Preventing Database Deadlocks in Applications

Mark Grechanik, B. M. Mainul Hossain, Ugo Buy
University of Illinois at Chicago
Chicago, IL 60607
{drmark,bhossa2,buy}@uic.edu

Haisheng Wang*
University of Illinois at Chicago and Oracle Corp.
Redwood City, CA 94065
haisheng.wang@oracle.com

ABSTRACT

Many organizations deploy applications that use databases by sending *Structured Query Language (SQL)* statements to them and obtaining data that result from the execution of these statements. Since applications often share the same databases concurrently, database deadlocks routinely occur in these databases resulting in major performance degradation in these applications. Database engines do not prevent database deadlocks for the same reason that the schedulers of operating system kernels do not preempt processes in a way to avoid race conditions and deadlocks – it is not feasible to find an optimal context switching schedule quickly for multiple processes (and SQL statements), and the overhead of doing it is prohibitive.

We created a novel approach that combines run-time monitoring, which automatically prevents database deadlocks, with static analysis, which detects hold-and-wait cycles that specify how resources (e.g., database tables) are held in contention during executions of SQL statements. We rigorously evaluated our approach. For a realistic case of over 1,200 SQL statements, our algorithm detects all hold-and-wait cycles in less than two seconds. We built a toolset and experimented with three applications. Our tool prevented all existing database deadlocks in these applications and increased their throughputs by up to three orders of magnitude.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; H.2.4 [**Database Management**]: Systems—*Concurrency*

General Terms

Algorithms, Performance, Experimentation

Keywords

database deadlock, Petri net, hold-and-wait cycle

*Dr.Wang completed the work on this project when he was a postdoc in Dr.Grechanik's research group at the Department of Computer Science, University of Illinois at Chicago.

ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia

1. INTRODUCTION

Many organizations and companies deploy *Database-centric applications (DCAs)*, which use databases by sending *transactions* to them—atomic units of work that contain *Structured Query Language (SQL)* statements [12]—and obtaining data that result from the execution of these SQL statements. When DCAs use the same database at the same time, concurrency errors are observed frequently that are known as *database deadlocks*, which is one of the main reasons for major performance degradation in these applications [23, 15]. The responsibility of relational database engines is to provide layers of abstractions to guarantee *Atomicity, Consistency, Isolation, and Durability (ACID)* properties [12]; however, these guarantees do not include freedom from database deadlocks.

In general, deadlocks occur when two or more threads of execution lock some resources and wait on other resources in a circular chain, i.e., in a *hold-and-wait cycle* [8]. Even though database deadlocks occur within database engines and not within DCAs that use these databases, these deadlocks affect the performance of the combined software system, i.e., the DCAs that interact with their databases. A condition for observing database deadlock is that a database should simultaneously service two or more transactions that come from one or more DCAs, and these transactions contain SQL statements that share the same resources (e.g., tables or rows). In enterprise systems, database deadlocks may appear when a new transaction is issued by a DCA to a database that is already used by some other legacy DCA, thus making the process of software evolution error-prone, expensive and difficult.

There are two main reasons why preventing database deadlocks is a hard and open problem. First, databases are general tools that process arriving transactions on demand, making it infeasible to find all hold-and-wait cycles statically. Second, database engines are designed to execute transactions efficiently, and imposing runtime analysis for finding all hold-and-wait cycles adds significant overhead. In short, database engines do not prevent database deadlocks for the same reason that the schedulers of operating system kernels do not preempt processes in a way to avoid race conditions and deadlocks—it is not feasible to find an optimal context switching schedule quickly for multiple processes (and transactions), and the overhead would be prohibitive.

Currently, database deadlocks are typically detected within database engines at runtime using special algorithms that analyze whether transactions hold resources in cyclic dependencies, and these database engines resolve database deadlocks by forcibly breaking the hold-and-wait cycle [2, 22, 28, 16, 15, 12]. That is, once a deadlock occurs, the database rolls back one of the transactions that is involved in the circular wait. Doing so effectively resolves the database deadlock; this is why the database research community has considered this problem solved for a long time. However, this

resolution degrades the performance from the software engineering position, since DCAs should repeat the rolled back transactions to ensure functional correctness.

Unfortunately, this solution is only partially effective even though it is widely used as part of the defensive programming practice, whereby programmers write special database deadlock exception-handling code that typicahandwgraphly repeats aborted transactions. Searching for "database deadlock exception" on the Web yields close to 2,500 web pages, many of which instruct programmers how to handle database deadlock exceptions for different databases. By the time that a database deadlock is resolved, the damage to the performance of the DCA is done, since rolling back transactions, issuing exceptions inside the DCA, and executing defensive code within exception handlers to retry aborted transactions incur a significant performance penalty. Our experiments show in Section 5 that database deadlocks result in up to three orders of magnitude of worsening performance of client/server DCAs when scaling the load up to only 1,000 clients!

To make things worse, when transactions are discarded, the results of valuable and long-running computations are lost, as it is especially evident in case of multi-level and long-lived transactions [12, pages 206-212]. Our interviews with different Fortune 100 companies confirmed that database deadlocks occur on average every two to three weeks for large-scale enterprise DCAs, some of which have been around for over 20 years, with an estimated annual cost of DCA support close to $500K per company. For instance, database deadlocks still occur every ten days on average in a commercial large-scale DCA that handles over 70% of cargo flight reservations in the USA.

Major database vendors acknowledge this problem, and they publish different advice to software engineers on how to avoid database deadlocks and how to handle exceptions that are thrown as result of aborted transactions. These vendors also release tools that help these engineers debug and understand the causes of database deadlocks. Different database deadlock avoidance programming patterns help database designers and programmers structure their code, transactions and data, so that they can avoid database deadlocks [14, 11, 20][27, pages 249–252]. For example, Microsoft [1][19], Oracle[2] and DB2 [5, pages 341-352] published guidelines for minimizing database deadlocks in SQL Server, Oracle, and DB2 databases respectively showing that this problem has not been solved in major databases. These guidelines include, among others, accessing database objects in some order that prevents forming cyclic dependencies (with a drawback of losing parallelism), and keeping transactions short and in one batch. Since these solutions are manual and error-prone, it is important to prevent database deadlocks automatically.

Interestingly, even if the cause of a database deadlock is understood, it is often not possible to fix it, since it would involve drastic redesign by changing the logic of the DCA to avoid certain interleavings of SQL statements among different transactions [17]. In addition, fixing database deadlocks may introduce new concurrency problems, and frequently these fixes reduce the occurrences of database deadlocks instead of eliminating them [21]. Developers need approaches for preventing database deadlocks in order to achieve better performance of software, but unfortunately, there are no tools that prevent database deadlocks.

We created a novel approach that detects all hold-and-wait cycles among resources in SQL statements and prevents database deadlocks automatically using the information about these cycles. This paper makes the following contributions.

[1] `http://msdn.microsoft.com/en-us/library/ms191242(v=sql.105).aspx`

[2] `http://docs.oracle.com/javadb/10.8.2.2/devguide/cdevconcepts53677.html`

Step	Transaction T_1	Transaction T_2
1	UPDATE authors SET citations=100 WHERE paperid=1	
2		UPDATE titles SET copyright=1 WHERE titleid=2
3	SELECT title, doi FROM titles WHERE titleid=2	
4		SELECT authorname FROM authors WHERE paperid=1

Table 1: Example of a database deadlock that may occur when two transactions T_1 and T_2 are issued by DCA(s).

