Developer's Responsibility or Database's Responsibility? Rethinking Concurrency Control in Databases

Chaoyi Cheng[*][†]
The Ohio State University

Mingzhe Han[*]
The Ohio State University

Nuo Xu[‡]
The Ohio State University

Spyros Blanas
The Ohio State University

Michael D. Bond
The Ohio State University

Yang Wang
The Ohio State University

ABSTRACT

Many database applications execute transactions under a weaker isolation level, such as READ COMMITTED. This often leads to concurrency bugs that look like race conditions in multi-threaded programs. While this problem is well known, philosophies of how to address this problem vary a lot, ranging from making a SERIALIZABLE database faster to living with weaker isolation and the consequence of concurrency bugs.

This paper studies the consequences, root causes, and how developers fix 93 real-world concurrency bugs in database applications. We observe that, on the one hand, developers still prefer preventing these bugs from happening. On the other hand, database systems are not providing sufficient support for this task, so developers often fix these bugs using ad-hoc solutions, which are often complicated and not fully correct. We further discuss research opportunities to improve concurrency control in database implementations.

1 INTRODUCTION

In an ideal world, a database application developer should encapsulate a sequence of operations into a transaction, and execute multiple transactions concurrently with a SERIALIZABLE database [23], so that the application developer does not need to reason about how concurrent transactions may interleave.

The real world, however, is far from ideal: multiple studies have shown that, in practice, databases are often configured with a weaker isolation level, such as READ COMMITTED, probably due to performance reasons [14, 24]; to perform concurrency control, application developers often use ad-hoc solutions (e.g., use external locks), rather than relying on the database [27]. In both cases, the application developers are responsible for reasoning about the interleaving of concurrent transactions, which may introduce subtle correctness issues [18, 27, 28].

The gap between theory and practice motivates us to rethink the following question about concurrency control in databases: should we insist that the database takes full control of concurrency? Or should we assume that the developers must take part in this task and, if so, what additional support should database systems offer to developers to understand isolation violations?

[*]Chaoyi Cheng and Mingzhe Han contributed equally to this paper.
[†]Currently at the University of Pennsylvania.
[‡]Currently at the University of Southern California.

To answer this question, we have performed a study to a question unanswered in prior studies: while multiple studies have shown that using weakly isolated or ad-hoc transactions *could* introduce anomalies, do these anomalies actually happen or matter in practice? As some researchers have conjectured [5], perhaps the contention level in real-world applications is not high enough to trigger these anomalies frequently. Even if these anomalies are triggered, perhaps they do not violate any application-level constraints. Even if they do violate application-level constraints, perhaps it is okay for the administrator to repair their effects manually. As one can imagine, if these conjectures are correct, there is less motivation to pursue a SERIALIZABLE database implementation.

We have carried out the study by collecting 93 concurrency related issues from different real-world open-source applications and analyzing their root causes, consequences, and how developers fix them. We find that 1) many of them do cause violation of application constraints, like overselling, double spending, and orders with the same ID, which cause users to complain; 2) in almost all cases, developers do want to prevent these anomalies from happening (only one case discusses the possibility of letting the user fix the effect of the anomaly manually); 3) many cases require a significant amount of effort to fix, often leading to lengthy online discussions and significant redesign of the application. Developers rarely use stronger isolation to fix a bug: in only 7 cases developers finally choose to upgrade the isolation level, and in many others they turn to ad-hoc solutions like external locking and application-level versioning. Based on the developer discussion, a common concern of moving to a stronger isolation level is performance, particularly long locking time and deadlock that are more likely to happen under stronger isolation.

From the study, we conclude that issues caused by concurrency control are of concern in many real-world applications: on the one hand, developers are reluctant to use SERIALIZABLE transactions due to performance concerns; on the other hand, weakly isolated or ad-hoc transactions are hard to reason about and thus error prone.

The second half of the paper discusses possible solutions. In short, we believe that to get the best performance, some effort from developers is necessary, since developers often have application-specific knowledge that can help to improve performance. This means we need new abstractions and support to incorporate developers' effort. On the other hand, automatic analysis, in particular combined analysis of SQL and web application code, may still be able to improve over the state of the art. We identify both challenges and opportunities in this direction.

