Leopard: A Black-Box Approach for Efficiently Verifying Various Isolation Levels

Keqiang Li[1], Siyang Weng[1], Lyu Ni[1], Chengcheng Yang[1], Rong Zhang[1], Xuan Zhou[1], Jianghang Lou[2],
Gui Huang[2], Weining Qian[1], Aoying Zhou[1]
East China Normal Unversity[1], *Alibaba Group*[2]
{kqli, syweng}@stu.ecnu.edu.cn, {lni, ccyang, rzhang, xzhou, wnqian, ayzhou}@dase.ecnu.edu.cn,
{jianghang.loujh, qushan}@alibaba-inc.com

Abstract—**Isolation Levels (IL) act as correct contracts between applications and database management systems (DBMSs). The complex code logic and concurrent interactions among transactions make it a hard problem to expose violations of various ILs stated by DBMSs. With the recent proliferation of new DBMSs, especially the cloud ones, there is an urgent demand for a general way to verify various ILs. The core challenges come from the requirements of: (a) lightweight (verifying without modifying the application logic in workloads and the source code of DBMSs), (b) generality (verifying various ILs), and (c) efficiency (performing efficient verification on a long running workload). For lightweight, we propose to deduce transaction dependencies based on time intervals of operations collected from client-sides without touching the source code of DBMSs. For generality, based on a thorough analysis to existing concurrency control protocols, we summarize and abstract four mechanisms which can implement ILs in all commercial databases we have investigated. For efficiency, we design a *two-level pipeline* to organize and sort massive time intervals in a time and memory conservative way; we propose a *mechanism-mirrored verification* to simulate the concurrency control protocols implemented in DBMSs for high throughputs. *Leopard* outperforms existing methods by up to $114\times$ in verification time with a relative small memory usage. In practice, *Leopard* has a superpower to verify various ILs on any workload running on all commercial DBMSs. Moreover, it has successfully discovered 17 bugs that can not be found by other existing methods.**

I. Introduction

The concept of isolation level (IL) was first introduced in [1] with the name "degrees of consistency". IL serves as a correctness contract between applications and DBMSs. The strongest IL is serializable, but it usually exhibits a relative poor performance. A weak IL, on the other hand, offers better performance but sacrificing the guarantees of a perfect isolation. For example, snapshot isolation allows some data corruptions to facilitate a high concurrency [2]. Thus, commercial DBMSs support various ILs to provide applications with a trade-off between consistency and performance [3].

Although theoretical correctness proofs have been provided for almost all ILs, the practical implementations might not strictly follow the definitions [4], [5]. Moreover, to provide both serializability and scalability in distributed systems, the distributed DBMS usually uses a consensus protocol to interact with the atomic commit protocol [6]–[8], which would lead to an extraordinarily complex protocol combinations over multiple remote machines [9]–[11]. This complexity also result in a number of subtle bugs [12]–[16]. Thus, it is significantly important to perform a thorough verification of ILs when deploying critical applications on a DBMS. However, verifying various ILs has always been a tough work [12], [17]–[24] and the key challenges are summarized as follows:

Black-box Verification (C1). Most concurrency control protocols are non-deterministic, which means that the final database state might be different even given the same input [25]. Thus, the traditional differential testing method [26] which checks the final database state is impractical for IL verification. Existing studies can be broadly classified into kernel-oriented and workload-oriented methods. The kernel-oriented methods usually instrument the kernels to catch the internal execution state of DBMSs [24], [27]–[32]. However, modifying the kernels is laborious or even impossible, especially for cloud service provided by the third party. The workload-oriented methods rely on specific workloads to make transaction dependencies easy to obtain [12], [17]–[20]. However, these methods have a limited scope of applications, and might neglect some subtle bugs in other critical application scenarios. Thus, it is more attractive to design a black-box verifying method which is independent of kernels and workloads. However, verifying ILs in a black-box mode has been proved to be an NP-complete problem [33], [34].

Various Isolation Levels (C2). There are various ILs in DBMSs [35]–[39]. Even for the same IL, there might exist some subtle differences between different DBMSs. The main reason is that different DBMSs might combine different concurrency control protocols to implement these ILs. Take repeatable read as an example, InnoDB [40] allows lost update anomalies in this level of isolation, while PostgreSQL [41] and Oracle [42] do not since they use the *first updater wins* mechanism in their implementations [43]. Considering such a variety of ILs, it is rather laborious to verify ILs one by one.The differences between ILs increase the complexity of designing a general approach for IL verification. Most of the existing work [12], [44] has been designed to only verify the strongest IL, i.e., serializable. Although *Elle* [20] has tried to find diffrent IL anomalies, it could not distinguish the repeatable read and serializable in PostgreSQL [16].

Efficient Verification (C3). Many applications are suffering from isolation-related bugs that might cause data corruptions, and they might be exploited by determined adversaries to make some mischief and profits [4], [5]. Therefore, it is necessary to timely verify each transaction executed on a DBMS, such that bugs can be reported and fixed as soon as possible. However, the online transaction processing always continuously runs with a high throughput, then the produced massive transactions pose a significant challenge on the efficiency of IL verification. We observe that previous studies often fail to preform an efficient IL verification on a long running workload. For example, the verification time of *Cobra* [12] grows superlinearly with the number of transaction. *Elle* verifies IL using a specified workload in an offline way.

We propose *Leopard* to address above challenges. It exhibits excellent properties of (a) *lightweight* (doing verification in a black-box mode), (b) *generality* (verifying various ILs), and (c) *efficiency* (achieving efficient verification even for a long running workload). To address **C1**, we propose to collect *interval-based traces* from client-sides without touching any source code of DBMS or changing application logic. This method is workload-insensitive and can be applied to any DBMS. Specifically, the traces contain the execution time interval of each database operation and can be leveraged by our verification method to deduce the operation orders and transaction dependencies. To address **C2**, we summarize and abstract the implementation of various ILs into four classic mechanisms, including *consistent read*, *first updater wins*, *serialization certifier* and *mutual exclusion*. These four mechanisms constitute all ILs in commercial DBMSs we have investigated. Thus our method can be generally applied to various IL verification. To address **C3**, we first design a *two-level pipeline* to sort massive streaming traces produced by a running workload. Based on the sorted traces, we propose *mechanism-mirrored verification*, which directly simulates the internal processing of DBMS to verifying the four mechanisms. In this way, the verification process can catch up with the performance of DBMSs. Additionally, we also design some garbage collection methods to remove unnecessary structures to reduce the memory usage. In summary, we make the following contributions.

1) We are the first work to design a lightweight IL verification framework for verifying ILs in a black-box mode.
2) We summarize and abstract the implementation of various ILs in commercial DBMSs into four mechanisms.
3) We design *two-level pipeline* and *mechanism-mirrored verification* methods to provide efficient IL verification.
4) We launch extensive experiments and show that *Leopard* outperforms existing methods by up to 114× in verification time with a relative small memory usage. Moreover, we have successfully found 17 bugs in commercial DBMSs which existing methods could not find.

II. Background

In this section, we first introduce the notations used in our paper. Then, we abstract and summarize four mechanisms to implement isolation levels (ILs) of commercial DBMSs.

A. Transaction Dependencies

A database $\mathbb{D}$ has a set of records. A transaction t consists of several operations typed read or write, ended with either commit or abort as a terminal operation. A write creates a new version for a record while a read queries a specific database snapshot. A database snapshot is consistent with versions created at the time of the snapshot creation. A commit installs all versions created by a transaction while an abort discards them. For a transaction t, we denote $r_t(rs)$ as a read in t with its read set rs, and denote $w_t(ws)$ as a write in t with its write set ws. Each element in rs (resp. ws) is an accessed version by r (resp. w). We denote x^i as the i^{th} version of record x, and x^{i+1} as its direct successor version.

ILs define the degree to which a transaction must be isolated from the data modifications made by any other transaction. As an isolation anomaly can be indicated by a specific transaction dependency pattern [37], the first step of IL verification is to get dependencies between all transactions. In general, there are three kinds transaction dependencies between any two committed transactions (denoted as t_m and t_n): 1) If t_m installs a version x^i and t_n installs x^i's direct successor x^{i+1}, t_n has a **direct write-dependency** (ww) on t_m. 2) If t_m installs a version x^i and t_n reads x^i, t_n has a **direct read-dependency** (wr) on t_m. 3) If t_m reads a version x^i and t_n installs x^i's direct successor version x^{i+1}, t_n has a **direct anti-dependency** (rw) on t_m.

B. Isolation Level Implementations

Various ILs exist in DBMSs [35]–[39], which prohibit isolation anomalies by different concurrency control protocols including *optimistic concurrency control* (*OCC*) and *two-phase locking* (*2PL*), *multi-version concurrency control* (*MVCC*), *serializable snapshot isolation* (*SSI*) and *timestamp ordering* (*TO*), etc. However, as Fig. 1 shows, even for the same IL, DBMSs might mix up multiple concurrency control protocols in different ways, which results in some subtle differences.

Although there exist various implementations of ILs, we discover that almost all concurrency control protocols in commercial DBMSs can be implemented by assembling the following four classic mechanisms: *consistent read* (*CR*), *first updater wins* (*FUW*), *serialization certifier* (*SC*) and *mutual exclusion* (*ME*). In Fig. 1, we summarize the implementations of ILs in popular commercial DBMSs. For example, *serializable* in PostgreSQL takes all the above four mechanisms to eliminate isolation anomalies while *repeatable read* (*RC*) may suffer from *write skew* that is a kind of isolation anomaly prohibited by *SC* [41]. Note that there still exist some other mechanisms [25], [45]–[49], almost all of which stay only in the academic papers instead of in practical products.