- We introduce our abstraction and a performance model that hide the complexity of database engines, instead concentrating on abstract operations (i.e., read and write) on resources (e.g., database tables) and on how these operations lock and release these resources. Using our abstraction, we developed a Petri net model for representing transactions and we designed an algorithm to detect all hold-and-wait cycles in this model. The algorithm returns exact execution scenarios that lead to database deadlocks, thus enabling programmers to understand and analyze these scenarios.

- We implemented our algorithm and evaluated it with a random SQL generator [30, 24, 1], and we showed that for an extreme case of 100 transactions, each containing 50 SQL statements (i.e., a total of 5,000 SQL statements), it takes a little over 6.5 hours for our algorithm to detect all hold-and-wait cycles. For a realistic case of a large-scale DCA that contains 50 transactions, each of which includes a dozen of SQL statements, all cycles are found in less than two seconds.

- Using this information about hold-and-wait cycles, we designed and built a supervisory control program that prevents database deadlocks by intercepting transactions sent by DCAs to databases, detecting a potential deadlock, and delaying a conflicting transaction thereby breaking a deadlock cycle.

- We implemented our approach in a tool and experimented using three client/server DCAs. Our tool prevented all existing database deadlocks in these DCAs and increased their throughputs by approximately up to three orders of magnitude for 1,000 clients. Our tool is publically available at `http://www.cs.uic.edu/~drmark/REDACT.htm`.

2. THE PROBLEM

In this section we show how DCAs use databases, give an illustrative example of a database deadlock, and formulate the problem statement.

2.1 An Illustrative Example

Consider the example of database deadlock shown in Table 1. Transactions T_1 and T_2 are independently sent by DCAs to the same database at the same time. When the first DCA executes

Figure 1: A lock graph for the transactions shown in Table 1. The lock graph shows the hold-and-wait cycle T_1 →`authors`→ T_2 →`titles`→ T_1.

the `UPDATE` statement in Step 1, the database locks rows of table `authors` in which the value of attribute `paperid` is 1. Next, the second DCA executes the `UPDATE` statement in Step 2 and the database locks rows of table `titles` in which attribute `titleid` is 2. When the `SELECT` statement in Step 3 is executed as part of transaction T_1, the database attempts to obtain a read lock on the rows of table `titles`, which are exclusively locked by transaction T_2 of the second DCA.

Since these locks cannot be imposed simultaneously on the same resource (i.e., these locks are not compatible), T_1 is put on hold. Finally, the `SELECT` statement in Step 4 is executed as part of transaction T_2; the database attempts to obtain a read lock on the rows of table `authors`, which are exclusively locked by transaction T_1 of the first DCA. At this point both T_1 and T_2 are put on hold resulting in a database deadlock. Once an algorithm within the database engine detects this hold-and-wait cycle, the database engine resolves this database deadlock by aborting either the transaction T_1 or the transaction T_2.

Figure 1 shows the lock graph for the transactions appearing in Table 1. Transactions are depicted as rectangles and resources (i.e., tables) are shown as ovals. Arrows directed towards resources designate locks held by transactions on those resources; arrows in the opposite direction designate transactions that are waiting to obtain resource locks. The lock graph shows the hold-and-wait cycle T_1 →`authors`→ T_2 →`titles`→ T_1. The same reasoning applies if the granularity of locks is coarser, for example, at the table level, – when interleaving steps occur as shown in Table 1, a database deadlock is highly likely.

2.2 How DCAs Use Databases

Many enterprise-level DCAs are written in general-purpose programming languages (e.g., Java); they communicate with relational databases by using standardized *application programming interfaces (APIs)*, such as *Java DataBase Connectivity (JDBC)*. Using JDBC, programs pass SQL statements as string parameters in API calls that send these SQL statements to databases for execution. For example, the API call `executeQuery` of the class `Statement` takes a string containing an SQL statement that is sent to a database for execution. Once executed, values of database attributes that are specified in SQL statements are returned to DCAs using JDBC's `ResultSet` interface. These SQL statements are executed as part of a transaction that is delimited by statements "`begin transaction`" (by setting the connection's autocommit mode to `false`) and "`end transaction`" with the subsequent API call `commit`. In case a transaction is not explicitly delimited in the source code, each SQL statement is taken to be a separate transaction, which may be committed automatically.

2.3 The Performance Model

A performance model for analyzing the impact of database deadlocks is shown in Figure 2. This model uses a standard template for discrete time analysis in the performance evaluations of different systems [9]. Using this model we pursue two goals: to understand in which situation database deadlocks present a big performance

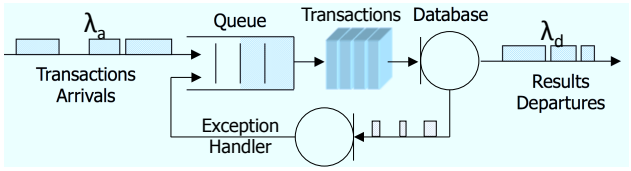

Figure 2: Performance model for database deadlocks.

problem rather than small inconvenience and to determine when our proposed solution will be most beneficial.

Transactions arrive from DCAs at the arrival rate λ_a; arrivals can be modeled as a normal distribution with some mean arrival time rate. These transactions are put into the Queue that models a mechanism for analyzing arriving transactions for cycles in SQL statements. Once this analysis is performed, a batch of transactions is sent to the Database that executes these transactions and outputs results at some departure rate λ_d. If database deadlocks occur, some transactions are aborted and an exception handling mechanism delivers exceptions back to the DCA which retries these aborted transactions. This process is represented using the feedback loop that delivers some transactions back into the Queue at a rate that is proportional to the arrival rate λ_a. That is, we assume that the frequency of database deadlocks is proportional to the transaction arrival rate, which we observed in different projects. We exploited this observation in our recent work where we developed an effective method to reproduce database deadlocks [13].

A relation between two independent variables, λ_a and λ_d is important to determine the impact of database deadlocks. Consider two cases when $\frac{\lambda_d}{\lambda_a} >> 1$ and $\frac{\lambda_d}{\lambda_a} \geq 1$. The underlying physical event for the departure rate λ_d is the time it takes by the database to process transactions and to produce results. Thus, the first case $\frac{\lambda_d}{\lambda_a} >> 1$, where the average time per transaction is measured in milliseconds or seconds rather then minutes or hours, means that transactions are processed by the database much faster than they arrive – this is typical for smaller applications where transactions manipulate small amounts of data without applying complex operations like joins and aggregations. However, existing database deadlock detection algorithms take time, often many seconds to detect cyclic dependencies among executing transactions, leading to a significant overhead. We summarily add this overhead to the Exception Handler processing element in the feedback loop. Our simulation with the performance model showed nonlinear decrease in the throughput (measured as the number of successfully processed transaction in some time interval) for short-running transactions with the high-rate of arrival, meaning that the system loses its scalability when deadlock detection time is equal to or greater than the mean transaction completion time. When a database deadlock occurs, recovering from it is easy and little overhead is involved. Clearly, this is a case for non-mission-critical applications where the impact of database deadlocks is small.

The other case, $\frac{\lambda_d}{\lambda_a} \geq 1$, where the average time per transaction is measured in hours or days, involves long-running and complex transactions for mission-critical and scientific applications. Examples include batch financial and retail applications, various biological sequence analyses, complex process simulations, and online transaction processing tasks that involve data mining big data sets whose sizes are measured in terabytes. In this case, transactions arrive at about the same rate with which they are processed. Any database deadlock resulting in aborting a long-running transaction that is put back into the queue will have a devastating effect on the performance of the system. Our simulation with the performance

model showed a nonlinear decrease in the throughput time, meaning that the system loses its scalability under realistic conditions.

