SQLInspect: A Static Analyzer to Inspect Database Usage in Java Applications

Csaba Nagy
University of Namur, Belgium
Università della Svizzera italiana (USI), Switzerland
csaba.nagy@unamur.be

Anthony Cleve*
University of Namur, Belgium
anthony.cleve@unamur.be

ABSTRACT

We present *SQLInspect*, a tool intended to assist developers who deal with SQL code embedded in Java applications. It is integrated into Eclipse as a plug-in that is able to extract SQL queries from Java code through static string analysis. It parses the extracted queries and performs various analyses on them. As a result, one can readily explore the source code which accesses a given part of the database, or which is responsible for the construction of a given SQL query. SQL-related metrics and common coding mistakes are also used to spot inefficiently or defectively performing SQL statements and to identify poorly designed classes, like those that construct many queries via complex control-flow paths. *SQLInspect* is a novel tool that relies on recent query extraction approaches. It currently supports Java applications working with JDBC and SQL code written for MySQL or Apache Impala. *Check out the live demo of SQLInspect at http://perso.unamur.be/~cnagy/sqlinspect.*

KEYWORDS

MySQL, Apache Impala, Embedded SQL, Static Analysis, Metrics, Bad Smells, Concept Location, Java, JDBC, Eclipse

ACM Reference Format:
Csaba Nagy and Anthony Cleve. 2018. SQLInspect: A Static Analyzer to Inspect Database Usage in Java Applications. In *ICSE '18 Companion: 40th International Conference on Software Engineering, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183496

1 INTRODUCTION

SQL is the query language of mainstream relational databases, but it is also used by new-generation distributed database management systems. It was number two on the list of most popular programming languages in the 2017 survey of StackOverflow[1]. For comparison, JavaScript was the first and Java the third on the same list. In Java, a standard way to communicate with the DBMS is to send SQL statements to the database via JDBC and then process the answer with the result. This either requires the developer to write the queries and to *embed* them as strings in the source code, or to use a higher-level API like an ORM which translates the database accesses to SQL. It is quite rare that an SQL statement is just statically embedded in the code. More frequently, it is constructed dynamically via string operations on variables scattered throughout different source code locations. In the end, the statement is rarely visible to the developer in its final form as it is sent to the database. An error report may occur that will make it visible, but it would probably be too late. This query embedding mechanism makes it difficult to work with the database and to avoid the construction of erroneous or inefficient queries.

One problem is *dynamicity*. That is, to collect the SQL statements of dynamic string operations, they have to be tracked down during execution (e.g., by dynamic analysis), which is not always feasible during the development and it limits the analysis to a single execution. Using a static analyzer would reap benefits, but it cannot resolve corner-cases of dynamically assembled strings. Another problem is the *weak tool support* of the code interacting with the database. Understandably, DBMSs limit their tool support to work within the boundaries of the database. They cannot afford to invest much effort in the development of the client side; likewise, the situation is the same for client applications. As a result, it is the developer who has a hard time managing the SQL code in between.

Third parties have realized this problem, and now there are some tools available. For example, *XRebel*[2] supports the tracking of SQL and NoSQL queries so one can explore the number of affected rows and required execution time for each database access. For this purpose, it *dynamically* catches the SQL strings sent to the database. The *Eclipse Data Tools Platform*[3] provides an environment for Eclipse to work with data-centric systems. Its *SQL Development Tools* package assists in editing SQL through code completion, formatting, and dialect specialization. One can explore the execution plan of a given query too. It makes it easier to work with SQL inside the IDE, but *does not handle the SQL embedded in the code.* The only tool designed for this is *Alvor* [2], developed as an Eclipse plug-in. It statically evaluates strings passed to JDBC methods for subsequent execution and to check syntax/semantics correctness or object availability. Alvor was a Google code project, but it seems to have been discontinued, and its functionality remains *limited to checking the semantic correctness of the extracted queries.*

> *SQLInspect is a novel tool for statically extracting SQL queries embedded in Java and performing various analyses on them. It seeks to provide support for developers to accomplish SQL-related maintenance tasks in the code, where it encounters data.*

*This work was supported by the University of Namur and by the Fonds de la Recherche Scientifique-FNRS under EOS Project 30446992 SECO-ASSIST.

