2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00219

# Database Deadlock Diagnosis for Large-Scale ORM-Based Web Applications

Zhiyuan Dong*, Zhaoguo Wang*, Chuanwei Yi*, Xian Xu*, Jinyuan Zhang*, Jinyang Li†, Haibo Chen*
*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
†Department of Computer Science, New York University

***Abstract*—Today, most database-backed web applications depend on the database to handle deadlocks. At runtime, the database monitors the progress of transaction execution to detect deadlocks and abort affected transactions. However, this common detect-and-recover strategy is costly to performance as aborted transactions waste CPU resources.**

**To avoid deadlock-induced performance degradation, developers aim to reorganize the application code to remove deadlocks. Unfortunately, doing so is difficult for web applications. Not only do their implementations include hundreds of thousands of LoCs, but they also use third-party object-relational mapping (ORM) frameworks which hide database access details. Consequently, it is hard for developers to accurately diagnose deadlocks.**

**We propose WeSEER, a deadlock diagnosis tool for web applications. To overcome the opacity of ORMs, WeSEER performs concolic execution on unit tests to extract a web application's transactions as a sequence of template statements with symbolic inputs as well as path conditions that enable the sequence. WeSEER then analyzes the extracted transactions based on fine-grained lock modeling to identify potential deadlocks and report the code locations that cause them. We implement WeSEER for Java-based (OpenJDK) web applications, and use it to analyze two popular open-source e-commerce applications, *Broadleaf* and *Shopizer*. WeSEER has successfully identified 18 potential deadlocks in Broadleaf and Shopizer. Eliminating these identified deadlocks can result in up to 39.5× and 4.5× throughput improvement for *Broadleaf* and *Shopizer*, respectively.**

***Index Terms*—database locking, deadlock diagnosis, concolic execution**

## I. Introduction

Nowadays, web applications often use database systems to manage their data and protect critical business logic with database transactions. Most commercial databases leverage locks to provide isolation among concurrent transactions. When transactions wait to acquire the locks of resources (e.g., database tables or rows) already held by others in a circular hold-and-wait pattern, they incur deadlocks [1].

Existing databases, including MySQL [2], PostgreSQL [3], and SQL Server [4], commonly adopt the detect-and-recover strategy to handle deadlocks. As a transaction executes, the database starts a hold-and-wait cycle detection procedure if other transactions block its statement. If a cycle is found, the database will roll back one of the involved transactions to break the cycle. Although effective, this method is detrimental to performance as aborted transactions waste the processing power of both the application and the database.

To avoid performance degradation, web application developers try to eliminate deadlocks on the application side [5], [6]. For instance, they could rewrite or reorganize part of the application code or introduce application-level locking to avoid deadlocks. However, doing so requires developers to be able to first identify the code, which may cause a deadlock. Unfortunately, such deadlock diagnosis is highly non-trivial for real-world web applications due to two challenges.

First, web applications often have a large code base today. For example, the ten most popular open-source web applications based on the number of GitHub stars have 160K LoC on average, ranging from 50K to 406K LoC. Thus, it is laborious to manually comb through the code for possible deadlocks [5].

Second, most web applications use the object-relational mapping (ORM) frameworks to access databases in the object-oriented programming paradigm [7]–[10]. ORM frameworks automatically translate operations on objects into database statements and hide the details of database access for optimization [11]. On the one hand, ORM may merge multiple operations into a single statement. On the other hand, ORM may also buffer statements at the application side and issue them to the database in a batch, to reduce network roundtrips. Existing works require manually or statically extracting transaction logic from application code [12], [13], which is impractical.

We propose WeSEER, a deadlock diagnosis tool for large-scale ORM-based web applications. To cope with the barrier of ORM, WeSEER performs concolic execution on a web application's API unit tests to extract transaction statements. Concolic execution combines symbolic and concrete execution in which some of the program variables are marked as symbolic, and their symbolic values can be propagated to other variables through symbolic execution. Using concolic execution, WeSEER can collect a trace of transaction statements with symbolic inputs and path conditions that enable the trace. Afterward, WeSEER performs an offline analysis of the traces to identify potential deadlocks and their triggering conditions. Unlike prior work [12]–[16] that identifies transaction conflicts or deadlocks based on table-level locks, WeSEER extracts conflict conditions that will lead to deadlocks assuming fine-grained row-level locks. Using an SMT-solver, WeSEER solves the path and conflict conditions necessary for deadlocks and prepares a detailed report on the deadlock conditions and code locations.

We have prototyped WeSEER for Java-based web applications. Our implementation of WeSEER's concolic execution engine is based on OpenJDK8. WeSEER's deadlock analyzer is written in Java using the Z3 SMT solver. We use WeSEER to analyze two large-scale open-source e-commerce web applications, Broadleaf [17](190K LoC) and Shopizer [18](92K LoC), and have identified 18 previously unknown deadlocks. We reproduce the identified deadlocks in experiments and showed that the performance of Broadleaf and Shopizer can be improved by up to 39.5× and 4.5× once the deadlocks are removed by fixing code on the application side.

In summary, our contributions are as follows:

- A deadlock diagnosis tool that could extract transaction statements from ORM-based web applications using concolic execution, and identify potential deadlocks.
- A concolic execution engine based on OpenJDK, that could run large-scale Java web applications.
- A deadlock analyzer that models fine-grained row-based locking to analyze transctions' conflict conditions that could lead to actual deadlocks using the SMT solver.
- An evaluation using two real-world, large-scale web applications, *Broadleaf* [17] and *Shopizer* [18]. WeSEER is able to detect 18 deadlocks, and removing them can lead to significant performance improvements (up to 39.5× for *Broadleaf* and up to 4.5× for *Shopizer*).

## II. Motivation & Challenges

### A. Eliminating Deadlocks in the Application Code

Today, most applications rely on the underlying database to handle the deadlock. Commercial database systems, including MySQL [2], PostgreSQL [3], and SQL Server [4], adopt the detect-and-recover scheme to handle deadlocks. When a transaction needs to wait for locks held by other transactions, the database will attempt to detect deadlocks by finding hold-and-wait cycles among transactions or employing other mechanisms such as wait-die, wound-wait, and timeout. If a potential deadlock is found, the database will choose a victim transaction to abort. However, such a deadlock recovery strategy usually incurs high performance overhead as the CPU resources spent by victim transactions are wasted.

To avoid performance degradation due to deadlocks, developers often attempt to remove potential deadlocks by rewriting relevant parts of the application code [5], [5], [6], [6]. Since manual deadlock diagnosis is a serious burden, existing works such as STEPDAD [12] and REDACT [13] try to diagnose deadlocks automatically. However, these works require developers to manually extract each transaction's SQL statements from the application code. After extraction, they use graph-based algorithms to discover potential hold-and-wait cycles statically. The diagnostic information can be used for test case generation [12] or runtime deadlock prevention [13]. However, these existing works have limited practicality because modern web applications have very large codebases that rely on ORM (Object Relational Mapping) frameworks to interact with the database. As such, it is impractical to manually extract transaction statements from web applications.

### B. ORM frameworks obscure transaction details

Modern web applications access the database using ORM frameworks [7]–[10], which expose the abstraction of *persistent objects*. Each object is mapped to a database row and the ORM translates object accesses to SQL statements. The resulting ORM-generated statements are often complex and difficult to understand [19]. For example, many statements contain multiple JOINs, in order to merge results from different tables. Also, statements can contain table and column aliases which further complicate understanding.

```
CREATE TABLE Order {
  ID int, PRIMARY KEY (ID) };
CREATE TABLE Product {
  ID int, QTY int, PRIMARY KEY (ID) };
CREATE TABLE OrderItem {
  ID int, O_ID int, P_ID int, QTY int,
  PRIMARY KEY (ID),
  FOREIGN KEY (O_ID) REFERENCES Order(ID),
  FOREIGN KEY (P_ID) REFERENCES Product(ID) };
```

```
SELECT * FROM OrderItem oi
  JOIN Order o ON o.ID == oi.O_ID
  JOIN Product p ON p.ID == oi.P_ID
  WHERE oi.O_ID == ?; /* Q4 */
UPDATE Product SET QTY=? WHERE ID=?; /* Q6 */
```

```
@Transactional
void finishOrder(Long orderId) {
  if (orderId == -1) return;
  // o is read from read cache
  Order o = orm.find(Order.class, orderId);
  // Order Items are loaded lazily
  for(OrderItem oi:o.getOrdItems()){//Trigger Q4
    ... // Select Q5 sent to DB
    updateQuantity(oi);
  }
  // Update Q6 sent to DB
}
void updateQuantity(OrderItem oi) {
    Product p = oi.getProduct();
    int p_qty = p.getQty();
    int oi_qty = oi.getQty();
    if (p_qty >= oi_qty) {
      // Updates are buffered
      p.setQty(p_qty - oi_qty); // Trigger Q6
    } else {
      throw new Error("No enough products"); }}
```

Fig. 1: Broadleaf's *finishOrder* function. The example includes the simplified table schema, ORM-based application code, and the ORM-generated SQL templates. The SQL statements Q4, Q5, and Q6 correspond to the 4th, 5th, and 6th statements sent by *finishOrder*.

ORM frameworks also obscure transaction logic through optimization: 1) They use **read cache** to buffer data for future reads. 2) They use **write-behind cache** to batch updates to the same object. In particular, the ORM defers writes and only flushes a transaction's dirty objects when the transaction commits or the application invokes flush operation. 3) They perform **lazy loading** so an object is not immediately fetched from the database upon its read, but only upon the actual use of the object. Consequently, the order of statements issued can be different from the order of operations on the object.

Because of ORM, it is impractical to statically (or manually) extract transaction statements. We illustrate the challenges involved using an example (Fig. 1), which is derived from the open-source e-commerce application Broadleaf [17]. The application function *finishOrder* finishes the processing of an order and is protected by a database transaction declared by the *Transactional* annotation (Line 1). First, the application uses *orm.find()* to read the target order (Line 5). Because persistent *Order o* has already been fetched outside the transaction, it is stored in the read cache, and no statements are sent to the database. Further, the ORM framework applies lazy loading to $o$'s list of order items. Thus, at the beginning of the loop (Line 7), Q4 is issued to fetch the target order items implicitly. In this example, only one order item is fetched, and the loop body is executed only once. Before *updateQuantity* is invoked, Q5 is sent by other application logic.