2.4 The Problem Statement

Our main goal is to prevent database deadlocks. Our solution is conservative, i.e., it should statically detect all hold-and-wait cycles that can occur in SQL statements that belong to different transactions. These hold-and-wait cycles may potentially result in database deadlocks. However, depending on interleavings in different execution scenarios, not all of these cycles will lead to database deadlocks, meaning that *false positives (FPs)* are possible. For example, when tables contain little or no data, locks may be released by the database engine almost instantaneously or not imposed at all. Alternatively, a deadlock will not occur if conflicting transactions attempt to lock disjoint sets of rows in at least one of the tables involved. Evidently, the rows being locked depend on the outcome of `WHERE` clauses appearing in queries. Thus, database deadlocks may or may not happen depending on the data stored in database tables, even if hold-and-wait cycles are present, leading to FPs.

On the one hand, a problem with FP hold-and-wait cycles is that they are not known to be FPs until transactions are executed by the database engine. Using our mechanism to prevent a database deadlock by delaying a conflicting transaction imposes an overhead in addition to some reduction in the level of parallelism. On the other hand, given a high cost of database deadlocks, it is an experimental question to determine how FPs affect our solution, and we address this question with our experiments in Section 5.

In addition, our approach should not depend on a specific database engine or require modifications of the kernels of database engines. Similar to operating systems, database engines are very complex, fragile, and closed software systems; a solution that requires their modifications is unlikely to be practical. Moreover, our approach should not depend on specific architectures of DCAs.

Finally, it is important to give exact scenarios that lead to database deadlocks. Lock graphs, an example of which is shown in Figure 1 demonstrate the resulting hold-and-wait cycle, but not execution paths that lead to it. To help programmers understand how database deadlocks occur, we provide exact scenarios that show specific orders of execution of SQL statements and sequences of lock acquisitions that lead to database deadlocks.

3. OUR APPROACH

In this section we describe our key ideas and the abstraction on which they are based, and give an overview of our approach for *pReventing databasE Deadlocks from AppliCation-based Transactions (REDACT)*.

3.1 Our Abstraction

REDACT is based on our abstraction that represents relational databases as sets of resources (e.g., database tables) and transactions that DCAs issue to databases as sets of abstract operations, i.e., reading from and writing into resources, and these abstract operations also issue synchronization requests. Using this abstraction unifies DCAs that use the same databases in a novel way: their independently issued transactions become abstract operations with resource sharing requests. With this abstraction, we hide the complex machinery of database engines and concentrate on abstract operations performed by SQL statements and the engines' locking properties that we associate with these abstract operations.

3.2 Key Ideas

REDACT rests on two key ideas: 1) to detect statically all hold-and-wait cycles among abstract operations that belong to different transactions with specific execution scenarios that lead to these hold-and-wait cycles, and 2) to use the information about all detected hold-and-wait cycles at runtime to prevent database deadlocks by holding back one operation (i.e., SQL statement) that participates in the hold-and-wait cycle hence breaking it. An important element of these ideas is that we separate detection of hold-and-wait cycles and prevention of database deadlocks: the former is done statically and the latter is done dynamically during executions of DCAs that use shared databases. This separation enables us to avoid expensive computations at runtime making database deadlock prevention very fast and efficient, since a simple lookup function is involved.

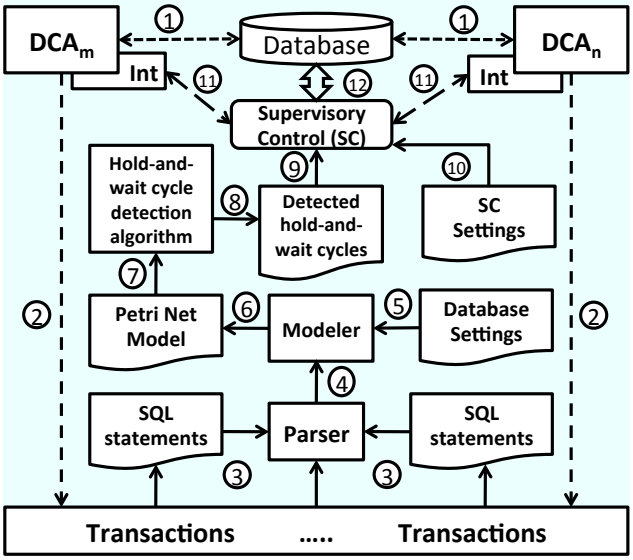

Figure 3: The workflow of REDACT.

In REDACT, we make conservative approximations about how database engines process transactions. In general, processing transactions includes compiling SQL statements, building and optimizing execution plans, and detecting shared accesses by analyzing execution plans with respect to the data that is stored in the database. Different database engines implement this process differently. Our idea is to utilize information that all database engine vendors release on how these database engines impose and release locks on different resources for different types of SQL statements – it is called a database *locking strategy* [31, page 184]. Using a locking strategy, we can approximate the complex sequences of steps of transaction processing by modeling conservatively SQL statements using our abstract operations that issue synchronization requests according to a given locking strategy.

3.3 Overview of REDACT

The workflow of REDACT is shown in Figure 3, and this workflow shows two DCAs (i.e., DCA$_m$ and DCA$_n$) that use the shared Database as shown with dashed arrows labeled with (1). In this setting, database deadlocks occur at some rate. Here we use the terms transaction and SQL statement interchangeably.

The first step in the workflow involves extracting transactions that contain SQL statements from these DCAs as shown with dashed arrows labeled with (2). This is a one-time manual effort that may be required (as it was done for three subject DCAs in this paper). Once transactions are extracted, the static analysis phase starts. First, (3) SQL statements that are contained in these transactions are parsed, (4) and the resulting parse trees are inputted into the Modeler that automatically transforms SQL statements into the ab-

stract operations and synchronization requests. Specific details of modeling are described in Section 4.2. In REDACT, we extracted the SQL parser from Apache Derby database. The Modeler (5) uses database settings that include a locking strategy (6) to produce a Petri Net model, which we describe along with the hold-and-wait cycle detection algorithm in Section 4. This model is created using the *Petri net XML modeling language (PNML)* [4] and it (7) serves as the input to the algorithm that (8) detects all hold-and-wait cycles, that are in turn (9) used as inputs to the Supervisory Control (SC). This step concludes the static phase of REDACT.

At this point, we describe the dynamic phase during which DCAs are run and database deadlocks are prevented automatically. Our goal is to divert SQL statements from DCAs to the SC at runtime, so that the SC determines whether executing these SQL statements may result in hold-and-wait cycles and consequently, a database deadlock. Diverting SQL statements is accomplished in REDACT by using the *interceptor pattern* that is implemented using a framework with call-backs associated with particular events [29].