2 BACKGROUND AND RELATED WORK

Imagine a purchase operation that reads the current item count with a SELECT statement, checks that the item count is positive, and then decrements the item count with an UPDATE statement. If two statements are encapsulated in a SERIALIZABLE transaction, then concurrent execution of multiple purchase operations will encounter no problems. If the database uses a weaker isolation level, such as READ COMMITTED, or if these two statements are not encapsulated in a database transaction, then it is possible that two purchase operations execute SELECT concurrently, both find the item count is positive, and then both decrement the item count, resulting in a negative item count. Since this problem resembles a concurrency bug (more specifically, an atomicity violation) in a multi-threaded application written in an imperative shared-memory programming language [21], we call it a "concurrency bug" in the rest of the paper.

To address this problem under weaker isolation levels, the developer may add an additional check at the end of the transaction to ensure the item count is not negative. Alternatively, instead of using database transactions, the developers may rely on external locking services, such as Redis, to prevent the same item count from being read and updated concurrently. This approach is dubbed *ad-hoc transactions* in prior work [27], and we inherit the term in this paper.

Multiple works have shown that using weakly isolated or ad-hoc transactions can lead to concurrency bugs [18, 27, 28]. A number of works try to identify concurrency bugs in database applications by modeling transaction execution as a serialization graph and searching for cycles in the graph [12, 16, 18–20, 28]. However, these works leave a long-standing question unaddressed: do these correctness issues actually matter in practice [5]? As one can imagine, the answer affects how the community should approach this problem.

3 A STUDY OF REAL-WORLD CONCURRENCY BUGS

This section presents our study about how concurrency bugs manifest and get fixed in practice. Many real-world applications today choose to open-source their code and openly discuss their bugs online, which gives us the chance to perform this study.

Methodology. We have investigated bugs from two sources. The first source is the code and bug reports of open-source database applications, which primarily include eCommerce applications but include a few others like online gaming and chatting applications. The second source is the discussions on the websites of Object-Relational Mapping (ORM) tools and database vendors. In these discussions, application developers describe the anomalies they meet and ask for possible ways to fix them.

We searched both the commit history of the source code and the online bug reports and discussions to find bugs that are caused by executing transactions in parallel. The key challenge we met is choosing the keywords to search: while we initially started with keywords like SERIALIZABLE or READ COMMITTED, we found that in practice, developers often don't use such specific keywords. Instead, they often use more general terms like "race conditions," which makes it hard to distinguish our targets from classic concurrency bugs in multi-threaded applications. We have no perfect solution for this problem. We searched for bugs or commit messages that contain keywords from each of the following two groups: the first group is related to concurrency bugs, such as "race" and "concurrent"; the second group is related to transactions, such as "SQL" and "transaction." To determine whether a found bug is relevant to our study, we manually read each bug report, as well as its corresponding source code when needed.

In addition, since the "SELECT FOR UPDATE" command was introduced precisely to address the concurrency issues caused by weaker isolation, it is an accurate indicator of our target. Therefore, we searched for "FOR UPDATE" in both the source code of commits and online discussions. Of course, this makes our study biased toward bugs fixed using "SELECT FOR UPDATE."

Since deadlocks are a well-studied problem, we haven't included them in our study, unless a deadlock is related to the isolation level the database is using.

Using the above methodology, we have found 93 bugs from 46 different applications, ORM tools, and databases. Readers can refer to our online document [1] for the detailed information of each bug. Among the databases they use, all except MongoDB claim to support SERIALIZABLE, though it is known that Oracle does not actually provide SERIALIZABLE isolation [3].

Despite our best efforts to understand these issues, we failed to fully understand a small number of them: because some issues are still under investigation, we know their consequences but not their root causes and how to fix them; because some issues are found by looking at how code changed between commits without clear documentation, we sometimes know how to fix them but not their consequences or root causes.

Limitations. First, due to this work's focus on open-source applications, we don't know whether its conclusions hold for commercial applications. Researchers or practitioners from industry could carry out a similar study on closed-source commercial applications to evaluate this question. Second, since our study is naturally biased toward applications with concurrency issues, this study cannot answer what percentage of applications experience concurrency issues. However, the study at least shows that many applications experience concurrency issues.

3.1 Consequences of Concurrency Bugs

The studied issues lead to both correctness and availability issues. For two issues, we were not able to identify the exact consequences.

Data inconsistency (78 issues). Thirty of the issues cause the miscounting problem, which means the application gets the wrong count of certain items. Among these 30 issues, 14 of them cause the overselling problem, which means the seller sells more items than she has; three of them cause the double-spending problem, which means a buyer is able to use a coupon, voucher, etc., more than once. The others include miscounting the number of people or items internally.

