IsoBugView: Interactively Debugging Isolation Bugs in Database Applications

Drew Ripberger
The Ohio State University
ripberger.8@osu.edu

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

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

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

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

ABSTRACT

Database applications frequently use weaker isolation levels, such as Read Committed, for better performance, which may lead to bugs that do not happen under Serializable. Although a number of works have proposed methods to identify such isolation-related bugs, the difficulty of analyzing reported bugs is often underestimated, since these bugs often involve multiple complicated transactions interleaved in a specific order and they often require users' feedback to improve the accuracy of bug analysis.

This paper presents IsoBugView, a tool to visualize isolation bugs and incorporate users' feedback: to address the challenge that a complicated bug may include much information and thus is hard to present, IsoBugView displays a high-level overview of the bug first and displays further information of individual pieces if the developer needs further investigation. To incorporate users' feedback, IsoBugView embeds hook functions into the backend analysis tool to preprocess a dependency graph and postprocess a found cycle and further allows a user to apply predefined hook functions in its graphic user interface. Our experience shows that IsoBugView has greatly improved our productivity of analyzing isolation bugs.

PVLDB Reference Format:
Drew Ripberger, Yifan Gan, Xueyuan Ren, Spyros Blanas, and Yang Wang. IsoBugView: Interactively Debugging Isolation Bugs in Database Applications. PVLDB, 15(12): 3726 - 3729, 2022.
doi:10.14778/3554821.3554885

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/drewrip/isobugview.

1 INTRODUCTION

Database systems provide different isolation levels to regulate how concurrent transactions may interleave. Among them, Serializable, which means the database guarantees that concurrent execution of multiple transactions is always equivalent to a serial execution of them, provides strong correctness guarantees so that a database application developer does not need to reason about concurrency. However, multiple studies have shown that, in practice, many database applications are using weaker isolation levels, such as Read Committed and Snapshot Isolation, probably to gain better performance [2, 9]. As a trade-off, this may lead to anomalies (called isolation bugs in this paper) that do not happen under Serializable, and even security issues [11].

A number of prior works have developed theories and algorithms to identify isolation bugs [1, 3, 5–7, 10, 11]. Most of them rely on the dependency graph theory, which models transactions as vertices, models dependencies among transactions as edges, and models isolation bugs as certain types of cycles in the graph. However, we observe the difficulty of analyzing the reported cycles is often overlooked. First, real-world applications often have long and complicated transactions and an isolation bug often involves multiple transactions interleaved in a specific order. As a result, in our experience, interpreting the text output of prior works usually involves manually drawing the dependency graphs on a board and frequently going back and forth between the graph and the original SQL transactions. Second, such procedure often needs application-specific knowledge from the developers to improve the accuracy of bug finding, and we often need to modify the source code of corresponding tools to incorporate such application-specific knowledge.

This paper presents IsoBugView, a tool to visualize isolation bugs and incorporate developers' knowledge. To address the challenge that a complicated bug may include much information and thus is hard to visualize, IsoBugView displays a high-level overview of the bug first and displays further information if the developer needs further investigation. To incorporate developers' knowledge, IsoBugView has refactored the underlying analyzer tool so that it can introduce users' knowledge into the analysis without changing its source code. We further integrate several types of mostly commonly used knowledge into IsoBugView's graphic user interface.

We have applied IsoBugView to a number of database applications, and found it greatly improves our productivity of analyzing reported bugs.

2 BACKGROUND AND RELATED WORK

Suppose an online shopping application has provided a transaction to purchase an item: it uses a select statement to read the item count, checks the item count is positive, and then uses an update statement to decrement the item count. Suppose two purchase transactions are executed concurrently and the isolation level of the database is set to Read Committed, which asks that read operations retrieve

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554885

a committed value. It is possible that both transactions execute their "select" statements at the same time, confirm that the item count is positive, and both decrement the item count, which means that the final item count becomes negative. This will never happen under Serializable, but is allowed by Read Committed since both "select" statements indeed see committed values.

