Mining, Analyzing and Evolving Data-intensive Software Ecosystems

Csaba Nagy, Michele Lanza, and Anthony Cleve

Abstract Managing data-intensive software ecosystems has long been considered an expensive and error-prone process. This is mainly due to the often implicit consistency relationships between applications and their database(s). In addition, as new technologies emerged for specialized purposes (e.g., key-value stores, document stores, graph databases), the common use of multiple database models within the same software (eco)system has also become more popular. There are undeniable benefits of such multi-database models where developers use and combine technologies. However, the side effects on database design, querying, and maintenance are not well-known. In this chapter, we elaborate on the recent research effort devoted to the mining, analyzing, and evolving data-intensive software ecosystems. We focus on methods, techniques, and tools providing developers with automated support. We cover different processes, including automatic database query extraction, bad smell detection, self-admitted technical debt analysis, and evolution history visualization.

Name of First Author
Name, Address of Institute, e-mail: name@email.address

1 Introduction

Data-intensive software ecosystems comprise one or several databases and a collection of applications connected with the former. They constitute critical assets in most enterprises, since they support business activities in all production and management domains. They are typically old, large, heterogeneous and highly complex.

Database interactions play a crucial role in data-intensive applications, as they determine how the system communicates with its database(s). When the application sends a query to its database, it is the database's responsibility to handle the query with its best performance, and the developer has limited control: If the query is not well-formed or not handled correctly in the program code, it generates extra load on the database side that affects the application's performance [1] [2]. In the worst case, it can lead to errors, bugs, or even security vulnerabilities such as code injection.

In the last decade, the emergence of novel database technologies, and the increasing use of dynamic data manipulation frameworks have made data-intensive ecosystem analysis and evolution even more challenging.

In particular, the increasing use of NoSQL databases poses new challenges for developers and researchers. A prominent feature of such databases is that they are *schema-less*, offering greater flexibility in handling data. This freedom strikes back when it comes to maintaining an evolving data-intensive application [3, 4]. Another challenging trend is the development of *hybrid* multi-database architectures [5], also called *hybrid polystores*, where relational and NoSQL databases co-exist within the same system and must be queried and evolved jointly and consistently.

We present recent research initiatives aiming to address those challenges. In Section 2 we discuss mining techniques to determine how data is stored and managed in a data-intensive ecosystem. Those techniques can automatically identify and extract the database interactions from the system source code. Section 3 elaborates on static analysis and visualization techniques that exploit the mined information on the storage and manipulation of the ecosystem data. In Section 4 we summarize the findings of empirical studies related to data-intensive software ecosystems. We provide concluding remarks and anticipate future directions in Section 5.

2 Mining Techniques

2.1 Introduction

Managing a data-intensive software ecosystem requires a deep understanding of its architecture, since it consists of many subsystems which depend on one another. They use each other's public services and APIs—they communicate. A subsystem may rely on one or more databases and their data will likely travel through the entire ecosystem. For maintaining and evolving this interconnected system network, it is fundamental to understand how data is handled all over it.

In this section, we present approaches to mining how data is stored and managed in a data-intensive ecosystem. Such knowledge can serve various purposes, *e.g.,* reverse engineering, re-documentation, visualization, or quality assurance approaches.

We present two techniques to study the interaction points in the applications where they communicate with databases. Both techniques are based on static analysis, and hence, do not require the application to execute, but only the source code. This can be particularly important for an ecosystem where dynamic analysis is even more challenging, if not impossible, in most situations.

In Section 2.2, we present a static approach to identifying, extracting, and analyzing database accesses in Java applications. This technique can be used for applications with libraries that communicate through SQL statements (*e.g.,* JDBC or Hibernate). It locates the database interaction points and traces back the potential SQL strings at that location, and can handle dynamically constructed strings. In Section 2.3, we show a similar static approach for NoSQL databases. This technique was developed to analyze MongoDB usage in Java and JavaScript. JavaScript is a highly dynamic language where types are often not available explicitly. The approach tries to alleviate the limitations of static analysis by using heuristics.

2.2 Static Analysis of Relational Database Accesses

Database manipulation is central in the source code of a data-intensive system. It serves data to all its other parts, and enables the application to query the information needed for all operations and persist changes to its actual state. The database manipulation code is often separated in the codebase. For example, object-oriented languages usually follow the DAO (Data Access Object) design pattern to isolate the application/business layer from the persistence layer. A DAO class implements all the functionality required for fetching, updating, and removing its domain objects. For example, a *UserDao* would be responsible for handling *User* objects mapped to *user* entities in the database.

The complexity of the manipulation code depends on the APIs or libraries used for database communication, and many libraries are available depending on the developers' needs. Several factors may determine the library's choice, such as the programming language, database, and required abstraction level. A large-scale empirical study by Goeminne *et al.* [6] found, for example, JPA, Hibernate, and JDBC among the most popular ones. Moreover, many systems use combinations of multiple libraries. From the mining point of view (and also from the developer's perspective), these libraries can partly or completely hide the actual SQL queries executed by the programs, generating queries at runtime before sending them to the database server.

Fig. 1 shows an example code snippet executing a SQL query through the JDBC API. The `Statement.execute(...)` sends the query to the database (line 10).

It is part of the standard `java.sql` API that provides classes for "*accessing and processing data stored in a data source (usually a relational database).*"[1] The query string is a result of string operations (*e.g.,* lines 9, 14) depending on conditions (*e.g.,* lines 8, 19).

```
1  public class ProviderMgr {
2      private Statement st;
3      private ResultSet rs;
4      private boolean ordering;
5
6      public void executeQuery(String x, String y){
7          String sql = getQueryStr(x);
8          if (ordering)
9              sql += "order by " + y;
10         rs = st.execute(sql);
11     }
12
13     public String getQueryStr(String str){
14         return "select * from " + str;
15     }
16
17     public Provider[] getAllProviders(){
18         String tableName = "Provider";
19         String columnName = (...) ? "provider-id" :
               "provider_name";
20         executeQuery(tableName, columnName) ;
21         // ...
22     }
23 }
```

Fig. 1 Java code example executing a SQL query

Fig. 2 presents a similar example usage of Hibernate to send an HQL query to the database. HQL (the Hibernate Query Language) is the SQL-like query language of Hibernate. It is fully object-oriented and understands notions like inheritance, polymorphism and association. The example shows a code snippet using the `SessionFactory` API. The HQL statement on line 10 queries products belonging to a given category. Hibernate transforms this query to SQL and sends it to the database when the `list()` method is invoked on line 12.

Meurice *et al.* addressed the problem of recovering traceability links between Java programs and their databases [7]. They propose a static analysis approach to identify the source code locations where database queries are executed and extract the set of actual SQL queries for each location. The approach is based on algorithms that operate on the application's call graph and the methods' intra-procedural control flow. It considers three of the most popular database access technologies used in Java systems, according to Goeminne *et al.* [6], namely JDBC, Hibernate, and JPA.

[1] https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html

```
1  public class ProductDaoImpl implements ProductDao {
2      private SessionFactory sessionFactory;
3
4      public void setSessionFactory(SessionFactory
           sessionFactory) {
5          this.sessionFactory = sessionFactory;
6      }
7
8      public Collection loadProductsByCategory(String
           category) {
9          return this.sessionFactory.getCurrentSession()
10             .createQuery("from Product product where
                   category=?")
11             .setParameter(0, category)
12             .list();
13     }
14 }
```

Fig. 2 Java code snippet executing an HQL query

An overview of Meurice *et al.*'s approach can be seen in Fig. 3. First, it takes the application source code and database schema as input. It parses the schema and analyzes the source code to identify the locations where the application interacts with the database. Then, it extracts the SQL queries sent to the database at these locations and then parses the queries. The final output is a set of database access locations, their queries, and the database objects (tables and columns) impacted/accessed at these locations.

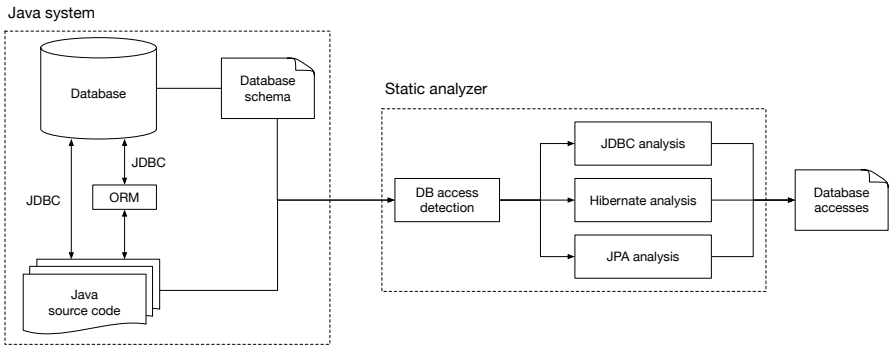

Fig. 3 Overview of the query extraction approach

Different technologies (*i.e.,* JDBC, Hibernate, or JPA) require different analysis approaches, like SQL dialects require specific parsers. A static extraction approach of SQL queries constructed through deeply embedded string operations would require inter-procedural data- and control-flow analyses.

In some cases, even such techniques might not be able to extract the entire query. Thus, a query might be incomplete when its parts cannot be retrieved statically. For example, a user may enter credentials in a login form to be validated in the database. The user input is known at runtime, but the static analyzer only sees that the `email` and `password` variables are used to construct the query. The parser must tolerate such incomplete statements, and in the end, the extraction process balances precision and the computation overhead of in-depth static analyses.

Meurice *et al.* evaluated their approach on three open-source systems (Oscar, OpenMRS and Broadleaf) with sizes ranging from 250 kLOC to 2,054 kLOC having 88–480 tables in their databases. The first two are popular electronic medical record (EMR) systems, and the third one is an e-commerce framework. They could extract queries for 71.5–99% of database accesses with 87.9–100% of valid queries.

In their follow-up work [8], they analyzed the evolution of the same three systems as they have been developed for more than seven years. They jointly analyzed the changes in the database schema and the related changes in the source code by focusing on the database access locations.

