IsoDiff: Debugging Anomalies Caused by Weak Isolation

Yifan Gan
The Ohio State University
gan.101@osu.edu

Xueyuan Ren
The Ohio State University
ren.450@osu.edu

Drew Ripberger
Sycamore High School
drew.ripberger@gmail.com

Spyros Blanas
The Ohio State University
blanas.2@osu.edu

Yang Wang
The Ohio State University
wang.7564@osu.edu

ABSTRACT

Weak isolation levels, such as Read Committed and Snapshot Isolation, are widely used by databases for their higher concurrency, but may introduce subtle correctness errors in applications that only experts can identify.

This paper proposes IsoDiff, a tool to help a developer debug the anomalies caused by weak isolation for an application. To address the challenge that the number of anomalies can be non-polynomial with respect to the number of types of transactions, IsoDiff finds a representative subset of anomalies involving different transactions, operations, and problematic patterns. To reduce false positives, IsoDiff proposes two novel methods (correlation detection and timing relationship check) to eliminate as many false positives as possible and further provides a mechanism to incorporate the developer's feedback to eliminate the remaining ones.

The evaluation of IsoDiff on TPC-C and seven real applications under Snapshot Isolation and Read Committed isolation shows that IsoDiff can balance computation time and the coverage of anomalies; it can automatically eliminate a significant portion of false positives; and its feedback mechanism allows a developer to express the root cause of false positives, which can eliminate many false positives with only a small number of developer hints.

PVLDB Reference Format:
Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas and Yang Wang. IsoDiff: Debugging Anomalies caused by Weak Isolation. *PVLDB*, 13(11): 2773-2786, 2020.
DOI: https://doi.org/10.14778/3407790.3407860

1. INTRODUCTION

In database systems, *isolation* defines how concurrent transactions can interleave, serving as a contract between the applications and the database system. Database systems support multiple isolation levels to offer applications a trade-off between correctness and performance. Because of higher concurrency, and thus better performance, "almost all SQL databases use Read Committed as the default isolation level, with some only supporting Read Committed or Snapshot Isolation" [17] and 86% of all responses to a 2017 survey of DBAs say that "most" or "all" of their transactions run under Read Committed [30].

However, comparing to Serializable, such weak isolation levels are hard to understand and thus can introduce anomalous executions a developer may not be aware of. To address this problem, there is substantial work in formally defining isolation levels weaker than Serializable [10–13, 17, 21, 35]. Based on these theories, existing works model the execution of transactions as a dependency graph and identify anomalies by searching for certain types of cycles in this graph (called Δ cycles in this paper) [20, 23, 39].

However, even with the help of these theories and tools, correctly identifying and repairing isolation anomalies is not easy even for experts because existing methods may produce many false negatives and false positives:

First, since the number of cycles in a graph can be non polynomial with respect to the number of vertices and edges, it is infeasible to identify all Δ cycles. As a result, existing tools usually just determine the existence of Δ cycles by reporting one or a few cycles among many other real problems, which will introduce false negatives [12, 20, 23, 39]. This is not ideal for debugging purposes, since a developer will naturally focus on the simplest fix for one anomaly, instead of thinking of fundamental changes to the application to tackle multiple anomalies at once.

Second, existing methods can introduce a large number of false positives mainly due to two reasons. First, because of various kinds of constraints in the application, the execution corresponding to a Δ cycle may never happen in practice. Second, some applications can tolerate certain unserializable executions, which means that even if a Δ cycle occurs, it is not a real problem to the application.

To address these challenges, this work proposes IsoDiff, a tool to help debug anomalies caused by using weaker isolation levels, with an emphasis on Snapshot Isolation and Read Committed.

To solve the dilemma that identifying all Δ cycles is infeasible and identifying a few may miss fundamental solutions, IsoDiff introduces an algorithm to identify a representative subset of Δ cycles. It searches for Δ cycles involving different transactions, operations, and problematic patterns, and tries to identify the simplest way to eliminate these Δ cycles.

To reduce false positives, IsoDiff takes two complementary approaches. On the one hand, it introduces new algorithms to identify false positives automatically: IsoDiff (1) identi-

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407860

```
1  purchase(customer_id,item_price)
2  BEGIN
3     SELECT @t = total FROM Order WHERE id =
          @customer_id;
4     UPDATE Order SET total = @t + @item_price
          WHERE id = @customer_id;
5  COMMIT
```

Figure 1: Logic of a purchase transaction.

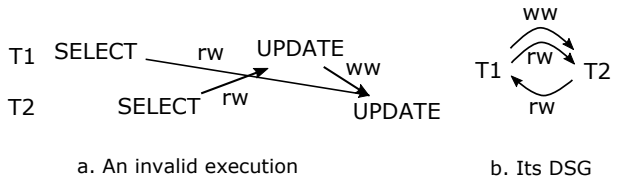

Figure 2: Identifying anomalies using the DSG.

fies co-occurring dependencies (*correlations*) that reflect relationships between different operations from the SQL trace of an application, and checks whether such correlations will invalidate a Δ cycle; (2) performs a *timing relationship* check that is rooted in the observation that, for operations op_1 and op_2, it is impossible for op_1 to happen before op_2 and op_2 to happen before op_1 simultaneously.

Eliminating all false positives automatically is extremely hard, if not impossible. Therefore, IsoDiff introduces a mechanism to incorporate a developer's knowledge. Asking a developer to analyze each Δ cycle is infeasible because of the large number of Δ cycles, but our observation is that many false Δ cycles share the same root cause. Following this observation, IsoDiff defines a developer's knowledge as certain properties on the dependency graph. Such presentation is flexible to describe various root causes and can be seamlessly integrated into the previous algorithms.

We have applied IsoDiff to TPC-C and seven open-source applications to measure their anomalies with Snapshot Isolation and Read Committed isolation. We find that:

- IsoDiff successfully reduces the computation overhead and thus makes analysis tractable even for complex applications: the longest experiment produces a dependency graph with around 0.1K vertices and 130K edges, and it takes 46 minutes to identify and validate 40K Δ cycles on a single machine.
- The timing relationship check of IsoDiff invalidates up to 85% of the found Δ cycles; the correlation check of IsoDiff invalidates up to 55% of the found Δ cycles.
- By manually analyzing the reports for two applications, we find the false positives are caused by a few root causes, which allows developers to remove false positives by analyzing a small number of Δ cycles and providing feedback.

2. BACKGROUND AND MOTIVATION

To motivate the problem, we show a concrete example about how a weaker isolation level can introduce problems. Suppose an online shopping application has provided a transaction (Figure 1) to purchase an item and suppose multiple purchase transactions can be executed concurrently.

Among all isolation levels, Serializable, which means the concurrent execution of multiple transactions is equivalent to a serial execution, frees the developers from the frustrating task of reasoning about concurrent interleavings. When using weaker isolation levels, however, the situation is less ideal. For example, Read Committed only guarantees a read operation will retrieve a committed value, which may cause problems for the example in Figure 1: if a user executes two purchases concurrently, both transactions may execute the "select" statement at the same time and then both add the price of the item to the original value of total, which means the final total value will only include one item. Such anomaly is allowed by Read Committed since both select statements indeed read committed values.

To identify anomalous executions—executions that can happen under a weaker isolation level but cannot happen under Serializable—the theoretical foundation is Adya et. al's definition [12] of isolation levels, which models the execution of concurrent transactions as a dependency serialization graph (DSG) and defines different isolation levels as preventing different types of cycles in the DSG.

To be concrete, this work models a transaction as a list of operations (e.g. read, write, etc) and defines different types of dependencies across operations: a read dependency (wr) occurs if one operation reads an object created by another operation; a write dependency (ww) occurs if one operation overwrites an object created by another operation; an anti-dependency (rw) occurs if one operation overwrites an object read by another operation. For an execution, it builds the DSG, in which vertices represent transactions and edges represents dependencies between transactions. Then it defines isolation levels as certain properties on the DSG. For example, the Serializable isolation level disallows a cycle consisting of any type of edge; the Read Committed isolation level disallows a cycle consisting of wr and ww edges (in other words, it allows cycles with at least one rw edge).

Consider the example in Figure 1: Figure 2.a presents the problematic execution mentioned above, which can be converted to the DSG in Figure 2.b. We can find two cycles in the DSG, both with one rw edge. Therefore, we can know this execution is not allowed by Serializable, which disallows any cycles, but is allowed by Read Committed, which allows cycles with at least one rw edge.

