CodeCritics Applied to Database Schema: Challenges and First Results

Julien Delplanque*†, Anne Etien*, Olivier Auverlot*, Tom Mens†, Nicolas Anquetil* and Stéphane Ducasse*
* Université de Lille, CRIStAL, CNRS, UMR 9189, Lille, France
RMoD Team, Inria Lille Nord Europe
{firstname.lastname}@inria.fr
† Software Engineering Lab, Université de Mons, Belgium
julien.delplanque@student.umons.ac.be, tom.mens@umons.ac.be

***Abstract*—Relational databases (DB) play a critical role in many information systems. For different reasons, their schemas gather not only tables and columns but also views, triggers or stored functions (*i.e.,* fragments of code describing treatments). As for any other code-related artefact, software quality in a DB schema helps avoiding future bugs. However, few tools exist to analyse DB quality and prevent the introduction of technical debt. Moreover, these tools suffer from limitations like the difficulty to deal with some entities (*e.g.,* functions) or dependencies between entities. This paper presents research issues related to assessing the software quality of a DB schema by adapting existing source code analysis research to database schemas. We present preliminary results that have been validated through the implementation of *DBCritics*, a prototype tool to perform static analysis on the SQL source code of a database schema. *DBCritics* addresses the limitations of existing DB quality tools based on an internal representation considering all entities of the database and their relationships.**

Keywords*—database software quality, database critics, design smells, quality assessment

I. Introduction

Relational databases represent the heart of numerous information systems. Information Systems have been defined as "the software and hardware systems that support *data-intensive* applications"[1]. These systems query, update, and present large sets of data with treatments that can reside in the source code of the applications accessing the database (DB), or be directly included in the DB schema (*e.g.,* views or stored functions). These systems and their databases have often been built decades ago. Their requirements changed continuously, leading to evolution in the application code but also in the database structure (which again impacted the application code). After years of evolution, the quality of the database structure and the behaviour it includes may have deteriorated, resulting (as for the application) in a difficulty to perform further evolution or in increased risk of future errors. For example, a query using `SELECT *` in a database view will easily accept any change in the underlying table definition (addition or removal of a column), but can result in an execution error when the result of the query is manipulated.

The usual answer to DB evolution is to rely on metadata describing the DB schema. But these metadata do not consider, for example, the body of internal functions nor the requests building views. As a consequence the issues described above (similar to code smells), cannot be detected. Yet they may have important impact on the application using the DB. Quality analysis should therefore go beyond metadata only, and also consider internal treatments (*e.g.* stored procedure, triggers).

This paper adopts a software engineering approach by considering a DB similarly to a program and proposing to adapt existing software quality tools and techniques to it. Such an approach should be language dependent as is the case for most static analysis techniques in reverse engineering. It requires to reverse engineer the DB schema and to analyse its properties. Adaptation of software engineering approaches to DB have already shown their efficiency [4], [9], [10] but these approaches never take into account views and functions and do not detect smells.

It is important to stress that this paper does not consider the quality of the DB in terms of query efficiency or data consistency, but in terms of quality of the code they embed. This paper proposes to assess the database software quality for example to minimize the technical debt of a DB schema. We present some first results through the description of a research prototype tool, called *DBCritics*, that addresses these issues.

The remainder of the paper is organized as follows. Section II presents some scenarios in which assessing DB schema quality is relevant and highlights some underlying issues. Section III describes our tool. Section IV evaluates the usefulness of the tool on two DB. Section V positions our tool against other existing solutions. Finally, Section VI concludes this paper and discusses some perspectives.

II. DB CodeCritics Usage Scenarios and Underlying Issues

This paper proposes a code critics approach and tool dedicated to DB schemas. The purpose is to *assess the code quality of an existing DB schema* through static analysis. It is the structure and treatments of the schema itself that will be studied, and not the code of the application using the DB. In this section, we present three scenarios in which code critics can be used on a DB schema, and we highlight some underlying issues that such an approach raises.

[1] From https://www.journals.elsevier.com/information-systems/ with our emphasis. Last consulted on 5 January 2017

A. Detecting Smells in Database

Database administrators (DBA) need tools to highlight smells, anti-patterns and violations of business rules. The spaghetti query antipattern [8] aims to detect queries that are too complex to understand, maintain or debug. Some naming convention could need to be checked like prefixing all key columns names with `k_`. This first example is a *generic* smell applicable to any database and even to any database management system (DBMS). In contrast, *company-specific* or *DB-specific* smells depend on their domain and their use. A DB code critics checking tool can provide a snapshot of the DB schema quality, and could be used before and/or after each commit to detect possible deterioration of the DB quality.

