A Characteristic Study of Deadlocks in Database-Backed Web Applications

Zhengyi Qiu, Shudi Shao, Qi Zhao, Guoliang Jin
North Carolina State University
{zqiu2, sshao, qzhao6, guoliang_jin}@ncsu.edu

***Abstract*—Deadlocks in database-backed web applications could involve different numbers of HTTP requests, and they could be caused by locks explicitly requested in application code or implicitly requested by databases during query execution. To help developers understand these deadlocks and guide the design of tools for combating these deadlocks, we conduct a characteristic study with 49 deadlocks collected from real-world web applications developed following different programming paradigms. We provide categorization results based on HTTP request numbers and resource types, with a special focus on categorizing deadlocks on database locks. We expect our results to be useful for application developers to understand web-application deadlocks and for tool researchers to design comprehensive support for combating web-application deadlocks.**

I. Introduction

Web applications are now an important platform for companies to deliver content and services to customers. By the nature of web applications, they are concurrent and thus subject to deadlocks. With the development of cloud platforms for hosting web applications, they become more and more popular. Coupled with the wide availability of hand-held devices, deadlocks become a more critical problem as deadlocks could happen more often with an increasing user base.

In web applications, the core business logic is a group of *request handlers*, which are responsible for handling incoming HTTP requests. Depending on the number of requests involved, deadlocks can be categorized as *inter-request deadlocks* where the deadlocks happen between request handlers for two or more requests, *intra-request deadlocks* where the deadlocks happen within a request handler while handling one request, and *non-request deadlocks* where the deadlocks happen without involving request handling but in other execution phases of the web applications, e.g., when the applications start, shutdown, restart, or perform background tasks.

Web-application deadlocks could involve different types of resources. As web applications are commonly backed by databases on the server-side, database locks could be one important type of resources involved in web-application deadlocks. Language-level synchronization objects can also be involved, depending on the support for concurrency and synchronization provided by different web-application development languages. For example, Java has more mature support for multithreading compared with PHP and Python. Lastly, as different paradigms and frameworks for developing web applications, e.g., Object Relational Mapping (ORM) and event-driven Node.js, are being proposed and adopted, synchronization objects in these frameworks and libraries can also be involved. Among these lock types, database locks are unique in that SQL queries could lead to *implicit* lock acquisition due to database internals.

This work was supported in part by US NSF grant no. CCF-2008056. The first two authors contributed equally to this work and are listed alphabetically.

Most existing work on deadlocks focus on multi-threaded programs, including characteristic studies [41], [51] and various techniques for detection [14], [15], [24], [25], [31], [32], [36], [39], [40], [45], [46], [49], [56], [60], [69], avoidance [36], [65], [66], prevention [47], [69], testing [59], and fixing [30], [50]. Since these general techniques focus on modeling language-level locks, they will not be able to handle deadlocks on database locks that are not explicitly requested in application code. For web-application deadlocks not related to concurrent request handlers or database locks, it is also not clear how helpful existing techniques are.

Specific to web-application deadlocks, existing techniques all focus on database-lock deadlocks, and detect-and-recover is the most well-known approach. Specifically, major databases, e.g., MySQL, PostgreSQL, and SQL Server, provide deadlock detection capability [10], [13], [19]. Upon a detected deadlock, a victim will be chosen, and the web application could retry the victim transaction. Databases also provide error logs with which application developers can diagnose the deadlocks and fix the root cause of these deadlocks if they choose to.

However, deadlocks on database locks are difficult to understand even with database logs. For example, someone posted the following question on StackOverflow upon seeing error logs about a deadlock from MySQL/InnoDB [23].

> "Why MySQL starts deadlocking when this simple command of scheduling a job is executed concurrently? If it is really true that InnoDB is expected to create deadlocks even in normal circumstances, then how is MySQL expected to be used in any production database which is expected to have more concurrent users? Am I missing something?"

Since the aforementioned StackOverflow question has no accepted answer yet, we use a deadlock example from the MySQL manual [1] shown in Listing 1 to illustrate the challenges of deadlock understanding. In Listing 1, three transactions try to insert the same value on the primary key in sequence, and then the first transaction rolls back, after which, the second and third transactions will be in a deadlock.

To fully understand how this sequence of queries leads to the deadlock, one needs to know the locking strategy followed by the underlying database storage engine and different locks

```
CREATE TABLE t1 (i INT, PRIMARY KEY (i)) ENGINE = InnoDB;

START TRANSACTION; /* TX1 */
INSERT INTO t1 VALUES(1);
                START TRANSACTION; /* TX2 */
                INSERT INTO t1 VALUES(1);
                                START TRANSACTION; /* TX3 */
                                INSERT INTO t1 VALUES(1);
ROLLBACK;
```

Listing 1. An example from MySQL's official manual

requested by different queries being executed. Note that sometimes multiple locks could be requested during different phases of executing one query. While some manuals for the locking strategy used by database storage engines are usually provided by vendors, they do not seem to be enough to help application developers quickly understand web-application deadlocks on database locks, as exemplified by the aforementioned StackOverflow question. Facing these challenges, application developers could benefit from a characteristic study of real-world deadlocks on database locks, with which they can learn common patterns and acquire the necessary knowledge on database locking useful for deadlock understanding.

Beyond the detect-and-recover approach with support primarily from the database community, the software-engineering community has also contributed to the testing of deadlocks in database-backed applications [44] and prevention of deadlocks on database locks [43]. However, existing techniques only model the locking behavior in database queries very conservatively, and the example in Listing 1 is beyond the capability of these techniques. It is unclear how well existing techniques can cover real-world deadlocks on database locks.

To complement the current state of the art, in this work, we conduct a characteristic study of real-world deadlocks from database-backed web applications. We start our study with the following research question:

- **RQ1:** What are the common types of deadlocks in web applications regarding the number of HTTP requests and deadlock resources, and how these characteristics are impacted by application differences?

To answer **RQ1**, we do keyword search in the bug-tracking systems of 106 database-backed web applications, covering applications developed with major paradigms and languages, and find 49 real-world deadlocks in web applications. We characterize these 49 deadlocks based on the number of HTTP requests and deadlock resources involved in them. Our results suggest that inter-request deadlocks on database locks are not only the most common but also the most challenging type of deadlocks in web applications, which is worth further investigation. As our keyword search only returns deadlocks in a subset of web applications, we also study the relationship between application characteristics and the number of deadlocks, and our results suggest that both development paradigm and project history could affect the number of deadlocks.

We proceed with the following two research questions to further study web-application deadlocks on database locks:

- **RQ2:** What are the common types of web-application deadlocks on database locks?
- **RQ3:** What are the common fixing strategies of web-application deadlocks on database locks?

To answer **RQ2** and **RQ3**, we use the 36 deadlocks on database locks that we collect while answering **RQ1**, and we further complement the bug set with 27 deadlocks based on StackOverflow questions. We characterize these 63 deadlocks into four different hold-and-wait cycles, depending on the complexity of resources involved. To make our study results useful for developers to understand database-lock deadlocks they may encounter, we further divide three out of the four types of cycles into 12 patterns and provide an example for each pattern. For each example, we describe the queries and the locks requested by these queries in detail. Among all the different categories of database-lock deadlocks, existing work [43], [44] may only be able to handle one pattern that is the most straightforward. Compared with the patterns, we find fixing strategies more straightforward to understand, and we also summarize our findings.

Overall, we expect our results can (1) ease the task of deadlock understanding for application developers and (2) guide tool researchers and developers to design and implement comprehensive tool support for deadlocks in web applications.

II. Methodology

In this section, we first describe our methodology on how we collect and analyze bug reports related to deadlocks from real-world web applications developed using different programming paradigms and frameworks, and we then describe our methodology on how we collect and analyze StackOverflow posts related to deadlocks on database locks. To answer **RQ1**, we use deadlock reports from real-world web applications. To answer **RQ2** and **RQ3**, we use real-world web-application deadlocks on database locks labeled after answering **RQ1** together with StackOverflow questions.