Following these definitions, existing works try to identify anomalies by searching cycles that are allowed in weaker isolation levels but not allowed in Serializable. They usually first build a static dependency graph (SDG), which is essentially a DSG with all the possible (anti-)dependency edges, and then search for certain types of cycles inside the static dependency graph. For example, for Read Committed we should identify cycles with at least one rw edge; for Snapshot Isolation prior work shows that we should search for cycles with two consecutive rw edges [20].

Although prior work provides a solid theoretical foundation for identifying anomalies caused by using weaker isolation levels, for the purpose of debugging prior work suffers from a high number of false negatives and false positives. Since they aim to determine the existence of Δ cycles, they only need to identify one Δ cycle [20, 23, 39] and thus miss opportunities to tackle multiple anomalies at once; because they do not take other constraints, such as application semantics, into consideration, they may report anomalies that never happen or do not matter to the application.

3. DESIGN

Formally, this paper has the following goal: Given a set of transactions $\mathbb{T}$ and a specific database isolation level $\mathcal{I}$, find a representative subset of concurrent executions allowed under $\mathcal{I}$ but not allowed under isolation level SERIALIZABLE. Before describing the details of our solution, we first present definitions that are used throughout this paper.

DEFINITION 3.1. *Transaction Class. A transaction class T consists of a list of operations $[op_1, op_2, ..., op_n]$ interacting with the database. Each operation is a tuple with two elements, in which the first is the action (i.e. read or write), and the second is the designated table and column.*

Similar as previous works [23, 39], we use the concept of transaction class to group "similar" transactions instances, which have the same list of operations, but may have different values for these operations at runtime. Section 3.2 describes how we summarize transaction classes from the traces of transaction instances.

DEFINITION 3.2. *Operation Dependency. $op_i \rightarrow op_j \Leftrightarrow (op_i.t.c = op_j.t.c) \wedge (op_i.action = write \vee op_j.action = write) \wedge$ (op_j reads/writes a value read/installed by op_i).*

An operation can be either a *read* or a *write* and a dependency requires at least one operation in the pair to be a write. Hence, dependencies have three types: *ww* dependency (write dependency), *wr* dependency (read dependency), and *rw* dependency (anti-dependency) [12].

DEFINITION 3.3. *Transaction Dependency. Two transaction classes $T_i \rightarrow T_j \Leftrightarrow \exists op_m \in T_i, \exists op_n \in T_j, op_m \rightarrow op_n$.*

When there are multiple operation dependencies between two transaction classes, we merge such dependencies at the transaction level, and record the mapping from the transaction dependency to its operation dependencies. The type of a transaction dependency is the union of all the types of dependencies of its corresponding operations.

DEFINITION 3.4. *Dependency Graph. We define two types of dependency graphs. A transaction dependency graph is $G_T := (V_T, E_T)$, where $V_T = \{T : T$ is a transaction class$\}$, $E_T = \{(T_i, T_j) : T_i \rightarrow T_j\}$; an operation dependency graph is $G_{op} := (V_{op}, E_{op})$, where $V_{op} = \{op : op \in T\}$, $E_{op} = \{(op_i, op_j) : op_i \rightarrow op_j\} \cup \{(op_i, op_j) : op_i$ and op_j are consecutive operations in a transaction class$\}$.*

As discussed in Section 2, to identify anomalies which might happen under a certain isolation level, IsoDiff builds a static dependency graph with all the possible dependency edges that might happen. Therefore, between a pair of operations op_i and op_j which might have dependencies, since either can happen first, IsoDiff will draw two edges $op_i \rightarrow op_j$ and $op_j \rightarrow op_i$ in G_{op} and between the corresponding transaction classes in G_T. Section 3.3 presents the details.

DEFINITION 3.5. *Δ cycle. A Δ cycle is a cycle that can happen in the transaction dependency graph under $\mathcal{I}$ but cannot happen under* SERIALIZABLE. *When $\mathcal{I}$ is* READ COMMITTED, *a Δ cycle should contain at least one rw edge; when $\mathcal{I}$ is* SNAPSHOT ISOLATION, *a Δ cycle should contain at least two consecutive vulnerable rw edges (i.e. an rw edge with no ww edge between the same pair of transactions).*

Algorithm 1: Overview of IsoDiff

```
input  : Database traces L
input  : Database isolation level I
input  : Parameters to balance computation time
         and accuracy ℓ and k
output: Ways to remove all anomalies
1  T ← get_txn(L)
2  G_T, G_op ← gen_graph(T)
3  Correlation ← get_corr(T, L);
4  while true do
5      Δcycles ← search_cycle(k, ℓ, Correlation,
         G_T, G_op, I)
6      if Δcycles = ∅ then
7          break
8      solution ← set_cover(Δcycles)
9      remove edges related to solution in G_op and G_T
10     report ← report ∪ solution
11 return report
```

The definition of Δ cycle for READ COMMITTED is derived from [12], and the definition of Δ cycle for SNAPSHOT ISOLATION is derived from [20]. At a high level, the major goal of IsoDiff is to search Δ cycles in the transaction dependency graph, because they represent anomalies that can happen under $\mathcal{I}$ but cannot happen under SERIALIZABLE.

3.1 Overview of IsoDiff

Algorithm 1 presents an overview of IsoDiff: IsoDiff first generates all the transaction classes of the target application by parsing the trace of the application (line 1); then it builds both the transaction dependency graph (G_T) and the operation dependency graph (G_{op}) from the transaction classes (line 2); then it searches for correlation among different dependency edges, which can be used to refine the following search (line 3); the core of IsoDiff is a multi-iteration algorithm, each iteration of which tries to find a subset of problems (line 5), find the simplest way to repair them (line 8), and remove the corresponding edges by simulating the solution (line 9). Finally IsoDiff will report all solutions to the developer (line 11) for feedback, and the developer may re-run IsoDiff with new feedback till no problem is left.

The following sections will present each step in detail, using the code in Figure 1 as an example.

3.2 Generating transaction classes (get_txn)

There are two ways to generate transaction classes for a database application: one is to record and parse the SQL trace from the application (dynamic analysis) and the other is to analyze the source code of the application to extract possible SQL transactions (static analysis). Dynamic analysis is easy to implement but may miss rare transactions; static analysis in theory can capture all transactions but whether it can scale to large applications is questionable. Similar as prior work [23, 39], IsoDiff uses dynamic analysis.

In the first step, we run the application and configure the database to record all the transactions in SQL format. Note that we don't record the exact read and write sets of each transaction, since they depend on the values of the parameters of each transaction: analyzing transactions with

specific parameter values would increase the chance of missing dependencies. Instead, IsoDiff assumes two statements accessing the same column may have a dependency.

In the second step, IsoDiff uses the open-source SQL parser pglast [6] to parse each SQL statement into an Abstract Syntax Tree (AST). The content of the AST depends on the statement. For example, for a select statement "select name from Users where id=1", the root of the AST is a node indicating this AST is for a select statement; it has three children: a "target list" node identifies the result column(s) of the statement (name in this example); a "fromClause" node describes the content of the from clause (Users in this example); and a "whereClause" node describes the content of the where clause. Both fromClause and whereClause may link to other ASTs if they include subqueries.

In the third step, IsoDiff converts each AST into a sequence of read and/or write operations. To achieve this, IsoDiff first walks through each AST and converts the AST into a set of operations: the root node is an operation, whose type is determined by itself (i.e. select, update, etc) and whose target is the combination of the fromClause node and the target list node; the whereClause is another operation, whose type is "read" and whose target is determined by its content; if the AST link to other ASTs, IsoDiff will recursively walk through them and generate more operations. IsoDiff then connects these operations based on their order of execution. To be specific, for each statement, IsoDiff orders operations in the where clause ahead of the main operations; for complicated where clauses, IsoDiff orders operations of JOIN ahead of others and orders operations of a subquery ahead of the operations of its outer statements; when there exist multiple similar operations, IsoDiff orders them as they appear in the statement.

Finally, IsoDiff identifies all distinct sequence of operations and marks each distinct sequence as a transaction class. Note that this approach is different from some previous works that define transaction classes based on application semantics (i.e. transactions generated by one application function belongs to one transaction class) [20]: on the one hand, if an application function has internal if branches or loops, it may generate different sequences of operations at runtime and they are considered as different transaction classes in IsoDiff; on the other hand, if multiple application functions generate the same sequence of operations, they are considered as the same transaction class in IsoDiff. We choose this approach because the operation sequence is critical for the purpose of identifying anomalies. Previous works [20] manually "split" the corresponding transaction class if an application function has internal if branches, which is what IsoDiff achieves automatically.

