An Empirical Study of Transaction Throughput Thrashing Across Multiple Relational DBMSes

Young-Kyoon Suh[a,1,*], Richard T. Snodgrass[b], Sabah Currim[c]

[a] *Super-Computing R&D Center, KISTI, Daejeon, Republic of Korea, 34141*
[b] *Department of Computer Science, University of Arizona, Tucson, Arizona, 85721*
[c] *Alumni Association, University of Arizona, Tucson, Arizona, 85721*

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. Thrashing behavior in DBMSes is a serious concern to database administrators (DBAs) as well as to DBMS implementers. From an engineering perspective, therefore, it is of critical importance to understand the causal factors of DBMS thrashing. However, understanding the origin of thrashing in modern DBMSes is challenging, due to many factors that may interact with each other.

This article aims to better understand the thrashing phenomenon across multiple DBMSes. We identify some of the underlying causes of DBMS thrashing. We then 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 important engineering implications for improvements in transaction processing.

Keywords: DBMS Thrashing, Transaction, Throughput, Factors, Structural Causal Model, Empirical Study

[*] Corresponding author

Email addresses: [a] youngkyoon.suh@gmail.com (Young-Kyoon Suh), [b] rts@cs.arizona.edu (Richard T. Snodgrass), [c] scurrim@email.arizona.edu (Sabah Currim)

[1] The author was at the University of Arizona when starting this research.

1. Introduction

Database management systems (DBMSes) are a core component of current information technology (IT) systems [1]. DBMSes have been widely adopted to serve a variety of workloads such as on-line analytical processing (OLAP) [2] and on-line transaction processing (OLTP) [3] applications. Over the last five decades, achieving high performance in handling workloads has been a primary goal in the database community. Accordingly, various methodologies and techniques have been proposed to enhance the efficiency of DBMSes and thereby, of database applications.

Many DBMS performance issues have been addressed and resolved over the decades, but scalability is still regarded as a major concern [1, 4]. When a scalability bottleneck is encountered in a DBMS, transaction throughput can drop. In the worst case the DBMS may experience performance degradation exhibited by thrashing [5], in which a drastic reduction in throughput occurs.

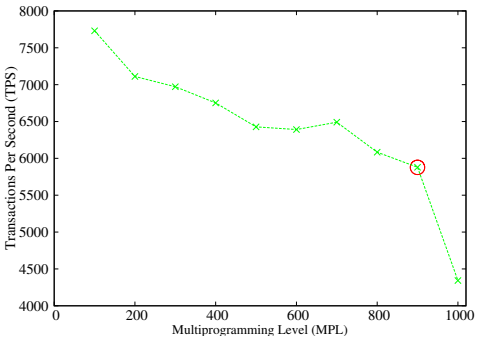
1.1. The DBMS Thrashing Problem

DBMS thrashing [6, 7, 8, 9] is a precipitous drop of transaction throughput over increasing multiprogramming level (MPL), defined as the number of active database connections. Interestingly, DBMS thrashing is observed in modern high-performance DBMSes, as demonstrated in Figure 1.

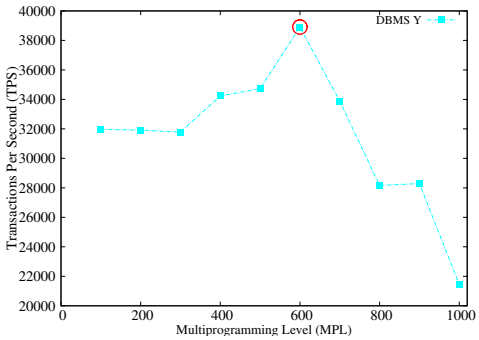
We studied three proprietary DBMSes[2] (denoted by X, Y, and Z) and two open-source DBMSes (MySQL and PostgreSQL). Each DBMS was run exclusively on a dedicated machine with the same hardware specification as will be described in Table 2. One DBMS was run on Windows due to its unavailability on Linux.

Note that the configurations differ across the DBMSes in Figure 1, specifically, in the number of processors (numProcs) on each machine running a DBMS, as our goal in these initial experiments are just to see whether each DBMS can be induced to thrash. This variation may explain why PostgreSQL's (Figure 1(e)) performance appeared to be about 2x better than that of MySQL (Figure 1(d)), primarily because of twice more processors. It may also account for why DBMS Y's (Figure 1(b)) performance appeared to dominate the others by up to about 10x, perhaps due to the most eight processors. Moreover, that 10x more tuples were scanned compared to the

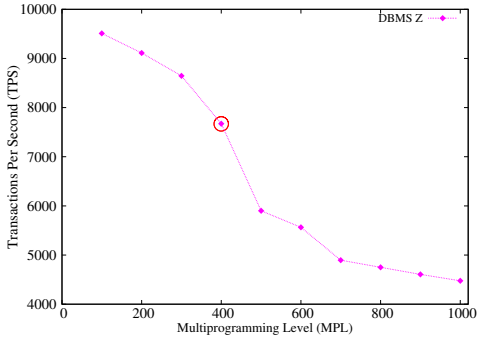
[2]As legal agreements do not allow us to report the performance evaluation results of the commercial, their names are anonymized in Figure 1.



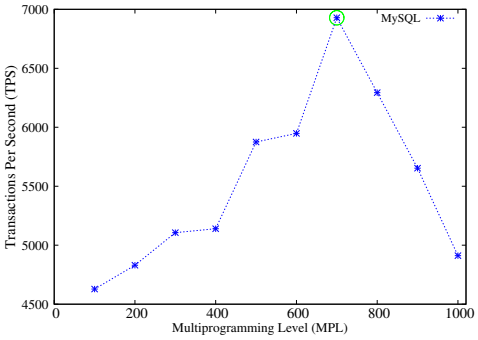
(a) DBMS X's Thrashing (numProcs=1)



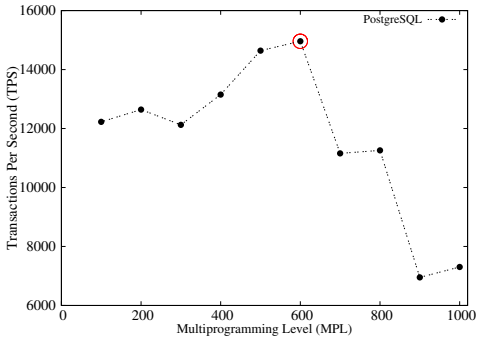
(b) DBMS Y's Thrashing (numProcs=8)



(c) DBMS Z's Thrashing (numProcs=8, 10% more tuples scanned)



(d) MySQL's Thrashing (numProcs=2)



(e) PostgreSQL's Thrashing (numProcs=4)

Figure 1: Observed DBMS Thrashing

others, may explicate why the performance of DBMS Z was poorer than that of DBMS Y's given the same number of processors. That said, a focus on these experiments was not to see "different performance" under different configuration across the DBMSes. The real attention was on "something else," i.e., thrashing, to be discussed now.

In the figure the x-axis shows varying MPL from 100 to 1,000 in steps of 100, and the y-axis represents the number of completed transactions, also known as transactions per second (TPS)[10], that we observed for a run at each MPL. The transactions in Figure 1 are all read-only, with the range of selected rows overlapping between the transactions within the same MPL. (We will describe how we generate transactions in greater detail in Section 5.)

As shown in Figure 1, we observed that thrashing occurred in all the DBMSes. For DBMS X, thrashing started at an MPL of 900, marked with a red circle. At that MPL, there is a sharp drop in throughput, indicating that DBMS X entered the "thrashing phase." The MPL of 900 is thus called the "thrashing point (TP)," after which DBMS X experienced thrashing.

The figure shows that this thrashing phenomenon was also observed in the other DBMSes, whose thrashing occurred at MPLs of 600, 400, 700, and 600 marked with a red circle. In particular, there were slight bumps at MPLs of 900 and 800 (and 1000) after DBMS Y's and PostgreSQL's thrashing occurred. But such bumps were within the thrashing phases of those two DBMSes: The thrashing phase persisted until the last MPL of 1,000.

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

Thrashing behavior in DBMSes is a serious concern to DBAs (database administrators) engaged in developing OLTP or OLAP systems, as well as DBMS implementers developing te chnologies related to concurrency control. In general, if thrashing occurs in a DBMS, many transactions may be aborted, perhaps resulting in little progress in transaction throughput over time. From an engineering perspective, it is of critical importance to understand the factors of DBMS thrashing.

However, understanding the origin of thrashing in modern DBMSes is challenging. The challenge stems from many factors that may correlate with each other and contribute to thrashing. No *structural causal model* has been proposed to articulate why and when DBMSes thrash. If such a model existed, then DBAs and DBMS developers would better understand the root causes of DBMS thrashing and perhaps could predict the occurrence

of thrashing when running their workloads on their DBMSes and could then take directed corrective actions (similarly to how errors in database design can be predicted [11] and then addressed).

1.2. Research Questions

In this article we address the following three research questions.

1. What *factors* impact the prevalence of DBMS thrashing?
2. How do the factors of DBMS thrashing *relate*?
3. How much *variance* is accounted for by each factor influencing DBMS thrashing, as well as how much variance is explained by all the proposed factors in concert?

The database community has not adequately addressed these three important questions regarding DBMS thrashing. To answer the questions, this article takes a fundamentally different approach (to be described shortly) than the analytical and simulation methods taken by a rich body of existing literature [9, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22].

The first question concerns what factors can contribute to DBMS thrashing. Prior research indicates that various factors, grouped into different complexity constructs (by relevance) such as i) *resource complexity* [7, 8, 23] defined as system resources, ii) *transaction complexity* [6, 9, 12, 13, 24] as transaction characteristics, and iii) *processing complexity* as DBMS's transaction processing capability [9, 10, 12, 19], all impact DBMS thrashing, as well as does iv) *schema complexity* defined as database schema, additionally considered in this article. Section 3.2 describes the complexity constructs and each of these factors in further detail.

The second question about how all the factors relate is one of the key contributions of this article. This question is addressed in Section 4. Prior research looked at the impact of one factor in isolation; this is the first time, that we are aware, that a structural causal model for DBMS thrashing has been developed and tested. In this article, we show an initial model shown in Figure 3 and then proceed with a refined model to be shown in Figure 6.

The last question concerns a quantitative approach to explaining DBMS thrashing. To answer this question, we collect empirical data on actual DBMSes. We conducted rigorous experiments by first operationalizing and then setting (for independent variables) or measuring (for dependent) variables within the context of the experiments. The measured data can then

be used to test the relationships hypothesized from the model via a variety of statistical tools including regression [25], correlation [25], path [26], and causal mediation [27] analyses.

In the course of addressing these three questions, this article proposes a novel structural causal model that can explicate the occurrence of thrashing in DBMSes better. This model (a) helps the database community better understand the phenomenon of thrashing, (b) provides DBMS implementers and DBAs with engineering implications drawn from the model, and (c) guides researchers to identify and investigate other as-yet-unknown factors that may contribute to DBMS thrashing.

1.3. Contributions

This article presents the following contributions.

- Elaborates a methodological perspective that treats DBMSes as experimental subjects and uses that perspective to study DBMS thrashing.
- Proposes a novel 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 recent research infrastructure [28], to schedule and run large-scale experiments over multiple relational DBMSes, providing empirical data to evaluate the proposed structural causal model.
- Proposes a novel thrashing analysis protocol (TAP) that can be applied to the collected data.
- Conducts rigorous statistical analyses on the data refined by 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, by proposing an initial tested model that can be extended and refined.