For any given *OrderItem oi*, *updateQuantity* checks and decreases the remaining quantity of *Product p* with the required quantity of *oi*, where *oi*'s foreign key is *p*'s primary key. As *p* is already fetched by Q4, no statements are sent due to read caching (Line 14). *p.setQty(...)* modifies the remaining quantity of *p* when *p*'s quantity is enough. Otherwise, the transaction is aborted, and the API returns an error to the client. Please note that, with write-behind cache, the ORM framework defers Q6 for *p*'s modification and implicitly sends Q6 before the transaction tries to commit (Line 11). As the code of Line 19 triggers Q6, we call it Q6's triggering code.

### C. Challenges of Deadlock Diagnosis in Modern Web Apps

Identifying deadlocks in real-world ORM-based web applications has the following challenges: First, it is difficult to statically extract transaction statements from the application source code. To analyze potential deadlocks, it is an important first step to identify the set of transactions used by the applications. However, it is difficult to infer the transaction statements of modern web applications based on static information, due to three problems: 1) Static program analysis, such as symbolic execution [20] and model checking [21], does not scale to large scale applications with tens or hundreds of thousands lines of code, because these techniques can require enumerating all code paths, 2) Static approaches can be inaccurate because not all object accesses will be translated into SQL statements due to ORM caching. 3) Existing static approaches [12], [13] assume that, for each code path, the execution order of SQL statements is the same as the programming order of data access operations. Unfortunately, this assumption does not hold because of ORM's lazy loading and write-behind cache mechanisms, which defer the execution of SQL statements.

Second, when analyzing deadlocks, using coarse-grained lock modeling simplifies analysis but results in increased false positive rates. Coarse-grained locking is popular among existing deadlock diagnosis solutions [12], [13], which report potential conflicts among transactions if their statements access a common database table and at least one of them writes to the table. In other words, coarse-grained lock modeling conservatively assumes the statements acquire table-level locks. However, because real-world databases typically use fine-grained row-level locks, coarse-grained locking is unacceptable due to false positives.

Third, it is challenging to map a deadlock to its source code location. The diagnostic tool must help developers identify the source code which triggers deadlock-prone SQL statements. However, due to ORM's write-behind cache, the code sending SQL statements may not be the code triggering them. For example, Q6 is sent in Line 12, while its triggering code is Line 19. Furthermore, an SQL statement may be triggered by normal object accesses rather than ORM operations. Thus, we cannot identify the triggering code by simply recording the stack trace and the invocation sequence of ORM operations.

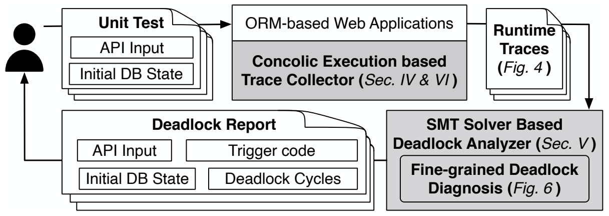

Unit Test
API Input
Initial DB State
ORM-based Web Applications
Concolic Execution based Trace Collector (Sec. IV & VI)
Runtime Traces (Fig. 4)
Deadlock Report
API Input
Trigger code
Initial DB State
Deadlock Cycles
SMT Solver Based Deadlock Analyzer (Sec. V)
Fine-grained Deadlock Diagnosis (Fig. 6)

Fig. 2: WeSEER's Architecture. Gray blocks represent the components provided by WeSEER. The developers run the web applications with unit tests, which specify the target API, API input, and initial database state. During the execution, the trace collector collects the runtime traces and feeds them to the deadlock analyzer. After identifying the deadlocks, the deadlock analyzer reports the deadlocks, with information including the involved API, inputs, initial DB state, SQL statements, and triggering code location that causes the deadlocks.

## III. Our Approach

We propose WeSEER, the first diagnostic tool for identifying database deadlocks for modern ORM-based web applications. In this section, we give an overview of WeSEER's approach and its key innovation. In the following sections (Sec IV—VI), we delve into the details of the components.

**WeSEER's architecture.** Fig. 2 shows the overall architecture of WeSEER, which consists of a dynamic trace collector and an offline deadlock analyzer. The trace collector extracts transaction statements and their path conditions by running an application's unit tests using concolic execution [22]–[24]. Based on the collected trace, the deadlock analyzer encodes the path and fine-grained lock conflict information for the SMT-solver to precisely identify the conditions of a deadlock. WeSEER also reports rich information for developers to understand and reproduce the deadlock.

### A. Extracting transactions using concolic execution

**Preliminaries on concolic execution.** Concolic execution [22]–[25] is a hybrid program analysis technique that combines symbolic and concrete execution. A concolic execution engine maintains 1) a symbolic store that maps each variable to its symbolic expression, and 2) path conditions encoded in the first-order logic formula to track the branches taken on the execution path. By assigning symbolic expressions to target program variables, concolic execution models all possible states of these variables in the execution path, determined by the path conditions. Consider integer variable $a$ with concrete value 1 and symbolic alias $sym_a$. After executing $b = a + 1$, the engine assigns the concrete value 2 to $b$ and updates its symbolic value to $sym_a + 1$. Furthermore, assume that a branch statement $if(b == 8)$ is executed with the else branch being taken, then $sym_a + 1 \neq 8$ is recorded as the path condition.

**WeSEER's use of concolic execution.** Concolic execution is a form of dynamic program analysis. In detail, WeSEER uses concolic execution to run a web application's unit tests in order to extract transaction statements. WeSEER's concolic engine provides three interfaces: *start_concolic()*, *end_concolic()*, and *make_symbolic(variable)*. To prepare unit tests for concolic execution, programmers 1) use *start_concolic* and *end_concolic* to wrap a code section where concolic execution should be enabled, and 2) use *make_symbolic(variable)* to mark inputs to web application's API as symbolic.

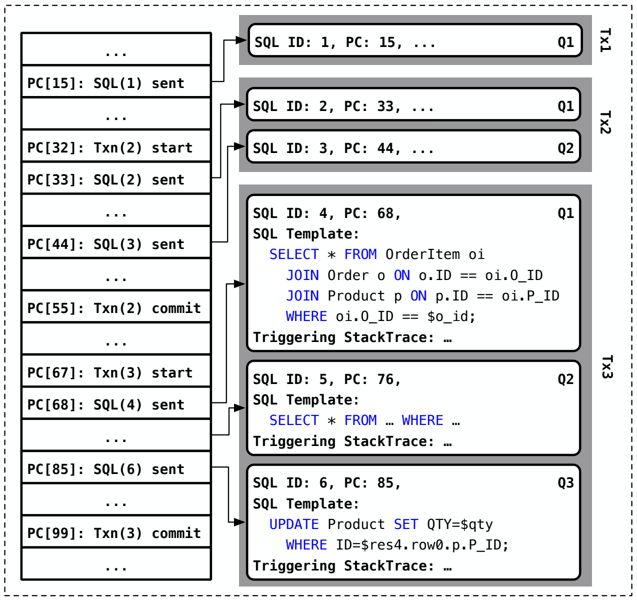
We extend WeSEER's concolic execution engine to dynamically track database operations (transaction begin/commit/abort and SQL statements) and record them in traces (Sec. IV-A). This dynamic approach allows us to sidestep the opaqueness and complexity of ORM. During concolic execution, WeSEER (1) marks the API inputs and SQL results (which reflect the database state) as symbolic variables to track their data flows; (2) records the execution's control flow as path conditions; (3) records the SQL statements (or templates) and SQL parameters' symbolic values. The resulting trace represents all possible states of API input and database states for the analyzed code path. They are then used by WeSEER's offline deadlock analyzer. Fig. 3 shows an example trace.

## B. Fine-grained deadlock diagnosis using an SMT Solver

Given the traces of extracted transactions, the deadlock analyzer aims to identify conditions that can lead to actual deadlocks. In particular, WeSEER assumes fine-grained row-level locking. It encodes transactions' path conditions and conflict information as first-order logic formula and uses an SMT-solver to determine how deadlocks can arise.

**Preliminaries on SMT solving.** Satisfiability Modulo Theories (SMT) solvers [26]–[30] are tools for determining the satisfiability of first-order logic formulas. For a given formula, the SMT solver may output 1) SAT (satisfiable) together with an arbitrary satisfying assignment of the formula's variables, 2) UNSAT (unsatisfiable), or 3) a timeout. Suppose the formula is a conjunction of path conditions: $(sym_a+1 \neq 8) \wedge (sym_a > 3)$ where symbolic alias $sym_a$ is a variable. The SMT solver may output the satisfying assignment $sym_a == 4$. If we change the formula to $(sym_a + 1 \neq 8) \wedge (sym_a == 7)$, then the SMT solver would return UNSAT. SMT solver and concolic execution are usually used together [25]. SMT solver is used to find a (concrete) satisfying assignment of program variables among the possible states modeled by the recorded symbolic expressions. WeSEER uses Z3 [27] for SMT solving.

