SPECIAL ISSUE PAPER

A framework for efficient regression tests on database applications

Florian Haftmann · Donald Kossmann · Eric Lo

Abstract Regression testing is an important software maintenance activity to ensure the integrity of a software after modification. However, most methods and tools developed for software testing today do not work well for database applications; these tools only work well if applications are stateless or tests can be designed in such a way that they do not alter the state. To execute tests for database applications efficiently, the challenge is to control the state of the database during testing and to order the test runs such that expensive database *reset* operations that bring the database into the right state need to be executed as seldom as possible. This work devises a regression testing framework for database applications so that test runs can be executed in parallel. The goal is to achieve linear speed-up and/or exploit the available resources as well as possible. This problem is challenging because parallel testing needs to consider both load balancing and controlling the state of the database. Experimental results show that test run execution can achieve linear speed-up by using the proposed framework.

Keywords Database applications · Regression tests

F. Haftmann
i-TV-T AG, 85774 Unterföhring, Germany

D. Kossmann
ETH Zurich, Zurich, Switzerland
e-mail: donald.kossmann@inf.ethz.ch

E. Lo (✉)
ETH Zurich, Zurich, Switzerland
e-mail: eric.lo@inf.ethz.ch

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.

Unfortunately, changing a database application is very costly. The most expensive part is to carry out tests after a new version of the release software has been developed. As an example, large software vendors like Microsoft spend 50% of their development cost on testing. As another example, SAP has currently a product release cycle of 18 months of which 6 months are used only to execute tests. Typically, larger-scale system tests that cover the whole application must be carried out every night or at least once a week. Furthermore, to ensure the integrity of the application after the change, regression tests [22] must be carried out with every check-in of new code to check whether the modifications have adversely affected the overall functionality.

In order to carry out regression tests, most organizations have test installations of all their software components and special test database instances. Furthermore, companies make use of a variety of tools that support regression testing; the most popular tool is the JUnit framework that was developed to carry out regression tests for Java applications [3,14]. The advantages of regression testing have also been quantified

🖄 Springer

in several empirical studies [2,22,15]. Unfortunately, testing a database application cannot be carried out automatically using these tools and, thus, requires a great deal of manual work today. The reason for the need of manual work is that the current generation of regression test tools has not been designed for database applications. All these tools have been designed for *stateless* applications. In other words, these tools assume that test runs can be executed in any order. For database applications this important assumption does not hold: a test might change the test database and, thus, impact the result of another test. For example, a test that checks the reporting component of an order management application must always be executed against the same test database in order to make sure that the report shows the same orders every time this test is executed. As a result, it is necessary to manually control the database state at the beginning of each test.

Controlling the state of the test database during regression testing is a daunting 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). This work devises a framework to automate this task and gives strategies to execute regression tests in the most efficient way. The goal of this work is to exploit the available resources for testing as well as possible. If several machines are available, the goal is to achieve linear speed-up; that is, the running time of executing all tests decreases linearly with the number of machines. Furthermore, to fully utilize the resources of a machine (e.g., disks, multiple CPUs and co-processors), test runs are executed concurrently by different threads using the same database instance on the same machine. In order to achieve the overall speed-up, it is important to balance the load on all machines – just as in all parallel applications [8]. 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 expensive database reset operations is minimized. As a result, testing database applications 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.

Another interesting problem about database application testing is that today there are no standard ways to benchmark testing frameworks for database applications. In the software engineering community, the most common way to benchmark testing frameworks is to execute tests against real database applications (e.g., [22, 13]). This approach, however, cannot effectively evaluate the performance of the frameworks in different aspects. For example, it is not possible to increase the number of real test runs in order to examine the scalability of the testing frameworks.

In order to tackle the above problems, this work makes the following contributions:

– A framework for database application regression tests is proposed. The framework includes tools to record and automate test run execution for database applications. To execute tests efficiently, the framework generalizes our earlier work in [11,12] such that tests can be executed on multiple machines in parallel and each machine can execute tests by multiple threads concurrently. In addition, the framework allows machines to join or leave the test farm during test run execution.
– Alternative scheduling strategies are presented in order to determine which test runs are to be executed on which machine/thread and in which order.
– A methodology to evaluate database application testing frameworks is presented. It describes ways to synthesize a database test application including a test database and test runs. It also describes how to simulate a database reset operation based on these synthetic components.
– The efficiency of the proposed framework and the trade-offs of the proposed scheduling strategies are studied through comprehensive simulations.

The remainder of this paper is organized as follows: Sect. 2 gives an overview of regression tests on database applications. Section 3 describes the basic ideas for centralized testing (i.e., test runs are executed on one machine sequentially). Section 4 describes the framework for parallel testing on multiple machines and multiple threads. Section 5 describes various scheduling methodologies for the parallel testing framework. Section 6 contains the details of the methodology to evaluate database application testing frameworks. Section 7 includes the results of performance experiments that compare the trade-offs and effectiveness of the proposed techniques. Section 8 gives an overview of related work. Section 9 describes conclusions and possible avenues for future work.

2 DB application regression tests

2.1 Overview

Figure 1 shows how users interact with a database application. The application provides some kind of interface through which the user issues requests, usually a GUI. The application interprets a request, thereby issuing
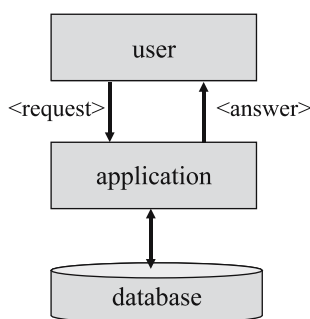
Fig. 1 Architecture of database applications

possibly several requests to the database. Some of these requests might be updates so that the state of the database changes; e.g., a purchase order is entered or a user profile is updated. In any event, the user receives an answer from the application; e.g., query results, acknowledgments, or error messages.

The purpose of regression tests is to detect changes in the behavior of an application after the application or its configuration has been changed. This paper focuses on the so-called black-box tests; i.e., there is no knowledge of the implementation of the application available [20]. As shown in Fig. 2, the regression consists of two phases. In the first phase (Fig. 2a), test engineers or a test case generation tool create test cases. In other words, *interesting* requests are generated and issued to a regression test tool. The regression test tool forwards these requests to the application and receives an answer from the application for each request, just as in Fig. 1. During the first phase, the application is expected to work correctly so that the answers returned by the application are correct and the new state of the test database is expected to be correct, too. The regression test tool stores the requests and the correct answers. For complex applications, many thousands of such requests (and answers) are stored in the repository. If desired, the regression test tool can also record the new state of the test database, the response times of the requests, and other quality parameters in the repository. The regression test tool handles error messages that are returned by the application just like any other answer; this way, regression tests can be used in order to check that the right error messages are returned.

Usually, several requests are bundled into *test runs* and failures are reported in the granularity of test runs. For instance, a test run could contain a set of requests with different parameter settings that test a specific function of the application. Therefore, the requests of a test run are always executed sequentially in the same order. Bundling requests into test runs improves the manage-

ability of regression tests. If a function is flawed after a software upgrade, then the corresponding test run is reported, rather than reporting each individual failed request. Furthermore, bundling series of requests into test runs is important if a whole business process, a specific sequence of requests, is tested. The regression test tool in this paper records test runs and stores them as XML files. This way, management and evolution of test runs are easier (e.g., a test engineer can modify a test run directly).

After the application has changed (e.g., customization or a software upgrade), the regression test tool is started in order to find out how the changes have affected the behavior of the application (Fig. 2a). Basically, there are three steps in executing a test run: (1) *Setup* (2) *Execute* (3) *Report and Clean-up*. During the *Setup* step, the regression test tool prepares necessary data (e.g., input parameters) and the test database for a test run to be executed in the next step. During the *Execute* step, the test tool re-issues automatically the requests of a test run recorded in its repository to the application and compares the answers of the updated application with the answers stored in the repository. Possibly, the tool also looks for differences in response time and for inconsistencies in the test database. In the last step (*Report and Clean-Up*), the regression test tool reports whether the executed test run *failed* or not; failed test runs are test runs for which differences were found. Finally, the last step also frees up all allocated resources for the executed test run. After the execution of all test runs, an engineer examines the failed test runs in order to find bugs and misconfigurations in the application. If the differences are intended (no bugs), then the engineer updates the repository of the regression test tool and records the new (correct) behavior of the application. For traditional software testing tools like JUnit, the *Setup* step and the *Report and Clean-up* step are coded by humans manually.

For database applications, the test database plays an important role during test run execution. The answers to requests in a test run strongly depend on the particular test database instance. Typically, companies use a version of their operational database as a test database so that their test runs are as realistic as possible. As a result, test databases can become very large. Logically, the test database must be reset after each test run is recorded (Phase 1) and before each test run is executed (the *Setup* step of Phase 2). This way, it is guaranteed that all failures during Phase 2 are due to updates at the application layer (possibly, bugs). Again, the *Setup* step and the *Report and Clean-up* step are coded by humans manually in traditional database application testing. The regression test tool

Fig. 2 Regression tests

(a) Phase 1: Teach-In   (b) Phase 2: Regression Test
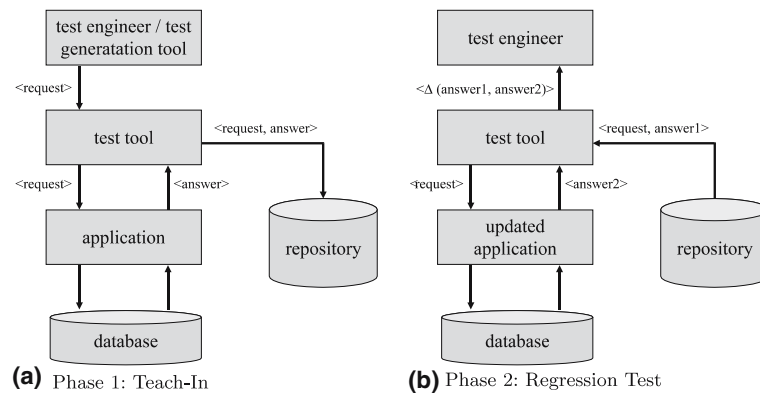
in this paper, however, includes built-in functions to automate and optimize these two steps.

