Stress Testing of Transactional Database Systems

Jorge Augusto Meira[1,2], Eduardo Cunha de Almeida[1], Gerson Sunyé[3], Yves Le Traon[2], Patrick Valduriez[4]

[1] Universidade Federal do Paraná, Brazil
{ jmeira, eduardo}@inf.ufpr.br
[2] University of Luxembourg, Luxembourg
{jorge.meira, yves.letraon}@uni.lu
[3] INRIA - University of Nantes, France
gerson.sunye@univ-nantes.fr
[4] INRIA & LIRMM - University of Montpellier, France
patrick.valduriez@inria.fr

Abstract. Transactional database management systems (DBMS) have been successful at supporting traditional transaction processing workloads. However, web-based applications that tend to generate huge numbers of concurrent business operations are pushing DBMS performance over their limits, thus threatening overall system availability. Then, a crucial question is how to test DBMS performance under heavy workload conditions. Answering this question requires a testing methodology to set up non-biased conditions for pushing a particular DBMS over its normal performance limits (i.e., to stress it). In this article, we present a stress testing methodology for DBMS to search for defects in supporting very heavy workloads. Our methodology leverages distributed testing techniques and takes into account the various biases that may affect the test results. It progressively increases the workload along with several tuning steps up to a stress condition. We validate our methodology with empirical studies on two popular DBMS (one proprietary, one open-source) and detail the defects that have been found.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous

Keywords: database, stress testing, testing methodology, transaction

1. INTRODUCTION

The constant race for productivity leads to deploy systems upon workloads that become quickly much beyond the boundaries they were originally designed for. A natural question is how to validate those systems, with a dedicated and tooled methodology, for acknowledging their boundaries and specially seeking what defects might arise whether these boundaries are overtaken. This lack of validation methodology, compared to the huge pressure for deploying large-scale systems, is critical for transactional database management systems (DBMS). Over the last 30 years, DBMS have been used successfully for processing concurrent business operations (e.g., reservations, sales). But during the last decade, the highly concurrent nature of Internet-based systems and the increasing number of connected devices (e.g., tablets, smart-phones, and data sensors) have changed the situation with web-based applications producing huge numbers of operations, with varying load peaks.

This situation is pushing DBMS performance far over their limits, thus making them a potential bottleneck and a threat for overall system availability [Florescu and Kossmann 2009; Stonebraker et al. 2007]. For instance, when processing a heavy transactional workload, the query optimizer of a major commercial DBMS may significantly underestimate the CPU requirements [Soror et al. 2008]. Furthermore, a query optimizer may become inaccurate when the DBMS is running transactional

Supported by the Fonds National de la Recherche, Luxembourg 902399, CNPQ grant 481715/2011-8, Fundacao Araucaria grant 20874 and SERPRO.

workloads that shift along the day [Storm et al. 2006]. Whereas improving the accuracy of a query optimizer upon heavy workload conditions has been the subject of much research work [Gu et al. 2012; Soror et al. 2008; Storm et al. 2006], testing DBMS performance under such stress conditions is an open issue.

In software testing, stress testing is important to validate software upon heavy load conditions with the objective of determining the boundaries beyond which performance is degraded. In general, validation follows a systematic approach to guide a testing campaign in the form of a methodology. In the case of a DBMS upon heavy workload conditions, what we need is a testing methodology to set up non-biased conditions for pushing a particular DBMS over its normal limits with realistic usage scenarios. Among the biases to be isolated, we consider the influence of the operating system (OS) and hardware (such as memory space), and the different knobs for tuning a DBMS.

Several recent tools, including Agenda and AppPerfect, provide test drivers for submitting concurrent database operations based on TPC-like workloads (e.g., TPC-C [1], YCSB [Cooper et al. 2010]). However, they are not suited for stress testing for several reasons. First, the nature of the diagnosis from a benchmark is limited to report performance metrics rather than defects related to the load, which are more likely targeted by software testing practice. Second, they miss a methodology that is required for driving testing campaigns [Ghezzi et al. 2003]. Third, they do not take into account the biases that may affect the test results. Finally, they do not scale up the test load to stress conditions.

In this article, we present a stress testing methodology (STEM) for searching defects related to the load of transactional DBMS. A defect is an imperfection of the system that may result in incorrect results. In the specific case of load-related defects, we claim that they can only be revealed from stressing workloads combined to mistuning in the DBMS software (e.g., insufficient admission control). STEM leverages distributed testing techniques and takes into account the various biases that may affect the test results. It progressively increases the workload along with several tuning steps (a type of step-stress testing [Xiong 1998]) through a distributed test coordination architecture. This distributed architecture allows increasing the load by several orders of magnitude up to stress conditions. We validate our methodology with empirical studies on two popular DBMS (one proprietary, one open-source) and detail defects that have been found.

The rest of the article is structured as follows. The next section discusses related work. Section 3 presents our testing hypotheses and the protocols to validate them. Section 4 presents our stress testing methodology. Section 5 describes our validation through experimentation. Section 6 concludes the article.

2. RELATED WORK

Performance testing helps validating how a system works upon certain workloads. In general, validation is related to some quality factors, such as: availability, reliability, response time, and throughput. In the context of DBMS, the workload is generally part of a benchmark methodology to reproduce a real database environment. Different benchmark approaches have been proposed during the last decades, but the focus is on providing comparison metrics [Jain 1991] rather than software testing. Examples of such benchmarks are TPC-like (TPC-C, TPC-H) [2], and the cloud data service benchmark YCSB [Cooper et al. 2010].

There are two particular types of performance testing explored in the DBMS context: load testing and stress testing. In *load testing*, validation is executed on top of a specific expected workload. Several tools, such as Hammerora, Oracle Application Testing Suite, and AppPerfect, provide a test driver to submit operations based on a TPC-like benchmark. Their objective is to allow engineers

[1]http://www.tpc.org/tpcc/
[2]http://www.tpc.org/

to write test cases and load scenarios. However, they concentrate on presenting performance results and do not seek for load-related defects. They also miss a testing methodology and do not take into account to the various biases that may affect the test results (e.g., memory, network, CPU usage). In addition, they are based on a single test driver [ISO9646 1991] that only allows testing in a point-to-point connection. Reaching stress conditions requires multiple and distributed test drivers that communicate with the system under test (SUT) interface. The testing tool Agenda [Deng et al. 2005] provides both a methodology and a test driver. However, it focuses on database applications, rather than the DBMS itself. Agenda presents a technique for checking database properties (e.g., ACIDity), and does not focus on performance issues.

