TESTING RELATIONAL DATABASE USING SQLLint

Hassan Reza and Kris Zarns
University of North Dakota
School of Aerospace Sciences
Grand Forks, ND 58202, USA
reza@aero.und.edu

Abstract – Database systems are an important element of any web based systems. As such, it is very important to test database systems adequately because if any defects go undetected in the design and/or implementation of database systems, it may jeopardize dependability of web based systems. Web-based systems are examples of systems that heavily rely on databases; they are expected to be highly available and reliable. From time to time, database may require changing. As such, any application that is relying on the database needs to be considered for the changes before any changes that can be made to databases. Examples of changes include semantics of the database query language. In this paper, we discuss an approach for generating test cases for queries whose meaning may have been altered by changes in query semantics.

Key Words – SQL, MySQL, Query Regression Testing, Query Semantics, SQLLint, Database Testing

I. INTRODUCTION

Database is a collection of related data. In relational database, the data is organized using tables (relations), records (tuples) and fields (attributes) [9]. Database systems are an important element of any web-based systems. As such, it is very important to test database systems adequately because if any defects go undetected in the design and/or implementation of database systems, it may jeopardize dependability of these systems. Web-based systems are examples of systems that heavily rely on databases; they are expected to be highly available and reliable, the side effect of this high availability and reliability is reluctance for upgrading the database system unless absolutely necessary. At some point this aversion is outweighed by the desire for new functionality and fixes to existing problems. However, upgrading to a new version means more than just handling server down time. Each application that is relying on the database needs to be considered before any changes can take place. Specifically, changes in the semantics of the database query language need to be checked against all of the queries in the existing applications. When there are many changes in semantics, how can we efficiently identify which queries are affected by which changes, and once the queries have been corrected, how can we guarantee that future releases won’t be corrupted by developers who are not aware of the changes in semantics?

The combination of Linux, Apache, MySQL [11], and Php/Perl/Python (LAMP) is a common architecture in web based applications. LAMP provides developers with an open source solution to the problem of rapidly developing systems that need to combine a web server, a database, and some business logic into a desktop or browser based application. As each of these open source projects age, newer versions are released with improvements to performance and additional features. Depending on the type of application in place, there can be major drawbacks to upgrading to the latest version. The new version may not be fully backward compatible, it may require newer hardware, or there may be no seamless way to transition from the old system to the new system. At times these drawbacks can delay upgrading until the point that the older version is no longer supported and no new work is being done on outstanding bugs. This paper will focus on the issue of detecting problems with backwards compatibility in MySQL, many of these compatibility problems are in the MySQL online documentation, however detecting the presence of broken SQL commands in existing applications can be difficult, and the existing test provided by MySQL may not be sufficient to detect unintentional breaks in backwards compatibility. An example of a known incompatibility is the change in the precedence of the comma operator as it is applied as an alternate syntax for inner join. In version 3 of MySQL, the syntax “SELECT * FROM x,y LEFT JOIN z ON y.id = z.y_id WHERE x.id = y.x_id” was equivalent to “SELECT * FROM x INNER JOIN y ON x.id = y.x_id LEFT JOIN z ON y.id = z.y_id”. With the change in precedence of the comma operator in version 5.0, the LEFT JOIN in the first query will be performed before the IMPLICIT INNER JOIN of the comma operator.

In this paper, we focus on developing a method for identifying and/or generating a set of test cases for those queries whose meaning may have been altered by changes in MySQL query semantics. This is done in the context of a series of web applications that have been developed using a very out-dated version of MySQL. The applications in question may need to be available 99.999999999% a year.

Examples of these applications include weather related information systems, road monitoring systems, natural disaster watches and warnings as well as amber alerts systems (A system for notifying officials and citizens statewide of missing children).
In this work, we have studied the key qualities of web based database driven applications; the work points out problems that are present in testing of database applications that may not be considered for structured applications, then it evaluates a number of existing test suites in an attempt to identify one that fits well with the problem of creating regression tests to handle the problem of shifting semantics due to MySQL upgrades. Finally, it is decided to modify an existing program, SQLLint. This program is used to check for semantic query errors. The modification comes in the form of replacing the classes of errors checked for by SQLLint with new classes of errors representing errors that may be caused by shifts in MySQL query semantics.

II. NON-FUNCTIONAL QUALITIES OF WEB APPLICATIONS

There are several qualities that are of primary importance to web based database systems. Examples include security, scalability, availability, performance, usability, maintainability, and portability. The most important of which are security, usability, and performance. Databases store massive amounts of related data in a consistent format. Unauthorized access to this data, inappropriate modification of this data, and/or destruction of this data can have drastic results.

