ that comprises the
       modules of an architecture graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$.
Output: A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, i.e. an
        annotation of the modules of $\mathbf{G_s}$ with a sequential id's, via a
        mapping $Y : \mathbf{V_s} \rightarrow \mathbf{N}$.
$notYetVisited \ = \ (Relation \cup View \cup Query)$;
$algoID \ = \ $size($notYetVisited$);
$module \ = \ null$;
while $notYetVisited \ > \ 0$ do
    find $module$ with zero incoming edges from $notYetVisited$;
    remove $module$ from $notYetVisited$;
    remove edges starting from $module$;
    $module.ID \ = \ algoID$;
    $algoID \ = \ algoID \ - \ 1$;
end
```

- *Consistency*: all the modules will be correctly rewritten.

In Theorem 5.6 and Theorem 5.11 in Section 5.4, we demonstrate why we need the principled visit of the nodes of the graph in a manner obeying the topological sort; had we not followed the topological sort it would be impossible to guarantee these correctness properties. Therefore, in the rest of our deliberations, unless explicitly mentioned, the propagation of the impact of events follows the topological sort.

Once the topological sort has been completed, we are ready to interactively work with the user towards highlighting the impact of a change and rewriting the graph accordingly. These two tasks are explained in the following two subsections.

5.3.2 Detection of affected nodes and status determination

The assessment of the impact of an event to the ecosystem is a process that results in assigning every affected module with a *status* that characterizes its policy-driven response to the event. In contrast to the policy, which is an annotation of each module with a directive on how to respond to a potential future event, a status is the decided reaction to an actual event, after it has reached the module. The status determination task is reduced in (a) determining the affected modules in the correct order, and, (b) making them assume the appropriate status. Algorithm *Status Determination* (Alg. 5.2) details this process. In the following, we use the terms *node* and *module* interchangeably.

1. Before assessing the event, all modules are set to status NO_STATUS. At the end of the algorithm's execution, the modules that will have retained this status will be the ones that have not been affected by the event.

2. Whenever an event is assessed, we start from the module over which it is assessed and visit the rest of the nodes by following the topological sorting of the modules to ensure that a module is visited after *all* of its data providers have been visited. A visited node assesses the impact of the event internally (cf., "intra-module processing") and, if there is reason to change its NO_STATUS status, due to incoming notifications from its providers, it obtains a new status, which can be one of the following: (a) BLOCK, meaning that the module is requesting that it remains structurally and semantically immune to the tested change and blocks the event (as its immunity obscures the event from its data consumers), (b) PROPAGATE, meaning that the module concedes to adapt and propagate the event to any subsequent data consumers.

3. If the status of the module is PROPAGATE, the event must be propagated to the subsequent modules. To this end, the visited module prepares *messages* for its data consumers, notifying them about its own changes. These messages are pushed to a common *global message queue* (where messages are sorted by their target module's topological sorting identifier).

4. The process terminates when there are no more messages and no more modules to be visited.

Intra-module processing. A module starts by retrieving from the global queue *all* the messages containing the events that concern it. For message processing within each module, a *local queue* is used. The processing of the messages is performed as follows:

1. First, the module probes its schemata for their reaction to the incoming event, starting from the input schemata, next to the semantics and finally to the output schema. Naturally, relations deal only with the output schema.

Algorithm 5.2: Status determination

```
Input: A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a global
        queue $Q$ that facilitates the exchange of messages between modules.
Output: A list of modules $Affected\ Modules \subseteq \mathbf{V_s}$ that were affected by the
        event and acquire a status other than $NO\_STATUS$.
$Q$={$original\ message$}, $Affected\ Modules = \emptyset$;
forall $node \in \mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$ do
    $node.status\ =\ NO\_STATUS$;
end
while size(Q) > 0 do
    visit module ($node$) in head of $Q$;
    insert $node$ in $Affected\ Modules$ list;
    get all messages, $Messages$, that refer to $node$;
    SetStatus($node$, $Messages$);
    if $node.status\ ==\ PROPAGATE$ then
        insert $node.Consumers\ Messages$ to the $Q$;
    end
end
return $Affected\ Modules$   Procedure SetStatus ($Module,\ Messages$)
    $Consumers\ Messages\ =\ \emptyset$;
    forall $Message \in\ Messages$ do
        decide status of $Module$;
        put messages for $Module$'s consumers in $Consumers\ Messages$;
    end
```

2. Within each schema, the schema probes both itself and its contained nodes (attributes) for their reaction to the incoming event. At the end of this process, the schema assumes a status as previously discussed.

3. Once all schemata have assumed status, the output schema decides the reaction of the overall module; if any of the schemata raises a veto (BLOCK) the module assumes the BLOCK status too; otherwise, it assumes the PROPAGATE status.

4. Finally, in case a PROPAGATE status is assumed, it prepares and inserts into the global queue appropriate messages for all its consumers.

Observe that a module may receive multiple messages. Typically this is due to the following two reasons: (a) cases of self-join, where a provider feeds (directly or indirectly) a consumer via multiple one paths (and thus, a change in the provider concerns more than one schemata of the consumer – observe here that it is not obligatory that these schemata have identical reaction towards the event) and (b) a deletion of an attribute in a view might affect both the semantics and the output schema of the view, producing thus, two messages to its consumers: one that notifies that output attributes have changed and another notifying that the semantics of the view has changed (e.g., a part of the SELECT clause has been dropped due to the attribute deletion).

Message structure and content. Each message msg is a quadruple $msg(n, s, e, p)$ with the following parts:

- n is the recipient module of the message.
- s is the specific schema of n, to which the message is sent (note that due to this information, we can also find who the sender of the message was, since an input schema has exactly one provider)
- e is the event that this message carries.
- p are message parameters containing additional information needed for some events (e.g., the new name of an attribute for attribute addition or renaming events).

All possible evolution events (as presented in Section 2.2) performed on relations, views and queries generate initial messages that fall into the following types:

- DELETE_ATTRIBUTE: the user deletes an attribute from the output schema
- RENAME_ATTRIBUTE: the user renames an attribute from the output schema.
- ADD_ATTRIBUTE: the user adds another attribute to the output schema of a module.
- DELETE_SELF: the user deletes a whole module.
- RENAME_SELF: when the user renames a whole module.
- ALTER_SEMANTICS: the user changes the semantics of a module.

Once the module has determined its reaction, it constructs messages for its data consumers. The contents of the messages depend on the type of event. Here, we list some examples of such cases.

- When a message is processed saying that an attribute is going to be deleted, the input schema of the consumers that are connected to that attribute is informed that the attribute will be deleted.
- If the whole module is going to be deleted then the consumers of this module will receive a message in their input schema saying that the provider of that input schema is going to be deleted.
- Likewise, when an attribute is going to be renamed, the input schema of the consumers that are connected to that attribute is informed that the attribute will have a new name from now on.
- If the whole module is going to be renamed, then the consumers of this module will receive a message in their input schema saying that the provider of that input schema is going to be renamed.
- When a module processes a message saying that a new attribute is going to be added to its output schema, it informs all of its consumers in their input schema that a new attribute was added to their provider.
- Finally when a module processes a message saying that its semantics have changes, it informs all its consumers that it changed its semantics.

Maestros for the local processing. To facilitate the local, independent nature of message processing by the modules, each module awakes a maestro that handles the probing of schemata as well as the decision making on what status will the schema assume. A *maestro* is a simple piece of software (implemented as an abstract interface, later materialized on a case by case basis) that is specialized on the combination $type\ of\ event \times module\ type$. For each type of module, there is a specialized maestro that takes care of status determination and rewriting for each possible event that can be received.

In terms of software architecture, the decision for this structuring of the code was done in order to decentralize event processing. It allows the reasonably smooth extension of the architecture with new types of events or modules at the price of some code reusability. In terms of algorithmic principles, we gain the benefits of module independence at the price of a common queue of messages.

In [98], we present how events are processed inside modules, organized by the type of the incoming message that the module is called to handle. For each event, we explain the structure of the incoming message and the list of steps that have to take place (organized per schema, if more than one schemata of the module are involved).

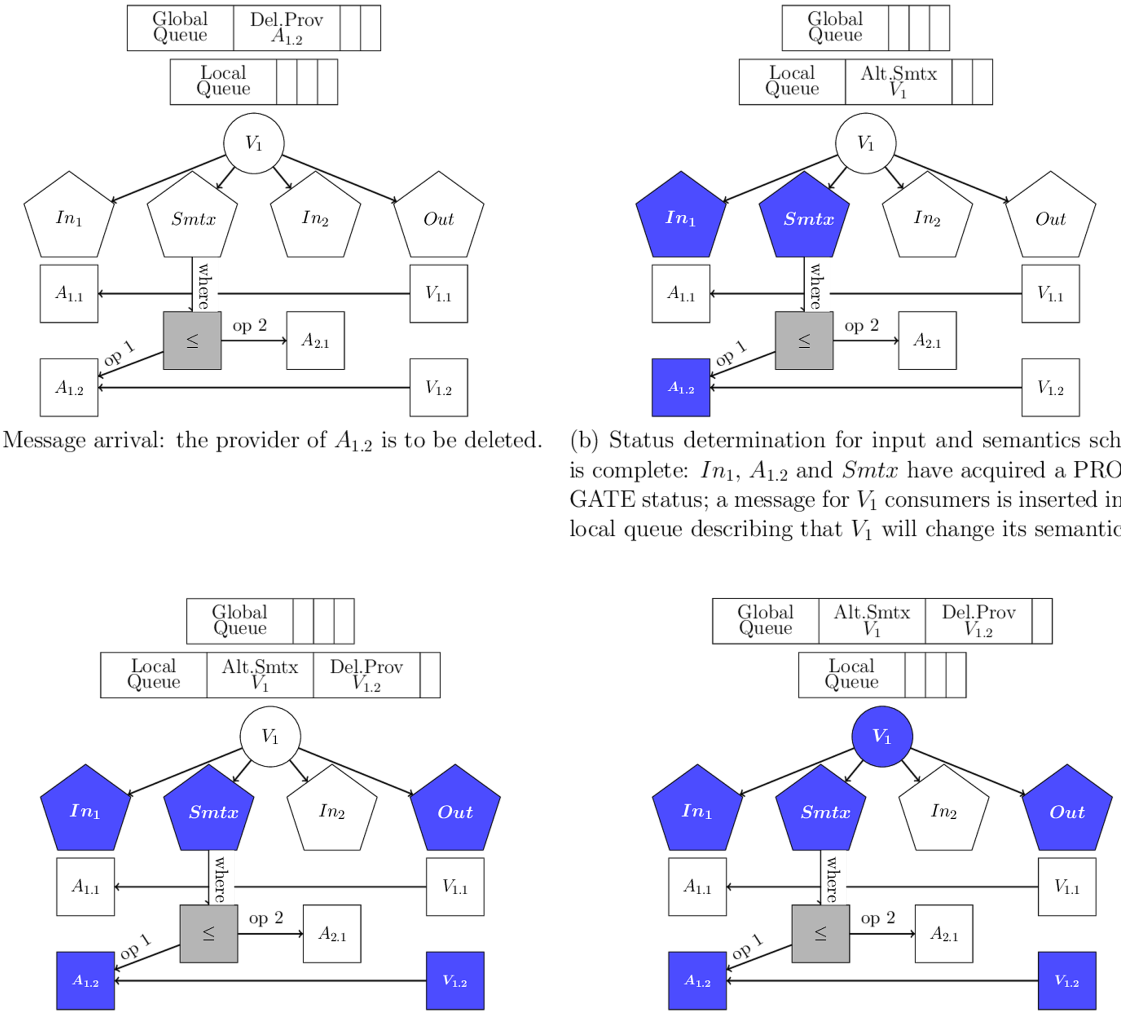
Assume a message with a provider attribute deletion event for the attribute named $A_{1.2}$, that a view module V_1 receives, as depicted in Figure 5.12 (a).

- Initially, the maestro of the V_1 module will find the attribute with name $A_{1.2}$ in the input schema that fetched the message to the module, denoted as $A_{1.2}$, too. Then, $A_{1.2}$ checks its policy for the event (*provider attribute deletion*) and acquires a status. The same status is assumed for the input schema node of the module V_1 as well. If there is any connection between $A_{1.2}$ and the semantics schema, then the semantics schema checks its policy for the alter semantics event, assumes a status, and creates messages for V_1's consumers, that describe that the semantics of V_1 will change. The newly created messages are kept in a local message queue of the maestro, as depicted in Figure 5.12 (b).

- Then, if there are attributes in the output schema of V_1 that are connected with $A_{1.2}$ (denoted as $V_{1.2}$), the maestro checks their policy for the event and acquires for each one a status. The output schema node of the module V_1 acquires a status as well. Finally, the maestro, for each of the $V_{1.2}$ attributes finds their consumers so as to notify them that their provider attributes are to be deleted. Those messages are also kept in the local message queue of the maestro, as depicted in Figure 5.12 (c).

- When all the above reach to an end, the V_1 module checks the statuses of the input, semantics, and output nodes. If none of them has acquired a BLOCK status, then the module acquires status PROPAGATE and notifies the consumers of V_1 about the change, by inserting all the messages of the local message queue in the global message queue, as depicted in Figure 5.12 (d).

Theoretical guarantees. Previous models of Architecture Graphs ([28]) allow queries and views to directly refer to the nodes representing the attributes of the involved relations. Due to the framing of modules within input and output schemata



a) Message arrival: the provider of $A_{1.2}$ is to be deleted.

(b) Status determination for input and semantics schema is complete: In_1, $A_{1.2}$ and $Smtx$ have acquired a PROPAGATE status; a message for V_1 consumers is inserted in the local queue describing that V_1 will change its semantics.

(c) Status determination for the output schema is complete: $V_{1.2}$ and Out have acquired a PROPAGATE status; a message for the consumers of $V_{1.2}$ is inserted in the local queue describing that $V_{1.2}$ will be deleted.

(d) Status determination for the entire module is complete (V_1 acquires a PROPAGATE status) and all messages are moved from the local to the global queue

Figure 5.12: Status determination example

and the topological sorting, in Theorem 5.4, and Theorem 5.5 we prove that the process (a) terminates and (b) correctly assigns statuses to modules.

5.3.3 Query and view rewriting to accommodate change

Once the first step of the method, *Status Determination*, has been completed and each module has obtained a status, their rewriting would intuitively seem straightforward: each module gets rewritten if the status is PROPAGATE and remains the same if the status is BLOCK. This would require only the execution of the *Graph Rewrite* step – in fact, one could envision cases where *Status Determination* and *Graph Rewrite* could be combined in a single pass. Unfortunately, although the decision on *Status*

Determination can be made locally in each module, taking into consideration only the events generated by previous modules and the local policies, the decision on rewriting has to take extra information into consideration. This information is not local, and even worse, it pertains to the subsequent, consumer modules of an affected module, making thus impossible to weave this information in the first step of the method, *Status Determination*. The example of Fig. 5.13 is illustrative of this case.

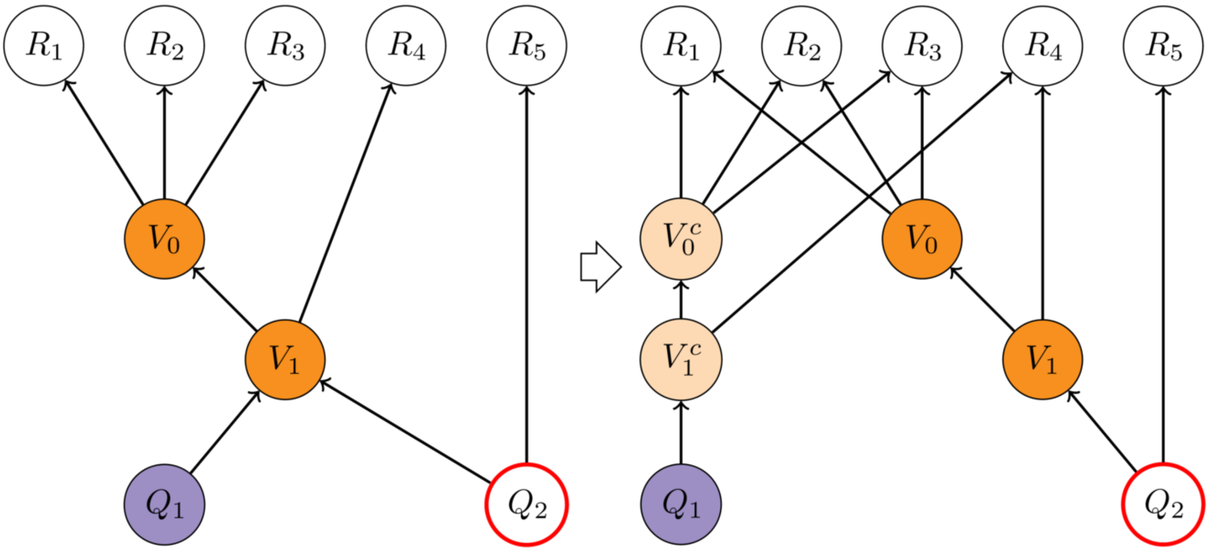


Figure 5.13: Block rewriting example

Figure 5.13 depicts our reference example, which consists of 5 relations, 2 views and 2 queries. We have omitted the full names of the nodes, for illustration purposes. Assume now that the database administrator wants to change V_0, which is the V_Course view of our reference example, in a way that all modules depending on V_0 are going to be affected by that change (e.g., attribute addition, or attribute deletion/rename for an attribute common to all the modules of the example). Assume now that all modules except Q_2 accept to adapt to the change, as they have a PROPAGATE policy annotation. Still, the vetoing Q_2 must be kept immune to the change; to achieve this we must retain the previous version of *all* the nodes in the path from the origin of the evolution (V_0) to the blocking Q_2. As one can see in the figure, we now have two variants of V_0 and V_1: the new ones (named V_0^c and V_1^c) that are adapted to the new structure of V_0 (now named V_0^c), are depicted in the leftmost part of the right figure, with lighter color, and the old ones, that retain their name, are depicted in the rightmost part of the figure. The latter are immune to the change and their existence serves the purpose of correctly defining Q_2.

Algorithm 5.3: Path check

```
Input: An architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a list of modules
       $Affected\ modules$, affected by the event, and the $Initial\ Event$ of the
       user.
Output: Annotation of the modules of $Affected\ modules$ on the action
        needed to take, and specifically whether we have to make a new
        version of it, or, implement the change that the user asked on the
        current version
foreach $Module \in Affected\ modules$ do
    if $Module.status == BLOCK$ then
        CheckModule($Module$, $Affected\ modules$, $Initial\ Event$);
        mark $Module$ not to change;                ▷ Blockers do not change
    end
end
Procedure CheckModule ($Module$, $Affected\ modules$, $Initial\ Event$)
    if $Module$ has been marked then
        return;                                     ▷ Notified by previous block path
    end
    if $Initial\ Event == ADD\_ATTRIBUTE$ then
        mark $Module$ to apply change on current version;   ▷ Blockers ignore
          provider addition
    else
        mark $Module$ to keep current version as is and apply the change on a
          clone;
    end
    foreach $Module\ provider \in Affected\ modules$ feeding $Module$ do
        CheckModule($Module\ provider$, $Affected\ modules$, $Initial\ Event$);
          ▷ Notify path
    end
```

The crux of the problem is as follows: if a module has PROPAGATE status and none of its consumers (including both its immediate and its transitive consumers) raises a BLOCK veto, then both the module and all of these consumers are rewritten to a new version. However, if any of the immediate consumers, or any of the transitive consumers of a view module raises a veto, then *the entire path towards this vetoing node must hold two versions of each module*: (a) the new version, as the module has accepted to adapt to the change by assuming a PROPAGATE status, and, (b) the old version in order to serve the correct definition of the vetoing module. Exceptionally, if the event vetoed involves a relation, the veto freezes any other change and the event is blocked.

To correctly serve the versioning purpose, the adaptation process is split in two steps. The first of them, *Path Check*, works from the consumers towards the providers in order to determine the number of variants (old and new) for each module. Whenever the algorithm visits a module, if its status is BLOCK, it starts a reverse traversal of the nodes, starting from the blocker module towards the module that initialized the flow and marks each module in that path (a) to keep its present form and (b) prepare for a cloned version where the rewriting will take place. A *cloned version* is an identical copy of a module's subgraph, with the same providers but with different name. For example, if we already have a view in SQL as:

```
CREATE VIEW vn AS SELECT c FROM t;
```

then its clone would be

```
CREATE VIEW vn_Clone AS SELECT c FROM t;
```

The only exception to this rewriting is when the module of the initial message is a relation module and the event is an attribute deletion, in which case a BLOCK signifies a veto for the adaptation of the relation.

Finally, all nodes that have to be rewritten are getting their new definition according to their incoming events. Unfortunately, this step cannot be blended with *Path Check* straightforwardly: *Path Check* operates from the end of the graph backwards, to highlight cases of multiple variants; rewriting however, has to work from the beginning towards the end of the graph in order to correctly propagate information concerning the rewrite (e.g., the names of affected attributes, new semantics, etc.). So, the final part of the method, *Graph Rewrite*, visits each module and rewrites the module as follows:

Algorithm 5.4: Graph Rewrite

```
Input: A list of modules $Affected\ modules$, knowing the number of versions
       they have to retain, initial messages of $Affected\ modules$
Output: Architecture graph after the implementation of the change the user
        asked
if any of $Affected\ modules$ has status BLOCK then
    if initial message started from Relation module type AND event ==
     DELETE_ATTRIBUTE then
        return;
    else
        foreach $Module \in\ Affected\ modules$ do
            if $Module$ needs only new version then
                proceed with rewriting of $Module$;
                connect $Module$ to new providers ;  ▷ new version goes to new
                 path
            else
                clone $Module$;              ▷ clone module, to keep both versions
                connect cloned $Module$ to new providers;    ▷ clone is the new
                 version
                proceed with rewriting of cloned $Module$;
            end
        end
    end
    foreach $Module \in\ Affected\ modules$ ;                    ▷ no blocker node
     do
        proceed with rewriting of $Module$ ;      ▷ all modules fix their edges
         internally
    end
end
```

- If the module must retain only the new version, once we have performed the needed change, we connect it correctly to the providers it should have.

