Sharing DBMS among Multiple Users while Providing Performance Isolation: Analysis and Implementation

David T. McWherter

CMU-CS-08-144
July 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Mor Harchol-Balter, Chair
Christos Faloutsos
Bruce M. Maggs
Hans Zeller

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: DBMS, Databases, Multi-user system, Resource allocation, Query Prioritization

Wizard of Oz*: They have one thing you haven't got: a diploma. Therefore, by virtue of the authority vested in me by the Universitartus Committiartum E Pluribus Unum, I hereby confer upon you the honorary degree of ThD.*
Scarecrow*: ThD?*
Wizard of Oz*: That's... Doctor of Thinkology.*

Abstract

Database Management Systems (DBMS) are at the core of many modern applications, ranging from e-Commerce (e.g. Amazon.COM), web applications (e.g. flickr), online banking, telephony, and even traditional brick-and-mortar retailers. DBMS can be a significant source of delay in these applications, making DBMS the performance bottleneck: users can spend orders of magnitude more time waiting for the DBMS than for anything else (e.g. the web server). Delays often frustrate users, which hurts companies' profits, since frustrated users buy less and are more likely to take their business elsewhere. Adding more capacity (hardware) can reduce delays, but it is usually both more difficult and costly to add capacity to DBMS than to other computer systems (e.g. web servers). Without adding capacity, prioritization can exploit the fact that some users (or queries) are more important than others. Prioritization can give better performance and less delay to high-priority (important) users at the expense of low-priority (less important) users. While prioritization is usually easy in computer systems, prioritization in DBMS is extremely difficult due to complexities inherent to DBMS architectures. As a result, many basic questions concerning DBMS prioritization remain open.

This thesis studies the implementation of prioritization in DBMS (commonly in commercial applications) with high- and low-priority users. The goal is to provide high-priority users with performance isolation, whereby high-priority response times are not affected by low-priority users. I consider common approaches to provide prioritization and experiment with real-world DBMS and benchmark workloads to ensure that the results are applicable to real-world systems. The heart of this work is a performance evaluation of common prioritization approaches, coupled with in-depth statistical analyses to reveal each approach's deficiencies. Our evaluations reveal previously unknown and non-intuitive performance trends about DBMS prioritization, and our analyses provide insight for developing new algorithms and new models for more effective DBMS prioritization. Key algorithmic and modeling contributions of this thesis include the Preempt-On-Wait (POW) lock prioritization algorithm, and the Isolated Demand Decomposition (IDD) modeling method.

Acknowledgments

First, and foremost, I must thank my mother, Loretta "Cookie" McWherter, and my father, David McWherter, whose influences on me have surely led to the construction of this thesis. I love you both.

I thank Corina Bardasuc for being a great companion and roommate, always having a knack of helping me through my problems, even while at the same time making countless more problems for me.

Without my advisor, Mor Harchol-Balter, I would never have gotten through the CMU PhD program, for two reasons: (1) She told me I had to defend and get out by August, and (2) She guided and taught me along the entire process, giving me tremendous insight in many areas of life. She is the only reason I can explain my research to anybody, because of her remarkable ability to forget everything I ever told her about my research, which forced me to re-explain everything from scratch every week.

Countless students and professors at CMU were instrumental to my development. Of particular importance are Adam Wierman, Taka Osogami, David Koes, Benoit Hudson, and Gregory Hartman, as well as the Zephyr crew, including Corey Kosak, Karen Van Dusen, Peter Dinda, Peter Berger, Pete Su, and Stewart Clamen.

The swing dancing community in Pittsburgh is responsible for making me the person that I am today. They transformed me from a shy, timid, and awkward person into a socially-capable dancer and organizer. The dancers helped me find a true joy of both dance and friendship which I never knew I had.

It has been an honor to run the Chicken Swing swing dance, and I thank everybody who came, danced, and supported the dance throughout these years for bringing me unmeasurable joy. Many deserve special credit for their help: Jeffy "Cupcake" Altman for helping run the dance, Katie Rivard for being such an awesome teaching partner (with the ability to both keep me in line, and translate my firehose of gibberish for normal people to understand it), and Yakov Chodosh for inspiring me to start the dance.

Many others deserve special honors for making the dance community, and my social home, a better place: Lisa Tamres and John Fulmer are amazing teachers who really helped get me involved in the community. Lisa Tamres is also the most tireless and persistent organizer I've ever seen, which inspired me to work harder. Bobby Dunlap is responsible for enabling my dance addiction by running Swing City. Joe Forman is a great friend and always makes sure everything runs smoothly. Lisa Matt is also a great friend, as well as a dancing and teaching partner.

I thank coffee and tea. These fine beverages have nourished and stimulated me throughout my tenure at CMU.

In particular, I thank the elves who helped me run the CSD espresso machine. Chief espresso elves Francisco Pereira and Paul Bennett were great leaders for caffeinating the CS department. Elves William

Lovas and Robert Simmons are amazing for their ability to pick up running the machine when the thesis loomed. I thank Elf Bart Nabbe for hacking our machine to heck. Finally, I think Elf Jennifer Landefeld for doing all the really hard work.

Furthermore, I thank the many fine coffee shops which provided countless skinny decaf mochas and enabled as many hours of writing and research: Tazzo d'Oro, Coffee Tree Roasters, 21st Street Coffee and Tea, Kiva Han, Crazy Mocha, Aldo's Coffee, and, of course, Starbucks.

Catherine Copetas and **Sharon Burks** and **Debbie Cavlovich** were all incredibly helpful in navigating me through the twisty maze of departmental requirements towards graduation, and helping to run the espresso machine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 High Level Picture

The concept of sharing is found everywhere in everyday life. Children share toys at daycare, drivers share the road, people share elevators, retail shoppers share cashiers, coffee drinkers share Starbucks coffee shop baristas. The list is endless. Despite the fact that we have learned to share essential *communal resources*, we often do so reluctantly or begrudgingly, since people are fundamentally greedy. We believe that *we are the most important* and our needs are the *highest-priority*. When waiting in line at the coffee shop, we do not care whether the person in front of us ever gets or enjoys their "venti skinny triple-shot double-pump hazelnut latte." The only thing we care about is when *we* will get *our* "grande non-fat half-caf black and white mocha."

Sharing is hard because of the *delay* that it takes for us to acquire resources and get our tasks done. When resources are abundant, delays are low, and sharing is relatively easy because we wait less. When resources are scarce, delays are high, and sharing is much harder because we wait forever to use them. Adding lanes to a heavily trafficked road reduces the delay and suffering caused by traffic jams. Building more Starbucks or hiring more baristas reduces the delay and unendurable suffering you spend waiting behind the guy who is ordering a dozen drinks for everybody in his office.