Example. Using Figure 1 as the example, IsoDiff will see one transaction class: $T_1 = [r_1(id),\ r_2(total),\ r_3(id),\ w_4(total)]$, in which r_1 and r_2 are from the SELECT statement and r_3 and w_4 are from the UPDATE statement.

3.3 Building dependency graphs (gen_graph)

Algorithm 2 shows how IsoDiff generates both the transaction dependency graph G_T and the operation dependency graph G_{op}.

IsoDiff examines all the operation pairs between different transaction classes (lines 2–3) and if they may be involved in a dependency, IsoDiff adds corresponding edges in G_{op} and G_T (lines 5–6); since the dependency may happen in either direction, IsoDiff adds edges in both directions (line 7); furthermore, IsoDiff adds edges for consecutive operations in G_{op} (lines 8–10).

Algorithm 2: Generating dependency graphs

input : Transaction set $\mathbb{T}$
output: Transaction dependency graph $G_T(V_T, E_T)$
Operation dependency graph $G_{op}(V_{op}, E_{op})$

```
1  pre-processing(𝕋)
2  for T1 ∈ 𝕋, T2 ∈ 𝕋 and T1 ≠ T2 do
3      for op1 ∈ T1 and op2 ∈ T2 do
4          if op1 and op2 access the same column and
            one is a write then
5              add op1 → op2 and op2 → op1 to G_op
6              add T1 → T2 and T2 → T1 to G_T (merge
                if the same edge exists)
7              add_mapping(T1 → T2, op1 → op2) and
                (T2 → T1, op2 → op1)
8  for T ∈ 𝕋 do
9      for consecutive op1 ∈ T and op2 ∈ T do
10         add op1 → op2 to G_op
11 return ⟨G_T, G_op⟩
```

Before starting, IsoDiff pre-processes the transaction classes to facilitate the following cycle search (line 1), which includes the following functions.

Replicating transaction classes. At runtime, there might be multiple instances of the same type of transaction class, and they may have dependencies as well. There are two ways to capture such dependencies among the same transaction class: the first is to add a self-loop to the node representing the corresponding transaction class in G_T; the second is to replicate the corresponding transaction class so that G_T can contain multiple nodes for the same transaction class. IsoDiff uses the second approach, because the cycle search algorithm it uses assumes no self-loops. However, the replication approach creates a question about how many times we should replicate a transaction class. We answer this question with the following definition and theorems.

Definition 3.6. *A Δ cycle C_1 is redundant to C_2 if breaking C_2 always leads to the breaking of C_1.*

For the purpose of debugging, IsoDiff only needs to report one of these Δ cycles, preferably the shorter one, because fixing one will fix the other one automatically. Based on this definition, we have proved the following theorems.

Theorem 3.1. *For Read Committed, a Δ cycle where instances of one transaction class T appear more than twice must be redundant to a shorter Δ cycle.*

Proof. Without losing generality, suppose there is a Δ cycle C_1 in which three instances T_1, T_2, T_3 of class T appear: $C_1 = A \to T_1 \to ... \to T_2 \to ... \to T_3 \to B \to ... \to A$. Since T_1 and T_2 are of the same type and there exists an edge $A \to T_1$, there must exist a similar edge $A \to T_2$. Similarly, since T_2 and T_3 are of the same type and there exists an edge $T_3 \to B$, there must exist a similar edge $T_2 \to B$. Therefore, we can construct two cycles $C_2 = A \to T_2 \to ... \to T_3 \to B \to ... \to A$ and $C_3 = A \to T_1 \to ... \to T_2 \to B \to ... \to A$,

and we can prove that 1) one of them must be a Δ cycle and 2) C_1 is redundant to that one. Obviously, both C_2 and C_3 are shorter than C_1.

To prove 1), recall that a Δ cycle for READ COMMITTED must contain at least one rw edge. In C_1, if the rw edge appears in $T_1 \rightarrow ... \rightarrow T_2$, then C_3 must contain the rw edge; if the rw edge appears in $T_2 \rightarrow ... \rightarrow T_3$, then C_2 must contain the rw edge; if the rw edge appears in other places, then both C_2 and C_3 must contain the rw edge.

To prove 2), without losing generality, suppose C_2 is a Δ cycle. For C_2, all its edges exist in C_1 except $A \rightarrow T_2$: if we break C_2 by removing an edge which is not $A \rightarrow T_2$, then of course we break C_1 as well since this edge exists in C_1; if we break C_2 by removing $A \rightarrow T_2$, since $A \rightarrow T_2$ is similar to $A \rightarrow T_1$, doing so will remove $A \rightarrow T_1$ as well, breaking C_1. Therefore, breaking C_2 will always lead to the breaking of C_1, which means C_1 is redundant to C_2. □

THEOREM 3.2. *For* SNAPSHOT ISOLATION, *a Δ cycle in which instances of a transaction class T appear more than three times must be redundant to a shorter Δ cycle.*

PROOF. The proof is similar to that of Theorem 3.1. The difference is that, for SNAPSHOT ISOLATION, the Δ cycle must contain two consecutive rw edges, which may span two segments of $T_i \rightarrow ... \rightarrow T_{i+1}$, and that is why we need one more copy to preserve the two consecutive rw edges. □

Simplifying loops. The second task the pre-processing achieves is to simplify the transaction classes caused by loops: if a transaction has an internal loop and at runtime, different transaction instances execute the loop with different number of iterations, IsoDiff will classify these instances into different transaction classes. This phenomenon would not hurt the accuracy of IsoDiff, but would certainly increase its computation complexity. IsoDiff addresses this problem with the following theorem.

THEOREM 3.3. *In a transaction class, if a sequence of operations is repeated continuously, then repeating it for more than twice will not generate new Δ cycles in G_T.*

PROOF. IsoDiff only searches simple cycles (i.e. cycles in which each vertex appears once) in G_T, because other cycles can always be decoupled into simple cycles. In a simple cycle in G_T, each vertex (i.e. transaction class) has exactly one incoming edge and one outgoing edge. Therefore, even if we repeat a sequence of operations more than twice in a transaction class, at most two of them will be involved in any simple cycle in G_T. □

This theorem indicates that, assuming a transaction has an internal loop and each iteration generates the same sequence of operations, then IsoDiff only needs to mark the one with two iterations as a transaction class and can ignore all others. Of course, if different iterations executes different sequence of operations, then this theorem does not apply.

Tagging unique IDs. A common solution developers use to avoid dependencies is to make transaction operations commutative, by using a unique ID (e.g. customer_ID) with each transaction. Such unique IDs can be inferred from dynamic analysis: if the values of a certain variable are always different between any pair of concurrent transactions, IsoDiff can infer that this variable is a unique ID. However, this approach requires a SQL trace from a real concurrent execution. For most of our applications, however, since we trigger their functions manually, we don't have access to such a real execution and thus have to manually tag unique IDs based on our best understanding about the application. In practice, when the trace about a real concurrent execution can be collected, automatic dynamic analysis should be feasible.

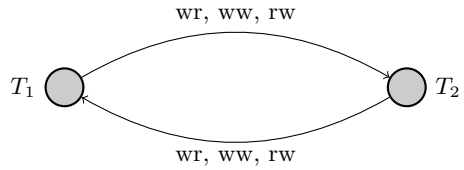

Figure 3: Transaction Dependency Graph

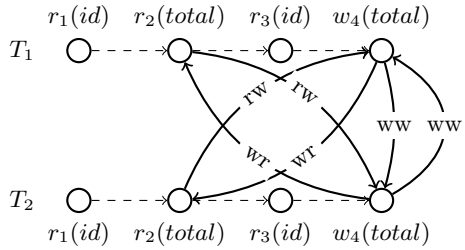

Figure 4: Operation Dependency Graph

Example. Taking Figure 1 as the example, for READ COMMITTED, IsoDiff generates G_T as shown in Figure 3 and G_{op} as shown in Figure 4. Note that for READ COMMITTED, IsoDiff replicates transaction classes twice, and this example assumes *id* is not unique: assuming uniqueness would eliminate all dependency edges.