- If the module needs both the old and the new versions, we make a clone of the module to our graph, perform the needed change over the cloned module and connect it correctly to the providers it should have.

- If the module retains only the old version, we do not perform any change.

One could possibly argue that we could have used a principled way to mark the paths of the blocker modules, starting from the blocker module and visiting all the affected modules with ID smaller than the blocker's ID, marking them to have two versions in the new schema. Unfortunately, this method would have been insufficient as it would not be able to guarantee that the affected modules that are not in the path of a blocker module will not be marked to obtain two versions too. For example, in Figure 5.13, the $Q_1.ID$ could be either 8 or 9 after the topological sorting. If $Q_1.ID$ is 9, then the aforementioned ID based traversal could be used. If $Q_1.ID$ is 8, then $Q_2.ID$ is 9 and the aforementioned ID based traversal would mark the Q_1 module to obtain two versions, which is wrong.

How much cloning is required? Each execution of the *Path Check* and the *Graph Rewrite* algorithms involves one event only. For each such event, a cloning is required whenever (a) the event involves deletion or semantics update, (b) a view module initiates the propagation of an event due to its PROPAGATE policy, and, (c) some of its (possibly transitive) data consumers raises a veto. In this case, the entire path till the blocker (blocker excluded) must be cloned. If there are two blockers that have the same provider, then there is no extra duplication. For a given event that fulfils the aforementioned conditions, assuming n blockers in an event, and m paths, $m \leq n$, involving M nodes (excluding the blockers), we need M extra cloned modules. If the graph contains V views and Q queries, the maximum impact is when all of them are affected by an event. A worst case scenario can be conceived when there is a root view and all other views and queries defined over this view either directly or transitively. Assume now that the root view is affected in a way that all views and queries are affected (e.g., change of semantics) and all queries are blockers, although all views are propagators (because if another view is a blocker, its queries are protected). Then, we need to clone V views, which is the maximum amount of cloning that can happen in an event. Our reference example is in fact such a worst case (see Fig. 5.13). Practically speaking, this possibility is rare (observe for example Fig. 5.15 on how a large subset of the Drupal ecosystem is constructed).

Returning to the rewriting process of modules with a PROPAGATE status, we can summarize this process as follows:

- Whenever the attributes of a modules output schema are deleted, renamed or inserted, the subsequent consumer schemata are adopted accordingly;
- Whenever entire modules are deleted or renamed, the respective schemata are deleted or renamed accordingly.

In the following paragraphs, we are going to discuss the way the rewriting process is performed within each module. Initially, we need to distinguish two categories, depending on the type of the module that is rewritten: (a) the rewriting processes that apply to Relation modules, and, (b) the processes that apply to Query/View modules. This differentiation is mainly due to the fact that, in contrast to the Relation modules that contain only an output schema, the Query/View modules additionally contain a semantics schema and a set of input schemata (one per provider). Therefore, queries and views require a different treatment.

In the following two subsections, we are going to briefly describe for each event, the steps that are followed in the module rewriting mechanism, when the module is accepting the change (its status is PROPAGATE). Naturally, if the status is BLOCK, no rewriting is required at the internals of the module.

Relation module rewriting

For each of the events applied to a relation (as presented in Table 5.1), we perform the following steps for rewriting the affected relation module and propagating the event towards the rest of the graph:

Attribute addition When a new attribute is added to a relation module, the user is prompted for the name of the new attribute and the module checks if it is available or already in use by another attribute. If all conditions are met then the new attribute is added to the output schema of the module and a message with the addition along with the new attribute name is propagated to all dependent modules.

Attribute deletion When an attribute is deleted, the output schema searches for the specific attribute and deletes it. Similarly, a message for the deletion is propagated to all dependent modules.

Attribute rename When an attribute is renamed, the output schema searches for the specific attribute and renames it with the name provided by the user, unless there is conflict with any attribute having the same name in the output schema of the module; in this case the user is prompted to change it. Again, a message for the renamed attribute is generated.

Self (module) deletion When a relation module is deleted, its output schema with all its attributes are deleted and the module node itself is also deleted. A message for the deletion is propagated to all dependent modules.

Self (module) rename When a relation module is renamed, the user is prompted for the new name of the module. If it is unique, the module and its output schema are renamed accordingly. Moreover, a message with the renamed relation is propagated to all connected query/view modules, in order to update their input schemata with the new name.

Query/View module rewriting

Query and View modules have the same events, thus we do not separate their rewriting methods. The steps for rewriting (a PROPAGATE status is assumed) are as follows:

Attribute addition The user adds a new attribute by selecting it out of the list of attributes belonging in the output schemata of the query/view providers and sets a unique alias name for this attribute. In case there is a GROUP BY clause in the semantics schema, the user is prompted for adding the new attribute to the groupers or using any aggregate function. In any case, the new attribute is directly added to the output schema of the module. If the attribute was not used before in the query/view, it is added in the respective input schema and finally all the needed connections between the output node and the semantics (if applicable) and input node are set. Moreover, its name is propagated to the modules that are connected, in order to let them know the name of the new attribute.

Attribute provider addition When an attribute is added in a provider of the module, the input schema of the module adds a new attribute node with the specified name. If there is any connection to the semantics schema due to a GROUP BY clause, the user is again prompted for adding the new attribute to the groupers or using any aggregate function. Finally, when all conditions are met, the new attribute is added to the output schema of the module (checking to see if there are any conflicts with the name of the new attribute and if so, the user is prompted accordingly). Moreover, the name of the new attribute is propagated to the modules that are connected, in order to update their input schemata accordingly.

Attribute deletion For the case that an output attribute is deleted, it is first removed from the output schema. All connections of the output attribute with the semantics schema are removed and finally, if the attribute is not used in the semantics tree, it is removed from the respective input schema.

Attribute provider deletion When an attribute of a provider module is deleted, it is initially removed from the corresponding input schema of the module. If it is used in a condition in the semantics tree, then this condition is set to true or false, depending on the operator which connects this condition with the semantics tree (true if the condition was connected to an AND operator and false if it was connected to an OR operator). Finally, if this attribute is part of the SELECT clause of the query, it is removed from the output schema. See Fig. 5.14 for how the aforementioned example of Fig. 5.12 is rewritten.

Attribute rename When an attribute is renamed, the output schema searches for the specific attribute and renames it with then name provided by the user, unless there is conflict with any attribute already present in the output schema of the module, having the same name with the one chosen for renaming (in this case the user is prompted again). Moreover, its name is propagated to the modules that are connected, in order to let them know the new name of the attribute in order to update their schemata.

Attribute provider rename When an attribute is renamed in one of the providers of the module, the attribute is initially renamed in the corresponding input schema of the module. If there is a connection between any attribute of the output schema with the same name, this attribute is also renamed, unless the name is already used by any attribute of the output schema, in which case, the user is prompted for a new name. Finally, this new name is further propagated to the modules that are connected to this current one.

Module deletion When a query/view module is deleted, the schemata nodes of the module with all their attributes are deleted and the module node itself is also deleted.

Module rename When a query/view module is renamed, the user is prompted for the new name of the module and if it is unique in the graph, then the module and its output, semantics and input schema nodes are renamed accordingly. Moreover, the new name is propagated to the modules that are connected to this query/view, in order to know the new name of the module and update their input schemata.

Provider module deletion When a provider module is deleted, the respective input schema of the module that receives this notification is deleted. The steps applied to the deletion of a provider attribute are performed for all attributes of the the deleted provider module.

Provider module rename When a provider module is renamed, the module that receives this notification initially checks if its input schema that corresponds to the renamed provider had the same name with the old provider name (not always the case, since there could be an AS rename in the query/view definition). If this is the case, the input schema of the module is renamed accordingly (unless the new name conflicts with an AS rename of any other input schema of the module). If the new name due to conflicts cannot be set to the input schema, the user is prompted for a new one.

Alter of semantics When a query/view module changes it semantics, the user is prompted to alter the where and the group by clause of the module and the semantics tree is rewritten from this input. When an alter of semantics message arrives from any of the module's providers, and the module has PROPAGATE semantics, then, as we have already discussed in the previous subsection, there is no rewriting at all at the module.

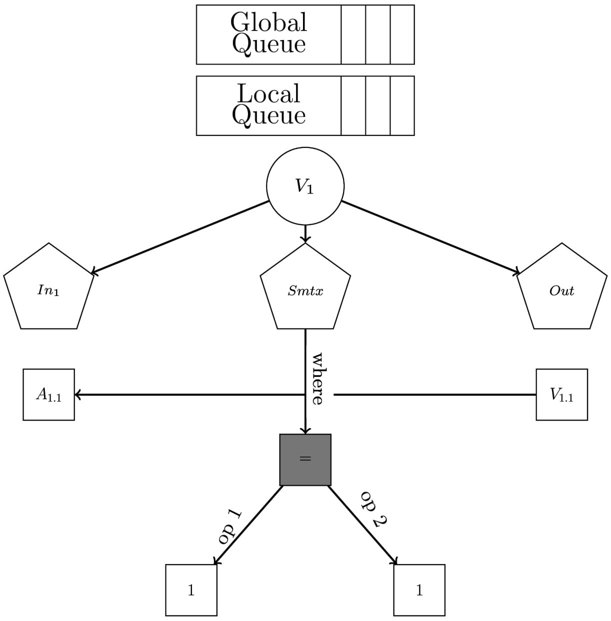


Figure 5.14: Rewriting for the example of Fig. 5.12

5.4 Theoretical Guarantees

In this section, we provide a set of theoretical guarantees regarding the correct annotation of the graph with events and policies and the termination and confluence properties of our proposed algorithms.

5.4.1 Language Properties

Theorem 5.1. *The entries of Table 5.1 cover* completely *the space of node types with the events they can sustain.*

Proof. The table that contains the events that each node type can receive was described earlier in Section 5.2 (Table 5.1). In Table 5.2 we have replaced the ✓ symbols of the table's cells with the numbers of the default policy rules, according to the numbering scheme of Figure 5.7. Two cells in the ALTER_SEMANTICS column are annotated with a ✓ and without a reference to a rule; we explain why in the following text.

The events that are user generated and pertain to views and queries are:

UQV.1 ADD_ATTRIBUTE

UQV.2 DELETE_ATTRIBUTE

UQV.3 RENAME_ATTRIBUTE

			ADD		DELETE		RENAME		ALTER
			ATTR	ATTR PROV	SELF	PROV	SELF	PROV	SMTX
QUERY	OUT	SELF	1	2	3		4		
		ATTRS			5	7	6	8	
	IN	SELF		11		9		10	✓
		ATTRS				12		13	
	SMTX	SELF							14
VIEW	OUT	SELF	15	16	17		18		
		ATTRS			19	21	20	22	
	IN	SELF		25		23		24	✓
		ATTRS				26		27	
	SMTX	SELF							28
RELATION	OUT	SELF	29		30		31		
		ATTRS			32		33		

Table 5.2: The space of events that can be received by each node type according to the line number in the rules of the policy file

UQV.4 DELETE_SELF

UQV.5 RENAME_SELF

UQV.6 ALTER_SEMANTICS

As mentioned previously in Section 5.2.2, the events that are user generated and pertain to relations are:

UR.1 ADD_ATTRIBUTE

UR.2 DELETE_SELF

UR.3 RENAME_SELF

UR.4 DELETE_ATTRIBUTE

UR.5 RENAME_ATTRIBUTE

Our policy language covers all these events that are related with the user interaction and are the marks of Table 5.1 that are in the lines that contain the OUT and SMTX keywords on queries, views and relations.

Due to the message propagation mechanism, additional events occur. These events (also described in Section 5.2.2) are received by the IN nodes of the query and view modules. These events are:

MP.1 ADD_ATTRIBUTE_PROVIDER

MP.2 DELETE_PROVIDER

MP.3 RENAME_PROVIDER

MP.4 ALTER_SEMANTICS

For our policy language to cover the three of the four previous events (MP.1, MP.2, and MP.3) that are related to the message propagation mechanism, additional policies are needed. The exception of MP.4 is due to the fact that the IN schema node who receives such an event forwards it to the respective SMTX node who is actually the one responsible for the handling of this event. Therefore, there is no need to define a policy at the IN schema node, as the event will be appropriately handled at the SMTX node.

In Table 5.2 we have all the above combinations of events and node types, thus, our default 33 rule have completely covered the space of node types with their incoming events.

Precisely, lines 1 to 14 concern queries.

As previously stated at the exception of MP.4, the 14 rule (QUERY.SMTX.SELF: on ALTER_SEMANTICS then <policy>;) covers two events, since the IN node forwards this message to the SMTX node which is the only responsible for the policy over the ALTER_SEMANTICS event.

Likewise, lines 15 to 28 concern views.

The 28 rule (VIEW.SMTX.SELF: on ALTER_SEMANTICS then <policy>;) covers two events just like 14 rule does, for the exact same reasons.

```
1 QUERY.OUT.SELF: on ADD_ATTRIBUTE then <policy>; which is for UQV.1
2 QUERY.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>; // which is for MP.1 in output schema node
3 QUERY.OUT.SELF: on DELETE_SELF then <policy>; which is for UQV.4
4 QUERY.OUT.SELF: on RENAME_SELF then <policy>; which is for UQV.5
5 QUERY.OUT.ATTRIBUTES: on DELETE_SELF then <policy>; which is for UQV.2
6 QUERY.OUT.ATTRIBUTES: on RENAME_SELF then <policy>; which is for UQV.3
7 QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>; which is for MP.2
8 QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>; which is for MP.3
9 QUERY.IN.SELF: on DELETE_PROVIDER then <policy>; which is for MP.2
10 QUERY.IN.SELF: on RENAME_PROVIDER then <policy>; which is for MP.3
11 QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>; which is for MP.1 in input schema node
12 QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>; which is for MP.2
13 QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>; which is for MP.3
14 QUERY.SMTX.SELF: on ALTER_SEMANTICS then <policy>; which is for both UQV.6 and MP.4
```

Table 5.3: Query policies with the addressed events

```
15 VIEW.OUT.SELF: on ADD_ATTRIBUTE then <policy>; which is for UQV.1
16 VIEW.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>; which is for MP.1 in output schema node
17 VIEW.OUT.SELF: on DELETE_SELF then <policy>; which is for UQV.4
18 VIEW.OUT.SELF: on RENAME_SELF then <policy>; which is for UQV.5
19 VIEW.OUT.ATTRIBUTES: on DELETE_SELF then <policy>; which is for UQV.2
20 VIEW.OUT.ATTRIBUTES: on RENAME_SELF then <policy>; which is for UQV.3
21 VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>; which is for MP.2
22 VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>; which is for MP.3
23 VIEW.IN.SELF: on DELETE_PROVIDER then <policy>; which is for MP.2
24 VIEW.IN.SELF: on RENAME_PROVIDER then <policy>; which is for MP.3
25 VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>; which is for MP.1 in input schema node
26 VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>; which is for MP.2
27 VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>; which is for MP.3
28 VIEW.SMTX.SELF: on ALTER_SEMANTICS then <policy>; which is for both UQV.6 and MP.4
```

Table 5.4: View policies with the addressed events

Finaly, lines 29 to 33 concern relations.

```
29 RELATION.OUT.SELF: on ADD_ATTRIBUTE then <policy>; which is for UR.1
30 RELATION.OUT.SELF: on DELETE_SELF then <policy>; which is for UR.2
31 RELATION.OUT.SELF: on RENAME_SELF then <policy>; which is for UR.3
32 RELATION.OUT.ATTRIBUTES: on DELETE_SELF then <policy>; which is for UR.4
33 RELATION.OUT.ATTRIBUTES: on RENAME_SELF then <policy>; which is for UR.5
```

Table 5.5: Relation policies with the addressed events

As one may notice the 33 rules cover all the 35 events that may appear in each one of the nodes. The inequality of the numbers is because of the exception of MP.4. Therefore, the fact that the 33 rules cover 33 events plus the two events that do not need any rule proves that all the events (UR.*, UQV.*, and MP.*) are covered by our policy rules.

Moreover, if we override the 33 default rules, then, the most refined policy will be enforced for each node. □

Theorem 5.2. *The policy overriding mechanism is correct (assigns the correct policy to each node).*

Proof. One node may have more than one policies for a specific event. This occurs because a policy over an event may be set in any of the following three rules:

1. Rules about all the nodes of the *Architecture Graph*.

2. Rules about all modules and their attributes.

3. Rules that apply to all the attributes of a *specific schema*.

4. Rules that apply to *specific attribute* nodes.

The golden standard of correctness requires that whenever a node has more than one policies for the same event, the one that perseveres is the policy defined at the finest level of detail.

The overriding mechanism is correct because the following sequence of events is guaranteed: initially, it we apply the most general rules for all the nodes of the graph, then the rules per module type, then the rules referring to the attributes of specific schemata, and finally, the rules that apply to specific attributes.

Observe that this is independent on whether the policies are assigned a priori, during the construction of the graph, or, on demand, whenever a specific node needs to determine its policy. □

Theorem 5.3. *The extra rules*

- *<moduleType>: ON * THEN <policy>;*

- *<namedNode>: ON * THEN <policy>;*

- *NODE: ON <event> THEN <policy>;*

- *NODE: ON * THEN <policy>;*

can correctly cover up the events of the Table 5.2 and correctly override each other mechanism (assign the correct policy to each node).

Proof. We need to prove that these rules will cover up all the events that a node might receive, as well as that these rules correctly override each other. The more general rules are the ones that contain the keyword NODE. These rules are applied first. Then the rules that apply to modules and finally the rules that are applied to specific nodes of the graph. Within each rule, the rules that contain the keyword * are preceding over the others rules.

The rule: *NODE: ON * THEN <policy>;* is translated to all the 33 rules described in Figure 5.7 and prove their correctness in Theorem 5.1. So all the events are covered. This rule is also the first one to be applied in our graph, regardless of its position.

The rule *NODE: ON <event> THEN <policy>;* is translated to the rules that apply for the specified event type. We can follow the columns of the table 5.2 in order to see that for:

- ATTRIBUTE_ADDITION, the rules of Figure 5.7 that apply are: rule 1 for the queries, rule 15 for the views and rule 29 for the relations.

- ATTRIBUTE_PROVIDER_ADDITION, the rules of rules of Figure 5.7 that apply are: rule 2 and 11 for the queries and rule 16 and 25 for the views.

- DELETE_SELF, the rules of Figure 5.7 that apply are: rule 3 and rule 5 for queries, rule 17 and rule 19 for views, rule 30 and rule 32 for relations.

- DELETE_PROVIDER, the rules of Figure 5.7 that apply are: rule 7, rule 9 and rule 12 for the queries, rule 21, rule 23 and rule 26 for the views.

- RENAME_SELF, the rules of Figure 5.7 that apply are: rule 4 and rule 6 for the queries, rule 18 and rule 20 for the views, rule 31 and rule 33 for the relations.

- RENAME_PROVIDER, the rules of Figure 5.7 that apply are: rule 8, rule 10 and rule 13 for the queries, rule 22, rule 24 and rule 27 for the views.

- ALTER_SEMANTICS, the rules of Figure 5.7 that apply are: rule 14 for the queries and rule 28 for the views.

This rule is the second one that is applied in our graph, in order to correctly override the more general first rule (*NODE: ON * THEN <policy>;*).

The rule *<moduleType>: ON * THEN <policy>;* is translated to the set of rules that apply to the specific module type. For example,

- for the query module type, these rules are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, and, 14,
- for the view module type, these rules are: 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, and, 28,
- for the relation module type, these rules are: 29, 30, 31, 32, and, 33.

This rule is the third one that is applied in our graph, in order to correctly override the two more general rules.

Finally, the rule *<namedNode>: ON * THEN <policy>;* is translated to the rules that apply to the module type of the specific *<namedNode>*. This means that the rules that are generated have the *<namedNode>*'s name. For example, for a relation named TRANSCRIPT,

1. the rules will start with TRANSCRIPT_SCHEMA
    - for the ATTRIBUTE_ADDITION event, which is for the rule 29,
    - for the DELETE_SELF event, which is for the rule 30,
    - for the RENAME_SELF event, which is for the rule 31,
2. and two more rules that are for the attributes of the TRANSCRIPT relation, starting with TRANSCRIPT_SCHEMA.ATTRIBUTES
    - for the DELETE_SELF event of the attributes, which is for the rule 32,
    - for the RENAME_SELF event of the attributes, which is for the rule 33.

This rule is applied after the rules per module type have been applied and before rules with specific events for specific nodes are applied. □

5.4.2 Theoretical Guarantees for the Status Determination Algorithm

First, we prove that the mechanism for message propagation works correctly at the inter-module level.

Theorem 5.4. *The message propagation at the inter-module level terminates.*

Proof. The summary of the architecture graph is a directed acyclic cycle. This is due to the fact that (i) a query depends only on views and relations, and (ii) relations do not depend on anything (in the context of this chapter, we do not consider cyclic foreign key dependencies).