Stress testing is a form of performance testing with the objective of determining the boundaries beyond which performance is degraded. A stress testing approach for distributed hash tables (DHT) is presented by de Almeida et al. [2010], in which message routing algorithms are stressed in order to validate churn-tolerance and load balance. While our work focus on database transactions reproducing a complex workload, de Almeida et al. [2010] focused on DHT with simple API and simple key-value operations (e.g., put, get). Another difference from our work is the necessity for tuning performance. DHT do not provide any tuning setup leaving to the distribution of the keys (i.e., consistent hashing) the only mechanism to draw better performance in terms of load balance. In our context, we require to tune a number of configuration knobs to draw better performance from the DBMS in terms of resource consumption (CPU, disk, and memory) due to the complex nature of the workload.

In the last decade, stress conditions are becoming more typical due to the popularity of Internet-based systems. Our methodology precisely aims at reproducing such extreme conditions for seeking load-related defects. Stress testing methodologies are generally used by engineers for product failures. These methodologies use a testing method called step-stress testing that progressively increments the complexity of the SUT upon some type of load or operation. Their goal is to accelerate the SUT lifetime in order to expose defects expected to appear all the way through [Xiong 1998] rather than exposing load-related defects.

In the DBMS context, the obsolete TPC-B benchmark was designed to be a stress test with the goal of validating the integrity of transactions upon significant disk processing (i.e., I/O operations). TPC-B reproduces an overnight batch processing with many concurrent transactions without human think times. However, TPC-B is bounded by the resources of a single-sited test driver that cannot scale-up to reproduce large-scale workloads. In addition, it is not designed to reveal a broader range of load-related defects, only focusing on the integrity of transactions.

3. STRESS TESTING FOR DBMS

Stress testing is a hard task. It requires both a test driver and a testing methodology to cope with stress loads and guide the testing campaign. Our methodology for stress testing DBMS is built on top of empirical software engineering hypotheses, which are rather isolated conjectures for empirical investigations. In this section, we present those hypotheses and the protocols to validate them.

Hypothesis 1. *During the test, the variation of the transaction workload exercises distinct parts of the DBMS source-code and enable the validation of different functionalities.*

Validation protocol 1. *Present evidence that more source-code parts are exercised compared to the state of the art in functional testing, through code-coverage or log file analysis. The evidence can be presented in terms of blocks of code ignored by functional testing (e.g., loops, exceptions or conditions).*

Stress testing requires an incremental test approach that drives the DBMS from an initial configuration setup towards a production setup in order to test the functionalities related to transactional workloads. Therefore, we require to gradually increment the workload up to a stress condition, which is beyond the DBMS's state of usage in a real workload environment. The objective is to search for defects related to different scales of concurrent transactions. Suppose that a DBMS is directly tested upon stressing load conditions instead of using an incremental test approach. The results will not show whether the defect appears at smaller scales. In fact, different types of defects appear while a system treats different workloads, because different parts of the source-code are exercised [de Almeida et al. 2010]. We can verify that different parts of the code are exercised either by code coverage analysis or by analyzing the message entries from the log files. The earlier is straightforward by executing the DBMS with its code instrumented by some coverage tool (e.g., GNU/gcov, EMMA). The latter treats the DBMS as a black box and requires to monitor both the response time on transactions and the entries from the log files. Indeed, a higher response time is expected, since the DBMS deals with more transactions and rejects the ones that overtake its configured capacity. Then, the DBMS issues more log entries informing rejected requests or errors that can be analyzed to prove that more functionalities were exercised, for instance, comparing the entries issued from different test steps.

HYPOTHESIS 2. *The OS has no effect on the test results, whether the same defect is revealed on different OS types or the resource consumption remains steady along testing.*

VALIDATION PROTOCOL 2. *Validation can be provided in two ways: (i) analyzing the resource consumption of the OS, providing evidences that such consumption works as expected (e.g., the consumption remains steady along testing); (ii) comparing the test results on different OSs (e.g., Linux, BSD, Windows), providing evidences that the detected defect is the same in all of them.*

Beyond the test driver, other software parts are not supposed to interfere with the test results, especially the OS. We make two assumptions when the same defect is exposed on different OS: (i) the OS is robust, and (ii) the OS is correct. These assumptions demand to continually monitor the OS health (i.e., CPU, memory, I/O, swapping, and networking). The main objective is to ensure that the OS responds to any possible request to the DBMS until the test ends.

HYPOTHESIS 3. *The DBMS does not use all of the allocated resources upon stress workload.*

VALIDATION PROTOCOL 3. *To compare the inconsistencies between the declared DBMS setup and the effective usage of the DBMS resources.*

This last hypothesis is related to the DBMS defects exposed by the stress workload. Since we assume the OS to be robust and the DBMS to be tuned, the defects are the major cause for preventing the DBMS from allocating all the available resources. This also requires to monitor the health of both the DBMS and the OS.

4. STRESS TESTING METHODOLOGY

In this section, we describe STEM, our methodology for stress testing transactional DBMS. In real systems, the workload might arise either from a sustained increase in usage, as when the user community grows, or from a temporary spike. A natural question is what happens if the DBMS is mistuned (e.g., insufficient admission control). In this article, we claim that mistuning is crucial for detecting defects in the DBMS internal code. Therefore, the goal of STEM is to capture the performance degradation and expose system defects in the internal code due to the combination of a stress workload and mistuning. A defect differs from a performance degradation in the sense that the system does not provide the expected service as specified (i.e., transaction requests are not accepted while they should be). STEM follows an incremental approach: the DBMS is tested from its simplest setup to the most complex one, which is done by tuning the setup gradually.