**WeSEER's use of SMT for deadlock diagnosis.** Like existing approaches [12], [13], WeSEER's deadlock analyzer discovers potential hold-and-wait cycles by analyzing the conflict pattern of transactions at the coarse-granularity of tables (Sections V-A and V-B). However, unlike existing work, we make WeSEER's deadlock diagnosis more precise by further checking the trigger conditions of these potential deadlocks (Sec. V-C). To perform this check, the analyzer



```
...
PC[15]: SQL(1) sent
...
PC[32]: Txn(2) start
PC[33]: SQL(2) sent
...
PC[44]: SQL(3) sent
...
PC[55]: Txn(2) commit
...
PC[67]: Txn(3) start
PC[68]: SQL(4) sent
...
PC[85]: SQL(6) sent
...
PC[99]: Txn(3) commit
...
```

```
Tx1
SQL ID: 1, PC: 15, ...                    Q1

Tx2
SQL ID: 2, PC: 33, ...                    Q1
SQL ID: 3, PC: 44, ...                    Q2

Tx3
SQL ID: 4, PC: 68,                        Q1
SQL Template:
  SELECT * FROM OrderItem oi
    JOIN Order o ON o.ID == oi.O_ID
    JOIN Product p ON p.ID == oi.P_ID
    WHERE oi.O_ID == $o_id;
Triggering StackTrace: …

SQL ID: 5, PC: 76,                        Q2
SQL Template:
  SELECT * FROM … WHERE …
Triggering StackTrace: …

SQL ID: 6, PC: 85,                        Q3
SQL Template:
  UPDATE Product SET QTY=$qty
    WHERE ID=$res4.row0.p.P_ID;
Triggering StackTrace: …
```

Fig. 3: The runtime trace of Fig. 1's *finishOrder* transaction's API. The trace contains 1) transaction-related information (SQL template and parameters) and path conditions, 2) necessary information for deadlock reporting, such as the mapping from reordered statement to the code location (stack trace) of its triggering code. *res4.row0.p.ID* is the *p.ID* column of Q4's database result's first row.

generates the conflict conditions for each conflict edge of the hold-and-wait cycle including predicates constructed from SQL parameters and query conditions (Sec. V-C4). Doing so requires modeling which database index(es) may be used for execution (Sec. V-C2) and how locks are acquired at the granularity of rows (Sec. V-C3). The analyzer encodes the conjunction of conflict conditions and path conditions related to the hold-and-wait cycle into a first-order logic formula. The formula is checked for satisfiability by an SMT solver. If the solver reports SAT, then the analyzer confirms the deadlock with a satisfying assignment of the API input and database state that can be used to reproduce this deadlock. If the solver reports UNSAT or timeout, then no deadlock is reported.

## C. Mapping deadlocks to their triggering code

For preparing the deadlock diagnostic report, WeSEER's analyzer uses an ORM-aware tracking mechanism to map a SQL statement to its triggering code. To do so, we model ORM operations and handle them differently. Combined with stack traces recorded during concolic execution, WeSEER can identify the triggering code locations for SQL statements involved in the deadlocks.

## IV. Making Concolic Execution Practical for Collecting Transaction Traces

It is non-trivial to apply concolic execution to modern web applications because of their large and complex code base. Each execution can encounter a large number of branches, resulting in numerous path conditions. For example, executing the unit test of *Broadleaf*'s *Ship* API (shown in Table I) results in 656K path conditions, which cause timeouts in SMT solvers. To simplify concolic execution, we observe that many path conditions are unrelated to the application logic and can be pruned. In particular, we find that the database driver, the language's built-in classes, and container classes contribute the most to unnecessary path conditions. Thus, our main strategy for simplification is to disable concolic execution and only rely on concrete execution when running the functions of these libraries and classes. If the "ignored" function takes a symbolic input, the engine generates a new symbolic variable to represent its output. There are no path conditions that relate the function's input and output symbolic variables. After simplification, the number of path conditions of *Broadleaf*'s *Ship* API's unit test is reduced from 656K to 2.7K.

### A. Handling Database Driver Functions

Database drivers send SQL statements to the database and parse query results. Their internal logic is not related to the application logic and should be ignored for concolic execution. However, the trace collector needs to track their use to record transactions' life cycles and SQL statements. Our trace collector works for the popular ORM (JDBC [31] and ODBC [32]), and handles four kinds of database driver functions: (1) Functions that start and finish a transaction. WeSEER monitors them to infer each transaction's life cycle. (2) Functions that prepare SQL statements using the input SQL templates and parameters. WeSEER records the SQL templates and (symbolic) SQL parameters passed to the preparation methods. WeSEER also associates the prepared SQL statement with its corresponding transaction. (3) Functions that submit the SQL statements and return an object representing the database result. Once these methods are called, the corresponding SQL statement's information is recorded into the trace. (4) Functions that retrieve the values from a given database result object. WeSEER assigns symbolic aliases to their return values, representing the fetched database state.

### B. Handling Built-in classes

WeSEER's concolic execution engine works for Java whose language built-in classes include String and BigDecimal (high-precision number). Their implementation's internal logic is complex, causing numerous path conditions under concolic execution. Instead of completely ignoring them, we model the semantics of String and BigDecimal by approximating them using simpler alternatives. In particular, we model BigDecimal as Z3's floating numbers because float numbers are capable of supporting the numeric ranges used in the unit tests. In other words, our execution engine maps the functions of high-precision numbers to corresponding float number operations. Likewise, we model Java String as Z3's string [33], [34].

### C. Handling Container classes

Container classes, such as *Map* ($key \rightarrow value$) and *Set* ($key \rightarrow key$) are a special case of the languages' built-in classes that establish a mapping from key to value. They have various implementations, such as hash table-based and tree-based implementations. The hash function and traversal of trees may introduce lots of unnecessary path conditions.

**Algorithm 1:** Handling map.

```
1  $PC$ // The global path conditions
2  $arrId$ // The map's unique id corresponding to a Z3 array
3  $keyOf$ // The map's mapping from its value to key
4  get($key, retValue$):
5      if $retValue \neq null$:
6          $PC$.append("$key = keyOf[retValue]$")
7          return true
8      else
9          $PC$.append("read($arrId, key$) = $False$")
10         return false
11
12 put($key, value, retValue$):
13     if get($key, retValue$):
14         $keyOf$.remove($retValue$)
15     else
16         $PC$.append("write($arrId, key, True$)")
17     $keyOf[value] \leftarrow key$
18
19 remove($key, retValue$):
20     if get($key, retValue$):
21         $PC$.append("write($arrId, key, False$)")
22         $keyOf$.remove($retValue$)
```

#### 1) Intuitive approach for containers

Intuitively, we can model the containers as Z3 [27]'s array theory [35]. Z3's array theory exposes the array read and write operations, $v \leftarrow read(A, i)$ and $A' \leftarrow write(A, i, v)$. The term $A$ and $A'$ means the symbolic array, $i$ means the symbolic (or concrete) index, and $v$ means the symbolic (or concrete) value being read or written. Besides, these operations are recorded as path conditions for each symbolic array access, enlarging the number of states for SMT solving. As the value $v$ might be an object rather than a primitive, all the fields of the objects need to be recorded. Web applications' heavy use of complex objects, which contains numerous fields, makes the problem more serious. Our observation of the web applications' use of containers leads us to avoid such complex handling of symbolic objects and containers (object arrays).

#### 2) Handling the containers

For symbolic containers, we observe that there exist one-to-one mappings for their key and value. For example, the ORM caches are *map* from persistent objects' unique keys to the objects themselves. *set*'s key and value are equivalent and thus have one-to-one mapping. Therefore, only the keys need to be recorded for each symbolic array access, as one unique key is binding to its value. The containers are uniformly encoded as Z3's $array{<}KeySort, Bool{>}$, where $KeySort$ can be $Int$, $Float$, or $String$, and $Bool$ represents the existence.

Alg. 1 describes how we generate path conditions for *map*'s methods, where $key$ and $value$ are the parameters and $retValue$ is the return value. If $key$ exists, then $retValue$ must be previously put into the *map*. Due to the one-to-one mapping, we record the certain constraint into path condition (Line 6), where $keyOf$ maintains the mapping from value to its corresponding key. Otherwise, we record the non-existence of $key$ in the $arrId$'s Z3 array (Line 9). Further, *get* is reused by *put* and *remove* to determine the existence of $key$ (Lines 13 and 20). For *put* operation, if $key$ already exists, the old value $retValue$ is removed from $keyOf$ (Line 14). Otherwise, we record that $key$ exists in $arrId$'s Z3 array (Line 16). Finally, $keyOf$ is updated accordingly (Line 17). For *remove* operation, if $key$ exists, the existence of $key$ and $keyOf$'s $retValue$ entry should be cleared (Line 21).

## V. The Deadlock Analyzer

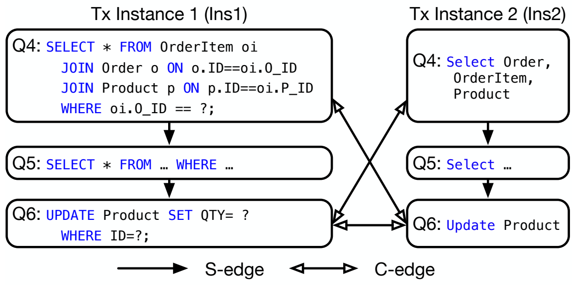
Similar to existing works [12], [13], WeSEER uses a graph-based deadlock detection algorithm. The key difference is that WeSEER identifies potential conflicts in a more fine-grained manner: it gives more precise conflict conditions under which deadlocks occur, reducing the high false positive rates of coarse-grained approaches used by existing works.

### *A. SC-Graph and Deadlock Cycle*

Before describing the fine-grained deadlock detection algorithm, we first introduce the SC-Graph and how a deadlock cycle is found. In WeSEER's SC-Graph, each vertex is one SQL statement of a given transaction instance, assumed to execute atomically. There are two kinds of directed edges. The S(ibling)-edges connect consecutive SQL statements from the same transaction instance in the chronological execution order. The two-way C(onflict)-edges connect SQL statements from different transaction instances if they have potential conflicts (e.g., they access the same table and at least one is write). One directed cycle in the SC-Graph corresponds to one hold-and-wait cycle among involved transactions. The directed cycles indicate potential deadlocks and thus are called deadlock cycles. Fig. 4 shows an example SC-graph of *finishOrder* transaction's (Fig. 1) two instances. Considering the deadlock cycle [ins1.Q4→ins1.Q6→ins2.Q4→ins2.Q6], both instances hold shared locks on *Product* table and try to obtain exclusive locks on the same table. Thus, they may result in a deadlock.

Traditional SC-graph is **coarse-grained** as the C-edges are established if two SQL statements access the same table and at least one is write. Reporting deadlocks based on a coarse-grained SC-graph can have two problems: (1) The coarse-grained SC-graph's inaccuracy. The potential conflicting statements in reported cycle may not access the same database row at runtime. (2) The SC-graph alone is not enough. Some potential deadlocks may be impossible as the SQL parameters might be against the path conditions. Thus, deadlock diagnosis based on coarse-grained SC-graph has the problem of high false-positive rates.