B. Migrating from a DBMS Version to Another One

DBMS (*e.g.,* PostgreSQL, MySQL or Oracle) are constantly evolving and new versions are regularly released to introduce new features or to fix bugs. In theory, the new version of a DBMS should be backward compatible with older versions. In practice, it may happen that the behaviour or name of an instruction has been changed from one version to the next. Upgrade migration patches are rarely provided due to the risk of breaking a database and all the applications that rely on it. In case of open source DBMS like PostgreSQL, new releases only come with a textual change log. The task of DB schema migration is left to the DBA. He must identify what impact the changes will have on his schema, and correct them accordingly. For example, PostgreSQL documents that for versions after 9.0, PL/PgSQL variables will take preference over a table or view column with the same name[2]. If behaviour of the same code was changed after migration, this may lead to errors. In this case, detecting occurrences of variables with the same name that columns used in requests in the same function may prevent future errors.

C. Maintaining Consistency between Different Forks of a Database Schema

A DB schema may be used as a basis for multiple software projects, each one adapting the schema to its needs. For example, the tool corresponding to the DB schema of our research laboratory managing members, teams or funding is shared with another laboratory, together with all the applications using the DB. Each laboratory has its own DB based on the initial schema. However, some modifications have been performed on the initial schema to adapt the DB to the needs of new users. The laboratory that adopted the tool also benefits from maintenance to accommodate new features and/or bug fixes. Each change in the master DB needs to be ported to the slave DB with the risk that both DB continue to evolve separately, thus drifting further apart. Ensuring naming convention between forks reduces maintenance efforts.

D. Underlying Issues

Based on these scenarios, general issues relative to the assessment of DB schema quality can be highlighted. First, a DB schema contains table and column descriptions but also views, functions, sequences, triggers and constraint definitions. All these entities and the relationships between them are potentially subject to quality defects.

Second, it has been shown that checking for domain-specific or system-specific smells in software provides better defect prevention than checking for generic smells [7]. We expect the same to happen with DB schema smells.

Finally, automatic detection of quality problems is important, but the ultimate goal is to resolve them, preferably in an automatic way. Moreover, resolving an issue on an entity may imply changes on other entities. Renaming a column `c1` in a table `tA` that is used by a view `vA` implies to change the definition of `vA` (*i.e.,* drop it and create a new version). Furthermore, if `vA` is used in another view `vB` the second also needs to be temporarily dropped and then recreated possibly exactly as it was before if it does not use `c1`. This change impact depends on the entities and the changes. Such impact analysis is an issue by itself.

III. DBCritics

This section presents our approach to statically analyse database schema quality and avoid future quality problems. It is implemented in a prototype tool called *DBCritics*.

A. DBCritics Overview

DBCritics takes into account *any* type of DB entity: tables, views, columns, functions, triggers, indexes, constraints, etc. *DBCritics* takes as input an abstract representation of the DB schema (including its structure and treatment), conform to a PL/SQL (Procedural Language / Structured Query Language) meta-model that extends the *FAMIX* meta-model [6].

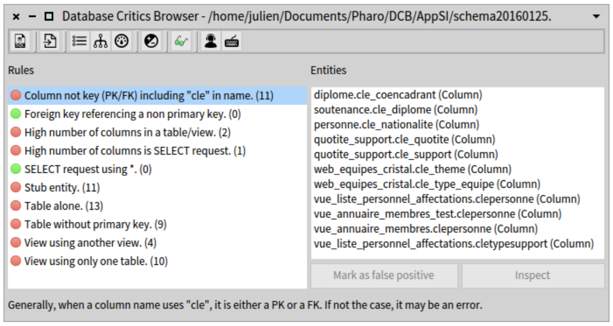
Each DBMS defines and uses its own PL/SQL. For example Oracle and PostgreSQL syntax are very close but nevertheless slightly different. Currently, *DBCritics* supports PostgreSQL. The generation of a model relies on the parsing of a PostgreSQL dump of the DB schema. Using a dump guarantees a certain degree of syntactic validity of the schema and also allows to focus on a standard way to define the schema. For example, with PostgreSQL, primary key definitions are not specified in the `CREATE TABLE` requests but are separated in an `ALTER TABLE` request.

