"Overloaded!" — A Model-Based Approach to Database Stress Testing

Jorge Augusto Meira[1,2(✉)], Eduardo Cunha de Almeida[2], Dongsun Kim[1], Edson Ramiro Lucas Filho[2], and Yves Le Traon[1]

[1] SnT Research Center, University of Luxembourg, Luxembourg City, Luxembourg
{jorge.meira,dongsun.kim,yves.letraon}@uni.lu
[2] C3SL - Federal University of Paraná, Curitiba, Brazil
{jmeira,eduardo,erlfilho}@inf.ufpr.br

Abstract. As a new era of "Big Data" comes, contemporary database management systems (DBMS) introduced new functions to satisfy new requirements for big volume and velocity applications. Although the development agenda goes at full pace, the current testing agenda does not keep up, especially to validate non-functional requirements, such as: performance and scalability. The testing approaches strongly rely on the combination of unit testing tools and benchmarks. There is still a testing methodology missing, in which testers can model the runtime environment of the DBMS under test, defining the testing goals and the harness support for executing test cases. The major contribution of this paper is the MoDaST (**Mo**del-based **Da**tabase **S**tress **T**esting) approach that leverages a state transition model to reproduce a runtime DBMS with dynamically shifting workload volumes and velocity. Each state in the model represents the possible running states of the DBMS. Therefore, testers can define state goals or specific state transitions that revealed bugs. Testers can also use MoDaST to pinpoint the conditions of performance loss and thrashing states. We put MoDaST to practical application testing two popular DBMS: PostgreSQL and VoltDB. The results show that MoDaST can reach portions of source code that are only possible with non-functional testing. Among the defects revealed by MoDaST, when increasing the code coverage, we highlight a defect confirmed by the developers of VoltDB as a major bug and promptly fixed.

1 Introduction

Scalable and high performance data processing is one of the key aspects for successful business operations as the volume of incoming transactions is getting larger for most application areas. Over the last 40 years traditional "one-size-fits-all" Database Management Systems (DBMS), such as DB2, Oracle, PostgreSQL, have been successful in processing transactions. However, the recent growth of the transaction workload (e.g., Internet, Cloud computing, Big Data) is challenging these DBMS requiring revisiting their kernel. Even new DBMS are being designed ground up to better tackle these workloads.

S. Hartmann and H. Ma (Eds.): DEXA 2016, Part I, LNCS 9827, pp. 207–222, 2016.
DOI: 10.1007/978-3-319-44403-1_13

Although the development agenda goes at full pace, with the recent appearance of a great deal of new DBMS, the testing agenda does not keep up, specially for validating non-functional requirements, such as performance, robustness and scalability. With the increasing demand in volume and velocity of transactions, many load conditions challenge the DBMS in unexpected ways that the state of the art in testing tools cannot exercise. Different bugs can be found in the literature describing that the root cause is linked to different conditions of transient load shifts or sudden spikes. The result in the DBMS can be treacherous leading to a number of non-functional failures, such as: poor performance query plans [1], backpressure[1], lock escalation (for lock-based mode) [2], poor performance estimations [3], performance degraded mode and load shedding [4]. Many of these failures are also called "Heisenbugs" [5], because the root cause are not easy to detect and may elude the bug-catcher for years of execution.

1.1 Motivation

The bug-catching task becomes even harder when the existing testing methodologies for transaction processing only validate functional requirements [6,7]. The validation of non-functional requirements is still an open issue and strongly rely on the combination of unit testing tools[2] (e.g., Jepsen, JUnit, Jmeter, PeerUnit [8]) and benchmarks to reproduce specific workloads (e.g., TPC-like, YCSB).

The main problem with this combination is that it is strictly based on tools and does not adhere to a general methodological testing approach. In general, this combination has to be conservative to eke out the "ideal" testing environment: test cases mimic any benchmark workload and then execute on top of an unit testing tool. However, the expected environment grounds testing with a proper methodological approach to define the testing goals and the harness support for executing test cases. Writing and executing test cases come later.

The major contribution of this paper is such a methodological approach that can eventually be implemented on top of any unit testing tool with the benchmark of your choice. Figure 1 shows the impact on PostgreSQL of executing test cases with and without a testing methodology. The impact is measured by the code coverage ratio of our methodological approach and the same test case reproducing the TPC-C benchmark workload on top of a unit test tool without following any testing methodology. First, we see that the impact of shifting the transaction load in PostgreSQL can only be analyzed when testing is driven by a methodological approach. Second, we notice that the load shifting exercises PostgresSQL in different code portions. More interestingly, when the DBMS is upon heavy loads (rightmost bar), the throughput goes down, but exercising almost 60 % of the source code of the kernel (12 % more than the steady condition).

This result shows that even the kernel of a mature DBMS, such as PostgreSQL, is not acquainted to non-functional testing, which would reveal the

[1] https://voltdb.com/docs/UsingVoltDB/DesignAppErrHandling.php.
[2] VoltDB testing: https://voltdb.com/blog/how-we-test-voltdb.

