A PROOF

A.1 Property 1

PROPERTY 1. *If there exist multiple time-overlapping operations writing the same data, the first returned write operation would always firstly acquire the lock.*

PROOF. Suppose the operation that firstly acquires the lock is op, and the end operation (i.e., commit or abort) of its corresponding transaction is op'. Since the client sends operations inside a transaction serially to the database, we can infer that the client sends op' after receiving the response of op. That is, the start time of op' is larger than the end time of op, i.e., $op.t_e < op'.t_s$. Meanwhile, other operations writing the same data are blocked by the data's write lock until op' releases the lock. Thus, these operations execute after $op.t_e$ and $op'.t_s$, and they must return later than op. Taken together, we prove that the lock is acquired by the first returned write. □

Property 1 Always Holds Regardless of Network Delays. As shown in Fig. 1, considering two operations op_1 and op_2 with overlapped time intervals, where op_1 is involved in transaction txn_1 and op_2 is involved in transaction txn_2. Suppose they write the same data item. We assume that op_1 executes at time t_1 inside the database (i.e., server side), while op_2 executes after op_1 inside the database at time t_2. According to S2PL, the lock acquired by op_1 is held until txn_1 commits inside the database at time t_c.

The interactions between the client side and server side include: ❶ The client sends the operation op_1 in transaction txn_1 to the database on the server side at t_{1s}; ❷ The client receives the result of op_1 at t_{1e}; ❸ The client sends the commit operation for txn_1 to the database at t_{cs}; ❹ The client receives the commit result at t_{ce}; ❺ The client sends the operation op_2 in transaction txn_2 to the database at t_{2s}; ❻ The client receives the result of op_2 at t_{2e}.

Network delay might reorder the arrival of operation results at the client, i.e., affecting t_{1e}, t_{ce}, and t_{2e} in this example. Moreover, as each request always follows the order of request sending on the client side, operation execution inside the database, and result reception on the client side, we can infer that $t_{1s} < t_1 < t_{1e}$, $t_{cs} < t_c < t_{ce}$, and $t_{2s} < t_2 < t_{2e}$. Besides, both the client sends the operations and the database executes the operations within a transaction one after another, and the commit operation is sent by the client when it receives the completion results of all the previous operations in that transaction. Hence, txn_1's commit operation is not sent until op_1's result has arrived at the client, i.e., $t_{1e} < t_{cs}$. On this basis, we can derive the event order as follows:

$$t_{1s} < t_1 < t_{1e} < t_{cs} < t_c < t_{ce}. \tag{1}$$

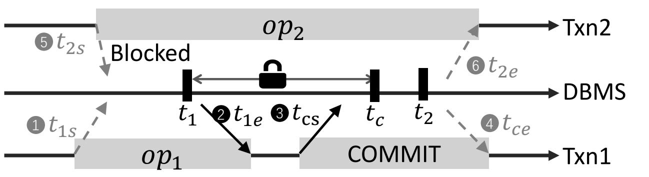


Figure 1: Example of Property 1

Under S2PL, txn_1's write lock is held inside the database until it commits. Network delays might slow message transmissions between the client and server, but it cannot reorder lock operations. That is, op_2 cannot start inside the database until txn_1's commit is processed inside the database. Thus, we have

$$t_c < t_2 < t_{2e}. \tag{2}$$

Based on Eq. 1 and Eq. 2, we have:

$$t_{1e} < t_c < t_{2e}.$$

Therefore, Property 1 remains valid even in the presence of network delays.

A.2 1-minimality Guarantee of *Pisco*

1-minimality Guarantee of *Pisco*. Theorem 1 proves that *Pisco* provides a guarantee of 1-minimality for its final reduced case C_r. In this case, it satisfies the following two properties: (a) no irrelevant operation is retained in C_r, and (b) every bug-related operation is retained in C_r.

LEMMA A.1. *No bug irrelevant operation is retained in C_r, with C_r as the final reduced case produced by Pisco.*

PROOF. Suppose that there exists a bug irrelevant operation op that is retained in the final reduced case C_r produced by Pisco. Since *Pisco* retains op only if removing its reduction unit prevents the bug from reproducing, then we can infer that op's reduction unit must include at least one operation op^* that is indispensable for triggering the bug. In addition, as all operations in op's reduction unit depend on op, then op^* also depends on op. Thus, removing op changes the execution result of op^* and would prevent the bug reproduction. This contradicts the assumption that op is irrelevant to the bug. □

LEMMA A.2. *Every bug-related operation is retained in C_r, with C_r as the final reduced case produced by Pisco.*

PROOF. Let op be an operation that is truly indispensable for triggering the bug. That is, any reduced case that omits op will fail to reproduce the bug. Since any attempted reduction unit removal that prevents reproduction is rolled back by *Pisco*'s trial-and-error strategy, op is kept in the final reduced case C_r. □

THEOREM A.3. *The final reduced case C_r produced by Pisco is 1-minimality.*

PROOF. From *Lemma* 1, we know that C_r contains no irrelevant operations. From *Lemma* 2, we know that C_r contains every operation that is indispensable for reproducing the bug. Thus, C_r contains exactly the operations required for bug reproduction. Therefore, removing any operation from this case prevents its reproduction, which guarantees 1-minimality. □

B COMPLEXITY ANALYSIS

B.1 Operation Batch Sequence Construction

Given a raw case consisting of N operations, we analyze the individual time complexity of constructing a batch. For each operation op to be added, we first check whether op conflicts with any operation within the batch. Next, we search all operations that ① conflict with op, ② time-overlap with op, and ③ have not yet been added to the batch. We then deduce the execution order between op and these overlapping operations. After checking *op*, we determine whether the current batch can be frozen.