1.4. Organization

The next section reviews existing related literature. Section 3 identifies several factors from the literature. We then present an initial structural causal model based on the factors and their hypothesized correlations, thereby providing a set of hypotheses to be empirically tested. Next, we describe how the factors, or variables, can be operationalized to observe their influence on thrashing and then propose a novel thrashing measurement protocol. We conduct an exploratory evaluation of the initial model. Subsequently, we refine the initial model and perform confirmatory evaluation of the refined causal model. The model in turn provides several implications for treating workloads more efficiently. Lastly, we summarize our findings and list some remaining opportunities to augment the explanatory power by improving the amount of variance explained by the final model.

2. Related 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 [6, 9, 14, 15, 16, 17]. There have been several more recent contributions [1, 7, 8, 23] as well.

In general, there are three ways to understand any observable phenomenon (in this case, thrashing). The first way is to build and evaluate an *analytical model*. Tay [17] proposed an analytical model and simulation to understand, compare, and control the performance of concurrency-control algorithms. He suggested from his analytical model that data contention thrashing was due to blocking rather than to restarts, and that resource contention (competing for a transaction to finish the computation) caused thrashing to occur earlier. Möenkeberg also pointed out through his analytical model that too many active transactions in the system could incur excessive lock requests [16].

Thomasian introduced a methodology based on queuing theory to evaluate the performance of concurrency control methods in transaction processing systems [29]. Through a series of analytical models [18, 19, 20], 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, due to a snowball effect, which refers to the blocking of transactions causing further blocking of transactions.

A benefit of the analytical modeling is the conciseness of the resulting model. Once all the variables used in the model are fully understood, it is

easy to follow the entire flow of how the model is derived. At the end, we can expect the well-summarized model in a clean mathematical form.

The second approach is *simulation*, used in many studies [9, 14, 15, 16, 17]. One advantage lies in its flexibility [29], as a user can try various 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 [30], may relieve a hard burden of developing simulations.

Much of the existing work relies on simulation as well as analytical models. A drawback of these analytical and simulation methodologies is that the analytic and simulation results may not hold true for actual DBMSes. It is hard to generalize their results to real DBMSes. The analytical and simulation methods can be limited in capturing complex behaviors among transactions and resources (CPU, I/O, and memory) in a current DBMS [31].

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 carried out before multi-core processors existed.

The third way is to utilize an *empirical approach* [32, 33], measuring a real system. Recent studies [1, 7, 8, 23] used an actual DBMS. They examined transaction throughput bottlenecks that emerged on a DBMS running on a multi-core system (open source DBMSes (MySQL [1, 8], PostgreSQL [23], or their own DBMS (Shore-MT) [7]). 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 third approach is that each evaluation was conducted on one (or in a few cases, two) open-source DBMS(es). Their conclusions only apply directly to those DBMS(es). Their work could not definitively demonstrate that their results could be generalized to other (non-open source) DBMSes. As there are large differences between existing DBMSes, 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. As demonstrated in Figure 1, the thrashing phenomenon is observed across DBMSes. This implies that it is of critical importance to study thrashing behavior by regarding DBMSes as a general class of computational artifacts.

A recent study [34] attempts to expose and diagnose violations of atomicity, consistency, isolation, and durability (ACID) [35]—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) [36, 37] systems

(TokyoCabinet [38], LightningDB [39], and SQLite [40]), one commercial key-value store, and one open-source (MariaDB [41]) and three proprietary OLTP database servers. The approach is somewhat similar to that taken here, in that both 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.

None of the existing work considers correlations among factors contributing to thrashing. All previous investigations emphasize only a single factor. As more factors emerge, their interactions cannot be ignored in understanding the causality of DBMS thrashing.

In short, while there have been a good number of analytical models and simulations and empirical work to understand DBMS thrashing, the existing work has not yet broadened to address the following concerns: 1) simulation or analytical study of models rather than real DBMSes, 2) study restricted to one (or at most two) specific DBMS rather than multiple DBMSes, and 3) little consideration of relationships of factors.

In contrast, we utilize a novel, different empirical methodology, termed *ergalics* [42]: developing a structural causal model to better explain the DBMS thrashing phenomenon. Our empirical study 1) identifies variables affecting the thrashing phenomenon, 2) constructs a structural causal model based on the variables and their hypothesized correlations, 3) operationalizes each of the variables—determines how to measure each of the variables and then designs experiments setting or observing the variables along with the variable measurement decision, 4) collects data by running the designed 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 how to design an experimental methodology across a variety of DBMSes. It is not easy to state a consistent operationalization for a variable across very different DBMSes. The second challenge is to conduct such large-scale experiments over several DBMSes. Managing such experiments can be difficult.

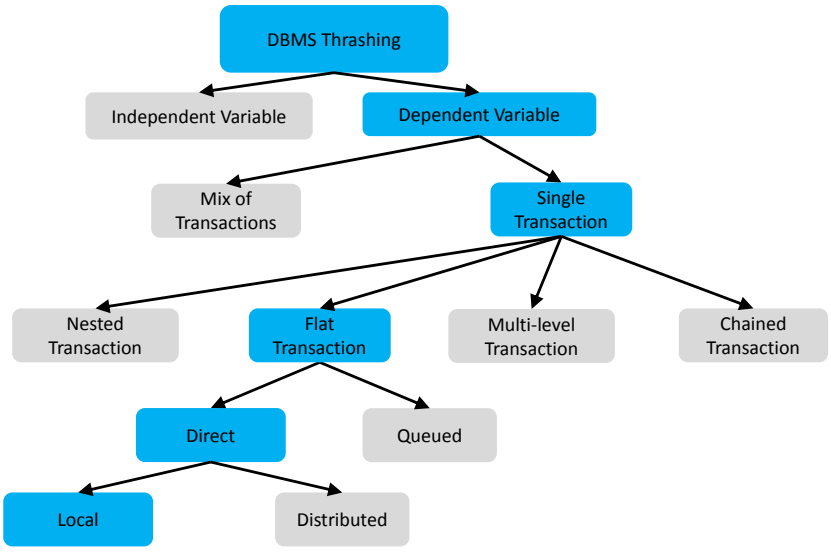


Figure 2: Taxonomy of DBMS Thrashing

3. Exploring Potential Thrashing Factors

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 2 presents a taxonomy for measuring DBMS thrashing. We treat DBMS thrashing as a "dependent" variable, in that it is *observed*, not manipulated. The variable is captured by the "maximum MPL" after which thrashing behavior emerges, called the *thrashing point*, discussed in Section 3.2.5.

There are several factors contributing to DBMS thrashing. These factors are treated as "independent" variables, defined as potentially impacting on thrashing behavior. We introduce the independent variables in this section.

We observe the dependent variable of DBMS thrashing by operationalizing—intervening (or, setting the values of) or observing (or, measure the values of)—the independent variables. In Section 5 we elaborate in detail how to operationalize them. Note that variable operationalization indicates how to set an independent variable and measure a dependent variable.

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. Here we cover only the single transaction type due to complexity of the mixed type.

In the context of Gray's categorization [43], our interest lies in a *flat transaction* that contains an arbitrary number of actions, that is, has the ACID properties [35] 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 are not covered due to complexity.

We focus on *direct transactions* where the terminal interacts directly with the server; in this study 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. *Distributed transactions* are not covered in this study because of their complexity.

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

3.1. Terminology

A *transaction* is represented by a single SQL statement, either a select or an update. A *client* is specified by a `Connection` instance created by a JDBC driver [44] configured for connecting to a DBMS subject, and it is implemented within 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 3.2.2), 3) call `commit`() to execute the commit of that transaction, 4) keep running the same transaction until a specified connection time limit (CTL), indicating how long a connection persists in the DBMS, 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," is a group of batches whose size increases by steps of a certain number. The concept of a batchset is introduced to determine the specific thrashing point (TP). (A full view of a batchset is later given in Figure 4.)

3.2. Relevant Variables

Drawing from the existing literature [6, 7, 8, 9, 10, 12, 13, 19, 23, 24] and our understanding, we have included several independent variables (i.e., the number of logical processors, transaction size, and referenced row subset region) to be covered in detail shortly and one dependent variable (i.e., average transaction processing time) that can influence DBMS thrashing, in the initial causal model to be shown later in Figure 3. These variables are grouped into four constructs in the model, which are described in the following.

3.2.1. The Resource Complexity Construct

Resource complexity is a construct related to system resources that a DBMS utilizes. This construct includes one variable: "number of processors" (*numProcs*).

The Number of Processors. This is an independent variable capturing the number of logical processors (hereafter, processors) available to DBMSes. This variable addresses level of parallelism from which the DBMSes can benefit from when executing their transactions. As pointed out in Section 2, recent studies [7, 8, 23] have paid much attention to the influence of concurrency by multi-core processors on thrashing. If we can utilize more cores, 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 cores are available to a DBMS, thrashing may increase because the degree of core contention becomes greater as workloads increase. The impact of multi-cores on thrashing should thus be better understood.

3.2.2. The Transaction Complexity Construct

Transaction complexity is a construct governing transaction properties that cause DBMS thrashing. In this construct we include the following two variables: "selectivity factor" (*SF*) and "reference row subset region" (*ROSE*), to be described. These two variables capture the number of rows requested by an individual transaction and the contention among transactions, respectively.

Selectivity Factor. This variable literally indicates selectivity factor [45] of a read or write transaction. It actually captures *transaction size*, introduced by Thomasian [9]. Transaction size represents the number of objects requested by transactions. Transaction size can impact transaction throughput [6]. If a transaction is large, that is, if a transaction accesses 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 a transaction is small, that is, references just a few objects, then fewer locks will be issued, fewer lock conflicts will be incurred, and thus the chance of thrashing will decrease.

Reference Row Subset Region. This variable called "referenced row subset region (ROSE)" captures the maximum region of rows (objects) that can be referenced by transactions. The variable captures *effective database size* in a previous study [9]. The variable is also involved with transactions' hot spots [12, 13, 24]. 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.

ROSE can affect DBMS thrashing. If ROSE is small, the degree of contention on referencing the same rows will significantly increase. Substantial locking overhead can be incurred during the transaction processing. If ROSE is large, on the other hand, contention will decrease. Many rows in the extended ROSE, however, can be referenced by transactions. Many locks on these many rows can be issued, thereby charging heavy lock management overhead and contributing to a significant drop in throughput.

3.2.3. The Schema Complexity Construct

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

Presence of Primary Keys. This variable captures the influence of I/O on DBMS thrashing. 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 see that PK correlates with TP.

3.2.4. The Processing Complexity Construct

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

Transaction Response Time. This is defined as the end-to-end time of completing a transaction on a DBMS.

It seems reasonable that the faster an individual transaction is processed, the higher the throughput in the end, delaying the onset of thrashing.

Certainly, the average transaction response time is lowest when one user exclusively submits transactions to the DBMS. As more active sessions are added, however, average transaction response time increases smoothly (or radically). Since having only one active session cannot realize a high throughput, typically the DBMS allows many concurrent sessions (MPL) to run their own transactions at the same time. As the number of user sessions grows, an average individual response time might be increased, but overall transaction throughput might keep rising (and perhaps falling later). Thus, we speculate that the shorter response time, the higher throughput, thereby slowing down the onset of thrashing.

In the meantime, this speculation may not hold. The response time (which also includes computation and I/O time in performing a transaction) may rise sharply due to synchronization overhead [9, 10, 12, 19, 22], i.e., lock conflicts or snapshot isolation, among the concurrent transactions. The overhead may eventually lead to an intolerable response time, at that point most likely resulting in a significant drop in transaction throughput, or thrashing.