In the modern, digital era, delay is found in many computer systems and Internet services that we have come to rely on. Since the dawn of the Internet, people regularly experience delays when checking their mail at `gmail.com` or `hotmail.com`, shopping online at `Amazon.COM`, or viewing photos at `flickr.com`. Internet services are particularly susceptible to highly variable delays, dependent on how many people are trying to use and *share* the service at the same time. In fact, any computer system that is used by many people is susceptible to delays in the same way.

We know deep inside that delays are both painful and costly. **Painful**. We have better things to do with the finite time we spend on this mortal coil than waiting in line. Customers, such as those shopping at online stores, find delays to be frustrating [59, 69]. This frustration is partly due to *uncertainty*: customers do not know when delays will arise, or how long those delays will be, or for what reason they are waiting. Customers are also frustrated because delays *waste customers' time* (it is often hard to be productive through delays) and, during many tasks, delays can make it harder to concentrate and focus [61]. **Costly.** Time is

money. When customers experience too much delay and are frustrated, they are more likely to switch to competitors, costing companies in future business and profits. This problem is even worse in online stores and e-Commerce, since competitors are merely a click away. Likewise, when customers spend time waiting for delays, it keeps them from working and making money themselves.

DataBase Management Systems (DBMS) are some of the most shared resources on the planet. Almost every person interacts with DBMS on a daily basis, and almost every time, that DBMS is being shared with countless other people. Hospitals track patients and medical histories with DBMS. Governments use DBMS to store tax and social security records. Companies track all employees, all sales, and all inventories with DBMS. Banks and ATMs use DBMS to store and manage peoples' financial records. Phone companies use DBMS to record call records and track the locations of cell phones in the cell phone network. Almost all online services, ranging from online stores like `Amazon.COM` to online communities like `facebook.com` rely on DBMS. DBMS are *ubiquitous* and inextricably integrated into modern society.

Like any other computer system, DBMS have only limited resources that users compete for, which results in delay. DBMS delays, however, can be much larger and more unpredictable than in other computer systems. DBMS delays can be orders of magnitude larger than the delays seen in other systems. Furthermore, it is usually harder to fix delays in DBMS than in other systems by scaling and adding more hardware to increase capacity and improve performance. Centralized DBMS implementations can be improved by adding faster or additional CPUs or disks, but there are typically several difficulties: (i) limits on device speeds or the number of devices, (ii) high-end hardware necessary for high-end DBMS implementations are very expensive and have high price/performance ratios, and (iii) there are often diminishing performance returns as one invests more money. Furthermore, while many computer systems (such as web servers) can be scaled using distributed system implementations, DBMS do not lend themselves to distributed implementations. Distributed DBMS are often just as costly and difficult to scale as centralized DBMS.

We must devise novel approaches to compensate for the delays users experience in DBMS, due to the fact that it is so hard to add resources to high-end DBMS implementations.

We rely on the observation that not all users (or tasks) are created equal: Some are more important than others. Everybody knows that when resources are scarce (as they are in DBMS), it is best to *prioritize* users (or tasks) and give resources to the most important users first. *Prioritization* is the most powerful tool we have to efficiently use our resources. When you have too much work on your desk, you work on the most important tasks first. When police and ambulances turn on their sirens, drivers pull over to allow them to pass through the road and intersections without delay. When people order at Starbucks coffee shops, people who order drip coffee are served immediately (because it is quick and easy), while people who order lattes and cappuccinos are forced to wait (because those drinks are time-consuming to make and require use of the limited espresso machine resource).

This thesis addresses how to share a DBMS among users with different performance requirements, and how to cope with delays. We investigate how to implement query prioritization in a DBMS to give high-priority users better query response times and less delay. Our primary goal is to provide *performance isolation* to high-priority queries, so that low-priority users do not hurt high-priority query response times, and high-priority queries run as if they were alone in the DBMS. A secondary goal is to make DBMS performance *predictable*, to reduce the frustration due to the *uncertainty* that delay causes.

Prioritization, and in particular, DBMS query prioritization, is not a new idea. DBMS query prioritization is much more difficult than prioritization in many other types of computer systems, because DBMS queries affect the performance of one another in many intricate ways, such as due to locking, data dependencies, and competition for caches. As a result, low-priority queries can cause significant delay for high-priority queries.

Existing research on DBMS query prioritization has many limitations and leaves many open questions. Most such research is hard to apply to real-world commercial systems, because it focuses on specialized "real-time" DBMS or highly-specialized or simplified workloads which are not widely used commercially. Furthermore, much of the existing research yields contradictory results, making prioritization even more difficult to use.

The key idea developed in this thesis is that to effect DBMS query prioritization, one must combine (i) performance evaluation of real-world systems with (ii) statistical analysis and queueing-theoretic modeling of system performance in order to design better algorithms. Performance evaluation is essential, as many of the important issues in prioritization stem from the specific characteristics of real-world system implementations. Likewise, analysis and modeling is essential to gain insight and understanding of the systems being considered, making algorithm design easier. Each of the main contributions of this thesis arise from this type of analysis/model/design process, and could not have been possible if any of these steps had been left out.

The main impact of this thesis is that we show how to implement DBMS query prioritization that can be used to completely eliminate the delays experienced by high-priority queries (due to low-priority queries) in real-world DBMS. High-priority queries thus see good performance isolation: predictable and low response times.

The rest of this introduction proceeds as follows.

Section 1.2 provides important background on DBMS, and the specific DBMS considered in this thesis. Section 1.3 provides important background on the workloads considered in this thesis. Section 1.4 provides important background on the DBMS query prioritization mechanism studied in this thesis. Section 1.5 discusses why DBMS are particularly challenging from a performance standpoint: it is hard to scale DBMS to reduce delays, and delays are hard to predict. Section 1.6 motivates why prioritization is an effective technique in DBMS, and discusses the impact of DBMS query prioritization. Section 1.7 outlines the scope of the thesis, describing the systems considered. Section 1.8 provides a roadmap for the remainder of the thesis, and outlines the key ideas developed throughout the research.

1.2 DBMS Fundamentals

DBMS are used everywhere. Since their introduction in the 1970's, Relational Database Management Systems (DBMS) have become the defacto standard means by which almost all data on the planet is stored and managed. Businesses use DBMS to store purchases, inventory, accounting, employee, and customer records; Schools store student records and grades; Hospitals store patient records, histories, and test results. With the advent of the Internet and the WWW, DBMS are now central to almost all e-Commerce applications. This is especially the case with online marketplaces (e.g. Amazon.COM), online customer service (e.g. Verizon Online), online communities (e.g. XBox Live), and interactive "Web 2.0" applications such as such as Facebook, Flickr, and others. Given their ubiquity, almost all of us interact with DBMS-based services on a daily basis, and thus, we all depend on their performance.