We propose two approaches to solve the above problems separately and consequently reduce the false-positive rate. First, we adopt a fine-grained database lock modeling to establish C-edges in finer granularity. A C-edge is established between two SQL statements if and only if their conflict conditions (generated by the fine-grained database lock modeling) can be satisfied. Second, we encode both the conflict conditions and path conditions into a first-order logic formula and feed it into the SMT solver. Only when the solver returns SAT, do we consider a deadlock to have been found.



```
Q4: SELECT * FROM OrderItem oi
      JOIN Order o ON o.ID==oi.O_ID
      JOIN Product p ON p.ID==oi.P_ID
      WHERE oi.O_ID == ?;
Q5: SELECT * FROM … WHERE …
Q6: UPDATE Product SET QTY= ?
      WHERE ID=?;

Q4: Select Order,
     OrderItem,
     Product
Q5: Select …
Q6: Update Product
```

Fig. 4: The SC-graph of two instances of the *finishOrder* transaction in Fig. 1. The instance 2's statements are simplified.

### *B. Three-Phase Deadlock Diagnosis*

The intuitive approach is to construct a large SC-Graph involving transactions of all collected traces and then encode (1) the cycle detection problem, (2) conflict conditions between every pair of SQL statements, and (3) all path conditions into a first-order logic formula. If the solver reports SAT for the formula, then one potential deadlock is found. However, this brute-force approach significantly increases the time for solving due to the excessive number of conditions considered.

WeSEER proposes a novel three-phase deadlock diagnosis method to efficiently detect deadlock cycles. In the early phases, WeSEER applies coarse-grained but efficient algorithms to filter out most impossible deadlock cases. In the later phases, WeSEER applies more fine-grained but costly detection algorithms to diagnose deadlocks accurately among the remaining cases. Fig. 5 shows the procedure. The deadlock analyzer passes all the traces to the first phase and initiates the diagnosis. Each collected trace has two instances, representing the concurrent execution of the same API.

**The Transaction-Level Phase** adopts the most coarse-grained algorithm to identify transactions that may cause potential deadlocks. It analyzes the tables accessed by the transactions and builds a transaction conflict graph, where vertexes represent transactions. A directed edge is established between two transactions if at least one writes a common table the transactions access, representing potential conflicts between them. Directed cycles in the graph are termed transaction conflict cycles. This phase filters combinations of transactions that cannot form transaction conflict cycles. As no circular waits exist in such combinations, they cannot cause deadlocks.

**The Coarse-Grained Phase** builds a coarse-grained SC-graph for transactions in every input transaction conflict cycle and detects all coarse-grained deadlock cycles (mentioned in Sec. V-A). For each coarse-grained deadlock cycle, WeSEER filters out statements not connected by C-edges (e.g., Q5 in Fig. 5(b)) because they do not contribute to the deadlocks, reducing the number of statements to process in the following phase. Then, we feed the simplified coarse-grained deadlock cycles to the next phase as input.

**The Fine-grained Phase** applies the most accurate algorithm. It leverages fine-grained database lock modeling to generate conflict conditions for all C-edges in the input coarse-grained deadlock cycle. Then, it prepares the path conditions recorded before each transaction's last statement in the cy-

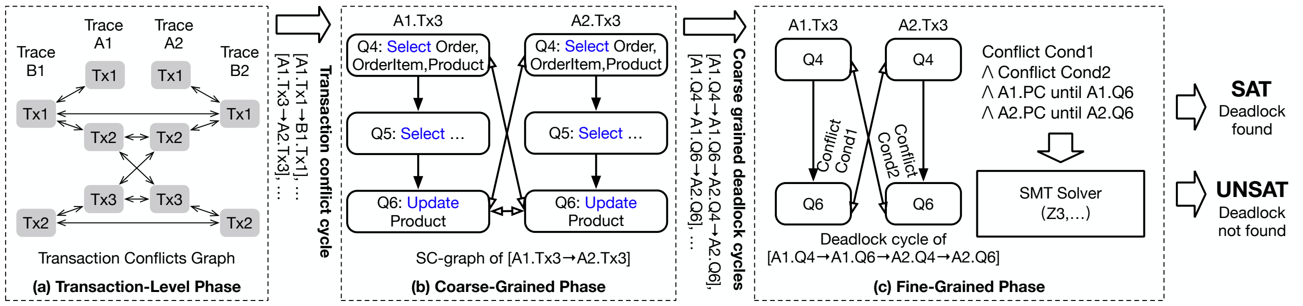


Fig. 5: WeSEER's three-phase deadlock diagnosis. In the transaction-level phases, the transactions are represented as gray vertexes, and each bi-directional arrow corresponds to two directed edges established between two conflicting transactions. The mirror edges are omitted. For example, as there are edges between $B1.Tx1$ and $A1.Tx2$, the edges between $B1.Tx1$ and $A2.Tx2$ are omitted. In the coarse-grained phase, WeSEER builds the SC-graph and finds coarse-grained deadlock cycles for input transaction conflict cycles. In the fine-grained phase, WeSEER encodes the conflict conditions and path conditions and leverages the SMT solver to identify potential deadlocks.

$$
\begin{aligned}
&\textbf{SELECT} \ldots \textbf{ FROM } tab\, alias_1 \,[\,\textbf{JOIN } tab\, alias_n \textbf{ ON} \ldots]^* \\
&\qquad\qquad \textbf{WHERE} \ldots \\
&\textbf{UPDATE } tab \textbf{ SET } col_1 = \ldots \,[, col_n = \ldots]^* \textbf{ WHERE} \ldots \\
&\textbf{INSERT INTO } tab \textbf{ VALUES } (param_1, \ldots, param_n) \\
&\textbf{DELETE FROM } tab \textbf{ WHERE} \ldots
\end{aligned}
$$

Fig. 6: The syntax of statements WeSEER supports.

$$
\begin{aligned}
Qcond &::= Icond \wedge Ncond & ArithExp &::= var\ NumOp\ var \\
, Icond &::= Icond \wedge Exp \mid Exp & &\mid\ var\ NumOp\ number \\
Ncond &::= Disj & StrExp &::= var\ StrOp\ var \\
Disj &::= Disj \vee Conj \mid Conj & &\mid\ var\ StrOp\ string \\
Conj &::= Conj \wedge Term \mid Term & NumOp &::= \neq \mid = \mid < \mid > \mid >= \mid <= \\
Term &::= id \textbf{ is null} \mid Exp & StrOp &::= \neq \mid = \\
Exp &::= ArithExp \mid StrExp
\end{aligned}
$$

Fig. 7: The grammar of query conditions WeSEER supports.

cle. The subsequent path conditions are omitted as they are recorded after the code location where potential deadlocks may occur. Finally, it conjuncts all generated conflict conditions and prepares path conditions into a first-order logic formula. We feed this formula to the SMT solver. A potential deadlock is found if the solver reports SAT.

## C. Fine-Grained Database Lock Modeling

To give precise conflict conditions between two statements with potential conflicts, WeSEER models the database locking procedure to describe the conflicts between SQL statements in a fine-grained manner. As a representative set of database systems (such as MySQL, PostgreSQL, etc.) acquire locks during database index (or index, for short) traversing, we decide to model database locking by simulating the indexing procedure. WeSEER first analyzes given statements to infer all possible database indexes to be used. Then, it models database lock acquiring accordingly. Two SQL statements have potential conflicts if and only if there exists a pair of their locks that have potential conflicts. Finally, based on the above conflicting locks, we generate the conflict condition between two potentially conflicting SQL statements.

### 1) The target SQL statements

Fig. 6 gives the SQL syntax that WeSEER currently supports. $tab$ means the table, $alias$ is the table alias, $col$ is the table column; $param$ represents the SQL parameters. The statements' query conditions are the conjunction of predicates in the Join and Where clauses. Fig. 7 shows the grammar of query conditions ($Qcond$), which is the conjunction of conditions related to indexes ($Icond$) and conditions unrelated to indexes ($Ncond$). We say a condition (or predicate) is related to an index if contains table columns indexed by that index. Variable($var$) represents either SQL parameters or pairs of table aliases and columns.

### 2) Inferring the database indexes to be used

Databases leverage indexes to fetch data and acquire locks accordingly. We use the term $index(table, type, columns)$ to represent a database index, where $table$ is the index's corresponding table, $type$ is the index type ($pri$ for the primary index, $sec$ for the secondary index), $columns$ is the set of data columns indexed by the index. If no indexes are used, the database will scan the whole table. WeSEER needs to infer all possible indexes to analyze how locks are acquired.

When a SQL statement tries to access data, it is possible that not all target tables' indexes are used. The reason is that databases prefer fetching data with indexes to scanning the whole table for efficiency. Taking Q4 (in Fig. 1) as an example. The database tends to first use $index(OrderItem, sec, O\_ID)$, as Q4's Where clauses contain predicates related to this secondary index. Then, it fetches data for *Order* and *Product* tables with their corresponding indexes. Note that $index(OrderItem, sec, P\_ID)$ will not be used. Otherwise, the database has to scan *Product* table first to prepare input corresponding to *OrderItem.P_ID* column.

To find such preferred indexes and predicates, we present a graph-based algorithm. The index usage graph is constructed for each SQL statement. One vertex is created for each unique SQL parameter or table alias in the statement. One directed edge represents the database that can use the SQL parameters or table data of its source vertex to access its target vertex's table leveraging the given index. The edges correspond to one pair of the index and predicates related to the index. They are established only when the statement's query conditions are related to their corresponding indexes. Fig. 8 shows Q4's index usage graph. Since predicates $p2$ and $p3$'s variables correspond to the indexed table column, we establish two directed edges for each of them. As one of $p1$'s variables is

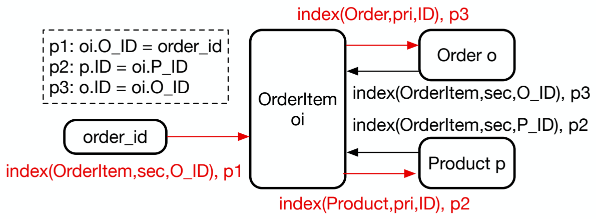


Fig. 8: The index usage graph for Q4 in Fig. 1. $p1$, $p2$, and $p3$ are predicates of Q4's Join and Where clauses. Each edge is tagged with its database index and related predicates. The red edges represent all possible database indexes for Q4.

the SQL parameter, we establish only one directed edge from *order_id* to *oi*. This algorithm can also be applied to UPDATE and DELETE statements as they have Where clauses. INSERT statements' query conditions can be treated as equations on inserted database rows' primary keys.

