Correct Web Service Transactions in the Presence of Malicious and Misbehaving Transactions

by

John Thomas Ravan III

Bachelor of Science
The Citadel, The Military College of South Carolina 2011

Master of Science
The Citadel Graduate College 2014

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2021

Accepted by:

Csilla Farkas, Major Professor

Shankar M. Banik, Committee Member

John R. Rose, Committee Member

Lannan Luo, Committee Member

Jorge Crichigno, Committee Member

Tracey L. Weldon, Interim Vice Provost and Dean of the Graduate School

Dedication

This dissertation is dedicated to my wife, Michelle, who has been a constant source of support and encouragement during the challenges of graduate school and life. I am truly thankful for having you in my life. This work is also dedicated to my two sons, J.R. and James Ravan. I pray that this allows me the opportunity to give you the life that you deserve.

Acknowledgments

I would like to take this opportunity to thank the University of South Carolina for allowing me to complete my Ph.D. in Computer Science. I have been able to learn so much in the areas of Computer Science that will allow me to continue my contributions to the field in the future.

I would also like to thank The Citadel for their support during my tenure at University of South Carolina. While completing my degree through distance learning, The Citadel allowed me to use facilities and resources that enabled me to complete my degree program. The Citadel will always be my home and I will always be grateful for their support.

I want to thank my advisors Dr. Csilla Farkas and Dr. Shankar Banik. Both of you have been there for me from the very beginning. There is no way to quantify the number of hours we have spent working on this contribution and both of you have been there to give the feedback that I needed no matter how hard it may have been.

I also want to thank Tidal Grounds coffee shop. With a family consisting of my wife and two young children it's hard to find the quiet time needed to dedicate toward a dissertation. I met many wonderful people who encouraged me and never let me give up. Without that support group this work would not have been made possible.

Last, but certainly not least, I want to thank my wife for her support through all the years. From the very first day when we decided to embark on this journey together you have always been there for me and given me the time I've needed to dedicate toward this work. This could not have been possible without you by my side each day.

Abstract

Concurrent database transactions within a web service environment can cause a variety of problems without the proper concurrency control mechanisms in place. A few of these problems involve data integrity issues, deadlock, and efficiency issues. Even with today's industry standard solutions to these problems, they have taken a reactive approach rather than proactively preventing these problems from happening. We deliver a solution, based on prediction-based scheduling to ensure consistency while keeping execution time the same or faster than current industry solutions. The first part of this solution involves prototyping and formally proving a prediction-based scheduler.

The prediction-based scheduler leverages a prediction-based metric that promotes transactions with a high performance metric. This performance metric is based on the transaction's likelihood to commit and its efficiency within the system. We can then predict the outcome of the transaction based on the metric and apply customized lock behaviors to address consistency issues in current web service environments. We have formally proven that the solution will increase consistency among web service transactions without a performance degradation. The simulation was developed using a multi-threaded approach to simulate concurrent transactions. Our empirical results show that the solution performs similarly to industry solutions with the added benefit of ensured consistency. This work has been published in IEEE Transactions on Services Computing.

The second part of the solution involves building the prediction-based metric mentioned previously. In the initial solution we assumed that the categorization of

transactions is provided in advance. To incorporate the ability to dynamically adjust transaction reputations we extended the four category solution to a dynamic reputation score. The attributes used in the reputation score are system abort ranking, user abort ranking, efficiency ranking, and commit ranking. With these four attributes we were able to establish a dynamic dominance structure that allowed for a transaction to promote or demote itself based on its performance within the system. This work has been submitted to ACM Transactions on Database Systems and awaiting review.

Both phases provide a complete solution of prediction-based transaction scheduling that provides dynamic categorization no matter the transactional environment.

Future work of this system would involve extending the prediction-based solution to a multi-level secure database with an added dimension. Our goal is to increase concurrency of multi-level secure transactions without creating a covert channel. The dimension provides a security classification in addition to attributes for dynamic reputation that allows for transactions to establish dominance. Our reputation score would provide a cover story for timing differences of transactions of different security levels to allow for a more robust scheduling algorithm. This would allow for high security transactions to gain priority over low security transactions without creating a covert timing channel.

Preface

The following dissertation took many years of sacrifice and dedication to complete. It took hours of meetings, proofreading, reading, rereading, editing, starting over, and simply scratching our heads sometimes until solutions presented themselves. But before this dissertation even started to take shape it started with me in a small blue collar town in South Carolina where, in my eyes, Computer Science was a luxury and not a necessity.

I grew up in Woodruff, South Carolina. Woodruff is a small blue collar town in Upstate South Carolina with a population of a little over 4,000 residents. I grew up working hard and enjoying the little things in life like most of my family. I was an outdoorsman and loved hunting and fishing like most of my friends. On rainy days me and my friends would play video games inside to pass the time. It started with playing simple single user games and then progressed into multi-user online role playing games where groups of us would connect at night to accomplish certain missions. It was then when I began to find a new love that could overcome the outdoors. I couldn't decide if I enjoyed the company of friends playing together or the technology that made the game possible more at the time but little did I know that a basic love for technology and the things it made possible would lead me into a career path where I could potentially further that technology for generations to come.

My senior year of high school comes and just like all of my other friends we're deciding what college we plan to attend and more importantly what we plan to study. After my love of video games and the technology that made it possible I began to appreciate technology in the general sense. No matter if it was a new computer

that was released or a new circular saw for woodworking, I could begin to see the technology that was needed to make these advancements possible. With technology as my primary love then Computer Science became more and more apparent as my focus. Much to my surprise, studying Computer Science has very little to do with video games or circular saws.

There were many times that I never thought I was going to make it. Taking 18-20 hours a semester, studying on the weekends, taking summer classes. Some days I never thought it would end but I was able to push through. In 2011 I graduated The Citadel with a B.S. in Computer Science. Three years later I was able to complete my M.S. in Software Engineering.

During my junior year I was able to start working as a software engineering intern for a company local to Charleston, SC. Ever since then I have worked as a software engineer in some capacity for multiple companies. Working as a software engineer while also being involved in academia was the best pairing I could give myself going forward. I could see both sides of the coin in how technology really did further multiple advancements. I could see the theory of software design and architecture while using the newest software framework that implemented those architectural concepts. I could intelligently defend why it was so important that software engineers understand the underlying theoretical concepts rather than simply importing a framework and continuing on. It allowed me to see that academia had a place and it wasn't in a silo with other researchers but it was meant to be paired with society. I couldn't stop.

In August of 2014 I started my Ph.D. in Computer Science. Much of the same struggles I faced in my previous years in academia would continue but, once again, I was able to overcome. When you read the following work don't make the mistake of thinking this work was done in a silo of academia. What you have before you is a culmination of over 12 years of experience of computer science education and industry software engineering. The work before you provides a solid theoretical contribution

but is also backed by experimentation results and prototypes that were built using the latest technologies and processes used by hundreds of companies for their applications. I never understood why there were resources built for academia proof of concepts that were designed to address an industry problem. This work marries the two concepts together. In this work academia crosses the finish line with industry driving the car.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

2PL . . . . . Two Phase Locking

ACID . . . . . Atomicity Consistency Isolation Durability

BPEL . . . . . Business Process Execution Language

DBMS . . . . . Database Management System

DRP . . . . . Dynamic Reputation Prototype

ELM . . . . . External Lock Manager

HCHE . . . . . High Commit, High Efficiency

HCLE . . . . . High Commit, Low Efficiency

LCHE . . . . . Low Commit, High Efficiency

LCLE . . . . . Low Commit, Low Efficiency

MLS . . . . . Multi-level Secure

PBS . . . . . Prediction-Based Scheduler

RCDS . . . . . Resource Category Data Structure

Chapter 1

Introduction

Consistency among multiple interleaved transactions in a web service context has always been an issue for researchers and database administrators. Isolation and atomicity are two of the four ACID properties that are often relaxed in order to prevent a performance bottleneck. However, when these properties are relaxed, the database can reach an inconsistent state when concurrent transactions interleave incorrectly. This causes data to become corrupted, expensive compensation transactions to be executed, and cascading rollbacks on multiple nodes to be completed before processing can continue.

1.1 Motivations

By looking at a practical use case we can more clearly see the issue and the need for a solution that ensures consistency. In Figures 2.1 & 2.2, we see five web services executing on three different database instances. The first four web services create a common business process created by BPEL (*Web Services Business Process Execution Language Version 2.0* n.d.). The web services are: WS_1 (decrement inventory by product ID), WS_2 (process payment), WS_3 (add order by user ID), and WS_4 (delete user payment info). The goal of the process is to allow a customer to purchase a product from an e-Commerce site. WS_5 (delete user payment info) and WS_3 execute within the same database instance. With the relaxed properties in the web service context, concurrent executions of WS_5 and WS_3 could cause an inconsistent state on $Node_3$. This would then cause a cascading rollback to execute and revert the

committed operations of WS_1 and WS_2. Existing research shows that many solutions have been presented in the past to address this issue (e.g., Alomari, A. Fekete, and Röhm 2014, Alrifai et al. 2009, Bailis et al. 2014, Greenfield et al. 2007, Jacobi and Lichtenau 1999, and Jang, K. Fekete, and Greenfield 2007).

The most influential research that inspired the prediction-based solution was the Promises Model. The Promises model presented by Alan Fekete et al. (e.g., Greenfield et al. 2007 and Jang, K. Fekete, and Greenfield 2007) is an elegant solution that "promises" a particular transaction that the requested resource will be available while allowing concurrent transactions to still execute on that resource. The Promises solution is robust in that it allows the "strengthening" or "weakening" of promises after they have already been made. This allows existing promises on resources to be modified without breaking the existing promise entirely. However, the solution introduces backwards compatibility issues along with a potential bottleneck at the occurrence of registering a promise for a particular transaction.

However, none of the existing work improves currency control based on the performance of the transactions. That is the likelihood that the transaction will commit and the computational cost of the transaction. In our work we provide improvements for concurrency control using these performance characteristics.

1.2 Contributions

We provide a prediction-based solution to support efficient and consistent concurrency control. Our approach is based on building a reputation for each transaction using its efficiency rate (i.e., computational cost) and the outcome (i.e., commit or abort). Using these properties transactions are categorized into four categories. The priorities associated with each category impact the transaction's scheduling. Our aim is to prevent cascading rollbacks and inconsistent database state while supporting practical concurrency control. We provide new lock types corresponding to transaction

categories. Using these locks transaction scheduler will be able to determine which lock requests to permit. These eventually determine transaction scheduling, delays, and aborts. Our expectation is that prediction-based scheduling will increase both efficiency and consistency.

We identified two research areas in the context of prediction-based scheduling within web service environments that need to be addressed. These are:

- transactional correctness within concurrency control
- dynamic reputation for transactions

1.2.1 Transactional Correctness

In this work we developed the theoretical foundation for the prediction-based scheduling. This included the development of a framework, associated concepts, and technologies. A completely new concurrency control paradigm was developed in order to elevate particular transactions over others. In this paradigm there are three actions used to determine the course of action for a particular transaction. These three actions either grant, elevate, or decline an transaction to enable concurrent operations and prevent deadlock. By ensuring the transactional correctness within the prediction-based solution, we can then use this foundation to build upon in regards to other research areas. The work discussed in this area is documented in Chapter 2.

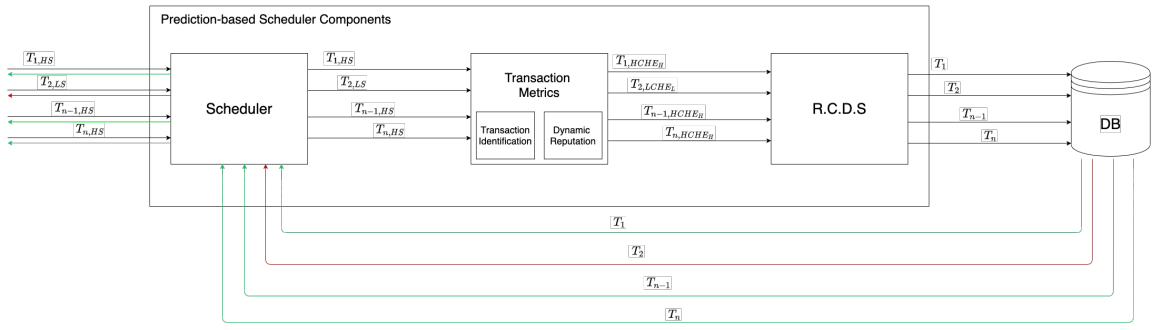


Figure 1.1: Overall System Model of Prediction-based Scheduler

1.2.2 Dynamic Reputation for Transactions

In the previous section, the categorization of the transaction is assumed in order to continue forward with the decision model. This section of the dissertation involves the work needed to establish a dynamic reputation management system to allow for dynamic reputation. It involves building a reputation score for each transaction based on the transactions ranking of efficiency, commits, system aborts, and user aborts. Once a reputation score is provided we can then use dynamic reputation management to dynamically promote and demote transactions. This then allows the system to adapt to its environment dynamically. The work discussed in this area is documented in Chapter 3.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows: Chapter 2 outlines the research done in regards to transactional correctness while preserving concurrent operations within a web service environment. This work is published in Ravan, Banik, and Farkas 2020. Chapter 3 addresses the research done in order to build a reputation for a given transaction that is dynamic to its environment and its changing attributes. This work is submitted to ACM Transactions on Database Systems and awaiting response. Chapter 4 addresses the future work needed to adapt the prediction-based scheduler in Chapter 2 and Chapter 3 to a multi-level secure database and other possible solutions. Chapter 5 contains the concluding remarks regarding all areas of research.

Chapter 2

Prediction-Based Scheduler

2.1 Overview

In this chapter, we present the foundation of the prediction-based scheduler. The problem defined here is the primary motivation for the current work and the anchor for the subsequent extensions. The work presented in Chapter 2 is taken from Ravan, Banik, and Farkas 2020.

2.2 Introduction

Consistency among multiple interleaved transactions in a web service context has always been an issue for researchers and database administrators. Isolation and atomicity are two of the four ACID[1] properties that are often relaxed in order to prevent a performance bottleneck. However, when these properties are relaxed, the database can reach an inconsistent state when concurrent transactions interleave incorrectly. This causes data to become corrupted, expensive compensation transactions to be executed, and cascading rollbacks on multiple nodes to be completed before processing can continue. By looking at a practical use case we can more clearly see the issue and the need for a solution that ensures consistency. In Figures 2.1 & 2.2, we see five web services executing on three different database instances. The first four web services create a common business process created by the Business Process Execution Language (BPEL) *Web Services Business Process Execution Language Version 2.0*

[1]The four traditional database properties are Atomicity, Consistency, Isolation, and Durability

n.d. The web services are: WS_1 (decrement inventory by product ID), WS_2 (process payment), WS_3 (add order by user ID), and WS_4 (delete user payment info). The goal of the process is to allow a customer to purchase a product from an e-Commerce site. WS_5 (delete user payment info) and WS_3 execute within the same database instance. With the relaxed properties in the web service context, concurrent executions of WS_5 and WS_3 could cause an inconsistent state on $Node_3$. This would then cause a cascading rollback to execute and revert the committed operations of WS_1 and WS_2. Existing research shows that many solutions have been presented in the past to address this issue (e.g., Alomari, A. Fekete, and Röhm 2014, Alrifai et al. 2009, Bailis et al. 2014, Greenfield et al. 2007, Jacobi and Lichtenau 1999, and Jang, K. Fekete, and Greenfield 2007). The most influential research that inspired the prediction-based solution was the Promises Model.

The Promises model presented by Alan Fekete et al. (e.g., Greenfield et al. 2007 and Jang, K. Fekete, and Greenfield 2007) is an elegant solution that "promises" a particular transaction that the requested resource will be available while allowing concurrent transactions to still execute on that resource. The Promises solution is robust in that it allows the "strengthening" or "weakening" of promises after they have already been made. This allows existing promises on resources to be modified without breaking the existing promise entirely. However, the solution introduces backwards compatibility issues along with a potential bottleneck at the occurrence of registering a promise for a particular transaction.

Our prediction-based solution allows transactions to build a reputation linked to a category (categories are defined in Definition 4 & 5). The priorities associated with the category impact the transaction's scheduling in order to prevent cascading rollbacks. There are lock behaviors associated with the categories that enable preemptive scheduling in order to increase efficiency. The contributions in the prediction-based solution address the two issues of efficiency and consistency where existing research

solutions fall short. Section 2.3 outlines the problem along with a use-case scenario. Section 2.4 discusses the existing research that has already taken place in regards to the problem. Section 2.5 outlines the system model for the solution. Section 2.6 discusses the algorithms needed for the solution and their psuedocode. Section 2.7 provides the formal proofs for the given solution. Section 2.8 illustrates the simulation results gathered from the prototype.

2.3 Problem Definition

Transactions in traditional database systems satisfy the four ACID properties: Atomicity, Consistency, Isolation, and Durability. ACID transactions guarantee leaving the database in a consistent state. However, in a web service environment ACID properties and transactional concurrency cannot be fulfilled, due to performance issues. Common occurrences in web service environments involve the collaboration of multiple web services to fulfill a particular task without coupling all the needed operations in one service. Languages, such as BPEL (Business Process Execution Language) *Web Services Business Process Execution Language Version 2.0* n.d., allow a large majority of these business processes to execute web services that must access the underlying database management systems (DBMS). These transactions may be required to relax the isolation property of the database in order to accommodate concurrent transactions.

In the event that a transaction fails (abort), and there are transactions depending on the successful execution of the failed transaction, a cascading rollback must occur, due to the inconsistency of the database. This occurs often in the web service environment due to the relaxed atomicity and isolation properties. The relaxation of the atomicity property allows operations within a transaction to commit any changes performed by the operation permanently before the entire transaction has been completed. The relaxation of the isolation property allows the interleaved execution of

operations on a particular resource from concurrent transactions. This improves performance tremendously; however, in the event an operation fails, causing an abort to be issued, every transaction depending upon the failed operation must be aborted as well in order to restore consistency. These compensation transactions have become a standard practice in web service environments but sacrifice the consistency of the system. The next section will discuss a use-case scenario where this problem exists.

2.3.1 Use-Case Scenario

Assume that an e-commerce web site allows users to purchase products through an online interface. When a customer places an order a ticket is generated (see Figure 2.1) containing information about the user and purchase. The ticket from the site is then handed off to a business process (starting at WS_1 in Figure 2.2) containing multiple web services, on multiple databases in a distributed context. Figure 2.2 displays the architecture of the back-end business process in sequence.

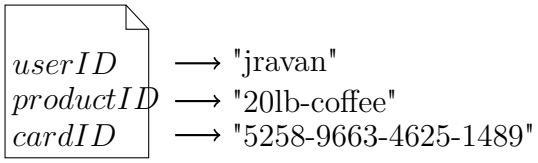
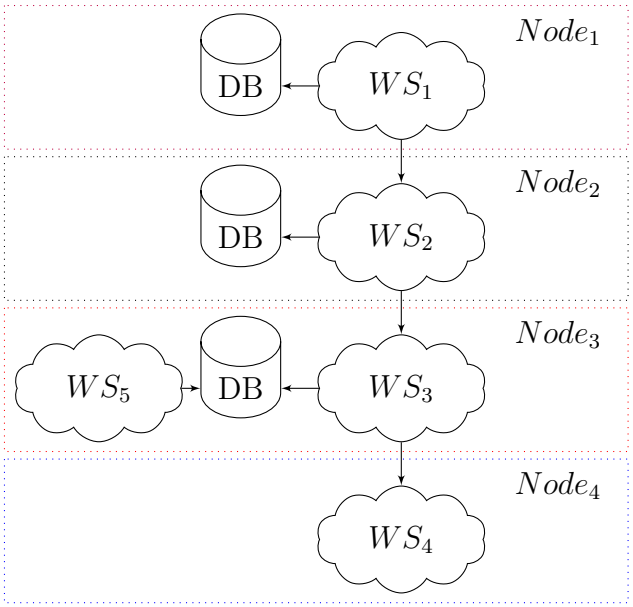


Figure 2.1: e-Commerce Web Site Ticket Example

Each dotted rectangle within Figure 2.2 represents a distributed node containing a public web service executing transactions on an underlying database. In order for a successful purchase to take place within the e-commerce environment, the business process containing the web service sequence WS_1, WS_2, WS_3, and WS_4 must execute successfully. WS_1 ($DecrementInventoryByProductID$) uses the $productID$ provided from the current ticket in order to query the database regarding the requested product. WS_2 ($ProcessPayment$) uses the payment information provided in the ticket to charge the user. WS_3 ($GetOrderByUserID$) is a service that takes the given $userID$ and $productID$, and returns a record of purchase for the user.



$WS_1 = DecrementInventoryByProductID$
$WS_2 = ProcessPayment$
$WS_3 = GetOrderByUserID$
$WS_4 = GenerateReceipt$
$WS_5 = DeleteOrderByUserID$

Figure 2.2: Business Process for e-Commerce Web Site

WS_4 (*GenerateReceipt*) simply generates a receipt from the ticket information and is returned to the user's interface. Once WS_4 has completed processing, and only when it is finished, a purchase is considered successful.

The issue is that multiple web services may be executed on a single database instance. Without concurrency control, interleaved executions, such as the case with WS_3, and WS_5, may cause inconsistencies.

Assume WS_3 and WS_5 are $T_{GetOrderByUserID}$ and $T_{DeleteOrderByUserID}$. The deletion transaction contains four operations: $R_2(b)$, $W_2(b)$, $R_2(a)$, and $W_2(a)$. The transaction will first READ from the database to ensure the user exists and then WRITE out the order information if the order does exist[2]. The other transaction, $T_{GetOrderByUserID}$, contains a single $R_1(a)$ and $R_1(b)$. Each transaction contains a C_i

[2]In this instance the WRITE operation executed would write a $NULL$ or empty record to the database in order to "delete" the record.

operation that represents the COMMIT operation that the database executes after all operations have completed. Figure 2.3 shows the two transactions along with the operations contained within.

$$W_3 = T_{GetOrderByUserID} = R_1(a)R_1(b)C_1$$
$$W_5 = T_{DeleteOrderByUserID} = R_2(b)W_2(b)R_2(a)W_2(a)C_2$$

Figure 2.3: e-Commerce Web Service Transaction Sequences

For the transactions listed above, we know the commit rate, number of transactions executed, and the average execution time. Commit rate is the percentage of successful commits over the total amount of attempted executions of that particular transaction (Definition 1). Number of transactions represents the total number of executions attempted (commits and aborts combined) for that particular transaction. Efficiency rate represents the average efficiency of the particular transaction to reach a completion state; whether it is a failure or success (Definition 2). A transaction with a very long execution time will have an efficiency rate that is much lower than a transaction that executes and completes very quickly. The attribute data listed previously is consolidated into Table 2.1 below.

Table 2.1: Transaction Metrics

Transaction Metrics			
Transactions	**Commit Rate**	**# of Trans.**	**Eff. Rate**
$T_{DeleteOrderByUserID}$	98%	200	98%
$T_{GetOrderByUserID}$	97%	520	99%

In a web service context, without the prediction-based solution, the isolation property will be relaxed and a serializable execution will be created. This serializable execution will be based on the conflicting operations of the two transactions. Figure 2.4 displays an example of a generated serializable execution from $T_{GetOrderByUserID}$ and $T_{DeleteOrderByUserID}$.

$$T_{Schedule} = R_1(a)R_2(b)R_1(b)W_2(b)R_2(a)W_2(a)$$

Figure 2.4: Generated Schedule

The schedule listed in Figure 2.4 is a well-formed serializable schedule. Between each operation within the schedule, a commit operation C_i is executed. This shows how the Atomcity property of a database is relaxed during the execution of a schedule. Therefore, if $R_2(a)$ fails, a cascading rollback will be issued causing the operations of $T_{GetOrderByUserID}$ to be rolled back regardless of their successful execution. The operations of $T_{GetOrderByUserID}$ must be rolled back since they are a part of $T_{Schedule}$ and a COMMIT operation has been executed already. If $T_{GetOrderByUserID}$ was executed as a part of the business process outlined in the use case, then the operations of the previous web services (WS_1 and WS_2) must be rolled back.

Conversely, if the data in Table 2.1 were taken into consideration before generating the serializable execution, $T_{GetOrderByUserID}$ could be given a more restrictive concurrency control mechanism, such as locking. Table 2.1 contains data that proves instances of $T_{GetOrderByUserID}$ have a high rate of commit with a high percentage of efficiency throughout the system. Table 2.1 also displays that instances of $T_{DeleteOrderByUserID}$ could use locking techniques due to its history. Therefore, $T_{GetOrderByUserID}$ and $T_{DeleteOrderByUserID}$ could execute within the same schedule using traditional locking techniques, commit changes quickly and successfully, and prevent a cascading rollback from reverting the effects of WS_1 and WS_2 by using the reputation provided by the separate transactions. In this solution, we aim to provide concurrency control that is based on the metadata of the transactions. Our hypothesis is that if transactions could be treated differently based on their history, stronger concurrency control techniques could be leveraged, such as locking, which ensures the consistency property for web service transactions. Section 2.7 walks through this exact use-case scenario with two-phase locking and the prediction-based solution. The

next section will discuss the existing research that has taken place to remediate this problem without the current knowledge of transaction metrics.

2.4 Related Work

Ensuring successful concurrency control in web service transactions has been studied in depth for some time (e.g., Jang, K. Fekete, and Greenfield 2007, Greenfield et al. 2007, Alrifai et al. 2009, Dai, Yang, and B. Zhang 2009, Gao and Wu 2005, Ferreira et al. 2012, K.-W. Lee and Kim 2000, and Olmsted 2015). The Promises model presented by Alan Fekete et al. (e.g., Jang, K. Fekete, and Greenfield 2007 and Greenfield et al. 2007) is an elegant solution that "promises" a particular transaction that the requested resource will be available while allowing concurrent transactions to still execute on that resource. Alomari et al. present a solution involving an External Lock Manager (ELM) that resides outside of the DBMS Alomari, A. Fekete, and Röhm 2014. This allows a layer of separation between the application and the DBMS in order to schedule transactions using special business logic according to the environment. In the solution presented by Alrifai et al. Alrifai et al. 2009, an edge chasing solution using dependency graphs is incorporated in order to detect dependencies between globally scheduled transactions. The solution was tested parallel to the well known 2-Phase Locking Protocol (2PL) and provided promising results regarding efficiency. However, the Alrifai et al. solution becomes less efficient when the number of dependency cycles are detected. The model of the different lock types comes from Christian Jacobi et al. Jacobi and Lichtenau 1999 with research in concurrent locking with parallel database systems. The researchers extended the use of the native lock types in the existing database structure in order to speed up thread processing on multi-processor machines. Prediction-based concurrency control has been proposed by Eunhee Lee et al. E. Lee et al. 2001 with the entity-radius solution. This solution uses the concept of multiple entity radii that attempts to predict the next user based

on their location. The prediction is generated from the location within a radius of the replicated site and their navigation speed. The solution provided is elegant with excellent experimental results but no formal proof or analysis of the algorithm is provided. Other solutions to prevent business process cancellation or rollback when participants of the process do not behave correctly involve global views of the process Bailis et al. 2014, Riegen et al. 2010. Support for these types of solutions are based on the well-known Oasis specifications of WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity Feingold and Jeyaraman 2009, Little and Wilkinson 2009, Freund and Little 2009. These solutions are well-designed; however, they require the presence of a global coordinator throughout the entire business process. In the next section, we will discuss the prediction-based solution's system model generated from the current available work in the research community.