3.4 Finding correlations (get_corr)

DEFINITION 3.7. *Dependency correlation. $op_i \leftrightarrow op_j$ is correlated with $op_k \leftrightarrow op_m$ if $op_k \leftrightarrow op_m \Rightarrow op_i \leftrightarrow op_j$ ($op_i \leftrightarrow op_j$ means $op_i \rightarrow op_j$ or $op_i \leftarrow op_j$)*

We are interested in finding such dependency correlations for two reasons: first, both previous works and this work show such correlations are helpful to remove false positives. For example, a prior work [20] shows that to find anomalies for SNAPSHOT ISOLATION, we should try to search for cycles with two consecutive vulnerable rw edges, i.e. rw edges not correlated with ww edges. Furthermore, our work shows how to extend such ideas to general cases (Section 3.6). Second, in the applications we investigated, such correlations are quite common: a transaction often has multiple statements using the same ID (e.g. customer ID, shopping cart ID, etc), and thus if one of them has a dependency edge to another transaction, all others will have similar edges.

IsoDiff searches for such correlations in two steps: first, it checks, for each transaction class T, whether there exists any pair of operations op_i and op_j, which *always* access the same row. Once again, such relationship can be found by either static or dynamic analysis and IsoDiff uses the dynamic approach: as long as for all transaction instances of T in the trace, op_i and op_j access the same row, IsoDiff marks op_i and op_j as correlated. Algorithm 3 lines 2-5 present the pseudo

Algorithm 3: Analyze correlations in transactions

input : Transaction set $\mathbb{T}$, Database traces $\mathcal{L}$
output: Correlated edges

```
1  C ← ∅; Correlation ← ∅
2  for transaction class T ∈ 𝕋 do
3      for operation op_i, op_j ∈ T do
4          if for any instance of T, op_i and op_j access
             the same row then
5              Add pairs (op_i, op_j) and (op_j, op_i) to C;
6  for each dependency edge op_k ↔ op_m do
7      for op_x: (op_x, op_k) ∈ C do
8          for op_y: (op_y, op_m) ∈ C do
9              Add (op_x ↔ op_y, op_k ↔ op_m) to
                Correlation
10 return Correlation
```

code to perform such search. Note that i could be equal to j since an operation always correlates with itself. A limitation of our current implementation is that it only searches for operations accessing a single row and ignores those accessing a range of rows: the latter is our future work.

In the second step, IsoDiff checks each dependency edge $op_k \leftrightarrow op_m$: if the previous search shows that op_k (or op_m) always access the same row as op_x (or op_y), we can conclude that if $op_k \leftrightarrow op_m$ actually happens, then $op_x \leftrightarrow op_y$ must happen as well. In other words, $op_x \leftrightarrow op_y$ is correlated with $op_k \leftrightarrow op_m$. Algorithm 3 lines 6-9 present the pseudo code to perform such checking. Note that, once again, x and y could be equal to k and m.

As a special case of utilizing such correlation, IsoDiff looks for vulnerable rw edges (i.e., rw edges not correlated with ww edges) when isolation level is Snapshot Isolation [23]. Such edges are later used for searching Δ cycles for Snapshot Isolation, which should contain two consecutive vulnerable rw edges. Section 3.6 further shows how to utilize such correlation in general cases.

Compared to previous solutions, which try to find correlation by searching "a transaction modifies the rows it select" [20, 23], IsoDiff's solution is more general and can detect correlations through more patterns like accessing different tables with the same key.

Example. Taking Figure 1 as the example, it only has one transaction class [$r_1(id)$, $r_2(total)$, $r_3(id)$, $w_4(total)$], and IsoDiff can find that in all instances, all four operations always access the same row, so IsoDiff will mark all pairs as correlated. Then if targeting Snapshot Isolation, IsoDiff can find theres exist an rw edge between duplicates of this transaction class $T_1.r_2(total) \rightarrow T_2.w_4(total)$, but since $T_1.r_2(total)$ always accesses the same row as $T_1.w_4(total)$, which means $T_1.r_2(total) \rightarrow T_2.w_4(total) \Rightarrow T_1.w_4(total) \rightarrow T_2.w_4(total)$, IsoDiff can conclude that this rw edge is not vulnerable since it is always correlated with a ww edge.

3.5 Searching for Δ cycles (search_cycle)

Since the number of cycles in a graph can be non polynomial, it is infeasible to find all of them. Therefore, IsoDiff introduces a multi-iteration algorithm to find a representative subset of Δ cycles.

Algorithm 4: Search Δ cycles

input : k, ℓ, $\mathcal{C}$, G_T, G_{op}, $\mathcal{I}$
output: Δ_{cycle}: set of valid cycles on operation level

```
1  Δ_cycle ← ∅
2  for each dangerous path dp in (G_op, C) do
3      T_src = dp.src in G_T
4      T_dst = dp.dst in G_T
5      P_T ← k-shortest-path(k, [T_dst, T_src], G_T)
6      if P_T = ∅ then
7          continue
8      for each path_T in P_T do
9          for each path_op in MapPath_{T→op}(path_T, l)
             do
10             cycle_op ← path_op ∪ dp
11             for each pair of op_i and op_j in cycle_op do
12                 if op_i is ordered before op_j in the
                     same transaction class then
13                     add op_i → op_j to cycle_op
14             if cycle_op is valid then
15                 Δ_cycle ← Δ_cycle ∪ cycle_op
16 return Δ_cycle
```

For representativeness, IsoDiff follows a few principles in each iteration: first, for each dangerous path, which is essentially a pattern a Δ cycle must have, IsoDiff should find at least one Δ cycle involving it.

Definition 3.8. *Dangerous path. When $\mathcal{I}$ is* Read Committed*, a dangerous path is one rw edge. When $\mathcal{I}$ is* Snapshot Isolation*, a dangerous path is two consecutive vulnerable rw edges.*

Both definitions are derived from the definitions of Δ cycles. One can easily prove that in a graph, the number of dangerous paths is polynomial to the number of edges.

Second, if one dangerous path is involved in multiple Δ cycles, IsoDiff should first try to find shorter ones, because shorter ones are easier to analyze and breaking them can often break longer ones as well.

Third, if for the same dangerous path and for the same length, there are still many Δ cycles, IsoDiff should try to select Δ cycles that involve different transaction classes.

Algorithm 4 presents the details of IsoDiff's cycle search algorithm: in the first phase, for each dangerous path, IsoDiff uses the k-shortest-path algorithm [43] to search for paths between the two ends of the dangerous path in the transaction dependency graph G_T (lines 2-5). This idea serves several purposes: first, by performing the search for each dangerous path, IsoDiff will not miss any dangerous paths, satisfying our first principle; second, by searching for paths between the two ends of the dangerous path, we can eventually connect the found path and the dangerous path to form a Δ cycle (line 10); third, by using the k-shortest-path algorithm, we focus on shorter cycles, satisfying our second principle, while limiting the overhead of the algorithm to polynomial; finally, by searching in G_T first instead of G_{op}, IsoDiff tries to diversify the types of transaction classes involved in these Δ cycles, satisfying our third principle.

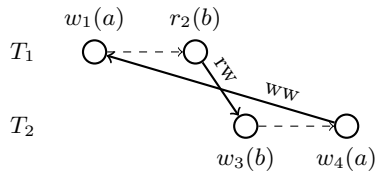

Figure 5: Example of timing violation.

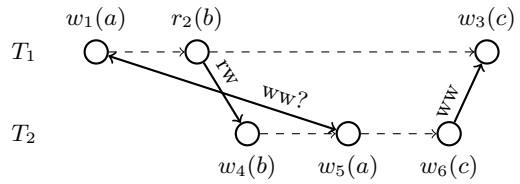

Figure 6: Example of a Δ cycle invalidated by correlation.

In the second phase, IsoDiff converts Δ cycles in G_T to paths in G_{op} (lines 8-13) to check their validity (line 14). However, a Δ cycle in G_T can be mapped to many paths in G_{op}, because one dependency edge in G_T could be mapped to various dependency edges in G_{op} (see Definition 3.3). To limit the overhead, IsoDiff randomly samples l of them ($MapPath_{T\to op}(path_T, l)$) to balance the overhead and the accuracy of this algorithm.

