Leopard: A General Test Suite for Isolation Level Verification

Peiyuan Liu
East China Normal University
pyliu@stu.ecnu.edu.cn

Siyang Weng
East China Normal University
syweng@stu.ecnu.edu.cn

Keqiang Li
East China Normal University
kqli@stu.ecnu.edu.cn

Lyu Ni
East China Normal University
lni@dase.ecnu.edu.cn

Chengcheng Yang*
East China Normal University
ccyang@dase.ecnu.edu.cn

Rong Zhang*
East China Normal University
rzhang@dase.ecnu.edu.cn

Weining Qian
East China Normal University
wnqian@dase.ecnu.edu.cn

Dian Qiao
Huawei Technologies Co., Ltd.
qiaodian@huawei.com

ABSTRACT

Isolation levels (ILs) play as the correctness contract between applications and a database management system (DBMS). They are implemented to ensure the correct database status $w.r.t.$ complex business logic under concurrent transaction processing. However, verifying the IL implementations of a DBMS has always been a challenging task due to the 1) inaccessibility of the codebase; 2) non-deterministic execution orders among concurrent transactions; 3) various IL definitions and implementations and 4) requirement of highly efficient IL verification. To expose potential IL anomalies in DBMSs, we propose to design and implement a general IL test suite *Leopard*. *Leopard* can be configured to verify various IL implementations in different DBMSs in a black-box way by analyzing client-side workload traces. With the help of *Leopard*, we have successfully discovered 24 bugs (14 fixed, 16 confirmed and 8 open reported) in several commercial DBMSs.

1 INTRODUCTION

Isolation Level (IL) was first introduced with the name "degrees of consistency" [9]. It serves as the correctness contract between applications and DBMSs. In general, the higher the IL, the lower the performance. Commercial DBMSs support different ILs to allow developers to make a trade off between consistency and performance [9]. In our recent work [11], we have found that current IL verification tools lack generality and have low verification efficiency, causing serious limitations in the application scenarios.

More specifically, the IL verification methods proposed in existing work can be broadly classified into kernel-oriented [2, 12, 16] and workload-oriented methods [1, 5, 10, 18]. However, there exist four challenges that could not be well addressed by these methods. The first challenge is the inaccessibility of the codebase with complex code logic inside DBMSs, especially for cloud services provided by a third party [19]. This makes the kernel-oriented methods inapplicable since they rely on instrumenting kernel codes to catch the internal execution state after writing DBMSs. The second challenge is the non-deterministic execution order among concurrent transaction operations, which would make it difficult to capture the transaction dependencies. In light of this, the workload-oriented methods propose to impose specific restrictions on the workloads such that the transaction dependencies can be easily obtained. Obviously, they cannot be adapted to arbitrary workloads. The third challenge is the existence of various ILs in DBMSs, and even for the same IL, the implementations in different DBMSs might have subtle differences. However, none of existing work is general enough to handle the correctness verification for any type of IL. The last challenge is the requirement of highly efficient IL verification such that the IL verification can be performed in an online fashion. Then, the bugs can be reported and fixed in time. However, previous studies often detect IL anomalies by performing complex cycle searches on the dependency graph, which makes them fail to scale to a high-throughput DBMS.

To address above challenges, we propose *Leopard*, a novel black-box IL testing suite to accomplish the target of *generality*, *efficiency* and *scalability* for IL verification. It provides an *implementation mechanism mirrored verification* solution. That is, while issuing a workload to a DBMS, *Leopard* also simulates its concurrency control protocols (CCPs) to see if the DBMS results conform to the possible schedules allowed by these mechanisms. Firstly, to avoid touching the codebase of DBMSs or specifying the workload, we propose to collect *time interval* workload traces which contain the execution time interval of each transaction operation from the client-sides. Then, we further design a novel time interval based dependency deduction approach which could effectively track dependencies among concurrent transactions. Secondly, to be general for verifying diverse ILs, we launch a thorough study of various implementations of ILs and finally abstract four types of implementation mechanisms, which can be orchestrated to realize all ILs in our investigated 18 commercial DBMSs. Finally, to keep up with the DBMS's throughput, we design a *two-level pipeline* algorithm for efficient trace sorting and delivering, and base on which we design an *implementation mechanism mirrored verification* to replay data evolution by simulating the implementation of concurrency control protocols inside DBMSs. To perform online verification for distributed DBMSs, we also propose a distributed verification method by a data-oriented trace sharding such that *Leopard* can scale out to the system under test.

*Rong Zhang and Chengcheng Yang are corresponding authors.

In this demonstration, we will showcase three aspects of *Leopard* by providing an online web service [7]. 1) Generality: we adapt *Leopard* to different DBMSs with various ILs and different workloads. 2) Efficiency: we compare the efficiency of *Leopard* with the throughput of DBMS. 3) Scalability: we scale out *Leopard* by configuring parallel verification instances in a distributed way. *Leopard* can be adapted to MySQL/PostgreSQL-compatible DBMSs (e.g., TiDB and OpenGauss) and has helped us find 24 bugs [13].