Since the summary graph is a DAG, we can topologically sort it and propagate the messages according to this topological order. Thus, all that it takes for the message propagation mechanism to terminate is: (a) each module emits a message only once for each session to its consumers that are related with the event/parameter; (b) the graph is finite. Since both assumptions hold, the algorithm terminates. □

Theorem 5.5. *Each module in the graph will assume a status once the message propagation terminates; this status is the same, independently from the order of processing the incoming messages.*

Proof. Each module gathers from the common message queue *all* the messages that concern it. For each message, the module and its schemata, assume a status. A module's status can change only in the following order: NO_STATUS < PROPAGATE < BLOCK, meaning that if a module has assumed a PROPAGATE status, it can not change it to NO_STATUS but it may change it to BLOCK. Therefore, if a message that will ignite a BLOCK policy is found anywhere in the list of incoming messages, this BLOCK status will eventually be assumed and not overridden later. Otherwise, a PROPAGATE status will be assumed. At the end of the message processing, the module retains the final status it assumed. □

Theorem 5.6. *Messages are correctly propagated to the modules of the graph.*

Proof. For the node that receives the initial event we need to prove that:

1. the node either acquires BLOCK status, therefore the message propagation mechanism stops, or,
2. the node acquires PROPAGATE status and notifies its consumers about the change.

For all the other nodes we need to prove that:

3. that a module will not be affected if none of its providers was affected by the imminent change,

4. there is no module that receives a message while its provider has a BLOCK status,

5. there is no module that should have received a message when it was its turn to acquire a status but the message was not in its input message list,

The first two propositions stand because of the rewrite maestros mechanism. The modules communicate using a global list of messages. The rewrite maestro keeps a local list of all the outgoing messages of the module to its consumers. When the module finishes processing all its incoming messages, the maestro checks the module's status and if it is BLOCK, then returns, *without* adding the outgoing messages to the global list, which proves the first proposition. If the status of the module is PROPAGATE then the output messages *are added* in the global list, so the consumers of the module are notified, which proves the second proposition.

For the third proposition: One (or more) input schema node(s) of a consumer module is connected via directed edges to the output schema node(s) of its providers. Due to its inherent construction, the modules which are eventually visited by the message propagation mechanism, have at least one of their providers affected. For the same reason, the modules that are not visited by the mechanism (a) either do not have any provider affected, or, (b) a block policy terminated the message propagation in provider modules, earlier.

For the fourth proposition: The messaging mechanism dictates that each message is propagated from the output node of the *provider* module towards the input schemata of all *consumer* modules, unless a BLOCK policy explicitly halts the propagation. Since a BLOCK policy terminates the message propagation from this *provider* module, we guarantee that there is no *consumer* module to receive any message from *provider* module.

For the fifth proposition: The messaging mechanism uses a *list* to transfer the messages between the modules. This *list* is sorted by the ID numbers that the modules have acquired by the topological sort algorithm (described in Figure 5.1). Since the *list* is topologically sorted too, we guarantee that there there is no module that should have received a message when it was its turn to acquire a status but the message was not in its input message list. □

Theorem 5.7. *The message propagation at the intra-module level: (i) terminates, with (ii) each node assuming a unique status according to its policy and the status precedence* *constraints.*

Proof. We visit the schemata of a module in a fixed order: input schemata, semantics schema, output schema. For each of these schemata, we may visit its attributes too. All these constructs are finite and visited only once. This is a task that the maestros perform and the very reason for their existence, otherwise we could have allowed message propagation via the graph's edges within the modules too. Therefore, the algorithm terminates and (i) is proved.

For requirement (ii) we need to prove the following:

1. *for all messages*, vetoes override adaptation,

2. *per message*, for all the appropriate nodes (and only them) the status of the most detailed nodes overrides the decision of the status of the schema,

3. if any of the schemata of a module has status BLOCK, the module assumes status BLOCK.

Regarding the first proposition: as already stated at the proof of Theorem 5.5, a node's status can change only in the following order: NO_STATUS < PROPAGATE < BLOCK, meaning that if a node has assumed a PROPAGATE status, it can not change it to NO_STATUS but it may change it to BLOCK.

Regarding the second proposition: every time a schema is probed on an event (a) the appropriate nodes within a schema are asked about their policy, or/and, (b) the schema itself is asked about its policy. Table 5.2 describes the relationship between events and nodes prompted, in the lines that say ATTRS the (a), (b) take place, while in the lines that say SELF only the (b) takes place. This is the correct and desired behavior. When an attribute acquires a status, the schema node is prompted to acquire the same status. The completeness of the language guarantees that *all* nodes have a policy for any incoming event that can arrive to them. Therefore, in all occasions (i) the correct nodes are prompted for a response, (ii) the policy of the appropriate nodes prevails, (iii) it is impossible that such a policy does not exist. Therefore, for each message all nodes acquire the correct status.

Regarding the third proposition: the proposition is inherently supported. □

5.4.3 Theoretical Guarantees for the Path Check Algorithm

We are going to prove that Path Check Algorithm terminates and all modules at the end have the correct number of versions they need to keep.

Theorem 5.8. *The module traversal terminates and the visited modules have the correct notification of how many versions they need.*

Proof. Algorithm *Path Check* sequentially passes from each of the affected modules with BLOCK status and for each of them executes method *CheckModule*. If we can prove that *CheckModule* terminates, then the algorithm terminates too.

The algorithm has as input: (i) a *finite* set of modules (each module with BLOCK status starts the *CheckModule* method once), and (ii) the initial event placed by the user.

In every step, the *CheckModule* method marks the module to keep two versions, and finds the providers of this module through which the module was marked about the change. These provider modules are listed in the set of the affected modules. If there are no more modules this means that the method reached the module from which the change started.

Since this is a recursive procedure, the providers of the providers of those modules are also marked and so on. The condition that inspects whether the visited module was previously marked, is done by the following line:

```
If(Module has been marked) Then return;
```

of the *CheckModule* method. This condition makes sure that the recursive traversals of the method terminate as soon as possible –since those modules have already been marked by a previous traversal– and every module that is part of the path that goes from a blocker module to the source of the changes has been marked to keep two versions. □

5.4.4 Theoretical Guarantees for the Graph Rewrite Algorithm

We are going to prove that (a) *Graph Rewrite* Algorithm terminates, (b) when *Graph Rewrite* terminates, all modules have the correct connections at the inter-module level, and (c) all modules are correctly rewritten at the intra-module level.

Termination and confluence at inter-module level

First, we prove that the mechanism for graph rewriting works correctly at the inter-module level.

Theorem 5.9. *The graph rewriting at the inter-module level terminates.*

Proof. The Graph Rewriting Algorithm terminates when all the notified modules that accepted the change (meaning that those modules acquired a PROPAGATE status) are rewritten. The algorithm has as input the list of the affected modules, each one having its status and the number of versions it needs to keep, and the initial messages that each affected module received. In the special case of the DELETE_ATTRIBUTE event coming from a Relation module, the algorithm terminates right away. Otherwise, each one of the affected modules (which is a finite list of modules) is rewritten *once*, so the algorithm terminates.

We need to prove:

1. that each module is rewritten only once for each one of the messages it received,
2. that there is a finite list of messages, and
3. that there is a finite number of modules that are going to be rewritten.

For 1 and 2 since the incoming messages of a module are finite (as proved earlier in theorem 5.4), and maestros are only once executed per message we are sure that each module is rewritten once per message received. For 3 the number of modules that acquired PROPAGATE status is finite, since the graph is finite. Therefore, since all assumptions hold, the algorithm terminates. □

Theorem 5.10. *The graph will be in the correct form after the rewrite.*

Proof. We need to prove that:

1. All the modules that have no status will not be rewritten.
2. All the modules with BLOCK status will not be rewritten.
3. If there is no vetoer in the graph, then all the modules with PROPAGATE status will be rewritten.

4. If there is any vetoer, then the modules with PROPAGATE status will be rewritten (i) themselves –since there is only one version needed– if they are not part of a blocker path, or, (ii) as clones –since there are two versions needed– as parts of a blocker path. In both cases the modules will be connected to the appropriate path.

A module is part of a blocker path when the module has PROPAGATE status but at least one of its descendants acquired status BLOCK.

We need two paths, the “new providers” in which all the nodes accepted the change, and the “old providers” in which we keep the old definitions of all the affected modules because a module declined the change. If a module needs to keep only one version then the path with the “new providers” is the one that this module belongs to. If a module needs to keep two versions then the path with the “old providers” that did not want to accept the change is the one that this module belongs to, while its clone belongs to the path with the “new providers” that accepted the change, thus providing right essence to the modules that need to keep only one version. The number of versions a module has to keep is given by the algorithm depicted in Algorithm 5.3.

If none of the modules vetoed, then the *Graph Rewrite* Algorithm does a traversal visiting the affected nodes, in order to apply the change the user asked. The algorithm works only with the modules that have PROPAGATE status, thus 1, 2 and 3 are proved. For 4:

1. If the module needs to keep two versions we clone the module, we connect the cloned module to its new providers (if it is the module that started the event then we connect it to the providers that the original has), and we proceed with the rewrite of the cloned module. This way, the cloned modules are all in one path, and the modules that vetoed are all in the other path.

2. If the module belongs to a path without blocker modules, then it needs to keep only the new version we connect it to the path of the new providers and we proceed with the internal rewriting of the module.

□

Termination and confluence at intra-module level

Theorem 5.11. *The rewriting of modules at the intra-module level terminates and each module is rewritten correctly.*

Proof. Sections 6 and 6 describe the intra module rewriting process, where we begin our module rewriting from the input schema, continue to the semantics schema and terminate to the output schema. There is only one exception at the aforementioned rule and that is on the attribute addition of query/view modules, where we start from the output schema of the module and move towards the semantics and input schemata.

In both cases, we start rewriting from the one border of the module (input or output) and terminate to the other border of the module (output or input, respectively). Because of the previous statement, we guarantee that our method terminates because: (a) we perform a single visit per affected node, and (b) we work with a finite number of nodes. As for the validity of the intra-module rewriting, each event is rewritten as described in sections 6 and 6 and whenever information is needed, either the modules passes that information from the provider module to the consumer module, or prompt the user to provide the needed information. □

5.5 Experiments

We assessed our method for its usefulness and scalability with varying graph configurations and policies; in this section, we report our findings. As already mentioned, all the material for this work, including input ecosystems, links to the source code (publicly available at git) and results can be found in the following web page: `http://www.cs.uoi.gr/~pmanousi/publications/2013_ER/index.html`. We have employed a real-world case, based on 7 major revisions of Drupal in the period 2003 - 2007 as the testbed for our experimentation. To further stress-test our method with more complicated scenarios, we have also performed a controlled experiment, based on a widely used benchmark, TPC-DS, to allow the evaluation of the effect of different problem parameters (like policy annotation and graph size) to the effectiveness, efficiency and required user effort of our method. Before proceeding, we describe the fundamental metrics that we employ for the assessment of our experiments.

5.5.1 Effectiveness and Effort Metrics

In this subsection, we discuss the metrics used to assess the efficiency, the effort spent for the annotation of the graph and the effectiveness metrics that we employ in our experiments. Evaluating efficiency is straightforward, as we assess the breakdown of the time spent for each of the 3 steps of our method. For the rest of the metrics, we provide a more detailed discussion, right away. We conclude this subsection with a note on the experimental configuration of each experiment.

Effectiveness: do we gain from annotating the graph with policies and testing what-if scenarios this way?

How can we assess the effort gain of a developer using the highlighting of affected modules of Hecataeus? This gain should be contrasted to the effort spent in the case where he would have to perform all checks by hand. We employ the $\%AM$ metric, measuring the *fraction of Affected Modules of the ecosystem* as the gain that amounts to the percentage of useless checks the user would have made. We exclude the object that initiates the sequence of events from the computation, as it would be counted in both occasions. Formally, $\%AM$ is given by the Equation 5.1.

$$\%AM = 1 - \frac{\#Affected\ Modules}{\#(Queries \cup Views)} \tag{5.1}$$

Moreover, to assess the extent of rewritings that are automated by our method, for each event we measure the *number of Rewritten Modules* as the sum of the number of modules that were cloned (new versions of affected modules) with the number of existing adapted modules. We denote this measurement with the RM metric, given by the Equation 5.2.

$$RM = \#Adapted\ Modules\ +\ \#Cloned\ Modules \tag{5.2}$$

Policy annotation effort: how much time does it take to setup the policy rules in order to work with our what-if analysis tool?

How hard is it to annotate the graph with policies? How much time does the user have to spend for authoring the rules?

Our method comes with the possibility of using syntactic sugar rules that make life easy and fast. For the rare occasion when the user does not want to use these syntactic shortcuts, for every specific module that gets into the user's focus, the user has to provide as many rules as necessary to override the default policies. In the worst case, this requires $9 + 5 \times |input\ schemata|$ rules for a full re-specification of a query/view module. When the syntactic sugar is used, *one rule is sufficient to fully invert the policy of a module*. Of course, rules for more detailed subsets of the module can also override this default. In any case, in order to write these rules, the user has to locate the module in the graph and invoke the graphical policy editor; however, locating the module is actually the difficult part of the annotation. To address this task, Hecataeus comes with a layout containing the filesystem of the project that the user investigates. Initially, the user has to find the file that contains the query he wants to change its policy. When the user selects a file, the queries that are in this file, are highlighted in the visual representation of the *Architecture Graph* in our tool, providing a smaller set of modules that need to be searched. Finally, when the user finds the module he wants to differentiate from the global policy, he adds only *one* line of text to the policy file that says that this query has a specified policy. *We repeatedly monitored the annotation time, using a wristwatch, and this task takes at most 1 minute for each query that the user wants to set a specific policy.*

In each experiment, we also discuss the number of rules required for the execution of the experiment. We believe the annotation effort is practically negligible.

Experimental configuration

In all our experiments, we need to fix the following parameters for our experimental setup: (a) an ecosystem comprising a database schema surrounded by a set of queries and possibly a set of views, (b) a workload of events that are sequentially applied to the above configuration, and, (c) a palette of "profiles" that determine the way the ecosystem's architecture graph is annotated with policies towards the management of hypothetical events; hence, these profiles simulate the intention of the administrating team for the management of the ecosystem.

5.5.2 Replaying the Evolution of Drupal

Ecosystem. In this experiment, we have employed Drupal, versions 4.1.0 to 4.7.11 [3] as our experimental testbed. Drupal is a Content Management System (CMS) that is

[3] http://ftp.drupal.org/files/projects/

written in PHP language, which contains SQL queries in its php script files. We used some of the major versions of this project that took place between 2003 and 2007. As one may observe in Table 5.6, there are no views in this project; that is why we decided to split the experiment in two setups, described in Sections 5.5.2 and 5.5.2 respectively.

Original evolution scenario

In this setup, we replay the original evolution of the Drupal project, raising each one of the events that really occurred, having no blocker modules.

Events. We have used the actual events that evolved the database schema of Drupal between the major versions we describe in Table 5.6. For example, in order to get to version 4.2.0 we had to perform 6 attribute additions and 2 attribute deletions.

Policies. The default reaction for the original scenario was to accept all changes between two subsequent versions. Thus, the policy for all modules was to propagate all events that occur on the system; this is expressed by having only one rule in the policy file (`NODE: on * THEN PROPAGATE;`).

Modified scenario with view cloning

In the second setup, we replay an alternative evolution case of the Drupal project, in order to examine the effect of cloning of views on the overall system. Specifically, we added a view named "*UNView*", that is used to join the `USERS` and `NODE` relations. Then, we rewrote all queries joining the two tables to use the view instead. Moreover, we added one extra query that asks for all the attributes of *UNview* which would act as a blocker to all events that ultimately reach it. This setting allows us to see how view cloning operates ”in the microscope”.

Events. We have also used the actual events as in the previous setup. The only difference to the previous approach is that, when there was an attribute deletion in `USERS` or `NODE` relations, we performed the deletion to the *UNview* module, instead of the `USERS` or `NODE` relation modules.

Policies. The policy again was to propagate all the changes in all modules except for the additional query; this is expressed by the following two rules:

- `NODE: on * then PROPAGATE;` and
- `Qadditional: on * then BLOCK;`

Drupal Version	Published at	Relations	Queries	Attr. Add.	Attr. Del.	Table Add.	Table Del.
4.1.0	2003-02-01	38	240	6	2	0	0
4.2.0	2003-08-01	38	247	1	5	3	1
4.3.1	2003-12-01	40	260	8	4	1	1
4.4.3	2005-06-01	40	263	16	5	16	4
4.5.8	2006-03-14	52	284	12	11	4	1
4.6.11	2007-01-05	55	332	14	11	7	5

Table 5.6: Drupal dataset from ver. 4.1.0 to ver. 4.7.11

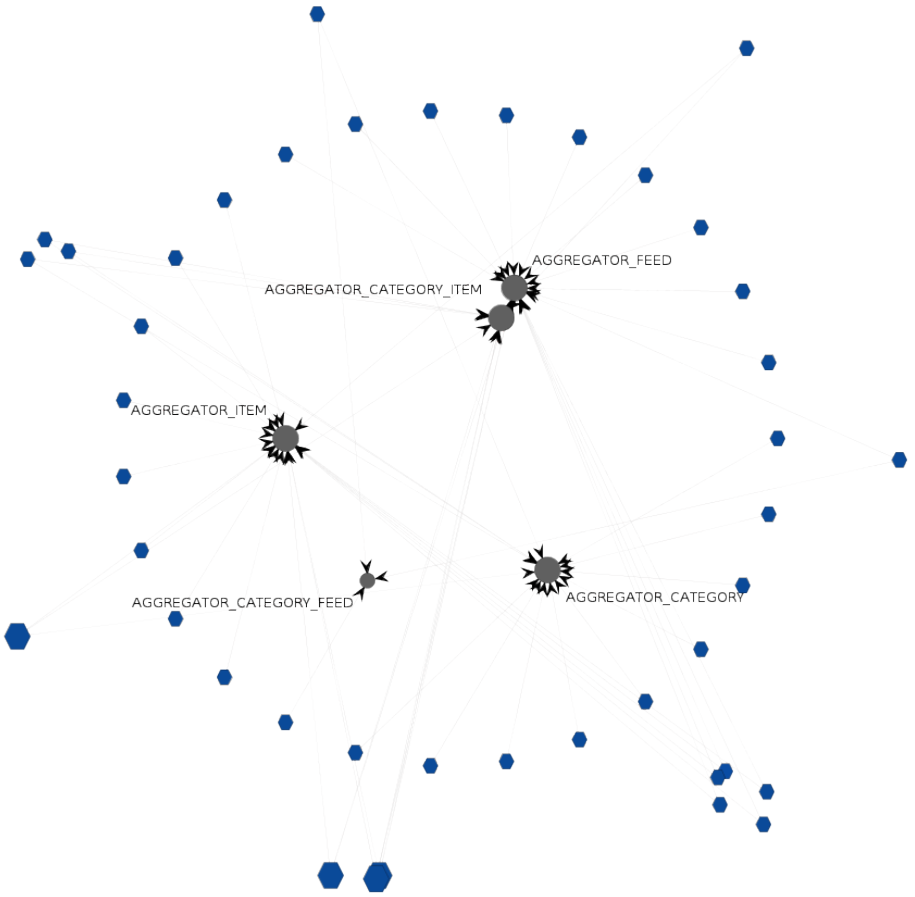


Figure 5.15: Drupal 4.1.0 cluster with queries asking same tables as arcs.

Experimental Protocol. We have used the following sequence of actions. First, we annotate the architecture graph with policies. Next, we sequentially apply the events over the graph – i.e., each event is applied over the graph that resulted from the application of the previous event. For each event we measure the elapsed time for each of the three algorithms, along with the number of affected, cloned and adapted modules. The experiment was performed in a typical PC with an Intel i5 CPU at 2.90GHz and 32GB main memory and only one core being used.

Annotation effort

The "real world" experiment was conducted using the syntactic sugar policy annotation rules. When we used the setup that is described in Section 5.5.2 we did not have to write any rules (the default one is generated by our tool). When we used the setup that is described in Section 5.5.2, we had to write only one rule, in order to block the propagation of events to the extra query we deliberately inserted in the ecosystem.

Effort gains

In both variants of our experiments, we can see that the effectiveness is way too high for all events. This is because the average number of affected modules is small compared to the size of the graph. More precise results about this experimental setup you may see in Table 5.7, where we notice that the minimum benefit for the developers is 71% while the average benefit ranges between 91% - 98.5%!

Metric		Version 4.*					
		1.0	2.0	3.1	4.3	5.8	6.11
%AM	min	97.1	88.2	94.2	71	93.1	79.7
	avg	**98.2**	**97.3**	**96.8**	**91**	**98.5**	**97.3**
	max	99.2	98.8	99.6	99.6	100	99.7
RM	min	2	4	2	2	1	2
	avg	**5.4**	**7.8**	**9.3**	**24.7**	**5**	**10**
	max	8	30	16	77	20	68

Table 5.7: Results of the original evolution scenario of Drupal

In the experimental setup that is described in Section 5.5.2, we can see that the metrics have not changed significantly. Also due to the blocker query and the *UNview* modules, we now have clones! This way, the query that was marked to block all the changes remains intact, while the rest of the ecosystem evolves.

The minimum number of clones per event is 0. Also, since the height of our tree is only one level, the maximum number of clones per event can not be greater than 1. Those metrics are displayed in Table 5.8.

Metric		Version 4.*					
		1.0	2.0	3.1	4.3	5.8	6.11
%AM	min	97.1	88.5	94.3	70.7	93.2	79.5
	avg	**98.2**	**96.8**	**96.9**	**91**	**98.5**	**97.2**
	max	99.2	98.8	99.6	99.6	100	99.7
RM	min	3	4	2	2	1	2
	avg	**5.4**	**8.4**	**9.2**	**24.7**	**5**	**10**
	max	8	30	16	79	20	70
$\sum$ *Cloned*		0	4	1	3	3	5

Table 5.8: Results of the modified evolution scenario of Drupal

Efficiency assessment

The time needed to perform the adaptation of the ecosystem is practically negligible. Table 5.9 displays the time needed for the original Drupal experimental setup, described in Section 5.5.2. Table 5.10 displays the time needed for the modified Drupal experimental setup, described in Section 5.5.2. The experiments of the Drupal project were conducted with cold cache (it is interesting to note that in all occasions, the processing of the first event took an order of magnitude higher than the rest of the events; here we report the min, max and average of *all* events for each step).