Through topologically sorting the given statement's index usage graph, the sequences of indexes used for data fetching are generated. Because an index can be used when its input data is available and SQL parameters are always available, the topological sorts should start from vertexes of SQL parameters. Considering the index usage graph in Fig. 8, it has two possible topological sorts starting from *order_id*: *order_id* $\rightarrow$ *oi* $\rightarrow$ *o* $\rightarrow$ *p* and *order_id* $\rightarrow$ *oi* $\rightarrow$ *p* $\rightarrow$ *o*. By enumerating the edges used in the topological sorts, we collect the possible indexes with the predicates related to them, shown in red.

### 3) *Generating the locks to be acquired*

WeSEER generates all possible locks for each given SQL statement. One lock contains (1) index for locking ($index$), (2) lock granularity ($ROW$, $RANGE$, $TABLE$), (3) whether it is shared ($S$) or exclusive ($X$), and (4) predicates ($cond$) for range locks. Alg. 2 shows how to generate the locks. As lock conflicts always happen on the commonly accessed table, Alg. 2 generates locks only for the common (or target) table.

*GenSharedLocks* generates shared locks. Input $isEmpty$ means whether the statement fetches an empty result. The algorithm first infers all pairs of $sql$'s possible indexes and related predicates. For every pair containing indexes of the target table, it generates shared locks accordingly. Row/range locks are acquired for unique or non-unique indexes. Meanwhile, for secondary indexes, an extra row lock is acquired to protect the fetched row on the primary index. When no data rows are fetched, range locks are acquired to protect empty read sets. Note that table-level locks are acquired when no indexes are used. *GenExclusiveLocks* generates exclusive locks for the input SQL statement and the target table. WeSEER assumes the database executes UPDATE/INSERT/DELETE statements by selecting database rows first and then acquiring exclusive locks for target rows. It generates exclusive locks for each updated row on the primary index and the secondary indexes whose related table columns are modified accordingly. After then, WeSEER compares two SQL statements' locks and infer potential conflicts between them. If the two SQL statements have locks on the same database index and at least one of the locks is exclusive, then the statements have potential conflicts.

**Algorithm 2:** Generate locks.

```
1  GenSharedlocks($sql, targetTable, isEmpty$):
2      $indexPredPairs \leftarrow$ InferPossibleIndexes($sql$)
3      $locks \leftarrow$ empty set
4      foreach $index, preds$ pair in $indexPredPairs$:
5          if $index.table \neq targetTable$:
6              continue
7          if not $isEmpty$: // when there exist rows fetched
8              if $index$ is unique:
9                  if $preds$ represents a point query:
10                     $locks$.add($index, ROW, S$)
11                 else:
12                     $locks$.add($index, RANGE, S, preds$)
13             if $index$ is secondary:
14                 $pindex \leftarrow$ primary index of $index.table$
15                 $locks$.add($pindex, ROW, S$)
16         else: // when no rows are fetched
17             $locks$.add($index, RANGE, S, preds$)
18     if $locks$ is empty: // No indexes are used
19         $locks$.add(NULL, $TABLE, S$)
20     return $locks$
21
22 GenExclusivelocks($sql, targetTable$):
23     $locks \leftarrow$ empty set
24     $pindex \leftarrow$ primary index of $targetTable$
25     $locks$.add($pindex, ROW, X$)
26     foreach $index$ written by $sql$:
27         if $index$ is unique:
28             $locks$.add($index, ROW, X$)
29         else:
30             $locks$.add($index, RANGE, X, NULL$)
31     return $locks$
```

### 4) *Generating the conflict conditions*

For potentially conflicting SQL statements, WeSEER generates their conflict conditions. If the conflict conditions are satisfied, then WeSEER confirms the deadlock related to the statements. Alg. 3 describes how to generate conflict conditions. The basic idea is assuming a database result ($r$), such that one database row in $r$ satisfies both $sql_w$'s and $sql_r$'s query conditions. Assuming that $sql_w$ is the UPDATE/INSERT/DELETE statement, while $sql_r$ can be any statement.

Unfortunately, we cannot naively treat the conjunction of $sql_w$'s and $sql_r$'s query conditions as the conflict condition because their variables' naming is not unified. For example, Q4 and Q6 both access *Product* table, and use different names *p.ID* and *ID* (no table alias is used for Q6) for the same table column. Thus, we unify $sql_w$'s and $sql_r$'s query conditions and then conjoin them. The unifying procedure of $sql_r$ (*GenUnifiedCondForRead* in Line 5) associates every identifier in $sql_r$'s query conditions with $r$. For example, in the unified query condition of Q4, *p.ID* is replaced with *r.p.ID*, which is a variable corresponding to $ID$ column of database result $r$'s table alias $p$. The unifying procedure of $sql_w$ (*GenUnifiedCondForWrite* in Line 6) requires more steps. As $sql_r$ may have multiple table aliases of the common table, $sql_w$ needs to generate one unified query condition for every $sql_r$'s table alias. Then, the disjunction of these conditions is returned as $sql_w$'s final unified query condition. Assuming $sql_r$ has two table aliases *p1* and *p2* related to table *Product*, the unified query condition of Q6 is $(r.p1.ID = ...) \vee (r.p2.ID = ...)$.

**Algorithm 3:** Generate conflict conditions.

```
1  GenConflictCond($sql_w, sql_r, isEmpty_r$):
2      $r \leftarrow$ symbol of unified database row
3      $comTable \leftarrow$ the commonly table of $sql_r$ and $sql_w$
4      $aliases \leftarrow$ all $sql_r$'s table aliases related to common table
5      $conflictCond \leftarrow$ GenUnifiedCondForRead($sql_r, r$)
6          $\wedge$ GenUnifiedCondForWrite($sql_w, r, aliases$)
7          $\wedge$ GenAssociatedCond($r$, $sql_r.res$)
8      $locks_w \leftarrow$ GenExclusiveLocks($sql_w, comTable$)
9      $locks_r \leftarrow$ GenSharedLocks($sql_r, comTable, isEmpty_r$)
10     foreach range lock $l_r$ in $locks_r$:
11         if exists $l_w$ in $locks_w$ such that $l_r.index = l_w.index$:
12             $rangeCond \leftarrow$ GenRangeConflictCond($l_r, r$)
13             $conflictCond \leftarrow conflictCond \vee rangeCond$
14     return $conflictCond$
15
16 GenRangeConflictCond($lock_r, r$):
17     transfer $lock_r.cond$ to $rangeCond$ with following steps:
18         "$var = exp$" to "$var \geq exp \wedge var \leq exp$"
19         "$var \neq exp$" to "$var < exp \vee var > exp$"
20         "$var < exp$" to "$var \leq var_g \wedge exp \leq var_g$"
21         "$var \leq exp$" to "$var \leq var_g \wedge exp < var_g$"
22         "$var > exp$" to "$var \geq var_l \wedge exp \geq var_l$"
23         "$var \geq exp$" to "$var \geq var_l \wedge exp > var_l$"
24     $rangeCond \leftarrow$ GenUnifiedCondForRead($rangeCond, r$)
25     return $rangeCond$
```

Further, we need to associate the assumed database result $r$ with $sql_r$'s database result $sql_r.res$ collected at runtime (e.g., Q4's $res4$). *GenAssociatedCond* generates the associated condition between $r$ and $sql_r.res$, representing that there exists one row ($row$) in $sql_r.res$, such that all columns in $r$ have the same value with corresponding columns in $row$. After unifying $sql_r$'s and $sql_w$'s query conditions and the associating condition between $r$ and $sql_r$'s database results, their conjunction is treated as the conflict condition.

The current conflict conditions do not consider range locks [36], such as gap locks and next-key locks commonly used in commercial databases [37], [38]. Range locks' protection ranges depend on the runtime database state. Thus, the SQL statements' query conditions may not precisely indicate the actual protection range of range locks.

We observe that the actual protection range is always the superset of the range lock's condition. Therefore, if $sql_w$ writes a database row within the enlarged range of the given range lock's condition, then one potential conflict exists. *GenRangeConflictCond* in Alg. 3 generated conflict conditions for a shared range lock $lock_r$. Note that $lock_r.cond$ is the conjunction of predicates related to $lock_r.index$, it can be parsed to predicates in the form of $var\ op\ exp$, where $var$ is the variable representing columns related to $lock_r.index$, $op$ can be $<$, $>$, $\leq$ and $\geq$, and $exp$ can be any expression. The equality and inequality operators are also supported by transferring them to equivalent expressions containing $<$, $>$, $\leq$, and $\geq$. Given one expression indicating a range, the algorithm extends its protection ranges by replacing $exp$ with a new variable $var_g$ or $var_l$ (where $var_g > id$ and $var_l < id$). Take the condition $id > 4 \wedge id \leq 8$ as an example, its enlarged expression is $id \geq var_l \wedge 4 \geq var_l \wedge id \leq var_g \wedge 8 < var_l$. The range of $id$ specified by the original condition is $(4, 8]$, while the enlarged range is $[var_l, var_g)$, where $4 \geq var_l \wedge 8 < var_l$.

To generate $sql_r$'s conflict conditions related to range locks, we find all $sql_r$'s possible shared range locks that may conflict with $sql_w$'s exclusive locks (Lines 10-13) and disjoin their conflict conditions (generated by *GenRangeConflictCond* in Line 12) with the origin conflict condition. The disjoined conditions serve as the final conflict conditions returned by *GenConflictCond*. Fig. 9 uses an example illustrating how WeSEER's SMT Solver-based deadlock analyzer works.

## *D. Discussion*

We now discuss WeSEER' limitation in terms of how its design choices can miss deadlocks (false negatives) or report deadlocks that can never be triggered (false positives).

**False negative.** WeSEER uses the test cases written by developers to collect transaction traces. As the test cases do not cover all code paths of the target web application, WeSEER can miss deadlocks hidden in paths unexplored by the test cases, thus resulting in false negatives.

WeSEER's concolic execution engine cannot comprehensively capture transaction traces and their path conditions if they depend on shared variables whose values are affected by thread interleaving. As the concolic execution engine does not explore all thread interleaving, false negatives can occur.

