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

Keqiang Li[1], Siyang Weng[1], Peiyuan Liu[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]

Shanghai Engineering Research Center of Big Data Management, East China Normal University[1], Alibaba Group[2]

{kqli, syweng, pyliu}@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 of existing concurrency control protocols, we summarize and abstract four mechanisms which can implement ILs in all commercial DBMSs 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 23 bugs that cannot 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 offers better performance but sacrifices the guarantees of a perfect isolation. Thus, commercial DBMSs support various ILs to provide applications with a trade-off between consistency and performance [2].

Although theoretical correctness proofs have been provided for almost all ILs, the practical implementations might not strictly follow the definitions [3], [4]. 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 [5]–[7], which would lead to an extraordinarily complex protocol combination over multiple remote machines [8]–[10]. This complexity also results in a number of subtle bugs [11]–[15]. 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 [11], [16]–[23] and the key challenges are summarized as follows:

Black-box Verification (C1). Most concurrency control protocols are non-deterministic, that is the final database state might be different even given the same input [24]. Thus, the traditional differential testing method [25] 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 first type usually instruments the kernels to catch the internal execution state of DBMSs [23], [26]–[31]. However, modifying the kernels is laborious or even impossible, especially for cloud services provided by the third party [32]. The second type relies on specific workloads to make transaction dependencies easy to obtain [11], [16]–[19]. However, these methods have a limited scope of application scenarios, and might neglect some subtle bugs in other 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 proven 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 verifications. Most of the existing work [11], [44] has been designed to only verify the strongest IL, i.e., *serializable*. Although *Elle* [19] has tried to find different IL anomalies, it could not distinguish the *repeatable read* and *serializable* in PostgreSQL [15].

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

Chengcheng Yang and Rong Zhang are corresponding authors.

some mischief and profits [3], [4]. 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 massive transactions pose a significant challenge on the efficiency of IL verification. Previous studies often fail to preform an efficient IL verification on a long running workload. For example, the verification time of *Cobra* [11] grows superlinearly with the number of transactions. *Elle* can only verify ILs 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 a 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 a DBMS to verify 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 and reduce the memory usage. It's worth noting that, *Leopard* does not guarantee the correctness of ILs since the exact execution time point of each database operation is not available in the black-box mode. Thus, we position it as a bug finding tool which verifies the test cases with best efforts. 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) The experiments show that *Leopard* outperforms existing methods by up to $114\times$ in verification time with a relative small memory usage. Moreover, we have successfully found 23 bugs (13 fixed, 15 confirmed and 8 open reported) 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 (IL) 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 of 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

DBMSs often define different ILs, e.g., *read committed* (*RC*), *repeatable read* (*RR*), *snapshot isolation* (*SI*), and *serializable* (*SR*), to prevent different isolation anomalies [35]–[39]. Specifically, there exist no unified definitions of ILs and each DBMS has its specific definition (more details can be found in [45]). To implementing these ILs, the community has devised many concurrency control protocols (*CCP*), including *optimistic concurrency control* (*OCC*), *two-phase locking* (*2PL*), *multi-version concurrency control* (*MVCC*), *serializable snapshot isolation* (*SSI*) and *timestamp ordering* (*TO*).

After carefully investigating 18 popular DBMSs, we discover that all *CCP*s in these DBMSs eliminate isolation anomalies through the following four classic mechanisms: *consistent read* (*CR*), *mutual exclusion* (*ME*), *first updater wins* (*FUW*), and *serialization certifier* (*SC*). In table I, we summarize the implementations of ILs in popular DBMSs. For example, *serializable* in PostgreSQL takes all the above four mechanisms to eliminate isolation anomalies, while *snapshot isolation* may suffer from *write skew* that is a kind of isolation anomaly prohibited by *SC* [41]. Note that there still exist some other mechanisms [24], [46]–[50], 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. To eliminate the *read skew* anomaly which might lead to a transaction see an inconsistent state of the database, *MVCC* takes *CR* to see a snapshot

TABLE I
THE IMPLEMENTATIONS OF ISOLATION LEVELS IN POPULAR DBMSS

DBMS	CCP	IL	CR	ME	FUW	SC
PostgreSQL [41], OpenGauss [51], yugabyteDB [52]	2PL+MVCC +SSI [41]	SR SI RC	✓ ✓ ✓	✓ ✓ ✓	✓ ✓	✓
InnoDB [40], SQL server [53], Aurora [54], PolarDB [55],	2PL+MVCC	SR,RR, RC	✓	✓		
TiDB [7]	2PL+MVCC Percolator [56]	RR,RC SI	✓ ✓	✓		✓
RocksDB [57]	2PL+MVCC OCC+MVCC	SI SI	✓ ✓	✓	✓	✓
SQLite [58]	2PL	SR		✓		
FoundationDB [59]	OCC+MVCC	SR	✓			✓
SingleStore [60]	2PL+MVCC	RC	✓	✓		
CockroachDB [6]	TO+MVCC	SR	✓			✓
Spanner [5]	2PL+MVCC	SR	✓	✓		
Oracle [42], SAP HANA [61], Oceanbase [62], NuoDB [63]	2PL+MVCC	SI RC	✓ ✓	✓ ✓	✓	

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 different isolation levels, there exist transaction-level *CR* and statement-level *CR*. Transaction-level *CR* provides a consistent view of the database at the beginning of a transaction, while statement-level *CR* at the beginning of an operation.

Mutual Exclusion (*ME*) uses the locking strategy to provide a kind of exclusive accesses on shared resources. *2PL* takes the mechanism of *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*) prevents transactions which modify the same record from concurrent executions. *Lost update* indicates a transaction does not see the update of another transaction when updating the same record. To eliminate the *lost update* anomaly, most DBMSs take *FUW* to ensure that transactions modifying the same record should execute in a serial order. DBMSs often combine *CR* and *FUW* mechanisms to guarantee *snapshot isolation* [41], [61]–[63].

Serialization Certifier (*SC*) guarantees the transaction execution is conflict serializable [64]. Many *CCP*s takes *SC* to eleminate the serialization anomaly. The anomaly might lead to a transaction schedule could not be transformed into a serial schedule by swapping non-conflicting operations. However, each *CCP* has its own "certifier". Specifically, *SSI* uses *write skew* anomalies as certifier, *TO* uses the timestamp ordering as certifier, and *OCC* uses the conflicts checking as certifier. [45] describes more details of IL implementations.

III. LEOPARD FRAMEWORK

The framework of *Leopard* is illustrated in Fig. 1, 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 client-side invocation and completion timestamps of each

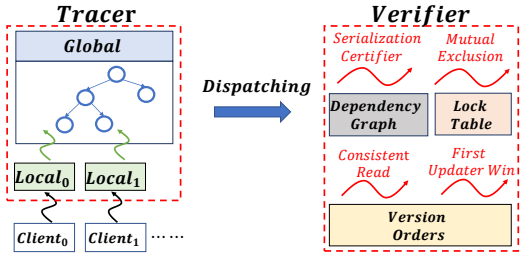
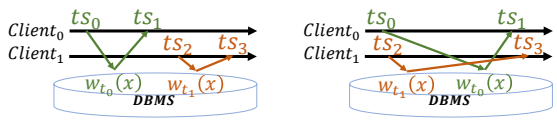


Fig. 1. Leopard Framework

operation and without modifying 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. 1 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.