5.5.3 Controlled experiment with TPC-DS

To better control and assess the behavior of our algorithms, we need a more complex environment than Drupal. In fact, our experience with several CMS's reveals that the internal structure of the database is intentionally kept as simple as possible, obviously in an attempt to maximize performance. Thus, we have employed a decision support benchmark, TPC-DS, as the testbed for our controlled experiment. We start with a description of the experimental setup.

Step		Version 4.*					
		1.0	2.0	3.1	4.3	5.8	6.11
1	min	110	171	65	56	37	76
	avg	**1311**	**1732**	**1135**	**913**	**678**	**728**
	max	8048	8913	9035	8498	10426	11888
2	min	2	2	2	1	1	1
	avg	**4**	**4**	**7**	**7**	**5**	**8**
	max	11	15	25	28	24	64
3	min	105	154	76	48	29	59
	avg	**362**	**506**	**560**	**659**	**251**	**364**
	max	1282	1947	1773	2830	1812	1328

Table 5.9: Drupal project times (in microseconds) for “original” setup

Step		Version 4.*					
		1.0	2.0	3.1	4.3	5.8	6.11
1	min	120	323	81	46	40	124
	avg	**1357**	**2227**	**1241**	**929**	**632**	**686**
	max	8124	10782	9707	8957	87896	9706
2	min	3	3	3	1	1	1
	avg	**4**	**19**	**18**	**14**	**10**	**13**
	max	13	51	149	68	123	99
3	min	114	801	82	72	25	88
	avg	**395**	**8128**	**1443**	**2051**	**1316**	**2193**
	max	1364	15244	11917	13103	9177	11713

Table 5.10: Drupal project times (in microseconds) for “modified” setup

Experimental setup for TPC-DS

Ecosystem. We have employed TPC-DS, version 1.1.0 [99] as our experimental testbed. TPC-DS is a benchmark that involves star schemata of a company that has the ability to *Sell* and receive *Returns* of its *Items* with the following ways: (a) the *Web*, or, (b) a *Catalog*, or, (c) directly at the *Store*. Moreover the company keeps data of *Customers*, regarding their *Income* band, or their *Demographics* data and additionally keep data about the *Promotion* of their *Items*. To handle advanced SQL constructs in the queries of TPC-DS, we had to add views for the handling of WITH clauses and to make modifications to queries containing keywords as LIMIT, HAVING in order to remove parser-offending parts that Hecataeus' parser cannot handle. To test the effect of graph size to our method's efficiency, we have created 3 graphs with gradually decreasing number of query modules: (a) a large ecosystem, *WCS*, with queries using all the available fact tables, (b) an ecosystem *CS*, where the queries to WEB_SALES have been removed, and (c) an ecosystem *S*, with queries using only the STORE_SALES fact table.

Events. The event workload consists of 51 events simulating a real case study of the Greek public sector. See Table 5.11, left, for an analysis of the module sizes within each scenario. In Table. 5.11, right, we present the breakdown of the workload (listing the percentage of each event type as *pct*).

Policies. We have annotated the graphs with policies, in order to allow the management of evolution events. We have used two "profiles": (a) *MixtureDBA*, consisting of 20% of the relation modules annotated with BLOCK policy, and, (b) *MixtureAD*, consisting of 15% of the query modules annotated with BLOCK policy. The first profile corresponds to a developer-friendly DBA that agrees to prevent changes already within the database. The second profile tests an environment where the application developer is allowed to register veto's for the evolution of specific applications (here: specific queries). We have taken care to pick queries that span several relations of the database.

	Graph size			*Event type*	*percentage*
	S	CS	WCS	Attribute Add	37.3%
Queries	27	68	89	Attribute Rename	43.2%
Views	25	48	95	Attribute Del	13.7%
Relations	25	25	25	Relation Rename	1.9%
Sum	**77**	**141**	**218**	View alter semantics	3.9%

Table 5.11: Experimental configuration for the TPC-DS ecosystem

Experimental Protocol. We have used the following sequence of actions. First, we annotate the architecture graph with policies. Next, we sequentially apply the events over the graph – i.e., each event is applied over the graph that resulted from the application of the previous event. The experiment was performed with hot cache in order to measure the time. For each event we measure the elapsed time for each of the three algorithms, along with the number of affected, cloned and adapted modules. The experiment was performed in a typical PC with an Intel Quad core CPU at 2.66GHz and 1.9GB main memory with only one core was being used.

Effectiveness assessment: How useful is our method for the application developers and the DBA's?

In this subsection, we discuss the evaluation of the effort gain metrics for our controlled experiment. We evaluated the %AM metric for each of the 51 events of the workload, performed over all three ecosystems (S, CS, WCS) and for both the policy annotation profiles ($MixtureDBA$ and $MixtureAD$). In the upper part of Table 5.12 we demonstrate the minimum, average and maximum value of the %AM metric for all these 51 runs, organized annotation policy and ecosystem. The results demonstrate that the effort gains compared to the absence of our method are significant, as, on average, the effort is around 90% in the case of the AD mixture and 97% in the case of the DBA mixture. As the graph size increases, the benefits from the highlighting of affected modules that our method offers, increase too. Observe that in the case of the DBA case, where the flooding of events is restricted early enough at the database's relations, the minimum benefit in all 51 events ranges between 60% - 84%.

	%AM - Mixture AD			*%AM - Mixture DBA*		
	S	CS	WCS	S	CS	WCS
min	21%	35%	30%	60%	78%	84%
avg	**89%**	**91%**	**92%**	**97%**	**96%**	**97%**
max	100%	100%	100%	100%	100%	100%
	RM - Mixture AD			*RM - Mixture DBA*		
	S	CS	WCS	S	CS	WCS
min	1	0	0	0	0	0
avg	**6.22**	**10.00**	**13.47**	**2.47**	**5.00**	**6.22**
max	38	66	117	22	27	30

Table 5.12: Effectiveness assessment as fraction of affected modules (%AM) and number of rewritten modules (RM) of the "controlled" experiment

Likewise, we evaluated the RM metric for each of the 51 events of the workload.

The results demonstrate that the minimum number of modules needing a rewrite is 0 for almost all combinations of mixture and graph size for the event workload. This happened in both the $MixtureDBA$ and the $MixtureAD$ cases for different reasons – still both related to a veto. In the case of $MixtureDBA$, if a relation vetoes a possible change to it, then the event is immediately blocked and no rewriting or cloning takes place. Similarly, if a query vetoes a change in a relation (eg. attribute deletion), again, the event is frozen no rewriting or cloning takes place. At the same time, observe that the average and maximum number of modules needing a rewrite increases as the size of the graph increases. This is expected, as the increase in the graph size signifies that the new queries can possibly use some of the tables/views of the smaller graph (remember that the graphs are constructed by adding views and queries each time). Then, every event affects more modules as the graph increases. Another worth-mentioning fact is that when the $MixtureDBA$ policy is used, the number of the modules needing rewrite drops, since the flooding of events is restricted early enough, inside the database.

Policy annotation effort: how many rules does one have to write in order to work with our what-if analysis tool?

In this subsection, we discuss the effort of the user for the annotation of the $Architecture$ $Graph$ ecosystem with policies, over the conducted controlled experiments. We have worked with both policy mixtures and observed the effort needed as the number of blockers increases. Remember that in the MixtureDBA policy of the "controlled" experiment we block events at relations; we have set 20% of our *relation* modules to block the events that they receive and kept the size of the *relation* modules is the same in all three experiments (*S*, *CS*, and *WCS*). In the MixtureAD policy –in the same experimental setup– we set about 15% of the *query* modules as blockers. Here, the number of the blockers depends on the numbers of the query modules, which is different for each one of the *S*, *CS*, and *WCS* experiments.

Table 5.13 displays our results. We have one column for the MixtureDBA and one column per size (*S*, *CS*, *WCS*) for the MixtureAD policy. The first rows explain the annotation policy, the nature and number of interesting modules (relations in the first case and queries in the latter) and the number of blockers within each configuration. The next three rows explain the effort spent to annotate the ecosystem without the syntactic sugar: we list the number of default rules that have to be

	Mixture	DBA	AD	AD	AD
	Size	all	S	CS	WCS
	Modules of interest	Relations	Queries	Queries	Queries
	Modules	25	27	68	89
	Blockers	5	4	10	12
Without syntactic sugar	Default	33	33	33	33
	Extra	25	36	90	118
	Total	58	69	123	151
With syntactic sugar	Default	**1**	**1**	**1**	**1**
	Extra	5	4	10	12
	Total	**6**	**5**	**11**	**13**

Table 5.13: Modules and rules for policy annotation effort.

defined for completeness reasons, the extra rules that pertain to the individual blocker modules and the sum of these two measures. Finally, the last group of columns, refers to the case where the syntactic sugar was available, in a manner similar to the previous.

For the case where the syntactic sugar was not used, we have to mention the following observations and clarifications. At first, the number of default rules (33) seems quite high. However, we should also mention that Hecataeus supports the user gracefully by offering the template list ready to the user. At the same time, the number of extra rules per blocking module is about 9 rules per module. Although the numbers for the entire ecosystem reach to a high level, in the regular case where the annotation is performed in an incremental fashion, the ratio of 9 rules per module seems quite tolerable.

For the case where we have exploited the syntactic sugar, the set of rules needed decreases dramatically. This is due to the dramatic decrease in both the default rules (1 rule only) and the necessary rules per module (again one rule only). Specifically, observe that we can annotate with PROPAGATE policies the entire graph using only one rule (*NODE: ON * THEN PROPAGATE;*), and for each one of the blocker modules, we need to use, again, only one rule (*<namedNode>: ON * THEN BLOCK;*). Overall, the savings in effort and the speedup are too high both in batch and incremental ways of using our method.

Efficiency: how fast can we interact with our what-if analysis tool?

In this subsection, we evaluate the time needed to complete the process of what-if analysis with our tool. Efficiency plays an important role in the design and administration process of an ecosystem, if we wish to allow the involved stakeholders to interactively test alternative configurations and scenarios for policy annotations or restructuring of the ecosystem's architecture to accommodate forthcoming changes gracefully. To this end, we investigate the effect of policy annotation and graph size to the completion time and its breakdown in the three phases of our method.

Effect of policy to the execution time. In the case of *Mixture DBA* we follow an aggressive blocking policy that stops the events early enough, at the relations, before they start being propagated in the ecosystem. On the other hand, in the case of *Mixture AD*, we follow a more conservative annotation approach, where the developer can assign blocker policies only to some module parts that he authors. In the latter case, it is clear that the events are propagated to larger parts of the ecosystem resulting in higher numbers of affected and rewritten nodes. If one compares the execution time of the three cases of the AD mixture in Fig. 5.16 with the execution time of the three cases of the DBA mixture, the difference is in the area of one order of magnitude. It is however interesting to note the internal differences: the status determination time is scaled up with a factor of two; the rewriting time, however is scaled up by a factor of 10, 20 and 30 for the small, medium and large graph respectively!

Another interesting finding concerns the **internal breakdown of the execution time** in each case. A common pattern is that *path check is executed very efficiently*: in all cases it stays within 2% of the total time (thus practically invisible in the graphic). In the case of the AD mixture, the analogy between the status determination and the graph rewriting part starts from a 24% - 74% for the small graph and ends to a 7% - 93% for the large graph. In other words, *as the events are allowed to flow within the ecosystem, the amount of rewriting increases with the size of the graph*; in all cases, it dominates the overall execution time. This is due to the fact that rewriting involves memory management (module cloning, internal node additions, etc) that costs much more than the simple checks performed by *Status Determination*. In the case of the DBA mixture, however, where the events are quickly blocked, the times are not only significantly smaller, but also equi-balanced: 57% - 42% for the small graph (*Status*

Determination costs more in this case) and 49% - 50% for the two other graphs. Again, this is due to the fact that the rewriting actions are the time consuming ones and therefore, their reduction significantly reduces the execution time too.

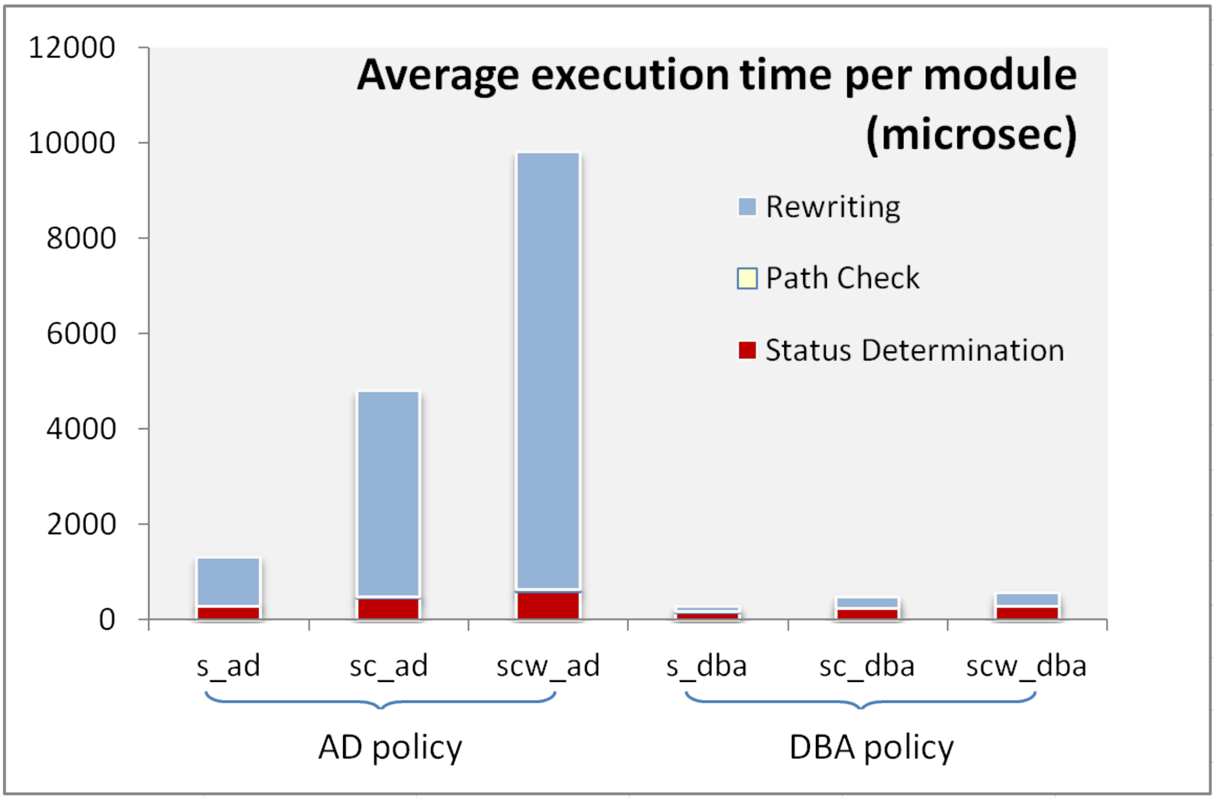


Figure 5.16: Efficiency assessment for different policies, graph sizes and phases

Effect of graph size to the execution time. To assess the impact of graph size to the execution time one has to compare the three different graphs to one another within each policy. In the case of the AD mixture, where the rewriting dominates the execution time, there is a linear increase of both the rewriting and the execution time with the graph size. On the contrary, the rate of increase drops in the case of the DBA mixture: when the events are blocked early, the size of the graph plays less role to the execution time.

Overall, the main lesson learned from these observations is that the annotation of few database relations significantly restricts the rewriting time (and consequently the overall execution time) when compared to the case of annotating modules external to the database. In case the rewriting is not constrained early enough, then the execution cost grows linearly with the size of the ecosystem.

5.6 Conclusions

In this chapter we have addressed the problem of adapting a data-intensive ecosystem in the presence of policies that regulate the flow of evolution events. Our method allows (a) the management of alternative variants of views and queries, and, (b) the rewriting of the ecosystem's affected modules in a way that respects the policy annotations and the correctness of the rewriting (even in the presence of policy conflicts). Our experiments confirm that the adaptation is performed efficiently as the size and complexity of the ecosystem grow. All the material for this work, including input ecosystems, links to the source code (publicly available at git) and results can be found in the following web page: `http://www.cs.uoi.gr/~pmanousi/publications/2013_ER/index.html`.

We continue with the visualization of the *Architecture Graph*, and a number of circular placement algorithms that depict the graph, as well as the affected nodes when a "what-if" scenario has run.

Chapter 6

Data-Intensive Ecosystem Visualization

6.1 Introduction

So far, we have described a method to obtain the queries of a data-intensive project from it's source code, proposed a set of properties to describe a well constructed data-intensive ecosystem, based on the database schema of the project, and, provided a way to provide better formed queries so as to achieve higher maintainability in a project. To achieve those tasks, we have introduced a number of rigorous constructs, one of which is the *Architecture Graph* that was introduced in Chapter 5 and describes the map of how software is coupled to the database schema. As we are using a graph, a necessity is to be able to depict it, hiding any visual clutter that exists due to the great number of edges, and if applicable to depict extra information of the projects' structure via its visualization. In this chapter we describe a set of visualization algorithms that perform the aforementioned tasks and an evaluation of those algorithms with a user study.

Developers of data-intensive ecosystems construct applications that rely on underlying databases for their proper operation, as they typically represent all the necessary information in a structured fashion in them. The symbiosis of applications and databases is not balanced, as the latter act as "dependency magnets" in these environments: databases do not depend upon other modules although being heavily depended upon, as database access is performed via queries specifically using the structure of the underlying database in their definition.

On top of having to deal with the problem of tight coupling between code and data, developers also have to address the disperse location of the code with which they work, in several parts of the code base. To quote [78] (the emphasis is ours): "Programmers spend between 60-90% of their time reading and navigating code and other data sources ...*Programmers form working sets of one or more fragments corresponding to places of interest* ...Perhaps as a result, *programmers may spend on average 35% of their time in IDEs actively navigating among working set fragments*..., since they can only easily see one or two fragments at a time."

The aforementioned two observations (code-data dependency and contextualized focus in an area of interest) have a natural consequence: developers would greatly benefit from the possibility of jointly exploring database constructs and source code that are tightly related. E.g., in the development and maintenance of a software module, the developer is interested in a specific subset of the database tables and attributes, related to the module that is constructed, modified or studied. Similarly, when working or facing the alteration of the structure of the database (e.g., attribute deletions or renaming, table additions, alteration of view definitions), the developer would appreciate a quick reference to the set of modules impacted by the change.

This *locality of interest* presents a clear call for the construction of a map of the system that allows developer to understand, communicate, design and maintain the code and its internal structure better. However, although (a) circular graph drawing methods have been developed for the representation general purpose graphs [83], [86], [84], and, (b) visual representations of the structure of code have been used for many decades [76], [75], [78], [79], the representation of data-intensive ecosystems has not been adequately addressed so far.

The research question that this chapter addresses is the provision of a visual map of the ecosystem that highlights the correlation of the developed code to the underlying database in a way that supports the locality of interest in operations like program comprehension, impact

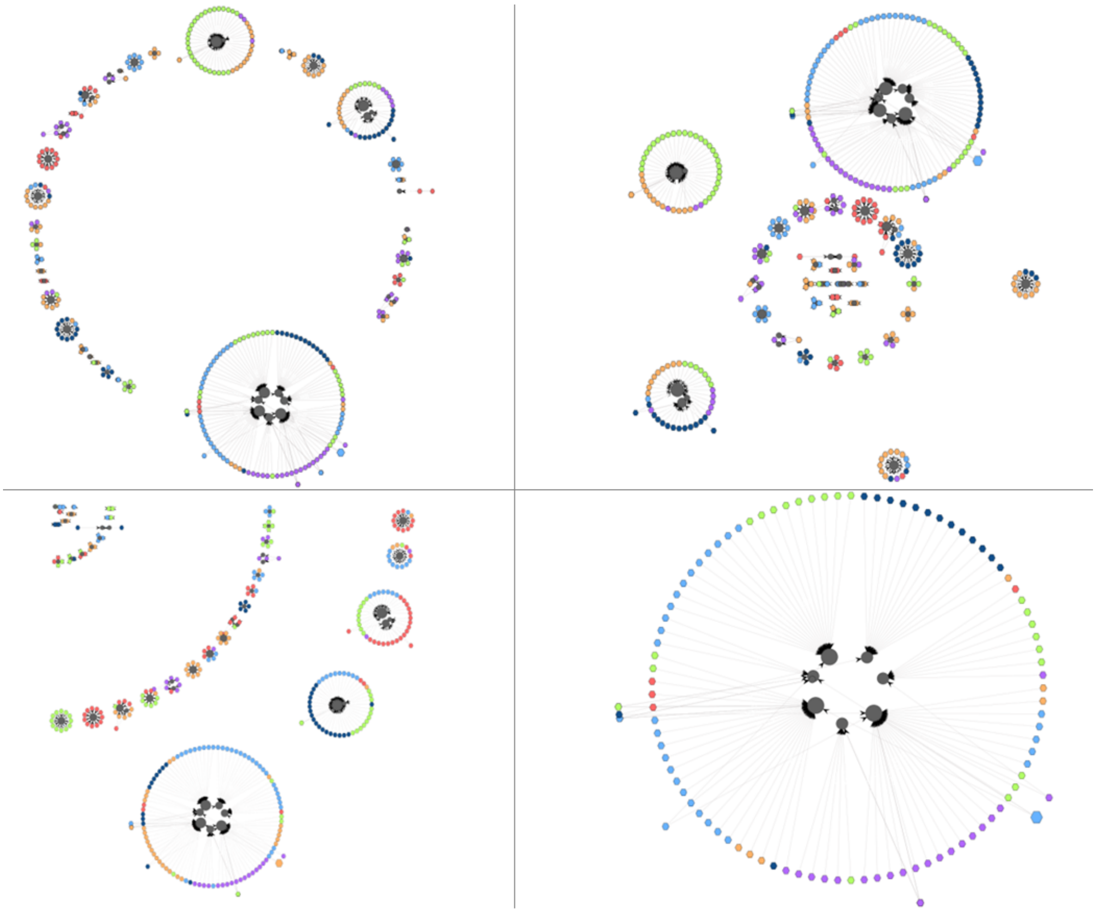


Figure 6.1: Alternative visualizations for Drupal. Upper Left: Circular layout; Upper Right: Concentric circles; Lower Left: Concentric Arches. Lower Right: zoom in a cluster of Drupal

analysis (for potential changes at the database layer), documentation etc.

