Lauca: Generating Application-Oriented Synthetic Workloads

Abstract—The synthetic workload is essential and critical to the performance evaluation of database systems. When evaluating the database performance for a specific application, the similarity between synthetic workload and real application workload determines the credibility of evaluation results. However, the workload currently used for performance evaluation is difficult to have the same workload characteristics as the target application, which leads to inaccurate evaluation results. To address this problem, we propose a workload duplicator (Lauca) that can generate synthetic workloads with highly similar performance metrics for specific applications. To the best of our knowledge, Lauca is the first application-oriented transactional workload generator. By carefully studying the application-oriented synthetic workload generation problem, we present the key workload characteristics (transaction logic and data access distribution) of online transaction processing (OLTP) applications, and propose novel workload characterization and generation algorithms, which guarantee the high fidelity of synthetic workloads. We conduct extensive experiments using workloads from TPC-C, SmallBank and micro benchmarks on both MySQL and PostgreSQL databases, and experimental results show that Lauca consistently generates high-quality synthetic workloads.

Index Terms—Performance evaluation, synthetic workload, OLTP applications

I. INTRODUCTION

Performance evaluation is essential to the development of database management systems (DBMSs). Application-oriented database performance evaluation refers to the evaluation of database system's capability for a *specific* application [1]. The application scenarios include the followings:

Database selection. DBMS is the basic support system for applications, hence the application developer needs to select an appropriate system with satisfactory performance for their target workloads. It is an effective means to evaluate candidate DBMSs by using the test workload constructed in accordance with the actual application workload.

PoC testing. PoC (proof of concept) testing is indispensable for database vendors, especially some nascent database vendors, when marketing their products to application companies. Building a test workload that is as similar as possible to the counterpart's application is the most powerful guarantee for PoC conclusions.

Application-oriented performance optimization. Generally we can find that for application companies, the database performance is not as good as the expectation for online workloads; but for database vendors, developers can not reproduce the performance problems during the optimization because of lack of ideal evaluation workloads.

At present, there are three main ways to evaluate database performance. The evaluation workloads are real online work-

loads, standard benchmark workloads and synthetic workloads, respectively. However, existing approaches are suboptimal for application-oriented database performance evaluation due to the following deficiencies:

Data privacy issue. Due to privacy concerns, it is often not possible for database vendors to obtain the real online workloads from their clients to evaluate the system performance. In fact, for the company's own testers, data privacy protection is also cumbersome. Therefore, the evaluations have to be done using simulation workloads.

Application oblivious. One can employ standard benchmarks to evaluate the database performance without the data privacy issue. Nevertheless, standard benchmarks are designed for general-purpose evaluations and their workloads are too general to imitate a specific application, and thus produce inaccurate evaluation results.

There have been some recent works on generating application-oriented synthetic workloads for online analytical processing (OLAP) applications [4], [5]. However, it is still an unexplored area against OLTP applications, where the gap motivates this work.

In this paper, we propose Lauca, a transactional workload generator for application-oriented database performance evaluation. The essential goal is that the synthetic workload generated by Lauca is as similar as possible to the real application workload, so that the performance metrics evaluated by Lauca are more informative. In order to achieve this goal, we define the key workload characteristics of OLTP applications, i.e., *transaction logic* and *data access distribution*, which determine the critical runtime behaviors of OLTP applications on database systems, including transaction conflict intensity, deadlock possibility, distributed transaction ratio and cache hit ratio. Transaction logic is first time proposed for catching potential business logic for exquisite workload simulation; data access distribution is described from the perspectives of skewness, dynamics and continuity to promise the reality of synthetic workloads.

To address the data privacy issue, Lauca is designed to isolate production environment and evaluation environment, as shown in Fig. 1, where components involving real application data are executed in the production environment by data owners. By analyzing the workload traces, we can extract the transaction logic and data access distribution of target application workload, which are used to generate the synthetic workloads for evaluation purposes. Database schema and data characteristics are extracted from the real database instance to generate the synthetic database on evaluation side.
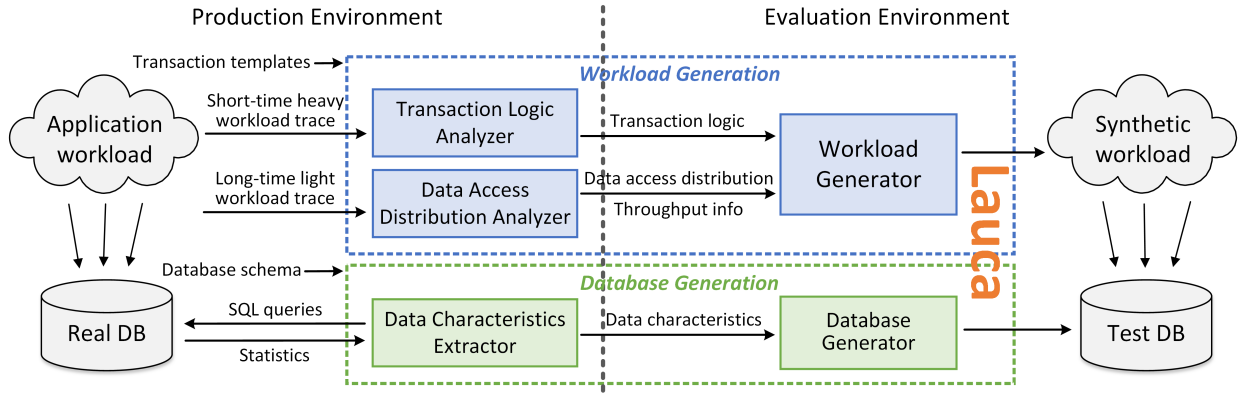
Fig. 1. The overall architecture and workflow of Lauca

To the best of our knowledge, Lauca is the first transactional workload generator for application-oriented database performance evaluation. Extensive experimental results show that transaction logic and data access distribution we proposed can effectively characterize the workloads of OLTP applications, and the deviations in performance metrics between the synthetic workload generated by Lauca and the real application workload are consistently less than 10%.

II. PRELIMINARIES

A. Problem Definition

Before giving the formal problem definition, we enumerate the natural requirements of application-oriented database performance evaluation as follows.

Fidelity. The evaluation workload should be highly similar to the real application workload. **Performance metrics** (e.g., *throughput*, *latency*, *utilizations of various physical resources*, etc) obtained by the evaluation are expected to be the same as running in the real application environment. The *similarity* between evaluation workload and real application workload is measured by the *deviations in performance metrics*. The smaller the deviation, the higher the similarity.

Sheltering. Data privacy protection is the basic requirement for commercial applications, so real data and workloads generally cannot be directly used for database performance testing.

Scalability. The target application may have huge data scale and high request concurrency/throughput. It requires that the workload generation toolset is scalable to multiple nodes and can support parallel database and workload generation.

Extending. Sometimes, we need to extend the current application workload to measure the database performance under synthetic workloads with the expected scale. Since this paper is aimed at transactional workloads, the main extending directions are request concurrency and request throughput.

Based on these requirements, we formulate the problem of application-oriented synthetic workload generation in Def. 1.

Definition 1: **Application-oriented synthetic workload generation:** Generating synthetic workloads that are highly similar to the target application workloads, which is not only required to have small deviations in performance metrics on
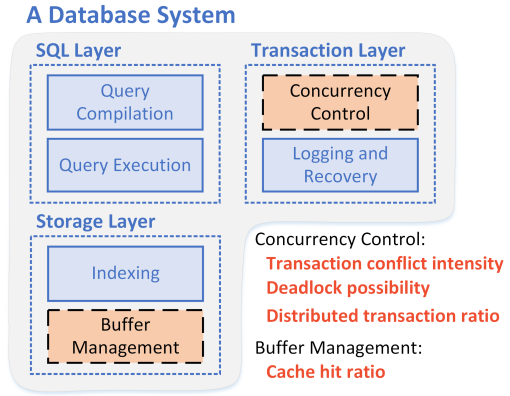

Fig. 2. Database-side workload characteristics that may be affected

the databases, but also to promise the properties of sheltering, scalability and extending.

B. Workload Characteristics

In order to make the synthetic workload and the real application workload have the same execution cost on the same database system and thus have the same performance metrics, we must analyze which workload characteristics need our attention and control during synthetic workload generation. Notice that our work is to simulate workloads for a specific application upon a certain database system, some factors are fixed, such as database schema, transaction templates, the implementation mechanisms of DBMS, etc. The *transaction template* consists of several SQL operations with symbol parameters, and sometimes contains conditional branches and loop structures. We generate the synthetic workload by instantiating the symbol parameters in the transaction templates.

TABLE I
APPLICATION-SIDE WORKLOAD CHARACTERISTICS

DataBase-side workload characteristics	Application-side workload characteristics
Transaction conflict intensity	Transaction logic, Data access distribution
Deadlock possibility	Transaction logic, Data access distribution
Distributed transaction ratio	Transaction logic
Cache hit ratio	Data access distribution

As can be seen from the Fig. 2, a database system is mainly divided into three parts, i.e., SQL Layer, Transaction Layer and Storage Layer. Since we focus on transactional workloads which commonly involve no query optimizations, the generation strategies for synthetic workloads have little effect on the execution cost of SQL layer. But in transaction layer, the workload generation strategy has a significant impact on the execution cost of Concurrency Control module. For a stand-alone database system, the main workload characteristics that may be affected here are transaction conflict intensity and deadlock possibility. If it is a distributed database system, we also need to pay attention to the impact on distributed transaction ratio. For a given database system, transaction log volume is the main performance factor to Logging and Recovery module. And the log volume is determined by transaction templates and data characteristics, which is also fixed for a given application. Indexing and Buffer Management are two main modules in storage layer. When database schema and transaction templates are fixed, the execution cost of indexing is certain. However cache hit ratio in buffer management module is heavily influenced by workload generation strategies, which shall be taken seriously.

