Lauca: Generating Application-Oriented Synthetic Workloads

Yuming Li[1], Rong Zhang[1], Yuchen Li[2], Ke Shu[3], Shuyan Zhang[1], Aoying Zhou[1]
[1]East China Normal University, [2]Singapore Management University, [3]PingCAP Ltd.
{liyuming@stu, rzhang@dase, syzhang@stu, ayzhou@dase}.ecnu.edu.cn, yuchenli@smu.edu.sg, shuke@pingcap.com

***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, database performance may not be as good as the expectation for online workloads; but for database vendors, developers cannot 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, which are launched by using real online workloads, 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 do performance evaluation. 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 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 the 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.

The overall architecture and workflow of Lauca are presented in Fig. 1. We give an overview of Lauca from two lines which are synthetic database generation and synthetic workload generation. The inputs of *Database Generator* are database schema and data characteristics. Since the data characteristics are tedious and need to be obtained from the real

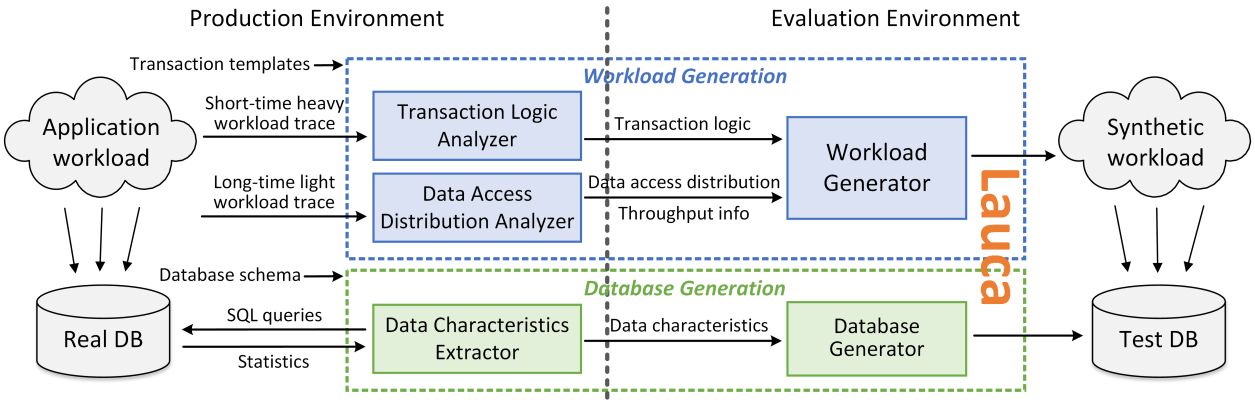


Fig. 1. The overall architecture and workflow of Lauca

database, we provide a *Data Characteristics Extractor* which can help us get this information automatically by using simple SQL queries. The inputs of *Workload Generator* are transaction templates and workload statistics, where transaction templates are the skeleton diagrams of target application's transactions and workload statistics mainly include transaction logic and data access distribution. By analyzing workload traces, we can extract the transaction logic and data access distribution for target application workloads. 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. 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 only executed in the production environment by data owners.

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 workloads should be highly similar to the real application workloads. ***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.

TX_1
Start Transaction
update S set $s_3 = s_3 + p_{1,1}$ where $s_1 = p_{1,2}$;
update T set $t_3 = t_3 + p_{2,1}$ where $t_1 = p_{2,2}$ and $t_2 = p_{2,3}$;
select z_2, z_3 from Z where $z_1 = p_{3,1}$;
update Z set $z_2 = z_2 - p_{4,1}$ where $z_1 = p_{4,2}$;
Commit

TX_2
Start Transaction
call storedProcedureName(in $p_{1,1}$, in $p_{1,2}$, out $r_{1,1}$);
Commit

TX_3
Start Transaction
IF { select z_3 from Z where z_1 between $p_{1,1}$ and $p_{1,2}$; }
ELSE { select z_3 from Z where $z_1 = p_{2,1}$; }
insert into S values ($p_{3,1}$, $p_{3,2}$, $p_{3,3}$);
LOOP { insert into T values ($p_{4,1}$, $p_{4,2}$, $p_{4,3}$); }
Commit

Fig. 2. Examples of transaction templates

Security. 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.

Extensibility. In some cases, the request concurrency and throughput of the current real application workload cannot meet the test requirements, so the real application workload needs to be able to be extended by our tool for generating synthetic workloads with desired scales.

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 the databases, but also to promise the properties of security, scalability and extensibility.