The first step is to add *interceptors* to DCAs. We show these interceptors as partial rectangles with the label `Int` that are superimposed on the rectangles that designate DCAs in Figure 3. The goal is to intercept JDBC API calls that take SQL statements as string parameters. Instead of (1) sending these statements to the database, (11) the interceptors divert the statements to the supervisory controller, whose goal is to quickly look up if hold-and-wait cycles are present in the SQL statements that are currently in the execution queue. In doing so, the SC utilizes the information that is obtained from the static analysis phase as well as (10) its settings that enable stakeholders to fine-tune the SC for specific environments. For example, it is possible to specify a delay time for a conflicting transaction, which we use in our experiments in Section 5. Smaller delay time increases the probability of database deadlocks, while larger delay time enables REDACT to prevent database deadlocks at the cost of introducing higher overhead in case of FPs. It is an experimental question to determine this trade-off.

To make the SC efficient, each SQL statement is given a unique hash key, and the information about hold-and-wait cycles in SQL statements is coded using hash keys to avoid significant overhead when looking up SQL statement in the execution queue. If no hold-and-wait cycles are present, (12) the SC forwards these SQL statements to the database for execution, otherwise, it holds back one SQL statement (or a transaction to which this SQL statement belongs) while allowing others to proceed, and once these SQL statements are executed and results are sent to the DCAs, the held back SQL statement is sent to the Database, thus effectively preventing the database deadlock. This concludes the description of the workflow for REDACT.

4. THE MODEL AND THE ALGORITHM

In this section we give the background on Petri nets, specify our modeling approach, and discuss the algorithm for detecting hold-and-wait cycles and we argue for its correctness.

4.1 Background on Petri Nets

Ordinary Petri nets are directed graphs with two kinds of nodes called *places* and *transitions* [34]. Figure 4 shows an example of a Petri net modeling the transactions appearing in Table 1. Net transitions are represented as bars; places are represented as circles. Places may contain *tokens* represented as solid dots—each place may be assigned zero or more tokens. For example, place `p1_init` in Figure 4 is assigned one token. Arcs connect transitions to places and vice versa. A directed arc from a place to a transition represents preconditions that are required for an event

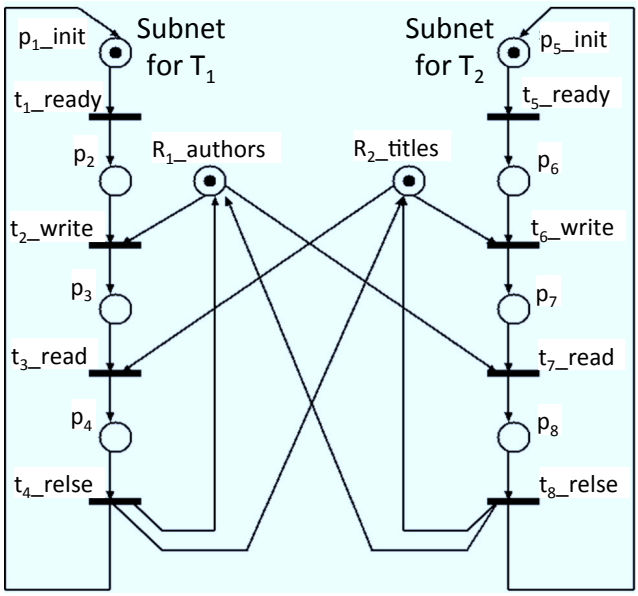

Figure 4: Petri net model for the illustrative example that is shown in Table 1.

associated with that transition to occur. Conversely, arcs from transitions to places represent the outcome of the event associated with the transition. A transition is enabled if all its input places contain at least one token each. An enabled transition may *fire* – a token is consumed from each of the transition's input places and a token is added to all of the transition's output places. For example, when transition `t1_ready` fires, the token in place `p1_init` is removed and a token is added to place `p2`, enabling transition `t2_write`, since both its input places `p2` and `R1_authors` now hold tokens.

DEFINITION 1. *An ordinary Petri net $\mathcal{N} = (P,T,A,M_0)$ is a directed, bipartite graph with node sets $P = \{p_1,\ldots,p_n\}$ (the places) and $T = \{t_1,\ldots,t_m\}$ (the transitions). $A \subseteq (P \times T) \cup (T \times P)$ is the arc set, and for each $p \in P$, $M_0(p)$ defines the initial assignment of tokens to place p. The set of input transitions of a place p is denoted by ${}^\bullet p = \{t|(t,p) \in A\}$. Similarly, the set of output transitions of place p is denoted as $p^\bullet = \{t|(p,t) \in A\}$. The sets of input and output places for a transition t are similarly defined as ${}^\bullet t$ and $t^\bullet$.*

4.2 Modeling Transactions

We model database transactions using a subclass of Petri nets called S^4R nets [32], which consist of a set of disjoint *process subnets*, each modeling a sequential process in a concurrent system. Subnets are connected to each other by a place subset, called the *resource places*, which model resources shared by the process subnets. Typically, operations in different process subnets may require one or more resources shared with other process subnets. In addition, each process subnet consists of one main loop, which starts at an initial place for that subnet; however, no additional cycles are contained in each subnet. We use process subnets to model database transactions and resource places to model database locks, such as locks on database tables or rows. The special structure of S^4R nets has allowed us to build an efficient algorithm for detecting potential deadlocks in DCAs.

The S^4R net is shown in Figure 4 whose two subnets model the transactions that appear in Table 1. Resource places `R1_authors` and `R2_titles` model the locks on the two tables appearing in the example. Our models contain four types of transitions: `ready`, `read`, `write`, and `release`. These transitions are associated

with operations that we introduced as part of our abstraction in Section 3.1. SQL statements `INSERT` and `UPDATE` are modeled using the operation `write`; `SELECT` statements are modeled using the operation `read`. The operation `release` designated in Figure 4 as `relse` specifies that all acquired locks are ready to be released. States and resources are modeled as places. Resource places always have tokens at the initial state to indicate their availability for transactions.

If a transition models an SQL statement that accesses and manipulates some resources, then arcs connect places that designate these resources with that transition. Doing so addresses two issues at the same time: executing the abstract operation that a transition specifies and obtaining a lock on a resource by moving the token from the resource into the transition's output places. For example, when the transition `t1_ready` fires in the model that is shown in Figure 4, a token is placed into place `p2`. Since the token is still in the resource place `R1_authors`, transition `t2_write` is enabled, which corresponds to the execution of the SQL statement `UPDATE` for T_1 in Table 1. Since the token is taken from place `R1_authors`, transition `t7_read` is no longer enabled. The hold-and-wait cycle can be reached by the following transition firing sequence: `t1_ready` → `t5_ready` → `t6_write` → `t2_write`. Unlike lock graphs, these sequences enable stakeholders to understand, analyze, and debug database deadlocks, which is one of the goals of this paper.

4.3 The REDACT Algorithm

A brute-force approach for finding hold-and-wait cycles does not work – for N transactions, each combination should be explored, i.e., the powerset of these transactions, $2^N - N - 1$. Assuming that it takes less than 0.1 sec to explore each combination, it will take one year to explore all combinations for 30 transactions. Of course, they can be explored in parallel – running analysis in 1,000 virtual machines reduces the analysis time to less than 30 hours. However, better results are possible.

Our insight is to use a *depth-first search (DFS)*-based cycle search algorithm, which we extend for S^4R nets. The algorithm iterates through transactions (i.e., subnets) and analyzes if any transition within a subnet is connected to a resource. If no resource is shared, then the analysis complexity is effectively the order of $O(S \times T)$, where S is the number of subnets and T is the number of transitions, i.e., no cycles are possible. In the worst case, all subsets of subnets have cycles, and the complexity is exponential. However, our main insight is that this situation rarely occurs, if ever – there are very few cycles, and this is what makes detection and prevention of database deadlocks difficult. In fact, the main problem is to detect these very few cycles that materialize as database deadlocks. In case there are few and far cycles between transactions, and the analysis time will be quite small even for a large number of resources, as we show in our experiments in Section 5.

