Simplified Evaluation Framework for Query Extraction Techniques

Ivan Grgurina*, Dejan Škvorc*
*University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia
Corresponding author: ivan.grgurina@fer.hr

***Abstract*—The growing number of database management systems and applications that use them directly or through some form of proxy has led to the problems when trying to understand the queries sent from the application to the database. There are multiple ways to extract queries from the application, network or database management systems and the way to extract largely depends on the format of the application, technology used and the security concerns. Given these non-functional requirements, we generalize the possible functional features that can be used for evaluation: *query extraction* and subsequent *query modification*. The results are then presented using three dimensions: technique (approach), requirement (non-functional criterion), and feature (functional criterion) where applicable. Each combination is analyzed and given the possible use case. Developers can then use this to make an informed decision for their applications.**

***Keywords*—query extraction; query modification; embedded SQL; data builder; entity builder; SQL proxy**

I. Introduction

Data is everywhere. Users extract data and analyze it using queries. The queries that are used are usually hard-coded or dynamically generated from the user input. In both cases, the actual query is hidden inside the application. To extract the query, specialized tools need to be used that conform to the development or execution environment of the application. The technology, format, security and the possibility to modify application on the fly all have to be considered. For example, different tools have to be used when the application source code is available compared to when the application is only available in the executable format.

This paper presents the evaluation framework using the four main criteria for choosing the right query extraction technique for individual use case from the seven different techniques. Section II discusses scientific work and tools available for the seven techniques. Then the criteria is discussed and set in Section III forming a basis for the framework. Different techniques are introduced in Section IV, alongside the full analysis of their positive and the negative aspects. Each technique is individually considered with results available in tabular format and explained with more details in the text. Section V offers a consolidation and presents example decision making process when using the evaluation framework. Section VI concludes the paper.

II. Related work

The field of static and dynamic analysis of source code is very active, with no exception in query extraction and analysis. With reference to static methods, one of the most relevant is the work of Christensen et al. [1] who set the standard with their static analysis of Java programs that is still used today [2].

While some, like Ngo [3], set out to automatically identify all the possible database interaction points using static analysis, others like Linares [4] tried to extend that to automatically generate the documentation of the database usage in source code using the similar methods.

There is also a programming language called RASCAL [5] that is heavily used in research community for source code analysis and manipulation, particularly when it comes to the PHP programming language, one example of it being a PHP AiR tool [6]. Anderson [7] added a support for analysis of SQL queries to PHP AiR. There are other efforts [8] to develop an interactive tool for analyzing SQL queries.

Another problem occurs when doing static analysis of something that has dynamic elements in it, like dynamically generated queries when the application uses data builder or even entity builder. Gould [9] was at the forefront of that problem when he tackled the static checking of dynamically generated queries in database applications. Nagy [10] tackles the static analysis of both embedded and dynamic database usage in Java applications, while Meurice [11] focuses more on the dynamic usage.

With regard to to dynamic techniques, one possibility is to attach a proxy to the application as a customized SQL client. On the subject of JDBC, there is no better than the dynamic data management framework Apache Calcite [12], with its many moving parts in the background, including the Volcano optimizer [13] to optimize incoming queries, SQL parser and multiple adapters for different database engines. If something simpler is needed, JDBC does offer a way to log generated statements [14].

To make it easier for developers, there are multiple ORM frameworks. Some of them include the support for dynamic extracting, logging and modifying queries, albeit in very limited capability. Django has a Query Inspector [15] which provides a middleware for inspecting and reporting SQL queries executed for each web request, while Entity Framework [16] has a support for logging and intercepting database operations.

The most natural place for a proxy is on the network, where the tools used for manual interception are Wireshark [17] and Microsoft Message Analyzer [18], which is now deprecated with no replacement, but it did allow for some very useful things [19] when it came to query interception. There is also a number of network tools that enable query logging, one notable example being ProxySQL [20].

III. Evaluation criteria and methodology

In this section, we present the evaluation framework used to compare different query extraction and modification techniques.