We propose the concept of transaction template to allow users to simply specify the transactions of target applications. It is difficult to extract accurate transaction definitions like the stored procedures from an expatiatory program, if stored procedures are not used for applications. Since a considerable proportion of transactions in practical applications are not executed as stored procedures [25], this paper proposes the

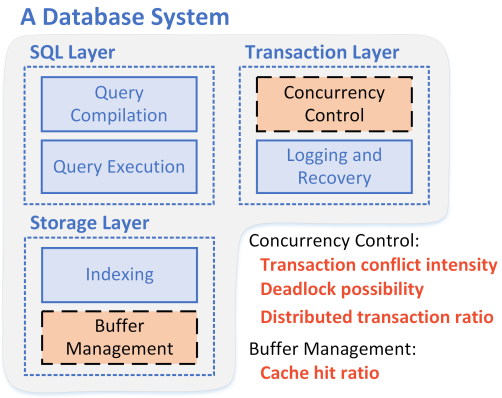


Fig. 3. Database-side workload characteristics that may be affected

transaction template to define the skeleton diagram of these transactions without specifying exact relationships among SQL operations. The transaction template is defined as follows.

Definition 2: **Transaction template:** It consists of SQL operations, branch structures (if any), and loop structures (if any), where the parameters in SQL operations are symbolized. All relationships among SQL operations, parameters and return items, as well as the judgment conditions in branch and loop structures are ignored.

Fig. 2 presents some sample transaction templates. If the transaction is executed as a stored procedure, the corresponding transaction template contains only one SQL operation, that is, a stored procedure call, as TX_2 in Fig. 2. In addition, the stored procedure needs to be created in the test database to support evaluation.

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. We generate the synthetic workload by instantiating the symbolic parameters in transaction templates.

A database system is mainly divided into three parts as in Fig. 3, 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 ratios. For a given database system, transaction log volume is the main

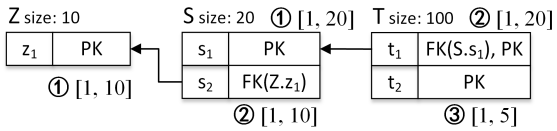


Fig. 4. Determining the domains of key columns

performance factor to *Logging and Recovery* module. And the log volume is determined by transaction templates and data characteristics, which are 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 settled. 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-side 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 workload traces. 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 traces to get data access distributions, 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.

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 do not 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. 4. 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. 4). Then, the

domains of foreign key columns can be determined by the domains of referenced primary key columns (as ② in Fig. 4). 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. 4). 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 the 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. Here, we first give an intuitive explanation of why transaction logic matters through a concrete example; then we give a formal definition of transaction logic; finally, we present how to efficiently extract transaction logic from workload traces.

A. Intuitive View

We use the sample transaction template TX_1 in Fig. 2 to illustrate the necessity of transaction logic. The tables Z, S and T in TX_1 can refer to Fig. 4. z_2, z_3, s_3 and t_3 are double typed non-key columns. $p_{i,j}$ is the j^{th} symbolic parameter of i^{th} SQL operation. When generating synthetic workloads based on TX_1, we usually instantiate the symbolic parameters with random values generated by given data distributions. However, there may be significant performance differences between such synthetic workloads and the real application workloads. Now suppose we have deployed a distributed database, and tables Z, S, and T are all hash-partitioned by the first column of the primary key. Due to business logics, there are usually some correlations between SQL parameters in real workloads. For example, parameters $p_{1,2}$ and $p_{2,2}$ are of the same value in a high probability (e.g., 99%) in actually. Thus the first two SQL operations in TX_1 do not bring many distributed transactions. But in the synthetic workloads, 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], [12]. More than that, if parameters $p_{3,1}$ and $p_{4,2}$ in the last two SQL operations of TX_1 are always the same in the real workload, there will be no deadlocks. However, these two parameters may be different during the synthetic workload generation, 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 characteristic 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, it calls for a general framework to model transaction logic that abstracts from different applications. We propose to 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 decides the hidden semantics among/in SQL operations. Even if the transaction is executed as a stored procedure, the relationship among parameters of the stored procedure call may still affect the execution cost of the transaction. There are four types of relationships we care about after the investigation of existing 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**' refers to the relationship between two items, e.g., two parameters, that can be represented by a linear function, and it is a complement and extension to the equal relationship. Fourthly, '**between relationship**' is proposed for predicates like '$col\ between\ p_{i,j}\ and\ p_{i,j+1}$', in which $p_{i,j+1}$ has a between relationship with $p_{i,j}$. In order to simplify the problem, we only consider the relationship (i.e., dependency) between current parameter and the previous parameters (or return items).

We give the formal definition of transaction logic in Def.3. O_i represents the i^{th} operation in the transaction template; $r_{i,j}$

(resp., $p_{i,j}$) is the j^{th} return item (resp., parameter) of O_i, where i and j are both count from 1; Δ denotes the average increment of one parameter relative to another; Pr represents the probability that the parameter dependency should be satisfied; (a, b) are two coefficients for characterizing the linear function. We use the abbreviations ER, IR, LR and BR to denote the equal relationship, inclusive relationship, linear relationship and between relationship, respectively.

