Extraction of Embedded Queries via Static Analysis of Host Code

Petros Manousis[1], Apostolos Zarras[1], Panos Vassiliadis[1], George Papapstefanatos[2]

[1] Dept. of Computer Sc. & Eng., Univ. Ioannina (Hellas)
{pmanousi, zarras, pvassil}@cs.uoi.gr
[2] ATHENA Research & Innovation Center, IMIS
gpapas@imis.athena-innovation.gr

Abstract. Correctly identifying the embedded queries within the source code of an information system is a significant aid to developers and administrators, as it can facilitate the visualization of a map of the information system, the identification of areas affected by schema evolution, code migration, and the planning of the joint maintenance of code and data. In this paper, we provide a solution to the problem of identifying the location and semantics of embedded queries with a generic, language-independent method that identifies the embedded queries of a data-intensive ecosystem, regardless of the programming style and the host language, and represents them in a universal, also language-independent manner that facilitates the aforementioned maintenance, evolution and migration tasks with minimal user effort and significant effectiveness.
Keywords: Reverse engineering of database queries, Query extraction, Embedded queries.

1 Introduction

To operate properly, data-intensive applications rely on *embedded queries*, that are programmatically constructed (typically, in progressive, incremental fashion) to facilitate the retrieval of data from the underlying databases. *Identifying the location and semantics of these queries and making them available to developers is very important.* In a most common scenario, database schema migration, refactoring and evolution require the appropriate visualization and inspection of data-related code, spread across multiple modules and files, for evaluating the impact of the schema change to the overall software ecosystem. As another example, when an administrator wants to modify a part of the database, it is imperative that the developers of the surrounding applications are informed on the change and have the means to identify the parts of the code that are going to be affected by that change [1],[2].

Yet, obtaining these queries is an extremely painful process. An embedded query is, typically, progressively constructed via a sequence of source code statements that modify the query internals according to user choices. In the past, the most popular way to perform this task was via *string-based* embedded queries

```
1 $result = db_query('SELECT source, alias FROM {url_alias} WHERE source in
      (:system) AND language = :language_none ORDER BY pid asc;', $args);
```

```
1  function _profile_get_fields($category,$register=FALSE) {
2   $query = db_select('profile_field');
3   if ($register) {
4    $query->condition('register',1);
5   }
6   else {
7    $query->condition('category',db_like($category),'LIKE');
8   }
9   while (!user_access('administer users')) {
10   $query->condition('visibility',PROFILE_HIDDEN,'<>');
11  }
12  return $query->fields('profile_field')->orderBy('category','ASC')
13    ->orderBy('weight','ASC')->execute();
14 }
```

Fig. 1: Embedded queries of Drupal-7.39; string (top) and object based(bottom)

(Fig. 1 top). String-based queries were authored in SQL and parts of the query clauses were added or modified according to the context via if statements.

However, programming practice has departed from the traditional string-based construction of embedded queries and, *developers now employ certain reusable host language facilities (e.g., a specific API provided by the host language), to programmatically construct and execute the respective queries*. We call this way of query construction *object-based* as queries are formed as objects of the host language that are further manipulated by functions of an API that is responsible for the integration with the database. See Fig. 1 for the construction of such a query; the query is represented by an object, under the variable $query and further modified by the host PHP code via calls to the methods of a database-related API.

The state of the art methods and tools on query extraction do not support a general, easily understood and language-independent method for the identification of embedded queries, especially when it comes to object-based ones (see Section 6). The current methods and tools work only in specific environments (e.g., Java, or C#) via translating the object-based queries to string-based ones, or examine only the queries that are most likely to be generated by the execution flow of the source code [3,1].

To address these shortcomings, in this paper, we propose a principled, customizable language-independent *method that is able to (a) identify the embedded queries of a data-intensive ecosystem, regardless of the programming style and the host language, as well as by finding all their variations due to branching statements, and at the same time, (b) represent them in a universal,* language-independent *manner that can later facilitate migration or reconstruction, with (c) minimal user effort and significant effectiveness.*

