DBStorm: A Cost-effective Approach for Generating Valid Workload to Test Transaction Processing

Anonymous Author(s)

ABSTRACT

The interdependence between components in a database system forms a huge software with complex execution logic inside codes. This complexity can result in subtle bugs that can not be exposed easily by manually constructed test cases. Moreover, nondeterministic characteristics of transaction processing exacerbate the difficulty to generate valid workloads to detect bugs. Until now, an automatic technique to generate **diverse valid workloads** for comprehensive testing of a transaction processing is still vacant. In this paper, we introduce **DBStorm**, which aims to generate diverse valid workloads for testing transaction processing in a cost-effective way. It is the first work that can resolve the contradiction between the indeterminacy of transaction processing and the deterministic requirement for valid workload generation in an efficient way. We absorb the idea of stochastic testing techniques to effectively explore the huge test space. Then we design a set of *deterministic data generation* mechanisms to capture the nondeterministic evolution of database states during transaction processing. Finally, we orchestrate these mechanisms with modest overhead by a lightweight in-memory structure *miniature shadow*. Illustrated by the theoretical analyses, *DBStorm* is effective and efficient in testing transaction processing. In our experiments, *DBStorm* demonstrates its excellent ability in accomplishing a valid workload generation with modest overhead. Compared with state-of-the-art *SQLancer*, *DBStorm* outperforms it in generation speed (7.1×), validity (1.2×) and coverage(1.5×). More importantly, *DBStorm* has helped to discover 17 bugs from the production-level database systems.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

transaction processing, bug detection, workload generation

ACM Reference Format:

Anonymous Author(s). 2022. DBStorm: A Cost-effective Approach for Generating Valid Workload to Test Transaction Processing. In *Proceedings of The 37th IEEE/ACM International Conference on Automated Software Engineering (Conference ASE 2022).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference ASE 2022, October 10–14, 2022, Michigan, United States

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00

https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Transaction processing by guaranteeing the ACID properties forms the "interface contract" between an application program and a database management system (*DBMS*). It mainly resolves the issues from 1) *concurrency*, i.e., incorrect effects that may come from concurrent or parallel program executions; 2) *failures*, i.e., incorrect effects that would come from interrupting program executions because of process or computer failures. Therefore, it has always been an important functionality for mission-critical applications.

Even though theoretical proofs have been done for almost all transaction processing mechanisms, the implementations may not be as strict as the definitions [48, 50]. It may be worse for distributed *DBMS*s that involve consensus protocols interacting with the atomic commit protocol [23, 27, 44], and then have extraordinarily complex interactions over multiple remote machines. Indeed, even for production-level *DBMS*s, they often meet transaction violations [2–4, 8, 10, 13, 25, 45]. Therefore, before deploying an application on a *DBMS*, comprehensive testing for the functionality of transaction processing is imperative, but it has been tough work to compose a valid workload in which operations do access data with a theoretical access distribution. The key challenges can be summarized as follows:

Test Coverage (C1) A program with a high test coverage has more of its source code executed during testing, having a lower chance of containing undetected software bugs compared to a program with a low test coverage [18]. A comprehensive testing is required to cover *statements*, *functions*, *branches*, *conditions* and so on [35]. But complex code logic behind *DBMS*s constructs an enormous test space, especially for transaction processing, which challenges the comprehensive enumeration of test cases. For example, any module with a succession of n branches in it can have up to 2^n paths within it. In PostgreSQL $v12.7$, only for branches, its size is more than $180K$. In such a situation, it is impossible for traditional benchmarks [5–7] to explore all code paths due to limited workloads. Even though there are automatic test case generation designs for query processing [9, 11, 12, 28, 39, 42, 51], it is still vacant for testing transaction processing.

Workload Validity (C2) A valid workload expects to run through all layers of a *DBMS*, which triggers more intensive contentions, builds more complex transaction dependencies and so on [17]. However, a relational model has strict integrity constraints, e.g., primary/foreign key constraints, so a valid workload should comply with these constraints, which is a tough thing considering the generation efficiency [19, 39, 42, 51]. The transaction processing in most production-level *DBMS*s are nondeterministic, i.e., the final results are unpredictable even for the same workload [31]. It is impossible to launch static workload generation and final result checking to verify transaction processing [29]. Since the evolution of database state continuously, dynamically and quickly, it challenges the validities of generated workload.

Testing Efficiency (C3) For application-oriented database developments, new architectures and new design techniques for *DBMSs* constantly emerge [23, 36, 44, 46]. This trend leads to the source code of *DBMSs* changing very fast. Therefore, there is an urgent demand to test a *DBMS* in a cost-effective and time-saving way, which raises a practical requirement on the portability of testing techniques to support recursive testings.

In this paper, we propose a valid workload generation approach, *DBStorm*, to address these challenges so that we can comprehensively test transaction processing in a cost-effective way. We first take a stochastic model to define the test space for database and workload, which provides a feasible way to explore test space (addressing **C1**). We design a set of *deterministic data generation* mechanisms to capture the nondeterministic evolution of database (addressing **C2**). By integrating *deterministic data generation* mechanisms, a lightweight structure *miniature shadow* is organized to populate databases and generate valid workloads, which facilitates a cost-effective way to test transaction processing (addressing **C3**).

Throughout the paper, we provide detailed theoretical analyses of the design of *DBStorm*, and quantitatively analyze the generation cost by both time and space complexities. Practically, we run extensive experiments to demonstrate the effectiveness and efficiency of *DBStorm*. The result shows that *DBStorm* outperforms state-of-the-artwork with a great predominance. We also run *DBStorm* on the popularly used production-level *DBMSs*. Surprisingly, we have successfully found 17 representative bugs and gotten positive feedback, which further gives strong evidence to the usefulness and necessity of this work.

2 PRELIMINARY

We first introduce the concepts of database and workload (Sec. 2.1). Then we illustrate the motivation for testing transaction processing (Sec. 2.2-2.3) and present the overall architecture of *DBStorm* (Sec. 2.4).

2.1 Database and Workload

A database consists of one or more tables portrayed by a schema. Each table contains several records, if any. For a database, its state is all of the records stored at a point in time. We denote pk as a primary key attribute, fk as a foreign key attribute, and $attr$ as a non-key attribute. Each record consists of a pk, several fks and $attr$s, if any. We denote D as the domain of an attribute, G (resp.$\tilde{G}$) as the theoretical (resp. empirical) data distribution of a fk or an $attr$. Because of the primary key constraint, data on pk is unique as the identifier of any record. Tab. 1 shows the notations used in our paper.

A workload W is a set of transactions made up of operations running on a database with N_r records. We denote F (resp.$\tilde{F}$) as the theoretical (resp. empirical) access distribution of W on N_r records. N_o is the number of operations in W among which N_v operations can access records. Suppose record i is accessed N_i times where $0 \le i \le N_r - 1$, then the total access times of all records $N_t = \sum_{i=1}^{N_r} N_i$.

There are four different semantics for an operation, i.e., *select*, *insert*, *delete* and *update*. *select* represents a filtering read on N_r records, which can be divided into *item-read* or *predicate-read*. *item-read* means *select* filtering records by one and only one pk; *predicate-read* means *select* filtering records by a predicate, which may be constructed from fk or $attr$ by logic operators, i.e., *AND* ($\wedge$), *OR* ($\vee$) or *NOT* ($\neg$). *insert* (resp. *delete*) represents a write to add (resp. remove) a record to (resp. from) the database; *update* is launched by a *delete* and then an *insert* for the same record. Since *insert*, *delete* or *update* changes the database state and only involves one record, we call them *item-write*.

Table 1: Notations

Notations	Description
S	a schema
pk	a primary key attribute
fk	a foreign key attribute
$attr$	a non-key attribute
D	the domain of an attribute
N_r	the number of records in a database
G,$\tilde{G}$	the data distribution of a fk or $attr$
W	a workload
F,$\tilde{F}$	the access distribution of a W on a database
N_o	the number of operations in W
N_v	the number of operations accessing records in W
N_i	the accessed times of record i ($0 \le i \le N_r - 1$)
N_t	the totally accessed times of N_r records
$\overline{T}$	a transaction template
T	an instantiated $\overline{T}$
ap	an access parameter in a operation
dp	a data parameter in a operation

```
Start Transaction
    SELECT Y.pk, Y.attr_0 FROM Y  JOIN Z ON Y.fk_0 = Z.pk
    WHERE Y.attr_0 < ap_0,0 AND Y.attr_0 >= ap_0,1;
    INSERT INTO Y VALUES(ap_1,0 , ip_1,0 , ip_1,1) ;
    UPDATE T SET Y.attr_0 = ip_2,0 WHERE Y.pk = ip_2,0;
    DELETE FROM Y WHERE Y.pk = ap_3,0 ;
Commit
```

Figure 1: Example Transaction Template: $ap_{i,j}$ or $dp_{i,j}$ is the j^{th} parameter in the i^{th} operation.

Transaction Template denoted as $\overline{T}$, is a transaction sketch where the parameters in operations are symbolized [30]. An example is shown in Fig. 1. The parameters in operations include *access parameters* (*ap*) to identify records, e.g., $ap_{2,0}$, and *data parameters* (*dp*) to indicate the new record added into database, e.g., $dp_{2,0}$. After instantiating $\overline{T}$, we have an instantiated transaction T.