The popularity and ubiquity of DBMS is due to the fact that DBMS *enable users to share data* in a database. The key functionality DBMS take on to enable the sharing of data are: (i) manage the physical devices that store and process the data (CPUs, network, disk drives, etc), (ii) guarantee that users have reliable views of the data, by providing transactional ACID (Atomicity, Consistency, Isolation, and Durability) properties, (iii) provide powerful query languages that can perform both simple and complex tasks, and (iv)

optimize performance by choosing the correct algorithms to execute queries.

DBMS are increasingly shared. Sharing in DBMS is becoming more common in two ways.

An ever-increasing number of users are starting to use established DBMS-based services, such as online retailers. For example, online retailers typically see more customers making purchases every year. At the same time, new applications and functions are being developed that use the same DBMS used by existing DBMS-based services. For instance, an online retailer may decide to implement a view-tracking and recommendation system, which causes a completely new workload to be generated for the retailer's existing DBMS.

Sharing of DBMS is on the rise because (i) DBMS make it easy for different users to share data, and (ii) data management costs increasingly dominate the cost of storage [37, 58, 89]. As a result, companies are inclined to centralize their data storage needs into a single DBMS. It is simply not feasible to manage many independent replicas of huge data sets, especially in the face of many updates and strict data freshness and consistency requirements. As a result, applications must share the DBMS with all other applications that need the same data.

Delays in shared DBMS are large and unpredictable. When a DBMS is shared by many users, delays can cause performance can be both extremely variable and difficult to predict. Delays are caused, as in all systems, by many users trying to share limited resources. Sometimes, a query will pass through the DBMS and see no delay, and have a *good response time*, and other times, the same query will encounter significant delays, and the response time can be worse by *orders of magnitude.*

Each DBMS query needs to perform a particular computation, and requires the use of physical and logical resources (including CPU(s), I/O devices, memory, network, locks, work queues, and so forth). Often, a query will have to *wait* for a needed resource, because it is currently being used by another query (or set of queries), and has to wait in a *queue.* Queueing queries incur delays until the resource becomes available.

Almost all of the resources in a DBMS can be sources of queueing delays, but this thesis will focus on three resources which turn out to be large sources of delay, and thus important resources for prioritization in DBMS: CPU, I/O, and Locks. These resources differ in importance based on the DBMS and workload considered. This fact will be studied in depth in Chapter 2.

Delays are often more variable, harder to predict, and harder to eliminate in DBMS than in other computer systems for a number of reasons. This issue is further discussed in Section 1.5.2.

The primary reason delays are difficult to predict and eliminate in DBMS is that DBMS queries compete with each other in nearly arbitrary and complex ways. The result is that under very similar circumstances, delay can be extremely different. For example, a set of queries could run concurrently on a DBMS without ever having to queue or wait for one another. Another time, the same set of queries may run concurrently and experience unbearably long delays, slowing those queries by orders of magnitude. Which possibility occurs depends on countless factors, depending on the exact data stored in the DBMS, the exact queries being executed, the state of the DBMS resources (e.g. devices and caches), the DBMS implementation, and so on.

Some of the biggest issues are that cause such wildly unpredictable delays are that (i) DBMS queries have extremely variable resource demands: one query may need a single I/O request, and another may need gigabytes of I/O. Long running queries can sometimes get in the way of the short running queries. (ii) DBMS have numerous background tasks which cause extra work at various resources, increasing competition and

delay. (iii) Some resources, such as locks, are acquired and held for the duration of query execution, which increases contention for those resources, which increases the time other queries wait for those resources. Thus, in DBMS, *delays can be amplified*. Other examples are discussed in Section 1.5.2.

Dealing with delays in DBMS is difficult. Either eliminating or predicting the delays that occur in a DBMS are extremely difficult. This problem is discussed in detail in Section 1.5. In summary, one could decrease queueing times by either (i) adding resources to the DBMS, using faster or more hardware to increase the capacity, or (ii) re-engineer the DBMS to use existing resources more efficiently. Unfortunately, both of these approaches are difficult and costly, and suffer from diminishing returns, particularly for high-end DBMS.

State of the art for DBMS query prioritization. Despite the fact that the concept is relatively simple and has been around for decades, DBMS prioritization is sadly not widely available. When DBMS do implement prioritization, it is typically extremely limited. In particular, DBMS typically implement query prioritization only for CPU resources (and ignore I/O and Lock resources), and do not provide any choice in the scheduling policies that are used. Some commercial DBMS vendors who provide such limited prioritization include IBM (who provides db2gov and Query Patroller [20]) and Oracle (who provides Database Resource Manager [67]). Academic research has studied the problem of DBMS query prioritization, but the results are often contradictory and rely on either highly-specialized DBMS implementations or highly-simplified workloads that do not resemble the DBMS or workloads found in real-world commercial systems. The focus of this thesis is to study query prioritization in the context of real-world commercial systems.

DBMS implementations considered in this thesis. A variety of DBMS implementations are considered in this thesis, including including a commercial DBMS, open-source DBMS (PostgreSQL [52]), and a research-oriented DBMS storage manager (Shore) [17].

As indicated earlier, a key distinction between different DBMS implementations is whether CPU, I/O, or Locks (or other resources) are primarily responsible for queueing delay. Different DBMS implementations can trade off some resource usage for other resource usage. One of the ways this can be done is based on the choice of concurrency control algorithms used to provide transactional ACID semantics. There are three primary classes of concurrency control implementation: (i) Two-Phase Locking (2PL), (ii) Multi-Versioning Concurrency Control (MVCC), and (iii) Optimistic Concurrency Control (OCC).

Each of these algorithms are described in further detail in the introduction to Chapter 3. The basic idea is that 2PL always uses locks to ensure that queries see consistent data. While this implementation is conceptually simple and has little overhead, queries can hold locks for very long periods of time, and this reduces concurrency and increases delays. MVCC can eliminate many locks, but may require additional CPU and I/O resources as well as additional storage (in memory and on disk) in exchange, which can lead to increased delays as well. OCC eliminates the need for almost all locks, and merely restarts queries whenever data inconsistencies are detected during a final verification phase. OCC can be extremely inefficient due to excessive restarts, which often happens when the DBMS has too many users, but also trades off CPU and I/O for locks. Almost all commercial DBMS use either 2PL or MVCC, and OCC is generally unused in DBMS due to its performance problems.

The three DBMS implementations in this thesis: IBM DB2, PostgreSQL, and Shore, cover the two main types of concurrency control: 2PL and MVCC. IBM DB2 and Shore both use 2PL while PostgreSQL uses MVCC. It will be seen in Chapter 2 and Chapter 3 that the DBMS concurrency control algorithm will be a significant factor in understanding query prioritization, primarily due to the resource tradeoffs they make. Table 1.1 summaries the DBMS implementations and concurrency control algorithms used in the thesis.

Market	DBMS	Concurrency Control
Commercial	Brand X	2PL
Open-Source	PostgreSQL	MVCC
Research	Shore	2PL

Table 1.1: The DBMS implementations that are used in this thesis.