Table I. Workload Setup Parameters

Workload	**Condition**
Min	if equal to declared max_conn, with $\|T\| = 1$
Med	if equal to $\|T\| * max\_conn$, with $1 < \|T\| \leq 10$
Max	if equal to $\|T\|^2 * max\_conn$, with $1 < \|T\| \leq 10$

STEM is based on the joint increment of two variables, concerning the DBMS knobs on one hand and the workload on the other hand. The DBMS knobs correspond to the setup parameters to be tuned for a workload. The workload corresponds to the number of submitted transactions and how they are submitted. The variation of these *dependable variables* is necessary for testing the SUT under different angles in order to find defects related to different workload levels and more specifically, defects related to the performance degradation boundaries.

4.1 Dependable Variables

The number of knobs both in the DBMS and OS is rather large. We only focus on the DBMS knobs since we assume that the OS has been completely tuned up to the hardware capacity. In fact, we pay special attention to the knobs that are impacted the most by transactional load spikes, including the buffer pool management ($work\_mem$), and the admission control (max_conn). Other knobs can also be explored, including group commit, and log flushing, however, we do not focus on them due to space constraints.

The DBMS knobs can accept a large domain of values, from their minimum up to maximum capacity. For the sake of clarity, we assume that each knob can only have three values related to the DBMS capacity: min, med, max.[3] To setup these values, we follow the rules of thumb from the DBMS documentation. Once the DBMS is tuned, the workload is created based on its capacity. Moreover, the workload is related to the way the tests are executed. To mimic real distributed clients and reproduce a stress workload, it is required to deploy many distributed testers, denoted by T, for submitting a large number of transactions. The number of distributed testers depends on the size of the workload (see Table I).

The minimum size workload (min) is set equally to the max_conn configuration and executed by one tester, denoted by $t \in T$. The objective is to establish the performance baseline due to low concurrency. The medium size workload (med) is set to $|T| * max\_conn$ to increase the degree of concurrency. The objective is to evaluate the robustness of the DBMS. The maximum size workload (max) is set to $|T|^2 * max\_conn$ to stress the DBMS. In addition, the DBMS server and the tester machines must be separated from each other to avoid any interference on the performance results.

4.2 Execution Sequence

The execution sequence consists of many steps to drive the DBMS from an initial state up to the stress state (i.e., the incremental test approach). The steps are linked to many setup combinations for each knob. This consists of running the DBMS with all values assigned to dependable variables, which leads to 3^3 setup combinations of $work\_mem$, max_conn, and *workload*. We used pairwise testing[4] to narrow the number of variations to 9 combinations. Moreover, we verified whether some of these combinations are considered useless, for instance, a DBMS must be started with a sufficient amount

[3]The process of identifying the variables and then decomposing their input domains into equivalent classes (from a testing perspective) is called "category-partition testing" in the software testing community [Ostrand and Balcer 1988].
[4]*"Pairwise testing is a combinatorial method in which all possible pairs of parameter values are covered by at least one test [Kuhn et al. 2008]."*

Table II. STEM's Execution Sequence

Setup	**Work_mem**	**Max_conn**	**Workload**	**Test Objective**
1	Min	Min	Min	Installation
2	Max	Min	Max	Degradation baseline
3	Max	Max	Min	Tuning
4	Max	Max	Med	Robustness
5	Max	Max	Max	Stress

of memory to handle a large number of concurrent connections. That is, we cannot set *work_ mem= min* and *max_ conn=max*. As such, the final number of combinations was set to 5 (see Table II) and form the basis of the steps taken by STEM, which are:

(1) Initial DBMS setup upon initial workload;
(2) Tuned buffer pool size upon stress workload;
(3) Tuned DBMS upon a minimal amount of transactions;
(4) Tuned DBMS upon stress workload up to the DBMS performance limit;
(5) Tuned DBMS upon stress workload beyond the DBMS performance limit (up to the performance degradation);

The first step aims at seeking for defects related to any DBMS functionality, installation defects, and misconfiguration, not necessarily related to the DBMS installation package. The goal of the second step is to establish the degradation baseline for the DBMS, which is done by setting the admission control to its minimum. The third step aims at validating functional aspects after a tuning phase. For instance, to validate whether the observed value along testing corresponds to the one expected after tuning. The purpose of the fourth step is to search for defects related to the DBMS performance limit (not yet the stress condition). To establish the DBMS performance limit, we monitor the health of the DBMS and compare the results with the performance baseline established in the second step. The objective is to avoid pushing the DBMS beyond such limit that defines the boundary of the medium size workload. The goal of the fifth step is to push the DBMS beyond its performance limit, reproducing stress load conditions.

The second, fourth, and fifth steps must be executed in a distributed manner to reproduce a large-scale environment. In this context, the test driver cannot have any scalability issue for managing a large number of transactions without perturbing the test execution [de Almeida et al. 2010].

4.3 Database Specification

In STEM, the database schema and the transaction specification follow the TPC-C benchmark. TPC-C provides 9 tables that implement a wholesale supplier application and 5 types of transactions including: entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. Transactions are submitted by emulated users and the results are displayed at Remote Terminal Emulators (RTE). TPC-C also specifies the pacing of transactions by the emulated users.

We couple our test driver with a distributed implementation of the TPC-C benchmark [de Lima et al. 2009]. We chose this specific implementation, since it was implemented on top of peer-to-peer technology. Therefore, our test driver is allowed to scale-up the number of the emulated users.

4.4 Testing Architecture

The general distributed test driver architecture is composed of a test controller and distributed testers. The role of the controller is to synchronize the execution of test cases across the distributed testers (i.e., test drivers). The role of the testers is to invoke the instructions described within the test cases at the interface of the SUT. The instructions are typically calls to the SUT interface wrapped up in test case actions (e.g., transactions).

Testing a distributed system in realistic conditions requires the distribution of the testers. Therefore, the ability to distribute and coordinate a large number of testers multiplies the potential load we can generate to progressively push a system out of its boundaries. This is an important feature that we exploit in this article.

In STEM, the execution of the tests is based on the Peerunit[5] testing architecture [de Almeida et al. 2010; de Almeida et al. 2010] for reproducing large-scale workloads. Peerunit is deployed on a set of nodes containing one controller and a set of distributed testers. In our implementation, a tester reproduces an emulated user submitting transactions to the DBMS interface. The controller synchronizes the execution of the transactions through a distributed message exchange algorithm deployed across multiple testers.