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 [11] and sampling representative subsets from the graph [21]. 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 can be decomposed into 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 a 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. It's worth noting that there still exist some manual efforts for our *Leopard* framework. That is, the *Verifier* should be aware of the IL definitions of the tested DBMS, such that it can



(a) No Overlap (b) Overlap

Fig. 2. All Cases between Traces of Two Conflicting Operations

find out how to combine the four mechanisms. Fortunately, these definitions are usually described in the DBMS's official web documents. In table I, we summarize the IL definitions and the combination of the four mechanisms of popular DBMSs. 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. Note that, *Leopard* can be also deployed offline. Specifically, the client only logs the time interval of each database operation and store these *interval-based traces* in the persistent storage. Then, our Leopard can sort (i.e. dispatch) and verify these traces in an offline fashion.

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 and dispatch massive traces to *Verifier* (Section IV-C).

A. Interval-based Trace

To perform IL verification in a black-box mode, we log the client-side time interval of each operation in all clients connected to the 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 the 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, the timestamps appearing in traces require clock synchronizations. If test clients are deployed on a single machine (resp. multiple machines), we use its hardware (resp. software) time for clock synchronizations. Specifically, for *Leopard*'s distributed deployments, we synchronize clocks with a centralized logical timestamp generation method, which is widely used by distributed databases, e.g., TiDB and OceanBase.

B. Deducing Transaction Dependencies

As each IL has a specific restriction on the allowed dependency patterns [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. 2. 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

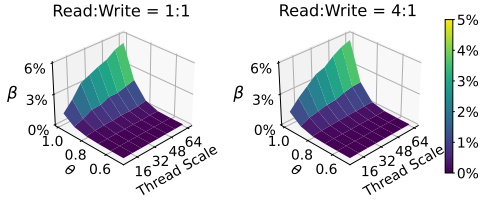


Fig. 3. Overlapping Ratio β in YCSB-A

$\mathcal{T}_1 = \{ts_2, ts_3, w_{t_1}(x)\}$. If the time intervals of the two traces do not overlap (see Fig. 2(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 (see Fig. 2(b)). 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* [65] 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 denote β as the ratio of the number of uncertain dependencies to the total number of actual dependencies. As shown in Fig. 3, 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 is kept 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, the designs in *Leopard* can further eliminate uncertain dependencies.

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.

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 traces whose before timestamps are less than watermark to *Verifier* (lines 2, 8); (b) the global buffer fetches traces from

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*.

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

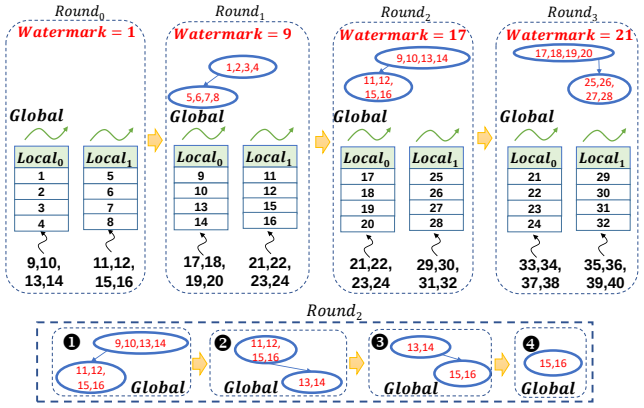


Fig. 4. Example of Batch-based Two-level Pipeline

local buffers (lines $4 \sim 5$); (c) the global buffer updates the watermark (line 7). Note, the traces in each local buffer are naturally sorted since 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*. Due to space constraints, we leave the formal proofs of all theorems to our technical report [66].

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

Optimizations. Our trace sorting procedure also takes advantage of three optimizations, which are multi-thread processing, batch dispatching, and adaptive fetching. Specifically, to fully utilize the parallelism of modern CPUs, we propose to partition the traces, and then parallelly sort and verify the traces with multi-threads. Moreover, as we observe that there usually exist some consecutive traces in a local buffer whose before timestamps do not interleave with any trace in other local buffers, we propose a batch processing optimization which aims to directly dispatch these consecutive traces to the verifier at once. Further, as the watermark has a significant impact on the size of global buffer, we propose an adaptive method which fetches traces and controls the watermark according to the distribution of timestamps in each local buffer.

Fig. 4 gives a running example. Each trace is represented by its before timestamp, and we set batch size as 4 traces. 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 $Round_1$, the global buffer first fetches a batch of traces from each local buffer and sorts them with the min-heap. The watermark is set as the smallest before timestamp of the two local buffers, which is 9. Then, it pops the heap and get a trace batch $\langle 1, 2, 3, 4\rangle$. As all the traces in the trace batch are smaller than the watermark and the first trace 5 in the heap root, so it directly dispatches them as a batch. Similarly, $\langle 5, 6, 7, 8\rangle$ can be also dispatched as a batch.

In $Round_2$, it repeats the steps in $Round_1$, that is, fetches a batch of traces from each local buffer into the global buffer, sorts them with the min-heap, and the watermark is set as 17. Specifically, it takes the following steps to dispatch the traces in batches. ❶ It pops the heap and get traces $\langle 9, 10, 13, 14\rangle$. However, as $\langle 13, 14\rangle$ are large than the first trace 11 in current heap root. Then, it split the popped trace batch and dispatches $\langle 9, 10\rangle$ as a batch. Next, it inserts $\langle 14, 15\rangle$ into the global buffer as a batch. ❷ Similarly, it pops traces $\langle 11, 12, 15, 16\rangle$, and dispatches traces $\langle 11, 12\rangle$. Then, it inserts $\langle 15, 16\rangle$ into the global buffer as a batch. ❸ It directly dispatches traces $\langle 13, 14\rangle$ which is smaller than the watermark and the first trace 15 in current heap root. ❹ Finally, it directly dispatches traces $\langle 15, 16\rangle$ which is smaller than the watermark. Similarly, in $Round_3$, it repeats the steps as described in $Round_2$.

Complexity Analysis. The space complexity is $O(n_{local} \cdot (s_{local} + s_{client}))$, where n_{local} is the number of local buffers, s_{local} is the size of each local buffer and s_{client} is the size of traces temporally stored in each client. Note, our adaptive fetching optimization can bound the global buffer size as $O(n_{local} \cdot s_{local})$. As each trace dispatch consists of a heap push and a heap pop, the time complexity of each dispatched trace is $O(log(n_{local} \cdot s_{local}))$. Moreover, the batch processing can further reduce the time complexity to $O(\frac{1}{B} \cdot log(\frac{n_{local} \cdot s_{local}}{B}))$, where B is the average batch size.

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 sees the snapshot of a database as of the beginning of a transaction, while the statement-level consistent read sees the snapshot as of 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 visible snapshot 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 visible snapshot time intervals as follows.