More specifically, to determine whether *op* conflicts, we maintain a hash set of the data items accessed by the current batch, allowing $O(1)$ time to check for conflicts. Then, suppose there are k threads, and each thread contains, on average, m operations that conflict and time-overlap with *op*. Since operations within each thread are already sorted by timestamp, we do not need to iterate over all operations within each thread. Consequently, the search complexity is $O(k \cdot m)$. Moreover, we mirror the DBMS's internal state using hash structures, allowing us to deduce the order between *op* and each time-overlapping operation in $O(1)$ time. Thus, the overall deduction time complexity for all time-overlapping operations is $O(k \cdot m)$. Besides, with the help of the written data set and the hashmap that records data items accessed by all remaining operations, we can determine whether the current batch could be frozen with a complexity of $O(1)$. Finally, assume that we require checking p operations, the individual time complexity is $O(k \cdot m \cdot p)$.

Regarding the entire operation batch sequence, suppose that the average batch size is $|\mathcal{B}|$, resulting in $\frac{N}{|\mathcal{B}|}$ batches in total. The overall time complexity is $O(\frac{k \cdot m \cdot p \cdot N}{|\mathcal{B}|})$. In practice, testing workloads are typically generated through long-running, high-throughput testing, which produces a large number of operations using a fixed number of threads. As a result, k, m, and p are much smaller than N Therefore, the overall time complexity could be simplified to $O(N)$.

B.2 Recursive Operation Reduction

We analyze the worst-case complexity of our dependency-aware case reduction. Suppose a bug case consists of N operations, where only M operations are related to the bug. Note, M is generally much smaller than N, i.e., $M << N$.

Our *Pisco* aims to aggregate bug-related operations in the bug case. To this end, for any operation *op*, we define the reduction unit of *op* as the set containing the operation *op* itself and all operations that depend on *op*. To construct the reduction unit for each operation, we propose to build a directed *operation dependency graph* $\mathcal{G} = (V, E)$. In this graph, each node in V represents an operation, and each edge $e =< op, op' >$ represents that op' directly accesses the data version created by *op*. On this basis, for any operation *op*, its reduction unit can be constructed by traversing all of its "reachable" operations in the operation dependency graph. Note, each operation has its specific reduction unit, and its reduction unit exactly equals the union of its direct successors' reduction units.

Pisco applies multiple rounds of divide-and-conquer and trial-and-error to validate whether each reduction unit is related to the bug. In each round, it takes all operations without predecessors in the operation dependency graph as inputs. There are two main reasons for this step. First, as each operation's reduction unit contains all operations reachable from it, the union of reduction units of operations without predecessors covers all operations in the dependency graph. Second, these reduction units are the largest units, whose removal substantially enhances the divide-and-conquer reduction efficiency.

Then, it uses the divide-and-conquer and trial-and-error approach to remove the reduction units irrelevant to the bug. Note, remove a reduction unit also requires remove all its contained operations from the dependency graph. At that, it retains all reduction units that are necessary for the bug reproduction. Then, we collect the operations corresponding to the retained reduction units and put them in the bug reproduction set. This is because these operations would influence all operations in their reduction units, and they are indispensable for bug reproduction. Next, we remove the newly collected operations from the dependency graph. Then, we can infer that all operations without predecessors in the operation dependency graph are the newly collected operations' direct successors. Then, we gather the reduction units of the newly collected operations' direct successors and start the next round of divide-and-conquer and trial-and-error process, which aims to find finer-grained reduction units containing bug-related operations.

Suppose *Pisco* employs k rounds of the divide-and-conquer and trial-and-error in total. The i-th round takes n_i operations as input (i.e., n_i reduction units) and finally retains m_i reduction units. Then, m_i operations corresponding to the retained reduction units are put in the bug reproduction set. On this basis, we have $\sum_{i=1}^{k} n_i \leq N, \sum_{i=1}^{k} m_i \leq M$.

More specifically, the i-th round takes n_i operations as input. Then, these n_i operations are evenly divided into two subsets. Each subset takes a trial to validate whether the bug can still be reproduced after removing all the reduction units corresponding to this subset's operations. If such removal prevents the bug from being reproduced, we recursively divide this subset. In the worst case, the m_i bug-related operations are always divided into different subsets. In such cases, the divide-and-conquer can be divided into two phases according to the number of divided subsets.

In the first phase, the number of subsets is less than m_i, and each of them contains at least one bug-related operation. Thus, they must be further divided until the number of subsets is larger than m_i. At that time, some of these subsets do not contain bug-related operations, which can be safely removed. Since the number of subsets doubles with each iteration, the process consists of $(\lfloor \log_2 m_i \rfloor + 1)$ iterations. As a result, it takes $2+4+\cdots+2^{\lfloor \log_2 m_i \rfloor + 1} = 2^{\lfloor \log_2 m_i \rfloor + 2} - 2 \leq 4 \cdot m_i$ trials.