In theory, a test run is okay (does not fail), if all its requests produce correct answers and the state of the test database is correct after the execution of the test run. In this work, we relax this criterion and only test for correctness of answers. The reason is that checking the state of the test database after each test run can be prohibitively expensive and is difficult to implement for black box regression tests. Furthermore, in our experience with real applications, this relaxed criterion is sufficient in order to carry out meaningful regression tests. For checking the integrity of the test database, the interested reader is referred to previous work on change detection of databases, e.g., [4]. If necessary, the techniques presented in that work can be applied in addition to the techniques proposed in this work.

2.2 Definitions

Based on the observations described in the previous subsection, the following terminology is used:

Test database $\mathcal{D}$**:**   The state of an application at the beginning of each test. In general, this state can involve several database instances, network connections, message queues, etc.

Reset $\mathcal{R}$**:**   An operation that brings the application back into state $\mathcal{D}$. This operation is part of the *Setup* step as described in the previous section. Since testing changes the state of an application, this operation needs to be carried out in order to be able to repeat tests. Depending on the database systems used (and possibly other stateful components of the application), there are several alternative ways to implement $\mathcal{R}$; in any event, $\mathcal{R}$ is an expensive operation. For example, in [11], resetting the database of a real-life application takes about 2 min. Resetting a database involves reverting to a saved

copy of the database and restarting the database server process, thereby flushing the database buffer pool.

Request Q**:**   The execution of a function of the application. The result of the function depends on the parameters of the function call (encapsulated in the request) and the state of the test database at the time the request is executed. A request can have side effects; i.e., change the state of the test database.

Test run T**:**   A sequence of requests $Q_1, \ldots, Q_n$ that are always executed in the same order. For instance, a test run tests a specific business process that is composed of several actions (login, view product catalog, place order, specify payment, etc.). The test run is the unit in which failures are reported. It is assumed that the test database is in state $\mathcal{D}$ at the beginning of the execution of a test run. During the execution of a test run the state may change due to the execution of requests with side effects.

Schedule S**:**   A set of test runs $\mathcal{T}$ and reset operations in a particular order. Typically, regression testing involves executing many test runs and reset operations are necessary in order to put the application into state $\mathcal{D}$ before a test run is executed.

Failed test run: An execution of a test run in which at least one of the requests of the test run returns a different answer than expected. Failed test runs typically indicate bugs in the application (as mentioned in the previous subsection, the state of the database at the end of a test run is not inspected).

False negative: A test run fails although the behavior of the application has not changed (and there is no bug). One possible cause for a false negative is that the application was in the wrong state at the beginning of the execution of the test run. False negatives are very

expensive because they trigger engineers looking for bugs although the application works correctly.

False positive: A test run does not fail, although the application has a bug. Again, a possible explanation for false positives is that the application is in a wrong state at the beginning.

The goal of this work is to find schedule S that exploit the available hardware resources for testing, minimize the number of reset operations, and avoid false negative and false positive as much as possible.

3 Centralized scheduling strategies

This section presents basic scheduling strategies in a centralized setting. It repeats some of the finding of [11]. In this setting, tests are run on only one machine and test runs are executed sequentially (one-after-another). For the same set of test runs $\mathcal{T}$, the goal of these strategies is to iteratively develop schedules for $\mathcal{T}$ such that later iterations learn from prior ones and the number of database resets (and the overall running time) of the schedules decreases. In addition, the strategies guarantee that there are no false negatives; false negatives arise, for example, if no resets are executed and a test run is executed and the test database is not in state $\mathcal{D}$ at the beginning due to the prior execution of test runs with requests that have side effects. A naive approach to avoid false negatives is to carry out a reset before the execution of each test run. Unfortunately, this approach has a very poor performance (testing can take weeks instead of hours). The following techniques were proposed in [11] in order to find good schedules in a centralized setting. They are generalized for parallel testing in Sect. 4.

3.1 Optimistic++

The Optimistic++ strategy executes the test runs in a random order. The key idea is to apply resets lazily. If a test run fails (i.e., a request returns an unexpected result), then there are two possible explanations for this failure: (a) the application program has a bug, or (b) the test database was in a wrong state due to the earlier execution of other test runs. In order to make sure that only bugs are reported, the Optimistic++ strategy proceeds as follows in this event:

– Reset the test database (i.e., execute $\mathcal{R}$).
– Re-execute the test run that failed.

If the test run fails again, then the test run is reported so that engineers can look for a potential bug in the application program. If not, then the first failure was obviously due to the test database being in the wrong state. In this case, the test run is not reported and Optimistic++ remembers the test runs that were executed before this test run and records a *conflict* in a conflict database (Sect. 3.4 describes the conflict database in details). Formally, a conflict is denoted as $\langle T_i \rangle \rightarrow T$, with $\langle T_i \rangle$ a sequence of test runs that can be executed in sequence without any database reset, 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. Based on the collected conflicts in the conflict database, the Optimistic++ strategy tries to avoid running a test run twice, in all subsequent tests. For instance, if all or a superset of the test runs in $\langle T_i \rangle$ have been executed, then Optimistic++ will automatically reset the test database (i.e., execute $\mathcal{R}$) before the execution of T, thereby avoiding a failure of T due to a wrong state of the test database.

The Optimistic++ strategy (and all the following strategies which extend Optimistic++) is susceptible to two phenomena called *false positives* and *spurious resets*. False positives means a test run does not fail although it should fail (Sect. 2.2). However, this phenomenon is rare in practice. Also, this risk is affordable because in return it is possible to execute a large number (thousands) of test runs which would otherwise not be possible. Spurious resets means that the strategy may carry out unnecessary database resets. Given a conflict $\langle T_i \rangle \rightarrow T$, Optimistic++ resets the test database before the execution of T if all or a superset of test runs in $\langle T_i \rangle$ precede T. However, when one or more of the additional test runs in a superset of $\langle T_i \rangle$ (i.e., those test runs that are not part of $\langle T_i \rangle$) undoes those effects of $\langle T_i \rangle$ that are in conflict with T, the reset triggered by Optimistic++ is unnecessary. Nevertheless, again, this phenomenon is very rare in practice and the cost of such spurious resets is tolerable.

3.2 Slice

Slice extends the Optimistic++ strategy. Rather than executing the test runs in a random order, the Slice heuristics use the conflict information in order to find a schedule in which as few resets as possible are necessary. For example, if test run T_1 tests the insertion of a purchase order into the database and T_2 tests a report that counts the number of purchase orders, then T_2 should be executed *before* T_1. The conflict information is gathered in the same way as for the Optimistic++ strategy. If there is a conflict between test runs $\langle T_i \rangle$ and T,

and there are no known conflicts between T and any of the test runs in $\langle T_i \rangle$, then Slice executes T *before* $\langle T_i \rangle$. At the same time, however, Slice does not change the order in which the test runs in $\langle T_i \rangle$ are executed because those test runs can be executed in that order without requiring a database reset. Such a sequence of test runs that can be executed successfully between two resets is called a *slice*.

The Slice heuristics can best be described by an example with five test runs $T_1, \ldots, T_5$. At the beginning, no conflict information is available, so that the five test runs are executed in a random order. Let us assume that this execution results in the following schedule ($\mathcal{R}$ denotes the database reset operation, T_i denotes the execution of test run T_i):

$$\mathcal{R} \ T_1 \ T_2 \ T_3 \ \mathcal{R} \ T_3 \ T_4 \ T_5 \ \mathcal{R} \ T_5$$

From this schedule, we can derive the two conflicts: $\langle T_1 T_2 \rangle \rightarrow T_3$ and $\langle T_3 T_4 \rangle \rightarrow T_5$. Correspondingly, there are three slices: $\langle T_1 T_2 \rangle$, $\langle T_3 T_4 \rangle$, and $\langle T_5 \rangle$. Based on the conflicting information in the conflict database and the collected slices, Slice executes T_3 before $\langle T_1 T_2 \rangle$ and T_5 before $\langle T_3 T_4 \rangle$ in the next iteration. In other words, Slice executes the test runs in the following order: $T_5 T_3 T_4 T_1 T_2$. Let us assume that this execution results in the following schedule:

$$\mathcal{R} \ T_5 \ T_3 \ T_4 \ T_1 \ T_2 \ \mathcal{R} \ T_2$$

In addition to the already known two conflicts, the following conflict is added to the conflict database: $\langle T_5 T_3 T_4 T_1 \rangle \rightarrow T_2$. The slices after this iteration are: $\langle T_5 T_3 T_4 T_1 \rangle$ and $\langle T_2 \rangle$. As a result, the next time the test runs are executed, the Slice heuristics try the following order: $T_2 T_5 T_3 T_4 T_1$.

The Slice heuristics reorder the test runs with every iteration until reordering does not help anymore either because the schedule is perfect (no resets after the initial reset) or because of cycles in the conflict data (e.g., $\langle T_1 \rangle \rightarrow T_2, \langle T_2 \rangle \rightarrow T_3, \langle T_3 \rangle \rightarrow T_1$).

