A FRAMEWORK TO DISTIL SQL QUERIES OUT OF HOST LANGUAGES IN ORDER TO APPLY QUALITY METRICS

A thesis

Presented to the

Department of Information and Computing Sciences

Utrecht University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Huib J. van den Brink

January 2007

UTRECHT UNIVERSITY

The Undersigned Faculty Committee Approves the

thesis of Huib J. van den Brink:

A framework to distil SQL queries out of host languages

in order to apply quality metrics

---

prof. dr. S. D. Swierstra, Chair
Software Technology Group

---

dr. J. Hage, Thesis Supervisor
Software Technology Group

---

dr. T. Kuipers, Chief Technology Officer
Software Improvement Group

---

ir. R. C. van der Leek, Thesis Supervisor
Software Improvement Group

Approval Date

ABSTRACT OF THE THESIS

A framework to distil SQL queries out of host languages
in order to apply quality metrics
by
Huib J. van den Brink
Master of Science in Computer Science
Utrecht University, 2007

Many software systems that use a database lack maintainability and adjustability, for the produced queries are hard to reconstruct and inspect. For instance, the relations between the queries, established by the host language, are not easy to identify. This information however is vital when adapting the application in order to prevent unexpected behaviours. The area of this master thesis is within that of the program analysis. We focus on queries embedded in a host language, while by reconstructing and analysing the queries, insight is gained in its complexity. Also the interaction between the queries and the host language is taken into account. For instance, the explicit structure and relations within a database often are complemented with implicit relations established in the application itself. Therefore analysing the application with respect to the SQL queries it contains, is needed to detect and distil those relationships. Analysis techniques, such as constant propagation and partial evaluation, are instrumental for reaching this goal. Finally, a framework is constructed for extracting SQL queries out of host languages and for indicating the quality of the queries. The framework therefore includes means for indicating the quality of the relations between data, determined from how SQL query results are used.

ACKNOWLEDGEMENTS

First of all, I would like to thank dr. Tobias Kuipers for offering me this opportunity to learn many things about software quality, by providing me this thesis project at the Software Improvement Group. Thanks to my supervisor ir. Rob van der Leek for supporting and guiding me in the technical subjects. And finally I would like to express my appreciation for all the other colleagues at the SIG for their care and accompany, like Dr. Harro with his certificates and weekly Wednesday joke.

Thanks to the supervisor on behalf of the Utrecht University, viz. dr. Jurriaan Hage, for guiding me in the right direction and reviewing this thesis. Also I would like to thank my fellow students, both Dutch and international, for their cooperation in the numerous projects we performed.

Last but certainly not least, I would like to thank my parents and sisters. All of their constant love and moral support has provided me with the motivation to persevere.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

This thesis project was performed in Amsterdam, The Netherlands, on behalf of the Software Improvement Group and approved by the Universiteit Utrecht.

1.1 DESCRIPTION OF THE COMPANY

The Software Improvement Group (SIG), is involved in the area of software change management and has the activities of assessing software products and supporting the process of development. This field entails quality control by applying build management, performing renovation management, software monitoring and risk assessment as well as other activities. Large organizations often use software systems that were constructed in the past, possibly by several vendors. As time goes by, there are new requirements and or demands for adjustments. Then the issue arises of losing oversight of the design and architecture, which however once was present, during construction. In order to revive such knowledge, and in order to be able to give an estimation about the impact and risks of specific changes, analysis of the source code has to be performed. The SIG looks at both the process characteristics and the consistency of the development progress of currently performed software activities as well as assessing legacy systems.

1.2 MOTIVATION

Many applications depend on a database system. The communication and relation with the database system is usually expressed via the Structured Query Language (SQL). The SQL queries are used to access and manipulate data in the database. As a result a great number of source files contain many queries in different ways. Depending on the programming language used, the queries are either isolated from, or embedded within the host language.

When queries are embedded in a host language, it is not easy to obtain an overview of the queries that are produced. However, when the queries can be collected and reconstructed, insight can be gained into the quality of the individual queries. Taking this one step further, the interaction with the host language can be taken into account, because applications can use queries and query results in an undesirable way. This helps in saying something about the maintainability, adaptability and the testability of an application with respect to the database access. And in order to improve the usage of queries in an application, one needs to know which queries exactly require refactoring, and what the factors are that makes it complex. This insight also helps in a maintenance phase, if one knows what to look for, improvement can be performed efficiently by focussing on the right aspects.

This master thesis project consists of a few closely related parts. The first aim is to design a model to extract SQL queries out of host languages as much as possible, because in order to inspect the queries they first have to be reconstructed. Then, as a second part, the implicit relations that are not present in the database scheme, but are established by the implementation and semantics of an application, are detected. Finally quality metrics

expressing the results of the defined analyses are defined and presented. As a result, a framework enabling and supporting these analyses is delivered.

In order to obtain an indication of the accuracy and effectiveness of the defined measurements, a case study is presented, showing how the metrics reflect the situation as it is present in the application. The case study was performed on a few commercial software applications, as is explained in section 8.

1.3 RELATED WORK

The field of measuring the level of quality of queries, by defining metrics for them, is rather new. Only very little research[16, 15, 14, 17] about this topic has been performed. This related work however only focuses on the isolated queries without taking the interaction with the host language into regard.

Work that does take Java as a host language into account, is performed in [18]. It differs from this thesis in the sense that it is very restrictive. This thesis provides a solution that is able to handle all common SQL dialects, while the host language could be any imperative programming language. Functional languages however were not taken into account, because in practise they rarely are used for commercial software applications. Further, the sources provided to the SIG in order to assess them, almost always are incomplete and do not contain executables or class files. For this reason the model designed and implemented in the constructed framework requires only the minimum presence of source code.

Other important research has been performed by J. Henrard. His thesis Program Understanding in Database Reverse Engineering[20] defines and describes techniques for performing data structure extraction. However, the focus here is more on reverse engineering and less on quality measurements and metrics.

Other areas with respect to database usage, specifically aim at means to structurally avoid bad practises, rather than detecting those occurrences in existing software products.

Designing a database and obtaining subsets of the data contained by the database is not trivial as shown by the large number of different approaches applied by both experienced and inexperienced programmers. The creation of queries in many programming languages involves great freedom. Closer integration with the types and relations present in the database scheme then improves the quality and even performance of applications using and constructing queries.

The domain specific language HaskellDB[5] applies type inferencing and type checking within the host language with respect to the types of the database fields. These compile time checks prove to be valuable, although they still leave room for unnecessary performance loss and undesirable structures creating artificial relations. However, mechanisms have been developed to improve the administration of the relations. Abstracting out the management of the relations, like frameworks as Hibernate[2] and JDO[21] do, tend to prevent programmers from applying bad practises as for example a missuse of foreign keys. This type of research

however has a completely different scope than this thesis, because it does not detect weak relations and only tries to provide means to improve the database mapping.

1.4 THESIS STRUCTURE

This thesis consists of two major subjects. The first part handles the distillation and reconstruction of dynamically created queries, in order to enable the quality analyses to process them further. The second part takes the queries and the host language, i.e. the surrounding source code, into regard, by locating undesirable relations among queries. This is performed by looking at the usage of the query results and by looking at the dependencies the different queries have with respect to each other.

This thesis than is structured as follows. First a background is given in chapter 2 about the usage of the structured query language in a variety of programming languages. This helps to gain insight in the various structures and analyses to perform. Further the different tools and techniques are discussed that were used for fulfilling this thesis.

Chapter 3 then explains how queries that dynamically are constructed by the host language, can be reconstructed statically. In chapter 4 the relations queries can have among each other are considered, while chapter 5 discusses the quality impacts of those relations.

The design, the different components and the internal representations used in the constructed framework are explained in chapter 6. For instance the term 'query container', which contains information about the collected queries and is used at various places, is explained in more detail there.

The different quality metrics that are performed on the queries and the interaction with the surrounding source code of the host language, are specified in chapter 7. In order to obtain an indication of the accuracy and effectiveness of the defined measurements, a case study is presented in chapter 8. This chapter then shows how the metrics reflect the situation as it is present in the application. And in order to be reliable, the case study was performed on a few commercial software applications, showing how the analyses work in practise. Finally the conclusion is given in chapter 9.

A paper[6] containing the findings of this thesis, was submitted to the Eleventh European Conference on Software Maintenance and Reengineering (CSMR 2007) and is added to the appendix.

CHAPTER 2
ANALYZING QUERIES EMBEDDED IN HOST LANGUAGES

This chapter elaborates on the techniques, concepts and programming languages related to this thesis project. The topics addressed are SQL and its relation to the programming language in which it is embedded. Further an overview of the parsers and lexers at my disposal is given. Finally, some common analysis techniques are discussed.

2.1 STRUCTURED QUERY LANGUAGE

SQL[22] is a language to express operations to be performed on data contained in a database. This involves actions that add, modify and retrieve data from relational database management systems[8]. The four basic operations to be performed are called CRUD, which stands for Create, Read, Update and Delete.

2.1.1 Embedment in programming languages

Many programming languages contain embedded constructs in order to access a database. In most cases a SQL query, of some dialect, is situated in the host language and can be executed either statically at compile time, as in a preprocessor, or dynamically at runtime. The structure of the embedment can vary from highly formalized to an unconstrained construct.

An example of a formal embedding can be seen in the programming languages COBOL, C and in some occasions Java, i.e. in the cases where SQLJ, Hibernate or other supporting frameworks are used.

```
1  // declare host variables
2  int id = 2;
3  String firstName = null, lastName = null;
4  java.sql.Date dob = null;
5
6  // perform SELECT to get the customer details for
7  // the customer #2 from the customers table
8  #sql {
9    SELECT
10     first_name, last_name, dob
11   INTO
12     :firstName, :lastName, :dob
13   FROM
14     customers
15   WHERE
16     id = :id
17 };
```

Fragment 2.1. SQL located in special blocks

The example above shows a query as it would be used in SQLJ. As can been seen, the whole query is situated within a special `#sql` block. The final SQL query then is constructed by a preprocessor. But this doesn't always have to be the case, as for instance Hibernate and JDO solve everything at runtime.

The opposite of the formal notation, i.e. a loosely construct as used in PHP, JavaScript and sometimes in Java, is shown below.

```
1 String table = "my_table";
2 String query  = "SELECT * FROM " + table;
3        query += "WHERE id = " + id;
4 ResultSet rs = stmt.executeQuery(query);
```

Fragment 2.2. Query build up by strings

Here the query is constructed by concatenating pieces of a string together, forming a legal SQL query. The concatenated query fragments tend to be error prone due to a lack of any syntactic and type checking.

2.2 LANGUAGE SPECIFICS

Since this thesis project copes with a variety of languages, this section explains the languages that are taken into account. The focus lies on the embedment of the SQL queries and the usage of the results of these queries.

The reason why we focus on the languages Visual Basic, PL/SQL and COBOL are as follows. This set represents three generations of software engineering. The oldest is COBOL, the newest Visual Basic. Besides this, the three belong to different categories and disciplines of languages. In PL/SQL for instance the queries and language constructs are closely related, for PL/SQL actually is SQL extended with some basic constructs, like for instance the `if` statement. In Visual Basic however the queries, or parts thereof, are placed in strings. COBOL uses the formalized way of embedding SQL queries, whereas statements, assignments, calls and some basic arithmetic is defined outside such blocks. Most COBOL applications are legacy software, enabling analyses to gain insight. Visual Basic projects usually contain rapidly developed products with a lack of quality and maturity.

2.2.1 Visual Basic 6

Visual Basic[28] is a language containing functions and subroutines. Where functions return a value and subroutines only perform operations, both flavours can take parameters by value and by reference. This should be taken into account when analysing variables and constructing call graphs. Unlike Visual Basic .NET, Visual Basic 6 and Visual Basic Script do not have inheritance, but overloading and importing other classes is allowed.

As shown below, queries are constructed by gluing pieces of string together, using variables from outside.

```
1 Set rsService =
2       Database.Get( "SELECT I.ISPID, VPO.SERVICEID " _
3                   & "FROM VPORDERS VPO, ISP I " _
4                   & "WHERE I.ISPNAME = VPO.ISP " _
5                   & "AND VPO.ORDERID = ?", _
```

```
6                           intOrderID )
```

Fragment 2.3. Query construction in Visual Basic

The '_' character means that the statement continues on the next row, while the '&' glues the pieces of string together. Both the values for `I.ISPID` and `VPO.SERVICEID` are stored into a set, that can be read by the rest of the application. To insert the `intOrderID` into the query, the question mark token acts as a placeholder. By providing the value then as subsequent parameter, the value is propagated, i.e. interpolated, into the query before execution.

For the usage of the result, the following structure is quite common.

```
1 Dim intISP
2 intISP = 0
3 If stService.RecordCount = 1 Then
4     intISP = stService.ISPID
```

Fragment 2.4. Using the result of the query

First a variable is declared, then the number of results is checked and finally the resulting value is assigned and can be used throughout the rest of the application. This check normally is used to ensure that there is a result that can be read and used without getting an exception.

In Visual Basic stored procedures can be invoked. Stored procedures are queries stored in the DBMS and can be invoked for execution by applications. The different database systems then provide means to express some logic in the stored procedures. Oracle for instance defined PL/SQL (Procedural Language/Structured Query Language) in order to facilitate some logic in stored procedures. Executing those stored procedures often is performed by using ActiveX Data Objects (ADO). Consider for example the following stored procedure.

```
CREATE PROCEDURE GetUserId @userid INT, @status INT OUTPUT AS..
```

Fragment 2.5. The definition of a stored procedure

In combination with some piece of plain SQL defined within the stored procedure, the following example will execute this stored procedure with some user id as parameter. The assumption made here, is that a database connection has been established and a record set has been created.

```
Recordset.Open "EXEC GetUserId " & UserId, Connection,
                                  adOpenStatic, adLockOptimistic
```

Fragment 2.6. Invoking a stored procedure

A more elegant solution, however, is to use the Command object and output parameters as shown below.

```
1 Dim Cmd As New ADODB.Command
2 Dim Param As ADODB.Parameter
3
4 Cmd.CommandText = "GetUserId"
5 Cmd.CommandType = CommandTypeEnum.adCmdStoredProc
6
7 Set Param = Cmd.CreateParameter("userid", adInteger,
```

```
                                          adParamInput, 8, UserId)
Cmd.Parameters.Append Param

Set Param = Cmd.CreateParameter("status", adInteger,
                                       adParamOutput, 8, Status)
Cmd.Parameters.Append Param

Set Cmd.ActiveConnection = Connection
Set Recordset = Cmd.Execute()
```

Fragment 2.7. Using a Command object

First, the stored procedure is located. Then the user id is given as input parameter, and afterwards the result is stored in a variable called `Status`. Finally the actual execution of the stored procedure is initiated by invoking the execute method.

Finally, I would like to close this section with an example showing the variety in which stored procedures can be used.

```
Dim objDB as Database
Dim lng as Long
Dim objRS as Recordset

'For stored procedures that don't return rows
Set objDB = DBEngine.Workspaces(0).OpenDatabase("", False,
              False, "ODBC;_ DSN = macdsn;uid=test;pwd=test:")

'Stored procedures which return rows
lng = Db.ExecuteSQL("StoredProcedure_Name")
Set Ss = Db.OpenRecordset("StoredProcedure_Name",
                             dbOpenSnapshot, dbSQLPassThrough)

Column1.text = objRS(0)             'Column one
Column2.text = objRS!ColumnName     'Column two
Column3.text = objRS("ColumnName") 'Column three
```

Fragment 2.8. Variants in accessing stored procedures and their results

First, a connection with the database is established. Then the stored procedure is executed, followed by the retrieval of the result, which is stored in a record set. This record set is opened and the results can be read. Notice the different ways for retrieving the results from the record set and the way to access the stored procedure itself.

2.2.2 PL/SQL

The basic construct in PL/SQL[9, 13] is a block. In a block, constants and variables can be declared and used in order to complement the query and to store the query results. Statements in a PL/SQL block include SQL statements, loops and other control flow constructs, exceptions and calls to other blocks. PL/SQL blocks that specify procedures (without result value) or functions (having a result value) can be grouped into packages, exposing only a subset of these. Parameters can be either of type '`IN`' (default), '`OUT`' and '`IN OUT`', indicating the directions of the data flow.

PL/SQL is a tuplebased language, for the result of a query is placed into a tuple in order to

make it available to the rest of the application. The following example shows a combined SQL statement and its usage.

```
BEGIN
   IF (df$jnd.ID = FALSE OR df$rec.ID is NULL) THEN
      SELECT MH_MG_SEQ.nextval
      INTO   df$rec.ID
      FROM   DUAL;
      df$jnd.ID := TRUE;
   END IF;
EXCEPTION WHEN OTHERS THEN
      df$errors.push(SQLERRM, 'E', 'AR', SQLCODE,
                     'df$MH_MTRING.up_autogen.ID.others');
      df$errors.raise_failure;
END;
```

Fragment 2.9. A query embedded in PL/SQL

This example shows a query selecting one row out of a database. The computed result is placed into a one element tuple `df$rec`. Note however that the tuple may contain multiple elements, depending on the `SELECT` part. The select-into construct however does not suffice when more than one row, resulting in more than one tuple, is the result of the evaluation. In that case a cursor is introduced.

The example below shows the way loops access the results produced by the query.

```
BEGIN
   li_cursor := dbms_sql.open_cursor;

   FOR rec IN
   (
      SELECT object_name
        FROM user_objects
       WHERE object_type = 'PROCEDURE' AND
             status = 'INVALID'
   )
   LOOP
      BEGIN
         dbms_sql.parse( li_cursor,
                        'ALTER PROCEDURE ' || rec.object_name
                        || ' COMPILE',
                         dbms_sql.native );
         li_result := dbms_sql.EXECUTE( li_cursor );
      EXCEPTION
         WHEN OTHERS THEN
            errmessage := sqlerrm;
            dbms_output.put_line('compile procedure: ' ||
                                 rec.object_name || '; ' ||
                                 errmessage );
      END;
   END LOOP;

   dbms_sql.close_cursor( li_cursor );
END;
```

Fragment 2.10. Accessing the results of the queries using a cursor

The `for` construct enables the loop to access one row, i.e. tuple, at a time. As can be seen, nested blocks are allowed and the construct `'abc' || 'def'` concatenates both pieces to the string `'abcdef'`. The result of the query presented above is a list containing `object_name`'s. The loop then considers one row at a time. This single `object_name` then is used to alter an existing, already defined, procedure. In this way defined procedures, containing blocks, can at runtime be overwritten with new definitions. In the example above the object to be compiled is, in some way, redefined. Because the runtime behaviour is affected by the contents of the database, it is impossible to statically analyse such queries.

Exceptions can be of different types. For instance when no tuple is returned, the exception `NO_DATA_FOUND` is thrown. But it is also possible to raise user-defined exceptions. The `OTHERS` keyword applies to all of them, and can be used to handle an exception regardless of its type.

2.2.3 COBOL

In COBOL[12] queries are sited within EXEC SQL blocks. An example of a simple select query is shown below.

```
1  EXEC SQL
2       SELECT
3               CINFCSTN,
4               CINFCSNM,
5               CINFBAND,
6               CINFFXTR
7       INTO
8               :CINFCSTN,
9               :CINFCSNM,
10              :CINFBAND,
11              :CINFFXTR
12             FROM IABSTJNF
13             WHERE CINFCSTN = 3325
14 END-EXEC.
```

Fragment 2.11. An EXEC SQL block in COBOL

The begin and end are clearly marked with the keywords `EXEC SQL` and `END-EXEC`. In the blocks one or more queries, separated by semicolons, are allowed, while nesting EXEC SQL blocks is not. The variable names starting with an colon, are actually references to the variables of the host language. So `CINFCSTN` is the name of a column in the table `IABSTJNF`, whereas '`:CINFCSTN`' is a COBOL field. Therefore the results of the select are directly pushed into the variables. Finally the period signifies the end of a COBOL sentence.

For pushing values into a database, the same kind of construct is used:

```
1  EXEC SQL
2       INSERT INTO IAFMMUDB
3       VALUES (:IAFMCDAT, :IAFMUYDU, :IAFMSECS, :IAFMCGNR,
4               :IAFMBSRT, :IAFMSELN, :IAFMNVAL, :IAFMECKD,
5               :IAFMPMS1, :IAFMPMS2, :IAFMPMS3, :IAFMPMS4,
6               :IAFMPMS5, :IAFMPMS6, :IAFMPMS7, :IAFMCEVV)
7  END-EXEC.
```

Fragment 2.12. Inserting a row into an table

And the same holds for updating values:

```
1 EXEC SQL
2      UPDATE IABSTJNF
3                         SET CINFBREC = :CINFBREC,
4                             CINFBAND = :CINFBAND,
5                             CINFFXTR = :CINFFXTR
6                          WHERE CINFCSTN = 3325
7 END-EXEC.
```

Fragment 2.13. Updating values in the database

All the other possible database operations are performed in a similar way.

The usage of the variables however is subject to less straightforward constructs.

```
1 05 UB07BAND              PIC 9(8).
2 05 UB07ECVV              PIC S9(9)V99 COMP-3.        MIG-98
3
4 MOVE UB07BAND TO CINFBAND.
5 MOVE UB07ECVV TO CINFFXTR.
```

Fragment 2.14. Variables and types in COBOL

Here, `PIC 9(8)` stands for any number between 0 and +99,999,999, the S9(9)V99 means that the variable `UB07ECVV` can contain any number from 999,999,999.99 to +999,999,999.99. Finally, the MOVE command transfers data from one area of storage to another. So after this program point the variable effectively is renamed.

Finally I should mention that, based on the experiences within the SIG, it is quite uncommon to use stored procedures in COBOL applications.

2.3 ANALYSIS

Since much research already has been performed in the field of program analysis[25] many generic platforms have arisen in order to support common analysis techniques. This section briefly discusses common techniques and considerations to be made.

The field of automatic program analysis often is used for optimisations performed by transformations benefiting the performance. Soundness in this setting means the preservation of program semantics, which of course is crucial for the semantics of the execution. In for instance bug detection systems[33] however, the meaning of soundness somewhat differs. In this setting soundness means the ability to detect *all* possible errors, even when this implies finding false positives. Besides this, it is common knowledge that analysis problems often are statically undecidable and therefore approximations are used. This suggests that unsound analyses, which don't actually have to be unsafe, still can provide useful information. At least by eliminating the number of points to manually inspect, insight can be gained in for instance quality, that otherwise never would be possible by hand. So in this situation false positives are allowed to some extent. The game is about sometimes being wrong but sound, i.e. detecting all possible errors including false positives. Or the other option, being safe, i.e. never being

wrong and thus never have false positives, and risking the chance of being incomplete by not detecting every bug. Finally the cost of all the calculations should be taken into account.

As far as the SIG is concerned however, in many cases the provided source code is a snapshot, and therefore incomplete. It then is not hard to imagine that many of the potential useful information is not present, and that performing whole-program analyses would make no sense. Especially when interprocedural analysis is performed, so depending on related components, the absence of parts does complicate the process of finding evidence for assumptions. For the dead code elimination analysis for instance, not everything that appears to be 'dead', also really is.

A common topic within the field of program analysis, is the undecidable halting problem. Given a description of a program and its initial input, it is not possible to determine if the program, when executed on its input, ever completes and stops. A pure SQL query however always terminates, but the result used in the rest of the application perhaps not, due to loop structures, method calls and possibly thrown exceptions and so on.

2.3.1 The field of SQL and the quality of the queries

Very little research and documentation is currently available about heuristics and metrics regarding SQL and their relation to the rest of the application. However test case coverage calculation[10] uses analysis within the same area as this thesis. Commonalities exist in the first part of the process, while the final goal and focus differ. Identifying this resemblance, the similarities serve a different purpose.

The field of reverse database engineering by program understanding[20] also provides useful information with regard to relations and dependencies in both the queries as well as the surrounding application code. Legacy software sometimes consists of deteriorated structures due to the changes over time. Besides this, legacy systems often lack of up-to-date documentation[24]. Reconstruction of relationships and structures here proves to be vital. As a consequence, research is performed in this area, which proved to be quite useful for this thesis project.

With respect to the research performed by Brass and Goldberg[16, 15, 14, 17], that only addresses the semantics of the queries, the objective of this thesis goes one step further by involving the host language the query is embedded in. Instead of analysing isolated queries and focussing on creating a better query producing some equivalent query result, the focus of this thesis is on the semantics of the operations performed within the host language. By this shift of focus, matters like incorrect use of foreign keys, by artificially creating them within the host language, are able to get recognized.

While the focus differs, commonalities do exist. Both this thesis and the research performed by Brass and Goldberg assumes syntactically correct queries and the absence of knowledge about the task the query is supposed to solve.

2.3.2 Control flow and data flow analysis

Data flow analysis (DFA) represents source code as a graph. Nodes are elementary blocks of the source code, i.e. a straightline piece of code without any jumps like ' `x = y`', and edges describe how control might pass from one block to another. It should come as no surprise, that for constructing this information, a control flow graph is used.

A control flow graph (CFG) shows all possible paths present in a program by determining what elementary blocks may lead to what other elementary blocks. An if statement for instance has two forward flows, the path for `true` and one for the `false` condition. Each node in the graph represents a basic block, while the directed edges are used to represent jumps in the control flow.

One instance of DFA is the reaching definition analysis, as discussed in the next section.

2.3.3 Use-Definition chain

UseDefinition chains[25] model the relationship between the definitions of variables and their uses in a sequence of assignments. Basically it consists of a set of all the instructions that are required to set the variable for each variable reference. It's counterpart, the definitionuse chains links definitions with all their uses.

