Coo: Consistency Check for Transactional Databases

Haixiang Li, Yuxing Chen, Xiaoyan Li
lihaixiangDB@gmail.com;axinggu@gmail.com;li_xiaoyan@pku.edu.cn

ABSTRACT

In modern databases, transaction processing technology provides ACID (Atomicity, Consistency, Isolation, Durability) features. Consistency refers to the correctness of databases and is a crucial property for many applications, such as financial and banking services. However, there exist typical challenges for consistency. Theoretically, the current two definitions of consistency express quite different meanings, which are causal and sometimes controversial. Practically, it is notorious to check the consistency of databases, especially in terms of the verification cost.

This paper proposes Coo, a framework to check the consistency of databases. Specifically, Coo has the following advancements. First, Coo proposes partial order pair (POP) graph, which has a better expressiveness on transaction conflicts in a schedule by considering stateful information like Commit and Abort. By POP graph with no cycle, Coo defines consistency completely. Secondly, Coo can construct inconsistent test cases based on POP cycles. These test cases can be used to check the consistency of databases in accurate (all types of anomalies), user-friendly (SQL-based test), and cost-effective (one-time checking in a few minutes) ways.

We evaluate Coo with eleven databases, both centralized and distributed, under all supported isolation levels. The evaluation shows that databases did not completely follow the ANSI SQL standard (e.g., Oracle claimed to be serializable but appeared in some inconsistent cases), and have different implementation methods and behaviors for concurrent controls (e.g., PostgreSQL, MySQL, and SQL Server performed quite differently at Repeatable Read level). Coo aids to comprehend the gap between coarse levels, finding more detailed and complete inconsistent behaviors.

KEYWORDS

Database, ACID, Consistency, Isolation Levels, Data Anomalies

1 INTRODUCTION

Nowadays, real-world applications rely on databases for data storage, management, and computation. Transaction processing is one of the key components to guaranteeing the consistency of data. Especially, financial industries like securities companies, banks, and e-commercial companies often have zero tolerance for the inconsistency of any data anomalies in any form for their core transaction data. However, there exist typical challenges for consistency, and there is no direct and simple method to guarantee or check the consistency.

Motivation. Obtaining consistency for databases is vital yet it is known to be notorious and challenging from several perspectives. (i) **It lacks standards.** Theoretically, there exist two classical definitions with different meanings for consistency. These definitions are casual and consistency is guaranteed by either eliminating certain types of anomalies [35] or satisfying integrity constraints [34, 49]. The former divides consistency into four degrees, and each degree gradually forbids four standard anomalies. This is very mature in designing 2PL [31] protocol and standard isolation levels [34]. The latter checks consistency by verifying if the result satisfies integrity constraints. However, lacking the quantified standard of consistency may cause confusion or misuse of databases in production. For example, Oracle claimed the Serializable level supported in their databases by preventing all four standard anomalies yet proved to be only Snapshot Isolation level (more detailed anomalies shown in Table 4). Practically, it requires huge effort to design a good black-box testing tool for consistency checks. This is twofold. (ii) **It has a high knowledge bar.** The learning cost for users is high from setting up environments and modifying system modules, to understanding/modifying test cases and analyzing and debugging anomalies. Application scenarios are sometimes limited as some database services are closed-source or cloud-based where users often are not allowed to make changes or collect intermediate profiles. (iii) **It has a high verification cost.** Neither collecting nor checking is cost-effective [27, 40, 47, 55]. It is proved to be a NP-complete problem [24, 43] to verify a serializable commit order of all transactions via little known information of read-from dependency from input and output profiles (e.g., Cobra [48]). Some excellent works by random tests (e.g., Elle [16, 39]) can simulate some anomaly cases, but may waste a lot of time and computation on checking consistent transactions. Worse, the anomaly behaviors by these random tests sometimes can be hardly analyzed and reproduced.

These real-time [25, 37, 42, 46, 47, 53, 55] or post-verify [16, 24, 48] solutions are often costly and user-side burdened. This drives us to a root-cause question that can we discover, define, and generate all forms of data anomalies so that we can feed them all into databases and cost-effectively check once and for all. To address the question, we discuss current challenges of lacking of standards from two aspects, i.e., the formal definition of (1) data anomalies and (2) consistency.

Challenge 1. Define data anomalies. The ANSI SQL [34] specifies four isolation levels and four data anomalies. This standard is classical and has been widely used in real databases. However, the definition of data anomalies is casual and has been controversial from time to time [19]. The standard anomalies are single-object and avoided by lock-based protocols, yet more complex data anomalies, which are occasionally reported case by case as shown in Table 1, are hardly fit into defined levels. Existing literature [14, 18, 34] revised the definition to some extent. However, there is still little research to define and classify complete data anomalies, resulting in that the anomalies can still be ambiguous interpretations without a formal expression. For example, *Long Fork Anomaly* [28] and *Prefix Violation* [26] have the same expression yet reported by different instances. Many deadlocks (e.g., [41]), both local and global, are introduced and discussed, yet we think they are also a form of anomalies.

Challenge 2. Relate inconsistency to all data anomalies.

Table 1: A thorough survey on data anomalies in existing literature.

No	Anomaly, reference, year	Examples or expressions in original papers	Our expressions (Table 2)
1	Dirty Write [34] 1992	$W_1[x_1]...W_2[x_2]...((C_1$ or $A_1)$ and $(C_2$ or $A_2)$ in any order)	Dirty Write
2	Lost Update [18] 1995	$R_1[x_0]...W_2[x_1]...C_2...W_1[x_2]$	Lost Update Committed
3	Dirty Read [34] 1992	$W_1[x]...R_2[x]...(A_1$ and C_2 in either order)	Dirty Reads
4	Aborted Read [52] 2015, [14] 2000	$W_1[x:i]...R_2[x:i]...(A_1$ and C_2 in any order)	Dirty Reads
5	Fuzzy/Non-repeatable Read [34] 1992	$R_1[x]...W_2[x]...C_2...R_1[x]...C_1$	Non-repeatable Read Committed
6	Phantom [34] 1992	$R_1[P]...W_2[y \text{ in } P]...C_2...R_1[P]...C_1$	Phantom
7	Intermediate Read [52] 2015, [14] 2000	$W_1[x:i]...R_2[x:i]...W_1[x:j]...C_2$	Intermediate Read
8	Read Skew [18] 1995	$R_1[x_0]...W_2[x_1]...W_2[y_1]...C_2...R_1[y_1]$	Read Skew Committed
9	Unnamed Anomaly [45] 2000	$R_3[y]...R_1[x]...W_1[x]...R_1[y]...W_1[y]...C_1...R_2[x]...W_2[x]...R_2[z]...W_2[z]...C_2...R_3[z]...C_3$	Step IAT
10	Fractured Read [28] 2017, [17] 2014	$R_1[x_0]...W_2[x_1]...W_2[y_1]...C_2...R_1[y_1]$	Read Skew Committed
11	Serial-concurrent-phenomenon [23] 2014	$R_1[x_0]...W_2[x_1]...W_2[y_1]...C_2...R_1[y_1]$	Read Skew Committed
12	Cross-phenomenon [23] 2014	$R_1[x_0]...R_2[y_0]...W_3[x_1]...C_3...W_4[y_1]...C_4...R_2[x_1]...R_1[y_1]$	Step IAT
13	Long Fork Anomaly [28] 2017	$R_4[x_0]...W_1[x_1]...R_3[y_0]...R_3[x_1]...W_2[y_1]...R_4[y_1]$	Step RAT
14	Causality Violation Anomaly [28] 2017	$R_1[x_0]...W_2[x_1]...C_2...R_3[x_1]...W_3[y_1]...C_3...R_1[y_1]$	Step IAT
15	Read-only Transaction Anomaly [32, 50] 2004	$R_2[x_0,0]...R_2[y_0,0]...R_1[y_0,0]...W_1[y_1,20]...C_1...R_3[x_0,0]...R_3[y_1,20]...C_3...W_2[x_1,-11]...C_2$	Step IAT
16	Write Skew [18] 1995	$R_1[x_0]...R_2[y_0]...W_1[y_1]...W_2[x_1]$	Write Skew
17	Predicate-based Write Skew [33] 2005	$R_1[P]...R_2[P]...W_1[y_1 \text{ in } P]...W_2[x_1 \text{ in } P]$	Predicate-based Write Skew
18	Read Partial-committed [51] 2019	$R_1[x]...W_2[x]...W_2[y]...C_2...R_1[y]...C_1$	Read Skew Committed
19	Prefix violation [26] 2015	$R_1[x,1]...W_1[x,2]...R_2[y,1]...W_2[y,2]...R_3[x,2]...R_3[y,1]...R_4[y,2]...R_4[x,1]$	Step RAT

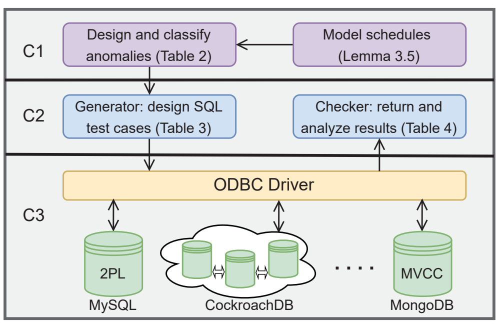

Figure 1: Coo framework. Contributions are C1: theoretical basis, C2: consistency check modules, and C3: evaluation and analysis of eleven databases.

There exist two previous works defining the consistency of databases. The first by Jim Gray et al. [35] defined several levels of consistency, which are strongly related to the ANSI SQL standard anomalies and 2PL protocol [36]. The second [34, 49] defined the consistency such that the final result is the same as one of the serializable schedules. However, both definitions hardly correlate with newly reported or undiscovered anomalies. For example, new anomalies like *Full Write Skew* (in Table 2) are hard to be quantified into previous definitions and their levels. Not to mention that with slightly different schedules (e.g., *Non-repeatable Read* and *Non-repeatable Read Committed*) may behave quite differently between databases (result in Table 4). Lacking complete mapping between data anomalies and inconsistency may lead to incomplete and sometimes non-reproducible consistency check (e.g., Elle [16, 39]).

Contribution (C). This paper proposes Coo, which contributes to pre-check the consistency of databases, filling the gap in contrast to real-time or post-verify solutions. Figure 1 shows the framework of Coo, which contributes to the following three aspects:

- **C1: Coo has theoretical basics.** We propose *Partial Order Pair (POP) Graph*, considering stateful information (i.e., commit, and abort), can model any schedule, compared to the traditional conflict graph which is limited to model transaction history. For example, we will show that *Read Skew* (without stateful information) and *Read Skew Committed* (with stateful information), which were treated as the same previously, are completely different anomalies (i.e., different formal expressions in Table 2 and different evaluation behaviors in Table 3). By POP cycles, we can define all data anomalies, correlating reported known (e.g., Dirty Read and Deadlocks) and unexposed mysterious anomalies.
- **C2: Coo is black-box and cost-effective.** The core consistency check modules are a generator and a checker, which are independent of databases. The generator provides SQL-like quires and schedules based on our definition of data anomalies, and the checker recognizes the consistent and inconsistent behaviors of the executed schedules. Each defined anomaly will be tested individually by issuing parallel transactions via ODBC driver to tested databases. The consistency check is accurate (all types of anomalies), user-friendly (SQL-based test), and cost-effective (one-time checking in a few minutes).
- **C3: Eleven databases are evaluated.** Through the evaluation, both centralized and distributed, we unravel the consistent and inconsistent behaviors under different isolation levels. And we are, as far as our knowledge goes, the first to propose methods for distributed evaluation. We can specifically show the occurrence of anomaly types in non-consistent databases or at their weaker isolation levels. Our evaluation found that some databases (e.g., Oracle, OceanBase, and Greenplum) claimed to be serializable but can not avoid some *IAT* anomalies (defined in Section 3.3). Also, we in-depth analyze the behaviors of different databases at different isolation levels with various implementation methods (e.g., different behaviors to designed anomaly cases by PostgreSQL, MySQL, and SQL Server in Repeatable Read level).