They made several interesting observations. For example, the very same tables could be accessed by different data manipulation technologies within the programs. Database schemas also expanded significantly over time. Most schema changes consisted in adding new tables and columns. A significant subset of database tables and columns were not accessed (any longer) by the application programs, resulting in "dead" schema elements.

Co-evolving schema and programs is not always a trivial process for developers. Developers seem to refrain from evolving a table in the database schema since this may make related queries invalid in the programs. Instead, they probably prefer to add a new table by duplicating its data and incrementally updating the programs to use the new table instead of the old one. Sometimes the old table version is never deleted, even when not accessed anymore.

2.3 Static Analysis of NoSQL Database Accesses

NoSQL ("Not Only SQL") technologies emerged to tackle the limitations of relational databases. They offer attractive features such as scale-out scalability, cloud readiness, and schema-less data models [9]. New features come at a price, however. For example, schema-less storage allows faster data structure changes, but the absence of explicit schema results in multiple co-existing implicit schemas. The increased complexity makes developers' operational and maintenance burden heavier [10, 11].

Efforts have been made to address the challenges of NoSQL systems. A popular one is to support schema evolution in the schema-less NoSQL environment [12]. For example, researchers study automatic schema extraction [13], schema generation [14], optimization [15], and schema suggestions [16]. Behind the scenes, such approaches mainly rely on a static analysis of the source code or the data, operating on the part of the source code implementing the database communication.

Cherry *et al.* addressed the problem of retrieving database accesses from the source code of JavaScript applications that use MongoDB [17].

Static analysis of JavaScript is known to be extremely difficult. Existing techniques [18, 19] usually struggle to handle the excessively dynamic features of the language [20], and approaches with type inference [21], data flow [22], or call graphs [23] need to balance between scalability and soundness. Cherry *et al.* alleviate the limitations of the static analysis by using heuristics.

Fig. 4 presents a typical schema definition in Mongoose,[2] a popular object-modeling library to facilitate working with MongoDB in JavaScript. First, the `mongoose` module is included using the built-in `require` function. Then a schema is created through the `mongoose.Schema(...)` API call. In Mongoose, a `Schema` is mapped to a MongoDB collection and defines the structure of the documents within that collection. To work with a `Schema`, a `Model` is needed in Mongoose. Hence, line 9 creates a `Model` in Fig. 4. Finally, the model gets exported to be used externally (line 11).

```
1  const mongoose = require("mongoose");
2
3  let SmartphoneSchema = new mongoose.Schema({
4      name: String,
5      price: Number,
6      inStock: Boolean
7  });
8
9  const Smartphone = mongoose.model("smartphones",
       SmartphoneSchema);
10
11 module.exports = Smartphone;
```

Fig. 4 Mongoose schema definition example

Fig. 5 shows an example usage of the model in Fig. 4. The model is imported using the `require` function (line 1). An instance of a model is a `Document` in Mongoose that can be created and saved in various ways. The example uses the `Document.save()` method of the iPhone Document instance (line 5). Finally, line 8 shows a simple query to find documents.

Cherry *et al.* look for similar MongoDB interactions in JavaScript applications. The aim is to identify every statement that operates with the database. For this purpose, they gathered method signatures from reference guides of MongoDB Node Driver 3.6 and Mongoose 5.12.8. They selected the methods that access the database for one of the following operations: (1) creates a new collection/document; (2) updates the content of documents or a collection; (3) deletes documents from a collection; (4) accesses the content of documents. Overall, they identified 179 methods, 74 from MongoDB Node Driver and 105 from Mongoose.

[2] https://mongoosejs.com/

```
1  Smartphone = require("./smartphones.js");
2
3  // ...
4  iPhone = new Smartphone("iPhone 13 Pro", 999, true);
5  await iPhone.save();
6
7  // ...
8  iPhones = await Smartphone.find({name: /iPhone/});
```

Fig. 5 Mongoose query example

Fig. 6 presents an overview of the main steps of the approach.

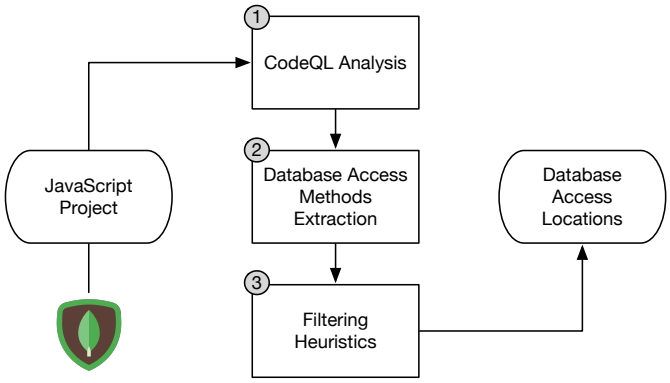

Fig. 6 Approach overview

First, it analyzes a JavaScript project with CodeQL,[3] a code analysis engine developed by GitHub. Code can be queried in CodeQL as if it were data in an SQL-like query language. Accordingly, the approach then runs queries to find the database access methods. The next step applies filtering heuristics to improve the precision by eliminating method calls in potential conflict with other APIs. They defined seven heuristics. For example, the "*the receiver should not be '_'*" heuristic avoids potential collisions with the frequently used Lodash[4] library. The outcome is a list of source code locations accessing the database with details of the access (*e.g.,* API used, receiver, context).

An example use case of the approach is the analysis of the evolution of systems database usage. Cherry *et al.* presented two case studies on Bitcore[5] and Overleaf.[6]

[3] https://codeql.github.com/

[4] https://lodash.com/

[5] https://github.com/bitpay/bitcore

[6] https://github.com/overleaf/web

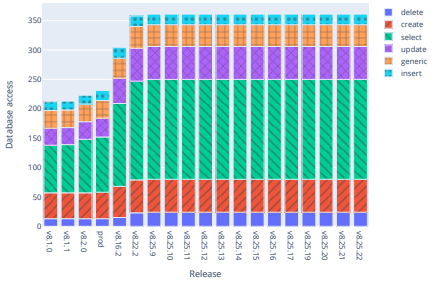

Fig. 7 Evolution of Bitcore

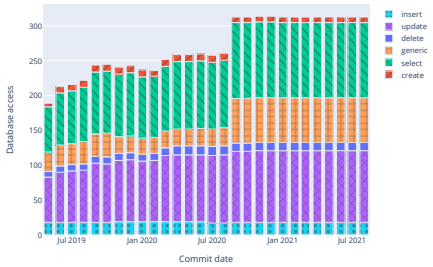

Fig. 8 Evolution of Overleaf

Fig. 7 shows the database access methods in the different releases of Bitcore. The project has 4.2K stars and 2K forks on GitHub. It has a multi-project infrastructure with a MongoDB database in its core. The most represented database operation is *select* with 170 distinct method calls. One can also see a major change in the number of database accesses around v8.16.2. A closer look reveals that a commit[7] adds numerous models and methods interacting with it. It is a new feature: "*[Bitcore] can now sync ETH and get wallet history for ERC20 tokens*"—says the commit message.

Fig. 8 shows the evolution of Overleaf, a popular online, collaborative LaTeX editor. Overleag's database usage differs from Bitcore with more prominent data modifications. Indeed, there are more updates (34%, 108) than selects (32%, 103). There was also an abrupt change in database accesses between September and October 2020. Overleaf was migrated from MongoJS to MongoDB Node Driver then.

Cherry *et al.* evaluated the accuracy of their approach on a manually validated oracle of 307 open-source projects. They reached promising results, achieving a precision of 78%.

Such an approach is the first step toward additional database access API usage analyses in JavaScript applications. It is required, for example, to analyze the evolution of systems [3], help their developers propagate schema changes [24], or identify antipatterns [2].

2.4 Reflections

We presented two static analysis approaches to study how applications communicate with their databases. We first discussed communication with relational databases. The programming context here was Java, a statically typed language. Then we learned a technique for MongoDB as an example of a popular NoSQL database in the dynamically typed language context of JavaScript applications.

[7] https://github.com/bitpay/bitcore/commit/d08ea9

To extract SQL queries, pioneer work was published by Christensen *et al.* [25]. They propose a static string analysis technique that translates a given Java program into a flow graph and then generates a finite-state automaton. Gould *et al.* propose a method based on an interprocedural data-flow analysis [26, 27]. Maule *et al.* use a similar k-CFA algorithm and a software dependence graph to identify the impact of relational database schema changes upon object-oriented applications [28]. Brink *et al.* present a quality assessment approach for SQL statements embedded in PL/SQL, COBOL, and Visual Basic code [29]. The initial phase of their extracts the SQL statements from the source code using control and data-flow analysis techniques. Annamaa *et al.* presented Alvor, a tool that statically analyzes SQL queries embedded into Java programs. Their approach is based on an interprocedural path-insensitive constant propagation analysis [30] similar to the one presented by Meurice *et al.* [31]. Ngo and Tan use symbolic execution to extract database interaction points from web applications [32]. They work with PHP applications of sizes ranging from 2–584 kLOC. Their method can extract about 80% of database interactions. PHP applications were also studied by Anderson *et al.*, who proposed program analysis for extracting models of database queries [33, 34, 35]. They implement their approach in Rascal as part of the PHP AiR framework. A similar approach was also presented by Manousis *et al.* [36]. They describe a language-independent abstraction of the problem and demonstrate it with a tool implementation for C++ and PHP.

Recent research also targeted Android, where SQL is preferred instead of higher-level abstractions (*e.g.,* an ORM) that may affect performance. Lyu *et al.* studied local database usage in Android applications [37]. They look for invocations of SQLite APIs and their queries. Li *et al.* presented a general string analysis approach for Android [38]. They define an intermediate representation (IR) of the string operations performed on string variables. This representation captures data-flow dependencies in loops and context-sensitive call site information.