Sixteen of the issues cause the non-unique ID problem, which means the application expects each row in a table to have a unique ID, but this constraint is violated due to concurrency bugs. This can cause various problems at the application level, such as orders not being processed properly, when they have the same ID.

Apart from the above two types, we have observed other data inconsistency issues that are hard to classify, like a table containing mixed data from two transactions.

Unavailability (13 issues). Nine of the issues cause certain processes in the applications to crash or freeze. Four of the issues cause a deadlock. Note that although we do not target deadlocks, we study deadlocks related to the isolation level.

Severity. A few of the bug reports report the severity of the problems in practice:

—"I would say there is about 2–5% chance this might happen. (I have about 100 orders per day)" [7]

—"It is pretty rare, but has happened about 5 times over the past few months in a set of about 10,000 orders" [9]

—"a customer called about having thousands of duplicated and even quadruplicated products ..." [11]

These comments imply that even for applications that are usually idle, they still experience enough contention to manifest isolation level–related anomalies periodically.

3.2 Root Cause of Concurrency Bugs

We initially tried to classify the root causes of these issues using classic database terminology, like lost updates and non-repeatable reads, but we found it difficult to classify some of the bugs using this terminology. Therefore, we introduce our own terminology and discuss why some issues are hard to classify.

Read followed by relevant Write (52 issues). They are caused by concurrent transactions each performing a read operation first and then performing a write operation (update/insert/delete) relevant to the read. Read after relevant write can be further categorized into two types: the first type is read followed by update on the same data item, which we call *RW1* in this paper. For example, if two transactions read the item count in parallel, and then update the item count, the final item count may be inconsistent. In our study, 42 issues are caused by RW1, and it is the leading cause of the miscounting problem.

While we initially classified RW1 issues as lost updates, we find that some of them do not fit with the intuitive definition of lost updates. To be concrete, in the following transaction logic,

Read(a); Check a>=1; b=a-1; Update(a)=b; (Classic lost update)

concurrent execution of the transaction may cause a lost update, since one transaction's update may be overwritten by the other. However, in the following transaction logic,

Read(a); Check a>=1; Update(a)=a-1; (No update is lost)

concurrent execution of the transaction does not lead to any update being "lost." Instead, the final value of "a" may become negative, which is of course still problematic.

In general, we observe that the second case above is less problematic: if an anomaly happens, the application can at least detect it by checking whether the value of "a" is negative. The first case, on the other hand, loses an update silently. In our paper, RW1 encapsulates both cases.

The second type, which we call *RW2*, is more subtle: it is caused by reading a set of records followed by a Write that will affect the result of a concurrent read on the same set of records. Examples of such reads include "select MAX", "select count" and "read the current position." Unlike transactions with RW1 issues, transactions with RW2 issues may not insert/update/delete the same row in the end. For example, consider the common pattern to use "select MAX" to get the maximal value from a table, and then to insert a new row with MAX+1 as the ID, expecting this ID to be unique. Concurrent execution may lead to two transactions getting the same MAX and then inserting two orders with the same ID. In this execution, their insert operations do not operate on the same row, which makes it different from RW1. In our study, 10 issues are caused by RW2, and it is the leading cause of the non-unique ID problem. Again, we find RW2 does not fit well with the intuitive definition of any classic database anomaly: it may look similar to write skew, but in RW2, concurrent transactions may read the same rows.

Inappropriate error handling (10 issues). They cause the user process to either crash or freeze. Most of them are caused by insufficient handling of exceptions. For example, if the database executes two deletes on the same row, it will throw an exception for one of them, which kills the process if the exception is not caught. Interestingly, one issue is caused by over-handling of an exception. In this issue, an update transaction and a delete transaction execute concurrently. When the update transaction finds the data is deleted, it reinserts the data back into the database. As a result, the user who issued the delete transaction will find the delete transaction succeeds but the deleted row is still in the database. In one issue, the error is actually generated by the application: when a user clicks "purchase" twice, perhaps because of network lag, if the database does not have constraints to detect duplicate orders, then this problem generates two orders.

Lock timeout (4 issues). These issues are caused either by deadlock or by executing too many concurrent transactions. Again, while we don't target this problem specifically, we find lock timeout is one of the major concerns of using strong isolation.