For lock modeling, WeSEER assumes that databases prefer using indexes to full table scans. However, this assumption may fail: a full table scan is preferred when the query is not selective enough, causing WeSEER to miss a deadlock and thus incur false negatives.

**False positive.** Developers may sometimes use ad-hoc transactions that rely on application-level synchronization mechanisms to prevent database deadlocks [5]. As WeSEER does not take into consideration application-level synchronization, it is possible that deadlocks reported by WeSEER will not take place at runtime, thus resulting in false positives.

For lock modeling, WeSEER assumes that databases use all possible join orders if multiple ones exist for one SQL statement. In reality, however, the database can choose the most effective one. Therefore, WeSEER may wrongly assume that some database indexes are used and thus report a deadlock that cannot happen, leading to false positives.

As part of future work, we believe WeSEER can be improved in two ways. First, we can remove the false positives and false negatives resulting from inaccurate lock modeling. In particular, WeSEER can query the database for its concrete execution plan on a given SQL statement. Such functionalities exist in many commercial and open-source databases [39]–[41]. The concrete execution plan contains sufficient information to determine whether and how database indexes are used for a particular SQL statement, thereby avoiding incorrect assumptions on indexes used when modeling locking. Second, we can develop a framework to automatically reproduce the deadlocks according to WeSEER's report. Doing so helps eliminate all false positives and removes the burden on developers to manually verify reported deadlocks.

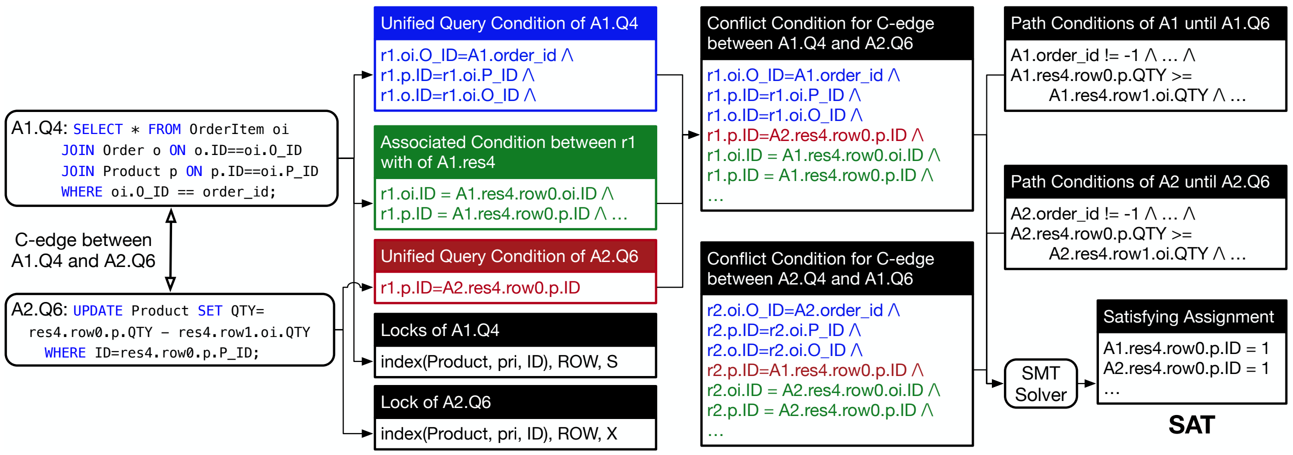

A1.Q4: SELECT * FROM OrderItem oi JOIN Order o ON o.ID==oi.O_ID JOIN Product p ON p.ID==oi.P_ID WHERE oi.O_ID == order_id;
C-edge between A1.Q4 and A2.Q6
A2.Q6: UPDATE Product SET QTY= res4.row0.p.QTY – res4.row1.oi.QTY WHERE ID=res4.row0.p.P_ID;
Unified Query Condition of A1.Q4
r1.oi.O_ID=A1.order_id ∧ r1.p.ID=r1.oi.P_ID ∧ r1.o.ID=r1.oi.O_ID ∧
Associated Condition between r1 with of A1.res4
r1.oi.ID = A1.res4.row0.oi.ID ∧ r1.p.ID = A1.res4.row0.p.ID ∧ …
Unified Query Condition of A2.Q6
r1.p.ID=A2.res4.row0.p.ID
Locks of A1.Q4
index(Product, pri, ID), ROW, S
Lock of A2.Q6
index(Product, pri, ID), ROW, X
Conflict Condition for C-edge between A1.Q4 and A2.Q6
r1.oi.O_ID=A1.order_id ∧ r1.p.ID=r1.oi.P_ID ∧ r1.o.ID=r1.oi.O_ID ∧ r1.p.ID=A2.res4.row0.p.ID ∧ r1.oi.ID = A1.res4.row0.oi.ID ∧ r1.p.ID = A1.res4.row0.p.ID ∧ …
Conflict Condition for C-edge between A2.Q4 and A1.Q6
r2.oi.O_ID=A2.order_id ∧ r2.p.ID=r2.oi.P_ID ∧ r2.o.ID=r2.oi.O_ID ∧ r2.p.ID=A1.res4.row0.p.ID ∧ r2.oi.ID = A2.res4.row0.oi.ID ∧ r2.p.ID = A2.res4.row0.p.ID ∧ …
Path Conditions of A1 until A1.Q6
A1.order_id != -1 ∧ … ∧ A1.res4.row0.p.QTY >= A1.res4.row1.oi.QTY ∧ …
Path Conditions of A2 until A2.Q6
A2.order_id != -1 ∧ … ∧ A2.res4.row0.p.QTY >= A2.res4.row1.oi.QTY ∧ …
SMT Solver
Satisfying Assignment
A1.res4.row0.p.ID = 1 A2.res4.row0.p.ID = 1 …
SAT

Fig. 9: An end-to-end example demonstrating how WeSEER encodes the conflict conditions and path conditions for the deadlock cycle in Fig. 5(c). For the C-edge between trace $A1$'s statement $A1.Q4$ and trace $A2$'s statement $A2.Q6$, we show the unified query conditions for the SELECT statement (in blue) and the UPDATE statement (in red), where $r1$ and $r2$ are the assumed database results for unifying. The associated condition (in green) between $r1$ and $A1.res4$ (database result of $A1.Q4$) is also presented. Symbolic values are associated with their trace ID for distinction (e.g., $order\_id$ in $A1.Q4$ is replaced with $A1.order\_id$). As the locks do not contain potentially conflicting range locks, no conflict conditions for range locks are required. Then, WeSEER conjoins the conflict conditions of both C-edges and the path conditions until the last involved SQL statement ($A1.Q6$ and $A2.Q6$) of both traces. The resulting formula is fed to the SMT solver, and an SAT result is returned, which means the input deadlock cycle is confirmed to be a deadlock.

## VI. Modeling the ORM to Map Statements to Triggering Code

WeSEER reports information to help developers understand how the applications cause deadlocks. We try to establish the mapping from SQL statements to their triggering code. We analyze different ORM operations and handle them accordingly.

First, the ORM operations can be eager or lazy ones: Those eager operations send SQL statements immediately. Therefore, WeSEER directly records the current stack trace for SQL statement submission inside eager operations; Lazy operations buffer the SQL statements and send them when needed, which can be divided into lazy read and lazy write ones. Read operations may return objects which are dynamically generated for lazy read operations. When these objects are first accessed, the ORM sends SQL statements and loads the database results into these objects. As the SQL statements are sent immediately before the objects are used, the code that accesses them can be treated as their triggering code. Therefore, WeSEER records the current stack trace for any SELECT statements, no matter they are eager or lazy.

Lazy write operations can be further classified into explicit and implicit ones. The ORM delays the submission of SQL statements triggered by explicit lazy write, until the transaction commits or the application forces flushing. The remaining problem is how to associate the operations with corresponding SQL statements. With the observation that a well-developed ORM framework should have internal helper functions that translate given persistent objects to corresponding SQL statements. We specially handle these functions to associate the persistent objects with their corresponding SQL statements. Consequently, the mapping from the explicit lazy write operations to corresponding SQL statements can be established.

The remaining implicit lazy write operations are triggered when the application modifies in-memory persistent objects returned by ORM read operations. WeSEER tracks all persistent objects returned by ORM read operations and records the stack trace of the last modification to them. Therefore, the mapping from the invocations of implicit lazy write operations to corresponding SQL statements is also established.

## VII. Evaluation

### A. Implementation

We implemented WeSEER's concolic execution engine or trace collector based on the HotSpot VM of OpenJDK8 [43]. We implement the deadlock analyzer with Java and use Microsoft's Z3 [44] of version *4.8.14* as the SMT solver. Although WeSEER currently targets Java web applications, we believe WeSEER's design is general because it does not exploit the properties of any specific programming languages, ORM frameworks, web applications, or database drivers. WeSEER can diagnose deadlocks of web applications written by programming languages other than Java, as long as we implement WeSEER with target languages' concolic execution engines.

### B. Experimental setup

**Hardware Configuration.** Three machines are used, each of which is equipped with two 10-core Intel Xeon E5-2650 processors, 64 GB of RAM, and an Intel X520 10GbE NIC. They are dedicated as the web, database, and client server.

