EXPLORING CAUSAL FACTORS OF DBMS THRASHING

by

Young-Kyoon Suh

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2015

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Young-Kyoon Suh, titled Exploring Causal Factors of DBMS Thrashing and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Date: 6 April 2015

Richard T. Snodgrass

Date: 6 April 2015

Sabah Currim

Date: 6 April 2015

Peter J. Downey

Date: 6 April 2015

John K. Kececioglu

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Date: 6 April 2015

Dissertation Director: Richard T. Snodgrass

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of the source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Young-Kyoon Suh

ACKNOWLEDGEMENTS

First and above all, I praise God, the almighty for always being with me during my PhD study and granting me the capability to finish this dissertation. This dissertation would not appear without the assistance and guidance of several people. I would, hence, like to thank to all of them.

I would like to express the deepest appreciation to my thesis advisor and committee chair, Professor Richard T. Snodgrass. He continually and persuasively conveyed a spirit of adventure in regard to research, teaching, and mentoring. I was truly blessed to get to work with him and obtain several achievements during my PhD. My experience with him for many years was incredibly delightful, unforgettable. If it were not for his sincere supervision and considerate care, this dissertation would not have been possible.

I would also like to thank my committee members, Professors John Kececioglu and Peter J. Downey. They have continually provided their sharp and insightful comments to improve this thesis work and showed great support.

Moreover, I cannot thank you enough to Dr. Sabah Currim. Her expertise in statistics guided me to successfully conduct complicated evaluations needed in this dissertation. The collaboration with her in other projects was an amazing experience, which won't be forgotten in my career.

I was a lucky guy to get to intern in the Department of Advanced Development Engineering at Teradata. I met really good people during my internship. I would love to give my special thanks to my mentors, Ahmad Ghazal and Alain Crolotte, and my manager, Pekka Kostamma. Thanks to their internship offer, I gained an exciting industrial experience beyond academia.

I also want to thank to the Department of Computer Science at the University of Arizona. Without their financial support, I would not survive during my PhD study.

Thanks to all my close friends from campus, Drs. Jaehoon Choi, Kyoung-Ha Lee, and Hyojoon Seok, and Ajeya Naithani for the joyful gatherings and all their supports. Also thanks to my long-term friends who always trust me, Drs. Bon-Hyun Koo, Dohyung Kim, Hangkyu Kim, Changsoo Lee, and Jaekwon Yoo, and Myoungho Choi, Byungsam Jang, Jinhyuk Lee, Changmin Lee, Jinseok Park, and Duyeol Yoo. I also would like to give my deep gratitude to my American Mom, Peggy Ford, who has sympathized all my frustrations and put full support in her prayers at all times during my stay in Tucson.

I would love to express my deep thanks to my family. I sincerely appreciate to my parents, parents-in-law, uncle, ant, lovely cousins, and sister-in-law (Sungeun) for their long-term prayer, support, and patience. I want to express my warmest gratitude to my brother, Youngbo, for being in my side throughout my PhD study. I am sure that he will also successfully complete his PhD at Texas A&M in a couple of years.

And finally, my lovely wife, Hyeryeong, I would not have finished this long-term work without your support, encouragement, and prayer. I just want to say, thank you, and love you forever.

April 2015, Tucson, USA

DEDICATION

I would like to dedicate my dissertation to the Almighty God, to all of my family members and friends, and to my lovely wife Hyeryeong.

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF FIGURES – *Continued*

LIST OF TABLES

LIST OF TABLES – *Continued*

ABSTRACT

Modern DBMSes are designed to support many transactions running simultaneously. DBMS thrashing is indicated by the existence of a sharp drop in transaction throughput. The thrashing behavior in DBMSes is a serious concern to DBAs engaged in on-line transaction processing (OLTP) and on-line analytical processing (OLAP) systems, as well as to DBMS implementors developing technologies related to concurrency control. If thrashing is prevalent in a DBMS, thousands of transactions may be aborted, resulting in little progress in transaction throughput over time. From an engineering perspective, therefore, it is of critical importance to understand the factors of DBMS thrashing.

However, understanding the origin of modern DBMSes' thrashing is challenging, due to many factors that may interact. The existing literature on thrashing exhibits the following weaknesses: (i) methodologies have been based on simulation and analytical studies, rather than on empirical analysis on real DBMSes, (ii) scant attention has been paid to the associations between factors, and (iii) studies have been restricted to one specific DBMS rather than across multiple DBMSes.

This dissertation aims at better understanding the thrashing phenomenon across multiple DBMSes. We identify the underlying causes and propose a novel structural causal model to explicate the relationships between various factors contributing to DBMS thrashing. Our model derives a number of specific hypotheses to be subsequently tested across DBMSes, providing empirical support for this model as well as engineering implications for fundamental improvements in transaction processing. Our model also guides database researchers to refine this causal model, by looking into other unknown factors.

CHAPTER 1

INTRODUCTION

In this chapter, we introduce the problem of DBMS thrashing, discuss how to approach the problem, provide a list of contributions, and lastly present the organization of this dissertation.

1.1 The Problem

The focal problem of this dissertation is "DBMS thrashing." In this section, we define the term of DBMS thrashing and see examples of thrashing phenomena observed in our experiments on several modern relational DBMSes. We then consider the challenge of understanding the causality of DBMS thrashing.

1.1.1 Background

Database management systems (DBMSes) are a core component of current information technology (IT) systems [27]. DBMSes have been widely adopted to serve a variety of workloads such as on-line analytical processing (OLAP) [5] and on-line transaction processing (OLTP) [63]. Over the last five decades, achieving high performance in handling workloads has been one of the primary goals in the database community. Accordingly, various methodologies and techniques have been proposed to enhance the efficiency and speed of DBMSes and database applications.

Many DBMS performance issues have been addressed and resolved over these decades, but *scalability* is still regarded as a major concern [27, 34]. Scalability is a desirable attribute of a network, system, or process; namely, it indicates the ability of the system to accommodate an increasing number of objects, to process growing volumes of work gracefully, and/or to be amenable to enlargement [2].

When a scalability bottleneck is encountered in a DBMS, transaction (or query) throughput can drop. For instance, the throughput may not increase as the workload increases, but we can still obtain the same throughput because of underlying bottleneck(s). In the worst case, the DBMS may experience performance degradation initiated by *thrashing*.

In the thrashing case, the throughput drops dramatically. Thus, the transaction (or query) response gets extremely slow to end users using applications on top of the DBMSes [14, 89].

Peter Denning originally coined the term of thrashing in the context of operating system [14]. When page faults occur over increasing workloads, workload throughput drops sharply, as demonstrated in Figure 1.1. In this dissertation, we conduct an in-depth analysis on the factors and their correlations with the thrashing phenomenon in the context of DBMSes.

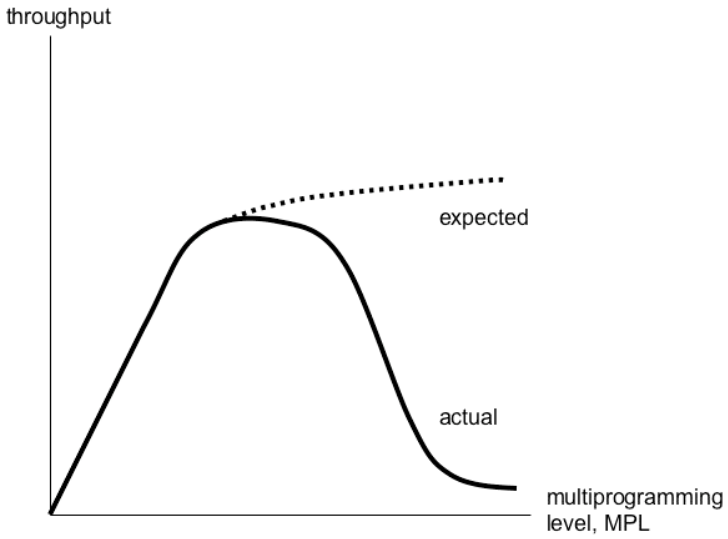


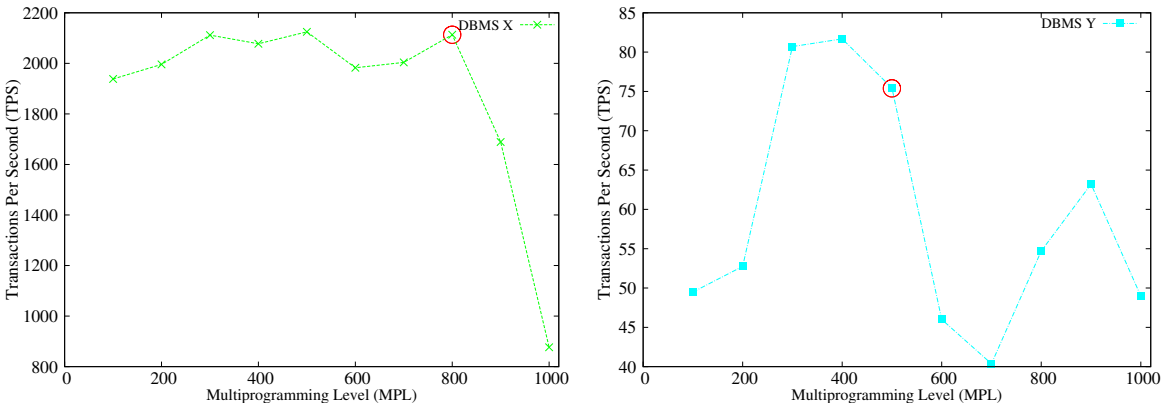
Figure 1.1: The Phenomenon of Thrashing

1.1.2 DBMS thrashing

We define *DBMS thrashing* as a precipitous drop of transaction throughput over increasing multiprogramming level (MPL) [35, 37, 83, 89]. Interestingly, DBMS thrashing is observed in modern high-performance DBMSes, which are designed to seamlessly support many concurrent transactions, as demonstrated in Figure 1.2.

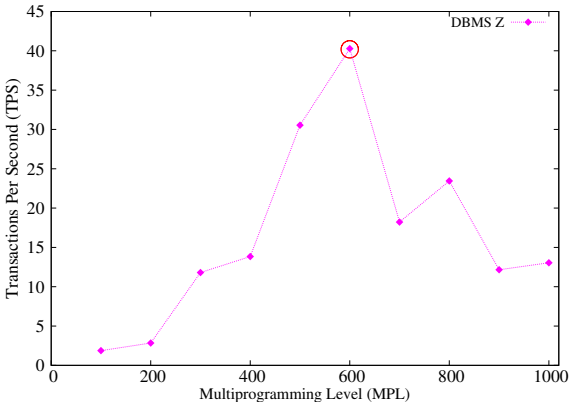
Here are our configuration details. We used a total of five relational DBMSes, including three proprietary ones, named X, Y, and Z, respectively, and two open-source DBMSes (MySQL [53] and PostgreSQL [79]). We also attempted to use another proprietary DBMS W, but we could not collect its data because of some technical challenges to be discussed in Section 7.1.1. Thus, DBMS W was excluded in our experiment. (For the commercial DBMSes, legal agreements disallow us to report their performance evaluation results.)

Each DBMS was run exclusively on a dedicated machine, equipped with Intel Core i7-870

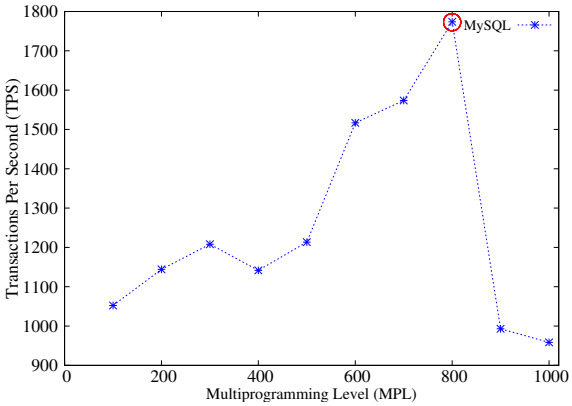


(a) DBMS X's Thrashing

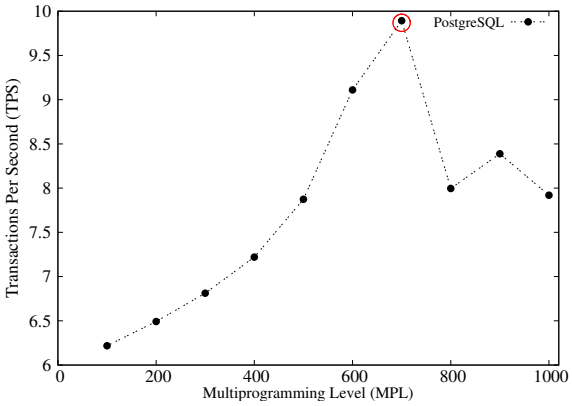
(b) DBMS Y's Thrashing



(c) DBMS Z's Thrashing



(d) MySQL's Thrashing



(e) PostgreSQL's Thrashing

Figure 1.2: Observed DBMS Thrashing

Lynnfield 2.93GHz quad-core processor on a LGA 1156 95W motherboard with 8GB DDR3 1333 dual-channel memory and Western Digital Caviar Black 1TB 7200rpm SATA hard drive. Each DBMS was running on RedHat Enterprise Linux Server release 6.4 or Windows Server 2008 R2 Enterprise. We enabled only a single core with hyper-threading disabled. In other words, only one

CPU processor was provided to the machine. (In later experiments described in Section 3.4.1, we expose more processors to each DBMS, to see how much the increased processors contribute to thrashing.)

For each DBMS, we populated a database schema of two tables, with 1M rows consisting of five integer columns and two variable-sized chars. One of the integer columns served as the row number that was sequentially increased; the other columns were assigned values that were randomly generated. Each client was given a pre-generated read-only transaction with 1% selectivity, where the selected row range between clients could potentially overlap. (For more detail on the schema and transaction, refer to the transaction size paragraph in Section 3.4.3.)

We presented exactly the same workload to all the DBMSes. One exception was in one of the DBMSes, for which we were not able to populate the 1M rows due to some unknown technical reasons. Instead, the 1M cardinality was simply reduced to 30K only for that DBMS. Accordingly, we adjusted the corresponding workload along with the reduced tables.

We varied MPL from 100 to 1,000 by a granularity of 100. At each value of MPL, a client opened a session to a DBMS and kept running the same transaction (consisting of a single `SELECT` statement with range predicates with the 1% selectivity factor) on that DBMS three times for about two minutes. After all the repetitions for an individual MPL, we calculated the number of completed transactions per second (TPS) at that MPL.

In this experiment, we observed that thrashing occurred in all the DBMSes with increasing MPLs, as illustrated in Figure 1.2. Figure 1.2(a) shows the DBMS X's thrashing, which started at an MPL of 800 marked with a red circle. As can be seen in Figure 1.2(a), several bumps were observed, but DBMS X was still in the "smooth phase" until an MPL of 800. After that, however, a significant drop suddenly happened in throughput; that is, DBMS X entered the "thrashing phase." Thus, the MPL of 800 is called "thrashing point," after which DBMS X experienced the thrashing. The phase transition remained unchanged.

DBMS Y also experienced thrashing, as illustrated in Figure 1.2(b). An MPL of 500, marked with a red circle, was DBMS Y's thrashing point. Although several bumps between MPLs of 700 and 900 were observed, they were part of DBMS Y's thrashing phase. Thus, we see the bumps in the thrashing phase.

Figure 1.2(c) illustrates thrashing observed in DBMS Z. Its throughput kept increasing in the

smooth phase until an MPL of 600. Thereafter, we observed that thrashing occurred in DBMS Z. The thrashing point was the MPL of 600, marked with a red circle. The bumps at MPLs of 800 and 1,000 were still in the thrashing phase.

Figure 1.2(d) also shows thrashing that occurred in MySQL. (In the figure, two processors—one core with hyper-threading enabled—were provided. For one processor, we didn't see thrashing. What happened was that as an MPL value increased, the MySQL's throughput increased and then got saturated at an MPL of 600. Thereafter, the throughput began to drop but didn't thrash until an MPL of 1,000. Thus, we suspect that MySQL would have eventually thrashed if we kept increasing up to an MPL of 2,000 or more.) The throughput overall kept increasing and reached the top at an MPL of 800. Thereafter, the throughput precipitously dropped at an MPL of 900; then, the thrashing phase persisted. The thrashing point was the MPL of 800 marked with a red circle.

Thrashing was also observed in PostgreSQL, as shown in Figure 1.2(e). Its throughput continuously grew in the smooth phase and was highest at an MPL of 700. Thereafter, the throughput suddenly fell off; PostgreSQL's thrashing was detected right after the thrashing point marked with a red circle. Despite the bump at an MPL of 900, the thrashing phase persisted.

(The cause of the weird bumps was identified in our logs. We saw in those logs that the DBMSes temporarily denied and later accepted incoming sessions. The transactions from the accepted sessions got to be completed, thereby making such bumps by increasing throughput. We suspect that although the number of maximum sessions allowed was greater than an MPL of 1,000, too many outstanding transactions overwhelmed the DBMSes and thus were blocked before the bumps occurred. Other possibilities can be examined in the future work.)

These experiments demonstrate two things. One is that thrashing occurs in modern DBMSes. Second is that this thrashing phenomenon is observed across all DBMSes studied.

The thrashing behavior in DBMSes is a serious concern to DBAs (database administrators) engaged in developing OLTP or OLAP systems, as well as DBMS implementors developing technologies related to concurrency control. Weikum [89] also mentioned "Although data-contention thrashing is certainly not a common everyday problem, there are real applications, for example, in banking and financial trading, where system administrators are concerned about it (even when fine-grained, row-level locking is used)."

In general, if thrashing occurs in DBMSes, thousands of transactions may be aborted, resulting in making little progress in transaction throughput over time. From an engineering perspective, it is of critical importance to understand the factors of DBMS thrashing.

Understanding the origin of modern DBMSes' thrashing is challenging. Let's revisit Figure 1.2. "Did the thrashing happen because primary keys were not specified the tables?" "Is it because of too many rows simultaneously requested for reads by many transactions?" "Is it because transactions' response time suddenly went up?" "Did the thrashing stem from enabling only one processor?" "Are there any other factors involved?" "Would it be the case that these or other factors correlated with each other and then together contributed to the thrashing?" These questions can be raised for better understanding why the thrashing occurred across DBMSes.

The real problem, however, is that there is no existing *structural causal model*—a diagram showing how origins are related in a structural form—that can clearly explain why DBMSes thrash. If such a model existed, then DBAs and DBMS developers would better understand the root causes of DBMS thrashing and perhaps would predict the occurrence of thrashing when running their workloads on their DBMSes.

Therefore, the purpose of this dissertation lies in constructing the first structural causal model to achieve an initial understanding of the causality of DBMS thrashing.

1.2 Approach

The following is our thesis statement, for pursuing a better explanation of the causality of DBMS thrashing.

"*We propose and test a novel structural causal model that explicates the phenomenon of thrashing. We then draw novel implications from the model for engineering and tuning DBMSes for better performance.* "

The thesis statement implies the following three research questions:

1. What factors do impact on DBMS thrashing?
2. How do the factors of DBMS thrashing *relate*?
3. How much *variance* does each factor influence DBMS thrashing as well as by all the factors in concert?

The database community has overlooked these three important questions regarding DBMS thrashing. In this dissertation, we approach the questions in a fundamentally different fashion from an analytical and simulation method taken by a rich body of existing literature [3, 13, 18, 25, 43, 44, 76, 80, 81, 82, 83, 92, 93].

The first question concerns what factors can contribute to DBMS thrashing. To answer this question, we need to conduct literature survey and utilize our prior knowledge regarding transaction processing. Once this question is responded, then we can identify as many relevant factors as possible, and in turn, we can prioritize them along with their expected strength on DBMS thrashing for further investigation on the following question.

The second question above explores the relationship between the identified factors. We should take into account the fact that the factors may be correlated with each other, or even be moderated by, mediated through, or moderated-mediated by another factor, in the presence of a phenomenon, or DBMS thrashing. The way we respond to this question is to construct a structural causal model of DBMS thrashing and test the relationships exhibited in the model. If we can successfully uncover the relationship, the established model can help DBMS implementors, DBAs, and database researchers to better understand how the factors are associated with each other, and what factors directly or indirectly influence DBMS thrashing. That is, the model can exhibit the "causality" between the factors to better understand the thrashing phenomenon across DBMSes.

The last question concerns a quantitative approach to explaining DBMS thrashing. To answer this question, we first collect empirical data on actual DBMSes. The data can be obtained by conducting rigorous experiments through *operationalizing* (setting or measuring) each variable (factor) of the model. In the experiments, DBMS thrashing is treated as a *dependent* variable. As mentioned earlier, DBMS thrashing is measured as the thrashing point tolerated by DBMSes. Each factor is treated as an independent variable. After controlling specific values of *independent* variables, we then measure the thrashing point in the experiments. The measured data can be used to test the relationships hypothesized from the model; that is, in the test we can conduct a variety of evaluations including regression, correlational, path, and causal mediation analyses with the data.

Regression [24] involves identifying the relationship between the dependent variable and one or more independent variables. The way we perform the regression is to derive a regression equation on the dependent variable related to independent variables and then calculate the value of *R-squared* (R^2) [24] for the equation. R^2 is the ratio of the explained variance to the sample

variance of the dependent variable. The computed R^2 tells us how well the model can fit the measured data, or how much variance of DBMS thrashing can be explained by the model. As a result, if the R^2 is close to 1 (100% variance), we have identified all contributing factors. If the R^2 is close to zero (0% variance), the model is not useful, as the factors included in the model cannot explain much of the variance . Whenever the R^2 is between 0 and 1, this implies that there are factors not included in the model. Note that high R^2 values are exceedingly rare in most disciplines. Generally, each factor when incorporated into the model makes a discernible increase in R^2, the amount of variance explained.

Correlational [24] analysis can be conducted on the measured data, in conjunction with the regression to better understand why a factor was excluded; i.e., if a variable was excluded in a regression model, there can be some reasons that (i) it was not correlated with dependent variable, or (ii) it was highly correlated with another independent variable already in the model, or (iii) in the absence/presence of a third independent variable, the effect of the third variable on the independent variable on the dependent variable are opposite directions. If the coefficient value of a factor is positive (negative) and high (low), then the factor is said to have a strong (weak) positive (negative) influence on DBMS thrashing.

Path analysis [56] can be performed to account for the proportions of variance and the correlations among variables in the model. Path analysis can be complementary to correlational analysis, in that one benefit is to see which variables can be included in the model. We can also see via path analysis 1) how the path of the variables can be ordered in the model and 2) whether a (direct or indirect) path is significant or not to the model.

Causal mediation analysis (CMA) [30] can also be conducted to better understand mediation effects among variables. Mediation, to be discussed in Chapter 4, indicates that an effect of an independent variable (X) is propagated through an intermediate variable (M) to an dependent variable (Y). If the path analysis reveals the existence of M on a significant path to the dependent variable, CMA can help further identify the mediation relationship among X, M and Y.

If we can determine in concert R^2, coefficient values of factors, significant paths, and the presence of mediation in the model, database researchers can explore other factors (when $R^2 < 1$) and know which factors or paths are significant and which paths are concerned about mediation. DBMS developers can also better engineer their DBMSes by minimizing or eliminating significant factors and paths contributing to thrashing.

To answer these three questions, this dissertation aims at constructing a novel structural causal model for better explicating the thrashing occurrence in DBMSes. The established model can be utilized for the following three main purposes: 1) providing with DBMS implementors and DBAs not only a new engineering implication that was not known but also further evidence of existing implications, 2) helping database community better understand the phenomenon of thrashing, and 3) guiding database researchers to investigate other unknown factors that contribute to DBMS thrashing.

1.3 Contributions

This dissertation presents the following contributions.

- Elaborates a methodological perspective that treats DBMSes as experimental subjects and uses that perspective to study DBMS thrashing.
- Proposes the first structural causal model to explicate the origins of DBMS thrashing in transaction processing.
- Presents and tests a space of hypotheses that in concert through further investigation can refine and support (or not) the proposed model or suggest a better model.
- Extends a novel research infrastructure, called AZDBLAB, to schedule and run large-scale experiments over multiple relational DBMSes, providing empirical data to evaluate the proposed structural causal models.
- Proposes a novel thrashing analysis protocol (TAP) that can be applied to the collected data,
- Conducts rigorous statistical analyses on the empirical data retained through TAP,
- Suggests engineering implications to DBMS developers and DBAs, for further improving DBMSes' transaction processing,
- Guides database researchers to examine other unknown factors, contributing to DBMS thrashing.

1.4 Organization

This dissertation consists of twelve chapters and one appendix. In Chapter 1, we introduce the problem of DBMS thrashing. The following chapter describes the novel research area from which our thrashing study originates. We then explore several factors affecting thrashing by reviewing existing literature, and then discuss how these identified factors can be operationalized to observe thrashing in our experiments. Chapters 4 through 10 discuss in detail our novel empirical methodology to study DBMS thrashing. We propose a structural causal model based on the factors and their hypothesized correlations, and we provide a space of hypotheses to be tested on the model. We then present our research infrastructure, designed to manage large-scale experiments and extended to support thrashing-specific experiments used in this dissertation. Subsequently, we describe how to collect empirical data through experiments, and we propose a data analysis protocol for thrashing measurement. We in turn revise the proposed model that reflects several technical challenges that are encountered while running the experiments. Next, we conduct exploratory evaluation of the revised model and present the results. We further refine the revised model to reflect changes brought from the exploratory results, and we then propose the final structural causal model of thrashing. After that, we conduct confirmatory evaluation of the final model and present the results. Chapter 11 draws several engineering implications from the final model and suggests future research directions on this thrashing topic. In Chapter 12 we conclude by summarizing our findings and mentioning remaining opportunities to improve the amount of variance explained by the final model. In the Appendix we present the detailed common and concrete logical schema for data collection, include all the statistical analysis outputs for evaluating the model, and provide the actual code written for observing DBMS thrashing in our experiments.

CHAPTER 2

ERGALICS

In this chapter, we introduce the general concept of *ergalics* and then show a few our database ergalics projects relevant to the thrashing study in this dissertation.

2.1 General Concept

The term ergalics has been defined as "a natural science of computational tools" [45, 67, 70]. In other words, ergalics pursues the science on computational tools and computation itself. The goal of ergalics is to articulate and evaluate scientific theories of computational tools and of computation itself, and therefore, to investigate general theories and laws governing the behavior of these tools in various contexts.

Ultimately, ergalics leads to better understanding of interesting phenomena occurring in these computational tools or computation itself. The deep understanding through ergalics can provide a basis for better-engineered designs in software systems. (That is where ergalics can be utilized.) To reach better understanding such phenomena, ergalics makes uses of empirical generalization, as shown in Figure 2.1.

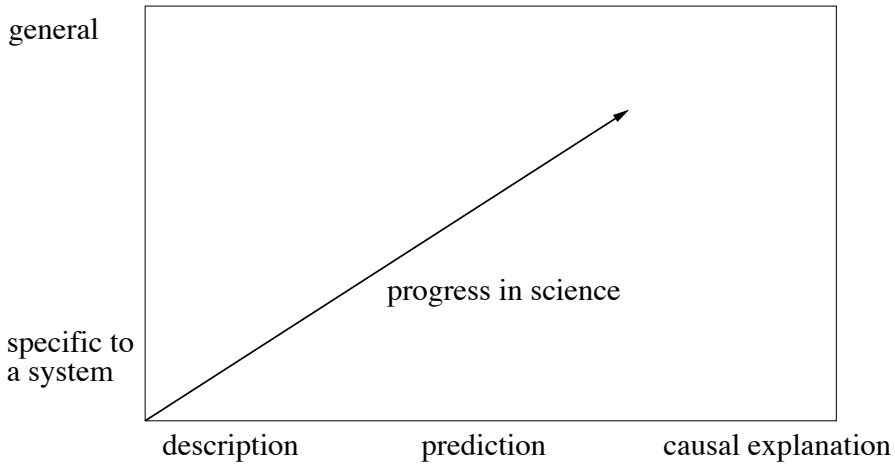


Figure 2.1: Empirical Generalization (from [7])

Empirical generalization moves along two conceptual dimensions. As described in Cohen's book [7], science in general, including ergalics, progresses on the *x*-axis and *y*-axis in the figure. Ergalics moves on the *x*-axis in the figure, from describing phenomena to being able to make predictions to deep causal explanation. This way, we can reach better understanding. Ergalics also moves on the *y*-axis from statements about one specific system to statements that hold true for all instances within a class of systems, to general statements across classes.

The class of computational tools and computation to which empirical generalization utilized by ergalics is applied is fairly wide. The class covers algorithms, hardware, formalisms, languages, and software systems. If we expand the sub-categories of software systems, we can include compilers (e.g., GCC), DBMSes (e.g., IBM DB2 [61], MySQL [50], Oracle [51], PostgreSQL [78]), GUIs (e.g., Java Swing, MFC), network protocols (e.g., DNS, TCP/IP). internet browsers (e.g., MS Internet Explorer, Google Chrome, Mozilla Firefox), operating systems (e.g., Linux, Windows), and software development environments (e.g., Eclipse, MS Visual Studio).

Ergalics focuses on achieving generality that can be applied across systems. To reach the generality, ergalics explores a variety of factors rather than different mechanisms. (It is evident that identifying a factor (e.g., concurrency) present across the systems is more general than discovering a mechanism (e.g., latches [42]) specific to a system.) It then establishes and validates the relationships between (or among) factors.

Specifically, ergalics uses a scientific methodology consisting of a sequence of the following tasks. A researcher 1) tries to identify as many factors as possible based on his intuition and existing literature survey, 2) constructs and articulates a "structural causal model" based on the hypothesized relationships between the identified factors, 3) designs an experiment to observe the phenomenon and collects empirical data by running the experiment on an experiment subject, 4) conducts exploratory evaluation of the model using the data, 5) makes possible adjustments to the model before, after, or even in the middle of the evaluation, 6) runs the same experiment and gather new data, and 7) conducts confirmatory evaluation of the refined model. Once the identified model is confirmed, the researcher can draw some engineering implications of the confirmed model.

In the following section, we walk through two ergalics projects associated with databases that we have been involved with. These projects help us frame the same empirical methodology that can be applied to the subject of DBMS thrashing in this dissertation.

2.2 Database Ergalics

In the database field, while very strong mathematical and engineering work has been done, the scientific approach has been much less prominent. The deep understanding of DBMSes obtained through the scientific approach can lead to better engineered designs. Database ergalics focuses on better understanding relational DBMSes as a general class.

As an example, we found that DBMS optimizers sometimes choose a non-optimal (suboptimal) plan for a query. This phenomenon is called *query suboptimality* [9]. Suboptimality is indicated by the existence of a query plan that performs more efficiently (or runs faster) than the DBMS chosen plan, for the same query. To better understand the impact of different factors on the suboptimality of query performance and the association between operators, we utilized the ergalics' empirical generalization strategies, treating DBMS subjects as a general class.

To examine the suboptimality phenomenon, we needed accurate query execution time measurement. However, we found it very difficult to measure query time, because of intrinsic variation in query time, to be discussed in the following section.

2.3 Exploring the Causal Foundation of Query Time

The study of query suboptimality required us to measure a number of query executions, while conducting our experiments to observe the suboptimality phenomenon. However, it was surprisingly hard to obtain accurate and precise measurements of the time spent executing a query, because of many sources of variance in query time on Linux. We attempted to explain why a particular measurement wasn't repeatable using a structural causal model on measures available in Linux. The experience of building such a model greatly helped us construct a structural causal model of thrashing.

A Structural Causal Model: Linux in general provides per-process and overall measures for each Linux process. The per-process measures of interest to us are under `/proc/`*pid*`/stat`, and the overall process measures under `/proc/stat`. Query time is dominated by CPU and I/O components. Among these measures a DBMS query process on Linux is characterized by the following measures: (a) *user time* (in ticks), the number of ticks in which that process was running in user mode, (b) *system time* (in ticks), the number of ticks in which a request from that process was being handled by the operating system, (c) *major fault* (in counts), the number

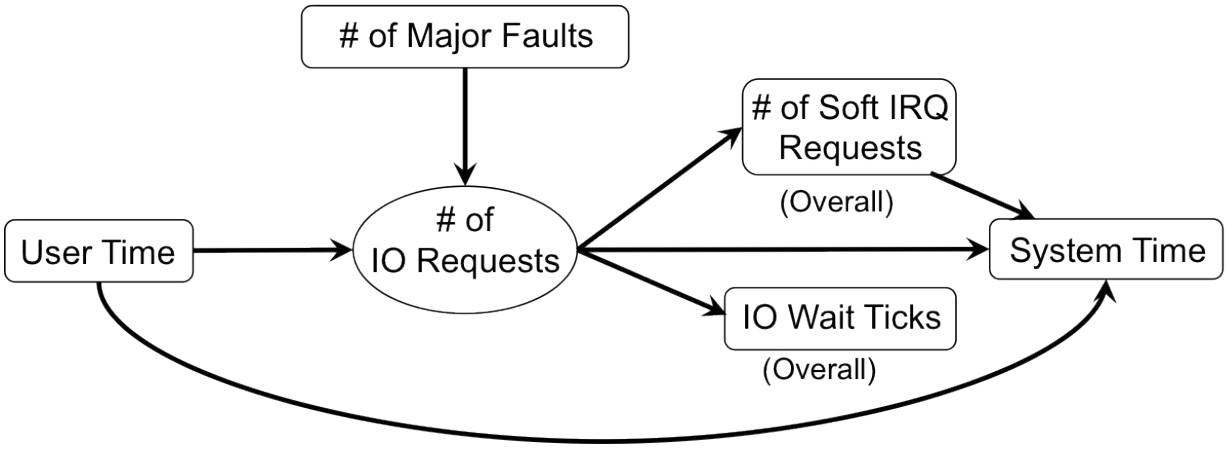
of major page faults, which cause the process to be blocked while that page is swapped in, thus incurring I/O, (d) *softIRQ* (in counts), the number of soft interrupt requests, (e) *IOWait* (in ticks), the number of ticks in which the system had no processes to run because all were waiting for I/O. For user and system time, we have both per-process and overall measures. For major faults, we have a per-process measure, and for softIRQ and IOWait, we have overall measures.

We developed an initial structural causal model of these measures, as shown in Figure 2.2(a). Within this composite model, nodes are variables to be measured and directed arcs hypothesize correlations. The variable denoted "# of IO Requests," a count of such requests, cannot be measured, and thus it is considered a latent variable, depicted with an oval.

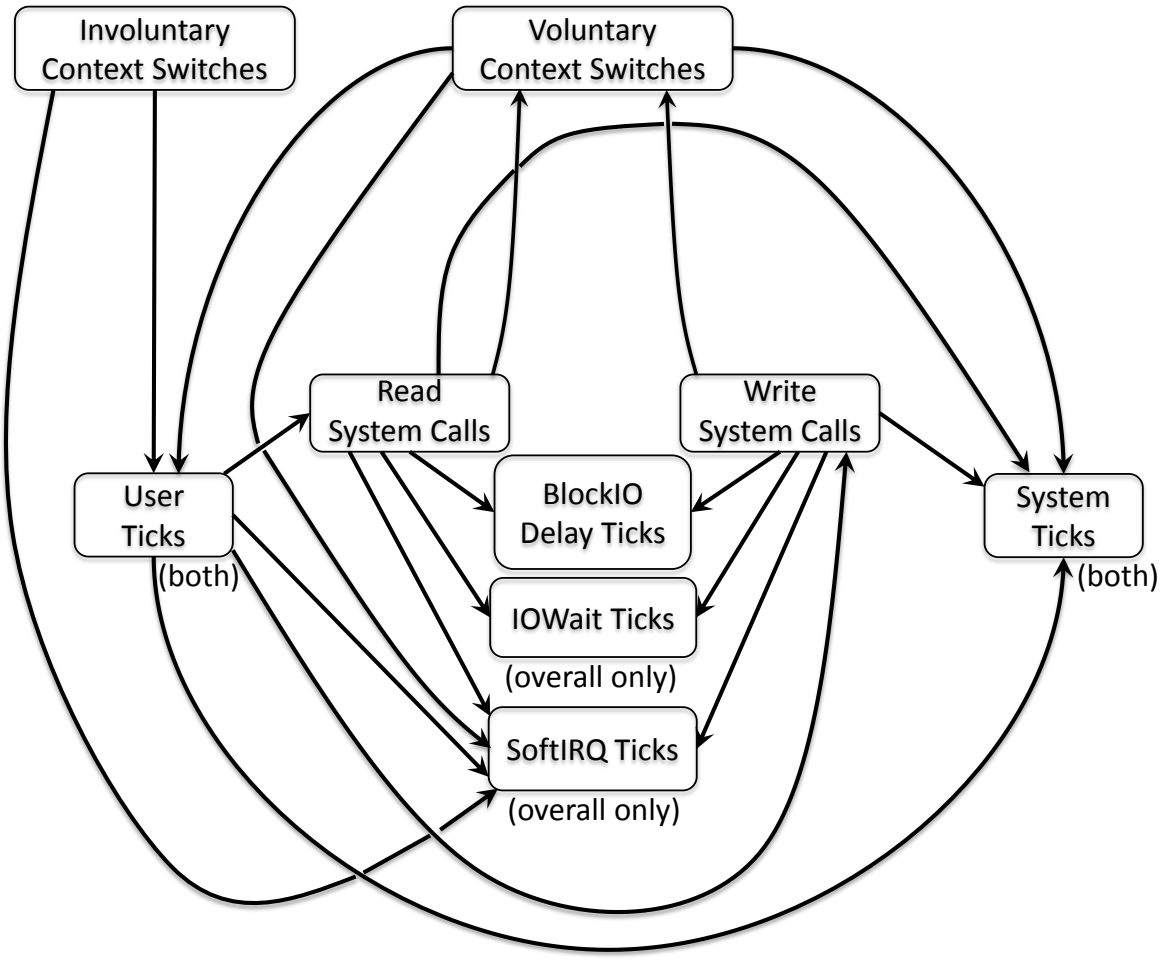
Later, we found that there existed more relevant measures not only under the aforementioned directories (`/proc/`*pid*`/stat` and `/proc/stat`), but also under `/proc/`*pid*`/status`, and `/proc/`*pid*`/io` in Linux. We also leveraged a C struct `taskstats` [62] provided by NetLink facility (a system allowing user-space applications to communicate with Linux kernel), in order to catch a terminating process(es) before completing query execution in our experiments. This struct aligns fairly well with `/proc/`*pid*`/stat`.

Figure 2.2(b) provides an extended model of these measures. In addition to the previous measures in Figure 2.2(a), we added to the extended model more measures: (f) *involuntary context switches*, the number of context switches occurring when a process explicitly yields its CPU to another, (g) *voluntary context switches*, the number of context switches occurring when the system suspends a process and switches its control to another, (h) *read system calls*, (i) *write system calls*, and (j) *blockIO delay ticks*, the total time the process has been blocked on synchronous IO. For (h), (i), and (j) we have per-process measures. Since these measures related to I/O become available, we remove the latent measure of the number of IO requests, shown in Figure 2.2(a).

The intuition behind this model in Figure 2.2 is that the DBMS in its normal processing (measured by user ticks) reads data from the database, expressed as a read system call to read in the block. That system call incurs its own system time, additional blockIO delay ticks, and additional interrupt requests, some of which are softIRQs. If all processes are blocked on I/O, that request could add to the (overall) IOWait ticks. These predicted correlations are all positive. For example, if user time increases for a different query, it is predicted that the number of IO requests (Read or Write System Calls) may increase, which could itself increase softIRQ ticks, IOWait ticks, blockIO Delay ticks, and amount of system time. The DBMS presumably does other things



(a) An Initial Model



(b) An Extended Model

Figure 2.2: Models of Linux Process Behavior (from [10] and [11])

that require system time. As timer interrupts are in fact soft interrupts, user time directly affects softIRQ ticks as well. Each read or write system call that requires data to be read from disk, will likely induce a context switch if there is another process scheduled, hence the number of context switches is directly correlated with such system calls. The number of voluntary context switches directly increases the time spent in system mode, hence increasing the system time.

An Empirical Methodology for Evaluating the Model: In this methodology, we treated DBMS experiment subjects as black-box, incorporated each subject as a plugin into our central laboratory system, called AZDBLAB [72], to be discussed later. We then developed a Java-code based scenario of measuring query time, ran the scenario on each subject plugged into AZDBLAB, and collected query execution data measured in the scenario. We also needed to inspect the validity of the measured data. We thus developed a timing protocol, termed the Tucson Timing Protocol (TTP) [11], which was applied to our empirical data with 2.4 million query executions collected over almost a year.

2.4 Exploring the Causal Foundation of Query Suboptimality

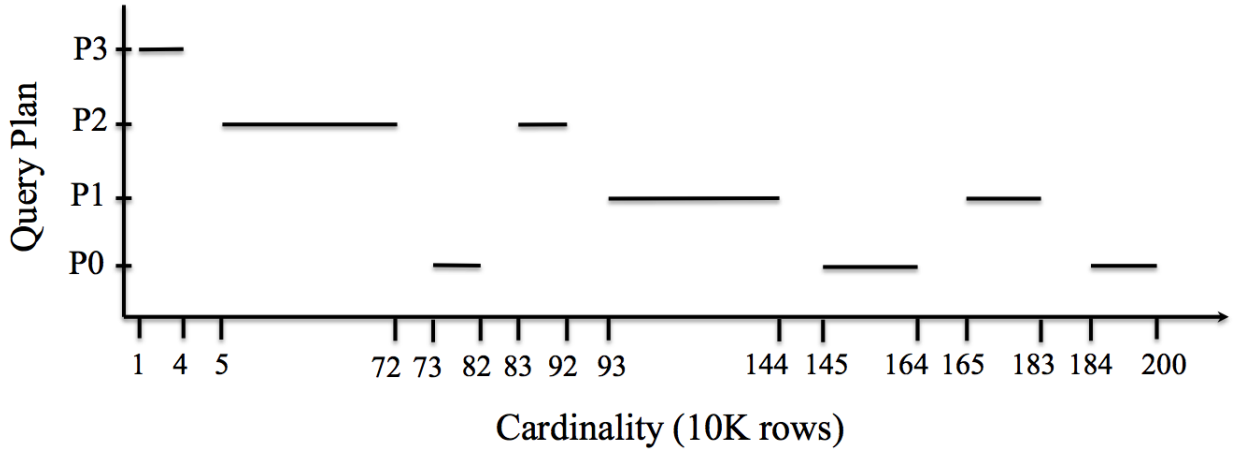
We developed another structural causal model of query suboptimality, based on our experience we obtained in the query time variation study. The experience of constructing this model provided great insight into how to organize thrashing variables into a model, how to develop a scenario for observing DBMS thrashing, how to operationalize variables, and how to evaluate the model of thrashing. Let's now look at how we approached the suboptimality problem.

Consider a simple select-project-join (SPJ) query, with a few attributes in the `SELECT` clause, a few tables referenced in the `FROM` clause, and a few equality predicates in the `WHERE` clause. This query might be an excerpt from a more complex query, with the tables being intermediate results.

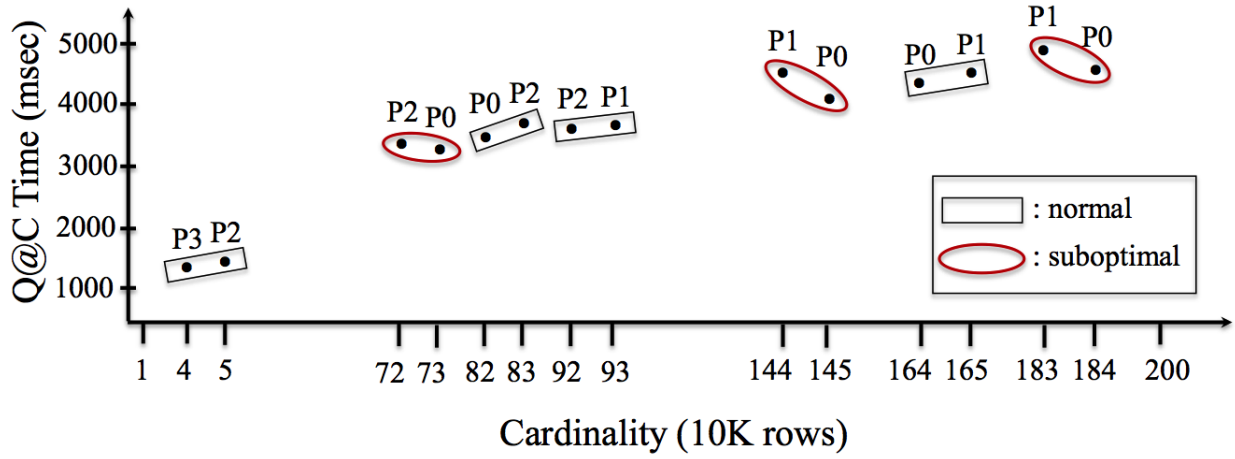
```
SELECT t0.id1, t0.id2, t2.idt, t1.id1
FROM ft_HT3 t2, ft_HT2 t1, ft_HT1 t0
WHERE (t2.id4=t1.id1 AND t2.id1=t0.id1)
```

The optimizer chooses different plans for this query as the cardinality of the `ft_HT1` table varies, an experiment that we elaborate later in depth.

Figure 2.3(a) represents the plans chosen by a particular DBMS as the cardinality of `FT_HT1` decreases from 2M tuples to 10K tuples in units of 10K tuples. The *x*-axis depicts the estimated cardinality and the *y*-axis a plan chosen for an interval of cardinalities. Thus, Plan P0 was chosen for 2M tuples, switching to Plan P1 at 1,830,000 tuples, back to P0 at 1,640,000 tuples, and so on, through the sequence P0, P1, P0, P1, P2, P0, P2, and finally P3 at 40,000 tuples.



(a) Chosen Query Plans



(b) Plan Change Points

Figure 2.3: An Example of Suboptimality (from [9])

Figure 2.3(b) indicates the query times executed at adjacent cardinalities when the plan changed, termed the "query-at-cardinality" (Q@C) time. For some transitions, the Q@C time at the larger cardinality was also larger, as expected. But for other transitions, emphasized in red ovals, the Q@C time at the larger cardinality was smaller, such as the transition from plan P2 to P0 at the pair at 720,000 tuples. Such pairs identify suboptimal plans, as P2 required 3.2sec while P1 at a

larger cardinality required 3.1sec. For the pair at 1,440,000 tuples, P1 required 4.9sec whereas P0 at a larger cardinality required only 4.6sec. This query exhibits seven plan change points, three of which are suboptimal.