[1] https://insights.stackoverflow.com/survey/2017

[2] https://zeroturnaround.com/software/xrebel/

[3] https://www.eclipse.org/datatools/

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183496

2 USE CASES

Our aim was to combine *state-of-the-art* software analysis techniques in *SQLInspect* and provide a compelling, working solution for developers to assess the Java code using embedded SQL. This is very helpful for data-centric applications where higher level APIs are avoided in favor of efficiency or because of limited resources, e.g., distributed database applications or Android development.

For this purpose, (I) we implemented a *static control-flow based SQL extraction* technique that is configurable and able to deal with Java applications that have hundreds of thousands of lines of code. (II) The queries extracted are then analyzed by an *internal parser* for *MySQL* and *Apache Impala*, which can *tolerate unresolved code fragments*, i.e., parts of string operations that cannot be statically resolved due to the dynamicity problem. (III) It *resolves identifiers of database objects* to show the slice of the source code working with given schema objects. (IV) SQL quality *metrics* are provided for database accesses so one can assess methods and classes that pester the database too much or complex queries which handle lots of schema elements. (V) SQL *bad smells* are also detected, e.g., common coding mistakes within the extracted queries. (VI) A *tree-matching algorithm* enables the developer to perform a configurable *search for a given query* within the entire source code. Should a query show up in an error report, one can search for its location in the source code. This search can be configured so that some parts of the query are ignored, e.g., literal values or identifier names. Below, we elaborate on the main use cases.

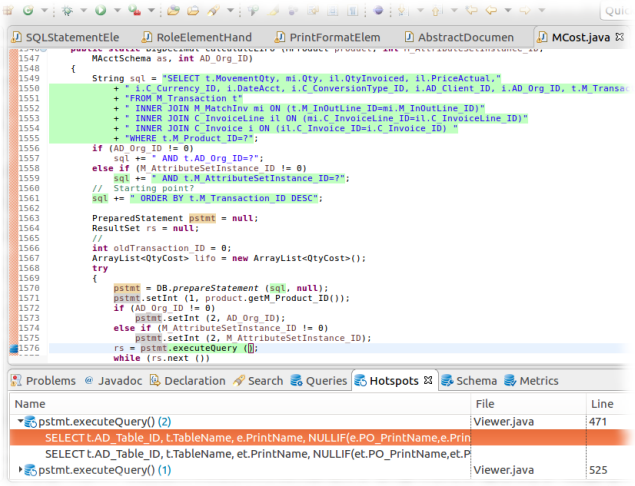

Figure 1: A screenshot of *SQLInspect* with a query selected in the editor.

Query Inspection. To help us better understand the SQL statements and the way they were constructed, *SQLInspect* takes advantage of the features of Eclipse. *Hotspots*, the locations in the source code where the queries are sent to the database, are marked with *markers*. Hence, if a statement sends a query to the database, a marker will indicate this in the editor. Other source code elements taking part in the construction (e.g., variables, string concatenations) are also annotated. Therefore, the whole slice responsible for the creation of the query can be highlighted (see Figure 1). This allows the developer to readily debug or modify the query and the source code involved in its construction.

Query Search. It is not unusual for information systems to have to work with hundreds of tables and pester the database with several thousands of different SQL queries coming from different source code locations. If a query causes an error on the database side, it can be extremely hard to track it down in the Java classes. Imagine, for instance, when a DBA spots a `SELECT` statement causing heavy database load. But it just appears randomly from time-to-time, and the application does not report an error message. On the database side, it is only the `SELECT` that is visible for the DBA, and its source code location remains unknown. If we do not get an exception or other indication of the misbehaving query on the client side, it can be quite hard to locate it.

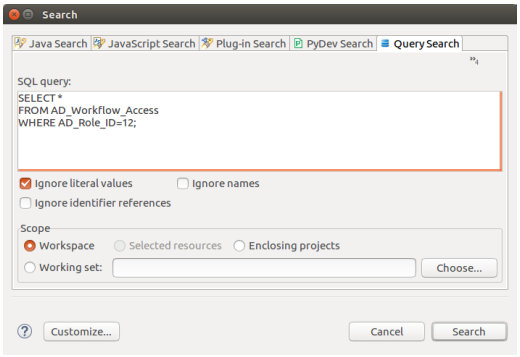

Figure 2: The *query search* feature is integrated into the search page of Eclipse.

We implemented a query search feature in *SQLInspect* to address such a situation. A search can be initiated under the search menu of Eclipse just like a normal text or Java search, and the results will show up in a tree view. The query given here is parsed by the internal SQL parser, and a tree matching algorithm looks for statements in the code with similar syntax trees. This query matching can be configured so as to ignore literal values or identifier names. The algorithm also handles the case where a query in the source code cannot be fully resolved. More details about this algorithm can be found in our early research achievement paper [9].