1.3 Workload Background

Workloads are the stream of queries that are given as input to a DBMS. Of course, every different DBMS application has its own unique workload.

To properly evaluate DBMS design decisions, one would ideally implement and evaluate that design with every possible combination of DBMS and real-world workload. While it is conceivable that one could consider the dozen or two most popular DBMS products, it is simply inconceivable to evaluate every possible real-world workload. In fact, it is nearly impossible to even evaluate with a single real-world workload. This is due to the fact that companies do not share their workloads with researchers (or other companies), so as to ensure their customers' privacy, to protect their trade secrets, and so forth.

In practice, to evaluate DBMS designs, researchers are forced to either (i) invent their own workloads, or (ii) use industry standard benchmark workloads. Both approaches have complementary strengths and weaknesses.

Inventing a workload has two main benefits: it is usually easy, and the workload can be easily parameterized to test individual DBMS features (sometimes this may be called micro-benchmarking). The main drawback of inventing a workload is that the workload has little or no relationship to actual real-world workloads. As a result, it is hard to relate conclusions about DBMS designs for the invented workload to real-world workloads.

Industry standard benchmark workloads, on the other hand, clearly relate to real-world systems, since they are typically designed with the help of DBMS vendors and consumers so as to realistically represent the workloads found in the real world. The main drawbacks for industry standard benchmark workloads are that (i) the benchmark usually calls for implementation of an extremely functional and relatively complex system with many details to worry about, and (ii) it is more difficult to correlate improvements to individual DBMS features to performance results (since the performance of the benchmark depends on the system as a whole).

The strengths and weaknesses of using either invented or industry standard benchmark workloads in DBMS performance evaluations are summarized in Table 1.2.

This thesis focuses on industry standard benchmark workloads, because existing DBMS prioritization research has generally ignored these workloads, and because prioritization depends on complex interactions between queries, which are found in such benchmarks.

Industry standard benchmarks are categorized according to different classes of workloads. These classes are chosen to be relevant to the industry, and have been identified by the DBMS community over many decades. These workload classes include (but are not limited to) OLTP (online transaction processing), transactional web, data warehousing, decision support, and ETL (Extract, Transform, and Load) workloads.

	Ease of Implementation	Real-World Relevance	Micro-Benchmarking
Industry Standard Benchmark Workload	Hard	Strong	Poor
Invented Workload	Easy	Poor	Strong

Table 1.2: The strengths and weaknesses of using either invented workloads or industry standard benchmark workloads to evaluate DBMS performance.

The primary workloads considered throughout this thesis are OLTP and transactional web (TransWeb) workloads. Workloads in these classes are found in many commercial DBMS applications, particularly in online systems and e-Commerce, and thus have wide-reaching relevancy.

Transactional web workloads are representative of the types of workloads one might expect to find in an online retailer, such as `Amazon.COM`. Transactional web workloads are categorized by having users who perform operations such as browsing a database of widgets through the web, maintain a shopping cart and user information, and purchase those widgets via a check-out process. OLTP workloads, on the other hand, are representative of the types of workloads often found in order-entry and inventory management systems, such as one might find at a Walmart or Best Buy. OLTP workloads are categorized by users who want to enter and deliver orders, record payments, check the status of orders, and monitor stock levels.

To evaluate OLTP and transactional web workloads, this thesis relies on industry-standard DBMS benchmarks designed by the Transaction Processing Performance Council (TPC) [21], a non-profit corporation comprised of DBMS and hardware companies that are interested in categorizing the performance of real-world DBMS applications. Members of the TPC include companies such as Microsoft, IBM, Oracle, Sybase, Teradata, Ingres, Sun, HP, Intel, Dell and AMD.

The key advantage of TPC benchmark workloads is that they are extremely realistic. TPC benchmarks require the implementation of a fully-functional DBMS application, whose user interactions and database schema and structure are representative of real-world workloads. Specifically, the workloads used in this research are based on the TPC benchmarks TPC-C [22] and TPC-W [23]. TPC-C is an OLTP benchmark consisting of a fully functional retail inventory management system. TPC-W is a transactional web benchmark that consists of a fully functional online book store. Due to the fact that TPC-W and TPC-C model a real-world application, and include relatively complex queries and schemas, they are much more effective for understanding real-world system performance than simple micro-benchmarks used in many other DBMS performance studies.

Both TPC-W and TPC-C are both transactional workloads that read and write data in the DBMS, and are representative of the types of workloads found in many retail environments. The key distinction is that TPC-W models systems that are designed for online retailers, which must handle large numbers of concurrent users, while TPC-C models systems that are designed for smaller "in house" systems typically used by store clerks and managers. The ramification of this is that TPC-C has more complex and a greater variety of queries than TPC-W, and TPC-C has stronger consistency requirements than TPC-W.

TPC-W is described further in Appendix A.1 and TPC-C is described further in Appendix A.2.

	Ease of Implementation	Performance Isolation
Admission Control	Stronger	Weaker
Internal Prioritization	Weaker	Stronger

Table 1.3: The strengths and weaknesses of using either admission control or internal device and resource prioritization to effect query prioritization in DBMS.

1.4 Prioritization Mechanism Background

There are two common mechanisms used to implement query prioritization in DBMS: (i) Admission control and (ii) Internal device and resource prioritization. The study of both of these mechanisms is central to this thesis.

Admission control limits the number of users in the DBMS concurrently. The MultiProgramming Level (MPL) is the maximum number of concurrent users that admission control will allow into the DBMS. When there are additional queries, they will be queued outside the DBMS[1], and will be admitted when the currently executing queries are completed. Admission control can be applied to all of the queries in a system, or only certain classes of queries.

Internal prioritization of devices and resources in a DBMS reorders the execution of queries within the DBMS execution engine. Without internal prioritization, when a resource (such as CPU, I/O, or a Lock) is needed by multiple queries, the DBMS will allocate that resource on a first-come, first-serve (FCFS) basis, without respect to the relative importance of each query. As a result, all queries are likely to wait to acquire essential system resources. Internal prioritization attempts to reduce high-priority query delays by ensuring that high-priority queries are given resources first.

Admission control and internal prioritization provide complementary strengths and weaknesses, making them both ideal for study in this research. (i) Admission control's primary strength is that it is *easy to implement*, but its primary weakness is that it has a *limited effect* on providing performance isolation to queries. Admission control can be used even when the DBMS has not been designed to handle it, via a simple external implementation. Once a query is admitted to the DBMS, however, it runs freely, and can interfere with all other queries in the system. (ii) Internal prioritization's strength, on the other hand, is that it can be *very effective* at providing performance isolation to queries. Unfortunately, its main weakness is that it is *difficult to implement*, requiring that the DBMS be rewritten and engineered to explicitly support it. As a result, internal prioritization cannot be used with applications that rely on legacy DBMS implementations. Furthermore, most commercial DBMS provide either limited or no support for internal prioritization, and users cannot make full use of internal prioritization until DBMS vendors provide more complete implementations. These strengths and weaknesses are summarized in Table 1.3.