In Peerunit, test cases are implemented in Java and deployed on all available testers. Each test case is implemented as a class with a set of actions implemented as *annotated* methods, i.e., methods adorned with a particular metatag, or *annotation*, which provides coordination information. During a test, the DBMS is exercised by a sequence of actions. Usually, an action has an input data set and generates an output data set, which can be both stored in text files or in a database. For instance, inserting data into the database and then retrieving such data.

4.5 STEM Versus Benchmarks

DBMS are generally evaluated through benchmarks, which provide a metric to compare performance, not to find defects. STEM provides a testing methodology that aims at finding defects related to a stressing workload.

The key difference between benchmarking and STEM is the nature of the diagnosis, which is performed by isolating the various knobs (i.e., configuration and test data workload) that play a role in defect detection. With STEM, the incremental steps are used to tune these different knobs, and this tuning allows a more precise diagnosis of what causes a given defect to appear. Actually, the diagnosis task is guided thanks to which steps provoke a defect to appear. One defect is related to stress load, in that case the cause of the defect is not to be found in the OS, but in the internal code of the DBMS that is supposed to manage the overflow of transactions.

5. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of STEM. Our objective is twofold: (i) validate STEM with two popular DBMS, and (ii) validate our testing hypotheses. In the experiments, we used a leading commercial DBMS, denoted by DBMS-X, and PostgreSQL version 8.3.

We chose an older version of PostgreSQL to validate if STEM is able to reveal defects undetected in an established bug-list.

PostgresSQL is studied first for two main reasons: (i) the source code is accessible making it easier for us to play with it; (ii) the DBMS serves as the storage system of a number of cloud computing solutions where load peaks are expected. Some of these solutions include: Greenplum, EnterpriseDB Cloud Database, and “vPostgres”.

[5]http://peerunit.gforge.inria.fr/

Table III. PostgreSQL's Result Overview

Step	**Number of Transact.**	**Completed Transact.**	**Rejected Transact.**	**Execution Time (sec)**	**Response Time (sec)**	**Linux CPU Usage (%)**	**Linux Memory Usage (MB)**
1	100	100	0	1	≈ 0.5	25	100
2	10,000	585	9,415	8	≈ 1	57	100
3	2,000	1,404	596	2	≈ 1	55	300
4	20,000	2,199	17,801	15	≈ 2	87	450
5	200,000	5,583	194,417	51	≈ 5	90	500

Table IV. DBMS-X's Result Overview

Step	**Number of Transact.**	**Completed Transact.**	**Rejected Transact.**	**Execution Time (sec)**	**Response Time (sec)**	**Linux CPU Usage (%)**	**Linux Memory Usage (MB)**
1	100	100	0	1	≈ 0.2	50	515
2	10,000	957	9,043	6	≈ 0.8	80	1,000
3	2,000	2,000	0	2	≈ 0.6	70	825
4	20,000	13,479	6,521	8	≈ 1	80	1,000
5	200,000	30,784	169,252	20	≈ 1.1	80	1,000

During the PostgreSQL tests, we couple the experiments with the GNU/gcov code-coverage tool, to understand the consequences of each step on the execution of the DBMS. The coverage is important to demonstrate that an incremental test methodology helps reaching portions of the code that are usually not exercised with functional tests. The coverage analyzes three main packages: (i) *Freespace* that implements the seek for free space in disk pages; (ii) *Page* that initializes pages in the buffer pool; and (iii) *Manager* that implements the shared-row-lock.

This section is organized as follows. The next subsection describes our implementation in a cluster. Section 5.2 describes our experiments based on incremental testing. Section 5.3 discusses the influence of experimental variables with the couple PostgreSQL/FreeBSD. Section 5.4 discusses the experimental results.

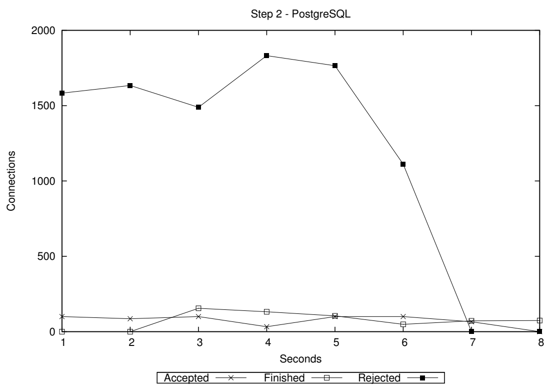
5.1 Cluster Configuration

All of our experiments were conducted on a cluster machine of the Grid5000 platform[6]. We used 11 "Sun Fire X2200 M2" machines connected by Gigabit Ethernet, where each node was configured with 2 duo-core AMD Opteron 2218 at 2.613GHz and 8GB of main memory. We used one node to run exclusively the DBMS server and ten nodes to run the clients, where each client is managed by a Peerunit tester[7]. To avoid performance interference, the clients and the server were executed on separate nodes, even for the smaller experimentations. To evaluate a possible interference of the OS on the results (Hypothesis 2), the server node ran the experiments with two different OS: GNU/Linux Debian 5.0.9 and FreeBSD 8.2. All client nodes ran GNU/Linux Debian 5.0.9. In all experiments reported in this article, each tester was configured to run in its own Java Virtual Machine (JVM). The cost of test coordination was negligible and is not discussed in this article (see the work of de Almeida et al. [2010] for a complete discussion on test coordination overhead).

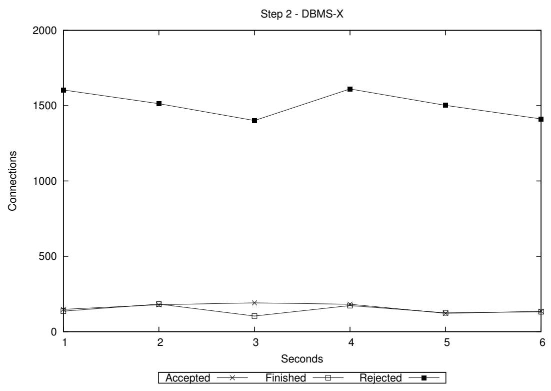
5.2 Incremental Testing

We conduct the experimentation through the 5 steps of STEM for searching load-related defects. Before every step execution, the database is recreated and reloaded to ensure independence between test executions and to be in compliance with repeatable automated testing. Tables III and IV summarize the results presenting the workload, the number of completed and rejected transactions, the testing execution time, the average response time for a single transaction and the resource consumption

[6]`http://www.grid5000.fr`
[7]The implementation is available for download at: http://web.inf.ufpr.br/ealmeida/research/tpc-c-c3sl



(a) PostgreSQL



(b) DBMS-X

Fig. 1. Number of Different Connections vs. Elapsed Time in Degradation Baseline Test (Step 2)

of the GNU/Linux OS. Both tables present only the GNU/Linux's results, since DBMS-X presented a steady resource consumption during testing.

