APPENDIX

A. Proof

Theorem 1: Algorithm 1 dispatches traces in a monotonically increasing order of before timestamps.

Proof. Since traces are produced by test clients in the time order, the pushed traces are earlier than the unpushed ones. Moreover, the traces of each local buffer $\mathcal{L}_i$ are organized in a FIFO manner, so the first trace in each local buffer $\mathcal{L}_i[0]$ has:

$$\mathcal{L}_i[0].ts_{bef} \leq \min_{\mathcal{T} \in \mathcal{L}_i} \mathcal{T}.ts_{bef} \tag{1}$$

Because the watermark $\mathcal{W}$ is the smallest before timestamp among all traces in local buffers and only the traces with before timestamp smaller than $\mathcal{W}$ are fetched into the global buffer, the traces in the global buffer meet:

$$\forall \mathcal{T} \in \mathcal{G}, \mathcal{T}.ts_{bef} \leq W, where\ W = \min_{\mathcal{L}_i \in \mathbb{L}} \mathcal{L}_i[0].ts_{bef} \tag{2}$$

According to Algorithm 1, the dispatched trace $\mathcal{T}_0$ is picked from the top of min-heap, which satisfies:

$$\mathcal{T}_0.ts_{bef} \leq \min_{\mathcal{T} \in \mathcal{G}} \mathcal{T}.ts_{bef} \tag{3}$$

By combining the above three inequalities, we obtain:

$$\begin{aligned} \mathcal{T}_0.ts_{bef} &\leq \min_{\mathcal{T} \in \mathcal{G}} \mathcal{T}.ts_{bef} \\ &\leq \min_{\mathcal{L}_i \in \mathbb{L}} \mathcal{L}_i[0].ts_{bef} \\ &\leq \min_{\mathcal{L}_i \in \mathbb{L}, \mathcal{T} \in \mathcal{L}_i} \mathcal{T}.ts_{bef} \end{aligned} \tag{4}$$

Therefore, the dispatched trace has the minimal before timestamp among all traces in the global buffer, local buffers and clients. In other words, Algorithm 1 always returns the trace with the smallest before timestamp. Since each call of Algorithm 1 returns the earliest trace at that time, these dispatched traces form a monotonically increasing order of before timestamps. □

Theorem 2: Given a read operation op, its trace $\mathcal{T}$ and its visible snapshot time interval $\mathcal{S}^{\mathcal{T}} = (ts_{bef}, ts_{aft})$. Suppose the read operation op has a wr dependency on the version x^i of a record x. Then we can restrict the before timestamp of the visible snapshot time interval as $max(\mathcal{S}^{\mathcal{T}}.ts_{bef}, \mathcal{V}.ts_{bef})$ where $\mathcal{V}$ is the version installation time interval of the version x^i, and decrease the after timestamp of the visible snapshot time interval as $min(\mathcal{S}^{\mathcal{T}}.ts_{aft}, \mathcal{V}'.ts_{aft})$ where $\mathcal{V}'$ is the version installation time interval of the version x^{i+1}.

Proof. Because the read operation op fetches the version x^i, the timestamp of creating the snapshot appears after (resp. before) the timestamp of installing the version x^i (resp. x^{i+1}). Thus, the before timestamp of the visible snapshot time interval, i.e., $\mathcal{S}^{\mathcal{T}}.ts_{bef}$, should not be smaller than the before timestamp of the version installation time interval $\mathcal{V}$ of the version x^i, i.e., $\mathcal{V}.ts_{bef}$. Similarly, the after timestamp of the visible snapshot time interval, i.e., $\mathcal{S}^{\mathcal{T}}.ts_{after}$, should not be larger than the after timestamp of the version installation time interval $\mathcal{V}'$ of the version x^{i+1}, i.e., $\mathcal{V}'.ts_{aft}$. □

Property 1: The candidate version set contains a minimum number of versions that are possibly visible to a given read.

Proof. Let x^i denote the i^{th} version of data x, which is included in the candidate version set yet remains invisible to a particular read operation. For x^i to be invisible, one of the two conditions must hold. In the first case, x^i is installed after the read operation, i.e., x^i is a future version*.*

In the second case, the version x^i appears before the read operation but has been overwritten by another version. As discussed above, we could not determine the chronological order of the pivot overlap version*,* pivot version *and* overlap version *since their exact install times are not available. That is, any version among them is possibly visible to the read operation. Thus, the version x^i must not be one of the three versions, which implies that x^i is a* garbage version*.*

Since our approach excludes all the future versions *and* garbage versions *from the candidate version set, this is contradicted with the initial assumption. The theorem is proven.* □

Property 2: Given two transactions t_0 and t_1 that acquire two locks on the same record through write operations, there exists at most one possible order in which a ww dependency can be deduced.

