Testing Database Applications

Carsten Binnig
University of Heidelberg
Germany
carsten.binnig@informatik.uni-heidelberg.de

Donald Kossmann
ETH Zurich
Switzerland
kossmann@inf.ethz.ch

Eric Lo
ETH Zurich
Switzerland
ecllo@inf.ethz.ch

ABSTRACT
Testing database application is challenging because most methods and tools developed for application testing do not consider the database state during the test. In this paper we demonstrate three different tools for testing database applications: HTDGen, HTTrace and HTPar. HTDGen generates *meaningful* test databases for database applications. HTTrace executes database applications testing *efficiently* and HTPar extends HTTrace to run tests in *parallel*.

1. INTRODUCTION
Database applications are becoming increasingly complex. They are composed of many components and stacked in several layers. Furthermore, most database applications are subject to constant change; for instance, business processes are re-engineered, authorization rules are changed, components are replaced by other more powerful components, or optimizations are added in order to achieve better performance for a growing number of users and data. The more complex an application becomes, the more frequently the application and its configuration must be changed.

The SIKOSA project is a joint research project of several universities in Western Europe. One goal of the SIKOSA project is to develop new techniques and tools in order to automate the testing and quality assurance of database applications. The goal is to reduce the burden of programmers and engineers (i.e., people) to guarantee the quality of a database application and to provide a (computer-based) infrastructure that automatically checks diverse quality metrics. This project has several apparent results: (a) the quality of the database applications can be dramatically increased – obvious errors which might slip a human's attention can be detected; (b) the cost of testing can be reduced; (c) the time to market of a computer system can be reduced; (d) new computer technology (e.g., fast hardware) can be leveraged in order to achieve even higher quality and reduced costs in the long run.

As part of this project, a suite of tools for testing database applications has been developed: HTDGen, HTTrace, and HTPar. First, HTDGen is a tool for generating a test database such that we could use that to test a database application thoroughly. Second, HTTrace is a tool to run regression tests on database applications. It "learns" during each test and devises a new test run execution schedule after each test. The next test will then be executed according to the new devised schedule so as to minimize the testing time. Third, HTPar is an extension of HTTrace. It is used to execute test runs in parallel on a single or several machines. Its goal is to exploit the available resources (e.g., machine) as well as possible and/or achieve linear speed up in testing.

2. HTDGEN: A TEST DATABASE GENERATOR
In order to execute tests on a database application the database itself has to be initialized with interesting data (called test database). Most methods for generating test databases only consider the *database schema* [9, 4] or generate *random data* for a given statistical distribution [3, 6]. However, these generated databases are inadequate to cover many critical execution paths of the application. It is because these generated test databases never take the embedded *SQL queries* of the database application into account. As a consequence, there is a gap between the generated test databases and the queries of the application during test: this leads to the fact that many queries of the application get no (meaningful) results from the generated databases and thus many execution paths of the application cannot be tested. In order to test the critical execution paths of a database application in a meaningful way, we designed HTDGen. HTDGen is a test database generator that considers both the database schema and the queries of the database application during data generation. Currently, HTDGen supports database applications with embedded SQL statements and it generates relational test databases for testing these applications. To show the idea, consider the following pseudocode fragment from a database application:

```
foreach price in SELECT price FROM Product do
 if(price>=0 && price<=10)
   //do something
 else if(price>10)
    //do something else
 end if
end foreach
```

To test all execution paths of this fragment, HTDGen collects the embedded SQL query Q (`SELECT price FROM Product`), the schema S_D of the target database (e.g., S_D includes the schema of table `Product` and integrity constraints) and the results R of the SQL query (e.g., a table with three rows: -5, 5 and 15 for the attribute `price` of the table `Product`; either given by the testers or by running a code analyzer). Then HTDGen sends Q, S_D

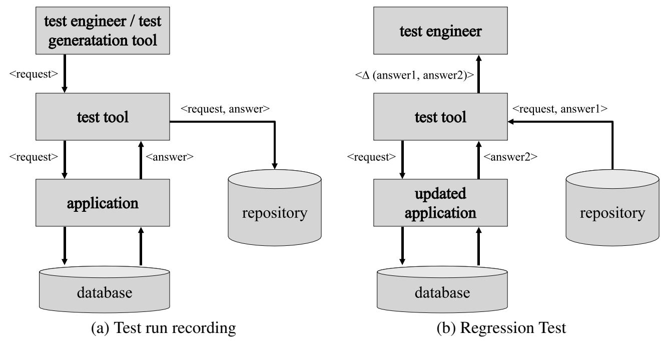

Figure 1: Regression Tests

and R to a reverse query processing engine called SPQR [2]. The reverse query processing engine reverse processes the query and generates a test database D that fulfills all the integrity constraints in S_D and guarantees that $R = Q(D)$. Essentially, SPQR transforms the inputs from HTDGen into a set of constraints and uses a model checker to find possible solutions. The results of SPQR are translated by HTDGen into a set of SQL `INSERT` statements and inserted the data into the test database.