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

Note that if a parameter $p_{i,j}$ has dependency \$1, then both dependencies \$2 and \$3 do not need to exist because $p_{i,j}$ can be represented by $(p_{i,l} + \Delta)$. For instance, in Section IV-A, the dependencies \$2 of parameter $p_{2,2}$ in TX_1 includes [$p_{1,2}$, $p_{2,2}$, ER, 99%], and there is a dependency [$p_{3,1}$, $p_{4,2}$, ER, 100%] for parameter $p_{4,2}$. Moreover, since operations in the loop structure are usually performed multiple times, we need to consider the value changes of SQL parameters in these operations during loop executions. Dependencies \$3 are used to represent the linear relationships between the values of the same parameter in successive runs for loop operations. For example, [$p_{i,j}$, LR, 90%, $(2, 1)$] indicates that there is a 90% probability that the value of $p_{i,j}$ in current loop execution is $2x+1$ where x 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, e.g., <$p_{i,l}$, $p_{i,j}$>, that satisfy the between relationship in transaction templates, and then traverse workload traces to get the average increment, i.e., Δ, for each pair. 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, e.g., $p_{m,n}$ (resp., return items, e.g., $r_{u,v}$), and count the number of transaction instances in which the pairs, e.g., <$p_{m,n}$, $p_{i,j}$> (resp., <$r_{u,v}$, $p_{i,j}$>) satisfy the equal relationship (resp., equal relationship or inclusive relationship). Assuming that there are totally V parameters together with return items in the transaction template, there are nearly $\frac{V^2}{2}$ numbers of above pairs. The complexity of step 3 is O(KV^2).

Step 4: Collect statistics for LR. LR only involves numeric typed parameters and return items, and the return items must be 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., <$p_{m,n}$, $p_{i,j}$>, <$r_{u,v}$, $p_{i,j}$>) 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. 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, 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 or equal to 1.

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 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.

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 $N = 10^4$.

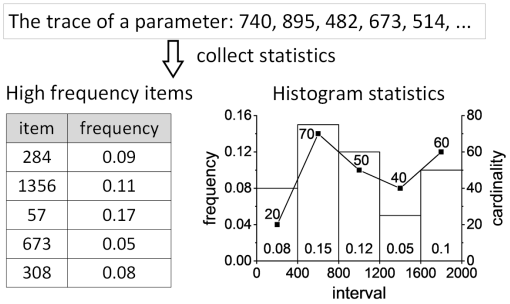


Fig. 5. An example for illustrating S-Dist

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

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 in 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. 5 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.

Before using S-Dist to instantiate the symbolic parameters, we need first do data transformation for HFI, because the data items in HFI are extracted from the real workload and may not exist in the synthetic database. Suppose the random column generator in Section III for our example is 'index = ranInt[1, 400], value = index * 5', where 400 is the column cardinality.

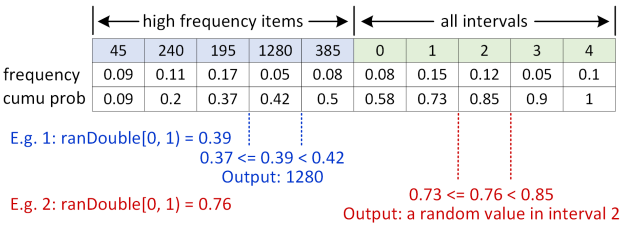


Fig. 6. An example of parameter generation

We regenerate the data items in HFI using our column generator, as shown in Fig. 6, where, for example, data item 57 is replaced by 195. For ensuring that the generated parameters conform to the desired frequency distribution, we calculate a cumulative probability array based on the frequencies of high frequency items and all intervals ('cumu prob' in Fig. 6). Then, we use random values between 0 and 1, which can be mapped to the cumulative probability array utilizing binary search, to select appropriate parameter values for filling the predicate. In Fig. 6, there are two parameter generation examples. The complexity of parameter generation is O(log$(H+I)$), which is dominated by the binary search.

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. 6, 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.

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 i^{th} second,

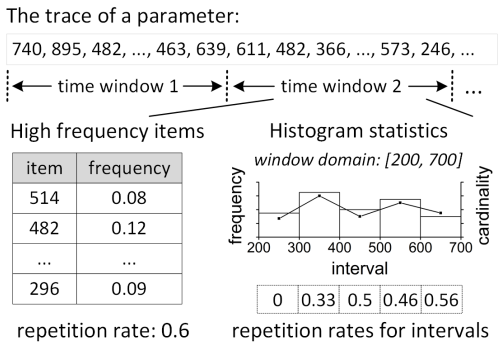


Fig. 7. An example for illustrating C-Dist

all workload requests only access the i^{th} record in the table. Assume that the database throughput 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. 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 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. 7 continues the example in Fig. 5 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.