In order to ensure that the synthetic workload and the real application workload are consistent on these four database-site workload characteristics, we define and manipulate two application-side workload characteristics, namely transaction logic and data access distribution. Transaction logic depicts the relationship among SQL parameters and return items, and it is a representation of the potential business logic of target applications. The potential business logics, such as data items accessed by different SQL operations in a transaction satisfy a certain correlation in a probability, usually cannot be seen at a glance and need to be obtained by analyzing the workload trace. Transaction logic has a big impact on transaction conflict intensity, deadlock possibility and distributed transaction ratio. The technical details about transaction logic are available in Section IV. Data access distribution is used to characterize the access distribution of SQL operations on data items. We analyze the workload trace to get the data access distribution, namely the data distributions of SQL parameters, for parameter instantiation during the workload generation. Data access distribution has a big impact on transaction conflict intensity, deadlock possibility and cache hit ratio. Section V presents all the details about data access distribution.

Though in some DB implementations, the transaction layer and the storage layer can be merged together, factors affecting DB performance are still consistent with our previous analysis. The correspondence between Database-side workload characteristic and Application-side workload characteristics are summarized in Table I.

C. Overview of Lauca

The infrastructure of Lauca can be divided into two types of components, which are running in the Production Environment or the Evaluation Environment, as shown in Fig. 1. Below we will briefly describe the workflow of Lauca from the two
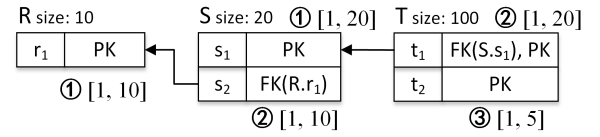

Fig. 3. Determining the domains of key columns

lines of synthetic database generation and synthetic workload generation.

Database Generation. The input of Database Generator mainly consists of two parts: database schema and data characteristics. Since data characteristics are tedious and need to be obtained from the real database, we provide a Data Characteristics Extractor which can help us get this information automatically by using simple SQL queries. In Lauca, we only focus on some basic data characteristics (e.g., column domain) of the test database, because the workload characteristics of synthetic workload are the key factors affecting the database performance. The technical details are available in Section III.

Workload Generation. Transaction templates of the target application are one of the inputs to the Workload Generator. After instantiating all the symbol parameters in the transaction template with concrete values, we can get a concrete transaction that can be run on the test database. In order to guarantee the similarity between the synthetic workload and the real application workload, parameter instantiation in workload generator satisfies the transaction logic and data access distribution respectively extracted by Transaction Logic Analyzer and Data Access Distribution Analyzer. The input of transaction logic analyzer is short-time (e.g., ten minutes) heavy workload traces, which contain all the parameters and return items of each SQL operation. The input of data access distribution analyzer is long-time (e.g., a full workload cycle) light workload traces, which contain only pivotal SQL parameters. The workload trace can be logged by application clients or the database server. The related technical details are available in Section VI.

III. DATABASE GENERATION

All data characteristics needed for database generation are the same as Touchstone [5], such as table size, column domain, column cardinality of unique values, etc. The generation of test database is actually the generation of multiple tables, while satisfying primary/foreign key constraints, as well as data characteristics of non-key columns.

Without loss of generality, we assume that primary/foreign key columns are integers. The primary key is the identifier of the record and generally has no physical meaning, so we don't need to consider the data characteristics of these key columns. We simply generate primary keys sequentially and generate foreign keys within the referenced primary key domain randomly, which can promise the uniqueness of primary keys and the referential integrity of foreign keys. Before the generation of key columns, there are three steps to determine their domains, with an example shown in Fig. 3. First, if the primary key contains only a single column, the domain is $[1, s]$, where s is the table size (as ① in Fig. 3). Then, the

domains of foreign key columns can be determined by the domains of referenced primary key columns (as ② in Fig. 3). In the third step, we settle down the domains of non-foreign key columns in the composite primary key, e.g., column $T.t_2$ (as ③ in Fig. 3). The most usual and reasonable situation is that there is only one non-foreign key column in the composite primary key. And the domain of this column is $[1, \frac{s}{\prod |d_{fk}|}]$, where d_{fk} is the domain of one of the foreign key columns within composite primary key. Other situations are handled similarly. Due to the cascaded reference, the second and third steps may need to be performed multiple times.

The random column generator [5], which contains a random index generator and a value transformer, is used to generate values for non-key columns, while satisfying the desired data characteristics, especially the cardinality of unique values. The output of random index generator is the integers from 1 to n, where n is the column cardinality. Given an index, the transformer deterministically maps it to a value in the domain of the column. We adopt different transformers based on the data type of the column. For numeric types, e.g., integer, we simply pick up a linear function which uniformly maps the index to the column domain. For string types, e.g., varchar, there are some seed strings pregenerated randomly, which satisfy the length requirements. We first select a seed string based on the input index, such as the $(i\%k)^{th}$ seed string, where i is the index and k is the number of all seed strings. Then we concatenate the index and the selected seed string as the output value.

In summary, the generation of each table is independent of each other. And for each table, we can achieve parallel data generation with multiple threads on multiple nodes by assigning a primary key range to each thread.

IV. Transaction Logic

The transaction logic proposed in this paper is not an extension of predicate logic in work [6], but rather a representation of potential business logic for OLTP applications. In this section, we first give an intuitive explanation of why transaction logic matters through a concrete example; then we clearly give a formal definition of transaction logic; finally, we present how to efficiently extract transaction logic from the workload trace.

A. Intuitive View

The following is a sample transaction template (abbr., TT) for illustrating the necessity of transaction logic.

Start Transaction
update S set $s_3 = s_3 +$ *?* where $s_1 =$ *?*;
update T set $t_3 = t_3 +$ *?* where $t_1 =$ *?* and $t_2 =$ *?*;
select r_2, r_3 from R where $r_1 =$ *?*;
update R set $r_2 = r_2$ - *?* where $r_1 =$ *?*;
Commit

The tables in TT can refer to Fig. 3. r_2, r_3, s_3 and t_3 are double typed non-key columns. Now suppose we have deployed a distributed database, and tables R, S, and T are all hash-partitioned by primary keys. When generating the

synthetic workload based on TT, we usually instantiate the SQL parameters with random values generated by given data distributions. However, there may be significant performance difference between such synthetic workload and the real application workload. The real workload usually has correlations among parameters in predicates to promise business logics. For example, the parameters in predicates '$s_1 =$?' and '$t_1 =$?' are the same in a high probability (e.g., 99%) in actually. Therefore, the first two SQL operations in TT do not bring many distributed transactions. But in the synthetic workload, the above two parameters are randomly generated, and they are likely to be different, which will lead to serious distributed transactions. As we know, the ratio of distributed transactions has a crucial impact on database performance [7]. More than that, the parameters in predicates of the last two SQL operations in TT, i.e., '$r_1 =$?', are always the same in real workload, so there is no deadlock in actually. However, these two parameters may be different during the generation of synthetic workload, resulting in deadlocks possibly. The occurrence of deadlocks can seriously affect the database performance. Through this example, it is intuitive that the potential business logic of the target application, namely the transaction logic we proposed, is an important workload characteristics for synthetic workload generation, and must be guaranteed to be consistent with the real application workload.

B. Definition of Transaction Logic

The workloads of actual applications vary greatly, and the potential business logic is even more difficult to depict. Therefore, the definition of transaction logic must jump out of the perspective of application business. We can define the transaction logic using *transaction structure information* and *parameter dependency information*. Branches and loops are common structures in transactions. The probability that each branch will be executed and the average numbers that loop operations will be executed have important influence on transaction execution cost. These need to be described in the transaction structure information.

The relationship among SQL parameters and return items in a transaction template decides the hidden semantics among/in SQL operations. There are four types of relationships we care about after the investigation of existed OLTP benchmark workloads and actual application workloads around. Firstly, '**equal relationship**' is the most common one. For example, two SQL parameters are equal under a certain probability. Secondly, '**inclusive relationship**' is also familiar. Because the SQL result set may be a set of tuples, the value of a SQL parameter might be an element in returned values of one previous return item. Thirdly, '**linear relationship**' is a complement and extension to the equal relationship, which has stronger expressive ability. Fourthly, '**between relationship**' is proposed for predicates '*col between* p_1 *and* p_2' and '*col* $\geq p_1$ *and col* $\leq p_2$', where p_2 has a between relationship with p_1. In order to simplify the problem, we only consider the relationship (i.e., dependency) between current parameter and the previous parameters (or return items).

Now we give the formal definition of transaction logic in Def.2. Let's start with some terminology. O_i represents the ith operation in the transaction template; $p_{i,j}$ (resp., $r_{i,j}$) is the jth parameter (resp., return item) of O_i, where i and j are both count from 1. We use the abbreviations ER, IR, LR and BR to denote the equal relationship, inclusive relationship, linear relationship and between relationship, respectively.