Our study includes three types of web applications, (1) classical ones that access databases by constructing SQL queries directly, (2) those implemented on top of ORM frameworks, and (3) those implemented on top of the Node.js framework. For classical web applications, we start with the application list from the study of performance antipatterns in classical web applications [62]. For ORM web applications, we start with the application list from the study of concurrency control in ORM web applications [29]. For Node.js applications, we start with the application list from the concurrency-bug study for Node.js applications [64] but exclude those that are just libraries but not complete applications. We also include other open-source web applications that we are aware of and those we run into during our study, e.g., some StackOverflow questions mention the names of web applications we originally do not include. To this end, our application set includes 11 classical, 77 ORM, and 18 Node.js web applications.

We follow the methodology taken by existing studies on concurrency bugs in multi-threaded applications [51], performance bugs in web applications [62], [67], [68], and non-deadlock concurrency bugs in web applications [37], [57], [64] while collecting and studying bug reports related to deadlocks.

Table I. Web applications and numbers of bugs being studied and their overall characteristics

			Non-Request		Intra-Request		Inter-Request		
Programming Paradigm	**Application (Abbreviation)**	**Server-Side Development Language**	**Thread Sync. (Lock Only)**	**Database Lock**	**Thread Lock**	**Database Lock**	**Thread Lock**	**Database Lock**	**Cache Lock**
Classical	MediaWiki (MW)	PHP	-	-	-	2	-	16	1
	Odoo (OD)	Python	-	-	-	4	-	-	-
	Drupal (DPL)	PHP	-	-	-	-	-	3	-
	Sonar (SNR)	Java	1 (1)	-	-	-	-	2	-
	BugZilla (BZ)	Perl	-	-	-	-	-	1	-
	OpenMRS (MRS)	Java	3 (2)	-	-	-	-	1	-
	Gerrit (GRT)	Java	3 (2)	-	-	-	2	-	-
ORM	Gitlab (GL)	Ruby on Rails	-	-	-	-	-	1	-
	Discourse (DC)	Ruby on Rails	-	-	-	-	-	1	-
	Spree (SPR)	Ruby on Rails	-	-	-	-	-	1	-
	Openstreetmap (OSM)	Ruby on Rails	-	-	-	-	-	1	-
	Lobsters (LOB)	Ruby on Rails	-	-	-	-	-	1	-
	AWX (AWX)	Django / Python	-	-	1	-	-	-	-
	Sentry (SEN)	Django / Python	-	-	2	-	-	-	-
Node.js	Ghost (GHO)	Javascript	-	1	-	-	-	1	-

To collect bugs related to deadlocks, we first search for closed bug reports in each application's issue-tracking system with the keyword "deadlock(s)." We do not include other keywords in our search because we would like to study bugs that are determined by application developers as deadlocks, under which case we believe the well-known word "deadlock(s)" will appear in the bug report. After keyword search, we obtain a total of 546 bug reports, i.e., 384 reports from 10 classical web applications, 148 reports from 22 ORM web applications, and 14 reports from 7 Node.js web applications.

With this initial set of bug reports, we filter out ones that only mention the word "deadlock(s)" but are actually not deadlocks. For example, sometimes application developers may call a hang bug due to infinite loops as deadlock. We also filter out bug reports that do not contain sufficient information for us to understand. A bug report typically contains some bug description, followed by some discussion and comments on possible causes and fixes, some intermediate fixes, and the final committed fix. Every bug report is manually inspected and discussed by at least two authors to ensure the objectivity of our conclusions. We determine the root cause of each bug by examining each bug report to understand what particular reasons in program code, schemas, or database behaviors cause the deadlock bugs, and we determine the fix strategy of each bug by inspecting its accepted patch for changes in program code, queries, or schema and reviewing the patch submitter's description of the fix.

Following this process, our final set has 49 closed reports with sufficient information for us to determine that their root causes are deadlocks. In comparison, the study of concurrency bugs in multi-threaded applications includes 31 deadlocks [51]. Table I shows the names and the numbers of deadlocks for each application. Note that the previous concurrency-bug study on Node.js applications and libraries states that they found no deadlock [64]. For the two Node.js deadlocks we find, one of them is reported after the study is published. The other is reported before the study is published, but the deadlock happens during the application start phase after a database upgrade, which could be the reason why it was not included by the authors of the previous study [64].

To further complement our understanding of deadlock patterns, we also search questions on StackOverflow for analysis.

Table II. Accumulated numbers of deadlocks involving different numbers of requests and different types of resources

	Thread Sync (Lock Only)	Database Lock	Cache Lock	Total
Non-Request	7 (5)	1	0	8
Intra-Request	3 (3)	6	0	9
Inter-Request	2 (2)	29	1	32
Total	12	36	1	49

We use 35 different combinations of tags and keywords, e.g., "deadlock," "database," "MySQL," and "web application," for question search. For searches returning more than 50 questions, we include the first 50 with the highest votes. Otherwise, all returned questions are included. To this end, we obtain an initial set of 81 unique questions. We then manually filter out questions without sufficient information for us to understand, e.g., questions with no answer or no discussion. Following this process, we finally obtain a set of 27 questions.

For each bug report and StackOverflow question, two authors first independently examine all available information, including description, discussion, database log, source code, and fixes to make their own conclusion. Then, the two inspectors cross-check with each other with more authors involved in the discussion to reach a final conclusion.

III. **RQ1:** Overall Deadlock Characteristics

In this section, we first discuss the overall characteristics of the deadlocks we collect, and we then discuss how application differences affect these characteristics.

A. *Overall Characteristics of Collected Deadlocks*

We first categorize web-application deadlocks based on the number of HTTP requests and the types of resources involved in deadlocks. On request numbers, we categorize them into non-request, intra-request, and inter-request deadlocks, which need zero, one, and more than one HTTP request, respectively. On resource types, we differentiate database locks, thread synchronization that includes locks and condition variables, and other locks explicit in application code, e.g., cache locks. Table I shows the numbers of deadlocks involving different numbers of requests and different types of resources for each application, and Table II shows the accumulated numbers.

Table III. Patterns of database-lock deadlocks and their numbers

Pattern	Nested TXes	Simple Cycles			Cycles with a Lock Held by Multi TXes			Cycles with Lock Queues						Total
		Pattern-1	Pattern-2	Pattern-3	Pattern-4	Pattern-5	Pattern-6	Pattern-7	Pattern-8	Pattern-9	Pattern-10	Pattern-11	Pattern-12	
# in App	4	6	0	0	2	4	7	2	2	3	0	3	3	36
# in SO	0	2	6	1	5	1	8	0	2	0	1	0	1	27

Among the 8 non-request deadlocks, 7 of them are on thread locks in applications developed with Java. These deadlocks either happen during the starting, shutdown, or restarting phase, or they are triggered while performing offline or background tasks. In these scenarios, deadlocks happen due to concurrency internal to the language but not due to external HTTP requests that arrive concurrently. Thus, it is not surprising that these 7 deadlocks are in applications developed with Java. 5 of them only involve locks. Among them, 4 are fixed by removing unnecessary locks, and the remaining 1 is fixed by changing application logic to make the two deadlock parties not concurrent; The other 2 involve condition variables, and they are fixed by adding the missing signal or removing the untimely wait. Overall, they are similar to classical thread deadlocks in Java. The remaining non-request deadlock is in a Node.js web application. The deadlock happens when the web application starts after a database upgrade, and it involves concurrent `UPDATE` queries. To fix this deadlock, programmers choose to issue these queries sequentially.

