Enhancing Parallelism and Scalability of Database-Centric Applications in Presence of Database Deadlocks

BY

B.M. MAINUL HOSSAIN
B.S., University of Dhaka, Bangladesh, 2002
M.S., University of Dhaka, Bangladesh, 2003

THESIS

Submitted as partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Chicago, 2014

Chicago, Illinois

Defense Committee:

Mark Grechanik, Chair and Advisor
Ugo Buy
Jon Solworth
Prasad Sistla
Lenore Zuck
Mohak Shah, GE Global Research
dedicated to my mother
ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Dr. Mark Grechanik for his uncomparable guidance and support over the years. I am grateful to him for all of his motivations and encouragements, above all, for his belief in my abilities. He guided me to determine the direction of my research and made it interesting to me. I am indebted and grateful to him, without whose support this dissertation would not have been completed.

I would also like to thank my other committee members Dr. Ugo Buy, Dr. Jon Solworth, Dr. Prasad Sistla, Dr. Lenore Zuck and Dr. Mohak Shah for their valuable comments and supports regarding my dissertation and research.

I am greatly thankful to my two friends, Tathagata Dasgupta and Gurudev Devanla, for their continuous encouragements and suggestions towards my PhD degree. I also offer my regards to all of those who I have not mentioned, but who have supported me in any respect during any point of my life, that helped me to come at this level and produce this dissertation.

Finally, I would like to thank my wife, Sumaiya Sayed, whose love and inspiration was truly a blessing during the entire period of my PhD. It is hard to find words to express my gratitude to this wonderful lady.

BMH
Portions of chapters 4, 5, 6 and 7 in this work have been published previously in proceedings of the 2012 Workshop on Dynamic Analysis (WODA 2012), proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013) and proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST 2013).
Contribution of Authors

Chapter 1 represents the motivation of my work and describes the problems with research objectives. Chapter 2 represents the context of the problems and the assumptions that are made. Chapter 3 is a literature review that shows the related work and justifies the significance of my research objectives. Chapter 4 provides the details on the tool for generating random SQL statements with seeded hold-and-wait cycles. I was the primary developer of the tool and followed the similar strategy as (Hussain et al., 2012). Chapter 4 includes some portion of the manuscript (Hussain et al., 2012) for which I was a coauthor. My work was significant for the conclusions of this manuscript because I contributed by analyzing various experimental data for comparison and played a role in the writing of the manuscript. Chapter 5, 6 and 7 represents published manuscripts (Hossain et al., 2013), (Grechanik et al., 2013b) and (Grechanik et al., 2013a). I was the first author for (Hossain et al., 2013) and second author for (Grechanik et al., 2013b) and (Grechanik et al., 2013a). I was a key person behind the ideas, formulation of the problems and solutions represented in these manuscripts. I implemented the solution $PD^2$, designed and conducted the experiments, analyzed the experimental results to draw conclusions. My research mentors, Dr. Mark Grechanik and Dr. Ugo Buy contributed by participating in helpful discussions and contributed to the writing. Dr. Haisheng Wang contributed to the implementation of the algorithm for cycles detection. Chapter 8 draws some conclusions and indicates the directions of future research. I anticipate that my research work will lay a foundation for a new direction of research and impact the development of commercial software tools.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 An Illustrative Example</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Some ways for database deadlocks to come into existence</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Problems with Database Deadlocks</td>
<td>6</td>
</tr>
<tr>
<td>1.2.1 Testing for Database Deadlocks</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2 Reproducing Versus Simulating Database Deadlocks</td>
<td>10</td>
</tr>
<tr>
<td>1.2.3 Preventing Database Deadlocks</td>
<td>12</td>
</tr>
<tr>
<td>1.3 Our Objectives and Research Contributions</td>
<td>15</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>17</td>
</tr>
<tr>
<td><strong>2</strong> CONTEXT OF DATABASE DEADLOCKS</td>
<td>18</td>
</tr>
<tr>
<td>2.1 How DCAs Use Databases</td>
<td>18</td>
</tr>
<tr>
<td>2.1.1 Java DataBase Connectivity (JDBC)</td>
<td>19</td>
</tr>
<tr>
<td>2.1.2 Hibernate</td>
<td>21</td>
</tr>
<tr>
<td>2.1.3 Overview of Our Solution</td>
<td>24</td>
</tr>
<tr>
<td>2.1.4 Performance Model</td>
<td>26</td>
</tr>
<tr>
<td>2.2 Our Assumptions</td>
<td>28</td>
</tr>
<tr>
<td>2.2.1 Structure and Extraction of SQL statements</td>
<td>28</td>
</tr>
<tr>
<td>2.2.2 Locking Strategies</td>
<td>32</td>
</tr>
<tr>
<td>2.2.3 False Positive Cycles</td>
<td>33</td>
</tr>
<tr>
<td><strong>3</strong> RELATED WORK</td>
<td>36</td>
</tr>
<tr>
<td>3.1 Testing for Database Deadlocks</td>
<td>36</td>
</tr>
<tr>
<td>3.2 Preventing Database Deadlocks</td>
<td>38</td>
</tr>
<tr>
<td>3.3 Random SQL statements generation</td>
<td>42</td>
</tr>
<tr>
<td>3.4 Random Test Data Generation for Different Domains</td>
<td>43</td>
</tr>
<tr>
<td><strong>4</strong> RANDOM SQL STATEMENTS GENERATION</td>
<td>48</td>
</tr>
<tr>
<td>4.1 Benchmark for SQL Statements</td>
<td>48</td>
</tr>
<tr>
<td>4.2 Our Approach</td>
<td>50</td>
</tr>
<tr>
<td>4.2.1 Stochastic Grammar Model</td>
<td>51</td>
</tr>
<tr>
<td>4.2.2 Benefits</td>
<td>53</td>
</tr>
<tr>
<td>4.3 Implementation</td>
<td>55</td>
</tr>
<tr>
<td>4.3.1 Configuration Options</td>
<td>57</td>
</tr>
<tr>
<td>4.3.2 Example of Generated Statements</td>
<td>58</td>
</tr>
<tr>
<td>4.4 Seeding Hold-and-wait Cycles</td>
<td>59</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 CYCLES DETECTION</td>
<td>62</td>
</tr>
<tr>
<td>5.1 Hold-and-wait Cycle</td>
<td>62</td>
</tr>
<tr>
<td>5.2 Modeling Transactions</td>
<td>63</td>
</tr>
<tr>
<td>5.2.1 Our Abstraction</td>
<td>63</td>
</tr>
<tr>
<td>5.2.2 Background on Petri Nets</td>
<td>63</td>
</tr>
<tr>
<td>5.2.3 Modeling Using Petri Nets</td>
<td>65</td>
</tr>
<tr>
<td>5.2.4 Benefits of Using Petri Nets</td>
<td>66</td>
</tr>
<tr>
<td>5.3 Cycles Detection Algorithm</td>
<td>68</td>
</tr>
<tr>
<td>5.3.1 Completeness of the algorithm</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2 Complexity of the algorithm</td>
<td>75</td>
</tr>
<tr>
<td>5.4 Cycle Detection GUI Tool</td>
<td>76</td>
</tr>
<tr>
<td>6 PREDICTING AND PREVENTING DATABASE DEADLOCKS</td>
<td>79</td>
</tr>
<tr>
<td>6.1 Our Key Ideas</td>
<td>79</td>
</tr>
<tr>
<td>6.1.1 Ideas for STEPDAD</td>
<td>79</td>
</tr>
<tr>
<td>6.1.2 Ideas for REDACT</td>
<td>80</td>
</tr>
<tr>
<td>6.2 Architecture of ((PD)^2)</td>
<td>81</td>
</tr>
<tr>
<td>6.3 Supervisory Control</td>
<td>83</td>
</tr>
<tr>
<td>7 EMPIRICAL DATA AND EVALUATION</td>
<td>85</td>
</tr>
<tr>
<td>7.1 Evaluation of Cycles Detection Algorithm</td>
<td>85</td>
</tr>
<tr>
<td>7.1.1 Comparison with INA algorithm</td>
<td>87</td>
</tr>
<tr>
<td>7.1.2 Analyzing Experimental Results</td>
<td>88</td>
</tr>
<tr>
<td>7.2 Subject DCAs for STEPDAD and REDACT</td>
<td>89</td>
</tr>
<tr>
<td>7.3 Evaluation of STEPDAD</td>
<td>90</td>
</tr>
<tr>
<td>7.3.1 Methodology</td>
<td>91</td>
</tr>
<tr>
<td>7.3.2 Analyzing Experimental Results</td>
<td>94</td>
</tr>
<tr>
<td>7.3.3 Discussion</td>
<td>98</td>
</tr>
<tr>
<td>7.4 Evaluation of REDACT</td>
<td>99</td>
</tr>
<tr>
<td>7.4.1 Methodology</td>
<td>100</td>
</tr>
<tr>
<td>7.4.2 Analyzing Experimental Results</td>
<td>101</td>
</tr>
<tr>
<td>7.4.2.1 Delay in Supervisory Control</td>
<td>103</td>
</tr>
<tr>
<td>7.4.3 Discussion</td>
<td>105</td>
</tr>
<tr>
<td>7.5 Threats to Validity</td>
<td>105</td>
</tr>
<tr>
<td>8 CONCLUSIONS AND FUTURE WORK</td>
<td>107</td>
</tr>
<tr>
<td>8.1 Conclusions</td>
<td>107</td>
</tr>
<tr>
<td>8.2 Future Work</td>
<td>108</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPENDICES</td>
<td>110</td>
</tr>
<tr>
<td>Appendix A</td>
<td>111</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>113</td>
</tr>
<tr>
<td>VITA</td>
<td>122</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>EXAMPLE OF A DATABASE DEADLOCK</td>
</tr>
<tr>
<td>II</td>
<td>EXAMPLE OF A FALSE POSITIVE CYCLE</td>
</tr>
<tr>
<td>III</td>
<td>A DATABASE DEADLOCK INVOLVING TWO TRANSACTIONS</td>
</tr>
<tr>
<td>IV</td>
<td>RESULTS OF EXPERIMENTS ON CYCLES DETECTION ALGORITHM.</td>
</tr>
<tr>
<td>V</td>
<td>EXPERIMENTS WITH INA.</td>
</tr>
<tr>
<td>VI</td>
<td>SUBJECT DCAS AND THEIR DATABASES.</td>
</tr>
<tr>
<td>VII</td>
<td>RESULTS OF EXPERIMENTS ON STEPDAD.</td>
</tr>
<tr>
<td>VIII</td>
<td>RESULTS OF EXPERIMENTS ON REDACT.</td>
</tr>
<tr>
<td>IX</td>
<td>RESULTS OF EXPERIMENTS WITH DIFFERENT WAITING TIME IN SC.</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>56</td>
</tr>
<tr>
<td>10</td>
<td>57</td>
</tr>
<tr>
<td>11</td>
<td>58</td>
</tr>
<tr>
<td>12</td>
<td>61</td>
</tr>
<tr>
<td>13</td>
<td>65</td>
</tr>
<tr>
<td>14</td>
<td>68</td>
</tr>
<tr>
<td>15</td>
<td>69</td>
</tr>
<tr>
<td>16</td>
<td>73</td>
</tr>
<tr>
<td>17</td>
<td>76</td>
</tr>
<tr>
<td>18</td>
<td>82</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>CS</td>
<td>Computer Science</td>
</tr>
<tr>
<td>UIC</td>
<td>University of Illinois at Chicago</td>
</tr>
<tr>
<td>DCA</td>
<td>Database-Centric Application</td>
</tr>
<tr>
<td>ECA</td>
<td>Enterprise Claim Application</td>
</tr>
<tr>
<td>SC</td>
<td>Supervisory Control</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java DataBase Connectivity</td>
</tr>
</tbody>
</table>
Nowadays, many organizations and companies deploy applications that are database-centric – they use databases by sending *Structured Query Language (SQL)* statements to them and obtaining data that result from executions of these statements. Since *database-centric applications (DCAs)* often share the same databases concurrently, database deadlocks routinely occur in these databases resulting in major performance degradation. Unfortunately, relational databases do not guarantee freedom from 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.

Currently, database deadlocks are typically detected only at runtime within database engines using special algorithms that analyze whether resources are held in cyclic dependencies, and these database engines resolve database deadlocks by forcibly breaking the hold-and-wait cycle. That is, once a deadlock occurs, the database rolls back one of the SQL statements that is involved in the circular wait. Doing so effectively resolves the database deadlock; this is why the database community has considered this problem solved for a long time. However, this resolution causes severe performance degradation, since DCAs should repeat the rolled back SQL statements to ensure functional correctness. Doing so not only reduces parallelism in executing SQL statements by databases, but also requires
these databases to re-execute the same SQL statements thus decreasing the throughput and negatively affecting the scalability of the DCAs.

In this dissertation, we present a novel solutions to a fundamental problem of predicting and preventing database deadlocks ($PD^2$) in DCAs, and we rigorously evaluate these solutions on several different open-source and commercial DCAs, some of which still exhibit database deadlocks after having been used for close to two decades. In evaluating ($PD^2$), we addressed various research questions that establish the effectiveness of our solutions. The results of this evaluation is made available along with the testbed and the tools that implement the solutions. The contributions are summarized below:

1. A novel abstraction and a performance model that hide the complexity of database engines while enabling analysis, prediction and prevention of database deadlocks.

2. A novel approach that combines dynamic analysis that automatically identifies and prevents database deadlocks with static analysis for detecting hold-and-wait cycles that specify how resources (e.g., database tables) are held in contention during executions of SQL statements.

3. A suite of new ($PD^2$) tools that were developed, evaluated, applied to different open-source and commercial DCAs, and are available to the broader community.

4. Advances that enhance parallelism and scalability of DCAs in different domains by alleviating difficulties in reaching the full performance potential of these DCAs.
CHAPTER 1

INTRODUCTION

1.1 Motivation

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 (Gray and Reuter, 1992) – and obtaining data that result from execution of these SQL statements. When DCAs use the same database at the same time, concurrency errors known as database deadlocks, are the cause of major performance degradation in applications (Nonemacher, 2006; Hellerstein et al., 2007). The responsibility of relational database engines is to provide layers of abstractions to guarantee Atomicity, Consistency, Isolation, and Durability (ACID) properties (Gray and Reuter, 1992); however, these guarantees do not include freedom from database deadlocks.

Consider a situation where different DCAs are executed in a complex enterprise workflow and these DCAs access the same database. This situation occurs in many well-known enterprise applications. For example, every minute thousands of users configure and order computers using online ordering systems from companies like Dell and Hewlett-Packard, and these systems include different DCAs that communicate with backend databases. Even though the source code may be free of deadlocks, performance of DCAs is seriously worsened by insidious database deadlocks that occur when DCAs issue transactions against the
database. In a survey of 148 enterprises that use DCAs, 92% said that improving system performance was a top priority (Yuhanna, 2009). Developers need efficient performance management tools for predicting database deadlocks in order to achieve better performance of software. The application performance management market is over USD 2.3 billions and growing at 12% annually, making it one of the fastest growing segment of the application services market (Ashley, 2006)(Garbani, 2008). For example, with computer ordering systems users select components as part of their computer configurations, and DCAs issue transactions that lock specific rows for some selected components in the database. A system administrator may use a different DCA to update components and lock corresponding rows for these components in the same database. Thus a circular wait is created, and if this situation is repeated often the system will be unusable.

Moreover, DCAs may run successfully for years and access the database without any problems. As a part of software evolution process, new DCAs can be developed at some later point that might need to share the same database. Developers of the earlier DCAs had no way to know about the development of the new DCAs. At the same way, the developers of the new DCAs might develop their applications without considering the earlier DCAs that share the same database. Hence, developers can develop DCAs independently without considering the fact that other DCAs might share the same database. As a result, earlier DCAs or newly developed DCAs work fine when they access the database independently, however, various kind of problems arise when DCAs share the database with other DCAs and database deadlock is one of those important problems.
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 (Coffman et al., 1971). Even though database deadlocks occur within database engines and not within DCAs that use these databases (see Section 2.1), these deadlocks affect the performance of the combined software system (i.e., the DCAs that interact with their databases) by reducing parallelism and decreasing throughput drastically. 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 a legacy DCA, thus making the process of software evolution error-prone and difficult.

1.1.1 *An Illustrative Example*

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

EXAMPLE OF A DATABASE DEADLOCK THAT MAY OCCUR WHEN TWO TRANSACTIONS $T_1$ AND $T_2$ ARE ISSUED BY DCA(S).

<table>
<thead>
<tr>
<th>Step</th>
<th>Transaction $T_1$</th>
<th>Transaction $T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UPDATE authors SET citations=100 WHERE paperid=1</td>
<td>UPDATE titles SET copyright=1 WHERE titleid=2</td>
</tr>
<tr>
<td>2</td>
<td>SELECT title, doi FROM titles WHERE titleid=2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>SELECT authorname FROM authors WHERE paperid=1</td>
</tr>
</tbody>
</table>

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, and 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 I. 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 $T_1 \to \text{authors} \to T_2 \to \text{titles} \to T_1$. The same reason-
ing applies if the granularity of locks is coarser, i.e., at the table level, – when interleaving steps occur as shown in Table I, a database deadlock is highly likely.