**Web Applications and Database.** We diagnose two popular Java E-commerce web applications in the GitHub. One application is *Broadleaf* [17] (version *broadleaf-6.0.9-GA*), and we choose its official Java demo website [45] (version *broadleaf-6.0.9-GA*) as the front-end. The other application is *Shopizer* [18] (version *2.12.0*), which contains the front-end in the same code repository. *Broadleaf* has 1.5K GitHub stars

TABLE I: The target APIs' information. As *Broadleaf* and *Shopizer* provide different forms of input, only common and representative inputs are shown. Shopizer does not contain *Payment* API.

| API Name | API Description | Input description | Times in Broadleaf | Times in Shopizer |
|---|---|---|---|---|
| Register | Register one user | username, email, password, password for confirmation | 1 | 1 |
| Add | Add one product to cart | userId, productId | 3 | 3 |
| Ship | Edit user's shipment information | userId, shippment address, phone number, ... | 1 | 1 |
| Payment | Edit user's payment information | userId, payment address, phone number, ... | 1 | - |
| Checkout | Checkout the order | userId | 1 | 1 |

TABLE II: Deadlocks found by WeSEER [42]. Description and fixing approaches of involved API and transactions are provided. Deadlocks (d3, d4), (d5, d6), (d7∼9), (d12, d13), and (d14∼16) can be prevented by the same fixing approach, respectively.

| App. | Id. | Deadlock APIs | Descriptions of deadlock-prone Txn | App-level fixing approaches |
|---|---|---|---|---|
| Broadleaf | d1 | Register — Register | Create a new user | f1: Use correct ORM operation |
| | d2 | Add2 — Add2 | App-level locks protecting cart | f2: Use MySQL UPSERT mechanism |
| | d3, d4 | Add2,Add3 — Add2,Add3 | Create a new order item | f3: Separate SELECT from original transaction |
| | d5, d6 | Add2,Add3 — Add2,Add3 | Create order and fulfillment items | f4: Move forward ORM flush |
| | d7, d8 | Add2,Add3 — Add2,Add3 | Calculate shopping cart's price | f5: Separate SELECT from original transaction |
| | d9 | Add2,Add3 — Ship | | |
| | d10 | Ship — Ship | Create address information | f6: Reorder SQL statements |
| | d11 | Ship — Ship | Calculate shopping cart's price | f7: Separate SELECT from original transaction |
| | d12, d13 | Ship — Ship | Calculate shopping cart's price | f8: Separate SELECT from original transaction |
| Shopizer | d14 | Ship,Checkout — Ship,Checkout | Price the order's products | f9: Force serial execution with app-level locks |
| | d15 | Ship,Checkout — Checkout | Price/Commit the order's products | |
| | d16 | Checkout — Checkout | Commit the order's products | |
| | d17 | Checkout — Add2,Add3,Ship,Checkout | Commit/Price the order's products | f10: Ensure the same locking order |
| | d18 | Checkout — Add2,Add3,Ship,Checkout | Commit/Read the cart's products | f11: Ensure the same locking order |

and 190K LOCs, while *Shopizer* has 2.7K starts and 92K LOCs. Both applications' ORM framework is Hibernate [7] (version *5.2.17*). The database is MySQL *5.7.25*.

The deadlock diagnosis is not completely free because we are not the target web applications' developers. It still takes lots of effort to analyze both web applications, considering their large codebases. We have to 1) understand the application logic, 2) reproduce reported deadlocks, 3) propose fixing approaches, and 4) write performance evaluations. We believe this effort is acceptable for web application developers.

**Workload.** Our target workload uses multiple clients **sequentially** issuing HTTP requests to invoke APIs shown in Table I, simulating one customer who uses browsers to access the web applications. Note that *Add* is invoked three times, and each invocation runs different code paths due to different database states. We use *Add1*, *Add2*, and *Add3* to distinguish them. We sequentially run one unit test for each API in Table I for trace collection. The former unit test's database state after execution is used as the next one's initial database state. We also use the coarse-grained approach of STEPDAD [12] and REDACT [13] for deadlock diagnosis. However, the approach is impractical, as it outputs 18,384 hold-and-wait cycles among the transactions provided by WeSEER's trace collectors.

### *C. Diagnosing and Fixing Deadlocks*

After checking the deadlocks reported by WeSEER, we manually confirm 18 deadlocks (shown in Table II) and propose the fixing approaches.

#### *1) Deadlocks in Broadleaf*

All *Broadleaf*'s deadlocks are due to the pattern that range locks of some transactions' SELECT statements block the other's INSERT. For d1, the deadlock-prone transaction uses the wrong ORM operation *merge* to insert new database rows. The *merge* operation issues one SELECT statement followed by one INSERT statement and thus may cause deadlock. We fixed this deadlock by replacing *merge* with *persist*, which issues one INSERT statement only. As the application doesn't use eliminated SELECT statements, this solution doesn't change application semantics.

For d2, the deadlock-prone transaction first checks the existence of one target database row. If the row exists, the transaction updates it. Otherwise, it inserts a new row. When the target row does not exist, a range database lock is acquired, and the deadlock is possible to occur. We leverage MySQL's UPSERT mechanism [46] to replace the transaction logic with a semantically equivalent SQL, and thus avoid the deadlock.

For d3∼d4, d7∼d9, and d11∼d13, all the deadlock-prone transactions of issue SELECT statements first. These statements may return empty results and acquire range locks, which unnecessarily block the other transactions' insertion. To avoid such deadlocks, we modify the application so that these statements are issued in a separate transaction from the deadlock-prone transaction. As *Broadleaf* uses application-level locks to protect consistency, this approach has no correctness issues.

For d5 and d6, the deadlock cycles are formed due to the statements reordering caused by ORM write-behind cache. We move forward an ORM flush and thus avoid the deadlocks. As this approach does not modify the statements, it is correct.

For d10, the deadlock-prone transaction scans target address information and then inserts new information into the same table. Range locks acquired by the scan can cause deadlocks. We rewrite the transaction to a semantically equivalent one, which inserts first and then scans, avoiding the deadlock.

#### *2) Deadlocks in Shopizer*

All the deadlocks in *Shopizer* are caused by accesses to the *Product* table. For d14∼d16, all deadlock-prone transac-

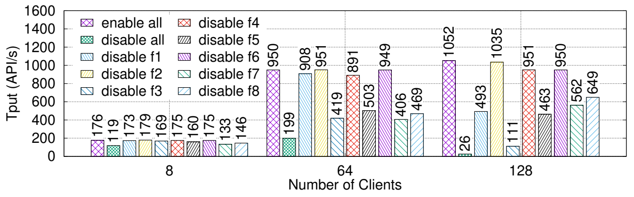


Fig. 10: Performance impact of Broadleaf's deadlocks.

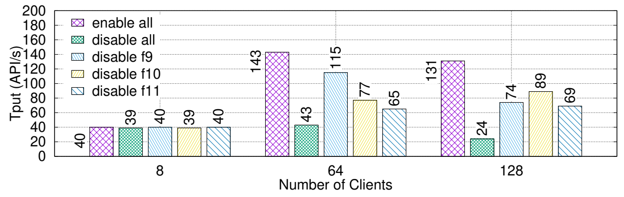


Fig. 11: Performance impact of Shopizer's deadlocks.

TABLE III: Time (milliseconds) spent by different JDK versions for executing unit tests of *Broadleaf*.

| JDK Version | Register | Add1 | Add2 | Add3 | Ship | Payment | Checkout |
|---|---|---|---|---|---|---|---|
| Original | 9 | 822 | 117 | 107 | 211 | 114 | 103 |
| Interpretive | 42 | 6953 | 1062 | 1070 | 3062 | 1328 | 1391 |
| Interpretive+Concolic | 147 | 43409 | 5079 | 4842 | 17231 | 7196 | 6336 |

tions read-modify-write common database rows and can cause deadlocks. We use application-level locks to enforce the serial execution of these transactions, preventing deadlocks. For d17 and d18, *Checkout*'s deadlock-prone transactions update multiple database rows in *Product* table, which are concurrently accessed by other transactions. If they access common database rows in different orders, deadlocks occur. We fix the deadlocks by enforcing the transactions to access the common rows in the same order. The above two approaches do not modify SQL statements and thus are correct.

### D. The Performance Evaluation

Figures 10 and 11 shows the performance after we manually fix the deadlocks. The "enable all"/"disable all" legends mean that all fixing approaches are enabled/disabled. For each application, we disable one fixing approach each time and enable the rest to show the performance impacts. Comparing "enable all" and "disable all", 39.5× and 4.5× performance improvement are achieved for *Broadleaf* and *Shopizer*, respectively. After deadlocks are fixed, the database server's CPU resources are saved due to fewer transaction aborts. For example, the number of transaction aborts per second reduces from 904 ("disable all") to 0 ("enable all"), for the 128-client *Broadleaf* experiment. The result shows that fixing the deadlocks can achieve performance similar to or better than that of leaving them handled by databases. Further, no deadlocks are triggered for "enable all", showing the effectiveness of WeSEER's deadlock diagnosis and our deadlock fixing approach.

### E. The Concolic Excecution Overhead

This section presents the overhead for concolic executing *Broadleaf*'s unit tests. For development ease, the concolic execution engine is based on the interpretive execution version of OpenJDK8's HotSpot VM, which disables performance optimization such as the JIT compiler. We evaluate the execution time of the original JDK version (**Original**), the interpretive execution JDK version (**Interpretive**), and the concolic execution JDK version (**Interpretive+Concolic**). Table III shows the result. As WeSEER requires executing each unit test only once, we consider the second-level overhead acceptable.

## VIII. Related Work

**Language-level deadlock**. There are numerous works targeting solving language-level deadlocks. They cannot be directly applied to handle deadlocks of database lock due to the mismatched abstraction (lock/unlock V.S. SQL). Model Checking [47] can enumerate all possible code paths but is too costly for large-scale applications. Static analysis [48] cannot be applied to ORM-based web applications, as SQLs are generated dynamically. WeSEER uses the dynamical deadlock cycle detection similar to [49]–[51]. Deadlock Immunity [52], [53] uses runtime-collected information to prevent deadlocks.

**Deadlock of database lock**. REDACT [13] is a deadlock prevention tool that provides deadlock immunity [52] for database deadlocks. It records the potential deadlock cycles and prevents them from occurring at runtime. STEPDAD [12] is a test case generation tool. It intercepts the database drivers to increase the probability of concurrent execution of statements belonging to the same deadlock cycle. However, they cannot be applied to the problem studied by WeSEER due to the challenges described in Sec. II-C. Tang et al. [5] study ad hoc transactions written by developers on the application side. They summarize the data access patterns causing deadlocks and how application-level locks help prevent deadlocks. Qiu et al. [6] do a survey on deadlocks of database-backed applications. They study how deadlocks are caused inside the database and summarize the common deadlock patterns.

**SC-graph**. SC-graph originally aims for transaction chopping [54] . Many existing works [14]–[16], [55]–[57] use SC-graph to design high-performance concurrency control, while WeSEER uses SC-graph for detecting deadlock cycles and applies conflict conditions to the SC-graph's C-edges.

## IX. Conclusion

We present WeSEER, the first tool which uses information collected by concolic execution and fine-grained database lock modeling for deadlock diagnosis. Thus, WeSEER supports large-scale ORM-based web applications. WeSEER diagnose 18 real-world deadlocks in large-scale Java web applications, Broadleaf and Shopizer. Compared to MySQL, manually fixing these deadlocks achieves up to 39.5× and 4.5× performance improvement for Broadleaf and Shopizer, respectively.

