Pisco: An Isolation Bug Case Reduction and Deduplication Framework

Siyang Weng*
East China Normal University
syweng@stu.ecnu.edu.cn

Hongyu Yang
East China Normal University
yanghy@stu.ecnu.edu.cn

Zirui Hu
East China Normal University
zrhu@stu.ecnu.edu.cn

Rong Zhang*
East China Normal University
rzhang@dase.ecnu.edu.cn

Zhicheng Pan
East China Normal University
zcpan@stu.ecnu.edu.cn

Chengcheng Yang†
East China Normal University
ccyang@dase.ecnu.edu.cn

Xuan Zhou
East China Normal University
xzhou@dase.ecnu.edu.cn

Yuxing Chen
Tencent Inc.
axingguchen@tencent.com

Xiaolong He
Tencent Inc.
jaxhe@tencent.com

Anqun Pan
Tencent Inc.
aaronpan@tencent.com

ABSTRACT

Practical implementations of Isolation levels (ILs) might deviate from their theoretical definitions, resulting in isolation bugs. Triggering isolation bugs usually requires concurrent test cases, which causes high debugging complexity. These test cases typically contain numerous irrelevant operations, resulting in large and redundant raw cases that violate the three essential criteria for bug report submission, i.e., *reproducibility*, *conciseness*, and *uniqueness*. Achieving these criteria involves three key challenges: 1) deterministically reproduce bugs despite high concurrency; 2) efficiently reduce cases without losing bug-triggering operations; 3) accurately distinguish unique bugs from the bug reports having complex transaction behaviors. To address these challenges, we propose an isolation bug case reduction and deduplication framework *Pisco*. First, we propose to simulate the DBMS's internal state to infer the order of conflicting operations for deterministic bug reproduction. Second, we introduce a dependency-aware divide-and-conquer strategy for efficient case reduction. Finally, we design a domain knowledge-driven, multi-agent collaboration framework for accurate bug deduplication. Extensive experiments show that *Pisco* reduces the cases to their minimal forms in just 20.0%/33.3% of the time required by *C-Reduce*/*DDMin* and has a deduplication ratio of up to 91.6%.

PVLDB Reference Format:
Siyang Weng, Hongyu Yang, Zirui Hu, Rong Zhang, Zhicheng Pan, Chengcheng Yang, Xuan Zhou, Yuxing Chen, Xiaolong He, and Anqun Pan. Pisco: An Isolation Bug Case Reduction and Deduplication Framework. PVLDB, 19(6): 1413 - 1426, 2026.
doi:10.14778/3797919.3797944

*Siyang Weng and Rong Zhang are also affiliated with Shanghai Engineering Research Center of Big Data Management.
†Chengcheng Yang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797944

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBHammer/Pisco.

1 INTRODUCTION

Isolation levels (ILs) act as a correctness contract for concurrent transactions [2]. To strike a balance between correctness and performance, DBMSs generally provide multiple ILs [14] that are implemented following various concurrency control protocols [21, 32]. However, the practical implementations might not strictly follow theoretical IL definitions [16] and then cause isolation bugs [19].

There exist many works focusing on detecting isolation bugs [3, 19, 28, 33, 41, 42, 69]. Generally, these studies first generate various test cases consisting of concurrent transactions and run them on the database. Then, they leverage a test oracle to verify whether the database correctly handles the cases. For clarity, we refer to the original test cases as *raw cases*. Due to the high throughput of DBMS, a raw case usually involves numerous concurrent operations. As shown in Fig. 1, the bug #42487 of TiDB [70] is triggered by the raw case containing 23,261 operations executed by 12 concurrent threads. Thus, the large number of operations in raw cases significantly increases the complexity of bug analysis and debugging. The primary reason is that isolation bug reproduction usually requires enumerating all possible execution orders of concurrent operations, due to the nondeterministic concurrency control protocols inside the DBMS and the complex access dependencies among operations.

However, the isolation bugs are typically caused by a few operations across 2~4 transactions [16], indicating that most operations in raw cases are *bug-irrelevant*. The presence of irrelevant operations makes manually processing such cases highly labor-intensive. Thus, database communities generally refuse to analyze unrefined raw cases and discourage their direct submissions [51]. An empirical study of transactional bug reports submitted between 2018~2022 across 6 widely-used DBMSs [16] reveals that successfully accepted bug reports should adhere to the following three criteria:

- *Reproducibility.* A bug report should specify an exact order of operations in the test case, where the bug can be deterministically reproduced (> 94% cases).

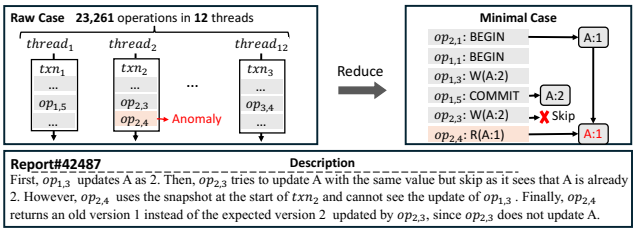


Figure 1: Bug Case of TiDB #42487 under Repeatable Read IL

- *Conciseness.* A bug report should consist of several small-scale transactions. Most bug cases have less than four transactions, containing 8 ∼ 10 operations in total (> 98% cases).
- *Uniqueness.* A bug report should present a new bug that has not been reported before. Detecting duplicate reported bug cases is time-consuming. For example, MySQL's official data shows an average of 114 days per report for duplication analysis [51].

The key challenges to meet the above criteria are as follows:

Deterministic Bug Reproduction (C1). Determining a feasible *server-side* execution order that could stably reproduce the isolation bug is the primary prerequisite for bug analysis. However, ordering large-scale concurrent operations for bug reproduction is non-trivial [47]. Specifically, the execution traces collected from the client side record execution time interval of each operation instead of its exact timestamp. Thus, the concurrent execution makes a large number of operations overlap in time. Numerous studies have been investigated to deduce the concurrent operation order. They can be broadly divided into two categories: *enumeration* and *dependency inference*. Enumeration-based methods exhaustively explore all possible operation orders within a bug case and replay them to detect whether the bug is triggered [39, 47, 65]. However, this approach suffers from the combinatorial explosion dilemma. Moreover, dependency inference-based methods infer the order by building dependency graphs from unique operation timestamps [38, 49], under the strict assumption that operations do not overlap in time.

Efficient Case Reduction (C2). To facilitate efficient bug reproduction and analysis, we should reduce a raw case to its minimal form. This requires eliminating all *bug-irrelevant* operations from the raw case. However, it is particularly challenging in the presence of complex operation dependencies. Previous studies [18, 36, 62, 75] usually use pre-defined rules to iteratively remove operations and check whether the removal would affect the bug triggering. For example, delta debugging [75] is a widely used method that evenly divides operations into multiple subsets. Then, it combines divide-and-conquer and trial-and-error strategies to recursively remove subsets. However, these methods overlook operation dependencies arising from conflicting accesses on the same data items. Consequently, they often mistakenly divide interdependent operations related to the bug across different subsets. This would make each subset unable to be safely removed without affecting the bug triggering, leading to inefficient trials. Then, these subsets must be recursively divided and checked, causing more trials.

Accurate Case Deduplication (C3). To avoid submitting a duplicate bug report to the database community, it is of great importance to identify whether there exists a reported bug having the same root cause (i.e., produced by the same faulty code) as the bug case to be submitted. Existing studies [78] generally leverage information retrieval or machine learning based report embedding techniques to address the issue of duplicate detection, with textual similarity as the dominant metric [18]. However, textual comparisons do not perform well in distinguishing different isolation bugs. This is because isolation bugs are closely related to the data access behaviors of operations in the bug case. Even if two bug reports reveal the same bug, their corresponding test cases might have different database schemas and transactions. Furthermore, isolation bug cases usually contain fewer than 10 SQL statements [16] and a high proportion of the SQL texts are reserved keywords (e.g., *SELECT*) [24, 27, 44, 72], both of which make the textual comparison rather hard to identify the duplicate bug reports.

To address these challenges, we propose an isolation bug case reduction and deduplication framework *Pisco*. The goal of *Pisco* is to generate a bug report with the properties of reproducibility, conciseness, and uniqueness. To address **C1**, we propose to use the execution trace to mirror the internal state (e.g., version chains, lock tables) of the DBMS, and then leverage the concurrency control protocol to deduce the execution order of conflicting operations with time overlaps. To address **C2**, we build a directed operation dependency graph to capture all dependencies between operations. Then, we rely on the data access dependency to form reduction units, where interdependent operations are aggregated in the same reduction unit. On this basis, we further design a dependency-aware divide-and-conquer strategy for case reduction. To address **C3**, we design a domain knowledge-driven multi-agent collaboration framework of large language models (LLMs) to deduplicate the bug case. To avoid excessive LLM invocations, we further design a coarse-grained filtering method based on IL-related features to select a small set of bug reports for fine-grained report ranking.

In summary, we make the following contributions.

- We design and open-source isolation bug case reduction and deduplication framework *Pisco*, which is available at [59].
- We propose an internal DBMS state mirroring based execution order inference method, a dependency-aware divide-and-conquer strategy, and a domain knowledge-driven multi-agent collaboration framework, which successfully address the challenges C1-C3 in reducing and deduplicating large-scale IL bug cases.
- We release a dataset of isolation bug cases [58], which can support future research on detecting and mitigating isolation bugs.
- Extensive experiments demonstrate that *Pisco* reduces bug cases to their minimal forms with only 20.0%/33.3% of the time compared to *C-Reduce*/*DDMin*. Moreover, it correctly identifies the duplicates for 91.6% bug cases; all unique bugs submitted with the assistance of *Pisco* were accepted within two days.

2 PRELIMINARY

2.1 Isolation Level

In DBMSs, an isolation level (IL) defines the degree to which operations in one transaction remain isolated from concurrent operations in other transactions. A stricter IL might reduce performance in exchange for stronger correctness guarantees. For example, a stricter IL imposes stricter serializability constraints in 2PL-based DBMSs, leading to more blocking among concurrent transactions. For the OCC-based DBMSs, a stricter IL requires more rigorous validation at commit time, leading to more transaction aborts. Weaker ILs relax constraints on accessible data versions to improve performance,

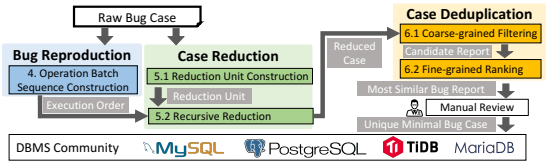


Figure 2: Architecture of *Pisco*

but accessing an incorrect version violates the IL's formal definition [1, 4], such as Dirty Reads, Observed Transaction Vanishes, and Non-repeatable Reads. Meanwhile, prior work has shown that more than 94% of transaction bugs can be deterministically triggered by executing operations in a specific order [16]. *Pisco* is designed for such deterministic isolation bugs. Note, multi-version concurrency control (MVCC) [32] and strict two-phase locking (S2PL) [21] are widely adopted by modern commercial DBMSs [35, 52, 55]. Thus, we introduce our technical concepts based on these protocols.