5.2.1 *Step 1*

Objective: . Ensure correctness of the installation.

Parameters: . $|T| = 1$; $Max\_conn = 100$ (default); $Workload = 100$;

The objective of this step is to ensure the proper installation of the DBMS, which remains with its default values. In addition, we test the DBMS functionalities related to the transactional workload (i.e., the classical functional test). The expected result for this test step is the DBMS accepting and completing all the requested transactions.

In PostgreSQL, the default values for *Maximum Connections* and *Workload* parameters are both set to 100, with a similar configuration for DBMS-X. Both DBMS pass this first test step in which 100 transactions are submitted and all of them are accepted and completed successfully, with an average response time of 0.5 seconds for PostgreSQL and 0.2 seconds for DBMS-X.

5.2.2 *Step 2*

Objective: . Establish the degradation baseline.

Parameters: . $|T| = 10$; $Max\_conn = 100$ (default); $Workload = 10,000$;

In the second step, we allocate a larger amount of memory and workload for searching the performance boundary and the response time degradation. The expected result is the DBMS accepting the concurrent requests up to their setup values and refusing the exceeding ones, occasionally issuing a default error message[8].

In this step, PostgreSQL takes approximately 8 seconds to execute while DBMS-X takes 6 seconds. This execution time relates the execution of the entire test step: the submission of all the requests and the retrieval of all the responses from the DBMS, which can be either a transaction commit/abort or a connection rejection.

[8]In PostgreSQL, when the number of requests exceeds the configured value, it issues the following message: *"Sorry too many clients already"*.

During the test, ten testers ($|T| = 10$) submit a number of 10,000 transaction requests. The DBMS were configured to accept at most 100 concurrent requests. Figure 1 shows that both DBMS correctly accept a number of requests within their setup value, as expected. In spite of the high concurrency, no accepted transaction is aborted.

In PostgreSQL, we observe that the workload has a direct impact on the code coverage (see Figure 5). Indeed, as the workload increases, the code coverage of three packages also increases. First, in the "freespace" package, when the load increases, we observe that PostgreSQL takes more time to find free space. Second, in the "page" package, which initializes more pages. Third, in the "transaction manager" package, where the increase of code coverage is expected due to the a larger number of concurrent transactions waiting for the tuples to be unlocked.

In both DBMS, we do not observe any performance degradation. All transactions finish in less than one second. This behavior is also expected, since the raise of the load is not sufficient to overtake any resource of the system, as we will see in Section 5.3. Since this is the initial phase of STEM and the DBMS behave as expected, the computed response time becomes the *baseline time* for the subsequent steps.

5.2.3 *Step 3*

Objective: . Ensure the correctness of the tuning knob.
Parameters: . $|T| = 1$; $Max\_conn = 2,000$; $Workload = 2,000$;

In the third step, the objective is to ensure that the tuning knobs are correct, based on the results of the previous step. Tuning consists of setting the DBMS knobs to their maximum values, in function of the available resources.

In DBMS-X, little intervention was required to tune CPU, memory and storage. In fact, all the major commercial DBMS provide automatic resource management features to minimize human intervention [Abouzour et al. 2011]. A required intervention was tuning the *Maximum Connections* knob to cope with $2,000$ concurrent requests. The results show that DBMS-X passes the test accepting and committing all transactions with low response time degradation.

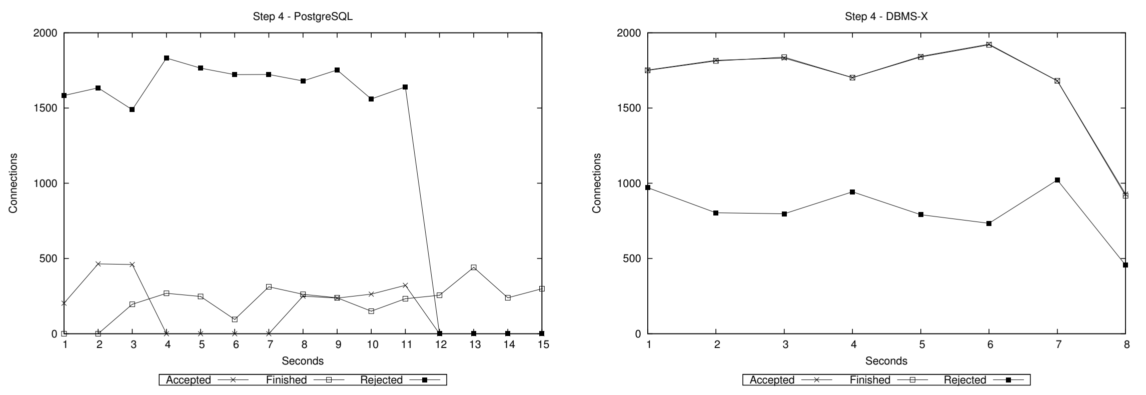
In the case of PostgreSQL, the *Maximum Connections* knob was fixed in 2,000 concurrent connections, which is the expected value for this test. In fact, the setup value of this knob depends on some of the *Buffer Pool* variables, which correspond to the size of the shared buffers used by each new connection. We set the shared buffers parameter with respect to the rules of thumb of the DBMS.

From this third step on, we set the shared buffers parameter to the maximum value of 32,768. Every time this parameter is set, we require to evaluate the correlated kernel parameter *SHMMAX*. This kernel parameter specifies the largest shared memory segment size for both OS, Linux and FreeBSD. According to the PostgreSQL documentation, the following formula is recommended to calculate this value:

$$SHMMAX = (250kB + 8.2kB * shared\_buffers + 14.2kB * max\_conn)$$

In this third step, PostgreSQL took 2 seconds to treat all the 2,000 submitted transactions, preserving the same response time observed at step 2. However, it generated an important number of 596 error messages during the execution: 231 in the first second and 365 in the second one. We do not observe any degradation on the OS resource consumption. From the functional point of view, this is considered as a *defect*, since the observed number of connections differs from the expected one.