Among the 9 intra-request deadlocks, 3 of them are on thread locks in applications developed with Django, which is a Python-based web framework with ORM support. These deadlocks are all due to recursive lock operations on the same lock, and they are fixed by either using a reentrant lock instead of a normal lock or removing unnecessary calls that try to acquire the same lock. In the remaining 6 intra-request deadlocks on database locks, 4 happen in *Odoo*, which uses PostgreSQL as its backend database, and 2 happen in *MediaWiki*, which involves asynchronous execution, and we will discuss their deadlock patterns in Section IV.

Among the 32 inter-request deadlocks, 3 of them are not on database locks, where 2 are on thread locks and 1 is on a cache lock. They happen all due to missing `unlock` calls while handling one HTTP request, and they are fixed by adding the missing `unlock` calls. The remaining 29 are all on database locks, and we will detail them in Section IV.

From the discussion above, we can see that existing techniques can handle the studied deadlocks on thread synchronization or cache lock, regardless of the request number.

From the accumulated numbers in Table II, we can see that inter-request deadlocks are more common than non-request and intra-request deadlocks, which is likely due to the nature of web applications that their core logic is handling concurrent requests. We can also see that deadlocks on database locks are more common, which is likely due to the deep coupling between web applications and databases. To this end, inter-request deadlocks on database locks are the most common, and they are also the most challenging for existing techniques to handle due to two challenges, i.e., they require new techniques to (1) analyze the relationship between different requests and (2) model database locking behavior. To handle non-request and intra-request deadlocks on database locks, while they would not exhibit the first challenge, we still need to handle the second challenge.

For the relationship between request numbers and deadlock resources, we can see that both thread locks and database locks could be involved in non-request, intra-request, and inter-request deadlocks. Therefore, they are two orthogonal dimensions for web-application deadlocks.

B. Application Differences vs. Deadlock Characteristics

For the relationship between deadlock resources and development languages, deadlocks on thread synchronization are more common in web applications developed with languages that provide mature support for concurrency and synchronization, i.e., Java in our case, but deadlocks on threaded synchronization can also happen in applications developed with other languages, as more languages have now gradually added support for concurrency and synchronization.

For the relationship between development paradigms and numbers of deadlocks, we can see classical applications have more deadlocks compared with web applications based on ORM frameworks or Node.js. Note that we also searched many applications with results of zero deadlocks, as described in Section II. This result could be due to two reasons. First, classical web applications generally have a longer development history. Secondly, ORM web-application developers reportedly prefer not to use transactions in their code [29], which is a necessary condition for database-lock deadlocks to happen.

IV. **RQ2:** Patterns of Database-Lock Deadlocks

Following the process discussed in Section II, we identify 36 deadlock bugs on database locks from real-world web applications and 27 such deadlocks from StackOverflow questions. As database-lock deadlocks happen between concurrent transactions, but the source of concurrency does not affect the patterns for database-lock deadlocks much, we include all non-request, intra-request, and inter-request cases in this section.

From these cases, we summarize four patterns of deadlocks on database locks that differ on the types of resources involved in deadlock hold-and-wait cycles, and Table III shows the overall results. Specifically, in the order of increasing complexity, the four cycle patterns are: (1) *Nested Transactions*, where a program creates two database connections in one thread, starts a transaction in each connection, and requests two conflicting locks, and this is similar to deadlocks caused by nested lock acquisition in multi-threaded programs; (2) *Simple Cycles* that involve locks on two rows; (3) *Cycles with a Lock Held by Multiple Transactions*, which involve locks that can be held by multiple transactions simultaneously, and they require extra modeling efforts; and (4) *Cycles with Lock Queues* that further involve lock queues, which is due to how locks are implemented internally in databases.

In this section, we first provide necessary background concepts on database locking, and we then detail the four deadlock cycle patterns with more subpatterns and concrete examples. As our goal is to help application developers understand database-lock deadlocks that they may encounter in the future, our subpatterns and examples are very detailed. We do not try to exhaustively enumerate all possible patterns that may occur in theory, but we categorize and show real-world cases that we see in web applications and StackOverflow questions.

A. Background on Locking Strategy

All database-lock deadlocks that we study are either on MySQL/InnoDB or PostgreSQL. Both MySQL/InnoDB and PostgreSQL use multiversion concurrency control (MVCC) and provide four isolation levels following SQL standard, i.e., Read Uncommitted, Read Committed, Repeatable Read, and Serializable, but their locking strategies are different. In the deadlocks that we study, 32 application and 22 StackOverflow deadlocks are on MySQL/InnoDB, and 4 application and 5 StackOverflow deadlocks are on PostgreSQL.

To understand the deadlocks on PostgreSQL locks, only general knowledge of standard SQL is needed, e.g., the concepts of clustered index, secondary index, primary key, and non-primary index. Such knowledge is assumed in this section. Next, we describe concepts that are fundamental for understanding deadlocks on MySQL/InnoDB locks. Due to space limitations, we are not trying to be comprehensive in this subsection, but we focus on two concepts, i.e., lock modes and lock types. Later in this section, we will describe the mode and type of locks being requested by each query in our examples.

In MySQL/InnoDB, locks can be in two modes: (1) a *shared* (S) lock permits the transaction that holds the lock to read some rows, and (2) an *exclusive* (X) lock permits the transaction to modify some rows. Locks can be in one of four types: (1) *Record Lock*, which is a lock on an index record, (2) *Gap Lock*, which is a lock on a gap between index records, or a lock on the gap before the first or after the last index record, (3) *Next-Key Lock*, which is a combination of a record lock on the index record and a gap lock on the gap before the index record, and (4) *Insert-Intention Lock*, which is a type of gap lock set by `INSERT` operations prior to row insertion.

Under MVCC, locks are requested automatically for SQL queries based on the isolation level, and queries could be blocked if the requested locks conflict with locks granted to other transactions. Unless otherwise specified, the isolation level in our studied bugs is repeatable read. All locks are released when a transaction is committed or aborted. Transactions can be started and committed explicitly, or a query that is not in any transaction is a transaction by itself.

B. Cycles with Nested Transactions

The 4 intra-request deadlocks in PostgreSQL-backed *Odoo* are due to nested transactions in one execution thread, where one request handler first makes a database connection, starts one transaction, and requests one lock, and it then makes a new database connection within the same execution thread, starts a new transaction, and requests a conflicting lock.

```
CREATE TABLE live_measures(
  UUID VARCHAR(40) NOT NULL, ...
);
ALTER TABLE live_measures ADD CONSTRAINT PK_LIVE_MEASURES
     PRIMARY KEY(UUID);

START TRANSACTION; /* TX1 */
UPDATE live_measures SET ... WHERE UUID=2;
                START TRANSACTION; /* TX2 */
                DELETE FROM live_measures WHERE UUID=1;
UPDATE live_measures SET ... WHERE UUID=1;
                DELETE FROM live_measures WHERE UUID=2;
```

Listing 2. *Sonar* #11097

C. Simple Cycles

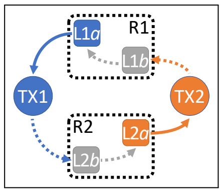

Fig. 1. A simple cycle

Figure 1 shows the simple deadlock cycle with locks on two resources `R1` and `R2`. In the diagram, transaction `TX1` holds lock `L1a` and waits for lock `L2b`, and transaction `TX2` holds lock `L2a` and waits for lock `L1b`. Further, locks `L1a` and `L1b` are conflicting, and locks `L2a` and `L2b` are conflicting. Note that `L1a` and `L1b` can be one lock, and `L2a` and `L2b` can also be one lock. Depending on how many SQL queries are involved, we further divide deadlocks with simple cycles on database locks into three categories with four, three, and two queries, respectively.

[Pattern-1] Simple Cycles with Four Queries

Description: Pattern-1 deadlocks involve four queries from two transactions, with two queries from each transaction, and these queries access the database with primary-key values or unique-index values specified.