From an engineering perspective, the goal is to understand the prevalence of suboptimality and its factors. We identified factors relating query suboptimality, and we constructed and tested a structural causal model to explain the suboptimality phenomenon.

A Structural Causal Model: Figure 2.4 shows the predictive model of query suboptimality. The model concerns four general constructs that we hypothesize will play a role in suboptimality: *optimizer complexity*, *query complexity*, *schema complexity*, and *plan space complexity*. Some constructs include several specific variables that contribute to that construct as a whole. Our model distills many of the widely-held assumptions about query optimization. The model's contribution is the specific structure of the model and the specific operationalization of the factors included in the model. For the detailed explanation of this model, refer to our paper [9].

The Evaluation of the Structural Causal Model: To evaluate the model, we needed to collect empirical data from DBMSes through a series of experiments managed by AZDBLAB. However, we were challenged by the fact that we wanted a methodology that worked for proprietary DBMSes: we could not make any internal changes to the DBMS. We, thus, developed a methodology that applied both to proprietary and open-source DBMSes, that ensured the repeatability of a query, and that obtained a query execution time.

Based on the query execution data retained throughout TTP [11], we conducted a thorough correlational analysis, providing strong support for the model. Note that our experience of developing such a methodology—including the protocol—greatly contributed to establishing a similar methodology employed for our thrashing study.

Engineering and Tuning Implications: After the evaluation of the model, we uncovered several surprising results that provide systematic clues as to where current optimizers come up short and how they can be further improved.

Here are some of the lessons we learned from the suboptimality study.

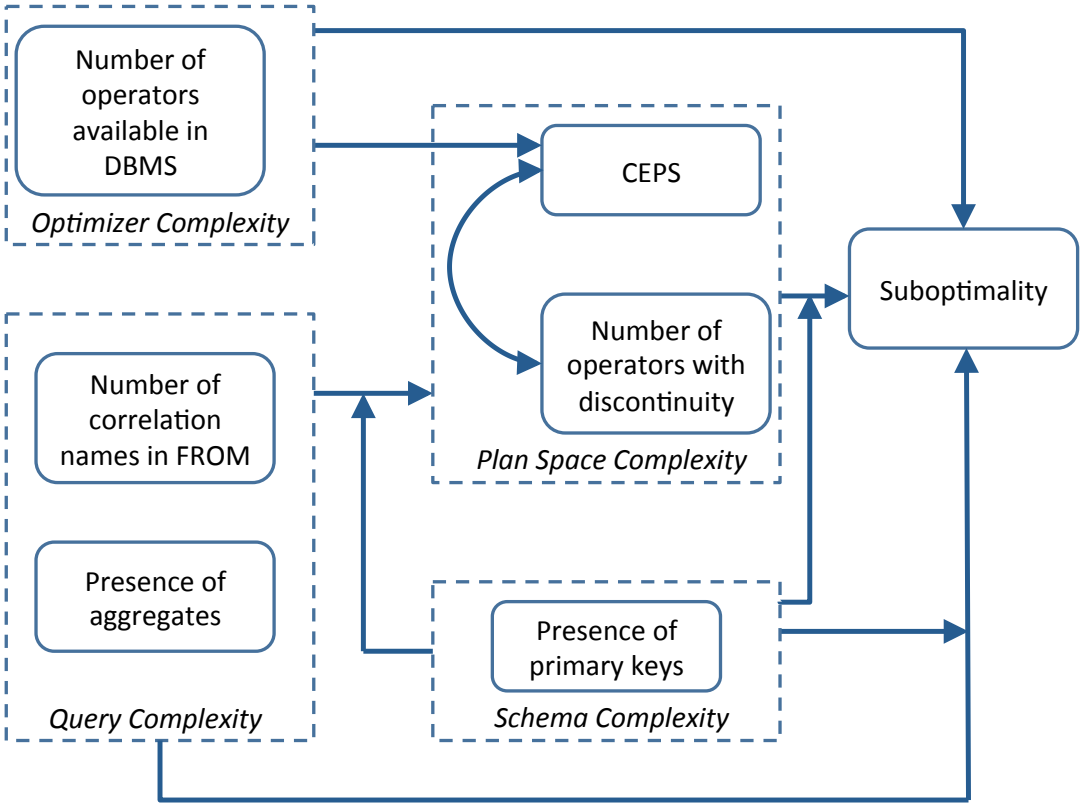


Figure 2.4: A Predictive Model of Suboptimality [9]

- For many queries, a majority of the ones we considered, the optimizer picked the wrong plan for at least one cardinality, even when the cardinality estimates were completely accurate and even for our rather simple queries.
- A quarter of the queries exhibited significant suboptimality ($>= 20\%$ of the runtime) at some cardinality.

These two results indicate that there is still research needed on the query suboptimality topic. Fortunately, the causal model in Figure 2.4 helps point out specifically where that research should be focused.

- Some queries showed significant *query thrashing*, with a plan change at almost every cardinality. While this phenomenon was first visualized by Haritsa et al. [12, 23] on some complex queries, we have shown that it is present even in a surprising percentage of simple queries.

- Furthermore, some queries exhibited many changes *to a suboptimal plan* as the cardinality was varied.
- The causal model and our experimental results suggest that more research is needed to improve the cost model of *discontinuous operators* (e.g. hash join).
- We also show that it may well be useful to explicitly take *cardinality estimate uncertainty* into account.
- This research indicates that aggregates are *not* a problem, so that aspect of query optimization is in good shape.

For additional lessons, refer to our paper [9].

The methodology introduced in our suboptimality study suggests fairly specifically where additional engineering is needed (the cost model of discontinuous operators and accommodating cardinality estimate uncertainty) and is not needed (costing of aggregation).

The same ergalics methodology, explained in Section 2.1, is applied in this dissertation, to better understand the origins of DBMS thrashing. In the following chapter we discuss what kind of variables can influence thrashing.

CHAPTER 3

EXPLORING FACTORS OF THRASHING

This chapter discusses related work of DBMS thrashing and its shortcomings compared to our work, presents the taxonomy for measuring thrashing, and identifies several factors of DBMS thrashing.

3.1 Prior Work

There is a rich body of existing literature concerning load control and thrashing in transaction processing systems. Much of this work was done from the 1980's until the early 2000's [3, 13, 43, 44, 76, 83, 89, 93], and there were several more recent contributions [27, 28, 35, 37].

In general, there are three ways to understand a phenomenon (such as thrashing). The first way is to build and evaluate an analytical model. Tay [76] used an analytical model and simulation to understand, compare, and control the performance of concurrency-control algorithms. He studied the effect of changing MPL, transaction length (number of locks), and granularity of locks [21] (e.g., intention shared, shared, intention exclusive, and exclusive lock mode for concurrent accesses on hierarchical database objects—files, pages, and records). He then constructed an analytic model and compared the model's predictions with his simulation results. He suggested that data contention thrashing was due to blocking rather than to restarts, and resource contention (competing for a transaction to finish the computation) caused thrashing to occur earlier. Moenkeberg also pointed out that too many active transactions in the system could incur excessive lock requests [44].

Thomasian introduced a methodology based on *queuing theory*, to evaluate the performance of concurrency control methods in transaction processing systems [84]. Suppose that in a closed queuing system there are transactions with a total of (k+1) steps consisting of k steps with mean duration (s) and an additional step for releasing all locks. The following is one of his analytical models, calculating the mean block delay (W_l) (that is, average waiting time before the lock is released) of a transaction (T_k) in terms of an active transaction holding the requested lock of T_k at

the j-th step:

$$W_l = \Sigma_{i=1}^{k} \frac{2j}{k(k+1)}[(k-j)(s+u)] + s' = \frac{k-1}{3}[r+u] + s', \tag{3.1}$$

where (k-j) is the number of remaining steps, u is the expected mean waiting time by a lock conflict per lock request, r is the mean transaction residence (elapsed) time when no lock contention, and s' is the remaining processing time of a step which the lock conflict happened.

Using this equation Thomasian determined that W_l proportionally increases with the number of locks held by the transaction (j). He later computed the parameters to determine the level of lock contention for standard locking and the onset of thrashing, Through a series of subsequent analytical models, he found out that the mean number of active transactions in the closed system increases with the MPL, reaching a maximum point before thrashing occurs (that is, the thrashing point described in Section 1.1, due to a snowball effect, referring to the blocking of transactions causing further blocking transactions.

Thomasian later summarized in his survey paper three factors that contribute to thrashing in a transaction processing system with two-phase locking (2PL) scheme [83]. He claimed that a transaction processing system is susceptible to thrashing under the following three circumstances: 1) when the *degree of transaction concurrency* (or MPL) is increased, 2) when average transaction length temporarily increases in the system, and 3) when *effective database size*, a subset of objects accessed by transactions (or degree of contentions), varies in time due to different database access pattern.

Hotspots, where data access is non-uniform, were noted as another possible cause of thrashing in previous studies [13, 93] using analytical models and evaluating the models by simulation. Hotspots indicate that most of data access requests (e.g., 80%) are concentrated on only a few (e.g. 20%) database objects. Hotspots make transactions requesting the same data serialized around its lock, thereby eventually leading to thrashing.

A benefit of using the analytical modeling approach is the conciseness of an analytic model. Once we fully understand the meaning of all the variables used in the model, it is not hard to follow the entire flow of how the model is derived. At the end, we can expect the nicely summarized model in a clean mathematical form.

As mentioned, simulation was used in many of these studies [3, 13, 43, 44, 76, 83]. One

advantage of using simulation lies in its flexibility [84], as a user can try a variety of configurations in a tool. Simulation can also be used to understand complex systems. Simulation tools for extended queuing network models, described in Lavenberg's book [38], are basically equipped with GUI. They are also designed to relieve a tiresome burden of developing simulations. (There are some other simulation tools such as DiskSim for disk subsystem simulation or ns2 for network simulation related to TCP, routing, and multicasting.) Measurements by simulation can be performed at multiple levels: for instance, processor and disk utilization and mean queue lengths can be measured at device levels, and response time and throughput can be measured at workload levels, while running the simulation of the model provided parameters and their specific values. The measured results can be provided for subsequent statistical analysis.

Most of the traditional works rely on simulation as well as analytical models. A drawback behind these analytical and simulation methodologies is that the analytic and simulation results may not hold true for actual DBMSes. Specifically, it is hard to generalize their results to real DBMSes. The analytical and simulation methods have a limitation in capturing the complex behaviors among transactions and resources (CPU, I/O, and memory) in a current DBMS.

Another limitation in prior work is that the recent architectural trend of *multi-core processors* was not reflected. Some of the work just discussed was done before multi-core processors existed.

The third way is to utilize an empirical approach using a real system. Recent studies [27, 28, 35, 37] used an actual DBMS. They examined transaction throughput bottleneck that emerged on a DBMS running on a multi-core system (open source DBMSes (MySQL [27, 37] or PostgreSQL [28]) or their own DBMS (Shore-MT) [35]). These works aimed at improving multi-core scalability in DBMSes. They identified major bottlenecks in scalability and provided relevant engineering solutions for the DBMSes.

A drawback of this approach is that each evaluation was conducted on one (or in a few cases, two) open-source DBMS(es). Their conclusions only applied to those DBMS(es). In particular, Jung et al. [37], attacking the scalability problem in DBMSes on multicores, emphasized: "Our evaluation has demonstrated a collapse of transaction throughput under high load, even read-only load, for all the database systems we analyzed. In the case of the open-source systems, MySQL and Shore-MT, we could identify latch contention in the lock manager as the bottleneck. While we could not perform the same in-depth analysis on the commercial DBMS X, its observable behavior is similar enough to the open-source systems to suspect that the cause is similar."

As mentioned above, their work did not demonstrate that their results could be generalized to other (non-open source) DBMSes. Considering that each DBMS is very different, it is hard to say that their analysis is generalizable to other DBMSes, and even applying the tailored solution to those DBMSes may not work, either. As demonstrated in Figure 1.2, the thrashing phenomenon is observed across DBMSes. This implies that it is of critical importance to study the thrashing behavior by regarding DBMSes as a general class of computational artifacts.

A recent study [94] attempts to expose and diagnose violations of atomicity, consistency, isolation, and durability (ACID) [20]—properties that modern database systems typically guarantee, under power failure. In this study the authors use a total of eight widely-used systems: three open-source embedded NoSQL (often interpreted as Not only SQL) [4] systems (TokyoCabinet [17], LightningDB [74], and SQLite [29]), one commercial key-value store, and one open-source (MariaDB [41]) and three proprietary OLTP database servers. The approach is somewhat similar to this dissertation, in that both works 1) concern transactions, 2) take an empirical method using real systems, and 3) identify root causes of a phenomenon observed in database systems. However, their work focuses on understanding the causality of the ACID property violation in the presence of electrical outages, while our work aims at understanding the causality of thrashing.

Another drawback is that none of existing works take into account structural correlations between factors contributing to thrashing. All previous investigations emphasize only a single factor. As more factors emerge and interact, their correlations cannot be ignored in understanding the causality of DBMS thrashing. As pointed out in our previous studies discussed in Section 2.2, we cannot ignore the correlations (like relative importance or moderation) between factors in understanding a phenomenon.

In short, existing work exhibits the following weaknesses: 1) simulation or analytical study of models rather than real DBMSes, 2) study restricted to one specific DBMS rather than multiple DBMSes, and 3) little consideration of relationships of factors.

To overcome these limitations, we utilize a novel, different empirical methodology: developing a structural causal model to better explain the DBMS thrashing phenomenon. Our empirical study 1) identifies variables affecting a phenomenon (e.g., thrashing), 2) constructs a structural causal model based on the variables and their hypothesized correlations, 3) designs how to operationalize (set or measure) the (independent or dependent) variables, 4) collects data measured while running

large-scale experiments across multiple DBMSes, and 5) tests the structural causal model via statistical techniques applied to the data. The model easily visualizes the relationships among identified factors of thrashing.

There are also some challenges using our empirical methodology. One is to how to experiment with a variety of DBMSes under the same controls. Since DBMSes are very different in terms of performance and code complexity. It is not easy to make the same or fair operationalization for a variable across very different DBMSes. The second challenge is to conduct such huge experiments over several DBMSes. Managing such experiments is tough, laborious, long-term.

3.2 Taxonomy for Thrashing Measurement

There is a spectrum of granularities—depending on what is being measured and how it is measured—in building a structural causal model of DBMS thrashing. Based on this spectrum, we identify the variables of the model and collect the variables' values for testing the model.

Figure 3.1 provides the taxonomy for measuring DBMS thrashing.

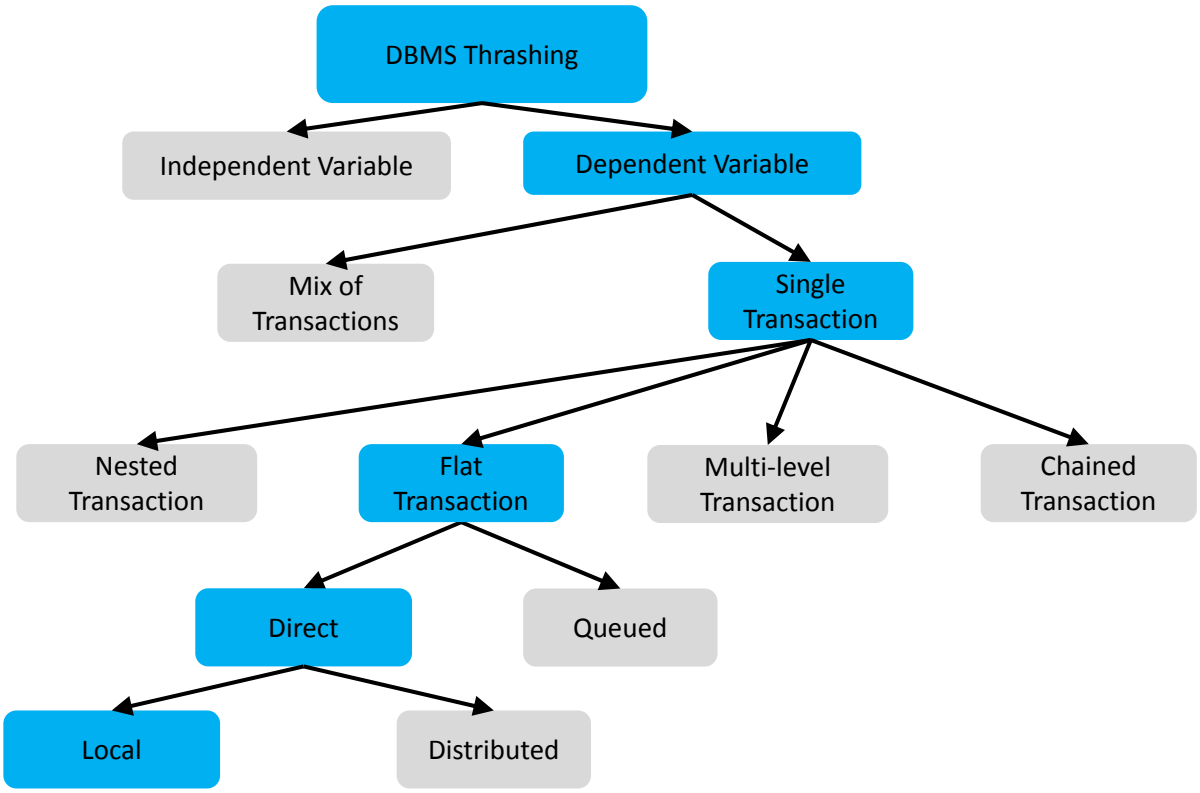


Figure 3.1: Taxonomy of DBMS Thrashing

In this taxonomy, DBMS thrashing is treated as a dependent variable, in that it is *observed*, not manipulated. We discuss in detail how to specifically measure DBMS thrashing in Section 3.4.5. There are several factors contributing to DBMS thrashing. These factors are treated as independent variables, in that they are *manipulated*, not observed. We observe the dependent variable of DBMS thrashing, by intervening (or, setting the values of) the independent variables. We introduce the independent variables in the next section. We elaborate in detail how to operationalize the independent variables in Section 3.4.

Transactions are used to observe DBMS thrashing. Transaction types can be divided into single or mixed. Single transaction type concerns a read-only or write-only transaction. A mixed transaction type involves both reads and updates within the same transaction. *In this dissertation, we cover the single transaction type*. The mixed transaction type is much more complicated than the single type. After reaching a sufficient understanding of the thrashing occurrence on the single type, we will proceed with the mixed type in the future work.

In the context of Jim Gray's categorization [22], our interest lies in a *flat transaction* that contains an arbitrary number of actions, that is, has the ACID properties [20], at the same level. A *nested transaction* is a tree of transactions, the sub-trees of which are either nested or flat transactions. A *multi-level transaction* is a special type of nested transaction that can commit ahead the results of the sub-transactions before the surrounding upper-level transaction commits. A *chained transaction* is the one that at commit passes its processing context to the next transaction and then implicitly starts the next transaction in one atomic operation. These other types of transactions will be considered in the future work, as they are more complicated than the flat transaction.

We focus on *direct transactions* where the terminal interacts directly with the server, but we do not cover *queued transactions* that are delivered to the server through a transactional queue interacting with clients. Lastly, we cover *local transactions* with no remote access. We will consider *distributed transactions* in future work because of their complexity.

Before identifying and operationalizing the factors of thrashing, we define several terms related to transactions in the subsequent section.

3.3 Terminology

A *transaction* is represented by a single SQL concerning read or update for convenience. A *client* is specified by a `Connection` instance created by a JDBC driver [48] configured for connecting to a DBMS subject, and it is implemented by a Java thread. Provided that the `Connection` instance has been already created, for the client to run its transaction, execute its commit, and finish its run, we 1) create a `Statement` object from the `Connection` object, then 2) invoke `Statement.execute()` from the `Statement` object by passing to that method an SQL statement representing the transaction, (to be presented in Section 3.4.3,) 3) call `commit()` to execute the commit of that transaction, 4) keep running the same transaction until a specified connection time limit, (to be described shortly,) is reached, and lastly 5) terminate the client's connection to that DBMS experiment subject.

A *batch* is a group of clients, each executing its own transaction. The size of a certain batch is equal to an MPL number. For the batch to run its clients' transactions, we create as many `Connection` objects as the size of the batch and then run each client's transaction in the batch via the aforementioned `Statement.execute()` in parallel. A *batchset*, called a "trial" in our study, is a group of batches whose size increases by steps of a certain number. The concept of a batchset is introduced to determine a specific thrashing point (as discussed in Section 1.1.1). In our study a batchset includes ten batches, each successively increasing by 100. Thus, the biggest batch in the batchset consists of 1,000 clients.

Figure 3.2 visualizes a batchset. As mentioned above, a batchset contains a set of ten batches, each consisting of a multiple of 100 clients. Each of the clients repeatedly runs its own transaction comprising a single SQL statement until the connection limit.

Now we introduce several casual factors, which are hypothesized to influence DBMS thrashing.

3.4 Identifying and Operationalizing Variables

Most of the factors have been mentioned in the literature [28, 37, 46, 83]. We have also included several independent variables (i.e., the number of processors, buffer space, transaction size, and referenced row subset region) to be shown shortly, based on our intuition of other factors (i.e., connection time limit) that can influence DBMS thrashing. We group these factors into four *constructs* in accordance with their relevance. (See Figure 4.1 on page 57.) These constructs will form our model to be presented in the following section. In the subsequent paragraphs we

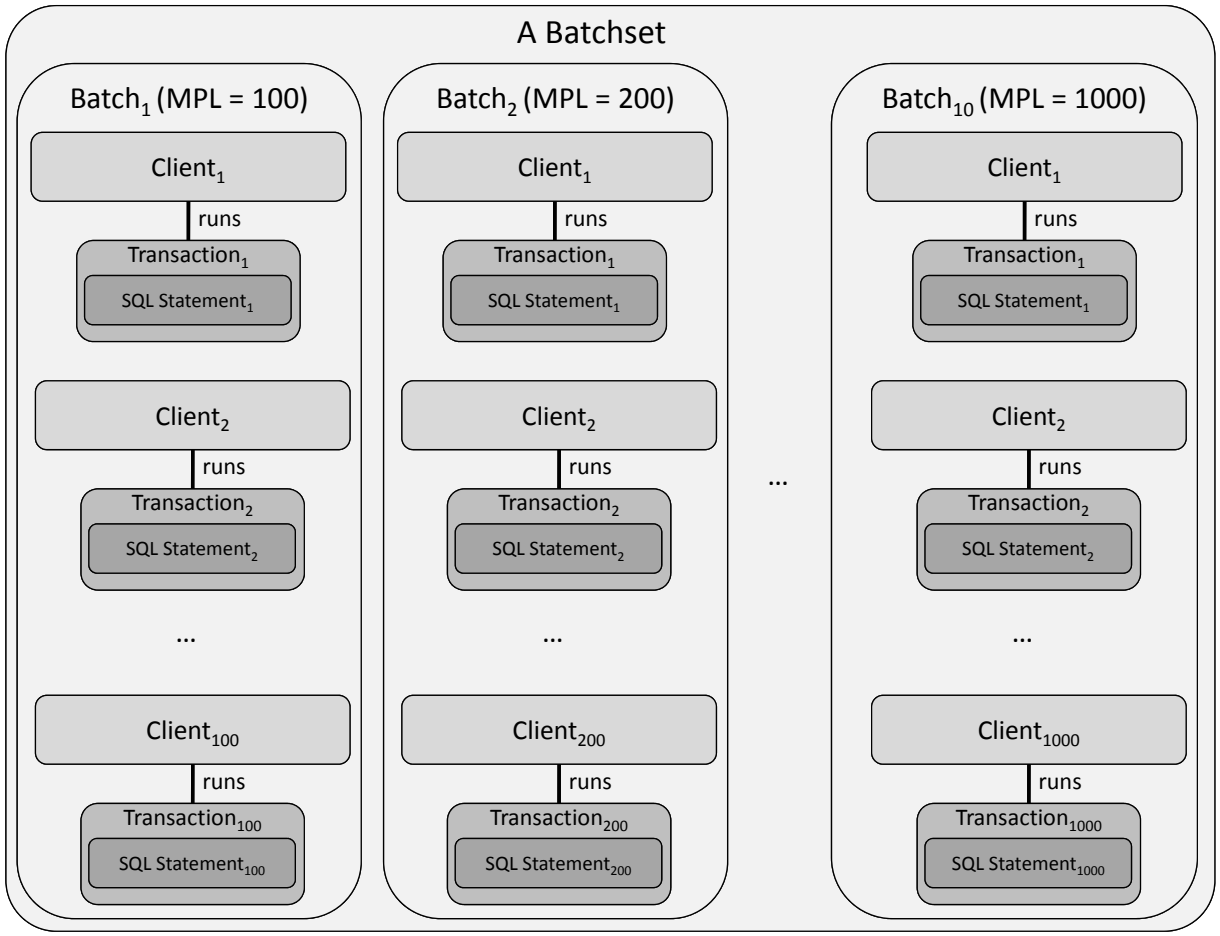


Figure 3.2: Batchset Visualization

introduce each construct and its specific variables and elaborate how each of the variables can be operationalized (that is, set or measured) in our experiment. Variable operationalization means how we actually set an independent variable and measure a dependent variable.

3.4.1 The Resource Complexity Construct

The *resource complexity* construct concerns system resources that a DBMS relies on. In this construct we include two variables: "DBMS buffer space" and "number of processors (numProcs)." These two variables are identified as important resources from which the DBMS can benefit when treating transactions.

DBMS Buffer Space: This is an independent variable capturing the size of the DBMS buffer cache (space). This variable is included because I/O is one of the dominant factors affecting transaction throughput [46]. Buffer space is typically correlated with disk I/O. A DBMS uses

its buffer cache to store objects fetched from disk for faster access subsequently. In general, if the buffer cache size is sufficiently big, the DBMS can avoid substantial disk I/O, as more data can be cached. The DBMS can then respond to user requests with the cached data without retrieving the data from disk again. The amount of disk I/O affected by the cache size will influence throughput.

We can operationalize this variable as follows. Defining 100% as the default buffer cache size provided by each DBMS, we can try 100%, 50%, and 25%. For example, one of our DBMSes provides a configuration parameter such as `DB_CACHE_SIZE`. Considering that 2GB is set by default in that DBMS, 2GB, 1GB, and 500MB can be correspondingly set to its buffer cache size by executing on its client tool the following SQL statement:

`ALTER SYSTEM SET DB_CACHE_SIZE =` *`integer [K | M | G]`*.

In MySQL we can set its buffer pool size using a configuration parameter, called `innodb_buffer_pool_size` [49]. To set 50% of the default size (128M), for instance, we can add in `my.cnf`, which is read when MySQL starts up, the following line:

`innodb_buffer_pool_size=64M`.

Likewise, for the other relational DBMSes the same set of percentages can be applied based on their default buffer cache size, by executing such an SQL statement or setting their own configuration variables.

Number of Processors: This is an independent variable capturing the number of processors that can be used by DBMSes. This variable addresses level of parallelism from which the DBMSes can benefit when treating their transactions. As pointed out in Section 3.1, recent studies [28, 37] have paid attention to the influence of concurrency by multiprocessors on thrashing. If we can utilize more processors, we can benefit from increased parallelism in processing. Enhanced parallelism may speed up transaction processing in DBMSes, which can service more transactions that are concurrently running. That said, the above studies actually report that as more processors are available to a DBMS, thrashing can occur because the degree of processor contention becomes greater as workloads increase. We thus need to better understand the impact of processors on thrashing.

The operationalization for this variable relies on machine specification. Our individual experimental machine is configured with four cores supporting hyper-threading. We use five

values for this variable: one processor specified by one core with hyper-threading disabled, two processors by one core with hyper-threading enabled, four processors by two cores with hyper-threading enabled, six processors by three cores with hyper-threading enabled, and eight processors by four cores with hyper-threading enabled. That is, we make a batchset run on a DBMS running on the experimental machine configured with one, two, four, six or eight processors. In Linux, we can alter numProcs by setting a specific number to "`maxcpus`" available in `/boot/grub/grub.conf`. For a different operating system running on one of our DBMSes, numProcs can be altered by our proficient lab staff via a BIOS option.

3.4.2 The Environment Complexity Construct

Environment complexity is the construct concerning the environmental constraints on clients running their transactions on a DBMS. We include in this construct the following one variable: "connection time limit" (CTL).

Connection Time Limit: This is an independent variable capturing a running environment of transactions in practice. Specifically, CTL captures how much time (in minutes) will be given to each client to run its transactions on a DBMS subject. For instance, CTL in Figure 1.2 was set to two minutes to every client in a batch.

This variable addresses how significantly the length of connection time of a client can affect the thrashing point. Some transactions will not finish if the client connection time was low. A small CTL will give less time for the transactions to start to conflict. Therefore, the overall transaction throughput will be low, perhaps leading to thrashing. On the other hand, a big CTL may give enough time to complete a transaction that was in conflict with other transactions. The overall throughput for the big CTL will be higher compared to that of the small CTL, and thus thrashing may get delayed or even fail to take place.

We can operationalize this variable, by providing a set of specific time lengths—two, four, and eight minutes—in an experiment specification to be discussed in Section 5.3. Each value of CTL is stored in a global variable in our thrashing scenario code to be presented later, and the variable is used within a method (specifically, `stepC`() at Line 19 in Listing 4 in our Appendix), in which we make sure whether the elapsed time since a batch run has started surpassed the specified value (e.g. 2 minutes). That is, we capture start time right before making the batch run, and during the

batch run in a `while` loop we check to see if the elapsed time relative to the start time is over the specified CTL value. If so, we immediately terminate the batch run.

3.4.3 The Transaction Complexity Construct

Transaction complexity is the construct governing transaction properties that cause DBMS thrashing. We include in this construct the following two variables: "selectivity factor" (SF) and "reference row subset region" (ROSE), to be described shortly in the subsequent paragraphs. These two variables capture transaction size specified by the number of rows requested by an individual transaction (intra-transaction) and the contention among transactions (inter-transaction), respectively.

Transactions used in our experiment use the database schema as described in Section 1.1.2. The schema is composed of two tables, each consisting of seven columns, five of which are non-null integers, and the other two are variable-sized character columns. A value for the first integer column, called `id1`, is sequentially generated (and used for operationalizing SF). Each tuple is 200 bytes long, consisting of the total 20 bytes from the integer type columns and the total 180 bytes from the variable-sized character columns. Each table is populated with 1M [1] tuples having column values randomly generated except the first integer column.

Transaction Size: This variable has been introduced by Thomasian [83]; it captures the number of objects requested by transactions. As discussed elsewhere [83], transaction size can impact transaction throughput. If transaction size is big, that is, a transaction has access to many objects, many locks may be issued, and accordingly, many lock conflicts will occur when the requested objects are already held by other transactions. The lock management overhead can create substantial latency in transaction processing. Thus, transaction throughput may fall significantly. On the other hand, if transaction size is small, that is, a transaction references a few objects, then fewer locks will be issued, and fewer lock conflicts will be incurred, and thus a chance of thrashing will decrease.

Transaction size can be specified by *selectivity factor* (SF) [64] of a read or update transaction. SF is defined as the ratio of output to input tuples: an estimate of what fraction of input tuples will

[1] As mentioned earlier in Section 1.1.1, we populated 30K tuples on one of the DBMSes, as the DBMS suffered from running out of memory on populating 1M tuples for an unknown reason.

be returned as output tuples given a relation (table) in a database.

The way we operationalize the variable is to vary the number of rows accessed by each transaction. As discussed in Section 3.2, the type of transaction is read-only or write-only. For simplicity, we use a single SQL statement within each transaction, to be exemplified shortly.

A read-only transaction can be specified by a `SELECT` statement. For a read-only transaction, defining 1% as the maximum SF, we vary the SF from 1% to 0.01% by factors of 10. If SF is set to 1%, then we generate a transaction reading 1% of consecutive tuples in a randomly-chosen table.

A write-only transaction can be specified by an `UPDATE` statement. We don't use `INSERT` and `DELETE` for the write-only transaction. It is because `INSERT` and `DELETE` statements change the cardinality of a table, which is not desirable. However, using `UPDATE` can preserve the same semantics for operationalizing the write-only (hereafter, update-only) transaction, without incurring any cardinality change.

Unlike a read-only transaction where the maximum SF is 1%, for a update-only transaction we vary the SF linearly. Instead we try 1%, 0.75%, 0.50%, and 0.25%. So if SF is set to 1% we generate transactions updating 1% of tuples in a randomly-chosen table. The reason for taking different scales for each type of transaction is that we think the linear variation of update-only SF is enough to see thrashing, considering that an update-only transaction incurs *exclusive* locks while a read-only transaction incurs *shared* locks.

The SQL statements associated with the read-only and update-only transaction will be shown shortly after we discuss the following variable.

Reference Row Subset Region: This variable captures the region of rows (objects) that can be referenced by transactions. The variable has been also termed *effective database size* in a previous paper [83]. The variable called "referenced row subset region (ROSE)" indicates the maximum region of rows that can be accessed by the transactions. ROSE is specified as a percentage. The transactions can be restricted to access only a subset of rows—for instance, the first quarter (25%) or the first half (50%) of the rows—in a randomly-chosen table in our schema.

ROSE can matter to DBMS thrashing. If ROSE is too small, the degree of contention on referencing the same rows will significantly increase. Substantial locking overhead can be incurred during the transaction processing. In other words, *hotspot regions* around the rows in high

contention can be created. If ROSE is large, on the other hand, the contention will decrease. A greater number of rows in the extended ROSE, however, can be referenced by transactions. Accordingly, a lot of locks on these rows can be issued, thereby charging heavy lock management overhead and thus contributing to significant drop in throughput.

We can operationalize this variable in the following. For each transaction in a batch we first randomly choose a table from our database schema and define the entire table rows as 100%. We can then decrease the proportion of the rows to 75%, 50%, and 25% from 100% for ROSE. For example, if ROSE is set to 50%, then a transaction can be restricted to reference only the first half rows in the randomly-chosen table. If ROSE is set to 100%, then all the rows in the table can be referenced by the transaction. The value of ROSE will determine the range of rows that a transaction can reach, and it will be used for specifying and restricting the transaction. In the next paragraph we show what a transaction run by a client looks like in our experiment.

Given a combination of values of SF (s) and ROSE (r) as described above, a read-only or update-only transaction to be generated will look like the corresponding one as follows:

```
SELECT column_name
FROM table_name
WHERE id1 >= a and id1 < b
```

or

```
UPDATE table_name
SET column_name = v
WHERE id1 >= a and id1 < b
```

where *table_name* = a randomly-chosen table in the schema, a = an integer value randomly chosen in the interval of [0, ($c \times r$) - ($c \times s$)], $b = a + c \times s$, c = the cardinality of the table (1M or 30K), and v = an integer value or a character string randomly generated along with a chosen column (*column_name*), and `id1` is used as sequential row ID.

Note that our transaction is very fast, and thus, it can be typically completed within around 1 msec. Depending on systems contexts and a chosen DBMS, however, the completion time varied significantly. For instance, we observed that in the worst case it took about five minutes to finish a single transaction. That said, such a completed transaction is not counted, as the five-minute completion time exceeds our connection time limit of two-minutes. In addition, our transaction is less computational but I/O-dominant, as can be seen above. Since I/O is regarded as one of the

dominating factors affecting DBMS thrashing, we expect that in our experiments these transactions will address the effect of I/O on thrashing. In the future work we will consider compute-bound transactions as well.

3.4.4 The Processing Complexity Construct

Processing complexity is the construct concerning the transaction processing capability of a DBMS. In this construct we include the single variable of transaction response time.

Transaction Response Time: This is an important factor contributing to DBMS thrashing. The response time includes computation and I/O time needed for performing a transaction. It also includes additional processing delay, mainly caused by lock conflicts ("synchronization overhead") [83]. These conflicts can occur among transactions accessing the same objects for the purpose of read and write. If there is no lock conflict, transactions can be processed without or with minimum delay. Otherwise, the locking overhead will be included in response time. In particular, this variable captures different performance on the same transactions across different DBMSes and reflects the influence on thrashing caused by distinct code and performance of different DBMSes.

This variable can be specified by *average transaction processing time* (ATPT). We cannot directly control the variable, but we can measure this variable. To operationalize (measure) this variable, we need to leverage a batchset. As illustrated in Section 3.3, we generate a batchset containing ten batches each consisting of 100, 200, . . . , 1,000 clients, respectively. Note that each client in a batch has its respective transaction generated based on the combination of values set for SF and ROSE as described above. We then present the batchset to a DBMS subject; at each value of MPL (equivalent to a batch) in that batchset we run all the transactions of the clients in that batch, measure via JDBC, and record the elapsed times of the transactions executed on that DBMS until a given CTL (as described in Section 3.4.2) as is reached. Once the run of the batchset is completed, then we average the recorded elapsed times for all the transactions executed in the batchset. We set the averaged elapsed time to ATPT (average transaction processing time) associated with the batchset. This method can be applied to any DBMS. It is hard to directly observe what locking schemes on objects (pages, tables, tuples, and columns) are employed by different DBMSes, but we think that the calculated ATPT can not only reflect the impact of the schemes on the thrashing point but also capture different characteristics among the DBMSes with respect to code and performance.

3.4.5 The Thrashing Point

This variable is the dependent variable of DBMS thrashing. The variable captures a specific value of the MPL where DBMS thrashing occurs in a given batchset; it allows us to quantitatively measure the degree of thrashing observed in a DBMS. We observe the value of the thrashing point in the following. We 1) generate a batchset along as described in Section 3.4.4, 2) run the transactions in the clients in each batch in the generated batchset on that DBMS (in the same way as ATPT is measured), 3) count the number of executed transactions in that batch after a specified CTL elapses and then derive TPS (transactions per second) for that batch, 4) repeat the same batch run multiple (three or five) times and compute the average TPS for that batch, 5) calculate the average TPS for each of the remaining batches in the batchset in the same way, and finally 6) derive the thrashing point for the batchset based on the following Equation 3.2:

$$\text{The thrashing point} = \begin{cases} |B_i| & \text{if } \textit{TPS}_i > 1.2 \times \textit{TPS}_{i+1}, \text{ and } \forall j > i+1, \textit{TPS}_i > \textit{TPS}_j \\ \infty & \text{otherwise,} \end{cases} \tag{3.2}$$

where $i > 0$, B_i is the i-th batch in a given batchset, and $\textit{TPS}_i$ is the measured TPS for B_i.

A specific thrashing point is equal to the size of a batch (that is, a particular MPL), at which the measured TPS value is more than 20% of the TPS measured at the subsequent batch, and it is greater than the rest of TPS measured at all the subsequent batches in the same batchset. For instance, the thrashing point of DBMS X in Figure 1.2(a) is an MPL of 800 ($|B_8|$), as the TPS measured at an MPL of 900 ($|B_9|$) was beyond 20%, lower compared with that of the previous MPL of 800. Also, the TPS at the thrashing point was higher than the TPS at the last MPL of 1,000. Note in the figure that each point (or MPL), far apart by steps of 100, corresponds the size of each batch in the batchset used in that experiment. Once this thrashing phase begins, we expect that the phase change should remain same until the last MPL is reached in a trial. The calculated thrashing point quantitatively represents how many MPLs can be tolerated by a DBMS for the batchset. The 20% threshold was determined, based on our observation that once a measured TPS dropped by 20% or more than a previous TPS (at the thrashing point), there was no case that subsequent TPSes were bigger than the thrashing point's TPS. (See our sanity check results to be exhibited in Figures 8.5 and 10.5.)

In the following chapter, we construct a structural causal model of thrashing based on the variables, and we also describe the hypothesized correlations drawn from the model.

CHAPTER 4

A PROPOSED STRUCTURAL CAUSAL MODEL OF THRASHING

This chapter proposes a structural causal model that explicates the origins and the inter-relationships of DBMS thrashing.

There is no existing structural causal model for explaining the thrashing phenomenon. Using the variables introduced earlier in Section 3.4, we have developed a preliminary model for DBMS thrashing, as exhibited in Figure 4.1.

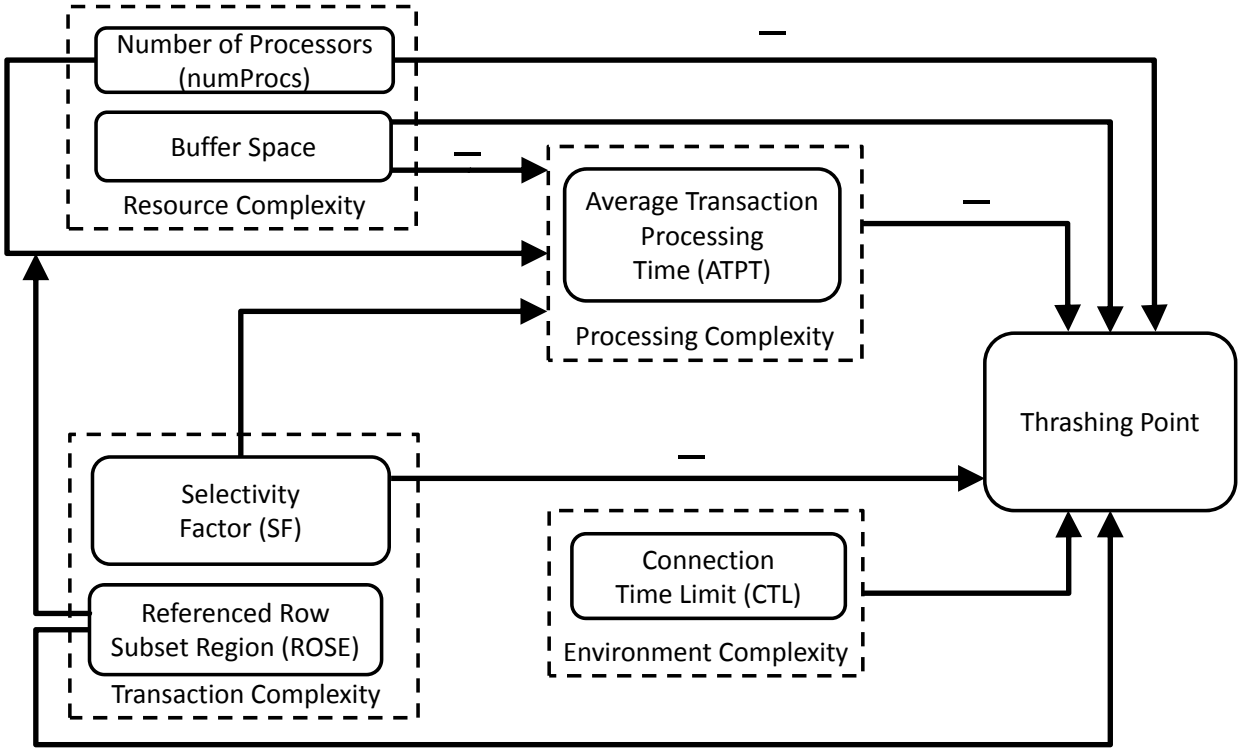


Figure 4.1: A Proposed Model of Thrashing

Our model has one dependent (outcome) variable, representing DBMS thrashing, on the far right. This variable is represented by the "thrashing point" in the model, as discussed in Section 3.4.5. Also, the model includes several independent variables affecting the thrashing point. These variables come from the identified factors introduced in Section 3.4.

Our model suggests how the variables are structurally organized to explicate the origins of thrashing. The model contains several relationships between or among the variables. Each of the relationships is represented by an arrow in the model.

The model includes some arrows above which a minus sign is marked. The sign '-' indicates a negative correlation between two variables, meaning that as a variable X increases, a dependent variable Y decreases. For instance, the top arrow in the model represents such a negative correlation between numProcs and the thrashing point; namely, as the number of processors increases, the thrashing point will decrease. On the other hand, the arrow without the minus sign indicates a positive correlation between two variables. As buffer space increases, for example, the thrashing point will increase. We discuss each arrow with its direction shortly.

As discussed in Section 3.4, our model is composed of four general constructs: resource complexity, transaction complexity, environment complexity, and processing complexity. Each construct is represented in a dashed line. Some of the constructs, such as resource complexity and transaction complexity, include multiple variables that contribute to that construct as a whole.

In the resource complexity construct we have two variables of buffer space and numProcs, as mentioned in Section 3.4.1. The model expects that the lack of the system resources may affect thrashing.

In the environment complexity construct we have the variable of CTL as described in Section 3.4.2. The model expects that the lack of CTL in running transactions may contribute to thrashing.

In the transaction complexity construct we have two variables of ROSE and SF as mentioned in Section 3.4.3. The model expects that these two variables associated with workload characteristics cause thrashing as well as affect the ATPT variable, to be discussed in the subsequent paragraph.

In the middle of the figure is the construct of processing complexity described in Section 3.4.4. We have in this construct transaction response time specified by the following variable: "average transaction processing (ATP) time." In the model it is dependent on some of the variables (buffer space and numProcs) on its left and exerts influence on the outcome variable (thrashing point) on its right. The relationship among these variables—buffer space, numProcs, ATPT, and thrashing point—in the model is called *mediation* [24]. The model expects that the processing complexity construct specified by ATPT behaves as a *mediator* [26] contributing to thrashing.

Mediation captures the effect of an intermediate variable (M) located between an independent variable (X) and a dependent variable (Y). Mediation indicates a relationship, such that X carries two effects on Y, one *direct* from X to Y, and the other *indirect* through M. Changes in X influences changes in M, which in turn influences changes in Y. In Figure 4.1, X corresponds to buffer space or numProcs, Y to the thrashing point, and M to ATPT treated as a *mediating* variable. The model expects that ATPT carries the indirect effects of buffer space and numProcs on the thrashing point. The indirect effects represent how the thrashing point is influenced by buffer space and numProcs through a sequence; buffer space and numProcs affect ATPT, which subsequently affects the thrashing point. The arrows from buffer space and numProcs go through ATPT to the thrashing point.

Furthermore, the model reveals that there is another relationship between numProcs and ATPT, which is conditional along with ROSE. The relationship among these variables— numProcs, ATPT, and ROSE—is called *moderation* [24].

