Exploring Transaction Anomalies under Weak Isolation Levels for General Database Applications

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Yifan Gan, Computer Science, B.E.; M.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2021

Dissertation Committee:

Yang Wang, Advisor

Spyros Blanas

Feng Qin

Xiaodong Zhang

Abstract

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

To help developers debug the anomalies caused by weak isolation, we proposed IsoDiff, a tool to analyze transactions in database applications and identify possible anomalous executions under given weak isolation levels. 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: 1) correlation detection; 2) timing relationship check, to eliminate as many false positives as possible.

We further extend the work of IsoDiff on several respects. First, we design a random yet deterministic mechanism to provide stable performance and consistent result with control of single random seed. Second, we incorporate domain knowledge from developer via a generalized feedback framework to allow developers express their requirements and eliminate false positives with their specific usage scenarios. Third, we make augmentation on IsoDiff in consideration of practical database management system settings.

To show the effectiveness, we evaluate IsoDiff against TPC-C and other real applications under Snapshot Isolation and Read Committed, showing 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. The extensive evaluation on IsoDiff shows its generalizability and flexibility of handling various requirements from database management system and database application developers.

Dedicated to my family, friends and all associated with The Ohio State University

Acknowledgments

The six years of PhD journey in the Ohio State University has been one of the most precious experiences in my life. I feel very glad to have the great opportunity to work on problems that are both interesting and challenging, and collaborate with a group of smart, kind and supportive people.

First, I would like to express my sincere gratitude to my advisor, Yang Wang, for his invaluable guidance and strong support during my doctoral program. His brilliant research idea, substantial knowledge on various research areas and professional attitude set a vivid exemplar for me not only on the research work but also a philosophy of understanding and solving problems.

Second, I would like to thank all the committee members (Spyros Blanas, Feng Qin and Xiaodong Zhang) of my thesis dissertation not only for their helpful suggestion and feedbacks on the thesis, but also for their help on my career making it possible continue working on cutting-edge area for distributed system and database after my doctoral program.

Then, I would like to show my appreciation to all my colleagues and friends. Rong Shi, Fang Zhou, Sixiang Ma, Xueyuan Ren, Drew Ripberger and other friends in the lab have been my precious memory during my doctoral program. My PhD journey wouldn't be more joyful without working with them. I would also give thanks to

Dachuan Huang, Haowei Wu and Haicheng Chen, who provided great help including diverse research topics and future career.

Finally, I want to thank my parents, Chunlin Gan and Yanjun Li, for their understanding and unconditional support on decisions I've ever made. Also, I'd like to thank my adorable cat, Ricecake, for his consistent accompanying during my PhD journey.

Vita

August 2015 - Dec 2021	Ph.D., Computer Science and Engineering, The Ohio State University
May 2021	M.S., Computer Science and Engineering, The Ohio State University
June 2015	B.Eng., Computer Science and Technology, Huazhong University of Science and Technology
2020	Software Engineering Intern, Facebook Inc., Remote, U.S.A.
2018	Software Development Engineering Intern, Amazon.com, Seattle, WA, U.S.A.
2017	Software Development Engineering Intern, Amazon Web Service, Seattle, WA, U.S.A.

Publications

Research Publications

Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. IsoDiff: Debugging Anomalies Caused by Weak Isolation. In *the 46th International Conference on Very Large Data Bases (VLDB'20)*, Aug. 2020.

Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wPerf: Generic Off-CPU Analysis to Identify Critical Waiting Events. In *the 13th USENIX Symposium on Operating Systems Design and Implementation(OSDI'18)*, Oct. 2018

Rong Shi, Yifan Gan, Yang Wang. Evaluating Scalability Bottlenecks by Workload Extrapolation. In *IEEE 26th International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sep. 2018

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Distributed System	Prof. Yang Wang
Database Concurrency Control	Prof. Yang Wang and Spyros Blanas

Table of Contents

Page

List of Tables

List of Figures

Figure **Page**

Chapter 1: Introduction

In computer system, a transaction defines a sequence of operations that exchange and manipulate data in the storage, for example, in a database. To ease the burden of reasoning concurrency and ensure the data integrity, researchers and software developers proposed ACID: *atomicity*, *consistency*, *isolation*, and *durability*, which is a set of properties for database transactions to guarantee data validity in spite of errors. Although ACID provides strong guarantee on the data integrity, the performance, especially under the circumstance of concurrency, is greatly impacted by the *isolation* property of ACID.

In database systems, the term *isolation*, from ACID, defines how concurrent transactions can interleave, serving as a contract between the applications and the database system, and there are different isolation degrees for developer to choose. Nowadays, modern database systems usually support multiple isolation levels to offer applications a trade-off between correctness and performance. As the name suggests, the isolation level SERIALIZABLE guarantees that any transactions running concurrently can be serialized as a sequence of executing those concurrent transaction, thus providing strong guarantees on the data integrity in spite of concurrent transactions. On the other hand, a weaker isolation, suck as READ COMMITTED and SNAPSHOT ISOLATION,

allows concurrent transaction to access intermediate state of data accessed by other transaction, so that it provides higher throughput, and better performance.

This poses an opportunity for application developers, as weaker isolation levels than Serializable, such as Read Committed and Snapshot Isolation, provide more concurrency and thus better 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" [22] and 86% of all responses to a 2017 survey of DBAs say that "most" or "all" of their transactions run under Read Committed [42].

However, comparing to Serializable, such weak isolation levels are hard to understand and sometimes risky, as it could 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, 12, 13, 16, 22, 28, 49]. 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 [27, 30, 55]. These special cycles represent a subset of possible execution on the given transactions, which only show in the weaker isolation level and the strong isolation level, Serializable, prohibits such types of cycles. In this thesis, we call such types of cycles Δ cycles, as they differentiate the weaker isolation level from Serializable.

Even though understanding and recognizing anomalies under weak isolation become feasible 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 [13, 27, 30, 55]. 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. It would be reasonable to provides more details, including more anomalous executions and summary of commons on the anomalies, for developers to have better understanding on problems possibly introduced in their systems.

- Second, existing methods can introduce numerous 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. For example, if an e-commerce system allows only one purchase could be made per customer, the problem of double spending, i.e., making one payment on two orders, will not happen when running on such e-commerce platform. 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 thesis introduces 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. Furthermore, this algorithm provides parameters to balance the coverage of Δ cycles and computation time, which allows a developer to gradually increase computation time till coverage is satisfactory.

To reduce false positives, IsoDiff takes two complementary approaches. On the one hand, it introduces new algorithms to identify false positives automatically: IsoDiff (1) identifies 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. The complexity arises from not only the problem itself a non-polynomial problem, but also there are constraints to consider from practice. 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. While IsoDiff allows an expert developer to write code to express such properties for maximal flexibility, in our studies, we find most of such properties can be categorized into a few types. We demonstrate the

flexibility and effectiveness of feedbacks with case study on specific implementation of database management system.

Since IsoDiff runs a non-polynomial algorithm to search Δcycle set, we try to improve it with data structure and algorithm tailored for the search task. We also introduce a random and deterministic framework to guarantee the performance of IsoDiff is stable and result is consistent controlled with a single seed.

We have applied IsoDiff to TPC-C and other 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.

- We introduce a systematic approach, called feedbacks, to express special requirements to IsoDiff with developers' knowledge expressed via short code snippet, which can further increase the analysis accuracy of IsoDiff checker.

Outline. The rest of this thesis is organized as following. Chapter 2 gives background of anomalies related to weak isolation levels and our motivation on designing IsoDiff. Chapter 3 and Chapter 4 present the design and implementation of IsoDiff respectively. In Chapter 5, we evaluate IsoDiff on TPC-C and real applications. We describe our effort on extending IsoDiff with different perspectives in Chapter 6, Chapter 7 and Chapter 8. Finally, we conclude the contribution of this thesis in Chapter 9.

Chapter 2: Background and Motivation

This chapter gives an overview of ACID transaction, definition of different isolation levels from different work. After introducing weak isolation levels, it will review anomalies caused by weak isolation levels. The final part discuss about previous work on detecting, analyzing and preventing anomalies of weak isolation level, and the motivation of this thesis.

2.1 ACID Transactions

Database is a computer system that collects and organizes data for retrieving and manipulating expeditiously. In database application, a *transaction* is a sequence of read and write operations interacting with a shared data storage. One transaction executes atomically as if a single operation, even though actually it performs multiple reads and writes with the database. The benefit of this abstraction is that developers can simplify the design of their program involving transactions, which they could treat as a single, inseparable piece of code for data accessing. To achieve it, developers summarized four desirable properties for database transactions: *Atomicity*, *Consistency*, *Isolation*, and *Durability*, and they are often referred to as *ACID properties* for transactions. We summarize the definition of the ACID property as following:

- *Atomicity.* The *atomicity* property of transaction is the "all or nothing" guarantee: the transaction is successfully executed if *all* of its operations are completed as expected; otherwise the transaction is discarded, and the database is restored to the state before executing the transaction. If all operations in transaction are executed, we say the transaction is *committed*; if the transaction is discarded, we say the transaction is *aborted*.

- *Consistency.* The *consistency* property of transaction guarantees that the state transition of a database from one legitimate state to another. By legitimate, it refers to that a list of invariant constraints, specified by the database or possibly by application developers, should not be violated. One common example is that two rows of data in the same table should not have same identifier, or in another word, same *primary key*. Before the actual transaction is committed, the Database Management System (DBMS) checks if there's any violation of given constraints: if so, the transaction will be aborted, otherwise it will be committed.

- *Isolation.* Originally, the *isolation* property of transaction guarantees that transactions running concurrently cannot see each other's intermediate results, so that the programmers of database application can treat concurrent transaction as solely a set of transaction running in sequence. This isolation guarantee is also referred to as Serializable. However, developers define more isolation levels than Serializable essentially for the best suit of their design, which is often the performance. These isolation levels are usually weaker than Serializable

as they allow concurrent transactions observe intermediate states in some degree, thus allowing more transactions running concurrently.

- *Durability.* The *durability* property guarantees that the effects of one transaction are durable if it is accepted by DBMS. By durable it means that the effects will be pertained even if the system may fail due to unexpected circumstances, e.g., power outage, node crash, and the records should be recovered if the system come back to normal. Durability is usually guaranteed by logging and recovery techniques.

With ACID property, transactions set clear boundary from other implementation code, and provide a data-accessing model easy for application developers to reason about the correctness. With atomicity and durability, developers are free from partially updating data when interacting with database, and their transactions are persistently kept once they are committed. With consistency, developers ensure that transactions can only transit from one valid state to another, following developer-defined rules. With isolation level, developers can determine how multiple transactions should be handled when they are running concurrently.