![Figure 1. A lock graph for the transactions shown in Table I. The lock graph shows the hold-and-wait cycle $T_1 \rightarrow \text{authors} \rightarrow T_2 \rightarrow \text{titles} \rightarrow T_1$.]

1.1.2 Some ways for database deadlocks to come into existence

Database deadlocks can come into existence in various ways. They can also be unrevealed during the development phase of DCAs and can appear in later time when the DCAs are in operation. For example, when more DCAs are added to the system and share some existing databases, that increases the likelihood of the addition of new transactions that might lead to hold-and-wait cycles and hence, increase the probability of database deadlocks. Database deadlocks can also be introduced when DCAs grow in complexity and complicated operations (i.e., join, nested SQL statement, etc.) are added into the transactions as a part of software evolution process. Another reason, that increase the probability database deadlocks, is when more data is added to the database. In this case, database engine needs more time to execute the SQL statements over the data. As a result, trans-
actions require locks on the data for a longer period of time, which in turns, increase the probability of hold-and-wait cycle and eventually, leads to database deadlocks. Sometimes, further SQL statements are added into transactions or order of the SQL statements inside the transactions are modified to facilitate additional functionalities or to upgrade some features in the system. That is another way for database deadlocks to come into existence.

1.2 Problems with Database Deadlocks

The problem with the database deadlocks is that they impact scalability and performance of DCAs. Unfortunately, it is not easy to address this problem. While there is a research on detecting and resolving deadlocks in software application, to the best of our knowledge there is no approach that prevents 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 (Griggs, 2003; Garcia-Molina, 1979; Lomet, 1980)(Rahimi and Haug, 2010, pages 249–252). For example, Microsoft\(^1\)(Kehayias and Krueger, 2011), Oracle\(^2\) and DB2 (Bruni et al., 2006, 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


\(^{2}\)http://docs.oracle.com/javadb/10.8.2.2/devguide/cdevconcepts53677.html
database objects in the same order, avoiding user interaction in transactions, and keeping transactions short and in one batch.

But, it is not always possible to follow these guidelines as it involves making changes and doing rearrangements within the transactions or inside the source code of DCAs. It is even more difficult in legacy systems that have been successfully running for a long time. Not only is this approach manual and error-prone, but it also interferes with query optimization performed by database engines by imposing an execution order of SQL statements, thereby preventing database engines to select an optimal execution schedule. Therefore, it is important to reproduce database deadlocks using DCAs automatically during testing to see what avoidance patterns fit best.

Obviously, attempting to design and write DCAs that avoid database deadlocks introduces significant complexity that negatively impacts the productivity of software engineers by requiring them to analyze and seek solutions without addressing this problem comprehensively.

Depending on the DCAs and the nature of database deadlocks, an avoidance pattern can be adopted by the developers that fits best for their case or database deadlocks can be prevented from the system. In the following sections, we discuss on testing, simulating and preventing database deadlocks.

1.2.1 Testing for Database Deadlocks

Testing applications to determine how they cause database deadlocks is important as part of ensuring correctness, reliability, and performance of these applications. Our inter-
views 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. 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. In this case, a test engineer would wait for ten days in order to detect a single database deadlock, which is obviously impractical. Thus, it is important to produce database deadlocks efficiently, so that database administrators, testers, and developers can understand the runtime concurrent behavior of their DCAs and ways to improve it.

Unfortunately, it is very difficult to reproduce database deadlocks consistently and systematically, since identifying execution scenarios that lead to database deadlocks requires sophisticated reasoning about the combined behavior of DCAs and their databases. The result of this process is overwhelming complex and a significant cost of reproducing database deadlocks.

There are many execution paths in a DCA (assuming that it doesn’t contain only a few lines of code) that it might take and at any given time, it can be at any point of its execution path. As a result, when multiple DCAs execute simultaneously, depending on the paths that they take, database engines can fetch various combinations of transactions.

For example, consider a system where 50 DCAs run simultaneously and they access a shared database. All these DCAs can send a transaction to the database at the same time. Hence, $2^{50}$ different combinations of transactions are possible in that system and the database can receive any one of these combinations of transactions at any particular time.
Moreover, an exponential number of interleavings are possible among the SQL statements of transaction that are issued to the database at any given time. Let’s consider a scenario with $t$ transactions, each containing $s$ statements. In this case, there are $st$ statements in total, ready to be executed by the database.

For the first transaction, there are $\binom{ts}{s}$ ways that the $s$ statements of the transaction could be ordered in a sequence of $t \times s$ statements. Once the order of the statements of the first transaction are determined, there are $(t \times s - s)$ remaining statements to be ordered. So, for the second transaction, there are $\binom{ts-s}{s}$ ways that the $s$ statements of the second transaction could be ordered. Then, $\binom{ts-2s}{s}$ ways for the third transactions, and so on. In general, for $t$ transactions, each containing $s$ SQL statements, the total number of interleavings are,

$$
\binom{ts}{s} \binom{ts-s}{s} \binom{ts-2s}{s} \cdots \binom{2s}{s} \binom{s}{s}
$$

$$
= \frac{(ts)!}{s!(ts-s)!} \times \frac{(ts-s)!}{s!(ts-2s)!} \times \frac{(ts-2s)!}{s!(ts-3s)!} \times \cdots \times \frac{(2s)!}{s!s!} \times \frac{s!}{s!}
$$

$$
= \binom{ts}{s} ! \cdot \frac{1}{(s!)^t}
$$

As it is shown, there are an exponential number of interleavings among the SQL statements of transactions and all these interleavings can be materialized by the database engine for execution. There is no guarantee that the interleavings, that cause database deadlocks, will occur frequently in subsequent executions, which makes the problem of consistent reproduction of database deadlocks hard and challenging.
1.2.2 Reproducing Versus Simulating Database Deadlocks

An alternative approach of reproducing database deadlocks is to simulate it using *mock objects* that throw exceptions when a transaction is sent to the database. That is, a mock object represents databases, making it easy for programmers to test their exception-handling code without having actual databases.

While this idea offers a simple implementation and may be effective in a number of situations, there are drawbacks. In DCAs, different exception objects propagate through layers of software by being caught and sometimes rethrown. The logic of exception handling depends on the application, and it is often unclear for testers how an exception should be handled, if at all.

Actual exception objects contain a wealth of information about the database deadlock which testers can utilize to understand the cause of the exception and report it. It is very difficult to generate this information through mock objects so that they reflect real database deadlocks. For example, *Apache Derby* database includes an error message within the thrown exception object in occurrence of a deadlock, which contains the transaction IDs, the statements, and the status of locks involved in a deadlock situation. It is very hard or impossible to precisely generate this error message through mock objects. Moreover, the execution of SQL statements may produce many exceptions at the same time, which is more likely in case of the execution of batch statements. In this case, all exceptions are added to a chain of exceptions and the head of this chain is thrown into the applications. Developers can iterate through all exceptions starting from the head of the exception chain.
Only actual exception object can provide this scenario which is not possible by using mock objects. So, providing information to understand how database deadlocks are caused is as important as generating database deadlocks. Moreover, in certain scenarios, it might be the case that timing constraints on the application (e.g., high-frequency stock trading) is important and exact time of the occurrence of deadlock exceptions are significant. This is why it is often more valuable to actually reproduce database deadlocks rather than just simulating them.

The other drawback is that using mock objects do not allow testers to observe actual database deadlocks and to obtain from the database engine SQL execution traces showing how database deadlocks happen. Database deadlocks often involve complex interactions among different database objects with hold-and-wait cycles among transactions that can not be simulated by mock objects. An SQL Server has over a dozen of documented kinds of database deadlocks\(^1\). For example, database deadlock may occur during savepoint rollback—a complicated set of locks are imposed by objects of the database engine leading to a database deadlock even if there are no hold-and-wait cycles in SQL statements of the transactions that are concurrently issued by different DCAs to the same database.

Moreover, a transaction, sent by some component of a DCA, to a database may invokes an internal program that sets off a series of invocations. For example, this internal program can be a trigger that calls an external component that sends a new transaction to a different

database that invokes a stored procedure that executes a different transaction that results in a database deadlock. In this kind of situation, it is not only important to identify what components catch database deadlock exceptions, but also to determine the execution trace that leads to the deadlock. Knowing a precise set of invocations that leads to a database deadlock enables developers to design a strategy to avoid this deadlock.

In addition, database deadlock resolution and exception throwing mechanisms are not perfect. In some cases, database deadlocks are incorrectly processed by database layers leading to null pointer exceptions\(^1\). Mock objects offer very limited benefits in such cases, because the source of the problem can be traced only with the actual database deadlocks.

### 1.2.3 Preventing Database Deadlocks

Preventing database deadlocks is another way of addressing the problems caused by them. In some situations, it is often not possible to fix the problem of database deadlocks, even if the cause is understood, since it would involve drastic redesign by changing the logic of the DCA to avoid certain interleavings of SQL statements among different transactions (Jula et al., 2008). In addition, fixing database deadlocks may introduce new concurrency problems, and frequently these fixes reduce the occurrences of database deadlocks instead of eliminating them (Lu et al., 2008). Therefore, 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.

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. Otherwise, for the purpose of finding all hold-and-wait cycles, database engine requires to know the application logic of every DCA and all of the possible interleaving among the transactions that could be issued by those DCAs. It also needs to be aware of any new addition of DCAs that could issue new set of transactions. This is neither practical, nor it is the purpose of a database management system. Second, and more importantly, database engines are designed to execute transactions efficiently, and imposing runtime analysis for finding all hold-and-wait cycles adds significant overhead. As it is described in 1.2.1, there are exponential number of possible combinations of transactions, any one of which could be issued by the DCAs at any given time and there are exponential number of possible interleavings among transactions, any one of which can lead to hold-and-wait cycles. Significant amount of time is required to analyze this many of possible interleavings, which prohibits database engine to analyze it during the runtime to reduce the overhead on performance.

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 of doing it is 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 hold-and-wait cycles (Bernstein et al., 1987; Hellerstein et al., 2007; Agrawal et al., 1987; Gray and Reuter, 1992; Mohan et al., 1986; Rosenkrantz et al., 1978; Hofri, 1994). 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 and this is why the database 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 aborted 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, where programmers should write special database deadlock exception-handling code that should repeat 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 these exceptions for different database engines. 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. Doing so not only reduces parallelism in executing SQL statements by databases, but also requires DCAs and their databases to re-execute the same SQL statements thus decreasing the throughput and negatively affecting the scalability of the DCAs.
To make things worse, when transactions are discarded, the results of valuables and long-running computations are lost, as it is especially evident in case of multi-level and long-lived transactions (Gray and Reuter, 1992, 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. 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, and this challenge persists and it is getting worse as this DCA evolves.

1.3 Our Objectives and Research Contributions

In this work, we enhance the parallelism and scalability of database-centric applications by preventing the occurrences of database deadlocks. We also present a solution for testing for database deadlocks in applications, so that testers can investigate the causes of database deadlocks and adopt defensive strategies to avoid them.

Our research is transformative, since it solves a decades old problem using a novel combination of ideas from different areas, and our program leads to practical tools for enhancing the performance, scalability, and reliability of Database-Centric Applications (DCAs). The result of our work is a foundation for a new direction supported by a set of tools for lightweight automated DCA performance management and maintenance.
Our work can be readily used by developers and database administrators to detect, reproduce, and prevent database deadlocks. Eventually, our technique can be used to develop tools for performance management of commercial applications.

The major contributions of our work are as follows:

1. We present a novel abstraction and a performance model that hides the complexity of database engines while enabling analysis, prediction and prevention of database deadlocks.

2. We devise programming and testing methodologies that enables programmers and testers to improve defensive programming of DCAs and their testing to balance the trade-off between the overhead of handling database deadlocks and performance and scalability of DCAs.

3. We introduce a novel approach that combines dynamic analysis, which automatically prevents database deadlocks, with static analysis for detecting hold-and-wait cycles that specify how resources (e.g., database tables) are held in contention during executions of SQL statements.

4. We develop and evaluate a suite of new $$(PD)^2$$ tools that will be applied to different open-source and commercial DCAs; these tools are available to the broader community.

5. We enhance parallelism and scalability of DCAs in different domains by alleviating and removing difficulties in reaching the full performance potential of these DCAs.
1.4 Outline

The dissertation is organized as follows.

• Chapter 2 presents the context of database deadlocks with examples on how DCAs use databases and provides a performance model to show when the database deadlocks cause major performance degradation. This chapter also describes the assumptions that we made for our work.

• Chapter 3 discusses the related work on predicting and preventing database deadlocks along with random SQL statements and test data generation.

• Chapter 4 provides the details on our tool for generating random SQL statements and strategy for introducing circular way of resource sharing among SQL statements that forms hold-and-wait cycles.

• Chapter 5 describes the cycles detection algorithm with a proof of completeness of the algorithm. This chapter also includes our approach for modeling transactions with Petri nets.

• Chapter 6 explains our solution of testing for database deadlocks and preventing database deadlocks. The architecture of our solution is describes in this chapter.

• Chapter 7 contains the empirical results and evaluations of our methods and approaches, with discussions on the outcomes of the experiments.

• Finally, Chapter 8 draws some conclusions and indicates the directions of future research.
CHAPTER 2

CONTEXT OF DATABASE DEADLOCKS

In this chapter, we describe the context of database deadlocks where we show how DCAs interact with the databases. We also give an overview of our solution with a performance model that explains the situations where our solution can be used effectively.

2.1 How DCAs Use Databases

Many enterprise-level DCAs are written in general-purpose programming languages (e.g., Java); they communicate with relational databases by sending transactions and obtaining data. Transactions contain Structured Query Language (SQL) statements and since they are atomic in nature, either all of the SQL statements in a transaction have to be executed by the database engines or none of the SQL statements executes. In case of a failure after partial execution of a transaction, database engine discards all changes made by the partial execution of that transaction.

In Figure 2, we show an architectural model where a DCA uses a database. As a part of communication, DCA sends transaction to the database. For the sake of simplicity, in this case, we show a transaction that contains only one SQL statement. Once the transaction is sent to the database, database engine parses the transaction, converts it into relational algebra operations, decides an optimal way of executing those operations and
finally, computes the results and sends it back to the DCA. As it is shown in the figure, DCA uses a connector for the communication purpose.

There are many available technologies that serve as a connector and facilitate connections between DCAs and databases. Among those, *Java Database Connectivity (JDBC)* and *Hibernate* are two most commonly used technologies. In the following sections, we describe JDBC and Hibernate.

### 2.1.1 Java DataBase Connectivity (JDBC)

*Java DataBase Connectivity (JDBC)* is a relational databases oriented data access technology that defines how a client can access a database. It facilitates application programming interfaces (APIs), which are java methods and DCAs interact with databases by invoking those API methods. Using JDBC APIs, DCAs can 1) Connect to a database 2) Send queries to a database to retrieve and update data 3) Process the results received from the database.

We give an example of a Java method in Figure 3 that shows how the JDBC APIs are used by an application.
Figure 3. Example of an application using JDBC.

As shown in Line 1, the method JDBCExample takes three parameters: `url`, `user` and `password`. These values are used as parameters to invoke `getConnection` method of `DriverManager` object provided by JDBC, as shown in Line 5. The `getConnection` API is used to connect to a database driver and log into the database, and it returns a JDBC `Connection` object.

In Line 6, an `Statement` object is created by invoking `createStatement` method of `Connection` object. Program sends SQL statements, to the database for execution, as string parameters of the API call `executeQuery` available in `Statement` object. The retrieved results of the query are kept into JDBC’s `ResultSet` object, after the successful execution of the query, as it is shown in Line 7. Once the results are retrieved, program can iterate through the results and process it by executing a loop (Line 8-10).
These SQL statements are executed as part of a transaction (not shown in the code fragment) that is delimited by statements “begin transaction” (by setting the connection’s autocommit mode to \texttt{false}) and “end transaction” with the subsequent API call \texttt{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 by the database.

The entire code segment (Line 5-9) is kept inside a \texttt{try} block to handle any sort of exceptions during the execution. An exception could be thrown for various reasons. For example, if the transaction, sent to the database, tries to access a table that is not present in the database, or there occurs a database deadlocks, then an exception is thrown. Depending on the type of the exception and the application’s logic, an exception is handled inside a \texttt{catch} block.

As it is shown in Line (12-15) of Figure 3, the program resends the transaction to the database in occurrence of a database deadlock which is identified by the associated exception number of the \texttt{SQLException} object.

Thus, JDBC presents an interface to programmers that allows them to concentrate on their application logic without using complicated reasoning about database engines.