Considering the following example, the chains as shown below are constructed.

$[int\ y = 0;]^1$
$[int\ x = 0;]^2$
$[x = x + y;]^3$
$[y = x + 1;]^4$
$[x = 1 - x;]^5$
$[storeToDb(x);]^6$
$[x = 3;]^7$

	x	y
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\emptyset$
3	$\{2\}$	$\{1\}$
4	$\{3\}$	$\emptyset$
5	$\{3\}$	$\emptyset$
6	$\{5\}$	$\emptyset$
7	$\emptyset$	$\emptyset$

UseDefinition

	x	y
1	$\emptyset$	$\{3\}$
2	$\{3\}$	$\emptyset$
3	$\{4, 5\}$	$\emptyset$
4	$\emptyset$	$\emptyset$
5	$\{6\}$	$\emptyset$
6	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$

DefinitionUse

The usedefinition maps for each basic block the points where the used variables are able to get a value assigned. The third block for instance uses both the variables `x` and `y`, which are assigned a value in the blocks `1` and `2`. Because of the control flow, a value can be assigned at multiple places. In the case of an `IF` statement for example, the assign can take place at both branches. The definitionuse at the other hand constructs a mapping between the variable that is assigned a value, and the locations where that assigned value is used. So the first `x` in the third block is used in both block `4` and `5`, but not in `6` since it is reassigned in `5`.

By using the information available in the usedefinition chain, a variable occurring within a query can be traced back to its type and perhaps some more static information. The definitionuse chain, on the other hand, can provide the locations where the results of the queries are used.

2.3.4 Constant propagation

Constant propagation calculates for each program point if the variables contained by the subsequent block have a constant value when the execution reaches that point. If this is true, then the variables can be substituted by their values. This technique is called constant folding, which simplifies the arithmetical expressions of which all operands are known, by calculating the result and replacing the arithmetical expression by that result. The following example illustrates the benefit of this.

```
1 int a = 30;
2 int b = 9 - a / 5;
3 int c;
4
5 c = b * 4;
6 if(c > 10)
7     c = c - 10;
8
9 return c * (60 / a);
```

Fragment 2.15. Source without constant propagation

When performing a forward constant propagation, the previous knowledge is used throughout the rest of the expressions. This would result in the following.

```
1 int a = 30;
2 int b = 9 - 30 / 5;
3 int c;
4
5 c = 3 * 4;
6 if(12 > 10)
7     c = 12 - 10;
8
9 return 2 * (60 / 30);
```

Fragment 2.16. The result after propagating constants

First the assignment to `b` can be complemented by propagating the value of `a`. Now the `b` is known, the first assignment to `c` could be made concrete. Then the `if` branch gets known and likewise the newly assigned value to `c`. Finally the result is completed.

However if the `if` statement remains nondeterministic, the value of `c` also gets nondeterministic. In that case the `c` could contain a set of valid values, one value for entering the true branch of the `if`, and one for not entering it.

For extracting SQL this technique means that constants and other information, like type information, can be propagated into the strings building the queries.

```
1 public String tablePrefix = "gui_";
2
3 public String nameTable = tablePrefix + "users";
4 int standardRetirement = 65;
5 int increaseRetirement = 2;
6 String retirement = standardRetirement + increaseRetirement + "";
7
8 String query = ""
```

```
9         + "SELECT * "
10        + "FROM " + nameTable + " "
11        + "WHERE age > " + retirement + ";";
```

Fragment 2.17. Query benefiting from constant propagation

In this example the name of the table used is the combination of two provided strings, while `retirement` is a direct reference to a number. The resulting query after performing the constant propagation and folding, is shown below.

```
1 SELECT *
2 FROM gui_users
3 WHERE age > 67;
```

Fragment 2.18. Result after propagation and constant folding

Here the constant folding results in the table 'gui_users', while the propagation places it in the `FROM` clause. The same holds for the `retirement` that first folds the `65` and `2` into `67`, and than propagates it into the query condition. The types of the variables `nameTable` and `retirement` now are determined to respectively `String` and `int`, because `retirement` is composed out of two integers and therefore remains an integer value.

2.4 PARSERS AND LEXERS

Since I dealt with a variety of programming languages, lexers and parsers are involved. Some structure, i.e. an internal representation, is constructed to perform the analysis on. And that is exactly what lexers and parser do, transforming the source text into a data structure.

To understand the capabilities and the features of the lexers and parsers at my disposal, this section elaborates on the lexers and tokenizers used within the SIG.

2.4.1 ASF+SDF

ASF+SDF[32] is a specification language for defining both the syntax (SDF) and the semantics (ASF) of a programming language. Besides this, it supports analysis and transformation of programs written in such programming languages. The steps taken are the following. First a parse table, containing enough information for both the lexical and the contextfree syntax, is generated to parse source code. Then the grammar optionally is extended in order to enable the rewrite rules to be applied. Finally a parse tree is constructed by the SGLR parser. The SGLR parser is a scannerless (no scanner is used to tokenise the input), generalized (finding all possible derivations for a certain string), LR parser. The SGLR parser interprets SDF parse tables, while taking characters as input and delivering as result parse trees or parse forests. The ASF engine then can apply rewrite rules as defined, mutating the tree.

The total design is shown in figure 2.1.

The SIG possesses several grammars supporting a variety of SQL dialects. These grammars then are used by JJForester, a parser generator, tree builder, and visitor generator for Java. It takes an SDF grammar as input and generates Java code for the representation of abstract syntax trees. After parsing a software program, a walk over that tree can be defined in Java using JJTraveler, a visitor combinator framework for Java. In order to simplify this, a visitor

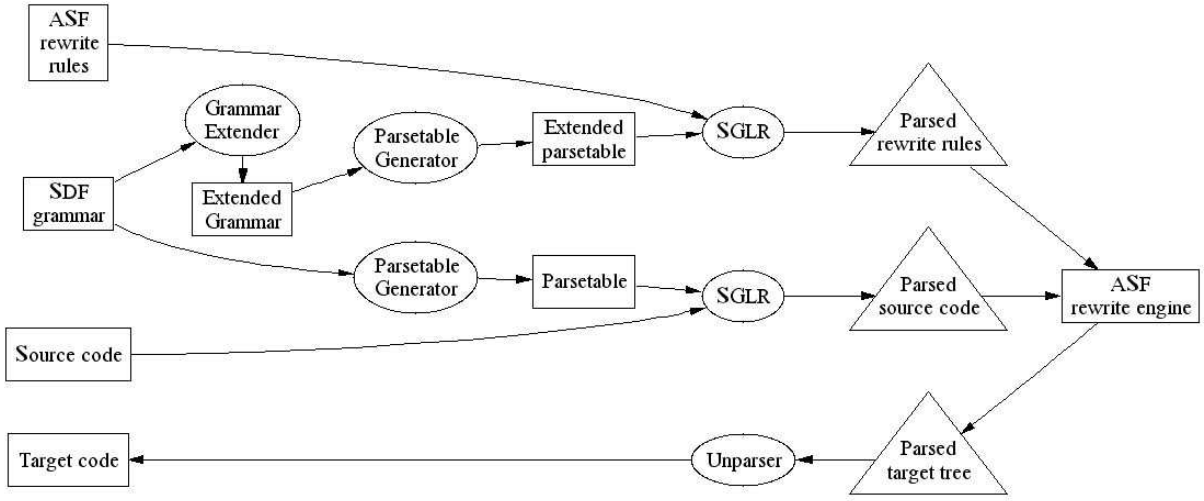


Figure 2.1. The architecture of ASF+SDF

and a treewalker are provided within the framework of the SIG (as explained in more detail in section 2.5), supporting the most common operations.

2.4.2 ANTLR

ANTLR[27], ANother Tool for Language Recognition, is a tool that accepts grammatical language descriptions and generates programs that recognize sentences in those languages. It allows one to augment the grammars with simple operators and actions to tell ANTLR how to build abstract syntax trees (AST).

ANTLR is an LL(k) parser and accepts three types of grammar specifications, i.e. parsers, lexers, and treewalkers. The aim of ANTLR is to generate recognisers that are humanreadable so that one is able to consult the output and in that way understand its behaviour.

The SIG maintains a variety of lexical definitions for ANTLR, in order to tokenise the following languages: PLI, Visual Basic 6, C++, Delphi, RPG, Gupta, Coolgen, Uniface, Powerhouse, Ingres, OPC Control Language, IDMS (Integrated Database Management System), CSide and Ideal. After tokenization a pattern matching technique on the token level is applied to extract information from sources in these languages. Furthermore, the SIG possesses full contextfree grammars for VB.NET, C# and Java.

After parsing a software program and constructing an abstract syntax tree, a walk over that tree can be defined and performed in Java.

2.4.3 Others

When looking only for particular information within a software application, it sometimes is simpler to bypass the traditional parsing and analysis techniques. In these cases other techniques are used. For instance in order to assess Tcl on the lines of comments present, regular expressions are used. The same goes for PowerHouse, where the get

statements in loops and procedures are counted. And even call graphs for COBOL code are constructed using regular expressions only.

2.5 Framework provided by the SIG

It is not surprising the SIG often has to assess systems containing SQL queries, for it is by far the most used way to communicate with a database system. In order to respond to the demands of the market, the SIG already has constructed a framework able to parse valid SQL queries. Its limitation however is that only complete syntactic valid SQL is allowed. It may be clear that retrieving simple fragments out of source code doesn't suffice here. Note that all analyses are performed in a static way, so no empirical learning, by executing the source and profiling the runtime behaviour, is applied. This thesis extends the basic query measurements on stored procedures by handling SQL queries embedded in a variety of host languages, and by defining metrics that also take the surrounding source into account.

The SIG is able to fully parse most common SQL dialects. For the different programming languages however, it would take too much effort to construct and maintain the different grammars needed, especially when you realize that the different clients of the SIG use a large variety of programming languages. For this reason only tokenizers are available to me when analysing the programming languages containing embedded queries. As a consequence, this restriction limits me in the level of detail when constructing flow graphs.

When analysing source code, the initial search for specific information always is coarse grained, ignoring all surrounding information. In order to anticipate on specific software projects to be assessed by the SIG, the constructed framework implements a model that allows the SIG to easily add patterns and constructs that should be taken into account by the analysis. It is not the objective to handle all possible cases, but to provide means to the SIG to easily handle any specific case encountered. So instead of defining a fixed pattern to look for, in for example SQL, the objective is to define a mechanism that takes the description of a pattern, applies it and returns the result. This principle for instance is applied in section 5.2.3, that enables the SIG to create custom patterns. Another application of this flexibility, is the mechanism that enables filtered queries out of some host language to be put in the regular SQL analysis, without adapting the latter one. The results of the SQL analysis than are treated as intermediate results for the specific host language analysis.

In order to assess software of customers in a short period of time, many useful algorithms have been implemented by the SIG. This section discusses metrics and algorithms applied by the SIG to obtain an indication of quality, using the techniques and tools discussed earlier on.

The SIG framework formed the foundation of this thesis project. By reusing available structures and operations, many of the tasks to perform already were provided. For this reason the capabilities and features within the framework are explored in this section.

2.5.1 Analyses present within the SIG framework

When assessing software systems, automatic inspection of the source code can give a good indication about its quality. Although monitoring the changes over time can say something about the quality, a diff showing only the changed pieces never is enough, so there

is more to it.

One indication of quality for imperative languages, such as Visual Basic for instance, is obtained by counting the occurrences of the `if`, `call` and `case` statements as well as the `function`, `array` and `variable` declarations present within the software system to assess. Also the number of unique and total operators and operands is computed (Halstead). Another measurement is counting the lines of comments present, locating duplicate code by clone detection and looking for unreachable code blocks.

A more extensive example is the McCabe complexity indication[23], that computes an index to express the complexity. The higher the index, the more complex the software is. The McCabe metric[1] takes a flow graph and computes the index by taking the number of edges/arcs minus the number of nodes plus two. Taking this one step further, the NPath metric signals the execution path complexity by measuring the number of acyclic execution paths. This means that it determines all execution paths, except for the possible iterations of a loop.

When assessing a software system, effort is made to detect the extremals in order to further investigate those spots. In many cases the top ten largest methods, queries or other properties are determined. Information like the percentage of methods with more then fifty lines, indicating an undesirable situation, then is derived. The longest method is determined by looking at the number of lines and by looking at the number of statements it contains. Other characteristics considered are methods that take the highest number of parameters, having the most unique or shadowed literals and containing the highest number of exit paths.

Of course, for the different languages, language specific analyses are present. Consider for example PL/SQL. When inspecting PL/SQL source code, a mapping between functions and the stored procedures they use, is constructed. And an even more PL/SQL specific measurement is the number of triggers present. Another example is the open socket analysis performed for C code.

Systems are often given a maintainability index (MI)[30]. This index is calculated by performing a number of metrics, and indicates the ability to be maintainable. For instance, the index of a Visual Basic application is calculated by taking the average of the 'lines of code', the McCabe index, the 'Halstead' and the percentage of the comments present.

The framework itself however is obligated to remain flexible and reusable, so effort is taken to ensure this. One case for instance required an analysis to determine the impact of changing the size of account numbers. In such situations you want to be able to reuse as much of the available algorithms as possible. In fact, a coarse granularity flow graph was constructed in order to determine the locations where those values were used in order to be able to change the format of the account numbers. When traversing a tree for instance, node visitors can be reused by strictly dividing and separating functionality into different and unrelated visitors. In this way existing visitors can be combined or extended to produce results that meet new requirements. The surrounding framework then is responsible for distributing the intermediate results, making the support for new languages or metrics a minimum effort.

[1] i.e. a system of related measures that facilitates the quantification of some particular characteristic[1]

2.5.2 Metrics applied to SQL by the SIG

The SIG is able to fully parse queries, that are complete and syntactically correct, from the most common SQL dialects. Queries constructed in a host language therefore currently are not taken into account, for they are neither complete nor syntactically correct. The analyses currently performed by the SIG, on isolated queries, are described below.

Surprisingly few papers are written about metrics and heuristics useful to SQL statements. In contrast, well-investigated areas are optimisations of the queries for the specifics of a certain DBMS. Other research is performed in the field of testing applications[10]. Then the focus lies on calculating the input ranges for the test cases. By looking at the queries embedded in the language, the range of possible inputs can be identified. By applying input data with as much variety as possible, the test coverage improves significantly.

For this reason the SIG defined their own standard sets of quality measurements as described below.

For isolated SQL queries a CRUD table is constructed. CRUD describes the basic operations of a database, and stands for Create, Read, Update and Delete. Examples of those operations respectively are, inserting new rows, selecting data out of the database, changing information and deleting data. In order to gather quality information, such a table is constructed for every stored procedure. So the four columns of the table stand for the four types of operation, while each row belongs to a different database table. This makes it easy to see which database tables are altered or used in what way per stored procedure. By looking at the number of database tables used in the queries, an estimation can be made about the complexity of a single query.

For the software systems to assess, both the longest query as well as the query containing the highest number of operators, i.e. $+$, $-$, $/$, and $\star$, are collected. Also a dependency graph is constructed in order to visualize how stored procedures invoke each other.

For each isolated stored procedure, from which the content normally is delivered to the SIG in a dump file, the following metrics are collected.

- Lines of code
- McCabe's cyclomatic complexity
- Number of queries
- Number of statements
- Number of parameters
- Number of constants
- Number of used tables

While each of the metrics above itself say little about the complexity as a whole, the combination is very meaningful[29].

In order to perform all the analyses described above, the following steps are taken. First all

comments within the queries are removed and the queries themselves are converted to upper case, since the Structured Query Language is case insensitive. Then the queries are parsed with an SDF grammar resulting in a AST (as described in section 2.4.1). Finally a visit over the abstract syntax tree is performed to collect the value defined by the metric.

The nesting of queries and the types the columns use, are not taken into account.

Recapitulating, over the years the SIG constructed a framework able to fully parse valid and complete SQL queries, assessing them as isolated SQL particles, without extracting them from a host language or taking any interaction and usage characteristics of the surrounding source code into consideration.

CHAPTER 3
DISTILLING QUERIES OUT OF HOST LANGUAGES

This chapter explains how queries embedded in host languages can be obtained. These collected queries then are used to identify the quality of applications with respect to the queries they contain, as will be explained in the rest of the chapters.

For embedded queries there generally are two types of constructions used within the different host languages. Queries build out of string concatenation and queries located in designated blocks. First I will discuss the string concatenation as this is used in Visual Basic 6, followed by the usage of blocks such as in COBOL.

For queries embedded within a host language, the extraction introduced in this chapter also gathers specific information available in the source code. For instance, constants referenced outside the query are collected. And in the cases where no value is to be found, at least an attempt is made to discover the type of the value dynamically inserted.

3.1 SQL QUERIES BUILT BY MEANS OF STRING CONCATENATION

When queries are constructed by concatenating strings, many language constructs should be taken into account. The issues to be addressed when gathering queries in these situations, is the great freedom of expression. The ways the queries are specified are limited only by the creativity of the creator. However some constructs are more common than others. Therefore a tradeoff is made in the framework to support the most common constructs by default, while adding specific structures should take minimum effort. In this way the framework can be adjusted for a specific situation.

The following paragraphs show each of the steps performed during query reconstruction. The result of the string reconstruction process then is shown after section 3.1.5 in the fragments 3.7 and 3.8.

3.1.1 Context

The SIG framework contains a tokenised scanner, using Antlr, for Visual Basic, making this the starting point for analysing this programming language. Because of the tokenizer, not being a parser, no fully abstract syntax trees are constructed. This enables a time efficient and robust analysis suitable for a variety of dialects, but imposes some complexity into the designed model. This is because languages requiring the reconstruction of queries by string concatenation[7], some degree of dataflow analysis has to be performed.

3.1.2 Starting points

First of all the locations are detected where the start of a query is defined. Starting points are located by inspecting the string literals. If a string literal begins with one of the valid SQL start words, like SELECT and so on, it is marked as possible start of a query. Inner queries now also are collected as being independent, thus resulting in invalid start points. This will be resolved later in the process.

All valid case insensitive start symbols are: ”SELECT”, ”INSERT”, ”UPDATE”, ”DELETE”, ”CREATE”, ”ALTER” and ”DROP”. Altering this list for specific cases, like extending it, is not a problem.

For each starting point a query container is constructed, having all intermediate and final results and warnings about the query. This query container therefore will be updated in each step performed within the process, dealing with that query.

Having all these points collected, they can be used for some form of program slicing[19]. While a backward slice says something about the used variables and their possible values, a forward slice collects information about the usage of the query result, as will be explained and illustrated in the following sections.

Now the start of a query is determined, a forward flow reveals the rest of the query parts present in the source. The forward flow enables us to abstract from specific database frameworks. For Java for instance, one could think of detecting the method invocation of `java.sql.Statement.executeQuery(String)`, that obtains a query result, and than performing a backward flow to collect the query. But than all different frameworks are to be supported. And in the case of a company that has created their own database layer or framework, without including it in the sources to assess, queries would not be detected. For that reason, that any appearance of a query is collected, a forward flow is used beginning from the start symbols.

3.1.3 Strings directly concatenated

This section focuses on strings that are combined with other strings. This stage detects if a string literal is continued on the same or on the next row, depending on where the row semantically ends. The merging of strings therefore will continue until a row is found not being part of the concatenation. To achieve this, only certain tokens are allowed between the strings. This holds for all different languages. Java for instance uses the + sign while Visual Basic has the `&` sign for this. Adjacent to those operators, strings and identifiers, including function calls, are allowed. The different languages have different means to specify when a row semantically ends. For Java this for instance is the semicolon, while for Visual Basic this can be defined with the regular expression [^_]\n, which means that a row semantically ends when a newline is encountered that is not preceded by an underscore.

If the query consists of several string literals or continues on the next row, we forward flow until no more strings with concatenation operators and identifiers in between occur. This is illustrated in the following example.

```
1 query  = "SELECT a "
2        + "FROM b "
```

```
3       + "WHERE id = " + nr;
```

Fragment 3.1. Traditional concatenation

The string starting with the `SELECT` keyword is identified as being the start of a query, as explained in the previous section. The different strings all have the concatenation operator in between, enabling the merge of them. The forward flow ends when the semicolon is encountered.

As we detect the parameter `nr` we insert the sign `?` and add a parameter object to the query container. The parameter object contains the name of the inserted variable. The type of `nr` at this stage is unknown and is set to 'null'. This poses the question what to do with anonymous parameters as shown below. The variable name related to the question mark defined within the query is unknown at this stage.

```
1 query  = "SELECT a "
2        + "FROM b "
3        + "WHERE id = " + nr + " AND user = ?";
```

Fragment 3.2. Anonymous parameter

The order of the parameters in the parameter object reflect the order of usage within the query. In this case we just add the parameter to the parameter object as having no name (null) at the correct position. When a variable is used multiple times, multiple references to the same parameter object are added to the query container maintaining the list of input parameters.

Parameters however are not limited to variable names. Function calls returning a result value also are allowed, in order to insert its result into the query. For the gathering of queries this is handled as follows.

```
1 strQuery = "UPDATE VPORDERS SET" & "NumberPorting=" & fncSQLString (
     strNumberPorting, nr)
```

Fragment 3.3. Function calls as parameters

The Visual Basic code shown above will be constructed to:
`UPDATE VPORDERS SET NumberPorting= ?` having as parameter '`fncSQLString (strNumberPorting, nr)`'. In this case the complete function call is seen as 'parameter' to be inserted into the query. At this stage variables and function calls are not distinguished. The type placed in the parameter object is simply a function type. For constructing this function type in the way function types are proposed[4] for Java SE 7, i.e. Dolphin, the type of the individual parameters have to be identified, as well as the return type of the invoked function. For Visual Basic this return type can be retrieved both by a return statement or an assignment to the function name. So a '`fncSQLString = 3`' somewhere in the body of the function indicates a numerical return type. If multiple types would have been encountered, the return type is unknown and gets assigned the `any` type, indicating that nothing specific is known about the return type. Finally the function type `int(String | int)` then is constructed. This should be read as the return type, followed by parenthesis and the types of the parameters separated by the '`|`' sign.

For every nonanonymous parameter a warning will be added to the query container for SQL

injection[3]. When inserting variable values direct into the query, quotation marks within the text could treat part of that text as SQL statements making the security aspect vulnerable. An illustration of this construct is shown below.

```
1 SELECT fieldlist
2 FROM table
3 WHERE field = '$EMAIL
    ';
```

Fragment 3.4. Host language

```
1 SELECT fieldlist
2 FROM table
3 WHERE field = '
    Alice@WTFU.edu';
```

Fragment 3.5. Resulting query

```
1 SELECT fieldlist
2 FROM table
3 WHERE field = '
    Trudy'
4 OR 'true'='true';
```

Fragment 3.6. Exposed vulnerability

This PHP example shows a variable directly inserted into the query. When providing a normal mail address, the result is a regular query. However, when providing the value `Trudy' OR 'true'='true` as input, note the missing opening and closing quote mark, all rows are returned because the criteria always evaluates to true. For this vulnerability reason, a warning is added to the query container for each variable directly inserted into a query. A similar warning is produced by the software product 'Microsoft FxCop' for inspecting .NET code. Instead of inserting values of variables directly into a query, it is better to provide those values to the framework handling the database access, while using question marks in the query to indicate the positions of the variables. Then the framework facilitating the database access takes care of escaping dangerous characters, just as it is designed to do. For this is only a brief explanation, one should be warned. Many vulnerability variants exist using the SQL injection principle. This is not simply solved by escaping the quote characters.

Looking for parts to concatenate is performed in a forward flow fashion. As a consequence inner queries are detected before that inner query itself is inspected. When a inner query is detected it is being removed from the list containing the starting points of the individual queries. For this reason inner queries are never inspected as an isolated query.

3.1.4 Locating the variable name the query is assigned to

For detecting the variable name the query or its result is assigned to, we look what is preceding the first line of the query string literal. If this is an assign token with a identifier standing just before it, this is regarded as the variable name the query itself is assigned to. In the case of a variable being assigned the return value of a function call, it could either contain a query or a query result. For
`result = Database.Get("SELECT a FROM b WHERE c")`
the variable `result` is detected.

In many cases the queries or their results are never actually assigned to a variable. For instance in: `If Not SQL.Execute( "UPDATE ... " )`. For this construct a warning is added to the query container. Binding results to variables separate concerns. Combining control flow decisions and the retrieval of information in most cases not is a good idea. An `UPDATE` for instance returns the number of affected rows. By not binding such results, error handling of this semantic form is never performed.

To ensure unrelated parts never are analysed needlessly, we never look back more then ten tokens when there is no evidence to be found of some assign construction.

3.1.5 Forward flow on the variables containing the query

Once the variable name the query is bound to is known, effort can be made to detect other concatenation constructs. This step detects the locations where strings are appended to the query. As shown below a variety of constructs is possible.

```
1 query  = "SELECT a "
2 query += "FROM b "
3 query  = query + "WHERE id = " + nr;
```

Fragment 3.7. Traditional concatenation

Most programming languages contain similar constructs for handling strings. In order to gather all query particles, a forward flow from the end of the first line of the query is performed. This ends when the variable containing the query is reassigned, when the scope ends or when the end of the file is reached.

Special cases, as the one shown below, create the need for a flexible analysis model. For Visual Basic this situation is handled as follows by the analysis performed.

```
1 strQuery = "SELECT x FROM y WHERE z "
2 strQuery = "SELECT a FROM b WHERE a IN ( " & strQuery & " ) "
```

Fragment 3.8. Concatenation is flexible

When the variable the query is assigned to is encountered, it is being replaced with the previously contained value.