There are also dynamic approaches. Cleve *et al.* explored aspect-based tracing and SQL trace analysis for extracting implicit information about program behavior and database structure [39]. Noughi *et al.* mined SQL execution traces for data manipulation behavior recovery and conceptual interpretation [40, 41]. Oh *et al.* proposed a technique to extract dependencies between web components (*i.e.,* Java Server Pages) and database resources. Using the proxy pattern, they dynamically observe the database-related objects in the Java standard library.

Some recent approaches also targeted NoSQL databases. In particular, they extract models from the JSON document database [42, 43, 44, 13]. Some approaches also deal with schema generation [14], optimization [15], and schema suggestions [16]. Also interesting to note is the work of Störl *et al.*, who studied schema evolution and data migration in a NoSQL environment [12]. As a similar approach to Cherry *et al.* [17], Meurice *et al.* implemented an approach to extract the database schema of MongoDB applications written in Java [3]. They applied their method to analyze the evolution of Java systems.

3 Analysis Techniques

3.1 Introduction

Once we mined information on the storage and management of data in the ecosystem, we can analyze it for various purposes.

In this section, we present two techniques. First, we show static analysis approaches in Section 3.2, which rely on the mining techniques introduced in the previous section. Then we present visualization methods in Section 3.3 to analyze the dependencies between the database and different components of an ecosystem.

3.2 Static Analysis Techniques

A database is a critical component of a data-intensive ecosystem. It has to be readily available, and its response time influences the usability of the entire ecosystem. It has been shown that the structure of a database can evolve rapidly, reaching hundreds of tables or thousands of database objects [31]. Moreover, because the application code and the database depend on each other, they evolve in parallel [8], resulting in increased complexity of the database communication code. This layer must remain reliable, robust, and efficient. Here, we show example approaches to help maintain database interactions between systems of an ecosystem and their databases.

3.2.1 Example 1: SQLInspect—A Static Analyzer

Static analyzers[8] can help detect fault-prone and inefficient database usage, *i.e.,* code smells, in early development phases. A lightweight analyzer can pinpoint a mistake already in the IDE before the developer commits it.

Nagy *et al.* presented SQLInspect,[9] a tool to identify code smells in SQL queries embedded in Java code [45, 2]. SQLInspect implements a combined static analysis of the SQL statements in the source code, the database schema, and the data in the database. Its static analysis relies on the approach of Meurice *et al.* presented in Section 2. It uses a path-sensitive string analysis algorithm to extract SQL queries from the Java code and implements smell detectors on the ASG of a fault-tolerant SQL parser. The supported SQL smells are based on the *SQL Antipatterns* book of Karwin [1]. SQLInspect can also perform additional analyses: (1) It supports the inspection of the (interprocedural) slice of the statements involved in the query con-

[8] The term "static analysis" is conflated. In this section, we call "static analyzer" a tool implementing source code analysis algorithms and techniques to automatically find bugs. The more general term, "static analysis" (or static program analysis) is the analysis of a program performed without executing it.

[9] https://bitbucket.org/csnagy/sqlinspect/

struction; (2) Table/column access analysis can help determine which Java methods access specific tables or columns; (3) It calculates embedded SQL metrics (*e.g.,* number of joins, nested select statements) to identify problematic or poorly designed classes and SQL statements. The tool is also available as an Eclipse plug-in. A screenshot of a query slice in the Eclipse plug-in can be seen in Fig. 9.

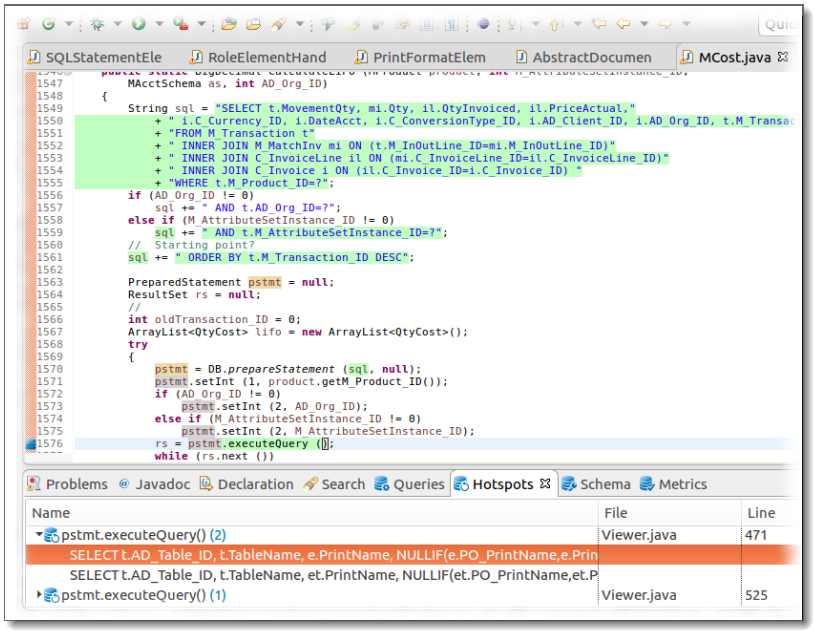

Fig. 9 A query slice in the Eclipse plug-in of SQLInspect

SQLInspect has been used in various studies. Muse *et al.* relied on it to study the prevalence, impact, and evolution of SQL code smells [46]. They also used it in an empirical study of data access self-admitted technical debt [47]. Gobert *et al.* employed SQLInspect to study developers' testing practices of database access code [48, 49]. Ardigò *et al.* also relied on it to visualize database accesses of a system [50, 51].

3.2.2 Example 2: Preventing Program Inconsistencies

Any software system is subject to frequent changes [52], which often hit the database schema [53, 54, 55]. When the schema evolves, developers need to adapt the applications where it accesses the changed schema elements [28, 56, 57]. This adaptation process is usually achieved manually. Thus, it can be error-prone and result in database or application decay [58, 59].

Meurice *et al.* proposed an approach to detect and prevent program inconsistencies under database schema changes [60]. Their what-if analysis simulates future database schema modifications to estimate how they affect the application code. It recommends to developers where and how they should propagate the schema changes to the source code. The goal is to ensure that the programs' consistency is preserved.

The core idea of the approach is first to analyze the evolution of the system, focusing on its schema changes. They collect metrics to estimate the effort required in the past for adapting the applications to database schema changes. For example, they look for renamed or deleted tables and columns and estimate from the commits the time needed to solve them in the code. To analyze the codebase and the schema, they rely on the previous analysis approach we also presented in Section 2. They run the analysis on each earlier system version and build a historical dataset. This dataset is designed to replay database schema modifications and estimate their impact on the source code. It describes all versions of database tables' columns with their links to source code entities where they are accessed in the application code.

Meurice *et al.* demonstrated their what-if analysis on three open-source Java systems of significant size: OpenMRS,[10] Broadleaf Commerce,[11] and OSCAR.[12]

They collected 323 database schema changes and randomly selected 130 for manual evaluation. They compared the tool's recommendations to the developers' actual modifications. The approach made 204 suggestions for the 130 cases: 99% were correct, and only 1% were wrong. The tool missed recommendations for 5% (6/130) of the changes. The results show impressive potential in detecting and preventing program inconsistencies.

3.3 Visualization

3.3.1 Introduction

Software visualization is "the use of [..] computer graphics technology to facilitate both the human understanding and effective use of computer software." [61]. It is a specialization of *information visualization* [62]. In the 18th century, starting with Playfair, the classical methods of plotting data were developed. In 1967 Bertin published "Semiology of Graphics" [63], where he identified the basic elements of diagrams. Later, Tufte published a theory of data graphics that emphasized maximizing the density of useful information [64, 65, 66]. Bertin's and Tufte's theories led to the development of the information visualization field, which mainly addresses the issues of how certain types of data should be depicted. The goal of information visualization is to visualize any kind of data. According to Ware [67], visualization is the preferred way of getting acquainted with large data.

[10] www.openmrs.org

[11] www.broadleafcommerce.org

[12] www.oscar-emr.com

Software visualization deals with software, both in terms of run-time behavior (dynamic visualization) and structure (static visualization). It has been widely used by the reverse engineering and program comprehension research community [68, 69, 70, 71, 72] to uncover and navigate information about software systems. In the more specific field of software evolution, mining software repositories, and software ecosystems, visualization has also proven to be a key technique due to the sheer amount of information that needs to be processed and understood.

Here we show two foundational approaches in the scientific literature to visualize applications, databases, and their interactions.

3.3.2 Example 1: DAHLIA

Data access APIs enable applications to interact with databases. For example, JDBC provides Java APIs for accessing relational databases from Java programs. It allows applications to execute SQL statements and interact with a SQL-compliant database. JDBC is considered a lower-level API with its advantages and disadvantages. For example, clean and simple SQL processing or good performance vs. complexity and DBMS-specific queries. Higher-level APIs such as Hibernate ORM (Object-Relational Mapping) try to tackle the object-relational impedance mismatch [73]. In return, such mechanisms partially or wholly hide the database interactions and the executed SQL queries. In this context, manually recovering links between the source code and the databases may prove complicated, hindering program comprehension, debugging, and maintenance tasks.

Meurice *et al.* developed DAHLIA to help developers with a software visualization approach [7, 74]. It allows developers to analyze database usage in highly dynamic and heterogeneous systems. The goal is to support software comprehension and database program co-evolution. For this purpose, DAHLIA extracts and visualizes database interactions in the source code to derive useful information about database usage. It relies on the data-access extraction described in Section 3 [31] and analyzes the evolution of the system by mining its development history [60].

DAHLIA has multiple views. It can visualize the *database as a city* using the 3D city metaphor [75]. This view represents a database table as a 3D building with its height, width, and color calculated from database usage metrics. For example, metrics for the building height/width can be the number of files accessing the given table or the number of code locations accessing the given table. Metrics for the building color can be the database access technology distribution. An example of this view can be seen in Fig. 10.

Another view shows the *code city*. However, compared to the traditional code city, this city maps database metrics to the buildings. For example, the user may ask to calculate the buildings' height/width from the number of accessed tables by the given file or the number of database access locations in the given file. For the color, the user may use metrics such as the access technology distribution (*i.e.,* different colors for database access technologies).