The rest of this paper is organized as follows. Section 2 presents the preliminary. Section 3 introduces our new model to define data anomalies and correlate inconsistency. Section 4 evaluates our model with real databases. Section 5 surveys the related work. Section 6 concludes the paper.

2 PRELIMINARY

This section provides the preliminary that will be used and extended in the following section.

Objects, Operations, Transactions. We consider storing data **objects** $Obj = \{x, y, ...\}$ in a database. Operations are divided into two groups, i.e., object-oriented operations and state-expressed operations. **Object-oriented operations** are operations on objects by reading or writing. Let Op_i describe the possible invocations set: reading or writing an object by transaction T_i. **State-expressed operations** are operations to express states of transactions, consisting of Commit (C) and Abort (A). **Transaction** is a group of operations, interacting objects, with or without a state-expressed operation at the end, representing a committed or an active state. We use subscripts to represent the transaction number. For example, $Op_i[x_n]$ is x-oriented operations by transaction T_i; C_i and A_j are the committed and abort operations by T_i and T_j, respectively.

Schedules. An Adya [15] history H comprises a set of transactions T on objects, an order E over operations Op in T. The E is persevered the order within a transaction and obeyed the object version order $<_s$. A **schedule** S is a prefix of H.

Example 2.1. We show an example of a schedule S_1 in the following:

$$S_1 = R_1[x_0]\ R_3[x_0]\ W_1[y_1]\ R_3[y_1]\ C_3\ W_2[x_1]\ R_1[y_1]\ A_1. \quad (1)$$

which involves three transactions, where $T_1 = R_1[x]W_1[y]R_1[y]A_1$, $T_2 = W_2[x]$, and $T_3 = R_3[x]R_3[y]C_3$ are aborted, active, and committed transactions respectively. The set of operations is $Op(S_1) = \{R_1[x], R_3[x], W_1[y], R_3[y], W_2[x], R_1[y]\}$. For operations on the same object, we have the version order, e.g., $R_1[x_0] <_s W_2[x_1]$. Note we don't have version order between two reads, e.g., ($R_1[x_0]$, $R_3[x_0]$) or between different objects, e.g, ($R_3[x_0]$, $W_1[y_1]$), meaning reversing these operations may be an *equivalent* schedule.

Conflict dependency and Conflict graph. Every history is associated with a conflict graph (also called directed serialization graph) [20, 54], where nodes are committed transactions and edges are the conflicts (read-write, write-write, or write-read) between transactions. The conflict graph is used to test if a schedule is serializable. Intuitively, an acyclic conflict graph indicates a serializable schedule, thus the consistent execution and final state. Figure 2(a) depicts the graphic representation of S_1.

3 CONSISTENCY MODEL

This section introduces a new consistency model called Coo that can correlate all data anomalies. Specifically, we first proposed *Partial Order Pair (POP) Graph*, which also considers state-expressed operations. We then show any schedule can be represented by a POP graph and our checker can check an anomaly via its POP cycle. Lastly, our generator constructs both centralized and distributed test cases based on POP cycles for the evaluation.

3.1 Partial Order Pair Graph

Adya's model introduced some non-cycle anomalies [15, 16] like Dirty Reads and Dirty Write. The reason is that they did not consider state-expressed operations in conflict graph, yet these operations sometimes may be equivalent to object-oriented ones [29]. We strive to map all anomalies via cycles by considering these state-expressed operations. We first formally define POPs as extended conflicts in the following.

Definition 3.1. **Partial Order Pair (POP).** Let T_i, T_j be transactions in a Schedule S and $T_i \neq T_j$. A Partial Order Pair (POP) is the combination of object-oriented and state-expressed operations from T_i and T_j and satisfies:

- both transactions operate on the same object;
- at least one operation affects the object version (a write or a rollback of a write).

Lemma 3.2. *There exist at most 9 POPs in an arbitrary schedule, i.e.,* $POP = \{WW, WR, RW, WCW, WCR, RCW, RA, WC, WA\}$.

Proof. The proof can be trivially achieved by enumerating all possible combinations of object-oriented and state-expressed operations. Let T_i, T_j be transactions in a Schedule S and $p_i \in T_i$ with $q_j \in T_j$ being object-oriented operations that access the same object, $(p_i, q_j) \in \{W_iW_j, W_iR_j, R_iW_j\}$. The following is a list of all possible combinations.

1. $p_i - q_j$: Both transactions T_i and T_j are still active.

The transaction T_i ends before T_j:

2. $p_i - C_i - q_j$: T_i commits before q_j;
3. $p_i - A_i - q_j$: T_i aborts before q_j;
4. $p_i - q_j - C_i$: T_i commits after q_j;
5. $p_i - q_j - A_i$: T_i aborts after q_j;

The transaction T_i ends after T_j:

6. $p_i - q_j - C_j$: T_j commits after p_i;
7. $p_i - q_j - A_j$: T_j aborts after p_i.

The operation p_i will not affect the operation q_j in combination 3 due to the timely rollback of T_i. So does combination 7. We obtain 15 cases by substituting $\{W_iW_j, R_iW_j, W_iR_j\}$ into (p_iq_j) of the remaining 5 combinations.

Among them, $W_iW_jC_j$ and W_iW_j both have the identical effect of modifying the accessing object by W_j, we group them together as POP WW. Similarly, we use POP WR to represent W_iR_j and $W_iR_jC_i$ and POP RW to represent R_iW_j and $R_iW_jC_j$. Because read operations are not affected by a commit or abort, we put $R_iW_jA_i$ and $R_iW_jC_i$ into RW. Similarly, we put $W_iR_jC_j$ into WR. Three cases with committed of T_i, i.e., $W_iC_iR_j[x]$, $W_iC_iW_j[x]$, and $R_iC_iW_j[x]$, are specified as types WCR, WCW, and RCW, respectively.

Finally, we have three special combination cases, i.e., $W_iR_jA_i$, $W_iW_jC_i$, and $W_iW_jA_i$, that are more complex as they have two version changing states. As for $W_iR_jA_i$, we have first changing state by W_iR_j then second changing state by R_jA_i. W_iR_j belongs to POP WR and $R_jA_i[x]$ belongs to new POP RA. Likewise, $W_iW_jC_i$ has WW and WC POPs, and $W_iW_jA_i$ has WW and WA POPs.

In summary, these 15 combination cases are grouped into 9 types POPs, i.e., $WW, WR, RW, WCW, WCR, RCW, RA, WC, WA$. □

Note that RA, WA, and WC are from the combination of a cycle, meaning RA, WA, and WC existed only when the cycle already existed, and this cycle is a 2-transaction cycle on a single object. Let $\mathcal{F} : POP(S) \to T(S) \times T(S)$ be the map between POPs and the transaction orders, e.g., $\mathcal{F}(W_iC_iR_j[x]) = (T_i, T_j)$. In terms of POPs and their orders, we can define POP graphs.

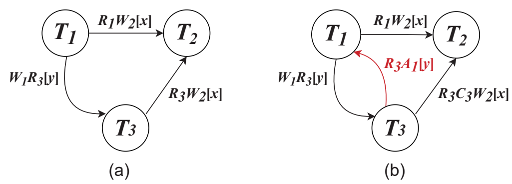

Figure 2: Comparison of (a) conflict and (b) POP graphs.

Definition 3.3. **Partial Order Pair Graph (POP graph).** Let S be a schedule. A graph $G(S) = (V, E)$ is called Partial Order Pair Graph (POP graph), if vertices are transactions in S and edges are orders in POPs derived from S, i.e (i) $V = T(S)$; (ii) $E = \mathcal{F}(POP(S))$.

Conflict and POP graphs differ in edges and expressiveness. Example 3.4 exemplifies the distinction between them.

Example 3.4. Continuing Example 2.1, we obtain objects $Obj = \{x, y\}$, and operations $Op[x] = \{R_1[x_0]R_3[x_0]C_3W_2[x_1]\}$ and $Op[y] = \{W_1[y_1]R_3[y_1]C_3R_1[y_1]A_1\}$ from S_1. Note that we don't put A_1 in $Op[x]$ as they don't have a write on object x by T_1. We derive POP from these operations, i.e. $\{R_1W_2[x], R_3C_3W_2[x], W_1R_3[y], R_3A_1[y]\}$. The Conflict graph and the POP graph for S_1 are shown in Figure 2. Note that edges from T_3 to T_2 are different in conflict (RW) and POP (RCW) graph. This time, by a POP graph, the Dirty Read is expressed by a cycle formed by T_1 and T_3.

Lemma 3.5. *Arbitrary schedules can be represented by POP graphs.*

Proof. Given an arbitrary schedule S with $Op(S)$ being the set of operations by transactions $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$. First, we can derive sets of operations for variables from S, $\{OP[x] | x \in Obj(S)\}$. Then we can find all the combination cases in each object operation set $Op[x]$. Finally, we classify them into POPs referred to the proof of Lemma 3.2. Through the above method, we can get the POP set $POP(S)$ corresponding to the schedule S. Then, by $\mathcal{F}$, we get the ordering between transactions based on POPs. We can model POP graphs using the transactions set and the dependent orders between transactions. □

3.2 Consistency and Consistency Check

With POP cycles, we now are ready to define data anomalies, then define consistency with no data anomaly.

Definition 3.6. **Data Anomaly**. The schedule exists a data anomaly exists if the represented POP graph has a cycle.

The definition of data anomalies by POP graphs differs from conflict graph one in three aspects. Firstly, POP graphs model schedules instead of histories (e.g., Full Write in Table 2). Secondly, POP graphs can express all anomalies with state-expressed (e.g., Dirty Read in Definition 3.4). Thirdly, POP graphs can model more distinct anomalies (e.g., Read Skew and Read Skew Committed in Table 2 are different but considered as the same by conflict graph). We now define the consistency of a schedule.

Definition 3.7. **Consistency** Schedule S satisfies consistency if the represented POP graph exists no cycle.

Checker. By definition 3.7, consistency, no data anomalies, and acyclic POP graphs are equivalent. Likewise, inconsistency, existing data anomalies, and existing POP cycles are equivalent. So a **consistency checker** is to test if a schedule exists a data anomaly, i.e., if the represented graph has a cycle. In theory, the consistency check is **sound**: if it reports an anomaly in a schedule, then that anomaly should exist in every history of that schedule. The consistency check is **complete**: if it reports an anomaly in a schedule, then a POP cycle exists in the schedule of that anomaly. As a schedule is a prefix of history, the anomaly occurring in the schedule also occurs in the corresponding histories. So the soundness is correct. As we defined that the anomaly schedule exists a POP cycle, the completeness is also correct.