Security needs to be considered for all components of web based applications. A security failure in any component exposes all of its underlying components. If the security of one component is bypassed and work has not been done to secure the underlying components it may be possible for an attacker to gain control of hardware that the system is running on thereby affecting other systems running on the same hardware. The main security concern at the database level is who has permission to what parts of the database. If the database can be configured to give different users various types of access to tables, tests should be developed to make sure that these permissions are set correctly and enforced. At the API level, access to the database should be rationalized on the server side, in other words users should not be allowed to run arbitrary SQL queries. Instead, the application should be given access to different types of queries with the option to parameterize those queries as desired. A set of commands to run different types of queries should be provided to the application with some flexibility in how the underlying queries are parameterized. These commands need to exist on the server side and need to be the only method for a user to access the database. Otherwise the user can bypass the security by accessing the database with a modified client. The input to these commands also need to be sanitized, if the system evaluates arbitrary strings sent from the user as parameters without first removing escape characters, it may be possible for the user to read and modify tables and fields that they are not supposed to have access to.

Scalability, successful web based applications need to be able to scale up to support increasing numbers of users as the customer base of the application increases. Scaling the system up to support more users may require the database to be spread across additional hardware, or it may require updates to the platform that the database is running on.

Availability, depending on the purpose of the application, access to the database may be required continuously. In some cases it may be unacceptable to shut down the production database system. This may be a requirement because of contractual obligation that the system be highly available, or it may be a logical requirement where important data could be lost if the system were to go down at any time.

Maintainability, web based database applications rely on a complex series of technologies to provide a complete system. Maintenance comes in the form of upgrading the application to add new features, fixing a bug that’s been found in the application, upgrading the database, the webservice, or the language that client or server side components are written in.

Portability, a measurement of whether the data in the database be migrated to another database, or a newer version of the same database. New database features may be required to support new features in the application or to improve the scalability of the system. To improve portability, the application should avoid using features that are specific to this database system. Whenever possible standard SQL syntax should be used and non-standard features should be avoided.

III. TESTING OF DATABASE SYSTEMS

Testing databases system is a challenging task. As discussed by Chan et al.[1], Suarez-Cabal et al.[6], and Slutz et al.[2], there are some important differences between structural applications and database applications that need to be considered when developing tests for database applications. First generating meaningful and effective test cases for database applications is difficult. For example testing databases for semantics integrity constraints and participation constraints is considered to be difficult task [8].In a structured application, a known algorithm is tested with a set of inputs of known type to find a result. In a database application, the algorithm is replaced by a SQL query that is generated by the application, the number and type of inputs can vary each time the query is created, and the output can vary in size, type, and format. To form a query, the database schema needs to be considered. Each application will be connected to a different schema, and knowledge of that schema will be needed to generate tests that properly traverse it to arrive at a meaningful result. The set of all SQL queries that are both syntactically correct and semantically correct for a given schema will contain two subsets, queries that are meaningful to the application and queries that are not. The requirements of the application need to be considered to determine which queries are meaningful.
The second difference is a matter of state. Unlike structured applications, database applications also need to include the state of the database as an input. There are actually two separate concerns here; determining the state that the database should be in before running a test case, and scheduling test cases so that changes in database state do not render subsequent test cases inconsequential. For nontrivial databases like the ones used in real world applications, it is not practical to reset the state of the database for each test that is run. Each test case has the potential to alter the state of the database and thereby the input to the next test case. While the entire database can be considered as an input to a test case, understanding what parts of the database the current and all previous test cases interact with can help to separate related test cases so that a minimum of state resets need to be performed during a series of tests. The third difference between structured applications and database applications is verifying that the test was a success after the test case has been run. Again, the state of the entire database is considered to be one of the inputs to test case, and now the state of the entire database is considered to be one the outputs as well as the result of the query. Verifying that the test case was performed correctly involves comparing the input and output state to be certain any changes that should have been enacted by the query were, and that no other changes occurred as well as verifying that the output of the query is correct.

Another concern with database systems is platform independence, platform independence of the database as well as the application that is accessing the database. According to Heckel et al.[7], test cases and their expected results should be considered platform independent, it isn’t until the tests are executed that the platform needs to be considered. Test cases are generated at the application level and should be generated without regard to the platform that the database is being run on. The API used to access the database should remain the same even if the database is moved to another platform, and for a given database state, the results of a test case should remain the same. This is a test of the drivers underlying the API. In the case where the database is on a given platform and the application accessing it changes platforms, the same principle holds true. The API should remain the same regardless of the platform it is implemented on, so the test cases can be developed without consideration of the platform they will be run on, and the platform only needs to be considered when the test cases are run.

IV. RELATED WORKS

A number of academic applications have been developed to provide tools for testing SQL applications as well as the MySQL test suite itself. Examples include RAGS [2], AGENDA [3], SIKOSA [4], and DITTO [5].