Our method visualizes the ecosystem as a graph where all modules are modelled as nodes of the graph and the provision of data from a database module –e.g., a table– to a software module is denoted by an edge. To automatically detect "regions" of the graph with dense interconnections (and to visualize them accordingly) we cluster the ecosystem's nodes. Then, we present three circular graph drawing methods for the visualization of the graph (see Fig. 6.1). Our first method places all clusters on a embedding "cluster" circle, our second method splits the space in layers of concentric circles and our last method employs concentric arcs. In all our methods, the internal visualization of each cluster involves the placement of relations, views and queries in concentric circles, in order to further exploit space and minimize edge crossings.

This Thesis extends the work of [1] that describes the Section 6.2 with the results of the visualization of impact analysis and the user study evaluation of the three methods of [1] plus the "what-if" visualization.

6.2 Graph Layout Methods for Data-Intensive Ecosystems

The fundamental modeling pillar upon which we base our approach is the *Architecture Graph* $G(V, E)$ of a data-intensive ecosystem. The Architecture Graph is a skeleton, in the form of graph, that traces the dependencies of the application code from the underlying database. In our previous research [33], we have employed a detailed representation of the queries and relations involved; in this paper, however, it is sufficient to use a summary of the architecture graph as a zoomed-out variant of the graph that comprises only of *modules* (relations, views and queries) as nodes and edges denoting data provision relationships between them. Formally, a *Graph Summary* is a directed acyclic graph $G(V, E)$ with V comprising the graph's module nodes and E comprising relationships between pairs of data providers and consumers.

In terms of visualization methods, *the main graph layout we use is a circular layout*. Circular layouts are beneficial due to a better highlight of node similarity, along with the possibility of minimizing the clutter that is produced by line intersections. We place clusters of objects in the periphery of an embedding circle or in the periphery of several concentric circles or arches. Each cluster will again be displayed in terms of a set of concentric circles, thus producing a simple, familiar and repetitive pattern.

Our method for visualizing the ecosystem is based on the principle of *clustered graph drawing* and uses the following steps:

1. Cluster the queries, views and relations of the ecosystem, into clusters of related modules. Formally, this means that we partition the set of graph nodes V into a set of disjoint subsets, i.e., its clusters, $C_1, C_2, \ldots, C_n$.

2. Perform some initial preprocessing of the clusters to obtain a first estimation of the required space for the visualization of the ecosystem.

3. Position the clusters on a two-dimensional canvas in a way that minimizes visual clutter and highlights relationships and differences.

4. For each cluster, decide the positions of its nodes and visualize it.

6.2.1 Clustering of Modules

In accordance with the need to highlight locality of interest and to accomplish a successful visualization, it is often required to reduce the amount of visible elements being viewed by placing them in groups. This reduces visual clutter and improves user understanding of the graph as it applies the principle of proximity: similar nodes are placed next to each other. To this end, in our approach we use clustering to group objects with similar semantics in advance of graph drawing.

We have implemented an average-link agglomerative clustering algorithm [100] of the graph's nodes, which starts with each node being a cluster on its own and iteratively merges the most similar nodes in a new cluster until the node list is exhausted or a sued-defined similarity threshold is reached.

The distance function used in our method evaluates node similarity on the grounds of common neighbours. So, for nodes of the same type (e.g., two queries, or two tables), similarity is computed via the Jaccard formula, i.e., the fraction of the number of common neighbours over the size of the union of the neighbours of the two modules. When it comes to assessing the similarity of nodes of different types (like, e.g., a query and a relation), we must take into account whether there is an edge among them. If this is the case, the nominator is increased by 2, accounting for the two participants. Formally, the distance of two modules, i.e., nodes of the graph, M_i, M_j is expressed as:

$$dist(M_i, M_j) = 1 - \begin{cases} \dfrac{|neighbors_i \cap neighbors_j|}{|neighbors_i \cup neighbors_j|}, & if \ \nexists \ Edge(i,j) \\ \\ \dfrac{|neighbors_i \cap neighbors_j| + 2}{|neighbors_i \cup neighbors_j|}, & if \ \exists \ Edge(i,j) \end{cases} \tag{6.1}$$

6.2.2 Cluster Preprocessing

Our method requires the computation of the area that each cluster will possess in the final drawing. In our method, each cluster is constructed around three bands of concentric circles: an innermost circle for the relations, an intermediate band of circles for the views (which are stratified by definition, and can thus, be placed in strata) and the outermost band of circles for the queries that pertain to the cluster. The latter includes two circles: a circle of *relation-dedicated queries* (i.e., queries that hit a single relation) and an outer circle for the rest of the queries. This heuristic is due to the fact that in all the studied datasets, there was a vast majority of relation-dedicated queries; thus, the heuristic allows a clearer visualization of how queries access relations and views.

In order to obtain an estimation of the required space for the visualization of the ecosystem, we need to perform two computations. First, we need to determine the circles of the drawing and the nodes that they contain (this is obtained via a topological sort of the nodes and their assignment to strata, each of which is assigned to a circle), and second, we need to compute the radius for each of these circles (obtained via the formula $R_i = 3 * \log(nodes) + nodes$). Then, the outer of these circles gives us the space that this cluster needs in order to be displayed.

6.2.3 Layout of Cluster Circle(s)

We propose three alternative circular layouts for the deployment of the graph on a 2D canvas.

Circular cluster placement with variable angles.

In this method, we use a single circle to place circular clusters on. As already mentioned, we have already calculated the radius r of each cluster. Given this input, we can also compute R, the radius of the embedding circle. We approximate the contour of the inscribed polygon of the circle, computed via the sum of twice the radius of the clusters by the perimeter of the embedding circle, which is equal to $2\pi * R$ (Fig. 6.2). We take special care that the layouts of the different clusters do not overlap; to this end, we introduce a white space factor w that enlarges the radius R of the cluster circle (typically, we use a fixed value of 1.8 for w). Then, $R = \sum_{i=0}^{|C|} \frac{2 * \varrho_i}{2\pi * w}$, where C is the set of clusters, and ϱ_i the radius of cluster i. As the arc around which each cluster will be placed is expanded, this leaves extra whitespace between the actually exploited parts of the clusters' arcs. Given the above inputs, we can calculate the angle ϕ that determines the sector of a given cluster, as well as its center coordinates (c_x, c_y) via the following equations:

$$\phi = 2 * \arccos\left(\frac{2 * R^2 - \varrho^2}{2 * R^2}\right), \; c_x = \cos\left(\frac{\phi}{2}\right) * R * w, \; c_y = \sin\left(\frac{\phi}{2}\right) * R * w \qquad (6.2)$$

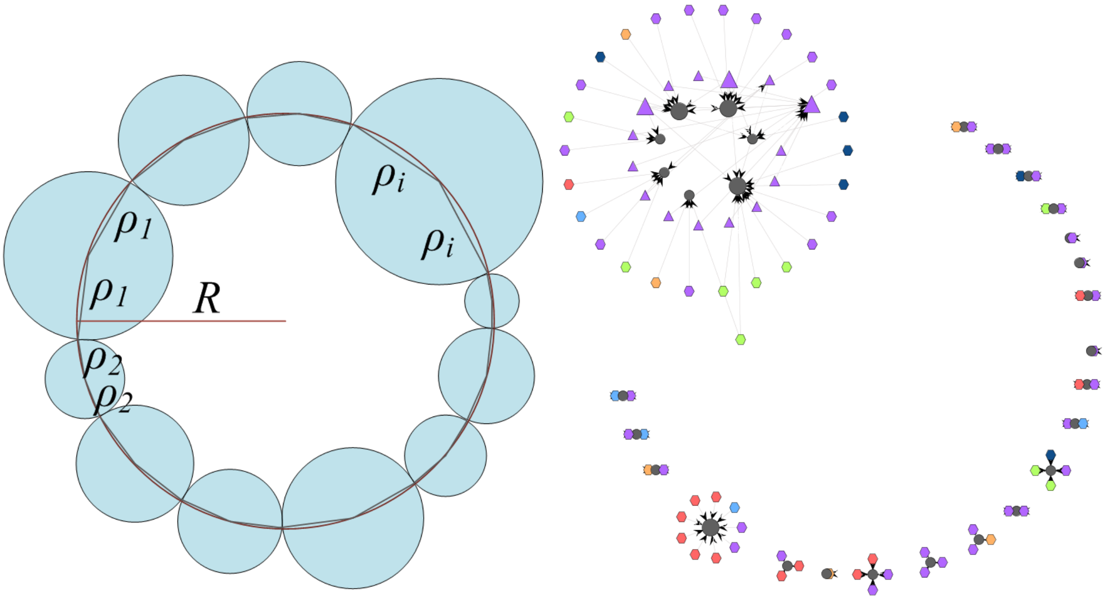


Figure 6.2: Circular cluster placements (left) and the BioSQL ecosystem (right)

Concentric cluster placement.

This method involves the placement of clusters to concentric circles. Each circle includes a different number of segments, each with a dedicated cluster. The proposed method obeys the following steps:

1. Sort clusters by ascending size in a list L^C

2. While there are clusters not placed in circles
    (a) Add a new circle and divide it in as many segments as $S = 2^k$, with k being the order of the circle (i.e., the first circle has 2^1 segments, the second 2^2 and so on)
    (b) Assign the next S fragments from the list L^C to the current circle and compute its radius according to this assignment
    (c) Add the circle to a list L of circles
3. Draw the circles from the most inward (i.e., from the circle with the least segments) to the outermost by following the list L.

Practically, the algorithm expands a set of concentric circles, split in fragments of powers of 2 (Fig. 6.3). As the order of the introduced circle increases, the number of fragments increases too ($S = 2^k$), with the exception of the outermost circle, where the segments are equal to the number of the remaining clusters. By assigning the clusters in an ascending order of size, we ensure that the small clusters will be placed on the inner circles, and we place bigger clusters on outer circles since bigger clusters occupy more space.

Radius Calculation. We need to guarantee that clusters do not overlap. This can be the result of two problems: (a) clusters of subsequent circles have radii big enough, so that they meet, or, (b) clusters on the same circle are big enough to intersect. To solve the first problem, we need to make sure that the radius of a circle is larger than the sum of (i) the radius of its previous circle, (ii) the radius of its larger cluster, and (iii) the radius of the larger cluster of the current circle. For the second problem, we compute R_i as the encompassing circle's periphery ($2 * \pi * R_i$) that can be approximated the sum of twice the radii of the circle's clusters. Then, to avoid the overlapping of clusters, we set the radius of the circle to be the maximum of the two values produced by the aforementioned solutions and we use an additional whitespace factor w to enlarge it slightly (typically, we use a fixed value of 1.2 for w).

$$R_i = w * \max \begin{cases} R_{i-1} + b_{i-1} + b_i \\ \\ \frac{1}{\pi} * \sum_{j=1}^{|C|} \varrho_j \end{cases} \tag{6.3}$$

where (a) b_α: is the rad of biggest cluster of circle α, and (b) ϱ_j: is the rad of cluster c_j which is part of C, where C is the set of clusters of circle i.

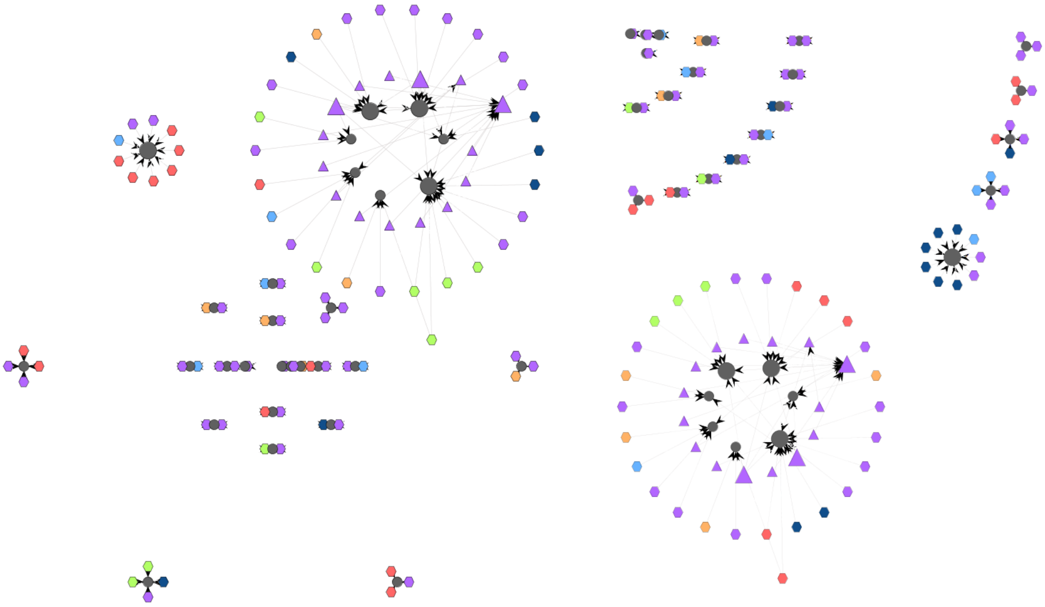


Figure 6.3: Concentric cluster placement for BioSQL: circles (left), arcs (right)

Clusters on concentric arches

It is possible to layout the clusters in a set of concentric arcs, instead of concentric circles (Fig. 6.3). This provides better space utilization, as the small clusters are placed upper left and there is less whitespace devoted to guard against cluster intersection. Overall, this method is a combination of the previous two methods. Specifically, (a) we deploy the clusters on concentric arches of size $\frac{\pi}{2}$, to obtain a more compact layout, and (b) we partition each cluster in proportion to the cluster's size by applying the method expressed by equation (6.2).

6.2.4 Layout of Nodes inside a Cluster

The last part of the visualization process involves placing the internals of each cluster within the area designated to the cluster from previous computations. As already mentioned, each cluster is aligned in terms of several concentric circles: an innermost circle for relations, a set of intermediate circles for views and one or more circles for queries, as we previously stated at section 6.2.2. Now, since the radii of the circles have been computed, what remains to be resolved is the order of nodes on their corresponding circle. We order relations via a greedy algorithm that promotes the adjacency of similar relations (i.e., sharing the large amount of views and queries). Once relations have been laid out, we place the rest of the views and queries in their corresponding circle of the cluster via a traditional barycenter-based method [101] that places a node in an angle that equals the average value of the sum of the angles of the nodes it accesses.

6.3 Visualization of impact analysis and zoom in of queries

The aforementioned visual maps provide the developer with multiple ways to view and explore the whole software ecosystem. They offer an overview for the developer to quickly identify blocks of codes, modules and data structures that are highly interconnected and thus need special attention when re-factoring or evolving parts of the system. When it comes to the evolution of the system, and the tasks that require more effort from the developer, such as the identification of the parts of code that need to change, impact analysis of the change, testing, validation of the change, etc., the visual representation must offer detailed representation of the structure of a module, such as an embedded query or a software module. For that, in this section, we present a method that helps the developer identify the specific parts of the code that change due to an evolution event. For example in Figure 6.4 we have a rename event that happened in BLOCK_ROLE relation. To help the developers that have to find those locations, we have implemented a simple, yet really helpful as we see later in Section 6.5, visualization method.

In this visualization method, we place the nodes to follow a stratification from left to right, based on their input dependencies. Leftmost part is for the queries that have no one using them, them they follow the view or the nested queries and finally we have the relations. Based on the placement of the high level nodes we arrange their low level nodes. Following the same tactic, we first place the nodes of the output schema, then, we continue to the semantics schema and finally we place the input schema. Observe in Figure 6.4 that for the first two queries besides the input schema, the output schema nodes (placed under the high level node) are also affected, whilst the last query does not use the attribute that changed in its output schema.

Looking further in the visualization of the low level nodes of a query, in Figure

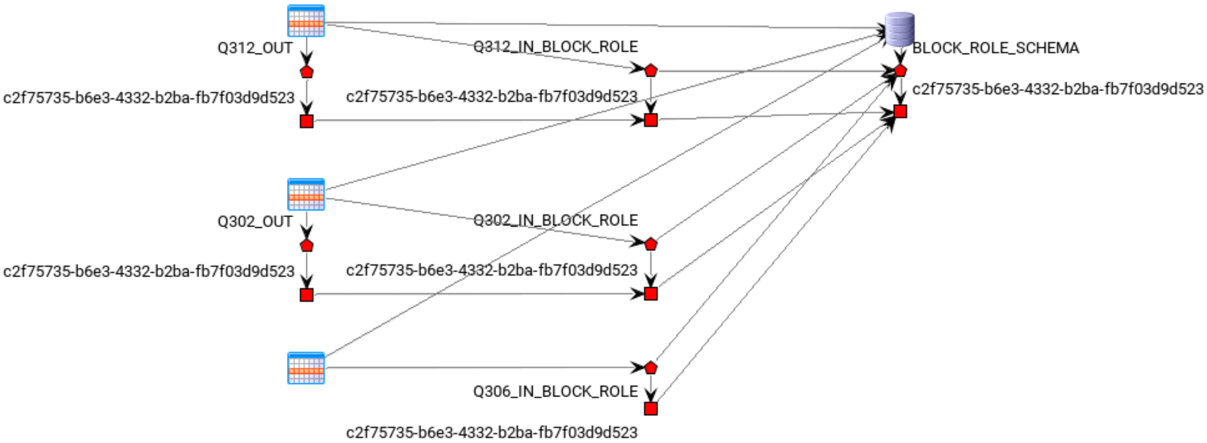

Q312_OUT
c2f75735-b6e3-4332-b2ba-fb7f03d9d523
Q312_IN_BLOCK_ROLE
c2f75735-b6e3-4332-b2ba-fb7f03d9d523
BLOCK_ROLE_SCHEMA
c2f75735-b6e3-4332-b2ba-fb7f03d9d523
Q302_OUT
c2f75735-b6e3-4332-b2ba-fb7f03d9d523
Q302_IN_BLOCK_ROLE
c2f75735-b6e3-4332-b2ba-fb7f03d9d523
Q306_IN_BLOCK_ROLE
c2f75735-b6e3-4332-b2ba-fb7f03d9d523

Figure 6.4: Zoom in a rename attribute impact analysis event of BLOCK_ROLE relation.

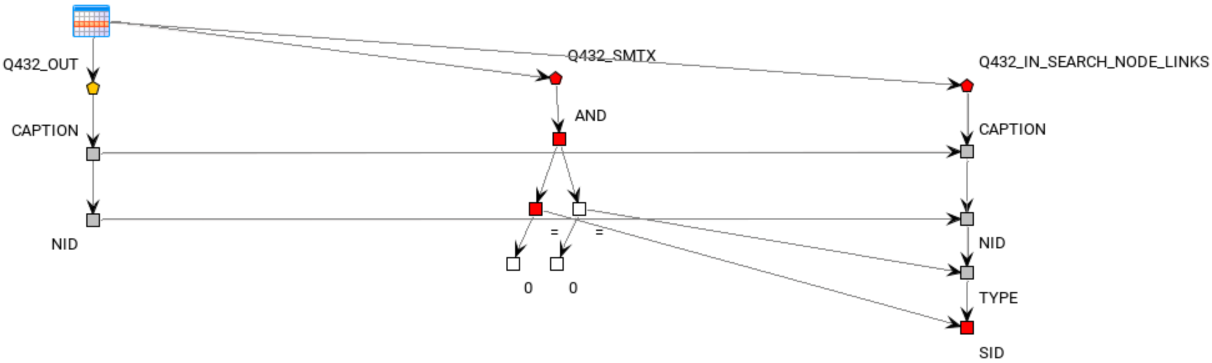


Figure 6.5: Zoom in a remove attribute impact analysis event of SEARCH_NODE_LINKS relation.

6.5 we depict the zoom in view of query Q432 that is having a WHERE clause statement. Using the previous idea, we implemented the semantics schema and its nodes as a tree, and we connect the leaves of the tree to the input nodes that they use in their conditions.

Additionally, in both cases, we keep in parallel the lines of the low level nodes, by placing the input attribute in the same screen height with the output attribute.

6.4 Experiments

In this section, we present our experimental assessment of the proposed visualization methods. We start with the discussion of the experimental method and then we assess our method against aesthetic criteria and objective measurements.

6.4.1 Experimental Method

In order to evaluate our work, we have used some well known open source projects that contain database queries. Table 6.1 provides a list of the projects. In order to convert the software of the analyzed tools to the graph representation that we use in this work, we performed a sequence of steps. We retrieved the database definition from the source code, using the algorithms and tools described in Chapter 3. Then we post-processed the queries in order to be parsable by our tool, Hecataeus that ultimately converts the ecosystem to an architecture graph and visualizes it for the user.