Example: Listing 2 shows the deadlock in *Sonar* #11097 [5]. The four queries involved in the deadlock either `UPDATE` or `DELETE` one row with values of the primary key specified. Thus, they all acquire a record lock for its corresponding row in the exclusive mode, but the two transactions acquire the two locks in the opposite order, resulting in a deadlock.

[Pattern-2] Simple Cycles with Three Queries

Description: Pattern-2 deadlocks involve three queries from two transactions, with two queries in one transaction and one query in the other transaction. The one query could request multiple locks due to several different reasons: full table scan during query execution, multiple tables being involved, or multiple indexes being involved.

Examples: Listings 3 and 4 show two examples where one query leads to a full table scan and locks multiple primary-key records, and they are based on StackOverflow questions #40653848 [16] and #1851528 [20], respectively. In Listing 3, the `SELECT` subquery of `INSERT` in `TX2` will perform a full table scan and acquire a shared record lock on each primary-key record that satisfies the `WHERE` condition. Although there is an index on `type`, the database engine still decides to perform

```
CREATE TABLE problem_table(
   id INT(11) NOT NULL,
   type enum('TYPE1','TYPE2','TYPE3') NOT NULL,
   source VARCHAR(16) DEFAULT NULL,
   PRIMARY KEY (id),
   KEY type_idx (type), ...
);

START TRANSACTION; /* TX1 */
UPDATE problem_table SET ... WHERE id=2;
                    START TRANSACTION; /* TX2 */
                    INSERT INTO temp SELECT ... FROM
                          problem_table p WHERE p.type IN
                          ('TYPE1', 'TYPE2') AND p.source='FOO';
UPDATE problem_table SET ... WHERE id=1;
```

Listing 3. StackOverflow #40653848

```
CREATE TABLE jobs(
   jid INT(11) NOT NULL,
   status VARCHAR NOT NULL, ...
   PRIMARY KEY (jid)
);

START TRANSACTION; /* TX1 */
UPDATE jobs SET ... WHERE jid=2;
                     START TRANSACTION; /* TX2 */
                     SELECT ... FROM jobs WHERE status='new' FOR
                           UPDATE;
UPDATE jobs SET ... WHERE jid=1;
```

Listing 4. StackOverflow #1851528

```
START TRANSACTION; /* TX1 */
INSERT INTO phppos_sales VALUES (...);
                     START TRANSACTION; /* TX2 */
                     CREATE temporary TABLE temp SELECT ... FROM
                           phppos_sales_items INNER JOIN
                           phppos_sales ON ... INNER JOIN ...
                           WHERE ...;
INSERT INTO phppos_sales_items VALUES (...);
```

Listing 5. StackOverflow #23768456

a full table scan. In Listing 4, the SELECT FOR UPDATE query in TX2 will perform a full table scan as well and acquire an exclusive record lock on each primary-key record that satisfies the WHERE condition. The database engine performs a full table scan in this case, as the field in the WHERE condition is not indexed. In both cases, the two queries from TX1 request exclusive record locks on two rows but in an order opposite with the order that the query from TX2 locks the same two rows during the full table scan.

Listing 5 shows an example based on StackOverflow question #23768456 [18], and it is one example where one query locks rows from two different tables due to joined tables. In TX2, the SELECT subquery of CREATE is performed on a table joined from two existing tables. For each row matching the WHERE condition, the corresponding row in table phppos_sales_items will be locked first, and then the corresponding row in table phppos_sales will be locked. On the other hand, the two queries in TX1 request exclusive record locks on the two rows of these two tables in a different order, resulting in a deadlock.

Listing 6 shows an example based on StackOverflow question #2560070 [21], where one query locks rows from two tables due to foreign keys. In TX1, the SELECT FOR UPDATE query requests an exclusive lock on the row with id=1000 in table A. Then, the INSERT query in TX2 first gets an exclusive record lock on the row with id=1 just being inserted in table

```
create table A (id INT(11) PRIMARY KEY);
create table B (
   id INT(11) PRIMARY KEY,
   aid INT(11), ...
   FOREIGN KEY (aid) REFERENCES A(id)
);

START TRANSACTION; /* TX1 */
SELECT * FROM A WHERE id=1000 FOR UPDATE;
                     START TRANSACTION; /* TX2 */
                     INSERT INTO B (id, aid, ...)
                           VALUES (1, 1000, ...);
INSERT INTO B (id, aid, ...)
      VALUES (1, 1000, ...);
```

Listing 6. StackOverflow #2560070

```
CREATE TABLE tab1 (
   id INT(11) NOT NULL AUTO_INCREMENT,
   sn VARCHAR(20) NOT NULL,
   is_fetch TINYINT(1) NOT NULL DEFAULT '0' , ...
   PRIMARY KEY (id),
   KEY sn (sn),
   KEY is_fetch (is_fetch),
);

START TRANSACTION; /* TX1 */
SELECT sn FROM tab1 WHERE is_fetch=0
      LIMIT 200 FOR UPDATE;
                     START TRANSACTION; /* TX2 */
                     INSERT IGNORE INTO tab1 (sn, is_fetch, ...)
                           VALUES ('4287', 0, ...);
UPDATE tab1 SET is_fetch=1
    WHERE sn in ('4287', ...);
```

Listing 7. StackOverflow #24327317

B, and it will then request a shared record lock on the row with id=1000 in table A, as the primary key of table A is a foreign key in table B. However, this request from TX2 is blocked due to the lock on that row held by TX1. Finally, the INSERT query in TX1 will also try to insert into table B, but it gets blocked during duplicate-key checking by TX2, as a row satisfying id=1 has been inserted into table B by TX2 already.

Listing 7 shows an example based on StackOverflow question #24327317 [3], where one query locks rows in two indexes. In TX1, the SELECT FOR UPDATE query acquires an exclusive next-key lock on every record in index is_fetch satisfying is_fetch=0 and a gap lock on the range after the last record satisfying is_fetch=0. These ranges are locked to prevent other transactions from inserting records satisfying is_fetch=0 in the is_fetch index concurrently. Then, the INSERT query in TX2 inserts a row whose is_fetch field equals 0. It successfully inserts the record to the primary index and acquires an exclusive lock on the newly inserted primary-index record, but it gets blocked while requesting an exclusive insert-intention lock on secondary index is_fetch, as it falls into the range after last is_fetch=0 record, which has been locked by TX1. Finally, the database engine chooses to perform a full table scan based on existing data in the table while executing the UPDATE query in TX1. During this process, it tries to acquire an exclusive next-key lock on every primary-key record, including the newly inserted row, and thus gets blocked as the new row is inserted by TX2.

[Pattern-3] Simple Cycles with Two Queries

Description: Pattern-3 deadlocks involve two queries from two transactions, and each query requests multiple locks.

```
CREATE TABLE fruit_setting (
  id BIGINT(20) NOT NULL AUTO_INCREMENT,
  aid VARCHAR(32) NOT NULL,
  eid VARCHAR(32) NOT NULL,
  mykey VARCHAR(32) NOT NULL, ...
  PRIMARY KEY (id),
  KEY i_aid_mykey (aid, mykey),
  UNIQUE KEY i_eid_mykey (eid, mykey), ...
);
INSERT INTO fruit_setting (id, aid, eid, mykey, ...)
    VALUES (1, 'a', 'b', 'a', ...);
INSERT INTO fruit_setting (id, aid, eid, mykey, ...)
    VALUES (2, 'a', 'a', 'a', ...);

START TRANSACTION; /* TX1 */
UPDATE fruit_setting SET ... WHERE
    aid='a' and mykey='a';
              START TRANSACTION; /* TX2 */
              UPDATE fruit_setting SET ... WHERE
                  eid IN ('a', 'b') and mykey='a';
```

Listing 8. StackOverflow #65519414

Example: Listing 8 shows an example based on StackOverflow question #65519414 [2]. The table schema contains 2 different indexes. One consists of columns `eid` and `mykey`, and the other consists of `aid` and `mykey`. The `UPDATE` query in `TX1` updates the records via searching in the order of index `i_aid_mykey`. Since the two existing rows have the same values for `aid` and `mykey`, the two rows will be accessed in an order based on the values of primary key `id`. Specifically, the query will request an exclusive lock first on the row with `id=1` and then on the row with `id=2`. On the other hand, the `UPDATE` query in `TX2` updates the records via searching in the order of index `i_eid_mykey`. With the two existing rows, it will request exclusive locks on the two rows in an opposite order as the query in `TX1`, resulting in a deadlock.