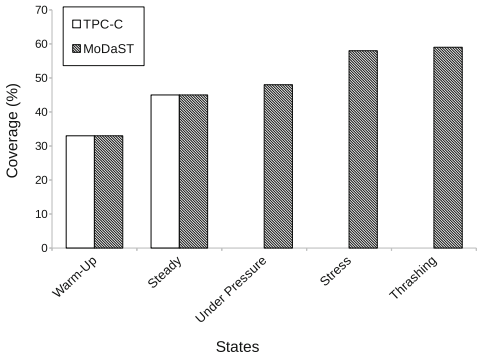

Fig. 1. Example of inherent limitations of the existing testing tools.

bugs that we discuss in this paper (see Fig. 2). To come up with a general non-functional testing approach for transaction processing in the Big Data era, it is important to define a running model of the Database System Under Test (DUT) that allow reproducing and harnessing load shifts.

1.2 Contribution

This paper presents, MoDaST (**Mo**del-based **Da**tabase **S**tress **T**esting), a novel methodological approach to DBMS stress testing. This approach focuses on testing scalability and performance of DBMS with dynamically changing load levels by using a test model for database systems. The approach leverages a state machine model with observable runtime states: warm-up, steady, under-pressure, stress, and thrashing. The model allows us to infer and explore internal states of the DUT even if black-box testing is only available. The observable states can basically be used for guiding the testing goals with test cases forcing the state transitions. More importantly, MoDaST allows reproducing the state transitions for regression or to figure out what is the exact condition that revealed a bug.

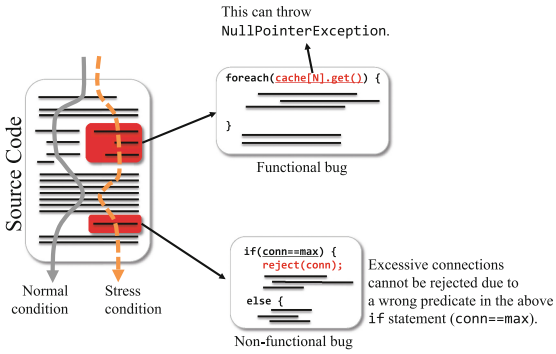

Fig. 2. Conceptual execution paths under normal and stress conditions.

To evaluate MoDaST and show that it can be put to practical application, we applied it to two different classes of real-world DBMS (i.e., SQL and NewSQL) collected from open source projects: PostgreSQL and VoltDB. We designed a distributed stress testing environment by using a cluster facility and a distributed testing driver to shift the submission volume and velocity of the workload. We collected performance monitoring data and code coverage to figure out whether there is any potential defect. We also conducted a comparative study between MoDaST and unit testing running the TPC-C benchmark to find out which one can correctly test different behaviors of DBMS and cover more source code.

The results of the experiments showed that our approach successfully drove the DUTs into the different states specified in the test model. MoDaST found out that DUTs actually follows the model by observing performance data. In addition, the results revealed that our approach explored different performance behaviors and increased test coverage up to 20 % in certain code packages of PostgreSQL and 12 % for VoltDB compared to the baseline technique. Newly covered lines by MoDaST exposed three new bugs. In particular, one of the bugs had significant impact on VoltDB by affecting not just non-functional but also functional requirements upon heavy load conditions. This bug was confirmed and promptly fixed by the VoltDB hackers after our reporting[3].

Overall, this paper makes the following contributions: **1- Database state model**: We designed a running model to infer the internal states of DBMS based on performance observations. Among different possible states, our model detects performance loss and thrashing states at runtime. **2- MoDaST, a model-based DB stress testing approach**: We introduce a novel testing approach to force state transitions in the model by shifting the transaction loads. The state transitions allow exercising different source code portions of the DBMS that would never be exercised by single test cases of unit testing tools. **3- Empirical evaluation**: We present empirical evaluation results by applying MoDaST to popular open-source DBMS. Based on the evaluation results, we identified and reported potential bugs. One of them was confirmed as "major bug" and promptly fixed by the core developers.

The remainder of this paper is organized as follows. Section 2 we discuss the related work. Section 3 describes our model-based approach to database stress testing. Section 4 we present empirical results stress testing two popular DBMS. Finally, Sect. 5 concludes with future directions.

2 Related Work

Stress testing is designed to impose heavy loads such as HTTP requests or database queries at the same time to ensure the reliability of the system In DBMS, performance/stress testing validates the system from different angles. Commonly, this validation is executed through a benchmark pattern to reproduce a production environment. Since the DebitCredit benchmark [9], several benchmarks were presented along the last decades. These benchmarks focus

[3] https://issues.voltdb.com/browse/ENG-6881.

on comparing metrics (e.g., response time, throughput, and resource consumption) [3]. The TPC-like benchmarks offer different workload levels to evaluate databases from two perspectives: OLTP or OLAP. In contrast, the Yahoo Cloud Serving Benchmark (YCSB) [10] is designed to evaluate four specific features of distributed databases: Performance, Scalability, Availability and Replication. There are another type of benchmarks focusing on DBMS availability: R-cubed [11], DBench-OLTP [12], Under Pressure Benchmark [13].

