DAHLIA 2.0: A Visual Analyzer of Database Usage in Dynamic and Heterogeneous Systems

Loup Meurice and Anthony Cleve
PReCISE Research Center, University of Namur, Belgium
{loup.meurice,anthony.cleve}@unamur.be

***Abstract*—Understanding the links between application programs and their database is useful in various contexts such as migrating information systems towards a new database platform, evolving the database schema, or assessing the overall system quality. However, data-intensive applications nowadays tend to access their underlying database in an increasingly dynamic way. The queries that they send to the database server are usually built at runtime, through String concatenation, or Object-Relational-Mapping (ORM) frameworks. This level of dynamicity significantly complicates the task of adapting programs to an evolving database schema. In this paper, we present DAHLIA 2.0, an interactive visualization tool that allows developers to analyze the database usage in order to support data-intensive software evolution and more precisely, program-database co-evolution.**

I. Introduction

Maintaining and evolving large software systems is becoming increasingly complex in the case of *data-intensive* software systems (DISS). These systems manipulate a huge amount of data usually stored in a relational database, by means of possibly complex and dynamic interactions between the application programs and the database. When the database schema evolves, developers often need to adapt the source code of the applications that accesses the changed schema elements. This adaptation process is usually achieved manually. Furthermore, nowadays, a large variety of frameworks and libraries can be used to access the database. In particular, Object-relational mapping (ORM) technologies provide a high level of abstraction upon a relational database that allows developers to use the programming language they are comfortable with instead of using SQL statements and stored procedures. As a consequence, the interactions between the program source code and the database may become more dynamic, and thus more complex to understand.

For instance, Hibernate and JPA (Java Persistence API) offer to Java developers a way for mapping an object-oriented domain model to a traditional relational database. Their primary feature is to map Java classes to database tables (and Java data types to SQL data types). Hibernate and JPA provide both an SQL inspired language called, respectively, Hibernate Query Language (HQL) and Java Persistence Query Language (JPQL) which allows to write SQL like queries using the mappings defined before. In addition, both also provide a way to perform CRUD operations (Create, Read, Update, and Delete) on the instances of the mapped entity classes. Figure 1 shows an example of JPA code accessing a database. Those mechanisms partially/fully hide the actual executed SQL query by making the access more abstract.

(1) Sample JPQL query. Customer selection according to a given id

```
1 EntityManagerFactory emf = ...;
2 EntityManager em = emf.createEntityManager();
3 Order order = ...;
4 Integer cust_id = order.getCustomerId();
5 Customer cust = (Customer)em.createQuery("SELECT c FROM Customer c
      WHERE c.cust_id=:cust_id")
6     .setParameter("cust_id", cust_id).getSingleResult();
```

(2) JPA operation on a mapped entity class instance. Creation and insertion of a new order

```
1 EntityManager entityManager = entityManagerFactory.createEntityManager
      ();
2 entityManager.getTransaction().begin();
3 Order order= createNewOrder();
4 entityManager.persist( order );
5 entityManager.getTransaction().commit();
6 entityManager.close();
```

Fig. 1. Samples of JPA accesses.

In this context, manually recovering the links between the source code and the database schema and understanding it may prove complicated due to higher levels of abstraction and dynamicity. This paper addresses this particular problem. It presents DAHLIA 2.0, a visualization tool allowing developers to analyze the database usage in highly dynamic and heterogeneous systems. Our tool extracts and visualizes the database accesses executed in the source code in order to derive useful information about the database usage. DAHLIA 2.0 is a support to software comprehension and to database-program co-evolution.

The remainder of this paper is structured as follows. Section II discusses the related work. Section III presents DAHLIA 2.0 and its main features. In Section IV, we conclude the paper and anticipate future directions.

II. Related Work

While the database schema evolution literature is very large [15], researchers have only recently started to pay more attention to the analysis of of database schema and application code co-evolution. Qiu *et al.* [14] empirically analyzed the co-evolution of relational database schemas and code in ten open-source database applications from various domains. They studied specific change types inside the database schema and *estimated* the impact of such changes on PHP code. Karahasanoić [5] studied how the maintenance of application consistency can be supported by identifying and visualizing the impact of changes in evolving object-oriented systems, including changes originating from a database schema. However,

[0]The first author is supported by the F.R.S.-FNRS via the DISSE project.

he focused on object-oriented databases rather than relational databases.

