Achieving Performance Predictability in Transactional Databases

by

Jiamin Huang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2019

Doctoral Committee:

Associate Professor Barzan Mozafari, Chair
Associate Professor Karthik Duraisamy
Professor H. V. Jagadish
Associate Professor Thomas Wenisch

Jiamin Huang

jiamin@umich.edu

ORCID iD: 0000-0002-1271-9309

Acknowledgments

I am deeply grateful to my advisor, Professor Barzan Mozafari. He has been my guide and my friend throughout my Ph.D. career, and has always been exceedingly generous with his time and advice during challenging or difficult periods. He has been constantly supportive of me and respected my decisions, even at times when I decided to try a different research direction. Without his encouragement and help, I would not have been able to successfully merge the algorithms proposed in this dissertation into MySQL and foster such real-world impact. His contribution to my research has improved not only my understanding of how to do independent and principled research, but also my research paper writing and presentation skills, as he is an excellent writer and presenter and has very high standards for both skills. I am sincerely grateful and appreciative to have had him as my advisor. My Ph.D career has been a difficult journey, but with his advice, guidance and support, it is also an experience I will never forget.

I would also like to express my gratitude to my dissertation committee members. Both Karthik Duraisamy and H.V. Jagadish were kind enough to serve as my dissertation committee members, and they both provided detailed feedback and useful advice on this dissertation, which helped improve its completeness by a lot. I had a delightful experience working with Thomas Wenisch, who has been incredibly insightful and endlessly kind. Besides my committee members, I also thank Professor Grant Schoenebeck for his great help in our collaboration. Even from my short experience with him I could tell that he is a truly wise person.

I have also received great help from my fellow students in the database group at the University of Michigan. Boyu Tian is a good friend of mine, and a collaborator on one of my research projects. He is intelligent and inventive, and I am so glad I was able to work with him. I thank Jarrid Rector-Brooks for his generous help in the creation of VProfiler. He cleaned up my initial version of code and implemented a lot of functionalities in VProfiler.

I am sincerely grateful to the people who helped us with the adoption of our transaction scheduling algorithms. Without their effort my research would not have been able

to make such a big impact. I would like to thank Mark Callaghan for connecting us with people at MySQL, MariaDB and Percona. The independent evaluations by Alexey Stroganov and Dimitri Kravtchuk helped to prove the validity of our algorithms, which finally led to the integration of our algorithms into the codebases of these database systems. Sunny Bains, Laurynas Biveinis, and Jan Lindström communicated with me throughout the entire process, provided feedback on my implementation, and helped to improve the code. They are all brilliant individuals and I am very fortunate to have remotely met and worked with them.

Finally, and most importantly, I thank my dear parents and brother, for all of their support over the years. I am endlessly grateful that my parents have always respected my life decisions, and have been supportive of me, whatever choices I made or paths I forged. They devoted all of their time to my brother and me, hoping for nothing but for us to live better lives. Although they were unable to receive much education themselves, they are both intelligent and experienced, and worked tirelessly to ensure that their children are well-educated. My family shares in both my struggles and my joys. Throughout my difficulties, they've always encouraged me to carry on. I am glad that I can now make them all proud.

Table of Contents

List of Figures

List of Tables

Abstract

While much of the research on transaction processing has focused on improving overall performance in terms of throughput and mean latency, surprisingly less attention has been given to performance predictability: how often individual transactions exhibit execution latency far from the mean. Performance predictability is increasingly important when transactions lie on the critical path of latency-sensitive applications, enterprise software, or interactive web services.

This dissertation proposes a systematic approach to solving performance predictability issues in transactional database systems. We propose the first profiler (to the best of our knowledge) to diagnose these issues and identify sources of unpredictability. Based on our findings from applying this tool to MySQL, we introduce techniques to mitigate the sources of unpredictability. Specifically, this dissertation makes three main contributions:

- **Variance Profiler (VProfiler):** In contrast to most software profiling tools that quantify *average* performance, we propose a profiler called VProfiler that, given the source code of a software system and programmer annotations indicating the start and end of semantic intervals of interest, is able to identify the dominant sources of latency variance in a semantic context.

- **Studying Causes of Unpredictability in Existing Database Systems:** We conduct the first *quantitative study* of major sources of variance in MySQL, Postgres (two of the largest and most popular open-source products on the market), and VoltDB (a non-conventional database). Based on our findings, we investigate alternative algorithms, implementations, and tuning strategies to reduce latency variance without compromising mean latency or throughput. For instance, we realize that locking in various components is the major source of variance in traditional database systems, and thus propose techniques, including our VATS scheduling algorithm and parallel logging, to mitigate this.

- **Contention-Aware Transaction Scheduling:** Nearly all existing database systems use a First-In-First-Out strategy for deciding which transaction should be granted

the lock on a row when it becomes free. Inspired by our VATS algorithm, we further study the effect of lock scheduling in database systems, and propose contention-aware scheduling algorithms (LDSF and bLDSF) that reduce average latency, and thereby improve the overall predictability of database systems.

VProfiler enables us to identify the root causes of performance variance in existing database systems. By introducing algorithms targeting those root causes, we not only improve performance predictability by up to 5.6x, but also improve average performance by up to 6x. **Most notably, our VATS algorithm has been merged into MariaDB and our LDSF algorithm has been made the default scheduling algorithm in Oracle MySQL and Percona, starting from version 8.0.3.**

Chapter 1

Introduction

This dissertation studies the problem of unpredictable performance in modern transactional database systems. In this chapter, we start by introducing the concept of performance predictability in the context of modern database systems and discussing its importance in mission-critical and user-interfacing applications. We then describe our overall methodology for studying and improving performance predictability, and discuss the research challenges involved. We conclude the chapter with an outline of our contributions in the rest of this dissertation.

1.1 Performance Predictability in Transactional Databases

Transactional databases are a mission-critical component of enterprise software for efficient storage and manipulation of data. A significant portion of database research on transaction processing has focused on improving overall performance and scalability, for example, by developing new techniques for concurrency control, query optimization, indexing, and other sophisticated ideas. These strategies, however, have been vetted primarily in terms of their effect on the *average performance* of the database, such as its throughput and mean transaction latency. In other words, the focus has been on average performance and running more and faster transactions *overall*.

While peak transaction processing throughput is clearly important, the *predictability* of performance—the disparity between average and high-percentile tail latencies—has emerged as an equally important metric in situations where individual transaction latencies are mission-critical or affect end-user experience. Examples include database-backed web services or database clouds with service-level agreements.[1] However, performance

[1] In this dissertation, we do not target real-time applications, which require hard (rather than statistical) guarantees, e.g., airplane control systems.

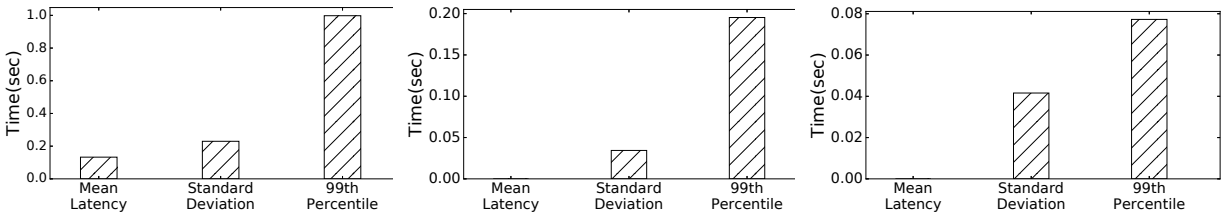

Figure 1.1: Mean, standard deviation, and 99th percentile latencies in MySQL (left), Postgres (center), and VoltDB (right).

predictability has often been overlooked by traditional efforts that focus on throughput and mean latency. In fact, some optimization strategies (e.g., asynchronous logging and group commit [105, 181]), have deliberately improved throughput at the expense of penalizing latency for *some* transactions. In other cases, developers have not even vetted the impact of their design and implementation decisions on performance predictability. For example, while the contribution of database components to mean latency has been studied [102], an analogous study to identify the sources of latency variance is missing. As such, today's complex DBMSs might have overlooked alternative design decisions that could deliver the same or comparable average performance, but with significantly lower variance.

At the fine time-scale of individual transactions, the performance of existing databases is incredibly unpredictable, with orders of magnitude gaps between mean and high percentile latencies[2] (see figure 1.1).

1.2 Challenges and Our Methodology

Generally speaking, two approaches can be taken to achieve performance predictability in a transactional database system.

Bottom-up vs. Top-down Approach— In a bottom-up approach, one could aim to build an entirely new DBMS from scratch that is specifically designed to be predictable. For example, some prior work proposes to move consistency guarantees from database systems to the application layer itself [86]. Others have advocated for the use of table scans for all queries [42, 103, 158, 163, 190], or to consider only query plans with bounded worst-case [39, 40]. The latter is similar to how real-time databases try to meet deadlines [28, 119, 152]. Despite their merits, these proposals have not had widespread adop-

[2]While some of this variance is inherent and due to some transactions doing more work than others, our study reveals that dominant sources of variance are often a performance pathology and avoidable (see Section 3.1 for the distinction).

tion for transaction processing. One reason is that users are often reluctant to completely abandon well-established and matured DBMSs in exchange for academic prototypes. Another reason, however, is that these proposals promise higher throughput or predictability by sacrificing mean latency (e.g., by always using scan-only plans [190]). Despite their success in long-running analytics [158], such tradeoffs are less appealing to transactional and latency-critical applications. Instead, an ideal solution is one that delivers the same mean latency and throughput as existing solutions, but with much lower variance. Adopting such solutions, especially if compatible with existing DBMSs, would be a "no-brainer" for most users (see Section 3.7).

Thus, in this dissertation, we advocate a top-down approach, wherein we identify and mitigate the performance pathologies that lead to variance in existing, widely used transaction processing systems—an approach that can have more immediate impact on real-world deployments. As such, we address some of today's popular DBMSs in an attempt to understand their major sources of variance, and seek design alternatives for overcoming variance-inducing performance pathologies. For the same reason, we also restrict ourselves to solutions that reduce variance without sacrificing mean latency or throughput. In addition to the immediate benefits to massive user-bases of these products, the insight gained in this process can inform future bottom-up attempts at designing new databases.

Challenges— A top-down approach, however, comes with its own challenges. Gaining performance insight into any software system as complex as a DBMS requires effective profiling tools. Unfortunately, existing profilers can only study a system in terms of its *average performance*, e.g., by breaking down overall run time into the average latency of individual functions, or counting the number of times a function is invoked. Such information offers little help to our understanding of the root causes of overall latency variance. What we need is a systematic way of quantifying the contribution of individual functions to overall variance, which is something that has not been realized in any existing profiling tool. Moreover, latency variance of each function is only important to the performance profile insomuch as it affects transaction latencies. For example, it may not matter if a background I/O operation exhibits large variance in execution time, as long as the user-perceived latency of a transaction is unaffected. This problem becomes more complicated when we move our focus from traditional database systems to modern ones, such as VoltDB. Unlike traditional database systems where the execution of a transaction happens in one thread, these modern systems employ an event-driven model and process transactions asynchronously, which means that the execution of a transaction can span multiple threads, and there can even be overlaps. It becomes very

difficult to tell which portion of the execution time should be considered as part of the transactions' latencies in this case.

Why Now?— It is both critical and timely to systematically study and manage performance variance of transactional databases for several reasons. First, advancements in hardware parallelism and better transaction processing techniques have enabled microsecond latencies and millions of concurrent transactions [44, 126, 204]. As mean performance improves, the impact of performance perturbations (e.g., due to a slow I/O request) relative to the latency of a transaction grows. Second, an increasing number of DBaaS providers guarantee service level agreements (SLAs), which, if violated (even for a subset of transactions), result in financial penalties [5, 9, 150, 151]. Finally, modern DBMSs have become some of the most complex software systems. As such, subtle interactions of complex code paths lead to vexing performance anomalies.

1.3 Contributions and Outline

In this dissertation, we address the problem of performance unpredictability in transactional database systems and aim to find techniques to improve it without sacrificing raw performance. Our first step is to understand the root causes of unpredictable performance. Unfortunately, existing software profilers are mostly designed for analyzing "average" performance, and hence offer little or no insight into this problem. We therefore propose VProfiler, the first profiler—to the best of our knowledge—designed specifically for diagnosing the root causes of performance unpredictability in a large codebase (such as that of a database system). We describe the detailed design and implementation of VProfiler in Chapter 2.

In Chapter 3, we use VProfiler to conduct the first *quantitative study* of major sources of variance in MySQL and Postgres (two of the largest and most popular open-source products on the market), as well as VoltDB (as an example of a non-traditional database). Based on our findings, we investigate alternative algorithms, implementations, and tuning strategies that can reduce latency variance without compromising throughput or mean latency. In particular, we propose a new lock scheduling algorithm, called Variance-Aware Transaction Scheduling (VATS), and a lazy buffer pool replacement policy.

Our findings indicate that scheduling can be an effective means of reducing contention, and thereby performance variability. Thus, in Chapter 4, we further study a new problem that has been overlooked by researchers: When there are multiple lock requests

on the same object, which one(s) should be granted first? Nearly all existing systems rely on a FIFO (first in, first out) strategy to decide which transaction(s) should be granted the lock. In this dissertation, however, we show that lock scheduling choices have significant ramifications on the overall performance of a transactional system. While there is a large body of research on job scheduling outside the database context, lock scheduling presents subtle but challenging requirements that render existing methods for scheduling inapt for a transactional database. By carefully studying this problem in Chapter 4, we propose the concept of contention-aware scheduling, formally study the hardness of the problem, and propose novel lock scheduling algorithms (LDSF and bLDSF) which guarantee a constant factor approximation of the best scheduling.

Finally, we conclude in Chapter 5 by summarizing the major contributions of this dissertation, and briefly discuss future research directions.

Chapter 2

Profiling for Performance Variance

2.1 Introduction

Profiling tools are a key means of gaining performance insight into complex software systems. Existing profilers provide performance statistics about an application's function call hierarchy. For example, they can answer questions such as which functions are called most often from a particular context, or how much time is spent inside each function. The answers, however, are almost always expressed in terms of *average* performance [49, 93, 98, 178, 182]. Even when multiple runs are used to infer a latency histogram [187], users can still only find out which functions contribute the most to the *overall* latency in each execution time range. However, an increasing number of modern applications come with performance requirements that are hard to analyze with existing profilers. In particular, delivering predictable performance is becoming an increasingly important criterion for *real-time or interactive applications* [4, 14–18, 23, 190], where the latency of individual requests is either mission-critical or affecting user experience.[1]

Existing profilers can only study such systems in terms of their *average performance* breakdown, for example, by attributing average latency to contributions of individual functions. Such profilers offer little help in quantifying the contribution of individual functions to the overall latency variance—a problem that is much more challenging.

The second problem is that performance predictability might only matter in terms of a *semantic interval* that encapsulates the end-user's experience or interaction with the system. For example, a background I/O operation exhibiting large variance in execution time may not matter as long as the graphical user interface does not freeze and users can continue interacting with the system. Similarly, in a database system, perfor-

[1] In this dissertation, we do not target those real-time applications that require hard (rather than statistical) guarantees, e.g., airplane control systems.

mance predictability may only matter insofar as it concerns transaction latencies, and thus latency variance of other functionalities (e.g., periodic log flushing to a separate disk) is irrelevant to the performance profile as long as they do not affect transaction latencies. Unfortunately, semantic intervals (e.g., the end-to-end latency of a database transaction or a web-server request) may not always correspond to a single thread or a single top-level function. A transaction might start on one thread and be handed off to and completed on a different thread. In event-based software architectures, a user session or transaction might create multiple events on a shared work queue, whereby multiple worker threads process events in a round robin fashion. The notion of a user transaction or session is inherently a *semantic* one, and it cannot be automatically detected by a generic profiler. For example, the processing of the last event associated with a session may not necessarily correspond to the user's perception of a session end, e.g., post-commit cleanups do not affect a user's latency perception.

Although faster storage and increased hardware parallelism have helped to improve mean performance in general, mitigating performance variance has largely remained an open problem. With increasing complexity of modern applications, subtle interactions of difficult-to-analyze code paths often lead to vexing performance anomalies [4, 14–18, 23, 190]. The rising popularity of cloud-services and service-level agreements in mission-critical applications has increased the need for performance predictability. In light of these trends, we believe it is critical and timely to undertake a systematic approach to diagnosing performance variance in a semantic context.

In this work, we propose a profiling framework, called VProfiler, that can solve both problems. Given the source code of an application and a minimal effort in demarcation of semantic intervals and synchronization primitives, VProfiler identifies the dominant sources of latency variance of semantic intervals. VProfiler iteratively instruments the application source code, each time collecting fine-grain performance measurements for a different subset of functions invoked during the semantic interval of interest. By analyzing these measurements across thread interleavings, VProfiler aggregates latency variance along a backwards path of dependence relationships among threads from the end of an interval to its start. Then, using a novel abstraction, called a *variance tree*, VProfiler carefully reasons about the relationship between overall latency variance and the variances and covariances of the execution time of culprit functions, providing insight into the root causes of performance variance.

We evaluate VProfiler's efficiency by analyzing three popular open-source projects, MySQL, Postgres, and Apache Web Server, identifying major sources of latency variance of transactions in the former two and of web requests in the latter. We present the results

we found in these case studies, and leave the detailed analysis to the next chapter. In addition to their popularity, we have chosen these three systems for several reasons. First, due to their massive, legacy, and poorly-documented codebases, manual inspection of these source codes is a challenging task (e.g., MySQL has 1.5M lines of code and 30K functions). Second, transactional databases and web servers are a key component of many interactive applications.

Contributions— We make the following contributions:

1. We introduce a novel abstraction, called a *variance tree*, to reason about the relationship between overall latency variance and the variances and covariances of the execution time of culprit functions. Using this abstraction, we present VProfiler as the first profiling tool that can efficiently and rigorously decompose the variance of the execution time of a semantic interval by analyzing application source code and identifying the major contributors to its variance (Section 2.3).

2. We use VProfiler to analyze MySQL, Postgres, and Apache Web Server, and successfully identify a handful of functions in these massive codebases that contribute the most to their latency variance (Section 2.4).

We discuss the scope of our work in Section 3.1, and introduce VProfiler in Section 2.3. We evaluate VProfiler's efficiency in Section 2.4.1. We discuss related work and conclude in Sections 4.8 and 3.8.

2.2 Scope

In this section, we briefly discuss the scope of our work.

Defining Predictability— There are many mathematical notions for capturing *performance predictability* in a software system. One could aim at minimizing the latency variance or tail latencies (e.g., 99th percentile). Alternatively, one could focus on bounding these quantities, e.g., ensuring that the 99th percentile remains under a fixed threshold. To obtain a standardized measure of dispersion (or spread) for a distribution, statisticians sometimes calculate the ratio of standard deviation to mean (a.k.a. coefficient of variation).

While there are many choices, in this dissertation we focus on identifying the sources of latency variance (and thereby standard deviation), but we only consider solutions that reduce the variance but do not increase mean latency (or reduce throughput). For example, simply padding all latencies with a large wait time will trivially reduce variance

but will increase mean latency. While certain applications might tolerate an increase in mean latency in exchange for lower variance [42,86,103,158,163,190], such tradeoffs may not be acceptable to most latency-sensitive applications. Thus, in this dissertation, we restrict ourselves to ideal solutions, i.e., those that reduce variance without negatively impacting mean latency or throughput. In fact, as shown in Section 2.4, not only do our findings reduce variance, but they also *reduce* mean latency and coefficient of variation. As reported in Section 3.7, one of the variance solutions we discovered with the aid of VProfiler has been quickly adopted (and made a default policy) by MySQL's major distributions.

Finally, while we do not directly minimize tail latencies, reducing variance serves as a surrogate for reducing high-percentiles too [195]. For example, our techniques reduce overall variance by 82%, and 99th percentile latency by 50% for the TPC-C benchmark.

Diagnosis, Not Automated Fixing— There are two steps involved in performance debugging. One is identifying the root cause of a performance problem (variance, in our case), and the other is resolving the issue. Like all profilers, VProfiler focuses on the former. While automating the second step is challenging (e.g., it requires knowing the programmer's original intention), the first step is equally important. In fact, to the best of our knowledge, no existing profiler can identify the true sources of performance variance in a systematic fashion, and VProfiler is the first in this regard (see Section 4.8).

Though resolving the issue ultimately requires manual inspection, the manual effort needed is often proportional to the extent to which the profiler localizes the sources of the problem. As reported in Section 2.4, VProfiler's findings allow us to examine only a handful of functions (out of tens of thousands) and dramatically reduce latency variance with modest programming efforts across MySQL, Postgres, and Apache Web Server.

Inherent vs. Avoidable Variance— It is important to note that performance variance is sometimes inherent and cannot be avoided. For example, processing a query that performs more work will inherently take longer than one that performs less work.[2] Avoidable sources of variance are those that are not caused by varying amounts of work, but rather due to internal artifacts in the source code, such as scheduling choices, contention, I/O, or other performance pathologies. For example, two transactions requesting similar amounts of work but experiencing different latencies indicate a performance anomaly that might be avoidable. Determining whether an identified source of latency variance is avoidable or not requires the programmer's understanding of what constitutes *inherent*

[2]In prior work, we have studied the variance of performance caused by external factors (such as changes in the workload environment) and strategies for mitigating them [144,145].

work in a given application. VProfiler simply reports dominant sources of performance variance so that programmers can focus their attention on only a handful of culprit functions.

Software vs. I/O Delays— In distributed and cloud-based applications, variance in network delays can cause variance in user-perceived latencies. In VProfiler, variance in I/Os such as network traffic or disk (synchronous or asynchronous) operations manifests as variance in the functions that receive the result of the I/O operations, providing programmers an indication that I/O variance is the root cause.

2.3 VProfiler

With the complexity of modern software, there are many possible causes of latency variance, such as I/O operations, locks, thread scheduling, queuing delays, and varying work per request. Although there are a variety of tracing tools that provide some visibility into application internals (e.g., strace to gain visibility into I/O operations, and DTrace [94] to profile performance), these tools do not directly report performance variation or identify outlying behavior. Moreover, most tracing tools aggregate and report results according to the application call hierarchy, which often does not correspond well to user-visible performance metrics, such as request or transaction processing time. Finally, general-purpose tracing tools introduce substantial (sometimes order-of-magnitude) slowdowns, when collecting fine-grain measurements. For example, we report the overhead of DTrace in Section 2.3.3.4. The overhead of these tools skews application behavior and obscures root causes of latency variance. In this section, we introduce VProfiler, a novel tool for automatically instrumenting a subset of functions in an application's source code to profile execution time variance at fine time scales with minimal overhead (to preserve the behavior of the system under study).

2.3.1 Semantic Profiling

A key objective of VProfiler is to quantify performance means and variances over *semantic intervals* rather than report results that are tightly coupled to the application's call graph. A semantic interval is a programmer-defined execution interval that corresponds to a repeated application behavior, which the programmer wishes to profile. Our intent is that a semantic interval should correspond to a single request, session, connection, transaction, and so on, thus allowing the programmer to analyze per-request latency and variance. Note that a semantic interval may encompass concurrent execution spanning

multiple threads, or include the time a particular request/context was waiting in a queue or was blocked awaiting some resource.

VProfiler comprises an online trace collection phase and an offline analysis phase. The trace collection phase gathers start and end timestamps of semantic intervals, runtime profiles of a specific set of instrumented functions, and dependency relationships among threads and tasks needed to reconstruct a latency breakdown for each semantic interval. Then, VProfiler performs an offline analysis of these traces to characterize each semantic interval and output a variance profile. If the developer determines that this profile provides insufficient detail, VProfiler selects a new set of functions to instrument to refine the variance profile, and the trace collection phase is repeated. We detail this iterative refinement procedure in Section 2.3.3.4.

VProfiler conceptually divides execution on all threads into *segments*. Each segment is conceptually labeled as either executing on behalf of a single semantic interval, or it is unlabeled to indicate background activity unassociated with any particular request or execution that services multiple requests. As the notion of a semantic interval is application-specific, it must be provided to VProfiler by the programmer via manual annotation. The programmer, via a simple API, enters three kinds of annotations, indicating: (1) when a new semantic interval is created (e.g., transaction start), (2) when a semantic interval is complete (e.g., transaction commit), and (3) when a thread begins executing on behalf of a specific semantic interval (e.g., worker thread dequeues and executes an event associated with the semantic interval).

The first two of these annotations are straight-forward: they provide bounds for the semantic interval. The average performance and overall variance reported by VProfiler is the mean and variance of the time difference between these start and end annotations. However, VProfiler does not seek to merely report these overall aggregate metrics. Rather, it sub-divides the latency between these annotations and attributes it to the execution of particular functions or wait times on specific resources/queues. Furthermore, VProfiler does not require that the start and end of an interval lie on the same thread. Rather, VProfiler considers relationships across threads where one thread unblocks execution of another. It follows such dependence edges backwards from the end of the semantic interval to discover the critical path from the end annotation back to the start timestamp. VProfiler relies on instrumentation added to an application's synchronization primitives that potentially block execution (e.g., locks, condition variables, and message/task queues) to log these dependence edges. We expand on this idea more in Section 2.3.3.2.

The third annotation is designed specifically for task-based concurrent programming models (e.g., Intel's Threaded Building Blocks), where a semantic interval is decoupled from any particular worker thread. Rather, execution on behalf of a transaction or request proceeds as a sequence of (possibly concurrent) tasks that are dequeued from work queues. In such a framework, a program annotation indicates when a worker thread begins processing a task on behalf of a specific semantic interval. The thread is assumed to continue working on behalf of the semantic interval until another explicit annotation indicates execution on behalf of a new interval. In addition, VProfiler instruments functions that enqueue a task, recording a "created-by" relationship, thereby building a directed graph among the tasks. VProfiler uses this graph to provide a breakdown of latency and variance of execution within the semantic interval and distinguish periods where no task is executed and the semantic interval is delayed due to queuing.

Before discussing VProfiler's algorithm for profiling semantic intervals, we first discuss its model for analyzing performance variance of a function invocation, which is the fundamental building block of VProfiler's approach.

2.3.2 Characterizing Execution Variance

A *segment* is a contiguous time interval on a single execution thread that may be labeled as part of, at most, a single semantic interval. In this section, we discuss the concepts VProfiler employs to quantify variance with respect to a single executing segment. We describe how VProfiler analyzes entire semantic intervals in subsequent sections.

VProfiler analyzes performance variance by comparing the duration of particular function invocations in an executing segment across other invocations of the same function in different semantic intervals (i.e., it analyzes variances of invocations of the same function across different requests). VProfiler uses a novel abstraction, the *variance tree*, to reason about the relationship between latency variance and the call hierarchy rooted at a particular function invocation.