2.5 System Model

This section outlines the system model on which the solution is built. Definitions for new concepts are introduced first, considering they will be used to describe the different components that consist of the system model. Later subsections explain in detail the different components and processes required for the prediction-based solution. We assume that all the transactions are well-formed and the scheduler is legal.

2.5.1 Definitions

Definition 1. *(Commit Rate) - the commit rate of a transaction T_i, denoted as $C_r(T_i)$, is calculated by the given formula:*

$$|c_i| = \#\ of\ executions\ of\ T_i\ ending\ with\ a\ COMMIT\ result$$

$$|a_i| = \#\ of\ executions\ of\ T_i\ ending\ with\ a\ ABORT\ result$$

$$C_r(T_i) = \frac{|c_i|}{|c_i + a_i|}$$

Once this ratio is calculated, it is compared to the overall ratio of commit to abort executions for all transactions.

Definition 2. *(Efficiency Rate) - the efficiency rate of a transaction T_i, denoted as $E_r(T_i)$, is based on how its execution time compares to all transactions executed within the execution environment. The efficiency rate is calculated based on the given formula:*

$$E_r(T_i) = \frac{AVG(AVG(T_1), ..., AVG(T_n))}{AVG(T_i)}$$

Definition 3. *(Categorization Thresholds) - categorization thresholds are upper and lower limits of both commit rate and efficiency rate that are used when categorizing transactions*[3]*.*

Definition 4. *(Transaction Categories) - Let $T = \{T_1, \ ... \ , T_n\}$ be a set of transactions and $C = \{HCHE, HCLE, LCHE, LCLE\}$ a set of category names. The mapping τ associates a category name with each transaction as follows:*

$$\tau : T_i \rightarrow \left\{ \begin{array}{ll} HCHE & \text{if } C_r(T_i) > 0.5 \text{ and } E_r(T_i) > 1 \\ HCLE & \text{if } C_r(T_i) > 0.5 \text{ and } E_r(T_i) \leq 0.5 \\ LCHE & \text{if } C_r(T_i) \leq 0.5 \text{ and } E_r(T_i) > 1 \\ LCLE & \text{if } C_r(T_i) \leq 0.5 \text{ and } E_r(T_i) \leq 0.5 \end{array} \right\}$$

In order to ensure correct lock techniques are selected, transactions cannot be simply characterized as good or bad based on their metrics. For example, a transaction T_1 may have a 100% commit rate, but it may have a long execution time. A transaction with these characteristics should not be penalized when, in fact, it is

[3]The categorization bounds are static values that do not change throughout the system execution. Both the efficiency rate and the commit rate bounds are currently at 50% creating full coverage of all transactions.

a well behaving transaction. On the other hand, a transaction T_2 that has a 100% commit rate and has an extremely short execution time should not be treated with the same priority as T_1. This is where certain of levels must be established to ensure the most appropriate selection is made.

In order to establish levels, a system of categories was put in place base on the metrics mentioned previously. The first categorization is based solely on the efficiency of the transaction. In this categorization, there are two attributes: *high efficiency (HE)* and *low efficiency (LE)*. A transaction that has been labeled as *HE* is considered to execute with an efficiency in the upper 50% of all transactions executed within the system[4]. The second attribute, *LE*, is any transaction where its efficiency rate is in the lower 50% of all transactions executed.

The second categorization that the levels are built on is based solely on the outcome of the transaction. These attributes are *high commit (HC)* and *low commit (LC)*, which are much more simple to define. A transaction with a *HC* attribute has committed successfully over 50% of executions (upper 50% of all transactions). A transaction with an *LC* attribute has failed over 50% of its executions (lower 50% of all transactions). This categorization correlates directly with the commit rate defined (see Definition 1).

With these two categorizations and two attributes, a four-level system was devised in order to select appropriate lock types. The four categories devised are *high commit-high efficiency (HCHE)*, *high commit-low efficiency (HCLE)*, *low commit-high efficiency (LCHE)*, and *low commit-low efficiency (LCLE)*. Depending on the level in which the transaction has been placed, different lock types will be granted in order to perform concurrency control.

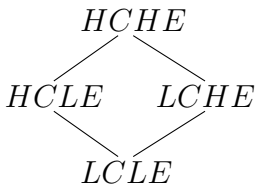
---

[4]See Section 2.5.6 and Section 2.5.1 for more clarification

Definition 5. *(Transaction Category Dominance) - The Transaction Category Dominance is a pair, denoted as $C_D = (C, L)$, where C is the set of transaction categories, and L is a partial order of the categories such that:*

$$HCHE > HCLE > LCLE$$

$$HCHE > LCHE > LCLE$$



Note that the categories $HCLE$ and $LCHE$ are not comparable. That is, we cannot establish a dominance relation between transactions in $HCLE$ and $LCHE$ categories.

However, in our model, we focus on reducing the need of compensation for aborted transactions. Therefore, we prioritize the commit property over the efficiency property. We introduce the dominance relation below in Table 2.2.

$$HCLE > LCHE$$

Table 2.2: Category Priorities

	Priority
HCHE	I
HCLE	II
LCHE	III
LCLE	IV

Definition 6. *(Conflicting Operations) - two operations are conflicting:*

1. *they are contained within two different transactions,*

2. *both operations are operating on the same data item, and*

3. *at least one of the operations is a WRITE*

Example 6.1. *Let o_1 be a READ operation on data item $UserID$ in transaction T_1 and let o_2 be a WRITE operation on $UserID$ in T_2. These two operations are conflicting.*

Definition 7. *(Compatibility) - a data item is locked in a non-compatible mode if:*

1. *the data item is locked by write-lock, or*

2. *the data item requesting the lock is categorized with a lower priority*

Definition 8. *(Legal Scheduler) - a prediction-based scheduler is legal if:*

1. ***Grant:*** *(denoted as the addition symbol, +) a transaction is permitted to lock a data item if the data item is not already locked in a non-compatible mode (see Definition 7) by an other transaction*[5]*.*

2. ***Decline:*** *(denoted as the subtraction symbol, -) a transaction T_i is denied to lock a data item if the data item is already locked in a non-compatible mode (see Definition 7) by another transaction T_j where $\tau(T_i) \leq \tau(T_j)$*

3. ***Elevate:*** *(denoted as lowercase delta, δ) a transaction T_i is permitted to lock a data item if all the non-compatible locks (see Definition 7) on the data item are being held by transactions $T_1, ..., T_k$ such that $\tau(T_j) < \tau(T_i)$ for all $j = 1, ..., k$ in this case $T_1, ..., T_k$ must first release the locks on the data item before T_i is permitted to lock the data item.*

[5]In the event that a lock is requested of a resource that has not issued any locks, then the lock will be automatically granted. There is no conflict, and therefore, the compatibility matrices do not apply.

Example 8.1. *A read-lock for a data item R has been granted to transaction T_1 within the HCHE transactional category. Transaction T_2 within the HCLE is also requesting a read-lock on data item R. This does not cause a conflict and therefore the grant action will be taken and the requested lock will be issued to T_2.*

Example 8.2. *A write-lock for a data item R has been granted to transaction T_1 within the HCHE transactional category. Transaction T_2 within the HCLE is requesting a write-lock to data item R. This causes a conflict and therefore the decline action will be taken and the requested lock will not be issued to T_2.*

Example 8.3. *A read-lock for a data item R has been granted to transaction T_1 within the LCLE transactional category. Transaction T_2 within the HCHE is requesting a write-lock to data item R. However, this causes a conflict; since T_2 contains a higher priority than T_1; the elevate action will be taken and the requested lock will be issued to T_2.*

2.5.2 Environment

In a web services environment, multiple web services can submit transactions to a database to be executed. As mentioned before, if only one transaction can be executed at any given time, then other concurrent transactions from other web services must wait. This causes a significant performance degradation; therefore, concurrent transactions from multiple web services must be handled. This is where the Database Management System's (DBMS) scheduler plays an important role in ensuring concurrent transactions preserve consistency within the database. The scheduler receives transactions that are submitted from web services and generates a serializable schedule based on the conflicting operations contained within the transactions.

After receiving the concurrent transactions (e.g., T_1, T_2, ... , T_N) the scheduler is responsible for analyzing the conflicting operations within the transactions and generating a serializable schedule that will then be executed on the database itself (e.g.,

T_{Sched}). This serializable execution is guaranteed to leave the database in a consistent state after a successful execution due to the analysis performed by the scheduler and the ordering of the operations within the transaction. Figure 2.5 displays the generation of T_{Sched} from multiple web service transactions. Figure 2.6 displays a diagram of the database scheduler in a web service context.

$$
\begin{aligned}
T_1 &= \{\ W_1(r_1)R_1(r_1)W_1(r_2)R_1(r_2)\ ...\ W_1(r_n)R_1(r_n)C_1\ \} \\
T_2 &= \{\ W_2(r_1)R_2(r_1)W_2(r_2)R_2(r_2)\ ...\ W_2(r_n)R_2(r_n)C_2\ \} \\
&\vdots \\
T_N &= \{\ W_n(r_1)R_n(r_1)W_n(r_2)R_n(r_2)\ ...\ W_n(r_n)R_n(r_n)C_N\ \} \\
\hline
T_{Sched} &= \{\ T_1T_2\ ...\ T_N\ \}
\end{aligned}
$$

Figure 2.5: Generation of T_{Sched} from Concurrent Web Service Transactions

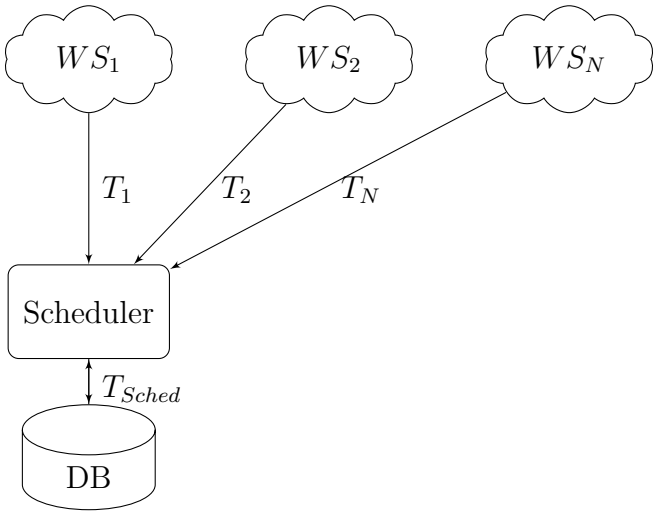


Figure 2.6: Web Service Environment with Scheduler

The architecture shown in Figure 2.6 is a typical web service environment that currently exists. The existing web service environment treats all transactions equally. Every transaction submitted to the DBMS scheduler is compiled into a schedule and submitted to the database. The drawback of this type of architecture is the case of a cascading rollback. In the event of a cascading rollback, operations that have successfully completed must be reverted, due to the failure of a dependant operation

in a separate transaction. The prediction-based architecture used in the presented solution adds a new logical component (transaction metrics). This component is added at the scheduling level to analyze certain metadata about transactions submitted to the scheduler. Table 2.1 displays the type of data that will be calculated in order to make an accurate prediction on the likelihood that a transaction will either commit or abort. Once implemented, transactions will build a history, or a reputation, based on commit rate and efficiency rate. Figure 2.7 displays the addition to the existing architecture in the prediction-based solution.

2.5.3 Transaction Metrics

The transaction metrics logical unit will contain all processing logic to accurately categorize the transactions submitted to the database. When a transaction is submitted to the scheduler to be added to a serializable schedule, the scheduler will look to the transaction metrics to determine the locking action that should be used for that particular transaction. All transaction metrics will be computed within the component before the scheduler issues a query so that the maximum time required for a response will be of complexity *O(1)*. These metrics are updated as the execution environment matures and the Transaction Categorization Graph (see Section 2.5.6) becomes more densely populated.

2.5.4 Lock Compatibility Among Transaction Categories

There are two main lock types in a DBMS: shared read-locks and exclusive write-locks. Shared read-locks allow read access to R, to be accessed by multiple parties. Resource R can be a relation, tuple, or even an individual cell in a database, depending on the lock granularity set by the system. Exclusive write-locks only allow write access to the particular resource for the transaction that holds the lock. At any point in

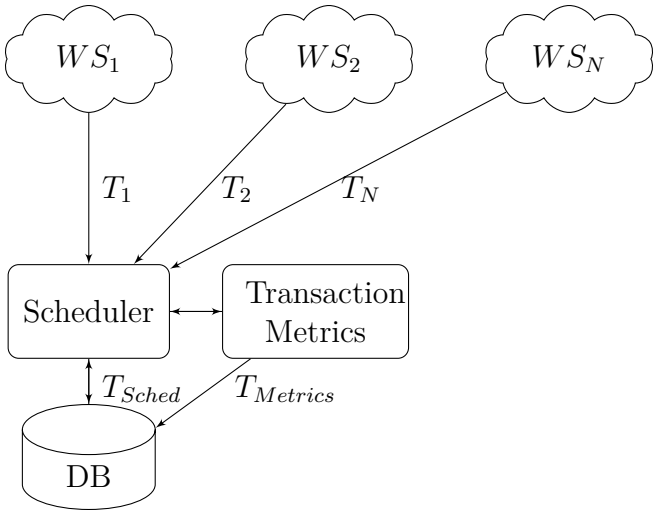


Figure 2.7: Web Service Environment with Transaction Metrics at Scheduling Level

time, in the execution of multiple concurrent transactions, there is only one exclusive write-lock per resource, and only the lock holder can manipulate the data.

To improve the performance of the system during the execution of concurrent transactions, we force lower priority transactional categories to release locks to higher priority categories. The system leverages the existing two-phase locking (2PL) protocol in order to obtain locks with the addition of prediction-based metrics. This allows transactions with a better reputation to execute more quickly, while transactions with a poor reputation do not create a bottleneck for later transactions. In order to successfully prioritize transactions, an objective priority was placed on each of the four transactional categories mentioned above. Definition 5 and Table 2.2 displays the transaction category dominance.

There are two compatibility matrices that were created as a result of the four transactional categories and two lock types. These matrices explicitly define the actions that should be taken when transactional categories request locks to the resources already granted locks. The actions defined within the matrices are derived based on the category prioritization made in Definition 5 and Table 2.2. Table 2.3 displays the lock compatibility for all transactional categories where a read-lock has already been

Table 2.3: Read-Lock Compatibility

	already granted lock			
requested lock	$HCHE_r$	$HCLE_r$	$LCHE_r$	$LCLE_r$
$HCHE_r$	+	+	+	+
$HCLE_r$	+	+	+	+
$LCHE_r$	+	+	+	+
$LCLE_r$	+	+	+	+
$HCHE_w$	-	δ	δ	δ
$HCLE_w$	-	-	δ	δ
$LCHE_w$	-	-	-	δ
$LCLE_w$	-	-	-	-

Table 2.4: Write-Lock Compatibility

	already granted lock			
requested lock	$HCHE_w$	$HCLE_w$	$LCHE_w$	$LCLE_w$
$HCHE_r$	-	δ	δ	δ
$HCLE_r$	-	-	δ	δ
$LCHE_r$	-	-	-	δ
$LCLE_r$	-	-	-	-
$HCHE_w$	-	δ	δ	δ
$HCLE_w$	-	-	δ	δ
$LCHE_w$	-	-	-	δ
$LCLE_w$	-	-	-	-

granted. Table 2.4 displays the lock compatibility for all transactional categories where a write-lock has already been granted. There are three actions that can be taken in regards to the lock compatibility matrix: *grant action*, *decline action*, and *elevate action* (see Definition 8). The next section will address the issue of multiple locks that are granted for a single resource.

2.5.5 Resource Category Data Structure

In Table 2.3, there are multiple *grant actions* (see Definition 8) that could potentially cause multiple read-locks to be granted for a single resource. It is appropriate for a resource to grant multiple read-locks; however, in order to properly handle the *elevate*

actions, the comparison must be made with the granted lock of the highest priority. For example, a resource R_a has granted two read-locks to transactions T_1 and T_2 with categories $HCLE_r$ and $LCLE_r$ respectively, while these two locks are still granted, a transaction T_3 categorized as $HCLE$ requests a write-lock $HCLE_w$ to R_a. If the evaluation based on Table 2.3 is completed by using the read-lock granted to T_2, then an *elevate action* would be issued, since the requesting lock contains a higher priority than the granted lock. However, if the evaluation is completed by using the read-lock granted to T_1, then a *decline action* would be issued, since the requesting lock contains a priority of equal standing with the granted lock.

When situations such as this arise in the system, the evaluation completed in the compatibility matrix should be completed against the granted lock with the highest priority. By evaluating against the granted lock whose transaction has been categorized with the highest priority, this prevents starvation of transactions with categorizations of lower priority. Using the example illustrated above, if the comparison was completed with transaction T_2 instead, then an *elevate action* would be issued and the locks of transaction T_2 would be preemptively dropped and, therefore, cause T_2 to compete for the lock again. If transactions are continually submitted to the system that are categorized with a higher priority than T_2, then locks obtained by T_2 will continually be dropped and will never successfully complete.

In order to ensure that Table 2.3 is used with the transaction categorized with the highest priority, a new data structure is introduced. This data structure is used in order to maintain knowledge of all granted locks for a particular resource. It also allows efficient access to the lock with the highest priority for comparisons. The data structure is a combination of two established data structures used throughout computer science: linked list and min-heap (min-priority queue). The first part of the data structure, the linked list, contains all resources in the system that have a read-lock granted. It is a linear singly-linked list to allow processing in one direction.

Each node in the linked list has a single reference to the root node of a min-heap. The min-heap contains all granted locks for the resource that has a reference to the min-heap root node. The min-heap property is calculated by the category priority of the locks that are granted. Since the highest priority, as shown in Table 2.2, is represented by the lowest integer value, then the lock with the highest priority will make its way to the root node of the min-heap by nature of min-heap properties Cormen et al. 2009, p.162. This accommodates efficient processing for accessing the highest priority of all locks granted to a particular resource. More details of how the Resource Category Data Structure is used are outlined in Section 2.6. Figure 2.8 displays a graphical representation of the Resource Category Data Structure.

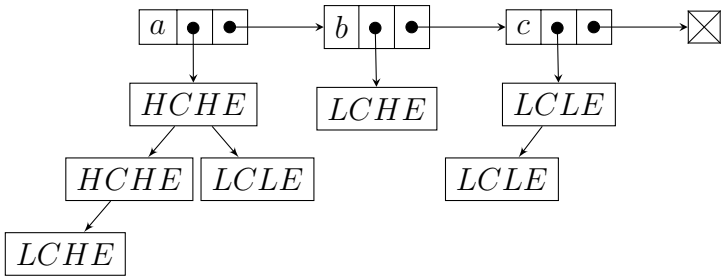


Figure 2.8: Resource Category Data Structure

2.5.6 Categorization Graph

The categorization graph is a graphical representation of the transaction metrics. It is grouped into four sections where each section represents a category that a transaction can be placed. Categorization bounds (see Definition 3) separate the graph into four sections and determine what category the transaction will receive once placed. Transactions are categorized based on the percentages of their previous execution metrics in comparison to the other transactions executing on the same system. This allows the system to flex accordingly with different environments. Depending on the metrics that each transaction possesses, the transactions will be categorized into the previous four categories listed above: $LCLE$, $HCLE$, $HCHE$, and $LCHE$. Trans-

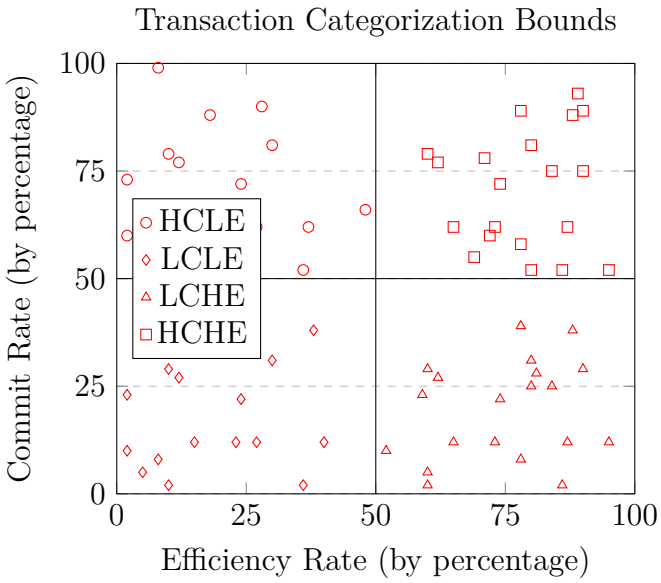


Figure 2.9: Categorization Graph

actions categorized as $LCLE$ must have an efficiency rate that is in the 50 percentile or lower where the commit rate is also in the 50 percentile or lower. Transactions categorized as $HCLE$ must have an efficiency rate that is also in the 50 percentile or lower, however, the commit rate must be in the 50 percentile or higher. $LCHE$ transactions must have an efficiency rate that is in the 50 percentile or higher where the commit rate is in the 50 percentile or lower. Transactions categorized as $HCHE$ must have an efficiency rate that is in the 50 percentile or higher where the commit rate is also in the 50 percentile or higher. Figure 2.9 and Definition 4 show the categorizations described above. The next section outlines the required algorithms for the prediction-based solution.

2.6 Algorithms

There are two algorithms needed in order to accomplish the goals of the prediction-based solution. The first algorithm determines the proper action required for a given operation. This is outlined in Algorithm 1. The second algorithm uses the functionality of the first algorithm in order to perform this action and execute the operations

accordingly. This is outlined in Algorithm 2. The next section explains the functionality of the algorithms and their complexity.

2.6.1 Determine Scheduler's Action for each Operation

The first algorithm needed for our solution (outlined in Algorithm 1) determines the action that our prediction-based solution is required to perform for a given operation. The algorithm's input parameters are the transaction being executed (T_i), two instances of the proposed RCDS (outlined in Section 2.5.5) for both READ and WRITE operations, the requesting lock for the given operation o, and the data item d that is is operating on ($l_{req} = (o, d)$). The algorithm's output after successful execution will be the matrix intersection of either the Read-Compatibility Matrix or the Write-Compatibility Matrix (outlined in Table 2.3 and Table 2.4), depending on the operation type. This intersection will be an action. There are three different actions the prediction-based solution leverages: *grant action*, *decline action*, and *elevate action*. These actions are defined in Definition 8.

The logic for determining the proper action for a particular operation begins by determining the type of operation requesting a lock on a particular data item (l. 2). From there we determine if the the RCDS for WRITE operations has a lock granted for the particular data item that the operation is requesting a lock for (l. 3). From this point, the logic depends on whether or not a READ or WRITE operation is requesting a lock. For the sake of covering the most complex branch of logic, we will discuss if the operation is a WRITE operation. After checking the RCDS for WRITE operations, we are then checking the RCDS for READ operations (l. 4). If both are empty, we update the RCDS and return a grant action to be taken (l. 6). When we update the RCDS, we make an entry in the the RCDS of the operation type with the data item that the lock is granted for and the category of T_i. This continual update ensures that the granted lock with the highest category is always referenced when

making comparisons. Continuing in the logic of the algorithm, if the RCDS for read operations is not empty then we compare the top category of the transaction that the operation is in (l. 7). If the operation's transaction has a category with a higher priority than the highest category of the granted lock then we update the RCDS as mentioned above and then return an elevate action (l. 9). Otherwise we return a decline action (l. 11). Throughout, the logic of the algorithm more comparisons much like the ones outlined above are executed in order to make sure the correct action is returned for processing. After successful execution of Algorithm 1 one of the three given actions will be returned for the scheduler to perform. The next algorithm outlines how that action is performed in the prediction-based scheduler.

2.6.2 Execute Schedules

The next algorithm (outlined in Algorithm 2) is responsible for executing the operation on the database, but before the operation can be performed, there are numerous conditions to be taken into consideration. This procedure begins by iterating through each operation contained within a serializable schedule. Before the operation can be executed, the action must be obtained. This is accomplished from the previous algorithm outlined in Algorithm 1 (l. 4).

If the action obtained is a *decline action* (l. 6), then by definition, the operation would wait until all locks were released, and then it would perform the operation. If the action obtained is an *elevate action* (l. 9), then all current locks granted for the operations resource would be dropped and then the operation would be executed. The final action, *grant action* (l. 12) requires no additional logic and the operation can be executed immediately.

Before each execution of any operation, an entry will be inserted into the RCDS with the resource and the category of the transaction. After each execution, the cor-

Algorithm 1 Determine Scheduler's Action

Input: $T_i, RCDS_{read}, RCDS_{write}, l_{req} = (o, d)$
Output: $RM(x, y)$ or $WM(x, y)$ *(intersection in matrices equates to an action)*

```
1: function DETERMINE_SCHEDULER_ACTION
2:     if o is a WRITE then
3:         if RCDS_write(d) is empty then
4:             if RCDS_read(d) is empty then
5:                 Update RCDS_write
6:                 return grant action (+)
7:             else if RCDS_read(top) < T_i then
8:                 Update RCDS_write
9:                 return elevate action (δ)
10:            else
11:                return decline action (-)
12:            end if
13:        else
14:            if RCDS_write(top) ≥ T_i then
15:                return decline action (-)
16:            else
17:                if RCDS_read(d) is empty then
18:                    Update RCDS_write
19:                    return elevate action (δ)
20:                else if RCDS_read(top) < T_i then
21:                    Update RCDS_write
22:                    return elevate action (δ)
23:                else
24:                    return decline action (-)
25:                end if
26:            end if
27:        end if
28:    else
29:        if RCDS_write(d) is empty then
30:            Update RCDS_read
31:            return grant action (+)
32:        else
33:            if RCDS_write(top) ≥ T_i then
34:                return decline action (-)
35:            else
36:                Update RCDS_read
37:                return elevate action (δ)
38:            end if
39:        end if
40:    end if
41: end function
```

Algorithm 2 Execute Schedule

Input: *Transaction T*
Output: N/A

```
 1: procedure EXECUTE_SCHEDULE
 2:     for all operations in T do
 3:         Obtain required action returned
 4:         from (Algorithm 1)
 5:
 6:         if action is a decline (-) action then
 7:             Wait until all locks are released.
 8:             Then execute operation.
 9:         else if action is an elevate (δ) action then
10:             Drop locks of lower priority
11:             transactions and execute operation
12:         else action is grant (+) action
13:             Execute operation
14:         end if
15:
16:         Release operation's locks and update RCDS
17:     end for
18:
19:     Update Transaction Metrics
20:
21:     Send aborted transactions back to
22:     scheduler to be rescheduled
23: end procedure
```

responding entry will be removed from the RCDS. This ensures the most appropriate action is always performed.