Moderation represents an association between two variables X and Y when the strength or direction depends on a third variable M. In Figure 4.1, X corresponds to numProcs, Y to ATPT, and M to ROSE. In the moderation the effect of X on Y is influenced or dependent on M. Specifically, see the arrow pointing from ROSE to the arrow from numProcs to ATPT in the model. We have the association between numProcs and ATPT, and we expect that the direction or level of the association will be moderated by variation in ROSE. The association will be conditional on the changes of ROSE, which is treated as a *moderating* variable or *moderator* [26].

Note particularly that the moderation concerns the mediation in the model. Such a type of a relationship is known as *moderated mediation* [58]. Moderated mediation indicates that there exists a mediating effect of an independent variable (X) through a mediator (M) on a dependent variable (Y), and the mediation through M is conditional on another variable (W). The model expects that there exists the mediation through ATPT from numProcs and buffer space to the thrashing point, and the mediation associated with numProcs is conditionally influenced by or dependent on the variation in ROSE; that is, the indirect effect of numProcs is conditional on ROSE through the mediation of ATPT to the thrashing point. (The indirect effect of buffer space is also mediated through ATPT, but it is independent of ROSE. Observe that there is no arrow from ROSE onto the association between buffer space and ATPT in the model.)

For more detail on mediation, moderation, and moderated mediation in a model, refer to Hayes'

book and other references [24, 26, 30, 31, 32, 58, 85]. We discuss testing the existence and significance of these relationships in the model in Section 8.6.

Given these four constructs and seven specific variables depicted in Figure 4.1, let's now examine the correlations between these variables. Such relationships are specific associations between the constructs (or, their variables), as hypothesized by this predictive causal model.

One factor of our model is the resource complexity construct. We hypothesize that the resource complexity construct has influence over DBMS thrashing both directly, as depicted by the top line, and indirectly, via the processing complexity construct.

Consider numProcs first. Multiprocessors present a great opportunity to DBMSes. Multiprocessors enable DBMSes to process more transactions than a single processor does, due to increased parallelism. Hence, transaction processing in DBMSes can be sped up by using multiprocessors.

There is, on the contrary, a drawback in taking advantage of multiprocessors. Enabling more processors could incur serious competition among the processors in utilizing shared resources available within the DBMS internals. The increased processor contentions might hurt transaction throughput. As reported in recent studies [28, 37], the processor contentions seem to offset the increased parallelism. We therefore hypothesize the following.

Hypothesis 1: As the number of processors increases, the thrashing point will decrease.

We now turn to the relationship with processing complexity. As mentioned in Section 3.4.1, the increased parallelism using more processors can speed up transaction processing. Although overall throughput could fall off due to the contentions among the processors, we anticipate that as far as individual transaction response time is concerned, it will be faster when more processors are presented. We thus hypothesize as follows.

Hypothesis 2: As the number of processors increases, ATPT will decrease.

Consider buffer space. When data pages are accessed by transactions, if the pages do not exist in the buffer pool, then the pages must be fetched from disk and brought into the buffer pool. If the buffer pool does not have room to accommodate the new data pages, the buffer manager of the DBMS should first evict some pages resident in the buffer pool and then bring new pages into the buffer frames of the evicted pages. If subsequent transactions request the flushed pages, the DBMS should read the flushed pages back from disk and place them back into the empty frames,

after purging other pages resident in the pool. The lack of buffer space could add substantial latency, attributed to I/O [46]. The increased latency could reduce transaction processing time.

If the buffer pool space is sufficiently big, on the other hand, the buffer manager can place into the frames all the requested pages without victimizing any pages in the buffer. Therefore, DBMSes can save much I/O and the latency of transaction processing will be much shorter.

Consider the correlation with the processing complexity construct. The lack of DBMS buffer cache could bring substantial disk I/O, as described above. If transactions request data pages not resident in the buffer pool, which is already full, some buffer pages should be flushed out to disk. Then, the requested blocks should be read from disk and place them into the clean frames.

If subsequent transactions request the flushed pages, some other buffer pages should be evicted, and the previously evicted blocks should be brought back into the clean buffer frames. This I/O overhead significantly increases ATPT. If the buffer pool is large enough to keep all the pages referenced by transactions, such disk I/O will not be needed. We therefore hypothesize as follows.

Hypothesis 3: As buffer space increases, the thrashing point will increase.

Hypothesis 4: As buffer space increases, ATPT will decrease.

We now turn to the processing complexity construct. ATPT directly influences DBMS thrashing. Specifically, as transactions' response time gets shorter, the overall throughput can rise. On the contrary, as the response time gets longer, the throughput may significantly fall at a certain point, resulting in thrashing. Therefore, we hypothesize the following.

Hypothesis 5: As ATPT increases, the thrashing point will decrease (i.e., thrashing will occur earlier).

Consider the transaction complexity construct. We hypothesize a negative correlation between SF and the thrashing point. As SF goes up, many rows are requested by transactions. If these rows are already held by other transactions, the lock management overhead on the rows will be charged to DBMSes. In particular, when update-only transactions request many rows, a lot of exclusive locks may be around in the DBMSes. If the rows are already occupied by other transactions, the transactions should then wait until the rows are available after the locks are released. For this reason, the transactions in lock conflicts cannot be processed quickly, and accordingly, the overall throughput may be saturated and later fall off because of the increased lock management overhead. Among read-only transactions the overhead may not be as significant as update-only transactions,

since there are no lock conflicts on the requesting rows. Still, we expect that there should exist some overhead managing the shared locks on the rows. Therefore, we hypothesize the following.

Hypothesis 6: As SF increases, the thrashing point will decrease.

We hypothesize that SF has a positive correlation with ATPT. Specifically, the more rows are requested by transactions, the more I/O is required. The incurred I/O will add a significant latency in transaction processing. In particular, if the transactions concern updates, and the requested rows are already occupied by other transactions, the lock conflicts will further delay the transactions' response time. We thus hypothesize the following.

Hypothesis 7: As SF increases, ATPT will increase.

We hypothesize a positive correlation between ROSE and the thrashing point. As ROSE grows in size, the contentions among update-only transactions requesting the same rows may compete less due to fewer lock conflicts. Accordingly, the thrashing point may be increased. As ROSE gets shrunk, on the other hand, the degree of contention on the rows will be much heavier. The increased contention will lead to more lock conflicts, leading to thrashing resulting from a significant drop in throughput. For read-only transactions the correlation may be not so significant. However, we still think that there exist to some extent lock management overhead among the same rows requested by the read-only transactions. Therefore, we hypothesize the following.

Hypothesis 8: As ROSE increases, the thrashing point will increase.

We hypothesize that ROSE will moderate the correlation between numProcs and ATPT. Specifically, as numProcs increases, ATPT will decrease, as described in Hypothesis 2. As ROSE decreases, however, ATPT will increase, considering that as numProcs increases, processor contentions will increase as reported in previous studies [28, 35, 37]. The direction of the relationship will change as ROSE decreases.

Hypothesis 9: ROSE will moderate the strength of the correlation between numProcs in the resource complexity construct and ATPT in the processing complexity construct on the thrashing point.

We hypothesize that as CTL decreases, the thrashing point will decrease (move leftward). When the same rows are requested by update-only transactions running on DBMSes, a lot of lock conflicts will occur. The DBMSes then need some time to handle the conflicts, by waiting for the locks in conflict to be released, or detecting and resolve deadlocks if any. Moreover, DBMSes

	a	*b*	*c*	*d*
I	numProcs / ATPT *low*	numProcs / Thrashing Pt. *high*	Buffer Space / ATPT *high*	Buffer Space / Thrashing Pt. *high*
II	SF / ATPT *high*	SF / Thrashing Pt. *high*		ROSE / Thrashing Pt. *medium*
III	N/A	CTL / Thrashing Pt. *medium*		
IV	ATPT / Thrashing Pt. *medium*			

Table 4.1: Hypothesized correlations and Levels on the Proposed Model

may block any incoming transactions while resolving the conflicts. If CTL is short, DBMSes may run short of time to treat transactions in contention or to unblock and then resume processing the blocked transactions within CTL. Even in the case of read-only transactions, as CTL decreases DBMSes may also run out of time to treat all the transactions within CTL, as a significant I/O may be incurred especially when SF is high. A significant drop in throughput results from the lack of CTL. For these reasons, we hypothesize the following.

Hypothesis 10: As CTL increases, the thrashing point will increase.

The proposed causal model relates five interventional independent variables and two non-interventional but measurable dependent variables. We have a total of nine correlations including three correlations with ATPT and six correlations with the thrashing point. We identify expected correlation factor of 0.7 or greater as a "high" level of correlation, from 0.3 to 0.7 as "medium" and those below that as a "low" level of correlation [9]. We group these expected correlations that act in similar ways in Table 4.1. In each box is a correlation between two variables and a predicted level. We designate such pairs by group and letter within group. We focus on the directly-linked correlations in Figure 4.1. An example is correlation (I*a*) the box at the top left, which concerns the correlation between numProcs and ATPT.

Group I indicates the correlations involving the resource complexity construct with the processing complexity construct and the thrashing point. I-(a) concerns the correlation between numProcs and ATPT. We expect the level of this correlation to be low: the enhanced parallelism by the increase of processors will improve ATPT, but since transactions are I/O-dominant and less computational, the benefit will be hidden by the substantial I/O latency. I-(b) indicates the

correlation between numProcs and the thrashing point, and its strength is expected to be high. As previously pointed out, we predict that processor contentions will be one of the significant factors contributing to thrashing. I-(c) represents the correlation between buffer space and ATPT. We expect that the correlation's strength to be high, as response time will be dominated substantially by I/O. I-(d) is the correlation between buffer space and the thrashing point. The strength of I-(d) is expected to be high, because the amount of I/O, one of the dominating factors affecting throughput [46], should be very sensitive to the size of buffer space.

Group II shows the correlations involving the transaction complexity construct with ATPT and the thrashing point. II-(a) indicates the correlation between SF and ATPT. We expect the strength to be high, as SF will mainly determine the amount of I/O strongly affecting ATPT as described above. II-(b) represents the correlation between SF and the thrashing point. The level of the correlation will be high, as SF will mainly determine the amount of I/O substantially influencing transaction throughput as mentioned before. II-(d) is the correlation between ROSE and the thrashing point. ROSE will affect the thrashing point very little when ROSE is big. However, if ROSE is too small, the contentions on the same rows will significantly rise, leading to thrashing. In that case the strength will be high. We thus expect the overall strength of the relationship between ROSE and the thrashing point to be in the middle, or medium.

Group III shows the correlation between CTL and the thrashing point. Overall, the strength of III-(b) is expected to be medium. If CTL is short then thrashing will occur, as elaborated in Hypothesis 10. If CTL is long, on the contrary, thrashing will be less likely to occur as transactions are given enough time to be processed, helping DBMSes to keep steady throughput over time.

Group IV shows the direct association between ATPT and the thrashing point. ATPT is considered one of the influential factors on the thrashing point. Increased ATPT reduces the number of completed transactions within unit time, resulting in a significant drop in overall throughput. In contrast, decreased ATPT will get DBMSes to process more transactions within unit time. Thus, the overall throughput will go up, thereby preventing thrashing from occurring early. Therefore, we expect the overall strength of the correlation to be medium.

In the following chapter, we present our research infrastructure helping us to collect empirical data for evaluating the proposed model.

CHAPTER 5

THE AZDBLAB SYSTEM

In this chapter we introduce the AZDBLAB research infrastructure for running experiments. This infrastructure has been designed for supporting large-scale empirical DBMS studies [72]. AZDBLAB has been around for seven years. Many collaborators have contributed to different pieces of AZDBLAB. Doctoral student Rui Zhang served as a chief programmer (CP) of AZDBLAB until Summer 2010, and since then we have taken over his CP position, assuming responsibility for operating and advancing the AZDBLAB system. We have also extended the AZDBLAB system to support this study of transaction processing.

Figure 5.1 presents the architecture of AZDBLAB. AZDBLAB consists of decentralized monitoring schemes (Observer, mobile apps, web apps, and watcher), the lab shelf DBMS server, executors, experiment subjects (DBMSes), and scenario plugins. We have contributed to the maturity of the system by not only implementing and enhancing many components (Observer, executors, experiment subjects, scenario plugins, and lab shelf schema design), but also assisting other students to develop and improving some other components (mobile apps, web apps and watcher) in AZDBLAB. In the subsequent sections we explain in more detail each component.

5.1 Lab Shelves

A lab shelf comprehensively stores all the data provenance related to experiments. The schema of a lab shelf captures *who*, *what*, *when*, *which*, *where*, *why*, and *how*, complying with the 7-W model [60]. It is a fully append-only database [66]. More detail on the schema is provided in our appendix A.1 and A.2.

The lab shelf data are managed by a dedicated DBMS server running on a separate machine. We have contributed to enhancing the design of the lab shelf schema with other collaborators and to managing the lab shelf server and the data.

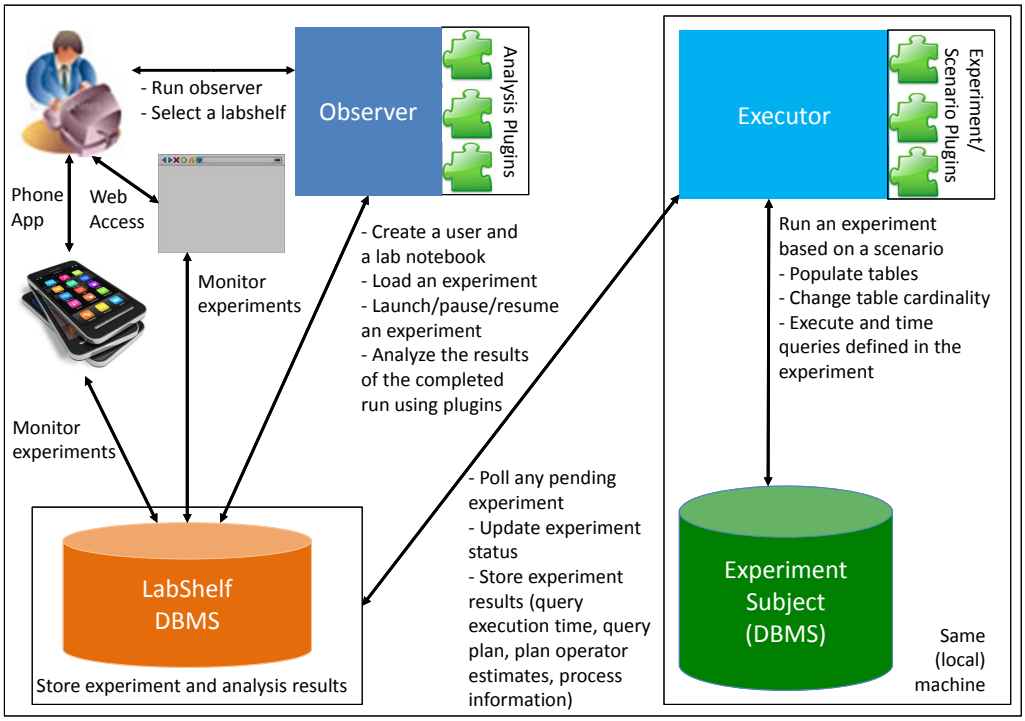


Figure 5.1: AZDBLAB Architecture

5.2 Decentralized Monitoring Tools

In this section we present a variety of novel, decentralized monitoring schemes used in the AZDBLAB research infrastructure.

5.2.1 Observer

Observer is an integrated GUI for conducting experiments involving a large number of queries or transactions and analyzing the run results. The GUI has been evolving; it is implemented with Java Swing API [52]. This GUI is reusable in other projects that follow an equivalent experimental framework to that of AZDBLAB. For instance, the GUI has been successfully ported to the automated causal analysis project, which also runs experiments related to ergalic theories [68].

Figure 5.2 shows the main GUI that a user can see. The Observer GUI has a tree-structured form; it consisting of left-hand and right-hand tree nodes.

A left-hand tree node is associated with data corresponding to experiment provenance. For instance, '`Users`' node contains users in a lab shelf selected by the user. In the figure, under the '`Users`' node, we can see a lab shelf user (node), named '`yksuh`', under which there exists a notebook node, named '`VLDB2014`', which represents a lab notebook on the lab shelf. We have

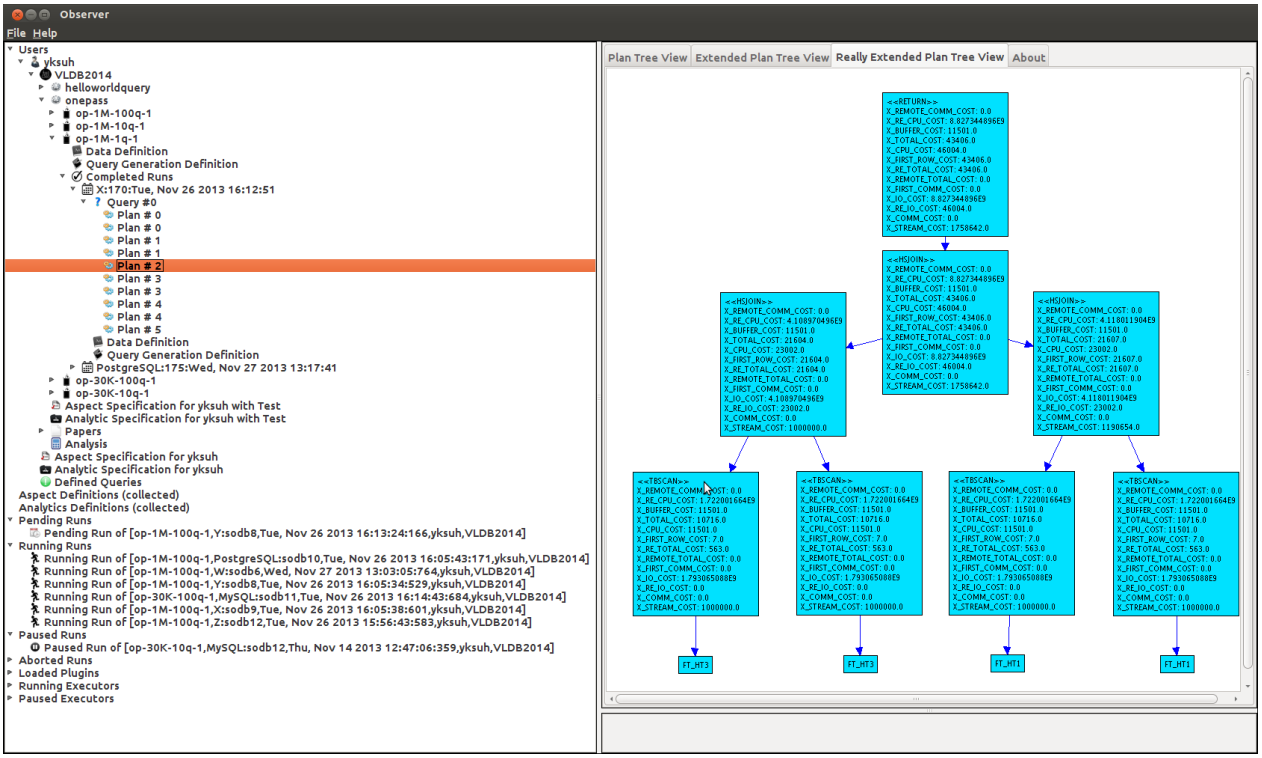


Figure 5.2: AZDBLAB Observer

helped much improve the left-hand tree nodes, by 1) restructuring some nodes such as “Runs” and “Executors”, 2) resolving ambiguous node labeling, and 3) helping other developers to add a new node and revise an existing node.

In the right-hand side, more detailed information on a chosen left-hand node is provided in tabbed views. For the selected plan node (‘`Plan # 2`’) to be discussed shortly, the corresponding, expanded plan tree is exhibited with various operator-wise cost estimates, as shown in Figure 5.2. We have developed showing three different levels of the plan tree operator estimates in the right-hand side tabs.

This GUI allows the user to proceed with running an experiment. The experiment is conducted along with a *scenario*. Figure 5.2 shows several experiments such as ‘`op-1M-1q-1`’ under the ‘`onepass`’ scenario node.

A scenario specifies what an experiment is concerned about and what specific steps should be executed. The user is responsible for writing code for the scenario and placing the code in a sub-directory, named `plugins`, in the directory of an AZDBLAB executable. The user also needs to write and load into AZDBLAB a simple XML experiment specification, which contains the

scenario name, experiment configuration parameters, table schema and population direction, and query or batchset generation. We have written most scenarios that have been run in AZDBLAB. We also have participated in managing the XML schema for the experiment specification. More detail regarding the scenario and the XML specification is given in Section 5.3.

The user can then schedule an experiment on a specific DBMS. This scheduled experiment instance is termed as 'experiment run' and its status is in 'pending'. Observer updates its GUI to show the pending run. If an executor, to be discussed in Section 5.4, has any assigned pending run, the executor starts to execute that pending run, whose status then is updated to 'running'. As the run proceeds, the user can monitor the run's status through this GUI. Note that the user can also pause and resume the running run if necessary. Once the run is completed, the user can check the experiment results of the run. In Figure 5.2, two completed onepass runs of DBMS X and PostgreSQL DBMS are illustrated. The completed run on DBMS X shows that its executor detected six change points while running query #0 over varying cardinalities of `ft_HT1` table on DBMS X. Among the observed plans of query #0, the Plan #2's tree is shown in Figure 5.2.

The Observer GUI was mainly designed and developed by Rui Zhang and later revised by other students including us. Our biggest contribution on the GUI was to complete the plan tree tabs with plan operator estimates and to implement a seamless workflow of scheduling an experiment and monitoring the status (e.g., pending, running, paused, unpaused, and completed) of the experiment run.

5.2.2 Web Apps

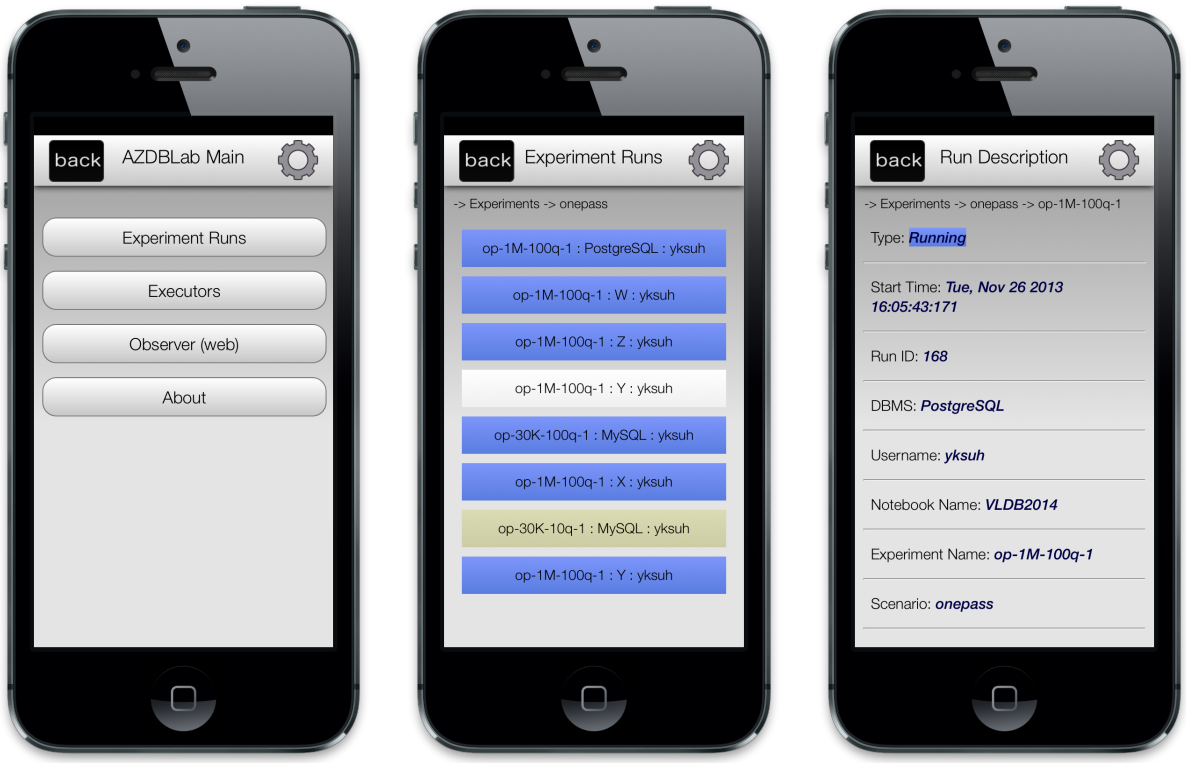
AZDBLAB also provides web apps using Ajax [88]. The AZDBLAB web app is available at `https://aw.cs.arizona.edu/AZDBLAB`. A user can access the main page of the web app via that URL and make a request through a web browser. The AjaxManager, a Java class, then takes and converts to the corresponding Java API the user's request. The AjaxManager in turn via the API interacts with the lab shelf server, pulls the requested data from a chosen lab shelf, and sends the data to the user. The web app provides a similar GUI and functionality as Observer, without requiring direct access to the lab shelf DBMS server, thereby achieving greater security.

The web apps were initially designed and developed by undergraduate students Jordan Martin and Dave Gallup. Later, the apps were completed by undergraduate student Haziel Zuniga. We assisted Haziel Zuniga to implement the web app functionalities.

5.2.3 Mobile Apps

To more flexibly monitor ongoing runs, mobile apps are also available for a user. The mobile apps are relatively small and have simple functionality, compared to that of Observer. We have built both Android and iOS apps. These mobile apps communicate with an AZDBLAB web server via an API written in JSP and Java, which in turn calls methods from AjaxManager to retrieve information about user logins, experiment and executor statuses. Through these methods, the mobile apps allow a user to select a lab shelf and view currently pending, running, and paused experiments as well as executors being used in that lab shelf. Communication happens with the server via `GET` and `PUT` `http` requests.

Figure 5.3 illustrates the AZDBLAB mobile (iOS) app. When a user starts the app, the user is presented with a login screen, into which the user's credentials must be provided. The credentials are validated through the JSP API. User logins are stored on the server in XML files (one per user login) that contain the user name, password hash, and superuser status. Once the credentials are validated, the user can see the main screen consisting of four menus, as shown in Figure 5.3(a).



(a) Main (b) Experiment Runs (c) A Running Run

Figure 5.3: AZDBLAB Mobile Apps

'`Experiment Runs`' lists all the runs in a chosen lab shelf, as illustrated in Figure 5.3(b). (The runs are the ones as shown in Figure 5.2.) An item in blue indicates a currently running run, one in white a pending run, and one in gray a paused run. To get a full list of information about an experiment, he can choose one from the list, and a full detail of the experiment will be brought up as a new page, as illustrated in Figure 5.3(c). A running run on PostreSQL is exhibited in Figure 5.3(c). This browsing and viewing process is essentially the same for viewing executors. '`Executors`' provides currently running executors, '`Observer`' shows the same view as the web app's one, and the '`About`' provides a description of AZDBLAB system.

To allow this application to run on multiple mobile OSes with minimal effort, we built it using the PhoneGap development library [57]. Using PhoneGap, we were able to write the majority of the software using platform-agnostic web standards such as HTML and CSS for the UI and Javascript and JQuery for the networking and other general logic. We have tested the app on both iOS and Android. We plan on testing the app on the Windows Phone in the future.

The mobile apps were designed and developed by undergraduate student Benjamin Dicken and later completed by undergraduate student Haziel Zuniga. We provided mentoring for both students to implement and enhance the mobile apps.

5.2.4 Watcher

Watcher is a utility program used for watching an executor's behaviors during an experiment. The program frees a user from keeping an eye at all times on how the experiment goes. Watcher immediately sends a user via email the notification messages when we encounter the following situations where 1) an executor gets stopped, 2) a DBMS unexpectedly shuts down, 3) a running experiment gets paused by an exception, and 4) the lab shelf server is not responsive. The user then can go to the terminal of the machine and then resolve the reported problem related to the notification.

This program was developed by doctoral student Jennifer Dempsey. We assisted her to design, implement, and test the Watcher program.

5.3 Plugins

AZDBLAB contains several plugin components, which are separated from the abstract or generic ones. The development philosophy of AZDBLAB is to move up all common classes and methods

into parent abstract classes, and instead to specifically implement different or concrete classes and methods through plugins, which can be dynamically integrated into AZDBLAB depending on what experiments are conducted.

There are several classes of plugins: lab shelf connector, experiment subject, evaluation, protocol, and scenario. Each class is deployed as an XML or a jar file. In this section, we discuss each of these plugin classes.

Lab Shelf Connector: A lab shelf connector is an XML file, which contains the profile of a lab shelf, which should be authenticated by the central lab shelf server. The profile information is comprised of a username, a password, and a connect string for the lab shelf server. The profile is encrypted when created, and it is later decrypted when a user selects a specific lab shelf. The connector's creation/selection is done through Observer. We have written and managed most of the lab shelf connector plugin XML files.

Evaluation: The evaluation plugin is a Java class, and it is deployed as a jar file. This plugin evaluates completed experiment runs by producing further information used in performing an in-depth analysis. For instance, if an experiment involves query execution, an evaluation plugin parses a rich source of information associated with the query execution. The objects of the information can be overall Linux measures like *utick* (user time) or *stick* (system time), processes involving query execution, and the processes' specific Linux measures like utick, stick, major faults, or block IO delay (IO delay time) [11]. The evaluation plugin installs additional tables, parses these information, and populates the installed tables with the extracted values in AZDBLAB. These values are used by a protocol plugin, to be described in the next section.

Undergraduate student Matthew Johnson most contributed to developing the evaluation plugin code. We have extended his code to include further changes for extracting the measures from experiment runs and storing the values into a lab shelf.

Protocol: Protocol plugin is a Java class, and it is also deployed as a jar file. This plugin is designed to run a query execution time measurement protocol on the completed experiment runs [11]. While running the measurement protocol, the plugin installs and populates with the extracted values the tables or views for calculating query time at a cardinality.

This code was originally designed by doctoral student Andrey Kvochko, further revised by us including undergraduate student Derek Merrick, and later re-factored by Haziel Zuniga.

Experiment Subject: The experiment subject plugin is a Java class, and it is deployed as a jar file as well. The plugin's common methods are provided in an abstract class in the core package of AZDBLAB. We implement in a subclass the methods specific to each DBMS subject and deploy the subclass as a jar file. When an executor, to be described in Section 5.4, begins to run an experiment, the designated DBMS experiment subject plugin is dynamically loaded and used for the experiment.

Rui Zhang framed each initial experiment subject plugin, and then we completed it.

Scenario: To conduct an experiment, as discussed in Section 5.2.1, a user needs to write and load into AZDBLAB the experiment's specification in XML. The specification contains the following information: experiment name, table schema definition, table population direction, and task-specific information such as query generation details or predefined queries' location (for query-related experiments). In addition, the specification should also include the name of a *scenario* in which the experiment is conducted.

Note that for our thrashing study, we have extended the specification to include 1) several environment parameters such as the number of processors and DBMS buffer space and 2) batchset generation directions.

A scenario plugin, a Java class, specifies a series of steps used in an experiment. A user is responsible for designing and developing the scenario, which should be implemented along with a three-tiered scenario hierarchy. For example, the user can write concrete scenario source Java code, called '`OnePassScenario`', to observe the query suboptimality phenomenon mentioned in Section 2.4. As a reminder, the query suboptimality indicates that when the execution plans of a query change between two adjacent cardinalities (called a *change point*), the actual elapsed time of that query at a lower cardinality is greater than that of a higher cardinality. ('`OnePass`' is named after the technical decision of decreasing the cardinality only in one direction from the maximum to the minimum.)

Rui Zhang designed and implemented several initial scenarios for observing query suboptimality discussed in Section 2.4. We took over his scenario code and completely modified it,

resulting in the `OnePass` scenario plugin. We also wrote from scratch several different scenario plugins including the `Exhaustive` scenario plugin used to check the monotonicity assumption of query time [9]. In particular, for our study we have designed and written a concrete scenario source code, called "`DBMSThrashingScenario`." Further details about the scenario implementation are provided in Section A.3.

Note that while running the thrashing scenario we first collect data after finishing a batch run and then store the measured data into the lab shelf server for further analysis. Thus, the communication to the lab shelf server does not disturb our measurement.

The next section discusses an executor, which is another important component in AZDBLAB.

5.4 Executor

An executor is a stand-alone Java application as observer. It conducts an experiment on a DBMS, co-located with the executor, as illustrated in Figure 5.1. The executor utilizes a DBMS experiment subject plugin.

A user can launch an executor and schedule a pending run to the executor. The executor can begin the experiment by loading the run's scenario and experiment subject plugins, as described in the prior sections. The executor then creates and populates tables, executes transactions (or queries) on the tables, records transactions' (or queries') run results into AZDBLAB, and finishes the run, as explained in Section 5.2.1. If any exception occurs during the experiment, the executor can pause the run. Later, the user can resume the run via observer. To avoid undesirable latency by network traffic, we ensure that the executor must run on the same machine running a DBMS. Currently, executors can be run on all the seven relational DBMSes. We plan to support more relational DBMSes.

Rui Zhang developed an initial executor, and we revised and completed the executor working with the experiment subject plugins. Jennifer Dempsey integrated her watcher code into the executor.

In the following chapter we discuss our efforts to measure DBMS thrashing.

CHAPTER 6

THRASHING METROLOGY

Metrology is the science of weights and measures or of measurement [36]. In this dissertation we use a novel, scientific methodology, termed "thrashing metrology," for measuring DBMS thrashing. In this chapter, we discuss the overall thrashing metrology including environment setups, experiment running, and the data analysis protocol.

6.1 Experiment Setups

In this section we elaborate the environment settings for running experiments.

Cold Cache: As with our prior work [9, 11, 71], cold cache measurement is used in our thrashing metrology. We first flush the hard disk cache by reading a huge dummy file. We then flush the Linux operating system (OS) cache by executing two consecutive commands: "`sudo /usr/local/sbin/setdropcaches 1; sync`." We do not flush the cache of a different OS on which one of our DBMSes runs, as there's no known way of flushing. We subsequently flush the buffer cache of each DBMS experiment subject by executing its respective flushing command. This cold cache scheme gets executed every time before running each transaction of the clients in a given batch.

The Thrashing Observance Scenario: As described in Section 5.3, we have implemented the thrashing observance scenario as a plugin written in Java, along with the three-tiered scenario development hierarchy. The detail of the scenario is presented in Section A.3.

Experiment Specifications: We described each independent variable's operationalization in Section 3.4. To realize the operationalization, we wrote an experiment specification in XML, including the aforementioned thrashing scenario name and the independent variables' specific parameters. We then loaded the specification into the Observer GUI.

DBMS Subjects: As mentioned before, we have employed a total of the five relational DBMS subjects—three proprietary and two open-source DBMSes—for running the thrashing scenario in our experiment. Each of the DBMS subjects is JDBC-compliant and easily plugged into our AZDBLAB infrastructure, where it has been successfully running experiments.

Based on this experiment setup, we have conducted large-scale experiments to observe thrashing across the DBMSes. In the subsequent section, we elaborate the running of these experiments.

6.2 Running an Experiment

To conduct a thrashing experiment, we choose an experiment node associated with the loaded experiment specification and simultaneously launch from the node experiment runs on the five DBMS subjects via the Observer GUI. In AZDBLAB, while an existing run is running on a chosen DBMS subject, we can make a new run pending on the same DBMS subject through the GUI, so that the new run can get automatically started when the existing run gets finished. When a run gets paused by an exception (or a user pauses the run for some reasons), we can resume the run through the Observer GUI. In this way, we have managed these runs over about a cumulative year.

In our experiment, each run contains (4 (read SF values) + 3 (update SF values)) $\times$ 4 (ROSE values) = 28 batchsets. Given a batchset in a run, we measure the elapsed times of transactions executed in each batch in that batchset and compute the TPS value of that batch. We record into our lab shelf DBMS server the measured data as well as the associated batchset parameters along with the batchset schema presented in Section A.2. The measured data are used for the exploratory and confirmatory evaluation of our model, to be discussed in Chapters 8 and 10, respectively. As a result, we have a total of 90 completed runs, consisting of 4 (numProcs values) $\times$ 5 (the number of DBMSes) $\times$ 2 (with and without PK) = 40 runs for the exploratory evaluation and 5 (numProcs values) $\times$ 5 (the number of DBMSes) $\times$ 2 (with and without PK) = 50 runs for the confirmatory evaluation. The descriptive statistics of the exploratory and confirmatory evaluation data will be presented in Sections 8.2 and 10.2, respectively.

6.3 Thrashing Analysis Protocol (TAP)

Before proceeding with the evaluation of the model using the measured data, we had one question: How can we make sure that the data are clean enough to proceed with the evaluation of the model?

Is there a way of assessing the validity of the acquired data?

To address this concern, we have designed a sophisticated thrashing analysis protocol, called TAP, which performs a suite of sanity checks on the measured data, drops data failing to pass the checks, and then computes a thrashing point for each batchset in the retained data.

Terminology: We defined the terms of batch and batchset in Section 3.3. In the TAP description, we use three terms related to actually executing a designed batchset and its batch in our experiment. A batchset "run" (BSR) indicates that a group of different (28) batchsets is scheduled and run in its entirety on a specific DBMS. If the same group of batchsets gets scheduled to run on two different DBMSes, we then have two runs of the group, or two BSRs. A batchset "instance" (BSI) refers to a batchset scheduled and executed on a specific DBMS; that is, it represents an individual batchset included in a BSR.

A batch "instance" (BI) is a batch scheduled and executed on a specific DBMS; that is, a BI refers to a batch included in a BSI. If we execute a batch—run the transactions of the clients in that batch—in a given batchset scheduled on two different DBMSes, then we have two BIs each from two BSIs. A "batch instance execution" (BE) indicates a repetition of a given BI on a chosen DBMS. If we repeat executing the same BI on the DBMS three times, we then have three BEs for that batch.

6.3.1 Step 0: Run Batchset Experiments on DBMSes

TAP begins by scheduling a batchset on a DBMS and executing the batchset, or making a BSR. Before making the BSR, we need to configure an experimental machine on running a chosen DBMS subject, by 1) disabling as many unnecessary daemons as possible, 2) turning off Turbo mode, 3) turning on the NTP daemon, 4) turning off all other DBMS subjects, 5) setting the number of processors specified in the experiment, and 6) checking if the central lab shelf server is responsive, as mentioned in our timing work [9, 71].

6.3.2 Step 1: Perform Sanity Checks

TAP uses a sequence of nine sanity checks partitioned into three classes. The first class is the experiment-wide cases for which not a single violation should occur in a run, consisting of five sanity checks. (1) The *number of missing batches* indicates how many BIs were not executed in a BSI, for whatever reason. This case should not happen. (2) The *inconsistent processor*

configuration violation occurs when the number of processors specified in a given experiment specification is inconsistent with the current processor configuration on an experiment machine. We can figure out how many processors are currently enabled by examining the output from `more /proc/cpuinfo` on Linux. For DBMS Z running on the other OS, we tell our lab staff to manually set the specified number of processors through BIOS, as mentioned in Section 3.4.1. (3) The *number of missed BE violations* represents how many BIs did not have the specified number of repetitions (typically, three or five) required. If any BI was executed with fewer than the number of the designated repetitions, we catch the BI as a violation. (4) The *number of other executor violations* indicates how many BSIs were run together with another executor running on a different DBMS subject. We enforce that only one executor runs a BSI at a time. When an executor gets launched to run a BSI, if another executor is already running its batchset, then the launched executor must terminate immediately. (5) The *other DBMS process violation* identifies how many BSIs were run together with other DBMS processes. We enforce that when a BSI gets run on a chosen DBMS, all the other DBMSes' processes should be deactivated, except those of the chosen DBMS.

The second class of sanity checks concerns BEs. Each of these three sanity checks could encounter only a few isolated violations. However, we expect the violation percentage to be low. (6) The *zero TPS violation* catches BEs where TPS was zero at the starting MPL in their BSIs. We don't expect thrashing to occur at the minimum MPL. If too many violations are found, then we assume that the experimental machine or the corresponding DBMS subject was unreliable temporarily, due to some unknown reasons. It thus is hard to trust the rest of data in the same batchset. (7) The *connection time limit violation* identifies the BEs that violated the specified CTL. In this violation, we count the number of BEs that ran over or terminated earlier than the given CTL. (8) The *abnormal ATPT violation* check indicates how many BEs revealed abnormally high or low ATPT. The high ATPT could occur in the situations, when a) many unexpected daemon processes suddenly appeared at the BEs, or b) the DBMS subject suffered from resolving deadlocks among too many transactions. The low ATPT could happen, because most clients in the same batch could temporarily disconnect from the DBMS subject due to blocking, so the actual workload was too low to that DBMS subject. The outliers are determined as the ones above and below the average plus and minus two standard deviations of the ATPT, computed on the BEs associated with the same BI, respectively. We also expect this number to be very low.

The final class involves one check over a BSI. (9) The *transient thrashing violation* check examines how many BSIs experienced transient thrashing, which indicates that the TPS measured at a determined thrashing point is lower than any of the TPSes measured at the MPLs bigger than the thrashing point. Once the thrashing phase begins, it will be rare to observe that the throughput rises back up at greater MPLs.

6.3.3 Step 2: Drop Batch Executions

In Step 2, TAP drops BEs that exhibit specific problems throughout Step 1. We take the union of BEs caught by the second class of sanity checks ((6), (7), and (8)), and remove the BEs from the original data.

6.3.4 Step 3: Drop Batchset Instances

In Step 3, we look at the retained BEs for each BSI, and then determine if these BEs in concert exhibit specific problems. If so, we drop the entire BSI (having its own BEs). Step 3 consists of the three sub-steps. Step 3-(i) drops BSIs that do not have all the MPLs in the corresponding BSIs. Recall that each batchset has ten batches. If there's any BI dropped in the same BSI, we drop that BSI. Step 3-(ii) drops raw BSIs revealing transient thrashing, as specified in (9). Step 3-(iii) drops retained BSIs showing transient thrashing. These steps enhance the accuracy of thrashing point calculation in the next step.

6.3.5 Step 4: Compute the Thrashing Point

At this step we calculate the thrashing point for each of the retained BSIs, along with Equation 3.2 on page 55. After finishing Step 4, we proceed with conducting evaluation on the model with the measured data: the retained BSIs and their corresponding thrashing points.

CHAPTER 7

A REVISED STRUCTURAL CAUSAL MODEL OF THRASHING

After we initiated exploratory experiment runs to collect data, we encountered several technical challenges. While addressing the challenges we had to make changes to our experiment scenario code. Based on the ergalics methodology discussed in Section 2.1, we had to revise the proposed model by removing or adding variables even before conducting exploratory evaluation of these runs on the model. In this chapter we discuss these challenges and how we resolved them. After that, we present the revised model reflecting the resolved challenges.

7.1 Technical Challenges

In this section, we elaborate the technical issues and present our approach on each of the issues.

7.1.1 Batchset Run Time

As in a previous paper [37], we initially used two minutes for CTL, plus one minute for think (stoppage) time. (A total of three minutes were used for each batch run.) Considering we have ten batches in each batchset, a single batchset run was estimated to complete in about 90 minutes. However, the 90-minute batchset run time estimation did not take into consideration several non-trivial challenges. After analyzing the output logs written while conducting our experiments, we were able to identify the challenges, including

(i) duplicated table population for batchsets included in an experiment run,

(ii) substantial access latency to a separated lab shelf from DBMS W (as described in Section 1.1.1) running in a virtual machine,

(iii) slow disk drive flushing time (see Section 6.1) for DBMS W running in a virtual machine,

(iv) long wait time for a client to disconnect from an experiment subject after executing a batch,

(v) runs paused by exceptions that occurred because of runtime errors during experiments, and

(vi) idle time until resuming the paused runs.

Due to these unfortunate latencies, the actual batchset run took up to about 12 hours.

Now let's briefly discuss what the problems were and how to tacked them. Regarding (i), we found that for each batchset, tables accessed by a batch were unnecessarily populated over and over again. This repeated table population per batchset incurred a significant loading-time latency, ranging from five minutes up to an hour. We wasted a day only because of table loading per batchset included in an experiment run. (Recall that in a single experiment run we had many (specifically, 28) batchsets as mentioned in Section 6.2). To save the duplicate loading time, we applied the "table copy approach." At the beginning of a run (that gets initiated or unpaused), we populate the original tables and then copy them to the corresponding temporary tables. After a batchset is studied, we clear (drop) the original tables. For the subsequent batchsets in the same run, we copy rows back into the original tables from the cloned temporary tables, instead of repopulating them. The copy typically took half of the existing table population time. Therefore, the table ready time was sped up by a factor of two per batchset.

To resolve (ii) and (iii), we have decided not to use DBMS W running inside a virtual machine, as mentioned in Section 1.1.1. One reason is consistency. Other than DBMS W, the other DBMSes do not need to run in a virtual machine. Moreover, even if the data from DBMS W are missing in our dataset, we do not think the model evaluation would be seriously hurt by the exclusion of DBMS W.

Regarding (iv), we use the `while` loop for running the transactions of the clients in a batch, as described in Section 3.4.2. In the termination condition of the `while` loop, we check to see if the elapsed time surpasses CTL set from an experiment specification. When the specified CTL was reached, some clients in a mid-size or a large batch (e.g., more than an MPL of 500) were still waiting for `Connection.close()` preceded by `Statement.close()`; namely, the clients could not close their connections immediately after the given CTL. This unexpected wait prevented the ongoing experiment from proceeding promptly with the subsequent batch execution. In particular, there was such a significant delay when the clients in the update group were committing for close. (This makes sense, in that transactions would have tried to released their locks at the same "commit" time because of the reached CTL.)

Furthermore, we saw that the elapsed-time checking at the `while` loop head sometimes takes place much later than the given CTL for some unknown reason. It seems that when the specified

CTL was reached, the main program control, yielded to the threads (representing the clients in a batch) to run transactions inside a DBMS, failed to immediately fall back from the threads to the `while` head; the control still stayed in the threads much longer than the given CTL. The delayed control flow seems caused by long transactions that were still ongoing beyond CTL. Specifically, some clients running the long transactions couldn't close their connections immediately after CTL. (Note that the last transaction finished after CTL per client will not be counted, and its elapsed time will not be considered in the ATPT calculation, either.) We tried to interrupt the threads by invoking `Thread.sleep`() every a second, which did not work well.