Consistent read (*CR*) provides a consistent view of the database at a specific time point. For example, *MVCC* takes *CR* to generate a snapshot of the database by maintaining multiple physical versions of each record. Specifically, a read in *MVCC* sees the changes made by earlier operations within the same transaction and the changes made by other transactions committed before a specific time point. To provide

DBMS	Concurrency Control	IL	ME	CR	FUW	SC
PostgreSQL [6], OpenGauss [18]	2PL+MVCC +SSI [6]	SR	✓	✓	✓	✓
		SI	✓	✓	✓	
		RC	✓	✓		
InnoDB [19], Aurora [20], PolarDB [21], SQL server [22]	2PL+MVCC	SR,RR, RC	✓	✓		
TiDB [23]	2PL+MVCC	RR,RC	✓	✓		
	Percolator [24]	SI		✓		✓
RocksDB [25]	2PL+MVCC	SR	✓	✓		
	OCC+MVCC	SR		✓		✓
SQLite [26]	2PL	SR	✓			
FoundationDB [27]	OCC+MVCC	SR		✓		✓
SingleStore [28]	2PL+MVCC	RC	✓	✓		
CockroachDB [29]	TO+MVCC	SR		✓		✓
Spanner [30]	2PL+MVCC	SR	✓	✓		
yugabyteDB [31]	2PL+MVCC	SR,RR,RC	✓	✓	✓	✓
Oracle [14], NuoDB [15], SAP HANA [16]	2PL+MVCC	SI	✓	✓	✓	
		RC	✓	✓		

Fig. 1. Isolation Level Implementations in DBMSs. RC: Read Committed, RR: Repeatable Read, SR: Serializable, SI: Snapshot Isolation.

different ILs, there exist transaction-level *CR* and statement-level *CR*, which provide a consistent view of the database at the beginning of a transaction and an operation, respectively.

Mutual exclusion (*ME*) uses the locking strategy to provide a kind of exclusive access to a shared resource. *2PL* takes *ME* to generate serializable histories by detecting conflicts and delaying the conflicting transactions. Specifically, for every transaction following *2PL*, a phase during which locks are acquired is distinguished from and strictly followed by a phase during which locks are released.

First updater wins (*FUW*) ensures the first transaction committing a write to the record x successes. Then, to avoid lost update anomalies, all the other concurrent transactions writing x would abort. DBMSs often combine *CR* with *FUW* to guarantee the snapshot isolation [41], [50]–[52].

Serialization certifier (*SC*) guarantees schedules are conflict serializability [53], which is the most important one for the practice of DBMSs. For example, *OCC* executes a transaction in three phases: read, validation and write. Specifically, in the validation phase, *OCC* takes *SC* to generate serializable schedules by aborting the transaction conflicting with other active ones. *TO* takes *SC* to generate serializable schedules by ordering the transactions on timestamps. *SSI* takes *SC* to avoid *write skew* that violates the *serializable* [36]. Details to verify these four mechanisms are presented in Section V.

III. Leopard Framework

The framework of *Leopard* is illustrated in Fig. 2, which contains two components, i.e., *Tracer* and *Verifier*.

Tracer. *Tracer* continuously collects traces from each client connected to the DBMS in a black-box mode. It collects the invocation and completion timestamps of each operation and does not modify any application logic. Thus, it is workload-insensitive and generally applied to any DBMS. To help *Verifier* deduce the operation orders and transaction dependencies, *Tracer* needs to sort the traces according to their timestamps. As each client continuously generates its own traces individually, one naive idea is to use a min-heap to store exact one trace for each client. To dispatch a trace to *Verifier*, it pops the heap and gets the trace with globally smallest timestamp. In addition, it also fetches one new trace from the client which generates the popped trace previously. However, this would lead to high synchronization and communication costs between the clients and min-heap. *Tracer* proposes to use a *two-level pipeline* which consists of local and global buffers to address this issue. As Fig. 2 shows, the local buffers asynchronously buffer traces from each client and slice the traces into batches. The global buffer then batch fetches traces from local buffers into its min-heap round by round. Moreover, it uses a watermark to coordinate the order of trace fetching between local buffers. The watermark can also help control the size of min-heap and reduce the heap maintaining cost.

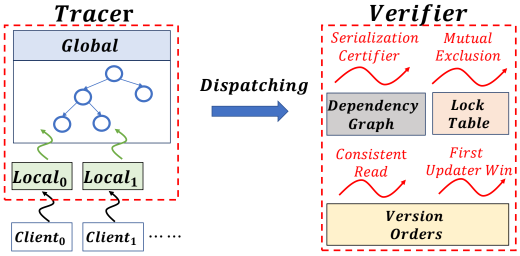


Fig. 2. Leopard Framework

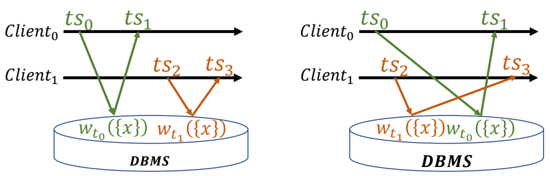
Verifier. DBMSs usually exhibit a high throughput with massive operations executed in a short period of time. As a result, it poses a significant challenge on efficient IL verification. To improve the efficiency of IL verification, previous studies usually focus on optimizing the cycle searching process on the dependency graph, such as splitting the graph into isolated segments [12] and sampling representative subsets from the graph [22]. Unfortunately, due to the inherent high complexity of cycle searching on a large graph, these methods fail to verify each operation in an efficient fashion. To this end, *Verifier* proposes to directly simulate the workflow of concurrency control protocols inside the DBMS. The main reason is that the time spent in concurrency control is much less than other components, such as the query execution and disk access. *Verifier* abstracts the implementation of various ILs into four mechanisms. In this way, verifying various ILs is equivalent to verifying the four mechanisms. Specifically, *Verifier* tries to mirror the internal states of the DBMS, such as the version orders, lock table and dependency graph. To this end, it processes traces the same as the operation processing of DBMS and executes each dispatched trace on these internal states. Then, *Verifier* uses these internal states to check whether there exists a violation of the four mechanisms. In addition, the structures of these internal states would accumulate as time goes on. To save the memory usage, *Verifier* periodically prunes the obsolete structures that are not involved in the active transactions.

IV. Trace Management

In this section, we first introduce the concept of *interval-based trace* (Section IV-A). Then we discuss the opportunity of deducing transaction dependencies in a black-box mode (Section IV-B). Finally, we describe how to efficiently sort massive traces before dispatching them to *Verifier* (Section IV-C).

A. Interval-based Trace

To perform IL verification in a black-box mode, we log the time interval of each operation in all clients connected to the



(a) No Overlap (b) Overlap

Fig. 3. All Cases between Traces of Two Conflicting Operations

DBMS. The trace of an operation consists of 1) the timestamp before the operation executed ts_{bef}; 2) the timestamp after the operation executed ts_{aft}; 3) operation type and the data touched by the operation. Specifically, for a read (resp. write) operation, we log its belonging transaction t and its read set rs (resp. write set ws). For a commit/abort operation, we only log its belonging transaction t. Thus, the trace logging process does not need to modify the application logic and DBMS kernel. We formalize the trace for an operation by $\mathcal{T} = \{ts_{bef}, ts_{aft}, r_t(rs)/w_t(ws)/a_t/c_t\}$.

Note that, the timestamps appearing in traces require clock synchronizations. If the workload runs on a single machine, we use its hardware time for clock synchronizations. For distributed databases, we log traces with a software-level clock synchronization service such as *Network Time Protocol* and *Amazon Time Sync Service* [7], which can provide a highly accurate and reliable time reference.

B. Deducing Transaction Dependencies

As each IL has a specific restriction on the allowed dependency patterns during transaction processing [37], it is critical to determine the dependencies between transactions. The interval-based traces provide an opportunity to do this because each trace represents a specific operation issued to the DBMS. For example, consider the two conflicting operations $w_{t_0}(\{x\})$ and $w_{t_1}(\{x\})$ in Fig. 3. Both of them create a new version on the record x. Suppose the traces of the two operations are $\mathcal{T}_0 = \{ts_0, ts_1, w_{t_0}(\{x\})\}$ and $\mathcal{T}_1 = \{ts_2, ts_3, w_{t_1}(\{x\})\}$. If the time intervals of the two traces do not overlap (see Fig. 3(a)), we can deduce that t_1 has a ww dependency on t_0. Otherwise, we can not determine the dependencies between t_0 and t_1 since the exact time points of the two write operations are not available. Therefore, the overlapped traces would lead to uncertain dependencies.

However, the main operations in a transactional workload are item reads/writes and the time intervals of their traces are usually very short. This indicates that the time intervals are not likely to be overlapped. To explore it practically, we run a standard benchmark *YCSB-A* [54] on PostgreSQL with a single table and 1 million records. We vary the skew parameter θ, the thread scale, and the read/write ratio to simulate different contentions and different ratios of the three dependencies. Here, we define $\beta = B/A$ to represent the ratio of uncertain dependencies, where A is the total number of actual dependencies and B is the number of uncertain dependencies due to overlapped trace time intervals. As shown in Fig. 4, the skew parameter and thread scale are the major factors which affect the ratio β. This indicates that a high contention between operations would lead to a high ratio of uncertain dependencies. Moreover, we also observe that the value of β keeps relative small (below 6%) in all cases. This indicates that most of the dependencies can be directly deduced by the interval-based traces. In Section V, we will take advantage of the four verifying mechanisms to further eliminate uncertain dependencies.

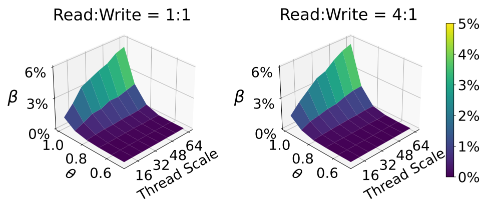


Fig. 4. Overlapping Ratio β in YCSB-A

C. Two-Level Pipeline

The trace sorting procedure is critical for determining the operation orders and transaction dependencies. As sorting either by before timestamp or after timestamp can help check whether two time intervals overlap, we sort all traces by the before timestamp in this paper. To sort the massive traces continuously generated by multiple clients in an online fashion, we design a *two-level pipeline* which consists of several local buffers and a global buffer. The local buffers cache the streaming traces from each client asynchronously. In the meanwhile, the global buffer fetches and sorts all traces from each local buffer. The global buffer is organized as a min-heap whose time complexity increases logarithmically with the number of traces. It also uses a watermark to coordinate the order of the traces fetching from the local buffers and control the size of global buffer. Specifically, the watermark is set as the smallest before timestamp among all traces in local buffers.