A. Functional features

The main feature that has to be supported by all the techniques is a query extraction. Without the ability to extract queries in some shape or form, even the most basic, the technique can't be considered. Furthermore, some techniques have the ability to modify queries after extraction and then send the modified queries to the database. Such feature is very welcome, as it expands upon the possibilities of what can be done with the tool. With pure query extraction, a lot is already gained - information which databases, tables and columns are used most often, how are they used together, which tables are joined most often, what is the most used join predicate and so on. A comfortable decisions can be made by database administrators and architects to improve upon the database design based solely on that information. The subsequent ability to modify and execute modified queries after extraction opens up a lot of possibilities. Thus, the criteria for functional features in this paper is *query extraction* and *subsequent query modification.*

B. Non-functional requirements

In order to extract and modify queries, techniques have to respect some limits imposed on them by the application environment. Depending on whether the application is available as a source code which can be freely manipulated by the technique or if that luxury is not available and thus the technique has to rely on the executable application format, different techniques have developed over the years to tackle both problems. Henceforth, the first non-functional requirement is *application format*, which can be either executable or source code. One other thing to consider here is that some techniques can require modifying the original application, which in turn requires the application to be compiled again. Sometimes the changes can be implemented using external metadata files, in which case that's a huge advantage over changing a huge chunks of application code. Therefore, a second criterion is *application modification gradient*. It's a closest approximation of the amount of work that has to be done on the original application in order to get it to the state where the technique in question can be successfully used with said application. This can also differ for query extraction and query modification. Notice this only includes changes to the original application, not any work that has to be done in the separate tool. Another limitation that can arise is from the technology used, both on the side of the application and the database management systems. The *technology*, both application and database, is third criterion. Last one that is considered is *security*. Given the fact that some techniques require different ways of handling raw data, i.e. encrypted query that is being transferred via network using specific database protocol versus plain-text query embedded into application code, the need for security criterion becomes obvious. Depending on the user requirements, it may not be possible to extract queries without using tools that can handle encrypted queries over the network.

C. Evaluation framework

To set up an evaluation framework for query extraction techniques, three distinct dimensions are used for analysis. Results should take into consideration a technique, a requirement and a feature. Techniques are categorized based on their underlying technical intricacies and then compared based on their functional and non-functional criteria. Both requirements (non-functional criteria) and features (functional criteria) are used to analyze each given technique and assess its weaknesses and advantages for each possible scenario.

IV. Results

There are different ways to divide query extraction techniques into categories. The most common one used here is dividing them into static and dynamic techniques. Static techniques are also known as parsing and can be subdivided into embedded and builder subcategories. When it comes to embedded parsing, query extraction happens on full plain-text SQL queries embedded into the application code. Not all queries are plain-text and most are actually created using builders, either data builders like JDBC and ADO.NET or entity builders like Entity Framework and Hibernate.

Dynamic query extraction relies on interception using proxies and can be done in different places in query life-cycle. It's possible to distinguish application proxies, either in the form of the library or full pledged ORM proxy, network proxies, and database proxies.

Using this categorization of techniques, it's possible to discern a certain positive and negative aspects of each. The complete comparison is shown in Table I.

Results are divided into subsections, where each subsection corresponds to the technique or approach that can be used in the query extraction. Each technique is then analyzed using the other two dimensions, requirements and features.

A. Static embedded parser

Static technique that parses plain-text SQL queries requires knowledge about the SQL dialect and its version, as well as the access to the plain-text application code files. Some programming languages have database libraries that enable developers to write embedded queries, most often assigned directly to the string variables which are then executed. Such queries are usually not changing very often so they pertain to the more static or long running jobs that applications do regularly. They are not suitable

Table I: Comparison of the requirements between techniques.

