UPPSALA
UNIVERSITET

UPTEC IT 14 008
Examensarbete 30 hp
Juni 2014

A Modular Framework Approach to Regression Testing of SQL

Oskar Eriksson

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A Modular Framework Approach to Regression Testing of SQL

Oskar Eriksson

Regression testing of SQL statements in database management systems is a time-consuming process that requires much developer effort. Adding and updating test cases is tedious and validating test case results cannot be done without considering database state. These problems can lead to fewer test cases being used, likely affecting test coverage and the quality of testing negatively. Therefore, it is important to automate and reduce the required workload for this kind of testing.

This work proposes a framework for minimizing the required developer effort for managing and running SQL regression tests through a modular framework design. Modularity is achieved through well-defined interfaces and makes it possible to customize large parts of the framework functionality, such as the user interface and test execution, without affecting the rest of the system. While the extensibility is the main result of the thesis work, the default component implementations offer some alleviation of developer effort compared with other existing tools. However, more advanced component implementations should be a high priority for future users, as some usability limitations still exists.

Handledare: Ruslan Fomkin
Ämnesgranskare: Tore Risch
Examinator: Lars-Åke Nordén
ISSN: 1401-5749, UPTEC IT 14 008
Tryckt av: Reprocentralen ITC

Acknowledgements

This master thesis is a result of a project conducted in collaboration with Starcounter AB, a company developing the Starcounter database management system at the time of this thesis work. The work has been carried out at Starcounter's office in Stockholm, Sweden during the spring of 2014.

Special thanks to Ruslan Fomkin for supervising the work and providing crucial guidance and support throughout the entire thesis work.

I would also like to thank my reviewer Tore Risch at Uppsala University for additional feedback and help with the thesis.

Contents

1 Introduction 1

2 Software and Database Testing 3

2.1 Testing in Database Management Systems 4

3 Problem Statement 8

3.1 Goals 8

3.2 Use Case Scenarios 9

4 Method 11

4.1 Study of Literature and Existing Tools 11

4.2 Development Methodology 12

4.3 Evaluation Methodology 12

5 Existing Approaches to DBMS Testing 13

5.1 Related Research 13

5.2 Existing Tools for DBMS and SQL Testing 15

5.3 Comparison of SQL Testing Tools 16

5.3.1 Input and Output 16

5.3.2 Data Population 18

5.3.3 Result Handling 19

5.3.4 Discussion 20

5.4 Conclusion 21

6 Proposed Prototype 23

6.1 Requirements 24

6.2 Modular Framework Components 25

6.2.1 InputHandler 27

6.2.2 SQLTestCase 28

6.2.3 TestExecutor 29

6.2.4 OutputHandler 29

6.3 Prototype Component Implementations 30

6.4 The Internal Storage 32

6.5 Using the Framework API 32

6.5.1 Running Tests 32

6.5.2 Build Server Functionality 34

6.5.3 Using Custom Components 34
6.5.4 Internal Storage API 35
7 Evaluation and Discussion 36
7.1 Architecture and Design 36
7.2 Implementation 38
8 Limitations and Future Work 41
8.1 The Prototype Components 41
8.2 Core Framework Extensibility 42
8.3 Integration with Existing Tools 43
9 Conclusion 44
Bibliography 46
Appendix A 49
A.1 Tools Comparison Questions 49
Appendix B 50
B.1 Interfaces and Abstract Classes 50
B.2 Methods 51
B.3 Test Configuration Properties 52

1 Introduction

Testing of database management systems (DBMSs) is just as important as testing of any software, but regression testing SQL statement execution in a DBMS can be problematic for several reasons. For example, the database must be in the same state at the start of each test, the state of the database must be considered when verifying the correctness of test results, and test cases may need to be executed in order [1]. This can make adding and updating SQL test cases time-consuming and the process of regression testing can require a lot of developer effort. Furthermore, existing tools for regression testing does not provide optimal user interface and are lacking extensibility. This is likely to impact the number of test cases, test coverage and the quality of tests negatively, as much work must be spent on managing test cases. Minimizing the required developer effort for managing and running regression tests is therefore a high priority.

This thesis work will attempt to solve this problem by developing a framework prototype that will reduce the required workload, and attempt to automate as much of the testing process as possible. Like many other similar tools, regression testing using this framework will be carried out by executing tests and comparing results with previously validated expected results, to decide if the test failed or passed. However, this prototype framework will be an attempt at minimizing the effort required for adding, updating and running tests as much as possible, by providing an improved user interface and robust extensibility for further improvements and customization.

This will be solved using a modular framework design, providing support to switch and update framework components without affecting the rest of the framework. Simple default implementations will be included in the framework, but more advanced functionality can be implemented through the creation of new component implementations.

The rest of the thesis is organized as following:

- **Section 2** contains a description of the purpose and methods of software testing.
- **Section 3** presents a description of the problem to be solved and the goals of the project
- **Section 4** describes the methods used to solve the problem throughout the thesis work
- **Section 5** contains a description of existing tools for DBMS testing and a detailed comparison between four of these
- **Section 6** presents the proposed solution of this work

- **Section 7** evaluates the proposed solution in regards to usability and extensibility of the design and implementation
- **Section 8** discusses the limitations of the solution and gives suggestions for future improvements
- **Section 9** contains the conclusions reached from the results of the thesis work.

2 Software and Database Testing

Testing can be an efficient way of discovering faults and other limiting characteristics of any software, and even though testing can never prove the correctness of software, it can be an effective way to minimize errors [1]. It is also the most commonly used method for validating software [2]. Sommerville [3, p. 206] clarifies this by summarizing the goals of testing into two points; to demonstrate to both developers and users that a piece of software fulfills its requirements, and to find incorrect or undesirable behavior, i.e. software faults. Furthermore, testing during the development process can help with not only finding errors or bugs, but also to make sure that the implementation stays true to the software specification and requirements during the entire course of software development [4].

Software testing can be done in a number of different ways and at different levels in the software. A few of the different testing methods are listed below [3, pp. 42, 211-221]:

- Unit testing – Verifies the functionality of specific parts of the code, usually individual functions or methods. Tests are run in isolation from the rest of the system, so faults should be easily identifiable in the component under test. Unit tests can also form a sort of executable design specification that will always be up to date if tests are run regularly. [5]
- Component testing – Verifies that the interfaces of components, modules and composite components of a system behaves according to the specification. This type of testing can find faults that are not detectable in unit testing, but arises as components are combined and used together, because of incorrect interactions.
- System testing – Verifies the functionality and behavior of a complete system, integrating all components. Component compatibility and interactions through interfaces are tested. System testing overlaps with component testing as both are focused at testing interfaces, but system testing exercises the entire system, not subsets of the system components. System tests should validate the functionality of system behavior emerging when all components are used together.
- Acceptance testing – Verifies that a complete system meet the specifications and fulfills the requirements of the users. This type of tests may reveal problems in the requirements specification. For example, some needs of the users may have been neglected in the specification, or the performance of the software may not be acceptable. This type of testing often uses real data supplied by the customer, not simulated test data.

All of these testing methods can be used to perform regression testing. Regression tests are performed by running a set of test cases, for example unit tests, after a change has been made to a system. If the test cases were successfully executed before the change was made, then by passing the regression tests the developer can be confident that no functionality under test has been broken as a result of the modification. If the regression test cases cover all the functionality of a system, this will make sure that a change has not introduced new bugs, re-introduced old bugs (regressed), or changed the behavior of the software in unexpected ways [3, p. 223]. Regression testing is an important part of developing any type of software since modifications to a system or subsystem, such as updates, bug fixes, or enhancements are unavoidable [6].

There are many useful techniques and tools that can be used for regression testing. Many unit testing frameworks provide automation that helps carry out regression testing of unit tests. So called xUnit-based frameworks are often used for this, especially in the extreme programming development methodology [3, pp. 65-66]. Examples of xUnit-based frameworks include JUnit [7], a popular Java unit test framework, and the NUnit [8] framework for .NET. These are just two of many similar tools meant to simplify unit testing. They allow the developer to easily create a number of unit test cases, which can then be run as a test suite when needed, effectively serving as regression tests.

Regression testing can be performed at different levels of testing (unit, component, system, etc.) and JUnit and NUnit are just two examples of tools that can be useful, as many other tools exist for different testing levels. However, most of the testing techniques used are designed for traditional imperative or object-oriented languages [1] and are not suited for testing database management systems or applications.

A popular approach to automate testing is by running tests on a build server [9] [10]. A build server can provide a stable, isolated environment for coordinating distributed development of code (avoiding local machine differences when building solutions), as well as providing automated regression tests. As such, it helps with alleviating some of the workload of testing, by automating the execution of test runs.

2.1 Testing in Database Management Systems

Testing is important for any type of software, including database management systems [11]. Today's DMBSs consist of many software components and layers, and they are increasingly complex. Therefore,

testing is increasingly important for DBMS development [12] [13]. Additionally, the increasing number of device utilizing Internet-based systems has led to increased focus on DBMSs performance, illustrating that testing must not only examine the correctness of the behavior, but also test the amount of concurrent work load that will be handled by a DBMS [14]. However, that area of testing is beyond the scope of this thesis work.

Another aspect of DBMS testing involves verifying the correctness of executed SQL statements. The SQL language provides an interface to all the DBMS functionality [15]. It can be very useful in testing, as SQL statements are relatively simple and short, providing simplicity, while still providing the functionality of complex queries and data manipulation statements. The complexity is related to the fact that a single statement not only requests the work (retrieving some results, or updating the database data), but also defines how results are structured. For example, different queries may request the same data entries, but with different properties, sorting, filters, etc. As such, SQL execution involves many different system components [16, p. 529], and since changes to a component can propagate through the system, regression testing SQL execution is especially important.

One of the difficulties in SQL testing lies in the fact that the state of the database will greatly influence the result of a test. As such, the database state must be controlled at the start of every test run. Additionally, if a test case alters the state of the database in any way, it can affect the results of all later test cases. In this case, some kind of ordering must be enforced to keep the results consistent. This is one of the reasons why the unit testing tools mentioned above do not work for regression testing of SQL statements, as they assume that test cases can be executed in any order without changes to the result. This problem is not limited to testing of DBMSs, but also applies to DBMS application testing [12] [1].

A common way of regression testing SQL functionality is to issue queries to a test database and compare the results returned by the DBMS with some expected results, often generated beforehand [17] [13]. Depending on the type of SQL statements used for testing, the state of the database may also need to be inspected to deduce test results. As an example, a successful execution of an UPDATE statement does not prove that a value was actually updated. Knowing the expected results beforehand is another non-trivial issue with DBMS testing today, but there are a few different approaches to achieve this.