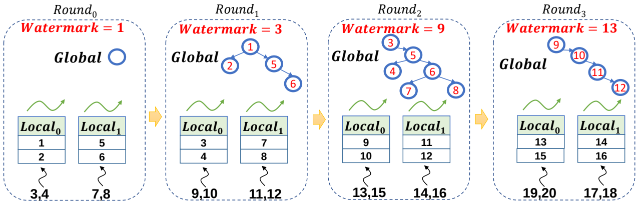


Fig. 5. Example for Dispatching Trace

Based on the *two-level pipeline*, we design a round-by-round algorithm to dispatch traces. As shown in Algorithm 1, each round consists of four stages: (a) the global buffer dispatches the traces whose before timestamps are less than watermark to *Verifier* (lines 2, 8); (b) the global buffer fetches traces from the local buffers (lines 4 $\sim$ 5); (c) the global buffer updates the watermark (line 7). Note that, the traces in each local buffer are naturally sorted since the traces are generated in increasing order of before timestamps in each client; (d) each client pushes traces into its corresponding local buffer (*line* 6). Theorem 1 proves that the *two-level pipeline* is guaranteed to dispatch ordered traces to *Verifier*.

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

Proof 1: Supposing we have n_{local} local buffers and W is the watermark that is the smallest before timestamp among all traces in local buffers. According to Algorithm 1, the dispatched trace $\mathcal{T}$ satisfies:

Algorithm 1 Dispatching Trace

Input: n_{local} : the number of local buffers; $local_i$: the i^{th} local buffer;

Output: a trace dispatched to *Verifier*.

```
procedure DISPATCH()
    $\mathcal{T}$ = global.top()
    while $\mathcal{T}$ == null or $\mathcal{T}.ts_{bef} > watermark$ do
        for i=0 to $n_{local} - 1$ do
            fetch all of traces in $local_i$ and sort them in global buffer;
            push the traces produced by client $i$ into $local_i$;
        $watermark = \min_{0 \leq i < n_{local}} local_i[0].ts_{bef}$
    return $\mathcal{T}$
```

$$\mathcal{T}.ts_{bef} \leq \min_{\mathcal{T}_i \in global} \mathcal{T}_i.ts_{bef} \tag{1}$$

$$\mathcal{T}.ts_{bef} \leq W = \min_{0 \leq i \leq n_{local}-1} local_i[0].ts_{bef} \tag{2}$$

Since the traces in each client C_i are generated in an increasing order of before timestamps, then we have:

$$\begin{aligned} local_i[0].ts_{bef} &\leq \min_{\mathcal{T}_j \in local_i} \mathcal{T}_j.ts_{bef} \\ &\leq \min_{\mathcal{T}_j \in C_i} \mathcal{T}_j.ts_{bef}, 0 \leq i \leq n_{local} - 1 \end{aligned} \tag{3}$$

From Equations (1)-(3), we can infer that the dispatched trace has the minimum before timestamp among all traces in the global buffer, local buffers and clients. Thus theorem is proved.

Figure 5 shows a running example based on two clients. In $Round_0$, the global buffer and watermark are initialized as $\varnothing$ and 1. Then, the two clients push their collected traces into the local buffers. In $Round_1$, as there exist no traces in the global buffer, it does not dispatch any trace to *Verifier*. Then, the global buffer fetches traces from each local buffer and sorts them with the min-heap. After that, the clients fill the empty local buffers with their newly collected traces. Finally, the watermark is set as the smallest before timestamp of the two local buffers, which is 3. In $Round_2$, the global buffer first dispatches all traces whose before timestamps are smaller than the watermark to *Verifier*, which are trace 1 and 3. Then, it repeats the three steps in $Round_1$, i.e., fetching traces 3, 4, 7 and 8 into the global buffer, pushing traces 9, 10, 11 and 12 into the local buffers and setting watermark as 9. Similarly, in $Round_3$, it repeats the four steps as described in $Round_2$.

Algorithm 2 Mechanism-mirrored Verification

Input: *op*: operation; $\mathcal{T}$: the trace of *op*; *traces*: the trace set dispatched from the *Tracer*.

Output: *bug descriptor*

```
procedure CONSISTENTREAD
    $\mathcal{S}^{\mathcal{T}} \leftarrow$ the snapshot generation time interval of op;
    $OV \leftarrow$ use traces to construct ordered versions of each record;
    $CV^{\mathcal{T}} \leftarrow candidate\_version\_set(OV, \mathcal{S}^{\mathcal{T}})$
    for each version $x^i$ in the read set of op do
        if $x^i \notin CV^{\mathcal{T}}$ then
            return a CR violation in bug descriptor;
        if there exists only one version in $CV^{\mathcal{T}}$ matches $x^i$ then
            Deduce a $wr$ dependency;
procedure MUTUALEXCLUSION
    $LT \leftarrow$ use traces to construct a lock table for each record;
    for each lock $l_i$ released by op do
        for each conflicting lock $l_j$ in $LT$ do
            if each of possible orders indicates incompatible locks then
                return an ME violation in bug descriptor;
            else
                Deduce a $ww$ dependency;
procedure FIRSTUPDATERWINS
    $\mathcal{S}^{\mathcal{T}} \leftarrow$ the snapshot generation time interval of op;
    $OV \leftarrow$ use traces to construct ordered versions of each record;
    for each version $x^i$ in the write set of op do
        for each version $x^j$ in $OV$ do
            if each of possible orders indicates concurrent versions then
                return an FUW violation in bug descriptor
            else
                Deduce a $ww$ dependency;
procedure SERIALIZATIONCERTIFIER
    for each dependency $d$ deduced from $\mathcal{T}$ do
        $DG \leftarrow DG \cup \{d\}$
        if $d$ causes a prohibited dependency pattern then
            return an SC violation in bug descriptor
```

Optimizations. The performance of trace sorting depends on the number of traces in the global buffer. We observe that the distribution of timestamps in each local buffer has a significant impact on the size of global buffer. For example, if the operations in a client are executed much slower than other clients, then the timestamps in the corresponding log buffer are much smaller than that of others. As a result, it would hinder the increase of watermark, and the size of global buffer will keep increasing due to the accumulation of traces which are fetched from other local buffers. To address this issue, we prefer to fetch traces from the local buffer with the smallest timestamp. Further, to keep the size of global buffer stable, we always restrict that the speed of trace dispatching is equivalent to the speed of trace fetching.

Complexity Analysis. The space complexity is $O(n_{local} \cdot (s_{local} + s_{client}))$, where s_{local} is the size of each local buffer and s_{client} is the size of traces temporally stored in each client. The time complexity of each dispatched trace is $O(log(n_{local} \cdot s_{local}))$.

V. Isolation Level Verification

In this section, based on the dispatched interval-based traces, we introduce our *mechanism-mirrored verification* to verify the four classic implementation mechanisms.

A. Verifying Consistent Read

Consistent read (*CR*) provides a consistent view of the database at a specific time point. In general, there exist two cases for the *CR* verification. In the first case, an operation sees the changes made by earlier operations within the same transaction. In the second case, an operation sees the visible changes made by other transactions. Specifically, the second case can be further classified into the transaction-level and statement-level consistent reads. The transaction-level consistent read generates the snapshot of a database at the beginning of a transaction, while the statement-level consistent read generates the snapshot at the beginning of an operation.

As we could not obtain the exact time point of each operation under the black-box mode, we propose a time interval based verification approach, which leverages the snapshot generation time interval of each operation and potential version evolution of each record to guide the *CR* verification. We first formally define the version installation and snapshot generation time intervals as follows.

Definition 1: **Version Installation Time Interval.** Suppose a write operation creates a new version and $\mathcal{T}$ is the operation trace. Then, the version installation time interval indicated by $\mathcal{T}$ is defined as $\mathcal{V}^{\mathcal{T}} = (\mathcal{T}.ts_{bef}, \mathcal{T}.ts_{aft})$.

Definition 2: **Snapshot Generation Time Interval.** Suppose an operation op reads a snapshot of the database which is generated during the execution of operation op_s. Then, the generation time interval of the snapshot read by op is defined as $\mathcal{S}^{\mathcal{T}} = (\mathcal{T}_s.ts_{bef}, \mathcal{T}_s.ts_{aft})$, where $\mathcal{T}$ and $\mathcal{T}_s$ are traces of op and op_s, respectively.

The *version installation time interval* contains the exact time point when a version is created. For example, consider the trace $\mathcal{T} = \{ts_0, ts_1, w_{t_0}\}$. It indicates that a write operation in transaction t_0 creates a new version between ts_0 and ts_1, then we have $\mathcal{V}^{\mathcal{T}} = (ts_0, ts_1)$. Additionally, the *snapshot generation time interval* contains the exact time point when a specific snapshot is generated. For example, suppose there exist two consecutive reads r_t and r'_t in a transaction t. $\mathcal{T} = \{ts_0, ts_1, r_t\}$ and $\mathcal{T}' = \{ts_2, ts_3, r'_t\}$ are their corresponding traces. In transaction-level consistent read, the snapshot is generated by an operation at the beginning of a transaction.Thus, both $\mathcal{S}^{\mathcal{T}}$ and $\mathcal{S}^{\mathcal{T}'}$ are set as (ts_0, ts_1). However, in statement-level consistent read, the snapshot is generated at the beginning of each operation. Thus, $\mathcal{S}^{\mathcal{T}}$ and $\mathcal{S}^{\mathcal{T}'}$ are set as (ts_0, ts_1) and (ts_2, ts_3), respectively.

Next we discuss how to find *CR* violations based on the two kinds of time intervals. If all the *version installation* and *snapshot generation time intervals* do not overlap with each other, then we can directly determine the visibility of each version to a given read operation. Let's consider the first case in *CR* verification. There exists no time interval overlaps in the same transaction. Then, the *CR* mechanism can be verified by checking if a read operation sees a version which should be invisible to it. However, the read and write operations are often executed parallelly inside a DBMS, which would lead to a certain number of overlapped time intervals (see Fig.4).

To address this issue, we propose to leverage the snapshot generation time interval of a read operation to find a "candidate version set" to help with the *CR* verification process. Specifically, the candidate version set contains all the versions that are possibly visible to the read operation. If there exists no version in the candidate version set matching the read set of a read operation, we can confirm that a *CR* violation occurs. Intuitively, the size of candidate version set has a critical impact on the effectiveness of our *CR* verification. The smaller the size is, the stricter checks it poses for the *CR* verification, and the higher probability a *CR* violation can be detected.