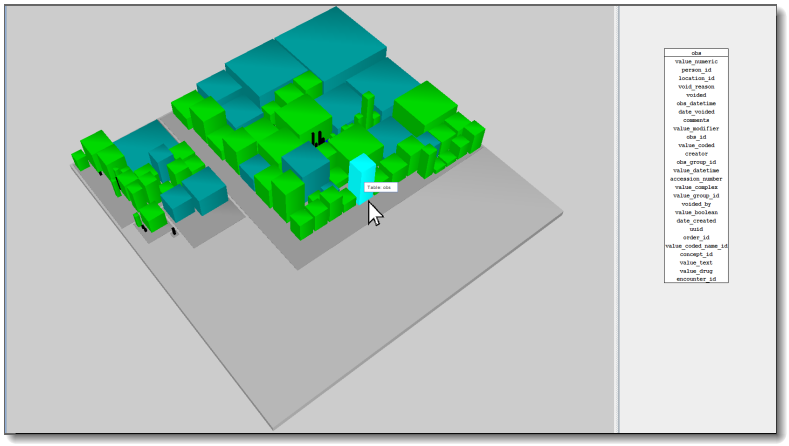

Fig. 10 A 3D database city in DAHLIA. The right panel shows details of the selected table.

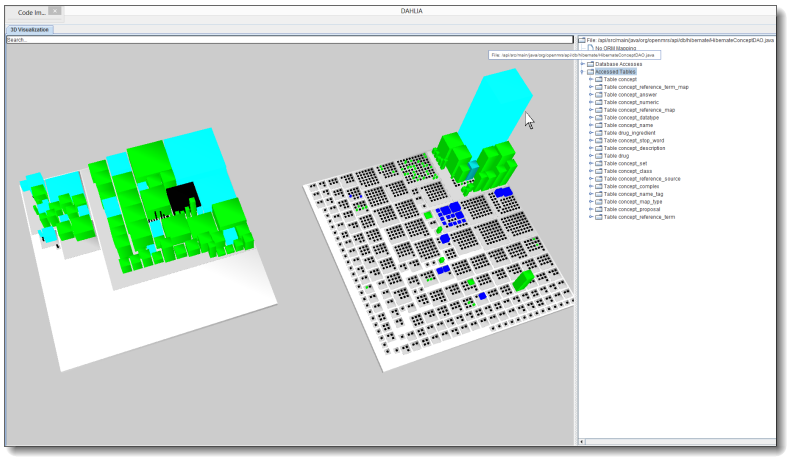

Fig. 11 Database (left) and Code (right) cities side-by-side in DAHLIA.

Visualizing *links between the database and code cities* is also possible by showing the two cities side-by-side in one view. When the user selects a table, DAHLIA highlights all the files accessing it. This view can be seen in Fig. 11. In the figure, the green tables are accessed with Hibernate mapping, and the black tables are without any ORM mappings. A table's height represents its number of columns, and its width is the number of SQL queries accessing it. The green files use Hibernate, the blue files use JDBC, and the black ones do not access the database. A file's height represents the number of accessed tables, and its width represents the number of locations accessing the database. In Fig. 10, the user selected `HibernateConceptDAO.java` (highlighted in the right code city with cyan). DAHLIA highlighted all its tables in the left database city with a cyan color.

Overall, DAHLIA is a visualization tool to analyze the database usage of dynamic and heterogeneous systems by showing the links between the source code and the database. It was designed to deal with systems using multiple database access technologies, aiming to support database program co-evolution.

3.3.3 Example 2: M3TRICITY

As we could see in the previous example, the city metaphor for visualizing software systems in 3D has been widely explored and has led to various implementations and approaches. Now we look at M3TRICITY,[13] a code city visualization that focuses on data interactions [50, 51].

Data is usually managed using databases, but it is often simply stored in files of various formats, such as CSV, XML, and JSON. Data files are part of a project's file system and can thus be easily retrieved. However, a database is usually not contained in the file system, and its presence can only be inferred from the source code, which implements the database accesses.

M3TRICITY represents data files in the city and maps simple metrics on their meshes (i.e., their shapes rendered in the visualization). It adds the database to the visualization using the free space of the sky above or the underground below the city. M3TRICITY infers the database schema using SQLINSPECT [2]. It also collects metrics of the database entities, such as the number of columns of tables or the number of classes accessing them. The city layout uses a history-resistant arrangement proposed by Pfahler *et al.* [76], *i.e.,* new entities remain at their reserved place throughout the evolution. The resulting view seamlessly integrates data sources into a software city and enables a comprehensive understanding of a system's source code and data.

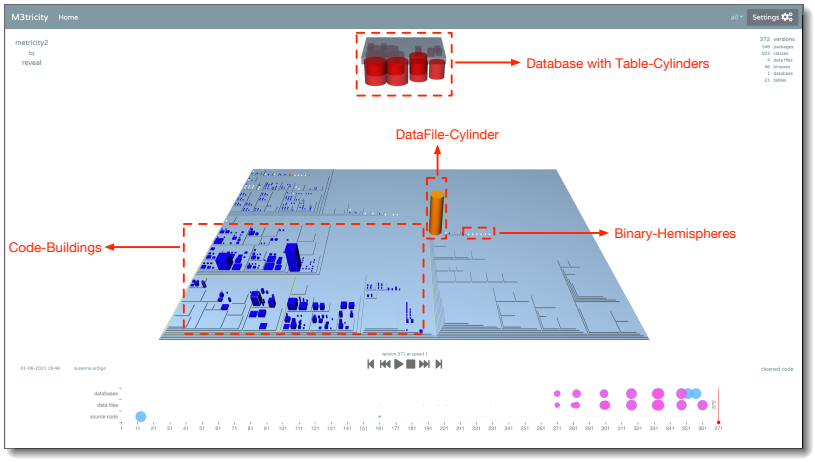

Fig. 12 The main page of M3TRICITY

[13] M3TRICITY is a web application available online at https://metricity.si.usi.ch/v2.

Fig. 12 shows a screenshot of m3triCity visualized in m3triCity. The software city is in the center with the database cloud above the city. Information panels present the repository name (top left), the system metrics (top right), the actual commit (bottom left), and its commit message (bottom right). The timeline at the bottom depicts the evolution of the project, where one can spot significant changes in the metrics. The evolution can be controlled with the buttons below the city.

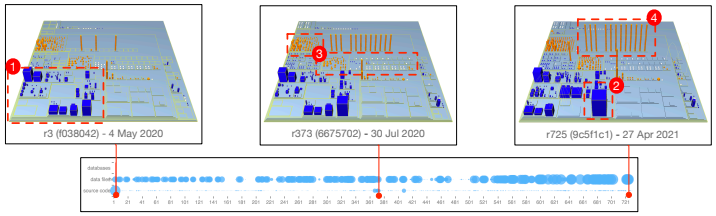

Fig. 13 The evolution of the SwissCovid Android App in m3triCity

Fig. 13 presents the evolution of the official SwissCovid Android App[14] and highlights three revisions. The timeline at the bottom shows regular contributions to data files (blue bubbles in the middle). Indeed, the XML files grow from an initial 10k to 25k lines. Interesting districts can be spotted in the evolution. The Java classes are primarily located on the bottom-left side of the city ①. The robust `SecureStorage.java` class ② stands out. This is the encrypted implementation of `android.content.SharedPreferences`,[15] the primary storage implementation with a critical role in the contact tracing app of Switzerland. We can see the neighborhoods of resource files ③ with tiny PNG and SVG files and folders of smaller layout XMLs. Another noticeable district is a folder with `strings.xml` files of various languages ④. The initial version supported three official Swiss languages (Italian, French, and German). As the app evolves, the XMLs grow, and the number of languages increases to twelve.

Overall, m3triCity is an example of adding "data" as first-class citizens to a widely used software visualization approach, the city metaphor.

3.4 Reflections

In this section, we learned a static analysis approach to identify SQL code smells in the database communication layer of data-intensive applications. We could also see a what-if analysis technique to detect and prevent program inconsistencies under database schema changes. Then we discussed two visualization methods to analyze dependencies between the database and different components of an ecosystem.

[14] https://github.com/SwissCovid/swisscovid-app-android

[15] https://developer.android.com/reference/android/content/SharedPreferences

Statatic Analysis. Common mistakes in SQL have been in the interest of many researchers. Brass *et al.* worked on automatically detecting logical errors in SQL queries [77] and then extended their work by recognizing common semantic mistakes [78]. Their SQLLint tool automatically identifies such errors in (syntactically correct) SQL statements [79]. There are also books in this area: *The Art of SQL* [80] and *Refactoring SQL Applications* [81] provide guidelines for writing efficient queries, while the book of Bill Karwin [1] collects antipatterns.

In the realm of embedded SQL, Christensen *et al.* proposed a tool (JSA, Java String Analyzer) to extract string expressions from Java code statically [25]. They also check the syntax of the extracted SQL strings. Wassermann *et al.* proposed a static string analysis technique to identify possible errors in dynamically generated SQL code [27]. They detect type errors (e.g., concatenating a character to an integer value) in extracted query strings of valid SQL syntax. In a tool demo paper, they present their prototype tool called JDBC Checker [82]. Anderson and Hills studied query construction patterns in PHP [35]. They analyzed query strings embedded in PHP code with the help of the PHP AiR framework.

Brink *et al.* proposed a quality assessment of embedded SQL [29]. They analyze query strings embedded in PL/SQL, Cobol, and Visual Basic programs [83]. Many static techniques deal with embedded queries for SQL injection detection [84]. Their goal is to determine whether a query could be affected by user input. Yeole and Meshram published a survey of these techniques [85]. Marashdeh *et al.* also surveyed the challenges of detecting SQL Injection vulnerabilities [86].

Some papers also tackle SQL fault localization techniques. Clark *et al.* proposed a dynamic approach to localize SQL faults in database applications [87]. They provide command-SQL tuples to show the SQL statements executed at database interaction points. Delplanque *et al.* assessed schema quality and detected design smells [88]. Their tool, DBCritics, analyze PostgreSQL schema dumps and identify design smells such as missing primary keys or foreign key references. Alvor by Annamaa *et al.* can analyze string expressions in Java code [30]. It checks syntax correctness, semantics correctness, and object availability by comparing the extracted queries against its internal SQL grammar and by checking SQL statements against an actual database.