Concretely, *DBCritics* allows a user to select and evaluate rules on a model of the DB schema. The result of the evaluation is stored in a *report* that includes the state of each rule evaluation (violation or success). Figure 1 shows a screenshot of the graphical user interface of *DBCritics* main widget.

B. Rules

DBCritics relies on the notion of *rules*. A rule describes a property the DB schema should satisfy. Some rules are mandatory because they identify plain errors. Some rules are optional as they identify quality problems that may hamper future maintenance or evolution. A severity criterion allows to classify the rules, helping the user to concentrate his correction efforts. Three severity levels are provided by default: information, warning and error. Others may be added by the user if needed.

[2] https://wiki.postgresql.org/wiki/What' s_new_in_PostgreSQL_9.0



Database Critics Browser - /home/julien/Documents/Pharo/DCB/AppSI/schema20160125.

- ● Column not key (PK/FK) including "cle" in name. (11)
- ● Foreign key referencing a non primary key. (0)
- ● High number of columns in a table/view. (2)
- ● High number of columns is SELECT request. (1)
- ● SELECT request using *. (0)
- ● Stub entity. (11)
- ● Table alone. (13)
- ● Table without primary key. (9)
- ● View using another view. (4)
- ● View using only one table. (10)

```
diplome.cle_coencadrant (Column)
soutenance.cle_diplome (Column)
personne.cle_nationalite (Column)
quotite_support.cle_quotite (Column)
quotite_support.cle_support (Column)
web_equipes_cristal.cle_theme (Column)
web_equipes_cristal.cle_type_equipe (Column)
vue_liste_personnel_affectations.clepersonne (Column)
vue_annuaire_membres_test.clepersonne (Column)
vue_annuaire_membres.clepersonne (Column)
vue_liste_personnel_affectations.cletypesupport (Column)
```

Generally, when a column name uses "cle", it is either a PK or a FK. If not the case, it may be an error.

Fig. 1. The *DBCritics* GUI. Left panel: Rules that have been run on the model. Right panel: Entities violating the selected rule.

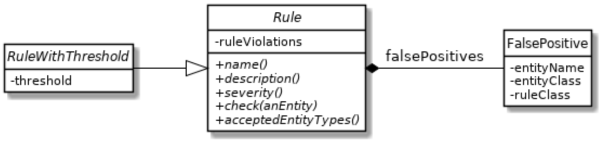


Fig. 2. UML class diagram of the rule implementation in *DBCritics*.

Figure 2 presents the concept of rule as defined in *DBCritics*. Each rule holds the list of entities violating it. This list can be reviewed by the user to mark false positives that are also stored to ignore future detections of these violations.

The `acceptedEntityTypes()` method specifies the entity types that the rule checks. The method `check(anEntity)` specifies the violation constraints. The abstract `Rule` class has been extended to be able to define rules using a threshold (`RuleWithThreshold`). These rules can be parametrized by the user to fit his needs. For example, a `Table` is considered too big if it has $\geq n$ columns, where n is specified by the user.

C. Examples of Rules

Next to the severity criterion, other means exist to classify rules, such as the entity type on which the rule is applied. Some rules are generic (*i.e.,* applicable to any DB) or specific (*i.e.,* only applicable to a particular DB or DBMS). Some rules concern the code, while others focus on the entity structures and the relationships between entities.

DBCritics is a research prototype developed in Pharo[3]. A first set of rules has been defined with a focus on diversity (*e.g.* focus on code, entity, relationship, or severity, ...). Rules are not prepared automatically. They have been defined by extending either `Rule` or `RuleWithThreshold` and overriding the methods presented in the previous section. These rules have been created based on a combination of the experience of DBA, related works and adaptation of existing rules for code smells detection.

Rule 1: *Detect use of* $\star$ *in* `SELECT` *request.* Using $\star$ in the request selects all the columns of the used tables. The structure of the request result thus changes after column addition or removal in one of these tables, what can cause problems to the program using it.

Rule 2: *Foreign key referencing a non primary key*. Uniqueness of the reference is not guaranteed and leads to semantic errors.

Rule 3: *Too many columns in SELECT request*. This rule identifies queries that may be complex to maintain (spaghetti query anti-pattern).

Rule 4: *Table without primary key*. A table should always have a primary key.

Rule 5: *Column not key (PK/FK) using the name convention for key (*e.g.*"k_" in name)*. If a naming convention exists, it is as important to use it for key columns as not to use it for non-key columns.