Lastly, we show how to find a minimal candidate version set. In order to efficiently prune the versions that must be invisible to the read operation, we first classify the versions into five categories according to the relationships between version installation and snapshot generation time intervals.

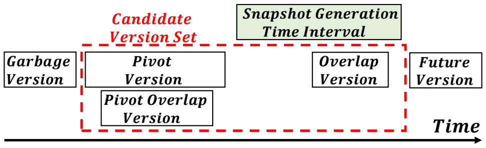


Fig. 6. Minimizing Candidate Versions

1) *Future version*. The version whose version installation time interval appears after the snapshot generation time interval.
2) *Overlap version*. The version whose version installation time interval overlaps with the snapshot generation time interval.
3) *Pivot version*. The version whose version installation time interval appears before the snapshot generation time interval, and its after timestamp is closest to the before timestamp of snapshot generation time interval.
4) *Pivot overlap version*. The version whose version installation time interval overlaps with that of the pivot version.
5) *Garbage version*. The version whose version installation time interval appears before that of the pivot version.

Fig. 6 demonstrates which versions should be included in the candidate version set. We can see that it only consists of the *overlap versions*, *pivot versions* and *pivot overlap versions* whose installation time intervals are close to the snapshot generation time interval. Specifically, the *future version* arrives after the snapshot generation time interval, so it is invisible to the read operation. In contrast, the *garbage version* arrives before the snapshot generation time interval. However, it will be overwritten by the versions in the candidate version set. Thus, it must be invisible to the read operation. For the rest three versions, as their exact arrive times are not available in hand, either one of the three versions might be seen by the read operation. For example, if the *overlap version* arrives before the read operation, but after the *pivot version* and *pivot overlap version*, then the read operation sees the *overlap version*. As another example, the read operation sees the *pivot overlap version* (resp. *pivot version*) if that version arrives after the *pivot version* (resp. *pivot overlap version*) and the *overlap version* arrives after the read operation. We proceed to show the effectiveness of our *CR* verification approach. Theorem 2 proves our approach finds a minimal set of candidate versions.

Theorem 2: The candidate version set contains a minimum number of versions that are possibly visible to a given read operation.

Proof 2: Suppose a version x^i falls into the candidate version set but is impossibly visible to a given read operation. There exist two cases if x^i must be invisible to the read operation. In the first case, the version x^i appears after the read operation. Then, we can infer that the version installation time interval of x^i must not overlap with the snapshot generation time interval. Otherwise, x^i is possibly visible to the read operation since we could not determine the chronological order of the exact read operation time point and version installation time point. This implies that 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 arrive 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.

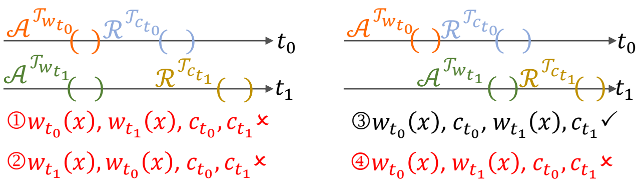
Since our approach excludes all the future versions *and* garbage versions *from the candidate version set, this is contradicted with the initial assumption. The theorem is proved.*

Additionally, if the candidate version set has only one version matching the read set of a read, then we can deduce the read transaction has a wr dependency on the write transaction creating this version. That is, although the time intervals of the two operations are overlapped, we can still confirm that the write must happen before the read. The dependencies deduced in a specific mechanism can be used by other ones to improve the effectiveness of their *IL* verification. For example, the ww dependencies deduced in the *mutual exclusive* or *first update wins* mechanism (see Sections V-B and V-C) can help the *CR* verification determine the installation order of versions with overlapped time intervals. With ordered versions, only the *overlap versions* and the last version among the *pivot overlap versions* and *pivot versions* need to be added to the candidate version set. Take the *serialization certifier* as another example, it first uses the dependencies deduced in other mechanisms to build a dependency graph, and then detects *SC* violation by checking whether an invalid dependency pattern exists.

For the reasons above, in our implementation, we verify the four mechanisms in parallel and continuously transfer the deduced dependencies between them. Note that, the dependencies deduced in each mechanism is based on the assumption that there exist no bugs inside the DBMS. However, with the cooperation of the four verification mechanisms, the potential bugs hidden inside the DBMS are likely to deduce several contradictory dependencies. Then, they would be identified as a violation of isolation levels in the bug descriptor.

The pseudo-code of the *CR* verification method is shown in Algorithm 2. Given a read operation *op*, it first uses *op*'s trace $\mathcal{T}$ to get the snapshot generation time interval (line 2). Then, it uses the traces dispatched from *Tracer* to construct the ordered versions of each record, which are sorted according to the after timestamp of their corresponding version installation time intervals (line 3). With the help of ordered versions, it generates a minimal candidate version set $CV^{\mathcal{T}}$ that contains all the versions possibly visible to *op* (line 4). Next, for each version x^i in the read set of *op*, it checks whether there exists a version in $CV^{\mathcal{T}}$ matches x^i. If not, it reports a *CR* violation in the bug descriptor (lines 6 $\sim$ 7). Otherwise, if there exists only one match, it can deduce that *op* has a *wr* dependency on the write which creates that version (lines 8 $\sim$ 9).

Complexity Analysis. The time complexity of a read operation in our *CR* verification is $O(n_r \cdot n_v)$, 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 for each record is carried out by the write operations. Specifically, the ordered versions of each record are stored as a sorted linked list, and each record version created by a write operation is added to the list with the insertion sort method. Thus, the time complexity of a write operation is $O(n_w \cdot n_v)$, where n_w is the average number of versions in the write set of an operation. The space complexity is $O(n_x \cdot n_v)$, where n_x is the total number of recently accessed records. As a long-running workload might change its working set and continuously create new versions for each record, we observe that n_v and n_x would increase as time goes on. To alleviate this issue, we propose to asynchronously prune garbage versions and records that do not conflict with current active transactions.


$\mathcal{A}^{\mathcal{T}_{w_{t_0}}}$
$\mathcal{R}^{\mathcal{T}_{c_{t_0}}}$
t_0
$\mathcal{A}^{\mathcal{T}_{w_{t_1}}}$
$\mathcal{R}^{\mathcal{T}_{c_{t_1}}}$
t_1
① $w_{t_0}(x), w_{t_1}(x), c_{t_0}, c_{t_1}$ ×
② $w_{t_1}(x), w_{t_0}(x), c_{t_0}, c_{t_1}$ ×
$\mathcal{A}^{\mathcal{T}_{w_{t_0}}}$
$\mathcal{R}^{\mathcal{T}_{c_{t_0}}}$
t_0
$\mathcal{A}^{\mathcal{T}_{w_{t_1}}}$
$\mathcal{R}^{\mathcal{T}_{c_{t_1}}}$
t_1
③ $w_{t_0}(x), c_{t_0}, w_{t_1}(x), c_{t_1}$ ✓
④ $w_{t_0}(x), w_{t_1}(x), c_{t_0}, c_{t_1}$ ×

(a) Triggering a Violation (b) Deducing a ww Dependency

Fig. 7. Mutual Exclusion Verification Examples

B. Verifying Mutual Exclusion

Mutual exclusion (*ME*) uses the locking strategy to ensure exclusive accesses to the shared resources. An *ME* violation happens when a transaction acquires an incompatible lock holding by another transaction. Similar to the *CR* verification case, it is impractical to get the exact lock acquiring and releasing time in the black-box mode. We propose a time interval based approach to address this issue.

Definition 3: **Lock Acquiring and Releasing Time Intervals.** Suppose $\mathcal{T}$ (*resp.* $\mathcal{T}'$) is the trace of a lock acquiring (*resp.* releasing) operation. Then, the lock acquiring (*resp.* releasing) time interval indicated by $\mathcal{T}$ (*resp.* $\mathcal{T}'$) is defined as $\mathcal{A}^{\mathcal{T}} = (\mathcal{T}.ts_{bef}, \mathcal{T}.ts_{aft})$ (*resp.* $\mathcal{R}^{\mathcal{T}'} = (\mathcal{T}'.ts_{bef}, \mathcal{T}'.ts_{aft})$).

The lock acquiring and releasing time intervals contain the exact time points when a lock is acquired and released. On the one hand, if all the time intervals do not overlap with each other, then we can directly determine the order of lock acquiring and releasing on each record. On the other hand, for overlapped time intervals that are generally caused by parallel data accesses, we propose to leverage the mutual exclusion between locks to guide the verifying process.

Next, we discuss how to use time intervals to identify incompatible locks and deduce dependencies between transactions. Specifically, consider two transactions t_0 and t_1. Suppose they acquire two locks with operations $w_{t_0}(x)$ and $w_{t_1}(x)$, and then release the two locks with c_{t_0} and c_{t_1}. $\mathcal{T}_{w_{t_0}}$, $\mathcal{T}_{w_{t_1}}$, $\mathcal{T}_{c_{t_0}}$ and $\mathcal{T}_{c_{t_1}}$ are the traces of these four operations. As the exact lock acquiring and releasing time points are not available in the black box mode, there might exist multiple possible orders of lock operations for any given operation traces. We broadly classify these orders into two cases.

(1) If each of the possible orders of lock operations is identified to have incompatible locks, then we can infer that there must exist an *ME* violation inside the DBMS. For

example, consider the four lock operations in Fig. 7(a). There exist two possible orders of lock operations (① and ②). However, both of them are incompatible locks since a lock is acquired by two write operations simultaneously.

(2) Otherwise, from Theorem 3, we observe that we can deduce exact one ww dependency between t_0 and t_1. For example, consider the four lock operations with two possible orders in Fig. 7(b), there exists only one order (③) in which a ww dependency can be deduced. Thus, we deduce a ww dependency (recall that we assume there exists no bugs inside the DBMS when deducing dependencies), and then take this dependency together with the dependencies deduced from other verifying mechanisms to check whether contradictory dependencies exist.

Theorem 3: Given two transactions t_0 and t_1, for any overlapped time intervals of two conflicting locks, there exists at most one possible order in which a ww dependency can be deduced. Specifically, each of the other possible orders is identified to have incompatible locks.