2.2 Motivation Example

Let's illustrate five invalid workloads in Fig. 2, i.e., IW_{0-4}, based on table Y and Z (used throughout the whole paper). We analyze their invalidations in transaction processing considering the different layers of a *DBMS*. IW_0 has a syntax error, which is blocked out by *Parser* layer. IW_1 and IW_2 have the semantic error or with an invalid *where* clause, blocked out by *Validation* layer. Although IW_3 goes through *Transaction* layer, 67 violates the integrity constraint, which prevents it from further diving into the *Storage* layer. IW_4 touches the code of *Storage* layer, however, it does not access record, which is caused by the ignorance of the evolution of the database state. Since the goal of transaction processing is to guarantee the ACID property on a database, IW_4 is incapable of testing transaction

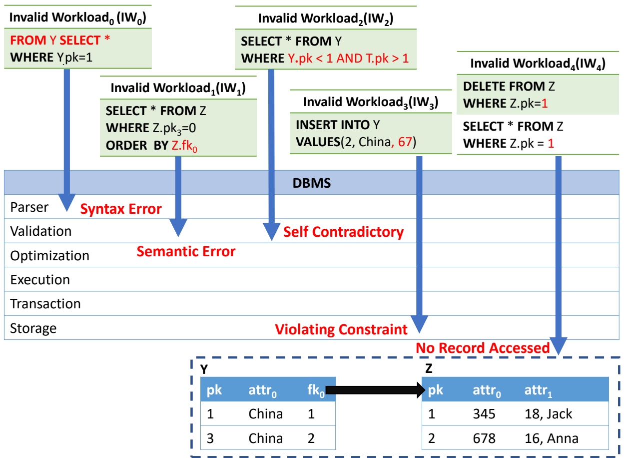


Figure 2: Example Invalid Workloads (IW) Distributed in Different Layers of a DBMS

processing. Notice that access distribution is an imperative means to test transaction processing with controllable contentions.

Suppose a workload W containing N_o operations runs on a database with N_r records complying with a theoretical access distribution F. After running W, N_v operations do access records with totally N_t times.

***α* validity** is the ratio of operations that **do access** records in database, i.e., **Accessibility** as $\alpha = \frac{N_v}{N_o}$;

***β* validity** is the goodness of fit between the empirical access distribution and the theoretical one, i.e., **Goodness** as

$\beta(N_t) = P\left(\chi^2_{N_r-1} \geq \chi^2\right)$ where $\chi^2 = \sum_{i=1}^{N_r} \frac{(N_i - N_t p_i)^2}{N_t p_i}$, $p_i = F(i) - F(i-1)$, and $\chi^2_{N_r-1}$ is distributed as a chi-squared variable with degree of freedom $N_r - 1$.

Both α and β decide the contention that should be handled by transaction processing, so they are critical for testing transaction processing. We then formalize *Valid Workload* in Def. 1 by α and β. For example, if α=50% and β=99% for workload W, it means only half of the operations in W can access the data successfully, and the access distribution has a 1% deviation from the theoretical one.

Definition 1. ***Valid Workload**: It is a workload that accesses database with a theoretical access distribution, which has* **accessibility** $\alpha = 1$ *and* **goodness** $\beta(N_t) \to 1$ *in probability as visit times* $N_t \to +\infty$.

2.3 Stochastic Testing Techniques

Stochastic testing technique has been proved to be an effective way to trigger bugs in large-scale systems, which can do as well as the enumerated tests [19, 33, 43]. Usually, it takes the profile-based method by extracting all features of the test space and encapsulating them into a *Stochastic Model*. Each feature can be represented by the pair $< seed, dist >$, with its candidate values in *seed* and the sampling distribution *dist* on *seed*. For example, for *type* of an attribute *attr*, $seed = \{integer, string, boolean\}$ and $dist$=$Uniform$, which means that when generating a database schema, each type in *seed* has an equal chance to define an attribute. When each feature in the *Stochastic Model* is instantiated, a complete test scenario is generated. *Stochastic Model* has the advantages in comprehensiveness and extensibility [43], i.e., features can be flexibly extended.

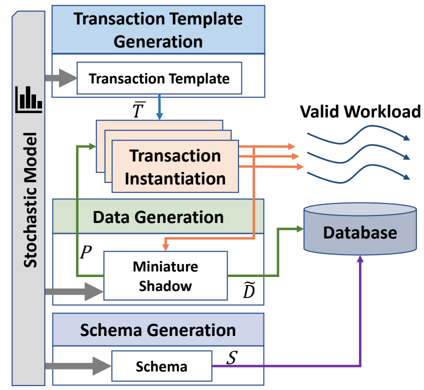


Figure 3: Architecture

2.4 DBStorm Architecture

As shown in Fig. 3, *DBStorm* consists of five components, i.e., *Stochastic Model*, *Schema Generation*, *Data Generation*, *Transaction Template Generation*, and *Transaction Instantiation*. *DBStorm* profiles all features of test space into *Stochastic Model*. Sampling from *Stochastic Model*, *Schema Generation* creates a schema S to portray a database and *Transaction Template Generation* composes transaction templates ($\overline{T}$). *Data Generation* integrates three *deterministic data generation* mechanisms into a structure *miniature shadow* that is used to populate database, i.e., $\tilde{D}$. On the other hand, by *miniature shadow*, *Transaction Instantiation* instantiates all parameters in $\overline{T}$, i.e., P. Then all instantiated $\overline{T}$s are loaded as a valid workload W on a database. During W running, the modifications to the database are consistently reflected into *miniature shadow*, which makes *miniature shadow* catches up with the evolution of database states to guarantee workload validity.

3 SCHEMA & TRANSACTION TEMPLATE GENERATION

A schema defines the collection of database objects to portray a database, mainly including tables, primary/foreign key attributes (pk/fk) and non-key attributes (*attr*), et. al. The relationships between database objects are a kind of one-to-many dependency, which can be represented by a *directed acyclic graph* (*DAG*) [21]. We organize all features about a schema into a *Stochastic Model* (*SM*), which is then taken to generate a *DAG* satisfying the constraints in a relational model.

A concrete *SM* example for the schema with respect to table Y and Z is shown in Fig. 4, and the involved objects make up a *DAG*. Each feature is a pair to declare the domain by *seed* and the sampling distribution by *dist*, formatted as <*seed*, *dist*>. For example, the number of tables is defined by $seed_{number} = \{1, 2, 3\}$ with $dist_{number}$=$uniform$, which means we can create a database with 1-3 tables in the same probability. For fk or $attr$, we sample the number of fk or $attr$ in each table by feature *number*; feature *size* decides the number of columns for a compound attribute; feature *type* gives the column type; for fk, its reference table is defined by feature *reference*. In table Z for example, it has one primary key pk typed int, and two *non-key* attributes, e.g., $attr_1$ is composed of two columns C_{12} and C_{13}. We instantiate each feature in *SM* by *topological sorting* on the *DAG*, so as to ensure the relationship among database objects to generate semantic correct workloads.

The time and storage complexities of generating a schema is $O(N_a)$ with N_a as the number of attribute in a schema. The database objects without dependencies can be generated in parallel, which greatly speeds up the generation process.

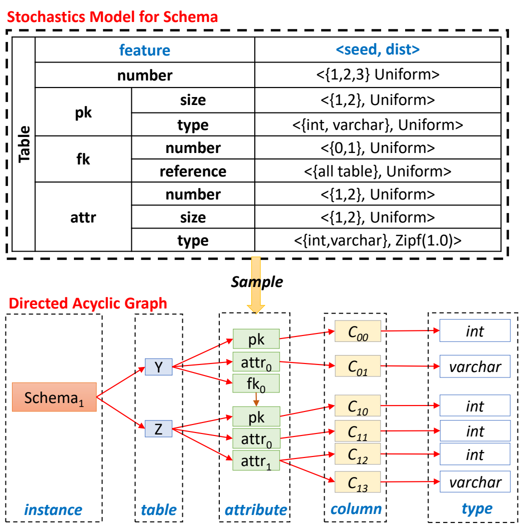


Figure 4: Example Schema for Table Y and Z

There are different *SMs* for different semantic operations which are represented by *abstract syntax tree* (*AST*). An example *select* is shown in Fig. 5. Each node on *AST* belongs to *reserved words*, *parameters*, *database objects*, or *predicates*, which is instantiated by *depth-first-search* to generate the parameterized operation satisfying *SQL* semantics.

To ensure semantic correctness, the database objects in a parameterized operation must comply with the definition of the database schema. For the example in Fig. 5, since the sub-tree rooted at *FROM* is Y joins with Z, the sub-tree rooted at *SELECT* must be chosen from the attributes of table Y or Z. Each predicate is a sub-tree rooted at an operator, e.g., *OR*, with the leaf being either a database object or a parameter. We take a recursive traversal to check whether there is any contradiction for a predicate by two steps:

1. Checking the compatibility on left and right children, e.g., to check whether $Y.attr_0 = ap_{0,0}$ and $Y.attr_0 >= ap_{0,1}$ are self-contradictory;
2. Checking the compatibility on root, e.g., to check whether $Y.attr_0 = ap_{0,0}\ OR\ Y.attr_0 >= ap_{0,1}$ is contradictory.

The execution control structures, i.e., if and *loop*, are sampled to organize these parameterized operations into a transaction template. The time and storage complexities of generating a transaction template is $O(N_o \cdot N_p)$ with N_o as the number of operations in a workload and N_p as the depth of the logic operator in a *predicate-read*. Since parameterized operations can be generated and organized into a transaction template in parallel, it does not bottleneck the generation performance.

4 DETERMINISTIC DATA GENERATION

In this section, we design three *deterministic data generation* mechanisms to resolve the challenges in data generation.