In sum, transaction processing time can be an important variable to capture the impact (strength and direction) on thrashing caused by distinct code and (concurrency control) performance of different DBMSes.

3.2.5. The Thrashing Point

This is the dependent variable of DBMS thrashing. The variable captures a specific value of the MPL where DBMS thrashing first occurs in a given batchset. That is, it allows us to quantitatively measure the degree of thrashing observed in a DBMS.

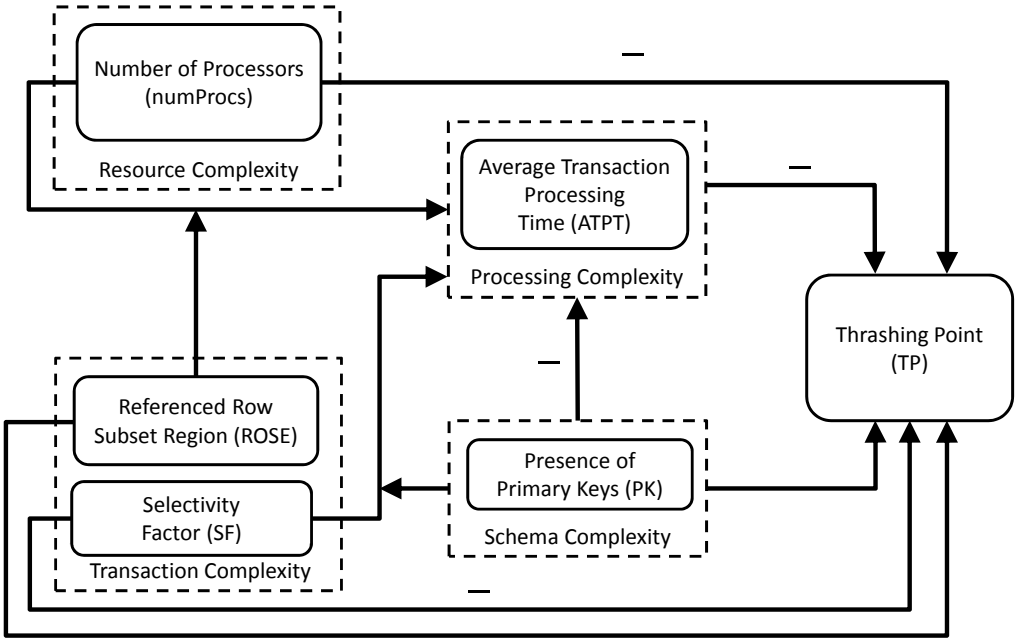


Figure 3: An Initial Model of Thrashing

4. An Initial Model of DBMS Thrashing

We now elaborate on an initial structural causal model of DBMS thrashing that includes the variables. We later refine this model in Section 8.

4.1. Model Description

Figure 3 exhibits an initial model of DBMS thrashing. This model helps us answer the second and third research questions presented in Section 1.2.

In the model we have the resource complexity construct described in Section 3.2.1. This construct abstracts system resources important to process transactions on DBMSes. The model includes this construct because of its significance on transaction throughput. In the construct we include the variable of the number of (logical) processors (numProcs), since numProcs is treated as one of the primary system resources, and it emerges as an important factor of thrashing as pointed by recent works [7, 8, 23].

Next, the model includes the transaction complexity construct described in Section 3.2.2. This construct is deeply involved in the characteristics of transactions causing throughput thrashing. The model therefore includes

this construct. In the construct we see the variables of reference row subset region (ROSE) and selectivity factor (SF), each representing contention among transactions and transaction size influencing DBMS thrashing [9, 12, 19, 20, 22]. These two variables concern transaction characteristics and thus are grouped into the construct together.

In the model the schema complexity construct is included. This construct includes one variable, PK, as described earlier (in Section 3.2.3). Primary keys can help DBMSes significantly reduce I/O in transaction processing. Because the PK variable concerns relational schema, we include the variable in this construct, capturing the influence of the schema on thrashing.

In the middle of the figure is the construct of processing complexity described in Section 3.2.4. The construct captures transaction processing capability of DBMSes, impacting thrashing. The model thus has this construct. In this construct we include the variable of average transaction processing time (ATPT). Transaction-blocking by lock conflicts delays transaction processing times [9], resulting in thrashing. ATPT thus is treated as an important factor of thrashing. It depends on some of the variables in the model.

The rightmost is the dependent variable of DBMS thrashing, represented by "thrashing point (TP)" in the model, as discussed in Section 3.2.5. An arrow from one variable (construct) to another indicates a correlation between two variables. The model includes some arrows above which a minus sign is marked, indicating a negative correlation between two variables, meaning that as a variable X increases, a dependent variable Y decreases.

In the model ATPT is dependent on some of the variables (numProcs and PK) on its left and exerts influence on the outcome variable (TP) on its right. The relationship among these variables—numProcs, PK, ATPT, and TP—in the model is called *mediation* [25]. In other words, mediation means 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 3, for instance, X corresponds to numProcs, Y to TP, and M to ATPT.

In addition, the model reveals that there is another relationship between numProcs and ATPT, which is *conditional* along with ROSE. The relationship among these variables—numProcs, ROSE, and ATPT—is called *moderation* [25], which means an association between two variables X and Y when the strength or direction depends on a third variable M. In Figure 3, for instance, 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.

	a	*b*	*c*
I	numProcs / ATPT *low*	numProcs / TP *medium*	N/A
II	SF / ATPT *high*	SF / TP *high*	ROSE / TP *medium*
III	PK / ATPT *high*	PK / TP *high*	
IV	ATPT / TP *medium*		

Table 1: Hypothesized Correlations and Levels on the Initial Model

Note particularly that the moderation concerns the mediation in the model. Such a relationship is known as *moderated mediation* [46]. The initial model expects that there will exist the mediation through ATPT from numProcs and from PK to TP, and in particular the mediation associated with numProcs will be conditionally influenced by or dependent on the variation in ROSE.

4.2. Hypotheses

Table 1 represents hypothesized correlations and levels between variables in the initial model. (A space of hypotheses drawn from the initial model is given and elaborated elsewhere [47].) Group I indicates the correlations involving the resource complexity construct with the processing complexity construct and with TP. Hypothesis I-(a) [48, 49] 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. Overall, the level of the correlation will be low.

Hypothesis I-(b) [7, 8, 23] indicates the correlation between numProcs and TP, and its strength is expected to be medium. As previously pointed out in the previous studies, processor contention is treated as one of the significant factors contributing to thrashing. Since our transactions are I/O

dominant, however, the effect of processors on thrashing will get offset by I/O. Thus, we expect the strength to be medium.

Group II shows the correlations involving the transaction complexity construct with ATPT and TP. Hypothesis II-(a) [9, 10] 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.

Hypothesis II-(b) [6, 9, 10] represents the correlation between SF and TP. The level of the correlation will be high, as SF will mainly determine the amount of I/O substantially influencing transaction throughput.

Hypothesis II-(c) [9] is the correlation between ROSE and TP. When ROSE is large, row contention among transactions will get reduced. The influence of ROSE on TP will thus be little. When ROSE is small, contention for the same rows will significantly rise, thereby leading to thrashing. In that case the strength will be high. The overall level will be in the middle, or medium.

Group III indicates the correlations of PK with the processing complexity construct and TP. Hypothesis III-(a) indicates the relationship 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. Hypothesis III-(b) represents the correlation between PK and TP. In the presence of PK, the DBMS can increase the overall throughput because of the saved I/O time. Thus, TP can be significantly increased. Therefore, we expect the level of the correlation to be high.

Group IV shows the direct association between ATPT and TP. Hypothesis IV-(a) [19, 20] predicts that the increase of ATPT will decrease the number of completed transactions within unit time, resulting in a significant drop in overall throughput. In contrast, the decrease of ATPT will help DBMSes finish more transactions within unit time. Thus, the overall throughput will go up, thereby delaying the onset of thrashing. Therefore, we expect the overall strength of the correlation to be medium.

The following section discusses operationalizing each variable in the model.

5. Variable Operationalization

The Number of Processors. The operationalization for this variable relies on machine specification. As described in Table 2, our individual experimental machine is configured with a quad-core processor chip supporting hyper-threading. We use four (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. In Linux, we can alter numProcs by setting a specific number to "`maxcpus`" available in `/boot/grub/grub.conf`. For Windows running SQL Server, numProcs can be altered by our proficient lab staff via a BIOS option.

Selectivity Factor. The way we operationalize the variable is to vary the number of rows accessed by each transaction. As mentioned before, 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, stating 1% as the maximum SF, we vary the SF from 0.01% up to 1% 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. The maximum of 1% is determined to be that needed to obtain a reasonable TPS value at an MPL. (It is not strictly necessary to cap the maximum SF at 1%, though SF greater than 1% may not be prevalent in real-world applications.)

A write-only transaction can be specified by an `UPDATE` statement. We don't use `INSERT` and `DELETE` for the write-only transaction, because the two statements change the cardinality of a table, which introduces other effects, thereby confounding the experiment. In contrast, `UPDATE` preserves the same semantics for operationalizing the write-only (hereafter, update-only) transaction, because it avoids a change in cardinality. The influence of insert/deletes on thrashing may be considered in the future work.

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 the linear variation of update-only

SF is sufficient 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. We operationalize this variable in the following manner. 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.

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:

Read-only trans.	Update-only trans.
`SELECT` *column_name*	`UPDATE` *table_name*
`FROM` *table_name*	`SET` *column_name* = *v*
`WHERE` *id1* >= *a* `and` *id1* < *b*	`WHERE` *id1* >= *a* `and` *id1* < *b*

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

For simplicity we use transactions on a single table, more complicated transactions over multiples tables are left as future work.

Presence of Primary Keys. This variable is a dichotomous variable (that is, primary keys are either present or absent). We can operationalize PK by telling a DBMS whether or not to treat `id1` as the primary key column of a table when populating the table.

Batchset Generation and Execution. As visualized in Figure 4, a batchset (termed in Section 3.1) contains a set of ten batches, each successively increasing by 100. Thus, the biggest batch in the batchset consists of 1,000

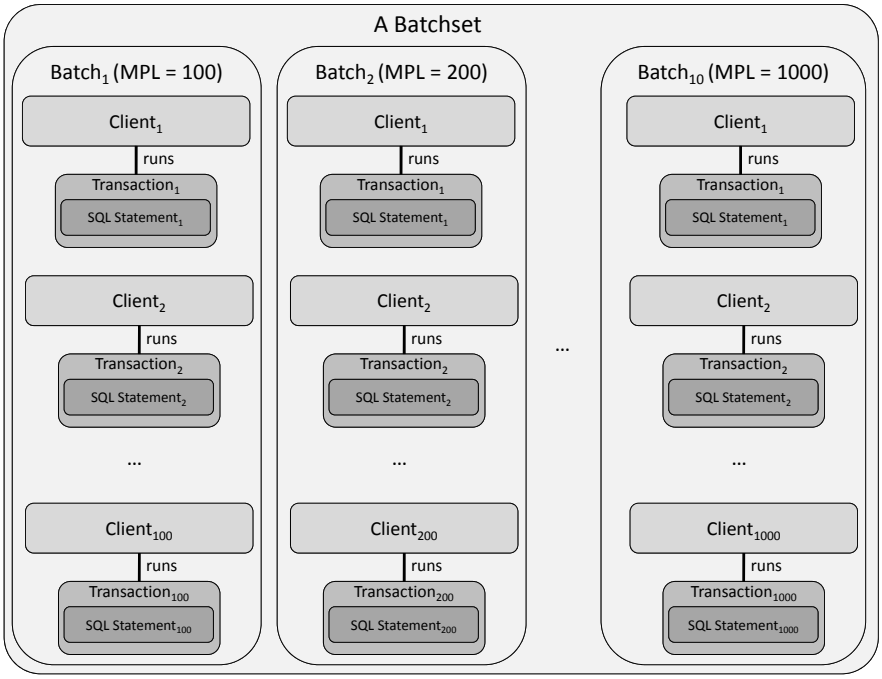


Figure 4: Batchset Visualization

clients. Each client in a batch has its respective transaction generated in the way mentioned in the paragraph of ROSE. Transactions used in our experiment as shown in Figure 1 use a simple database schema. 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 two character columns. Each table is populated with 1M (or 30K) tuples having column values randomly-generated except the first column (for the DBMS[3] running on Windows).