When taking control flow into account, different paths may result in different queries. An `if` statement for instance can add different things to the query in its two branches. When the aim is to collect as much present information possible, all the different branches are to be taken into account. In most cases the path taken is unknown, creating the need to add the piece to the query container in order to not ignore it.

```
1 strQuery = "SELECT VPS.SERVICEID FROM VPS WHERE VPS.PCODE = 0 "
2 If strSubscription <> "" Then
3     strQuery = strQuery & "AND VPS.CLARIFYSUBSCRIPTION = '" &
          strSubscription & "' "
4 strQuery &= "AND VPS.PSTN = '" & blnPSTN & "' "
```

Fragment 3.9. Control flow issue

Taking the example above, the resulting query contains three lines. At least when we cannot be correct, we can try to be complete. Because different execution paths lead to different results, multiple variants of the same query can be constructed. In the constructed framework the possible different query parts simply are added to the related query container, enabling the construction of the different variants with minimum effort.

When it is detected that the traced variable name is not involved in concatenation, but in a new

assign activity, the trace stops. The obvious reason for this is that the query is overwritten by something else. In practise, the composition of queries by strings takes place within a single scope, simplifying the shadowing, because a variable then always references the same value store. And for the languages where strings are mutable, the aliases, i.e. multiple variables referencing the same string, are currently not taken into account by the framework, because these situations in practise rarely occur.

```
1 query  = "SELECT a "
2 query += "FROM b "
3 query  = "SELECT c"
```

Fragment 3.10. Reuse of variable names

In the example above, only the first two lines are collected. The third line will be treated as an individual query and will get its own query container.

But what if the concatenation is performed by invoking a function. An example of this is shown below.

```
1 query = "SELECT a FROM b "
2 query  = String.Concat(query, "AND id = -1");
```

Fragment 3.11. Concatenation through functions

This construct easily could lead to the misinterpretation of an assign without concatenation, thus creating a query of only the first line. For this reason the rule is introduced that when any function call is encountered, taking as parameter the variable name the query is assigned to, we should regard it as some form of concatenation. In practise this turns out to work very well.

Recapitulating the following now has been achieved.

```
1  'SELECT x FROM y WHERE z
2 strQuery  = "SELECT ";
3 strQuery  = strQuery & " x "
4 strQuery &= "FROM y "
5 strQuery  = String.Concat(strQuery, "WHERE z ");
6
7 strQuery = "SELECT a FROM b WHERE a IN ( " & strQuery & " ) "
```

Fragment 3.12. Concatenation is flexible

After performing the concatenation reconstruction analysis for the specifics of Visual Basic, the following query is derived.

```
1 SELECT a FROM b WHERE a IN
2 (
3   SELECT x FROM y WHERE z
4 )
```

Fragment 3.13. Query after reconstruction

However, the resulting query is not guaranteed to be a syntactically correct and complete query. By feeding every query to the SQL parser the queries that are not correct, are detected. Two things can be done then. The first option is to filter all malformed queries, or as a second option, requiring manual inspection. Because the parser points out the location in the

query not being valid, manual effort is limited. For this reason the constructed framework does not filter malformed queries after reconstruction.

3.1.6 Gathering parameter information

Queries often have places that are variable. For instance a username is inserted into a query. These variable parts, named holes, are filled at runtime, by making use of the variables present in the host language. In some occasions the variables are directly inserted when concatenating strings. In the other case the holes are marked by question marks in the query, while the variables are provided as parameters to the mechanism invoking the database.

After the concatenation is completed, we turn to the mapping between the holes and the names of the related variables. Two common scenarios exist. The first is the case where the variables are used during string concatenation, and secondly the case where the parameters are provided to the function executing the query. A combination of these is shown below.

```
1 SQL.Execute("SELECT WHERE a = " & varA & " AND b=? AND c=" & varC, varB )
```

Fragment 3.14. Query construction in Visual Basic

During string concatenation the query container is filled with three parameters, of which the names of the first and last are known, viz. `varA` and `varC`. The step of gathering information detects and binds the second question mark to the name of its related variable, i.e. `varB`.

As a next step, the reconstructed queries are investigated on the different variables they use. Each query itself then is adjusted to only contain question marks instead of variables, while the related variable names are added to the query container. This is visualized in the following example.

```
1  Dim nr As String
2
3  id = "-1"
4  id = 13
5  usrName = "admin"
6  nr = " ( " & id & " OR 99 ) "
7
8  query  = "SELECT a " _
9         & "FROM b " _
10        & "WHERE pid = " & nr & " AND user = ?"
11
12 Database.Get(query, usrName)
```

Fragment 3.15. A query using variables

The reconstructed query then initially is as follows.

```
1 SELECT a
2 FROM b
3 WHERE pid = ? AND user = ?
```

Fragment 3.16. Query after reconstruction

Now the two question marks relate to parameter containers, belonging to a certain query container, which will be used later on to add more information to.

It is not uncommon for companies to have an intermediate layer between database and business logic. In some occasions the company has constructed a framework to handle the requests to the database. Such frameworks typically are not supplied with the software to analyse by the SIG. As a consequence knowledge stored in such intermediate layers or frameworks cannot be derived. This imposes a problem for the retrieval of the used variables, as exposed in the example below.

```
1 Database.Get("SELECT a FROM ?", varName, Null)
```

Fragment 3.17. Unclear parameter construction

Or the same issue written in an other construct

```
1 strSQL = "SELECT a FROM ?"
2 Database.Get( strSQL, varName, Null)
```

Fragment 3.18. Unclear parameter construction

When creating frameworks, methods can expect extra parameters before or after the parameters related to the query. In the case of the example above, two parameters are provided while the query itself takes one. It is not possible to assume the `Null` to be excluded, because a boolean or something else could have preceded the variables used by the query. As a consequence non-matching parameters are ignored.

Now the variable names, related to and used by the queries, are collected, information is gathered about those variables. First all possible values the variable can be assigned to are located. Take the following example into regard.

```
1 typeId = "string"
2 typeId = 13
3 varname = "AND type = " & typeId
```

Fragment 3.19. Values contained by variables

In this case the `typeId` can have both the values '`string`' and '`13`'. Control flow in this case is not taken into account, and we also collect the values defined in dead code. The reason for this is that the programmer assigns the values for a reason, regardless of the exact location in the execution path.

The constructed framework however does not perform constant propagation, because this is highly language specific, while the inspected sources did not show query locations where in practise this would turn out beneficial for the result.

Once the possible values are collected, some information of the types can be derived. For instance by inspecting the characteristics of the value, it could be said that the value is a boolean, a number or some text. Having this in mind, with realising this is less precise than the actual type probably is, this information still is useful. When the values appear to always be a number, while the defined type of the variable within the programming language is a `String`, it can be seen as a bad construction. This information also allows the type to be compared with the type of the database column. Only when all three, i.e. the type of the value, the type of the variable and the type of the database column, are equal it should be

regarded as a good construct.

For detecting the type assigned to the variable, the following construct is matched and collected in the case of Visual Basic.

```
1 Dim varname As Long
```

Fragment 3.20. Types assigned to variables

While this differs for each programming language, all the typed languages have in common that it can be defined somewhere in a similar fashion.

When detecting values assigned to variables, there is always the possibility of exceptional situations. This is especially the case since the SIG does not have the means, ie. a grammar, for constructing a parse tree for Visual Basic code at its disposal. One typical exceptional construct I encountered is shown below.

```
1 If boolean And varname = otherVar Then ...
```

Fragment 3.21. Comparisons look like assigns

In Visual Basic a comparison has the same construct as an assign, with the only difference that the location of the expression is within the conditional part of a `if` or `while` statement. The second thought however is that when an equality check is done, the possibility exist that the left-handside (lhs) contains the value specified on the right-handside (rhs). Making it legal to assume that the right-handside is a possible value of the variable, i.e. the left-handside.

For fragment 3.15, the related parameter containers, after gathering all the additional information, contain the following information.

	First question mark	Second
Variable name	`nr`	`usrName`
Type	`String`	`unknown`
Values	`{" ( " & id & " OR 99 ) "}`	`{"admin"}`

The first parameter then is, recursively, further investigated by collecting the available information of `id`. Both the `"-1"` and `13` then are collected as values that possibly could be assigned to the `id` variable. In this way all information present about an query is collected. For all the different languages a variety of syntax is used, while all the languages preserve the same semantics. Out of each of the imperative languages, the same kind of information can be distilled.

Summarizing, the following reconstruction of queries on the source displayed below, is able to be performed at this stage.

```
1 public static final String LOGIN_STATUS = "ls1";
2 private int port;
3 private boolean isActive;
4
5 query  = "SELECT * FROM Stat ";
6 query += "WHERE status = '" + LOGIN_STATUS + "' AND ";
7 query += "port = " + port + " AND state = ?;";
```

```
8 db.execute(query, isActive);
```

Fragment 3.22. A query as how it could be defined within a programming language

The analysis on the Visual Basic source code then results in the following annotated structure:

```
1 SELECT * FROM Stat
2 WHERE status = 'ls1' AND
3 port = <int> AND state = <boolean>;
```

Fragment 3.23. A syntactically correct query with known types

The reason to obtain the type information is that any inconsistency can point out problems. For instance, when a query contains numerical information that is cited within a `String` in the host language, or the other way around, that an `Int` is defined as string in the database, a weak use of the data is indicated.

3.2 LANGUAGES USING FORMAL BLOCKS

This section discusses languages like COBOL that have the ability to contain queries as a language construct. For COBOL for instance the queries are situated in `EXEC SQL` blocks, although not all of those blocks are required to contain a query.

3.2.1 Locating the queries

The query parts within this type of language are itself not assigned or bound to a variable. Reusability then is achieved by invoking the procedures from different places in the source. For the PL/SQL language the SIG has the means to construct an AST (abstract syntax tree) using SDF. COBOL however is handled by the means of pattern matching using regular expressions, this in order to flexibly support the different variants of the COBOL language. For locating and collecting the queries within the trees, a Visitor is constructed collecting the nodes that indicate a query, just as the keywords defined in paragraph 3.1.2. The start node of the query than is taken as a subtree, containing the whole query. In order to support the development of analyses that use the obtained query AST, a pretty printer for PL/SQL parse trees was constructed. In practice it was used to visualize the resulting queries from transformed trees.

For the constructed abstract syntax trees, as is the case for PL/SQL, the collection of queries consists of locating the start node of the query. The rest of the complete query then is contained in the subtree.

For other languages, like COBOL, the regular expressions first filter all `EXEC SQL` blocks. Then all blocks not containing queries are filtered. Finally, the queries are unwrapped out of their blocks and where needed, semicolons are added in order to provide the PL/SQL parser valid SQL.

The first observations were that the analysis was remarkable slow. The reason for this was as follows. The regular expressions filtering all COBOL that was not SQL, erased the selection by replacing it with spaces. This is useful, for the line number information is being preserved for the SQL analysis. The drawback however was that the parser is scannerless, thus not efficiently eliminating the white spaces. So as a solution, one pass is performed removing all

superfluous spaces, while preserving newlines for the location information. This reduced the analysis time by 90 percent.

Even while this type of programming language natively supports queries in their grammar, the dynamic creation of queries still is allowed, and used. Pieces of strings are concatenated in order to be executed some point in time. One of the sources assessed by the SIG contained the following PL/SQL source.

```
1  EXECUTE IMMEDIATE
2  'CREATE TRIGGER trg_' || ps_table_name || '_insert ' ||
3  'BEFORE INSERT ON ' || ps_table_name || ' FOR EACH ROW  ' || CHR(10) ||
4  'DECLARE                                                ' || CHR(10) ||
5  '   ls_obj_name             mwxf.obj_name%TYPE;         ' || CHR(10) ||
6  'BEGIN                                                  ' || CHR(10) ||
7  '   IF :new.adm_object_id IS NULL                       ' || CHR(10) ||
8  '   THEN                                                ' || CHR(10) ||
9  '      :new.adm_object_id := mwxf_pk.get_new_object_id; ' || CHR(10) ||
10 '   END IF;                                             ' || CHR(10) ||
11 '   ls_obj_name := NULL;                                ' || CHR(10) ||
12 '   mwxf_pk.instantiate( ''' || ps_class_name || ''',
13                          ls_obj_name, :new.adm_object_id
14                       );                                ' || CHR(10) ||
15 'END;'
```

Fragment 3.24. Dynamic construction of queries in PL/SQL

This example illustrates dynamically constructed PL/SQL code, and thus is dynamically constructing a `CREATE` query. The resulting PL/SQL code actually is as follows.

```
1  CREATE TRIGGER trg_ps_table_name_insert
2  BEFORE INSERT ON ps_table_name FOR EACH ROW
3  DECLARE
4     ls_obj_name   mwxf.obj_name%TYPE;
5  BEGIN
6     -- skip the body
7  END;
```

Fragment 3.25. Resulting query

Because this construct in PL/SQL is regarded as bad practice, it is flagged as an undesirable situation by adding a warning to the query container.

As shown below, queries that are built from strings can behave rather curiously when interacting with the surrounding PL/SQL code.

```
1  declare
2   l_dept    pls_integer := 20;
3   l_nam     varchar2(20);
4   l_loc     varchar2(20);
5  begin
6   execute immediate
7     'select dname, loc from dept where deptno = :1'
8     into l_nam, l_loc
9     using l_dept ;
10 end;
```

Fragment 3.26. Obtaining query results from queries in strings

When examining the query in the string more closely, a colon followed by a number is encountered. Colons indicate a variable. However when followed by a number, it references the parameters passed at the USING keyword. The provided variables have a 1-based index. In this special construct the query is a combination of what is present in the string and the regular PL/SQL language constructs like the INTO keyword, which is outside of the string in order to enable interaction with the surrounding source code.

Because this construct is language specific, an instance of the string reconstruction analysis, like the one for Visual Basic, performing exhaustive analysis in order to reconstruct the strings and the surrounding expressions to a PL/SQL query, was not added to the framework. Therefore the queries built by strings are detected, while this occurrence itself already is seen as a bad construct.

3.2.2 Determining variables used within the query

PL/SQL distinguishes local variables from database tables and columns by the semantics of the host language. The host language contains declared variables, by the use of the DECLARE keyword or within the function signature, and therefore knows that it is a variable when it appears in the query. The analysis retrieving the variables used by the queries, first collects all variable names that are declared in the host language as a variable. When variables however have the same name as existing tables or columns a shadowing mechanism takes place. PL/SQL evaluates them in the prioritised order of column, variable and finally table. This means that a variable name will sooner reference a column than a declared variable. For this reason the analysis performed by the constructed framework takes all declared variables and when a database scheme is present, all possibilities to shadowing are detected, results in a warning added to the query containers that apply to this shadowing mechanism. However, when no scheme is present, declared variables are assumed not to be shadowed by columns in the tables, for PL/SQL.

Now that all declared variables present are collected, the analysis retrieving the variables matches these with the identifiers used within the query. As a result then, it is known which identifiers that are used within the query do reference a PL/SQL variable. This set than helps when performing data flow analysis to gather more information about the uses and relations among queries.

For COBOL this is far more straightforward, because the variables are prefixed with a colon. Since no abstract syntax tree for COBOL is constructed, regular expressions are used to extract the names of the referenced variables. This approach proved to be suitably flexible, for certain COBOL dialects allow white spaces between the colon and the variable name. A simple transformation, involving regular expressions, was performed to convert the source to basic COBOL.

As far as collecting the related result variables is concerned, only select statements have query results that are bound to some variable or cursor. The PL/SQL example below shows the

different occurrences of variables referencing the query result.

```
1  SELECT kd_std, kd_dk
2    INTO vrbld_std, vrbld_dk
3    FROM dkfr
4   WHERE kd_dk = k_prtvr.kd_dk;
5
6  FOR rec IN
7  (
8     SELECT object_name
9       FROM user_objects
10     WHERE object_type = 'PACKAGE' AND status = 'INVALID'
11 )
12 LOOP
13    BEGIN
14       RTUQ( rec.object_name );
15    END;
16 END LOOP;
17
18 PROCEDURE slct(do_ins OUT number) AS
19    OPEN do_ins FOR
20       SELECT x FROM y;
```

Fragment 3.27. Result variables in PL/SQL

In order to collect the variables, and identify them as result variable, the `INTO` node of the `SELECT` query, if present, is visited. For the example above this results in `vrbld_std` and `vrbld_dk`. Collecting cursors is done by visiting the directly surrounding statement of the query, i.e. the parent node, and determining whether it is a loop node that defines a cursor variable. In this way `rec` is found. Finally the formal parameters of the procedure can act as the return value, containing the query result, as is the case for `do_ins`.

CHAPTER 4
DETECTING RELATIONSHIPS AMONG DATA CONTAINED BY A DATABASE

When it is known what queries depend on other queries, the impact of changing a single query also is known, because the depending queries are at risk of being affected then. Further, the related queries also indicate a relation between the database tables and columns they use and address. The next chapter then elaborates on this indication and improves it, in order to be able to address the quality of the database scheme, if present.

In order to define strategies to detect relations among data in a database, research was performed about how this manifests in real-life applications. And since little to no documentation is to be found about relations established in host languages with respect to data contained by a database, research about this subject was compelled to be a empirical study, taking the retrospective element into regard. The approach taken involved observing the queries and query results contained by existing software projects, leading to an effective solution that functions well in practise.

This chapter handles the discovered dependencies and relations among queries and query results, because a well-formed and -used query is isolated in the usage of the input parameters as well as the use of the result it delivers. Results that are given as an input to other queries are highly likely to create undesired dependencies, as illustrated in the fragments 4.3 and 4.4.

4.1 Determining relations between queries themselves

The first step in finding relations in the host language code containing the queries, is to determine how the queries themselves relate to each other. This section explains how this knowledge can be retrieved.

4.1.1 Using CRUD tables for identifying suspicious relations between queries

A CRUD table, as explained in section 2.5.2, indicates which tables and columns are used by what query. Identifying relations can be done by analysing the CRUD table for suspicious patterns as will be explained in this section.

By the use of constructed CRUD tables, relations can be detected. The opposite however also is true, queries that definitely do not have a relation with each other, are eliminated from the set of related queries. When a read on the same table is performed by multiple queries, the queries get related because the query result is a subset out of the same table, and thus the same data set. Also the pressure on the various tables is measured this way, i.e. how often it is used compared to the other tables. The more a table, or more specific a column, is used and queried, the stronger the indication of an central key factor in the application.

The constructed framework creates a context sensitive CRUD table. Since the context, i.e. scope, changes with inner queries, the more traditional CRUD table is extended to contain context information. An example of a context changing inside a query is given below.

```
1 INSERT INTO a (b, c)
2 SELECT d, e
3 FROM f
4 WHERE g
```

Fragment 4.1. CRUD operations context sensitive

```
1 SELECT x
2 FROM f
3 WHERE z
```

Fragment 4.2. CRUD operation in one context

The constructed CRUD table for the framework contains data of multiple queries. Inner queries of the same, or another operation type, will not cause any conflict and are just added to the table, while referencing the query containing the subquery. In the CRUD table, queries themselves are for the output visually represented as numbers, because each query is assigned a distinctive number. The internal representation of a single CRUD entry however actually is a reference to the corresponding query container. The representation, like for instance the `toString` method of the CRUD object for fragment 4.1 and 4.2, then is visualized as follows.

	Create	Read	Update	Delete
A	$\{4\}$	$\emptyset$	$\emptyset$	$\emptyset$
F	$\emptyset$	$\{4, 5\}$	$\emptyset$	$\emptyset$

Here, the number `4` refers to the query of fragment 4.1 and `5` to the one of 4.2. The `INSERT` statement performs its operation on table `a`, while reading from table `f` to obtain the content. The independent second query also reads from table `f` and therefore is added to the read column of the CRUD table.

The different type of queries can be divided in the following categories.
Almost all dependencies among data or queries are realised by the use of `SELECT` queries. Mutating data contained by the database (`UPDATE`, `INSERT` and `DELETE`) never change the semantics of the database structure, and therefore is not capable in affecting the relations or dependencies present. Mutating table structures (`CREATE`, `ALTER` and `DROP`) however do not affect the artificial relations constructed by the semantics programmed in the host language. Both the queries mutating the data as well as the queries mutating the database structure, do not affect how the data in the database get related by the semantics of the surrounding application, because this only is expressed by the `SELECT` queries.

How CRUD definitions can help to detect relations is illustrated as follows. In order to detect foreign keys, take for instance the following two queries.

```
1 Set juveniles =
2       Database.Get( "SELECT id " _
3                   & "FROM Users " _
4                   & "WHERE age < 18; " )
5
6 commaSepList = toCommaSepList(juveniles);
7
8 Set simples =
```

```
9       Database.Get( "SELECT name " _
10                  & "FROM Courses, CourseUser " _
11                  & "WHERE Courses.id = CourseUser.course " _
12                  & "AND CourseUser.user IN (" & commaSepList & ");" )
```

Fragment 4.3. Incorrect use of foreign keys

This example assumes a school database containing three tables. The Users table contains the name and age of the students. The Course table contains the name of the courses given. And finally the CourseUser table realizes a many-to-many relation. The table CourseUser therefore contains two foreign keys. One pointing at the primary key of the Users table, the other pointing at the id of the Courses table. The aim is to get all the courses that are attended by at least one juvenile.

As can be seen, the host language created a dependency between `Users.id` and `CourseUser.user`. The reason that this dependency is undesirable can be detected by looking at the context of `commaSepList`, i.e. that it is related to `CourseUser.user` because of the semantics of the SQL query. `commaSepList` itself of course is related to columns defined in the `SELECT` and `FROM` part of the query.

The preferred situation is shown below, where the relation is used within the query itself.

```
1 SELECT DISTINCT Courses.name
2 FROM Courses, CourseUser, Users
3 WHERE CourseUser.course = Courses.id
4 AND CourseUser.user = Users.id
5 AND Users.age < 18;
```

Fragment 4.4. Correct use of foreign keys

Which is equivalent to the following query that uses the SQL `JOIN` construct.

```
1 SELECT DISTINCT Courses.name
2 FROM Courses JOIN (CourseUser JOIN Users ON CourseUser.user = Users.id)
3 ON Courses.id = CourseUser.course
4 AND Users.age < 18;
```

Fragment 4.5. Correct use of foreign keys by using the JOIN construct

The CRUD matrix in this case then is constructed as follows.

	Create	Read	Update	Delete
Users	$\emptyset$	$\{1, 3\}$	$\emptyset$	$\emptyset$
Courses	$\emptyset$	$\{2, 3\}$	$\emptyset$	$\emptyset$
CourseUser	$\emptyset$	$\{2, 3\}$	$\emptyset$	$\emptyset$

In this table the `1` refers to the first query in the example above, while `2` refers to the second query in the example above and `3` refers to the query in the finally presented solution.

4.1.2 Comparing queries for inequality and duplication

When the CRUD matrix suspects an undesirable relation between two queries, this can be further investigated by comparing the two queries with each other. By determining the

amount of equality this suspicion could be increased or decreased. If the two queries use some of the same tables but are totally unequal in every other aspect, it is highly unlikely that they do have a relation, provided that no other evidence is present about a possible relation.

Also when it can be identified that two queries are variants of a similar structure of each other, a union can be performed on the sets of dependencies, because what is known for one, is also very likely to hold for the other query.

In order to predict the relation between the different queries, and to determine to what extent the same data resources or properties are used, the outcome of a comparison between two queries is the percentage of equality. So the aim is to construct a table containing the equality for each combination of two different queries.

Determining equality means more then detecting code duplication. For this reason, the first step in comparing is propagating the defined aliases to obtain a query reflecting the database structure.

The propagation of aliases used within the query involves replacing alias identifiers by that of table and column identifiers. The abstract syntax tree is transformed and converted such that alias names are replaced by the original defined qualified name while the alias nodes and constructs themselves are removed from the tree.

```
1 SELECT *
2 FROM UserRights as rights, User u
3 WHERE rights.user = u.id;
```

Fragment 4.6. Query using aliases

This fairly straightforward transformation results in the following query, provided that the query above is given as input.

```
1 SELECT *
2 FROM UserRights, User
3 WHERE UserRights.user = User.id;
```

Fragment 4.7. Query with propagated aliases

With all alias nodes removed, the query reflects the database scheme, which simplifies the process of comparing different queries witch each other.

As a second step the `SELECT` queries are divided into five parts, knowing the `SELECT`, the `FROM`, the `WHERE`, the `GROUP BY` and the `HAVING` part. For the listings in the `SELECT` and `FROM` part, the comparison is just a matter of comparing the tables and columns used in both queries. The criteria specified in the `WHERE`, `GROUP BY` and `HAVING` parts first are split to single criterion sections. By dividing the different parts, comparison can be done with the order of the different criteria or the left and right hand sides within a criterion not influencing the result. The keywords `AND` and `OR` clearly delimiter the different individual criteria. Each criterion itself then is dividable in two parts. When encountering an = sign, the left hand side and right hand side can be taken separately for comparison. The `IN` keyword however

imposes a more flexible construct by providing means to be followed by either a list of values like `lhs IN (value1, value2)` or an inner query like `lhs IN (SELECT a FROM b)`.

As a first approach the different parts of the query, such as the `SELECT`, `FROM`, `WHERE` and so on, were prioritised with the idea that some parts are more likely to contain variables than others. For instance the columns selected in the `SELECT` part say little about the equality when structure and relations are involved. However, after performing some experiments it turned out that the result for this approach actually already was achieved by the construction of the CRUD tables. The final approach therefore was to give the different parts a weight according to their size. The size of a query part is measured by counting the number of left hand sides and right hand sides it contains. For the query `SELECT a FROM b, c WHERE d = e`, the `SELECT` part has a size of one, while the `FROM` and `WHERE` both have the size of two.