2 SYSTEM IMPLEMENTATION

In this section, we introduce the workflow of *Leopard* and elaborate its design details. Specifically, *Leopard* implements its IL verification with three steps: 1) collecting and sorting client-side workload traces, 2) deducing the real-time database states (i.e., data evolution) by replaying traces, and 3) verifying the data evolution by mirroring the data access procedure inside the DBMS in a distributed way.

2.1 System Workflow

Fig. 1 illustrates the workflow of *Leopard*. It provides a general correctness verification for IL implementations in a black-box way. Firstly, on the *workload controller*, each test client thread keeps submitting transaction operations and collecting client-side traces locally. Then, the *trace dispatcher* distributes local traces to each individual *verifier* according to the trace sharding policy. Note, multiple *verifier* instances can be deployed and run in parallel $w.r.t.$ database throughput. Specifically, each *verifier* instance takes actions of sorting traces in the *trace manager*, constructing data structures associated with concurrency control protocols and catching data evolution in the *state evolver*, and checking IL anomalies in the *anomaly checker*. Finally, the *dashboard* presents the verification reports.

2.2 Workload Controller

The complex concurrent scheduling of DBMSs makes it rather difficult to capture dependencies among transactions. On the one hand, the workload-oriented methods (e.g., Elle [10] and Cobra [5]) rely on constructing specific workload patterns to facilitate exposing the dependencies among transactions easier. However, this greatly limits their application scenarios and test ability. On the other hand, some kernel-oriented methods [2] instrument the source code of a DBMS to expose the exact execution timestamps of transaction operations inside the DBMS on the server side. Then, they construct dependencies of data evolution based on the collected timestamps. However, instrumenting source code of a DBMS is laborious and even impossible (e.g., for a cloud DBMS), and this might also affect the normal transaction executions. To address these issues, we propose to leverage client-side operation timestamps to infer transaction dependencies. By logging operation traces in the client side, we can avoid specifying the application logic or modifying the DBMS kernel.

More specifically, on the *workload controller*, each test thread connected to the DBMS continuously send transaction operations to the server, and the trace of each operation (including *commit* and *abort*) is logged on the client side. Specifically, the trace of an operation consists of 1) the timestamp before execution ts_{bef}, i.e., sent by the client thread; 2) the timestamp after execution ts_{aft}, i.e., result arriving at the client side; 3) operation type and the data accessed by the operation. For a read (resp. write) operation, we log its belonging transaction t and its read set rs (resp. write set ws). We formalize the trace of an operation in a given transaction t by $\mathcal{T} = \{ts_{bef}, ts_{aft}, r_t(rs)/w_t(ws)/a_t/c_t\}$. Note, the operation's exact execution time inside the DBMS could not be obtained in the black-box mode, but is covered by the time interval specified by ts_{bef} and ts_{aft}. Thus, these traces are called interval-based traces. Notice that traces from a single thread are ordered by timestamps since its operations are sent and executed in order in the DBMS.

2.3 Workload Dispatcher

To support online IL verification, the verification process should catch up with the DBMS's throughput. This imposes the requirement of verifying traces parallelly. Fortunately, as the IL verification usually deals with conflicting operations which access the same records, then we can borrow the idea of database sharding to divide traces into partitions based on the accessed data. That is, the traces from the test clients are firstly partitioned by the workload dispatcher, and then are dispatched to different verifier instances to facilitate parallel verifying.

2.4 Verifier

To guarantee ILs, the database community has proposed various concurrency control protocols (CCPs), which can be classified into lock-based CCP (e.g., 2PL) and timestamp-based CCP (e.g., MVCC, OCC and TO)[3]. Different CCPs take different actions on data and would produce different data evolution footprints. Generally, there are three critical data structures for implementing these CCPs, which are lock table, version chain, and dependency graph. The main idea of *verifier* is to deduce the data evolution (dependency) according to the implementations of a CCP in a DBMS. If a different evolution is deduced compared to that generated by the DBMS, then an anomaly is detected. Specifically, the *verifier* first organizes traces by its *trace manager* (in §2.4.1), then composes critical data structures by its *state evolver* (in §2.4.2), and finally deduces evolutions and detects anomalies in the *anomaly checker* (in §2.4.3).