3.3 Consistency Check in Practice

This part discusses the consistency check in practice. As each POP cycle may express an anomaly scenario, it is neither cost-effective nor possible to test infinite cycles. Our test cases involve trading off the cost and time spent against the completeness. We want as less as test cases to express as much as the database's inconsistent behaviors. By soundness, an anomaly may exist in different schedules or histories. We consider exploring the simplest form for a data anomaly, which will be used for the design and classification of data anomalies for the evaluation. As most known data anomalies (e.g., Dirty Write and Dirty Read) are single-object, we start with one object POP cycles.

Lemma 3.8. *A POP cycle with three transactions ($N_T = 3$) by one object ($N_{Obj} = 1$) exists a cycle with two transactions.*

Proof. We exclude POPs RA, WA, and WC in our discussion, as these POPs appeared in a two-transaction one-object cycle, which need no proof. We first assume the POP cycle is $G = \{\{T_1, T_2, T_3\}, \{(T_1, T_2), (T_2, T_3), (T_3, T_1)\}$. We let $\{(p_1, q_2), (p_2, q_3), (p_3, q_1)\}$ be the object-oriented operations in forming cycle $G = \{(T_1, T_2), (T_2, T_3), (T_3, T_1)\}$. We let $<_s$ denote the version order. So the graph can be represented by $\{p_1 <_s q_2; p_2 <_s q_3; p_3 <_s q_1\}$. As each POP should have a write operation, we have the following situations.

If $p_1 = W$, (i) if p_1 happens before p_2, i.e., $p_1 <_s p_2$, since $p_2 <_s q_3$, then $p_1 <_s q_3$, meaning a POP from T_1 to T_3. By original POP from T_3 to T_1, T_1 and T_3 forms a cycle. (ii) if p_1 happens later than p_2, i.e., $p_2 <_s p_1$, meaning a POP from T_2 to T_1, then, T_1 and T_2 forms a cycle.

If $p_1 = R$, then $q_2 = W$. Likewise, (i) if $q_2 <_s p_3$, then T_1 and T_2 forms a cycle. (ii) if $p_3 <_s q_2$, then T_2 and T_3 forms a cycle. □

Lemma 3.9. *A POP cycle with any number of transactions ($N_T \geq 3$) by one object ($N_{Obj} = 1$) exists a cycle with two transactions.*

Proof. The proof is by induction. The theorem holds for $N_T = 3$ by Lemma 3.8. We first assume theorem holds for $N_T < k$.

When $N_T = k$, we assume the POP cycle is $G = \{\{T_1, T_2, ..., T_k\}, \{(T_1, T_2), (T_2, T_3), ..., (T_k, T_1)\}$. We let $\{(p_1, q_2), (p_2, q_3), ..., (p_k, q_1)\}$ be the object-oriented operations in forming cycle $G = \{(T_1, T_2), (T_2, T_3), ..., (T_k, T_1)\}$. We let $<_s$ denote the version order between operations. So the graph can be represented by $\{p_1 <_s q_2; p_2 <_s q_3; ...; p_k <_s q_1\}$. As each POP should have a write operation, we have the following cases.

If $p_1 = W$, (i) if p_1 happens before p_{k-1}, i.e., $p_1 <_s p_{k-1}$, since $p_{k-1} <_s q_k$, then $p_1 <_s q_n$, meaning a POP from T_1 to T_n. By original POP from T_k to T_1, T_1 and T_k forms a cycle. (ii) if p_1 happens later than p_{k-1}, i.e., $p_{k-1} <_s p_1$, meaning a POP from T_{k-1} to T_1, then, we remove T_k and achieve a new cycle $G' = \{(T_1, T_2), (T_2, T_3), ..., (T_{k-1}, T_1)\}$. Based on the assumption, when $n = k - 1$ the theorem is true.

If $p_1 = R$, then $q_2 = W$. Likewise, (i) if $q_2 <_s p_{k-1}$, then T_1, T_2, and T_k forms a cycle. It can be reduced to 2-transaction cycle by lemma 3.8. (ii) if $p_{k-1} <_s q_2$, then, we remove T_1 and T_k, and achieve a new cycle $G' = \{(T_2, T_3), (T_3, T_4), ..., (T_{k-1}, T_2)\}$. Based on the assumption, when $n < k$ the theorem is true. □

In general, if one cycle only involves one object, we can find representative cycles of exactly two transactions. This property is meaningful, as when only one object involves, evaluating two-transaction cycles is sufficient to represent cycles with more transactions. Next, we consider a POP cycle with more than one object.

Lemma 3.10. *A POP cycle has more than two POPs accessing to one object exists a cycle with at most two connected POPs accessing this object.*

Proof. We first assume the POP cycle is $G = \{\{T_1, T_2, \ldots, T_n\}, \{(T_1, T_2), (T_2, T_3), \ldots, (T_n, T_1)\}$. The POP edges accessing the same object x are $\mathcal{F}(POP_i[x]) = (T_i, T_{i+1})$ and $\mathcal{F}(POP_j[x]) = (T_j, T_{j+1})$, $j > i$. We assume $\{(p_i q_{i+1}[x]), (p_j q_{j+1}[x])\}$ are the object-oriented operations in forming edges of $POP_i[x]$ and $POP_j[x]$. Then G can be simplified into the following graphs.

If $p_i = W$, (i) if $p_i <_s p_j$, since $p_j <_s q_{j+1}$, then $p_i <_s q_{j+1}$, meaning a POP from T_i to T_{j+1}. We get $G' = \{\{T_1, T_2, ...T_i, T_{j+1}...T_n\}, \{(T_1, T_2), \ldots, (T_i, T_{j+1}), \ldots, (T_n, T_1)\}$ with a new POP accessing x edge (T_i, T_{j+1}). (ii) if $p_j <_s p_i$, meaning a POP from T_j to T_i. We get $G' = \{\{T_i, T_{i+1}, ...T_j\}, \{(T_i, T_{i+1}), \ldots (T_{j-1}, T_j), (T_j, T_i\}$ with a new POP edge (T_j, T_i). The adjoining edges (T_j, T_i) and (T_i, T_{i+1}) with ordering $p_j <_s p_i <_s< q_{i+1}$ are both accessing the same object x. (ii-a) There will be no new POP edges between them until $p_j = q_j = R$, which is $\mathcal{F}^{-1}(T_j, T_i) \in \{R_j W_i, R_j C_j W_i\}$ and $\mathcal{F}^{-1}(T_i, T_{i+1}) \in \{W_i R_j, W_i C_i R_j\}$. (ii-b) Otherwise, meaning a POP from p_j and q_{i+1}, causing the POP cycle to continue to be simplified to $G' = \{\{T_i, T_{i+1}, ...T_j\}, \{(T_j, T_{i+1}), \ldots (T_{j-1}, T_j)\}$ with a new POP edge (T_j, T_{i+1}).

If $p_i = R$, then $q_{i+1} = W$. (i) If $q_{i+1} <_s p_j$, since $p_j <_s q_{j+1}$ then $q_{i+1} <_s q_{j+1}$, meaning a POP from T_{i+1} to T_{j+1}. We get $G' = \{\{T_1, T_2, ...T_i, T_{i+1}, T_{j+1}...T_n\}, \{(T_1, T_2), \ldots, (T_{i+1}, T_{j+1}), \ldots, (T_n, T_1)\}$ with a new POP edge (T_{i+1}, T_{j+1}). The adjoining edges (T_i, T_{i+1}) and (T_{i+1}, T_{j+1}) with ordering $p_i <_s q_{i+1} <_s< q_{j+1}$ are both accessing the same object x. (ii-a) If $q_{j+1} = W$, the graph G' can be continues to simplify by the POP $\mathcal{F}^{-1}(T_i, T_{j+1}) \in \{R_i W_{j+1}, R_i C_i W_{j+1}\}$. (ii-b) Otherwise, if $q_{i+1} = R$, POP edges are $\mathcal{F}^{-1}(T_i, T_{i+1}) \in \{R_i W_{i+1}, R_i C_i W_{i+1}\}$ and $\mathcal{F}^{-1}(T_{i+1}, T_{j+1}) \in \{W_{i+1} R_{j+1}, W_{i+1} C_{i+1} R_{j+1}\}$.

By repeating the above steps on object x, we can obtain the cycle with only one or two edges operating on this object. And if two edges remained, then these two edges are connected. □

Theorem 3.11. *A POP cycle has N_{Obj} ($N_{Obj} \geq 1$) objects exists a POP cycle with at most $2N_{Obj}$ transactions.*

Proof. When $N_{Obj} = 1$, Lemma 3.9 has proven the theorem.

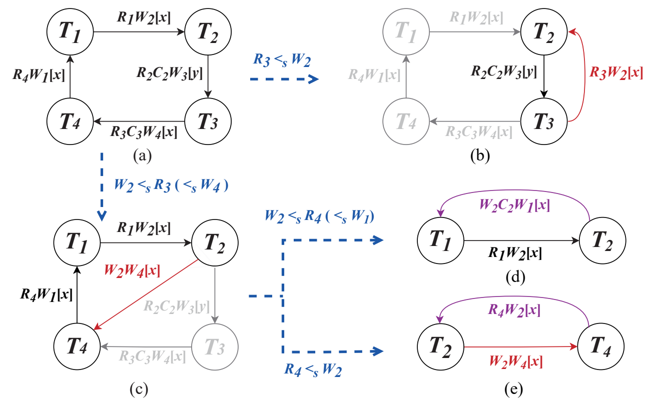

Figure 3: A 4-transaction cycle to its simplified cycles.

When $N_{Obj} \geq 2$, we prove it by contradiction. Without loss of generality, we assume that there exists a cycle with $2N_{Obj} + 1$ transactions and N_{Obj} objects that can not be simplified. The cycle must then include three POP edges accessing the same object, e.g x. However, by Lemma 3.10, we can proceed to simplify the cycle to at most two POPs accessing x, making the original cycle at most $2N_{Obj}$ transactions, which contradicts the assumption of the simplest cycle. □

Example 3.12. Figure 3(a) depicts a 4-transaction POP cycle $G = \{\{T_1, T_2, T_3, T_4\}, \{(T_1, T_2), (T_2, T_3), (T_3, T_4), (T_4, T_1)\}\}$ with $POPs = \{R_1W_2[x], R_2C_2W_3[y], R_3C_3W_4[x], R_4W_1[x]\}$. To simplify, (i) if $R_3 <_s W_2$, we obtained a new POP from T_3 to T_2, and a 2-transaction POP cycle $G' = \{(T_2, T_3), (T_3, T_2)\}$ as shown in Figure 3(b). (ii) if $W_2 <_s R_3$, then T_1, T_2, and T_4 forms a cycle as shown in Figure 3(c) by a new POP from T_2 to T_4. By lemma 3.8, we keep simplifying. (ii-a) if $W_2 <_s R_4$, since $R_4 <_s W_1$, then $W_2 <_s W_1$, meaning a POP from T_2 to T_1 (Figure 3(d)). (ii-b) if $R_4 <_s W_2$, then T_2 and T_4 forms a cycle (Figure 3(e)).