For obtaining the percentage of equality the different parts are compared. For instance when comparing `WHERE a = b AND c = d` with `WHERE a = b AND c = e` the left hand side of the AND of the first query part matches either the lhs or rhs of the other query part. The comparison of `c = d` and `c = e` results in a match of 50%. Since '`c = d`' is 50% of total amount of criteria specified in the WHERE, a 25% of inequality is registered for the WHERE part. The comparison is commutative, thus `a = b` compared to `b = a` is regarded as equal, focussing more on the semantic equality then the syntactic equality.

When comparing columns like `SELECT table1.column FROM table1` with `SELECT column FROM table1, table2`, without having knowledge about the database scheme, the column referenced in the `SELECT` part can only be inferred by inspecting the `FROM` part. The two columns are only equal if they both reference the same table. The first query proves that `table2.column` can not exist.

As explained in the previous section, for relations the focus lies on the `SELECT` queries. For this reason only `SELECT` queries are compared to each other, i.e. without taking the other query operation types into regard.

Another application of this technique is detecting queries having great similarities, i.e. consisting of large parts that are identical to that of other queries. Duplicated queries are undesirable for redundant definitions tend to be error prone when maintaining and changing those queries.

For example, I have encountered the following code in one of the production systems provided to the SIG.

```
1 IF (nowait_flag) THEN
2     IF df$old_rec.the_rowid is null THEN
3        SELECT       ID
4        ,            AANVRAAG_ID
5        -- this goes on the same for 55 more rows
6        ,            AD_WIJZIG_USR
7        INTO         df$tmp_rec.ID
```

```
8         ,            df$tmp_rec.AANVRAAG_ID
9         -- this goes on the same for 56 more rows
10        ,            df$tmp_rec.AD_WIJZIG_USR
11        FROM      MH_MT_HUIZ
12        WHERE             ID = df$old_rec.ID
13        FOR UPDATE NOWAIT;
14    ELSE
15        SELECT       ID
16        ,            AANVRAAG_ID
17        -- this goes on the same for 55 more rows
18        ,            AD_WIJZIG_USR
19        INTO         df$tmp_rec.ID
20        ,            df$tmp_rec.AANVRAAG_ID
21        -- this goes on the same for 56 more rows
22        ,            df$tmp_rec.AD_WIJZIG_USR
23        FROM      MH_MT_HUIZ
24        WHERE rowid = df$old_rec.the_rowid
25        FOR UPDATE NOWAIT;
26    END IF;
27 ELSE
28    IF df$old_rec.the_rowid is null THEN
29        SELECT       ID
30        ,            AANVRAAG_ID
31        -- this goes on the same for 55 more rows
32        ,            AD_WIJZIG_USR
33        INTO         df$tmp_rec.ID
34        ,            df$tmp_rec.AANVRAAG_ID
35        -- this goes on the same for 55 more rows
36        ,            df$tmp_rec.AD_WIJZIG_USR
37        FROM      MH_MT_HUIZ
38        WHERE             ID = df$old_rec.ID
39        FOR UPDATE;
40    ELSE
41        SELECT       ID
42        ,            AANVRAAG_ID
43        -- this goes on the same for 55 more rows
44        ,            AD_WIJZIG_USR
45        INTO         df$tmp_rec.ID
46        ,            df$tmp_rec.AANVRAAG_ID
47        -- this goes on the same for 55 more rows
48        ,            df$tmp_rec.AD_WIJZIG_USR
49        FROM      MH_MT_HUIZ
50        WHERE rowid = df$old_rec.the_rowid
51        FOR UPDATE;
52    END IF;
53 END IF;
```

Fragment 4.8. Four queries with only a minimal variety

This sample is 485 lines long with only two variables in it, making a total of four combinations. The outer `if` determines if the update is or is not `NOWAIT`. Then the only line differing in the two paths of the inner `if` statement is the `WHERE` line, which can exist of two possibilities. As can be seen, only a few percent of the total is not identical. In PL/SQL[13] the `condition` clause, used in the `WHERE` statement, is defined[9] as `expression '=' expression` and similar variants. So both the table and the value

looked for could be made variable, eliminating over hundred lines of redundant code. Two small `if` statements and one huge query also would do, eliminating 75% of the source and improving the software as a whole, by making it less tedious and easier to reason about.

With this type of code duplication, it is known that certain queries are just a different version of another query. Derived from that, its relation to that other query and the data sources also is known. This differs from the clone detection mechanism currently used within the SIG framework, because the in this thesis constructed analysis detects equality even when the order of lines or two expressions in a comparison are changed, while the clone detection just looks for sets of lines present in the program, that are equal to a sequence of lines somewhere else. This also means that context is not taken into regard, while the comparison constructed in this thesis is able to detect if certain parts within a query are equal, by isolating the parts to compare, identifying the places where a variable in combination with a preceding conditional would suffice.

4.1.3 The number of shared variables among queries

By measuring the number of variables that a query uses, but that is also used by other queries, a relation between the queries can be identified. If many queries point to a certain variable, that variable must contain information related to the tables and columns referenced in the different queries. Thus when a parameter has a value, which is used in several queries, the semantics are that something is in common regarding the queries using this information.

This very same indication of a relation however, is retrieved by inspecting the CRUD table, by comparing the equality of the different queries and by looking at the number of queries per procedure, as is performed in the case of PL/SQL.

4.1.4 Language independency due to reusability

The process of determining relations between the queries themselves highly focuses on the query as an entity and only very little on the surrounding host language. This enables the analysis to achieve language independency to some extent by reusing functionality for analysing queries across languages. In practise this proved to be very applicable as my experiences described below explain.

The reconstruction of queries in Visual Basic 6 is producing valid SQL as result. These queries than are being passed to the PL/SQL parser enabling the standard analyses used for PL/SQL to work on the constructed trees. Currently the CRUD table and comparison analysis are performed on the queries contained by Visual Basic code.
The other way around is also true. When a string, as discussed and described in section 3.2, is encountered in the PL/SQL abstract syntax tree, the check to determine if the string creates a query dynamically is performed by the same code as used for the Visual Basic 6 analysis. Transforming pieces of Visual Basic code in valid PL/SQL code and vice versa proves to be sufficient for some of the analyses performed.

The same holds for the analysis of COBOL sources. The extracted queries are fed to the

PL/SQL analyses and the results of those particular analyses are used when conducting the measurements, as discussed in chapter 7, for the COBOL source. Combined with the analyses performed specifically for the COBOL language all the required information is extracted. After all, languages do differ so the aim is to minimize effort for adding different language constructs. For the constructed framework, adding the full support for COBOL source only took two weeks.

4.2 Query results establishing relations

One way of retrieving an indication of relations between queries is by looking for certain patterns appearing in the data flow. For this part of the analysis the central focus is on the query result.

By determining relations between query results, i.e. by inspecting the usage of the query result in the host language with respect to the control flow, a relation between the data addressed by those queries also is likely to exist in some way. This section discusses patterns that indicate locations where queries get related by their results and the control flow present.

4.2.1 Query results used in other queries

When query results flow in other queries, the data obtained by the queries providing the input relate to the data obtained by the query consuming this information. Therefore in *all* cases this means a relation. As a consequence this construct means a structural inefficient approach, for the Structured Query Language is powerful enough to express the intentions.

A first basic example is the following query.

```
1 result = Database.Get("SELECT userId FROM User WHERE name = 'Alice'");
2
3 Database.Get("SELECT street FROM Address WHERE user_id = " & result.userId)
```

Fragment 4.9. Query using query result

Here, the query result of the first query is used to build the dependent query. Rather than this does imply a relation establishing a artificial foreign key by the host language, it in fact uses a normal foreign key, as could be defined by the database scheme. The actual case is an improperly use of the Structured Query Language, which results in a performance penalty. The optimised query then could be defined follows.

```
1 SELECT street
2 FROM User, Address
3 WHERE Address.user_id = User.userId
4 AND User.name = 'Alice';
```

Fragment 4.10. The preferred combined query

Legitimate reasons[16] for still applying this approach with separated queries are avoiding very large and complex queries by splitting them. The performance penalty then just is taken for granted.

A special case is when the related queries reference the same table. Take for instance the next two queries. The objective in this example is to retrieve the names of everybody as old as

Alice.

```
1 SELECT age
2 FROM User
3 WHERE name = 'Alice';
```

Fragment 4.11. Get the age of an user

```
1 SELECT name
2 FROM User
3 WHERE age = ?;
```

Fragment 4.12. Get everyone with a certain age

With the question mark indicating a parameter referencing the result of the first query. This relation indicates an inefficient usage, and therefore the queries can be combined to the following.

```
1 SELECT name
2 FROM User
3 WHERE age in ( SELECT age
4                FROM User
5                WHERE name = 'Alice' );
```

Fragment 4.13. Get all users as old as Alice

This enables the DBMS to resolve the requested result in an efficient way.

For completeness, it should be mentioned that combining multiple queries in order to improve performance, does not necessarily imply creating inner queries, as is shown in the example below.

```
1 SELECT name
2 FROM User
3 WHERE gid = ( SELECT id
4                FROM Groups
5                WHERE name = 'teacher'
                      );
```

Fragment 4.14. Get all teachers

```
1 SELECT User.name
2 FROM User
3 INNER JOIN Groups
4 ON User.gid = Groups.id
5 WHERE Groups.name = 'teacher';
```

Fragment 4.15. Avoiding inner queries

Both queries evaluate to the same result. Depending queries thus can in some occasions be combined with the use of join constructs. This however does not solve the problem of combining many queries without increasing the complexity of the resulting query.

4.2.2 Query results in conditional statements

An `IF` statement condition relying on the result of a query, and which is regulating the execution of another query, imposes a relation of the following form.

```
1 SELECT MH_MN_SEQ.nextval
2 INTO df$rec.ID
3 FROM DUAL;
4
5 IF df$rec.ID IS NULL THEN
6     BEGIN
7
8         SELECT MH_MT_HUIZ.JND_GOEDGEKEURD
9         INTO df$jnd.ID
```

```
10          FROM DUAL;
11
12          IF df$jnd.ID = FALSE THEN
13              df$jnd.ID := TRUE;
14          END IF;
15
16      END;
17  END IF;
```

Fragment 4.16. Suspicious data flow

First a `SELECT` query is executed and the result is stored in `df$rec.ID`. The `IF` condition then determines if another query is to be executed or not. That second query then also is followed by an `IF` statement, basing its condition on the latter query result.

Not only is this a bad practise, the locations of the queries with respect to the data flow of the query results indicate a relation. It is the query result that 'binds' the two queries throughout the control flow.

4.2.3 Query results used in loop conditionals

A pattern in order to detect opportunities for merging multiple queries, can be defined by inspecting inner loops, and has the following structure, in pseudo code.

```
1  loop query-result1 {
2          loop query-result2
3                  body
4  }
```

Fragment 4.17. Suspicious data flow

First of all, inner loops tend to be highly inefficient, creating an opportunity to optimise the performance dramatically here.

To illustrate this somewhat further, a real life example then looks as follows.

```
1  PROCEDURE set_tarieven
2  (
3     pn_tariefgroep_id   IN   td_tariefgroep.tariefgroep_id%TYPE,
4  )
5  AS
6     ln_tarief_id   td_tarief.tarief_id%TYPE;
7  BEGIN
8
9     FOR lr_tariefklasse IN
10    (
11       SELECT tkl.tariefklasse_id
12         FROM td_tariefklasse tkl,
13              td_tariefgroep tgr
14        WHERE tgr.tariefgroep_id = pn_tariefgroep_id
15          AND tkl.tariefschema_id = tgr.tariefschema_id
16    )
17    LOOP
18
19       FOR lr_afklasse IN
```

```
20       (
21          SELECT avk.afklasse_id
22            FROM td_afklasse avk,
23                 td_tariefgroep tgr
24           WHERE tgr.tariefgroep_id = pn_tariefgroep_id
25             AND avk.tariefschema_id = tgr.tariefschema_id
26       )
27       LOOP
28          ln_tarief_id := set_tarief
29          (
30             lr_tariefklasse.tariefklasse_id,
31             lr_afklasse.afklasse_id,
32          );
33
34       END LOOP;
35    END LOOP;
36 END;
```

Fragment 4.18. Suspicious data flow

All rows of the query result of the first query are being visited, while for each of these rows all rows of the query result of the second query are being visited. The body of the inner loop uses data from the results of both queries. Also since the value of parameter 'pn_tariefgroep_id' never changes, the inner query produces the same result for each run of the outer loop.

CHAPTER 5
QUALITY ANALYSIS ON THE HANDLING OF RELATED DATA

The aim of this chapter is to explain the nature of the detected relations in a general, language and implementation independent manner. This is performed by analysing dataflow and control flow, for they are able to point out locations that are more likely than other program points to contain or realize undesirable dependencies.

The difference between a relation or association and a dependency is that the first is just a link that is present, while a dependency has far more consequences, since a dependency points out that some parts are unable to go without other parts.

The constructed framework provides means to detect certain patterns. By enabling the set of patterns that is looked for, to be expanded with low effort, valuable information could be gathered, while reacting on the situations that are discovered to occur often, yielding an adaptable approach.

5.1 QUALITY ANALYSIS ON QUERIES DEALING WITH REFERENCES

Data becomes information when that data is associated with a small group of other data. These associations, or relations, are used to filter out the particles that contain information about the specifics one is searching for. A concrete example is having all the usernames and all the permissions present in a system. Both useless if it cannot be determined what permissions a specific person has. It is the relation here that makes it useful.

5.1.1 Foreign keys

This section discusses the constructs used to handle the relations defined by foreign keys. By observing the preferred situation, distinctions can be made with the situations that maintain relations without having them specified within the database scheme.

First the definition[8] of the term 'foreign key' is given:

- *A foreign key is a field or group of fields in a database record that point to a key field or group of fields forming a key of another database record in some table.*

Thus, a foreign key, also called a foreign keyword, in a database table is a key from another table that refers to (or targets) a specific key, usually the primary key, in the table being used.

As a characteristic, a foreign key implies that values in one table must also appear in another table. This leads to the conclusion that foreign keys just are present, or they are not, even when the database scheme is not aware of that. And in compliance with this, foreign keys also are present when certain data is copied, and thus duplicated, in for instance a report, logging

or statistics table. For instance user 'Alice' with id '1' can be placed in the logging table as the name 'Alice' with log row id '4', thus without having a reference in the traditional way when a immutable meaningless number is used for this purpose. This of course imposes problems when the user changes name, for the reference no longer is valid then.

In order to enable the relational database management system (RDBMS) to maintain the data integrity, the database scheme is suited to contain foreign key constraint information. By having these constraints specified, each `INSERT`, `UPDATE`, `DELETE`, `ALTER TABLE` and `DROP TABLE` instruction triggers the RDBMS to check if all foreign keys still point to some existing location.

It is needless to say that those benefits fall away when the host language itself maintains all foreign key administration without embedding this in the database scheme.

Our example used here is that of a User that belongs to a Group that has Permissions. The normal[22] and desirable query constructed to obtain the rights of a certain user is as follows.

```
1 SELECT rwx
2 FROM User, Groups, Permission
3 WHERE User.name = 'Alice'
4 AND User.group_id = Groups.id
5 AND Groups.perm_id = Permission.id;
```

Fragment 5.1. Suspicious data flow

In this example `User.group_id` and `Groups.perm_id` are foreign keys, pointing at respectively the `Groups.id` and `Permission.id`.

A naive approach however would be splitting the query by its foreign keys as shown by the following three queries.

```
1 SELECT group_id
2 FROM User
3 WHERE User.name = 'alice';
```

Fragment 5.2. Obtain the group the user belongs to

```
1 SELECT perm_id
2 FROM Groups
3 WHERE id = ?;
```

Fragment 5.3. Obtain the permissions the group has

```
1 SELECT rwx
2 FROM Permission
3 WHERE id = ?;
```

Fragment 5.4. Obtain the permission specifics

The question marks stand for the parameter values, which actually are the result of the previous query. By following the query results with respect to the other queries a relation between the queries is pointed out, as explained in section 4.2.1.

When using cross reference tables to facilitate many to many relations, the structure of the queries themselves remain the same. The only difference is additional tables and foreign keys.

Anomalies on this base case are not easy to detect. The next sections however will elaborate on constructs likely to be present when an own way of the foreign key mechanism was implemented.

5.1.2 Semantics of foreign keys within one table

When multiple columns have a relation with each other within a single table, in such way that data references each other, the columns should be relocated in different tables with the right references, i.e. with the usage of foreign keys.

Consider the following data contained by some table called `StudentsAndCourses`.

student name	course	teacher
Alice	mathematics	Trudy
Alice	mathematics	Eve
Alice	biology	Carol
Bob	mathematics	Trudy
Bob	english	Dave

From the content of this table, it gets clear that students can participate in several courses, and that the courses can have multiple teachers. This redundancy of data could be eliminated by redesigning the database structure.

The intended scheme, as designed according to the Boyce-Codd Normal Form[8], includes foreign keys as shown in the table definitions below.

Student
id
name

Student table

StudentCourse
stud_id
course_id

Student Course relation

Course
id
name

Course table

Teacher
name
course_id

Teacher table

Analysing the data contained by the tables within a database could indicate this situation with the missing appropriate foreign keys. Analysing database contents however is outside the scope of this thesis, while finding evidence for this type of misuse of the foreign key, is almost impossible to detect within the host language.

To illustrate this, lets go back a step and focus on the table preserving the foreign key functionality internally. Queries likely to be constructed then, in order to add, change or mutate the database contents, have a similar structure as the four basic examples shown below.

Gets the names of the courses the student participates.

```
1 SELECT DISTINCT course
2 FROM StudentsAndCourses
3 WHERE student = 'Alice';
```

Fragment 5.5. Get the courses a student participates

To delete a certain course of a student.

```
1 DELETE FROM StudentsAndCourses
2 WHERE student = 'Alice'
3 AND course = 'mathematics';
```

Fragment 5.6. Remove a course the student participates

To delete all courses from a certain teacher.

```
1 DELETE FROM StudentsAndCourses
2 WHERE teacher = 'Trudy';
```

Fragment 5.7. Removes a teacher and its courses

In order to let a teacher take over all the courses of some type.

```
1 UPDATE StudentsAndCourses
2 SET teacher = 'Eve'
3 WHERE course = 'mathematics'
```

Fragment 5.8. Reassign a course its teacher

Observing the structure of the different queries, there is no indication of a misuse, i.e. the absence of a foreign key. There is no fundamental distinction here with the queries used in the case of a present foreign key.

The only evidence that possibly could be found within the host language is where content is added to the database. A loop, with perhaps an inner loop, creating a query, with some parameters not changing, while others do change, could indicate this type of 'missing foreign key' construct.

```
1 INSERT INTO StudentsAndCourses (student, course, teacher)
2 VALUES ('Alice', 'mathematics', 'Trudy');
```

Fragment 5.9. Add a course the student participates

When comparing this query with the way it is done in a database that has a scheme containing foreign keys, the following is being observed. Adding content is highly unlikely to be performed within an inner loop. For the students and courses example, adding a course to a student is not likely to use a inner loop.

```
1 INSERT INTO StudentCourse (stud_id, course_id)
2 VALUES (sid, cid);
```

Fragment 5.10. Adding foreign key values

The only loop to be expected here is when several courses are added for the student in one go. In that case the loop preserves one parameter, the student id, and varies the course id. Unfortunately, this is the same pattern occurring in the table without the foreign keys. So even this approach cannot point out the misuses of foreign keys within one table. This aspect of the quality of the data model can not be derived from the queries.

5.1.3 Locating the usage of joins

Following the previous sections, the investigated way to detect joins, is by inspecting the structure of the queries. Note that the analysis is not allowed to assume that a database scheme is present, so the generic approach taken is presented here, assuming explicit table and column information.

First the `WHERE` part of the `SELECT` query focussed on is collected. Then all comparisons that are combined with the `AND` or `OR` keywords are filtered as shown in the example below.

```
1 SELECT Groups.name
2 FROM User, Groups
3 WHERE User.uid = Groups.user
4 AND User.name = ?
```

Fragment 5.11. Manually specified JOIN

```
1 SELECT a.b
2 FROM c, a
3 WHERE c.d = a.e
4 AND c.f = g
```

Fragment 5.12. Structure of a JOIN

In this example all comparisons, i.e. a '=' token preceded and followed by a value or reference, are bound with the `AND` or `OR` criterion. Then it can be spotted that the table reference `c`, of the structure example 5.12, is referenced in multiple comparison expressions. Finally different columns of `c` are used, allowing the specified JOIN to be indicated.

This mechanism also holds for multiple joins, as will be explained in the examples below.

```
1 SELECT Permission.rwx
2 FROM Groups, Permission, User
3 WHERE Permission.gid = Groups.id
4 AND User.uid = Groups.user
5 AND User.name = ?
```

Fragment 5.13. Multiple manually specified joins

```
1 SELECT a.j
2 FROM c, a, e
3 WHERE a.b = c.d
4 AND e.f = c.g
5 AND e.h = i
```

Fragment 5.14. Structure of the joins

In this case the table references `c` and `e` are referenced in multiple comparison expressions. Also different columns of each filtered table are used, allowing the indication of the specified joins. So `e.f` and `e.h` as well as `c.d` and `c.g` hold for the specified pattern, identifying two coherent joins in total.

In the previous two examples the assumption was made that the column reference also included the table information. However, when this information not is present, it can be derived if the database scheme is present. The column can only belong to one table specified in the `FROM` part. Thus this approach is limited to only those systems where database schemes are present.

Finally the aim is to not rely on the foreign keys specified in the database scheme, for the objective is to detect JOINS also when not structurally specified. Then it can be made clear that the pattern specified before does not hold for all situations.

```
1 SELECT User.id
2 FROM User, Type
3 WHERE User.street = ?
4 AND User.housenr = ?
5 AND User.type = Type.name
6 AND Type.categorie = ?
```

Fragment 5.15. Misidentified JOIN

Now both the `User-Type` as well as the `User-?` relations are matched as a join, while this only is valid for the `User-Type` in the third line.

This problem can be solved by taking the selected columns, i.e. `SELECT User.id`, into

regard. The mismatch in example 5.15 is corrected as follows. The tables of the columns selected can be seen as the end of a chain of tables, that are linked through joins. This is clarified by the three tables below.

$User$
$User \leftrightarrow ?^1$
$User \leftrightarrow ?^2$
$User \leftrightarrow Type$
$Type \leftrightarrow ?^3$

$?^1 \rightarrow User$
$?^2 \rightarrow User$
$Type \rightarrow User$
$?^3 \rightarrow Type$

$?^3 \rightarrow Type \rightarrow User$

The left table shows at the top the database table used in the selected items. The rest of the table shows a schematic representation of the different comparisons, linking database tables to each other. The different question marks refer to the different input parameters. Now the middle table shows the reconstruction of the chain, for it is known what the end of the chain can be. When one identifier points at the end of the chain, itself is being inspected for references to it. Thus '$?^1$' has no references to it any further, while for '`Type`' other references do exist, enlarging the chain.

Finally the right table shows the fully constructed chain, linking the different tables to each other by the use of joins. This longest chain contains two arrows, while with N arrows, $N - 1$ joins exist. Thus correctly identifying one join for example 5.15. And as displayed below, this technique also holds for example 5.14.

a
$a \leftrightarrow c$
$e \leftrightarrow c$
$e \leftrightarrow i$

$c \rightarrow a$
$e \rightarrow c$
$i \rightarrow e$

$i \rightarrow e \rightarrow c \rightarrow a$

This chain does correctly identify the two defined joins.

This approach does not count `A.x = A.x` as a join, and counts `A.x = B.y AND A.x = B.y` as just one join. Only joins that are actually used to produce the result are counted, as explained in the following example.

```
1 SELECT Permission.rwx
2 FROM Permission, User, Groups
3 WHERE Permission.type = '-1'
4 OR User.uid = Groups.user
5 AND User.loginname = ?;
```

Fragment 5.16. Joins not related to the query result

Now a join is present between `User` and `Group`, but this is not with regard to the result, which only focuses on `Permission`. The trace from the select items thus excludes the unused joins.

This technique also can be used to identify unused tables specified in the `FROM` clause. In order to do so, information must be present about the tables the columns belong to. Then the information specified in the select items, the `FROM` clause and the conditions in the `WHERE` clause, can be linked by the techniques discussed earlier in this section.

```
1 SELECT A.x, B.y
2 FROM A, B
3 WHERE A.id = -1;
```

Fragment 5.17. Table used by the select items

```
1 SELECT A.x
2 FROM A, B
3 WHERE A.id = -1;
```

Fragment 5.18. Unused table in the FROM clause

Although the table `B` in the left example is not used in the `WHERE` clause, the specified `B` is used in the select items. However, when a specified table is neither used in the select items nor in the `WHERE` clause, an unused table is identified as illustrated by the right example.

5.1.4 Semantics of the SQL keyword UNION

A union takes the result of the lhs query and the rhs query, adds them, and performs a `DISTINCT` on it. This differs from the '`UNION ALL`' in that way, that the latter does not perform a distinct. Multiple rows with the same results then are allowed.