MySQL provides a test suite; the suite comes with a number of predefined regression test cases that must be used to show that MySQL was properly installed. The system contains a test language interpreter, mysqltest, and a script to run all of the tests, msqll-test-run.pl. Additional tests can be developed and added to the test program. Users are urged to research any failed tests to determine if they are bugs and if they have to be reported.

RAGS [13] (Stochastic Testing of SQL) are an automated system for generating and executing large amounts stochastic testing of SQL statements. The RAGS system was developed for Microsoft SQL Server. RAGS focuses on generating tests that can be verified and contain the most commonly used syntax, and the ability to configure all aspects of the class of tests being generated; it applies the configurations that maximize the defect detection rate. Stochastic testing supported by RAGS may help to identify new and interesting defects (backward incompatibilities). Using RAGS, it is possible to tune a system to generate tests and check their backwards compatibility by comparing the result of the tests against different versions of MySQL. However, the main issue with RAGS is validating the output results.

AGENDA [3] (A test GENerator for Database Applications) is a combination of tools for parsing a schema, generating a programmatically identifiable database state from that schema, generating test cases, and to test the resulting state and output from executing those test cases. The system requires as input the database schema, the source code for the application, and a selection of sample attributes. The sample attributes are examples of the class of values that would exist in each field of each table. After AGENDA has parsed this input and created a sample database state, the tester chooses from a selection of heuristics to guide AGENDA’s input generation. Once the database state and application input are generated, the application is run with the input and the output state of the database is checked as well as the output of the application. AGENDA is written for PostgreSQL.

SIKOSA [4] is a combination of tools (HTDGEN, HTTrace, and HTPar) for generating a test database, adaptively scheduling regression tests, and parallelizing tests for a database application. HTDGEN takes the schema and the set of queries from the application and generates database states that are meaningful in the context of the application. This approach is meant to increase the number of execution paths that will be reached by the automatically generated tests over the number of tests that would be reached by randomly generated data. HTTrace records the initial state and all subsequent states of the database while running the application against user provided tests. After a complete run HTTrace determines which tests conflict with each other and require the database to be reset. On subsequent runs, these tests are run separately to minimize the amount of time needed to reset the tests by combining tests that are not conflicting but require a reset. HTPar takes this optimization further by determining which tests can be run in parallel. The
recording ability of HTTrace is convenient for comparing results from the same batch of tests against different versions of the software.

DITTO [5] (Database Interaction Testing Tool) is a set of tools for analyzing a database application logging the state of its database before and after queries, and determining what level of coverage existing tests provide. It does this by first parsing the source and inserting logging code around each query, and then executing the application against user supplied tests, finally it compares the logged results to see which of the applications queries have been tested.

Another approach to testing SQL is to check for queries that are syntactically correct but semantically flawed and meaningless. The Structured Query Language (SQL) [10] is a semi-declarative language used by the applications to access the information stored in relational database systems. The most frequent SQL queries used in applications are those that retrieve information from one or more tables of the database [8]. Brass et al[8] describe a system, SQLLint, for checking queries at runtime for known classes of semantic errors. SQLLint focuses on semantic query errors, they define semantic errors as follows: “A semantic error means that a legal SQL query was entered, but the query does not or not always produce the intended results, and is therefore incorrect for the given task.”[8]. Errors are split into two groups, those that require awareness of the task being performed to detect the error and those that can be decided to be errors regardless of the task. SQLLint focuses on the second group that can be decided without knowledge of the task. SQLLint takes as input the schema for the system, and queries. The database state is not considered. Classes of errors include unnecessary complications, inefficient formulations, and violations of standard patterns, duplicates, and possible runtime errors. Unnecessary complications include things like inconsistent conditions.

Ex: “SELECT * FROM employees WHERE type = “fulltime” and type = “parttime”;”.

A tuple can have only one type, so this query will always return an empty set. Inefficient formulations include errors where redundant constraints on output are implemented thereby increasing runtime. These sorts of errors are normally handled by query optimizers. Violations of standard procedures include errors like missing join conditions. In this case, two tables have been joined using the join condition but there are no restrictions placed on the join. The result of this join may vary depending on database engine; typically the two tables will be joined on every tuple. Duplicate errors refer to the generation of identical tuples or tuples with one or more identical fields. This class is concerned with having duplicates when there should be none and excluding duplicates that should be included in the result. One type of duplicate error is misuse of the DISTINCT or GROUP keywords.

Ex: “SELECT DISTINCT name FROM employee AS M, employee AS E WHERE E.manager_id = M.id AND E.job = “Analyst”;”

In this case, the query will only report one name if there are two different managers with the same name. The distinct was intended to handle the situation where a manager has multiple analysts on their team; however it should have been applied to a unique identifier. The class of possible runtime errors includes problems with type conversion. In this case, results that were expected to be a number may be returned as a string and then inserted into another table that is expecting a number.