A prepared batchset is presented 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 until a CTL (connection time limit) value—two

[3]The above 30K cardinality was chosen only for this DBMS, as populating tables with 1M on that DBMS was not successful due to unknown system errors.

minutes—is reached, as described above in Section 3.1, and we make the same batch run multiple times for repeatability. In each batch run we 1) measure via JDBC and record the elapsed times of the transactions executed on that DBMS and 2) count the number of completed transactions. These two measured data are individually used to later measure the values of dependent variables introduced in Sections 3.2.4 and 3.2.5, whose operationalization is discussed in the following.

Transaction Response Time. This variable is captured in our model by *average transaction processing time* (*ATPT*). We cannot directly control the variable, but we can measure it. To operationalize (measure) this variable, as mentioned above we complete a run of a given batchset. We then average the recorded elapsed times for all the transactions executed in the batchset. The averaged elapsed time is then the ATPT 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 the calculated ATPT seems to not only reflect the impact of those schemes on TP but also captures different characteristics among the DBMSes with respect to code and performance.

Thrashing Point. To operationalize the thrashing point (*TP*), for a completed run of a given batchset we 1) compute the average transactions per second (TPS) for a batch, based on the recorded data of the number of executed transactions for multiple runs of that same batch, 4) calculate the average TPS for each of the remaining batches in the batchset in the same way, and finally 5) derive the TP for the batchset based on the following Equation 1:

$$\text{TP} = \begin{cases} |B_i| & \text{if } TPS_i > 1.2 \times TPS_{i+1}, \text{ and } \forall j > i+1, TPS_i > TPS_j \\ \infty & \text{otherwise,} \end{cases} \quad (1)$$

where $i > 0$, B_i is the i-th batch in a given batchset, and TPS_i is the measured TPS for B_i.

A specific TP 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 the values of TPS measured at all the subsequent batches in the same batchset. For instance, the TP of DBMS X in Figure 1(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 TP was higher than

the TPS at the last MPL of 1,000. In the figure that each point (or MPL), in steps of 100, corresponds the size of each batch in the batchset used in that experiment. Once thrashing begins, the observed thrashing should continue until the last MPL in a trial.

The calculated TP quantitatively represents how many MPLs can be tolerated by a DBMS for the batchset. The 20% threshold was determined based on our observation, such that once a measured TPS dropped by 20% or more than a previous TPS (at the TP), there was no case that subsequent TPSes were bigger than the TP's TPS.

In the following we describe a methodology for measuring DBMS thrashing as TP, to collect empirical data, necessary to test the aforementioned hypotheses in Section 4.2.

6. Thrashing Metrology

Metrology is the science of weights and measures or of measurement [50]. In this section we discuss the overall thrashing metrology for measuring DBMS thrashing based on the variables described in the previous section. The metrology includes a thrashing observance scenario, experiment infrastructure, and data analysis protocol.

Before elaborating on the scenario, we introduce the configurations of our system used for the experiments in the next.

6.1. Experiment Configuration

Table 2 summarizes the system configuration used in the experiments in our study. Our experimental machines are of identical hardware (H/W): i) a quad-core processor chip supporting hyper-threading, ii) 8GB RAM, and iii) 1TB SATA HDD with 7200 RPM. Each machine runs Red Hat Enterprise Linux (RHEL) 6.4 with a kernel of 2.6.32, except one running Windows Server 2008 dedicated for one of the DBMSes. Each machine exclusively runs one of the five DBMSes.

This machine configuration represents a single value for these controlled independent variables (whereas in Section 5 we listed other independent variables, which are each given several values). While we do not vary the system configuration, we feel that other configurations (thus, other values for these independent variables) could exhibit similar predicted behavior. Thus our causal model predicts certain interactions and certain behaviors (values) of

System Specification			
H/W	CPU	Intel Core *i*7-870 Lynnfield 2.93GHz LGA 1156 95W Quad-Core Processor (in one socket)	
	RAM	Kingston ValueRAM 8GB (4GB (2 × 2GB) 240-Pin DDR3 SDRAM DDR3 1333 (PC310600) Dual Channel)	
	HDD	Western Digital Caviar Black WD1001FALS 1TB 7200 RPM SATA 3.0Gb/s 3.5" Internal Hard Drive	
S/W	OS	Linux	RHEL release 6.4 Santiago (kernel: 2.6.32)
		Windows	Windows Server 2008 R2 Enterprise 6.1
	DBMS	MySQL 5.5.8	
		PostgreSQL 8.4.9	
		three proprietary DBMSes X, Y, and Z	

Table 2: Environment Setup

the dependent variables as impacted by the specific settings of the independent variables. We mention in passing that we assume the configuration DBMSes i) support a concurrency control model of locking or snapshot isolation, ii) rely on a filesystem (rather than direct I/O), and iii) are able to support up to 1,000 connections (threads).

6.2. The Thrashing Observance Scenario

Figure 5 depicts our thrashing measurement scenario, the structure of our experiments running a hierarchy of eight levels, ending at a particular execution of a particular batch (at an MPL) of a particular batchset for the underlying table(s), as part of a particular experiment run started on a stated date and time on a designated machine using a specified release of a specific DBMS, in the context of a specified experiment setup (stating batchsets, the characteristics of the data, and various other parameters), of a selected experiment scenario.

For each batch execution (during the two-minute CTL), the number of completed transactions is counted to derive transactions per second (TPS). We make five batch executions for each batch. For each batch TPS gets averaged among the batch executions.

Scenario (e.g., Thrashing)
Experiment (batchsets, data spec, scenario parameter(s))
DBMS (DBMSes X, Y, and Z, PostgreSQL, and MySQL)
Machine (`sodb6`, `sodb8`–`sodb11`)
Experiment Run (on a particular date)
Batchset Instance (batchset number within batchsets)
Batch Instance (an MPL of 100–an MPL of 1,000)
Batch Execution (1–5)

Figure 5: Hierarchy of Batch Executions

As an example of this hierarchy, each point in Figure 1 corresponds to the averaged TPS at each MPL. For the data in Figure 1, we utilized the Thrashing Scenario, an Experiment specifying a nineteenth set (BS19) of 28 batchsets, specifying data with a cardinality of 1M (30K) rows, and specifying the scenario parameters of 1) five executions per batch, 2) one core with hyper-threading disabled, and 3) declared primary keys. We ran this batchset on DBMSes X, Y, Z, MySQL, and PostgreSQL on the `sodb9`, `sodb8`, `sodb6`, `sodb11`, and `sodb10` machines from experiment runs for batchset number 19, respectively, thus identifying a particular batchset instance, and examining all ten batches (MPLs).

The Thrashing scenario is a plug-in that we added to a laboratory information system, called AZDBLab [28].

6.3. AZDBLab *Infrastructure*

AZDBLab is a DBMS-oriented research infrastructure, supporting large-scale scientific studies across different DBMSes [28]. It has been developed for more than seven years by many people's contribution. AZDBLab is operated on a hardware lab of dedicated machines: one each for each DBMS subject and one to host the central DBMS server to store data collected from the DBMS subjects. Currently, a total of seven relational DBMSes are supported, both open source and proprietary. The infrastructure has been sufficiently robust to collect empirical (query execution) data on the Linux and Windows operating systems, totalling over 9100 hours (more than one year, 24x7) of cumulative time.

6.4. Running Experiments

Leveraging AZDBLab, we made tens of experimental runs of the Thrashing scenario. To run the scenario, we used five relational DBMS subjects available in AZDBLab, as shown in Figure 1. (The remaining two DBMSes could not be used because of some technical issues that emerged on those two while running the scenario.)

While running the experiments AZDBLab collected each batchset's transaction elapsed time and TPS data measured on the DBMS subjects.

Our experience studying varying query time [51, 52] and query suboptimality [53] is that AZDBLab facilitated developing and running the thrashing scenario with multiple configurations of the experiments. (That said, this does not mean that AZDBLab is a requirement for this thrashing study.) We plan to make AZDBLab in public sometime in the next year after enhancing the system.

Before proceeding with the evaluation of the model using the measured data, we asked two related questions: How could we make sure that the data were clean enough to proceed with the evaluation of the model? And is there a way of assessing the validity of the acquired data? To address this underlying concern, we designed a sophisticated data analysis protocol.

6.5. Thrashing Analysis Protocol

The protocol, termed *Thrashing Analysis Protocol* (*TAP*), performs a suite of sanity checks on the measured data, drops data failing to pass the checks, and then computes the TP (thrashing point) for each batchset in the retained data. TAP consists of the four steps.

In Step 1 TAP conducts eight 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 four sanity checks. (1) The *number of missing batches* indicates how many batch instances (BIs) that were supposed to have been executed with a batchset (10), but were not, for whatever reason. (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. (3) The *number of missed BE violations* represents how many BIs did not have the specified number of repetitions required. (4) 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.

The second class of sanity checks concerns batch executions (BEs). Each of these three sanity checks could encounter only a few isolated violations, but we expect the violation percentage to be low. (5) The *zero TPS violation* catches BEs where TPS was zero at the starting MPL in their BSIs. (6) The *connection time limit violation* identifies the BEs that violated the specified CTL. (7) The *abnormal ATPT violation* indicates how many BEs revealed abnormally high or low ATPT.

The final class involves one check over a BSI. (8) The *transient thrashing violation* examines how many BSIs experienced transient thrashing, which indicates that the TPS measured at a determined TP is lower than any of the TPSes measured at the MPLs bigger than the TP. Once the thrashing phase begins, it will be rare to observe that the throughput rises back up at greater MPLs.

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 ((5), (6), and (7)), and remove those BEs from further consideration.

In Step 3 TAP looks at the retained BEs for each BSI and then determines 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 is 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.

In Step 4 TAP calculates the TP for each of the retained BSIs along with Equation 1. After finishing Step 4, we proceed with conducting (exploratory or confirmatory) evaluation on the model with the measured data: the retained BSIs and their corresponding TPs.

Now we discuss the exploratory analysis on the initial model.

7. Exploratory Model Analysis

In this section we summarize the experimental data used for the exploratory analysis and present the evaluation results on the initial model.

7.1. Descriptive Statistics

For the exploratory evaluation we used the aforementioned five relational DBMSes. We operationalized four different numProcs values and two values (that is, presence and absence) for PK. As exhibited in Table 3, we completed

	Exploratory	Confirmatory
Number of Experiment Runs	40	50
Number of BatchSet Instances (BSIs)	1,120	1,400
Number of Batch Instances	11,200	14,000
Number of Batch-instance Executions (BEs)	33,600	70,000
Total Hours	3,127	5,170

Table 3: Desc. Stat. for the Exploratory and Confirmatory Exp. Runs

40 experimental runs, each taking a few days to a week on a single machine (with the disk drive humming the entire time), totaling 3,127 hours (about 4 months) of cumulative time.