Proof 3: Suppose there exist two possible orders in which two ww dependencies can be deduced. On the one hand, if t_0 has a ww dependency on t_1, then we can infer that the exact lock acquiring time of t_0 must happen before that of t_1. On the other hand, if t_1 has a ww dependency on t_0, then the exact lock acquiring time of t_1 must happen after the lock releasing time of t_0. To make the two ww dependencies possibly deduced, the lock acquiring time interval of t_0 (i.e., $\mathcal{A}^{\mathcal{T}_{w_{t_0}}}$) must overlap with the lock acquiring and releasing time intervals of t_1 (i.e., $\mathcal{A}^{\mathcal{T}_{w_{t_1}}}$ and $\mathcal{R}^{\mathcal{T}_{c_{t_1}}}$). Similarly, $\mathcal{R}^{\mathcal{T}_{c_{t_0}}}$ must also overlap with $\mathcal{A}^{\mathcal{T}_{w_{t_1}}}$ and $\mathcal{R}^{\mathcal{T}_{c_{t_1}}}$.

From $\mathcal{A}^{\mathcal{T}_{w_{t_0}}}$ overlaps with $\mathcal{A}^{\mathcal{T}_{w_{t_1}}}$ and $\mathcal{R}^{\mathcal{T}_{c_{t_1}}}$, we have $\mathcal{A}^{\mathcal{T}_{w_{t_1}}}.ts_{aft} < \mathcal{A}^{\mathcal{T}_{w_{t_0}}}.ts_{aft}$. Additionally, the lock releasing time interval must happen after the lock acquiring time interval, then we have $\mathcal{A}^{\mathcal{T}_{w_{t_0}}}.ts_{aft} < \mathcal{R}^{\mathcal{T}_{c_{t_0}}}.ts_{bef}$. Taken together, we have $\mathcal{A}^{\mathcal{T}_{w_{t_1}}}.ts_{aft} < \mathcal{R}^{\mathcal{T}_{c_{t_0}}}.ts_{bef}$, which indicates that $\mathcal{R}^{\mathcal{T}_{c_{t_0}}}$ would not overlap with $\mathcal{A}^{\mathcal{T}_{w_{t_1}}}$ and $\mathcal{R}^{\mathcal{T}_{c_{t_1}}}$ simultaneously. This is contradicted with the initial assumption. The theorem is proved.

The pseudo-code of the *ME* verification method is shown in Algorithm 2 (lines 10 $\sim$ 17). Based on the traces dispatched from the *Tracer*, it first constructs a lock table to organize the locks on each record (line 11). The lock table contains the time intervals of lock acquiring and releasing operations. When an operation releases its previously acquired locks, the verification process first refers to the lock table and finds all locks that conflict with the released locks (lines 12 $\sim$ 13). Next, for each released lock l_i and its conflicted lock l_j, it enumerates all possible orders of the lock operations based on their lock acquiring and releasing time intervals. If each of the possible orders indicates incompatible locks, then it reports an *ME* violation in bug descriptor (lines 14 $\sim$ 15). Otherwise, it deduces a ww dependency from the possible orders (line 17).

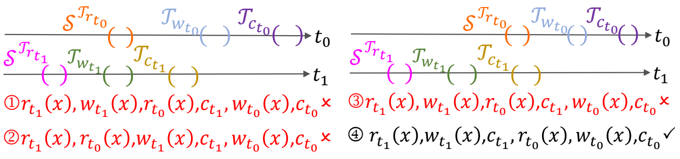
Complexity Analysis. The time complexity of a lock releasing (or acquiring) operation in our *ME* verification is $O(n_l \cdot n_t)$, where n_l is the average number of locks released (or acquired) by an operation and n_t is the average number of conflicted locks on each record in the lock table. Specifically, 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 with the insertion sort method. Thus, the time complexity of maintaining the lock table for each operation is $O(n_l \cdot n_t)$. Further, for any two conflicted locks, the cost of enumerating all possible orders of the lock operations is a constant value. This is because there are at most 4 possible orders if we enforce the lock acquiring operation must happen before lock releasing operation. 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, to reduce the size of n_x^l and n_t, we propose to asynchronously prune the locks of committed or aborted transactions that do not conflict with the locks of active transactions.

C. Verifying First Updater Wins

First updater wins (*FUW*) addresses the issue of lost update which might happen in concurrent transactions. The lost update occurs when the update of a transaction is overwritten by the update of another concurrent transaction. For example, suppose transaction t_0 and t_1 write independently using their own previously read values. If neither t_0 nor t_1 sees the update made from each other, then the first update on the record will be overwritten by the second one from the other transaction. *FUW* ensures that the updates of concurrent transactions are serializable. At first, *FUW* only permits the first updating transaction to commit or abort, and the other transaction has to wait for the execution result of the first updating transaction. If the first updating transaction commits successfully, then the other transaction would be forced to abort. If the first updating transaction aborts and rollbacks its update, then the other transaction would attempt to proceed with its update.

To verify *FUW* in the black-box mode, we propose to use the time intervals of snapshot generation, version installation and transaction commit/abort to identify whether there exist other concurrent versions during the period of a transaction execution. Specifically, the snapshot generation time interval contains the exact time point when a transaction get the snapshot of all its read records. The version installation and transaction commit/abort time intervals contain the exact time points when the transaction updates the record and then ends with a committed/aborted status.

The process of *FUW* verification is somewhat similar to the *ME* verification. As the aborted transactions always rollback their updates and would not lead to the case of lost update, we only consider the committed transactions in *FUW* verification. More specifically, suppose there exist two concurrent transactions operating on the same record. For the three kinds of time intervals mentioned above, if each possible order of their corresponding operations is identified to have concurrent record versions, then there must exist an *FUW* violation inside the DBMS. For example, consider the two transactions t_0 and t_1 which update the same record x in Fig. 8(a). The snapshot generation time interval of t_0 (i.e., $\mathcal{S}^{\mathcal{T}_{r_{t_0}}}$) lies between the time intervals of snapshot generation and transaction commit of t_1


$\mathcal{S}^{\mathcal{T}_{r_{t_0}}}$
$\mathcal{T}_{w_{t_0}}$
$\mathcal{T}_{c_{t_0}}$
t_0
$\mathcal{S}^{\mathcal{T}_{r_{t_1}}}$
$\mathcal{T}_{w_{t_1}}$
$\mathcal{T}_{c_{t_1}}$
t_1
①$r_{t_1}(x), w_{t_1}(x), r_{t_0}(x), c_{t_1}, w_{t_0}(x), c_{t_0}$ ×
②$r_{t_1}(x), r_{t_0}(x), w_{t_1}(x), c_{t_1}, w_{t_0}(x), c_{t_0}$ ×
③$r_{t_1}(x), w_{t_1}(x), r_{t_0}(x), c_{t_1}, w_{t_0}(x), c_{t_0}$ ×
④ $r_{t_1}(x), w_{t_1}(x), c_{t_1}, r_{t_0}(x), w_{t_0}(x), c_{t_0}$ ✓

(a) Triggering a Violation (b) Deducing a ww Dependency

Fig. 8. First Updater Wins Verification Examples

(i.e., $\mathcal{S}^{\mathcal{T}_{r_{t_1}}}$ and $\mathcal{T}_{c_{t_1}}$). Then we can infer that, for both of the two possible orders ① and ②, there must exist two concurrent versions. This is because the snapshot seen by t_0 does not contain the uncommitted update of t_1. As a result, the case of lost update would happen in t_1. Otherwise, there must exist exact one possible order in which a ww dependency can be deduced (see Theorem 4). Takes Fig. 8(b) as another example, for the possible orders ③ and ④, only ④ can deduce a ww dependency. Thus, we use ④ to deduce a ww dependency and transfer it to other verifying mechanisms.

Theorem 4: Given two committed transactions t_0 and t_1, for any overlapped time intervals of the two transactions, there exists at most one possible order in which a ww dependency can be deduced. Specifically, each of the other possible orders is identified to have concurrent versions.

Proof 4: The proof is similar to that of Theorem.3.

The pseudo-code of the *FUW* verification method is shown in Algorithm 2 (lines 18 $\sim$ 26). Given a write operation *op*, it first gets the generation time interval of the snapshot observed by *op* (line 19). Then, based on the trace set dispatched from the *Tracer*, it constructs the ordered versions of each record (line 20). With the help of ordered versions, it checks whether there exists a concurrent version regarding each record updated by *op*. Specifically, for each version x^i in the write set of *op* and its conflicted version x^j (lines 21 $\sim$ 22), it enumerates all possible orders of their operations of snapshot generation, version installation and transaction commit. If each of the possible orders indicates concurrent record versions, then it reports a *FUW* violation in *bug descriptor* (lines 23 $\sim$ 24). Otherwise, it deduces a ww dependency (line 26).

Complexity Analysis. 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.

D. Verifying Serialization Certifier

Serialization certifier (*SC*) applies some certifier-based approaches to guarantee that the transactions executed inside a DBMS are conflict serializable. Conflict serializability means that the dependencies between parallel transactions is equivalent to the dependencies between serial transactions.

A general approach of verifying conflict serializability is to build a dependency graph (*DG*) and performs cycle searches on the graph [12], [21]–[23]. Specifically, each node in *DG* corresponds to a committed transaction and each directed edge corresponds to a dependency between two transactions. If a cycle is found in *DG*, then it indicates that a violation of conflict serializability occurs. However, the complexity of cycle searching increases super-linearly with the scale of *DG*. To avoid the high cost of cycle searching, commercial DBMSs usually employ a lightweight certifier-based approach.

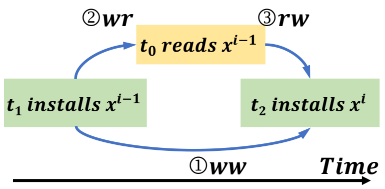


Fig. 9. Deducing Dependencies

For example, PostgreSQL uses the serializable snapshot isolation (*SSI*) technique as a certifier to guarantee the conflict serializability of transactions. Specifically, the certifier achieves this goal by avoiding write skew anomalies for snapshot isolation [41]. The snapshot isolation can be implemented by the lightweight *CR* and *FUW* mechanisms mentioned before. Write skew happens in the situation where each transaction writes to the individual version it sees, while the final result is not equivalent to that of any serial transactions. Fortunately, it can be efficiently detected by checking whether there exist two consecutive rw dependencies [41]. Thus, the certifier would abort one of the three transactions if there exist two rw dependencies between them. Takes CockroachDB as another example, it uses the multi-version timestamp ordering (*MVTO*) technique as a certifier to guarantee the conflict serializability. Specifically, the certifier does not allow a transaction with an older timestamp to have a dependency on the transaction with a newer timestamp. So the cycles would never appear in *DG*.