In the second phase, each iteration begins with m_i subsets, each of which contains a bug-related operation. Since these m_i subsets are evenly divided from the n_i input operations, each subset initially contains at most $\frac{n_i}{m_i}$ operations, all of which have no predecessors in the operation dependency graph. Then, *Pisco* divides each subset into two smaller subsets and validates each one with a trial. Next, *Pisco* divides each subset into two smaller subsets and validates each smaller subset with a trial. Note that one of the two smaller subsets does not contain the bug-related operation, then each subset can halve the number of its operations in an iteration. That is, each iteration first divides m_i subsets into $2 \cdot m_i$ smaller subsets, and then uses $2 \cdot m_i$ trials to remove the m_i smaller bug irrelevant subsets and retain the other m_i smaller bug-related subsets. As a result, it takes $\log(\frac{n_i}{m_i})$ iterations to reduce each subset containing $\frac{n_i}{m_i}$ operations to the smallest subset that exactly contains one bug-related operation. Then, the second phase takes $2 \cdot m_i \cdot \log(\frac{n_i}{m_i})$ trials to reduce all m_i subsets.

In summary, the total number of trials taken by *Pisco* is $\sum_{i=1}^{k} 4 \cdot m_i + 2 \cdot m_i \cdot \log(\frac{n_i}{m_i})$. By Jensen's inequality, $\sum_{i=1}^{k} 4 \cdot m_i + 2 \cdot m_i \cdot$

$\log(\frac{n_i}{m_i}) \leq 4 \cdot M + 2 \cdot M \log(\frac{N}{M})$. That is, the number of trials taken by *Pisco* is $O(M \log(\frac{N}{M}))$, where in general $M << N$.

Complexity Analysis of DDMin. DDMin also proposes a divide-and-conquer approach to validate whether each operation is related to the bug. Since DDMin does not deduce the execution order of conflicting operations with time overlaps, it might replay bug-related conflicting operations in an incorrect order, failing to trigger the bug even if the removed subset is bug irrelevant. In the worst case, it fails to reproduce the bug when attempting to remove each subset. Then, DDMin would divide this subset for further validations. In such cases, each iteration of the divide-and-conquer approach doubles the number of subsets and halves the operations in each subset. As a result, in the i-th iteration, DDMin validates 2^i subsets. When each subset contains only one operation, the iteration ends. Given a bug case with N operations, dividing the case into subsets containing a single operation requires ($\lfloor \log_2 N \rfloor + 1$) iterations. Thus, it takes $\sum_{i=1}^{\lfloor \log_2 N \rfloor + 1} 2^i = 2^{\lfloor \log_2 N \rfloor + 2} - 2 \leq 4 \cdot N$ trials. That is, the number of trials taken by DDMin is $O(N)$.

Complexity Analysis of C-Reduce. C-Reduce attempts to remove operations in a thread or a transaction at a time. In the worst case, bug-related operations are distributed in all threads, preventing these threads from removal. Then, C-Reduce must validate each transaction one by one. As a result, C-Reduce takes $T_{C-Reduce} = |TH| + |TXN|$ trials in the worst case, while $|TH|$ is the number of threads and $|TXN|$ is the number of transactions in the bug case. That is, the number of trials taken by C-Reduce is $O(|TH| + |TXN|)$.

B.3 LLM Enhanced Candidate Report Ranking

Suppose there exist K candidate bug reports for ranking, and each pair of candidate bug reports is repeatedly compared k times. Since each round filters half of the retained reports, then we can infer that the ranking procedure consists of $\lceil \log K \rceil$ rounds and there exist $\frac{K}{2^{i-1}}$ reports in the i-th round. Therefore, the total number of report pairs across all rounds is $\sum_{i=1}^{\lceil \log K \rceil} \frac{K}{2^i}$. As each pair of candidate bug reports is compared k times, the total number of LLM enhanced comparisons is $O(k \cdot K)$.

C MUTUAL INFORMATION CALCULATION

Probability Calculation. We calculate mutual information via a counting-based approach rather than estimation [2]. That is, probabilities are directly derived from feature frequencies in the existing bug reports, which effectively avoids the additional errors introduced by the probability estimation method. Specifically, let N denote the total number of bug reports (including duplicates), f_i represent the i-th feature, and y_j represent the j-th bug. Then, we have

- $p(f_i) = \frac{n(f_i)}{N}$, where $n(f_i)$ denotes the number of reports in which f_i appears.
- $p(y_j) = \frac{n(y_j)}{N}$, where $n(y_j)$ denotes the number of reports that are associated to y_j.
- $p(f_i, y_i) = \frac{n(f_i, y_j)}{n(F,Y)}$, where $n(f_i, y_j)$ denotes the number of reports that associated to y_j in which f_i appears, and $n(F, Y)$ is the total number of the feature-bug pairs.