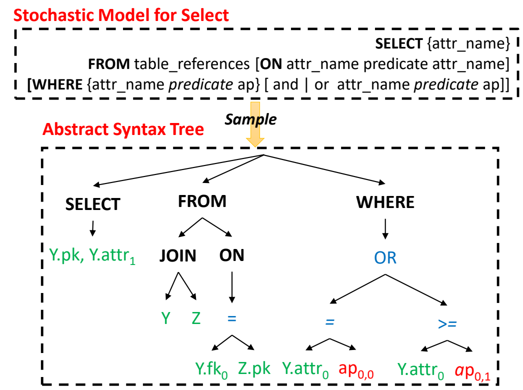


Figure 5: Example Parameterized Operation Generation

4.1 Challenges

DBStorm generates data to 1) launch database population and 2) generate valid workload. The main challenge of database population is **how to support a lightweight recursive testing**. The naive approach to recursive testing is moving database from one disk to another, which suffers from high latency, including multiple disk I/Os and network round trips. For valid workload generation, the main challenge is **how to capture database state under indeterminacy of transaction processing with modest overhead**, e.g., IW_4 in Fig. 2. A brute-force way to generate valid workloads is querying runtime database state to instantiate a transaction template while blocking concurrent transactions. However, querying runtime database state is not an time-saving thing and occupy lots of hardware resource. Furthermore, preventing from instantiating transaction templates concurrently limits throughputs of workload.

To address these challenges, we propose three *deterministic data generation* mechanisms to capture the nondeterministic evolution of database state in a cost-effective way. Specifically, Δ*-tolerant stable attribute* monitors database state on foreign keys and non-key attributes (Sec. 4.2) while *partition-based primary key* masters database state on primary key attributes (Sec. 4.3); *distribution constrained function* provides a calculation-based data generation to facilitate recursive testing (Sec. 4.4). Taking advantage of the design of *virtual column*, we incorporate them into a lightweight structure *miniature shadow* (Sec. 4.5) to populate database (Sec. 4.6) efficiently.

4.2 Δ-tolerant Stable Attribute

To capture database state in a cost-effective way, we propose Δ-*tolerant stable attribute* ($SAttr^\Delta$), as defined in Def. 2. $SAttr^\Delta$ is designed to portray database state by data distribution within Δ tolerance in probability. If $\Delta = 0$, it means we can perfectly capture database state even after a massive number of modifications. In Theorem 1, it is proved that the probability to be a well bounded $SAttr^\Delta$ can be realized by controlling the parameter values for *insert*, *delete* and *update* with rules in $R1 - R3$. Since $P\left(\left|\tilde{G}_{N_o} - \tilde{G}_0\right| \le \Delta\right) \to 1$ as $N_r \to +\infty$, we can decrease Δ by increasing the number of records N_r, and then we can capture database state in probability.

Definition 2. Δ***-tolerant Stable Attribute***($SAttr^\Delta$)*: Given an attribute attr in a database, there is an empirical data distribution* $\tilde{G}_{N_o}$ *after* N_o *modifications, i.e.,* insert, delete *or* update. *We define* *attr as $SAttr^{\Delta}$ iff. there is a deviation between $\tilde{G}_{N_o}$ and the theoretical data distribution G within Δ tolerance, i.e., $\left|\tilde{G}_{N_o} - G\right| \leq \Delta$ where $0 \leq \Delta \leq 1$.*

$SAttr^{\Delta}$ provides a lightweight mechanism to monitor database state with the storage cost $O(N_a)$ for the theoretical data distributions where N_a is the number of attributes in database. Moreover, it avoids intensive contentions among clients because there is no need for concurrency control on unchanged information, i.e., the theoretical data distribution.

Theorem 1. *Given an attribute $attr$ in database, there are N_r initial records sampled from a theoretical data distribution G on its domain D_{attr} with $\partial G(x)/\partial x > 0$ for $x \in D_{attr}$. There is an empirical data distribution $\tilde{G}_{N_o}$ after N_o modifications which follow:*

- *R1) the data parameter of insert is sampled from G;*
- *R2) when the number of records is less then N_r, delete is prohibited, or else, the parameter in delete is randomly instantiated;*
- *R3) for update, it is launched by a delete and then an insert on the same record.*

Then the probability that $attr$ is a $SAttr^{\Delta}$ is bounded by

$$P\left(\left|\tilde{G}_{N_o} - G\right| \leq \Delta\right) > 1 - 2 \cdot e^{-2 \cdot N_r \Delta^2}. \quad (1)$$

Proof. Suppose there are N'_r records in the database after N_o modifications. There are three different semantic modifications, i.e., *insert*, *delete* or *update*. *update* is launched by a *delete* and then an *insert* on the same record. *insert* adds its data parameter into the database. *R1* requires that the data parameter of *insert* is sampled from G. Notice that the initial records are also sampled from G. Therefore, according to *DKW's Inequalities* [37], we have

$$P\left(\left|\tilde{G}_{N_o} - G\right| \leq \Delta\right) > 1 - 2 \cdot e^{-2 \cdot N'_r \Delta^2}. \quad (2)$$

R2 requires $N_r \leq N'_r$ while *delete* removes a record from $attr$, so we have

$$1 - 2 \cdot e^{-2 \cdot N'_r \Delta^2} \geq 1 - 2 \cdot e^{-2 \cdot N_r \Delta^2}. \quad (3)$$

Thus, the result is desired.

□

4.3 Partition-based Primary Key

Because the primary key attribute (*pk*) complies with a unique constraint, its state cannot be captured by $SAttr^{\Delta}$. We propose to launch *partition-based primary key* (*PPK*) management to capture database state on *pk* by controling modifications to only a part of keys, called dynamic keys, specified by a partitioning strategy P deterministically. Each dynamic key couples with a pair $<L, \lambda>$ where L ($L \in \{free, read, write\}$) is a lock for dealing with contentions, and λ ($\lambda \in \{0, 1\}$) represents its existence state in the database. L and λ are initialized to $free$ and 0 for *lock-free* and *non-existence*.

Since static keys deterministically reside in the database, there is no need to sketch other more information about their states. Therefore, the storage cost for *PPK* is $O(N_d)$ where N_d is the number of dynamic keys. By adjusting the ratio between static and dynamic keys, we can control the overhead on memory consumption.

4.4 Distribution Constrained Function

We define to take *distribution constrained function* (*DCF*), as defined in Def. 3 to support a lightweight recursive testing. The main idea is that *DCF* deterministically calculates dependent variable following a theoretical data distribution. Taking advantage of *DCF*, we hold the distribution on the initial database satisfying the requirements of $SAttr^{\Delta}$, while deterministically populating the database for recursive testing with only 1 disk I/O.

Definition 3. ***Distribution Constrained Function****: It is a function DCF(X) whose dependent variable Y follows a theoretical data distribution G, i.e., $Y = DCF(X)$ and $Y \sim G$.*

Lemma 1. *Suppose G is a theoretical data distribution and its inverse function G^{-1} exists. Let U be a uniform random variable, i.e., $U \sim Uniform(0, 1)$. Then $G^{-1}(U)$ follows G, i.e., $G^{-1}(U) \sim G$ [20].*

In Lemma. 1, given a uniform random variable $U \sim Uniform(0, 1)$, the inverse function $G^{-1}(U)$ follows G where G is a theoretical data distribution. Therefore, $G^{-1}(U)$ provides a feasible way to construct *DCF*. The storage complexity of *DCF* is $O(N_a)$ where N_a is the number of attributes in database. In such a way, we can then calculate the initial database quickly by a specified theoretical distribution G without data migration.

4.5 Integrating Deterministic Data Generation

However, even taking these *deterministic data generation* mechanisms, we still face the following tough issues. (**I1**) To guarantee $SAttr^{\Delta}$, the domain (all values) of an attribute needs to be stored physically, which may cause tremendous memory consumption. (**I2**) If the type of an attribute is other than the numeric, e.g., $varchar$, it is impossible to define a *DCF*, i.e., calculating G^{-1} where G is a theoretical data distribution. To alleviate these issues, we virtually couple an integer typed column called *virtual column* (*VC*) with each attribute; *VC*'s domain is logically defined as $D_{VC} = [0, |D_{attr}|)$ where D_{attr} is the domain of an attribute; then we build a bijection (Φ) between *VC* and $attr$. For example, given a $varchar$ typed attribute, its domain (D_{attr}) contains 10^5 strings with length 10^3 characters, which costs $10^2 MB$; however, taking advantage of *VC*, we logically define $D_{VC} = [0, 10^5)$ and a bijection Φ from D_{VC} to D_{attr}. In such a way, we only bookkeep D_{VC} and Φ instead of all the values physically, which reduces the storage complexity from $O(|D_{attr}|)$ to $O(1)$ (addressing **I1**). Since *VC* is an integer typed column, it is also practical to implement *DCF* (addressing **I2**).