To identify such issues, the *de facto* theoretical foundation is the definition of isolation levels by Adya et al. [1] which many other works adopt to identify non-serializable executions [1, 3, 6, 7, 10, 11]. The Adya et al. definition models the execution of transactions as a dependency graph, in which nodes are transactions and edges represent dependencies across transactions; isolation levels are defined based on whether they prevent different types of cycles in the dependency graph.

Following this definition, prior works try to identify potential non-serializable executions by searching for cycles that are not allowed by Serializable but are allowed by the target isolation level. For example, for Read Committed, we should search for cycles with at least one read–write dependency edge; for Snapshot Isolation, we should search for cycles with at least two consecutive read–write dependency edges [3, 6].

3 DESIGN AND IMPLEMENTATION OF ISOBUGVIEW

While the theoretical foundation of identifying non-serializable executions is well studied, we find understanding the output of existing tools is a challenging task due to the following reasons: to understand a detected cycle, we often need to convert it from a text format into a figure, usually by drawing it on a board. However, for complicated transactions, we often need to go back and forth from the figure to the original SQL statements to understand how the cycle may be generated at application level. On top of this, not all pertinent information regarding how a database is used in practice can be deduced from its state or logs. For instance, an application may enforce additional constraint (e.g., two customers cannot have the same ID) to prevent non-serializable executions, and without such application-specific knowledge, existing analysis often incurs false positives. As a result, we often need to change the source code of the tool in an application-specific manner to filter certain cycles. All are affecting our productivity when analyzing isolation bugs.

To address these challenges, we have built IsoBugView, a tool to visualize isolation bugs and incorporate users' feedback.

3.1 Overview of IsoBugView

The IsoBugView system is comprised of two distinct parts: a backend that enumerates and checks cycles leveraging a modified version of the IsoDiff tool [5] (Section 3.2), and a novel frontend GUI, that presents tools to assist developers in understanding and debugging the cycles found in the backend (Section 3.3).

All that must be provided to IsoBugView is an SQL log file of running the target application and a corresponding schema CSV file that outlines the structure of the database that was operated on. The user needs to submit these two files through the IsoBugView GUI, which will transfer these files to the IsoBugView backend.

On submission the IsoBugView backend will perform the analysis in the background while the user is brought to the main status page that encompasses the majority of IsoBugView's functionality. Until the search is complete there will be no cycles present and there will be a loading screen shown. On completion of the graph search, the developer will be able to view all cycles and perform detailed examination of each cycle through its frontend GUI.

When examining a certain cycle, if the user finds it is a false positive, s/he can provide feedback to IsoBugView. For example, s/he could indicate that a certain dependency edge will not exist because of an application constraint, or a certain transaction does not need to be analyzed because it is always executed serially, etc. The IsoBugView GUI will submit such feedback to the backend, which will re-run the analysis with the additional information.

3.2 IsoBugView Backend

We implement IsoBugView backend based on IsoDiff [5], but with major modifications to support the functions of IsoBugView.

IsoDiff takes the logs and schema files provided by the DBMS as the input, and can develop a dependency graph of the transactions that were executed in the log. Using the dependency graph, IsoDiff can begin to enumerate cycles that represent potentially problematic executions by the application, that are not serializable but are allowed by the database's weaker isolation level (Read Committed and Snapshot Isolation are currently supported).

To facilitate the building of the IsoBugView GUI, we have modified the implementation of IsoDiff to enrich its output. To be concrete, IsoDiff and similar tools model complicated SQL statements as lists of READ/WRITE operations and output found non-serializable executions as a list of involved READ/WRITE operations together with the dependencies or partial orders among these operations. To help a developer better understand a non-serializable execution, IsoBugView needs to map such READ/WRITE operations back to their original SQL statements. To accomplish this goal, we have modified IsoDiff to connect READ/WRITE operations to original SQL statements when parsing these statements and to carry such connection throughout the analysis to the final output.