\subsection*{2.1.2 Hibernate}

Hibernate provides a framework for \texttt{Java} language to bridge object oriented model and relational database model. Technical difficulties arise, when a relational database management system (RDBMS) is used by a program written in an object-oriented programming language, where database tables are directly mapped into an object or a class. Those diffi-
culties are mainly because of data type differences, transactional differences and structural and integrity differences between RDBMS and object-oriented programming language. Hibernate provides APIs to facilitate the mappings between RDBMS tables and Java classes. It also offers the interface to query the database and update the data inside the database.

In Hibernate, the association between the class and table is specified in xml format by using mapping files. A mapping file is used for providing Hibernate with the metadata to determine how to load and store objects of the class.

As an example, we give the definition of a class (Person) in Figure 4 and the corresponding mapping file in Figure 5. The Person class contains four private member variables (id, age, firstname, lastname) and a public constructor method Person().

```java
public class Person
{
    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person()
    {
    }
}
```

In the mapping file, hibernate-mapping is the root element which contains all the class elements.

The class elements are used to specify mappings between a Java class and corresponding database tables. Class name is specified using the name attribute of the class element and the database table name is specified using the table attribute.
The `id` element maps the unique ID attribute in class to the primary key of the database table. The name attribute of the `id` element refers to the property in the class and the column attribute refers to the column in the database table.

The `generator` element within the `id` element is used to automatically generate the primary key values. The `property` element is used to map a Java class field to a column in the database table. The attributes of `generator` and `property` elements are specified according to the hibernate guidelines.

Once all mappings are specified, various Hibernate APIs are used to connect to the database, to query the database and update the data in the database.

There are other connectivity technologies similar to JDBC and Hibernate, each with own advantages and disadvantages. Developers choose among technologies that match best for their DCAs.
2.1.3 Overview of Our Solution

To address the problems caused by database deadlocks, we provide our solution predicting and preventing database deadlocks \((PD)^2\). We provide two major techniques under the \((PD)^2\): technique for testing for database deadlocks in DCAs and technique for preventing database deadlocks in DCAs.

To implement our solution, first of all, we need to extract all of the transactions from all of the DCAs as it is shown as the innermost circle in the Figure 6. In next phase, we model the extracted transactions by using Petri nets and run our cycle detection algorithm to statically detect all hold-and-wait cycles among those extracted transactions, which is shown as the the circle in the middle of the Figure 6. We describe our modeling approach and the cycles detection algorithm in Chapter 5.

For the purpose of testing for database deadlocks, our approach, Systematic TEsting in Presence of DAtabase Deadlocks (STEPDAD), enables testers to instantiate database deadlocks in DCAs with a high level of automation and frequency. In STEPDAD, we introduce a mechanism for scheduling executions of DCAs to issue transactions that have simultaneous hold-and-wait cycles to the same database. As a slightly different approach, we also replicate transactions that have hold-and-wait cycles by issuing them simultaneously from different client applications to further increase the probability of observing database deadlocks. The purpose is to increase the probability of observing database deadlocks that result from these hold-and-wait cycles and hence, detect the deadlocks in DCAs.
In order to prevent database deadlocks, our approach, *pReventing databases Deadlocks from AppliCation-based Transactions (REDACT)*, uses the information of hold-and-wait cycles that exist in transactions, to prevent database deadlocks at runtime, by holding back one transaction that participates in the hold-and-wait cycle.

As it is shown in the outermost circle of Figure 6, both in STEPDAD and REDACT, we need the information of hold-and-wait cycles which is detected in cycles detection phase.

Our solution is not beneficial in all situations. In the following section, we present a performance model which is used to understand in which situation database deadlocks present a big performance problem rather than small inconvenience and to determine when our proposed solution will be most beneficial.
2.1.4 Performance Model

A performance model for analyzing the impact of database deadlocks is shown in Figure 7. This model uses a standard template for discrete time analysis in the performance evaluations of different systems (Dummler and Schomig, 1999).

Transactions arrive from DCAs at the arrival rate $\lambda_a$ (transactions per unit time), and 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 hold-and-wait 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 $\lambda_d$ (transactions per unit time). 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 $\lambda_a$. That is, we assume that the frequency of database deadlocks is proportional to the transaction arrival rate, which we observed in different projects.

The relation between two independent variables, $\lambda_a$ and $\lambda_d$ is important to determine the impact of database deadlocks. Consider two cases when $\frac{\lambda_d}{\lambda_a} \gg 1$ and $\frac{\lambda_d}{\lambda_a} \leq 1$. The underlying physical event for the departure rate $\lambda_d$ is the time it takes by the database to process transactions and to produce results. Thus, the first case $\frac{\lambda_d}{\lambda_a} \gg 1$, where the average time per transaction is measured in milliseconds or seconds, means that transactions
are processed by the database much faster than they arrive – this is typical for smaller applications where transactions retrieve smaller amounts of data without applying complex operations like joins and aggregations. Database deadlocks do not cause major performance degradation in this case.

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 time (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 the database deadlock detection time is equal to or greater than the mean transaction completion time.

The other case, $\frac{\lambda_d}{\lambda_a} \leq 1$, where the average time per transaction is measured in minutes or 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 approximately the same rate that they are processed – if a database deadlock results in aborting a long-running transaction that is put back into the Queue, it will have a devastating effect on the performance of the system. Our simulation with the performance model showed nonlinear decrease in the throughput time, meaning that the system loses its scalability under realistic conditions.

2.2 Our Assumptions

In this section, we describe the assumptions that we made in various parts of our work.

2.2.1 Structure and Extraction of SQL statements

In this work, we deal with three of the most commonly used SQL statements: SELECT, INSERT and UPDATE. We represent relational databases as sets of resources (e.g., database tables) and model SELECT statement as a read operation from these tables, and INSERT and UPDATE statements as write operations into these tables. With this abstraction, we hide the complex machinery of database engines and concentrate on read/write operations performed by SQL statements.

Although, the SELECT, INSERT and UPDATE statements are formed by following a complex set of grammar rules, we give a simple example of each type in Figure 8 to explain the structure of these statements.
Figure 8. Structure of SQL statements.

In Figure 8, \( \text{col}_i \) represents the \( i \)-th column of the table \( \text{table}_\text{name} \) and \( \text{val}_i \) represents the value for \( \text{col}_i \), where \( i = 1, 2, \ldots, n \). The \( \text{where}_\text{clause} \) specifies the criteria, expressed in the form of predicates and the statements affect the rows that meet the criteria specified in \( \text{where}_\text{clause} \).

SQL statements have mutable and immutable parts, and the \( \text{where}_\text{clause} \) is optional. In Figure 8, capitalized words, shown within the SQL statements, are immutable and cannot be modified by the applications. Other part of the statements (non-capitalized), could be arbitrarily built by the applications by using the variables or constants. But, in this work, we assume that the \( \text{table}_\text{name} \) in SQL statements are also immutable, and it is not assigned through a variable or not provided during the runtime. Of course, the examples given in Figure 8 are very simple and there are various kind of SQL statements, with complicated structures, that contain nested queries, join operations, aggregate functions, etc. Our assumption on \( \text{table}_\text{name} \) also applies on those SQL statements, which means all table names belong to a complicated SQL statement are available during the static time.

Our solution involves extracting transactions, that contain SQL statements, from Database-Centric Applications (DCAs) and we perform this extraction process statically. Since, we represent relational databases as sets of resources (e.g., database tables) and model SQL statement as \text{read/write} \ operations, we need to collect the information of these resources.
and type of operations performed by the extracted SQL statements. According to our assumption on table names, we are able to gather the information of resources from the extracted SQL statements and the type of operations (i.e., read/write) are also available in the immutable part of the statements. Moreover, in these SQL statements, we don’t need to analyze the predicates of the \textit{where clause} parts. Because, the \textit{where clause} parts determine the rows of tables that should be affected by the read/write operations performed by these SQL statements. But, in our solution, we consider the \textit{table-level} locking strategy that locks the entire table instead of some rows of a table (see Section 2.2.2). Hence, we can safely ignore the information given as the predicates of \textit{where clause}.

At first glance, it appears to be tedious and laborious work for programmers to extract SQL statement from the source code of DCAs. But, we assume that this is a one-time effort that could be done manually, as it is done for three subject DCAs in this dissertation. 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.

We observe that in many projects it is a small fraction of code that deals with obtaining values from databases, and most code is written to implement application logic that processes these values. This observation is confirmed by our previous study (Grechanik et al., 2010) that shows that out of 2,080 randomly chosen Java programs in Sourceforge, there
is approximately one SQL statement per 2,200 LOC on average. Extrapolating this result means that programmers may have to extract 450 SQL statements for a project with one million LOC and such expense is acceptable.

Also, there are various tools available to aid the extraction process of SQL statements from applications. For example, Toad for Oracle is used for the purpose of extracting SQL Statements that are embedded inside of a program written in PL/SQL, which is Oracle Corporations procedural language extension for SQL and the Oracle relational database. Moreover, extraction of SQL statements from applications has successfully been performed by the researchers (Halfond and Orso, 2005) and it is a subject to our future work.

Since we extract all transactions from all DCAs before we model them, we can safely assume that the execution time of a transaction doesn’t depend on the various programming language constructs (i.e., various loops, conditional constructs, artificial delays etc.), used to implement DCAs. A transaction can be short-lived or long-lived, irrespective of the application that it belongs to, depending on the structure of SQL statements that it contains. For example, SQL statements containing operations like nested join, data casting or sorting usually take more time than the SQL statements that do not contain these operations. Another factor, that affects the execution time of a transactions, is the amount of data that the transaction needs to access. For example, consider the first statement of transaction $T_1$ given in Table I which is “UPDATE authors SET citations=100 WHERE paperid=1”.

---

1 http://www.toadworld.com/products/toad-for-oracle/default.aspx
This statement updates the rows of authors table where the value of the attribute paperid is 1. This SQL statement can be short-lived if there are only a few rows in authors table where paperid is 1. Same SQL statement can be long-lived if there are a lot of rows in authors table where paperid is 1.

2.2.2 Locking Strategies

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 (Team, 2007, page 184). Using a locking strategy, we approximate the complex sequences of steps of transaction processing by conservatively modeling SQL statements using abstract operations according to a given locking strategy.

In fact, since all major databases expose interfaces that allow database administrators to modify locking strategies easily, we use this flexibility in setting locking strategies to adjust them to match approximations. If we determine that most database deadlocks occur when locking strategy is set on the table level, we can attempt to model the system by changing the locking strategy to be set on the row level and recompute all hold-and-wait cycles using more precise models that are based on row-level locking.
If not enough information can be obtained from SQL statements at compile time to determine what rows are likely to be operated on in the database and subsequently, spurious deadlocks may be reported, we can go back to the table-level locking. Moreover, there are situations when row-level locking systems can lock the entire tables instead of locking rows in the table. For example, a row-level locking is automatically escalated to table-level locking when the \textit{WHERE} clause of a statement cannot use an index or a high number of single-row locks would be less efficient than a single table-level lock.\footnote{http://docs.oracle.com/javadb/10.6.1.0/devguide/rdevconcepts8424.html}

With this flexibility, we achieve the goal of making our approach database engine-independent while ensuring that it is highly adjustable for a specific database platform. In this work, we follow the conservative approach and model the system by using table-level locking strategy.

\subsection{2.2.3 False Positive Cycles}

As a part of our solution, we statically detect all hold-and-wait cycles (shown in Figure 1) that can occur in SQL statements that belong to different transactions by running our \textit{cycles detection algorithm} (see Chapter 5). We assume that all of these cycles lead to database deadlocks. But, depending on different execution scenarios, not all of these cycles will lead to database deadlocks, meaning that false positives cycles are possible.

For example, a deadlock will not occur if conflicting transactions attempt to lock disjoint sets of rows in at least one of the tables involved, whereas the rows being locked depend
TABLE II

EXAMPLE OF A SCENARIOS WHERE DATABASE DEADLOCK DOESN'T OCCUR BUT A HOLD-AND-WAIT CYCLE IS DETECTED.

<table>
<thead>
<tr>
<th>Step</th>
<th>Transaction $T_1$</th>
<th>Transaction $T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UPDATE authors SET citations=100 WHERE paperid=1</td>
<td>UPDATE titles SET copyright=1 WHERE titleid=3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SELECT title, doi FROM titles WHERE titleid=4</td>
<td>SELECT authorname FROM authors WHERE paperid=2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

on the outcome of WHERE clauses appearing in queries. We elaborate this scenario using an example given in Table II, which is a slightly modified version of the example given in Table I.

In this example, both transaction $T_1$ and transaction $T_2$ access the authors table in incompatible mood ($T_1$ writes and $T_2$ reads). But, $T_1$ only requires to locks the rows in which the value of attribute paperid is 1 and $T_2$ requires to locks the rows in which the value of attribute paperid is 2. Hence, they require locks on disjoint sets of rows of authors table. At the same way, $T_1$ access the rows of titles table in which titleid= 3 and $T_1$ access the rows of titles table in which titleid= 4. So, they require the locks on disjoint sets of rows of titles table. Hence, no transaction needs to wait for resources locked by another transaction and eventually, no deadlock occurs in this situation.
But, in our approach, since we model the system assuming table-level locking strategy, when a transactions requires to access some rows in a table, we lock the entire table instead of rows of the table, irrespective of the conditions specified in \texttt{WHERE} clause. As a result, our approach detects a hold-and-wait cycle in transactions $T_1$ and $T_2$ of Table II, using the same reasoning shown for the example in Table I. But, this hold-and-wait cycle is never materialized into a deadlock.

Alternatively, when tables contain no data, locks may be released by the database engine almost instantaneously or not imposed at all. In the example given in Table I, when both the \texttt{authors} and \texttt{titles} contain no data, there exists no resources for holding or waiting on. But, our approach doesn’t consider the volume of data in the tables and reports a hold-and-wait cycle. Again, a hold-and-wait cycle is reported that is never materialized into a deadlock.

Finally, in our context, the \texttt{soundness} property of our \texttt{cycles detection algorithm} holds when all cycles, detected by the algorithm, lead to database deadlocks and the \texttt{completeness} property of the algorithm holds when all cycles, that exists in transactions and lead to database deadlocks, are detected by the \texttt{cycles detection algorithm}. As it is shown with examples above, our approach is not \texttt{sound} since it reports false positive hold-and-wait cycles and according to our assumptions, these cycles lead to database deadlocks which is not true. But, our approach is \texttt{complete} since it reports all hold-and-wait cycles that exist in the SQL statements of all extracted transactions. We give our proof on \texttt{completeness} in Section 5.3.1.
CHAPTER 3

RELATED WORK

Our work consists of the techniques for testing for the database deadlocks and preventing database deadlocks. We use Petri nets for modeling transactions and a random generator for the purpose of SQL statements generation. In this section, we present the related work, explored before by the research community in various ways.

3.1 Testing for Database Deadlocks

Language-based approaches offer different type systems and annotation facilities for programmers to annotate programs, so that type checkers can analyze and detect deadlocks (Gerakios et al., 2011; Boyapati et al., 2002). Given that DCAs contain embedded SQL statements, this approach requires a combination of two type systems: one for SQL and the other for the host language in which DCA is written.

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 (Engler and Ashcraft, 2003). Williams et al. (Williams et al., 2005) 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 (Jula et al., 2008). It uses detected hold-and-wait cycles to prevent database deadlocks, but unlike our approach Dimmunix is not designed to reproduce them and the deadlock patterns in Dimmunix are loose approximations that result in many false positives, especially since control-flow of DCA is not applicable to detect database deadlocks.

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 (Cai and Chan, 2012). A dynamic approach Sammati provides automatic deadlock detection and recovery for POSIX threaded applications (Pyla and Varadarajan, 2010). Unlike MagicFuzzer and Sanmati, our approach performs its analysis at compile time to detect deadlocks and prevent it at runtime.

Pike is a concurrency bug detector that automatically identifies when an execution of a program triggers a concurrency bug (Fonseca et al., 2011). Pike is related to STEPDAD in that it uses a scheduler to control thread interleaving by intercepting certain library calls and forcing a thread to run at a time that is randomly chosen by the scheduler in an attempt to reproduce a bug. In that, Pike is complementary to STEPDAD, which can use ideas
from Pike to improve its scheduler. Unlike Pike, STEPDAD deals with database deadlocks, and it is unclear how Pike can be extended to handle such deadlocks.

Williams et al. (Williams et al., 2005) defined a deadlock detection algorithm for Java libraries similar to our strategy for generating deadlock supervisors. In contrast with our method, they derive lock graphs directly from Java source code. We prefer to build lock graphs from Petri net models because we believe that lock graph construction will be more efficient from these models than the case of Java sources. In addition, Petri net models that we consider for lock graph generation already include control supervisors for enforcing mutual exclusion constraints, whereas our original Java sources evidently do not include such supervisors.