Indeed, we observe an increasing number of new backend processes created by each new transaction request. To create a backend process, besides getting a piece of shared-memory, PostgreSQL manages



(a) PostgreSQL (b) DBMS-X

Fig. 2. Number of Different Connections vs. Elapsed Time Under Robustness Test (Step 4)

the state of each process within an array. When it takes a large number of requests, due to the *Maximum Connection* setup, the backend processes fill out the array and eventually overtake its limit. Consequently, the new backend processes are dropped and the objective of *Maximum Connections = 2,000* is never reached (Hypothesis 3 validation).

Another important result is the test boundary established in this step. When $|T| = 1$, the submitted load is bounded by the resources of the lone tester machine, which can only submit 1,000 requests per second without compromising its performance. While we could increase the $Workload$, the results of PostgreSQL would not be different from the ones presented here. To effectively reach a larger number of $Workload$ requests and reach a stress load, a distributed test approach is necessary.

5.2.4 *Step 4*

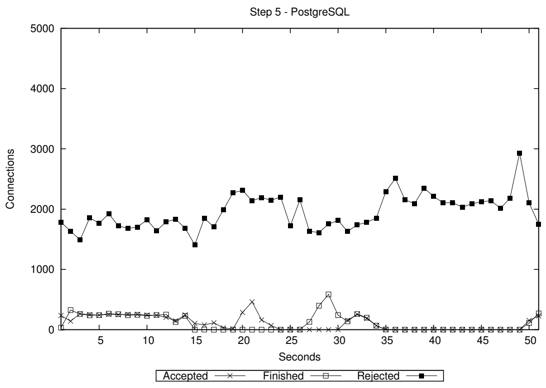
Objective: . Robustness Test.
Parameters: . $|T| = 10$; $Max\_conn = 2,000$; $Workload = 20,000$;

In this step, we explore the robustness of the DBMS upon a stress load with *Maximum Connections = 2,000.* As for the second step, the expected result is the DBMS finishing all the accepted transactions with the remaining average response time around 1 second.

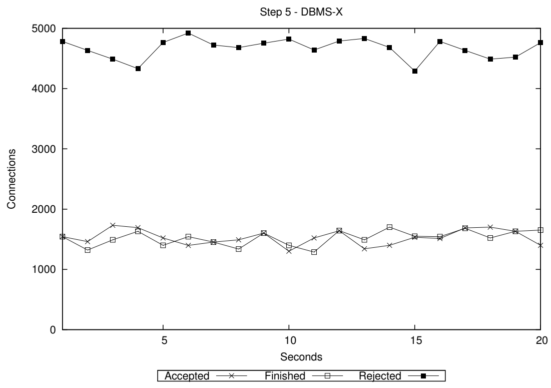
From the previous results, we conclude that PostgreSQL cannot handle $2,000$ concurrent transactions. The entire test ran during 15 seconds and the submission of requests lasts up to 11 seconds. Figure 2(a) shows a nonlinear behavior of PostgreSQL, which initially accepts around 200 requests and takes 3 seconds to treat them. PostgreSQL accepts almost 500 requests and treats them in the following seconds. Analyzing the OS health, we diagnosed that the system becomes CPU-bound while handling the submitted load. However, the memory usage remains low, since the accepted requests never reach the expected *Maximum Connections.* At the end, the average response time is twice as much as the *baseline time* (Step 2).

We also diagnosed that the accepted requests produce more modifications in disk pages. When some request performs a modification, PostgreSQL grants an exclusive lock. Then, the backend process, which is responsible for such request, requires to lock other backends, trying to extend the disk pages concurrently. This process is executed at every locking attempt, causing an important number of context swaps, one for each request.[9] From the presented slopes, the number of accepted requests are

[9]The requests are managed into the buffer manager.



(a) PostgreSQL



(b) DBMS-X

Fig. 3. Number of Different Connections vs. Elapsed Time Under Stress Test (Step 5)

not too different from Step 2. However, the behavior of PostgreSQL becomes chaotic without any stability in the response time. From the testing point of view, this is a *defect*: PostgreSQL should have a similar behavior as the one observed at Step 2, rejecting the exceeding requests and taking care of the accepted ones.

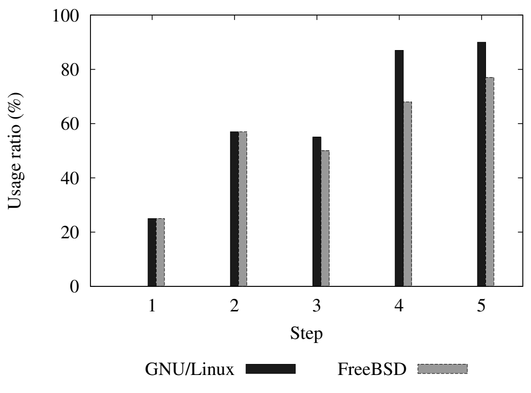
In DBMS-X, the entire test run in 8 seconds with a stable response time for transactions. DBMS-X finishes all the accepted transactions within an average response time of 1 second (in Figure 2(b) the behavior of the "Finished" slope follows the "Accepted" one). However, DBMS-X only accepts 67.39% of the concurrent requests, never reaching the expected *Maximum Connections*, even with the same resource consumption of the earlier steps. When DBMS-X gets closer to the expected *Maximum Connections*, it starts issuing a network error message. In DBMS-X's documentation, this message is due to two main reasons: a network misconfiguration (e.g., wrong SQL URL, port, or IP address) or a network backend process issue (e.g., offline process, network port is closed, network or hardware failure). In earlier steps this message is never issued, therefore from a functional point of view, such inability of reaching the expected *Maximum Connections* is a *defect.*