Generator. We provide two classifications. The first is based on primitive conflict dependencies, i.e., WR, WW, and RW, i.e., (i) **Read Anomaly Type (RAT)**, if the cycle has at least a WR POP; (ii) **Write Anomaly Type (WAT)**, if the cycle does not have a WR POP, but have at least a WW POP; (iii) **Intersect Anomaly Type (IAT)**, if the cycle does not have WR and WW POPs. This classification closely relates to three traditional conflicts and current knowledge, leading to a better evaluation and analysis of POP behaviors. So based on our classification, Read Skew ($R_1[x_0]W_2[x_1]W_2[y_1] R_1[y_1]$) and Read Skew Committed (We named it) ($R_1[x_0]W_2[x_1] W_2[y_1]C_2R_1[y_1]$) are different anomalies in different categories. Read Skew with WR belongs to RAT, while Read Skew Committed without WR and WW belongs IAT.

By Theorem 3.11, given finite number of transactions (N_T) and objects (N_{obj}), the simplified cycles are also finite and can be determinedly evaluated. This classification controls the real number of evaluation cases. The second is based on N_T and N_{obj} in cycles, i.e.,: (i) **Single Data Anomaly (SDA)**, if $N_T = 2$, $N_{obj} = 1$; (ii) **Double Data Anomaly (DDA)**, if $N_T = 2$, $N_{obj} = 2$; (iii) **Multi-transaction Data Anomaly (MDA)**, otherwise. So the SDAs and

Table 2: Data anomaly formal expression, classification, and their POP combinations in POP cycles.

Types of Anomalies		No	Anomalies	Formal expressions	POP Combinations
RAT	SDA	1	Dirty Read [14, 34, 52]	$W_i[x_m]\ldots R_j[x_m]\ldots A_i$	$W_iR_j[x]-R_jA_i[x]$
	SDA	2	Non-repeatable Read [34]	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots R_i[x_{m+1}]$	$R_iW_j[x]-W_jR_i[x]$
	SDA	3	Intermediate Read [14, 52]	$W_i[x_m]\ldots R_j[x_m]\ldots W_i[x_{m+1}]$	$W_iR_j[x]-R_jW_i[x]$
	SDA	4	**Intermediate Read Committed**	$W_i[x_m]\ldots R_j[x_m]\ldots C_j\ldots W_i[x_{m+1}]$	$W_iR_j[x]-R_jC_jW_i[x]$
	SDA	5	**Lost Self Update**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots R_i[x_{m+1}]$	$W_iW_j[x]-W_jR_i[x]$
	DDA	6	**Write-read Skew**	$W_i[x_m]\ldots R_j[x_m]\ldots W_j[y_n]\ldots R_i[y_n]$	$W_iR_j[x]-W_jR_i[y]$
	DDA	7	**Write-read Skew Committed**	$W_i[x_m]\ldots R_j[x_m]\ldots W_j[y_n]\ldots C_j\ldots R_i[y_n]$	$W_iR_j[x]-W_jC_jR_i[y]$
	DDA	8	**Double-write Skew 1**	$W_i[x_m]\ldots R_j[x_m]\ldots W_j[y_n]\ldots W_i[y_{n+1}]$	$W_iR_j[x]-W_jW_i[y]$
	DDA	9	**Double-write Skew 1 Committed**	$W_i[x_m]\ldots R_j[x_m]\ldots W_j[y_n]\ldots C_j\ldots W_i[y_{n+1}]$	$W_iR_j[x]-W_jC_jW_i[y]$
	DDA	10	**Double-write Skew 2**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots W_j[y_n]\ldots R_i[y_n]$	$W_iW_j[x]-W_jR_i[y]$
	DDA	11	Read Skew [18]	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots W_j[y_n]\ldots R_i[y_n]$	$R_iW_j[x]-W_jR_i[y]$
	DDA	12	**Read Skew 2**	$W_i[x_m]\ldots R_j[x_m]\ldots R_j[y_n]\ldots W_i[y_{n+1}]$	$W_iR_j[x]-R_jW_i[y]$
	DDA	13	**Read Skew 2 Committed**	$W_i[x_m]\ldots R_j[x_m]\ldots R_j[y_n]\ldots C_j\ldots W_i[y_{n+1}]$	$W_iR_j[x]-R_jC_jW_i[y]$
	MDA	14	**Step RAT** [26, 28]	$\ldots W_i[x_m]\ldots R_j[x_m]\ldots$, and $N_{obj}\geq 2$, $N_T\geq 3$	$\ldots W_iR_j[x]\ldots$
WAT	SDA	15	Dirty Write [34]	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots A_i/C_i$	$W_iW_j[x]-W_jA_i/C_i[x]$
	SDA	16	**Full Write**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots W_i[x_{m+2}]$	$W_iW_j[x]-W_jW_i[x]$
	SDA	17	**Full Write Committed**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots C_j\ldots W_i[x_{m+2}]$	$W_iW_j[x]-W_jC_jW_i[x]$
	SDA	18	Lost Update [18]	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots W_i[x_{m+2}]$	$R_iW_j[x]-W_jW_i[x]$
	SDA	19	**Lost Self Update Committed**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots C_j\ldots R_i[x_{m+1}]$	$W_iW_j[x]-W_jC_jR_i[x]$
	DDA	20	**Double-write Skew 2 Committed**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots W_j[y_n]\ldots C_j\ldots R_i[y_n]$	$W_iW_j[x]-W_jC_jR_i[y]$
	DDA	21	**Full-write Skew**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots W_j[y_n]\ldots W_i[y_{n+1}]$	$W_iW_j[x]-W_jW_i[y]$
	DDA	22	**Full-write Skew Committed**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots W_j[y_n]\ldots C_j\ldots W_i[y_{n+1}]$	$W_iW_j[x]-W_jC_jW_i[y]$
	DDA	23	**Read-write Skew 1**	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots W_j[y_n]\ldots W_i[y_{n+1}]$	$R_iW_j[x]-W_jW_i[y]$
	DDA	24	**Read-write Skew 2**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots R_j[y_n]\ldots W_i[y_{n+1}]$	$W_iW_j[x]-R_jW_i[y]$
	DDA	25	**Read-write Skew 2 Committed**	$W_i[x_m]\ldots W_j[x_{m+1}]\ldots R_j[y_n]\ldots C_j\ldots W_i[y_{n+1}]$	$W_iW_j[x]-R_jC_jW_i[y]$
	MDA	26	**Step WAT**	$\ldots W_i[x_m]\ldots W_j[x_{m+1}]\ldots$, and $N_{obj}\geq 2$, $N_T\geq 3$, and not include $(\ldots W_{i1}[y_n]\ldots R_{j1}[y_n]\ldots)$	$\ldots W_iW_j[x]\ldots$
IAT	SDA	27	Non-repeatable Read Committed [34]	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots C_j\ldots R_i[x_{m+1}]$	$R_iW_j[x]-W_jC_jR_i[x]$
	SDA	28	**Lost Update Committed**	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots C_j\ldots W_i[x_{m+2}]$	$R_iW_j[x]-W_jC_jW_i[x]$
	DDA	29	Read Skew Committed [18]	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots W_j[y_n]\ldots C_j\ldots R_i[y_n]$	$R_iW_j[x]-W_jC_jR_i[y]$
	DDA	30	**Read-write Skew 1 Committed**	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots W_j[y_n]\ldots C_j\ldots W_i[y_{n+1}]$	$R_iW_j[x]-W_jC_jW_i[y]$
	DDA	31	Write Skew [19]	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots R_j[y_n]\ldots W_i[y_{n+1}]$	$R_iW_j[x]-R_jW_i[y]$
	DDA	32	**Write Skew Committed**	$R_i[x_m]\ldots W_j[x_{m+1}]\ldots R_j[y_n]\ldots C_j\ldots W_i[y_{n+1}]$	$R_iW_j[x]-R_jC_jW_i[y]$
	MDA	33	**Step IAT** [23, 28, 32, 45, 50]	Not include $(\ldots W_{i1}[x_m]\ldots R_{j1}[x_m]\ldots$ and $\ldots W_{i2}[y_n]\ldots W_{j2}[y_{n+1}]\ldots)$, $N_{obj}\geq 2$, $N_T\geq 3$	$\ldots R_iW_j[x]\ldots$

DDAs are finite, which will be evaluated one by one, while MDAs are infinite, which will be evaluated by one of the typical cases. The four standard anomalies are SDAs. We think this classification is sufficient to illustrate the core idea and explore relatively complete inconsistent behaviors. But we do not limit classifications with more one-to-one mapping anomalies of fixed transactions and objects for a more detailed evaluation. We also plan our future work to test databases with more random cycles by a larger number of transactions and objects.

Table 2 shows all data anomalies types and their classification. The anomaly names with BOLD font are 20+ new types of anomalies that have never been reported (We named them with "committed" when it has a WCW, WCR, or RCW POP). Those reported in Step RAT and Step IAT are a tiny portion of them. Unlike previous tools (e.g., Elle [16]) which randomly issue queries and found anomaly by accident, our generator provides exact sequences of schedules (more details in Section 4.2), making the consistency check determined and explainable, meaning it is easy to reproduce and to debug/analyze the result.

Corollary 3.13. *If a schedule satisfies consistency, then the schedule does not have any data anomalies in Table 2.*

The current research mainly focused on centralized databases. There is little research on distributed consistency and it remains ambiguous to do a distributed check. We first define distributed data anomalies.

Definition 3.14. **Distributed Data Anomalies** The distributed data anomaly exists if the represented POP graph has a cycle, and it has at least two objects storing at distributed partitions.

The **distributed consistency check** is to test if a distributed data anomaly exists. The standard anomalies are not distributed ones and are insufficient for a distributed check as they are single-object. By our classification, we can construct a distributed data anomaly by a DDA or MDA. We particularly designed the test cases to access the different objects from different partitions sometimes from different tables. The design is required by table partitioning and the data is expected to insert/update in different partitions/shards (e.g., by PARTITION BY RANGE in SQL).

4 EVALUATION

In this part, we will evaluate 11 real databases with 33 designed anomaly test cases.

4.1 Setup

We deployed 2 Linux machines each with 8 cores (Intel(R) Xeon(R) Gold 6133 CPU @ 2.50GHz) and 16 GB memory. The centralized evaluation only used one machine. We tested distributed OceanBase, TDSQL, and CockroachDB by their cloud services. We installed UnixODBC for the common driver, and some database drivers are installed by the trial version connector from *CData* [1]. The tests are coded with C++. Each transaction is issued with one thread/core. The deadlock or wait_die timeout is often set to 20 seconds depending on the cases. The source code is available on Github [3]. We execute transactions in parallel while using *timesleep* (e.g., 0.1 second in centralized tests) between queries to force execution sequences.

We evaluated eleven real databases, i.e., MySQL [7], MyRocks [6], TDSQL [12], SQL Server [11], TiDB [13], Oracle [9], OceanBase [8], Greenplum [4], PostgreSQL [10], CockroachDB [2], MongoDB [5]. Most databases support four standard isolation level, i.e., Serializable (SER), Repeatable Read (RR), Read Committed (RC), and Read Uncommitted (RU). MongoDB supports only Snapshot Isolation(SI) level. Greenplum supports SER, RC and RU levels. OceanBase support two modes, i.e., MySQL (RR and RC supported) and Oracle modes (SER, RR, and RC supported). TiDB supports RR and RC levels, as well as its Optimistic (OPT) level. SQL Server also supports two additional SI levels in optimistic mode, i.e., the default one (SI) and the read-committed snapshot level (RCSI). Table 4 shows their default ("★") and other supported levels. Some levels in one database perform the same, so we put them together (e.g., RC and RU in PostgreSQL). We exclude to present MyRocks and TDSQL in most cases, as they perform the same as MySQL.