Approaches for preventing deadlocks using transactional memory are gaining increasing popularity (Koskinen and Herlihy, 2008; Volos et al., 2012). Unfortunately, database deadlocks often occur in the distributed setting where there is no shared memory. In contrast to STEPDAD, these approaches may be applicable to reproduce deadlocks among different threads within the same process, but not for distributed environment where external databases and DCAs could be located on different computers.

3.2 Preventing 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 (Qin et al., 2007). This solution cannot be used in the context of our research,
since rolling back a DCA significantly worsens its performance – retrying these transactions incurs a significant performance penalty.

Like our solution REDACT, snapshot isolation partially addresses the problem of database deadlocks by avoiding conflicting concurrent updates that may lead to inconsistent snapshots (Cahill et al., 2009). In contrast to REDACT, exceptions are thrown when snapshot isolation is violated, leading to the same performance problem that REDACT addresses 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 (Kallman et al., 2008). It is a new concept and our goal of future work is to experiment to compare REDACT and H-Store.

Different algorithms that use Petri nets have been created to generate supervisory controls to prevent deadlocks (Iordache and Antsaklis, 2006; Li et al., 2008). The work of Yamalidou et al defined a tractable method for enforcing sets of linear mutual exclusion constraints on the reachable markings of a Petri net (Yamalidou et al., 1996). Different algorithms have been proposed for enforcing freedom from deadlock in manufacturing systems (Li and Zhou, 2008; Zhou and Fanti, 2004). He and Lemmon defined a method based on a construction called a Petri net unfolding (He et al., 2000).

However, Xie and Giua later discovered that He and Lemmon’s method can sometimes allow deadlocks to occur (Xie and Giua, 2004). Unlike REDACT, most of these methods suffer from high complexity and it is unclear how to use these methods in the context of the database deadlock problem. Moreover, many Petri net-based approaches suffer from
exponential complexity in the sizes of the models that they operate on, thus making it
difficult to adjust these approaches to work in the REDACT context.

LaFortune et al (Wang et al., 2009) and Iordache et al (Iordache et al., 2002) 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.

Similar to our method, Wang et.al. (Wang et al., 2008)(Wang et al., 2009) propose a Petri
net based supervisory control approach to avoid deadlocks in concurrent software written in
Java. They specifically use supervision based on place invariants to create the supervisory
controller. Their approach consists of two phases. First, in the so-called offline phase a C
program is coupled with the compiler that is used to compile the target concurrent software
in order to generate a Control Flow Graph (CFG) of the original code. The generated CFG
is then translated into a Petri net. In contrast, our work targets database deadlocks and
we analyze Java code to recover SQL statements that are sent to databases and our work
avoids computational complexity of using siphons.

Our work uses the Petri net to generate a control supervisor that prevents the concurrent
software from experiencing deadlocks. The implementation uses the concept of a Petri net
siphon. In brief, a siphon is a place subset satisfying the following behavioral condition. If
all places in a siphon have no tokens in a given net state, then all such places will never receive a token in all subsequent states of the net (Murata, 1989).

Any deadlock state for a Petri net must contain at least one empty siphon (Reisig, 1985). In summary, the supervisory controllers of Wang et al. cleverly exploit this property by preventing siphons contained in the net from becoming empty in an effort to impose freedom from deadlock. A disadvantage of their method is computational complexity because Petri net may contain a number of siphons exponential in Petri net size.

In addition, the controllers preventing siphons from becoming empty may introduce additional siphons requiring the analysis to be repeated. In contrast, we are using the work of Yamalidou et al. that defined a tractable method for enforcing sets of linear mutual exclusion constraints on the reachable markings of a Petri net $\mathcal{N}$. Yamalidou et al. showed that the supervisory controllers generated in this fashion are maximally permissive (Yamalidou et al., 1996).

Other authors have investigated supervisory control for enforcing freedom from deadlock on Petri net models. For instance, Iordache et al. (Iordache et al., 2002) originally defined a method based on Petri net siphons. The problem with approach is that a Petri net may have a number of siphons exponential in the size of the net. This case is similar to the work of Wang et al. that we discussed above (Wang et al., 2008).

Li and Zhou define tractable algorithms for enforcing freedom from deadlock in so-called flexible manufacturing systems (Li and Zhou, 2008). However, their algorithms are applicable to a special class of Petri nets, called $S^3 PR$ nets, which model such systems. Nets
obtained from translation from Java source code do not fall into this class of nets. Kevin He and Lemmon defined a method based on a construction called a Petri net unfolding (He et al., 2000). However, Xie and Giua later discovered that He and Lemmon’s method can sometimes allow deadlocks to occur (Xie and Giua, 2004). Additional methods are discussed elsewhere (Iordache and Antsaklis, 2006; Li et al., 2008).

Approaches for preventing deadlocks using transactional memory are gaining increasing popularity (Volos et al., 2012), but unfortunately, database deadlocks often occur in the distributed setting where there is no shared memory. In contrast to REDACT, these approaches are applicable to prevent deadlocks among different threads within the same process, and not for distributed environments where external databases and DCAs are located on different computers. A remaining problem with transactional memory approaches is that it worsens performance, and locking mechanisms are used to improve performance, resulting in situations where deadlocks still occur.

3.3 Random SQL statements generation

While RUGRAT for SQL leverages the grammar of a programming language to generate programs, there are other program generation techniques that are not based on grammars. For example, Sreenivasan and Kleinman describe a technique for synthesizing programs that produce close-to-realistic workloads for hard drives (Sreenivasan and Kleinman, 1974). The approach composes individual workloads to match certain probability distributions. Unlike this approach, RUGRAT’s goal is to create SQL statements that use a wide variety of complex SQL features.
A few other approaches are created for generating SQL statements and query sets. Probabilistic test data generation has been successfully used in testing relational database engines, where complex SQL statements are generated using a random SQL statement generator (Slutz, 1998). QGEN, a flexible, high-level query generator optimized for decision support system evaluation. QGEN generates arbitrary query sets, which conform to a selected statistical profile without requiring that the queries be statically defined or disclosed prior to testing (Poess and Stephens, 2004). QGEN links query syntax with abstracted data distributions, enabling users to parameterize their query workload to match an emerging access pattern or data set modification.

Another recent approach for random SQL generation is a work by Khurshid et al. that generates syntactically and semantically correct SQL queries as inputs for testing relational databases (Abdul Khalek and Khurshid, 2010). They leverage the SAT-based Alloy tool-set to reduce the problem of generating valid SQL queries into a SAT problem. With their approach, SQL query constraints are translated into Alloy models, which enable it to generate valid queries that cannot be automatically generated using conventional grammar-based generators.

3.4 Random Test Data Generation for Different Domains

In the remainder of this section, we focus on related grammar-based test input generation techniques. Grammar-based test input generation was pioneered by Hanford (Hanford, 1970) and Purdom (Purdom, 1972) in the 1970s and can be roughly divided into two broad
categories, random and systematic. Additional related work can be found in a survey article on generating programs for compiler testing (Boujarwah and Saleh, 1997).

Several earlier pieces of work have used probabilistic grammar-based random program generation before (Murali and Shyamasundar, 1983; Maurer, 1990; Burgess, 1994; Sirer and Bershad, 1999; Sirer, 1999; Yoshikawa et al., 2003; Yang et al., 2011; Cuoq et al., 2012). However earlier work mostly focused on testing and debugging. These approaches thus tried to systematically cover corner cases and bugs that are otherwise hard to find. To simplify debugging, the focus was on triggering these corner cases with minimized, focused programs or program fragments. From our perspective, the earlier approaches could be described as generating a collection of maximally diverse micro-benchmarks of rare program shapes. We aim at end-to-end benchmarking and therefore generate large, complex benchmark SQL statements that are close to realistic SQL statements but satisfy specific user-defined constraints.

An early expressive language for grammar-based random program generation is presented by Maurer (Maurer, 1990). That is, the DGL Data-Generation Language is more expressive than context-free languages, as it supports various actions. The approach generates test suites in the C programming language for functional testing of VLSI circuits.

Sirer and Bershad (Sirer and Bershad, 1999) describe probabilistic testing with production grammars. A production grammar is a context-free grammar that can be enhanced with probabilities and actions. The work also introduces the concrete domain specific lan-
language (DSL) lava for specifying production grammars. The lava language was used to generate Java bytecode programs for testing Java virtual machines.

Other than testing C compilers, Cuoq et al. used Csmith for testing static analyzers (Cuoq et al., 2012). They tested Frama-C, a 300kLOC size framework for analysis and transformation of C programs and found 50 bugs.

Combinatorial coverage of grammar production rules is an alternative to stochastic production rule coverage. Purdom has defined a pioneering algorithm for generating small test programs from a given programming language grammar. That is, Purdom’s algorithm generates programs that cover each production rule of a given context-free grammar (Purdom, 1972).

Boujarwah et al. implement Purdom’s algorithm for a subset of Java (Boujarwah et al., 1999). However the implementation has not been applied to generate entire programs and no empirical results are available.

Lämmel and Schulte (Lämmel and Schulte, 2006) describe the general-purpose syntax-driven test-data generator Geno. Geno works on grammars written in a hybrid of EBNF and algebraic signatures. Geno systematically achieves a user-defined combinatorial coverage of the grammar’s production rules. Geno supports computations during test data generation, yielding expressiveness similar to attribute grammars. However Geno is also not available for experimentation.

In recent work, Hoffman et al. present YouGen, a practical tool for combinatorial production rule coverage (Hoffman et al., 2011). Similar to earlier work, YouGen takes as
input a context-free grammar. YouGen has a wider range of configuration options than previous combinatorial production rule coverage generators.

Exhaustive test program generation aims at enumerating all possible test programs up to a given size. Coppit and Lian describe yagg, a generator for test data generators that exhaustively enumerate all possible test data up to a given length (Coppit and Lian, 2005). The yagg tool supports context-free input grammars that can be enriched with semantic actions.

ASTGen by Daniel et al. systematically generates small Java programs (Daniel et al., 2007). However, ASTGen requires the user to combine several generators. More importantly, many generated programs have compile errors, and they do not have complex structures (e.g., only value equality ‘==’ is supported in conditions and no deep ‘if’ nesting is possible).

Majumdar and Xu describe a directed test program generation technique that attempts to exhaust the execution paths of a particular compiler or program analysis tool under test (Majumdar and Xu, 2007). The technique converts a given context-free grammar into a symbolic grammar, exhaustively derives all possible symbolic strings (programs) up to a certain size, and uses these strings in a dynamic symbolic or concolic execution as inputs to the program under test. This directed search yields a small set of representative test programs, as the symbolic reasoning prevents the generation of concrete input programs that cover the same path in the program under test. On the other hand, symbolic reasoning is very expensive, which limits the scalability of the technique. The corresponding tool,
CESE, has been used to generate small test programs. RUGRAT on the other hand can quickly generate very large random test programs independent of any particular program under test.

Beyond grammar production rules, other models of programming language specifications exist. Such models often encode rich semantic information and can be covered systematically by program generators. Given the richness of the information encoded in these models, test case generators are typically slower and focus on generating small programs that are focused on testing specific features.

For example, Zhao et al. capture the rules under which individual compiler optimizations can be applied in temporal logic (Zhao et al., 2009). The JTT tool then systematically generates focused test programs to test individual compiler optimizations. However it is not clear how this approach scales to entire applications and especially large-scale benchmark applications.

Generating random images is widely used to evaluate image processing and pattern recognition algorithms (Mantere and Alander, 2000; Reghbati and Lee, 1988). Essentially, finding images with desired properties to evaluate specific algorithms is difficult and laborious; not always these images can be located on the Internet. Yet it is important to obtain images that have specific geometric figures that highlight certain properties of algorithms that use these images. Generating images with desired properties is a standard practice in image processing and pattern recognition.
CHAPTER 4

RANDOM SQL STATEMENTS GENERATION

We automatically generate transactions (SQL statements) to run experiments on our cycle detection algorithm that detects all hold-and-wait cycles in those transactions and we develop a tool Random Utility Generator for pRogram Analysis and Testing (RUGRAT) for the purpose of transactions generation. RUGRAT-generated transactions can be used as benchmarks. In this chapter, we present the importance of benchmarks and describe our approach to implement our benchmark generator tool RUGRAT. In the next chapter, we describe the cycle detection algorithm where we use RUGRAT-generated transactions as input to that algorithm.

4.1 Benchmark for SQL Statements

A benchmark is a point of reference from which measurements can be made in order to evaluate the performance of hardware or software or both (McDaniel, 1994). Benchmarks are important, since organizations and companies use different benchmarks to evaluate and choose mission-critical software for business operation (Kanoun and Spainhower, 2008). Businesses are often confronted with a limited budget and stringent performance requirements while developing and deploying enterprise applications, and benchmarking is often the only way to choose proper infrastructures from a variety of different technologies for these applications. For example, application benchmarks play a crucial role in the U.S.
Department of Defense acquisition process (William A. Ward, 2005). Given that corporations spend between 3.4% and 10.5% of their revenues on technologies, biased or poorly suitable benchmarks lead to wrong software and hardware architecture decisions that result in billions of dollars of losses every year (Nash, 2007).

Consider a situation where different test input generation techniques are evaluated to determine which one achieves higher test coverage in a shorter period of time (Park et al., 2012). Typically, test input generators use different algorithms to generate input data for each application run, and the cumulative statement coverage is reported for all runs as well as the elapsed time for these runs. On one extreme, “real-world” applications of low complexity are poor candidate benchmarks, since most test input data generation approaches will perform very well by achieving close to 100% statement test coverage in very few runs. On the other extreme, it may take significant effort to adjust these approaches to work with a real-world distributed application whose components are written in different languages and run on different platforms. Ideally, a large number of different benchmark applications are required with different levels of code complexity to appropriately evaluate test input data generation tools.

Writing benchmark application from scratch requires a lot of manual effort, not to mention that a significant bias and human error can be introduced (Joshi et al., 2008). In addition, selecting commercial applications as benchmarks negatively affects reproducibility of results, which is a cornerstone of the scientific method (Schwab et al., 2000), since commercial benchmarks cannot be easily shared among organizations and companies for legal
reasons and trade-secret protection. For example, Accenture Confidentiality Policy (item 69) states that source code, which is generated by the company and relates to its business, research and development activities, clients or other business partners, or employees are considered confidential information and other companies have similar policies. Finally, more than one benchmark is often required to determine the sensitivity of the approaches based on the variability of results for applications that have different properties.

Ideally, users should be able to easily generate benchmark applications with desired properties. 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 have desired properties that are specific to these aspects of SQL optimization, for example, complicated nested SQL statements that contain multiple joins. Since the meaning of SQL statements does not matter for performance evaluation, this generator creates semantically meaningless but syntactically correct SQL statements thereby enabling users to automatically create low-cost benchmarks with significantly reduced bias.

4.2 Our Approach

In this section, we describe our approach of random SQL statements generation that we use to implement our tool RUGRAT and generate transactions that access shared resources (database tables). We use the terms ‘SQL statements’ and ‘transactions’ interchangeably.

\[1\]https://policies.accenture.com/Pages/0001-0100/0069.aspx
RUGRAT-generated transactions maintain the desired properties that we specify as configurations. We implement the tool by using 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.

4.2.1 Stochastic Grammar Model

We generate transactions using the reverse notion of parsing SQL statements. Consider that every transaction is an instance of the SQL grammar. Typically, grammars are used in compiler construction to write parsers that check the syntactic validity of a program and transform its source code into a parse tree. An opposite use of the grammar is to generate branches of a parse tree for different production rules, where each rule is assigned the probability with which it is instantiated in a program. These grammars and parse trees are called stochastic, and they are widely used in natural language processing, speech recognition, information retrieval (Cohen and Kimelfeld, 2010), and also in generating SQL statements for testing database engines (Slutz, 1998).

We obtain random transactions by an algorithm based on the stochastic grammar model. Starting with the top production rules of the grammar, each nonterminal is recursively replaced with its corresponding production rule. When more than one production rule can be chosen to replace a nonterminal, a rule is chosen based on the probability that is assigned to it. (This is a configurable parameter.) Thus the higher the probability that is assigned to a rule, the more frequently its instances will appear in generated programs. Terminals are replaced with randomly generated identifiers and values that preserve the syntax rules.
of the given language. Termination conditions for this process of generating transactions include the limit on the size of the transaction in terms of the number of SQL statements that it contains. With the stochastic grammar model we guarantee that the generated transaction is syntactically correct.

In addition to the rules that are found in a typical context-free grammar of a programming language, our approach takes into account additional rules and constraints that are imposed by the SQL statements specification. For example, number of tables used in statements or percentage of keys in a table can be restricted by a defined value. With such an enhanced stochastic grammar model it is ensured that the generated SQL statements are syntactically correct and compiles. The construction process can be fine-tuned by varying the ranges of different configuration parameter values and limiting the grammar to a subset of the production rules that are important for evaluating specific tools.

