2011

A Unit Test Approach for Database Schema Evolution

Katarina Grolinger
Western University, kgroling@uwo.ca

Miriam A M Capretz
Western University, mcapretz@uwo.ca

A Unit Test Approach for Database Schema Evolution

Katarina Grolinger, Miriam A. M. Capretz

Department of Electrical and Computer Engineering
Faculty of Engineering
The University of Western Ontario
London, ON, N6A 5B9, Canada
Telephone: (519) 661- 2111 ext. 85478
Fax: (519) 850-2436
{kgrolinger, mcapretz}@uwo.ca

ABSTRACT

Context: The constant changes in today's business requirements demand continuous database revisions. Hence, database structures, not unlike software applications, deteriorate during their lifespan and thus require refactoring in order to achieve a longer life span. Although unit tests support changes to application programs and refactoring, there is currently a lack of testing strategies for database schema evolution.

Objective: This work examines the challenges for database schema evolution and explores the possibility of using various testing strategies to assist with schema evolution. Specifically, the work proposes a novel unit test approach for the application code that accesses databases with the objective of proactively evaluating the code against the altered database.

Method: The approach was validated through the implementation of a testing framework in conjunction with a sample application and a relatively simple database schema. Although the database schema in this study was simple, it was nevertheless able to demonstrate the advantages of the proposed approach.

Results: After changes in the database schema, the proposed approach found all SELECT statements as well as the majority of other statements requiring modifications in the application code. Due to its efficiency with SELECT statements, the proposed approach is expected to be more successful with database warehouse applications where SELECT statements are dominant.

Conclusion: The unit test approach that accesses databases has proven to be successful in evaluating the application code against the evolved database. In particular, the approach is simple and straightforward to implement, which makes it easily adoptable in practice.

Keywords

Database Schema Evolution, Database Testing, Unit Testing, Mock Objects.

1. INTRODUCTION

An organization's business system is usually supported by an information system comprised of several programs and one or more databases. Today's rapid business requirement changes necessitate the evolution of databases in order to accommodate new requirements [17]. Additionally, technological changes, new operating systems, and new programming languages may require database modifications. Thus, a database evolves in a similar fashion to any other software application. Nevertheless, there are also significant differences between databases and software applications. As opposed to software applications, database queries, procedures and views are not designed to manage changes [3], hence making database evolution difficult. Because there are usually multiple applications running on a single database, database changes may require subsequent modifications to these applications [1].

This work concentrates on schema evolution, which involves a change to the database structure, rather than data evolution, which entails a modification of the data over time. Changes to the database schema can be classified into two main categories: requirement modifications and database refactoring.

The most common schema changes caused by requirement modifications, involve enabling the storage of supplementary data, including addition of attributes (columns) in an existing tables and addition of new tables or constraints [4]. Database refactoring refers to changes in the database schema that are intended

to improve its maintenance and evolution [1]. Although each refactoring step can be clearly defined, database refactoring is considered risky due to its potential for adversely affecting all applications relying on the database [3]. As a result, many companies avoid database refactoring [6] at the expense of further degrading their database schema quality. For example, even if a column requires a name change, it is often left unchanged. Similarly, if a column needs to be deleted from the table, it is often left in the table to avoid possible data loss or application problems. While this approach is not acceptable for application development [19, 20], it is often practiced with databases because of the potential dangers inherent in altering a database schema. Moreover, application development tools support refactoring [21, 22, 23] and unit tests provide a safety net for application refactoring. In contrast, databases have no tools, such as unit test frameworks, which would provide a safety net for database application developers [1]. However, the continual avoidance of database refactoring will eventually lead to poorly structured databases that will be increasingly difficult to maintain, thus challenging the development of new applications that rely on the existing database. Therefore, the proposed approach applies to both categories: requirement modifications and database refactoring.

Database schema needs to evolve in order to adapt to requirement changes and to facilitate the maintenance of databases. Consequently, applications relying on the database also need to evolve to accommodate the changed database schema. The first component of the schema evolution challenge, the change of the actual database schema, has been documented in detail [1] and supported through a variety of commercial tools [25, 26, 27, 28]. However, the second component, the process of effectively locating and changing the application code to reflect the database changes, has attracted attention in the research community [3, 4, 5], but it lacks tools or methods that are widely accepted.

This paper explores the use of existing software evolution strategies that facilitate the adaptation of the application code related to changes in its respective database schema. In particular, a unit-testing architecture that accesses databases is devised in order to identify the application code requiring modification.

The paper is organized as follows: Section 2 reviews related work identifying the main gaps in the database schema evolution and the associated application code testing. Subsequently, Section 3 describes the proposed approach for database schema evolution supported by unit tests, while the implementation of the approach is illustrated in Section 4. Finally, Section 5 illustrates the validation results, and the conclusions are presented in Section 6.

2. RELATED WORK

Ambler and Sadalage present details of how to change and refactor the database schema [1]. The aspects of the application code that require modification after a database schema change depend on the patterns of data access, which can be encapsulated using two main approaches [1]:

1) *Encapsulation through the use of views and database stored procedures*. In this case, data is only accessed through views and stored procedures, so that any change in the database schema will only affect these two components. This approach is not commonly used due to the limitations of updating views and stored procedures [1]. Moreover, this process requires different skill sets, as the stored procedures and views, rather than the application code, require modification.

2) *Encapsulation through the application code*. With this commonly used approach, data in the application is accessed through the Data Access Objects (DAO), so that changes to the database queries also affect the application code [4]. Since this is the dominant approach in modern business applications [1], as well as with legacy systems, this is the architecture on which the proposed approach will be developed.

Once the database schema is changed, applications accessing the database may need to be adapted accordingly. In particular, impact analysis seeks to identify all the code that will behave differently or that are required to behave differently as a result of the database schema changes [3, 4]. Similar to impact

analysis, our work attempts to determine the application code that requires change due to a database schema change. However, while impact analysis tools use string searches and/or flow analysis to evaluate existing code, we exploit unit tests as a tool for locating code that requires change. Several of the impact analysis methods are based merely on string searching and pattern matching; more effective methods use data flow analysis in combination with string analysis. For example, Maule et al. [4] analyze the data flow, starting from the location where the query is executed and proceeding through the calling objects. The challenge is to determine to which point in the hierarchy the flow analysis is necessary. Nevertheless, the process of analyzing many layers would be computationally expensive, while analysing fewer layers would possibly introduce a higher error rate.

Karahasanovic and Sjoberg [5] introduce visualization tools into impact analysis. These tools help to identify the impact of changes in the database schema of object-oriented applications. Specifically, these authors use a Schema Evolution Management Tool (SEMT), which receives a Java code and database files as input and subsequently identifies relationships between the packages, classes and interfaces. Once the database schema is modified, SEMT identifies the potential objects in the application that require change, and consequently, it alters the shape of those objects in the graph. However, in the case of large systems, the graph display may become cumbersome.

Another way of approaching the problem of application adaptation to the evolved database is the integration of schema evolution and data migration into development. An example of such approach is proposed by Draheim et al. [2], where the evolution process starts with the modification of the application object model, which is followed by a change in the database schema, and subsequently, the data migration. Specifically, the object model change is detected by the upgrade generator, which compares the old and the new model structures and generates the code for the database schema changes and data migration. Although tools that integrate schema evolution into development and data migration [2] have been gaining popularity in recent years, especially in the case of small to medium size systems, existing applications cannot benefit from them due to the need to integrate the approach in the initial development stage.

Furthermore, the process of adapting the application to the evolved database is an integral part of the PRISM workbench [6, 7]), which is proposed by Curino et al. The PRISM workbench is a tool for evaluating schema changes, executing changes to the database schema, rewriting the queries, adapting the applications, and migrating the data. Thus, PRISM is an ambitious tool, with the goal of automating the complete transition process from the initial assessments to the running of new applications on the transformed database. Nevertheless, it is difficult to envision how online query rewriting would work on complex data warehouse queries and how it could be implemented without significantly compromising system performance.