Using *what-if analysis* [2] for changes that occur in the schema/structure of the database was proposed by Papastefanatos *et al.* [11]–[13]. They presented Hecataeus, a framework that allows the user to anticipate hypothetical database schema evolution events and to examine their impact over a set of *queries and views* provided as input *by the user*. Unlike our tool, Hecataeus does not work at the *source code* level and does not consider the presence of *different* database access technologies.

Maule *et al.* [7] proposed an impact analysis approach for schema changes. They studied a commercial object-oriented content management system and statically analyzed the impact set of relational database schema changes on the source code. They implemented their approach for the ADO.NET (C#) technology. Liu *et al.* [6] proposed an approach to extract the attribute dependency graph out of a database application from its source code by using static analysis. Their purpose was to aid maintenance processes, particularly impact analysis. They implemented their approach for PHP-based applications.

Finally, several previous papers identify, extract and analyze database *usage* in application programs. The purpose of these approaches ranges from error checking [3], [16], SQL fault localization [1], to fault diagnosis [4].

In [8], we presented DAHLIA 1.0, a visualization tool that allows us to analyze the evolution history of a database over its lifetime. That tool makes use of the well-known city metaphor of CodeCity [17]; a city building represents a database table and its height, width and color are computed from its historical information. DAHLIA 1.0 allows developers to answer some questions related to database schema evolution like how the database schema evolves over time and which developer is the specialist of a given database schema part. While DAHLIA 1.0 only focuses on the database schema and its history, DAHLIA 2.0 now considers the database usage and the dependencies between the application source code and the database schema.

III. DAHLIA 2.0

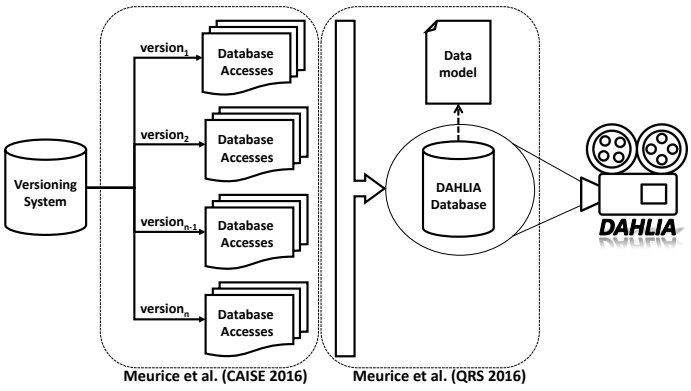


Fig. 2. Overview of our tool-supported approach.

Figure 2 depicts the overview of our tool-supported approach. That latter consists of two steps: (1) we extract relevant data related to the database usage of the given DISS and (2) we visualize that database usage. One strength of DAHLIA 2.0 is that it is not only restricted to the analysis of on unique system version; contrariwise, our approach can consider several versions of a DISS at the same time.

A. Data Extraction

Let us define $V = \{v_0, ..., v_n\}$, the set of system versions (available from the versioning repository) on which the developer wishes to focus ($n \geq 0$). For each successive version (and its corresponding source code version), we use the tool support we developed in another previous work dedicated to database access recovery in Java source code [10]. We use this static analysis approach, especially designed for Java systems, to automatically locate and extract all the database accesses that use JDBC (industry standard for database-independent connectivity between the Java programming language and relational databases), Hibernate and JPA. For each detected database access (and for each version), we extract the exact code locations where the access is executed as well as the database tables/columns manipulated by the access. Moreover, we also detect and extract all the ORM mappings defined between a class (resp. attribute) and a table (resp. column). For each version, we determine where and how each table/column is mapped to the Java code elements.

At this point, we obtain for each version, a set of the database accesses detected by our static analysis as well as the code location of each access and the database tables and columns involved in it. The code location of a given access is expressed by the minimal program path necessary for creating and executing the database access. The below example shows sample information gathered for a database access where a SQL query is executed at line 124 in DatabaseUtil.java. The current method in which the query execution occurs is called by OrderChecker.java at line 56. The database objects involved in this query are the $drug\_order$ table and $units$, one of its columns.