One approach is to execute the queries on other vendors' DBMSs and compare the resulting output and database states (of course the initial state of the databases must be the same). This can be very hard in practice, as only SQL functionality that exists on all DBMSs used can be tested (i.e. no new SQL features

or vendor-specific SQL dialect constructions) [18] [13], and there may also be difficulties with the actual comparison of the results as in Slutz [19], where character string handling and numeric type coercion proved problematic. However, the Sqllogictest (SLT) test harness [20] uses this approach and works by comparing results between different DBMSs. Although, the usefulness of SLT for SQL regression testing is limited, as discussed on the official website:

> "In order to maximize the portability of scripts across database engines, it is suggested that test scripts stick to the basic CREATE TABLE syntax. Use only a few common data types such as:
>
> - INTEGER
> - VARCHAR(30)
> - REAL
>
> Remember, the purpose of Sqllogictest is to validate the logic behind the evaluation of SQL statements, not the ability to handle extreme values. So keep content in a reasonable range: small integers, short strings, and floating point numbers that use only the most significant bits of and a 32-bit IEEE float." [20]

Another approach, which will be used in this work, is the use of a so-called baseline tool. Such tools take as input a set of test cases, execute them in a DBMS, and then store the results. The results are then validated by manual inspection and if accepted they will be used as expected results and automatically be compared with results in coming test rounds. These results are called the baseline [17] [21, p. 120]. This is a common approach to SQL regression testing, used by many DBMS developers today [22] [23]. However, if the database is large, it can be non-trivial to validate test case results by inspection as these can be very large as well. For example, manually checking that every result row of a test query is correct when the result set has several hundred rows is time-consuming.

The expected results used in a baseline tool can be serialized and stored in a number of ways to allow for quick comparisons. One alternative used by Slutz [19] is to calculate a checksum over all values of a returned result. In this way, a regression test run will calculate a checksum from test run results and compare it to the expected checksum. The main drawback of this approach is that a checksum mismatch only proves the existence of an error, not which values of the result are incorrect [17]. Also, as checksums can only represent a finite range of values, two matching checksums cannot be guaranteed to correspond to the same results. Another option is to store the results in a way so that the contents of

the result set is preserved and can be used to deduce the exact values that changed, with the drawback of slower comparisons. Examples include XML-files [17] or string representations in simple text files [22].

Baseline tools can be quick and easy to use when new functionality is to be tested, but in general, baseline tool test cases require a lot of maintenance as development progresses [17]. Other drawbacks include the already mentioned fact that the database must be in exactly the same state and contain exactly the same data for every test run in order for the baseline to be valid.

Researchers have attempted to develop tools that can be used to automate the process of regression testing DBMS by automatically populate databases, generate SQL test queries, and verify results. This research is presented along with other existing tools in Section 5.

3 Problem Statement

A problem with testing SQL in DBMSs lies in the fact that the state of the database must be considered when deducing testing outcomes, as the state will determine the test results and may need to be controlled. Of course, almost all applications contain some kind of state, so this problem is not limited to DBMS testing. It is however especially hard to avoid state modification in SQL testing, since many SQL statements perform some kind of state modification. [12]

Regression testing also requires knowing expected results of all test cases beforehand in order to verify results. In the case of regression testing SQL statements in DBMSs, not only are expected results for every test case needed, but the data in the database must always be the same at the start of every test run, otherwise the expected results will not stay consistent. Furthermore, the results of query statements are sets and differences in result set sorting can be correct or incorrect depending on the test case statement, making the comparison with expected results even more complicated.

These considerations lead to significant developer effort being required for adding and updating test cases. This is problematic as it can lead to new test queries rarely being added, and in turn lead to some DBMS features not being thoroughly tested, which could become a serious issue. Therefore it is vital that adding and updating tests can be done efficiently and that as much of the testing process as possible is automated.

The proposed solution of this work is to reduce the workload of managing and running regression tests in DBMSs to a minimum by developing a new baseline test framework prototype. The framework must provide SQL test developers with a fast and easy user interface for adding, updating and running regression tests of SQL test cases. It should also automate the process of generating expected results as much as possible, and provide extensibility for further customization, improvements and enhancements.

3.1 Goals

In order to create a robust framework prototype for solving the problem it will be necessary to first research the current state of DBMS testing and evaluate existing tools used to regression test SQL. This will help to identify requirements and suggestions for the prototype framework solution. Several questions must also be considered for the design of the prototype:

- What existing solutions for regression testing of SQL exist, and how do they work?

- How can the inspection and establishment of the expected results be simplified in practice?
- How can the number of required steps for running regression tests and managing test suites be reduced?
- How can test cases be simple to create and update while still robust enough to support verification of more than just SQL statement results?
- In general, how can the usage of this kind of regression testing tool be simplified?

Knowledge gained from this study will then be used to design and implement the framework prototype, with goals of high usability and minimal required developer effort when adding, updating and running test cases. Finally, an evaluation of the proposed prototype with regards to extensibility and required developer effort will determine how well the problem has been solved, and identify fields of further improvements.

3.2 Use Case Scenarios

Several different use case scenarios should also be supported by the framework prototype, in order to make it a robust testing tool and simplify the process of regression testing:

- **Build server functionality** – A build server can function as an isolated environment for building solutions, but can also provide a good testing environment. The framework should support build server specific functionality to help automate testing, by providing two different modes of execution; one for running on a local machine and another for running on a build server. The behavior of the framework should be appropriate for the type of machine on which it is run, which can be specified through a user-configurable setting. Running regression tests on build servers can help automate the testing process, and supporting it is therefore a high priority.
- **Adding new test cases** – Adding new test cases should be simple and fast in order to minimize required developer effort. A user should be able to get generated results for new tests without having to wait for a regression test run of all existing test cases to complete. An example of a solution is to ignore already existing test cases when new test cases are run. Then only new tests will be executed and results will be generated.
- **Debugging mode** – The framework prototype should support a mode for debugging purposes, providing additional debugging tool functionality for failed test cases, and allowing users to cancel test runs when failed tests are encountered.

- **Different result comparison methods** – Test case results should be able to be verified using different comparison methods. This should be supported through an extensible design, so that the verification methods can be changed in future work. The prototype should, by default, provide comparison of results formatted as readable strings or results stored as checksums.

4 Method

The methods used in this thesis project were chosen to allow for efficiency and adaptability of evolving requirements and obstacles, to ensure that the outcome of the thesis work would be relevant and allow for a detailed analysis of the work.

4.1 Study of Literature and Existing Tools

To gain knowledge about the nature of DBMS testing, the first phase of the thesis work consisted of researching scientific literature on the subject, as well as examining existing tools used today. This provided a basis for the thesis work to be built upon.

The literature study consisted of mostly scientific articles from online literature databases, as no suitable books describing DBMS testing was found. SCOPUS, IEEE Xplore and Google Scholar were used as primary literature databases for finding literature. Search strings consisted mostly of combinations of “DBMS”, “SQL”, “regression”, “testing”, and “framework” and through references in relevant literature additional resources were found.

While the literature study helped establish a foundation and get a basic understanding about the state of DBMS testing in general, literature regarding the use of regression testing tools for SQL was very sparse. The study of existing tools and frameworks complemented the literature study by giving more specific hands-on details about regression testing solutions, while also providing insights on different alternatives for a solution in this work.

As a previous framework used for regression testing of SQL already existed at Starcounter, it was included in the evaluation of existing tools, as it would also give further insight of any needs and requirements of Starcounter. Three other publicly available tools were found by examining testing methods and tools used by other DBMS vendors (MySQL, SQLite, and PostgreSQL). A set of questions and design aspects was established to be used in the comparison, to illustrate similarities and different approaches for minimizing developer effort and ease of use. These questions are listed in full in Appendix A. Knowledge gained from this can later be used, in conjunction with the problem description and discussions with the supervisor, to establish a set of requirements for the prototype framework.

4.2 Development Methodology

The framework prototype design will be derived from the knowledge gained from the literature and tools study, as well as from discussions with the supervisor. The framework prototype will be developed in .NET using the Visual Studio 2012 IDE.

Since the development phase will be short (about 6 weeks), no specific well-established software development methodology will be used, as it is likely to result in added overhead. Instead, a kind of agile approach is going to be used, where only simple features of limited functionality are designed and implemented initially, and later these will be refactored and improved upon as the understanding of the problem grows, until they contain all the necessary functionality required. This should be a good fit for the project, as requirements are expected to evolve throughout the design and development phases.

Regular meetings and discussions with the supervisor will also help with the development phase, as problems that emerge during implementation should be able to be dealt with quickly, and any uncertainties regarding usage of the prototype solution can be discussed immediately.

4.3 Evaluation Methodology

The developed framework prototype will be evaluated with regards to three main aspects; usability, extensibility and similarities or differences to other frameworks and tools. By going through the requirement specification many aspects of usability and extensibility will be examined. Furthermore, the results of the existing tools comparison should prove useful for quick and easy comparisons with the prototype.

The usability aspect will be further analyzed through ordinary usage of the framework prototype, to identify enhancements and find fields of further improvement. Questions like how the required developer effort for adding, updating, running and managing test suites has been minimized or can be further improved will be given extra focus, and also be discussed with the supervisor and future user. The modularity of components and extensibility of the framework will also be considered of extra importance, and evaluated through discussions and source code examination.

5 Existing Approaches to DBMS Testing

This section contains an overview of some of the existing tools supporting regression testing of SQL today, but also tools for DBMS testing in general. Proposed solutions in related research are first introduced, followed by a description of other existing tools as well as a detailed comparison of a subset of these.

5.1 Related Research

Most research done in the field of DBMS testing focus on automatic generation of test queries and automatic test database population using either randomness or more complex techniques to yield more relevant test data or queries. Very little research focusing on test result verification exists, although Khalek et al. [13] presents an example of this.

Haftmann et al. [12] have proposed a framework to execute regression testing of DBMS applications more efficiently, by running regression tests in parallel over several machines as well as ordering test cases to minimize the number of database resets needed to keep test case results consistent. This work is motivated by the huge amounts of time and resources that are spent on testing in the industry. Another approach to minimize costs of regression testing is shown by Wong et al. [6], where a technique to minimize the number of regression tests needed to find faults is presented. Here the sheer number of test cases is regarded as the main problem. This work is not related specifically to DBMS testing, but regression testing in general. Also related to this is the work by Bowman [4] where an investigation of the possibility of using mutation testing to evaluate the adequacy of DBMS test suites is made, making it possible to optimize test suites based on this evaluation.

Chays et al. [1] have created a prototype implementation for automatic data population of databases to be used for DBMS application testing, given a database schema and example values. Their focus has been automatic population of meaningful data that still conforms to the constraints posed on the database schema. This is related to the work of Lo et al. [24], where a framework including a database generator called QAGen is presented. QAGen is a “Query-Aware” database generator that takes not only the database schema, but also the test queries as input. The framework also includes tools to help automate test case construction and execution.