Proof. Suppose there exist two possible orders in which two ww dependencies can be deduced. That is, t_0 has a ww dependency on t_1, and t_1 has a ww dependency on t_0. Since t_0 has a ww dependency on t_1, the exact lock-acquiring time of t_0 must happen before that of t_1. In this case, t_0 releases the lock before t_1 acquires it. Thus, the timestamp before t_0 releasing the lock, i.e., $\mathcal{R}^{\mathcal{T}_{c_{t_0}}}.ts_{bef}$, is smaller than the timestamp after t_1 acquiring the lock, i.e., $\mathcal{A}^{\mathcal{T}_{w_{t_1}}}.ts_{aft}$. That is, $\mathcal{R}^{\mathcal{T}_{c_{t_0}}}.ts_{bef} < \mathcal{A}^{\mathcal{T}_{w_{t_1}}}.ts_{aft}$. Besides, t_0 acquires the lock before releasing it, i.e., $\mathcal{A}^{\mathcal{T}_{w_{t_0}}}.ts_{aft} < \mathcal{R}^{\mathcal{T}_{c_{t_0}}}.ts_{bef}$. Consequently, we can deduce that $\mathcal{A}^{\mathcal{T}_{w_{t_0}}}.ts_{aft} < \mathcal{A}^{\mathcal{T}_{w_{t_1}}}.ts_{aft}$. Similarly, since t_1 has a ww dependency on t_0, we can also deduce that $\mathcal{A}^{\mathcal{T}_{w_{t_1}}}.ts_{aft} < \mathcal{A}^{\mathcal{T}_{w_{t_0}}}.ts_{aft}$. However, the two inequalities are mutually contradictory. Therefore, at most one valid ww dependency order exists for t_0 and t_1. □

Property 3: Given two committed transactions t_0 and t_1 with time overlap and conflicting writes, there exists at most one possible order in which a ww dependency can be deduced.

Proof. The proof is similar to that of Property 2. Specifically, suppose there exist two possible orders in which two ww dependencies can be deduced. That is, t_0 has a ww dependency on t_1, and t_1 has a ww dependency on t_0. Since t_0 has a ww dependency on t_1, t_1 reads the version x^0 created by t_0. Thus, the timestamp before t_0 commits x^0, i.e., $\mathcal{T}_{c_{t_0}}.ts_{bef}$, is smaller than the timestamp after t_1 reading a visible snapshot including x^0, i.e., $\mathcal{S}^{\mathcal{T}_{r_{t_1}}}.ts_{aft}$. That is, $\mathcal{T}_{c_{t_0}}.ts_{bef} < \mathcal{S}^{\mathcal{T}_{r_{t_1}}}.ts_{aft}$. Besides, t_0 reads a visible snapshot before its commit, i.e., $\mathcal{S}^{\mathcal{T}_{r_{t_0}}}.ts_{aft} < \mathcal{T}_{c_{t_0}}.ts_{bef}$. Consequently, we can deduce that $\mathcal{S}^{\mathcal{T}_{r_{t_0}}}.ts_{aft} < \mathcal{S}^{\mathcal{T}_{r_{t_1}}}.ts_{aft}$. Similarly, since t_1 has a ww dependency on t_0, we can also deduce that $\mathcal{S}^{\mathcal{T}_{r_{t_1}}}.ts_{aft} < \mathcal{S}^{\mathcal{T}_{r_{t_0}}}.ts_{aft}$. However, the two inequalities are mutually contradictory. Therefore, at most one valid ww dependency order exists for t_0 and t_1. □

Property 4: A *garbage transaction* t is not a part of any future cycle on *DG*.

Proof. C1 *in Definition 4 shows that the in-degree of* t *is zero unless a future transaction creates a new dependency on* t*. Let* $\mathcal{T}_k$ *be the trace of the first operation of any committed transaction in the future.* C2 *in Definition 4 deduce that* $\mathcal{T}_t.ts_{aft} \leq \mathcal{S}_e \leq \mathcal{S}^{\mathcal{T}_k}.ts_{bef}$*, so any future transaction would not have dependencies on* t*. Taken together, the in-degree of* t *will keep as zero, i.e., no edge comes from* t*. Thus,* t *is not a part of any future cycle on* DG*.*

□

B. Record Identification

With explicit record identifiers. Explicit record identifiers are commonly introduced by test workload generators [1-5] (e.g., Jepsen, Cobra, Viper, TxCheck, and APTrans) to help isolation-level verifiers identify records. Based on the provided explicit record identifiers, *Leopard* can directly identify records in the workload. Specifically, existing test workload generators typically adopt one of the following two strategies to identify records.