Algorithm 5: $MapPath_{T\to op}$

```
   input  : path_T, ℓ
   output: L_path_op: List of path_op
1  L_path_op ← ∅
2  count ← 0
3  while count ≤ ℓ do
4      path ← ∅
5      for each edge E_T ∈ path_T do
6          SetE_op ← map E_T to operation dependencies
7          randomly pick E_op from SetE_op
8          path ← path ∪E_op
9      if path not in L_path_op then
10         L_path_op ← L_path_op ∪ path
11         ℓ ← ℓ + 1
12 return L_path_op
```

Algorithm 5 shows the detail of $MapPath_{T\to op}$: for each edge in G_T, this function maps it to edges in G_{op} and randomly chooses one of them (lines 5-8); after generating a path, this function checks whether it has already been generated (lines 9-11); it repeats this procedure to get l paths.

Example. In Figure 3, IsoDiff can find two *rw* edges and thus two dangerous paths for Read Committed. Correspondingly, it will find two Δ cycles in G_T: $T_1 \xrightarrow{rw} T_2 \to T_1$ and $T_1 \to T_2 \xrightarrow{rw} T_1$. For each of them, IsoDiff can map it to two paths in G_{op}. For example, $T_1 \xrightarrow{rw} T_2 \to T_1$ can be mapped to $\{T_1.r_2 \xrightarrow{rw} T_2.w_4,\ T_2.w_4 \xrightarrow{ww} T_1.w_4,\ T_1.r_2 \to T_1.w_4\}$, and $\{T_1.r_2 \xrightarrow{rw} T_2.w_4,\ T_2.r_2 \xrightarrow{wr} T_1.w_4,\ T_1.r_2 \to T_1.w_4,\ T_2.r_2 \to T_2.w_4\}$. Note that some of these paths are equivalent so IsoDiff will skip them (see Section 4).

3.6 Validating Δ cycles

Unlike the Adya et. al's work [12], which generates a dependency graph from a real execution, our work inherits [20]'s idea to build a dependency graph with all possible dependency edges. In this approach, however, there is a chance that a Δ cycle we find would never happen in practice. Furthermore, it is also possible that a Δ cycle can happen but it is tolerable by the application. (For example, an application may take a compensating action to reverse the real-world effect of the violation, such as cancelling a duplicate order). Both cases will introduce false positives to IsoDiff. In this section, we present two techniques to detect invalid Δ cycles automatically, and one technique to incorporate the developer's knowledge into the analysis.

Timing violation. It is possible that a Δ cycle found in the previous step has an internal cyclic happened-before relationship, which will never happen. For example, consider the transactions in Figure 5: for Read Committed, IsoDiff can find the following Δ cycle: $T_1.r_2 \xrightarrow{rw} T_2.w_3$ and $T_2.w_4 \xrightarrow{ww} T_1.w_1$. However, this execution should never happen: it is impossible for r_2 to happen before w_3 and for w_4 to happen before w_1 simultaneously.

To exclude such invalid Δ cycles, IsoDiff performs a validity check in G_{op}: for each Δ cycle mapped to G_{op}, IsoDiff will check whether it has any internal cycles since a cycle in G_{op} indicates a cyclic happened-before relationship, which violates timing constraint. In Figure 5, one can see that after adding transaction internal edges $T_1.w_1 \xrightarrow{rw} T_1.r_2$ and $T_2.r_3 \xrightarrow{rw} T_2.w_4$, the graph will contain a cycle.

To summarize and to avoid confusion, a Δ cycle in G_T means a violation of Serializable, which is the target of IsoDiff. When mapping a Δ cycle to G_{op}, the resulting graph may or may not contain a cycle: a cycle in G_{op} indicates violation of timing constraint, which invalidates the Δ cycle.

Correlation. As discussed in Section 3.4, dependency edges may be correlated. This means that for a found Δ cycle, there may exist some other edges correlated with the Δ cycle. Such correlated edges may form new cycles that are not allowed by the target isolation level and thus invalidate the original Δ cycle.

Consider the example in Figure 6. For Read Committed, our algorithm can find the following Δ cycle: $T_1.r_2 \xrightarrow{rw} T_2.w_4$ and $T_2.w_6 \xrightarrow{ww} T_1.w_3$. However, if $T_1.w_1 \xleftrightarrow{rw} T_2.w_5$ is correlated with any of these two edges, we have no way to place this edge: choosing $w_5(a) \to w_1(a)$ will cause a timing violation, and choosing $w_1(a) \to w_5(a)$ will generate a new cycle $T_1.w_1 \xrightarrow{ww} T_2.w_5$ and $T_2.w_6 \xrightarrow{ww} T_1.w_3$, which consists of no *rw* edges and thus is not allowed by Read Committed. Therefore, we can conclude that the original Δ cycle is invalid.

Interestingly, the concept of a "vulnerable rw" edge [20,23] is a special case of such correlation in Snapshot Isolation. Suppose an *rw* edge is correlated with a *ww* edge of the same direction: if the *rw* edge is involved in a cycle, the *ww* edge must be involved in another cycle, which replaces the *rw* edge with the *ww* edge; the latter cycle is harder to satisfy by Snapshot Isolation than the former, and thus we only need to check the latter. In other words, if an *rw* edge is correlated with a *ww* edge, we can treat it as a *ww* edge. IsoDiff generalizes this idea to general cases.

To check whether such correlation will invalidate a Δ cycle, IsoDiff first finds all edges that are correlated with the Δ cycle, then enumerates all the possible combinations of their directions: if one combination is valid under the target isolation level, then the original Δ cycle is valid; otherwise, the original Δ cycle is invalid.

To find edges that are correlated with the Δ cycle, IsoDiff first relies on correlated edges found in Section 3.4. In addition, IsoDiff searches for the following correlation: if $op_1 \rightarrow op_2$ and $op_2 \rightarrow op_3$ and all operations access one row, IsoDiff adds $op_1 \rightarrow op_3$ as a correlated edge.

The enumeration procedure has a complexity of $O(2^n)$, where n is the number of correlated edges. To reduce computation overhead, first, IsoDiff checks the validity after adding each correlated edge, instead of checking it after enumerating all, so as to prune invalid correlated edges earlier. Second, IsoDiff sets a limit on the enumeration time, i.e., it reports a Δ cycle whose correlated edges cannot be enumerated within a given amount of time as valid. In Section 5.3 we show that a limit of 15s allows IsoDiff to fully enumerate a high percentage of Δ cycles in many applications.

Developer knowledge. IsoDiff provides a mechanism to incorporate the developer's knowledge about certain properties of the dependency graph. This is useful for false positives that can not be automatically captured by IsoDiff (see examples in Section 5.1), and cases where unserializable executions do not violate the isolation semantics of the application, and it's hard, if not impossible, to infer such application semantics automatically. Because of the large number of Δ cycles, it is infeasible to ask a developer to validate every Δ cycle. However, during our case studies, we observe that many false positives share the same root cause, which allows IsoDiff to eliminate many false positives with from a single developer hint. For example, the TPC-C specification suggests that the *stock-level* transaction can tolerate inconsistent results. In this case, incorporating developer knowledge—stock-level should be excluded from analysis—into IsoDiff eliminates many false Δ cycles altogether.

Following this idea, IsoDiff allows a developer to express his/her knowledge as certain properties on the dependency graph. For flexibility, IsoDiff allows an expert to submit code to preprocess the dependency graph or check a Δ cycle; for simplicity, IsoDiff has included some common properties for a non-expert to customize: 1. remove a transaction class; 2. remove a dependency edge; 3. mark two dependency edges as correlated; 4. mark two dependency edges as exclusive (i.e., they should not happen together).

3.7 Eliminate found △ cycles (set_cover)

In this step IsoDiff simulates the simplest way to reduce the Δ cycles found in each iteration (i.e. remove some edges). The most common way is to remove dependency edges around a certain column, either by making the corresponding transactions commutative (e.g., always access different rows), or using stronger protection for statements accessing the column. The typical examples include using unique customer IDs or shopping cart IDs to identify rows or marking "select" statement with "for update", which essentially changes an *rw* or *wr* edge into a *ww* edge. Therefore, we formalize our problem as follows: find the minimal number of columns so that every Δ cycle includes at least one edge related to these columns. We call such columns $TargetColumns$ in the rest of this paper.

This problem can be converted to the Set Cover Problem: "Given a set of elements {1, 2, . . . , n} (called the universe) and a collection S of m sets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of S whose union equals the universe" [7]. To make the conversion, we can define each Δ cycle as an element and define each column as a set, which includes all the Δ cycles that involve any dependencies related to this column. In this way, we will try to find the minimal number of columns that can cover all Δ cycles. Although the Set Cover Problem is NP hard, it has a well-known approximate solution, which identifies the set that can cover the most number of uncovered elements in each iteration [15]. IsoDiff uses this solution.