By *virtual column*, we design a structure *miniature shadow* (*MS*) to capture runtime database state with less overhead. *MS* integrates the information of implementing all *deterministic data generation* mechanisms in Alg. 1 and an example is shown in Fig. 6. To implementing $SAttr^{\Delta}$, *MS* bookkeeps *virtual column*, i.e., domain D_{VC} and bijection function Φ, and the theoretical data distribution G where G^{-1} can be calculated to implement *DCF*. Alg. 1 samples VC_{attr} from D_{VC} by G, (*line 2*), and then maps values for $attr$ by Φ (*line 3*). For *DCF*, it takes G^{-1} to generate VC_{attr}, which has the normalized VC_{pk} as the dependent variable of G^{-1} (*line 7*), and then maps values of $attr$ by Φ (*line 8*). Since primary keys are managed by *PPK*, *MS* stores the partition strategy P, all dynamic keys, and its *virtual column* domain. For *PPK*, Alg. 1 partitions VC_{pk} into static and dynamic keys and map value by Φ (*line 5*).

pk	D_{VC}	[0,2)
	P	$VC_{pk}\%2$
	dynamic keys	$\langle L, \lambda \rangle = \langle free, 0 \rangle$ *for* $VC_{pk} = 1$
	Φ	$pk = VC_{pk}$
attr$_0$	$\langle G, D_{VC} \rangle$	$\langle Uniform, [0,1) \rangle$
	Φ	$attr_0 = \begin{cases} China, & VC_{attr_0} = 0 \\ Japan, & VC_{attr_0} = 1 \end{cases}$
fk$_0$	$\langle G, D_{VC} \rangle$	$\langle Zipf(1.0), [0, Z.pk.N_r) \rangle$
	Φ	$Z.pk.\Phi$

Figure 6: Example Miniature Shadow for Table Y

An example *MS* is shown in Fig. 6. Table Y has three attributes, i.e., pk, $attr_0$ and fk_0. Based on *virtual column*, we define the bijection Φ for each attribute. Notice that fk_0 has its Φ directly from the referenced *pk*. For $attr_0$ and fk_0, we follow the requirement of $SAttr^{\Delta}$ by encapsulating a theoretical data distribution G and D_{VC} into *MS*. For *pk*, we implement *PPK*, i.e., embedding the partition strategy P and the states of dynamic keys in *MS*.

Algorithm 1 Deterministic Data Generation

Input Miniature Shadow *MS*;

1: **procedure** $SAttr^{\Delta}()$
2: $\quad VC_{attr} \leftarrow attr.sample(MS.attr. < G, D_{VC} >)$;
3: $\quad$ **return** $MS.attr.\Phi(VC_{attr})$;
4: **procedure** $PPK(VC_{pk})$
5: $\quad$ **return** $MS.pk.P(VC_{pk}), MS.pk.\Phi(VC_{pk})$;
6: **procedure** $DCF(VC_{pk})$
7: $\quad VC_{attr} \leftarrow MS.attr.G^{-1}(\frac{VC_{pk}}{MS.pk.N_r})$;
8: $\quad$ **return** $MS.attr.\Phi(VC_{attr})$;

In Alg. 1, *inversion* (*line 7*) provides a way to generate a value from the domain of primary key to a non-primary-key *attr* following a distribution G; *sampling* (*line 2*) produces a value from G; *partitioning* (*line 5*) divides all primary keys into dynamic keys and static ones. The storage complexity of *MS* that incorporates three *deterministic data generation* mechanisms is $O(N_a + N_d)$, where N_a is the number of attributes in a database and N_d is the number of dynamic keys.

Table 2: Example Database Population for Table Y

$\boxed{VC_{pk}}$	pk	$\boxed{VC_{attr_0}}$	$attr_0$	$\boxed{VC_{fk_0}}$	fk_0
$\boxed{0}$	0	$\boxed{0}$	*China*	$\boxed{1}$	1

4.6 Database Population

Based on Alg. 1, we provide an approach to populate database. For *pk* generation, we take *PPK* to partition all primary keys, assign dynamic keys into *MS*, and all VC_{pk} are mapped into values of *pk* by Φ. For *attr* and fk generation, we take *DCF* to deterministically calculate the initial data, i.e., VC_{attr}, and then we map VC_{attr} by Φ into values. Tab. 2 shows an example of database population by *MS* in Fig. 6. Table Y only contains one records, i.e., $VC_{pk} = 0$, which is a static key; the corresponding VC_{attr_0} is calculated by G^{-1}, i.e., 0, and then the corresponding value for $attr_0$ is mapped by Φ, i.e., *China*. We take the same method for $attr_0$ to fill values to fk_0.

The time complexity of database population is $O(N_r/N_t)$ where N_r is the numbers of records and N_t is the number of generating threads. Since all columns can be generated in parallel based on *DCF*, it is vertically scalable to populate database. The records can be loaded into database as batches, which can also be generated in parallel for horizontal scalability. Both ways speed up recursive testings.

5 VALID WORKLOAD GENERATION

As described in Sec. 2, we define valid workload having accessibility $\alpha = 1$ and goodness in distribution $\beta(N_t) \to 1$ in probability as $N_t \to +\infty$. For generating a valid workload, we take *miniature shadow* that incorporates the above three *deterministic data generation* mechanisms to capture runtime database state. In this section, we first introduce the instantiation of parameterized operations in a transaction template $\overline{T}$ (Sec. 5.1), and then we describe the instantiation of $\overline{T}$ during workload running (Sec. 5.2).

5.1 Operation Instantiation

We classify the parameters in $\overline{T}$ into *access parameter* (*ap*) and *data parameter* (*dp*), where *ap* identifies records accessed by an operation, and *dp* is injected into the database.

Instantiating *dps*. Since foreign key (fk) and non-key attributes (*attr*) should comply with $SAttr^{\Delta}$, we take $SAttr^{\Delta}(MS.attr)$ in Alg. 1 to instantiate *dp*, which does not break the theoretical data distribution as proved in Theorem 1.

Instantiating *aps* of item-read/write. Since primary key (*pk*) takes the role of record identification, it determines the accessed records involved by operations. In Alg. 1, $PPK(VC_{pk})$ is designed to capture runtime database state on *pk*, so we can ensure each *item-read/write* accesses some records after instantiating *aps*, i.e., $\alpha = 1$. As illustrated in Theorem. 2, if we make *aps* of *item-read/write* sampled from the theoretical access distribution F, then we can achieve the deviation between the empirical access distribution and the theoretical one as little as possible. Note that both the existence of *pk* and the type of operation determine the sample domain with respect to F. Concretely, for *update* or *select*, we take the static keys and dynamic keys having $\lambda = 1$ as the sampling domain; for *insert* (resp. *delete*), we take the dynamic keys having $\lambda = 0$ (resp. $\lambda = 1$) as the sampling domain.

Theorem 2. *Let workload W only contain* item-read/write *and the ratio of valid operations $\alpha = 1$. If the accessed records are sampled from the theoretical access distribution F, then there is little deviation between the empirical access distribution and the theoretical one, i.e., $\beta(N_t) \to 1$ in probability as $N_t \to +\infty$, where N_t is the total access times of all records.*

Proof. W only contains *item-read/write* operations, and then $\alpha = 1$ means each operation accessing one and only one record. Further, because the accessed records are sampled from F, we have $N_t^{-1}(N_1, N_2, \cdots, N_{N_r})^T \to (p_1, p_2, \cdots, p_{N_r})^T$ in probability according to the *Law of Large Number*. $\beta(N_t)$ is a continuous function of the differences between the empirical data distribution and the theoretical one. Applying *Continuous Mapping Theorem* from Chapter 2.2 of [41], we have $\beta(N_t) \to 1$ in probability as $N_t \to +\infty$. □

Instantiating *aps* of predicate-read. Randomly assigning values for a predicate may lead to invalid workloads. Suppose we have two *preds*, i.e., $pred_0 = Z.attr_0\ BETWEEN\ ap_0\ AND\ ap_1$ and $pred_1 = Y.pk < ap_2\ AND\ Y.pk >= ap_3$. If we instantiate them as

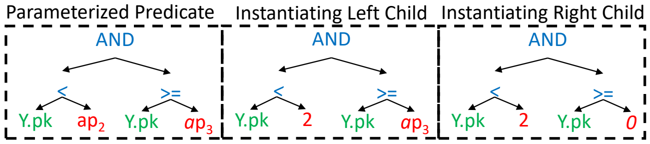


Figure 7: Example Instantiating Predicate-read

$ap_0 = 6$, $ap_1 = 3$, $ap_2 = ap_3 = 1$, both $pred_0$ and $pred_1$ are unsatisfiable, i.e., no record accessed. For generating a valid workload, we should not only conform to the semantics of an operator itself, e.g., $ap_0 \leq ap_1$ in $pred_0$, but also make predicate satisfiable for a given database state, e.g., $ap_2 = 2$ and $ap_3 = 0$ in $pred_1$ is satisfiable for database state of Tab. 2.

Algorithm 2 Instantiating Predicate-read

```
Input    Miniature Shadow MS;
 1: procedure INST_PRED(pred)
 2:     push_down_NOT(pred);
 3:     if pred.root == AND || pred.root == OR then
 4:         do
 5:             pred.left = Inst_Pred(MS, pred.left);
 6:             pred.right = Inst_Pred(MS, pred.right);
 7:         while pred == ∅
 8:     else
 9:         do
10:             pred ← sample(MS);
11:         while pred ∩ D_pred.attr == ∅
12:     return pred;
```

Alg. 2 instantiates a *predicate-read* where the predicate *pred* is satisfiable for a given database state indicated by *miniature shadow MS*. It first pushes down logic operator *NOT* to a leaf node, if any (*line 2*). If the root of *pred* is a logic operator *AND* or *OR* (*line 3*), it recursively instantiates left and right children rooted at *pred* until there are results from the instantiated *pred* (*line 4-7*); otherwise, it samples data from *MS* to instantiate the parameters, until *pred* has an intersection with the domain of *attr* involved in *pred*, i.e., $D_{pred.attr}$ (*line 9-11*). Fig. 7 demonstrates an example based on *MS* in Fig. 6. It instantiates the left and right children, i.e., $Y.pk < ap_2$ and $Y.pk >= ap_3$ respectively. Both ap_2 and ap_3 are sampled from a *MS* shown in Fig. 6. Let the sampling results are $ap_2 = 2$ and $ap_3 = 0$. Both $Y.pk < 2$ and $Y.pk >= 0$ intersect with the domain of *pk*, i.e., $D_{pk} = [0, 2)$. Thus $Y.pk < 2$ *AND* $Y.pk >= 0$ is satisfiable for current database state shown in Tab. 2.