2.4.1 Trace Manager. In each *verifier*, its *trace manager* takes the role of a global trace sorting of its assigned client-side interval-based local traces, which is crucial for subsequent dependency deduction on accessed data. Note that, traces from the same client are already naturally sorted in the order of ts_{bef}/ts_{aft}. It then makes the sorting as a merge of traces from different clients. For efficient sorting, it designs a batch-based *two-level pipeline* to keep delivering ordered traces for constructing dependencies (see Fig. 1). The first-level is that *trace manager* stores the ordered traces from the same thread into the same *local buffer*. The second-level is that *trace manager* collects traces in each *local buffer* into *global buffer*, which launches a min heap sort according to the ts_{bef} of traces. Specifically, in each round, *trace manager* fetches a batch of traces from *local buffers* in a unified manner into the *global buffer* and updates the min heap. Next, it obtains the minimum ts_{bef} across all *local buffers* as the *watermark*, and then any traces in the min heap (i.e., *global buffer*) with ts_{bef} lower than the *watermark* can be delivered to the *state evolver*. This is because traces in each local buffer are ordered. After delivering these traces, *trace manager* starts the next round of

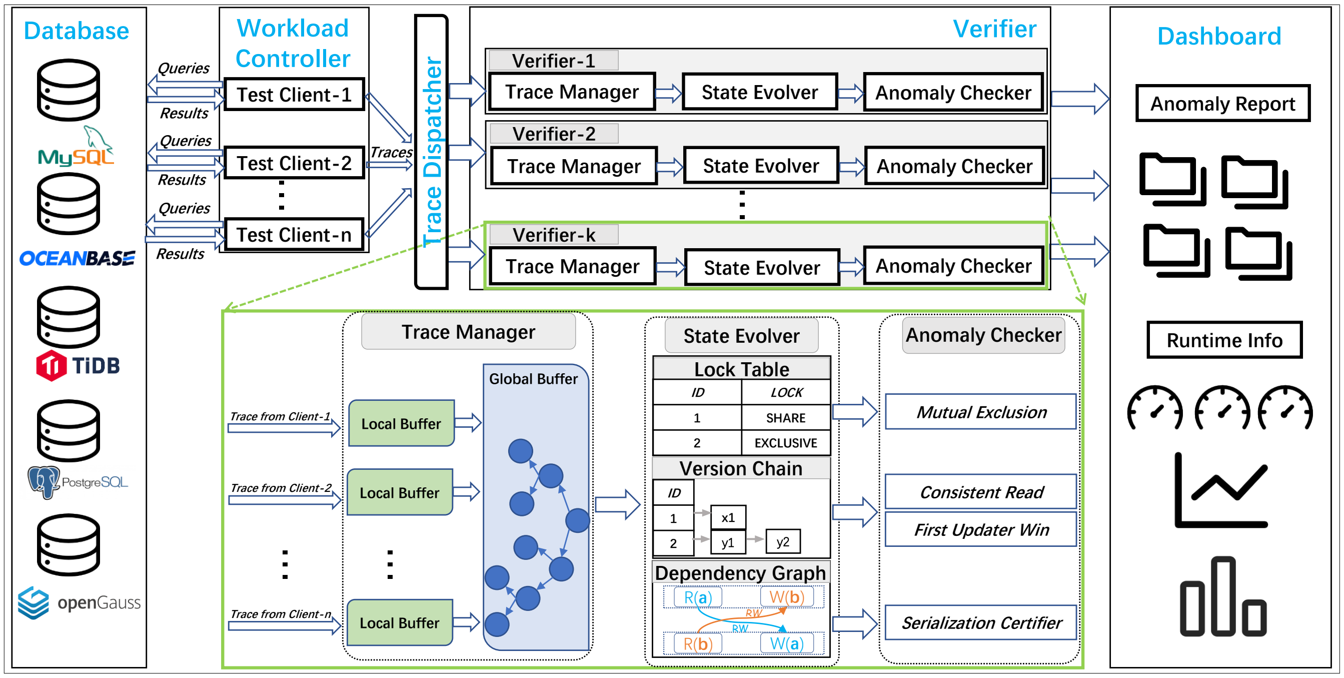


Figure 1: The Architecture of Leopard

trace fetching, sorting, watermark updating and trace delivering. To avoid memory explosion, usually the batch size from *local buffers* is equal to the size of delivered traces. Note, when the number of traces in *global buffer* exceeds a predefined *threshold*, only traces with ts_{bef}=*watermark* are fetched from *local buffers* in the next round. Since ts_{bef} in each *local buffer* is monotonically increasing, it ensures the *watermark* keeps increasing, and at least the newly fetched traces from *local buffers* can be delivered in the next round of delivering. This avoids unbounded storage expansion of the *global buffer*. The *two-level pipeline* can guarantee that *global buffer* keeps delivering traces in order of ts_{bef}, which helps construct the data structures of lock table, version chain and dependency graph for subsequent anomaly deducing.