To incorporate a user's application-specific feedback, we have modified IsoDiff to add two hook functions: The first hook function is executed after a dependency graph is generated but before cycle search is performed. This hook function can manipulate the generated dependency graph to add/delete nodes, edges, etc. The second hook function is executed after a cycle is found to determine whether the cycle should be filtered out. By providing an implementation of these two hook functions to IsoDiff, a user can provide feedback to IsoDiff without changing the source code of IsoDiff. We have provided some predefined hooks through the IsoBugView GUI. In practice, we expect a common user should be able to use the predefined hooks in most of the cases and an experienced user could implement his/her own hooks when necessary.

3.3 IsoBugView GUI

We implement IsoBugView GUI as a web service. After an analysis is complete, IsoBugView will show a web page like Figure 1. A sidebar on the left is populated with a list of isolation bugs, each named by the transactions they contain (①). Looking at the entry of the list in Figure 1 are a few buttons that serve to help a user notate the analysis and build their knowledge of the graph: the middle

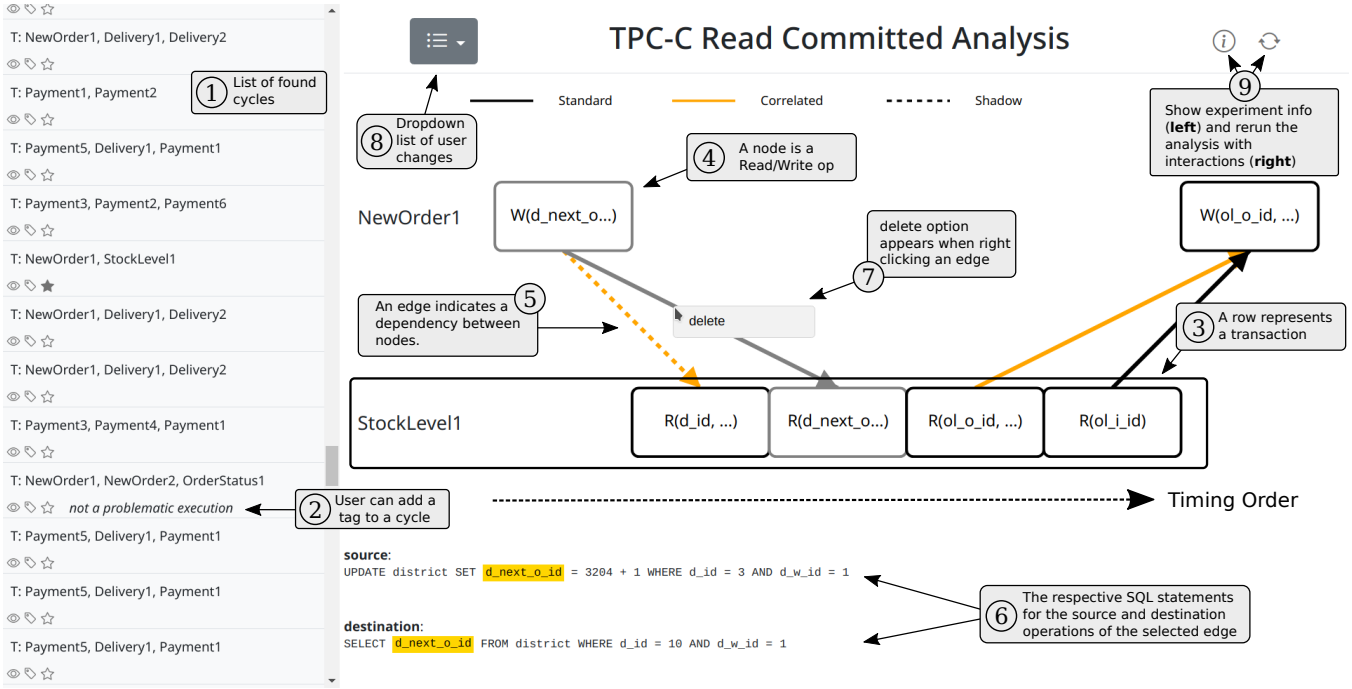

Figure 1: An example analysis of a TPC-C Read Commited execution using IsoBugView

"tag" button allows a user to add a tag or short note underneath the cycle in case it is of specific interest, an example of which can be seen in the ② entry of the list; the right star button gives a user the ability to favorite a cycle so they may quickly recognize it later; the leftmost "eyeball" button generates the graph visualization of the cycle in the graph area to the right.