Some of existing performance testing tools attempt to test database systems under different levels of workload. Jepsen[4], Hammerora[5], AppPerfect[6] and Oracle Application Testing Suite[7] provide a test driver to build up test cases on top of TPC-like benchmarks. Agenda [7] provides its own methodology and test driver, but this tool can only generate functional test cases. JMeter is also a well known and widely applied load testing tool for different applications, including DBMS. But, it was designed to load test functional requirements.

The main disadvantage of these tools is the lack of a high level testing methodology, like MoDaST. In software testing, the testing methodology is the foundation over which the tools are used [14]. Otherwise, test cases will be narrowed to reproduce specific load conditions that cannot reflect a far more aggressive real-world production environment with load spikes and shifts after a while in steady condition state [1,4]. In addition, these tools cannot correlate performance loss and related defects to specific its root causes.

Finally, techniques to generate test cases can be used to boost testing results of MoDaST for specific testing goals. For instance, [15,16] presents a technique to generate queries with cardinality constraints for validating multidimensional histograms A complementary technique to generate test databases is presented in [17]. Although MoDaST is a testing model, rather than a data/query generation tool, it was built for validating write-mostly, while the mentioned techniques are meant to read-mostly database systems assessments.

3 Approach: MoDaST

This section describes our Model-based Database Stress Testing (MoDaST) approach. Figure 3 shows an overview of this approach. MoDaST consists of the Database State Machine (DSM) and a test driver. The DSM represents a set of observable states of a DBMS and its transition function. The test driver defines the load model of each state and commences performance testing by giving a specific load to the DUT. Then, the driver observes the current performance data of the DUT and figures out state transitions by giving the data to DSM. The remainder of this section details MoDaST.

[4] https://aphyr.com/tags/jepsen.
[5] http://hammerora.sourceforge.net/.
[6] http://www.appperfect.com/.
[7] http://www.oracle.com/technetwork/oem/app-test/index.html.

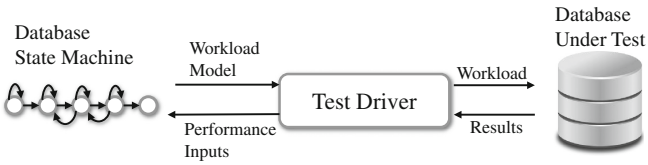

Fig. 3. Architectural overview of MoDaST.

3.1 The Database State Machine (DSM)

The DSM models how a DUT behaves at given workload levels. In particular, DSM focuses on representing observable states of a DUT with respect to performance (i.e., performance behaviors). The behaviors of a DUT can be represented by the following states: Warm-up (s_1), Steady (s_2), Under Pressure (s_3), Stress (s_4), Thrashing (s_5). We formally define the DSM and its corresponding states in Definition 1. Figure 4 depicts the DSM, the running states and transitions.

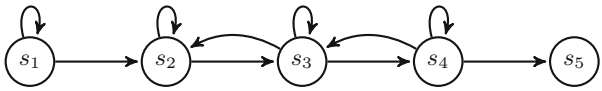

Fig. 4. The Database State Machine (DSM) and the observable states.

Definition 1. *The Database State Machine (DSM) denoted as T, is a 5-tuple* $(\mathcal{S}, s_1, \mathcal{F}, \beta, \tau)$ *where:*

- $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5\}$ *is a set of states,*
- $s_1 \in \mathcal{S}$ *is the initial state,*
- $\mathcal{F} \subset \mathcal{S}$ *is the set of final states, where* $\mathcal{F} = \{s_5\}$ *in DSM,*
- β *is the set of performance input defined by Definition 2,*
- τ *a state transition function defined by Definition 6.*

The detailed information about every state is available in Sect. 3.1. To describe each state in detail, it is necessary to define the performance input, β. Based on the performance input, the DSM determines state transitions, τ.

Performance Input: DSM takes three different performance input from a DUT to infer its current internal state. The input, β, is the set of (1) the performance variation, (2) the transaction efficiency, and (3) the performance trend.

Definition 2. *The* **performance Input***, denoted by* β*, is a tuple of three performance variables:* $\beta =< \Delta, \delta, \varphi¿$*, where* Δ *is the performance variation (Definition 3),* δ *is the transaction efficiency (Definition 4), and* φ *is the performance trend (Definition 5), respectively.*

The **performance variation** represents the stability of transactions treated by a DUT. This is denoted by Δ as shown in Definition 3. MoDaST makes n observations ($n > 1$) and computes the number of transactions treated per second (y) for each observation. For example, if $\Delta \to 0$, the DUT is processing a steady number of incoming transactions.