Definition 2: **Transaction logic:** For a transaction template, the transaction logic consists of transaction structure information and parameter dependency information, which are specified as:

- Transaction structure:
  #1 Execution probability of each branch in the branch structures.
  #2 Average numbers of executions for operations in the loop structures.
- Parameter dependencies for each parameter $p_{i,j}$:
  $1 [p_{i,l}, p_{i,j}, \text{BR}, \Delta]$
  $2 A list of *dep-item*, where *dep-item* $\in \{$
      $[p_{m,n}, p_{i,j}, \text{ER}, \xi]$, $[p_{m,n}, p_{i,j}, \text{LR}, \xi, (a,b)]$,
      $[r_{x,y}, p_{i,j}, \text{ER}, \xi]$, $[r_{x,y}, p_{i,j}, \text{LR}, \xi, (a,b)]$,
      $[r_{x,y}, p_{i,j}, \text{IR}, \xi]$ $\}$;
      $m \le i$; $x < i$; and if $m = i$, then $n < j$.
  $3 A list of $[p_{i,j}, \text{LR}, \xi, (a,b)]$, where O_i must be an operation in the loop structure.

Note that if a parameter $p_{i,j}$ has the dependency $1, then both dependencies $2 and $3 do not need to exist because $p_{i,j}$ can be represented by $(p_{i,l} + \Delta)$, where Δ is the average increment counted from the workload trace. In a dependency, ξ is the probability that the corresponding dependency is satisfied; (a,b) are two coefficients for characterizing the linear relationship. For example in Section IV-A, the dependencies $2 of parameter $p_{2,2}$ in TT includes $[p_{1,2}, p_{2,2}, \text{ER}, 99\%]$, and there is dependency $[p_{3,1}, p_{4,2}, \text{ER}, 100\%]$ for parameter $p_{4,2}$. Moreover, since the operations in the loop structure usually are performed multiple times, we need to consider the changes of SQL parameters in these operations during the loop execution. Dependencies $3 are used to represent the linear relationships between the values of the same parameter in successive runs for loop operations. For example, $[p_{i,j}, \text{LR}, 90\%, (2,1)]$ indicates that there is a 90% probability that the value of $p_{i,j}$ in current loop execution is $(2 * v + 1)$ where v is the value of $p_{i,j}$ in previous loop execution. If a parameter that is not in an operation inside a loop structure, it can only have dependency $1 or dependencies $2.

C. Extraction Algorithm

The transaction logic is the embodiment of application business logic, so it generally does not change frequently over time. Therefore, the extraction of transaction logic does not require workload traces across a long time. Since the transaction logic analysis of each transaction template is independent of each other, the following extraction algorithm will be introduced for a single transaction template. Our extraction algorithm consists of six steps. The input is a transaction template and corresponding workload traces of K transaction instances.

Step 1: Structure information. By traversing workload traces, we can count the number of executions of each operation in the transaction template, and use it to calculate the execution probability of each branch and the average numbers of executions for loop operations.

Step 2: Identify BR. First, we identify all the parameter pairs $<p_{i,l}, p_{i,j}>$ that satisfy the between relationship in the transaction template, and then traverse the workload trace to get the average increment (i.e., Δ). After that, we construct the dependency $1 for $p_{i,j}$, and the subsequent steps 3-6 can skip the processing of $p_{i,j}$.

Step 3: Collect statistics for ER and IR. For each parameter $p_{i,j}$ in the transaction template, we traverse its previous parameters $p_{m,n}$ (resp., return items $r_{x,y}$), and count the number of transaction instances in which the pairs $<p_{m,n}, p_{i,j}>$ (resp., $<r_{x,y}, p_{i,j}>$) satisfy the equal relationship (resp., equal relationship or inclusive relationship). $[<r_{2,3}, p_{4,1}>, \text{ER}, 100]$ is an example statistical result indicating that there are 100 ones among the K transaction instances where $p_{4,1}$ and $r_{2,3}$ are equal. Assuming that there are totally V parameters together with return items in the transaction template, there are nearly $\frac{V^2}{2}$ numbers of $<p_{m,n}, p_{i,j}>$ and $<r_{x,y}, p_{i,j}>$ pairs. The complexity of this step is $O(KV^2)$.

Step 4: Collect statistics for LR. LR only involves numeric typed parameters and return items, and the return items must from the operations based on the primary key filtering. Since the calculation of coefficients (a,b) for LR requires two transaction instances, we randomly select N groups of transaction instances (two for each group) from K transaction instances. Then we calculate the coefficients (a,b) for each pair (i.e., $<p_{m,n}, p_{i,j}>$, $<r_{x,y}, p_{i,j}>$) based on the N groups of transaction instances. Notice that the LR with coefficients $(1,0)$ needs to be ignored here, because it has been represented by ER. $[<p_{2,1}, p_{3,2}>, \text{LR}, 100, (1,2)]$ is an example statistical result indicating that there are 100 ones among N groups of transaction instances where $p_{3,2} = p_{2,1} + 2$. The complexity of step 4 is $O(NV^2)$.

Step 5: Determine ER, IR and LR by trade-offs. With the statistics obtained in steps 3-4, we can easily construct the dependencies $2 for each parameter $p_{i,j}$. However, there may be a lot of dependencies for each parameter, and some may have very small probabilities (i.e., ξ), so we need to make a trade-off among these dependencies to remove noises and alleviate following calculations. We pick the most important dependencies, such as those with high probability, and ensure that the sum of probabilities of picked dependencies is less than 1. We are more inclined to reserve ER, and we can find that ER is much more important than IR and LR in experiment section.

Step 6: Construct LR for loop structure. For multiple runs of operations in a loop structure, we use dependencies $3 to characterize the change of parameters. By traversing workload traces, the value changes in successive runs are counted for each SQL parameter in the loop structure. The calculation
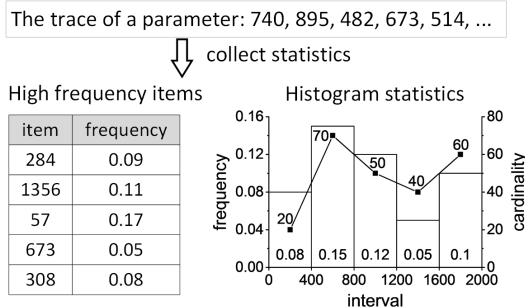
The trace of a parameter: 740, 895, 482, 673, 514, ...

⇩ collect statistics

High frequency items

item	frequency
284	0.09
1356	0.11
57	0.17
673	0.05
308	0.08

Histogram statistics

Fig. 4. An example for illustrating S-Dist

	high frequency items					all intervals				
	45	240	195	1280	385	0	1	2	3	4
frequency	0.09	0.11	0.17	0.05	0.08	0.08	0.15	0.12	0.05	0.1
cumu prob	0.09	0.2	0.37	0.42	0.5	0.58	0.73	0.85	0.9	1

E.g. 1: ranDouble[0, 1) = 0.39
0.37 <= 0.39 < 0.42
Output: 1280

E.g. 2: ranDouble[0, 1) = 0.76
0.73 <= 0.76 < 0.85
Output: a random value in interval 2

Fig. 5. An example of parameter generation

of coefficients (a, b) is similar as in step 4. Then based on the statistics, we construct the dependencies $3 and maintain it independently with dependencies $2. Suppose there is an example dependency $[p_{5,3}, LR, 90\%, (1, 1)]$, which indicates the probability that $p_{5,3}$ is increased by 1 relative to the value in last run is 90% during the loop execution.

One thing to note is that if steps 3-4 encounter an operation in the loop structure, only the trace data of the first execution is used. The complexity of our extraction algorithm is dominated by steps 3-4, which is $O(KV^2 + NV^2)$. And in practical evaluations, K and N can be set to tens of thousands, and V may be tens or hundreds, so the extraction of transaction logic is very fast. In our experiments, we found that the extraction of transaction logic often takes only a few seconds with $K = 10^4$ and $N = 10^4$.

V. DATA ACCESS DISTRIBUTION

Data access distribution has long been considered as an important application workload characteristic [8]. In this section, we present how to characterize and manipulate the skewness (Section V-A), dynamics (Section V-B) and continuity (Section V-C) of data access distribution for synthetic workload generation.

A. Skewed Data Access Distribution

As introduced in Section II-C, synthetic workload generation is actually the instantiation of symbolic parameters in transaction templates. Therefore the data access distribution of synthetic workload is determined by the values of these instantiated parameters. Our analysis of data access distribution only needs the light workload traces containing pivotal parameters which are used to index the data involved of SQL operations. In the following content, we present S-Dist for describing the skewness of data access, which has a serious impact on database performance.