Unnecessary concurrent execution (3 issues). Interestingly, we find 3 issues are caused by transactions executing concurrently that should be executed serially. For example, if a transaction writes order information and the second transaction operates on the order, then it does not make sense to execute these two transactions concurrently. In practice, this problem can be caused by developer misunderstanding (e.g., a developer believes transaction A *started* after B means A will be serialized after B, while even a strict SERIALIZABLE database does not provide such guarantee), or caused by misusing asynchronous execution in the programming language.

Interleaved consecutive updates (2 issues). These issues are caused by consecutive updates to different rows: when interleaved with other update/delete transactions, this may leave the table in an inconsistent state, in which it contains partial data from one transaction.

Others (22 issues). For the remaining issues, we either cannot find the exact root causes from the online discussions, or find it hard to categorize them. For example, some reported bugs are still under investigation so there is no conclusion. Some can be fixed by "disabling a plugin," from which we don't know the exact root cause.

3.3 How Developers Fix Concurrency Bugs

In general, we find the fixes to concurrency bugs to be unexpectedly diverse and ad-hoc, and in some cases they even follow contradictory philosophies, indicating that developers are still struggling with the challenge of fixing concurrency bugs.

Select FOR UPDATE (22 issues). This semantic, which allows the developer to control which rows to lock, was introduced exactly to address anomalies caused by weak isolation. In our study, it is the most popular solution. However, keep in mind that this number may be biased by our dedicated search for "FOR UPDATE" (Section 3). While this approach should be able to address most of the data inconsistency issues, we observe that many developers still fix bugs in other ways, probably due to performance concerns, which we discuss later.

Use the uniqueness constraint (8 issues). This category addresses the problem by adding a uniqueness constraint to a certain column, declaring this column should have unique values. If this constraint is violated, the database will report an error. This approach is effective at addressing the "non-unique ID" problem.

Additional locking (11 issues). This category applies additional locking to prevent anomalies. They include using programming language semantics like "synchronized," using distributed locking services like Redis [4], or implementing a lock table inside the database. However, as some developers correctly point out, programming language–level locks are not sufficient in a distributed setting where multiple web servers access the database concurrently.

Additional versioning (8 issues). This category addresses the problem by implementing optimistic concurrency control (OCC) in the application: the fixed application adds a "version" column in the table, updates the version column when a transaction touches a row, and before committing checks that versions of the touched rows have not been modified.

Additional "if" check (4 issues). This category addresses the problem by rechecking the condition (e.g., item count is larger than 0) before committing the transaction. This approach is useful for addressing the RW1 problem under READ COMMITTED, since the update will lock the row until the end of the transaction. However, it cannot address the RW2 problem since the update/insert/delete could happen on different rows.

Queuing/serial execution (6 issues). This category addresses the problem by executing transactions serially. Since executing all transactions serially is unacceptable, some of these cases rely on an external queuing system (e.g., Redis): they put conflicting transactions in the same queue and non-conflicting transactions in different queues, so that non-conflicting transactions can execute in parallel.

Stronger isolation (7 issues). One of the cases upgrades the isolation level to SERIALIZABLE, and one upgrades isolation level from READ COMMITTED to REPEATABLE READ. The other five encapsulate a sequence of operations into an atomic transaction.

Weaker isolation (3 issues). Surprisingly, developers ultimately decide to either downgrade isolation levels or not encapsulate operations inside a transaction. In these cases, developers mainly try to address the long lock time or deadlocks, so using weaker isolation is reasonable. However, this approach may lead to new data inconsistency issues.

"Swallow" the error (3 issues). This category addresses the problem by using application-level exception handling to catch and ignore the error. It is useful to handle errors caused by concurrently deleting/inserting the same row.

Clear the effect manually (1 issue). There exists a long-standing argument that we may not need a technical solution to prevent concurrency bugs from happening [5]. Instead we can use business procedures to clear the effects of a concurrency bug after it happens. For example, if a shop oversells an item, it can cancel the order and send an apology email to the affected buyers, perhaps with some compensation. However, in our study, we find only one issue discussed this approach. We believe part of the reason is due to the characteristics of the open-source community. In this community, the developer and the user belong to different organizations, and if the user sees a problem like overselling, it is natural for her to think, "this is a bug that I need to report to the developer," rather than, "this is a problem I need to handle." On the developer side, pushing such problems to the user is probably not good for business. In big IT companies where the user and the developer are under the same organization, the situation might be different.