There are many open questions and unresolved issues involving query prioritization using either admission control or internal prioritization. (i) Admission Control: While admission control is widely available, it is not well-understood how to tune and configure admission control to achieve desired performance isolation. It is not even clear that admission control can always be used to meet a given set of performance goals. (ii) Internal Prioritization: Internal prioritization is found in a few commercial DBMS, but those

[1] Other variants of admission control may drop queries, but those are not appropriate for DBMS, since they keep essential work from being done. I only consider non-dropping admission control.

implementations are fairly limited. Commercial DBMS typically only allow some (but not all) devices and resources to be scheduled, and there is no control over the scheduling policies used on each resource. It is not well-understood which devices and resources are most important for a DBMS to schedule to effect query prioritization. While some commercial DBMS can schedule some resources (such as CPU), it is unclear whether scheduling of those resources alone can provide sufficient performance isolation. Furthermore, it is not clear what scheduling policies are most effective for each device and resource in the system.

1.5 Difficulties in Managing DBMS Delays

Managing performance and delays in DBMS is more challenging than doing so in many other computer systems, for two primary reasons: (i) it is difficult and costly to scale DBMS and add capacity in order to reduce or eliminate user delays, and (ii) it is difficult to *model and predict* the performance and delays that users sharing a DBMS will experience. These reasons are addressed respectively in Section 1.5.1 and Section 1.5.2 below.

1.5.1 Scaling up DBMS to eliminate delays is hard

DBMS are essentially the primary tool that enables users to share data, and users who want that data must access it through the DBMS. As indicated in Section 1.1, when users are forced to share a limited resource, they incur delays, which cost time, money, and frustration. DBMS, being a limited resource, are stuck with performance problems due to delays.

One may ask why a DBMS is a *limited* resource if we can always build a bigger or more powerful DBMS to reduce delays. Computer science provides three typical approaches to improve system capacity and performance: **Scale Up**, **Scale Out**, and **Improve Efficiency**. While these approaches are often easy to apply to many computer systems, they are much harder to apply to DBMS. The DBMS industry often focuses on improving DBMS performance via these approaches, but the process is extraordinarily difficult and expensive.

- **Scale Up** involves running a centralized DBMS on bigger and faster high-end computer hardware. The main drawback to this approach is its cost. High-end hardware is much more expensive, and has higher price/performance ratios than lower-end hardware. High-end hardware is more expensive because there are fewer customers for such hardware (thus economies of scale do not kick in) and because those customers' businesses depend critically on that hardware to succeed.

  Even worse, high-end DBMS push existing hardware to its limits and beyond, and newer, more complex systems and technologies must be built specifically to handle DBMS needs. Companies' database requirements are growing extremely rapidly: databases double in size every 12-18 months, query rates are increasing dramatically, and data freshness requirements are getting stronger [9]. Hardware developers can barely keep up with this growth.

- **Scale Out** involves purchasing many lower-end (or commodity) systems, and networking them together to create a single *distributed* DBMS, with independent processing nodes. Using commodity hardware improves the price/performance ratio drastically, making it appear more attractive than Scale Up. The main drawback, however, is that such systems can be more costly to manage and maintain than centralized DBMS.

Scale Out for DBMS has been investigated with research systems such as R* [39], Gamma [27], and Bubba [13]. Commercial DBMS also typically provide limited distributed functionality, such as DB2 DPF and Oracle RAC.

Making DBMS Scale Out efficiently, however, is extremely difficult. Many fundamentally difficult problems must be solved, related to data freshness and consistency, data partitioning, load balancing, data shipping, and query shipping. Often, the overheads due to these problems can diminish the speedup provided by Scale Out.

- **Improve Efficiency** involves making the DBMS to better utilize the limited resources that it has available. Improving efficiency in DBMS is extremely difficult, as it requires significant investments of time, money, and effort to profile, analyze, and re-engineer, the DBMS to address the underlying performance issues. DBMS are complex systems, and performance problems can arise in almost any system component, ranging from poor algorithm choices, query plans, cache policies, and so forth. Even at its best, however, improving efficiency cannot eliminate performance delays, and can only reduce them.

Finally, even if it was relatively easy to add capacity to a DBMS, delays would only be improved temporarily. In any system, spare capacity gradually disappears as the system is always used to do far more. All personal computer users are intimately aware of this fact — the computers we use today are many times faster than the computers we used just a few years ago, but the perceived performance improvement is negligible, since we run far more complex software programs on them. As Professor William Wulf has said:

> Although the hardware costs will continue to fall dramatically and machine speeds will increase equally dramatically, we must assume that our aspirations will rise even more. Because of this, we are not about to face either a cycle or memory surplus. For the near-term future, the dominant effect will not be machine cost or speed alone, but rather a continuing attempt to increase the return from a finite resource — that is, a particular computer at our disposal. [88]

1.5.2 Analyzing and predicting DBMS performance is hard

Delay prediction is important. Not only is it difficult to scale DBMS capacity so as to eliminate user delays, it is also extremely difficult to *predict* the magnitude of delays that queries will experience in a DBMS. Prediction of DBMS performance is important for three major reasons. First, prediction is necessary to ensure that DBMS users will (continue to) receive good performance when an additional query workload is to be given to a DBMS. Such circumstances arise when adding functionality to an existing DBMS-based service. Second, prediction is necessary when deciding what type of hardware must be purchased so as to ensure that queries receive sufficient performance (or to sufficiently reduce delays). Third, prediction is necessary to help decide how to configure essential DBMS configuration parameters that affect query performance and delays.

Another important issue is that even if we cannot do anything to address delays in DBMS, if we can accurately predict DBMS performance, we can provide users with *a warning* when they are likely to receive poor performance, or *an estimate* of how long they will have to wait. This can reduce the frustration that users experience even when delays are large.

Predicting delay in DBMS is difficult. Predicting performance in DBMS, unfortunately, is much harder than predicting performance in many other computer systems. This difficulty is primarily due to the

fact that delays in DBMS are often more variable, larger, harder to predict, and harder to eliminate than in other computer systems. The following are several important reasons that this is the case:

(i) DBMS queries, even those in the same application workload, have variable resource demands (requirements). Some queries may require only a millisecond of CPU time to complete, while others may require hours. Likewise, some queries may require no I/O, others issue only one I/O request, while others yet issue hundreds of megabytes or gigabytes of I/O. Queries can easily get "stuck" waiting behind slow operations and computations, resulting in excessive delay.