4.2 Construction of Test Cases

We constructed all 33 types of data anomalies described in Table 2. Note that the SDA and DDA are finite and one-to-one mapping anomalies, yet MDA denotes a set of anomalies. So we design one typical case for each of the MDAs. Step RAT, Step WAT, and Step IAT have designed schedules with three WR, WW, and RW POPs, respectively. For example, the schedule S_2 of the Read Skew anomaly can be executed in the following orders:

$$S_2 = R_1[x_0]\ \underline{W_2[x_1]\ W_2[y_1]}\ R_1[y_1] \tag{2}$$

However, $W_2[y_1]$ may be waited as $W_2[x_1]$ may be waited by conflicting to $R_1[x_0]$, making the other conflict disappear. So, we may let non-conflict operations start first to simulate the complete conflicts. For example, the schedule S_3 of Read Skew anomaly can be executed in the following orders:

$$S_3 = R_1[x_0]\ \underline{W_2[y_1]\ W_2[x_1]}\ R_1[y_1] \tag{3}$$

In the schedule S_3, note that $W_2[y_1]$ starts earlier than $W_2[x_1]$, as $W_2[y_1]$ does not conflict with $R_1[x_0]$. After the execution, we assure the occurrence of two conflicts by ($R_1[x_0]$, $W_2[x_1]$) and ($W_2[y_1]$, $R_1[y_1]$). S_2 and S_3 are actually equivalent schedule with the same version order $<_s$. We then give the schedule of S_4 of Read Skew Committed anomaly in the following:

$$S_4 = R_1[x_0]\ W_2[y_1]\ W_2[x_1]\ C_2\ R_1[y_1] \tag{4}$$

As this time, the traditional conflict graph treated these S_3 and S_4 no difference. However, we recognized different POPs in S_3 and S_4, and later our evaluation will illustrate different performances under different isolation levels. Tables 3(A) and 3(B) depict the detailed preparation and execution steps by SQL queries for S_3 and S_4 by PostgreSQL at the RC level. In all our tests, the *Begin* command is alone with the first operation while the *Commit* command could be any order after schedule if not mentioned. PostgreSQL passed Read Skew schedule but found an anomaly result by Read Skew Committed schedule. Previous works (e.g., Elle [16]) often only

Table 3: PostgreSQL Evaluation by Read Skew and Read Skew Committed at the RC level and by Lost Update Committed and Step WAT at the SER level.

Preparation	
1	DROP TABLE IF EXISTS t1
2	CREATE TABLE t1 (k INT PRIMARY KEY, v INT)
3	INSERT INTO t1 VALUES (0, 0)
4	INSERT INTO t1 VALUES (1, 0)

A	**Generator:** Read Skew ($R_1[x_0]W_2[y_1]W_2[x_1]R_1[y_1]$)				
Q	Session 1: T_1-SQL	Operations		Session 2: T_2-SQL	Result
1	Begin				-
2	SELECT * FROM t1 WHERE k=0	$R_1[x_0]$			(0,0)
3				Begin	-
4			$W_2[y_1]$	UPDATE t1 SET v=1 WHERE k=1	-
5			$W_2[x_1]$	UPDATE t1 SET v=1 WHERE k=0	-
6	SELECT * FROM t1 WHERE k=1	$\cancel{R_1[y_1]}$ $R_1[y_0]$			Snapshot (1,0)
7			C_2	Commit	-
8	Commit	C_1			-
Checker: Pass (**P**) with consistency					

B	**Generator:** Read Skew Committed ($R_1[x_0]W_2[y_1]W_2[x_1]C_2R_1[y_1]$)				
Q	Session 1: T_1-SQL	Operations		Session 2: T_2-SQL	Result
1	Begin				-
2	SELECT * FROM t1 WHERE k=0	$R_1[x_0]$			(0,0)
3				Begin	-
4			$W_2[y_1]$	UPDATE t1 SET v=1 WHERE k=1	-
5			$W_2[x_1]$	UPDATE t1 SET v=1 WHERE k=0	-
6			C_2	Commit	-
7	SELECT * FROM t1 WHERE k=1	$R_1[y_1]$			MVCC+RC (1,1)
8	Commit	C_1			-
Checker: Anomaly (**A**) detected					

C	**Generator:** Lost Update Committed ($R_1[x_0]W_2[x_1]W_1C_2[x_2]$)				
Q	Session 1: T_1-SQL	Operations		Session 2: T_2-SQL	Result
1	Begin				-
2	SELECT * FROM t1 WHERE k=0	$R_1[x_0]$			(0,0)
3				Begin	-
4			$W_2[x_1]$	UPDATE t1 SET v=1 WHERE k=0	-
5			C_2	Commit	-
6	UPDATE t1 SET v=1 WHERE k=0	$W_1[y_1]$			Abort by rules
Checker: Rollback (**R**) by rules (WCW)					

D	**Generator:** Step WAT $W_1[x_1]W_2[y_1]W_3[z_1]W_3[y_2]W_2[x_2]W_1[z_2]$			
Q	Session 1: T_1-SQL	Session 2: T_2-SQL	Session 3: T_3-SQL	Result
1	Begin			-
2	$W_1[x_1]$: UPDATE t1 SET v=1 WHERE k=0	Begin		-
2		$W_2[y_2]$: UPDATE t1 SET v=2 WHERE k=1	Begin	-
4		$W_2[x_2]$: UPDATE t1 SET v=2 WHERE k=0	$W_3[z_3]$: UPDATE t1 SET v=3 WHERE k=2	W_2 waited
5			$W_3[y_3]$: UPDATE t1 SET v=3 WHERE k=1	W_3 waited
6	$W_1[z_1]$: UPDATE t1 SET v=1 WHERE k=2			2PL Wait deadlock
Checker: Deadlock (**D**) detected				

detected anomaly cases, but in this paper, we also in-depth analyze the potential anomaly cases that are prevented by databases as shown in Table 3(A, C, and D).

For the construction of distributed databases, we let keys (e.g., k=0 and k=1 in the Read Skew) spread into different distributed partitions (e.g., by PARTITION BY RANGE). Greenplum, which by default has a write lock for a table/segment, needs to enable a global detector for supporting concurrent writes. Another way is to simulate the cases with multiple tables and each table having one row/key.

4.3 Consistency Check in Databases

This part provides a general summary of the evaluation results. Table 4 shows the overall evaluation result of 11 databases with different isolation levels by 33 test cases constructed via SQL queries (except for MongoDB). The evaluation is cost-effective and reproducible, as we do not rely on the time- and resource-consuming random workloads but specifically and determinedly generate representative inconsistent scenarios. The average time spent for each level to finish 33 tests is around 1 minute. The original and executed schedules are available for analysis and debugging. The result behaviors are classified into two types, i.e., anomaly (A) and consistency. For anomaly occurrence, data anomalies are not recognized by databases, resulting in data inconsistencies, meaning the executed schedule with no equivalent serializable execution (or a POP cycle). While for the consistent performance, databases either pass (P) the anomaly test cases with a serializable result (no POP cycle) cycle or rollback transactions due to rules (R), deadlock detection (D), or timeout (T) reached.

SER level: All tested databases can guarantee no anomalies except Oracle, OceanBase, and Greenplum. These three databases claimed SER levels yet performed equivalent to the SI level. As most knowledge, researchers discover Oracle's inconsistency at its SER level by the Write Skew anomaly [19]. However, we found that anomalies also happened when feeding test cases of Write-read Skew, Write-read Skew Committed, Write Skew Committed, Step RAT, and Step IAT. These are similar anomalies yet previous work is hard to quantify such cases. More importantly, by Coo, we can build an infinite number of various-object Step RAT and Step IAT to reproduce anomaly scenarios, which are non-trivial by traditional tests or by CC protocols.

Weaker isolation levels: Unlike the original isolation levels having a coarse sense of a few anomalies, we can recognize and analyze many more newly found anomalies between levels, and some anomalies are confused to fit into one specific level (e.g., Lost Update Committed aborted in PostgreSQL as shown in Table 3(C) and appeared in MySQL at the RR level). More POPs are allowed at weaker levels and some anomalies are expected to be appeared by combinations of these allowed POPs. Roughly speaking, anomalies of RAT types have most Pass cases. In contrast, anomalies of WAT types have the most different rollback cases while databases occur most anomalies with test cases of IAT types. We explain more details of POP behaviors and anomaly occurrences in the right following.

4.4 Detailed Evaluation of POP Graphs

This part explains more details of POP behaviors and data anomaly occurrences. Specifically, we discuss consistency or consistent behaviors via POP and POP cycles. Firstly, POPs are the unit of conflicts that are handled by CC protocols (e.g., MVCC [20] and 2PL [31]). CC protocols perform different rules to allow or forbid these POPs. Roughly speaking, MySQL/RocksDB and TiDB are mainly using 2PL, and support MVCC at RR and RC levels. SQL Server uses pure 2PL and supports MVCC at its SI level. Other databases support MVCC at all levels and use 2PL for write locks. Secondly, POP cycles are specific anomalies, and consistency is guaranteed if cycles are destroyed based on these POP behaviors.

4.4.1 *POP Behaviors*.

This part discusses the behaviors of POPs at different isolation levels. Table 5 shows a summary of behaviors of three primitive POPs, i.e., WR, WW, and RW, corresponding to our test cases in three types, i.e., RAT, WAT, IAT. Core CC protocols used in different databases are 2PL and MVCC. Some are using combined protocols.

The **WR** is waited by MySQL at the SER level or SQL Server at SER, RR, and RC levels, and is allowed in other cases. First, WR is indeed allowed in 2PL databases at the RU level, as they allow a read on an uncommitted write. Second, WR is allowed by MVCC (e.g., PostgreSQL at all levels and MySQL at RR and RC levels) by reading the old committed version, transforming it into RW. For example, MySQL executed the Intermediate Reads ($W_1[x_1]\ R_2[x_1]\ W_1[x_2]$) as expected at the RU level but into a non-anomaly ($W_1[x_1]\ R_2[x_0]\ W_1[x_2]$) at the RC level.

The **WW** is waited by most evaluated databases at any level, except MongoDB directly aborts it and TiDB, at OPT level, prewrites it in private. This is very different from the ANIS SQL standard that considers WW as a Dirty write and forbids it at any level. In practice, the WW is somehow waited (not immediate abort) by the 2PL Wait strategy. For example, MySQL and SQL Server passed Full Write anomalies ($W_1[x_1]W_2[x_2]W_1[x_3]$), as they executed it into a non-anomaly ($W_1[x_1]W_1[x_3]C_1W_2[x_2]$), transforming WW into WCW (will discuss later).

The **RW** is allowed by most evaluated databases at any level, except at the SER level, 2PL databases wait for it and CockroachDB aborts it. Note SQL Server by using 2PL at the RR level still waits for RW. For example when executing Write Skew anomaly ($R_1[x_0]$ $R_2[y_0]W_2[x_1]W_1[y_1]$) at the SER level, 2PL databases (e.g., SQL Server) waited for each other by two RWs, i.e., ($R_1[x_0]W_2[x_1]$ and $R_2[y_0]W_1[y_1]$), yielding deadlocks. PostgreSQL allowed each RW in Write Skew but aborted it when two consecutive RWs were formed by the SSI [44] at the SER level while passed it as an anomaly at other levels.