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** i is valid && $\Gamma[i] < \mathcal{C}[i] * \Psi[i]$ **then**
8: Put the parameter p in $\Omega^t[i]$, $\Gamma[i]++$
9: **end if** // i is invalid when p is not in the current window domain.
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

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. Finally, in lines 11-13, we generate random parameters added to each interval to reach the cardinality requirement. Now for parameter generation against a certain interval, we only need to randomly select a candidate parameter as the output. In addition, 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 workloads.

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. We present the details of workload generation from three levels: thread model, transaction execution and parameter instantiation.

Thread Model. Users can configurate 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. 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. 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.

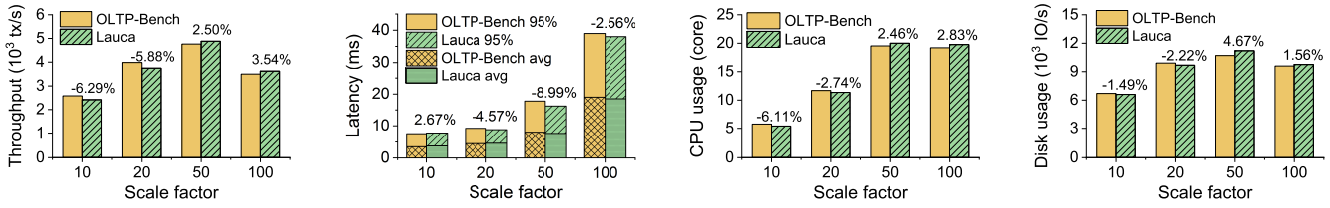
Workloads. Two standard benchmarks are used throughout the experiments: TPC-C [9] and SmallBank [10]. TPC-C is one of the most widely used industrial-level OLTP benchmark. It involves nine tables and five types of transactions that simulate the activities found in complex OLTP applications. SmallBank abstracts the operations in banking applications, and it includes three tables and 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 and Setup. In the transaction logic extraction, the number K of transaction instances and the group 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. Both MySQL and PostgreSQL are deployed on a single server, respectively. By default, Lauca/OLTP-Bench/YCSB is deployed on a single server that does not deploy any database system.

A. Fidelity of Synthetic Workloads

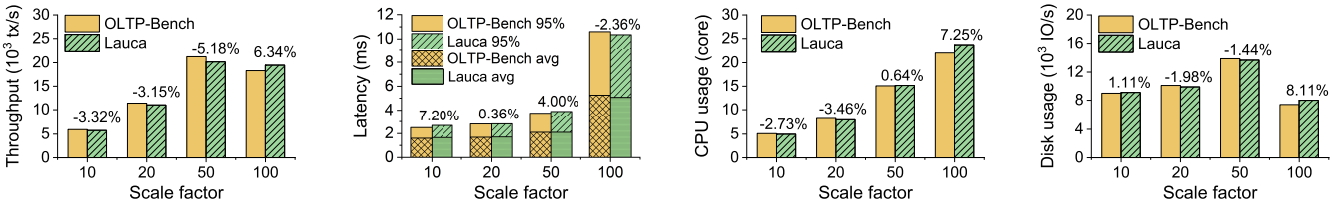
The similarity between synthetic workloads and real application workloads is called workload *fidelity*, which is the most essential target when designing Lauca. It is measured by performance deviations obtained by running synthetic workload and real application workload on the same database system.

Fig. 8-9 show the deviations in performance metrics between the synthetic workloads generated by Lauca and the real workloads generated by OLTP-Bench, under different scale factors. The concurrency of database requests is the same as the scale factor. There are two groups of experiments, respectively, for the executions of TPC-C workloads on PostgreSQL



(a) Throughput (b) Latency (c) CPU usage (d) Disk usage

Fig. 8. Deviations in performance metrics for TPC-C workloads on PostgreSQL database



(a) Throughput (b) Latency (c) CPU usage (d) Disk usage

Fig. 9. Deviations in performance metrics for SmallBank workloads on MySQL database

database and SmallBank workloads on MySQL database. In Fig. 8(a) and Fig. 9(a), we present the transaction execution throughputs of real workloads and synthetic workloads. From the results we can see that the throughputs of the two workloads are very similar, and the biggest deviation is as low as 6.29% in Fig. 8(a) and 6.34% in Fig. 9(a) respectively. For average latency and 95% latency metrics, in Fig. 8(b) and Fig. 9(b), we can see that the synthetic workload is very close to the real workload on both two metrics, with the maximum deviation between the two workloads is only 8.99% and 7.20% respectively. Fig. 8(c)-8(d) and Fig. 9(c)-9(d) report the CPU and Disk usages of the 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.