For *predicate-read* containing no other operators, Theorem. 3 illustrates that the expectation of $1 - \alpha$ converges to 0 as fast as $N_r^{-1/2}$. If setting N_r a big number, we can guarantee accessibility α is almost 1. However, the expectation of that $1 - \alpha$ converges to 0 may be a little slow for *predicate-read* with multiple logic *AND* operators, as discussed in Remark. 1.

The time complexity of Alg. 2 is $O(N_p)$ where N_p is the depth of a predicate. Notice that Alg. 2 may iterate multiple times in *line 7*, if the set represented by *pred* is $\varnothing$. However, the number of iteration can be decreased by sampling the parameter that has a high probability in the database.

Theorem 3. *Let workload W only contain predicate-read without logic operator, i.e, AND ($\wedge$), OR ($\vee$) or NOT ($\neg$). If we instantiate predicates in operations with Alg. 2, the expectation of $1 - \alpha$ converges to 0 as fast as $N_r^{-1/2}$ with N_r as the number of records in database, i.e.,*

$$E(1 - \alpha) \leq O(N_r^{-1/2}) \tag{4}$$

Proof. Let G be the theoretical data distribution on attribute $attr$ which belongs to $SAttr^{\Delta}$. Since $\partial G(x)/\partial x > 0$ where $x \in D_{attr}$ as declared in Theorem. 1, then given a $\tilde{G}$ sampling from G, we have

$$E(1 - \alpha|\tilde{G}) \leq \int \left|\tilde{G} - G\right| dx \tag{5}$$

According to Chapter 7.2 of [41], we have

$$E(\int \left|\tilde{G} - G\right| dx) \leq O(N_r^{-1/2}) \tag{6}$$

Thus, the result is desired. □

Remark 1. *Suppose $pred^o$ is a predicate-read without logic operators. There are four types of predicate-read composing of $pred^o$s by logic operators, which are:*

(1) $pred^{\vee} = \bigvee_{i=0}^{N_p-1} pred_i^o$ where $N_p \geq 1$. $pred_i^o$ is a predicate-read of type $pred^o$, as illustrated in Theorem. 3, each $pred_i^o$ follows Inequality. 4. As the semantics of disjunctive normal form $\vee$, $pred^{\vee}$ follows Inequality. 4.

(2) $pred^{\wedge} = \bigwedge_{i=0}^{N_p-1} pred_i^o$ where $N_p \geq 1$. Let $attr_i$ involve in $pred_i^o$ and $E(1-\alpha_i)$ represent the expectation of invalid ratio on $attr_i$ which follows Inequality. 4. As the semantics of conjunctive normal form $\wedge$, we have

$$E(1 - \alpha) \leq \sum_{i=0}^{N_p-1} E(1 - \alpha_i) \leq O(N_p \cdot N_r^{-1/2})$$

(3) $pred^{\vee\wedge} = OP_{i=0}^{N_p-1}\ pred_i^o$ where $1 \leq N_p$ and $OP \in \{\bigvee, \bigwedge\}$. The composite predicate $pred^{\vee\wedge}$ is decomposed into the ones typed $pred^{\vee}$ or $pred^{\wedge}$.

(4) $pred^{\neg} = \neg pred^{\vee\wedge}$. In line 4-5 of Alg. 2, $\neg$ is pushed down to the involved leaf node, if any. In such a way, $pred^{\neg}$ is then decomposed into $pred^{\vee\wedge}$.

5.2 Transaction Instantiation

During workload running, most-production-level *DBMS*s change the database state nondeterministically [31]. Taking advantage of *miniature shadow*, we can capture the evolution of database state in a modest overhead. Then we propose Alg. 3 to instantiate parameters in $\overline{T}$ during workload running. We divide the instantiation into two phases 1) instantiating $\overline{T}$ by *MS*, i.e., *Instantiate($\overline{T}$)* (*line 1-9*), and 2) applying modifications to *MS*, i.e., *Apply(T)* (*line 10-16*). Alg. 3 maintains a *Write Set* (*WS*), i.e., modification to database, and a Lock Table (*LT*), i.e., the acquired locks on dynamic keys in *MS*. It instantiates each parameterized operation in $\overline{T}$ as described in Sec. 5.1, while it acquires the involved read locks into *LT* (*line 4*). The locks are released after execution (*line 9*), which prevent others from modifying the dynamic keys concurrently. If an operation *op* is executed successfully, the modifications to database, if any, are logged into *WS* (*line 6*); otherwise, Alg. 3 rollbacks $\overline{T}$ (*line 8*).

Notice that invalid operations may appear in some special cases. For example, there may be two identical insert operations in a transaction, which are coincidentally assigned the same parameters. The second insertion is expected to fail for the constraint on the

Algorithm 3 Transaction Template Instantiation

```
Input    Miniature Shadow MS;     Write Set WS = ∅;
         Lock Table LT = ∅;        Read Lock Timeout t;
 1: procedure INSTANTIATE(T̄)
 2:     for each operation op in T̄ do
 3:         MS.temp_apply(WS);
 4:         LT ← MS.instantiate_acquire(op, LT, t)
 5:         if success == DBMS.execute(op) then
 6:             WS.log(op);
 7:         else
 8:             DBMS.rollback();
 9:         LT.release();
10: procedure APPLY(T)
11:     LT ← MS.acquire(WS, LT)
12:     if success == T.commit() then
13:         MS.apply(WS);
14:     else
15:         DBMS.rollback();
16:     LT.release();
```

primary key. To deal with this problem, Alg. 3 logs the uncommitted record in WS (*line 6*) and temporarily overwrites the corresponding dynamic keys in *MS* until instantiation finishes (*line 3*). In such a way, the transaction to be instantiated, i.e., $\overline{T}$, is sensitive to the uncommitted record written by itself.

If we have modifications to database, i.e., $WS \neq \varnothing$, we apply WS to *MS* to capture database changes, i.e., *Apply(T)* (*line 10-16*). It acquires the involved write locks into LT (*line 11*) before committing the instantiated transaction T (*line 12*). If T committed, it applies WS to *MS* (*line 13*). Finally, it releases the write locks in LT (*line 16*).

Alg. 3 may acquire multiple read/write locks on *MS* at a time, which may cause deadlocks. Taking advantages of WS, it acquires write locks in order, which can prevent deadlocks between writes. Alg. 3 breaks deadlocks between reads and writes by setting an timeout t for acquiring read locks (*line 4*).

If the number of acquiring read locks exceeds an given threshold, the corresponding operation fails to instantiate parameters, which leads to an invalid operation. Suppose the deadlocks disappear in Alg. 3, Theorem. 4 illustrates that each *item-read/write* in workload can access some records, i.e., $\alpha = 1$. Therefore, such a predicate instantiation algorithm is effective to generate valid workload for *item-read/write* executed concurrently.

The time complexity of Alg. 3 is $O(N_o \cdot N_p)$ where N_o is the number of operation in a workload and N_p is the depth of a predicate. Notice that the contention on dynamic keys may exacerbate the time to acquire locks, however, which can be mitigated by increasing the number of dynamic keys.

Theorem 4. *Let workload W only contain* item-read/write. *If Alg. 3 without occurring deadlock instantiates the parameters of transaction template during workload running, then each* item-read/write *can access some records, i.e., $\alpha = 1$.*

Proof. Let $attr$ be the attribute involved in access parameters of *item-read/write*, which has the following two types:

(1) *static key*. Since the static keys stay in a database and are unchanged during workload running, there is a static key to be accessed by the operation whose access parameter is a static key, i.e., $\alpha = 1$.

(2) *dynamic key*. According to Alg. 3, each dynamic key is locked while being read or written, so only one thread can read or write at any time. Therefore, any dynamic key is consistent with the changes of database. We take a dynamic key with $\lambda = 1$ to instantiate *read*, *update* or *delete*, and take a dynamic key with $\lambda = 0$ to instantiate *insert*; furthermore, because there is no deadlock during instantiation, the read locks can be acquired by each operation. Therefore, a dynamic key can be accessed by each operation whose access parameter is a dynamic key, i.e., $\alpha = 1$.

□

---

6 ANALYSIS AND DISCUSSIONS

In this section, we discuss the validity of the generated workload as the definition. Then we analyze the cost of database population and valid workload generation. Finally, we introduce the usability of *DBStorm*.

α Validity For workload only having *item-read/write*, *DBStorm* has $\alpha = 1$. For workload containing *predicate-read*, *DBStorm* achieves that the expectation of $1 - \alpha$ converges to 0 and the convergence rate depends on the depth of the logic operator in a *predicate-read*. However, in real transactional workload, the depth of the logic operators is less than 3 [5–7]. In such a way, even in the worst case, the convergence rate is faster than $3N_r^{-1/2}$ where N_r is the number of records. Therefore, by increasing N_r, *DBStorm* can achieve $\alpha = 1$ with few errors even though the generated workload contains *predicate-read*, which demonstrates much better testing ability compared to related work as shown in Sec. 7.

β Validity For workload only having *item-read/write*, *DBStorm* has $\beta(N_t) \to 1$ in probability as $N_t \to +\infty$ illustrated in Theorem 2. However, for workload having *predicate-read*, since the complexity and diversity in the semantics of a predicate, *DBStorm* can not achieve full β validity, i.e., $\beta = 1$. To deal with this issue, we should control each record accessed by *predicate-read*, which is still very difficult under the nondeterministic evolution of the database state.