2.2 Isolation Bug Case

Existing IL testing methodologies consist of three phases: test case generation, test case execution, and execution history verification. First, the generation phase [19, 28, 42] aims to generate numerous test cases. Formally, a test case C is denoted as a set of transactions, i.e., $C = \{txn_1, txn_2, \cdots, txn_n\}$. Each transaction txn_i is an ordered list of operations, i.e., $txn_i = \{op_{i,1}, op_{i,2}, \cdots, op_{i,m}\}$, where $op_{i,j}$ is the j-th operation (a single SQL statement). The operations within a transaction are executed sequentially, whereas the operations in different transactions might execute concurrently, resulting in overlapping execution time intervals observed from the client side. Second, the execution phase [15] executes C and records the execution results as execution traces. The trace of an operation op is formally defined by op's ❶ start timestamp $op.t_s$ and end timestamp $op.t_e$; ❷ accessed data item $op.d$; ❸ operation type op_{type}; and ❹ returned results (whether the operation succeeds and the returned values of read operations). Since these information are easily collected from the client, *Pisco* can be integrated with common black-box transactional testing tools, e.g., Jepsen [28] and TxCheck [30]. Moreover, this phase produces an execution order $\prec$ over the operations in C. Finally, the verification phase [3, 33, 41, 69] verifies whether there exist isolation bugs. If an isolation bug is found, we use a boolean variable $\phi(C, \prec)$ to indicate whether replaying operations in C under the order $\prec$ could reproduce the isolation bug.

2.3 Bug Case Reduction and Deduplication

Generally, an isolation bug is triggered by a small number of operations in the bug case C_B, and the other operations are *irrelevant*. These irrelevant operations will increase the debugging complexity, making it difficult for developers to identify the root cause. Therefore, it is of great importance to remove the irrelevant operations, and this process is referred to as *bug case reduction*. Moreover, a bug case may share the same root cause with the submitted bug reports, i.e., produced by the same lines of faulty code that should be repaired [18, 29, 50], such as the same file or function. Submitting these duplicate bug cases will waste debugging effort. Thus, it is necessary to deduplicate bug cases before submitting them.

Definition 1. ***Isolation Bug Case Reduction and Deduplication**: Given a bug case C_B covering a set of operations involved in transactions, let $\mathbb{C} = \{C \mid C \subseteq C_B\}$ denote the power set of C_B, and $\mathbb{BR}$ denote the set of existing bug reports. The goal is to (1) find a minimal bug case $C_M \in \mathbb{C}$ that could reproduce the bug under a specific operation order $\prec_B$, i.e., $\phi(C_M, \prec_B) = true \wedge \nexists C' \subset C_M \, s.t. \exists \prec_{B'}, \phi(C', \prec_{B'}) = true$, and (2) check whether there exists a bug report $br \in \mathbb{BR}$ that describes the same bug as C_M.*

3 FRAMEWORK OVERVIEW

Fig. 2 shows the framework of *Pisco*, which has three components.

Bug Reproduction. The prerequisite for reducing a bug case is to identify a feasible operation execution order that could stably reproduce the bug. One naive approach is to enumerate all possible orders until finding one order that successfully reproduces the bug [10]. However, this would lead to a combinatorial explosion dilemma [47]. To address this issue, we propose to deduce the operation execution order based on their access conflicts. Specifically, isolation bugs are typically triggered when operations access incorrect data versions that violate the specific IL definition. Moreover, the specific data version accessed by an operation is determined by its relative order with respect to other operations writing to the same data, referred to as conflicting operations. Thus, if conflicting operations access data in the same order as in the raw case, we can ensure that the original incorrect version access behaviors would be reproduced. In light of this, we propose to deduce the execution order of conflicting operations by mirroring the internal state (e.g., version chains, lock tables) of DBMS. Then we can leverage the concurrency control protocol to simulate the schedule (e.g., lock acquisition and release, retrieve visible versions) of conflicting operations in the original bug-triggering scenario. After that, we further construct operation batch sequences by dividing conflicting operations into different batches and putting non-conflicting operations in the same batch. In this way, all the operations in a batch can be executed in parallel during the bug reproduction.

Case Reduction. Next, we come to eliminate bug-irrelevant operations. A common approach is delta debugging [75], which is a divide-and-conquer method that divides operations into subsets and iteratively removes subsets by checking whether the bug is still triggered after removal. However, this approach ignores operation dependencies and usually mistakenly divides interdependent operations related to the bug across multiple subsets. As a result, none of the subsets can be safely removed without affecting the bug triggering, which further leads to redundant trials and replay costs. To address this issue, we first build a directed *operation dependency graph* to depict dependencies between all the operations. On this basis, we can group interdependent operations into *reduction units*. Guided by these dependencies, we further apply a divide-and-conquer strategy that recursively reduces operations at the granularity of reduction units, thus improving efficiency.

Case Deduplication. After case reduction, we finally verify whether the bug has already been reported in the DBMS's community, so as to avoid unnecessary efforts of manual analysis. To this end, previous studies [18, 67] usually rank potentially duplicate bug reports in the community based on the textual similarity of report descriptions, and then pick the top-ranked ones for manual review. However, isolation bugs are closely tied to the data access behaviors of operations in bug cases, which are not easily revealed by the textual descriptions. To address this issue, we propose to leverage a domain knowledge-driven multi-agent collaboration framework of LLMs, which have demonstrated superior capabilities in DBMS issues [80]. Moreover, to reduce the cost of LLM invocations, we

first design a coarse-grained filtering method based on IL-related features [16] to select a handful of candidate bug reports. Then, we propose an LLM-enhanced fine-grained ranking method to accurately rank the candidates. Finally, the agents judge the duplication of the reduced case based on the top-ranked bug report.

4 BUG REPRODUCTION

In this work, isolation bugs can be reliably reproduced by executing operations in a specific *bug-triggering order*. Thus, we focus on how to infer such an order. Previous work, such as *Oracle Database Replay* [49], cannot determine the order of time-overlapping operations without unique timestamps. As shown in Fig. 3, while we know that op_1 precedes op_2, the relative order between op_1 and op_3 (or op_2 and op_3) remains unknown. Although this issue can be addressed by incorporating the additional *conflict metadata* (e.g., version stamps), such metadata must be extracted either from the DBMS kernel or specific customized workloads, restricting the generality. Note, an isolation bug is triggered when an operation accesses an incorrect data version. The version accessed by an operation is determined by its execution order relative to other write operations on the same data item, since they create versions of this data item. To capture the interferences between operations, we define two operations as conflicting if they access the same data and at least one of them is a write [1, 77]. If conflicting operations are executed in the original bug-triggering order, they would access the "expected" data versions and reproduce the bug [49]. In light of this, we propose to deduce the execution order of conflicting operations. Then, we further construct an *operation batch sequence* (see Algo. 1). Specifically, we divide conflicting operations into a sequence of batches and put non-conflicting operations in the same batch. In this way, all operations in a batch can be executed in parallel during the bug reproduction.

Specifically, we leverage the execution traces of operations (see § 2.2) to construct the batches. When constructing a new operation batch $\mathcal{B}$, we check operations that are not included in previous batches following the order of their *end timestamps*, which are arranged in an array O (line 3). Specifically, we fetch the next operation op that does not conflict with any operation in $\mathcal{B}$ (line 7). Then, we collect op and its time-overlapping operations into a set $\mathcal{S}_O$ (line 8), since we can directly derive the execution order of non-overlapped operations. Then, we deduce the execution order between the checking operation and its time-overlapping operations. Note, operations cannot be added to a batch if they conflict with operations already in the batch, i.e., conflicting with $\mathcal{B}$.

Order Inference for Time-overlapping Operations. To deduce the execution order of time-overlapping operations, we propose to simulate the state of DBMS's internal data structures related to the concurrency control, such as version chains and the lock table. This is because such state [30] before an operation's execution determines the result returned by this operation. A version chain $\mathcal{V}$ tracks the version evolution of each data item, where each write operation appends the new data version it creates to $\mathcal{V}$. Besides, to save memory usage, we periodically prune the obsolete versions that are invisible to all active transactions. Moreover, a lock table $\mathcal{L}$ tracks the lock contentions between different operations, while the lock is released after its holder's transaction ends. We next discuss the checking operation $w.r.t$ the operation type,

Algorithm 1: Operation Batch Sequence Construction

Input: Bug case C_B.
Output: Operation Batch sequence $\mathcal{S}$.
1 Batch sequence $\mathcal{S} \leftarrow \emptyset$, lock set $\mathcal{L} \leftarrow \emptyset$ and version chain $\mathcal{V} \leftarrow \{d : \emptyset | \forall d \in$ all accessed data items$\}$;
2 Hashmap $\mathcal{A}_C \leftarrow$ the total access number of each data item;
3 Array $O \leftarrow$ all operations in C_B, ordered by end timestamp;
4 **while** $\exists op \in C_B, op \notin \mathcal{S}$ **do**
5 $\mathcal{B}, \mathcal{A}_W \leftarrow \emptyset$;
6 **while** $O.hasNext() \wedge |\mathcal{A}_W| \neq |\mathcal{A}_C|$ **do**
7 $op \leftarrow O.nextNonConflict(\mathcal{B})$;
8 $\mathcal{S}_O \leftarrow \{op\} \cup \{op' | \forall op' \in O, op'$ overlaps with $op\}$;
9 **forall** *read operation* $op_r \in \mathcal{S}_O$ **do**
10 **if** $\nexists op'' \in \mathcal{B}, op_r$ *conflicts with* $op'' \wedge op_r$ *reads version* $\in \mathcal{V}$ **then**
11 update $\mathcal{A}_C$ with op_r, $B+ = op_r$;
12 **forall** *write operation* $op_w \in \mathcal{S}_O$ **do**
13 **if** $\nexists op'' \in \mathcal{B}, op_w$ *conflicts with* $op'' \wedge op_w.d \notin \mathcal{L} \wedge$
14 $\forall op' \in \mathcal{S}_O, op'$ *writes* $op_w.d, op_w.t_e \leq op'.t_e$ **then**
15 update $\mathcal{A}_C$, $\mathcal{A}_W$ with op_w, $B+ = op_w$;
16 O-=$\mathcal{B}$, $\mathcal{S}.add(\mathcal{B})$, Update $\mathcal{V}$, $\mathcal{L}$ with $op \in \mathcal{B}$;
17 **return** $\mathcal{S}$;

i.e., read and write. Note, if a read operation op_r is incorrectly ordered after a subsequent write operation op_w in the case, op_r would read the version created by op_w instead of the version it read in the original bug-triggering scenario. To prevent reads from being scheduled late, we propose to check all the read operations first among time-overlapping operations.

❶ 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 batches, the checking operation could read that version when it is executed in the new batch $\mathcal{B}$ (lines 9-11). Then, this operation can be added to $\mathcal{B}$. Otherwise, the required version has not been created, and the read operation should wait until a write operation creates it. As a result, op_r cannot be added to $\mathcal{B}$.