Without loss of generality, we assume the form of predicates which determined the data access distribution, for OLTP application workloads, can be expressed as '*col op para*'. We use *high frequency items* (abbr., HFI) and *histogram statistics* (abbr., HS) extracted from the workload trace to represent S-Dist for each parameter. HFI records H hottest data items (i.e., concrete parameter values) with the highest occurrence frequency. The column domain is evenly divided into I intervals, and then the frequency and cardinality of the
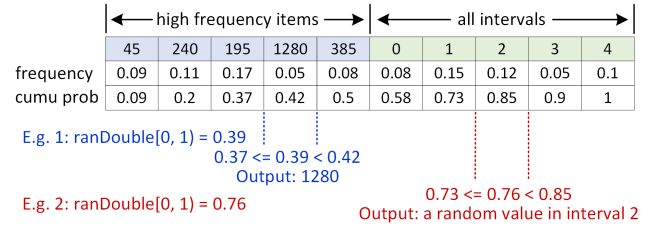
parameters, except the ones in HFI, falling on each interval are counted as HS. Two statistics, i.e., frequency and cardinality, are used to accurately characterize the access skewness for intervals. For instance, although the frequency of an interval is not very high, the cardinality of this interval is very small, and then parameter values in this interval can still cause high conflicts. Fig. 4 is an example of S-Dist. In this example, both H and I are 5, and the corresponding column is an integer with domain [0, 2000]. In HFI, the hottest item is 57 with frequency 0.17; in the first interval of HS, there are 20 unique parameter values with the total access frequency 0.08.

The data items in HFI of S-Dist are from the workload trace running on real database. However, the data in synthetic database generated in Section III is generally completely different from the data in real database. For example, the real data 57 in HFI in Fig. 4 may not exist in our synthetic database. So before instantiating the symbolic parameters using S-Dist, we need first do data transformation for HFI. Suppose the column generator for our example is 'index = ranInt[1, 400], value = index * 5', where 400 is the column cardinality. First, we regenerate the data items in HFI using our column generator, as shown in Fig. 5, where, for example, data item 57 is replaced by 195. Then, we calculate a cumulative probability array based on the frequencies of high frequency items and all intervals ('cumu prob' in Fig. 5). Finally, we use a random generator to generate values between 0 and 1, which can be mapped to the cumulative probability array and used to select appropriate parameter values for filling the predicate. In Fig. 5, there are two parameter generation examples. In this way, we can easily ensure that the generated parameters conform to the desired frequency distribution. The complexity of parameter generation is $O(\log(H+I))$, which is dominated by the binary search over the cumulative probability array.

Additionally, in order to control the cardinality of parameters generated on each interval, we redefine a random index generator for parameter generation, that is 'index = $\lfloor$ ranInt[0, cdn_i) / cdn_i * cdn_{avg} $\rfloor$ + $minIdx_i$', where cdn_i is the expected cardinality of the generated parameters on the target interval, cdn_{avg} is the average cardinality for each interval, and $minIdx_i$ is the minimum index of the target interval. The value transformer remains unchanged. In Fig. 5, the random index generator for interval 2 is 'index = $\lfloor$ ranInt[0, 50) / 50 * 80 $\rfloor$ + 161', with $cdn_2 = 50$, $cdn_{avg} = 400/5 = 80$, $minIdx_2$ = 80 * 2 + 1 = 161.

Although the parameter in above example is of integer type, our approach is generic. For all numeric parameters corresponding to non-key columns, the representation of S-Dist and

the parameter generation are exactly the same. For parameters on key columns, there are small differences. Since the primary key in our synthetic database is generated sequentially, the domains of key columns in synthetic database may be different from that in real database. Therefore, when collecting statistics for S-Dist, we use the domains of key columns in the real database to divide the intervals and construct HS. But during the synthetic workload generation, we use the domains of key columns in the synthetic database to support parameter generation. For string typed parameters, the biggest difference is how we divide the intervals. When constructing HS, for a string typed parameter, the interval which it belongs to is calculated by $h\%I$, where h is the hash code of the parameter. The parameter generation is similar to numeric types, and noted that the value transformer for parameter generation is consistent with that for synthetic database generation.

B. Dynamic Data Access Distribution

S-Dist can well depict the skewness of data access distribution throughout the workload cycle. However, if the data access distribution is dynamically changing, S-Dist is inaccurate or even completely wrong. Let's take a simple example. Suppose there is a table with 10^3 records and a workload with a period of 10^3 seconds. In the ith second, all database requests of the workload only access the ith record in the table. Assume that the throughput of the database is stable during this period. At this time, if S-Dist is used to express the entire data access process, we actually find that there is no hot data and the data access distribution is very uniform. Obviously, this is quite different from the fact. For the synthetic workload generated according to this S-Dist, the transaction conflict intensity on database must be much lower than that of the real workload. So in this section, we propose D-Dist to characterize both dynamics and skewness of data access distribution.

In order to catch such kind of dynamics, we divide the workload trace of a parameter into multiple equal-length time windows according to the log timestamp. For the parameter trace in any time window, we generate its individual S-Dist, and the D-Dist for entire parameter traces is defined as a list of S-Dists. Assuming that the entire workload cycle is one day and the time window size is one second, D-Dist of the parameter is constructed by 24*3600=86400 S-Dists. During the generation of synthetic workload, we instantiate symbolic parameters using the S-Dist corresponding to the generation time. In addition, for numeric parameters, the used data range in a time window may be much smaller than column domain. In order to improve the accuracy of HS, the intervals can be divided according to the data range of the current window when collecting the statistics. Certainly, it is also necessary to use the corresponding index range for each interval when generating the parameters.

C. Continuous Data Access Distribution

In some applications, the hot degree of the data is closely related to time, and the specific manifestation is that data may

The trace of a parameter:

740, 895, 482, ..., 463, 639, 611, 482, 366, ..., 573, 246, ...

←— time window 1 —→ ←— time window 2 —→ ...

High frequency items

item	frequency
514	0.08
482	0.12
...	...
296	0.09

repetition rate: 0.6

Histogram statistics

window domain: [200, 700]

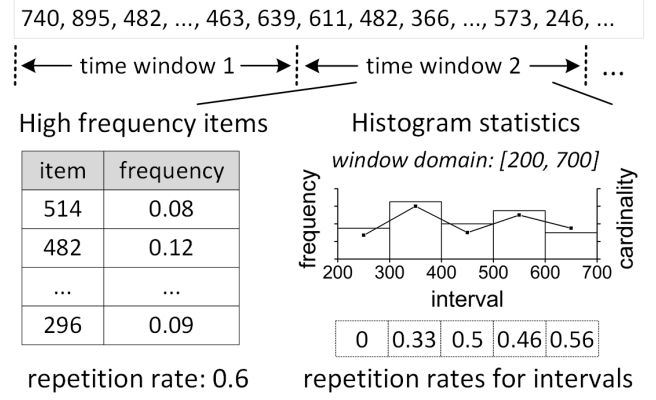| 0 | 0.33 | 0.5 | 0.46 | 0.56 |

repetition rates for intervals

Fig. 6. An example for illustrating C-Dist

be accessed continually for a period of time. We call this the continuity of data access distribution. For example, for online food delivery applications, steamed buns are generally ordered frequently in the morning, and coffees may be the favorite one during the afternoon. Previously, D-Dist is defined to catch the skewness of data access in time windows, while ignoring the continuity of data access between successive time windows. When using it to generate the synthetic workload, the data accessed between successive time windows may be completely different, which results in a lower cache hit ratio. So we present C-Dist to characterize the continuity, dynamics, and skewness of data access distribution.

We introduce the concept of repetition rate for C-Dist to describe the continuity of data access based on D-Dist. When collecting the statistics, we count the repetition rate between the high frequency items in current time window and that in previous time window, as well as the repetition rates of parameters for all intervals. Fig. 6 continues the example in Fig. 4 by adding repetition rates for both HFI and HS. In this example, we can see that the repetition rate for HFI is 0.6, that is, there are three high frequency items are kept from the previous time window. The repetition rates for five intervals in HS are: 0, 0.33, 0.5, 0.46 and 0.56. Assuming that cdn_1 is 15, then there are $15*0.33 \approx 5$ parameters in interval 1 that have appeared in the previous time window.

In order to ensure the repetition rates in C-Dist, we need to pregenerate the candidate parameters for each time window. In Alg. 1, we list the detailed generation process of candidate parameters. In lines 1-2, we generate all the high frequency items that satisfy the expected repetition rate. In lines 3-10, we traverse all the parameters in the previous time window and select the repeated parameters for each interval until the repetition rate on the interval is met. In the 6th line, we deduce the index of the parameter according to its value, so as to identify the interval number which it belongs to in the current time window. If the index is not in the index domain of the current time window, the parameter is ignored. And for string typed parameters, e.g., '296#$dgtckuy$', the front part of the

Algorithm 1 Candidate parameter generation

Input: High frequency items Φ^{t-1} and candidate parameters for intervals Ω^{t-1} of the previous time window, repetition rate κ for HFI, repetition rates Ψ for HS, cardinalities $\mathcal{C}$ in HS for intervals, number of high frequency items H, number of intervals I

Output: High frequency items Φ^t and candidate parameters for intervals Ω^t of the current time window

1: Randomly select $H * \kappa$ items from Φ^{t-1} and put them in Φ^t.
2: Randomly generate $H * (1 - \kappa)$ items with the corresponding column generator and put them in Φ^t.
3: Initialize an array $\Gamma \leftarrow 0...0$. The length of Γ is I.
4: Initialize a list $P \leftarrow$ all parameters in Ω^{t-1}.
5: **for all** parameter p in P **do**
6:     Identify the interval number i of parameter p.
7:     **if** $\Gamma[i] < \mathcal{C}[i] * \Psi[i]$ **then**
8:         Put the parameter p in $\Omega^t[i]$, $\Gamma[i] + +$
9:     **end if**
10: **end for**
11: **for all** interval $i = 0$ to $(I - 1)$ **do**
12:     Randomly generate $(\mathcal{C}[i] - \Gamma[i])$ parameters belonging to the interval i using the column generator and put them in $\Omega^t[i]$.
13: **end for**
14: **return** Φ^t and Ω^t

---

value split by '#' character is the index we need. Finally, in lines 11-13, we generate random parameters added to each interval to reach the cardinality requirement. Based on these candidate parameters, our parameter generation mechanism is still consistent with Fig. 5. And now for a certain interval, we only need to randomly select a candidate parameter as the output. Last but not least, if the candidate parameters are generated online during the synthetic workload generation, the workload generator, i.e., Lauca, may become the performance bottleneck, thereby affecting the correctness of the evaluation result. Therefore, we can generate the candidate parameters for all time windows offline and store them on disk, and then read them as needed when generating the synthetic workload.