Cost Let the numbers of attributes and operations are N_a and N_o; the depth of logic operators is N_p; the numbers of records and dynamic keys are N_r and N_d; the number of generating threads is N_t. For both database population and workload generation, the storage complexity is $O(N_a + N_d)$, i.e., the storage overhead of *miniature shadow*. The time complexity of database population is $O(N_r/N_t)$ while workload generation is $O(N_o \cdot N_p)$. These complexities are linear scalable with the corresponding variable. Specifically, N_a depends on the schema, which is usually small for testing transaction processing, e.g., 92 for *TPC-C* [6]. N_o depends on the transactions in a workload, e.g., even in the worst case, the number of operations in *new-order* of *TPC-C* is less than 66. N_p is usually short, e.g., less than 3 in *TPC-C*. To test transaction processing, the complexity of concurrency and contention is more important to trigger bugs instead of the number of records in the database, which is different from testing analytical database system [32]. The number of dynamic keys N_d is controllable by *DBStorm*. Although a large N_d meets fewer contentions in workload generation, i.e., is good for improving validity, our experiment shows even a small N_d has a good guarantee of workload validity. Therefore, *DBStorm* provides a cost-effective valid workload generation approach for testing transaction processing, well verified by experiments in Sec. 7.

Usability Combing with *Stochastic Model*, *DBStorm* can generate diverse valid workloads to automatically explore the complex code logic in transaction processing, which contributes greatly to having luxuriant test cases during the developing a database system. Additionally, *DBStorm* can easily combine with other testing techniques to reinforce testings, i.e., *chaos engineering* [15, 16]. *DBStorm*has a good adaptability by defining different configurations for *Stochastic Model* to integrate with verification tools of transaction processing, e.g., *Jepsen* [10] and *Cobra* [45] et.al, to accomplish bug detection.

7 EXPERIMENTS

In this section, we launch sufficient experiments to answer the following questions:

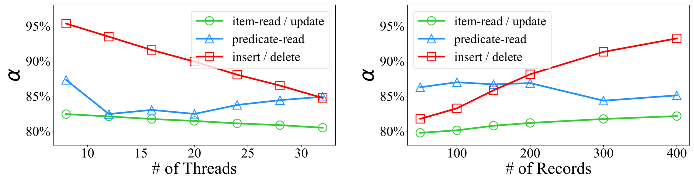
- How validity is *DBStorm* in generating workloads with respect to the definition of valid workload? (Sec. 7.1)
- What are the benefits of valid workload incorporating stochastic testing techniques? (Sec. 7.2)
- Is cost-effective *DBStorm* in populating database and generating valid workload? (Sec. 7.3)

Setup. *DBStorm* is implemented by Java (v.1.8). We conduct experiments in two CentOS 7.9 systems on four 16-core machines, connected using 10 Gigabit Ethernet. Each machine is equipped with 2 Intel Xeon Silver 4110 @ 2.1 GHz CPUs, 160 GB memory, 4 TB HDD disk configured in RAID-5, and 4 GB RAID cache. Since *DBStorm* is straightforward to run against production-level *DBMS*s, we select one of the most popular open-source ones, i.e., PostgreSQL (v12.7), to carry out the experiments. To demonstrate the power of *DBStorm*, we integrate it with verification tool *Jepsen* [10] and run it on several production-level *DBMS*s, even the well-developed commercial one. Its competence is further proved by the bugs exposed [14], which have not been detected by other approaches.

Workload. *TPC-C*, *SmallBank* and *YCSB* are popularly used to benchmark database performance. We take these three benchmarks to demonstrate their code coverage ability compared to *DBStorm*, which are all implemented by *OLTP-Bench* [24].

To further expose the technical designs of *DBStorm*, we extend *YCSB* [22] to a *YCSB* variant, i.e., *YCSB-SQL*, which contains several transactions based on two tables, i.e., table Y and Z in Fig. 2. Concretely, table Y has $2K$ keys, and table Z has 20 keys referred by table Y. *YCSB-SQL* covers three types of transactions with relational semantics, i.e., T_s, T_u, and $T_{i/d}$. T_s evenly contains two *item-read*s on table Y and two *predicate-read*s on table Y and Z; T_u contains four *update*s on table Y; $T_{i/d}$ contains two *insert*s and two *delete*s on table Y. All of them are designed to quantitatively evaluate the validity of workload generated by *DBStorm*. By default, the ratios of T_s, T_u and $T_{i/d}$ are 60%, 20% and 20%, respectively. Considering Δ-*tolerant stable attribute*, the values in $attr_0$ of table Y are sampled from a $Uniform$ distribution on a domain $\{China, Japan, Russia, Britain\}$. Considering *partition-based primary key*, there is 20% dynamic keys in table Y. *YCSB-SQL* issues 24 threads to load 4.8K transactions under *Serializable*. Note that *YCSB-SQL* can be generated by *DBStorm* with a simple *Stochastic Model* demonstrated in [14].

Baselines. Fuzz is a popular way to generate test cases [34]. We compare *DBStorm* with three state-of-the-art *DBMS* fuzzers on both workload validity and code coverage, including two generation-based fuzzers *SQLSmith* [12] and *SQLancer* [39], and one mutation-based fuzzer *Squirrel* [51]. We compile and run those tools with their default configurations.



(a) Varying Number of Threads (b) Varying Number of Records

Figure 8: α Validity in YCSB-SQL

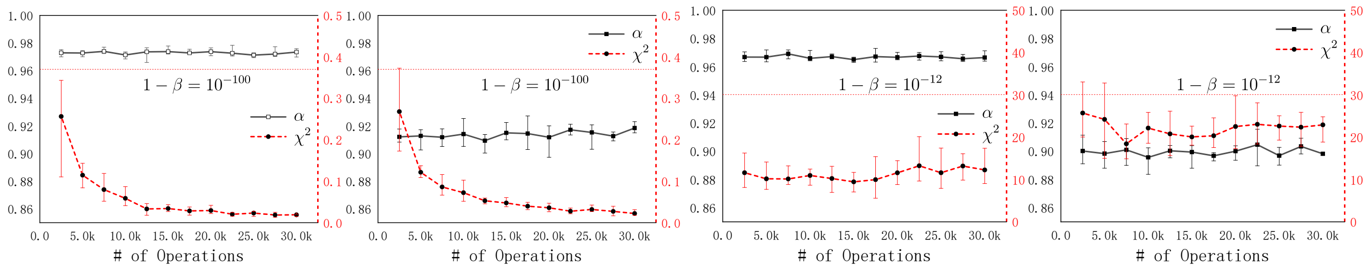
7.1 Workload Validity

As described in Sec. 2, we define workload validity by accessibility α and distribution fit goodness $\beta(N_t)$. In this section, we first evaluate the workload validity generated by *DBStorm*. Then we compare *DBStorm* with state-of-the-art work with respect to workload validity.

7.1.1 α Validity. *Miniature shadow* incorporates three *deterministic data generation* mechanisms to generate a valid workload. To explore its effectiveness, we run *YCSB-SQL* on PostgreSQL. Then we calculate α of each type of operations, including *item-read*, *predicate-read*, *update*, *insert* and *delete*. Since there are the same principles to instantiate parameters in *item-read* and *update*, we merge than as *item-read/update*, which is the same for *insert* and *delete*. The results are shown in Fig. 8. Because the parameters in a *predicate-read* are sampled from a theoretical data distribution, its α fluctuates around 80% even after a large number of transactions are executed. *Miniature shadow* takes timeout mechanisms to deal with the deadlock caused by out-of-order locking on dynamic keys, which may lead to invalid operations for *item-read/update* and *insert/delete*. For *insert/delete*, we make conversion between *insert* and *delete* to reduce the times to acquire read locks and to avoid the failure of instantiating parameters. Thus, α of *insert/delete* is better than *item-read/update*.

As increasing the number of threads, because the parameters in *insert/delete* are sampled from dynamic keys that are protected by locking during execution, the deadlock among threads is more intensive, which decreases the validity as shown in Fig. 8(a). However, we can decrease the contention by increasing the number of records in table Y to improve its validity as shown in Fig. 8(b). In contrast, since the parameters in *item-read/update* may sample from static keys, there have a slight impact on α as varying the number of threads and records, as shown in Fig. 8(a) and Fig. 8(b). In summary, *deterministic data generation* mechanisms are effective to make different types of operations access records.

7.1.2 α and β Validities. Since *predicate-read* accesses records with complex semantics, it is difficult to control its β validity as discussed in Sec. 6. To demonstrate the workload validity from both α and β, we disable *predicate-read* in *YCSB-SQL* and then run it on PostgreSQL. In Fig. 9, we demonstrate both α and β under different theoretical access distributions, i.e., $F \sim Uniform/Zipfian(0.4)$. From the results, we can see that α is close to 1 and is stable, as varying the number of operations in workload. Therefore, *DBStorm* is effective in guaranteeing operations to access records. However,



(a) 8 Threads, $F \sim Uniform$ (b) $F \sim Uniform$ (c) 8 Threads, $F \sim Zipfian(0.4)$ (d) $F \sim Zipfian(0.4)$

Figure 9: α and β Validities in YCSB-SQL w/o *Predicate-read*

under a higher contention, i.e., by more threads, *DBStorm* has to deal with more intensive contentions on dynamic keys, and then it causes a slight drop in α, as shown in Fig. 9(b) and 9(d). In a $Uniform$ distribution, *DBStorm* can bound the distribution deviation $1-\beta$ below 10^{-100}, as shown in Fig. 9(b). In a $Zipfian$ distribution with skewness 0.4, β is still larger than $1-10^{-12}$, as shown in Fig.9(d). In summary, it is effective for *DBStorm* to apply three *deterministic data generation* mechanisms to generate valid workloads.