To efficiently detect the violation of conflict serializability, we propose to follow the idea of certifier-based approach inside the DBMS and directly use the *DG* to check whether there exists a violation of *SC*. Now the main challenge is how to build the *DG* in black-box mode. Fortunately, the wr dependencies can be obtained in the *CR* verification method and the ww dependencies can be obtained in the *ME* and *FUW* verification methods. Additionally, the rw dependencies can be further deduced from wr and ww dependencies. Fig. 9 illustrates the deducing method. Each rectangle represents a version installation time interval (colored green) or a snapshot generation time interval (colored yellow). If there exist two dependencies which indicate that transaction t_2 has a ww dependency on t_1 and t_1 has a rw dependency on t_0, then we can deduce that t_1 has a rw dependency on t_0.

The pseudo-code of the *SC* verification method is shown in Algorithm 2 (lines 27 $\sim$ 31). It first uses the dependencies deduced from the traces (including dependencies obtained from the other verification methods and the dependencies deduced by itself) to build a *DG* (lines 28 $\sim$ 29). Then, it checks whether there exists a dependency causing a dependency pattern that should be prohibited by the certifier inside the DBMS (line 30). If so, it reports a *SC* violation (line 31). For example in PostgreSQL, it checks whether a dependency leads to two consecutive rw dependencies.

As the dependencies are continuously deduced by the four verification methods, the size of *DG* would keep growing and lead to a large amount of memory consumption. To address this issue, we propose to asynchronously prune the garbage transactions (see Definition 4) and their dependencies to reclaim the memory space. Theorem 5 guarantees that the garbage transactions can be safely pruned without affecting the result of *SC* verification.

Definition 4: **Garbage Transaction.** A transaction t is a *garbage transaction* if t satisfies that: (*C1*) the in-degree of t is zero and (*C2*) $ts_{aft} \leq \mathcal{S}_e$. Here, ts_{aft} is the after timestamp of t's commit or abort operation, and $\mathcal{S}_e$ is earliest snapshot generation timestamp of any trace that has not been verified.

Theorem 5: A *garbage transaction* t is not a part of any future cycle on *DG*.

Proof 5: From C1 *in Definition 4, we can infer that the in-degree of* t *would be 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 future. From* C2 *in Definition 4, we can deduce that* $\mathcal{T}_t.ts_{aft} \leq \mathcal{S}_e \leq \mathcal{S}^{\mathcal{T}_k}.ts_{bef}$*. This indicates that any future transaction would not have dependencies on* t*. Taken together, the in-degree of* t *will keep as zero. However, the in-degree of* garbage transaction t *must be large than zero if* t *is contained in a cycle. Thus,* t *is not a part of any future cycle on* DG*. The theorem is proved.*

Complexity Analysis. 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*.

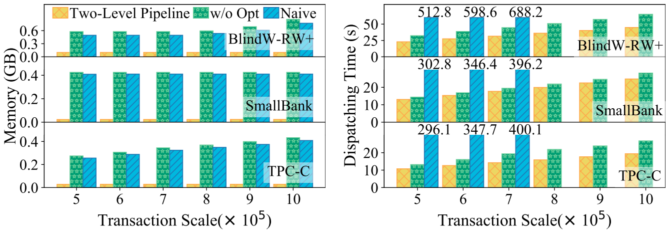
VI. Experimental Evaluation

In this section, we launch sufficient experiments to answer the following questions:

(1) Is our design efficient in verification?
  (a) How efficient is the *two-level pipeline* in dispatching traces? (Section VI-A)
  (b) How efficient is *mechanism-mirrored verification* in verifying the four mechanisms? (Section VI-B)
  (c) Can the throughput of *Leopard* surpass the throughput of a DBMS? (Section VI-C)
(2) How effective is *Leopard* to deduce dependencies for operations and transactions even with overlapped time intervals? (Section VI-D)
(3) Can *Leopard* outperform state-of-the-art work, including Cobra (Section VI-E) and Elle (Section VI-F)?

Environment. *Leopard* is implemented by Java ($v.1.8$). Our experiments are conducted on two servers, and each server is equipped with 2 Intel Xeon Silver 4110 @ 2.1 GHz CPUs, 120 GB memory, 4 TB HDD disk configured in RAID-5 and 4 GB RAID cache. The servers are connected using 10 Gigabit Ethernet.

Settings. We select one of the most popular DBMS, i.e., PostgreSQL ($v12.7$), to explore technical designs in *Leopard*. Specifically, the serializable of PostgreSQL is implemented by all the four mechanisms introduced in our paper, which can evaluate *Leopard* comprehensively. In default, we do garbage collection on all mechanisms and implement the optimizations of *two-level pipeline* as described in Section IV-C. By running *Leopard* on popular DBMSs, we demonstrate its ability in exposing transactional bugs.



(a) Memory (b) Dispatching Time

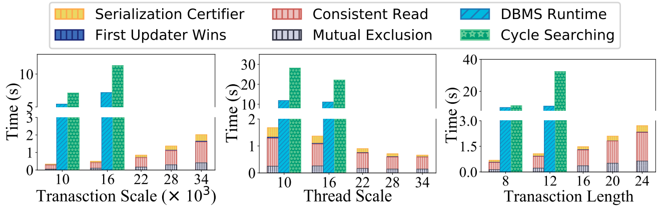
Fig. 10. Two-Level Pipeline Performance

Workload. TPC-C [55] and SmallBank [56] are used to check the ability of *Leopard* in verifying workload with complex application logic. By default, they populate database with *scale factor=1*. *BlindW* designed by *Cobra* [12] is a key-value workload with simple application logic and can be extended quantitatively to evaluate the designs in *Leopard*. In default, it creates a table sized $2K$ with values of 140 fixed-length strings. Each transaction has 8 operations and keys are accessed uniformly under serializable. For different evaluation purposes, we generate three workload variants of *BlindW*:

(1) *BlindW-W* contains 100% *blind-write* transactions with uniquely written values, i.e., a write not preceded by a read to the same key. Because *blind-write* can not access the record version before creating a new version, it is a tough scenario for tracking ww dependencies (Section VI-D).
(2) *BlindW-RW* evenly contains *item-read* and *blind-write* transactions. *BlindW-RW* can build three types of dependencies, used to evaluate the usefulness of the *mechanism-mirrored verification* on deducing dependencies (Section VI-D).
(3) *BlindW-RW+* replaces 50% *item-read* in *BlindW-RW* with 10-keys range-read, which challenges the performance of *Leopard* with more dependencies (Section VI-B).

A. Two-level Pipeline

We design a *two-level pipeline* to quickly order and dispatch traces. To demonstrate its efficiency, we compare it with the naive approach, which collects all traces from multiple clients and sorts them in a global buffer. Furthermore, we have proposed some optimizations to *two-level pipeline* as described in Section IV-C. Specifically, we prefer to dispatch traces from the client with the smallest timestamp to speed up the dispatching; and we guarantee there are as many traces going into *two-level pipeline* as the ones going out to keep a stable memory usage. To demonstrate the effectiveness of optimizations, we also compare *Leopard* with the one without optimization, i.e., *w/o Opt*. Because the timestamp distribution in traces affects the performance of *two-level pipeline* greatly, we run TPC-C, SmallBank and *BlindW-RW+* on PostgreSQL, which have different timestamp distributions.



(a) Varying Transaction Scale
(b) Varying Thread Scale
(c) Varying Transaction Length

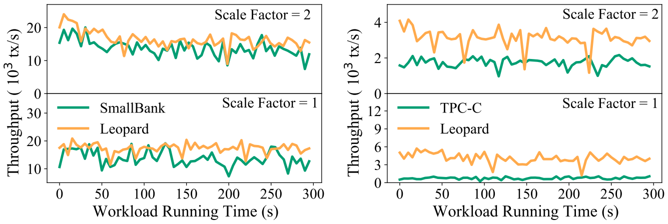
Fig. 11. Verification Time

As varying the number of transactions, i.e., transaction scale, we collect the memory usage and the dispatching time of *two-level pipeline* in Fig. 10. Our approach is consistent far better than the other two approaches on memory usage as in Fig. 10(a). The naive approach has similar maximal memory consumption with *Leopard w/o Opt*. This is due to the accumulation of a huge amount of traces in the global buffer when the distribution of timestamps in each client is extremely uneven. The dispatching time of *Leopard* is faster than the other two approaches $w.r.t$ all three workloads and the naive approach is the worst as in Fig. 10(b). The reason is that the naive approach sorts traces synchronously, while our approach sorts a batch of traces asynchronously. Specifically, running *BlindW-RW+* on PostgreSQL, it has the highest throughput (maximum traces), which then has the longest dispatching time. When transaction scale is 7×10^5, the naive approach takes 17× longer time to dispatch traces than ours; it will be worse when increasing the transaction scale, so we do not plot its time for transaction scale $> 7 \times 10^5$. In summary, *two-level pipeline* can dispatch traces efficiently.

B. Mechanism-mirrored Verification

Verifying ILs is equivalent to verify the execution of the four mechanisms. The naive approach is to build a *dependency graph* and do cycle searching. However, the verification efficiency of the naive way is significantly affected by the number of traces. For verification efficiency, we propose a *mechanism-mirrored verification* approach by simulating the workflow of concurrency control protocols inside a DBMS, which only occupies a small part of DBMS runtime theoretically. We vary the critical factors affects verification time, including 1) transaction scale which controls the number of transactions; 2) thread scale which controls the data contentions; 3) transaction length which controls the number of operations in a transaction. Since *BlindW-RW+* can be extended to control the three factors quantitatively, we compare the verification time of our approach with DBMS runtime and the naive cycle searching approach by running it on PostgreSQL as in Fig. 11. In default, we launch 24 threads to issue $20K$ transactions on PostgreSQL and the transaction length is 8.