REQUIREMENT	STATIC TECHNIQUES			DYNAMIC TECHNIQUES			
	embedded parser	data builder parser	entity builder parser	application library proxy	application ORM proxy	network proxy	database proxy
Application format							
Query extraction	Source code.	Source code.	Source code.	Both.	Both.	Executable format.	Executable format.
Query modification	Source code.	Source code.	Source code.	Both.	Both.	Executable format.	Executable format.
Modifications							
Query extraction	No change.	No change.	No change.	Minor configurable changes.	Minor changes.	Configuration.	Configuration.
Query modification	Rewriting embedded SQL statements.	Rewriting data builder code and SQL.	Rewriting entity builder code.	Minor changes.	Major changes. Very limited.	No changes. Done inside proxy.	No changes. Database engine extension.
Technology - Application							
Query extraction	Parsing different programming languages.	Parsing code in different programming languages.	Parsing code in different programming languages.	Has to use the same programming language as application.	Depends on the ORM, which means it heavily depends on the application code.	Doesn't depend on the application code.	Doesn't depend on the application code.
Query modification	Modifying different programming languages	Modifying code in different programming languages.	Modifying code in different programming languages.	Has to use the same programming language as application.	Depends on the ORM, which means it heavily depends on the application code.	Doesn't depend on the application code.	Doesn't depend on the application code.
Technology - Database							
Query extraction	Parsing different SQL dialects.	Data query builder can be database-agnostic.	Entity builder is database-agnostic.	Needs parser for each database engine.	Doesn't depend on the database vendor because ORM handles that.	Depends on the database vendor for database engine network protocol (extraction) and parsing SQL statements.	Heavily depends on the database vendor.
Query modification	Modifying different SQL dialects.	Data query builder can be database-agnostic.	Entity builder is database-agnostic.	Needs adapter for each database engine.	Doesn't depend on the database vendor because ORM handles that.	Depends on the database vendor for database engine network protocol and changing SQL statements.	Heavily depends on the database vendor.
Security concerns	Read/write access to source code.	Read/write access to source code.	Read/write access to source code.	None.	None.	Network protocol traffic is encrypted in production. Access control for apps that connect to this external proxy.	Direct access to the database requires access control for apps that connect to this external proxy.

for arbitrary user input, so anything more complicated cannot be done here with ease.

In order to make this technique work, it needs to know how to parse both an application code written in some programming language and an embedded SQL code on top of that. Figure 1a shows the first step is to parse the application code, either completely or just to identify the *hotspots* where the embedded SQL code is located. The output from the first step is a list of SQL statements that is then processed in the second step to analyze SQL statements and modify them using the abstract syntax tree or the relational algebra.

B. Static data builder parser

Taking into account the complexity of maintaining plain-text embedded SQL statements in application code, developers are using data builders like JDBC to simplify their code and enable them easier variable user input for their SQL statements. JDBC and its relatives provide an API that can be used as part of the application logic to select or modify data inside the database. The SQL statement is still built mostly manually, similar to embedded queries, but there is now more control over the transaction.

For this technique shown in Fig. 1b to work, it needs to know how to parse application code and to identify spots where the JDBC API is used. It also needs to use data flow and control flow techniques to find out where the variable user input is located and take that into account when modifying the SQL statements. SQL statements no longer have to be pure strings, but are now complex objects in memory, and thus dynamically generated at runtime. This means that extraction and modification is more complex as the static data builder parser doesn't have full information, e.g. where a query is conditionally different depending on the dynamic parameter that is unknown at the compile-time. This technique can't answer the question which SQL statements will be executed and which will be omitted in particular execution scenario, but it can still modify both statements to work in the updated setting.

C. Static entity builder parser

When the performance is not so critical, developers prefer to use entity builders like Entity Framework or Hibernate. For a relatively small performance overhead, developers gain the ability to model their relational data structure using simple classes called entities that use object-relational mapping to map attributes in entities to columns in database tables. There is an entire set of API methods that has to be used when communicating with the database, so in order to make this technique shown in Fig. 1c to work, parser has to know how to parse application code and identify hotspots where the framework calls the database (like the Entity Framework call to the DbContext), find out what is the generated statement based on the actions done over the entity set and find out what is the correlation between the generated statement and the user input. With that, the extraction part is done. Doing modifications to this format requires only application code modification, since there is no plain-text SQL. Since all the SQL functionality is done using native application code, this technique is limited by the ability to modify such code. For example, in language integrated queries - LINQ, commonly used with Entity Framework, all the queries are created using a set of extension methods like filters and projections. Thus, to modify the query, one has to modify the set of extension methods that are applied on the set of entities in the database context.