Definition 3. *The* **performance variation**, Δ, *is the dispersion of the number of treated transactions per second and formally defined as:*

$$\Delta = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \mu)^2}, \tag{1}$$

where $\mu = \frac{1}{n}\sum_{i=1}^{n}(y_i)$

The **transaction efficiency**, δ, is the proportion between the number of transactions treated by a DUT and requested by clients. This enables to define the upper bound number of transactions in concurrent execution with steady behavior. For example, if $\delta \to 1$ across a number of performance observations, the DUT is successfully treating most of transactions requested by clients.

Definition 4. *The* **transaction efficiency**, *denoted by* δ, *is the proportion of the transactions treated per second (y) by the number of transactions requested per second (z):*

$$\delta = \frac{y}{z} \tag{2}$$

The **performance trend**, φ, is a metric explaining the expected performance slope of a DUT within a certain size of a sliding window as described in Definition 5. As shown in Fig. 5, φ can be computed by the distance between the

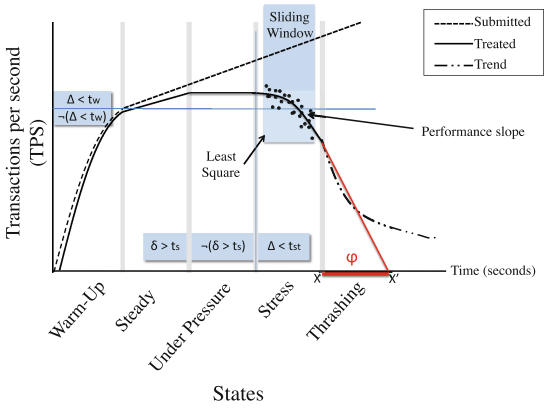

Fig. 5. DSM and its performance input. The X-axis is time in seconds and the Y-axis represents transactions per second. This shows relationships between performance input and states in DSM.

current time (observation) and expected time when the transaction efficiency of the DUT converges to 0 (i.e., $\delta = y/z = 0$, where $z \neq 0$). Section 3.2 describes how to compute the distance in detail.

Definition 5. *The* **performance trend** *is a function defined as*

$$\varphi = x' - x \tag{3}$$

where x is the current time and x' represents the point that the tangent line crosses the time axis.

States: The DSM models, with a state machine, the database performance states found in the literature in which non-functional bugs were reported [1–5]. The following paragraphs describe each state in detail.

State 1 — Warm-up: This state is the startup process of the DUT. In this state, the DUT initializes internal services such as transaction management service. Although some transactions can be treated during this state, performance is not stable since the DUT focuses on filling memory caches. DSM defines the Warm-up state by using performance variations (Δ in Definition 3).

The DSM infers that a DUT is in the *Warm-up* state if Δ is not converging to 0 after the startup of the DUT. In other words, $\neg(\Delta < t_w)$, where t_w is the warm-up threshold value. Otherwise (i.e., $\Delta < t_w$ holds), the transition to the next state (i.e., **Steady**) is triggered. Each DBMS has a unique t_w value. Section 4 explains how to determine the value.

State 2 — Steady: The DSM infers this state if the performance variation, Δ, is converging to 0. Once the DUT is in this state, it never comes back to the *Warm-up* state again since all the internal services are already initialized and running. In addition, the memory cache of the DUT is filled to provide the expected performance, which indicates that the DUT can correctly treat most of incoming transaction requested by clients in time. Specifically, this can be represented as $\delta > t_s$, where t_s is the steady threshold value. Each DBMS has a different value for the threshold that may vary based on the type of the expected workload and the available hardware environment.

State 3 — Under Pressure: This state implies that a DUT is on the limit of performance. The DUT goes to the state if δ approaches to zero, which means that a set of unexpected load is coming to the DUT. Unexpected loads include shifts and sudden spikes (e.g., Black Friday or Christmas) that affect performance [1,4,13]. In this state, the DUT can still deal with the similar amount of transactions processed in the previous state (*Steady*). However, it cannot properly treat a certain amount of transactions in time since the total amount requested by clients is beyond the limit of the DUT. Although this situation can be transient, it might need an external help from the DB administrator (DBA) to go back to *Steady*. For example, DBA can scale up the DUT's capability or set up the DBMS to reject a certain amount of the incoming transactions until the load decreases to an acceptable amount (i.e., $z \rightarrow y$ and $\delta > t_s$).

State 4 — Stress: a DUT goes into this state when the number of transactions requested by clients is beyond the performance limit. This state is different from the *Under Pressure* state since the performance variation (i.e., Δ) increases. The DUT in this state is highly vulnerable to crash if no external help is available. For example, the DBA should consider additional solutions such as adopting database replica, adding more cluster machines, or killing long running transactions (normally triggered by bulk loads). If an appropriate solution is performed, the DUT can go back to the *Under Pressure* state and $\Delta < t_{st}$, where t_{st} is the stress threshold value.