Visualization. Since the seminal works of Reiss [89] and Young & Munro [90], many studied 3D approaches to visualize software systems. The software as cities metaphor has been widely explored and led to diverse implementations, such as the Software World approach by Knight *et al.* [91], the visualization of communicating architectures by Panas *et al.* [92, 93], Verso by Langelier *et al.* [94], CodeCity by Wettel *et al.* [75, 95], EVO-STREETS by Steinbrückner & Lewerentz [96], CodeMetropolis by Balogh & Beszedes [97], and VR City by Vincur *et al.* [98].

Some approaches considered presenting the databases together with the source code, and interestingly, most use the city metaphor such as DAHLIA [7, 74] and M3TRICITY [50, 51]. Zirkelbach and Hasselbring presented RACCOON [99], a visualization approach of database behavior, which uses the 3D city metaphor to show the structure of a database based on the concepts of entity-relationship diagrams. Marinescu presented for enterprise systems a meta-model containing object-oriented entities, relational entities and object-relational interactions [100].

4 Empirical Studies

4.1 Introduction

In this section, we discuss recent empirical studies relying on large collections of data-intensive software systems, focusing on (1) the (joint) use of database models and technologies, (2) the prevalence and impact of SQL bad smells, (3) the challenges and best practices when testing the database manipulation code and (4) the presence of self-admitted technical debt in this code.

4.2 The (Joint) Use of Data Models and Technologies

In the last decade, non-relational database technologies (*e.g.,* graph databases, document stores, key-value, column-oriented) have emerged for specialized purposes. The joint use of database models has increased in popularity since there are benefits of such *multi-database* architectures where developers combine various technologies. However, the side effects on design, querying, and maintenance are not well-known.

Benats *et al.* [5] conducted an empirical study of the use of (multi-)database models in open-source database-dependent projects. They mined four years of development history (2017–2020) of a total number of 33 million projects by leveraging Libraries.io.[16] They identified projects relying on one or several databases and written in popular programming languages (1.3 million projects). After applying filters to eliminate "low-quality" repositories and remove project duplicates, they ultimately gathered a dataset of 40,609 projects. They analyzed the dependencies of those projects to assess (1) the popularity of the different database models, (2) the extent that they are combined within the same systems, and (3) how their usage evolved. They found that most current database-dependent projects (54.72%) rely on a relational database model, while NoSQL-dependent systems represent 45.28% of the projects. However, the popularity of SQL technologies has recently decreased concerning NoSQL datastores.

Regarding programming languages, the authors noticed that Ruby and Python systems are often paired with a PostgreSQL database. At the same time, Java and C# projects typically rely on a MySQL database. Data-intensive systems in JavaScript/TypeScript are essentially paired with document-oriented or key-value databases

The study results confirm the emergence of *hybrid data-intensive systems*. The authors found the joint use of different database models (e.g., relational and non-relational) in 16% of all database-dependent projects. In particular, they found that more than 56% of systems relying on a key-value database also use another technology, typically relational or document-oriented. Wide-column dependent systems follow the same pattern, with over 47% being hybrid. This shows the complimentary usage of SQL and NoSQL in practice.

[16] https://libraries.io/

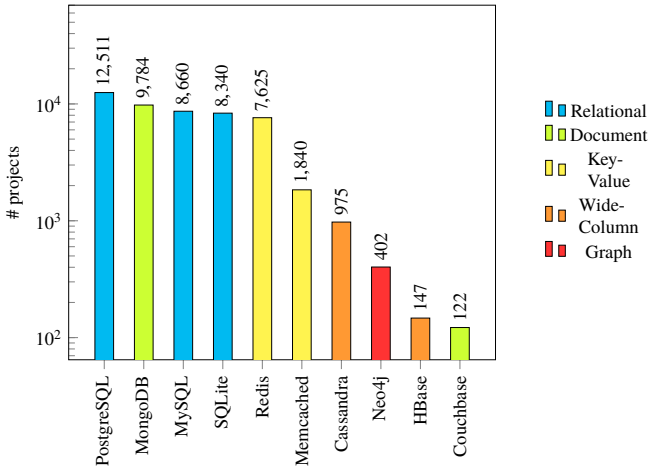

Fig. 14 Usage of database management systems (2020)

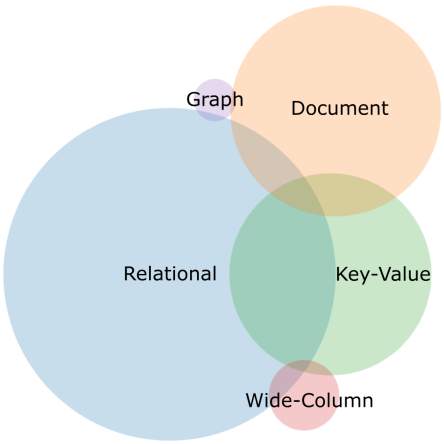

Fig. 15 Distribution of hybrid database-dependent projects

The authors then examined the evolution of these systems to identify typical transitions in terms of database models or technologies. They observed that only one percent of the database-dependent projects evolved their data model over time. The majority (62%) were not initially hybrid but once relied on a single database model. In contrast, 19% of those projects became "mono-database" after initially using multiple database models.

4.3 Prevalence, Impact and Evolution of SQL Bad Smells

Muse *et al.* [46] investigated the prevalence and evolution of SQL code smells in data-intensive open-source systems. Their study relies on the analysis of 150 open-source software systems that manipulate their databases through popular database access APIs—Android Database API, JDBC, JPA and Hibernate. The authors analysed the source code of each project and studied 19 traditional code smells using the DECOR tool [101] and 4 SQL code smells using SQLInspect [2]. They also collected bug-fixing and bug-inducing commits from each project using PyDriller [102].

They first studied the prevalence of SQL code smells in the selected software systems by categorizing them into four application domains—Business, Library, Multimedia, and Utility. They found that SQL code smells are prevalent in all four domains, some SQL code smells being more prevalent than others. Then, they investigated the co-occurrence of SQL code smells and traditional code smells using association rule mining. The results show that while some SQL code smells have statistically significant co-occurrence with traditional code smells, the degree of association is low. Third, they investigated the potential impact of SQL code smells on software bugs by analysing their co-occurrences within the bug-inducing commits. They performed Cramer's V test of association and built a random forest model to study the impact of the smells on bugs. The analysis results indicate a weak association between SQL code smells and software bugs. Some SQL code smells tend to show a higher association with bugs compared to others. Finally, the authors performed a survival analysis of SQL and traditional code smells using Kaplan-Meier survival curves to compare their survival time. They found that SQL code smells survive longer than traditional code smells. A large fraction of the source files affected by SQL code smells (80.5%) persist throughout the whole snapshots, and they hardly get any attention from the developers during refactoring. Furthermore, significant portions of the SQL code smells are created at the very beginning and persist in all subsequent versions of the systems.

The study shows that SQL code smells persist in the studied data-intensive software systems. Developers should be aware of these smells and consider detecting and refactoring SQL and traditional code smells separately, using dedicated tools.

4.4 Database Code Testing (Best) Practices

Software testing enables development teams to maintain the quality of an evolving software system. The database manipulation code is often neglected in this context, although it requires special attention.

Gobert *et al.* [48] conducted an empirical study on the challenges and perils of testing database manipulation code. They first mined open-source systems from Libraries.io and analysed 6,622 projects that relied on database access technologies. They found automated tests and database manipulation code in only 332 projects.

Then, they further examined the 72 projects for which they could execute the tests and collect coverage reports.

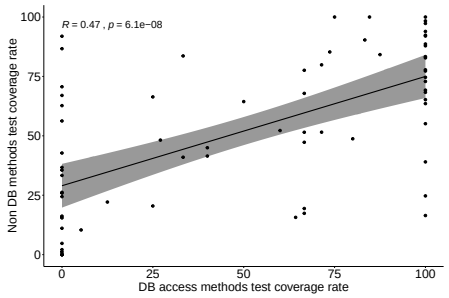

Fig. 16 Test coverage rates of Non-DB access methods vs. DB access methods

Fig. 16 shows a scatter plot of all projects analyzed and their respective test coverage rates. The results indicate that the *database manipulation code was poorly tested*: 46% of the projects did not test half of their DB methods, and 33% did not test DB communication. A significant number of projects with the highest coverage rate had, in fact, full coverage. The authors found a mean value of 2.8 database methods for projects with full coverage. There are slightly fewer projects (48.6%) in the figure with lower coverage for database methods. However, considering only the projects above the median (*i.e.,* with at least five database methods), there is a more significant difference: 59% have a smaller coverage for database methods than regular methods. Similarly, while 46% of the projects cover less than half of their database methods, this number increases to 53% for projects above the median.

Such a poor test coverage motivated the authors to understand the reasons holding back the developers from testing database access code. They qualitatively analysed 532 selected questions from popular Stack Exchange websites and identified the *problems* that hampered developers in writing tests. They distilled the results in a *taxonomy of issues* with 83 different problems grouped into 7 main categories. They found that developers mostly look for insights on general best practices to test database access code. They also have more technical questions related to DB handling, mocking, parallelisation or framework/tool usage.

In a follow-up study, the same authors examined the *answers* to these questions [49]. They manually labelled the top three highest-ranked answers to each question and built a *taxonomy of best practices*. Overall, they examined 598 answers to 255 questions and listed 363 different practices in the taxonomy.

The category in the taxonomy with the highest number of tags and questions was related to the testing environment, *e.g.,* proposed various tools and configurations. The second most exhaustive category was database management, *e.g.,* initialising or cleaning up a database between tests. Other categories included code structure or design guidelines, concepts, performance, processes, test characteristics, test code, and mocking.