JDBC access: `'SELECT DISTINCT units FROM drug_order WHERE units is NOT NULL'`
Program path: [**OrderChecker.java**, line=**56**] → [**DatabaseUtil.java**, line=**124**]
Database schema objects:
↪ Database Tables: [**drug_order**]
↪ Database Columns: [**drug_order.units**]

We use a second tool support we implemented and detailed in [9] allowing us to organize the collected data of each version according to a particular data model. The central element of that model is the version. That model is composed of 4 main different parts:

a) The source code history: this part represents the history of the source code objects. A Java file may contain several classes, methods and attributes. Each code object may exist in several versions and, for each version, the object has a particular position in the code expressed as a couple of coordinates: a begin line and column, and an end line and column.

b) The database schema history: the database schema evolves over time and may have a different set of schema

objects. Only table and column objects are considered in the model. The database tables and columns may be present in several versions. Depending on the version, a column may have a different type.

c) The ORM mapping history: by means of an ORM (e.g., Hibernate/JPA), developers can define a mapping between an entity class and a table or between an attribute and a column. An ORM mapping may exist in several versions.

d) The database access history: this part represents the history of the database accesses, i.e., of the source code locations that provide an access to the database. Those database accesses use a particular technology (e.g., JDBC, Hibernate or JPA) to query the database and are located at a particular position in the source code (i.e., in a particular file, method and line). For each database access, the set of accessed database objects is recorded. A database access may, in turn, exist in several versions.

More details about the data model and the process organizing the collected data according to that data model are given in [9]. From that process, we obtain a relational database which stores the collected data according to the data model. This database can be thus queried.

B. Data Visualization

The database obtained by the previous data extraction phase is the only input required by DAHLIA 2.0. The visualization tool queries that database in order to compute and visualize information related to the database usage of a given DISS.

As previously explained, we extended our visualization tool DAHLIA 1.0 to allow developers to analyze the database usage of a system. In [8], DAHLIA 1.0 only considered the database schema history. The novelty in that extension is that DAHLIA 2.0 is now able to analyze the database usage by exploiting the links between the program source code and the database. The main role of DAHLIA 2.0 is to provide developers with a visual support to database-program co-evolution by analyzing the dependencies between the code and the database; it will thus allow assessing the costs of a future system change (e.g., *what if I modify that database table?*). We list below some of the novel features implemented in DAHLIA 2.0[1][2].

Visualizing the Database City: this features reuses the 3D city-metaphor that facilitates the visualization of very large database schemas. A database table is represented as a 3D building. We use the building height, width and color for representing database usage metrics. The user may select the metrics to affect to each dimension and may thus customize the city according to his/her needs. An example of metrics for the building height/width may be the number of files accessing the given table, the number of code locations accessing the given table, etc. Metrics for the building color may be the database access technology distribution (e.g., a particular color for all the database tables accessed by a given technology), the ORM mapping distribution (e.g., a particular color for all the database tables that are mapped to the code via an ORM), etc. That kind of metrics permits developers to instinctively detect the "sensitive" database parts hardly linked (accessed) to the code.

Figure 3 depicts an example of 3D database city that one can visualize within DAHLIA 2.0. Each building (table) has a height denoting the number of columns, a width representing the number of accessing queries sent from the source code, and a color representing the database access technology. As illustrated, the user can visualize the 2D corresponding table form.

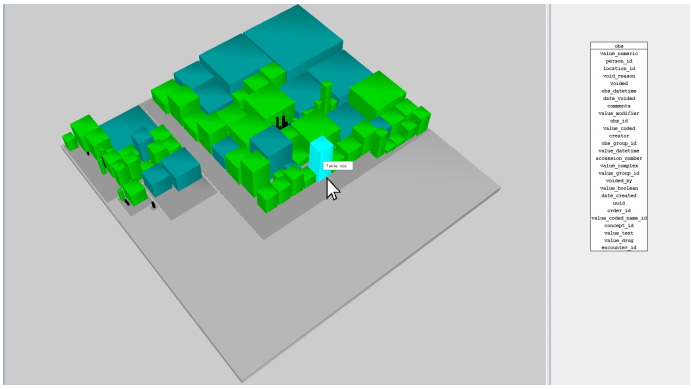


Fig. 3. A 3D database city as visualized within DAHLIA 2.0. The right panel shows the 2D form of a selected table.

Visualizing the Code City: that feature proposes a visualization of the program source code similar to [17] by representing a file as a 3D building. The novelty we propose is also into the metrics affected to the building height, width and color. For instance, the user may affect some metrics to the building height/width like the number of accessed tables by the given file, the number of locations in the given file accessing the database, the number of (SQL) queries detected in the given file, etc. Concerning the building color, the user may use metrics like the access technology distribution (e.g., a particular color for the files using a given database access technology), the ORM mapping distribution (e.g., a particular color for the files defining some ORM mappings), etc. Here again, that kind of metrics will allow the immediate detection of "sensitive" code parts.