Of course, the difficulty of making transactions commutative highly varies, and indiscriminately marking select statements with "for update" clauses will hurt performance. If a developer finds a column to be irreparable, he/she can report it to IsoDiff, and IsoDiff can remove this column from the candidate set and retry.

Example. For the example in Figure 1, IsoDiff can find that the "total" column is a $TargetColumn$. The developer may implement this suggestion by enforcing that concurrent transactions will never have the same customer_id.

3.8 Limitations

False negatives: IsoDiff relies on the dynamic analysis of SQL traces, which means that the analysis is complete only with respect to the SQL trace that is provided as input. IsoDiff will thus miss rare events (transaction code paths) that have never occurred during the trace collection period. Furthermore, IsoDiff only captures anomalies caused by using weaker isolation levels, and will miss any errors not within this scope, like writing wrong transactions or any implementation bugs in the connector, middleware or database system. IsoDiff shares such limitation with prior work that also relies on trace analysis [23, 24, 39, 46].

False positives: As discussed in Section 3.6, IsoDiff makes best effort to detect false positives automatically and relies on the developer's feedback to eliminate the remaining ones. Though IsoDiff cannot eliminate all false positives automatically, its novel detection algorithms can significantly reduce false positives compared to previous works.

Inaccuracies due to approximation: IsoDiff uses approximation in the multi-iteration cycle search and to find solutions to the Set Cover Problem. This means that, like other approximate solutions to NP-hard problems, the result may not be the optimal. We believe this is a necessary trade-off between the computation overhead and the accuracy of the analysis.

4. IMPLEMENTATION

Transforming AST to operation lists. IsoDiff uses pglast [6] to parse SQL traces into Abstract Syntax Tree (AST). We implement the transformation from the AST to the operation list with about 1100 lines of Python code.

Given the complexity of AST, we set our goal to implement only necessary functions to cover our target applications. Despite such limited scope, we find it's still a challenging task: our applications are all online transaction processing (OLTP) applications, but unlike the traditional believe

that OLTP transactions are relatively simple [37], we find many real OLTP applications actually include complicated statements like JOIN statement, subqueries, and function calls. For a JOIN operation, we model it as two read operations with the involved columns; for a subquery, we implement a recursive function to retrieve all its operations; for a function call, we model it as read operations on columns that are passed in as arguments, since we do not observe any functions to modify columns in our applications.

Main part of IsoDiff. The main part of IsoDiff, which includes building dependency graphs, finding correlations, searching Δ cycles, and the approximate set cover algorithm, is implemented in C++ with around 2000 lines of code. To optimize the performance of IsoDiff, we use multi-threading to parallelize the task of cycle search, which dominates most of running time. To be specific, in Algorithm 4, we spawn one worker thread per core, assign each dangerous path as a task, and dispatch them to different worker threads through a shared buffer. Once all tasks are finished in one round, the main thread retrieves all Δ cycles for set cover analysis, and then updates the dependency graphs accordingly. To avoid searching redundant cycles involving different copies of the same transaction class, we compute a hash for each cycle, by mapping duplicates of a transaction class to the same one, and compare newly found cycles with ones already searched.

Using IsoDiff. To analyze a database application with IsoDiff, a user needs to first run the database application and collect the SQL traces. In our experiments, we use MySQL and configure the "general_log" option to 1 to enable trace collection. Then the user can run IsoDiff over the collected trace. IsoDiff will output suggested $TargetColumns$ and the found Δ cycles, represented by the specific sequences of operations that lead to anomalies. IsoDiff has two key parameters k and l (see Section 3.5) to balance accuracy and overhead. We suggest the user to gradually increase these two parameters till the result (i.e. the number of $TargetColumns$) becomes stable.

5. EVALUATION

Our evaluation tries to answer the following questions:

- How effectively can IsoDiff find anomalies in real applications (Section 5.1)?
- What is the overhead of IsoDiff (Section 5.2)?
- What is the effect of each individual technique of IsoDiff (Section 5.3)?

Applications. To answer these questions, we have applied IsoDiff to TPC-C and seven real applications: TPC-C [37] is a popular OLTP benchmark, which is a simplified version of real OLTP applications. Lightning Fast Shop (LFS) [3] is an online shop and ecommerce solution based on Python, Django and jQuery. OpenCart [4] is an open-source e-commerce platform for online merchants based on PHP. PrestaShop [5] is another PHP-based open-source e-commerce web platform providing shopping cart experience for both merchants and customers. Shoppe [8] is a Rails-based platform providing e-commerce functionality for Rails applications. WooCommerce [9] (wc) is an open-source e-commerce plugin for WordPress on a new or existing WordPress site. Attendize [1] is a ticket selling and event management platform to help users run and manage events. FrontAccounting (fa) [2] is an open-source web-based accounting system covering the Enterprise Resource Planning (ERP) chain for small enterprises. They were selected from GitHub based on their popularity (almost all of them have more than 500 stars) and the number of contributors (most of them have more than 200 contributors).

To retrieve SQL traces for these applications, 1) for TPC-C, we use its default workload generator; 2) for shoppe, lfs, opencart, prestashop, and wc, we use the traces provided by [39]; 3) for attendize and fa, we manually trigger different functions of these applications.

Table 1 presents the statistics of the dependency graphs of these applications. Note that since Read Committed needs to duplicate transaction classes once and Snapshot Isolation needs to duplicate twice, they have different statistics.

We run the automatic algorithms of IsoDiff on all cases. We manually analyze the reports of IsoDiff on TPC-C under Read Committed and FrontAccounting under Snapshot Isolation to check false positives.

Testbed. We run all experiments on CloudLab [16]. Each machine is equipped with two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz, 192GB ECC DDR4-2666 memory and one Intel DC S3500 480GB 6G SATA SSD. We use multiple machines to run different experiments in parallel, though each experiment runs on one machine.

5.1 Anomalies in real applications

Table 2 presents the number of anomalies IsoDiff finds for each application, which is quantified as the number of $TargetColumns$. Unsurprisingly, Snapshot Isolation introduces much fewer anomalies than Read Committed, since Snapshot Isolation is known to be stronger than Read Committed.

We further manually analyze the reports of TPC-C under Read Committed and FrontAccounting under Snapshot Isolation. We try to fix true problems with simple solutions, though such solutions may hurt the performance, and an efficient solution may require a significant re-design of the application, which is out of the scope of this paper.

5.1.1 TPC-C under Read Committed

Our analysis reveals both true problems and false positives. The true problems can be summarized as follows:

- Select followed by update. Similar as in Figure 1, if two concurrent transactions read the same value and update it, the final value may just include one update. In TPC-C, such pattern happens for the *new-order* transaction, which reads and updates the *district-d_next_o_id* and *stock-s_quantity* values, and for the *payment* transaction, which reads and updates the *customer-c_balance* value.
- Multiple selects interleaved with multiple updates. When one transaction updates multiple values and another transaction reads multiple values, the reads may get an unserializable result. In TPC-C, this happens between *order-status* transaction, which reads *customer-c_balance* and *orders-o_carrier_id*, and *delivery* transaction, which updates these values.

The first problem is particularly troublesome since it may introduce incorrect balance or stock values. The second problem may cause a customer to see a confusing order status (e.g. balance is changed but order is not shown). To

Table 1: Dependency graph statistic for Read Committed and Snapshot Isolation (dp = dangerous path).

Application	Read Committed					Snapshot Isolation				
	G_T		G_{op}		#dp	G_T		G_{op}		#dp
	#V	#E	#V	#E		#V	#E	#V	#E	
tpcc	20	182	906	1,490	646	30	420	1,359	3,414	0
shoppe	56	124	1,096	736	328	84	282	1,644	1,728	2,280
lfs	154	526	2,730	8,986	3,496	231	1,200	4,095	20,484	128,304
opencart	150	242	1,780	2,670	1,212	225	552	2,670	6,048	0
prestashop	420	854	5,668	9,770	4,580	630	1,938	8,502	22,086	162
wc	110	642	1,998	46,838	23,048	165	1,452	2,997	131,412	481,896
attendize	48	230	2,212	5120	2,398	72	522	3,318	12,264	21,540
fa	156	142	1982	506	252	234	324	2,973	1,188	1,944

Table 2: Number of *TargetColumns* IsoDiff finds for each application (RC=Read Committed; SI=Snapshot Isolation).