Nevertheless, schema evolution is risky due to the paucity of tests that target it [3, 6]. Unit testing is commonly used for non-database code, and it is proven to result in high quality code [29, 30, 31]. However, for application code accessing database, unit testing is generally criticized for two reasons: firstly, the goal of the unit tests is to test only one feature at a time, and secondly, the tests need to run quickly [8]. Although there are some unit test approaches that do access databases, these tests often sacrifice performance for quality [10].

DbUnit [9] is an extension of JUnit, which is a commonly used application testing framework. DbUnit offers functionalities that configure the database for testing purposes as well as that return the database to its original state after the tests have been completed. The test developer creates data sets, using XML or other methods, which will be used to produce the database state required for tests. Since different tests require different database states, the process of creating numerous data sets may become labour intensive. Although DbUnit configures the database appropriately for different tests, it does not relate to changes in the application code that result from schema modifications.

A promising approach is proposed by Christensen et al., who suggest a unit-test framework for database

applications [10, 11]. This framework breaks the rule of tests' independency with the objective of simplifying the writing and maintenance of unit tests so that they are less labour intensive and their execution performance is improved. Similar to JUnit, this framework uses `setUp` and `tearDown` methods. In addition, it uses a dependency hierarchy that is defined by the test developer. When a single test is initiated, the `setUp` method is executed for all classes on which the test is dependent. However, this approach suffers from the fact that each test is dependent on other tests, not only for the order of execution, but also for the data used by the tests. Thus, when one test fails, all of the other tests fail as well. This approach could possibly succeed in discovering application code that requires change due to database schema modifications. However, the process of configuring the database to a specific state for each unit test involves extensive effort in the coding of the `setUp` and `tearDown` methods. Moreover, this method fully executes queries, which may be a time consuming process.

Additionally, the testing of database applications also involves the problem of how to populate databases with meaningful data. For instance, DBMonster [12] and DataFactory [13] can assist in populating a database with large amounts of information. However, the data created with those tools is often not meaningful for functional applications testing; rather, it is mostly used for application scaling tests. Chays et al. propose the use of a test GENerator for Database Applications (AGENDA) [14, 15, 16], which generates database states based on the SQL statements used in the application and on the tester's suggestions as captured in the input files. Specifically, the tester specifies sample values for the attributes that can be partitioned into groups, each of which are expected to cause the application to behave differently.

For each test repetition, the database needs to be returned to the same state. Although DBUnit [9] will enable this functionality, the process of specifying states for each test is a burden on testers. Accordingly, Willmor and Embury propose methods that combine the AGENDA and DBUnit approaches [18]. In particular, they suggest the use of an extended SQL language to specify test cases and permitted transitions. Once the test specifications are written, this system automatically transfers the database into the required state.

We have also looked at add-ons for commercially available unit testing frameworks to evaluate their functionality in relation to database schema evolution. In particular, JUnit add-ons have been reviewed, as JUnit is the most commonly used unit testing framework [23]. JUnit add-ons are a collection of helper classes for JUnit [24] that simplify unit test writing. However, there were no JUnit add-ons that were related to database schema evolution.

There are numerous commercial tools that deal with database management. Some of these tools, including DB2 Change Management Expert [25], Oracle Change Management Pack [26] and MySQL Workbench for Schema Change [27], are offered by database vendors, while others are generic, such as Embarcadero Change Manager [28]. The main functionalities of these tools include implementing database schema changes, comparing different databases or schema versions, visualizing change prior to implementation and analyzing the impact of database schema changes on database objects, such as views and stored procedures. These commercial tools are effective for applications that encapsulate data access through the use of database views and stored procedures, since, in those applications, all data is accessed using views and stored procedures, and the application does not directly access other database objects, primarily tables. Unfortunately, the majority of applications use encapsulation through DAO objects [1]. In such cases, the impact analysis of schema changes on the application code is required and is not provided by these commercial database management tools.

For any testing method to be fully accepted in the business world, it needs to be both simple and effective. Unit testing for software applications is popular and efficient because it detects numerous problems early in the development process and it is relatively simple to implement. However, traditional unit tests have not been used to access databases [8], and consequently, they have not been used during the testing of database schema evolution.

3. SCHEMA EVOLUTION SUPPORTED BY UNIT TESTS

Currently, there is a lack of methods and tools that support schema evolution testing [6, 7]. Nevertheless, test suites facilitate the writing of unit tests for software applications, hence enabling testing during the coding phase [23]. In order to be effective, unit tests have to be used frequently and efficiently [8]. For instance, full unit tests for large systems should be performed rapidly. In order to achieve the required speed, traditional unit tests do not access the database; rather, they access mock objects [8]. As a result, these unit tests are not beneficial for database schema changes, especially since the database is not usually accessed until the application code is executed, which occurs very late in the development process.

Extensive studies have been conducted on the efficiency of unit testing and test driven development [29, 30, 31]. These studies have demonstrated that the use of such approaches and related technologies leads to an improved quality code. However, the majority of these unit tests mock access to the database. The proposed approach aims to modify commonly used unit-testing strategies to include database access while maintaining a short execution time. Consequently, the proven power of unit testing [29, 30, 31] can be harnessed for query validation on the evolved database.

3.1. SYSTEM ARCHITECTURE

The proposed architecture, depicted in Figure 1, includes unit testing that accesses the database. For every query in the application code, traditional unit tests use mocks to mimic the results that are retrieved from the database. In fact, there are two possible ways to deal with queries in the proposed architecture:

1) *No change in the database schema.* In this case, there is no need to access the database, and therefore, a traditional unit test is sufficient. Hence, in this situation, the proposed approach uses traditional mocks. As demonstrated in Figure 1, the traditional mocked result set is returned to a mock.

2) *Change in the database schema.* In this instance, a query validation for the modified database is required. However, the data manipulation language (DML) statements cannot be sent to the database in their existing form, as the process would be slow and impractical. Unit tests need to run quickly so that developers can execute them on a frequent basis. Since data warehouse queries commonly run for several minutes, or even hours, they cannot be executed within unit tests. Therefore, the query should be modified in a way that maintains validation while also decreasing the time required for execution. The Query Modifier, which is the module that deals with this modification, sends a modified query to the Executor/Validator, which executes the query and retrieves the results. If the query is modified in such a way that it does not retrieve any data, the result set is empty and the Executor/Validator determines whether or not the validation was successful. Subsequently, the result is sent back to the mock and the unit testing proceeds. Furthermore, the Executor/Validator has the additional functionality of logging detailed information into the log file in the case that the query validation failed. This functionality assists in identifying statements that require modification due to a change in the database schema.

The core of the system is the Query Modifier, which changes the query so that it is still validated when it is sent to the database while the execution time is minimized. It works in conjunction with the Executor/Validator, which is responsible for retrieving the validation results and transferring them to the mocks.