Based on $p(f_i)$, $p(y_j)$, and $p(f_i, y_j)$, the mutual information of f_i can be calculated as:

$$w_i = \sum_{y_j \in Y} p(f_i, y_j) \log\left(\frac{p(f_i, y_j)}{p(f_i)p(y_j)}\right). \tag{3}$$

Handling Sparse Features and Over-fitting. A sparse feature means that a feature f_i appears in a few bug reports. In this situation, even if f_i is related to a specific bug y_j, f_i might not occur in the bug report associated with y_j. This makes it rather hard to capture their relationship. For example, a bug y_j that writes an incorrect version could be triggered by either an UPDATE or an INSERT statement. However, if only the case where y_j is triggered by an UPDATE statement is collected in the dataset, then the feature f_i representing the occurrence of INSERT statements would not appear in any bug report associated with y_j in the dataset. As a result, the mutual information between f_i and y_j is calculated as 0, which mistakenly represents that f_i is irrelevant to y_j. In other words, the mutual information only associates y_j with the features observed in the dataset (e.g., UPDATE statements), ignoring other relevant features such as INSERT statements that also cause the same bug, leading to the issue of overfitting to the collected bug reports.

To address this issue, the previous study proposes to apply the *Laplace smoothing* technique [1] in the calculation of mutual information. Specifically, *Laplace smoothing* assigns a small constant (e.g., 1) to the count of each feature-bug pair, which represents a potentially relevant but unobserved relationship. With this mechanism, the feature-bug pairs unseen in the dataset would have non-zero mutual information, which ensures that their relationships are not ignored. Following this approach, we also introduce *Laplace smoothing* to address the sparse feature issue. Specifically, we replace the joint probability calculation of $p(f_i, y_j)$ as follows:

$$p_{\text{LS}}(f_i, y_j) = \frac{n(f_i, y_j) + 1}{M + n(F, Y)}.$$

Here, we add a small constant 1 to each feature-bug pair count $n(f_i, y_j)$ to represent the potential relationship between the feature f_i and bug y_j. Moreover, we add the total number of distinct feature-bug pairs M to the denominator such that the sum of the replaced joint probability is guaranteed to be 1.

Note that our isolation bug report dataset has collected transactional features observed in isolation bugs as extensively as possible, which helps a lot in mitigating the impact of sparse features. Specifically, we have collected a comprehensive set of transactional bug reports from the year 2018 to 2025 based on recent studies [3, 4] and the official report repositories of the DBMSs tested in our experiments, including MySQL, TiDB, and MariaDB. Our dataset not only represents the scenario of transactional bugs in up-to-date DBMSs well, but also captures the relatively comprehensive transactional features that are relevant to bugs in these DBMSs.

D ACCURACY OF ORDER INFERENCE.

Operation Traces Collected from the Original Bug Triggering Scenario. In the original bug triggering scenario, the test client concurrently sends a large number of operations to the database. The database executes these operations according to its concurrency control implementation. Then, the database returns the result to the client. The trace of an operation op is formally defined by op's 1) start timestamp $op.t_s$ and end timestamp $op.t_e$; 2) accessed data item $op.d$; 3) operation type op_{type}; and 4) returned results, including the success or failure message, as well as the returned data versions of read operations.

Accurate Operation Order Inference Ensures Correct Simulation of the DBMS's State. *Pisco* focuses on isolation bugs that can be deterministically triggered by executing operations in a specific order, which account for over 94% of transactional bugs and cover the majority of isolation bugs [3]. In these deterministic scenarios [4, 5, 6, 7], the DBMS's state (e.g., version chains, lock tables) is determined by the execution order of conflicting operations. Then, if we can guarantee that the deduced execution order of conflicting operations is exactly the same as that in the original bug triggering scenario, we can also ensure that the simulation of version chains and lock tables is also consistent with that in the original bug triggering scenario. For example, suppose the current sate of simulated version chains and lock table is s_1, and the next deduced operation is a write operation op that writes on the data item A. Suppose from the operation traces, we observe that op writes successfully. Then, we should simulate the next state by appending a new version to A's version chain and add a write lock on A's lock table.

Combining Correct DBMS State Simulation with Operation Traces Ensures Accurate Next Operation Inference. *Pisco* employs a result-driven order inference approach that takes both the simulated DBMS state and operation results into account. Here, operation results can be found in the operation traces collected from the original bug triggering scenario, including the success or failure message, as well as the returned data versions of read operations. Note, we only consider conflicting operations with overlapping time intervals in the deduction phase. Suppose we have deduced the execution order of a certain number of operations, based on which the DBMS state is also simulated. Then, we leverage the following two rules to deduce the next operation:

❶ To prevent reads from being scheduled late, we propose to check all the read operations first among time-overlapping operations. If the checking operation is a read operation op_r, it is required to read the same data version as in the original bug-triggering execution. If the required version has been created by a write operation in previous operations with deduced order, the checking operation could read that version when it is set as the next operation. Then, this operation can be considered as the next operation. Otherwise, the required data version has not been created, and the read operation should wait until a write operation creates it. As a result, op_r cannot be considered as the next operation.