Table/Column Access Analysis. A more general use case is when one wants to explore the classes and methods in the source code from which a given part of the database is accessed. If the schema description is available, *SQLInspect* can parse it and resolve the identifiers in the SQL. With this information, it is possible to determine which Java methods access specific tables or columns. *SQLInspect* provides a view to browse the database schema, and a search for methods accessing selected schema elements can be easily launched from it. This is an effortless way of seeing which part of the source code is responsible for handling specific database tables or columns.

Embedded SQL Bad Smells. To provide guidance on the improvement of SQL queries, *SQLInspect* implements algorithms to identify SQL bad smells based on the query antipatterns defined in the *SQL Antipatterns* book of Bill Karwin [5]. We rely here on our earlier bad smell detection method, which also motivated us to improve and extend our work, where we give more details on these bad smells [8].

Embedded SQL Quality Metrics. *SQLInspect* has information both on the structure of a query and on the source code responsible for constructing it. This enables us to implement a wide range of quality metrics to identify problematic or poorly designed classes and SQL statements. For example, one can investigate which classes are responsible for the majority of queries, the most complex queries, and the queries using many joins. To this end, quality metrics for embedded SQL queries have been implemented following Brink et al. [3]. These metrics can be explored in a view or they can be exported for further investigation in XML format.

3 IMPLEMENTATION DETAILS

SQLInspect was developed using Java technologies for *Eclipse Oxygen* (4.7). Its overall design is illustrated in Figure 3. Here, there are two main components. The *SQL Analyzer* is responsible for performing all the analyses related to the SQL code, and the Java-related analyses are implemented in an *Eclipse plug-in*. While *SQLInspect* is seamlessly integrated into Eclipse, both key components have command-line interfaces for debugging or scripting purposes.

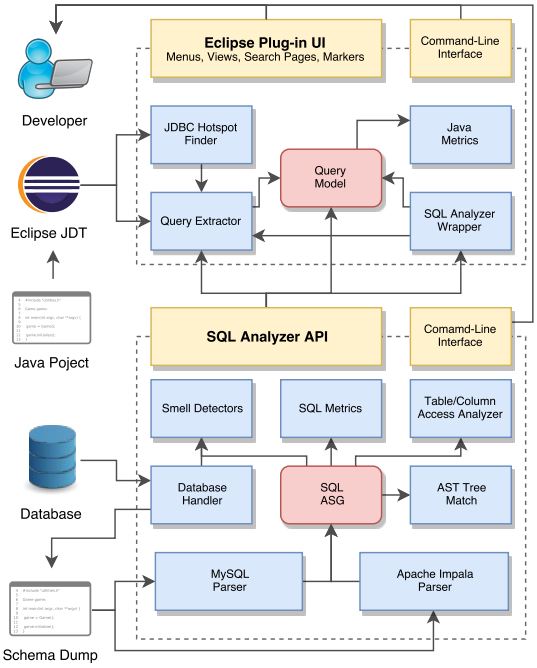

Figure 3: Design overview of *SQLInspect*.

The typical workflow takes place within the IDE. It starts with an analysis that can be initiated from the main menu or the context menu of the Eclipse project. The results can then be investigated in different *views*. Some problem *markers* and ruler *annotations* are created in the standard way in Eclipse, to indicate the location of a warning. We also created *search pages* under the *Search menu* of Eclipse. What is more, *SQLInspect* can be configured with project-specific settings via standard *property pages*.

When an analysis has been executed, *SQLInspect* processes all the compilation units of the Java project. *Eclipse JDT* API calls are invoked to parse the Java classes, and then the queries are extracted in two main steps. (I) First, we look for API calls sending SQL queries to the databases. These are called *hotspots* following the terms of Alvor [2] and JSA[4]. For JDBC, this is implemented by the *JDBC Hotspot Finder* (see Figure 3) using the *Visitor* pattern provided by JDT. While traversing the AST, it looks for given method calls like the `Statement.execute()` call. It also handles the usage of `PreparedStatements`, where the SQL statement is first 'prepared' in an object for execution. It looks for the definitions of these objects where the SQL string is provided as a parameter. (II) In the second phase, the *Query Extractor* takes all the hotspots and extracts all the SQL strings passed to them.

