Chapter 11
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.
This chapter elaborates on the recent research effort devoted to mining, analyzing, and evolving data-intensive software ecosystems. It focuses on methods, techniques, and tools providing developers with automated support. It covers different processes, including automatic database query extraction, bad smell detection, self-admitted technical debt analysis, and evolution history visualization.

Csaba Nagy
Software Institute, Università della Svizzera italiana, e-mail: `csaba.nagy@usi.ch`

11.1 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 it with the best performance. The developer has limited control: If the query is not well-formed or not handled correctly in the program code, it generates an extra load on the database side that affects the application's performance [53] [82]. In the worst case, it can lead to errors, bugs, or 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 and evolving data-intensive applications [72, 93]. Another challenging trend is the development of *hybrid* multi-database architectures [17], 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 11.2 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 11.3 elaborates on static analysis and visualization techniques that exploit the mined information about storing and manipulating the ecosystem's data. In Section 11.4 we summarize the findings of empirical studies related to data-intensive software ecosystems. We provide concluding remarks and anticipate future directions in Section 11.5.

11.2 Mining Techniques

11.2.1 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. Maintaining and evolving this interconnected system network requires a fundamental understanding of 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 its source code. This can be particularly important for an ecosystem where dynamic analysis is even more challenging, if not impossible, in most situations.

In Subsection 11.2.2, 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 dynamically constructed at those locations. In Subsection 11.2.3, 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, and the approach tries to alleviate the limitations of static analysis by using heuristics, e.g., when types are not available explicitly.

11.2.2 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.* [43] found JPA, Hibernate, and JDBC among the most popular ones in 3,707 GitHub Java projects. 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 entirely hide the actual SQL queries executed by the programs, generating queries at run-time before sending them to the database server.

Listing 11.1 shows an example code snippet executing a SQL query through the JDBC API. `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 final query results from string operations (e.g., lines 9, 14) depending on conditions (e.g., lines 8, 19).

Listing 11.1: Java code example executing a SQL query

```
 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 }
```

Listing 11.2 presents a similar example usage of Hibernate to send an HQL query to the database. HQL (Hibernate Query Language) is the SQL-like query language of Hibernate. It is fully object-oriented and understands inheritance, polymorphism, and association. The example shows a 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.

Listing 11.2: Java code snippet executing an HQL query

```
 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()
```

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

```
            .createQuery("from Product product where category=?")
            .setParameter(0, category)
            .list();
    }
}
```

Meurice *et al.* addressed the problem of recovering traceability links between Java programs and their databases [70]. They proposed a static analysis approach to identify the source code locations where database queries are executed, extracting the 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.* [43], namely JDBC, Hibernate, and JPA.

An overview of Meurice *et al.*'s approach can be seen in Figure 11.1. 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 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.

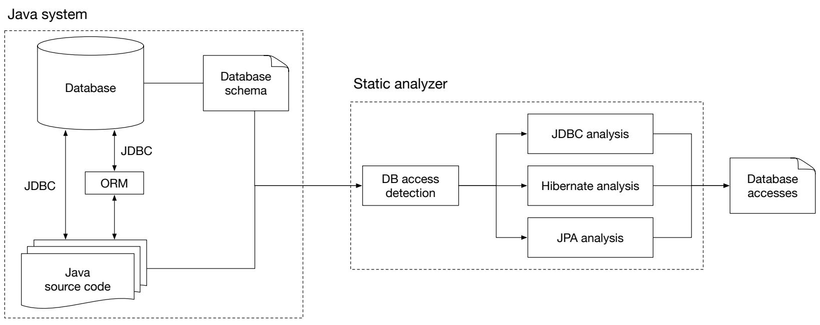


Fig. 11.1: Overview of the query extraction approach

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

In some cases, even such techniques cannot extract the entire query. Thus, a query might be incomplete. For example, a user may enter credentials in a login form to be validated in the database. The user input is known at run-time, 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,[2] OpenMRS,[3] and Broadleaf[4]) 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 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 [73], 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, different data manipulation technologies could access the same tables 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.

11.2.3 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 [69]. 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 complicates developers' operational and maintenance burden [92, 5].

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 [106]. For example, researchers study automatic schema extraction [1], schema generation [45], optimization [77], and schema suggestions [49]. 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 [26].

[2] `https://www.oscar-emr.com`

[3] `https://www.openmrs.org`

[4] `https://www.broadleafcommerce.org`

Static analysis of JavaScript is known to be extremely difficult. Existing techniques [54, 10] usually struggle to handle the excessively dynamic features of the language [107], and approaches with type inference [51], data flow [63], or call graphs [37] need to balance between scalability and soundness. Cherry *et al.* alleviate the limitations of the static analysis by using heuristics.

Listing 11.3 presents a typical schema definition in Mongoose,[5] 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. Finally, line 9 creates a `Model` which is exported to be used externally.

Listing 11.3: Mongoose schema definition example

```
1 const mongoose = require("mongoose");
2
3 let CarSchema = new mongoose.Schema({
4     brand: String,
5     model: String,
6     price: Number
7 });
8
9 module.exports = mongoose.model("cars", CarSchema);
```

Listing 11.4 shows an example usage of the model in Listing 11.3. 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 `tesla` instance (line 5). Finally, line 8 shows a simple query to find documents.

Listing 11.4: Mongoose query example

```
1 Car = require("./cars.js");
2
3 // ...
4 tesla = new Car("Tesla", "Model S", 95000);
5 await tesla.save();
6
7 // ...
8 tesla = await Car.find({name: /Tesla/});
```

Cherry *et al.* look for similar MongoDB interactions in JavaScript applications. They aim is to identify every statement operating 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) creating a new collection/document; (2) updating the content of documents or a collection; (3) deleting documents from a collection; (4)

[5] `https://mongoosejs.com/`

accessing the content of documents. Overall, they identified 179 methods, 74 from MongoDB Node Driver and 105 from Mongoose.

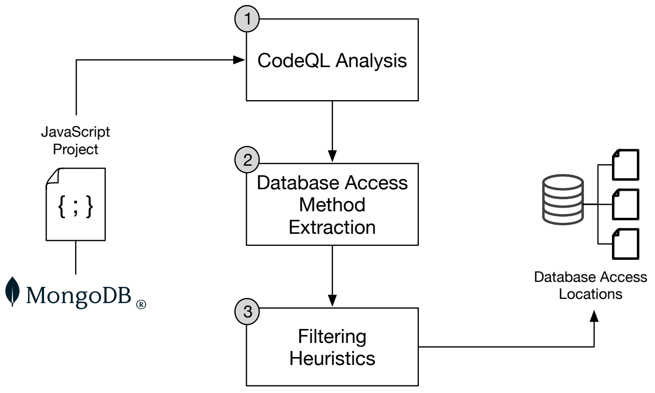


Fig. 11.2: Overview of the MongoDB data access analysis by Cherry *et al.* [26]

Figure 11.2 presents an overview of the main steps of the approach. First, it analyzes a JavaScript project with CodeQL,[6] 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[7] 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[8] and Overleaf.[9]

Figure 11.3 shows the database access methods in the different releases of Bitcore, a project with 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[10] adds numerous models and methods interacting

[6] `https://codeql.github.com/`

[7] `https://lodash.com/`

[8] `https://github.com/bitpay/bitcore`

[9] `https://github.com/overleaf/web`

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

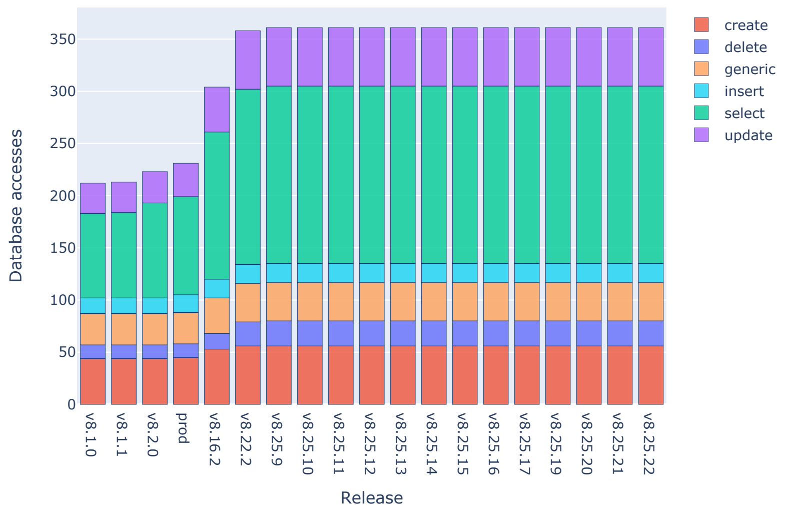


Fig. 11.3: Evolution of Bitcore

with it. It is a new feature: "*[Bitcore] can now sync ETH and get wallet history for ERC20 tokens*"—says the commit message.