A `UNION` means that data from the left hand side query result have some same semantics as the data delivered by the right hand side query. This for instance is illustrated in the following example.

```
1  -- Select all events in the future
2  SELECT scheduled_event
3  FROM User
4  WHERE name = 'Alice'
5
6  UNION
7
8  -- Select all events in the past
9  SELECT event
10 FROM Log
11 WHERE user = 'Alice'
```

Fragment 5.19. UNION selecting data with equivalent semantics

The column `Log.user` can be seen as a artificial foreign key to the 'primary key' `User.name`. As can be seen, the usage of `UNION` instead of `UNION ALL` do not change the semantics. With respect to the relation, i.e. reference, the query semantics stay the same, for only the query result changes. Another aspect is that the `UNION` is not bound to only two queries. Any number of queries, with a minimum of two, can be used in order to combine the results of them. This than means that all of the involved queries, bounded by the `UNION`, share the same semantics with respect to the selected columns producing the query results.

When focussing on the aspect of performance, [16] states that when the several queries use the same `SELECT` and `FROM` definition, the `UNION` construct could be replaced by an OR criterion added to one of both queries, benefiting the performance.

5.2 QUALITY ANALYSIS BY INSPECTING THE USAGE OF QUERY RESULTS

By inspecting the dataflow of the result variables, that temporarily store the obtained data, an indication is obtained of what role the query plays within the host language. This sections explains what semantics the different patterns have.

5.2.1 Semantics of a relation between SELECT and INSERT

As explained before, the `INSERT` queries can provide information when collecting evidence of data referencing each other within a database. The ending of section 5.1.2 indicates that an `INSERT` query can be preceded by multiple `SELECT` queries, retrieving information that establish relations among the data added. And by observing the query results, a relation can be detected between the `SELECT` and `INSERT` queries, as is explained in section 4.2.1. These relations however do not necessarily mean an undesirable dependency, but while this situation not necessarily indicates a misuse of foreign keys, it does indicate an inefficient SQL construct.

Focussing again on the example showed in section 5.1.2, an optimisation could be suggested as imposed by section 4.2.1. The separate `SELECT` and `INSERT` queries can be combined in a more efficient way, i.e. benefiting the performance, as shown below.

```
1 INSERT INTO StudentCourse (stud_id, course_id)
2 SELECT Student.id, Course.id
3 FROM Student, Course
4 WHERE Student.name = 'Alice'
5 AND Course.name = 'mathematics';
```

Fragment 5.20. Combination of INSERT and SELECT

The benefit is gained here because the DBMS is optimised to solve the combined request efficiently. By splitting the requests accommodating the actual intention, the DBMS is bypassed to perform optimisations of this kind.

For the case of fragment 5.9 the values are likely to be propagated to the query through three variables. Rather than these are query results, it is more likely that the values are read out of hidden form fields or other GUI constructs.

This observation leads to the following pattern. When having more than one `SELECT` query directly followed by an `INSERT` query, while using the results of the `SELECT` query within that `INSERT` query, a desired dependency and an inefficient usage of the information needed to establish the relation, is indicated. For this inefficient use, the framework adds a warning to the involved query containers.

For isolated `INSERT` queries, i.e. `INSERT` queries not making use of other queries but that are using parameters, the intention with respect to relations and dependencies is undecided.

5.2.2 Manually performed CASCADE DELETE

A common scenario with respect to foreign keys, is that when the item referenced is removed, the related information also should be removed. For example, when a `User` has a `Car` and the user is removed out of the system, the referenced car also should be removed. This is solved by the construct of a `CASCADE DELETE`. When a `CREATE TABLE` or `ALTER TABLE` defines a column as being a foreign key by using the keyword `REFERENCES`, and thus referencing some primary key, the `ON DELETE CASCADE` option can be set. This option ensures that the rows referencing some primary key are removed when that primary key itself is removed.

For our analysis, this means that `DELETE` queries themselves do not contain information about if the delete command is cascaded or not. However, if the database scheme itself is present, these deletes related to the cascade behaviour are in any case being flagged as desirable situations.

Now placing this in context, when in the scheme no foreign key is specified, patterns can be used signalling a cascade delete that is performed manually. The patterns used to indicate this undesirable situation are explained below.

The base case is a User that can have multiple Cars while a Car can only belong to one User. When the User is being removed, the Car also is.

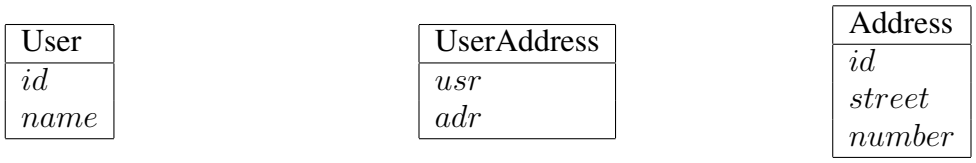
```
1  -- Retrieve the user to remove
2  SELECT id
3  INTO usr_id
4  FROM User
5  WHERE name = 'Alice';
6
7  -- Remove the user
8  DELETE FROM User
9  WHERE name = 'Alice';
10
11 -- Remove the cars the user owns
12 DELETE FROM Car
13 WHERE usr = usr_id
```

Fragment 5.21. Delete a referenced item

As can be seen, only three points can vary here. The first `DELETE` query can reference either the id already retrieved, or state the same criterion as the `SELECT` query does. The second optional variation is the number of `DELETE` queries following the `SELECT` query. Other items referencing the User, in the same way the Car does, can be removed in the same way. The last point is the order of the `DELETE` queries. They can go in any order, as long as the `SELECT` precedes them.

In order to determine what queries do belong to the chain of queries matching the pattern, scoping should be taken into account. By inspecting the different scopes, the logical sequence of the queries is determined. For PL/SQL for instance the scope changes when a `LOOP` statement, `IF` statement or block, i.e. the `BEGIN` till `END` statement, is encountered. This also holds for functions and procedures, since they start with a block statement and thus automatically change scope with the rules stated above.

In order to illustrate that this principle holds for all situations, the following more extensive example describes a User that can have multiple Addresses, while an Address can belong to multiple Users. In a database scheme this many to many relation is solved with a cross-reference table. First the database scheme is presented to clarify the structure used.



The scheme contains a user, that consists of a name and an address, and the address consists of a street name and a house number. The `usr` and `adr` in the cross reference table `UserAddress` respectively reference the id of the user and the id of the address.

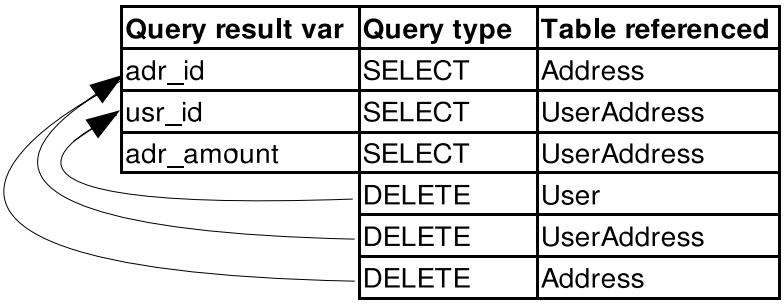
Now the queries needed to achieve the following problem are given. When an Address is being deleted, also all related Users should be deleted out of the system, but only if they no longer have any Address left. This can be done in the following way.

```
1  -- Find address to delete
2  SELECT id
3  INTO adr_id
4  FROM Address
5  WHERE street = 'Great Ocen Road'
6  AND nr = 13;
7
8  -- Retrieve all users living at that address
9  FOR usr_id IN
10 (
11    SELECT usr
12    FROM UserAddress
13    WHERE adr = adr_id
14 )
15 LOOP
16    -- Get the nr of addresses the user has
17    SELECT COUNT(*)
18    INTO adr_amount
19    FROM UserAddress
20    WHERE usr = usr_id;
21
22    -- Remove users only if the user
23    -- no longer has any address
24    IF adr_amount < 2 THEN
25       DELETE FROM User
26       WHERE id = usr_id
27    END IF;
28 END LOOP;
29
30 -- Remove foreign keys
31 DELETE FROM UserAddress
32 WHERE adr = adr_id;
33
34 -- Remove address
35 DELETE FROM Address
36 WHERE id = adr_id;
```

Fragment 5.22. Delete a N to N relation

First the id of the address to be removed is obtained. Then for each user living on that address the number of addresses it has is requested. If the user only has that address, that is going to be removed, the user itself is removed. Now all references to the address to be removed are removed. Finally the address itself is removed.

Recapitulating, the following pattern is observed in the example above.



Query result var	Query type	Table referenced
adr_id	SELECT	Address
usr_id	SELECT	UserAddress
adr_amount	SELECT	UserAddress
	DELETE	User
	DELETE	UserAddress
	DELETE	Address

Figure 5.1. The pattern in terms of relations

This is the schematic representation of the semantics of the PL/SQL code from fragment 5.22. First three `SELECT` queries are encountered placing their results in the variables `adr_id`, `usr_id` and `adr_amount`. Then the three `DELETE` queries use the first two parameters. The third parameter only is used for control flow. All queries reference only one table.

Notice that the mechanism as described in section 4.2.2 detects the `adr_amount` in the conditional expression, for it references the query result from the `SELECT` query preceding the `IF` statement.

Generalizing these observations, the following pattern can be stated as an undesirable manual cascade delete operation that should have been performed by the DBMS itself. The definition than is as follows:
A delete query using data from the result of a select query, followed or preceded by one or more other delete queries (that not necessarily use data from a select query), implies an operation performed by the host language fulfilling the semantics of a cascade delete.

5.2.3 Generalizing the definitions of the patterns

Generalizing the idea that the order and operation type of queries define a pattern, as is the case for the previous few sections, the following pattern description can be defined.

$$n \in \textbf{Num} \quad \text{numerals}$$

$$
\begin{array}{rcl}
cardinality & ::= & n \mid ? \mid + \mid * \\
queryOperation & ::= & \texttt{SELECT}\ cardinality \\
 & \mid & \texttt{INSERT}\ cardinality \\
 & \mid & \texttt{UPDATE}\ cardinality \\
 & \mid & \texttt{DELETE}\ cardinality \\
 & \mid & \texttt{CREATE}\ cardinality \\
 & \mid & \texttt{ALTER}\ cardinality \\
 & \mid & \texttt{DROP}\ cardinality \\
pattern & ::= & queryOperation+
\end{array}
$$

The abstract syntax above describes how patterns are composed.

A chain of related queries always start with one `SELECT` query, which gathers information out of the database. The chain following this first `SELECT` than can be specified as a pattern. The pattern exists of one or more query operations, while a query operation is of the type `SELECT` or `INSERT` etc. Each query operation has a cardinality of how often it may occur in the chain. This cardinality can either be a fixed number, but also the 'zero or one time', the 'one or more times' and the 'zero or more times' amount.

A pattern always matches a minimum of two queries, i.e. the preceding `SELECT` query and the query or queries specified in the pattern. It is the pattern that specifies the order and the quantities of the queries, that follow the `SELECT` query.

$$
\begin{array}{rcll}
Q & \in & \textbf{Query} & \text{queries} \\
qRes & \in & \textbf{QueryResult} & \text{variables containing query result}
\end{array}
$$

Here, the set of all queries is placed in Q, while the variables used by the preceding `SELECT` query to place its query results in, is located in $qRes$.

$$
var(Q) = \left\{ \substack{var\text{: } \textbf{Query} \rightarrow \mathcal{P}(\text{vars}) \\ v \mid v \text{ are the variables related to the holes in Q}} \right\}
$$

This function collects all the variables used by the query. The variables thus transfer information from the host language to the DBMS by means of the query. These referenced variables typically occur in the criterion specification of the query.

Now the following rule is specified excluding certain queries from the pattern.
$\exists queryOperation \in pattern : var(queryOperation) \cap qRes \neq \emptyset$
In the chain there must exist at least one query operation for which the following holds. The query references at least one of the result variables of the preceding `SELECT` query.

Another variant of this rule is as follows, where it is required that all queries in the chain reference at least one result variable of the preceding `SELECT` query.
$\forall queryOperation \in pattern : var(queryOperation) \cap qRes \neq \emptyset$

The construct described in paragraph 5.2.1, where `INSERT` queries relate to `SELECT` queries, would be detected by the pattern `(SELECT *) (INSERT 1)`. Multiple selects, but at least one, i.e. the one preceding the pattern, should be used in the `INSERT` query in order to match the pattern. This pattern then is combined with the rule that at least one query in the chain does use some of the result variables of the first `SELECT` query. This is because the rule that all queries should use some of the query results of the first query, is too restrictive and would not detect the pattern when some of the `SELECT` queries in between do not use a query result.

For paragraph 5.2.2, describing the manual cascade delete, the following pattern is defined: `(DELETE 2) (DELETE *)`. Also this time the rule is applied that at least one query in the chain should use the result variable of the preceding `SELECT` query. The result than is a set of query chains that identify the locations where a manual cascade deletes occurs, and which queries are involved in that operation.

The constructed framework takes two inputs for this analysis, viz. the query containers and a definition of the pattern. In order to abstract from the specific programming language, the approach taken does not inspect the AST, tokens or raw source. So instead of using visitors or regular expressions, the query containers specify all the information needed. The query container object therefore contains the type of the query, references query containers it used query results from, and contains the scope, i.e. the context, of the query, as more thoroughly explained in section 6.2.2. By inspecting the logical sequence of the queries within a scope, the order of execution is determined, while when the scope ends the remaining queries are unrelated. In this way independent query sequences are defined. For instance, when an `INSERT` must be followed by a `DELETE` according to some pattern, the latter one is not allowed to occur in a context unrelated to the context the `INSERT` is in.

When consuming the queries in a greedy fashion, problems in the pattern could occur. For instance, a pattern defined as `(DELETE *)` followed by a `(DELETE 1)` would always result in an empty set, because all delete queries would have been consumed by the time `(DELETE 1)` in the pattern is reached.
In order to prevent this, the pattern is normalized before its use. The problem described above only exists when adjacent pattern items are of the same query operation type. So the solution to avoid the problem described above is to compare the adjacent cardinalities and change them when needed.

First, the + cardinality is considered as syntactic sugar. So all occurrences of the + cardinality are rewritten to two query operations that have the cardinality `1` and `*`. Thus `(DELETE +)` is converted to `(DELETE 1) (DELETE *)`. Now the problem is reduced to order the query operations by their cardinality. The `n` cardinalities are placed in the front, followed by the `?` cardinalities, leaving the zero or more cardinalities at the end of the chain. Finally, the redundant query operations are eliminated. All the fixed number cardinalities are added and joined into one query operation, while the multiple + and `*` query operations are reduced to only one single appearance.

To illustrate these rules, take the following pattern example:
`(DELETE +) (DELETE *) (DELETE ?) (DELETE 3).`

The first step of desugaring results in:
`(DELETE 1) (DELETE *) (DELETE *) (DELETE ?) (DELETE 3)`
By applying the order we get:
`(DELETE 1) (DELETE 3) (DELETE ?) (DELETE *) (DELETE *)`
Finally, eliminating the redundant query operations results in:
`(DELETE 4) (DELETE ?) (DELETE *)`

In this example the `?` cardinality may look unnecessary, but in the case of `(DELETE 1) (SELECT ?) (DELETE *)` the zero or one cardinality expands the capabilities of the pattern match, by allowing both the presence as well as the absence of the `SELECT` query between the `DELETE` queries.

So the given pattern now is able to match on the queries present, by inspecting their type and sequence. The result then are sequences of query containers that match the defined pattern. These chains then however still can be invalidated when none or not all of the matched queries use the query result of the preceding `SELECT` query, depending on the rule specified in the pattern. By filtering the invalidated chains, the final result consists of all query sequences matching the given pattern.

When inspecting the occurrences of queries in the host language, the same query could be collected by different patterns. For only one pattern at a time is applied, no ambiguity can exist, while the pattern can have several matches per scope present. Take for instance the query presented below.

```
1 SELECT a INTO result FROM b WHERE c = d;
2 DELETE FROM b WHERE e = result;
3 SELECT x INTO result FROM y WHERE u = v;
4 INSERT INTO z VALUES (result)
```

Fragment 5.23. Sequence of queries

Now when the two patterns `(DELETE +) (SELECT *)` and `(INSERT +)` are applied for matching, the first pattern collects the `SELECT`, `DELETE` and `SELECT` queries, while the second pattern collects the last `SELECT` and the `INSERT` query. Remember that patterns specified, are always implicitly preceded by a `SELECT` query in order to begin the chain of related queries.

The nesting of patterns, like for instance the definition
`( SELECT 1 ((INSERT 2) (SELECT ?))* )`
is not added to the constructed framework. This is because the non-nested pattern
`(SELECT 1) (INSERT 2) (SELECT ?)`
actually already does collect *all* occurrences of such a (minimal) query chain.

CHAPTER 6
CONSTRUCTED FRAMEWORK

The SIG has a framework in order to facilitate program analyses. The by this thesis constructed framework is embedded in the existing framework, and thus expanding its functionality. As a consequence, the framework that I created makes use of the functionality already present. This mainly involves the scanners, parsers and tree walking mechanisms. The constructed analyses than were defined and implemented in my framework, now acting as a part of the SIG framework.

The architecture designed for the framework remained quite general, because all different languages are processed in a different way. For some languages parse trees are constructed, while for other languages only tokenizers are available, and for some the only means of extraction are the use of regular expressions. So the framework defines what information should be gathered, while the instances for the different languages than are able to define visitors or regular expressions, in order to obtain that information. The reason why the intermediate representation was used, instead of inspecting raw source for instance, is that already existing analyses in the SIG framework then are able to be used directly in the constructed framework. Reusing analyses plays a major role in the SIG framework, in order to keep it flexible and affordable.

This chapter explains the characteristics of the constructed framework and how this is realized. First the organization of the provided artefacts is discussed. Then the different steps performed in the constructed framework are explained, followed by the internal representation of the collected items and the analysis results.

6.1 ARTEFACT CONTAINER

The artefact container is present to contain fixed information like the database scheme. This object therefore also is concerned with reconstructing the scheme when only `CREATE` statements are present. When the scheme is omitted however, some of the analyses are limited in their capabilities.

When a variable in a PL/SQL query is not shadowed by a database column, the scheme is vital, as explained in section 3.2.2. For this situation an omitted database scheme results in the default behaviour of assuming that no shadowing takes place, because this construct is rather rare in practise. And also for some other analyses the absence of the scheme implies that it becomes less accurate or even can not be performed. The analysis of query equality for instance tends to become less accurate when many tables are used that contain columns with the same name. Another example is the detection of joins by inspecting the query structure, which can not be performed without database scheme, as explained in section 5.1.3. The column references in the `WHERE` clauses are not obligated to define the related table, like for instance `Group.id` does. If the conditional part of the query does not specify the table information, like `FROM User, Group WHERE id = -1;`, the related table of the

referenced column can not be derived without the database scheme. Only one of the options `User.id` and `Group.id` can be present in the database scheme, assuming that the query is valid. Without database scheme this information can not be derived.

6.2 QUERY CONTAINERS

While for each programming language the intermediate representation of the program to analyse differs, creating the need of different instances for the various programming languages, all of the analyses have something in common. This resulted in some common structures to abstract from the specific instances.

A query container stores all retrieved artefacts and intermediate results of a single query. One general query container is defined, which is extended by the query containers specialized for the different programming languages.

For every query detected within the host language a query container is constructed. The container is assigned a unique number, in order to visualise the different containers. It is initialised with the operation type of the query, like `SELECT`, the filename, the path and the line number indicating the location of the query.

Once the queries are detected, the query string representations are added. With string concatenations this is achieved by string reconstruction and for PL/SQL, that uses complete parse trees, a pretty printer is used to achieve this. The variables used in the query, as well as the result variables, are added as a variable description object, containing the following information.

- The name of the variable
- The type, like for instance `String` or `boolean`
- The table it is related to
- The column it is related to

For '`WHERE User.name = :var`' this then results in a variable description that has a name `var` and is related to the table `User` and the column `name`.

Finally, scope information about the surrounding source where the query is in, is stored in the query container, as explained in more detail in section 6.2.2.

6.2.1 Language specific deviations

Since Visual Basic source is tokenized by the SIG framework, the string concatenation analysis stores information in the Visual Basic query container about the token iterator with respect to where the first line of the query string starts and ends. Other information added is the name of the variable the query string is assigned to. This then is used to forward flow and collect all other appended strings. Other languages using tokenizers are able to reuse this container without adapting it, because the same type of information is collected then. The token types for instance are static integers, regardless of which tokenizer is used.

The same holds for the PL/SQL query container, because its structure is applicable to all languages that use parse trees as an intermediate representation. It can be used without adaptation for any tree of the type `Visitable`, while other type of trees only imply minimal adjustment. This type of container holds the following additional information, besides the standard items like scoping and variables.

- The parse tree of the source file the query is in
- The subtree of the query itself

The PL/SQL instance has a reference to the parse tree representing the source file the query is in. This abstract syntax tree is used by the different analyses to filter the required information, such as declared variables and the statement the query is in, as needed for the analysis that detects relations among queries. The different query containers thus can contain trees of different source files, while a single container only has the tree of the file where the query is in. The query subtree contains the parsed and complete query, while the root node of this subtree is the query start.

For COBOL, the queries first are obtained by means of filtering with regular expressions, resulting in clean queries, which then is fed into the PL/SQL analysis resulting in PL/SQL query containers. Those containers than are converted to COBOL containers, in order to remove obsolete information, like the maintained trees, that no longer are needed then.

The Java analysis, containing a parse tree, only required minimal information to collect. Therefore, the basic query container sufficed and was given location information, the type of the query and the string representing the collected query. If more extensive investigation would have been required, the query container also used by the PL/SQL analysis, could be reused.

6.2.2 Scope

Some analyses, like for instance the pattern detection analyses of section 5.2.3, use scoping information of the queries. Fore each query a scope identifier is stored in the query container. A scope identifier uniquely distinct each scope in the following way.

For each instance of a programming language, the different scopes are numbered. In this way they can language independently be stored in the query container. By iterating over the query containers then, a subset of the queries belonging to a certain scope, then is able to be obtained.

As explained in section 5.2.2, scopes in PL/SQL are defined by `BEGIN` and `END` keywords. This notation is used below to illustrate how the different scopes are numbered.

```
1 -- 1
2 BEGIN
3
4   -- 1.1
5   BEGIN
6   END;
7
8   -- 1.2
```

```
9   BEGIN
10
11    -- 1.2.1
12    BEGIN
13    END;
14
15    -- 1.2.2
16    BEGIN
17    END;
18
19  END;
20
21 END;
22
23 -- 2
24 BEGIN
25   -- 2.1
26   BEGIN
27   END;
28 END;
```

Fragment 6.1. Distinction between scopes

Each level inherits the scope identification of its ancestor, and adds an incremented number to it. In this way, each scope is identified unique, preserving scoping information. In order to determine if scope a is in scope b, one only has to take b and check if scope a starts with the scope identifier of b. For instance `1.2.1` starts with `1` and `1.2`, as can be validated in the example above.

6.3 THE DIFFERENT STEPS PERFORMED

The following sections handle the different operations to be performed in order to obtain the required measurements. Each of the steps described in this section is represented in the constructed framework as an object, to be extended by the instances of the different supported languages.

6.3.1 Locate queries

First, an instance of the query locator for a specific language detects the starting points of the different queries and initiates the query containers. This entails collecting all strings present and let them assess by the generic part of the constructed framework. This is because the algorithm determining if a string entails the start of a new query is performed in a language independent way, making it reusable.

For the tokenized languages, the position of the token iterator is stored in the created query container, while for the parsed languages, the subtree is placed in it. For the languages where regular expressions are used, the line number and column position are stored in the container to identify the start of the query location.

6.3.2 Collect queries

When dealing with languages that use string concatenation to define queries, this step involves the reconstruction of the queries by inspecting the various strings present.

The constructed framework facilitates the extraction of queries embedded in strings. However, the different programming languages all have their own way to compose and construct the queries. For this reason the implementation of the framework enables a plug-in structure, containing a framework generic part and a language specific part, in order to fulfil all needs.

Concerning the different programming languages, the tokenised scanner exposes different types of tokens. This part just cannot be generic. But for languages having the same kind of tokenization, the only adaptations to be made are when language derivation occurs. The framework takes different ways of concatenation in mind, resulting in some amount of flexibility. While things like `&=` and += are easy to support, some other structures are not so straightforward due to the absence of a complete parse tree. In Java for instance a line semantically ends with a semicolon, creating the possibility for having multiple statements in one line. In Visual Basic the line semantically just ends when a newline character is encountered. No other delimiter is used there. Adding support for new languages means doing a intelligent copy of Visual Basic instance, i.e. the parts that specifically are performed for Visual Basic. With the structure and approach already present, this takes minimal effort.

The following derivations, like the tokens used and also the structural difference in language constructs, are parameterised. The assign tokens like =, the concatenation tokens like '`&`', '`&=`' and '`var = var & ""`', how function calls manifest like '`exec("select ?", varName);`' and finally how parameters out of the host language are passed to the query. In order to remain flexible, the parameterization takes place by extending the base objects and thus creating an instance of the analysis for some particular programming language.

For the languages that use regular expressions to extract information, this step in the process entails filtering the query blocks. Languages that use parse trees however, already obtained the complete query in the previous step, when the subtree was located.

The analysis to determine which scope each query is in, as discussed in section 6.2.2, is defined in each instance of specific programming language. This cannot be generalized, because it relies on the, sometimes complex, language semantics and is obtained by the use of tokenizers, tree visitors or regular expressions.