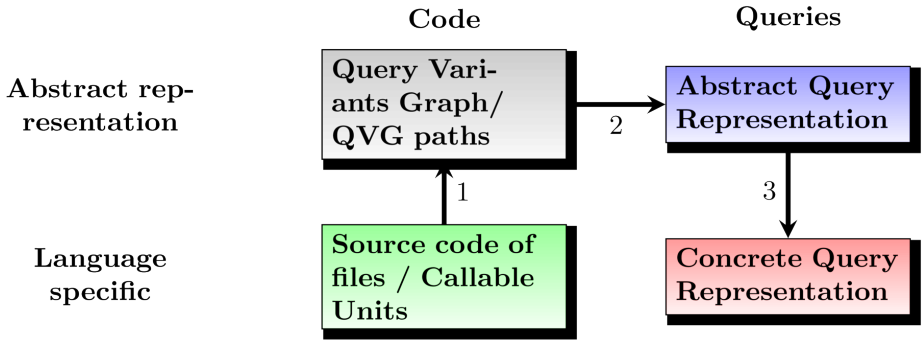


Fig. 2: The steps of our method

Our method consists of four parts, depicted in Fig. 2. As discussed in Section 2, we start with source code files as input. Initially, we decompose the input files to their structural parts (functions/methods) and we keep only these parts of the code that host queries. For simplicity reasons, in this paper we focus on SPJ string-based queries and data-retrieval object-based operations. Still, our approach is also applicable to a wider class of queries as well as DML operations. In the context of our language-independent approach, we uniformly will hereafter refer to functions/methods/procedures/routines as *Callable Units*. In general, a Callable Unit is: "a sequence of program instructions that perform a specific task, packaged as a unit"[3]. For those Callable Units we create an abstract representation of their code that we call *Query Variants Graph (QVG)*. A QVG is a tree-like graph representation of a Callable Unit that uses the database. Due to the existence of branch and loop statements in the code, our next task is to traverse the Query Variants Graph and find every possible variation of a query that could occur at runtime. The result is a set of *QVG paths*, i.e., path traversals from the root of the QVG till one of its leafs. Observe that our representation abstracts the syntax details of the host language, thus it is language-independent, depending only on premises like Callable Units, and branch and loop statements that are practically universal. Our next step is the extraction of queries from the QVG paths and their representation into a generic, language-independent model. To represent queries in our model, we introduce an extensible pallet of *Abstract Data Manipulation Operators* with fundamental data transformation and filtering operators. This facilitates a universal representation of queries, independently of the source language (thus the need for extensibility). So, in Section 3 we present how queries are represented as combinations of these operators, via a model of representation which we call *Abstract Query Representation* (AQR). An AQR is a directed acyclic graph with nodes that describe the database-related parts of the code and its purpose is to formally represent the queries. Finally, we can exploit the Abstract Query Representation for various purposes, by converting the abstract representation to a specific, target language, a facil-

[3] `https://en.wikipedia.org/wiki/Subroutine`

ity useful both for the understandability of the queries and for different kinds of migrations – e.g., either between database engines (from MySQL to Oracle) or to completely different environments, like MongoDB. This part is (shortly) discussed in Section 4.

Our discussion is supported by our experimental assessment, presented in Section 5. We have tested our method with systems built in different source languages (PHP and C++) and achieve very high numbers of recall and correctness (larger than 80%) with quite low user effort.

2 Source Code to Query Variants Graph

In this section, we address the problem of identifying all the variants of the queries that exist in the source code of a given information system. To do so, we initially abstract the input of this step, which is the source code of the information system, to a Query Variants Graph that removes the language-specific control statements such as branch and loop statements of the host language. Next, we generate every possible query variant via traversing the QVG paths. Thus, the result of this step is a set of QVG paths for every query-related Callable Unit of the information system.

2.1 QVG Construction

Starting from a set of files that constitute the source code of an information system, our first step is to identify the query-related files and skip everything else. Then, we decompose these files to their Callable Units and we perform a second layer filtering keeping only the query-related Callable Units, such as those of Fig. 1 which query two relational tables.

Extraction of Callable Units The first intermediate step towards abstracting the source code in language-independent format is the extraction of Callable Units. We initially check whether a file contains any database-related code statement either checking for query-related statements through string-based pattern matching or for query-related object initializations. If there is no such statement, we skip the file. Otherwise, we split it to its Callable Units. Similarly, we omit Callable Units without embedded queries in them. Thus, we end up working only with query-embedding Callable Units, significantly reducing the amount of work and resources needed to be invested in the subsequent steps.