After all operations have completed processing, the transactional metadata for all operation executions is retained. This information is then used to populate the Transactional Metrics (l. 19).

The final step in the execution process is to reschedule any transactions that were preemptively aborted due to the *elevate action*. These transactions are entered back into the scheduler for rescheduling. This prevents any starvation that could potentially occur from preemptive transactional aborts (l. 22).

2.6.3 Complexity Analysis

Analyzing the algorithms used within the prediction-based solution ensures that the solution is feasible with the added overhead. The complexity analysis presented uses Big O notation for an upper bound of the algorithm's worst case scenario.

2.6.4 Algorithm 1: Determine Action for Operation

All operations within this algorithm execute in constant time, $O(1)$, and therefore, the overall complexity analysis of the entire algorithm will equate to $O(n)$.

2.6.5 Algorithm 2: DBMS Execute Schedule Algorithm

Algorithm 2 uses the logic from Algorithm 1 within its processing, and from the previous complexity analysis, we see that Algorithm 1 has a complexity of $O(n)$. Algorithm 2 also contains an overarching `for` that steps through each operation within the schedule provided to execute. There are no nested iterations within the overarching `for` which equates to $O(n)$ complexity in operations. Outside of the `for` there is an additional iteration for each data item recorded from the previous executions. With these two complexities combined the final complexity analysis is $O(n^2)$. Although the complexity is polynomial, any calling components will see the execution behave in constant time. This is made possible by concurrent execution of `EXECUTE_SCHED`.

2.6.6 Primary Contributor to Performance

After analyzing the algorithms used within the prediction-based solution, we see the overhead associated with the new solution is minimal. This is worth noting that the largest contributor to the overhead of the solution is not the added algorithmic complexity, but the waiting associated with restrictive concurrency control that this solution provides. As stated throughout the presented solution, locking ensures con-

sistency but can drastically degrade performance. In the next section we analyze the formal correctness of the algorithms used within the Prediction-based solution.

2.7 Analysis

In this section, we analyze the benefits of adding the prediction-based solution to the industry accepted solution of two-phase locking (2PL, Bernstein, Hadzilacos, and Goodman 1986, pp. 53-56). This analysis formally displays the correctness and feasibility of the prediction-based solution. This is accomplished by a step-by-step comparison of both solutions with the use case example outlined in Section 2.3.1.

From the use case example, let us say that $T_{GetOrderByUserID}$ is T_1 and $T_{DeleteOrderByUserID}$ is T_2. Figure 2.10 displays these transactions.

$$T_1 = R_1(a)R_1(b)C_1$$
$$T_2 = R_2(b)W_2(b)R_2(a)W_2(a)C_2$$

Figure 2.10: Example Transactions T_1 and T_2

From transactions T_1 and T_2 a schedule is generated to execute the two transactions concurrently. The schedule generated is legal and abides by all scheduling rules. To analyze the difference of a 2PL scheduler versus a 2PL scheduler with prediction-based metrics, we will assume the schedule generated is non-serializable. Figure 2.11 displays the generated schedule from the use case scenario outlined in Section 2.3.1.

$$T_{Schedule} = R_1(a)R_2(b)W_2(b)R_1(b)R_2(a)W_2(a)C_2C_1$$

Figure 2.11: Generated Non-Serializable Schedule

We outline the sequence of actions below when $T_{Schedule}$ executes only using 2PL. We will use this sequence as our base for comparison with our prediction-based solution. There are three cases that we will analyze; T_1 has a higher classification than T_2, T_1 has a lower classification than T_2, and T_1 has an equal classification T_2.

1. T_1 obtains a shared read-lock to resource a
2. T_2 obtains a shared read-lock to resource b
3. T_2 obtains an exclusive write-lock on resource b
4. T_1 waits for T_2 to release exclusive write-lock on resource b
5. T_2 obtains a shared read-lock to resource a
6. T_2 waits for T_1 to release shared read-lock on resource a
7. Deadlock!

2.7.1 T_1 with Higher Priority than T_2

The first case we will analyze is the case of T_1 having a higher classification than T_2. Let's suppose that T_1 has a category classification of $HCHE$ and T_2 has a category classification of $LCLE$. As you can see from the sequence above, using only 2PL with the given schedule ends in deadlock. This is caused by T_1 waiting for a lock that T_2 holds on resource b and T_2 waiting for a lock that T_1 holds on resource a. Neither can release their locks until they have obtained all the locks required for their transaction to complete. This, therefore, causes the schedule to halt in deadlock. However, if we were to add the prediction-based metrics to the existing 2PL protocol, we would get the following procedure instead:

1. T_1 obtains a shared read-lock to resource a
2. T_2 obtains a shared read-lock to resource b
3. T_2 obtains an exclusive write-lock on resource b
4. T_1 attempts to obtain shared read-lock on resource b:
    - The system gets the transaction with the highest category from the RCDS

- The system compares the category of the requesting transaction, T_1, with the category of the top of RCDS which is T_2
- T_1 contains the highest category, therefore, transactions with locks on resource b, T_2, will be dropped
- T_1 obtains a shared read-lock to resource b

5. T_1 growing phase is complete
6. T_1 executes successfully
7. T_1 releases all locks
8. T_1 shrinking phase is complete
9. T_2 is sent to scheduler for rescheduling
10. Execution complete!

The sequence of operations above prevents deadlock by removing the indefinite wait for resource b by T_1. Instead of a circular wait between transactions T_1 and T_2, we can use the prediction-based metrics to give precedence to the transaction of the higher category. This forces the transaction with a lower priority, T_2, to drop its locks and allows T_1 to finish successfully, therefore, preventing deadlock.

2.7.2 T_1 with Lower Priority than T_2

The second case we will analyze is the case of T_1 having a lower classification than T_2. Let's reverse the categories from the previous situation, and suppose that T_1 has a category classification of $LCLE$ and T_2 has a category classification of $HCHE$. In this case we get the following sequence:

1. T_1 obtains a shared read-lock to resource a

2. T_2 obtains a shared read-lock to resource b
3. T_2 obtains an exclusive write-lock on resource b
4. T_1 attempts to obtain shared read-lock on resource b:
    - The system gets the transaction with the highest category from the RCDS
    - The system compares the category of the requesting transaction, T_1, with the category of the top of RCDS, which is T_2
    - T_2 contains the highest category, therefore, T_1 will wait until the lock is released
5. T_2 obtains a shared read-lock to resource a
6. T_2 attempts to obtain an exclusive write-lock on resource a:
    - The system gets the transaction with the highest category from the RCDS
    - The system compares the category of the requesting transaction, T_2, with the category of the top of RCDS which is T_1
    - T_2 contains the highest category, therefore, transactions with locks on resource a, T_1, will be dropped
    - T_2 obtains a shared read-lock to resource a
7. T_2 growing phase is complete
8. T_2 executes successfully
9. T_2 releases all locks
10. T_2 shrinking phase is complete
11. T_1 is sent to scheduler for rescheduling
12. Execution complete!

In this case, the prediction-based solution prevents deadlock by elevating the lock precedence in T_2 and dropping the locks of T_1. This allows T_2 to execute successfully while T_1 is sent back to the scheduler to be rescheduled into another schedule.

2.7.3 T_1 with Equal Priority to T_2

The last and final case we will address is the case of T_1 having an equal classification of T_2. Although the classification holds no bearing in this case, we'll say that both T_1 and T_2 have a category of $HCHE$. The following sequence shows the actions taken in this case:

1. T_1 obtains a shared read-lock to resource a
2. T_2 obtains a shared read-lock to resource b
3. T_2 obtains an exclusive write-lock on resource b
4. T_1 attempts to obtain shared read-lock on resource b:
    - The system gets the transaction with the highest category from the RCDS
    - The system compares the category of the requesting transaction, T_1, with the category from the RCDS, T_2
    - T_2 contains an equal category therefore T_1 will wait until the lock is released
5. T_2 obtains a shared read-lock to resource a
6. T_2 attempts to obtain an exclusive write-lock on resource a:
    - The system gets the transaction with the highest category from the RCDS
    - The system compares the category of the requesting transaction, T_2, with the category from the RCDS, T_1
    - T_1 contains an equal category therefore T_2 will wait until the lock is released

7. Deadlock!

In this case we see that the prediction-based solution provides no added benefit that 2PL does not already address. However, the prediction-based solution performs exactly the same as 2PL would in this case.

2.7.4 Theoretical Contribution

In this section, we formally analyze the prediction-based solution and provide the associated theoretical contribution: assurance of consistency for all transactions while preserving an acceptable state of efficiency. The outline of the analysis is based very closely to the proof of correctness for two-phase locking (2PL) presented by Phil Bernstein Bernstein, Hadzilacos, and Goodman 1986, pp. 53-56.

In order to prove the correctness of the prediction-based solution we prove that each history generated by the prediction-based solution is serializable. However, before we can do that we must characterize the actions involved within the analysis. We say that $T_1 \rightarrow T_2$ (T_1 *precedes* T_2) if there are conflicting operations between the two serializable histories contained in $C(H)$. $C(H)$ is the set of all conflicting operations in a serializable history H. A *cycle* is present in the serializability graph $SG(H)$ if there exists a path $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$ within the schedule where $n > 1$. By showing that a cycle will never exist within $SG(H)$ we can show that the scheduler used within the prediction-based solution will always produce serializable histories with no cycles, and therefore, preserve consistency.

The prediction-based solution consists of three actions: grant action, decline action, and the elevate action (See Definition 8). We will analyze the actions involved in the situations described in Sections 2.7.1, 2.7.2, and 2.7.3.

Proposition 1. *Let H be a history produced by a prediction-based scheduler. If $o_i[x]$ requires a grant action (+) (outlined in Table 2.3 & Table 2.4) then $o_i[x] \notin C(H)$*

The grant action for an operation $o_i[x]$, according to Table 2.3 & Table 2.4, will only be required when the only locks granted and requested are read-locks. Therefore, there is no conflict and no operations contained within $C(H)$.

In the event that a decline action is required of an operation $p_i[x]$, then there exists one or many conflicting operations $q_j[x]$ that holds a lock to the required resource x. As mentioned in Algorithm 2, the scheduler will wait for all unlock operations $qu_j[x]$ before granting a lock operation $pl_i[x]$.

Proposition 2. *Let H be a history produced by a prediction-based scheduler. If $p_i[x]$ requires a decline action (-) (outlined in Table 2.3 & Table 2.4), then there exists an operation $q_j[x]$ $(i \neq j)$ and $p_i[x] \in C(H)$ and $q_j[x] \in C(H)$. Therefore, $qu_j[x] < pl_i[x] < p_i[x] < pu_i[x]$.*

The final two actions that could potentially be required for database operations are the *elevate action* and the *decline action*. These actions behave very similarly; however, the *elevate action* premptively drops the conflicting locks, so the new operation can obtain a lock to the resource, while the *decline action* waits for all conflicting locks to drop before locking the resource. Therefore, before an operation $p_1[x]$ can issue a lock operation $pl_1[x]$, all operations within the set $\{Q : q_1[x], q_2[x], .., q_n[x]\}$ must drop their locks. This causes all unlock operations $qu_i[x] < pl_1[x]$. Although these actions are different, in regards to serializability, they are identical, and therefore, they are combined into one proposition.

Proposition 3. *Let H be a history produced by a prediction-based scheduler. If $p_1[x]$ requires an eleveate action (δ) or a decline action (-) (outlined in Table 2.3 & Table 2.4) then there exists one or many $q_i[x]$ where $q_i[x] \in \{Q : q_1[x], q_2[x], .., q_n[x]\}$ and $Q \in C(H)$. Therefore, $\forall$ $q_i[x]$ in Q then $qu_i[x] < pl_1[x] < p_i[x] < pu_i[x]$.*

Now that we have formal propositions for each operation within the scheduler, we can formally prove serializability of the prediction-based solution. This proof is

done in three steps. First, we show that if $T_1 \rightarrow T_2$ is in $SG(H)$, then there exists a conflicting operation on a resource x where T_1 must have released its lock on x *before* T_2 was able to obtain a lock on x. The second step shows that for any path $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$ in $SG(H)$ we can show by transitivity that every T_i must release locks for conflicting operations before each T_{i+1}. The third and final step uses contradiction to assume that $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$. This contradiction shows that it is impossible for a cycle to occur because then T_1 would have performed an unlock operation before a lock operation. The following lemmas and theorem formalize these three steps.

Lemma 1. *Let H be a prediction-based history, and suppose $T_1 \rightarrow T_2$ is in $SG(H)$. Then, for some resource x and some conflicting operations $p_1[x]$ and $q_2[x]$ in H, $pu_1[x] < ql_2[x]$.*

Proof: By having $T_1 \rightarrow T_2$, there must exist conflicting operations $p_1[x]$ and $q_2[x]$ contained in $C(H)$ where $p_1[x] < q_2[x]$. Proposition 1 does not conflict in this case since there are conflicting operations. By looking at Proposition 3 we see,

1. $pl_1[x] < p_1[x] < pu_1[x]$
2. $ql_2[x] < q_2[x] < qu_2[x]$

Since the lemma states that $T_1 \rightarrow T_2$ we then have the operation sequence $pl_1[x] < p_1[x] < pu_1[x] < ql_2[x] < q_2[x] < qu_2[x]$, therefore, proving that $pu_1[x] < ql_2[x]$. □

Lemma 2. *Let H be a prediction-based history, and let $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$ be a path in $SG(H)$, where $n > 1$. Then, for some resources x and y, and some operations $p_1[x]$ and $q_n[y]$ in H, $pu_1[x] < ql_n[y]$.*

Proof: Rather than the previous proof by contradiction, we will now prove by induction. The base step, where $n = 2$, is proven in Lemma 1. We begin by supposing

the lemma will hold true for $n = k$ where $k \geq 2$ and $n = k + 1$. By induction we see the path $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_k$. Induction on the lemma shows that there exists resources x and z and operations $p_1[x]$ and $o_k[z]$ in H where $pu_1[x] < ol_k[z]$. To prove for $k + 1$ we use Lemma 1 for $T_k \rightarrow T_{k+1}$. Therefore, $\exists$ resource y and conflicting operations $o'_k[y]$ and $q_{k+1}[y]$ in H where $o'u_k[y] < ql_{k+1}[y]$. Proposition 3 shows that $ol_k[z] < o'u_k[y]$, and therefore, via the transitive property of operation precedence, $pu_1[x] < ql_{k+1}[y]$[6]. □

Theorem 1. *Prediction-Based schedulers (PBS) guarantees serializable execution when used with 2-phase locking (2PL).*

Proof: In order to show this is true, we use a proof by contradiction. We assume that, by contradiction, $SG(H)$ contains a cycle where $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$ and $n > 1$. The proof provided by the previous lemma (Lemma 2) states that for some resources x and y and conflicting operations $p_1[x]$ and $q_2[y]$ in H, $pu_1[x] < ql_2[y]$. However, this contradicts with Proposition 3 due to an unlock operation occurring on a resource before the lock operation[7]. Therefore, the contradiction fails and $ql_2[y] < pu_1[x]$. □

2.8 Experimental Simulation Results

The simulation model involved creating a test environment that could be run without user interaction. To accomplish the simulation a web-application was built using Java as the programming language and Spring Boot as the dependency injection framework. An in-memory H2 database solution was used in order to simulate data transactions within the application as well. Once the application was built, it was containerized into a Docker container and deployed on cloud computing resources

[6]Once again, Proposition 1 does not apply in this situation since there are conflicting operations.

[7]See footnote 6.

using Digital Ocean as the computing power. The simulation ran many hours generating thousands of results that were uploaded into an Amazon S3 bucket for retrieval. The results were pulled and analyzed using Tableau to discover the patterns and trends within the data. The experimentation setup involved running 3 solutions (the 2PL solution, the no-locking solution, and the prediction-based solution) with varying workloads through 7 different test cases (Table 2.5 outlines the seven test cases developed according to the percentage of each transaction category [Definition 4]). These test cases ensured that a wide range of testing workloads were covered in all categorization scenarios. To simulate the overhead of compensation transactions in comparison to the prediction-based scheduler we used the SAGA implementation pattern to calculate the additional execution time Garcaa-Molrna and Kenneth 1987. This pattern is built by using equal and opposite actions to each operation in a business process to revert the changes of an operation that fails. This metric allows us to simulate the overhead of a compensation transaction for each transaction that fails.

In analyzing the results, we generated 14 different plots (two for each test case). The plots visualize all executions across all three solutions in comparison to the workload. In each graph, there are three lines representing the comparison between the solutions. There are three levels of line thickness for each of the execution times increasing in the order of no-locking execution time, traditional 2PL execution time, and prediction-based execution time. The graphs appear to show only two lines due to the prediction-based solution and the traditional solutions execution time being so close. Figures 2.19 & 2.20 are another view of the same data showing the comparison of the prediction-based solution against the 2PL and the no-locking solution when consistency is lost and retained. In all cases, we see the execution time increase when consistency is lost in the no-locking solution (top graph), and execution time remains comparable to the 2PL solution when consistency is kept by the prediction-based solution (bottom graph). For additional graphs please see Appendix A.

In all test cases, we see that consistency was retained in both the 2PL solution and the prediction-based solution, but when the no-locking solution loses consistency, the efficiency decreases due to the compensation transaction. Our empirical results confirm that the performance of our prediction-based solution performs to the traditional 2PL locking solution with the added benefit of deadlock avoidance due to the lock action (see Definition 8) within the PBS.

Table 2.5: Simulation Test Cases

Simulation Test Cases				
Test Case #	**HCHE**	**HCLE**	**LCHE**	**LCLE**
1	100%	0%	0%	0%
2	75%	25%	0%	0%
3	50%	25%	25%	0%
4	25%	25%	25%	25%
5	0%	25%	25%	50%
6	0%	0%	25%	75%
7	0%	0%	0%	100%

2.9 Conclusion

In summary, transactions within a web service context have always been given a higher priority to efficiency over consistency, and for good reason. When web services are collaborated together by business process languages, the time it takes to complete can be a duration of hours and performance cannot be sacrificed. However, allowing the underlying database to reach an inconsistent state frequently is not acceptable. With the prediction-based solution, we ensure consistency without the performance hit of traditional locking. The three scheduler actions (*grant, decline,* and *elevate*) combined with the four transactional categories ($HCHE$, $HCLE$, $LCHE$, and $LCLE$) develop a solution that can easily be extended to a distributed web service context, in order to improve the current concurrency control mechanisms available today. Our ongoing work aims to prevent malicious transactions from corrupting databases in a web

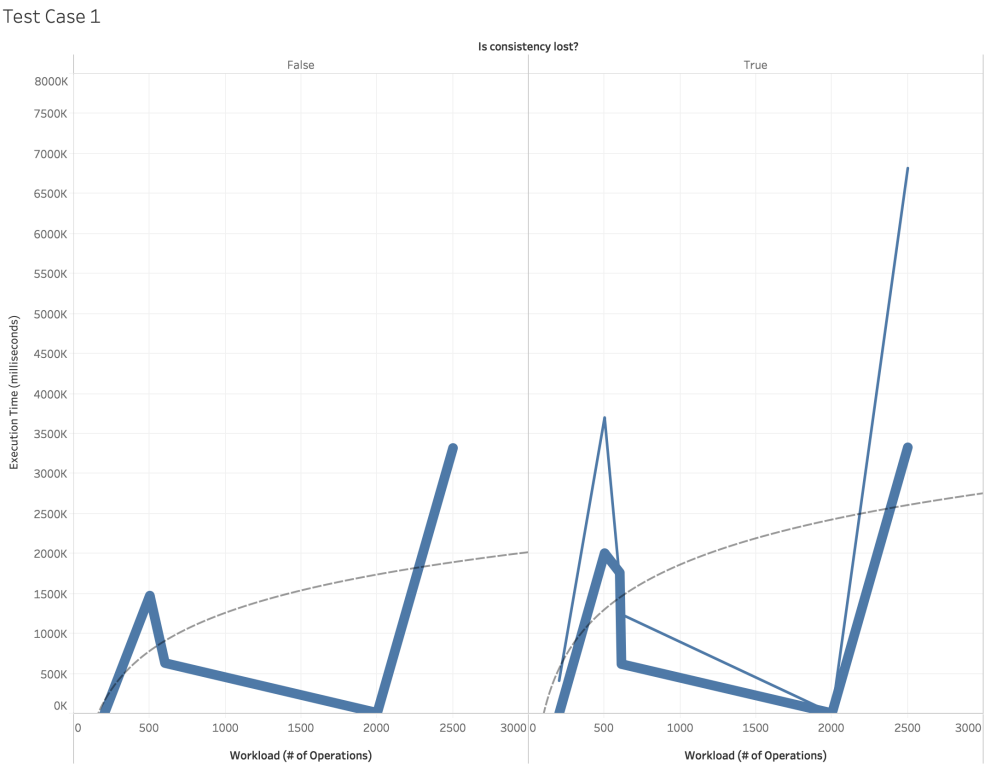


Figure 2.12: Simulation Results for Test Case 1

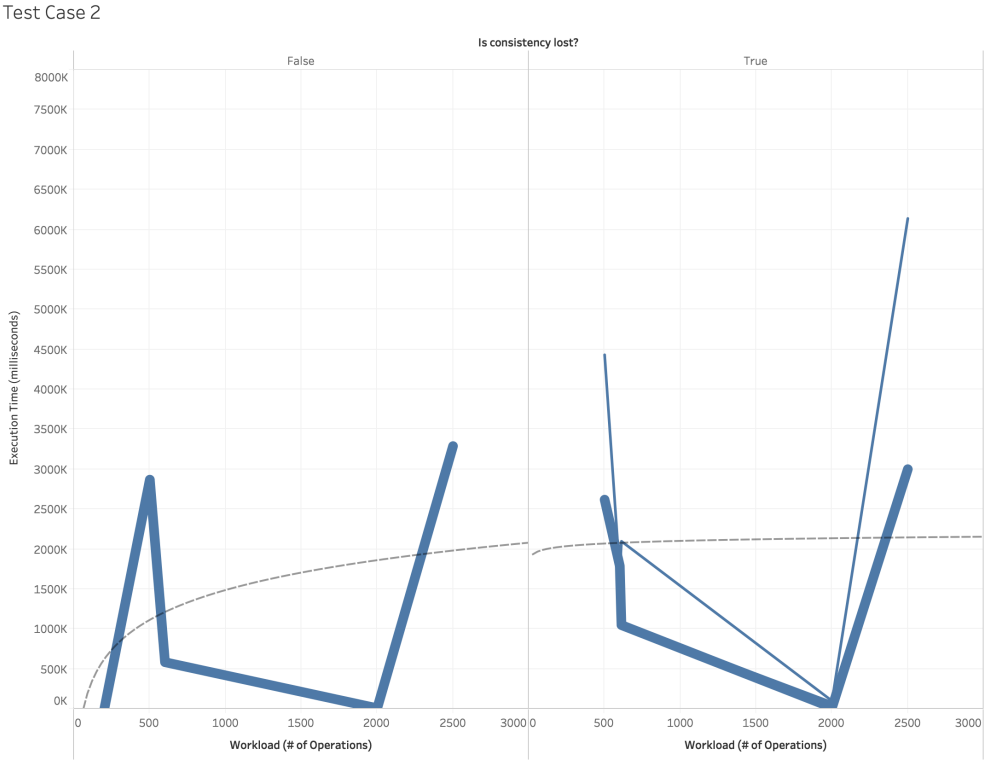


Figure 2.13: Simulation Results for Test Case 2

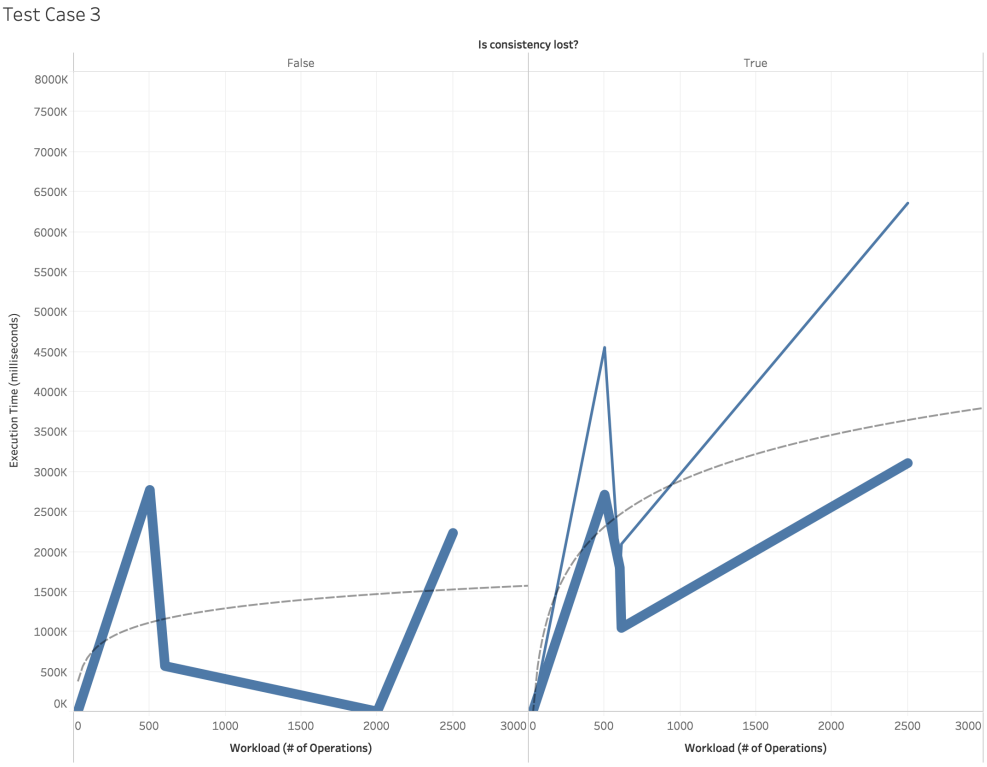


Figure 2.14: Simulation Results for Test Case 3

service environment. Liu and Jajodia proposed a multi-phase confinement system that provided a certain level of intrusion tolerance for database systems Liu and Sushil Jajodia 2001. We believe their model can be expanded using the metrics similar to the ones presented here.

2.9.1 Concluding Remarks

Now that we have established a foundation for prediction-based schedulers, we can now focus on the reputation of the transactions themselves. In the next chapter, this is where we will place our focus. Chapter 2 presented the solution and operated under the assumption that the reputation of the transactions were already established. Chapter 3 focuses on exactly how the transactions establish their reputation and also dynamically increase or decrease their reputation.