The algorithm `Redact` is shown in Algorithm 1. This algorithm takes as its input the S^4R net $\mathcal{N}$ whose process subnets model SQL statements in different transactions. Hold-and-wait cycles are computed and returned in Line `21` of the algorithm. The algorithm recursively calls the procedure `ComputeAllCycles` (see Line `22`) that performs a depth-first search on the subnets of $\mathcal{N}$.

Lines 2 and 3 in Algorithm 1 initialize three variables: (1) a stack of transitions to be searched, (2) a variable, *cycles*, holding all detected hold-and-wait cycles, and (3) a variable, *cycle*, holding transition sequences potentially leading to a cycle. Line 4 extracts the process subnets from input net $\mathcal{N}$. Lines 5–18 iterate the following actions on each subnet. First, all transitions in the subnet are pushed on the stack (Line 6). Next, a transition t is popped from the

Algorithm 1 The REDACT algorithm.

```
1:  Redact( Petri net 𝒩 )
2:  cycles ← ∅{Initialize global variable.}
3:  stack ← ∅, cycle ← {∅}{Initialize local variables.}
4:  GetSubnets(𝒩) ↦ {S} {A subnet of a Petri net models some transaction.}
5:  for all s ∈ {S} do
6:      stack.push( GetTransitions(s) )
7:      while stack ≠ ∅ do
8:          stack.pop() ↦ t ∈ {T}
9:          if ∀p ∈ {P}, Resource(p)=true, ∃a ∈ {A}|a = (p,t) then
10:             cycle ↦ {t}
11:             for all w ∈ {T}|∃(p,w) ∈ {A} do
12:                 ComputeAllCycles(w,cycle,GetSubnet(w))
13:             end for
14:         end if
15:     end while
16:     𝒩 = 𝒩 − s
17: end for
18: return cycles
19: ComputeAllCycles( Transition t, Cycle c, Subnet s )
20: localstack ← ∅, 𝒱 ← {∅}{Init local stack and list of visited transitions.}
21: if ∃p ∈ {P}, Resource(p)=true,|(p,t) ∈ {A} then
22:     cycle ↦ cycle ∪t
23:     if •t ∩GetInitialPlaces(s) = ∅ then
24:         localstack.push( TransitionsPreceding(t) in s )
25:     end if
26:     while localstack ≠ ∅ do
27:         localstack.pop() ↦ u ∈ {T}
28:         if u ∈ GetTransitions(s) then
29:             GetFirstTransition(c)↦ v
30:             if v = u ∨ v ∈ TransitionsPreceding(u) in s then
31:                 cycles ↦ cycles ∪ { cycle }
32:             end if
33:         else if u ∉ 𝒱 then
34:             𝒱 ↦ 𝒱 ∪ u
35:             cycle ↦ cycle ∪ u
36:             if ∀q ∈ •u HoldsToken(q)=true then
37:                 for all (∃q ∈ •u|Resource(q)) ∧ (∃(q,e) ∈ {A}|e ≠ u do
38:                     ComputeAllCycles(e,cycle,s)
39:                 end for
40:             end if
41:             if •u ∉GetInitialPlaces({s}) then
42:                 localstack.push( TransitionsPreceding(u) in s )
43:             end if
44:         end if
45:     end while
46: end if
```

stack and checked for a structural conflict with transitions in other subnets. A structural conflict between net transitions occurs when the transitions share an input place with each other. In this case, t could be in conflict with another transition w, if t and w share a resource place as an input. This means that t and w model computations requiring the same database lock. If there is an arc between t and some resource, t is added to the potential cycle being explored, and procedure `ComputeAllCycles` is called to further explore the cycle. Finally, the process subnet that is considered in each `while` loop iteration is removed from further consideration in Line 16.

Lines 19–46 in Algorithm 1 specify the body of the procedure `ComputeAllCycles`. The procedure takes as input (1) a transition, t, (2) the cycle, c, under exploration, and (3) the process subnet, s, to which t belongs. Line 20 initializes a local stack of transitions and a list of transitions, $\mathcal{V}$, that have been visited. Line 21 checks whether t requires any locks. If this is not the case, the procedure just returns; otherwise, the procedure explores t. In this case, t is added to the cycle under construction and t's predecessor transitions in the process subnet of t are pushed on the local stack for further exploration (Lines 22–25).

Lines 26–32 iteratively pop a transition u from the local stack and check whether the first transition in the cycle under exploration is either u or a predecessor of u in u's subnet. In this case, a cycle

App	LOC	DB Size	T	S	T_{DB}	T_{trans}	Rows
HIM	3,421	248MB	4	2	10	6	1,330,107
UCOM	2,127	29MB	2	2	8	2	250,532
DAN	6,034	371MB	2	2	13	2	1,270,897

Table 2: Subject DCAs and their databases. The columns show lines of code (LOC) in DCAs, the size of DB in megabytes, the number of transactions, T in the DCA and how many SQL statements, S at most are contained in each transaction, the number of tables in the database, T_{DB}, the number of tables used in transactions, T_{trans}, and the total number of rows.

is detected and the cycle under consideration is added to the list of discovered hold-and-wait cycles. Otherwise, in Lines 34–35 u is added to the list, $\mathcal{V}$, of visited transitions, and to the cycle currently being explored. If u is enabled (Line 36), a new search is started from u by invoking `ComputeAllCycles` recursively on all transitions that may share resources with u (Lines 37–39). If u is not enabled, Lines 41–43 push u predecessor transitions in u's process subnet on the local stack and the loop beginning at Line 26 is repeated. The list of cycles is returned in Line 18.

5. EXPERIMENTAL EVALUATION

In this section we describe the results of experimental evaluation of REDACT on three small Java DCAs. We seek to answer the following research questions.

RQ1: How efficiently does the REDACT algorithm detect hold-and-wait cycles in large-scale transactions?

RQ2: How effectively does the REDACT approach prevent database deadlocks?

The rationale for RQ1 is to determine if our REDACT algorithm is practical for detecting hold-and-wait cycles in transactions. We observe that many enterprise large-scale DCAs contain less than 20 transactions, each of which includes a dozen SQL statements. However, we want to experiment in the extreme to show if our REDACT algorithm can handle very large inputs. The rationale for RQ2 is determine how much performance can be gained by preventing database deadlocks when compared with the standard defensive programming practice that we described in Section 1. To address RQ2 is to experiment with DCAs to determine REDACT's overhead and compare it with other solutions.

5.1 Methodology

To evaluate RQ1, we should experiment with different number of transactions that contain different numbers of SQL statements that use different database resources (e.g., tables). This methodology requires a large number of different complex SQL statements that contain hold-and-wait cycles. One way to address the problem is to select transactions from commercial DCAs as benchmarks; however, doing so negatively affects reproducibility of results, which is a cornerstone of the scientific method, since commercial benchmarks cannot be easily shared among companies for legal reasons and trade-secret protection. Unfortunately, in many cases, existing TPC benchmarks fall short of evaluating complex database features, and they do not lead to database deadlocks [36].