Rule 6: *Stub entities* are used but not defined in the DB schema. This rule detects for example the call of functions not defined in the schema either intentionally if they correspond to system entity like `pg_class` table or `count()` function or involuntarily if the name of the function is misspelled or a removed function is still called.

Rule 7: *Isolated table*. A table that is referenced by no entity and does not use any table cannot be accessed through natural join (based on foreign keys). It is certainly not used, or with other criteria that foreign keys what can lead to semantic errors.

Rule 8: *Unused functions* could be removed (if it is not used by an external program). This rule does not detect issue but may help in the cleaning of the schema.

Rule 9: *View using another view* is a poor design for the future evolution of the database. For example, if a view `vA` uses a view `vB` and `vB` needs to be modified, `vB` has to be deleted as well as `vA` and both views need to be recreated in an order satisfying usage dependencies between the views.

Rule 10: *View using only one table* has no reason to exist since a simple `SELECT` request can do the job.

Rule 11: *Too many columns in a table* may be an illustration that the table has several concerns and should be decomposed.

Rule 5 is specific to our concrete example. The other rules are generic. Rule 3 and 11 have a threshold that must be defined by the user. Rule 4 represents an error because it means that the DB schema is not in the first normal form. Rule 6 can correspond to an error if the undefined entity is not a system one. All other presented rules correspond to warnings or information. Rules 1 and 3 focus on the code whereas all others focus on entities and their relationships.

D. False Positives

Sometimes, rules can be too strict: some issues can be acceptable in certain contexts. For example, referencing the `pg_class` while using inheritance between tables is normal. Yet, it will appear as a violation of rule 6. It may also happen that a DBA voluntarily leaves known smells in the schema because of lack of time or too high level of risk to fix them. In *DBCritics*, these bad smells can be tagged as false positives for a given rule.

IV. Case Studies

This section evaluates the usefulness of *DBCritics* on two case studies: the PostgreSQL version of the WikiMedia database [2] and, AppSI, a database used at the Université de

[3] http://pharo.org

Lille. Table I summarises the sizes of these databases with the minimum and maximum number of entities of each kind over all analysed versions.

WikiMedia is an open source collaborative editing software project that runs Wikipedia. It currently has three versions of its database schema for three DBMS: MySQL/MariaDB, PostgreSQL and SQLite. We analysed 25 different versions of the PostgreSQL database schema representing the different versions available on Github [1] modified by 23 contributors.

AppSI aims to manage members, teams, funding support, etc. in departments of the Université de Lille. It is a proprietary DB developed by a single DBA. We analysed 12 consecutive versions of the database schema.

TABLE I. MIN/MAX NUMBER OF ENTITIES PER TYPE AND MIN/MAX LINES OF CODE FOR EACH DATABASE.

	Tables	Columns	Views	Functions	Triggers	LOC
WikiMedia	30/51	196/353	0/1	3/5	2/3	1,435/2,453
AppSI	71/91	583/974	30/52	46/67	12/16	4,910/7,006

DBCritics has been used to analyse the quality of the different versions of these two DBs. Three aspects were analysed: the number of rule violations per version; the proportion of "violating entities" (*i.e.,* entities that violate at least one rule); and the "time-to-fix" of a rule violation (only for violations that actually get fixed). For this experiment, we applied the rules described in the previous section. WikiMedia had 67 violations on average over all considered versions, while AppSI had 76 violations on average.

A. Violation Count Per Version

Figure 3 shows the evolution of the number of violations (*i.e.,* unique pairs (entity, rule)) over time. We observe that the number of violations tends to increase (from 37 to 66 for WikiMedia and from 54 to 87 for AppSI). A possible explanation would be that contributors are unaware of these violations because of the lack of tools similar to *DBCritics*.

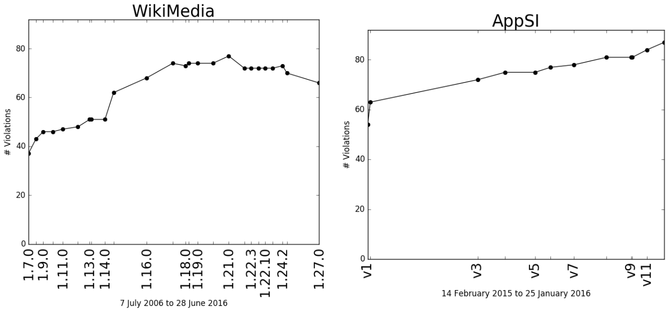


Fig. 3. Violation count per version for WikiMedia and AppSI.

