# REDACT: Preventing Database Deadlocks from Application-Based Transactions

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

Haisheng Wang
Oracle Corp.
Redwood City, CA 94065
haisheng.wang@oracle.com

## ABSTRACT

In this demonstration, we will present a database deadlocks prevention system that visualizes our algorithm for detecting hold-and-wait cycles that specify how resources (e.g., database tables) are locked and waited on to be locked during executions of SQL statements and utilizes those cycles information to prevent database deadlocks automatically.

## 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 deadlocks, concurrency

## 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 [7] – and obtaining data that result from execution of these SQL statements. When DCAs use the same database at the same time, concurrency errors are observed frequently, and these errors are known as *database deadlocks*, which is one of the reasons for major performance degradation in these applications [14, 9].

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* [4]. Currently, database deadlocks are typically detected within database engines 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 [10, 9, 7]. 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. Unfortunately, this solution is only partially effective. It causes performance degradation, since DCAs should repeat the rolled back transactions to ensure functional correctness. 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 [7, pages 206-212].

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 [12]. In addition, fixing database deadlocks may introduce new concurrency problems, and frequently these fixes reduce the occurrences of database deadlocks instead of eliminating them [13]. Developers need approaches for preventing database deadlocks in order to achieve better performance of software, however, there are currently no tools that prevent database deadlocks.

We demonstrate an algorithm by a simulator with which users can enter SQL statements and detect all hold-and-wait cycles that potentially lead to database deadlocks. This detection is done statically, and its results are used to prevent database deadlocks at runtime. We evaluated our algorithm using this simulator with a random SQL generator [17, 15, 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 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. A video accompanying this demonstartion is available online.[1]

Using this information about hold-and-wait cycles, we designed and built a generator for a supervisory control program that prevents database deadlocks by intercepting SQL statements sent to databases by DCAs, detecting a potential deadlock, and delaying an SQL statement thus effectively breaking the deadlock cycle.

We implemented the approach 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 [8]. Our tool is publically available at `http://www.cs.uic.edu/~drmark/REDACT.htm`.

[1] http://www.youtube.com/watch?v=XB12nOJm8-s


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

## 2. 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)*

### 2.1 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.

### 2.2 Key Ideas

REDACT rests on two key ideas: 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 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. The 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.

### 2.3 Overview of REDACT

The workflow of REDACT is shown in Figure 1, 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.

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). At first glance, it appears to be tedious and laborious work for programmers to extract SQL statement from the source code of DCAs. In reality, it is a practical and modest exercise that takes little time. We observed in industry that all transactions with SQL statements are available in separate documents for many enterprise applications. The explanation is simple – transactions contain complicated SQL statements that should be debugged and tested by database analysts using specialized SQL development environments before they are used by developers of DCAs.

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 abstract operations and synchronization requests. 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. This model is created using the Petri net XML modeling language (PNML) [2] 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 Generator that (10) outputs a Supervisory Control (SC). This step concludes the static phase of REDACT.

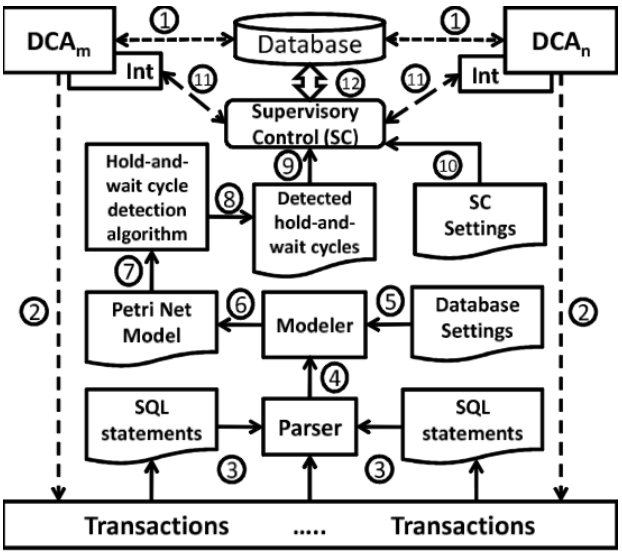


**Figure 1: The workflow of REDACT.**

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 SC at runtime, so that it 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 callbacks associated with particular events [16]. In REDACT, we use AspectJ[2] to instrument subject DCAs with aspects to intercept SQL statements, even though different binary rewriting tools can be used for this purpose.