6.3.3 Complete queries

The total PL/SQL analysis, which itself also contains all steps described in these sections, is the component all other analyses rely on. This is because after obtaining the queries, the PL/SQL analysis is performed over them. So the analysis of languages other than the PL/SQL language, are wrappers around the PL/SQL analysis. This enabled us to reuse algorithms, like for instance the analysis constructing the CRUD table and the one comparing equality among queries. These tables however contain information about multiple queries, and thus not are stored in a single query container. Instead, the generic part of the framework stores these constructed tables, making them accessible for the different language specific instances.

For the string concatenation cases, the extracted queries thus are fed into the PL/SQL analysis. And after obtaining those results, the involved instance containing the query completion specifics, performs dataflow in order to find the related variables and the dependencies among queries.

The same holds for the languages that use regular expressions to extract information. There, dataflow is performed by some defined regular expressions, in order to obtain the names of the used variables. Languages that use parse trees determine which identifiers reference a variable in the host language by inspecting the tree. Identifiers from the tree are collected and compared with a database scheme, if present. Otherwise the semantics of the programming language is taken into account, with for instance declared variables.

In order to facilitate the process of collecting information out of trees or with the use of regular expressions, extractors are present, isolating the functionality of extracting certain information or language constructs. Current extractors easily are extended, accompanied or replaced by adapted extractors. The extractors themselves rely on existing pattern matching mechanisms provided by the SIG framework.
This design provides means to expand requirements with minimal effort. Often, the companies that hire the SIG to perform an assessment, require specific information to gather. Take for instance a bank that migrated from a 9 digit number to a 10 digit one. When assessing the impact and change requirements, only specific extractors are needed and thus created. The same holds for finding and handling specific structures present in some language.

6.3.4 Find relations between queries

This step involves finding the dependencies between queries, by the use of control flow. Here the queries depending on each other are detected. These dependencies involve result variables that are used in other queries, as explained in section 4.2.1, the result variables used in conditional statements, as described by section 4.2.2, and the result variables used in loops as section 4.2.3 discusses.

This step also involves detecting the defined patterns of section 5.2.3, as for instance the manual cascade delete. As explained there, this can be performed in a language independent way, by only inspecting the query containers. The query containers after all already contain the needed scope information.

When a dependency is encountered, a reference to that query is added to the query container that it depends on. Each query thus knows what queries depend on the information it produces.

6.3.5 Construct measurements

As a last step, the measurements object collects the numbers related to the different metrics, from the analysis results. This information is obtained from the query containers and the derived tables containing information about multiple queries, as for instance the CRUD and query equality tables. This step mainly involves adding up the different numbers and adding the measurement results to the SIG framework.

Because the different languages may require additional metrics, as for instance queries in strings are undesirable in PL/SQL, these results are able to be added to the measurements by the language specific instance of the measurements object.

6.4 Conclusion

The proof of concept of the approach described in this chapter was provided when the SIG asked to, on a short term, collect queries out of a commercial Java project. The SIG framework provided the basis to construct a fully parsed AST by making use of PuppyCrawl and Antlr. In a matter of days basic support for this language was added and the required information collected.

As can be seen, the approach differs from the one described in [7]. The benefit gained by the approach taken in this thesis, is that no parse tree is needed, for no extensive flow graph is required. The framework also is not required to know how queries are executed by the normal library, which is highly platform dependent. And any proprietary library accessing the database and facilitating the other layers of the analysed application is allowed to be omitted for the analysis.

CHAPTER 7
QUERY MEASUREMENTS

When analysing queries, the focus can lie on both the structure of the query itself, as well as the use of its result by the surrounding application code. The difference between a well structured and a ill structured query can lay in the matter of complexity, and can be determined by applying some defined metrics.

After gathering all kinds of information by performing the analyses described in the previous chapters, this information is used to derive an indication of the quality and the level of maturity of the incorporated SQL statements. The other aspect is the identification of the places where undesirable constructs are used. Analysis is performed over the analyses results in order to make those results concrete and directly usable. In the final result the queries actually are visualized as query locations. And by assigning scores to the queries for the different indications of misuses, a prioritised list is formed pointing out the locations most likely to contain opportunities for improvement.

The following sections address the measurements performed for each encountered query. Some of them use the results of the analyses defined in the previous sections.

7.1 QUERY OCCURRENCE MEASUREMENTS

When inspecting the quality of an application, not only the query structures or complexity should be taken into regard. The software architecture quality with respect to the database access does matter too.

In order to gain insight in the number of queries present, the query occurrence measurement counts the different queries encountered. The results then are expressed per file, for this helps to gain insight in the structure of the system. Files situated in the persistence layer are expected to contain queries, while for the domain and graphical user interface layer this is highly undesirable. Another indication is being retrieved about how centralized the handling of the queries is. Having the queries spread throughout the system does not benefit maintainability nor reliability or stability.

The measurements are further categorized by type. This categorization can be defined in several ways. The constructed framework shows the occurrence of each query operation type per file, as is visualized in the table below.

	SELECT	INSERT	UPDATE	DELETE	CREATE	ALTER	DROP
TCDMUY07.COB	14	11	25	6	0	0	0
PJT_PARAM.trg	0	0	0	0	2	0	0

Now it can easily be seen what happens in which files, with respect to the query operations performed.

Another categorization however could have been as follows.

- Retrieving information
- Mutating data
- Changing structure

The first item then refers to the `SELECT` queries retrieving information out of the database. The second item points the `INSERT`, `UPDATE` and `DELETE` queries, only affecting the data contained by the database. The last item then refers to operations changing the database scheme, i.e. `CREATE`, `ALTER` and `DROP`. This approach abstracts from being too detailed, while preserving the semantics of the types of the operations performed.

7.2 VARIABLES USED BY QUERIES

This section describes the different measurements performed by the constructed framework, with respect to the holes in the queries, which at runtime are interpolated with the values contained by variables.

7.2.1 Number of result variables

This measurement counts the number of result variables that are used to store the query result in. An example is shown below.

```
1 SELECT name, street, phone
2 INTO :name, :street, :phone
3 FROM User
4 WHERE id = :uid
```

Fragment 7.1. Query using result variables

This select query contains three result variables, i.e. the `:name`, `:street` and `:phone`. And, as will be explained in the next section, it contains one input variable, viz. the `:uid`.

In order to show that this metric is not always equal to the number of select items, i.e. the columns selected for the query result, take for instance the following PL/SQL example.

```
1 CURSOR cur
2 IS
3    SELECT id, name
4    FROM   User;
5
6 FOR rec IN cur
7    LOOP
8 -- et cetera
```

Fragment 7.2. Cursor as result variable

Now there is only one result variable, viz. the `cur`, while `rec.id` and `rec.name` indeed still result in the amount of select items. But the query result now initially is stored centralized, probably decreasing complexity.

7.2.2 Number of parameters

This straightforward metric counts the number of variables used in the query and its subqueries. Two ways of counting can be identified.

The first one is just by counting the number of holes in the query, allowing duplicate variables to be counted. For the example below this measurement then gives two for the `:gen` variable.

```
1 WHERE User.gender = :gen
2 OR Groups.gender = :gen
```

Fragment 7.3. Query using a variable multiple times

The second approach only counts the different unique variables used. Thus resulting in one for the example above.

The constructed framework applies the first approach. This is because the number of variable places in a query tells something about its complexity. The more parts that are variable, the less understandable the semantics get.

7.2.3 The context of the variables

Variables in a query and its subqueries can be categorized by the location, i.e. context, it is used in. A condition, i.e. in the `WHERE` part, for instance is allowed to contain variables, but when the table queried is variable, an undesirable pattern is indicated, for this construct is error prone, increases complexity and is harder to test.

One operation performed, is collecting the `WHERE` clauses that have the possibility to insert a variable value within an equation, i.e. both the lhs and the rhs are allowed to contain variables. In that case the column referenced or value filtered can differ each time the query is constructed. When variables used in the `WHERE` clause evaluate to one value only, this not is seen as an undesirable construct, but rather as a setting. The database scheme finally is needed to determine from the collected values if they imply a value to filter or a (table and) column reference. When no scheme is present, the default assumption is the filtered value, because this is the most often occurring situation. Values used by the query to filter the result, e.g. `WHERE name = :uname`, are common and therefore marked as desirable. This pattern is shown in the following example.

```
1  orderId = "VPO.ORDERID";
2
3  if(enabled)
4     col = "I.ISPNAME";
5  else
6     col = "I.USRNAME";
7
8  Set rsService =
9        Database.Get( "SELECT I.ISPID, VPO.SERVICEID " _
10                   & "FROM VPORDERS VPO, ISP I " _
11                   & "WHERE " & col & " = VPO.ISP " _
12                   & "AND " & orderId & " = -1" )
```

Fragment 7.4. A query with variety in the table reference

While the `col` can have multiple values, the `orderId` column referred to in the left hand side, always is evaluated to one single value. In order to resolve the value of `col` as a column reference, aliasing is taken into account. The database scheme then is inspected for the presence of '`ISP.ISPNAME`', resulting in a match for the column reference. When tables or columns are referenced dynamically, i.e. when they are able to have a different value each time the query is constructed, this is added to the measurement result. The `col` in this example clearly does match this pattern.

The following contexts are distinct in increasing order of being harmful.

- Variable criterion - `WHERE` clause
- Variable result column -select item
- Variable additional criterion - `HAVING` clause
- Variable grouping - `GROUP BY` cause
- Variable table - `FROM` clause

Variables used in the `WHERE` clause is common practise. The column queried for the result however normally is fixed. The same holds for the `HAVING` and `GROUP BY`. Finally varying the table queried is undesirable for this indicates a lack of structure in database design.

A measurement result then for instance could be the following table, showing two query locations.

Variables	in `WHERE`	in select items	in `HAVING`	in `GROUP BY`	in `FROM`
dump65.out@1369	1	1	0	0	0
dump65.out@1380	1	1	0	0	0

This example however still does not tell which query is worse. This remains a human effort, because it is difficult to specify how many variables in an expected context is equally wrong as one variable in a harmful context. Are 10 or 100 variables in a `WHERE` clause less, equally or more harmful than one variable in the `FROM` clause?

The advice here is to inspect the top five of every context manually, in order to be able to say something about the quality of the usage of queries within the application.

7.3 QUERY STRUCTURE RELATED MEASUREMENTS

The structure of a query says something about its complexity and thus its understandability. The following sections handle these types of measurements.

7.3.1 Equality compared to the other queries

For each query the equality with respect to all the other queries is taken. For each pair of two queries, the equality is expressed by a number between 0% and 100%.

Most measurements involve counting the collected artefacts. For instance the number of result variables and the amount of queries it depends on, i.e. the number of used query results of

other queries, result in a concrete number. But for the amount of equality compared to the other queries this is not so straightforward. When two queries are 100% similar, both queries score 100. Now when they match all the other queries 0%, their equality score will be 100. This then is less than a query that has 10% equality with 15 other queries, resulting in a score of 150.

Two equality measurements now can be defined. The first one is to take per query the highest equality percentage with respect to some other query. The queries that show the most resemblance then gain the highest score. The second measurement is to sum up all equality percentages in order to gain insight in queries that have parts that are similar to queries at many other places. This is also the default for the constructed framework, because queries having a lot in common with many other queries are likely to be positioned in a place lacking design. This than is a interesting entry point for manual inspection.

7.3.2 Number of tables used

A simple measurement for complexity is performed by counting the number of tables used by each query. The more tables used, the less understandable the query gets, because of the many joins and nesting they are forced to use.

Instead of inspecting the `FROM` clause, this information is gained out of the CRUD table, because this table already was constructed previously, simplifying this measurement. As explained in section 4.1.1, a context sensitive CRUD table contains information about the different query operation types a query performs on what database tables.

7.3.3 Inner queries

A subquery[22] is a SQL SELECT statement that is placed in the predicate of any other SQL statement. For the usage of inner queries, each level deeper implies a significantly increased complexity. This is because the usages of intermediate results tend to be less understandable and more difficult to test. With each level deeper many more states are involved. It is difficult to reconstruct all those states in test cases, limiting the test possibilities.

This increased complexity however could be taken for granted when the performance is benefiting from this construct. There are situations that the DBMS can solve the required result quicker than the host language performing it manually.

As a measurement, this is expressed as the number of inner queries contained per query location, because the amount of inner queries reflect the complexity of the query as a whole. Having three queries sequenced with the `UNION` keyword, at one level deep, is considered more harmful than one query at a depth of level two.

7.3.4 Number of queries combined by the UNION keyword

With the use of the `UNION` and `UNION ALL` keywords, the final query result is conducted by combining the results of different smaller `SELECT` queries. The more isolated `SELECT` queries there are combined, the less selfevident the query as a whole and its result

becomes, as explained in section 5.1.4.

This measurement counts, per query, the number of used `UNION` and `UNION ALL` keywords.

7.4 RELATIONAL MEASUREMENTS

As will be explained in the following sections, the relations among queries can be measured and expressed in a number, for identifying those query locations that are likely to contain undesirable situations.

7.4.1 The number of depending queries

Multiple `INSERT` or `DELETE` queries can use data obtained by one `SELECT` query as is explained in the sections 5.2.1 and 5.2.2. The analysis results these sections constructed, is used by this measurement of dependent queries. For the `SELECT` queries it is counted how many other queries there are, that use its query result, creating a dependency on the particular `SELECT` query.

The queries not producing a result, i.e. not retrieving information out of a database, do not have this type of dependency, because they just perform actions. Their result, i.e. side effecting, is not used directly as input for other queries.

7.4.2 The number of joins used

By using high quantities of joins, applications get more difficult to adapt. This adaptability then can be measured by counting the amount of occurrences of the `JOIN` keyword.

As explained in section 5.1.3, the number of joins also can be derived from the structure of the queries. Per query then can it be counted how many joins it uses to produce its result. Due to time limitations however, and because this information cannot reliably be resolved without a database scheme, no robust implementation was added to the constructed framework. So instead only the `JOIN` keywords are counted.

For this measurement the type of joins, i.e. the joins defined by the query structure as well as the join keywords, are treated as equal, for the semantics and complexity remain the same, leaving how to define the joins to what the creator prefers.

Another thought is that this approach partly can be substituted if less accuracy is allowed, by measuring the amount of tables referenced in the `FROM` clause. After all, the tables specified there normally are connected in some way in the criterion part of the `SELECT` query. To prove this instead of assuming it, the approach discussed in section 5.1.3 can be used.

7.5 CONCLUSION

The first approach to express the quality of each query was to create a composition of all measurements in a penalty score. However, just like metrics on software applications, quality never can be expressed in a single number. Quality is measured by inspecting the different aspects separately. The same holds for the quality analysis on queries embedded in

host languages. So the final way to observe quality is performed by looking at the amount of occurrences as defined by the metrics previously described.

In the constructed framework, the results are exported to an Excel sheet, enabling manual inspection easily to identify the top of wrong queries for each type of measurement. In this way queries with a large amount of used variables are spotted independent of queries with no parameters but with several levels of inner queries. In practise this approach proved to be very useful.

The metrics that proved to be the most useful, i.e. reflecting the quality, are the measurements with regard to the number of variables used, the amount equality and the number of inner queries. The number of used variables and result variables address the quality of the data model and the inefficient usage of the structured query language. The equality aspect expresses maintainability, while the nesting level affects the complexity and testability.

In this stage, after constructing the measurements, it is too soon to say something about how the results of the defined metrics are to be interpreted. Only time and experience can point out what values the different metrics are allowed to have in order to regard the software system as healthy. The focus of this thesis is to provide means to perform the different metrics, while performing an extensive study at the semantics of the results, by comparing the results of many software systems, is out of the scope of this thesis.

CHAPTER 8
CASE STUDY RESULTS

In order to determine how effective the defined metrics of the previous chapter are, analysis was performed on a few commercial software products. The outcome then is discussed in this chapter, in order to show how the different measurements reflect the actual situation as it is present in the software. Further, the cases were used during this thesis to indicate how mature the constructed framework was, and how it would function in practise.

So for the case study, the metrics defined in the previous chapter were performed on the following commercial software products.

- PL/SQL source of a bank
- PL/SQL source of an energy supplier
- COBOL source of a bank
- Visual Basic 6 source of a telecom company

Here the banks mentioned above refer to different banks. Experiments with respect to the defined metrics, resulted for the different projects in quantities that are explained in the following sections. The highlighted values in the tables there imply that the specific file or query had the highest score for that measurement within the project.

8.1 PL/SQL SOURCE OF A BANK

The assessment, already performed by the SIG before this thesis, resulted for this project in the following general conclusions. The source code quality was found to be acceptable and the architecture was considered to be alright. In total there were almost 74,000 lines of code, while about a tenth of this was regarded as fairly complex.

	amount	SEL	INS	UPD	DEL	CREA	ALT	DRP
All	2994	1978	319	258	194	176	0	0
mwxf_mnt.pks	36	21	0	0	0	0	0	0
STTAB.pks	195	148	18	11	17	0	0	0
spt_mn.pks	122	57	51	14	0	0	0	0
ORDELI.pks	83	53	3	26	1	0	0	0
ACVR.pks	101	66	6	8	20	0	0	0
PR_VAL.vw	2	0	0	0	0	2	0	0

As to be expected, most queries are of the `SELECT` type. The two `CREATE` queries are situated in the file taking care of views, so this does not indicate an anomaly. No `ALTER` or `DROP` queries were encountered, which is quite normal for an application.

Besides the basic measurements defined in the previous chapter, per language additional measurements are optionally performed. For the PL/SQL projects we measured the number of

queries located in strings. As we did not expect to encounter queries composed by strings within the PL/SQL source, the following results were quite interesting. The table below shows for each file listed above, how many queries were composed out of strings.

All	69
mwxf_mnt.pks	15
STTAB.pks	1
spt_mn.pk	0
ORDELI.pks	0
ACVR.pks	1
PR_VAL.vw	0

In total 69 queries turned out to be constructed by means of string concatenation. This is considered to be an undesirable situation, because PL/SQL provides specific language constructs for queries to be defined in.

The metrics as defined in the previous chapter resulted in the following quantities.

file	line	res vars	vars	nesting	union	join	tables	equality	dependencies
CLSPJT4.pks	442	25	1	0	0	0	3	172%	0
ACVR.pks	1279	0	126	0	0	0	1	126%	0
ACVR.pks	513	0	1	3	0	0	6	0%	0
llalv.vw	1	0	0	0	47	0	48	0%	0
STTAB.pks	3179	1	0	0	0	0	1	7288%	0
GEO.pks	1157	1	4	0	0	0	2	460%	66

The query most resembling parts of other queries is the one in 'STTAB.pks'. The 7288% here implies that the query is totally equal to 73 other queries or half equal to 146 other queries, while there are almost 3,000 queries present. So in total quite some duplication is present among the queries.

The nesting is within the acceptable limits, but the amount of queries combined with the use of `UNION` keyword does make it complex. Especially considering the mount of tables used. Also the enormous amount of used variables, i.e. even for a `INSERT` query, signals complex source code surrounding such query. This indication of complexity is amplified by the amount of queries that directly are depending on a single other query, i.e. 66 for this project. Changing such a query can imply that the behaviour of the dependent queries change, limiting the adaptability of the application.

For this PL/SQL project, no database scheme was present, implying that the found variables were not considered to be shadowed by the columns present in the database table. Finally, no join keywords were used and all variables were located outside the `FROM`, `GROUP BY`, `HAVING` clauses and select items.

8.2 PL/SQL SOURCE OF AN ENERGY SUPPLIER

For this PL/SQL project the query occurrence measurements resulted in the following quantities.

	amount	SEL	INS	UPD	DEL	CREA	ALT	DRP
All	5475	3427	787	912	245	0	0	0
dump221.out	59	22	4	7	2	0	0	0
dump222.out	211	71	7	71	56	0	0	0
dump208.out	206	101	40	28	22	0	0	0

Quite some queries were present and, as expected, most of them were of the type `SELECT`, while having more `UPDATE` than `INSERT` queries also is normal. No `CREATE`, `ALTER` or `DROP` queries were found, which is standard for an application without its initialisation.

This application, just as the previous PL/SQL project, contained some queries constructed by the means of string concatenation, as shown in the following table.

All	104
dump221.out	24
dump222.out	6
dump208.out	15

Unlike our expectance, both this project, as well as the project described in the previous section, contained queries constructed out of strings. This is regarded as a bad practise as already explained in the previous section.

The different measurements as described in the previous chapter, for this project are as follows.

file	line	res vars	vars	nesting	union	join	tables	equality	dependencies
dump444.out	178	59	2	0	0	0	1	97%	0
dump359.out	166	0	66	1	0	0	11	9%	0
dump249.out	136	0	20	4	0	0	1	69%	0
dump98.out	2692	0	10	0	2	0	2	81%	0
dump65.out	832	0	33	0	0	0	18	653%	0
dump296.out	3349	0	17	0	0	0	2	3302%	0
dump218.out	1353	2	3	0	0	0	1	186%	37
dump65.out	1369	0	2	0	0	0	2	351%	0

For this application, the maximum amount of result variables used by a single query is more than twice compared to the previous PL/SQL project. However, the maximum amount of holes in a query is only the halve of what is present in the previous project. The last entry, i.e. a `SELECT` query in 'dump65.out', contains one variable in the select items and one in the `WHERE` clause, indicating a malformed database structure or a bad practise in the host language, as explained in section 7.2.3. In total this was encountered twice in this project.

While having almost twice as much queries compared to the previous PL/SQL project, the biggest duplication in a single query was 3302%, meaning that the query is equal to 33 other queries or half equal to 66 other queries, while for the previous project this equality was 7288%. Relative little duplication thus is present among these queries for this application.

The nesting metric indicates that complex queries are present. This indication is confirmed by the queries that use a large number of tables. For the query in 'dump65.out' this is remarkable because this query is not being composed out of queries combined with the `UNION` construct. Refactoring this query therefore probably will simplify the application. No join keywords were used, and from the point of view of the surrounding source, the complexity of queries depending directly on a single query is at maximum 37, i.e. only the half of what is the case for the previous project.

8.3 COBOL

The general conclusions of the SIG for this project, as already defined before this thesis, are as follows. The COBOL source was found to be rather complex, with about 50% of code duplication. Further, a lack of logical architecture was encountered, while the project contained over 600,000 lines of code.

The number of queries encountered are as follows.

	amount	SEL	INS	UPD	DEL	CREA	ALT	DRP
All	496	282	65	115	34	0	0	0
x.COB	59	24	9	21	5	0	0	0
y.COB	56	14	11	25	6	0	0	0
z.COB	50	19	11	11	9	0	0	0

This first overview shows that the different queries are well centralised, because 33.3% of all queries are contained by just three files. Although this led to large files, the queries at least are not scattered throughout the many source files.

The other observation is that 56.8% of all queries are `SELECT` queries, 23.3% are `UPDATE` operations, 13.1% is `INSERT` and 6.9% are `DELETE` queries. Per file the composition of these query types hardly differ, thus showing a normal usage of the different query operation types.

No database scheme was present, but because only a few tables were used per query, it had only little influence. The check of equality for instance tends to become less accurate when many tables are used that contain columns with the same name. Because of time constraints only a limited dataflow was performed, resulting in a measurement of the dependent queries, not solid enough to draw conclusions from. This metric therefore was omitted in the table below.

file	line	res vars	vars	nesting	union	join	tables	equality
x.COB	3485	48	1	0	0	0	1	9%
x.COB	3601	0	49	0	0	0	1	4%
x.COB	1158	23	5	0	0	0	5	15%
w.COB	1502	13	2	0	0	0	2	34%

The use of a lot of variables, made the surrounding source complex, but kept the queries themselves very simple and basic. The low amount of duplication among queries can be explained be explained by the many unique variables used, and by the fact that for a project of this size not many queries were defined. Only one query per 1,200 lines of code was present.

Even more, the fact that no inner queries and unions were detected implies a low complexity in the queries. This conclusion is supported by the observation that all variables were located outside the `FROM`, `GROUP BY`, `HAVING` clauses and select items.

So the queries in this project proved to be simple and straightforward. No nesting of queries occurred, only a handful of queries used more than two tables, while the most resembling queries are only 34% equal. In order to achieve this simplicity, the programmers used many, uniquely named variables to handle the data throughout the application. So the work was shifted from the queries and the DBMS to the host language. This leads to moving data around quite a bit in the host language, meaning unnecessary indirection. This is exactly what the assessment of the SIG revealed, but now also is supported by the point of view of the queries.

8.4 VISUAL BASIC 6

Only a small part of the total project was provided to the SIG, i.e. three files containing about 10,000 lines. The result of the extraction was placed in a single dump file, but could just as well have been separated as it was in the three files.

	amount	SEL	INS	UPD	DEL	CREA	ALT	DRP
All	78	54	8	15	1	0	0	0

For every 128 lines of code a query was detected, while the quantities were logically divided over the different query types. Most of the queries performed the retrieval of information, only a few added information and a few updated information in the database. No queries affecting the database scheme were encountered.

For the Visual Basic programming language no parse trees are constructed, limiting the ability to perform extensive data flow. Therefore the detection of relations among queries was not robust enough to perform. With no database scheme present, the results of the metrics were as follows.

file	line	res vars	vars	nesting	union	join	tables	equality
dump.out	95	0	45	0	0	0	1	4%
dump.out	260	1	2	1	0	0	1	36%
dump.out	269	1	2	0	0	0	4	57%
dump.out	487	1	1	0	0	0	1	1221%

For the result variables, `Set` objects were used consistently. The sets than act as a single point of entry, in order to provide the results of the different selected columns, simplifying the surrounding source. However, for the variables related to the holes in the queries, the `INSERT` queries use quite some variables, which indicates some complexity at that point in the surrounding source. The variables finally were directly inserted in the strings composing the query, jeopardizing the security by creating the risk of SQL injection as explained in section 3.1.3.