The price to pay To extract the appropriate information from the source files, we need to perform simple extractions from the source code. This requires (a) *physical level information* like the location of the source code and the parts of it that are to be ignored (e.g., binary files), (b) *query-related information* denoting the terms signifying a query, and, (c) *language-specific information*.

Concerning the query-related information, as already mentioned, we discern between two categories of hosting. In the first case, where queries are handled as strings, we need to know the API functions that use that string, so as to perform slicing in order to find the query strings (in our example of Fig. 1 the

function contains the complete query string). In the second case, where queries are handled as objects and their definition is manipulated via a dedicated API for query construction, we need to know the API functions that construct an object-based query.

The way we do this is by splitting the original project to Callable Units on the basis of a formally specified grammar that requires the user to enter *once per language*: (i) how the comments start and end (both single-line and multiple-line comments), (ii) how the string values are described in the host language (eg. in C++ this is done by using the character: '"'), (iii) if there are characters that "escape" the string value markers (e.g., in C++ the character: '\'), (iv) finally how to treat the branch and loop statements of the host language. In this grammar, we treat nearly all loop statements similarly to branch statements. Remember that we are doing *static* analysis to dig out the query semantics. As loops are typically populating filters with values produced at runtime, we only need to handle the contents of the loop once, to identify the used expression along with the usage of an artificial set-valued pseudo-constant without practically misrepresenting the query's semantics.

Query Variants Graphs Having explained the input and the method for the extraction of Callable Units, we now move on to describe the abstract representation of the code.

A Query Variants Graph is a graph with nodes the blocks of the source code, without branch and loop statements. The edges correspond to the control flow of the code (aka they "consume" the branch and loop statements). A formal definition of the Query Variants Graph is described in Definition 1.

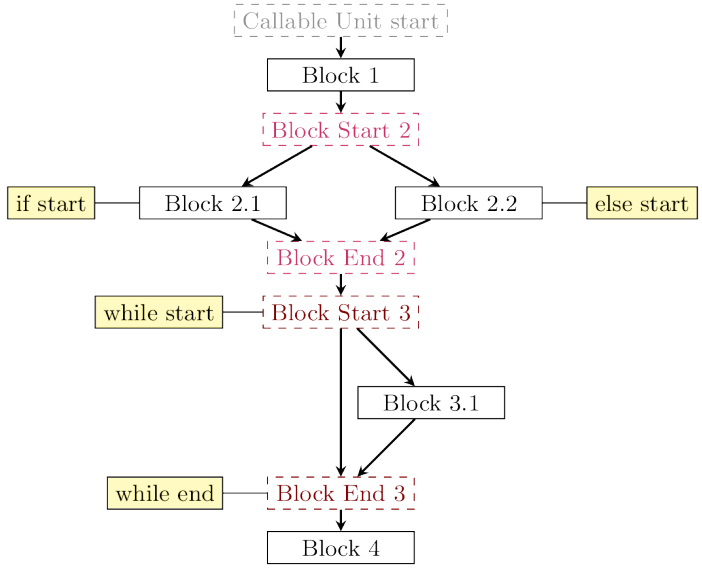
Definition 1. ***Query Variants Graph*** **-** *a directed rooted graph $QVG(V, E, r)$, where V is the set of nodes of the graph corresponding to elements of a Callable Unit, E, the set of directed edges connecting elements of the Callable Unit together, and r belongs to V is the root node, with the following properties:*

1. *The root of the graph corresponds to the entire Callable Unit CU.*
2. *Sibling nodes have the following properties:*
   - *they share the same code both among them and also with their parent, both before and after the branching/looping statement of their parent*
   - *each sibling replaces the branching/looping block (including the branch/loop statement) of their parent with exactly one alternative execution block*
   - *for every alternative branching/looping block there is exactly one sibling node.*

Algorithm 1 serves the creation of the Query Variants Graph tree. A Callable Unit is decomposed to its blocks, starting with the first branch or loop block. The code of that block is split to its components and each one of them becomes a "sibling" node of the QVG. After that, the remaining code is checked again for branch/loop blocks, and, of course, the "siblings" are checked for branch/loop blocks too.

```
1  function _profile_get_fields($category,$register=FALSE) {
2   $query = db_select('profile_field');
3   if ($register) {
4    $query->condition('register',1);
5   }
6   else {
7    $query->condition('category',db_like($category),'LIKE');
8   }
9   while (!user_access('administer users')) {
10   $query->condition('visibility',PROFILE_HIDDEN,'<>');
11  }
12  return $query->fields('profile_field')->orderBy('category','ASC')
13    ->orderBy('weight','ASC')->execute();
14 }
```


(a) The example of Fig. 1 bottom, annotated with (a)sequential blocks (with horizontal labels) and (b) Loop and branch blocks (with vertical labels).