2.2 Weak Isolation and Anomalies

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 interleaving of transactions. Even though SNAPSHOT ISOLATION is a desirable isolation level, but it comes with drawback on the performance. When developers are willing to have better performance, they

usually seek for a weaker isolation than SNAPSHOT ISOLATION, but the situation is less ideal.

Generally, a transactions can be treated as a list of read and write operations, in which each operation is associated with data it access. When two or more transactions are running concurrently, the operations on these transactions could access the same piece of data in the database. A *history* captures the running sequence and the effects of operations of transaction(s). *Data conflict* happens if two operations from two transactions access the same data item, and at least one of the two operations is a write operation. Operations in concurrent transactions could interleave in different way, which build different data conflicts. The isolation property of transaction rules out data conflicts and provides guarantees to prevent from improper data accessing in concurrently running transactions.

ANSI Isolation Definition. In ANSI-SQL Standard [10], it introduces four different isolation levels (from strongest to weakest): *Serializable*, *Repeatable Read*, *Read Committed*, and *Read Uncommitted*. *Serializable* follows classic definition that transactions running concurrently could be treated as running one by one in certain order. The rest of three isolation levels are defined with undesirable consequences, called *Phenomena*: *Dirty Read*, *Non-repeatable Read*, and *Phantom*. Berenson et al. described those Phenomena [16], which is summarized as following:

- **P1 (Dirty Read)**: Suppose one transaction T_1 modifies one data item. Another transaction T_2 then reads the data item before the time when T_1 actually COMMITS or ABORTS. If T_1 happens to execute a ROLLBACK, T_2 reads a data item aborted and never really existed, which obviously is incorrect.

- **P2 (Non-repeatable or Fuzzy Read)**: Transaction T_1 reads a data item x, and another transaction T_2 then updates or removes the same data item x and makes it commit. If T_1 tries to read the data item x again, it notices that the value of data item x is different from pervious read, or even worse, the data item x disappears in the database.

- **P3 (Phantom)**: Suppose transaction T_1 reads a set of data items based on some predicate C. Another transaction T_2 inserts one data item satisfying the predicate C, or change one data item from satisfying the predicate C to not satisfying the predicate C or vice versa, or delete one data item satisfying the predicate C, then commits. If T_1 re-reads with the same predicate C, the data set it gets is different from its first read on the predicate C.

Isolation Level	**Dirty Read(P1)**	**Fuzzy Read(P2)**	**Phantom(P3)**
Read Uncommitted	Possible	Possible	Possible
Read Committed	Impossible	Possible	Possible
Repeatable Read	Impossible	Impossible	Possible
Serializable	Impossible	Impossible	Impossible

Table 2.1: ANSI SQL Isolation Level Definition with Three Original Phenomena

All Phenomena listed above will not happen in Serializable, but could happen in a weak isolation level. Four isolation levels defined in ANSI-SQL Standard can be expressed to prohibit one or more Phenomena mentioned above and summarized in Table 2.1. To be specific, Read Uncommitted allows all Phenomena to happen; Read Committed proscribes Dirty Read (P_1); Repeatable Read prohibits both

Dirty Read (P_1) and Fuzzy Read (P_2); and SERIALIZABLE disallow all three Phenomena P_1- P_3.

Critiques on ANSI Isolation Definition. Berenson et al. criticized that definition introduced in ANSI-SQL Standard [10] was ambiguous as it may lead to a common misconception that prohibiting these three anomalies implies SERIALIZABLE. These Phenomena could be interpreted in either broad way or restrictive way. A broad interpretation means to exclude a larger set of execution histories than restrictive interpretation. However, even with broad interpretation, definition of isolation from ANSI-SQL Standard still didn't guarantee true SERIALIZABLE. To solve the ambiguity of definition in ANSI-SQL Standard, Berenson et al. proposed various isolation levels [16], which are defined by disallowing various Phenomena:

- P_0: $w_1[x]$ ... $w_2[x]$ ... (c_1 or a_1)
- P_1: $w_1[x]$ ... $r_2[x]$ ... (c_1 or a_1)
- P_2: $r_1[x]$ ... $w_2[x]$ ... (c_1 or a_1)
- P_3: $r_1[x]$ ... $w_2[y \text{ in } P]$ ... (c_1 or a_1)

Proscribing P_0 is to disallow transaction T_2 to update the data item x if an uncommitted transaction T_1 is updating the same data item x. Proscribing P_1 is to disallow transaction T_2 to read same uncommitted data item from transaction T_1. Proscribing P_2 is to prevent transaction T_2 reading modification on data item from an uncommitted transaction T_1. P_3 is related to predicates in transaction, which captures the Phantom Read problem. Based on that, Berenson et al. further defined different isolation levels with locks. The definition of isolation level with lock

is described in Table 2.2. As seen in the table, isolation level Read Uncommitted proscribes P_0; isolation level Read Committed proscribes P_0 and P_1; Repeatable Read proscribes all Phenomena that Read Committed prohibits plus P_2; and Serializable proscribes all Phenomena from P_0 to P_3.

Isolation Level	**Proscribed Phenomena**	**Read Locks**	**Write Locks**
Degree 0	none	none	short write locks
Degree 1	P0	none	long write locks
Degree 2	P0,P1	short read locks	long write locks
Repeatable Read	P0,P1,P2	long data-item read locks, short predicate read locks	long write locks
Degree 3	P0,P1,P2,P3	long read locks	long write locks

Table 2.2: ANSI SQL Isolation Level Definition Based on Locking

Graph Approach to ANSI Isolation Definition. With the definition, the four isolation levels are precisely described by avoiding four Phenomena mentioned in Table 2.2. However, Adya et al. reviewed these definitions of isolation provided by preventative approach, i.e., using locking approach, and criticized that the preventative approach is overly restrictive [13]. The preventative approach prohibits all histories that not allowed by concurrency control mechanism implemented by locking, and rules out optimistic and multi-version implementations, resulting in disallowing many legal histories that are proscribed by locking implementation but permitted by optimistic and multi-version implementations.

A simple counter example is that Phenomenon P_0 can happen in optimistic concurrency control as uncommitted transactions can access and modify the local copy of the same object concurrently. Serializable can be provided by allowing

some transactions to commit and abort the other transactions that modify the same objects. Therefore, prohibiting Phenomenon P_0 is overly restrictive to implementation adopting optimistic concurrency control.

The reason of overly restriction is that the isolation levels were defined based on lock scheme. Instead of using lock scheme, Adya et al. proposed a more general, i.e, implementation independent, approach to describe different isolation levels based on representing history of concurrent transactions based on graph [12, 13]. In this approach, *database* is modeled as a set of objects. More specifically, in relational database, each tuple or row is an object. The model of *transaction* is almost similar to previous definition, which is a list of read and write operations on objects in a total order that the operations appears in the transaction. Unlike previous definition, each object is associated a *version number* which is related to operations from transactions: when a transaction, e.g., T_i, writes an object x, it creates a new version m of x, e.g., $x_{i.m}$, and when a transaction reads an object x, it reads a version of x created by other transactions previously. A *history*, capturing what happens in an execution of a database, contains two parts: a partial order of events that reflects all operations (including read, write, commit and abort) of those transactions; and a total order on the version number for each object.

The graph approach to defining ANSI Isolation is to represent *committed* transactions and their relations in a directed graph, which is called Direct Serialization Graph (DSG). A DSG reflects a specific *history* H running a set of transactions, i.e., DSG(H), in which each transaction is a node, and edges are the *conflicts* between two different transactions from the transaction set. In this approach, the *conflicts*, or the

edges in the graph, can be classified into three categories based on operations and orders of source and destination.

- **Direct Read Dependency** (wr edge). A transaction T_j *directly read depends* on another transaction T_i if either of following two conditions is satisfied:
  - T_i installs some version of object x_i, and T_j reads x_i (item-read dependency).
  - T_j performs a read based on some predicate *Pred.*, which includes the object x that the installation of new version by T_i changes the result of predicates *Pred.* (predicate-read dependency).

  If transaction T_i and T_j have direct read dependency in *history* H, an edge $T_i \xrightarrow{wr} T_j$ will be added to the DSG(H).

- **Direct Anti-Dependency** (rw edge). A transaction T_j *directly anti-depends* on another transaction T_i if either of following two conditions is satisfied:
  - T_i reads some object version x_k, and T_j installs the next version of object x (item-anti-dependency).
  - T_i performs a read based on some predicate *Pred.*, and T_j installs a newer version of object x, which changes the matching result of predicate *Pred.* (predicate-anti-dependency).

  If transaction T_i and T_j have direct anti-dependency in *history* H, an edge $T_i \xrightarrow{rw} T_j$ will be added to the DSG(H).

- **Direct Write Dependency** (ww edge). A transaction T_j *directly write depends* on another transaction T_i if T_i installs a version x_i and T_j installs the next

version of the object x. If transaction T_i and T_j have direct write dependency in *history* H, an edge $T_i \xrightarrow{ww} T_j$ will be added to the DSG(H).

In general, we say T_j conflicts with T_i, or T_j depends on T_i, if there's one or more direct dependency from T_i to T_j, which shows $T_j \leftarrow T_j$ in Direct Serialization Graph. Based on Direct Serialization Graph, Adya et al. proposed new Phenomena to define isolation levels free from implementation. The new definition on isolation levels was based on proscribing certain cycles in the Direct Serialization Graph:

- Read Uncommitted proscribes any transaction histories that the corresponding Direct Serialization Graph consists of cycles composed of only write dependency edges.
- Read Committed proscribes any transaction histories that the corresponding Direct Serialization Graph consists of cycles formed only with read dependency edges and write dependency edges.
- Repeatable Read proscribes any transaction histories that the corresponding Direct Serialization Graph consists of cycles that only contains read dependency edges, write dependency edges and item anti-dependency edges.
- Serializable proscribes any transaction histories that the corresponding Direct Serialization Graph consists of any cycles.

In this thesis, we often use alternative terms to these edges. In DSG, edges representing write dependency conflict are also called ww edges; edges for read dependency conflict are called wr edges; edges with anti-dependency conflict are equal to rw edges. We will use these terms interchangeably in the rest of thesis.