## Acknowledgment

We thank the anonymous reviewers for their insightful feedback which pointed out several limitations of WeSEER's lock modeling and suggested querying the database for concrete execution plans as future work (Sec. V-D). We thank Zhanghan Wang for his contributions to earlier work on WeSEER. This work is supported by the National Natural Science Foundation of China under Grant Nos. 62272304, 62132014, and 61925206. Jinyang Li is partly supported by NSF Award 2220407. The corresponding author is Zhaoguo Wang (zhaoguowang@sjtu.edu.cn).

## References

[1] E. G. C. Jr., M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971. [Online]. Available: https://doi.org/10.1145/356586.356588

[2] Oracle, "Deadlock detection - mysql 5.7 reference manual," https://dev.mysql.com/doc/refman/5.7/en/innodb-deadlock-detection.html, 2022.

[3] T. P. G. D. Group, "Explicit locking - postgresql documentation," https://www.postgresql.org/docs/current/explicit-locking.html, 2023.

[4] Microsoft, "Deadlocks - transaction locking and row versioning guide," https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver16\#deadlocks, 2022.

[5] C. Tang, Z. Wang, X. Zhang, Q. Yu, B. Zang, H. Guan, and H. Chen, "Ad hoc transactions in web applications: The good, the bad, and the ugly," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 4–18. [Online]. Available: https://doi.org/10.1145/3514221.3526120

[6] Z. Qiu, S. Shao, Q. Zhao, and G. Jin, "A characteristic study of deadlocks in database-backed web applications," in *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, Z. Jin, X. Li, J. Xiang, L. Mariani, T. Liu, X. Yu, and N. Ivaki, Eds. IEEE, 2021, pp. 510–521. [Online]. Available: https://doi.org/10.1109/ISSRE52982.2021.00059

[7] Hibernate, "Hibernate orm," https://github.com/hibernate/hibernate-orm, 2022.

[8] SQLAlchemy, "The python sql toolkit and object relational mapper," https://www.sqlalchemy.org/, 2022.

[9] D. Project, "Object relational mapper," https://www.doctrine-project.org/projects/orm.html, 2022.

[10] NHibernate, "The object-relational mapper for .net," https://nhibernate.info/, 2022.

[11] Hibernate, "Hibernate orm 5.2.18.final user guide," https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html, 2022.

[12] M. Grechanik, B. M. M. Hossain, and U. A. Buy, "Testing database-centric applications for causes of database deadlocks," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 2013, pp. 174–183. [Online]. Available: https://doi.org/10.1109/ICST.2013.19

[13] M. Grechanik, B. M. M. Hossain, U. A. Buy, and H. Wang, "Preventing database deadlocks in applications," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 356–366. [Online]. Available: https://doi.org/10.1145/2491411.2491412

[14] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li, "Scaling multicore databases via constrained parallel execution," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1643–1658. [Online]. Available: https://doi.org/10.1145/2882903.2882934

[15] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, "Transaction chains: achieving serializability with low latency in geo-distributed storage systems," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 276–291. [Online]. Available: https://doi.org/10.1145/2517349.2522729

[16] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, J. Flinn and H. Levy, Eds. USENIX Association, 2014, pp. 479–494. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/mu

[17] BroadleafCommerce, "Broadleafcommerce," https://github.com/BroadleafCommerce/BroadleafCommerce, 2022.

[18] shopizer ecommerce, "shopizer," https://github.com/shopizer-ecommerce/shopizer, 2022.

[19] Z. Wang, Z. Zhou, Y. Yang, H. Ding, G. Hu, D. Ding, C. Tang, H. Chen, and J. Li, "Wetune: Automatic discovery and verification of query rewrite rules," in *Proceedings of the 2022 International Conference on Management of Data, SIGMOD Conference 2022, Philadelphia, PA, USA, June 12 - June 17, 2022*. ACM, 2022. [Online]. Available: https://doi.org/10.1145/3514221.3526125

[20] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT - a formal system for testing and debugging programs by symbolic execution," in *Proceedings of the International Conference on Reliable Software 1975, Los Angeles, California, USA, April 21-23, 1975*, M. L. Shooman and R. T. Yeh, Eds. ACM, 1975, pp. 234–245. [Online]. Available: https://doi.org/10.1145/800027.808445

[21] Y. Vizel, G. Weissenbacher, and S. Malik, "Boolean satisfiability solvers and their applications in model checking," *Proc. IEEE*, vol. 103, no. 11, pp. 2021–2035, 2015. [Online]. Available: https://doi.org/10.1109/JPROC.2015.2455034

[22] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[23] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2007.41

[24] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008. [Online]. Available: https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[25] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018. [Online]. Available: https://doi.org/10.1145/3182657

[26] T. Balyo, M. J. H. Heule, and M. Järvisalo, "SAT competition 2016: Recent developments," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, S. Singh and S. Markovitch, Eds. AAAI Press, 2017, pp. 5061–5063. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14977

[27] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[28] E. Goldberg and Y. Novikov, "Berkmin: A fast and robust sat-solver," *Discret. Appl. Math.*, vol. 155, no. 12, pp. 1549–1561, 2007. [Online]. Available: https://doi.org/10.1016/j.dam.2006.10.007

[29] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Exponential recency weighted average branching heuristic for SAT solvers," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, D. Schuurmans and M. P. Wellman, Eds. AAAI Press, 2016, pp. 3434–3440. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12451

[30] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001, pp. 530–535. [Online]. Available: https://doi.org/10.1145/378239.379017

[31] Oracle, "Java jdbc api," https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/, 2022.

[32] Microsoft, "Microsoft open database connectivity (odbc)," https://learn.microsoft.com/en-us/sql/odbc/microsoft-open-database-connectivity-odbc?view=sql-server-ver16, 2022.

[33] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: a z3-based string solver for web application analysis," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and

M. Mezini, Eds. ACM, 2013, pp. 114–124. [Online]. Available: https://doi.org/10.1145/2491411.2491456

[34] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang, "Z3str2: an efficient solver for strings, regular expressions, and length constraints," *Formal Methods Syst. Des.*, vol. 50, no. 2-3, pp. 249–288, 2017. [Online]. Available: https://doi.org/10.1007/s10703-016-0263-6

[35] L. M. de Moura and N. Bjørner, "Generalized, efficient array decision procedures," in *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 2009, pp. 45–52. [Online]. Available: https://doi.org/10.1109/FMCAD.2009.5351142

[36] D. B. Lomet, "Key range locking strategies for improved concurrency," in *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, R. Agrawal, S. Baker, and D. A. Bell, Eds. Morgan Kaufmann, 1993, pp. 655–664. [Online]. Available: http://www.vldb.org/conf/1993/P655.PDF

[37] Oracle, "Innodb locking - mysql 5.7 reference manual," https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html, 2022.

[38] H. Kimura, G. Graefe, and H. A. Kuno, "Efficient locking techniques for databases on modern hardware," in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012*, R. Bordawekar and C. A. Lang, Eds., 2012, pp. 1–12. [Online]. Available: http://www.adms-conf.org/kimura_adms12.pdf

[39] Orcale, "Explain output format - mysql 5.7 reference manual," https://dev.mysql.com/doc/refman/5.7/en/explain-output.html, 2023.

[40] T. P. G. D. Group, "Explain - postgresql documentation," https://www.postgresql.org/docs/current/sql-explain.html, 2023.

[41] Microsoft, "Display and save execution plans," https://learn.microsoft.com/en-us/sql/relational-databases/performance/display-and-save-execution-plans?view=sql-server-ver16, 2023.

[42] Z. Dong, "Database deadlocks found by weseer," https://github.com/windybeing/WeSEER-DatabaseDeadlocks, 2023.

[43] OpenJDK, "jdk8u," https://hg.openjdk.java.net/jdk8u/jdk8u/, 2022.

[44] Z3Prover, "Z3," https://github.com/Z3Prover/z3, 2022.

[45] BroadleafCommerce, "Demosite," https://github.com/BroadleafCommerce/DemoSite, 2022.

[46] Orcale, "Insert ... on duplicate key update statement," https://dev.mysql.com/doc/refman/5.7/en/insert-on-duplicate.html, 2022.

[47] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003. [Online]. Available: https://doi.org/10.1023/A:1022920129859

[48] D. R. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, M. L. Scott and L. L. Peterson, Eds. ACM, 2003, pp. 237–252. [Online]. Available: https://doi.org/10.1145/945445.945468

[49] P. Joshi, C. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, M. Hind and A. Diwan, Eds. ACM, 2009, pp. 110–120. [Online]. Available: https://doi.org/10.1145/1542476.1542489

[50] Y. Cai and W. K. Chan, "Magicfuzzer: Scalable deadlock detection for large-scale applications," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds. IEEE Computer Society, 2012, pp. 606–616. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227156

[51] M. Eslamimehr and J. Palsberg, "Sherlock: scalable deadlock detection for concurrent programs," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 353–365. [Online]. Available: https://doi.org/10.1145/2635868.2635918

[52] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 295–308. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/jula/jula.pdf

[53] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu, "UNDEAD: detecting and preventing deadlocks in production software," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 729–740. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115684

[54] D. E. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction chopping: Algorithms and performance studies," *ACM Trans. Database Syst.*, vol. 20, no. 3, pp. 325–363, 1995. [Online]. Available: https://doi.org/10.1145/211414.211427

[55] H. Chen, R. Chen, X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan, "Fast in-memory transaction processing using RDMA and HTM," *ACM Trans. Comput. Syst.*, vol. 35, no. 1, pp. 3:1–3:37, 2017. [Online]. Available: https://doi.org/10.1145/3092701

[56] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds. ACM, 2015, pp. 87–104. [Online]. Available: https://doi.org/10.1145/2815400.2815419

[57] J. Wang, D. Ding, H. Wang, C. Christensen, Z. Wang, H. Chen, and J. Li, "Polyjuice: High-performance transactions via learned concurrency control," in *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, A. D. Brown and J. R. Lorch, Eds. USENIX Association, 2021, pp. 198–216. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/wang-jiachen