To get around this problem, we used two alternative ways applied to each DBMS. One way was to set the same timeout in seconds as the given CTL to a `Statement` object before starting to run a batch. The corresponding method concerns `setQueryTimeout()` [54] in `Statement`. This method allows us to set the number of seconds for which the JDBC driver will wait for the `Statement` object while running a transaction. Once the timeout is reached, an `SQLTimeoutException` can be thrown from `setQueryTimeout()`. After that, we could successfully terminate the execution of the batch without or with a minimum delay. Note that previously the average wait delay after one batch execution was about three minutes, and the worst delay time reached up to about an hour on a certain DBMS. After invoking `setQueryTimeout()`, the wait delay of each batch execution was managed from zero msec to one minute—within an average of 30 secs—depending how big a batch is.

However, the timeout approach was not effective in DBMS Y. For DBMS Y we separately invoked a client-wise thread for "aborting" transactions after the specified CTL. In that thread, we call `Connection.commit`() followed by `Statement.cancel`(). This abort thread finally worked on DBMS Y.

Also, we removed many trivial client-wise console output (for debugging purpose) written to an external log file. We realized that writing logs incurred some latency too. Consequently, the total wait time per experiment run was considerably decreased, by a factor of six.

Concerning (v), one type of runtime error was to reach the maximum number of connections configured in a DBMS. There sometimes happened the lack of available connections when a batch run started because of connections not immediately reclaimed after a previous batch run in spite of sufficient think time. To handle this error, we increased the default number of maximum connections to 20% or more. Another error associated with a particular DBMS was due to the lack

of transactional log file size causing a dynamic exception. To resolve this error, we had the log file size automatically increase whenever reaching the maximum file size.

With respect to (vi), we had a number of runs running simultaneously across DBMSes. Several runs were paused at times by exceptions caused by dynamic errors. To reduce the idle time, we first tried to prevent runs from getting paused by fixing all known errors, and we monitored the runs more frequently than before, so that any paused run can be unpaused as soon as we see it. These optimizations in concert sped up our experiments by a factor of seven.

However, our plan for the CTL variable operationalization was still too ambitious. We planned to increase the values of CTL with two, four, and eight minutes. Considering running the transactions of the clients in a batch repeated multiple times, such operationalization represented a significant burden: adding months of experiment time. Furthermore, we realized that no matter how long CTL was set, there was little influence on the dependent variables of ATPT and thrashing point. Suppose that we increased CTL to four minutes. Although some long transactions may get to be completed thanks to the increased CTL, we cannot predict whether ATPT and the thrashing point will increase or decrease, as many short transactions will also be completed within the increased CTL. In this sense, the size of CTL did not matter to us. We thus had to remove the CTL variable and its relevant s from the proposed model while still fixing CTL to two minutes, considering that the ATPT value of the data from the early exploratory experiments was around only 13.17 secs, small enough to measure the reasonable values for the dependent variables.

To compensate for the removed CTL variable, in the confirmatory experiment (to be discussed in Chapter 10) we increased the number of repetitions for running the transactions of the clients in a batch to five from three used in the exploratory experiment; namely, we collected more samples for the confirmatory evaluation of our model. Increasing the number of the samples brings the following two advantages. First, the accuracy of the measured batch execution data can be increased, and the repeatability of measurement can be reinforced. Second, we can retain more samples than otherwise, in case that the dirty samples detected through TAP should be thrown out.

7.1.2 Buffer Space Variable

Another technical challenge was to operationalize the buffer space variable in the model. Each DBMS has its own way of adjusting its buffer cache size. For instance, MySQL uses a single "value" (in bytes) to be set to `innodb_buffer_pool_size`, as described in Section 3.4.1.

PostgreSQL uses an "interval" of shared memory (buffer) segment size using `SHMMIN` and `SHMMAX`, plus `SHMALL` representing the total amount of available memory (in bytes or in pages) [77]. PostgreSQL can dynamically adjust the buffer size along with system circumstance.

The proprietary DBMSes also have their own ways of adjusting buffer space size. DBMSes X and Y used a single value like MySQL. DBMSes didn't guarantee that the specified buffer space size remained fixed. The DBMSes could dynamically change based on the specified (minimum) size when treating its workloads as done in PostgreSQL. DBMSes W and Z used a certain "percentage," with which the DBMSes could adjust their buffer pool size along with how much memory was free in their underlying systems.

Moreover, for some DBMSes there were additional constraints with respect to the configuration of their buffer cache size. For example, PostgreSQL had a constraint such that an allocation of more than 40% of RAM (random access memory) to its buffer pool might not work better than a smaller amount. MySQL reserved 10% more memory for managing buffers and control structures.

Such a different configuration prevented us from making the consistent operationalization to the buffer space variable. Hence, we also had to remove the variable from the model. Instead, we examined another way of reflecting the effect of I/O on DBMS thrashing, to be discussed next.

7.1.3 Physical Memory Variable

To address the I/O effect, we thought that another variable could be introduced into the model. The variable was "physical memory." By directly adjusting the amount of physical memory, we thought that it was possible to see the impact of I/O on DBMS thrashing. If the given amount of physical memory isn't enough to handle transactions, page swapping between RAM and disk will frequently occur. Then the increased I/O latency will delay the transaction processing, thereby evincing DBMS thrashing. If the memory is sufficient, on the contrary, thrashing may not occur as the bottleneck by I/O gets eliminated. In this way we can perhaps examine the relationship between physical memory and DBMS thrashing.

However, operationalizing physical memory was a daunting task. We tried to figure out how to control the amount of physical memory in BIOS or through operating system. Adjusting the physical memory size was beyond our knowledge. We might remove some of RAM cards from the slots on the main board in our experiment machine, which was not feasible in reality. We thus searched for another alternative of capturing the I/O effect on thrashing, to be discussed next.

7.1.4 Index Scheme Variable

Without an index on a table, a DBMS has to do full scans when treating transactions. However, the DBMS can speed up transaction processing time by taking advantage of the table's index to avoid the full scans.

We could consider several indexing schemes, such as B-tree, hash, bitmap indexes, etc., for our thrashing metrology. One challenge was to determine which common indexing scheme should be chosen and applied to any DBMS to preserve the consistency of operationalization. Although every DBMS supported indexing, not all DBMSes allowed users to specify a specific indexing scheme.

Utilizing "primary keys" was one option to see the I/O effect on thrashing. When creating a table with primary key declared, we realized that DBMSes automatically create the primary index of the table. If we could make transactions refer to a primary key column(s), we expected that the primary index would be exploited when processing the transactions. That way, we thought we could capture the effect of I/O on thrashing via the primary index.

Another advantage of utilizing primary keys came from its easy operationalization: we could simply declare them when populating a table. It was enough to have a simple flag for representing the primary key presence of a column. Therefore, we decided to include the presence of primary keys as a new variable in the model that we revised to address these technical challenges.

7.2 The Revised Model of Thrashing

In this section we introduce a new construct including the primary key variable discussed in Section 7.1.4. We in turn present the revised model, consisting of the new and retained constructs.

7.2.1 The Schema Complexity Construct

The *schema complexity* construct concerns relational schema that may affect DBMS thrashing. In this construct we include one variable: "Presence of Primary Keys (PK)." This PK variable is identified as a factor whose presence in tables will reduce transaction processing time, thereby increasing the thrashing point.

Presence of Primary Keys: This is a new independent variable capturing the influence of I/O on DBMS thrashing. As mentioned in Section 7.1.4, DBMSes can save substantial I/O in the presence

of PK when treating transactions referencing PK columns in a table, by avoiding full scans through the primary index of the table. This reduced I/O can lead to the increase of transaction throughput, and thus, DBMS thrashing may occur later. We think that PK is correlated with the thrashing point.

7.2.2 The Revised Model and Correlations

Figure 7.1 exhibits the revised model. Compared to the proposed model shown in Figure 4.1 on page 57, the revised model does not have the variables of buffer space and CTL and their associations. Specifically, the two edges from buffer space to ATPT and the thrashing point were removed, and deleted were also the edges from CTL to the thrashing point.

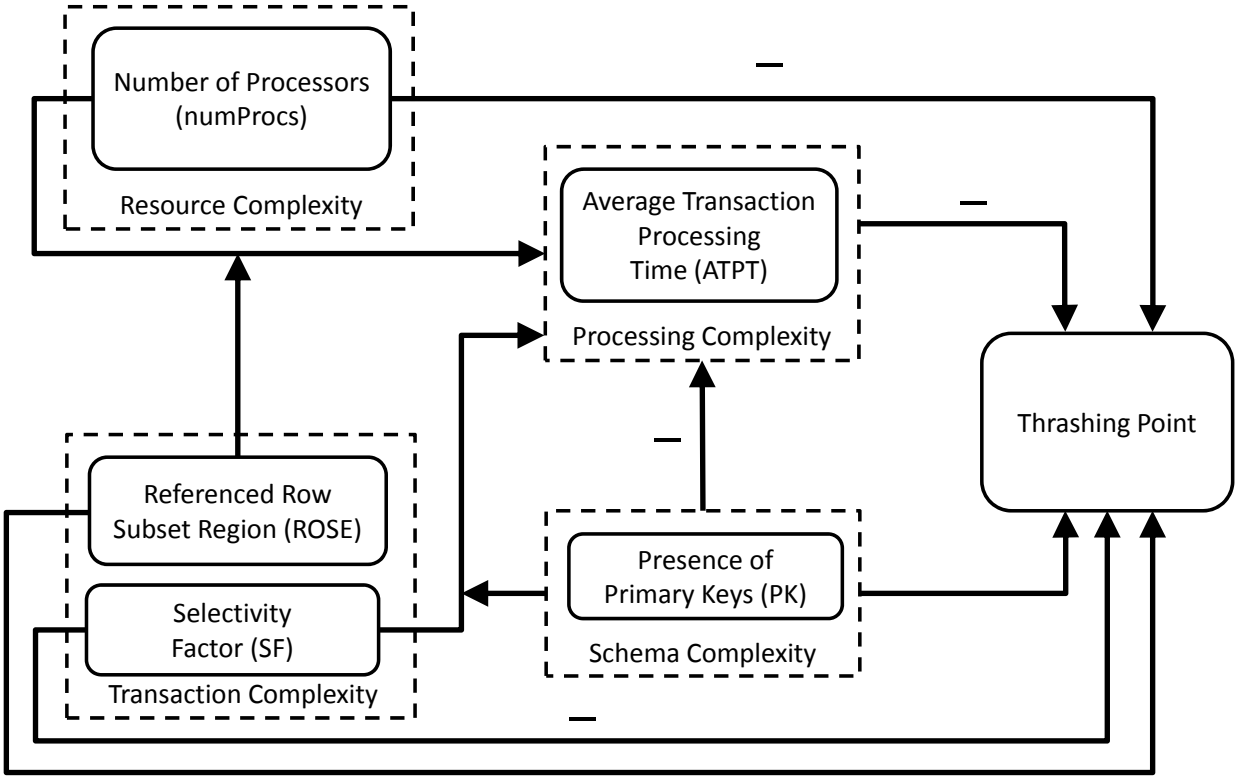


Figure 7.1: A Revised Model of Thrashing

Instead, we now have the new variable of PK. Accordingly, we have added three arrows in the model. One arrow concerns ATPT. Another arrow concerns the thrashing point. The third arrow concerns the association between the two constructs of transaction complexity and processing complexity. All other variables and their associations remain the same as before.

Because of the removed buffer space and CTL variables in Figure 7.1, the associated Hypotheses 3, 4, and 10 (as described on pages 62-63) have been removed. The remaining

hypotheses are unchanged; namely, a total of seven hypotheses are still valid.

Regarding the presence of PK variable, we hypothesize that in the presence of PK the thrashing point will increase. When PK exists, a DBMS can speed up processing transactions accessing the PK columns. As a result, the overall transaction throughput will increase, leading to an increase of the thrashing point as well.

Hypothesis 8: In the presence of primary keys the thrashing point will increase.

Also, we hypothesize that ATPT will decrease if PK exists. Primary keys allow DBMSes to avoid doing full scans on the tables referenced by transactions. Considering that I/O is one of the biggest components in transaction processing, The DBMSes can significantly benefit from a primary index automatically created when the primary keys are declared. The I/O saved through a primary index will suppress a significant rise in processing delay incurred when transactions simultaneously read or update rows. Furthermore, for a given SF, ATPT will be reduced in the presence of PK. Specifically, the slope between SF and ATPT will go lower when PK is present. Therefore, we hypothesize the following.

Hypothesis 9: In the presence of primary keys, ATPT will decrease.

Hypotheses 10: The presence of primary keys will moderate the relationship between SF in the transaction complexity and ATPT in the processing complexity.

The revised model in Figure 7.1 has made several changes to the existing correlations shown in Figure 4.1. Table 7.1 exhibits the hypothesized correlations and hypothesized levels based on the revised model. Note that because of the deleted variables, the related correlations—I-(c) and I-(d)—are left empty. Although omitted in Table 7.1, groups II and IV remain the same as shown in Table 4.1.

Group III is replaced by the new correlations introduced by the presence of PK. III-(a) indicates the relationship of the presence of PK with ATPT. We expect the level of this relationship to be high. As addressed before, the utilization of a primary index created by PK allows a DBMS to avoid full scans when the DBMS reads the rows referenced by transactions. The DBMS can then substantially reduce I/O time in the presence of PK. The reduced I/O time will significantly contribute to decreasing ATPT. III-(b) represents the correlation between PK and the thrashing point. In the presence of PK, the DBMS can increase the overall throughput because of the saved

	a	*b*	*c*	*d*
I	numProcs / ATPT *low*	numProcs / Thrashing Pt. *high*	N/A	N/A
II	SF / ATPT *high*	SF / Thrashing Pt. *high*		ROSE/ Thrashing Pt. *medium*
III	PK / ATPT *high*	PK / Thrashing Pt. *high*		
IV	ATPT / Thrashing Pt. *medium*			

Table 7.1: Revised Hypothesized Correlations and Levels

I/O time. Thus, the thrashing point can be significantly increased. Therefore, we expect the level of the correlation to be high.

In the following chapter we discuss the exploratory evaluation of the revised model.

CHAPTER 8

EXPLORATORY EVALUATION OF THE REVISED MODEL

Many months of programming effort and experimental runs completed over a cumulative year have made it possible for us to evaluate and refine our model. Testing our model is conducted in two steps: exploratory and confirmatory evaluation. In the exploratory evaluation the revised model gets tested and further refined. In the confirmatory evaluation the refined final model, to be presented in the following Chapter 9, gets tested. We conduct the exploratory and confirmatory evaluations separately on the data from a group of read batchsets (hereafter, read group) or a group of update batchsets (hereafter, update group). Depending on what type of transactions are presented to DBMSes, their thrashing behavior will be very different. Hence, it is unreasonable to perform the evaluations on the combined data from the read and update groups.

As mentioned in Section 6.2 on page 76, the exploratory data consist of a total of 28 batchsets generated based on the combination of (4 read SFs + 3 update SFs) $\times$ 4 ROSE values. For each batchset we have a total of 40 runs made based on 4 numProcs values $\times$ 5 DBMSes $\times$ 2 cases with and without PK. These 40 runs took about 4 cumulative months. We describe in greater detail the exploratory data in Section 8.2.

In this chapter, we discuss statistical tools employed for the exploratory evaluation, check assumptions required by the tools, show descriptive statistics about the runs, and lastly present a series of analysis results using the tools. All the statistical test commands and outputs are provided in Appendix A.4.1 for further details. We discuss the confirmatory evaluation of the refined model in Chapter 10.

8.1 Consideration of Multiple Linear Regression

Multiple linear regression (MLR) is a well-known statistical tool for modeling the relationship between a continuous dependent variable and multiple independent (explanatory) variables that are continuous or dichotomous [24]. The revised model shown in Figure 7.1 contains one dependent variable of the thrashing point, whose type is continuous. The independent variables are either

continuous or dichotomous. MLR, thus, is the most appropriate tool to evaluate the model.

(Some might claim that logistic regression [39], a special variant of MLR, can be another tool, since the dependent variable's values are measured as limited and discrete. Logistic regression, however, is not considered in this dissertation, as we see the dependent variable of thrashing point as continuous. But the possibility of logistic regression is left in our future work. It may be possible to do the transformation of the dependent variable to be dichotomous by coding whether or not a given batchset encountered thrashing.)

After fitting the model using MLR, we want to check whether our data obtained from the exploratory experiment can satisfy the assumptions required for MLR. In general it is rare for data to perfectly satisfy all the assumptions, but if a lot of violations are detected then it is not easy to trust the validity of the data fitting for the model using MLR. For the purpose of checking the validity, we test our data and fitted model against the assumptions.

The assumptions of MLR we need to check can be summarized as follows [24, 90].

A Continuous Dependent Variable: The dependent variable in our study concerns the thrashing point. To measure the thrashing point, given a batchset we vary MPL from 100 to 1,000 by steps of 100. If thrashing is detected within the MPL range, then the thrashing point is determined as a multiple of 100. Although we choose representative, discrete values for the thrashing point, the dependent variable is by nature continuous. Hence, we see no violation on this assumption.

Two or More Continuous or Nominal Independent Variables: Our model has a total of five independent variables, among which we have three continuous and one nominal independent variables. The continuous ones are numProcs, ROSE, and SF (although their values are inevitably operationalized as discrete ones in our experiment). The nominal variable is the presence of PK. No violation, therefore, is observed as well.

Sample (Data) Size: According to Tabachnick's and Fidell's book [75], the number of samples should ideally be $50 + 8 \times k$, for testing a full regression model, or $104 + k$ when testing individual independent variables, where k is the number of independent variables. As will be shown in Figure 8.7 on page 98, our data comprise a total of 487 (read: 188 and update: 299) samples, which are sufficiently large. Therefore, this assumption is satisfied.

Independence of Residuals: This assumption implies that there is no auto-correlation in the residuals from a regression analysis. The assumption is also known as "independence of residuals." A residual (or error) is the difference between a predicted value and the observed value. If the residuals are correlated, then regression may underestimate the standard error of coefficients of variables.

The assumption can be tested via the Durbin-Watson (D-W) statistic [15]. The D-W statistic is always between 0 and 4. Conventionally, a value of 2 indicates that no auto-correlation. The D-W statistic is conditioned on sample size.

We used `durbinWatsonTest()` in R [59] to test the assumption against our data. As a result, we obtained the D-W statistics of 0.87 and 0.90 on thrashing point for the read and update groups, respectively. We also obtained the respective D-W statistics of 0.49 and 0.52 on ATPT for the read and update groups. (The *p*-values of these computed D-W statistics were zero.) As these computed D-W statistics were fewer than 2, it seemed that our data had some type of positive correlation among the residuals.

To examine why such auto-correlation was detected in the combined samples from the five DBMSes, we computed the DBMS-specific D-W statistics. We then explored the presence of the violation using each per-DBMS sample size and the D-W table [16]. We found that 1) each sample size per DBMS was low, 2) some of the DBMSes had auto-correlated samples, and 3) there were some clustered values in ATPT and thrashing point in the violating DBMSes. Therefore, testing this assumption was unsuccessful.

The violation might affect the amount of variance explained for ATPT and thrashing point. However, it is not a concern about using MLR for our data. First, the violation is to some extent expected, in that a limited number (e.g, only five) of DBMS subjects were used in our experiment. Second, the overall violation results from only one (or at most two) of the DBMSes that has violating samples. Third, in general the violation does not invalidate the regression models [15]. To succeed this test in the future, we should collect more per-DBMS samples and then check to see if the D-W test can be passed at each DBMS and the overall level.

Homoscedasticity: This assumption tests whether the variance of residuals is constant along the line of the best fit. `ncvTest()` in R can be used for testing this assumption. The *p*-values of the test were 0.26 and 0.67 for the read and update groups, respectively. This means that we cannot

reject the null hypothesis such that the error variance is constant on the fitted values of the model. The assumption is satisfied on our data.

Multicollinearity: This assumption concerns whether two or more independent variables are highly correlated with each other. We tested the assumption, R's `vif()`, yielding variance inflation factors (VIFs) on the data. The outcome of `vif()` demonstrates that none of our independent variables has the square root of its corresponding VIF higher than 2, which means no high correlation of the independent variables. Thus, our data satisfy the assumption.

No Significant Outliers: This assumption implies that there are no influential outliers included in the data. We can test this assumption by using Cooks Distance [8]. If the Cook's Distance of a certain observation (sample or point) is greater than 1, it is classified as influential, which can be considered removed from the data. As shown in Figures 8.1(a) and 8.1(b), our data, tested via `cooks.distance()` in R, does not show significant outliers.

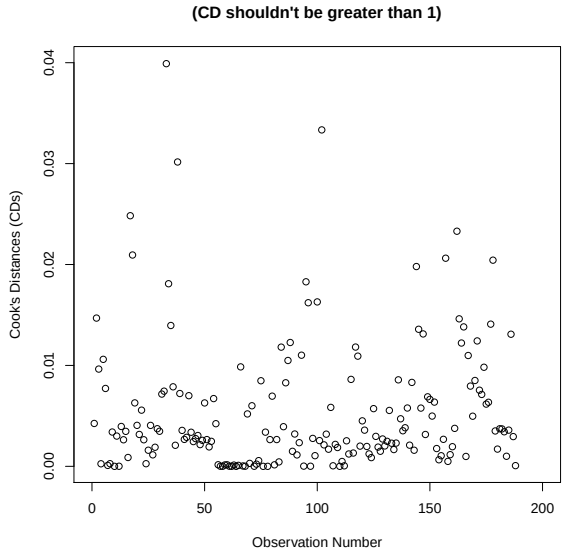
Normality of Residuals: This assumption requires that the residuals (errors) in the data are approximately normally distributed. This assumption is considered as the most important one in MLR [24]. We can test the assumption by looking at the histogram of the residuals to see if a normal curve can be superimposed on the histogram. As depicted in Figures 8.1(c) and 8.1(d), the residuals of our data appear to approximately follow normal distributions.

All things considered, our data against the assumptions of MLR are successfully validated.

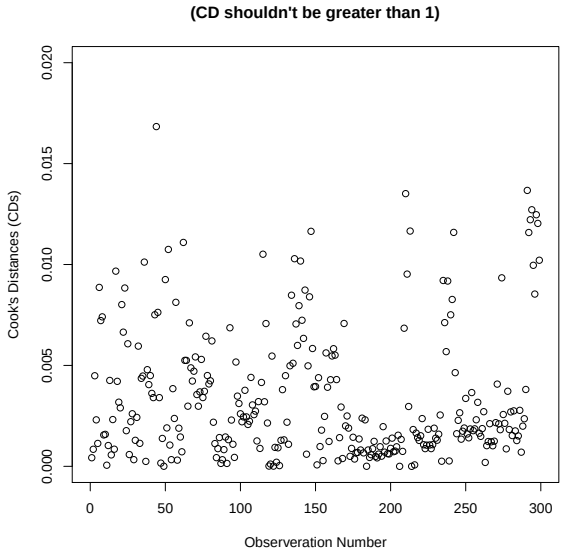
In the form of an MLR equation, the revised model in Figure 7.1 can be expressed as shown below:

$$\begin{aligned} \textit{thrashingPt} &= \alpha + c1 \times \textit{numProcs} + c2 \times \textit{ATPT} + c3 \times \textit{SF} + c4 \times \textit{ROSE} + c5 \times \textit{PK} \\ \textit{ATPT} &= \beta + a1 \times \textit{numProcs} + a2 \times \textit{ROSE} + i1 \times \textit{numProcs} \times \textit{ROSE} + \\ &\quad a3 \times \textit{SF} + a4 \times \textit{PK} + i2 \times \textit{SF} \times \textit{PK}, \end{aligned} \tag{8.1}$$

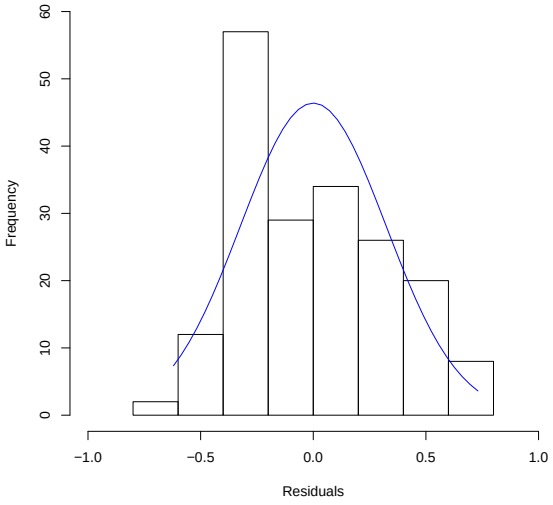
where *thrashingPt*: the thrashing point, *ATPT*: ATPT, α and β: the respective axis intercepts of *ATPT* and *thrashingPt*, *numProcs*, *SF*, *ROSE*, and *PK*: numProcs, (read or update) SF, ROSE, and PK, respectively, $a1$, $a2$, $a3$, and $a4$: the respective coefficients of *numProcs*, *SF*, *ROSE*, and



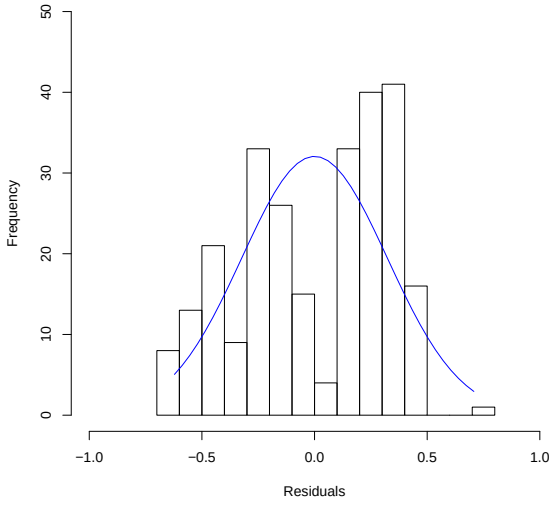
(a) The Outlier Test on the Read Batchset Group



(b) The Outlier Test on the Update Batchset Group



(c) Test of Normality of Residuals on the Read Batchset Group



(d) Test of Normality of Residuals on the Update Batchset Group

Figure 8.1: Testing Multi-Linear Regression Assumptions on the Exploratory Evaluation Data

PK on *ATPT*, $c1$, $c2$, $c3$, $c4$, and $c5$: the respective coefficients of *numProcs*, *ATPT*, *SF*, *ROSE*, and *PK* on *thrashingPt*, $i1$ and $i2$: the respective levels (coefficients) of the correlations between *numProcs* and *ROSE* and between *SF* and *PK*.

8.2 Descriptive Statistics

Tables 8.1 and 8.2 show the overall statistics on the batchsets used in the exploratory experiment: cumulative hours of experiment runs, and the numbers of BSIs, BIs, and BEs, respectively. As exhibited in Table 8.1, it took about 4 months to collect the exploratory evaluation data from the five DBMSes. The most cumulative hours were spent on DBMS X because of nontrivial termination delay and table loading time described in Section 7.1.1. Meanwhile, DBMS Z had the fewest hours thanks to the shortest loading time. For the other DBMSes a reasonable amount of time was spent.

DBMS	Hours
DBMS X	891
MySQL	530
DBMS Y	677
PostgreSQL	580
DBMS Z	449
Total	3,127 (130 days = about 4 months)

Table 8.1: Cumulative Hours of Experiment Runs for the Exploratory Evaluation

Total Batchset Instances (BSIs)	28 (batchsets) $\times$ 40 (runs) = 1,120
Total Batch Instances (BIs)	11,200
Total Batch Executions (BEs)	33,600

Table 8.2: Information about the Batchsets Executed for the Exploratory Evaluation

Regarding the batchset statistics shown in Table 8.2, each run included 28 batchsets, and we had 40 completed runs, as described in Section 6.2. Therefore, we had a total of 1,120 BSIs. As each BSI had ten batches, we had 1,120 $\times$ 10 = 11,200 BIs. We executed each batch three times. We thus had 11,200 BI $\times$ 3 = 33,600 BEs.

Tables 8.3, 8.4, and 8.5 exhibit the results of the sanity checks of TAP, described in Section 6.3. As shown in Table 8.3, we didn't see any violation in the experiment-wide sanity checks; no missing batches, no mismatching processor configuration, no missed BEs, no other running executors, and no other DBMS processes. We thus can proceed onto the next sanity checks.

Table 8.4 shows the results of the BE sanity checks. Regarding (6), we didn't see many zero TPS violations. Only four violations were observed in the data. The percentage was only about 0.01%, which was very low as expected. Concerning (7), about 5.72% of violations were observed across all BEs in our data. For such violations, after CTL some clients were still unable to

1	*Number of Missing Batches*	0
2	*Number of Inconsistent Processor Configuration Violations*	0
3	*Number of Missed Batch Executions*	0
4	*Number of Other Executor Violations*	0
5	*Number of Other DBMS Process Violations*	0

Table 8.3: Experiment-Wide Sanity Checks in the Exploratory Evaluation

6	*Percentage of Zero TPS*	0.01% (4/33600)
7	*Percentage of Connection Time Limit Violations*	5.72% (1923/33600)
8	*Percentage of Abnormal ATPT*	1.10% (371/33600)

Table 8.4: Batch Execution Sanity Checks in the Exploratory Evaluation

9	*Percentage of Transient Thrashing*	0%(0/1120)

Table 8.5: A Batchset Instance Sanity Check in the Exploratory Evaluation

immediately disconnect from a DBMS although our timeout approach mentioned in Section 7.1.1 was applied. It seems that since many Java threads hung around a BE (particularly at higher MPLs), it was hard for the main program control to make a timely context swap following the completion of BE. That said, we did not have a concern on these violations, as the percentage was still low. With regards to (8), about 1.10% violations were observed. As can be seen in Table 8.5, we observed no violations of (9), concerning the batchset sanity check.

Now that we see that the original data from our experiment passed all of these sanity checks, we proceed onto the remaining steps of TAP.

Table 8.6 shows how many BEs were valid at the beginning of Step 2 and after Step 2.

At Start of Step 2	33,600 BEs
At Start of Step 2	1,120 BSIs
After Step 2	31,093 BEs (7.46% dropped)
At Start of Step 3	1,112 BSIs (7.14% dropped)
After Step 3-(i)	1,003 BSIs
After Step 3-(ii)	1,003 BSIs
After Step 3-(iii)	1,003 BSIs (9.80% dropped)

Table 8.6: The Number of Batch Executions and Batchset Instances after Each Sub-Step

At Step 2, we dropped the BEs identified as problematic in Table 8.5, and consequently about 7.46% BEs were dropped. Table 8.6 also exhibits how many BSIs were dropped after each sub-step in Step 3. As indicated by the last row of Step 2, we initially had 1,120 BSIs, and only 8 BSIs

were dropped when starting Step 3. Step 3-(i) dropped BSIs that did not have any BI (MPL) in the corresponding BSIs. Recall that each batchset had ten batches. If there was any BI dropped in the same BSI, we dropped that BSI at this step. There were 109 BSIs dropped. Step 3-(ii) dropped BSIs revealing transient thrashing at Step 1, as indicated by Table 8.5. Again, no BSIs showed any violations. Step 3-(iii) again dropped BSIs revealing transient thrashing among the retained BSIs until this step. No BSIs were dropped. We dropped about 9.80% BSIs throughout Step 3.

Table 8.7 shows the measured thrashing statistics based on the thrashing point, which we calculated for each of the retained BSIs at Step 4. Several conclusions from this table can be drawn as follows. First, *every* DBMS exhibited thrashing. The number of thrashing BSIs differed by about a factor of three across DBMSes. This phenomenon, thus, is likely to be a fundamental aspect of either the algorithm (concurrency control), the creator of the algorithm (human information processing), or system operation context (environment). Our model covers all of these effects. Specifically, about half of the BSIs (487 out of 1,003) exhibited thrashing somewhere in the range of varying MPLs in the exploratory experiment. In particular, there were of the thrashing 487 BSIs for which the DBMSes revealed "early" thrashing, that is, under MPL 300.

DBMS	# of Thrashing BSIs	# of Retained BSIs	Thrashing Percentage
DBMS X	48	201	23.88%
MySQL	73	197	37.05%
DBMS Y	99	199	49.74%
PostgreSQL	115	190	60.52%
DBMS Z	152	216	70.37%
Total	487	1,003	49.45%

Table 8.7: Statistics about the Batchset Instances Faced with Thrashing

8.3 Correlational Analysis

We examined Hypotheses 1–10 summarized in Table 8.8, using the strength and significance of correlations of variables involved. (We'll also see the overall fit and validity of the model via regression on each group in the next section.) To test the hypotheses, we computed pairwise coefficients between an independent variable and a dependent variable via `cor.test()` of R.

Table 8.9 lists the hypotheses followed by the computed correlation factors (coefficients of Equation 8.1), when testing each hypothesis for the *read* group. In the table, "NS" denotes not significant at the 0.05 level, the accepted standard for significance. "—" denotes no prediction

Hypothesis 1:	"*As the number of processors increases, the thrashing point will decrease*."
Hypothesis 2:	"*As the number of processors increases, ATPT will decrease*."
Hypothesis 3:	"*As ATPT increases, the thrashing point will decrease*."
Hypothesis 4:	"*As SF increases, the thrashing point will decrease*."
Hypothesis 5:	"*As SF increases, ATPT will increase*."
Hypothesis 6:	"*As ROSE increases, the thrashing point will increase*."
Hypothesis 7:	"*ROSE will moderate the strength of the correlation between the number of processors and ATPT*."
Hypothesis 8:	"*In the presence of primary keys, the thrashing point will increase*."
Hypothesis 9:	"*In the presence of primary keys, ATPT will decrease*."
Hypothesis 10:	"*The presence of primary keys will moderate the correlation between SF and ATPT*."

Table 8.8: Revised Hypotheses

Variable	*Thrashing Pt.*	*ATPT*
numProcs	H1: 0.13	H2: -0.13
ATPT	H3: -0.38	—
SF	H4: NS	H5: NS
ROSE	H6: NS	—
PK	H8: 0.40	H9: -0.36

Table 8.9: Testing Hypotheses 1–10: Correlations on the Read Batchset Group

arising from the revised model. Regarding a correlation factor c, if |c| is $\sim$ 0.3, $\sim$ 0.7, and $\sim$ 1, the level of the correlation is considered *low*, *medium*, and *high*, respectively. The predicted level shown in Table 7.1 is compared with the actual level determined along with the cutoff of c.

Hypotheses 2 and 3 were supported and significant, and their actual levels were consistent with our model, as can be seen in Table 8.9. Hypotheses 8 and 9 were also supported and significant although the strength was medium close to high that the model predicted. Hypotheses 4, 5, and 6 were not significant. Hypothesis 1 was significant but not supported, because the direction of the actual correlation was positive (counter-intuitive). The correlation coefficient of ROSE and ATPT was treated as zero, as there was no hypothesized correlation between them.

Table 8.10 exhibits the hypotheses followed by the computed correlation factor observed when testing each hypothesis for the update group. As seen in the case of the read group, Hypotheses 1, 2, and 3 were significant and supported. Their actual levels were medium, medium, and low, close to high, low, medium that the model predicted, respectively. Hypotheses 4, 5, 6, 8, and 9 were not significant. It seemed that it was more challenging to answer the hypotheses on the update group.

Variable	*Thrashing Pt.*	*ATPT*
numProcs	H1: -0.38	H2: -0.33
ATPT	H3: -0.11	—
SF	H4: NS	H5: NS
ROSE	H6: NS	—
PK	H8: NS	H9: NS

Table 8.10: Testing Hypotheses 1–10: Correlations on the Update Batchset Group

8.4 Regression Analysis

We also ran a regression over the independent variables of the revised model over the data from the exploratory experiment. For the Equation 8.1, we tested its overall fit on each group, using `lm()` of R [59]. For the read group our model explained 11% of the variance of the dependent variable of thrashing point, and for the update group our model explained 14% of the variance of the thrashing point. Overall, the above percentages of variance explained in both groups were lower than expected.

The following reasons may account for the low amount of variance explained for the thrashing point. First, the effects of some variables affecting the thrashing point were not consistent across DBMSes. In the read group the direction from ATPT to the thrashing point was positive on DBMS Y, whereas it was negative on the other DBMSes. In the update group, the direction from numProcs to the thrashing point was negative on DBMS Y, PostgreSQL, and MySQL while it was positive on DBMSes X and Z.

Another suspicion was from the per-DBMS feature-scaling we performed on the dependent variables of thrashing point and ATPT. The DBMSes were distinct in terms of code and performance, and thus they responded much differently to the dependent variables in our experiments; given the same batchset a DBMS was faced with thrashing while another DBMS was not. It was difficult to obtain high explained variance based on the data differently scaled across the DBMSes.

The third suspicion was that 1) we didn't fully understand the variables like ROSE and SF, and 2) other factors significantly impacting on the thrashing point could probably exist. Although we tried to extract as many, supposedly significant variables as possible, we realize that the above variables were not understood well, and there might be more significant variables, too.

The fourth suspicion was concerned about some violations by `durbinWatsonTest()`. The amount of variance explained might be decreased by serial correlation of error terms found in

our model. To handle the auto-correlation violation, more thrashing samples should have been collected in the exploratory experiment.

The last suspicion was from the choice of the MLR statistical tool for testing the model. MLR was chosen as the most appropriate one for our analysis, but there may exist other appropriate tools, such as the logistic regression mentioned before or structural equation modeling [87] (SEM) for analyzing our data. Investigating a better statistical tool is left in the future work.

We also did regressions on the dependent variable of ATPT. Our model explained 12% and 11% of the variance accounting for ATPT on the read and update groups, respectively. We doubt that these low variance accounting for ATPT also result from the above four suspicions.

8.5 Path Analysis

We performed path analysis on the revised model in Figure 7.1. We took the standardized estimates (referred as *regression* or *path coefficients* [91]) of the independent variables (e.g., numProcs, ROSE, SF, and PK) and their hypothesized moderations (e.g, ROSE:numProcs[1] and PK:SF) participating in the regressions on the dependent variables (e.g, ATPT and thrashing point). We then computed the weights of direct and indirect paths to the dependent variables. For more detail on the regression outputs, refer to the paragraph of regression analysis in Section A.4.1.

Concerning the read group, Figure 8.2 depicts the path diagrams on the revised model in Figure 7.1. Note that in Figure 8.2 each arrow is either in *black* or in *gray*, depending on whether or not it is significant at the significance level (α) of 0.05, respectively. For instance, since the arrow from numProcs to ATPT was significant, it is rendered in black in Figure 8.2(a). Since the arrow from ROSE onto the arrow from numProcs to ATPT was not significant, on the other hand, it is rendered in gray.

Also note that the weight of each arrow is placed above that arrow. For example, the weight of the arrow from numProcs to ATPT is -0.87, and the moderating arrow from ROSE has a weight of 0.108, which corresponds to the regression coefficient of the moderation of ROSE on the relationship of numProcs to ATPT.

[1] ':' indicates a correlation between a moderator and an independent variable. Recall that in Figure 7.1 ROSE (a moderator) affects the relationship of numProcs (an independent variable) to ATPT (a independent variable). The moderation is reinterpreted as the effect of ROSE on the correlation between numProcs and numProcs, contributing to the variance of ATPT.

Furthermore, we have the incoming arrows denoted by "e_{atpt}" and "e_{tp}," which indicate the error variances of ATPT and thrashing point, respectively. Regarding the other constructs in the model depicted in Figure 7.1, there are no error variances associated with them, since the constructs have only independent variables with no incoming arrows. The error variances of e_{atpt} and e_{tp} can be computed as 1- $\sqrt{(1-R^2)}$ (e.g, e_{tp} = $\sqrt{(1-0.11)}$ = 0.943) [19]. These two error variances will be used to estimate the fits of the full and reduced (to be described shortly) path models in Figure 8.2.

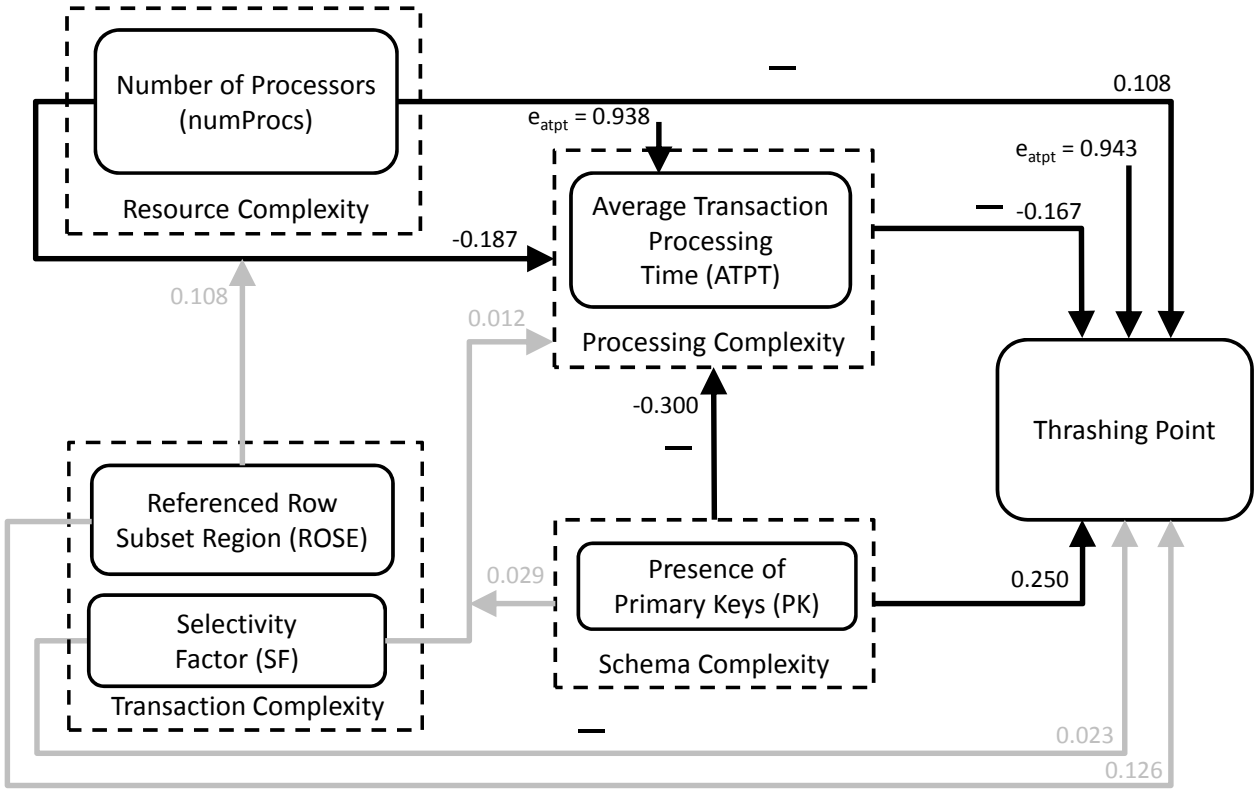
Figure 8.2(a) shows some paths that are not statistically significant, involving ROSE and SF. As discussed in Section 8.3, the correlations associated with ROSE and SF were not significant. Similarly, all the paths associated with ROSE and SF did not contribute much to the variances of ATPT and thrashing point. Hence, the insignificant paths are considered removed.

Figure 8.2(b) depicts the reduced path model to remove the paths that were not significant. We then again performed regression analysis on the reduced model. As a result, the weights of the arrows got slightly increased or decreased, compared to those of Figure 8.2(a). However, all of the arrows were significant. The direct effect of PK to ATPT (-0.287) was stronger than that (-0.117) of numProcs to ATPT. In addition, the total (indirect + direct) effect (-0.287 $\times$ (-0.170) + 0.215 = 0.264) of PK on thrashing point was slightly greater than that (-0.117 $\times$ (-0.170) + 0.130 = 0.150) of numProcs and that (-0.170) of ATPT on thrashing point. In the read model, therefore, the most significant factor on ATPT and on thrashing point was PK. We also see that numProcs and PK had positive correlations with thrashing point while ATPT had a negative correlation with thrashing point.

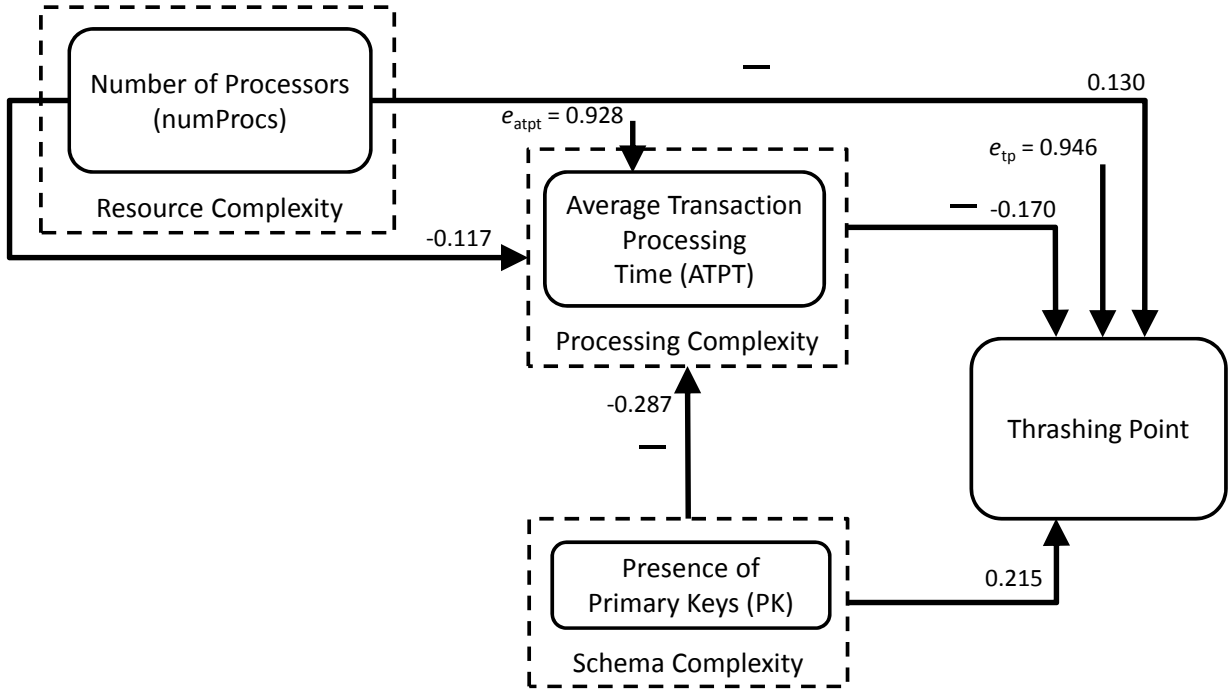
As expected, e_{tp} was decreased due to the removal of the insignificant paths. In the meantime, e_{atpt} was slightly decreased. This decrease appeared attributed to the path deletion as well, perhaps contributing to eliminating some outliers in the fit of ATPT. Further possibilities can be explored in future work.