2.3.2.1 Variance Tree

We can gain insight into why latency variance arises in an application by subdividing and attributing execution time within a segment across the call graph, similar to a conventional execution time profile generated by tools such as `gprof` [93]. However, rather than identifying functions that represent a large fraction of execution time, we instead calculate the variance and covariance of each component of the call graph across many invocations to identify those functions that contribute the most to performance variabil-

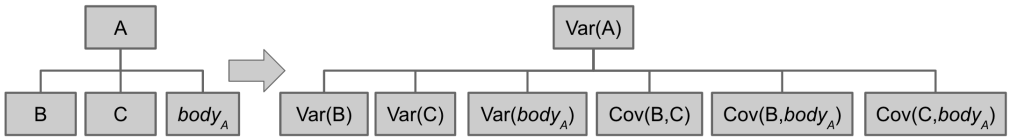

Figure 2.1: A call graph and its corresponding *variance tree* (here, $body_A$ represents the time spent in the body of A).

ity. Two key challenges arise in this approach: (1) managing the hierarchical nature of the call graph and the corresponding hierarchy that arises in the variance of execution times, and (2) ensuring that profiling overhead remains low. We first discuss the former challenge and address the latter in Section 2.3.3.4.

Each variance tree is rooted in a specific function invoked over the course of an application. We measure latency and its variance across invocations. For example, in an event processing system, a dispatcher function that dequeues events from a task queue and invokes the specific code associated with the task might comprise the root of the variance tree. VProfiler will build a variance tree beginning at the topmost function whose execution is included within the segment.

Figure 2.1 (left) depicts a sample call graph comprising a function A invoking two children B and C, and it includes the execution time in the body of A. We can label each node in a particular invocation of this call graph with its execution time, yielding the relationship that the execution time of the parent node is the sum of its children, for example:

$$E(A) = E(B) + E(C) + E(body_A) \tag{2.1}$$

where E represents the execution time of a function. Figure 2.1 (right) shows a corresponding visualization of the variances and covariances in a variance tree representation.

$$Var(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} Var(X_i) + 2 \sum_{1 \le i \le j \le n} \sum Cov(X_i, X_j) \tag{2.2}$$

The variance tree allows VProfiler to quickly identify sub-trees that do not contribute to latency variability, as their variance is (relative to other nodes) small. Identifying the root causes of large variance, however, is not so trivial. The variance of a parent node is always larger than any of its children, so simply identifying the nodes with the highest variance is not useful for understanding the cause of that variance. Furthermore, some variance arises because invocations may perform more work and manipulate more data (e.g., a transaction accessing more records). Such variance is not an indication of a mitigable pathology as the variance is inherent; our objective is to identify sources of

variance that reveal performance anomalies that lead to actionable optimization opportunities. Similarly, high covariance across pairs of functions can be an indicator of a correlation between the amount of work performed by such functions.

At a high level, our goal is to use the variance tree to identify functions (or co-varying function pairs) that (1) account for a substantial fraction of overall latency variance and (2) are informative; that is, functions where analyzing the code will reveal insight as to why variance occurs. To unify terminology, we refer to the variance of a function or co-variance of a function pair as a *factor*.

Identifying factors that account for a large fraction of their parents' variance is straightforward. What is more complicated is identifying functions that are informative. We address this question in the next section.

2.3.2.2 Ranking Factors

Algorithm 1: Factor Selection

Inputs : t: variance break-down tree,
k: maximum number of functions to select,
d: threshold for minimum contribution

Output: s^*: top k most responsible factors

$h \leftarrow$ empty list;
foreach *node* $\phi \in t$ **do**
  $\phi^* \leftarrow$ `factor_of(`ϕ`)`;
  if $\phi^* \notin h$ **then**
    $\phi^*.contri \leftarrow t.contri$;
    $h \leftarrow h \cup \phi^*$;
  else
    $\phi'.contri \leftarrow \phi'.contri + \phi.contri$;
end
foreach $\phi \in h$ **do**
  $\phi.score =$ `specificity(`ϕ`)` $\cdot\ \phi.contri$;
end
Sort h in descending order of $\phi.score$;
$s^* \leftarrow$ empty list;
for $i \leftarrow 1$ **to** k **do**
  $\phi \leftarrow h[i]$;
  if $\phi.contri \geq d$ **then**
    $s^* \leftarrow s^* \cup \phi$;
end
return s^*;

Our intuition is that functions deeper in the call graph implement narrower and more specific functionality, hence, they are more likely to reveal the root cause of latency variance. For example, consider a hypothetical function `WriteLog` that writes several log records to a global log buffer, but must first acquire the lock on the log buffer (`Lock`), copy the log data to the log buffer (`CopyData`), and finally release the lock (`Unlock`). Suppose `WriteLog`'s variance accounts for 30% of its transaction's latency variance, but `CopyData`'s accounts for 28%. Analyzing `CopyData` is likely more informative even though it accounts for slightly less variance than `WriteLog`, because its functionality is more specific. Further investigation may reveal that the variance arises due to the size of log data being copied, suggesting mitigation techniques that reduce log size variance.

Based on this intuition, VProfiler ranks factors using a score function that considers both the magnitude of variance attributed to the factor and its relative position in the call graph. A particular factor may appear in a call graph more than once if a function is invoked from multiple call sites. When ranking factors, VProfiler aggregates the variance/covariance across all call sites.

To quantify a factor's position within the call graph, VProfiler assigns each function a height based on the maximum depth of the call tree beneath it. For factors representing the covariance of two functions, VProfiler uses the maximum height of the two functions. It uses a specificity metric that is a decreasing function of the factor's height ϕ:

$$specificity(\phi) = (height(call\_graph) - height(\phi))^2 \tag{2.3}$$

where $height(call\_graph)$ is the height of the root of the call graph, and $height(\phi)$ is the factor's height. Here we use square to give specificity a higher weight.

VProfiler uses a score function that jointly considers specificity and variance:

$$score(\phi) = specificity(\phi) \sum_i V(\phi_i) \tag{2.4}$$

where $V(\phi_i)$ represents variance or covariance of a specific instance (call site) of a factor within the call graph.

Given the *variance tree*, we now describe an algorithm to select the top-k factors based on their score. The pseudocode is shown in Algorithm 1. For each node in the tree, we determine if the corresponding factor is already in list h. If not, we insert the factor and its (co-)variance into h. Otherwise, we accumulate the (co-)variance represented by the node into the existing element in h (lines 1 to 10). Once we have calculated total (co-)variance of each factor, we calculate their *score* values using Equation 2.4 (lines 11

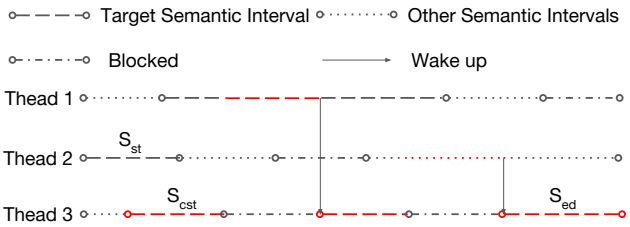

Figure 2.2: A critical path (marked red) constructed for a semantic interval not involving the start segment (S_{st}).

to 13). Then, we sort factors in descending *score* order, selecting the top k whose total (co-)variance is greater than a threshold d (lines 14 to 23).

2.3.3 Profiling a Semantic Interval

We next describe how VProfiler aggregates latency variance of an entire semantic interval from individual execution segments. Furthermore, we describe the iterative refinement method used to recursively add details to the variance tree over a sequence of experimental trials to provide the developer a sufficiently informative latency and variance breakdown of semantic segments.

VProfiler's offline analysis phase characterizes variance in each semantic interval, starting at its final segment (containing the interval completion annotation). VProfiler then aggregates latency and variance of preceding functions on the same thread until it encounters an incoming wake-up edge indicating execution on this segment was triggered by an event elsewhere (e.g., a lock being freed). It then follows this edge, continuing aggregation along the target thread, and so on, following all incoming dependence edges. This backwards traversal terminates when it reaches the creation timestamp of the semantic interval. Note that, in complex executions, the backwards trace may not actually include the segment that created the event, as this segment may not lie on the critical path that determined the end time of the interval (i.e., the end timestamp of the interval may not be improved if its start timestamp were earlier). Figure 2.2 shows an example of this. Intuitively, we conceive of VProfiler as assigning "blame" for accumulated delay leading to the completion of a semantic interval; blame propagates backwards along segments and their dependencies.

2.3.3.1 VProfiler Workflow

Given a complete variance tree, factor selection (Algorithm 1) identifies the top factors that a developer should investigate further to identify the root causes of semantic in-

terval variance. However, collecting a complete variance tree is infeasible due to the enormous size and complexity of call graphs in modern software systems. Instrumenting every function adds significant overhead to execution time, and the variance tree will no longer be representative of unprofiled execution. Hence, VProfiler iteratively refines the instrumentation to build variance trees, starting from their roots until the profile is sufficient for a developer to identify key sources of variance.

In addition to constructing one or more variance trees (rooted at specific functions in each run), to aggregate results over the course of a semantic interval, VProfiler must trace the following data in each run:

1. $\langle tid, sid, ts, te, state \rangle$. Each such 5-tuple describes a segment. tid is the id of the thread on which a segment is executed. sid is a unique identifier of the semantic interval assigned when the interval is created (e.g., a transaction id). ts and te are the starting and ending segment timestamps and $state$ indicates whether the segment was executing, waiting on a task or message queue, or blocked on a lock or I/O.

2. $\langle tid, sid, f, fs, fe \rangle$. Each such 5-tuple describes a function invocation that was selected for instrumentation during this run. f is a function name. fs and fe are the start and end timestamps of an invocation of f.

3. $\langle tid, tid', t \rangle$. Each such 3-tuple indicates that thread tid was woken up by thread tid' at time t.

4. $\langle tid, ts, tid', ts' \rangle$. Each such 4-tuple represents the event creation relationship. Thread tid created an event at time ts, and that event was picked up by thread tid' at time ts'.

2.3.3.2 Tracking Segment Dependencies

VProfiler must track segment dependencies at run-time to construct the 3- and 4-tuples indicating when a thread wakes or creates another thread. For this tracking, VProfiler requires instrumentation of synchronization operations and operations that enqueue tasks in task-based runtimes.

We abstract generic blocking synchronization primitives as having an `acquire(object)` and `release(object)` function.[3] This pattern covers a number of primitives, including

[3]Note that the developer must supply a comprehensive list of acquire and release function names to VProfiler to instrument.

locks, mutexes, condition variables, and semaphores. VProfiler instruments the acquires and releases and tracks lock ownership at run-time using a large hash map of `[oid` $\rightarrow$ `tid]`, where `oid` is an identifier of the synchronization object (e.g., the lock address) and `tid` is the ID of the last thread that holds the object.

To track task relationships in task-based applications, VProfiler instruments the enqueue and dequeue operation on the task queues. We assume a model where the consumer threads pop objects off the queue, and block when the queue is empty. The producer threads push objects, potentially waking a worker thread to accept the task. We assume an abstract API comprising `enqueue(queue, task)` and `dequeue(queue)` functions. Here, VProfiler also maintains a hash table `[task` $\rightarrow$ `tid]` at run-time, where `task` is a unique task identifier and `tid` is the ID of its producer thread.

VProfiler does not instrument the OS scheduler. Hence, if a runnable thread is pre-empted due to CPU over-subscription, VProfiler will include the time the thread is runnable but waiting as part of the execution time of the segment. This limitation of VProfiler can be overcome by instrumenting the OS to log thread switches due to time slice expiration or pre-emption. In the workloads we study, pre-emption is rare as the number of concurrent application threads is tuned not to exceed the number of available cores, so there is no need to instrument the scheduler.

2.3.3.3 Aggregating Segments

Algorithm 2 shows VProfiler's pseudocode for post-processing the variance trees and segment relationship output for an individual semantic interval. This is a recursive function, and the initial call should pass in the id of semantic interval of interest, its end segment, *nil* as its start timestamp, and the end timestamp of the end segment as arguments. In Algorithm 2, a segment S is described by its 5-tuple: $\langle T, C, ts, te, s \rangle$, where T is the thread on which S was executed, C is a unique identifier of interval S, ts and te are the start and end timestamps and s is the state of the segment.

The high-level idea of Algorithm 2 is to construct the critical path for the given semantic interval and to analyze the execution time of the target function in each segment on the critical path. Figure 2.2 shows an example of how a critical path is constructed. The algorithm starts from the ending segment and follows any wake-up/created-by relationship backwards. Note that when the algorithm follows a wake-up relationship backwards, it only processes the waker segment up to the point where the blocked segment starts, and then returns to the thread of the blocked segment.

Algorithm 2 maintains a table of execution times comprising $\langle cid, f, et \rangle$ tuples where cid is the semantic interval id, f is the name of a profiled function (or `other` for time spent

in uninstrumented functions), and et is the total execution time of the function over the course of the semantic interval (as functions may be invoked more than once). Thus, there is a single row for each (semantic interval, function) pair. We process one semantic interval at a time, aggregating time spent in each function while walking backwards along the segments included in the interval, starting at the final segment. The key step in the algorithm is `monitor_exec_time`. For a given semantic interval C, a thread T, and a start and end timestamp ts and te, this step finds all $\langle tid, cid, f, fs, fe \rangle$ tuples that match the semantic interval id and thread id and also overlap the segment's duration. The execution of each function overlapping the segment is then clipped against the bounds of the segment and aggregated into the execution time table. The table is then output when all semantic intervals have been processed.

2.3.3.4 Iterative Refinement

In each iteration, VProfiler identifies the top-k factors when profiling a subset of functions, starting at the root of the call graph. This profile is then returned to the developer, who determines if the profile is sufficient. If not, the children of the top-k factors are added to the list of functions to be profiled, instrumentation code is automatically inserted by VProfiler, and a new profile is collected. In detail:

Initialization (Algorithm 3, line 1 to 3). VProfiler starts with an empty variance tree, and initializes the list of functions to profile with the root.

Variance Break Down (Algorithm 3, line 5 to 8). For each profiled function, VProfiler automatically instruments the code to measure the latency of all invocations of the function and the latency of each child. The variance and co-variances of these children are added to the variance tree, thereby expanding the tree by one level.

Factor Selection (Algorithm 3, line 9 to 17). After the variance tree is expanded, VProfiler performs factor selection to choose the top-k highest scoring factors within the tree, which are then reported. If the profile is insufficient, the developer requests another iteration, which adds the children of these top-k functions to the list to be profiled.

Manual Inspection (Algorithm 3, line 15). Once the selected nodes are presented to the user, some manual inspection is required. Depending on the node type, the user needs to focus on different aspects of the involved functions. For a covariance node, the user has to determine what the two functions do, why their execution times are correlated, and how to de-correlate them. For a variance node, the user must mostly focus on determining whether the function is specific enough. A specific function is

Algorithm 2: Aggregate Segments

Inputs : *sid*, the target semantic interval id,
S, the segment to be aggregated,
B, a hard beginning timestamp to start aggregation (could be *nil*),
E, a ending timestamp to stop aggregation

```
1  function aggregate_segment (sid, S, B, E)
2      if execute_for_target_semantic_interval(S, sid) then
           /* max will handle nil value correctly                    */
3          monitor_exec_time(S.C, S.T, max(B, S.ts), E);
4      end
5      else if is_blocked(S) then
6          S' ← find_waking_segment(S.T, S.te);
7          aggregate_segment(S'.C, S'.T, max(B, S.ts), E);
8      end
9      if B = nil ∨ S.begin() > B then
10         P ← get_previous_segment(S);
11         while P ≠ nil do
12             if B ≠ nil ∧ P.te ≤ B then
13                 return;
14             end
15             if P.sid = sid then
16                 accumulate_wait_time(P.te, S.ts);
17                 break;
18             end
19             P ← get_previous_segment(P);
20         end
21         if P ≠ nil then
22             aggregate_segment(sid, P, B, P.te);
23         end
24         else
25             P ← find_generator(S);
26             if P ≠ nil then
27                 T ← find_generating_time(S);
28                 accumulate_wait_time(T, S.ts);
29                 aggregate_segment(sid, P, B, T);
30             end
31         end
32     end
33 end
```

one that provides sufficient insight on how to reduce the variance. In this process, the number of child functions invokved from the reported function is an important metric

Algorithm 3: Iterative refinement.

Inputs : v: the starting function (i.e., entry point),
k: maximum number of functions to select,
d: threshold for minimum contribution

Output: s^*: top k most responsible factors

```
1  t ← tree with Var(v) as root;
2  l ← {Var(v)};
3  e ← true;
4  while e do
5      foreach factor f ∈ l do
6          if is_variance(f) then
7              c ← var_break_down(f);
8              t.add_children(f, c);
9          end
10     end
11     s* ← select_factors(t, k, d);
12     l.clear();
13     e ← false;
14     foreach factor f ∈ s* do
15         if needs_break_down(f) then
16             l ← l ∪ f;
17             e ← true;
18         else if is_variance(f) then
19             mark_as_selected(f);
20     end
21 end
22 return s*;
```

to consider, but not a decisive one. A function with a large number of child functions is usually not specific enough. However, a function with only a few child functions may not be specific enough either. For example, `row_row_ins_index_entry` is a functions in MySQL that invokes three other functions, `dict_index_is_clust`, `row_ins_clust_index_entry` and `row_ins_sec_index_entry`. Here `ins` is short for `insert`. Even without much understanding of how MySQL works, we can guess from the function names that there are two types of indices in MySQL, `clust_index` and `sec_index`, and this function delegates the work to one of `row_ins_clust_index_entry` and `row_ins_sec_index_entry` depending on the type of index passed into this function. Therefore, although the

number of child functions is very small in this case, we still need to further break down this variance node in order to discover where variance occurs.

Note that, ultimately, VProfiler's output is heuristic. It identifies code that contributes to variance, but a developer must analyze this code to determine if the variance is inherent or is indicative of a performance pathology.

VProfiler uses a parser to automatically inject instrumentation code as a prolog and epilog to each function that is selected for profiling, using a source-to-source translation tool and then recompiling the binary. Our approach is similar to conventional profilers, such as `gprof`, except that VProfiler instruments only a subset of functions at a time.

2.3.4 Implementation

Our current implementation of VProfiler only supports C/C++ applications.[4] VProfiler takes in a list of synchronization primitives used in the application, creates instrumented wrappers for them, and replaces all synchronization function calls with the wrapper calls in order to construct the wake-up graph of the threads and created-by relationships between tasks. The programmer also uses an API provided by VProfiler to mark the creation and completion of a semantic interval, and the places where a thread starts working on behalf of a semantic interval. In addition, the programmer provides a script that copies the instrumented source code files to the source repository, compiles the source code, and runs the application. Given these inputs, VProfiler automatically instruments the appropriate functions, runs the application, and returns k factors with the highest score (k=3 by default). For each selected factor, VProfiler asks the programmer whether to investigate it further or not. (In our experience, in many cases, this decision is usually straightforward and does not require a deep understanding of the source code.) If the programmer deems it necessary, VProfiler re-instruments the selected function(s) and reruns the application to collect new measurements.

2.4 Evaluation

In this section, we aim to evaluate the efficiency of VProfiler. We apply VProfiler to a few complex, real-life software systems (e.g., MySQL), and make modifications to these systems based on VProfiler's findings (we defer this part to the next chapter). The questions then becomes: (1) how much overhead does VProfiler incur to the profiled system, and (2) how much programming effort (e.g., lines of code or hours) is involved in

[4] We also plan to add support for Java applications in our next release.

Config	Function Name	Percentage of Overall Variance
128-WH	`os_event_wait` [A]	37.5%
128-WH	`os_event_wait` [B]	21.7%
128-WH	`row_ins_clust_index_entry_low`	9.3%
2-WH	`buf_pool_mutex_enter`	32.92%
2-WH	`img_btr_cur_search_to_nth_level`	8.3%
2-WH	`fil_flush`	5%

Table 2.1: Key sources of variance in MySQL.

Function Name	Percentage of Overall Variance
`LWLockAcquireOrWait`	76.8%
`ReleasePredicateLocks`	6%

Table 2.2: Key sources of variance in Postgres.

implementing these modifications (to measure how local and specific VProfiler's findings are). We answer the first question in Sections 2.4.1, and the second in Section 2.4.2.

Specifically, we conduct case studies on three popular open-source systems: MySQL (a thread-per-connection database), Postgres (a process-per-connection database), and Apache Web Server (an event-based server application). For MySQL and Postgres, we treat each 'transaction' as a semantic interval, while for Apache we treat each 'web request' as a semantic interval.

Finally, as a measure of practicality of our findings, in Section 3.7 we report on the real-world adoption of some of the optimizations discovered using VProfiler.

In summary, our experiments indicate the following:

1. VProfiler's profiling overhead is an order of magnitude lower than DTrace, and its factor selection algorithm reduces the number of required runs by several orders of magnitude compared to a naïve drill-down strategy.

2. VProfiler successfully reveals the actual sources of variance in these large and complex codebases.

3. VProfiler requires minimum manual effort during both the profiling phase and the optimization phase afterwards.

Function Name	Percentage of Overall Variance
(ap_pass_brigade, apr_file_open)	22%
(ap_pass_brigade, basic_http_header)	15.5%
apr_bucket_alloc	11.8%

Table 2.3: Key sources of variance in Apache HTTPD Server.

Application	Semantic intervals annotations	Avg. manual inspection time per run	Modified lines of code
MySQL	9 lines of code	6 minutes	235
Postgres	7 lines of code	10 minutes	355
Apache	4 lines of code	12 minutes	45

Table 2.4: Our manual effort while using VProfiler.

2.4.1 Instrumentation Overhead

We first report the overhead introduced by VProfiler's online instrumentation. In our case studies, we almost never needed to profile a function with more than 100 children. Nonetheless, as shown in Figure 2.3, we studied the overhead of VProfiler for MySQL as we varied the number of children under an instrumented function from 1 to 500 running the TPC-C workload [160]. As one would expect, the overhead tends to grow as the number of children grows (since we need to measure the execution time of more functions). However, in all cases, overheads are below 14% in terms of both latency and throughput.

As a baseline, we also report the same types of overhead using DTrace, a programmable profiler for troubleshooting arbitrary software. Similar to VProfiler, one can use DTrace to measure the execution time of a parent function and its children, and then compute variances using eq. (2.2).

DTrace's key advantage is that, unlike VProfiler, it instruments the binary code and does not need the source code. However, this flexibility comes at a cost in the performance of the profiling code. As shown in Figure 2.3(left), DTrace's overhead (on both latency and throughput) is significantly higher than VProfiler, and grows rapidly with the number of traced children, whereas VProfiler's overhead stays below 6%. This is expected as DTrace must use heavy-weight mechanisms to inject generalized instrumentation code at run-time, while VProfiler inserts minimal profiling code prior to compilation of the source. DTrace incurred 10-20x higher overheads than our source-level instrumentation, and scales worse when tracing more children. VProfiler gains an advantage over

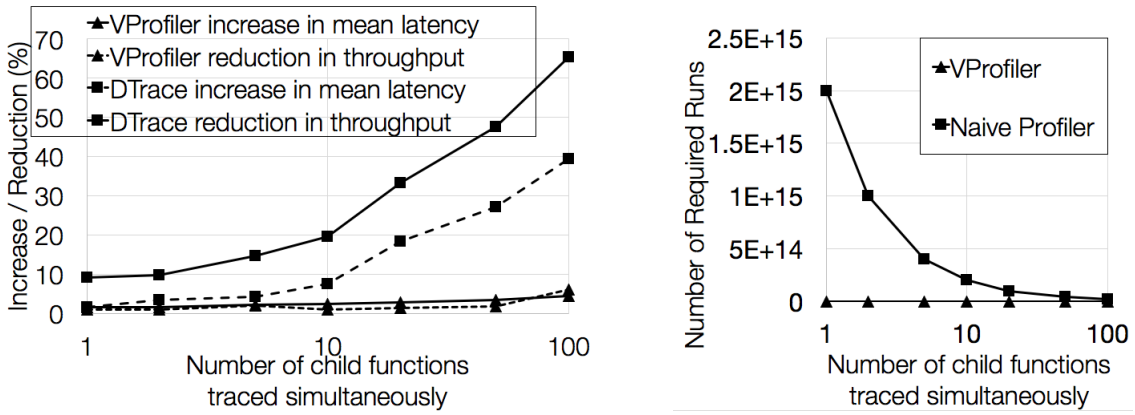

Figure 2.3: (Left) Profiling overhead of VProfiler vs. DTrace. (Right) Number of runs needed for the profiler to identify the main sources of variance.

DTrace because its instrumentation is minimal and inserted into the source, rather than via binary modification.

VProfiler vs. Naïve Profiling— We also compare against a naïve profiling strategy, which is similar to VProfiler, except that it decomposes every factor rather than only a few important ones. In total, there are 2×10^{15} nodes in MySQL's static call graph, 4.5×10^{14} of which are leaves. A naïve profiler has to break down every non-leaf, and thus the number of runs needed is extremely large. VProfiler's selection strategy needs significantly fewer runs to locate the main sources of variance, as confirmed in Figure 2.3(right).

2.4.2 Manual Effort

The manual effort in using VProfiler includes (i) annotating the semantic intervals, (ii) inspecting the variance profile returned at each iteration, and (iii) making enhancements to address pathologies VProfiler identifies. Quantifying these efforts objectively is difficult, as they depend heavily on the programmer's familiarity with the codebase. However, here we simply report the experiences of one of the co-authors (who had no prior experience with these codebases) performing the case studies on MySQL, Postgres and Apache. Table 2.1, 2.2 and 2.3 show the results of the profiles. We leave the detailed discussion of these results and the implications to the next chapter. Here we only focus on the manul efforts during these case studies. As reported in Table 2.4, the annotation of semantic intervals requires only a few lines of code. We expect this to hold in most cases, as the notion of a semantic interval is typically intuitive to developers (e.g., a request or a transaction). Identifying the synchronization functions is similarly straight-forward, as codebases typically use a well-defined API for synchronization. Finally, the actual opti-

Application	Number of VProfiler runs	Variance tree height	Variance tree breadth
MySQL	37	19	245025
Postgres	16	8	16900
Apache	17	15	36

Table 2.5: Statistics of the final variance trees.

mization modifications were quite small, due to the specificity of the functions identified by VProfiler.

2.4.3 Variance Trees

Table 2.5 reports some statistics of the final variance tree for each application. Compared to Postgres and Apache Web Server, studying latency variance in MySQL required more runs but also less time inspecting the returned profile at each run. This is because MySQL's source code is more hierarchical with many functions simply delegating the work to others. For the same reason, the height of the final variance tree of MySQL is larger than the other two. The breadth of the variance tree is mainly affected by the function that has the largest number of children. Note that with factor selection, VProfiler always looks only at k selected factors, and, therefore, most of the nodes in the variance tree are simply (yet safely) ignored.