Dataset	Version	Description	R	V	Q	E
BioSQL	1.0	A generic relational schema covering sequences, features, sequence and feature annotation, a reference taxonomy, and ontologies	28	15	79	104
ZenCart	1.3	An open source shopping cart software	106	0	149	158
Drupal	7.41	An open source content management platform	75	0	355	379
OpenCart	1.5.6.1	An open source shopping cart software	115	0	650	824

Table 6.1: Datasets Used (R: Relations, V: Views, Q: Queries, E: Edges)

6.4.2 Assessment of Objective Criteria

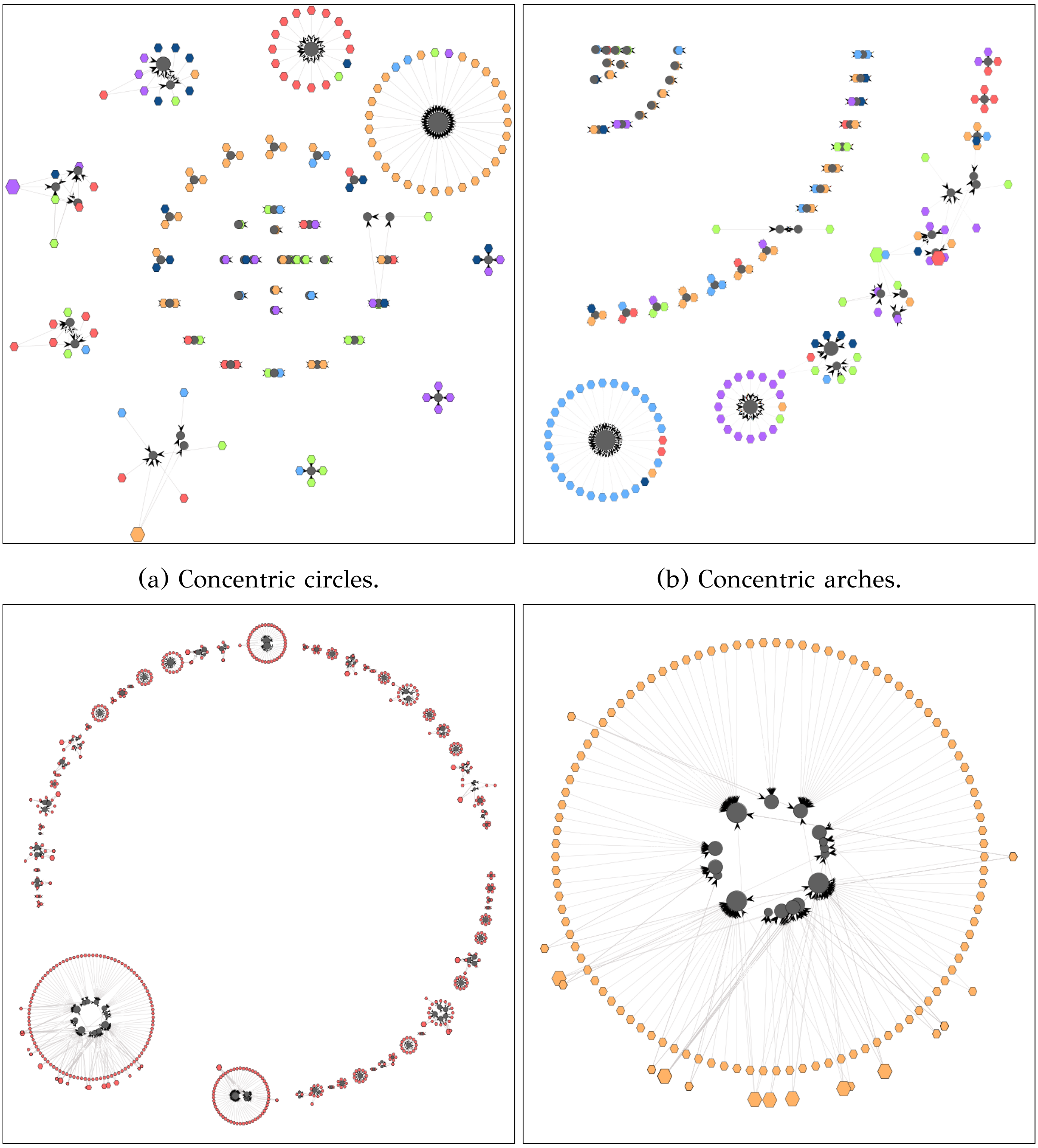
We assess the behavior of each of the proposed parts of our approach with specific objective criteria. First, we discuss what is the outcome and benefit from our clustering and preprocessing steps. Then, we assess the different methods in terms of how efficiently they exploit space.

Clustering Effectiveness. The first important result concerns the effectiveness of the clustering of the graph nodes. *In all the studied datasets, our clustering produced a clean separation of the graph in connected components that are completely disjoint and isolated*! In other words, clustering produced clusters that have no edges between their nodes (inter-cluster edges). This resulted in the elimination of all the visual clutter that these edges would incur. The second piece of good news is that the number of clusters ranges within 20 – 60 clusters in all four cases, thus it is presentable in a 2D screen canvas (see Table 6.2 for the exact numbers). We should, however, emphasize that as all the examined systems, except for BioSQL, are CMS's, the results should by no means be generalized to other types of information systems.

In Table 6.2, we give two objective criteria that are typical metrics used in all the related literature, specifically *number of edge crossings* and *average edge length*, as well as a metric that is specific to our method, and concerns the area covered by the clusters, expressed as *average cluster radius*. It is important to note that, *in the absence of inter-cluster edges due to our clustering, all these objective measures are independent from the visualization method* that follows the original clustering.

Method Effectiveness. A second important part of our method has to do with the effectiveness of the employed methods in terms of space utilization. For each method, we measure (a) the rectangle produced by the farthest pairs of coordinates, (b) the sum of the area covered from all the clusters (on the basis of their outermost circle), and, (c) the resulting percentage of area covered by the visualized clusters. Remember that the area covered depends on the particularities of the method; then, our system autoscales the result to fit in the screen. Thus, the area of the bounding rectangle is a clear indicator of the scale factor of the drawing: between two drawings that will ultimately have to fit in the same screen, the one with the larger area needed, will require more zoom-out.

Table 6.3 reports on the area needed for the visualization of the graph. Underlined blue shows the winner method that requires the least area and bold red highlights the



(a) Concentric circles.

(b) Concentric arches.

(c) Circular layout.

(d) Zoom in a cluster

Figure 6.6: Examples of ZenCart (upper) and OpenCart (lower)

Dataset	Number of clusters	Avg cluster radius	Avg cluster edge crossings	Avg edge length
BioSQL	22	12.00	3.50	92.11
ZenCart	41	11.85	0.46	69.21
Drupal	37	25.01	15.62	497.71
OpenCart	59	28.32	84.08	751.32

Table 6.2: Objective measures for all four data sets

worst performance. We observe that concentric circles always loses and the winner methods are divided, with concentric arches winning for the three smaller cases and circular layout for the largest one. At the same time, the amount of covered area is fixed for all methods (as cluster sizes are fixed before laying them out in the 2D canvas). The percentage of covered area, in all layouts is very small (2%-7% for Zencart and OpenCart, 7%-15% for Drupal and 16%-23% for BioSql).

Dataset	Circular layout	Conc. circles	Conc. arches	Covered area
BioSQL	196243	**275943**	193641	44550
ZenCart	2007636	**2162420**	1238296	50739
Drupal	2329122	**3232093**	1612675	253173
OpenCart	5775976	**18392055**	9711796	461425

Table 6.3: Area occupied by graph

6.4.3 Aesthetic criteria

In this subsection, we assess whether our major design choice of using circular layouts as well as the detailed technical choices for each of the proposed methods were appropriate according to the fundamental principles for visual object representations [80].

Proximity is achieved via clustering: we place nodes with similar semantics closely to each other (in our case nodes belonging to the same clusters of relations, views and queries). *Connectedness* is inherently achieved via the choice of graph representation.

Similarity and *proportion* are encompassed in our methods via several decisions. We use the same *shape* for nodes of same type (relations, queries, views), the same *color* for nodes that belong in similar physical structures. The queries that belong to the same files are coloured with the same color. If there exist files that are placed in the same folder, then those files get a light change of the color of the parent folder. The relations and the views are always gray and purple, respectively. We scale the *size* of each node according to its graph degree, so that larger size denotes larger degree of interrelationship.

Closure and *isolation* are also inherently supported via the idea of circular visualization of each cluster: the outermost circles of each cluster provide a visual border that separate it from other clusters. We take special care for clusters not to intersect and we enhance their surrounding space with intentionally injected white-space. The same care is paid for individual nodes too.

Clutter avoidance is one of the fundamental problems our methods try to battle. As the main source of visual clutter is the overwhelming presence of edges –especially, when they cross– we take every possible means to minimize the number of pixels-per-edge, without taking them out of the diagram, at the same time. To this end, we decrease the strength of edge coloring to light gray and highlighting neighbours only when a user interactively picks a node to inspect. The idea of clustering nodes, putting similar nodes closely tries to minimize the span of edges throughout the entire canvas. Within each cluster, we implement an adjustment of the barycentre method to radial layouts to minimize edge crossings. A second source of clutter is the overlapping of clusters or nodes: we intentionally tune the methods to avoid such phenomena.

Finally, we avoid any other *emphasis* of individual clusters or nodes and leave this aesthetic tool available for arbitrary usages where there will be a need for emphasizing some parts of the graph. We do not inject visual hierarchies, contrast, asymmetry or discontinuity to emphasize any part of the graph. We also avoid any special purpose focal points to guide the visual flow of the user's optical navigation over the screen. Flow occurs, however, in the concentric methods, as there is a flow from smaller towards larger clusters. This makes these methods more suitable for arrangements where cluster size is also important for the users' work.

6.4.4 Comparison to general purpose graph visualizations

In this section we will discuss alternative visualization methods and we compare them to our approach. Our visualizations were implemented with Jung, a software library that provides a common and extensible language for the modelling, analysis, and visualization of data that can be represented as a graph or network. Jung is written in Java, which allows Jung-based applications to make use of the extensive built-in capabilities of the Java API, as well as those of other existing third-party Java libraries. Jung supports several layouts, out of which we discern the following prominent ones:

- A simple, random Circle Layout (Fig. 6.7) that places vertexes randomly on a circle
- The Fruchterman-Reingold algorithm [102], denoted as FRLayout (Fig. 6.8)
- Meyer's "Self-Organizing Map" layout [103] denoted as ISOMLayout (Fig. 6.9)
- The Kamada-Kawai algorithm [104], denoted as KKLayout in Jung (Fig. 6.10),
- The SpringLayout (Fig. 6.11), which is a simple force-directed spring-embedder [101]

We applied all these layouts on our datasets to compare them with our layouts. The result is shown in the figures bellow. In Jung's circular layout all nodes are randomly placed on a periphery of a circle with radius $R = \frac{\#nodes}{2\phi}$. With this radius there should be no overlaps, however as we see in Fig. 6.7 nodes do overlap where in some parts of the circumference of the circle there are empty (white) spaces. In the remaining Jung layouts, the nodes of the graph appear to be randomly placed on the canvas.

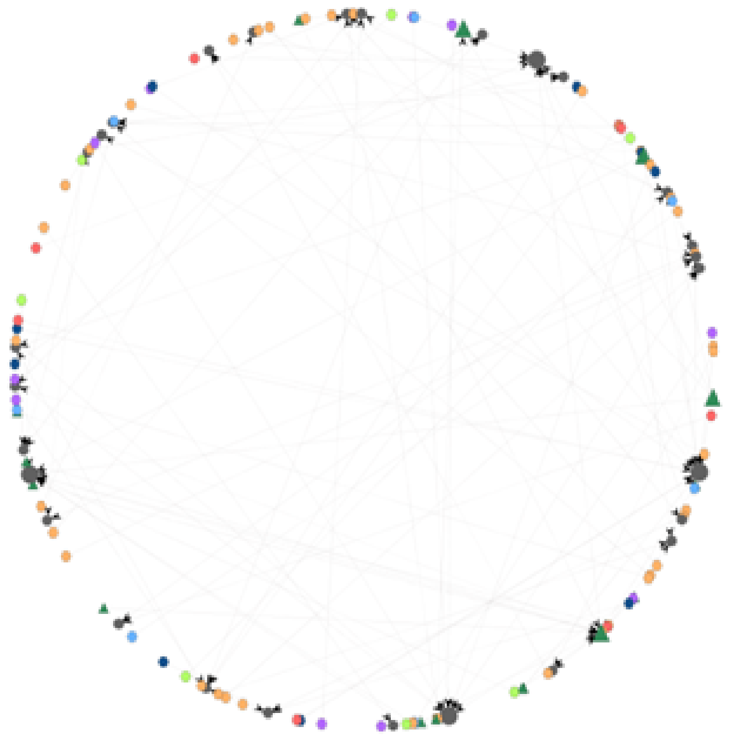


Figure 6.7: BioSql visualized via a circular algorithm by Jung.

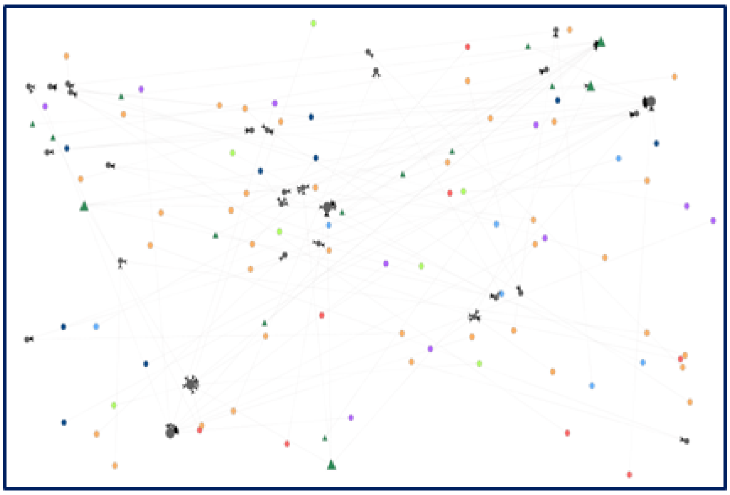


Figure 6.8: BioSql visualized via the FR algorithm by Jung.

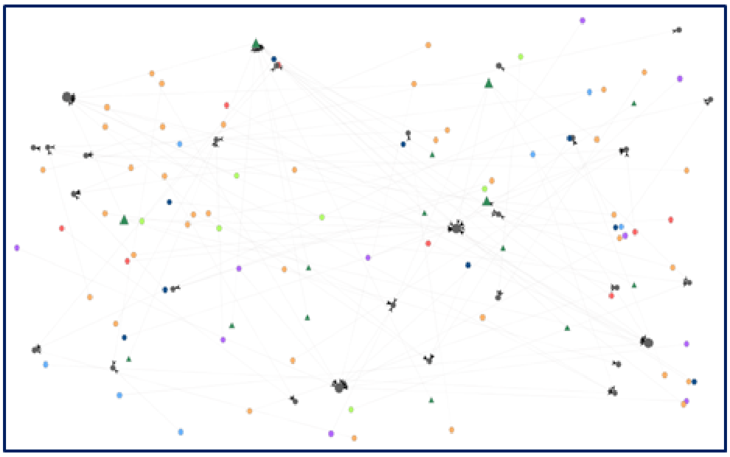


Figure 6.9: BioSql visualized via the Self-organizing algorithm by Jung.

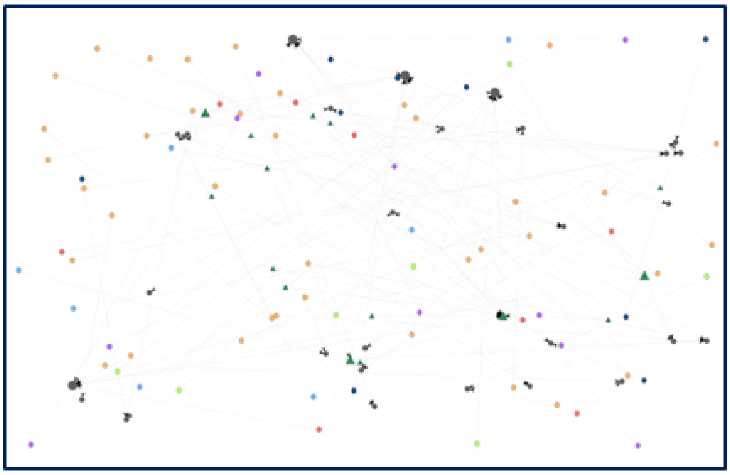


Figure 6.10: BioSql visualized via the KK algorithm by Jung.

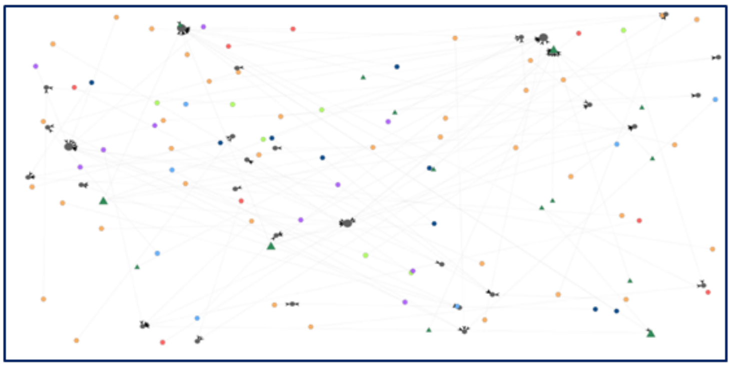


Figure 6.11: BioSql visualized via the spring layout algorithm by Jung.

6.5 User study evaluation

In order to evaluate how our tool can be used in real situations, we conducted a user study. The study referred to the context of *maintaining* the source code of an application, specifically Drupal 7.41, in the presence of schema evolution events at its underlying database. We believe that our tool, Hecataeus, is most important for designers and administrators that are not extensively familiarized to both the source code of the application and the database. This can happen in many settings, e.g., (a) when the developer team is different from the database administration team, (b) when a newcomer joins a team, or (c) when a new project starts, etc. Given this background, we have assessed our method, using the Drupal project as a testbed and a population of graduate students as maintainers. All of our maintainers had just completed a course of "software and database evolution". All of our maintainers have experience in programming, since they all performed a set of programming exercises in the course we mentioned earlier. Finally, all of our maintainers had a database related project to examine regarding its evolution.

Our protocol is as follows:

- We trained the maintainers to the Hecataeus tool. We demonstrated the tool, and we provided them a video that describes what one may do using Hecataeus.
- We assigned the questionnaire of Table 6.4 to all maintainers.
- We randomly split the maintainers in two groups, and we instructed the ones that were at the first group to initially work with Hecataeus and then with a tool of their own choice. The other group had to work in reverse order, using Hecataeus in the end.
- We asked the maintainers to complete the task and then fill an on-line questionnaire.

As we mentioned earlier the graph is parted of queries that were mined from the source code files of the project and the database schema which was also mined from the source code of the project. Drupal has no views, actually nearly all open source projects have no views, besides one (BioSQL). Therefore, when we mention files, at the same time we refer to nodes of the *Architecture Graph*.

Task 1	1. Which files change when we rename the COMMENT.CID to COMMENT.COMMENT_ID column? 2. How much time (in minutes) did you need to find which files change when we rename the COMMENT.CID to COMMENT.COMMENT_ID column?
Task 2	1. How easy was to keep you code unchanged on one file (comment.pages.inc), while adapt all others that use the COMMENT table, when you add a new column to it. 2. Measure the time needed (in minutes) to keep you code unchanged on one file (comment.pages.inc), while adapt all others that use the COMMENT table, when you add a new column to it.
Task 3	How easy is to find what specific parts of the queries have to change for the rename of the first question (COMMENT.CID to became COMMENT.COMMENT_ID)?
Task 4	Give a score from 0 to 5 on 1. your tools 2. Hecataeus' Single Circle 3. Hecataeus' Concentric Circles 4. Hecataeus' Concentric Arcs layouts on how helpful they are to find where views could be used in this project (a number of questions using simultaneously more than one tables)?

Table 6.4: Tasks that the participants of user study were asked to complete.

6.5.1 Effectiveness

The first task of the user study was designed with the goal to assess the effectiveness of our tool. The effectiveness is measured by the combination of the number of the files that the users found, and which files that the users found but are not correct (those are files that we deliberately injected in the source code of the project). The number of files that are to change due to the rename of COMMENT.CID to COMMENT.COMMENT_ID is 7. Additionally, we have measured the time it took to our maintainers to find the files.

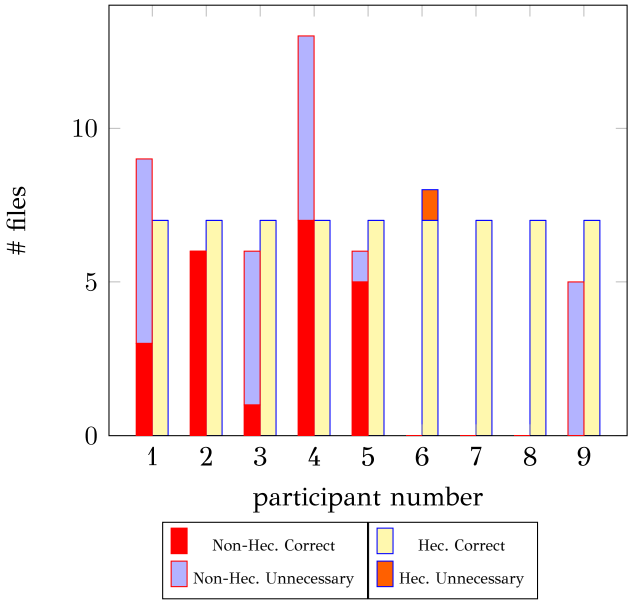


Figure 6.12: Effectiveness measured via correct and unnecessary files retrieved for maintenance by the users. Five of the users did not notice the information area of Hecataeus that stated which files changed, and used the highlight event of Hecataeus tool to find the files, by clicking on the COMMENT node, therefor they gave one additional file in their answer.