Definition 1: **Version Installation Time Interval.** Suppose a write operation creates a new version and $\mathcal{T}$ is the operation

Algorithm 2 Mechanism-mirrored Verification

Input: *op*: operation; $\mathcal{T}$: the trace of *op*; *traces*: the trace set dispatched.
Output: *bug descriptor*

```
1:  procedure ConsistentRead
2:      S^T ← the visible snapshot time interval of op;
3:      OV ← use traces to construct ordered versions of each record;
4:      CV^T ← candidate_version_set(OV, S^T)
5:      for each version x^i in the read set of op do
6:          if x^i ∉ CV^T then
7:              return a CR violation in bug descriptor;
8:          if there exists only one version in CV^T matches x^i then
9:              Deduce a wr dependency;
10: procedure MutualExclusion
11:     LT ← use traces to construct a lock table for each record;
12:     for each lock l_i released by op do
13:         for each conflicting lock l_j in LT do
14:             if each of possible orders indicates incompatible locks then
15:                 return an ME violation in bug descriptor;
16:             else
17:                 Deduce a ww dependency;
18: procedure FirstUpdaterWins
19:     S^T ← the visible snapshot time interval of op;
20:     OV ← use traces to construct ordered versions of each record;
21:     for each version x^i in the write set of op do
22:         for each version x^j in OV do
23:             if each of possible orders indicates concurrent versions then
24:                 return an FUW violation in bug descriptor
25:             else
26:                 Deduce a ww dependency;
27: procedure SerializationCertifier
28:     for each dependency d deduced from T do
29:         DG ← DG ∪ {d}
30:         if d causes a prohibited dependency pattern then
31:             return an SC violation in bug descriptor
```

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: **Visible Snapshot Time Interval.** Suppose an operation *op* sees a snapshot of the database as of an operation op_s. Then, the visible time interval of the snapshot saw 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 *visible snapshot time interval* contains the exact time point when a specific snapshot is visible. 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, an operation sees the snapshot as of 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, an operation sees the snapshot as of 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 *visible snapshot 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.3).

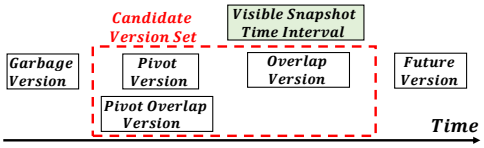


Fig. 5. Minimizing Candidate Versions

To address this issue, we propose to leverage the visible snapshot 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 visible snapshot time intervals.

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

Fig. 5 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 visible snapshot time interval. Specifically, the *future version* arrives after the visible snapshot time interval, so it is invisible to the read operation. In contrast, the *garbage version* arrives before the visible snapshot 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.

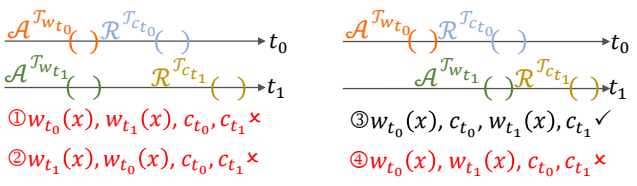
Theorem 2: The candidate version set contains a minimum number of versions that are possibly visible to a given read operation.

Additionally, if the candidate version set has only one version matching the read set of a read, then we can deduce that the read must happen after the write creating this matched version, even though the time intervals of the two operations are overlapped. That is, we can still confirm that the read transaction has a wr dependency on the write transaction creating this matching version. 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* violations 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 visible snapshot 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 before 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.



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

Fig. 6. 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 on the record x with two write operations $w_{t_0}(x)$ and $w_{t_1}(x)$. When t_0 and t_1 commit, their commit operations c_{t_0} and c_{t_1} would release the locks posed on x. $\mathcal{T}_{w_{t_0}}$, $\mathcal{T}_{w_{t_1}}$, $\mathcal{T}_{c_{t_0}}$ 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 the 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. 6(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. 6(b), there exists only one order (③) in which a ww dependency can be deduced. Thus, we deduce a ww dependency (note we assume there are no bugs inside the DBMS when deducing dependencies), and take this dependency with the dependencies deduced from other verifying mechanisms to check whether contradictory dependencies exist.

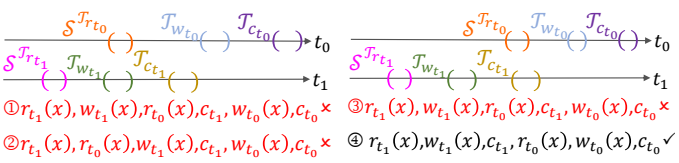
Theorem 3: Given two transactions t_0 and t_1, for any overlapped time intervals of two conflicting locks, there exists at most one possible order in which a ww dependency can be deduced. Specifically, each of the other possible orders is identified to have incompatible locks.

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



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

Fig. 7. First Updater Wins Verification Examples

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 visible snapshot, version installation and transaction commit/abort to identify whether there exist other concurrent versions during the period of a transaction execution. Specifically, the visible snapshot 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. 7(a). The visible snapshot time interval of t_0 (i.e., $\mathcal{S}^{\mathcal{T}_{r_{t_0}}}$) lies between the time intervals of visible snapshot and transaction commit of t_1 (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. 7(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.

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 visible 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 visible snapshot, 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 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 [11], [20]–[22]. 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.

Specifically, the concurrency control protocols inside the DBMSs often take advantage of *SC* to guarantee the conflict serializability, and each protocol has its specific "certifier". For example, the *SSI* protocol of PostgreSQL uses two consecutive rw dependencies as its certifier. Specifically, the certifier achieves this goal by avoiding *write skew* anomalies for *snapshot isolation* [41]. *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, its *TO* protocol uses the timestamp ordering as its certifier. Specifically, the certifier does not allow a transaction with an older timestamp to have a dependency on the transaction with a newer timestamp. Thus, the cycles would never appear in *DG*. More details of the certifier can be found in [45].

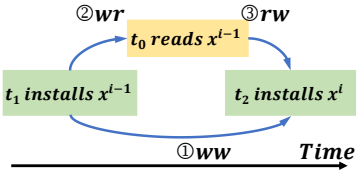


Fig. 8. Deducing Dependencies

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. 8 illustrates the deducing method. Each rectangle represents a version installation time interval (colored green) or a visible snapshot time interval (colored yellow). If there exist two dependencies which indicate that transaction t_2 has a ww dependency on t_1 and t_0 has a wr dependency on t_1, then we can deduce that t_2 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 usage. 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 pruned without affecting *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 visible snapshot 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*.

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) How efficient are *two-level pipeline* and *mechanism-mirrored verification*? (Section VI-A and VI-B) Can the throughput of *Leopard* surpass that of a DBMS? (Section VI-C)
(2) How effective is *Leopard* to deduce transactions dependencies 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 and Settings. *Leopard* is implemented by Java 8. Our experiments are conducted on four servers connected using 1 Gigabit Ethernet. Each server is equipped with 2 Intel Xeon Silver 4110 @ 2.1 GHz CPUs, 120 GB memory, and 4 TB HDD disk. We deploy one centalized DBMS, i.e., PostgreSQL ($v12.7$) and one distributed DBMS, i.e., OceanBase [62] ($v3.1$), to explore technical designs in *Leopard*. Experiment results on more DBMSs are put in [66]. Note that, OceanBase is deployed on three machines with three replicas and a client is deployed on one node. In default, *Leopard* switches on all optimizations and garbage collections, and we take PostgreSQL to demonstrate the performance of our technical designs.