(b) Query Variants Graph of modified reference example.

Fig. 3: The example of Fig. 1 in two representations: (a) text and (b) graph.

2.2 QVG Path Identification

In this subsection, we address the problem of identifying the different variants of a query that may occur during the execution of the code. This is done via a DFS-like algorithmic approach, where we traverse every Query Variants Graph path, regardless of whether the path contains query-related code or not. Algorithm 2 formally describes how we identify the variants of a query.

We perform a top-down traversal of the graph and we keep all code statements encountered from the root to each visited node, in a variable, called QP in our algorithm.

```
   Input: A Callable Unit (CU)
   Output: The root node for the Query Variants Graph of CU Callable Unit's
           source code (along with the rest of the tree that is constructed).
 1 Block = new node;
 2 Block =CreateGraph(CU, Block);
   Procedure CreateGraph(CU, Parent)
 1 |   Block = new node;
 2 |   branches = code of the first branch/loop block;
 3 |   if branches ≠ ∅ then
 4 |   |   if branches ≠ contain final alternative then
 5 |   |   |   branches += empty branch statement;
   |   |   end
 6 |   |   BlockStart = new node;
 7 |   |   preceding = code before the start of first branch block;
 8 |   |   if preceding ≠ ∅ then
 9 |   |   |   Block = preceding;
10 |   |   |   link BlockStart to Block;
11 |   |   |   link Block to Parent;
   |   |   end
12 |   |   else
13 |   |   |   link BlockStart to Parent;
   |   |   end
14 |   |   BlockEnd = new node;
15 |   |   foreach sibling ∈ branches do
16 |   |   |   link BlockEnd to CreateGraph(sibling, BlockStart);
17 |   |   |   remove examined code;
   |   |   end
18 |   |   return CreateGraph(M, BlockEnd);        ▷ Code after 1st branch
   |   end
19 |   else
20 |   |   Block = all CU code;                  ▷ Block without branch/loop
21 |   |   link Block to Parent;
22 |   |   return Block;
   |   end
```

Algorithm 1: Creation of Query Variants Graph

Initially, we start from the root node of the QVG($CU.Block$), with an empty list of query variants (named $queryVariants$) and an empty string statement (named $codeUpToNow$). For each node that we visit, we append in QP the code statements of the visited node. Then, we check if the visited node has any children nodes. If the node has no children nodes and QP is not empty, then we have finished with a traversal and we add the contents of QP to the $queryVariants$ list. The contents of QP are the code statements from the $CU.Block$ node up to a "leaf" node of QVG. If the node we visited has children nodes, then for each one of them we recursively call the *TraversePaths* procedure, giving as starting

```
   Input: A Callable Unit (CU)
   Output: The database-related QVG paths of a Callable Unit (queryVariants).
   Variables: queryVariants = ∅, codeUpToNow = ∅;
1  TraversePaths(CU.Block, codeUpToNow, queryVariants);
   Procedure TraversePaths(v, codeUpToNow, queryVariants)
1  |  QP = codeUpToNow + statements of v;
2  |  if v has no children then
3  |  |  if QP ≠ ∅ then
4  |  |  |  queryVariants+ = QP;
   |  |  end
   |  end
5  |  else
6  |  |  forall the w : children of v do
7  |  |  |  TraversePaths(w, QP, queryVariants);
   |  |  end
   |  end
```

Algorithm 2: Creation of QVG paths for a Callable Unit CU

node the child node that we want to visit, as "up to now" string statements the QP variable and as list, the $queryVariants$ list of paths.

The difference of *TraversePaths* procedure to the well known DFS algorithm is that we do not mark the nodes we visit. This is because we may encounter a node in more than one traversals, coming from different ancestor nodes. Thus, the information that is kept in a node (the contents of QP that are the code statements that we encountered till the node we have reached) differs on each traversal, and marking it as visited would produce wrong results. Observe the Query Variants Graph of Fig. 3b: the bottom *Block 4* node is used in four different traversals, marking it as visited after the first traversal would result in ignoring its statements in the remaining three traversals.