5.2.5 *Step 5*

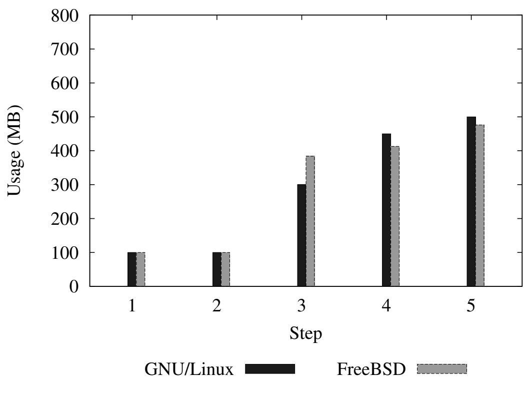
Objective: . Stress Test.
Parameters: . $|T| = 10$; $Max\_conn = 2,000$; $Workload = 200,000$;

Figures 3(a) and 3(b) illustrate the elapsed times for the experiments. In PostgreSQL, similarly to Steps 2 and 4, we observe that the number of accepted and finished transactions were much smaller than the rejected ones (2.79% accepted/finished with an average response time of 5 seconds). We notice that it accepts and finishes a uniform number of transactions until the 14th second when a thrashing state begins (i.e. in this state, a large amount of computer resources is used to do a minimal amount of work, with the system in a continual state of resource contention).

In addition, we observe at the 35th second that PostgreSQL enters a failure state and refuses any new connection attempt. From code coverage analysis, we notice that two particular code parts are more exercised compared to Step 4: the free space check on pages and the buffer page code. When a transaction is about to update the database, PostgreSQL tries to find free space in data files either by seeking pages with enough free space or extending the data file itself with more pages. At stressing stages, PostgreSQL is not capable of managing a large number of requests for finding such free space. Hence, it starts issuing "InvalidBuffer" messages. This behavior is considered as a *defect*, since the disk has available space with low throughput of 18M of writing operations, which is not too different



(a) CPU Usage



(b) Memory Usage

Fig. 4. PostgreSQL's Resource Consumption at Each Step

compared to Step 3 with 14M of writing throughput. Furthermore, this result validates our claim that STEM is helpful to push the SUT to its performance limit and reveal load-related defects (Hypothesis 1 validation).

In DBMS-X, we observe a similar response time compared to Step 4 of 1.1 seconds, even accepting only 15.39% of the requests. DBMS-X shows the same behavior as Step 4 rejecting new requests. It also does not reach the expected *Maximum Connections* of 2,000, but provides a low response time for transactions and never switches to a thrashing state.

5.3 Influence of Experimental Variables

Along with the experimentation, we analyze the health of the OS as well as the network traffic to evaluate whether they influence the testing results (Hypothesis 2 validation). We use the "dstat", "slabtop", and "netstat" tools to provide complete information about CPU, memory (including kernel cache information), network, and disk consumption. In the DBMS-X tests, we observe a steady resource consumption, especially at Steps 2, 4, and 5 that provide heavier workloads. Therefore, we consider they did not interfere in the test results.

In the PostgreSQL tests, we execute it on top of two different OS (GNU/Linux and FreeBSD), since we do not observe a steady resource consumption. Both OS are known to be robust upon heavy load conditions. Table III shows that the system becomes CPU-bound only upon peak conditions at Steps 4 and 5, with PostgreSQL rejecting a large number of requests and delaying the response time. Figure 4(a) shows that the CPU consumption increases from Step 1 to 5, 360% on GNU/Linux and 308% on FreeBSD. This growth is not only due to admission control, but also to manage a larger number of accepted transactions. PostgreSQL uses a multiprocess model to handle transaction requests. Naturally, the more requests it receives, the more CPU it consumes.

We investigate whether some other OS resources are contributing to the faulty behavior. In fact, the memory consumption increases at different steps (see Figure 4(b)), but the overall consumption is not high compared to DBMS-X. Actually, the CPU was the only resource affected by the incremental workload due to context switches in transaction management, while the consumption of the other resources remained constant during the tests. In addition, networking results present a small deviation with only a small impact of received and sent packages. For instance, the amount of sent packages from Steps 3 to 5 increases from 250K to 320K. The I/O throughput does not increase too much as well, with writing operations ranging from 14M at Step 3 to 18M at Step 5. With memory essentially free, low I/O throughput and networking, we observe the testing results are not influenced when

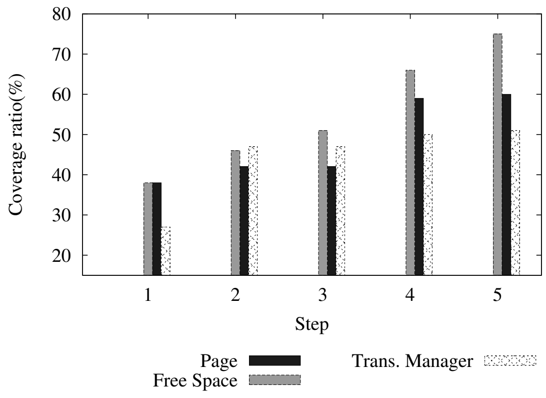


Fig. 5. PostgreSQL's Code Coverage

STEM is incremented.

5.4 Discussion

Our experiments validated Hypothesis 1, which states that we can analyze the test results through log files (DBMS-X) or code coverage (PostgreSQL).

Through log files, we treat the DBMS as a black-box by analyzing its output traces with respect to the documentation. In addition, we analyze the expected performance metrics compared to the observed one at different steps. In this way, we identified the DBMS-X's defect on the network backend process.

The code-coverage approach provides a fine grained manner to reach and understand the source of the defect. We also use it for validating that STEM exercises parts of the system that would never be covered otherwise. As illustrated by Figure 5, code-coverage increases mainly at Steps 4, and 5 where a large number of transactions were submitted. In particular, STEM allows to force many robustness mechanisms to be exercised and tested. These limit cases are precisely those that may occur under real large-scale conditions.

STEM presents three advantages compared with the current testing approaches that are similar to Steps 1 to 3 executed directly and without an incremental approach. First, the methodology for implementing and executing the tests is simple. Second, we can diagnose the root of the defects, and under which precise execution conditions the defect can be reproduced. The isolation of the execution conditions that may provoke a defect is especially crucial for diagnosing a problem and fixing it. Third, the incremental approach was able to expose different defects at each step.