Application	RC	SI	Total Columns
tpcc	11	0	86
shoppe	12	2	159
lfs	162	10	419
opencart	19	0	407
prestashop	112	1	834
wc	31	8	106
attendize	33	7	251
fa	11	6	371

solve these problems, we mark the select statements that touch the corresponding columns with "for update".

The false positives are summarized below:

- Transaction can tolerate unserializable result. The *stock-level* and *new-order* transactions suffer from the same problem of "multiple reads interleaved with multiple updates" as described above. However, the TPC-C specification mentions "full serializability and repeatable reads are not required for the Stock-Level business transaction", which indicates *stock-level* can tolerate unserializable results. Therefore, we provide a devleoper hint that *stock-level* transactions should not be considered in the analysis.
- Transaction is not executed concurrently. The *delivery* transaction will get the oldest order and deliver it, so if two *delivery* transactions execute in parallel, they may try to deliver the same order. However, the TPC-C specification mentions "the Delivery transaction is intended to be executed in deferred mode through a queuing mechanism", which suggests the delivery transaction is not executed concurrently. Therefore, we provide a developer hint to remove a replica of *delivery*.
- Commutative updates. Two concurrent *payment* transactions executing statements "update .. SET d_ytd = d_ytd + value ..." are considered as *ww* conflict by IsoDiff, but because they are commutative, we can re-order them to break the cycle. Therefore, we provide a developer hint to remove such dependency edges.
- False *rw* dependency from single-row select to insert. This means the select statement is querying a row added by a later insert statement. In other words, the select is querying a non-existent row. Though in theory this could happen, in TPC-C, the select usually uses an ID in the where clause, which is retrieved from a previous statement, so it should never query a non-existent row. Therefore, we provide a developer hint to remove the *rw* edge.
- Application logic. This happens between an *order-status* transaction and a *new-order* transaction: *order-status* will first retrieve the latest order and then retrieve its corresponding order lines; *new-order* will insert a new order and the corresponding order lines. IsoDiff finds that they can create a similar problem as "multiple selects interleaved with multiple updates": order-status's "select latest" has an *rw* edge to *new-order*'s first insert, and then *new-order*'s second insert has an *wr* edge to *order-status*'s second select. However, if *order-status*'s "select latest" happens before *new-order*'s insert, these two transactions should work on different orders and thus the *wr* edge should not happen. Therefore, we provide a feedback that if there is an *rw* from *order-status*'s "select latest" to a *new-order*'s insert, there should be no dependency between their following operations.

Some of these false positives, like "commutative updates", may be captured automatically by more accurate analysis; some of them, like "transaction can tolerate unserializable result", is hard to capture without the developer's feedback.

5.1.2 *FrontAccounting under* Snapshot Isolation

Similarly, our analysis reveals both true problems and false positives. We summarize the true problems below:

- Write skew. We observe the classic write skew pattern [13]: a transaction computes the sum of a range of values, and then inserts a new row, which can affect the sum value; under Snapshot Isolation, two concurrent transactions may get the same sum, which should never happen under Serializable. We solve this problem by marking the select statement as "for update", which forces the second transaction to block or abort if it tries to read the same range.
- Unserializable range read. We observe this problem involving three transactions: two concurrent transactions T1 and T2 insert new rows and a third transaction T3 performs a "select max" statement, which only includes the row from T1 but not from T2. However, T2 has a *rw*

Table 3: Min/Max number of *TargetColumns* with different k and l (RC=Read Committed; SI=Snapshot Isolation).

Application	RC		SI	
	Min	Max	Min	Max
tpcc	11	19	0	0
shoppe	12	21	2	5
lfs	62	103	10	57
opencart	19	29	0	0
prestashop	112	162	1	12
wc	31	43	8	17
attendize	33	46	7	24
fa	11	19	6	13

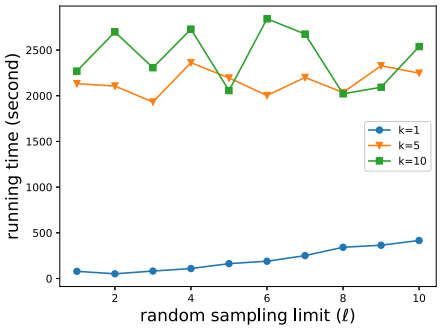

Figure 7: Running time of IsoDiff on *woocommerce (wc)* with different settings.

dependency to T1, which means T2 must be serialized before T1. Therefore, if the row inserted by T2 has a larger value than T1, there is no way to serialize them. In other words, T3 may return a value that is not the max at any moment in any serial execution. In order for this anomaly to happen, T1, T2, and T3 must come from the same user, so we solve this problem by using application-level locking (on the user ID) to never issue T1 and T2 concurrently for the same user (i.e., make T1 and T2 commutative).

The false positives are summarized below:

- False *rw* dependency from single-row select to insert: this is the same as described above. We provide a developer hint to remove the dependency.
- False dependency between select count(*) and update: IsoDiff marks this as a dependency since the statement select count(*) and the update may access the same rows, but the count of the corresponding rows is not affected by the update, so we provide a developer hint to remove this dependency. This false positive may be automatically captured by improving the accuracy of our analysis.

5.2 Overhead

IsoDiff has two parameters k and l to balance overhead and accuracy: k is used by the k-shortest-path algorithm and l determines the number of samples to get when mapping a Δ cycle to operation level. This section studies how of these parameters affect the overhead and result of IsoDiff.

For all experiments, we try k=1,5,10 and l from 1 to 10. Table 3 summarizes how they affect the result of IsoDiff. As one can see, for some applications the results vary significantly across different settings. This result confirms that arbitrarily repairing a few anomalies misses opportunities to tackle multiple anomalies at once.

To measure the impact of the k and l parameters on overhead, we measure the running time of IsoDiff on *woocommerce* (wc) under Snapshot Isolation. This setting is the most time consuming one because *woocommerce* has the most dangerous paths of all applications.

Figure 7 presents the result. In general, the running time grows with k, which is as expected. The fluctuation with different l values is due to the following reasons: first, IsoDiff maps a Δ cycle to l random operation level paths, and some of them may have more correlated edges than the others, which requires more time to perform correlation check; second, the running time is affected by the iteration in which IsoDiff prunes critical columns or edges, which incurs some randomness as well. The slowest setting ($k = 10, l = 6$), which finds about 20K valid Δ cycles and 19K invalid ones, takes about 46 minutes.

We further profile the time spent in different functions in this setting and find that 99.4% of the time is spent in search_cycle. Note that search_cycle has already been parallelized while other parts have not.

We also measure the memory consumption of different settings and find that the memory consumption is around 3GB and is stable across different applications and different settings. Of course this number may increase with a much larger application, which requires more space to store both the dependency graph and the found cycles.

To handle larger applications or reach a larger k and l, which require more running time, one can consider running IsoDiff in a distributed setting, which should not be hard since the time consuming part of IsoDiff is highly parallelizable (Section 4). To reduce memory consumption, another possibility is to use disk-based graph processing engines [14, 22, 25, 31, 32, 40, 47, 48], which makes a trade-off between memory consumption and running time.

5.3 Effects of individual techniques

In this section, we evaluate the effects of two individual techniques in IsoDiff: checking timing constraint and checking correlation. Note that correlation are used in two ways: to check whether a found Δ cycle is valid and to identify non-vulnerable *rw* edges for Snapshot Isolation.

Table 4 presents the results. As one can see, the effects of these techniques are significant for some applications: checking time constraint can invalidate up to 85% of the found Δ cycles; checking correlation can invalidate up to 55% of the found Δ cycles; and up to 94% of the *rw* edges are non-vulnerable. The real impact of identifying non-vulnerable *rw* edges may be larger than the numbers shown in the table, because for Snapshot Isolation, the dangerous path should include two consecutive vulnerable *rw* edges, and thus one *rw* edge marked as non-vulnerable may make a few other vulnerable *rw* edges harmless. Such results have confirmed the necessity of automatic checking.

For correlation checking, we further record the percentage of Δ cycles whose correlated edges can be enumerated under the time limit (i.e., 15s). As shown in Table 5, the coverage is at least 96% with some of them reaching 100%.

Table 4: Effects of individual techniques: Timing = (# cycles invalidated by timing check) / (# all found cycles); Corr = (# cycles invalidated by correlation check) / (# all found cycles); NV = (# non-vulnerable rw edges) / (# all rw edges); N/A means IsoDiff does not find any Δ cycles.