State 5 — Thrashing: This state represents that the DUT uses a large amount of computing resources for a minimum number of transactions. The DUT experiences resource contention and cannot deal with any new transaction in this state. In this state, it is no longer possible to come back to the previous one as any external intervention is useless. The DSM detects the transition to the *Thrashing* state if $\varphi < t_{th}$, where t_{th} is the thrashing threshold value. Predicting the thrashing state is explained in Sect. 3.2.

State Transitions: The state transition function, τ, determines whether the DUT changes its internal state based on observed performance data. This function takes *performance input* ($< \Delta, \delta, \varphi >$) from the test driver and gives the next state $s \in S$ as described in Definition 6. In each state, the DSM examines the current values of performance input and compares the values with threshold values[8] (i.e., t_w, t_s and t_{st}). Table 1 summarizes the threshold values.

Definition 6. *The state transition function, τ, is defined as:*

$$\tau : \mathcal{S} \times \beta \rightarrow \mathcal{S} \tag{4}$$

where $\forall s \in \mathcal{S}, \exists p \in \beta$ *and* $\exists s' \in \mathcal{S} | (s, p) \rightarrow (s')$.

Table 1. Threshold values for state transitions.

	Target state				
States	s_1	s_2	s_3	s_4	s_5
s_1	$\neg(\Delta < t_w)$	$\Delta < t_w$	-	-	-
s_2	-	$\delta > t_s$	$\neg(\delta > t_s)$	-	-
s_3	-	$\delta > t_s$	$\neg(\delta > t_s)$	$\Delta > t_{st}$	-
s_4	-	-	$\neg(\Delta > t_{st})$	$\Delta > t_{st}$	$\varphi < t_{th}$
s_5	-	-	-	-	$\varphi = 0$

[8] The values used in the experiments are specified in the Sect. 4 since it is variable depending on the DUT.

3.2 Predicting the Thrashing State

In addition to stress testing, MoDaST can predict crashes before a DUT actually goes into the *Thrashing* state. This indicates the time remaining until *out-of-service* and allows DBA to react before service failure.

The first step to predict the *Thrashing* state is computing the performance slope. MoDaST uses the Least Squares method [18] that approximates the relationship between independent (x) and dependent (y) variables in the form of $y = f(x)$. The testing time in seconds is denoted by x and y denotes the corresponding throughput at time x. It allows the computation of the three required coefficients (i.e., a_0, a_1 and a_2) for the quadratic function (i.e., performance slope): $f(x) = a_0x^2 + a_1x + a_2$

Our approach computes the coefficients by using recent p observations[9] of (x, y) (i.e., sliding window). The quadratic function estimates the performance slope as shown in Fig. 5. Once the performance slope is identified it is possible to calculate the derivative $f'(x)$ (i.e., tangent line), considering the current observation x_i. By using the tangent projection in the axis x, MoDaST can estimate the performance trend, φ, according to Definition 5. If the value is converging to the thrashing threshold (t_{th}), we assume that DUT may crash at any moment (i.e., transition from the stress to thrashing state).

3.3 The Test Driver

The goal of the test driver is to generate different load conditions and collect the performance input for the DSM. The test driver is built on top of the PeerUnit distributed testing framework [8]. PeerUnit allows building, coordinating and executing distributed test cases, which are key features for stress testing.

Since the performance of a DBMS can be affected by both the number of connections and transactions, it is necessary to test both the connection and transaction management modules by using two workload cases as follows: **Case #1**: The goal of this case is to submit a heavy load to the **connection** module of the DUT. The number of connections is gradually increased for each step. In this case, the driver submits only one transaction per connection; **Case #2**: The goal of this case is to submit a heavy load to the **transaction** module of the DUT instead of the connection module. The number of transactions is gradually increased for each step. In this case, the driver submits an increasing number of transactions per connection (i.e., fixed number of connections).

4 Empirical Evaluation

We applied our approach to two DBMS running TPC-C: VoltDB 4.5 and PostgreSQL 9.3. These subjects are selected for several reasons. First, both of them are ACID open-source RDBMS. In addition, they have representative characteristics of each category: PostgreSQL is a centralized disk-oriented DBMS and

[9] p is defined by the least squares correlation coefficient [18].

VoltDB is a distributed in-memory DBMS. The experiment procedure has four steps: (1) Submit the load condition, (2) Analyze the execution, (3) Collect the code coverage data, and (4) Proceed to the comparative study.

The experiments are executed on a HPC platform [19]. We used two different configurations: (1) 11 machines for PostgreSQL (one DBMS server and ten testers) and (2) 13 machines for VoltDB (three DBMS server and ten testers). Each machine has dual Xeon X5675@3.07 GHz with 48 GB of RAM running Debian GNU/Linux and connected by the Infiniband QDR (40 Gb/s) network. Our approach is implemented in Java 7. To collect the code coverage information of PostgreSQL, we used the GNU/Gcov, which is supported by default by the DBMS. For VoltDB, the code coverage is measured by Eclemma JaCoCo, since the DBMS is implemented in Java.