❷ If the checking operation is a write operation op_w, it can be executed in the new batch only if it could acquire the lock of its accessed data. It implies that a write can be added to $\mathcal{B}$ if it satisfies both of the following conditions (lines 12-15): 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, op_w can be added to $\mathcal{B}$ only if the lock for the data item it writes is not held by any operation in previous batches (i.e., $op_w.d \notin \mathcal{L}$). For example, consider the operation op_3 that writes to data item A in Fig. 3. As the operation op_1 holds the lock on A, op_3 must wait until the lock is released. ② *The write operation acquires the lock.* Database lock contention occurs when multiple write operations attempt to modify the same data. From Property 1, we know that the first returned write would always first acquire the lock, regardless of network delays (due to space constraints, proofs and complexity analysis are in [59]). Therefore, only the first returned write can be added to $\mathcal{B}$ (line 14).

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

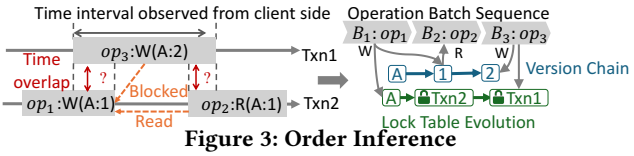


Figure 3: Order Inference

Termination Condition of Batch Construction. We now discuss when to "*freeze*" a batch, i.e., stop adding operations to it and start a new one. A batch is *frozen* if either of the following conditions is met: ① all remaining operations conflict with those already in the batch, or ② all operations in the test case have been processed. Condition ① is time-consuming for the exhaustive examining of all remaining operations. However, if the current batch has written to all data items accessed by the remaining operations, then every remaining operation must conflict with the batch. Naturally, we use a hashmap $\mathcal{A}_C$ to count the number of accesses each data item will have from all remaining operations, initializing each counter to the total accesses in the test case (line 2). When an operation is added to the batch, we decrement the counter of each data item it accesses, and remove a data item from $\mathcal{A}_C$ once its counter is zero (lines 11, 15). We also maintain a *written data set* $\mathcal{A}_W$ containing data items that (i) have been written by operations in the current batch, and (ii) will be accessed by remaining operations. When adding an operation, we add its written data items to $\mathcal{A}_W$ if they are still in $\mathcal{A}_C$ (line 15). Finally, we freeze the batch when the number of items in $\mathcal{A}_C$ equals the size of $\mathcal{A}_W$, indicating all remaining operations conflict with the current batch (line 6). When a batch $\mathcal{B}$ is freezed, we remove operations in $\mathcal{B}$ from $\mathcal{O}$, add $\mathcal{B}$ to the batch sequence $\mathcal{S}$, and update $\mathcal{L}$ and $\mathcal{V}$ using $\mathcal{B}$ (line 16). The complexity of this procedure is $O(N)$ for a case with N operations.

Example 1. In Fig. 3, we process operations in ascending order of their end timestamps, i.e., op_1, op_3, op_2. Since they only access the data item A, the hashmap $\mathcal{A}_C$ contains A:3. ① We first check op_1, overlapping in time with op_3. They are conflicting writes to A. Based on Property 1, the first returned conflicting write acquires the lock and is executed first. Thus, op_1 is put into the batch B_1, while updating the version chain of A by A:1 and adding the lock of A to the lock table. Meanwhile, A is added to the written data set $\mathcal{A}_W$ and its counter in $\mathcal{A}_C$ is decreased by 1. Since $|\mathcal{A}_W| = |\mathcal{A}_C|$, B_1 is frozen. ② Then, it comes to op_3. Since it overlaps with the read operation op_2, we first check op_2. Since op_2 reads the existing version A:1, it is directly added to the new batch B_2 and the counter of A in $\mathcal{A}_C$ is decreased by 1. As Txn2 still holds the lock on A, op_3 cannot be added to B_2. Since all operations are checked, B_2 is frozen. Now, Txn2 ends and releases the lock on A. ③ Since op_3 can obtain a lock on A, it is added to the new batch B_3. The lock table and the version chain are updated accordingly, and $|\mathcal{A}_C|$= $|\mathcal{A}_W|$=0. Finally, B_1, B_2 and B_3 form the operation batch sequence.

Adaptability. Regardless of the implementation to handle edge cases inside the database, *Pisco* can correctly simulate the DBMS's state evolution (including the version chain and lock table) and infer the bug-triggering order. This is because *Pisco* ensures a correct and closed loop between the simulation of DBMS's state evolution and the operation execution order inference. More importantly, this design can also be adapted to infer execution order for DBMSs implementing other concurrency control protocols (e.g., OCC and TO). More detailed discussions are put in [59].

Data Version Disambiguation. Note, we distinguish different versions of a data item according to its data values. However, a value might appear multiple times in the version chain and could not be used to identify the version accessed by the read operation. To address this issue, we propose to enumerate all possible batch sequences based on each matched version. Specifically, for each matched version, we construct an operation batch sequence under the assumption that the read operation accesses this version. If the bug can be triggered following this operation batch sequence, we identify this version as the accessed version. Note, the version chain retains only values visible to active transactions, which effectively reduces the versions to be enumerated and results in a low overhead.

5 CASE REDUCTION

Isolation bugs are usually related to a small size of operations ($\leq$ 10) [16]. Next, we need to remove bug-irrelevant operations for effective debugging. To this end, we first construct operations' reduction units to capture data access dependencies between operations (§ 5.1). Then, we propose a divide-and-conquer and trial-and-error strategy to recursively reduce reduction units (§ 5.2).

5.1 Reduction Unit Construction

Delta debugging [75] is a widely used method that takes the idea of divide-and-conquer and trial-and-error to reduce the bug case. The naive delta debugging method usually evenly divides operations, which neglects the operation dependencies in the test case. Thus, it may divide bug-relevant operations into different subsets. Since removing bug-related operations prevents the bug from being reproduced, these subsets cannot be directly removed. Thus, each subset requires to be recursively divided into smaller subsets. However, such recursive divisions would make the trial-and-error mechanism face a serious combinatorial explosion problem.

To address this issue, we introduce the *reduction unit* (RU) for each operation. An RU consists of a given operation together with all operations that depend on it. For a write operation, its RU comprises the write itself and all later operations that access the data item it updates, since subsequent operations either access the version created by this write operation, or create a new version based on this version. For a read operation, its RU consists only of that read, since it does not generate a new version of any data item.

To construct the RU for each operation, we define 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. Therefore, the RU of op can be constructed by traversing all of its "reachable" operations in $\mathcal{G}$. Note, we define an operation op' to be "reachable" from op if there exists a directed path from op to op'.

Besides, we observe that an operation's RU exactly equals the union of its direct successors' RUs. Thus, we propose a recursive RU construction method with a time complexity of $O(|E| + |V|)$. Specifically, the method starts with the operations with no predecessors. For each encountered operation op, it checks if op has direct successors. If so, we first recursively construct the RUs of its successors. Then, these RUs and op itself form the RU of op, denoted as $RU(op)$. If not, the RU of op simply consists of itself.

5.2 Recursive Operation Reduction

After constructing the RUs, interdependent operations would be checked together within the same RU when applying the delta

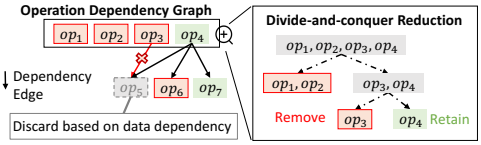


Figure 4: Example of Recursive Operation Reduction

Algorithm 2: Recursive Operation Reduction

Input: Operation Dependency Graph $\mathcal{G} = (V, E)$, Bug Case C_B
Output: Minimal Reduced Case C_M

1 $S \leftarrow \{ op \mid \forall op \in V, \nexists op', s.t., < op', op > \in E\}$;
2 **while** $S \neq \emptyset$ **do**
3 $\quad S_R, C_B \leftarrow$ `DivideConquer`(S, C_B);
4 $\quad S \leftarrow \{op \mid \forall op' \in S_R, \forall op \in C_B, \exists e =< op', op >\}$;// successors
5 **return** C_B
6 **Function** `DivideConquer`(S, C_B):
7 $\quad$ **if** $(C_B \setminus \cup_{op \in S} RU(op))$ *still triggers the bug* **then**
8 $\quad\quad$ **return** $\emptyset, C_B \setminus \cup_{op \in S} RU(op)$;// trial and error
9 $\quad$ **if** $|S| = 1$ **then**
10 $\quad\quad$ **return** S, C_B;
11 $\quad S_1, S_2 \leftarrow$ Divide S evenly;
12 $\quad R_1, C_1 \leftarrow$ `DivideConquer`(S_1, C_B);
13 $\quad R_2, C_2 \leftarrow$ `DivideConquer`(S_2, C_B);
14 $\quad$ **return** $R_1 \cup R_2, C_1 \cap C_2$

debugging method. Next, Algo. 2 introduces how to recursively reduce operations at the granularity of RUs. It takes the operation dependency graph $\mathcal{G}$ and the bug case C_B as the inputs. Then, we collect all operations without predecessors (line 1), and then apply the idea of divide-and-conquer to remove the bug-irrelevant RUs from their RUs (line 3). This is because ❶ each operation's RU contains all operations reachable from it, and the union of these RUs covers all operations in the bug case; ❷ these RUs are the largest units, and removing them can greatly reduce the case. After one round of the divide-and-conquer, only operations with bug-related RUs are retained with the reduced bug case C_B, which are collected as the bug reproduction set S_R (line 3). This is because they would influence the results of all other operations in their RUs, and are indispensable for bug reproduction. Next, we gather the collected operations' direct successors as S (line 4) for the next round. The above steps repeat recursively until all operations in S_R have no successors, leaving an empty S for the next round (line 2).

For each round (lines 6-14), it first validates whether C_B still triggers the bug after removing the RUs of all operations in S (line 7). If so, all these bug-irrelevant RUs are removed from C_B and then returned (line 8). Otherwise, at least one operation in S has a bug-related RU. In such a case, if S contains only one operation, this operation is retained and directly returned (line 10); otherwise, operations in S are evenly split into two disjoint subsets S_1 and S_2, which are recursively reduced by `DivideConquer` (lines 11-13). Then, each invocation of `DivideConquer` returns the bug-related operations and the reduced cases, i.e., $R_1 \& C_1$ and $R_2 \& C_2$. Finally, the union $R_1 \cup R_2$ collects all bug-related operations in each subset, and the intersection $C_1 \cap C_2$ retains bug-related operations in both recursive branches that are returned as outputs. After a round (line 3), Algo. 2 collects the direct successors of the retained operations in S_R, which form a new set S for the next round (line 4).

Example 2. Figure 4 shows an example of our recursive reduction with C_B={$op_1, \cdots op_7$}. First, we identify operations with no predecessors, i.e., S={op_1, op_2, op_3, op_4}. Then, we divide them into two groups, which are S_1={op_1, op_2} and S_2={op_3, op_4}. Next, we apply the trial-and-error strategy to S_1 and S_2. We observe that the RUs in S_1 are irrelevant to the bug and thus can be safely removed, i.e., C_B={$op_3, \cdots, op_7$}. Then, we divide the operations in S_2= {op_3, op_4} into two groups. After applying the trial-and-error validation to each group, we find that the RU of op_3 is irrelevant and can be removed safely, i.e., C_B={op_4, op_6, op_7}. Consequently, the retained operation is S_R= {op_4}. Then, we start a new round of the reduction process for the direct successors of op_4, i.e., S={op_5, op_6, op_7}. Note, op_5 is removed from C_B as part of a cascade triggered by the removal of op_3, and we continue to divide op_6 and op_7 into two groups. Finally, the RU of op_6 is found to be irrelevant to the bug and removed directly, i.e., C_B={op_4, op_7}. Since op_7 has no successor, the reduction terminates. In summary, C_B includes {op_4, op_7}.