Application	Read Committed		Snapshot Isolation		
	Timing	Corr	Timing	Corr	NV
tpcc	76%	7.8%	N/A	N/A	93%
shoppe	85%	2.3%	52%	9.6%	69%
lfs	26%	21%	10%	35%	19%
opencart	85%	1.6%	N/A	N/A	91%
prestashop	17%	11%	0	0	37%
wc	67%	10%	29%	16%	94%
attendize	53%	10%	0	55%	88%
fa	63%	4.0%	0	2.6%	57%

Table 5: The coverage of Δ cycles with correlated edges.

App	RC	SI	App	RC	SI
tpcc	100%	N/A	prestashop	99.0%	100%
shoppe	100%	100%	wc	96.4%	100%
lfs	96.8%	100%	attendize	99.4%	100%
opencart	99.9%	N/A	fa	100%	100%

6. RELATED WORK

Isolation levels in practice. To balance performance and ease of programming, the database community has introduced a number of isolation levels, including Serializable [29], strict Serializable, Snapshot Isolation [13], Read Committed [13], Repeatable Read [13], Read Committed Snapshot [26], Parallel Snapshot Isolation [33], and others.

Though (strict) Serializable is usually preferred by developers and is used in many research works (including but not limited to [27, 28, 34, 36, 38, 41, 42, 44, 45]), a number of studies have shown that Read Committed and Snapshot Isolation are the most widely used isolation levels in practice [17, 30]. This is the first major motivation of our work.

Defining isolation levels. Given so many different isolation levels, how to precisely define them has become a challenging and critical problem in the database community.

The early ANSI standard [10] defines different isolation levels as preventing different phenomenon. For example, it defines Read Committed as preventing "dirty read", defines Repeatable Read as preventing "non-repeatable read" upon Read Committed, and defines Serializable as preventing "phantom read" upon Repeatable Read.

This definition is criticized by the following work for its inaccuracy [13]. Instead, the following work [13] defines isolation levels as different locking strategies. For example, Read Committed should hold write lock to the end of the transaction and hold read lock to the end of the operation; Serializable should hold both write and read locks to the end of the transaction.

However, this definition is criticized by following works as well because it does not work with databases that do not use lock implementation. To achieve both precision and generality, Adya et. al. define different isolation levels as preventing different types of cycles in the dependency serialization graph [12]. For example, Serializable should prevent any cycles and Read Committed should prevent cycles consisting of ww and wr edges. Feketa et. al. extend this definition to Snapshot Isolation, showing that Snapshot Isolation should only allow cycles with at least two consecutive rw edges [20]. Such graph-based definition has become today's standard in the database community.

A number of recent works [17, 35] attempt to make such definition more developer friendly by defining isolation from the external view. For example, Crooks et. al. [17] define Snapshot Isolation as a transaction T's operations must read from the same state s, which is the state that T transitions from.

Given the long history of finding the right definition of isolation levels, precisely understanding different isolation levels is not an easy task, especially for non-experts, which creates the concern that developers may not be able to use them properly. This is the second motivation of our work.

Finding anomalies of using weaker isolation levels. To find potential anomalies, previous works first build a dependency graph with all possible dependency edges and then search for certain types of cycles in the dependency graph. Fekete et.al. apply this idea to study the anomalies for Snapshot Isolation [18], and its following work [23] tries to automate the whole procedure. ACIDRain [39] applies this idea to find security vulnerabilities in database applications. For example, it shows that, by exploiting the anomaly, a malicious user may be able to spend his/her money twice, causing the classic "double spending" problem. To understand how frequently anomalies actually happen, Fekete et.al. build a microbenchmark to quantitatively predict the frequency of anomalies at runtime [19].

IsoDiff is built upon many ideas in this line of work, but tries to reduce both false negatives and false positives for the purpose of debugging.

7. CONCLUSION AND FUTURE WORK

This paper proposes IsoDiff, a tool to help developers debug anomalies caused by weaker isolation levels. Our analysis on TPC-C and real applications has demonstrated the effectiveness of IsoDiff and revealed potential directions to further improve such tools. In the future, we plan to utilize static analysis to automatically extract transactions and correlations, and to further improve the accuracy of false positive detection. Furthermore, our analysis shows anomalies are common in applications that use weaker isolation levels, which confirms the suspicion that developers find it difficult to translate abstract examples of anomalies to their effects on the application logic. Developing usable tools to support application development under weaker isolation levels is a promising avenue for future work.

Acknowledgements

We are grateful to the anonymous reviewers for their insightful comments. Chunzhi Su provided invaluable feedback on early versions of this project. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1908020 and SHF-1816577.

8. REFERENCES

[1] Attendize. https://github.com/Attendize/Attendize.

[2] FrontAccounting. https://github.com/FrontAccountingERP/FA.

[3] Lightning Fast Shop. https://github.com/diefenbach/django-lfs.

[4] OpenCart. https://github.com/opencart/opencart.

[5] PrestaShop. https://github.com/PrestaShop/PrestaShop.

[6] Python-pglast. https://github.com/lelit/pglast.

[7] Set cover problem. https://en.wikipedia.org/wiki/Set_cover_problem.

[8] Shoppe. https://github.com/tryshoppe/shoppe.

[9] WooCommerce. https://github.com/woocommerce/woocommerce.

[10] ANSI X3. 135-1992. *American National Standard for Information Systems-Database Language-SQL*, 1992.

[11] A. Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. 1999.

[12] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 67–78. IEEE, 2000.

[13] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[14] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He. Venus: Vertex-centric streamlined graph computation on a single pc. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1131–1142, April 2015.

[15] V. Chvatal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3):233–235, Aug. 1979.

[16] CloudLab. https://www.cloudlab.us.

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

[18] A. Fekete. Allocating isolation levels to transactions. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 206–215, 2005.

[19] A. Fekete, S. N. Goldrei, and J. P. Asenjo. Quantifying isolation anomalies. *PVLDB*, 2(1):467–478, 2009.

[20] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

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

[22] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-Scale Graphs in a Single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, page 77–85, New York, NY, USA, 2013. Association for Computing Machinery.

[23] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1263–1274. VLDB Endowment, 2007.

[24] S. Koch, T. Sauer, M. Johns, and G. Pellegrino. Raccoon: Automated verification of guarded race conditions in web applications. 2020.

[25] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[26] Microsoft SQL Server – SET TRANSACTION ISOLATION LEVEL (Transact-SQL). https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql.

[27] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting More Concurrency from Distributed Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, 2014. USENIX Association.

[28] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 517–532, Savannah, GA, 2016. USENIX Association.

[29] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[30] A. Pavlo. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *SIGMOD'17*, page 3, 2017.

[31] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 410–424, New York, NY, USA, 2015. Association for Computing Machinery.

[32] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[33] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[34] C. Su, N. Crooks, C. Ding, L. Alvisi, and C. Xie. Bringing Modular Concurrency Control to the Next Level. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 283–297, New York, NY, USA, 2017. Association for Computing Machinery.

[35] A. Szekeres and I. Zhang. Making consistency more consistent: A unified model for coherence, consistency and isolation. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for*

Distributed Data, PaPoC '18, pages 7:1–7:8, New York, NY, USA, 2018. ACM.

[36] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[37] Transaction Processing Performance Council. The TPC-C home page. http://www.tpc.org/tpcc/.

[38] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[39] T. Warszawski and P. Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *SIGMOD'17*, pages 5–20. ACM, 2017.

[40] C. Wu, G. Zhang, Y. Wang, X. Jiang, and W. Zheng. Redio: Accelerating Disk-Based Graph Processing by Reducing Disk I/Os. *IEEE Transactions on Computers*, 68(3):414–425, March 2019.

[41] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining ACID and BASE in a Distributed Database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, Broomfield, CO, 2014. USENIX Association.

[42] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 279–294, New York, NY, USA, 2015. ACM.

[43] J. Y. Yen. Finding the k shortest loopless paths in a network. *management Science*, 17(11):712–716, 1971.

[44] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: an evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[45] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang. Bcc: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *PVLDB*, 9(6):504–515, 2016.

[46] K. Zellag and B. Kemme. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *The VLDB Journal*, 23(1):147–172, 2014.

[47] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, Feb. 2015. USENIX Association.

[48] X. Zhu, W. Han, and W. Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.