2.4.4 The Choice of the Specificity Function

As discussed in Section 2.3.2.2, VProfiler uses a quadratic function in its factor selection phase to quantify the specificity of a factor. We experimented with several specificity formulations, including linear, quadratic, and cubic functions. We then compared the quality of their findings. The linear function assigned insufficient weight to the height of a factor, causing an important factor with 18.2% contribution to the overall variance to be missed in an early iteration. On the other hand, the cubic function ultimately yielded exactly the same factors as the quadratic function, providing no additional benefit. Consequently, we have chosen the quadratic function defined equation (2.3) as our default choice of the specificity function in VProfiler.

2.5 Related Work

Call Graph Profilers— These profilers gather execution times and counts for a function and its call descendants [93]. Some tools support shared subroutines, mutual recursion, or dynamic method binding [178]. There are even domain-specific techniques, e.g., for SQL applications [106]. While call graph profilers provide valuable information for long-running operations, VProfiler aims to improve predictability and thus aggregates and reports variance.

Call Path Profilers— Call path profilers can report the resource usage of function calls in their full lexical contexts [98]. Users can learn how much of a program's time is spent in specializable calls to various functions. Techniques to reduce profiling overhead include (i) sampling [88], which collects frequency counts without full instrumentation of procedures' code, and (ii) incremental profiling, which instruments only a few functions of interest [49]. Some profilers [182] extend call path profiling to parallel applications, and use semantic compression to reduce time and space overhead. VProfiler analyzes function invocations in context, but it uses a technique similar to incremental profiling by monitoring only a few functions at a time to reduce overhead. However, VProfiler aggregates over a semantic interval, and selects the most interesting functions at each iteration automatically.

Event Profilers— A variety of tracing tools collect event traces similar to the inputs to VProfiler's analysis. Many of these tools support concurrent and distributed request traces (e.g., [45], [175]). Recent frameworks provide extensible tracing, allowing users to define their own events and provide a LINQ-like query language for trace analysis (e.g. [83], [135]), which allows the introduction of concepts like our semantic intervals into the trace output. However, these profilers are not focused on analysis of variance, and do not provide an equivalent abstraction to our variance tree. VProfiler's post-processing could likely be modified to adopt such tools for managing instrumentation and generating traces in lieu of our source-to-source instrumentation.

Trace Profilers and Statistical Profilers— Trace-based profilers [34, 46, 60, 70, 114, 149, 166, 173, 185, 199] can offer detailed full-system information by instrumenting the source code, from one point in a program to another. However, such profilers are usually post-mortem and the profile data is not available during execution. Moreover, their overhead in tightly-coupled parallel applications can be quite high.

Statistical or sampling-based profilers sacrifice accuracy for lower overhead and online availability: At regular intervals, they probe the program's call stack using interrupts and collect the information they need [53, 93, 139, 140, 177, 194].

VProfiler belongs in the class of trace-based profilers. Its distinguishing contribution lies in capturing semantic intervals across interleaving threads, and identifying informative high-variance functions through the use of the variance tree.

Transactional Profiling— There has been some work on transactional profiling, wherein a transaction is a unit of work similar to our more generic concept of a semantic interval. Whodunit [58] profiles transactions in generic multi-tier applications and can track transactions that flow through shared memory, events, stages, or via message passing, and identify request types that can cause high CPU usage or high contention. VProfiler's goals are quite different from Whodunit in that it seeks to locate functions in the system that cause high variance in latency, whereas the latter focuses on automatically establishing transaction contexts and identifying *request types* that might cause high CPU utilization.

Similarly, AppInsight [165] captures the concept of a transaction for mobile applications. However, AppInsight uses a very limited notion of a transaction, as a user manipulation of the UI and all the operations it triggers.

Instead of profiling transactions, there is also some work on *passively* predicting the performance of transactions using machine learning techniques [142, 143, 202].

Performance Diagnosis— DBSherlock [203] relies on outlier detection and causality analysis to diagnose the root cause of performance anomalies from telemetry data and other statistics (collected from the application and the operating system). Chopstix [51] proposes a diagnostic tool to continuously monitor low-level OS events, such as cache misses, I/O, and locking. Reconstructing these events offline helps users reproduce intermittent bugs that are hard to catch otherwise. X-ray [43] dynamically instruments program binaries and collects performance summaries to find the root cause of performance anomalies. Reference executions can also be used to identify symptoms and causes of performance anomalies [172]. Darc [187] is able to identify the root causes of any peak for a given function by analyzing its latency distribution across multiple runs and determining the major contributor of each bucket. VarianceFinder [169] locates the root causes of variances in a distributed system by using a two-tier method. However, it focuses on the variance of requests taking exactly the same execution path. Spectroscope [170] diagnoses performance changes by comparing request flow during non-problem period and problem period.

VProfiler is also a diagnostic tool that uses instrumentation to collect information that it needs. However, unlike tools that focus on detecting individual anomalies/outliers, VProfiler's approach is based on the mathematical definition of variance. In contrast, Darc finds only the outlying latency contributors, which may or may not be related to large variance contributors. For example, in our case study on Postgres, the latency of the `RecordTransactionCommit` function is only 10% of the `ResourceOwnerRelease` function, while the former contributes 37.7% more to the overall variance than the latter. VProfiler is also capable of profiling multi-threaded programs. VarianceFinder ignores the variance caused by the difference in execution paths for the same type of requests, while VProfiler can also account for such situations. Spectroscope is not applicable to our case, as there is usually no clear definition of a problem period.

Unpredictability in Multi-tier Server Stacks— Many modern applications run in a cloud environment or on top of a complex software stack [71,183,205]. Here, the performance unpredictability could originate in different layers of the system [72,75,132], or be the result of cross-stack communications. While handling this type of unpredictability is out of the scope of this dissertation, we believe that variance trees will shed some light on this problem and plan to pursue this direction in the future.

2.6 Summary

In this chapter, we laid the foundation for our study of performance predictability issues in modern transactional database systems. We presented a novel profiler, called VProfiler, for identifying the major sources of latency variance in a semantical interval of a software system. By breaking down the variance of latency into variances and covariances of functions in the source code and accounting for thread interleavings, VProfiler makes it possible to calculate the contribution of each function to the overall variance.

Chapter 3

Improving Performance Predictability in Existing Database Systems

3.1 Background

In this section, we briefly discuss the scope of our work.

Defining Predictability— There are different mathematical notions for capturing *performance predictability*. One could *minimize* latency variance or seek to impose bounds on high percentile latencies (e.g., limiting 99th percentile latency). Statisticians have also used the ratio of standard deviation to mean (a.k.a coefficient of variation) as a standardized measure of dispersion for a distribution.

Since in this dissertation we target statistical (rather than hard) guarantees, we focus on identifying the sources of latency variance (and thereby standard deviation). Minimizing variance also serves as a surrogate for reducing high-percentile latencies [195]. Hence, our techniques reduce both latency variance and 99th percentile latency (see Section 4.6).

Desirable Solutions— Simply padding all latencies with a large wait time would trivially reduce variance but would also increase mean latency, and, thus, it would have little practical value. While long-running queries and OLAP applications might tolerate an increase in mean latency in exchange for predictability or higher throughput [39, 40, 92, 158, 163, 190], the same tradeoff is less appealing to many latency-critical OLTP applications.[1] Thus, in this dissertation we restrict ourselves to ideal solutions, i.e., those that reduce variance without negatively impacting mean latency or through-

[1]This is perhaps why, despite the success of scan-only query plans in OLAP [55, 158], similar proposals [190] have not had widespread adoption in transaction processing.

put. In fact, not only do our findings reduce variance, but they also *reduce* mean latency and coefficient of variation (see Section 4.6). Such solutions are much more desirable in practice. For example, some of this dissertation's findings have been already adopted (and even made a default policy) by some of the largest open-source communities (Section 3.7).

Inherent versus Avoidable Variance— It is important to note that performance variance is sometimes inherent and cannot be avoided. For example, processing a transaction that updates 10 tables inherently involves more work than one that updates only one table.[2] Avoidable sources of variance are those that are not caused by varying amounts of work requested by the user, but are rather due to internal artifacts of the DBMS itself, such as scheduling choices, contention, I/O, or other performance pathologies in the source code. For example, two transactions requesting similar amounts of work, but experiencing different latencies, indicate a performance anomaly that might be avoidable.

3.2 Case Studies

In this section, we conduct a case study of MySQL and Postgres (as popular open-source traditional DBMSs), and VoltDB (as a popular modern DBMS) to identify their main causes of latency variance.

3.2.1 Latency Variance in MySQL

In this section, we use VProfiler to analyze the source code of MySQL 5.6.23 and characterize the main sources of variance therein. Here, we only report our findings using the TPC-C benchmark. However, in Section 4.6 we evaluate our techniques using five different benchmarks with various degrees of complexity and contention.

Setup— We use the OLTP-Bench [73] framework to run the TPC-C workload under two configurations. First, we study a 128-warehouse configuration with a 30 GB buffer pool on a system with 2 Intel(R) Xeon(R) CPU E5-2450 processors and 2.10GHz cores. Second, we study a reduced-scale 2-warehouse configuration with a 128M buffer pool on a machine with 2 Intel Xeon E5-1670v2 2.5GHz virtual CPUs. The reduced-scale configuration exaggerates buffer pool contention, revealing latency sources that may arise in workloads with a working set significantly larger than the available memory. We

[2] In prior work, we have studied the variance of performance caused by external factors (such as changes in the workload environment) and strategies for mitigating them [144, 145].

Config	Function Name	Percentage of Overall Variance
128-WH	`os_event_wait` [A]	37.5%
128-WH	`os_event_wait` [B]	21.7%
128-WH	`row_ins_clust_index_entry_low`	9.3%
2-WH	`buf_pool_mutex_enter`	32.92%
2-WH	`img_btr_cur_search_to_nth_level`	8.3%
2-WH	`fil_flush`	5%

Table 3.1: Key sources of variance in MySQL.

refer to these configurations as 128-WH and 2-WH, respectively. In both cases, we use a separate machine to issue client requests to the MySQL server.

Summary— Table 3.1 summarizes the key variance sources in MySQL identified by VProfiler. Whereas MySQL has one of the most complex code bases with over 1.5M lines of code and 30K functions, VProfiler narrows down our search by automatically identifying a handful of functions that contribute the most to the overall transaction variance. This demonstrates VProfiler's value: one only needs to manually inspect these few functions to understand whether their execution time variance is inherent or is caused by a performance pathology that can be mitigated or avoided. Next, we explain the role of each function found by VProfiler.

os_event_wait()— MySQL uses its own cross-platform API for synchronization; `os_event_wait` is one of the central functions in this abstraction layer. The implementation of `os_event_wait` yields little insight into why the transaction has to wait. We thus examine the context for the two most significant call sites invoking `os_event_wait` (referred to as A and B in Table 3.1). Both call sites occur within `lock_wait_suspend_thread`, a function used to put a thread to sleep when its associated transaction requests a lock on a record that cannot be granted due to a conflict. These two specific call sites correspond to locks acquired during select and update statements, respectively.

This implies that variability of wait time for contended locks is the largest source of variance in MySQL. Motivated by this finding, we later propose a variance-aware transaction scheduling in Section 3.3, which seeks to minimize variance of wait times by optimizing the order in which locks are granted to waiting threads.

row_ins_clust_index_entry_low()— This function inserts a new record into a clustered index. VProfiler reports that none of this function's children exhibit significant variance, but the main variance arises in the body of the function itself due to varying code paths

Function Name	Percentage of Overall Variance
LWLockAcquireOrWait	76.8%
ReleasePredicateLocks	6%

Table 3.2: Key sources of variance in Postgres.

taken based on the state of the index prior to the insert. The variance here is thus inherent to the index mutation, not a performance pathology.

buf_pool_mutex_enter()— This function is called by other functions when they access the buffer pool. This function is called from various sites, but the call most responsible for its variance occurs in `buf_page_make_young`, which moves a page to the head of the `LRU` list. InnoDB uses this list to maintain the order of buffer page replacements based on a variant of the least recently used algorithm. Upon certain types of accesses, a page is moved to the head of the `LRU` list. Threads must acquire a lock before modifying the list. That lock is acquired in `buf_pool_mutex_enter`. The variance in this function reflects varying wait times, while other threads are reordering the `LRU` list. In Section 3.4.1, we propose a strategy for mitigating this problem.

btr_cur_search_to_nth_level()— This function traverses an index tree level by level to place a tree cursor at a given level, and then it leaves a shared or exclusive lock on the cursor page. Its runtime, thus, varies with the depth to which the tree must be traversed. The variance here is inherent to the index traversal, not a performance pathology.

fil_flush()— MySQL uses `fil_flush` to flush redo logs generated by a transaction. When the operating systems uses disk buffering, the latency variance of disk I/O is exposed in `fil_flush` (rather than the `write` system calls). The variance here is inherent to the I/O, but might be mitigated by logging to faster I/O devices, e.g., [41, 154, 196].

3.2.2 Latency Variance in Postgres

In this section, we use VProfiler to analyze the source code of Postgres 9.6—another popular DBMS. For Postgres, we use the same setup as in Section 3.2.1. Here, we use TPC-C with 32-warehouses and a 30 GB buffer pool. Table 3.2 shows the top two functions in the Postgres source code identified by VProfiler as the main sources of variance (the top source dominates, accounting for 76.8%).

LWLockAcquireOrWait()— Postgres uses write-ahead logging for atomicity and durability; before a transaction commits, all its redo logs must be flushed to disk. To ensure that only one transaction is flushing redo logs at a time, each transaction calls the

`LWLockAcquireOrWait` function to acquire a single global lock, called `WALWriteLock`, before writing its logs. The latency variance in `LWLockAcquireOrWait` is due to varying wait times to acquire this lock. A natural solution is to either reduce contention on this global lock, or to allow multiple transactions to flush simultaneously. The former may be attempted by accelerating I/O (e.g., tuning the I/O block size or placing the logs on NVRAM [41, 196] or SSD [61, 168]), whereas the latter can be attempted by a variety of parallel logging schemes (e.g., [31, 33, 200]). Both strategies have proven effective in improving throughput and mean latencies [41, 196]. However, VProfiler's findings, regarding `LWLockAcquireOrWait`'s contribution to the overall latency variance, call for vetting these strategies in terms of improving predictability as well. We study some of these ideas for Postgres in Sections 3.4.2 and 3.5.4.

ReleasePredicateLocks()— Postgres uses predicate locking to avoid phantom problems (read conflicting with later inserts). As a transaction accesses rows in the database, locks are acquired to prevent other transactions from inserting new rows into its selected range. Upon commit, all its predicate locks are released by calling `ReleasePredicateLocks`. The execution time of this function varies with the number and type of conflicts discovered during this release phase. Since `ReleasePredicateLocks` accounts for only 6% of overall variance, we do not pursue it further.

3.2.3 Latency Variance in VoltDB

We also used VProfiler on VoltDB's source code to identify its major sources of variance. VoltDB is an event-based system in which transactions are wrapped up as stored procedure invocations. Each event needs to wait in a queue before a worker thread is available to process it. VProfiler reports that almost 99.9% of latency variance in VoltDB is due to the variance in the waiting time of these events in different queues. This finding leads to the number of worker threads as a tuning parameter to control the queue size. As shown in Figure 3.1, adjusting this parameter in VoltDB can eliminate 60.9% of the total latency variance, reducing it by 2.6x.

3.3 Variance-Aware Transaction Scheduling

According to VProfiler's findings from Section 3.2, lock wait times account for a significant portion of overall latency variance (over 59.2% in case of MySQL). Hence, in this section we propose a lock scheduling algorithm that can dramatically reduce latency variance.

3.3.1 Problem Setting

Traditional databases often rely on variants of 2-phase locking (2-PL) for concurrency control [1, 12, 21]. Conceptually, each database object b has its own queue Q_b. When a transaction T requests a lock on b, the lock is immediately granted if (i) no other locks are currently held on b by other transactions, or (ii) the current locks on b are compatible with the requested lock type and there are no other transactions currently waiting in Q_b.[3] When a lock on b cannot be granted immediately, transaction T is suspended and placed in Q_b until its lock can be granted. In general, each transaction may wait in multiple queues during its lifetime, and each queue may contain multiple transactions waiting in it. Let $Q_b = \{T_1, \cdots, T_n\}$ denote the transactions currently waiting to be granted a lock on b.

Whenever all the currently held locks on b are released, the *lock scheduling* (a.k.a. *transaction scheduling*) problem is the decision regarding which transaction(s) in Q_b must be granted the lock next. The transaction scheduler might choose one of the exclusive (e.g., write) requests, or choose one or more of the inclusive ones.

The default transaction scheduling in many databases (including MySQL [24] and Postgres [26] among others) is the First-Come-First-Served (FCFS) algorithm. In FCFS, whenever the lock on b becomes available, the transaction which has arrived in Q_b the earliest, say T_e, is granted the lock. Additionally, all the other transactions in Q_b whose requests are compatible with that of T_e are also granted a lock. In other words, T_e is selected based on the amount of time it has spent in the current queue (not in the system). Fairness and simplicity have contributed to FCFS's popularity. However, FCFS does not even minimize mean latency, let alone latency variance.

Challenge of unpredictable remaining times— A key challenge in transaction scheduling is the lack of prior knowledge regarding a transaction's remaining time. In other words, when a transaction arrives in Q_b, the system is only aware of its *age* (i.e., elapsed since its birth), but does not know when it will finish and release its locks once it is granted a lock on b. For example, it may need to wait on a few other locks before it can proceed to completion. In fact, our studies reveal that there is very little correlation between a transaction's age and its overall latency in practice (figure 3.2). Thus, any scheduling strategy must account for the fact that remaining times are unknown and hard to estimate.

[3] To prevent the writes from starving, new read requests may not be granted if there are write requests ahead of them.

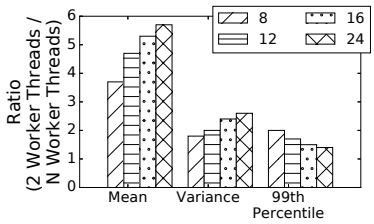

Figure 3.1: Effect of different numbers of worker threads on VoltDB's performance (2 is the default value).

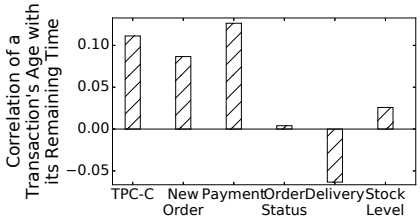

Figure 3.2: Correlation between a transaction's age and its remaining time for different transaction types (TPC-C).

A Convex Loss Function— As discussed in Section 3.1, our ultimate goal is to reduce latency variance and tail latencies. However, solely minimizing variance as a loss function may lead to undesirable side effects. For example, a scheduling algorithm that deliberately adds a large delay to every completed transaction (before allowing it to leave the system) will have a near-zero variance. However, it will also significantly increase mean latency, and is, hence, impractical. To exclude such algorithms, a more effective loss function is the so-called L_p norm, which if minimized, will indirectly reduce both mean and variance (and, thereby, tail) latencies. When n transactions finish with latencies $\langle l_1, \cdots, l_n \rangle$, their L_p norm (denoted as $||.||_p$) is defined as

$$L_p = ||\langle l_1, \cdots, l_n \rangle||_p = (\sum_{i=1}^{n} |l_i|^p)^{1/p} \tag{3.1}$$

where $p \geq 1$ is a real-valued number. The larger the p value, the more we penalize deviations of the l_i values from the mean. For example, as $p \to \infty$, L_p norm approaches the max value of the list. A typical value of p in practice is 2. However, our results in this section hold for all $p \geq 1$ values.

3.3.2 Our VATS Algorithm

Let $A(T)$ denote the age of transaction T when it arrives at a queue Q_b. Q_b is the set of transactions waiting to be granted a lock on b. We define the history H_b of an object b to be the schedule of prior (and current) transactions holding a lock on b. In the following, we drop b from our notation for convenience. Let F be some advice about the future (our algorithm will not make use of such advice, but we will compare our algorithm to other algorithms that may).

A scheduler $S = (S_f, S_a)$ is a set of two functions: $S_f, S_a : H \times Q \times F \rightarrow 2^Q$. When the lock becomes available, the function S_f determines which transactions from Q should be granted a lock. S_f cannot grant two exclusive locks on b simultaneously. When a new transaction arrives at Q, the function S_a decides which transactions should be granted a lock. When other locks are currently held, S_a can only choose from transactions acquiring inclusive locks compatible with the currently held locks.

Let $R(T)$ be a random variable indicating T's remaining time once it is granted a lock on b. Finally, let a menu M be a sequence of transactions, where each transaction has an age and an arrival time at the queue. This will define a problem instance.

We define the p-performance of a schedule S on a menu M to be the expected L_p norm of the vector of transaction completion times of S on M.

Our Algorithm— Given a menu, we aim to design a scheduler that minimizes the expected p-performance. To this end, we define our scheduler as $S^{VATS}=(S_f^{VATS}, S_a^{VATS})$ where:

- S_f^{VATS} grants the lock to the eldest transaction, i.e., one with the largest age.
- S_a^{VATS} never grants any locks.

In general, optimal scheduling is an NP-complete problem when the $R(T)$ values are known [157]. Additionally, the online problem of scheduling even on one processor is impossible to do with a competitive ratio of $O(1)$.[4]

Interestingly, and counter-intuitively, we show that optimal scheduling becomes easier when the remaining times are not known! Specifically, we avoid the above negative results by assuming that $R(T)$ values are i.i.d. random variables drawn from some (unknown) distribution D.[5]

[4]That is, for every scheduler S, there exists a menu M where the optimal offline algorithm performs $\omega(1)$ better than S.

[5]To be more precise, what happens after transactions are granted a lock may depend on our schedule itself, as similar transactions could interact in the future on other queues. For simplicity, in this discussion we ignore this complication.

We now show that our VATS algorithm performs optimally, even against algorithms that know the distribution D (i.e., algorithms that receive $F = D$ as an advice). Note that VATS does not use or need any distributional information or advice on future. Interestingly, this holds even if the menu and distribution are chosen adversarially.

Theorem 3.1. *Fix any menu M, $p \geq 1$, and distribution D with finite expected L_p norm. Let the $R(T)$s be i.i.d random variables drawn from D. Then the p-performance of VATS is optimal against all schedulers, even those that are given D as advice about the future.*

Proof. Assume for the sake of contradiction that there exists a menu M of ℓ transactions $T_1, T_2, \ldots, T_\ell$, where a schedule S has p-performance better than S^{VATS}. We will transform S into S^{VATS} by a series of ℓ transformations: $S = S_0 \to S_1 \to S_2 \to \cdots \to S_\ell = S^{VATS}$. We will show after each transformation that the performance of the schedule improves. This yields a contradiction to the assumption that the p-performance of S was better than that of S^{VATS}.

In the kth transformation, we modify S_{k-1} so that if ever S_{k-1} schedules a transaction $T_{k'} \neq T_k$ when T_k is the eldest transaction in the queue, then S_k will transpose the order of T_k and T'_k, but otherwise run identically to S_{k-1}.

Note that $S_\ell = S^{VATS}$, because S_ℓ will run the eldest transaction, no matter which one it is.

Let $T_{S_{k-1},1}, T_{S_{k-1},2}, \ldots, T_{S_{k-1},\ell}$ and $T_{S_k,1}, T_{S_k,2}, \ldots, T_{S_k,\ell}$ be the order of transactions scheduled in S_{k-1} and S_k respectively. Note that these may be random variables, in that the ith transaction scheduled might depend on the randomness of the scheduler, as well as the time that previous transactions held onto the lock. Let $U_S(T)$ be the time it takes between when T arrives and when the lock is first free under schedule S. Let $W_S(T)$ be the set of transactions scheduled while T is in the queue (including T) under schedule S.

To compare the performance of S_{k-1} and S_k, we create a coupling between two different drawings D_1 and D_2 of the $R(\cdot)$s so that for all i, $R_{D_1}(T_{S_{k-1},i}) = R_{D_2}(T_{S_k,i})$. First note that there is no dependency problem here because (by induction on i) under this coupling, $T_{S_{k-1},i}$ and $T_{S_k,i}$ will be scheduled at the same time. Also, since the $R(\cdot)$s are all drawn i.i.d, this is a valid coupling, which is to say that D_1 and D_2 are (marginally) drawn from the same distribution. Note that the performance of S_{k-1} and S_k are respec-

tively,

$$\int_{D_1} \left(\sum_i |A[T_{S_{k-1},i}] + U_{S_{k-1}}(T_{S_{k-1},i}) + \sum_{T_j \in W_{S_{k-1}}(T_{S_{k-1},i})} R(T_j)|^p \right)^{1/p}$$

and

$$\int_{D_2} \left(\sum_i |A[T_{S_k,i}] + U_{S_k}(T_{S_k,i}) + \sum_{T_j \in W_{S_k}(T_{S_k,i})} R(T_j)|^p \right)^{1/p}$$

To show that the first is greater than the second, we fix some realization of D_1. Using our coupling, this gives us a realization of D_2. We will show that no matter what the realization is we have:

$$\begin{aligned} &\sum_i |A[T_{S_{k-1},i}] + U_{S_{k-1}}(T_{S_{k-1},i}) + \sum_{T_j \in W_{S_{k-1}}(T_{S_{k-1},i})} R(T_j)|^p \\ &< \sum_i |A[T_{S_k,i}] + U_{S_k}(T_{S_k,i}) + \sum_{T_j \in W_{S_k}(T_{S_k,i})} R(T_j)|^p \end{aligned}$$

Note that the summands are identical except, possibly, for the terms of T_k and $T_{k'}$. Let $W_k = W_{S_{k-1}}(T_k) \cap W_{S_k}(T_k)$ be the transactions scheduled while T_k is in the queue in both schedules. Define $W_{k'}$ analogously. Let W' be the transactions scheduled between k and k'. Then, $W_{S_{k-1}}(T_k) = W_k \cup \{T_{k'}\} \cup W'$, $W_{S_{k-1}}(T_{k'}) = W_{k'}$, $W_{S_k}(T_k) = W_k$, and $W_{S_k}(T_{k'}) = W_{k'} \cup \{T_{k'}\} \cup W'$.