2.4.2 State Evolver. The lock table serves the lock based CCPs by managing the lock acquiring and releasing. A given DBMS might take different locking strategies under different ILs. Timestamp-based CCPs rely on pre-commit validation to avoid anomalies by checking the timestamp-based partial order from the perspective of either data or transactions. Specifically, by maintaining multiple physical versions of each record (i.e., changes of the data over time) in the version chain, MVCC allows read (r) operations to access historical versions, and thus to avoid conflicting with concurrent write (w) operations to boost performance. In addition, transactions may form a specific dependency pattern based on how and in what order they access a record. For example, ww dependency represents that a transaction writes a data after another transaction's write. Various simple dependencies can form complex dependency patterns, such as cycles. Different ILs prohibit specific dependency patterns (i.e., anomalies). Dependency graph is then constructed to detect various dependency patterns.

The *state evolver* takes the ordered traces from *trace manager* to construct these three data structures (i.e., lock table, version chain and dependency graph) according to the specific IL implementation of a DBMS. For example, when verifying Read Committed of MySQL, an UPDATE will impose exclusive locks on the relevant rows in lock table, and the commit of the UPDATE will append new versions to the version chain of relevant data; while for verifying Serializable of PostgreSQL, if concurrent transaction T_1 reads a record before T_2 writes to the same record, a read-write (rw) dependency from T_1 to T_2 would be added in the dependency graph. Note that traces in a partition can only construct a subgraph of the dependency $w.r.t.$ data. To obtain the dependency graph for all transactions, inter-nodes communication is inevitable based on the accessed data in transactions. Nevertheless, the communication cost is low if *verifiers* are deployed in a single cluster, because the verification of each cross-node dependency only needs one network round trip.

Moreover, among these three data structures, lock table has a static size which can only be updated, while version chain and dependency graph grow along with transaction executions. To avoid memory explosion, it is crucial to perform garbage collection on these two structures. *Leopard* discards garbage versions periodically which will not be used for verification anymore. Specifically, if the end timestamp of a trace is earlier than the oldest start timestamp of active transactions, this version is not visible to any subsequent transactions and will be garbage collected.

2.4.3 Anomaly Checker. To achieve a general verification technique for various implementations of ILs, after carefully investigating 18 popular DBMSs, *Leopard* has summarized and abstracted four mechanisms, i.e., *Consistent Read* (CR), *Mutual Exclusion* (ME), *First Updater Wins* (FUW), and *Serialization Certifier* (SC), which can be orchestrated to realize almost all ILs in commercial DBMSs. Furthermore, to provide online IL anomaly detection ability, *Leopard* proposes a *mechanism-mirrored* verification method, which is to simulate the implementation of CCPs inside a DBMS by combining these four mechanisms in *verifier* instances. Since concurrency control is just one of the components of transaction processing, reproducing only a concurrency control mechanism from the client side should be faster than the complete transaction processing in a DBMS, which facilitates our online verification. The main ideas of anomaly detection with the client-side *interval-based traces* are summarized as followings.

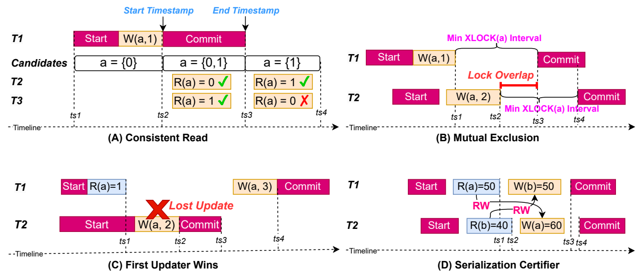


Figure 2: Verification Examples with the Length of Rectangle as the Client-side Time Interval

Consistent Read (CR) provides a consistent view of the database at a specific time. As a black-box testing suite, *Leopard* cannot obtain the exact execution time of an operation, which determines the version to read. We leverage the visible snapshot time interval of each operation, i.e., ts_{bef} and ts_{aft}, and potential version evolution of each record to guide the CR verification. Based on the trace's interval, we can distinguish the execution order between operations that do not overlap in time. Note, we have conducted experiments [17] to show that the non-overlapping rate is more than 90% even under skewed data with high concurrency. For overlapping operations, we identify the candidate version set for the active transaction *w.r.t.* the consistency requirement, which is either statement-level or transaction-level. Any read outside the candidate set is considered as a CR anomaly. For example in Fig. 2A, data item a has 3 candidate version sets along the timeline according to the commit of T_1's write. Before sending *commit* of T_1 (at ts_2), the read candidate of a can only be the initial value of $a = 0$. During the *commit* time interval of T_1, i.e., $[ts_2, ts_3]$, both 0 and 1 can be the read candidate, for we cannot know the exact time point when the write takes effect. So the reads in $[ts_2, ts_3]$ of T_2 and T_3 can read either 0 or 1. But after *commit* of T_1 (at ts_3), $a = 1$ is determined, so the read $R(a) = 0$ of T_3 after ts_3 is wrong.