(ii) DBMS have numerous background tasks that run periodically to improve performance, or perform maintenance. For example, DBMS often use special processes to scan the DBMS buffer pool for dirty pages and write them back to disk. These processes introduce additional work for certain resources sporadically, which can periodically increase delay.

(iii) Some resources, such as locks, are held for long periods of time, and cannot be used by other queries during that time. This is in contrast to other systems, where resources such as locks are usually held for only short periods of time. For instance, (in DBMS using two-phase locking (2PL) for concurrency control) a query can hold locks for (almost) its entire execution, including the time that it spends waiting for other resources. Thus, *in a DBMS, delays can be amplified*: When a query waits for a resource, it holds other resources longer, increasing contention on those resources, increasing the time other queries wait for resources, and so on.

(iv) Some resources maintain significant amounts of state which affect the delay incurred at those resources. When one query uses such a resource, the resource's state changes, resulting in poor performance for other queries that need that resource. For example, a disk drive has a physical head that moves very slowly, and if one query forces it to move, then not only does that query wait for the head to move, future queries will need to wait for the head to move again. Likewise, a CPU has an on-chip cache, that is used to improve instruction throughput. When executing on the CPU, one query can cause much data to be evicted from the cache, which causes future queries to experience delay when reading data from main memory.

Queueing theory is often not applicable. Even queueing theory, which has been extremely effective in analyzing and predicting the performance and delays in many other computer systems, is difficult to apply to DBMS. Researchers such as Kleinrock, Erlang, and Jackson [63] developed methods of modeling computer systems as networks of queues and servers through which jobs (e.g. queries) flow. Countless analysis methods, such as Matrix Analytic Methods, Mean Value Analysis, and Heavy Traffic Analysis, have been developed to predict important performance statistics (e.g. mean response time) from these models. Queueing theoretic tools have been tremendously effective at helping practitioners design better computer systems, including telephone systems [29], network switches [25], the ARPANET [50, 51], web servers [38], and countless others.

Unfortunately, the fundamental design, architecture, and operation of DBMS makes much of queueing theory hard to apply. The central problem is that queueing theory makes many critical assumptions to make analysis tractable, which are often violated in DBMS. Some of the most important assumptions which DBMS violate are that queueing theory usually assumes that systems are *work conserving*, that job/query sizes are **exponential**, and that job/query sizes are independently and identically distributed (**IID job sizes**). DBMS violate each of these assumptions as described below:

- **Work conserving:** Jobs/queries in queueing systems have a predetermined and fixed amount of work (their "size" or "service time") that needs to be done in the system. For instance, when a job arrives at the system, one can determine that it needs, say, 5 seconds of CPU service time.

Of all the assumptions violated by DBMS, work conservation is the most significant. The amount of work needed by a given DBMS query may be completely different, depending on the state of the DBMS and depending on what other queries are executing during the time the query executes. As an example, two DBMS components which cause the DBMS to violate work conservation include DBMS caches and locking:

Caches. DBMS make extensive use of caches, either in the DBMS buffer pool, the CPU architecture, or the disk drives. Caches can radically change the amount of work that needs to be done for a given query, depending on whether the data it needs is present in the cache. Furthermore, the DBMS buffer pool can actually create **extra** work for a query, since the buffer pool is typically not write-through, meaning that (usually) dirty pages have to be written back to disk on evictions.

Locking. DBMS use locking to ensure that all queries see consistent views of data. Locking can force one query to stop mid-execution, to wait for another query to finish executing before it can proceed. The amount of time spent waiting for locks is variable, and depends on many factors, including the number of queries in the DBMS, the data that each query accesses, and whether that data is being read or written.

- **Exponential job sizes:** Queueing systems typically assume that query/job sizes are distributed according to an exponential distribution. The primary consequence of this is that queries are *memoryless*, and at any point in time in their execution, a query is equally likely on finishing, or it has "constant failure rate." This assumption makes it easy to analyze the queueing model using Markov chains. While some approaches, such as hyper-exponential distributions, can be used to model other distributions, these approaches often cause state-space growth in the Markov chains, which can also make analysis intractable.

  In DBMS, jobs are almost never exponentially distributed, and are often much less so than in other systems. A query which has just begun typically has a "low failure rate," and is unlikely to be finished. Typically, a query exhibits a "high failure rate" once it has been executing long enough. Furthermore, the distribution of DBMS queries often exhibit a "heavy tail," in which case a few queries are extremely long-running. In this case, once a query has executed long enough, it once again exhibits a low failure rate.

- **IID job sizes:** Independence of job sizes is almost always needed during queueing analysis. Typically, independence allows one to determine time-average behavior of the system as a whole, since the distribution of work in the system is *stationary* (in a probabilistic sense). The work needed by a DBMS queries, however, is often not independent of the work needed by another.

  The same factors that cause DBMS to violate work conservation assumptions (described above) cause one query to change the DBMS state, such that the amount of work needed by another query changes. For example, one query may evict critical data from the buffer pool, forcing another query to do extra I/Os to get that data back from disk.

  Additionally, DBMS queries themselves are often not independent, since the users who issue the queries are attempting to complete a very specific task. In e-Commerce, for instance, a user will issue several browsing-related queries, to find a book, before issuing a query to add the book to their shopping cart, which will happen many times before issuing a query to check out and complete their purchase.

The fact that DBMS violate many of the assumptions in queueing theory cause nearly all DBMS researchers significant difficulties. It is important that this does not preclude the use of queueing theory to

understand DBMS performance. Many researchers address this problem by analyzing models of extremely simplified DBMS implementations or workloads. By ignoring many important system details, such as work conservation, locking, and inter-query dependencies, the analysis becomes tractable. Unfortunately, the performance results may not relate to the performance of real DBMS, in particular, when the complicating details that must be ignored for tractability may also greatly affect system performance.

This thesis takes approaches that are commonly used by other DBMS researchers to address this problem: to augment simplified DBMS performance models to incorporate the critical factors that affect performance, with a focus on specific situations. A key difference in our research is that we focus on particularly complex query workloads (OLTP and Transactional Web workloads) running on standard commercial DBMS. As a result, we must address different types of performance issues than are typically considered in preexisting research. In order to make sure that our approach correctly predicts DBMS performance, our approach depends on both (i) adapting queueing models and analysis techniques to accommodate DBMS when they violate standard queueing theoretic assumptions, and (ii) verifying, via performance evaluation studies of fully-implemented systems, that these models are accurate.

1.6 Impact of Prioritization

As indicated in Section 1.1, prioritization is important whenever users are forced to share and both (i) *resources are limited* and (ii) *some users are more important than others*. DBMS are textbook examples of systems in which prioritization is important: (i) resources are limited since it is difficult to scale DBMS resources, and (ii) some DBMS users *are* often more important, need better response times, and need better performance isolation than other users.

Since the issue of limited resources in DBMS was discussed in Section 1.5, this section focuses on the issue of differentiating high- and low-priority users in real-world scenarios where users are forced to share DBMS.