D. Cycles with a Lock Held by Multiple Transactions

Deadlocks involving a lock held by multiple transactions cannot be modeled with the simple cycle already described, and Figure 2 shows the deadlock cycle that we come up with to model deadlocks involving such locks. In the diagram, transactions `TX1` and `TX2` both hold the same lock on record `R`. Then, they both request the exclusive lock, which conflicts with the lock held by the other transaction, and thus the two transactions get blocked by each other, resulting in a deadlock. Depending on the type of the lock held by multiple transactions, we further divide them into three types. Below, we omit the **Description** paragraph if the pattern name is self-explanatory and we do not have more to add.

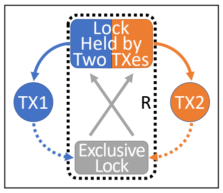

Fig. 2. A cycle with a lock held by multiple transactions

[Pattern-4] Multiple TXes Holding One Shared Record Lock

Description: The lock held by multiple transactions is a shared record lock, and this is the classical conversion case [58].

Examples: Listing 9 shows an example based on StackOverflow question #5353877 [22]. First, the `SELECT` subqueries of `INSERT` in both transactions acquire a shared record lock on the row with `id=10` in table `trades`. Then, the `UPDATE` queries in both transactions ask for an exclusive record lock on the same row, but both get blocked by the shared record lock held by the other transaction. Listing 10 shows a similar example from *Lobsters* #39 [17]. The `INSERT` queries in both transactions acquire a shared record lock on the row with `id=1` in table `stories`, but this is due to foreign key, which is the same as the case in Listing 6.

```
CREATE TABLE tradeshistory (
  PRIMARY KEY (id), ...
);
CREATE TABLE trades (
  PRIMARY KEY (id), ...
);

START TRANSACTION; /* TX1 */
INSERT INTO tradeshistory (SELECT
    trades.* FROM trades WHERE id=10);
              START TRANSACTION; /* TX2 */
              INSERT INTO tradeshistory (SELECT
                  trades.* FROM trades WHERE id=10);
UPDATE trades SET ... WHERE id=10;
              UPDATE trades SET ... WHERE id=10;
```

Listing 9. StackOverflow #5353877

```
CREATE TABLE votes ( ...,
  story_id BIGINT(20) NOT NULL, ...,
  FOREIGN KEY (story_id) REFERENCES stories(id);
);
CREATE TABLE stories (
  id BIGINT(20) NOT NULL PRIMARY KEY, ...
)

START TRANSACTION; /* TX1 */
INSERT INTO votes (story_id, ...)
    VALUES (1, ...);
              START TRANSACTION; /* TX2 */
              INSERT INTO votes (story_id, ...)
                  VALUES (1, ...);
UPDATE stories SET ... WHERE id=1;
              UPDATE stories SET ... WHERE id=1;
```

Listing 10. *Lobsters* #39

[Pattern-5] Multiple TXes Holding One Shared Gap Lock

Example: The example from MySQL's official manual in Listing 1 as mentioned in Section I is a Pattern-5 deadlock. In `TX1`, the `INSERT` query acquires an exclusive record lock on the row inserted. In `TX2` and `TX3`, the `INSERT` query asks for a shared record lock during duplicate-key checking because the query inserts the primary key. When `TX1` is rolled back, the `INSERT` queries in `TX2` and `TX3` both get the shared gap lock because the row inserted by `TX1` does not exist anymore. Then, the `INSERT` queries in both transactions ask for the same exclusive insert-intention lock, but both get blocked by the shared gap lock held by the other transaction.

[Pattern-6] Multiple TXes Holding One Exclusive Gap Lock

Description: The lock held by multiple transactions is an exclusive gap lock. Although in the exclusive mode, an exclusive gap lock can be held by multiple transactions simultaneously.

Example: Listing 11 shows *MediaWiki* #214035 [6]. With existing data in table `page_restrictions`, the `DELETE` queries in both transactions acquire an exclusive gap lock on the same range, as the `WHERE` conditions in both queries match no existing rows but fall into the same range. Then,

```
CREATE TABLE page_restrictions (
  pr_id INT unsigned NOT NULL PRIMARY KEY AUTO_INCREMENT,
  pr_page INT NOT NULL,
  pr_type VARBINARY(60) NOT NULL, ...
)
CREATE UNIQUE INDEX pr_pagetype ON page_restrictions
    (pr_page,pr_type);

START TRANSACTION; /* TX1 */
DELETE FROM page_restrictions WHERE
    pr_page=125 and pr_type='move';
              START TRANSACTION; /* TX2 */
              DELETE FROM page_restrictions WHERE
                  pr_page=150 and pr_type='move';
INSERT INTO page_restrictions
    (pr_page,pr_type,...) VALUES
    (125,'move',...);
              INSERT INTO page_restrictions
                  (pr_page,pr_type,...) VALUES
                  (150,'move',...);
```

Listing 11. *MediaWiki* #214035

```
CREATE TABLE cache_config(
  cid VARCHAR(255) NOT NULL, ...
  PRIMARY KEY (cid)
);

START TRANSACTION; /* TX1 */
DELETE FROM cache_config WHERE cid=1;
              START TRANSACTION; /* TX2 */
              DELETE FROM cache_config WHERE cid=1;
INSERT INTO cache_config (cid, ...)
    VALUES(1, ...);
```

Listing 12. *Drupal* #2336627

the INSERT queries in both transactions ask for an exclusive insert-intention lock on the same range, and they get blocked by the exclusive gap lock held by the other transaction.

Besides DELETE, SELECT FOR UPDATE or UPDATE can also have WHERE conditions matching no rows, thus acquiring exclusive gap locks and causing the same type of deadlocks.

E. Cycles with Lock Queues

Each MySQL/InnoDB record internally maintains a queue, and queries requesting locks on the same record are queued in the order these requests are made. Therefore, queries enqueued later need to wait for queries enqueued earlier. Figure 3 shows the deadlock cycle that we come up with to model deadlocks involving such wait relationships on lock queues. In the diagram, (1) TX1 acquires La, (2) TX2 requests Lb but gets blocked by TX1, and TX2 is put into the queue corresponding to record R, and (3) TX1 requests Lc that conflicts with Lb being requested by TX2, and thus TX1 is blocked by TX2 and put into the same queue after TX1. Deadlocks involving lock queues all have three queries, and we further divide such deadlocks based on the query types and lock types involved in the deadlock. We group the examples for Patterns 7, 8, and 9 together as they share the same query pattern. ‘X’ and ‘S’ in the following pattern names are lock modes.