Relational database engines are routinely tested using complex SQL statements that are generated using random SQL statement generators [30, 24, 1]. Suppose that a claim is made that a relational database engine performs better at certain aspects of SQL optimization than some other engine. The best way to evaluate this claim is to create complex SQL statements as benchmarks for this evaluation in a way that these statements stress properties that are specific to these aspects of SQL optimization. Since the meaning of SQL statements does not matter for our evaluation, this generator creates semantically meaningless but syntactically correct SQL statements thereby enabling users to automatically create low-cost benchmarks with reduced bias. We use this approach to generate random transactions and seed them with hold-and-wait cycles, and we use these generated transactions to evaluate our REDACT algorithm to address RQ1.

To address RQ2, our goal is to determine how using REDACT with supervisory control (SC) affects the performance of DCAs. Recall that even if a hold-and-wait cycle is detected, it does not necessarily always lead to a database deadlock. Unfortunately, it is not feasible to know in advance if a hold-and-wait cycle would materialize in a deadlock due to a large combinatorial space of possible interleavings. Thus, the SC will conservatively delay an SQL statement (i.e., a transaction to which this SQL statement belongs) to break a hold-and-wait cycle. To evaluate the impact of REDACT and its SC, we should experiment under different conditions.

We aligned our methodology with the guidelines for statistical tests to assess randomized approaches in software engineering [3]. Since database deadlocks are not easy to reproduce, different runs of the DCA may reveal different number of deadlocks and different impacts of REDACT. Our goal is to collect highly representative samples of runs when applying different approaches, perform statistical tests on these samples, and draw conclusions from these tests. Since our experiments involve the probability of encountering database deadlocks, it is important to conduct the experiments multiple times to pick the average to avoid skewed results.

5.2 Subject DCAs

We evaluate REDACT with three Java DCAs whose characteristics are shown in Table 2. `HIM` is a program for maintaining health information records. `DAN` is a demographic analysis program. Finally, `UCOM` is a program for obtaining statistics on how users interact with Unix systems using their commands. These DCAs are based on simplified specifications from real-world applications that came from different projects at Accenture. Subjects DCAs as well as their databases are available from Sourceforge[3]. Each DCA consists of the server component that spawns multiple threads that use its backend database, and a client component that submits client requests and obtains data from the server.

5.3 Experiments With Subject DCAs

In our experiments, we used JMeter `http://jmeter.apache.org` to run subject DCAs with different numbers of clients. In the baseline experiment (type B), database deadlock exception handling is disabled in the subject DCAs, that is, once a database rolls back a transaction, its data is lost. It is the fastest but also incorrect execution that gives us a baseline for performance – the maximum throughput of the DCAs that is measured in the number of executed transactions in a predefined time interval. In the experiment where database deadlock exceptions are handled gracefully (type G) using the defensive programming practice as we described in Section 1, rolled back transactions are retried until successfully executed. Finally, type R experiment uses the REDACT approach that incurs the SC overhead, but it decreases the cost of deadlock resolution.

For each DCA, we carried out experiments with one, 10, 100, and 1,000 clients for 15 mins per experiment and we measured the throughput as the total number of executed transactions. Since exhibiting database deadlocks requires specific interleavings of trans-

[3]http://sourceforge.net/projects/redactapps

Transactions, X	*SQL Stmts per X*	*%X in Cycles*	*Places*	*Transitions*	*Arcs*	*Resources*	$Cycles_{det}$	*Time (in Seconds)*
20	50	10	1960	1000	4244	960	184	1.15
		15	1958	1000	4260	958	846	1.2
		20	1952	1000	4282	952	6477	1.81
	100	10	3997	2020	8690	1977	445	3.36
		15	3974	2020	8730	1954	17760	4.19
		20	3955	2020	8756	1935	650667	12.34
	200	10	7915	4000	17296	3915	5035	12.18
		15	7922	4000	17380	3922	131099	15.07
		20	7922	4000	17394	3922	5022808	89.09
50	10	4	1001	550	2154	451	3	0.59
		7	1005	551	2141	454	3	0.69
		9	1000	550	2176	450	18	0.59
	25	4	2495	1300	5394	1195	87	1.58
		7	2497	1300	5386	1197	106	1.61
		9	2493	1300	5410	1193	764	1.67
	50	4	4890	2500	10600	2390	317	4.63
		7	4883	2500	10588	2383	5873	4.95
		9	4886	2500	10624	2386	17002	5.48
100	10	4	1999	1100	4308	899	10	1.21
		7	2003	1103	4336	900	99	1.27
		9	2001	1101	4348	900	192	1.59
	25	4	4995	2600	10856	2395	660	4.96
		7	4993	2600	10872	2393	57636	7.63
		9	4998	2600	10866	2388	669185	22.91
	50	4	9789	5000	21260	4789	13996	19.12
		7	9775	5000	21328	4775	8449320	184.93
		9	9763	5000	21414	4763	947718814	23793.94

Table 3: Results of experiments on Cycle Detection Algorithm for various number of transactions(X) and SQL statements per transactions. Column *%X in Cycles* represents the percentage of transactions that are randomly chosen to be involved in cycles.The constructed Petri net model is reported as the columns Places, Transitions, Arcs and Resources. Total number of cycles deteted is reported in $Cycles_{det}$ column.The execution time of the algorithm is reported in the Time column. All times are in seconds and rounded to two decimal points.

actions, we repeated each experiment 10 times. Thus, the total number of experiments is equal to three DCAs $\times$ three types (B,G,R) $\times$ four client settings x 10 times = 360 experiments. We report statistical results (average, median, min, max, variance) for 10 runs for the number of observed deadlocks and the throughputs.

5.4 Threats to Validity

A threat to the validity of this experimental evaluation is that our subject programs are relatively small; it is difficult to find large open-source DCAs that use nontrivial databases. Large DCAs may have millions of lines of code and use databases whose sizes are measured in thousands of tables and attributes. Those DCAs and databases may have different characteristics compared to our smaller subject programs. On the one hand, increasing the size of applications to millions of lines of code is unlikely to affect the time and space demands of our analyses because REDACT only considers transactions. Thus, the source code of DCAs is ignored in the cycle analysis, which is focused on the transactions that these DCAs issue to their databases.

On the other hand, increasing the size and the number of transactions may have a significant impact on the cost of cycle analysis. The algorithm that we propose in this paper has the exponential complexity, and even though it is unlikely to encounter DCAs that have hundreds of distinct transactions that contain hundreds of SQL statements each, most of which share resources in cyclic dependencies (it is really a pathological case, since one should question the design of such a system!). However, it is a limitation of REDACT and a potential threat to validity when dealing with ultra large-scale transactions. In addition, it may be more challenging to use the SC for executions of large and complex applications to prevent database deadlocks when there are too many FP hold-and-wait cycles. Evaluating this impact is a subject of future work.

Additional threats to validity of this study is that we used graduate students as programmers who created DCAs, and this task should be tackled by professional programmers. However, most of these students have at least one year of professional programming experience, thereby reducing this threat to validity.

Finally, there are over two dozen of different kinds of database deadlocks. In this paper, we experimented only with circular database deadlocks (most frequently occurring based on our observations) that are realized from hold-and-wait cyclic locks on resources by transactions. It is unclear how well REDACT will perform on other kinds of database deadlocks, so this is a threat to external validity of our results.