Figure 4 shows an example of 3D code city as visualized within DAHLIA 2.0. Each building - a Java file in that case - has a height, i.e., the number of locations accessing the database), a width, i.e., the number of methods in the file, and a color for the database access technology(ies) used in the file (black = none, green = Hibernate, blue = JDBC, mix = JDBC & Hibernate).

Visualizing the links between the Database and Code city: that feature proposes a dual visualization; the database and code cities are side-by-side according to the metrics chosen by the user. It enables the user to assess the costs of a future database schema change or a code refactoring step. The user can click on a particular database table and visualize which

[1]Each of those features can be applied to any considered system versions; the latest version as well as any older one.

[2]All the features of DAHLIA 1.0 are included in DAHLIA 2.0. We only present in that paper the new features. The former ones are detailed in [8].

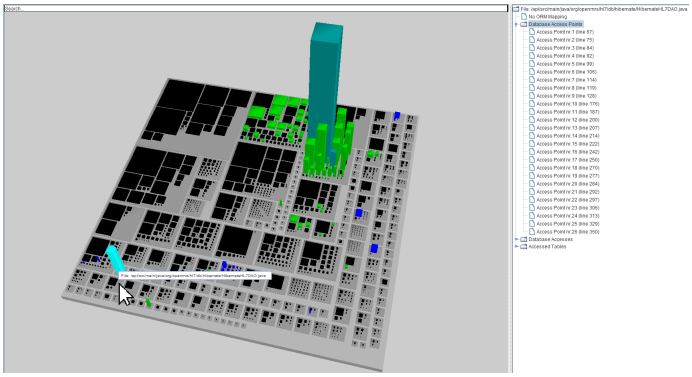


Fig. 4. A 3D code city as visualized within DAHLIA 2.0. The right panel shows information about a selected file, i.e., the access locations, the accessed tables and the ORM mappings.

part of the program source code accesses it. The results of a click on a table will be (1) the highlighting of all the files accessing it and (2) the accurate detection of all the code locations accessing it. Figure 5 depicts the database and code cities as visualized within DAHLIA 2.0. The database (left) and code (right) cities are side-by-side. The green tables are tables with Hibernate mapping, black tables are tables without any ORM mappings. The table height represents the number of columns while the table width is the number of SQL queries accessing it. The green files are files using Hibernate, blue files are files using JDBC and black files do not access the database. The file height represents the number of accessed tables while the width represents the number of locations accessing the database. Figure 5 illustrates the following scenario: the user plans to refactor the Java file HibernateConceptDAO.java and wishes to assess the costs of that evolution phase with help of DAHLIA 2.0. The user clicks on the HibernateConceptDAO class (highlighted building in the right city depicted in cyan). DAHLIA automatically and instantly highlights (cyan color) all the database tables (left city) accessed by that class. By this way, the user can directly have an estimation of the required effort to perform that evolution phase.

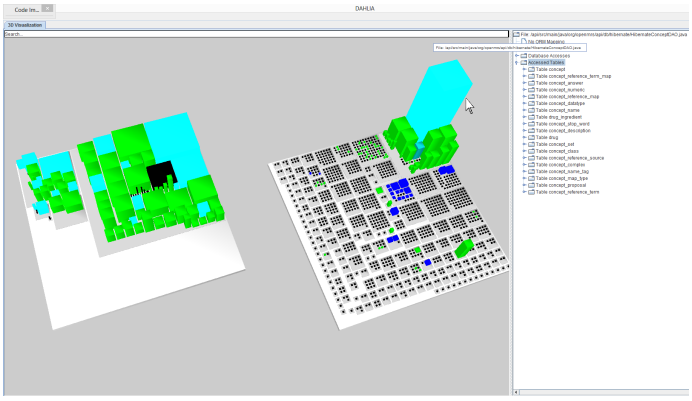
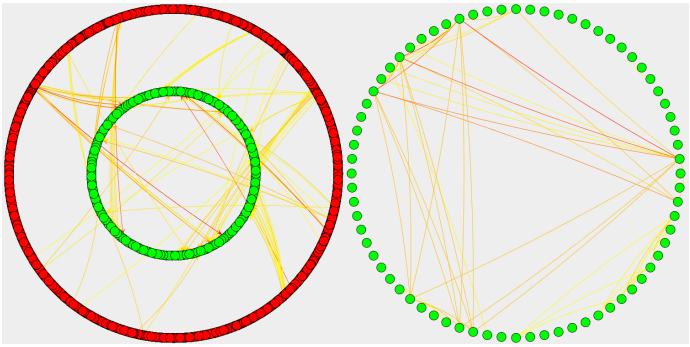


Fig. 5. Database (left) and Code (right) cities side-by-side as visualized within DAHLIA 2.0. The right panel shows information about a selected file.