In addition, we compared the fits of the full and reduced path models. The estimated fits were as follows [19, 56]:

$$\begin{aligned} \text{Fit for the full path model} &= 1 - e_{atpt}^2 \times e_{tp}^2 = 1 - 0.938^2 \cdot 0.943^2 = 0.2176 \\ \text{Fit for the reduced path model} &= 1 - e_{atpt}^2 \times e_{tp}^2 = 1 - 0.928^2 \cdot 0.946^2 = 0.2293. \end{aligned} \tag{8.2}$$



(a) The Full Path Model



(b) The Reduced Path Model

Figure 8.2: Path Diagrams of the Read Batchset Group in the Exploratory Experiment

The summary statistic showing the relative fit (Q) of the reduced model to the full model was

$$Q = \frac{1 - \text{fit of the full model}}{1 - \text{fit of the reduced model}} = \frac{1 - 0.2176}{1 - 0.2293} = 1.015. \tag{8.3}$$

To perform the significance test to compare the fit of the two models, we calculated a χ statistic (W_r) [47] in the following (N = sample size, d = number of dropped paths):

$$W_r = -(N - d) \times \ln Q = -(188 - 5) \times \ln (1.015) = 2.562. \tag{8.4}$$

W_r is distributed from χ with degree of freedom (df) = d. According to the χ distribution table [86], the χ critical value [47] (CV) is equal to 11.070 (df = 5 and α = 0.05). As W_r (2.562) is smaller than CV (11.070), we conclude that the reduced model did not fit the data any worse than the full model, which includes the eliminated paths.

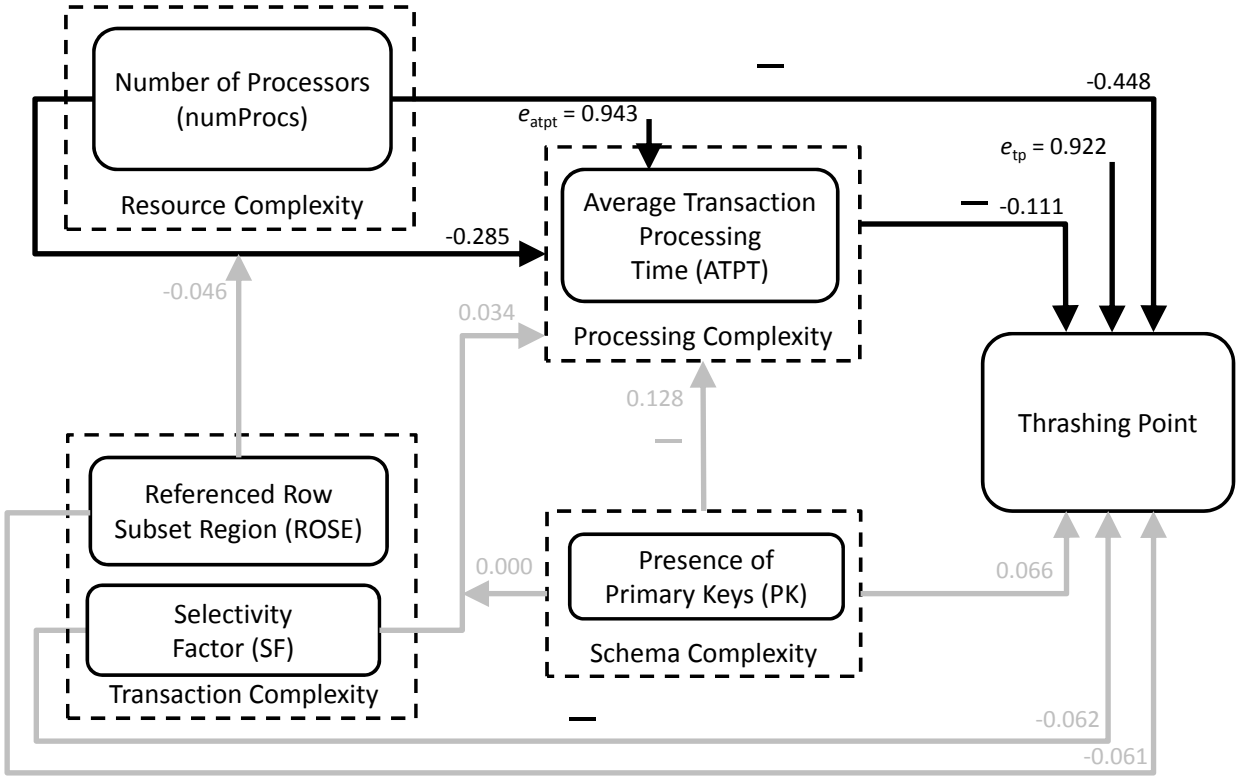
Figure 8.3 depicts the path diagrams of the revised model on the update group. Figure 8.3(a) shows several paths that were not significant. Specifically, the paths involving PK were not significant.

Figure 8.3(b) shows the reduced path model that removes all the paths that were not significant. As with the read group, we performed regression analysis on the reduced model. Consequently, all the retained paths were significant even though their weights were slightly increased or decreased. The total effect (-0.296 $\times$ (-0.096) - 0.434 = -0.406) of numProcs to thrashing point was much stronger than that (-0.096) of ATPT to thrashing point. In the update model, therefore, the most significant factor on ATPT and on thrashing point was numProcs. As predicted, numProcs and ATPT had negative correlations with thrashing point in our model.

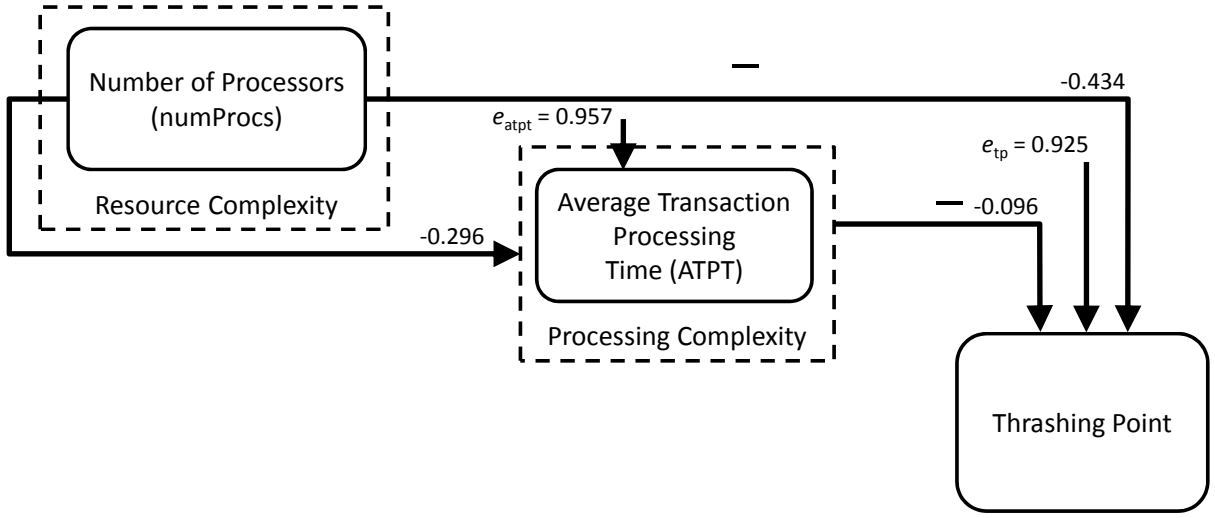
e_{atpt} and e_{tp} in Figure 8.3(b) were slightly increased because of the deleted paths, compared to those of the full model in Figure 8.3(a).

We also compared the fits of the full and reduced models on the update group. As with the read group above, we estimated the fits as follows:

$$\begin{aligned} \text{Fit for the full path model} &= 1 - e_{atpt}^2 \times e_{tp}^2 = 1 - 0.943^2 \cdot 0.922^2 = 0.2441 \\ \text{Fit for the reduced path model} &= 1 - e_{atpt}^2 \times e_{tp}^2 = 1 - 0.957^2 \cdot 0.925^2 = 0.2164. \end{aligned} \tag{8.5}$$



(a) The Full Path Model



(b) The Reduced Path Model

Figure 8.3: Path Diagrams of the Update Batchset Group in the Exploratory Experiment

The summary statistic showing the relative fit (Q) of the reduced model to the full model was

$$Q = \frac{1 - \text{fit of the full model}}{1 - \text{fit of the full model}} = \frac{1 - 0.2441}{1 - 0.2164} = 0.9647. \tag{8.6}$$

As was done with the read group, we calculated a χ statistic (W_u) as follows (N = 299, d = 7):

$$W_u = -(N - d) \times \ln Q = -(299 - 7) \times \ln (0.9647) = 10.494. \tag{8.7}$$

The χ critical value (CV) is equal to 14.067 (df = 7 and α = 0.05) in accordance with the χ distribution table. W_u (10.494) is smaller than CV (14.067). Hence, we conclude that the reduced path model did not fit the data more poorly than did the full path model on the update group.

8.6 Causal Mediation Analysis

So far we have tested the individual correlations and the overall fit of the revised model shown in Figure 7.1. In the model we still need to finish testing the remaining mediation and moderation relationships related to H7 and H10. (Recall that we have described the concept of mediation, moderation, and moderated mediation on pages 58–59.) In this section we provide the details of how we tested such relationships and present the evaluation results.

Technically, the revised model includes two mediations that we expect to be moderated by ROSE and PK. We need a proper statistical tool for testing the strength and significance of the moderated mediations in the model. As mentioned in Section 1.2, one method is causal mediation analysis (CMA) [30]. CMA works by 1) building and fitting one or more linear regression models, 2) connecting to the model for the outcome variable from the model involving a mediator, and then 3) calculating the estimates of the direct, indirect, and moderated effects from the fitted models. Hence, CMA is a perfect fit for the revised model that satisfies such conditions.

To apply CMA, our data need to satisfy the assumptions of linearity and independence among variables [85]. Let's first discuss the linearity assumption. In the read group every variable except ROSE and SF were significant, as shown in Table 8.9. In the update group only the variables of numProcs and ATPT were significant on the thrashing point, as seen in Table 8.10. The linearity assumption, therefore, was satisfied on the rest of the variables in the model.

Let's move on to the independence assumption. Our model's variables were independent of each other, as addressed before. Thus, there was no violation of this assumption.

Although our data had some variables such as ROSE and SF violating linearity with the thrashing point, CMA was the best tool for testing the moderated mediations in the revised model. To get around the violation issue, we decided not to test the mediations associated with the variables

that were not significant in each group. This was because we did not adequately understand the variables that were found to be insignificant. (Considering the variables is left in the future work.)

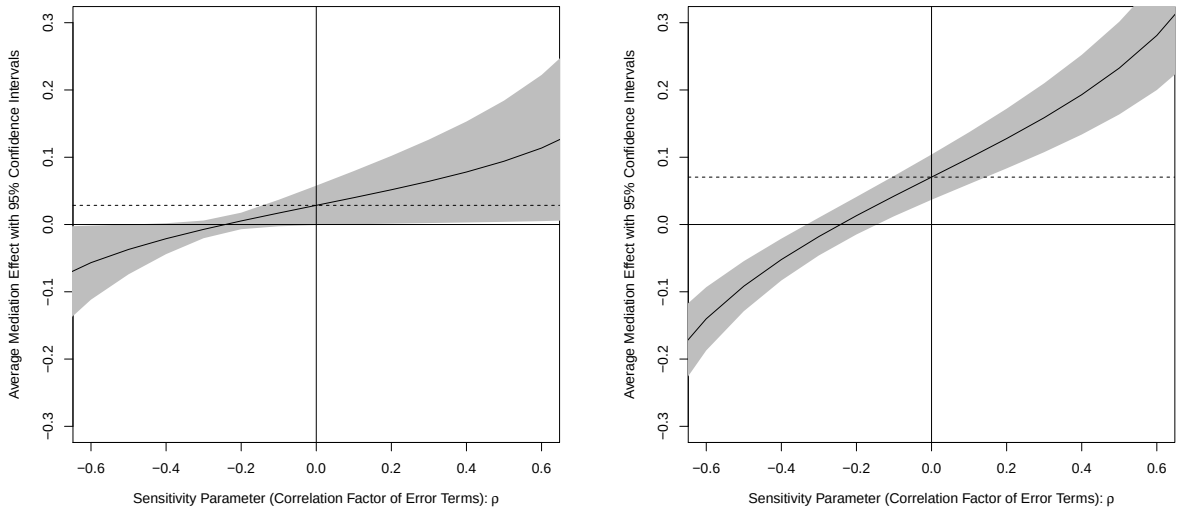
To proceed with CMA, we used R's library package called `mediation` [85]. This easy-to-use software is freely downloadable via the Comprehensive R Archive Network (CRAN) [65]. It contains useful methods (`mediate()` for testing a mediation relationship and `test.modmed()`) for testing a moderated mediation relationship.

We now elaborate in detail the results of our evaluation using the `mediation` library. To test the mediations of the model, we used the `mediate()` function. In the read group, we found that the mediating effects of numProcs and PK via ATPT on thrashing point were statistically different from zero. In the update group, only the mediating effect of numProcs via ATPT on the thrashing point was statistically significant. Our findings from these results revealed that the mediation through ATPT occurred on the thrashing point in both groups, although the origins of indirect effects were different. We also tested the moderated mediations using the `test.modmed()` function. No moderated mediations were statistically significant. There are two possible reasons. First, it seems that we don't understand the moderated mediations in the model. Another is that the variables of ROSE and SF were not significant on the thrashing point. (These possibilities should be investigated in our future work as well.)

To make this CMA complete, we also had to check the violation of another assumption, called *Sequential Ignorability* (SI) [30, 85]. SI implies two premises: (a) operationalizing independent variables is independent of all potential values of the dependent and mediating variables, and (b) the observed mediator is also independent of all potential values of the outcome variable.

The SI assumption cannot be tested directly using the measured data [40]. We need to indirectly test the SI assumption by fitting the data in the model and then conducting *sensitivity analysis* (SA) on the mediations included in the model [30, 32]. SA examines how robust the mediating results are to the SI assumption violation. SA can be conducted by 1) yielding a mediation object (`med.out`) by passing to `mediate()` the regressed models of ATPT and the thrashing point and then 2) invoking `medsens()` with `med.out` in the `mediation` package.

Figure 8.4 graphically shows the results for the read group's SA based on differing values of the sensitivity parameter ρ, equal to the correlation of error terms (residuals) between the two models of the dependent variables of ATPT and thrashing point in Equation 8.1 on page 94. We plot the



(a) Sensitivity Analysis with Respect to Error Correlation between the Mediator of the Number of Processors and Outcome Models

(b) Sensitivity Analysis with Respect to Error Correlation between the Mediator of the Presence of Primary Keys and Outcome Models

Figure 8.4: Sensitivity Analysis for the Read Batchset Group in the Exploratory Experiment

estimated average mediation effect (AME) with 95% confidence interval (CI) on varying the values of ρ. The dashed line shows the estimated mediation effect independent of ρ. The solid line and gray regions represent the estimated AME at a specific ρ and their 95% CIs, respectively. For the SI assumption to be held true, the estimated AME should not be zero when ρ is equal to 0 [30].

Figure 8.4(a) indicates the results for SA for examining whether there exists the mediating effect of ATPT from numProcs on the thrashing point. As can be seen in the figure, the corresponding AME was about 0.03, which was small. When ρ was in [-0.3, -0.2], the estimated AME was zero within the 95% CI. Note that a large value of ρ indicates the existence of strong confounding between the mediator and the outcome, and thus a serious violation of the SI assumption [33]. The ρ range indicates a high degree of the robustness of the mediating effect to the potential violation of SI assumption.

Figure 8.4(b) indicates the results for SA for examining whether there exists the mediating effect of ATPT from PK on the thrashing point. As can be seen in the figure, the corresponding AME was about 0.07, which was small, too. When ρ was also in [-0.3, -0.2], the estimated AME was zero within the 95% CI. Based on these results testing the SI assumption on the fitted models was satisfied. Hence, the mediations via ATPT from numProcs and PK to the thrashing point in the read group were identified.

Figure 8.5 illustrates the results for the update group's SA based on varying values of the sensitivity parameter ρ. It indicates the result for SA for examining whether there exists the mediating effect of ATPT from numProcs on the thrashing point. As can be seen in the figure, the corresponding AME was about 0.04, which was small. When ρ was in [-0.2, -0.1], the estimated AME was zero. Testing the SI assumption, therefore, was satisfied. Hence, the mediation via ATPT from numProcs was identified.

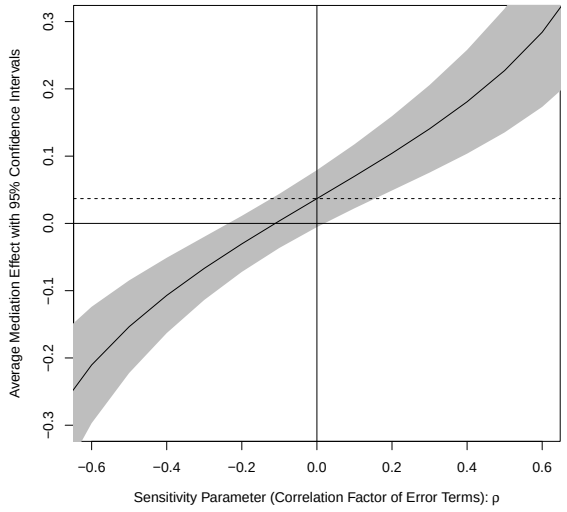


Figure 8.5: Sensitivity Analysis for the Update Batchset Group in the Exploratory Experiment

Note that it was not applicable to conduct SA on the path from PK to ATPT to the thrashing point, because the correlation between PK and ATPT was not significant as exhibited in Table 8.10.

In summary, through these analyses we confirmed the mediation through ATPT to the thrashing point from both numProcs and PK in the read group and from numProcs in the update group.

In the next chapter, we present the final structural causal model, modified based on these exploratory evaluation results.

CHAPTER 9

THE FINAL STRUCTURAL CAUSAL MODEL OF THRASHING

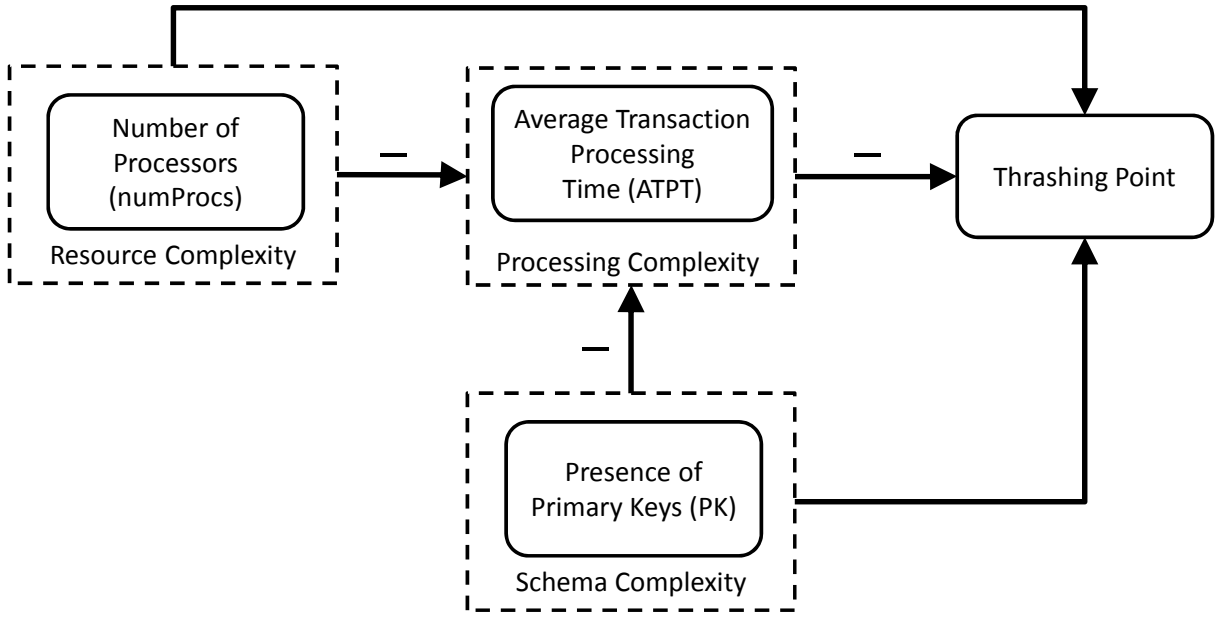
We have finished the exploratory evaluation of the revised model. From this evaluation, we have learned several things. First, our understanding of some variables was not sufficient. In the correlational analysis discussed in Section 8.3, we couldn't observe the associations of ROSE and SF with the dependent variables of ATPT and thrashing point on both of the read and update groups. Additionally, PK was not significant in the update group. We thought these variables were related to the thrashing point, but this was not observed in our data. It seems that 1) our operationalization of these variables might be faulty, 2) the variables would be indeed unrelated to the thrashing point, and 3) overall we didn't understand the variables correctly. More scrutiny seems needed to examine why we couldn't observe such an influence on the thrashing point.

Second, the mediation through ATPT on the thrashing point was identified in the model for both groups although the strength was weak. We observed a negative correlation between ATPT and thrashing point on multiple DBMSes. The mediations moderated by PK and ROSE were not observed in our data. It seems attributed to the fact that PK and ROSE were not significant on the thrashing point. More investigations of these insignificant variables need to be done here as well.

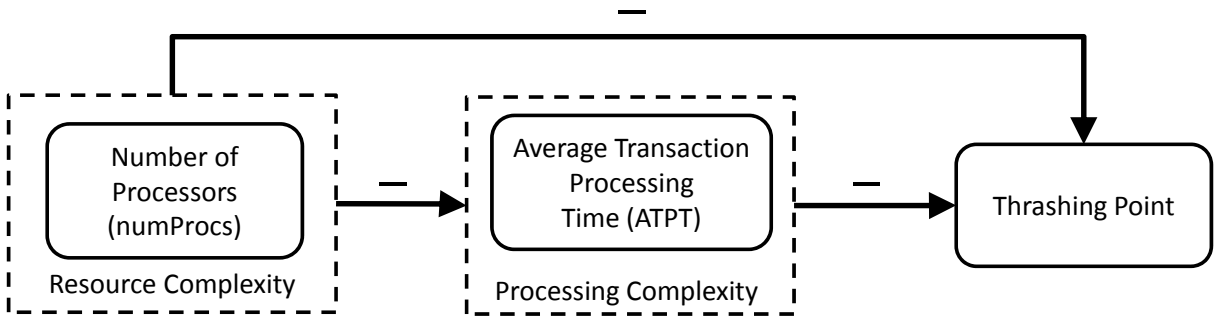
Third, it may not be desirable to have a single common model of explicating DBMS thrashing regardless of transaction type. The reasons are three-fold. 1) The direction of a common correlation was not consistent between the read and update groups. As observed in the read group, for instance, the direct effect of PK on the thrashing point was positive as opposed to our prediction. It made us rethink that in the read group having more processors may actually be helpful to DBMSes, as the disadvantage of contentions among the processors may be hidden by the benefit of increased parallelism helping enhance the throughput. This makes sense, in that DBMSes will experience less lock management overhead when treating read-only transactions than handling update-only ones. In the update group, on the contrary, our prediction was correct. The expected contentions seemed occurring when treating update-only transactions, thereby negatively affecting the thrashing point. This was also consistent with what prior research [34, 35] indicated. 2) PK was insignificant in the update group while it was significant in the read group. 3) The amount

of variance explained by the model was different between the two groups. For these reasons, we think that our model should be divided up into two parts, each explicating the respective thrashing origins in the read and update groups, respectively.

Based on these above Exploratory evaluation lessons, we have brought the final changes into the revised model. Figure 9.1 shows the final structural causal model that consist of the two parts, each representing the respective group.



(a) The Model of the Read Batchset Group



(b) The Model of the Update Batchset Group

Figure 9.1: The Final Model of DBMS Thrashing

Figure 9.1(a) exhibits the model of explicating the variance of DBMS thrashing on the read group. We have removed the transaction complexity constructs and the associated correlations. The elimination was due to the lack of our understanding on the removed variables and associations. In

addition, we changed the direction of the correlation between numProcs and the thrashing point: - $\rightarrow$ +. In other words, we hypothesize that as numProcs increases, the thrashing point will increase in the read group.

Table 9.1 shows the hypothesized correlations and levels on the model in Figure 9.1(a). Note that compared to Table 7.1, group II no longer exists because of the removal of the variables and correlations in the final model. Every cell stays the same as in Table 7.1, except the levels that get decreased to medium from high or increased to medium from low. This is because of our observation that these correlations were weaker or stronger than we thought in the revised model.

	a	*b*
I	numProcs / ATPT *low*	numProcs / Thrashing Pt. *low*
III	PK / ATPT *medium*	PK / Thrashing Pt. *medium*
IV	ATPT / Thrashing Pt. *medium*	

Table 9.1: Hypothesized Correlations and Levels on the Read Batchset Group

Figure 9.1(b) shows the model of explaining the variance of DBMS thrashing on the update group. We have removed the transaction complexity constructs and the associated correlations. In addition, we do not have the schema complexity construct any more. Thus, this model gets simpler than before. As opposed to Figure 9.1(a), we keep the same direction of the association of numProcs with the thrashing point as before: for the update group as numProcs increases, the thrashing point will decrease, as indicated by the top row in Table 8.8. That is because we see that the inter-processor contentions on the internal structures within DBMSes will increase as the number of processors increases.

Table 9.2 shows the hypothesized correlations and levels on the model in Figure 9.1(b). As seen in Table 9.1, groups II and III are left empty in Table 9.2. Every cell stays the same as in Table 7.1, except that the level of IV-(a) is decreased to low from medium. This is because we saw that this correlation was weaker than we thought in the revised model.

	a	*b*
I	numProcs / ATPT *medium*	numProcs / Thrashing Pt. *low*
IV	ATPT / Thrashing Point *low*	

Table 9.2: Hypothesized Correlations and Levels on the Update Batchset Group

CHAPTER 10

CONFIRMATORY EVALUATION OF THE FINAL MODEL

Our prior efforts made for the exploratory evaluation have helped us to not only gather the new empirical data faster but also to make it easier to finish the evaluation of the final model.

As mentioned in Section 6.2 on page 76, the confirmatory data comprise the same 28 batchsets used in the exploratory experiment. For each batchset we have a total of 50 runs made based on 5 numProcs values $\times$ 5 DBMSes $\times$ 2 cases with and without PK. These 50 runs took about 7 cumulative months. We describe in greater detail the confirmatory data in Section 10.2.

In this chapter, we present the evaluation results of the final model. Let's first discuss the results of testing the MLR assumptions, described in Section 8.1, on the confirmatory evaluation data.

10.1 Testing Multi-Linear Regression Assumptions

The assumptions of (i) a continuous dependent variable and (ii) two or more continuous or nominal independent variables were still hold satisfied as nothing changed in the final model except the removal of variables and associations. As will be seen in Table 10.2 on page 118, we collected a total of 482 samples (read: 148 and update: 334) beyond Tabachnick's and Fidell's recommendation described in Section 8.1. Thus, the assumption of (iii) sample size was satisfied.

The assumption of (iv) independence of residuals, however, was not satisfied in either the read or update groups. `durbinWatsonTest()` yielded the D-W (Durbin-Watson) statistics of 1.44 and 1.11 on thrashing point and those of 0.58 and 0.33 on ATPT. We thus detected some type of auto-correlation in our data, as did in the exploratory evaluation. To further look into this violation, we also calculated the per-DBMS D-W statistics for thrashing point and ATPT. As a result, we could find the same issues such as 1) low sample size, 2) some DBMSes revealing auto-correlated samples, and 3) clustered values of ATPT and thrashing point across DBMSes. That said, this violation is not concerned about using MLR for our data for the same reasons as mentioned in the fifth paragraph on page 93. In the future, nevertheless, it is still needed to collect more samples per DBMS and see if the D-W test can be passed at an individual DBMS and the overall level.

If this violation persists in bigger sample sizes, a correction using the Cochrane-Orcutt estimation method [6] may be applied to remedy the model.

Testing the assumption of (v) homoscedasticity via `ncvTest()` was successful, as the p-values of `ncvTest()` were 0.20 and 0.47 for the read and update groups, respectively. The assumption of (vi) multicollinearity was passed in both of the groups, as none of the independent variables in the model had the square root of its VIF fewer than two, meaning that there was no correlation of the variables.

Figure 10.1 exhibits the results of testing the assumptions of (vii) no significant outliers and (viii) normality of residuals. The assumption of (vii) was satisfied in the read and update groups, as shown in Figures 10.1(a) and 10.1(b). Testing the assumption of (viii) was successful on the read and update groups. As illustrated in Figures 10.1(c) and 10.1(d), the residuals of our data revealed approximate normality.

In short, we see that our data were successfully validated on the assumptions.

In the form of an MLR equation, the model on the read and update groups, depicted in Figures 9.1(a) and 9.1(b), can be given as seen in the following Equations 10.1 and 10.2:

$$\begin{aligned} \textit{thrashingPt} &= \alpha + c1 \times \textit{numProcs} + c2 \times \textit{ATPT} + c3 \times \textit{PK} \\ \textit{ATPT} &= \beta + a1 \times \textit{numProcs} + a2 \times \textit{PK} \end{aligned} \tag{10.1}$$

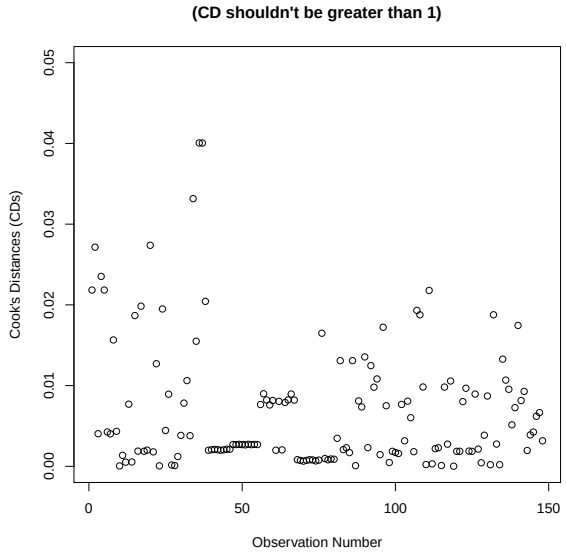
$$\begin{aligned} \textit{thrashingPt} &= \alpha + c1 \times \textit{numProcs} + c2 \times \textit{ATPT} \\ \textit{ATPT} &= \beta + a1 \times \textit{numProcs}, \end{aligned} \tag{10.2}$$

where the corresponding descriptions of the variables, intercepts, and coefficients were already provided in Section 8.1. The evaluation results on the above equations are presented in the next section.

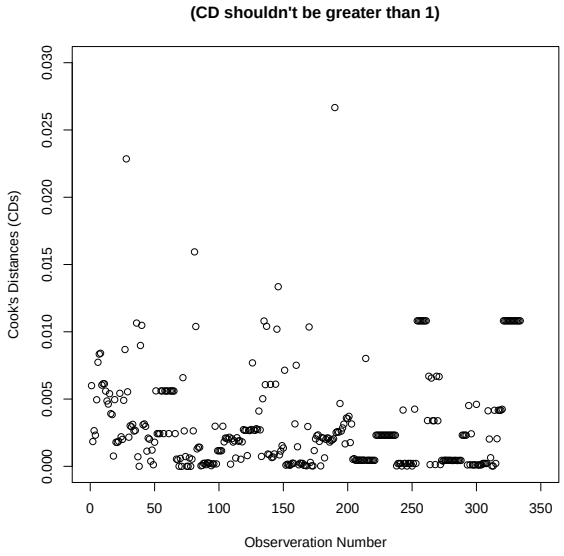
10.2 Descriptive Statistics

Tables 10.1 and 10.2 show the overall statistics on the batchsets used in the confirmatory experiment: about cumulative 7 months, and the numbers of BSIs, BIs, and BEs, respectively.

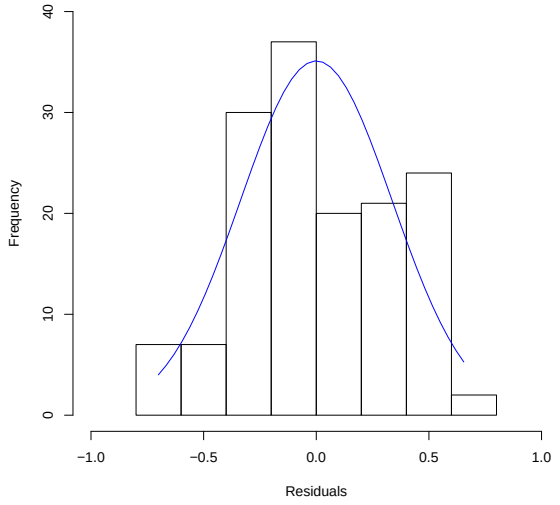
This experiment took about 65% more in cumulative hours, compared to the cumulative hours (about four months) in Table 8.1. There were two reasons of why such a large amount of time



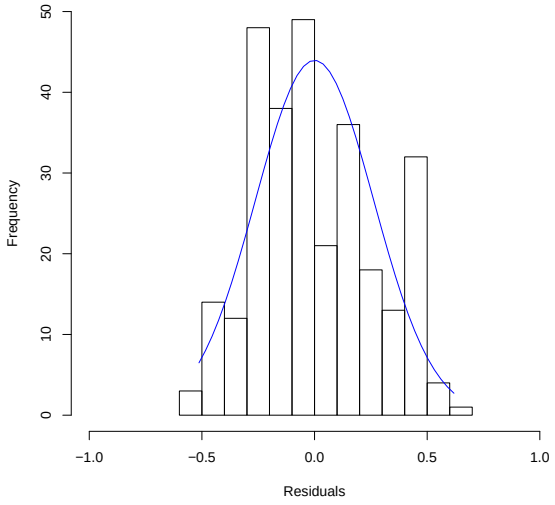
(a) The Outlier Test on the Read Batchset Group



(b) The Outlier Test on the Update Batchset Group



(c) Test of Normality of Residuals on the Read Batchset Group



(d) Test of Normality of Residuals on the Update Batchset Group

Figure 10.1: Testing Multi-Linear Regression Assumptions on the Confirmatory Evaluation Data

was spent in the confirmatory experiment; 1) we collected ten more runs from the five DBMSes than we did in the exploratory evaluation data, and these runs resulted from one more value (six processors) added when operationalizing numProcs, and 2) we increased the number of batch repetitions at each MPL by five, to achieve better accuracy and precision in TPS and thrashing point calculation.

DBMS	Hours
DBMS X	1,107
MySQL	1,038
DBMS Y	1,020
PostgreSQL	1,038
DBMS Z	967
Total	5,170 (215 Days = about 7 months)

Table 10.1: Cumulative Hours of Experiment Runs for the Confirmatory Evaluation

Total Batchset Instances	1,400
Total Batch Instances	14,000
Total Batch Executions	70,000

Table 10.2: Information about the Batchsets Executed for the Confirmatory Evaluation

In the case of DBMSes X and Y, it took about 20% and 34% more time. As discussed in Section 7.1.1, we had some significant connection termination delay on DBMS X and lengthy table population time on DBMSes X and Y. These problems were resolved in the run of the exploratory experiment. As a result, the problems did not arise in the confirmatory experiment.

Regarding the confirmatory batchset statistics, the same 28 batchsets were run with PK and without PK on five DBMSes running on our experiment machine configured with one, two, four, six, and eight processors. We then had 28 (batchsets) $\times$ 2 (PK or non-PK) $\times$ 5 (DBMSes) $\times$ 5 (numProcs) = 1,400 BSIs. As each BSI had ten batches, we had 14,000 BIs. In the confirmatory experiment each BI was executed five times, and thus we had a total of 70,000 BEs.

Tables 10.3, 10.4, and 10.5 show the sanity check results on the confirmatory evaluation data.

As illustrated in Table 10.3, no violation is observed in the confirmatory experiment-wide sanity checks. Regarding the BE sanity checks exhibited in Table 10.4, we have 0.04% zero TPS violations, 3.48% connection time limit violations, and 0.97% abnormal ATP violations. These rates are close to or even lower than those of the exploratory evaluation data in Table 8.4. As exhibited in Table 10.5, we have no violations of (9) concerning the batchset sanity check.

The raw data have passed all of these sanity checks, and we thus proceed onto the rest steps of TAP.

At Step 2 we drop the BEs identified as problematic in Table 10.5. Table 10.6 shows how many BEs are valid at the beginning of Step 2 and after Step 2: about 4.39% BEs were dropped. Table 10.6 also exhibits how many BSIs were dropped after each sub-step in Step 3. As indicated

1	*Number of Missing Batches*	0
2	*Number of Inconsistent Processor Configuration Violations*	0
3	*Number of Missed Batch Executions*	0
4	*Number of Other Executor Violations*	0
5	*Number of Other DBMS Process Violations*	0

Table 10.3: Experiment-Wide Sanity Checks in the Confirmatory Evaluation

6	*Percentage of Zero TPS*	0.04% (25/70000)
7	*Percentage of Connection Time Limit Violations*	3.48% (2434/70000)
8	*Percentage of Abnormal ATPT*	0.97%(679/70000)

Table 10.4: Batch Execution Sanity Checks in the Confirmatory Evaluation

9	*Percentage of Transient Thrashing*	0% (0/1400)

Table 10.5: A Batchset Instance Sanity Check in the Confirmatory Evaluation

by the last row of Step 2, interestingly we had no dropped BSIs when starting Step 3; every BSI survived. At Step 3-(i) we dropped BSIs that did not have ten batches. There were 61 BSIs dropped at this step. Step 3-(ii) dropped BSIs revealing transient thrashing at Step 1. Here there was no need to drop BSIs, as indicated by Table 10.5. Step 3-(iii) again dropped BSIs revealing transient thrashing among the remaining BSIs until this step. No BSIs were dropped. In sum, we dropped about 4.35% BSIs throughout Step 3.

At Start of Step 2 *At Start of Step 2*	70,000 BEs 1,400 BSIs
After Step 2	66,927 BEs (4.39% dropped)
At Start of Step 3	1,400 BSIs (0% dropped)
After Step 3-(i)	1,339 BSIs
After Step 3-(ii)	1,339 BSIs
After Step 3-(iii)	1,339 BSIs (4.35% dropped)

Table 10.6: The Number of Batch Executions and Batchset Instances after Each Sub-Step

At Step 4, we computed the value of the thrashing point for each of the retained BSIs in the same way as was done with the exploratory experiment. Table 10.7 shows the measured thrashing statistics based on the calculated thrashing point after Step 4.

The percentage of the thrashing batchsets was decreased to 36% from 49% in the exploratory experiment. While investigating why the thrashing rate dropped in the confirmatory evaluation data, we discovered the issue of *thrashing repeatability*. Specifically, for the same batchset some DBMSes did reveal thrashing in the exploratory but not in the confirmatory experiment, and vice

DBMS	# of Thrashing BSIs	# of Retained BSIs	Thrashing Percentage
DBMS X	39	265	15%
MySQL	76	264	29%
DBMS Y	86	262	33%
PostgreSQL	140	268	52%
DBMS Z	141	280	50%
Total	482	1,339	36%

Table 10.7: Statistics about the Batchset Instances Faced with Thrashing

versa; it was not repeatable to observe thrashing. This unrepeatability was a real challenge in the evaluation of our model, as the more thrashing samples we had, the better we could explain the variance of thrashing.

Despite the challenge we reaffirmed in the confirmatory experiment the findings from the exploratory evaluation: 1) every DBMS showed thrashing, and 2) there were still many BSIs (about one-thirds (149)) for which the DBMSes experienced "early" thrashing, that is, under MPL 300, in the confirmatory experiment.

Now, let's move onto our statistical analysis on the retained BSIs.

10.3 Correlational Analysis

As we did in the exploratory experiment, we conducted the confirmatory pair-wise correlational analysis on the final model. Regarding the read group, Table 10.8 summarizes the refined hypotheses drawn from the model shown in Figure 9.1(a).

Hypothesis 1:	"*As the number of processors increases, the thrashing point will increase*."
Hypothesis 2:	"*As the number of processors increases, ATPT will decrease*."
Hypothesis 3:	"*As ATPT increases, the thrashing point will decrease*."
Hypothesis 4:	"*In the presence of primary keys, the thrashing point will increase*."
Hypothesis 5:	"*In the presence of primary keys, ATPT will decrease*."

Table 10.8: Hypotheses on the Read Batchset Group

Only the relevant hypotheses in Table 8.8 are left, as shown in Table 10.8. There is no difference from that table except H1, which just reflects the revised direction of influence of numProcs on thrashing point shown in Figure 9.1(a).

Table 10.9 lists the hypotheses followed by the computed correlation factors (coefficients of Equation 10.1) when testing each hypothesis for the read group. All the hypotheses (H1–H5)

were supported and significant. Most of the actual levels in Table 10.9 were consistent with our prediction presented in Table 9.1.

Variable	*Thrashing Pt.*	*ATPT*
numProcs	H1: 0.30	H2: -0.32
ATPT	H3: -0.26	—
PK	H4: 0.20	H5: -0.55

Table 10.9: Testing Hypotheses 1–5: Correlations on the Read Batchset Group

The actual levels of H2, H3, and H4 were medium, low, and low. They were slightly different from the predicted levels of low, medium, medium, respectively. That is, the actual levels of the correlations between numProcs and ATPT, between ATPT and thrashing point, and between PK and thrashing point were respectively stronger and weaker than we predicted. Since such a minor difference is acceptable, we claim that our model for the read group was empirically confirmed.

Table 10.10 lists the hypotheses about the final model in Figure 9.1(b) on the update group. Only the relevant hypotheses among ones exhibited in Table 8.8 were left, too. No refinement was made to these hypotheses regarding the update group.

Hypothesis 1:	"*As the number of processors increases, the thrashing point will decrease*."
Hypothesis 2:	"*As the number of processors increases, ATPT will decrease*."
Hypothesis 3:	"*As ATPT increases, the thrashing point will decrease*."

Table 10.10: Hypotheses on the Update Batchset Group

Table 10.11 exhibits the hypotheses followed by the computed correlation factor observed when testing each hypothesis for the update group.

Variable	*Thrashing Pt.*	*ATPT*
numProcs	H1: -0.36	H2: -0.16
ATPT	H3: -0.13	—

Table 10.11: Testing Hypotheses 1–3: Correlations on the Update Batchset Group

All the hypotheses (H1–H3) were supported and significant. In addition, the actual levels were all low except that of H1 whose level was medium. The levels were almost consistent with the predicted levels from the model. Hence, our model for the update group was empirically supported as well.

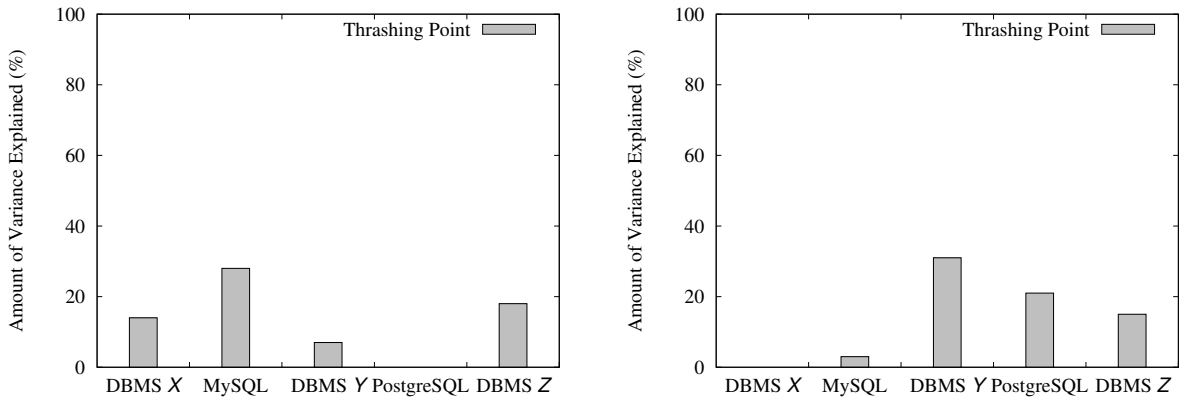
10.4 Regression Analysis

We ran a regression over the independent variables of the final model that predicts the thrashing point over the overall data from the confirmatory experiment. We also performed per-DBMS regression analysis on the final model to see how well our model fitted in each DBMS data.

For the Equations 10.1 and 10.2, we computed the overall fit of the model on the read and update groups using `lm()` in R. For the read group our model explained 14% of the variance of the dependent variable of DBMS thrashing. The explained variance was slightly increased by 3% compared with that of the exploratory evaluation (11%). This was a surprising result considering that thrashing measurement was more strict (as we increased the number of repetitions of a batch run from three to five), leading to a slight drop in the percentage of the thrashing samples. For the update group our model explained 15% of the variance of DBMS thrashing. The percentage was very similar to that of the exploratory evaluation (14%) on page 100 in Section 8.4. These results also showed that the identified empirical model for each group was supported on the confirmatory evaluation data.

We also examined the DBMS-specific amount of variance explained for the thrashing point on the read and update group, as illustrated in Figure 10.2. Regarding the read group shown in Figure 10.2(a) the highest amount of variance explained was 28% on MySQL, which was twice that of the overall (14%). The lowest amount of variance explained was 7% on DBMS Y, which was half that of the overall (14%). We could not determine the amount of variance explained by the model for PostgreSQL, because 1) initially too few thrashing samples were collected, 2) fewer were retained throughout TAP and then used for evaluation, and thus, 3) the p-value of the goodness-of-fit on the model was not significant. PostgreSQL appeared robust to thrashing on the read group in our experiment. Further investigations using more thrashing samples from PostgreSQL need to be conducted in future work.