Others (20 issues). For the remaining issues, we either cannot find how developers fix the issue since the issue is still open, or cannot find a good category for the fix. For example, one solution alleviates the problem by removing the rate limit on transaction execution. Its key observation is that making transactions run faster can reduce the chance of contention.

3.4 Questions and Answers

Based on the results from this study, we try to answer the questions raised at the beginning of the paper.

Do weakly isolated or ad-hoc transactions actually cause anomalies in real-world applications? Quick answer: Yes. Considering the popularity of these solutions, whether they cause any real problems in terms of correctness has been a long-standing question [5]: perhaps they rarely cause any real issues, because real-world applications do not have a high contention level? Perhaps they do cause race condition–like phenomena, but they don't cause any real violations of application semantics?

From our study, we believe it's safe to conclude that many real-world applications are experiencing anomalies caused by weakly isolated or ad-hoc transactions. As a consequence, the users of these applications complain to application developers. The fact that popular open-source applications today often have a diversified set of users may exacerbate this problem, since the application must support various workloads.

How much effort does it require to handle anomalies in real-world applications? Quick answer: A lot. We separate this question into how much effort it requires to reproduce, diagnose, and fix a concurrency bug caused by weaker isolation.

We find that all issues that have a description about how to reproduce them can be reproduced by running one or two application functions (e.g., create an order, make a payment, etc), including the transactions they invoke. Developers usually need to run multiple instances of each function to increase concurrency. It's unclear to us whether this is because there are no complicated bugs or this is simply due to the fact more complicated ones are less likely to happen. Of course, such concurrent tests are nondeterministic, and require some "luck" to trigger the anomaly, but in general, we believe that developers are not struggling to reproduce anomalies.

By looking at the issues' discussions, diagnosing a concurrency bug usually does not take a lot of discussion. We believe this is because most of the bugs involve transactions from only one or two application functions and are caused by straightforward patterns (e.g., RW1 and RW2).

Fixing concurrency bugs, however, seems to be the most challenging and time-consuming part. First, as mentioned above, in contrast to the popular philosophy that we don't need a technical solution to prevent anomalies but can instead rely on business solutions [5], our study shows that, in the open-source community, users do expect a technical solution from developers. Second, we find that many of these issues lead to lengthy discussions, which involve multiple rounds of discussions since proposed solutions in the early rounds are found to be flawed. They often lead to significant redesign of the applications and complicated solutions.

Why not SERIALIZABLE or "Select FOR UPDATE"? Quick answer: Long locking. While a general answer to this question is that they are bad for performance, "performance" can refer to many different things, such as throughput, latency, and scalability. This question is particularly troublesome since some of the fixes, like serial execution, are usually considered bad for performance as well.

While many discussions do not explicitly discuss this question, among those that do, we find there is a general concern about deadlock or long locking time:

——"In my experience dealing with money for 4 years is that whatever locking mechanic you try to use, it goes wrong anyway. ... In any case, I had dealt with too many deadlock situations in MySQL due - probably when PHP scripts died due to fatal error or some other problem and your database left with the lock active." [7]

——"A common workaround here is to adjust the transaction-isolation = READ COMMITTED rather than using the default repeatable read isolation. This relaxes the locking required by InnoDB." [6]

——"You can use Locking statement as [another discussant] mentioned, for me it has caused more issues like ... longer transaction time and the dreaded 'Concurrent Limit Exceeded'" [10]

The discussion in the WooCommerce application [8] further demonstrates the problem of locking: some developers suggested trying different lock timeouts from a minute to an hour, while one developer argued, "it is a poor approach to architect software based on guessing deadlock timeouts." Finally the developers turned to a queuing-based approach.

From these discussions, our best answer to "why not SERIALIZABLE or Select FOR UPDATE?" is that, since these approaches increase locking time and the chance of deadlocks, the developers struggle to set up an appropriate lock timeout value: using a short timeout may cancel normal transactions; using a long timeout may delay the detection of deadlock or dead processes. As a result, developers are willing to turn to other approaches to avoid this problem.

4 WHAT CAN WE DO?

From our study, we conclude that, for many applications, the current database concurrency control mechanisms are unsatisfactory: on the one hand, SERIALIZABLE transactions often generate performance concerns; on the other hand, weakly-isolated or ad-hoc transactions require a significant amount of effort and are error-prone.

In this section, we discuss potential solutions. Overall we argue that, although databases and automated analyses can ameliorate concurrency bugs to some extent, concurrency control is likely to be largely developers' responsibility for the foreseeable future. In the short term, developers can use certain existing database features to address many concurrency issues. In the long term, some combination of new database features that expose concurrency control to developers and better education of developers can help address long-standing challenges in using concurrency control.