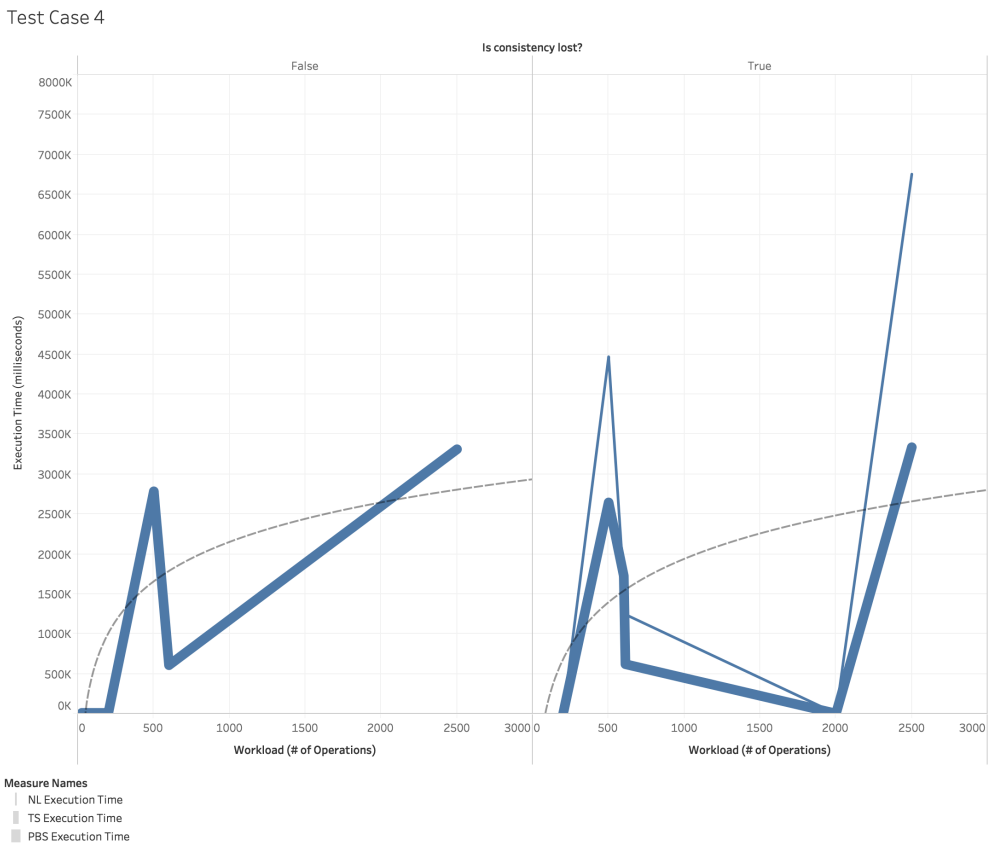


Figure 2.15: Simulation Results for Test Case 4

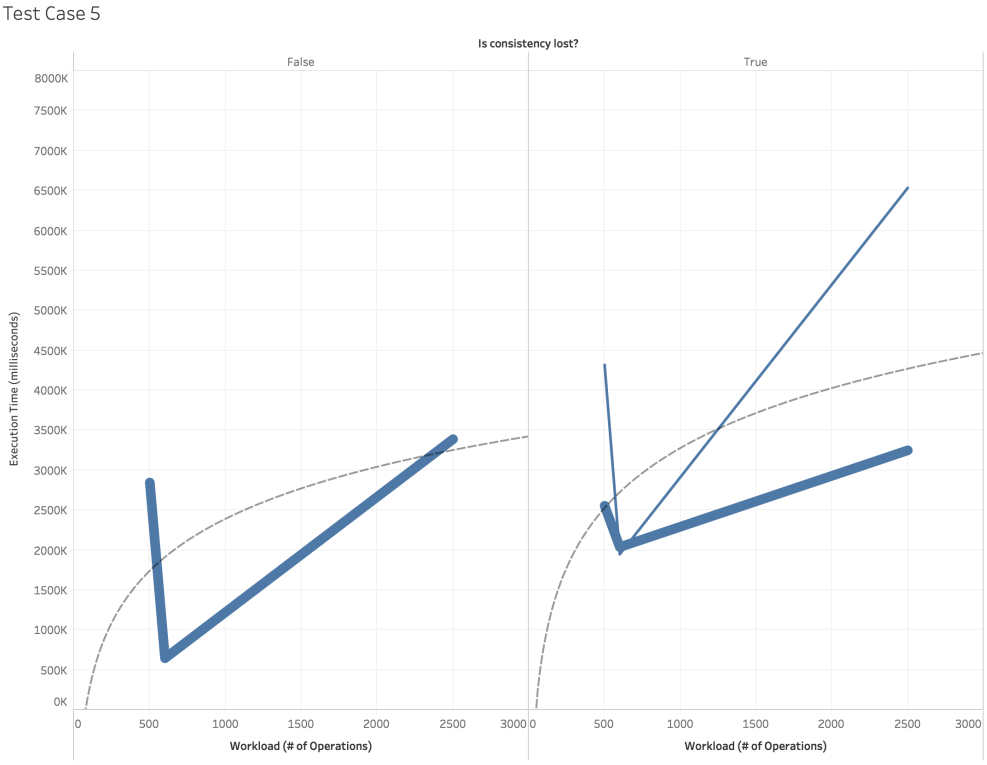


Figure 2.16: Simulation Results for Test Case 5

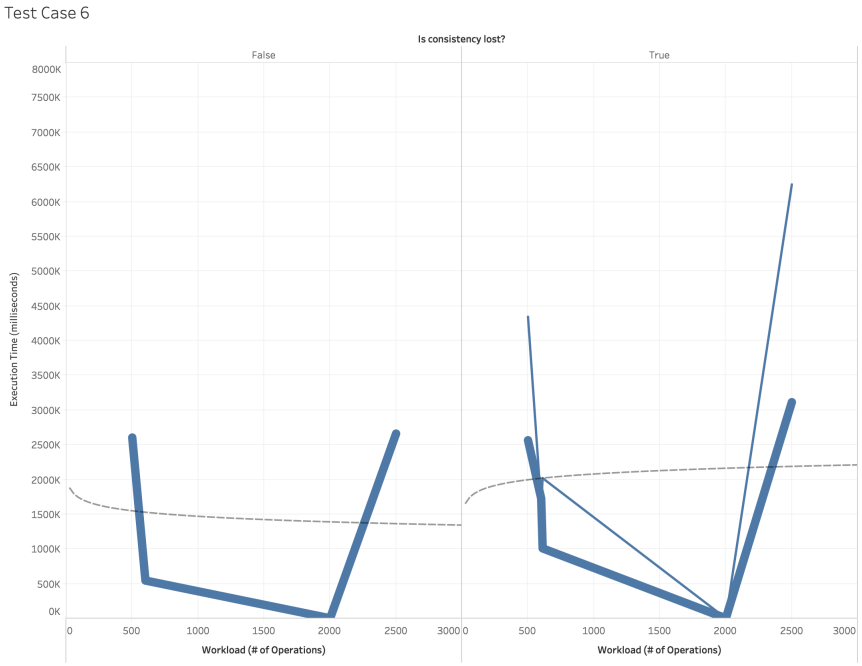


Figure 2.17: Simulation Results for Test Case 6

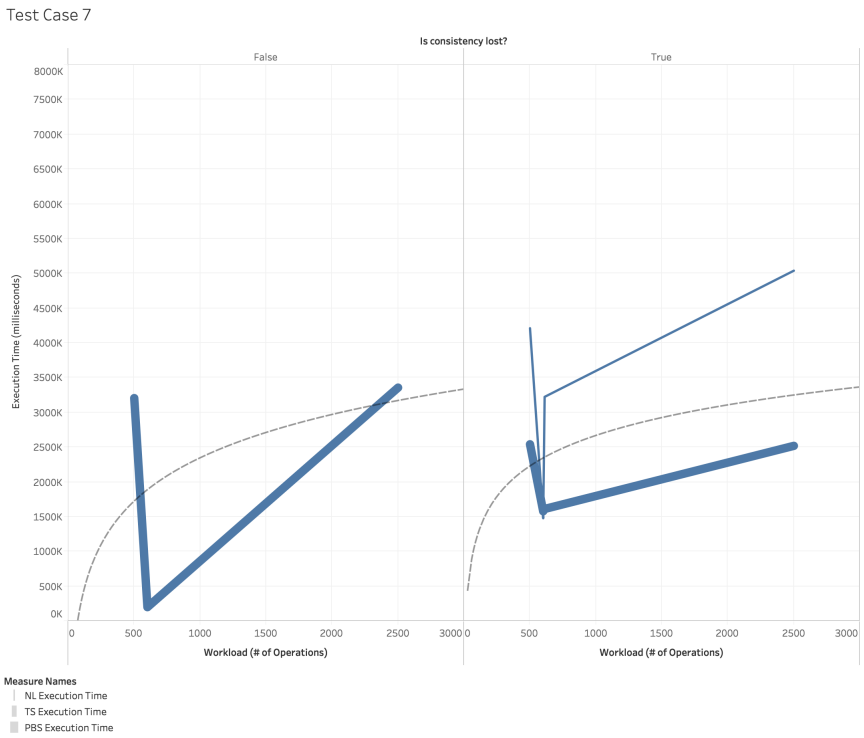


Figure 2.18: Simulation Results for Test Case 7

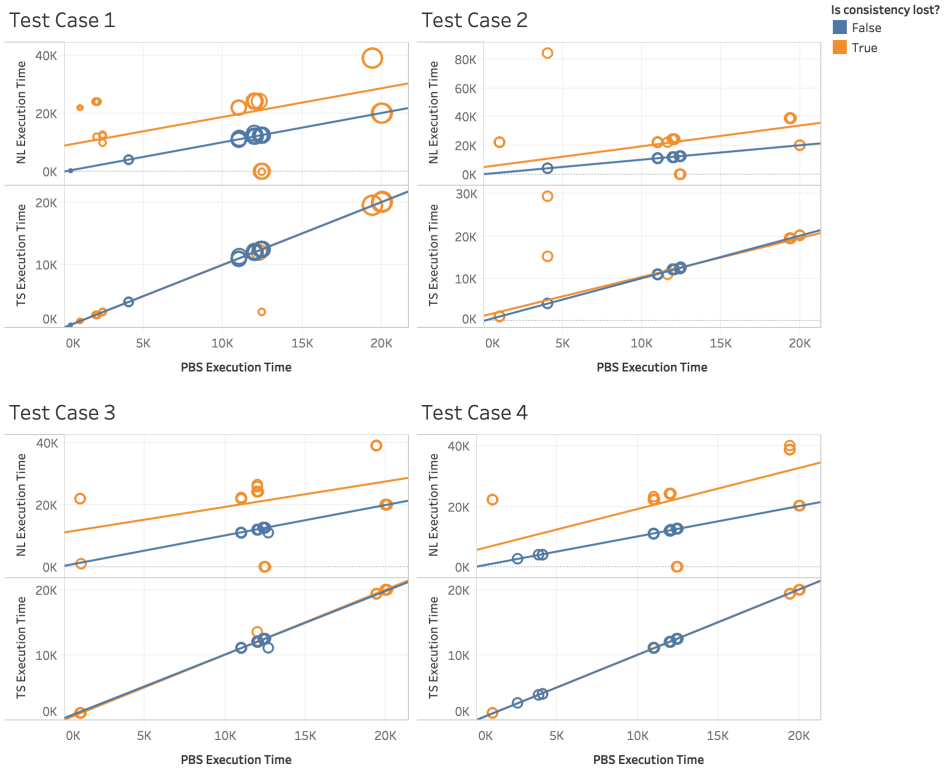


Figure 2.19: Consistency Lost/Kept for Test Cases 1-4

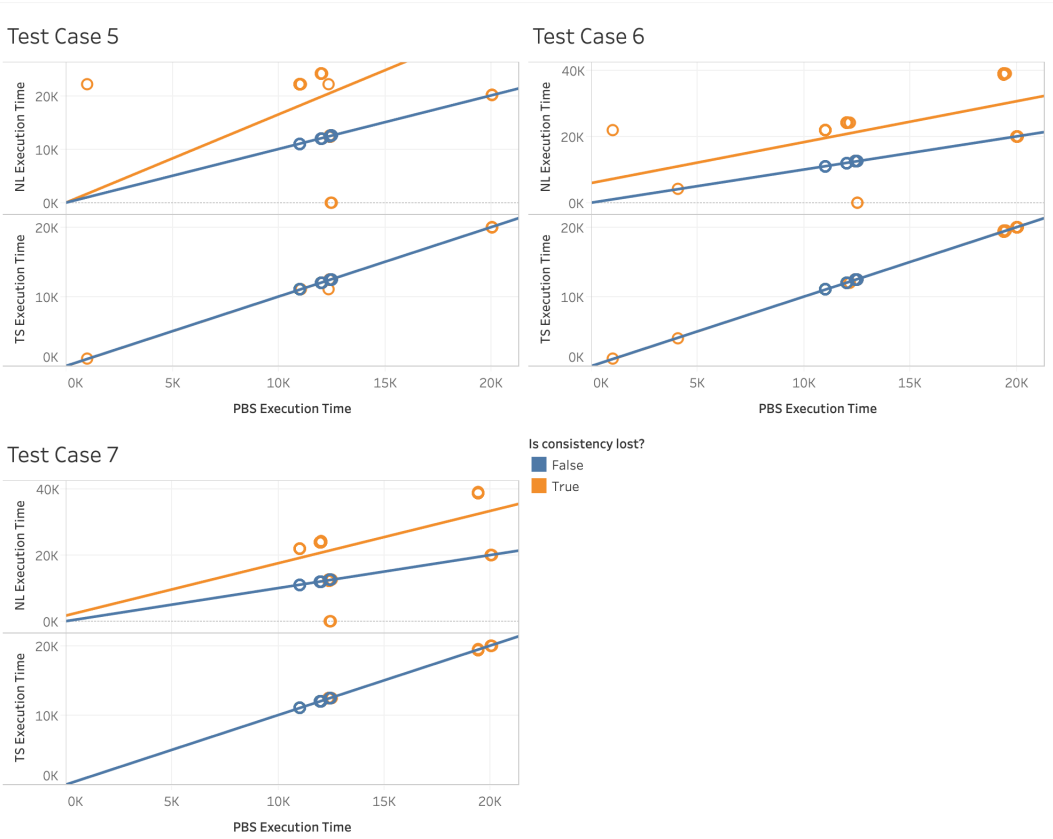


Figure 2.20: Consistency Lost/Kept for Test Cases 5-7

Chapter 3

Dynamic Transactional Reputation

3.1 Overview

In this chapter, we present the dynamic reputation management system. The work contributed here extends the existing prediction-based scheduler presented in Chapter 2. The work here in Chapter 3 is submitted to ACM Transactions on Database Systems and awaiting review.

3.2 Introduction

Concurrency control is a problem that has always been at the forefront of researchers for some time now. In traditional database systems, the schedulers provide ACID properties to transactions that are executed to ensure consistency and correctness. While this is the ideal solution for all transaction scheduling, it's not feasible when transactions are moved to a web service context. The overhead of the locking within web service transactions eliminates the use of traditional scheduling techniques. In order to increase efficiency of many concurrent transactions, the transactional properties of atomicity and isolation are relaxed to prevent the overhead of locking. While this increases efficiency of concurrent transactions, this places the database at a much higher risk of reaching an inconsistent state where data needs to be repaired. The current industry standard is to abandon locking and generate compensating transactions to fix the effects when consistency is lost however, compensating transactions can be very expensive when multiple conflicts occur. In previous work (Ravan, Banik, and

Farkas 2020) we developed a transaction scheduler that provided a prediction-based analytic to the transaction executing. We used transaction metrics from previous executions to place the transaction in a hierarchical category where we then provided targeted locking. For those transactions that were considered well-behaving, we allow concurrent executions with no locking but those that could potentially cause a cascading rollback, we provided locking to ensure that no other transactions were affected. Going forward in this work we'll simply refer to the prediction-based scheduler as PBS for brevity.

In this work, we build upon the PBS by focusing on the reputation management of the transactions. We use bit-wise scoring based on commit ranking (see Definition 11), efficiency ranking (see Definition 12), user ranking (see Definition 13), and system ranking (see Definition 14) of transactions entering the system. The bit-wise score is then used in a linear fashion to determine locking behaviors for those transactions. Higher scores receive precedence over lower scores therefore allowing for a more granular scheduler that was previously restricted to four categories (see Ravan, Banik, and Farkas 2020).

The scores are calculated after each execution so that the most recent execution's metrics can be taken into consideration. This also prevents adding overhead to the transaction scheduler when transactions enter the system. The score will have already been calculated and a scheduling decision can be made at execution time. The following work details the reputation management system. This chapter is organized in the following order; Section 3.3 outlines the problem along with a use-case scenario. Section 3.4 discusses the existing research that has already taken place in regards to the problem. Section 3.5 outlines the system model for the solution. Section 3.6 illustrates the simulation results gathered from the prototype.

3.3 Problem Definition

In order to define the problem of stale transaction categories we must first discuss the problem of concurrent database transactions in a web service environment.

In traditional database systems, transactions are executed with ACID properties to ensure correctness, durability, and consistency among all transactions that are executed on the system. When transactions are moved to a web service context where concurrent transactions occur frequently, the traditional model of transaction correctness is not feasible to deploy. Multiple interleaving transactions with the locking required in ACID systems causes an overhead that is not acceptable for the end user. In order to accommodate concurrent transactions in a web service environment that execute in an acceptable time frame, locking is removed and transactions are allowed to execute and commit independently. While all transactions are executing and committing successfully then there are no solutions and the lack of locks works. However, in the event that a transaction fails and there are transactions that are dependent downstream then a cascading rollback occurs reverting the effects of the downstream transaction. All transactions are put on hold until a compensation transaction, generated by the scheduler to fix the results of the failed transaction, can execute successfully. This causes a lot of overhead in the system that can be avoided if the failed transaction can be isolated from dependent transactions.

In our previous work (see Ravan, Banik, and Farkas 2020) we presented a prediction-based transaction scheduler that provides appropriate run times for web service environments. The scheduler predicts the outcome of a transaction based on the transactions execution history. The transaction is then placed into one of four categories based on whether commit rate and execution time. We then provided custom locking actions based on the transactional category. Transactions in categories with high commit rates and low execution times are allowed to execute concurrently while trans-

actions in categories with low commit rates and long execution times are locked to prevent downstream effects.

Our previous work addresses the problem of cascading rollbacks and compensation transactions, however a new problem presents itself in a transaction that has been incorrectly categorized or its metrics have changed and it needs to be re-categorized. This becomes a problem when a transaction with a high commit rate is locked due to its category when it can execute concurrently without any undesired side effects. The other extreme and more disastrous use case is a transaction with a low commit rate that should be locked but executes concurrently with other transactions and causes those transactions to abort their executions. In these situations we need the ability to promote or demote a transaction as its execution history changes.

As we discuss the use of the transaction's execution history, another problem presents itself; what do we do with transactions that are new to the system and do not have any execution history? If we are to address the problem of transactions with no execution history then a reputation score should take the place of the previous four category solution. An objective reputation score allows for a linear approach for transaction comparisons that provides two benefits; a more granular comparison (i.e. comparing transactions that would normally be within the same category and would previously conflict) and a default score for a transaction with no history. In the previous four category solution, there is no default category for transactions with no history.

In our previous solution, we defined three different locking actions; grant (+), decline (-), and elevate (δ). The decline action causes a transaction to wait for resources to become available. Due to the restrictive four categories of our previous solution, there are a large number of scenarios where a decline action would occur. By transitioning to a solution that is more granular, we could potentially turn decline actions into elevate actions. The elevate action will abort a transaction of a lower

category in order for a transaction of a higher category to be granted access to needed resources. By design this prevents transactions that are well-behaving from being hindered by transactions that are not well-behaving. However, if we don't include the cost of an aborted transaction in our calculation then we could do more harm than good by elevating too frequently. Transitioning to a new metric will allow us too refine our rules for elevation to prevent elevate actions that cause more harm than benefit. See Tables 2.3 and 2.4 for reference. In the next section we walk through an airline ticketing use-case scenario to better explain the problems identified.

3.3.1 Use-Case Scenario

In order to better explain the problems, let's look at a use case scenario of an airline ticketing system. Let's say we have two users that are attempting to reserve seats on an airline. The first user places a ticket, or a seat, in their shopping cart using an airline's online reservation system. While in the shopping cart, that seat is no longer available for reservation even though the transaction has not been completed. Simultaneously, a second user attempts to reserve a seat on the same airline and same flight. There are no seats available so that user is denied a reservation. Later on, the first user that placed the initial reservation in their shopping cart does not purchase the reservation in time. Their reservation is then expired and made available again.

In this scenario, the seat is made available due to the reconciliation efforts of the reservation system, however, in the end the airline loses profit. A seat that could have been purchased by the second user was not available due to the first user having placed the seat in their shopping cart. See the scenario diagram in Figure 3.1. Figure 3.2 shows the same scenario with the transaction scheduler and database contained within the same logical unit. In both figures the solid lines represent the user transactions submitted to the database while the dotted lines represent the response from the

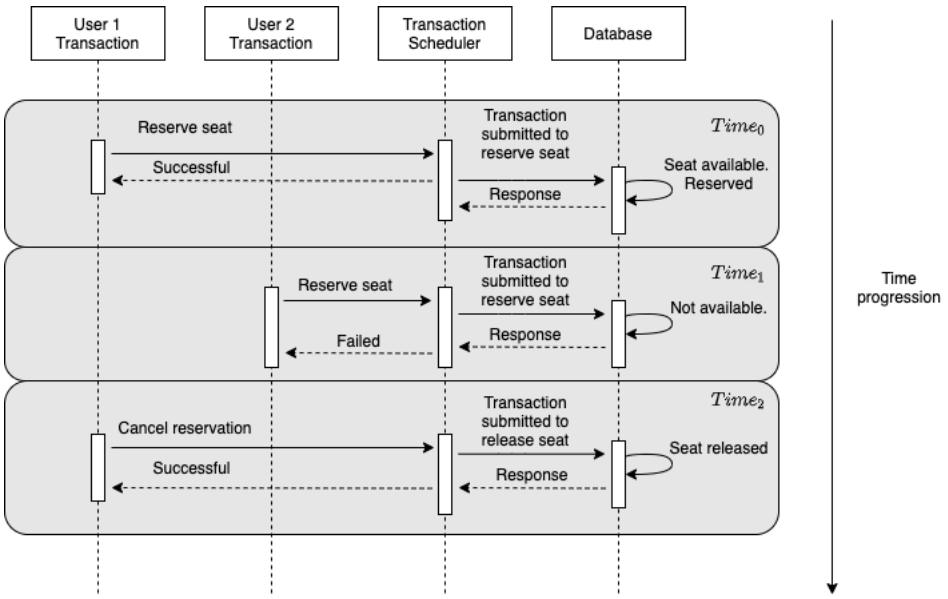


Figure 3.1: Airline Reservation Use Case

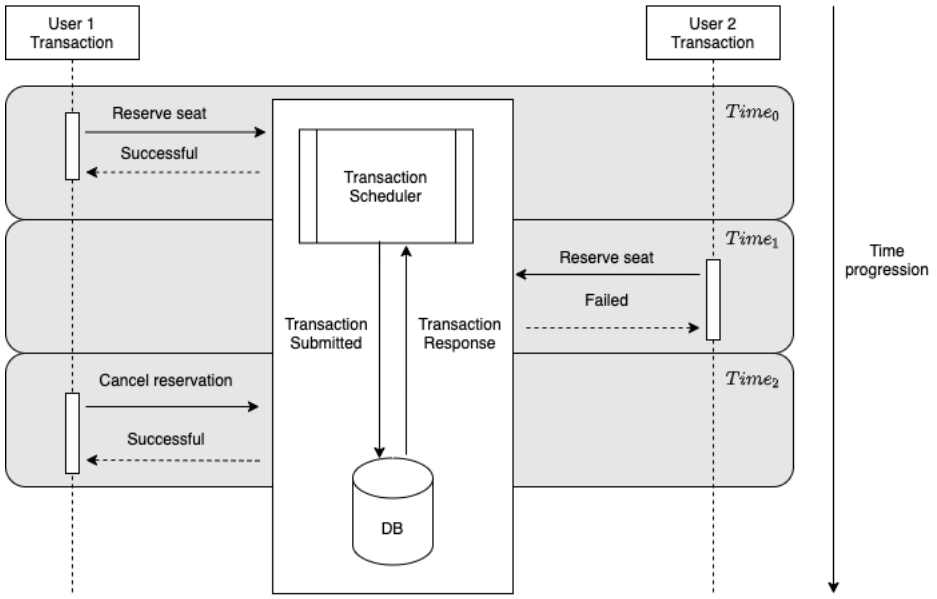


Figure 3.2: Airline Reservation Use Case System Model

transaction. The gray swim lanes labeled T_0, T_1, and T_2 show the transactions at certain time intervals.

If the given scenario were to play out in a system that maintained the reputation of transactions entering the system, then the behavior of the first user would be tracked would be taken into consideration. The next time the user were to submit a similar

transaction, that reputation would be taken into consideration and could potentially prevent the user from getting precedence over the seat reservation. This reinforces good user behavior in the system and also increases profit for the reservation system. In the next section, we discuss the related work that influenced the current problem and research.

3.4 Related Work

There are many resources available for dynamic reputation management that shaped our understanding of the challenges of maintaining reputation of nodes/services. Many of the reputation systems were centered around maintaining a reputation in a distributed or decentralized system. These resources were Clark, Stewart, and Hopkinson 2017, De Paola and Tamburo 2008, and Hu et al. 2010. While this helped our understand of the challenges of reputation management, our system would not experience the challenges associated with a decentralized trust management solution since the service and the database are both contained within the prediction-based scheduler itself. Other reputation management solutions were focused on the reputation or trustworthiness of web pages. Those solutions were presented in Melnikov et al. 2018, J. Wang, Peng, and D. Zhang 2008, and O. Q. Zhang et al. 2012. These were also helpful but didn't address the reputation of database transactions directly which is the focus of the reputation of the prediction-based solution. The majority of reputation management systems are provided with the context of P2P or ad-hoc mobile networking in mind. Resources focused on dynamic reputations within networking environments are Chiejina, Xiao, and Christianson 2014, De Paola and Tamburo 2008, Hu et al. 2010, and Y. Sun and Zhao 2019.

We also looked at various multi-agent reputation systems to understand how their reputation management systems were leveraged. Reputation management within multi-agent systems, web service selection, and e-commerce has been researched more

in depth than reputation within database systems. This gave us a different perspective into reputation management that we could transfer to our database solution. The solutions we reviewed were "Multiagent reputation management to achieve robust software using redundancy" by Rajesh Turlapati (see Rajesh Turlapati and Huhns 2005), "Multiagent System for Reputation–based Web Services Selection" by Wang et. al. (see H. Wang et al. 2006), and "Swarm Intelligence Based Reputation Model for Open Multi Agent Systems" by Mahmood et. al. (see Mahmood et al. 2006).

Understanding how the user plays a role with the outcome of the transaction is a key piece of our solution. Looking at the work presented by Lomet et. al. (Lomet, Vagena, and Barga 2006) motivated our inclusion of the User Ranking (see Definition 13) in our solution.