Figure 11.4 shows the evolution of Overleaf, a popular online, collaborative LaTeX editor. Overleaf'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.

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 [72], help their developers propagate schema changes [2], or identify antipatterns [82].

11.2.4 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

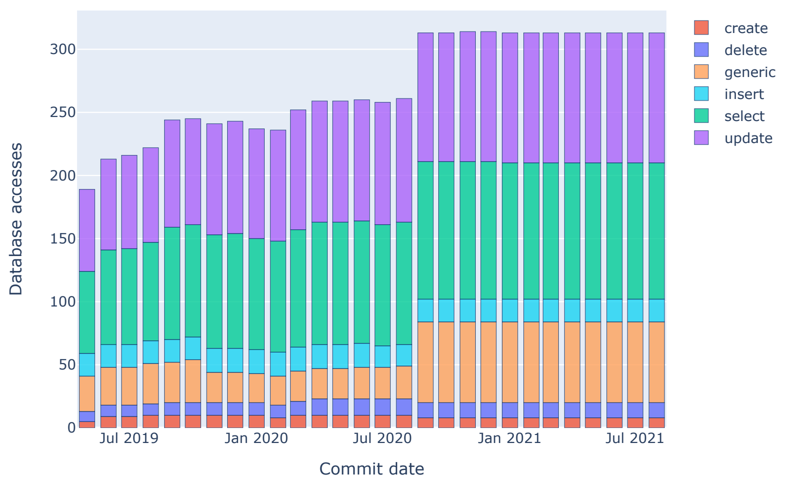


Fig. 11.4: Evolution of Overleaf

learned a technique for MongoDB as an example of a popular NoSQL database in the dynamically typed language context of JavaScript applications.

To extract SQL queries, pioneer work was published by Christensen *et al.* [27]. 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 [47, 116]. 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 [68]. Brink *et al.* present a quality assessment approach for SQL statements embedded in PL/SQL, COBOL, and Visual Basic code [23]. They extract 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 [11] similar to the one presented by Meurice *et al.* [75]. Ngo and Tan use symbolic execution to extract database interaction points from web applications [83]. 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 [8, 9, 7]. They implement their approach in Rascal as part of the PHP AiR framework. A similar approach was also presented by Manousis *et al.* [64]. 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 [62]. They look for invocations of SQLite APIs and their queries. Li *et al.* presented a general string analysis approach for Android [60]. 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 [30]. Noughi *et al.* mined SQL execution traces for data manipulation behavior recovery and conceptual interpretation [78, 84]. 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, e.g., to extract models from the JSON document database [20, 39, 15, 1]. Some approaches also deal with schema generation [45], optimization [77], and schema suggestions [49]. Also interesting to note is the work of Störl *et al.*, who studied schema evolution and data migration in a NoSQL environment [106]. As a similar approach to Cherry *et al.* [26], Meurice *et al.* implemented an approach to extract the database schema of MongoDB applications written in Java [72]. They applied their method to analyze the evolution of Java systems.

11.3 Analysis Techniques

11.3.1 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 Subsection 11.3.2, which rely on the mining techniques introduced in the previous section. Then we present visualization methods in Subsection 11.3.3 to analyze the dependencies between the database and different components of an ecosystem.

11.3.2 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 [75]. Moreover, because the application code and the database depend on each other, they evolve in parallel [73], 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.

11.3.2.1 Example 1: SQLInspect—A Static Analyzer

Static analyzers[11] 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,[12] a tool to identify code smells in SQL queries embedded in Java code [81, 82]. 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 11.2. It uses a path-sensitive string analysis algorithm to extract SQL queries from the Java code and implements smell detectors on the abstract semantic graph of a fault-tolerant SQL parser. The supported SQL smells are based on the *SQL Antipatterns* book of Karwin [53]. SQLInspect can also perform additional analyses: (1) It supports inspecting the interprocedural slice of the statements involved in the query construction; (2) It can perform a table/column access analysis to 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 Figure 11.5.

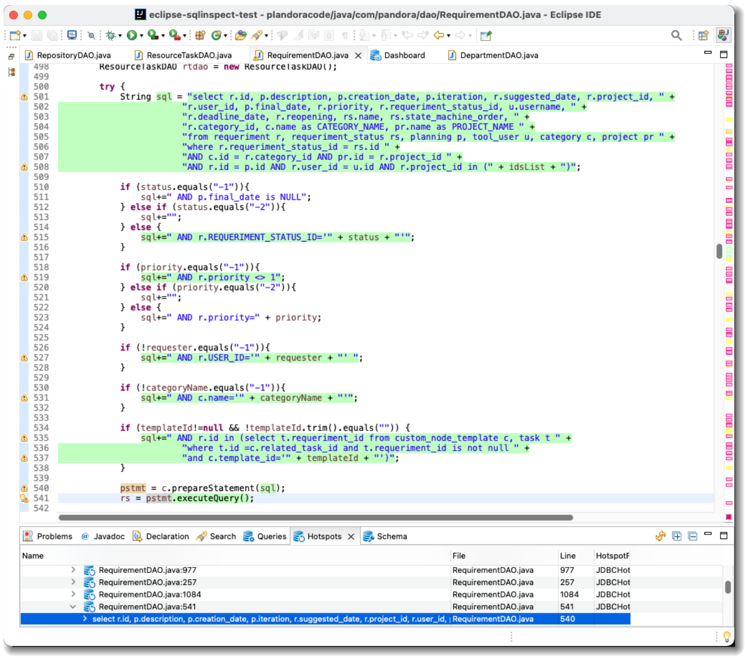
SQLInspect has been used in various studies. Muse *et al.* relied on it to study SQL code smells [80], and the prevalence, composition, and evolution of self-admitted technical debt in data-intensive systems [79]. Gobert *et al.* employed SQLInspect to study developers' testing practices of database access code [40, 41]. Ardigò *et al.* also relied on it to visualize database accesses of a system [12, 13].