Unlike previous analyses that discussed only primitive conflicts, we, in this paper, explain more POPs. We exclude the discussion of RA, WA, and WC in most cases, as they (i) exist only in a 2-transaction single-object cycle and (ii) perform similar to RW, WW, and WW, respectively. With the Wait strategy, the second operation of primitive POPs waits for the first one to be committed, meaning WR, WW, and RW will turn into WCR, WCW, and RCW, respectively. We then discuss more detailed behaviors of WCR, WCW, and RCW.

Table 4: The consistency check results of 11 databases. Anomaly (A): Data anomalies are not recognized by the database, resulting in data inconsistencies. Consistency: The databases passed (P) the fed anomalies test case; or the database rollback a data anomaly by Rules (R), Deadlock Detection (D), or Timeout (T) to guarantee consistency.

The primary classification	The data anomaly instance test cases from Table 2	Test case	MongoDB 4.4.4	CockroachDB v19.2.2	PostgreSQL 12.4			Greenplum 6.20.0		OceanBase 2.2.50 (MySQL)/ OceanBase 3.1.2 (MySQL)		OceanBase 2.2.50 (Oracle)/ OceanBase 2.2.77 (Oracle)			Oracle 12.1.0/ Oracle 21.3.0		TiDB 4.0.5/ TiDB 5.4.0			SQL Server 15.0				MySQL 8.0.20/ MyRocks 8.0.26/ TDSQL V2.0.1			
Iso-level	★ for default levels		SI	**SER**	**SER**	RR	★RC/RU	**SER**	RC/RU	RR	RC	**SER**	RR	★RC	**SER**	★RC	RR	RC	OPT	**SER/RR**	SI	★RC/RCSI	RU	**SER**	★RR	RC	RU
RAT	Dirty Read	1	P	**P**	**P**	P	P	**P**	P	P	P	**P**	P	P	**P**	P	P	P	P	**P**	P	P	A	**P**	P	P	A
	Non-repeatable Read	2	P	**P**	**P**	P	P	**P**	P	P	P	**P**	P	P	**P**	P	P	P	P	**P**	P	A	A	**P**	P	P	A
	Intermediate Read	3	P	**P**	**P**	P	P	**P**	P	P	P	**P**	P	P	**P**	P	P	P	P	**P**	P	P	A	**P**	P	P	A
	Intermediate Read Committed	4	P	**P**	**P**	P	P	**P**	P	P	P	**P**	P	P	**P**	P	P	P	P	**P**	P	P	A	**P**	P	P	A
	Lost self update	5	P	**P**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**P**	R	P	P	**P**	P	P	P
	Write-read Skew	6	A	**P**	**R**	A	A	**A**	A	A	A	**A**	A	A	**A**	A	A	A	A	**D**	A	D	A	**D**	A	A	A
	Write-read Skew Committed	7	A	**P**	**R**	A	P	**A**	P	A	P	**A**	A	P	**A**	P	A	P	A	**D**	A	D	A	**D**	P	P	A
	Double-write Skew 1	8	R	**R**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**D**	R	D	A	**D**	P	P	A
	Double-write Skew 1 Committed	9	R	**R**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**D**	R	D	A	**D**	P	P	A
	Double-write Skew 2	10	R	**P**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**D**	R	D	A	**D**	P	P	A
	Read Skew	11	P	**P**	**P**	P	P	**P**	P	P	P	**P**	P	P	**P**	P	P	P	P	**D**	P	A	A	**D**	P	P	A
	Read Skew2	12	P	**P**	**P**	P	P	**P**	P	P	P	**P**	P	P	**P**	P	P	P	P	**D**	P	A	A	**D**	P	P	A
	Read Skew2 Committed	13	P	**P**	**P**	P	P	**P**	P	P	P	**P**	P	P	**P**	P	P	P	P	**D**	P	A	A	**D**	P	P	A
	Step RAT	14	A	**P**	**R**	A	A	**A**	A	A	A	**A**	A	A	**A**	A	A	A	A	**D**	A	D	A	**D**	A	A	A
WAT	Dirty Write	15	R	**P**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**P**	R	P	P	**P**	P	P	P
	Full Write	16	R	**P**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**P**	R	P	P	**P**	P	P	P
	Full Write Committed	17	R	**P**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**P**	R	P	P	**P**	P	P	P
	Lost Update	18	R	**R**	**R**	R	A	**R**	A	R	A	**R**	R	A	**R**	A	A	A	R	**P**	R	A	A	**D**	A	A	A
	Lost Self Update Committed	19	R	**P**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**P**	R	P	P	**P**	P	P	P
	Double-write Skew 2 Committed	20	R	**P**	**R**	R	P	**R**	P	R	P	**R**	R	P	**R**	P	P	P	R	**D**	R	D	A	**D**	P	P	A
	Full-write Skew	21	R	**D**	**D**	D	D	**D**	D	T	T	**T**	T	T	**D**	D	D	D	R	**D**	D	D	D	**D**	D	D	D
	Full-write Skew Committed	22	R	**D**	**D**	D	D	**D**	D	T	T	**T**	T	T	**D**	D	D	D	R	**D**	D	D	D	**D**	D	D	D
	Read-write Skew 1	23	R	**R**	**R**	R	A	**R**	A	R	A	**R**	R	A	**R**	A	A	A	R	**D**	R	A	A	**D**	A	A	A
	Read-write Skew 2	24	R	**R**	**R**	R	A	**R**	A	R	A	**R**	R	A	**R**	A	A	A	R	**D**	R	A	A	**D**	A	A	A
	Read-write Skew 2 Committed	25	R	**R**	**R**	R	A	**R**	A	R	A	**R**	R	A	**R**	A	A	A	R	**D**	R	A	A	**D**	A	A	A
	Step WAT	26	R	**D**	**D**	D	D	**D**	P	T	T	**T**	T	T	**D**	D	D	D	R	**D**	D	D	D	**D**	D	D	D
IAT	Non-repeatable Read Committed	27	P	**P**	**P**	P	A	**P**	A	P	A	**P**	P	A	**P**	A	P	A	P	**P**	P	A	A	**P**	P	A	A
	Lost Update Committed	28	R	**R**	**R**	R	A	**R**	A	R	A	**R**	R	A	**R**	A	A	A	R	**P**	R	A	A	**D**	A	A	A
	Read Skew Committed	29	P	**P**	**P**	P	A	**P**	A	P	A	**P**	P	A	**P**	A	P	A	P	**D**	P	A	A	**D**	P	A	A
	Read-write Skew 1 Committed	30	R	**R**	**R**	R	A	**R**	A	R	A	**R**	R	A	**R**	A	A	A	R	**D**	R	A	A	**D**	A	A	A
	Write Skew	31	A	**R**	**R**	A	A	**A**	A	A	A	**A**	A	A	**A**	A	A	A	A	**D**	A	A	A	**D**	A	A	A
	Write Skew Committed	32	A	**R**	**R**	A	A	**A**	A	A	A	**A**	A	A	**R**	A	A	A	A	**D**	A	A	A	**D**	A	A	A
	Step IAT	33	A	**R**	**R**	A	A	**A**	A	A	A	**A**	A	A	**R**	A	A	A	A	**D**	A	A	A	**D**	A	A	A

The **WCR** occurred when the committed write is read. There are two cases: After a transaction with the write operation is committed, other transactions can read the data. For example, the Read Skew Committed ($R_1[x_0]W_2[x_1]W_2[y_1]C_1R_1[y_1]$) was executed as expected by most databases (e.g., PostgreSQL, MySQL) at RC and RU levels. The WCR is formed by $W_2[y_1]C_1R_1[y_1]$ (compared to the Read Skew ($R_1[x_0]W_2[x_1]W_2[y_1]R_1[y_1]$), where WCR does not exist). However, at the SER or RR level, by snapshot enabling (e.g., PostgreSQL), requiring to read a snapshot version y_0, Read Skew Committed was executed into a non-anomaly ($R_1[x_0]W_2[x_1]W_2[y_1]C_1$ $R_1[y_0]$), transforming WCR into RW.

The **WCW** occurred when the write is allowed after the concurrent write is committed. The WCW is allowed in most cases but not allowed in Databases with only write locks (e.g., PostgreSQL and Oracle) at SER, SI, and RR levels. For example, the Dirty Write ($W_1[x_1]W_2[x_2]C_1$) was passed in MySQL, as it was executed into a non-anomaly ($W_1[x_1]C_1W_2[x_2]$), where only one WCW exists. However, PostgreSQL aborted at SER and RR levels due to WCW POP. Similar cases are full Write and Full Write Committed.

The **RCW** is very much the same behavior as RW and is allowed in all databases. For example, the Intermediate Read and Intermediate Read Committed having RW and RCW, respectively, performed quite the same at different levels. At SER, 2PL databases actually executed the Intermediate Read into the Intermediate Read Committed. Similar cases happened between Read Skew 2 and Read Skew 2 Committed, between Read-write Skew 2 and Read-write Skew 2 Committed. One exception is that Oracle handled RW and RCW differently, as it passed Write Skew (with RW) but abort Write Skew Committed (with RCW).

In summary, at the SER level, 2PL databases (e.g., MySQL, SQL Server) do not allow WW, WR, and RW by 2PL Wait. However, they allow WCW, WCR, and RCW (as shown from SDA cases in Table 4). Other databases (e.g., PostgreSQL and Oracle) do not allow WW at all levels and do not allow WCW at SER and RR levels,

Table 5: Databases behaviors when meeting WW, WR, and RW POPs. 2PL(wait)/2PL(abort) stands for the waiting/abort of POPs, and MV(trans) stands for the transformation from WR to RW by MVCC.

POPs	DBs	SER	RR	RC	RU
WR	MySQL/TDSQL	2PL(wait)	MV(trans)	MV(trans)	allow
	SQL Server	2PL(wait)	2PL(wait)	2PL(wait)	allow
	SQL Server (SI)	/	MV(trans)	MV(trans)	/
	TiDB	/	MV(trans)	MV(trans)	/
	TiDB (OPT)	/	/	MV(trans)	/
	Oracle	MV(trans)	/	MV(trans)	/
	OceanBase (Oracle)	MV(trans)	MV(trans)	MV(trans)	/
	OceanBase (MySQL)	/	MV(trans)	MV(trans)	/
	Greenplum	MV(trans)	/	MV(trans)	MV(trans)
	PostgreSQL	MV(trans)	MV(trans)	MV(trans)	MV(trans)
	CockroachDB	MV(trans)	/	/	/
	MongoDB	/	MV(trans)	/	/
WW	MySQL/TDSQL	2PL(wait)	2PL(wait)	2PL(wait)	2PL(wait)
	SQL Server	2PL(wait)	2PL(wait)	2PL(wait)	2PL(wait)
	SQL Server (SI)	/	2PL(wait)	2PL(wait)	/
	TiDB	/	2PL(wait)	2PL(wait)	/
	TiDB (OPT)	/	/	prewrite	/
	Oracle	2PL(wait)	/	2PL(wait)	/
	OceanBase (Oracle)	2PL(wait)	2PL(wait)	2PL(wait)	/
	OceanBase (MySQL)	/	2PL(wait)	2PL(wait)	/
	Greenplum	2PL(wait)	/	2PL(wait)	2PL(wait)
	PostgreSQL	2PL(wait)	2PL(wait)	2PL(wait)	2PL(wait)
	CockroachDB	2PL(wait)	/	/	/
	MongoDB	/	2PL(abort)	/	/
RW	MySQL/TDSQL	2PL(wait)	allow	allow	allow
	SQL Server	2PL(wait)	2PL(wait)	allow	allow
	SQL Server (SI)	/	allow	allow	/
	TiDB	/	allow	allow	/
	TiDB (OPT)	/	allow	allow	/
	Oracle	allow	/	allow	/
	OceanBase (Oracle)	allow	allow	allow	/
	OceanBase (MySQL)	/	allow	allow	/
	Greenplum	allow	/	allow	allow
	PostgreSQL	SSI(allow)	allow	allow	allow
	CockroachDB	abort	/	/	/
	MongoDB	/	allow	/	/

Table 6: Anomalies at different isolation levels.