In the simplest of situations, some classes of users demand that prioritization be applied so as to improve their performance, and minimize their delays:

- **Service Level Agreements (SLAs) for users.** Some customers may be high-priority simply because they pay for the privilege of receiving better performance. A frequent customer of an online retailer may realize that it would save a significant amount of time and frustration to pay for a subscription "gold service" membership, that gives them better performance.

Sometimes, companies have more indirect means of employing query prioritization, which helps make more profits by increasing functionality:

- **Customers who are likely to buy more.** An online book retailer receives a steady stream of users, who generate queries to browse the book collection, add books to their shopping cart, checkout, and so forth.

  The retailer may create a High- and Low-Priority class in a number of ways: (a) Regular customers are high-priority. (b) Customers with a history of buying profitable "big ticket" items are high-priority. (c) Customers who have profitable items in their shopping cart are high-priority. (d) Customers that

have been receiving slower-than-average response times are high priority (to make up for prior poor performance).

- **Additional Functionality.** It is extremely common, particularly in DBMS-driven e-Commerce applications, to periodically add additional functionality and services to existing systems. Existing DBMS query workloads are often more important than new functionality, because more customers depend on the existing functionality. Thus, existing functionality may be high-priority, while new functionality may be low-priority.

- **Federated DBMS and MultiDatabases.** Relatively new to the DBMS market are tools called Federated DBMS, or MultiDatabases, which provides a unified logical view of many different and separate DBMS [65]. The *Federator* is told about the existence of other DBMS, the set of which makes up a *Federation* of DBMS. A user sends a query, which refers to any subset of data within the Federation, to the Federator. The Federator automatically issues queries to the Federation DBMS and unifies the results to answer the user's query.

  In a typical Federated DBMS deployment, each DBMS in the Federation is "legacy" and is dedicated to a particular application. The Federator is brought in only to answer decision-support and similar questions that needs information from both DBMS. The Federator is needed because it is too costly, politically infeasible (e.g. security and data ownership issues), or or technologically infeasible (e.g. each department uses a different brand of DBMS, and relies on their proprietary features) to unify the databases into a single DBMS.

  Sometimes, in a Federated DBMS, each DBMS is dedicated to a particular application, which is of utmost importance, and the Federator queries are essential, but have weaker performance requirements. Thus, each DBMS's dedicated Local workload is high-priority, while Federator queries are low-priority. (These priorities will be used throughout this thesis for Local and Federator workloads.)

Sometimes, careful use of query prioritization can be used to benefit *all users*, rather than just a select few high-priority users:

- **SJF and mean response time improvement.** Many DBMS have workloads with extremely variable query sizes, where most queries are very short, but a few have very long running times. It is a well-known result from queueing theory that in systems with variable job sizes, giving shorter queries higher priority than longer queries improves the average response time.

  In DBMS, a query can often be identified as short- or long-running before execution even begins, simply by looking at the query text or query plan. DBMS may be able to mark short-running queries as high-priority, effectively mimicking the behavior of the effective Shortest Job First (SJF) scheduling policy, and minimizing overall average query response time.

- **Fairer performance.** On first glance, prioritization is meant to provide performance *inequity*, but it can also provide users with performance *equity*.

  With or without prioritization, DBMS queries often have variable response times, because they experience delays caused by other users' queries, or other performance artifacts. When users often revisit the DBMS, some users may experience regularly poor performance, simply by getting "unlucky." Prioritization can be applied to improve the future performance of these unlucky users, to make up for the poor prior performance. Thus (at least in the long term), all users are more likely to see less variable average response times.

High-Priority Workload	Low-Priority Workload	Shared Database	Prioritization Mechanism
OLTP TPC-C	OLTP TPC-C	Yes	Internal Prioritization
TransWeb TPC-W	TransWeb TPC-W	Yes	Internal Prioritization
TransWeb TPC-W	TransWeb TPC-W	No	Admission Control

Table 1.4: A summary of the shared DBMS system configurations that are considered in this thesis research.

1.7 Scope

The work in this thesis does not cover all forms of sharing and query prioritization in DBMS. The research is scoped by the workloads considered, the query prioritization mechanisms considered, and the DBMS system configurations considered.

Workloads. As described in Section 1.3, this research focuses on two main workloads, based on TPC benchmark specifications: (i) OLTP workloads, and in particular, the TPC-C benchmark, and (ii) Transactional Web workloads, and in particular, the TPC-W benchmark.

Prioritization Mechanisms. As described in Section 1.4, this research only considers two query prioritization mechanisms in DBMS: (i) internal prioritization of DBMS resources and devices, and (ii) external scheduling via the use of admission control to limit the number of queries from a given user allowed into the DBMS at any time.

System Configurations. Three specific system configurations will be considered in this work, each consisting of a high-priority and a low-priority workload that share a DBMS. The systems differ based on (i) the particular high- and low-priority workloads, (ii) the prioritization mechanism used to give the high-priority workload performance isolation, and (iii) whether the two workloads share the same database data or use different databases stored on the same DBMS. In each case, the goal is to ensure that the high-priority workload gets good performance isolation and good response times (despite the presence of low-priority queries in the DBMS), and to give low-priority queries get "best-effort" service, but that low-priority queries must not starve. These system configurations are described below and summarized in Table 1.4.

The first system configuration is depicted in Figure 1.1, and consists of a single DBMS shared by two OLTP TPC-C workloads: one high-priority, and the other low-priority. Each of these workloads share the same database, as if each workload is sharing and accessing the same company and inventory data. Internal prioritization is used on resources and devices within the DBMS to ensure that high-priority queries receive performance isolation.

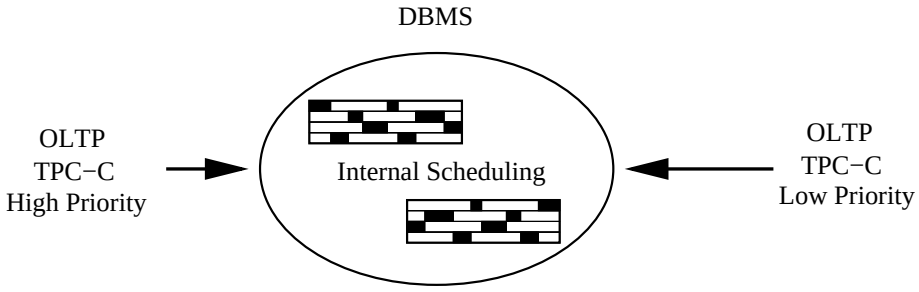


Figure 1.1: The first system configuration, comprised of two OLTP workloads sharing a DBMS that uses internal prioritization to prioritize high-priority users.