Comparison Work. *Cobra* [11] and *Elle* [19] are the state-of-the-art work for verifying ILs. *Cobra* only verifies serializable key-value stores. It enumerates all possible dependency graphs on which it verifies ILs by cycle searching. *Elle* requires its workload to make all dependencies manifest, based on which it then builds a dependency graph and does the cycle search.

Workload. TPC-C [67] and SmallBank [68] are used to check the ability of *Leopard* in verifying workload with complex application logic. TPC-C dose not suffer from *write skew* anomalies [69], so it is insufficient for serializable verification. However, SmallBank is to benchmark the strategies to achieve serializable by eliminating *write skew* anomalies. Thus, we take both of them in our experiments. By default, they populate database with *scale factor=1*. Specifically, SmallBank populates the database with $1,000$ accounts, while TPC-C populates the database with 1 warehouse. We also take *BlindW* designed by *Cobra* [11] for our evaluation. In default, it creates a table sized $2K$ (*scale factor=1*) with values of 140 fixed-length strings; each transaction has 8 operations and keys are accessed uniformly under *serializable*. For different evaluation purposes, *BlindW* is extended to three variants:

(1) *BlindW-W* contains 100% *blind-write* transactions (a transaction writes a value without reading it) with uniquely written values. Because *blind-write* does not access the record version before creating a new version, it is a tough scenario for tracking ww dependencies (Section VI-D).

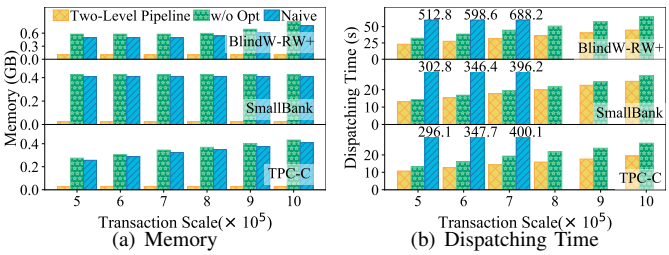


Fig. 9. Two-Level Pipeline Performance

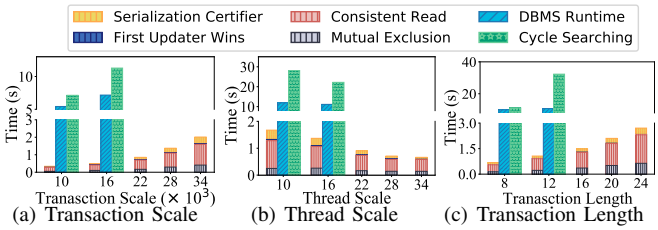


Fig. 10. Verification Time on Blind-RW+ Workload

(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 sort and dispatch traces. To show its efficiency, we compare it with the naive approach, which collects traces from multiple clients and sorts them in global buffer. To demonstrate the effectiveness of optimizations to *two-level pipeline*, we also compare *Leopard* with the one without optimization, i.e., *w/o Opt*. As 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.

As varying transaction scales, we collect the memory usage and the dispatching time of *two-level pipeline* in Fig. 9. Our approach is consistent far better than the other two approaches on memory usage and dispatching time. In Fig. 9(a), the naive approach has similar maximal memory consumption with *Leopard w/o Opt* 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 naive approach has the worst dispatching time as in Fig. 9(b). The reason is that the naive approach sorts traces synchronously, while *Leopard* sorts a batch of traces asynchronously. Specifically, running *BlindW-RW+* on PostgreSQL, it has the maximum traces, which then has the longest dispatching time. When transaction scale $>7\times10^5$, the naive approach takes too much time to dispatch traces and we do not plot its time any more. In short, *two-level pipeline* can dispatch traces efficiently.

B. Mechanism-mirrored Verification

Verifying ILs can be decomposed into verify the execution of the four mechanisms. The naive approach is to build a

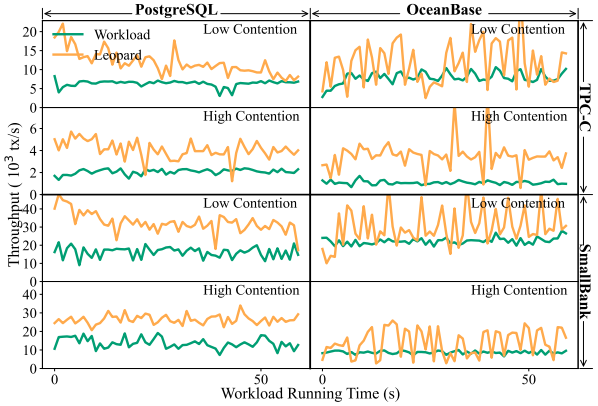


Fig. 11. DBMS Throughput vs. Leopard Throughput

dependency graph and do cycle searching, whose efficiency is significantly affected by the number of traces. We then propose an *mechanism-mirrored verification* by simulating the workflow of concurrency control protocols inside a DBMS, which only occupies a small part of the DBMS runtime theoretically. We vary the critical factors affecting verification time, including 1) transaction scale to control the number of transactions; 2) thread scale to control the data contentions; 3) transaction length to control the number of operations in a transaction. Since *BlindW-RW+* can control the three factors quantitatively, we compare the verification time of our approach with the DBMS runtime and the naive cycle searching approach in Fig. 10.

The verification time of our four mechanisms is linear with transaction scale as shown in Fig. 10(a), which benefits from 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 accordingly, as shown in Fig. 10(b). Increasing transaction length expands the read or write set in a transaction. The verification time then linearly increases with transaction length, as shown in Fig. 10(c). This is because the complexity of our verification method is linearly with the size of read or write set. Note, ours can significantly outperform the two baseline approaches, so we do not plot their figures for thread scale>16, transaction scale>16K and length>12. In short, *Leopard* can efficiently verify ILs.

C. Comparison with DBMS Throughput

To show the verification efficiency of *Leopard*, we compare the performance of DBMSs with that of *Leopard* by running TPC-C and SmallBank on both PostgreSQL and OceanBase. Since contentions greatly affect DBMS throughputs as well as operation overlappings, we launch the experiments under both high and low contentions. Specifically, on PostgreSQL, for a high contention, we run TPC-C and SmallBank with 24 threads and 1 scale factor; for a low contention, we run the workloads with 24 threads, each of which is coupled with a scale factor. Since OceanBase is deployed on three machines, we expand the workloads in proportion, that is 72 threads and 3 scale factors (resp. 72 threads and 72 scale factors) for a high (resp. low) contention. We run each workload for 300s and send the traces to *Leopard* every $0.5s$.