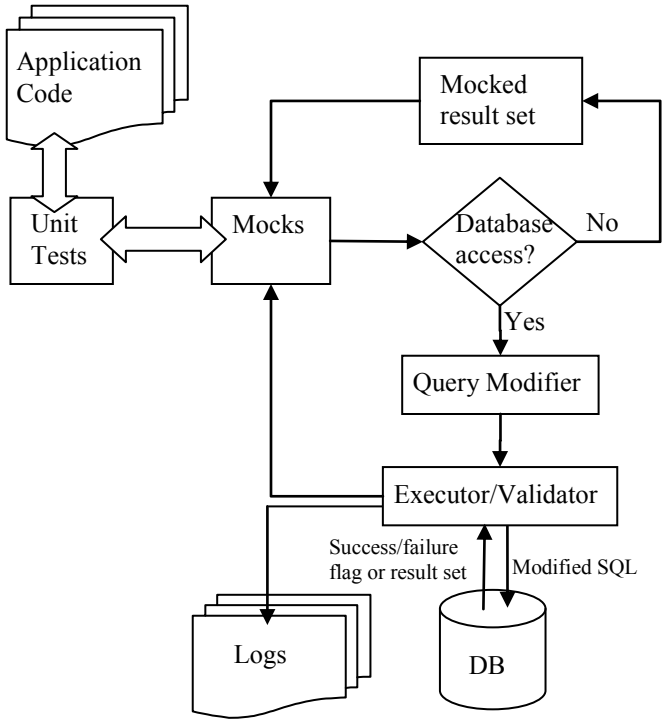

Figure 1: Proposed architecture for the unit test accessing a database

Our approach focuses on SELECT statements, which are typical DML statements for data warehouse applications, but it also manages INSERT, UPDATE and DELETE statements.

3.2. QUERY MODIFIER

When a query is sent to the database, it is processed in several steps. First, in the query parsing phase, the query elements are evaluated in comparison to the database, and the most effective execution plan is determined. Unless the query includes a significant number of tables, this step is very fast, executing in less than one second. The second step involves accessing data, and its duration depends on how much data the query accesses. This step may take long, even several hours, if the query needs to access a large amount of data. The last step involves sorting, if requested, and returning the result set, which can be slow depending on the size of the set returned to the application. If there is a problem with the query, the first step of query parsing will fail, and an error will be returned to the calling method.

Therefore, in order to evaluate a query in a database, it is not necessary to fully execute the query. Rather, only the first step, query parsing, is necessary, and the remaining steps can be avoided, since those parts decrease the execution performance. Accordingly, this work proposes the use of mocks that access the database and validate the query without fully executing it. Specifically, these mocks alter the query in order to avoid a full query execution. The modification of mocks can be performed using two approaches:

1) A query can be sent to the database solely for the verification of referenced objects, and, in this case, no result set is returned. This verification can be easily performed by appending the WHERE clause criteria that can always be evaluated as false (‘`WHERE …and 2=1`’).
2) A query can be parsed, and a few rows of the result set returned if further data processing in the unit test is required. Depending on the database management system, this can be performed by limiting the number of rows in the result set by using a feature such as `rownum < 3` for Oracle or `limit < 3` for MySQL and PostgreSQL. Although the data retrieved in this manner will be incomplete, it can still be useful depending on how the unit test will use it.

In most cases, the first approach can be used. However, in certain cases, such as during the process of retrieving data from the database when receiving values of sequence numbers, the second approach would be used.

The Query Modifier processing is demonstrated in Figure 2. If the statement is categorized as SELECT, DELETE or UPDATE, the Query Modifier proceeds to append criteria that evaluates as false. In case of UNION, MINUS or INTERSECT queries, each separate query is appended with the same additional criteria. However, an INSERT statement proceeds differently, depending on whether it is a direct INSERT statement, like the command INSERT INTO table VALUES(…), or an indirect INSERT statement, such as selecting from another table, where the command stipulates INSERT INTO table_a (SELECT … FROM table_b). Indirect INSERT statements are treated in the same way as a SELECT statement, and the SELECT part of the INSERT statement is appended with additional criteria. Conversely, direct INSERT statements do not contain a SELECT part, but they specify the values that need to be inserted. Thus, direct INSERT statements do not need to be appended; rather, they can be sent to the Executor/Validator without changes.

Furthermore, some database systems provide the option of having the database itself append the query. For instance, some databases, such as Oracle, have policies that may be set up to append queries with additional criteria [12]. In this situation, a separate database account can be used for running unit tests, and for that account, a policy could be specified to append the query with the appropriate criteria. Since this approach is less generic, as it is specific to the database vendor, our implementation appends the query from the unit test code.

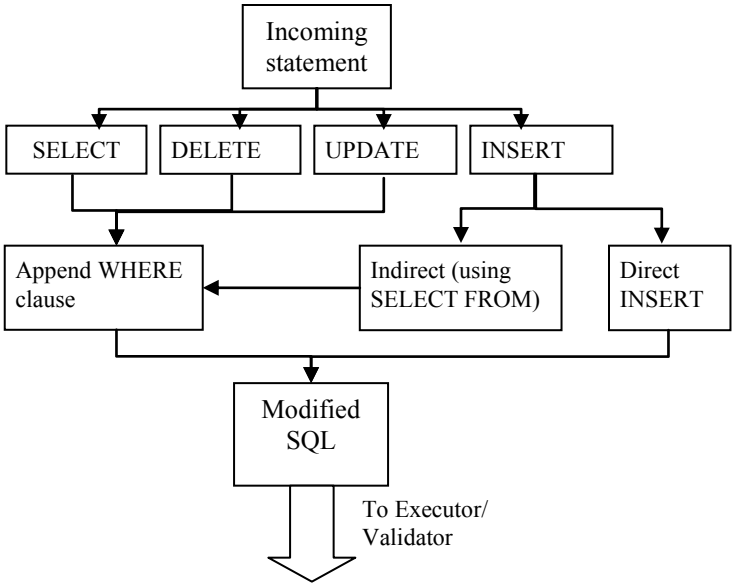

Figure 2: Query Modifier

Although the process of accessing the database during unit tests will reduce the performance of the tests, the database does not necessarily need to be accessed every time the unit test is executed. Rather, the database should only be accessed when there is a schema change and a query re-evaluation is needed. As shown in Figure 1, there is a switching component that changes traditional mocks to mocks that access a database. This latter group of mocks is not separate from the original set; the main difference between this group and traditional mocks is that this group appends a query, sends it to the database for parsing, and then transfers the result back to the original mock object.

The ability of mocks to access the database is an integral part of our proposed unit test framework and not separate from the traditional mocks. For this framework, it is not necessary to write a new set of mocks; rather, a generic query execution mechanism is integrated with mocks that access the database.

3.3. EXECUTOR/VALIDATOR

The Executor/Validator is responsible for interacting with the database. Specifically, it receives a modified query from the Query Modifier and sends it to the database for execution. Once the database execution is complete, the Executor/Validator is responsible for retrieving the results of the execution. Since the query is appended with criteria that evaluate as false, the Executor/Validator will only receive information regarding the success or failure of the statement. The result of the query, however, is sent back to the mock that originated it. Nevertheless, the result of the unit test is the result of the statement execution.

After the execution of unit tests, the developer will know which tests have failed and which ones have succeeded. However, details of the reasons for the failure are not provided. Changes to the database schema will likely cause several unit tests with database access to fail. Therefore, additional logging capabilities that assist in determining the cause of the failure and performing the required code changes are provided. Each DML statement that fails, along with the database error message, is written into the log file with the unit test method that initiated it.

While this procedure provides information about the statements that failed, it does not give details about the origin of the statement. Some statements are dynamic, and their creation may involve several methods and classes. In order to assist in tracking the creation of such statements, the call stacks for the error are also logged. Currently, this information is inserted into a text file, which is available to assist the developer with the code changes. However, we are considering the creation of a user interface that would present this information in a more user-friendly manner.

Our approach targets changes to the software application that are directly related to modifications in the database schema; therefore, it does not consider changes to stored procedures within the database. In rare cases, however, these stored procedures are tested through the use of unit test suites, such as TSQLUnit [32] or utPLSQL [33], rather than application unit test suites. In such cases, the proposed approach can be adapted to test the stored procedures.

4. IMPLEMENTATION

To validate our proposed approach, we have utilized Java 1.5 for the software application, JUnit 4 as the unit testing framework, and the database Oracle 10g. Although the implementation with other development languages, testing frameworks and databases would require an adaptation to those systems, the overall approach should remain the same.