We address one main goal—to allow experimenters to automatically generate SQL statements that have desired properties for evaluating various approaches and tools on SQL statements and database transactions. We do not see RUGRAT as a replacement of real-world SQL statements and database transactions for evaluating approaches and tools. We rather see RUGRAT as a tool that enables experimenters to quickly generate a large number of SQL statements that have desired properties. The goal of RUGRAT is thus to supplement evaluations of tools using real-world application benchmarks. In a way, we see RUGRAT as a rapid prototyping tool for producing a set of SQL statements and database transactions for initial evaluation of various approaches and tools.
Statements generated by RUGRAT cover a wide variety of language constructs that are important for evaluation purposes. Sample constructs include various kind of join operations, data types, nested queries, foreign keys, etc. RUGRAT can also seed hold-and-wait cycles in transactions which ensures that resources (tables) are shared in transactions in a circular fashion (see Section 4.4). The intention is to verify if the cycles detection algorithms can detect the seeded hold-and-wait cycles. Existing program generators often do not take into consideration such language constructs and do not add them to generated programs.

In addition, there is a requirement that generated SQL statements should represent real-world DCAs using software metrics. This requirement is motivated by the needs of the potential RUGRAT users. As a consequence, we tune the default parameters of RUGRAT such that generated statements are as similar as possible to what one would consider a normal hand-written statements. We implement this issue by varying the probabilities that are assigned to different production rules of the language grammar.

4.2.2 Benefits

Our approach scales to generating statements that at the same time are large, have complex properties, and are similar to hand-written SQL statements and transactions. While RUGRAT-generated programs are similar to hand-written programs, our approach provides multiple benefits over benchmarking with hand-written applications.
First, using RUGRAT one can easily generate a large variety of random statements. Such a large set of statements can complement hand-written statements in DCAs, which are often relatively small sets of statements.

Second, our approach scales down from realistic applications to toy applications that only contain a specified set of language features. This down-scaling is useful during tool development. That is, at an early tool development stage, a tool may only be able to handle a few SQL features. At this point a tool developer may still want to test her tool on large applications. However, hand-written applications often use multiple SQL features and it may be hard to find hand-written applications that only use a given set of SQL features, especially when looking for a variety of larger applications.

Third, since RUGRAT-generated applications do not have external dependencies, it is very easy to compile, install, execute, and test RUGRAT-generated SQL statements. In contrast, hand-written statements are often difficult to install and execute. On the other hand, before a realistic hand-written application can be tested, external dependencies have to be resolved. For example, additional systems such as applications, servers, and communication infrastructure has to be installed and configured. A survey on evaluating static analysis tools and benchmarks showed that most user-reported failures in software repositories are false failures, i.e., failures that will not be fixed as they do not concern the code (Wedyan et al., 2009). Instead, the false failures are mostly installation failures, which may be caused by poor documentation and difficult deployment procedures. RUGRAT users
avoid this potential pitfall as RUGRAT-generated statements do not require any installation and can be compiled and executed immediately.

Finally, our approach enables more experiments at lower cost by providing, on demand, many high-quality programs in short time. When evaluating a tool with hand-written SQL statements, if the tool developer wants to evaluate the tool with more statements, she needs to explore code repositories with specific requirements, which can be time-consuming. However, with RUGRAT she can generate such SQL statements automatically in a short amount of time by specifying such requirements as parameters to RUGRAT.

4.3 Implementation

We explain the RUGRAT’s SQL statements generation process by using an example shown in Figure 9. This figure represents an abstract syntax tree corresponding to a set of grammar production rules and it also shows how a statement is generated by exploring the tree starting from the root to the leaf (terminal). The arrows indicates the branches of the tree that have been selected by the process for this particular example. RUGRAT keeps the immutable part (i.e., select, from, desc, etc.) of the SQL statements as it is and randomly generates the values for the mutable part (i.e., SelectExp, TableExp, ChooseVar, etc.). We explain the immutable and mutable parts of SQL statements in 2.2.1.

When there are multiple rules available for a non-terminal, RUGRAT randomly chooses one that satisfies the overall program configuration. For example, if the limit on the maximum number of select statements is reached, RUGRAT skips the SELECT rule in further statement generation.
At first glance such a blind random generation process may seem simplistic. However, SQLs contain many complex features that impose additional well-formedness rules on generated statements. It is therefore more challenging to generate syntactically correct statements, especially if we want the generated statements to have a wide variety of complex SQL features. Our goal is to let the user choose the complexity of the generated statements as well as the mix of SQL features the generated programs should be using.

As we want to generate benchmark statements, an important additional constraint is that RUGRAT-generated SQL statements should resemble real-world SQL relatively closely.

There are certain limitations in our current RUGRAT implementation. For instance, RUGRAT currently only supports frequently used types of SQL statements. To extend
RUGRAT by including new grammar rules for a diverse set of SQL statements are future work.

4.3.1 Configuration Options

RUGRAT is highly configurable. Users can specify various requirements as parameters in a configuration file and RUGRAT-generated statements satisfy the constraints given as requirements. Some of the important parameters are *NumOfTables, NumOfQueries, NumOfTransactions, DataTypes, joinQuery.Types, PercentageOfkeys, Nesting.Maximum* which are respectively used to specify the number of tables, number of queries, number of transactions, list of data types that could be present within the statements, allowed types of join operations, percentage of attributes that should be specified as the primary key of a table and, the maximum level of nesting allowed in a statements generated by RUGRAT. We show an example of a configuration file in Figure 10. Most of these parameters have a default value. More-

```plaintext
# Mon Sep 23 20:12:05 CDT 2013
dbConnPassword=""
NestingLevel=0
flagInsert Nested=0
insertQ=yes
MaxAttributes=5
PercentageOfkeys=10
NumOfTransactions=5
currentTransaction=0
flagUpdate Nested=0
updateQ=yes
flag Nested=0
dbConnName=""
TablesInS.Maximum=2
simpleOperators.String="|
ExecutesINDB.flag=false
dbConnUrl=""
Nesting.Maximum=3
dbConnUsername=""
MinAttributes=5
dbConnDriver=""
selectQ=yes
Update.NeededSelectQ=0
NumOfTables=50
simpleOperators.Numerals=\!,\=,<,\>,\>,\<
DataTypes=int,float,char,varchar
ResourcesInvolvedInCycle=5
query.Types=select,update,insert
joinQuery.Types=join,left join,right join
NumOfQueries=10
onExprOperator.Types=\!,\=,<,\>,\<,\>,\>,\<,\>||
```

Figure 10. Example of a configuration file.
over, some parameters are interdependent, e.g., ‘ResourcesInvolvedInCycle’, which is used to specify the size of a hold-and-wait cycle (see Section 4.4). One of the most important features of RUGRAT is allowing users to specify the database mode. In this mode, an instance of database is created with all tables and all generated statements are automatically executed against that database. This mode is activated by setting the flag variable ‘ExecuteINDB.flag’.

4.3.2 Example of Generated Statements

We show an example of RUGRAT-generated SQL statements in Figure 11 that contains twelve instances of select, insert and update statements.

```sql
1 update T_45 set T_45_P_4='u7k5tkuuzu494' , T_45_P_0=6 , T_45_P_1='bkg413ztzw2x'
2 insert into T_38 (T_38_P_1,T_38_P_0) values ('1f9sangpqa327','1kypgy3rmujo')
3 insert into T_1 (T_1_P_1,T_1_P_4) values ('4hdhutdtn10ba',5)
4 select T_3_P_4 from T_1 left join T_3 on T_1_P_0 < T_3_P_0 where T_1_P_4 <= 3 and T_1_P_0 > 3 or T_3_P_1 >= 2 and T_1_P_1 <= 9 and T_3_P_2 != 2 and T_2_P_1 < 2
5 select T_10_P_1 from T_10,T_49 where T_49_P_3 || '1ueskum4dxxf' or T_10_P_3 || 'dwhpb2ser3fu' and T_10_P_0 || 'ekuni5oi7e90'
6 select T_6_P_4 from T_19,T_6
7 insert into T_31 (T_31_P_1,T_31_P_2) values (9,1)
8 select T_30_P_2 from T_30,T_9 where T_9_P_0 >= 0 and T_30_P_4 < 5 and T_30_P_3 >= 7 and T_30_P_0 = 6 or T_9_P_4 = 4 or T_30_P_2 < 0 or T_9_P_3 = 10 order by T_30_P_2 desc
9 update T_32 set T_32_P_2=9 , T_32_P_0='iknnttznw2dh' , T_32_P_4=8 , T_32_P_3=4 where T_32_P_4 <= 6 and T_32_P_3 <= 3
10 select T_1_P_0 from T_3 right join T_2 on T_3_P_0 >= T_2_P_0
11 select T_1_P_4 from T_2 left join T_3 on T_2_P_0 <= T_3_P_0 order by T_1_P_4 desc
12 insert into T_44 (T_44_P_2,T_44_P_4) values ('18ez9oivqrm2','1cwr6grghksoi')
```

Figure 11. Example of RUGRAT-generated SQL statements.
The generated statements are labeled 1 to 12 and shown in the leftmost column of the Figure 11. The format of a table name is $T_i$ and a property (attribute) name is $T_i \cdot P_j$, where $i$ and $j$ are integers and the name of an attribute is prefixed with the table name that it belongs to. As it is shown in the figure, RUGRAT generates statements with various structures. For example, it generates update statements without a where clause (statement 1) and with a where clause (statement 9), generates select statements that include cartesian product operations (statements 5, 6 and 8), left join operations (statements 4 and 11), right join operation (statement 10), order by clauses (statements 8 and 11), and where clause (statement 4), generates insert statements that store integer type values (statement 7), string type values (statements 2 and 12), and both integer and string type values in a single statement (statement 3).

4.4 Seeding Hold-and-wait Cycles

We use RUGRAT for the purpose of random transactions (SQL statements) generation. But, there may not exist any hold-and-wait cycles in randomly generated transactions, especially when the transactions contain very few SQL statements. However, we can ensure the presence of hold-and-wait cycles in transactions by setting corresponding configuration parameter and we call this process seeding hold-and-wait cycles. We define hold-and-wait cycle in Section 5.1 and describe our seeding process in the following.

Let us assume that we have a set $T$ of $n$ transactions, and a set $R$ of $m$ resources where $T = T_1, T_2, T_3, \ldots, T_n$ and $R = R_1, R_2, R_3, \ldots, R_m$. Let us also define $s$, the size of
a cycle, as the number of resources or equivalently the number of transactions associated with that cycle.

RUGRAT starts with randomly choosing $s$ distinct transactions ($T_{r_1}, T_{r_2}, T_{r_3}, \ldots, T_{r_s}$) and $s$ distinct resources ($R_{r_1}, R_{r_2}, R_{r_3}, \ldots, R_{r_s}$) from the set $T$ of transactions and the set $R$ of resources respectively, where $r_i$ is the $i$-th random number and an $r_i$ for transaction is different from an $r_i$ for resource, since they are generated separately for transactions and resources.

While randomly chosen transactions and resources are available, a hold-and-wait cycle of the following order is formed: $T_{r_1} \leftrightarrow R_{r_1} \leftrightarrow T_{r_2} \leftrightarrow R_{r_2} \leftrightarrow T_{r_3} \leftrightarrow R_{r_3} \leftrightarrow \ldots \ldots \ldots \leftrightarrow T_{r_{s-1}} \leftrightarrow R_{r_{s-1}} \leftrightarrow T_{r_s} \leftrightarrow R_{r_s} \leftrightarrow T_{r_1}$, where a transaction $T_{r_i}$ is holding the resource $R_{r_i}$ and waiting for the resource $R_{r_{i-1}}$. Here, every resource is accessed by two transactions and every transaction accesses two resources in the same way as it is described in the formation of a hold-and-wait cycle in Section 5.1. Once the artificial cycle is defined, RUGRAT inserts statements in transactions and those statements access the resources in such way that the artificially defined hold-and-wait cycle in materialized in those transactions. In this way, RUGRAT seeds the cycle in transactions.

In Figure 12, we give an example of a hold-and-wait cycle and RUGRAT-generated statements to explain how that cycle is formed through accessing common resources (tables) by the statements in different transactions.

The cycle is shown in the beginning of Figure 12. Next, for each transaction involved in the cycle, the query (statement) which is holding a resource (table) and the query which
Figure 12. Seeding a Hold-and-wait cycle

is waiting on a resource (table) are reported. In this cycle, transaction $T_1$ is waiting for resource (table) 49 that is held by transaction $T_3$, transaction $T_3$ is waiting for resource (table) 40 that is held by transaction $T_4$, and so on. The final transaction in the cycle is the first transaction $T_1$, which means $T_1$ is holding resource (table) 48 that the transaction $T_2$ is waiting on. In this way, each transaction is holding a resource and waiting on another resource in a circular fashion and forms the hold-and-wait cycle.
In this chapter, we present our hold-and-wait cycles detection algorithm. We also give some backgrounds on Petri nets as we model database transactions using Petri nets before we use them as input to the cycle detection algorithm. We also describe our abstraction, discuss the complexity and the completeness of the cycles detection algorithm in this chapter.

5.1 Hold-and-wait Cycle

A hold-and-wait cycle is formed when there is a set of waiting transactions, $T = T_1, T_2, ..., T_s$, such that $T_1$ is waiting for a resource held by $T_2$, $T_2$ is waiting for a resource held by $T_3$ and so on until $T_s$ is waiting for a resource held by $T_1$.

For example, $T_1 \hookrightarrow R_1 \hookrightarrow T_2 \hookrightarrow R_2 \hookrightarrow T_3 \hookrightarrow R_3 \hookrightarrow \ldots \hookrightarrow T_{(s-1)} \hookrightarrow R_{(s-1)} \hookrightarrow T_s \hookrightarrow R_s \hookrightarrow T_1$ is a hold-and-wait cycle, where the '$\hookrightarrow$' in the direction of transaction $T_i$ to resource $R_i$ means transaction $T_i$ is holding the resource $R_i$ and the symbol '$\hookrightarrow$' in the direction of resource $R_{(i-1)}$ to transaction $T_i$ means transaction $T_i$ is waiting for the resource $R_{(i-1)}$, for $i = 1$ to $s$ and by considering $R_0 = R_s$ and $T_{(s+1)} = T_1$.

Here, $s$ is the size of a cycle and defines the number of resources or equivalently the number of transactions associated with that cycle. Every resource is accessed by two transactions and every transaction accesses two resources. For example, resource $R_2$ is accessed...
by both $T_2$ and $T_3$ and transaction $T_2$ accesses resources $R_1$ and $R_2$. In general, a resource $R_i$ is accessed by $T_i$ and $T_{(i+1)}$ and a transaction $T_i$ accesses two resources $R_i$ and $R_{(i-1)}$.

5.2 Modeling Transactions

Modeling of transactions is based on our abstraction. In this section, we describe the concept of that abstraction followed by some background on Petri nets that we used for the purpose of modeling.

5.2.1 Our Abstraction

As an abstraction we 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. This abstraction unifies DCAs that share 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.

5.2.2 Background on Petri Nets

Ordinary Petri nets are directed graphs with two kinds of nodes called places and transitions (Wang et al., 2009). Figure 13 shows an example of a Petri net modeling the transactions appearing in Table I. For convenience, we show Table I here again as Table III.

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 $p_{1\text{ init}}$ in Figure 13 is assigned one token. Arcs connect transitions to
TABLE III

EXAMPLE OF A DATABASE DEADLOCK THAT MAY OCCUR WHEN TWO TRANSACTIONS T_1 AND T_2 ARE ISSUED BY DCA(S).

<table>
<thead>
<tr>
<th>Step</th>
<th>Transaction T_1</th>
<th>Transaction T_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UPDATE authors SET citations=100 WHERE paperid=1</td>
<td>UPDATE titles SET copyright=1 WHERE titleid=2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>UPDATE titles SET copyright=1 WHERE titleid=2</td>
</tr>
<tr>
<td>3</td>
<td>SELECT title, doi FROM titles WHERE titleid=2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>SELECT authorname FROM authors WHERE paperid=1</td>
</tr>
</tbody>
</table>

places and vice versa. A directed arc from a place to a transition represents preconditions that are required for an event 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 t_1.ready fires, the token in place p_1.init is removed and a token is added to place p_2, enabling transition t_2.write, since both its input places p_2 and R_1.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 $p^* = \{t \mid (p, t) \in A\}$. The sets of input and output places for a transition $t$ are similarly defined as $^t = \{p \mid (p, t) \in A\}$ and $t^*$.

5.2.3 Modeling Using Petri Nets

We model database transactions using a subclass of Petri nets called $S^4 R$ nets (Tricas et al., 1999), 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. The special structure of $S^4R$ nets has allowed us to build an efficient algorithm for detecting potential deadlocks in DCAs. Figure 13 is essentially the $S^4R$ net representation of the transactions in Table III.

Resource places $R_{1\text{.authors}}$ and $R_{2\text{.titles}}$ 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 5.2.1. We extracted the SQL parser from Apache Derby database to parse the SQL statements. INSERT and UPDATE are modeled using the operation write, SELECT statements are modeled using the operation read, as we mentioned in Section 2.2. The operation release designated in Figure 13 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.