First, some approaches abstract SQL statements as key-value operations, where each operation explicitly specifies the accessed key. In this setting, record identities are directly available [1–3]. Second, the other methods propose to explicitly attach a record identifier column for each table [4, 5]. For example, APTrans [5] augments the schema with additional immutable identifier columns and instruments workloads to expose record identifiers. Such approaches enable accurate record identification at the cost of modifying workload and schema.

Without explicit record identifiers. When workloads do not expose record identifiers explicitly, *Leopard* identifies records from SQL predicates observed at the client side. Specifically, *Leopard* proposes the *predicate region* to serve as the identifier, which represents the records that an operation would access. The identifier is used to determine whether two operations access at least one same record. If so, these two operations are treated as conflicting, and *Leopard* should deduce their dependency. To explain this process, we first introduce the concepts of *domain* and *predicate region*.

Domain. Let the *domain* $\mathcal{D}$ denote the set of all possible values accessed by operations, which is typically defined by the table schemas. Each column involved in these tables defines a dimension in the domain. Importantly, the domain represents the space of all possible values, rather than the set of concrete records stored in the database. For example, if a query accesses a single table, the domain consists of all possible values in that table; if it involves a join of multiple tables, the domain is the Cartesian product of possible values from the involved tables.

Predicate region. Given a predicate p, the *predicate region* $\mathcal{R}(p) \subseteq \mathcal{D}$ is the subset of possible values in the domain $\mathcal{D}$ that satisfy p. Each predicate region serves as an *identifier* representing the set of records that would be accessed by the operation. For predicates with multiple conditions combined via `AND`, `OR`, or `NOT`, the region is obtained by using standard set operations [6]:

$$\mathcal{R}(p_1 \wedge p_2) = \mathcal{R}(p_1) \cap \mathcal{R}(p_2), \quad \mathcal{R}(p_1 \vee p_2) = \mathcal{R}(p_1) \cup \mathcal{R}(p_2),$$

$$\mathcal{R}(\neg p_1) = \mathcal{D} \setminus \mathcal{R}(p_1)$$

Then, consider two operations op_1 and op_2 that access records identified by $\mathcal{R}_1$ and $\mathcal{R}_2$, their relationships can be classified as follows:

1) **Identical.** If $\mathcal{R}_1 = \mathcal{R}_2$, op_1 and op_2 access one or more same records. In this case, *Leopard* treats op_1 and op_2 as conflicting.
2) **Subsuming.** If $\mathcal{R}_1 \subseteq \mathcal{R}_2$, the records accessed by op_1 contain all records accessed by op_2, and *Leopard* treats op_1 and op_2 as conflicting. Similarly, if $\mathcal{R}_2 \subseteq \mathcal{R}_1$, then op_1 and op_2 are also treated as conflicting.
3) **Disjoint.** If $\mathcal{R}_1 \cap \mathcal{R}_2 = \emptyset$, the operations are considered to access different records and are non-conflicting.
4) **Overlapping but non-subsuming.** If $\mathcal{R}_1 \cap \mathcal{R}_2 \neq \emptyset$ but neither region subsumes the other, the two operations might access the same records whose values lie in $\mathcal{R}_1 \cap \mathcal{R}_2$. However, based solely on client-side information, *Leopard* cannot determine whether $\mathcal{R}_1 \cap \mathcal{R}_2$ contains any record in the database. Therefore, *Leopard* conservatively treats the operations as non-conflicting.

Moreover, if a predicate is too complex to compute the region (e.g., including user-defined functions), *Leopard* conservatively classifies it into the fourth category mentioned above (overlapping but non-subsuming). That is, these operations are considered to access different records. As a result, some dependencies might be missed, which might further lead to more false negatives in isolation bug finding.

However, in practice, most identifications can be accomplished based on the first three categories, in which the records can be precisely identified. This is because most of the transactional operations access records via equality predicates. For example, in TPC-C [7], only one operation template contains a range predicate, and such operations occur with a frequency of fewer than 8.4×10^{-4}; all operations in SmallBank [8] only contain equality predicates. In such cases, the records are easily identified by the predicates.

Note, although a predicate region might contain multiple records, it is compatible with the record-level locks used in the *mutual exclusion* (*ME*) mechanism. Specifically, each predicate region serves as an identifier and can be viewed as a "virtual record", to which record-level locks are applied. Then, operations that acquire record-level locks on the same identifier are verified by the *ME* mechanism. In addition, when one identifier subsumes another, the former must contain all records included by the latter. As a result, the locks on them are mutually exclusive, and operations acquiring these locks are also verified by the *ME* mechanism. However, in practice, such cases are rare because most transactional operations use equality predicates, which generally produce identifiers that are either identical or disjoint from others, rather than subsuming any other identifier.