Concerning the update group shown in Figure 10.2(b) the highest amount of variance explained was 31% on DBMS Y, which was twice more than that of the overall (15%). The lowest amount of variance explained was 3% on MySQL, which was about one fifth compared with that of the overall (15%). We could not determine the amount of variance explained on DBMS X, either, for the same reason as described above. DBMS X did not thrash on the update group in our experiment. For further examinations more thrashing samples from DBMS X need to be collected and evaluated in the future work.



(a) The Read Group (b) The Update Group

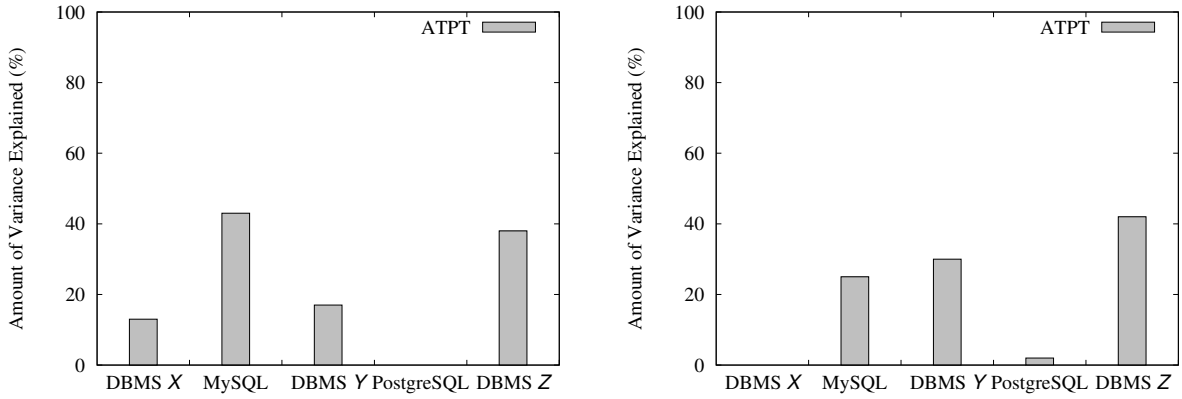
Figure 10.2: Per-DBMS Amount of Variance Explained for Thrashing on the Confirmatory Evaluation Data

Overall, the goodness-of-fit of the identified model varied across DBMSes, as the model of each group tended to fit better in some DBMSes than in others. Recall that given the same batchset some DBMSes thrashed while others did not. We see that this different reaction to the presence of thrashing across the DBMSes not only yielded such a difference in the degree of fit on the common model but also made it difficult to gain a higher percentage of explained variance on the overall data. For the DBMSes showing the low variance explained, more thrashing samples need to be collected from those DBMSes and evaluated in each group.

We also performed regression analysis on the dependent variable of ATPT. Our model explained 29% of the variance accounting for ATPT on the read group. The amount of variance explained by the model increased by more than twice, compared to that of the exploratory experiment (12%). The removed variables and associations did not reduce the amount of variance explained in the confirmatory experiment. We found that most of the DBMSes agreed with the direction of the correlations of numProcs and PK with ATPT. We feel that such an agreement contributed to the increase of the amount of variance explained for ATPT.

In contrast, the amount of variance explained by the model for the update group was 8%, which decreased from 11% of the exploratory evaluation. We attribute the decrease to the variables and associations removed from the revised model. (For a check, we computed the variance explained for the revised model in Figure 7.1 with the confirmatory evaluation data, and the explained variance was 12%.)

We also calculated the per-DBMS variance explained for ATPT on the read and update groups, as illustrated in Figure 10.3. Concerning the read group shown in Figure 10.2(a) the highest amount of variance explained was 43% for MySQL, significantly greater than the overall explained variance (29%). The lowest amount of variance explained was 13% for DBMS X, less than half of the overall explained variance (29%).



(a) The Read Group (b) The Update Group

Figure 10.3: Per-DBMS Amount of Variance Explained for Average Transaction Processing Time on the Confirmatory Evaluation Data

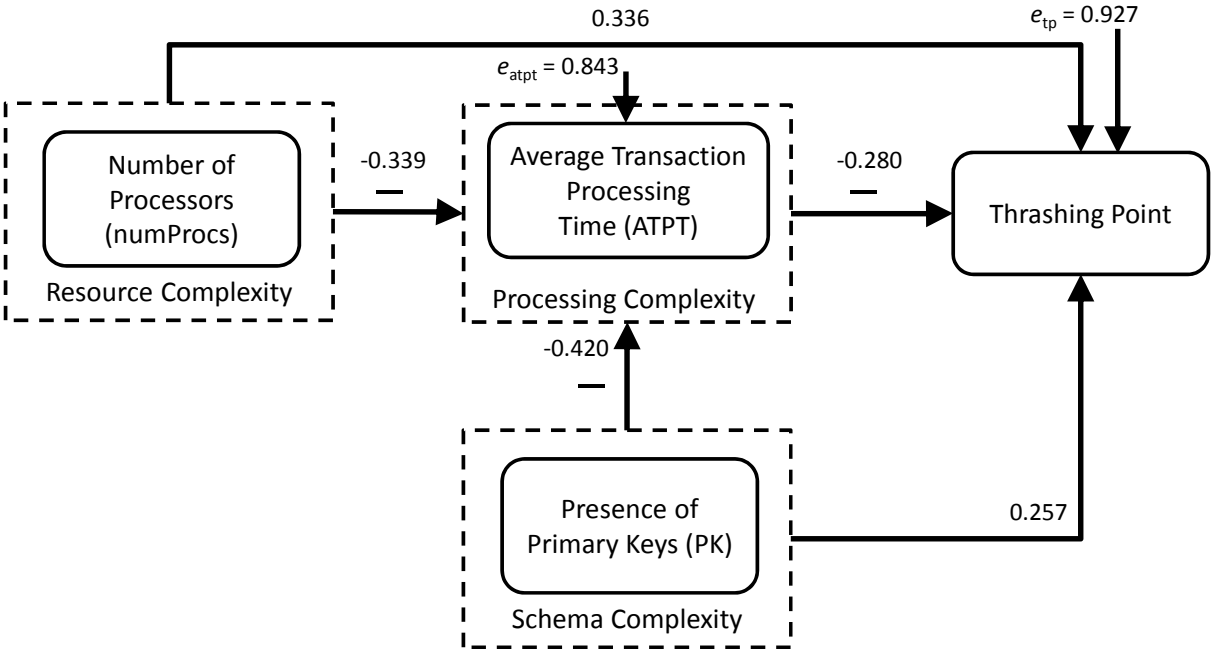
Regarding the update group shown in Figure 10.3(b) the highest amount of variance explained was 42% for DBMS Z, about five times greater than the overall explained variance (8%). The lowest amount of variance explained was 3% for PostgreSQL, less than half of the overall explained variance (8%). For the same reason as mentioned in Figure 10.2 on page 123, we could not obtain the explained variance for ATPT in the read and update groups on PostgreSQL and DBMS X, respectively. In the future work further investigations appear to be conducted with more samples from these two DBMSes.

Overall, the identified model for ATPT on each group also appeared to better fit the data from a subset of the DBMSes but not as much for the DBMSes taken together as a group. We explore possible elaborations to our model in Chapter 12.

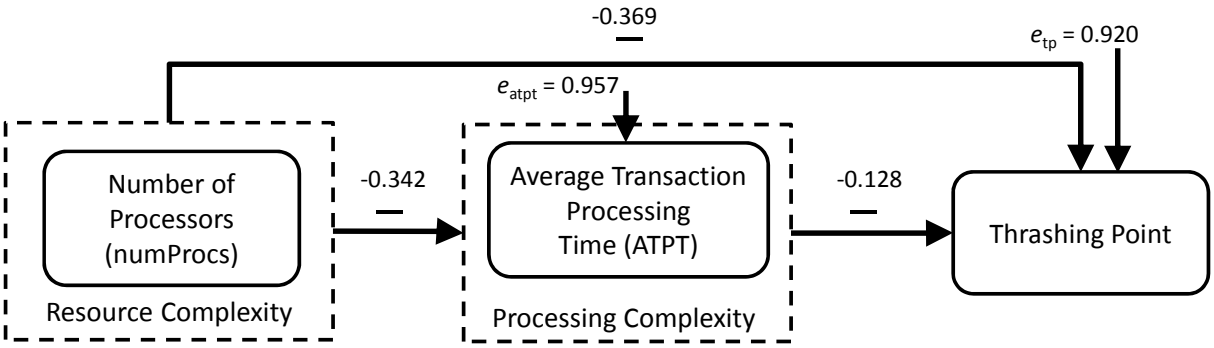
10.5 Path Analysis

We also conducted path analysis on the final model in Figure 9.1. Figure 10.4 shows the path diagrams on the final model. We used the path coefficients from the regression outputs of the read and update model fits, as provided in the paragraph of regression analysis in Section A.4.2.

Regarding the read group, all the paths were significant as shown in Figure 10.4(a). The direct effect (-0.420) of PK to ATPT was stronger than that (-0.339) of numProcs to ATPT. In addition, the total (indirect + direct) effect (-0.339 $\times$ (-0.280) + 0.336 = 0.430) of numProcs on thrashing point was stronger than that (-0.420 $\times$ (-0.280) + 0.257 = 0.375) of PK and that (-0.280) of ATPT on thrashing point.



(a) The Read Path Model



(b) The Update Path Model

Figure 10.4: Path Diagrams of the Batchset Groups in the Confirmatory Experiment

In the read model the most significant factors on ATPT and thrashing point were PK and numProcs, respectively. Note that in the exploratory evaluation the significance of numProcs was not as big as that of PK on thrashing point. One more value (or, six processors) added in the

operationalization of numProcs, perhaps, might have contributed to the increased significance in the confirmatory data. More investigations about the increase should be done in future work. That said, we reaffirm that numProcs and PK had positive correlations with thrashing point while ATPT had a negative correlation with thrashing point, as described in Section 8.5.

e_{atpt} and e_{tp} were decreased compared to those of the reduced model in Figure 8.2(b). The decreases of e_{atpt} and e_{tp} were attributed to the increased R-squared values (on the regressions on ATPT and thrashing point) from 0.14 to 0.29 and from 0.11 to 0.14, respectively.

Concerning the update model all the paths were significant, as illustrated in Figure 10.4(b). The total effect (-0.342 $\times$ (-0.128) - 0.369 = -0.325) of numProcs to thrashing point was stronger than that (-0.128) of ATPT to thrashing point. In the update model, the most significant factor on ATPT and on thrashing point was numProcs. We reaffirm that numProcs and ATPT had negative correlations with thrashing point in our model, as discussed in Section 8.5.

In Figure 10.4(b) e_{atpt} was increased, but e_{tp} was decreased compared to those of the reduced model in Figure 8.3(b). The increase of e_{atpt} resulted from the decrease of the R-squared value (on the regression of ATPT) to 0.08 from 0.14 of the reduced exploratory model. The decrease of e_{tp} was attributed to the increase of the R-squared value (on the regression of thrashing point) to 0.14 from 0.11 of the reduced exploratory model.

The fits of the read and update path models could be estimated as follows:

$$\begin{aligned} \text{Fit for the read path model} &= 1 - e_{atpt}^2 \times e_{tp}^2 = 1 - 0.843^2 \cdot 0.927^2 = 0.3893 \\ \text{Fit for the update path model} &= 1 - e_{atpt}^2 \times e_{tp}^2 = 1 - 0.957^2 \cdot 0.920^2 = 0.2248. \end{aligned} \tag{10.3}$$

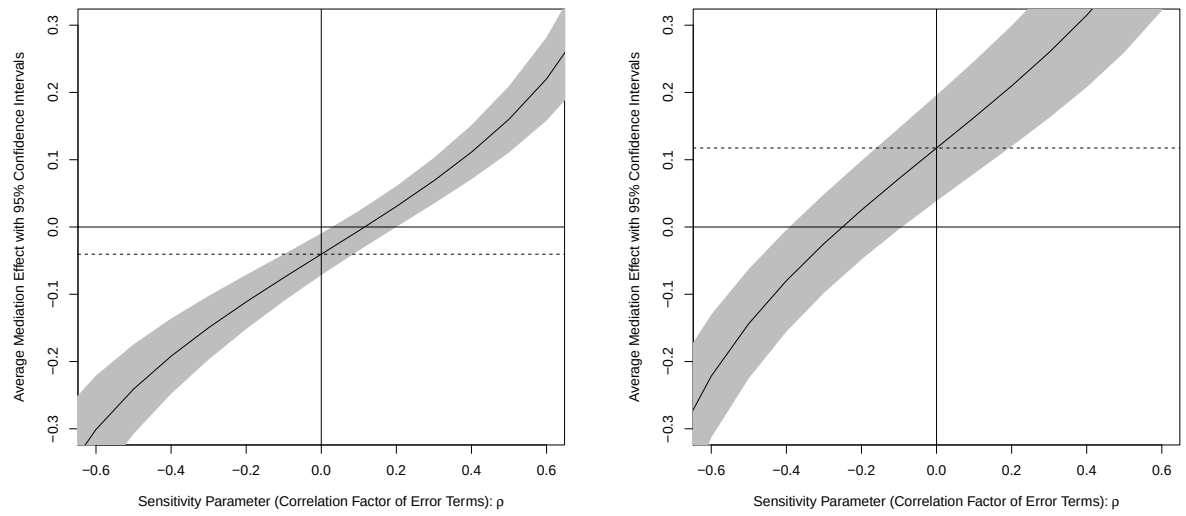
These estimated fits (0.3893 and 0.2248) in the read and update models were slightly increased, compared to those (0.2293 and 0.2164) of the reduced models in Equations 8.2 and 8.5, respectively. The increased fits resulted from the above improved R-squared (on the regressions of ATPT and thrashing point).

10.6 Causal Mediation Analysis

We also conducted the mediation analysis on the model, via `mediate()` in the `mediation` package in R, as was done with the exploratory evaluation. For the read group, we reaffirmed the

statistical significance of the mediating effects of PK and numProcs via ATPT on the thrashing point. For the update group, we also verified the statistical significance of the mediating effect of numProcs via ATPT on the thrashing point. These results show that the mediation via ATPT on the thrashing point was empirically verified.

We also conducted sensitivity analysis on the fitted models of ATPT and thrashing point for the read and update groups. Figure 10.5 shows the results for the read group's SA based on varying values of the sensitivity parameter ρ, or the correlation of residuals between ATPT and thrashing point in Equation 10.1.



(a) Sensitivity Analysis with Respect to Error Correlation between the Mediator of the Number of Processors and Outcome Models

(b) Sensitivity Analysis with Respect to Error Correlation between the Mediator of the Presence of Primary Keys and Outcome Models

Figure 10.5: Sensitivity Analysis for the Read Batchset Group in the Confirmatory Experiment

Figure 10.5(a) indicates the results for SA for examining whether there exists the mediating effect of ATPT from numProcs on the thrashing point. As can be seen in the figure, the corresponding AME was about -0.04, which was small and negative. When ρ was in [0.1, 0.2], the estimated AME was zero within the 95% CI. Note that a large value of ρ indicates the existence of strong confounding between the mediator and the outcome, and thus it is a serious violation of the SI assumption [33].

Figure 10.5(b) indicates the results for SA for examining whether there exists the mediating effect of ATPT from PK on the thrashing point. As can be seen in the figure, the corresponding AME was about 0.12, which was small and positive, too. When ρ was also in [-0.3, -0.2], the

estimated AME was zero within the 95% CI. Based on these results testing the SI assumption on the fitted models was satisfied. Therefore, the mediations via ATPT from numProcs and PK to the thrashing point in the read group was confirmed.

Figure 10.6 presents the results for the update group's SA based on varying values of the sensitivity parameter ρ, or the correlation of residuals between ATPT and thrashing point in Equation 10.2. The SA result examines whether there exists the mediating effect of ATPT from numProcs on the thrashing point. As can be seen in the figure, the corresponding AME was about 0.06, which was small and positive. When ρ was [-0.2, -0.1], the estimated AME was zero. Testing the SI assumption on the fitted models was satisfied. Thus, the mediations via ATPT from numProcs to the thrashing point in the update group was confirmed.

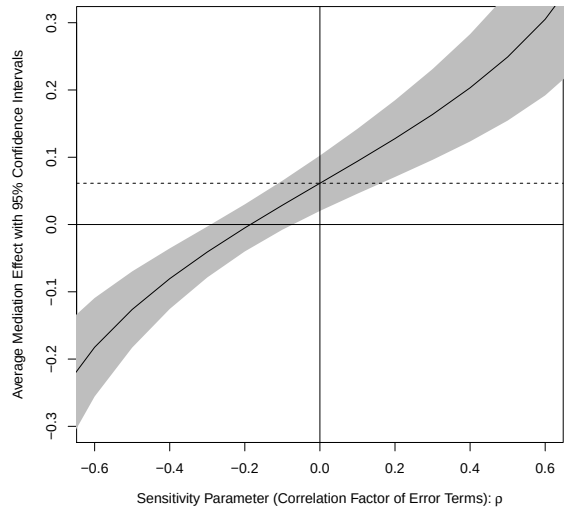


Figure 10.6: Sensitivity Analysis for the Update Batchset Group in the Confirmatory Experiment

Summary: All the confirmatory evaluation results provide empirical support for the very first structural causal model of DBMS thrashing that we have identified in this dissertation.

The following chapter will draw engineering implications from the confirmed structural causal model of DBMS thrashing.

CHAPTER 11

ENGINEERING IMPLICATIONS

While developing the final model through a series of large-scale experiments managed by AZDBLAB over about a year, we uncovered several surprising results that provide system-context indications as to how modern transaction processing can be further improved.

Here are some of the overall lessons we learned from this DBMS thrashing study. At the end of each lesson, we marked which one was known before and reaffirmed by further evidence (*reaffirmed*) or which one was not known before and thus new (*new*) in this thesis work.

- Every DBMS used in our experiment exhibited thrashing (new). Previous studies [28, 37] pointed out that some open-source DBMSes such as PostgreSQL and MySQL experienced thrashing. However, it was surprising to see that some proprietary DBMSes (X, Y, and Z) were also vulnerable to thrashing.

- Some DBMSes showed extensive thrashing at a certain point while increasing MPLs (reaffirmed). While this phenomenon was studied in much prior work [28, 35, 37, 46, 55] on some complex workloads, we showed that it was present even in simple workloads—read-only or update-only transactions—including select and projection statements only.

- Furthermore, some DBMSes even revealed early thrashing: under an MPL of 300 (new).

- Lastly, thrashing was not repeatable in DBMSes (new). The unrepeatability of DBMS thrashing is twofold. The first is *intra-unrepeatability* such that given the same batchset thrashing was not always observed in DBMSes; for instance, for the same batchset the DBMSes revealed thrashing in the exploratory experiment but not in the confirmatory experiment, and vice versa. The second is *inter-unrepeatability* such that although the same batchset was presented to DBMSes, some DBMSes experienced thrashing while others not.

These four results reveal that we need further the research on the thrashing topic in conjunction with the new era of multicore architecture. Fortunately, the causal model in Figure 9.1 on page 112 helps point out specifically where that research should be further focused.

Our model also conveys the following engineering implications for DBAs and database researchers, for better understanding and properly coping with thrashing.

1. When treating read-only workloads, we advise that

   - DBAs should enable multiprocessors, so that DBMSes can utilize their parallel processing ability when serving the workloads. In our experiment, DBMSes showed that they could increase thrashing point as more processors were available. DBAs thus may consider exposing to DBMSes as many processors as their systems can support.
   - DBAs should continue to utilize primary keys whenever possible. We have observed in our experiment that although workloads were increased, DBMSes were still scalable, resulting from their primary index made available by primary keys. We thus can infer that reducing I/O overhead can be a critical factor in preventing transaction throughput from falling off at a lower MPL. DBAs therefore may consider several alternatives for improving I/O: increasing physical memory, creating secondary indexes, and enabling disk drive and file system caches.
   - DBAs should pay attention to the response time of transactions. Another key factor in increasing thrashing point was to shorten transaction response time. In our experiment, we have observed that the thrashing point decreases as ATPT increases. When DBMS thrashing occurs, DBAs should check if the response time of transactions in their workloads sharply rose at a certain point. To reduce the response time, we prefer as many processors as possible and consider specifying primary keys on the tables referenced by the transactions, as addressed above. We challenge database researchers to identify other ways of improving the response time.
   - DBAs should be aware that the thrashing phenomenon tends to be DBMS-specific. As mentioned before, we observed that even if the same batchset was presented to all the DBMSes under the same condition, a certain DBMS was encountered with thrashing whereas another didn't even see thrashing. This implies that DBAs should check their specific transaction management and tuning parameters like the maximum number of connections and size of shared buffer.

2. When treating update-only workloads, we advise that

   - DBAs should be aware that using many processors may not help increase the thrashing point. We observed in our experiment that as the number of processors increased, the thrashing point fell, which was different from what was observed in the read-only transactions. When treating concurrent update-only transactions, DBMSes are charged with substantial lock management overhead on the same objects referenced by the transactions. Given multiprocessors, it seems that DBMSes struggle to tolerate the inter-processor contention on the synchronization constructs related to exclusive lock management. DBAs thus should be alert to increasing the number of processors for serving update-only workloads.
   - When thrashing is observed, DBAs need to examine whether the response time of transactions sharply went up at a certain point. As in the case of the read-only workloads, we observed that reducing ATPT could help increase the thrashing point in update-only workloads as well. Although we did not see the significance of primary keys on ATPT in our data, we still think that I/O could be one of the influential factors in decreasing response time. As far as the significance of I/O to reduce response time in update-only transactions is concerned, more research is needed.
   - Increasing the number of processors may improve ATPT. We observed in our experiment that DBMSes could speed up processing the update-only transactions as the number of processors increased. It seems that multiprocessors may enable intra-parallelism within the same transaction, contributing to decreasing response time. However, it was not helpful to have more processors in increasing the thrashing point, as mentioned in the first bullet above. DBAs therefore needs to be careful to make more processors available to DBMSes to reduce transaction response time. We challenge database researchers to further examine what other alternatives (including I/O) can improve response time, resulting in increasing the thrashing point.

Last but not least, we feel that there exist many other factors affecting DBMS thrashing present in read-only and update-only workloads, considering that the amount of variance explained by our model admits other significant factors. This thrashing research can be expanded by identifying such factors, including such factors in the model, and investigating their statistical significance on thrashing.

CHAPTER 12

CONCLUSIONS AND FUTURE WORK

This dissertation research has explored many factors of thrashing present in modern relational DBMSes. It has presented an evolution of our structural causal model of explicating the origins of thrashing. We have proposed a refined model of thrashing, which we have evaluated with actual data obtained from various experiments manipulating the operationalized factors.

The final model of the read group has explained about 14% and 28% of the variances of thrashing point and average transaction processing time (ATPT), respectively. Thrashing point has statistically significant correlations with ATPT, numProcs, and PK, among which numProcs is the most significant to thrashing point. ATPT has significant correlations with numProcs and PK, and PK is more significant to ATPT. The model also reveals significant mediations from numProcs and PK through ATPT to thrashing point.

The final model of the update group has explained about 15% and 8% of the variances of thrashing point and and ATPT, respectively. Thrashing point has statistically significant correlations with numProcs and ATPT, and numProcs is more significant to thrashing point. ATPT has a significant correlation with numProcs, the only significant factor to ATPT in the model. The model shows a significant mediation from numProcs through ATPT to thrashing point as well.

The final model also has drawn several engineering implications that can be helpful to DBAs and can motivate database researchers. One important implication behind the model is that transaction response time is one of the most significant factors of thrashing observed in our data. The model advises that when thrashing is detected, DBAs need to examine whether the response time of transactions in their workloads suddenly went up at a certain point. The model also suggests that decreasing the response time can increase thrashing point. To reduce the response time of read-only workloads, the DBAs should consider increasing the number of processors and specifying a primary key for every table. To speed up the processing time of update-only workloads, the DBAs may consider using increasing the number of processors. As our data reveal that increasing the number of processors has a negative correlation with the thrashing point,

however, database researchers need to further explore other ways of decreasing the response time of the update-only workloads.

To the best of our knowledge, our model is the first to be identified based on the empirical data from multiple relational DBMSes. Our model leaves room for elaboration, implying that there still exist many other unknown origins of DBMS thrashing.

In future work this dissertation leaves room for (i) investigating bumps that appeared after thrashing points in the five DBMSes, (ii) considering different types—compute-bound, mixed, nested, multi-level, chained, queued, and distributed—of transactions including short transactions (that were tried but did not work) as described in Appendix A.5, (iii) finding ways of operationalizing the excluded variables—buffer space and physical memory, (iv) adding more values per variable for our operationalization, (v) refining the operationalization (described in Section 3.4.3) and re-examining the significance of ROSE and SF to thrashing point and ATPT in the read and update groups, (vi) using logistic regression or structural equation modeling, (vii) re-examining the correlation between PK and thrashing point in the update group, (viii) examining the increased error variance of ATPT in the reduced model of the exploratory evaluation, (ix) inspecting moderated mediations that were not statistically significant, and (x) collecting more sample from the DBMSes.

Moreover, the database community can propose refinements to the final model to further improve its explanatory power, by (i) studying the effects of a more variety of variables: different isolation levels [20] and different types of indexes, (ii) employing a more variety of DBMS subjects, (iii) exploring unknown relationships between the variables in the final model, (iv) looking into the impact of a variety of predicates involving joins, IN, and aggregates beyond simple range scans, and (v) taking into account other statistical tools such as nonlinear regression [1] for the model.

Last but not least, our model has limited explanatory power since it has been empirically tested with only thrashing samples. Our model can thus predict how much DBMSes can tolerate thrashing for a given batchset. In the subsequent work it will also be interesting to expand or redesign our model by considering non-thrashing and thrashing samples together, so that the future model can predict whether given a batchset DBMSes thrashes or not.

APPENDIX A

In this appendix we present conceptual and logical design of the schema used for storing our data into a lab shelf. We also provide relevant notes associated with the conceptual-to-logical design mappings. We then include R outputs associated with the statistical analysis results presented in Chapters 8 and 10. We lastly provide the implementation of our scenario for observing thrashing in our experiments.

A.1 Common Lab Shelf Schema

In this section we discuss the conceptual and logical design of the common lab shelf schema of AZDBLAB.

A.1.1 Conceptual Design

AZDBLAB can keep track of all the provenance of the data related to an experiment. AZDBLAB uses common lab shelf schema, which is designed to store experiment-wide provenance data. The common schema has evolved as diverse experiments have been done in AZDBLAB. For storing the provenance specific to the data of an experiment, concrete schema can be designed and linked to the common schema. The concrete schema used in this dissertation will be discussed in Section A.2.

Figure A.1 depicts the common entity-relationship (ER) lab shelf schema provided in an internal technical guide [73].

Here are the descriptions of notations used in the schema. A rectangle indicates an entity type. A double-lined rectangle means a weak entity type that is dependent on another entity type. A diamond means a relationship between two entity types. A double-lined diamond also indicates a weak relationship type involving a weak entity type. A solid ellipse attached to an entity type represents an attribute.

A dotted ellipse is an optional attribute to that entity type. A single-underlined attribute is the key of an entity type, and a double-underlined attribute is the key of a weak entity type.

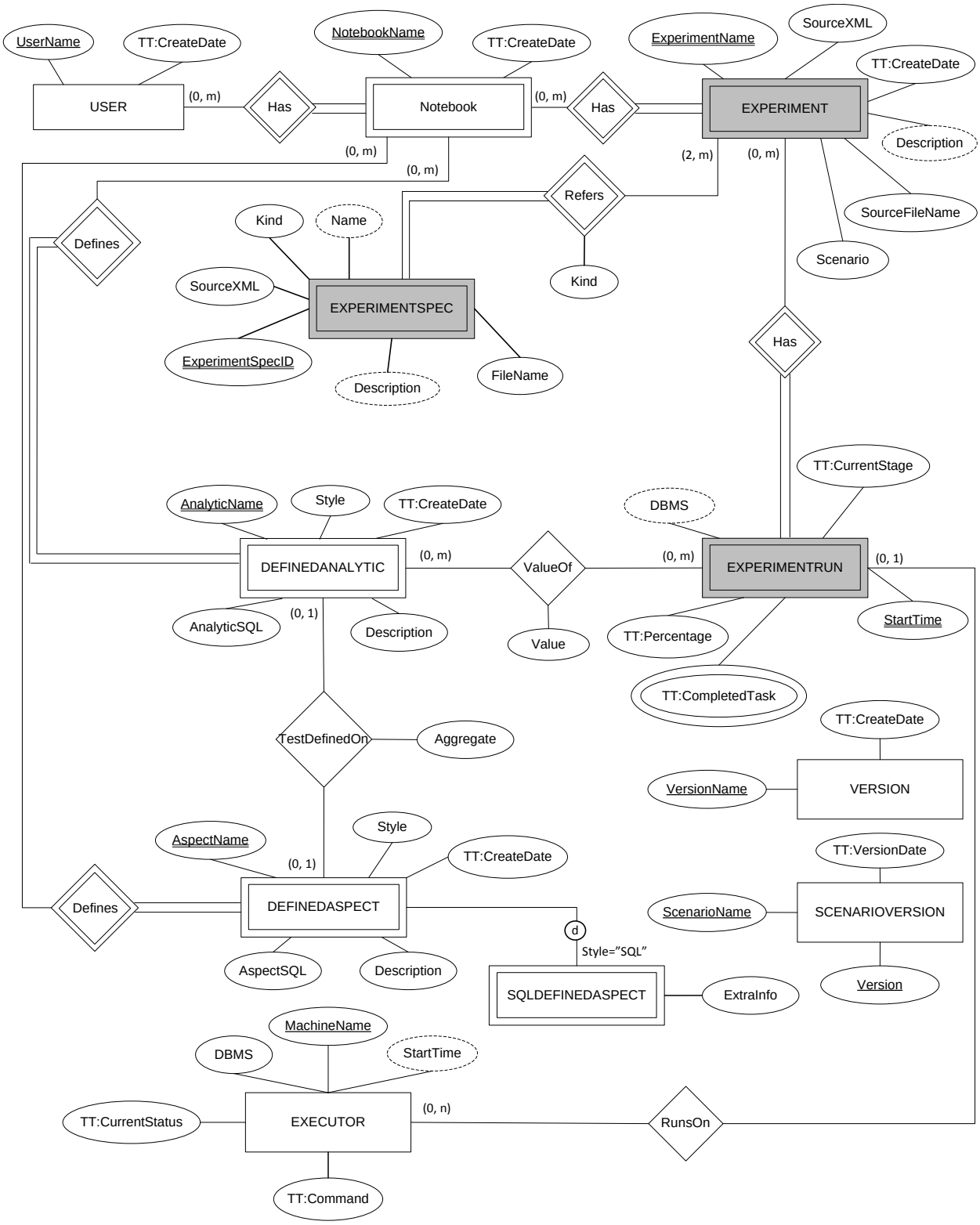


Figure A.1: Common Entity-Relationship Schema of AZDBLAB

A parenthesized annotation means cardinalities between entity types. For instance, '(0, m)' refers to that zero or more (week) entities concern *m* entities.

The notation of "TT:" means transaction time [69]. ⓓ indicates a component of an entity type. For example, the "SQLDEFINEDASPECT entity type is the component of the "DEFINEDASPECT" entity type.

Note that the shaded entity types—Experiment, ExperimentSpec, and ExperimentRun—are connected to other entity types defined in the concrete schema. Each entity type refers to each node shown in Figure 5.2. For instance, the USER entity type represents a lab shelf user ('yksuh'), the NOTEBOOK entity type indicates a notebook ('VLDB2014'), etc.

Transaction Time Attributes: There are few attributes whose value history is retained. EXPERIMENTRUN and EXECUTOR have transaction time attributes (denoted "TT:"); those two attributes are the only time-varying attributes in the entire schema. None of the entity types or relationship types are time-varying. The USER, NOTEBOOK, EXPERIMENT, EXPERIMENT, EXPERIMENTRUN, DEFINEDASPECT and DEFINEDANALYTIC entity types can be vacuumed by explicitly deleting them (which cascade delete other related entity types and relationship types); hence their (transaction time) existence time is a single contiguous period terminate at "Now." The beginning time of their existence period is denoted with CreateDate in all these entity types. EXECUTOR.StartTime is a derived attribute which is the most recent transaction time when EXECUTOR.CurrentStatus is equal to Running.

Derived Attributes: The following are the derived attributes. EXPERIMENT.Description is extracted from EXPERIMENT.Source. EXPERIMENT.Source.ReadDate is the date and time that the source of the experiment is read. This is identical to the CreateDate in EXPERIMENT. There are also derived VALUEOF relationship types between DEFINEDANALYTIC and EXPERIMENTRUN, EXPERIMENT, NOTEBOOK, and USER. As these computation are fast, they are done on demand, and thus not materialized.

The next section provides the details of the logical design and mapping notes of the common schema.

A.1.2 Logical Design

We map each ER diagram of the entities and relationships in the common schema to the corresponding logical tables. We give the schema for the resulting tables, in alphabetical order. The notes and rational for the design decisions follows.

Table A.1: The Logical Design of the Common Schema

Logical Tables	*DDL*
AZDBLAB_ANALYTICVALUEOF	CREATE TABLE AZDBLAB_ANALYTICVALUEOF(RUNID NUMBER(10) NOT NULL, ANALYTICID NUMBER(10) NOT NULL, ANALYTICVALUE VARCHAR2(100) NOT NULL, FOREIGN KEY (RUNID) REFERENCES AZDBLAB_EXPERIMENTRUN (RUNID) ON DELETE CASCADE, FOREIGN KEY (ANALYTICID) REFERENCES AZDBLAB_DEFINEDANALYTIC (ANALYTICID) ON DELETE CASCADE, PRIMARY KEY (RUNID, ANALYTICID));
AZDBLAB_COMPLETEDTASK	CREATE TABLE AZDBLAB_COMPLETEDTASK(RUNID NUMBER(10) NOT NULL, TASKNUMBER NUMBER(10) NOT NULL, TRANSACTIONTIME TIMESTAMP NOT NULL, PRIMARY KEY (RUNID, TASKNUMBER), FOREIGN KEY (RUNID) REFERENCES AZDBLAB_EXPERIMENTRUN (RUNID) ON DELETE CASCADE);
AZDBLAB_DEFINEDANALYTIC	CREATE TABLE AZDBLAB_DEFINEDANALYTIC(ANALYTICID NUMBER(10) NOT NULL PRIMARY KEY, USERNAME VARCHAR2(50) NOT NULL, NOTEBOOKNAME VARCHAR2(50), ANALYTICNAME VARCHAR2(100) NOT NULL, STYLE VARCHAR2(50) NOT NULL,

Continued on next page

Logical Tables	*DDL*
	CREATEDATE DATE NOT NULL, DESCRIPTION VARCHAR2(1000), ANALYTICSQL CLOB NOT NULL, FOREIGN KEY (USERNAME) REFERENCES AZDBLAB_USER(USERNAME) ON DELETE CASCADE, UNIQUE(USERNAME, ANALYTICNAME));
AZDBLAB_DEFINEDASPECT	CREATE TABLE AZDBLAB_DEFINEDASPECT(ASPECTID NUMBER(10) NOT NULL PRIMARY KEY, USERNAME VARCHAR2(50) NOT NULL, NOTEBOOKNAME VARCHAR2(50), ASPECTNAME VARCHAR2(100) NOT NULL, STYLE VARCHAR2(50) NOT NULL, DESCRIPTION VARCHAR2(1000), ASPECTSQL CLOB NOT NULL, FOREIGN KEY (USERNAME) REFERENCES AZDBLAB_USER(USERNAME) ON DELETE CASCADE, UNIQUE(USERNAME, ASPECTNAME));
AZDBLAB_EXECUTOR	CREATE TABLE AZDBLAB_EXECUTOR(MACHINENAME VARCHAR2(100) NOT NULL PRIMARY KEY, DBMS VARCHAR2(100) NOT NULL, CURRENTSTATUS VARCHAR2(20) NOT NULL, COMMAND VARCHAR2(20) NOT NULL);
AZDBLAB_EXECUTORLOG	CREATE TABLE AZDBLAB_EXECUTORLOG(MACHINENAME VARCHAR2(100) NOT NULL, TRANSACTIONTIME TIMESTAMP NOT NULL, CURRENTSTATUS VARCHAR2(100) NOT NULL, COMMAND VARCHAR2(20) NOT NULL,

Continued on next page

Logical Tables	*DDL*
	PRIMARY KEY (MACHINENAME, TRANSACTIONTIME));
AZDBLAB_EXPERIMENT	CREATE TABLE AZDBLAB_EXPERIMENT(EXPERIMENTID NUMBER(10) NOT NULL PRIMARY KEY, USERNAME VARCHAR2(100) NOT NULL, NOTEBOOKNAME VARCHAR2(100) NOT NULL, EXPERIMENTNAME VARCHAR2(100) NOT NULL, SCENARIO VARCHAR2(100) NOT NULL, SOURCEFILENAME VARCHAR2(100) NOT NULL, CREATEDATE DATE NOT NULL, SOURCEXML CLOB NOT NULL, UNIQUE(USERNAME, NOTEBOOKNAME, EXPERIMENTNAME), FOREIGN KEY(USERNAME, NOTEBOOKNAME) REFERENCES AZDBLAB_NOTEBOOK(USERNAME, NOTEBOOKNAME) ON DELETE CASCADE);
AZDBLAB_EXPERIMENTRUN	CREATE TABLE AZDBLAB_EXPERIMENTRUN(RUNID NUMBER(10) NOT NULL PRIMARY KEY, EXPERIMENTID NUMBER(10) NOT NULL, MACHINENAME VARCHAR2(100), DBMS VARCHAR2(100) NOT NULL, STARTTIME TIMESTAMP NOT NULL, CURRENTSTAGE VARCHAR2(1000) NOT NULL, PERCENTAGE NUMBER(10) NOT NULL, FOREIGN KEY (EXPERIMENTID) REFERENCES AZDBLAB_EXPERIMENT (EXPERIMENTID) ON DELETE CASCADE, FOREIGN KEY (MACHINENAME) REFERENCES AZDBLAB_EXECUTOR (MACHINENAME) ON DELETE CASCADE, UNIQUE(EXPERIMENTID, STARTTIME));

Continued on next page

Logical Tables	*DDL*
AZDBLAB_EXPERIMENTSPEC	CREATE TABLE AZDBLAB_EXPERIMENTSPEC(EXPERIMENTSPECID NUMBER(10) NOT NULL PRIMARY KEY, NAME VARCHAR2(100) NOT NULL, KIND VARCHAR2(1) NOT NULL, FILENAME VARCHAR2(100) NOT NULL, SOURCEXML CLOB NOT NULL);
AZDBLAB_NOTEBOOK	CREATE TABLE AZDBLAB_NOTEBOOK(USERNAME VARCHAR2(100) NOT NULL, NOTEBOOKNAME VARCHAR2(100) NOT NULL, CREATEDATE DATE NOT NULL, FOREIGN KEY (USERNAME) REFERENCES AZDBLAB_USER(USERNAME) ON DELETE CASCADE, PRIMARY KEY (USERNAME, NOTEBOOKNAME));
AZDBLAB_REFERSEXPERIMENTSPEC	CREATE TABLE AZDBLAB_REFERSEXPERIMENTSPEC(EXPERIMENTID NUMBER(10) NOT NULL, KIND VARCHAR2(1) NOT NULL, EXPERIMENTSPECID NUMBER(10) NOT NULL, FOREIGN KEY (EXPERIMENTID) REFERENCES AZDBLAB_EXPERIMENT (EXPERIMENTID) ON DELETE CASCADE, FOREIGN KEY (EXPERIMENTSPECID) REFERENCES AZDBLAB_EXPERIMENTSPEC (EXPERIMENTSPECID) ON DELETE CASCADE, PRIMARY KEY (EXPERIMENTID, KIND, EXPERIMENTSPECID));
AZDBLAB_RUNLOG	CREATE TABLE AZDBLAB_RUNLOG(RUNID NUMBER(10) NOT NULL, TRANSACTIONTIME TIMESTAMP NOT NULL,

Continued on next page

Logical Tables	*DDL*
	CURRENTSTAGE VARCHAR2(1000) NOT NULL, PERCENTAGE NUMBER(10) NOT NULL, FOREIGN KEY (RUNID) REFERENCES AZDBLAB_EXPERIMENTRUN (RUNID) ON DELETE CASCADE, PRIMARY KEY (RUNID, TRANSACTIONTIME));
AZDBLAB_SCENARIOVERSION	CREATE TABLE AZDBLAB_SCENARIOVERSIO(SCENARIONAME VARCHAR2(100) NOT NULL, VERSION VARCHAR2(100) NOT NULL, VERSIONDATE DATE NOT NULL, PRIMARY KEY (SCENARIONAME, VERSION));
AZDBLAB_USER	CREATE TABLE AZDBLAB_USER(USERNAME VARCHAR2(100) NOT NULL PRIMARY KEY, CREATEDATE DATE NOT NULL);
AZDBLAB_VERSION	CREATE TABLE AZDBLAB_VERSION(VERSIONNAME VARCHAR2(100) NOT NULL, CREATEDATE DATE NOT NULL, PRIMARY KEY (VERSIONNAME));

The following notes concern each relation mapped from a partial weak entity or relationship type.

All data items stored in a lab shelf are considered important experimental records, so they should be non-deletable by regular users. Although deletion options are provided for different hierarchies of entity types, including `USER`, `NOTEBOOK`, and `EXPERIMENT`. The deletion options are only available for the chief programmer. Deletions are provided with exporting options. Importing is provided to recover data from exported files. Invariant: export + delete + import = nothing.

All data items in `NOTEBOOK` should be append only, no changes are allowed once data are inserted. The only exceptions are in `EXPERIMENTRUN` and `EXECUTOR`, where the progress

information is updated periodically. But the changes in progress are logged into RUNLOG and EXECUTORLOG tables, respectively.

To facilitate primary and foreign keys and efficiency (space and time), surrogate (integer) keys are defined for EXPERIMENT, EXPERIMENTRUN, TEST, TESTSPEC, DEFINEDASPECT and DEFINEDANALYTIC .

ANALYSIS: Is a weak entity type of NOTEBOOK, so the resulting table has a foreign key (UserName, NotebookName) . There is a unique constraint (UserName, NotebookName, AnalysisName). The primary key is AnalysisID.

ANALYTICVALUEOF: A test may have zero or multiple analytics defined upon. Each row provides the value of the defined analytic of the test. Primary key and foreign keys are (TestID, AnalyticID). They are from TEST and DEFINEDANALYTIC respectively. There are also such relationship types between DEFINEDANALYTIC and all other tables on the weak entity type chain from USER down to EXPERIMENTRUN . All these relationship types are not materialized since they can all be efficiently generalized from the ANALYTICVALUEOF between TEST and DEFINEDANALYTIC .

COMPLETEDTASK: Is a weak entity type of ANALYSIS, so the resulting table has a foreign key RunID. This table contains the TaskNumber which indicates a task number and the TransactionTime which is the time to insert a record.

DEFINEDANALYTIC: Also a weak entity type of NOTEBOOK, so the resulting table has the same foreign key structure as DEFINEDASPECT for the same reason. The unique constraint is on (UserName, AnalyticName). The primary key is AnalyticID.

DEFINEDASPECT: A weak entity type of NOTEBOOK. However, it is allowed the NotebookName attribute to be empty, so physically, the foreign key UserName refers to UserName in USER table. There is a unique constraint on (UserName, AspectName) . The primary key is AspectID.

EXECUTOR: Command column allows three options: "T," "R," "N," and "P," representing Terminate, Resume, No Command, and Pause, respectively.

EXECUTORLOG: This is created for the very similar reason as RUNLOG. EXECUTORLOG is used to keep track of the execution status and command of Executors. The primary key is (MachineName, TransactionTime). The foreign key is MachineName, referring to EXECUTOR. StartTime attribute is not stored; it is computed on demand.

EXPERIMENT: A weak entity type of NOTEBOOK, so the resulting table has foreign keys (UserName, NotebookName). There is a unique constraint on (UserName, NotebookName, ExperimentName). The primary key is ExperimentID. It is a surrogate key which is not derived from the data columns. The strategy is to utilize surrogate key to avoid complex string keys such as UserName, NotebookName and ExperimentName. This provides both space and time efficiencies. Description can be derived from the Source, so there is no need to materialize this content.

EXPERIMENTRUN: A weak entity type of EXPERIMENT, so the resulting table has a foreign key ExperimentID. The primary key is RunID, which also a surrogate key chosen according to the same strategy stated in EXPERIENT. Although DBMS can be derived from Source in EXPERIMENT, for the sake of convenience, DBMS names are stored here.

Logically, each experiment run instance falls into one of four sub-categories: PendingRun, RunningRun, PausedRun, and AbortedRun. These four categories can be differentiated by the content of CurrentStage and Percentage columns. PendingRun, CompletedRun, and AbortedRun can be identified by CurrentStage with value "Pending," "Completed," and "Aborted," respectively. RunningRun is determined by not taking the value "Pending," "Completed," and "Aborted" for CurrentStage. Percentage value lies in between 0 to 100. MachineName is NULL when the CurrentStage is "Pending."

EXPERIMENTSPEC: A weak entity type of EXPERIMENT, so the resulting table has a foreign key ExperimentID. The primary key is ExperimentSpecID. Kind column allows two options: "D" and "Q," representing DataDefinitionSpec and QuerySpec, respectively.

NOTEBOOK: A weak entity type of `USER`, so the resulting table has a foreign key `UserName`.

REFERSTESTSPEC: One test may refer to zero to four definition types. It is four if all four of the `TESTSPEC`s kind are not present in the experiment specification file. A particular `TESTSPEC` item might be referred by multiple tests. The primary key is (`TestID`, `TestSpecID`) and foreign keys are (`TestID`, `TestSpecID`) as well, from `TEST` and `TESTSPEC`, respectively.