As mentioned above, research has also been made in the field of query generation. Khalek and Khurshid [2] present a framework that generates valid SQL test queries, with regard to the constraints of the schema. Queries are generated based on only the schema as input and should be easy to use to test any DBMS. Similar work has been presented by Bruno et al. [25] where queries are automatically generated using specified constraints, and by Slutz [19] where huge amounts of SQL queries are randomly generated very quickly. A database schema is the only input to the tool presented in [19], just as in [2]. However, both these tools generate no erroneous queries, which should also be used when testing a DBMS.

There are also tools that generate queries based on grammars. The MySQL Random Query Generator [26] tool is a popular framework that can generate queries based on a specified grammar. However, supplying a grammar that can generate queries with good test coverage is not trivial and a large drawback of this tool. [2]

While automatic data population and query generation are important aspects of automating testing of SQL, utilizing these tools are beyond the scope of this work, but may prove useful in complementing the proposed framework prototype in future work.

The process of automatically verifying the correctness of test results is an important and much less researched area, but still a non-trivial problem. Very few commercial or academic DBMS testing generators can automatically generate expected results of tests [13]. Slutz [19] handles this by executing all generated queries on several different vendors' DBMSs and comparing the results. Of course, the previously mentioned drawbacks such as only being able to compare common features apply (see Section 2.2). Chordia et al. [17] describe four different approaches that were used to verify queries when testing Entity SQL. Among these is the baseline tool approach mentioned in Section 2.2 (and the chosen approach for this thesis work). Binnig et al. [18] takes a new approach to the problem of result verification by using a methodology where a query is used to generate an expected result, and then a database instance is populated with data, so that if the query is correctly executed, the expected result is returned.

Khalek et al. [13] present in their work a tool for both automatic query-aware database generation as well as automatic test oracle generator. A test oracle is a mechanism used to determine the result of a test by examining output [27, p. 468]. The data population and automatic result verification can be done

by only taking a database schema and an SQL query as inputs. Combined with other work in test query generation, this could provide an almost entirely automated test tool for DBMSs.

5.2 Existing Tools for DBMS and SQL Testing

Even though regression testing of DBMSs requires state management, there are plenty of xUnit-based tools like TSQLUnit [28] and PLUTO [29] that provide functionality to unit test DBMS application code. In TSQLUnit every test case is implemented as stored procedures with "test" prefixed in the name. It handles database state changes by executing every test case in a transaction. After the test case has been run the transaction is rolled back to remove any resulting side-effects from the test case execution. PLUTO functions in a similar way but also provides more options for setup and teardown phases for each test.

In the case of pure DBMS testing (not DBMS application testing), many DBMS vendors use the baseline approach previously discussed for verifying the correctness of SQL execution. A few examples of such tools are the Starcounter SQL framework, MySQL test framework [22], PostgreSQL test suite [23], and Sqllogictest [20] test harness:

- The Starcounter framework is used to regression test SELECT statements in the Starcounter DBMS. It is written in .NET and uses the Starcounter DBMS .NET integration to populate a database, execute queries and verify the results. It provides the functionality of not only comparing expected and actual results, but also exceptions, execution plans, and more.
- The MySQL test framework consists of a set of test cases and programs meant to verify the functionality of MySQL Server. Most of the standard test cases consist of SQL statements, but the framework supports other test language constructs as well.
- The PostgreSQL regression test consists of a set of test cases and a program for running them to verify the functionality of the PostgreSQL SQL implementation. It provides the user with the ability to run the regression test both sequentially and in parallel.
- The Sqllogictest (SLT) test harness is a program and a set of tests used to verify the correctness of execution of SQL statements by comparing results between different vendors' DBMSs. SLT is used by SQLite [30], but supports a number of different DBMSs.

In contrast to the first three tools mentioned, SLT's primary focus is not to reduce the required effort of regression testing, but to compare the correctness of SQL execution by comparing results of different DBMSs. As such, it may not, strictly speaking, be classified as a baseline tool. The fact that it is not meant to provide regression testing is reflected in the fact that the user of SLT is recommended to only use common data types like INT, REAL, and VARCHAR (see Section 2.2), to avoid problems with running the same test cases across different DBMSs. However, it is similar to baseline tools in many other aspects. Furthermore, it supports storing expected results for future usage. Combined with only testing a single DBMS, this makes SLT basically work exactly like a baseline framework, even though it is not its main purpose.

Another DBMS testing tool is the SQL Test Suite [31], developed by the National Institute of Standards and Technology (NIST), National Computing Centre Limited (NCCL), and Computer Logic R&D. It is used to verify that SQL implementations conform to the SQL standard specifications. It verifies the results of SQL statements on custom tables against predetermined results, similar to how the above baseline tools work.

5.3 Comparison of SQL Testing Tools

This section contains a detailed analysis and comparison of four of the tools mentioned in the previous section; the Starcounter and MySQL frameworks, the PostgreSQL test suite, and the Sqllogictest (SLT) test harness. Differences and similarities between the tools are presented and advantages and disadvantages of different approaches are discussed. The tools have been evaluated by reading documentation, examining source code, and through actual usage and execution of the tools.

5.3.1 Input and Output

All the tools take files containing SQL statements and some additional information as input. The MySQL test framework, PostgreSQL test suite and SLT test harness generate additional files containing both the test input as well as actual results of a test run as output, to be used as expected results. The reason why the same information is put into different files is because these tools use diff utilities for result comparison, more on that in Section 5.3.3. The Starcounter framework, on the other hand, uses only one file for each set of test cases, containing both test case information provided by the user and

expected results. New versions of the file are automatically generated as output during test runs, containing actual results of the last test run. If expected results in input files are outdated or incorrect, these generated files can be used as new input files. The MySQL test framework and PostgreSQL regression test suite also has additional options for input, such as ignoring certain tests, running scripts, etc.

A difference between the Starcounter framework and the other tools is how test files are structured. The other tools use test files containing collections of SQL statements (and SQL results). In the Starcounter framework, test files contain collections of test case declarations. Each such test case has a large set of input parameters like description, statement, literal values, etc. While this allows for clear and fine-grained control of each individual test case, it makes adding new tests harder compared to the other tools, where only a SQL statement is added.

Specifying what tests to run also differs between the Starcounter framework and the other tools. Using the Starcounter framework requires a user application to execute test runs by calling framework API methods from user source code, specifying the name of every test file to run. As such, the developer can control exactly which collection of test cases are run when executing a regression test, but on the other hand, all test files must be manually added to the code in order to run all tests. The user application should also perform the data population needed, using Starcounter's .NET-integration before and after calling the framework to execute test runs. This approach allows for setup and teardown of data in-between the execution of two different test files. However, this makes it impossible to run a single test from command-line, since data population must be done before executing the test. This approach also requires more effort, as a regression test application must be created.

The MySQL framework, PostgreSQL test suite, and SLT harness instead use scripts to run all specified test files. As there is no need for any API calls to initiate a test run, these tools can be used from command-line, and they also support running of single test files through command-line arguments. This is possible because data population is supported within the test files (see Section 5.3.2). The initial task of setting up a regression test suite should therefore be simple.

In addition to the generated files discussed above, the Starcounter framework, MySQL framework, and PostgreSQL test suite will write any failing test cases, including the actual and expected results, to files at the end of a test run. This simplifies the process of updating outdated expected results, as the absence of correctly executed test results in these files will simplify the inspection of the actual results.

5.3.2 Data Population

Regression testing using baseline tools requires the database to be in the exact same state whenever a regression test run is started, as data values must be the same in order for the previously stored generated expected results to be valid. This can be particularly difficult when using the SLT harness for testing of several DBMSs, as the state of databases must be synchronized. Of course, it is still possible to regression test using a single DBMS in SLT. This essentially makes SLT a baseline tool and removes this issue.

While much effort has gone into tools for automatically database population, as mentioned in Section 5.1, the tools compared here, have no such features. As such, test developers have to manually populate the databases with meaningful data values. These tools provide two different approaches to database population.

The MySQL test framework, PostgreSQL test suite, and SLT harness support data population statements inside test files. The MySQL test framework also provides support for dedicated data population files. In these files, SQL statements are used to populate a database before, in-between, or after test cases are run. The Starcounter framework does not support data setup or teardown inside test runs, and the task of data population is put on the user application calling the framework API. However, Starcounter's .NET-integration makes it possible to populate databases by just declaring database classes and instantiating objects of those classes, making data population very simple while also removing any need for population inside the test files. Since no data population code is in the test files, just like in the MySQL framework, this has the potential of creating more readable test files. A drawback of the Starcounter approach of data population is that setup and teardown (removing test specific data) of data can only be done before and after the execution of all tests in a test file.

The Starcounter approach for populating large databases is simpler and easier extendable than using the other tools. Instead of large collections of SQL INSERT INTO statements, small amounts of .NET for-loops instantiating database classes can be used to insert data, making population code both shorter and more easily updated. However, integration with other automatic data population tools will probably be much simpler when using the other tools, as generated population statements can simply be put in dedicated population files or in front of test cases in test files.

5.3.3 Result Handling

All four tools use simple text files as input, containing test cases and expected results. However, two different comparison methods are used to verify the results.

The MySQL test framework, PostgreSQL test suite, and SLT harness all uses a diff utility [32] to compare the contents of the file containing the expected results with the actual results serialized to file during a test run. The diff utility is used to compare the contents of two files and will output any differences between them [32]. As such, it can also be used to find which specific test cases failed. As described in Section 5.3.1 the file containing the expected results also contains all the test case statements from the input test file, making verification a simple matter of serializing all SQL statements and results of a test run to file, and then comparing the files using a diff utility.

SLT also supports the other approach, which is also used by the Starcounter framework. They perform comparisons of results during test result serialization, as a part of the test run. This means that failures can be handled and reported by the tool itself, in contrast to a diff utility handling this afterwards, and makes it easier to customize how the information is presented to the user.

In addition, the SLT harness supports hashing of results if the size of the results surpasses a specified hash threshold. This will avoid the necessity to compare every single value in the results by hashing all values in a result to a single value. This single value can then be compared to verify the correctness of the result. The drawback is of course the same as in [19] where Slutz use a similar approach to verify results. If the expected and actual hash values differ there is no way to get information about which values in the result set caused the mismatch.