Considering that each of the 40 completed runs contained 28 batchsets (operationalized by a combination of four ROSE values and seven SF values), we had a total of $28 \times 40 = 1{,}120$ BSIs. As each BSI had ten batches, we had $1{,}120 \times 10 = 11{,}200$ BIs. As each batch was executed three times, we had 11,200 BIs $\times$ 3 = 33,600 BEs. In short, our exploratory study concerns the 40 completed runs, involving 33,600 BEs.

TAP was applied to the 33,600 BEs. In Step 1 we identified all the BEs revealing problems. We then dropped in concert 2,507 BEs (about 7.46%) in Step 2. In Step 3 we subsequently dropped in concert 109 BSIs (about 9.8%) (from 1,112 BSIs calculated based on the BEs retained after Step 2). In effect, a total of 1,003 BSIs survived throughout Steps 2–3. For the retained 1,003 BSIs we calculated the TPs based on Equation 1 in Step 4. As a result, we detected a total of 487 thrashing BSIs from the five DBMSes.

Several conclusions from the thrashing BSIs can be drawn. First, every DBMS exhibited thrashing. The number of thrashing BSIs differed by about a factor of three among the five DBMSes. The DBMS thrashing phenomenon thus must be a fundamental aspect of either the algorithm (concurrency control), the creator of the algorithm (human information processing), or system operation context (environment). Note that our model covers all of these effects. 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 under MPL 300.

The thrashing 487 samples consisted of a group of 188 read batchsets (hereafter, *read group*) and a group of 299 update batchsets (hereafter, *update*

group). Our evaluation was performed separately on these two groups.

7.2. Exploratory Evaluation Results

In this section we discuss the statistical analysis results on the initial model in Figure 3.

7.2.1. Correlational Analysis

We performed correlational analysis using `cor.test`() of R [54]. We tested hypothesized correlations and levels exhibited in Table 1.

For the read group, I-(a), III-(a), III-(b), and IV-(a) were supported while II-(a), II-(b), and II-(c) were not supported. I-(b) was not supported, because the actual direction of its correlation was positive while the hypothesized direction was negative as depicted in Figure 3. In the update group I-(a), I-(b), and IV-(a) were supported whereas II-(a), II-(b), and II-(c), and III-(a), III-(b) were not supported. It seemed that the variables of SF and ROSE had some unknown challenges in empirically observing correlations with ATPT and TP.

7.2.2. Causal Mediation Analysis

We performed causal mediation analysis [27] via `mediate`() in the `mediation` package in R. We tested the mediation through ATPT to TP, moderated by ROSE and PK. In both groups the moderated mediation was not significant, perhaps because of ROSE and SF that were not significant. However, the mediation through ATPT to TP from numProcs was significant, as was that from PK in the read group and from numProcs in the update group.

7.2.3. Regression Analysis

The performance of the model as a whole was evaluated via the value of *R-squared* (R^2) [25], the ratio of explained variance to sample variance of a dependent variable of the model. The computed R^2 tells us how well the model can fit the measured data. If the value of R^2 is close to 1 (100% variance), all contributing factors are identified. 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.

Our regression analysis was conducted using multi-linear regression, a well-known statistical tool for modeling the relationship between a continuous dependent variable and multiple independent (explanatory) variables that are

continuous or dichotomous [25]. MLR was a perfect fit for the initial model, satisfying the variable condition for the application of MLR. (Testing the assumptions of MLR is discussed in Section 9.2.)

Using MLR, we first performed regression on TP. We used `lm`() of R and examined the overall fit of the model on each group. The amount of variance explained (R^2) for TP was 11% (14%) for the read group (update group). We also performed regressions on ATPT that has dependency on the variables in the model. The initial model explained 12% (11%) of the variance of ATPT for the read (update) group.

7.2.4. Path Analysis

We performed the path analysis on the initial model, to determine which paths were statistically significant, using regression coefficients provided by the outcome of `lm`().

In the read group the significant paths were (a) from numProcs to ATPT, (b) from PK to ATPT, (c) from numProcs to TP, (d) from ATPT to TP, and (e) from PK to TP in Figure 3, and their weights were (a): −0.19, (b): −0.30, (c): 0.11, (d) −0.17, and (e): 0.25, respectively.

On the other hand, there were not significant paths: (i) from ROSE to TP, (ii) from SF to TP, and (iii) from SF to ATPT, in addition to (iv) from numProcs to ATPT, moderated by ROSE, and (v) from SF to ATPT, moderated by PK. We thought we understood these paths, and thus we included them in the initial model. However, it seemed that the operationalization of ROSE and SF unexpectedly incurred confounding behavior. In future work, other ways of operationalizing these variables can be further explored, and we discuss possible improvements in Section 7.3.

To continue our model exploration, we performed a regression analysis on the reduced model, removing the confounding paths. Consequently, all the retained paths in the reduced model were significant. Their weights were (a): −0.12, (b): −0.29, (c): 0.13, (d) −0.17, and (e): 0.22, respectively. There was little difference between the corresponding weights of the full (initial) and reduced models. In other words, there was little discrepancy between the full and reduced models. This implies that the reduced model did not fit the data any worse than the full model including the eliminated paths.

In the update group the significant paths were (f) from numProcs to ATPT, (g) from numProcs to TP, and (h) from ATPT to TP in Figure 3, and their weights were (f): −0.29, (g): −0.45, (h): −0.11, respectively.

The above insignificant paths in the read group also resulted in being not

significant in the update group, plus the paths involving PK. Again, we included all these paths in the initial model for the update group, as we thought we understood them, which was not true. The better operationalization of PK as well as ROSE and SF need be investigated in the future work.

As did in the read group, we performed a regression analysis on the reduced model, eliminating the paths that were not significant. As a result, the reduced model yielded no insignificant paths. The respective weights of (f), (g), and (h) were $-.30$, -0.43, and -0.10. Both the full and reduced models did not reveal statistical discrepancy. Hence, the reduced model did not fit the data more poorly than did the full model on the update group.

7.3. Lessons

We have learned several things from the preliminary study on the initial model. First, some variables were not statistically significant. Specifically, the associations of ROSE and SF with ATPT and TP (thrashing point) were not significant for the read and update groups. In addition, the correlations involving PK were not significant in the update group. There might be a general lack of understanding of those variables or perhaps their operationalization.

Here are some possible suggestions to better operationalize the variables. Concerning ROSE one way would be to have each transaction in a batch reference a contiguous chunk of table rows in sequence within a specified region (by a ROSE value). If a transaction reaches the last chunk of rows in the region, then the next transaction could access the first chunk of rows at the beginning of the region. If so, any row in that region would be referenced with equal probability by the transactions. Another would be to have every transaction access the same number of discrete rows (using '`IN`' predicate) with even access probability. For SF, it would be possible to consider extending the value range from the minimum cardinality (or a single row) to the maximum cardinality (or 1M rows) of the table. It is hard to see a potential problem with operationalizing PK, because it is too obvious.

Also, it would be worth exploring the existence of other variables perturbing the associations involving ROSE, SF, and PK (only in the update group).

Second, the read and update groups showed differences in the observed correlations. 1) The direction of a correlation between the same corresponding variables was not consistent between the read and update groups. The correlation of numProcs with TP was positive in the read group whereas it

was negative in the update group. 2) PK was not significant in the update group while it was significant in the read group. 3) The amount of variance explained by the model for the read group was different than that of the update group.

Third, we observed across the DBMSes different thrashing behaviors between the read and update groups. Specifically, for DBMS X the average TP of the update (read) group was about 614 (870), and for DBMS Y, it was about 266 (462) in our exploratory experiments. The other DBMSes also revealed a similar tendency: the overall thrashing on the update group occurred earlier, perhaps because of heavier locking overhead.

For these reasons, we concluded that our model should be divided up into two parts, explicating the respective thrashing origins for the read and update groups, resulting in the refined model to be discussed now.

8. A Refined Model of DBMS Thrashing

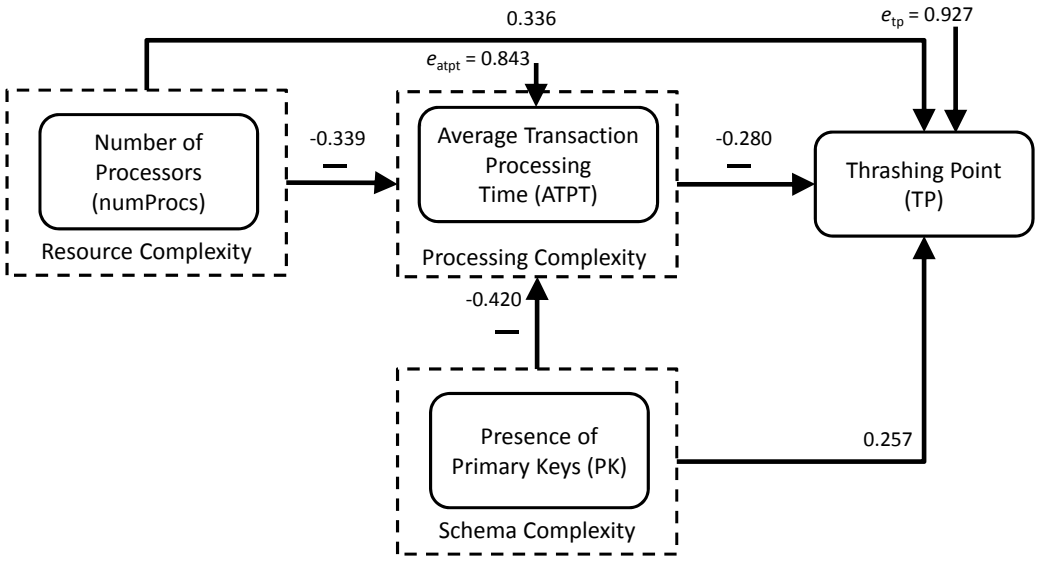
In this section we give a refined model of DBMS thrashing and present and discuss in detail a space of hypotheses related to the refined model. The refined model is depicted in Figure 6. The specific values above arrows result from our path analysis to be discussed in Section 9.3.4. We elaborate on the values in that section.

8.1. A Refined Model for the Read Group

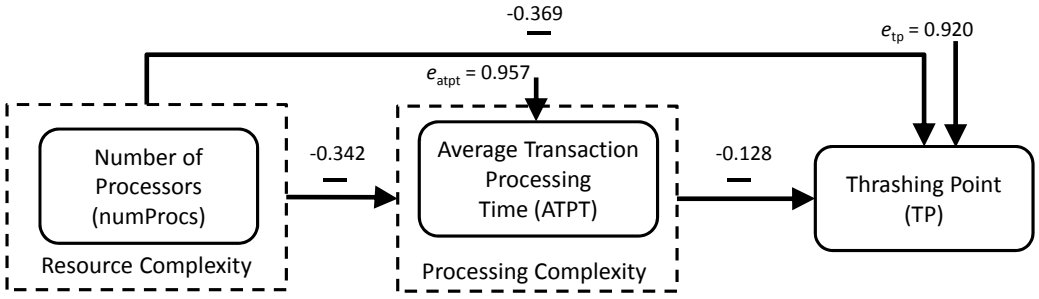
As seen in Figure 6(a), the variables of ROSE and SF and their associated associations were removed because we do not adequately understand how they interact with other variables of interest. The direction of the correlation between numProcs and TP was changed from negative to positive because we observed that the benefit of utilizing more processors for the read group dominated the expected contention among the processors, we feel due to much less locking overhead. So we now expect the direction to be positive.