[1] P. Alvaro and K. Kingsbury, "Elle: Inferring isolation anomalies from experimental observations." VLDB, pp. 268-280, 2020.

[2] C. Tan, C. Zhao, S. Mu, and M. Walfish, "Cobra: Making transactional key-value stores verifiably serializable." OSDI, pp. 63–80, 2020.

[3] J. Zhang, Y. Ji, S. Mu, and C. Tan, "Viper: A fast snapshot isolation checker." EuroSys, pp. 654–671, 2023.
[4] Z.-M. Jiang, S. Liu, M. Rigger, and Z. Su, "Detecting transactional bugs in database engines via graph-based oracle construction." OSDI, pp. 397–417, 2023.
[5] H. Xu, S. Liu, X. Zhu, Q. Zhuang, W. Lu, and X. Du, "Anomaly pattern-guided transaction bug testing in relational databases." arXiv:2511.17377, 2025.
[6] F. Hausdorff, "Set Theory." vol. 119. American Mathematical Society, 2021.
[7] "TPC-C benchmark" http://www.tpc.org/tpcc/.
[8] M. Alomari, M. Cahill et al., "The cost of serializability on platforms that use snapshot isolation." ICDE, pp. 576–585, 2008.

C. Complexity Analysis

Complexity Analysis of Algorithm 1. We denote $|\mathbb{L}|$ and $|\mathcal{L}|$ as the number of local buffers and the size of traces in each local buffer, respectively. Intuitively, the space complexity of local buffers is $|\mathbb{L}| \cdot |\mathcal{L}|$. For initializing global buffer, we fetch all traces of local buffers into the global buffer. After that, taking advantage of adaptive fetching, we guarantee that the traces fetched to the global buffer have the same size as the dispatched traces. In such a way, the space complexity of global buffer equals that of local buffers, i.e., $|\mathbb{L}| \cdot |\mathcal{L}|$. In summary, the space complexity of *two-level pipeline* is $O(2 \cdot |\mathbb{L}| \cdot |\mathcal{L}|)$. Support B is the average batch size, the number of nodes in the global buffer can be reduced to $\frac{|\mathbb{L}| \cdot |\mathcal{L}|}{B}$. Meanwhile, since we can dispatch a batch of trace at once, the time complexity is $O(\frac{1}{B} \cdot log(\frac{|\mathbb{L}| \cdot |\mathcal{L}|}{B}))$.

Complexity Analysis of *CR* Verification. The time complexity of a read operation in our *CR* verification is $O\left(n_r \cdot n_v\right)$, where n_r is the average number of versions in the read set of an operation and n_v is the average number of record versions. Note that, the construction of ordered versions is carried out by the write operations. Specifically, the ordered versions of each record are stored as a linked list, and each record version is added to the list by insertion sort. Thus, the time complexity of adding a version is $O(n_v)$. The space complexity is $O(n_x \cdot n_v)$, where n_x is the total number of recently accessed records. We observe that n_v and n_x would increase as the workload continuously creates new versions and records. To alleviate this issue, we propose to asynchronously prune garbage versions and records that do not conflict with current active transactions.

Complexity Analysis of *ME* Verification. The time complexity of a lock releasing (or acquiring) operation in *ME* verification is $O(n_l \cdot n_t)$, where n_l is the average number of locks released (or acquired) by an operation and n_t is the average number of conflicted locks on each record. Specifically, the lock acquiring and releasing time intervals of each record are stored as a sorted linked list in the lock table, and each time interval is added to or removed from the list by insertion sort. Thus, the time complexity of updating the lock table for an operation is $O(n_l \cdot n_t)$. Further, as the lock acquiring operation must happen before lock releasing operation, there are at most 4 possible orders of the lock operations, so the cost of enumerating all possible orders can be ignored. The space complexity is $O(n_x^l \cdot n_t)$, where n_x^l is the total number of recently locked records. Similar to the *CR* process, we asynchronously prune the locks of transactions that do not conflict with active transactions.

Complexity Analysis of *FUW* Verification. The time complexity of our *FUW* verification is $O(n_w \cdot n_v)$, where n_w is the average number of versions in the write set of an operation and n_v is the average number of record versions. For the space complexity, it uses the ordered version lists maintained by the *CR* verification mechanism and does not incur extra space cost.

Complexity Analysis of *SC* Verification. The time complexity of verifying a trace $\mathcal{T}$ in *SC* is related to the implementation of the certifier inside the DBMS. For example, the time complexities of PostgreSQL and CockroachDB are $O(d)$, where d is the average degree of each node in *DG*. This is because they only need to track two consecutive rw dependencies or check whether a transaction with an older timestamp has a dependency on the transaction with a newer timestamp. The space complexity of *DG* is $O(n_t^2)$ where n_t is the number of transactions in *DG*.