Q1. Is there anything users can do in the short term?

Given that significant redesigns of the database engine will not happen in the short term, we provide suggestions that users may incorporate in the meantime.

(1) All RW1 issues can be addressed by using the SNAPSHOT ISOLATION isolation level, since transactions of RW1 issues will update the same data item.
(2) Many RW2 issues can be addressed by using a unique constraint or the auto increment feature, since the corresponding transactions usually try to ensure certain IDs are unique. However, we observe two engineering challenges for this approach. First, many applications today use an object–relational mapping (ORM) layer to manage their data, instead of issuing SQL statements directly. In this case, using a unique constraint or the auto increment feature requires changes to the ORM layer, which is beyond the application developers' control. Second, different database vendors may have different keywords for these features, so relying on such features may create problems for portability. Furthermore, there exist RW2 issues that cannot be addressed by unique constraints or the auto increment feature, though they do not appear frequently in our study. A typical example is the write skew problem.
(3) For timeout issues caused by long locking or deadlock, we argue it is better to have a fundamental solution, rather than tuning timeout intervals or degrading isolation levels. For example, an OCC-based database may be preferable to a locking-based database. The fact that several applications in our study manually implement OCC is evidence for this point. This suggestion aligns well with suggestion (1) above (i.e., use SNAPSHOT ISOLATION for RW1 issues) since a common implementation of SNAPSHOT ISOLATION uses multi-versioning and OCC. For

locking-based solutions, using a database that implements more robust deadlock detection mechanisms than a simple timeout may be worthwhile.

(4) Even the SERIALIZABLE isolation level has its limitations. There are problems that have to be handled correctly at the application level. For example, exceptions caused by duplicate inserts or deletes must be handled properly (Section 3.2). Developers should not assume that a strong isolation level is a panacea that addresses all concurrency related issues.

Q2. How much can automatic analysis help?

Multiple works have shown that, by analyzing the patterns of transaction statements either at run time or offline, it is possible to make a SERIALIZABLE database more efficient [29, 30]. We envision such analyses can be performed both within the database and on database applications.

For example, suppose an application only contains two types of transactions: transaction T1 executes "Write(A); Read(B)" and transaction T2 executes "Write(A); Read(C)". A smart developer can determine that no matter how these transactions interleave, the result is SERIALIZABLE (assuming the writes to A are indivisible). However, today's lock-based databases will prevent them from executing concurrently, since Write(A) needs to hold a write lock on A until the end of the transaction; today's OCC-based databases will abort one transaction since both update A. This situation could be improved with automatic analysis.

First, databases can perform more accurate analysis during transaction execution, to determine whether a concurrent execution is SERIALIZABLE. However, more accurate online analysis usually incurs a higher overhead. Furthermore, without access to the source code of the application, the database cannot make any assumptions about the patterns of transactions, which means it may miss optimization opportunities for specific applications. Consider again the above example: existing database implementations will prevent concurrent execution of T1 and T2 as discussed above. To improve the situation, recent work has explored using predicate locking and satisfiability testing for transaction scheduling [17]. Other recent works build a dependency graph at run time; they determine on the fly that executing T1 and T2 concurrently does not violate SERIALIZABLE, since the dependency graph contains no cycles [15, 30]—which is more accurate than the existing concurrency control mechanisms but will incur higher run-time overhead.

Second, automated analysis of *application code* can potentially determine that concurrency control is completely unnecessary for executing T1 and T2. While prior work has applied static program analysis to database applications to understand their behavior under various isolation levels [13, 22, 25], to our knowledge prior work has not used static analysis to automatically infer concurrency control requirements. For example, could we design an algorithm to analyze the application code to know that T1 and T2 can execute concurrently with each other and cannot produce a non-SERIALIZABLE execution? In practice, however, we observe several challenges with existing and future static analyses for database applications: 1) Sound static program analysis struggles to be precise; it will overapproximate the behavior of transactions and the transactions they are likely to be concurrent with. 2) Web applications are often written in dynamically-typed languages such as JavaScript and PHP. Compared with Java and C/C++, these languages are harder to analyze with precision, and the analysis tools are not as advanced. 3) In our experience, parsing SQL statements is already a challenging task, since different databases often have vendor-specific grammar or keywords that deviate from the SQL standard, and many of the vendor-specific keywords are related to concurrency control and thus cannot be ignored. 4) Most existing analysis tools rely on an abstract model to determine allowed behaviors of a certain isolation level, while a database implementation often puts more constraints on such allowed behaviors. It is debatable whether it is better to analyze these problems based on a generic model or based on an implementation-specific model.