Quite some duplication was encountered, because from the 78 queries, the most resembling query is to be considered identical to 12 other queries. The queries themselves finally are to be considered not complex, because not much nesting and no unions were encountered, while the maximum amount of used tables is only four.

No join keywords were used and all variables were located outside the `FROM`, `GROUP BY`, `HAVING` clauses and select items, indicating no anomaly in that sense.

8.5 JAVA

Sometimes it is not necessary to perform all metrics on a software application. This was the case when the SIG assessed a Java application and wanted to find out how many queries it contained. The expectation was that no queries were to be found, for the application logic was positioned in stored procedures, containing queries at the database side. In only a matter of days the support for locating queries in Java was added.

The assessment, performed during this thesis, resulted in the following general conclusions for this project. Over 300,000 lines of code were encountered, while the persistence package contained about a quarter of this. The source was found to be quite complex and contained rather complicated data access. Further, a large amount of code duplication, i.e. about one third of the total source, was detected.

A surface scan in order to detect queries in the Java source of a bank was performed, and 235 queries were detected.

	amount	SEL	INS	UPD	DEL	CREA	ALT	DRP
All	235	28	12	140	45	10	0	0

After a first manual inspection we noticed that many queries were only for logging purposes. So valid queries were passed to a logging method, implemented by their own. For this project then the extra rule was added to ignore queries passed to this method. This only took minimum effort and led to the situation where 65 queries were found, with only a third of them inside the persistence package. So the adapted results now look as follows.

	amount	SEL	INS	UPD	DEL	CREA	ALT	DRP
All	65	28	1	19	15	2	0	0

Most of the queries were of the type `SELECT` and no `ALTER` or `DROP` queries were found, indicating a normal situation. The two `CREATE` queries were located in the batch package, making their presence somewhat less abnormal.

The queries were mainly divided over three different packages.

package	amount
persistence	22
batch	20
common	22

While it is quite common for such packages to contain queries, the fact that the application embeds some queries while the fast majority of the queries is contained in stored procedures at the DBMS side, the presence of the queries were regarded as undesirable.

8.6 CONCLUSION

With the former SIG framework no indication could be obtained about the quantities of queries in the different host languages. As a result no measurements, like for instance the equality or the amount of used variables, could be performed. Now the constructed framework does support these metrics, queries embedded in languages are no longer ignored.

For all the analysed software projects, the experiment revealed, after manual inspection, that the obtained values actually do reflect the situation, as it is present in the software application, as described in the previous sections.

For the variables related to the encountered holes in the queries, not all holes were located in the `WHERE` clauses, the `SET` clause of an `UPDATE` query or in the `VALUES` section of an `INSERT` query. In two occasions, the column selected in a `SELECT` query was variable, indicating a bad practise. For the remaining variables, the `INSERT` and `UPDATE` queries, are the most likely to contain many parameters. When for instance having some form taking input from the user, many values can be inserted into the different columns of a table, while the use of arrays in that respect is somewhat cumbersome, for the positions than are required to be mapped to the different fields of the form.

No `ALTER` queries were encountered, but in some occasions, as experienced in my research, they are used to define the foreign keys. The `CREATE` queries then only define the table, column and data types, while the `ALTER` queries specify the relations like primary and foreign keys for instance.

Finally, the findings of this case study did not contradict the results of the assessments performed by the SIG.

CHAPTER 9
CONCLUSIONS AND FUTURE WORK

This chapter discusses the results and conclusion of this, partly empirical, study. The in the previous chapter inspected commercial software systems, gained an understanding of how to define the level of quality of applications, with respect to the queries they contain, to how the queries interact with the surrounding source, as well as to the database schemes they use.

9.1 CONSTRUCTED FRAMEWORK

As a part of this thesis, a framework was constructed in order to simplify measurements on queries used in all kinds of different programming languages. As a proof of concept, the framework was used to perform a variety of measurements, as defined in chapter 7 and clarified in chapter 8.

Each time the support for a different programming language was added to the analyses, new insights were gained. This proved to be very useful when regarding the aim to construct a general and flexible approach that could be conducted for all kinds of different programming languages. Creating new instances for the different analyses, in order to support new languages, helped to improve the structure of the constructed framework.

Before this expansion of their framework, the SIG was not able to obtain a notion of the quantities of the queries that were present in applications. And in the cases of strings forming queries, the SIG was not able to reconstruct the queries in order to determine its quality. Finally, the SIG had no metrics defined for queries with respect to the application they are embedded in. This thesis provided an answer to these problems. The results likewise sufficed to be useful for software assessments performed by the SIG.

9.2 KNOWN LIMITATIONS

The type of problems focussed on in this thesis, impose some limitations when regarding other undesirable constructs in using the structured query language, as will be explained in this section.

9.2.1 Immature functionality

Some parts of the instances for the different programming languages in the constructed framework could be expanded and improved, in order to gather more information benefiting the accuracy of the quality indication.

For instance, expanding the performed data flow analysis would improve the analysis detecting dependencies between queries, as well as it would improve the analysis finding the values associated with the variables that are used by queries to fill the holes during execution.

And in the different instances, only limited interprocedural analysis are performed, leaving

room for improvement in this aspect. For instance, the impact analysis then could point out which methods, files, packages or modules are affected when adapting a single query. Luckily however, already existing means in the SIG framework can directly be integrated with the constructed framework, making this somewhat more easy.

Finally, section 5.1.3 elaborates on how to detect joins by inspecting the structure of queries. The instances of the different programming languages in the framework however do not yet fully support this, even when a database scheme is present. When expanding this in order to make it more functional and robust, it could provide more insight in the complexity of the database scheme, and how that is reflected in the queries.

9.2.2 Scope

The scope of the research included relational databases and imperative programming languages, but excluded other database models and functional languages. Also the domain specific languages (DSL) were not taken into regard. The reason for this is that, for assessments of software systems by the SIG, only relational databases and imperative programming languages are encountered. The analysed SQL however was not bound to a single SQL dialect. This was enabled by the presence of many grammars in the SIG framework, describing the different SQL dialects.

9.2.3 Undetected undesirable situations

Though much effort is put in detecting a variety of undesirable situations, some are likely to stay undetected by the model imposed. The reason for this simply is that when the undesirable construct is conducted by the design specifics, the symptoms get more subtle and less obvious to indicate.

One example for instance, as published by TheDailyWtf.com[26], can be seen below. This snippet shows the dataaccess layer of an existing commercial enterprise .NET application.

First an alias is created for defining the SQL language.

```
1  public class SqlWords
2  {
3    public const string SELECT = " SELECT ";
4    public const string TOP = " TOP ";
5    public const string DISTINCT = " DISTINCT ";
6    public const string FROM = " FROM ";
7    public const string INNER = " INNER ";
8    public const string JOIN = " JOIN ";
9    public const string INNER_JOIN = " INNER JOIN ";
10   public const string LEFT = " LEFT ";
11
12   /* SNIP */
13
14   public const string ORDER_BY = " ORDER BY ";
15   public const string ASC = " ASC ";
16   public const string DESC = " DESC ";
17 }
```

Fragment 9.1. Typing the Structured Query Language

And this is complemented by other classes as `SqlTables`, containing all the table names, and `SqlMisc`, containing things even as `NUMBERS_ONE = "1"`

Now any combination will do, if you just accept the cumbersome aspect of it.

```
1  public class SqlQueries
2  {
3    public const string SELECT_ACTIVE_PRODCUTS =
4      SqlWords.SELECT +
5      SqlWords.STAR +
6      SqlWords.FROM +
7      SqlTables.PRODUCTS +
8      SqlWords.WHERE +
9      SqlColumns.PRODUCTS_ISACTIVE +
10     SqlWords.EQUALS +
11     SqlMisc.NUMBERS_ONE;
12
13   /* SNIP */
14
15   public const string UPDATE_LOGON =
16     SqlWords.UPDATE +
17     SqlTables.CREDENTIALS +
18     SqlWords.SET +
19     SqlColumns.CREDENTIALS_LOGON_NAME +
20     SqlWords.EQUALS +
21     SqlMisc.PARAMS_FIRST +
22     SqlWords.COMMA +
23     SqlColumns.CREDENTIALS_LOGON_PASS +
24     SqlWords.EQUALS +
25     SqlMisc.PARAMS_SECOND;
26 }
```

Fragment 9.2. Building a query by using the defined types

However this seems extraordinaire and exceptional, the problem of detecting such constructions lies in the fact of the subtle use of indirect SQL. One would first have to perform global constant propagation throughout the whole application, in order to detect the start location of the query. Otherwise the identification of the presence of the query would not be possible. So here are choices and tradeoffs to be made. Is it feasible to perform such time consuming activities, or is it sufficient to detect the spots where SQL queries are started in strings. Note that the SQL language only has a finite set of start symbols like `SELECT`, `INSERT` and `UPDATE` and so on.

9.3 FUTURE WORK

As explained in section 9.2.1, not all analyses are robust or mature enough to be optimally used in practise. Therefore, improving the lacking functionality of the constructed framework should be considered as future work. The other categories of future work are explained in the following sections.

9.3.1 Extending the current analyses

One of the considerations to be made, is adding support for identifying the columns addressed by the asterisk in `SELECT` queries. The construct `SELECT *` then is proved to

require only a few columns, enabling the analysis to point out that for instance `SELECT A, B, C` would suffice. This latter `SELECT` statement then is regarded as an improved version of the first, enabling a software assessment to suggest this change, by supplying the replacements. This then also would improve the documentation generation product the SIG offers.

9.3.2 Continuing research in this area

The field of measuring the quality of queries, and in special with respect to the semantics defined by the host language constructing and using the queries, is rather new and under-investigated.

It is in my belief that the field of software engineering would benefit from more extensive research about this topic. This is because many applications use a database, enabling many opportunities for improvement in this area. By extensively observing and comparing the results of the different metrics, for the different programming languages, a more precise impression could be obtained about its accuracy. Besides this, a better understanding of the semantics of the different located structures could be obtained, leading to new patterns to be detected.

Further, linguistic studies as performed on the Rijksuniversiteit Groningen[31, 11] try to understand words by looking at the context. In order to predict and identify contexts, frequency of usage and other statistics are used. Applying this type of research on queries with respect to the language they are embedded in, could improve the accuracy of the measurements.

A paper containing the findings of this thesis was submitted to the System Quality and Maintainability (SQM07), which is a Special Sessions Program within the Eleventh European Conference on Software Maintenance and Reengineering (CSMR 2007). Also here the subject was neglected, for it was not handled before. The submitted paper can be found in appendix A.

9.4 Conclusions

This thesis has set a first step in locating bad practises with respect to queries and database involvement in software applications. While the analyses performed by this thesis still could be improved further more, an effective indication of the quality of software products already is obtained. Besides this indication, suggestions for improvement directly are derived from the results the different measurements produce.

Finding evidence for undesirable relations, like the functionality of foreign keys established by the host language proofs to be hard. The reason why, lies in the semantics of the relations and dependencies. At first, relationships between queries and between query results are detected. Then the nature is determined, indicating if it is a true dependency or not. But to obtain the reason of the dependency tend to be unreliable, for no general pattern or characteristic ensuring the motivation can be given. Intentions can be expressed in many code constructs. A very large amount of highly specific patterns could reflect the programmer's objective. This then provides opportunities for artificial intelligence and self-learning systems,

but this is out of the scope of this thesis.

Suspicious spots, however, are indicated. Regarding the detection of a lack of structure within a database, manual inspection still is needed. The analysis performed luckily shows where to look and also what to look for. This makes the process of detecting ill practises feasible, for it would not be possible to inspect thousands lines of code manually, with actually be able to understand the relations as specified by the code and database. The analyses constructed clearly show its use, by enabling the SIG to gain a quality indication, with minimal effort to be performed by hand.

BIBLIOGRAPHY

[1] Glossary of software engineering terminology. *Institute of Electrical and Electronic Engineers*. ANSI/IEEE Std 729-1983, New York.

[2] *Hibernate Reference Documentation*. Version 3.1.1.

[3] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. *Columbia University*.

[4] Gilad Bracha, Neal Gafter, James Gosling, and Peter von der Ahé. Closures for java. August 2006. Published on Blogs.sun.com.

[5] Bjørn Bringert and Anders Høckersten. Student paper: Haskelldb improved. In *Haskell Workshop 2004*.

[6] Huib J. van den Brink. Quality metrics for sql queries embedded in host languages. In *Eleventh European Conference on Software Maintenance and Reengineering (CSMR 2007)*. Software Improvement Group, 2007.

[7] Aske Simon Christenson, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the International Static Analysis Symposium (SAS 2003)*. University of Aarhus, Denmark, ACM Press, June 2003.

[8] Thomas M. Connolly and Carolyn E. Begg. *Database Systems : A Practical Approach to Design, Implementation and Management*. Addison-Wesley, third edition, 2002. ISBN: 0-201-708574.

[9] Oracle Corporation. *Oracle7 Server SQL Reference*, February 1996. Release 7.3.

[10] Bassel Daou, Ramzi A. Haraty, and Nash'at Mansour. Regression testing of database applications. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2001), Las Vegas, NV*, pages 285–290. Lebanese American University, ACM Press.

[11] Bart de Boer. Computer modelling as a tool for understanding language evolution. In *Proceedings of the First International Workshop on the Emergence and Evolution of Linguistic Communication*. AI Department, Rijksuniversiteit Groningen, 2004.

[12] Micro Focus. *COBOL Language Reference*, May 1995.

[13] Michael Gertz. *Oracle / SQL Tutorial (PL/SQL)*. Database and Information Systems Group, Department of Computer Science, University of California, January 2000. Version 1.01.

[14] Christian Goldberg and Stefan Brass. Detecting logical errors in sql queries. In *16th Workshop On Foundations Of Databases (2004)*. Martin-Luther-Universit ät Halle-Wittenberg.

[15] Christian Goldberg and Stefan Brass. Proving the safety of sql queries. In *Fifth International Conference on Quality Software (QSIC'05)*. Martin-Luther-Universit ät Halle-Wittenberg.

[16] Christian Goldberg and Stefan Brass. Semantic errors in sql queries: A quite complete list. In *Fourth International Conference on Quality Software (QSIC'04)*, pages 250–257. MartinLutherUniversit ät HalleWittenberg.

[17] Christian Goldberg, Stefan Brass, and Alexander Hinneburg. Detecting semantic errors in sql queries. Technical report, MartinLutherUniversit ät HalleWittenberg, 2003.

[18] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *26th International Conference on Software Engineering (ICSE 2004)*. University of California, Davis, ACM Press, September 2004.

[19] GrammaTech, Inc. Dependence graphs and program slicing. CodeSurfer Technology Overview.

[20] Jean Henrard. *Program Understanding in Database Reverse Engineering*. PhD thesis, Facultes Universitaires NotreDame de la Paix namur, August 2003.

[21] David Jordan. A comparison between java data objects (jdo), serialization and jdbc for java persistence. Object Identity, Inc.

[22] Rick F. van der Lans. *Introduction to SQL: Mastering the Structured Query Language*. AddisonWesley Professional, third edition, November 1999. ISNB: 0201596180.

[23] Thomas J. McCabe and Arthur H. Watson. Software complexity. *Journal of Defense Software Engineering 7*, December 1994.

[24] Isabel Michiels, Dirk Deridder, Herman Tromp, and Andy Zaidman. Identifying problems in legacy software preliminary findings of the arriba project. *Vrije Universiteit Brussel, Universiteit Gent, Universiteit Antwerpen*, 2003. In ELISA workshop at ICSM.

[25] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. SpringerVerlag, 2005. Corrected 2nd printing, The Technical University of Denmark and The Imperial College of Science, Technology, and Medicine. ISBN: 3540654100.

[26] Alex Papadimoulis. Enterprise sql. `http://thedailywtf.com/forums/1/64833/ShowThread.aspx`, March 2006. TheDailyWtf.com.

[27] Terence Parr et al. *ANTLR Reference Manual*. University of San Franciso, January 2005. Version 2.7.5.

[28] Evangelos Petroutsos. *Mastering Visual Basic .Net*. Sybex, Alameda, CA, 2002. ISBN: 0782128777.

[29] Julien Rentrop. Software benchmarking by use of source code metrics. Master's thesis, Informatica Instituut, Universiteit van Amsterdam, July 2006.

[30] Bert Scalzo. Engineering better pl/sql. *Quest Software*, March 2006. Oracle product architect.

[31] Gertjan van Noord. Grammarbased natural language understanding, 2003.

[32] Joost Visser and Jeroen Scheerder. A quick introduction to sdf. *Centrum voor Wiskunde*

en Informatica (CWI), April 2000.

[33] Yichen Xie, Mayur Naik, Brian Hackett, and Alex Aiken. Soundness and its role in bug detection systems. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*. Computer Science Department of the Stanford University, ACM Press, June 2005.

APPENDIX

Quality metrics for SQL queries embedded in host languages

Quality metrics for SQL queries embedded in host languages

The following paper is one of the products of this thesis and contains the findings of my research. It was submitted to the System Quality and Maintainability (SQM07), which is a Special Sessions Program within the Eleventh European Conference on Software Maintenance and Reengineering (CSMR 2007).

Quality metrics for SQL queries embedded in host languages

Huib J. van den Brink
Institute of Information and Computing Sciences
Utrecht University
hjbrink@cs.uu.nl

Rob C. van der Leek
Software Improvement Group, The Netherlands
r.vanderleek@sig.nl

Abstract— **Many software systems that use a database lack maintainability and adjustability, for the produced queries are hard to reconstruct and inspect. For instance, the relations between the queries, established by the host language, are not easy to identify. This information however is vital when adapting the application in order to prevent unexpected behaviours. This paper focuses on queries embedded in a host language, while by reconstructing and analysing the queries, insight is gained in its complexity. Also the interaction between the queries and the host language is taken into account. For instance, the explicit structure and relations within a database often are complemented with implicit relations established in the application itself. Therefore analysing the application with respect to the SQL queries it contains, is needed to detect and distil those relationships. As a result, a framework is presented for extracting SQL queries out of host languages and for indicating the quality of the queries.**

I. Introduction

Many applications depend on a database system. The communication and relation with the database system is usually expressed via the Structured Query Language (SQL). The SQL queries are used to access and manipulate data in the database. As a result a great number of source files contain many queries in different ways. Depending on the programming language used, the queries are either isolated from, or embedded within the host language.

When queries are embedded in a host language, it is not easy to obtain an overview of the queries that are produced. However, when the queries can be collected and reconstructed, insight can be gained into the quality of the individual queries. Taking this one step further, the interaction with the host language can be taken into account, because applications can use queries and query results in an undesirable way. This helps in saying something about the maintainability, adaptability and the testability of an application with respect to the database access. And in order to improve the usage of queries in an application, one needs to know which queries exactly require refactoring, and what the factors are that makes it complex. This insight also helps in a maintenance phase, if one knows what to look for, improvement can be performed efficiently by focussing on the right aspects.

This paper presents a few closely related parts. First, a model to extract SQL queries out of host languages is presented, because in order to inspect the queries they first have to be constructed. Then, as a second part, the implicit relations that are not present in the database scheme, but are established by the implementation and semantics of an application, are detected. Finally quality metrics expressing the results of the defined analyses are defined and presented.

In order to obtain an indication of the accuracy and effectiveness of the defined measurements, a case study is presented, showing how the metrics reflect the situation as it is present in the application. The case study was performed on a few commercial software applications, as is explained in section VIII.

II. Related work

The field of measuring the level of quality of queries, by defining metrics for them, is rather new. Only very little research[4], [3], [2], [5] about this topic has been performed. This related work however only focuses on the isolated queries without taking the interaction with the host language into regard.

Work that does take Java as a host language into account, is performed in [6]. It differs from the approach of this paper in the sense that it is very restrictive. This paper presents an approach that is able to handle all common SQL dialects, while the host language could be any imperative programming language. Besides this, the sources to analyse often are incomplete. For this reason the model designed and implemented in this paper requires only the minimum presence of source code.

Other important research has been performed by J. Henrard. His thesis Program Understanding in Database Reverse Engineering[7] defines and describes techniques for performing data structure extraction. However, the focus here is more on reverse engineering and less on quality measurements and metrics.

III. Distilling queries out of host languages

For embedded queries there generally are two types of constructions used within the different host languages. Queries build out of string concatenation and queries located in designated blocks, making the SQL grammar a part of the host language, as with for instance COBOL.

For the string concatenation case, when queries are constructed out of strings, many constructs have to be taken into account. The issues to be addressed when gathering queries in these situations, is the great freedom of expression. The ways the queries are specified are limited to the creativity of the creator. However some constructs are more common than other ones. Therefore a trade-off is be made in the framework to support the most common constructs, while adding specific structures should take minimum effort. In this way the framework could be adjusted for the specific software to gather queries from.

For reconstructing the queries, one has to collect the strings present in the source, inspect those strings and finally reconstruct the query by performing concatenation. By only focussing on the strings creating the queries, one abstracts from specific programming languages and frameworks, even without a database scheme present. Therefore, to filter the strings creating a query, the framework only accepts strings that start with one of the case insensitive keywords: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `ALTER`, or `DROP`. Than a backward flow reveals the variable the string is assigned to. Finally, a forward flow is able to gather the remaining strings composing the query.

The different programming languages all have the same semantics with respect to string concatenation. In order to define these semantics per programming language, the syntactics are parameterised for the analysis. For combining two strings for instance, the operators `+`, `||` or `&` are provided for respectively Java, PL/SQL and Visual Basic. The same holds for operators like `+=` and `&=`. Also the semantic ending of a statement is required in order to perform the analysis, i.e. `;` for Java and [^_]\n for Visual Basic.

While this approach leaves room for false positives, i.e. query parts not used as a query, it in practise turned out to be very effective. More details on how to reconstruct queries is given in [1].

IV. Software architecture quality with respect to queries

In order to gain insight in the number of queries present, the query occurrence measurement counts the different queries encountered. The results then are expressed per file, for this helps to gain insight in the structure of the system. Files situated in the persistence layer are expected to contain queries, while for the domain and graphical user interface layer this is highly undesirable. Another indication is being retrieved about how centralized the handling of the queries is. Having the queries about anywhere in the system does not benefit maintainability nor reliability or stability.

The measurements are further categorized by type. This categorization can be defined in several ways. The constructed framework shows the occurrence of each query operation type per file, as is visualized in the table below.

	SELECT	INSERT	UPDATE	DELETE	CREATE	ALTER	DROP
TCDMUY07.COB	14	11	25	6	0	0	0
PJT_PARAM.trg	0	0	0	0	2	0	0

Now it can easily be seen what happens in which files, with respect to the query operations performed.

Another categorization however could have been as follows.

- Retrieving information
- Mutating data
- Changing structure

The first item then refers to the `SELECT` queries retrieving information out of the database. The second item points the `INSERT`, `UPDATE` and `DELETE` queries, only affecting the data contained by the database. The last item then refers to operations changing the database scheme, i.e. `CREATE`, `ALTER` and `DROP`. This approach abstracts from being too detailed, while preserving the semantics of the types of the operations performed.

V. Query relational analyses

In order to define strategies to detect relations among data in a database, research was performed about how this manifests in real-life applications. And since little to no documentation is to be found about relations established in host languages with respect to data contained by a database, research about this subject was compelled to be a empirical study, taking the retrospective element into regard. The approach taken involved observing the queries and query results contained by existing software projects, leading to an effective solution that functions well in practise.

This section handles the discovered dependencies and relations among queries and query results, because a well-formed and -used query is isolated in the usage of the input parameters as well as the use of the result it delivers. So results that are given as an input to other queries are highly likely to create undesired dependencies.

A. The actions performed by the different queries

For the SQL queries a CRUD table is constructed. CRUD describes the basic operations of a database, and stands for Create, Read, Update and Delete. Examples of those operations respectively are, inserting new rows, selecting data out of the database, changing information and deleting data. In order to gather quality information, such a table is constructed for every source file. So the four columns of the table stand for the four types of operation, while each row belongs to a different database

table. This makes it easy to see which database tables are altered or used in what way per source file. By looking at the number of database tables used in the queries, an estimation can be made about the complexity of a single query.

B. Comparing queries for inequality and duplication

When the CRUD table suspects a undesirable relation between two queries, this can be further investigated by comparing the two queries with each other. By determining the amount of equality this suspicion could be increased or decreased. If the two queries use some of the same tables but are totally unequal in every other aspect, it is highly unlikely that they do have a relation, provided that no other evidence is present about a possible relation.

Also when it can be identified that two queries are variants of a similar structure of each other, a union can be performed on the sets of dependencies, because what is known for one, is also very likely to hold for the other query. Besides this, duplicated queries are undesirable, for redundant definitions tend to be error prone when maintaining and changing those queries.

In order to predict the relation between the different queries, and to determine to what extent the same data resources or properties are used, the outcome of a comparison between two queries is the percentage of equality. So the aim is to construct a table containing the equality for each combination of two different queries.

Determining equality means more then detecting code duplication. The objective is to measure the semantical equality as much as possible. So for instance comparing the conditional parts of two queries should be done in a commutative way. For this reason two steps are performed when comparing queries. The first step is propagating the defined aliases to obtain a query reflecting the database structure, while the second step is the actual comparing itself.

The propagation of aliases used within the query involves replacing alias identifiers by that of table and column identifiers. The abstract syntax tree is transformed and converted such that alias names are replaced by the original defined qualified name while the alias nodes and constructs themselves are removed out of the tree.

```
SELECT *
FROM UserRights as rights, User u
WHERE rights.user = u.id;
```

Fragment 1. Query using aliases

This fairly straightforward transformation results in the following query, provided that the query above is given as input.

```
SELECT *
FROM UserRights, User
WHERE UserRights.user = User.id;
```

Fragment 2. Query with propagated aliases

With all alias nodes removed, the query reflects the database scheme, which simplifies the process of comparing different queries witch each other.