Figure 3 shows the pseudocode of the *Slice* algorithm. For each slice $\langle s_m \rangle$, it tries to move it before one of its predecessor slices in order to avoid the reset that must be carried out between slice $\langle s_{m-1} \rangle$ and $\langle s_m \rangle$. The exact criterion that determines whether a slice is moved before another slice is given in Fig. 4. As mentioned earlier, slice $\langle s_m \rangle$ can be moved before slice $\langle s_k \rangle$ if it can be expected that no reset will be needed between $\langle s_m \rangle$ and $\langle s_k \rangle$. More precisely, slice $\langle s_m \rangle$ is moved in front of the first slice that precedes it that satisfies the movable criterion, if there are any such slices.

It should be noted that the Slice heuristics are not perfect. There are situations in which the Slice heuristics produce sub-optimal schedules. So far, no polynomial

```
Input:    (1) sequence of slices ⟨s₁⟩⟨s₂⟩⟨sₙ⟩ obtained
               from the most recent iteration;
          (2) conflict database c :: slice × test run → bool;
Output: new sequence of slices

/* iterate over all slices starting at Slice 2 */
int m := 2;
while m ≤ n do
        /* can we move this slice to an earlier point */
        if ∃k < m : movable(c, ⟨sₖ⟩, ⟨sₘ⟩) then
            k := max{k < m|movable(c, ⟨sₖ⟩, ⟨sₘ⟩)};
            move(m, k);
        fi
        m := m +1;
od
```

Fig. 3 Slice algorithm

$$\text{movable}(c, \langle s_k \rangle, \langle s_m \rangle) = \neg \exists t \in \langle s_k \rangle : c(\langle s_m \rangle, t)$$

Fig. 4 Criterion to move $\langle s_m \rangle$ before $\langle s_k \rangle$

algorithm has been found that finds an optimal schedule (minimum number of reset) given a set of test runs and conflicts. Although it has not been proven yet, the problem is believed to be $\mathcal{NP}$ hard.

3.3 Graph-based heuristics

Graph-based heuristics are alternatives to the Slice heuristics. These heuristics were inspired by the way deadlock resolution is carried out in database systems [21]. The idea is to model conflicts as a directed graph in which the nodes are test runs and the edges $T_j \rightarrow T_k$ are conflicts indicating that T_j might update the test database in such a way that a reset ($\mathcal{R}$) is necessary with probability w in order to execute T_k after T_j.[1]

In [11], a set of graph-based heuristics was proposed and the best heuristics, *MaxWeightedDiff*, is presented here. In order to illustrate this approach, we apply it to a very simple example. This example has three test runs and one conflict $T_1 \rightarrow T_3$ which is not known initially:

iteration	conflict graph	order	schedule
(1)		$T_1 T_2 T_3$	$\mathcal{R} T_1 T_2 T_3 \mathcal{R} T_3$
(2)	$T_1 \rightarrow T_3, T_2 \rightarrow T_3$	$T_3 T_1 T_2$	$\mathcal{R} T_3 T_1 T_2$

In the first iteration, the conflict graph has no edges. As a result, a random order is used; e.g., $T_1 T_2 T_3$. Using the Optimistic++ approach, the database must be reset to execute T_3. As a result, two edges are inserted into the conflict graph ($T_1 \rightarrow T_3, T_2 \rightarrow T_3$) because either T_1 or

---

[1] Similar to the Slice heuristics, in graph-based heuristics, conflicts are detected and accumulated across iterations.

Table 1 MaxWeightedDiff example. Conflict graph after first iteration: $\mathcal{R}T_1T_2T_3\mathcal{R}T_3$

Test run	Fan-in	Fan-out	Fan-in−fan-out
T_3	1	0	1
T_1	0	1/3	−1/3
T_2	0	2/3	−2/3

T_2 or both might be in conflict with T_3 (no other information is available). In other words, an edge $T_i \rightarrow T_j$ in the conflict graph means that T_i possibly overwrites data in the test database needed by T_j. MaxWeightedDiff assigns weights to each edge and the weights assigned are a measure for the probability that T_i is indeed in conflict with T_j. In this example, without further knowledge, it is more likely that T_2 is in conflict with T_3 rather than T_1 is in conflict with T_3 because T_1 does not hurt T_2 and, therefore, is likely not to hurt T_3, too. The further *left* a test run is, the less weight is given to its out-going edge. More precisely, the following function is used in order to assign weights. For

$$\mathcal{R} \; T_1 \; T_2 \; \ldots \; T_n \; T_x \; \mathcal{R} \; T_x$$

the following weights are assigned for edge $T_i \rightarrow T_x$ $(i = 1, \ldots, n)$:

$$\frac{i}{\sum_{j=1}^{n} j}$$

In the example, the weights for the edge $T_1 \rightarrow T_3$ and $T_2 \rightarrow T_3$ are $1/(1 + 2) = 1/3$ and $2/(1 + 2) = 2/3$, respectively. Weights of edges are accumulated across iterations. Therefore it is possible that an edge has a weight greater than 1. Weights are only accumulated if *new* conflicts are recorded in the conflict database.

At the beginning of each iteration, MaxWeightedDiff first calculates the difference between the fan-in and the fan-out of each node in the conflict graph (i.e., *fan−in − fan-out*). Then, it schedules the test runs according to the difference in descending order. Thus, in the example, MaxWeightedDiff will start with the order $T_3T_1T_2$ in the second iteration. T_3 is the first test run because it has the maximum difference (i.e., *fan-in − fan-out*=1) based on the weighted edges (see Table 1). Similarly, T_1 follows T_3 because it has the second largest difference and so on. This criterion favors test runs that do not hurt many other test runs but are potentially hurt by many other test runs. As a result, this reordering results in a perfect schedule with only one reset at the beginning. Therefore, no new edges are inserted into the conflict graph and, thus, the same order will be generated for all future iterations, too.

3.4 Conflict management

Optimistic++, Slice, MaxWeightedDiff, and their extensions for parallel testing (Sect. 5) require the management of conflict information. As mentioned in Sects. 3.1 and 3.2, conflicts are recorded in the form $\langle s \rangle \rightarrow t$ for Slice and Optimistic++, with $\langle s \rangle$ a sequence of test runs and t a test run. For the graph-based heuristics, edges are represented in the form $s \rightarrow t$ which is a special case of $\langle s \rangle \rightarrow t$ so that the same management component can be used.

Conflicts are recorded when regression tests are executed and the control strategies learn. Conflict information is needed by the control strategies in order to determine in which order to execute test runs and when to reset the test database. A conflict database that supports the various heuristics was proposed. Interested readers are referred to [11]. Here, only one of the operations of the conflict database that is important to this work is presented:

– **testSub($\langle s \rangle$, t)**: Test whether there is a sequence of test runs $\langle s' \rangle$ such that $\langle s' \rangle \rightarrow t$ is recorded in the conflict database and $\langle s' \rangle$ is a sub-sequence of $\langle s \rangle$. Sub-sequence is defined as follows: all test runs of $\langle s \rangle$ are also in $\langle s' \rangle$ and they occur in both sequences in the same order. The testSub operation is needed by the Optimistic++ in order to decide where to place resets and by Slice in order to decide if a slice is movable. This operation is also needed when a new conflict $\langle s \rangle \rightarrow t$ is supposed to be inserted into the conflict database. If **testSub($\langle s \rangle$, t)** returns *true*, then the conflict $\langle s \rangle \rightarrow t$ is not recorded in the conflict database because $\langle s' \rangle \rightarrow t$ which is recorded in the conflict database superimposes $\langle s \rangle \rightarrow t$. As a result, the conflict database can be more compact and use less memory.

Staleness of the information in the conflict database is not a problem. As a result, it is not necessary to adjust the conflict database if test runs are modified, test runs are disabled, new test runs are added, or if the application is upgraded. All these events impact conflicts between test runs. Nevertheless, those events need not be propagated to the conflict database for two reasons: (a) It does not hurt if a conflict is not recorded; for example, all approaches work correctly even if the conflict database is empty. (b) It does not hurt if a *phantom* conflict is recorded in the conflict database that does not exist in reality. Such phantom conflicts might result in spurious resets or sub-optimal test run orderings, but all approaches continue to work correctly. In order

to deal with the staleness of the conflict database, we recommend scratching it periodically (e.g., once a month). Such an approach is much simpler than trying to keep the conflict database up-to-date all the time.

4 Parallel testing

4.1 Architecture

This section generalizes the basic techniques presented in Sect. 3 to parallel testing, i.e., execute tests on N machines ($N \geq 1$) and execute several test runs concurrently on each machine. The goal is to devise an efficient test run execution framework that could exploit the available machines as well as possible. Figure 5 shows the architecture of the framework. In the figure, there are N separate and independent installations of the application. Each installation consists of one application on top of a database and both the application tier and the database tier are installed on the same machine. Note that the framework indeed supports different kinds of setup. For example, it is possible that several installations are on a single machine or that a single installation spans several machines (i.e., the application tier and/or the database tier could be distributed on a cluster of machines). What is important is that the installations do not share the underlying database(s) and, thus, do not interfere. For presentation purposes, the term *machine* is used to denote an installation of the application and its underlying database(s) in the remainder of this paper. In order to better exploit the hardware resources, the framework allows test runs to be executed concurrently on different threads using the same database instance on the same machine (i.e., same installation) locally. However, this time, the concurrent test runs do share the same database and might interfere. How to control the multi-programming level (i.e., the number of test threads n on each installation in Fig. 5) depends on the available resources of the hardware. In this work, this multi-programming level is fixed for the duration of the test execution. In future work, we plan to investigate ways to dynamically adjust the multi-programming level according to the load of the machines and the degree of conflicts between concurrent test runs.

As mentioned in the introduction, parallel testing is a two-dimensional scheduling problem. In addition to deciding in which order to execute the test runs, a scheduling strategy must partition the test runs. In the case of multiple machines, parallel testing can decrease the number of resets by executing test runs that are in conflict on different machines. On the other hand, test runs interfere if they are executed concurrently on

the same machine (i.e., test database). Interference can increase the number of resets. As a result, conflict information ought to be taken into account in order to decide on which machine to execute which test run. Furthermore, it is important to balance the load on all machines so that the resources are used as well as possible. Load balancing should be carried out taking the current load of machines and the estimated length of test runs into account.