Monotonic Atomic View. Monotonic Atomic View is a weak isolation level related to Read Committed but with DBMS using lock-based implementation via long write-locks and short read-locks. Here I would like to clarify the difference between these two isolation levels. Both transaction level prohibit anomalies *Dirty Writes* and *Dirty Reads*, which are described above. However, comparing to Read Committed, Monotonic Atomic View proscribes one more phenomenon called Observed Transaction Vanishes (OTV) described in several publications [14, 15]. The anomaly of OTV can be summarized as one transaction observes part of updates from another transaction while missing the other part. Suppose we have transaction $T_1 : w_{1.1}(x), w_{1.2}(y)$, and $T_2 : r_{2.1}(x), r_{2.2}(y)$. It's possible that $r_{2.1}(x)$ observes the update performed by $w_{1.1}(x)$, while $r_{2.2}(y)$ misses the update of $w_{1.2}(y)$. This is allowed by Read Committed as it allows any cycle with rw edges, however, this is impossible for database with lock based implementation: if $r_{2.1}(x)$ observes the update from $w_{1.1}(x)$, the following read should also read the committed result. Monotonic Atomic View is also defined as PL-2L in Adya's work [12], which is an intermediate weak isolation between Read Committed and Repeatable Read.

Snapshot Isolation. Snapshot Isolation was first introduced by Berenson et al. [16], which provides a description about Snapshot Isolation on the operational implementation: a transaction T_i running under Snapshot Isolation reads committed data at the beginning of the transaction, which is called *start-timestamp* $t_{i-start}$. Modifications on the item read by the transaction T_i after $t_{i-start}$ are not visible to T_i. When the transaction T_i is about to commit, it issues a *commit-timestamp* $t_{i-commit}$ if there's no other concurrent transactions committed with updates overlapping with

the write set of T_i, or it will abort. It is also called *First-committer-wins* rule on proscribing lost updates.

Although the *First-committer-wins* rule provides an intuitive description that captures the problem Snapshot Isolation tries to solve, the description is related to implementation and lacks of generalization on graph approach. Ayda et al. proposed a general definition of Snapshot Isolation in graph approach in [12, 13] with *time-precedes order*, which is a partial order $\prec_t$ on the start timestamp $t_{i-start}$ and commit timestamp $t_{i-commit}$ for transaction T_i with two rules:

1) $t_{i-start} \prec_t t_{i-commit}$, which implies commit always happens after start of the same transaction;

2) Either $T_{i-start} \prec_t T_{j-commit}$ or $T_{j-commit} \prec_t T_{i-start}$ on transaction T_i and T_j, which describes that the start and commit timestamps on different transactions have partial orders defined by the time transactions start and commit.

With start dependency, Snapshot Isolation can be described as an isolation level that proscribes following phenomenon combined: 1) read/write dependency edges without start dependency edges; 2) the execution history contains one cycle with exact one anti-dependency edge. Checking if one execution history is allowed by Snapshot Isolation then translates into the problem identifying if the corresponding dependency serialization graph doesn't contain above two patterns above-mentioned. Later, Fekete et al. revisited the definition of Snapshot Isolation [27], and gave a more practical definition that Snapshot Isolation only allows cycles with two or more consecutive anti-dependency edges. The revised definition removes the requirement of *start dependency* on reasoning anomalies under Snapshot Isolation, bringing the light upon unifying graph approach used on other isolation level.

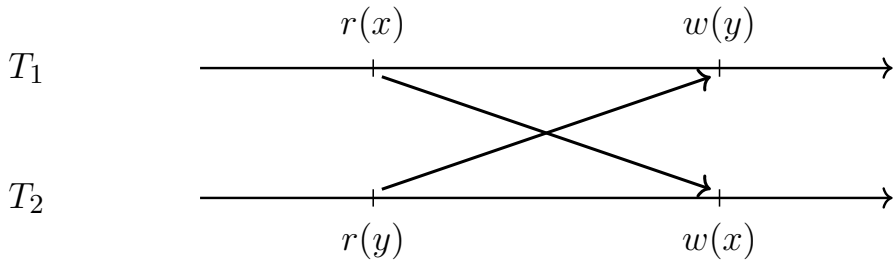


Figure 2.1: *Write Skew* Anomaly under Snapshot Isolation

With *First-committer-wins* rule, Snapshot Isolation naturally prohibits anomalies like Lost Update, but it may not prescribe other anomalies. One example is *write skew* anomaly. Suppose we have two data item x and y in the database with initial value both set to 0, and we have two transactions: both transactions try to make the sum of x and y equal to 1, but transaction T_1 checks on x and modifies y, whereas transaction T_2 checks on y and modifies x. Like the example illustrated in Figure 2.1, it's possible that both transaction T_1 and T_2 read 0 on the object they check, and increment the counterpart by 1, resulting in the sum of x and y equal to 2, thus violating the constraints. Such problem cannot happen under isolation level Repeatable Read, as it clearly show a cycle contains item anti-dependency edges (rw edges) in the Direct Serialization Graph, thus Repeatable Read prohibits such anomaly.

Isolation from Client View. A number of recent works [22, 49] attempt to make such definition more developer friendly by defining isolation from the external view. For example, Crooks et. al. [22] define Snapshot Isolation as a transaction T's operations must read from the same state s, which is the state that T transitions from.

The state-based specification of isolation level naturally maps to what clients are able to observe and makes it feasible to compare different implementation of isolation guarantees. Szekeres et. al. [49] proposed a unified model on ordering guarantees via a single operation relationship, *result visibility*, to formulate isolation guarantees.

2.3 Isolation Level in Popular Database

Before discussing what isolation levels are used in popular database system, we should first look at what are the popular database systems. Searching from DB Ranks [23], the top four are Oracle DB [40], MySQL [38], Microsoft SQL Server [47] and PostgreSQL [44]. We will discuss one by one.

Oracle DB. In the document from Oracle Database [41], Oracle Database supports Read Committed, Serializable and Read-Only Isolation level which behaves in a similar way to Serializable but only for read transactions. However, the actual isolation level supported by Oracle DB is different from what they claimed. Read Committed should be Monotonic Atomic View (MAV) or PL-2L [12]. This can be regarded as a lock based implementation of Read Committed, which guarantees the one transaction T_1 won't miss effect from another transaction T_2 once it observes the version of objects installed by T_2.

As for the other isolation, Serializable, specified in Oracle DB documentation, actually performs at the level of Snapshot Isolation. We can test it by triggering a *Write Skew* phenomenon and observe inconsistent state after concurrently running two transactions.

MySQL. MySQL is a popular database from open source community [38]. Isolation level supported by MySQL includes Read Uncommitted, Read Committed, Repeatable Read, Serializable. The default isolation level is Repeatable Read. For Read Uncommitted and Serializable, the guarantees provided by MySQL is same as what these isolation levels are defined. However, Read Committed and Repeatable Read are not the same as specified. For Read Committed, it's similar to the Read Committed in Oracle DB, which is a lock based implementation of Read Committed, and is slightly stronger than Read Committed defined in graph approach. As for Repeatable Read, it's actually weaker than the Repeatable Read defined in graph approach, as it allows *Lost Update* [39].

Microsoft SQL Server. Microsoft SQL Server is a relational database product from Microsoft. It supports Read Uncommitted, Read Committed via lock scheme, Read Committed via snapshot, Repeatable Read, Snapshot Isolation and Serializable. Like previous produce, Read Committed is stronger than the isolation level with same name but defined using graph approach, and it can be viewed as PL-2L defined in Adya's work [12].

PostgreSQL. PostgreSQL is an open source project of relational database management system. PostgreSQL adopts multi-version concurrency control (MVCC) to support concurrent transactions. It supports Read Committed, Repeatable Read and Serializable, but it has similar naming issue with isolation level like other DBMS. The actual isolation level for Repeatable Read is Snapshot Isolation in PostgreSQL, and Read Committed is actually Monotonic Atomic View.

2.4 DB Concurrency Control vs. Transaction Isolation Level

In database management system (DBMS), *Concurrency Control* refers to the concept and solution that the DBMS guarantees the integrity of data carefully while processing transactions concurrently. More specifically, concurrency control provides an environment that one transaction can execute as if it's the only transaction interacting with the database while actually there are multiple transaction running concurrently.

Based on the philosophy of handling concurrent transactions, concurrency control can be categorized into many approaches based on different criteria, and three popular approaches are: *Pessimistic*, *Optimistic* or *Multi-version*. The first two approach could be regarded as two extremes on opposite and the third category tries to mitigate the problem of contention among concurrent running transactions with different views.

The first approach is pessimistic concurrency control. Just as the name implying, in this type of concurrency control, the DBMS assumes pessimistically that concurrently running transactions affect each other, so that it would block some transactions until other transactions finish. One example of pessimistic concurrency control is using lock scheme in concurrency control. When using lock scheme to implement concurrency control for database management, the DBMS issues lock on the objects, which could be column, row, table, or even one database, that one transaction intends to access, at the beginning of the transaction. The lock will be held during the life time of the transaction, and will be released after the transaction commits. For example, when one transaction T_1 accesses some objects, i.e., obtaining locks for the objects, other transaction T_2 intending to access the same or a subset of the objects will try to grab the locks of those already locked objects, hence will be blocked from

accessing those objects. After the transaction T_1, which holds the locks of the objects, finishes execution and commits its update to the database, it releases the obtained locks, then the transaction T_2 can grab the locks and execute its transaction. In some transactions, it involves multiple objects to process and keep sync. If using lock scheme for multi-object transactions, the transaction could be roughly divided into two phases in terms of its action with locks. In the first phase, it continues to grab locks for all objects it needs to process the transaction; and in the second phase, it executes the transaction logic and releases all the locks. Some deadlock detection or prevention mechanisms are required for such multi-object transaction, like Banker's Algorithm introduced in the work [25].

The second approach is optimistic concurrency control (OCC). In this approach, concurrent running transactions are handled with an optimistic hope that concurrent running transactions rarely access same objects so that it's very unlikely that transactions running concurrently actually affect each other. The idea of optimistic concurrency control was proposed in an early work [33], and has been implemented in multiple database systems [35, 54, 64]. In the nutshell, optimistic concurrency control works in following steps. When processing transaction with OCC, the DBMS first generates the read set and write set that the transactions require access to. After retrieving the read set and write set, the DBMS analyzes the read set and write set, and decides the transaction to be committed or aborted based on analysis. To analyze read set and write set, the DBMS checks if there are other transaction also accessing some objects in the read set and write set: if so, the transaction will be aborted; otherwise, it will be committed. The optimistic approach outperforms the pessimistic approach when transactions have low contention as the optimistic approach

won't over restrictively prevent the system running current transactions as long as they don't actually cause contention. The pessimistic and optimistic approach can be combined into same DBMS to handle different workloads, which is referred as a federated approach to improve performance of DBMS [48, 58].