The threshold values are specified in Table 2. They were set based on the available hardware, the workload of our choice and the architecture of the DBMS. For instance, VoltDB does not need threshold values for the warm-up and thrashing states. Since VoltDB is an in-memory database, the warm-up process is basically instant. The thrashing state was not observed on the VoltDB. The t_s threshold is limited by 90 % of the transaction rate acceptance and the t_{st} is limited by 10 % of the transaction acceptance rate compared to the previous state "tps_{up}" (i.e., *Under Pressure*). For the t_{th}, we used one second. The slide window is set to 60 observations (i.e., $p = 60$).

Table 2. Threshold values for the state transitions. VoltDB does not need values for the warm-up and thrashing states since this DBMS does not experience these states.

	PostgreSQL	VoltDB
t_w	0.1	–
t_s	0.9	0.9
t_{st}	0.1 * tps_{up}	0.1* tps_{up}
t_{th}	1	–

The remainder of this section is guided by four research questions that are, respectively, related to: 4.1 performance results, 4.2 code coverage, 4.3 defects, and 4.4 thrashing prediction.

4.1 Does DSM Properly Reflect Performance Behaviors of a DUT?

This is the baseline question since our approach assumes that the DUT follows the DSM as designed. PostgreSQL experienced all the states of DSM as shown in Fig. 6. It presented an unexpected behavior concerning the ability to maintain a stable performance. During the execution of the workload case #1, the test driver increased the number of connections sequentially as described in Sect. 3.3. According to the specification of PostgreSQL, it can process 2,000 concurrent connections (i.e., defined by the MAX_CONNECTION configuration). However,

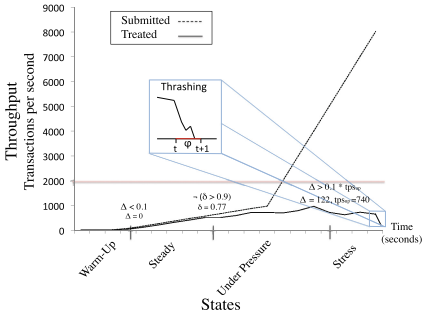

Fig. 6. Performance results of PostgreSQL.

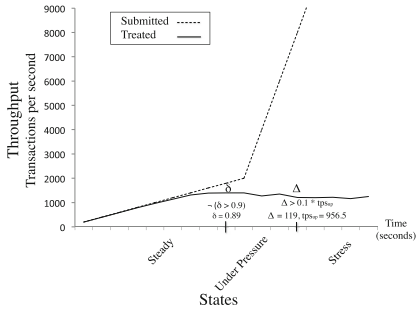

Fig. 7. Performance results of VoltDB.

the DUT could not deal with any workload greater than 1,000 concurrent connections as shown in Fig. 6[10]. For the workload case #2, the test driver increased the number of transactions with a fixed number of connections. PostgreSQL's behavior was more stable in this case and did not reach the thrashing state. However, it stayed either in the under pressure or stress states.

VoltDB presented consistent results in terms of throughput stability. Thus, the DUT was mostly either in the steady or under pressure states for both workload cases (see Fig. 7). However, the connection module was forced into stress state triggering a backpressure condition when applying the workload case #1. This condition occurs when a burst of incoming transactions was sustained for a certain period of time. This period must be sufficient to make the planner queue full. More information about this condition will be described in Sect. 4.3.

A curious reader may ask what would happen if instead of using MoDaST, we execute stress testing with a combination of standard benchmark on top of a popular testing tool, like jepsen or jmeter. We call this combination as baseline approach. By definition, this baseline approach considers performance constraints to ensure the DBMS on the steady state during measurement time. For example, one of the constraints defined by TPC-C as "Response Time" is: "At least 90 % of all transactions of each type must have a Transaction RT less than 5 s...". Thus, it is not possible to explore any stress condition of the DUT, since it never reached the under pressure nor the stress states. In both workload cases, the baseline approach only contemplates the steady state.

Answer*: MoDaST drove a DUT into each state of the DSM while the baseline technique can only explore two initial states.*

4.2 How Much Does Our Approach Cover the Source Code of a DUT (i.e., Code Coverage)?

This question is about our assumption that some execution paths in DBMS source code can only be explored when a certain amount of workload is requested.

[10] The thrashing state is only observable in the workload case #1.

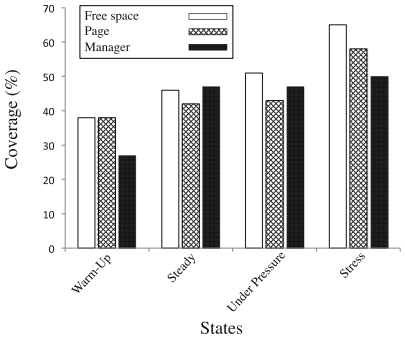

Fig. 8. Code coverage results of PostgreSQL. This focuses on three major modules: Free Space, Page, and Manager.