On the other hand, we also observe some opportunities: 1) Web applications are often smaller in terms of code size than other software on the stack such as the database and the OS kernel, and thus stress analysis scalability less. 2) By their distributed nature, concurrent web requests usually do not share data in memory (shared data is stored in the database). This should simplify static program analysis, which struggles to represent heap objects precisely.

In summary, we envision that combined analysis of SQL and web application code presents open research challenges and opportunities for achieving better concurrency control.

Q3. Is it possible to completely automate concurrency control? If not, how can concurrency bugs be presented to developers of application to be understandable? How can one incorporate insights from developers?

Even if automatic analysis can improve concurrency control significantly, developer effort is still likely to be valuable, since developers have knowledge about application-specific constraints and invariants. For example, certain types of transactions may not need SERIALIZABLE results; an application may limit each customer to a single shopping cart so that there will never be concurrent orders from the same customer; some applications may allow the customer to see an inconsistent result temporarily (e.g., payment is made but order status is not updated to show the payment), as long as the customer can refresh to see the correct result later. Such additional invariants or relaxed requirements often allow developers to craft transactions in a more efficient manner.

In addition, there is potential for improved performance in situations like updating hot records: Whereas a database system today needs to infer contention, a more harmonious interaction would allow users to convey which records are likely to be hot. Such a declaration from the user could trigger more aggressive operation batching (to minimize lock ping-pong effects among concurrent writers) and operation reordering if supported by program analysis (to shorten the write lock holding time). It is very hard, if not impossible, for any algorithm to figure out such application-level constraints or knowledge automatically.

Therefore, we believe that, to get the best performance, some effort from developers will always be necessary. The research opportunity is to create new user introspection and control mechanisms for concurrency control. The major barriers towards this goal are 1) how to present concurrency bugs in a manner that is understandable by users, and 2) how to incorporate user feedback and automatically suggest modifications to SQL or application code to fix the concurrency bugs.

Recent research works have taken promising first steps to identify [18, 20] and visualize [2, 26] concurrency bugs. These works present isolation violations in a manner that assumes a deep understanding of serializability theory, such as by presenting a dependency graph and the associated operation sequence that minimally reproduces an observed concurrency bug. The research challenge is presenting possible isolation violations in an application-aware manner, possibly using natural language. Consider the distance between "below is a minimal dependency graph *G* that reproduces the concurrency bug" and "the ORDER transaction allows the same customer to create two orders with the same order ID, but the PAYMENT transaction will only find the first order with this ID and ignore the other". The former presentation assumes combined expertise in serializability theory and application-specific logic, while the latter does not.

Database systems today provide multiple ways users can control concurrency control behavior, including additional SQL keywords (e.g., FOR UPDATE), table- and row-level locking, and save points. Prior studies have shown that applications already use these mechanisms extensively [27]. The research gap is automating the use of these mechanisms to resolve a specific concurrency bug. A naive approach is to take a trial-and-error approach where a list of modifications is progressively applied, while a concurrency bug checker from prior work will test if the concurrency bug still manifests. A promising research direction is developing algorithms to automate this process and point out which modifications will not work and why. Ideally, this procedure can take into account vendor-dependent nuances of the implementation of concurrency control.

We acknowledge that any degree of application control of the concurrency control protocol will inadvertently lead to undesirable side effects. For example, an application using a new mechanism to control concurrency may break the isolation of a traditional database transaction that has not taken this new mechanism into consideration; an application may also attempt to weaken isolation in a manner that breaks the database's rollback support for all transactions. As a result, introducing new mechanisms that control the behavior of concurrency will naturally raise new research questions. As examples, how can transactions written with the assumption of explicit control of locks coexist with traditional transactions? When can a database system guarantee automatic rollback support if applications can control the locking protocol? We thus argue that creating the appropriate mechanisms is a major research challenge, and we foresee that adding such mechanisms into existing database systems will encounter formidable technical and practical challenges.

Q4. How can developer education help?