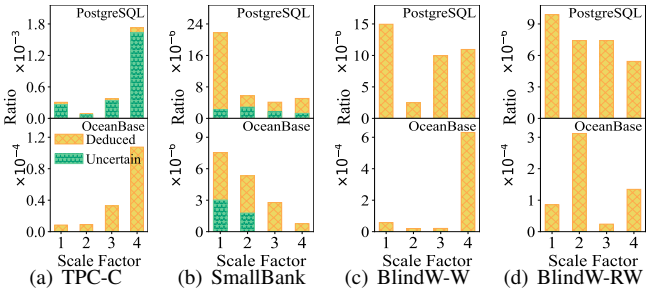


Fig. 12. Deducing Dependencies

Fig. 11 shows the results. Except running TPC-C on OceanBase under a low contention, for all the other workloads, the throughput of *Leopard* dominates that of DBMSs. Even though SmallBank on OceanBase has the highest throughput under a low contention, it has simpler application logic than TPC-C, i.e., short transactions with only item-read/write and no range-read, which greatly decreases the burden on verification. So *Leopard* is fast enough to accomplish verification for SmallBank. But when running TPC-C on OceanBase under a low contention, the high throughput with complex application logic burdens *Leopard* in verification. As verifying ILs usually focuses on the conflicting operations, operations which are not conflict can be verified in parallel. Then we can scale *Leopard* to multiple instances by partitioning traces. Traces belonging to a data partition can be mapped to a *Leopard* instance, and multiple *Leopard* instances can work in parallel (more details of *Leopard*'s distributed deployments are presented in [66]). So when running TPC-C on OceanBase under a low contention, 3 *Leopard* instances are depolyed to do verification. In short, *Leopard* can be scaled out for the high throughput DBMS.

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 IL verification, which can expose the order between conflict operations. To demonstrate its effectiveness, we run TPC-C, SmallBank, *BlindW-W* and *BlindW-RW* on PostgreSQL and OceanBase for 20 minutes to cover trace overlappings as much as possible. As shown in Fig. 12, we plot the ratio of the deduced dependencies (yellow bar).

The ratio of uncertain dependencies is generally a small number ($< 10^{-3}$). The complex application logic in TPC-C and SmallBank makes some uncertain dependencies cannot be deduced (green bar). Specifically, in TPC-C, many transactions read/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 SmallBank, transaction *amalgamate* always writes the same values, and duplicate values can not be distinguished in its candidate version set. Since *blind-writes* in both *BlindW-W* and *BlindW-RW* write distinct values, *Leopard* can expose these dependencies well as shown in Fig. 12(c)-12(d). In short, *Leopard* can perceive more dependencies effectively.

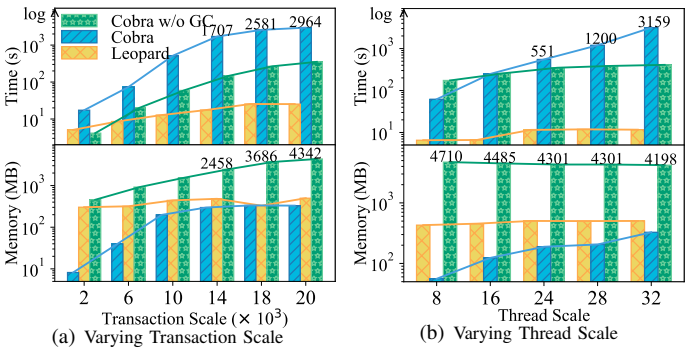


Fig. 13. Comparison with Cobra on Blind-RW Workload

E. Comparison with Cobra on Efficiency

Cobra is designed only to verify serializable key-value stores [11]. So we set the IL of PostgreSQL as *serializable*. Moreover, it enables garbage collection if meeting fence transactions which are inserted every 20 transactions in our experiment; *Cobra w/o GC* disables fence transactions.We report the verification time and memory usage in Fig. 13. 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. 13(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, but *Cobra* has the lowest verification efficiency. Specifically, for $20K$ transactions, *Leopard* outperforms *Cobra* by $114\times$ in verification efficiency with almost the same memory usage. In Fig. 13(b), 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 $271\times$ slower than *Leopard* when thread scale=32. Thus, *Leopard* has a much better scalability $w.r.t$ the scales of transactions and threads.

F. Comparison with Elle on Bug Cases

We run *Leopard* on several popular DBMSs. Although these DBMSs have been tested by other testing tools, e.g., *Elle* [19], we still discover 23 bugs (13 fixed, 15 confirmed and 8 open reported), demonstrated in [70]. Here, we explain one bug.

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 *consistent read*.

```
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}✖
```

Elle depends on the cycle in dependency graph to verify ILs. But the dependency graph of the above bug case has no cycle, so it fails to find it. *Leopard* is more general in verifying ILs without specific requirements on workload and can expose more subtle bugs. More bugs and performance comparison with *Elle* refer to our technical report [66].

VII. Related Work

Verifying isolation levels (IL) is usually achieved by elaborating workloads or instrumenting source codes of DBMSs [4], [11], [19], [71]. However, none of current work is general to verify various ILs in a black-box mode with arbitrary workloads. *Cobra* [11] can only exposes serializability violations of transactional key-value stores based on a workload following a specific application logic. By an expensive graph traverse, it prunes garbage transactions. *Elle* [19] specifies a short workload to expose version orders in history and its verification is only based on a graph cycle detection. Both *Cobra* and *Elle* can not verify arbitrary workloads, e.g., TPC-C. Mai et.al [4] diagnose violations of the ACID properties in a white-box method by injecting power faults while replaying its workloads. Yu et.al [71] design a DBMS that can provide verifiable proofs of transaction correctness and semantic properties. But it is only verifiable for the *serializable* IL. In contrast, *Leopard* can verify the workload with any application logic and is not limited to a specific IL.

Some work proposes to detect anomalies in application workloads [3], [17], [20]–[22] instead of DBMSs. *IsoDiff* [20] takes an analysis of application codes to debug anomalies (caused by weak isolations) on the representative subsets of dependency graphs. *Rushmon* [21] monitors real-time anomalies caused by the asynchronous algorithm for the inconsistency-tolerant applications on weak isolation systems. It takes the idea of serializability to achieve detection by sampling the dependency graph, which is not a comprehensive checking method. *ConsAD* [22] quantifies isolation anomalies by detecting cycles in the dependency graph. But it costs a lot to analyze the application logic to track dependencies. Todd et.al [3] aim to detect potential isolation anomalies in web applications. They reason the possible concurrent interleavings among clients to generate workloads to violate the ACID principle. In contrast, *Leopard* aims to verify various ILs in DBMSs without modifying application logic.

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. *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 deducing all dependencies, and digging up all bugs is still impossible. We leave it as our future work.

Acknowledgements: This work is supported by NSFC (62072179, 62202171), Alibaba AIR Project and Shanghai Pujiang Program (21PJ1403200).

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. 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

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

[18] 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.

[19] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies from experimental observations," *arXiv preprint arXiv:2003.10554*, 2020.

[20] 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.

[21] 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.

[22] 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.

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

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] N. Singh and A. K. Singh, "Data privacy protection mechanisms in cloud," *Data Science and Engineering*, vol. 3, no. 1, pp. 24–39, 2018.

[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] "Leopard IL_Description," https://github.com/DBHammer/Leopard/blob/main/IL-Description.pdf.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li, "opengauss: An autonomous database system," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 3028–3042, 2021.