11.3.2.2 Example 2: Preventing Program Inconsistencies

Any software system is subject to frequent changes [58], which often hit the database schema [96, 31, 112]. When the schema evolves, developers need to adapt the applications where it accesses the changed schema elements [68, 25, 88]. This

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

[12] `https://bitbucket.org/csnagy/sqlinspect/`



```java
        ResourceTaskDAO rtdao = new ResourceTaskDAO();

        try {
            String sql = "select r.id, p.description, p.creation_date, p.iteration, r.suggested_date, r.project_id, " +
                         "r.user_id, p.final_date, r.priority, r.requeriment_status_id, u.username, " +
                         "r.deadline_date, r.reopening, rs.name, rs.state_machine_order, " +
                         "r.category_id, c.name as CATEGORY_NAME, pr.name as PROJECT_NAME " +
                         "from requeriment r, requeriment_status rs, planning p, tool_user u, category c, project pr " +
                         "where r.requeriment_status_id = rs.id " +
                         "AND c.id = r.category_id AND pr.id = r.project_id " +
                         "AND r.id = p.id AND r.user_id = u.id AND r.project_id in (" + idsList + ")";

            if (status.equals("-1")){
                sql+=" AND p.final_date is NULL";
            } else if (status.equals("-2")){
                sql+="";
            } else {
                sql+=" AND r.REQUERIMENT_STATUS_ID='" + status + "'";
            }

            if (priority.equals("-1")){
                sql+=" AND r.priority <> 1";
            } else if (priority.equals("-2")){
                sql+="";
            } else {
                sql+=" AND r.priority=" + priority;
            }

            if (!requester.equals("-1")){
                sql+=" AND r.USER_ID='" + requester + "' ";
            }

            if (!categoryName.equals("-1")){
                sql+=" AND c.name='" + categoryName + "'";
            }

            if (templateId!=null && !templateId.trim().equals("")) {
                sql+=" AND r.id in (select t.requeriment_id from custom_node_template c, task t " +
                         "where t.id =c.related_task_id and t.requeriment_id is not null " +
                         "and c.template_id='" + templateId + "')";
            }

            pstmt = c.prepareStatement(sql);
            rs = pstmt.executeQuery();

```

Name	File	Line	HotspotF
RequirementDAO.java:977	RequirementDAO.java	977	JDBCHot
RequirementDAO.java:257	RequirementDAO.java	257	JDBCHot
RequirementDAO.java:1084	RequirementDAO.java	1084	JDBCHot
RequirementDAO.java:541	RequirementDAO.java	541	JDBCHot
select r.id, p.description, p.creation_date, p.iteration, r.suggested_date, r.project_id, r.user_id,	RequirementDAO.java	540	

Fig. 11.5: A query slice in the Eclipse plug-in of SQLInspect

adaptation process is usually achieved manually. Thus, it can be error-prone resulting in database or application decay [102, 103].

Meurice *et al.* proposed an approach to detect and prevent program inconsistencies under database schema changes [74]. 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 presented in Section 11.2. 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 the columns of database tables and links them to source code entities where they are accessed in the application code.

Meurice *et al.* demonstrated their what-if analysis on the three open-source Java systems (OpenMRS, Broadleaf Commerce, and OSCAR) used for the query extraction approach in Subsection 11.2.2. Recall that OpenMRS and OSCAR are electronic medical record systems, and Broadleaf is an e-commerce framework. The largest one, OSCAR, has 2,054 kLOC and 480 tables.

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.

11.3.3 Visualization

11.3.3.1 Introduction

Software visualization is *"the use of [. . .] computer graphics technology to facilitate both the human understanding and effective use of computer software"* [100]. It is a specialization of *information visualization* [24]. In the 18th century, starting with Playfair, the classical methods of plotting data were developed. In 1967 Bertin published "Semiology of Graphics" [18], where he identified the basic elements of diagrams. Later, Tufte published a theory of data graphics that emphasized maximizing the density of useful information [108, 109, 110]. 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 [115], visualization is the preferred way of getting acquainted with large data.

Software visualization deals with software 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 [104, 66, 19, 36, 105] 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.

11.3.3.2 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 [50]. 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 [70, 71]. 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. 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 11.3 [75] and analyzes the evolution of the system by mining its development history [74].

DAHLIA has multiple views. It can visualize the *database as a city* using the 3D city metaphor [118]. 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 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 Figure 11.6.

Another view shows the *code city* view, which, compared to the traditional code city [119], 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).

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 Figure 11.7 for OpenMRS, a medical record system that we also analyzed in Subsection 11.2.2. The green tables are accessed with Hibernate mapping in the figure, and the black tables are without 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. The user selected a Java file highlighted in the right code city with cyan. DAHLIA highlighted all its tables in the left database city with 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.

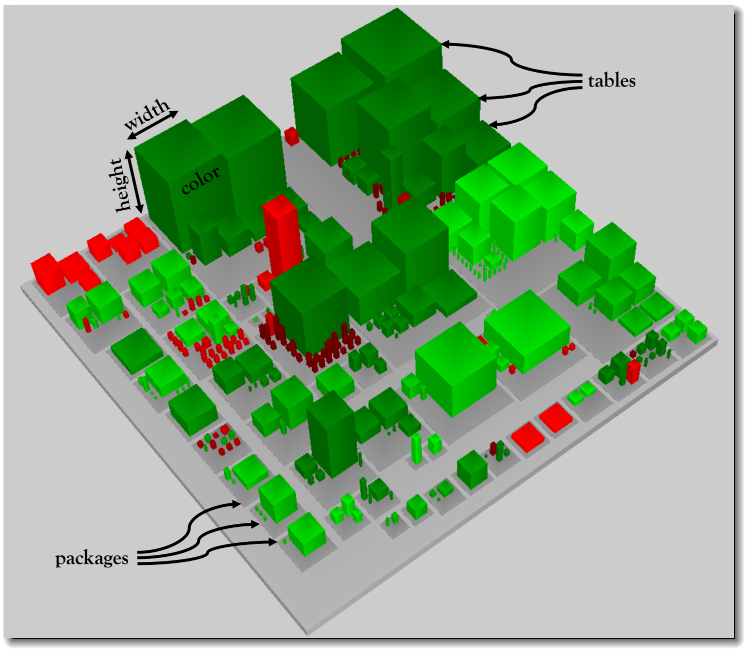


Fig. 11.6: A 3D database city in DAHLIA

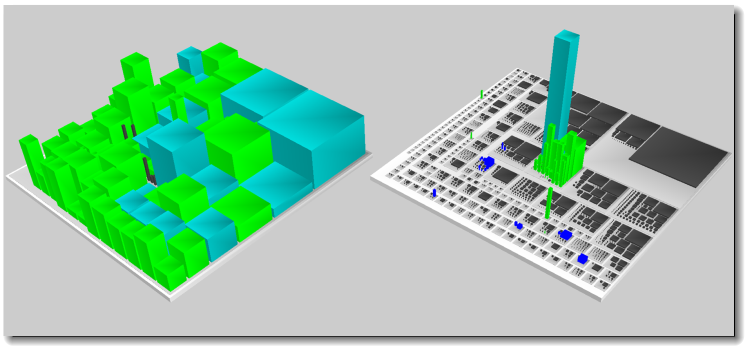


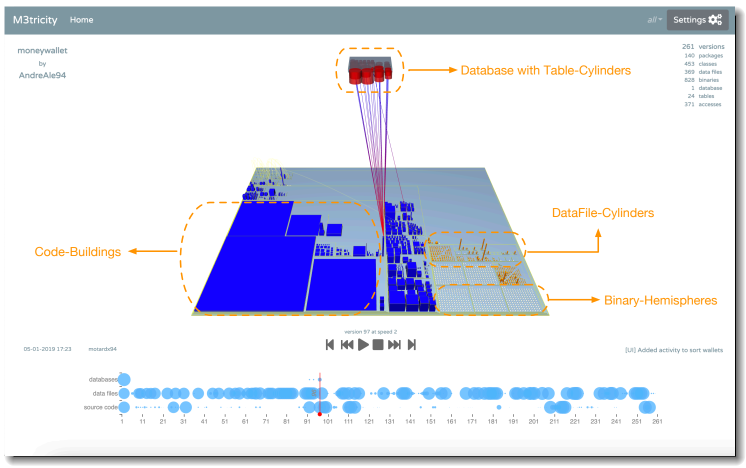
Fig. 11.7: Database (left) and Code (right) cities side-by-side in DAHLIA

11.3.3.3 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 focusing on data interactions [12, 13].