The third approach is multi-version concurrency control (MVCC) [17]. Unlike previous two approaches, this approach views objects in database with version number, and allows the same object to exist with different version number. Transactions are assigned with specific identity, i.e., version, which provides the order of transactions. Based on the version, one transaction can only observe the objects with committed version, which is smaller than the version assigned to the transaction, even though there is a later version updated by other transaction. Transactions that only execute read operations only read from a previously committed version (or called snapshot) from database hence it won't be blocked by updates from other transaction. On the other hand, transactions doing write still need to be carefully handled to prevent bad effects like *Lost Update*, and it can use either pessimistic approach (lock scheme) or optimistic approach (first commit wins) to solve the problem. The implementation of MVCC usually involves two approaches aforementioned but extends the DBMS with flexibility to manage concurrent transactions and their orders at the cost of handling decomposing dangling objects with obsolete version number, which is called garbage collection.

So far, we have discussed several concurrency control mechanisms. Actually, it is quite tightly connected to the concept of *Isolation*, which is one of the ACID properties for transaction. The concept transaction was proposed to simplify the work of interacting with shared system in the multi-user environment as it provides

a guarantee that the programmer can encapsulate the programming logic into one block and assume the block can run safely even in the concurrent system. The ACID property is the requirement to the DBMS that programming logic block running on DBMS should guarantee the block will run as unbroken piece. In ACID, *isolation* is the guarantee that multiple transactions can be safely isolated while running concurrently. Thus, *isolation* is a promise provided by DBMS that the transactions running on the system guaranteed to avoid some undesirable result based on the *isolation level*, whereas *concurrency control* is the implementation of such guarantee in DBMS.

Here's an example of the relation between *isolation* and concurrency control as we just mentioned. Most of concurrency control mechanisms are designed to make transaction run at the isolation level SERIALIZABLE, which ease developers' design on handling concurrency issue. Adya et al. provides the definition of isolation [12, 13] based on the graph (Direct Serialization Graph, or DSG) which is generated from transaction history containing the order of running transaction and the version order of related objects. For isolation level SERIALIZABLE, it prohibits any cycle on the DSG. In other word, the history is not SERIALIZABLE if the DSG of the history contains any cycle. A concurrency control mechanism can be designed according to the statement. Taking OCC as example, it needs to validate the transaction and decide whether to commit or abort it after getting the read set and write set. By tracking the activity of all transactions in DBMS and accumulating transaction dependencies, the DBMS could formulate a DSG of current execution history, try to add the transaction under check into the DSG, and check if it causes a cycle in the DSG.

In summary, the relationship between *isolation* and *concurrency control* is that both concepts are related to the concurrency issue in database, in which *isolation*

is defined as guidance providing *what* the concurrency issue is, whereas *concurrency control* provides the solution of *how* to handle the concurrency issue.

2.5 Anomaly Analysis on Weak Isolation

Though (strict) SERIALIZABLE is usually preferred by developers and is used in many research works (including but not limited to [36, 37, 48, 51, 53, 57, 58, 60, 61]), a number of studies have shown that READ COMMITTED and SNAPSHOT ISOLATION are the most widely used isolation levels in practice [22, 42].

Unlike SERIALIZABLE, which guarantees to serialize concurrent transactions as if running them one by one in certain order, weak isolation levels allow some undesirable Phenomena which could lead to unexpected results developers unwilling to see, and sometimes the anomalies brought by weak isolation is too vital to ignore for the database-backend applications.

```
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 2.2: Logic of a purchase transaction.

Example of finding anomalies. 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 2.2) to purchase an item and suppose multiple purchase transactions can be executed concurrently under isolation level READ COMMITTED. READ COMMITTED only guarantees a read operation will

retrieve a committed value, which may cause problems for the example in Figure 2.2: 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, that analyzing anomalies via modeling transaction execution history by *Direct Serialization Graph* (DSG), proposed by Adya et. al's definition of isolation levels [13], provides opportunities to view the problem of analyzing anomalies as to check if certain cycle pattern exists in the DSG of the given transaction history.

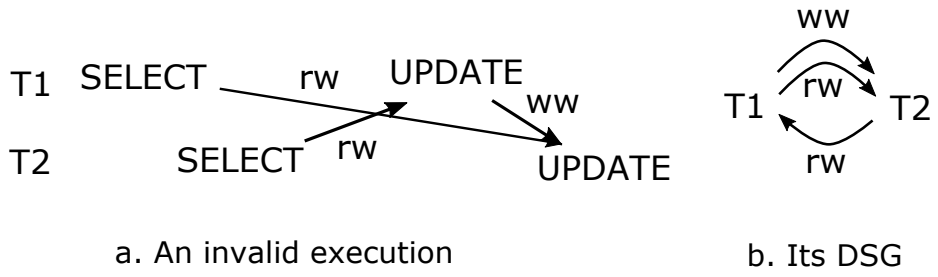


Figure 2.3: Identifying anomalies using the DSG.

Consider the example in Figure 2.2: Figure 2.3.a presents the problematic execution mentioned above, which can be converted to the DSG in Figure 2.3.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.

Related Work. Fekete et al. [27] showed that TPC-C [52], a prominent benchmark for OLTP[1] transactions, actually should work well under SNAPSHOT ISOLATION. A following work [30] of [27], proposed by Jorwekar et al., attempted to check general database application with the definition of SNAPSHOT ISOLATION, and determined whether the database application was safe to run under SNAPSHOT ISOLATION. The refined definition of isolation level SNAPSHOT ISOLATION can also be used for concurrency control in the implementation of multi-versioned database. Cahill et al. [18] proposed a modification to the concurrency control algorithm in database to automatically detect and prevent anomalies allowed by SNAPSHOT ISOLATION, and made it possible to run transactions under SNAPSHOT ISOLATION while enjoying the guarantee of SERIALIZABLE.

ACIDRain [55] 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 [26].

Elle [31] targets on checking if black-boxed databases provide the isolation guarantees they claim, by carefully selecting transactions sent from client side and verifying if the concurrent execution results on the target database are compliant to Adya's formalism [13]. This work is one part of a project, Jepsen [11], which is a software library on testing safety properties of distributed systems.

[1]OLTP stands for Online Transaction Processing.

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, and seek for opportunities to improve the precision of weak isolation anomalies analysis against database applications by considering other factors like application specification and feedback from developers.

Chapter 3: Design of IsoDiff

In this chapter, we will describe the design of IsoDiff, a tool aiming to find anomalies under weak isolation for database application without triggering those anomalies in the runtime. First, we will introduce the model for analysis and related definitions; then we will discuss how to convert a real database application to the model for further analysis; after that, we will show the detail of how IsoDiff analyze the converted model and output *potential* anomalies with transaction execution history to trigger those anomalies.

3.1 Model and Definitions

Formally, this work 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 thesis.

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 [30, 55], 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 3.3 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) [13].

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 $op_i \to op_j$ and $op_j \to op_i$ in G_{op} and between the corresponding transaction classes in G_T. Section 3.4 presents the details.

Definition 3.5. *Δcycle. A Δcycle 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 Δcycle should contain at least one rw edge; when $\mathcal{I}$ is* Snapshot Isolation, *a Δcycle 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 $\mathcal{L}$
input : Database isolation level $\mathcal{I}$
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 Δcycle for Read Committed is derived from [13], and the definition of Δcycle for Snapshot Isolation is derived from [27]. At a high level,

the major goal of IsoDiff is to search Δcycles in the transaction dependency graph, because they represent anomalies that can happen under $\mathcal{I}$ but cannot happen under SERIALIZABLE.

3.2 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 2.2 as an example.

3.3 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 [30, 55], 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) [27]: 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 [27] manually "split" the corresponding transaction class if an application function has internal if branches, which is what IsoDiff achieves automatically.

Example. Using Figure 2.2 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.

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⟩
```

3.4 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).

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 $T_1 \to ... \to T_2$, then C_3 must contain the rw edge; if the rw edge appears in $T_2 \to ... \to T_3$, 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 $A \to T_2$: if we break C_2 by removing an edge which is not $A \to T_2$, then of course we break C_1 as well since this edge exists in C_1; if we break C_2 by removing $A \to T_2$, since $A \to T_2$ is similar to $A \to T_1$, doing so will remove $A \to T_1$ 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 $T_i \to ... \to T_{i+1}$, 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.

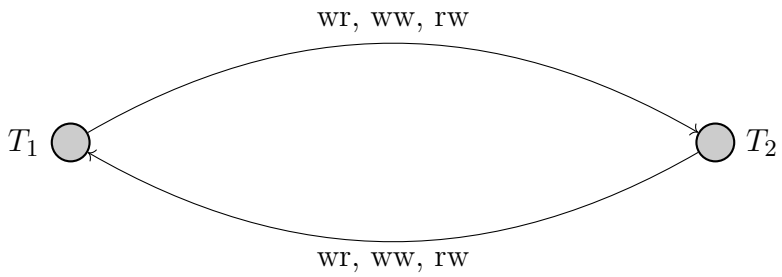


Figure 3.1: Transaction Dependency Graph

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)

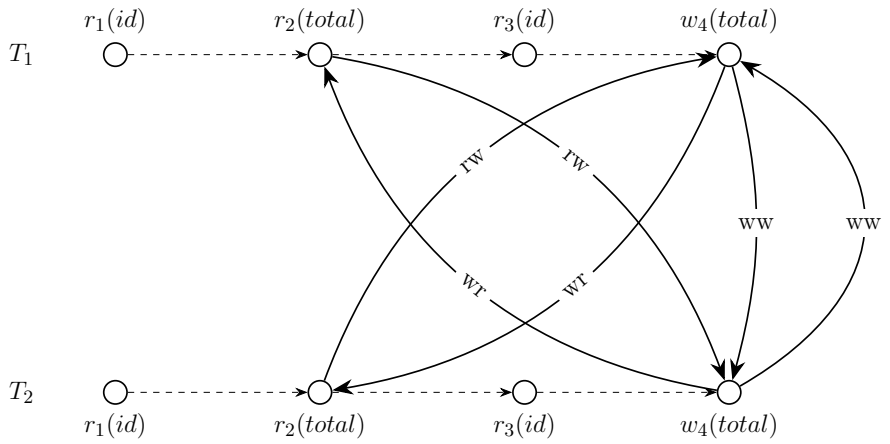


Figure 3.2: Operation Dependency Graph

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.

Example. Taking Figure 2.2 as the example, for Read Committed, IsoDiff generates G_T as shown in Figure 3.1 and G_{op} as shown in Figure 3.2. 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.

3.5 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 [27] 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 3.7). 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.