❷ If the checking operation is a write operation, it can be set as the next operation only if it could acquire the lock of its accessed data. It implies that a write operation can be set as the next operation if it satisfies both of the following conditions: ① *The write operation is not blocked by any operation's lock in the previous batches.* The DBMSs generally use the locking strategy (e.g., strict two-phase locking) to provide exclusive access to shared data. Then, operations would hold locks on their accessed data until the corresponding transactions end. Thus, the write operation can be set as the next operation only if the lock for the data item it writes is not held by any previous operations with deduced order. ② *The write operation acquires the lock.* Database lock contention occurs when multiple write operations attempt to modify the same data. From Property 1 in our paper, we know that the first returned write operation would always first acquire the lock. Therefore, only the first returned write can be set as the next operation.

Handling Edge Cases. Since our *Pisco* employs a result-driven order inference approach that takes both the simulated DBMS state and operation results into account, the edge case handling inside the database is implicitly reflected in the operation results that are recorded in the operation traces. For example, if our *Pisco* infers that a write operation can be set as the next operation since it could acquire the lock of its accessed data, but the operation trace shows that the write operation fails and is rolled back. Then, we can infer that the edge cases might be handled inside the database, such as the deadlock detection method inside the database has found a deadlock and rolls back the write. Nevertheless, our *Pisco* does not need to simulate the edge case handling process inside the database, and it only needs to track the edge case handling result. That is, since the operation trace shows that the write operation fails and is rolled back, *Pisco* neither adds a write lock to the operation's written data item nor appends a data version to the written data item's version chain when simulating the next state. Next, we take the version pruning and deadlock detection as examples to showcase that the edge-case handling does not influence the accuracy of the inferred order.

Version Pruning in MVCC. Pisco employs a result-driven order inference approach that takes both the simulated DBMS state and operation results into account. Specifically, for a read operation, its operation trace collected from the original bug triggering scenario includes the returned data version of this read operation. When deducing the operation execution order, it is required to read the same data version as in the original bug-triggering execution. We can leverage the state of version chain to deduce whether this read can be set as the next operation. In addition, when simulating version chains to infer execution order, *Pisco* adopts a conservative strategy that removes only versions not to be accessed by any remaining unordered operations. Consequently, version cleanup on the simulated version chains preserves the correctness of order inference, regardless the version pruning progress inside the database.

Deadlock Detection in 2PL. The deadlock detection and handling mechanism inside the database determines the execution results of operations involved in a deadlock. That is, the edge case handling inside the database is implicitly reflected in the operation results. Fortunately, the returned result (i.e., the success or failure message) has been recorded in the operation traces collected from the original bug triggering scenario. Thus, after deducing the next operation, our *Pisco* does not need to simulate the edge case handling when processing this operation inside the database, and it only needs to track the edge case handling result. Then, it uses the result of that operation to simulate the next database state after executing the next operation.

E ADAPTATION

E.1 Adaptation to OCC- and TO-based DBMSs

***Pisco*'s Order Inference Method.** Determining an execution order that could stably reproduce the isolation bug is the primary prerequisite for bug analysis. *Pisco* infers this order by simulating the DBMS's internal state, such as the state of version chains and lock table. It aims to simulate the DBMS's conflict resolution mechanism according to specific concurrency control protocols and thus mirror the same internal state evolution of the DBMS. Specifically, ❶ For operations whose time intervals have no overlaps, their order can be inferred directly based on their time intervals recorded in the client traces. ❷ For time interval overlapped operations that read and write to the same data version of a data item, the write operation that creates a new version must precede the operation reading this version. Since a version must be created before it can be read, the order can also be inferred. ❸ For time interval overlapped operations that read and write different data versions of a data item, *Pisco* determines their order by simulating the internal state evolution of the key data structures inside the DBMS. The simulation strategy depends on the concurrency control protocol used by the tested system. Specifically, *Pisco* simulates the state evolution of the version chain and the lock table since it targets MVCC and 2PL-based DBMSs. If the tested DBMS uses other concurrency control protocols, *Pisco* can be adapted by designing specialized simulation strategies to infer the execution order.

Adaptation to OCC-based DBMSs. For Optimistic Concurrent Control (OCC) based DBMSs, *Pisco* could infer the execution order of operations by simulating the state of version chains. Specifically, we can leverage the following two rules to help us accomplish the simulation. ❶ For any given transaction txn_i, all operations in txn_i should access the latest data versions before txn_i begins. For example, if an operation $op_{i,j}$ in txn_i accesses a version v_k of a data item A, then we can infer that txn_i must begin after the commit of the transaction creating (i.e., writes) the data version v_k. Moreover, if another transaction $txn_{i'}$ creates a data version newer than v_k, then commit operation of $txn_{i'}$ must happen later than the begin operation of txn_i. With this rule, we can infer the order between start and commit operations among different transactions, guaranteeing that each transaction accesses the same versions as in the original bug-triggering scenario. ❷ During the execution of a transaction txn_i, if another concurrent transaction $txn_{i'}$ modifies a data item accessed by txn_i and commits earlier, $txn_{i'}$ would fail to commit in its commit phase and should be aborted. With this rule, we could infer the order among commit operations in different transactions based on their commit states and accessed/modified data items. This ensures that we can reproduce the same version chain evolutions as in the original bug-triggering scenario. To summarize, in OCC-based DBMSs, we aim to order the begin and commit operations in the bug case. This is because a transaction's begin operation determines the versions accessed by all operations in this transaction; the commit operations append new versions to the version chain, and their order determines the version order and version evolution in the version chain. Besides, the order of other operations does not affect the result.