Coming back to our reference example, we can see that the Query Variants Graph of Fig. 3b provides four different traversals. The *Block 1* and *4* nodes are used in all traversal. The first traversal uses the *Block 2.1* node and does not use the *3.1* node. The second traversal differs to the previous one only in one place: instead of *2.1* node, this traversal uses the *2.2* node. The other two traversals of Query Variants Graph of the $\_profile\_get\_fields$ Callable Unit use the *3.1* node that was previously excluded from the traversals.

3 From QVG Paths to Abstract Query Representations

In this section, we introduce a universal way to represent the query variants that we obtained from the QVG traversals. Moreover, since this is an abstract query representation, it should be able to describe any database query, despite of how it was created (object-based or string-based queries).

To represent queries, we use an extensible pallet of Abstract Data Manipulation Operators ($ADMO$) that represent the different parts of a query. Our

Table 1: Abstract Data Manipulation Operator with a description of the part of a query that they represent

Source	Describes a provider of information in a query (e.g., a table in SQL).
Projector	Describes an output attribute (e.g., the SELECT attributes in SQL).
Comparator	Describes a filter that the output of the query should fulfil (e.g., the conditions of the WHERE clause in SQL).
Grouper	Used for summarizing of the output (used for grouping the incoming data in groups, each group identified by a unique combination of grouper values, e.g., the attributes of the GROUP BY clause in SQL).
Ordering	Used for sorting of the output (e.g., the attributes of the ORDER BY clause in SQL).
Limiter	Used for restricting the size of the output (e.g., the TOP/LIMIT clauses of an SQL query)
Aggregator	Used for applying an aggregate function to a input attributes (e.g., the MIN, MAX, COUNT, SUM, AVG functions in SQL)

operators cover the relational algebra, therefore we are able to represent queries embedded in relational database management systems. The operators are given in Table 1. The Abstract Data Manipulation Operator pallet is extensible; new operators can be added to cover cases of non relational databases.

Input: A QVG path of a Callable Unit (P), a mapping (M) of the API functions to ADMOs
Output: The Abstract Query Representation of P.

```
1 Add Start node for AQR ;
2 foreach QVGNode N ∈ P.nodes do
3     functionsOfNode = split contents of N to its functions;
4     foreach F ∈ functionsOfNode do
5         FAMDOs = M(F);          ▷ Find the ADMO nodes for function F
6         foreach fadmo ∈ FAMDOs do
7             Set function's F parameters to fadmo's ADMO parameters;
8             Add fadmo to AQR ;
          end
      end
  end
9 Add End node for Abstract Query Representation ;
```

Algorithm 3: Transforming a QVG path to its AQR representations

Definition 2. ***Abstract Query Representation (AQR)*** **-** *An abstract query representation $AGR = (V, E)$ is a directed acyclic graph whose nodes, V, are Abstract Data Manipulation Operators that describe a part of the query. An edge $e \in E$ from a node v_i to a node v_j specifies that the execution of the statement represented by v_i precedes the execution of the statement represented by v_j. The*

set of nodes $V = Start \cup Nodes \cup End$, is a union that comprises the following nodes:

- *A node Start that specifies the beginning of a query variant q.*
- *A set of nodes Nodes that represent Abstract Data Manipulation Operators which serve for generating the different parts of the query variant q. Each one of the nodes is an Abstract Data Manipulation Operator (ADMO) as described in Table 1.*
- *A node End that serves for concluding the generation of q.*

Algorithm 3 formally describes the AQR construction from QVG paths. For the string-based constructed queries, the mapping of the SQL parts to the AQR nodes is a straightforward procedure. Using as reference the example of Fig. 1 we tokenize the first parameter of *db_query* function (which is our input) to the parts that are between the capitalized words. Then, we add Projector operators for each of the values that follow the SELECT keyword as a parameter to each node. We add a Source operator for the value that follows the FROM keyword, with its parameter (*url_alias* in our example). We add comparator operators (with their parameters) for the values that follow the WHERE & AND keywrods. Finally, we add an orderding operator (with its parameters) for the value that follows the ORDER BY keyword. Table 1 describes all possible keyword - ADMO combinations for the SQL queries.