To support the execution of test runs in a general way, the framework uses a two-step scheduling architecture (see Fig. 5). There is a global scheduler to coordinate the whole test run execution; and there is a local scheduler on each machine to coordinate the local test run execution. The global scheduler has a global input queue of test runs; how to order the test runs in the global input queue depends on the scheduling strategy (Sect. 5).

Each machine is installed with a local scheduler which communicates with the global scheduler and coordinates the local test run execution. Figure 6 describes the basic operations of the local scheduler. In order to achieve load balancing, the local scheduler simply asks the global scheduler for a new test run whenever a thread is free. For example, at the beginning, if a machine M_i has two free threads, P_j and P_k, the local scheduler of M_i first sends a request to the global scheduler for a new test run T_j for P_j. Then, the local scheduler sends another request to the global scheduler for a test run T_k for P_k. The principle applies to all machines/threads until all N machines (and their corresponding threads) are busy. Sometimes, the global scheduler replies to the local scheduler with a *reset-database* signal followed by a test run T_k due to the parallel Optimistic++ policy in Sect. 5.1; in this case, the local scheduler first resets the database and then executes T_k (the details of scheduling a database reset will be described shortly; in Sect. 4.2). The local scheduler continues the execution until the global scheduler replies to the local scheduler with a *finished* signal that indicates no more test runs are left in the global input queue.

The global scheduler keeps a history of test runs that are dispatched to machine M_i. When a machine M_i has completed the execution of a test run T_k, the local scheduler L_i of machine M_i notifies the global scheduler it has executed T_k and is ready to execute a new test run. The global scheduler selects the next test run to be dispatched to machine M_i from the global input queue. In most cases, the global scheduler selects the first test run from the global input queue, but there are occasions in which it is beneficial not to select the first test run from the queue. For example, if it is known that T_{12} and T_7 are in conflict in Fig. 5, then it might be beneficial not to execute T_{12} on M_1 if T_7 is in M_1's execution history. Instead,

Fig. 5 Architecture of the database application testing framework

Fig. 6 Local scheduling
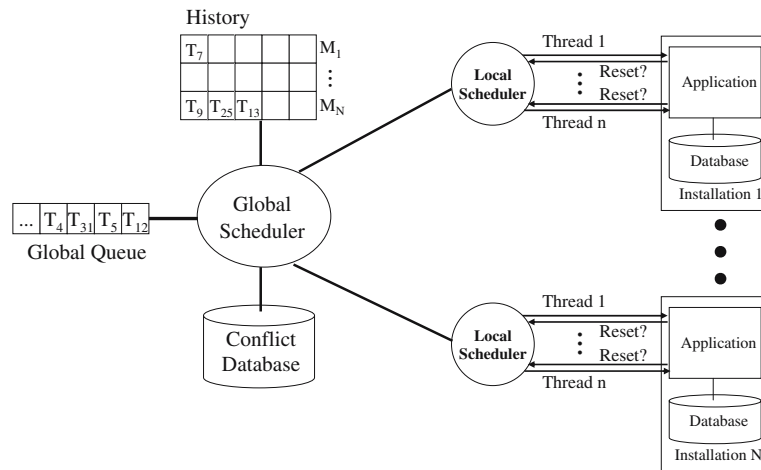
```
Input: number of threads n

/* initialize n free threads */
free := {P_1, P_2, . . . , P_n}

/* request a test run whenever a thread is free */
while (free ≠ ∅) do
        request global scheduler G for a test run
        if G replies with a test run T_k then
                free := free − P /* get a free thread P */
                execute T_k by thread P
        else if G replies with a ⟨reset-database⟩ signal then
                /* for Parallel Optimistic++ */
                schedule a database reset
        else G replies with a ⟨finished⟩ signal then
                wait all busy threads finished and exit
        fi
od

/* Event routine: a test run finished */
on thread P finished execution of T_k:
        free := free ∪ P
        if T_k fails then
                notify G that T_k has caused a database reset
                schedule a database reset
        fi
```

T_5 (the next test run in the global input queue) is dispatched to M_1 and T_{12} will wait until another machine becomes available. In order to decide which test run to execute next, the global scheduler takes the conflict database, the history, and the order in the input queue into account. Alternative policies for such optimizations for global scheduling are described in Sect. 5.

If a reset has been carried out by M_i in order to execute T_k, the local scheduler L_i informs the global scheduler and the global scheduler updates its history information and the conflict database in the following ways:

– *Conflict database*: A conflict $H_{M_i} \rightarrow T_k$ is inserted into the conflict database. Here H_{M_i} represents the sequence of test runs (except T_k) recorded in the history of M_i. This conflict follows directly from the definition of a conflict and the fact that a reset was necessary in order to execute T_k after the execution of the test runs in H_{M_i}.

– *History*: The history for M_i is flushed; i.e., $H_{M_i} := T_k$. The updates of all test runs that were executed on M_i before T_k are undone due to the reset so that these test runs need not be recorded in the history anymore.

Consider an example in which seven test runs have been executed by two threads on one machine locally. Assume that the following schedule is obtained from the machine: (A database reset terminates the execution of test runs in both threads in a machine. How to schedule such resets is described in Sect. 4.2.)

| Thread 1: | $\mathcal{R}$ | T_1 | T_2 | $\mathcal{R}$ | T_2 | T_3 | $\mathcal{R}$ | T_8 |
| Thread 2: | | T_5 | T_6 | | T_7 | T_8 | | |

In this schedule, T_2 is the first test run that fails because the database was in a wrong state for T_2. At this point, the local scheduler notifies the global scheduler that T_2 has caused a database reset. By considering the history of the machine, the global scheduler inserts a conflict $\langle T_1 T_5 T_6 \rangle \rightarrow T_2$ into the conflict database. The second test run that fails is T_8. Again, by considering the history of the machine, the global scheduler inserts the conflict $\langle T_2 T_7 T_3 \rangle \rightarrow T_8$ into the conflict database. These collected information would be used for reordering the test runs in the global input queue and for global scheduling of future regression tests (Sect. 5).

The architecture in Fig. 5 is applicable to any number of machines and any number of threads on each

machine. A machine can join or leave the test run execution seamlessly during runtime. To join the execution, a machine just simply starts its local scheduler and the local scheduler will ask the global scheduler for a new test run. A machine can leave by simply not asking the global scheduler for more test runs to execute. In particular, the system degrades gracefully if a machine breaks during the execution of tests; in this event, the global scheduler must reschedule active test runs from the broken machine.

4.2 Scheduling database resets

A question that is specific to local scheduling is how to schedule a database reset when a test run fails, potentially due to a wrong state of the test database. The question is what happens to the test runs that are executed in concurrent threads? This question does not arise in global scheduling because a reset caused by a failure of a test run on one machine does not impact the concurrent execution of test runs on other machines.

Conceptually, scheduling a reset in a machine is related to the problem of scheduling a check point for database recovery in a database engine [9]. There are several options:

– *lazy*: Concurrent test runs that have started are completed, but no new test runs are started as soon as a test run fails and a database reset becomes necessary.

– *eager*: Concurrent test runs are aborted and the database reset is carried out immediately. After the database has been reset, all test runs that have not completed must be restarted.

– *deferred*: The database reset is deferred and the failed test run is re-scheduled to be executed at the end. In other words, the first reset is carried out after every test run has been tried once. After that, a database reset is carried out and the test runs that failed in the first round are re-scheduled (using one of the heuristics presented in Sect. 5).

The trade-offs between these three alternatives are fairly complex and the choice for the best approach depends on the number of conflicts between test runs. Exploring these trade-offs is one avenue for the future work. For the purpose of this work, the *lazy* strategy was chosen because it is very robust and minimizes the amount of wasted work by failed test runs.

Another related question is how to restart the execution of test runs after a database reset. Again, there are several options:

– *single-threaded*: Execute the failed test run again in isolation; i.e., without starting any other test runs concurrently. The advantage of this approach is that if the test run fails again, it is guaranteed that this failure is due to a potential bug in the application and not due to the database being in a wrong state at the time of the failure due to the execution of update requests by concurrent test runs.[2] The disadvantage is that parallelism is lost.

– *multi-threaded*: Execute the failed test run concurrently with other test runs. Clearly, the advantage of this approach is that no parallelism is lost. The disadvantage is that if the test run fails again, that test run must be executed yet another time until it is successful or fails in a single-threaded environment.

– *mix*: Many other strategies are conceivable. For instance, it is possible to restart with a lower degree of parallelism and/or in single-threaded mode after another failure.

Again, the choice between these alternatives depends on the number of conflicts and the probability that a test run fails after a restart due to other concurrent test runs. We plan to study these effects in detail as part of future work. For the purpose of this work, the *single-threaded* approach was chosen because this approach is robust and minimizes the amount of wasted work due to re-executing failed test runs.

5 Global scheduling strategies

This section presents how the centralized scheduling strategies of Sect. 3 can be extended for the parallel testing framework of Sect. 4. As shown in Fig. 5 and described in Sect. 4, there are two decisions that need to be made by the global scheduler in order to schedule test runs for parallel execution:

– Determine the order of test runs in the global scheduler's input queue of test runs.

– Define a criterion that specifies the situations in which the global scheduler will *not* select the test

---

[2] A test run is *not* a transaction and can span several database transactions. Typically, every request or a small number of requests are implemented by the application as a single database transaction in order to achieve recoverability of long running business transactions. As a result, test runs (which represent such long running business transactions) do see updates carried out by other, concurrent test runs.

run at the head of its input queue for execution in reply to a request by a local scheduler.