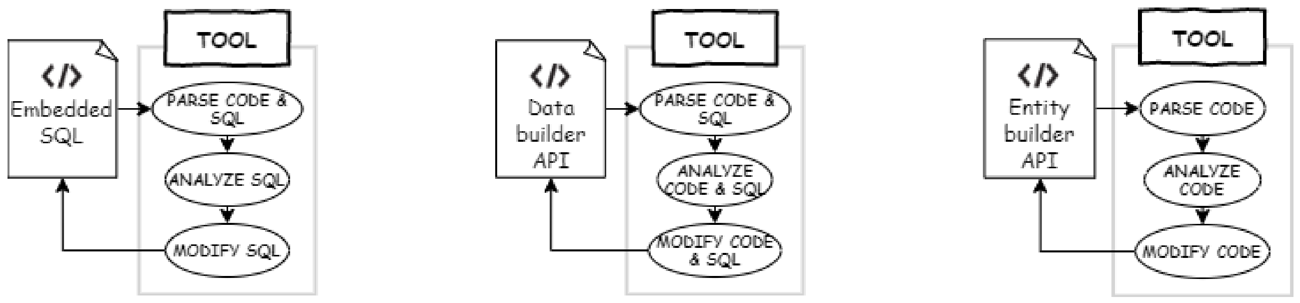
D. Dynamic application library proxy

Compared to the techniques mentioned so far that work in the static context, it's also possible to write new code in the form of a separate middleware that will intercept the query and then send the modified query to the database, while also handling the response and notifying the caller application. If that proxy is tightly integrated with the application, so much that it requires changing the application code and possibly even the architecture, then it's called dynamic application library proxy (Fig. 2a). It most often utilizes data builders like JDBC and acts as JDBC driver that the original application uses to communicate to the database, while also offering the ability to connect to that database itself.

One such framework, Apache Calcite [12], acts as a JDBC driver when using relational database engines, so the application setup can be reduced to changing how it connects to the database by setting it to connect to the Apache Calcite proxy instead. The interception of the SQL query is then relatively simple, as the application sends the query through the driver. That query is then available inside the proxy, where it is parsed into abstract syntax tree, validated and transformed into relational algebra tree, where the optimization with specific rules can occur. Finally, the proxy uses it's own drivers to connect to appropriate database and execute transformed and optimized query. This is very similar to how database engines work internally with execution plans, but since it's extracted into a proxy, it can be independent from any specific database vendor by using adapters.