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 [82]. 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.* [87], 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.



moneywallet
by
AndreAle94

261 versions
140 packages
453 classes
369 data files
828 binaries
1 database
24 tables
371 accesses

version 97 at speed 2

05-01-2019 17:23 motardx94

[UI] Added activity to sort wallets

Fig. 11.8: The main page of M3TRICITY

Figure 11.8 shows a screenshot of MoneyWallet[14] visualized in M3TRICITY. It is an expense manager Android app with an SQLite database. 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

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

[14] `https://github.com/AndreAle94/moneywallet`

evolution of the project, where one can spot significant changes in the metrics. The evolution can be controlled with the buttons below the city.

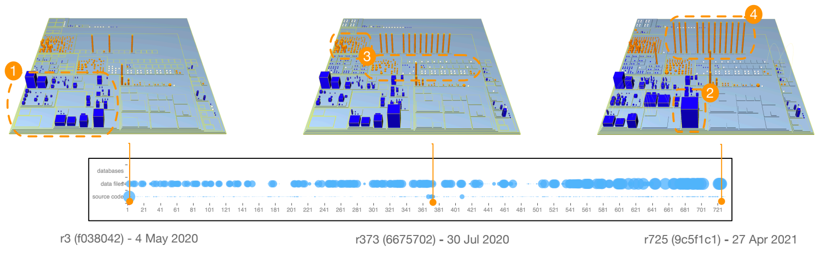

1
2
3
4
databases
data files
source code
r3 (f038042) - 4 May 2020
r373 (6675702) - 30 Jul 2020
r725 (9c5f1c1) - 27 Apr 2021

Fig. 11.9: The evolution of the SwissCovid Android App in M3TRICITY

Figure 11.9 presents the evolution of the official SwissCovid Android App[15] 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`,[16] 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.

11.3.4 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.

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

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

Static 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 [21] and then extended their work by recognizing common semantic mistakes [22]. Their SQLLint tool automatically identifies such errors in (syntactically correct) SQL statements [44]. There are also books in this area, e.g., *The Art of SQL* [35], *Refactoring SQL Applications* [34], and *SQL Antipatterns* [53].

In the realm of embedded SQL, Christensen *et al.* proposed a tool (JSA, Java String Analyzer) to extract string expressions from Java code statically [27]. 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 [116]. 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 [46]. Anderson and Hills studied query construction patterns in PHP [7]. 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 [23]. They analyze query strings embedded in PL/SQL, Cobol, and Visual Basic programs [111]. Many static techniques deal with embedded queries for SQL injection detection [98]. Their goal is to determine whether a query could be affected by user input. Yeole and Meshram published a survey of these techniques [120]. Marashdeh *et al.* also surveyed the challenges of detecting SQL Injection vulnerabilities [65].

Some papers also tackle SQL fault localization techniques. Clark *et al.* proposed a dynamic approach to localize SQL faults in database applications [28]. 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 [32]. Their tool, DBCritics, analyzes PostgreSQL schema dumps and identifies design smells such as missing primary keys or foreign key references. Alvor by Annamaa *et al.* can analyze string expressions in Java code [11]. 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 [90] and Young & Munro [121], many have 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.* [56], the visualization of communicating architectures by Panas *et al.* [85, 86], Verso by Langelier *et al.* [57], CodeCity by Wettel *et al.* [118, 119], EVO-STREETS by Steinbrückner & Lewerentz [101], CodeMetropolis by Balogh & Beszedes [16], and VR City by Vincur *et al.* [114].

Some approaches considered presenting the databases together with the source code, and interestingly, most use the city metaphor, such as DAHLIA [70, 71] and M3TRICITY [12, 13]. Zirkelbach and Hasselbring presented RACCOON [122], 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 [67].

11.4 Empirical Studies

11.4.1 Introduction

In this section, we discuss recent empirical studies relying on large collections of data-intensive software systems. The discussed studies cover three main aspects (1) the (joint) use of database models and access technologies (Subsection 11.4.2); (2) the quality of the database manipulation code (Subsection 11.4.3 and Subsection 11.4.4); (3) the way this part of the code is tested (Subsection 11.4.5).

11.4.2 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 (i.e., using different database models for various purposes in the same system, such as a key-value store for caching and a relational database for persistence) 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.* [17] conducted an empirical study of (multi-)database models in open-source database-dependent projects. They mined four years of development history (2017–2020) of 33 million projects by leveraging Libraries.io.[17] 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 gathered a dataset of 40,609 projects. They analyzed the dependencies of those projects to assess (1) the popularity of 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 compared to 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

[17] `https://libraries.io/`

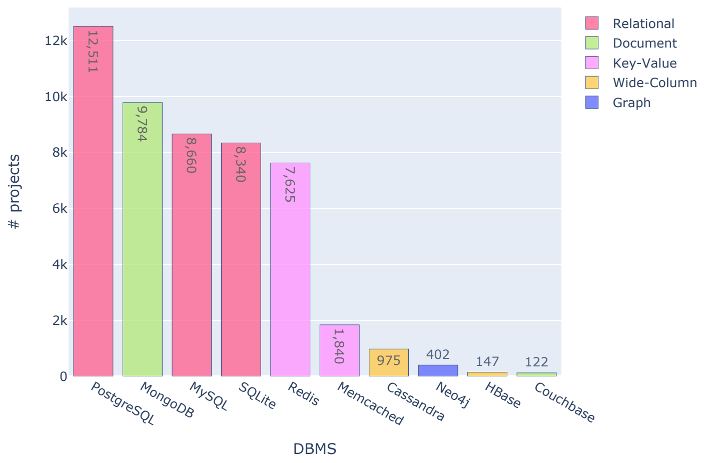


Fig. 11.10: Usage of database management systems (2020)

follow the same pattern, with over 47% being hybrid. This shows the complimentary usage of SQL and NoSQL in practice.

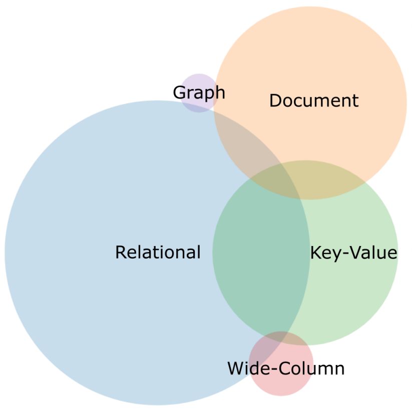


Fig. 11.11: Distribution of hybrid database-dependent projects

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.

11.4.3 Prevalence, Impact and Evolution of SQL Bad Smells

Muse *et al.* [80] 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 [48] and 4 SQL code smells using SQLInspect [82]. They also collected bug-fixing and bug-inducing commits from each project using PyDriller [99].

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 show a higher association with bugs than 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.

11.4.4 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.* [79] 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 analysed 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, and conducted a survival analysis to understand the evolutionary dynamics of SATDs.

They analysed 361 sample data-access SATDs manually, 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.

11.4.5 Database Code Testing (Best) Practices

Software testing allows developers to maintain the quality of a software system over time. The database access code fragments are often neglected in this context, although they require specific attention.

Gobert *et al.* [40] empirically analysed the challenges and perils of database access code testing. They first mined open-source systems from Libraries.io to find projects relying on database manipulation technologies. They analysed 6,622 projects and found automated tests and database manipulation code in only 332 projects. They further examined the 72 projects for which the tests could be executed and analyzed the corresponding coverage reports.