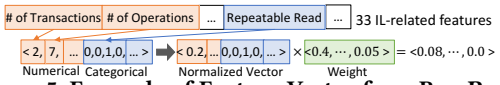


Figure 5: Example of Feature Vector for a Bug Report

Complexity Analysis and Discussion. The complexity of this procedure is $O(M \log \frac{N}{M})$ for a bug case with N operations and M bug-related operations in the worst case. Furthermore, *Pisco* holds a guarantee of 1-minimality [75]. Note, variants of the same bug might not be reduced to the same pattern, and duplicate cases cannot be directly identified by their patterns, complicating the case deduplication. More detailed discussions are put in [59].

6 CASE DEDUPLICATION

A bug case may share the same root cause with the existing bug reports [18, 29, 50]. Submitting such *duplicate* bug cases will waste debugging effort. Therefore, we focus on efficiently deduplicating the bug cases in this section. One naive method is to manually check each bug report, which is particularly laborious. To address this issue, we propose to search for the most likely duplicate candidate report for manual confirmation. Unfortunately, the text similarity-based method is not suitable for deduplicating isolation bug reports, due to the limited information in the short bug report as discussed in § 1. Considering the superior capabilities of LLMs in various DBMS tasks [6, 7, 43, 56, 60, 68, 80], we also choose LLM to handle this task. However, directly applying LLMs to this task is inefficient due to two reasons. First, leveraging LLMs to exhaustively compare the bug case with all existing bug reports will lead to high cost [22]. Second, LLMs lack specific IL knowledge, limiting the accuracy. Although *D-Bot* [80] incorporates DBMS expertise for system diagnosis, it is not designed for the deduplication task. Specifically, it neither encodes IL-specific semantics nor outputs a definitive binary decision required for determining whether two bug reports correspond to the same underlying issue. To address these issues, we first introduce a coarse-grained filtering method to narrow down the candidate reports (see § 6.1). Then, we propose a novel LLM-based multi-agent collaboration framework for fine-grained candidate ranking (see § 6.2). Finally, the agents deduplicate the reduced case based on the top-ranked bug report.

6.1 Coarse-grained Filtering

Previous study [16] has summarized 55 transactional features from numerous transactional bug reports. Among them, we employ 33 IL-related features, e.g., the number of transactions and the IL used in the case, to distinguish bug reports. As shown in Fig. 5, we apply one-hot encoding to the categorical feature, e.g., the 3-rd bit

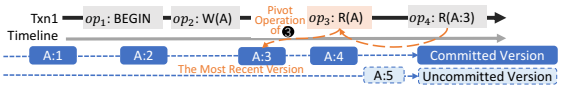


Figure 6: Example of Different Access Behaviors

represents *repeatable read* in this example. Besides, the numerical features, e.g., the number of transactions, are normalized to $[0, 1]$. Then, all 33 feature encodings are concatenated to represent a bug report. We further note that these features do not contribute equally. For instance, since many reports come from the same DBMS, the *DBMS Type* feature offers limited value. Conversely, the *SELECT FOR UPDATE* operation type is rare and thus more valuable. To measure such feature importance, we employ mutual information [71] to capture how much a feature helps distinguish bug reports.

More specifically, let F denote the features of a bug report, and f_i represent the i-th feature of F. Moreover, we use Y to represent the set of all the confirmed bugs in the community, and use y_j to denote the j-th bug in Y. Then, the importance (i.e., weight) of the feature f_i in distinguishing bug reports can be calculated as: $w_i = \sum_{y_j \in Y} p(f_i, y_j) \log\left(\frac{p(f_i,y_j)}{p(f_i)p(y_j)}\right)$, where $p(f_i)$, $p(y_j)$, $p(f_i, y_j)$ are the probability of feature f_i, the probability of bug y_j and the joint probability of f_i and y_j, respectively. Specifically, the ratio of $\frac{p(f_i,y_j)}{p(f_i)p(y_j)}$ represents the dependence between f_i and y_j. A higher ratio indicates that the feature is more related to the bug. We calculate probabilities based on frequencies in the existing bug reports by the counting-based approach [53]. For example, $p(f_i)$ is estimated as the ratio of the number of reports in which f_i appears to the total number of bug reports. Besides, we adopt *Laplace smoothing* [57] to handle sparse features, which captures the relationships between features and bugs unseen in our dataset. Note, we have collected isolation bug reports as extensively as possible [58], which helps a lot in mitigating the impact of sparse features.

By summing over all reports, the mutual information w_i represents the overall ability of feature f_i to distinguish bug reports and is therefore used as its weight. Then, they are used to calculate the cosine similarities between the reduced bug case and existing bug reports. Next, we select the top K bug reports by similarity as candidate bug reports for the subsequent LLM analysis.

6.2 Fine-grained Ranking

After obtaining the candidate duplicate bug reports, we use LLMs to rank them. To assist in identifying version accesses that violate the IL definition, we provide the LLM with the theoretically expected access behaviors of their operations (see § 6.2.1). To facilitate ranking, we decompose the ranking task into a series of pairwise comparisons by a multi-agent workflow (see § 6.2.2).

6.2.1 Expected Access Behavior Derivation. After investigating the official documentations of popular DBMSs, we summarize the expected access behavior (❶–❺) of each operation under different ILs as follows. Specifically, ❶ *The operation reads the snapshot created at its transaction's start time*, which is examplified by the IL of *snapshot isolation* (SI) in SQL Server. In Fig. 6, op_4 would access A:1. ❷ *The operation reads the snapshot created before its transaction's first non-transaction-control operation*, which is examplified by the ILs of *serializable* (SR) and *repeatable read* (RR) in PostgreSQL. In Fig. 6, op_4 would access A:2. ❸ *The operation reads the snapshot created before its transaction's first read operation*, which is examplified by the read operations under the ILs of SR and RR in InnoDB. In Fig. 6, op_4 would access A:3. ❹ *The operation reads the most recent committed version before it*, which is widely used by the IL of *read committed* (RC). In Fig. 6, op_4 would access A:4. ❺ *The operation reads the most recent version before it no matter whether the version is committed*, which is examplified by the IL of *read uncommitted* (RU) in SQL Server. In Fig. 6, the operation op_4 would access A:5.

Generally speaking, given an operation op that is included in the transaction txn, the specific version it should access is the most recent committed (or uncommitted) version before its *pivot operation* in txn. Here, we determine the *pivot operation* by the IL used in the database [48, 52, 55]. In Fig. 6, under the IL of SR in InnoDB, the *pivot operation* of op is the first *read* operation in txn, i.e., op_3. To capture the most recent version before the *pivot operation*, we replay all operations in the inferred order from § 4 and trace the version evolution during replay. We construct two version chains for each data item to record committed and uncommitted versions, respectively. A version is added to the uncommitted (*resp.* committed) version chain when it is created (*resp.* commmitted), together with the written (*resp.* committed) timestamps. In this way, we can derive operations' *expected access behaviors* by finding their *pivot operations* and traversing the version chains.

6.2.2 LLM Enhanced Candidate Report Ranking. As shown in Fig. 7, we employ an LLM-based multi-agent collaboration framework to rank the candidate reports, which includes three parts.

Domain-Specific Expert Agents Preparation. There exist various ILs in DBMSs, and even for the same IL, there might exist some subtle differences in different DBMSs. Fortunately, a previous study [41] has abstracted the implementation of various ILs into four classic mechanisms, including *consistent read* (CR), *mutual exclusion* (ME), *first updater wins* (FUW), and *serialization certifier* (SC). To comprehensively incorporates all these implementation knowledge [37], we propose four domain expert agents to perform the pairwise bug reports comparison collaboratively. Each agent focuses on one mechanism that implements the IL related to the bug case, prompted by the following three mechanism-specific contexts:

❶ *Textual description.* It is the semantic definition of the isolation level mechanism that is extracted from the DBMS document. Take the *consistent read* (CR) mechanism as an example, its textual description specifies "The transaction-level CR sees the snapshot of a database as of the beginning of a transaction, while the statement-level CR sees the snapshot as of the beginning of an operation".

❷ *Running example.* It provides a transactional trace demonstrating the behaviors of concurrent transactions. The example in Fig. 7 demonstrates the behaviors of transaction-level CR. It contains three transactions txn_1, txn_2, and txn_3 that operate on the same data item A. In this example, the BEGIN operation of txn_2 ($op_{2,1}$) sees a snapshot of the database with A:1. Then, the following operation $op_{2,2}$ within txn_2 also reads this snapshot. That is, even if txn_3 updates A:2 before $op_{2,2}$, $op_{2,2}$ still reads the value of A:1.

❸ *Pseudo-code.* It gives a concise pseudo-code of the isolation level mechanism. The code shows the pseudo-code of transaction-level CR. The read operation in a transaction scans the version chain for the accessed data item and returns the latest committed version visible to the transaction. The visible version is determined based on the timestamp of the transaction's BEGIN operation.

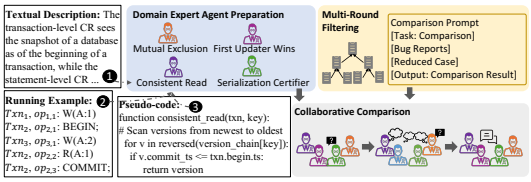


Figure 7: Workflow of LLM Ranking

Multi-Round Filtering. We next leverage expert agents to filter candidate bug reports in multiple rounds. In each round, the reports are randomly divided into pairs, and for each pair, the expert agents determine which report is more similar to the reduced bug case. To mitigate the potential hallucination issue [26], we adopt a multi-round comparison strategy. It involves comparing the target bug case against a pair of bug reports multiple times and determining the final decision through majority voting. For example, suppose the expert agents return an incorrect result with probability p, then the probability of mistakenly filtering a more similar report can be denoted as $p_e = \sum_{i=\lceil k/2 \rceil}^{k} C_k^i p^i (1-p)^{k-i}$ after repeating k (k is usually an odd number) comparisons. This indicates that p_e decreases exponentially as k increases. Besides, the predictions of LLMs are sensitive to the permutations within the inputs [46, 76]. Then *Pisco* incorporates a random permutation method, which permutes the order of two candidate bug reports and the textual contents within each report. At the end of each round, the selected report from each pair is retained for the next round, while the others are discarded. This process continues until only one bug report remains, which is identified as most similar to the reduced bug case. Next, we introduce the comparison details.