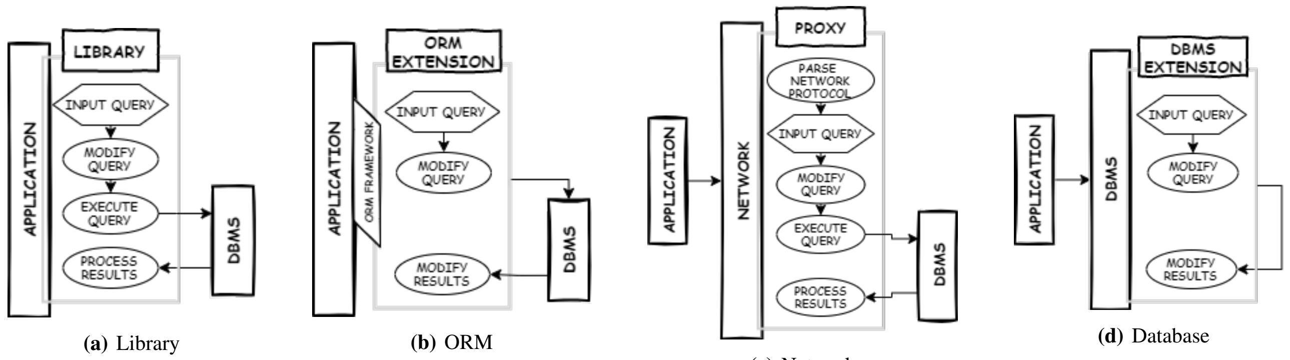
E. Dynamic application ORM proxy

The technique shown in Fig. 2b is similar to the static entity builder parser, but instead of modifying the application code by rewriting what projection or filter methods are applied, the changes are done at runtime. Common ORMs like Entity Framework and Hibernate generate appropriate SQL statements at runtime that are then executed on given database engine. It is possible, but not easy, to intercept and rewrite queries at the time when they're processed inside the ORM. It requires writing a proxy code that uses internal API of the ORM in order to manipulate final SQL statement generation process. For example, anytime Entity Framework sends a command to the database, this command can be intercepted by application code. This is most commonly used for logging SQL, but can also be used to modify or abort the command.



(a) Embedded **(b)** Data builder **(c)** Entity builder

Figure 1: Static techniques



(a) Library **(b)** ORM **(c)** Network **(d)** Database

Figure 2: Dynamic techniques

F. Dynamic network proxy

When the application sends the query to the database, it uses a specific network protocol for that database vendor, so to be able to extract a query, network traffic has to be intercepted and then the underlying network protocol can be decoded. Tools like Wireshark and Microsoft Message Analyzer can be used. The steps are shown in Fig. 2c. Another problem that appears is the fact that the network traffic is and should be encrypted, so it requires an appropriate key exchange in place to decrypt the traffic. The application can then call this external proxy using any means of inter-process network communication as it's implemented as a separate executable reachable over a network. While this is most commonly used for load balancing, it can also be used to intercept the query, modify it and send the modified query to the database.

G. Dynamic database proxy

The last possibility is putting the proxy on the database server where the database engine is located, as displayed in Fig. 2d. Writing a library that extends the database engine and intercepts the SQL statements when they arrive at the database, and then modifies them just before they are executed would make for a proxy that is independent from any application format and can be strictly enforced at the database level. It does require an intricate knowledge of the inner workings of the database engine, and the database engine has to actually allow for extensions. Depending on the vendor, that may not be possible. Commercial products like Oracle and Microsoft SQL Server are not very open for custom extensions to their proprietary engines.

V. Discussion

With such a wide variety of techniques available for query extraction, it seems hard to find the most suitable one. The main argument here is it all depends on the use case. Four main requirements were set to help make the selection. Table I shows side-by-side comparison of all the techniques. If the application source code is available and the application has only embedded queries, the choice is using static embedded SQL parsing. Most often the application source code is not available and all that's available is the application executable, in which case only external dynamic proxies can work. When both is available, next step is to look over how the application uses database, whether is it through embedded queries, data builder or entity builder. That limits the choice to one of the static methods and some of the dynamic methods. If it doesn't use entity builder, it's not possible to use static nor dynamic entity builder technique. If it's not using data builder, that rules out application library proxy in most cases as they are most often attached to data builder framework. So far this only takes into consideration using the techniques without modifying the original application. Each technique requires some form of change to make it work. If there is the option available to modify the original application, the second argument *modifications* describes how much change is needed to make the technique work. Probably the biggest limiter is the technology, as it's the first thing that can be incompatible. Some techniques like static parsing embedded queries are limited by both the programming language used for application source and the SQL dialect used for embedded queries, while some like database proxy don't depend on the application, only the database engine vendor. Finally, security concerns can be hard to handle if they require adding a whole new set of access control layers.

VI. Conclusion

Extracting queries from an application is a complex topic that covers different software engineering principles that make multiple discussed techniques a viable option. From extracting embedded statements from source code, where anything is possible but very complex, to dynamic entity builders proxies that are simple to setup but don't offer a lot of possible modifications out of the box, there is something to cover every imaginable scenario. The paper gives an overview of the seven available techniques and analyzes their positive and negative aspects. In addition, we present a tool in the form of a simplified evaluation framework for query extraction techniques with four main criteria to choose the right one for the individual use case.

For future work, emerging query extraction techniques can be classified with respect to the simplified evaluation framework. The generic evaluation framework itself can also be extended with additional criteria to fit a specific domain where the requirements might differ slightly.