7.1.3 Comparison with Baselines. Because *SQLSmith*, *SQLancer* and *Squirrel* can only generate workloads with a single thread, then we take a single thread generation mode of *DBStorm* for comparison. All tools run by their default configurations for 1 hour on PostgreSQL. Then we calculate the number of valid operation N_v, the number of operations N_o, and α validity, as in Tab. 3. From results, *DBStorm* outperforms *SQLSmith*, *SQLancer* and *Squirrel* not only in α, but also in the generating speed. *SQLSmith* aims to find crash bugs in query engine and considers little about semantic-correctness of generated query, so its α is as low as 1.58%, with less probability to touch the code of transaction processing. Because *Squirrel* is a mutation-based fuzzer and lacks a sound syntax/semantic-aware mutation strategy, its α is as low as 0.86%. *DBStorm* outperforms *Squirrel* by 8.9× in generation speed. *SQLancer* takes *pivoted query synthesis* to generate valid workloads with its α 71.32% and also fails in processing complex semantics in *predicate-read*. *DBStorm* with α=88.64% is 1.2× better than *SQLancer*. For generation speed, *DBStorm* is up to 7.1× faster than *SQLancer*. Thus *DBStorm* outperforms all of the state-of-the-art.

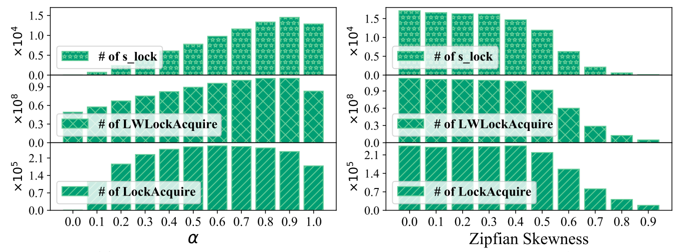
Table 3: Validity Comparison with Baselines

	N_v	N_o	$\alpha = N_v/N_o$
SQLSmith	6,150	387,139	1.58%
SQLancer	***92,000***	***129,000***	$\boxed{71.32\%}$
Squirrel	886	102,642	0.86%
DBStorm	***816,732***	***921,394***	$\boxed{88.64\%}$

7.2 Workload Validity Benefits

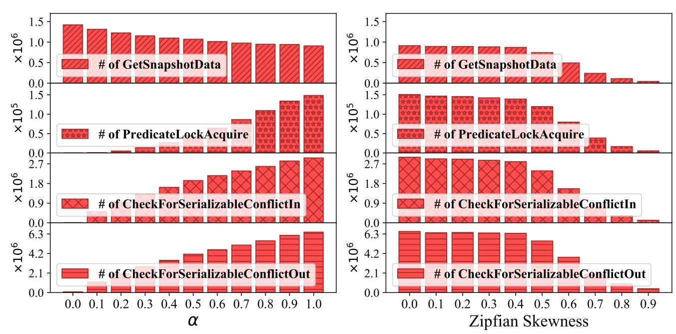
In this section, comparing with the baselines, we demonstrate the benefits of a valid workload generated by *DBStorm*.

7.2.1 α and β Validities Benefits. For inspecting the deep code logic of transaction processing, a valid workload is expected to run through all implementation layers of a database system, which triggers more intensive contentions, builds more complex transaction



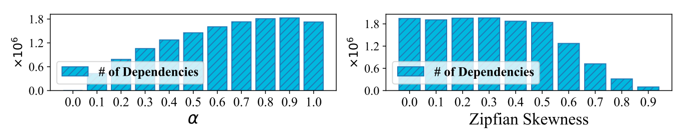
(a) Benefits from α Validity (b) Benefits from β Validity

Figure 10: Three Level Locks



(a) Benefits from β Validity (b) Benefits from β Validity

Figure 11: Serializable Snapshot Isolation

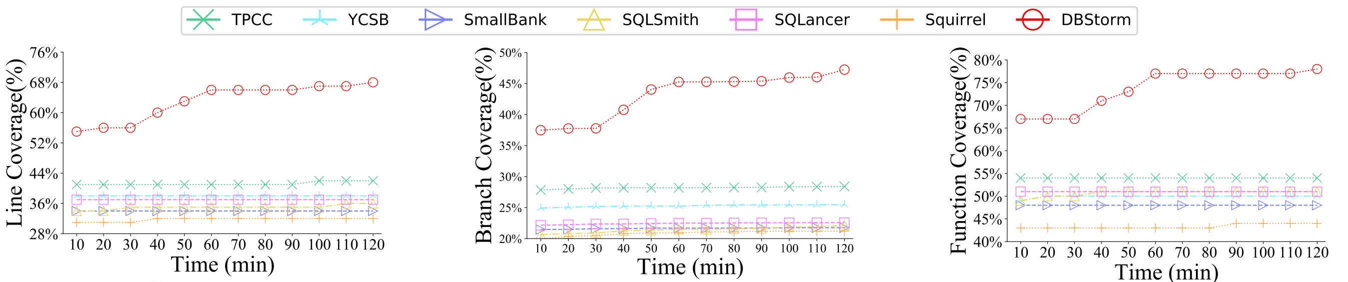


(a) Benefits from β Validity (b) Benefits from β Validity

Figure 12: Transaction Dependencies

dependencies and so on [17]. To demonstrate it practically, we vary α and β validities an in Fig. 8 and Fig. 11. In PostgreSQL, there are three levels of locks, i.e., *spin lock*, *light weight lock* and *regular lock*, which are crucial to handle contentions [38]. Thus we instrument source codes of PostgreSQL to fetch the number of acquiring for these four types locks, i.e., *s_lock*, *LWLockAcquire* and *LockAcquire*, as shown in Fig. 10(a) and 10(b).

To implement *Serializable Snapshot Isolation*, PostgreSQL takes *predicate Lock* to track read-write dependencies and then abort one of transaction in a dangerous structure [38]. Specifically, there are four critical functions about *predicate lock*, i.e., *PredicateLockAcquire*,



(a) Line Hit (b) Branch Hit (c) Function Hit

Figure 13: Coverage Comparison with Baselines

GetSnapshotData, *CheckForSerializableConflictIn* and *CheckForSerializableConflictOut*, as shown in Fig. 11(a) and 11(b).

Since dependencies between transactions also indicate the difficulty of transaction processing, we collect the number of dependencies during workload running. Because we can not control β validity of *predicate-read*, we disable *predicate-read* in *YCSB-SQL*.

In Fig 10(a), the number of acquiring locks grows firstly and then goes down as increasing α. The reason is that when α is small, the workload seldom accesses records and then has little requirement for locks; when increasing α, the valid workload frequently accessing records makes the time acquiring locks increase; when α reaches 0.9, the contention reduces the workload throughput, and then the number of acquiring locks goes down as well as the dependencies in Fig. 12(a). No matter whether the operation accesses the record or not, it should take a snapshot, which is indicates by the number of *GetSnapshotData* and is negatively correlated with the contention as shown in Fig. 11(a). The more intensive contention increases the number of read-write dependencies, which increase the number of *PredicateLockAcquire*, *CheckForSerializableConflictIn* and *CheckForSerializableConflictOut*, as shown in Fig. 11(a).

Besides, the theoretical access distribution of a workload also affects the contention, which is an important feature of testing transaction processing. To demonstrate it, we vary the theoretical access distribution by a $Zipfian$ skewness while setting $\alpha = 1$ and $\beta > 1 - 10^{-12}$. Notice that $\beta > 1 - 10^{-12}$ can be well achieved as shown in Fig. 9. From results, the number of acquiring locks goes down with increasing skewness, as in Fig. 10(b). The reason is that workload throughput is reduced by intensive contentions and then the number of transaction dependencies becomes less, including read-write dependencies, as in Fig. 11(b) and Fig. 12(b). In summary, both α and β validities decide contentions in a workload, so they are important factors in testing transaction processing, especially for concurrency control.

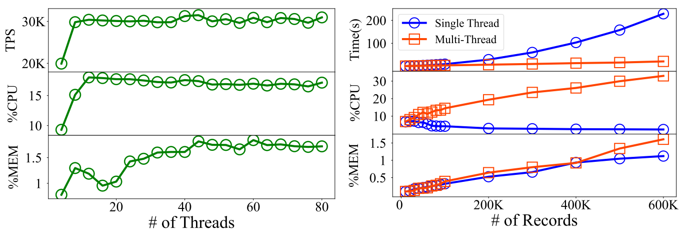
7.2.2 Comparing with Baselines. A testing tool with high code coverage suggests it has a lower chance of containing undetected bugs compared to a testing tool with low code coverage [18]. Based on a stochastic testing technique, *DBStorm* succeeds in generating a valid workload to test transaction processing comprehensively. To demonstrate the benefits of valid workload on improving code coverage, we compare *DBStorm* with three benchmarks and three state-of-the-art fuzzers. All tools generate 2 hour workloads with their default configurations (details in [14]), which are run on PostgreSQL. Since *DBStorm* focuses on testing transaction processing, we instrument four kernel modules in PostgreSQL mainly for transaction processing, i.e., locking manager (*lmgr*), transaction manager (*transam*), snapshot manager (*snapmgr*) and tuple visibility rules (*heapam_visibility*). *DBStorm* outperforms the baselines in terms of code coverage on lines, branches and functions as in Fig. 13. The results show that *DBStorm* not only achieves high coverage but also has advantages in generation speed.