The *Query Extractor* implements a control-flow path-sensitive query extraction algorithm following the method introduced in [7]. However, to integrate it into the IDE, we implemented this approach on the Eclipse AST and made it configurable to meet project-specific requirements. Once a hotspot is identified, we recursively resolve the query expression. This resolution follows the control-flow in a backward direction and terminates if an expression is a constant expression according to the JLS2 specification[4], or if the analysis reaches given thresholds (e.g., max depth or number of branches). During the resolution, a *Query Model* is constructed in a tree. It specifies relations between Java elements that build the fragments of the query. Lastly, the potential query strings are generated from this model.

The resulting strings are then given to the *SQL Analyzer* along with the schema dump (if it was provided). Its *MySQL* and *Apache Impala Parsers* are written using ANTLR 4.7. Both parsers construct an AST defined by a common SQL model, and at the end of an analysis session, an identifier resolution is performed on this AST. The resulting *SQL ASG* (Abstract Semantic Graph) serves as input of the *Smell Detector* algorithms, *SQL Metrics* and the *Table Access Analyzer*. These analyzers are implemented as visitors, each one realizing a single pre-order traversal of the SQL AST.

4 EVALUATION

We tested the performance of *SQLInspect* and evaluated the results on open-source Java systems, relying on JDBC as their main database access library. *ADempiere* is an ERP system under active development with 3,422 downloads in October 2017. It was forked from *Compiere* in 2006. Therefore, the origin of the code is the same but it evolved over a decade independently. *Plandora* is a medium-sized project management system, and *Plazma* is an ERP and CRM. All of the projects can be downloaded from *https://sourceforge.net*.

Table 1 shows the results of the performance benchmark. These tests were performed on an *Intel(R) Core(TM) i5-6300 CPU @ 2.40GHz* machine with 16GB RAM. Eclipse was run with a limit of 1024MB maximum heap size (`-Xmx`), and for the query extraction, we used our intra-procedural path-sensitive implementation limited to maximum 100 CFG branches with a maximum nesting level of 10. In the table, *LOC* refers to the lines of code (excluding blank lines and comments), *Extr.*, *Pars.* and *An.* show the time needed for the SQL extraction, the SQL parser (with identifier resolution) and the remaining analyzers (table/column access analysis, bad smell detectors and metric calculations). The *Mem.* column shows the total memory consumption of Eclipse during the analysis. All the values are averages of five executions. For the sake of comparison, we included the measurements of Annamaa et al. [2], as Alvor was

[4]https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html

evaluated on Compiere and Plazma too. It can be seen that the systems have evolved since 2010. Still, the figures are impressive. An important difference however is that for memory usage, we measured the total memory consumption for our whole analysis process, which includes more analysis steps than those for Alvor.

Project	LOC	Time (s)			Memory
		Extr.	Pars.	An.	(Mb)
Adempiere	797,482	157.9	38.8	0.2	1,969
Compiere	400,383	78.0	24.8	0.2	1,359
Compiere[Alvor]	319,570	120.0	10.4	-	445*
Plandora	93,777	14.2	7.3	0.7	97
Plazma	182,771	52.1	4.1	0.0	38
Plazma[Alvor]	48,520	3.8	0.5	-	64*

Table 1: Benchmark results on open-source systems

Project	No of. DB Acc. Classes	No. of Hotspots	No. of Queries	Max. of Queries
Adempiere	514	1,978	3,579	128
Compiere	480	1,164	2,844	512
Compiere[Alvor]	-	1,343*	-	-
Plandora	62	424	571	72
Plazma	24	140	146	3
Plazma[Alvor]	-	94*	-	-

Table 2: Statistics of hotspots on the benchmark systems

Table 2 provides statistics of the hotspots identified and the queries extracted. The first column shows the number of classes with hotspots. The next two columns show the total number of hotspots and queries, while the last column lists the max. number of queries we found for a hotspot. Again, results of Alvor are included. We notice here, however, that a hotspot in *SQLInspect* is slightly different than that in Alvor. For us, a hotspot is the execution point of a query, while for Alvor, it is the point where a string is finally assembled. For a `Statement` in JDBC, these are the same, but for a `PreparedStatement` the execution (e.g. `executeQuery()` call) is preceded by one or more `Connection.prepareStatement()` calls. As a result, Alvor counts more hotspots. Another difference is that Alvor uses a regular language to describe the queries for each hotspot, while we generate all the possible query strings. Therefore a direct comparison of the number of queries is not possible.