Collaborative Comparison. Given a pair of candidate bug reports, each expert agent first selects the report that is more similar to the reduced bug case based on its mechanism-specific knowledge. For each candidate bug report, we provide both its textual description and the associated bug-triggering case. For the reduced case, we provide the database schema, the number of threads and transactions, and the sequence of operations with annotated execution order SQL statements, and the corresponding threads and transactions. For each read operation, we specify its actual and expected read versions (i.e., its *expected access behavior*), enabling the agent to deduce which operation violates the IL.

However, when an agent specializes in a mechanism unrelated to the bug involved in the comparison, its judgment is less trustworthy. If we apply a simple majority voting as taken by the standard ensemble prompting [40, 66, 74], such irrelevant agents might dominate the voting and produce an incorrect conclusion. To address this issue, we require each expert to attach a confidence score to its comparison. If all agents return the same result with high confidence, then they reach a consensus. In contrast, if the agents return conflicting results with similar confidence scores, they fail to reach a consensus. This is possibly because the agent might ignore the key details about the bug case with a single step of reasoning. To resolve such disagreements, *Pisco* proposes a stepwise verification with Chain-of-Thought (CoT) prompting [73]. Concretely, each agent unfolds its one-step reasoning about the bug case execution process into a step-by-step analysis using CoT prompting. In these steps, the agent deduces the execution details of each operation, including its execution result and the resulting DBMS's internal state. If the disagreement still exists, it indicates that the bug might span across multiple mechanisms, hindering a single expert agent from reasoning comprehensively. To address this issue, *Pisco* adopts the simultaneous-talk paradigm [5], enabling expert agents to exchange their intermediate reasoning traces throughout multi-round discussions. In each round, each agent independently provides its conclusion and reasons, sharing them through a shared memory pool [23]. Next, each agent validates and revises its conclusion by referring to the shared context. Finally, the discussion is iteratively performed until all agents reach a consensus. Suppose there exist K candidate bug reports, and each report pair is repeatedly compared k times, the total number of LLM enhanced comparisons is $O(k \cdot K)$.

Finally, it uses a similar pipeline as the fine-grained ranking to make a duplication judgment. The only difference is that we prompt agents to judge whether the reduced case is a duplicate of the top-ranked bug report.

Table 1: Raw Bug Case Dataset and Statistics

Bug ID	Issue ID	#Cases	Case Size (#Operations)		
			Min	Avg	Max
B1	MariaDB#19535	117	15787	18211.4	24411
B2	MariaDB#26642	44	7558	12511.3	17085
B3	MySQL#100328	21	6377	13967.7	24288
B4	MySQL#113228	33	7519	19303.8	23737
B5	TiDB#20535	22	8279	19472.4	23750
B6	TiDB#21218	22	9076	19499.2	23711
B7	TiDB#21506	22	9395	19535.4	23653
B8	TiDB#28212	22	5015	17899.2	19635
B9	TiDB#36718	124	11610	19222.2	22588
B10	TiDB#21447	116	6161	20054.2	24737
B11	TiDB#42487	83	5718	20987.4	24275
B12	TiDB#44303	22	6035	18995.5	24506
B13	TiDB#44379	22	7080	19323.2	24333
Total	–	**670**	**5015**	**19602.7**	**24737**

Integrating *Pisco* with Jepsen. To demonstrate *Pisco*'s generalizability, we illustrate how *Pisco* can be integrated with Jepsen. Jepsen is a widely used black-box testing framework for detecting transactional isolation anomalies. During testing, its test client already records all information required by *Pisco* except the start and end timestamps of each operation. To this end, we can simply extend the Jepsen client to record a client-side timestamp whenever an operation is sent or completed. In this way, the operation-level traces from Jepsen can be directly consumed by *Pisco*.

7 EVALUATION

7.1 Setup

Our experiments are conducted on a server equipped with 2 Intel Xeon Gold 6126 processors, 250 GB memory, a 450 GB SSD, and an NVIDIA RTX 6000 Ada GPU. By default, the number of retained candidate bug reports is 16 (i.e., K=16) in the coarse-grained filtering phase, and each candidate bug report pair is compared 5 times (i.e., k=5) in the fine-grained filtering phase with the Qwen 2.5-7B, which is a popular and open-source model that could run on our GPU.

7.1.1 Baselines. Since there exist no studies addressing all three procedures, we evaluate each procedure of *Pisco* independently.

For bug reproduction, we compare *Pisco* against *Random* [42] and *Sequential* [19]. Specifically, *Random* does not control the execution order of operations between threads, but forces the order within a thread. The *Sequential* method executes all operations serially as the original bug-triggering execution (inferred in § 4).

For case reduction, we compare *Pisco* with *DDMin* [75] and *C-Reduce* [62], as they represent different reduction methodologies. That is, *DDMin* represents the general divide-and-conquer methodology and *C-Reduce* represents the rule-based methodology. Since

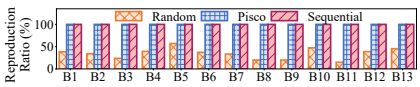


Figure 8: Reproduction Ratio

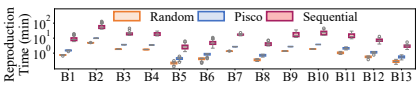


Figure 9: Reproduction Time

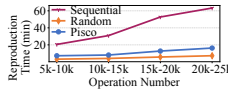


Figure 10: Bug Reprod.

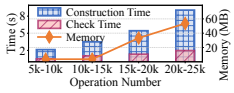


Figure 11: Batch Const.

C-Reduce cannot be directly applied to our case reduction scenario, we adapt it with two IL-specific rules. That is, removing operations at the granularity of transactions and threads.

For case deduplication, we compare *Pisco* against five baselines, which are *REP* [67], *SABD* [64], *D-Bot* [80], manual deduplication (*Manual*) and a native LLM-based deduplication (*LLM-N*). Specifically, *REP* measures the similarity by an improved BM25 [63] based on the textual description and categorical features in the bug report (e.g., bug type and DB version). *SABD* employs neural network-encoded text vectors to calculate similarities. *Manual* relies on the manual check of expert developers. *D-Bot* [80] is a multi-agent database diagnosis framework, and we replace D-Bot's general-purpose agents with IL-specific ones. As *D-Bot* might retain conflicting conclusions from different agents to prevent missing root causes, we obtain its final conclusion by majority voting. *LLM-N* directly prompts the LLM to identify the duplicated report.

7.1.2 Dataset Collection and Statistics. We adapt the state-of-the-art transactional workload generator *DBStorm* [42] to construct the test cases for three widely used DBMSs, which are MySQL, TiDB, and MariaDB. Then, the transactional bug verifier *Leopard* [41] is used to detect isolation bugs from these test cases. After running for one month, we obtain 670 bug cases, which are manually classified as 13 unique bugs by three expert developers as in Table 1. Since operations executed after the bug occurrence are irrelevant to the bug, we exclude all these operations from each bug case. In addition, to construct the bug repository for deduplication, we have collected a comprehensive set of transactional bug reports from the year 2018 to 2025 based on recent studies [16, 30] and the official report repositories of the DBMSs tested in our experiments. Finally, we find that all 670 generated bug cases are duplicates in our bug repository. All reports are open-sourced for future research work [58].

7.2 Evaluation of Bug Reproduction

7.2.1 Comparison of Bug Reproduction Methods. We report the bug reproduction ratio and runtime. Specifically, reproduction ratio measures the ratio that the bugs are successfully reproduced.

Reproduction Effectiveness. Figure 8 shows the average reproduction ratio of each bug. The *Random* method has a much lower reproduction ratio than other methods, which varies from 15% to 37% under all cases. It is because it does not consider interferences between conflicting operations, and the bug is reproduced only when the bug-related conflicting operations happen to have the same execution order as the original bug-triggering execution. In contrast, both *Pisco* and *Sequential* ensure 100% reproduction ratio by enforcing a correct execution order for conflicting operations.

Reproduction Efficiency. Figure 9 shows that *Random* reproduce the cases fastest due to its unconstrained parallel execution, whereas *Pisco* is slightly slower than *Random*. The main reason is that *Pisco* executes operations in a batch in parallel, but forces operation batches to execute sequentially to avoid indeterministic interferences between conflicting operations. In contrast, *Sequential* executes all operations sequentially, resulting in the poorest performance. Notably, *Pisco* has the same reproduction ratio as *Sequential* while being up to 14.6× faster.

7.2.2 Bug Reproduction Scalability. We divide the 670 test cases into four groups according to the number of operations (i.e., case size), which are $5k$~$10k$, $10k$~$15k$, $15k$~$20k$, and $20k$~$25k$.

Bug Reproduction. As shown in Fig. 10, *Sequential* has the steepest time growth when the case size is increased, and it reproduces the largest cases for more than 60 minutes. This is due to its full operation serialization. *Pisco* scales well with the speedup of intra-batch parallelism. *Random* remains the most time-efficient, but at the expense of a low reproduction ratio. Note, the reproduction time of *Sequential* is only 9× slower than that of *Random* and *Pisco*. This is because most isolation bugs are triggered under high contention, limiting the number of non-conflicting operations in each batch of *Pisco*. Even for *Random*, the conflicting operations are still serialized implicitly by the database's locking mechanism, decreasing its parallel execution performance in practice.

Batch Construction. To evaluate the scalability of our *operation batch sequence construction* method, we measure the average memory usage and construction time under varied case sizes. Besides, we display the time to check if overlapping operations can be added into the same batch (*check time*). Fig. 11 shows that the construction time increases linearly with the case size, consistent with the time complexity analyzed in § 4. Moreover, the check time also increases linearly with the case size, as our efficient mirror of the DBMS's internal state ensures that each operation is checked in constant time. Lastly, we observe that the memory consumption stays steady at first and then grows linearly with increasing case sizes. The reason is that our Java-based implementation first pre-allocates fixed-sized memory, and then the memory expands linearly when the pre-allocated memory is exhausted.

7.3 Evaluation of Case Reduction

7.3.1 Comparison of Case Reduction Methods. We compare *Pisco*'s case reduction method with *C-Reduce* and *DDMin* baselines.

Reduction Effectiveness. We observe that *C-Reduce* and *DDMin* can only reduce 53% and 70% cases to their minimal sizes on average as shown in Fig. 12. By contrast, *Pisco* guarantees that all the cases can be reduced to their minimal sizes. As shown in Fig. 13, *C-Reduce* and *DDMin* retain 19.2× and 2.9× more operations than *Pisco* on average. This is because they ignore the execution order among concurrent operations and may fail to reproduce the bug even after removing bug-irrelevant operations. This would further mistakenly recognize these operations as bug-relevant that should be retained. The reduced case size of *C-Reduce* is much larger than *DDMin* and *Pisco*. This is because it reduces operations in the same transaction or thread, making it hard to remove bug-irrelevant operations mixed with bug-related ones within the same scope.