Most suggestions considered the testing environment and recommended various tools or configurations. The second largest category was database management, where many addressed database initialisation and clean-up between tests. Other categories pertained to code structure or design, concepts, performance, processes, test characteristics, test code, and mocking. The authors illustrated the two taxonomies through intriguing examples.

4.5 Self-admitted Technical Debt in Database Access Code

Developers sometimes choose design and implementation shortcuts due to the pressure from tight release schedules. However, shortcuts introduce technical debt that increases as the software evolves. The debt needs to be repaid as quickly as possible to minimize its impact on software development and quality. Sometimes, technical debt is admitted by developers in comments and commit messages. Such debt is known as self-admitted technical debt (SATD).

In data-intensive systems, where data manipulation is a critical functionality, the presence of SATD in the data access logic could seriously harm performance and maintainability. Understanding the composition and distribution of the SATDs across software systems and their evolution could provide insights into managing technical debt efficiently.

Muse *et al.* [47] conducted a large-scale empirical study on the composition and distribution of SATD across data-intensive software systems and their evolution, providing insights into the prevalence, composition, and evolution of SATD. The authors analyzed 83 open-source systems relying on relational databases and 19 systems relying on NoSQL databases. They detected SATD in source code comments obtained from different snapshots of the subject systems. Then, they conducted a survival analysis to understand the evolutionary dynamics of SATDs.

Next, they conducted a manual analysis of 361 sample data-access SATDs, investigating the composition of data-access SATDs and the reasons behind their introduction and removal. They identified 15 new SATD categories, out of which 11 are specific to database access operations. They found that most of the data-access SATDs are introduced in the later stages of change history rather than at the beginning. They also discovered that bug fixing and refactoring are the main reasons behind the introduction of data-access SATDs.

4.6 Reflections

Studies on code quality. Other researchers studied frequent errors and antipatterns in SQL queries. The book of Karwin [1] is the first to present SQL antipatterns in a comprehensive catalogue. Khumnin *et al.* [103] present a tool for detecting logical database design antipatterns in Transact-SQL queries.

Another tool, *DbDeo* [104], implements the detection of *database schema* smells. *DbDeo* has been evaluated on 2925 open-source repositories; their authors identified 13 different types of smells, among which "index abuse" was the most prevalent. De Almeida Filho *et al.* [105] investigate the prevalence and co-occurrence of SQL code smells in PL/SQL projects. Arzamasova *et al.* propose to detect antipatterns in SQL logs [106] and demonstrate their approach by refactoring a project containing more than 40 million queries. Shao *et al.* [107] identified a list of database-access performance antipatterns, mainly in PHP web applications. Integrity violation was addressed by Li *et al.* [108], who identified constraints from source code and related them to database attributes.

Studies on evolution. Several other authors studied how data-intensive systems relying on a relational database evolve. Curino *et al.* [54] present a study of the structural evolution of the Wikipedia database, intending to extract both a micro-classification and a macro-classification of schema changes. They also study the frequency distribution of those schema changes. In addition to a schema evolution statistics extractor, the authors propose a tool that operates on the differences between subsequent schema versions and semi-automatically extracts the set of possible schema changes that have been applied. In this study, a period of four years has been considered, corresponding to 171 successive versions of the Wikipedia database schema. Their study shows the need for automated support for schema evolution. Vassiliadis *et al.* [55, 109] studied the evolution of individual database tables over time in eight different software systems. They report on their observations on how evolution-related properties, like the possibility of deletion, or the updates a table undergoes, are related to observable table properties like the number of attributes or the time of birth of a table. Through a large-scale study on the evolution of databases, they also tried to determine whether Lehman's laws of software evolution hold for evolving database schemas [110]. They conclude that the essence of Lehman's laws remains valid in this context, but that specific mechanics significantly differ regarding schema evolution. Dimolikas *et al.* [111] studied the evolution of tables in a relational schema over time concerning the structure of the foreign keys to which tables are related. Their study considers the evolution history of six database schemas, and reveals the update behavior of tables depend on their topological complexity. Cleve *et al.* [112] show that mining database schema evolution can have a significant informative value in the context of reverse engineering. They introduce the *global historical schema*, an aggregated schema of all previous versions of a database schema. They then analyse this schema to better understand the current version of the database schema and, therefore, facilitate future schema changes. Lin *et al.* [113] study the so-called *collateral* evolution of applications and databases, in which the evolution of an application is separated from the evolution of its persistent data, or the database. They investigated how application programs and database management systems in popular open-source systems (Mozilla, Monotone) cope with database schema and format changes. They observed that collateral evolution could lead to potential problems. The number of schema changes reported is minimal. In Mozilla, 20 table creations and 4 table deletions are reported in 4 years. During 6 years of Monotone schema evolution, only 9 tables were added while 8 tables were deleted.

Qiu *et al.* [57] conducted a large-scale empirical study on ten popular database applications from various domains to analyze how schemas and application code co-evolve. In particular, they study the evolution histories from the respective repositories to understand whether database schemas evolve frequently and significantly, and how schemas evolve and impact the application code. In their approach, the authors try to estimate the impact of a database schema change in the code. This estimation is performed with a simple difference extractor calculating changed source lines between two versions. Goeminne *et al.* [114] study the co-evolution between code-related and database-related activities in data-intensive systems combining several ways to access the database (native SQL queries and Object-Relational Mapping). They empirically analyzed the evolution of SQL, Hibernate and JPA usage in a large and complex open-source information system. Interestingly, they observed that using embedded SQL queries is still common today.

Other studies exclusively focus on NoSQL applications. Störl *et al.* [12] investigated the advantages of using object mapper libraries when accessing NoSQL data stores. They overview Object-NoSQL Mappers (ONMs) and Object-Relational Mappers with NoSQL support. As they say, building applications against the native interfaces of NoSQL data stores create technical lock-in due to the lack of standardized query languages. Therefore, developers often turn to object mapper libraries as an extra level of abstraction. Scherzinger *et al.* [4] studied how software engineers design and evolve their domain model when building NoSQL applications, by analyzing the denormalized character of ten open-source Java applications relying on object mappers. They observed the growth in complexity of the NoSQL schemas and common evolution operations between the projects. The study also shows that software releases include considerably more schema-relevant changes: >30% compared to 2% with relational databases. Ringlstetter *et al.* [115] examined how NoSQL object-mappers evolution annotations were used. They found that only 5.6% of 900 open-source Java projects using Morphia or Objectify used such annotations to evolve the data model or migrate the data.

Studies on technical debt. Other empirical studies are related to technical debt in data-intensive systems. Albarak and Bashoon [116] propose a taxonomy of debts related to the conceptual, logical, and physical design of a database. For example, they claim that ill-normalized databases (*i.e.,* databases with tables below the fourth normal form) can also be considered technical debt [117]. To tackle this specific type of debt, they propose an approach to prioritize tables that should be normalized. Foidl *et al.* claim that technical debt can be incurred in different parts (*i.e.,* software systems, data storage systems, data) of data-intensive systems and different parts can further affect each other [118]. They propose a conceptual model to outline where technical debt can emerge in data-intensive systems by separating them into three parts: software systems, data storage systems and data. They present two smells as examples. Missing constraints, when referential integrity constraints are not declared in a database schema; and metadata as data, when an entity-attribute-value pattern is used to store metadata (attributes) as data.

Weber *et al.* [119] also identified relational database schemas as potential sources of technical debt. In particular, they provided a first attempt at utilizing the technical debt analogy for developing processes related to the missing implementation of implicit foreign key (FK) constraints. They discuss the detection of missing FKs, propose a measurement for the associated TD, and outline a process for reducing FK-related TD. As an illustrative case study, they consider OSCAR, a large Java medical record system used in Canada's primary health care. Ramasubbu and Kemerer [120] empirically analyze the impact of technical debt on system reliability by observing a 10-year life cycle of a commercial enterprise system. They also examine the relative effects of modular and architectural maintenance activities in clients. They conclude that technical debt decreases the reliability of enterprise systems. They also add that modular maintenance targeted to reduce technical debt is about 53% more effective than architectural maintenance in reducing the probability of a system failure due to client errors.

5 Conclusion

This chapter summarized the recent research efforts devoted to mining, analyzing and evolving data-intensive software ecosystems.

We have argued that (1) both the databases and the programs are essential ecosystem artifacts; (2) mining, analyzing and visualizing what the programs are doing on the data may considerably help in understanding the system in general, and the databases in particular, (3) database interactions may suffer from quality problems and technical debt and should be better tested and (4) software evolution methods should devote more attention to the program-database co-evolution problem.

The research community still faces many challenges in the near future, given the increasing complexity and heterogeneity of data-intensive software ecosystems. To fully embrace the *DevOps* movement, developers need better support for database-related evolutions *at runtime* [121]. This is the case, for instance, when developing microservices applications deployed on distributed computing architectures such as the *cloud-edge continuum* [122].