Another aspect that must be considered when comparing the results of test queries is the sorting of the result set. If a test result has multiple rows, the sorting of the rows must be consistent with the sorting of the expected results in order to verify the result. The Starcounter SQL framework, MySQL test framework and SLT test harness all handle this by having the user either explicitly use the ORDER BY clause in test queries, or by signaling the tools to sort the results of a test case after execution. None of the tools handle this automatically. The PostgreSQL test suite seems to have a very relaxed approach to this problem. As the same queries are executed in the same data by the same DBMS, they assume that the result ordering will stay consistent across platforms without the use of ORDER BY clauses. They also argue that row ordering of results is another aspect of testing that should be verified, and that more types of query plans will be exercised when not using ORDER BY [33]. However, internal changes to the

DBMS can affect the ordering and execution plans in this situation, which can be a significant drawback in regression testing, as expected results will become outdated. Of course, it is still possible to test queries without using ORDER BY or other sorting to test row ordering in the other tools as well.

When testing SQL queries all of the four tools verify query results and error messages. The Starcounter framework also verifies execution plans. This is simple using the Starcounter .NET-integration, as the Starcounter DBMS returns an object containing this information when executing a query. The MySQL framework, PostgreSQL test suite, and SLT harness support this only if the DBMS under test provides statements returning this information (for example, MySQL has the EXPLAIN statement, which returns an execution plan of a given query), as these tools can only verify information that is output as statement results by the DBMS.

Execution plans can be useful for verifying that indices are used correctly and allows for more specified and detailed query execution verification, making tests more useful. But if the DBMS under test does not support any statement returning this kind of information, or if even more aspects of execution are to be verified, extending the verification functionality of either of these tools is hard. None of the tools provide any special support for customizing result verification, and implementing such changes would likely require rewriting or adding large amounts of code in the tools. The same applies to the aspect of automating the result ordering mentioned above. In the case of SELECT queries, the use of ORDER BY can be easily determined by examining the test case statement and the tools should be able to (by default) automatically sort queries without ORDER BY clauses. Again, no special extensibility is provided, and code rewriting is needed in all four tools to accomplish this. However, this issue should not require as much code rewriting as changing the verification functionality, at least in the Starcounter framework, where minor updates to the class reading input should be sufficient.

5.3.4 Discussion

All the tools in this comparison require users to manually manage text files for test input and to copy files or file contents containing generated results after inspection, to use as expected results for tests. No automated process or graphical user interface aids this task. The reason is most likely that since the most common action is simply running the test suite, launching from a command-line interface or similar is more convenient than using a GUI. For test case management however, a GUI can be helpful for grouping and formatting test cases. A problem with using simple text files as input is that the ability

to group test cases is limited, and files can become difficult to manage as the number of test cases grows. Additionally, checking for test coverage is hard and none of the tools provide any additional support for this.

Another interesting aspect is that some of the tools require data population to be handled inside the test files, making them larger and less structured. While the Starcounter and MySQL frameworks provide other approaches to make test files more structured and easy to inspect, the above issues of test coverage checking and grouping is still a problem.

For the actual comparison and evaluation of test run results, two different methods has been encountered and examined. Results can either be serialized to file and verified using a diff utility or verified within the tool itself during a test run. A diff utility is simple to use for this, but having the tool handle the comparison gives more control over result verification and test run reporting. This approach is more natural for a framework designed for the Starcounter DBMS as the .NET-integration will most likely be used to execute queries.

While all four tools provide the possibility to verify statement results, but also more detailed information like execution plans through special statements, none of the tools provide any good way of extending the functionality of result verification. In fact, the extensibility of many aspects of functionality is lacking in all four tools. The MySQL framework is probably the most robust of these tools and provides rich functionality with many different features, but it too is lacking in support for extending the framework functionality.

5.4 Conclusion

While several tools exist for regression testing of SQL in DBMSs today, the required developer effort is still high, making testing a time-consuming process. Furthermore, the tools used by popular DBMS vendors in the industry does not use very efficient user interfaces, especially lacking support for managing test suites and checking test coverage of test cases. The extensibility aspect is also lacking in the tools examined in this thesis.

The modular framework approach of this thesis work should alleviate these problems, by supporting custom implementation of framework components. Since the framework will focus on executing test

cases and not perform any automatic data population or query generation, integration of some of the tools in Section 5.1 may be a perfect fit for future work.

6 Proposed Prototype

The large amount of required developer effort is a big problem in the field of database testing and minimizing it is one of the goals with this work, but defining a concrete idea of what approach to user interaction is the best is difficult. As many different aspects influence the optimal user interaction method (test developer preferences, changing requirements based on different phases of testing, etc.) many different approaches needs to be considered and evaluated. A modular framework could provide the possibility to do this by supporting different kinds of user interface implementations (GUI based, text file based, etc.), and allow users to define their own implementations. It was therefore decided early in the design process to focus on a modular framework approach.

Modularity is defined as having logically related code grouped into separate independent components with low coupling. Each such component should function as a subsystem in itself and have a single, well-defined purpose. Dependencies between components should be few (components should be loosely coupled) and each components should have a well-defined interface. This leads to component implementation changes having minimal effect on other components, allowing for easy modifications of parts of a system. [34, pp. 197-199] [35, pp. 15-18]

Another reason for the focus on a modular design was that it can provide an extensible design, making it possible to add support additional types of SQL statements as test cases in the future. Additionally, changing framework requirements is to be expected, as prototype development is part of a larger iterative process.

As such, the focus of the project shifted from creating a framework that would be as simple as possible to use, to creating a modular framework that would be able to support a wide range of requirements, including great usability. This approach would allow any user to create custom components to provide different user interface implementations, customized test case execution, handling of different test cases types, and more.

This approach enabled the project to concentrate on creating a good framework core structure with only simple component implementations, while leaving more advanced component implementations to future users.

6.1 Requirements

The main purpose of the framework prototype is to provide an easy way for test developers to manage and run regression tests of SQL statements in a DBMS. As such, one of the most important requirements is to minimize the required effort of updating and adding new test cases, to alleviate the workload of test developers and make sure that tests are often added and always kept up to date.

In this work, a test case is defined as a single SQL statement with some additional information. Examples of additional parameters include test case descriptions, which should be supported in order to describe the purpose of each test case and keeping track of what is tested by a test case, how it is tested, etc. Test case parameters should also be used to support parameterized statements, where literal values are extracted from a statement but specified in a separate test case parameter. Test cases should also be able to be grouped, in order to simplify structuring of test cases and checking for test coverage without having to search through all test cases.

Executing a regression test run of all test cases should also be as simple as possible, to encourage frequent test runs. Additionally, the test developer should be notified of any failed test cases and investigating and debugging failed test cases should be simple and well-supported. Functionality of running test cases in parallel should also be supported as it will result in faster (and maybe more frequent) test runs, but also exercise parallel SQL execution in the DBMS.

Finally, test cases should be able to be run on different data in different kinds of scenarios to allow for testing of very specific datasets if needed, or for example, testing query execution with or without indices, etc. This could also be used to serialize query results to checksums when running on a data set that returns very large results, while serializing to readable strings when running on other data sets. The developer should also be able to verify different aspects of statement execution, not only query results. Therefore, the framework should provide the possibility of verifying, for example, errors and query execution plans.

These requirements are general for any SQL testing tool and contain all the requirements posed by Starcounter as well, so no Starcounter specific requirements were taken into account for the development of the framework prototype.

Three additional, less important, requirements was also formulated. They include functionality for running only a single test case or collection of test cases for debugging purpose, automatically launching

a database if none is available, and providing the option to choose between sequential or in parallel test case execution.

6.2 Modular Framework Components

The proposed framework design uses modular components defined by three interfaces and an abstract class, available to any user to implement or extend. These components fill the roles of representing test cases in a test run, reading test cases from user input, executing test cases, and writing test run output.

To execute a test run the TestRunner class is called from user source code. TestRunner acts as the core of the framework and coordinates the different components. It also provides the framework API (see Appendix B) to users. TestRunner will call the InputHandler component to read tests and instantiate objects representing test cases. Each test case object is a subclass of the SQLTestCase component and should contain all test case information read from input. The TestRunner will then invoke the TestExecutor component, which will execute test case objects and evaluate their results. As illustrated by Figure 1, the TestExecutor component calls the actual test case objects to perform class-specific execution and evaluation logic (see Section 6.2.2). Finally, the OutputHandler component will be invoked by TestRunner. It takes care of presenting test run outcome and results to the user, by examining executed test case objects.

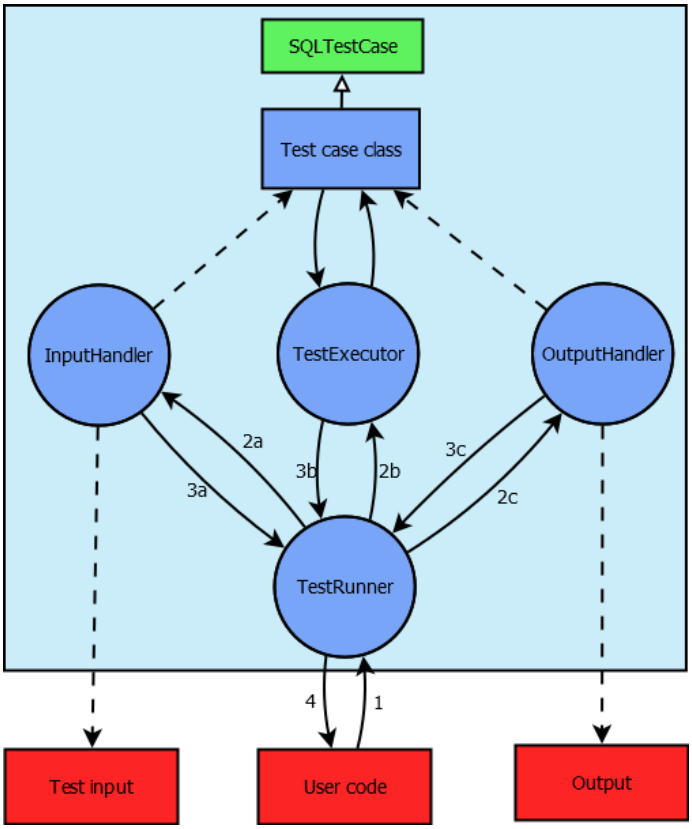

Figure 1. An overview of the program flow during a test run. A call to the TestRunner class from user code starts the test run. TestRunner then invokes the modular components to read test input, execute tests and present test run output.

The framework supports several different modes of execution. For example, a debugging mode exists where additional information about test execution and assertion failures should be used to immediately identify failed test cases. A special mode is also supported for running the framework in a build server (see Section 6.5.2). Build server specific functionality should be implemented by each modular component, as a Boolean parameter indicating when running in a build server is sent as a parameter to each component during a test run.

Additional test run options are also provided by the framework. The default behavior of the system is to run test cases in parallel, but sequential execution can also be used. The design of the TestExecutor component provides this option by having two different methods for executing test cases. Furthermore, the InputHandler, TestExecutor and OutputHandler components are by default run in three different threads for concurrent reading, execution and output of test cases (as in Figure 1), but they can also be run in the same thread, which may be useful for example when debugging. This is handled by TestRunner and is configurable through a test configuration property (see Appendix B).