3. HTTRACE: EFFICIENT TEST RUN EXECUTION

Given a test database, the next step is to create test runs and execute the test runs on the database application. A test run is defined as a sequence of requests (that are always executed in the same order) and the expected responses of the application. It is assumed that the test database is in state D at the beginning of the execution of a test run. During the execution of a test run the state of the database may change due to the execution of the requests. For example, a test that checks the reporting component of an order management application must always be executed against the same state of the test database in order to make sure that the report shows the same orders every time this test is executed. Controlling the state of the test database D is a challenging task, if many tests (possibly thousands) need to be executed and if some of these tests involve updates to the database (e.g., tests that test the insertion of a new order).

HTTrace [7] is a capture-and-replay tool developed as part of the SIKOSA project. HTTrace was designed for carrying out black-box database application regression tests efficiently. Figure 1 shows the design of HTTrace. HTTrace has two components. One component (Figure 1a) is a tool for test engineers to record test runs. During the recording phase, the application is expected to work correctly so that the answers returned by the application are correct. Furthermore the state of the test database D after a test run is recorded is expected to be correct, too. HTTrace records the requests and the answers of the application at the end of a test run. After the application code has changed (e.g., customization or a software upgrade), the test engineers can run the recorded test runs by invoking the playback component of HTTrace (Figure 1b) in order to find out how the changes have affected the behavior of the application. This kind of process is called regression testing. HTTrace re-issues automatically the requests recorded in its repository to the application and compares the answers of the updated application with the answers stored in the repository.

Logically, the test database must be reset after each test run is recorded (Figure 1a) and executed (Figure 1b). This way, it is guaranteed that all failures during the playback phase are due to updates of the application layer (possibly, bugs). However, resetting the database is a time consuming process: it takes about two minutes to reset a 100MB database [7]. Therefore, HTTrace is equipped with a set of control strategies to minimize the number of database resets during testing. The basic idea is to apply database resets lazily. During each regression test, it *learns* which test runs are possibly hurting (conflicting) with each others. As a result, in subsequent regression tests, HTTrace reorders the possibly conflicting test runs in order to reduce the number of database resets. If the number of database resets can be reduced, obviously the regression test time can be reduced, too.

4. HTPAR: PARALLEL TEST RUN EXECUTION

Executing test runs in parallel is not uncommon when many test runs need to be executed. HTPar [8] is an extension to HTTrace so that test runs can be executed concurrently on a single or several machines. The goal is to exploit the available resources as well as possible. If several machines are available, the goal is to achieve linear speed-up; that is, the running time of executing the test decreases linearly with the number of machines. In order to achieve this speed-up, it is important to balance the load on all machines - just as in all parallel applications [5]. At the same time, however, it is also important to control the state of the test database(s) and to execute the test runs in such a way that the number of database reset operations is minimized - just as for non-parallel testing in HTTrace. As a result, parallel testing involves solving a two-dimensional optimization problem: (a) *partitioning:* deciding which test runs to execute on which machine; and (b) *ordering:* deciding in which order to execute the test runs on each machine.

HTPar supports two parallel testing modes: Shared-Nothing mode and Shared-Database mode. In Shared-Nothing mode (SN), there are N separate and independent installations of the application and its underlying database. The installations do not share state and, thus, do not interfere. In Shared-Database mode (SDB), there is only one installation of the application and its underlying database and test runs are executed concurrently on this instance. In this case, concurrent test runs interfere because they read and update the same database. Figure 2 shows the architecture of HTPar. The architecture is applicable to both Shared-Nothing mode and Shared-Database mode. The scheduler of HTPar has an input queue of test runs (e.g. T_{12}, T_5, T_{31}, ... in Figure 2). How to order the test runs in this input queue depends on the scheduling strategy (SN or SDB) [8]. At the beginning, the scheduler takes the first test run from its input queue and submits it for execution to Machine 1 in the SN mode or to Thread 1 in the SDB mode. (Figure 2 shows machines in the SN mode, but the same principles apply to feeding test threads in the SDB mode.) Furthermore, the scheduler submits the second test run to the second machine/thread and so on until all N machines/threads are busy.