Observe that when using Hecataeus, all but one user (maintainer number 6) had a perfect score. On the other hand, when the users used the tools of their own choice, only one managed to find all the files (maintainer number 4), but he also reported a number of files (6) that were not containing source code as files that needed maintenance. Additionally, there were 3 people who did not manage to find any related files, stating that their tool had too much input for them to check. Regarding the exceptions:

- In the case when the users employed Hecataeus to detect the locations of impact of the schema change, we had one user who found an additional file on top of the correct ones. That user stated that he did not use the information area of our tool as the other participants did, but he used an effect of our tool, the "click to highlight" effect. This effect, when a node is clicked, highlights the nodes that are connected to it. Then, the participant reported those nodes in his answer.

- In the case where the users had used the tools of their choice, we had the majority of the users (5) to state, apart from the correct files, additional ones (or completely wrong, as participant number 9 did). That was because –as we already mentioned– we had intentionally added some commented out code that was using that table and one would need time to examine if this code is runnable or not.

Regarding the time measurement, Figure 6.13 describes in logarithmic scale the time needed (in minutes) for both Hecataeus and Non-Hecataeus tools that the participants used. Observe that using Hecataeus resulted in faster completion times in all cases besides one (the case of user number 4, who had found all the correct files plus 6 *additional*).

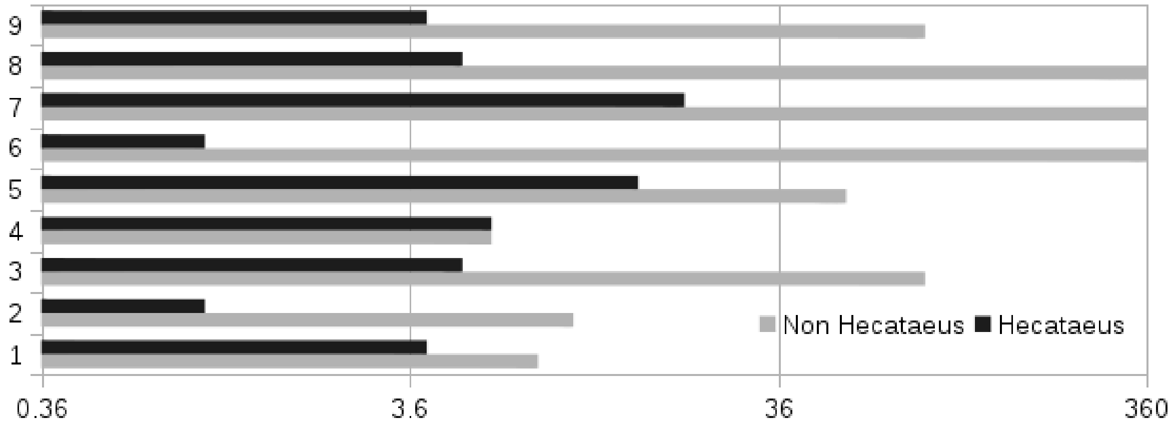


Figure 6.13: Time needed (in minutes) for other tools and Hecataeus. Tie was only in one situation where the result was wrong (6 additional files were reported that need maintenance).

6.5.2 User Satisfaction

The second task of the user study was designed to evaluate the satisfaction of users with our tool. We asked the users to perform a large number of changes in the database-related code. The maintainers had to change the code of the files that use the COMMENT table and add a new column, wherever this table was used except of one file (comment.pages.inc) which should remain unchanged, retaining its previous code. User satisfaction is measured by the user selection of a number ranging between 0 and 5. Figure 6.14 depicts the measurements of Task 2. Out of 9 users, 6 stated that Hecataeus was more useful on this task, and none said that his tool was better than Hecataeus. The average number for the user satisfaction in efficiency for Hecataeus regarding Task 2 was: 4.1 units of user satisfaction while the average number for the other tools was: 2.2 units of user satisfaction. An interesting observation is that we can discriminate 3 users that were really happy with Hecataeus compared to their selection tool, another 3 that were as happy as with their tools and another group of 3 that were happier or slightly happier with Hecataeus, compared to the tool of their selection.

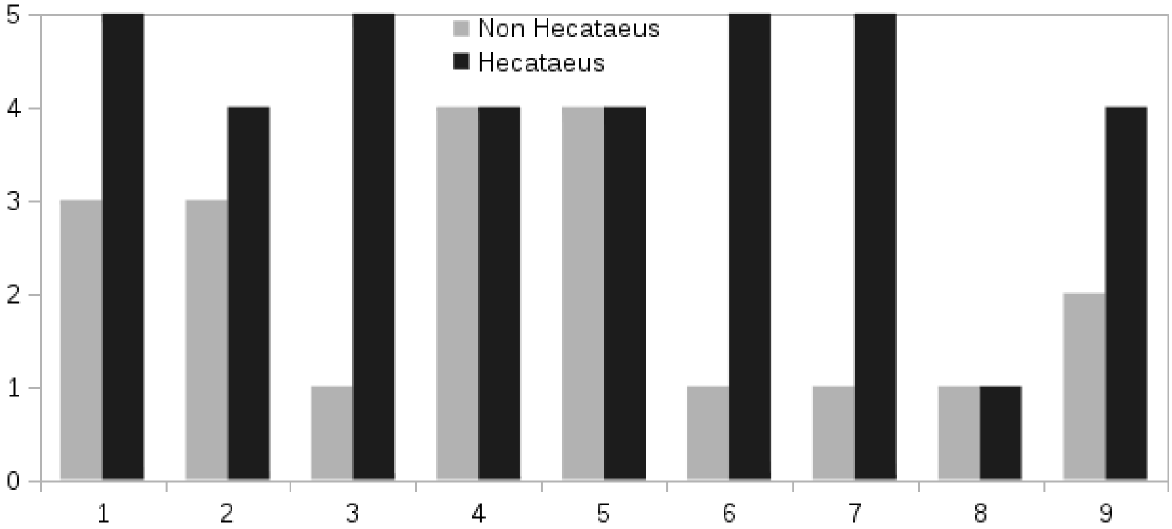


Figure 6.14: User satisfaction in 0 to 5 scale on how helpful was Hecataeus and the tool of their choice to perform complex changes in the files that use a specific table.

Regarding the time measurement of Task 2, we depict in Figure 6.15 that Hecataeus completes the task faster than any other tool or technique used, except for one situation (maintainer number 4, who also stated that the tool of his selection and Hecataeus were equally helpful for performing the second task).

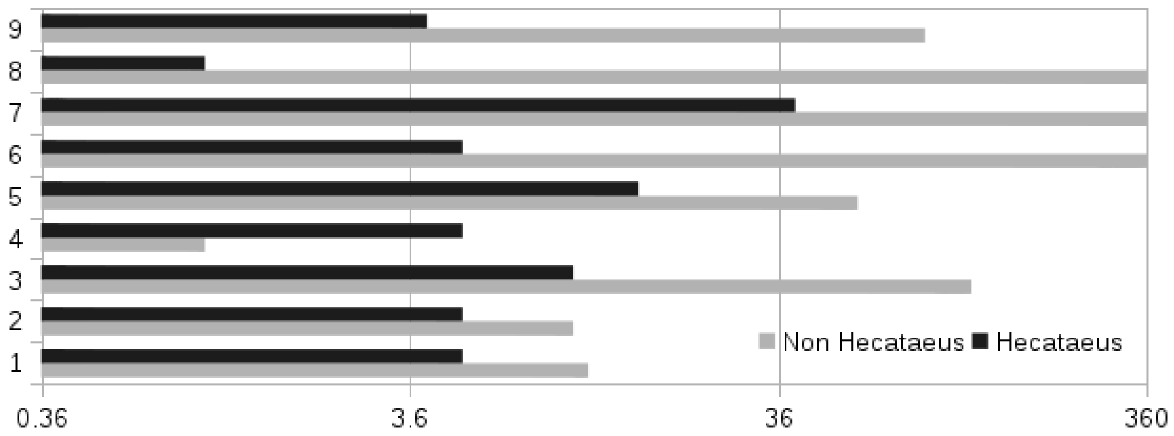


Figure 6.15: Time needed (in minutes) for Task 2, which is to change a number of files but leave one unmodified, due to a database schema alteration.

6.5.3 Code understanding

Regarding the next tasks, we wanted to measure the code understanding that our tool provides to its users. In Task 3, the participants were asked to evaluate how easy was to use our tool to locate where the queries should change, given an event. In Task 4, the participants had to compare the three visualization methods (clusters on a circle, clusters on concentric circles, and clusters on concentric arcs) on their appropriateness to identify possible database evolution steps. The users were requested to perform Task 3 and 4 both with Hecataeus and their tool of choice. Both tasks' unit of measurement is the user satisfaction expressed as a number between 0 and 5.

As depicted in Fig. 6.16, we observe that all participants, except user 8, found our tool was really helpful. Apart from participant with number 8 who gave small value (1) to both Hecataeus and the tool of his selection, Hecataeus had 4 perfect scores and another 4 nearly perfect scores. The average number of the user satisfaction with Hecataeus for Task 3 is 4.1 units, compared to 1.7 units of the rest of the tools.

Regarding the score of each one of the visualization methods used for code understanding, we observed that the concentric methods (both circles and arcs) had better values compared to single circle. The clear winner of the three Hecataeus methods is the concentric arcs, with average value 4.2 units of user satisfaction; second comes the concentric circles, with average value 4.1 units of user satisfaction, and, the single circle is the last one with an average value of 3.7 units of user satisfaction. When the users evaluated the tools of their choice, no one was happy with the results of their tool (the average value was only 1.6 units of user satisfaction). Figure 6.17 depicts

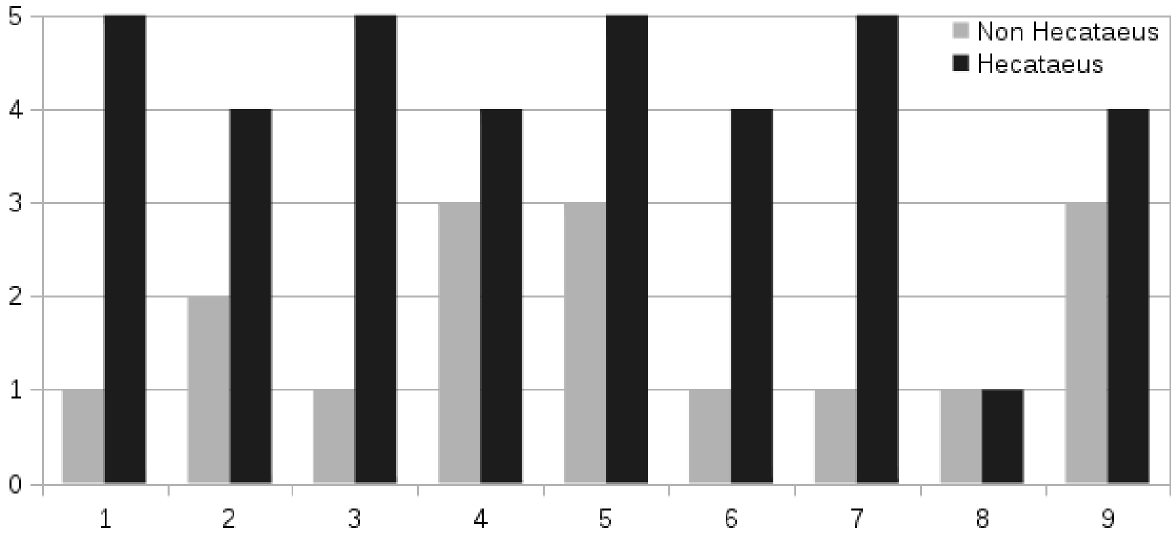


Figure 6.16: Task 3 measurements. The users evaluated on how useful the visualization technique is, when they want to identify specific parts of the code that change (impact analysis).

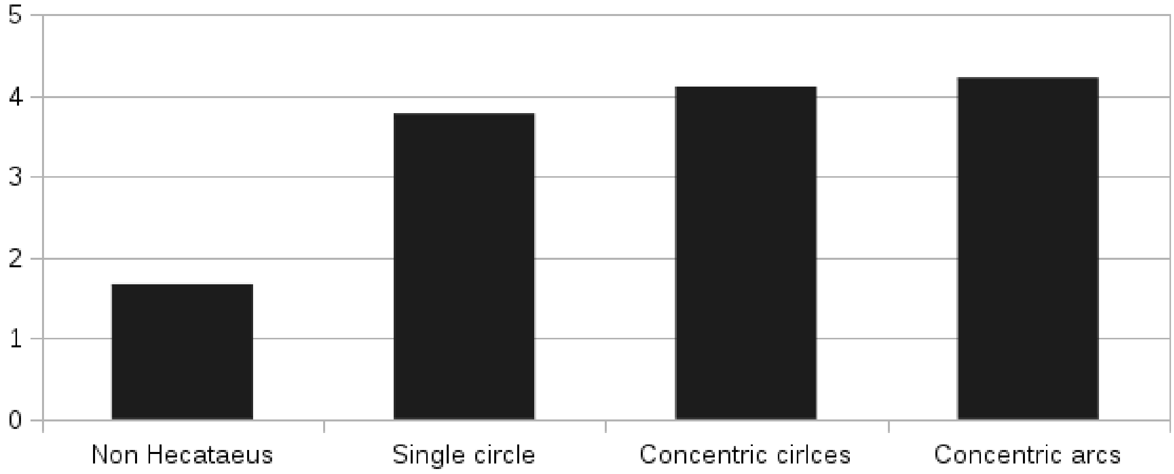


Figure 6.17: Task 4 measurements. The concentric methods are more useful on code understanding, regarding the evolution of a database related project.

the average results of Task 4.

6.5.4 Threats to validity

We have addressed the threat of maturation by employing inverse order of tool usage between the two groups of participants.

Additionally, we informed the participants that this evaluation is anonymous and we will not know who answered what, plus we *emphasized* that we only wanted the truth out of this evaluation, thus the reactive arrangements and the reactivity threats are also excluded from our user study.

In order to guarantee the integrity of the presented results we had to resort in the exclusion of participants whose actions or reports were incoherent. So, although we report on 9 participants, the test was originally conducted with 12 participants, 3 of which were excluded.

1. One of the excluded participants gave the exact same results in Hecataeus answer as he had given using the tool of his selection. He had correctly found 5 out of the 7 files, but the additional two he mentioned in his answer could not be related to the requirements of Task 1. So he was either unwilling to use the tool, or he had mistakenly used it. The fact that the answer is exactly the same as in his own tool, made us exclude him from our study.

2. Two of the participants also gave the exact same answers in Hecataeus and their selection tools. The fact here is that in the project that was given to them, in order to work with Hecataeus, we had an auxiliary file, intentionally injected, that had a connection to the COMMENT table. That file does not exist in the project that was given to be examined by the tools of their choice. Therefore, we excluded those participants from the survey too.

6.6 Conclusions

Concluding, in this Chapter we have explored how visual maps of data-intensive information systems support the work of their developers.

We had already seen the three possible circular layouts of a visual map of the information system in [1] and in this Chapter we proposed an impact analysis visualization method, which we evaluated additionally to the three aforementioned visualization methods with a user study. As we have seen in the user study our tool and its methods are helping the users to have a better code understanding of a new project, with less effort and time consumed.

Chapter 7

Conclusions and Future Work

In the current chapter, we summarize our findings and our major contributions and describe directions for future work. Section 7.1 summarizes the contributions of this thesis, and ideas for future work are discussed in Section 7.2.

7.1 Conclusions

In this research we have proposed the map that connects the software part of the data-intensive ecosystems to the database part, which is the *Architecture Graph*. To create this graph, we have to know the database schema of the data-intensive ecosystems and the queries that use it. To obtain the queries, we have to extract them from the source code files of the projects we examined. Then, having the *Architecture Graph*, we have proposed a metric that are related to both data and software, which metric helps us evaluate the quality, understandability, and maintainability of the project we were interested in. Additionally, we have suggested restructures in both parts (software and database) of the project so as to achieve higher software and database schema quality. Then, we have addressed the problem of "what-if" analysis for the evolution of the schema of a data-intensive ecosystem, where we use policies over the potential events that regulate the evolution of the *Architecture Graph*. Finally, we have proposed a method to depict the *Architecture Graph*'s "what-if" analysis in a way that reduces visual clutter, and evaluated the visualization methods presented in [1] with a user study.

In Chapter 3, we have proposed a 4 step method for the query extraction problem that is both host language and programming style independent. The first step is to keep only the code that is related to the database. The second step is to create every possible query version that might occur due to the branch and loop statements of the host language, using the Query Variants Graph representation, which is both host language independent, and programming style independent. *The third step is to create an abstract representation for each query, using the Abstract Query Representation that is query language independent*. Finally, the fourth step is based on this abstract representation which we can "export" in a concrete query language, that can be used for migrating a project from one querying syntax to another (e.g. migrate from SQL Server to Oracle[1], or even MongoDB).

In Chapter 4, we have proposed a metric that assesses the coupling of the application to the database of a project. We have introduced in a principle manner, the fundamental ideas, properties and constraints that evaluate a well-designed data-intensive ecosystem project. We have evaluated our metric using a real world ecosystem, following its evolution, and we have observed that our metric is in sync with the Lehman's Lows of evolution. Moreover, we have proposed an algorithm that rewrites parts of the schema and software to obtain better metric values, which additionally simplifies the developers effort during evolution events.

In Chapter 5, we have introduced the *Architecture Graph* that models the data-intensive ecosystems with input and output schemata, the events that represent imminent schema changes and finally the policies that regulate whether the event is accepted (and propagated to a module's consumers) or not. To achieve that, we have introduced a language for policies that is complete over the events we examined and concise enough to only need a few policy lines to describe the general ecosystem's policy and specific component's policies that differ. Additionally we rewrite the software and database code, performing the event's schema changes when all nodes have accepted the event. When there are conflicts on an event acceptance over a database view change, we proposed a solution that creates two variants of the view: (a) one

[1] `https://www.w3schools.com/sql/sql_top.asp`

that performs the needed rewrites for those that accepted the change, and (b) another that retains the original definition of the view. Then, we rewrite the view's consumers to follow the view definition they want. Our contributions are:

- The impact assessment where a status determination algorithm (Algorithm 5.2) makes sure that the nodes of the ecosystem are assigned a status concerning when they are affected by an event depending on their reaction to the event (accepted or blocked).
- The conflict resolution and variants calculation of Algorithm 5.3, where queries with different reactions to an event of a view can get satisfied using a number of view variants (one for the blockers and another for the accepters).
- The module rewriting of Algorithm 5.4 where, since the status and the number of variants have been determined, we rewrote the software and the database schema to restructure the nodes and edges of the *Architecture Graph*.

In Chapter 6, we have addressed the problem of visually depicting the *Architecture Graph*. In this Thesis we have extended the work originally presented in [1] with a "what-if" visualization algorithm and a user study. The visualization of the "what-if" scenarios is performed by, depicting only the affected nodes, in parallel lines minimizing the visual clutter of the edge crossings. From the user study we conclude that:

- The visualization is a neat tool to help in code understanding of a data intensive project that it is new for the developers.
- The time needed to perform simple tasks is better when there is a tool to help besides performing it, also displaying it, since the users can easily evaluate the success or failure of the task.

7.2 Future work

In this section we discuss the ideas for additional research over the open issues that this work has not cover.

In Chapter 3, we talked about the query extraction problem, and we presented the Query Variants Graph that describe how a query is constructed due to branch and loop statements of the hosting language, and with the Abstract Query Representation we are able to describe every query in a more abstract way that we can then translate to two up to now concrete query languages. Some of the open issues in this area are: (i) an extension to the Abstract Data Manipulation Operators in order to cover even more query language components, (ii) writing the needed code to facilitate the use of our tool in more host languages besides C++ and PHP, (iii) a way to help the users of our software avoid writing any line of code but work with pattern recognition when they search for query functions that interact with a query object, or query string, and, finally, (iv) methods to translate the Abstract Query Representation to more than the two presented (SQL and MongoDB) concrete query languages.

In Chapter 4, we introduced the fundamental requirements to evaluate a well-designed Data Intensive Information Systems project. To this end, we have proposed the Data-To-Software Coupling metric based on those requirements, which assesses, in a principled and multigranular way, how strongly coupled a software part of a project is to the underlying data. We have also showed that the evolution of a popular project is in line with the quality metric proposed. Moreover, we have described an algorithm on how we can rewrite parts of the system, based on our metric, to improve the understandability and reduce maintenance costs. Concerning future work, a table utilization metric would be useful, so as to know whether a query uses all the attributes of a table or not, which would suggest an extension to the presented rewriting method with an additional algorithm that would “clean up” the output variables of the views that are not used by any query. Additionally, apart from coupling, the exploration of metrics that are common to software and absent from databases, such as cohesion or complexity would also be an interesting research path to follow.

In Chapter 5, we initially described the Architecture Graph which depicts a data-intensive ecosystem, and then we described that using policies an ecosystem can withstand a change gracefully using policies that denote that an imminent is accepted or not. Then, we proposed a set of algorithms that could facilitate schema changes even when there existed contradicting policies for a change. The future work of this chapter can continue in several directions. For example, the change events can address the assessment of complicated events, involving a set of possible changes simultaneously applied over either the same or different modules. This would also involve some extra ”garbage collection” of views that are redundant or useless. The possibility of adding more semantics to the Architecture Graph is also a possible path for future research. For example, constraints that are not necessarily extracted from the reverse engineering of the database, like functional or conditional functional dependencies, or logical constraints within the source code (e.g., pre- and post-conditions over the correctness of a stored procedure) can also become part of the graph. Adding more kinds of sources, like for example, web-services, or XML stores to the graph is also a possibility. Providing hints to the DBA's or the developers for policies in a semi-automatic way can also help with the annotation of the graph.

Finally, concerning the visualization techniques presented in Chapter 6, the issues that remain to be explored, include alternative visualization methods and improved space utilization of the 2D canvas.