Observe that since a string-based query might be modified in the source code, we may need to perform slicing (forward slicing, as mentioned in [4]) to find out whether our query was modified or not (in our example it is not happening). In our approach, we perform slicing only on the code of the Callable Unit that we examine. Inter slicing techniques that use dependency graphs to identify the parts of the queries that are constructed in other Callable Units (e.g., see [5]) have not proved to be necessary in our experiments; of course, they are a clear extension for future work.

In the case of object-based constructed queries, we need some additional input in order to construct the AQR out of the variants we obtained from Algorithm 2. We initially retrieve the contents of the variants and we decompose the statements of those variants to the API functions of the project we examine, as we need to map the functions of the project's API to the Abstract Data Manipulation Operators of Table 1. This is work performed exactly once, and it is project-related (since each project has it's own API). In Section 5 we discuss the developer's effort for this task. In Fig. 4 we see the creation of an AQR that comes from the first traversal of the $\_profile\_get\_fields$ Callable Unit. The project's API functions are translated to Abstract Data Manipulation Operators.

The AQR representation allows us to compare queries on the similarity of their structure. That is useful because we might obtain query variants (in one of the Callable Units that we examine) with identical structure (albeit, possibly with different values). This is due to branch/loop blocks in the source code of a Callable Unit that are unrelated to the query-object, and since we consider all query variants as valid for our research, we need to identify the same

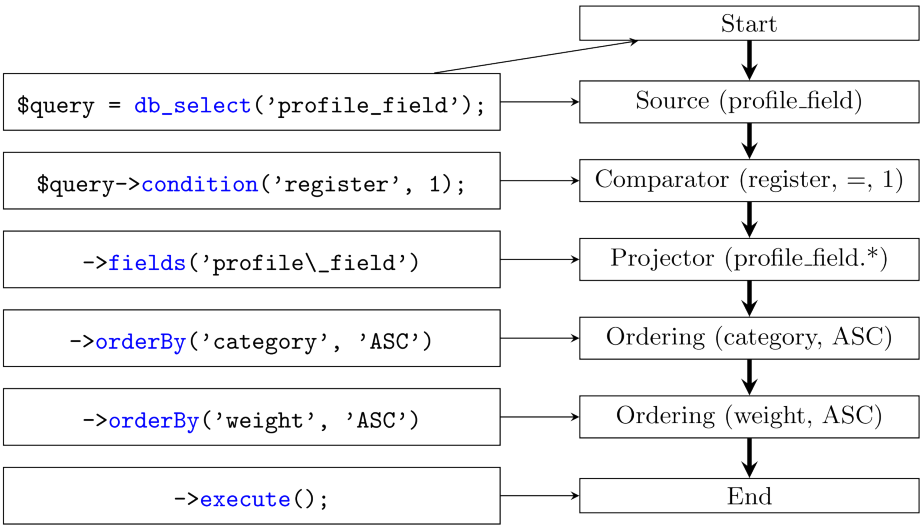


Fig. 4: Abstract query representation of the third QVG path of object-based query of Fig. 1. On the left we have the source code that constracts the query and on the right we have the AQR nodes with their parameters.

ones here. Therefore, we can use the Abstract Query Representation, and see if there are any AQRs with the exact same operators, carrying the exact same ADMO parameters. Since we need only one of those queries, we eliminate the AQR duplicates. This is a rather simple task, since a simple walk over the Abstract Data Manipulation Operators of the Abstract Query Representation can provide us the information needed for the comparison.

4 From AQRs to Concrete Query Representations

The Abstract Query Representation would be of small use, if we could not translate the AQRs to concrete queries for a specific query environment, so, the next step of our method is to be able to transform the model representation of AQR to a text-based representation of a concrete query environment. The query environments on which we have up to now performed this model-to-text transformation are SQL and MongoDB.

To export the Abstract Query Representation to a concrete language we need to gather the nodes of the AQR in groups and use those groups for the output parts of each language. Due to lack of space we do not formally describe any of the export methods of SQL and MongoDB. The interested reader is kindly referred to: `http://cs.uoi.gr/~pmanousi/publications/queryExtraction/`.

5 Evaluation

We have evaluated our method using two ecosystems written in *different programming languages*. The first ecosystem we used is the Clementine[4] music player project, which is written in C++ and it stores the information of the tracks of the music library of its users in a database. The second ecosystem is Drupal, which is the most popular CMS on sites with heavy traffic[5]. Drupal[6] is written in PHP and it stores the contents of the web pages it manages in a database. Table 2 contains more details, such as the number of lines of code, the number of files, and the number of subfolders of the projects we used for our evaluation.