Due to space constraints, we leave the experimental results for TPC-C workloads on MySQL database and SmallBank workloads on PostgreSQL database in our technical report [26]. The experimental results are the same as in Fig. 8-9. Overall, whether for complex workloads of TPC-C or simple workloads of SmallBank on MySQL or PostgreSQL database, the synthetic workloads generated by Lauca are consistent with the real workloads from OLTP-Bench on various performance metrics. It conforms that the synthetic workload generated by Lauca has extremely high fidelity.

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.

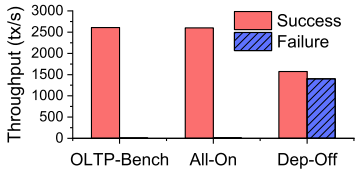


Fig. 10. The impact on transaction semantics

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. 10 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 failure throughput refers to the throughput of failed transactions. In Fig. 10, 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 large number of transaction rollbacks. And because no new order is generated, Delivery transactions cannot be successfully executed too. Overall, the transaction logic can effectively ensure that the transaction semantics of the synthetic workload are consistent with the real workload.

To further explore the impact of transaction logic on other aspects, we focus on three types of transactions that can be successfully executed, namely the Payment, OrderStatus and StockLevel transactions. In Fig. 11, we present the transaction throughputs, latencies and deadlock throughputs of workloads generated by OLTP-Bench and Lauca. There are five groups

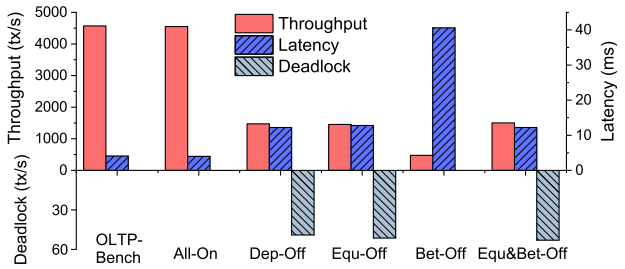


Fig. 11. The impact on transaction conflict intensity, deadlock possibility and scan data volume

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. 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. Under the Equ-Off, read and write operations on the same record are likely to become on different records, which significantly increase the likelihood of transaction conflicts and deadlocks. 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.

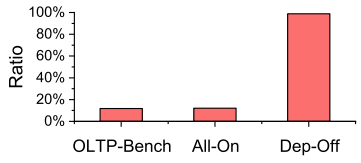


Fig. 12. The impact on distributed transaction ratio

The distributed transaction ratio has been proved to have a significant impact on database performance in many works [7], [12], [13]. We simulate a distributed environment on the single-node MySQL database by assuming that the data is hash-partitioned into five virtual nodes according to Warehouse ID. If the data in a transaction involves multiple virtual nodes, then the transaction is considered a distributed transaction. We count the distributed transaction ratios of workloads on the application side. In Fig. 12, 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. 12 are only

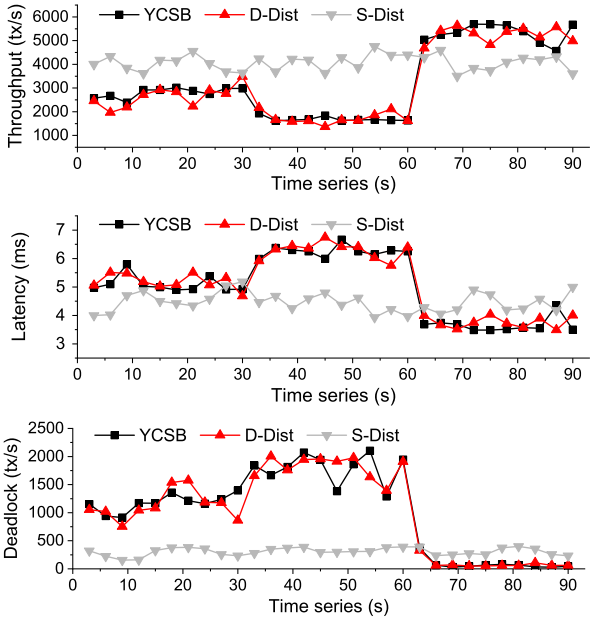


Fig. 13. Exploring S-Dist and D-Dist for skewed and dynamic workloads

Payment transactions. As can be seen from the results, when we turn off the parameter dependency information (Dep-Off), the distributed transaction ratio increases dramatically. This is because the Warehouse ID parameters of four write operations in the Payment transaction are randomly generated under Dep-Off, thus there is a high probability of becoming distributed transactions. Overall, the control of transaction logic can effectively guarantee the similarity of distributed transaction ratio between synthetic workload and real workload.

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. 13 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. 13 shows the dynamic changes of transaction throughput, latency and deadlock throughput for workloads generated by YCSB and Lauca. Lauca 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. 13), which is defined for the whole workload time and does not take into account the dynamic changes of the workload.

