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:

$$\mathcal{T}.ts_{bef} \leq \min_{\mathcal{T}_i \in global} \mathcal{T}_i.ts_{bef} \tag{4}$$

$$\mathcal{T}.ts_{bef} \leq W = \min_{0 \leq i \leq n_{local}-1} local_i[0].ts_{bef} \tag{5}$$

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

$$\begin{aligned} local_i[0].ts_{bef} &\leq \min_{\mathcal{T}_j \in local_i} \mathcal{T}_j.ts_{bef} \\ &\leq \min_{\mathcal{T}_j \in C_i} \mathcal{T}_j.ts_{bef}, 0 \leq i \leq n_{local}-1 \end{aligned} \tag{6}$$

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. 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. This 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.*

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. More details about Serialization Certifier

The concurrency control protocols (*CCP*) in popular DBMSs often take advantage of the *serialization certifier* (*SC*) mechanism to guarantee that the executed transactions are conflict serializability [65]. However, each concurrency control protocol has its specific "certifier". Table II lists the the certifiers used by popular DBMSs. Next, we discuss how each certifier is used.

The *serializable snapshot isolation* (*SSI*) protocol implements the *SC* mechanism based on *snapshot isolation*, and uses the detection of the *write skew* anomalies as its certifier. Specifically, the *write skew* anomaly 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.

The *timestamp ordering* (*TO*) protocol uses the transaction timestamp ordering as its certifier when implementing its *SC* mechanism. Specifically, the certifier checks whether a transaction with an older timestamp has a dependency on another transaction with a newer timestamp. If so, it would abort the old transaction.

The *optimistic concurrency control* (*OCC*) protocol specifies that a transaction in the DBMS is executed in three phases: *read*, *validation*, and *write*. In the *read* phase, the transaction performs read and write operations on records without blocking. When the transaction prepares to commit, the *validation* phase uses the conflicts checking as a certifier to guarantee the conflict serializability. If there exist conflicts,

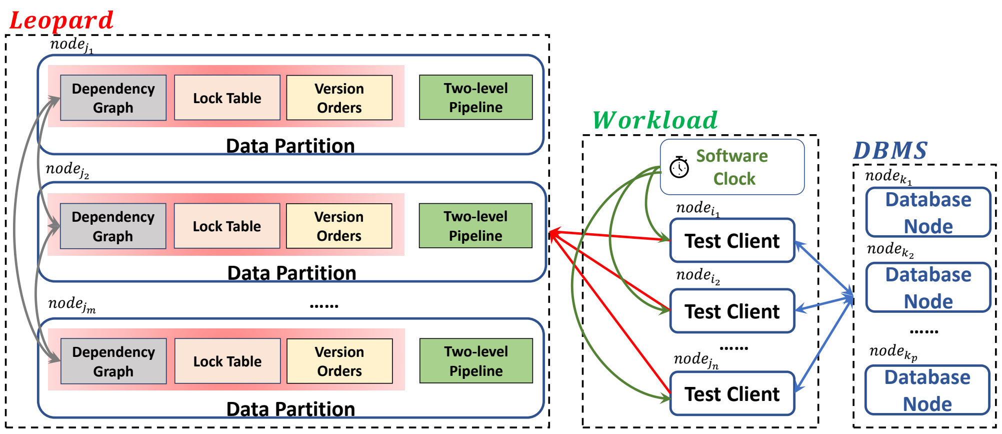

Fig. 1. Distributed Deployments of Leopard

TABLE II
SERIALIZATION CERTIFIER

CCP	Certifier
SSI	detecting write skew anomalies based on snapshot isolation
TO	ordering transactions as a monotonically increasing timestamp
OCC	checking the transaction conflicting with other active ones
Percolator	detecting lost update anomalies before committing a transaction

the certifier would abort the transaction; otherwise, it commits the transaction in *write* phase.

The *percolator* [74] protocol[1] also implements the idea of *serialization certifier* mechanism. Similar to the three concurrency control protocols mentioned above, *percolator* use its certifier to eliminate some specific anomalies when the transaction starts to commit. However, the certifier of *percolator* only guarantees snapshot isolation, and could not guarantee the conflict serializability. Specifically, *percolator* uses the check of concurrent versions as a certifier to eliminate *lost update* anomaly. That is, when a transaction starts to commit, the certifier checks whether a newer version has been installed since the transaction begins. If there exist concurrent versions, the certifier would abort the transaction; otherwise, it commits the transaction. We consider *percolator* as a special case of implementing the *serialization certifier* mechanism.

III. Multiple Machine Deployment

Fig. 1 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.

[1]Strictly speaking, *percolator* is a transaction processing framework, not only a concurrency control protocol.

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.

IV. 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.