RUNLOG: A weak entity type of `EXPERIMENTRUN`, so the resulting table has a foreign key `RunID`. Primary key is (`RunID`, `StartTime`). `RUNLOG` is induced by the `TransactionTime` attributes: `CurrentStage` and `Percentage`. The `TransactionTime` column record the transaction time whenever a progress update is coming from Executor(s). And the most recently data item will be updated to `EXPERIMENTRUN` table.

RUNSON: This is represented with a `MachineName` attribute in `EXPERIMENTRUN` as a foreign key to `EXECUTOR`.

SATISFIESASPECT relationship type: A query may satisfy either zero or multiple aspects. Each tuple in this relationship type represents the fact that a query, defined by `QueryID`, satisfies an aspect, defined by `AspectID`. Primary key and foreign keys are (`QueryID`, `AspectID`). They are from `QUERY` and `DEFINEDASPECT` respectively.

SCENARIOVERSION: Same as its ER.

USER: : Same as its ER.

VERSION: Same as its ER.

A.2 Batchset Lab Shelf Schema

In our experiment we need to collect the values of the independent and dependent variables discussed in Section 3.4. We thus have defined concrete lab shelf schema, called “batchset” schema

extending the common schema. In this section we describe in detail conceptual and logical design of the batchset schema.

A.2.1 Conceptual Design

The batchset schema specifies the structure of an experiment. It not only records the values of the variables but also captures detailed experiment provenance related to batchset execution data. Based on these data we calculate and store into this schema a thrashing point for each batchset.

Figure A.2 depicts the batchset conceptual schema. The EXPERIMENTSPEC entity type, originating from the common schema in Figure A.1, is composed of the TABLECONFIG, DATADEFINITION, and BATCHSETDEF entity types. TABLECONFIG captures a experiment table schema specification, which contains table names and random seed number for each table population. DATADEFINITION captures a table population specification, which contains the following information: table prefix, table names, the number of tables, the number of columns in a table, column names, column types, column lengths, minimum (default) and maximum table cardinality, and column value generation method (sequential or random). BATCHSETCONFIG captures batchset configuration, containing a set of specified values of the independent variables. Based on these specified values we generate a variety of batchsets used in our experiments.

The shaded EXPERIMENT entity type is in the Has relationship with the BATCHSET entity type, which represents a batchset.

BatchSetNumber is an attribute to record a sequential number assigned to each batchset generated from an experiment specification. It is the partial key of the BATCHSET entity type.

The following four attributes are extracted from EXPERIMENT.Source, and accordingly they can be set. ReadSelFactor and UpdateSelFactor are attributes to record the respective SF of a read-only and update-only transaction. ROSE is an attribute to set the specific value (in percentage) of ROSE. BatchSzIncr is the attribute for representing the batch granularity, which is the size difference between two adjacent batches in a batchset. We use the constant value (100) for the attribute.

The tertiary Has relationship type involves the EXPERIMENTRUN, BATCHSET and BATCHSETRUNRESULT entity types. BATCHSETRUNRESULT has the BatchSetRunNumber attribute representing a batchset run number. It is the partial key of BATCHSETRUNRESULT.

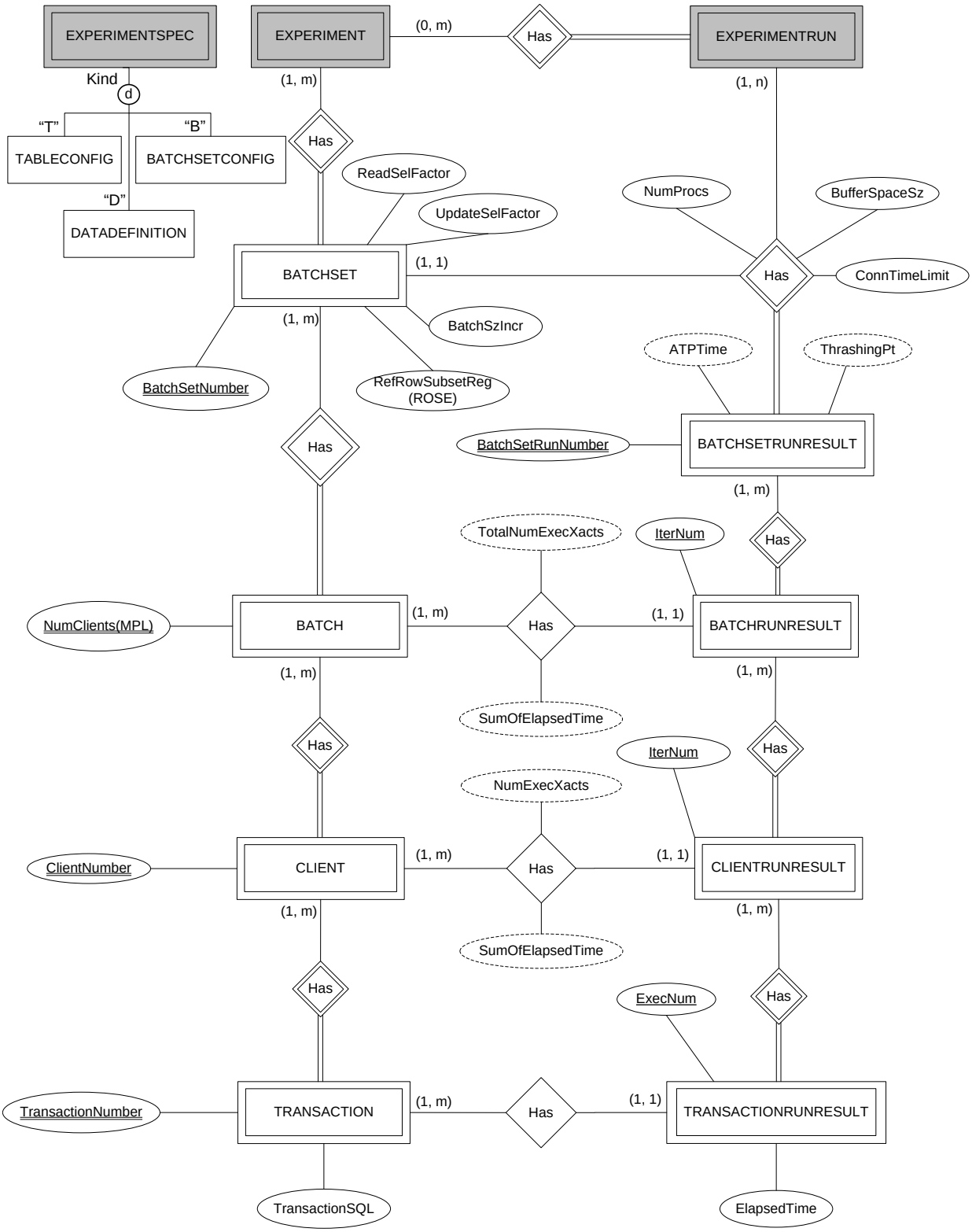


Figure A.2: Batchset Entity-Relationship Schema

The Has relationship type has the five attributes, among which three attributes can be set, and the other two attributes are derived. NumProcs is an attribute to store the number of processors configured for an experiment run. The BufferSpaceSz attribute indicates the amount of DBMS buffer space for the run; we store this number as percentage. ConnTimeLimit is an attribute to store the value of CTL given from an experiment specification. These three attributes are extracted from EXPERIMENT.Source. The two derived attributes of Has are ATPTime and ThrashingPt, which store the values of ATP time and the thrashing point calculated over a batchset run, respectively. The calculation is done based on some of the other derived attributes, to be described shortly. In particular, the ThrashingPt attribute captures the dependent variable of our DBMS thrashing study.

There exists the Has relationship type between the BATCHSET and BATCH entity types. BATCH captures a batch in a batchset. BATCH has NumClients representing an MPL value. The NumClients attribute, which can be set, is the partial key of the BATCH entity type. If a batch is complete to run, the run result is kept by the BATCHRUNRESULT entity type, which has the IterNum attribute for storing the current iteration (repetition) number. The IterNum attribute is the partial key of BATCHRUNRESULT.

There exists the Has relationship type between the BATCH and BATCHRUNRESULT entity types. The Has relationship type has the following two derived attributes: TotalNumExecXacts for storing the total number of the transactions executed by a batch run and SumOfElapsedTime for storing the sum of the measured elapsed time of the transactions executed in the batch run. The values of TotalNumExecXacts and SumOfElapsedTime can be calculated based on some of the attributes to be described shortly. These two attributes are used for calculating the values of ATPTime and ThrashingPt in BATCHSET. Given a batchset, ATPTime can be derived by dividing the sum of all the values of SumOfElapsedTime by the sum of all the values of TotalNumExecXacts associated with the (ten) batches in the batchset. For calculating the value of ThrashingPt, we compute the value of TPS at each batch by dividing the value of TotalNumExecXacts of that batch by the value of ConnTimeLimit, and we then apply Equation 3.2 presented on page 55.

There is the Has relationship type between the BATCH and CLIENT entity types. CLIENT captures a client in a batch. The CLIENT entity type has ClientNumber representing the client's number in the batch. The ClientNumber attribute, which can be set, is the partial key of the

CLIENT entity type. The client's transaction execution result is kept by the CLIENTRUNRESULT entity type, which has the IterNum attribute for storing the current iteration number. IterNum, which can be set too, is the partial key of CLIENTRUNRESULT.

The Has relationship type is between the CLIENT and CLIENTRUNRESULT entity types. The following are the derived (observed) attributes of Has: NumExecXacts for storing the total number of the transactions run by this client, and SumOfElapsedTime for storing the sum of the elapsed times of the transactions. NumExecXacts and SumOfElapsedTime are used to compute the values of TotalNumExecXacts and SumOfElapsedTime in the previous Has relationship type, by summing up the values of NumExecXacts and SumOfElapsedTime of the clients in the batch.

There exists the Has relationship type between CLIENT and TRANSACTION. The TRANSACTION models a transaction to be executed by the client. TRANSACTION has TransactionNumber, representing the transaction's number in the client. TransactionNumber can be set. The TransactionNumber attribute is the partial key of the TRANSACTION entity type. TransactionSQL is an attribute to store an SQL statement(s) associated with a transaction. For each completed transaction, the transaction's run result is kept by the TRANSACTIONRUNRESULT entity type, which has the ExecNum attribute to specify the current execution number of this transaction. This ExecNum variable, which can be set, is the partial key of the TRANSACTIONRUNRESULT entity type, which has ElapsedTime is a derived attribute for storing the transaction's end-to-end time (millisecond) measured in Java. ElapsedTime is used to compute the value of SumOfElapsedTime attached to the Has relation entity type between CLIENT and CLIENTRUNRESULT. Between the TRANSACTION and TRANSACTIONRUNRESULT entity types is the Has relationship type.

The logical design of the batchset schema will be discussed in the next section.

A.2.2 Logical Design

Table A.2 represents the logical design of the batchset ER schema. We give the schema for the resulting tables in alphabetical order. The notes and rational for the design decisions follows.

Table A.2: The Logical Design of the Batchset Schema

Logical Tables	*DDL*
AZDBLAB_BATCH	CREATE TABLE AZDBLAB_BATCH(BatchID NUMBER(10) NOT NULL PRIMARY KEY, BatchSetID NUMBER(10) NOT NULL, MPL NUMBER(10) NOT NULL, UNIQUE (BatchSetID, MPL), FOREIGN KEY (BatchSetID) REFERENCES AZDBLAB_BATCHSET(BatchSetID) ON DELETE CASCADE);
AZDBLAB_BATCHHASRESULT	CREATE TABLE AZDBLAB_BATCHHASRESULT(BatchRunResID NUMBER(10) NOT NULL PRIMARY KEY, BatchSetRunResID NUMBER(10) NOT NULL, BatchID NUMBER(10) NOT NULL, IterNum NUMBER(10) NOT NULL, ElapsedTime NUMBER(10) NOT NULL, SumExecXacts NUMBER(10) NOT NULL, SumXactProcTime NUMBER(10) NOT NULL, UNIQUE (BatchSetRunResID, BatchID, IterNum), FOREIGN KEY (BatchSetRunResID) REFERENCES AZDBLAB_BATCHSETHASRESULT (BatchSetRunResID) ON DELETE CASCADE, FOREIGN KEY (BatchID) REFERENCES AZDBLAB_BATCH(BatchID) ON DELETE CASCADE);
AZDBLAB_BATCHSET	CREATE TABLE AZDBLAB_BATCHSET(BatchSetID NUMBER(10) NOT NULL PRIMARY KEY, ExperimentID NUMBER(10) NOT NULL, BatchSzIncr NUMBER(10) NOT NULL, ReadSF NUMBER(10,4) NOT NULL, UpdateSF NUMBER(10,2) NOT NULL,

Continued on next page

Logical Tables	*DDL*
	RosePct NUMBER(10,2) NOT NULL, UNIQUE (ExperimentID, BatchSzIncr, ReadSF, UpdateSF, RosePct), FOREIGN KEY (RUNID) REFERENCES AZDBLAB_EXPERIMENT(ExperimentID) ON DELETE CASCADE,);
AZDBLAB_BATCHSETHASRESULT	CREATE TABLE AZDBLAB_BATCHSETHASRESULT(BatchSetRunResID NUMBER(10) NOT NULL PRIMARY KEY, RunID NUMBER(10) NOT NULL, BatchSetID NUMBER(10) NOT NULL, NumProcs NUMBER(10) NOT NULL, BufferSpace NUMBER(10,2) NOT NULL, ConnTimeLimit NUMBER(10) NOT NULL, AvgXactProcTime NUMBER(10,2), ThrashingPt NUMBER(10), UNIQUE (BatchSetID, RunID), FOREIGN KEY (BatchSetID) REFERENCES AZDBLAB_BATCHSET(BatchSetID) ON DELETE CASCADE, FOREIGN KEY (RunID) REFERENCES AZDBLAB_EXPERIMENTRUN(RunID) ON DELETE CASCADE);
AZDBLAB_BSSATISFIESASPECT	CREATE TABLE AZDBLAB_BSSATISFIESASPECT(BatchSetID NUMBER(10) NOT NULL, AspectID NUMBER(10) NOT NULL, AspectValue NUMBER(10) NOT NULL, FOREIGN KEY (BatchSetID) REFERENCES AZDBLAB_BATCHSET(BatchSetID) ON DELETE CASCADE, FOREIGN KEY (AspectID) REFERENCES AZDBLAB_DEFINEDASPECT(AspectID) ON DELETE CASCADE, PRIMARY KEY (BatchSetID, AspectID)

Continued on next page

Logical Tables	*DDL*
	);
AZDBLAB_CLIENT	CREATE TABLE AZDBLAB_CLIENT(ClientID NUMBER(10) NOT NULL PRIMARY KEY, BatchID NUMBER(10) NOT NULL, ClientNum NUMBER(10) NOT NULL, UNIQUE (BatchID, ClientNum), FOREIGN KEY (BatchID) REFERENCES AZDBLAB_BATCH(BatchID) ON DELETE CASCADE);
AZDBLAB_CLIENTHASRESULT	CREATE TABLE AZDBLAB_CLIENTHASRESULT(ClientRunResID NUMBER(10) NOT NULL PRIMARY KEY, BatchRunResID NUMBER(10) NOT NULL, ClientID NUMBER(10) NOT NULL, IterNum NUMBER(10) NOT NULL, NumExecXacts NUMBER(10) NOT NULL, SumXactProcTime NUMBER(10) NOT NULL, UNIQUE (BatchRunResID, ClientID, IterNum), FOREIGN KEY (BatchRunResID) REFERENCES AZDBLAB_BATCHHASRESULT (BatchRunResID) ON DELETE CASCADE, FOREIGN KEY (ClientID) REFERENCES AZDBLAB_CLIENT(ClientID) ON DELETE CASCADE);
AZDBLAB_COMPLETEDBATCHSETTASK	CREATE TABLE AZDBLAB_COMPLETEDBATCHSETTASK(RunID NUMBER(10) NOT NULL, TaskNumber NUMBER(10) NOT NULL, TransactionTime NUMBER(10) NOT NULL, FOREIGN KEY (RunID) REFERENCES AZDBLAB_EXPERIMENTRUN(RunID) ON DELETE CASCADE,

Continued on next page

Logical Tables	*DDL*
	PRIMARY KEY (RunID, TaskNumber));
AZDBLAB_TRANSACTION	CREATE TABLE AZDBLAB_TRANSACTION(TransactionID NUMBER(10) NOT NULL PRIMARY KEY, ClientID NUMBER(10) NOT NULL, TransactionNum NUMBER(10) NOT NULL, TransactionStr VARCHAR(2000) NOT NULL, UNIQUE (ClientID, TransactionNum), FOREIGN KEY (ClientID) REFERENCES AZDBLAB_CLIENT(ClientID) ON DELETE CASCADE
AZDBLAB_TRANSACTIONHASRESULT	CREATE TABLE AZDBLAB_TRANSACTIONHASRESULT(TransactionRunResID NUMBER(10) NOT NULL PRIMARY KEY, ClientRunResID NUMBER(10) NOT NULL, TransactionID NUMBER(10) NOT NULL, NumExecs NUMBER(10) NOT NULL, MinXactProcTime NUMBER(10) NOT NULL, MaxXactProcTime NUMBER(10) NOT NULL, SumXactProcTime NUMBER(10) NOT NULL, SumLockWaitTime NUMBER(10) NOT NULL, UNIQUE (ClientRunResID, TransactionID), FOREIGN KEY (ClientRunResID) REFERENCES AZDBLAB_CLIENTHASRESULT (ClientRunResID) ON DELETE CASCADE, FOREIGN KEY (TransactionID) REFERENCES AZDBLAB_TRANSACTION (TransactionID) ON DELETE CASCADE);

The following notes are concerned about each relation, mapped from a partial weak entity type or relationship type.

All data items in each logical table should be append only, and no changes are allowed once data are inserted. The only exception exists in BATCHSETHASRESULT , where the column values of ATPTime and ThrashingPt are later computed based on the values in BATCHHASRESULT and then updated.

To facilitate primary and foreign keys and efficiency (space and time), surrogate (integer) keys are defined for BATCHSET, BATCHSETHASRESULT, BATCH, BATCHHASRESULT, CLIENT, CLIENTHASRESULT TRANSACTION, and TRANSACTIONHASRESULT.

BATCH: BATCH is a weak entity type of BATCHSET, so the resulting table has a foreign key (BatchSetID). There is a unique constraint (BatchSetID, MPL). The primary key is BatchID.

BATCHHASRESULT: BATCHHASRESULT is the logical design of BATCHRUNRESULT and Has in the ER schema. BATCHHASRESULT is a weak entity type of BATCHSETHASRESULT and BATCH, so the resulting table has two foreign keys (BatchSetRunResID, BatchSetID). There is a unique constraint (BatchSetRunResID, BatchID, IterNum). The primary key is BatchRunResID.

BATCHSET: BATCHSET is a weak entity type of EXPERIMENT, so the resulting table has a foreign key (ExperimentID). There is a unique constraint (ExperimentID, BatchSzIncr, ReadSF, UpdateSF, RosePct). The primary key is BatchSetID, equivalent to BatchSetNumber in the ER schema depicted in Figure A.2.

BATCHSETHASRESULT: BATCHSETHASRESULT is the logical design of BATCHSETRUNRESULT and Has in the ER schema. BATCHSETHASRESULT is a weak entity type of BATCHSET and EXPERIMENTRUN, so the resulting table has two foreign keys (BatchSetID, RunID). There is a unique constraint (BatchSetID, RunID). The primary key is BatchSetRunResID, equivalent to BatchSetRunNumber in the ER schema.

BSSATISFIESASPECT: BSSATISFIESASPECT is a weak entity type of BATCHSET and DEFINEDASPECT, so the resulting table has foreign keys (BatchSetID, AspectID). The primary key is BatchSetID and AspectID.

CLIENT: CLIENT is a weak entity type of BATCH, so the resulting table has a foreign key (BatchID). There is a unique constraint (BatchID, ClientNum). The primary key is ClientID.

CLIENTHASRESULT: CLIENTHASRESULT specifies the logical design of the CLIENTRUNRESULT entity, and the adjacent Has relationship type in the ER schema. CLIENTHASRESULT is a weak entity type of BATCHHASRESULT and CLIENT, so the resulting table has two foreign keys (BatchRunResID, ClientID). There is a unique constraint (BatchRunResID, ClientID, IterNum). The primary key is ClientRunResID.

COMPLETEDBATCHSETTASK: COMPLETEDBATCHSETTASK is a weak entity type of COMPLETEDBATCHSETTASK, so the resulting table has a foreign key (RunID). The primary key is RunID and TaskNumber.

TRANSACTION: TRANSACTION is a weak entity type of CLIENT, so the resulting table has a foreign key (ClientID). There is a unique constraint (ClientID, TransactionNum). The primary key is TransactionID.

TRANSACTIONHASRESULT: TRANSACTIONHASRESULT is the logical design of TRANSACTIONRUNRESULT and Has in the ER schema. TRANSACTIONHASRESULT is a weak entity type of CLIENTHASRESULT and TRANSACTION, so the resulting table has two foreign keys (BatchRunResID, ClientID). There is a unique constraint (ClientRunResID, TransactionID). The primary key is TransactionRunResID. For space and time efficiency, we derive and store the transaction run's summary information (the number of executed transactions, the minimum transaction processing time, the maximum transaction processing time, the sum of transaction processing time, and the sum of lock wait time) only, based on the measured transaction's execution records for that run. (The sum of lock wait time is computed by summing over the difference between a measured processing time and the minimum processing time.) This is different from the corresponding ER schema.

A.3 A DBMS Thrashing Observation Scenario

This section presents our scenario implementation for observing DBMS thrashing.

As discussed in the paragraph of scenario in Section 5.3, the scenario plugin implementation

in AZDBLAB should comply with an object-oriented hierarchy. The hierarchy consists of three layers. This layered architecture allows scenario development to be more efficient, extensible than otherwise.

The top layer is `Scenario`, which is an abstract class. This superclass contains several common methods needed for subclasses. For instance, `Scenario` has a method, named `recordRunProgress()`, which records the current progress of the experiment into the lab shelf DBMS. As an experiment's progress, including an event of finishing table population or complete a task (query or batchset mentioned above), is made, the method can be invoked by the subclasses inheriting `Scenario` and store into AZDBLAB the status.

`Scenario` also has a starting method, called `executeExperiment`(), which implements a series of an experiment procedure in which 1) a scenario-specific experiment gets launched and later completed (`executeSpecificExperiment()`), 2) we drop all the tables installed in the experiment (`dropExperimentTables()`), and 3) the experiment is finally finished by recording the completion status (`finishExperiment()`).

`executeSpecificExperiment`() should be implemented in the middle layer inheriting `Scenario`. The middle layer is developed based on a task. This middle layer, also an abstract class, represents a series of steps taking place in a specific task-based scenario. In a scenario, for instance, one step is to install tables for an experiment. The subsequent step is to perform a specific task. The last step is to uninstall the experiment tables and release all resources used in the experiment. In this way, the middle class designs and abstracts all the steps taken in the scenario. In this thesis `ScenarioBasedOnBatchSet` plays a role for the middle layer, and it provides the abstraction of the steps for our thrashing scenario.

The bottom layer is implemented by a subclass of the task-based middle class. This subclass implements in detail a task. Specifically, in the suboptimality scenario for a task we execute a query and collect the query execution data over varying cardinalities. In another scenario named `Exhausive`, for a task we execute all queries included in an experiment at a cardinality and record all the query execution data. Likewise, in the thrashing scenario for a task we execute a batchset including batches and gather the batchset execution data.

The concrete scenario is developed as a plugin and deployed as a jar file. If the user completes the concrete scenario class (the bottom layer) implementation, the user should build the class as

a jar file and place it in `plugins` under an AZDBLAB executable directory. When an executor begins to run an experiment, the executor locates the jar file of a scenario associated with that experiment and loads the Java class(es) from the jar file into AZDBLAB.

The function call hierarchy to run the DBMS thrashing scenario in our experiment can be summarized as follows:

`Main.runExperiment()` → `Scenario.executeExperiment()` →
`ScenarioBasedOnBatchSet.executeSpecificExperiment()` →
`DBMSThrashingScenario.studyBatchSet()`.

We now elaborate each of the methods in the hierarchy.

`Main.runExperiment()` : Listing 1 exhibits the `runExperiment()` method in the `Main` class. The method starts to run an experiment scheduled on a specific DBMS. It creates the experiment instance taking the function parameters for connecting to the central lab shelf server at Line 5. The function creates an experiment node instance representing the experiment instance at Line 9. At Line 11, the method extracts from the parsed experiment specification the configuration parameters to be passed at Line 24. At Line 14, the parsed parameters are initialized. Because of the lack of space we omit the initialization code. An experiment subject plugin manager instance, created at Line 16, locates a designated experiment subject plugin taking the function parameters needed for connecting to the chosen DBMS, and then the located experiment subject (plugin) instance is created at Line 18. At Line 21, we create an scenario instance, and then at Line 23, we check if the number of processors on an experimental machine is consistent with the corresponding value (`numProcs`) given from the experiment specification. At Line 24, all the extracted configuration parameters are internally set in the scenario instance, and at Line 31 we begin to run this scheduled experiment.

`Scenario.executeExperiment()`: Listing 2 exhibits the `executeExperiment()` method in the abstract `Scenario` class. This simple method conducts the scheduled experiment. It invokes in sequence `executeSpecificExperiment()` to execute the specified scenario in the experiment, `dropExperimentTables()` to drop all installed tables for the experiment, and then `finishExperiment()` to record the "completed" status in a lab shelf. While running an experiment, there may be an error thrown by the executor. If that's the case, the error will be

Listing 1 The `runExperiment()` Method

```
1  // Run an experiment (to observe DBMS thrashing).
2  public static void runExperiment(String lab_user_name, String lab_notebook_name,
3     String lab_experiment_name, String exp_user_name, String exp_password,
4     String machine_name, String connectString, String dbms, String start_time) throws Exception{
5     Experiment experiment = User.getUser(lab_user_name).getNotebook(lab_notebook_name)
6                              .getExperiment(lab_experiment_name); // Create an Experiment instance
7     try {
8         // Create an Experiment node for indicating the Experiment instance
9         ExperimentNode myExperiment = new ExperimentNode(experiment.getXactExperimentSource(),
10        experiment.getUserName(), experiment.getNotebookName());
11        myExperiment.processXML(); // Parse an experiment specification
12    } catch (FileNotFoundException e1) { e1.printStackTrace();}
13    // Initialize from the parsed XML the config. param. for setConfigParamters() below.
14    ...
15    // Create an ExperimentSubjectPluginManager instance
16    ExperimentSubjectPluginManager expSubPluginMan = new ExperimentSubjectPluginManager();
17    // Get an experiment subject instance
18    ExperimentSubject experiment_subject = expSubPluginMan.getExperimentSubject(exp_user_name,
19            exp_password,machine_name, connectString, dbms);
20    try {
21        Scenario scen = experimentRun.getScenarioInstance(); // Create an scenario instance
22        // Check if the specified number of processors is consistent with that of the machine
23        if(!CheckNumProcs(numProcs)){Main._logger.reportError("Inconsistent numProcs.");return;}
24        scen.setConfigParamters( // Set configuration parameters
25          numProcs, connTimeLimit, // number of processors & connection time limit
26          minReadSF,maxReadSF,incrReadSF,// read SF
27          minUpdateSF,maxUpdateSF,incrUpdateSF,// update SF
28          smallestMPL, largestMPL, incrMPL, // MPL variation
29          minROSEPct, maxROSEPct, incrROSEPct); // ROSE
30        scen.executeExperiment(); // start this experiment
31    } catch (Exception e) {...}
32 }
```

caught at Line 10, and the experiment will get paused at Line 11. The paused experiment can be unpaused by a user via Observer.

`ScenarioBasedBatchSet.executeSpecificExperiment()`**:** Listing 3 shows the detail of `executeSpecificExperiment()`. This method runs the scenario specified in the scheduled experiment. At Line 4 the method gets a run ID (`runID`) from this experiment run via `getRunID()` method of the member variable (`exp_run`). At Line 12 a batchset number is incremented. At Line 13 we obtain the maximum task number associated with `runID`. The maximum task number is equivalent to the batchset number that has been already studied in the

Listing 2 The `executeExperiment()` Method

```
1  // Execute an experiment.
2  public final void executeExperiment() throws Exception {
3          try {
4                  // Execute an experiment related to a chosen scenario
5                  executeSpecificExperiment();
6                  // drop all installed tables
7                  dropExperimentTables();
8                  // finish an experiment
9                  finishExperiment();
10         } catch (Exception ex) {
11                 pauseExperiment(ex.getMessage());
12                 throw new Exception("Experiment paused because of " + ex.getMessage());
13         }
14 }
```

current run. We don't have to restudy the already studied batchset. At Line 16 we set configuration variables for exponential back-off, designed to cope with a situation in which the network to the lab shelf server is not stable. At Line 17 `preStep()` installs and populates experiment tables. At Line 18 we study a batchset already existing or being generated based on the operationalized values of read and update SF and ROSE in `studyBatchSet()`.

The description of the arguments of `studyBatchSet()` is in the following. `runID` represents the current experiment run ID as mentioned above, `numProcs` indicates the number of processors set for this experiment, `buffCachSz` represents the configured buffer cache size, and `connTimeLimit` stores the value of CTL used in this experiment. The values of these three variables are parsed from the given experiment specification, as shown at Line 11 in Listing 1. The values are then passed to and set in the `Scenario` class. `dReadSF` and `dUpdateSF` indicate a specific SF value for a read-only and update-only transaction, respectively. `dRosePct` is a specific ROSE percentage. The minimum, maximum, and step values of these three variables are also parsed from the same specification, and each value of the variables is varied at Lines 38, 9, and 11, respectively.

Once the study of the running batchset is completed, we then record the "completed" status for the batchset and store the next task (batchset) number into AZDBLAB at Lines 24 and 26, respectively. If any exception happens in the middle of storing the run and task status, the exponential back-off is triggered at Line 27. If we fail 10 times, then an exception with a message of 'unstable JDBC connection' is thrown up to `Scenario`, and subsequently, the current

experiment run gets paused, which should be later resumed via Observer GUI. At Lines 37–38, `dReadSF` is set to the minimum read SF value for studying the next batchset if the current value is zero. Otherwise, its present value is multiplied by the increments (`incrReadSF`). We then continue to study the next batchset in the same run.

Listing 3 The `executeSpecificExperiment()` Method

```
1  // Execute a batchset experiment
2  protected final void executeSpecificExperiment() throws Exception {
3     // Get current RunID, Initialize batchSetNumber, finished last task number
4     int runID = exp_run_.getRunID(), batchSetNumToRun = 0, maxTaskNum = -1;
5     double dReadSF = 0;  // a specific read SF
6     // read SF operationalization
7     while(dReadSF <= maxReadSF){
8          // update SF operationalization
9          for(double dUpdateSF=minUpdateSF;dUpdateSF<=maxUpdateSF; dUpdateSF+=incrUpdateSF){
10           // ROSE operationalization
11          for(double dRosePct=minRosePct;dRosePct<=maxRosePct;dRosePct+=incrRosePct){
12               batchSetNum++;  // increment batchset number
13               maxTaskNum = getMaxTaskNum(runID);  // get the last finished task number
14               if(batchSetNum > maxTaskNum-1){
15                 // set configuration parameters for exponential back-off below
16                 int numTrials = 1, expBackoffWaitTime = 1000;
17                 preStep(); // initialize experiment tables
18                 studyBatchSet(runID, // study this batchset
19                    numProcs,connTimeLimit,dReadSF,dUpdateSF,dRosePct);
20                 // do exponential backoff when progress update fails
21                 while(numTrials <= Constants.TRY_COUNTS){
22                   try {
23                     // record progress
24                     recordRunProgress(100,String.format("Analyzed #%d BatchSet",batchSetNum));
25                     // store next task number
26                     putNextTaskNum(runID, batchSetNumToRun+1); break;
27                    }catch(Exception ex){ numTrials++; expBackoffWaitTime *= 2; // exp. backoff.
28                     try { Thread.sleep(currExpBackoffWaitTime);} catch (InterruptedException e) {}
29                    }
30                 }
31                 if(numTrials > Constants.TRY_COUNTS) // Exception occurs after 10 failures
32                   throw new Exception("JDBC connection to the lab shelf server is not stable.");
33                 numTrials = 1; expBackoffWaitTime = 1000;
34               } // end if
35           } // end "ROSE operationalization"
36         } // end "update SF operationalization"
37         if(dReadSF == 0) dReadSF = minReadSF;
38         else dReadSF*= incrReadSF;
39    } // end "read SF operationalization"
40 }
```

`DBMSThrashingScenario.studyBatchSet()`**:** Listing 4 shows `studyBatchSet()` in the concrete `DBMSThrashingScenario` class. This method studies an existing or a new batchset generated the passed parameters: a read SF (`dReadSF`), a update SF (`dUpdateSF`), and a ROSE percentage (`dRosePct`). At Line 8 we retrieve an existing batchset record from the lab shelf server or create a new batchset record in `stepA()`. As a result, a batchset ID (`batchSetID`) is returned. We use this ID and the other parameters—`runID`, `numProcs`, and `connTimeLimit`—to create in advance the batchset's run result record in the `BATCHSETHASRUN` table via `insertBatchSetResult()`. At Line 12 we call `stepB()` to set transaction generation parameters using `dReadSF`, `dUpdateSF`, and `dRosePct`. At Line 16 we retrieve from the lab shelf server a batch record matching the current MPL value. If such a batch record does not exist, then we create it. At Line 19 we invoke `stepC()`, which does the following tasks in sequence. We first retrieve or create the client records associated with the batch record. We then retrieve the transaction record associated with each of the client records, or we generate a client's transaction based on the configured generation parameters and store into the lab shelf server the generated transaction's record including an SQL statement, a value of the `TransactionSQL` attribute shown in Figure A.2. We subsequently execute the cold cache scheme as mentioned in Section 6.1 and then have each client run its own transaction until the passed CTL (`connTimeLimit`) is reached. Once the current batch run is over, the one-minute think (break) time is given before starting the next batch run. The batch run repeats as many times as specified in `Constants.MAX_ITERS` as shown at Line 18. Once all the batch run repetitions are done, `stepD()` disconnects the clients from the experiment `DBMS` subject. We then continue to run the transactions of the clients in the next batch, by increasing the current MPL value by steps of 100. In this way we can finish studying one batchset containing ten batches. Note that in the same experiment run multiple batchsets can be studied too, as seen in Listing 3.

A.4 Statistical Outputs by R

In this section we exhibit all the executed R commands and their corresponding outputs for the exploratory and confirmatory evaluations.

A.4.1 The Exploratory Evaluation

This section presents the statistical outputs shown along with our exploratory evaluation discussed in Chapter 8.

Listing 4 The `studyBatchSet()` Method

```
protected void studyBatchSet( // Study a batch set
  // runID, # of processors, connection time limit (CTL)
  int runID, int numProcs, int connTimeLimit,
  // read SF, update SF, and ROSE percentage.
  double dReadSF, double dUpdateSF, double dRosePct)
  throws Exception {
  // insert this batchset into database
  int batchSetID = stepA(dReadSF, dUpdateSF, dRosePct);
  // insert a batchset run result record in advance
  int batchSetRunResID = insertBatchSetRunResult(runID,batchSetID,numProcs,connTimeLimit);
  // set transaction generation parameters
  stepB(dReadSF, dUpdateSF, dRosePct);
  // run this batch of as many clients as MPL in this batchset
  // have each client run its own transaction repeatedly
  for (int MPL = smallestMPL; MPL <= largestMPL; MPL += incrMPL) {
    int batchID = insertBatch(batchSetID, MPL);
    // repeat running the transactions of the clients in the batch for X times
    for (int k = 1; k <= Constants.MAX_ITERS; k++) {// MAX_ITERS: # of batch executions
     stepC(batchSetRunResID, batchID, MPL, k);
     // wait for the one-minute of think time for the next iteration
     try{
       Thread.sleep(Constants.THINK_TIME); // one-minute break
     }catch(Exception ex){ ex.printStackTrace(); }
    }
    // release resources for the next iteration
    stepD();
  }
}
```

Correlational Analyses: Here is the output for the read group.

```
### all_r: thrashing samples captured in the read group
### H1: NUMPROCS and Thrashing Pt.
> cor.test(all_r$NUMPROCS, all_r$THRASHING_PT)
Pearson's product-moment correlation

data:  all_r$NUMPROCS and all_r$THRASHING_PT
t = 2.5904, df = 186, p-value = 0.009947
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.03149136 0.22675217
sample estimates:
      cor
0.1303858

### H2: NUMPROCS and ATPT
```

```
> cor.test(all_r$NUMPROCS, all_r$ATPT)

Pearson's product-moment correlation

data:  all_r$NUMPROCS and all_r$ATPT
t = -2.6248, df = 186, p-value = 0.009014
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.2283910 -0.0332179
sample estimates:
       cor
-0.1320844

### H3: ATPT and Thrashing Pt.
> cor.test(all_r$ATPT, all_r$THRASHING_PT)
Pearson's product-moment correlation

data:  all_r$ATPT and all_r$THRASHING_PT
t = -8.1596, df = 186, p-value = 4.734e-15
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.4643597 -0.2945983
sample estimates:
       cor
-0.3827048

### H4: READ_SF and Thrashing Pt.
> cor.test(all_r$READ_SF, all_r$THRASHING_PT)
Pearson's product-moment correlation

data:  all_r$READ_SF and all_r$THRASHING_PT
t = 0.4452, df = 186, p-value = 0.6567
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.1110048  0.1749173
sample estimates:
       cor
0.03262371

### H5: READ_SF and ATPT
> cor.test(all_r$READ_SF, all_r$ATPT)
Pearson's product-moment correlation

data:  all_r$READ_SF and all_r$ATPT
t = -0.579, df = 186, p-value = 0.5629
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.12830956  0.07012511
sample estimates:
```

```
cor
-0.0293817

### H6: ROSE and Thrashing Pt.
Pearson's product-moment correlation

data:  all_r$ROSE and all_r$THRASHING_PT
t = -1.8674, df = 186, p-value = 0.06341
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.273462770  0.007596024
sample estimates:
       cor
-0.1356617

### H8: PK and Thrashing Pt.
> cor.test(all_r$PK, all_r$THRASHING_PT)

Pearson's product-moment correlation

data:  all_r$PK and all_r$THRASHING_PT
t = 8.6669, df = 186, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.3160736 0.4827318
sample estimates:
      cor
0.4027353

### H9: PK and ATPT
> cor.test(all_r$PK, all_r$ATPT)
Pearson's product-moment correlation

data:  all_r$PK and all_r$ATPT
t = -5.2071, df = 186, p-value = 5.053e-07
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.4755242 -0.2250657
sample estimates:
       cor
-0.3566874
```

Here is the output for the update group.

```
### all_u: thrashing samples captured in the update group
### H1: NUMPROCS and Thrashing Pt.
> cor.test(all_u$NUMPROCS, all_u$THRASHING_PT)
Pearson's product-moment correlation
```

```
data:  all_u$NUMPROCS and all_u$THRASHING_PT
t = -6.5744, df = 297, p-value = 2.694e-10
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.4784073 -0.2695684
sample estimates:
       cor
-0.378799

### H2: NUMPROCS and ATPT
> cor.test(all_u$NUMPROCS, all_u$ATPT)

Pearson's product-moment correlation

data:  all_u$NUMPROCS and all_u$ATPT
t = -6.1616, df = 297, p-value = 2.34e-09
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.4335366 -0.2320949
sample estimates:
       cor
-0.3366621

### H3: ATPT and Thrashing Pt.
> cor.test(all_u$ATPT, all_u$THRASHING_PT)
Pearson's product-moment correlation

data:  all_u$ATPT and all_u$THRASHING_PT
t = -2.6127, df = 297, p-value = 0.0405
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.14836295  -0.07821498
sample estimates:
        cor
-0.11328896

### H4: UPDATE_SF and Thrashing Pt.
> cor.test(all_u$UPDATE_SF, all_u$THRASHING_PT)
Pearson's product-moment correlation

data:  all_u$UPDATE_SF and all_u$THRASHING_PT
t = -0.2518, df = 297, p-value = 0.8014
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.12782858  0.09898426
sample estimates:
       cor
-0.0146101
```

```
### H5: UPDATE_SF and ATPT
> cor.test(all_u$UPDATE_SF, all_u$ATPT)
Pearson's product-moment correlation

data:  all_u$UPDATE_SF and all_u$ATPT
t = -0.1649, df = 297, p-value = 0.8691
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.1228678  0.1039723
sample estimates:
         cor
-0.009570916

### H6: ROSE and Thrashing Pt.
>cor.test(all_u$ROSE, all_u$ATPT)
Pearson's product-moment correlation

data:  all_u$ROSE and all_u$ATPT
t = -1.2802, df = 297, p-value = 0.2015
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.18595077  0.03968073
sample estimates:
        cor
-0.07408304

### H8: PK and Thrashing Pt.
> cor.test(all_u$PK, all_u$THRASHING_PT)

Pearson's product-moment correlation

data:  all_u$PK and all_u$THRASHING_PT
t = 1.3001, df = 297, p-value = 0.1946
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.03853075  0.18706245
sample estimates:
       cor
0.07522836

### H9: PK and ATPT
> cor.test(all_u$PK, all_u$ATPT)
Pearson's product-moment correlation

data:  all_u$PK and all_u$ATPT
t = 2.657, df = 297, p-value = 0.18375
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.04238837 0.27930875
```

```
sample estimates:
      cor
0.1632005
```

Regression Analyses: Here is the output for the read group.

```
### regression on Thrashing Pt in the full path model.
> out.fit <- lm(formula = THRASHING_PT ~ PK + READ_SF + ROSE + ATPT + NUMPROCS, data = all_r)
> summary(out.fit)

Call:
lm(formula = THRASHING_PT ~ PK + READ_SF + ROSE + ATPT + NUMPROCS,  data = all_r)

Residuals:
     Min       1Q   Median       3Q      Max
-0.54022 -0.24221 -0.02016  0.23209  0.58479

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.32029    0.07919   4.045 7.97e-05 ***
PK             0.24987    0.05367   4.656 6.53e-06 ***
READ_SF        0.02265    0.05008   0.452   0.6516
ROSE           0.12564    0.07040   1.785   0.0761 .
ATPT          -0.16748    0.08072  -2.075   0.0395 *
NUMPROCS       0.10762    0.05047   2.132   0.0336 *
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3277 on 182 degrees of freedom
Multiple R-squared:  0.1383, Adjusted R-squared:  0.1127
F-statistic: 4.525 on 5 and 182 DF,  p-value: 0.0006524

### regression on Thrashing Pt in the reduced path model.
> out.fit <- lm(formula = THRASING_PT ~ PK + ATPT + NUMPROCS, data = all_r)
> summary(out.fit)

Call:
lm(formula = THRASING_PT ~ PK + ATPT + NUMPROCS, data = all_r)

Residuals:
    Min      1Q  Median      3Q     Max
-0.5727 -0.2337 -0.0296  0.2287  0.6288

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.39621    0.06799   5.827 2.76e-08 ***
PK             0.21472    0.04984   4.308 2.78e-05 ***
ATPT          -0.16999    0.08102  -2.098   0.0374 *
```

```
NUMPROCS       0.12997    0.07086   1.834 0.048373 .
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.2928 on 182 degrees of freedom
Multiple R-squared:  0.1207,Adjusted R-squared:  0.1051
F-statistic: 7.776 on 3 and 182 DF,  p-value: 6.765e-05

### regression on ATPT in the full path model
> med.fit <- lm(ATPT ~ PK:READ_SF + PK + READ_SF + ROSE + NUMPROCS + NUMPROCS:ROSE, data = all_r)
> summary(med.fit)

Residuals:
     Min       1Q   Median       3Q      Max
-0.54146 -0.21509  0.04979  0.16082  0.78462

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept)     0.57902    0.10750   5.386  2.41e-07 ***
PK             -0.29916    0.07122  -4.200  4.33e-05 ***
READ_SF         0.01230    0.06527   0.188     0.851
ROSE           -0.07394    0.14978  -0.494     0.622
NUMPROCS       -0.18655    0.08554  -2.005     0.031 *
PK:READ_SF      0.02919    0.12157   0.240     0.811
ROSE:NUMPROCS   0.10819    0.28347   0.382     0.703
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3231 on 181 degrees of freedom
Multiple R-squared:   0.15, Adjusted R-squared:  0.1194
F-statistic: 4.757 on 6 and 181 DF,  p-value: 0.0001585

### regression on ATPT in the reduced path model
> med.fit <- lm(ATPT ~ PK + NUMPROCS, data = all_r)
> summary(med.fit)

Call:
lm(formula = ATPT ~ PK + NUMPROCS, data = all_r)

Residuals:
     Min       1Q   Median       3Q      Max
-0.52009 -0.21736  0.05116  0.16370  0.76693

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.53511    0.04369  12.248  < 2e-16 ***
PK        -0.28739    0.05394  -5.328  3.1e-07 ***
NUMPROCS  -0.11716    0.07534  -1.555    0.012 *
---
```

```
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.317 on 181 degrees of freedom
Multiple R-squared:  0.1479,Adjusted R-squared:  0.1379
F-statistic: 14.84 on 2 and 181 DF,  p-value: 1.142e-06
```