[52] "yugabyteDB," https://www.yugabyte.com/.

[53] P.-Å. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos, "Real-time analytical processing with sql server," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1740–1751, 2015.

[54] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1041–1052.

[55] F. Li, "Cloud-native database systems at alibaba: Opportunities and challenges," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2263–2272, 2019.

[56] P. Bhatotia, A. Wieder, İ. E. Akkuş, R. Rodrigues, and U. A. Acar, "Large-scale incremental data processing with change propagation," in *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11)*, 2011.

[57] "Rocksdb," https://www.rocksdb.org/.

[58] "SQLite," https://www.sqlite.org/index.html/.

[59] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, J. Leach *et al.*, "Foundationdb: A distributed unbundled transactional key value store," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2653–2666.

[60] "Singlestore," https://www.singlestore.com/.

[61] 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.

[62] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi *et al.*, "OceanBase: a 707 million tpmC distributed relational database system," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3385–3397, 2022.

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

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

[65] 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.

[66] "Leopard Technical Report," https://github.com/DBHammer/Leopard/blob/main/Technical-Report.pdf.

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

[68] 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.

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

[70] "Leopard Bug List," https://github.com/DBHammer/Leopard/blob/main/Bug-List.pdf.

[71] 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."

[72] "Robert Tarjan," Depth-first Search and Linear Graph Algorithms (SWAT), 1971.

[73] "Elle github," https://github.com/pingcap/tipocket/tree/master/testcase/rw-register.

APPENDIX

I. Theorem Proof

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:

$$\mathcal{T}.ts_{bef} \leq \min_{\mathcal{T}_i \in global} \mathcal{T}_i.ts_{bef} \quad (1)$$

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

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

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

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 proven.

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 visible snapshot 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 proven.*

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 proven.

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.

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 proven.*

II. Leopard Implementation

A. Single Machine Deployment

Fig. 14 depicts our implementation details on a single machine. A workload consists of multiple clients. ❶ Each client issues the requested operations to a DBMS. ❷ Each client also receive results responded by a DBMS. Each client encodes an responded operations as an interval-based trace (trace for short). Note that, as described in Appendix II, we shard a database into several data partitions, and each partition is coupled with a parallelism unit. ❸ According to the data partition accessed by the operation, we branch off traces into the corresponding parallelism unit. Parallelism unit retrieves traces from clients and attempts to verify whether the workload satisfies the definition of isolation levels.

Specifically, the *local buffer*s cache the streaming traces from each client asynchronously. ❹ In the meanwhile, the *global buffer* fetches and sorts all traces from each *local buffer*. Based on sorted traces from *global buffer*, *Leopard* launches the verification. The verification includes three phases, including context preparation, forking and garbage

collection. ❺ Context preparation installs sorted traces into three contexts, i.e., *version orders*, *dependency graph* and *lock table*. Specifically, *version orders* maintains the version evolution of each record for verifying *consistent read* and *first updater wins*, *dependency graph* captures the transaction dependencies for verifying *serialization certifier*, and *lock table* temporarily saves the locks acquired by each operation for verifying *mutual exclusion*. ❻ Then, based on the prepared context, it forks four threads to verify the four implementation mechanisms in parallel. ❼ Finally, for catching up with a long running workload, garbage collection aggressively cleans up the contexts that have nothing to do with the successive verification.

B. Multiple Machine Deployment

Fig. 15 depicts the distributed deployments of our *Leopard* framework. Not only the tested DBMS can be deployed in a distributed environment, but also the test clients, the trace sorting and verifying components can be scaled to multiple machines. Next, we discuss the key issues of *Leopard*'s distributed deployments.

Clock Synchronizations among Test Clients. As we need to log the time interval of each database operation, the clock synchronization of timestamps is a critical issue when the test clients are deployed on multiple machines. We propose to make use of the clock synchronizing method inside the DBMSs. Specifically, if the data nodes are deployed in a single cluster with low network latencies, a centralized logical timestamp generation method is more appropriate. For example, both TiDB [7] and OceanBase [62] use a centralized timestamp oracle to allocate logical timestamps for all transactions. Moreover, OceanBase suggests that applications should avoid expensive acquisitions of cross-region timestamps. Our experimental results show that the latency of each timestamp acquisition is much smaller (at least $45\times$) than the latency of database operation.

Scale Leopard to Multiple Instances. As the verifying process of isolation levels usually focuses on the conflicting operations which accesses the same records, we can deploy multiple Leopard instances to do verification in parallel. That is, we can make use of the database sharding information provided by the DBMSs (or borrow ideas from popular database sharding algorithms), and divide all workload traces into partitions accordingly. Then, we can deploy each Leopard instance on a single node and use it to verify the traces belonging to one (or more) specific partitions. Note that, as the *CR*, *ME* and *FUW* mechanisms only concern the data conflicts, they can be served by each node individually. For the *SC* mechanism, as we we need construct the dependency graph whose nodes are transactions and edges are dependencies between transactions, it might need to verify the transaction dependencies which across different nodes. Nevertheless, it needs a low communication cost if the data nodes are deployed in a single cluster with low network latencies. This is because the verification of each cross-node dependency only needs one network round trip. Takes verifying *serialization certifier* of PostgreSQL as an example. It detects wheter there exist two consecutive rw dependencies. In the worst case, verifying *serialization certifier* only requires two network round trips.

III. More Experiments

In default, our experimental environment is the same as the one declared in our paper. Specifically, TiDB and OceanBase are deployed on three servers with three replicas and a client is deployed on one server. MySQL and PostgreSQL are deployed on one server individually.

A. Evaluation on Two Optimizations of Two-level Pipeline

An online transaction processing always continuously produces a massive amount of traces, which pose a significant challenge on the high-performance trace sorting. External merge sort is a traditional approach to sort massive amounts of data. We launch an experiment to compare it with our approach. We run the standard benchmark TPC-C with 24 threads and 1 warehouse on PostgreSQL and MySQL. In Fig. 16, as varying transaction scale, we collect the dispatching time of our approach and the external merge sort. From results, the dispatching time of our method outperforms the external merge sort method by up to $3.6\times$. Thus, taking advantage of two optimizations based on multi-thread and batch processing, our sorting method is more efficient than the external merge sort.

B. Evaluation on Clock Synchronization Cost

We conduct experiments to compare the latency of clock synchronization with the latency of operations in distributed DBMSs. Specifically, the clock synchronization is implemented on three servers. One server works as the centralized timestamp oracle, and the other two servers work as the clients which issue 100 threads to acquire timestamps concurrently. For the distributed DBMSs, they are deployed on three servers with three replicas.

We run standard benchmark TPC-C on two distributed DBMSs, i.e., TiDB and OceanBase, and report the tail latency of each operation. Specifically, as the workload contention heavily affects the operation latency, we simulate both the high and low contention scenarios. For a high contention, we run TPC-C with 72 threads and 3 warehouses (denoted as high contention), while for a low contention, we run TPC-C with 72 thread and 72 warehouses (denoted as low contention). For comparison, we report the tail latencies of clock synchronization service on the cluster which uses different network protocols, including TCP/IP and RDMA. Fig. 17 shows the results. We can see that,

- Under a high contention, the 50%/90%/95% latencies of TiDB (resp. OceanBase) are 60/1749/4125$\times$ (resp. 60/1158/2266$\times$) higher than that of the clock synchronization service based on the TCP/IP protocol; Based on the RDMA protocol, the 50%/90%/95% latencies of TiDB (resp. OceanBase) are 1322/38590/90884$\times$ (resp. 1150/25538/49930$\times$) higher than that of the clock synchronization service.

