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

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

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

Regarding the entire operation batch sequence, suppose that the average batch size is $|\mathcal{B}|$, resulting in $\frac{N}{|\mathcal{B}|}$ batches in total. The overall time complexity is $O(\frac{k \cdot m \cdot p \cdot N}{|\mathcal{B}|})$. In practice, since $p \ll N$, the time complexity can be further simplified to $O(\frac{k \cdot m \cdot N}{|\mathcal{B}|}) = O(N)$.

Recursive Reduction. Given a raw bug case consisting of N operations, our recursive operation reduction process takes multiple rounds. In each round, the bug case is recursively reduced at the granularity of reduction units, with a replay time complexity proportional to the number of remaining operations. Let n_i denote the operation number of the bug case before the i-th round. Since each recursive step can remove half of the operations in the reduced case, the time complexity of the i-th round is $O(\sum_{j=1}^{\lceil \log n_i \rceil} \frac{n_i}{2^j}) < O(n_i)$.

Assume that each round divides the current case into m reduction units ($m \geq 2$), and only the unit that containing the bug-related operations is retained for the next round. Thus, each round reduces the number of remaining operations by a factor of $\frac{1}{m}$. Note, the original bug case contains all operations, i.e., $n_1 = N$. After the i-th round, the reduced case size would be $n_i = \frac{N}{m^i}$, and the time complexity of the i-th round is $O(\frac{N}{m_i})$. Finally, the total time complexity of our recursive operation reduction is $O(\sum_{i=1}^{\lceil \log N \rceil} \frac{N}{m^i}) = O(N)$.

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 $O(k \cdot K)$.