5.2.4 Benefits of Using Petri Nets

Using Petri nets for modeling, it is possible to generate firing sequences of transitions that enable testers and developers to understand, analyze, and debug database deadlocks, which is one of the goals of this dissertation. 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 $t_1$\textunderscore ready fires in the model that is shown in Figure 13, a token is placed into place $p_2$. Since the token is still in the resource place $R_1$\textunderscore authors, transition $t_2$\textunderscore write is enabled, which corresponds to the execution of the SQL statement \texttt{UPDATE} for $T_1$ in Table III. Since the token is taken from place $R_1$\textunderscore authors, transition $t_7$\textunderscore read is no longer enabled. At the same way, when token is taken from place $R_2$\textunderscore titles after firing $t_6$\textunderscore write which corresponds to the execution of the SQL statement \texttt{UPDATE} for $T_2$ in Table III, transition $t_3$\textunderscore read is no longer enabled. Hence, the hold-and-wait cycle can be reached by the following transition firing sequence: $t_1$\textunderscore ready $\rightarrow$ $t_5$\textunderscore ready $\rightarrow$ $t_6$\textunderscore write $\rightarrow$ $t_2$\textunderscore write.

Our cycle detection algorithm can report this sequence as it models the transactions using Petri nets.

A slightly simpler modeling formalism is based on lock graphs, an example of which is shown in Figure 1. We chose the Petri net formalism over lock graphs since the latter have two main disadvantages. First, unlike Petri nets, lock graphs do not contain any concept of transition, which is important to understand the behavior that leads to deadlocks rather than only observing the hold-and-wait cycle. Information of transitions is required to present debugging information to developers who can understand how database deadlocks occur, and lock graphs do not provide this debugging information. Second, lock graphs contain strongly connected components for complicated transactions, especially nested and multi-level ones with complex looping behavior (e.g., correlated SQL queries), and analyzing lock graphs with strongly connected components can lead to significant computational complexity for 2-satisfiability problem (Aspvall et al., 1979).
5.3 Cycles Detection Algorithm

In this section, we describe the cycles detection algorithm step by step and explain the input and output of the algorithm followed by a proof of the completeness of the algorithm.

![Figure 14. Overview of the Cycle Detection Algorithm.](image)

**Input:** We model the transactions using Petri nets and represent them in *Petri Net Markup Language (PNML)*. PNML is an XML-based syntax for Petri nets, which is used as a standard interchange format for various Petri net tools. A PNML file, corresponding to a set of transactions, is used as input to the cycle detection algorithm as it is shown in Figure 14. For the purpose of running experiments, we generate the transactions using our tool RUGRAT that we described in Chapter 4 and shown within the first rectangle of Figure 14.

**Output:** The output of the algorithm is the list of detected hold-and-wait cycles that are found by the algorithm, for the given input. Figure 15 shows the output of the algorithm when the PNML file, corresponds to the transactions $T_1$ and $T_2$ appearing in Table III, is provided as input. As it is shown in the figure, at first, the algorithm reports the number of
1 cycle(s) detected.

Cycle #1:

T1
Transition: 1_1 Holding resource: AUTHORS
Transition: 1_2 Waiting on resource: TITLES

T2
Transition: 2_1 Holding resource: TITLES
Transition: 2_2 waiting on resource: AUTHORS

Figure 15. Output of the Cycle Detection Algorithm.

detected hold-and-wait cycles. Then, it sequentially reports information on each cycle. A cycle information contains a list of transactions that are involved in the cycle and for each transaction, it shows the transitions (SQL statements) of the transaction that are holding a resource and waiting on a resource. A transition can be identified by its id that is self-descriptive. There are two parts of an id, separated by a ‘.’. The first part represents the transaction id that it belongs to and the second part represents the SQL statement id that the transition corresponds to. For example, Transition: 2_1 represents statement 1 of transaction 2.

**Description:** 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 $\mathcal{N}$. Recall that, we use process subnets to model database transactions.
Algorithm 1 The Cycles Detection 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, ∃a ∈ {A} | a = (p, t) 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) → t
30:     if u ∈ 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, GetSubnet(e))
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
Lines 5–18 iterate the following actions on each subnet where each subnet is corresponding to some transaction. First, all transitions in the subnet are pushed on the stack (Line 6). Next, a transition $t$ is popped from the 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. In case of a conflict, $t$ is added to the potential cycle being explored, and procedure $\text{ComputeAllCycles}$ 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 $\text{ComputeAllCycles}$. 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, unless the predecessor of $t$ is the initial place of the net that $t$ belongs to (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 a predecessor of $u$ in $u$’s subnet. In this case, a cycle 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, \( V \), 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 \texttt{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.

The \textit{hold-and-wait} cycles detected by the algorithm correspond to transition firings leading to those \textit{hold-and-wait} cycles. In general, a \textit{hold-and-wait} cycle consists of sequences of consecutive transitions in different process subnets contained in an \( S^4R \) net. We call these sequences \textit{hold-and-wait sections}. In Figure 16, we show the \textit{hold-and-wait} cycle (with colored and dashed arrows) of the \( S^4R \) net that models the transactions of our example in Table III. We also show the \textit{hold-and-wait sections} (two of them, in this case) within the cycle in red color.

\textit{Hold-and-wait sections} are connected to each other in such a way that the first transition (i.e., the earliest transition to fire) in one such sequence is connected to the last transition (i.e., the latest transition to fire) in the next hold-and-wait section because the two transitions share a resource place as a common input. Moreover, the last hold-and-wait section is again connected to the first detected hold-and-wait section in a similar fashion. Evidently, it is impossible to fire all transitions in the hold-and-wait sections contained in a detected cycle as the last transition to be fired will miss a needed resource, while holding a resource needed by the next process subnet in the cycle. This pattern will be repeated among all
Figure 16. Hold-and-wait cycle and hold-and-wait sections.

hold-and-wait sections in circular fashion, meaning that all involved process subnets will be unable to proceed.

5.3.1 Completeness of the algorithm

In this section, we define a cycle in the context of a Petri net $\mathcal{N}$ and prove the completeness of the Algorithm 1. For that purpose, we define the following predicates:

$R(x)$, $x$ is a resource place.

$P(x, y)$, there is a path in Petri net $\mathcal{N}$ from transition $x$ to transition $y$.

$S(x, s)$, transition $x$ belongs to hold-and-wait section $s$ of $\mathcal{N}$.

$D(x)$, Algorithm 1 detects element $x$ as a part of a cycle.

$H(x)$, place $x$ holds a token.
Definition 2. A cycle $C$ in a Petri net $N = (P, T, A, M_0)$ is a sequence of transitions and places given by $C = t_1^w t_1^h R_1 t_2^w t_2^h R_2 t_3^w t_3^h R_3 ... t_n^w t_n^h R_n t_{n+1}^w$, where $n \leq$ total number of transactions (subnets) represented by $N$, and $t_{n+1}^w = t_1^w$, such that

$$\forall i = 1, 2, 3, ..., n; \ [(R_i \in P) \land R(R_i) \land (t_i^w, t_i^h) \in T) \land ((R_i, t_i^h) \in A) \land ((R_i, t_i^w) \in A) \land (S(t_i^w, i) \iff S(t_i^h, i) \land P(t_i^h, t_i^w) \land \neg P(t_i^w, t_i^h))$$

$$\forall i, j = 1, 2, 3, ..., n; [R_i \neq R_j]$$

Theorem 1 (Completeness). All hold-and-wait cycles in transactions are detected by Algorithm 1.

Proof. Let’s assume that there is a cycle $C$ in a Petri net $N$ which is not detected by the Algorithm 1, i.e., there is a sequence $C' \subseteq C$, which is not explored by the algorithm. But, in the following, we prove that this is not possible.

Initial Step:

If $S$ is the set of subnets in $N$, we can say from the Algorithm 1, $(\forall s \in S \ (\forall t \in s \ (\forall p \in P | R(p) \land (\exists a \in A | a = (p, t))))) \implies D(t)$, which ensures that the element $t_i^h$ of all cycles are always detected by the algorithm, i.e., $C' \subseteq C$.

If the length of $C'$ is 1 and $C'$ is not detected by the Algorithm 1, then there are three possibilities:

Case 1: $\exists i \mid ((t_i^h, R_i \subseteq C) \land D(t_i^h) \land \neg D(R_i))$

Case 2: $\exists i \mid ((R_i, t_i^w \subseteq C) \land D(R_i) \land \neg D(t_i^w))$

Case 3: $\exists i \mid ((t_i^w, t_i^h \subseteq C) \land D(t_i^w) \land \neg D(t_i^h))$
In case 1, we can say from the Algorithm 1, \((\forall q \in \bullet u \mid D(u) \land H(q) \land R(q)) \implies D(q)\). Since, \((t_i^h, R_i \subseteq C) \implies (R_i \in \bullet t_i^h \land R(R_i) \land H(R_i)))\). Therefore, \(D(R_i)\)

In case 2, we can say from the Algorithm 1, \((\exists p \in P \mid R(p) \land \exists a \in A \mid a = (p, t)) \implies D(t)\). Since, \(((R_i, t_{i+1}^w) \subseteq C) \implies (R(R_i) \land (R_i, t_{i+1}^w \in A))\). Therefore, \(D(t_{i+1}^w)\).

In case 3, we can say from the Algorithm 1, \((\forall (P(t, t_i^w) \land D(t_i^w)) \implies D(t))\). Since, \(((t_i^w, t_i^h) \subseteq C) \implies P(t_i^h, t_i^w)\). Therefore, \(D(t_i^w)\).

**Inductive Steps:**

Let’s assume that the the sequence \(C_i \subseteq C\) with length \(i\) is detected by the Algorithm 1. Now, the sequence \(C_{i+1} \subseteq C\) with length \(i + 1\) is nothing but an extension of \(C_i\) with one more element in the same three possible ways shown in the initial step, where \(t_i^h\) (case 1), \(R_i\) (case 2) and \(t_i^w\) (case 3) are three last elements of \(C_i\). Therefore, \(C_{i+1}\) is also explored by the Algorithm 1. Hence, cycle \(C\) in Petri net \(N\) is detected by the Algorithm 1.

\(\square\)

### 5.3.2 Complexity of the algorithm

A brute-force approach for finding hold-and-wait cycles does not work. For \(N\) transactions, there are \(2^N - N - 1\) combinations of transactions in total (the power set of \(N\) transactions, \(2^N\) and excluding the empty set and \(N\) sets with only a single transaction) that should be explored.

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 average number of of transitions in each subnets, i.e., no cycles are possible. In the worst case, all subsets of subnets have cycles, and the complexity of our algorithm 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 Chapter 7.

5.4 Cycle Detection GUI Tool

We implement a simulator (hold-and-wait cycle detection tool) and the corresponding GUIs are shown in Figure 17. On the left side of the figure, the GUI shows a list of transactions, one in each list box with SQL statements that constitute the transaction. On the right, a detected hold-and-wait cycle is visualized as a graph.

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 the left part of Figure 17.

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 is returned to the simulator, the user can visualize the detected cycles by clicking on the Result panel of the simulator as it is shown in the right part of Figure 17. Visualization of the cycles includes information about the associated transactions that form those cycles.
5.5 Evaluation of Cycles Detection Algorithm

We evaluate the cycle detection algorithm with various inputs and compared it with Integrated Net Analyzer (INA) \(^1\). INA is a tool package that supports various analyses of Petri net; many of the algorithms implemented in INA are described in greater detail in (Starke, 1990). We report the experimental results and the comparison with INA in Chapter 7.

\(^1\)http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/
CHAPTER 6

PREDICTING AND PREVENTING DATABASE DEADLOCKS

We address the problems of database deadlocks by implementing our solution predicting and preventing database deadlocks \((PD)^2\). There are two major parts in \((PD)^2\): *Systematic Testing in Presence of Database Deadlocks (STEPDAD)* for the purpose of testing for database deadlocks in DCAs and *pReventing database Deadlocks from Application-based Transactions (REDACT)* to prevent the occurrences of database deadlocks in DCAs. In this chapter, we describe our techniques that we use in STEPDAD and REDACT.

6.1 Our Key Ideas

In this section, we describe our key ideas of STEPDAD and REDACT. In both, we first use the *cycles detection algorithm* to statically detect all hold-and-wait cycles that belong to different transactions with specific execution scenarios that lead to these hold-and-wait cycles. Then, we use the information about hold-and-wait cycles to implement our key ideas.

6.1.1 Ideas for STEPDAD

Our solution rests on a key idea that transactions involved hold-and-wait cycles should be executed simultaneously in order to increase the probability that a deadlock will occur. We specifically introduce the mechanism of *execution hijacking (EH)* (Tsankov et al., 2011) for scheduling executions of DCAs in such a way that these transactions will be issued in
close temporal proximity. A rationale for this idea is that database schedulers are more likely to create interleavings of instructions that realize hold-and-wait cycles if transactions arrive at the same time. This idea is related to work on producing scheduling that causes concurrent programs to fail (Ben-Asher et al., 2006). Of course, there is no guarantee that the simultaneous arrival of transactions may result in a database deadlock—this is a hypothesis that we evaluate and report in Chapter 7.

Another idea is to replicate transactions that have hold-and-wait cycles by issuing them simultaneously from different client applications to further increase the probability of observing database deadlocks. Consider the motivating example that is shown in Table I. By adding two or more replicated transactions $T_1$ and $T_2$ we can increase the “density” of SQL statements that contain hold-and-wait cycles per time unit of processing. Our hypothesis is that by increasing the number of replicated transactions that contain hold-and-wait cycles and that are sent to the database simultaneously, we can increase the probability of occurrence of database deadlocks. We evaluate this hypothesis and report the results of our evaluation in Chapter 7.

6.1.2 Ideas for REDACT

The key idea in REDACT is 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.

6.2 Architecture of \((PD)^2\)

The architecture of \((PD)^2\) is shown in Figure 18. Both of the implementations of STEPDAD and REDACT are based on this architecture. But, they differ in Supervisory Control (SC) that we describe in a separate section. This architecture 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, DCAs communicate with databases using connector APIs (see Section 2.1) and database deadlocks occur at some rate.

The first step in the workflow of the architecture of \((PD)^2\) involves extracting transactions that contain SQL statements from these DCAs as shown with dashed arrows labeled with (2) (we use the terms transaction and SQL statement interchangeably). This is a one-time manual effort that we discussed in section 2.2.

In \((PD)^2\), we perform the analysis in two phases: static phase and dynamic phase. 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 input into the Modeler that automatically transforms SQL statements into the abstract operations. Specific details of modeling are described in Section 5.2. The Modeler (5) uses database settings that include a locking strategy (6) to produce a Petri Net model. This model (7) serves as the input to the cycle detection 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 \((PD)^2\).

At this point, we describe the dynamic phase during which DCAs are run and SQL
statements are divert 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
possible database deadlock. Diverting SQL statements is accomplished in \((PD)^2\) by using
the \textit{interceptor pattern} that is implemented using a framework with call-backs associated
with particular events (Schmidt et al., 2000). In \((PD)^2\), we use AspectJ\footnote{http://www.eclipse.org/aspectj, verified Nov 19, 2013.} to instrument

---

Figure 18. Architecture of \((PD)^2\).
subject DCAs and intercept SQL statements. We show these interceptors as partial rectangles with the label \texttt{Int} that are superimposed on the rectangles that designate DCAs in Figure 18. 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 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, the SC utilizes the information that is obtained from the static analysis phase as well as (10) its settings that enable testers and developers 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 for REDACT given in Chapter 7. 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 false positive cycles. 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. SC forwards (12) these SQL statements to the database for execution, depending on the strategies that we describe in Section 6.3.

6.3 Supervisory Control

In STEPDAD, first, SC checks whether executing SQL statements, that are ready to be executed, may result in the creation of hold-and-wait cycles and therefore deadlocks. Then,
SC forwards these SQL statements to the database for execution if they may form hold-and-cycles. Otherwise, SC determines what other SQL statements have hold-and-wait cycles with the pending SQL statements, for example, from DCA\textsubscript{n}. Then, it starts the mechanism of execution hijacking that forces DCA\textsubscript{n} to execute the statement containing the JDBC API call with the conflicting SQL statement. Once both SQL statements are pending at the SC, the SC forwards these SQL statements to the database for execution, thus increasing the probability that the database deadlock will actually occur. To implement our another idea of replicating transactions, SC creates mock client applications that send transactions, that are required to form cycles, to the database. In this case, deadlock exceptions could be thrown either into the mock applications or into the DCAs, depending on the transactions that are chosen as victim transactions and rolled back by the database engine.

Like in STEPDAD, in REDACT, at first, SC looks up for the presence of hold-and-wait cycles in SQL statements that are currently ready to be executed. If SC finds no hold-and-wait cycles during the look up, it forwards the 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 deadlocks.
CHAPTER 7

EMPIRICAL DATA AND EVALUATION

In this chapter, we present all of our experimental results, evaluate our approaches and compare it with other approaches. We also provide the research questions that we answer through these experiments. We conduct all experiments on Amazon EC2 \(^1\) with 2.7 GHz Intel Core i7 processor and 4 GB memory.