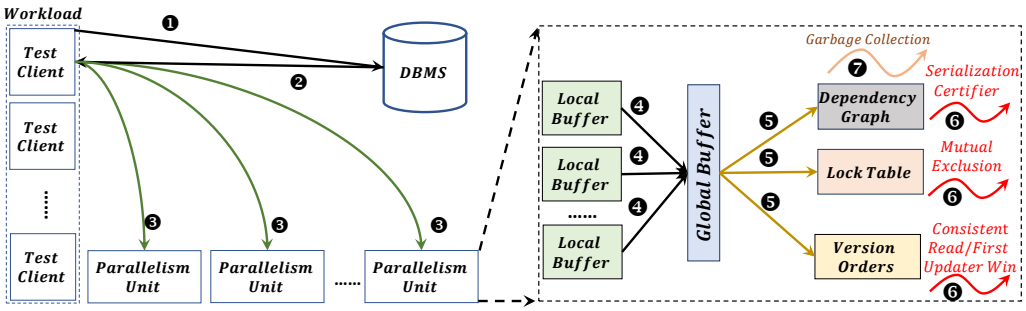


Fig. 14. Implementation Details of Leopard on A Single Machine

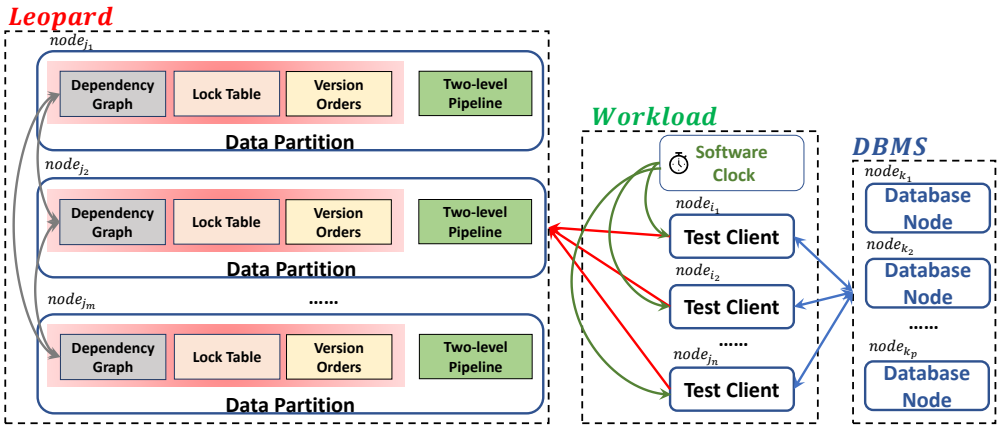


Fig. 15. Distributed Deployments of Leopard

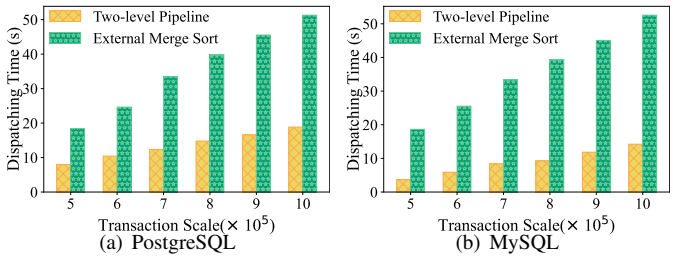


Fig. 16. Comparing Two-level Pipeline Sort with External Merge Sort

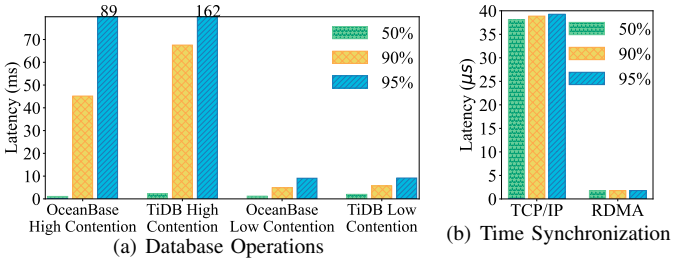


Fig. 17. Comparing Latencies of Time Synchronization and Database Operations

- Under a low contention, the 50%/90%/95% latencies of TiDB (resp. OceanBase) are 47/154/229× (resp. 45/116/229×) higher than that of the clock synchronization service based on the TCP/IP protocol. For the RDMA protocol, the 50%/90%/95% latencies of TiDB (resp. OceanBase) are 1035/3405/5049× (resp. 978/2554/5046×) higher than that of the clock synchronization service.

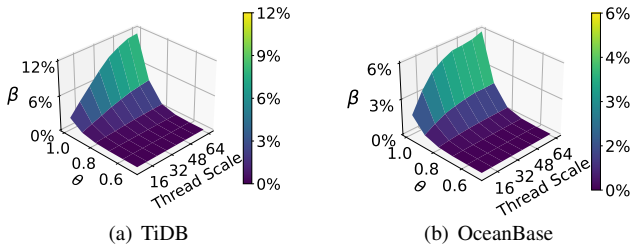


Fig. 18. Overlapping Ratio β

Usually, a distributed DBMS uses an extraordinarily complex protocol combination (e.g., the consensus protocol and the atomic commit protocol) to achieve transactional consistency over multiple remote machines, which would lead to a certain number of synchronous network round trips and disk data accesses. However, a logical timestamp acquisition only needs one network round trip. For DBMSs with particularly high throughputs, we suggest to use the RDMA network protocol to reduce the clock synchronization cost.

C. Overlapping Ratio on Distributed DBMSs

Workload contention has an obvious impact on overlapping ratio β. The standard benchmark YSCB-A can easily control workload contention by varying skew parameter θ and thread scale. Thus, we run YCSB-A on a table with 1 million records and the read/write ratio set as 4:1 to demonstrate the change