To better understand data lineage and how it can be used as a tool for building transactional reputations we looked at our previous research of data provenance and malicious transactions in Theppatorn et. al. (Rhujittawiwat et al. 2021).

A great deal of the existing work related to concurrency control (e.g., Alrifai et al. 2009, Bailis et al. 2014, Dai, Yang, and B. Zhang 2009, Feingold and Jeyaraman 2009, Ferreira et al. 2012, Freund and Little 2009, Gao and Wu 2005, Greenfield et al. 2007, Jang, K. Fekete, and Greenfield 2007, E. Lee et al. 2001, K.-W. Lee and Kim 2000, Little and Wilkinson 2009, Olmsted 2015, and Riegen et al. 2010) influenced our motivation for this work.

After studying the current environment of dynamic reputation systems, we believe this is a great opportunity to provide a dynamic reputation management solution particularly focused at database transactions within a web service environment.

3.5 System Model

This section covers the system model. Here we discuss the definitions, implementation, and environment involved in the dynamic reputation solution.

3.5.1 Definitions

This section outlines the definitions on which the solution is built. These definitions will be used to describe the different components that consist of the system model.

Definition 9. *(Compatibility) - a data item d_i within transaction T_i is locked in a non-compatible mode if:*

1. *d_i is locked by write-lock*
2. *$Dominates_S(T_i, T_j) = false$, or*
3. *$Dominates_W(T_i, T_j) = false$*

Definition 10. *(Legal Scheduler) - a prediction-based scheduler is legal if:*

1. ***Grant:*** *(denoted as the addition symbol, +) a transaction is permitted to lock a data item if the data item is not already locked in a non-compatible mode (see Definition 9) by an other transaction*[1]*.*
2. ***Decline:*** *(denoted as the subtraction symbol, -) a transaction T_i is denied to lock a data item if the data item is already locked in a non-compatible mode (see Definition 9) by another transaction T_j where $\tau(T_i) \leq \tau(T_j)$*
3. ***Elevate:*** *(denoted as lowercase delta, δ) a transaction T_i is permitted to lock a data item if all the non-compatible locks (see Definition 9) on the data item are being held by transactions $T_1, ..., T_k$ such that $\tau(T_j) < \tau(T_i)$ for all $j = 1, ..., k$ in this case $T_1, ..., T_k$ must first release the locks on the data item before T_i is permitted to lock the data item.*

[1] In the event that a lock is requested of a resource that has not issued any locks, then the lock will be automatically granted. There is no conflict, and therefore, the compatibility matrices do not apply.

Definition 11. *(Commit Ranking) - the commit ranking of a transaction T_i, denoted as CR_{T_i}, is based on the number times a transaction commits compared to other transactions in the system. If C_{T_i} is the number of commits for transaction T_i and T_c represents the subset of all transactions T_{all} with smaller or equal number of commits than T_i then the commit ranking is calculated based on the given formula:*

$$T_c = \{\, T \in T_{all} \mid C_T \leq C_{T_i} \,\}$$

$$CR_{T_i} = \frac{|T_c|}{|T_{all}|}$$

Example 11.1. *Let $C_{T_i} = 500$, $|D| = 450$, and $|T_{all}| = 600$ then $CR_{T_i} = 0.75$*

Definition 12. *(Efficiency Ranking) - the efficiency ranking of a transaction T_i, denoted as ER_{T_i}, is based on how its execution time $time_{T_i}$ compares to all transactions executed within the execution environment. If T_e represents the subset of all transactions T_{all} with larger or equal execution time than T_i then the efficiency ranking is calculated based on the given formula:*

$$T_e = \{\, T \in T_{all} \mid time_T \geq time_{T_i} \,\}$$

$$ER_{T_i} = \frac{|T_e|}{|T_{all}|}$$

Example 12.1. *Let $time_{T_i} = 400s$, $|T_e| = 30$, and $|T_{all}| = 250$ then $ER_{T_i} = 0.12$*

Definition 13. *(User Ranking) - the user ranking of a user in the system denoted as UR_i is based on the user's ranking among other users and their impact on the commit rate (see Definition 11) via forced aborts. If FA_{U_i} is the amount of forced aborts caused by user U_i and U_a represents the subset of all users U_{all} with more or equal number of forced aborts than U_i then UR_i is calculated on the given formula:*

$$U_a = \{\, U \in U_{all} \mid FA_U \geq FA_{U_i} \,\}$$

$$FA_{U_i} = \frac{|U_a|}{|U_{all}|}$$

Example 13.1. *Let* $FA_{U_i} = 4$, $|U_a| = 15$, *and* $|U_{all}| = 24$ *then* $UR_i = 0.625$

Definition 14. *(System Ranking) - the system ranking of a transaction* T_i, *denoted as* SR_{T_i}, *is the transaction's ranking among other transactions based on the number of times the transaction has been aborted by the system via elevate actions (see Definition 10) or any system causes. If* A_{T_i} *represents the number of system aborts for transaction* T_i *and* T_a *represents the subset of all transactions* T_{all} *with more or equal number of system aborts than* T_i *then* SR_{T_i} *is calculated on the given formula:*

$$T_a = \{\, T \in T_{all} \mid A_T \geq A_{T_i} \,\}$$

$$SR_{T_i} = \frac{|T_a|}{|T_{all}|}$$

Example 14.1. *Let* $A_{T_i} = 7$, $|T_a| = 10$, *and* $|T_{all}| = 320$ *then* $SR_{T_i} = 0.03125$

Definition 15. *(Reputation Score) - a Reputation Score of a transaction* T_i *denoted as* RS_{T_i} *is a 4-tuple of the calculated attributes for transaction* T_i *and user* U_{T_i}. *The attributes are the commit ranking, efficiency ranking, user ranking, and the system ranking multiplied by a weighted value* $w_i^{1...4}$ *that change the precedence of a particular score value. The Reputation Score is denoted as follows:*

$$RS_{T_i} =< w_i^1 \times CR_{T_i}, w_i^2 \times ER_{T_i}, w_i^3 \times UR_i, w_i^4 \times SR_{T_i} >$$

where CR_{T_i} *is the Commit Ranking (see Definition 11),* ER_{T_i} *is the Efficiency Ranking (see Definition 12),* UR_i *is the User Ranking (see Definition 13), and* SR_{T_i} *is the System Ranking (see Definition 14).*

Example 15.1. *Assuming* $w_i^{1...4} = 1$*, for a transaction* T_i *and a user* U_i *let* $CR_{T_i} = 0.35$*,* $ER_{T_i} = 0.75$*,* $UR_i = 0.23$*, and* $SR_{T_i} = 0.85$ *then* $RS_{T_i} =< 0.35, 0.75, 0.23, 0.85 >$

Definition 16. *(Strong Dominance) - Let* $RS_{T_i} =< w_i^1 \times CR_{T_i}, w_i^2 \times ER_{T_i}, w_i^3 \times UR_i, w_i^4 \times SR_{T_i} >$*, and* $RS_{T_j} =< w_j^1 \times CR_{T_j}, w_j^2 \times ER_{T_j}, w_j^3 \times UR_j, w_j^4 \times SR_{T_j} >$ *denote the reputation scores of transactions* T_i *and* T_j *respectively. Transaction* T_i *has strong dominance over transaction* T_j*, denoted as* $Dominates_S(T_i, Tj)$*, if and only if the following conditions are true:*

1. $w_i^1 \times CR_{T_i} \geq w_j^1 \times CR_{T_j}$,
2. $w_i^2 \times ER_{T_i} \geq w_j^2 \times ER_{T_j}$,
3. $w_i^3 \times UR_i \geq w_j^3 \times UR_j$*, and*
4. $w_i^4 \times SR_{T_i} \geq w_j^4 \times SR_{T_j}$

Example 16.1. *Assuming* $w_i^{1...4} = 1$ *and* $w_j^{1...4} = 1$*, for a transactions* T_i *and* T_j *and users* U_i *and* U_j *let* $RS_{T_i} =< 0.35, 0.75, 0.23, 0.85 >$ *and* $RS_{T_j} =< 0.15, 0.55, 0.03, 0.45 >$ *then* $Dominates_S(T_i, Tj)$

Definition 17. *(Weak Dominance) - Let* $SUM(T_i) = CR_{T_i} + ER_{T_i} + UR_i + SR_{T_i}$ *and* $SUM(T_j) = CR_{T_j} + ER_{T_j} + UR_j + SR_{T_j}$ *denote the sum of the attributes of the reputation scores for transactions* T_i *and* T_j *respectively. Transaction* T_i *has weak dominance over transaction* T_j*, denoted as* $Dominates_W(T_i, Tj)$*, if and only if the following conditions are true:*

1. $Dominates_S(T_i, T_j) = false$,
2. $Dominates_S(T_j, T_i) = false$*, and*
3. $SUM(T_i) \geq SUM(T_j)$

Example 17.1. *Assuming* $w_i^{1...4} = 1$ *and* $w_j^{1...4} = 1$*, for a transactions* T_i *and* T_j *and users* U_i *and* U_j *let* $RS_{T_i} =< 0.35, 0.75, 0.23, 0.65 >$ *and* $RS_{T_j} =< 0.15, 0.76, 0.03, 0.85 >$ *then* $Dominates_S(T_i, Tj) = false$ *but* $SUM(T_i) = 1.98$ *and* $SUM(T_j) = 1.79$ *therefore* $Dominates_W(T_i, Tj)$

Definition 18. *(Not Comparable) - Transaction* T_i *is not comparable to transaction* T_j *if and only if the following conditions are true:*

1. $Dominates_S(T_i, T_j) = false$*,*
2. $Dominates_S(T_j, T_i) = false$*,*
3. $Dominates_W(T_i, T_j) = false$*, and*
4. $Dominates_W(T_j, T_i) = false$

Example 18.1. *Assuming* $w_i^{1...4} = 1$ *and* $w_j^{1...4} = 1$*, for a transactions* T_i *and* T_j *and users* U_i *and* U_j *let* $RS_{T_i} =< 0.35, 0.75, 0.23, 0.65 >$ *and* $RS_{T_j} =< 0.25, 0.76, 0.12, 0.85 >$ *then* $Dominates_S(T_i, Tj) = false$*.* $SUM(T_i) = 1.98$ *and* $SUM(T_j) = 1.98$ *and* $Dominates_W(T_i, Tj) = false$ *therefore* T_i *has equal dominance over* T_j

3.5.2 Reputation Score

In order to construct the reputation of a transaction we construct a 4-tuple score (see Definition 15) based on the four transactional attributes of commit ranking (see Definition 11), efficiency ranking (see Definition 12), user ranking (see Definition 13), and system ranking (see Definition 14). Each of the four values associated with the reputation score is a normalized value between 0 and 1 that ranks the transaction among all transactions executing within the system. Each value also contains a weighted multiplier that can be used in order to reward or place a higher precedence on a particular attribute for a particular transaction. If the multiplier is equal to one then all attributes contribute to the score equally.

Previously, the solution only kept commit rate and efficiency rate into consideration when categorizing transactions. This created a four category system that wouldn't allow dominance of transactions within the same category. By using the reputation score, we can now establish dominance of transactions that would originally have been in the same category. The next section discusses this dominance structure.

3.5.3 Strong & Weak Dominance

In our previous work of PBS we presented the transaction dominance lattice (see Figure 3.3) that establishes dominance of transactions that were categorized. While the prediction-based solution adds categorization to provide a solution against conflicting transactions, there is still the limitation that two transactions of the same categorization can be in conflict. In this situation, the prediction-based solution reverts to existing 2PL with no added benefit.

$$HCHE > HCLE > LCLE$$

$$HCHE > LCHE > LCLE$$

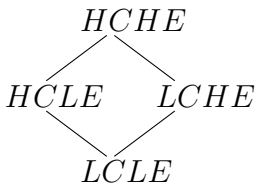


Figure 3.3: Transaction Category Dominance

By transitioning to a solution that contains a continuous spectrum of categorization, we can avoid the situation where two transactions of the same categorization cause a conflict. Tables 2.3 & 2.4 show the existing rules for the four category solution. The existing operations would apply given the rules of dominance (see Definitions 16,

17, and 18) to execute and eliminate situations in which transactions of the same category would normally conflict.

Strong Dominance (see Definition 16) is established when all four attributes of a reputation score of a transaction (commit ranking, efficiency ranking, user ranking, and system ranking) contain a value that is greater than all the values of the conflicting transactions reputation score. This the most preferred and easiest way to establish dominance between two conflicting transactions.

If Strong Dominance cannot be established then there is the ability to still obtain a "tie breaking" situation with Weak Dominance (see Definition 17). Weak Dominance can be established by taking a sum of all four attributes of each transaction's reputation score and doing a numerical comparison of the overall score. If the score of the conflicting transaction is greater than or equal to the other transaction, then Weak Dominance is established preventing a stalemate.

If neither Strong or Weak Dominance can be established then we have reach a state of Not Comparable (see Definition 18). If the two conflicting transactions are deemed Not Comparable then the precedence of the two transactions will take priority and the conflicting transaction mus wait for the first transaction to complete.

3.5.4 Locking Actions with Reputation Score

Now that we have a system for calculating a reputation score for all transactions that have entered the system, we can use that score and refine our existing locking actions. Tables 2.3 and 2.4 show the existing rules for when the three locking actions should be used. These rules were generated when we only had a four-category system. Now that we have reputation score, we need a much more refined formula that defines when to use the elevate action (δ) and the decline action (-).

In the previous model, an elevate or decline action would occur in situations where there was a conflict and the conflicting transactions (see Definition 6) were in different

categories. This did not take into account how closely similar or vastly different the two transactions were. It was very black and white could potentially cause extreme overhead due to excessive elevate actions.

In the new model that we present here, we would elevate the transaction that possesses either Strong or Weak Dominance over the conflicting transaction.

Let T_1 and T_2 be two conflicting transaction and $A = \{decline(-), elevate(\delta)\}$ be the set of locking actions for conflicting transactions. For two conflicting transactions, let the mapping τ below display actions taken assuming that T_1 is the transaction with precedence. Therefore:

$$\tau \rightarrow \left\{ \begin{array}{ll} elevate(\delta) & Dominates_S(T_1, T_2) = \text{false, and} \\ & Dominates_W(T_1, T_2) = \text{false} \\ decline(-) & Dominates_S(T_1, T_2) = \text{true} \\ decline(-) & Dominates_W(T_1, T_2) = \text{true} \end{array} \right\}$$

This model doesn't apply to grant (+) actions since there is no conflict and no need for conflicting locking action. In the next section we discuss how the reputation score is recalculated and rewards are applied.

3.5.5 Reputation Score Recalculation & Reward

The Reputation Score for a transaction allows us to prioritize transactions based on their transactional attributes. However, we need the ability to recalculate the reputation scores so that the correct actions are being taken. We want to be able to recalculate without the recalculation being a burden on the system with intense overhead.

In order to recalculate the reputation scores in such a way that it does not incur a lot of overhead, we recalculate based on the percentage of transactions that have been aborted. When greater than 10% of the transactions end execution via an

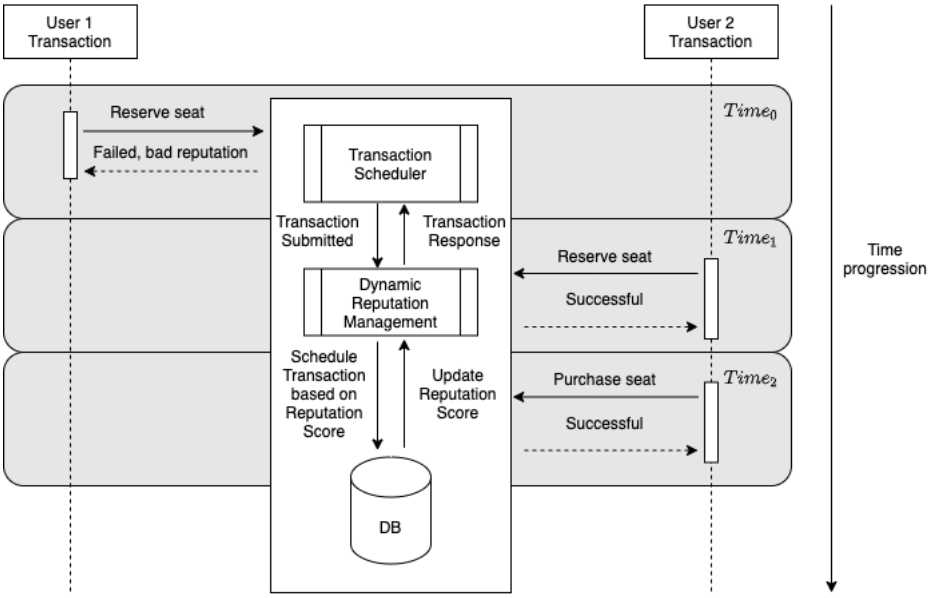


Figure 3.4: Use Case with Dynamic Reputation Management

abort within the system, then we recalculate their reputation scores in order to avoid situations that will cause transactions additional overhead or a premature abort.

Recalculating the scores based on this percentage allows the system to make correct decisions regarding locking actions without adding an additional burden to the system. The frequency of score recalculation is then dynamic and changing based on the needs of the system rather than a static time frame.

3.5.6 Use Case with Dynamic Reputation Management

Now if we take a look at the use case outlined in Section 2.3.1 with the Dynamic Reputation Management solution, we can prevent the behavior of user 1 from affecting the other users in the system. Figure 3.4 shows the use case with the reputation management solution embedded. In this scenario, the bad reputation of user 1 doesn't affect user 2. We have tracked the reputation of user 1 and therefore prevent user 1 from reserving the seat. This then allows the seat to be available for user 2 to reserve and purchase.

3.6 Empirical Results

Here we discuss the setup and execution of the simulation for dynamic reputations. The prototype is used to generate simulation results and verify results of the executions.

3.6.1 Application

In this section, we discuss the application environment used for executing transactions.

The application is written using Spring Boot and Java. Spring Boot is an application framework that allows Java applications to be containerized easily and manage dependencies within the application itself.

Within the application we have four transaction schedulers implemented

- No-Locking Scheduler (NoSQL)
- Traditional Scheduler (2PL)
- Prediction-Based Scheduler'
- Dynamic Reputation Scheduler

Each scheduler is executed with the same execution parameters. Those parameters include

- Users
- Transactions
- Rate of abort
- Rate of conflict
- Use Case (discussed in next section)

Each scheduler executes in its own dedicated thread and we record results for each execution for analysis. Table 3.1 is a schema diagram of the results that we capture for each transaction execution. Before we started running the schedulers we generated users and transactions with random rankings between 0 and 1 for an initial working set. We generated over 5800 users and over 11000 transactions. When the recalculation percentage threshold is reached we recalculated all of the rankings based on our definitions of each ranking (see Section 3.5.1). This recalculation takes place in its own thread to prevent blocking the schedulers from continuing with their executions.

3.6.2 Use Case Formulation

We have experimented with a variety of use cases to fine tune our empirical results. Table 3.2 shows the use cases that were executed initially to get a baseline of how transactions within the system would be affected, when a recalculation would occur, and what was the impact of that recalculation. Table 3.3 shows our final use case selection for the performance measurement of our approach.

For the graphs in this section we use the term **affected transactions** to identify transactions that were forced to abort by our scheduler.

Our first preliminary use case (Use Case Alpha shown in Figure 3.5) started with a high recalculation percentage to get a baseline test of the prototype. The high recalculation percentage (50%) caused that initially, the affected transactions spiked to approximately 3%, and then began to level off as the number of transactions within the system increased. We observed that with the 50% affected transactions rate, recalculation would never be triggered.

Our next preliminary use case (Use Case Beta shown in Figure 3.6) decreased the recalculation rate to 10% while increasing the conflicting and abort percentages in the system to 25%. This was enough of a change to cause a single recalculation in

Table 3.1: Execution History Attributes

Attribute Name	**Description**
userid	Unique identifier for the user
user_ranking	Decimal value containing user ranking defined in Definition 13
transactionid	Unique identifier for the transaction
commit_ranking	Decimal value containing user ranking defined in Definition 11
system_ranking	Decimal value containing user ranking defined in Definition 14
eff_ranking	Decimal value containing user ranking defined in Definition 12
num_of_operations	Integer value of the number of operations in the transaction
reputation_score	String representation of all rankings together
transaction_exec_time	Decimal representing to the total execution time in milliseconds
percentage_aborted	Decimal representing to percentage of aborted transactions over the total transactions during that particular execution
recalculation_needed	Boolean representing if the recalculation threshold was surpassed
time_executed	Timestamp representing the time of the execution
dominance_type	String representing what type of dominance was established (see Definitions 16, 17, and 18)
transaction_outcome	String representing whether the execution committed, aborted, or aborted due to higher dominance
scheduler_type	String representing which scheduler submitted the execution
use_case	String representing what use case this execution was a part of
category	String representing the category of the transaction if it was an execution from PBS
transaction_type	String representing whether it was as normal or compensation transaction

Table 3.2: Initial Use Cases

Name	Total #	Recalculation %	Conflict %	Abort %
Use Case Alpha	≈ 23,000	50	10	5
Use Case Beta	≈ 2,500	10	25	25
Use Case Gamma	≈ 1,900	5	40	40
Use Case Delta	≈ 1,200	7	50	50

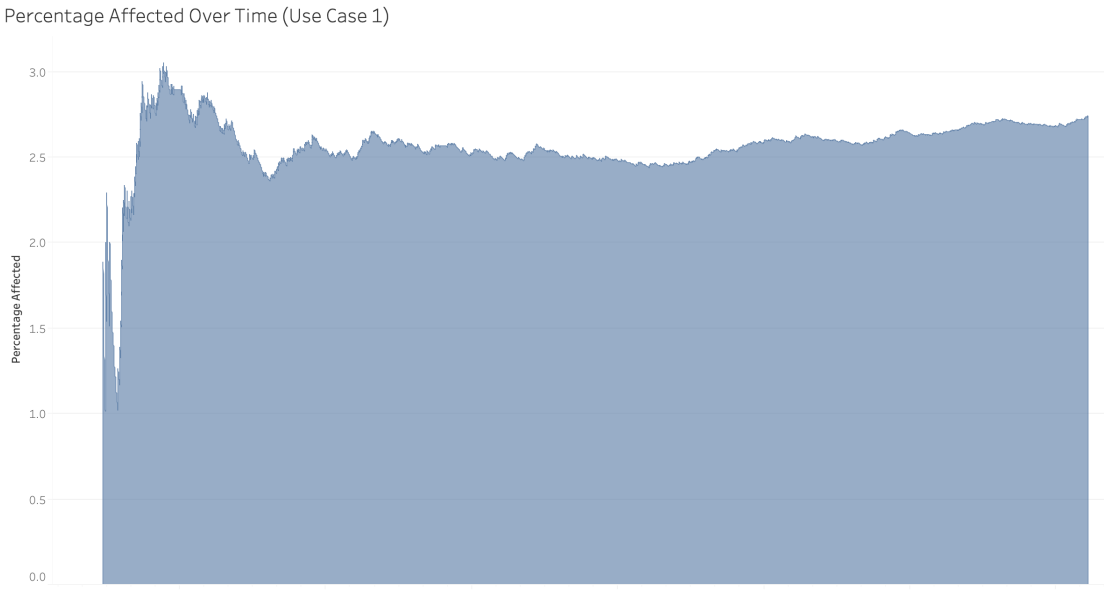


Figure 3.5: Use Case Alpha

the beginning. As the number of transactions increased in the system the number of affected transactions would grow too slowly to initiate another recalculation.

After seeing the results from Use Case Beta we set the parameters of the next execution to be a bit more exaggerated. Use Case Gamma (shown in Figure 3.7) lowers the recalculation percentage to 5% while the conflicting and abort percentages were set much higher at 40%. This caused, that the affected transactions quickly spiked to 100%; which kicked off a recalculation of the rankings within the system. After the recalculation the numbers are reset, but the conflicting and abort percentages are so high that the execution continues spiking the affected transactions to 100%. This repetition of spikes continues throughout the entire execution. While Use Case Gamma

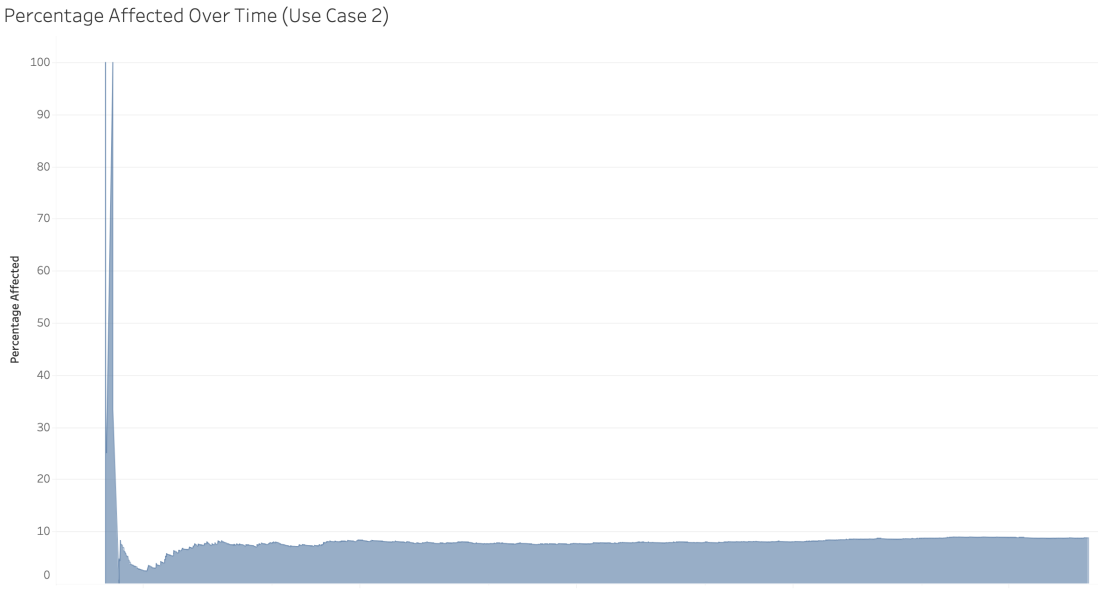


Figure 3.6: Use Case Beta

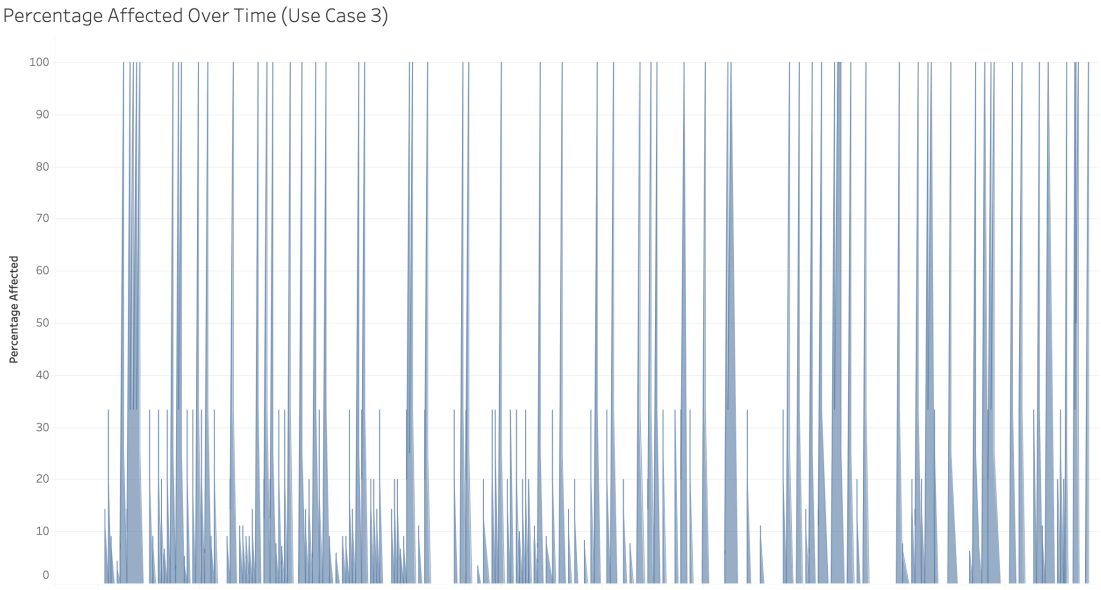


Figure 3.7: Use Case Gamma

is not an ideal system that transactions will execute within, it gave us a upper bound of what happens when there is a large percentage of affected transactions.