Bug visualization. The bug visualization is the main tool that empowers a developer to diagnose the problematic ways their application is interacting with the database. On clicking the view button, a graph is drawn, with nodes ordered from left to right following their timing order during the execution. To compute the order, IsoBugView GUI performs a topological sort based on the order of operations within a transaction and the dependencies among different operations.

In the figure, nodes on the same row are from the same transaction, whose name is shown on the left side of the row. For example, the nodes shown by ③ is a part of the "StockLevel" transaction. Every node maps to either a read or write operation on a specific column in one of the tables of the database (e.g., ④). The directed edges that connect the nodes correspond to a dependency between the operations (e.g., ⑤). The destination of an edge has a dependency on the source node. IsoBugView GUI colors a dependency edge based on its type: a solid black line indicates a normal dependency edge that is part of the cycle in the transaction dependency graph; a yellow line indicates a correlated edge, which always happens together with a normal edge [5]; a dotted line indicates a shadow edge, which should not be used during cycle detection but should be used during correlation analysis [4]. Note that an edge can be both correlated and shadow (e.g., ⑤).

The figure may be further dissected by clicking different elements of the graph. Selecting any one of the nodes generates a list, just below the visualization, of all of the SQL statements that make up the transaction the node is a part of and puts the statement the operation originates from in bold. References to the columns of interest are also highlighted in yellow in the statement. In addition to the nodes the edges of the graph can also reveal more information. By selecting any one of the edges, the SQL statements that contain the corresponding source and destination operations will be listed underneath the visualization (e.g., ⑥), and the referenced columns will be highlighted in yellow (e.g., d_next_o_id in ⑥). Providing the raw SQL code that corresponds to the dependencies and operations of interest let developers map the executions that IsoBugView generates directly back to their own application code. They can observe the statements and functions that have potentially worrisome executions, then begin to determine if they need to write safer code that is in line with the isolation level used by their application's database.

Incorporate users' feedback. In addition, as discussed above, IsoBugView has predefined a few commonly used hooks and allows a user to use these predefined hooks through the IsoBugView GUI. To be concrete, a user can "delete" an edge in the graph, maybe because a certain application-level constraint prevents the corresponding dependency edge from happening in practice; a user can also "delete" a transaction in the graph, maybe because that transaction can tolerate non-serializable result. We plan to expand the list of predefined hooks in the future.

To give a concrete example, by right clicking on an edge, as shown in ⑦, a user can remove the edge from the analysis. The

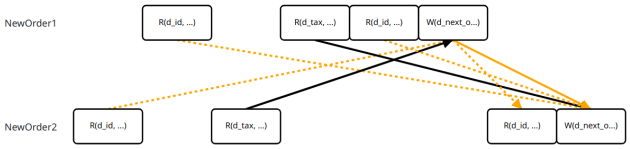

Figure 2: A potential isolation bug reported by IsoBugView.

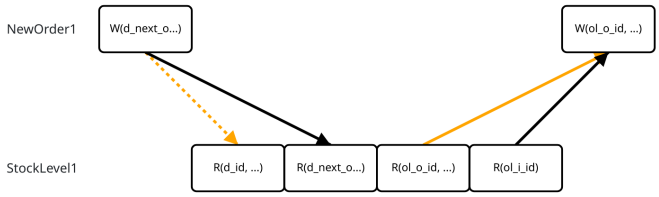

Figure 3: Another potential isolation bug reported by IsoBugView.

drop down list provided with ⑧ gives a breakdown of which edges are marked to be removed. By clicking the "rerun" button, the rightmost button shown by ⑨, the IsoBugView backend will run the analysis again with the original parameters, but the desired edges deleted. Once the new analysis is done, a user can observe the new result, which not only avoids the execution shown in Figure 1, but also other executions that involve the deleted edge.