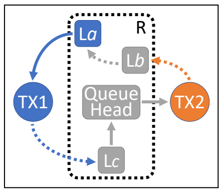

Fig. 3. A cycle with a lock queue

[Pattern-7] DELETE-DELETE-INSERT Acquiring X Record Lock, X Record Lock, and S Next-key Lock

```
CREATE TABLE user_properties (
  up_user INT NOT NULL,
  up_property VARBINARY(255) NOT NULL, ...
)
CREATE UNIQUE INDEX user_properties_user_property ON
    user_properties (up_user,up_property);

START TRANSACTION; /* TX1 */
DELETE FROM user_properties WHERE
    up_user=1 AND up_property='aaa';
              START TRANSACTION; /* TX2 */
              DELETE FROM user_properties WHERE
                  up_user=1 AND up_property='aaa';
INSERT INTO user_properties (up_user,
    up_property, ...) VALUES(1, 'aaa', ...);
```

Listing 13. *MediaWiki* #38116

[Pattern-8] DELETE-DELETE-INSERT Acquiring X Record Lock, X Next-Key Lock, and S Next-Key Lock

[Pattern-9] DELETE-DELETE-INSERT Acquiring X next-key Lock, X Next-Key Lock, and X Insert-Intention Lock

Examples: Listing 12 shows a Pattern-7 deadlock in *Drupal* #2336627 [8]. In TX1, the DELETE query first acquires an exclusive record lock on the row of cid=1 because it uses the primary key to search for records. In TX2, the DELETE query asks for the same exclusive record lock on the same row but gets blocked. Thus, TX2 is put into a wait queue corresponding to the row of cid=1. Finally, the INSERT query in TX1 wants to insert a record with cid=1. Because cid is the primary key of the table, it asks for a shared next-key lock to check if the primary key value to be inserted exists. This lock cannot be granted because it conflicts with the lock requested by the DELETE query in TX2. Thus, TX1 has to wait for TX2 that is currently the head of lock queue for the row of cid=1, completing the hold-and-wait cycle.

Listing 13 shows a Pattern-8 deadlock in *MediaWiki* #38116 [9]. Among all locks that it acquires, the DELETE query in TX1 acquires an exclusive record lock on the unique index satisfying the WHERE condition, as it uses the unique index to search for records. Then, the DELETE query in TX2 requests an exclusive next-key lock on the unique index, but it gets blocked due to the aforementioned exclusive record lock held by TX1. Based on comments from MySQL source code, since in a unique secondary index, there may be different delete-marked versions of a record where only the primary key values differ, next-key locks are used on a secondary index when locking delete-marked records. Finally, the INSERT query asks for a shared next-key lock to check if the new row with up_user=1 AND up_property='aaa' to be inserted may result in duplicates on the unique index. This lock cannot be granted because it conflicts with the lock requested by TX2. Thus, TX1 again has to wait for TX2.

Listing 14 shows a Pattern-9 deadlock in *MediaWiki* #30598 [7]. In this case, the two DELETE queries in both transactions use non-unique indexes to search for records. In TX1, the DELETE query acquires exclusive next-key locks on the two indexes satisfying the WHERE condition because the indexes are non-unique. Then, the DELETE query in TX2 requests the same locks and gets blocked by TX1, and it is put into wait queues corresponding to these two indexes. Finally,

```
CREATE TABLE wb_terms (
  term_row_id INT unsigned NOT NULL PRIMARY KEY
      AUTO_INCREMENT,
  term_entity_id INT unsigned NOT NULL,
  term_entity_type VARBINARY(32) NOT NULL, ...
);
CREATE INDEX wb_terms_entity_id ON wb_terms
     (term_entity_id);
CREATE INDEX wb_terms_entity_type ON wb_terms
     (term_entity_type);

START TRANSACTION; /* TX1 */
DELETE FROM wb_terms WHERE term_entity_id=1
    AND term_entity_type='A';
              START TRANSACTION; /* TX2 */
              DELETE FROM wb_terms WHERE term_entity_id=1
                  AND term_entity_type='A';
INSERT INTO wb_terms (term_entity_id,
    term_entity_type, ...) VALUES (1, 'A'...);
```

Listing 14. *MediaWiki* #44547

```
CREATE TABLE parent (id INT(11) PRIMARY KEY);
CREATE TABLE child (
  id INT(11) PRIMARY KEY,
  parent_id INT(11),
  FOREIGN KEY (parent_id) REFERENCES parent(id)
);

START TRANSACTION; /* TX1 */
INSERT INTO child (id, parent_id)
    VALUES (10, 1);
              START TRANSACTION; /* TX2 */
              SELECT id FROM parent WHERE id=1
                   FOR UPDATE;
SELECT id FROM parent WHERE id=1
    FOR UPDATE;
```

Listing 15. StackOverflow #41015813

the `INSERT` query in `TX1` wants to insert a record sharing the same values with the `DELETE` query on the non-unique indexes. Since the indexes are not unique, the `INSERT` query does not need to perform the duplicate key checking, but it will directly request an exclusive insert-intention lock. This lock cannot be granted because it conflicts with the lock requested by `TX2`. Thus, `TX1` has to wait for `TX2`.

[Pattern-10] `INSERT-SELECT FOR UPDATE-SELECT FOR UPDATE` Acquiring S Record Lock, X Record Lock, and X Record Lock

Example: Listing 15 shows an example based on StackOverflow question #41015813 [4]. The `INSERT` query in `TX1` inserts a row of `parent_id=1` into table `child`, and it acquires a shared lock on the record satisfying `id=1` in table `parent` because of the foreign-key constraint between these two tables. Then, `TX2`'s `SELECT FOR UPDATE` query will ask for an exclusive lock on the record satisfying `id=1` in table `parent`. This lock request from `TX2` is blocked by `TX1`. After that, `TX1`'s `SELECT FOR UPDATE` query also asks for an exclusive lock on the same record. This lock request from `TX1` cannot be granted because it conflicts with the lock requested by `TX2`, completing the deadlock cycle.

[Pattern-11] `INSERT-INSERT-DELETE` Acquiring X Record Lock, S Record Lock, and X Next-Key Lock

Example: Listing 16 shows *OpenMRS* #674 [11]. In `TX1`, the `INSERT` query inserts a new row in the table `cache_config`

```
CREATE TABLE cache_config(
  idset_key CHAR(40) NOT NULL,
  member_id INT(11) NOT NULL,
  PRIMARY KEY (idset_key, member_id)
);

START TRANSACTION; /* TX1 */
INSERT INTO reporting_idset
    (idset_key, member_id) VALUES (5, 5);
              START TRANSACTION; /* TX2 */
              INSERT INTO reporting_idset
                  (idset_key, member_id) VALUES (5, 5);
DELETE FROM reporting_idset
    WHERE idset_key=5;
```

Listing 16. *OpenMRS* #674

```
CREATE TABLE wbc_entity_usage (
  eu_row_id     BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  eu_entity_id  VARBINARY(255) NOT NULL,
  eu_aspect     VARBINARY(37) NOT NULL,
  eu_page_id    INT NOT NULL
);
CREATE UNIQUE INDEX eu_entity_id ON wbc_entity_usage (
    eu_entity_id, eu_aspect, eu_page_id ); ...

START TRANSACTION; /* TX1 */
INSERT INTO wbc_entity_usage
    (eu_page_id=10, eu_aspect='10', eu_entity_id='10');
              START TRANSACTION; /* TX2 */
              INSERT INTO wbc_entity_usage
                  (eu_page_id=10, eu_aspect='10',
                   eu_entity_id='10');
INSERT INTO wbc_entity_usage
    (eu_page_id=9, eu_aspect='9', eu_entity_id='9');
```

Listing 17. *MediaWiki* #192349

and acquires an exclusive record lock on that row. Then, the `INSERT` query in `TX2` tries to insert the same record and asks for a shared record lock on that row for duplicate-key checking. It gets blocked by `TX1` and is put into a wait queue. After that, the `DELETE` query in `TX1` tries to delete records satisfying `idset_key=5`, including the newly inserted record by the previous `INSERT` query in `TX1`. Since `idset_key` is part of the multi-column primary key, it will ask for an exclusive next-key lock on every record satisfying the where condition. This lock cannot be granted as it conflicts with the lock requested by `TX2`, completing the deadlock cycle.