Reduction Efficiency. We observe that *Pisco* takes the shortest time to reduce bug cases, consuming 20.0% of *C-Reduce*'s and 33.3% of *DDMin*'s average time in Fig.14. The reason is that our reduction-unit-based approach enables direct removal of bug-irrelevant subsets. To further analyze the reduction process, we track the size change of the largest bug case in dataset. We observe that *Pisco* reduces the case size much faster than the others in Fig. 16. It is

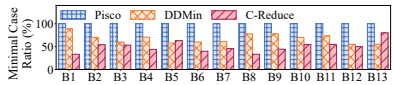


Figure 12: Case Reduction Effectiveness

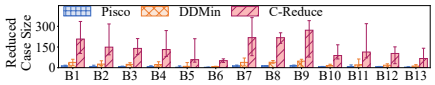


Figure 13: Reduced Case Sizes

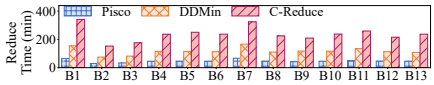
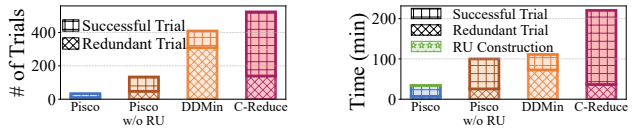


Figure 14: Case Reduction Efficiency



(a) Reduction Trials **(b) Reduction Time**

Figure 15: Breakdown of Trial Number and Reduction Time

because *C-Reduce* and *DDMin* overlook the dependencies among bug-related operations. Then they split bug-related operations into separate subsets and take more trials to remove them.

Ablation Study. To further analyze the reduction process, we break down the time and number of trials for both successful and redundant (i.e., failed) trials of different methods. Besides, we report the performance of *Pisco* when reduction units are disabled (*Pisco w/o RU*). Since *Pisco w/o RU* reduces bug cases at the same granularity as *DDMin*, it can be considered as *DDMin* equipped with a deterministic bug reproduction.

In Fig. 15, we observe that *Pisco w/o RU*, *DDMin*, and *C-Reduce* take about 4.5×/3.7×/3.9×, 29.4×/4.3×/12.2×, and 13.3×/16.7×/15.6× redundant/successful/total trials of *Pisco*. This is because *DDMin* and *Pisco w/o RU* often divide bug-related operations into different subsets, resulting in more divisions and trials for these subsets. Besides, since *DDMin* might replay bug-related conflicting operations in an incorrect order, it fails to trigger the bug even if the removed subset is bug-irrelevant. These trials are incorrectly treated as redundant, leading to much more divisions and trials. Finally, the additional redundant trials retain bug-irrelevant operations, causing reduced bug cases larger than *Pisco*. *C-Reduce* has less redundant trials than *DDMin*, because it removes fewer operations in each trial due to its small removing granularity, but takes more successful trials to remove bug-irrelevant operations.

Although *DDMin* and *Pisco w/o RU* require much more trials than *Pisco*, their runtimes are 3× and 2.1× larger, respectively. This is because most of the extra trials for them are performed after the case is progressively reduced and divided into many small subsets. As the case is continuously reduced, the latter trials take less validation time than the early reduction stages, i.e., contribute insignificantly to the total runtime. Moreover, the gap between *DDMin* and *Pisco w/o RU* is also small. This is because *DDMin* replays bug cases with high parallelism regardless of operation dependencies. *C-Reduce* is 1.7× slower than *DDMin* due to its more trials. Since *Pisco w/o RU* replays the case in a bug-triggering order, it can remove all irrelevant operations and obtain the minimal forms as *Pisco*.

7.3.2 Case Reduction Scalability. To evaluate the scalability of case reduction, we measure the average reduction time of different methods under varied case sizes. Moreover, we break down the total case

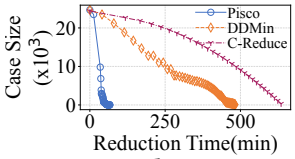


Figure 16: Reduction Proc.

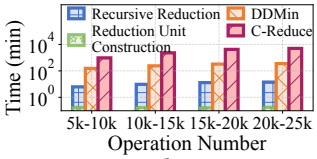


Figure 17: Reduction Time

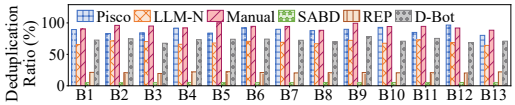


Figure 18: Deduplication Ratio

reduction time of *Pisco* into the reduction unit construction time and recursive reduction time. Figure 17 shows that *Pisco* always achieves the best performance and improves existing methods by up to 14.1×. In addition, both of the two components increase linearly with the case size. This is because *Pisco*'s divide-and-conquer strategy could always halve the set of reduction units at each iteration, and the total reduction time is dominated by the early iterations with larger case sizes. Besides, we construct all reduction units by a single scan on the dependency graph, with the time cost linear with the case size and its in-memory construction is significantly faster than the recursive reduction procedure that contains case reproductions, making its time cost negligible.

7.4 Evaluation of Case Deduplication

7.4.1 Comparison of Case Deduplication Methods. We compare *Pisco*'s case deduplication method with *REP* [67], *SABD* [64], *LLM-N*, *D-Bot* [80] and *Manual* baselines. Specifically, we propose *deduplication ratio* (*DR*) to measure the ratio that duplicate bug reports are correctly returned and bug cases are identified as duplicates. Note, by default, all bug cases in our dataset are duplicates which have been in our bug repository. Thus, *DR* also represents the recall in this scenario. Moreover, we report the financial cost of each method, including: (1) server cost, estimated by the rental price of instance on Tencent Cloud whose configuration is the same as the server configuration used in our experiment [13]; (2) LLM invocation cost for *Pisco* and *LLM-N*, based on Qwen2.5's token pricing as of July 2025 [12]; and (3) labor cost for the manual method, estimated using the 2024 U.S. median hourly wage of testers [54].

Deduplication Effectiveness. For the *DR* presented in Fig. 18, we observe that *Pisco* achieves the highest *DR* among all automatic approaches (91.6%), only approximately 6% lower than the manual deduplication (97.3%). In contrast, *REP* and *SABD* exhibit significantly lower ratios of 21.7% and 6.0%, respectively. This is because they rely solely on textual or semantic similarity, ignoring the underlying access behaviors that are critical to understand isolation bugs. *REP* improves recall by incorporating categorical features (e.g., bug type, DB version), but these categorical features still fail to precisely describe the root causes. Compared to other LLM methods, *Pisco* has a higher *DR* (about 20% higher than *D-Bot*). This benefits from *Pisco*'s collaborative comparison, preventing bug-irrelevant agents from dominating the final conclusion. In contrast, *D-Bot* resolves conflicting conclusions from its agents by majority voting. If a bug is only related to a few mechanisms, bug-irrelevant agents might dominate the voting and produce an incorrect conclusion.

Deduplication Efficiency. Figures 19 and 20 show the time and financial cost of different methods. *SABD* and *REP* achieve superior deduplication performance and low financial cost, but their *DR*s are quite low. *Manual* incurs substantially higher costs (41.6× time and 379.5× money) than *Pisco*. Besides, *LLM-N* and *D-Bot* cost much

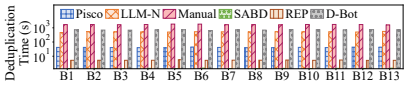


Figure 19: Time Cost of Deduplication

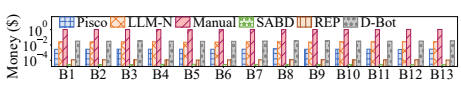


Figure 20: Financial Cost of Deduplication

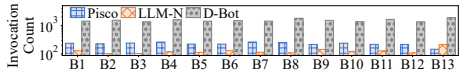
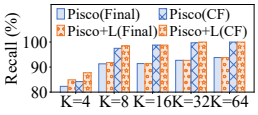
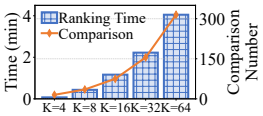


Figure 21: Invocations of Deduplication

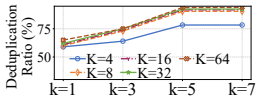


(a) Recall

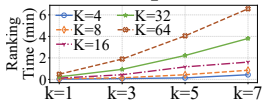


(b) Ranking Time

Figure 22: Influence of K in Coarse-grained Filtering



(a) Success Ratio



(b) Ranking Time

Figure 23: Influence of k in Repeated Comparison

Table 2: Ablation Studies of Deduplication Components

	Pisco	Pisco w/o CF	Pisco w/o EAB	Pisco w/o MA	Pisco w/o HM
DR (%)	91.6	94.0	78.6	69.0	60.3
Invoc.	234.2	1279.9	257.3	69.8	57.7
Money (¢)	0.6	4.0	0.2	0.2	0.1
Dedup. Time(s)	40.9	278.1	31.1	29.3	6.8

Table 3: Unique Case Ident.

Methods	Pisco	LLM-N	D-Bot	Manual
Acc. (U) (%)	98.2	72.4	76.6	100.0
Invoc.	226.4	125.6	1518.7	-
Money (¢)	0.6	3.8	6.4	14.9
Time(s)	43.2	511.5	712.6	1654.9

Table 4: Comparing LLMs

LLMs	DR(%)	Invoc.	Money(¢)	Time(s)
Deepseek-v3:671b	95.3	326.6	19.3	318.2
QWen3:235b	94.7	201.1	13.2	343.7
Llama3.1:405b	94.4	281.4	9.2	332.9
QWen2.5:7b	91.6	234.2	0.6	40.9

more time and money than *Pisco* , as *Pisco* has relatively small inputs from the coarse-grained filtering, while the other two take all reports in the repository as inputs and analyze them one by one.

In Fig. 21, *Pisco* invokes the LLM more than *LLM-N*, since *Pisco* invokes the LLM multiple times to analyze a bug report step-by-step, whereas LLM-N with a single LLM invocation. Note, each LLM invocation in *Pisco* analyzes part of the bug case, which is cheaper than that of LLM-N. *D-Bot* has the most LLM invocations, because it not only analyzes all bug reports in the repository one by one, but also invokes the LLM multiple times for each bug report.

7.4.2 Ablation Study. To evaluate the key components in *Pisco*'s deduplication pipeline, we conduct experiments by disabling the coarse-grained filtering (*Pisco w/o CF*), the expected access behavior derivation (*Pisco w/o EAB*), and the multi-agent workflow of collaborative comparison (*Pisco w/o MA*), respectively. Besides, we also evaluate the performance when disabling LLM hallucination mitigating techniques (*Pisco w/o HM*), including *multi-round majority voting* and *input order permutation* in the filtering phase, and *stepwise verification with CoT prompting* and *cross-agent knowledge sharing* in the collaborative comparison phase. Table 2 shows that *Pisco* takes about 234 LLM invocations to deduplicate a bug report, which incurs a modest cost in money and time. Besides, disabling the coarse-grained filtering causes a higher cost, while increasing the *DR* by only 2.4%. This is because *Pisco w/o CF* provides more candidate bug reports to the LLM. *Pisco w/o EAB* has less knowledge for LLM to understand the bug, decreasing *DR* by 13%. Meanwhile, since it takes less knowledge as inputs, its time and financial cost also decrease. However, its invocation count increases since the agents require more discussions to understand the buggy access behaviors. Besides, disabling the multi-agent workflow decreases the LLM invocation count, leading to lower time cost and financial cost, as well as a low *DR*. In addition, *Pisco* has an obvious 31.3% improvement in *DR* against *Pisco w/o HM*. This indicates that our proposed techniques mitigate LLM hallucinations effectively.