References

1. B. Karwin, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Programmers, 2010.
2. C. Nagy and A. Cleve, "Sqlinspect: A static analyzer to inspect database usage in java applications," in *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE 2018)*, 2018, pp. 93–96.
3. L. Meurice and A. Cleve, "Supporting schema evolution in schema-less NoSQL data stores," in *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017)*, 2017, pp. 457–461.
4. S. Scherzinger and S. Sidortschuck, "An empirical study on the design and evolution of NoSQL database schemas," in *Conceptual Modeling (ER 2020)*, G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, and H. C. Mayr, Eds. Springer, 2020, pp. 441–455.
5. P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "An empirical study of (multi-) database models in open-source projects," in *Proceedings of the 40th International Conference on Conceptual Modeling (ER 2021)*. Springer, 2021, pp. 87–101.
6. M. Goeminne and T. Mens, "Towards a survival analysis of database framework usage in java projects," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME 2015)*. IEEE Comp. Soc., 2015, pp. 551–555.
7. L. Meurice and A. Cleve, "DAHLIA: A visual analyzer of database schema evolution," in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 464–468.
8. L. Meurice, M. Goeminne, T. Mens, C. Nagy, A. Decan, and A. Cleve, *Analyzing the Evolution of Database Usage in Data-Intensive Software Systems*. John Wiley & Sons, 2018, pp. 208–240.
9. McKnight, "NoSQL Evaluator's Guide," 2014. [Online]. Available: https://www.mcknightcg.com/nosql-evaluators-guide/
10. S. Scherzinger, E. C. De Almeida, F. Ickert, and M. D. Del Fabro, "On the necessity of model checking NoSQL database schemas when building SaaS applications," in *Int. Workshop on Testing the Cloud (TTC 2013)*. ACM, 2013.
11. K. W. Alger and D. Coupal, "Building with patterns: The polymorphic pattern," 2022, (accessed: 23.09.2022). [Online]. Available: https://www.mongodb.com/developer/how-to/polymorphic-pattern/
12. U. Störl, M. Klettke, and S. Scherzinger, "NoSQL schema evolution and data migration: State-of-the-art and opportunities," in *23rd Int. Conf. Extending Database Technology, (EDBT 2020)*, 2020, pp. 655–658.
13. F. Abdelhedi, A. Brahim, H. Rajhi, R. Ferhat, and G. Zurfluh, "Automatic extraction of a document-oriented NoSQL schema," in *23rd Int. Conf. Enterprise Information Systems*, 2021.
14. P. Gómez, R. Casallas, and C. Roncancio, "Automatic schema generation for document-oriented systems," in *Database and Expert Systems Applications*. Springer, 2020, pp. 152–163.
15. M. J. Mior, "Automated schema design for NoSQL databases," in *2014 SIGMOD PhD Symposium*. ACM, 2014, pp. 41–45.
16. A. A. Imam, S. Basri, R. Ahmad, J. Watada, and M. T. González-Aparicio, "Automatic schema suggestion model for NoSQL document-stores databases," *Journal of Big Data*, vol. 5, 2018.
17. B. Cherry, P. Benats, M. Gobert, L. Meurice, C. Nagy, and A. Cleve, "Static analysis of database accesses in mongodb applications," in *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2022)*. IEEE Comp. Soc., 2022, pp. 930–934.
18. V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "JSAI: A static analysis platform for JavaScript," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 121–132.

19. E. Andreasen and A. Møller, "Determinacy in static analysis for jquery," in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2014)*, 2014, pp. 17–31.
20. K. Sun and S. Ryu, "Analysis of JavaScript programs: Challenges and research trends," *ACM Comput. Surv.*, vol. 50, no. 4, Aug. 2017.
21. S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Static Analysis*. Springer, 2009, pp. 238–255.
22. M. Madsen and A. Møller, "Sparse dataflow analysis with pointers and reachability," in *Static Analysis*. Springer, 2014, pp. 201–218.
23. A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for JavaScript IDE services," in *2013 Int. Conf. Software Engineering (ICSE 2013)*. IEEE, 2013, pp. 752–761.
24. A. Afonso, A. da Silva, T. Conte, P. Martins, J. Cavalcanti, and A. Garcia, "LESSQL: dealing with database schema changes in continuous deployment," in *IEEE 27th Int. Conf. Software Analysis, Evolution and Reengineering (SANER 2020)*, 2020, pp. 138–148.
25. A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proceedings of the 10th International Conference on Static Analysis (SAS 2003)*. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 1–18.
26. C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, 2004, pp. 645–654.
27. G. Wassermann, C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, p. 14, Sep. 2007.
28. A. Maule, W. Emmerich, and D. Rosenblum, "Impact analysis of database schema changes," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 451–460.
29. H. van den Brink, R. van der Leek, and J. Visser, "Quality assessment for embedded sql," in *7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, 2007, pp. 163–170.
30. A. Annamaa, A. Breslav, J. Kabanov, and V. Vene, "An interactive tool for analyzing embedded sql queries," in *Programming Languages and Systems*, K. Ueda, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 131–138.
31. L. Meurice, C. Nagy, and A. Cleve, "Static analysis of dynamic database usage in java systems," in *Proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016)*. Springer LNCS, 2016, pp. 491–506.
32. M. N. Ngo and H. B. K. Tan, "Applying static analysis for automated extraction of database interactions in web applications," *Information and Software Technology*, vol. 50, no. 3, pp. 160–175, 2008.
33. D. Anderson and M. Hills, "Query construction patterns in php," in *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017)*, 2017, pp. 452–456.
34. ——, "Supporting analysis of sql queries in php air," in *Proceedings of the 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM 2017)*, 2017, pp. 153–158.
35. D. Anderson, "Modeling and analysis of sql queries in php systems," Master's thesis, East Carolina University, Apr. 2018. [Online]. Available: http://hdl.handle.net/10342/6743
36. P. Manousis, A. Zarras, P. Vassiliadis, and G. Papastefanatos, "Extraction of embedded queries via static analysis of host code," in *Advanced Information Systems Engineering*, E. Dubois and K. Pohl, Eds. Springer, 2017, pp. 511–526.
37. Y. Lyu, J. Gui, M. Wan, and W. G. J. Halfond, "An empirical study of local database usage in android applications," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*, 2017, pp. 444–455.

38. D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, "String analysis for java and android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 2015, pp. 661–672.
39. A. Cleve and J.-L. Hainaut, "Dynamic analysis of sql statements for data-intensive applications reverse engineering," in *2008 15th Working Conference on Reverse Engineering*, 2008, pp. 192–196.
40. M. Mori, N. Noughi, and A. Cleve, "Mining sql execution traces for data manipulation behavior recovery," in *CAiSE*, 2014.
41. N. Noughi, M. Mori, L. Meurice, and A. Cleve, "Understanding the database manipulation behavior of programs," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014)*. ACM, 2014, pp. 64–67.
42. A. A. Brahim, R. T. Ferhat, and G. Zurfluh, "Model driven extraction of NoSQL databases schema: Case of MongodB," in *11th Int. Joint Conf. on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, 2019, pp. 145–154.
43. E. Gallinucci, M. Golfarelli, and S. Rizzi, "Schema profiling of document-oriented databases," *Inf. Systems*, vol. 75, pp. 13–25, 2018.
44. M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani, "Parametric schema inference for massive JSON datasets," *The VLDB Journal*, vol. 28, no. 4, pp. 497–521, 2019.
45. C. Nagy and A. Cleve, "Static code smell detection in sql queries embedded in java code," in *Proceedings of the 17th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2017)*. IEEE Computer Society, 2017, pp. 147–152.
46. B. A. Muse, M. Rahman, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "On the prevalence, impact, and evolution of SQL code smells in data-intensive systems," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR 2020)*. ACM, 2020, pp. 327–338.
47. B. A. Muse, C. Nagy, A. Cleve, F. Khomh, and G. Antoniol, "Fixme: Synchronize with database an empirical study of data access self-admitted technical debt," *Empirical Software Engineering*, vol. 27, no. 6, 2022.
48. M. Gobert, C. Nagy, H. Rocha, S. Demeyer, and A. Cleve, "Challenges and perils of testing database manipulation code," in *Proceedings of the 33rd International Conference on Advanced Information Systems Engineering (CAiSE 2021)*. Springer International Publishing, 2021, pp. 229–245.
49. ——, "Best practices of testing database manipulation code," *Information Systems*, vol. 111, p. 102105, 2023.
50. S. Ardigò, C. Nagy, R. Minelli, and M. Lanza, "Visualizing data in software cities," in *Proceedings of the 9th IEEE Working Conference on Software Visualization (VISSOFT 2021), NIER/TD*, 2021, pp. 145–149.
51. ——, "M3tricity: Visualizing evolving software & data cities," in *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*. IEEE, 2022, pp. 130–133.
52. M. M. Lehman, "Laws of software evolution revisited," in *Software Process Technology*, C. Montangero, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 108–124.
53. D. Sjøberg, "Quantifying schema evolution," *Information and Software Technology*, vol. 35, no. 1, pp. 35–44, 1993.
54. C. A. Curino, L. Tanca, H. J. Moon, and C. Zaniolo, "Schema evolution in Wikipedia: toward a web information system benchmark," in *Proceedings of the 2008 International Conference on Enterprise Information Systems (ICEIS 2008)*, 2008.
55. P. Vassiliadis, A. V. Zarras, and I. Skoulis, "How is life for a table in an evolving relational schema? birth, death and everything in between," in *Conceptual Modeling*, P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, and Ó. Pastor López, Eds. Springer, 2015, pp. 453–466.
56. T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 2014, pp. 1001–1012.