No	POP combinations	Example anomalies	Anomaly types
1	RW	Write Skew, Step IAT	IAT
2	RW, RCW	Write Skew Committed	IAT
3	RW, WCW	Lost Update Committed	IAT
4	RW, WCR	Read Skew Committed	IAT
5	all but no WW	Read Skew, Write-read Skew	RAT, IAT

Databases	SER	RR	RC	RU
MySQL/TDSQL	None	1. 2. 3.	1. 2. 3. 4.	5.
SQL server	None	None	1. 2. 3. 4.	5.
SQL server (SI)	/	1. 2.	1. 2. 3. 4.	/
TiDB	/	1. 2. 3.	1. 2. 3. 4.	/
TiDB (OPT)	/	/	1. 2. 3.	/
Oracle	1. 2.	/	1. 2. 3. 4.	/
OceanBase (Oracle)	1. 2.	1. 2.	1. 2. 3. 4.	/
OceanBase (MySQL)	/	/	1. 2. 3. 4.	/
Greenplum	1. 2.	/	1. 2. 3. 4.	/
PostgreSQL	None	1. 2.	1. 2. 3. 4.	1. 2. 3. 4.
CockroachDB	None	/	/	/
MongoDB (SS)	/	1. 2.	/	/

yet allow all other POPs, while PostgreSQL (using SSI) did not allow two consecutive RWs. At weaker isolation levels, all databases still forbid WW but gradually allow more POPs like RW and WCR.

4.4.2 *Data Anomalies Occurrence*.

This part discusses occurrences of anomalies at different isolation levels. Table 6 shows a summary of expected anomaly groups at different levels. We show 5 groups of different types of anomalies by different POP combinations. For example, Group 1 is the anomaly of any number of RW combinations. The typical anomalies are Write Skew and Step IAT in IAT. We found most databases allowed Group (1,2) or Group (1,2,3) at the RR level and allowed Group 4 furthermore at the RC level. While at the RU level, it allows anomalies formed by all POPs except WW. We show a more detailed evaluation of anomaly occurrences from two perspectives, i.e., (i) expected performance that anomalies should appear and (ii) unexpected performance that anomalies should have been forbidden, in the following.

The **RAT** exists at least one WR POP. (i) Based on our previous analysis, WR is indeed allowed only at the RU level by 2PL databases (e.g., MySQL and SQL Server). At the RU level, most schedules are executed as expected and anomalies are not detected. In contrast, at non-RU levels, RATs are mostly passed, as most of them are turned WR into RW by MVCC or WCR by 2PL Wait. For example, MySQL executed Intermediate Read ($W_1[x_1]R_2[x_1]W_1[x_2]$) as expected at the RU level but executed it into non-anomalies ($W_1[x_1]W_1[x_2]C_1R_2[x_2]C_2$) (WR to WCR by 2PL Wait) and ($W_1[x_1]$ $R_2[x_0]W_1[x_2]C_1C_2$) (WR to RW by MVCC) at SER and RR/RC levels, respectively. Interestingly, SQL Server executed Intermediate Read into one non-anomaly ($W_1[x_1]W_1[x_2]C_1R_2[x_2]C_2$) (WR to WCR by 2PL Wait) at all non-RU levels. (ii) RATs are not expected in non-RU levels, but some anomalies are reported, as they are executed into IATs. For example, at the RR level, Most DB executed both Write-read Skew ($W_1[x_1]R_2[x_1]W_2[y_1]R_1[y_1]$) and Write-read Skew Committed ($W_1[x_1]R_2[x_1]W_2[y_1]C_2R_1[y_1]$) are often executed into Write Skew ($W_1[x_1]$ $R_2[x_0]W_2[y_1]$ $R_1[y_0]$), except SQL Server did not allow RW , ending up as a deadlock. However, MySQL executed Write-read Skew into a non-anomaly ($W_1[x_1]W_2[y_1]$ $R_2[x_0]C_2R_1[y_1]$), (due to the timing of taking snapshot, more details in 4.5).

The **WAT** exists at least one WW POP and without WR. (i) WW is not allowed in all databases and at any level. For example, anomalies with all WWs like Full-write Skew, Full-write Skew Committed, and Step WAT are aborted in most databases at all levels. These anomalies are often detected as deadlocks (more detailed analysis in Section 4.7). (ii) However, we see some cases are passed. For example, Dirty Write ($W_1[x_1]W_2[x_2]C_1$) and Full-write anomalies were executed into non-anomalies (e.g., $W_1[x_1]C_1W_2[x_2]$ by Dirty Write) in most cases, transforming WW into WCW. Similar cases are Lost Self Update Committed, Double-write Skew 2 Committed, Read-write Skew 1/2, and Read-write Skew 2 Committed. However, some databases (e.g., PostgreSQL and Oracle), which disallowed WCW, aborted these two cases at SER, SI, OPT, and RR levels, but can execute the Dirty Write (abort version) ($W_1[x_1]W_2[x_2]A_1$) into a non-anomaly ($W_1[x_1]A_1W_2[x_2]C_2$).

The **IAT** does not exist WR and WW. (i) Most databases tolerate IATs at the non-SER level. At the RR level, most databases occurred anomalies with RW or RCW combinations. The typical anomalies are Write Skew, Step IAT, etc. At RC and RU levels, they further occurred anomalies with WCW or WCR POPs. The typical anomalies are Lost Update Committed, Read Skew Committed, etc. (ii) Oracle, OceanBase, and Greenplum claimed to support SER level yet it behaves similar to RR or SI equivalent level. They eliminated four standard anomalies but ignored some anomalies in IAT. We further discuss the behaviors of OceanBase by Read Skew Committed, Read-write Skew 1 Committed, and Write Skew Committed anomalies, which are with RW-WCR, RW-WCW, and RW-RCW POP combinations, respectively. At the RC level, OceanBase executed these three anomaly schedules as expected, reporting anomalies. While at the SER/RR level, OceanBase behaved quite differently. OceanBase (1) passed Read Skew Committed due to snapshot reading, transforming WCR into RW, (2) aborted Read-write Skew 1 Committed due to WCW abort rule, and (3) reported an anomaly by Write Skew Committed as it executed as expected.

In summary, at the SER level, no anomalies occurred except for Oracle, OceanBase, and Greenplum. As most knowledge, researchers discover Oracle not to be consistent at its SER level by the Write Skew anomaly. We found that anomalies also happened by Write-read Skew (both committed version and non-committed version), Step RAT, and Step IAT, although they executed into Write Skew eventually. At the RR level, most databases occurred anomalies with RW and RCW combinations (e.g., Read Skew, Write Skew, and Step RAT), except for SQL Server with a similar strong policy as SER level. Surprisingly, SQL Server has the same behaviors between SER and RR levels by our tests. At the RC level, most databases occurred all anomalies happened at the RR level, and anomalies with RW and WCW/WCR combinations (e.g., Lost Update Committed and Read-write Skew 1 Committed). At the weakest RU level, all databases only avoided WW, resulting in all kinds of anomalies without WW (e.g., Read Skew and Read Skew2). Thus, most anomalies occur at the RU level. And among all types of anomalies, IAT are the trickiest one with RW and other POPs, having the most anomaly cases.

Lesson learned: (i) Databases aim at consistency by avoiding all or partial POP cycles, and have different behaviors on different POPs. (ii) Different CC protocols are differently implemented between databases and between isolation levels. (iii) Developers still lack complete understanding between SER level and eliminating four standard anomalies, and between coarse isolation levels. Our evaluation can capture more insights and subtle behaviors between POPs, CC, and coarse isolation levels.

4.5 MVCC and Consistency

MVCC technology has three elements: multi versions, snapshot and data visibility algorithm. Multi versions with Read committed rule allow the newest committed objects to be read at RC levels. It helps to transform WR into RW. Snapshot, however, makes every read of the transaction consistent with exactly one committed version at SER and RR levels. It transforms WR and WCR into RW. For example, PostgreSQL and MySQL passed Non-repeatable Read Committed ($R_1[x_0]W_2[X_1]C_2R_1[x_1]$) at RR levels as it executed into a non-anomaly ($R_1[x_0]W_2[X_1]C_2\ R_1[x_0]$) but reported an anomaly at the RC level as expected. A similar case is Read Skew Committed.

MVCC sometimes are differently implemented. CockroachDB also consider Timestamp Ordering (TO) [20] in its CC protocols. Unlike traditional MVCC databases, reads are not waited/blocked, some read is waited in CockroachDB if an early uncommitted write found. For example, Write-read Skew Committed ($W_1[x_1]W_2[y_1]$ $R_2[x_1]C_2R_1[y_1]$) was executed into Write Skew by traditional MVCC databases like PostgreSQL, but into a non-anomaly ($W_1[x_1]W_2[y_1]$ $R_1[y_0]C_1R_2[x_1]C_2$) by the CockroachDB. Note that T_1 started earlier can read y_0, but T_2 started latter can not read x_0 but can read x_1 once T_1 committed.

Snapshot is the MVCC restricted to reading only one consistent version. Most databases (e.g., PostgreSQL, Oracle, and OceanBase 2.2.50) take the snapshot at the timestamp of first operation while some (i.e., MySQL and OceanBase 2.2.77) take the snapshot at the the first read. For example at the RR level, PostgreSQL executed the Write-read Skew Committed ($W_1[x_1]W_2[y_1]R_2[x_1]C_2R_1[y_1]$) into the Write Skew ($W_1[x_1]W_2[y_1]R_2[x_0]C_2R_1[y_0]$), printing anomaly found, as it takes the snapshot of x_0 and y_0. However, MySQL executed Write-read Skew Committed into a non-anomaly ($W_1[x_1]$ $W_2[y_1]R_2[x_0]C_2R_1[y_1]$), as it takes the snapshot of x_0 and y_1 at the first read.

Lesson learned: MVCC helps to transform WR into RW, and snapshot transforms WR and WCR into RW. Most databases (e.g., PostgreSQL) take snapshots at the beginning of the transaction while some (e.g., MySQL) at the first read.