Jumping into the code: in addition to the 3D support, the tool can list, on its right panel, a large set of information. For



(a) Dependencies between files and tables (b) Dependencies between tables

Fig. 6. Circular views as visualized within DAHLIA 2.0. Red bullets represent files, while green ones depict tables. Darker the color stronger the dependency.

instance, the user can decide to list (1) all the tables accessed by a given file, (2) all the precise code locations (precision in term of line of code) in a given file which allows accessing the database, (3) all the (SQL) queries accessing a given table (as well as their actual value, the accessed database objects and their execution location in the code), etc. Nevertheless, thanks to that information panel, the user has an accurate report on the database usage. Moreover, that panel allows the user to directly jump into the precise code locations which will require some modifications/adaptations in case of code/database change.

Visualizing the database-program dependencies within a circular view: that feature uses a circular visualization. Unlike the 3D visualization, this mode allows to directly visualize the dependencies between the program source code and the database in terms of intensity. Precisely determining the dependencies between a table and a file or between a table and another table can considerably help the user to assess the future impact of any change on the system. Figure 6 depicts two examples of circular visualization aiming to detect the dependencies (a) between the source code files and the database tables in term of access locations and (b) between the tables themselves in term of closeness - namely, two tables are close when they appear together in a same SQL query. Moreover, the user can decide to only select a particular table/file and display its dependencies with the other objects.

C. DAHLIA 2.0 Applied to Real-Life Systems

We selected 3 popular open-source data-intensive systems and we visualized their database usage within DAHLIA 2.0.

1) OSCAR (www.oscar-emr.com): OSCAR is an open-source ERM information system widely used in the health-care industry in Canada. Its primary purpose is to maintain electronic patient records and interfaces of a variety of other information systems used in the healthcare industry. OSCAR has been developed since 2002. OSCAR combines different ways to access the database like JDBC, Hibernate and JPA.

2) Broadleaf Commerce (www.broadleafcommerce.org): Broadleaf is an open-source, e-commerce framework written in Java on top of the Spring framework. It facilitates the development of enterprise-class, commerce-driven sites

by providing a robust data model, services, and specialized tooling that take care of most of the 'heavy lifting' work. Broadleaf has been developed since 2008. It uses a relational database accessed via JPA.

3) OpenMRS (www.openmrs.org): OpenMRS is a collaborative open-source project to develop software to support the delivery of healthcare in developing countries (mainly in Africa). It was conceived as a general-purpose EMR system that could support the full range of medical treatments. It has been developed since 2006. OpenMRS uses a MySQL database accessed via Hibernate and dynamic SQL (JDBC).

We selected those three systems because they have all a significant history and code size, and use different database access technologies. The use of several technologies together within the same system and the abstraction brought by ORM technologies may make comprehension and evolution difficult. Table I gives some characteristics of OSCAR, Broadleaf and OpenMRS. We respectively focused on 242, 118 and 164 versions of OSCAR, Broadleaf and OpenMRS in our data extraction process. The database schema size and the number of code lines of the latest considered version are included in Table I. Table II presents, for each system and technology supported, the total number of locations accessing the database in the latest considered system version.

TABLE I
SIZE METRICS OF THE SYSTEMS - CREATION DATE, NUMBER OF VERSIONS CONSIDERED IN THE VERSIONING REPOSITORY, NUMBER OF CODE LINES AND DATABASE SCHEMA SIZE.

System	Start Date	Versions	kLOC	Tables	Columns
OSCAR	11/2002	242	> 2000	512	15680
Broadleaf	12/2008	118	> 250	179	965
OpenMRS	05/2006	164	> 300	88	951

TABLE II
NUMBER OF DATABASE ACCESS LOCATIONS PER TECHNOLOGY

System	Database Accesses		
	JDBC	Hib	JPA
OSCAR	123 661	727	31 729
OpenMRS	77	687	0
Broadleaf	0	0	930

For each of those systems, we firstly computed the database, structured according to our data model (see Section III-A), containing all the required database usage information. The data extraction process for each system is further described in [9]. For each considered version, we can visualize the system database usage within DAHLIA 2.0. Figures 3, 4, 5 and 6 are visualizations of a recent version of OpenMRS. A large collection of pictures showing the use of DAHLIA 2.0 on each system is provided at www.info.fundp.ac.be/~lme/DAHLIA/ as well as an online video presenting DAHLIA 2.0 and its features.