In specific, because α of *Squirrel* is so small as in Tab. 3, it has the lowest coverage even lower than benchmarks. *TPC-C*, *Smallbank* and *YCSB* are well-known transactional benchmarks, which have fixed test cases, and then the code coverage is also fixed and limited. For example, we find *TPC-C* does not trigger the code on *serializable snapshot isolation* [38]. *DBStorm* surpasses *SQLancer* by about 1.5× in code coverage of transaction processing even though it has comparable α to *ours* in Tab. 3. And our predominance grows as we keep generating test cases. A low validity on generated workloads from *SQLSmith* leads to low coverage. Taking advantage of stochastic testing techniques, *DBStorm* can generate more diverse workloads than three benchmarks. Since the generated workload is valid, so *DBStorm* can trigger more codes on transaction processing than other fuzzers. Overall, *DBStorm* outperforms the tools in code coverage.

7.3 Generation Cost

In Fig. 14(a) and Fig. 14(b), we take *YCSB-SQL* to demonstrate the cost of *DBStorm* on workload generation and database population. In Fig. 14(a), as varying the number of threads from 4 to 80, we collect the throughput of workload, i.e., transaction per second *TPS*, and resources utilization by *%CPU* and *%MEM*. *DBStorm* generates up to 30K *TPS* with only 12 threads. On the same hardware and workload, the transaction processing capability of PostgreSQL is 10K. However, because of contention processing in *miniature shadow*, *TPS* does not increase even though we increase the generation threads. Even though the number of threads increases, the consumed CPU utilization is below 20%, and the consumed memory utilization is below 2%. The reason is that *DBStorm* takes the *deterministic data generation* mechanisms for instantiating transactions, which is cost-effective for all resource utilization.

Distribution constrained function provides a non-contention way to populate the database even with multi-threads. There are two strategies, i.e., *single thread* and *multi-threads*, for database population. Notice that there are 24 threads in the strategy of *multi-threads*. In Fig. 14(b), as varying the number of records of table Y from 10 to 600K, we collect the generation time and resources utilization by *%CPU* and *%MEM*. In *single thread*, the generation time increases with the number of records superlinearly; in contrast, by *multi-threads*, the generation time is linear scalability with the number of records. Since *multi-threads* takes more threads to speed up the generation, its CPU utilization is larger than the single-thread generation. Taking advantage of *deterministic data generation* mechanisms, the memory utilization is below 1.2%. In summary, *DBStorm* imposes modest hardware resources and has a fast generation speed on both database population and workload generation, so it is cost-effective for testing transaction processing.



(a) Workload Generation (b) Database Population

Figure 14: Generation Cost

8 RELATED WORK

Generation-based Testing for DBMS. *SQLsmith* [12] collects the schema from tested database systems to generate *select* only, which restricts its code coverage on transaction processing. *SQLancer* [39] generates queries to fetch a specific target row; if it fails, bugs likely exist in *DBMS*s under test. RAGS [42] takes a random way to generate queries for differential testing to detect correctness bugs in *DBMS*s. *APOLLO* [28] generates queries for two versions of the same *DBMS* to check performance bugs. Generation-based testing usually generates queries randomly and serially with an assumption of an unchangeable database state, which is impossible for transaction processing.

Mutation-based Testing for DBMS. According to the execution feedback of queries on *DBMS*, *GARan* [19] uses a genetic algorithm to mutate queries for improving the code coverage of query optimization/execution instead of transaction processing. *Ratel* [47] improves the code coverage by collecting coverage feedback to mutate queries. *Squirrel* [51] aims at finding memory corruption in *DBMS*s by mutating queries from a seed pool; it does not ensure the semantic and syntax correctness during mutation. Although mutation-based testing provides an effective way to instructively generate new cases by taking feedback from previous test cases, they are unaware of the structure of test cases during mutation, which often leads to invalid test cases, especially for transaction processing. In contrast, *DBStorm* can automatically generate a valid workload for testing transaction processing.

Consistency Anomaly Detection. *Cobra* [45] uses SMT solver to verify only the serializability in key-value stores under a special workload by injecting fence transactions, which leads to critical usage limitations. Kingsbury has developed *Jepsen*, a framework to test the safety properties of distributed systems[1]. As part of *Jepsen*, *Elle* [29] carefully designs the workload with which the verification is highly coupled, so it can not carry out the verification for diverse workloads. *IsoDiff* [26] debugs anomalies caused by weak isolations, which reduces the cost of cycle detection by searching representative subsets. *Rushmon* [40] checks anomalies by sampling for high efficiency, which may leak some anomalies. *ConsAD* [49] locates consistency anomalies by checking cycles in the dependency graph. However, *ConsAD* spends a lot of manpower to transform the application to complete the dependency tracking. The main problem with this work is the lack of ability to support consistent anomaly detection in a black-box mode under an arbitrary transactional workload.

9 CONCLUSION AND FUTURE WORK

In this paper, we have presented *DBStorm*, a cost-effective valid workload generator for testing transaction processing. To test complex code logic of transaction processing, we take a stochastic-based generation mechanism to achieve high coverage; we design the *deterministic data generation* mechanisms to guarantee the validity of generated workload; we sketch a novel resource conservative *miniature shadow* structure to generate a valid workload at runtime. We have provided sufficient proof to illustrate the effectiveness and efficiency of our design theoretically. In practice, *DBStorm* has exposed some critical bugs [14] in commercial database systems. Though *DBStorm* outperforms the related work, it still has several limitations. The stochastic testing technique expects to cover all test spaces probabilistically, but the generation lacks guidance, which may lower testing efficiency. A workload containing a large scale of operations deteriorates the indeterminacy of transaction processing, which makes bug reproduction tough work. To make *DBStorm* strong enough, we will proceed to study these problems.

REFERENCES

[1] 2020. Jepsen. https://github.com/jepsen-io/jepsen
[2] 2020. MySQL bugs. https://bugs.mysql.com/bug.php
[3] 2020. PingCAP TiDB Bug. https://github.com/pingcap/tidb/issues
[4] 2020. PostgreSQL Release Notes. https://www.postgresql.org/docs/release/
[5] 2020. TATP Benchmark. http://tatpbenchmark.sourceforge.net
[6] 2020. TPC-C Benchmark. http://www.tpc.org/tpcc/
[7] 2020. TPC-E Benchmark. http://www.tpc.org/tpce/
[8] 2021. Cockroachdb Bugs. https://github.com/cockroachdb/cockroach/issues
[9] 2021. How SQLite Is Tested. https://www.sqlite.org/testing.html
[10] 2021. Jepsen: PostgreSQL 12.3. https://jepsen.io/analyses/postgresql-12.3
[11] 2021. PostgreSQL Regression Tests. https://www.postgresql.org/docs/current/regress-run.html
[12] 2021. sqlsmith. https://github.com/anse1/sqlsmith
[13] 2021. Yugabyte Bugs. https://github.com/yugabyte/yugabyte-db/issues
[14] 2022. DBStorm. https://zenodo.org/record/6526423#.YnYtXuhBxnJ
[15] 2022. Netflix Chaos Monkey. https://netflix.github.io/chaosmonkey/
[16] 2022. PingCAP Chaos-Mesh. https://chaos-mesh.org/
[17] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering*. IEEE, 67–78.
[18] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.
[19] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. 2007. A genetic approach for random testing of database systems. In *Proceedings of the VLDB Endowment*. 1243–1251.
[20] George Casella and Hoger L.Berger. 2021. Statistical lnference. In *Cengage Learning*. 54.
[21] Edgar F Codd. 2002. A relational model of data for large shared data banks. In *Software pioneers*. Springer, 263–294.
[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
[23] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.
[24] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.
[25] Kevin P Gaffney, Robert Claus, and Jignesh M Patel. 2021. Database isolation by scheduling. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1467–1480.
[26] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: debugging anomalies caused by weak isolation. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2773–2786.
[27] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.
[28] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70.
[29] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554* (2020).
[30] Yuming Li, Rong Zhang, Yuchen Li, Ke Shu, Shuyan Zhang, and Aoying Zhou. 2019. Lauca: Generating Application-Oriented Synthetic Workloads. *arXiv preprint arXiv:1912.07172* (2019).
[31] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2047–2060.
[32] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.
[33] Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–24.
[34] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).
[35] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.
[36] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/-analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1771–1775.
[37] P.Massart. 1990. The Tight Constrant in the Dvoretzky-Kiefer-Wolfowitz Inequality. In *The Annals of Probability*. 1269–1283.
[38] Dan RK Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *arXiv preprint arXiv:1208.4179* (2012).
[39] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation*. 667–682.
[40] Zechao Shang, Jeffrey Xu Yu, and Aaron J Elmore. 2018. RushMon: Real-time isolation anomalies monitoring. In *Proceedings of the 2018 International Conference on Management of Data*. 647–662.
[41] Jun Shao. 2003. Mathematical Statistics. In *Springer Texts in Statistics*.
[42] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *Proceedings of the VLDB Endowment*. 618–622.
[43] Keith Stobie. 2005. Too darned big to test. *Queue* 3, 1 (2005), 30–37.
[44] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. CockroachDB: The resilient geo-distributed SQL database. In *SIGMOD*. 1493–1509.
[45] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *14th Symposium on Operating Systems Design and Implementation*. 63–80.
[46] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.
[47] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 328–337.
[48] Todd Warszawski and Peter Bailis. 2017. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 5–20.
[49] Kamal Zellag and Bettina Kemme. 2011. Real-time quantification and classification of consistency anomalies in multi-tier architectures. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 613–624.
[50] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. 2014. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation*. 449–464.
[51] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.