7.1 Evaluation of Cycles Detection Algorithm.

By implementing the cycles detection algorithm, we seek to answer the following research question.

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

To address RQ1, we run our cycles detection algorithm with different number of transactions that contain different numbers of SQL statements that use different database resources (e.g., tables). We use our tool RUGRAT to generate random transactions and use them to evaluate the algorithm.

The results of our experiments are shown in Table IV for various number of transactions \((T)\) and SQL statements per transactions \((S)\). Column \(\%X\) represents the percentage of

\(^{1}\text{http://aws.amazon.com/ec2/}\)
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 detected is reported in Cycles\textsubscript{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.

**TABLE IV. RESULTS OF EXPERIMENTS ON CYCLES DETECTION ALGORITHM.**

<table>
<thead>
<tr>
<th>T</th>
<th>S</th>
<th>%C</th>
<th>Places</th>
<th>Transitions</th>
<th>Arcs</th>
<th>Resources</th>
<th>Cycles\textsubscript{det}</th>
<th>Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>50</td>
<td>10</td>
<td>1960</td>
<td>1000</td>
<td>4214</td>
<td>960</td>
<td>184</td>
<td>1.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>1958</td>
<td>1000</td>
<td>4260</td>
<td>958</td>
<td>846</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>1952</td>
<td>1000</td>
<td>4282</td>
<td>952</td>
<td>6477</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>10</td>
<td>3997</td>
<td>2020</td>
<td>8600</td>
<td>1977</td>
<td>445</td>
<td>3.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>3974</td>
<td>2020</td>
<td>8730</td>
<td>1954</td>
<td>17760</td>
<td>4.19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>3955</td>
<td>2020</td>
<td>8756</td>
<td>1935</td>
<td>650667</td>
<td>12.34</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>10</td>
<td>7915</td>
<td>4000</td>
<td>17296</td>
<td>3915</td>
<td>5035</td>
<td>12.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>7922</td>
<td>4000</td>
<td>17380</td>
<td>3922</td>
<td>131099</td>
<td>15.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>7922</td>
<td>4000</td>
<td>17394</td>
<td>3922</td>
<td>5022808</td>
<td>89.09</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>10</td>
<td>2440</td>
<td>1250</td>
<td>5304</td>
<td>1190</td>
<td>426</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>2446</td>
<td>1250</td>
<td>5356</td>
<td>1196</td>
<td>1039</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>2426</td>
<td>1250</td>
<td>5376</td>
<td>1176</td>
<td>1276048</td>
<td>23.65</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>10</td>
<td>4991</td>
<td>2525</td>
<td>10910</td>
<td>2466</td>
<td>1035</td>
<td>4.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>4990</td>
<td>2525</td>
<td>10858</td>
<td>2465</td>
<td>18335</td>
<td>5.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>4967</td>
<td>2525</td>
<td>10976</td>
<td>2442</td>
<td>5453679</td>
<td>71.79</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>10</td>
<td>9922</td>
<td>5000</td>
<td>21580</td>
<td>4922</td>
<td>6709</td>
<td>18.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>9922</td>
<td>5000</td>
<td>21674</td>
<td>4922</td>
<td>106522</td>
<td>21.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>9905</td>
<td>5000</td>
<td>21764</td>
<td>4905</td>
<td>98016675</td>
<td>1951.01</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>4</td>
<td>1001</td>
<td>550</td>
<td>2154</td>
<td>451</td>
<td>3</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>1005</td>
<td>551</td>
<td>2141</td>
<td>454</td>
<td>3</td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>1000</td>
<td>550</td>
<td>2176</td>
<td>450</td>
<td>18</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>4</td>
<td>2495</td>
<td>1300</td>
<td>5394</td>
<td>1195</td>
<td>87</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>2497</td>
<td>1300</td>
<td>5386</td>
<td>1197</td>
<td>106</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>2493</td>
<td>1300</td>
<td>5410</td>
<td>1193</td>
<td>764</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>4</td>
<td>4890</td>
<td>2500</td>
<td>10600</td>
<td>2390</td>
<td>317</td>
<td>4.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>4883</td>
<td>2500</td>
<td>10588</td>
<td>2383</td>
<td>5873</td>
<td>4.95</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>4886</td>
<td>2500</td>
<td>10624</td>
<td>2386</td>
<td>17002</td>
<td>5.48</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>4</td>
<td>1999</td>
<td>1100</td>
<td>4308</td>
<td>899</td>
<td>10</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>2003</td>
<td>1103</td>
<td>4336</td>
<td>900</td>
<td>99</td>
<td>1.27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>2001</td>
<td>1101</td>
<td>4348</td>
<td>900</td>
<td>192</td>
<td>1.59</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>4</td>
<td>4995</td>
<td>2600</td>
<td>10856</td>
<td>2385</td>
<td>660</td>
<td>4.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>4993</td>
<td>2600</td>
<td>10872</td>
<td>2393</td>
<td>57636</td>
<td>7.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>4998</td>
<td>2600</td>
<td>10866</td>
<td>2388</td>
<td>669185</td>
<td>22.91</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>4</td>
<td>9729</td>
<td>5000</td>
<td>21260</td>
<td>4789</td>
<td>13996</td>
<td>19.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>9775</td>
<td>5000</td>
<td>21328</td>
<td>4775</td>
<td>8449320</td>
<td>184.93</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>9763</td>
<td>5000</td>
<td>21414</td>
<td>4763</td>
<td>947718814</td>
<td>23793.94</td>
</tr>
</tbody>
</table>
7.1.1 Comparison with INA algorithm

We compare our cycle detection algorithm with Integrated Net Analyzer (INA)\(^1\) which is a very useful and well-known tool package for editing, simulating and analyzing Petri nets and Coloured Petri nets. Details of the many of the algorithms implemented in INA are given in (Starke, 1990). The analysis part of the INA involves computing the structural information of the input nets, which is used to verify the deadlock-trap-property of the nets.

For the purpose of verifying this property, INA has to compute the list of all deadlocks which are minimal with respect to set inclusion and this deadlock detection part requires a considerable amount of computations as it is men-

\(^1\)http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/

<table>
<thead>
<tr>
<th>T</th>
<th>S</th>
<th>%C</th>
<th>CDA Time</th>
<th>INA Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>1.15</td>
<td>67.11</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>15</td>
<td>1.2</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>1.81</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>3.36</td>
<td>2794.73</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>15</td>
<td>4.19</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>12.34</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>10</td>
<td>12.18</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>15</td>
<td>15.07</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>89.09</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>1.61</td>
<td>80.54</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>15</td>
<td>1.7</td>
<td>3661.19</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>20</td>
<td>23.65</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>10</td>
<td>4.95</td>
<td>3780.33</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>15</td>
<td>5.58</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>20</td>
<td>71.79</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>10</td>
<td>18.67</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>15</td>
<td>21.29</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1951.01</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0.59</td>
<td>2.09</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>0.69</td>
<td>8.76</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>0.59</td>
<td>7.23</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>1.58</td>
<td>25.76</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>7</td>
<td>1.61</td>
<td>30.18</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>9</td>
<td>1.67</td>
<td>53040.66</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>7</td>
<td>4.95</td>
<td>396.11</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>9</td>
<td>5.48</td>
<td>44940.64</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>1.21</td>
<td>13.88</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>1.27</td>
<td>68.35</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>1.59</td>
<td>7080.45</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>4</td>
<td>4.96</td>
<td>20400.12</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>7</td>
<td>7.63</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>9</td>
<td>22.91</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>19.12</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>7</td>
<td>184.93</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>9</td>
<td>23793.94</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
tioned in the manual\(^1\) of INA. Our intension is to observe the time that the INA takes to analyse the input nets for the purpose of computing the deadlocks. We use the same input set that we use for the experiments of cycles detection algorithm and we appropriately convert this input set to run the experiments with INA.

We report the experimental results with INA in Table V for the same number of transactions \((T)\) and SQL statements per transactions \((S)\) that we report for the cycles detection algorithm in Table IV. Column \(\%X\) represents the percentage of transactions that are randomly chosen to be involved in cycles. The times required for cycles detection algorithm and the INA algorithm are reported in \textit{CDA Time} and \textit{INA Time} columns respectively. All times are in seconds and rounded to two decimal points. We ran each experiment for a maximum of 36 hours and we use dash instead of reporting time for the experiments that were not finished by that time.

### 7.1.2 Analyzing Experimental Results

In \textit{cycles detection algorithm}, for the number of transactions and SQL statements per transaction smaller than 50, the hold-and-wait cycles detection time is negligible 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

\(^1\)http://www2.informatik.hu-berlin.de/lehrstuehle/automaten/ina/manual.html
statements are chosen to be involved in cycles, our algorithm finds all cycles in less than two seconds.

On the other hand, with INA tool, half of the experiments were not finished their executions in 36 hours time frame. Not a single experiment was finished for 20 transactions and 25 transactions, each containing 200 statements, and 100 transactions each containing 50 statements. For other experiments, where INA finished its executions, the execution times were much higher than the execution times of cycles detection algorithm as it is shown in 7.1.1. For example, the only experiment that was finished with INA for 20 transactions, each containing 100 statements, took 2794.73 seconds (46.58 minutes), where the cycles detection algorithm took only 3.36 seconds. At the same way, for 100 transactions, each containing 25 statements, it took 20400.12 seconds (5.67 hours) with INA for the only experiment that finished its execution, where the cycles detection algorithm took 4.96 seconds. These results provide an answer to RQ1 that **our cycles detection algorithm efficiently detects hold-and-wait cycles in large-scale transactions.**

### 7.2 Subject DCAs for STEPDAD and REDACT

We evaluate STEPDAD and REDACT with three Java DCAs whose characteristics are shown in Table VI. The columns of the Table VI 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 in the database. HIM is a program for maintaining health information records. DAN is
TABLE VI

SUBJECT DCAS AND THEIR DATABASES.

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

a demographic analysis program. Finally, UCOM is a program for obtaining statistics on how users interact with Unix systems using their commands.

These DCAs were created by 32 graduate students (divided into three groups) from the University of Illinois at Chicago who wrote them as part of a graduate course of distributed object programming using simplified specifications from real-world applications that came from different projects at Accenture\(^1\). The subject DCAs as well as their databases are available from Sourceforge\(^2\). 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.

7.3 Evaluation of STEPDAD

In STEPDAD, we seek to answer the following research questions.

\(^1\)http://www.accenture.com/

**RQ2:** Can STEPDAD effectively reproduce database deadlocks for DCAs that issue transactions to the database?

**RQ3:** Is STEPDAD efficient in reproducing database deadlocks?

The rationale for both RQs is to compare STEPDAD with the baseline approach where multiple DCAs are run for a certain period of time until a database deadlock is registered or until a time period expires. Our goal is to show that STEPDAD is more effective and efficient than this baseline approach. We address RQ2 by measuring the number of deadlocks reported by STEPDAD vs. the baseline approach. The rationale for RQ3 is to determine the *mean time to database deadlock (MTTDD).* To address RQ3 we measure STEPDAD’s MTTDD and compare it with the baseline approach.

### 7.3.1 Methodology

We aligned our methodology with the guidelines for statistical tests to assess randomized approaches in software engineering (Arcuri and Briand, 2011). Since database deadlocks are not easy to reproduce, different runs of the DCA may reveal different number of deadlocks and different MTTDDs. 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. For each subject DCA, we ran each experiment 10 times with each approach to obtain a good representative sample.
1) **Independent Variables:** We have three independent variables: the subject DCA, the type of the experiment, and the number of DCA clients. There are three types of experiment: the Regular or baseline, where a subject DCA is run without STEPDAD, Controller-based, where the controller enables simultaneous delivery of transactions from subject DCA clients to the database for execution, and finally, Artificial injection, where artificial transactions are replicated to increase concurrency. For each subject DCA, we carried out experiments with two, four, and six clients for five minutes per experiment. In our experiments, we used JMeter [http://jmeter.apache.org](http://jmeter.apache.org) to run subject DCAs with different numbers of clients.

2) **Dependent Variables:** We have two dependent variables: the number of database deadlocks that are observed during the experiment and the running time of each subject DCA until at least one database deadlock is encountered. Since exhibiting database deadlocks requires specific interleavings of transactions, we repeated each experiment 10 times. We report statistical results (average, median, min, max, variance) for 10 runs for the number of observed deadlocks and the time taken to run the subject DCAs into these deadlocks.

Thus, the total number of experiments is equal to three DCAs \times three types (R,C,A) \times three client settings \times 10 times = 270 experiments.

### 7.3.1.1 Hypotheses

We introduce the following null and alternative hypotheses to evaluate how close the means are for MTTDD for different approaches. We seek to evaluate the following hypotheses at a 0.05 level of significance.
The primary null hypothesis is that there is no difference in the values of MTTDD between R, C, and A approaches for all subject DCAs.

An alternative hypothesis to \( H_0 \) is that there is statistically significant difference in the values of MTTDD between R, C, and A approaches for all subject DCAs.

Once we test the null hypothesis \( H_0 \), we are interested in the directionality of means, \( \mu \), of the results of control and treatment groups. We are interested to compare the effectiveness of STEPDAD versus the R and A approaches.

**H1 (MTTDD of R versus C).** The effective null hypothesis is that \( \mu_R = \mu_C \), while the true null hypothesis is that \( \mu_R \leq \mu_C \). Conversely, the alternative hypothesis is \( \mu_R > \mu_C \).

**H2 (MTTDD of R versus A).** The effective null hypothesis is that \( \mu_R = \mu_A \), while the true null hypothesis is that \( \mu_R \leq \mu_A \). Conversely, the alternative hypothesis is \( \mu_R > \mu_A \).

**H3 (MTTDD of C versus A).** The effective null hypothesis is that \( \mu_C = \mu_A \), while the true null hypothesis is that \( \mu_C \leq \mu_A \). Conversely, the alternative is \( \mu_C > \mu_A \).

The rationale behind the alternative hypotheses to \( H_1 \)–\( H_3 \) is that STEPDAD allows testers to quickly reproduce database deadlocks. These alternative hypotheses are motivated by our belief that by reducing the number of interleavings among transactions that contain hold-and-wait cycles in addition to increasing concurrency by replicating these transactions, they can result in reproducing database deadlocks much faster when compared to the baseline approach.
7.3.2 Analyzing Experimental Results

Experimental results on STEPDAD are summarized and shown in Table VII where each subject DCA (i.e., HIM, UCOM and DAN) is evaluated using (R)egular, (C)ontroller and (A)rtificial injection methodologies. Each DCA was run with 2, 4 and 6 clients for 5 mins per experiment. We measure the number of detected deadlocks in 5 mins of the experiment (in columndeadlocks) and also the time for the detection of the first deadlock within 5 mins (in columnTIME). For the columns Deadlock and Time, we report statistical results (MTTDD, median, min, max, variance) of 10 runs for each DCA/client setting. We use dash instead of reporting time for the experiments where no database deadlocks were reproduced.

We use one-way ANOVA and t-tests for paired two sample for means to evaluate the hypotheses that we stated in Section 7.3.1.

We use dash instead of reporting time for the experiments where no database deadlocks were reproduced. For DCAs HIM and UCOM with two clients for the regular baseline experiments, database deadlocks were not reproduced at all. Even for the subject DCA HIM with six concurrent clients database deadlocks were not reproduced in five out of ten regular baseline experiments. In contrast, with the artificial injection experiment, database deadlocks were reproduced in almost all experiments, except for two experiments with two clients for HIM, one experiment with four clients for HIM, and one experiment for two clients for UCOM.
TABLE VII. RESULTS OF EXPERIMENTS ON STEPDAD.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Clients</th>
<th>DCA</th>
<th>Deadlocks</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Avg</td>
<td>Med</td>
</tr>
<tr>
<td>HIM</td>
<td>R</td>
<td>2</td>
<td>0.6</td>
<td>0.5</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1.2</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>0.6</td>
<td>0.5</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.9</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>3.3</td>
<td>3.3</td>
<td>2.4</td>
<td>4.4</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>1.5</td>
<td>1.5</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.5</td>
<td>2.5</td>
<td>0.0</td>
<td>5.0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4.5</td>
<td>4.5</td>
<td>3.6</td>
<td>6.1</td>
</tr>
<tr>
<td>UCOM</td>
<td>R</td>
<td>2</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.2</td>
<td>0.0</td>
<td>0.1</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7.1</td>
<td>7.1</td>
<td>3.13</td>
<td>9.66</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1.3</td>
<td>1.3</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7.4</td>
<td>7.4</td>
<td>2.12</td>
<td>10.93</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>22.3</td>
<td>22.3</td>
<td>15.30</td>
<td>15.34</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>1.9</td>
<td>1.9</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10.9</td>
<td>10.9</td>
<td>6.15</td>
<td>7.43</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>22.3</td>
<td>22.3</td>
<td>15.30</td>
<td>15.34</td>
</tr>
<tr>
<td>DAN</td>
<td>R</td>
<td>2</td>
<td>0.9</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5.5</td>
<td>5.5</td>
<td>1.9</td>
<td>5.56</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>10.5</td>
<td>10.5</td>
<td>7.15</td>
<td>5.83</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1.9</td>
<td>1.9</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6.7</td>
<td>6.7</td>
<td>4.10</td>
<td>4.68</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>9.7</td>
<td>9.7</td>
<td>5.13</td>
<td>5.57</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>6.8</td>
<td>6.8</td>
<td>4.8</td>
<td>1.96</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11.3</td>
<td>11.3</td>
<td>7.16</td>
<td>7.34</td>
</tr>
</tbody>
</table>

In total, for the regular baseline approach, 20 out of 30 experiments did not result in any database deadlocks for HIM; 18 out of 30 experiments did not result in any database deadlocks for UCOM; and three out of 30 experiments did not result in any database deadlocks for DAN. For the Controller-based approach, only seven out of 30 experiments did not result in any database deadlocks for HIM; 12 out of 30 experiments did not result in any database deadlocks for UCOM; and all 30 experiments resulted in database deadlocks for DAN. Finally, for the artificially injected transactions approach, only three out of 30 experiments did not result in any database deadlocks for HIM; only one out of 30 experiments did not result in any database deadlocks for UCOM; and all 30 experiments resulted...
in database deadlocks for DAN. Based on these results, we can positively answer RQ1 that STEPDAD effectively reproduces at least one database deadlock.