As a second step the `SELECT` queries are divided into five parts, knowing the `SELECT`, the `FROM`, the `WHERE`, the `GROUP BY` and the `HAVING` part. For the listings in the `SELECT` and `FROM` part, the comparison is just a matter of comparing the tables and columns used in both queries. The criteria specified in the `WHERE`, `GROUP BY` and `HAVING` parts first are split to single criterion sections. By dividing the different parts, comparison can be done with the order of the different criteria or the left and right hand sides within a criterion not influencing the result. The keywords `AND` and `OR` clearly delimiter the different individual criteria. Each criterion itself then is dividable in two parts. When encountering an `=` sign, a left hand side and right hand side can be defined. The `IN` keyword however imposes a more flexible construct by providing means to be followed by either a list of values like `lhs IN (value1, value2)` or an inner query like `lhs IN (SELECT a FROM b)`.

As a first approach the different parts of the query, such as the `SELECT`, `FROM`, `WHERE` and so on, were prioritised with the idea that some parts are more likely to contain variables than others. For instance the columns selected in the `SELECT` part say little about the equality when structure and relations are involved. However, after performing some experiments it turned out that the actual result already was achieved than, by the construction of the CRUD tables. The final approach eventually was to give the different parts a weight according to their size. The size of a query part is measured by counting the number of left hand sides and right hand sides it contains. For the query `SELECT a FROM b, c WHERE d = e`, the `SELECT` part has a size of one, while the `FROM` and `WHERE` both have the size of two.

For obtaining the percentage of equality, the different parts are compared. For instance when comparing `WHERE a = b AND c = d` with `WHERE a = b AND c = e` a 25% of inequality is registered for the WHERE part. The comparison is commutative, thus `a = b` compared to `b = a` is regarded as equal, focussing more on the semantic equality then the syntactic equality.

C. Query results establishing relations

One way of retrieving an indication of relations between queries is by looking for certain patterns appearing in the data flow. For this part of the analysis the central focus is on the query result.

By determining relations between query results, i.e.

by inspecting the usage of the query result in the host language with respect to the control flow, a relation between the data addressed by those queries also is likely to exist in some way.

When query results flow in other queries, the data obtained by the queries providing the input relate to the data obtained by the query consuming this information. Therefore in *all* cases this means a relation. As a consequence this construct means a structural inefficient approach, for the Structured Query Language is powerful enough to express the intentions.

A first basic example is the following query.

```
result = Database.Get("SELECT userId FROM User
    WHERE name = 'Alice'");

Database.Get("SELECT street FROM Address WHERE
    user_id = " & result.userId)
```

Fragment 3. Query using query result

Here, the query result of the first query is used to build the dependent query. Rather than this does imply a relation establishing a artificial foreign key by the host language, it in fact uses a normal foreign key, as could be defined by the database scheme. The actual case is an improperly use of the Structured Query Language, which results in a performance penalty. The optimised query then could be defined follows.

```
SELECT street
FROM User, Address
WHERE Address.user_id = User.userId
AND User.name = 'Alice';
```

Fragment 4. The preferred combined query

Legitimate reasons[4] for still applying this approach with separated queries are avoiding very large and complex queries by splitting them. The performance penalty then just is taken for granted.

A special case is when the related queries reference the same table. Take for instance the next two queries. The objective in this example is to retrieve the names of everybody as old as Alice.

```
SELECT age
FROM User
WHERE name = 'Alice';
```

Fragment 5. Get the age of an user

```
SELECT name
FROM User
WHERE age = ?;
```

Fragment 6. Get everyone with a certain age

With the question mark indicating a parameter referencing the result of the first query. This relation indicates an inefficient usage, and therefore the queries can be combined to the following.

```
SELECT name
FROM User
WHERE age in ( SELECT age
               FROM User
               WHERE name = 'Alice' );
```

Fragment 7. Get all users as old as Alice

This enables the DBMS to resolve the requested result in an efficient way.

D. Query results in conditional statements

An `IF` statement condition relying on the result of a query, and which is regulating the execution of another query, imposes a relation of the following form.

```
SELECT MH_MN_SEQ.nextval
INTO df$rec.ID
FROM DUAL;

IF df$rec.ID IS NULL THEN
    BEGIN

        SELECT MH_MT_HUIZ.JND_GOEDGEKEURD
        INTO df$jnd.ID
        FROM DUAL;

        IF df$jnd.ID = FALSE THEN
            df$jnd.ID := TRUE;
        END IF;

    END;
END IF;
```

Fragment 8. Suspicious data fbw

First a `SELECT` query is executed and the result is stored in `df$rec.ID`. The `IF` condition then determines if another query is to be executed or not. That second query then also is followed by an `IF` statement, basing its condition on the latter query result.

Not only is this a bad practise, the locations of the queries with respect to the data flow of the query results indicate a relation. It is the query result that 'binds' the two queries throughout the control flow.

VI. Quality analysis on related data

The aim of this section is to explain the nature of the detected relations in a general, language and implementation independent manner. This is performed by analysing dataflow and control flow, for they are able to point out locations that are more likely than other program points to contain or realize undesirable dependencies.

The difference between a relation or associating and a dependency is that the first is just a link that is present, while a dependency has far more consequences, since a dependency points out that some parts are unable to go without other parts.

A. Semantics of the SQL keyword UNION

A union takes the result of the lhs query and the rhs query, adds them, and performs a `DISTINCT` on it. This differs from the '`UNION ALL`' in that way, that the latter does not perform a distinct. Multiple rows with the same results then are allowed.

A `UNION` means that data from the left hand side query result have some same semantics as the data delivered by the right hand side query. This for instance is illustrated in the following example.

```
-- Select all events in the future
SELECT scheduled_event
FROM User
WHERE name = 'Alice'

UNION

-- Select all events in the past
SELECT event
FROM Log
WHERE user = 'Alice'
```

Fragment 9. UNION selecting data with equivalent semantics

The column `Log.user` can be seen as a artificial foreign key to the 'primary key' `User.name`. As can be seen, the usage of `UNION` instead of `UNION ALL` do not change the semantics. With respect to the relation, i.e. reference, the query semantics stay the same, for only the query result changes. Another aspect is that the `UNION` is not bound to only two queries. Any number of queries, with a minimum of two, can be used in order to combine the results of them. This than means that all of the involved queries, bounded by the `UNION`, share the same semantics with respect to the selected columns producing the query results.

When focussing on the aspect of performance, [4] states that when the several queries use the same `SELECT` and `FROM` definition, the `UNION` construct could be replaced by an OR criterion added to one of both queries, benefiting the performance.

B. Manually performed CASCADE DELETE

A common scenario with respect to foreign keys, is that when the item referenced is removed, the related information also should be removed. For example, when a `User` has a `Car` and the user is removed out of the system, the referenced car also should be removed. This is solved by the construct of a `CASCADE DELETE`. When a `CREATE TABLE` or `ALTER TABLE` defines a column as being a foreign key by using the keyword `REFERENCES`, and thus referencing some primary key, the `ON DELETE CASCADE` option can be set. This option ensures that the rows referencing some primary key are removed when that primary key itself is removed.

For our analysis, this means that `DELETE` queries themselves do not contain information about if the delete command is cascaded or not. However, if the database scheme itself is present, these deletes related to the cascade behaviour are in any case being flagged as desirable situations.

Now placing this in context, when in the scheme no foreign key is specified, patterns can be used signalling a cascade delete that is performed manually. The patterns used to indicate this undesirable situation are explained below.

The base case is a User that can have multiple Cars while a Car can only belong to one User. When the User is being removed, the Car also is.

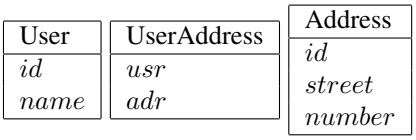
```
-- Retrieve the user to remove
SELECT id
INTO usr_id
FROM User
WHERE name = 'Alice';

-- Remove the user
DELETE FROM User
WHERE name = 'Alice';

-- Remove the cars the user owns
DELETE FROM Car
WHERE usr = usr_id
```

Fragment 10. Delete a referenced item

As can be seen, only three points can vary here. The first `DELETE` query can reference either the id already retrieved, or state the same criterion as the `SELECT` query does. The second optional variation is the number of `DELETE` queries following the `SELECT` query. Other items referencing the User, in the same way the Car does, can be removed in the same way. The last point is the order of the `DELETE` queries. They can go in any order, as long as the `SELECT` precedes them.

In order to determine what queries do belong to the chain of queries matching the pattern, scoping should be taken into account. By inspecting the different scopes, the logical sequence of the queries is determined. For PL/SQL for instance the scope changes when a `LOOP` statement, `IF` statement or block, i.e. the `BEGIN` till `END` statement, is encountered. This also holds for functions and procedures, since they start with a block statement and thus automatically change scope with the rules stated above.

In order to illustrate that this principle holds for all situations, the following more extensive example describes a User that can have multiple Addresses, while an Address can belong to multiple Users. In a database scheme this many to many relation is solved with a cross-reference table. First the database scheme is presented to clarify the structure used.



The scheme contains a user, which exists of a name, and the address that is a street name and the number of the house. The `usr` and `adr` in the cross reference table `UserAddress` respectively reference the id of the user and the id of the address.

Now the queries needed to achieve the following problem are given. When an Address is being deleted, also all related Users should be deleted out of the system, but only if they no longer have any Address left. This can be done in the following way.

```
-- Find address to delete
SELECT id
INTO adr_id
```

```
FROM Address
WHERE street = 'Great Ocen Road'
AND nr = 13;

-- Retrieve all users living at that address
FOR usr_id IN
(
   SELECT usr
   FROM UserAddress
   WHERE adr = adr_id
)
LOOP
   -- Get the nr of addresses the user has
   SELECT COUNT(*)
   INTO adr_amount
   FROM UserAddress
   WHERE usr = usr_id;

   -- Remove users only if the user
   -- no longer has any address
   IF adr_amount < 2 THEN
      DELETE FROM User
      WHERE id = usr_id
   END IF;
END LOOP;

-- Remove foreign keys
DELETE FROM UserAddress
WHERE adr = adr_id;

-- Remove address
DELETE FROM Address
WHERE id = adr_id;
```

Fragment 11. Delete a N to N relation

First the id of the address to be removed is obtained. Then for each user living on that address the number of addresses it has is requested. If the user only has that address, that is going to be removed, the user itself is removed. Now all references to the address to be removed are removed. Finally the address itself is removed.

So first three `SELECT` queries are encountered placing their results in the variables `adr_id`, `usr_id` and `adr_amount`. Then the three `DELETE` queries use the first two parameters. The third parameter only is used for control flow. All queries reference only one table.

Notice that the mechanism as described in section V-D detects the `adr_amount` in the conditional expression, for it references the query result from the `SELECT` query preceding the `IF` statement.

Generalizing these observations, the following pattern can be stated as an undesirable manual cascade delete operation that should have been performed by the DBMS itself. The definition than is as follows:
A delete query using data from the result of a select query, followed or preceded by one or more other delete queries (that not necessarily use data from a select query), implies an operation performed by the host language fulfilling the semantics of a cascade delete.

VII. Query quality analysis

When analysing queries, the focus can lie on both the structure of the query itself, as well as the use of its result by the surrounding application code. The difference between a well structured and a ill structured query can lay in the matter of complexity, and can be determined by applying some defined metrics.

After gathering all kinds of information by performing the analyses described in the previous chapters, this information is used to derive an indication of the quality and the level of maturity of the incorporated SQL statements. The other aspect is the identification of the places where undesirable constructs are used. Analysis is performed over the analyses results in order to make those results concrete and directly usable. By assigning scores for the different indications of misuses to the queries, which in the final result actually are visualized as query locations, a prioritised list is formed pointing out the locations most likely to contain opportunities for improvement.

The following sections address the measurements performed for each encountered query. Some of them use the results of the analyses performed in the previous sections.

A. Number of result variables

This measurement counts the number of result variables that are used to store the query result in, by making use of results of section V-A. An example is shown below.

```
SELECT name, street, phone
INTO :name, :street, :phone
FROM User
WHERE id = :uid
```

Fragment 12. Query using result variables

This select query contains three result variables, i.e. the `:name`, `:street` and `:phone`. And, as will be explained in the next section, it contains one input variable, viz. the `:uid`.

In order to show that this metric is not always equal to the number of select items, i.e. the columns selected for the query result, take for instance the following PL/SQL example.

```
CURSOR cur
IS
   SELECT id, name
   FROM   User;

FOR rec IN cur
   LOOP
-- et cetera
```

Fragment 13. Cursor as result variable

Now there is only one result variable, viz. the `cur`, while `rec.id` and `rec.name` indeed still result in the number of select items. But the query result now initially is stored centralized, probably decreasing complexity.

B. Number of parameters

This straightforward metric counts the number of variables used in the query and its subqueries. Two ways

of counting can be identified.

The first one is just by counting the number of holes in the query, allowing duplicate variables to be counted. For the example below this measurement then gives two for the `:gen` variable.

```
WHERE User.gender = :gen
OR Groups.gender = :gen
```

Fragment 14. Query using a variable multiple times

The second approach only counts the different unique variables used. Thus resulting in one for the example above.

The constructed framework applies the first approach. This is because the number of variable places in a query tells something about its complexity. The more parts that are variable, the less understandable the semantics get.

C. The context of the variables

Variables in a query and its subqueries can be categorized by the location, i.e. context, it is used in. A condition, i.e. in the `WHERE` part, for instance is allowed to contain variables, but when the table queried is variable, an undesirable pattern is indicated, for this construct is error prone, increases complexity and is harder to test.

One operation performed, is collecting the `WHERE` clauses that have the possibility to insert a variable value within an equation, i.e. both the lhs and the rhs are allowed to contain variables. In that case the column referenced or value filtered can differ each time the query is constructed. When variables used in the `WHERE` clause evaluate to one value only, this not is seen as an undesirable construct, but rather as a setting. The database scheme finally is needed to determine from the collected values if they imply a value to filter or a (table and) column reference. When no scheme is present, the default assumption is the filtered value, because this is the most often occurring situation. Values used by the query to filter the result, e.g. `WHERE name = :uname`, are common and therefore marked as desirable. This pattern is shown in the following example.

```
orderId = "VPO.ORDERID";

if(enabled)
   col = "I.ISPNAME";
else
   col = "I.USRNAME";

Set rsService =
      Database.Get( "SELECT I.ISPID, VPO.
          SERVICEID " _
               & "FROM VPORDERS VPO, ISP I " _
               & "WHERE " & col & " = VPO.ISP "_
               & "AND " & orderId & " = -1" )
```

Fragment 15. A query with variety in the table reference

While the `col` can have multiple values, the `orderId` column referred to in the left hand side, always is evaluated to one single value. In order to resolve the value of `col` as a column reference, aliasing is taken into account. The database scheme then is inspected for the presence of '`ISP.ISPNAME`', resulting in a match for the column reference. When tables or columns are referenced dynamically, i.e. when they are able to have a different value each time the query is constructed, this is added to the measurement result. The `col` in this example clearly does match this pattern.

The following contexts are distinct in increasing order of being harmful.

- Variable criterion - `WHERE` clause
- Variable result column - select item
- Variable additional criterion - `HAVING` clause
- Variable grouping - `GROUP BY` cause
- Variable table - `FROM` clause

Variables used in the `WHERE` clause is common practise. The column queried for the result however normally is fixed. The same holds for the `HAVING` and `GROUP BY`. Finally varying the table queried is undesirable for this indicates a lack of structure in database design.

A measurement result then for instance could be the following table, showing two query locations.

Variables	in `WHERE`	in select items
mwxf.pks@646	3	0
dump.out@1368	1	1

This example however still does not tell which query is worse. This remains a human effort, because it is difficult to specify how many variables in an expected context is equally wrong as one variable in a harmful context. Are 10 or 100 variables in a `WHERE` clause less, equally or more harmful than one variable in the `FROM` clause?

The advice here is to inspect the top five of every context manually, in order to be able to say something about the quality of the usage of queries within the application.

D. Equality compared to the other queries

As explained in section V-B, for each query the equality with respect to all the other queries is taken. For each pair of two queries, the equality is expressed by a number between 0% and 100%.

Most measurements involve counting the collected artefacts. For instance the number of result variables and the amount of queries it depends on, i.e. the number of used query results of other queries, result in a concrete number. But for the amount of equality compared to the other queries this is not so straightforward. When two queries are 100% similar, both queries score 100. Now when they match all the other queries 0%, their equality score will be 100. This then is less than a query that has 10% equality with 15 other queries, resulting in a score of 150.

Two equality measurements now can be defined.

The first one is to take per query the highest equality percentage with respect to some other query. The queries that show the most resemblance then gain the highest score. The second measurement is to sum up all equality percentages in order to gain insight in queries that have parts that are similar to queries at many other places.

E. Number of tables used

A simple measurement for complexity is performed by counting the number of tables used by each query. The more tables used, the less understandable the query gets, because of the many joins and nesting they are forced to use.

Instead of inspecting the `FROM` clause, this information is gained out of the CRUD table, because this table already was constructed previously, simplifying this measurement. As explained in section V-A, a context sensitive CRUD table contains information about the different query operation types a query performs on what database tables.

F. Inner queries

A subquery is a SQL SELECT statement that is placed in the predicate of any other SQL statement. For the usage of inner queries, each level deeper implies a significantly increased complexity. This is because the usages of intermediate results tend to be less understandable and more difficult to test. With each level deeper many more states are involved. It is difficult to reconstruct all those states in test cases, limiting the test possibilities.

This increased complexity however could be taken for granted when the performance is benefiting from this construct. There are situations that the DBMS can solve the required result quicker than the host language performing it manually.

As a measurement, this is expressed as the number of inner queries contained per query location, because the amount of inner queries reflect the complexity of the query as a whole. Having three queries sequenced with the `UNION` keyword, at one level deep, is considered more harmful than one query at a depth of level two.

G. Number of queries combined by the UNION keyword

With the use of the `UNION` and `UNION ALL` keywords, the final query result is conducted by combining the results of different smaller `SELECT` queries, as explained in section VI-A. The more isolated `SELECT` queries there are combined, the less self-evident the query as a whole and its result becomes.

This measurement counts, per query, the number of used `UNION` and `UNION ALL` keywords.

H. The number of depending queries

Multiple `INSERT` or `DELETE` queries can use data obtained by one `SELECT` query as is explained in the sections V-C, V-D and VI-B. The analysis results these sections constructed, is used by this measurement of dependent queries. For the `SELECT` queries it is counted how many other queries there are, that use its query result, creating a dependency on the particular `SELECT` query.

The queries not producing a result, i.e. not retrieving information out of a database, do not have this type of dependency, because they just perform actions. Their result, i.e. side effecting, is not used directly as input for other queries.

I. The number of joins used

By using high quantities of joins, applications get more difficult to adapt. This adaptability then can be measured by counting the number of `JOIN` keywords present.

As explained in [1], the number of joins present can also be derived from the structure of the queries. Per query it can be counted then how many joins it uses to produce its result.

Another thought is that this approach partly can be substituted if less accuracy is allowed, by measuring the amount of tables referenced in the `FROM` clause. After all, the tables specified there normally are connected in some way in the criterion part of the `SELECT` query. To prove this instead of assuming it, [1] presents an approach that can be used for this purpose.

J. Conclusion

The first approach to express the quality of each query was to create a composition of all measurements in a penalty score. However, just like metrics on software applications, quality never can be expressed in a single number. Quality is measured by inspecting the different aspects separately. The same holds for the quality analysis on queries embedded in host languages. So the final way to observe quality is performed by looking at the amount of occurrences as defined by the metrics previously described.

The metrics that proved to be the most useful, i.e. reflecting the quality, are the measurements with regard to the number of variables used, the amount equality and the number of inner queries. The number of used variables and result variables address the quality of the data model and the inefficient usage of the structured query language. The equality aspect expresses maintainability, while the nesting level affects the complexity and testability.

In this stage, after constructing the measurements, it is too soon to say something about how the results

of the defined metrics are to be interpreted. Only time and experience can point out what values the different metrics are allowed to have in order to regard the software system as healthy. The focus of this paper is to introduce and question the different metrics.

VIII. Case study

The in [1] presented framework simplifies the construction of measurements targeting queries used in all kinds of different programming languages. As a proof of concept, the framework was used to perform the earlier defined measurements on the following commercial software products.

- Visual Basic 6 source of a telecom company
- PL/SQL source of a bank
- PL/SQL source of an energy supplier
- COBOL source of a bank

Here, the banks mentioned above refer to different banks.

Experiments with respect to the defined metrics, resulted for the COBOL project in the following quantities.

	amount	SEL	INS	UPD	DEL
All	496	282	65	115	34
x.COB	59	24	9	21	5
y.COB	56	14	11	25	6
z.COB	50	19	11	11	9

This first overview shows that the different queries are well centralised, because 33.3% of all queries are contained by just three files. Although this led to large files, the queries at least are not scattered throughout the many source files.

The other observation is that 56.8% of all queries are `SELECT` queries, 23.3% are `UPDATE` operations, 13.1% is `INSERT` and 6.9% are `DELETE` queries. Per file the composition of these query types hardly differ, thus showing a normal usage of the different query operation types.

file	line	res vars	vars	nesting	tbls	eq
x.COB	3485	48	1	0	1	9%
x.COB	3601	0	49	0	1	4%
x.COB	1158	23	5	0	5	15%
w.COB	1502	13	2	0	2	34%

In the COBOL project quite some queries were used, but the queries themselves proved to be simple and straightforward. No nesting of queries occurred, only a handful of queries used more than two tables, while the most resembling queries are only 34% equal. In order to achieve this simplicity, the programmers used many, uniquely named variables to handle the data throughout the application. So the work was shifted from the queries and the DBMS to the host language. This leads to moving data around quite a bit in the host language, meaning unnecessary indirection.

IX. Conclusion

This chapter discusses the results and conclusion of this, partly empirical, study. By inspecting many commercial software systems, an understanding of how to define the level of quality of embedded queries was gained. This proved to be useful in order to say something about the role the host language can play by establishing relations between data contained by a database.

For all the analysed software projects, the experiment revealed, after manual inspection, that the obtained values actually do reflect the situation, as it is present in the software application.

Each time the support for a different programming language was added to the analyses, new insights were gained. This proved to be very useful when regarding the aim to construct a general and flexible approach that could be conducted for all kinds of different programming languages.

A. Future work

As future work the current analyses should be extended with for instance the support for identifying the columns addressed by the asterisk in `SELECT` queries. The construct `SELECT *` then is be proved to require only a few columns. The analysis then is able to point out that for instance `SELECT A, B, C` would suffice. This latter `SELECT` statement then is regarded as an improved version of the first, enabling a software inspection to suggest this change, by supplying the replacements.

It is in my belief that the field of software engineering would benefit from more extensive research about this topic. This is because many applications use a database, enabling many opportunities for improvement in this area. By extensively observing and comparing the results of the different metrics, for the different programming languages, a more precise impression could be obtained about its accuracy. Besides this, a better understanding of the semantics of the different located structures could be obtained, leading to new patterns to be detected.

B. Conclusion

Finding evidence for undesirable relations, like the functionality of foreign keys established by the host language proofs to be hard. The reason why, lies in the semantics of the relations and dependencies. At first, relationships between queries and between query results are detected. Then the nature is determined, indicating if it is a true dependency or not. But to obtain the reason of the dependency tend to be unreliable, for no general pattern or characteristic ensuring the motivation can be given. Intentions can be expressed in many code constructs. A very large amount of highly specific patterns could reflect the programmer's objective. This then provides opportunities for artificial intelligence and self-learning systems, but this is out of the scope of this

paper.

Suspicious spots, however, are indicated. Regarding the detection of a lack of structure within a database, manual inspection still is needed. The analysis performed luckily shows where to look and also what to look for. This makes the process of detecting ill practises feasible, for it would not be possible to inspect thousands lines of code manually, with actually be able to understand the relations as specified by the code and database. The analysis constructed clearly shows its use, by enabling us to gain a quality indication, with minimal effort to be performed by hand.

REFERENCES

[1] Huib J. van den Brink. A framework to distil sql queries out of host languages in order to apply quality metrics, January 2007.

[2] Christian Goldberg and Stefan Brass. Detecting logical errors in sql queries. In *16th Workshop On Foundations Of Databases (2004)*. Martin-Luther-Universität Halle-Wittenberg.

[3] Christian Goldberg and Stefan Brass. Proving the safety of sql queries. In *Fifth International Conference on Quality Software (QSIC'05)*. Martin-Luther-Universität Halle-Wittenberg.

[4] Christian Goldberg and Stefan Brass. Semantic errors in sql queries: A quite complete list. In *Fourth International Conference on Quality Software (QSIC'04)*, pages 250–257. Martin-Luther-Universität Halle-Wittenberg.

[5] Christian Goldberg, Stefan Brass, and Alexander Hinneburg. Detecting semantic errors in sql queries. Technical report, Martin-Luther-Universität Halle-Wittenberg, 2003.

[6] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *26th International Conference on Software Engineering (ICSE 2004)*. University of California, Davis, ACM Press, September 2004.

[7] Jean Henrard. *Program Understanding in Database Reverse Engineering*. PhD thesis, Facultes Universitaires Notre-Dame de la Paix namur, August 2003.