The verification time for the four mechanisms by our approach is linear with the transaction scale as shown in Fig. 11(a), which benefits from our timely garbage pruning. Increasing thread scale aggravates transaction contentions, leading to a higher abort rate. Since the aborted transactions are not involved in verification, the verification time for our approach decreases if increasing the thread scale, as shown in Fig. 11(b). Increasing transaction length makes the read or write set in a transaction expand accordingly. The verification time then linearly increases with transaction length, as in Fig. 11(c). This is because the complexity of our verification method is linear dependent on the size of write set and read set. Notice that, we can significantly outperform the naive method or the DBMS runtime in verification efficiency, so we do not plot all figures for comparison work for the scale problem. For example in Fig. 11(a), when transaction scale =16 × 10^3, cycle searching approach and DBMS runtime have been 24× and 15× slower than our approach. Thus, *mechanism-mirrored verification* can efficiently verify ILs.



(a) SmallBank
(b) TPC-C

Fig. 12. Workload Throughput vs. Leopard Throughput

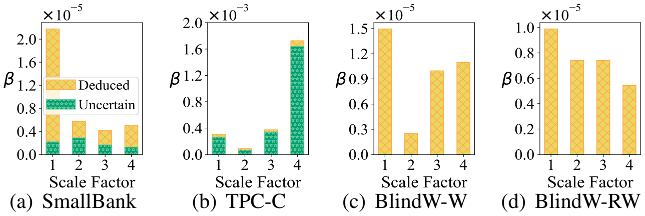
C. Comparison with DBMS Throughput

We have show the performance of each component in *Leopard*, including *two-level pipeline* and *mechanism-mirrored verification*. Here, we run TPC-C and SmallBank (with complex transaction logic) on PostgreSQL for 300s continuously to demonstrate the overall verification throughput. We divide the workload traces into batches every 0.5s, which are input to local buffers. A smaller database usually meets a higher contention, which leads to more complex dependencies, so we use small *scale factor*s. In Fig. 12, we compare the throughputs of the DBMS and *Leopard* as varying *scale factors*.

TPC-C has more complex application logic than SmallBank, so SmallBank has a higher throughput, i.e., issues more traces to *Leopard*. In Fig. 12(a), for the low contention in SmallBank, there is little difference between throughputs of PostgreSQL and *Leopard* under different *scale factor*s; for the complex TPC-C, *Leopard* outperforms PostgreSQL in throughput obviously under different *scale factor*s as in Fig. 12(b). Thus, *Leopard* can catch up well with transaction processing and will have better performance on complex workloads.

D. Effectiveness of Deducing Dependencies

Uncertain dependencies exist due to client-side trace overlappings. We have proposed to deduce wr, ww and rw dependencies during verifying the four implementation mechanisms. These dependencies can expose the order between conflict operations. To demonstrate effectiveness, we run SmallBank, TPC-C and two variants of *BindW*, i.e., *BlindW-W* and *BlindW-RW*, on PostgreSQL for 20 minutes to cover trace overlappings as much as possible. We plot the ratio of traces of conflict operations that overlap on time intervals by β. β is divided into *deduced* and *uncertain* to indicate the dependencies that *Leopard* can and cannot deduce respectively.



(a) SmallBank (b) TPC-C (c) BlindW-W (d) BlindW-RW

Fig. 13. Deducing Dependencies on PostgreSQL

β is generally a small number ($< 10^{-3}$) as shown in Fig. 13. For the complex application logic in SmallBank and TPC-C, there is still a part of dependencies that can not be deduced as shown in Fig. 13(a) and 13(b). Specifically, in SmallBank, transaction *amalgamate* always writes the same values, and duplicate values can not be distinguished in its candidate version set. In TPC-C, many transactions read and write a part of attributes instead of the whole record, which makes it impossible to deduce the dependencies of two operations if they operate on different attributes of the same record. In *BlindW-W*, all uncertain dependencies appearing in traces are ww dependencies, which can be well deduced by our approach, as shown in Fig. 13(c). The *blind-write* in *BlindW-RW* writes distinct values, which makes the candidate version set without duplicate values. Therefore, we can deduce all wr dependencies from overlapped traces as shown in Fig. 13(d). Thus, we can expose more uncertain dependencies effectively.

E. Comparison with Cobra on Efficiency

Cobra is the state-of-the-art work used only to verify serializable key-value stores [12]. It models transaction dependencies by a graph and check cycles in the graph. Moreover, it enables garbage collection by inserting fence transactions. We compare *Leopard* with *Cobra*, which issues a fence transaction every 20 transactions, and *Cobra w/o GC* which disables fence transactions .We report the verification time and memory usage in Fig. 14. Because of the long verification time of *Cobra* which will be exacerbated by range-read, we only run *BlindW-RW* here.

As increasing the transaction scale, the verification time of *Leopard* and *Cobra w/o GC* increases linearly and super-linearly, respectively; but *Cobra* is the worst, which spends much time identifying garbage dependencies on its polygraph as in Fig. 14(a). In memory usage, *Cobra w/o GC* cost the most to store all dependencies. *Leopard* is stable in memory usage which is almost the same as that of *Cobra* for a large transaction scale as in Fig. 14(b), but *Cobra* has the lowest verification efficiency as in Fig. 14(a). Specifically, for $20K$ transactions, *Leopard* outperforms *Cobra* by 114× in verification efficiency with almost the same memory usage. In Fig. 14(c) and 14(d), we vary the number of workload threads to generate $20K$ transactions. *Cobra w/o GC* has the worst memory usage. Though *Cobra* has a lower memory consumption, it has uncontrollable verification time on the high concurrent workload, which is 271× slower than *Leopard* when thread scale=32. Thus, *Leopard* has a much better scalability $w.r.t$ both the scales of transactions and threads.

F. Comparison with Elle on Bug Cases

Elle is another state-of-the-art work to verify ILs [20]. But it requires its workload to make the version order manifest, which cannot be used to verify arbitrary workload, so we do not launch performance comparison with it. Additionally, *Elle* can not distinguish repeatable read from serializable of PostgreSQL [16]. Since *Elle* has been used to test TiDB [8], we compare with it by explaining two bugs detected by *Leopard* which can not be located by *Elle* (see more comparison in full paper [57]).

```
CREATE TABLE t(a INT PRIMARY KEY, b INT);
INSERT INTO t(676, -5012153);
BEGIN TRANSACTION;--TID:739
UPDATE t SET b=-5012153 WHERE a=676;--TID:739
UPDATE t SET b=-852150 WHERE a=676;--TID:723✖
COMMIT;--TID:739
```

```
CREATE TABLE t(a INT PRIMARY KEY, b FLOAT);
INSERT INTO t(3873, -1.123);
UPDATE t SET b=-0.386 WHERE a=3873;--TID:904
UPDATE t SET b=0.484 WHERE a=3873;--TID:907
SELECT b FROM t WHERE a=3873;
--TID:914, Result:{-0.386}✖
```

Bug 1: Dirty Write. Transaction $TID = 739$ writes a record, i.e., a=676, and then another transaction $TID = 723$ also writes this record before 739 commits, which results in a dirty write [36]. We find that the first update does not modify the record, leading to *TiDB* acquiring no lock.

Bug 2: Inconsistent Read. Transaction $TID = 914$ reads the record written by the first update $TID = 904$, but does not read the latest one written by the second update $TID = 907$, which violates the linearizability.

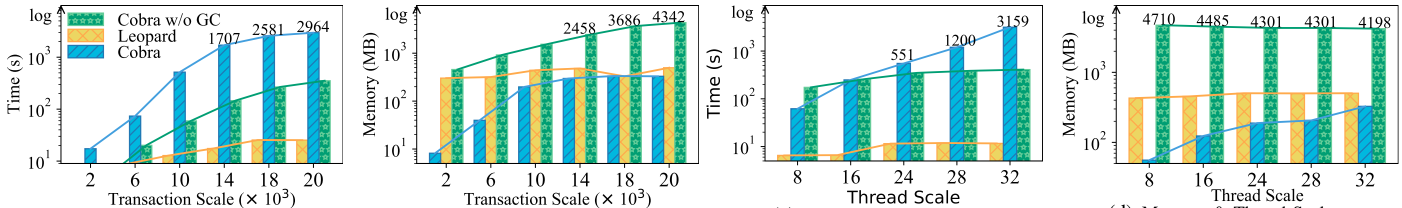
```
CREATE TABLE t(a INT PRIMARY KEY, b INT);
CREATE TABLE s(a INT PRIMARY KEY, b INT);
ALTER TABLE s ADD FOREIGN KEY(b) REFERENCES t(a));
INSERT INTO t(1, 2);
INSERT INTO s(2, 1);
BEGIN TRANSACTION;--TID:211
UPDATE t SET b=3 WHERE a=1;--TID:211
SELECT * FROM t, s WHERE t.a=s.b AND s.a>1
FOR UPDATE; --TID:324, Result:{2,1,2}✖
COMMIT;--TID:211
```

Listing 1. Incompatible Write Locks

```
CREATE TABLE t(a INT PRIMARY KEY, b INT);
CREATE TABLE s(a INT PRIMARY KEY, b INT);
ALTER TABLE s ADD FOREIGN KEY(b) REFERENCES t(a));
INSERT INTO t(1, 2);
INSERT INTO s(2, 1);
DELETE FROM s WHERE a=2;--TID:213
BEGIN TRANSACTION;--TID:412
INSERT INTO s VALUES(2,3);--TID:412
SELECT * FROM t WHERE a=2;
--TID:412, Result:{2,1},{2,3}✖
```

Listing 2. A Query that Returns two versions

Bug 3:Incompatible Write Locks. In Listing. 1, transaction $TID = 211$ acquires a long write lock on record 1 in table t, and another concurrent transaction $TID = 324$ successfully reads record 1 by *FOR UPDATE* statement, which violates the mutual exclusion between writes. It is worth noting that



(a) Time & Transaction Scales (b) Memory & Transaction Scales (c) Time & Thread Scales (d) Memory & Thread Scales

Fig. 14. Comparison with Cobra

324 accesses record 1 through *join* operator. Before accessing record 1, TiDB forgets the lock acquisition, leading to this bug.

Bug 4:A Query that Returns two versions. In Listing. 2, transaction $TID = 412$ returns two versions for a record. One is the version written by 412 itself, and the other is the deleted version, which should not be available. We report this problem to *TiDB* and confirmed that it was a known bug.