The validation entails making various changes to the database schema and observing the success of finding statements that require modification in the software application. After testing the system performance of each schema evolution scenario, the database schema and the data are restored to their original state, thus ensuring the same initial condition for all of the tests and facilitating the comparison of the test results. In order to quickly and efficiently restore the database schema and the data, we have selected a simple database schema for the first step of the validation; the initial schema is depicted in Figure 3.

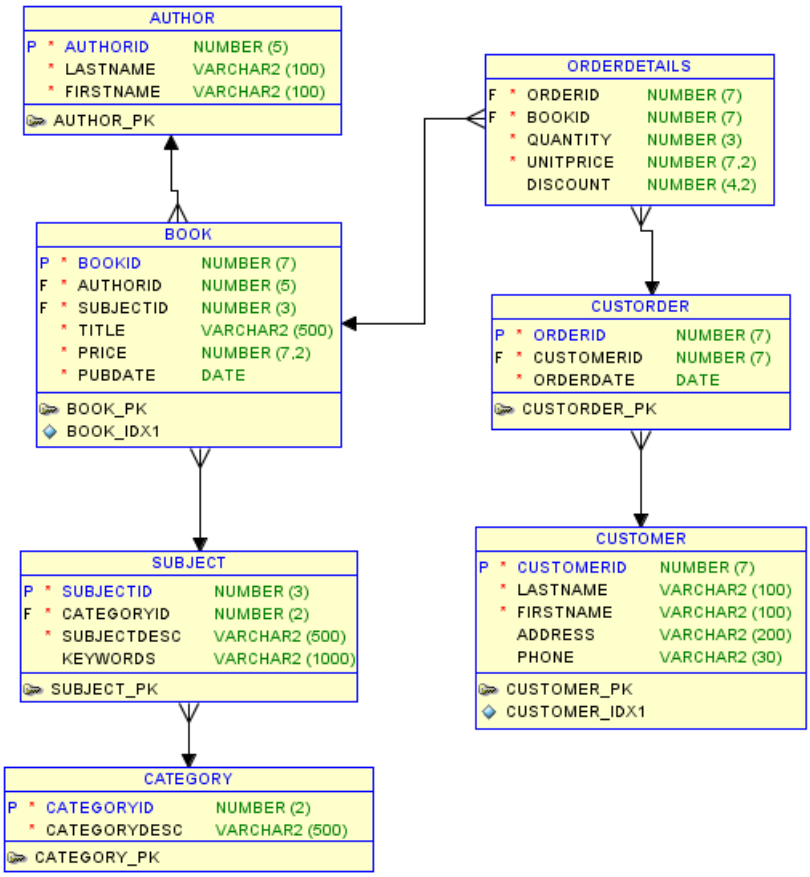

Figure 3: Database schema

4.1. IMPLEMENTATION ARCHITECTURE

In order to follow common development practices, we have utilized encapsulation through the application code, where the data is only accessed from the Data Access Objects (DAO). Therefore, all queries in the software application are sent to the database through the `SQLExecuter` class. `SQLExecuter` contains methods such as `execute()` and `executeUpdate()`, which send statements to the database, and `getResultSet()`, which retrieves the result set and returns it to the calling method.

Our implementation will be performed in two steps. The first step involves unit testing using traditional mocks without database access, and in the second step, the proposed mocks with database access are introduced. The implementation of traditional mocks at the outset will allow us the observation of the feasibility of introducing the proposed approach into a previously existing test framework.

4.1.1. TRADITIONAL MOCKS

The core implementation of our traditional mocks includes `MockResultSet` and `MockSQLExecuter` classes. Most database systems implement the `ResultSet` interface, and accordingly, our `MockResultSet` implements the `ResultSet` interface to mimic the result set returned from the database. In the `MockResultSet`, the methods that are utilized in the data access code, such as `getInt(String columnName)`, `getString (String columnName)` and `next()`, are implemented to mimic the behaviour of the database implementation of `ResutSet`; however, these methods do not have database access. An example of this implementation is shown in Figure 4.

```
public boolean next() throws SQLException {
    if (currentIterator==null){
        currentIterator = records.iterator();
    }
    if (currentIterator.hasNext()){
        currentRecord = currentIterator.next();
        recordIndex = 0;
        return true;
    }
    currentRecord = null;
    return false;
}
```

Figure 4: MockResultSet next method mimicking database ResultSet next method

Because there is no database interaction, the framework requires a way to add records into `MockResultSet`. Therefore, the method entitled `addRecord`, which is depicted in Figure 5, is called from the traditional unit test in order to set the data and mimic database interaction.

```
public class MockResultSet implements ResultSet {
    private Vector<Object[]> records = null;
    ...
    public void addRecord(Object[] aRecord){
        records.add(aRecord);
    }
```

Figure 5: Adding records to MockResultSet

Additionally, the `MockSQLExecuter` class mocks the `SQLExecuter` class, and accordingly, it overrides methods from the `SQLExecuter`. Figure 6 illustrates the `execAndGetResultSet` method from `MockSQLExecuter`. As demonstrated, this method only returns the `mockResultSet`, which was previously set using the `addRecord` method of the `MockResultSet` class.

```
private MockResultSet mockResultSet;

protected ResultSet execAndGetResultSet
(String sql, Object[] parameters) throws Exception {
    return mockResultSet;
}
```

Figure 6: Example of the MockSQLExecuter class method

An example of the unit test with traditional mocks is depicted in Figure 7. Specifically, the mocked result set is created with the desired values using `addRecord`. When the class `Book` is created, the `MockSQLExecuter` is passed in, so that the `Book` class will use the mock instead of the real `SQLExecuter`.

```
public void testGetBookSubject() throws Exception{
    rs = new MockResultSet();
    String subjectTarget= "Test Book Subject";
    rs.addRecord(new Object[]{subjectTarget});
    MockSQlExecuter mockSQLExecuter
                    = new MockSQlExecuter(rs);
    book = new Book(mockSQLExecuter);

    assertEquals(subjectTarget, book.getBookSubject(1));
}
```

Figure 7: Traditional mock

Although there are some alternative approaches to mock database access that may be simpler, they are not as generic as the presented approach.

4.1.2. MOCKS WITH DATABASE ACCESS

Depending on how traditional database mocks are implemented, the proposed approach may require different methods for implementing mocks with database access. Using a generic approach with `MockSQLExecuter` and `MockResultSet`, all of the changes required for implementing the proposed approach are within the `MockSQLExecuter` class. `MockSQLExecuter` class contains the switch that determines whether traditional mocks or mocks with database access will be used. One of the methods that have been modified to include database access for mocks is depicted in Figure 8. When using traditional mocks, the `useDBAccess` switch is set to false, and this code returns `MockResultSet`, which has been previously set in the test code. This process is the same as the one for mocks without database access. However, if the `useDBAccess` switch is on, the `MockSQLExecuter` method transfers the query to the `SQLExecuter`, which then sends the query to the database. If the query is successfully executed, the `MockResultSet` is returned to the caller test. Since we are appending criteria that always evaluate as false, the returned result set will be empty if the query is successfully executed. On the other hand, if there are validation errors on the query, the `MockSQLExecuter` will log additional information using the `logInfo(sql, String message)` method, and the unit test will fail. The `logInfo` method implements the logging of the executed statements, the database error and the call stack at the error point. The process of sending the query to the database and retrieving the result sets, along with the logging capability, represents the Executor/Validator functionality.

```
protected ResultSet execAndGetResultSet
    (String sql, Object[] parameters) throws Exception {
    final MockResultSet emptyResultSet = null;
    if (useDBAccess) {
        try {
            super.execAndGetResultSet(sql, parameters);
        } catch (Exception e) {
            logInfo(sql, e.getMessage());
            return (ResultSet) emptyResultSet;
        }
        return getResultSet();
    } else
        return getResultSet();
  }
```

Figure 8: **Core of mocks with database access (from MockSQLExecuter class)**

When `useDBAccess` is set to true, the method `execAndGetResultSet`, from the `MockSQLExecuter` class, calls the `execAndGetResultSet`, shown in Figure 8, from the parent `SQLExecuter` class `(super.execAndGetResultSet(…))`.