The final use case of our initial executions was Use Case Delta (shown in Figure 3.8). This use case is very similar to the previous use case and the results show that as well. In this use case we increase the recalculation percentage to 7% while also

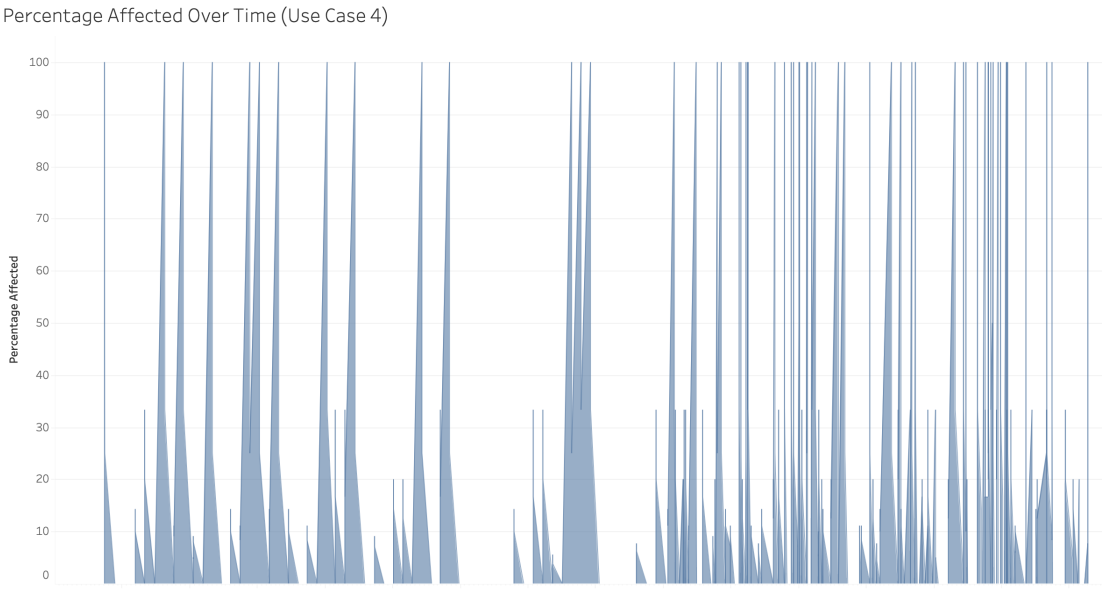


Figure 3.8: Use Case Delta

increasing the conflicting and abort percentages to 50%. There is nothing significant to note here other than the findings met our expectations for the given the parameters.

After seeing the results of the preliminary use cases we formulated the use cases documented in Table 3.3. In the next sections we present the use cases and our empirical results.

3.6.3 Use Cases

In this section, we discuss the use cases that were used to simulate executions within the schedulers. Input parameters collected as a set make up a single use case.

Use cases are the parameters used for a specific execution. Use cases have the following attributes

- **Name**
  - A simple name to uniquely identify the use case
- **Total Transactions Executed**

Table 3.3: Use Cases Executed

Name	Total #	Recalculation %	Conflict %	Abort %
Use Case 1	5,000	30	0	10
Use Case 2	5,000	30	50	10
Use Case 3	5,000	30	25	10
Use Case 4	5,000	30	20	10
Use Case 5	5,000	30	22	10
Use Case 6	5,000	30	75	10
Use Case 7	5,000	30	100	10

  - An integer representing how many transactions were executed within that use case

- **Recalculation Percentage**

  - This is a decimal number representing the percentage of aborted transactions over all transactions that is the threshold of when a recalculation should occur throughout the system

- **Conflicting Percentage**

  - This is a decimal number representing the percentage of transactions that will conflict during execution.

- **Abort Percentage**

  - This is a decimal number representing the percentage of transactions that will end in an abort during execution.

These act as the input parameters for the executions as each set of use case parameters cause a different load on the system. The varying load consists of how many times a recalculation occurs and how many times a transaction must be executed again due to conflict or an abort. Table 3.3 shows the use case parameters that were used for simulation.

Before the use cases executed in Table 3.3 there were numerous use cases executed as discovery metrics for the best results generation. The use cases listed in Table 3.3 outline the optimal executions to outline our contribution.

In the next section we discuss how the experimentation was executed, limitations of the experimentation, and the goal of the experimentation.

3.6.4 Execution

The goal of this section is to provide clarity around the execution of the experimentation and outline the goal of the experimentation in regards to our contribution.

The experimentation discussed in Sections 3.6.1 and 3.6.3 outlines the application architecture and the use cases used as a part of the execution to submit the solution to different workloads. The execution itself (shown in Figure 3.9) involves the execution of all four schedulers simultaneously with the same workload.

The primary limitation of the execution involves the number of transactions and users executed each time. The schedulers are not designed as fully functional schedulers that can accept multiple transactions and users but are subsets of the schedulers that only accept two users and two transactions each time. The primary goal of the scheduler executions is to validate the algorithms of the dynamic reputation scheduler among other comparative schedulers. Writing subsets of the schedulers that only accept pairs of transactions and users was much more feasible to implement as a prototype rather than a fully functional system. The flow of execution for the dynamic reputation scheduler is outlined in Figure 3.10 and the flow of the prediction based scheduler implemented in our previous work (see Ravan, Banik, and Farkas 2020) is outlined in Figure 3.11. The 2PL and NoSQL schedulers implement the standard algorithms that are defined.

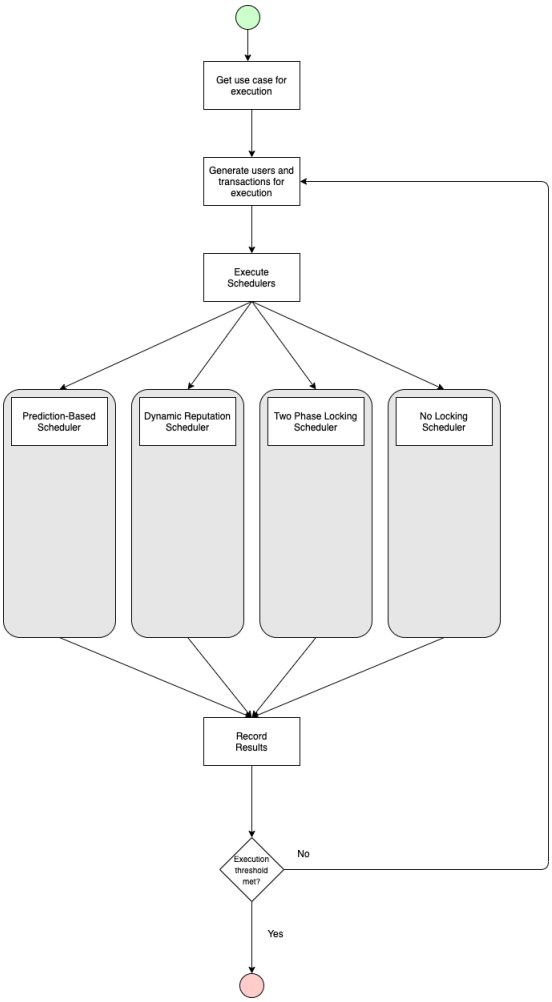


Figure 3.9: Flow of Execution

3.7 Analysis

When looking at the behavior of all the use case executions we did notice trends given the parameters that were used.

Use Cases 1 and 2 performed very similar with the parameters given. In both use cases, the number of aborted transactions never reached above 10% of the total transactions therefore a recalculation of rankings was never executed. As more transactions executed within the system, the more the percentage of aborted transactions plateaued and we never saw a recalculation take place.

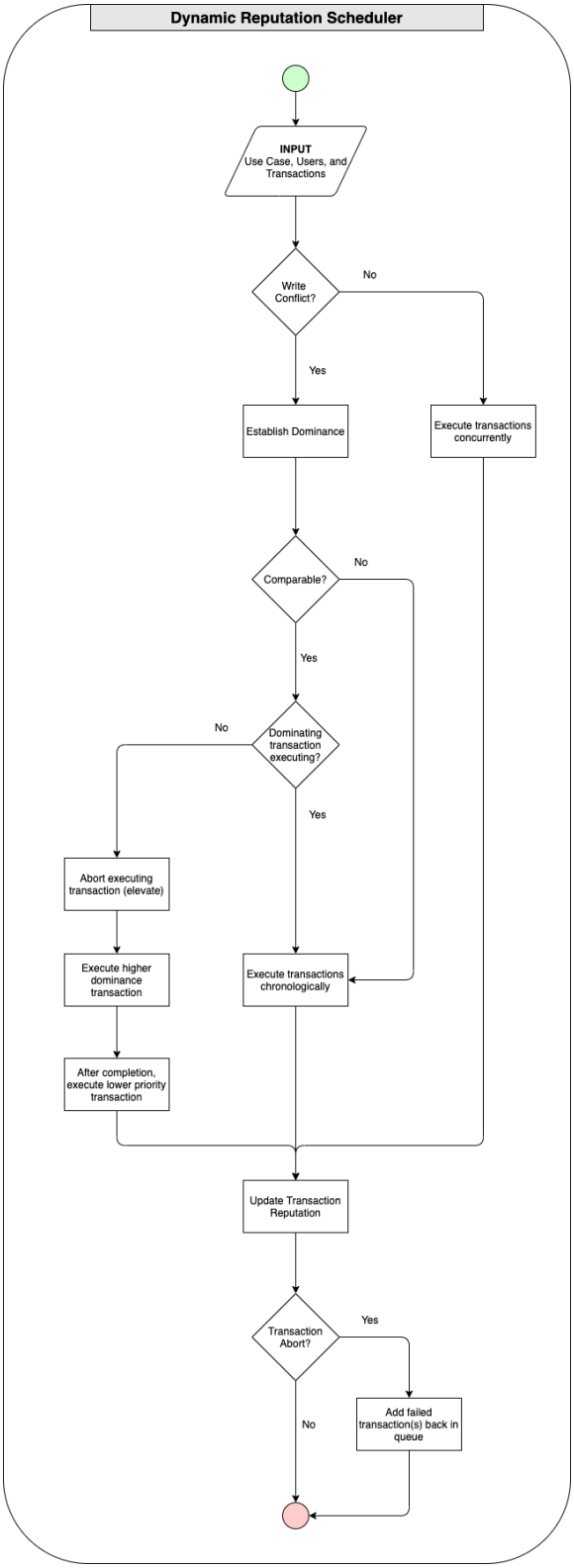


Figure 3.10: Flow of Dynamic Reputation Scheduler

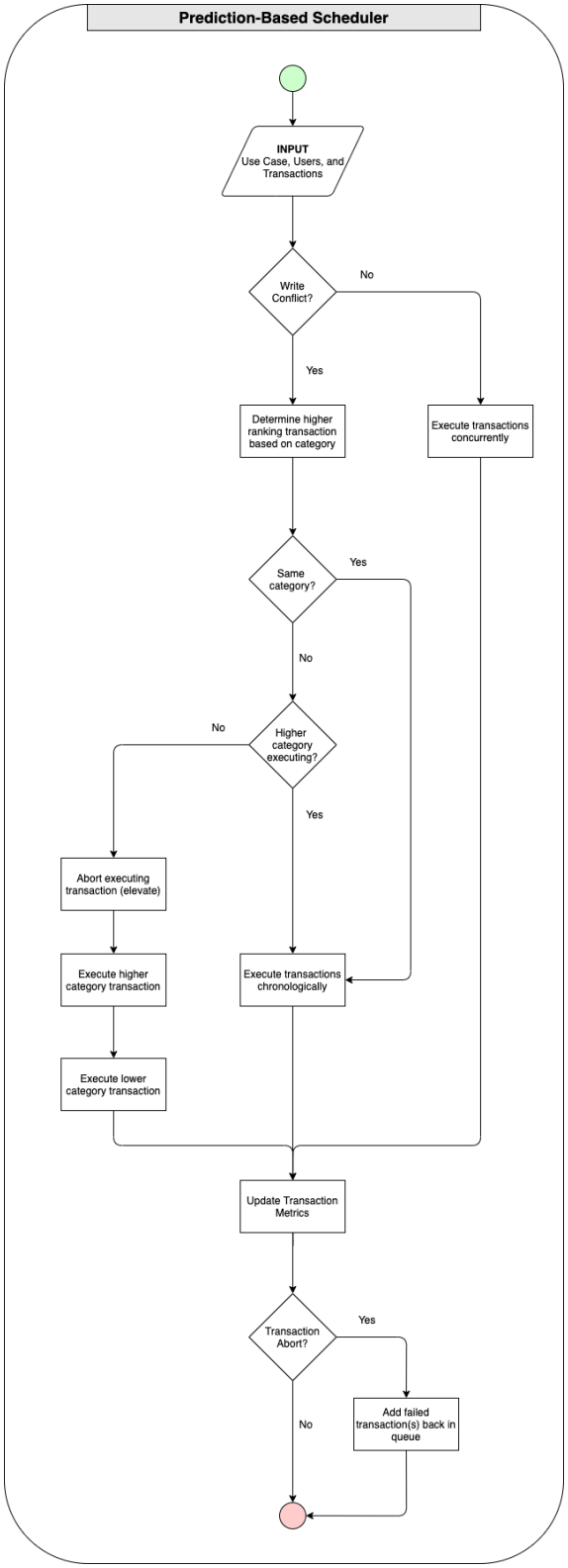


Figure 3.11: Flow of Prediction Based Scheduler

Use Cases 3 & 4 were purposely executed with extreme parameters in order to see the load the system would endure with multiple recalculations. There were multiple recalculations executed in asynchronous threads. The executions did not block the execution of the transactions but caused a great deal of CPU usage. These are parameters that we would not expect in a live system but were used specifically to see the load of recalculations.

Use Cases 5 & 6 were the use cases where all three schedulers were executing. All three schedulers executed with the same transactions, users, and use case parameters to get an equal comparison. This was the best indicator of what would be expected in a real world scenario.

3.7.1 Expectations

Before executing the simulation, our definitions (see Section 3.5.1) and system model (see Section 3.5) led us to certain expectations that would come from the simulation. The overall expectation and claim from the dynamic reputation solution is that it provides a low overhead resource management solution with consistent scheduling. That claim is supported by the following three expectations:

1. There will be greater reward and punishment from the dynamic reputation solution than from our previous prediction-based solution

2. We will see the reward and punishment reflected in the rankings of the users and transactions

3. The execution time will be comparable to both the previous prediction-based and 2PL scheduler to not indicate a serious overhead

3.7.2 Findings

Our first finding defends our first claim that there will be greater reward and punishment in our new solution than our previous prediction-based solution. We can determine a greater level of reward and punishment by looking at the percentage of transactions that were elevated due to a conflict. Our formula for calculating the percentage of elevated transactions is below:

$$T_{elevate} = \#\text{ of executions that caused an ELEVATE}$$

$$T_{total} = \text{Total }\#\text{ of executions}$$

$$P_{reward} = \frac{T_{elevate}}{T_{total}} \times 100$$

After analyzing the results we discovered that the P_{reward} for the dynamic reputation solution is 51.9% and the P_{reward} for the prediction-based solution is 7.1%. This is expected given that the reputation management solution involves a much more granular approach to establishing dominance that the four category system in the previous prediction-based solution. Therefore, this confirms that the dynamic reputation solution allows for a greater percentage of reward and punishment among the conflicting transactions.

Our second finding defends our second claim that we will see the reward and punishment reflected in the rankings of the users and transactions. By seeing a variance in the rankings of the users and transactions (see Definitions 11, 12, 13, and 14) then we can confirm that our recalculations are causing changes in dominance based on the reputations of users and transactions. Figure 3.12 and 3.13 are two graphs showing the variance in the growth/reduction of transaction and user rankings across executions. You can see the rankings shrinking and growing as recalculations occur. Figure 3.12 shows the transaction rankings changing throughout the system. The y-axis represents the variance. The x-axis is the transaction ID (not shown for

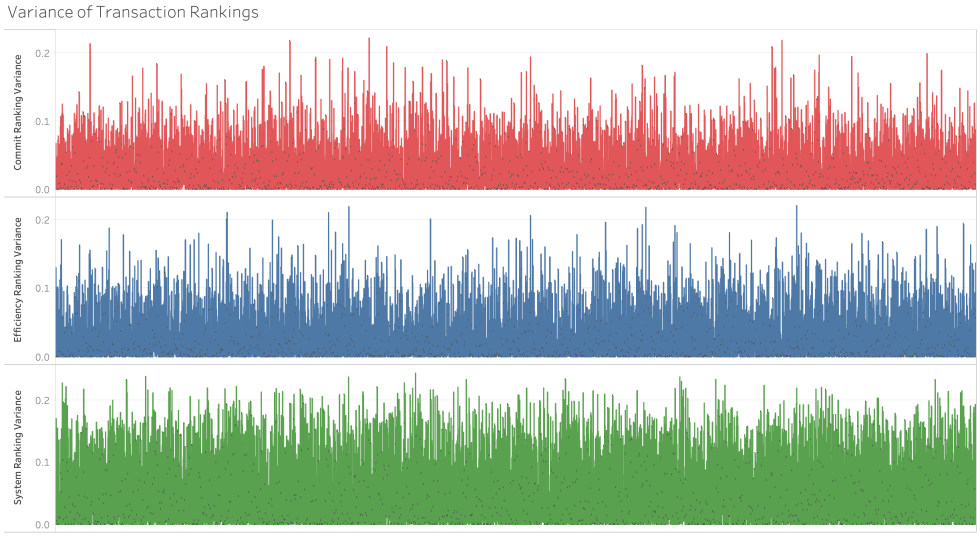


Figure 3.12: Variance of Transaction Rankings

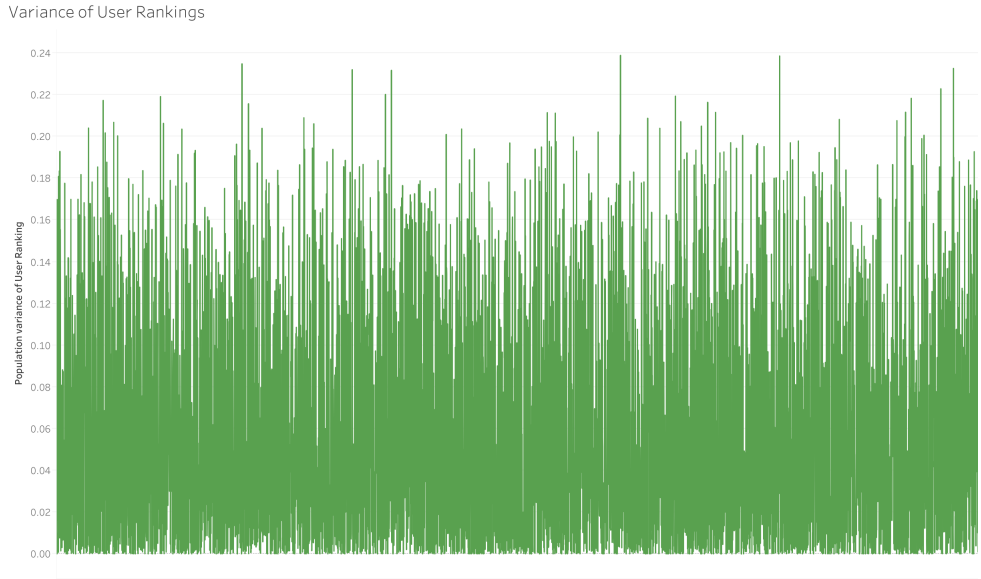


Figure 3.13: Variance of User Rankings

view ability). This graph represents all transactions that executed more than once within the system in order to show the changes in rankings between executions.

Figure 3.13 shows the same variance but for user rankings as they change throughout the system due to recalculation. The graph shows that the reward and punishment is actively being applied throughout the system.

Our third and final finding defends our third claim that the execution time will be comparable to both the previous prediction-based and 2PL schedulers. When comparing all of the schedulers execution time over different workloads (see Figure 3.14) we can see the differing execution times however, even with the differing execution times the overhead of the dynamic reputation system is comparable and a feasible solution. This defends our third and final claim that the dynamic reputation solution is a feasible solution but as we examine the data we can be more precise of when the solution should be legitimately considered.

After examining the data we see that the execution time is directly related to the workload. The defining difference of the workloads is the percentage of conflicting transactions in the workload. As the percentage of conflict increases in the differing workloads you can see the schedulers begin to execute at different execution times.

From the graph in Figure 3.14 we can deduce that the best environment for the dynamic reputation solution is within execution environments that contain greater than 20% conflicting transactions. Execution environments with high levels of conflict can benefit from the dynamic reputation solution while environments with less than 20% conflict would be impacted by the overhead of processing within the dynamic reputation solution.

With our findings defending all three of our claims we can with confidence claim that the dynamic reputation solution provides a low overhead resource management solution with consistent scheduling.

3.8 Conclusion

In closing we conclude that the dynamic reputation solution provides a system of greater reward and punishment than the previous prediction-based solution. The four category system of the prediction-based solution doesn't contain the level of granularity needed to establish dominance in conflicting situations. We also conclude

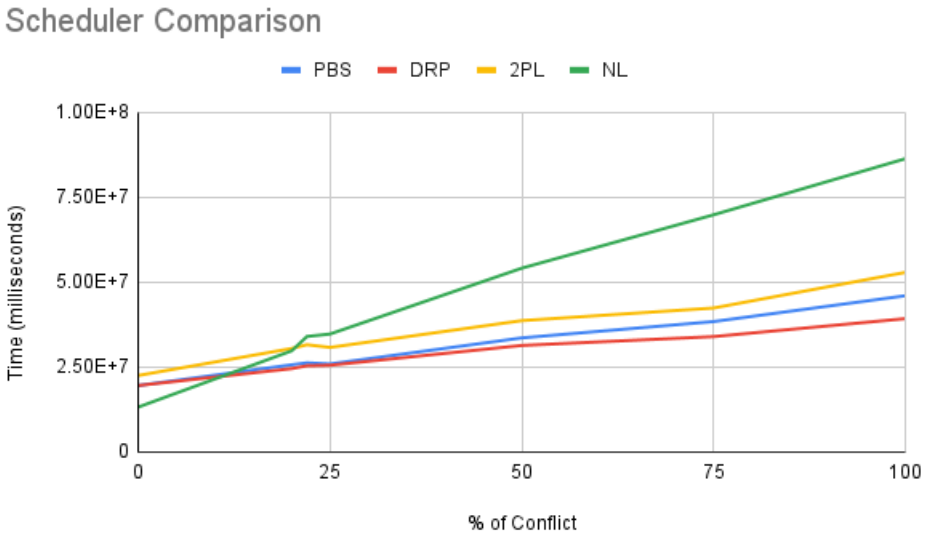


Figure 3.14: Scheduler Comparison

that the overhead of the dynamic reputation solution is comparable to the previous prediction-based and 2PL solutions therefore establishing a resource management solution with a feasible overhead and consistent scheduling. By analyzing our findings from the experimentation we can conclude that execution environments with workloads of greater than 20% conflicting transactions would benefit from the granularity of the dynamic reputation scheduler to provided consistency and efficient scheduling.

Chapter 4

Future Work

In this Chapter we discuss the potential for future work and extensions of the prediction-based scheduler. Primarily we discuss the work involved with multi-level secure databases.

4.1 Introduction

Now that the framework for transactional correctness has been developed (work published in Ravan, Banik, and Farkas 2020) within the prediction-based solution there is potential to extend the reputation score of the prediction-based scheduler to multi-level secure databases. The model would include the security classification and allow for a much more robust decision-model. This portion of the overall solution would focus on multi-level secure database systems and covert timing channels. By adding a security label component to the existing framework we can extend our reputation score to provide a cover story for the timing difference of transactions with differing security classifications.

The problem with multi-level secure databases is the possibility that there could be a covert channel allowing unauthorized access. The covert channel would provided the ability for a transaction of a lower security classification to access resources designed for a higher security classification. Existing research provides possible solutions but many of the solutions starve transactions of higher security classifications from gaining access to the resources needed (see Section 4.3). With that cover story in mind we can then provide a solution that elevates high security transactions would the presence

of a covert timing channel. With the prediction-based solution provided in Chapter 2 and the reputation score provided in Chapter 3, we can provide a cover story that determines locking priority based on the reputation of the transaction as a whole. This prevents the starvation of higher security classification transactions. By taking into consideration the security classification as a metric to calculate the reputation of the transaction, we also prevent the presence of covert channels.

4.2 Problem Definition

Multi-level secure databases (referred to as MLS databases going forward) differ from traditional databases in that there is content within the database that cannot be accessed by all users of the database. There is data within the database itself that contains a high security classification (or multiple security classifications) than other data. This is also more volatile than multi-tenant systems. Multi-tenant systems also contain data which cannot be accessed by all users, but the data within the system contains the same priority when it comes to transaction scheduling. Security classifications within MLS databases introduce priorities and therefore introduce the issues of starvation. MLS databases also introduce the issue of covert channels since there are multiple security classifications.

A covert channel is a channel of communication that performs communication outside of the normal access control mechanisms. This then makes it very hard to to secure using existing security measures since normal security measures are performed on the existing access control mechanisms. There are two main types of covert channels; storage channels and timing channels. Storage channels are a form of communication by modifying an existing storage location with data that would normally not be detected. Timing channels expose security issues by the presence or absence of a delay in transaction processing. Figure 4.1 shows the time delay difference between a normal operating system load to that of a delayed load operation.