Bibliography

[1] P. Manousis, P. Vassiliadis, and G. Papastefanatos, "Impact analysis and policy-conforming rewriting of evolving data-intensive ecosystems," *Journal on Data Semantics*, vol. 4, no. 4, pp. 231–267, 2015.

[2] E. Kontogiannopoulou, P. Manousis, and P. Vassiliadis, "Visual maps for data-intensive ecosystems," in *Conceptual Modeling - 33rd International Conference, ER 2014, Atlanta, GA, USA, October 27-29, 2014. Proceedings* (E. S. K. Yu, G. Dobbie, M. Jarke, and S. Purao, eds.), vol. 8824 of *Lecture Notes in Computer Science*, pp. 385–392, Springer, 2014.

[3] J. F. Roddick, "A survey of schema versioning issues for database systems," *Information & Software Technology*, vol. 37, no. 7, pp. 383–393, 1995.

[4] M. Hartung, J. F. Terwilliger, and E. Rahm, "Recent Advances in Schema and Ontology Evolution," in *Schema Matching and Mapping* (Z. Bellahsene, A. Bonifati, and E. Rahm, eds.), Data-Centric Systems and Applications, pp. 149–190, Springer, 2011.

[5] D. Sjøberg, "Quantifying Schema Evolution," *Information and Software Technology*, vol. 35, no. 1, pp. 35–44, 1993.

[6] C. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, "Schema Evolution in Wikipedia: Toward a Web Information System Benchmark," in *Proceedings of 10th International Conference on Enterprise Information Systems (ICEIS)*, 2008.

[7] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful Database Schema Evolution: the PRISM Workbench," *Proceedings of the VLDB Endowment*, vol. 1, pp. 761–772, 2008.

[8] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Automating the Database Schema Evolution Process," *VLDB Journal*, vol. 22, no. 1, pp. 73–98, 2013.

[9] D.-Y. Lin and I. 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)*, pp. 31–40, 2009.

[10] S. Wu and I. Neamtiu, "Schema evolution analysis for embedded databases," in *Proceedings of the 27th IEEE International Conference on Data Engineering Workshops (ICDEW)*, pp. 151–156, 2011.

[11] D. Qiu, B. Li, and Z. 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)*, pp. 125–135, 2013.

[12] I. Skoulis, P. Vassiliadis, and A. V. Zarras, "Open-source databases: Within, outside, or beyond lehman's laws of software evolution?," in *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings* (M. Jarke, J. Mylopoulos, C. Quix, C. Rolland, Y. Manolopoulos, H. Mouratidis, and J. Horkoff, eds.), vol. 8484 of *Lecture Notes in Computer Science*, pp. 379–393, Springer, 2014.

[13] I. Skoulis, P. Vassiliadis, and A. Zarras, "Growing Up with Stability: how Open-Source Relational Databases Evolve," *Information Systems*, vol. in press, 2015.

[14] P. Vassiliadis, A. Zarras, and I. Skoulis, "How is Life for a Table in an Evolving Relational Schema? Birth, Death and Everything in Between," in *Proceedings of the 34th International Conference on Conceptual Modeling (ER)*, p. to appear, 2015.

[15] M. M. Lehman and J. C. Fernandez-Ramil, *Software Evolution and Feedback: Theory and Practice*, ch. Rules and Tools for Software Evolution Planning and Management. Wiley, 2006.

[16] L. A. Belady and M. M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[17] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona, "The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review," *ACM Computing Surveys*, vol. 46, no. 2, pp. 1–28, 2013.

[18] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - the nineties view," in *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA*, p. 20, IEEE Computer Society, 1997.

[19] Oracle, "Oracle Change Management Pack." http://docs.oracle.com/html/A96679_01/overview.htm, 2014.

[20] IBM, "Schema changes." http://pic.dhe.ibm.com/infocenter/db2luw/v10r1/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.dbobj.doc%2Fdoc%2Fc0060234.html, May 2014.

[21] "IBM DB2 object comparison tool for Z/OS version 10 release 1." http://www-01.ibm.com/support/knowledgecenter/SSAUVH_10.1.0/com.ibm.db2tools.gou10.doc.ug/gocugj13.pdf?lang=en, May 2012.

[22] "SQL management studio for SQL server user's manual." http://www.sqlmanager.net/download/msstudio/doc/msstudio.pdf, December 2012.

[23] "Microsoft SQL server data tools: Database development zero to sixty." http://channel9.msdn.com/Events/TechEd/Europe/2012/DBI311, June 2012.

[24] "Django." https://www.djangoproject.com/.

[25] "South." http://south.readthedocs.org/en/latest/index.html.

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

[27] "Hecataeus." http://cs.uoi.gr/~pvassil/projects/hecataeus/index.html.

[28] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008* (W. Schäfer, M. B. Dwyer, and V. Gruhn, eds.), pp. 451–460, ACM, 2008.

[29] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou, "Policy-regulated management of ETL evolution," *J. Data Semantics*, vol. 13, pp. 147–177, 2009.

[30] G. Papastefanatos, P. Vassiliadis, A. Simitsis, K. Aggistalis, F. Pechlivani, and Y. Vassiliou, "Language Extensions for the Automation of Database Schema Evolution," in *ICEIS (1)* (J. Cordeiro and J. Filipe, eds.), pp. 74–81, 2008.

[31] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou, "Design metrics for data warehouse evolution," in *Conceptual Modeling - ER 2008, 27th International Conference on Conceptual Modeling, Barcelona, Spain, October 20-24, 2008. Proceedings* (Q. Li, S. Spaccapietra, E. S. K. Yu, and A. Olivé, eds.), vol. 5231 of *Lecture Notes in Computer Science*, pp. 440–454, Springer, 2008.

[32] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou, "HECATAEUS: regulating schema evolution," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA* (F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, eds.), pp. 1181–1184, IEEE Computer Society, 2010.

[33] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou, "Metrics for the prediction of evolution impact in ETL ecosystems: A case study," *J. Data Semantics*, vol. 1, no. 2, pp. 75–97, 2012.

[34] P. Manousis, P. Vassiliadis, and G. Papastefanatos, "Automating the adaptation of evolving data-intensive ecosystems," in *Proceedings of the 32nd International Conference on Conceptual Modeling (ER)*, pp. 182–196, 2013.

[35] C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++," *PVLDB*, vol. 4, no. 2, pp. 117–128, 2010.

[36] M. Mohania, "Avoiding re-computation: View adaptation in data warehouses," in *In Proc. of 8 th International Database Workshop, Hong Kong*, pp. 151–165, 1997.

[37] A. Gupta, I. S. Mumick, J. Rao, and K. A. Ross, "Adapting materialized views after redefinitions: techniques and a performance study," *Information Systems*, vol. 26, no. 5, pp. 323–362, 2001.

[38] A. Nica, A. J. Lee, and E. A. Rundensteiner, "The CVS algorithm for view synchronization in evolvable large-scale information systems," in *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings* (H. Schek,

F. Saltor, I. Ramos, and G. Alonso, eds.), vol. 1377 of *Lecture Notes in Computer Science*, pp. 359–373, Springer, 1998.

[39] M. Golfarelli and S. Rizzi, "A survey on temporal data warehousing," *IJDWM*, vol. 5, no. 1, pp. 1–17, 2009.

[40] R. Wrembel, "A survey of managing the evolution of data warehouses," *IJDWM*, vol. 5, no. 2, pp. 24–56, 2009.

[41] Z. Bellahsene, "View adaptation in data warehousing systems," in *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings* (G. Quirchmayr, E. Schweighofer, and T. J. M. Bench-Capon, eds.), vol. 1460 of *Lecture Notes in Computer Science*, pp. 300–309, Springer, 1998.

[42] Z. Bellahsene, "Schema evolution in data warehouses," *Knowl. Inf. Syst.*, vol. 4, no. 3, pp. 283–304, 2002.

[43] C. Quix, "Repository support for data warehouse evolution," in *DMDW* (S. Gatziu, M. A. Jeusfeld, M. Staudt, and Y. Vassiliou, eds.), vol. 19 of *CEUR Workshop Proceedings*, p. 4, CEUR-WS.org, 1999.

[44] M. Blaschka, C. Sapia, and G. Höfling, "On schema evolution in multidimensional databases," in *DaWaK* (M. K. Mohania and A. M. Tjoa, eds.), vol. 1676 of *Lecture Notes in Computer Science*, pp. 153–164, Springer, 1999.

[45] C. A. Hurtado, A. O. Mendelzon, and A. A. Vaisman, "Maintaining data cubes under dimension updates," in *ICDE* (M. Kitsuregawa, M. P. Papazoglou, and C. Pu, eds.), pp. 346–355, IEEE Computer Society, 1999.

[46] C. A. Hurtado, A. O. Mendelzon, and A. A. Vaisman, "Updating olap dimensions," in *DOLAP* (I.-Y. Song and T. J. Teorey, eds.), pp. 60–66, ACM, 1999.

[47] C. Kaas, T. B. Pedersen, and B. Rasmussen, "Schema evolution for stars and snowflakes," in *ICEIS (1)*, pp. 425–433, 2004.

[48] R. Wrembel and B. Bebel, "Metadata management in a multiversion data warehouse," *J. Data Semantics*, vol. 8, pp. 118–157, 2007.

[49] J. Eder, C. Koncilia, and D. Mitsche, "Automatic detection of structural changes in data warehouses," in *DaWaK* (Y. Kambayashi, M. K. Mohania, and W. Wöß, eds.), vol. 2737 of *Lecture Notes in Computer Science*, pp. 119–128, Springer, 2003.

[50] J. Eder, C. Koncilia, and D. Mitsche, "Analysing slices of data warehouses to detect structural modifications," in *CAiSE* (A. Persson and J. Stirna, eds.), vol. 3084 of *Lecture Notes in Computer Science*, pp. 492–505, Springer, 2004.

[51] J. Eder and C. Koncilia, "Changes of dimension data in temporal data warehouses," in *DaWaK* (Y. Kambayashi, W. Winiwarter, and M. Arikawa, eds.), vol. 2114 of *Lecture Notes in Computer Science*, pp. 284–293, Springer, 2001.

[52] M. Golfarelli, J. Lechtenbörger, S. Rizzi, and G. Vossen, "Schema versioning in data warehouses: Enabling cross-version querying via schema augmentation," *Data Knowl. Eng.*, vol. 59, no. 2, pp. 435–459, 2006.

[53] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings* (R. Cousot, ed.), vol. 2694 of *Lecture Notes in Computer Science*, pp. 1–18, Springer, 2003.

[54] C. Gould, Z. Su, and P. T. Devanbu, "Static checking of dynamically generated queries in database applications," in *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom* (A. Finkelstein, J. Estublier, and D. S. Rosenblum, eds.), pp. 645–654, IEEE Computer Society, 2004.

[55] G. Wassermann, C. Gould, Z. Su, and P. T. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, p. 14, 2007.

[56] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene, "An interactive tool for analyzing embedded SQL queries," in *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings* (K. Ueda, ed.), vol. 6461 of *Lecture Notes in Computer Science*, pp. 131–138, Springer, 2010.

[57] H. van den Brink, R. van der Leek, and J. Visser, “Quality assessment for embedded SQL,” in *Seventh IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2007), September 30 - October 1, 2007, Paris, France*, pp. 163–170, IEEE Computer Society, 2007.

[58] M. N. Ngo and H. B. K. Tan, “Applying static analysis for automated extraction of database interactions in web applications,” *Information & Software Technology*, vol. 50, no. 3, pp. 160–175, 2008.

[59] C. Nagy, L. Meurice, and A. Cleve, “Where was this SQL query executed? a static concept location approach,” in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015* (Y. Guéhéneuc, B. Adams, and A. Serebrenik, eds.), pp. 580–584, IEEE Computer Society, 2015.

[60] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[61] V. R. Basili, L. C. Briand, and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Trans. Software Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[62] L. C. Briand, J. W. Daly, and J. Wüst, “A unified framework for cohesion measurement in object-oriented systems,” *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

[63] J. Al-Dallal and L. C. Briand, “A precise method-method interaction-based cohesion metric for object-oriented classes,” *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 2, pp. 8:1–8:34, 2012.

[64] U. S. Poornima and V. Suma, “Significance of coupling and cohesion on design quality,” *CoRR*, vol. abs/1402.2375, 2014.

[65] L. C. I. David Longstreet, *Function Points Analysis Training Course*. www.SoftwareMetrics.com, 2004.

[66] M. P. Papazoglou and W. van den Heuvel, “Service-oriented design and development methodology,” *Int. J. Web Eng. Technol.*, vol. 2, no. 4, pp. 412–442, 2006.

[67] C. Legner and T. Vogel, “Design principles for B2B services - an evaluation of two alternative service designs,” in *2007 IEEE International Conference on Services Computing (SCC 2007), 9-13 July 2007, Salt Lake City, Utah, USA*, pp. 372–379, IEEE Computer Society, 2007.

[68] D. Athanasopoulos and A. V. Zarras, “Fine-grained metrics of cohesion lack for service interfaces,” in *IEEE International Conference on Web Services, ICWS 2011, Washington, DC, USA, July 4-9, 2011*, pp. 588–595, IEEE Computer Society, 2011.

[69] M. Perepletchikov, C. Ryan, and Z. Tari, “The impact of service cohesion on the analyzability of service-oriented software,” *IEEE Trans. Services Computing*, vol. 3, no. 2, pp. 89–103, 2010.

[70] A. Kazemi, A. Rostampour, A. Zamiri, P. Jamshidi, H. Haghighi, and F. Shams, “An information retrieval based approach for measuring service conceptual cohesion,” in *Proceedings of the 11th International Conference on Quality Software, QSIC 2011, Madrid, Spain, July 13-14, 2011.* (M. Núñez, R. M. Hierons, and M. G. Merayo, eds.), pp. 102–111, IEEE Computer Society, 2011.

[71] T. Sharma, M. Fragkoulis, S. Rizou, M. Bruntink, and D. Spinellis, “Smelly relations: measuring and understanding database schema quality,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (F. Paulisch and J. Bosch, eds.), pp. 55–64, ACM, 2018.

[72] R. Pottinger and A. Y. Halevy, “Minicon: A scalable algorithm for answering queries using views,” *VLDB J.*, vol. 10, no. 2-3, pp. 182–198, 2001.

[73] A. Y. Levy, A. Rajaraman, and J. J. Ordille, “Querying heterogeneous information sources using source descriptions,” in *VLDB’96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India* (T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds.), pp. 251–262, Morgan Kaufmann, 1996.

[74] O. M. Duschka and M. R. Genesereth, “Answering recursive queries using views,” in *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Sympo-*

sium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona, USA (A. O. Mendelzon and Z. M. Özsoyoglu, eds.), pp. 109–116, ACM Press, 1997.

[75] R. DeLine, G. Venolia, and K. Rowan, "Software development with code maps," *ACM Queue*, vol. 8, no. 7, p. 10, 2010.

[76] B. Johnson and B. Shneiderman, "Tree maps: A space-filling approach to the visualization of hierarchical information structures," in *IEEE Visualization*, pp. 284–291, 1991.

[77] S. G. Eick, J. L. Steffen, and E. E. S. Jr., "Seesoft-a tool for visualizing line oriented software statistics," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 957–968, 1992.

[78] A. Bragdon, S. P. Reiss, R. C. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010* (J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, eds.), pp. 455–464, ACM, 2010.

[79] P. Caserta and O. Zendra, "Visualization of the static aspects of software: A survey," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 7, pp. 913–933, 2011.

[80] C. Ware, *Information visualization: Perception for design.* Morgan Kaufmann, 2nd edition, 2004.

[81] J. Tidwell, *Designing interfaces - patterns for effective interaction design.* O'Reilly, 2006.

[82] T. Munzner, "A nested process model for visualization design and validation," *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 6, pp. 921–928, 2009.

[83] J. M. Six and I. G. Tollis, "A framework and algorithms for circular drawings of graphs," *J. Discrete Algorithms*, vol. 4, no. 1, pp. 25–50, 2006.

[84] I. Halupczok and A. Schulz, "Pinning balloons with perfect angles and optimal area," *J. Graph Algorithms Appl.*, vol. 16, no. 4, pp. 847–870, 2012.

[85] P. Hoffman, G. G. Grinstein, and D. Pinkney, "Dimensional anchors: A graphic primitive for multidimensional multivariate information visualizations," in *Workshop on New Paradigms in Information Visualization and Manipulation (NPIVM '99), in conjunction with the Eigth ACM International Conference on Information and Knowledge Management (CIKM '99), Kansas City, Missouri, USA, November 6, 1999, Proceedings.*, pp. 9–16, ACM, 1999.

[86] K. Misue, "Drawing bipartite graphs as anchored maps," in *Asia-Pacific Symposium on Information Visualisation, APVIS 2006, Tokyo, Japan, February 1-3, 2006* (K. Misue, K. Sugiyama, and J. Tanaka, eds.), vol. 60 of *CRPIT*, pp. 169–177, Australian Computer Society, 2006.

[87] G. M. Draper, Y. Livnat, and R. F. Riesenfeld, "A survey of radial methods for information visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 15, no. 5, pp. 759–776, 2009.

[88] S. J. Rysavy, D. Bromley, and V. Daggett, "DIVE: A graph-based visual-analytics framework for big data," *IEEE Computer Graphics and Applications*, vol. 34, no. 2, pp. 26–37, 2014.

[89] S. van den Elzen and J. J. van Wijk, "Multivariate network exploration and presentation: From detail to overview via selections and aggregations," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2310–2319, 2014.

[90] B. Bach, E. Pietriga, and J. Fekete, "Graphdiaries: Animated transitions andtemporal navigation for dynamic networks," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 5, pp. 740–754, 2014.

[91] K. Gallagher and D. Binkley, "Program slicing," in *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pp. 58–67, IEEE, 2008.

[92] A. Cleve, J. Henrard, and J. Hainaut, "Data reverse engineering using system dependency graphs," in *Proceedigns of the 13th Working Conference on Reverse Engineering (WCRE)*, pp. 157–166, 2006.

[93] M. Goeminne, A. Decan, and T. Mens, "Co-evolving code-related and database-related changes in a data-intensive software system," in *2014 Software Evolution*

Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014 (S. Demeyer, D. W. Binkley, and F. Ricca, eds.), pp. 353–357, IEEE Computer Society, 2014.

[94] T. Mens, L. Meurice, M. Goeminne, C. Nagy, A. Decan, and A. Cleve, *Analyzing the Evolution of Database Usage in Data-Intensive Software Systems*. 01 2017.

[95] R. Pressman, *Software Engineering: A Practitioner's Approach: European Adaption*. McGraw-Hill, 5 ed., April 2000.

[96] Y. Velegrakis, R. J. Miller, and L. Popa, "Preserving mapping consistency under schema changes," *VLDB J.*, vol. 13, no. 3, pp. 274–293, 2004.

[97] G. Papastefanatos, P. Vassiliadis, and A. Simitsis, "Propagating evolution events in data-centric software artifacts," in *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany* (S. Abiteboul, K. Böhm, C. Koch, and K. Tan, eds.), pp. 162–167, IEEE Computer Society, 2011.

[98] P. Manousis, "Database evolution and maintenance of their dependent applications via query rewriting," Master's thesis, Department of Computer Science, University of Ioannina, February 2013.

[99] Transaction Processing Performance Council, "The New Decision Support Benchmark Standard," April 2012. `http://www.tpc.org/tpcds/default.asp`.

[100] M. H. Dunham, *Data Mining: Introductory and Advanced Topics*. Prentice-Hall, 2002.

[101] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[102] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Softw., Pract. Exper.*, vol. 21, no. 11, pp. 1129–1164, 1991.

[103] B. Meyer, "Self-organizing graphs - A neural network perspective of graph layout," in *Graph Drawing, 6th International Symposium, GD'98, Montréal, Canada, August 1998, Proceedings* (S. Whitesides, ed.), vol. 1547 of *Lecture Notes in Computer Science*, pp. 246–262, Springer, 1998.

[104] T. Kamada and S. Kawai, “An algorithm for drawing general undirected graphs,” *Inf. Process. Lett.*, vol. 31, no. 1, pp. 7–15, 1989.

Author's Publications

1. Petros Manousis, Apostolos V. Zarras, Panos Vassiliadis, George Papastefanatos: Extraction of Embedded Queries via Static Analysis of Host Code. CAiSE 2017: 511-526

2. Dimitrios Gkesoulis, Panos Vassiliadis, Petros Manousis: CineCubes: Aiding data workers gain insights from OLAP queries. Inf. Syst. 53: 60-86 (2015)

3. Petros Manousis, Panos Vassiliadis, George Papastefanatos: Impact Analysis and Policy-Conforming Rewriting of Evolving Data-Intensive Ecosystems. J. Data Semantics 4(4): 231-267 (2015)

4. Petros Manousis, Panos Vassiliadis, Apostolos V. Zarras, George Papastefanatos: Schema Evolution for Databases and Data Warehouses. eBISS 2015: 1-31

5. Efthymia Kontogiannopoulou, Petros Manousis, Panos Vassiliadis: Visual Maps for Data-Intensive Ecosystems. ER 2014: 385-392

6. Petros Manousis, Panos Vassiliadis, George Papastefanatos: Automating the Adaptation of Evolving Data-Intensive Ecosystems. ER 2013: 182-196

Short Biography

My name is Petros Manousis and I am a PhD candidate at the Computer Science & Engineering Department of the University of Ioannina in Greece, under the supervision of Panos Vassiliadis. I received my MSc and BSc Degrees from the same institution in 2013 and 2008 respectively. I have been a member of the Distributed Management of Data Laboratory since 2011. My academic interests lie in the area of software engineering and data management with a particular emphasis on database ecosystem evolution.