The first step is to add *interceptors* to DCAs shown as partial rectangles with the label `Int` in Figure 1 that are superimposed on the rectangles that designate DCAs. These interceptors intercept JDBC API calls that take SQL statements are string parameters and instead of (1) sending these statements to the Database, (11) they divert them to SC, 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, SC utilizes the information that is obtained from the static analysis phase. 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) SC forwards these SQL statements to

[2]http://www.eclipse.org/aspectj, verified June 02, 2013.

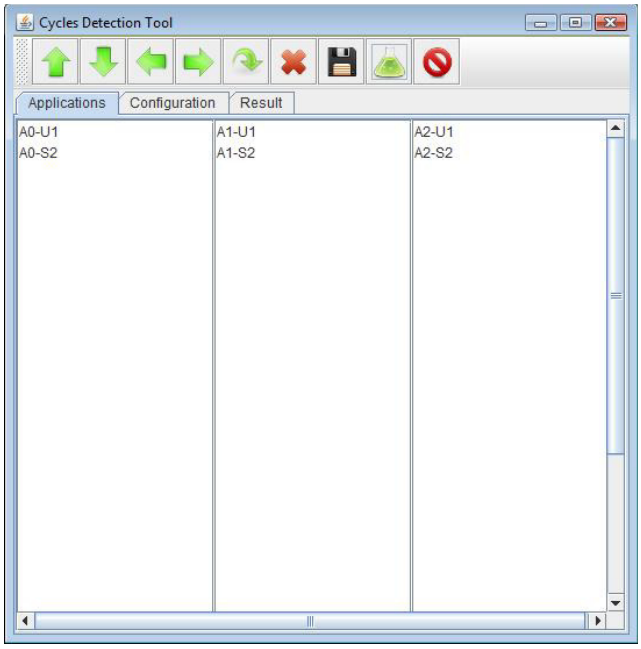


**Figure 2: Lists of statements in applications.**

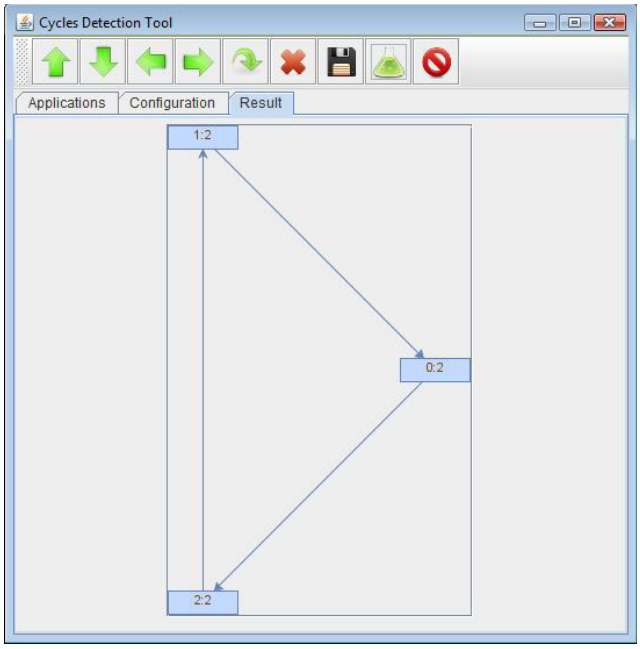


**Figure 3: Visualization of cycles**

the database for execution, otherwise, it holds back one SQL statement 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.