The method `execAndGetResultSet`, from the `SQLExecuter` class, calls the `execute` method. The `MockSQLExecuter` class overrides the `execute (String query)` method, from the `SQLExecuter` class, for the purpose of accessing the database in mocks. The process of overriding the `execute` method from the `MockSQLExecuter` class is depicted in Figure 9.

```
public boolean execute(String query) throws Exception {
    queryMocked = true;
    String modifiedQuery = appendCriteria(query);
    SQLExecuter sqlExecuter = new SQLExecuter();
    return sqlExecuter.execute(modifiedQuery);
}
```

Figure 9: Modifying the query before execution (MockSQLExecuter class)

The `execute` method of the `MockSQLExecuter` class calls the `appendCriteria(query)`, which appends the criteria evaluating as false: `WHERE … and 2=1`. Subsequently, the returned, modified query is transferred to the `SQLExecuter` for execution.

The UPDATE and DELETE statements require an implementation that is similar to that for the SELECT statements. Typical UPDATE and DELETE statements have the WHERE clause to which the criteria evaluating as false can be appended. If these statements do not have the WHERE clause, we add one containing a criteria evaluating as false, such as "2=1".

On the other hand, INSERT statements require a different procedure. Typical INSERT statements, which directly specify the values to be inserted, are not appended with additional criteria. Rather, these statements are sent to the database without modifications, and the changes are rolled back. In the case of an indirect INSERT statement that uses a SELECT statement to obtain the values (INSERT INTO tableA (SELECT column1, column2 FROM tableB)), the SELECT part is treated as a regular SELECT statement and is appended with the additional criteria.

5. VALIDATION RESULTS

The main objective of this work is not to provide a comprehensive solution that can manage all possible application issues arising from schema evolution, but to provide a generic solution that can be quickly and easily implemented and can detect most schema change problems early in the process. Complicated solutions often remain unimplemented due to the time and effort they require. Conversely, the proposed approach is easy to implement and does not require significant unit test changes in comparison with the traditional unit tests.

The validation of the proposed approach that has been carried out includes: the renaming of tables and columns, the addition and deletion of columns, the merging and splitting of tables, and changes to the column data types. Overall, while the validation was successful with SELECT statements, the success rate of the other statements depended on the specific change to the database schema. In particular, the process of changing column data types contained the most unreliable results. The results depended on the former and current data types as well as on the data used in the unit tests.

Section 5.1 reviews the proposed system behaviour with regards to different types of SQL statements. Since the total number of possible changes to the database schema is enormous, the validation analysis concentrates exclusively on the most common schema changes.

Larger schema changes are commonly made up of several simple ones. For instance, adding lookup table is composed of change(s) to original table (adding or renaming columns), creation of new table and addition of foreign key. Some complex schema changes, and their decomposition into simple ones, can be found in [1].

5.1. SELECT STATEMENTS

Data warehouse applications contain mostly SELECT statements, while online transaction processing (OLTP) applications contain a high number of all statement types, including SELECT, DELETE, INSERT and UPDATE. In hybrid applications, which are the most common types of applications, SELECT statements are dominant. Due to the dominance of SELECT statements, we evaluate the way in which various changes to the database schema influence SELECT statements as well as the success of the proposed approach in working with these statements.

In this work, all SELECT statements that required modifications as a result of database schema changes were successfully recognized. Depending on the schema change, the application logic may need to be revised, thus requiring the SELECT statements to be modified. However, application logic changes are outside of the scope of this research.

5.1.1. RENAMING TABLES, COLUMNS, VIEWS AND PROCEDURES

One of the most common and simple changes to the database schema involves renaming tables, columns,

views or stored procedures. Although the process of renaming is relatively easy, it is usually avoided due to the potential impact that it can have on the applications. For instance, consider the following SELECT statement:

```
SELECT c.lastname, info.get_orderdetails(o.orderID)
FROM customer c, v_shippedorders o
WHERE c.customerID = o.customerID and c.customerID = ?;
```

Where:
`v_shippedorders` is the view of all shipped orders that have not yet been received.
`info.get_orderdetails(o.orderID)` is the stored procedure returning the details of order `o.orderID`.

If the name of any referenced elements in the SELECT statement changes, the statement itself will need to be modified. This modification includes all referenced objects, including the `customerID` and `lastname` columns of the `customer` table, the `customerID` and `orderID` columns of the `v_shippedorders` view and the procedure `info.get_orderdetails`.

Consider a situation where the `lastname` column of the `customer` table is renamed to `name`. During the execution of the proposed unit tests with database access, the Query Modifier appends the SELECT statements before sending them to database for execution. Thus, the above SELECT statement is modified to:

```
SELECT c.lastname, info.get_orderdetails(o.orderID)
FROM customer c, v_shippedorders o
WHERE c.customerID = o.customerID and c.customerID = ? and 2=1'
```

In this case, the database engine tries to execute the modified statement and fails. Subsequently, it sends the error message back to the calling object. With Oracle as the database engine, the '`ORA-00904: LASTNAME: invalid identifier`' error is triggered. The Executor/Validator recognizes the error and sends it to the log file along with additional information about the failure, as shown in Figure 10. The log file identifies the statements that require modification as a result of the database schema change; it contains a paragraph dedicated to each identified statement. Specifically, the first line of each paragraph indicates the error that was transferred to the Executor/Validator by the database engine. The subsequent lines identify the DML statements that caused the error, and the final part of the paragraph contains the Call Stack referring to the code, in this case a Java class, from which the statement originated. For example, the portion of the log file shown in Figure 10 identifies two SELECT statements that require changes due to renaming of the `lastname` column from the `customer` table. From the Call Stack, we conclude that both statements were executed from the `Customer` class: the first statement was executed from the 26th line of the `getCustomerName` method and the second one was executed from the 37th line of `getShippedOrders`. After the statements are identified through the log file, they need to be changed to reflect the evolved database schema.

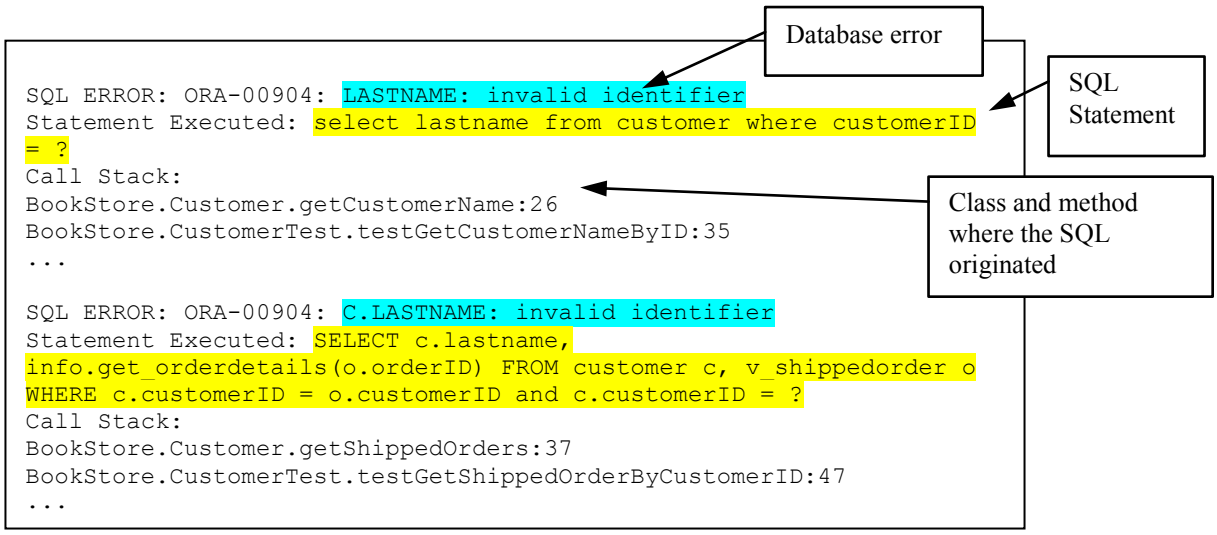

Figure 10: Log file identifying invalid SELECT statements

The proposed approach successfully identified the SELECT statements that required modification due to renaming of a column. However, the renaming of the constraints or the non-referenced objects does not affect the SELECT statement.