Table 2: Projects' descriptions and queries distribution per project

Project	Lines of code	Files	Sub-folders	Variant queries	Fixed queries	Total
Clementine	210053	3072	159	10	14	24
Drupal	325421	1096	137	10	84	94

Effectiveness We need to verify the extent to which our method retrieves and correctly reconstructs queries from the application scripts of the ecosystem. The performance measures for this kind of assessment are recall and correctness. *Recall* is defined as the fraction of the retrieved queries of each file over the actually existing ones. *Correctness* is defined as the fraction of the correctly reconstructed queries over the retrieved ones. A correct reconstruction of a query involves (a) retrieving all its structural parts and (b) assembling them correctly, in order to result in a correct and complete query. Table 2 depicts the distribution of queries that were either single path (fixed) queries or produced due to branch and loop statements of the host language (variant queries).

Recall To assess recall, we need to *manually* verify the percentage of queries that our method extracts with respect to the queries that actually exist in the code. Due to the vastness of the task, we have sampled the 10% of the database-related files. This is a standard practice in the software engineer community whenever the size of a project is too large for full manual inspection. We manually inspected the code of the evaluated files and we were unable to find any other query, besides the ones that our tool reported. In the functions that were repeating a query in one or more places in their source code, we reported only one occurrence of the query, since there was no variation. If a query changes, then we report the "new" query (the modified one) as well. Our manual inspection was further supported by automated searches in the source code. For Clementine, we decided to focus on a single table of the database. Then, we can search for all occurrences of the table's name. For Drupal, we took advantage of the fact that there are specific functions for querying the database, as prescribed by

[4] `https://www.clementine-player.org/`
[5] See `http://w3techs.com/technologies/market/content_management`
[6] `http://ftp.drupal.org/files/projects/drupal-7.39.tar.gz`

Table 3: Breakdown of generated queries per query class.

	Query class	Drupal-7.39	Clementine 1.2.3
Valid:	*all* parts fixed	28/94 (*29.7%*)	5/24 (*20.8%*)
Valid:	variable *values*	61/94 (*64.9%*)	14/24 (*58.3%*)
	overall	89/94 (**95.6%**)	19/24 (**79.1%**)
Invalid:	variable *structure*	05/94 (*04.4%*)	05/24 (*20.9%*)

its manual (both for string-based and for object-based queries): `https://api.drupal.org/api/drupal/includes!database!database.inc/function/`).

Correctness Regarding the correctness of our method, we examined the sample files on whether the queries that were translated to SQL query environment were correct or not. The correctness for the Drupal project is 95.6% and for the Clementine project is 79.1%. To explain what we considered as a correct query we created the following taxonomy of query classes:

1. *Fixed structure*: This class has the queries that can be translated to one of our concrete query environments and run without issues.
   (a) *All parts fixed*: queries that have no variable at all
   (b) *Variable values in "filtering"*: queries that contain a variable that gets its value at execution time but does not intervene with the query structure. In most cases this is a variable that is the second part of a comparison. In our reference example of Fig. 1, Line 7 contains the $*category* variable which can be replaced by a value, producing a valid query.
2. *Variable structure*: in this class we have variables that alter the query structure. This means that the data providers are unknown to us, so in order to produce a valid query we needed to know in advance the values of the parameters that were given to the calls of those Callable Units.

Table 3 contains the number of queries that belong to each classification for each one of our case studies, which consequently provide the precision measurement for our method. Observe that the internal breakdown for the different categories (rows in the table) is quite different for the two cases. However, *we do achieve 100% correctness and recall for the two first categories*. For the last category, we fail to produce an abstract representation due to the fact, that many times the variable structure refers to a variable table in the FROM clause that is assigned at runtime. A flexible handling of such occurrences (with variable tables involved) is part of future work.

User effort As previously stated at Section 3, there is a preprocessing step that is needed in order to translate the projects API database-related functions to Abstract Data Manipulation Operators. In Table 4 we describe the user's effort for the two projects that we examined. The effort is measured in the number of functions that needed translation from the project's API, and in the lines of

Table 4: User effort (Number of functions to translate / Lines Of Code)

Project	API func./LOC	Host lang. (func./LOC)	Method fixed input
Clementine (C++)	4/59	9/341	11
Drupal (PHP)	11/251	9/347	11

code that were written for the translation of those API functions to Abstract Data Manipulation Operator.