7.3.2.1 Testing the Null Hypothesis

We used ANOVA to evaluate the null hypothesis $H_0$ that the variation in an experiment is no greater than that due to normal variation of DCAs’ characteristics and measurement errors given the probabilistic nature of reproducing database deadlocks. The results of ANOVA confirm that there are large differences between the groups in terms of MTTDD for HIM for two clients with $F = 5.64 > F_{crit} = 3.35$ with $p \approx 0.009$ which is strongly statistically significant. Similarly, the results of ANOVA for MTTDD for UCOM for two clients with $F = 35.6 > F_{crit} = 3.35$ with $p \approx 9.4 \cdot 10^{-9}$ which is strongly statistically significant. However, the results of ANOVA for MTTDD for DAN for two clients are inconclusive with $F = 1.2 < F_{crit} = 3.35$ with $p \approx 0.31$, while for four clients the differences are statistically significant with $F = 4 > F_{crit} = 3.35$ with $p \approx 0.03$. Based on these results we reject the null hypothesis and we accept the alternative hypothesis $H_1$.

7.3.2.2 Comparing Baseline with Controller

To test the null hypothesis $H_1$, we applied two t-tests for two paired sample means, in this case MTTDDs for the regular baseline and the controller approaches. Since we did not reproduce database deadlocks with the regular baseline approach for a number of experiments, we cannot run t-tests on these results. Instead, we statistically evaluated UCOM for six clients, and DAN for four and six clients. Statistical evaluation of UCOM is inconclusive, and for DAN for four clients, $t > t_{crit}$ as $2.25 > 1.85$ with $p \approx 0.025$ and for
six clients it is inconclusive with $MTTDD_R = 51.6 > MTTDD_C = 47.2$. Based on these results we reject the null hypothesis for a general case $H_1$ and we accept the alternative hypothesis that states that MTTDD for STEPDAD with Controller is lower than MTTDD for the regular baseline approach.

### 7.3.2.3 Comparing Baseline with Artificial

To test the null hypothesis $H_2$, we applied two t-tests for two paired sample means, in this case MTTDDs for the regular baseline and the artificial injection approaches. Since we did not reproduce database deadlocks with the regular baseline approach for a number of experiments, we cannot run t-tests on these results. Instead, we statistically evaluated UCOM for six clients, and DAN for four and six clients. Statistical evaluation of UCOM show that the artificial approach results in smaller MTTDD with $MTTDD_R = 57.5 > MTTDD_C = 32.2$ with $t > t_{crit}$ as $2.27 > 1.83$ with $p \approx 0.027$. For DAN for four clients, the result is strongly statistically significant and for six clients it is inconclusive. Based on these results we reject the null hypothesis for a general case $H_2$ and we accept the alternative hypothesis that states that MTTDD for STEPDAD with Artificial injector is lower than MTTDD for the regular baseline approach.

### 7.3.2.4 Comparing Controller with Artificial

To test the null hypothesis $H_3$, we applied two t-tests for two paired sample means, in this case $MTTDD$s for the artificial injector and the controller approaches. Based on these results we accept the null hypothesis $H_3$ that say that MTTDD for STEPDAD with Artificial injector is the same as MTTDD for the controller-based approach.
7.3.3 Discussion

One important conclusion from our experiments is that replication of database deadlocks is as effective with scheduling transactions using the controller as with artificial injection. Essentially, this is not surprising, since artificially injecting multiple transactions that hold-and-wait cycles is likely to result in a higher frequency of database deadlocks—more contention is created. Of course, database engine is a complex mechanism that parses SQL statements in transactions, translates them into low-level relational algebra operators, and creates their execution plans that is later carried out by the engine. Simply timing transactions to arrive to the database engine at the same time may not always result in a significantly higher frequency of database deadlocks.

However, further analysis of our results shows that the variance of the measured numbers of reported database deadlocks for the approach with scheduling transactions using the controller is much smaller when compared with the variance using the artificial injection approach. We think that the main reason for it is that scheduling transactions to arrive to the database engine at the same time increases the probability that low-level relational algebra operators that form a hold-and-wait cycle may execute at the same time.

When we study the values for the time it takes to reproduce the first occurrence of a database deadlock, we can not derive generalized conclusions that hold for all of the experiments, since the results are inconclusive for some of the experiments. Hence, we can not conclude that any one of these three approaches, that we conduct experiments on, gives lowest MTTDD. Therefore, STEPDAD reproduces database deadlocks with a higher
frequency in compare to the baseline approach (as we can see from the experimental results) and reproducing the first occurrence of a database deadlock in less time (less MTTDD) is a subject to our future work.

Nonetheless, when we study the values for the time it takes to reproduce the first occurrence of a database deadlock, the approach with scheduling transactions using the controller takes less MTTDD when compared with the MTTDD using the artificial injection approach for the experiment with a larger number of clients. Increasing the number of concurrent operations that have hold- and-wait cycles and timing them to arrive to the database engine at the same time makes it much quicker to reproduce the first occurrence of a database deadlock. This conclusion may be useful for stress and load testing of DCAs, since it specifies a direction with which it is likelier to cause database deadlocks, thus finding this abnormal behavior quicker and with fewer resources.

7.4 Evaluation of REDACT

In REDACT, we seek to answer the following research question.

RQ4: How effectively does the REDACT approach prevent database deadlocks?

We address RQ4 by running experiment with DCAs to determine REDACT’s overhead and compare it with other solutions. 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 conduct experiments with different waiting times in SC and we report the experimental results in Section 7.4.2.1.

7.4.1 Methodology

Like STEPDAD, following the guidelines for statistical tests to assess randomized approaches in software engineering (Arcuri and Briand, 2011), we collect highly representative samples of runs in REDACT, perform statistical tests on these samples, and draw conclusions from these tests. For each subject DCA, we ran each experiment 10 times with each approach to obtain a good representative sample like we do it in STEPDAD.

1) Independent Variables: In REDACT, we have three independent variables: the subject DCA, the type of the experiment, and the number of DCA clients. There are three types of experiment: the baseline experiment (type B), graceful experiment (type G) and redact experiment (type R). In 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, 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. Like STEPDAD, we used JMeter http://jmeter.apache.org to run subject DCAs with different numbers of clients.

2) Dependent Variables: In REDACT, we have two dependent variables: the number of database deadlocks that are observed during the experiment and the throughput as the total number of executed transactions. We repeated each experiment 10 times and report statistical results (average, median, min, max, variance) for 10 runs for the number of observed deadlocks and the throughputs.

Thus, the total number of experiments, that we conduct for redact, is equal to three DCAs × three types (B,G,R) × four client settings x 10 times = 360 experiments.

7.4.2 Analyzing Experimental Results

The results of experiments with the subject DCAs are shown in Table VIII where 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 15 mins 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.

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 throughput 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 param-
eter usually set between 15-30 seconds, by default is set to 60 seconds\(^1\) 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!

### 7.4.2.1 Delay in Supervisory Control

As we described earlier, in REDACT, supervisory control (SC) holds back one transaction from a set of transactions that are ready to be executed and may form a hold-and-wait cycle. SC waits for a configurable waiting times before it looks again for the cycles in the newly arrived transactions including the held back transactions from the past. We experiment with three different waiting times in the SC, for three subject applications (i.e., HIM, UCOM and DAN), and four different client loads and the results are shown in Table IX. 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 and no database deadlocks were observed.

We decreased the time from 20 seconds, the value that we used in experiments that we show in Table VIII 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

\(^1\)[http://tinyurl.com/dbdeadlocktimeout]
TABLE IX. RESULTS OF EXPERIMENTS WITH DIFFERENT WAITING TIME IN SC.

<table>
<thead>
<tr>
<th>Application</th>
<th>Clients</th>
<th>Time (seconds)</th>
<th>Throughput (Avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>HIM</td>
<td>1</td>
<td>0.2</td>
<td>57.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>34.1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.2</td>
<td>93.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>90.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>72.7</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.2</td>
<td>80.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>71.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>61.5</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.2</td>
<td>70.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>52.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>45.3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.2</td>
<td>445.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>261.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.2</td>
<td>280.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>264.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>161.6</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.2</td>
<td>834.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>885.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>634.4</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.2</td>
<td>1284.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4804</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>4181.2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0.2</td>
<td>4333.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>443.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0.2</td>
<td>50040.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2952.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>292.1</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>0.2</td>
<td>101209.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>29554.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>2923.8</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.2</td>
<td>106245.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>101194.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>27685.4</td>
</tr>
</tbody>
</table>

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 IX, throughput increases by more than two orders of magnitude in some cases with no database deadlocks observed.
7.4.3 Discussion

We experimented REDACT with three applications. 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 RQ1 that the REDACT approach is very effective in preventing database deadlocks as it prevents database deadlocks thereby removing the need for costly deadlock resolution within the database engine.

7.5 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 we only considers transactions in STEPDAD and REDACT. 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 cycles detection algorithm that we
propose in this dissertation 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 our solution and a potential threat to validity when dealing with ultra large-scale transactions. In addition, it may be more challenging to use the supervisory control for executions of large and complex applications to prevent database deadlocks when there are too many false positive 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, recall that there are over two dozens of different kinds of database deadlocks. In this dissertation, we experimented only with circular database deadlocks that are realized from hold-and-wait cyclic locks on resources by transactions. It is unclear how well STEPDAD and REDACT will perform on other kinds of database deadlocks, so this is a threat to external validity of our results.
CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

In this dissertation, we address a fundamental problem of predicting and preventing database deadlocks \((PD)^2\) by targeting the development of rigorous techniques and tools that are creative, transformative, and can automate an important software-engineering task—enhancing parallelism and scalability of DCAs in presence of database deadlocks.

As an outcome of this dissertation, we developed a novel solutions to \((PD)^2\), and we rigorously evaluated the proposed solutions on three different open-source DCAs and to the best of our knowledge there is no other adequate solution that addresses this problem. As a part of our solution, we presented a novel abstraction and a performance model and introduced a novel approach that combines dynamic analysis (prevents database deadlocks) with static analysis (detects hold-and-wait cycles). As a result, we enhanced parallelism and scalability of DCAs and achieved better performance.

We also released tools, experimental infrastructure, and testbeds developed as a part of this work, which will let other researchers and practitioners build on our results and will ultimately advance knowledge and understanding within software engineering.

Moreover, the results of our work will lay a foundation for a new direction of research of interactions between DCAs and databases, and we will support it with our tools for
software development and evolution. The short-term impact of our work will be in open-source software engineering community, where developers will use our approach to build new DCAs and evolve existing DCAs; the longer-term impact will be on tools in commercial software development.

8.2 Future Work

Our research can be extended in several directions. Among those directions, there are a few distinct possibilities:

- Reducing the time in cycles detection algorithm in the static phase of our solution. Although, identifying the cycles is one time effort, currently, the time complexity of the cycles detection algorithm is exponential and takes significant amount of time for large set of transactions.

- Extracting transactions from DCAs automatically by using various techniques (i.e., static analysis) and also to implement the solution for other database locking mechanisms (i.e. row level locking). Recall that we consider only table level locking mechanism in this work.

- Performing more analysis to control the program execution so that the mean time to deadlock detection (MTTDD) is reduced in STEPAD. Hence, the first occurrence of deadlock appears rapidly in controller-based and artificial injection approaches of STEPAD.

- Using our technique for other forms of deadlocks. For example, our solution could be used to prevent deadlocks in Java programs where various synchronization mechanisms
end up in deadlocks. In this case, instead of detecting cycles in SQL statements, they should be detected in various threads.

- Rigorously evaluating our approach in DCAs from various domains and investigating the experimental results in those domains based on performance model. In that way, a guideline can be provided to the users regarding the domains where our solution might be more useful.
APPENDICES
Appendix A

PERMISSIONS

We have published some of the materials of this dissertation. In the following, we give a reproduction of the publisher’s web pages giving permission and outlining requirements for reuse.

Association for Computing Machinery (ACM)

Reprinted from the ACM policies on author rights and publishing, given under the subsection 2.5.\(^1\) (last verified Feb 14, 2014)

2.5 Permanent Rights held by original Owners/Authors

The original Owner/Author permanently holds these rights:

- Reuse of any portion of the Work, without fee, in any future works written or edited by the Author\(^\star\star\), including books, lectures and presentations in any and all media.

\(^\star\star\)Requests made on behalf of others, i.e., for contributions to the work of other authors or other editors, may require payment of the fee.

\(^1\)http://www.acm.org/publications/policies/copyright_policy
Institute of Electrical and Electronics Engineers (IEEE)

Reprinted from the official policies related to the IEEE copyright procedures, given under the subsection 8.1.4.1 (last verified Feb 14, 2014)

8.1.4 IEEE Copyright Policy and Procedures (from the PSPB Operations Manual)

B. Ownership and rights of IEEE copyrighted material

3. Prior to publication by the IEEE, all authors or their employers shall transfer to the IEEE in writing any copyright they hold for their individual papers. Such transfer shall be a necessary requirement for publication, except for material in the public domain or which is reprinted from a copyrighted publication.

4. In return for the transfer of authors’ rights, the IEEE shall grant authors and their employers permission to make copies and otherwise reuse the material under terms approved by the Board of Directors.

CITED LITERATURE


VITA

NAME: B.M. Mainul Hossain

EDUCATION: Ph.D., Computer Science
University of Illinois at Chicago, 2014

M.S., Computer Science
University of Dhaka, Bangladesh, 2003

B.S., Computer Science
University of Dhaka, Bangladesh, 2002

WORK EXPERIENCE:

Research Assistant and Teaching Assistant
Department of Computer Science
University of Illinois at Chicago
Aug 2009–Dec 2013

Software Engineer (intern)
Microsoft Corporation, Redmond, Washington
May 2013–Aug 2013

Researcher (intern)
Accenture Technology Lab, San Jose, California
May 2012–Aug 2012

Researcher (intern)
Accenture Technology Lab, Chicago, Illinois
May 2011–Aug 2011

Lecturer
Institute of Information Technology
University of Dhaka, Dhaka, Bangladesh
July 2008–Aug 2009

Software Engineer
Uniq Software & Systems Ltd, Dhaka, Bangladesh
AWARDS: One of the team members of the University of Illinois at Chicago team for Google Code Competition held on Jan 27, 2012 at Chicago and won the second best project prize.

One of the finalists of the USCC US Cyber challenge 2011. Also, attended the 2011 the US Cyber Challenge (USCC) Virginia Regional Cyber Security Boot Camp held in August 1st through the 5th, where I represented UIC. (http://www.uscyberchallenge.org/)

One of the finalists of the ITA Fall challenge 2010 and 2012 (two times), organized by Illinois Technology Association, where I represented the University of Illinois at Chicago. (http://illinoistech.org/page.aspx/Fall_Challenge).

One of the recipients of University of Dhaka Talent-pool scholarship and full tuition waver for excellence in B.Sc. (Honors) in the Faculty of Science, University of Dhaka, Bangladesh.

Recipient of Government scholarship and full tuition waiver in the Department of Computer Science & Engineering, University of Dhaka, Bangladesh, for excellence in H.S.C examination, 1998, Dhaka Board.


B. M. Mainul Hossain and Mark Grechanik. Effectively Parallelizing Rule-Driven Applications. In the 36th International Conference on Software Engineering, ICSE 2014, will be held in Hyderabad, India, 2014. (submitted)