Algorithm 3: Analyze correlations in transactions

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

$\mathcal{C} \leftarrow \emptyset$; $Correlation \leftarrow \emptyset$
for *transaction class* $T \in \mathbb{T}$ **do**
  for *operation* $op_i, op_j \in T$ **do**
    if *for any instance of* T, op_i *and* op_j *access the same row* **then**
      Add pairs (op_i, op_j) and (op_j, op_i) to $\mathcal{C}$;
for *each dependency edge* $op_k \leftrightarrow op_m$ **do**
  for op_x*:* $(op_x, op_k) \in \mathcal{C}$ **do**
    for op_y*:* $(op_y, op_m) \in \mathcal{C}$ **do**
      Add ($op_x \leftrightarrow op_y$, $op_k \leftrightarrow op_m$) to $Correlation$
return $Correlation$

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 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 [30]. Such edges are later used for searching Δ cycles for Snapshot Isolation, which should contain two consecutive vulnerable rw edges. Section 3.7 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" [27, 30], IsoDiff's solution is more general and

can detect correlations through more patterns like accessing different tables with the same key.

Example. Taking Figure 2.2 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.

3.6 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.

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.

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

$\Delta_{cycle} \leftarrow \emptyset$
for *each dangerous path dp in* (G_{op}, $\mathcal{C}$) **do**
    $T_{src} = dp.src$ in G_T
    $T_{dst} = dp.dst$ in G_T
    $\mathcal{P}_T \leftarrow$ `k-shortest-path`(k, $[T_{dst}, T_{src}]$, G_T)
    if $\mathcal{P}_T = \emptyset$ **then**
        continue
    for *each* $path_T$ *in* $\mathcal{P}_T$ **do**
        for *each* $path_{op}$ *in* $MapPath_{T \to op}$($path_T$, l) **do**
            $cycle_{op} \leftarrow path_{op} \cup dp$
            for *each pair of* op_i *and* op_j *in* $cycle_{op}$ **do**
                if op_i *is ordered before* op_j *in the same transaction class* **then**
                    add $op_i \rightarrow op_j$ to $cycle_{op}$
            if $cycle_{op}$ *is valid* **then**
                $\Delta_{cycle} \leftarrow \Delta_{cycle} \cup cycle_{op}$
return Δ_{cycle}

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 [59] 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.

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 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