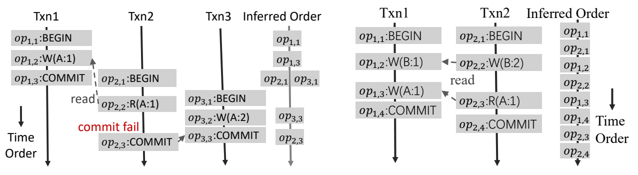
Example of OCC-based DBMSs. Fig. 2a shows an example. Suppose we have three transactions txn_1, txn_2, and txn_3 that access the same data item A. As analyzed above, in OCC-based DBMSs, our task is to order the start and commit operations of the three transactions. Since the order of operations whose time intervals have no overlaps can be easily deduced, we further focus on time interval overlapped operations, i.e., $op_{1,3}$&$op_{2,1}$ and $op_{2,3}$&$op_{3,3}$. First, as $op_{2,2}$ reads the version created by $op_{1,2}$, we can infer that txn_2 is executed based on the version chain updated by txn_1. According to the first rule, txn_2 begins after txn_1 commits, i.e., $op_{1,3} \prec op_{2,1}$. Second, transactions txn_2 and txn_2 are conflicted since they access or modify the same data item A. As $op_{2,3}$ in txn_2 fails to commit and $op_{3,3}$ in txn_3 successfully commits, we can infer that $op_{3,3}$ must execute earlier than $op_{2,3}$ based on the second rule, i.e., $op_{3,3} \prec op_{2,3}$.

Adaptation to TO-based DBMSs. The Timestamp Ordering (TO) based DBMSs assign each transaction a unique timestamp when the transaction enters the system. They use timestamps to determine the order of transaction execution, ensuring that conflicting operations are executed in timestamp order. On this basis, if the execution order of two conflicting operations in two transactions is deduced by *Pisco*, the timestamp order of the two transactions can also be deduced. Furthermore, the execution order of all the other conflicting operations in the two transactions can be also deduced according to their timestamp order. Specifically, we can leverage the following three rules to help us accomplish the timestamp simulation. ❶ If the time intervals of two conflicting operations recorded in the client traces have no overlaps, their associated transaction timestamp order can be inferred directly based on the time intervals of the two operations. ❷ If one of the two conflicting operations is a read operation and the other is a write operation, we can their execution order (i.e., timestamp order of associated transactions) according to their read-write relationship (i.e., a version's creator must precede its reader). ❸ Otherwise, we enumerate possible timestamp orders of the conflicting transactions until we find one order that reproduces the same execution results as those recorded in the traces. In this way, *Pisco* reproduces the same transaction timestamp order of the TO-based DBMSs as in the original bug-triggering scenario.

Example of TO-based DBMSs. Fig. 2b shows an example. Suppose the system uses the IL of *READ COMMITTED*, we have two conflicting transactions txn_1 and txn_2 that access the same data items A and B. In this case, $op_{1,2}$&$op_{2,2}$ and $op_{1,3}$&$op_{2,3}$ are conflicting operations with overlapped time intervals. From rule 2 described above, since $op_{2,3}$ reads the version created by $op_{1,3}$, we can infer that $op_{2,3}$ executes after $op_{1,3}$. Moreover, we can also infer that the timestamp of txn_1 is smaller than txn_2. Based on the timestamp order of the two transactions, we can infer that $op_{2,2}$ also executes after $op_{1,2}$. Additionally, according to the IL of *READ COMMITTED*, we can infer that $op_{1,4}$ executes before $op_{2,3}$ even if their time intervals are overlapped. To summarize, for each pair of conflicting operations in txn_1 and txn_2, the operation in txn_1 consistently precedes the operation in txn_2.

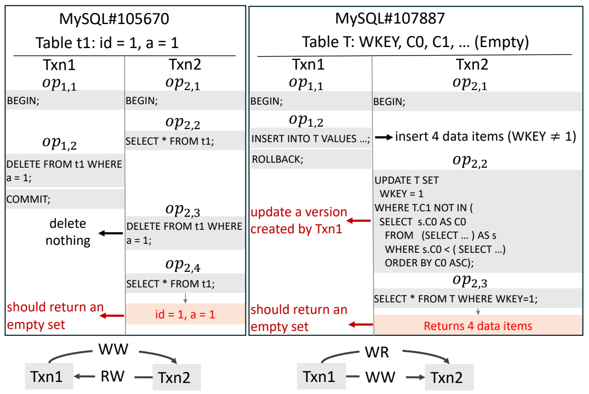
E.2 Adaptation of C-Reduce.

Since C-Reduce cannot be directly applied to our case reduction scenario, we first adapt it with two IL-specific rules. First, *operations in a transaction should be removed or retained together*. This is because operations within a transaction share the same execution context. For example, if an operation takes an incorrect snapshot, subsequent operations in the same transaction usually immediately read the wrong versions. Thus, we consider setting transaction as the smallest removal granularity. Second, *operations in a thread should be removed or retained together*. This is because not every thread always contains bug-related operations, and removing an entire bug-irrelevant thread could reduce all its operations at once. Thus, we consider setting thread as a large unit of operation removal.