Mutual Exclusion (ME) coordinates concurrent accesses to shared resources through locking, i.e., to ensure exclusive access to data. As the exact lock acquiring and releasing time points are not available in client-side, there might exist multiple possible orders of lock operations for the given conflict operation traces. We broadly classify the orders into two cases. Firstly, if each of the possible orders of lock operations is identified to be incompatible, then there must exist an ME violation inside the DBMS. Secondly, the valid order of locking should satisfy that each lock operation does not acquire a lock until the previous operation releases the lock. So, an ME violation happens when a transaction acquires an incompatible lock holding by another transaction. An example is shown in Fig. 2B. Two concurrent transactions T_1 and T_2 write to a. From the client side trace intervals of operations, we can claim that T_1's minimum exclusive lock interval on a is $[ts_1, ts_3]$, and T_2's minimum exclusive lock interval on a is $[ts_2, ts_4]$. The overlapping of the two exclusive lock intervals indicates an ME violation.

First Updater Wins (FUW) avoids lost update anomalies among concurrent transactions. The lost update anomaly occurs when transaction T_1 reads a data item and then T_2 updates the data item (possibly based on a previous read), then T_1 (based on its earlier read value) updates the data item and commits [8]. An example is shown in Fig. 2C. T_1 reads a=1, after which T_2 updates $a = 2$ and commits. However, T_1 updates its read a=1 to a=3 and commits, which causes the write of T_2 lost. If a transaction reads a data item and later writes it, *Leopard* will check if there exist concurrent committed write operations within the time interval from its read snapshot point to the commit of its write operation. If so, a lost update anomaly occurs. For example $[ts_2, ts_3]$ of T_2 lies in $[ts_1, ts_4]$ of T_1, thus an anomaly is detected.

Serialization Certifier (SC) guarantees that transactions executed inside a DBMS are conflict serializable. A general approach of verifying conflict serializability is to do cycle searching on its dependency graph. A cycle on the dependency graph indicates a violation of conflict serializability. However, cycle searching is costly. We propose to use a certifier-based approach. In this way, SC checks whether there exists a specific dependency pattern (i.e., certifier) on the dependency graph that should be prohibited by DBMS. For example, the SSI of PostgreSQL takes two consecutive *rw* dependencies as a certifier, which leads to a write skew anomaly, as shown in Fig. 2D. There are two concurrent transactions T_1 and T_2. T_1 reads a version of a before T_2 commits which also writes a, i.e., $ts_1 < ts_4$, and then writes b. Thus, there exists a *rw* dependency from T_1 to T_2. And T_2 reads a version of b before T_1 commits which also writes b, i.e., $ts_2 < ts_3$, then writes to a. Thus, there exists a *rw* dependency from T_2 to T_1. The two consecutive *rw* dependencies indicate a write skew anomaly.

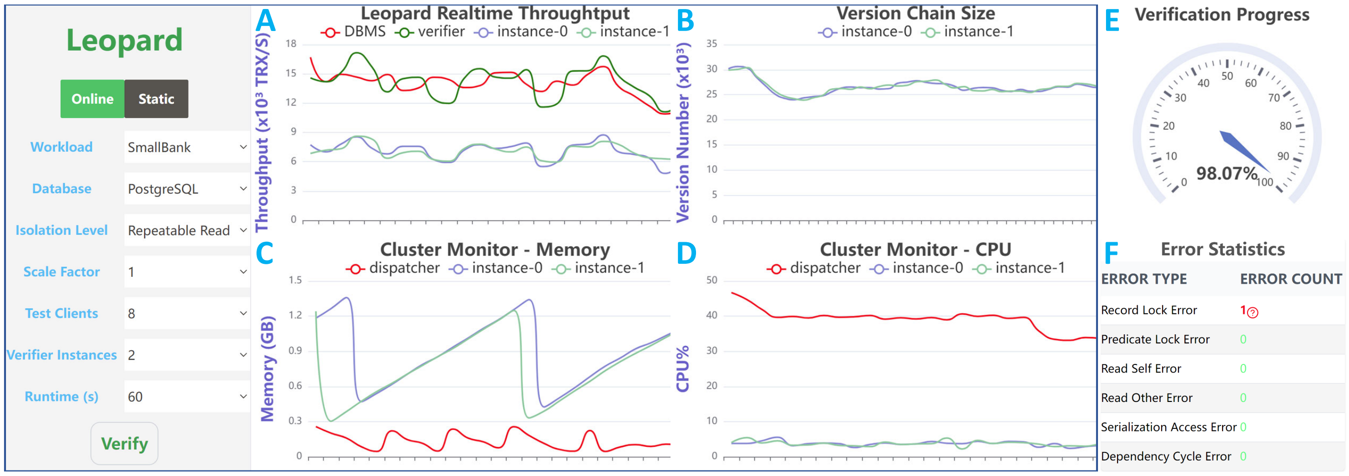
Note that we have sharded traces to different verifiers. Since CR, ME and FUW verifications only concern data conflicts, they can be served by each node individually. For SC, one network round trip inter-node communication is consumed to construct the dependency graph for active transactions.