5.5 Results

The results of experiments with the REDACT algorithm are shown in Table 3. For the number of transactions and SQL statements per transaction smaller than 50, the hold-and-wait cycle detection time is negligent and measured in seconds. For an extreme case of 100 transactions, each of which containing 50 SQL statements (i.e., a total of 5,000 SQL statements), it takes a little over 6.5 hours for our algorithm to detect all hold-and-wait cycles. For a realistic case of a large-scale DCAs contains 20 transactions, each of which includes 50 SQL statements and in which 20% of statements are chosen to be involved in cycles, our algorithm finds all cycles in

DCA			Deadlocks					Throughput, all transactions for 15mins				
Name	*Type*	*Clients*	*Avg*	*Med*	*Min*	*Max*	*Var*	*Avg*	*Med*	*Min*	*Max*	*Var*
HIM	B	1	0	0	0	0	0	58.4	58.5	55	61	3.82
		10	29.9	30.5	22	36	24.1	68.1	69	57	84	73.43
		100	80.6	79.5	64	96	131.16	25.9	26	22	30	8.1
		1000	92.3	97	19	147	2020.23	9.2	9	5	14	6.62
	G	1	0	0	0	0	0	55.2	56	50	58	8.4
		10	66.6	66.5	47	87	177.16	42.9	44.5	32	52	47.66
		100	97.4	80.5	48	251	3530.04	22.9	22.5	17	29	15.66
		1000	93.6	79	32	170	2618.71	9	9	5	14	7.11
	R	1	0	0	0	0	0	34.1	34.5	29	37	6.32
		10	0	0	0	0	0	72.7	74	65	77	14.68
		100	0	0	0	0	0	61.5	62.5	46	74	46.5
		1000	0	0	0	0	0	45.3	47	34	51	24.46
UCOM	B	1	0	0	0	0	0	435.3	440.5	398	476	715.34
		10	162.4	168	142	174	124.49	41.6	39.5	29	53	62.93
		100	1280.3	1268	1225	1413	3334.68	15.4	15	13	18	2.71
		1000	6899.5	6780.5	5188	9596	1290461	9.8	10	8	11	1.07
	G	1	0	0	0	0	0	455.8	456	419	489	540.84
		10	151.8	150	134	184	201.51	28.5	28.5	17	42	54.94
		100	1259.7	1244.5	1158	1400	6496.01	14.7	15	13	16	0.9
		1000	6873.1	6950	5607	7913	401380.8	11.4	11.5	8	15	4.27
	R	1	0	0	0	0	0	44	44	44	44	0
		10	0	0	0	0	0	161.6	160	155	174	35.82
		100	0	0	0	0	0	634.4	636	614	657	173.82
		1000	0	0	0	0	0	4181.2	4175.5	3463	4840	113323.3
DAN	B	1	0	0	0	0	0	115753.8	115614	113019	120336	3376321
		10	147.1	148.5	132	159	68.1	19.3	19.5	16	23	6.23
		100	1190.6	1195	1136	1251	1440.04	19.5	19	16	27	8.5
		1000	10994.3	10873	9246	13080	1573663	13.2	13	9	17	6.84
	G	1	0	0	0	0	0	115638	115660	114098	117210	656311.1
		10	160	162	141	185	153.33	21	21	17	25	6.22
		100	1189.7	1194.5	1134	1235	902.68	18	18	17	19	0.67
		1000	9468.1	9205.5	8597	11776	980265.2	13.4	12.5	10	21	13.16
	R	1	0	0	0	0	0	44	44	44	44	0
		10	0	0	0	0	0	292.1	292	283	299	27.66
		100	0	0	0	0	0	2923.8	2942.5	2825	2959	1779.07
		1000	0	0	0	0	0	27685.4	27740	26668	28924	438356.7

Table 4: Each subject DCA (i.e., HIM, UCOM, and DAN) is evaluated using (B)aseline, (G)raceful exception handling, and (R)EDACT methodologies. Each DCA was run with one, 10, 100, and 1,000 clients for 15 mins per experiment and we measured the throughput as the number of all executed transactions for 15mins of the experiment. For the columns Deadlock and Throughput we report statistical results (average, median, min, max, variance) for 10 runs for each DCA/client setting.

less than two seconds. These results provide an answer to RQ1 that **our REDACT algorithm efficiently detects hold-and-wait cycles in large-scale transactions.**

The results of experiments with the subject DCAs are shown in Table 4 and Table 5. When one client is used, database deadlocks do not occur, and the overhead of supervisory control is rather significant – it reduces the throughput by approx 72% for `HIM`. However, as the number of clients increases, so does the frequency of deadlocks. We can see that the average number of database deadlocks increases by two orders of magnitude for the DCA DAN when the number of clients increases from 10 to 1,000. At the same time, the thoughput drops by four orders of magnitude, since the overhead of database deadlock resolution algorithm within the database engine takes its toll even if discarded transactions are not retried. Recall that the database engine takes some time (it is a configurable parameter usually set between 15-30 seconds, by default is set to 60 seconds[4]) to locate cycles and resolve database deadlocks. In general, the timeout is set by database administrators who base their decision on the average time it takes to execute a transaction. This resolution time significantly worsens the performance of DCAs severely impacting their scalability!

We experimented with different conflicting transaction delay times for the SC that we show in Table 5. We decreased the time from 20 seconds, the value that we used in experiments that we show in Table 4 to 2 seconds and 0.2 seconds. Reducing the delay time force SC to perform unnecessary computation while scheduling process for newly arrived transactions needs to be put on hold. Exceptions are for 100 and 1000 clients of UCOM when decreasing the delay time decreased the throughput. The reason is in non-scalability of the design of the DCA, since lots of transactions were aborted for connection exceptions due to so many clients. However, as we observe from the experimental results in Table 5, throughput increases by more than two orders of magnitude in some cases with no database deadlocks observed.

And this is when REDACT is effective – it prevents database

[4]`http://tinyurl.com/dbdeadlocktimeout`

Application	*Clients*	*Time (seconds)*	*Throughput (Avg)*
HIM	1	0.2	57.4
		2	54
		20	34.1
	10	0.2	93.7
		2	90.6
		20	72.7
	100	0.2	80.6
		2	71.2
		20	61.5
	1000	0.2	70.7
		2	52.4
		20	45.3
UCOM	1	0.2	445.8
		2	261.2
		20	44
	10	0.2	280.2
		2	264.8
		20	161.6
	100	0.2	834.4
		2	885.1
		20	634.4
	1000	0.2	1284.5
		2	4804
		20	4181.2
DAN	1	0.2	4333.3
		2	443.9
		20	44
	10	0.2	55040.1
		2	2952.1
		20	292.1
	100	0.2	101209.9
		2	29554.5
		20	2923.8
	1000	0.2	106245
		2	101194.5
		20	27685.4

Table 5: Results of experiments with REDACT for three subject applications (i.e., HIM, UCOM and DAN), four different client loads, and three different waiting times in the SC. We report the average numbers of throughput values (i.e., executed transactions) of 10 runs for each distinct combination of the experimental settings. Each experiment lasted for 15 minutes, no database deadlocks were observed.

deadlocks thereby removing the need for this costly deadlock resolution within the database engine. For 1,000 clients for `HIM`, the average throughput is 9 transactions for the type G experiment, while REDACT's throughput is 45.3 transactions. For UCOM, the numbers are 11.4 versus 4,181.2 for REDACT, and for DAN the numbers are 13.4 versus 27,685.4 for REDACT on average. And for DAN, the throughput is improved by approximately three orders of magnitude. These results allow us to answer RQ2 that **the REDACT approach is very effective in preventing database deadlocks**.

