A PROOF

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

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

B COMPLEXITY ANALYSIS

Algorithm 1. Suppose there are l trace queues and the total number of operations is N. Scanning time-overlapping operations requires checking the head of each trace queue, so each scan visits l operations. Among these l operations, the time-overlapping ones require further inference (lines 9-14 in Algorithm 1). Note that each read operation needs to search its access data version in the version chain, while each write operation needs to search its write data in the lock set. To speed up searching, we organize them with a hash structure so that the inference of an operation has the complexity of $O(1)$. As a result, the total cost of each scan is $O(l)$.

Suppose the average batch size is $|\mathcal{B}|$, there are $\frac{N}{|\mathcal{B}|}$ batches in total. If it takes k rounds to scan the time-overlapping operations for each batch, constructing all batches has a complexity of $O(\frac{l \cdot N \cdot k}{|\mathcal{B}|})$. In particular, $k \ll N$. Therefore, the total complexity can be simplified as $O(\frac{l \cdot N}{|\mathcal{B}|})$.

Algorithm 2. The time complexity of reducing all reduction units irrelevant to the bug is $O(N)$, where N is the number of all operations in the bug case. At first, C_B contains all operations in the bug case, i.e., N operations. Suppose that S contains x reduction units on average, and the number of remaining operations in C_B is $|C_B|$. Since bug-related operations are included in a single reduction unit, executing the divide-and-conquer method (line 3 in Algorithm 2) once reduces the number of remaining operations in C_B to $\frac{|C_B|}{x}$. Thus, after the i^{th} execution of the divide-and-conquer method, the number of remaining operations in C_B becomes $\frac{N}{x^i}$. Notice that $\frac{N}{x^i} > 1$, the divide-and-conquer method is invoked $O(\log_x N)$ times. Each time, the divide-and-conquer method recursively repeats `DivideConquer` to reduce the irrelevant reduction units in S. During `DivideConquer`, the operations in $(C_B \setminus S)$ are replayed to validate whether S contains bug-related operations. Thus, executing `DivideConquer` costs $O(|C_B \setminus S|) < O(|C_B|)$. Since `DivideConquer` reduces half of the remaining reduction units in C_B each time, the divide-and-conquer method costs $\sum_{i=0}^{\log x} \frac{|C_B|}{2^i} < 2|C_B|$ in total. Therefore, the total cost of Algorithm 2 is $O(\sum_{i=1}^{\log_x N} 2\frac{N}{x^i}) = O(\frac{2(N-1)}{x-1}) = O(N)$.

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

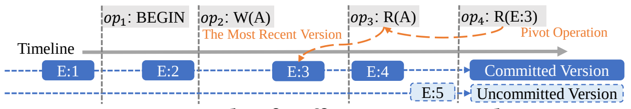

Figure 1: Example of Different Access Behaviors

3 EXPECTED ACCESS BEHAVIORS

Based on the authoritative official documentation of DBMSs, we classify the expected access behaviors of operations into five types:

(1) The operation reads the snapshot created at its transaction's start time. In this case, the operation should access the most recent version committed before its transaction's *begin* operation. In Fig. 1, the operation op_4 would access $E = 1$. For example, this mechanism is adopted by the IL of *snapshot isolation* (SI) in SQL Server.

(2) The operation reads the snapshot created before its transaction's first non-transaction-control operation. In Fig. 1, the operation op_4 would access $E = 2$. Take PostgreSQL as an example, this mechanism is adopted by its ILs of *serializable* (SR) and *repeatable read* (RR).

(3) The operation reads the snapshot created before its transaction's first read operation. In Fig. 1, the operation op_4 would access $E = 3$. For example, this mechanism is adopted by the ILs of SR and RR in InnoDB.

(4) The operation reads the most recent committed version before it. In Fig. 1, the operation op_4 would access $E = 4$. This mechanism is widely used by the IL of *read committed* (RC).

(5) The operation reads the most recent version before it no matter whether the version is commited. In Fig. 1, the operation op_4 would access $E = 4$. For example, this mechanism is adopted by the IL of *read uncommitted* (RU) in InnoDB and SQL Server.

4 BUGS REPORTED BY *PISCO*

In the past two years, *Pisco* has helped Leopard [30] to submit 12 unique bugs, which are listed in Table 1. Specifically, the *confirmation time duration* indicates how long it takes the official community to confirm the bug from the bug submission time, and the *rounds of interactions* indicate the number of times the developer requested additional bug information from the bug reporter. We observe that most of our submitted bugs are confirmed within 2 days without interactions, because the reduced cases submitted by *Pisco* are the minimal ones and can deterministically reproduce the bugs. An exception is for bug #2145 in DBX, which took a 7-day confirmation time duration. However, it is caused by a 6-day delayed response to the submitted bug, which is actually confirmed in one day without further interaction. We manually check the transactional bugs in MySQL community and find that it takes an average of 23.9 days and 2.8 rounds of interactions to confirm the bug reports, while TDSQL's developers take 10.4 days and 9.8 rounds of interactions. For instance, the bug #98642 was reproduced by a 7-day duration

Table 1: Bugs Reported By *Pisco*

Issue ID	Description	Confirmation Time(Day)	Rounds of Interaction
TiDB#42487	Transaction can't read its own update in repeatable read.	1	0
TiDB#45677	Update with sub query uses incorrect snapshot in RR isolation level.	2	0
TiDB#44379	Return deleted record in repeatable-read transaction.	2	0
TiDB#50393	Join between blob type with matching returns incorrect result.	1	0
TiFlash#8776	Like function behaves incorrectly for blob type args.	0	0
TDSQL#4056	JDBC Timeout would rollback the entire transaction.	1	0
TDSQL#3290	Update with sub query uses incorrect snapshot.	0	0
TDSQL#4612	INSERT IGNORE inserts incorrect duplicate values.	0	0
TDSQL#3522	Return incorrect data when using index.	0	0
TDSQL#3322	JDBC core dump when executing subquery.	0	0
TDSQL#3468	Get inconsistent snapshot with MySQL in RR isolation level.	1	1
DBX#2145	Update with sub query uses incorrect snapshot in RR isolation level.	7	0

after 13 rounds of interactions between the community and the reporter, providing two additional videos and a Docker image by the reporter.

In addition, *Pisco* also helps submit non-isolation bugs. The main reason is that these non-isolation bugs are triggered when operations read incorrect results, similar to the isolation bugs. Thus, *Pisco*'s reproduction and reduction methods remain applicable. For instance, TiDB #50393 stems from an incorrect join involving BLOB types. In this case, *Pisco* can reduce the case to the incorrect join operation and the operation that produces the version it reads.