The rearrangement inequality states that if x_1, x_2, y are all nonnegative numbers then $|x_1 + y|^p + |x_2|^p \leq |x_1|^p + |x_2 + y|^p$ if and only if $x_1 \leq x_2$. We apply the rearrangement inequality where:

$$x_1 = A(T_{k'}) + U_{S_{k-1}}(T_{k'}) + \sum_{T_j \in W_{k'}} R_{D_1}(T_j)$$

$$x_2 = A(T_k) + U_{S_{k-1}}(T_k) + \sum_{T_j \in W_k} R_{D_2}(T_j)$$

$$y = R_{D_1}(T_{k'}) + \sum_{T_j \in W'} R(T_j) = R_{D_2}(T_k) + \sum_{T_j \in W'} R(T_j).$$

The age of $T_{k'}$ when it is scheduled in S_{k-1} is $x_1 - R_{D_1}(T_{k'})$, and the age of T_k when $T_{k'}$ is scheduled in S_{k-1} is $x_2 - R_{D_2}(T_k)$. Since, at the time $T_{k'}$ is scheduled in S_{k-1}, T_k is older than $T_{k'}$, and since $R_{D_1}(T_{k'}) = R_{D_2}(T_k)$, we have that $x_1 < x_2$.

The theorem follows by noting that in the S_{k-1} schedule, the $T_{k'}$ term is x_1 and the T_k term is $x_2 + y$; while in the S_k schedule, the $T_{k'}$ term is $x_1 + y$ and the T_k term is x_2. □

In practice, we observe that $R(T)$ has a near-zero correlation with $A(T)$ (Section 3.5.6). Thus, the I.I.D. assumption of Theorem 3.1 seems plausible. Interestingly, even if the variance of the execution times were 0 (i.e., a correlation of -1), our theorem would be even more true, as not only would VATS gain by avoiding losses from old transactions, but it would also gain because such transactions would complete and release their locks faster.

In our implementation, we slightly modify VATS such that it grants as many locks as possible if a lock does not conflict with any of the locks in front of it in the queue (including both the granted locks and the ones still waiting), which is preserved in an eldest-first order, as a means to improve performance. We evaluate VATS in Section 3.5.2.

3.4 Additional Strategies

Based on our findings from Section 3.2, we present further strategies for improving performance predictability. Unlike our VATS algorithm which is a generic way of reducing variance in wait times, our techniques in this section are specific to MySQL, Postgres, and VoltDB. We use these techniques to illustrate VProfiler's effectiveness in localizing the sources of variance in a massive and complex codebase (e.g., MySQL or Postgres). VProfiler enables us to drastically reduce overall variance with minimal modification (ranging from changing tuning parameters to a few hundred lines of code) by examining only a handful of functions out of tens of thousands (see Section 4.6).

3.4.1 Lazy LRU Update (LLU)

As noted in Section 3.2.1, the lock on the LRU list is a main source of variance in MySQL when the working set exceeds the buffer pool size. Algorithm 4 shows how the LRU list is updated in MySQL. First, a mutex is acquired by calling `buf_pool_mutex_enter`, and then the page is moved to the head of the list by calling `buf_page_make_young`.

For better cache performance, MySQL does not implement the strict LRU policy. Instead, it splits the LRU list into two sublists, *young* and *old*. Replacement victims are selected from the old list, which by default contains 3/8 of the oldest pages. When a

Algorithm 4: How the LRU list is updated in MySQL

Inputs : p: the page to be moved to start of the `LRU` list;
b: the buffer pool

```
1 buf_pool_mutex_enter(b);
2 buf_LRU_make_block_young(b, p);
3 buf_pool_mutex_exit(b);
```

page is accessed, if it is currently in the old list, it is moved to the head of the young list, and the tail of the young list is placed at the head of the old list. To avoid frequent reordering of the list, MySQL does not maintain precise LRU ordering within the young list. However, when the working set exceeds 5/8 of the buffer pool, old pages are accessed frequently, and the lock on the LRU list becomes a bottleneck. Our idea is to further relax the precision of LRU tracking to avoid this contention, as described next.

To avoid excessive delays, our proposed algorithm, Lazy LRU Update (LLU), limits the time that `buf_pool_mutex_enter` waits for the lock. Specifically, we replace the mutex with a spin lock to control the wait time. When the buffer pool is sufficiently large, this lock is typically uncontended, and the overhead of a spin lock remains minimal. However, if a waiting thread cannot acquire the lock within 0.01ms, we abandon the attempt to update the global list and instead add the page to a thread-local backlog of deferred LRU updates, l. Later, when `buf_pool_mutex_enter` successfully acquires the lock for a different page, we first process the pages in l (after confirming that they have not been evicted) before moving the page that triggered the reordering.

3.4.2 Parallel Logging

As revealed by VProfiler in Section 3.2.2, over 70% of latency variance in Postgres is due to the variation of wait times in redo log flush operations. This leads us to another strategy for improving predictability: parallel logging, so that when a log file is unavailable, a transaction can write to other log files instead of having to wait. While there are sophisticated parallel logging schemes [31, 33, 200], we implement a simple variant that allows Postgres to use two hard disks for storing two sets of redo logs. A transaction only waits when neither of these sets is available, in which case it waits for the one with fewer waiters. Though parallel logging is well-studied for improving mean latencies, we vet its effectiveness in reducing latency variance in Section 3.5.4.

3.4.3 Variance-Aware Tuning

In many cases, the behavior of the culprit function identified by VProfiler can be controlled through external tuning parameters of the DBMS. Specifically, (i) in MySQL, `buf_pool_mutex_enter()` leads us to buffer pool size while `fil_flush()` leads us to `innodb_flush_log_at_trx_commit` parameter, (ii) in Postgres, `LWLockAcquireOrWait()` leads us to I/O block size, and (iii) in VoltDB, the queuing delay leads us to the number of worker threads.

3.5 Experiments

Our experiments aim to answer three key questions: (1) How effective are our techniques (VATS, LLU, parallel logging, and variance-aware tuning) in reducing tail latency and latency variance? (2) Does our reduction of latency variance come at the cost of sacrificing mean latency or throughput? (3) How effective and efficient is VProfiler compared to other profiling alternatives? In summary, our results indicate that:

- For contended workloads (TPC-C, SEATS, and TATP), our VATS algorithm makes the DBMS significantly more predictable (and even faster) without compromising throughput, with up to 6.3x, 5.6x, and 2.0x lower mean, variance, and 99th percentile latencies, respectively. As expected, for non-contended workloads (Epinions and YCSB), the choice of scheduling algorithm is immaterial. (Section 3.5.2)
- Our Lazy LRU Update algorithm makes MySQL faster and more predictable, with 1.4x, 1.2x, and 1.2x lower mean, variance, and 99th percentile latencies, respectively. (Section 3.5.3)
- Parallel logging improves both predictability and overall performance of Postgres, with 1.8x, 1.3x, and 2.4x lower mean, variance, and 99th percentile latencies, respectively. Also, variance-aware tuning can eliminate up to 88.3% of the overall latency variance. (Sections 3.5.4 and 3.5.5)
- VProfiler's profiling overhead is an order of magnitude lower than that of DTrace, and its factor selection algorithm reduces the number of required runs by several orders of magnitude compared to a naïve strategy. (Section 2.4.1)

These results are summarized in Table 3.3. We have also included and additional experiments in Section 3.5.6, showing that there is no correlation between a transaction's remaining time and its age (3.5.6).

System	Name of the Identified Function	Original contribution to overall variance	Modification	Modified lines of code or config	Ratio of overall latency variances (Orig. / Modified)	Ratio of overall 99^{th} latencies (Orig. / Modif.)	Ratio of overall mean latencies (Orig. / Modif.)
MySQL	`os_event_wait`	59.2%	replace FCFS with VATS	189	5.6x	2.0x	6.3x
MySQL	`buf_pool_mutex_enter`	32.92%	replace mutex with spin lock	46	1.6x	1.4x	1.1x
MySQL	`fil_flush`	5%	parameter tuning	2	1.4x	1.2x	1.2x
Postgres	`LWLockAcquireOrWait`	76.8%	parallel logging	355	1.8x	1.3x	2.4x
VoltDB	`[waiting in queue]`	99.9%	add # of worker threads	1	2.6x	1.4x	5.7x

Table 3.3: Impact of modifying each of the functions identified by VProfiler. The last 3 columns compare end-to-end transaction latencies before and after each modification. For example, modifying `os_event_wait` eliminates more than 82% of MySQL's total latency variance, i.e., the ratio of the transaction variance of original MySQL to modified MySQL is $1/(1-0.82)=5.56$.

3.5.1 Experimental Setup

The hardware and software used for our experiments in this section are the same as Section 3.2. For fairness, we used the same throughput of 500 transactions per second across all workloads and algorithms. Also, to rule out the effect of external load changes on latency variance, we used the OLTP-Bench [73] tool to sustain a constant throughput throughout the experiment, and measured mean, variance, and 99th percentile latencies for each algorithm and workload. In addition to TPC-C, we also used the following workloads for a more extensive evaluation:

- **SEATS [180]:** This benchmark is a simulation of an airline ticketing system where customers search flights and make online reservations. In our experiments, we used a scale factor of 50, leading to a highly contended workload.
- **TATP [197]:** TATP models a typical caller location system used by tele-communication providers. For TATP, we used a scale factor of 10, making it a contended workload (but not as contended as TPC-C).
- **Epinions [138]:** Epinions simulates a customer review website where users interact and write reviews for various products. We used a scale factor of 500 in our experiments. This workload has a very low contention.
- **YCSB [66]:** YCSB is a set of micro-benchmarks simulating data management applications that have simple workloads but require high scalability. The scale factor used was 1200, causing little or no contention.

Given that varying lock wait times are a major problem for MySQL, we evaluate VATS using MySQL. We evaluate LLU and parallel logging using MySQL and Postgres, respectively. Finally, we study variance-aware tuning for all three: MySQL, Postgres

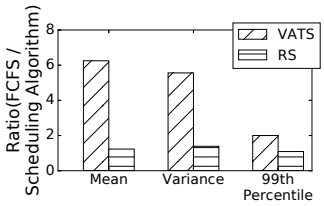

Figure 3.3: Effect of different scheduling algorithms on MySQL performance. For example, replacing FCFS with VATS makes MySQL 6.3x faster and 5.6x lower in variance.

and VoltDB. When results are similar across all workloads, we only report numbers for TPC-C as a representative workload.

3.5.2 Studying Different Scheduling Algorithms

We compare VATS to two other scheduling algorithms:

- **First Come First Served (FCFS):** This is the default scheduling in many DBMSs (including MySQL & Postgres).
- **Randomized Scheduling (RS):** Similar to VATS, except that transactions are sorted according to a random order rather than by age.

The results are shown in Figure 3.3 for TPC-C (see Table 3.4 for other workloads). In summary, FCFS is the least efficient scheduling algorithm for all three contended workloads. For example, for TATP, even a random scheduling (RS) improves upon FCFS by 25% in terms of latency variance. However, the randomness of RS can also be harmful. For SEATS, RS performs about 2 orders of magnitude worse than other algorithms (results omitted for space). The choice of lock scheduling algorithm does not make a difference for YCSB simply because it does not have any lock contention. In case of Epinions, the improvement is due to the fact that we place newly-granted locks at the head of the list, and thus the time for traversing the list is reduced (MySQL uses a global hash table where each bucket is a linked list storing some of the lock objects).

We have summarized VATS's improvement over FCFS in Table 3.4 for all workloads. Our VATS algorithm is consistently superior for contended workloads and comparable to no-contention ones. Most notably, VATS eliminates 84.1% of the entire latency variance of MySQL for TPC-C. In other words, replacing FCFS with VATS makes MySQL's latency variance 6.3x lower. On average, this number is 2.9x for all contended workloads, and 2.4x over all five workloads.

	Workload	Mean Latency	Variance	99th Percentile
Contended	TPCC	6.3x	5.6x	2.0x
	SEATS	1.1x	1.3x	1.1x
	TATP	1.2x	1.6x	1.3x
	Avg	**2.9x**	**2.8x**	**1.5x**
No Contention	Epinions	1.4x	2.6x	1.0x
	YCSB	1.0x	1.1x	1.1x

Table 3.4: Comparing VATS with MySQL's original (FCFS) lock scheduling in terms of overall transaction latency.

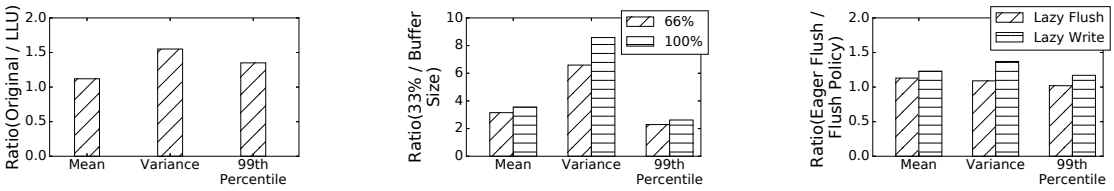

Figure 3.4: Effect of LLU, buffer pool size (in % of the entire database size), and log flush policy on MySQL (TPC-C).

3.5.3 Lazy LRU Update Algorithm

In this section, we evaluate our Lazy LRU Update (LLU) algorithm. We produce a memory-contended workload using the same 2-WH configuration from Section 3.2.1. As shown in Figure 3.4(left), LLU yields a more predictable (and even slightly faster) MySQL with 1.1x, 1.6x, and 1.4x lower mean, variance, and 99th percentile latencies. This improvement is because LLU avoids extremely long waits, delaying the re-ordering of buffer pages until the overhead is fairly low. This reduces the contention on the LRU data structure for memory-contended workloads.

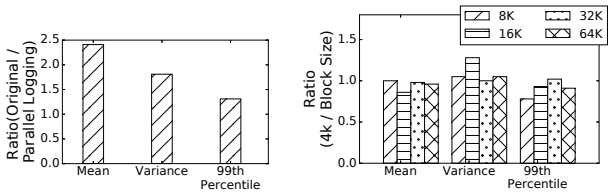

Figure 3.5: Effect of parallel logging and redo log block size on Postgres (TPC-C).

3.5.4 Parallel Logging

As discussed in Section 3.4.2, we implement a parallel logging scheme for Postgres. As shown in Figure 3.5(left), this significantly reduces mean, variance, and 99th percentile latencies, lowering them by 2.4x, 1.8x, and 1.3x, respectively.

3.5.5 Variance-Aware Tuning

In Section 3.4.3, we identified several tuning parameters in MySQL, Postgres, and VoltDB that affect latency variance.

We first investigate the buffer pool size for MySQL and TPC-C, as shown in Figure 3.4(center). We set the buffer pool size to 33%, 66%, and 100% of the overall database size, and report their relative performance compared to 33%. As expected, a larger buffer pool retains more data in memory, thus effectively reducing the number of page evictions, the number of I/O operations, and the degree of contention within the buffer pool. Consequently, the larger the buffer pool, the lower the mean, variance, and 99th percentile latencies. Ideally, a buffer pool as large as the entire database is recommended for both better average performance and greater predictability. However, depending on the working set size, a smaller buffer pool might be economically more appealing, while producing comparable results.

Second, we investigate MySQL's log flushing policies, as shown in Figure 3.4(right). The results indicate that deferring both write and flush operations to a log flusher thread minimizes transaction variances. This is not surprising: eagerly flushing logs prior to commit places highly variable disk write latencies on the transaction execution path. However, lazy flushing may lose forward progress (committed transactions) in the event of a crash.

In Postgres, another strategy for reducing the variance of redo log flushes is to accelerate the I/O operations by tuning an appropriate block size (see Section 3.4.3), which is by default 8 KB. Figure 3.5(right) shows that increasing the block size can reduce variance, but only to a certain extent. A larger block can reduce the number of write

operations per transaction, but when it becomes so large that the generated log records only occupy a small portion of a block, the transaction still has to write the whole block. In such cases, the disadvantage of writing more data than needed outweighs the advantage of fewer writes.

Finally, we explore the effect of number of worker threads on VoltDB's performance. In a nutshell, adjusting this parameter in VoltDB can eliminate 60.9% of the total latency variance, i.e., lower it by 2.6x (see Section 3.2.3).

3.5.6 Correlation of Transaction Age and Remaining Time

One might imagine that the larger a transaction age, the smaller its remaining time. Interestingly, this is not the case in practice due to the intertwined nature of contended transactions. Figure 3.2 shows the correlation between a transaction's age and its remaining time at the moment when scheduling decisions are made. As shown in Figure 3.2, the correlation of these two values is quite small regardless of the transaction type, indicating the difficulty in predicting the remaining time of a transaction given its age.

3.6 Related Work

Predictable Query Plans— There has been some pioneering work on enriching query optimizers to account for parameter uncertainties (caused by cardinality and cost estimates) when choosing a query plan [59, 64].

Florescu and Kossman [86] have taken the opposite direction by arguing for a radical DBMS redesign. They propose a new tiered architecture for web applications, where consistency maintenance is moved from DBMS to application layer. Others have advocated the use of table scans for all queries [42, 103, 158, 163, 190], or simply restricted themselves to query plans with a bounded worst-case [39, 40]. Although many of these techniques share scans and joins across multiple queries, and use always-on operators to reduce execution time [42, 55, 92, 103, 158, 190], they still have a negative impact on average latency. For instance, some of these approaches achieve predictability at the cost of increasing latency by 1.6x [59] or 3x [92]. As such, these solutions are more appropriate for long-running decision support queries than transaction processing. Thus, while successfully adopted by OLAP vendors [55, 158], these proposals have not had widespread adoption by major OLTP vendors, as foregoing low latency to achieve predictability is an unattractive trade-off for many latency-critical and transactional applications (see 'Desirable Solutions' in Section 3.1).

Instead of requiring richer statistics or dismissing traditional query optimizers altogether, we take a top-down approach (see Section 4.1 for the distinction) by carefully studying the entire source code of existing database systems to quantify and mitigate their root causes of performance variance. Moreover, we seek practical solutions that reduce variance without sacrificing mean latency, a decision that has helped the real-world adoption of our proposal (Section 3.7).

Real-Time Databases— Once an active area of research in the 1990s, real-time databases (RTDBs) [28, 119, 152] sought real-time performance guarantees by (i) requiring each transaction to provide its own deadline, and (ii) minimizing deadline violations by restricting themselves to mechanisms that bounded worst case execution times. In contrast, we study predictability in the context of today's conventional best-effort transaction processing systems, where sacrificing throughput or mean latency to obtain hard bounds on execution time may not be an appealing trade-off.

Variance-Aware Job Scheduling— Outside a database context, theoretical literature has examined the problem of scheduling general tasks to minimize completion time variance (CTV) and waiting time variance (WTV). These formulations assume a set of jobs with *known processing times* and seek a schedule that minimizes the variance of their completion or wait times. While CTV and WTV problems are both NP-complete [47, 123], there are several heuristics [62, 79, 201], dynamic programming solutions [69, 124], and polynomial-time approximations [125]. These techniques assume an *offline* setting, and thus do not apply to our transaction scheduling problem, since the remaining and arrival time of transactions are unknown in practice. In contrast, our VATS algorithm does not require such knowledge.

Profiling Literature— There is a large body of work on profiling techniques [49, 93, 98, 178, 182]. In a nutshell, VProfiler is the first profiler to systematically break down the contribution of individual functions to the overall latency variance and, with minimal help from programmers, distinguish execution times that are relevant to transaction latencies.

Performance Diagnosis and Prediction— There are several tools that help users diagnose performance anomalies or reproduce intermittent bugs, either by monitoring fine-grained, low-level OS events [51] or by collecting statistics from the application and the OS for post-mortem analysis [43, 203]. In contrast, we focus on finding the internal causes of performance variance by instrumenting the application code and relying on the mathematical definition of variance to narrow its search space.

Instead of profiling transactions, there is also some work on passively predicting the performance of transactions using machine learning techniques [142, 143, 202]. By reducing the performance variance, our work should ultimately make performance predictions easier.

3.7 Real-world Adoption

After observing the considerable impact of our small modifications on performance predictability (Table 3.3), we decided to share these results with the open-source community. In particular, our VATS algorithm was quickly adopted by MySQL distributions, and has even been made a default policy by MariaDB [6]. These MySQL distributions comprise over 2M+ installations around the world.

Meanwhile, the issue with LRU mutex contention found by VProfiler was independently identified by the MySQL community and addressed by multi-threaded flushing [19, 25] and other techniques [20, 22]. While their solution differs from our LLU technique, they still confirm the validity of VProfiler's finding regarding the cause of the performance pathology.

3.8 Summary

In this chapter, we presented our findings of applying VProfiler on several popular and complex open-source transactional database systems. We showed that, with VProfiler, we were able to identify the exact functions and their contributions to the variance of transaction latency under different configurations. For each of the root causes we found, we proposed solutions to mitigate it, either by introducing new algorithms, changing the implementation, or tuning configuration parameters that are related to those root causes. With just small modifications, we were able to improve not only on performance variance, but also on average performance.

Specifically, the most significant root cause we found from MySQL (i.e., time spent on waiting for locks) led us to a new opportunity for further research. Inspired by this finding, and our proposed VATS algorithm, we sought to further investigate the potential of lock scheduling in transactional database systems, as we explore in the next chapter.

Chapter 4

Contention-Aware Transaction Scheduling

4.1 Introduction

Lock management forms the backbone of concurrency control in modern software, including many distributed systems and transactional databases. A lock manager guarantees both correctness and efficiency of a concurrent application by solving the data contention problem. For example, before a transaction accesses a database object, it has to acquire the corresponding lock; if the transaction fails to get a lock immediately, it is blocked until the system grants it the lock. This poses a fundamental question: when multiple transactions are waiting for a lock on the same object, which should be granted first when the object becomes available? This question, which we call *lock scheduling*, has received surprisingly little attention, despite the large body of work on concurrency control and locking protocols [27, 50, 56, 68, 107, 113, 128, 133, 171]. In fact, almost all existing DBMSs[1] rely on variants of the first-in, first-out (FIFO) strategy, which grants (all) compatible lock requests based on their arrival time in the queue [3, 8, 10, 11, 13]. In this dissertation, we carefully study the problem of lock scheduling and show that it has significant ramifications on overall performance of a DBMS.

Related Work— There is a long history of research on scheduling in a general context [76, 97, 108, 109, 153, 156, 174, 176], whereby a set of jobs is to be scheduled on a set of processors such that a goal function is minimized, e.g., the sum of (weighted) completion times [104, 108, 156] or the variance of the completion or wait times [48, 54, 80, 122, 191].

[1]The only exceptions are MySQL and MariaDB, which have recently adopted our Variance-Aware Transaction Scheduling (VATS) [110] (see Section 4.8).

There is also work on scheduling in a real-time database context [27, 99, 107, 188, 189], where the goal is to minimize the total tardiness or the number of transactions missing their deadlines.

In this dissertation, we address the problem of lock scheduling in a transactional context, where jobs are transactions and processors are locks, and the scheduling decision is about which locks to grant to which transactions. However, our transactional context makes this problem quite different than the well-studied variants of the scheduling problem. First, unlike generic scheduling problems, where at most one job can be scheduled on each processor, a lock may be held in either exclusive or shared modes. The fact that transactions can sometimes share the same resources (i.e., shared locks) significantly complicates the problem (see Section 4.2.4). Moreover, once a lock is granted to a transaction, the same transaction may later request another lock (as opposed to jobs requesting all of their needed resources upfront). Finally, in the scheduling literature, the execution time of each job is assumed to be known upon its arrival [48, 130, 176, 191], whereas the execution time of a transaction is often unknown *a priori*.

Although there are scheduling algorithms designed for real-time databases [38, 134, 179, 198], they are not applicable in a general DBMS context. For example, real-time settings assume that each transaction comes with a deadline, whereas most database workloads do not have explicit deadlines. Instead, most workloads wish to minimize latency or maximize throughput.

Challenges— Several aspects of lock scheduling make it a uniquely challenging problem, particularly under the performance considerations of a real-world DBMS.

1. **An online problem.** At the time of granting a lock to a transaction, we do not know when the lock will be released, since the transaction's execution time will only be known once it is finished.
2. **Dependencies.** In a DBMS, there are dependencies among concurrent transactions when one is waiting for a lock held by another. In practice, these dependencies can be quite complex, as each transaction can hold locks on several objects and several transactions can hold shared locks on the same object.
3. **Non-uniform access patterns.** Not all objects in the database are equally popular. Also, different transaction types might each have a different access pattern.
4. **Multiple locking modes.** The possibility of granting a lock to one writer exclusively or to multiple readers is a source of great complexity (see Section 4.2.4).

Contributions— In this dissertation, to the best of our knowledge, we present the first formal study of lock scheduling problem with a goal of minimizing transaction latencies

in a DBMS context. Furthermore, we propose a contention-aware transaction scheduling algorithm, which captures the contention and the dependencies among concurrent transactions. The key insight is that a transaction blocking many others should be scheduled earlier. We carefully study the difficulty of lock scheduling and the optimality of our algorithm. Most importantly, we show that our results are not merely theoretical, but lead to dramatic speedups in a real-world DBMS. Despite decades of research on all aspects of transaction processing, lock scheduling seems to have gone unnoticed, to the extent that nearly all DBMSs still use FIFO. Our ultimate hope is that our results draw attention to the importance of lock scheduling on the overall performance of a transactional system.

In summary, we make the following contributions:

1. We propose a contention-aware lock scheduling algorithm, called Largest-Dependency-Set-First (LDSF). We prove that, in the absence of shared locks, LDSF is optimal in terms of the expected mean latency (Theorem 4.2). With shared locks, we prove that LDSF is a constant factor approximation of the optimal scheduling under certain regularity constraints (Theorem 4.3).
2. We propose the idea of granting only *some* of the shared lock requests on an object (as opposed to granting them *all*). We study the difficulty of the scheduling problem under this setting (Theorem 4.5), and propose another algorithm, called bLDSF (batched Largest-Dependency-Set-First), which improves upon LDSF in this setting. We prove that bLDSF is also a constant factor approximation of the optimal scheduling (Theorem 4.6).
3. In addition to our theoretical analysis, we use a real-world DBMS and extensive experiments to empirically evaluate our algorithms on the TPC-C benchmark, as well as a microbenchmark. Our results confirm that, compared to the commonly-used FIFO strategy, LDSF and bLDSF reduce mean transaction latencies by up to 300x and 290x, respectively. They also increase throughput by up to 6.5x and 5.5x. As a result, LDSF (which is simpler than bLDSF) has already been adopted as the default scheduling algorithm in MySQL [2] as of 8.0.3+.

4.2 Problem Statement

In this section, we first describe our problem setting and define dependency graphs. We then formally state the lock scheduling problem.

4.2.1 Background: Locking Protocols

Locks are the most commonly used mechanism for ensuring consistency when a set of shared objects are concurrently accessed by multiple transactions (or applications). In a locking system, there are two main types of locks: shared locks and exclusive locks. Before a transaction can read an object (e.g., a row), it must first acquire a shared lock (a.k.a. read lock) on that object. Likewise, before a transaction can write to or update an object, it must acquire an exclusive lock (a.k.a. write lock) on that object. A shared lock can be granted on an object as long as no exclusive locks are currently held on that object. However, an exclusive lock on an object can be granted only if there are no other locks currently held on that object. We focus on the strict 2-phase locking (strict 2PL) protocol: once a lock is granted to a transaction, it is held until that transaction ends. Once a transaction finishes execution (i.e., it commits or gets aborted), it releases all of its locks.