The schema testing is a software testing method that is different from any traditional software testing, because it requires conceptual design and logical design to validate SQL queries, database constraints (participation and cardinalities), and database operations (insert, update, and delete operations) [8].

In [11], the authors discussed a design of a tool; a tool was devised whose function is to facilitate testing of applications working with testing databases. The test data generation is basically based on specifications and boundary values. Input data were generated to fill a database satisfying integrity constraints and SQL queries of application. Once the output had been generated, then the actual results were compared against the expected results to detect the errors.

V. THE APPROACH

To resolve the issue of changes in MySQL [11] query semantics that may cause errors in existing applications, regression tests for the applications need to be generated and performed to ensure that the existing functionalities have been disturb because of new changes. Some of the changes in semantics will cause only subtle differences in the output of existing queries; many of the queries will not produce obvious errors. Manually inspecting each query to determine if it is affected by one of the changes is error prone, time consuming, and requires extensive human resources. It is unreasonable to develop regression tests for each query corresponding to each change. For example, there are 59 releases of version 3.23 alone, with each release introducing anywhere from 5 to 20 changes to MySQL.

For our applications, we will need to upgrade from version 3.23.20 to version 5.1.34. This upgrade, in turn, requires a total of 155 minor releases and 3 major releases. Therefore, it is important to consider the minor releases, because semantic shifts generally occur between minor releases.

SQLLint can be used to detect queries that match a certain class of semantic errors. For the propose of this study, additional classes of errors representing changes in query semantics will be used instead of the classes originally
created for SQLLint. Just as these classes contained different errors in SQLLint, we will add possible errors caused by semantic changes to our classes. By incorporating the version number of MySQL that introduced the change in semantics into the specification of the error, it will be possible to identify tests that will be required for partial upgrades. This will also make it easier to add additional classes and additional errors for later upgrades.

The new classes will be used to generate warnings at runtime about queries that may be affected by the semantic changes. Given the queries and a list of errors that may affect them, the queries should be examined to identify if they are flawed associated with semantic changes or if they are behaving correctly. Once they have been analyzed, regression tests can be generated to show that they are behaving appropriately for the current version of MySQL. The regression tests are necessary because maintenance will continue on these systems after all of the upgrades have been completed. It is always possible in maintenance that a query that has been fixed may be reverted by a designer that is unaware of the change in semantics or that an error may occur in revision control such that the previous version of the query is reintroduced. In some cases it may be possible to rewrite the query such that it performs correctly for older versions of MySQL as well as new versions. For example, the change in precedence of the comma join operator mentioned early on (see the introduction). Queries making use of the comma operator can be rewritten with explicit join syntax or parenthesis can be added to force the joins to occur in the correct order. Whenever possible this should be done so that the application will require less work to modify if the version of MySQL is ever downgraded, this will also help to improve portability to other systems by standardizing the formation of queries.

Once the upgrade has been completed and the test cases have been designed, and queries have been analyzed, it is important that for any queries that have been rewritten, the version of MySQL they have been rewritten for is documented. This is because there have been cases in MySQL where a change in semantics has occurred in one minor release and been repealed in another minor release. Without knowing which version the query was rewritten for, knowledge of the expected behavior of the query is lost. This is particularly important for queries that cannot be altered to work identically in both the older and newer version of MySQL. If these queries rely on the new semantics, then SQLLint will mark them as possible errors during the next round of regression tests and the queries will have to be analyzed again for a problem that has already been solved.

VI. CONCLUSION

Testing database using SQL is gaining an increasing attention. Web based systems are database driven applications; they are complex systems and are composed of a myriad of complicated technologies, all of which need to work together to provide a finished product. At the heart of web-based application is its data source, a database that can be accessed by either relational or non-relational database management system.

There are challenges associated with upgrading the database capability to a newer release. These changes are compounded by the requirement that existing applications are expected to continue running seamlessly during this upgrade process. Problems are introduced in the upgrade phase in the form of alterations to the query syntax accepted by the database and the semantics of the queries that are accepted. Finding syntax errors, for most part, is trivial; however finding semantics errors introduced by changes in the very meaning of the query language are difficult. A system for describing and locating semantic query errors in existing applications is in place; however this system focuses on classes of errors that can be shown to be faulty without evaluation of their results. This system SQLLint can be modified to support classes of defects that represent the changes in semantics caused by upgrading the database. In adding these new classes we should be able to identify queries in existing applications that need to be reconsidered due to the upgrade.

Once these candidate queries are identified along with their expected semantic errors, they can be analyzed and corrected. From there, regression tests based on the queries can be created to test for a relapse to forms of the query that were correct for the previous version of MySQL.

REFERENCES