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
```

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 3.1, 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).

3.7 Validating Δ cycles

Unlike the Adya et. al's work [13], which generates a dependency graph from a real execution, our work inherits [27]'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

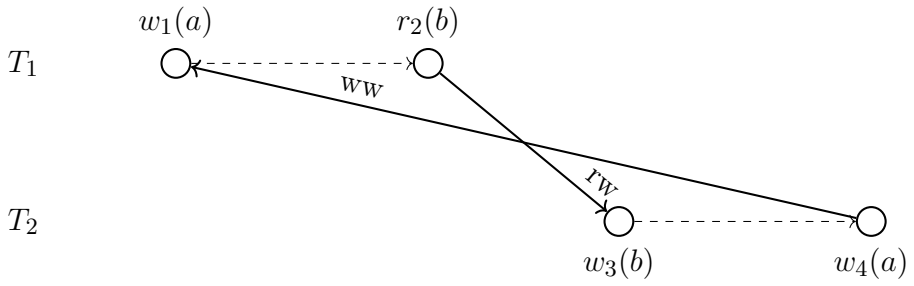


Figure 3.3: Example of timing violation.

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 3.3: 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

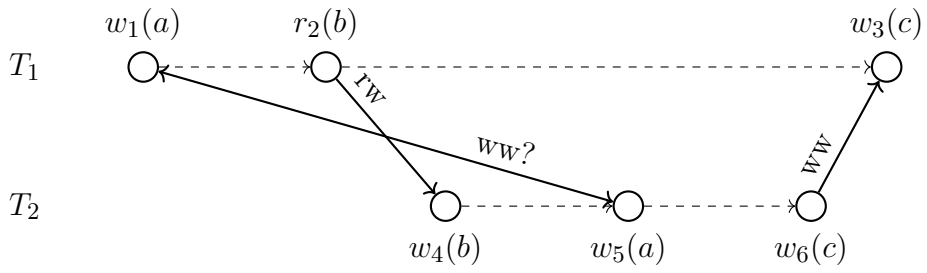


Figure 3.4: Example of a Δ cycle invalidated by correlation.

constraint. In Figure 3.3, 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 3.5, 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 3.4. 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 $w_5(a) \rightarrow w_1(a)$ will cause a timing violation, and choosing $w_1(a) \rightarrow w_5(a)$ 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 [27,30] 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 3.5. 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).

3.8 Eliminate found Δcycles (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 work.

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 [20]. 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 2.2, 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.

3.9 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 [30, 32, 55, 62].

False positives: As discussed in Section 3.7, 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.

Chapter 4: Implementation

4.1 Parsing Database Trace

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 [52], 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.

4.2 Implementation 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.

4.3 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 3.6) 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.

Chapter 5: Evaluation on IsoDiff

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 [52] 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.

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

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 [55]; 3) for attendize and fa, we manually trigger different functions of these applications.

Table 5.1 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 [21]. 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 5.2 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 work.

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 2.2, 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.

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

Table 5.2: Number of $TargetColumns$ IsoDiff finds for each application (RC=Read Committed; SI=Snapshot Isolation).

- 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 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 developer 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 [16]: 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* 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 5.3 summarizes how they affect the result of IsoDiff. As one can see, for some applications the results vary

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

Table 5.3: Min/Max number of $TargetColumns$ with different k and l (RC=Read Committed; SI=Snapshot Isolation).

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 5.1 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.

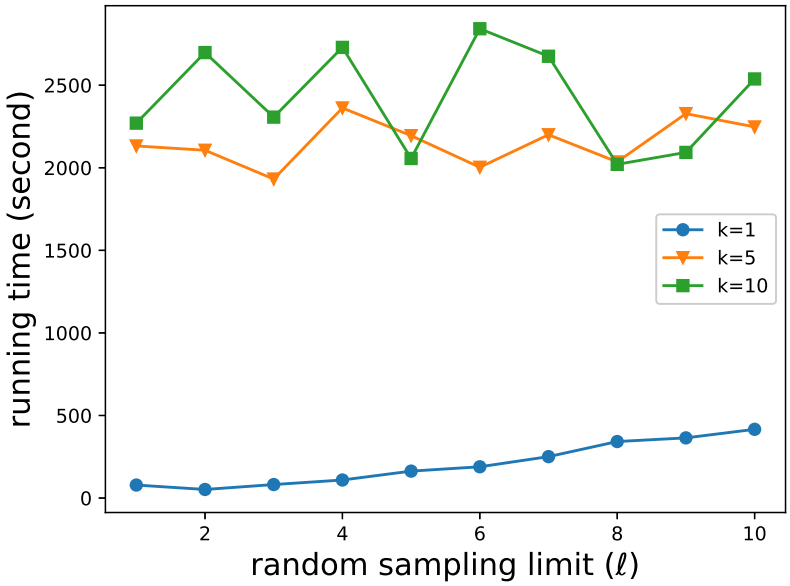


Figure 5.1: Running time of IsoDiff on *woocommerce (wc)* with different settings.

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 [19, 29, 34, 45, 46, 56, 63, 65], 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.

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 5.4: 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.

Table 5.4 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

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%

Table 5.5: The coverage of Δ cycles with correlated edges.

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 5.5, the coverage is at least 96% with some of them reaching 100%.

Chapter 6: Improving performance on IsoDiff

The performance of IsoDiff is one of the major bottlenecks affecting the extensibility of the checker on other applications at large scale. To improve the performance of IsoDiff, we start from improving path search algorithm on transaction level graph, then optimize data structure used in operation level cycle detection, and introduce a solution to randomly yet deterministically compare the improvement over the changes. In the last part of this section, we will give evaluation on the changes.

6.1 Improve with Efficient Algorithm and Data Structure

To start performance optimization, we measure running time of the first iteration on each part of IsoDiff, which includes searching dangerous path, searching k shortest path in transaction level graph, transforming cycles in transaction level into cycles in operation level, and validating cycles in operation level graph. Since most of the parts in IsoDiff run in multi-threads, we measure the running time of each part on every thread and sum up the total running time from all threads for each part. As for the parts running in single thread, it merely contributes a negligible portion of the total running time.

The first part we try to improve is the k shortest path search algorithm. On the one hand, we observe that, for some applications, it takes up to 50% of time on

searching paths on transaction level graph. On the other hand, it is expected that the searching time grows as the number of shortest path (i.e., the k in k shortest path) increases when developers want to explore more anomalies, so it is necessary to explore algorithms that are more efficient for k shortest path search.

In IsoDiff, we apply Yen's K Shortest Path algorithm [59] to find paths with which IsoDiff explores possible anomalies. The K Shortest Path algorithm looks for those paths with *least weight*, and generates result of K paths ranking in the order of least weight given inputs of source and destination node in a directed graph. The search algorithm works well for the requirement of IsoDiff, which prefers the least possibly complex anomalies on the given application, and is able to extend to more anomalies controlled by parameters. Nevertheless, there is still some space to improve the performance of IsoDiff. One observation is that, in IsoDiff, edges in the graph used for path search represent the dependency between different transactions and don't have precedence, so the graph is actually an *unweighted* graph. In most of the algorithms searching for shortest paths with weight, they maintain a heap data structure to rank the paths already searched and prioritize the paths with lower weight since it's more likely to get result from those paths. However, it's unnecessary to maintain such heap data structure if the graph is unweighted. We follow the similar idea of Dijkstra Shortest Path algorithm [24] while replacing the priority queue with a FIFO queue. The new algorithm pushes the source node into the queue, and continues searching path in the graph. Once the searched path contains the node which is the same as the destination node given in the input, it adds the path to the return result.

Another improvement is to incorporate loopy graphs for transaction level graph search in IsoDiff. Remind that, in IsoDiff, each transaction extracted from SQL

traces is duplicated twice or three times depending on the target isolation level, Read Committed or Snapshot Isolation respectively. Alternatively, to avoid the duplication, which we observed in the implementation of IsoDiff with loopless graph, we can use self loop edges to represent one transaction having conflicts with another transaction with the same type, and use one node to represent one type of transaction in the transaction level graph. This method was also adopted in other work [27, 30] to represent conflict graph. The benefit of the replacement is on twofold: by replacing loopless graph with loopy graph, the size of graph reduces thus the KSP search space reduces as well; the KSP algorithm with loopy graph automatically removes the redundant result that the paths explored are actually different but exactly the same if only considering their types of transaction. To implement graph search for the loopy graph in IsoDiff, we add two additional steps. In the first step, which is the initialization phase, IsoDiff builds a loopy graph corresponding to the original loopless graph, and in the second step, which is part of the execution phase, IsoDiff converts input and output for loopy graph back and forth as the validation phase requires Δcycles to be represented in the loopless form.

Besides improvement on the graph search, we try to improve the performance of operation level cycle validation, which prunes all the false positive cases generated during graph search. In cycle validation, each generated cycle is required to be inspected with causality check which detects whether there's a cycle in the operation level graph. By using Linux profiling tool *perf* [43], we find that the major bottleneck is on the timing violation check (i.e., cycle detection on operation level graph). Therefore, we replace the graph data structure with bitmaps for efficiently setting/resetting and checking graphs.

6.2 Compare Result Deterministically While Randomly

After making changes on IsoDiff, we need to evaluate the effectiveness of these changes. However, the comparison is not such a trivial step as comparing the running time of experiment running the version of IsoDiff before those changes along with experiment running the version of IsoDiff with patch applied. Since IsoDiff runs on multi-threads and randomly chooses one of the operation level edges corresponding to the transaction level edge, it's unfair to compare running time of two versions in consideration of the randomness. To solve the problem, we add a random seed into inputs to IsoDiff, and feed the seed into one random number generator (RNG) in the main thread. Considering that IsoDiff uses multiple threads to fully utilize CPU resource for transaction anomaly exploration, using single random number generator forbids multiple threads running deterministically as it's hard to control the thread scheduling which is managed by the underlying operating system.

Since the source of randomness is not only from program logic (i.e., picking random operation edges) but also the operating system (i.e., thread scheduling), we need to consider an approach to running IsoDiff deterministically with one seed regardless of thread scheduling. We notice that the main thread dispatches dangerous paths to worker threads, and the worker threads explore anomalies sorely based on the dangerous path assigned to the threads. For each dangerous path, we pass one random number generated from the RNG in the main thread to the worker threads. The worker threads will use the random number as seed to initialize the RNG in the worker thread, and that RNG will be the randomness source for the procedure exploring Δcycles associated with the dangerous path. Since the random seed associated to each

dangerous path is deterministic, IsoDiff can stably generate same result in spite of thread scheduling.

With providing fixed random seed to IsoDiff and assigning random seeds generated from the fixed random seed on each dangerous path, it is guaranteed that the execution of IsoDiff is fully deterministic upon the random seed in the main thread. However, this is not enough to evaluate between the original IsoDiff and the implementation with performance improvement. The reason is that, as we mentioned previously, the two version of IsoDiff searches different cycles as the graph data structures used for path search are different. To make fair comparison on the two version, we separate the path exploration and cycle validation into two parts for evaluation. To evaluate it, IsoDiff first generates all the paths by running the path exploration related functions and dumps them into a file. In another round, IsoDiff reads the file and validates cycles from the file generated from first round.

6.3 Evaluation on Improving Performance

To measure the performance improvement aforementioned, we will evaluate the improvements on IsoDiff as following:

- The improvement on k shortest path algorithm on path exploration in transaction level graph.
- The runtime stability on the same random seed with deterministic randomness integrated into IsoDiff.

Improvement with Loopy Graph. To evaluate the performance improvement on changing graph data structure and search algorithm, we run experiments on

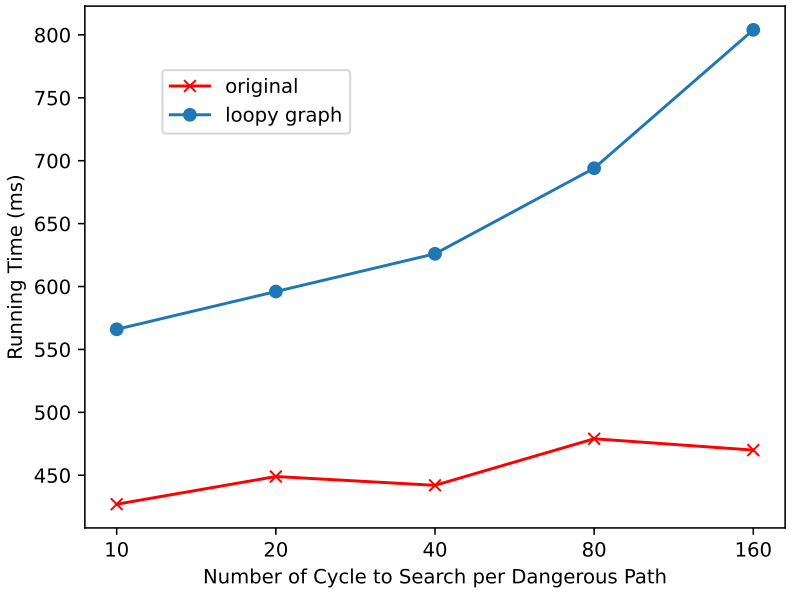


Figure 6.1: Performance Comparison of Transaction Cycle Search with Application *attendize*

some application changing the number of Δcycles required for each dangerous path without validating those cycles. The experiment results are shown as in Figure 6.1, 6.2 and 6.3. With required number of Δcycles per dangerous path increasing, the running time increases as well. The result shows that, for larger graph (e.g., Figure 6.2 and Figure 6.3), which requires longer time to search, using loopy graph can decrease the total running time spent on graph search. If the searching time is already short enough, using loopy graph may perform worse as it introduces more complexity on the building the loopy graph.

Stability with Deterministic Random Seed. Since the result generated by IsoDiff is numerous, we introduce deterministic random seed in IsoDiff to ensure that it can reproduce the same result controlled with the same random seed. If using the same

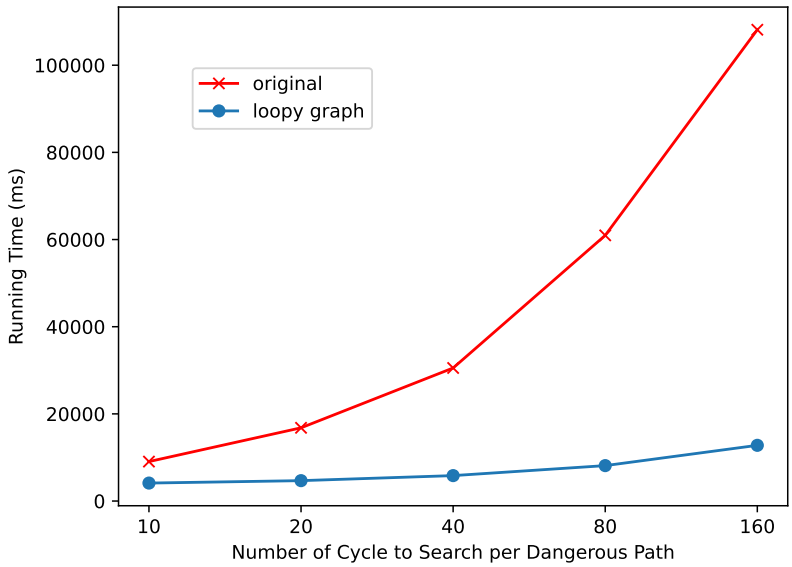


Figure 6.2: Performance Comparison of Transaction Cycle Search with Application *broadleaf*

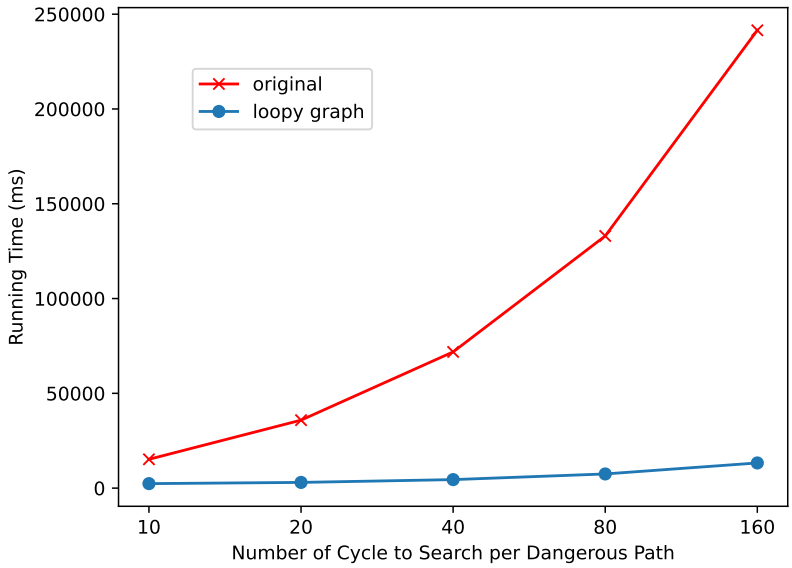


Figure 6.3: Performance Comparison of Transaction Cycle Search with Application *spree*

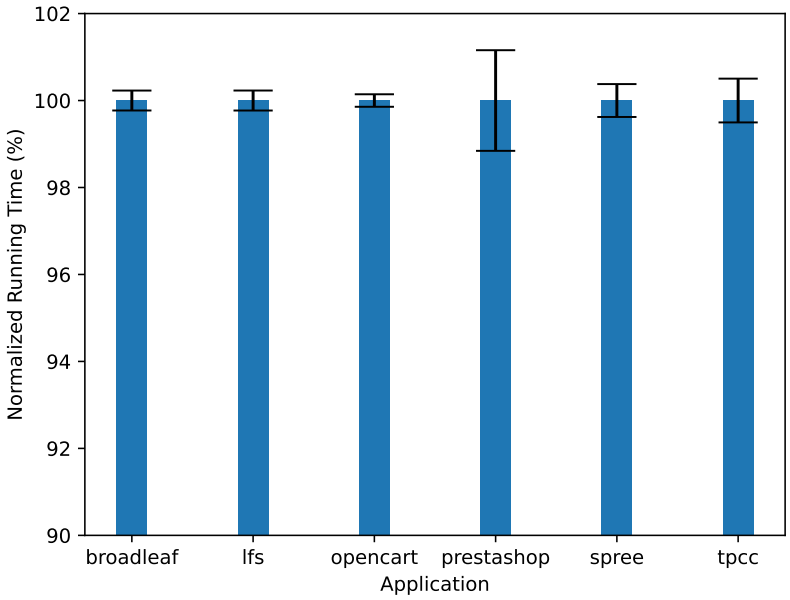


Figure 6.4: Normalized Running Time with Repeating Applications 5 Times with the Same Random Setting per Application

random seed, the result of IsoDiff is deterministic regardless of concurrent execution. We randomly select some applications and run them multiple times with same setting (not necessarily the same for different applications), and collect the running time of each test experiment. The result is shown in the Figure 6.4. We normalize the running time as percentage and set the average running time of repeated experiment with the same experiment setting on the application as 100%, and show the error bar in the same Figure. The result shows that the running time is stable for the same experiment setting, which includes number of Δcycle candidates per dangerous path, number of retry on validation of one Δcycle candidate, isolation level and a fixed random seed.

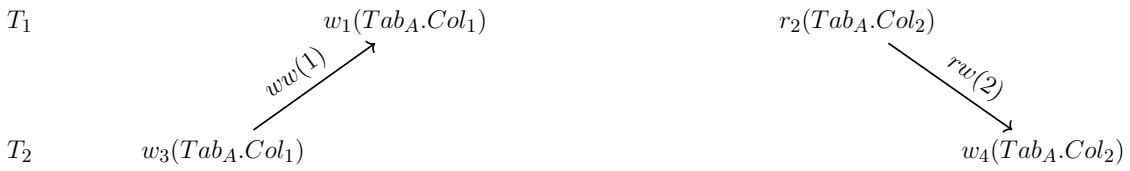
Chapter 7: Operation Granularity and Shadow Edge

In IsoDiff, we have tested different applications and benchmark, and collect results, i.e., Δcycles corresponding to different isolation levels. By looking at the Δcycles generated from running IsoDiff on those applications and comparing with original database applications, we find it potential to continue investigating and refining IsoDiff continuously. In Chapter 7, we describe one observation after studying report generated by IsoDiff, and the improvement we made on IsoDiff.

7.1 Commutative Operations

In section 5.1, we show the evaluation result for TPC-C under Read Committed. It presents some true problems and some cases captured by IsoDiff and verified to be actually false positive cases by hand. One false positive case is about *commutative update*. A typical scenario for *commutative update* is that two transactions both contain self increment update, i.e., SQL statement like "UPDATE .. SET Col_x = Col_x + value ...", which leads to a write dependency edge if two transactions access the same row. If there's another edge between the two transactions, the two edges could construct a cycle in DSG(H) where H is the execution history with anomaly.

Figure 7.1a depicts the DSG of the problem we just mentioned. The figure shows that there are conflicts between operation w_1 and w_3, and between operation r_2 and



(a) Example with Read Committed

(b) Example with Snapshot Isolation

Figure 7.1: Examples of Effects of Write Dependency Edge with Commutative Writes

w_4, which constitute the cycle between T_1 and T_2. The cycle indicates that there's anomaly with history corresponding to the Figure 7.1a, but there's no effect on the result of serializing the two transactions. In fact, no matter which transaction runs first or how the two transactions are interleaved, the final result happens to be the same as running the two transactions serially in some order, which means the write dependency edge (ww(1)) in the Figure 7.1a doesn't have an effect on the data item it writes.

A simple solution is to remove the write dependency edge from the graph, and let the cycle disappear in Figure 7.1a. However, this type of write dependency edge is necessary in some scenario as shown in Figure 7.1b. In this case, the write dependency edge (ww(3)) actually serializes the two transactions, whereas we may interpret it by mistake as a write skew problem if ignoring the ww edge. The conclusion of this example is that we should consider this type of write dependency edge.

7.2 Shadow Edge

The example shown in Figure 7.1 seems contradictory at first glance, but it actually could be unified using one approach. On the one hand, the conflict edge should be ignored for the case shown in Figure 7.1a. On the other hand, the conflict edge should be considered for the case shown in Figure 7.1b. We call such edge as *Shadow Edge*.

Definition 7.1. *Shadow edges are a set of special edges that changing the order of the operation at the two ends won't affect the result produced by the operations.*

Figure 7.2: Eliminate Δcycle by reordering commutative operation pair

To cope the issue introduced by shadow edge, we could first ignore this type of write dependency edge, i.e., shadow edge, during the phase of searching for cycles, then add it to the graph if it shows *correlation* with some edge in the enumerated cycle. With this approach, we will ignore the write dependency edge (ww(1)) in Figure 7.1a, so that we prune the false-positive case. Meanwhile, we could find that this approach solves the problem in Figure 7.1b simultaneously: we ignore the edge ww(3) first as it's a commutative conflict edge during cycle search phase, and then find it during the correlation check phase. Adding back the "shadow edge" with enumerating all possible directions, it fails to find one scenario in which all cycles in the graph are allowed by

Snapshot Isolation. The two transactions are thus actually able to be serialized when adding back the write dependency edge (ww(3)) shown in Figure 7.1b.

The reason behind the approach is that the *commutative operation* pair can be reordered on the transaction level graph to be a cycle-free graph if there are no other constraints. For example, the edge ww(1) of a Δcycles in Figure 7.1a can be rearranged as shown in Figure 7.2, so that it even doesn't have a Δcycle. However, if a shadow edge is raised from correlation check, we need to verify whether the shadow edge has the potential to invalidate Δcycle regardless of its direction as the example shown in Figure 7.1b.

7.3 Extend IsoDiff with Row Level Operation Granularity

So far, we have discussed what a shadow edge is, how it affects the validation of Δcycles, and one solution to the problem. The shadow edge shown in Figure 7.1 is mentioned to be a special SQL statement which makes two operations of the shadow edge commutative. We can identify such case with developers explicitly marked the operation as commutative or automatically identify such case via sophisticated SQL parsers.

Another scenario where we should consider shadow edges is when the underlying database management system runs with the minimum granularity on row level, which is a common practice on developing DBMS in consideration of performance. Previously, IsoDiff was implemented with considering operation granularity on column level, i.e., a conflict happens if and only if two operations access the same row and same column under the same table. However, if considering operation granularity on row level, we need to extend IsoDiff with a more general model that covers both cases.

If we consider transactions on row granularity, each operation may contain more than one object, i.e., multiple columns, in its target operation. For example, if we have one SQL statement as following:

```
UPDATE Customers
    SET Name = "Tom", City = "Columbus"
    WHERE ID = 12345;
```

Figure 7.3: Example of Update Statement with Multiple Columns

If we consider it with column level granularity, the SQL statement is converted as a sequence of three operations:

$$R(Customers.ID),\ W(Customers.Name),\ W(Customers.City)$$

Figure 7.4: Column Level Operation List for Statement in Figure 7.3

However, if we consider it with row level granularity, same type of operations can be grouped up in a single operation, making the aforementioned SQL statement as a sequence of two operations:

$$R(Customers.ID),\ W(Customers.Name,\ Customers.City)$$

Figure 7.5: Row Level Operation List Translated for Statement in Figure 7.3

The change of operation granularity from column level to row level aligns with implementation of some database management system that treats row as minimum unit to read and update. In a DBMS managing concurrency control via locks, it locks rows identified by SQLs' predicate and update corresponding column specified in one transaction. If another transaction intends to access some rows overlapped with rows locked by the first transaction, it waits until the locks are released by the first transaction and then proceeds. Similarly, if a DBMS uses multi-version to manage concurrency control and uses row level granularity, it stores metadata (e.g., version number or timestamp) per row and metadata will be updated if any column of that row is updated.

If conflicts are considered on the column level, it could be either two operations touch exactly the same row and column or not, if there's no further constraints added by developers or operation semantics. On the other side, considering conflicts on row level, however, could result in one additional scenario that both operations access different columns of the same row. In such case, the operations are *commutative* as changing the order doesn't affect the result observed after executing the operations. According to Definition 7.1, conflict edges consisting of such operations are shadow edges.

To support different scenario, IsoDiff implements conflict check on both conflict level. For conflicts on column level, IsoDiff checks if two operation access the same table and column, and mark a potential conflict if two operations access the same table and column. As for checking conflicts on row level, IsoDiff implements a function testing whether two operations 1) have overlapped table and column, 2) have overlapped table but no overlapped column, 3) have no overlapped table. In the first case, the

test function marks the conflict as *concrete* one; in the second case, the test function returns that the conflict edge is a shadow edge; in the last case, there's no edge between those two operations.

7.4 Test with Shadow Edge

To evaluate the effectiveness of shadow edges introduced into IsoDiff, we run experiments to test some applications under isolation level Snapshot Isolation with row level granularity. To make a fair comparison on the test with/without shadow edge, IsoDiff first searches Δcycle candidates ignoring shadow edge, and then runs the validation process on the Δcycle candidates twice with ignoring shadow edges in one round and considering shadow edges in another round, and finally compares whether one Δcycle candidate is valid in the round that ignoring shadow edges but invalid in the round taking shadow edges into consideration. Since a Δcycle candidate invalidated by validation check without considering shadow edges will automatically be invalidated with considering shadow edges, the experiment counts the difference of scenario with and without shadow edge.

Application	**#Δcycle without Considering Shadow Edges**	**Invalidated by Shadow Edges**
tpcc	0	0
broadleaf	4,121	2
shoppe	1,623	49
spree	81,386	6,495
oscar	78,668	153
attendize	57,597	6,979
fa	1,379	27

Table 7.1: Experiment of Invalidation by Shadow Edge with experiment setting (K=5, N=5, Snapshot Isolation)

The result is shown as in Table 7.1. From the experiment result, we observe that, for different applications, the portion of Δcycles invalidated by shadow edges varies because of the transaction set of each application is different from the others. We can learn from the result that the granularity of operation could affect the validation of Δcycles, and considering shadow edges can reduce false-positives on the searched Δcycle candidates.

Chapter 8: Extending IsoDiff With Feedback

IsoDiff is a tool to identify anomalies via enumerating anomalies, i.e., Δcycles, comparing the given isolation level with SERIALIZABLE. To better facilitate developers' implementation on their database application, IsoDiff incorporates developer knowledge into the analysis on the transaction set of the database application. We call such developer knowledge as *Feedback* to IsoDiff, which can be integrated into the procedure of analysis in the future.

8.1 Static and Dynamic Feedback

As mentioned in Chapter 5, the output of IsoDiff may generate some false-positives which cannot be identified only by looking at the transaction set of the database application and the target isolation level without "hints" in developers' head, which we refer as *Feedback*. We justify that the feedback is one crucial component to help developers identify anomalies on their specific database application along with extracting comprehensive transaction set and providing targeted isolation level. However, feedback varies from application to application as we state in the evaluation in Chapter 5: the false-positive cases are different in TPC-C benchmark and FrontAccounting, which is one of the database applications we evaluate.

To solve the variety problem of *feedback* on different database applications, IsoDiff implements a framework to express different *feedback* as mutations of the edges or nodes on static dependency graph originally built from transaction set of database application, or validation test on execution history of enumerated searching Δcycles. We refer to the former type of feedback as *static feedback*, and the latter type of feedback as *dynamic feedback*.

For example, if a developer is confident that one type of transaction will not conflict with other transactions with the same type in any situation (e.g., a purchase transaction is tightly coupled with one customer and one customer is guaranteed only to open one purchase transaction by other component of the system), it's safe to mark all transactions of that type except one as invisible, so that IsoDiff will skip the analysis on finding any Δcycles having conflict of two transaction instances of the same of the indicated type.

In Table 8.1, we summarize all the feedback implemented in IsoDiff with notes explaining how the feedback works specifically.

Feedback Interface and Management. We have introduced the concept of *static feedback* and *dynamic feedback* in Section 8.1. Following that, we describe the management of those feedbacks in IsoDiff. As we stated in previous section, any feedback falls into those two categories, so that IsoDiff designs two interfaces for the two feedback categories.

In IsoDiff, *Static Feedback* applies on the static dependency graph. Such feedback mutates static dependency graph once the graph is built at initialization phase. Changes are applied globally as Δcycle search and validation are based on the static dependency graph patched with all static feedback. The signature of interface applying

Feedback	**Type**	**Notes**
Select For Update	Static	Read operation in the target select statement won't have conflicts with write operations from other transaction conflicting with the write operation following the targeted select statement
Ignore Transaction	Static	Corresponding transaction nodes are removed from static dependency graph generated by IsoDiff
Self Increment Update	Static	Conflicts between two operations of the same transaction type and marked with self increment update will be marked as shadow edge on static dependency graph.
Atomic Insert	Dynamic	On validation phase, IsoDiff checks if there are other operations interfered with the target insert statement
Timing Violation	Dynamic	On validation phase, IsoDiff checks if there's cyclic dependency on some operations
Select Insert Exclusion	Dynamic	On validation phase, IsoDiff checks if select is querying a value from future insertion
Single Transaction	Static	Target transaction node will not be duplicated in regard to different isolation level on static dependency graph
Lock Based Database	Dynamic	Check if Δcycles are built from two transactions formulating read/write dependency edge and anti-dependency edge

Table 8.1: Summary on Feedback Implemented in IsoDiff

such feedback is expressed as shown in Figure 8.1. Any static feedback will inherit the interface and implement its own logic to mutate the static dependency graph.