4.2.2 Dependency Graph

Given the set T of transactions currently in the system, and the set O of objects in the database, we define the dependency graph of the system as an edge-labeled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$. The vertices of the graph $\mathcal{V} = T \cup O$ consist of the current transactions and objects. The edges of the graph $\mathcal{E} \subseteq T \times O \cup O \times T$ describe the locking relationships among the objects and transactions. Specifically, for transaction $t \in T$ and object $o \in O$,

- $(t, o) \in \mathcal{E}$ if t is waiting for a lock on o;
- $(o, t) \in \mathcal{E}$ if t already holds a lock on o.

The label $\mathcal{L} : \mathcal{E} \to \{S, X\}$ indicates the lock type:

- $\mathcal{L}(t, o) = X$ if t is waiting for an exclusive lock on o;
- $\mathcal{L}(t, o) = S$ if t is waiting for a shared lock on o;
- $\mathcal{L}(o, t) = X$ if t already holds an exclusive lock on o;
- $\mathcal{L}(o, t) = S$ if t already holds a shared lock on o.

We assume that deadlocks are rare and are handled by an external process (e.g., a deadlock detection and resolution module). Thus, for simplicity, we assume that the dependency graph $\mathcal{G}$ is always a directed acyclic graph (DAG).

4.2.3 Lock Scheduling

A lock scheduler makes decisions about which transactions are granted locks upon one or both of the following events: (i) when a transaction requests a lock, and (ii) when a lock is released by a transaction.[2] Let $\mathbb{G}$ be the set of all possible dependency graphs of the system. A scheduling algorithm $\mathcal{A} = (\mathcal{A}_{req}, \mathcal{A}_{rel})$ is a pair of functions $\mathcal{A}_{req}, \mathcal{A}_{rel} : \mathbb{G} \times O \times T \times \{S, X\} \to 2^T$. For example, when transaction t requests an exclusive lock on object o, $\mathcal{A}_{req}(\mathcal{G}, o, t, X)$ determines which of the transactions currently waiting for a lock on o (including t itself) should be granted their requested lock on o, given the dependency graph $\mathcal{G}$ of the system. (Note that the types of locks requested by transactions other than t are captured in $\mathcal{G}$.) Likewise, when transaction t releases a shared lock on object o, $\mathcal{A}_{rel}(\mathcal{G}, o, t, S)$ determines which of the transactions currently waiting for a lock on o should be granted their requested lock, given the dependency graph $\mathcal{G}$. When all transactions holding a lock on an object o release the lock, we say that o has *become available*. When the lock request of a transaction t is granted, we say that t is scheduled.

Since the execution time of each transaction is typically unknown in advance, we model their execution time using a random variable with expectation R. Given a particular scheduling algorithm $\mathcal{A}$, we define the latency of a transaction t, denoted by $l_{\mathcal{A}}(t)$, as its execution time plus the total time it has been blocked waiting for various locks. Since $l_{\mathcal{A}}(t)$ is a random variable, we denote its expectation as $\bar{l}_{\mathcal{A}}(t)$. We use $\bar{l}(\mathcal{A})$ to denote the expected transaction latency under algorithm $\mathcal{A}$, which is defined as the average of the expected latencies of all transactions in the system, i.e., $\bar{l}(\mathcal{A}) = \frac{1}{|T|} \sum_{t \in T} \bar{l}_{\mathcal{A}}(t)$.

Our goal is to find a lock scheduling algorithm under which the expected transaction latency is minimized. To ensure consistency and isolation, in most database systems $\mathcal{A}_{req}$ simply grants a lock to the requesting transaction only when (i) no lock is held on the object, or (ii) the currently held lock and the requested lock are compatible and no transaction in the queue has an incompatible lock request. This choice of $\mathcal{A}_{req}$ also ensures that transactions requesting exclusive locks are not starved. The key challenge in lock scheduling, then, is choosing an $\mathcal{A}_{rel}$ such that the expected transaction latency is minimized.

[2] These are the only situations in which the dependency graph changes. If a scheduler grants locks at other times, the same decision could have been made upon the previous event, i.e., a transaction was unnecessarily blocked. A lock scheduler is thus an event-driven scheduler.

Notation	Description
T	the set of transactions in the system
O	the set of objects in the database
$\mathcal{G}$	the dependency graph of the system
$\mathcal{V}$	vertices in the dependency graph
$\mathcal{E}$	edges in the dependency graph
$\mathcal{L}$	labels of the edges indicating the lock type
$\mathcal{A}$	a scheduling algorithm
$l_{\mathcal{A}}(t)$	the latency of transaction t under $\mathcal{A}$
$\bar{l}_{\mathcal{A}}(t)$	the expectation of $l_{\mathcal{A}}(t)$
$\bar{l}(\mathcal{A})$	the expected transaction latency under $\mathcal{A}$

Table 4.1: Table of Notations.

4.2.4 NP-Hardness

Minimizing the expected transaction latency under the scheduling algorithm is, in general, an NP-hard problem. Intuitively, the hardness is due to the presence of shared locks, which cause the system's dependency graph to be a DAG, but not necessarily a tree.

Theorem 4.1. *Given a dependency graph $\mathcal{G}$, when a transaction t releases a lock (S or X) on object o, it is NP-hard to determine which pending lock requests to grant, in order to minimize the expected transaction latency. The result holds even if all transactions have the same execution time, and no transaction requests additional locks in the future.*[3]

Given the NP-hardness of the problem in general, in the rest of this dissertation, we propose algorithms that guarantee a constant-factor approximation of the optimal scheduling in terms of the expected transaction latency.

[3]All missing proofs can be found in our technical report [184].

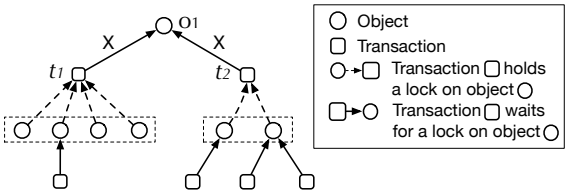

Figure 4.1: Transaction t_1 holds the greatest number of locks, but many of them on unpopular objects.

4.3 Contention-Aware Scheduling

We define contention-aware scheduling as any algorithm that prioritizes transactions based on their impact on the overall contention in the system. First, we study several heuristics for comparing the contribution of different transactions to the overall contention, and illustrate their shortcomings through intuitive examples. We then propose a particular contention-aware scheduling that formally quantifies this contribution, and guarantees a constant-factor approximation of the optimal scheduling when shared locks are not held by too many transactions. (Later, in Section 4.4, we generalize this algorithm for situations where this assumption does not hold.)

4.3.1 Capturing Contention

The degree of contention in a database system is directly related to the number of transactions concurrently requesting conflicting locks on the same objects.

our goal in contention-aware scheduling is to determine which transactions have a more important role in reducing the overall contention in the system, so that they can be given higher priority when granting a lock. Next, we discuss heuristics for measuring the priority of a transaction in reducing the overall contention.

Number of locks held— The simplest criterion for prioritizing transactions is the number of locks they currently hold. We refer to this heuristic as Most Locks First (MLF). The intuition is that a transaction with more locks is more likely to block other transactions in the system. However, this approach does not account for the popularity of objects in the system. In other words, a transaction might be holding many locks but on unpopular objects, which are unlikely to be requested by other transactions. Prioritizing such a transaction will not necessarily reduce contention in the system. Figure 4.1 demonstrates an example where transaction t_1 holds the most number of locks, but on unpopular objects. It is therefore better to keep t_1 waiting and instead schedule t_2 first, which holds fewer but more popular locks.

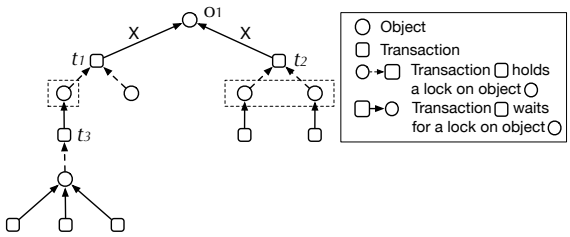

Figure 4.2: Transaction t_2 holds two locks that are waited on by other transactions. Although only one of t_1's locks is blocking other transactions, the blocked transaction (i.e., t_3) is itself blocking three others.

Number of locks that block other transactions— An improvement over the previous criterion is to only count those locks that have at least one transaction waiting on them. This approach disregards transactions that hold many locks, but on these locks no other transactions are waiting. We call this heuristic Most Blocking Locks First (MBLF). The issue with this criterion is that it treats all blocked transactions as the same, even if they contribute unequally to the overall contention. Figure 4.2 shows an example in which the scheduler must decide between transactions t_1 and t_2 when the object o_1 becomes available. Here, this criterion would choose t_2, which currently holds two locks, each at least blocking one other transaction. However, although t_1 holds only one blocking lock, it is blocking t_3 which itself is blocking three other transactions. Thus, by scheduling t_2 first, t_3 and its three subsequent transactions will remain blocked in the system for a longer period of time than if t_1 had been scheduled first.

Depth of the dependency subgraph— A more sophisticated criterion is the depth of a transaction's dependency subgraph. For a transaction t, this is defined as the subgraph of the dependency graph comprised of all vertices that can reach t (and all edges between such vertices). The depth of t's dependency subgraph is characterized by the number of transactions on the longest path in the subgraph that ends in t. We refer to this heuristic as Deepest Dependency First (DDF). Figure 4.3 shows an example, where the depth of the dependency subgraph of transaction t_1 is 3, while that of transaction t_2 is only 2. Thus, based on this criterion, the exclusive lock on object o_1 should be granted to t_1. The idea behind this heuristic is that a longer path indicates a larger number of transactions sequentially blocked. Thus, to unblock such transactions sooner, the scheduling algorithm must start with a transaction with deeper dependency graph. However, considering only the depth of this subgraph can limit the overall degree of concurrency in the system. For example, in Figure 4.3, if the exclusive lock on o_1 is granted to t_1 , upon its completion only one transaction in its dependency subgraph will be unblocked. On

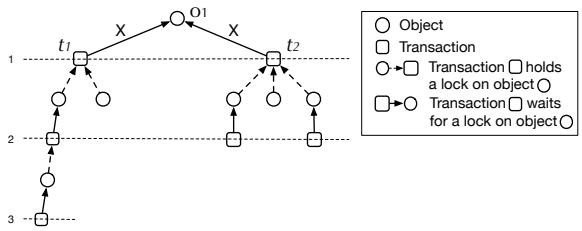

Figure 4.3: Transaction t_1 has a deeper dependency subgraph, but granting the lock to t_2 will unblock more transactions which can run concurrently.

the other hand, if the lock is granted to t_2, upon its completion two other transactions in its dependency subgraph will be unblocked, which can run concurrently.

Later, in Section 4.6.4, we empirically evaluate these heuristics. While none of these heuristics alone are able to guarantee an optimal lock scheduling strategy, they offer valuable insight in understanding the relationship between scheduling and overall contention. In particular, the first two heuristics focus on what we call *horizontal contention*, whereby a transaction holds locks on many objects directly needed by other transactions. In contrast, the third heuristic focuses on reducing *vertical contention*, whereby a chain of dependencies causes a series of transactions to block each other. Next, we present an algorithm which is capable of resolving both horizontal and vertical aspects of contention.

4.3.2 Largest-Dependency-Set-First

In this section, we propose an algorithm, called Largest-Dependency-Set-First (LDSF), which provides formal guarantees on the expected mean latency.

Consider two transactions t_1 and t_2 in the system. If there is a path from t_1 to t_2 in the dependency graph, we say that t_1 is dependent on t_2 (i.e., t_1 depends on t_2's completion/abortion for at least one of its required locks). We define the dependency set of t, denoted by $g(t)$, as the set of all transactions that are dependent on t (i.e., the set of transactions in t's dependency subgraph). Our LDSF algorithm uses the size of the dependency sets of different transactions to decide which one(s) to schedule first. For example, in Figure 4.4, there are five transactions in the dependency set of transaction t_1 (including t_1 itself) while there are four transactions in t_2's dependency set. Thus, in a situation where both t_1 and t_2 have requested an exclusive lock on object o_1, LDSF grants the lock to t_1 (instead of t_2) as soon as o_1 becomes available.

Now, we can formally present our LDSF algorithm. Suppose an object o becomes available (i.e., all previous locks on o are released), and there are $m + n$ transactions

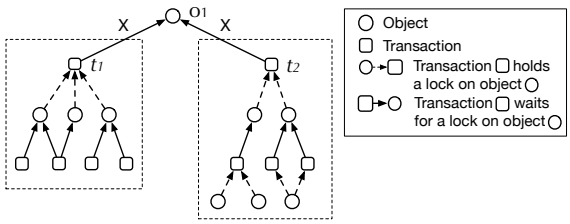

Figure 4.4: Lock scheduling based on the size of the dependency sets.

Algorithm 5: *Largest-Dependency-Set-First Algorithm*

Input : The dependency graph of the system $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$,
transaction t, object o, label $L \in \{X, S\}$
// meaning t has just released a lock of type L on o

Output: The set of transactions whose requested lock on o should be granted

if *there are other transactions still holding a lock on* o **then**
  return $\emptyset$;
end
Obtain the set of transactions waiting for a shared lock on o,
  $T^i \leftarrow \{t^i \in \mathcal{V} : (t^i, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^i, o) = S\} = \{t_1^i, t_2^i, \cdots, t_m^i\}$;
Obtain the set of transactions waiting for an exclusive lock on o,
  $T^x \leftarrow \{t^x \in \mathcal{V} : (t^x, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^x, o) = X\} = \{t_1^x, t_2^x, \cdots, t_n^x\}$;
Let $\tau(T^i) = \left|\bigcup_{i=1}^{n} g(t_i^i)\right|$;
Find a transaction $t^{\hat{x}} \in W$ s.t. $|g(t^{\hat{x}})| = \max_{t_i^x \in T^x} \left|g(t_i^x)\right|$;
if $\tau(T^i) < \left|g(t^{\hat{x}})\right|$ **then**
  return T^i;
else
  return $\{t^{\hat{x}}\}$;
end

currently waiting for a lock on o: m transactions $t_1^i, t_2^i, \cdots, t_m^i$ are requesting a shared lock o, and n transactions $t_1^x, t_2^x, \cdots, t_n^x$ are requesting an exclusive lock on object o. Our LDSF algorithm defines the priority of each transaction t_i^x requesting an exclusive lock as the size of its dependency set, $\left|g(t_i^x)\right|$. However, LDSF treats all transactions requesting a shared lock on o, namely $t_1^i, t_2^i, \cdots, t_m^i$, as a single transaction—if LDSF decides to grant a shared lock, it will be granted to all of them. The priority of the shared lock requests is thus defined as the size of the union of their dependency sets, $\left|\bigcup_{i=1}^{m} g(t_i^i)\right|$. LDSF then finds the transaction $t^{\hat{x}}$ with the highest priority among $t_1^x, t_2^x, \cdots, t_n^x$. If $t^{\hat{x}}$'s priority is higher than the collective priority of the transactions requesting a shared lock, LDSF grants the exclusive lock to $t^{\hat{x}}$. Otherwise, a shared lock is granted to all transactions $t_1^i, t_2^i, \cdots, t_m^i$. The pseudo-code of the LDSF algorithm is provided in Algorithm 5.

Analysis— We do not make any assumptions about the future behavior of a transaction, as they may request various locks throughout their lifetime. Furthermore, since we cannot predict new transactions arriving in the future, in our analysis, we only consider the transactions that are already in the system. Since the system does not know the execution time of a transaction *a priori*, we model the execution time of each transaction as a memoryless random variable. That is, the time a transaction has already spent in execution does not necessarily reveal any information about the transaction's remaining execution time. We denote the remaining execution time as a random variable R with expectation $\bar{R}$. We also assume that the execution time of a transaction is not affected by the scheduling.[4] Transactions whose behavior depends on the actual wall-clock time (e.g., stop if run before 2pm, otherwise run for a long time) are also excluded from our discussion.

We first study a simplified scenario in which there are only exclusive locks in the system (we relax this assumption in Theorem 4.3). The following theorem states that LDSF minimizes the expected latency in this scenario.

Theorem 4.2. *When there are only exclusive locks in the system, the LDSF algorithm is the optimal scheduling algorithm in terms of the expected latency.*

The intuition here is that if a transaction t_1 is dependent on t_2, any progress in the execution of t_2 can also be considered as t_1's progress since t_1 cannot receive its lock unless t_2 finishes execution. Thus, by granting the lock to the transaction with the largest dependency set, LDSF allows the most transactions to make progress toward completion.

However, this does not necessarily hold true with the existence of shared locks. Even if transaction t_1 is dependent on t_2, the execution of t_2 does not necessarily contribute to t_1's progress. Specifically, consider the set of all objects that are reachable from t_1 in the dependency graph, but are locked (shared or exclusively) by *currently running* transactions. We call these objects the *critical objects* of t_1, and denote them as $C(t_1)$.[5] For example, in Figure 4.5, we have $C(t_1) = \{o_1, o_2\}$. Note that not all transactions that hold a lock on a critical object of t_1 contribute to t_1's progress. Rather, only the transaction that releases the last lock on that critical object allows for the progress of t_1. In the example of Figure 4.5, t_2's execution does not contribute to t_1's progress, unless t_3 releases the lock before t_2.

Nonetheless, when the number of transactions waiting for each shared lock is bounded, LDSF is a constant-factor approximation of the optimal scheduler.

[4]For example, scheduling causes context switches, which may affect performance. For simplicity, in our formal analysis, we assume that their overall effect is not significant.

[5]Note that the critical objects of a transaction may change throughout its lifetime.

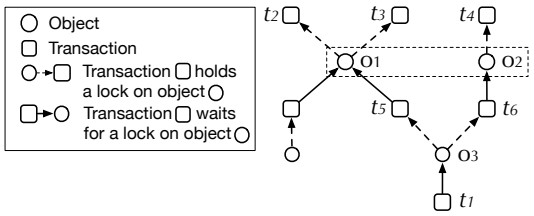

Figure 4.5: The critical objects of t_1 are o_1 and o_2, as they are locked by transactions t_2 and t_3. Note that, although o_3 is reachable from t_1, it is not a critical object of t_1 since it is locked by transactions that are not currently running, i.e., t_5 and t_6 which themselves are waiting for other locks.

Theorem 4.3. *Let the maximum number of critical objects for any transaction in the system be c. Assume that the number of transactions waiting for a shared lock on the same object is bounded by u. The LDSF algorithm is a $(c \cdot u)$-approximation of the optimal scheduling (among strategies that grant all shared locks simultaneously) in terms of the expected latency.*

4.4 Splitting Shared Locks

In the LDSF algorithm, when a shared lock is granted, it is granted to all transactions waiting for it. In Section 4.4.1, we show why this may not be the best strategy. Then, in Section 4.4.2, we propose a modification to our LDSF algorithm, called bLDSF, which improves upon LDSF by exploiting the idea of not granting all shared locks simultaneously.

4.4.1 The Benefits and Challenges

As noted earlier, when the LDSF algorithm grants a shared lock, it grants the lock to all transactions waiting for it. However, this may not be the optimal strategy. In general, granting a larger number of shared locks on the same object increases the probability that at least one of them will take a long time before releasing the lock. Until the last transaction completes and releases its lock, no exclusive locks can be granted on that object. In other words, the expected duration that the slowest transaction holds a shared lock grows with the number of transactions sharing the lock. This is the well-known problem of *stragglers* [65,78,85,155,206], which is exacerbated as the number of independent processes grows.

To illustrate this more formally, consider the following example. Suppose that a set of m transactions, $t_1, \cdots, t_m$, are sharing a shared lock. Let $R_1^{rem}, R_2^{rem}, \cdots, R_m^{rem}$ be a set

of random variables representing the remaining times of these transactions. Then, the time needed before an exclusive lock can be granted on the same object is the remaining time of the the slowest transaction, denoted as $R^{rem}_{\max,m} = \max\{R^{rem}_1, \cdots, R^{rem}_m\}$, which itself is a random variable. Let $\bar{R}^{rem}_{\max,m}$ be the expectation of $R^{rem}_{\max,m}$. As long as the R^{rem}_i's have non-zero variance[6] (i.e., $\sigma^2_i > 0$), $\bar{R}^{rem}_{\max,m}$ strictly increases with m, as stated next.

Lemma 4.4. *Suppose that $R^{rem}_1, R^{rem}_2, \cdots$ are random variables with the same range of values. If $\sigma^2_{k+1} > 0$, then $\bar{R}^{rem}_{\max,k} < \bar{R}^{rem}_{\max,k+1}$ for $1 \leq k < m$.*

We define the delay factor as $f(m) = \dfrac{\bar{R}^{rem}_{\max,m}}{\bar{R}^{rem}}$. According to Lemma 4.4, $f(m)$ is strictly monotonically increasing with respect to m. The exact formula for $f(m)$ will depend on the specific distribution of R_i's. For example, if R_i's are exponentially distributed (i.e., a memoryless distribution) with mean $\bar{R}$, then their CDF is given by $F(x) = 1 - e^{-x/\bar{R}^{rem}}$. Then, $f(m)$ can be computed as $f(m) = \sum_{i=1}^{m} \dfrac{1}{i}$ However, regardless of the distribution of the latencies, $f(m)$ is guaranteed to satisfy the following three properties:

C1. $f(1) = 1$;

C2. $f(m) < f(m+1)$;

C3. $f(m) \leq m$.

The first property is trivial: granting the lock to only one transaction at a time does not incur any delays. The second property is based on Lemma 4.4. The third is based on the fact that sharing a lock between a group of m transactions cannot be slower than granting the lock to them one after another and sequentially.

Since granting a shared lock to more transactions can delay the exclusive lock requests, it is conceivable that granting a shared lock to only a subset of the transactions waiting for it might reduce the overall latency in the system. Intuitively, when many transactions are waiting for the same shared lock, it would be better to grant the shared lock only to a few that have a higher priority (i.e., a larger dependency set), and leave the rest until the next time. This strategy can therefore reduce the time that other transactions have to wait for an exclusive lock, as illustrated in Figure 4.6. However, lock scheduling in this situation becomes extremely difficult. We have the following negative result.

[6]This assumption holds unless all instances of a transaction type take exactly the same time, which is unlikely.

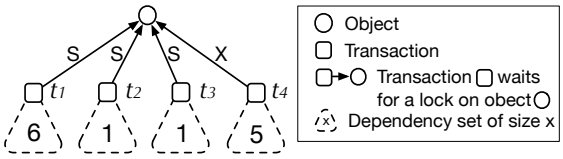

Figure 4.6: Assume that $f(2) = 1.5$ and $f(3) = 2$. If we first grant a shared lock to all of t_1, t_2, and t_3, all transactions in t_4's dependency set will wait for at least $2\bar{R}$. The total wait time will be $10\bar{R}$. However, if we only grant t_1's lock, then t_4's lock, and then grant t_2's and t_3's locks together, the transactions in t_4's dependency set will only wait $\bar{R}$, while those in t_2's and t_3's dependency sets will wait $2\bar{R}$. Thus, the total wait time in this case will be only $9\bar{R}$.

Theorem 4.5. *Let $\mathbb{A}_{\neg f}$ be the set of scheduling algorithms that do not use the knowledge of the delay factor $f(k)$ in their decisions. For any algorithm $\mathcal{A}_{\neg f} \in \mathbb{A}_{\neg f}$, there exists an algorithm $\mathcal{A}$, such that $\dfrac{\bar{w}(\mathcal{A}_{\neg f})}{\bar{w}(\mathcal{A})} = \omega(1)$ for some delay factor $f(k)$.*

According to this theorem, any algorithm that does not rely on knowing the delay factor is not competitive: it performs arbitrarily poor, compared to the optimal scheduling. Thus, in the next section, we take the delay factor $f(k)$ as an input, and propose an algorithm that adopts the idea of granting shared locks only to a subset of the transactions requesting it. We also discuss the criteria for choosing delay factors that can yield good performance in practice.

4.4.2 The bLDSF Algorithm

In this section, we present a simple algorithm, called bLDSF, which inherits the intuition behind the LDSF algorithm, but also exploits the idea that a shared lock does not have to be granted to all transactions waiting for it.

While LDSF measures the progress enabled by different scheduling decisions, our bLDSF algorithm measures the *speed of progress*. If a transaction t^x waiting for an exclusive lock is scheduled, $|g(t^x)|$ transactions will make progress over the next $\bar{R}$ (expected) units of time. Thus, the speed of progress can be measured as $\dfrac{|g(t^x)|}{\bar{R}}$. On the other hand, by scheduling a batch of transactions $t_1^i, t_2^i, \cdots, t_k^i$ waiting for a shared lock together, $|\bigcup_{i=1}^{k} g(t_i^i)|$ transactions will make progress over the next $f(k) \cdot \bar{R}$ units of time. The speed of progress can then be measured as $\dfrac{|\bigcup_{i=1}^{k} g(t_i^i)|}{f(k)\bar{R}}$.

The bLDSF algorithm works as follows. First, it finds the transaction waiting for an exclusive lock with the largest dependency set, denoted as $\hat{t^x}$. Denote the size of its

Algorithm 6: *The bLDSF Algorithm*

Input : The dependency graph of the system $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$,
transaction t, object o, label $L \in \{X, S\}$
// meaning t has just released a lock of type L on o

Output: The set of transactions whose requested lock on o should be granted

if *there are other transactions still holding a lock on* o **then**
  return $\emptyset$;
end
Obtain the set of transactions waiting for a shared lock on o,
  $T^i \leftarrow \{t^i \in \mathcal{V} : (t^i, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^i, o) = S\} = \{t^i_1, t^i_2, \cdots, t^i_m\}$;
Obtain the set of transactions waiting for an exclusive lock on o,
  $T^x \leftarrow \{t^x \in \mathcal{V} : (t^x, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^x, o) = X\} = \{t^x_1, t^x_2, \cdots, t^x_n\}$;
Let $\hat{t^i_1}, \hat{t^i_2}, \cdots, \hat{t^i_k}$ be the set of transactions in T^i such that $\frac{|\bigcup_{i=1}^{k} g(\hat{t^i_i})|}{f(k)}$ is maximized ;
Let $\hat{t^x}$ be the transaction in T^x with the largest dependency set;
if $\left|g(\hat{t^x})\right| \cdot f(k) \leq \left|\bigcup_{i=1}^{k} g(\hat{t^i_i})\right|$ **then**
  return $\{\hat{t^i_1}, \hat{t^i_2}, \cdots, \hat{t^i_k}\}$;
else
  return $\hat{t^x}$;
end

dependency set as $p = \left|g(\hat{t^x})\right|$. Then, bLDSF finds the batch of transactions, $\hat{t^i_1}, \hat{t^i_2}, \cdots, \hat{t^i_k}$, waiting for a shared lock such that $q = \frac{|\bigcup_{i=1}^{k} g(\hat{t^i_i})|}{f(k)}$ is maximized. When $q < p$, the system will make faster progress if $\hat{t^x}$ is scheduled first, in which case bLDSF will grant an exclusive lock to $\hat{t^x}$. Conversely, when $q > p$, the system will make faster progress if the batch of $\hat{t^i_1}, \hat{t^i_2}, \cdots, \hat{t^i_k}$ is scheduled first, in which case bLDSF will grant shared locks to $\hat{t^i_1}, \hat{t^i_2}, \cdots, \hat{t^i_k}$ simultaneously. When $q = p$, the speed of progress in the system will be the same under both scheduling decisions. In this case, bLDSF grants shared locks to the batch, in order to increase the overall degree of concurrency in the system. The pseudocode for bLDSF is provided in Algorithm 6.