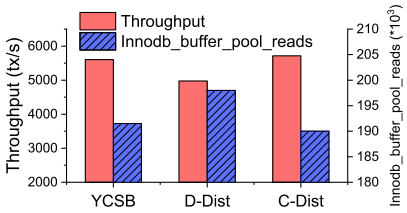


Fig. 14. Exploring C-Dist for continuous workloads

The evaluation workload in Fig. 14 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. 14 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 cannot 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 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. Performance of Lauca

In this section, we use TPC-C workload traces to study the performance of Lauca, which is deployed on four servers.

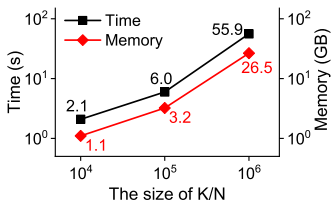


Fig. 15. Performance of transaction logic extraction ($K = N$)

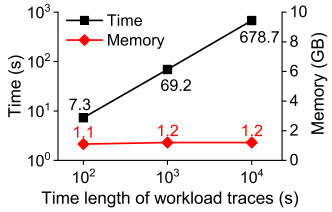


Fig. 16. Performance of C-Dist extraction

Fig.15 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 VII-A are all conducted when K and N are set to 10^4, and the 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.16 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.16, 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.

VIII. 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. 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. A similar point about the discrepancy between current benchmarking practice and the real world workloads was made in a recent work by Vogelsgesang et al. [24].

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] 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 trace. 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], [18], [19] 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. Workload analyzers [20], [21] are designed to study and better understand the application workloads, but neither of which can generate synthetic workloads. There are workload generators [22], [23] for database performance benchmarking. Jeong et al. [22] proposes a workload generator for simulating a realistic hardware resource consumption status. NoWog [23] introduces a workload description language for generating synthetic workloads benchmarking NoSQL databases. None of these works [22], [23] can be used to simulate the various workloads of actual OLTP applications for application-oriented database performance evaluation.

IX. Discussion

Data privacy protection. According to Lauca's workflow, only workload description information and workload statistics are exposed to testers during the evaluation. Neither the data in real databases nor the logged workload traces are visible to testers. Components in Lauca that involve real application data are manipulated by data owners in the production environment. Moreover, we can also anonymize the table names, column names and transaction names in both workload description information and workload statistics. But at present, we cannot prevent doing reverse engineering on workload statistics to get sensitive information.

Limitations of transaction logic. The transaction logic in Def.3 aims to support the common workloads in real world applications. There are still three deficiencies in our current work. Firstly, the relationships in Def.3 are not complete. For example, the relationship between two parameters may be represented by a quadratic function, which cannot be covered currently. Secondly, only the relationship of two data items is considered at present, without considering the relationship among more data items. Thirdly, the importance of different relationships may vary greatly and it is difficult to quantify them. 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 the workload changes frequently, the window size should be set to a small value, otherwise it can be set 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. At present, the number of intervals is selected by experiments. We can apply existing academic and industrial methods [27], [28] to choose the number of intervals in future work.

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.

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, 2016, 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] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "Qagen: generating query-aware test databases," in SIGMOD, 2007, pp. 341–352.

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

[20] 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.

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

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

[23] 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.

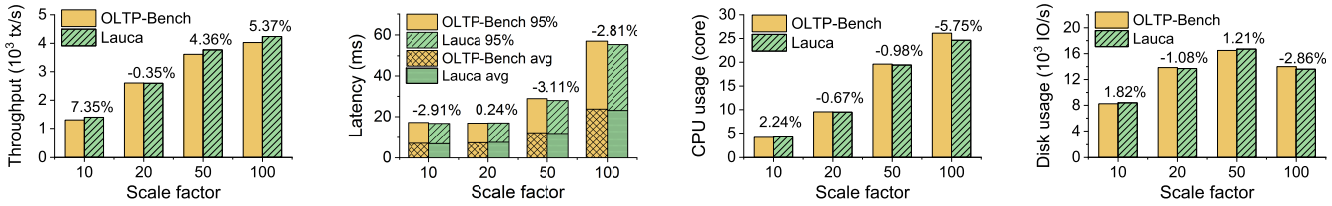
[24] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then, "Get real: How benchmarks fail to represent the real world," in DBTest, 2018, pp. 1:1–1:6.

[25] A. Pavlo, "What are we doing with our lives? Nobody cares about our concurrency control research," in SIGMOD, 2017, pp. 3.

[26] Technical report of Lauca, https://github.com/LiYuming/lauca-paper.