5.1.2 DIVIDING A TABLE INTO TWO TABLES

In addition to renaming, another common change to the database schema entails the division of a single table into two tables. As is the case with renaming, the proposed approach is also successful in this situation. With this change, not all queries referencing the divided table require modification. For example, consider the following change to the database schema:

```
CREATE TABLE phone
       (phoneid NUMBER,
       customerID NUMBER REFERENCES customer(customerid),
       phone VARCHAR(30));

INSERT INTO phone
       (SELECT ?, customerid, phone FROM customer);

ALTER TABLE customer DROP COLUMN phone;
```

Although the original `customer` table contained the `phone` column, a customer may have more than one phone number, and therefore, a new table, named `phone`, should be created. The only queries that require modification are those referencing the `phone` column in the `customer` table. Consider the following two SELECT statements:

```
SELECT firstname FROM customer WHERE customerID = ?;

SELECT phone FROM customer WHERE customerID = ?;
```

The first statement does not require a change, because it does not reference the `phone` column in the `customer` table, while the second statement needs to be modified to use a new `phone` table for obtaining the phone number. Accordingly, the proposed method correctly identifies only the second statement as requiring change. The elements referenced in the first statement did not change, and, as a result, the statement was executed successfully after the schema change. Therefore, based on the proposed method, this statement does not require modification. Figure 11 depicts the log file after the `customer` table has been divided.

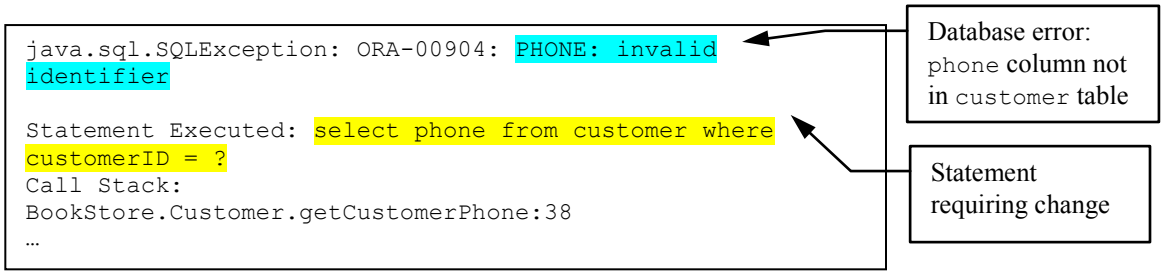

Figure 11: Log file after dividing the customer table into customer and phone tables

5.1.3. MERGING TABLES

Table merging, which is opposite to the previous change, is a less common modification. In this case, the success of the proposed approach depends on the way in which the merging is performed.

In a situation where the names of clients are in the `customer` table and the phone numbers are in the `phone` table, these two tables have to be merged into a single table because the system needs to keep only one phone number for each customer.

```
CREATE TABLE customerinfo (
       customerID NUMBER,
       lastname VARCHAR(400),
       firstname VARCHAR(400),
       address VARCHAR(400),
       phone VARCHAR2(100) );

INSERT INTO customerinfo
       (SELECT c.customerID, c.lastname, c.firstname, c.address, p.phone
       FROM customer c
       JOIN phone p on p.customerID = c.CustomerID);

DROP TABLE customer;

DROP TABLE phone;
```

In the previous example, the name for the new table, `customerinfo`, is different from the names of the tables that were merged, as those were named `customer` and `phone`. Consequently, the proposed approach successfully found all queries referencing both the `customer` and `phone` tables.

However, the situation is different if the name of the new table is the same as that of the merged table, that is, if the new table is named `customer`. The proposed approach will only find queries referencing the `phone` table, while queries referencing the `customer` table may require additional changes, depending on the application logic.

If there are constraints for additions or removals, including foreign keys, NOT NULL or check constraints, then there are no changes required for the SELECT statements.

Overall, when the database schema is changed, the proposed approach finds SELECT statements that become invalidated and hence require change.

5.2. DELETE STATEMENTS

Like the case with SELECT statements, when the changed object is referenced by a DELETE statement, the DELETE statement is successfully identified as requiring change.

A typical example of a DELETE statement is:

```
DELETE FROM custorder WHERE customerID = ? and orderdate = ?;
```

In this case, the records from the `custorder` table are deleted for the specified `customerID` and

orderdate. If the name of the table, which is custorder, or the names of the referenced columns, which are customerID and orderdate, change, the statement is properly identified as requiring modification. If the column names for other columns of the custorder table change, the statement does not require modification, and the proposed method correctly identifies the statement as not needing change.

After custorder is renamed to customerorder on the sample database, several statements appeared in the log file, a portion of which is depicted in Figure 12. In addition to correctly identifying that 'DELETE from custorder' requires change, it also recognized that the stored procedure v_shippedorder contains errors. When the v_shippedorder procedure was reviewed, it revealed that the stored procedure also referred to custorder table, which needed to be changed to customerorder.

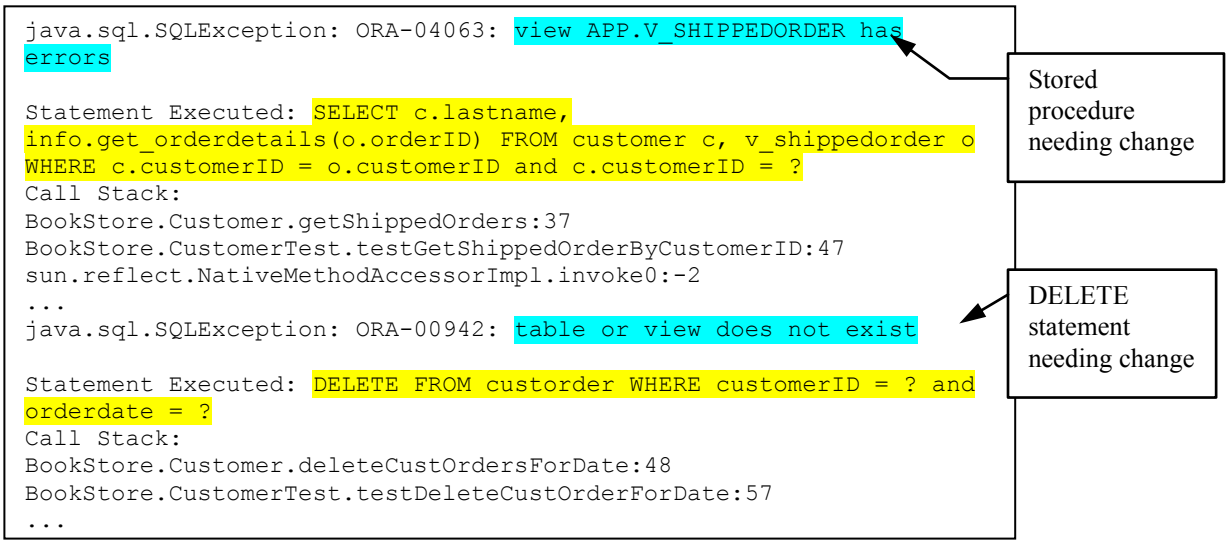

Figure 12: Log file after renaming custorder table to customerorder

Nevertheless, while this approach will verify the validity of a DELETE statement, it will not check whether or not the deletion will break any constraints. For instance, if a foreign key is added, the proposed approach will not find the DELETE or UPDATE statements that could possibly be affected by these keys. For example, if a foreign key referencing the orderID from the order table is added, the previous DELETE statement may be affected by it, depending on the data contained in the child and the parent table. In this case, the proposed approach does not identify the statement as needing change.