As mentioned above, the main responsibility of TestRunner is to execute regression tests by coordinating and invoking the modular component implementations. In order to transmit test case objects between components, TestRunner manages two queue data structures. References to the queues are sent to the components, and allow components to send and retrieve test case objects through the queues. As seen in Figure 2, the InputHandler component will enqueue test case objects read from input, to be dequeued by the TestExecutor. In the same way, the TestExecutor will enqueue executed test case objects, which will be dequeued by the OutputHandler. This is especially useful when components are run concurrently, as a test case can flow through all the components as soon as it is read.

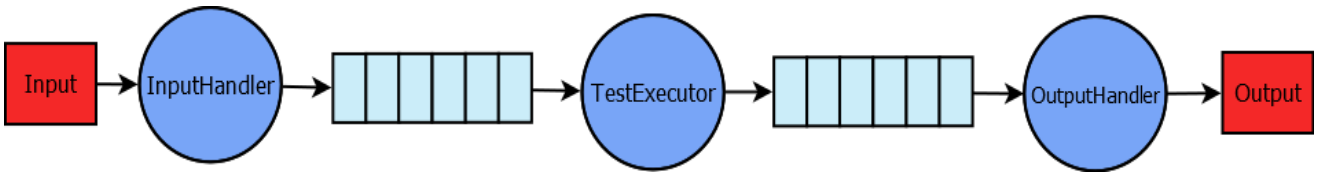

Figure 2. Test case objects flow between the framework components through shared queues

6.2.1 InputHandler

The InputHandler component constitutes part of the user interface. It reads test input and instantiates concrete test case objects populated with the read information. The IInputHandler interface defines an InputHandler component as a .NET iterator [36] that streams objects created from input through iteration. The details of how test case objects are read and created are of course up to the component implementation. The IInputHandler interface also defines a method ReadTests(). The ReadTests method takes two string parameters specifying where to read user test input from. The method serves to initialize the iterator and can, for example, be used to read and instantiate all test cases before iteration or prepare to read test cases one at a time during iteration.

To simplify inspection and test coverage checking, only descriptions, statements and other optional parameters should be needed for test cases provided by users. In order to still provide expected results, two different input sources can be used. User test input is a data source handled by the user to add and manage test cases to be run, and should only contain the information mentioned above. An internal data structure (see Section 6.4) can be used to hold additional test case information generated during test runs, especially expected test results, and work as a complement to user input when reading test

cases. The IInputHandler interface encourages making use of such an internal data store by sending an internal storage location as a parameter to the ReadTests method.

The InputHandler component serves an important part in the usability aspect of the framework. It constitutes a large part of the user interface, as it is responsible for instantiating test case objects from input. The modular design provides users with the ability to create, and even swap between, different component implementations utilizing different input methods. However, because the component is designed to read test case information from both the internal data store and user test input, it handles both loading of internal data as well as defines the user interface. Not only does this raise the complexity of the component, it also violates the rule that each modular component should have a single, well-defined purpose. This is discussed further in Section 8.1.

6.2.2 SQLTestCase

Each type of SQL statements handled by the framework should be implemented as a separate class, in order to reduce the complexity of the implementation and make full use of the modular framework design. The abstract SQLTestCase class serves as an interface to be implemented by all the test case types. It provides properties useful for all types of SQL test cases and enforces test case class specific implementations of execution and evaluation logic. It also allows all test case objects in a test run to be handled in the same way, by providing a common interface that can be used by the other components.

SQLTestCase contains many properties for representing a test case. They include description, SQL statement, expected and actual results, test outcome, execution time stamps, error messages, etc. These are general and should be useful to any type of test case, but creating subclasses of SQLTestCase allows for having additional class specific properties. SQLTestCase also defines two methods; Execute(Boolean checksumResults) for executing the SQL statement of a test case object and format results as readable strings or as checksums depending on the parameter value, and EvaluateResults() for evaluating the results held in the object to decide the outcome of the test case. These methods encapsulate the class-specific logic, allowing execution and evaluation logic to differ between test case subclasses without any changes to the framework itself. This is the main mechanism that makes it possible to add support for new types of test cases without rewriting the framework code.

6.2.3 TestExecutor

The TestExecutor component is responsible for the execution and evaluation of test cases objects and is defined by the ITestExecutor interface. The interface declares two methods that must be implemented by any TestExecutor component implementation in order to provide the option of sequential and parallel execution and evaluation of test case objects. Both methods take references to the two queue data structures described in Section 6 as parameters, and uses them to read and dequeue test case objects to execute and enqueue executed test objects to the OutputHandler. Parameters indicating the number of times to execute each test case, and whether results should be calculated to checksums are also included in the method signatures. The method for parallel execution also has a parameter specifying the maximum allowed execution time of each test case run in parallel.

Because of the encapsulation of execution and evaluation logic provided by SQLTestCase, a TestExecutor implementation only needs to call the test case objects' Execute and EvaluateResults methods to perform the actual execution and evaluation. However, a TestExecutor component should also handle things like managing execution thread(s), monitoring and handling failed execution attempts, handling of statement execution timeouts, etc. It should also log execution information and provide special functionality to support debugging when the framework is run in debug mode.

6.2.4 OutputHandler

The OutputHandler component is responsible for handling results and output of test runs, but also for implementing framework API functionality for managing any internally stored test case information. The IOutputHandler interface defines the methods to be implemented by any OutputHandler implementation. The AddOutput method is called by the framework at the end of each test run, and should handle the results of a test run, given all executed test case objects. AddOutput has string parameters specifying where user input was read from and the location of the internal data store, and also a test object queue that will be populated with all test case objects that has been executed by the TestExecutor. The user input information can be useful for allowing internal data to be organized by the structure of user test input, making it simple to find matching test cases. Other methods defined in the interface are AddTests, UpdateTests, and DeleteTests, which should be used to manage implementation specific internally stored test case information. These three methods has the same parameter signatures; a collection of test case objects, the location of the internal data store, and the user input to match.

Just like the InputHandler, the OutputHandler component suffers from not having a single, well-define purpose, as it is responsible for both handling test run output as part of the user interface and providing the internal storage API implementations. Repercussions of this are discussed in Section 8.1.

6.3 Prototype Component Implementations

The default InputHandler implementation matches test cases in user input text files (test collections) to test cases stored internally by using descriptions, statements and other user-supplied parameters. It then creates test case objects by reading test cases from user input files, and expected results and additional execution parameters from internal storage. Test cases in user input without a match in the internal storage are classified as new tests as expected results have not yet been established. Test cases that exist internally, but not in user input are deleted by calling the framework API method (see Section 6.5.4). To further improve usability and efficiency, the default behavior of the InputHandler component implementation is to only read new tests if at least one new test case is detected. This will avoid the need for a test developer to perform time-consuming regression test runs when only wanting to run new tests. This behavior can be changed however, using a special test configuration property RunAllTests (see Appendix B for a full list of test configuration properties).

The SQLQuery class is the prototype implementation of SELECT statement test cases, and a subclass of SQLTestCase. It contains a few class-specific properties, for example UsesOrderBy, a Boolean holding information about whether the SELECT query contains an ORDER BY clause. Since the use of an ORDER BY clause is only applicable to query test cases, it is appropriate as a class-specific property. The value of this property is automatically assigned by examining the query during execution. The execution and evaluation logic in SQLQuery uses a combination of user-supplied test case parameters and try-catch blocks to execute SELECT statements using different Starcounter API methods, and then store results or exceptions of execution. Evaluation is done by comparing the expected results to the actual results retrieved from execution, using string comparison or by calculating and comparing checksums. If no expected results exist, i.e. it is a new test case, no evaluation is performed.

The prototype TestExecutor implementation performs sequential execution by starting a single task to execute all test cases. Parallel execution, on the other hand, is more complex. Figure 3 shows the flow of test cases during parallel execution. The prototype TestExecutor implementation will start one execution task for each available CPU core. Each task will read test case objects to execute from the

queue structure populated by the InputHandler component. After execution, the test case objects will be sent to the second queue structure, to be read by the OutputHandler component. Each execution task also manages an array element containing a reference to the test case object currently being executed by the task. A managing thread will use this information in combination with events to monitor the progress and make sure that no task is stuck (throwing timeout exceptions or assertion failures otherwise). Both the parallel and sequential execution methods provide support to perform multiple executions of each test case in a test run.

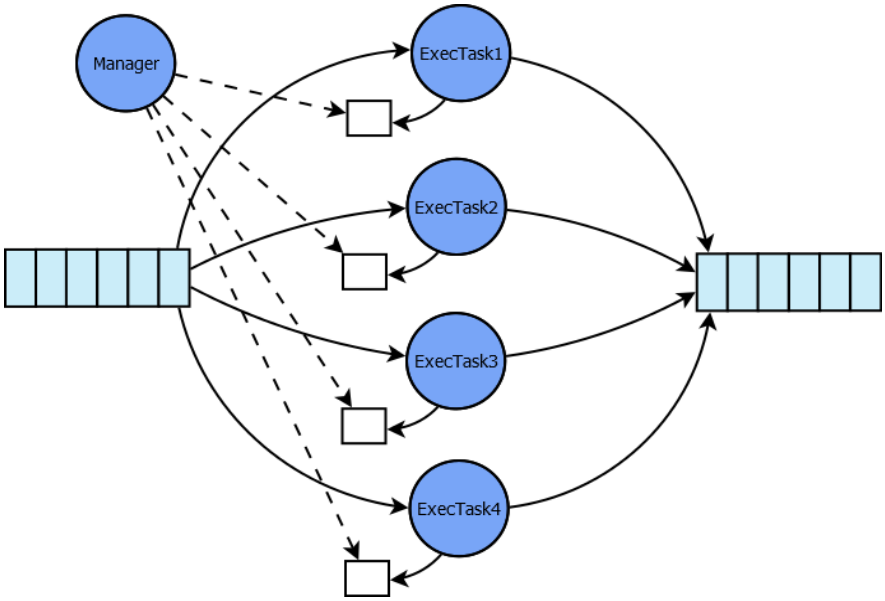

Figure 3. Parallel execution with four available CPU cores

The prototype implementation of the OutputHandler component handles test run output by printing to console and writing test cases to files. The implementation of the AddOutput method will go through all executed test cases and print a summary of the number of failed, passed and new test cases run. It will also write failed and new test cases (including test run results) to JSON formatted files, to be manually inspected by the user. Manual inspection of test results to find, or update, expected results is a time-consuming part of managing regression test cases. Using the JSON format for output allows the user to inspect test case results in a more structured way (using a JSON viewer software of choice) than compared to, for example, a plain text file. However, the simple prototype implementation of the OutputHandler component requires the use of another tool called the JSONValidator to accept expected results in new or failed test cases into the internal storage. This is discussed more in Section 8.1.