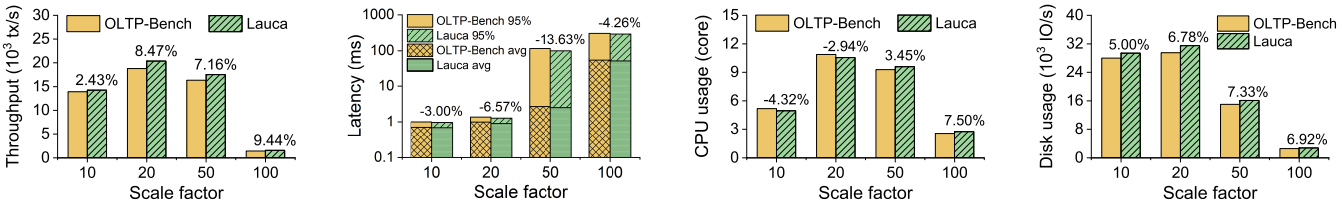
[27] L. Birgé, and Y. Rozenholc, "How many bins should be put in a regular histogram," in ESAIM: PS, 2006, pp. 24–45.

[28] Oracle Histograms, https://docs.oracle.com/database/121/TGSQL/.



(a) Throughput (b) Latency (c) CPU usage (d) Disk usage

Fig. 17. Deviations in performance metrics for TPC-C workloads on MySQL database



(a) Throughput (b) Latency (c) CPU usage (d) Disk usage

Fig. 18. Deviations in performance metrics for SmallBank workloads on PostgreSQL database

Additional experimental results: Fig. 17-18 present the experimental results for TPC-C workloads on MySQL database and SmallBank workloads on PostgreSQL database. The experimental results are substantially the same as in Fig. 8-9, and are not described in detail. There are two points need to be noted: 1) In Fig. 17, although the throughput at SF (scale factor) 100 is higher than that at SF 50 (see Fig. 17(a)), the disk IOPS at SF 100 is less than that at SF 50 (see Fig. 17(d)). This is because the group commit of MySQL flushes more transaction logs each time at SF 100. 2) For very large data scales and high request concurrencies, the performance of MySQL and PostgreSQL may be significantly degraded, as shown in Fig. 8(a),9(a),18(a), which is also a major bottleneck for stand-alone databases. For example in Fig. 18(a), the throughput decreases sharply at SF 100, this is caused by a low cache hit ratio. A large number of disk random reads and writes severely block the database worker threads. RAID cache can effectively support tremendous sequential disk write IO, but it does not work for random disk IO.

Transaction templates for TPC-C workloads: Below we present the actual input transaction templates of Lauca for five TPC-C's transactions. The symbolization of SQL parameters is done automatically by Lauca, so the parameters in the input transaction templates are all question mark symbols (i.e., '?'). But the symbol '?' in TX[transactionName, ?, true] is the unspecified transaction proportion, which can be specified by users or extracted from workload traces. The 'true' in TX[transactionName, ?, true] indicates that this transaction is executed in a precompiled manner. 'Branch', 'Separator' and 'EndBranch' are used to define the branch structure in transactions. 'Multiple' and 'EndMultiple' are used to define the loop structure in transactions. The 'batch' in Multiple[batch] indicates that write operations in the loop structure are executed in batch mode.

NewOrder transaction:

TX[NewOrder, ?, true]

SELECT C_DISCOUNT, C_LAST, C_CREDIT FROM CUSTOMER WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ?;

SELECT W_TAX FROM WAREHOUSE WHERE W_ID = ?;

SELECT D_NEXT_O_ID, D_TAX FROM DISTRICT WHERE D_W_ID = ? AND D_ID = ? FOR UPDATE;

UPDATE DISTRICT SET D_NEXT_O_ID = D_NEXT_O_ID + 1 WHERE D_W_ID = ? AND D_ID = ?;

INSERT INTO OORDER (O_ID, O_D_ID, O_W_ID, O_C_ID, O_ENTRY_D, O_OL_CNT, O_ALL_LOCAL) VALUES (?, ?, ?, ?, ?, ?, ?);

INSERT INTO NEW_ORDER (NO_O_ID, NO_D_ID, NO_W_ID) VALUES (?, ?, ?);

Multiple[batch]

SELECT I_PRICE, I_NAME, I_DATA FROM ITEM WHERE I_ID = ?;

SELECT S_QUANTITY, S_DATA, S_DIST_01, S_DIST_02, S_DIST_03, S_DIST_04, S_DIST_05, S_DIST_06, S_DIST_07, S_DIST_08, S_DIST_09, S_DIST_10 FROM STOCK WHERE S_I_ID = ? AND S_W_ID = ? FOR UPDATE;

UPDATE STOCK SET S_QUANTITY = ?, S_YTD = S_YTD + ?, S_ORDER_CNT = S_ORDER_CNT + 1, S_REMOTE_CNT = S_REMOTE_CNT + ? WHERE S_I_ID = ? AND S_W_ID = ?;

INSERT INTO ORDER_LINE (OL_O_ID, OL_D_ID, OL_W_ID, OL_NUMBER, OL_I_ID, OL_SUPPLY_W_ID, OL_QUANTITY, OL_AMOUNT, OL_DIST_INFO) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?);