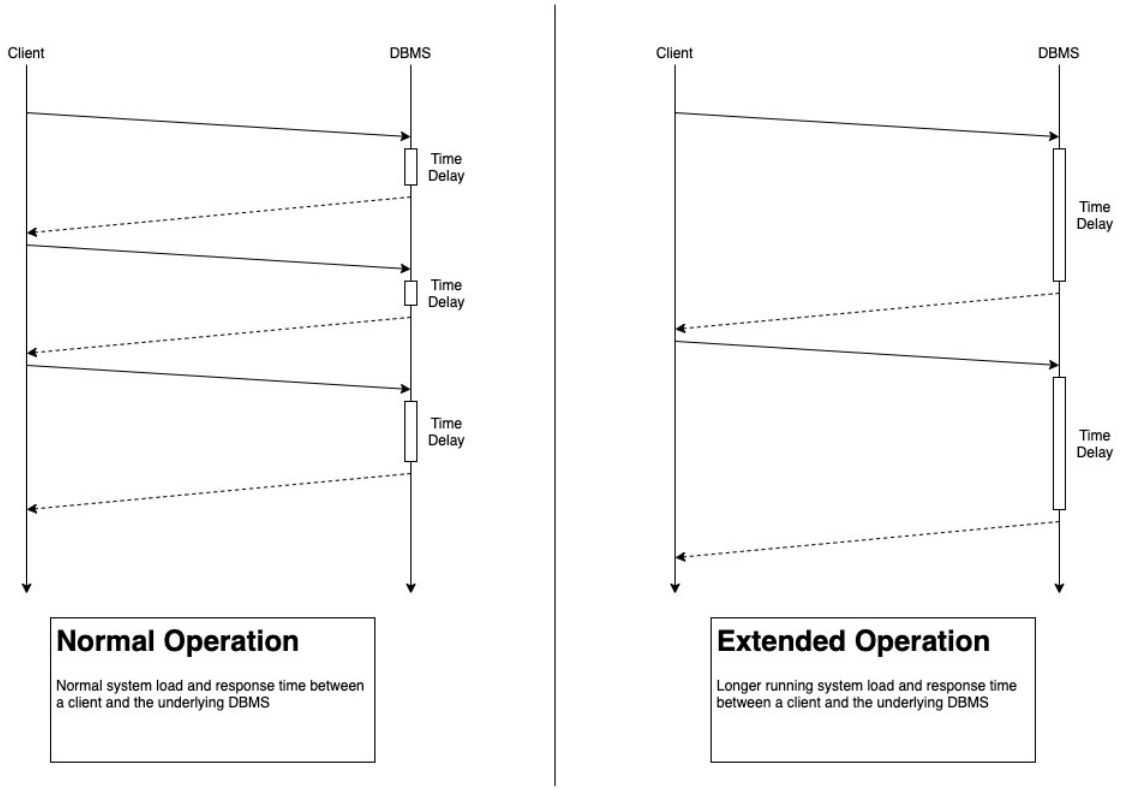


Figure 4.1: Covert Channel Exposure

This presence of a time delay can allow for information passing to occur and the attacker to infer components of the system architecture especially in MLS databases.

In order to prevent a timing covert channel within MLS database systems, there must not be a presence of a timing delay for transactions with lower security classifications. The transactions must incur the standard time delay as a normal transaction so that there is no suspicion that a covert channel is available.

4.2.1 Elevating The Priority

Most concurrency control algorithms solve this issue by giving a sort of precedence or priority to lower security classification transactions to prevent any time delay. However, this can cause issues with transactions with a higher security classifications if high security transactions continually conflict with lower transactions. The high security transactions can suffer from starvation and cause a huge performance hit.

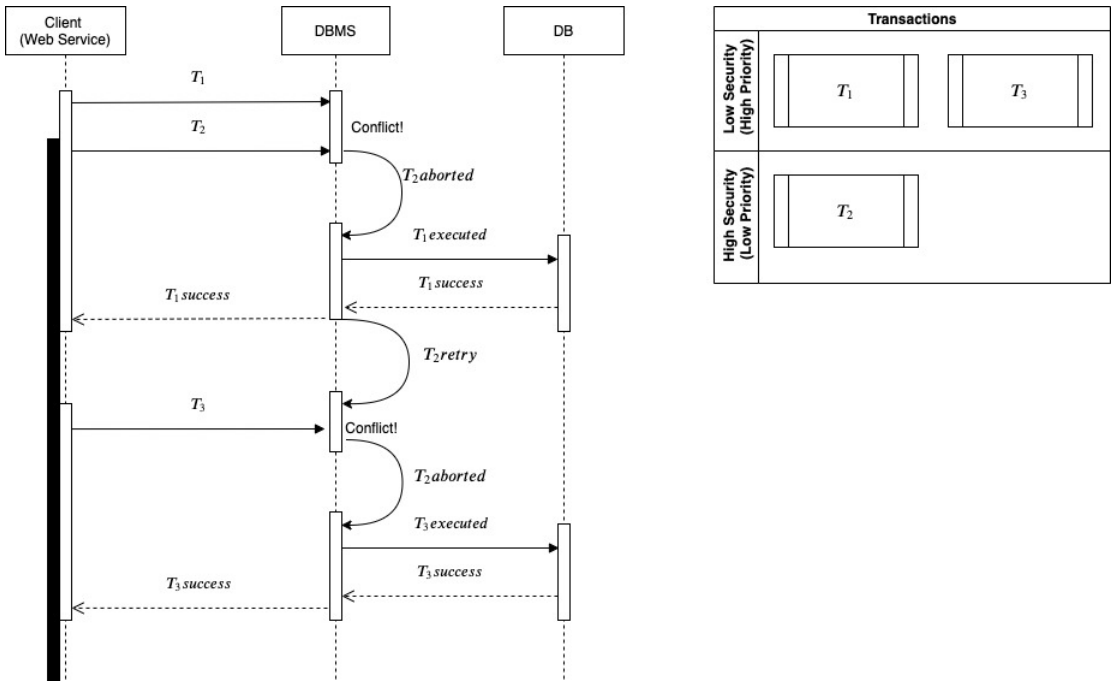


Figure 4.2: Web Service Transaction Starvation

Figure 4.2 shows an example of how a high security classification transaction can experience starvation with standard locking mechanisms.

Figure 4.2 shows an example of three transactions (T_1, T_2, and T_3) entering a system via a web service. T_1 and T_3 are of a low security classification while T_2 is of a high security classification. T_1 and T_2 enter the system at the same time and are accessing a shared resource between the two transactions. This causes a conflict and in order to prevent a timing covert channel, we abort the higher security transaction, T_2. T_1 is then able to execute successfully and complete. We then attempt to retry T_2 but coincidentally another transaction, T_3, is submitted for execution. The same process happens again, T_2 is aborted, and T_3 is executed successfully. The black vertical bar on the left side of Figured 4.2 shows the process of T_2 through all the aborts that happened. This illustrates the starvation of T_2 that happens when trying to prevent covert channels.

4.2.2 Problem Identified

After analyzing the system structure and architecture of an MLS database and seeing a use-case scenario, we see two problems that can be improved upon. The first being the existence of a timing covert channel due to a conflict among low and high security transactions. The second problem identified is the starvation of high security transactions due to the solution that many lock-based concurrency control algorithms present to address timing covert channels. Both of these problems have been addressed in past research (look at Samarati and Sandhu 2016 for example) by simply causing transactions of a higher-security classification to abort in order to make way for the lower-security classification and then adding a priority to higher-security transactions in order to prevent starvation.

But the problem with this approach is that there is no flexibility within the solution to abort a problematic low-security transaction. The current solution leverages a binary decision model and priorities within the recovery model to address the side effect of starvation. The real problem is providing a solution that eliminates the timing covert channel with minimal side effects to efficiency and consistency that have to be reconciled. This solution would provide a safe and reliable way to abort both high-security and low-security transactions to prevent covert channels depending on the system environment.

We believe that this problem can be addressed with the dynamic categorization and reputation that the prediction-based scheduler provides in Chapters 2 and 3.

4.3 Related Work

Multi-level secure databases are a huge area of research due to the security concerns that can arise within these databases and the consequences if a vulnerability is exposed. The consequences have been so great that many users of multiple security classifications use multiple databases with duplicated common resources to prevent

any security vulnerability from happening [1]. However, researchers understand the benefits of having a secure solution contained in a single database system with multiple security levels.

Jajodia et. al. is one of the main motivations for our work (see S. Jajodia, Mancini, and Setia 1998). In this publication we see a new locking protocol presented specifically for multi-level secure databases that prevents the starvation of lower security classification transactions. In this work, the researchers analyzed the existing two-phase locking protocol with additional policy additions to increase performance. In their analysis they discovered in order to increase performance and prevent starvation by increasing the fairness of all transactions, they needed to restrict the number of low security transactions executing. One quote from the work that motivates our work is,

> "Several concurrency control algorithms that are free from covert channels have been proposed in the literature. Most of these algorithms prevent covert timing channels by ensuring that transactions at lower security levels are never delayed by the actions of a transaction at a higher security level. This can be accomplished by providing a higher priority to low transactions whenever a data conflict occurs between a high transaction and a low transaction."

The prediction-based solution established in Chapter 2 provides a solution to elevate or demote transactions based on transactional attributes that are deemed necessary for the transaction's reputation score. As a part of the reputation score, the security level in which a transaction resides can be a part of the transaction's attributes necessary for ranking.

[1] This implementation is commonly found in DoD database systems and their Security Technical Implementation Guides (STIGs) https://public.cyber.mil/stigs/

Other, more recent, works that have been of influence for this solution involve Mahmoud and Alqumboz 2019 by Mahmoud and Alqumboz, Y.-G. Sun 2011 by Ying-Guan Sun, Hedayati et al. 2010 by Hedayati et. al., Shanwal and Kumar 2013 by Shanwal and Kumar, Sapra and Kumar 2014 by Sapra et. al., Kaur, Sarje, and Misra 2004 by Kaur, N. et. al., Costich and Moskowitz 1991 by Costich, O.L. et. al., Kaur et al. 2007 by Kaur, N. et. al, Keefe, Tsai, and Srivastava 1993 by Keefe T.F. et. al, and David and Son 1993 by David N. et. al. All of which have been built up on the work of the Bell–LaPadula Model, Biba Integrity Model, and lattice based security model (LBAC) (work referenced in Bell and LaPadula 1973, Biba 1977, & Denning 1976). An overview of multilevel secure databases and transaction processing can be found in Atluri et. al. (Vijay Atluri 1999).

4.4 Environment

A multi-level secure database is much like any traditional database system. The major difference is the presence of resources, users, and transactions with differing security levels. This can be resources within the system that contain a certain a security level. This can also be true for users who maintain a certain security level and therefore the transactions generated by the user contains a certain security level. Current architecture solutions leverage the Bell-LaPadula model (Bell and LaPadula 1973) to ensure that current security levels are maintained and data is not accessed inappropriately. The Bell LaPadula Model abides by two main rules to ensure secure data access. The two rules are:

1. A subject at a given security level may not read an object at a higher security level. This is known as the Simple Security Property

2. A subject at a given security level many not write to any object a lower security level. This is known as the * (star) Property

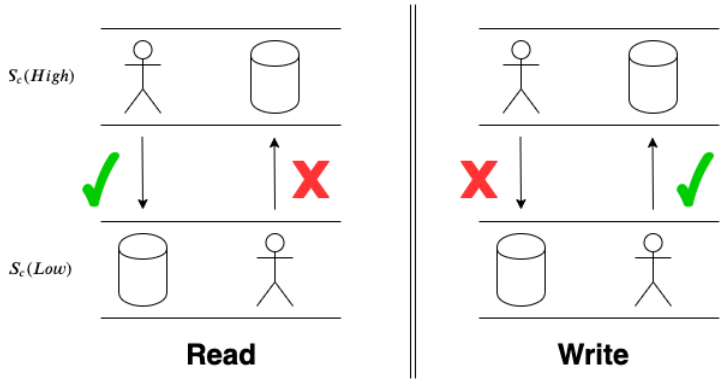


Figure 4.3: Bell-LaPadula Model

Both of these properties are shown in Figure 4.3. While this ensures proper data access control for security levels, the Bell-LaPadula Model doesn't protect against covert channels that occur due to concurrent transactions. Concurrent transactions cause issues when there are conflicting operations. When there is a conflict, one of the transactions must wait for the other transactions to finish processing before execution can continue processing. The presence (or even absence) of a time delay for the transaction to execute introduces a covert channel.

A covert channel is a security flaw where a means of communication to transmit unauthorized information is available via the normal means of communication. Timing channels are a form of covert channel where the presence of absence of a execution delay conveys unauthorized information about the underlying system. This work is documented by Girling in Girling 1987. Timing channels are difficult to prevent and many times requires review of the application source code directly to ensure all operations execute with the same timing delay. Common solutions to prevent covert timing channels in multi-level secure databases when there are conflicting operations is to abort the transaction with a higher security classification. This prevents the transaction with a lower security transaction from detecting a timing delay. The timing delay would communicate to the lower security transaction that resources of a higher security classification were present and therefore leaking unauthorized information.

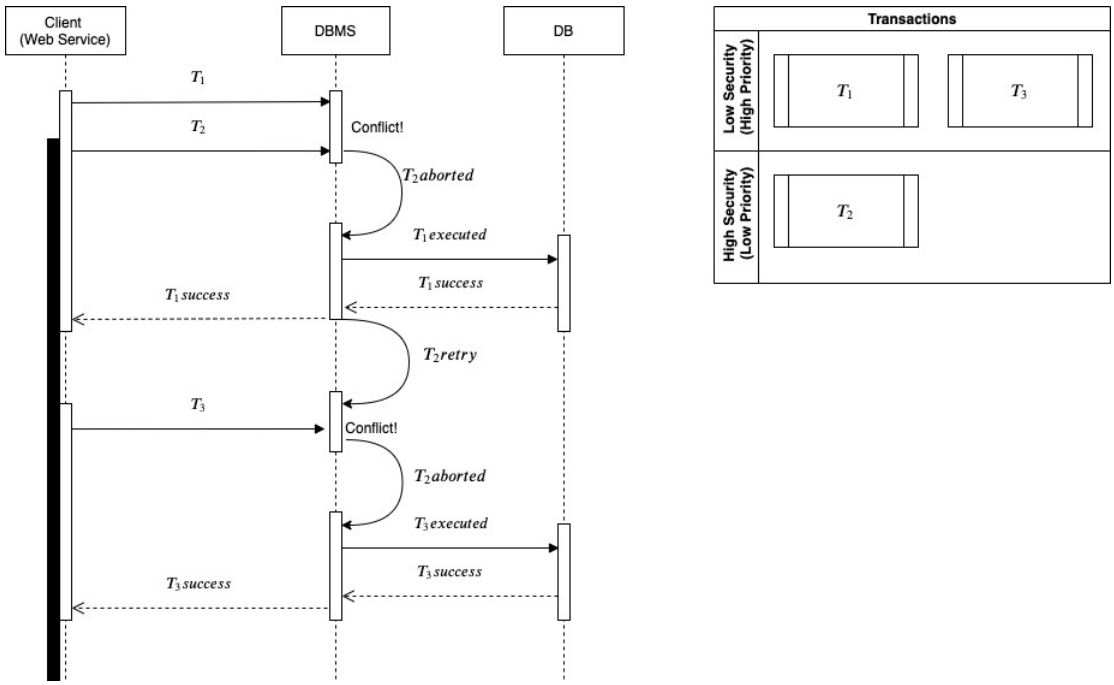


Figure 4.4: Transaction Starvation Exposes Timing Covert Channel

Figure 4.4 (referenced from Figure 4.2 in Section 4.2) shows the exposure of a timing covert channel.

4.5 Transaction Quality Measure

In this section we discuss the potential future work for implementing a security classification within the prediction-based scheduler (outlined in Chapters 2 and 3) that would address the issues found specifically in MLS databases.

Within the prediction-based solution, there are currently four attributes that allow for a reputation score to be calculated among transactions. Those four attributes are commit ranking, efficiency ranking, user ranking and system ranking (see Definitions 11, 12, 13, and 14 in Chapter 3). This allows for the formation of a reputation score to be calculated for each transaction in the system. Within these reputation scores, there is a dominance structure that causes transactions to be prioritized depending on if dominance can be established (outlined in Definitions 16 and 17). With all of these

components in place, we have a foundation for efficient transaction categorization within MLS databases.

A potential future state of the system is to use the existing reputation score within the existing prediction-based solution alongside a security label to create a two-tuple Transaction Quality Measure (TQM). This will involve an update dominance structure to ensure the absence of covert channels and also prevent starvation of higher security transactions within the system. The solution would allow for a better decision making model for which transactions are aborted and rescheduled. The new Transaction Quality Measure would take security classification into account, but it would not allow the security classification to be the only dictating factor. Extending the prediction-based solution would allow the other four attributes to be included within the decision process. Figure 4.5 is a representation of the existing reputation score defined in Chapter 3. Figure 4.6 shows the strong dominance structure of Transaction Quality Measures where SL is the security label and PV is the performance vector representing the existing reputation score. Figure 4.7 shows the weak dominance structure.

$$RS_{T_i} = < w_i^1 \times CR_{T_i}, w_i^2 \times ER_{T_i}, w_i^3 \times UR_i, w_i^4 \times SR_{T_i} >$$

Figure 4.5: Current Reputation Score

$$TQM_1(SL_1, PV_1) \geq TQM_2(SL_2, PV_2) \text{ iff } SL_1 \leq SL_2 \text{ and } PV_1 \geq PV_2$$

Figure 4.6: MLS Strong Dominance

$$TQM_1(SL_1, PV_1) \geq TQM_2(SL_2, PV_2) \text{ iff } SL_1 \geq SL_2 \text{ and } PV_1 > PV_2$$

Figure 4.7: MLS Weak Dominance

In order to prevent covert timing channels within multi-level secure database, the future solution would leverage the timing delay of the existing prediction-based solution to be used as a "cover story" for the timing difference between transactions of differing security levels. The cover story would allow for transactions to be aborted for conflicting transactions without introducing a covert timing channel for unauthorized disclosure of high security resources.

In summary, there are two well-known problems within multi-level secure databases. The first problem is the existence timing covert channels when transactions of multiple security levels are accessing a common resource. The presence or absence of a time delay provides the indication of high security resources that are available. The second problem, is brought on by the solution to multi-level secure databases. A solution to prevent against covert channels is to abort transactions of a higher security classification so that the time delay does not exist. However, this then causes these transactions to suffer from starvation and will never be executed. The solution presented in this section will address both problems and provide a way forward for more granular decision-making within multi-level secure database systems.

With the prediction-based scheduler in place and a solution for dynamic reputation of transactions, the possibilities for extension within multi-level secure databases is then feasible . Chapter 2 presented the solution and operated under the assumption that the reputation of the transactions were already established. Chapter 3 focuses on exactly how the transactions establish their reputation and also dynamically increase or decrease their reputation. With this work in place, extending the prediction-based system to multi-level secure databases is now possible.

4.6 Additional Future Work

In this section, we want to outline the future work opportunities of the prediction-based scheduler and how the work can be expanded upon. The work mentioned in this

section is not meant to be included in this dissertation, but rather listing outstanding opportunities for the current and proposed work to continue forward.

4.6.1 Snapshot Isolation

Another potential for future work is the ability to perform snapshot isolation within the different categorizations of transactions. This extension will be for both malicious and lower priority transactions that affect the majority of well-performing transactions. In this work we'll use snapshot isolation to execute certain categorizations of transactions on snapshots of the database in order to prevent the effects of low-performing transactions from affecting all transactions. Once the outcome of a transaction has been determined then the snapshot can either be discarded or merged.

4.6.2 Prediction-based Scheduling within Linked Databases

An additional extension is the issue of efficient concurrency control within linked database environments. Currently the Prediction-based solution addresses efficient concurrent transactions within a web-service environment that are contained within a single cluster. This particular area of research will address the problem through the lens of the Prediction-based solution. The difficulty of the problem within this work is adapting the existing framework of correctness built within the Prediction-based solution so that it will scale to linked database systems while preserving its existing capabilities. Figure 4.8 shows the system model for the Prediction-based solution within linked database systems.

4.6.3 PostgreSQL & MySQL

Two very commonly used databases within enterprise applications are PostgresSQL and MySQL. Both of which are open-source relational databases where their code is available to the public for modification and contribution. Open-source applications

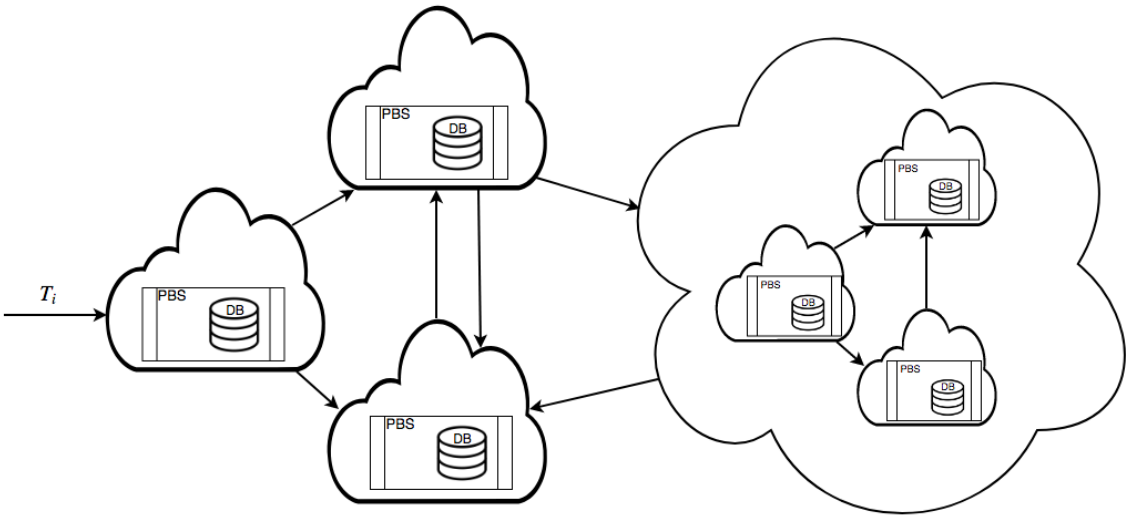


Figure 4.8: Prediction-based Scheduler within Linked Databases

tend to have very difficult review process which allows for quality code and reliable software. Both of these database management systems have provided their code on Github so that the community can contribute features, bug fixes, and enhancements accordingly. The code for PostgresSQL is located at https://github.com/postgres/postgres and the code for MySQL is located at https://github.com/mysql/mysql-server.

One opportunity for future work would be to fork one or both of these repositories on a controlled system and implement the algorithms of the prediction-based scheduler within the database management system itself. Currently, the prediction-based scheduler has been proven theoretically in a test environment using an in-memory database. This work has proven the viability of the solution and the consistency that it provides. By placing the prediction-based solution in the database management system itself, it would provide a beautiful marriage of academia and industry coming together for a common solution. The initial goal would be to get the algorithms working on a mirrored fork initially in a controlled test environment, then moving that solution to a clustered environment to ensure scalability, and eventually providing an official pull request of the prediction-based scheduler to the code maintainers of both systems so the solution would then be available to the general public in fu-

ture releases. This future work provides a direct road map from academic theory to impacting the global industry for the benefit of the masses.

4.7 Conclusion

In summary, there are a multitude of opportunities to extend the current work into new realms. The prediction-based scheduler along with dynamic transaction reputation provides a system of consistency and scalability that can be extended into multi-level secure databases, linked databases, and other systems of databases. This contribution will provide a foundation for future researchers to extend into realms of database scheduling and transaction execution that have not been discussed before.

Chapter 5

Conclusion

In this work, we have first analyzed the shortcomings to existing web service database transactions. These shortcomings involve the need for compensation transactions that, ultimately, are unnecessary overhead if ACID transaction properties can be leveraged. We analyzed the current solutions involving compensation transactions and maintaining consistency within a web service environment. As a result of this analysis, research and prototyping led to the development of a prediction-based solution. The final solution dynamically uses different concurrency control mechanisms depending on the attributes of the transaction. In the current work we have formally proven that this solution will ensure consistency by using dynamic concurrency control mechanisms (published in Ravan, Banik, and Farkas 2020).

The next effort focused on the formal categorization of transactions before entering the prediction-based solution. Here we manage a reputation so that transactions can be rewarded or punished based on their performance in the system. Their reputations are built by using four attributes that rank them in the system: user ranking, commit ranking, efficiency ranking, and system ranking. From there we build a reputation score and prioritize well behaving transactions in the event of a conflict. We built a prototype to show how the solution compares against other solutions and discovered an increase in efficiency when systems contain 20% or more conflicting transactions.

An additional effort of database recovery using provenance was completed (see Rhujittawiwat et al. 2021).

Finally, we have proposed the needed extensions to the future work that will address three additional areas of improvement. The first area is expanding the prediction-based solution to a multi-level secure database environment where multiple security classifications exist. This involves extending the existing reputation score to contain an additional attribute to prevent the existence of covert timing channels that can appear due to differing security classifications. By doing so we provide a cover story for the transactional environment that enables the promotion of high security transactions without the disclosure of a covert timing channel.

Additional future work involves snapshot isolation, the prediction-based solution within linked databases, and providing the prediction-based solution within the an open source DBMS.

Bibliography

Alomari, Mohammad, Alan Fekete, and Uwe Röhm (Mar. 2014). "Performance of Program Modification Techniques That Ensure Serializable Executions with Snapshot Isolation DBMS". In: *Inf. Syst.* 40, pp. 84–101. ISSN: 0306-4379. DOI: 10.1016/j.is.2013.10.002. URL: http://dx.doi.org/10.1016/j.is.2013.10.002.

Alrifai, Mohammad et al. (Oct. 2009). "Distributed Management of Concurrent Web Service Transactions". In: *Services Computing, IEEE Transactions on* 2.4, pp. 289–302. ISSN: 1939-1374. DOI: 10.1109/TSC.2009.29.

Bailis, Peter et al. (2014). "Scalable atomic visibility with RAMP Transactions". In: *ACM SIGMOD Conference*.

Bell, D. Elliott and Leonard J. LaPadula (1973). *Secure computer systems: Mathematical foundations*. Tech. rep. MITRE CORP BEDFORD MA.

Bernstein, Philip A, Vassos Hadzilacos, and Nathan Goodman (1986). *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., pp. 53–56. ISBN: 0-201-10715-5.

Biba, Kenneth J. (1977). *Integrity considerations for secure computer systems*. Tech. rep. MITRE CORP BEDFORD MA.

Chiejina, Eric, Hannan Xiao, and Bruce Christianson (Sept. 2014). "A Dynamic Reputation Management System for mobile ad hoc networks". In: *2014 6th Computer Science and Electronic Engineering Conference (CEEC)*, pp. 133–138. DOI: 10.1109/CEEC.2014.6958568.