We also compare *Pisco* with its variant (denoted as *Pisco+L*) , which applies *Laplace smoothing* during coarse-grained filtering. Specifically, we report the recall of different methods in the coarse-grained filtering (*Pisco (CF)*) phase. Moreover, we also report the final recall (*Pisco (Final)*) when the coarse-grained filtering phase returns K candidate bug reports, i.e., *DR*. As shown in Fig. 22a, we observe that *Laplace smoothing* improves the recall of coarse-grained filtering when $K \leq 8$, but the improvement is not significant when $K \geq 16$. This is because our extensively collected isolation bug reports help a lot in mitigating the impact of sparse features.

7.4.3 Effect of Retained Candidate Report Number K and Repeated Comparison Number k. To evaluate the effect of K for the coarse-grained filtering, we report the recalls of *Pisco (CF)* and the *Pisco (Final)*. In Fig. 22a, the recall of both *Pisco (CF)* and *Pisco (Final)* increases with K. This is because a smaller K is more likely to filter out the candidate duplicate bug report during the coarse-grained filtering phase. Additionally, Fig. 22b shows that both the ranking time and number of comparisons grow with K. This is because the more candidate reports are retained, the more comparisons and LLM invocations in the fine-grained ranking phase. In Fig. 23a, we observe that increasing k improves DR when k is small ($k \leq 5$). This is because most noisy comparisons have already been corrected after several rounds of repetitions, further increasing k brings limited benefit. Meanwhile, Fig. 23b illustrates that the ranking time increases with k due to more comparisons. Since K=16 and k=5 perform best in most cases, they are our default settings.

7.4.4 Effect of Unique Case Identification. To demonstrate the ability of identifying unique bugs, we construct a unique-case dataset by removing bug reports associated with the tested bug cases. We compare *Pisco*'s *accuracy of unique* (Acc.(U)) and cost with those of a naive LLM-based method (*LLM-N*), *D-Bot* and human deduplication (*Manual*) in Table 3. Acc.(U) measures the ratio that unique bug cases are correctly identified as unique. We observe that identifying unique bugs is easier than duplicate ones for LLM-based methods. That is, Acc.(U) is higher than *DR*. It is because the coarse-grained filtering always returns non-duplicate bug reports for judgement, while for a duplicate bug case, the coarse-grained filtering might not retrieve the duplicate report, increasing the risk of misjudgments. Since identifying unique cases follows the same pipeline as duplicate cases in *Pisco*, they have similar identifying cost.

7.4.5 LLM Comparison. We conduct additional experiments in an enhanced hardware environment to support three widely used models, including DeepSeek-v3, QWen3, and LLaMA3.1. Table 4 shows that using them could further improve the performance with more deduplication time due to larger parameter sizes. Note, DeepSeek-v3 and LLaMA3.1 have more invocation counts than other models. This is because they take more rounds of discussion to reach a consensus when expert agents have conflicting conclusions, as these two LLMs have been proven to be more likely to stand up for their previous conclusions [34].

7.5 Use Case of *Pisco*

Pisco has helped Leopard submit 12 unique bugs for TDSQL, TiDB, and a commercial database (anonymized as DBX). We define the *bug confirmation time* as the duration from submission to official confirmation, and define the *rounds of reporter-developer interactions* as the number of times the developers request additional information from the reporter. Most of our submitted bugs are confirmed within 2 days without interactions (details in our report [59]), as our submitted cases are minimal and deterministically reproduce bugs. We survey the transactional bugs in MySQL and find that it took an average of 23.9 days and 2.8 rounds of interactions to confirm the bugs, while TDSQL [8]'s developers took 10.4 days and 9.8 rounds of interactions. In MySQL, the bug #98642 was reproduced by a 7-day duration after 13 rounds of interactions, providing two additional videos and a Docker image by the reporter.

8 RELATED WORK

Bug Case Reproduction. It aims to find the correct server-side order to execute operations in the bug case and trigger the bug deterministically. In general, previous work can be classified into two kinds. The first kind leverages kernel information to infer the order. *Oracle Database Replay* [49] executes each operation and obtains its unique logic timestamp through kernel callbacks, enabling precise order inference. However, its strict execution orders, even on non-conflicting operations, limit the replay concurrency. To improve the replay efficiency, *DoppelGanger++* [38] detects and removes unnecessary dependencies. In addition, deterministic simulation testing is seen as a gold standard for deterministically reproducing bugs. For example, *FoundationDB* [79] relies on a deterministic simulator to reliably record and reproduce the bug cases. However, it requires substantial engineering effort to customize a simulator for each specific DBMS and does not generalize across systems. Besides, all these methods require DBMS kernel information and cannot be applied to black-box testing. The second kind infers the order by enumeration. Partial Order Reduction (*POR*) [11] prunes the search space by identifying operation orders with equivalent execution results. Dynamic POR (*DPOR*) [20] detects independent operations that access different data items and prunes their orders. *FlyMC* [47] and *SAMC* [39] extend *DPOR* by providing white-box information and heuristic rules extracted from domain knowledge to detect unnecessary interleavings (redundant orders) that lead to the same results. However, they all have to enumerate different orders to attempt to trigger the bug, limiting their scalability.

Bug Case Reduction. It aims to reduce the raw bug case, mostly leveraging delta debugging to remove bug-irrelevant operations. Some work focuses on reducing SQL test cases, which can be categorized into two types. The first type reduces single complex SQL statements, such as *SQLess* [45] and *APOLLO* [31]. Specifically, *SQLess* models dependencies among expressions in a SQL and applies delta debugging to remove bug-irrelevant expressions. *APOLLO* includes a debugging component with a reducer, which reduces bug-irrelevant expressions in each buggy SQL by trial-and-error. However, they cannot extend to large isolation bug cases as they only reduce a part of the SQL statement instead of removing entire SQL. The second type supports reducing bug cases containing multiple operations rather than a single SQL statement, such as *reducer.sh* [61]. This approach partitions all these operations into several chunks, and removes them via trial-and-error. However, since it does not control inter-thread operation ordering, it may not reproduce the bug even after removing irrelevant operations.

There also exist several common reduction methods. *DDMin* [75] takes a list of operations as input and applies a divide-and-conquer approach by evenly dividing that list. When the bug is deterministically triggered, *DDMin* can produce a minimized result. *C-Reduce* [62] is a reduction approach for C/C++ programs. It integrates advanced language-specific reduction strategies (e.g., function inlining) and a DDMin-like algorithm to reduce C/C++ buggy code. Based on domain-specific knowledge, it tries to remove bug-related operations together, mitigating the redundant trials caused by dividing bug-related operations into different subsets. Donaldson et al. [18] locate bug-related operations by finding a correct test case that differs minimally from the buggy one. However, since the isolation bug case involves non-deterministic concurrent scheduling of operations/transactions, simple repeated executions of a bug case without controlling execution order might not reproduce the bug.

Beyond reducing deterministic bug cases, a variety of work has explored interleavings in thread schedules for bug reduction. Zeller et al. [9] and *LEAN* [25] use the *DEJAVU* tool and multiprocessor deterministic replay (*MDR*) technique to capture the thread scheduling order of the failed execution, respectively. Then, they apply delta debugging to locate the buggy schedule and bug-related events. Additionally, *LEAN* builds a dependency graph based on function calls, which helps remove dependent events together. However, the tools employed by these two approaches are designed for specific systems (e.g., JVM), making them cannot be applied to DBMSs.

Bug Case Deduplication. It aims to detect duplicate bug reports and thus avoid redundant analysis. Previous work generally calculates the similarity between bug reports to identify duplicates. Specifically, *REP* [67] collects the classical textual statistics (e.g., word frequency) from the bug's descriptions to calculate the cosine similarity between bug reports. In contrast, *SiamesePair* [17]and *SABD* [64] propose to encode the textual description of the bug reports based on deep learning and calculates the cosine similarity between the encoded vectors. In particular, *SABD* additionally leverages an attention mechanism to focus on the key features in the encoded vectors, improving its ability to distinguish different bugs. However, since most terms in isolation bug reports are reserved keywords, the bug reports cannot be easily distinguished by the textual similarity that is used in the above methods.

9 CONCLUSION AND LIMITATION

This paper presents *Pisco*, an isolation bug case reduction and deduplication framework. It has the advantages of deterministic bug reproduction, efficient case reduction, and accurate case deduplication. However, *Pisco* has its own limitations. First, it can only handle isolation bug cases which are deterministically triggered by executing operations in a specific order. Second, it is not fully automatic and still requires a final manual duplicate validation step. Besides, it requires multiple LLM invocations for an accurate deduplication result. We leave them as the future work.

ACKNOWLEDGMENTS

This work is supported by NSFC (62461146205, 92270202), National Key R&D Program of China (2024YFC3308102), and Fundamental Research Funds for the Central Universities (YBNLTS2024-016).

REFERENCES

[1] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized isolation level definitions. In *ICDE*. 67–78.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, Vol. 24. 1–10.

[3] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. In *PACMPL*, Vol. 3. 1–28.

[4] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A framework for transactional consistency models with atomic visibility. In *CONCUR*, Vol. 42. 58–71.

[5] Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2024. ChatEval: Towards Better LLM-based Evaluators through Multi-Agent Debate. In *ICLR*.

[6] Lixiang Chen, Yuxing Han, Yu Chen, Xing Chen, Chengcheng Yang, and Weining Qian. 2025. AQETuner: Reliable Query-level Configuration Tuning for Analytical Query Engines. In *VLDB*, Vol. 18. 2709–2721.

[7] Sibei Chen, Ju Fan, Bin Wu, Nan Tang, Chao Deng, Pengyi Wang, Ye Li, Jian Tan, Feifei Li, Jingren Zhou, et al. 2025. Automatic Database Configuration Debugging using Retrieval-Augmented Language Models. In *SIGMOD*, Vol. 3. 1–27.

[8] Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2024. TDSQL: Tencent Distributed Database System. In *VLDB*, Vol. 17. 3869–3882.

[9] Jong-Deok Choi and Andreas Zeller. 2002. Isolating failure-inducing thread schedules. In *ISSTA*. 210–220.

[10] Justin Chu, Tingting Yu, Jane Huffman Hayes, Xue Han, and Yu Zhao. 2022. Effective fault localization and context-aware debugging for concurrent programs. In *Softw. Test. Verification Reliab.*, Vol. 32. e1797.

[11] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. 1999. State Space Reduction Using Partial Order Techniques. In *Int. J. Softw. Tools Technol. Transf.*, Vol. 2. 279–287.

[12] Ali Cloud. 2025. Pricing of qwen-max. https://bailian.console.aliyun.com/console?tab=model#/model-market/detail/qwen-max.

[13] Tencent Cloud. 2025. Pricing of Tecent Cloud Server. https://cloud.tencent.com/product/cvm?Is=sdk-topnav.

[14] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is believing: A client-centric specification of database isolation. In *PODC*. 73–82.