When a machine (or thread), say M_i, has completed the execution of a test run, say T_k, M_i notifies the scheduler that it is ready to execute a new test run. The scheduler keeps a history of all test runs that have been executed on M_i and correspondingly places T_k into its history for M_i. Furthermore, the scheduler selects the next test run to be executed on M_i from its input queue. In most cases, the scheduler selects the first test run from its input queue, but there are occasions in which it is beneficial not to select the first test run from the queue. In SN, for example, if it is known that T_{12} and T_{17} are in

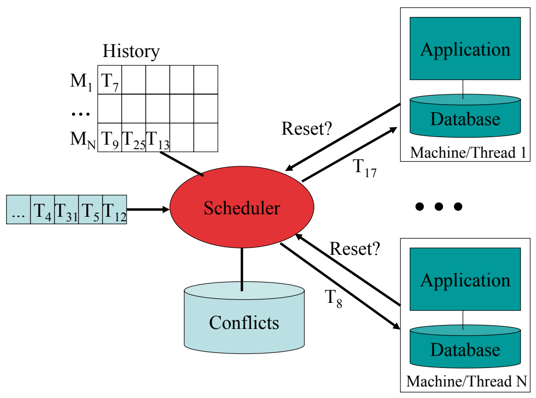

Figure 2: HTPar architecture

conflict[1] in Figure 2, then it might be beneficial not to execute T_{12} on M_1 after T_{17} has been executed on M_1 and instead execute T_5 on M_1 and wait until another machine becomes available for T_{12}. In order to decide which test run to execute next, the scheduler takes the conflict database, the history, and the order in the input queue into account. Alternative policies how such optimizations are applied for SN and SDB are described in [8].

When a machine/thread informs the scheduler that it has completed the execution of a test run, it also indicates whether it has to reset the database in order to execute the test run. Recall from the previous section that the database is reset whenever a test run fails in order to make sure that this failure is not due to the test database being in the wrong state. If a reset has been carried out by M_i in order to execute T_k, then the scheduler updates its history information and the conflict database.

Obviously, parallel testing can significantly reduce the running time of executing a set of test runs. In the SN mode, it can be expected that the speed-up is linear because there is no interference and if all test runs have roughly the same length (if not, bin packing must be applied to ensure load balancing). In fact, super-linear speed-up is possible because conflicting test runs can be executed on different machines so that the total number of resets is reduced. In the SDB mode, linear speed-up is only possible for small levels of concurrency and if database resets are rare. A database reset blocks all test run executions. Nevertheless, if a scheduling strategy makes sure that conflicting test runs are not executed concurrently, then significant speed-ups can be achieved in this mode, too, because hardware resources (disks, multiple CPUs and co-processors) are better exploited.

5. DEMONSTRATION

This demonstration will mainly show the benefits of the tools presented in this paper as well as the interplay of these tools. The example application used in this demonstration is an e-shop application that implements the TPC-W benchmark [1].

[1]Formally, a conflict is denoted as $\langle T_i \rangle \rightarrow T$, with $\langle T_i \rangle$ a sequence of test runs and T a test run. A conflict $\langle T_i \rangle \rightarrow T$ indicates that if $\langle T_i \rangle$ is executed, then the database must be reset before T can be executed. For example, one of the test runs in $\langle T_i \rangle$ could insert a purchase order and T could be a test run that tests a report that counts all purchase orders.

We first show the generation of a test database by HTDGen using some SQL statements extracted from the TPC-W application. (e.g. SQL statements from the search customer function).

Based on the generated test database, some test cases are recorded by HTTrace. The test cases are based on the use cases specified in the TPC-W benchmark (e.g., searching products, placing new orders, updating items of the e-shop, etc.). Afterwards the test cases are executed by HTTrace or HTPar. The test cases are executed in different orders so as to show the motivation of resetting the test database. Furthermore, the same test would be executed multiple times in order to demonstrate the optimization components proposed in HTTrace and HTPar could improve the testing time along iterations.

6. CONCLUSION

Testing is one of the most important and time consuming steps in software development. Testing database applications is much more challenging because the testing involves a stateful component: the backend database. While the testing of database application is still in its infancy, the SIKOSA project presents three tools (HTDGen, HTTrace and HTPar) which address the test database generation and the efficient test run execution problem. As part of future work, we plan to extend the tool suite by adding tools on testing the scalability of database applications. Furthermore, we plan to extend existing tools to deal with the evolution of test runs and test databases.

7. REFERENCES

[1] The TPC-W benchmark. `http://www.tpc.org/tpcw/default.asp`.

[2] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. Technical report, ETH Zurich, http://www.dbis.ethz.ch/research/publications/rqp.pdf, 2006.

[3] N. Bruno and S. Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.

[4] D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, verification and reliability*, 2004.

[5] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Comm. of the ACM*, 35(6):85–98, 1992.

[6] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[7] F. Haftmann, D. Kossmann, and A. Kreutz. Efficient regression tests for database applications. In *CIDR*, pages 95–106, 2005.

[8] F. Haftmann, D. Kossmann, and E. Lo. Parallel execution of test runs for database application systems. In *VLDB*, pages 589–600, 2005.

[9] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data for a variable set of general consistency constraints. *VLDB J.*, 2(2):173–213, 1993.