6. RELATED WORK

Some approaches use static program analysis to obtain information about deadlocks. RacerX is a static tool that uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks [10]. Williams et al. [35] defined a deadlock detection algorithm for Java libraries. In contrast with our method, these approaches derive lock graphs directly from Java and C++ source code, and they suffer from false negatives. These approaches are not currently applicable to detect database deadlocks, since analyzing source code of DCA will not detect cycles in transactions.

Dynamic approaches use runtime data to infer where deadlocks may occur or determine how to predict and resolve them in future program runs. An approach called Dimmunix "immunizes" programs against deadlocks by collecting deadlock patterns, which are subsets of control flow traces that lead to deadlocks [17]. Like REDACT, it uses detected hold-and-wait cycles to prevent database deadlocks.A fundamental difference between REDACT and Dimmunix is that the hold-and-wait cycles designate necessary conditions for deadlocks to occur, while deadlock patterns in Dimmunix are loose approximations that result in many FPs, especially since control-flow of DCA is not applicable to detect database deadlocks.

Rx is a dynamic approach that rolls back an application once a deadlock occurs to a checkpoint and retries it again with the hope that the deadlock will be avoided in subsequent executions [26]. This solution cannot be used in the context of REDACT, since rolling back a DCA significantly worsens its performance – retrying these transactions incurs a significant performance penalty.

Recent work on MagicFuzzer described a dynamic deadlock detection technique for C++ programs, where MagicFuzzer uses runtime information to prune the number of choices that may lead to deadlocks [7]. A dynamic approach Sammati provides automatic deadlock detection and recovery for POSIX threaded applications [25]. Unlike MagicFuzzer and Sanmati, REDACT performs its analysis at compile time to prevent deadlocks at runtime.

Like REDACT, snapshot isolation partially addresses the problem of database deadlocks by avoiding conflicting concurrent updates that may lead to inconsistent snapshots [6]. In contrast to REDACT, exceptions are thrown when snapshot isolation is violated, leading to the same performance problem that REDACT address with database deadlocks. H-Store addresses the database deadlock problem by running transactions single-threaded and avoiding conflicts by preventing multiple transactions from competing with one another [18]. It is a new concept and our goal of future work is to experiment to compare REDACT and H-Store.

Approaches for preventing deadlocks using transactional memory are gaining increasing popularity [33], but unfortunately, database deadlocks often occur in the distributed setting.

7. CONCLUSION

Since applications often share the same databases concurrently, database deadlocks routinely occur resulting in major performance degradation in these applications. To address this problem, we created a novel approach for preventing database deadlocks automatically, and we rigorously evaluated it. For a realistic case of over 1,200 SQL statements, our algorithm detects all hold-and-wait cycles in less than two seconds. We build a tool that implements our approach and we experimented with three DCAs. Our tool prevented all database deadlocks in these DCAs and increased their throughputs by approximately up to three orders of magnitude.

Acknowledgments

This work is supported by NSF CCF-1217928, CCF-1017633, and NSF CCF-0916139. We warmly thank our ESEC/FSE reviewers whose comments helped us improve the quality of this paper.

8. REFERENCES

[1] S. Abdul Khalek and S. Khurshid. Automated SQL query

generation for systematic testing of database engines. In *Proc. IEEE/ACM ASE*, pages 329–332. ACM, Sept. 2010.

[2] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, Nov. 1987.

[3] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. 33rd International Conference on Software Engineering (ICSE 2011)*, pages 1–10, Honolulu, Hawaii, May 2011.

[4] J. Billington, S. Christensen, K. Van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri net markup language: concepts, technology, and tools. ICATPN'03, pages 483–505, Berlin, Heidelberg, 2003. Springer-Verlag.

[5] P. Bruni, P. Becker, J. Henderyckx, J. Link, and B. Steegmans. *DB2 UDB for z/OS: Design Guidelines for High Performance and Availability*. IBM Redbooks, New York, NY, USA, Jan. 2006.

[6] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, Dec. 2009.

[7] Y. Cai and W. K. Chan. Magicfuzzer: scalable deadlock detection for large-scale applications. ICSE 2012, pages 606–616, Piscataway, NJ, USA, 2012. IEEE Press.

[8] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[9] M. A. Dimmler and A. K. Schmig. Using discrete-time analysis in the performance evaluation of manufacturing systems. In *IN SMOMS'99*, 1999.

[10] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[11] H. Garcia-Molina. A concurrency control mechanism for distributed databases which use centralized locking controllers. In *Proceedings of the Fourth Berkeley Workshop on Distributed Databases and Computer Networks*, pages 113–122, Berkeley, CA, USA, Aug. 1979.

[12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[13] M. Grechanik, B. M. Hossain, and U. Buy. Testing database-centric applications for causes of database deadlocks. In *ICST*, pages 1–10, Mar. 2013.

[14] J. Griggs. Database deadlock avoidance patterns. *http://c2.com/cgi/wiki?DatabaseDeadlockAvoidancePatterns*, Sept. 2003.

[15] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Found. Trends databases*, 1(2):141–259, Feb. 2007.

[16] M. Hofri. On timeout for global deadlock detection in decentralized database systems. *Inf. Process. Lett.*, 51:295–302, September 1994.

[17] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: enabling systems to defend against deadlocks. OSDI'08, pages 295–308, Berkeley, CA, USA, 2008. USENIX Association.

[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008.

[19] J. Kehayias and T. Krueger. *Troubleshooting SQL Server: A Guide for the Accidental DBA*. Red gate books, Cambridge, CB4 0WZ, UK, Sept. 2011.

[20] D. B. Lomet. Subsystems of processes with deadlock avoidance. *IEEE Trans. Software Eng.*, 6(3):297–304, 1980.

[21] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[22] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, Dec. 1986.

[23] M. Nonemacher. Deadlocks in j2ee. *Java Dev. Journal*, Apr. 2006.

[24] M. Poess and J. M. Stephens, Jr. Generating thousand benchmark queries in seconds. In *Proc. 13th VLDB*, pages 1045–1053. Morgan Kaufmann, Aug. 2004.

[25] H. K. Pyla and S. Varadarajan. Avoiding deadlock avoidance. PACT '10, pages 75–86, New York, NY, USA, 2010. ACM.

[26] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergiesï£¡a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3), Aug. 2007.

[27] S. K. Rahimi and F. S. Haug. *Distributed Database Management Systems: A Practical Approach*. Wiley-IEEE Computer Society Pr, New York, NY, USA, 1st edition, Aug. 2010.

[28] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.

[29] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[30] D. R. Slutz. Massive stochastic testing of SQL. In *Proc. 24rd VLDB*, pages 618–622. Morgan Kaufmann, Aug. 1998.

[31] D. S. Team. *Java Server Programming J2Ee 1.4 Ed. Black Book*. Wiley Publications, 2007.

[32] F. Tricas, J. Colom, and J. Ezpeleta. A solution to the problem of deadlocks in concurrent systems using Petri nets and integer linear programming. In *Proc. of the 11th European Simulation Symposium*.

[33] H. Volos, A. J. Tack, M. M. Swift, and S. Lu. Applying transactional memory to concurrency bugs. ASPLOS '12, pages 211–222, New York, NY, USA, 2012. ACM.

[34] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. A. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL*, pages 252–263, 2009.

[35] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.

[36] N. Yuhanna, M. Gilpin, and D. D'Silva. Tpc benchmarks don't matter anymore. *Forrester Research*, 2009.