Figure 11.12 shows a scatter plot of the analysed projects and their respective test coverage rates. The results show that the *database manipulation code was poorly tested*: 33% of the projects did not test DB communication, and 46% did not test half of their DB methods. A high number of projects with the highest coverage

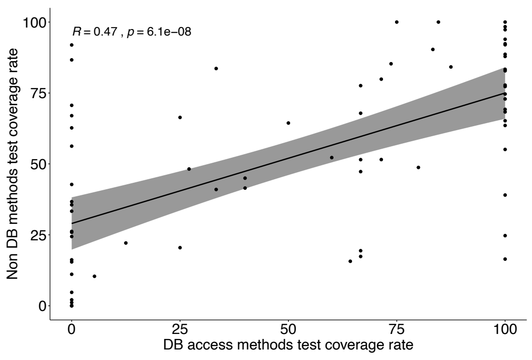


Fig. 11.12: Test coverage rates of Non-DB access methods vs. DB access methods

rate reached, in fact, full coverage. The authors found a mean value of 2.8 database methods for projects with full coverage. Slightly fewer projects in the figure (48.6%) had lower coverage for database methods. However, considering only the projects with at least five database methods (the median value), there is a more significant difference: 59% have a smaller coverage for database methods than for 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.

This poor test coverage motivated the authors to understand why developers were holding back from testing DB manipulation code. They conducted a qualitative analysis of 532 questions from popular Stack Exchange websites. They identified the *problems* that hampered developers in writing tests. Then they built a *taxonomy of issues* with 83 different problems classified into 7 main categories. They found out that developers mostly look for insights on general best practices to test database manipulation code. They also ask more technical questions related to DB handling, mocking, parallelisation or framework/tool usage.

In a follow-up study, the same authors analysed the *answers* to these questions [41]. 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, leading to 363 different practices in the taxonomy.

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

Most suggestions consider the testing environment and recommend various tools or configurations. The second largest category is database management, mainly addressing database initialisation and clean-up between tests. Other categories pertain to code structure or design, concepts, performance, processes, test characteristics, test code, and mocking.

11.4.6 Reflections

Studies on code quality. Other researchers studied frequent errors and antipatterns in SQL queries. The book of Karwin [53] is the first to present SQL antipatterns in a comprehensive catalogue. Khumnin *et al.* [55] present a tool for detecting logical database design antipatterns in Transact-SQL queries.

Another tool, *DbDeo* [95], implements the detection of *database schema* smells. *DbDeo* has been evaluated on 2,925 open-source repositories; their authors identified 13 different types of smells, among which "index abuse" was the most prevalent. De Almeida Filho *et al.* [6] investigate the prevalence and co-occurrence of SQL code smells in PL/SQL projects. Arzamasova *et al.* propose to detect antipatterns in SQL logs [14] and demonstrate their approach by refactoring a project containing more than 40 million queries. Shao *et al.* [94] identified a list of database-access performance antipatterns, mainly in PHP web applications. Integrity violation was addressed by Li *et al.* [59], who identified constraints from source code and related them to database attributes.

Studies on evolution. Several researchers studied how data-intensive systems relying on a relational database evolve. Curino *et al.* [31] analyse the evolution history of the Wikipedia database schema, to extract both a micro-classification and a macro-classification of schema changes. An evolution period of four years was considered, corresponding to 171 successive schema versions. In addition to a schema evolution statistics extractor, the authors propose a tool operating on the differences between subsequent schema versions to semi-automatically extract the schema changes that have possibly been applied. The results motivate the need for automated schema evolution support. Vassiliadis *et al.* [112, 113] study the evolution of individual database tables over time in eight different software systems. They observe that evolution-related properties, such as the possibility of deletion, or the updates a table undergoes, are related to observable table properties, such as the number of attributes or the time of birth of a table. Through a large-scale study on the evolution of databases, they also show that the essence of Lehman's laws of software evolution remains valid in the context of database schema evolution [97], but that specific mechanics significantly differ from source code evolution. Dimolikas *et al.* [33] analyse the evolution history of six database schemas, and reveal that the update behavior of tables depend on their topological complexity. Cleve *et al.* [29] show that mining database schema evolution can have a significant informative value in reverse engineering. They introduce the concept of *global historical schema*, an aggregated schema of all successive schema versions. They then analyse this global schema to better understand the current version of the database structure, intending to facilitate its evolution. Lin *et al.* [61] study the *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 were deleted.

Qiu *et al.* [88] present a large-scale empirical study on ten popular database applications from various domains to analyze how schemas and application code co-evolve. They studiy the evolution histories of the repositories to understand whether database schemas evolve frequently and significantly, and how schemas evolve and impact the application code. Their analysis estimates the impact of a database schema change in the code. They use a simple difference calculation of the source lines changed between two versions for this estimation. Goeminne *et al.* [42] 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 analyse the evolution of SQL, Hibernate and JPA usage in a large and complex open-source information system. They observed that using embedded SQL queries was still a common practice.

Other studies exclusively focus on NoSQL applications. Störl *et al.* [106] 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 creates 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.* [93] 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.* [91] 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. Several studies are related to technical debt in data-intensive systems. Albarak and Bashoon [3] 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 [4]. To tackle this type of debt, they propose to prioritize tables that should be normalized. Foidl *et al.* 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 [38]. They show that those three parts can further affect each other. They present two smells as illustrations: 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.* [117] identified relational database schemas as potential sources of technical debt. They provide a first attempt at utilizing the technical debt analogy for describing 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, which was also used to demonstrate the static analysis approach in Subsection 11.2.2. Ramasubbu and Kemerer [89] empirically analyse the impact of technical debt on system reliability by observing a 10-year life cycle of a commercial enterprise system. They examine the relative effects of modular and architectural maintenance activities in clients, and 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.

11.5 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;
4. software evolution methods should devote more attention to the program-database co-evolution problem.

The research community faces many challenges and open questions in the near future, given the increasing complexity and heterogeneity of data-intensive software ecosystems. For instance, to fully embrace the *DevOps* movement, developers need better support for database-related analyses and evolutions *at run-time* [52]. This is the case, in particular, when developing micro-services applications deployed on distributed computing architectures such as the *cloud-edge continuum* [76]. Furthermore, machine learning techniques, such as those described in Chapter 10, open the door to novel recommenders helping developers to design, understand, evolve, test and improve the performance of modern data-intensive systems.