[15] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially testing database transactions for fun and profit. In *ASE*. 1–12.

[16] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, Jun Wei, and Tao Huang. 2024. Understanding Transaction Bugs in Database Systems. In *ICSE*. 163:1–163:13.

[17] Jayati Deshmukh, KM Annervaz, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. 2017. Towards accurate duplicate bug retrieval using deep learning techniques. In *ICSME*. 115–124.

[18] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *PLDI*. 1017–1032.

[19] Wensheng Dou, Ziyu Cui, Qianwang Dai, Jiansen Song, Dong Wang, Yu Gao, Wei Wang, Jun Wei, Lei Chen, Hanmo Wang, Hua Zhong, and Tao Huang. 2023. Detecting isolation bugs via transaction oracle construction. In *ICSE*. 1123–1135.

[20] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *POPL*. 110–121.

[21] Jim N Gray, Raymond A Lorie, and Gianfranco R Putzolu. 1975. Granularity of locks in a shared data base. In *VLDB*. 428–451.

[22] Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. 2025. Token-Budget-Aware LLM Reasoning. In *ACL*, Vol. 2025. 24842–24855.

[23] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *ICLR*.

[24] Zirui Hu, Rong Zhang, Chengcheng Yang, Xuan Zhou, Quanqing Xu, and Chuanhui Yang. 2025. Artemis: A Customizable Workload Generation Toolkit for Benchmarking Cardinality Estimation. In *ICDE*. 4628–4631.

[25] Jeff Huang and Charles Zhang. 2012. LEAN: simplifying concurrency bug reproduction via replay-supported execution reduction. In *OOPSLA*. 451–466.

[26] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. In *TOIS*, Vol. 43. 1–55.

[27] Xuhua Huang, Zirui Hu, Siyang Weng, Rong Zhang, Chengcheng Yang, Xuan Zhou, Weining Qian, Chuanhui Yang, and Quanqing Xu. 2025. A Query-Aware Enormous Database Generator For System Performance Evaluation. In *SIGMOD*. 131–134.

[28] Jepsen. 2025. Distributed Systems Safety Research. https://jepsen.io/.

[29] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. 2021. Igor: Crash Deduplication Through Root-Cause Clustering. In *CCS*. 3318–3336.

[30] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting transactional bugs in database engines via graph-based oracle construction. In *OSDI*. 397–417.

[31] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woon-Hak Kang. 2019. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. In *VLDB*, Vol. 13. 57–70.

[32] Jongbin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the scan in mvcc databases. In *SIGMOD*. 938–950.

[33] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. In *VLDB*, Vol. 14. 268–280.

[34] Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. LLMs Get Lost In Multi-Turn Conversation. *CoRR* abs/2505.06120.

[35] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. In *VLDB*, Vol. 8. 1740–1751.

[36] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM Sigplan Notices*, Vol. 49. 216–226.

[37] Seungpil Lee, Woochang Sim, Donghyeon Shin, Wongyu Seo, Jiwon Park, Seokki Lee, Sanha Hwang, Sejin Kim, and Sundong Kim. 2025. Reasoning abilities of large language models: In-depth analysis on the abstraction and reasoning corpus. In *TIST*, Vol. 16. 137:1–137:52.

[38] Wonseok Lee, Jaehyun Ha, Wook-Shin Han, Changgyoo Park, Myunggon Park, Juhyeng Han, and Juchang Lee. 2024. DoppelGanger++: Towards Fast Dependency Graph Generation for Database Replay. In *SIGMOD*, Vol. 2. 1–26.

[39] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems. In *OSDI*. 399–414.

[40] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *EMNLP*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). 3045–3059.

[41] Keqiang Li, Siyang Weng, Peiyuan Liu, Lyu Ni, Chengcheng Yang, Rong Zhang, Xuan Zhou, Jianghang Lou, Gui Huang, Weining Qian, and Zhou Aoying. 2023. Leopard: A black-box approach for efficiently verifying various isolation levels. In *ICDE*. 722–735.

[42] Keqiang Li, Siyang Weng, Lyu Ni, Chengcheng Yang, Rong Zhang, Xuan Zhou, and Aoying Zhou. 2024. DBStorm: Generating Various Effective Workloads for Testing Isolation Levels. In *ISSTA*. 755–767.

[43] Sijia Li, Peng Cai, Zhifan Zhang, Huiqi Hu, Rong Zhang, Xuan Zhou, Quanqing Xu, and Chuanhui Yang. 2025. APQO: An Adaptive Framework for Parametric Query Optimization. In *PACMMOD*, Vol. 3. 1–25.

[44] Yuming Li, Rong Zhang, Xiaoyan Yang, Zhenjie Zhang, and Aoying Zhou. 2018. Touchstone: Generating Enormous Query-Aware Test Databases. In *ATC*. 575–586.

[45] Li Lin, Zongyin Hao, Chengpeng Wang, Zhuangda Wang, Rongxin Wu, and Gang Fan. 2024. SQLess: Dialect-Agnostic SQL Query Simplification. In *ISSTA*. 743–754.

[46] Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. 2024. Large Language Models to Enhance Bayesian Optimization. In *ICLR*.

[47] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar Heri Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *EuroSys*. 20:1–20:16.

[48] MariaDB. 2025. MariaDB Server. https://mariadb.org/.

[49] Konstantinos Morfonios, Romain Colle, Leonidas Galanis, Supiti Buranawatanachoke, Benoît Dageville, Karl Dias, and Yujun Wang. 2011. Consistent synchronization schemes for workload replay. In *VLDB*, Vol. 4. 1225–1236.

[50] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. 2022. An In-depth Analysis of Duplicated Linux Kernel Bug Reports. In *NDSS*.

[51] MySQL. 2025. MySQL Bugs. https://bugs.mysql.com/bug.php.

[52] MySQL. 2025. MySQL Database. https://www.mysql.com/.

[53] Bach Hoai Nguyen, Bing Xue, and Peter Andreae. 2016. Mutual information for feature selection: estimation or counting?. In *Evol. Intell.*, Vol. 9. 95–110.

[54] U.S. Bureau of Labor Statistics. 2025. Software Developers, Quality Assurance Analysts, and Testers. https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm.

[55] Oracle. 2025. Oracle Database. https://www.oracle.com/.

[56] Zhicheng Pan, Yuanjia Zhang, Chengcheng Yang, Ahmad Ghazal, Rong Zhang, Huiqi Hui, Xiaoju Wu, Yu Dong, and Xuan Zhou. 2025. Hyper: Hybrid Physical Design Advisor with Multi-agent Reinforcement Learning. In *ICDE*. IEEE, 1565–1578.

[57] Frédéric Pennerath, Panagiotis Mandros, and Jilles Vreeken. 2020. Discovering Approximate Functional Dependencies using Smoothed Mutual Information. In *SIGKDD*. 1254–1264.

[58] Pisco. 2025. The dataset of isolation bug cases. https://doi.org/10.5281/zenodo.17732684.

[59] Pisco. 2025. Technical Report and Source Codes of Pisco. https://github.com/DBHammer/Pisco.

[60] Xiangfei Qiu, Xiuwen Li, Ruiyang Pang, Zhicheng Pan, Xingjian Wu, Liu Yang, Jilin Hu, Yang Shu, Xuesong Lu, Chengcheng Yang, Chenjuan Guo, Aoying Zhou, Christian S. Jensen, and Bin Yang. 2025. EasyTime: Time Series Forecasting Made Easy. In *ICDE*. 4564–4567.

[61] Reducer.sh 2014. Reducer.sh – A powerful MySQL test-case simplification/reducer tool. https://www.percona.com/blog/reducer-sh-a-powerful-mysql-test-case-simplificationreducer-tool/.

[62] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI*. 335–346.

[63] Stephen E. Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4, 333–389.

[64] Irving Muller Rodrigues, Daniel Aloise, Eraldo Rezende Fernandes, and Michel Dagenais. 2020. A soft alignment model for bug deduplication. In *MSR*. 43–53.

[65] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, Hrishikesh B. Acharya, Kyriakos Zarifis, and Scott Shenker. 2014. Troubleshooting blackbox SDN control software with minimal causal sequences. In *SIGCOMM*. ACM, 395–406.

[66] Sonish Sivarajkumar, Mark Kelley, Alyssa Samolyk-Mazzanti, Shyam Visweswaran, and Yanshan Wang. 2024. An empirical evaluation of prompting strategies for large language models in zero-shot clinical natural language processing: algorithm development and validation study. In *JMIR*, Vol. 12. e55318.

[67] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *ASE*. 253–262.

[68] Wenwen Sun, Zhicheng Pan, Zirui Hu, Yu Liu, Chengcheng Yang, Rong Zhang, and Xuan Zhou. 2025. Rabbit: Retrieval-Augmented Generation Enables Better Automatic Database Knob Tuning. In *ICDE*. 3807–3820.

[69] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making transactional key-value stores verifiably serializable. In *OSDI*. 63–80.

[70] TiDB#42487 2023. Transaction Can't Read Its Own Update in Repeatable Read. · Issue #42487 · Pingcap/Tidb. https://github.com/pingcap/tidb/issues/42487.

[71] Jorge R Vergara and Pablo A Estévez. 2014. A review of feature selection methods based on mutual information. In *NEURAL COMPUT APPL.*, Vol. 24. 175–186.

[72] Qingshuai Wang, Hao Li, Zirui Hu, Rong Zhang, Chengcheng Yang, Peng Cai, Xuan Zhou, and Aoying Zhou. 2024. Mirage: Generating Enormous Databases for Complex Workloads. In *ICDE*. 3989–4001.

[73] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *NeurIPS*.

[74] Jamil Zaghir, Marco Naguib, Mina Bjelogrlic, Aurélie Névéol, Xavier Tannier, and Christian Lovis. 2024. Prompt engineering paradigms for medical applications: scoping review and recommendations for better practices. In *arXiv preprint arXiv:2405.01249*.

[75] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. In *TSE*, Vol. 28. 183–200.

[76] Yifan Zeng, Ojas Tendolkar, Raymond Baartmans, Qingyun Wu, Lizhong Chen, and Huazheng Wang. 2024. LLM-RankFusion: Mitigating Intrinsic Inconsistency in LLM-based Ranking. In *arXiv preprint arXiv:2406.00231*.

[77] Huidong Zhang, Luyi Qu, Qingshuai Wang, Rong Zhang, Peng Cai, Quanqing Xu, Zhifeng Yang, and Chuanhui Yang. 2023. Dike: A benchmark suite for distributed transactional databases. In *SIGMOD*. 95–98.

[78] Ting Zhang, DongGyun Han, Venkatesh Vinayakarao, Ivana Clairine Irsan, Bowen Xu, Ferdian Thung, David Lo, and Lingxiao Jiang. 2023. Duplicate bug report detection: How far are we?. In *TOSEM*, Vol. 32. 1–32.

[79] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Yang Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2023. FoundationDB: A Distributed Key-Value Store. In *Commun. ACM*, Vol. 66. 97–105.

[80] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2024. D-Bot: Database Diagnosis System using Large Language Models. In *VLDB*, Vol. 17. 2514–2527.