Overall, we found that *SQLInspect* performs well. The SQL extraction competes with the extraction implemented in Alvor and the parsing time remains good considering the actual size of the applications. The reason for the relatively high memory usage is that our current implementation keeps in memory the AST nodes for all the query parts regardless of whether the query is opened in an editor or not. We plan to optimize this in the future. We should also mention that we analyzed the systems in Eclipse by merely importing each as a whole project and we did not break them down into smaller projects. In practice, Eclipse projects are typically smaller in size than the complete systems we examined.

5 RELATED WORK

In the introduction, we mentioned tools that inspired us, but there are many more that deserve a mention here.

In Java, Alvor is the most similar tool to *SQLInspect*. However, Alvor does not go further than a semantic analysis of the embedded query, while *SQLInspect* carries out a wide range of analyses. The basic query extraction algorithm is also entirely different as Alvor relies on an interprocedural, path-*in*sensitive constant propagation algorithm, whereas *SQLInspect* is path-sensitive. There is also a plug-in named *Eclipse SQL Explorer*, but it serves a different purpose as a thin SQL client to query and browse JDBC-compliant databases. A notable tool here is also *DBScribe* [6] that can extract SQL statements from Java code, but for documentation purposes.

Outside of the world of Java, a recent work of Anderson et al. [1] implements a similar query extraction algorithm and query model for PHP applications. There is also a tool named *Django SQL Inspector*[5] for projects using Django, which extracts SQL queries embedded through more layers and measures similar SQL metrics.

It should be added that *SQLInspect* is a novel tool implementation of our research work on SQL extraction [7], concept location [9] and smell detection [8]. As a tool implementation, we have significantly extended our work with new features such as the *query search*, the *table access analysis* and the *quality metrics*.

6 CONCLUSION AND FUTURE WORK

SQLInspect targets a challenging and practical software engineering problem of developers working with data-centric applications written in Java; namely overcoming the hassle of maintaining SQL code embedded in classes. For this purpose, we implemented recent analysis techniques that we briefly described here. Besides the implementation details, we provided some use cases in which *SQLInspect* proved useful. Afterwards, we performed a preliminary evaluation on open-source systems to compare *SQLInspect* with the only tool, Alvor, which is available for a similar purpose but it has less functionality. The results are more than promising: *SQLInspect* successfully competes on the query extraction task, and it also offers a wider range of functionality. We have several ideas for continuing the development. For example, we would like to smoothly integrate *SQLInspect* into the build process of Eclipse with an incremental analysis, and we also would like to support more SQL dialects.

REFERENCES

[1] David Anderson and Mark Hills. 2017. Supporting Analysis of SQL Queries in PHP AiR. In *Proc. of the 17th IEEE Int. Work. Conf. on Source Code Analysis and Manipulation*.

[2] Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, and Varmo Vene. 2010. An Interactive Tool for Analyzing Embedded SQL Queries. In *Proc. of the 8th Asian Conf. on Prog. Languages and Systems (APLAS2010)*. Springer-Verlag, 131–138.

[3] Huib van den Brink, Rob van der Leek, and Joost Visser. 2007. Quality Assessment for Embedded SQL. In *Proc. of the 7th Int. Working Conf. on Source Code Analysis and Manipulation (SCAM2007)*. IEEE, 163–170.

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Proc. of the 10th Int. Conf. on Static Analysis (SAS2003)*. Springer-Verlag, 1–18.

[5] Bill Karwin. 2010. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming (Pragmatic Programmers)*. Pragmatic Bookshelf.

[6] Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. 2016. Documenting Database Usages and Schema Constraints in Database-centric Applications. In *Proc. of the 25th Int. Symp. on Soft. Testing and Analysis*. ACM, 12.

[7] Loup Meurice, Csaba Nagy, and Anthony Cleve. 2016. Static Analysis of Dynamic Database Usage in Java Systems. In *Proc. of the 28th Int. Conf. on Advanced Information Systems Engineering (CAiSE2016)*. Springer LNCS.

[8] Csaba Nagy and Anthony Cleve. 2017. Static Code Smell Detection in SQL Queries Embedded in Java Code. In *Proc. of the 17th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM 2017)*. IEEE Comp. Soc.

[9] Csaba Nagy, Loup Meurice, and Anthony Cleve. 2015. Where Was This SQL Query Executed? A Static Concept Location Approach. In *Proc. of the 22nd Int. Conf. on Software Analysis, Evolution, and Reeng. (SANER2015)*. IEEE Comp. Soc., 580–584.

[5] https://github.com/rory/django-sql-inspector