Clark, Michael R., Kyle Stewart, and Kenneth M. Hopkinson (Sept. 2017). "Dynamic, Privacy-Preserving Decentralized Reputation Systems". In: *IEEE Transactions on Mobile Computing* 16.9. Conference Name: IEEE Transactions on Mobile Computing, pp. 2506–2517. ISSN: 1558-0660. DOI: 10.1109/TMC.2016.2635645.

Cormen, Thomas H. et al. (2009). *Introduction to Algorithms*. 3rd. McGraw-Hill Higher Education. ISBN: 0070131511.

Costich, O.L. and I.S. Moskowitz (June 1991). "Analysis of a storage channel in the two phase commit protocol". In: *Proceedings Computer Security Foundations Workshop IV*, pp. 201–208. DOI: 10.1109/CSFW.1991.151587.

Dai, Yu, Lei Yang, and Bin Zhang (Mar. 2009). "QoS-Driven Self-Healing Web Service Composition Based on Performance Prediction". en. In: *Journal of Computer Science and Technology* 24.2, pp. 250–261. ISSN: 1860-4749. DOI: 10.1007/s11390-009-9221-8. URL: https://doi.org/10.1007/s11390-009-9221-8 (visited on 01/21/2020).

David, R. and S. Son (Oct. 1993). "A secure two phase locking protocol". In: *Proceedings of 1993 IEEE 12th Symposium on Reliable Distributed Systems*, pp. 126–135. DOI: 10.1109/RELDIS.1993.393466.

De Paola, Alessandra and Adriano Tamburo (Mar. 2008). "Reputation management in distributed systems". In: *2008 3rd International Symposium on Communications, Control and Signal Processing*, pp. 666–670. DOI: 10.1109/ISCCSP.2008.4537308.

Denning, Dorothy E. (May 1976). "A lattice model of secure information flow". In: *Communications of the ACM* 19.5, pp. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056 (visited on 05/02/2020).

Feingold, M. and R. Jeyaraman (Feb. 2009). "Web Services Coordination (WS-Coordination) Version 1.2". In: URL: http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os.pdf.

Ferreira, João E. et al. (June 2012). "Transactional Recovery Support for Robust Exception Handling in Business Process Services". In: *2012 IEEE 19th International Conference on Web Services*. ISSN: null, pp. 303–310. DOI: 10.1109/ICWS.2012.14.

Freund, T. and M. Little (Feb. 2009). "Web Services Business Activity (WS-BusinessActivity) Version 1.2". In: URL: http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os.pdf.

Gao, Zhengdong and Gengfeng Wu (Oct. 2005). "Combining QoS-based service selection with performance prediction". In: *IEEE International Conference on e-Business Engineering (ICEBE'05)*. ISSN: null, pp. 611–614. DOI: 10.1109/ICEBE.2005.38.

Garcaa-Molrna, Hector and Salem Kenneth (1987). "SAGAS". In: pp. 249–259. URL: http://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf.

Girling, C.G. (Feb. 1987). "Covert Channels in LAN's". In: *IEEE Transactions on Software Engineering* SE-13.2. Conference Name: IEEE Transactions on Software Engineering, pp. 292–296. ISSN: 1939-3520. DOI: 10.1109/TSE.1987.233153.

Greenfield, Paul et al. (2007). "Isolation Support for Service-based Applications: A Position Paper". In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pp. 314–323. URL: http://www.cidrdb.org/cidr2007/papers/cidr07p36.pdf.

Hedayati, Maysam et al. (June 2010). "Evaluation of performance concurrency control algorithm for secure firm real-time database systems via simulation model". In: *2010 International Conference on Networking and Information Technology*. ISSN: 2324-8203, pp. 260–264. DOI: 10.1109/ICNIT.2010.5508515.

Hu, Jianli et al. (July 2010). "A Reputation Based Attack Resistant Distributed Trust Management Model in P2P Networks". In: *2010 Third International Symposium on Electronic Commerce and Security*, pp. 237–241. DOI: 10.1109/ISECS.2010.59.

Jacobi II, Christian and Cédric Lichtenau (1999). "Highly Concurrent Locking in Shared Memory Database Systems". In: *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*. Euro-Par '99. London, UK, UK: Springer-Verlag, pp. 477–481. ISBN: 3-540-66443-2. URL: http://dl.acm.org/citation.cfm?id=646664.701052.

Jajodia, S., L. Mancini, and S. Setia (June 1998). "A fair locking protocol for multilevel secure databases". In: *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No.98TB100238)*. ISSN: 1063-6900, pp. 168–178. DOI: 10.1109/CSFW.1998.683167.

Jang, Julian, K. Fekete, and Paul Greenfield (July 2007). "Delivering Promises for Web Services Applications". In: *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pp. 599–606. DOI: 10.1109/ICWS.2007.70.

Kaur, N., A.K. Sarje, and M. Misra (Dec. 2004). "Performance Evaluation of Concurrency Control Algorithm for Multilevel Secure Distributed Databases". In: *Student Conference On Engineering, Sciences and Technology*, pp. 116–121. DOI: 10.1109/SCONES.2004.1564781.

Kaur, N. et al. (Dec. 2007). "A Feedback Based Secure Concurrency Control for MLS Distributed Database". In: *International Conference on Computational Intelligence and Multimedia Applications (ICCIMA 2007)*. Vol. 3, pp. 8–12. DOI: 10.1109/ICCIMA.2007.159.

Keefe, T.F., W.T. Tsai, and J. Srivastava (Dec. 1993). “Database concurrency control in multilevel secure database management systems”. In: *IEEE Transactions on Knowledge and Data Engineering* 5.6. Conference Name: IEEE Transactions on Knowledge and Data Engineering, pp. 1039–1055. ISSN: 1558-2191. DOI: 10.1109/69.250090.

Lee, Eunhee et al. (2001). “Prediction-based Concurrency Control for a Large Scale Networked Virtual Environment Supporting Various Navigation Speeds”. In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST ’01. Baniff, Alberta, Canada: ACM, pp. 127–134. ISBN: 1-58113-427-4. DOI: 10.1145/505008.505034. URL: http://doi.acm.org/10.1145/505008.505034.

Lee, Kang-Woo and Hyoung-Joo Kim (June 2000). “Consistency preserving in transaction processing on the Web”. In: *Proceedings of the First International Conference on Web Information Systems Engineering*. Vol. 1. ISSN: null, 190–195 vol.1. DOI: 10.1109/WISE.2000.882392.

Little, M. and A. Wilkinson (Feb. 2009). “Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.2”. In: URL: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os.pdf.

Liu, Peng and Sushil Jajodia (2001). “Multi-Phase Damage Confinement in Database Systems for Intrusion Tolerance”. In: *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*. CSFW ’01. Washington, DC, USA: IEEE Computer Society, pp. 191–. URL: http://dl.acm.org/citation.cfm?id=872752.873520.

Lomet, David, Zografoula Vagena, and Roger Barga (June 2006). “Recovery from "bad" user transactions”. In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. SIGMOD ’06. New York, NY, USA: Association for Computing Machinery, pp. 337–346. ISBN: 978-1-59593-434-5. DOI: 10.1145/1142473.1142512. URL: http://doi.org/10.1145/1142473.1142512 (visited on 02/06/2021).

Mahmood, S. et al. (Dec. 2006). “Swarm Intelligence Based Reputation Model for Open Multi Agent Systems”. In: *2006 IEEE International Multitopic Conference*, pp. 178–181. DOI: 10.1109/INMIC.2006.358158.

Mahmoud, Ahmed Y. and Mohammed Naji Abu Alqumboz (Oct. 2019). “Encryption Based On Multilevel Security for Relational Database EBMSR”. In: *2019 International Conference on Promising Electronic Technologies (ICPET)*. ISSN: null, pp. 130–135. DOI: 10.1109/ICPET.2019.00031.

Melnikov, Almaz et al. (May 2018). “Towards Dynamic Interaction-Based Reputation Models”. In: *2018 IEEE 32nd International Conference on Advanced Information*

Networking and Applications (AINA). ISSN: 2332-5658, pp. 422–428. DOI: 10.1109/AINA.2018.00070.

Olmsted, Aspen (Dec. 2015). "Long running, consistent, web service transactions". In: *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. ISSN: null, pp. 139–144. DOI: 10.1109/ICITST.2015.7412074.

Rajesh Turlapati and M. N. Huhns (Sept. 2005). "Multiagent reputation management to achieve robust software using redundancy". In: *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pp. 386–392. DOI: 10.1109/IAT.2005.105.

Ravan, John Thomas, Shankar M. Banik, and Csilla Farkas (2020). "Ensuring Consistent Transactions in a Web Service Environment with Prediction-Based Performance Metrics". In: *IEEE Transactions on Services Computing*, pp. 1–1. ISSN: 2372-0204. DOI: 10.1109/TSC.2020.2974733.

Rhujittawiwat, Theppatorn et al. (2021). "Database Recovery from Malicious Transactions: A Use of Provenance Information". In: *Proceedings of the 10th International Conference on Data Science, Technology and Applications, DATA 2021, Online Streaming, July 6-8, 2021*. Ed. by Christoph Quix, Slimane Hammoudi, and Wil M. P. van der Aalst. SCITEPRESS, pp. 39–48. DOI: 10.5220/0010553900390048. URL: https://doi.org/10.5220/0010553900390048.

Riegen, M. von et al. (Jan. 2010). "Rule-Based Coordination of Distributed Web Service Transactions". In: *Services Computing, IEEE Transactions on* 3.1, pp. 60–72. ISSN: 1939-1374. DOI: 10.1109/TSC.2009.27.

Samarati, Pierangela and Ravi Sandhu (Jan. 2016). *Database Security X: Status and prospects*. en. Google-Books-ID: YpF8CwAAQBAJ. Springer. ISBN: 978-0-387-35167-4.

Sapra, Pooja and Suresh Kumar (Feb. 2014). "Development of a concurrency control technique for multilevel secure databases". In: *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pp. 111–115. DOI: 10.1109/ICROIT.2014.6798308.

Shanwal, Sonakshi and Suresh Kumar (Sept. 2013). "Secure concurrency control algorithm for multilevel secure databases". In: *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*. ISSN: null, pp. 149–153. DOI: 10.1049/cp.2013.2309.

Sun, Ying-Guang (Aug. 2011). "Access control method based on multi-level security tag for distributed database system". In: *Proceedings of 2011 International Con-*

ference on Electronic Mechanical Engineering and Information Technology. Vol. 5. ISSN: null, pp. 2509–2512. DOI: 10.1109/EMEIT.2011.6023609.

Sun, Yuxin and Yuping Zhao (Dec. 2019). "Dynamic Adaptive Trust Management System in Wireless Sensor Networks". In: *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, pp. 629–633. DOI: 10.1109/ICCC47050.2019.9064321.

Vijay Atluri Sushil Jajodia, Binto George (Nov. 1999). *Multilevel Secure Transaction Processing*. en. Springer. ISBN: 978-0792377023.

Wang, H. et al. (Oct. 2006). "Multiagent System for Reputation–based Web Services Selection". In: *2006 Sixth International Conference on Quality Software (QSIC'06)*. ISSN: 2332-662X, pp. 429–434. DOI: 10.1109/QSIC.2006.43.

Wang, Jue, Jian Peng, and Daping Zhang (Dec. 2008). "Research on Dynamic Reputation Management Model Based on PageRank". In: *2008 International Conference on Computer Science and Software Engineering*. Vol. 3, pp. 814–817. DOI: 10.1109/CSSE.2008.927.

Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Accessed: 2015-02-06.

Zhang, O. Q. et al. (2012). "How to Track Your Data: Rule-Based Data Provenance Tracing Algorithms". In: *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 1429–1437. DOI: 10.1109/TrustCom.2012.175.

Appendix A

PBS Results

This section of the appendix presents more of the experimentation results from the prediction-based scheduler in Chapter 2.

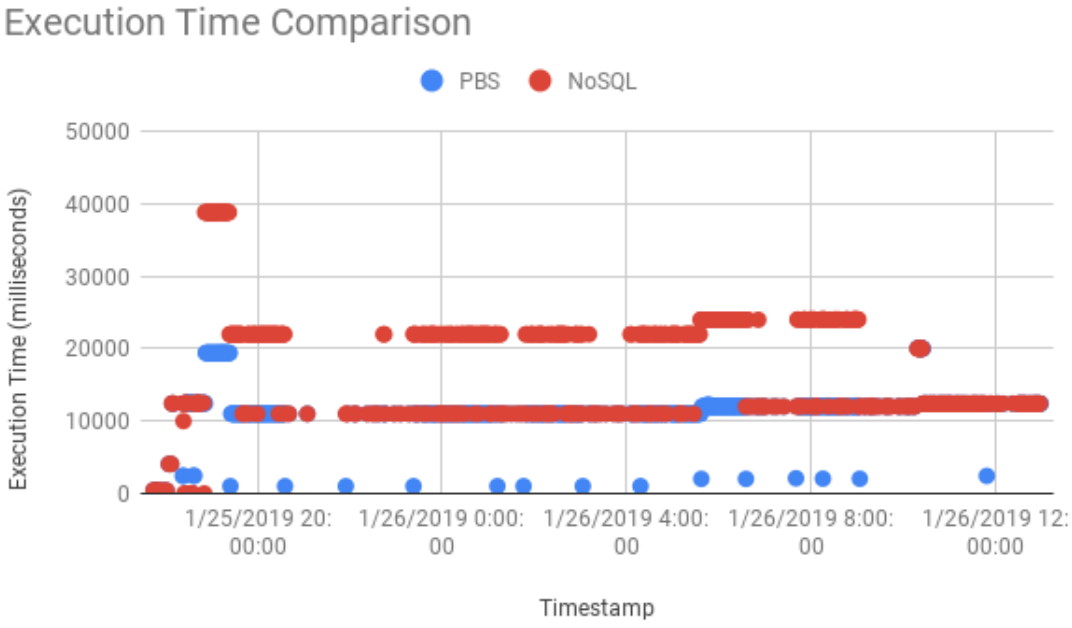


Figure A.1: Test Case 1

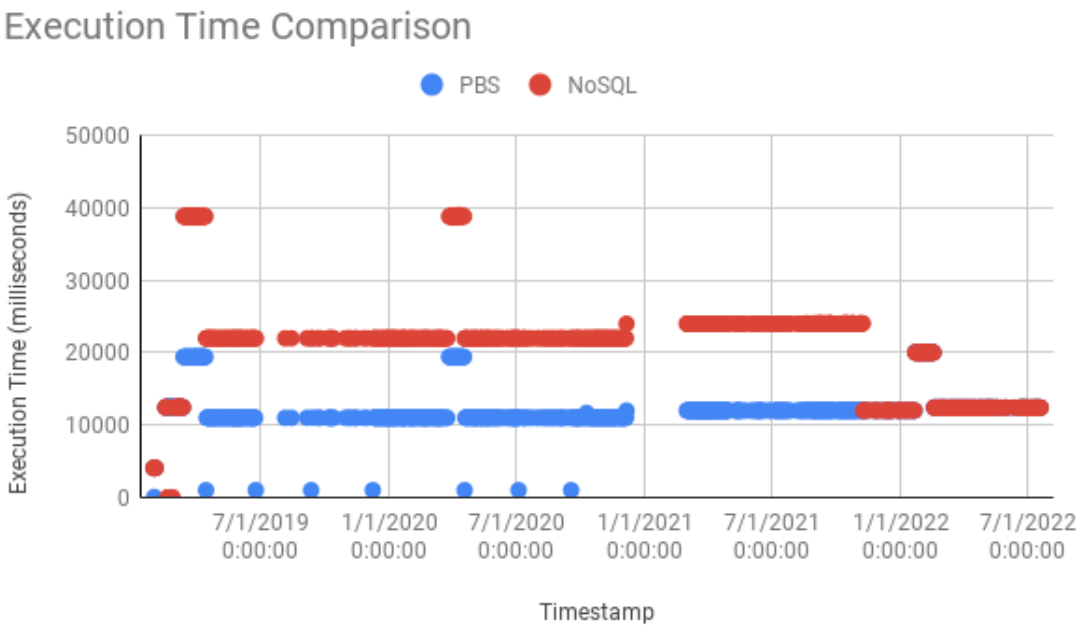


Figure A.2: Test Case 2

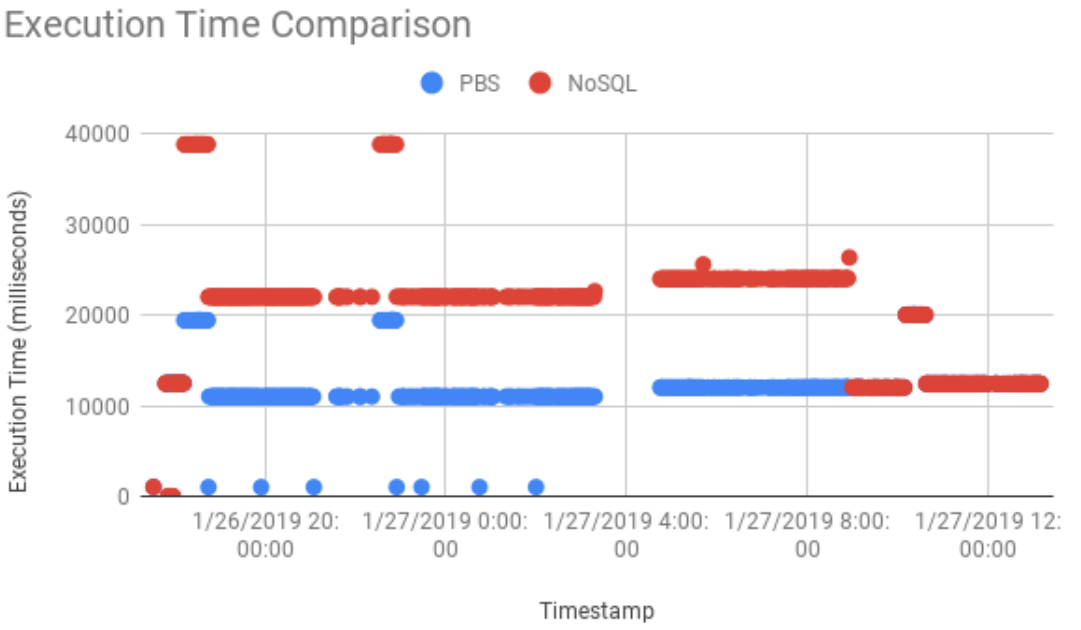


Figure A.3: Test Case 3

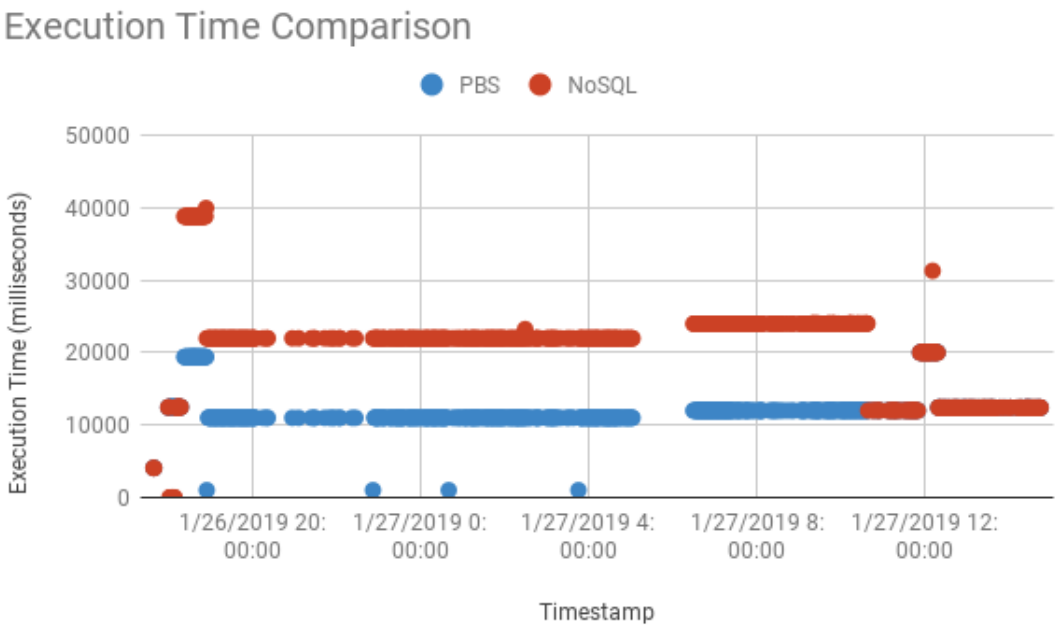


Figure A.4: Test Case 4

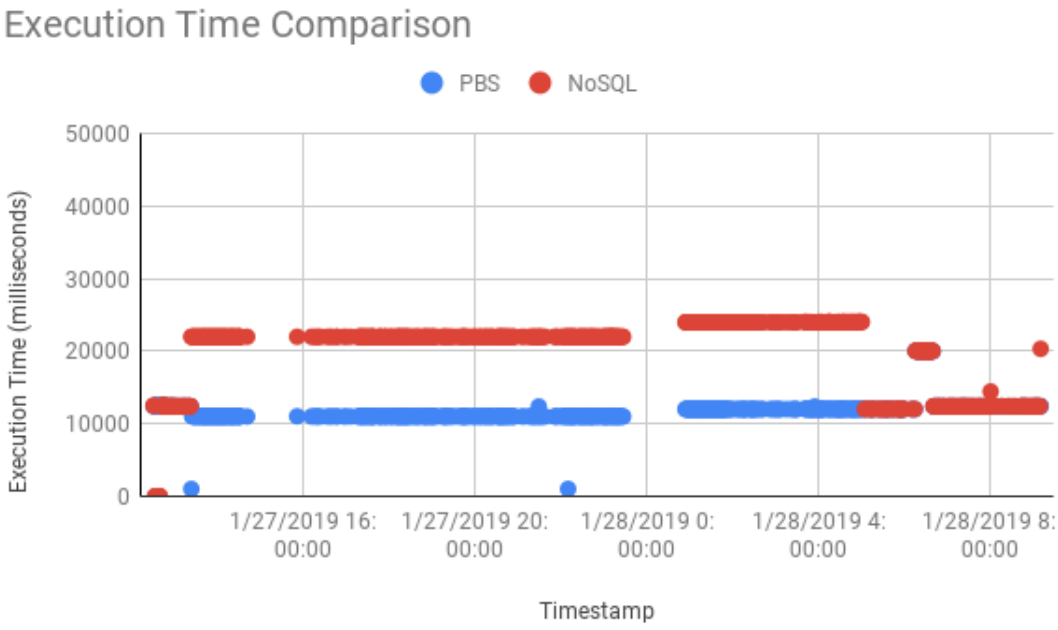


Figure A.5: Test Case 5

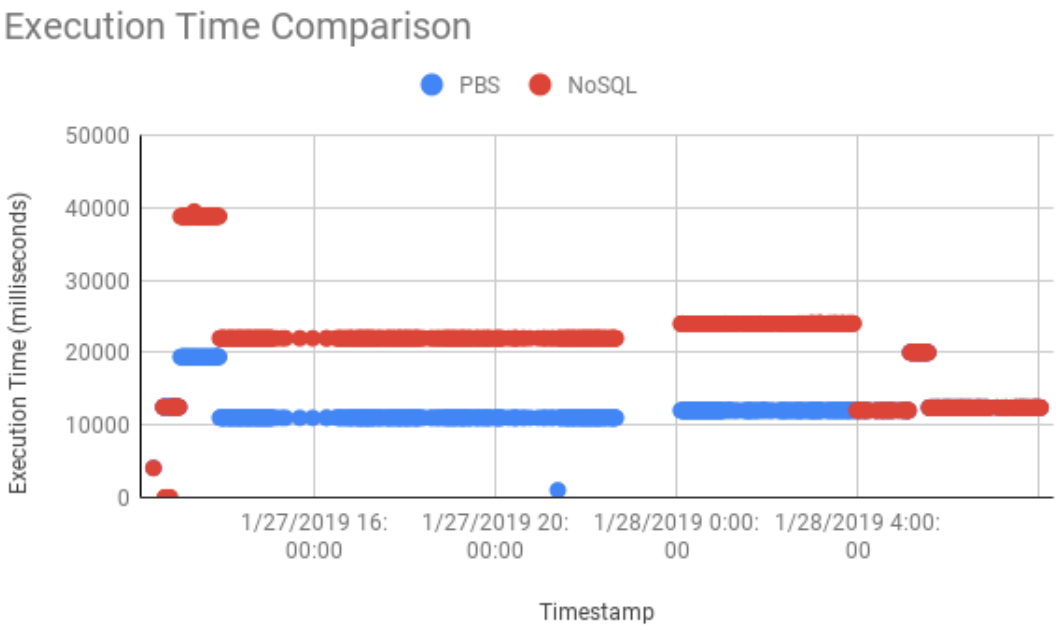


Figure A.6: Test Case 6

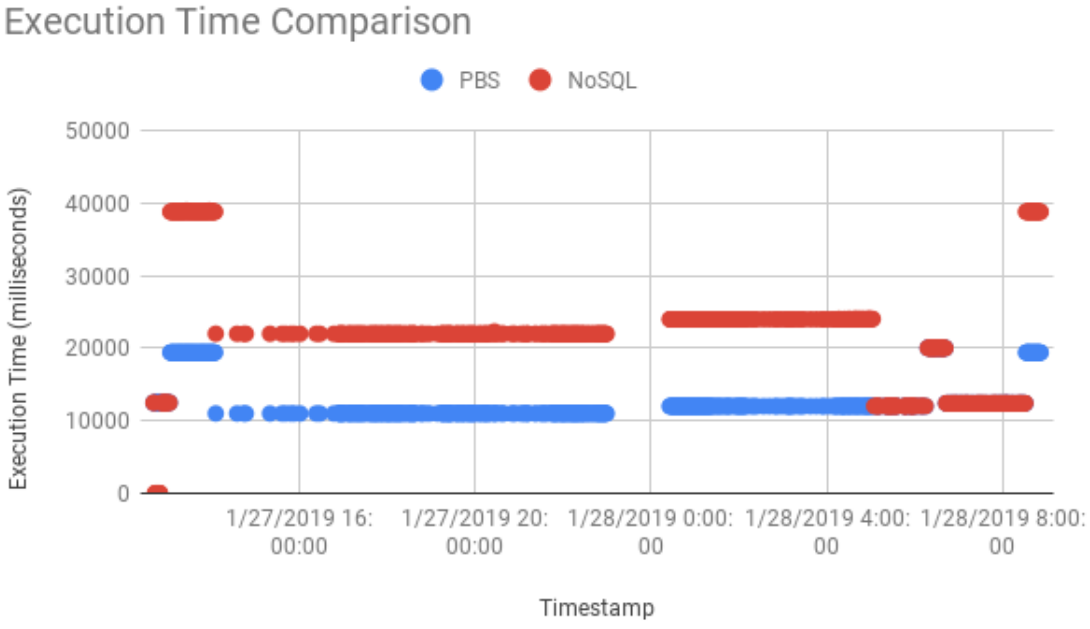


Figure A.7: Test Case 7