4 APPLICATION EXAMPLES

In this section, we show examples of how IsoBugView helps us understand potential bugs of running TPC-C under the Read Committed isolation level. We run the TPC-C implementation from the OLTPBench [8] upon a MySQL database, configure MySQL to log SQL traces, and also record the schema information from MySQL. We then submit the schema and log to the IsoBugView GUI.

Figure 2 shows a potential bug involving two NewOrder transactions. By clicking the first normal dependency edge and viewing the corresponding SQL statements, we can know that the first NewOrder transaction reads the d_next_o_id value, which is updated by the second NewOrder transaction; by checking the second normal dependency edge, we can see that the second NewOrder transaction reads the d_next_o_id value, which is then read by the first NewOrder transaction. By looking at the logic of the whole NewOrder transaction, we see that NewOrder reads the d_next_o_id, increments it by 1 and uses it as the ID of the new order, and then updates the d_next_o_id value to reflect the increment. Therefore, if two NewOrder transactions read the d_next_o_id value at the same time, they may end up using the same ID for different orders, which is problematic. This problem could be fixed by adding the “for update” keyword for the select statement.

Figure 3 shows another potential bug reported by IsoBugView, which is the same as the one shown in Figure 1. It involves a NewOrder transaction and a StockLevel transaction. By clicking the left normal edge, we can know that the NewOrder transaction updates the d_next_o_id value and the StockLevel transaction will read the updated d_next_o_id; by clicking the right normal edge, we can know that the StockLevel transaction reads the ol_i_id value and then the NewOrder transaction updates that value. In this case, the StockLevel transaction reads the updated d_next_o_id value but still the stale ol_i_id, which will not happen under a serializable execution. However, the TPC-C specification indicates that serializable result is not required for the StockLevel transaction. This is a typical example of how application-level constraint or requirement can introduce false positives into the analysis. Therefore, we use IsoBugView to remove StockLevel transaction and re-run the analysis, which results in a shorter list of potential bugs.

5 CONCLUSIONS

To improve the productivity of analyzing isolation bugs, we have built IsoBugView, a tool to visualize isolation bugs and incorporate users’ feedback. IsoBugView visualizes an isolation bug using the dependency graph and further allows a user to view detailed information of each node and edge in the graph. IsoBugView embeds hook functions into the underlying analysis tool to express application-specific feedback and further allows a user to submit such hook functions through the GUI. Our experience shows that such a tool is critical to the productivity of analyzing isolation bugs.

ACKNOWLEDGMENTS

We thank all reviewers for their insightful comments. This material is based in part upon work supported by the National Science Foundation under Grant Numbers CNS-1908020 and CCF-2118745.

REFERENCES

[1] A. Adya, B. Liskov, and P. O’Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering*.
[2] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC ’17)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/3087801.3087802
[3] Alan Fekete, Dimitrios Liarokapis, Elizabeth O’Neil, Patrick O’Neil, and Dennis Shasha. 2005. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.* 30, 2 (June 2005), 492–528. https://doi.org/10.1145/1071610.1071615
[4] Yifan Gan. 2021. Exploring Transaction Anomalies under Weak Isolation Levels for General Database Applications (Dissertation). https://github.com/SayuRanger/dissertation/blob/main/Phd_Thesis_Yifan_Gan.pdf. (2021).
[5] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *Proc. VLDB Endow.* 13, 11 (2020).
[6] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. 2007. Automating the Detection of Snapshot Isolation Anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB ’07)*. VLDB Endowment, 1263–1274. http://dl.acm.org/citation.cfm?id=1325851.1325995
[7] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 268–280.
[8] oltpbench [n.d.]. OLTPBench. https://github.com/oltpbenchmark/oltpbench.
[9] Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *SIGMOD 17*. 3.
[10] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 63–80. https://www.usenix.org/conference/osdi20/presentation/tan
[11] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD ’17)*. ACM, New York, NY, USA, 5–20. https://doi.org/10.1145/3035918.3064037