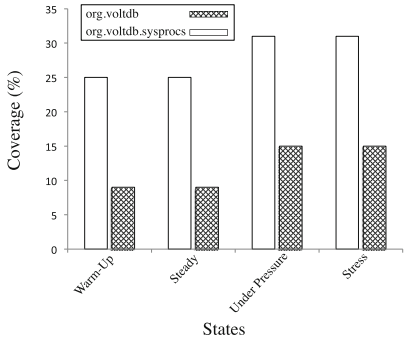

Fig. 9. Code coverage results of VoltDB. These packages are related to the concurrency control and server management

Figure 8 shows the code coverage results of PostgreSQL. Three packages presented a significant impact on the following modules: (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. The coverage increased mainly during the two last states: *stress* and *thrashing*. It occurs because those packages are responsible for managing disk page allocation and the transaction lock mechanism. PostgreSQL needed to execute functions dedicated to stress conditions to allocate extra resources and to deal with the high concurrency when the number of transactions increases.

VoltDB showed a notable improvement of code coverage results as shown in Fig. 9, even though it was not significant compared to that of PostgreSQL. We observed the improvement in two packages: *org.voltdb* and *org.voltdb.sysproc*. These packages manage the maximum number of connections and concurrent transactions. The package "org.voltdb.sysproc" is related to the basic management information about the cluster. The above-mentioned VoltDB classes were not covered when applying the baseline approach. Basically, the warm-up and steady states did not generate any concurrent condition. We see the same result for PostgreSQL testing.

Answer*: MoDaST allows to explore a larger part of source code of DUTs than the baseline technique since a certain part of source code can be executed only when pushing the DUT to heavy loads.*

4.3 Does Our Approach Find Bugs?

This question is correlated to the previous one; if MoDaST can explore more lines of code by submitting different load conditions, we may find new defects located in functions dealing with stress conditions. During the experiments, we found two potential defects (one from PostgreSQL and one from VoltDB) and one new unreported major bug (VoltDB).

We identified a performance defect of PostgreSQL, which is related to the inability to deal with the incoming connections, mainly in the workload case #1. Actually, the defect can be triggered either by transaction or connection flooding. PostgreSQL implements a backend process to deal with the incoming clients. Each client keeps one connection with the database. Thus, for each incoming connection, the DUT starts a new backend process.

Moreover, each connection holds one or more transactions, which proceed to modifications in the database. The modifications are made by *insert* and *update* operations that compose each transaction. The DBMS configuration allows to set the number of concurrent connections (i.e., MAX_CONNECTIONS) up to the resources limitations. In our experiments, the maximum value set for MAX_CONNECTIONS was 2,000. Despite of the limit, PostgreSQL was not able to reach the number of 2,000 open connections at any time. As the number of connections/transactions increases, PostgreSQL spends most of the computational power dealing with the table locks instead of creating new backend processes. From the testing point of view, we consider a potential *defect*.

VoltDB experienced a backpressure condition by applying the workload case #2. The increasing number of submitted transactions, via JDBC interface, fulfills the planner queue limit (i.e., 250) and raised up the message below: **(GRACEFUL_FAILURE): 'Ad Hoc Planner task queue is full. Try again.'** This can be considered a potential defect[11], once the planner queue is waiting for VoltDB planner. The planner became full and started to reject new operations.

The code coverage also enabled to reveal a functional bug. Figure 10 shows the code fragment where the defect was identified (Line 426). We reported this bug[12] to the developer community of VoltDB. This branch of code is responsible for ensuring that the DUT does not accept more concurrent connections than the maximum constraint allowed by the server resources. The bug rose up when our approach led the DUT to the stress state, which exposed it to a race condition in the connection module. The solution for this bug is to ensure that, even during race conditions, the number of concurrent connections never goes beyond the limit. Basically it should be guaranteed in the condition statement (i.e., IF) by replacing "==" by ">=". VoltDB developers created a bug report[13] as a major bug and promptly fixed after our reporting.

Answer*: The MoDaST found and reproduced three potential defects and one of them is confirmed as a major bug by the developers of VoltDB.*

```
  ...
426  if (m_numConnections.get() == MAX_CONNECTIONS.get()) {
427    networkLog.warn(''Rejected connection from '' +
  ...
```

Fig. 10. Example of bug only identified under stress conditions.

[11] http://zip.net/bmps8J.
[12] http://zip.net/byptRy.
[13] https://issues.voltdb.com/browse/ENG-6881.