5.3. INSERT STATEMENTS

Typical INSERT statements do not include a WHERE element, and therefore, additional criteria cannot be appended to it. Rather, an INSERT statement is executed in its existing form and after all data changes are rolled back.

The INSERT statements are checked for the existence of referenced objects. If any of the referenced objects, including tables, columns or views, do not exist, the INSERT statements will be identified. Also, the proposed method ensures that the number of columns matches the number of supplied values for the statement.

Since the proposed approach actually inserts the data, it performs validation in regard to the constraints. For example, consider the following statement:

```
INSERT INTO customer (customerID, lastname, firstname, phone)
VALUES (?, ?, ?, ?);
```

In this INSERT statement, the customer table contains the address column. This column allows the existence of NULLs (can remain empty), therefore, this INSERT statement is deemed successful in spite

of the fact that it does not contain the `address` column. Figure 13 displays the log file after the `address` column has been changed to NOT NULL. The database error refers to the NOT NULL constraint on the `address` column of the `customer` table in the APP schema, thus indicating the need to include the `address` column in the INSERT statement.

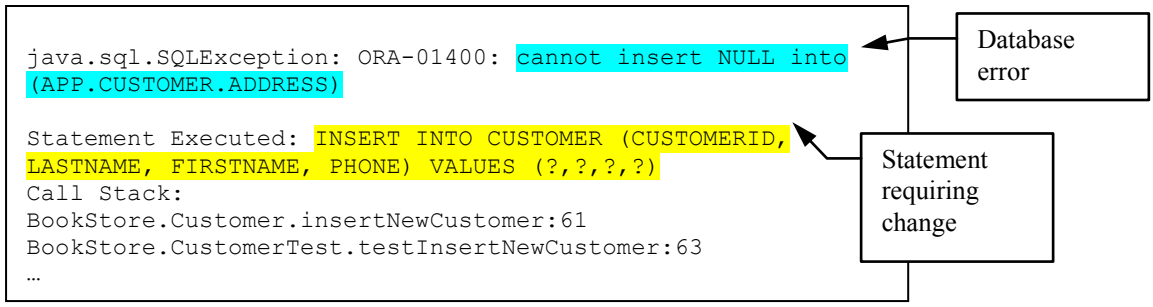

Figure 13: Log file after changing the address column to not allow nulls

All data referenced by foreign keys must already exist in order for this approach to work. If they do not, the statements may be identified as requiring change due to the absence of parent data.

In another type of INSERT statement, the inserted values are not directly specified but are selected from another table(s). An example of this INSERT statement is shown below:

```
INSERT INTO customer (customerID, lastname, firstname, phone)
       (SELECT ?, lastname, firstname, phone
       FROM individual
       WHERE individualID = ?);
```

The table named `individual` contains records of all individuals with names and phone numbers. When an individual becomes a customer, a record is inserted into the `customer` table by specifying a new `customerID` and obtaining the `firstname`, `lastname` and `phone` of the customer from the `individual` table.

When INSERT statements involve the process of selecting from another table, the SELECT element is treated as if it was a regular SELECT statement, and hence, we append additional criteria to the SELECT part of the INSERT statement.

As a result, our approach recognized when any of the referenced objects, either in a main INSERT statement or in a sub SELECT statement, were changed.

5.4. UPDATE STATEMENTS

As in the case of SELECT and DELETE statements, we append the WHERE clause of the UPDATE statements. Similar to SELECT, DELETE and INSERT statements, UPDATE statements are checked for the validity of referenced objects. For instance, consider a typical UPDATE statement:

```
UPDATE orderdetails
       SET quantity = ?
WHERE orderID = ?;
```

If any of the referenced elements, including the table `orderdetails` or the columns `quantity` and `orderID`, experience a change in name, the statement is correctly identified as needing modification. Figure 14 contains a portion of the log file referring to the UPDATE statement after the `quantity` column was renamed.

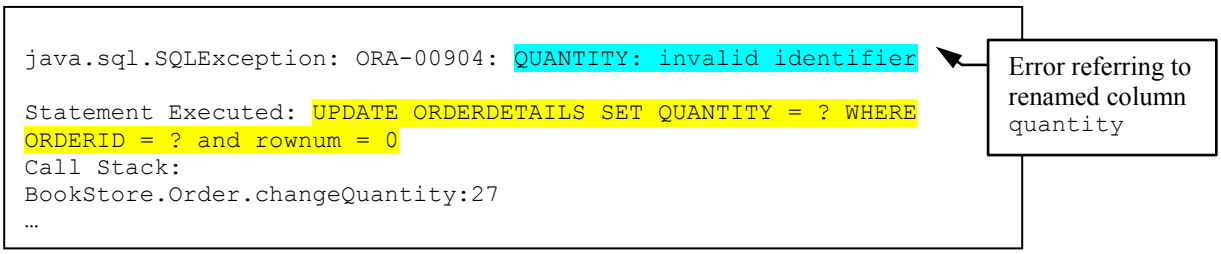

Figure 14: Log file after renaming the quantity column

Statements requiring change due to modifications such as renaming, removing tables, columns, views, and merging or splitting tables, were successfully found. However, the addition of columns, tables or views does not affect the existing UPDATE statements.

While SELECT statements are not affected by the addition or change of constraints, INSERT and UPDATE statements are affected by constraint changes. In this case, the proposed approach does not find statements requiring modification due to such changes. As opposed to INSERT statements, which are affected by the addition of new columns, UPDATE statements are not affected by this modification. Because we append the WHERE clause with the criteria that evaluate as false, no data is updated, and therefore, there is no validation relating to constraints.

5.5. EXECUTION TIME FINDINGS

In order to assess the execution time of this approach, we created a test sample set of 144 unit tests. With the proposed approach, the database was accessed 120 times, including 96 statements that were appended with the additional criteria and 24 statements that were executed without modifications, such as direct INSERT statements or statements retrieving sequence numbers. In total, the execution of 144 tests took 13 seconds on a workstation with a 2GHz processor and 3G RAM.

In the second step, the switch is turned off so that the database is not accessed. As a result, the unit tests in the first step were transformed into traditional unit tests where database access is mocked. In this step, the execution time took 1.6 seconds.

The process of transforming traditional unit tests into unit tests with database access increased the test execution time. Nevertheless, the execution still averaged approximately 0.09s per test. Although these tests are still relatively fast, their reduced speed in comparison to the traditional unit tests may not be acceptable for daily or hourly executions of large test suites. However, the proposed approach is solely intended for use with database schema changes; therefore, traditional unit tests can be used for daily operation whereas unit tests with database access should be utilized only when there is a database schema change.

5.6. GENERALIZED VALIDATION FINDINGS

The implementation demonstrated that the process of integrating the database access into a traditional unit test is relatively simple. In the case of our implementation, which used a traditional unit test with a mock `ResultSet` and `SQLExecuter` classes, all changes were contained within the `SQLExecuter` class. However, different database access mocking may add complications to this step. Although it is possible to integrate the proposed approach into an existing unit test suite, this integration is strongly dependent on the way in which the mocking is performed in the existing tests.