VI. WORKLOAD GENERATION

Given the transaction logic of each transaction template and the data access distribution of each parameter, the Workload Generator in Fig. 1 is responsible for generating synthetic workloads satisfying the desired configurations. Meanwhile, efficiently generating high concurrency/throughput synthetic workloads in a distributed environment is also an essential requirement for our workload generator. Now we will present the details of workload generation from three levels: thread model, transaction execution and parameter instantiation.

Thread Model. Users can configure multiple test nodes for deploying the workload generator and the number of test threads on each node, to simulate the concurrency. For each test thread, we establish a separate database connection. Lauca supports two different execution models for test threads to invoke transactions: *no-await-in-loop* and *fixed-throughput*. With the no-await-in-loop setting, all test threads repeatedly issue transactions without any think time between requests. In fixed-throughput setting, the user can specify a fixed request throughput or a throughput scale factor. If the throughput

scale factor is specified, we multiply the scale factor and the throughput of each time window obtained from the workload trace to get the target throughput. The test threads achieve the desired throughput by controlling the think time between transaction requests. When the required throughput exceeds the maximum throughput that the current test threads can achieve, the execution model degenerates to no-await-in-loop. Different execution models enable us to build extended synthetic workloads.

Transaction Execution. The test thread invokes different kinds of transactions according to their proportions extracted from workload traces. And the transaction proportions are adjusted periodically with the time window. During the execution of a transaction, the structure information of its transaction logic will be used to determine whether the operations in the branch structure need to be executed, and the number of executions of operations in the loop structure. For the execution of a specific SQL operation, we first instantiate all the symbolic parameters one by one, and then send the operation with concrete parameter values to the test database. After the operation is executed, the result set and the parameters are maintained in an intermediate state for the generation of other parameters in subsequent operations within the same transaction instance.

Parameter Instantiation. When instantiating the parameters, we first guarantee the consistency of transaction logic, and then ensure the data access distribution of synthetic workloads. For a parameter, *Case 1: if there is only dependency $1,* the value of this parameter can be calculated directly based on the increment Δ and the associated smaller parameter. *Case 2: if there are only dependencies $2,* we first attempt to instantiate the parameter by randomly selecting a dependency according to their probabilities. When no dependencies are selected, then we use the data access distribution to instantiate this parameter. *Case 3: if there are both dependencies $2 and $3,* the corresponding operation must be in the loop structure. For the first loop execution, we still instantiate the parameter based on dependencies $2 and data access distribution as Case 2. For non-first loop executions, we first attempt to instantiate the parameter using dependencies $3 based on the probability. If no dependencies $3 are selected, dependencies $2 and data access distribution are then used to instantiate the parameter.

Overall, transaction execution and parameter instantiation are independent of each other for all test threads, so our workload generator can be deployed on multiple nodes to efficiently generate the high concurrency/throughput synthetic workloads while satisfying the desired workload characteristics and configurations.

VII. DISCUSSION AND ANALYSIS

Sensitivity to Concurrency Control Mechanism. Concurrency control ensures DB consistency when transactions execute concurrently. Different Isolation Levels resolve different types of conflicts by making trade-offs between performance and data integrity, as shown in Table II.

Isolation level(IL)		Workload characteristics	
		Single-version	Multi-version
RU	Read uncommitted	W-W	
RC	Read committed	W-W, short R-W	W-W
RR	Repeatable read	W-W, long R-W	
SI	Snapshot isolation	/	W-W
SR	Serializability	W-W, long R-W, range R-W	

W-W conflicts must be resolved at any isolation level. For *RC*, in a single-version mode, it may meet the short R-W conflicts which can be avoid in multi-version mode by using previous version of data. For *RR*, data read by a transaction cannot be modified until the end of the transaction, and it is prone to have long R-W conflicts. *SI* is implemented in a multi-version way. It can avoid R-W conflicts by using two timestamps for a transaction. *SR* checks all possible conflicts between range reads and other writes to avoid phantom, which deteriorates performance greatly if workloads contain lots of range R-W conflicts. Isolation levels are closely related to conflicts, while pessimistic or optimistic implementation of concurrency control will only affect the cost of handling conflicts. Lauca defines transaction logic and extracts data access distribution to catch workload conflicts from the real workloads. So Lauca is applicable for DB evaluations under different concurrency control protocols and isolation levels.

Data Protection. Since we isolate **Production Environment** from **Generation (Evaluation) Environment**, testers cannot touch real data and workload traces. Only data statistics and workload description information are exposed. For further schema protection, we allow to anonymize names for tables, columns and transaction templates; we can insert noisy columns in tables, such as *bool, int* or *char* typed columns which will not deteriorate I/O much; column domain can be changed. For trace protection, we use only the parameter dependencies inside transaction templates. But at present, we cannot guarantee to conceal all sensitive information if reverse engineering is launched on our generated data.

Limitations of Transaction Logic. Transaction logic in Definition 2 aims to support the abstration to common workloads in real world applications. There are still deficiencies in our current work. Firstly, the defined parameter dependencies are not complete, which can not represent complex dependencies, e.g., a quadratic function. Secondly, only the relationship between two data items is considered at present, without considering the relationship among more data items. Thirdly, the importance of different relationships may vary greatly, which have been quantified. These issues are left to be addressed in future work.

Choosing the Window Size and Number of Intervals. The setting of the time window size depends on how often the target workload changes. If workloads change frequently, the window size should be set to a small value, otherwise to a larger value. We recommend setting the time window size to 1 second so that even second-level workload changes

can be captured. Histogram is widely used to represent data distribution statistics in industry databases, such as Oracle, MySQL and PostgreSQL.

VIII. EXPERIMENTS

Environment. Our experiments are conducted on four servers, and each server is equipped with 2 Intel Xeon Silver 4110 @ 2.1 GHz CPUs, 120 GB memory, 4 TB HDD disk configured in RAID-5 and 4 GB RAID cache. The servers are connected using 10 Gigabit Ethernet. The test platforms selected in our experiments are the most popular and advanced open source DBMSs: MySQL (v5.7.24) and PostgreSQL (v10.4), which are also widely used in industry.

Workloads. Two standard benchmarks are used throughout the experiments: TPC-C [9] and SmallBank [10]. TPC-C is one of the most widely used industrial-level online transaction processing benchmark. It involves nine tables and five types of transactions that simulate the activities found in complex OLTP applications. SmallBank abstracts the operations in banking applications, where the schema consists of three tables and the workload includes six types of transactions. All transactions of SmallBank perform simple read and update operations on a small number of tuples. We use OLTP-Bench [8], an extensible DBMS benchmarking testbed, to generate workloads of TPC-C and SmallBank, which act as real workloads for comparison. Workload traces are logged by OLTP-Bench and serve as input to Lauca for generating the synthetic workload. By the way, the original TPC-C implementation of OLTP-Bench artificially binds the test threads to the warehouses, which greatly reduces the conflict of generated workloads on the database. This additional binging is removed in our experiments to make the workload more practical. Yahoo! Cloud Serving Benchmark (YCSB) [11] is a collection of workloads that represent the large-scale web applications. We construct micro benchmark workloads based on YCSB, for simulating workloads with fine-grained control on skewness, dynamics and continuity.

Settings. In the transaction logic extraction, the number K of transaction instances and the groups number N of transaction instances are all set to 10^4. For the data access distribution extraction, the number H of items in HFI and the number I of intervals in HS are all set to 50, and the time window size is set to 1 second.

A. Fidelity of Synthetic Workloads

The similarity between synthetic workloads and real application workloads is called workload *fidelity*. Fidelity of synthetic workloads is the most essential target when designing Lauca. It is measured by performance deviations obtained by running synthetic workload and real application workload on the same database system.