References

1. Abdelhedi, F., Brahim, A., Rajhi, H., Ferhat, R., Zurfluh, G.: Automatic extraction of a document-oriented NoSQL schema. In: Int. Conf. Enterprise Information Systems (2021)
2. Afonso, A., da Silva, A., Conte, T., Martins, P., Cavalcanti, J., Garcia, A.: LESSQL: dealing with database schema changes in continuous deployment. In: IEEE 27th Int. Conf. Software Analysis, Evolution and Reengineering (SANER 2020), pp. 138–148 (2020). DOI 10.1109/SANER48275.2020.9054796
3. Albarak, M., Bahsoon, R.: Database design debts through examining schema evolution. In: International Workshop on Managing Technical Debt (MTD), pp. 17–23 (2016). DOI 10.1109/MTD.2016.9
4. Albarak, M., Bahsoon, R.: Prioritizing technical debt in database normalization using portfolio theory and data quality metrics. In: International Conference on Technical Debt (TechDebt), pp. 31–40. ACM (2018). DOI 10.1145/3194164.3194170
5. Alger, K.W., Daniel Coupal, D.: Building with patterns: The polymorphic pattern. `https://www.mongodb.com/developer/products/mongodb/polymorphic-pattern/` (2022). Accessed 2023-04-15
6. de Almeida Filho, F.G., Martins, A.D.F., Vinuto, T.d.S., Monteiro, J.M., de Sousa, Í.P., de Castro Machado, J., Rocha, L.S.: Prevalence of bad smells in PL/SQL projects. In: International Conference on Program Comprehension (ICPC), pp. 116–121. IEEE (2019)
7. Anderson, D.: Modeling and analysis of SQL queries in PHP systems. Master's thesis, East Carolina University (2018)
8. Anderson, D., Hills, M.: Query construction patterns in PHP. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 452–456 (2017). DOI 10.1109/SANER.2017.7884652
9. Anderson, D., Hills, M.: Supporting analysis of SQL queries in PHP AiR. In: International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 153–158 (2017). DOI 10.1109/SCAM.2017.23
10. Andreasen, E., Møller, A.: Determinacy in static analysis for jQuery. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 17–31 (2014). DOI 10.1145/2660193.2660214
11. Annamaa, A., Breslav, A., Kabanov, J., Vene, V.: An interactive tool for analyzing embedded SQL queries. In: Asian Symposium on Programming Languages and Systems (APLAS), *Lecture Notes in Computer Science*, vol. 6461, pp. 131–138. Springer (2010)
12. Ardigò, S., Nagy, C., Minelli, R., Lanza, M.: Visualizing data in software cities. In: Working Conference on Software Visualization (VISSOFT), NIER/TD, pp. 145–149. IEEE (2021). DOI 10.1109/VISSOFT52517.2021.00028
13. Ardigò, S., Nagy, C., Minelli, R., Lanza, M.: M3triCity: Visualizing evolving software & data cities. In: International Conference on Software Engineering (ICSE), pp. 130–133. IEEE (2022). DOI 10.1145/3510454.3516831
14. Arzamasova, N., Schäler, M., Böhm, K.: Cleaning antipatterns in an SQL query log. Transactions on Knowledge and Data Engineering **30**(3), 421–434 (2018)
15. Baazizi, M.A., Colazzo, D., Ghelli, G., Sartiani, C.: Parametric schema inference for massive JSON datasets. The VLDB Journal **28**(4), 497–521 (2019). DOI 10.1007/s00778-018-0532-7
16. Balogh, G., Beszedes, A.: CodeMetropolis - code visualisation in MineCraft. In: International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 136–141. IEEE (2013)
17. Benats, P., Gobert, M., Meurice, L., Nagy, C., Cleve, A.: An empirical study of (multi-) database models in open-source projects. In: International Conference on Conceptual Modeling (ER), pp. 87–101. Springer (2021)
18. Bertin, J.: Graphische Semiologie, second edn. Walter de Gruyter (1974)
19. Beyer, D., Lewerentz, C.: CrocoPat: A tool for efficient pattern recognition in large object-oriented programs. Tech. Rep. I-04/2003, Institute of Computer Science, Brandenburgische Technische Universität Cottbus (2003)

20. Brahim, A.A., Ferhat, R.T., Zurfluh, G.: Model driven extraction of NoSQL databases schema: Case of MongodB. In: Int. Joint Conf. on Knowledge Discovery, Knowledge Engineering and Knowledge Management, pp. 145–154 (2019). DOI 10.5220/0008176201450154
21. Brass, S., Goldberg, C.: Detecting logical errors in SQL queries. In: Workshop on Foundations of Databases (2004)
22. Brass, S., Goldberg, C.: Semantic errors in SQL queries: A quite complete list. Journal of Systems and Software **79**(5), 630–644 (2006). DOI 10.1016/j.jss.2005.06.028
23. van den Brink, H., van der Leek, R., Visser, J.: Quality assessment for embedded SQL. In: International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 163–170 (2007). DOI 10.1109/SCAM.2007.23
24. Card, S.K., Mackinlay, J.D., Shneiderman, B. (eds.): Readings in Information Visualization — Using Vision to Think. Morgan Kaufmann (1999)
25. Chen, T.H., Shang, W., Jiang, Z.M., Hassan, A.E., Nasser, M., Flora, P.: Detecting performance anti-patterns for applications developed using object-relational mapping. In: International Conference on Software Engineering (ICSE), pp. 1001–1012. ACM (2014). DOI 10.1145/2568225.2568259
26. Cherry, B., Benats, P., Gobert, M., Meurice, L., Nagy, C., Cleve, A.: Static analysis of database accesses in mongodb applications. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 930–934. IEEE (2022). DOI 10.1109/SANER2022.2022.00111
27. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: International Conference on Static Analysis (SAS), pp. 1–18. Springer (2003)
28. Clark, S.R., Cobb, J., Kapfhammer, G.M., Jones, J.A., Harrold, M.J.: Localizing SQL faults in database applications. In: International Conference on Automated Software Engineering (ASE), pp. 213–222. ACM (2011). DOI 10.1109/ASE.2011.6100056
29. Cleve, A., Gobert, M., Meurice, L., Maes, J., Weber, J.: Understanding database schema evolution: A case study. Science of Computer Programming **97**, 113–121 (2015). DOI 10.1016/j.scico.2013.11.025. Special Issue on New Ideas and Emerging Results in Understanding Software
30. Cleve, A., Hainaut, J.L.: Dynamic analysis of SQL statements for data-intensive applications reverse engineering. In: Working Conference on Reverse Engineering (WCRE), pp. 192–196 (2008). DOI 10.1109/WCRE.2008.38
31. Curino, C.A., Tanca, L., Moon, H.J., Zaniolo, C.: Schema evolution in Wikipedia: toward a web information system benchmark. In: International Conference on Enterprise Information Systems (ICEIS) (2008)
32. Delplanque, J., Etien, A., Auverlot, O., Mens, T., Anquetil, N., Ducasse, S.: Codecritics applied to database schema: Challenges and first results. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 432–436 (2017). DOI 10.1109/SANER.2017.7884648
33. Dimolikas, K., Zarras, A.V., Vassiliadis, P.: A study on the effect of a table's involvement in foreign keys to its schema evolution. In: International Conference on Conceptual Modeling (ER), pp. 456–470. Springer (2020)
34. Faroult, S., L'Hermite, P.: Refactoring SQL Applications. O'Reilly Media (2008)
35. Faroult, S., Robson, P.: The Art of SQL. O'Reilly Media (2006)
36. Favre, J.M.: GSEE: a generic software exploration environment. In: International Workshop on Program Comprehension (ICPC), pp. 233–244. IEEE (2001)
37. Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Efficient construction of approximate call graphs for JavaScript IDE services. In: International Conference on Software Engineering (ICSE), pp. 752–761. IEEE (2013)
38. Foidl, H., Felderer, M., Biffl, S.: Technical debt in data-intensive software systems. In: 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2019), pp. 338–341 (2019). DOI 10.1109/SEAA.2019.00058
39. Gallinucci, E., Golfarelli, M., Rizzi, S.: Schema profiling of document-oriented databases. Inf. Systems **75**, 13–25 (2018). DOI 10.1016/j.is.2018.02.007