We show that, when the number of transactions waiting for shared locks on the same object is bounded, the bLDSF algorithm is a constant factor approximation of the optimal scheduling algorithm in terms of the expected wait time.

Theorem 4.6. *Let the maximum number of critical objects for any transaction in the system be c. Assume that the number of transactions waiting for shared locks on the same object is bounded by v. Then, given a delay factor of $f(k)$, the bLDSF algorithm is an h-approximation of the optimal scheduling algorithm in terms of the expected wait time, where $h = cv^2 \cdot f(v)$.*

Unlike the LDSF algorithm, bLDSF requires a delay factor for its analysis. However, since the remaining times of transactions can be modeled as random variables, the exact form of the delay factor $f(k)$ will also depend on the distribution of these random variables. For example, the delay factor for exponential random variables is $f(k) = O(\log k)$ [67], for geometric random variables is $f(k) = O(\log k)$ [81], for Gaussian random variables is $f(k) = O(\sqrt{\log k})$ [115], and for power law random variables with exponent 3 is $f(k) = \sqrt{k}$. In Section 4.6.7, we empirically show that bLDSF's performance is not sensitive to the specific choice of the delay factor, as long as it is a sub-linear function that grows monotonically with k (conditions C1, C2, and C3 from Section 4.4.1). This is because, when the batch size is small, the difference between all sub-linear functions is also small. For example, when $b = 10$, $\sqrt{b} \approx 3.16$ and $\log_2(1+b) \approx 3.46$, leading to similar scheduling decisions. Even though $\sqrt{\log_2(1+b)} \approx 1.86$ is smaller than the other two, it can still capture condition C2 quite well.

4.4.3 Discussion

In our analysis, we have assumed no additional information regarding a transaction's remaining execution time, or its lock access pattern. However, with the recent progress on incorporating machine learning models into DBMS technology [35,142], one might be able to predict transaction latencies [143,202] in the near future. When such information is available, a lock scheduling algorithm could take that into account when maximizing the *speed of progress*: a transaction that will take longer should be given less priority. The priority of a transaction would then be the size of its dependency set divided by its estimated execution time. Likewise, a transaction performing a table scan will request a large number of locks, and will not make any progress until all of its locks can be granted. Thus, knowing a transaction's lock pattern in advance would also be beneficial. We leave such extensions of our algorithms (e.g., to hybrid workloads [146]) to future work.

4.5 Implementation

We have implemented our scheduling algorithm in MySQL. Similar to all major DBMSs, the default lock scheduling policy in MySQL was FIFO.[7] Specifically, all pending lock requests on an object are placed in a queue. A lock request is granted immediately upon its arrival only if one of these two conditions holds: (i) there are no other locks currently

[7]Now, our LDSF algorithm is the default (MySQL 8.0.3+).

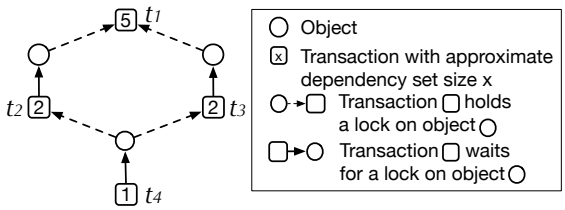

Figure 4.7: The effective size of t_1's dependency set is 5. But its exact size is only 4.

held on the object, or (ii) the requested lock type is compatible with all of the locks currently held on the object and there are no incompatible requests ahead of it waiting in the queue. Similarly, whenever a lock is released on an object, MySQL's scheduler scans the entire queue from beginning to the end. It grants any waiting requests as long as one of these conditions holds. As soon as the scheduler encounters the first lock request that cannot be granted, it stops scanning the rest of the queue.

One challenge in implementing LDSF and bLDSF is keeping track of the sizes of the dependency sets. Exact calculation would require either (i) searching down the reverse edges in the dependency graph in real-time, whenever a scheduling decision is to be made, or (ii) storing the dependency sets for all transactions and maintaining them each time a transaction is blocked or a lock is granted. Both options are relatively costly. Therefore, in our implementation, we rely on an approximation of the sizes of the dependency sets, rather than computing their exact values. When a transaction t holds no locks that block other transactions, $|g(t)| = 1$. Otherwise, let T_t be the set of transactions waiting for an object currently held by transaction t. Then, $|g(t)| \approx \sum_{t' \in T_t} |g(t')| + 1$. The reason this method is only an approximation of $|g(t)|$ is that the dependency graph is a DAG (but not necessarily a tree), which means the dependency sets of different transactions may overlap. Figure 4.7 illustrates an example, where the dependency set of t_1 is $\{t_1, t_2, t_3, t_4\}$ and is therefore of size 4. However, its effective size is calculated as one plus the sum of the effective sizes of t_2 and t_3's dependency sets, resulting in 5. To ensure that transactions appearing on multiple paths will not be updated multiple times, we also keep track of those that have already been updated.

Another implementation challenge lies in the difficulty of finding the desired batch of transactions in bLDSF. Calculating the size of the union of several dependency sets requires detailed information about the elements in each dependency set (since the dependency sets may overlap due to shared locks). Therefore, we rely on the following approximation in our implementation. We first sort all transactions waiting for a shared lock in the decreasing order of their dependency set sizes. Then, for $k = 1, 2, \cdots$, we calculate the q value (see Section 4.4.2) for the first k transactions. Here, we approximate

the size of the union of the dependency sets as the sum of their individual sizes. Let k^* be the k value that maximizes q. We then take the first k^* transactions as our batch, which we consider for granting a shared lock to.

In Section 4.6, we show that, despite using these approximations in our implementation, our algorithms remain quite effective in practice.

Starvation Avoidance— In MySQL's implementation of FIFO, when there is an exclusive lock request in the queue, it serves as a conceptual barrier: later requests for shared locks cannot be granted, even if they are compatible with the currently held locks on the object. This mechanism prevents starvation when using FIFO. In our algorithms, starvation is prevented using a similar mechanism. We place a barrier at the end of the current wait queue. Lock requests that arrive later are placed behind this barrier and are not considered for scheduling. In other words, the only requests that are considered are those that are ahead of the barrier. Once all such requests are granted, this barrier is lifted, and a new barrier is added to the end of the current queue, i.e., those requests that were previously behind a barrier are now ahead of one. This mechanism prevents a transaction with a small dependency set from waiting indefinitely behind an infinite stream of newly arrived transactions with larger dependency sets. An alternative strategy to avoid starvation is to simply add a fraction of the transaction's age to its dependency set size when making scheduling decisions. A third strategy is to replace a transaction's dependency set size with a sufficiently large number once its wait time has exceeded a certain timeout threshold.

Space Complexity— Given the approximation methods mentioned above, both LDSF and bLDSF only require maintaining the approximate size of the dependency set of each transaction. Therefore, the overall space overhead of our algorithms is only $O(|T|)$.

Time Complexity— In MySQL, all lock requests on an object (either granted or not) are stored in a linked list. Whenever a transaction releases a lock on the object, the scheduler scans this list for requests that are not granted yet. For each of these requests, the scheduler scans the list again to check compatibility with granted requests. If the request is found compatible with all existing locks, it is granted, and the scheduler checks the compatibility of the next request. Otherwise, the request is not granted, and the scheduler stops granting further locks. Let N be the number of lock requests on an object (either granted or not). Then, FIFO takes $O(N^2)$ time in the worst case. LDSF and bLDSF both use the same procedure as FIFO to find compatible requests that are not granted yet, which takes $O(N^2)$ time. For bLDSF, we also sort all transactions waiting for

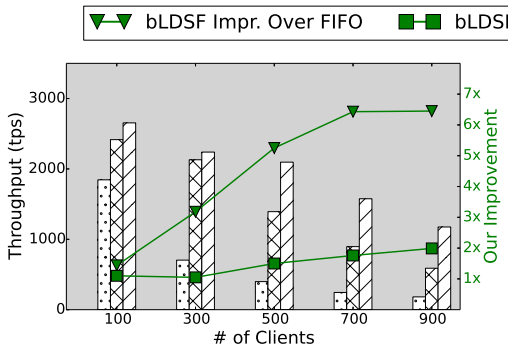

Figure 4.8: Throughput improvement with bLDSF (TPC-C).

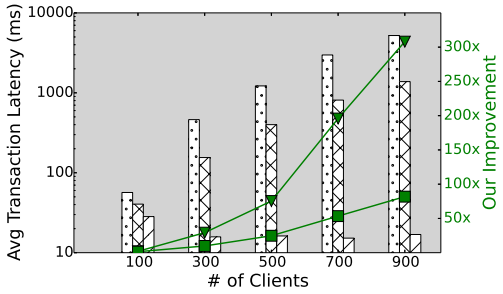

Figure 4.9: Avg. latency improvement with bLDSF (under the same TPC-C transactions per second).

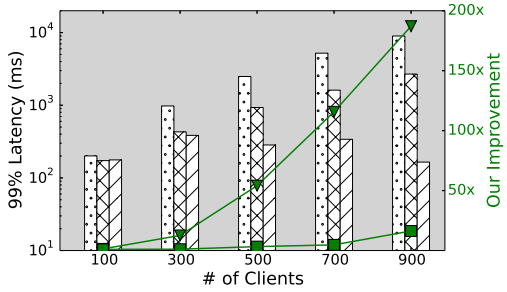

Figure 4.10: Tail latency improvement w/ bLDSF (under the same number of TPC-C transactions per second).

a shared lock by the size of their dependency sets, which takes $O(N \log N)$ time. Thus, the time complexity of LDSF and bLDSF is still $O(N^2)$.

4.6 Experiments

Our experiments aim to answer several key questions:

- How do our scheduling algorithms (LDSF and bLDSF) affect the overall throughput of the system?
- How do our algorithms compare against FIFO (the default policy in nearly all databases) and VATS (recently adopted by MySQL), in terms of reducing average and tail transaction latencies?
- How do our scheduling algorithms compare against various heuristics?

- How much overhead do our algorithms incur, compared to the latency of a transaction?
- How does the effectiveness of our algorithms vary with different levels of contention?
- What is the impact of the choice of delay factor on the effectiveness of bLDSF?
- What is the impact of approximating the dependency sets (Section 4.5) on reducing the overhead?

In summary, our experiments show the following:

1. By resolving contention much more effectively than FIFO and VATS, bLDSF improves throughput by up to 6.5x (by 4.5x on average) over FIFO, and by up to 2x (1.5x on average) over VATS. (Section 4.6.2)
2. bLDSF can reduce mean transaction latencies by up to 300x and 80x (30x and 3.5x, on average) compared to FIFO and VATS, respectively. It also reduces the 99th percentile latency by up to 190x and 16x, compared to FIFO and VATS, respectively. (Section 4.6.3)
3. Both bLDSF and LDSF outperform various heuristics by 2.5x in terms of throughput, and by up to 100x (8x on avg.) in terms of transaction latency. (Section 4.6.4)
4. Our algorithms reduce queue length by reducing contention, and thus incur much less overhead than FIFO. However, their overhead is larger than VATS. (Section 4.6.5)
5. As the degree of contention rises in the system, bLDSF's improvement over both FIFO and VATS increases. (Section 4.6.6)
6. bLDSF is not sensitive to the specific choice of delay factor, as long as it is chosen to be an increasing and sub-linear function. (Section 4.6.7)
7. Our approximation technique reduces scheduling overhead by up to 80x.

4.6.1 Experimental Setup

Hardware & Software— All experiments were performed using a 5 GB buffer pool on a Linux server with 16 Intel(R) Xeon(R) CPU E5-2450 processors and 2.10GHz cores. The clients were run on a separate machine, submitting transactions to MySQL 5.7 running on the server.

Methodology— We used the OLTP-Bench tool [74] to run the TPC-C workload. We also modified this tool to run a microbenchmark (explained below). OLTP-Bench generated transactions at a specified rate, and client threads issued these transactions to MySQL.

The latency of each transaction was calculated as the time from when it was issued until it finished. In all experiments, we controlled the number of transactions issued per second within a safe range to prevent MySQL from falling into a thrashing regime. We also noticed that the number of deadlocks was negligible compared to the total number of transactions, across all experiments and algorithms.

TPC-C Workload— We used a 32-warehouse configuration for the TPC-C benchmark. To simulate a system with different levels of contention, we relied on changing the following two parameters: (i) number of clients, and (ii) number of submitted transactions per second (a.k.a. throughput). Each of our client threads issued a new transaction as soon as its previous transaction finished. Thus, by creating a specified number of client threads, we effectively controlled the number of in-flight transactions. To control the system throughput, we created client threads that issued transactions at a specific rate.

Microbenchmark— We created a microbenchmark for a more thorough evaluation of our algorithm under different degrees of contention. Specifically, we created a database with only one table that had 20,000 records in it. The clients would send transactions to the server, each comprised of 5 queries. Each query was randomly chosen to be either a "SELECT" query (acquiring a shared lock) or an "UPDATE" query (acquiring an exclusive lock). The records in the table were accessed by the queries according to a Zipfian distribution. To generate different levels of contention, we varied the following two parameters in our microbenchmark:

1. skew of the access pattern (the parameter θ of the Zipfian distribution)
2. fraction of exclusive locks (probability of "UPDATE" queries).

Baselines— We compared the performance of our bLDSF algorithm (with $f(k)$=$\log_2(1+k)$ as default) against the following baselines:

1. **First In First Out (FIFO).** FIFO is the default scheduler in MySQL and nearly all other DBMSs. When an object becomes available, FIFO grants the lock to the transaction that has waited the longest.
2. **Variance-Aware Transaction Scheduling (VATS).** This is the strategy proposed by Huang et al. [110]. When an object becomes available, VATS grants the lock to the eldest transaction in the queue.
3. **Largest Dependency Set First (LDSF).** This is the strategy described in Algorithm 5, which is equivalent to bLDSF with $b = \text{inf}$, and $f(k) = 1$.
4. **Most Locks First (MLF).** When an object becomes available, grant a lock on it to the transaction that holds the most locks (introduced in Section 4.3.1).

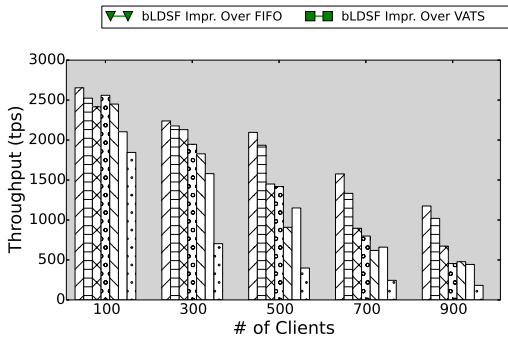

Figure 4.11: Maximum throughput under various algorithms (TPC-C).

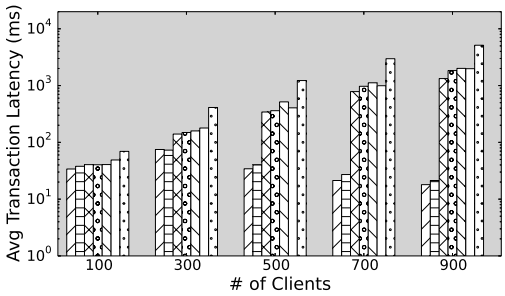

Figure 4.12: Transaction latency under various algorithms (TPC-C).

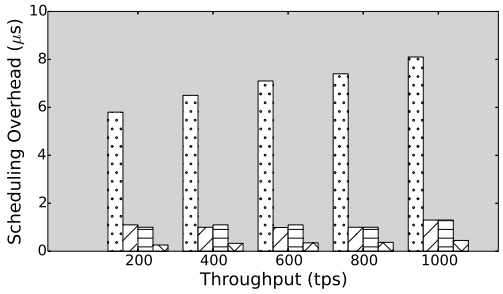

Figure 4.13: Scheduling overhead of various algorithms (TPC-C).

5. **Most Blocking Locks First (MBLF).** When an object becomes available, grant a lock on it to the transaction that holds the most locks which block at least one other transaction (introduced in Section 4.3.1).
6. **Deepest Dependency First (DDF).** When an object becomes available, grant a lock on it to the transaction with the deepest dependency subgraph (Section 4.3.1).

For MLF, MBLF, and DDF, if a shared lock is granted, all shared locks on that object are granted. For LDSF and bLDSF, we use the barriers explained in Section 4.5 to prevent starvation. For FIFO and VATS, if a shared lock is granted, they continue to grant shared locks to other transactions waiting in the queue until they encounter an exclusive lock, at which point they stop granting more locks.

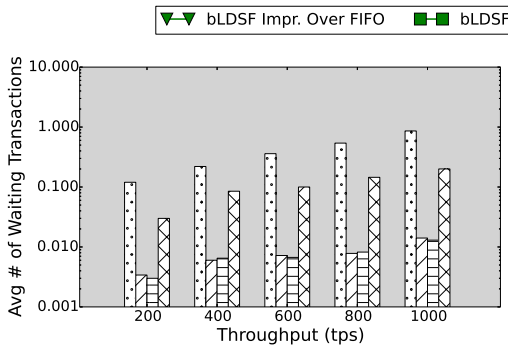

Figure 4.14: Average number of transactions waiting in the queue under various algorithms (TPC-C).

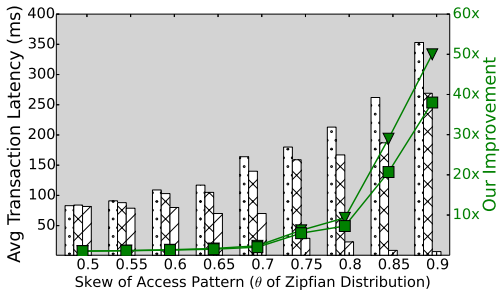

Figure 4.15: Average transaction latency for different degrees of skewness (microbenchmark).

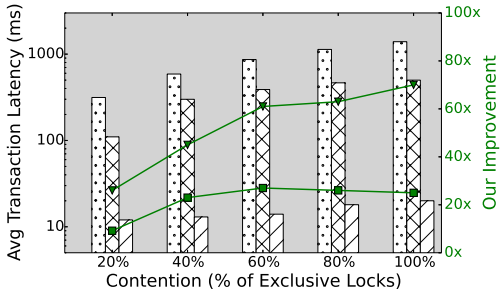

Figure 4.16: Average latency for different numbers of exclusive locks (microbenchmark).

4.6.2 Throughput

We compared the system throughput when using FIFO and VATS versus bLDSF, given an equal number of clients (i.e., in-flight transactions). We varied the number of clients from 100 to 900. The results of this experiment for TPC-C are presented in Figure 4.8.

In both cases, the throughput dropped as the number of clients increased. This is expected, as more transactions in the system lead to more objects being locked. Thus, when a transaction requests a lock, it is more likely to be blocked. In other words, the number of transactions that can make progress decreases, which leads to a decrease in throughput.

However, the throughput decreased more rapidly when using FIFO or VATS than bLDSF. For example, when there were only 100 clients, bLDSF outperformed FIFO by only 1.4x and VATS by 1.1x. However, with 900 clients, bLDSF achieved 6.5x higher throughput than FIFO and 2x higher throughput than VATS. As discussed in Section 4.4.2,

bLDSF always schedules transactions that maximize the speed of progress in the system. This is why it allows for more transactions to be processed in a certain amount of time.

4.6.3 Average and Tail Transaction Latency

We compared transaction latencies of FIFO, VATS, and bLDSF under an equal number of transactions per second (i.e, throughput). We varied the number of clients (and hence, the number of in-flight transactions) from 100 to 900 for FIFO and VATS, and then ran bLDSF at the same throughput as VATS, which is higher than the throughput of FIFO. This means that we compare bLDSF with FIFO at a higher throughput. The result is shown in Figure 4.9. Our bLDSF algorithm dramatically outperformed FIFO by a factor of up to 300x and VATS by 80x. This outstanding improvement confirms our Theorems 4.3 and 4.6, as our algorithm is designed to minimize average transaction latencies.

We also report the 99th percentile latencies in Figure 4.10. Here, bLDSF outperformed FIFO by up to 190x. Interestingly, bLDSF outperformed VATS too (by up to 16x), even though the latter is specifically designed to reduce tail latencies. This is because bLDSF causes all transactions to finish faster on average, and thus, those transactions waiting at the end of the queue will also wait less, leading to lower tail latencies.

4.6.4 Comparison with Other Heuristics

In this section, we report our comparison of both bLDSF and LDSF algorithms against the heuristic methods introduced in Section 4.3, i.e., MLF, MBLF, and DDF. Moreover, we compare our algorithms with VATS too.

First, we compared their throughput given an equal number of clients. We varied the number of clients from 100 to 900. The results are shown in Figure 4.11. LDSF and bLDSF achieve up to 2x and 2.5x improvement over the other heuristics in terms of throughput, respectively.

We also measured transaction latencies under an equal number of transactions per second (i.e, throughput). We varied the number of clients from 100 to 900 for the heuristics, and then ran bLDSF and LDSF at the maximum throughput achieved by any of the heuristics. For those heuristics which were not able to achieve this throughput, we compared our algorithms at a higher throughput than they achieved. The results are shown in Figure 4.12, indicating that MLF, MBLF, and DDF outperformed FIFO by almost 2.5x in terms of average latency, while our algorithms achieved up to 100x improvement over the best heuristics (MBLF with 900 transactions). Furthermore, bLDSF was better than LDSF by a small margin.

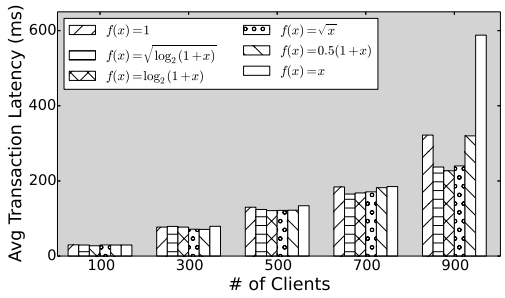

Figure 4.17: The impact of delay factor on average latency.

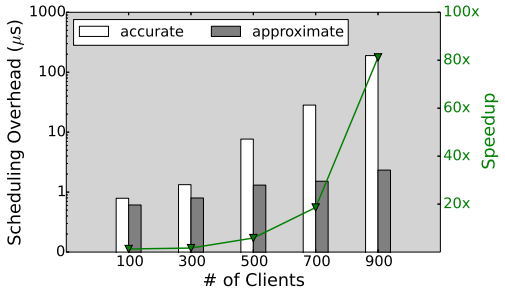

Figure 4.18: Scheduling overhead with and without our approximation heuristic for choosing a batch.

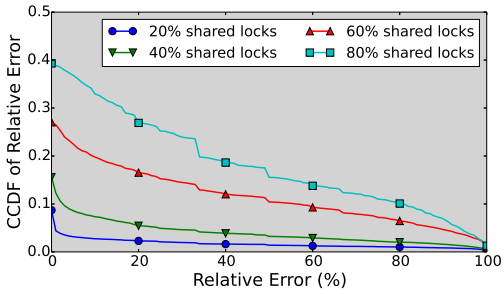

Figure 4.19: CCDF of the relative error of the approximation of the sizes of the dependency sets.

4.6.5 Scheduling Overhead

We also compared the overhead of our algorithms (LDSF and bLDSF) against both FIFO and VATS: the overhead of a scheduling algorithm is the time needed by the algorithm to *decide* which lock(s) to grant.

In this experiment, we fixed the number of clients to 100 while varying throughput from 200 to 1000. The result is shown in Figure 4.13. We can see that, although all three algorithms have the same time complexity in terms of the queue length (Section 4.5), ours resulted in much less overhead than FIFO because they led to much shorter queues for the same throughput. This is because our algorithms effectively resolve contention, and thus, reduce the number of waiting transactions in the queue. To illustrate this, we also measured the average number of waiting transactions whenever an object becomes available. As shown in Figure 4.14, this number was much smaller for LDSF and bLDSF. However, VATS incurred less overhead than LDSF and bLDSF, despite having longer queues. This is because VATS does not compute the sizes of the dependency sets.

4.6.6 Studying Different Levels of Contention

In this section, we study the impact of different levels of contention on the effectiveness of our bLDSF algorithm. Contention in a workload is a result of two factors: (i) skew in the data access pattern (e.g., popular tuples), and (ii) a large number of exclusive locks. There is more contention when the pattern is more skewed, as transactions will request a lock on the same records more often. Likewise, exclusive lock requests cause more contention, as they cannot be granted together and result in blocking more transactions. We studied the effectiveness of our algorithm under different degrees of contention by varying these two factors using our microbenchmark:

1. We fixed the fraction of exclusive locks to be 60% of all lock requests, and varied the θ parameter of the Zipfian distribution of our access distribution between 0.5 and 0.9 (larger θ, more skew).
2. We fixed the θ parameter to be 0.8 and varied the probability of an "UPDATE" query in our microbenchmark between 20% and 100%. The larger this probability, the larger the fraction of exclusive locks.

First, we ran FIFO using 300 clients, and then ran both VATS and bLDSF at the same throughput as FIFO. The results of these experiments are shown in Figures 4.15 and 4.16.

Figure 4.15 shows that when there is no skew, there is no contention, and thus most queues are either empty or only have a single transaction waiting. Since there is no scheduling decision to be made in this situation, FIFO, VATS and bLDSF become equivalent and exhibit a similar performance. However, the gap between bLDSF and the other two algorithms widens as skew (and thereby contention) increases. For example, when the data access is highly skewed ($\theta = 0.9$), bLDSF outperforms FIFO by more than 50x and VATS by 38x. Figure 4.16 reveals a similar trend: as more exclusive locks are requested, bLDSF achieves greater improvement. Specifically, when 20% of the lock requests are exclusive, bLDSF outperforms FIFO by 20x and VATS by 9x. However, when all the locks are exclusive, the improvement is even more dramatic, i.e., 70x over FIFO and 25x over VATS. Note that, although VATS guarantees optimality when there are only exclusive locks [110], it fails to account for transaction dependencies in its analysis (see Section 4.8 for a discussion of the assumptions made in VATS versus bLDSF). In summary, when there is no contention in the system, there are no scheduling decisions to be made, and all scheduling algorithms are equivalent. However, as contention rises, so does the need for better scheduling decisions, and so does the gap between bLDSF and other algorithms.