Figure 7-10 show the deviations in performance between synthetic workloads generated by Lauca and real workloads generated by OLTP-Bench, under different scale factors (SF). The concurrency of database requests is the same as the scale factor. We execute TPC-C and SmallBank workloads
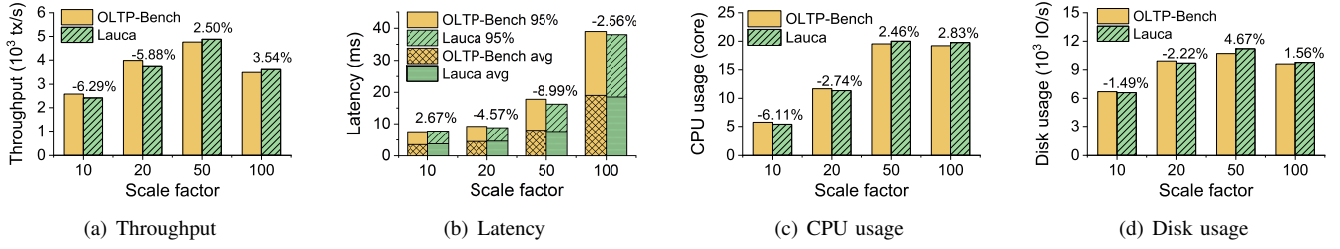
(a) Throughput   (b) Latency   (c) CPU usage   (d) Disk usage

Fig. 7. Deviations in performance metrics for TPC-C workloads on PostgreSQL database

(a) Throughput   (b) Latency   (c) CPU usage   (d) Disk usage
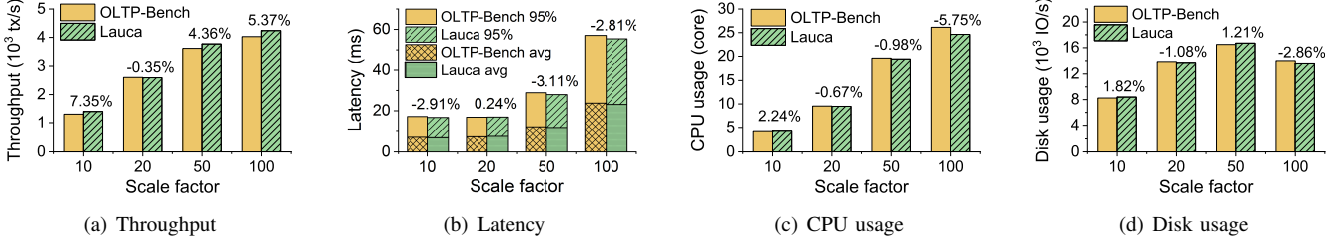
Fig. 8. Deviations in performance metrics for TPC-C workloads on MySQL database

(a) Throughput   (b) Latency   (c) CPU usage   (d) Disk usage
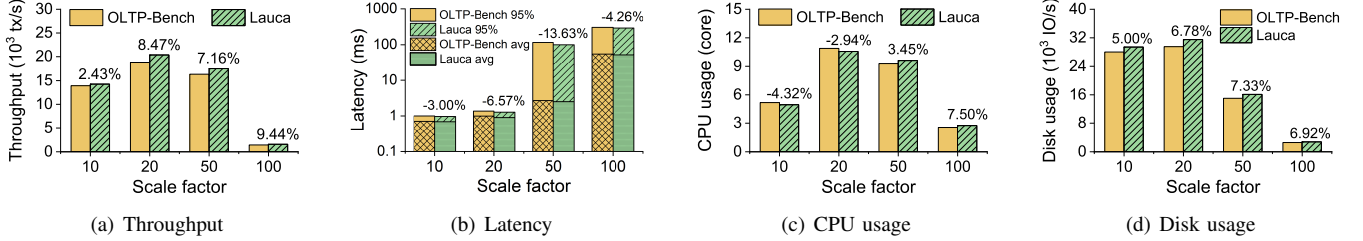
Fig. 9. Deviations in performance metrics for SmallBank workloads on PostgreSQL database

(a) Throughput   (b) Latency   (c) CPU usage   (d) Disk usage
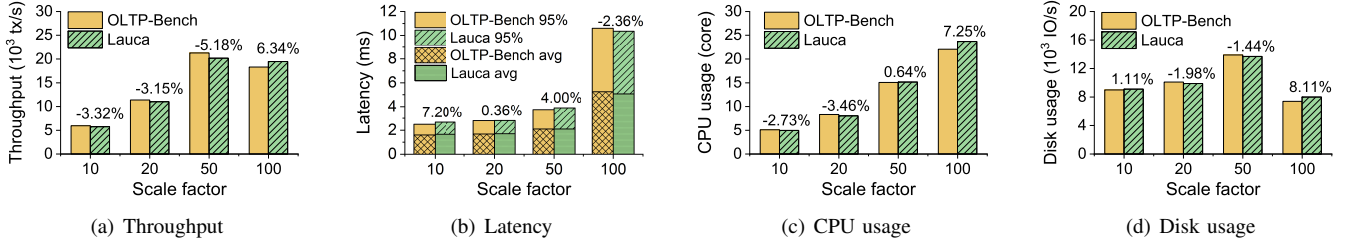
Fig. 10. Deviations in performance metrics for SmallBank workloads on MySQL database

on PostgreSQL and SmallBank respectively. In Figure 7(a), 8(a), 9(a) and 10(a), we present execution throughputs of real workloads and synthetic ones. From the results we can see that these two workloads are very similar in throughputs, among which the maximum deviation is as low as 9.44% in Figure 9(a). For average latency and 95% latency, synthetic workloads are very close to real workloads in Figure 7(b), 8(b), 9(b) and 10(b), with the maximum deviation among them is only 8.99% in Figure 7(b). In Figure 7(c)-7(d), 8(c)-8(d), 9(c)-9(d), and 10(c)-10(d), we report the CPU and Disk usages of these two workloads. The results show that the resource consumptions for executing synthetic workloads and real workloads on both PostgreSQL and MySQL databases are consistent, which further verify the high fidelity of synthetic workloads generated by Lauca.

B. Exploring Transaction Logic

Simulation of transaction logic of application workloads is an important feature of Lauca. In this section, we demonstrate

the impact of transaction logic on transaction semantics, transaction conflict intensity, deadlock possibility and distributed transaction ratio of synthetic workloads. All experiments in this section are run on MySQL database, with the workloads taken from TPC-C benchmark. Both the scale factor and the request concurrency are 20. Though transaction logic consists of structure information and parameter dependency information, in which structure information, such as the number of loop executions, has an obvious impact on performance, so here we mainly explore the impact of parameter dependency information on database performance.

Using the TPC-C workload including all five types of transactions, we study how the transaction logic affects the transaction semantics of synthetic workloads. Fig. 11 shows the throughputs of real workloads and synthetic workloads respectively generated by OLTP-Bench and Lauca. The transaction throughput is divided into two parts which are *success throughput* and *failure throughput*. Success throughput refers to the throughput of successfully executed transactions, while
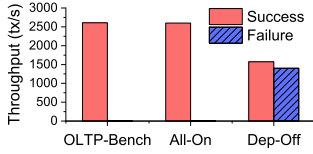
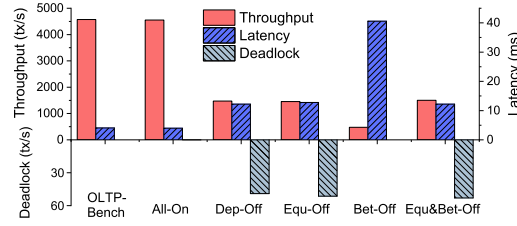Fig. 11. The impact on transaction semantics

Fig. 12. The impact on transaction conflict intensity, deadlock possibility and scan data volume
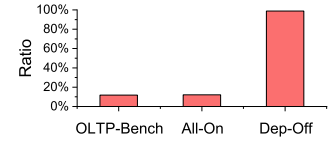

Fig. 13. The impact on distributed transaction ratio

failure throughput refers to the throughput of failed transactions. In Fig. 11, when we use all the information in transaction logic (All-On), Lauca presents excellent performance similarity to the real workload; when we turn off the parameter dependency information (Dep-Off), transaction failures increase sharply and the success throughput is much smaller than that of OLTP-Bench. This is because the insert operations in NewOrder transactions do not satisfy the primary/foreign constraints, leading to a larger number of transaction rollbacks. And because no new orders are generated, Delivery transactions cannot be successfully executed too. Overall, the transaction logic can effectively ensure that the transaction semantics of the synthetic workloads are consistent with the real workloads.

To further explore the impact of transaction logic on other aspects, in Fig. 12, we only retain the three types of transactions that can be successfully executed, namely Payment, OrderStatus, StockLevel in TPC-C workloads. We present the transaction throughputs, latencies and deadlock throughputs of workloads generated by OLTP-Bench and Lauca in Fig. 12. There are five groups of experimental results for Lauca, which are obtained by turning off different parts of transaction logic. Note that *latency* here is the average latency of successfully executed transactions, not including the failed ones. From the results, we can see that when we turn off the equal parameter dependency (Equ-Off), the throughput decreases significantly, the latency increases markedly, and a lot of deadlocks occur. Increased latency indicates that there is more lock waiting time and the transaction conflict is more intensive. Occurrence of these phenomena is because there are three pairs of read and write operations involving the same record in Payment transactions. When we turn off the equal parameter dependency, read and write operations on the same record are likely to become on different records, which greatly increase the likelihood of transaction conflicts and deadlocks, especially when these records are in a small table. When we turn off the between parameter dependency (Bet-Off), the latency increases drastically, and the throughput is very low. This is because the scan operation in StockLevel transactions will involve a large amount of data, and normally only about 20 records should be accessed. When we turn off both the equal and between dependencies, the scan operation does not read a large amount of data due to the role of C-Dist, so that the performance metrics of Dep-Off, Equ&Bet-Off and Equ-Off are similar. Overall, the results confirm that manipulating

transaction logic enables synthetic workloads with the same transaction conflict intensity, deadlock possibility and scan data volume as real workloads.