40. Gobert, M., Nagy, C., Rocha, H., Demeyer, S., Cleve, A.: Challenges and perils of testing database manipulation code. In: International Conference on Advanced Information Systems Engineering (CAiSE), pp. 229–245. Springer (2021)
41. Gobert, M., Nagy, C., Rocha, H., Demeyer, S., Cleve, A.: Best practices of testing database manipulation code. Information Systems **111**, 102105 (2023). DOI 10.1016/j.is.2022.102105
42. Goeminne, M., Decan, A., Mens, T.: Co-evolving code-related and database-related changes in a data-intensive software system. In: Software Evoluton Week (CSMR/WCRE) (2014)
43. Goeminne, M., Mens, T.: Towards a survival analysis of database framework usage in Java projects. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 551–555. IEEE (2015). DOI 10.1109/ICSM.2015.7332512
44. Goldberg, C.: Do you know SQL? about semantic errors in database queries. Tech. rep., Higher Education Academy (2008)
45. Gómez, P., Casallas, R., Roncancio, C.: Automatic schema generation for document-oriented systems. In: Database and Expert Systems Applications, pp. 152–163. Springer (2020)
46. Gould, C., Su, Z., Devanbu, P.: JDBC Checker: A static analysis tool for SQL/JDBC applications. In: International Conference on Software Engineering (ICSE), pp. 697–698 (2004). DOI 10.1109/ICSE.2004.1317494
47. Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. In: International Conference on Software Engineering (ICSE), pp. 645–654 (2004). DOI 10.1109/ICSE.2004.1317486
48. Guéhéneuc, Y.G.: Ptidej: A flexible reverse engineering tool suite. In: 2007 IEEE International Conference on Software Maintenance, pp. 529–530. IEEE (2007)
49. Imam, A.A., Basri, S., Ahmad, R., Watada, J., González-Aparicio, M.T.: Automatic schema suggestion model for NoSQL document-stores databases. Journal of Big Data **5** (2018)
50. Ireland, C., Bowers, D., Newton, M., Waugh, K.: A classification of object-relational impedance mismatch. In: International Confernce on Advances in Databases, Knowledge, and Data Applications, pp. 36–43 (2009). DOI 10.1109/DBKDA.2009.11
51. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Static Analysis, pp. 238–255. Springer (2009)
52. de Jong, M., van Deursen, A., Cleve, A.: Zero-downtime sql database schema evolution for continuous deployment. In: International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 143–152. IEEE (2017). DOI 10.1109/ICSE-SEIP.2017.5
53. Karwin, B.: SQL Antipatterns: Avoiding the Pitfalls of Database Programming. Pragmatic Programmers (2010)
54. Kashyap, V., Dewey, K., Kuefner, E.A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., Hardekopf, B.: JSAI: A static analysis platform for JavaScript. In: International Symposium on Foundations of Software Engineering (FSE), pp. 121–132. ACM (2014). DOI 10.1145/2635868.2635904
55. Khumnin, P., Senivongse, T.: SQL antipatterns detection and database refactoring process. In: ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD), pp. 199–205 (2017). DOI 10.1109/SNPD.2017.8022723
56. Knight, C., Munro, M.C.: Virtual but visible software. In: International Conference on Information Visualization (IV), pp. 198–205. IEEE (2000)
57. Langelier, G., Sahraoui, H., Poulin, P.: Visualization-based analysis of quality for large-scale software systems. In: International Conference on Automated Software Engineering (ASE), pp. 214–223. ACM (2005)
58. Lehman, M.M.: Laws of software evolution revisited. In: C. Montangero (ed.) Software Process Technology, pp. 108–124. Springer (1996)
59. Li, B., Poshyvanyk, D., Grechanik, M.: Automatically detecting integrity violations in database-centric applications. In: International Conference on Program Comprehension (ICPC), pp. 251–262 (2017). DOI 10.1109/ICPC.2017.37

60. Li, D., Lyu, Y., Wan, M., Halfond, W.G.J.: String analysis for java and android applications. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 661–672. ACM (2015). DOI 10.1145/2786805.2786879
61. Lin, D.Y., Neamtiu, I.: Collateral evolution of applications and databases. In: Joint Int'l Workshop on Principles of software evolution and ERCIM software evolution workshop, pp. 31–40. ACM (2009). DOI 10.1145/1595808.1595817
62. Lyu, Y., Gui, J., Wan, M., Halfond, W.G.J.: An empirical study of local database usage in Android applications. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 444–455 (2017). DOI 10.1109/ICSME.2017.75
63. Madsen, M., Møller, A.: Sparse dataflow analysis with pointers and reachability. In: Static Analysis, pp. 201–218. Springer (2014)
64. Manousis, P., Zarras, A., Vassiliadis, P., Papastefanatos, G.: Extraction of embedded queries via static analysis of host code. In: Advanced Information Systems Engineering (CAiSE), pp. 511–526. Springer (2017)
65. Marashdeh, Z., Suwais, K., Alia, M.: A survey on SQl injection attack: Detection and challenges. In: International Conference on Information Technology (ICIT), pp. 957–962 (2021). DOI 10.1109/ICIT52682.2021.9491117
66. Marcus, A., Feng, L., Maletic, J.I.: 3D representations for software visualization. In: ACM Symposium on Software Visualization, p. 27. IEEE (2003)
67. Marinescu, C.: Applications of automated model's extraction in enterprise systems. In: International Conference on Software Technologies (ICSOFT), pp. 254–261. SCITEPRESS (2019)
68. Maule, A., Emmerich, W., Rosenblum, D.: Impact analysis of database schema changes. In: International Conference on Software Engineering (ICSE), pp. 451–460 (2008). DOI 10.1145/1368088.1368150
69. McKnight: NoSQL Evaluator's Guide (2014)
70. Meurice, L., Cleve, A.: DAHLIA: A visual analyzer of database schema evolution. In: Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), pp. 464–468. IEEE (2014). DOI 10.1109/CSMR-WCRE.2014.6747219
71. Meurice, L., Cleve, A.: DAHLIA 2.0: A visual analyzer of database usage in dynamic and heterogeneous systems. In: Working Conference on Software Visualization (VISSOFT), pp. 76–80. IEEE (2016). DOI 10.1109/VISSOFT.2016.15
72. Meurice, L., Cleve, A.: Supporting schema evolution in schema-less NoSQL data stores. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 457–461 (2017). DOI 10.1109/SANER.2017.7884653
73. Meurice, L., Goeminne, M., Mens, T., Nagy, C., Decan, A., Cleve, A.: Analyzing the evolution of database usage in data-intensive software systems. In: Software Technology: 10 Years of Innovation, pp. 208–240. Wiley (2018). DOI 10.1002/9781119174240.ch12
74. Meurice, L., Nagy, C., Cleve, A.: Detecting and preventing program inconsistencies under database schema evolution. In: International Conference on Software Quality, Reliability & Security (QRS), pp. 262–273. IEEE (2016). DOI 10.1109/QRS.2016.38
75. Meurice, L., Nagy, C., Cleve, A.: Static analysis of dynamic database usage in Java systems. In: International Conference on Advanced Information Systems Engineering (CAiSE), pp. 491–506. Springer (2016). DOI 10.1007/978-3-319-39696-5%5C_30
76. Milojicic, D.: The edge-to-cloud continuum. IEEE Annals of the History of Computing **53**, 16–25 (2020)
77. Mior, M.J.: Automated schema design for NoSQL databases. In: 2014 SIGMOD PhD Symposium, pp. 41–45. ACM (2014). DOI 10.1145/2602622.2602624
78. Mori, M., Noughi, N., Cleve, A.: Mining SQL execution traces for data manipulation behavior recovery. In: International Conference on Advanced Information Systems Engineering (CAiSE) (2014)