4.6 Distributed Consistency

The above analyses are based on the centralized evaluation. This part discusses the evaluation of distributed databases. We deployed the data to be stored in different partitions/nodes. In Greenplum, as a write by default has a lock on one table/segments, we can distribute each row/keys to be in different tables. Note that SDAs (e.g., four standard anomalies) with one object are not suitable for the distributed consistency check. We want to observe the difference from the global CC protocols and deadlock detection. We evaluated 5 databases (i.e., MongoDB, CockroachDB, Greenplum, OceanBase, and TiDB) by DDAs and MDAs. We obtained the same results as Table 4, meaning these databases did a great implementation to maintain the consistent performance between centralized and distributed deployments.

We showcase the Write Skew ($R_1[x_0]R_2[y_0]W_2[x_1]W_1[y_1]$) anomaly occurred in distributed scenario by Greenplum. We let objects x and y be stored in two tables on two partitions. And then the Write Skew was executed as scheduled at the SER level, meaning an anomaly is found. Similar cases are Write Skew committed, Write-read Skew, etc. OceanBase at the SER level, MongoDB at the SI level, and TiDB at the RR level, existed similar anomalies in distributed scenarios.

Lesson learned: DDA and WDA type anomalies are suitable for distributed environments. CockroachDB has excellent consistent behaviors at the SER level as the centralized scenarios. But OceanBase and GreenPlum are not.

4.7 Deadlocks

Deadlocks occurred when multiple transactions wait for each other for resources. Deadlocks are usually found by periodically checking wait-for graphs [41]. Most databases (e.g., PostgreSQL, CockroachDB, and Oracle) use deadlock detection only for a small portion of data anomalies, and they detect deadlocks from anomalies Full-write Skew, Full-write Skew Committed, and Step WAT by two or three WW POPs waiting, as they have only locks on writes. In contrast, 2PL databases (e.g., SQL Server and MySQL) heavily detect deadlocks from all rollbacked cases, as they may have locks on both reads and writes, making WR, WW, and RW POPs wait for each other. Table 3(D) depicts an example of Step WAT rollbacked by PostgreSQL deadlock detection. The transaction which found deadlock is often aborted and the rest may continue to proceed. However, in PostgreSQL and Oracle, the transaction which found deadlock aborted while the rest are still waiting. They by default can not proceed and depend on *lock_timeout* to terminate. And OceanBase did not use any deadlock techniques at all, instead, it used timeout (e.g., 2PL Wait_die) to avoid deadlocks.

Lesson learned: (i) Deadlocks are caused by resource wait-for dependency by the 2PL Wait. (ii) Deadlocks are essentially special instances of data anomalies.

5 RELATED WORK

In this part, we surveyed the related work in more detail.

Consistency For database transactions, there are two classical definitions of C in ACID. First, the ANSI SQL [34] holds that consistency is met without violating integrity constraints; Second, Jim [35] believes that consistency can be divided into four levels, and each level excludes some data anomalies. Both of them are casual definitions and cannot directly and specifically guide the consistency verification of the database. Some [38, 39] reported that many databases do not provide the consistency and isolation guarantees they claimed. In fact, within the scope of the database, there is little research on the definition of consistency, not to mention the research on the relationship between consistency and data anomalies. Adya et al. [15] defines the relationship between conflict graphs and data anomalies. However, they can not correspond to some kinds of data anomalies (e.g., Dirty Read, Dirty Write and Intermediate Read [19, 34]). The reason behind this is that the stateful information like commit and abort cannot be modeled in the conflict graph. In contrast, this paper proposed a POP graph that can fully express the schedule with this stateful information. By POP graph, we are able to define all data anomalies and corresponding consistency to no anomalies.

Consistency check There exist two typical methods for checking databases consistency. One is by the white box method [25, 37, 42, 46, 47, 53, 55], where users often profile active transactions and conflicts to check non-serializable schedule. The white box method has a high knowledge bar and user-side burden to modify system code. As active transactions increases, the checking cost may exponentially increase, possibly affecting the performance of original transaction processing. The other is by the black box method [24, 48], where users do not make any modification for the system and check the result by some given workloads. Jepsen (including Elle [16], which is part of the Jepsen project) consistency check [39] is one of the popular tools in the industry. However, these methods usually issue random workloads to discover inconsistent behaviors. Such methods are not accurate, spending tons of computing resources. In contrast, Coo judiciously designed finite anomaly schedules, evaluating the consistency of databases once and for all. The evaluation is accurate (all types of anomalies), user-friendly (SQL-based test), and cost-effective (few minutes). The test is also possible for distributed databases. Test cases (i.e., DDA and MDA, which have more than one object) can be designed to force data to spread in different partitions.

Data anomalies, serializability, and consistency In recent years, there still exist extensive research works that focus on reporting new data anomalies, we make a thorough survey on data anomalies and show them in Table 1. These new data anomalies are constantly reported in different scenarios, indicating that data consistency in various scenarios is still full of challenges. The traditional knowledge has a shallow and inaccurate understanding between data anomalies and consistency. Previous work related conflict acyclic graph to consistency. They guarantee the serializable schedule to guarantee the consistency [21, 22, 30, 54]. The serialization is usually achieved by strong rules via eliminating three kinds of conflict relations (i.e., WW, WR, and RW) [30]. However, they can not quantify all data anomalies such as Dirty Read and Dirty Write. In this paper, Coo, by using the POP graph, can define all anomalies and correlate data anomalies to inconsistency.

6 CONCLUSION AND FUTURE WORK

This paper proposed Coo, which contributed to pre-check the consistency of databases, filling the gap in contrast to real-time or post-verify solutions. We systematically defined all data anomalies and correlated data anomalies to inconsistency. Specifically, we introduced an extended conflict graph model called Partial Order Pair (POP) Graph, which also considers state-expressed operations. By POP cycles, we can produce infinite distinct data anomalies. We classify data anomalies and report 20+ new types of them. We evaluated the new consistency model by ten real databases. The consistency check by predefined representative anomaly cases is accurate (all types of anomalies), user-friendly (SQL-based test), and cost-effective (one-time checking in a few minutes).

The research of predicate cases have not been discussed in this paper due to the limited space and is still on going. We think the model of this paper is compatible to extend to predicate cases (e.g., Phantom can be constructed by Non-repeatable Read, with predicate Select and replacing Update by Insert [3]).

REFERENCES

[1] 2022. CData. https://www.cdata.com.
[2] 2022. CockroachDB. https://www.cockroachlabs.com.
[3] 2022. Coo: Github open source code. https://github.com/Tencent/3TS/tree/coo-consistency-check.
[4] 2022. GreenPlum. https://greenplum.org.
[5] 2022. MongoDB. https://www.mongodb.com.
[6] 2022. MyRocks. https://myrocks.io.
[7] 2022. MySQL. https://www.mysql.com.
[8] 2022. OceanBase. https://www.oceanbase.com.
[9] 2022. Oracle. https://www.oracle.com.
[10] 2022. PostgreSQL. https://www.postgresql.org.
[11] 2022. SQL Server. https://www.microsoft.com/en-us/sql-server.
[12] 2022. TDSQL. https://cloud.tencent.com/document/product/557.
[13] 2022. TiDB. https://github.com/pingcap/tidb.
[14] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*. 67–78.
[15] Atul Adya and Barbara H. Liskov. 1999. Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. (1999).
[16] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280. https://doi.org/10.5555/3430915.3442427
[17] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3, Article 15 (July 2016), 45 pages. https://doi.org/10.1145/2909870

[18] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

[19] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD Conference*. ACM Press, 1–10.

[20] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (1983), 465–483.

[21] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[22] P. A. Bernstein, D. W. Shipman, and W. S. Wong. 1979. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering* SE-5, 3 (1979), 203–216.

[23] Carsten Binnig, Stefan Hildenbrand, Franz Farber, Donald Kossmann, Juchang Lee, and Norman May. 2014. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. 23, 6 (2014), 987–1011.

[24] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28.

[25] Lucas Brutschy, Dimitar K. Dimitrov, Peter Müller, and Martin T. Vechev. 2017. Serializability for eventual consistency: criterion, analysis, and applications. In *POPL*. ACM, 458–472.

[26] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 568–590. https://doi.org/10.4230/LIPIcs.ECOOP.2015.568

[27] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *CONCUR (LIPIcs)*, Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 58–71.

[28] Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2017. Algebraic Laws for Weak Consistency. (2017), 26:1–26:18.

[29] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *PODC*. ACM, 73–82.

[30] Ramez Elmasri and Shamkant B. Navathe. 2000. *Fundamentals of Database Systems, 3rd Edition.* Addison-Wesley-Longman.

[31] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633.

[32] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. 2004. A Read-Only Transaction Anomaly under Snapshot Isolation. *SIGMOD Rec.* 33, 3 (Sept. 2004), 12–14. https://doi.org/10.1145/1031570.1031573

[33] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528. https://doi.org/10.1145/1071610.1071615

[34] American National Standard for Information Systems – Database Language. Nov 1992. ANSI X3.135-1992. SQL.

[35] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *IFIP Working Conference on Modelling in Data Base Management Systems*. North-Holland, 365–394.

[36] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Readings in database systems (3rd ed.)*. 365–394.

[37] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. 2008. Dynamic detection of atomic-set-serializability violations. In *ICSE*. ACM, 231–240.

[38] AK. Kingsbury and K. Patella. 2013-2019. Jepsen (reports). http://jepsen.io/analyses

[39] M. Kleppmann. 2014-2019. Hermitage: Testing transaction isolation levels. https://github.com/ept/hermitage

[40] Gavin Lowe. 2017. Testing for linearizability. *Concurr. Comput. Pract. Exp.* 29, 4 (2017).

[41] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD Conference*. ACM, 2530–2542.

[42] Kartik Nagar and Suresh Jagannathan. 2018. Automated Detection of Serializability Violations Under Weak Consistency. In *CONCUR (LIPIcs)*, Vol. 118. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 41:1–41:18.

[43] Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J. ACM* 26, 4 (1979), 631–653.

[44] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861.

[45] Ralf Schenkel, Gerhard Weikum, N Weissenberg, and Xuequn Wu. 2000. Federated transaction management with snapshot isolation. *Lecture Notes in Computer Science* (2000), 1–25.

[46] Arnab Sinha and Sharad Malik. 2010. Runtime checking of serializability in software transactional memory. In *IPDPS*. IEEE, 1–12.

[47] William N. Sumner, Christian Hammer, and Julian Dolby. 2011. Marathon: Detecting Atomic-Set Serializability Violations with Conflict Graphs. In *RV (Lecture Notes in Computer Science)*, Vol. 7186. Springer, 161–176.

[48] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *OSDI*. USENIX Association, 63–80.

[49] G. Weikum and G. Vossen. 2002. Concurrency Control: Notions of Correctness for the Page Model. *Transactional Information Systems* (2002), 61–123.

[50] wikipedia. 2022. Read_Only_Transactions. https://wiki.postgresql.org/wiki/SSI#Read_Only_Transactions

[51] Du el at. Xiaoyong. 2017. Big data management. (2017).

[52] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 279–294.

[53] Min Xu, Rastislav Bodík, and Mark D. Hill. 2005. A serializability violation detector for shared-memory server programs. In *PLDI*. ACM, 1–14.

[54] Maysam Yabandeh and Daniel Gómez Ferro. 2012. A critique of snapshot isolation. In *EuroSys*. ACM, 155–168.

[55] Kamal Zellag and Bettina Kemme. 2014. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *VLDB J.* 23, 1 (2014), 147–172.