IV. CONCLUSIONS

We presented DAHLIA 2.0, a novel visualization tool that allows us to analyze the database usage of dynamic and heterogeneous systems by visualizing the links between the source code and the database. It aims to support database-program co-evolution in a DISS. Our tool can deal with systems using several database access technologies together like ORM. While our data extraction phase (described in Section III-A) is specifically designed for Java systems, the visualization relies on a data model detailed in [9] and could become technology-independent with some minor adaptations.

In the future, we plan to conduct an empirical analysis of database usage evolution over time, and a study of program-database co-evolution patterns in DISS. In particular, we wish to analyze the evolution history of DISS to study the impact of adopting a new database access technology as well as the required effort to migrate from a technology to another one. Finally, our ultimate objective will be to contribute to (partially) automate the database schema change propagation process itself, via source code transformation techniques.

REFERENCES

[1] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold. Localizing SQL faults in database applications. In *Proc. of ASE '11*, pages 213–222. IEEE Comp. Soc., 2011.

[2] M. Golfarelli, S. Rizzi, and A. Proli. Designing what-if analysis: Towards a methodology. In *Proc. of DOLAP '06*, DOLAP '06, pages 51–58. ACM, 2006.

[3] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. of ICSE '04*, pages 645–654. IEEE Comp. Soc., 2004.

[4] M. A. Javid and S. M. Embury. Diagnosing faults in embedded queries in database applications. In *Proc. of EDBT/ICDT'12 Workshops*, pages 239–244. ACM, 2012.

[5] A. Karahasanović. *Supporting Application Consistency in Evolving Object-Oriented Systems by Impact Analysis and Visualisation*. PhD thesis, University of Oslo, 2002.

[6] K. Liu, H. B. K. Tan, and X. Chen. Aiding maintenance of database applications through extracting attribute dependency graph. *J. Database Manage.*, 24(1):20–35, January 2013.

[7] A. Maule, W. Emmerich, and D. S. Rosenblum. Impact analysis of database schema changes. In *Proc. of ICSE 2008*, pages 451–460, 2008.

[8] L. Meurice and A. Cleve. DAHLIA: A visual analyzer of database schema evolution. In *CSMR-WCRE '14*, pages 464–468, 2014.

[9] L. Meurice, C. Nagy, and A. Cleve. Detecting and preventing program inconsistencies under database schema evolution. In *Proc. of QRS 2016*. IEEE Computer society, 2016. to appear. https://staff.info.unamur.be/lme/QRS2016/MeuriceEtAl.pdf.

[10] L. Meurice, C. Nagy, and A. Cleve. Static analysis of dynamic database usage in java systems. In *Proc. of CAiSE '16*, LNCS. Springer, 2016. to appear. https://staff.info.unamur.be/lme/CAISE16/MeuriceEtAl.pdf.

[11] G. Papastefanatos, F. Anagnostou, Y. Vassiliou, and P. Vassiliadis. Hecataeus: A what-if analysis tool for database schema evolution. In *Proc of CSMR '08*, pages 326–328, April 2008.

[12] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. What-if analysis for data warehouse evolution. In IlYeal Song, Johann Eder, and ThoManh Nguyen, editors, *Data Warehousing and Knowledge Discovery*, volume 4654 of *LNCS*, pages 23–33. Springer, 2007.

[13] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. Hecataeus: Regulating schema evolution. In *Proc of ICDE 2010*, pages 1181–1184, March 2010.

[14] D. Qiu, B. Li, and Z. Su. An empirical analysis of the co-evolution of schema and code in database applications. 2013.

[15] E. Rahm and P. A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, December 2006.

[16] M. Sonoda, T. Matsuda, D. Koizumi, and S. Hirasawa. On automatic detection of SQL injection attacks by the feature extraction of the single character. In *Proc. of SIN '11*, pages 81–86. ACM, 2011.

[17] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In Wilhelm Schfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE Companion*, pages 921–922. ACM, 2008.