The main limitation of the proposed approach lies in its lack of success with added, changed or removed constraints. However, this limitation does not affect SELECT statements, because they do not require any changes as a result of constraint modifications.

Some of the approach's limitations can be overcome by using temporary schema changes. For instance, if the table name is temporarily changed, the system will find all queries referencing the modified table. All statements can be identified and reviewed from the log file to determine whether or not they require any

changes. After the statement changes are performed, the table is reverted to its original name.

Although the proposed approach is primarily focused on database evolution changes, it is also successful in evaluating statements during the unit testing phase. Since, in unit testing, the statement is evaluated against the database, there is no need to run the full application for validating the statement. The ability to evaluate statements enabled the early detection of common writing mistakes, such as spelling and syntax errors, which would otherwise be detected only during run time.

This work aims to identify changes that are required in the application code as a result of modifications in the database schema, which is similar to impact analysis [3, 4, 5]. Impact analysis approaches analyse existing application code with regards to database schema changes and create an impact report. The impact report requires tools to conduct application code analysis and, therefore, it is highly dependent on the application implementation language; for instance, Maule et al. [4] concentrate on impact analysis for the C# code. This work also produces an impact report in the form of a log file, but it does not require a separate tool for impact analysis, as it is integrated in the unit testing.

6. CONCLUSIONS

Since modern business requirements change rapidly, databases should evolve in order to continue meeting organizational needs. Changes to the database schema have generally been avoided because of the impact they may have on the applications [3, 6].

Application changes are supported by unit tests, which represent a form of safety net for application changes and refactoring. Currently, there is no tool that provides such a safety net for database application changes in regards to schema evolution [1]. Furthermore, we could not find a study that specifically targets application testing in regards to database schema evolution.

In this work, we propose schema evolution that is supported by unit tests accessing the database. Although the queries are not fully executed and are only parsed, the process of parsing still enables the evaluation of referenced objects. This approach allows unit tests to be conducted quickly, since the queries are not fully executed but are still evaluated against the changed database.

The approach was validated by implementing a testing framework and using a simple database schema. It found most of the statements requiring modification due to schema evolution. The implementation was relatively simple and only required minor changes to traditional unit tests.

The next step would entail evaluating the approach with a set of real life applications containing a range of different applications from transaction processing systems to data warehouses. However, it would be a significant challenge to evaluate and compare the success rate, due to the varying frequencies of schema changes with different applications and due to the dependency of the success rate on the type of database application and the type of schema change.

Additionally, we plan to improve the usability of the produced results. For instance, the textual log file will be replaced with a user interface that will facilitate the message navigation and will direct the user precisely to the code location where the problem originated. Moreover, database error messages will be supplemented with more descriptive, user friendly messages.

7. REFERENCES

[1] S.W. Ambler and P.J. Sadalage, *Refactoring Databases: Evolutionary Database Design*, Addison-Wesley, 2006.

[2] D. Draheim, M. Horn and I. Schulz, "The Schema Evolution and Data Migration Framework of the Environmental Mass Database IMIS", in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, June 2004, pp. 341-344.

[3] G. Papastefanatos, F. Anagnostou, Y. Vassiliou and P. Vassiliadis, "Hecataeus: A What-If Analysis Tool for Database Schema Evolution", in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, Apr 2008, pp. 326 – 328.

[4] A. Maule, W. Emmerich and D. S. Rosenblum, "Impact Analysis of Database Schema Changes", in *Proceedings of the 30th International Conference on Software Engineering*, May. 2008, pp. 451-460.

[5] A. Karahasanovic and D. Sjoberg, "Visualizing Impacts of Database Schema Changed – A Controlled Experiment", in *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, Sep. 2001, pp. 358-365.

[6] C. A. Curino, H. J. Moon and C. Zaniolo, "Graceful Database Schema Evolution: the PRISM Workbench", in *Proceedings of the VLDB Endowment*, Vol. 1 , Issue 1., Aug. 2008, pp. 761-772.

[7] C. A. Curino, H. J. Moon, M. Ham and C. Zaniolo, "The PRISM Workbench: Database Schema Evolution Without Tears", in *Proceedings of the 2009 IEEE International Conference on Data Engineering*, 2009, pp. 1523-1526.

[8] T. Mackinnon, S. Freeman and P. Craig, Endo-Testing: Unit Testing with Mock Objects, Extreme Programming Examined, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001, pp. 287 – 301.

[9] DbUnit, http://www.dbunit.org/

[10] C. A. Christensen, S. Gundersborg, K. de Linde, "A Unit-Test Framework for Database Applications", in *Proceedings of the 10th International Database Engineering and Applications Symposium*, Dec. 2006, pp. 11-20.

[11] C. A. Christensen, S. Gundersborg, K. de Linde and K. Torp, "A Unit-Test Framework for Database Applications", http://dbtr.cs.aau.dk/DBPublications/ DBTR-15.pdf, May. 2006,

[12] DBMonster, http://dbmonster.kernelpanic.pl/

[13] DataFactory, http://www.quest.com/Quest_Site_Assets/ PDF/DSD_DataFactory_F.pdf

[14] D. Chays, 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, pp. 147-157.

[15] D. Chays, J. Shahid and P. G. Frankl , "Query-based Test Generation for Database Applications", in *Proceedings of the 1st International Workshop on Testing Database Systems*, Article No. 6, June 2008.

[16] Y. Deng, P. Frankl, D. Chays, "Testing Database Transactions with AGENDA", in *Proceedings of the 27th International Conference on Software Engineering*, May. 2005, pp. 78-87.

[17] D. de Vries and J. F. Roddick, "The case for mesodata: An empirical investigation of an evolving database system", *Information and Software Technology*, vol. 49, no. 9-10, 2007, pp. 1061-1072.

[18] D. Willmor and S. M. Embury, "An Intentional Approach to the Specification of Test Cases for Database Applications", in *Proceedings of the 28th International Conference on Software Engineering*, May. 2006, pp. 102-111.

[19] M. Fowler, *Refactoring : Improving the Design of Existing Code,* Addison-Wesley, 1999.

[20] M. Lippert and S. Roock, *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*, John Wiley & Sons, 2006.

[21] JUnit, http://www.junit.org

[22] csUnit, http://www.csunit.org

[23] P. Hamill, *Unit Test Frameworks*, O'Reilly Media, 2004.

[24] JUnit addons, http://junit-addons.sourceforge.net

[25] DB2 Change Management Expert, http://publib.boulder.ibm.com/infocenter/mptoolic/v1r0/index.jsp?topic=/com.ibm.db2tools.chx.doc.ug/chxucoview01.htm

[26] Oracle Change Management Pack, http://www.oracle.com/technology/products/oem/pdf/ds_change_pack.pdf

[27] MySQL Workbench for Schema Change, http://www.mysql.com/products/workbench

[28] Embarcadero Change Manager, http://www.embarcadero.com/products/change-manager

[29] L. Williams, G. Kudrjavets and N. Nagappan, "On the Effectiveness of Unit Test Automation at Microsoft", in *Proceedings of the 20th International Symposium on Software Reliability Engineering*, Dec. 1009, pp. 81-89.

[30] T. Bhat and N. Nagappan, "Evaluating the Efficacy of Test-driven Development: Industrial Case Studies", in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, 2006, pp. 356 – 363.

[31] E.M. Maximilien and L. Williams, "Assessing test-driven development at IBM", in *Proceedings of the 25th International Conference on Software Engineering*, May 2003, pp. 564 – 569.

[32] TSQLUnit, http://sourceforge.net/apps/trac/tsqlunit

[33] utPLSQL, http://utplsql.sourceforge.net