4.6.7 Choice of Delay Factor

To better understand the impact of delay factors on bLDSF, we experimented with several functions of different growth rates, ranging from the lower bound of all functions that satisfy conditions C1, C2, and C3 (i.e., $f(k) = 1$) to their upper bound (i.e., $f(k) = k$). Specifically, we used each of the following delay factors in our bLDSF algorithm, and measured the average transaction latency:

- $f_1(k) = 1$;
- $f_2(k) = \sqrt{\log_2(1+k)}$;
- $f_3(k) = \log_2(1+k)$;
- $f_4(k) = \sqrt{k}$;
- $f_5(k) = 0.5(1+k)$;
- $f_6(k) = k$.

The results are shown in Figure 4.17. We can see that all sub-linear functions (i.e., f_2, f_3, and f_4) performed comparably, and that they performed better than the other functions. Understandably, f_1 did not perform well, as it did not satisfy condition C2 from Section 4.4.1. Functions f_5 and f_6 did not perform well either, since linear functions overestimate the delay. For example, two transactions running concurrently take less time than if they ran one after another.

4.6.8 Approximating Sizes of Dependency Sets

We studied the effectiveness of our approximation heuristic from Section 4.5 for choosing a batch of shared requests. Computing the optimal batch accurately was costly, and significantly lowered the throughput. However, we measured the scheduling overhead, and compared it to when we used an approximation. We ran TPC-C, and varied the number of clients from 100 to 900. As shown in Figure 4.18, our approximation reduced the scheduling overhead by up to 80x.

We also measured the error of our approximation technique for estimating the dependency set sizes—the deviation from the actual sizes of the dependency sets—for varying ratios of shared locks in the workload. Figure 4.19 shows the complementary cumulative distribution function (CCDF) of the relative error of approximating the dependency set sizes. The error grew with the ratio of shared locks; this was expected, as shared locks are the cause of error in our approximation. However, the errors remained within a reasonable range, e.g., even with 80% shared locks, we observed a 2-approximation of the exact sizes in 99% of the cases.

4.7 Future Work

In this section, we present some initial ideas regarding extending the LDSF algorithm to handle Hybrid OLTP-OLAP (a.k.a. HTAP) workloads. Due to their drastically different characteristics, these two types of workloads are typically served by different systems with the data in OLAP systems periodically refreshed in order to keep it up to date with the transactional data. This solution suffers from data staleness as well as extra operational costs. A recent trend is to merge these two systems into one and to run OLAP queries on the most recent transactional data. In this section, we explore the possibility of supporting such hybrid workloads in a transactional database by extending our transaction scheduling algorithm. In particular, we focus on MySQL.

4.7.1 Background

Traditionally, database workloads have been categorized under two distinct types of database workloads: online transaction processing (OLTP) workloads, comprised of high-frequency, read-write queries each touching a small portion of the data, and online analytical processing (OLAP) workloads, comprised of ad-hoc, read-only queries that process large volumes of data and compute aggregate values. Due to drastically different requirements of these workloads, different database systems are used to support each: a transactional DBMS to support OLTP workloads, and an OLAP database (a.k.a. *data warehouse*) to support OLAP workloads. These two types of systems use specialized, and sometimes contradicting optimization techniques and data structures to achieve the best performance for their target workload [146]. In order to keep the data up to date, the data is copied from the OLTP database to the OLAP database periodically through ETL (Extract-Transform-Load) operations [82, 120, 161, 192, 193] However, the periodic nature of this process as well as the freshness gap are still unacceptable to many modern applications (e.g., fraud detection, IoT).

In recent years, there has been an increased demand for *real-time* or *operational business intelligence* (BI), whereby business decisions are made on the latest (i.e., up-to-date) transactional data. Achieving this goal requires designing a database system that is capable of supporting both OLTP and OLAP workloads. Designing such systems, a.k.a. HTAP (Hybrid Transactional-Analytical Processing) databases, is a challenging task. For example, frequent insertion/deletion naturally favors data organization schemes where all columns of the same row are stored consecutively, thus making row store a better option for OLTP workloads. On the other hand, since OLAP workloads usually contain

aggregation operations on a few columns, storing all values of the same column consecutively is more cache-friendly, and would also increase compression opportunities.

Designing a single system that can efficiently support HTAP workloads is a challenging problem. Several approaches have been proposed. BatchDB [137] merges two different systems into one by keeping two replicas of the data, each optimized for one type of workload. It uses a light-weight propagation of transactional updates to keep the OLAP replica up-to-date. Other systems maintain a snapshot of the transactional data for serving OLAP queries, either by utilizing the snapshots in multi-version concurrency control (MVCC) and refreshing relevant data before answering an OLAP query [159], or by using hardware-assisted virtual memory management in in-memory database systems [117, 118, 147]. To reduce the memory footprint of the snapshots—so that more memory can be allocated for storing intermediate results during query processing—data compaction schemes are proposed to compress the data while still allowing for efficient updates [89].

Yet another approach is to resolve the conflicting physical data layouts (row/column stores) required by each type of workload. SnappyData [146] uses a hybrid stores where the most recent data is kept in a row-store and periodically merged into a compressed column-store. Other proposals learn the data access pattern of a given workload offline to decide on the optimal data layout [95], or adaptively change the data layout as queries come in [36, 90]. One of the characteristics of OLTP workloads is that typically a small portion of the entire data is frequently updated, while the rest of the data remains almost unchanged. Some HTAP solutions take advantages of this observation, dividing their data into hot and cold and using different storage layouts for each [89, 127].

A key assumption behind our LDSF and bLDSF algorithms is that the execution time of transaction (time spent after acquiring all locks) follows the same distribution. While a reasonable approximation for most transactional workloads, this assumption is clearly violated in HTAP workloads where the execution time of an analytical query can be several orders of magnitude larger than that of a transaction. Next, we discuss initial ideas for extending our scheduling algorithm to support HTAP workloads, which we call hLDSF (hybrid-LDSF).

4.7.2 hLDSF: A Scheduling Algorithm for HTAP Workloads

hLDSF is an initial attempt to extend our LDSF algorithm to support HTAP workloads. Our intuition behind hLDSF is quite similar to LDSF. We still use the notion of *speed of progress*, but instead of assuming that all transaction execution times follow the same

distribution, we allow different types of transactions and OLAP queries in the workload to have execution times that follow different (and independent) random variables. For simplicity, in the rest of this section, we refer to both transactions and OLAP queries as transactions. Denote the execution times of the transactions waiting in a queue as random variables $R_1, R_2, \cdots, R_n$. hLDSF relies on the expected value of these random variables to measure the time it takes to make progress. hLDSF tries to find one or a batch of transactions that, if granted the lock, would maximize the speed in which we can make progress. The speed of progress we can achieve for the current workload—by granting the lock to a set of transactions—can be measured as $\frac{\sum_{i=1}^{m} |g(t_i^*)|}{\max(E(R_{t_1^*}), \cdots, E(R_{t_m^*}))}$, where $t^* \in G$ is the set of transactions to be granted the lock. The pseudocode for hLDSF is shown in Algorithm 7. As shown in Line 4, hLDSF groups all transactions requesting for shared locks, and schedules all of them together if it decides to grant a shared lock. In other words, it considers granting the lock either to one of the transactions requesting for an exclusive lock (the one that maximizes the value above, Line 5), or to grant the lock to the batch of shared transactions (Line 7 to 11).

Algorithm 7: *The hLDSF Algorithm*

Input : The dependency graph of the system $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$,
execution time random variables $R_{t_1}, R_{t_2}, \cdots$ for each $t_i \in \mathcal{V}$
transaction t,
object o,
label $L \in \{X, S\}$
// meaning t has just released a lock of type L on o

Output: The set of transactions whose requested lock on o should be granted

1 **if** *there are other transactions still holding a lock on* o **then**
2 | **return** $\emptyset$;
3 **end**
4 Obtain the set of transactions waiting for a shared lock on o,
$T^i \leftarrow \{t^i \in \mathcal{V} : (t^i, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^i, o) = S\} = \{t_1^i, t_2^i, \cdots, t_m^i\}$;
5 Obtain the set of transactions waiting for an exclusive lock on o,
$T^x \leftarrow \{t^x \in \mathcal{V} : (t^x, o) \in \mathcal{E} \text{ and } \mathcal{L}(t^x, o) = X\} = \{t_1^x, t_2^x, \cdots, t_n^x\}$;
6 Let $\hat{t^x}$ be the transaction in T^x such that $\frac{|g(\hat{t^x})|}{E(R_{\hat{t^x}})}$ is maximized ;
7 **if** $\frac{|g(\hat{t^x})|}{E(R_{\hat{t^x}})} \leq \frac{\sum_{i=1}^{m} |g(t_i^i)|}{\max(E(R_{t_1^i}), \cdots, E(R_{t_m^i}))}$ **then**
8 | **return** T^i;
9 **else**
10 | **return** $\hat{t^x}$;
11 **end**

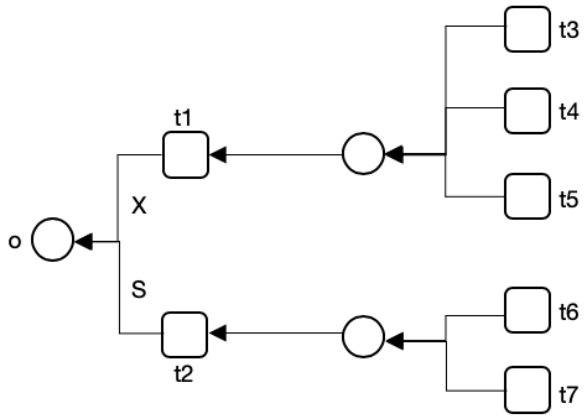

Figure 4.20: Example of how hLDSF works

Figure 4.20 shows a toy example to demonstrate how hLDSF can perform better than LDSF. Let t_1 and t_2 be an analytical query and a transaction, respectively, and let their expected execution times be $E(R_1)$ and $E(R_2)$, where $E(R_1) \approx 100 \cdot E(R_2)$. Under LDSF, the lock on o will be granted to t_1 first, because it has a larger dependency set. For simplicity, assume that transactions $t3$ to $t7$ all try to acquire shared locks, and their expected execution times are all $E(R_3)$. Granting the lock to t_2 first will result in a total latency of $7E(R_1) + 3E(R_2) + 5E(R_3)$. Under hLDSF, on the other hand, the lock will be granted to t_2 since $\frac{|g(t_1)|}{E(R_1)} < \frac{|g(t_2)|}{E(R_2)}$, and the total latency of all transactions will be $4E(R_1) + 7E(R_2) + 5E(R_3)$, which is much smaller than that of LDSF. We discuss the implementation of this algorithm in Section 4.7.3, and present some preliminary results on its performance in Section 4.7.4.

4.7.3 Implementation of hLDSF

As discussed in Section 4.7.1, most transactional databases suffer a significant slowdown when faced with HTAP (i.e., hybrid) workloads. MySQL, which we use in this dissertation as a target system, is not an exception. While its row-store data layout allows for extremely efficient transaction processing, it is a major disadvantage for OLAP queries, which mostly operate on the same column of numerous rows. Since MySQL relies on locking for concurrency control, depending on the isolation level, even running just a handful of OLAP queries can block most of the in-flight transactions, bringing down the entire system to a near halt.

Our focus in this section is not on building an entirely new DBMS for HTAP workloads. Rather, we aim to study the impact of the scheduling algorithm on increasing the

ability of an off-the-shelf transactional DBMS (i.e., MySQL) in coping with HTAP workloads. In other words, to implement our hLDSF algorithm, we only modify MySQL's lock manager, leaving its storage engine unchanged. Our implementation of hLDSF is quite similar to that of LDSF and bLDSF, except for the following modifications. Unlike LDSF's implementation, with hLDSF we now have to estimate the expected remaining time of each transaction. Predicting transaction's execution or remaining time is an extremely difficult open problem [142, 143]. Therefore, in our current implementation, we assume a template-based OLTP workload, whereby each transaction is generated from the same query *template* (a.k.a. transaction type), modulo actual constants in the query. For each transaction type, we compute and maintain their average remaining time whenever a scheduling occurs. We then use this value as our estimated remaining time of the transactions of a particular type, which in turn helps us decide on which transactions to grant the lock to.

4.7.4 Preliminary Experiments

In this section, we present some preliminary results to demonstrate the performance of hLDSF. First, we compared our implementation of the hLDSF algorithm to the original FIFO algorithm in MySQL. For this, we created an HTAP workload based on the TPC-C workload, with one extra OLAP query:

```
SELECT COUNT(DISTINCT OL_O_ID)
FROM WAREHOUSE, ORDERLINE
WHERE ? <= W_ID AND
          W_ID >= ? AND
          AND OL_W_ID = W_ID
GROUP BY W_STATE
```

We also reduced the ratio of the StockLevel transaction by 0.5%, and set the ratio of this OLAP query to 0.5%. In this experiment, we first get the maximum achievable rate under hLDSF with different number of clients, and then we run both algorithms under this rate. Figure 4.21 and 4.22 show the comparison of their average latency and 99th percentile, respectively. As we can see here, hLDSF fails to produce stable improvement over FIFO.

Our hypothesis for the result is that this is due to two main reasons: (i) the overall difficulty of supporting a HTAP workload in MySQL, which reduces the impact that the lock scheduling algorithm can have on overall performance, and (ii) the additional overhead of hLDSF as well as the inaccuracy of our remaining time estimation. Thus,

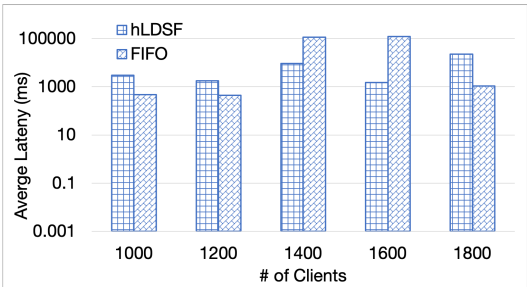

Figure 4.21: Avg. latency of hLDSF and FIFO.

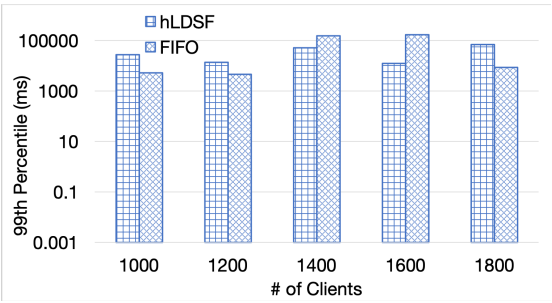

Figure 4.22: 99th percentile of hLDSF and FIFO.

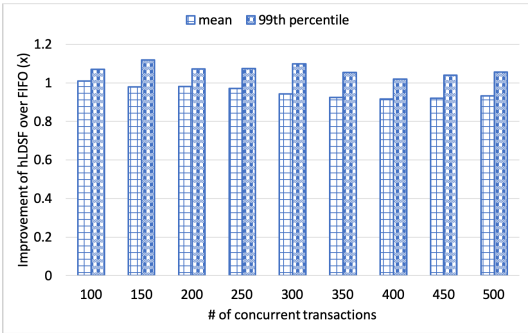

Figure 4.23: Improvement of hLDSF over FIFO in average latency in simulator

to rule out these two possibilities, we also implement a simulator that mimics a transactional database. However, instead of measure actual time, we measure the number of epochs each transaction spends in the simulator. Further, to avoid the estimation error, our simulator uses its *a priori* knowledge of each transaction (again, expressed in number of epochs).

In this simulator, we use a single thread to simulate multiple transactions running in parallel in order to avoid the randomness caused by thread scheduling. The simulator creates N transaction objects, each with a list of locks it requires as well as the number of epochs needed to finish execution once all locks have been acquired. We use a mixture of two types of transactions, one which acquires fewer locks and takes less time to finish, and another one which acquires significantly more locks and takes much longer to finish. The simulator then loops through all active transactions, making progress for each of them for one epoch. The scheduling decisions happen at the end of each epoch and are assumed to take zero time. In other words, once a transaction releases a lock, the scheduling happens immediately after, granting the lock to another transaction. As soon as a transaction finishes, the simulator creates a new one to maintain a fixed number of concurrent transactions in the system.

Similar to the experiment in MySQL, we varied the number of concurrent transactions in the simulator, and measured improvement of hLDSF over FIFO in terms of mean, 95th percentile and 99th percentile. As is shown in Figure 4.23, although not improving much on average latency, our hLDSF is able to improve on 95th percentile up to 20%.

4.8 Related Work

In short, the large body of work on traditional job scheduling is unsuitable in a database context due to the unique requirements of locking protocols deployed in databases. Although there is some work on lock scheduling for real-time databases, they aim at supporting explicit deadlines rather than minimizing the mean latency of transactions.

Job Scheduling— Outside the database community, there has been extensive research on scheduling problems in general. Here, the duration (and sometimes the weight and arrival time) of each task is known *a priori*, and a typical goal is to minimize (i) the sum of (weighted) completion times (SCT) [104, 108, 156], (ii) the latest completion time [63, 96, 167], (iii) the completion time variance (CTV) [48, 54, 122, 191], or even (iv) the waiting time variance (WTV) [80]. The offline SCT problem can be optimally solved using a Shortest-Weighted-Execution-Time approach, whereby jobs are scheduled in the non-decreasing order of their ratio of execution time to weight [176], if they all arrive at the same time. However, when the jobs arrive at different times, the scheduling becomes NP-hard [130].

None of these results are applicable to our setting, mainly because of their assumption that each processor/worker can be used by only one job at a time, whereas in a database, locks can be held in shared and exclusive modes. Moreover, they assume the execution time of each job is known, which is not the case in a database (i.e., the database does not know when the application/user will commit and release its locks). Finally, with the exception of [108, 156], prior work on scheduling either assumes that all tasks are available at the beginning, or that their arrival time is known. In a database, however, such information is unavailable.

Dependency-based Scheduling— Scheduling tasks with dependencies among them has been studied for both single machines [109, 174] and multiprocessors [76, 77, 84, 148]. Here, each job only needs one processor and once scheduled, it will not be blocked again. However, in a database, a transaction can request many locks, and thus, can be blocked even after it is granted some locks.

Real-time Databases (RTDB)— There is some work on lock scheduling in the context of RTDBs, where transactions are scheduled to meet a set of user-given deadlines [27, 29, 38, 57, 91, 99, 101, 107, 131, 134, 164, 179, 186, 188, 189, 198]. It is shown that the First-In-First-Out (FIFO) policy performs poorly in this setting [27, 29, 91, 131], compared to the Earliest-Deadline-First policy [134, 179, 198], which is also used in practice [38].

Unfortunately, the work in this area is not applicable to general-purpose database systems. First, in an RTDB, each transaction comes with a pre-specified deadline, while in a general-purpose database such deadlines are not provided. Second, a key assumption in this line of work is that the execution time of each transaction is known in advance, whereas in a general database the execution time of a transaction is only known once it is finished. Finally, the scheduling goal in an RTDB is to minimize the total tardiness or the number of missed deadlines. In other words, as long as a transaction meets its deadline, RTDBs do not care whether it finishes right before the deadline or much earlier. In contrast, general databases aim to execute transactions as fast as possible.

Scheduling in Existing DBMS— For simplicity and fairness [37], the First-In-First-Out (FIFO) policy and its variants are the most widely adopted scheduling policies in many of today's databases [30], operating systems [52], and communication networks [129]. FIFO is the default lock scheduling policy in MySQL [8], MS SQL Server [13], Postgres [11], Teradata [10], and DB2 [3]. Despite its popularity, FIFO does not provide any guarantees in terms of average or percentile latencies. Huang et al. [110] propose a scheduling algorithm, called Variance-Aware Transaction Scheduling (VATS), which aims at minimizing the variance of transaction latencies, and its optimality holds only when there are no shared locks in the system. In contrast, we focus on minimizing mean latency, and allow for both shared and exclusive locks. In short, designing optimal lock scheduling algorithms for databases has remained an open problem.

VATS— Based on the findings of a new profiler, called VProfiler [111], we have previously proposed Variance-Aware Transaction Scheduling (VATS) [110]. VATS prioritizes transactions according to their arrival time in the system, as opposed to FIFO, which prioritizes them according to their arrival time in the current queue. Our prior work proves the optimality of VATS in terms of minimizing the L_p-norm of transaction latencies [110], when there are no shared locks. In contrast, the current paper proves the optimality of bLDSF in terms of minimizing mean latency. More importantly, our analysis of VATS uses a simplifying assumption that models the latency of a transaction t as $l(t) = A(t) + U(t) + R \cdot (N(t) + 1)$, where $A(t)$ is the age of t (i.e., time since arrival), $U(t)$ is the time since t arrives in the current queue until the lock becomes available, and

$N(t)$ is the number of transactions in the current queue that will be scheduled before t. However, VATS does not account for the fact that $U(t)$ itself can be affected by the scheduling decision. In this dissertation, in our analysis of bLDSF, we have been able to remove both assumptions and hence, prove optimality under a more realistic setting. We consider both shared and exclusive locks, and account for the impact of our scheduling decision on the wait times of other transactions waiting for other objects in the system. Our experiments show that bLDSF's more realistic assumptions lead to better decisions (see Section 4.6).

Deadlock Resolution— The problem of *deadlock resolution* is about deciding which transaction(s) to abort in order to resolve a deadlock [87, 100, 112, 136, 162]. Typically, transactions with lower priority are aborted, in order to reduce the amount of work wasted. Here, transactions are prioritized based on their age [32, 87], deadline [116], or number of held locks [141]. Franaszek et al. [87] empirically show that an age-based priority improves concurrency, and reduces the amount of work wasted. Agrawal et al. [32] argue that choosing victims based on their age and number of held locks leads to fewer rollbacks, than (i) choosing a transaction randomly, or (ii) aborting the most recently blocked transaction. These proposals take contention into account only for deadlock resolution. In contrast, we focus on lock scheduling and show that contention-aware scheduling yields significant performance benefits in practice.

4.9 Summary

In this chapter, we studied a fundamental (yet, surprisingly overlooked) problem: lock scheduling in a transactional database system. Despite the massive body of work on transactional databases, the dramatic impact of lock scheduling on overall performance of a transactional system seems to have gone largely unnoticed—to the extent that every DBMS to date has simply relied on FIFO. To the best of our knowledge, we are the first to propose the idea of contention-aware lock scheduling, and present efficient algorithms that are guaranteed to reduce mean transaction latencies down to a constant-factor-approximation of the optimal scheduling. We also empirically confirm our theoretical analysis by modifying a real-world DBMS. The experiments in this chapter show that our algorithms reduce transaction latencies by up to two orders of magnitude, while delivering 6.5x higher throughput. Our algorithm has already been adopted by MySQL, and has started to impact real world applications.

Chapter 5

Conclusion and Future Work

This dissertation is an extensive study of performance unpredictability in modern transactional systems, a key, yet neglected problem in database research. Diagnosing the causes of performance unpredictability is challenging, due to a lack of suitable profilers. Existing software profilers are mostly designed for reporting *average*. To help diagnose performance unpredictability in large and modern codebases (such as a DBMS), we propose VProfiler, a novel profiling tool that supports *semantic intervals* and analyzes concurrent executions spanning multiple threads. We conduct a quantitative case study of performance unpredictability in real-world database systems, and present our findings and the lessons learned. We also introduce alternative algorithms, implementations, and tuning strategies that make performance more predictable.

One of our main findings is the importance of lock scheduling on the overall performance of a transactional database. Consequently, we formalize and investigate this problem, and introduce the concept of *Contention-Aware Transaction Scheduling (CATS)*. CATS aims to improve performance predictability through scheduling decisions that reduce overall contention in the system.

5.1 Contributions

The first major contribution of this dissertation is the design and implementation of VProfiler. As the first profiler designed to diagnose performance predictability issues, VProfiler introduces a novel abstraction called *variance tree* to capture performance variance in a software system. It employs critical path reconstruction to support modern software systems, where the execution of a *semantically defined interval* can happen on multiple threads. VProfiler is able to pinpoint a few most important variance contribu-

tors out of hundreds of thousands of functions within a complex codebase, making it much easier to closely study the actual root causes of performance unpredictability. The design of VProfiler is therefore a fundamental stepping stone of this dissertation, as it allows us to diagnose and analyze existing systems to better understand the causes of unpredictability.

Our second major contribution is our detailed case study of the most popular database systems under various configurations. Typically, users and companies are often reluctant to abandon a database system into which they have invested their time and resources. Our case study is designed to improve these existing systems: first to investigate their main sources of performance variance, and second to resolve them by modifying their underlying data structures, algorithms or parameters. In particular, we carry out our case study by running VProfiler on MySQL, PostgresSQL and VoltDB, and despite our lack of familiarity with their codebases, we are able to identify their main causes of performance variance. Based on our findings, we propose Variance-Aware Transaction Scheduling (VATS) and Lazy LRU Update (LLU) to improve MySQL's predictability, and implement Parallel Logging to improve PostgresSQL's predictability. We also introduce Variance-Aware Tuning, where we tune configuration parameters to reduce the performance variance. All of these techniques achieve predictability *without* sacrificing average performance. In fact, in some cases, our techniques even result in improved average performance.

The final major contribution of this dissertation is the introduction of *Contention-Aware Transaction Scheduling* for transactional database systems. One of the most important findings from our case study is that lock scheduling is a main source of performance unpredictability. We thus design a new abstraction called a *dependency graph*, which captures the dependency among in-flight transactions in the system, which we then use to develop our new scheduling algorithms LDSF and bLDSF. We provide theoretical guarantees regarding the optimality of these algorithms. They work by granting locks in a manner that maximizes the *speed* with which the transactions can make progress in the system. Experimental results show that our algorithms significantly increase throughput and reduce percentile latency.

Most notably, our VATS and LDSF algorithms are already adopted by MariaDB and MySQL, respectively, making a positive impact in the real world.

5.2 Real-World Adoption

A key advantage of our top-down approach is its easier adoption by existing systems. Therefore, after the case study with MySQL, we sought to merge our VATS algorithm into MariaDB, which is a fork of MySQL. After we submitted our initial pull request, VATS algorithm was reviewed and merged into MariaDB.