6 Related Work

The state of the art on query extraction includes some interesting techniques that facilitate various engineering tasks like error checking, fault diagnosis, query testing prior execution, and change impact analysis.

Specifically, Christensen et al. [3] propose an approach that identifies type and syntax errors in Java source code, due to queries are constructed through string concatenation. To this end, the authors perform static source code analysis, based on flow graphs and finite state automata. Gould et al. [6,7] use slicing to identify the SQL related parts in Java source code. Then, all the variations of a query are formed and tested for type (using the DB schema description) and syntax errors. In a similar vein, Annamaa et al. [8] propose a method for testing database queries before their actual execution. The method identifies SQL queries embedded into Java source code, via searching for a given set of related functions. Van den Brink et al. [9] use control and data-flow analysis to assess the quality of SQL queries, embedded in PL/SQL, COBOL and Visual Basic source code, while Ngo and Tan [10] rely on symbolic execution to extract database interaction points from PHP applications. Maule et al. [1] employ query extraction to identify the impact of relational database schema changes upon object-oriented applications. The proposed method targets C# applications and is based on data-flow analysis, performed via a k-CFA algorithm. Finally, Cleve et al. [11] propose a concept location technique that starts from a given SQL query and finds the specific source code location where the query is formed. This effort targets Java source code that uses JDBC or Hibernate.

Overall, although the existence of all these methods verifies the importance of the problem, the state of the art has dealt with query extraction (a) in a language-dependent way and (b) as the means, but not the main focus of research. Differently from the state of the art approaches, *we propose a general-purpose query extraction method that clearly separates technology-specific from technology-independent aspects*. Our method extracts *all* the variants of the queries that can be generated at runtime and *produces query representations in more than one target query languages*.

7 Conclusion and Future work

We have presented a method that identifies the embedded queries within a database-related software project, independently of host language and programming style. Our method constructs every variation of a query that can be produced due to branch and loop statements of the source code's host language during runtime, and represents the queries in a generic, language-independent way that facilitates the exporting of these queries to more than one concrete query environments. As next steps, we intend to improve the effectiveness of our method, by capturing more flexible query construction patterns. We also consider to extend the number of host languages (besides PHP and C++) for our method's usability.

References

1. Maule, A., Emmerich, W., Rosenblum, D.S.: Impact analysis of database schema changes. In: Proceedings of the 30th International Conference on Software Engineering (ICSE). (2008) 451–460
2. Manousis, P., Vassiliadis, P., Papastefanatos, G.: Automating the adaptation of evolving data-intensive ecosystems. In: Proceedings of the 32nd International Conference on Conceptual Modeling (ER). (2013) 182–196
3. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Proceedings of the 10th International Static Analysis Symposium (SAS). (2003) 1–18
4. Gallagher, K., Binkley, D.: Program slicing. In: Frontiers of Software Maintenance, 2008. FoSM 2008., IEEE (2008) 58–67
5. Cleve, A., Henrard, J., Hainaut, J.: Data reverse engineering using system dependency graphs. In: Proceedigns of the 13th Working Conference on Reverse Engineering (WCRE). (2006) 157–166
6. Gould, C., Su, Z., Devanbu, P.T.: Static checking of dynamically generated queries in database applications. In: Proceedigns of the 26th International Conference on Software Engineering (ICSE). (2004) 645–654
7. Wassermann, G., Gould, C., Su, Z., Devanbu, P.T.: Static checking of dynamically generated queries in database applications. ACM Trans. Softw. Eng. Methodol. **16**(4) (2007)
8. Annamaa, A., Breslav, A., Kabanov, J., Vene, V.: An interactive tool for analyzing embedded SQL queries. In: Proceedings of the 8th Asian Symposium on Programming Languages and Systems. (2010) 131–138
9. van den Brink, H., van der Leek, R., Visser, J.: Quality assessment for embedded SQL. In: Proceedings of the 7th IEEE International Conference on Source Code Analysis and Manipulation (SCAM). (2007) 163–170
10. Ngo, M.N., Tan, H.B.K.: Applying static analysis for automated extraction of database interactions in web applications. Information & Software Technology **50**(3) (2008) 160–175
11. Nagy, C., Meurice, L., Cleve, A.: Where was this SQL query executed? a static concept location approach. In: 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER). (2015) 580–584