4.4 Can Our Approach Predict Performance Degradation (e.g., the Thrashing State)?

This question is necessary because performance prediction is one of the advantages when using MoDaST. If the prediction is available, the DBA can apply several solutions for preventing DBMS crashes. Our approach could predict the thrashing states of PostgreSQL (see Fig. 6). However, due the instability of PostgreSQL, it crashed immediately after detecting $\varphi < t_{th}$. Thus, it is almost impossible to take any action to avoid such a state. VoltDB never went to the thrashing state under the two workload cases. This implies that $\varphi \gg t_{th}$ and VoltDB was highly stable. It does not mean that our approach was not effective. Rather, MoDaST correctly performed thrashing prediction for a stable DBMS. Due to our limited resources, we could not significantly scale up the number transactions. This remains as future work.

Answer*: The thrashing prediction showed to be precise, even with: (1) Performance instability of PostgreSQL; (2) Resources limitations to crash VoltDB.*

5 Conclusion

In this paper, we presented a novel model-based approach to database stress testing, MoDaST. It leverages a state machine to figure out the internal state of DBMS at run time. We evaluated MoDaST on two popular DBMS: PostgreSQL and VoltDB. Our results show that MoDaST can successfully infer their current internal state based on the state model. We also found out that submitting a high workload can lead to exercising the kernel in many different ways that is not possible by the current testing tools. Consequently, we identified new bugs in both DBMS. In particular, one of the bugs is already confirmed as "major bug" and promptly fixed by the VoltDB hackers. Our future work includes applying MoDaST to NoSQL and Streaming DBMS, since they implement different levels of concurrency control for transaction processing and, therefore, require different testing assumptions.

Acknowledgments. Supported by the Digital Inclusion Project: Ministry of Communication of Brazil, National Research Fund of Luxembourg and CNPq grant 441944/2014-0.

References

1. Soror, A.A., Minhas, U.F., Aboulnaga, A., Salem, K., Kokosielis, P., Kamath, S.: Automatic virtual machine configuration for database workloads. ACM Trans. Database Syst. **35**(1), 7:1–7:47 (2008)
2. Chang, J.W., Whang, K.Y., Lee, Y.K., Yang, J.H., Oh, Y.C.: A formal approach to lock escalation. Inf. Syst. **30**(2), 151–166 (2005)
3. Jain, R.: The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley Professional Computing. Wiley, Hoboken (1991)

4. Storm, A.J., Garcia-Arellano, C., Lightstone, S.S., Diao, Y., Surendra, M.: Adaptive self-tuning memory in DB2. In: Proceedings of the 32nd International Conference on Very Large Data Bases. VLDB 2006, pp. 1081–1092. VLDB Endowment (2006)
5. Gray, J.: Why do computers stop and what can be done about it? (1985)
6. Willmor, D., Embury, S.M.: An intensional approach to the specification of test cases for database applications. In: Proceedings of the 28th International Conference on Software Engineering, pp. 102–111. ACM (2006)
7. Deng, Y., Frankl, P., Chays, D.: Testing database transactions with agenda. In: Proceedings of the 27th International Conference on Software Engineering. ICSE 2005, pp. 78–87. ACM, New York (2005)
8. de Almeida, E.C., Marynowski, J.E., Sunyé, G., Valduriez, P.: Peerunit: a framework for testing peer-to-peer systems. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE 2010, pp. 169–170. ACM, New York (2010)
9. DeWitt, D.J., Levine, C.: Not just correct, but correct and fast: a look at one of Jim Gray's contributions to database system performance. SIGMOD Rec. **37**(2), 45–49 (2008)
10. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing. SoCC 2010, pp. 143–154. ACM, New York (2010)
11. Zhu, J., Mauro, J., Pramanick, I.: R-cubed (r3): rate, robustness, and recovery - an availability benchmark framework. Technical report, CA, USA (2002)
12. Vieira, M., Madeira, H.: A dependability benchmark for OLTP application environments. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 742–753. VLDB Endowment (2003)
13. Fior, A.G., Meira, J.A., de Almeida, E.C., Coelho, R.G., Fabro, M.D.D., Traon, Y.L.: Under pressure benchmark for DDBMS availability. JIDM **4**(3), 266–278 (2013)
14. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall, Upper Saddle River (1991)
15. Bruno, N., Chaudhuri, S., Thomas, D.: Generating queries with cardinality constraints for DBMS testing. IEEE Trans. Knowl. Data Eng. **18**(12), 1721–1725 (2006)
16. Lo, E., Binnig, C., Kossmann, D., Tamer Özsu, M., Hon, W.K.: A framework for testing DBMS features. VLDB J. **19**(2), 203–230 (2010)
17. Binnig, C., Kossmann, D., Lo, E.: Towards automatic test database generation. IEEE Data Eng. Bull. **31**(1), 28–35 (2008)
18. Radhakrishna Rao, C., Helge Toutenburg, S., Heumann, C.: Linear models and generalizations, least squares and alternatives, 3rd edition. AStA Adv. Stat. Anal. **93**(1), 121–122 (2009)
19. Varrette, S., Bouvry, P., Cartiaux, H., Georgatos, F.: Management of an academic HPC cluster: The UL experience. In: Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS 2014), Bologna, Italy. IEEE, July 2014