B. Violating Entities Proportion

Figure 4 shows the proportion of violating entities (black bars) against the total number of entities (grey bars) in each DB schema. On average, WikiMedia has more rule violations than AppSI (respectively 14% and 6%). The proportion of violating entities is globally stable over time (whereas the total number of entities is increasing). More analyses on more databases are required to check this impression.

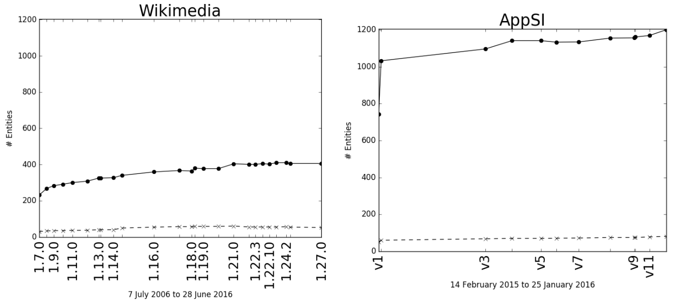


Fig. 4. Violating entities (dashed) proportion against entities count (solid) per version for WikiMedia and AppSI.

C. Time-to-fix of a Rule Violation

Only 3 of the 85 rule violations were corrected in AppSI on the 12 versions analysed, and 21 of the 87 rule violations for WikiMedia. Table II summarises the "time-to-fix" in number of days for those rule violations that got resolved. The table reveals that corrections occur faster for AppSI than for WikiMedia. An interpretation would be that when the unique DBA of AppSI is aware of a violation, he corrects it very quickly, since he knows his DB very well. When several contributors work on the same DB, knowledge is shared and more diffused.

TABLE II. MINIMUM, FIRST QUANTILE, MEDIAN, THIRD QUANTILE AND MAXIMUM OF THE "TIME-TO-FIX" OF RESOLVED RULE VIOLATIONS.

	Min	First quantile	Median	Third quantile	Max
WikiMedia	95	1227	1833	2403	3644
AppSI	3	/	125	/	278

These three experiments show that (1) there are rule violations in real life DBs; (2) their number increases over time with the number of entities; and (3) only few violations are fixed. A tool like *DBCritics* is thus needed to help in the violation correction. A deeper comparison between open-source and closed DB would be relevant. Moreover, it would be interesting to check whether the number of contributors really impacts the speed at which a rule is corrected.

D. Discussion About False Positives

Three categories of violations can be distinguished: (i) real design issues that require modifications of the schema, (ii) issues that the DBA considers correct despite the rule violation and (iii) issues due to limitations of DBCritics. Concerning the second category, in AppSI, table $person$ contains 26 columns and violates rule 11 for which the threshold is 25. However, this table contains 4 columns corresponding to computed values that are saved in the DB for performance reasons. Concerning the third category, *DBCritics* cannot yet manage views appearing in the schema as a table with an associated rule[4]; they are considered as tables by the tool although they are views. Some false violations are caused by these restrictions.

We have discussed with the DBA of AppSI to examine in detail the violations on one arbitrarily chosen version of its schema to get an idea of the proportion of each of these three

[4] https://www.postgresql.org/docs/9.5/static/rules-views.html

categories. In version v10, there were 81 rule violations, 51 fell in the first category, 8 in the second and 22 in the last one. Even if this proportion cannot be extrapolated to all versions of AppSI or any database, it gives a first idea. Deeper analysis should be done to generalise this result.

V. Related work

In the context of traditional programs, several approaches have been proposed and tools implemented to detect bad smells and avoid possible future bugs, like Lint[5] for *C* source code analysis and its variants for others programming languages: PyLint[6] for Python, JSLint[7] for Javascript, etc.

Normal forms ensure good practices in term of relational database design [5]. They describe how to organize tables, columns and primary/foreign keys to avoid data redundancy. It is relevant to check that they are still satisfied after schema evolutions. To check them, specifications, lists of functional dependencies or the way data is inserted in tables are needed. Unfortunately, such information is rarely embedded in the schema. Only data insertions or updates through triggers (*i.e.,* functions that are automatically launched before or after other insertions or modifications in the database) are present in the schema and may help to detect normal form violations by computed data. Most other normal form violations cannot be detected by static analysis of the DB schema. While they are certainly relevant, they require external information that cannot be obtained by reverse engineering.