Let's examine the set of hypotheses drawn from the refined model in Figure 6. Such relationships are specific associations between the constructs (or, their variables), as hypothesized by this predictive causal model.

One factor of the read model is the resource complexity construct. We hypothesize that this construct affects DBMS thrashing both directly, as depicted by the top line, and indirectly, via ATPT.



(a) The Model of the Read Group (Read Path Model)



(b) The Model of the Read Group (Update Path Model)

Figure 6: A Refined Model of DBMS Thrashing (Path Diagrams on the Batchset Groups in the Confirmatory Experiment)

Consider numProcs first. Multicores present a great opportunity to DBMSes [8, 23]. Multi-core processors enable DBMSes to process more transactions than a single processor does, due to increased parallelism. The previous studies [8, 23], however, reported that DBMSes experienced thrashing by multicores because of core contention, but their studies were conducted on one specific DBMS as mentioned in Section 2. In general, transaction processing in most DBMSes can be sped up by using multi-core processors. We thus hypothesize the following.

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

As mentioned before, the increased parallelism using more processors can speed up transaction processing. Although overall throughput could fall off due to contention among the processors, we anticipate that as far as individual transaction response time is concerned, it will be shorter when more processors are presented.

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

Concerning processing complexity, 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.

Hypothesis 3: As ATPT increases, the thrashing point will decrease (i.e., thrashing will occur earlier).

We hypothesize that in the presence of PK, TP will increase. PK allow DBMSes to avoid doing full scans on the tables referenced by transactions. When PK exists, a DBMS can thus speed up processing transactions accessing PK columns. As a result, the overall transaction throughput will increase, leading to an increase of TP as well.

Also, we hypothesize that ATPT will decrease if PK exists. Considering that I/O is one of the significant components in transaction processing, DBMSes can significantly benefit from a primary index automatically created when the PKs 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. That is, the slope between SF and ATPT will be lower when PK is present.

Hypothesis 4: In the presence of primary keys the thrashing point will increase.

Hypothesis 5: In the presence of primary keys, ATPT will decrease.

8.2. A Refined Model for the Update Group

ROSE and SF were eliminated in both portions of the model. PK was also removed, with its associated correlations. In comparison with Figure 6(a), the direction of the association between numProcs and TP was retained because of the supported hypothesis on the association.

Now let's discuss the hypotheses drawn from the update model. In contrast to the read model, we focus on a drawback in taking advantage of multi-core processors in update transaction processing. Enabling more processors could incur serious competition among the processors in utilizing shared resources available within the DBMS internals. Increased core contention can hurt transaction throughput, or slow down transaction processing.

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

Concerning the processing complexity construct, we expect that the more processors, the faster ATPT in the update group, because of the same reason described in the read model.

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

For the same reason described in the read model, we expect that the longer ATPT, the shorter TP.

Hypothesis 3: As ATPT increases, the thrashing point will decrease (i.e., thrashing will occur earlier).

8.3. The Revised Hypothesized Correlations and Levels

Table 4 exhibits the eight hypothesized correlations on the refined model. The corresponding levels, separated by '/' for the read/update groups, are shown in parentheses. Note that compared to Table 1, group II no longer exists because the variables of ROSE and SF and relevant correlations are removed in the refined model. In the read group every cell is as in Table 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 initial model.

	a	*b*
I	numProcs / ATPT (*low* / *medium*)	numProcs / TP (*low* / *low*)
III	PK / ATPT (*medium* / N/A)	PK / TP (*medium* / N/A)
IV	ATPT / TP (*medium* / *low*)	

Table 4: Revised Hypothesized Corr. and Lev. on the Read/Update Groups

In the update group we do not have group III because PK is not present in the refined model. The levels in group III are thus denoted as 'N/A.' Also, the level of Hypothesis IV-(a) is decreased to low from medium compared to Table 1, as we saw that this correlation was weaker than we thought in the initial model.

9. Confirmatory Model Analysis

In this section we provide descriptive statistics on the confirmatory experiment runs and present statistical analysis results on the refined model.

9.1. Descriptive Statistics

For the confirmatory evaluation, we used the same five relational DBMSes and kept the same operationalization except increasing one more value for numProcs. As previously provided in Table 3, we performed 50 experiment runs, totaling 5,170 hours (about 7 months) of cumulative time. Note that this data was completely separate from that used in the exploratory evaluation discussed above.

We had a total of 28 (batchsets) $\times$ 50 (runs) = 1,400 BSIs. As each BSI had ten batches, we had 1,400 $\times$ 10 = 14,000 BIs. As each batch was executed five times, we had 14,000 BIs $\times$ 5 = 70,000 BEs. In short, our confirmatory study concerns the 50 completed runs, involving 70,000 BEs, more than twice as many BEs as the exploratory study had.

As was done in the exploratory study, TAP was applied to the 70,000 BEs. Tables 5, 6, and 7 exhibit the step-wise TAP sanity check results. Table 5

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 5: Experiment-Wide Sanity Checks in the Confirmatory Evaluation

6	*Percentage of Zero TPS*	0.036% (25/70000)
7	*Percentage of Connection Time Limit Violations*	3.5% (2434/70000)
8	*Percentage of Abnormal ATPT*	0.97%(679/70000)

Table 6: Batch Execution Sanity Checks in the Confirmatory Evaluation

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

Table 7: A Batchset Instance Sanity Check in the Confirmatory Evaluation

shows no 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.

In the BE sanity checks shown in Table 6, we had 0.036% zero TPS violations, 3.5% connection time limit violations, and 0.97% abnormal ATP violations. These rates were very low. As exhibited in Table 7, we had no violations of (9) concerning the batchset sanity check. Our confirmatory data passed all of these sanity checks.

At Step 2 we drop the BEs identified as violation in Table 7. Table 8 shows how many BEs were valid at the beginning of Step 2 and after Step 2: about 4.39% BEs were dropped. Table 8 also exhibits how many BSIs were dropped after each sub-step in Step 3. As indicated 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 7. Step 3-(iii) again dropped BSIs revealing transient thrashing among the remaining BSIs until this step. No BSIs were dropped. In sum, we dropped in concert about 4.4% BSIs throughout Step 3.

At Step 4, we computed the TP for each of the retained BSIs in the same

At Start of Step 2	70,000 BEs
At Start of Step 2	1,400 BSIs
After Step 2	66,927 BEs (4.4% 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 (4.4% dropped)

Table 8: The Number of Batch Executions and Batchset Instances after Each Sub-Step

way as done with the exploratory experiment. We detected thrashing in a total of 482 samples, consisting of 148/334 samples from the read/update groups. The percentage of the thrashing batchsets varied decreased to 36% from 49% in the exploratory experiment.

We reaffirmed in the confirmatory experiment that 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.

9.2. Testing Assumptions of Multi-Linear Regression

Before fitting the model via multi-linear regression (MLR), we need to check several assumptions required by MLR. One key assumption is *independence of residuals*. This assumption expects that the residuals should be uncorrelated serially from one sample to the next; namely, the level of error should be independent of when a sample is collected (time dependency). (Note that patternless, random residuals imply independence of errors.)

The assumption was tested via the Durbin-Watson (D-W) statistic [55]. We employed `durbinWatsonTest()` in R [54] to test the assumption against our confirmatory data. The resulting D-W scores of TP were 1.4 and 1.1 for the read and update groups. For the data to not indicate auto-correlation, we need a score of two. But the computed scores indicated some type of positive correlation among the residuals in our data. We also computed the D-W scores of ATPT but still observed such a positive correlation.

Indeed, this violation was somewhat expected, in that 1) a limited number (e.g, only five) of DBMS subjects were used in our experiment, and 2) the overall violation results from only one (or at most two) of the DBMSes that had violating samples. To further look into this violation, we also calculated

the per-DBMS D-W statistics for TP. As a result, we identified several possible issues, including 1) per-DBMS low sample size, 2) some DBMSes revealing auto-correlated samples, and 3) clustered TP and ATPT across DBMSes. Nevertheless, in general the violation did not invalidate the model [55]. (In future work more samples per DBMS are needed and then checked to 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 [56] may be applied.)

Another key assumption is *homoscedasticity*. This assumption tests whether the variance of residuals is constant along the line of the best fit. `ncvTest`() of R was used for testing this assumption. The p-values of the test were 0.20 (0.47), greater than 0.05, for the read (update) group, which satisfied the homoscedasticity assumption.

The third key assumption is *multicollinearity*. This assumption tested is whether two or more independent variables are highly correlated with each other. `vif`() of R yielded variance inflation factors (VIFs) on the data. The outcome of `vif`() showed that none of our independent variables had the square root of its corresponding VIF higher than 2, meaning no high correlation of the independent variables. Our data thus satisfied the multicollinearity assumption.

The next key assumption is *no outliers*. We can test by using Cook's Distance (CD) [57]. We used `cooks.distance`() of R to test this assumption. The data did not show any significant outliers because the CD values were fewer than 1.

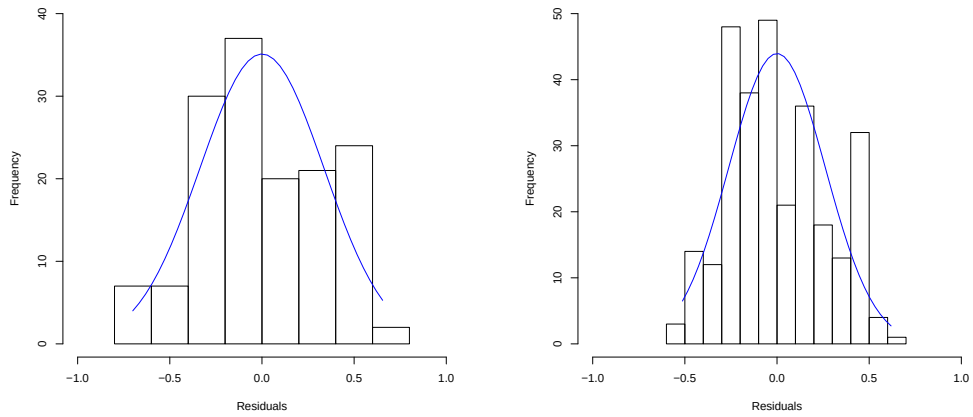
The last key assumption is (5) *normality of residuals*. This assumption indicates that the residuals should be approximately normally distributed; that is, the observed and predicted values should come from the same distribution [58]. A model satisfying this assumption is expected to predict values higher than actual and lower than actual with equal probability.

The assumption was tested by obtaining the histogram of the residuals and superimposing a normal curve on that histogram for each group as shown in Figure 7. The residuals of our data appeared to follow very approximate normality.

All things considered, our confirmatory data appears to satisfy the MLR assumptions.

9.3. Confirmatory Evaluation Results

We now evaluate the refined model in Figure 6.



(a) Read-only Batchset Group (b) Update-only Batchset Group

Figure 7: Testing Normality of Residuals on the Confirm. Eval. Data

Variable	*TP*	*ATPT*
numProcs	H1: 0.31	H2: -0.32
ATPT	H3: -0.26	—
PK	H4: 0.21	H5: -0.55

Table 9: Testing Hypotheses 1–5: Correlations on the Read Group

9.3.1. Correlational Analysis

We conducted the confirmatory pair-wise correlational analysis on the refined model.