During the study, we find a majority of developers are more accustomed to controling concurrency at the programming language level than at the database level. For example, many developers use the term "race condition" for isolation errors, and their first reaction to such issues often is, "we need to add a lock." This is perhaps not surprising because 1) locks are widely taught at the undergraduate level, while weaker isolation levels are not, and 2) though locks are perhaps just as hard to get right, they are easier to understand.

We observe two major challenges to improve education on these topics. First, early definitions of isolation levels are operational (e.g., under READ COMMITTED, a write lock is held until the transaction ends, while a read lock is held only until the read ends), which makes them easy to grasp especially for people who are already familiar with locks, but dictates a particular implementation mechanism. Unfortunately, more formal dependency graph–based definitions of isolation, though declarative and general, are harder to explain and even harder for developers to implement to analyze real transactions. Second, despite the best efforts by the research community to devise accurate definitions, database implementations continually evolve their isolation semantics and may deviate from the established definitions, further confusing users. We don't see an easy path to address this problem, which requires a collaborative effort between research and industry practice.

5 CONCLUSION

To understand the challenges of concurrency control in databases, this paper first studies real-world concurrency bugs in database applications to understand their root causes, consequences, and how developers fix them. Based on this study, we believe that to get the best performance, developers' effort is necessary, and thus databases need new abstractions and mechanisms to incorporate developers' effort. In addition, automated code analysis shows promising potential to improve the state of the art.

ACKNOWLEDGMENTS

We thank all reviewers for their insightful comments. This material is based in part upon work supported by Oracle America, Inc. and by the National Science Foundation under Grant Numbers CNS-1908020, CCF-2118745, and CSR-2106117.

REFERENCES

[1] DB Study Result. https://go.osu.edu/isolation-bug-study.
[2] Elle: Demo. https://github.com/jepsen-io/elle#demo.
[3] Hermitage: Testing transaction isolation levels. https://github.com/ept/hermitage.
[4] Redis. https://redis.io/.
[5] Understanding Weak Isolation Is a Serious Problem. http://www.bailis.org/blog/understanding-weak-isolation-is-a-serious-problem/.
[6] MySQL deadlock on catalogsearch_query. https://magento.stackexchange.com/questions/45845/mysql-deadlock-on-catalogsearch-query/45850#45850, Nov 2014.
[7] Race conditions in inventory tracking, order, payment status... 2776 . https://github.com/Sylius/Sylius/issues/2776, May 2015.
[8] Implement locking mechanism for data updates... 15780 . https://github.com/woocommerce/woocommerce/issues/15780, Jun 2017.
[9] Orders overwriting when placed at same time 17660 . https://github.com/woocommerce/woocommerce/issues/17660, Nov 2017.
[10] future method concurrency. https://salesforce.stackexchange.com/questions/219118/future-method-concurrency/219128#219128, May 2018.
[11] Duplicate products with identical SKUs created via API 27295 . https://github.com/woocommerce/woocommerce/issues/27295, Sep 2019.
[12] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering*, 2000.
[13] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. Static serializability analysis for causal consistency. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 90–104, New York, NY, USA, 2018. Association for Computing Machinery.
[14] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *PODC '17*.
[15] Dominik Durner and Thomas Neumann. No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *ICDE 2019*.

[16] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making Snapshot Isolation Serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[17] Kevin P. Gaffney, Robert Claus, and Jignesh M. Patel. Database isolation by scheduling. *Proc. VLDB Endow.*, 14(9):1467–1480, oct 2021.

[18] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. IsoDiff: Debugging Anomalies Caused by Weak Isolation. In *VLDB 2020*.

[19] Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the Detection of Snapshot Isolation Anomalies. In *VLDB '07*.

[20] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. In *VLDB 2020*.

[21] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 329–339, New York, NY, USA, 2008. Association for Computing Machinery.

[22] Kartik Nagar and Suresh Jagannathan. Automated detection of serializability violations under weak consistency, 2018.

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

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

[25] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. Clotho: Directed test generation for weakly consistent database systems. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[26] Drew Ripberger, Yifan Gan, Xueyuan Ren, Spyros Blanas, and Yang Wang. Isobugview: Interactively debugging isolation bugs in database applications. *Proc. VLDB Endow.*, 15(12):3726–3729, 2022.

[27] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. Ad hoc transactions in web applications: The good, the bad, and the ugly. In *SIGMOD '22*.

[28] Todd Warszawski and Peter Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *SIGMOD '17*.

[29] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via Modular Concurrency Control. In *SOSP '15*.

[30] 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. In *VLDB 2016*.