Weber *et al.* [11] studied the foreign key (FK) technical debt on the OSCAR Electronic Medical Record system and proposed to use the technical debt metaphor for developing processes related to missing FKs implementation. This work is related to ours in the sense that it aims to use software engineering methods on relational databases. But exactly as for normal forms, only data are used to measure the FK technical debt.

The concepts of smells and anti-patterns have also been used for databases [8]. Some software exists to detect database smells and anti-patterns. For example *SonarQube*[8] has a PL/SQL plugin [3] which allows to create rules that check the AST of the SQL source code using XPath queries. In contrast to *DBCritics*, relationships between entities are not taken into account.

Al-Barak and Bahsoon [4] did a work similar to the one in this paper by exploring the database design debt concept and manually analysing successive versions of MediaWiki's database schema. *DBCritics* aims to automate the process of database smell/debt detection when the model allows it. This automation allows one to check for smells regularly in a limited amount of time. Moreover, *DBCritics* also takes functions into account.

VI. Conclusion

Relational databases are at the core of many information systems. As any other artefact, their schemas evolve with time to implement new requirements or to resolve bugs. However, these evolutions can introduce new errors or smells in the database schema. In this paper we presented different scenarios where the analysis of database schema is relevant. We also introduced *DBCritics*, a tool to detect errors, smells or anti-pattern by applying software engineering methods. More specifically, SQL schemas are statically analysed and a set of rules that can be parametrized are run. *DBCritics* takes into account any kind of entity in the schema: tables, columns, views, triggers and functions. As a preliminary evaluation, we studied multiple versions of two DBs: WikiMedia and AppSI. This analysis shows that (i) rule violations can be found in open source as well as in proprietary DB schemas; (ii) the number of violations evolves with the number of entities; (iii) on both DBs some violations are fixed but not all of them.

Future work is twofold. First, we aim to provide more detailed empirical studies to evaluate for example (i) the difference of open-source and proprietary DB schemas; (ii) the relation between the number of contributors and the time to fix rule violations; (iii) the relation between the number of rule violations and the number of bugs in a DB schema and; (iv) the impact of using *DBCritics*. Second, next to the automatic detection of violations, we also aim to support automatic correction of them. Such an automation would require impact analysis of each correction and the effect of cascaded corrections.

Acknowledgment

This research was partially financed by FRFC research project T.0022.13 of the F.R.S.-FNRS, Belgium.

References

[1] Mediawiki github repository. https://github.com/wikimedia/mediawiki. Accessed 2016-11-26.

[2] Mediawiki website. https://www.mediawiki.org. Accessed 2016-11-26.

[3] Sonarqube plsql plugin website. https://www.sonarsource.com/why-us/products/languages/plsql.html. Accessed: 2016-11-16.

[4] Mashel Al-Barak and Rami Bahsoon. Database design debts through examining schema evolution. In *Int'l Workshop on Managing Technical Debt (MTD)*. IEEE, 2016.

[5] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[6] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why FAMIX and not UML. In *Int'l Conf. Unified Modeling Language (UML)*, volume 1723, 1999.

[7] Andre Hora, Nicolas Anquetil, Stéphane Ducasse, and Simon Allier. Domain specific warnings: Are they any better? In *Int'l Conf. Software Maintenance (ICSM)*, pages 441–450. IEEE, 2012.

[8] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 2010.

[9] Loup Meurice, Csaba Nagy, and Anthony Cleve. Static analysis of dynamic database usage in java systems. In *Int'l Conf. Advanced Information Systems Engineering (CAiSE)*, volume 9694 of *Lecture Notes in Computer Science*, pages 491–506. Springer, 2016.

[10] Csaba Nagy, Loup Meurice, and Anthony Cleve. Where was this SQL query executed? a static concept location approach. In *Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, pages 580–584. IEEE Computer Society, 2015.

[11] Jens H Weber, Anthony Cleve, Loup Meurice, and Francisco Javier Bermudez Ruiz. Managing technical debt in database schemas of critical software. In *Int'l Workshop Managing Technical Debt (MTD)*, pages 43–46. IEEE, 2014.

[5] www.unix.com/man-page/freebsd/1/LINT/

[6] https://www.pylint.org/

[7] http://jslint.com/

[8] http://www.sonarqube.org