6.4 The Internal Storage

The design of the framework encourages the use of an internal data structure for keeping additional test case information that is not included in user input. The component interfaces are designed with an internal data store in mind, but component implementations are in no way forced to implement this. However, using an internal data store can help improve the usability aspects of adding and updating test cases.

The default prototype implementations of the InputHandler and OutputHandler components read and write test case information from internally stored JSON files. The JSON format is well supported by many different tools, and can be easily inspected by users through the use of any JSON viewer software. It is therefore a good choice for simple data storage in the prototype implementation. The internal files hold serialized test case objects and are organized in separate files for each input test collection and result set identifier (see Section 6.5.1 for information about result set identifiers). As such, a test case in internal storage contains all the information of the corresponding test case in user input, but also additional information retrieved from execution, like expected results. As mentioned in Section 6.2.1, this allows for simple user test input without expected results.

6.5 Using the Framework API

This section describes the framework API and how to use the framework for regression testing and managing test collections.

6.5.1 Running Tests

The TestRunner class provides the framework API methods for executing test runs and managing any internally stored test case data. Furthermore, instances of the TestRunner class acts as test configurations, as they contain a large set of properties used during test runs. These properties can be configured from user source code to setup a test configuration. This means that keeping several instances of TestRunner can be useful for executing tests with different test run configurations. For example, for some test collections it might be preferable to store results as checksums instead of as readable strings, or do multiple executions of each test case. An example of how to do this is shown in Figure 4. All test run configuration properties are listed in Appendix B.

```
TestRunner testConfig1 = new TestRunner();
TestRunner testConfig2 = new TestRunner();
testConfig1.EnableResultChecksum = true;
testConfig1.ExecutionIterations = 1;
testConfig2.EnableResultChecksum = false;
testConfig2.ExecutionIterations = 3;
```

Figure 4. User source code for setting up two test configurations with different parameter values for execution iterations and use of checksums.

The process of executing a regression test run is very straightforward. To execute a test run with the default test configuration and component implementations, a simple call to the static method RunTestsWithDefaultConfig(String filename, String path) is needed. However, instantiating a TestRunner object makes it possible for the user to customize the test configuration, as described above. Starting a test run using the configuration of a TestRunner object is done by calling the RunTests(String filename, String path) method. Both of these methods take two string arguments specifying the name and path to the test collection to run. It is also possible to specify an optional integer identifying a result set to use for the test run. Using different result set identifiers makes it possible to have multiple sets of expected results for each test collection. These can be used for running the same tests multiple times, for example with different data in the database. How the result set identifiers are used to manage result sets is entirely up to the InputHandler and OutputHandler implementations.

```
TestRunner.RunTestsWithDefaultConfig("test1.txt", "C:\tests\");
testConfig1.RunTests("test2.txt", "C:\tests\", 1);
testConfig2.RunTests("test2.txt", "C:\tests\", 2);
```

Figure 5. API calls to execute test runs. The second and third calls execute test runs using the test configurations set up in Figure 4. These test runs execute the same tests but uses result set identifiers to match them to different result sets.

While RunTests and RunTestsWithDefaultConfig invoke an InputHandler, TestExecutor and OutputHandler component to do everything from reading tests to outputting results, it is also possible to run already instantiated test objects using the TestRunner method ExecuteAndEvalute(IEnumerable<SQLTestCase> tests) method. It will not perform any input or output handling, thus only calling the TestExecutor component with the supplied test objects. This method can be useful when, for example, wanting to rerun a set of tests.

6.5.2 Build Server Functionality

The differences between executing test runs on a build server and a local machine are component implementation specific. A property in TestRunner is used to keep track of when running on a build server. The property value is automatically read from an environment variable and is then passed to the different framework components. As such, the use of the build server functionality requires that this environment variable is set, but the actual difference in functionality is up to the individual component implementations.

The default prototype components behave slightly different when running on a build server. For example, failed test cases will trigger assertion failures during test runs when running locally, but not on build server. Also, any mismatches between test cases in user input and the internal data store (i.e. additions or deletions of test cases) should result in a failure, since this is not acceptable. However, the current implementation of InputHandler simply ignores such test cases for simplicity.

6.5.3 Using Custom Components

Implementing, updating or swapping custom framework components can be done without affecting the rest of the system. Creating a custom component is done by creating a class implementing the corresponding component interface or abstract class (see Section 6.2). However, creating new test case classes will most likely require an update of the InputHandler component as well, since it is responsible for instantiating objects of the test case classes from input. But for changing or updating other components nothing else has to be done.

To use custom component implementations in a test run configuration, the user should simply pass an instance of each component implementation to use to the TestRunner object through a call to Initialize([IInputHandler input], [ITestExecutor executor], [IOutputHandler output]). Executing a test run after this call will make use of the passed component instances. If some components are not specified in the call to Initialize, or if the call is never made, the framework will automatically use the default component implementations. This means that it is possible to switch between different component implementations by calling Initialize between test runs, or by using test configurations with different component configurations. Being able to swap between implementations like this can be very useful for trying out new user interface implementations, debugging component implementations, etc.

```
IInputHandler myHandler = new MyInputHandler();
TestRunner testConfig = new TestRunner();
testConfig.Initialize(input: myHandler);
testConfig.RunTests("Test3.txt", "C:\tests\");
```

Figure 6. A test run utilizing a custom InputHandler component and default TestExecutor and OutputHandler components.

6.5.4 Internal Storage API

The use of internal storage requires a way of manipulating that data. To manage internally stored test case data, several API methods are provided by the framework. Three methods exist for adding, updating and deleting any internal test case data. This is handled by the AddTests, UpdateTests, and DeleteTests methods in the TestRunner API. All of these methods take parameters specifying where to read user input from, which result set to use, and a collection of test case objects. The actual implementation of these methods is mostly abstracted to the OutputHandler component and the implementation details are of course related to the internal data structure used, if any.

These API methods can be used both by component implementations, and by other applications. For example, the default InputHandler component implementation makes calls to DeleteTests to remove internally stored test case data corresponding to test cases that is no longer in user input. Additionally, the JSONValidator application uses the AddTests and UpdateTests to add new expected results to the internal test case storage.

7 Evaluation and Discussion

Two main aspects have been considered when analyzing the prototype framework: usability and extensibility. This section contains discussions based on acceptance testing and comparisons to other similar frameworks and tools (introduced in Section 5) with regards to usability, extensibility, and other aspects in order to evaluate the proposed prototype framework.

7.1 Architecture and Design

The modular design of the prototype framework makes it possible to swap the default component implementations with custom implementations, as described in Section 6.5.3. As components are part of test configurations, it is even possible to use different component implementations for different test collections in the same test application. This functionality is not available in any of the known related tools and frameworks.

The modular framework design also abstracts much of the complexity that would otherwise have been necessary to handle in the core framework structure to the component implementations. For example, a feature of the framework prototype is the ability to execute test cases in parallel. Through the use of the component interface, the details of how to do parallel execution are now the responsibility of the TestExecutor implementation, as any TestExecutor component is required to have such a method. Changing how parallel execution and evaluation is performed therefore does not require changing the framework code, only updating or replacing the TestExecutor implementation. It should be noted that while SELECT queries can easily be executed in parallel, data manipulation statements (which are not supported in the prototype) will require much coordination and ordering to support parallel execution, as the execution of such statements will manipulate the state of the database and the results of following statements in the test run. Once again, this is a problem that can be solved by creating a custom TestExecutor component implementation.

The internal storage is another example of a component implementation specific detail. The logic behind both the use and organization of the internal data store is part of the InputHandler and OutputHandler implementations. So while the default prototype implementations of these components use internal JSON files, other component implementations could choose to not use any internal data at all, use other files structures or formats, or store such data in a database instead.

In the same way, all test case implementation details are encapsulated in the test case classes. This allows for great extensibility. For example, changing how statements are executed does not affect any other part of the system, as the Execute method of the test case implementation classes contains this logic. It also means that adding functionality for new types of SQL statements (the default components only support SELECT statements) can be done by creating a new test case class encapsulating type specific information and execution and evaluation logic. For example, a test case class representing data manipulation statements could be created. To use the new test case class for running tests, the InputHandler component implementation would most likely need to be updated as it is responsible for instantiating the test case objects from user input. However, the important point here is that support for different kinds of test cases and execution logic can be added without changing any framework code, just like how new component implementations can be used.

Compared to the Starcounter framework, this is a substantial improvement as it also only supports SELECT statements, but adding additional SQL statement types require rewriting framework code. The other analyzed tools and frameworks (see Section 5) are architecturally very different and have no internal representation of different types of test cases as they only write results directly to file. Therefore, a direct comparison of this functionality is not applicable, as their design supports all types of SQL statement using the same execution logic.

There is also some extensibility in the design of the core framework classes. When running regression tests, queue structures are used to transfer test case objects between components. The queue class implements the ITestQueue interface, which means any class implementing the interface can be used in the framework. Switching queue implementation class will however require rewriting actual framework code, as this is not a modular component and the TestRunner class instantiates the queue objects.

In conclusion, the design of the framework prototype has displayed good extensibility and support for future additions and improvements to the system. Modularity is achieved through well-defined interfaces and makes it possible to customize large parts of the framework, to enable optimal user interaction with better support for grouping and test coverage checking, and test execution to minimize the required effort of testing. However, limitations of the design and architecture of the core framework classes still exists, and future improvements to these areas are necessary to further improve extensibility. These limitations and proposed solutions are discussed in Section 8.

7.2 Implementation

A high priority of the thesis work has been to establish a good modular design of the framework to support future usage and extensibility. As such, the component implementations have not gotten the full attention of the work. Nevertheless, the default component implementations are an attempt as reducing required developer effort in the regression testing process.

Adding and updating tests is an important part of managing regression tests. Using the default framework components this can be done without stating the expected results in user input. Manual inspection for test case result validation is hard to avoid with this kind of testing, but the process of using valid expected results has also been made more efficient by keeping them stored internally. Instead of having the user inspect large output files containing all run test cases, as in the PostgreSQL test suite and Starcounter framework, and then using valid expected results by copying or in other ways including it as part of user input, this prototype implementation will keep them separated from user input. The process of adding valid results to internal storage is not optimal, as the simple prototype implementation of the OutputHandler component requires the use of the JSONValidator application for this (see Section 8.1). However, it is fairly automated and still an improvement. Furthermore, passed test cases will never need to be inspected, as only failed and new test cases will be written as part of test run output, and the use of the JSON format for output allows for simple and structured result inspection.