57. 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 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, 2013, pp. 125–135.
58. M. Stonebraker, D. Deng, and M. L. Brodie, "Database decay and how to avoid it," in *Proc. Big Data*, 2016, pp. 7–16.
59. M. Stonebraker, D. Deng, and M. L. Brodie, "Application-database co-evolution: A new design and development paradigm," in *New England Database Day*, 2017.
60. L. Meurice, C. Nagy, and A. Cleve, "Detecting and preventing program inconsistencies under database schema evolution," in *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability & Security (QRS 2016)*. IEEE Comp. Soc., 2016, pp. 262–273.
61. J. Stasko, J. Domingue, M. Brown, and B. P. (Eds.), *Software Visualization - Programming as a Multimedia Experience*. The MIT Press, 1998.
62. S. K. Card, J. D. Mackinlay, and B. Shneiderman, Eds., *Readings in Information Visualization — Using Vision to Think*. Morgan Kaufmann, 1999.
63. J. Bertin, *Graphische Semiologie*, 2nd ed. Walter de Gruyter, 1974.
64. E. Tufte, *Envisioning Information*. Graphics Press, 1990.
65. ——, *Visual Explanations*. Graphics Press, 1997.
66. ——, *The Visual Display of Quantitative Information*, 2nd ed. Graphics Press, 2001.
67. C. Ware, *Information Visualization: Perception for Design*, 2nd ed. Morgan Kaufmann, 2004.
68. M.-A. D. Storey, C. Best, and J. Michaud, "SHriMP Views: An interactive and customizable environment for software exploration," in *Proceedings of International Workshop on Program Comprehension (IWPC 2001)*, 2001.
69. A. Marcus, L. Feng, and J. I. Maletic, "3D representations for software visualization," in *Proceedings of the ACM Symposium on Software Visualization*. IEEE, 2003, p. 27.
70. D. Beyer and C. Lewerentz, "CrocoPat: A tool for efficient pattern recognition in large object-oriented programs," Institute of Computer Science, Brandenburgische Technische Universität Cottbus, Tech. Rep. I-04/2003, Jan. 2003.
71. J.-M. Favre, "Gsee: a generic software exploration environment," in *Proceedings of the 9th International Workshop on Program Comprehension*. IEEE, May 2001, pp. 233–244.
72. M.-A. D. Storey, K. Wong, and H. A. Müller, "How do program understanding tools affect how programmers understand programs?" in *Proceedings Fourth Working Conference on Reverse Engineering*, I. Baxter, A. Quilici, and C. Verhoef, Eds. IEEE Computer Society, 1997, pp. 12–21.
73. C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A classification of object-relational impedance mismatch," in *1st International Confernce on Advances in Databases, Knowledge, and Data Applications*, 2009, pp. 36–43.
74. L. Meurice and A. Cleve, "DAHLIA 2.0: A visual analyzer of database usage in dynamic and heterogeneous systems," in *Proceedings of the 2016 IEEE Working Conference on Software Visualization (VISSOFT 2016)*, 2016, pp. 76–80.
75. R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*. IEEE CS Press, 2007, pp. 92–99.
76. F. Pfahler, R. Minelli, C. Nagy, and M. Lanza, "Visualizing evolving software cities," in *Proceedings of the 2020 Working Conference on Software Visualization (VISSOFT 2020)*, 2020, pp. 22–26.
77. S. Brass and C. Goldberg, "Detecting logical errors in SQL queries," in *Proceedings of the 16th Workshop on Foundations of Databases*, 2004.
78. ——, "Semantic errors in SQL queries: A quite complete list," *Journal of Systems and Software*, vol. 79, no. 5, pp. 630–644, 2006, quality Software.
79. C. Goldberg, "Do you know SQL? about semantic errors in database queries," Higher Education Academy, Tech. Rep., 2008.
80. S. Faroult and P. Robson, *The Art of SQL*. O'Reilly Media, 2006.
81. S. Faroult and P. L'Hermite, *Refactoring SQL Applications*. O'Reilly Media, 2008.

82. C. Gould, Z. Su, and P. Devanbu, "JDBC Checker: A static analysis tool for SQL/JDBC applications," in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, 2004, pp. 697–698.
83. H. J. Van Den Brink and R. van der Leek, "Quality metrics for SQL queries embedded in host languages," in *11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, 2007.
84. 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 *Proceedings of the 4th International Conference on Security of Information and Networks (SIN 2011)*. ACM, 2011, pp. 81–86.
85. A. S. Yeole and B. B. Meshram, "Analysis of different technique for detection of SQL injection," in *Proceedings of the International Conference & Workshop on Emerging Trends in Technology (ICWET 2011)*. ACM, 2011, pp. 963–966.
86. Z. Marashdeh, K. Suwais, and M. Alia, "A survey on sql injection attack: Detection and challenges," in *Proceedings of the 2021 International Conference on Information Technology (ICIT 2021)*, 2021, pp. 957–962.
87. S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold, "Localizing SQL faults in database applications," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 213–222.
88. J. Delplanque, A. Etien, O. Auverlot, T. Mens, N. Anquetil, and S. Ducasse, "Codecritics applied to database schema: Challenges and first results," in *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017)*, 2017, pp. 432–436.
89. S. P. Reiss, "An engine for the 3D visualization of program information," *J. Visual Languages & Computing*, vol. 6, no. 3, pp. 299–323, 1995.
90. P. Young and M. Munro, "Visualising software in virtual reality," in *Proc. 6th Int. Workshop on Program Comprehension*. IEEE, 1998, pp. 19–26.
91. C. Knight and M. C. Munro, "Virtual but visible software," in *Proc. 17th Int. Conf. Information Visualization (IV 2000)*. IEEE CS Press, 2000, pp. 198–205.
92. T. Panas, R. Berrigan, and J. Grundy, "A 3D metaphor for software production visualization," in *Proceedings of IV 2003*. IEEE CS Press, 2003, p. 314.
93. T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc, "Communicating software architecture using a unified single-view visualization," in *Proc. 12th Int. Conf. Engineering Complex Computer Systems*. IEEE, 2007, pp. 217–228.
94. G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proc. 20th Int. Conf. Automated Software Engineering*. ACM, 2005, pp. 214–223.
95. R. Wettel and M. Lanza, "CodeCity: 3D visualization of large-scale software," in *Comp. 30th Int. Conf. Software Engineering (ICSE 2008)*. ACM, 2008, pp. 921–922.
96. F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *Proc. 5th Int. Symposium on Software Visualization*. ACM, 2010, pp. 193–202.
97. G. Balogh and A. Beszedes, "CodeMetropolis - code visualisation in MineCraft," in *Proc. 13th Int. Working Conf. Source Code Analysis and Manipulation*. IEEE, 2013, pp. 136–141.
98. J. Vincur, P. Navrat, and I. Polasek, "VR City: Software analysis in virtual reality environment," in *Proc. Int. Conf. Software Quality, Reliability and Security Companion*. IEEE, Jul. 2017, pp. 509–516.
99. C. Zirkelbach and W. Hasselbring, "Live visualization of database behavior for large software landscapes: The RACCOON approach," Department of Computer Science, Kiel University, Tech. Rep., 2019.
100. C. Marinescu, "Applications of automated model's extraction in enterprise systems," in *Proc. 14th Int. Conf. Software Technologies (ICSOFT 2019)*. SCITEPRESS, 2019, pp. 254–261.
101. Y.-G. Guéhéneuc, "Ptidej: A flexible reverse engineering tool suite," in *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 529–530.

102. D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proc of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 908–911.
103. P. Khumnin and T. Senivongse, "SQL antipatterns detection and database refactoring process," in *Proc. of SNPD 2017*, 2017, pp. 199–205.
104. 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)*. ACM, 2018, pp. 55–64.
105. F. G. de Almeida Filho, A. D. F. Martins, T. d. S. Vinuto, J. M. Monteiro, Í. P. de Sousa, J. de Castro Machado, and L. S. Rocha, "Prevalence of bad smells in PL/SQL projects," in *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 2019, pp. 116–121.
106. N. Arzamasova, M. Schäler, and K. Böhm, "Cleaning antipatterns in an SQL query log," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 3, pp. 421–434, Mar. 2018.
107. S. Shao, Z. Qiu, X. Yu, W. Yang, G. Jin, T. Xie, and X. Wu, "Database-access performance antipatterns in database-backed web applications," in *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)*. IEEE, 2020, pp. 58–69.
108. B. Li, D. Poshyvanyk, and M. Grechanik, "Automatically detecting integrity violations in database-centric applications," in *Proceedings of the 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC 2017)*, 2017, pp. 251–262.
109. P. Vassiliadis, A. V. Zarras, and I. Skoulis, "Gravitating to rigidity: Patterns of schema evolution, and its absence in the lives of tables," *Information Systems*, vol. 63, pp. 24–46, 2017.
110. I. Skoulis, P. Vassiliadis, and A. Zarras, "Open-source databases: Within, outside, or beyond Lehman's laws of software evolution?" in *CAISE '14*, ser. LNCS, vol. 8484. Springer, 2014, pp. 379–393.
111. K. Dimolikas, A. V. Zarras, and P. Vassiliadis, "A study on the effect of a table's involvement in foreign keys to its schema evolution," in *Conceptual Modeling*, G. Dobbie, U. Frank, G. Kappel, S. W. Liddle, and H. C. Mayr, Eds. Springer, 2020, pp. 456–470.
112. A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. Weber, "Understanding database schema evolution: A case study," *Science of Computer Programming*, vol. 97, pp. 113–121, 2015, special Issue on New Ideas and Emerging Results in Understanding Software.
113. D.-Y. Lin and I. Neamtiu, "Collateral evolution of applications and databases," in *Joint Int'l Workshop on Principles of software evolution and ERCIM software evolution workshop*. ACM, 2009, pp. 31–40.
114. M. Goeminne, A. Decan, and T. Mens, "Co-evolving code-related and database-related changes in a data-intensive software system," in *Proc. CSMR/WCRE*, 2014.
115. A. Ringlstetter, S. Scherzinger, and T. F. Bissyandé, "Data model evolution using object-NoSQL mappers: Folklore or state-of-the-art?" in *2nd International Workshop on BIG Data Software Engineering*, 2016, pp. 33–36.
116. M. Al-Barak and R. Bahsoon, "Database design debts through examining schema evolution," in *IEEE 8th International Workshop on Managing Technical Debt (MTD 2016)*, 2016, pp. 17–23.
117. M. Albarak and R. Bahsoon, "Prioritizing technical debt in database normalization using portfolio theory and data quality metrics," in *Proceedings of the 2018 International Conference on Technical Debt (TechDebt 2018)*. ACM, 2018, pp. 31–40.
118. H. Foidl, M. Felderer, and S. Biffl, "Technical debt in data-intensive software systems," in *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2019)*, 2019, pp. 338–341.
119. J. H. Weber, A. Cleve, L. Meurice, and F. J. Bermudez Ruiz, "Managing technical debt in database schemas of critical software," in *2014 Sixth International Workshop on Managing Technical Debt*, 2014, pp. 43–46.

120. N. Ramasubbu and C. F. Kemerer, "Technical debt and the reliability of enterprise software systems: A competing risks analysis," *Management Science*, vol. 62, no. 5, pp. 1487–1510, 2016.
121. M. de Jong, A. van Deursen, and A. Cleve, "Zero-downtime sql database schema evolution for continuous deployment," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE Press, 2017, pp. 143–152.
122. D. Milojicic, "The edge-to-cloud continuum," *IEEE Annals of the History of Computing*, vol. 53, pp. 16–25, 2020.