[Pattern-12] `INSERT-INSERT-INSERT` Acquiring X Record Lock, S Next-Key Lock, and X Insert-Intention Lock

Example: Listing 17 shows *MediaWiki* #192349 [12]. The first `INSERT` query in `TX1` acquires an exclusive record lock on both the row being inserted and the unique index with `eu_page_id=10, eu_aspect='10', eu_entity_id='10'`. The `INSERT` query in `TX2` requests a shared next-key lock on the unique index during duplicate-key checking. `TX2` gets blocked by `TX1` and is put into a wait queue. The second `INSERT` query in `TX1` passes the duplicate-key checking, as `eu_page_id=9, eu_aspect='9', eu_entity_id='9'` is not in the table, and it proceeds to request an exclusive insert-intention lock. When existing data in the table makes the insert-intention lock be on the record of `eu_page_id=10, eu_aspect='10', eu_entity_id='10'`, the insert-intention lock requested by `TX1` conflicts with the lock requested by `TX2`. Thus, `TX1` is also blocked by `TX2`.

Table IV. Fixing strategies

Fixing Strategies		Total
Fix	Enforcing lock order by changing query order	3
	Omitting unnecessary queries	3
	Omitting unnecessary SELECT FOR UPDATE locks	1
	Removing unnecessary transactions	4
	Avoiding concurrent execution with app-level lock	3
	Avoiding concurrent execution with ordered execution	3
	Avoiding conflicting by changing queries or logic	7
	Avoiding nested transactions	3
	Avoiding using database	1
Reduce	Splitting a large transaction into smaller ones	2
	Reduce the number of resources requested	3
Catch and retry		3

F. *Discussion*

Among those PostgreSQL-lock deadlocks, the 4 from applications are due to cycles with nested transactions, and the 5 from StackOverflow questions are of Patterns 1, 2, and 4. Deadlocks of these patterns can be understood with general SQL knowledge, while deadlocks on MySQL/InnoDB locks are more challenging for application developers to understand.

To help application developers in tackling this challenge, we categorize deadlocks on MySQL/InnoDB locks in fine granularity and provide a concrete example for each pattern that we observe in our deadlock set. We believe the knowledge gained through our examples will be valuable for application developers to understand and diagnose deadlocks that they may encounter, even for those beyond the patterns that we observe. For tool researchers and developers, our results suggest that existing tool support is not sufficient and call for more effort in this area. Specifically, our results on database-lock deadlocks reveal cycle patterns that existing techniques on deadlocks have not accounted for.

V. **RQ3:** Fixes for Database-Lock Deadlocks

Unlike hold-and-wait cycle patterns, the fixing strategies for database-lock deadlocks are much straightforward to understand. Table IV shows the different fixing strategies used for the 36 deadlocks from real-world applications and their corresponding numbers. On the high level, fixing strategies for database-lock deadlocks can be categorized as (1) completely eliminating the possibility of deadlocks, (2) reducing the chance of deadlocks, or (3) adding catch-and-retry.

The majority, i.e., 28 out of 36, of the studied database-lock deadlocks are completely fixed with various strategies. The first three strategies can be viewed as different ways to break the hold-and-wait cycle. The next three strategies can be viewed as different ways to avoid concurrent transactions. The last three strategies are more application-specific. In particular, avoiding nested transactions is only used to fix *Odoo* intra-request deadlocks, and the "avoiding using database" strategy is used when the data can be moved to cache.

5 deadlocks are not completely fixed, but developers either reduce transaction length or reduce the number of resources requested in transactions to reduce the chance of deadlocks. This could happen if a complete fix is too complex, and the chance of deadlocks can be reduced to an acceptable level.

In the remaining 3 cases, developers take the catch-and-retry approach by adding code to retry transactions on deadlocks, and the chance of deadlocks is likely considered as acceptable.

In the case of StackOverflow questions, 10 of them have accepted answers with fixing strategies proposed. The proposed strategies are no different from what we see in real-world web applications. Since the actual patch being applied in practice is only mentioned in one StackOverflow question, we do not include it in Table IV.

VI. Related Work

Earlier in this paper, we have discussed some related work on deadlocks in multi-threaded programs and web applications. Our results suggest that existing work cannot handle a large portion of real-world deadlocks in web applications, especially those inter-request deadlocks on database locks. While there are studies focusing on concurrency bugs in web applications [37], [57], [64], they do not cover deadlocks.

Server-side web applications have been the subject of a lot of existing research, and we next briefly discuss other related work on server-side web applications. Many different techniques have been proposed for improving their reliability [26]–[28], [38], [42], [53]–[55], [61], mostly focusing on program analysis, bug detection, input generation, or automated repair. Techniques focusing on the security aspect of web applications have also been proposed, e.g., auditing [48], [63], intrusion detection and recovery [33], [34], [52], identifying information disclosure [35]. However, none of them handles deadlocks.

VII. Threats to Validity

Our study may be subject to several validity threats. Next, we describe potential threats and our ways to address them. (1) We may not include all representative web applications. To minimize this threat, we choose popular open-source applications with a significant user base from state-of-the-art studies on web applications, and we search deadlocks in all applications from these studies that are still available. We further include StackOverflow questions to further enrich our understanding of deadlocks on database locks. Our results show the characteristics are similar for database-lock deadlocks from web-application bugs and those based on StackOverflow questions. So the characteristic study results can likely be generalized to other web applications. (2) We may miss relevant bug reports while searching for deadlocks. We mitigate this threat by using keyword search in both bug descriptions and comments together with bug categories and tags. (3) We inspect bug reports manually. To alleviate this threat, each report is examined by at least two authors, and the group discusses the bug report together to reach a consensus.

VIII. Conclusion

In this paper, we characterize deadlocks from real-world web applications based on the number of HTTP requests and the types of resources involved. For deadlocks on database locks, we further categorize their hold-and-wait cycle patterns and fix strategies. The patterns and concrete examples presented in this paper can help application developers understand and diagnose deadlocks that they may encounter. Our study results can also guide future research in combating deadlocks in web applications.

References

[1] "15.7.3 Locks Set by Different SQL Statements in InnoDB," https://dev.mysql.com/doc/refman/8.0/en/innodb-locks-set.html.

[2] "A puzzled MySQL deadlock caused by two update..in," https://stackoverflow.com/questions/65519414.

[3] "An insert caused a deadlock in InnoDB. How did this happen?" https://stackoverflow.com/questions/24327317.

[4] "Avoiding MySQL deadlock when upgrading shared to exclusive lock," https://stackoverflow.com/questions/41015813.

[5] "DB deadlock if user edits issue when project is being analyzed," https://jira.sonarsource.com/browse/SONAR-11097.

[6] "DBError 'Error: 1213 Deadlock found when trying to get lock' on WikiPage::doUpdateRestrictions," https://phabricator.wikimedia.org/T214035.

[7] "Deadlock INSERT IGNORE INTO `flaggedtemplates`," https://phabricator.wikimedia.org/T30598.

[8] "Deadlock on cache_config (DatabaseBackend::setMultiple())," https://www.drupal.org/project/drupal/issues/2336627.

[9] "Deadlock on user account creation leaves account in bad state," https://phabricator.wikimedia.org/T38116.

[10] "Deadlocks - PostgreSQL Documentation," https://www.postgresql.org/docs/current/explicit-locking.html#LOCKING-DEADLOCKS.

[11] "Deadlocks as a result of the speedup by joining on idset table," https://issues.openmrs.org/browse/REPORT-674.

[12] "deadlocks on INSERT IGNORE INTO wbc_entity_usage," https://phabricator.wikimedia.org/T192349.

[13] "Detecting and Ending Deadlocks - SQL Server technical documentation," https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver15#detecting-and-ending-deadlocks.