Deleting test cases used in the framework prototype is even simpler than in the MySQL framework and PostgreSQL test suite. To delete a test case from a test collection in the prototype implementation, it only has to be removed from the test collection file. The default InputHandler component will then delete the corresponding internal data during the next test run. However, both the MySQL framework and PostgreSQL test suite have multiple input files that have to be updated.

Each test case in a test collection can have additional test case parameters in order to customize and specify how the test case should be executed and evaluated; for example, whether query results should be sorted by the framework or not. If omitted from user input, these parameters can be given default values or be read from the matching internally stored test case. Additionally, the default framework components can deduce the values to use for these parameters during execution, removing the need for the user to add them in most cases and further reducing required developer effort. Taking into account that the test input files used by the default components does not contain expected results, managing

test collection files becomes even simpler. Although comments are the only supported approach for grouping and structuring of test cases inside a file, the size of the files will be substantially smaller compared to when using the Starcounter framework, where expected results and additional parameters make up most of the input files. Managing test cases, as well as checking for test coverage, will therefore require less effort. Other user interface methods could of course offer better and more easily used support for grouping, since plain text files is not the optimal format for this.

Executing a test run in the prototype framework is very similar to how it is done in the Starcounter framework, even though some improvements have been made to reduce the number of necessary API calls needed to run tests. It requires an application that populates data and calls the framework API functions. However, the prototype framework supports using multiple different test configurations in the same test run by instantiating TestRunner objects, something that other existing tools do not support. This can be very useful, as it allows users to run several test collections that might otherwise have had to be separated into two test applications to enable different configurations. As described in Section 6.5.1 it is also possible to execute a test run by calling a static method, reducing the number of statements required to perform a test run even further.

Another feature is the use of result set identifiers in test runs, allowing users to execute the same test collections on several different sets of data, by keeping multiple sets of expected results. This avoids the problem where the user would have to maintain two different test collections that would only differ in expected results, and will help minimize the overall effort required for regression testing.

The three low priority requirements mentioned in Section 6.1 have only been partly implemented. The prototype framework does automatically launch a database if none is available, but this is simply a consequence of how the hosting Starcounter system works, and a very minor usability improvement. The ability to run test cases sequentially is fully supported in the default framework components, but up to the TestExecutor component implementation used. Unfortunately, as already mentioned no functionality for running a single test case or collection of test cases exists. The only way to do this is to change the user code, or comment test cases in a test collection. However, since these were low priority requirements this is regarded as an acceptable limitation and subject to improvement in future work.

The default component implementations contain many small improvements that have led to improved usability compared to the Starcounter framework, even though the main focus of the work has been on the framework core and modular design. Unfortunately, the default components also suffer from many

limitations described in Section 8. For example, the debugging functionality is limited and should be improved upon.

8 Limitations and Future Work

There exist several limitations in the prototype framework. This section contains descriptions of these limitations and presents suggestions for improvements in future work.

8.1 The Prototype Components

The debugging functionality of the prototype framework has several limiting factors. There is currently no support for running (or ignoring) specific test cases in a test collection, other than for the user to manually comment out all other test cases. As this is a limitation of the default InputHandler component, a simple solution is to use a different component implementation that will handle this, for example by using command line arguments to specify the test cases to read (or ignore).

Another debugging limitation is that test cases resulting in errors that lead to crashes (either in the framework itself or in the Starcounter system) are hard to find. This is a consequence of the behavior of the default OutputHandler component, which waits to write output until all test cases have been executed. As such, there is no way to report which statement is responsible for a crash. Once again, a different component implementation can solve this issue. By having the OutputHandler implementation write output continuously, the user should able to see which test cases pass, and be able to deduce which are responsible for the crashes. If combined with more robust error handling in the TestExecutor component, this can be handled in an even better way.

Another limitation is related to the process of validating and accepting expected results of a test case. Because the default OutputHandler implementation is non-interactive, a tool called the JSONValidator has to be used in order to add or update internally stored data, including expected results. The JSONValidator is a small command-line application that de-serializes test case objects read from the JSON files output by the OutputHandler component, and calls the framework API to add or update internally stored test case information. Note that the JSONValidator is in no way part of the framework, or any modular component. It was created in order to have a simple non-interactive OutputHandler component implementation, while still automating as much of the result validation process as possible. However, this adds additional steps to the process of establishing expected results, and is not very user friendly.

A plan for future work is to improve the user interface by having a web-based interface utilizing Starcounter's REST API. This will allow for a more interactive user experience where adding and

rerunning test cases, inspecting and accepting new expected results, etc. can all be done using the web interface. Furthermore, it can greatly improve upon the functionality to group and structure test cases, as the grouping functionality is still very limited. Organizing test cases into different test collection files and using comments within files is currently the only way to group and structure test cases. Changing the user interface of the framework can be done by changing or creating new InputHandler and OutputHandler component implementations. This plan also solves the limitation above, as the need of the JSONValidator application will be removed.

These limitations are all related to the usability of the system, and improving upon them should be a high priority in future work. While good usability was a goal of the thesis work, the fact that these can be improved upon by utilizing the modularity of the framework design (as new component implementations can solve the problems) shows that the focus on modular design was a good approach.

However, some limitations exist in the modular component designs as well. A modular component is defined as having a single, well-defined purpose [34, p. 198]. While this requirement is fulfilled by the design of some components, the InputHandler and OutputHandler components violate this rule. The InputHandler not only defines the user interface for test input, it also loads user interface independent internal data. Similarly, the OutputHandler does not only handle the output part of the user interface, it also implements the API methods for managing internal data. This is bad for several different reasons. The complexity of the components is higher, since more functionality must be provided. It also makes it hard to change the user interface and internal storage implementations independently of each other. Since the user interface is likely to be changed more often than the internal storage implementation, this is a problem. A solution is to separate the internal data handling inside the current InputHandler and OutputHandler components into a new component, dedicated for internal test data management.

8.2 Core Framework Extensibility

While the extensibility of the framework prototype is quite robust and adaptable overall, there are certain aspects within the core classes that can be improved. One example is that the SQLTestCase abstract class directly references the SQLResult class as the type of the expected and actual result properties. This means that changing what kind of results should be stored and used for result verification will require changing the SQLResult class implementation. A better approach is to have the SQLTestCase class reference an interface, ISQLResult, which is implemented by SQLResult. This would

abstract the implementation details of the class, and better support the extendibility of classes holding SQL results.

Another extensibility limitation is related to the test configuration parameters. The TestRunner class currently holds all test configuration properties and since TestRunner is not a modular component, adding new configuration properties requires changing the TestRunner code, as well as updating the interfaces of any components using the properties. Also, in the current prototype only a subset of all configuration properties are passed to each component. A way to solve both of these issues is to create a new modular component representing test configurations. Implementations of the test configuration components can be instantiated in user code and passed as an argument to TestRunner, when starting a test run. This allows users to define their own test configuration parameters by creating implementations of the component. Additionally, if other component interfaces are updated to take a test configuration instance as an argument, it will make all test configuration parameters accessible to all components.

8.3 Integration with Existing Tools

Future improvements of the framework prototype could make use of the already existing tools for DBMS testing discussed in Section 5.1. As the framework prototype is mostly an attempt at alleviating the user interaction and process of result verification (which is a less researched area), tools for query generation and data population could be a great complement to this work.

As an example, the work of Khalek and Khurshid [2] provides a way of generating queries and Chays et al. [1] have proposed a prototype for automatic data population, both using only a database schema as input. Using these tools in conjunction with the proposed framework prototype could ideally provide an almost completely automating testing approach.

9 Conclusion

The task of regression testing SQL statements in a DBMS is a time-consuming process. Most DBMS vendors use their own regression testing tools, but not much work has been put into minimizing developer effort to make the testing process more efficient. Furthermore, the tools examined in this project do not provide optimal user interfaces and are lacking extensibility. The prototype framework presented in this thesis aims to fill the gap of a simple-to-use regression testing tool for SQL statements, as well as provide a modular structure that allows users to swap between custom component implementations.

The modular design of the proposed framework is a large part of the result of the thesis work and allows for simple extensibility of most of the functionality of the system. This sets this framework apart from other related tools, where the lack of extensibility is a major limitation. The default component implementations are the other part of the result, providing support for regression testing of SELECT queries and a simple but adequate user interface. The prototype is not only a working product but also proof that the modular approach works well for this kind of tool, allowing user interface, test execution logic, etc. to be customized by users.

While the resulting prototype fulfills most of the posed requirements, better user interface and debugging functionality should be highly prioritized tasks in any future work. The usability provided by the default component implementations is a small improvement compared to related tools, but should still be further improved upon. Fortunately, the modular framework structure should make this easy to implement through better component implementations, without any need of changing actual framework code.

A secondary result of this thesis work is the comparison between four different regression testing tools used by different DBMS vendors (MySQL, PostgreSQL, SQLite, and Starcounter). The comparison conclusions were taken into account when developing the framework prototype, but also gave additional insight into the state of DBMS testing in the industry and helped form the ideas behind the default user interface. These conclusions can also prove useful for any future project related to efficiency in DBMS regression testing.

While the proposed framework prototype is far from perfect, the goals and expected results of the thesis have been met. Acceptance testing of the prototype has been positive and it should be able to

provide a robust framework structure for the future, with user-created component implementations for optimal user interaction and even more functionality.

Bibliography

[1] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos and E. J. Weyuker, "A Framework for Testing Database Applications," in *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000.

[2] S. A. Khalek and S. Khurshid, "Automated SQL Query Generation for Systematic Testing of Database Engines," in *Proceedings of the IEEE/AM International Conference on Automated Software Engineering*, 2010.

[3] I. Sommerville, Software Engineering, 9th ed., Addison-Wesley, 2010.

[4] I. T. Bowman, "Mutatis Mutandis: Evaluating DBMS Test Adequacy with Mutation Testing," in *Proceedings of the Sixth International Workshop on Testing Database Systems*, 2013.

[5] M. Elhamali and L. Giakoumakis, "Unit-testing Query Transformation Rules," in *Proceedings of the 1st International Workshop on Testing Database Systems*, 2008.

[6] W. E. Wong, J. R. Horgan, S. London and H. Agrawal, "A Study of Effective Regression Testing in Practice," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, 1997.

[7] "JUnit," [Online]. Available: http://junit.org/. [Accessed 11 May 2014].

[8] "NUnit," [Online]. Available: http://www.nunit.org/. [Accessed 14 May 2014].

[9] "Bamboo," Atlassian, [Online]. Available: https://www.atlassian.com/software/bamboo. [Accessed 23 May 2014].

[10] "Team City, Continous Integration for Everybody," JetBrains, [Online]. Available: http://www.jetbrains.com/teamcity/. [Accessed 23 May 2014].

[11] Y. Deng, P. Frankl and D. Chays, "Testing Database Transactions with AGENDA," in *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[12] F. Haftmann, D. Kossmann and E. Lo, "A framework for efficient regression tests on database applications," *The VLDB Journal,* vol. 16, no. 1, pp. 145-164, January 2007.