(a) OCC-Based DBMSs (b) TO-Based DBMSs

Figure 2: Examples of Order Inference



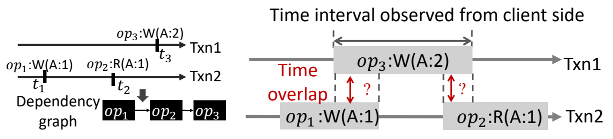
```
BEGIN;
DELETE FROM t1 WHERE a = 1;
COMMIT;
```

```
BEGIN;
SELECT * FROM t1;
DELETE FROM t1 WHERE a = 1;
SELECT * FROM t1;
```

```
BEGIN;
INSERT INTO T VALUES ...;
ROLLBACK;
```

```
BEGIN;
UPDATE T SET
  WKEY = 1
WHERE T.C1 NOT IN (
  SELECT  s.C0 AS C0
   FROM   (SELECT ... ) AS s
   WHERE s.C0 < ( SELECT ...)
   ORDER BY C0 ASC);
SELECT * FROM T WHERE WKEY=1;
```

Figure 6: Examples of MySQL#105670 and MySQL#114389



(a) Construction with Unique Timestamps (b) Construction with Time Intervals

Figure 3: Dependency Graph Construction

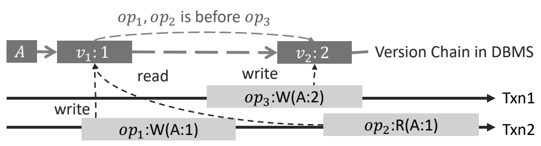


Figure 4: DBMS Instrumentation

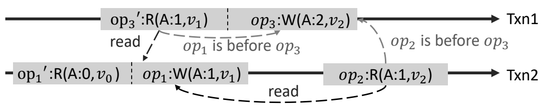


Figure 5: Constrained Test Case Generation

E.3 Adaptation of Oracle Database Replay

Native Oracle Database Replay. Oracle Database Replay [1] executes each operation through kernel callbacks. These callbacks provide critical metadata such as the unique operation identifier, read/write set, access type (e.g., read/write operation), and SCN (a monotonically increasing logical timestamp). With these metadata, it enables precise execution order inference and dependency graph construction. For example, Fig. 3a shows three operations op_1, op_2, op_3 that access the data item A with timestamps $t_1 < t_2 < t_3$. As such, we can infer that their exact execution order $op_1 \prec op_2 \prec op_3$. Then, a dependency graph can be constructed directly.

Unfortunately, this approach requires performing kernel callbacks (i.e., kernel instrumentation) to obtain some metadata inside the database, such as SCN. However, it is laborious and even impossible in production systems, especially for cloud-hosted or closed-source databases. As shown in Fig. 3b, when only client-side execution time intervals are observable, the overlapping time intervals would lead to ambiguity. Specifically, while we know that op_1 precedes op_2, the relative order between op_1&op_3 and op_2&op_3 remains unknown, making dependency inference impossible without any adaptations.

Adaptions of Oracle Database Replay. To build dependency graphs for these time-overlapping operations, Oracle Database Replay has to rely on additional conflict metadata. There are two common approaches to obtain the conflict metadata.

❶ *Extracting Conflict Metadata from the DBMSs.* This approach extracts internal conflict metadata from the database side, e.g., the version stamps in the version chains. These stamps enable precise ordering of time-overlapping operations by revealing the data version evolution and the sequence of versions accessed. For example, as shown in Fig. 4, op_1 writes A:1 (version v_1), while op_3 writes A:2 (version v_2). The version chain shows that v_1 precedes v_2, indicating that op_1 executes before op_3. Furthermore, since op_2 reads A:1 from v_1, it should execute before op_3. This yields dependency edges $op_1 \prec op_3$ and $op_2 \prec op_3$. However, extracting internal conflict metadata also requires instrumenting the DBMS kernel. In contrast, our *Pisco* directly uses client-side traces and concurrency control protocol to simulate DBMS's internal conflict metadata.

❷ *Extracting Conflict Metadata from the Workloads.* This approach imposes constraints for the test case generation, which aims to explicitly expose conflict metadata (e.g., version stamps) from the specific customized workloads side. To this end, each test case attaches a version stamp to each data item and requires each write operation to increment the version stamps of its written data items. Furthermore, to facilitate capturing the conflict metadata, each operation always explicitly fetches the version stamps of its accessed data items. Moreover, since a write operation cannot retrieve the version stamps of the data items to be written, it must be preceded by a read with the same predicate to fetch them. Then, the order of conflicting operations can be inferred from these version stamps. For example, before performing the write operations op_1 and op_3, their preceding reads op_1' and op_3' obtain the version stamps of the data items to be written. Since op_3' reads the version stamp v_1 created by op_1, we can infer that op_3 executes after op_1. Similarly, since op_2 reads the version stamp v_1 created by op_1, op_2 is before op_3. These inferences yield dependency edges $op_1 \prec op_3$ and $op_2 \prec op_3$. However, this approach relies on specific customized workloads, which makes it unable to support arbitrary workloads (i.e., many test cases are unsupported). In contrast, our *Pisco* does not have such a limitation.