2.5 Dashboard

The *dashboard* showcases the runtime actions of *Leopard*. It contains two parts, which are anomaly reports and runtime information. The report exposes the anomaly details and related traces, which help to locate and fix the problem. Runtime information contains the throughputs of online verifiers and the real-time status of inner data structures. It can be used to monitor the status of *Leopard* and help to set a proper configuration for online verification.

3 DEMONSTRATION PROPOSAL

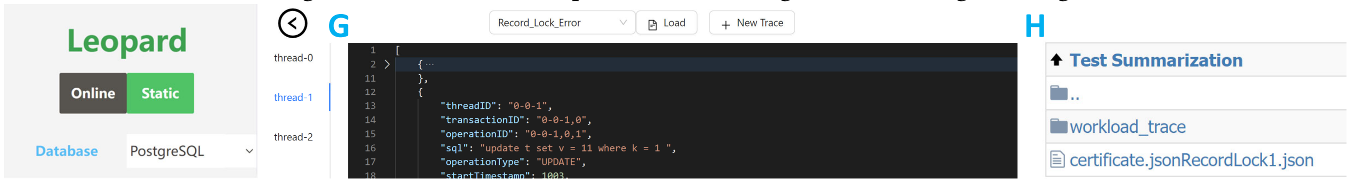
For demonstration, *Leopard* is deployed as a web service on a Ubuntu Server with a 16-core AMD EPYC Processor and 32G RAM, configured with one *trace dispatcher* and two *verifiers* (i.e., verifier instances) by default, which can be visited from the website



Workload	SmallBank
Database	PostgreSQL
Isolation Level	Repeatable Read
Scale Factor	1
Test Clients	8
Verifier Instances	2
Runtime (s)	60

ERROR TYPE	ERROR COUNT
Record Lock Error	1
Predicate Lock Error	0
Read Self Error	0
Read Other Error	0
Serialization Access Error	0
Dependency Cycle Error	0

Figure 3: Dashboard of Leopard for Controlling and Monitoring Running



```
[
    {…
    },
    {
        "threadID": "0-0-1",
        "transactionID": "0-0-1,0",
        "operationID": "0-0-1,0,1",
        "sql": "update t set v = 11 where k = 1 ",
        "operationType": "UPDATE",
        "startTimestamp": 1003,
```

Figure 4: Trace Editor and Bug Replaying Board

https://dbhammer.github.io/leopard/. In this section, we demonstrate usages and the performance of *Leopard* in IL verification. In scenario 1, we provide an online verification to show the performance of *Leopard* in Fig. 3. Though we have detected 24 bugs in several commercial DBMSs [13], it is not easy or even impossible to find any IL bug during running the example benchmark workloads in a short time. Then in scenario 2, to show the effectiveness of IL anomaly detection, we summarize and provide some example buggy workload traces from our previous practical experiences in Fig. 4. These workload traces can trigger the generation of anomaly reports. Notice that, we have documented all bugs detected by *Leopard* [13], among which, 16 bugs have been confirmed by developers through community forums or emails .

3.1 Scenario 1: Online Verification of ILs

Leopard supports verifying various ILs without modifying codes of DBMSs and imposes no restriction on workloads, i.e., workload and kernel independent. In Fig. 3, it provides various knobs to vary the workloads, databases, client sizes, IL and verifier instance sizes, such that the *generality*, *efficiency* and *scalability* of its verification ability can be well demonstrated.

Workload Type. We plug OLTP-Bench [6] into *Leopard*, which provides two popular OLTP benchmarks, i.e., SmallBank [14] and TPC-C. They are used to demonstrate that *Leopard* can launch IL verification for any arbitrary workload.

Database Type. Two representative databases, i.e., MySQL and PostgreSQL, are used to run the workload. Currently, *Leopard* has been well adapted to MySQL/PostgreSQL-compatible DBMSs. And it can also be adapted to other DBMSs if necessary.

Isolation Level. The classic ILs are supported by *Leopard*, e.g., Read Committed, Repeatable Read and Serializable. Note that DBMSs may have different implementations even for the same IL, and *Leopard* can distinguish these differences and perform verification. For example, for Repeatable Read, TiDB fetches the snapshot at the start of a transaction, but PostgreSQL fetches the snapshot at the first non-transaction-control statement.

Scale Factor, Test Clients, Verifier Instances and Runtime. Adjusting the scale factor and the number of test clients can control the degree of data contention and thus affect the overall throughput of a DBMS. For a high throughput DBMS, a single verifier instance may not be able to keep up with the database executions. For online verification, *Leopard* can be scaled horizontally by increasing verifier instances. Runtime is used to control the runtime of a round of the demo application scenario.