The distributed transaction ratio has a significant impact on database performance [7], [12], [13]. In order to accurately obtain the distributed transaction ratios of evaluation workloads, we assume that the data is hash-partitioned by Warehouse ID on five nodes, and count the distributed transaction ratios of workloads from the application site. In Fig. 13, it shows the distributed transaction ratios of workloads generated by OLTP-Bench and Lauca. Since OrderStatus and StockLevel are read-only transactions, the workloads used in Fig. 13 are only Payment transactions. As can be seen from the results, when we turn off the parameter dependency information (Dep-Off), the distribution transaction ratio increases dramatically. This is because at this point the Warehouse ID parameters of four write operations in the Payment transaction are randomly generated, thus there is a high probability of becoming distributed transactions. Overall, the control of transaction logic can ensure that the synthetic workload and the real workload have the same distributed transaction ratio in a distributed environment.

C. Exploring Data Access Distribution

This section demonstrates the ability of proposed data access distributions, namely S-Dist, D-Dist and C-Dist, to depict the skewness, dynamics and continuity of data access. Since the data access distribution of existing benchmark workloads is generally neither dynamic nor continuous, we build the evaluation workloads based on YCSB. All experiments in this section are carried out on MySQL database with a test table from YCSB. The size of test table is 10^6, and the concurrency of database request is 20.

The evaluation workload in Fig. 14 has only one type of transaction. The transaction consists of five pairs of read-write operations, each of which reads a record first and then updates it. The extended YCSB workload runs for 90 seconds, which is divided into three phases, and the data requests of each phase are within 10^3 records randomly selected. In the first phase, the data access distribution is Zipf distribution with parameter $s = 1$; the second phase is still the Zipf distribution, but the parameter $s = 1.2$; and the third phase is the uniform distribution. Fig. 14(a)-14(c) show the dynamic changes of transaction throughput, latency and deadlock throughput for workloads generated by YCSB and Lauca, respectively. Lauca
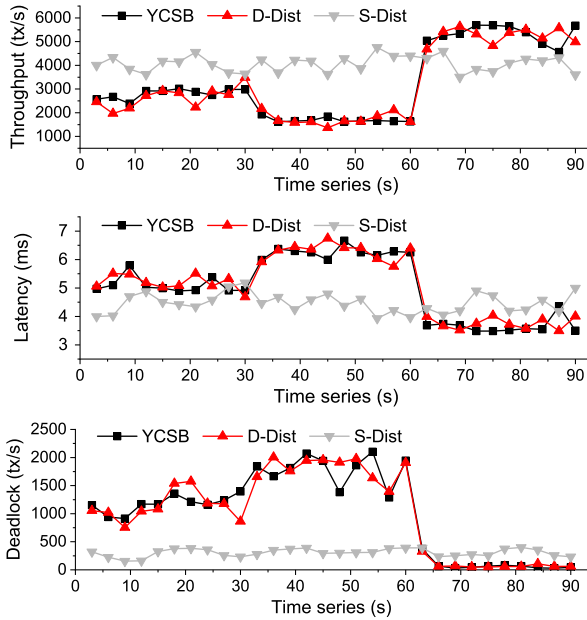
Fig. 14. Exploring S-Dist and D-Dist for skewed and dynamic workloads

has two groups of results, corresponding to S-Dist and D-Dist. It can be seen from the results that when using D-Dist, the synthetic workload generated by Lauca is dynamically consistent with the real workload generated by YCSB on throughput, latency and deadlock, indicating that D-Dist can well depict the dynamics of workloads. Meanwhile D-Dist is represented by S-Dist in each time window, which also shows that S-Dist can well characterize the skewness of workloads. But the global S-Dist is not working well (grey lines in Fig. 14), which is defined for the whole workload time and does not take into account the dynamic changes of the workload.
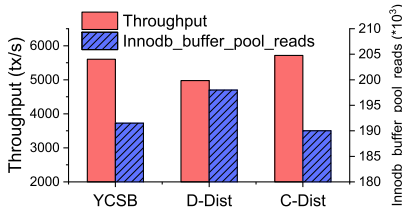

Fig. 15. Exploring C-Dist for continuous workloads

The evaluation workload in Fig. 15 is the single-row update transaction of YCSB, by running 100 seconds with 1 second a time window. The data requests in each time window is based on 10^3 random records, and the selected records for each time window are 50% coincident with the previous window. The *Innodb_buffer_pool_size* of MySQL is set to 16 MB. Fig. 15 presents the throughputs and *Innodb_buffer_pool_reads increments* for workloads generated by YCSB and Lauca. *Innodb_buffer_pool_reads* is the number of logical reads that InnoDB can not satisfy from the buffer pool, and have to read directly from disk. From the results, we can see that the disk access of D-Dist is significantly higher than that of YCSB, and its throughput is lower. This is because D-Dist is unable to catch the continuity of data access distribution, resulting

(a) on-CPU OLTP-Bench   (b) on-CPU Lauca

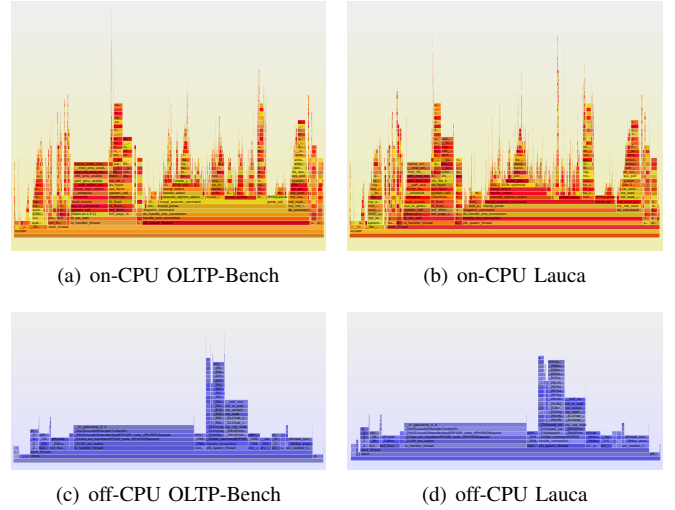
(c) off-CPU OLTP-Bench   (d) off-CPU Lauca

Fig. 16. Flame graph

in data requests in each time window are almost completely different with a low cache hit ratio. The performance of C-Dist is consistent with YCSB, which indicates that C-Dist can well characterize the data access continuity.

D. Exposing Micro Actions in Database

We look into DB performance by on-CPU and off-CPU flame graphs in Figure 16 to show the consumption of CPU cycles for MySQL codepaths based on the complicated TPC-C workloads with each rectangle represents a stack frame. Stack profile population is aligned along x-axis, sorted alphabetically, and stack depth is aligned along y-axis, representing the calling structure bottom-up. The width of a flame in on-CPU flame graph or off-CPU flame graph reprensents its occupation frequency of pushing into stacks. On-CPU graph presents the threads spending time running on-CPU while off-CPU demonstrates the threads waiting time for CPU, which may be blocked by I/O, locks, timers, etc.. Merging of on-CPU and off-CPU flame graphs is the complete statues of thread executing on CPU for MySQL codepaths. By comparing the flame graphs of real workloads from OLTP-Bench in Figure 16(a) and 16(c) and the one generated by Lauca in Figure 16(b) and 16(d), we find that we have similar flame graphs between the real one and the generated one, sampling the execution traces for 20 seconds. The micro execution status confirms that Lauca has perfect simulation ability.

E. Performance of Lauca

In this section, we use TPC-C workload traces to study the performance of components in Lauca.

Fig.17 shows the execution time and memory consumptions of transaction logic extraction under different sizes of K/N. It can be seen from the results that when both K and N are 10^4, the extraction time of transaction logic is only 2.1 seconds and the memory consumption is 1.1 GB. With the increase of K and N, the execution time and memory consumption increase almost linearly. The experiments in Section VIII-A are all conducted when K and N are set to 10^4, and the
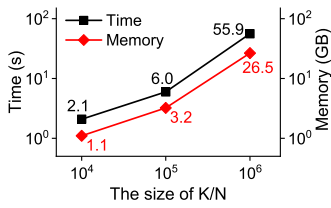
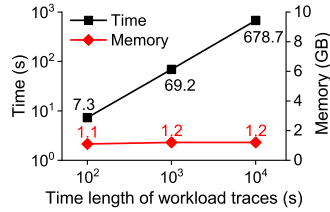Fig. 17. Performance of transaction logic extraction ($K = N$)

Fig. 18. Performance of C-Dist extraction

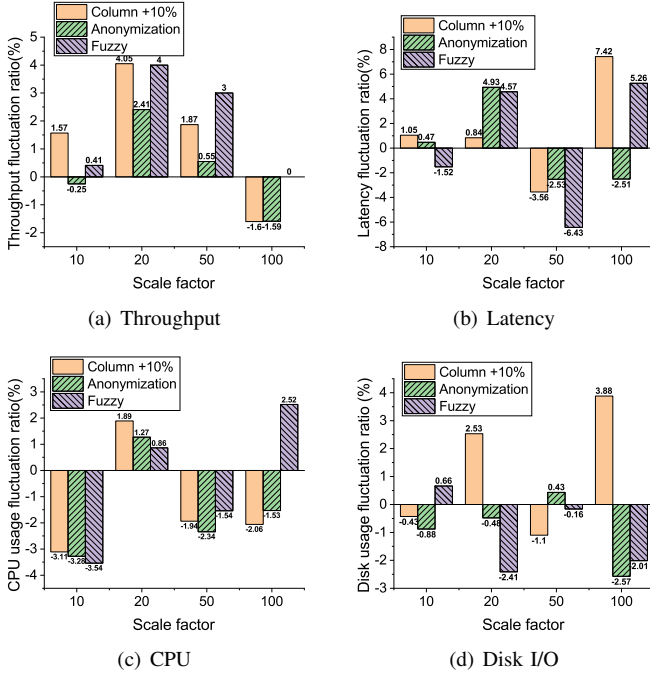(a) Throughput

(b) Latency

(c) CPU

(d) Disk I/O

Fig. 19. Performance fluctuation under data protection

high fidelity of generated workloads has been proved. Overall, the transaction logic extraction in Lauca is efficient and can be done in a few seconds while guaranteeing the fidelity of synthetic workloads.