Because transaction dependencies are critical to verify ILs, *Elle* carefully designs some specific workloads to expose the dependencies on which to check dependency cycle. *Elle* can not find the problem without dependency cycles, which greatly limits the usage scenarios. For example, Bug 1 exists by running SmallBank on *TiDB*. But there is no dependency cycle, so *Elle* fails to find it. *Leopard* is designed for all classic concurrency control protocols, which expects to expose more subtle bugs than cycle searching.

Summaries and Practical Applications. *Leopard* can be easily integrated with any workload generator, e.g., *OLTP-Bench* [58], without modifying any application logic and is general to verify all classic isolation levels. By running the existing synthetic and application workloads, it has detected 17 bugs [57] hidden in popular commercial databases. *Leopard* has proved to be efficient and effective in detecting isolation level bugs. It has helped our collaborators detect and locate a series of bugs hidden in their *DBMS*s, even though they have launched thorough testings by other tools, e.g., *Elle* on TiDB.

VII. Related Work

Verifying Isolation Levels of DBMS. The verification on isolation levels (ILs) is usually achieved by elaborating specific workloads or instrumenting the source codes of databases [5], [12], [20], [59]. However, none of these work is general to verify various ILs in a black-box mode with arbitrary workloads. *Cobra* [12] is one of state-of-the-art work. It can only expose serializability violation of transactional key-value stores by a workload following a specific application logic defined for read and write operations. By injecting fence transactions into its workload, *Cobra* prunes garbage transactions in its dependency graph by means of an expensive graph traverse. *Elle* [20] (part of the *Jepsen* project [60]) carefully designs a short workload that can generate manifest version orders in history for IL verification. It can not carry out the verification for other workloads, e.g., *TPC-C*, which may neglect some subtle bugs in transactional databases. Additionally, *Elle* can not distinguish repeatable read from serializable in PostgreSQL [16]. Mai et.al [5] aim to diagnose violations of the ACID properties by injecting simulated power faults while replay its workloads, which is a white-box method. Yu et.al [59] design a DBMS that can provide verifiable proofs of transaction correctness and semantic properties including atomicity and serializability. But it is only verifiable on the serializable IL, and the other ILs can not be verified. In comparison to these work, *Leopard* can achieve the verification for any application logic of workload and is not limited to a specific IL.

Detecting Isolation Anomalies in Application Workloads. Some other work trusts databases in providing correct ILs and proposes to detect anomalies in application workloads [4], [18], [21]–[23]. *IsoDiff* [21] takes an analysis of application codes to debug anomalies on dependency graphs which are caused by weak isolations. It reduces the cost of detection by limiting its search in representative subsets. *Rushmon* [22] monitors real-time anomalies and reports the "chaos" level caused by the asynchronous algorithm for the inconsistency-tolerant applications on weak isolation systems. It takes the idea of serializability and achieves real-time detection by sampling the dependency graph. However, the sampling-based approach may neglect some anomalies and is not suitable for comprehensive checking. *ConsAD* [23] quantifies isolation anomalies by detecting cycles in the dependency graph. But it needs a lot of manpower to analyze the application logic to complete the possible dependency tracking. Todd et.al [4] aim to detect potential isolation anomalies in web applications. They reason the possible concurrent interleavings among clients, so as to generate workloads that may violate the ACID principle. In contrast, *Leopard* aims to verify various ILs in database without modifying application workloads.

VIII. Conclusion and Future Work

Leopard abstracts four general implementation mechanisms for various ILs. It proposes to verify ILs in a black-box way based on client-side execution traces without modifying any application logic or instrumenting source code of DBMSs. *Two-level pipeline* and *mechanism-mirrored verification* are designed to accomplish efficient and effective verification. Compared with existing studies, *Leopard* has order-of-magnitudes improvement on performance and better ability in bug detection. However, the side-effect of time interval overlapping of traces prevents us from detecting all dependencies, and digging up all bugs is still impossible. We leave it as our future work.

References

[1] R. L. J. Gray, G. Putzolu, and I. Traiger, "Granularity of locks and degrees of consistency," *Modeling in Data Base Management Systems, GM Nijssen ed., North Holland Pub*, 1976.

[2] A. D. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. E. Shasha, "Making snapshot isolation serializable," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 492–528, 2005.

[3] A. Pavlo, "What are we doing with our lives? nobody cares about our concurrency control research," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 3–3.

[4] T. Warszawski and P. Bailis, "Acidrain: Concurrency-related attacks on database-backed web applications," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 5–20.

[5] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Torturing databases for fun and profit," in *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014, pp. 449–464.

[6] J. C. C. r and et al., "Spanner: Google's globally-distributed database," in *OSDI*, 2012, pp. 251–264.

[7] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "CockroachDB: The resilient geo-distributed SQL database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1493–1509.

[8] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "TiDB: a Raft-based HTAP database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.

[9] C. A. Stuardo, T. Leesatapornwongsa, R. O. Suminto, H. Ke, J. F. Lukman, W.-C. Chuang, S. Lu, and H. S. Gunawi, "Scalecheck: A single-machine approach for discovering scalability bugs in large distributed systems," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 359–373.

[10] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM symposium on cloud computing*, 2014, pp. 1–14.

[11] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 1–16.

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

[13] K. P. Gaffney, R. Claus, and J. M. Patel, "Database isolation by scheduling," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1467–1480, 2021.

[14] "Cockroachdb bugs," https://github.com/cockroachdb/cockroach/issues.

[15] "Yugabyte bugs," https://github.com/yugabyte/yugabyte-db/issues.

[16] "Jepsen: Postgresql 12.3." https://jepsen.io/analyses/postgresql-12.3.

[17] A. Fekete, S. N. Goldrei, and J. P. Asenjo, "Quantifying isolation anomalies," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 467–478, 2009.

[18] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan, "Automating the detection of snapshot isolation anomalies," *Proceedings of the VLDB Endowment*, 2007.

[19] A. Dey, A. Fekete, R. Nambiar, and U. Röhm, "Ycsb+ t: Benchmarking web-scale transactional databases," in *2014 IEEE 30th International Conference on Data Engineering Workshops*, 2014, pp. 223–230.

[20] P. Alvaro and K. Kingsbury, "Elle: Inferring isolation anomalies from experimental observations," *Proceedings of the VLDB Endowment*, vol. 14, no. 3, pp. 268–280, 2020.

[21] Y. Gan, X. Ren, D. Ripberger, S. Blanas, and Y. Wang, "Isodiff: debugging anomalies caused by weak isolation," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2773–2786, 2020.

[22] Z. Shang, J. X. Yu, and A. J. Elmore, "Rushmon: Real-time isolation anomalies monitoring," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 647–662.

[23] K. Zellag and B. Kemme, "Real-time quantification and classification of consistency anomalies in multi-tier architectures," in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 613–624.

[24] K. Nagar and S. Jagannathan, "Automated detection of serializability violations under weak consistency," *arXiv preprint arXiv:1806.08416*, 2018.

[25] Y. Lu, X. Yu, L. Cao, and S. Madden, "Aria: a fast and practical deterministic oltp database," *Proceedings of the VLDB Endowment*, 2020.

[26] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[27] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev, "Serializability for eventual consistency: criterion, analysis, and applications," in *SIGPLAN*, 2017, pp. 458–472.

[28] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic detection of atomic-set-serializability violations," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 231–240.

[29] A. Sinha and S. Malik, "Runtime checking of serializability in software transactional memory," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[30] W. N. Sumner, C. Hammer, and J. Dolby, "Marathon: Detecting atomic-set serializability violations with conflict graphs," in *International Conference on Runtime Verification*, 2011, pp. 161–176.

[31] M. Xu, R. Bodík, and M. D. Hill, "A serializability violation detector for shared-memory server programs," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 1–14, 2005.

[32] K. Zellag and B. Kemme, "Consistency anomalies in multi-tier architectures: automatic detection and prevention," *The VLDB Journal*, vol. 23, no. 1, pp. 147–172, 2014.

[33] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *TODS*, vol. 8, no. 4, pp. 465–483, 1983.

[34] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, 1979.

[35] A. X3, "American national standard for information systems-database language-sql," 1992.

[36] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[37] A. Adya and B. H. Liskov, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[38] N. Crooks, Y. Pu, L. Alvisi, and A. Clement, "Seeing is believing: A client-centric specification of database isolation," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017, pp. 73–82.

[39] A. Szekeres and I. Zhang, "Making consistency more consistent: A unified model for coherence, consistency and isolation," in *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, 2018, pp. 1–8.

[40] "Innodb," https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[41] D. R. K. Ports and K. Grittner, "Serializable snapshot isolation in postgresql," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1850–1861, 2012.

[42] "Oracle database," https://www.oracle.com/hk/database/technologies/.

[43] A. Fekete, E. O'Neil, and P. O'Neil, "A read-only transaction anomaly under snapshot isolation," *ACM SIGMOD Record*, vol. 33, no. 3, pp. 12–14, 2004.

[44] R. Biswas and C. Enea, "On the complexity of checking transactional consistency," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[45] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 18–32.

[46] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1629–1642.

[47] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 1–12.

[48] J. Wang, D. Ding, H. Wang, C. Christensen, Z. Wang, H. Chen, and J. Li, "Polyjuice: High-performance transactions via learned concurrency control," in *OSDI*, 2021, pp. 198–216.

[49] D. Tang and A. J. Elmore, "Toward coordination-free and reconfigurable mixed concurrency control," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 809–822.

[50] "NuoDB," https://nuodb.com/.

[51] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in sap hana database: the end of a column store myth," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 731–742.

[52] "Oceanbase," https://www.oceanbase.com/.

[53] G. Weikum and G. Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[54] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[55] "TPC-C benchmark," http://www.tpc.org/tpcc/.

[56] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The cost of serializability on platforms that use snapshot isolation," in *2008 IEEE 24th International Conference on Data Engineering*, 2008, pp. 576–585.

[57] "Leopard," https://github.com/Coconut-DB1024/Leopard.

[58] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, 2013.

[59] Y. Xia, X. Yu, M. Butrovich, A. Pavlo, and S. Devadas, "Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness."

[60] "Jepsen," https://github.com/jepsen-io/jepsen.