For *generality*, we can configure the workload, target DBMS and IL for various test scenarios with a specified running time. For *efficiency*, we can adjust the number of clients and the scale factor of DBMS to test the performance of *Leopard* under different DBMS throughputs. For *scalability*, we can change the number of verifier instances to show the scalability of *Leopard.*

After setting the running parameters on the dashboard of Fig. 3, users can click the *verify* button to run both the database and *Leopard.* As the verification framework runs, the back-end constantly monitors its working status and the front-end displays the runtime performance statistics and verification status on the dashboard in Fig. 3A–Fig. 3F.

Firstly, we plot the throughput of a DBMS and the throughput of *Leopard* by summarizing all the throughputs of *verifier* instances in real time (in Fig. 3A). Since the version chain and dependency graph in *state evolve* are supposed to expand as the running of workload, we show a real-time scale of these data structures to exhibit the effectiveness of garbage collection of *Leopard* (in Fig. 3B). At the same time, the interface monitors the runtime resource usages of *Leopard*, including memory (Fig. 3C) and CPU (Fig. 3D). Moreover, the verification progress is shown in Fig. 3E, which represents the running progress of the test scenario. After verification, it presents the detected anomalies in Fig. 3F. An analysis report is summarized and linked to bugs, which contains the traces and the details related to the anomalies as in Fig. 4H.

3.2 Scenario 2: Buggy Workload Trace Analysis

To demonstrate the IL anomaly detection ability, we abstract and summarize the patterns of IL bugs from our practical experiences, and construct several example buggy traces $w.r.t.$ different patterns in Fig. 4G, which can be loaded to *Leopard* to trigger bug reporting. Meanwhile, it allows users to edit and correct these example buggy traces, and then to test the response of *Leopard* to the modifications. Users can also prepare their own workload traces following our format guideline [4], and then load them to *Leopard* for verification. Finally, the bug summarization and the related details are presented in the test summarization (as in Fig. 4H) by clicking the question mark for detected anomalies (as in Fig. 3F.)

4 BUG CASE DEMONSTRATION

In this section, we show a snapshot versioning error recently found in XDB, which could not be detected by existing tools, e.g., Cobra and Elle.

```
1  /* Isolation Level: REPEATABEL READ */
2  CREATE TABLE t(k INT, v INT);
3  INSERT INTO t VALUES (1, 1);          /* T0 */
4
5  START TRANSACTION;                     /* Session -1 : T1 */
6  START TRANSACTION;                     /* Session -2 : T2 */
7  UPDATE t SET v = v + 1 WHERE k = 1;/* Session -1 : T1 */
8  /* Result: UPDATE 1 */
9  COMMIT;                                /* Session -1 : T1 */
10 SELECT * FROM t WHERE k = 1;           /* Session -2 : T2 */
11 /* Result (1 row) :    k | v
12                        1 | 1  */
13 COMMIT;                                /* Session -2 : T2 */
```

At the beginning, T_0 inserts a record $(k = 1, v = 1)$. Subsequently, transactions T_1 and T_2, belonging to two separate sessions (session 1 and 2), concurrently access this record. According to XDB's documentation, when the isolation level is REPEATABLE READ, it fetches the snapshot by the first non-transaction-control statement. Therefore, the SELECT statement at line 10 should read the latest committed version, which is $(k = 1, v = 2)$. However, XDB returns a stale version $(k = 1, v = 1)$. By leveraging timestamps, *Leopard* can infer that session 2 should read the updated data by session 1, which is inconsistent with the data in the read/write sets. Thus, *Leopard* can identify this as an error. However, Elle and Cobra rely only on read/write sets to infer dependencies among operations and ignore their time orders. In this case, T_0 and T_1 form a *ww* dependency, while T_0 and T_2 form a *wr* dependency. Since no cycles exist among T_0, T_1 and T_2, Elle and Cobra fail to detect this bug.

5 RELATED WORK

To mutate queries into new forms by changing operators, accessing new attributes/tables and varying expressions is a class of work for detecting query-related bugs. PQS [15] and TQS [20] belong to this class of work. Due to the complexity of computational logic, it is tough for a query engine to cover all cases, especially corner cases, for expressions, joins, aggregations, etc. These tools focus more on composing variant queries. For example, TQS is the most recent test framework which targets at detecting logic bugs in multi-table join processing. The bugs are detected by iterating variant joins on diverse attributes/tables. These tools care little about parallel processing/contention or cannot detect transactional bugs.