79. Muse, B.A., Nagy, C., Cleve, A., Khomh, F., Antoniol, G.: FIXME: Synchronize with database an empirical study of data access self-admitted technical debt. Empirical Software Engineering **27**(6) (2022). DOI 10.1007/s10664-022-10119-4
80. Muse, B.A., Rahman, M., Nagy, C., Cleve, A., Khomh, F., Antoniol, G.: On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. In: International Conference on Mining Software Repositories (MSR), pp. 327–338. ACM (2020). DOI 10.1145/3379597.3387467
81. Nagy, C., Cleve, A.: Static code smell detection in SQL queries embedded in Java code. In: International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 147–152. IEEE (2017). DOI 10.1109/SCAM.2017.19
82. Nagy, C., Cleve, A.: SQLInspect: A static analyzer to inspect database usage in Java applications. In: International Conference on Software Engineering (ICSE), pp. 93–96 (2018). DOI 10.1145/3183440.3183496
83. Ngo, M.N., Tan, H.B.K.: Applying static analysis for automated extraction of database interactions in web applications. Information and Software Technology **50**(3), 160–175 (2008). DOI 10.1016/j.infsof.2006.11.005
84. Noughi, N., Mori, M., Meurice, L., Cleve, A.: Understanding the database manipulation behavior of programs. In: International Conference on Program Comprehension (ICPC), pp. 64–67. ACM (2014). DOI 10.1145/2597008.2597790
85. Panas, T., Berrigan, R., Grundy, J.: A 3D metaphor for software production visualization. In: IV 2003, p. 314. IEEE Computer Society (2003)
86. Panas, T., Epperly, T., Quinlan, D., Saebjornsen, A., Vuduc, R.: Communicating software architecture using a unified single-view visualization. In: International Conference on Engineering Complex Computer Systems (ECCS), pp. 217–228. IEEE (2007)
87. Pfahler, F., Minelli, R., Nagy, C., Lanza, M.: Visualizing evolving software cities. In: Working Conference on Software Visualization (VISSOFT), pp. 22–26 (2020). DOI 10.1109/VISSOFT51673.2020.00007
88. Qiu, D., Li, B., Su, Z.: An empirical analysis of the co-evolution of schema and code in database applications. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 125–135. ACM (2013). DOI 10.1145/2491411.2491431
89. Ramasubbu, N., Kemerer, C.F.: Technical debt and the reliability of enterprise software systems: A competing risks analysis. Management Science **62**(5), 1487–1510 (2016). DOI 10.1287/mnsc.2015.2196
90. Reiss, S.P.: An engine for the 3D visualization of program information. J. Visual Languages & Computing **6**(3), 299–323 (1995)
91. Ringlstetter, A., Scherzinger, S., Bissyandé, T.F.: Data model evolution using object-NoSQL mappers: Folklore or state-of-the-art? In: International Workshop on BIG Data Software Engineering, pp. 33–36 (2016)
92. Scherzinger, S., De Almeida, E.C., Ickert, F., Del Fabro, M.D.: On the necessity of model checking NoSQL database schemas when building SaaS applications. In: International Workshop on Testing the Cloud (TTC). ACM (2013)
93. Scherzinger, S., Sidortschuck, S.: An empirical study on the design and evolution of NoSQL database schemas. In: International Conference on Conceptual Modeling (ER), pp. 441–455. Springer (2020)
94. Shao, S., Qiu, Z., Yu, X., Yang, W., Jin, G., Xie, T., Wu, X.: Database-access performance antipatterns in database-backed web applications. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 58–69. IEEE (2020). DOI 10.1109/ICSME46990.2020.00016
95. Sharma, T., Fragkoulis, M., Rizou, S., Bruntink, M., Spinellis, D.: Smelly relations: Measuring and understanding database schema quality. In: International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 55–64. ACM (2018). DOI 10.1145/3183519.3183529
96. Sjøberg, D.: Quantifying schema evolution. Information and Software Technology **35**(1), 35–44 (1993). DOI 10.1016/0950-5849(93)90027-Z

97. Skoulis, I., Vassiliadis, P., Zarras, A.: Open-source databases: Within, outside, or beyond Lehman's laws of software evolution? In: International Conference on Advanced Information Systems Engineering (CAiSE), *LNCS*, vol. 8484, pp. 379–393. Springer (2014). DOI 10.1007/978-3-319-07881-6%5C_26
98. Sonoda, M., Matsuda, T., Koizumi, D., Hirasawa, S.: On automatic detection of SQL injection attacks by the feature extraction of the single character. In: International Conference on Security of Information and Networks (SIN), pp. 81–86. ACM (2011). DOI 10.1145/2070425.2070440
99. Spadini, D., Aniche, M., Bacchelli, A.: PyDriller: Python framework for mining software repositories. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 908–911. ACM (2018). DOI 10.1145/3236024.3264598
100. Stasko, J.T., Brown, M.H., Domingue, J.B., Price, B.A.: Software Visualization - Programming as a Multimedia Experience. MIT Press (1998)
101. Steinbrückner, F., Lewerentz, C.: Representing development history in software cities. In: International Symposium on Software Visualization, pp. 193–202. ACM (2010). DOI 10.1145/1879211.1879239
102. Stonebraker, M., Deng, D., Brodie, M.L.: Database decay and how to avoid it. In: Proc. Big Data, pp. 7–16 (2016). DOI 10.1109/BigData.2016.7840584
103. Stonebraker, M., Deng, D., Brodie, M.L.: Application-database co-evolution: A new design and development paradigm. In: New England Database Day (2017)
104. Storey, M.A., Best, C., Michaud, J.: SHriMP views: An interactive and customizable environment for software exploration. In: International Workshop on Program Comprehension (IWPC) (2001)
105. Storey, M.A., Wong, K., Müller, H.: How do program understanding tools affect how programmers understand programs? In: Working Conference on Reverse Engineering (WCRE), pp. 12–21. IEEE (1997)
106. Störl, U., Klettke, M., Scherzinger, S.: NoSQL schema evolution and data migration: State-of-the-art and opportunities. In: International Conference on Extending Database Technology (EDBT), pp. 655–658 (2020). DOI 10.5441/002/edbt.2020.87
107. Sun, K., Ryu, S.: Analysis of JavaScript programs: Challenges and research trends. ACM Computing Surveys **50**(4) (2017)
108. Tufte, E.: Envisioning Information. Graphics Press (1990)
109. Tufte, E.: Visual Explanations. Graphics Press (1997)
110. Tufte, E.: The Visual Display of Quantitative Information, second edn. Graphics Press (2001)
111. Van Den Brink, H.J., van der Leek, R.: Quality metrics for SQL queries embedded in host languages. In: European Conference on Software Maintenance and Reengineering (CSMR) (2007)
112. Vassiliadis, P., Zarras, A.V., Skoulis, I.: How is life for a table in an evolving relational schema? birth, death and everything in between. In: International Conference on Conceptual Modeling (ER), pp. 453–466. Springer (2015)
113. Vassiliadis, P., Zarras, A.V., Skoulis, I.: Gravitating to rigidity: Patterns of schema evolution, and its absence in the lives of tables. Information Systems **63**, 24–46 (2017). DOI 10.1016/j.is.2016.06.010
114. Vincur, J., Navrat, P., Polasek, I.: VR City: Software analysis in virtual reality environment. In: Int. Conf. Software Quality, Reliability and Security, pp. 509–516. IEEE (2017). DOI 10.1109/QRS-C.2017.88
115. Ware, C.: Information Visualization: Perception for Design, second edn. Morgan Kaufmann (2004)
116. Wassermann, G., Gould, C., Su, Z., Devanbu, P.: Static checking of dynamically generated queries in database applications. Transactions on Software Engineering and Methodology **16**(4), 14 (2007). DOI 10.1145/1276933.1276935
117. Weber, J.H., Cleve, A., Meurice, L., Bermudez Ruiz, F.J.: Managing technical debt in database schemas of critical software. In: International Workshop on Managing Technical Debt, pp. 43–46 (2014). DOI 10.1109/MTD.2014.17

118. Wettel, R., Lanza, M.: Visualizing software systems as cities. In: International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), pp. 92–99. IEEE (2007)
119. Wettel, R., Lanza, M.: CodeCity: 3D visualization of large-scale software. In: International Conference on Software Engineering (ICSE), pp. 921–922. ACM (2008)
120. Yeole, A.S., Meshram, B.B.: Analysis of different technique for detection of SQL injection. In: International Conference & Workshop on Emerging Trends in Technology (ICWET), pp. 963–966. ACM (2011). DOI 10.1145/1980022.1980229
121. Young, P., Munro, M.: Visualising software in virtual reality. In: International Workshop on Program Comprehension (IWPC), pp. 19–26. IEEE (1998)
122. Zirkelbach, C., Hasselbring, W.: Live visualization of database behavior for large software landscapes: The RACCOON approach. Tech. rep., Department of Computer Science, Kiel University (2019)