The remainder of this section describes how the Optimistic++, Slice, and MaxWeightedDiff heuristics from Sect. 3 are extended to make these two global scheduling decisions for the parallel testing framework. There are similar heuristics proposed in [12], however, those heuristics are not applicable to the parallel framework of Sect. 4. More precisely, the heuristics in [12] are not applicable to support parallel test run execution on multiple machines *and* multiple test threads on each machine.

For all heuristics, the local scheduler also works exactly the same way as described in Fig. 6. Thus, whenever a test run T_k has caused a database reset in M_i, the local scheduler informs the global scheduler such that the global scheduler updates the conflict database and flushes the history for M_i, as described in Sect. 4.

5.1 Parallel Optimistic++

The parallel Optimistic++ strategy takes a simplistic scheduling approach. Figure 7 shows the algorithm details. The key ideas are as follows:

- The test runs are put into the global scheduler's input queue in random order.
- Upon receiving a request from a local scheduler L_i, the global scheduler always dispatches the first test run in the global input queue to L_i.
- Before a test run T_k is dispatched to a machine M_i by the global scheduler, if T_k is known to be in conflict with a subset of the test runs in M_i's history, in other words, if the **testSub($\langle H_{M_i} \rangle, T_k$)** operation[3] in Figure 7 returns *true*, then the global scheduler replies to the local scheduler of machine M_i with a *reset-database* signal followed by T_k. As a result, the local scheduler first resets M_i's database and then executes T_k after the database has been reset. This way, the parallel Optimistic++ strategy avoids unnecessarily executing a test run twice as a result of the test database of M_i being in the wrong state.

Since the parallel Optimistic++ policy needs to detect conflicts between test runs during execution, the global scheduler maintains a history and a conflict database as presented in Sect. 4.

___
[3] It is the operation mentioned in Section 3.4

```
/* initialize the global input queue Q with n test runs */
Q := (T_1, T_2, ..., T_n) /* Test runs are in random order */

/* Event routine: a local scheduler requests a test run */
on local scheduler L_i requests a test run:
    boolean replied := false
    if Q = ∅ then /* all test runs are dispatched */
        reply to L_i with a ⟨finished⟩ signal
        replied := true
    fi
    if (replied = false) do
        remove the first test run T_k from queue Q
        /* Parallel Optimistic++ testing */
        if (testSub(H_{M_i},T_k) = true)
            reply to L_i with a ⟨reset-database⟩ signal
        fi
        reply to L_i with T_k
        write T_k into M_i's history H_{M_i}
    od

/* Event routine: a machine resets its database */
on local scheduler L_i reports T_k caused a database:
    insert H_{M_i} → T_k into conflict database
    flush history H_{M_i}
```

Fig. 7 Global scheduling (Parallel Optimistic++)

5.2 Parallel Slice heuristics

The parallel Slice heuristics extend the parallel Optimistic++ strategy. The key idea is to schedule a whole slice rather than individual test runs in the global scheduler. Recall from Sect. 3 that a slice is a sequence of test runs that can be executed without a database reset; i.e., there are no conflicts within a slice. A slice is defined as a sequence of test runs that was executed successfully between two resets. Thus, during global scheduling, the global scheduler dispatches test runs in the same slice to the same machine.

Figure 8 shows the details of a global scheduler. This global scheduler is designed to work for both the parallel Slice heuristics and the parallel MaxWeightedDiff heuristics presented in Sect. 5.3. At the beginning, the parallel Slice heuristics behave exactly like the parallel Optimistic++ heuristics; that is, the order of test runs in the scheduler's input queue is random and the next test run is selected whenever a local scheduler requests a test run. Consider an example with eight test runs $(T_1, \ldots, T_8)$ and two machines M_1 and M_2 and assume the following schedules are the results of the first iteration (for presentation purposes, the schedule of a multi-threaded machine is flattened according to the timestamps of test runs in the history):

$$M_1 : \mathcal{R} \ T_1 \ T_2 \ T_3 \ \mathcal{R} \ T_3$$
$$M_2 : \mathcal{R} \ T_5 \ T_6 \ \mathcal{R} \ T_6 \ T_7 \ T_8$$

From these schedules, four slices can be identified: $\langle T_1 T_2 \rangle$, $\langle T_3 \rangle$, $\langle T_5 \rangle$, and $\langle T_6 T_7 T_8 \rangle$. Accordingly, the

```
/* initialize the global input queue Q with n test runs */
Q := (T₁, T₂, ..., Tₙ) /* Different heuristics, different order */

/* Event routine: a local scheduler requests a test run */
on local scheduler Lᵢ requests a test run:
    boolean replied := false
    if Q = ∅ then /* all test runs are dispatched */
        reply to Lᵢ with a ⟨finished⟩ signal
        replied := true
    fi
    int k := 1
    while (replied = false and k ≤ n) do
    /* test the k-th test run Q[k] of Q with Mᵢ's history */
        if (isGoodMatch(Q[k],H_Mᵢ) = true) then
            reply to Lᵢ with Q[k]
            Q := Q − Q[k]
            replied := true
            write Q[k] into Mᵢ's history H_Mᵢ
        else /* test the next test run Q[k + 1] in Q */
            k := k + 1
        fi
    od
    if (replied = false) then /* cannot find any good match */
        reply to Lᵢ with ⟨reset-database⟩ /* Par. Opt++ reset */
        reply to Lᵢ with Q[1] /* reply the first test run */
    fi

/* Event routine: a machine resets its database */
on local scheduler Lᵢ reports Tₖ caused a database:
    insert H_Mᵢ → Tₖ into conflict database
    flush history H_Mᵢ
```

Fig. 8 Global scheduling (Slice/MaxWeightedDiff)

following conflicts are detected: $\langle T_1 T_2 \rangle \to T_3$ and $\langle T_5 \rangle \to T_6$.

After each iteration, the parallel Slice heuristics prepares the global input queue for the next iteration as follows:

– Re-order the slices collected from each machine separately. The reordering procedure is the same as the centralized Slice algorithm in Fig. 3.
– Merge the re-ordered slices of each machine into one total order in a round-robin fashion. This total order would serve as the order of test runs of the global input queue in the next iteration.

Continuing the example, after the first iteration, the test runs executed on M_1 and the test runs executed on M_2 are re-ordered by the centralized Slice algorithm separately. As a result, there are two re-orderings, one for each machine ($\langle s \rangle$ denotes the delimiters of slices):

$$O_1 : \langle s \rangle T_3 \langle s \rangle T_1 T_2 \langle s \rangle$$
$$O_2 : \langle s \rangle T_6 T_7 T_8 \langle s \rangle T_5 \langle s \rangle$$

These two (partial) orders of test runs are merged to one (total) order of all test runs which serves as the order for the input queue of the global scheduler in the next iteration. This merge is carried out in a round-robin fashion.

```
Input: Test run T, History H_Mᵢ
Output: boolean

slice ⟨s⟩ := getSlice(T) /*⟨s⟩ is the slice test run T belongs to*/
if ∃t ∈ ⟨s⟩ and t has been dispatched to Mⱼ and j ≠ i
    return false
for each t ∈ H_Mᵢ do
    if (testSub(⟨s⟩), t) is true
        return false
    fi
od
return true
```

Fig. 9 isGoodMatch function (parallel Slice)

That is, the total order is:

$$\langle s \rangle T_3 \langle s \rangle T_6 T_7 T_8 \langle s \rangle T_1 T_2 \langle s \rangle T_5 \langle s \rangle$$

In the next iteration, the global scheduler starts with this re-ordered queue as input. Upon receiving a test run request from a local scheduler L_i, the global scheduler selects the next test run in the global input queue for M_1 based on the following rules:

– Rule 1: Dispatch test runs of the same slice to the same machine.

– Rule 2: Do not dispatch a test run T_k of a slice $\langle s \rangle$ to a machine M_i if any sub-subsequence $\langle s' \rangle$ in $\langle s \rangle$ is known to be in conflict with test runs in M_i's history. In other words, do not dispatch T_k to M_i if the **testSub($\langle s \rangle$, t)** operation (Sect. 3.4) returns *true* for any test run t in H_{M_i}.

The two rules above are implemented in the function isGoodMatch which is called in the algorithm of Fig. 8. Figure 9 shows the pseudocode of the isGoodMatch algorithm. If a candidate test run violates any of the two rules above, the isGoodMatch function returns *false* and the global scheduler considers the next test run in the global queue.

Back to the example: At the beginning of the second iteration, the global scheduler dispatches T_3 to M_1 and T_6 to M_2. If a third machine were available in this iteration of testing, then the global scheduler would *not* select T_7 because T_7 is in the slice of $\langle T_6 T_7 T_8 \rangle$ and T_7 should be executed on M_2 (Rule 1); instead, the global scheduler would select T_1. Likewise, if there are only two machines and M_1 has completed the execution of T_3, then the global scheduler would select T_1 for execution on M_1. Assume that indeed only two machines are available in this iteration and the following schedules are obtained on these two machines after the execution:

$$M_1 : \mathcal{R} \ T_3 \ T_1 \ \mathcal{R} \ T_1 \ T_2$$
$$M_2 : \mathcal{R} \ T_6 \ T_7 \ T_8 \ T_5 \ \mathcal{R} \ T_5$$

As a result, there are still the same four slices as after the first iteration. Two conflicts are added to the conflict database (in addition to the two conflicts detected in the first iteration): $\langle T_3 \rangle \to T_1$ and $\langle T_6 T_7 T_8 \rangle \to T_5$. At this point, there is a cyclic conflict between $\langle T_3 \rangle$ and $\langle T_1 T_2 \rangle$ so that no reordering is attempted for these two traces. Likewise, there is a cyclic conflict between $\langle T_6 T_7 T_8 \rangle$ and $\langle T_5 \rangle$ so that those two slices are not reordered either. As a consequence, the order in which the test runs are put into the global scheduler's input queue for the third iteration is the same as in the second iteration (after a round-robin merge).