Table 9 exhibits the computed correlation factor associated with each hypothesis on the read group. All the hypotheses (H1–H5) were supported and significant. The actual levels of H1 and H5 were consistent with our predictions in Table 4. The actual levels of H2, H3, and H4 were medium, low, and low while the predicted levels of low, medium, and medium, respectively. Note that the actual and predicted levels were very close. Such a minor difference was acceptable. Thus, the direct correlations of the read model were empirically confirmed by these correlational analysis results.

Table 10 shows the computed correlation factor associated with each hy-

Variable	*TP*	*ATPT*
numProcs	H1: -0.36	H2: -0.16
ATPT	H3: -0.13	—

Table 10: Testing Hypotheses 1–3: Correlations on the Update Group

pothesis for the update group. As for the read group, all the hypotheses (H1–H3) for the update group were supported and significant. The actual levels were consistent with the predicted levels by the model. Therefore, the direct correlations of the update model were also empirically confirmed by the correlational analysis results shown in Table 10.

9.3.2. Regression Analysis

We performed a regression over the independent variables of the refined model that predicts TP over the data obtained from the confirmatory experiment. Specifically, we computed the overall fit of the refined model for 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. For the update group our model explained 15% of the variance of DBMS thrashing. These regression results showed that the refined model could to some extent successfully explicate the source of variance on TP.

We also conducted a regression on ATPT that is dependent on some of the independent variables in the model. The refined model explained 29% of the variance for ATPT on the read group. The removed variables and associations did not decrease the variance explained. We found that in the confirmatory experiment most of the DBMSes agreed with the direction of the correlations of numProcs and PK with ATPT. We suspect that such an agreement made a contribution to the high variance explained.

In contrast, the amount of variance explained by the model for the update group was 8%. We attribute the decrease to the variables and associations removed from the exploratory model.

9.3.3. Causal Mediation Analysis

We conducted causal mediation analysis on the refined model, 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 TP. For the update group, we also verified the statistical significance of

the mediating effect of numProcs via ATPT on TP. These results demonstrate that the mediation via ATPT on TP in the refined model was empirically verified for each group.

9.3.4. Path Analysis

We performed the path analysis for the refined model in Figure 6, to determine what paths in the model were statistically significant, using the path coefficients from the regression outputs of the read and update model fits. We computed the *total* (indirect + direct) effect on each path in the model.

Regarding the read group, all the paths were significant as shown in Figure 6(a). The direct effect (-0.42) of PK to ATPT was bigger than that (-0.34) of numProcs to ATPT. In addition, the total effect of numProcs on TP was 0.43 ($= -0.34 \times (-0.28 + 0.34)$), which was greater than that ($-0.42 \times (-0.28 + 0.26)$ of PK and that (-0.28) of ATPT on TP.

For the read group, PK and numProcs were the most significant factors on ATPT and TP in the model, respectively. In the confirmatory experiment we added one more value (or, six processors) in the operationalization of numProcs. This perhaps may have contributed to the increased significance. More investigations about the increase are needed. That said, we reaffirm that numProcs and PK had positive correlations with TP, and ATPT also had a negative correlation with TP.

Concerning the update model, all the paths were significant, as illustrated in Figure 6(b). The total effect of numProcs to TP was $-0.34 \times (-0.13 - 0.37) = -0.33$, which is stronger than that (-0.13) of ATPT to TP. For the update group, it was numProcs that showed the most significant factor on both ATPT and TP. We reassert that numProcs and ATPT had negative correlations with TP in the refined model.

In Figure 6, the error variances of ATPT (e_{atpt}) and TP (e_{tp}) were 0.84 and 0.93 for the read group and 0.96 and 0.92 for the update group, each computed by $\sqrt{(1 - R^2_{atpt})}$ and $\sqrt{(1 - R^2_{tp})}$. There is much room to improve the model, such as exploring and then examining more variables and associations to further explain the thrashing variance. That said, our model is a contribution, as it has already identified the significance of several variables (that is, the number of processors, presence of primary keys, and average transaction processing time) and their associations that have a definite impact on thrashing.

Lastly, the overall fits of the read and update path models were estimated as follows.

$$\begin{aligned} \text{Fit for the read model} &= 1 - e_{atpt}^2 \times e_{tp}^2 = 1 - 0.84^2 \cdot 0.93^2 = 0.39 \\ \text{Fit for the update model} &= 1 - e_{atpt}^2 \times e_{tp}^2 = 1 - 0.96^2 \cdot 0.92^2 = 0.23 \end{aligned} \quad (2)$$

The fit (0.39) of the read model is lower than that (0.23) of the update model. This difference was primarily because the variance explained by ATPT was higher in the read model than in the update model. To increase the fit of the update model, the amount of variance explained by ATPT should be improved by looking into other possible variables and their correlations that have significant impact on ATPT.

The estimated fits also indicate that the update model has more room for improvement than the read model does. As mentioned above, it would be advantageous to explore other variables and investigate their causal relationships on TP. Again, our model provides a good basis for such variable and correlation exploration.

In sum, all the confirmatory evaluation results provide empirical support for the novel structural causal model of DBMS thrashing in this article that we introduce here.

10. Engineering Implications

While developing the refined 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.

- Every DBMS used in our experiment exhibited thrashing. Previous studies [8, 23] pointed out that some open-source DBMSes such as PostgreSQL and MySQL experienced thrashing. We show that proprietary DBMSes are also vulnerable to thrashing.

- Thrashing in DBMSes occurred extensively over increasing MPLs. In particular, some DBMSes experienced early thrashing, under an MPL of 300. It was surprising to see that DBMSes were sometimes overwhelmed by very simple read-only (update-only) including just select and projection (update) clauses.

- Lastly, thrashing was not repeatable in DBMSes. There were two flavors of DBMS non-repeatability that we observed. The first is "intra-non-repeatability": given the same batchset thrashing was not always observed in DBMSes. Specifically, for a particular batchset the DBMSes thrashing in the exploratory experiment but not in the confirmatory experiment, and vice versa. The second is "inter-non-repeatability": given the same batchset, thrashing was experienced by some DBMSes but not by others. We found that query optimizers of some of the DBMSes revealed a heuristic nature of execution plan selection. We therefore suspect that the non-repeatability characteristics is associated with such a heuristic.

These results suggest that research on thrashing should be pursued in conjunction with multi-core architectures. Fortunately, the causal model in Figure 6 helps point out specifically where that research might be focused.

Our model implies some engineering implications for DBAs and for DBMS researchers and developers, for better understanding and properly coping with thrashing. (These implications are complementary to the practical lessons already given in a previous literature [59].)

1. For read-only workloads, we have the following general advice.
    - DBAs should enable multi-core processors. DBMSes can utilize their parallel processing ability when serving the workloads. In our experiment DBMSes showed increased TP (thrashing point) as more processors were available. DBAs thus may consider exposing to DBMSes as many processors as available.
    - 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 approaches to improve I/O, including 1) increasing physical memory, 2) creating secondary indexes, 3) enabling caches from disk drive through file system up to DBMS buffer, and 4) replacing traditional disks with solid-state drives (SSDs) for reducing access latency [60]. These approaches

can delay thrashing, or perhaps even prevent it in some situations. In the future work the effectiveness of each of these approaches could be evaluated.

- DBAs should pay attention to response time of transactions. Another key factor in increasing TP was to shorten transaction response time. Our experiment showed that TP decreased as ATPT increased. 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 suggest using as many processors as possible and specifying primary keys on the tables referenced by the transactions, as mentioned 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, one DBMS may thrash whereas another didn't. This implies that DBAs should check their specific transaction management and tuning parameters such as the maximum number of connections and size of shared buffer.

2. When treating update-only workloads, we advise the following.

   - Using many processors may not help increase TP for update-only workloads. We observed in our experiment that as the number of processors increased, TP decreased, 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. When multi-core processors are employed, DBMSes may struggle to tolerate inter-processor contention on the synchronization constructs related to exclusive lock management. DBAs thus should be aware of processor contention caused by more provisioning, when serving update-only workloads.
   - When thrashing is observed, DBAs are encouraged 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 decreasing ATPT could help increase TP 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. In our experiment DBMSes sped up processing the update-only transactions as the number of processors increased. Multi-core processors may enable intra-parallelism within the same transaction, contributing to decreasing response time. However, it was not helpful to have more processors in increasing TP, as mentioned in the first bullet above. DBAs therefore should attempt to make more processors available to DBMSes to reduce transaction response time. Future work should address what other alternatives can improve response time, resulting in increasing TP.

11. Conclusions and Future Work

DBMS thrashing is an impediment to application performance. The less thrashing that occurs, the better. For that reason, it is important to understand this phenomenon, so that it can be predicted and possibly prevented.

We have explored many possible candidate factors affecting the thrashing phenomenon that are present in modern relational DBMSes. We presented a refinement of a structural causal model for explicating the sources of thrashing, which we evaluated with substantial empirical data that manipulated the underlying factors.

The refined model of the read group explained about 14% and 29% of the variances of TP (thrashing point) and average transaction processing time (ATPT), respectively. TP has statistically significant correlations with ATPT, numProcs, and PK, among which numProcs is the most significant to TP. 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 TP.

The refined model of the update group explained about 15% (8%) of the variances of TP (ATPT). TP has statistically significant correlations with numProcs and ATPT, and numProcs is more significant to TP. 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 TP as well.

The refined model suggests several engineering implications that can be helpful to DBAs and provide research questions to database researchers. An important implication of the model is that transaction response time is one of the most significant factors of thrashing. When thrashing is detected, DBAs should examine whether the response time of transactions in their workloads suddenly increased at a certain point. The model also suggests that decreasing the response time can increase TP. 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 increasing the number of processors has a negative correlation with TP, however, it would be helpful to to explore other ways of decreasing the response time of the update-only workloads.

To the best of our knowledge, our thrashing causal model is the first to be articulated across multiple relational DBMSes. Our model leaves room

for elaboration, as there are surely other unknown origins/causes of DBMS thrashing.

We suggest the following areas where future work would be fruitful: (i) consider different types—compute-bound, mixed, nested, multi-level, chained, queued, and distributed—of transactions, perhaps including inserts/deletes and involving multi-table and multi-attributes, (ii) refine the operationalization of ROSE and SF and reexamine their significance on TP and ATPT in the read and update groups, (iii) expand the range or number of parameter values for various operationalizations, (iv) consider in more detail moderated mediations that were not statistically significant, (v) reinvestigate the significance of PK on ATPT and TP in the update group, (vi) scrutinize the cause of the increased significance of numProcs for the read group in relation to the added value in the numProcs operationalization, (vii) apply the alternatives for improving I/O, mentioned in Section 10, to investigate their relative efficacy, (viii) explore other alternatives to reduce transaction response time in the update group, (ix) consider the impace of insertions and deletions and more complicated transactions over multiple tables, and lastly, (x) collect more samples from DBMSes and re-performing the D-W test against the data.