Empirical Hypothesis 1 states that different workloads exercise different parts of the code. This hypothesis is verified for PostgreSQL along with all test steps in two ways, one with the original source-code and another with an instrumented source-code, which generated code coverage information. After analysis, this information revealed that the more the workload increases, the more code is exercised. In DBMS-X, we verified this hypothesis by analyzing whether different log file entries appear. Our results show that the network backend process only issues error entries when treating large workloads.

Empirical Hypothesis 2 states that, if the DBMS manifests an unexpected behavior on different OS, or the OS resource consumption remains steady, then the OS has no effect on the test results. This hypothesis is verified for PostgreSQL, which had similar behavior on Linux and BSD. Moreover, system monitoring tools revealed a steady resource consumption for network, disk I/O, and memory. One could argue that Linux and BSD share part of their code, specially network, and a same defect could be present on both OS. Whether this could be possible, it is rather unlikely, since the PostgreSQL unexpected behavior revealed in Step 4 is not reproduced by DBMS-X.

Empirical Hypothesis 3 states that the DBMS under test should not use all allocated resources. The rationale behind this hypothesis is that we should not stress the OS, otherwise identifying the source of an abnormal behavior would be complex. This hypothesis is verified at the stress test (Step 5), where 200,000 transactions are sent. Monitoring tools reveal that both DBMS never reach their configuration limits, whereas the OS performance presents low impact on the overall results.

6. CONCLUSION

This article introduced STEM, a stress testing methodology for transactional DBMS built on top of empirical software engineering hypotheses. We can draw many interesting findings from the results. First, stress testing requires a distributed testing approach, where multiple test drivers are deployed across a cluster machine to submit a large number of transactions. The single driver approach, used in the related work, bounds the size of the load, thus leading to a classical functional test. The execution of STEM is straightforward, but in contrast to related work, requires many machines to execute. Actually, a classical functional test would only discover the PostgreSQL's defect found at an early step of STEM without any trace of a lower/upper bound number of accepted or rejected transactions. The defect happens due to a size limitation of an internal array that is responsible for managing backend processes. Upon concurrent requests, the array fills out quickly, thus preventing the treatment of new processes (including new transaction requests). A better approach would bound the size of such array by a related DBMS configuration parameter as expected from the DBMS documentation.

Next, the incremental approach was able to expose different defects in PostgreSQL at each step and a defect within the network backend process of a leading commercial DBMS. When DBMS-X gets closer to the expected maximum number of concurrent connections, it starts issuing an unexpected network error message. From its documentation, such message is due to network misconfiguration (e.g., wrong SQL URL, port, or IP address) or a network backend process issue (e.g., offline process, network port is closed, network or hardware failure).

Although we are not surprised by the source-code covered by our approach given that different steps exercised more features of the DBMS, we were impressed by how the increasing load helps to expose the related defects (also detected in a black-box testing manner through log file analysis). This approach turns out to be an important tool for DBMS test engineers due to the load generated by an increasing number of connected devices. Naturally, we conclude that STEM should be considered in DBMS's testing campaigns.

REFERENCES

Abouzour, M., Bowman, I. T., Bumbulis, P., DeHaan, D., Goel, A. K., Nica, A., Paulley, G. N., and Smirnios, J. Database Self-management: taming the monster. *IEEE Data Engineering Bulletin* 34 (4): 3–11, 2011.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing.* New York, NY, USA, pp. 143–154, 2010.

de Almeida, E. C., Marynowski, J. E., Sunyé, G., Le Traon, Y., and Valduriez, P. Efficient Distributed Test Architectures for Large-Scale Systems. In *International Conference on Testing Software and Systems.* Natal, Brazil, pp. 174–187, 2010.

de Almeida, E. C., Sunyé, G., Traon, Y. L., and Valduriez, P. Testing Peer-to-Peer Systems. *Empirical Software Engineering* 15 (4): 346–379, 2010.

de Lima, M. R., Sunyé, M. S., de Almeida, E. C., and Direne, A. I. Distributed Benchmarking of Relational Database Systems. In *Proceedings of the Joint International Conferences on Asia-Pacific Web Conference and Web-Age Information Management.* Suzhou, China, pp. 544–549, 2009.

Deng, Y., Frankl, P. G., and Chays, D. Testing Database Transactions with AGENDA. In *International Conference on Software Engineering*. St Louis, Missouri, USA, pp. 78–87, 2005.

Florescu, D. and Kossmann, D. Rethinking Cost and Performance of Database Systems. *ACM's Special Interest Group on Management Of Data Record* 38 (1): 43–48, 2009.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. *Fundamentals of Software Engineering (2. ed.)*. Prentice Hall, 2003.

Gu, Z., Soliman, M. A., and Waas, F. M. Testing the Accuracy of Query Optimizers. In *International Workshop on Testing Database Systems*. Scottsdale, Arizona, USA, pp. 11:1–11:6, 2012.

ISO9646. Open Systems Interconnection Conformance Testing Methodology and Framework, 1991.

Jain, R. The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation and modeling (book review). *SIGMETRICS Performance Evaluation Review* 19 (2): 5–11, 1991.

Kuhn, R., Lei, Y., and Kacker, R. Practical Combinatorial Testing: beyond pairwise. *IT Professional* vol. 10, pp. 19–23, 2008.

Ostrand, T. J. and Balcer, M. J. The Category-Partition Method for Specifying and Generating Fuctional Tests. *Communications of the ACM* 31 (6): 676–686, 1988.

Soror, A. A., Minhas, U. F., Aboulnaga, A., Salem, K., Kokosielis, P., and Kamath, S. Automatic Virtual Machine Configuration for Database Workloads. *ACM Transactions on Database Systems* 35 (1): 7:1–7:47, 2008.

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. The End of an Architectural Era (it's time for a complete rewrite). In *Very Large Data Bases*. pp. 1150–1160, 2007.

Storm, A. J., Garcia-Arellano, C., Lightstone, S., Diao, Y., and Surendra, M. Adaptive Self-tuning Memory in DB2. Seoul, Korea, pp. 1081–1092, 2006.

Xiong, C. Inferences on a Simple Step-Stress Model with Type-ii Censored Exponential Data. *IEEE Transactions on Reliability* vol. 47, pp. 142 – 146, 1998.