Even though the order of test runs in the global input queue did not change, the dynamic behavior of the global scheduler is different due to the additional conflicts recorded in the conflict database. At the beginning of the third iteration, T_3 is scheduled for execution on M_1 and T_6 is scheduled for execution on M_2. At this point, the state of the global input queue is as follows (again, using $\langle s \rangle$ in order to depict beginnings and endings of slices for presentation purposes):

$$T_7 T_8 \langle s \rangle T_1 T_2 \langle s \rangle T_5 \langle s \rangle$$

When M_1 completes the execution of T_3, the global scheduler selects the next test run in the global input queue for M_1. According to the rules of the `isGood-Match` function:

- T_7 and T_8 are not selected because they are part of a slice that is currently executed on another machine, M_2 (Rule 1).
- T_1 is not selected because there is a conflict $\langle T_3 \rangle \to T_1$ and T_3 is in the history of M_1 (Rule 2).
- T_2 is not selected because it is part of the same slice as T_1, and T_1 has a conflict (Rule 2).
- T_5 is selected because it does not violate any of the two rules.

As a consequence, T_5 is dispatched to M_1. This is in the spirit of the Slice heuristics (keeping slices intact) and at the same time gives the biggest hope to minimize the total number of database resets.

Now what happens next when M_1 completes the execution of T_5? At this point, there are no more candidate test runs for M_1 left in the global input queue. Rather than letting M_1 go idle, the parallel Slice heuristics simply send a *reset-database* signal to M_1 (a parallel Optimistic++ reset), and then dispatches the first test run from the global scheduler's input queue; i.e., T_7 in this example. Another possible approach in this case is to select the first test run in the queue that does not violate Rule 1. As a result, T_7 and T_8 are not selected because

T_6 has been executed on M_2. Instead, T_1 is selected in this example. For the purpose of this work, the first approach was chosen (i.e., select the first test run in the queue) because it is simple and as robust as the second approach.

In summary, the global scheduler of the parallel Slice heuristics in Fig. 8 has four ideas: (a) The order of test runs in the global scheduler's input queue is determined by applying the central Slice heuristics to each machine individually and then merging the partial orders (using a round-robin approach). (b) The global scheduler dispatches all test runs of the same slice to the same machine. (c) The global scheduler dynamically uses the history and conflict information in order to make sure that conflicting test runs are executed on different machines as much as possible. (d) Utilizing all available machine resources has higher priority than minimizing the number of database resets. In other words, if no suitable test run is found in the global input queue, the global scheduler selects the first test run from the queue, thereby causing an addition reset, rather than letting the machine go idle.

5.3 Parallel MaxWeightedDiff heuristics

The global scheduler in Fig.8 is also applicable to the parallel MaxWeightedDiff heuristics. Compared to the parallel Slice heuristics, the parallel MaxWeightedDiff heuristics use a different approach to determine the initial order of test runs in the global input queue (i.e., the order of test runs in Q in Fig. 8); and use a different criterion to select a test run from the global input queue for a machine (i.e., a different implementation for the `isGoodMatch` function).

The parallel MaxWeightedDiff heuristics work as follows: After each iteration, the parallel MaxWeightedDiff heuristics prepare the global input queue for the next iteration in exactly the same way as the MaxWeightedDiff heuristics for the centralized setting in Sect. 3.3. As a result, the test runs in the global input queue are ordered according to the weighted differences of the conflict edges.

When the global scheduler receives a request for a test run from, say, machine M_i, the global scheduler selects the first test run T_k from the global input queue, only if for every test run t contained in the history of M_i, the sum of the weights of the edges from t to T_k, summed over all t, is less than a threshold. Otherwise, this criterion is tested for the second test run, third test run and so on, until a suitable test run is found. The `isGood-Match` function shown in Fig. 10 captures the above semantics. The idea is to avoid dispatching a test run T_k that has high probability of causing an additional reset

```
Input: Test run T, History H_{M_i}
Output: boolean

sumweight := 0
for each t ∈ H_{M_i} do
    /* get the weight of t → T in the conflict graph */
    w := weight(t → T)
    sumweight += w
od
if sumweight > threshold
    return false
else
    return true
fi
```

Fig. 10 `isGoodMatch` function (MaxWeightedDiff)

on M_i as much as possible. In other words, the `isGood-Match` function in Fig. 10 returns *false* if the cumulated weights of edges exceeds the threshold. For instance, assume M_i's history has three test runs T_1, T_2 and T_3, and T_4 is the next candidate test run in the global input queue, if the cumulated weights of $T_1 \rightarrow T_4$, $T_2 \rightarrow T_4$ and $T_3 \rightarrow T_4$ is greater than the threshold, the global scheduler would not select T_4 for M_i and try the next test run in the queue. All experiments reported in this paper used 1 as the threshold because this setting was very good in all situations. As part of future work, we plan to study how to set the threshold as a ratio that depends on the conflicting information rather than a static value.

If there are no test runs in the global input queue with cumulated weights less than the threshold (i.e., the `isGoodMatch` function returns false for all test runs in the global input queue), one approach is to select the first test run in the input queue. Another possible approach is to select the test run in the input queue with the smallest cumulated weights as calculated above. For the purpose of this work, the first approach was chosen (i.e., select the first test run) because this approach is robust and minimizes the overhead of keeping the weights of all test runs in the global input queue.

6 Methodology to evaluate database application testing frameworks

In the software engineering community, a database application testing tool is usually evaluated by executing tests against real database applications (e.g., [22,13]). This approach can demonstrate the practicability of a tool for a certain class of applications. However, it cannot effectively evaluate the tool in different aspects. For example, the scalability with the number of real test runs cannot be studied because the number of test runs is

```
Query A:    SELECT conflict-with
            FROM Conflict
            WHERE testrun=?

Query B:    UPDATE Test-Run-Detail
            SET state=state+1
            WHERE testrun=?

Query C:    SELECT state, num-of-request
            FROM Test-Run-Detail
            WHERE testrun=?
```

Fig. 11 Parameterized queries in the synthetic database application

limited by the application. Furthermore, it is impossible to control the number of conflicts between test runs using such an approach.

In order to effectively evaluate a database application testing framework in various aspects, this work is based on the following methodology:

- *A synthetic database application*: A synthetic database application is developed. The synthetic database application follows the typical database application model as shown in Fig. 1. It provides an interface for a user or an external program to issue application requests. Each application request invokes a parameterized database query to a back-end database. The synthetic database application in this work consists of three parameterized queries as shown in Fig. 11.
- *A synthetic database*: A synthetic database that interacts with the above synthetic application is setup. It is a relational database with two relational tables, `Test-Run-Detail` (Fig. 12) and `Conflict` (Fig. 13). Both tables are accessed by the queries in the synthetic application.
- *Synthetic test runs*: A set of test runs are synthesized based on the above synthetic database application and database. The running time of a test run is adjustable by users. A longer test run bundles more application requests (a test run can issue the three application requests more than once) and a shorter test run bundles less. In addition, test runs are synthesized to have conflicts with some other test runs. The degree of conflict between test runs is adjustable.

The evaluation methodology is best described by an example. Consider the evaluation of a testing tool based on the execution of five synthetic test runs. The table `test-run-detail` in the test database is setup to have one tuple for each test run (see Fig. 12). The first tuple $\langle 1, 1, 8 \rangle$ in the table `test-run-detail` represents a synthetic test run T_1 (denoted by the attribute

testrun	state	num-of-request
1	1	8
2	2	41
3	3	12
4	4	115
5	5	76

Fig. 12 Relational table: test-run-detail

testrun	conflict-with
1	2
1	4
2	4
3	5
5	1
5	2

Fig. 13 Relational table: conflict

testrun) with 8 database requests (denoted by the attribute *num-of-request*). Initially, the value of the attribute *state* of a test run is the same as the value of the attribute *testrun*. In the example, there are six synthetic conflicts between the test runs and this conflict information is stored in the table Conflict in Fig. 13. The first tuple $\langle 1,2 \rangle$ in the table Conflict denotes T_1 is in conflict with T_2, i.e., executing T_1 then T_2 requires a database reset.

Assume that the five test runs in this example are going to be executed in this order: $T_1 T_2 T_3 T_4 T_5$. To execute test run T_1, the test tool first requests the synthetic application to select the set of test runs that are in conflict with T_1(Query A). In the example, T_2 and T_4 are in conflict with T_1. Then for each test run that is in conflict with T_1, the test tool requests the application to increase their *state* value by one (Query B). Thus now, the *state* values of T_2 and T_4 are 3 and 5, respectively. Finally, the test tool requests the application to compare T_1's *state* value with its *testrun* value (Query C). If the two values are different, this means there is at least one test run that conflicts with T_1 has been executed and thus T_1 has to trigger a database reset. In this example, since T_1 is the first test run, its *state* value is the same as its *testrun* value, thus T_1 does not fail. In order to model test runs with different workloads (and thus different execution time), the test tool repeatedly requests the application to execute Query A (with random parameter settings) until the pre-defined number of database requests of a test run has been reached. For T_1, the test tool repeatedly requests the application to execute Query A five additional times in order to reach 8 requests. After T_1 finished, T_2 is executed in a similar way. First, the test tool invokes the synthetic application to execute Query A and it finds that T_4 is hurt by T_2 (see Fig. 13). Then, the test tool requests the application to increase T_4's *state*

value by one using Query B. As a result, the *state* value of T_4 becomes 6 (note that T_1 has updated the state value T_4 once). Afterwards, the test tool requests the application to execute Query C for T_2 and it would get different values on *state* (value is 3) and *testrun* (value is 2) because T_1 is in conflict with T_2 (T_1 updated T_2's *state* value). Therefore, T_2 triggers a database reset and is re-executed after the reset.