After proposing our CATS algorithms, we followed the same process for integration. Since our bLDSF algorithm required a delay factor, which would depend on the actual workload, we decided to merge the simpler yet efficient variant, namely LDSF, into MySQL and Percona, which is another fork of MySQL. First, we made the necessary changes to the code to conform to MySQL's coding style, and handled corner cases that we had not considered in our experiments. We also addressed various test failures reported by MySQL and Percona teams that were not exposed by our own tests. Subsequently, both MySQL and Percona teams performed their own independent evaluations, using TPC-C and **SysBench** [121], respectively. They both observed the same performance (i.e., no improvement) for low-contention settings, while observing significant performance improvements for high-contention settings, e.g., up to 20x and 5x in terms of mean and 95th percentile latencies, respectively. Their independent tests led to the approval and merging of our patch into MySQL's main codebase as its default scheduling algorithm. Later, we further communicated with both teams to ensure they completely understand the logic and the code and can take it over for future maintenance needs. We also contributed additional test cases. Finally, MySQL 8.0.3 was released, as the first version of MySQL with our LDSF algorithm as the default scheduling algorithm. Later, Percona 8.0, which was was based on MySQL 8.0, also included LDSF.

5.3 Future Work

Based on our findings, we recommend further research on transaction scheduling. In particular, our results on hybrid scheduling are not conclusive. We see no obvious trend in the experiment results as the number of clients increase in both MySQL and the simulator under the HTAP workload, even with the original FIFO algorithm. This is counter-intuitive and might lead to some interesting findings about why our hLDSF is not working as expected in the experiments.

Another direction for future research is to extend our existing algorithms to different isolation levels. So far, our algorithms assume a serializible isolation level in the transactional database systems. However, this is a very strict requirement, and a lot of

workloads do not need such a strict isolation level. For example, MySQL defaults to the Repeatable Read isolation level [7]. Some of the assumptions we made in our current algorithms will break under different isolation levels. For example, under read committed, read locks are not required to be held until a transaction commits. On the contrary, they can be released as soon as the current query finishes. The algorithms themselves will have to be changed to adapt to these kind of differences.

Bibliography

[1] 15.5.2 InnoDB Transaction Model. http://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-model.html.

[2] Contention-aware transaction scheduling arriving in innodb to boost performance. http://mysqlserverteam.com/contention-aware-transaction-scheduling-arriving-in-innodb-to-boost-performance/.

[3] Db2 documentation. https://www.ibm.com/support/knowledgecenter/en/SSEPEK_12.0.0/perf/src/tpc/db2z_lockcontention.html.

[4] Erratic performance problems in oracle > 8.0.x. http://www.wikiguga.com/topic/c67f177136f542809da3106c5f875e68.

[5] Google Cloud SQL. http://code.google.com/apis/sql.

[6] Mariadb source repository. https://github.com/MariaDB/server/pull/248.

[7] Mysql :: Mysql 8.0 reference manual :: 15.7.2.1 transaction isolation levels. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html. Accessed: 2019-06-26.

[8] Mysql source code. https://github.com/mysql/mysql-server/blob/5.7/storage/innobase/lock/lock0lock.cc.

[9] Oracle database cloud service. http://cloud.oracle.com.

[10] Overview of teradata database locking. http://info.teradata.com/HTMLPubs/DB_TTU_16_00/index.html#page/General_Reference%2FB035-1091-160K%2Fmtg1472241438567.html.

[11] Postgres documentation. https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/README.

[12] Program2 Two-Phase Locking (2PL) vs. Timestamp Ordering (TSO) vs. A Real Protocol. http://cobweb.cs.uga.edu/~shasha/course/csci8370/prog2/prog2.pdf.

[13] Sql server, lock manager, and âĂIJrelaxedâĂİ fifo. https://blogs.msdn.microsoft.com/psssql/2009/06/02/sql-server-lock-manager-and-relaxed-fifo/.

[14] Google on latency tolerant systems: Making a predictable whole out of unpredictable parts. http://highscalability.com/blog/2012/6/18/google-on-latency-tolerant-systems-making-a-predictable-whol.html, 2002.

[15] Understanding flash: Unpredictable write performance. http://flashdba.com/2014/12/10/understanding-flash-unpredictable-write-performance/, 2004.

[16] Server unpredictable random performance hiccups. https://www.percona.com/forums/questions-discussions/mysql-and-percona-server/401-server-unpredictable-random-performance-hiccups, 2006.

[17] Erratic performance of SQL Server. http://www.bigresource.com/MS_SQL-Erratic-Performance-of-SQL-Server-6rJpQdOC.html, 2007.

[18] Query execution time is unpredictable. https://community.oracle.com/thread/961135?start=0&tstart=0, 2009.

[19] Mt lru flusher. https://blueprints.launchpad.net/percona-server/+spec/mt-lru, 2011.

[20] Parallel doublewrite buffer. https://blueprints.launchpad.net/percona-server/+spec/parallel-doublewrite, 2011.

[21] SQL Server Locking and You! https://www.brentozar.com/archive/2011/06/sql-server-locking/, 2011.

[22] Xtradb performance improvements for i/o-bound highly-concurrent workloads. https://www.percona.com/doc/percona-server/5.7/performance/xtradb_performance_improvements_for_io-bound_highly-concurrent_workloads.html, 2011.

[23] Unpredictable performance and slowness between our azure website and a sql server running on an azure vm. http://tinyurl.com/qzleb43, 2015.

[24] MySQL Transaction Lock Manager Source Code. https://github.com/mysql/mysql-server/blob/5.7/storage/innobase/lock/lock0lock.cc, 2016.

[25] Percona server 5.7: multi-threaded lru flushing. https://www.percona.com/blog/2016/05/05/percona-server-5-7-multi-threaded-lru-flushing/, 2016.

[26] Postgre Transaction Lock Manager Source Code. https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/proc.c, 2016.

[27] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transactions. *SIGMOD Rec.*, pages 71–81, 1988.

[28] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *TODS*, 1992.

[29] Robert K Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *TODS*, pages 513–560, 1992.

[30] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. Database support for efficiently maintaining derived data. In *EDBT*, pages 223–240, 1996.

[31] Rakesh Agrawal. A parallel logging algorithm for multiprocessor database machines. In *Database Machines*. 1985.

[32] Rakesh Agrawal, Michael J Carey, and Larry W McVoy. The performance of alternative strategies for dealing with deadlocks in database management systems. *TSE*, pages 1348–1363, 1987.

[33] Rakesh Agrawal and David J DeWitt. *Recovery architectures for multiprocessor database machines*. ACM, 1985.

[34] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, 2003.

[35] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *SIGMOD*, pages 1009–1024, 2017.

[36] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2o: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1103–1114. ACM, 2014.

[37] Sebastian Altmeyer, Sakthivel Manikandan Sundharam, and Nicolas Navet. The case for fifo real-time scheduling. Technical report, 2016.

[38] Rohan FM Aranha, Venkatesh Ganti, Srinivasa Narayanan, CR Muthukrishnan, STS Prasad, and Krithi Ramamritham. Implementation of a real-time database system. *Information Systems*, pages 55–74, 1996.

[39] Michael Armbrust, Kristal Curtis, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5, 2011.

[40] Michael Armbrust, Eric Liang, Tim Kraska, Armando Fox, Michael J. Franklin, and David A. Patterson. Generalized scale independence through incremental precomputation. In *SIGMOD*, 2013.

[41] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage; recovery methods for non-volatile memory database systems. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2015.

[42] Subi Arumugam, Alin Dobra, Christopher M Jermaine, Niketan Pansare, and Luis Perez. The datapath system: a data-centric analytic processing engine for large data warehouses. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2010.

[43] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[44] Peter David Bailis. *Coordination Avoidance in Distributed Databases*. PhD thesis, University of California, Berkeley, 2015.

[45] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[46] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Hot Topics in Operating Systems (HotOS)*, 2003.

[47] CR Bector, Yash P Gupta, and Mahesh Chander Gupta. V-shape property of optimal sequence of jobs about a common due date on a single machine. *Computers & operations research*, 1989.

[48] CR Bector, Yash P Gupta, and Mahesh Chander Gupta. V-shape property of optimal sequence of jobs about a common due date on a single machine. *Computers & operations research*, pages 583–588, 1989.

[49] Andrew R Bernat and Barton P Miller. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience*, 2007.

[50] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, pages 185–221, 1981.

[51] Sapan Bhatia, Abhishek Kumar, Marc E Fiuczynski, and Larry L Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[52] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel*. 2005.

[53] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications (IJHPCA)*, 2000.

[54] X Cai. V-shape property for job sequences that minimize the expected completion time variance. *EJOR*, 1996.

[55] George Candea, Neoklis Polyzotis, and Radek Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment (PVLDB)*, 2009.

[56] Michael J Carey and Michael Stonebraker. The performance of concurrency control algorithms for database management systems. In *VLDB*, pages 107–118, 1984.

[57] Sharma Chakravarthy, Dong-Kweon Hong, and Theodore Johnson. Real-time transaction scheduling: A framework for synthesizing static and dynamic factors. *RTSS*, pages 135–170, 1998.

[58] Anupam Chanda, Alan L Cox, and Willy Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. *ACM SIGOPS Operating Systems Review*, 2007.

[59] Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. Variance aware optimization of parameterized queries. In *SIGMOD*, 2010.

[60] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks (DSN)*, 2002.

[61] Shimin Chen. Flashlogging: exploiting flash devices for synchronous logging performance. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2009.

[62] Wen-Jinn Chen, Su-Mei Lin, and Jia-Chi Tsou. Sequencing heuristic for bicriteria scheduling in a single machine problem. *Journal of Information and Optimization Sciences*, 2006.

[63] Byung-Cheon Choi, Suk-Hun Yoon, and Sung-Jin Chung. Minimizing maximum completion time in a proportionate flow shop with one machine of different speed. *EJOR*, pages 964–974, 2007.

[64] Francis Chu et al. Least expected cost query optimization: An exercise in utility. In *PODS*, 1999.

[65] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric P Xing. Solving the straggler problem with bounded staleness. In *HotOS*, pages 22–22, 2013.

[66] Brian F Cooper et al. Benchmarking cloud serving systems with ycsb. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.

[67] Anirban DasGupta. Finite sample theory of order statistics and extremes. In *Probability for Statistics and Machine Learning*, pages 221–248. 2011.

[68] Laurent Daynès, Olivier Gruber, and Patrick Valduriez. Locking in oodbms client supporting nested transactions. In *ICDE*, pages 316–323, 1995.

[69] Prabuddha De, Jay B Ghosh, and Charles E Wells. On the minimization of completion time variance with a bicriteria extension. *Operations Research*, 1992.

[70] Luiz De Rose, Ying Zhang, and Daniel A Reed. Svpablo: A multi-language performance analysis system. In *Computer Performance Evaluation*. Springer, 1998.

[71] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[72] Christina Delimitrou and Christos Kozyrakis. ibench: Quantifying interference for datacenter applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, 2013.

[73] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment (PVLDB)*, 7, 2013.

[74] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[75] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi. Limplock: understanding the impact of limpware on scale-out cloud systems. In *ACM Symposium on Cloud Computing (SoCC)*, 2013.

[76] Jianzhong Du and Joseph Y-T Leung. Scheduling tree-structured tasks with restricted execution times. *Information processing letters*, pages 183–188, 1988.

[77] Jianzhong Du and Joseph Y-T Leung. Scheduling tree-structured tasks on two processors to minimize schedule length. *SIDMA*, pages 176–196, 1989.

[78] Andrea C Dusseau, Remzi H Arpaci, and David E Culler. Effective distributed scheduling of parallel workloads. *ACM SIGMETRICS Performance Evaluation Review*, pages 25–36, 1996.

[79] Samuel Eilon and IG Chowdhury. Minimising waiting time variance in the single machine problem. *Management Science*, 1977.

[80] Samuel Eilon and IG Chowdhury. Minimising waiting time variance in the single machine problem. *Management Science*, pages 567–575, 1977.

[81] Bennett Eisenberg. On the expectation of the maximum of iid geometric random variables. *Statistics & Probability Letters*, pages 135–143, 2008.

[82] Shaker H Ali El-Sappagh, Abdeltawab M Ahmed Hendawi, and Ali Hamed El Bastawissy. A proposed model for data warehouse etl processes. *Journal of King Saud University-Computer and Information Sciences*, 23(2):91–104, 2011.

[83] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[84] Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal online scheduling of parallel jobs with dependencies. In *STOC*, pages 642–651, 1993.

[85] Kurt B Ferreira, Patrick G Bridges, Ron Brightwell, and Kevin T Pedretti. The impact of system design parameters on application noise sensitivity. *Cluster computing*, pages 117–129, 2013.

[86] Daniela Florescu and Donald Kossmann. Rethinking cost and performance of database systems. *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2009.

[87] Peter Franaszek and John T Robinson. Limitations of concurrency in transaction processing. *TODS*, 1985.

[88] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead call path profiling of unmodified, optimized code. In *International Conference on Supercomputing (ICS)*, 2005.

[89] Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting transactional data in hybrid oltp&olap databases. *Proceedings of the VLDB Endowment*, 5(11):1424–1435, 2012.

[90] Florian Andreas Funke. *Adaptive Physical Optimization in Hybrid OLTP & OLAP Main-Memory Database Systems*. PhD thesis, Technische Universität München, 2015.

[91] Laurent George and Pascale Minet. A fifo worst case analysis for a hard real-time distributed problem with consistency constraints. In *ICDCS*, pages 441–448, 1997.

[92] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Shareddb: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment (PVLDB)*, 2012.

[93] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, 1982.

[94] Brendan Gregg. DTrace pid Provider return. http://tinyurl.com/jzpphne, 2011.

[95] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. Hyrise: a main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010.

[96] AGP Guinet and MM Solomon. Scheduling hybrid flowshops to minimize maximum tardiness or maximum completion time. *IJPR*, pages 1643–1654, 1996.

[97] Leslie A Hall, David B Shmoys, and Joel Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. In *SODA*, pages 142–151, 1996.

[98] Robert J Hall. Call path profiling. In *International Conference on Software Engineering (ICSE)*, 1992.

[99] Jayant R Haritsa, Michael J Canrey, and Miron Livny. Value-based scheduling in real-time database systems. *VLDBJ*, pages 117–152, 1993.

[100] Jayant R Haritsa, Michael J Carey, and Miron Livny. Data access scheduling in firm real-time database systems. *RTSS*, pages 203–241, 1992.

[101] Jayant R Haritsa, Miron Livny, and Michael J Carey. Earliest deadline scheduling for real-time database systems. In *RTSS*, pages 232–242, 1991.

[102] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2008.

[103] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2005.

[104] Cheng He, Joseph Y-T Leung, Kangbok Lee, and Michael L Pinedo. Improved algorithms for single machine scheduling with release dates and rejections. *4OR*, pages 41–55, 2016.

[105] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group commit timers and high volume transaction systems. In *HPTS*. 1989.

[106] Jonathan MD Hill, Stephen A Jarvis, Constantinos Siniolakis, and Vasil P Vasilev. Analysing an sql application with a bsplib call-graph profiling tool. In *Euro-ParâĂŹ98 Parallel Processing*, 1998.

[107] D Hong, Theodore Johnson, and Sharma Chakravarthy. *Real-time transaction scheduling: a cost conscious approach*. 1993.

[108] Johannes Adzer Hoogeveen and Arjen PA Vestjens. Optimal on-line algorithms for single-machine scheduling. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 404–414, 1996.

[109] WA Horn. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAP*, pages 189–202, 1972.

[110] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas Wenisch. A top-down approach to achieving performance predictability in database systems. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2017.

[111] Jiamin Huang, Barzan Mozafari, and Thomas Wenisch. Statistical analysis of latency through semantic profiling. In *EuroSys*, 2017.

[112] Jiandong Huang. *Real-time transaction processing: design, implementation, and performance evaluation*. PhD thesis, University of Massachusetts, 1991.

[113] Toshihide Ibaraki, Tiko Kameda, and Naoki Katoh. Cautious transaction schedulers for database concurrency control. *TSE*, pages 997–1009, 1988.

[114] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 1987.

[115] Gautam C Kamath. Bounds on the expectation of the maximum of samples from a gaussian. *URL http://www.gautamkamath.com/writings/gaussian_max.pdf*, 2015.

[116] Ben Kao and Hector Garcia-Molina. An overview of real-time database systems. In *Real Time Computing*, pages 261–282. 1994.

[117] Alfons Kemper and Thomas Neumann. One size fits all, again! the architecture of the hybrid oltp&olap database management system hyper. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 7–23. Springer, 2010.

[118] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.

[119] Young-Kuk Kim and Sang H Son. Supporting predictability in real-time database systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, 1996.

[120] Ralph Kimball and Joe Caserta. *The data warehouse ETL toolkit: practical techniques for extracting, cleaning, conforming, and delivering data*. John Wiley & Sons, 2011.

[121] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.

[122] Abba M Krieger and Madabhushi Raghavachari. V-shape property for optimal schedules with monotone penalty functions. *Computers & operations research*, pages 533–534, 1992.

[123] Wieslaw Kubiak. Completion time variance minimization on a single machine is difficult. *Operations Research Letters*, 1993.

[124] Wieslaw Kubiak. New results on the completion time variance minimization. *Discrete Applied Mathematics*, 1995.

[125] Wieslaw Kubiak et al. Fast fully polynomial approximation schemes for minimizing completion time variance. *Eur. Journal of Operational Research*, 2002.

[126] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 2013.

[127] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: hybrid oltp and olap on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326. ACM, 2016.

[128] Juhnyoung Lee and Sang Hyuk Son. Performance of concurrency control algorithms for real-time database systems., 1996.

[129] John P Lehoczky. Scheduling communication networks carrying real-time traffic. In *RTSS*, pages 470–479, 1998.

[130] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Annals of discrete mathematics*, pages 343–362, 1977.

[131] Hennadiy Leontyev and James H Anderson. Tardiness bounds for fifo scheduling on multiprocessors. In *ECRTS*, pages 71–71, 2007.

[132] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *ACM Symposium on Cloud Computing (SoCC)*, 2014.

[133] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In *SIGMOD*, pages 1659–1674, 2016.

[134] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *JACM*, pages 46–61, 1973.

[135] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[136] Philip P Macri. Deadlock detection and resolution in a codasyl based data management system. In *SIGMOD*, pages 45–49, 1976.

[137] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. Batchdb: efficient isolated execution of hybrid oltp+ olap workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 37–50. ACM, 2017.

[138] Paolo Massa and Paolo Avesani. An experimental study on epinions.com community. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2005.

[139] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 2004.

[140] Barton P Miller, Mark D Callaghan, Jonathan M Cargille, Jeffrey K Hollingsworth, R Bruce Irvin, Karen L Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 1995.

[141] Don P Mitchell and Michael J Merritt. A distributed algorithm for deadlock detection and resolution. In *PODC*, pages 282–284, 1984.

[142] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2013.

[143] Barzan Mozafari, Carlo Curino, and Samuel Madden. DBSeer: Resource and performance prediction for building a next generation database cloud. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[144] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. CliffGuard: A principled framework for finding robust database designs. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2015.

[145] Barzan Mozafari, Eugene Zhen Ye Goh, and Dong Young Yoon. Cliffguard: An extended report. Technical report, University of Michigan, Ann Arbor, 2015.

[146] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. Snappydata: A unified cluster for streaming, transactions, and interactive analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017.

[147] Henrik Mühe, Alfons Kemper, and Thomas Neumann. How to efficiently snapshot transactional data: Hardware or software controlled? In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 17–26. ACM, 2011.

[148] Richard R Muntz and Edward G Coffman Jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *JACM*, pages 324–338, 1970.

[149] Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. Vampir: Visualization and analysis of mpi resources. 1996.

[150] Vivek Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[151] Vivek R. Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *CIDR*, 2013.

[152] Patrick O'Neil et al. A two-phase approach to predictably scheduling real-time transactions., 1996.

[153] Jun Pei, Xinbao Liu, Panos M Pardalos, Wenjuan Fan, and Shanlin Yang. Scheduling deteriorating jobs on a single serial-batching machine with multiple job types and sequence-dependent setup times. *Annals of Operations Research*, pages 175–195, 2017.

[154] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the nvram era. *Proceedings of the VLDB Endowment (PVLDB)*, 2013.

[155] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 55–55, 2003.

[156] Cynthia Phillips, Clifford Stein, and Joel Wein. Scheduling jobs that arrive over time. In *WADS*, pages 86–97, 1995.

[157] Michael Pinedo. *Scheduling: theory, algorithms, and systems*. Springer Science, 2012.

[158] Lin Qiao, Vijayshankar Raman, Frederick Reiss, P Haas, and Guy Lohman. Main-memory scan sharing for multi-core cpus. *Proceedings of the VLDB Endowment (PVLDB)*, 2008.

[159] Weiping Qu, Vinanthi Basavaraj, Sahana Shankar, and Stefan Dessloch. Real-time snapshot maintenance with incremental etl pipelines in data warehouses. In *International Conference on Big Data Analytics and Knowledge Discovery*, pages 217–228. Springer, 2015.

[160] Francois Raab. Tpc benchmark c, standard specification revision 3.0. *Transaction Processing Performance Council*, 1995.

[161] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[162] Krithi Ramamritham. Real-time databases. *Distributed and parallel databases*, pages 199–226, 1993.

[163] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-time query processing. In *International Conference on Data Engineering (ICDE)*, 2008.

[164] Rajeev Rastogi, S Seshadri, Philip Bohannon, Dennis Leinbaugh, Avi Silberschatz, and S Sudarshan. Improving predictability of transaction execution times in real-time databases. *RTSS*, pages 283–302, 2000.

[165] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[166] Daniel A Reed, Phillip C Roth, Ruth A Aydt, Keith A Shields, Luis F Tavera, Roger J Noe, and Bradley W Schwartz. Scalable performance analysis: The pablo performance analysis environment. In *Scalable Parallel Libraries Conference (SPLC)*, 1993.

[167] Alex J Ruiz-Torres and Grisselle Centeno. Scheduling with flexible resources in parallel workcenters to minimize maximum completion time. *Computers & operations research*, pages 48–69, 2007.

[168] M. Sadoghi, K. A Ross, M. Canim, and B. Bhattacharjee. Making updates disk-i/o friendly using ssds. *Proceedings of the VLDB Endowment (PVLDB)*, 2013.

[169] Raja R Sambasivan and Gregory R Ganger. Automated diagnosis without predictability is a recipe for failure. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[170] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing performance changes by comparing request flows. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[171] Lui Sha, Ragunathan Rajkumar, and John P. Lehooczky. Concurrency control for distributed real-time databases. *SIGMOD Rec.*, 1988.

[172] Kai Shen, Christopher Stewart, Chuanpeng Li, and Xin Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS Performance Evaluation Review*, 2009.

[173] Sameer Shende, Allen D Malony, Janice Cuny, Peter Beckman, Steve Karmesin, and Kathleen Lindlan. Portable profiling and tracing for parallel, scientific applications using c++. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, 1998.

[174] Jeffrey B Sidney. Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, pages 283–298, 1975.

[175] Benjamin H. Sigelman, Luiz AndrÃl' Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[176] Wayne E Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, pages 59–66, 1956.

[177] Matthew J Sottile and Ronald G Minnich. Supermon: A high-speed cluster monitoring system. In *International Conference on Cluster Computing (CLUSTER)*, 2002.

[178] J Michael Spivey. Fast, accurate call graph profiling. *Software: Practice and Experience*, 2004.

[179] John A Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C Buttazzo. *Deadline scheduling for real-time systems: EDF and related algorithms*. 2012.

[180] M Stonebraker and A Pavlo. The seats airline ticketing systems benchmark.

[181] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *TODS*, 1985.

[182] Zoltán Szebenyi, Felix Wolf, and Brian JN Wylie. Space-efficient time-series call-path profiling of parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[183] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment (PVLDB)*, 2009.

[184] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. Contention-aware lock scheduling for transactional databases. *Technical Report*, https://web.eecs.umich.edu/~mozafari/php/data/uploads/lock-schd-report.pdf, 2017.

[185] Brian Tierney, William Johnston, Brian Crowley, Gary Hoo, Chris Brooks, and Dan Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *High Performance Distributed Computing (HPDC)*, 1998.

[186] D Towsley and SS Panwar. On the optimality of minimum laxity and earliest deadline scheduling for real-time multiprocessors. In *Workshop on Real Time*, pages 17–24, 1990.

[187] Avishay Traeger, Ivan Deras, and Erez Zadok. Darc: Dynamic analysis of root causes of latency distributions. In *ACM SIGMETRICS Performance Evaluation Review*, 2008.

[188] Shin-Mu Tseng, Yeh-Hao Chin, and Wei-Pang Yang. Scheduling real-time transactions with dynamic values: a performance evaluation. In *Workshop on Real-Time Computing Systems and Applications*, pages 60–67, 1995.

[189] Özgür Ulusoy and Geneva G Belford. Real-time transaction scheduling in database systems. *Information Systems*, pages 559–580, 1993.

[190] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment (PVLDB)*, 2009.

[191] Vina Vani and M Raghavachari. Deterministic and random single machine sequencing with variance minimization. *Operations Research*, pages 111–120, 1987.

[192] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for etl processes. In *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 14–21. ACM, 2002.

[193] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. On the logical modeling of etl processes. In *International Conference on Advanced Information Systems Engineering*, pages 782–786. Springer, 2002.

[194] Jeffrey Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *ACM SIGMETRICS Performance Evaluation Review*, 2002.

[195] M. Wainwright. Chapter 2: Basic tail and concentration bounds. http://www.stat.berkeley.edu/~mjwain/stat210b/Chap2_TailBounds_Jan22_2015.pdf.

[196] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment (PVLDB)*, 2014.

[197] A Wolski. Tatp benchmark description, 2009.

[198] Ming Xiong, Qiong Wang, and Krithi Ramamritham. On earliest deadline first scheduling for temporal consistency maintenance. *RTSS*, pages 208–237, 2008.

[199] Jerry C Yan. Performance tuning with aims/spl minus/an automated instrumentation and monitoring system for multicomputers. In *Twenty-Seventh Hawaii International Conference (HICSS)*, 1994.

[200] Robin Jun Yang and Qiong Luo. PTL: Partitioned logging for database storage on flash solid state drives. In *Web-Age Information Management (WAIM)*. 2012.

[201] Nong Ye, Xueping Li, Toni Farley, and Xiaoyun Xu. Job scheduling methods for reducing waiting time variance. *Computers & Operations Research*, 2007.

[202] Dong Young Yoon, Barzan Mozafari, and Douglas P Brown. DBSeer: Pain-free database administration through workload intelligence. *Proceedings of the VLDB Endowment (PVLDB)*, 2015.

[203] Dong Young Yoon, Ning Niu, and Barzan Mozafari. DBSherlock: A performance diagnostic tool for transactional databases. In *ACM Special Interest Groups on Management of Data (SIGMOD)*, 2016.

[204] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment (PVLDB)*, 2014.

[205] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[206] Roman Zajcew, Paul Roy, David L Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, et al. An osf/1 unix for massively parallel multicomputers. In *USENIX Winter*, pages 449–468, 1993.