[13] S. A. Khalek, B. Elkarablieh, Y. O. Laleye and S. Khurshid, "Query-aware Test Generation Using a Relational Contraint Solver," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[14] J. A. Meira, E. C. de Almeida, G. Sunyé, Y. L. Traon and P. Valduriez, "Stress Testing of Transactional Database Systems," *JIDM,* vol. 4, no. 3, pp. 279-294.

[15] "Structured Query Language (SQL)," Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/ms714670(v=vs.85).aspx. [Accessed 23 May 2014].

[16] T. Padron-McCarthy and T. Risch, Databasteknik, 1st ed., Studentlitteratur AB, 2005.

[17] S. Chordia, E. Dettinger and E. Triou, "Different Query Verification Approaches used to test Entity SQL," in *Proceedings of the 1st International Workshop on Testing Database Systems*, 2008.

[18] C. Binning, D. Kossmann, E. Lo and A. Saenz-Badillos, "Automatic Result Verification for the Functional Testing of a Query Language," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, 2008.

[19] D. R. Slutz, "Massive Stochastic Testing of SQL," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, 1998.

[20] "About Sqllogictest," [Online]. Available: http://www.sqlite.org/sqllogictest/doc/trunk/about.wiki. [Accessed 15 May 2014].

[21] E. Dustin, J. Rashka and J. Paul, Automated Software Testing: Introduction, Management, and Performance, Addison-Wesley Professional, 1999, p. 120.

[22] "Introduction to the MySQL Test Framework," [Online]. Available: http://dev.mysql.com/doc/mysqltest/2.0/en/mysqltest-introduction.html. [Accessed 17 February 2014].

[23] "PostgreSQL: Documentation 7.1: Regression Tests," The PostgreSQL Global Development Group, [Online]. Available: http://www.postgresql.org/docs/7.1/static/regress.html. [Accessed 12 March 2014].

[24] E. Lo, C. Binning, D. Kossmann, M. T. Özsu and W.-K. Hon, "A Framework for Testing DBMS Features," *The VLDB Journal,* vol. 19, no. 2, pp. 203-230, April 2010.

[25] N. Bruno, S. Chaudhuri and D. Thomas, "Generating Queries with Cardinality Constraints for DBMS Testing," *IEEE Transactions on Knowledge and Data Engineering,* vol. 18, no. 12, pp. 1721-1725, December 2006.

[26] "The MySQL Random Query Generator Documentation," Oracle, [Online]. Available: https://github.com/RQG/RQG-Documentation/wiki/Category:RandomQueryGenerator. [Accessed 16 May 2014].

[27] P. Jalote, An Integrated Approach to Software Engineering, Springer, 1997.

[28] "TSQLUnit," [Online]. Available: http://sourceforge.net/apps/trac/tsqlunit/. [Accessed 17 February 2014].

[29] "pluto-test-framework," [Online]. Available: https://code.google.com/p/pluto-test-framework/. [Accessed 17 February 2014].

[30] "How SQLite Is Tested," [Online]. Available: https://www.sqlite.org/testing.html. [Accessed 12 March 2014].

[31] "Strutured Query Language Test Suite," National Institute of Standards and Technology, 3 November 2003. [Online]. Available: http://www.itl.nist.gov/div897/ctg/sql_form.htm. [Accessed 20 May 2014].

[32] "diff," The IEEE and The Open Group, 2004. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009696799/utilities/diff.html. [Accessed 20 May 2014].

[33] "PostgreSQL: Documentation: 9.3: Test Evaluation," The PostgreSQL Global Development Group, [Online]. Available: http://www.postgresql.org/docs/9.3/static/regress-evaluation.html. [Accessed 12 February 2014].

[34] K. K. Aggarwal and Y. Singh, Software Engineering, New Age International (P) Limited, 2005.

[35] P. A. Laplante, Software Engineering for Image Processing Systems, Taylor & Francis, 2003.

[36] "IEnumerable Interface," Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/system.collections.ienumerable.aspx. [Accessed 6 May 2014].

[37] "The SQL Test Framework public repository," [Online]. Available: https://github.com/Starcounter/ScSqlTestFramework. [Accessed 22 May 2014].

[38] "The MySQL Test Framework, Dealing with Output That Varies Per Test Run," [Online]. Available: http://dev.mysql.com/doc/mysqltest/2.0/en/writing-tests-output-postprocessing.html. [Accessed 20 May 2014].

Appendix A

A.1 Tools Comparison Questions

The following aspects and questions were used to evaluate each of the tools used in the comparison:

- What input method is used? How are test cases submitted? What information is included in a test case?
- Are test cases of SQL statements supported, or only queries?
- How are test runs initialized? How does a user execute a test run?
- How are test run outcomes reported / presented to the user?
- How are failed test cases reported / presented to the user?
- Is there any support for grouping of test cases? How?
- What test run parameters (if any) are configurable? (Ex: Parallel vs sequential execution)
- What test case result properties are verified? (Ex: query results, execution plans, exceptions)
- Smallest test run granularity. (Can single test cases be run or only collections of test cases?)
- How are data population and/or other setup processes supported? Is data population in-between tests possible, or only before/after?
- How is data deletion (teardown) supported?
- Implementation language of the tool
- Are there any special dependencies for using the tool? (Ex: diff utility)
- How do users add new tests / update old tests?
- Ease of adding / updating tests and populating data
- Are databases automatically launched during a test run?
- Does the tool assume a new empty database when starting a test run?
- How are results verified? What comparison method is used? Ex: String comparison or checksums
- Are query results automatically ordered by the tool? Even if an ORDER BY clauses is used?

Appendix B

This section contains a listing of the framework prototype API interfaces, methods and properties. This information is available as code documentation at the public GitHub repository for the project [37].

B.1 Interfaces and Abstract Classes

The interfaces/abstract classes below define the framework components that should be implemented or extended by any custom component implementations.

- **SQLTestCase** – An abstract class to be subclassed by any test case type to be run. It contains properties that should be useful for all test cases, such as description, SQL statement, expected and actual results, etc. It also declares interface methods where test case type specific logic for execution and result evaluation should be implemented. Test case classes extending SQLTestCase represents test cases read from user input, and will most likely hold additional user-supplied parameters.

- **IInputHandler** – An interface defining the InputHandler component. An InputHandler implementation is responsible for instantiating test cases objects (of SQLTestCase) by reading data from test input. As such, the implementation of this component will form part of the user interface, deciding how users will submit tests to be run.

- **ITestExecutor** – An interface defining the TestExecutor component. A TestExecutor implementation handles how actual executions of test case SQL statements are run in the DBMS, given instances of SQLTestCase. This component should provide methods for both sequential and parallel execution of multiple test cases.

- **IOutputHandler** – An interface defining the OutputHandler component. Implementations of this component are responsible for handling the results of test runs, as well as adding, updating and deleting any internally stored test case information. This component, together with the InputHandler component, will decide how the user interface will work.

B.2 Methods

The public methods listed below are all a part of the TestRunner class.

- ***Initialize([IInputHandler input], [ITestExecutor executor], [IOutputHandler output]) –***

  Assigns component implementation instances to the test configuration. This method should be called before executing a test run in order to use custom components.

- ***RunTests(String filename, String path, [int resultSet]) –***

  Executes a test run of the test collection at path with name filename, by invoking all framework components for input, test execution and output. An optional argument can be used to verify results against specific results sets.

- ***RunTestsWithDefaultConfig(String filename, String path, [int resultSet]) –***

  Static method that executes a test run of the test collection at path with name filename, by invoking all framework components for input, test execution and output. The test run will use the default test configuration settings and components.

- ***ExecuteAndEvaluate(IEnumerable<SQLTestCase> tests) –***

  Executes a set of test case objects by invoking the TestExecutor component of the test configuration.

- ***AddTests(String filename, String path, int resultSet, IEnumerable<SQLTestCase> tests) –***

  Adds test case information to the implementation specific internal storage corresponding to the test collection at path with name filename, by invoking the OutputHandler component of the test configuration.

- ***UpdateTests(String filename, String path, int resultSet, IEnumerable<SQLTestCase> tests) –***

  Updates test case information in the implementation specific internal storage corresponding to the test collection at path with name filename, by invoking the OutputHandler component of the test configuration.

- ***DeleteTests(String filename, String path, int resultSet, IEnumerable<SQLTestCase> tests) –***

Deletes test case information from the implementation specific internal storage corresponding to the test collection at path with name filename, by invoking the OutputHandler component of the test configuration.

B.3 Test Configuration Properties

The properties listed below are all a part of the TestRunner class and can be used to configure test runs.

- *Boolean* ***ParallelExecution –***

  True if test cases should be run in parallel, false if they should be run sequentially.

- *int* ***ExecutionIterations –***

  The number of times to execute each test case during a test run.

- *Boolean* ***EnableResultChecksum –***

  True if results should be stored as checksums in the current test configuration, false if they should be stored as readable strings.

- *int* ***ResultSet –***

  Identifier used to verify results against a specific result set. Useful for running the same test cases on different data.

- *Boolean* ***OnClient –***

  Read-only Boolean indicating if running on a local machine or on a build server.

- *int* ***MaximumExecutionTime –***

  Maximum allowed execution time for a single test case, in milliseconds.

- *int* ***PollingInterval –***

  Number of milliseconds between statuses checks of the progress of parallel execution threads.

- *Boolean* ***DedicatedExecution –***

True if all test cases should be read before starting execution of test cases, and test run output should not start until all test cases has been executed.

- *Boolean* ***RunAllTests –***

  True if all test cases in user input should be run, no matter if new tests where read or not. False if only new test cases should be run if at least one new test case is read.

- *String* ***InternalTestPath –***

  Path to use as the location of the internal test case storage.

- *IInputHandler* ***inputHandler –***

  The InputHandler component to use for this test run configuration.

- *ITestExecutor* ***testExecutor –***

  The TestExecutor component to use for this test run configuration.

- *IOutputHandler* ***outputHandler –***

  The OutputHandler component to use for this test run configuration.

In addition to these properties, nine additional properties exist that are only used when running in the .NET debug mode. These are copies of a subset of the regular properties and allow users to have a dedicated test configuration when running in debug mode. The properties are prefixed with 'Debug':

- *Boolean* ***DebugParallelExecution***
- *int* ***DebugExecutionIterations***
- *Boolean* ***DebugEnableResultChecksum***
- *Boolean* ***DebugOnClient***
- *int* ***DebugMaximumExecutionTime***
- *int* ***DebugPollingInterval***
- *Boolean* ***DebugDedicatedExecution***
- *Boolean* ***DebugRunAllTests***
- *String* ***DebugInternalTestPath***