When a database reset is carried out, the test tool resets the table test-run-detail by setting the value of *state* to the value of *testrun*. If the database reset process is shorter than the user-specific time, the test tool suspends the execution until the database reset running time is over.

7 Performance experiments and results

This section presents the results of performance experiments in order to study the effectiveness of the alternative scheduling strategies for the framework. The experiments were carried out in order to answer the following questions:

- How well does the framework scale if the number of machines is increased? This question is answered by varying the number of machines in Sect. 7.2.
- How well does the framework scale if the number of threads (multi-programming level) is increased? This question is answered by varying the number of threads on five machines in Sect. 7.3.
- How well do the scheduling strategies improve by learning conflicts? This question is answered in Sect. 7.4.

7.1 Experimental environment

The experiments were done on six Linux AMD Opteron 2.2 GHz machines with 4 GB of main memory each. All machines were connected by a 1 G bit Ethernet. In all experiments, the global scheduler and the conflict database were installed on one dedicated machine. For the remaining five machines, each of them was installed with a local scheduler, a synthetic database application as described in Sect. 6, and a backend relational database for the synthetic database application. As the backend databases, IBM DB2 databases were used. This setup simulated the case of five installations on five separate machines. The framework components (e.g., schedulers, conflict database and algorithms) and the synthetic database applications were implemented in Java 1.4.

Table 2 summarizes the synthetic test runs characteristics. In all experiments, 1,000 synthetic test runs were

Table 2 Synthetic test runs

Number of test runs	1,000
Length of test runs	0 min (3 requests) to 3 min (12 K requests)
Number of conflicts	1,000 (low) to 100,000 (high)
Conflict distribution	Uniform

used. The length of a test run was chosen randomly in the range of 0 min (just three requests) to 3 min (around 12,000 requests) using a uniform distribution. These settings were inspired by the real testing environment in [12]. The number of conflicts between the test runs was varied from 1,000 (low) to 100,000 (high) and a uniform distribution was used in order to randomly generate conflicts between test runs when the test runs were synthesized. The simulated database reset was configured to be a 2 min process.

The running time and the number of resets of the parallel Optimistic++, parallel Slice and parallel MaxWeightedDiff heuristics in Sect. 5 were studied. In all experiments, the conflict database was initially empty. A total of thirty regression iterations was executed, thereby incrementally building up conflict information and improving the scheduling decisions. If not stated otherwise, this section reports on the average running time and average number of resets of the last ten iterations. The CPU overhead of the schedulers was also studied. This overhead, however, was negligible (only a few seconds) in all experiments so that it is not reported in this paper.

7.2 Varying the number of machines (one thread)

The first question is to study how well the framework scales with the number of machines. This experiment varied the number of machines from one to five and the number of threads on each machine was fixed to one.
Low Conflict: Table 3a shows the running times (in hours) and the number of resets for synthetic test runs with a low number of conflicts (1,000). Table 3b shows the number of executed test runs per hour per machine for the same set of experiments.

First, from Table 3a, all three strategies achieve a almost linear speed-up with a growing number of machines. The running time is almost five times as high if only one machine is available than if five machines are available. If only one machine is available, the test runs take about 1 day; if five machines are available, the test runs can be carried out within 5.5 h. Table 3b also shows that all three strategies execute a fairly stable number of test runs (34 – 40 test runs) per hour per machine, when the number of machines was varied from 1 to 5. Although not shown in this experiment, this scalability

would easily go beyond 5 machines up to the point at which load balancing and bin packing of test runs with different lengths actually matters or the scheduler itself becomes a bottleneck.

Second, all three strategies have roughly the same running times (see Table 3a). They only differ in the number of resets (parallel Slice and parallel MaxWeightedDiff outperform parallel Optimistic++ in this respect; the parallel MaxWeightedDiff heuristics are slightly better than the parallel Slice heuristics, but the margins are small). However, for the low conflict synthetic test runs, the number of resets is fairly low for all three strategies and executing resets does not impact the overall running time significantly. (Note that resets are executed in parallel with test runs and other resets on other machines.)
High conflict: Table 4a shows the results of the experiments carried out with a high number of conflicts (100,000) between the test runs. Table 4b shows the number of executed test runs per hour per machine for the same set of experiments.

Again, from Table 4a, it can be observed that all three strategies scale well with an increasing number of machines. In general, a regression test takes about 1.5 days for one machine and it takes less than 9 h if five machines are available.

Table 4b also shows that all three strategies execute a fairly stable number of test runs (24 – 30 test runs) per hour per machine, when the number of machines were varied from 1 to 5.

With a high number of conflicts, the parallel Slice heuristics are the winner with regard to the number of resets and running time. In the best case, the running time of the parallel Slice heuristics outperforms the parallel Optimistic++ and the parallel MaxWeightedDiff heuristics by 25%. This shows that the global scheduling of the parallel Slice heuristics successfully schedules conflicting test runs to different machines such that the number of resets are significantly reduced. The parallel MaxWeightedDiff heuristics also outperform the parallel Optimistic++ heuristics, however, the margins are smaller.

7.3 Varying the number of threads (five machines)

This experiment studied how well the framework scales if the multi-programming level is increased on all machines. The number of machines was fixed to five machines.
Low conflict: Table 5a shows the running times (in hours) and the number of resets of the alternative strategies with a varying numbers of threads on five machines and a low conflict scenario. Table 5b shows the number of executed test runs per hour for the same

Table 3 Low conflict, uniform, vary machines

Strategy	1 machine		2 machines		3 machines		4 machines		5 machines	
	Time	Reset	Time	Reset	Time	Reset	Time	Reset	Time	Reset
(a) Running time (hours), resets										
Par. Opt++	27.6	26	13.3	21	9.8	22	6.8	25	5.5	28
Par. Slice	24.9	7	13.1	9	9.7	8	6.3	9	5.4	7
Par. MWD	25.1	7	12.8	8	9.6	8	6.3	7	5.4	6
Strategy	1 machine		2 machines		3 machines		4 machines		5 machines	
(b) Test runs per hour per machine										
Par. Opt++	36.2		37.6		34.0		36.8		36.4	
Par. Slice	40.2		38.2		34.4		39.7		37.0	
Par. MWD	39.8		39.1		34.7		39.7		37.0	

Table 4 High conflict, uniform, vary Machines

Strategy	1 machine		2 machines		3 machines		4 machines		5 machines	
	Time	Reset	Time	Reset	Time	Reset	Time	Reset	Time	Reset
(a) Running time (hours), resets										
Par. Opt++	41.5	266	19.9	251	13.2	240	10.2	245	8.4	262
Par. Slice	33.8	167	16.4	138	10.8	124	8.3	117	6.3	115
Par. MWD	39.5	250	19.7	239	13.1	233	10.1	244	8.1	246
Strategy	1 machine		2 machines		3 machines		4 machines		5 machines	
(b) Test runs per hour per machine										
Par. Opt++	24.1		25.1		25.3		24.5		23.8	
Par. Slice	29.6		30.5		30.9		30.1		31.7	
Par. MWD	25.3		25.4		25.4		24.8		24.7	

set of experiments. Table 5a shows the running time of all three strategies scale well with an increasing number of test threads. Interference is not an issue if the number of conflicts is low. In particular, Table 5b shows that the throughput (number of executed test runs per hour) increases with an increasing number of threads. This observation holds for up to 50 concurrent threads.

Since the number of conflicts is low, all three scheduling strategies have almost the same performance: from Table 5a, both Parallel Slice and Parallel MaxWeighteDiff have the lowest number of resets, but in terms of response time, all three strategies are almost identical. *High conflict:* Table 6a shows the running times (in hours) and number of resets of the alternative strategies with different numbers of threads on five machines and on a high conflict scenario. Table 6b shows the number of executed test runs per hour for the same set of experiments. If the number of conflicts is high, interference matters. Comparing Table 5a and Table 6a, it can be seen that both the number of resets and the running time are much higher with a high number of conflicts. From Table 6b, the throughput (number of test runs executed per hour) of the three strategies increases with an increasing number of test threads. With 50 concurrent test threads, interference matters and the throughput of parallel Optimistic++ and parallel

MaxWeightedDiff drops. The parallel Slice heuristics, on the other hand, have a constant throughput even if the number of threads is increased to 100 (not shown in the tables).

In terms of response time, from Table 6a, parallel Slice is the winner as it has a significantly smaller number of resets.

7.4 Improvement with the number of iterations

As mentioned in the description of the test environment, we carried out 30 regression iterations for each set of experiments and reported the performance averaged over the last 10 out of these 30 iterations. This section studies how the three alternative strategies improve in these 30 iterations. Figure 14 shows the number of resets of each iteration in the experiments of executing 1,000 test runs with 5 machines and each machine has 10 test threads. This experiment was carried out with 1,000 conflicts (low) and a uniform distribution.