F THE POTENTIAL DIVERGENCE OF THE REDUCED PATTERNS.

The bug case variants for the same bug might not be reduced to the same pattern. More specifically, many isolation bugs are triggered when some critical database data structures (e.g., version chains or lock tables) reach a specific state [1, 2]. For example, suppose there exists a bug in the source code that implements version chain management, which is triggered when the version chain reaches a specific state. Specifically, in this bug, a new data version has already been added to the version chain, but a subsequent operation ignores it, resulting in reading an incorrect old data version. However, this bug can be triggered by any test case that makes the version chain reach that specific state, regardless of the state of other data structures inside the database (e.g., lock table). For example, given a specific data item A, the state evolution of its version chain can be made by two operations with quite different operation semantics, such as a point update on A or a range update query involving A. As a result, a bug can be triggered by test cases with different application logic and different transaction semantics, leading to different dependency graph patterns.

Take MySQL#105670 and MySQL#107887 as an example (officially confirmed as the same bug), where a transaction reads a stale version that should be deleted. As shown in Fig. 6, each bug case involves two interleaved transactions txn_1 and txn_2. In MySQL#105670, the bug case starts with a data item, whose initial version is $v_1 = \langle id = 1, a = 1\rangle$. Then, $op_{2,2}$ sets the snapshot of txn_2, making v_1 visible to it. After that, txn_1 deletes this data item and appends a deletion marker to its version chain. Since this data item is marked as deleted, the following operation $op_{2,3}$ deletes nothing. However, the subsequent SELECT operation $op_{2,4}$ ignores the deletion marker and returns the stale version v_1.

In contrast, MySQL#107887 starts with an empty table T. Then, $op_{1,2}$ inserts 4 data items into T, all with WKEY$\neq$ 1. Next, txn_1 is rolled back and appends deletion markers to the version chains of the inserted data items. After that, txn_2 starts $op_{2,2}$, an UPDATE operation whose subquery contains a SELECT operation. This SELECT operation ignores the deletion markers and retrieves the 4 data items for $op_{2,2}$. As a result, $op_{2,2}$ updates these data items and appends the updated versions to their version chains. Finally, $op_{2,3}$ reads the updated versions, but the data items should not exist.

To summarize, MySQL#105670 and MySQL#107887 are both triggered by the following process. First, one or more data items are prepared (either pre-existing or newly inserted). Then, these data items are marked as deleted by txn_1, and the corresponding deletion markers are appended to their version chains. Finally, a subsequent SELECT operation in another concurrent transaction txn_2 incorrectly ignores the deletion markers and thus returns their stale versions. Note, this bug must be triggered by the SELECT statement since the DELETE statement in $op_{2,3}$ does not ignore the deletion markers.

However, MySQL#105670 and MySQL#107887 have quite different dependency graph patterns, as well as transaction semantics. As shown in Fig. 6, the dependency graph of MySQL#105670 contains a cycle between txn_1 and txn_2. The cycle consists of an RW dependency and a WW dependency. The RW dependency is from txn_2 to txn_1 as $op_{2,2}$ in txn_2 reads the data item before $op_{1,2}$ in txn_1 writes it. The WW dependency is from txn_1 to txn_2 as $op_{1,2}$ writes the data item before $op_{2,3}$ writes it. In contrast, the graph of MySQL#107887 consists of a WW dependency and a WR dependency. The WW dependency is from txn_1 to txn_2 as $op_{1,2}$ inserts the 4 data items before $op_{2,2}$ updates these data items. The WR dependency is also from txn_1 to txn_2 since $op_{1,2}$ inserts the 4 data items before the SELECT statement involved in $op_{2,2}$ reads these data items.

Since variants of the same bug might not be reduced to the same pattern, duplicate cases cannot be directly identified by their patterns.

Table 1: Reported Bugs

DBMS	TDSQL	TiDB	DBX	Total
Submitted	6	5	1	12
Confirmed	6	5	1	12
Longest Bug Confirmation Time(Day)	1	2	1	2
Max Rounds of Reporter-Developer Interaction	1	0	0	1

G USE CASE OF PISCO

In the past two years, *Pisco* has helped Leopard [39] to submit 12 unique bugs for TDSQL, TiDB, and a commercial database (anonymized as DBX). Specifically, we define the *bug confirmation time* as the duration from bug submission to official confirmation, and define the *rounds of reporter-developer interactions* as the number of times the developers request additional bug information from the bug reporter. We observe that most of our submitted bugs are confirmed within 2 days without interactions, as our submitted cases are minimal and could deterministically reproduce the bugs (details in our technique report [54]). We survey the transactional bugs in the MySQL community and find that it took an average of 23.9 days and 2.8 rounds of interactions to confirm the bug reports, while TDSQL [7]'s developers took 10.4 days and 9.8 rounds of interactions. In MySQL, we observe that the bug #98642 was reproduced by a 7-day duration after 13 rounds of interactions between the community and the reporter, providing two additional videos and a Docker image by the reporter.