Fig.18 presents the execution time and memory consumptions of C-Dist extraction for workload traces with different time length. From the results, we can see that the extraction time of C-Dist is linear with the volume of workload traces while the memory consumption is constant. This is because C-Dist is a window-based data access distribution, and the workload trace of each time window can be removed from memory after processing. In Fig.18, for TPC-C workload traces with the time length of 10^4 seconds (transaction throughput is 3610.3 and log volume is 33.8 GB), the extraction time of C-Dist is 678.7s and the memory consumption is 1.2 GB. Since the maximum workload cycle in practical evaluations is generally one day, the result indicates that Lauca can effectively support the performance evaluation of high throughput workloads.

F. Exhibiting Data Protection

Lauca has been proved to have perfect simulation ability in Figure 7 - 10 with performance deviation less than 10%. In order to further cover sensitive information, we have

additional source data protection ways including inserting noisy data, anonymization and fuzzing domain size. Here we take Lauca as a baseline, which are integrated with those additional protection methods. In Figure 19, we show their performance fluctuation ratios in $Throughput$, $Latency$, CPU, and $DISK\ I/O$ to compare their simulation capabilities on TPC-C workloads by changing SF from $10\ to\ 100$. We insert 10% more $integer$ typed columns for tables to cover table structures, i.e., $Column + 10\%$; we anonymize table/column names to conceal the semantics, i.e., $Anonymization$; we expand domain size randomly by 8%-12% to shield domain information, i.e., $Fuzzy$. For $throughput$, data protection causes at most 4.05% performance deviation from Lauca, generated by $Column + 10\%$; for $latency$, the most serious deviation is 7.42% still caused by $Column + 10\%$; For CPU and $Disk\ I/O$, their performance deviations from Lauca are no more than 4%. Among the three methods, inserting new columns (noisy data) has more influence to our simulation results, since it may have to access additional more data. But we can insert smaller $bool$ or $char$ typed columns into tables instead of $integer$ typed ones, which can reduce the accessed volume of noisy data. Although all the methods may have negative effect to the simulation results, the deviation from Lauca is limited by less than 8%, if rigorous data protection is required.

IX. RELATED WORK

There are a number of benchmarks for database performance evaluation in different application areas. For OLAP applications, TPC-H, TPC-DS, and SSB [14] are the frequently-used benchmarks with defined standard database schemas and test queries. And there are TPC-C, TPC-E and SmallBank [10] benchmarks for evaluating the transaction processing capability of database systems. In addition, CH-benCHmark [15] and HTAPBench [16] can provide a unified assessment for hybrid transaction/analytical processing (HTAP) systems. Moreover, YCSB [11] is usually used to measure the throughput of cloud service systems, whose workload is simple but requires high scalability. However, the evaluation workloads of these standard benchmarks are abstractions for a class of applications, so they are too general to evaluate database performance for a specific application.

In order to obtain elaborate workloads of a target application, workload trace replaying is an optional method. Microsoft SQL Server equips two tools, i.e., SQL Server Profiler [17] and SQL Server Distributed Replay [2], for reproducing production workloads based on the SQL traces. Oracle Database Replay [3], [18] enables users to record workload traces on the production system with minimal performance impact and then to replay a full production workload that has the same concurrency and workload characteristics as the real one. Due to the data privacy issues, workload replay is difficult to apply in the practical database performance evaluation because it requires a real database state and original workload traces [3]. Additionally, workload extension (e.g.,

extending concurrency) is also a problem that is difficult to solve with current replay technologies.

Then workload simulation is necessary and urgent. There are workload-aware data and query generators [4], [5], [19], [20] for database performance evaluation of OLAP applications. The input of these works generally includes database schema, basic data characteristics and size specifications for intermediate results of query trees. The output is a synthetic database instance and instantiated test queries, conforming to the specified data and workload characteristics. And for OLAP applications, there are database scaling works [21], [22] which can scale up/down a given database instance, supporting application-specific database benchmarking. Workload analyzers [23], [24] are designed to study and better understand the application workloads, but neither of which can generate synthetic workloads. There are workload generators [25], [26] for database performance benchmarking. Jeong et al. [25] proposes a workload generator for simulating a realistic hardware resource consumption status. NoWog [26] introduces a workload description language for generating synthetic workloads benchmarking NoSQL databases. None of these works [25], [26] can be used to simulate the various workloads of actual OLTP applications for application-oriented database performance evaluation.

X. Conclusion

In this paper, we presented Lauca, a transactional workload generator for application-oriented database performance evaluation. Lauca uses transaction logic to depict the potential business logic of target applications, and data access distribution to characterize the access skewness, dynamics and continuity. Our results on various workloads and popular databases show that Lauca consistently generates high-quality synthetic workloads. Future work focuses on improving the definition of transaction logic and proposing new types of data access distribution for covering more application workloads.

References

[1] M. I. Seltzer, D. Krinsky, K. A. Smith, and X. Zhang, "The case for application-specific benchmarking," in HotOS, 1999, pp. 102–109.

[2] SQL Server Distributed Replay, https://docs.microsoft.com/en-us/sql/tools/distributed-replay/sql-server-distributed-replay?view=sql-server-2017.

[3] L. Galanis, S. Buranawatanachoke, R. Colle, B. Dageville, K. Dias, J. Klein, S. Papadomanolakis, L. L. Tan, V. Venkataramani, Y. Wang, et al., "Oracle database replay," in SIGMOD, 2008, pp. 1159–1170.

[4] E. Lo, N. Cheng, W. W. K. Lin, W. Hon, and B. Choi, "Mybenchmark: generating databases for query workloads," in VLDBJ, 2014, pp. 895–913.

[5] Y. Li, R. Zhang, X. Yang, Z. Zhang, and A. Zhou, "Touchstone: Generating enormous query-aware test databases," in USENIX ATC, 2018, pp. 575–586.

[6] A. J. Bonner and M. Kifer, "A logic for programming database transactions," in Logics for Databases and Information Systems, 1998, pp. 117–166.

[7] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K. Tan, and Z. Wang, "Towards a non-2pc transaction management in distributed database systems," in SIGMOD, 2006, pp. 1659–1674.

[8] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," in PVLDB, 2013, pp. 277–288.

[9] TPC-C benchmark, http://www.tpc.org/tpcc/.

[10] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The cost of serializability on platforms that use snapshot isolation," in ICDE, 2008, pp. 576–585.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in SoCC, 2010, pp. 143–154.

[12] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," in PVLDB, 2017, pp. 553–564.

[13] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," in PVLDB, 2010, pp. 48–57.

[14] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," in TPCTC, 2009, pp. 237–252.

[15] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. Kuno, R. Nambiar, T. Neumann, M. Poess, et al., "The mixed workload ch-benchmark," in DBTest, 2011, pp. 8.

[16] F. Coelho, J. Paulo, R. Vilaça, J. Pereira, and R. Oliveira, "Htapbench: Hybrid transactional and analytical processing benchmark," in ICPE, 2017, pp. 293–304.

[17] SQL Server Profiler, https://docs.microsoft.com/en-us/sql/tools/sql-server-profiler/sql-server-profiler?view=sql-server-2017.

[18] Y. Wang, S. Buranawatanachoke, R. Colle, K. Dias, L. Galanis, S. Papadomanolakis, and U. Shaft, "Real application testing with database replay," in DBTest, 2009, pp. 8.

[19] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "Qagen: generating query-aware test databases," in SIGMOD, 2007, pp. 341–352.

[20] A. Arasu, R. Kaushik, and J. Li, "Data generation using declarative constraints," in SIGMOD, 2011, pp. 685–696.

[21] Y. Tay, B. T. Dai, D. T. Wang, E. Y. Sun, Y. Lin, and Y. Lin, "Upsizer: Synthetically scaling an empirical relational database," in Information Systems, 2013, pp. 1168–1183.

[22] J. Zhang and Y. Tay, "Dscaler: Synthetically scaling a given relational database," in PVLDB, 2016, pp. 1671–1682.

[23] P. S. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee, "On workload characterization of relational database environments," in IEEE Transactions on Software Engineering, 1992, pp. 347–355.

[24] Q. T. Tran, K. Morfonios, and N. Polyzotis, "Oracle workload intelligence," in SIGMOD, 2015, pp. 1669–1681.

[25] H. J. Jeong and S. H. Lee, "A workload generator for database system benchmarks," in iiWAS, 2005, pp. 813–822.

[26] P. Ameri, N. Schlitter, J. Meyer, and A. Streit, "Nowog: a workload generator for database performance benchmarking," in DASC/PiCom/DataCom/CyberSciTech, 2016, pp. 666–673.