## 3. DEMONSTRATION OF REDACT

We will demonstrate two phases of REDACT. In the static phase, user loads the SQL statements (transactions) of all applications into the simulator by specifying the location of statements in the *configuration* panel. Then, the simulator lists and shows the loaded statements in the *Applications* panel as it is shown in Figure 2. At this point, user can click on an item from the list to see or edit the corresponding SQL statement. User can also move the statements to another application or rearrange the order within the same application by using the appropriate buttons (*left arrow, right arrow, up arrow, down arrow*) located at the upper part of the simulator. An option to move statement to specific application by inserting the identification number of destination application is also available. User can remove a statement by selecting it and clicking on the *delete* button. Finally, user can save the statements (transactions) to a desired location with all of the changes made by clicking on the *save* button.

When editing is done, user can click on the *listen* button that sends the transactions of all applications to the modeler. Modeler builds the model and sends it to the cycle detection algorithm. Cycle detection algorithm analyzes the model and detects all possible cycles among the transactions. Then, it writes the cycles into a file and also sends that back to the simulator. While processing, user can click on the *stop* button from the simulator at anytime to stop further processing.

When the system executes successfully and the result comes back to the simulator, user can visualize the detected cycles by clicking on the *Result* panel of the simulator as it is shown in Figure 3. Visualization of the cycles includes information about the associated transactions that form those cycles.

In the dynamic phase of REDACT, we will run the supervisory control(SC), where we provide the cycles detected in static phase as input. SC looks into those cycles and utilizes that information to prevent database deadlocks as we described in Section 2. SC writes the output into files in a regular time interval that is set as a configuration parameter. It also writes the output to the command prompt that shows the number of total statements executed, number of times each transaction executed, number of deadlock detected (which is zero for our REDACT system) and also number of other exceptions thrown during the execution.

In REDACT, SC determines if a transaction goes for execution or remains blocked in the waiting queue. All transactions, issued by the DCAs, go through a ready queue. SC periodically checks for the transactions in the ready queue. Then, it looks for the possible cycles among those transactions. For every cycle, SC carefully selects a transaction that is to be sent in the blocked state (waiting queue), to break the cycle. SC also maintains the waiting times of all transactions in the waiting queue. If there exist multiple transactions that can be selected as a blocked transaction, SC choose the transaction with the minimum waiting time as the blocked transaction. In this way, SC prevents indefinite waiting time for a transaction.

We will also demonstrate our tool that generates random SQL statements. This random generator is based on stochastic parse trees where language grammar production rules are assigned probabilities that specify the frequencies with which instantiations of these rules will appear in the generated statements. User can use these randomly generated statements of arbitrary length and complexity as input to the simulator.

## 4. RELATED WORK

Language-based approaches offer different type systems and annotation facilities for programmers to annotate programs, so that type checkers can analyze and detect deadlocks [6, 3]. Given that DCAs contain embedded SQL statements, this approach requires a combination of type systems: one of SQL and the other of the host language in which DCA is written. We are not aware of any language-based approach that can be currently applied to solve the problem of database deadlocks.

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 [5]. Williams et al. [19] 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 cannot be currently used to detect and prevent 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 [12]. 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.

LaFortune et al [18] and Iordache et al [11] proposed a Petri net based supervisory control approach to avoid deadlocks in concurrent software written in Java, and in the spirit of using supervisory controls and Petri nets, it is related work to REDACT. They specifically use supervision based on place invariants to create the supervisory controller. In contrast, our work targets database deadlocks and our work avoids computational complexity of their solution.

## 5. CONCLUSION

We will demonstrate our tool 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 applications. Our tool prevented all existing database deadlocks in these applications and increased their throughputs by approximately up to three orders of magnitude.

## 6. 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] 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.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

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

[5] 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.

[6] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. 7th ACM SIGPLAN TLDI '11, pages 15–28, New York, NY, USA, 2011. ACM.

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

[8] M. Grechanik, B. M. M. Hossain, U. Buy, and H. Wang. Preventing database deadlocks in applications. In *To Appear at ESEC/FSE*, 2013.

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

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

[11] M. V. Iordache, J. O. Moody, and P. J. Antsaklis. Synthesis of deadlock prevention supervisors using Petri nets. *IEEE Transactions on Robotics and Automation*, 18(1):59–68, 2002.

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

[19] 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.