EndMultiple

EndTX

Payment transaction:

TX[Payment, ?, true]

UPDATE WAREHOUSE SET W_YTD = W_YTD + ? WHERE W_ID = ?;

SELECT W_STREET_1, W_STREET_2, W_CITY, W_STATE, W_ZIP, W_NAME FROM WAREHOUSE WHERE W_ID = ?;

UPDATE DISTRICT SET D_YTD = D_YTD + ? WHERE D_W_ID = ? AND D_ID = ?;

SELECT D_STREET_1, D_STREET_2, D_CITY, D_STATE, D_ZIP, D_NAME FROM DISTRICT WHERE D_W_ID = ? AND D_ID = ?;

Branch

SELECT C_FIRST, C_MIDDLE, C_ID, C_STREET_1, C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_SINCE FROM CUSTOMER WHERE C_W_ID = ? AND C_D_ID = ? AND C_LAST = ? ORDER BY C_FIRST;

Separator

SELECT C_FIRST, C_MIDDLE, C_LAST, C_STREET_1, C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_SINCE FROM CUSTOMER WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ?;

EndBranch

Branch

SELECT C_DATA FROM CUSTOMER WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ?;

UPDATE CUSTOMER SET C_BALANCE = ?, C_YTD_PAYMENT = ?, C_PAYMENT_CNT = ?, C_DATA = ? WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ?;

Separator

UPDATE CUSTOMER SET C_BALANCE = ?, C_YTD_PAYMENT = ?, C_PAYMENT_CNT = ? WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ?;

EndBranch

INSERT INTO HISTORY (H_C_D_ID, H_C_W_ID, H_C_ID, H_D_ID, H_W_ID, H_DATE, H_AMOUNT, H_DATA) VALUES (?, ?, ?, ?, ?, ?, ?, ?);

EndTX

OrderStatus transaction:

TX[OrderStatus, ?, true]

Branch

SELECT C_FIRST, C_MIDDLE, C_ID, C_STREET_1, C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_SINCE FROM CUSTOMER WHERE C_W_ID = ? AND C_D_ID = ? AND C_LAST = ? ORDER BY C_FIRST;

Separator

SELECT C_FIRST, C_MIDDLE, C_LAST, C_STREET_1, C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT, C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT, C_SINCE FROM CUSTOMER WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ?;

EndBranch

SELECT O_ID, O_CARRIER_ID, O_ENTRY_D FROM OORDER WHERE O_W_ID = ? AND O_D_ID = ? AND O_C_ID = ? ORDER BY O_ID DESC LIMIT 1;

SELECT OL_I_ID, OL_SUPPLY_W_ID, OL_QUANTITY, OL_AMOUNT, OL_DELIVERY_D FROM ORDER_LINE WHERE OL_O_ID = ? AND OL_D_ID = ? AND OL_W_ID = ?;

EndTX

Delivery transaction:

TX[Delivery, ?, true]

Multiple

SELECT NO_O_ID FROM NEW_ORDER WHERE NO_D_ID = ? AND NO_W_ID = ? ORDER BY NO_O_ID ASC LIMIT 1;

DELETE FROM NEW_ORDER WHERE NO_O_ID = ? AND NO_D_ID = ? AND NO_W_ID = ?;

SELECT O_C_ID FROM OORDER WHERE O_ID = ? AND O_D_ID = ? AND O_W_ID = ?;

UPDATE OORDER SET O_CARRIER_ID = ? WHERE O_ID = ? AND O_D_ID = ? AND O_W_ID = ?;

UPDATE ORDER_LINE SET OL_DELIVERY_D = ? WHERE OL_O_ID = ? AND OL_D_ID = ? AND OL_W_ID = ?;

SELECT SUM(OL_AMOUNT) AS OL_TOTAL ##decimal FROM ORDER_LINE WHERE OL_O_ID = ? AND OL_D_ID = ? AND OL_W_ID = ?;

UPDATE CUSTOMER SET C_BALANCE = C_BALANCE + ?, C_DELIVERY_CNT = C_DELIVERY_CNT + 1 WHERE C_W_ID = ? AND C_D_ID = ? AND C_ID = ?;

EndMultiple

EndTX

StockLevel transaction:

TX[StockLevel, ?, true]

SELECT D_NEXT_O_ID FROM DISTRICT WHERE D_W_ID = ? AND D_ID = ?;

SELECT COUNT(DISTINCT (S_I_ID)) AS STOCK_COUNT ##integer FROM ORDER_LINE, STOCK WHERE OL_W_ID = ? AND OL_D_ID = ? AND OL_O_ID < ? AND OL_O_ID >= ? AND S_W_ID = ? AND S_I_ID = OL_I_ID AND S_QUANTITY < ?;

EndTX