In the first iteration, all strategies behaved in the same way: all of them caused 17 database resets. The parallel Optimistic++ did not improve with the number of iterations because it does not learn and does not re-order test runs. Both parallel Slice and parallel Max-WeightedDiff improved with the number of iterations,

Table 5 Low conflict, uniform, five machines, vary threads

Strategy	1 thread		10 threads		20 threads		30 threads		40 threads		50 threads	
	Time	Reset	Time	Reset	Time	Reset	Time	Reset	Time	Reset	Time	Reset
(a) Running time (hours), resets												
Par. Opt++	5.5	28	1.0	17	0.8	13	0.7	11	0.5	8	0.5	10
Par. Slice	5.4	7	0.9	5	0.6	4	0.5	4	0.4	4	0.4	4
Par. MWD	5.4	6	0.9	5	0.6	5	0.5	4	0.4	4	0.4	4
Strategy	1 thread		10 threads		20 threads		30 threads		40 threads		50 threads	
(b) Test runs per hour												
Par. Opt++	181.8		1000.0		1250.0		1428.6		2000.0		2000.0	
Par. Slice	185.2		1111.1		1666.7		2000.0		2500.0		2500.0	
Par. MWD	185.2		1111.1		1666.7		2000.0		2500.0		2500.0	

Table 6 High conflict, uniform, five machines, vary threads

Strategy	1 thread		10 threads		20 threads		30 threads		40 threads		50 threads	
	Time	Reset	Time	Reset	Time	Reset	Time	Reset	Time	Reset	Time	Reset
Par. Opt++	8.4	262	3.2	111	3.2	92	3.1	89	2.3	75	3.0	80
Par. Slice	6.3	115	3.0	94	2.8	80	2.6	79	2.3	67	2.3	66
Par. MWD	8.1	246	3.1	108	2.9	89	2.8	81	2.8	83	3.1	88
Strategy	1 thread		10 threads		20 threads		30 threads		40 threads		50 threads	
(b) Test runs per hour												
Par. Opt++	119.0		312.5		312.5		322.6		434.8		333.3	
Par. Slice	158.7		333.3		357.1		384.6		434.8		434.8	
Par. MWD	123.5		322.6		344.8		357.1		357.1		322.6	

thereby learning more and more conflicts and adjusting the order correspondingly. In this experiment, parallel Slice improved the fastest; it took only a few iterations before a good schedule was found. Parallel MaxWeightedDiff, on the other hand, learnt much slower and was bumpy; it sometimes needed more resets than in the previous iteration. Nevertheless, parallel MaxWeightedDiff improved gradually and had comparable performance to parallel Slice after 20 iterations.

8 Related work

Regression testing is a well-studied technique in software engineering. The most prominent framework is JUnit [3] for Java applications. JUnit is not directly applicable to testing database applications or more generally stateful applications. DBUnit [7] is an extension of JUnit, which facilitates the steps of setting up and resetting the test database during testing. Furthermore, products that support a parallel test environment for stateless applications are beginning to appear on the marketplace; e.g., TestStand [1]. These products can directly benefit from the results of this work. In the software engineering community, there has been a great deal of work in the general area of testing; e.g., white
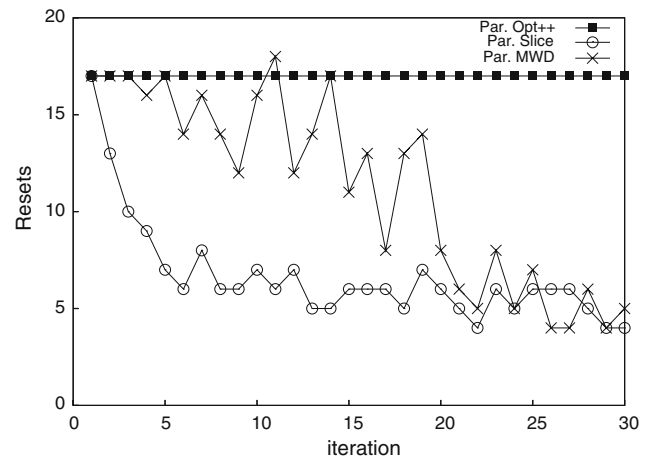
Fig. 14 Improvement with the number of iterations

box and black box testing, analysis of the coverage of test cases, and methodologies to plan and integrate the test phase into the software development life cycle [20]. In order to speed up the execution of testing, the selective execution of test runs has gained a great deal of attention (e.g., [18]). The idea is to execute only those test cases that are potentially affected by a change in the application. Clearly, all that work is orthogonal to the work presented in this paper.

In another aspect, the scheduling problem in this paper is related to compiler design for multi-CPU machines or CPUs with hyper-threading (e.g., [6]), but with totally different assumptions (CPU needs to know and analyze the code) and performance trade-off considerations.

In the database community, there is only very little work on testing. The most relevant work is the work described in [11] and [12]. This work generalizes the framework and algorithms of [11] and [12] for a parallel execution of test runs on multiple machines with multiple test threads. The RAGS system [19] generates a large number of SQL queries in order to test a relational database system. There has also been work on the generation of test databases based on integrity constraints defined in the database schema [5,17]. Furthermore, there has been work on quickly generating large databases with certain attribute value distributions in order to test the performance and scalability of a database system [10]. Again, all this work is orthogonal to the work presented in this paper.

9 Conclusion

This paper proposed a framework to speed-up the execution of a potentially large number of tests for database applications. In this framework, several machines are available for testing in parallel. To better exploit the resources of a single machine (e.g., multiple processors, disks, and co-processors), test runs can be executed by multiple threads on the same machine concurrently. A two-step scheduling architecture was proposed to execute tests on this framework. This approach has several advantages. First, it controls the state of the database and applies expensive database reset operations lazily, thereby minimizing the number of times that expensive database reset operations need to be carried out. Second, it makes dynamic decisions to carry out load balancing and schedule conflicting tests in the best possible way. Based on this general approach, three scheduling strategies were devised that differ in the way that they order the test runs and make dynamic scheduling decisions for concurrent test runs.

A careful methodology to benchmark various testing frameworks and testing tools for database applications was designed. It allows users to control the testing environment (e.g., the number of test runs and the number of conflicts between test runs) so as to test various aspects of a database application testing tool. By the means of simulation, it could be shown that all the three proposed strategies, parallel Optimistic++, parallel Slice and parallel MaxWeightedDiff could achieve linear speed-up in test run execution time. If the number of threads increases, all of them could also achieve super-linear speed-up. Only for a very high degree of concurrency (40 or more test threads) and under a high number of conflicts between test runs, the performance of parallel Optimistic++ and parallel MaxWeightedDiff deteriorate due to interference. Parallel Slice, on the other hand, is able to maintain linear speed-ups under all circumstances tested by the experiments.

The initial results obtained in this study are encouraging. Nevertheless, there is need for future work. First, it is possible to think of more sophisticated scheduling strategies (e.g., based on machine learning techniques). That way it might be possible to get even better results. Second, there is room for improvement with respect to the scheduling of the reset operation in local scheduling (Sect. 4.2). In addition, an important open question is how to dynamically control the multi-programming level (number of concurrent test threads) for local scheduling; we plan to adopt ideas from adaptive load control techniques to avoid lock contention thrashing in databases [16]. Finally, we still believe that the whole field of testing database applications is still in its infancy. There are still several aspects that nobody has ever studied; an example is testing non functional requirements such as scalability of a database application.

References

1. NI TestStand. http://www.zone.ni.com/zone/jsp/zone.jsp
2. Agrawal, H., Horgen, J., Krauser, E., London, S.: Incremental regression testing. In: IEEE Conference. on Software Maintenance, Montreal, Canada, 1993
3. Beck, K., Gamma, E.: Programmers love writing tests http://www.members.pingnet.ch/gamma/junit.htm (1998)
4. Chawathe, S., Garcia-Molina, H.: Meaningful change detection in structured data. In Proc. of the ACM SIGMOD Conference on Management of Data, pp. 26–37, Tucson, AZ, USA, 1997
5. Chays, D., Deng, Y., Frankl, P.G., Dan, S., Vokolos, F.I., Weyuker, E.J.: An AGENDA for testing relational database applications. Software Testing, verification and reliability, 2004
6. Chou Chang, M., Lai, F.: Efficient exploitation of instruction-level parallelism for superscalar processors by the conjugate register file scheme. IEEE Trans. Comput. **45**(3), 278–293 (1996)
7. DBUnit.org. Dbunit homepage. http://www.dbunit.org/
8. DeWitt, D., Gray, J.: Parallel database systems: future of high performance database systems. Comm. ACM **35**(6), 85–98 (1992)
9. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, Sanmateo (1993)
10. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: SIGMOD Conference, pp. 243–252 (1994)

11. Haftmann, F., Kossmann, D., Kreutz, A.: Efficient regression tests for database applications. In: Conference on Innovative Data Systems Research (CIDR), pp. 95–106, 2005
12. Haftmann, F., Kossmann, D., Lo, E.: Parallel execution of test runs for database application systems. In: VLDB, pp. 589–600, 2005.
13. Haraty, R.A., Mansour, N., Daou, B.: Regression testing of database applications. In: SAC, pp. 285–289. ACM, 2001
14. JUnit.org. Junit homepage. http://www.junit.org
15. Liu, Y.: Regression testing experiments and infrastructure. Oregon State University, Master Thesis 1999.
16. Mönkeberg, A., Weikum, G.: Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In: VLDB, pp. 432–443 (1992)
17. Neufeld, A., Moerkotte, G., Lockemann, P.C.: Generating consistent test data for a variable set of general consistency constraints. VLDB J. **2**(2), 173–213 (1993)
18. Rosenblum, D.S., Weyuker, E.J.: Using coverage information to predict the cost-effectiveness of regression testing strategies. IEEE Trans. Softw. Eng., **23**(3), 146–156 (1997)
19. Slutz, D.R. : Massive stochastic testing of SQL. In: VLDB, pp. 618–622 (1998)
20. Sommerville, I.: Software Engineering (5th ed.). Addison Wesley Longman Publishing Co., Inc., Reading (1995)
21. Weikum, G., Vossen, G.: Transactional Information Systems. Morgan Kaufmann Publishers, San Mateo (2001)
22. Wong, E., Horgen, J., London, S., Agrawal, H.: Effective regression testing in practice. In: IEEE Symposium of Software Reliability Engineering (ISSRE 97), pp. 264–274, Albuquerque, NM, USA, 1997