The model we have articulated explains about 14% (15%) of the DBMS thrashing variance for read-only (update-only) transactional workload. Note that in general high variance explained is exceedingly hard to obtain in many disciplines, as many factors can make a discernible increase in the explained variance. Our results imply much room for further refinement and elaboration. Specifically, the database community can propose further refinements to the refined model to hopefully increase its explanatory power, by (i) studying the effects of a yet to be proposed causal factors: different types of indexes or short transaction (selecting a single row only) rates in a batch, (ii) exploring a wider 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, (v) investigating whether thrashing can also be attributed to contention on the pool of threads handling active transactions, (vi) incorporating MPL as a (dependent) variable into the model and operationalizing it by measuring the number of sessions (potentially pooled across a limited number of active connections) as a representative of the fire hose approach [61], and finally, (vii) considering other statistical tools, such as nonlinear regression [62], logistic regression [63], and structural equation modeling [64], for the model.

In all cases, our model is the appropriate starting place for an elaborated causal model with more factors.

12. Acknowledgments

This work was supported in part by the National Science Foundation under grants IIS-0639106, IIS-1016205, IIS-0415101, and EIA-0080123, as well as by the EDISON project managed by the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT & Future Planning (NRF-2011-0020576). We thank John Kececioglu and Peter J. Downey for their insightful comments to improve this article. We also appreciate Tom Lowry and Tom Buchanan for maintaining our experimental instrument, a laboratory of ten machines and associated software.

References

[1] T. Horikawa, An Approach for Scalability-Bottleneck Solution: Identification and Elimination of Scalability Bottlenecks in a DBMS, SIGSOFT Software Engineering Notes (SEN) 36 (2011) 31–42.

[2] S. Chaudhuri, U. Dayal, An Overview of Data Warehousing and OLAP Technology, SIGMOD Record 26 (1997) 65–74.

[3] S. A. Schuster, Relational Data Base Management for On-Line Transaction Processing, Technical Report 81.5, Tandem Computers Incorporated, 1981.

[4] R. Johnson, I. Pandis, A. Ailamaki, Critical Sections: Re-emerging Scalability Concerns for Database Storage Engines, in: DaMoN '08, 2008, pp. 35–40.

[5] P. J. Denning, Thrashing, http://denninginstitute.com/pjd/PUBS/ENC/thrash08.pdf, 2008.

[6] G. Weikum, A. Möenkeberg, C. Hasse, P. Zabback, Self-Tuning Database Technology and Information Services: From Wishful Thinking to Viable Engineering, in: VLDB '02, 2002, pp. 20–31.

[7] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, B. Falsafi, Shore-MT: A Scalable Storage Manager for the Multicore Era, in: EDBT '09, 2009, pp. 24–35.

[8] H. Jung, H. Han, A. D. Fekete, G. Heiser, H. Y. Yeom, A Scalable Lock Manager for Multicores, in: SIGMOD '13, 2013, pp. 73–84.

[9] A. Thomasian, Concurrency Control: Methods, Performance, and Analysis, ACM Computing Surveys 30 (1998) 70–119.

[10] B. Mozafari, C. Curino, A. Jindal, S. Madden, Performance and Resource Modeling in Highly-Concurrent OLTP Workloads, in: SIGMOD '13, 2013, pp. 301–312.

[11] S. Currim, S. Ram, A. Durcikova, F. Currim, Using a Knowledge Learning Framework to Predict Errors in Database Design, Information Systems 40 (2014) 11–31.

[12] A. Dan, D. M. Dias, P. S. Yu, The Effect of Skewed Data Access on Buffer Hits and Data Contention in a Data Sharing Environment, in: VLDB '90, Morgan Kaufmann Publishers Inc., 1990, pp. 419–431.

[13] A. Dan, P. S. Yu, J. Y. Chung, Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability, The VLDB Journal (VLDBJ) 4 (1995) 127–154.

[14] M. J. Carey, S. Krishnamurthi, M. Livny, Load Control for Locking: The 'Half-and-Half' Approach, in: PODS '90, 1990, pp. 72–84.

[15] A. Mönkeberg, G. Weikum, Conflict-Driven Load Control for the Avoidance of Data-Contention Thrashing, in: ICDE '91, 1991, pp. 632–639.

[16] A. Mönkeberg, G. Weikum, Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing, in: VLDB '92, 1992, pp. 432–443.

[17] Y. C. Tay, N. Goodman, R. Suri, Locking Performance in Centralized Databases, ACM TODS 10 (1985) 415–462.

[18] A. Thomasian, Thrashing in Two-Phase Locking Revisited, in: ICDE '92, IEEE, 1992, pp. 518–526.

[19] A. Thomasian, A Two-Phase Locking Performance and Its Thrashing Behavior, ACM TODS 18 (1993) 579–625.

[20] A. Thomasian, A Performance Comparison of Locking Methods with Limited Wait Depth, IEEE TKDE 9 (1997) 421–434.

[21] P. S. Yu, D. M. Dias, S. S. Lavenberg, On the Analytical Modeling of Database Concurrency Control, J. ACM 40 (1993) 831–872.

[22] P. A. Franaszek, J. T. Robinson, A. Thomasian, Concurrency Control for High Contention Environments, ACM TODS 17 (1992) 304–345.

[23] T. Horikawa, Latch-Free Data Structures for DBMS: Design, Implementation, and Evaluation, in: SIGMOD '13, 2013, pp. 409–420.

[24] B. Zhang, M. Hsu, Modeling Performance Impact of Hot Spots, in: **Performance of Concurrency Control Mechanisms in Centralized Database Systems**, Prentice-Hall, Inc., 1995, pp. 148–165.

[25] A. F. Hayes, **Introduction to Mediation, Moderation, and Conditional Process Analysis: A Regression-Based Approach**, Guilford, 2013.

[26] E. J. Pedhazur, **Multiple Regression in Behavioral Research**, Thomson Learning, 1997.

[27] K. Imai, L. Keele, D. Tingley, A General Approach to Causal Mediation Analysis, Psychological Methods 15 (2010) 309–334.

[28] Y.-K. Suh, R. T. Snodgrass, R. Zhang, AZDBLab: A Lab Information System for Large-scale Empirical DBMS Studies, PVLDB 7 (2014) 1641–1644.

[29] A. Thomasian, Chapter 56: Performance Evaluation of Computer Systems, in: **Computing Handbook, Third Edition**, Chapman and Hall/CRC 2014, 2014, pp. 1–50.

[30] S. S. Lavenberg, **Computer Performance Modeling Handbook**, Academic Press, 1983.

[31] D. T. McWherter, B. Schroeder, A. Ailamaki, M. Harchol-Balter, Priority Mechanisms for OLTP and Transactional Web Applications, in: ICDE '04, 2004, pp. 535–546.

[32] J. Letchner, M. Balazinska, C. Ré, M. Philipose, Approximation Trade-offs in a Markovian Stream Warehouse: An Empirical Study, Information Systems 39 (2014) 290–304.

[33] M. Döhring, H. A. Reijers, S. Smirnov, Configuration vs. Adaptation for Business Process Variant Maintenance: An Empirical Study, Information Systems 39 (2014) 108–133.

[34] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, S. Singh, Torturing Databases for Fun and Profit, in: OSDI '14, 2014, pp. 449–464.

[35] J. Gray, The Transaction Concept: Virtues and Limitations (Invited Paper), in: VLDB '81, 1981, pp. 144–154.

[36] P. Atzeni, F. Bugiotti, L. Rossi, Uniform Access to NoSQL Systems, Information Systems 43 (2014) 117–133.

[37] R. Cattell, Scalable SQL and NoSQL Data Stores, SIGMOD Record 39 (2011) 12–27.

[38] FAL Labs, Tokyo Cabinet: A Modern Implementation of DBM, http://fallabs.com/tokyocabinet/, viewed on Feb 7, 2015.

[39] Symas Corporation, Symas Lightning Memory-Mapped Database (LMDB), http://symas.com/mdb/, viewed on Feb 7, 2015.

[40] Hwaci, SQLite, http://www.sqlite.org/, viewed on Feb 7, 2015.

[41] MariaDB Foundation, MariaDB: An Enhanced, Drop-in Replacement for MySQL, https://mariadb.org/, viewed on Feb 7, 2015.

[42] R. T. Snodgrass, P. Denning, The Science of Computer Science: Closing Statement: The Science of Computer Science (Ubiquity Symposium), Ubiquity 2014 (2014) 1–11.

[43] J. Gray, A. Reuter, **Transaction Processing: Concepts and Techniques**, 1st ed., Morgan Kaufmann Publishers Inc., 1992.

[44] Oracle Corporation, The Java Database Connectivity (JDBC), 2014. http://www.oracle.com/technetwork/java/javase/jdbc/index.html (accessed April 15, 2014).

[45] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access Path Selection in a Relational Database Management System, in: SIGMOD '79, 1979, pp. 23–34.

[46] K. J. Preacher, D. D. Rucker, A. F. Hayes, Addressing Moderated Mediation Hypotheses: Theory, Methods, and Prescriptions, Multivariate Behavioral Research 42 (2007) 185–227.

[47] Y.-K. Suh, Exploring Causal Factors of DBMS Thrashing, Ph.D. dissertation, Dept. of Computer Science, Univ. of Arizona, 2015.

[48] I. Pandis, R. Johnson, N. Hardavellas, A. Ailamaki, Data-oriented Transaction Execution, PVLDB 3 (2010) 928–939.

[49] J. Nilsson, F. Dahlgren, Improving Performance of Load-store Sequences for Transaction Processing Workloads on Multiprocessors, in: ICPP '99, IEEE, 1999, pp. 246–255.

[50] 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).

[51] S. Currim, R. T. Snodgrass, Y.-K. Suh, R. Zhang, M. Johnson, C. Yi, DBMS Metrology: Measuring Query Time, in: Proceedings of the 39th ACM SIGMOD International Conference on Management of Data (SIGMOD '13), ACM, 2013, pp. 261–272.

[52] S. Currim, R. T. Snodgrass, Y.-K. Suh, R. Zhang, A Better Way of Measuring Query Time, 2015. Under review.

[53] S. Currim, R. T. Snodgrass, Y.-K. Suh, R. Zhang, A Causal Model of DBMS Suboptimality, 2016. Under review.

[54] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, 2014.

[55] J. Durbin, G. S. Watson, Testing for Serial Correlation in Least Squares Regression, Biometrika 58 (1971) 1–19.

[56] D. Cochrane, G. H. Orcutt, Application of Least Squares Regression to Relationships Containing Auto-Correlated Error Terms, Journal of the American Statistical Association 44 (1949) 32–61.

[57] R. D. Cook, Detection of Influential Observations in Linear Regression, Technometrics 19 (1977) 15–18.

[58] Wikiversity, Multiple Linear Regression/Assumptions, http://en.wikiversity.org/wiki/Multiple_linear_regression/Assumptions, viewed on Oct 31, 2014.

[59] V. Holt, M. Ramage, K. Kear, N. W. Heap, The Usage of Best Practices and Procedures in the Database Community, Information Systems 49 (2015) 163–181.

[60] Y.-K. Suh, B. Moon, A. Efrat, J.-S. Kim, S.-W. Lee, Memory Efficient and Scalable Address Mapping for Flash Storage Devices, Journal of Systems Architecture 60 (2014) 357–371.

[61] T. Barclay, J. Gray, D. Slutz, Microsoft TerraServer: A Spatial Data Warehouse, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00), ACM, New York, NY, USA, 2000, pp. 307–318.

[62] T. Amemiya, Nonlinear Regression Models, Handbook of Econometrics 1 (1983) 333–389.

[63] J. S. Long, **Regression Models for Categorical and Limited Dependent Variables**, 2 ed., SAGE Publications, 1997.

[64] J. Ullman, Structural Equation Modeling: Reviewing the Basics and Moving Forward, Journal of Personality Assessment 87 (2006) 35–50.