Transactional bug detection cares more about whether concurrent transactions comply with the IL claimed by a DBMS. The main challenge comes from the construction of test oracle $w.r.t.$ non-deterministic parallel database accessing (modifying). Elle [10] and Cobra [5] are the state-of-the-art IL verifiers. Specifically, Cobra can only expose violations of serializability. Elle is adaptive to more ILs, but it still cannot support all the popular ILs. Both of them require to specify workload patterns which can explicitly expose dependencies among transactions based on data. As a result, they cannot be adapted to verify any workload. Moreover, they perform IL verification based on cycle searching on dependency graph, but the complexity of cycle searching increases superlinearly with the number of transactions [11]. Due to the lack of an effective garbage collection strategy, Elle's dependency graph increases dramatically with the increase of concurrent transactions, leading to a sharp increase in memory. Thus, neither Cobra nor Elle can scale out for online verification. DT^2[21] proposes a novel differential testing method for deterministically scheduling concurrent workloads in a serial fashion to detect transaction implementation flaws as well as compatibility issues in MySQL-compatible databases. However, its serial execution mechanism prevents it from verifying diverse parallel OLTP workloads.

Leopard is then designed to fill the gap between the IL verification tools and the real-world verification requirements, which presents good properties of generality, efficiency and scalability.

6 CONCLUSION

Leopard is an isolation level (IL) test suite. It is the first work which can perform online IL verification for various DBMSs in a black-box mode. We introduce an online web service to demonstrate the usage and ability of *Leopard* in this paper. *Leopard* is a general IL verifier that can work well with any workload generator. In addition to the workloads in benchmarks, *Leopard* can be also applied to well-tested in-production databases (e.g., MySQL, PostgreSQL, TiDB, OceanBase, and OpenGauss) by connecting to various private and public test case libraries provided by different database vendors. It helps us find 24 bugs in verifying transactions (more details in [13]), among which 16 are confirmed.

ACKNOWLEDGMENTS

This work is partially supported by NSFC (62072179, 62202171, 62307014), Shanghai Pujiang Program (21PJ1403200) and Shanghai Sailing Program (22YF1411300), the Ministry of Industry and Information Technology Key Laboratory of Performance and Reliability Testing and Evaluation for Basic Software and Hardware, and HuaWei Technologies.

REFERENCES

[1] Fekete A, Goldrei SN, and Asenjo JP. 2009. Quantifying Isolation Anomalies. *Proc. VLDB Endow.* 2, 1 (2009), 467–478.
[2] Sinha A and Malik S. 2010. Runtime checking of serializability in software transactional memory. In *IPDPS 2010.* 1–12.
[3] Bhargava B. 1999. Concurrency Control in Database Systems. *IEEE Trans. Knowl. Data Eng.* 11, 1 (1999), 3–16.
[4] Leopard Interval based Trace Format Guideline. 2023. https://github.com/DBHammer/leopard-demo/blob/main/Interval-based_Trace_Format_Guideline.pdf
[5] Tan C, Zhao C, Mu S, et al. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI 2020.* 63–80.
[6] Difallah DE, Polvo A, Curino C, et al. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.
[7] Leopard Online Demo. 2023. https://dbhammer.github.io/leopard/
[8] Berenson H, Bernstein P, Gray J, et al. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD 1995.* 1–10.
[9] Gray J, Lorie RA, Putzolu GR, et al. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *IFIP 1976.* 365–394.
[10] Kingsbury K and Alvaro P. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280.
[11] Li K, Weng S, Liu P, et al. 2023. Leopard: A Black-Box Approach for Efficiently Verifying Various Isolation Levels. In *ICDE 2023.*
[12] Brutschy L, Dimitrov D, Müller P, et al. 2017. Serializability for eventual consistency: criterion, analysis, and applications. In *POPL 2017.* 458–472.
[13] Leopard Bug List. 2023. https://github.com/DBHammer/leopard-demo/blob/main/Bug-List.pdf
[14] Alomari M, Cahill M, Fekete A, et al. 2008. The cost of serializability on platforms that use snapshot isolation. In *ICDE 2008.* IEEE, 576–585.
[15] Rigger M and Su Z. 2020. Testing Database Engines via Pivoted Query Synthesis. In *OSDI 2020.* 667–682.
[16] Xu M, Bodík R, and Hill MD. 2005. A serializability violation detector for shared-memory server programs. *ACM Sigplan Notices* 40, 6 (2005), 1–14.
[17] Leopard Technical Report. 2023. https://github.com/DBHammer/leopard-demo/blob/main/Technical-Report.pdf
[18] Jorwekar S, D Fekete A, Ramamritham K, et al. 2007. Automating the Detection of Snapshot Isolation Anomalies. In *VLDB 2007.* 1263–1274.
[19] Niharika Singh and Ashutosh Kumar Singh. 2018. Data Privacy Protection Mechanisms in Cloud. *Data Sci. Eng.* 3, 1 (2018), 24–39.
[20] Tang X, Wu S, Zhang D, et al. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. *Proc. ACM Manag. Data* 1, 1 (2023), 55:1–55:26.
[21] Cui Z, Dou W, Dai Q, et al. 2022. Differentially Testing Database Transactions for Fun and Profit. In *ASE 2022.* 35:1–35:12.