[14] "Drd: a thread error detector," https://www.valgrind.org/docs/manual/drd-manual.html.

[15] "Helgrind: a thread error detector," https://www.valgrind.org/docs/manual/hg-manual.html.

[16] "Innodb update locking," https://stackoverflow.com/questions/40653848.

[17] "mariadb deadlocked occasionally," https://github.com/lobsters/lobsters-ansible/issues/39.

[18] "mysql CREATE temporary table + Transaction causes deadlock," https://stackoverflow.com/questions/23768456.

[19] "MySQL Deadlock Detection - MySQL 8.0 Reference Manual," https://dev.mysql.com/doc/refman/8.0/en/innodb-deadlock-detection.html.

[20] "Mysql deadlock explanation needed," https://stackoverflow.com/questions/1851528.

[21] "MySQL return Deadlock with insert row and FK is locked 'for update'," https://stackoverflow.com/questions/2560070.

[22] "mysql transaction deadlock," https://stackoverflow.com/questions/5353877.

[23] "Why MySQL InnoDB creates so many deadlocks when Hangfire enques multiple jobs in parallel?" https://stackoverflow.com/questions/53854450/.

[24] R. Agarwal and S. D. Stoller, "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables," in *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, 2006, pp. 51–60, doi: 10.1145/1147403.1147413.

[25] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing (HVC)*, 2005, pp. 191–207, doi: 10.1007/11678779_14.

[26] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*, 2010, pp. 265–274, doi: 10.1145/1806799.1806840.

[27] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in web applications using dynamic test generation and explicit-state model checking," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 474–494, 2010, doi: 10.1109/TSE.2010.31.

[28] S. Athaiya and R. Komondoor, "Testing and analysis of web applications using page models," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 181–191, doi: 10.1145/3092703.3092734.

[29] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Feral concurrency control: An empirical investigation of modern application integrity," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015, pp. 1327–1342, doi: 10.1145/2723372.2737784.

[30] Y. Cai and L. Cao, "Fixing deadlocks via lock pre-acquisitions," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1109–1120, doi: 10.1145/2884781.2884819.

[31] Y. Cai and W. K. Chan, "Magicfuzzer: Scalable deadlock detection for large-scale applications," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 606–616, doi: 10.1109/ICSE.2012.6227156.

[32] Y. Cai, S. Wu, and W. K. Chan, "Conlock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 491–502, doi: 10.1145/2568225.2568312.

[33] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich, "Intrusion recovery for database-backed web applications," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 101–114, doi: 10.1145/2043556.2043567.

[34] R. Chandra, T. Kim, and N. Zeldovich, "Asynchronous intrusion recovery for interconnected web services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 213–227, doi: 10.1145/2517349.2522725.

[35] H. Chen, T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Identifying information disclosure in web applications with retroactive auditing," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 555–569.

[36] T. Cogumbreiro, R. Hu, F. Martins, and N. Yoshida, "Dynamic deadlock verification for general barrier synchronisation," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015, pp. 150–160, doi: 10.1145/2688500.2688519.

[37] J. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the server-side event-driven architecture," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017, pp. 145–160, doi: 10.1145/3064176.3064188.

[38] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, 2007, pp. 151–162, doi: 10.1145/1273463.1273484.

[39] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 237–252, doi: 10.1145/945445.945468.

[40] 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)*, 2014, pp. 353–365, doi: 10.1145/2635868.2635918.

[41] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, 2010, pp. 221–230, doi: 10.1109/DSN.2010.5544315.

[42] M. A. Ghafoor, M. S. Mahmood, and J. H. Siddiqui, "Effective partial order reduction in model checking database applications," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 146–156, doi: 10.1109/ICST.2016.25.

[43] M. Grechanik, B. M. M. Hossain, U. Buy, and H. Wang, "Preventing database deadlocks in applications," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 356–366, doi: 10.1145/2491411.2491412.

[44] M. Grechanik, B. M. Hossain, and U. Buy, "Testing database-centric applications for causes of database deadlocks," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*, 2013, pp. 174–183, doi: 10.1109/ICST.2013.19.

[45] P. Joshi, M. Naik, K. Sen, and D. Gay, "An effective dynamic analysis for detecting generalized deadlocks," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2010, pp. 327–336, doi: 10.1145/1882291.1882339.

[46] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 110–120, doi: 10.1145/1542476.1542489.

[47] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 295–308.

[48] T. Kim, R. Chandra, and N. Zeldovich, "Efficient patch-based auditing for web application vulnerabilities," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 193–206.

[49] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin, "Pulse: A dynamic deadlock detection mechanism using speculative execution," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2005, pp. 31–44.

[50] Y. Lin and S. S. Kulkarni, "Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 237–247, doi: 10.1145/2610384.2610398.

[51] S. Lu, S. Park, E. Seo, and Y. 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)*, 2008, pp. 329–339, doi: 10.1145/1346281.1346323.

[52] D. R. Matos, M. L. Pardal, and M. Correia, "Rectify: Black-box intrusion recovery in paas clouds," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware)*, 2017, pp. 209–221, doi: 10.1145/3135974.3135978.

[53] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Dangling references in multi-configuration and dynamic php-based web applications," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 399–409, doi: 10.1109/ASE.2013.6693098.

[54] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Auto-locating and fix-propagating for html validation errors to php server-side code," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011, pp. 13–22, doi: 10.1109/ASE.2011.6100047.

[55] H. V. Nguyen, H. D. Phan, C. Kästner, and T. N. Nguyen, "Exploring output-based coverage for testing php web applications," *Automated Software Engg.*, vol. 26, no. 1, pp. 59–85, Mar. 2019, doi: 10.1007/s10515-018-0246-5.

[56] H. K. Pyla and S. Varadarajan, "Avoiding deadlock avoidance," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 75–85, doi: 10.1145/1854273.1854288.

[57] Z. Qiu, S. Shao, Q. Zhao, and G. Jin, "Understanding and detecting server-side request races in web applications," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021, p. 842–854, doi: 10.1145/3468264.3468594.

[58] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 2nd ed. USA: McGraw-Hill, Inc., 2000.

[59] M. Samak and M. K. Ramanathan, "Multithreaded test synthesis for deadlock detection," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2014, p. 473–489, doi: 10.1145/2660193.2660238.

[60] ——, "Trace driven dynamic deadlock detection and reproduction," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014, pp. 29–42, doi: 10.1145/2555243.2555262.

[61] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of html generation errors in php applications using string constraint solving," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 277–287, doi: 10.1109/ICSE.2012.6227186.

[62] S. Shao, Z. Qiu, X. Yu, W. Yang, G. Jin, T. Xie, and X. Wu, "Database-access performance antipatterns in database-backed web applications," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 58–69, doi: 10.1109/ICSME46990.2020.00016.

[63] C. Tan, L. Yu, J. B. Leners, and M. Walfish, "The efficient server audit problem, deduplicated re-execution, and the web," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 546–564, doi: 10.1145/3132747.3132760.

[64] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in node.js," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 520–531, doi: 10.1109/ASE.2017.8115663.

[65] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke, "Gadara: Dynamic deadlock avoidance for multithreaded programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 281–294.

[66] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke, "The theory of deadlock avoidance via discrete control," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2009, pp. 252–263, doi: 10.1145/1480881.1480913.

[67] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM)*, 2017, pp. 1299–1308, doi: 10.1145/3132847.3132954.

[68] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, "How not to structure your database-backed web applications: A study of performance bugs in the wild," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 800–810, doi: 10.1145/3180155.3180194.

[69] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu, "Undead: Detecting and preventing deadlocks in production software," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 729–740, doi: 10.1109/ASE.2017.8115684.