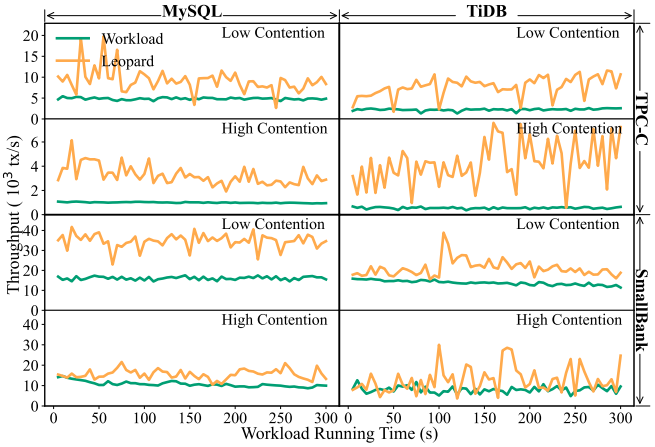


Fig. 19. Workload Throughput vs. Leopard Throughput on MySQL and TiDB

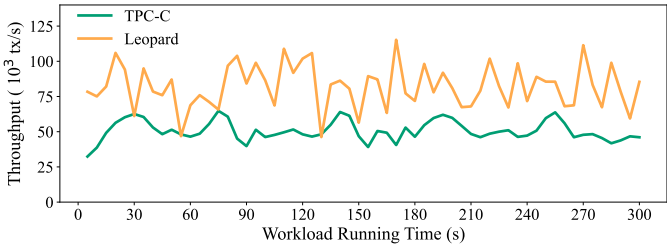


Fig. 20. Multiple-Leopard Throughput vs. High Throughput of OceanBase

of β on two distributed DBMSs, i.e., TiDB and OceanBase, as shown in Fig. 18(a) and Fig. 18(b). From the results, only when the skew parameter θ and thread scale are increased at the same time, β increases significantly. For the maximum value of β, TiDB's β is larger than the one of OceanBase. When increasing θ to 1 and thread scale to 64 on TiDB, β is still less than 10%. OceanBase has a smaller β than TiDB. This is because TiDB treats each transaction as a distributed transaction, while OceanBase tries to avoid the distributed transactions through the database sharding. The latency of DBMS processing operations increases the ratio of overlapping. We can see that although the overlapping ratio β in the distributed DBMS might be higher than the centralized DBMS, the value of β is still relatively small in all cases.

D. More Comparison with DBMS Throughput

To demonstrate the ability to keep up with DBMS throughputs, we launch the same experiments on two more popular DBMSs, i.e., MySQL ($v5.7$) and TiDB ($v5.0$). Fig. 19 shows the results. *Leopard* can be easily scaled out to keep up with the high throughput DBMS service.

In addition, to further illustrate the scalability of *Leopard*, we have also conducted experiments by running TPC-C on OceanBase which is deployed on 20 cloud nodes connected using 1 Gigabit Ethernet. Each cloud node is equipped with 32 cores, 64 GB memory, and 100 GB HDD disk. Specifically, we deploy our Leopard only on 10 nodes. As shown in Fig. 20, we can see that the throughput of OceanBase can be as high as $65K$ transactions per second. Note, in TPC-C, each transaction consists of $15 \sim 17$ operations on average. The peak throughput of *Leopard* can be as high as

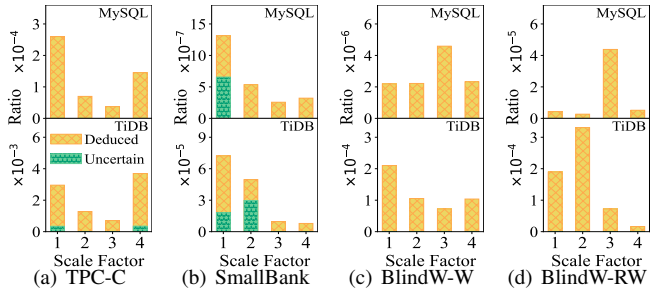


Fig. 21. Deducing Dependencies on MySQL and TiDB

$115K$ transactions per second, which is much higher than the throughput of OceanBase. This indicates that *Leopard* scales well in the distributed setting.

E. Deducing Dependencies on More DBMSs

To demonstrate the generality of our approach on deducing dependencies, besides PostgreSQL, we also run the same experiments with four benchmarks, i.e., TPC-C, SmallBank, BlindW-W and BlindW-RW, on MySQL and TiDB to demonstrate the generality of our approach. Fig. 21 show the experiment results, which are similar with the results of PostgreSQL and OceanBase. In summary, *Leopard* can expose more uncertain dependencies effectively.

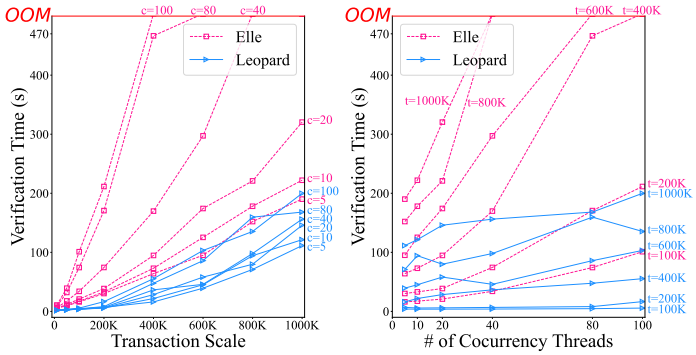
F. Comparison with Elle on Efficiency

Elle [19] carefully designs some specific workloads that can manifest dependencies for IL verification. Specifically, it takes the two properties to deduce dependencies, including *recoverability* and *traceability*. *Recoverability* means that every version of a read operation can be mapped to a specific write operation. *Traceability* means that the complete version order can be obtained by a read operation. Based on these two properties, *Elle* expects to deduce dependencies easily.

To find isolation anomalies, *Elle* puts these deduced dependencies into a graph, and search for cycles. Specifically, *Elle* applies Tarjan's algorithm to identify a cycle [72]. However, Tarjan's algorithm is superlinear with the number of transactions in the graph. To make matters worse, the number of transactions is massive for a long running workload. *Elle* then has poor scalability to a long running workload. To demonstrate it practically, we launch an experiment to compare the performance between *Leopard* and *Elle*.

We take the workload from the github repository of *Elle* [73], which can be verified by *Elle*. Specifically, the workload is composed of randomly generated transactions each of which covers one to five item-read/write operations and runs on a table of 1000 records. We run the workload on MySQL as varying the number of transactions (denoted as t) and the number of concurrency threads (denoted as c). Fig. 22 shows the verification efficiency of *Leopard* and *Elle*.

From results, *Elle*'s verification time rises dramatically with the number of transactions and the number of concurrency threads, which is caused by the superlinear time complexity of Tarjan's algorithm. By contrast, *Leopard*'s verification time is linear with the number of transactions and the number



(a) Time vs. # of Transactions (b) Time vs. # of Concurrency Threads

Fig. 22. Comparison with *Elle* on Efficiency

of concurrency threads. *Elle* lacks the garbage collection for the dependency graph, so it is not suitable for verifying a long running workload that contains massive transactions and dependencies. Moreover, the graph expands significantly as the number of concurrency threads increases, which has a large number of conflicting operations (i.e., many dependencies on the graph). So, when the number of transactions or concurrency threads is up to a specific threshold, *Elle* terminates the verification and throws the out of memory exception (OOM). For example, when the number of concurrency threads is 100, *Elle* throws OOM at the number of transactions $400K$.

G. More Bug Case Demonstrations

```
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
```

Bug 1: Dirty Write on TiDB. 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. We report this bug to *TiDB*, which is confirmed and fixed.

```
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
```

Bug 2: Violating Mutual Exclusion on TiDB. 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 write locks. We report this bug to *TiDB*, which is confirmed and fixed.

```
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}✖
```

Bug 3: A Query Returning Two Versions of a Record on TiDB. 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 bug to *TiDB* and confirmed that it is a known bug.