Acknowledgment

This research is co-sponsored by the European Union Regional Development Fund through a research grant KK.01.2.1.01.0109 "Cloud Computing Security During the Use of Mobile Applications". We acknowledge the support of the Ministry of Economy of the Republic of Croatia as well as our research partners OROUNDO Mobile GmbH Austria and OROUNDO Mobile GmbH Subsidiary Croatia.

References

[1] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. "Precise Analysis of String Expressions". In: *Proceedings of the 10th International Conference on Static Analysis*. SAS'03. San Diego, CA, USA: Springer-Verlag, 2003, pp. 1–18. ISBN: 3-540-40325-6. URL: http://dl.acm.org/citation.cfm?id=1760267.1760269.

[2] *Query Parser*. Uber. URL: https://eng.uber.com/queryparser/.

[3] Minh Ngoc Ngo and Hee Beng Kuan Tan. "Applying static analysis for automated extraction of database interactions in web applications". In: *Information and software technology* 50.3 (2008), pp. 160–175.

[4] Mario Linares-Vásquez et al. "Documenting database usages and schema constraints in database-centric applications". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM. 2016, pp. 270–281.

[5] Paul Klint, Tijs van der Storm, and Jurgen Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: Sept. 2009, pp. 168–177. DOI: 10.1109/SCAM.2009.28.

[6] Mark Hills and Paul Klint. "PHP AiR: Analyzing PHP systems with Rascal". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Feb. 2014, pp. 454–457. DOI: 10.1109/CSMR-WCRE.2014.6747217.

[7] D. Anderson and M. Hills. "Supporting Analysis of SQL Queries in PHP AiR". In: *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Sept. 2017, pp. 153–158. DOI: 10.1109/SCAM.2017.23.

[8] Aivar Annamaa et al. "An interactive tool for analyzing embedded SQL queries". In: *Asian Symposium on Programming Languages and Systems*. Springer. 2010, pp. 131–138.

[9] C Gould, ZD Su, and Premkumar Devanbu. "Static checking of dynamically generated queries in database applications". In: *Proceedings - International Conference on Software Engineering*. Vol. 26. June 2004, pp. 645–654. ISBN: 0-7695-2163-0. DOI: 10.1109/ICSE.2004.1317486.

[10] C. Nagy and A. Cleve. "SQLInspect: A Static Analyzer to Inspect Database Usage in Java Applications". In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. May 2018, pp. 93–96.

[11] Loup Meurice, Csaba Nagy, and Anthony Cleve. "Static analysis of dynamic database usage in java systems". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2016, pp. 491–506.

[12] Edmon Begoli et al. "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 221–230. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3190662. URL: http://doi.acm.org/10.1145/3183713.3190662.

[13] Goetz Graefe and William J. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search". In: *Proceedings of the Ninth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1993, pp. 209–218. ISBN: 0-8186-3570-3. URL: http://dl.acm.org/citation.cfm?id=645478.757691.

[14] *The best way to log JDBC statements*. Vlad Mihalcea. URL: https://vladmihalcea.com/the-best-way-to-log-jdbc-statements/.

[15] *Django Query Inspector*. Good Code Zagreb. URL: https://github.com/dobarkod/django-queryinspect.

[16] *Logging and intercepting database operations - EF6 — Microsoft Docs*. Microsoft. URL: https://docs.microsoft.com/en-us/ef/ef6/fundamentals/logging-and-interception.

[17] *Wireshark*. Wireshark Foundation. URL: https://www.wireshark.org/.

[18] *Microsoft Message Analyzer Operating Guide - Message Analyzer — Microsoft Docs*. Microsoft. URL: https://docs.microsoft.com/en-us/message-analyzer/microsoft-message-analyzer-operating-guide.

[19] Alexander Taubenkorb. *How can I decode SQL Server traffic with Wireshark? - Stack Overflow*. URL: https://stackoverflow.com/a/39764998.

[20] *ProxySQL*. ProxySQL LLC. URL: https://proxysql.com.