Here is the output for the update group.

```
### regression on Thrashing Pt. in the full path model
> out.fit <- lm(THRASHING_PT ~ PK + ATPT + NUMPROCS + UPDATE_SF + ROSE, data = all_u)
> summary(out.fit)

Call:
lm(formula = THRASHING_PT ~ PK + ATPT + NUMPROCS + UPDATE_SF + ROSE, data = all_u)

Residuals:
    Min      1Q  Median      3Q     Max
-0.6219 -0.2520  0.1127  0.2814  0.7071

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.73033    0.07957   9.178  < 2e-16 ***
PK             0.06562    0.04166   1.575   0.1165
ATPT          -0.11129    0.06575  -1.693   0.0417 .
NUMPROCS      -0.44805    0.06511  -6.881 4.61e-11 ***
UPDATE_SF     -0.06207    0.07207  -0.861   0.3900
ROSE          -0.06147    0.07252  -0.848   0.3974
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.368 on 293 degrees of freedom
Multiple R-squared:  0.1642,Adjusted R-squared:  0.1478
F-statistic: 4.964 on 5 and 293 DF,  p-value: 0.0002224

### regression on Thrashing Pt. in the reduced path model
> out.fit <- lm(THRASHING_PT ~ ATPT + NUMPROCS, data = all_u)
> summary(out.fit)

Call:
lm(formula = THRASHING_PT ~ ATPT + NUMPROCS, data = all_u)

Residuals:
    Min      1Q  Median      3Q     Max
-0.6155 -0.2298  0.1392  0.2930  0.7087

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.67286    0.04420  15.224  < 2e-16 ***
ATPT          -0.09553    0.06432  -1.485   0.0439 *
```

```
NUMPROCS      -0.43408    0.06446  -6.734 1.07e-10 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3274 on 293 degrees of freedom
Multiple R-squared:  0.1508,Adjusted R-squared:  0.1442
F-statistic: 22.81 on 2 and 293 DF,  p-value: 7.574e-10

### regression on ATPT in the full path model
> med.fit <- lm(ATPT ~ NUMPROCS + ROSE + NUMPROCS:ROSE + PK + UPDATE_SF + UPDATE_SF:PK,
data = all_u)
> summary(med.fit)

Call:
lm(formula = ATPT ~ NUMPROCS + ROSE + NUMPROCS:ROSE + PK + UPDATE_SF + UPDATE_SF:PK,
data = all_u)

Residuals:
     Min       1Q   Median       3Q      Max
-0.44572 -0.23773 -0.06927  0.17636  0.81572

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.2608179  0.1079845   2.415   0.0164 *
NUMPROCS      -0.2845868  0.1484092  -1.918   0.0463 .
ROSE           0.1071328  0.1235238   0.867   0.3866
PK             0.1278712  0.0945363   1.353   0.1774
UPDATE_SF      0.0342841  0.0979067   0.350   0.7265
NUMPROCS:ROSE -0.0457667  0.2105161  -0.217   0.8281
UPDATE_SF:PK  -0.0001347  0.1378434  -0.001   0.9992
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3008 on 292 degrees of freedom
Multiple R-squared:  0.1306,Adjusted R-squared:  0.1099
F-statistic: 9.255 on 6 and 292 DF,  p-value: 2.727e-09

### regression on ATPT in the reduced path model
> med.fit <- lm(ATPT ~ NUMPROCS, data = all_u)
> summary(med.fit)

Call:
lm(formula = ATPT ~ NUMPROCS, data = all_u)

Residuals:
    Min      1Q  Median      3Q     Max
-0.3354 -0.2563 -0.1009  0.2019  0.8917

Coefficients:
```

```
              Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.40509    0.03455   11.72  < 2e-16 ***
NUMPROCS      -0.29674    0.05959   -4.98 1.17e-06 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3168 on 293 degrees of freedom
Multiple R-squared:  0.08768,Adjusted R-squared:  0.08415
F-statistic:  24.8 on 1 and 293 DF,  p-value: 1.167e-06
```

Causal Mediation Analyses: Here is the output for the read group.

```
> library(mediation) ### include mediation library
## testing the mediation through ATPT by NUMPROCS
> med.out <- mediate(med.fit, out.fit, mediator = "ATPT",
treat = "NUMPROCS")
> summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

       Estimate 95% CI Lower 95% CI Upper p-value
ACME            0.02905      0.00632      0.05465       0
ADE             0.12354      0.03126      0.21488       0
Total Effect    0.15260      0.06459      0.24115       0
Prop. Mediated  0.18564      0.03934      0.48834       0

Sample Size Used: 188

Simulations: 1000

## mediation sensitivity analysis on NUMPROCS
> sens.out <-medsens(med.out)
> summary(sens.out)

Mediation Sensitivity Analysis for Average Causal Mediation Effect

Sensitivity Region

      Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
[1,] -0.4 -0.0211      -0.0440       0.0019         0.16       0.1090
[2,] -0.3 -0.0072      -0.0204       0.0059         0.09       0.0613
[3,] -0.2  0.0053      -0.0070       0.0176         0.04       0.0273
[4,] -0.1  0.0171      -0.0026       0.0368         0.01       0.0068
[5,]  0.0  0.0285      -0.0008       0.0578         0.00       0.0000

Rho at which ACME = 0: -0.2
```

```
R^2_M*R^2_Y* at which ACME = 0: 0.04
R^2_M~R^2_Y~ at which ACME = 0: 0.0273

## testing the mediation through ATPT by PK
> med.out <- mediate(med.fit, out.fit, mediator = "ATPT", treat = "PK")
> summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

               Estimate 95% CI Lower 95% CI Upper p-value
ACME             0.0593       0.0318       0.0899       0
ADE              0.2596       0.1807       0.3349       0
Total Effect     0.3189       0.2449       0.3935       0
Prop. Mediated   0.1857       0.0959       0.2971       0

Sample Size Used: 188

Simulations: 1000

> sens.out <-medsens(med.out)
> summary(sens.out)

Mediation Sensitivity Analysis for Average Causal Mediation Effect

Sensitivity Region

      Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
[1,] -0.3 -0.0191      -0.0489       0.0107         0.09       0.0607
[2,] -0.2  0.0136      -0.0161       0.0433         0.04       0.0270

Rho at which ACME = 0: -0.2
R^2_M*R^2_Y* at which ACME = 0: 0.04
R^2_M~R^2_Y~ at which ACME = 0: 0.027

## testing the mediation through ATPT by READ_SF
> med.out <- mediate(med.fit, out.fit, mediator = "ATPT",
treat = "READ_SF")
> summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

                Estimate 95% CI Lower 95% CI Upper p-value
ACME            0.000314    -0.008533     0.010285    0.96
ADE             0.015300    -0.096499     0.132477    0.76
Total Effect    0.015615    -0.095567     0.134702    0.77
```

```
Prop. Mediated  0.000418    -0.625022     0.667832    0.98

Sample Size Used: 188

Simulations: 1000

> sens.out <-medsens(med.out)
> summary(sens.out)

Mediation Sensitivity Analysis for Average Causal Mediation Effect

Sensitivity Region

       Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
 [1,] -0.9  0.0310      -0.1962       0.2581         0.81       0.6223
 [2,] -0.8  0.0194      -0.1328       0.1717         0.64       0.4917
 [3,] -0.7  0.0145      -0.1010       0.1300         0.49       0.3765
 [4,] -0.6  0.0115      -0.0794       0.1023         0.36       0.2766
 [5,] -0.5  0.0093      -0.0624       0.0809         0.25       0.1921
 [6,] -0.4  0.0074      -0.0480       0.0629         0.16       0.1229
 [7,] -0.3  0.0057      -0.0352       0.0467         0.09       0.0691
 [8,] -0.2  0.0040      -0.0235       0.0316         0.04       0.0307
 [9,] -0.1  0.0022      -0.0126       0.0170         0.01       0.0077
[10,]  0.0  0.0003      -0.0032       0.0038         0.00       0.0000
[11,]  0.1 -0.0018      -0.0129       0.0093         0.01       0.0077
[12,]  0.2 -0.0041      -0.0279       0.0196         0.04       0.0307
[13,]  0.3 -0.0068      -0.0440       0.0304         0.09       0.0691
[14,]  0.4 -0.0098      -0.0615       0.0419         0.16       0.1229
[15,]  0.5 -0.0133      -0.0812       0.0547         0.25       0.1921
[16,]  0.6 -0.0175      -0.1047       0.0696         0.36       0.2766
[17,]  0.7 -0.0230      -0.1348       0.0889         0.49       0.3765
[18,]  0.8 -0.0307      -0.1794       0.1179         0.64       0.4917
[19,]  0.9 -0.0454      -0.2690       0.1782         0.81       0.6223

Rho at which ACME = 0: 0
R^2_M*R^2_Y* at which ACME = 0: 0
R^2_M~R^2_Y~ at which ACME = 0: 0
```

Here is the output for the update group.

```
## testing the mediation through ATPT by NUMPROCS
> med.out <- mediate(med.fit, out.fit, mediator = "ATPT",
treat = "NUMPROCS")
> summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals
```

```
       Estimate 95% CI Lower 95% CI Upper p-value
ACME            0.04072      0.00369      0.08325    0.04
ADE            -0.45962     -0.59140     -0.33967    0.00
Total Effect   -0.41890     -0.55133     -0.30253    0.00
Prop. Mediated -0.10173     -0.23249     -0.00871    0.04

Sample Size Used: 299

Simulations: 1000

> sens.out <- medsens(med.out)
> summary(sens.out)

Mediation Sensitivity Analysis for Average Causal Mediation Effect

Sensitivity Region

      Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
[1,] -0.2 -0.0304      -0.0721       0.0112         0.04       0.0294
[2,] -0.1  0.0037      -0.0365       0.0439         0.01       0.0074
[3,]  0.0  0.0369      -0.0055       0.0792         0.00       0.0000

Rho at which ACME = 0: -0.1
R^2_M*R^2_Y* at which ACME = 0: 0.01
R^2_M~R^2_Y~ at which ACME = 0: 0.0074

## testing the mediation through ATPT by PK
> med.out <- mediate(med.fit, out.fit, mediator = "ATPT", treat = "PK")
> summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

       Estimate 95% CI Lower 95% CI Upper p-value
ACME           -0.01471     -0.04001      0.00109    0.09
ADE             0.06808     -0.01194      0.14839    0.09
Total Effect    0.05337     -0.02824      0.13223    0.18
Prop. Mediated -0.20640     -2.54565      2.00415    0.24

Sample Size Used: 299

Simulations: 1000

> sens.out <- medsens(med.out)
> summary(sens.out)

Mediation Sensitivity Analysis for Average Causal Mediation Effect
```

```
Sensitivity Region

Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
[1,] -0.2  0.0123      -0.0055       0.0301         0.04       0.0294
[2,] -0.1 -0.0015      -0.0178       0.0148         0.01       0.0074
[3,]  0.0 -0.0149      -0.0334       0.0036         0.00       0.0000

Rho at which ACME = 0: -0.1
R^2_M*R^2_Y* at which ACME = 0: 0.01
R^2_M~R^2_Y~ at which ACME = 0: 0.0074

## testing the mediation through ATPT by UPDATE_SF
> med.out <- mediate(med.fit, out.fit, mediator = "ATPT",
treat = "UPDATE_SF")
> summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

       Estimate 95% CI Lower 95% CI Upper p-value
ACME           -0.00378     -0.02494      0.01291    0.65
ADE            -0.06124     -0.20848      0.07853    0.41
Total Effect   -0.06502     -0.21510      0.07200    0.38
Prop. Mediated  0.02111     -0.93935      1.25419    0.74

Sample Size Used: 299

Simulations: 1000

> sens.out <- medsens(med.out)
> summary(sens.out)

Mediation Sensitivity Analysis for Average Causal Mediation Effect

Sensitivity Region

       Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
 [1,] -0.9  0.0557      -0.2429       0.3543         0.81       0.5886
 [2,] -0.8  0.0329      -0.1674       0.2333         0.64       0.4651
 [3,] -0.7  0.0232      -0.1273       0.1737         0.49       0.3561
 [4,] -0.6  0.0173      -0.0983       0.1330         0.36       0.2616
 [5,] -0.5  0.0131      -0.0744       0.1006         0.25       0.1817
 [6,] -0.4  0.0096      -0.0534       0.0726         0.16       0.1163
 [7,] -0.3  0.0064      -0.0342       0.0469         0.09       0.0654
 [8,] -0.2  0.0032      -0.0165       0.0228         0.04       0.0291
 [9,] -0.1 -0.0002      -0.0045       0.0041         0.01       0.0073
[10,]  0.0 -0.0038      -0.0254       0.0177         0.00       0.0000
[11,]  0.1 -0.0078      -0.0490       0.0334         0.01       0.0073
```

```
[12,]  0.2 -0.0123      -0.0735       0.0490         0.04       0.0291
[13,]  0.3 -0.0174      -0.0992       0.0645         0.09       0.0654
[14,]  0.4 -0.0232      -0.1267       0.0803         0.16       0.1163
[15,]  0.5 -0.0301      -0.1572       0.0971         0.25       0.1817
[16,]  0.6 -0.0384      -0.1924       0.1157         0.36       0.2616
[17,]  0.7 -0.0491      -0.2365       0.1383         0.49       0.3561
[18,]  0.8 -0.0645      -0.2998       0.1709         0.64       0.4651
[19,]  0.9 -0.0936      -0.4250       0.2378         0.81       0.5886

Rho at which ACME = 0: -0.1
R^2_M*R^2_Y* at which ACME = 0: 0.01
R^2_M~R^2_Y~ at which ACME = 0: 0.0073
```

Testing Assumptions: Here is the output for the read group.

```
> library(car) ### library for testing assumptions
> library(ggplot2) ### library for rendering graphs
### 1), 2), and 3): no commands
### 4) No correlation of residuals
> durbinWatsonTest(out.fit)
 lag Autocorrelation D-W Statistic p-value
   1       0.5589201     0.8715399       0
 Alternative hypothesis: rho != 0
> durbinWatsonTest(med.fit)
 lag Autocorrelation D-W Statistic p-value
   1       0.7445287     0.4932397       0
 Alternative hypothesis: rho != 0
### 5) Homoscedasticity
> ncvTest(out.fit)
Non-constant Variance Score Test
Variance formula: ~ fitted.values
Chisquare = 0.3423284    Df = 1     p = 0.5584883
### 6) No multicolinearity
> sqrt(vif(out.fit)) > 2
    PK       READ_SF  ROSE     ATPT     NUMPROCS
    FALSE    FALSE    FALSE    FALSE    FALSE
### 7) No significant outliers: cooks distance
> cd = cooks.distance(out.fit)
> plot(cd, xlim=c(0, 200), ylim=c(0, 0.04),
main="(CD shouldn't be greater than 1)", xlab="Observation Number",
ylab="Cook's Distances (CDs)")
### see Figure 8.1 (a)
### 8) normality of residuals
> h <- hist(out.fit$res,main="",xlab="Residuals",xlim=c(-1,1), ylim=c(0,60))
> xfit<-seq(min(out.fit$res),max(out.fit$res),length=40)
> yfit<-dnorm(xfit,mean=mean(out.fit$res),sd=sd(out.fit$res))
> yfit <- yfit*diff(h$mids[1:2])*length(out.fit$res)
```

```
> lines(xfit, yfit, col="blue")
### see Figure 8.2 (a)
```

Here is the output for the update group.

```
### 1), 2), and 3): no commands
### 4) No correlation of residuals
> durbinWatsonTest(out.fit)
 lag Autocorrelation D-W Statistic p-value
   1       0.5453204     0.9006176       0
 Alternative hypothesis: rho != 0
> durbinWatsonTest(med.fit)
 lag Autocorrelation D-W Statistic p-value
   1       0.7378241     0.5215879       0
 Alternative hypothesis: rho != 0
### 5) Homoscedasticity
> ncvTest(out.fit)
Non-constant Variance Score Test
Variance formula: ~ fitted.values
Chisquare = 0.170502    Df = 1     p = 0.6796661
### 6) No multicolinearity
> sqrt(vif(out.fit)) > 2
    PK       READ_SF  ROSE     ATPT     NUMPROCS
    FALSE    FALSE    FALSE    FALSE    FALSE
### 7) No significant outliers: cooks distance
> cd = cooks.distance(out.fit)
> plot(cd, xlim=c(0, 300), ylim=c(0, 0.02),
main="(CD shouldn't be greater than 1)",
xlab="Observation Number",
ylab="Cook's Distances (CDs)")
### see Figure 8.1 (b)
### 8) normality of residuals
> h <- hist(out.fit$res,main="",xlab="Residuals",xlim=c(-1,1), ylim=c(0,50))
> xfit<-seq(min(out.fit$res),max(out.fit$res),length=40)
> yfit<-dnorm(xfit,mean=mean(out.fit$res),sd=sd(out.fit$res))
> yfit <- yfit*diff(h$mids[1:2])*length(out.fit$res)
> lines(xfit, yfit, col="blue")
### see Figure 8.2 (b)
```

A.4.2 The Confirmatory Evaluation

This section presents the statistical outputs shown along with our confirmatory evaluation discussed in Chapter 10.

Correlational Analyses: Here is the output for the read group.

```
### all_r: thrashing samples captured in the read group
## H1: numProcs vs. Thrashing Pt.
> cor.test(all_r$NUMPROCS, all_r$ATPT)

Pearson's product-moment correlation

data:  all_r$NUMPROCS and all_r$THRASHING_PT
t = 3.741, df = 146, p-value = 0.0002627
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.1411518 0.4362848
sample estimates:
    cor
0.29576

## H2: numProcs vs. ATPT
> cor.test(all_r$NUMPROCS, all_r$ATPT)
##### all  samples
Pearson's product-moment correlation

data:  all_r$NUMPROCS and all_r$ATPT
t = -7.6998, df = 146, p-value = 6.595e-14
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.3894070 -0.2371818
sample estimates:
       cor
-0.3153213

## H3: ATPT vs. Thrashing Pt.
> cor.test(all_r$ATPT, all_r$THRASHING_PT)

Pearson's product-moment correlation

data:  all_r$ATPT and all_r$THRASHING_PT
t = -3.2048, df = 146, p-value = 0.00166
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.40112162 -0.09912601
sample estimates:
       cor
-0.2563696

## H4: PK vs. Thrashing Pt.
> cor.test(all_r$PK, all_r$THRASHING_PT)
Pearson's product-moment correlation

data:  all_r$PK and all_r$THRASHING_PT
t = 4.619, df = 146, p-value = 4.832e-06
```

```
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.1128874 0.2753890
sample estimates:
      cor
0.1954796

## H5: PK vs. ATPT
> cor.test(all_r$PK, all_r$ATPT)
Pearson's product-moment correlation

data:  all_r$PK and all_r$ATPT
t = -7.8658, df = 146, p-value = 7.489e-13
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.6497162 -0.4213030
sample estimates:
       cor
-0.5455624
```

Here is the output for the update group.

```
## H1: numProcs vs. Thrashing Pt.
> cor.test(all_u$NUMPROCS, all_u$THRASHING_PT)
Pearson's product-moment correlation

data:  all_u$NUMPROCS and all_u$THRASHING_PT
t = -6.6396, df = 331, p-value = 1.57e-10
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.4608757 -0.2604885
sample estimates:
       cor
-0.3649005

## H2: numProcs vs. Thrashing Pt.
> cor.test(all_u$NUMPROCS, all_u$ATPT)
## intended: with filtering out all_u$THRASHING_PT < 1
Pearson's product-moment correlation

data:  all_u$NUMPROCS and all_u$ATPT
t = -2.8268, df = 331, p-value = 0.005033
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.27474501 -0.05015675
sample estimates:
       cor
-0.1645833
```

```
## H3: numProcs vs. Thrashing Pt.
## intended: (without filtering out all_u$THRASHING_PT < 1)
> cor.test(all_u$ATPT, all_u$THRASHING_PT)

Pearson's product-moment correlation

data:  all_u$ATPT and all_u$THRASHING_PT
t = -2.4503, df = 331, p-value = 0.01479
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.23754546 -0.02638012
sample estimates:
       cor
-0.1334774
```

Regression Analyses: Here is the output for the read group.

```
### regression on Thrashing Pt.
> out.fit <- lm(formula = THRASHING_PT ~ PK + ATPT + NUMPROCS, data = all_r)
> summary(out.fit)

Call:
lm(formula = THRASHING_PT ~ PK + ATPT + NUMPROCS, data = all_r)

Residuals:
    Min      1Q  Median      3Q     Max
-0.7010 -0.2157 -0.1013  0.2698  0.6548

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.39534    0.07219   5.476 1.88e-07 ***
PK             0.25693    0.06900   3.372 0.007102 **
ATPT          -0.27968    0.08975  -3.116 0.002213 **
NUMPROCS       0.33585    0.08427   3.986 0.000107 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3398 on 144 degrees of freedom
Multiple R-squared:  0.1589,Adjusted R-squared:  0.1414
F-statistic: 9.068 on 3 and 144 DF,  p-value: 1.542e-05

### regression on ATPT
> med.fit <- lm(ATPT ~ NUMPROCS + PK, data = all_r)
> summary(med.fit)

Call:
lm(formula = ATPT ~ NUMPROCS + PK, data = all_r)
```

```
Residuals:
     Min       1Q   Median       3Q      Max
-0.55398 -0.22739 -0.05854  0.30701  0.87981

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.53014    0.05023  10.553  < 2e-16 ***
NUMPROCS      -0.33928    0.07790  -5.504  0.00615 **
PK            -0.41977    0.05348  -7.849 8.47e-13 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3144 on 145 degrees of freedom
Multiple R-squared:  0.2989,Adjusted R-squared:  0.2892
F-statistic:  30.9 on 2 and 145 DF,  p-value: 6.614e-12
```

Here is the output for the update group.

```
### regression on Thrashing Pt.
> out.fit <- lm(THRASHING_PT ~ ATPT + NUMPROCS, data = all_u)
> summary(out.fit)

Call:
lm(formula = THRASHING_PT ~ ATPT + NUMPROCS, data = all_u)

Residuals:
     Min       1Q   Median       3Q      Max
-0.51309 -0.22895 -0.05133  0.18175  0.61878

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.59802    0.04014  14.897  < 2e-16 ***
ATPT          -0.12790    0.04391  -2.913  0.00386 **
NUMPROCS      -0.36867    0.05183  -7.113 9.11e-12 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.2632 on 331 degrees of freedom
Multiple R-squared:  0.1581,Adjusted R-squared:  0.1522
F-statistic: 26.86 on 2 and 331 DF,  p-value: 2.042e-1

### regression on ATPT
> med.fit <- lm(ATPT ~ NUMPROCS, data = all_u)
> summary(med.fit)

Call:
lm(formula = ATPT ~ NUMPROCS, data = all_u)

Residuals:
```

```
     Min       1Q   Median       3Q      Max
-0.53745 -0.32605 -0.01386  0.32881  0.71923

Coefficients:
      Estimate Std. Error t value Pr(>|t|)
(Intercept)    0.62342    0.03973  15.690  < 2e-16 ***
NUMPROCS      -0.34264    0.06109  -5.609  4.3e-08 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.3535 on 332 degrees of freedom
Multiple R-squared:  0.08655,Adjusted R-squared:  0.0838
F-statistic: 31.46 on 1 and 332 DF,  p-value: 4.299e-08
```

Causal Mediation Analyses: Here is the output for the read group.

```
library(mediation) ## include the mediation library
### testing mediation through ATPT by numProcs
med.out <- mediate(med.fit, out.fit, mediator = "ATPT",
treat = "NUMPROCS")
summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

       Estimate 95% CI Lower 95% CI Upper p-value
ACME            -0.0401      -0.0739      -0.0104    0.01
ADE              0.2044       0.1147       0.2907    0.00
Total Effect     0.1643       0.0812       0.2484    0.00
Prop. Mediated  -0.2402      -0.6569      -0.0604    0.01

Sample Size Used: 148


Simulations: 1000

sens.out <-medsens(med.out)
summary(sens.out)

Mediation Sensitivity Analysis for Average Causal Mediation Effect

Sensitivity Region

     Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
[1,] 0.1 -0.0055      -0.0349       0.0239         0.01       0.0082

Rho at which ACME = 0: 0.1
```

```
R^2_M*R^2_Y* at which ACME = 0: 0.01
R^2_M~R^2_Y~ at which ACME = 0: 0.0082

### testing mediation through ATPT by PK
med.out <- mediate(med.fit, out.fit, mediator = "ATPT", treat = "PK")
summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

       Estimate 95% CI Lower 95% CI Upper p-value
ACME             0.1167       0.0394       0.2036     0.0
ADE             -0.0242      -0.1549       0.1140     0.7
Total Effect     0.0925      -0.0238       0.2123     0.1
Prop. Mediated   1.1811      -3.9671      10.6847     0.1

Sample Size Used: 148


Simulations: 1000

Mediation Sensitivity Analysis for Average Causal Mediation Effect

Sensitivity Region

      Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
[1,] -0.3 -0.0248      -0.0981       0.0486         0.09       0.0531
[2,] -0.2  0.0251      -0.0482       0.0985         0.04       0.0236
[3,] -0.1  0.0720      -0.0033       0.1472         0.01       0.0059

Rho at which ACME = 0: -0.3
R^2_M*R^2_Y* at which ACME = 0: 0.09
R^2_M~R^2_Y~ at which ACME = 0: 0.0531
```

Here is the output for the update group.

```
## testing the mediation through ATPT by NUMPROCS
> med.out <- mediate(med.fit, out.fit, mediator = "ATPT", treat = "NUMPROCS")
> summary(med.out)

Causal Mediation Analysis

Quasi-Bayesian Confidence Intervals

       Estimate 95% CI Lower 95% CI Upper p-value
ACME             0.0617       0.0258       0.1047    0.00
ADE             -0.2192      -0.3379      -0.0970    0.00
Total Effect    -0.1575      -0.2756      -0.0349    0.01
```

```
Prop. Mediated  -0.3741      -1.9040      -0.1354    0.01

Sample Size Used: 333


Simulations: 1000

> sens.out <-medsens(med.out)
> summary(sens.out)

Mediation Sensitivity Analysis for Average Causal Mediation Effect

Sensitivity Region

      Rho    ACME 95% CI Lower 95% CI Upper R^2_M*R^2_Y* R^2_M~R^2_Y~
[1,] -0.3 -0.0205      -0.0424       0.0014         0.09       0.0737
[2,] -0.2 -0.0046      -0.0216       0.0124         0.04       0.0328
[3,] -0.1  0.0103      -0.0078       0.0285         0.01       0.0082

Rho at which ACME = 0: -0.2
R^2_M*R^2_Y* at which ACME = 0: 0.04
R^2_M~R^2_Y~ at which ACME = 0: 0.0328
```

Testing Assumptions: Here is the output for the read group.

```
### 1), 2), and 3): no commands
### 4) No correlation of residuals
> durbinWatsonTest(out.fit)
 lag Autocorrelation D-W Statistic p-value
   1       0.2660653      1.442629       0
 Alternative hypothesis: rho != 0
> durbinWatsonTest(med.fit)
 lag Autocorrelation D-W Statistic p-value
   1       0.7061902     0.5864416       0
 Alternative hypothesis: rho != 0
### 5) Homoscedasticity
> ncvTest(out.fit)
Non-constant Variance Score Test
Variance formula: ~ fitted.values
Chisquare = 1.605772    Df = 1     p = 0.2050871
### 6) No multicolinearity
> sqrt(vif(out.fit)) > 2
           PK           ATPT NUMPROCS
        FALSE         FALSE      FALSE
### 7) No significant outliers: cooks distance
> cd = cooks.distance(out.fit)
> plot(cd, ylim=c(0, 0.05),
```

```
main="(CD shouldn't be greater than 1)", ylab="Cook's Distances (CDs)",
xlab="Observation Number")
### see Figure 10.1 (a)
### 8) normality of residuals
> h <- hist(out.fit$res,main="",xlab="Residuals",ylim=c(0,40), xlim=c(-1,1))
> xfit<-seq(min(out.fit$res),max(out.fit$res),length=40)
> yfit<-dnorm(xfit,mean=mean(out.fit$res),sd=sd(out.fit$res))
> yfit <- yfit*diff(h$mids[1:2])*length(out.fit$res)
> lines(xfit, yfit, col="blue")
### see Figure 10.1 (c)
```

Here is the output for the update group.

```
### 1), 2), and 3): no commands
### 4) No correlation of residuals
> durbinWatsonTest(out.fit)
 lag Autocorrelation D-W Statistic p-value
   1       0.4401782      1.111135       0
 Alternative hypothesis: rho != 00
> durbinWatsonTest(med.fit)
 lag Autocorrelation D-W Statistic p-value
   1       0.8300255     0.3310545       0
 Alternative hypothesis: rho != 0
### 5) Homoscedasticity
> ncvTest(out.fit)
Non-constant Variance Score Test
Variance formula: ~ fitted.values
Chisquare = 0.5052569    Df = 1     p = 0.4771994
### 6) No multicolinearity
> sqrt(vif(out.fit)) > 2
           PK          ATPT  NUMPROCS
        FALSE         FALSE         FALSE
### 7) No significant outliers: cooks distance
> cd = cooks.distance(out.fit)
> plot(cd, ylim=c(0, 0.03),
main="(CD shouldn't be greater than 1)", ylab="Cook's Distances (CDs)",
xlab="Observation Number")
### see Figure 10.1 (b)
### 8) normality of residuals
> h <- hist(out.fit$res,main="",xlab="Residuals",ylim=c(0,50), xlim=c(-1,1))
> xfit<-seq(min(out.fit$res),max(out.fit$res),length=40)
> yfit<-dnorm(xfit,mean=mean(out.fit$res),sd=sd(out.fit$res))
> yfit <- yfit*diff(h$mids[1:2])*length(out.fit$res)
> lines(xfit, yfit, col="blue")
### see Figure 10.1 (d)
```

A.5 Consideration of Short Transaction Percentage

To apply more varied data to the model, we also designed a new variable, called *short transaction percentage* (STP), meaning the percentage of clients in a batch that use short transactions referencing only a single row. That was because our existing transactions were somewhat long in terms of the number of rows referenced by the transactions. In our previous experiment the minimum SF per transaction was set to 0.01%, which was equivalent to 100 rows across the DBMSes, corresponding to an STP of 0%.

In our additional experiment we operationalized different values, specifically 25%, 50%, 75%, and 100%, for STP. Suppose that 25% is set to STP. This means that the first 25% clients in a batch have short transactions referring to a single row that is randomly selected, while the other 75% clients in that batch have regular (long) transactions discussed in the paragraph of transaction size on page 51. Given a ROSE value (r), such a short transaction in a read or update group can be represented by the following SQL statement:

```
SELECT column_name
FROM table_name
WHERE id1 = a
```

or

```
UPDATE table_name
SET column_name = v
WHERE id1 = a
```

where a = an integer value randomly chosen in the interval of [0, $(c \times r)$], and the other variables like v and c are the same as described in Section 3.4.3 on page 53.

In our preliminary experiment based on the STP operationalization, we found that the 50% for STR produced the most thrashing batchsets. Thus, we decided to fix the value of STP to 50% for our additional experiment, where we used one and eight processors (producing the most thrashing batchsets in the read and update groups, respectively) for numProcs, 25% (producing the most thrashing batchsets in both of the groups) for ROSE, and the existing set of percentages for the read and update groups for SF in the absence and presence of PK. After conducting the additional experiment using STP, we had a total of 20 (= 2 (numProcs) $\times$ 2 (PK or non-PK) $\times$ 5 (DBMSes)) additional runs, each including (3 (for read SF) + 4 (for update SF)) $\times$ 2 (for ROSE) = 14 batchsets. Thus, we additionally collected a total of 280 BSIs, among

which 253 were retained after TAP. The number of the thrashing BSIs was 120, which was about 47% of the retained 253 BSIs. Hence, operationalizing STP helped us collect more thrashing samples.

We then combined the existing and additional data and recomputed the amount of variance explained by the existing model (without including STP) in Figure 9.1. Unfortunately the explained variance for each group was not improved. It seems that the mixed thrashing samples from long and short transactions hurt the overall explained variance. Further investigations on the operationalization and the data of the STP variable are left in the future work.

REFERENCES

[1] Takeshi Amemiya. Nonlinear Regression Models. *Handbook of Econometrics*, 1:333–389, 1983.

[2] André B. Bondi. Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2nd International Workshop on Software and Performance (WOSP '00)*, pages 195–203. ACM, 2000.

[3] Michael J. Carey, Sanjay Krishnamurthi, and Miron Livny. Load Control for Locking: The 'Half-and-Half' Approach. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*, pages 72–84. ACM, 1990.

[4] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Record*, 39(4):12–27, May 2011.

[5] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[6] Donald Cochrane and Guy H Orcutt. Application of Least Squares Regression to Relationships Containing Auto-Correlated Error Terms. *Journal of the American Statistical Association*, 44(245):32–61, March 1949.

[7] Paul Cohen. ***Empirical Methods for Artificial Intelligence***. MIT Press, 1995.

[8] R. Dennis Cook. Detection of Influential Observations in Linear Regression. *Technometrics*, 19(1):15–18, 1977.

[9] Sabah Currim, Richard T. Snodgrass, Young-Kyoon Suh, and Rui Zhang. A Causal Model of DBMS Suboptimality, 2015. To be submitted.

[10] Sabah Currim, Richard T. Snodgrass, Young-Kyoon Suh, and Rui Zhang. DBMS Metrology: Measuring Query Time, 2015. To be submitted.

[11] Sabah Currim, Richard T. Snodgrass, Young-Kyoon Suh, Rui Zhang, Matthew Johnson, and Cheng Yi. DBMS Metrology: Measuring Query Time. In *Proceedings of the 39th ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, pages 261–272. ACM, 2013.

[12] Harish D., Pooja N. Darera, and Jayant R. Haritsa. On the Production of Anorexic Plan Diagrams. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 1081–1092. VLDB Endowment, 2007.

[13] Asit Dan, Daniel M. Dias, and Philip S. Yu. The Effect of Skewed Data Access on Buffer Hits and Data Contention an a Data Sharing Environment. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB '90)*, pages 419–431. Morgan Kaufmann Publishers Inc., 1990.

[14] Peter J. Denning. Thrashing. http://denninginstitute.com/pjd/PUBS/ENC/thrash08.pdf, 2008.

[15] J. Durbin and G. S. Watson. Testing for Serial Correlation in Least Squares Regression. *Biometrika*, 58(1):1–19, 1971.

[16] William N. Evans. Durbin Watson Tables. https://www3.nd.edu/~wevans1/econ30331/Durbin_Watson_tables.pdf, viewed on Apr 5, 2015.

[17] FAL Labs. Tokyo Cabinet: A Modern Implementation of DBM. http://fallabs.com/tokyocabinet/, viewed on Feb 7, 2015.

[18] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Concurrency Control for High Contention Environments. *ACM Transactions on Database Systems (TODS)*, 17(2):304–345, June 1992.

[19] Cal Garbin. Path Analysis vis Regression. http://psych.unl.edu/psycrs/statpage/pathex1.pdf, viewed on Apr 5, 2015.

[20] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases (VLDB '81)*, pages 144–154. VLDB Endowment, 1981.

[21] Jim Gray, Raymond Lorie, Gianfranco Putzolu, and Irving Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394. North Holland Publishing Company, 1976.

[22] Jim Gray and Andreas Reuter. ***Transaction Processing: Concepts and Techniques***. Morgan Kaufmann Publishers Inc., 1st edition, 1992.

[23] Jayant R. Haritsa. The Picasso Database Query Optimizer Visualizer. *Proceedings of the VLDB Endowment (PVLDB)*, 3(2):1517–1520, August 2010.

[24] Andrew F. Hayes. ***Introduction to Mediation, Moderation, and Conditional Process Analysis: A Regression-Based Approach***. Guilford, 2013.

[25] Hans-Ulrich Heiss and Roger Wagner. Adaptive Load Control in Transaction Processing Systems. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB '91)*, pages 47–54. Morgan Kaufmann Publishers Inc., 1991.

[26] Grayson N Holmbeck. Toward Terminological, Conceptual, and Statistical Clarity in the Study of Mediators and Moderators: Examples from the Child-Clinical and Pediatric Psychology Literatures. *Journal of Consulting and Clinical Psychology*, 65(4):599, 1997.

[27] Takashi Horikawa. An Approach for Scalability-Bottleneck Solution: Identification and Elimination of Scalability Bottlenecks in a DBMS. *SIGSOFT Software Engineering Notes (SEN)*, 36(5):31–42, September 2011.

[28] Takashi Horikawa. Latch-Free Data Structures for DBMS: Design, Implementation, and Evaluation. In *Proceedings of the 39th ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, pages 409–420. ACM, 2013.

[29] Hwaci. SQLite. `http://www.sqlite.org/`, viewed on Feb 7, 2015.

[30] Kosuke Imai, Luke Keele, and Dustin Tingley. A General Approach to Causal Mediation Analysis. *Psychological Methods*, 15(4):309–334, 2010.

[31] Kosuke Imai, Luke Keele, Dustin Tingley, and Teppei Yamamoto. Causal Mediation Analysis Using R. In *Advances in Social Science Research Using R*. Springer-Verlag, New York, 2010.

[32] Kosuke Imai, Luke Keele, and Teppei Yamamoto. Identification, Inference, and Sensitivity Analysis for Causal Mediation Effects. *Statistical Science*, 25(1):51–71, 2010.

[33] Kosuke Imai and Teppei Yamamoto. Identification and Sensitivity Analysis for Multiple Causal Mechanisms: Revisiting Evidence from Framing Experiments. *Political Analysis*, 21(2), 2013.

[34] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines. In *Proceedings of the 4th International Workshop on Data Management on New Hardware (DaMoN '08)*, pages 35–40. ACM, 2008.

[35] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT '09)*, pages 24–35. ACM, 2009.

[36] Joint Committee for Guides in Metrology. International Vocabulary of Metrology Basic and General Concepts and Associated Terms (VIM) (3rd Ed.), 2012. `http://www.bipm.org/utils/common/documents/jcgm/JCGM_200_2012.pdf` (accessed Dec 05, 2014).

[37] Hyungsoo Jung, Hyuck Han, Alan D. Fekete, Gernot Heiser, and Heon Y. Yeom. A Scalable Lock Manager for Multicores. In *Proceedings of the 39th ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, pages 73–84. ACM, 2013.

[38] Stephen S. Lavenberg. ***Computer Performance Modeling Handbook***. Academic Press, 1983.

[39] J. Scott Long. ***Regression Models for Categorical and Limited Dependent Variables***. SAGE Publications, 2 edition, 1997.

[40] Charles F Manski. ***Identification for Prediction and Decision***. Harvard University Press, 2009.

[41] MariaDB Foundation. MariaDB: An Enhanced, Drop-in Replacement for MySQL. `https://mariadb.org/`, viewed on Feb 7, 2015.

[42] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, March 1992.

[43] Axel Mönkeberg and Gerhard Weikum. Conflict-Driven Load Control for the Avoidance of Data-Contention Thrashing. In *Proceedings of the Seventh International Conference on Data Engineering (ICDE '91)*, pages 632–639. IEEE Computer Society, 1991.

[44] Axel Mönkeberg and Gerhard Weikum. Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*, pages 432–443. Morgan Kaufmann Publishers Inc., 1992.

[45] Clayton Morrison and Richard T. Snodgrass. Computer Science Can Use More Science. *Communications of the ACM*, 54(7):36–39, June 2011.

[46] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and Resource Modeling in Highly-Concurrent OLTP Workloads. In *Proceedings of the 39th ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, pages 301–312. ACM, 2013.

[47] NIST/SEMATECH. *e*-Handbook of Statistical Methods: Critical Values of the Chi-Square Distribution. `http://www.itl.nist.gov/div898/handbook/eda/section3/eda3674.htm`, viewed on Apr 5, 2015.

[48] Oracle Corporation. The Java Database Connectivity (JDBC), 2014. `http://www.oracle.com/technetwork/java/javase/jdbc/index.html` (accessed April 15, 2014).

[49] Oracle Corporation. Innodb Parameters: `innodb_buffer_pool_size`. `http://dev.mysql.com/doc/refman/5.5/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size`, viewed on Aug 19, 2014.

[50] Oracle Corporation. MySQL: The World's Most Popular Open Source Database. `http://www.mysql.com/`, viewed on July 5, 2014.

[51] Oracle Corporation. Oracle Database 12c. `http://www.oracle.com/us/products/database/overview/index.html`, viewed on July 5, 2014.

[52] Oracle Corporation. Getting Started with Swing. `http://docs.oracle.com/javase/tutorial/uiswing/start/index.html`, viewed on May 16, 2014.

[53] Oracle Corporation. MySQL 5.6.5. `http://dev.mysql.com/doc/relnotes/mysql/5.6/en/news-5-6-5.html`, viewed on Oct 25, 2014.

[54] Oracle Corporation. Java Platform Standard Edition 7 API Specification. `http://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html#setQueryTimeout(int)`, viewed on Sep 24, 2014.

[55] Andrew Pavlo, Evan P. C. Jones, and Stanley Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *Proceedings of the VLDB Endowment (PVLDB)*, 5(2):85–96, October 2011.

[56] Elazur J. Pedhazur. ***Multiple Regression in Behavioral Research***. Thomson Learning, 1997.

[57] PhoneGap Community. PhoneGap. `http://phonegap.com/`, viewed on May 16, 2014.

[58] Kristopher J. Preacher, Derek D. Rucker, and Andrew F. Hayes. Addressing Moderated Mediation Hypotheses: Theory, Methods, and Prescriptions. *Multivariate Behavioral Research*, 42(1):185–227, 2007.

[59] R Core Team. R*: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014. ISBN 3-900051-07-0.

[60] Sudha Ram and Jun Liu. Understanding the Semantics of Data Provenance to Support Active Conceptual Modeling. In *Proceedings of the Active Conceptual Modeling of Learning Workshop (ACM-L 2006) in conjunction with the 25th International Conference on Conceptual Modeling (ER 2006)*, pages 1–12, 2006.

[61] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):1080–1091, August 2013.

[62] Rami Rosen. ***Linux Kernel Networking***. Springer, 2014.

[63] Stewart A Schuster. Relational Data Base Management for On-Line Transaction Processing. *Perspectives on Information Management: A Critical Selection of Computerworld Feature Articles*, 1981.

[64] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymon A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 5th ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*, pages 23–34. ACM, 1979.

[65] R Project Organization. Comprehensive R Archive Network. `http://CRAN.R-project.org`, viewed on Oct 4, 2014.

[66] Richard T. Snodgrass. ***Developing Time-Oriented Database Applications in SQL***. Morgan Kaufmann, 1999.

[67] Richard T. Snodgrass. Database Ergalics. `http://www.cs.arizona.edu/people/rts/ergalics/`, viewed on March 28, 2014.

[68] Richard T. Snodgrass. Automated Causal Analysis. `http://www.cs.arizona.edu/projects/focal/ergalics/autocausalanalysis.html`, viewed on May 16, 2014.

[69] Richard T. Snodgrass and Ilsoo Ahn. Temporal Databases. *IEEE Computer*, 19(9):35–42, September 1986.

[70] Richard T. Snodgrass and Peter Denning. The Science of Computer Science: Closing Statement: The Science of Computer Science (Ubiquity Symposium). *Ubiquity*, 2014(6):1–11, June 2014.

[71] Young-Kyoon Suh, Richard T. Snodgrass, John Kececioglu, Peter Downey, and Cheng Yi. EMP: An Execution Time Measurement Protocol for a Compute-Bound Program on Linux, 2015. To be submitted.

[72] Young-Kyoon Suh, Richard T. Snodgrass, and Rui Zhang. AZDBLAB: A Lab Information System for Large-scale Empirical DBMS Studies. *Proceedings of the VLDB Endowment (PVLDB)*, 7(13):1641–1644, September 2014.

[73] Young-Kyoon Suh, Rui Zhang, and Minjun Seo. AZDBLAB Schema Guide. Technical report, Department of Computer Science, University of Arizona, 2014.

[74] Symas Corporation. Symas Lightning Memory-Mapped Database (LMDB). `http://symas.com/mdb/`, viewed on Feb 7, 2015.

[75] Barbara G. Tabachnick and Linda S. Fidell. ***Experimental Designs Using ANOVA***. Cengage Learning, 2007.

[76] Y. C. Tay, Nathan Goodman, and Rajan Suri. Locking Performance in Centralized Databases. *ACM Transactions on Database Systems (TODS)*, 10(4):415–462, December 1985.

[77] The PostgreSQL Global Development Group. Shared Memory and Semaphores. `http://www.postgresql.org/docs/9.2/static/kernel-resources.html#SYSVIPC`, viewed on Aug 19, 2014.

[78] The PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Database. `http://www.postgresql.org/`, viewed on July 5, 2014.

[79] The PostgreSQL Global Development Group. PostgreSQL 9.2.9 Documentation. `http://www.postgresql.org/docs/9.2/static/`, viewed on Oct 25, 2014.

[80] Alexander Thomasian. Thrashing in Two-Phase Locking Revisited. In *Proceedings of the Eighth International Conference on Data Engineering (ICDE '92)*, pages 518–526. IEEE Computer Society, 1992.

[81] Alexander Thomasian. A Two-Phase Locking Performance and Its Thrashing Behavior. *ACM Transactions on Database Systems (TODS)*, 18(4):579–625, December 1993.

[82] Alexander Thomasian. A Performance Comparison of Locking Methods with Limited Wait Depth. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):421–434, May 1997.

[83] Alexander Thomasian. Concurrency Control: Methods, Performance, and Analysis. *ACM Computing Surveys*, 30(1):70–119, March 1998.

[84] Alexander Thomasian. Chapter 56: Performance Evaluation of Computer Systems. In ***Computing Handbook, Third Edition***. Chapman and Hall/CRC 2014, 2014.

[85] Dustin Tingley, Teppei Yamamoto, Kentaro Hirose, Luke Keele, and Kosuke Imai. **mediation**: R Package for Causal Mediation Analysis. *Journal of Statistical Software*, 59(5):1–36, 2014.

[86] UCLA. Chi-Square Distribution Table. `http://www.socr.ucla.edu/Applets.dir/ChiSquareTable.html`, viewed on Apr 5, 2015.

[87] Jodie Ullman. Structural Equation Modeling: Reviewing the Basics and Moving Forward. *Journal of Personality Assessment*, 87(1):35–50, 2006.

[88] W3C. XMLHttpRequest. `http://www.w3.org/TR/XMLHttpRequest/`, January 2014.

[89] Gerhard Weikum, Axel Moenkeberg, Christof Hasse, and Peter Zabback. Self-Tuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 20–31. VLDB Endowment, 2002.

[90] Wikiversity. Multiple Linear Regression/Assumptions. `http://en.wikiversity.org/wiki/Multiple_linear_regression/Assumptions`, viewed on Oct 31, 2014.

[91] Green Sewall Wright. Correlation and Causation. *Journal of Agricultural Research*, 20(7):557–585, 1921.

[92] Philip S. Yu, Daniel M. Dias, and Stephen S. Lavenberg. On the Analytical Modeling of Database Concurrency Control. *J. ACM*, 40(4):831–872, September 1993.

[93] Bin Zhang and Meichun Hsu. Modeling Performance Impact of Hot Spots. In ***Performance of Concurrency Control Mechanisms in Centralized Database Systems***, pages 148–165. Prentice-Hall, Inc., 1995.

[94] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 449–464. USENIX Association, 2014.