```
void updateGraph(StaticDependencyGraph &graph);
```

Figure 8.1: Interface Signature of Static Feedback

On the other hand, *Dynamic Feedback* validates whether one searched cycle is actually a Δcycle. It examines whether the enumerated cycle is legitimate or a false-positive based on constraints including timing causality and implementation of target database management system. It returns *True* if the searched cycle doesn't violate rules set by the feedback, otherwise it returns *False*. The signature of interface for all dynamic feedbacks is as shown in Figure 8.2, and any specific dynamic feedback implements its independent logic to validate on the searched cycle. If the searched cycle passes validation checks from all feedbacks, it will be in the final report as one Δcycle.

```
bool graphCheck(DeltaCycle &graph);
```

Figure 8.2: Interface Signature of Dynamic Feedback

8.2 Extend IsoDiff on Other Isolation Level

IsoDiff searches cycles by looking at potential conflicts. For each cycle it searches, IsoDiff verifies whether these potential conflicts could happen together and even with

more conflicts introduced by correlated conflicts. The approach of searching cycle is based on the graph based definition for weak isolation levels. For example, READ COMMITTED allows cycles with at least one rw edge in DSG. The graph based definition is established on representing anomalies purely by conflicts between transactions, which is free from implementation. By following the graph based definition of isolation level, IsoDiff is possible to capture all possible anomalies that could happen on the given weak isolation level. In another word, from the perspective of algorithm, IsoDiff searches cycles to represent anomalies and is free from any implementation of database.

As mentioned before, a developer may have an idea of which isolation he intends to use with the database application. A further decision would be what database management system he wants to use on the database application. Different database may perform differently even with same isolation level [50], and the anomalies appearing with different database may change due to the implementation of the underlying database management system.

For example, if the underlying DBMS providing cursor for clients to access, it does provide more security on preventing more anomalies. The cursor based approach is a popular feature provided by many database vendors. When using cursor, the transaction assign a cursor with a SELECT SQL statement, and the row selected by the cursor is locked so that other transactions cannot access it. In this case, the cursor based approach protects the data item under cursor from updating by other transactions. The cursor based approach is defined as *PL-CS*, or called *Cursor Stability* (CS) [12], in terms of graph based definition. In *Cursor Stability*, it prevents *Dirty Writes*, *Dirty Reads*, which is similar to READ COMMITTED, plus one anomaly called *G-Cursor*. *G-Cursor* depicts the scenario that one transaction T_1 read one object x,

and updates on the object x, at the same time, another transaction T_2 does an update on the same object x in between the read and write operation in T_1. In the DSG, it causes one rw edges on the object x from T_1 to T_2, and one ww edges on objects x from T_2 to T_1, thus forms a cycle.

Another example is one isolation level called Monotonic Atomic View (MAV), which we have introduced in Section 2.2. MAV is a specific implementation of Read Committed with read/write lock, thus it naturally prohibits some anomalies which are allowed by general definition of Read Committed. Isolation level Monotonic Atomic View proscribes all anomalies proscribed by Read Committed plus one anomaly called *Predicate-Many-Preceders* (PMP) [50]. A history exhibits the anomaly PMP if two read operations in one transaction have intersection on the predicate on which their reads are based, and read *inconsistent* result lying on the intersection. This anomaly is allowed under isolation level Read Committed by database management system implemented with optimistic concurrency control but not allowed by DBMS implemented with two phase lock. For DBMS with two phase lock, the update that changes the overlapped portion of two read operations will prevent the second read from execution, so that it cannot happen in DBMS with such implementation.

Experiment. To express such database management system specific feedback, we add one feedback called *Lock Based Database* in IsoDiff (corresponding to the row with the same name on Table 8.1), and test it with applications running under Read Committed while enabling this feedback test on Δcycles validation. We run experiment with the setting (K=5, N=5) under isolation level Read Committed with operation granularity set to row level and column level. Table 8.2 shows experiment result with row level granularity, and Table 8.3 shows experiment result with column

level granularity. From experiment result, we observe that the feedback *Lock Based Database* can invalidate some Δcycle candidates.

Application	**Invalidated By LockDB check**	**Total Number of Invalidated Δcycles**
attendize	4,003	5,226
broadleaf	337	2,077
fa	267	543
flarum	481	4,115
lfs	135	980
opencart	23	374
oscar	4,395	12,877
prestashop	97	2,104
shoppe	170	670
spree	6,584	18,939
tpcc	1,372	2,056

Table 8.2: Experiment of Invalidation by Lock Based DB with experiment setting (K=5, N=5, Read Committed) on row level

Application	**Invalidated By LockDB check**	**Total Number of Invalidated Δcycles**
attendize	3,689	11,432
broadleaf	90	33,531
fa	226	1,053
flarum	317	25,908
lfs	35	16,013
opencart	42	2,472
oscar	505	239,759
prestashop	68	11,317
shoppe	465	2,227
spree	5,493	64,832
tpcc	1,468	3,143

Table 8.3: Experiment of Invalidation by Lock Based DB with experiment setting (K=5, N=5, Read Committed) on column level

Chapter 9: Conclusion

In this work, we demonstrate 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. IsoDiff is able to analyze anomalies associated with some isolation level for real database application as it builds abstract model following isolation level definition in graph approach and apply efficient and scalable search algorithm on exploring anomalies.

Besides that, we find that IsoDiff is a flexible tool to extend with requirement from different dimension. We provide feedback mechanism in IsoDiff for developers to interactively debug their database applications iteratively. We conduct thorough profiling on the performance of IsoDiff, and design a deterministic random mechanism to ensure the runtime and result is stable when using same experiment setting. We study the report generated from IsoDiff, and propose several approaches to further reducing the false positives based on the observation from the study. We also extend IsoDiff on other isolation levels closely related to database implementations.

In the future, IsoDiff can be extended by utilizing static analysis to automatically extract transactions and correlations, and to further improve the accuracy of false positive detection. Another branch we are considering is to hide the complexity of

using IsoDiff in terminal and text, and provide a friendly Graphics User Interface (GUI) for developers to debug their applications in browser. For experiments, the study of IsoDiff shows a promising approach to debugging complex database application, and we believe it benefits developers in delivering correct database application with confidence.

Bibliography

[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] Jepsen distributed systems safety research. https://jepsen.io/, November 2021.

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

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

[14] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment*, 7(3):181–192, 2013.

[15] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems (TODS)*, 41(3):1–45, 2016.

[16] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[17] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[18] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):1–42, 2009.

[19] 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.

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

[21] CloudLab. `https://www.cloudlab.us`.

[22] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen 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.

[23] DB-Engines Ranking. `https://db-engines.com/en/ranking`.

[24] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[25] Edsger W. Dijkstra and A.J.M. van Gasteren. Well-foundedness and lexical coupling (with A.J.M. van Gasteren). circulated privately, September 1990.

[26] Alan Fekete, Shirley N Goldrei, and Jorge Pérez Asenjo. Quantifying isolation anomalies. *PVLDB*, 2(1):467–478, 2009.

[27] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)*, 30(2):492–528, 2005.

[28] R Lorie J Gray, GF Putzolu, and IL Traiger. Granularity of locks and degrees of consistency. *Modeling in Data Base Management Systems, GM Nijssen ed., North Holland Pub*, 1976.

[29] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo 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.

[30] Sudhir Jorwekar, Alan Fekete, Krithi 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.

[31] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, nov 2020.

[32] Simon Koch, Tim Sauer, Martin Johns, and Giancarlo Pellegrino. Raccoon: Automated verification of guarded race conditions in web applications. 2020.

[33] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[34] Aapo Kyrola, Guy Blelloch, and Carlos 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.

[35] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4), 2011.

[36] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang 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.

[37] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang 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.

[38] MySQL. `http://www.mysql.com`.

[39] Testing MySQL transaction isolation levels. `https://github.com/ept/hermitage/blob/master/mysql.md`.

[40] Oracle Database. `https://www.oracle.com/database/`.

[41] Data Concurrency and Consistency for Oracle DB. https://docs.oracle.com/cd/E25054_01/server.1111/e25789/consist.htm#BABGEGBC.

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

[43] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.

[44] PostgreSQL. https://www.postgresql.org/.

[45] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy 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.

[46] Amitabha Roy, Ivo Mihailovic, and Willy 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.

[47] Microsoft SQL Server. https://www.microsoft.com/en-us/sql-server/.

[48] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao 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.

[49] Adriana Szekeres and Irene 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.

[50] Hermitage: Testing transaction isolation levels. https://github.com/ept/hermitage.

[51] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel 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.

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

[53] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel 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.

[54] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–15, 2014.

[55] Todd Warszawski and Peter Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *SIGMOD'17*, pages 5–20. ACM, 2017.

[56] 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.

[57] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince 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.

[58] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang 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.

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

[60] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: an evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[61] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. Bcc: reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *PVLDB*, 9(6):504–515, 2016.

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

[63] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander 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, February 2015. USENIX Association.

[64] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 465–477, 2014.

[65] Xiaowei Zhu, Wentao Han, and Wenguang 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.