The second system configuration is depicted in Figure 1.2, and consists of a single DBMS shared by two OLTP TPC-W workloads: one high-priority, and the other low-priority. Each of these workloads share the same database, as if each workload is sharing and accessing the same company and inventory data. Internal prioritization is used on resources and devices within the DBMS to ensure that high-priority queries receive performance isolation.

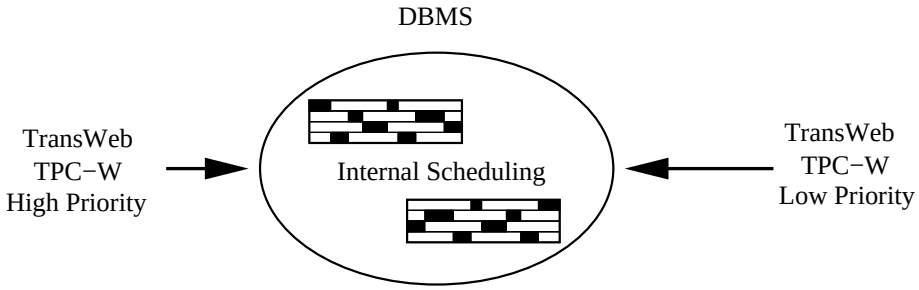


Figure 1.2: The second system configuration, comprised of two Transactional Web workloads sharing a DBMS that uses internal prioritization to prioritize high-priority users.

The third and final system configuration is depicted in Figure 1.3, and consists of a single DBMS shared by two different Transactional Web workloads: one high-priority, and the other low-priority. Unlike the other system configurations, the two workloads *do not* share the same database data. Admission control is used to limit the low-priority MultiProgramming Level (MPL), defined as the maximum number of low-priority queries in the DBMS at any one time. High-priority queries are always admitted into the DBMS and allowed to run.

Hi-Prio Wkld	Low-Prio Wkld	Prioritization Mechanism	Chapters
OLTP TPC-C	OLTP TPC-C	Internal Prioritization	Chapter 2 and Chapter 3
TransWeb TPC-W	TransWeb TPC-W	Admission Control	Chapter 2 and Chapter 4

Table 1.5: Summary of workload scenarios considered in the chapters of this thesis.

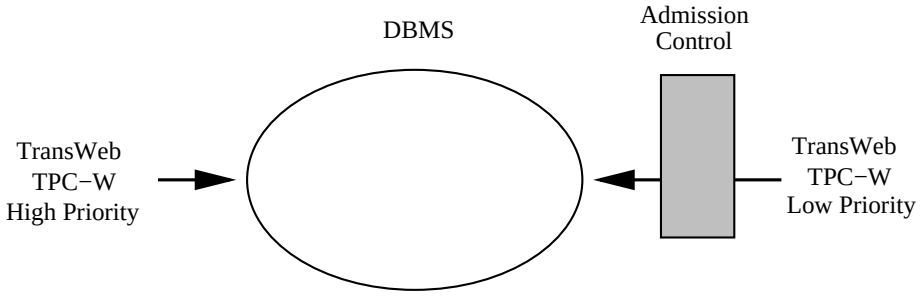


Figure 1.3: The third system configuration, comprised of two Transactional Web workloads sharing a DBMS that uses admission control to isolate high-priority users from low-priority users.

1.8 Roadmap

This thesis is organized in three parts, discussed in Chapter 2, Chapter 3, and Chapter 4. Chapter 2 and Chapter 3 will focus on how to provide a high-priority query class using internal prioritization for both OLTP and transactional web applications that share a DBMS. Chapter 4 changes direction and addresses how to share a DBMS between two transactional web applications in a single DBMS, using admission control to provide performance isolation.

Chapter 2 presents a comprehensive study of prioritization for OLTP and transactional web applications on several types of DBMS using different concurrency control methods. Chapter 2 reveals limitations for existing Lock scheduling approaches for providing prioritization for OLTP workloads in certain DBMS implementations. Chapter 3 addresses these limitations in Lock scheduling, and documents an in-depth statistical analysis of locking in OLTP workloads, and develops a new Lock scheduling algorithm called Preempt-On-Wait (POW) that outperforms existing algorithms.

Chapter 4 examines a system in which a DBMS is shared by two transactional web workloads, in which one workload is regulated by admission control. Chapter 4 consists of a comprehensive study of each workload's performance as the admission control MPL and the workload characteristics are varied. Furthermore, it demonstrates how to apply queueing analysis to predict the performance of the DBMS, and shows how to adjust the queueing analysis to reflect the difficulties predicting DBMS performance as described in Section 1.5.

1.8.1 Chapter 2: Prioritization in OLTP and Transactional Web Applications

The goal of Chapter 2 is to determine how to use internal prioritization to provide performance isolation for high-priority queries in a DBMS shared by either OLTP or Transactional Web workloads. Chapter 2 consists of two halves: (i) a comprehensive bottleneck analysis of the workloads and DBMS implementations, to determine which resources are most important to schedule to achieve query prioritization, and (ii) an evaluation study of various scheduling algorithms for different resources on each of the DBMS and workloads considered.

The primary questions addressed in Chapter 2 are:

(i) *What resources/devices should be scheduling be applied to?*

(ii) *What scheduling algorithms should be used?*

(iii) *How does the workload affect questions (i) and (ii)?*

(iv) *How does the DBMS implementation affect questions (i) and (ii)?*

By helping to answer the above questions, Chapter 2 helps to solve several important problems that arise in DBMS.

Implementing resource scheduling inside a DBMS can be a difficult task. DBMS are comprised of countless resources, some physical (e.g. I/O, CPU), and some virtual (e.g. Locks, Latches), and it is both difficult and time consuming to implement priority scheduling at each resource. In fact, it can actually hurt performance more than it helps to implement scheduling at unnecessary resources, due to the overhead involved (such as to manage priority queues). Chapter 2 helps developers to focus on implementing scheduling only on resources that are important from a performance standpoint. As a result, development costs are saved, bugs are reduced, and overheads are minimized.

Furthermore, different scheduling algorithms are more difficult to implement than others. For instance, it is more difficult to implement a lock scheduling algorithm with priority inheritance. Priority inheritance requires more book-keeping, but can help reduce the performance delays experienced by high-priority queries. Without a quantifiable understanding of how these algorithms improve performance, it is difficult to determine whether they are worth implementing.

The workloads considered throughout Chapter 2 will be the OLTP workload TPC-C and the transactional web workload TPC-W, which are discussed in further detail in Section 1.3. Chapter 2 will consider several DBMS implementations, including the commercial IBM DB2, the open-source PostgreSQL, and the research-oriented Shore storage manager, each of which are described in detail in Section 1.2.

Bottleneck Analysis

The bottleneck analysis consists of determining how much time the *average query* spends in each resource in the DBMS. The resource in which queries spend the majority of their time is *the bottleneck resource.* It is a well-known from queueing theory that query performance at the bottleneck resource dominates the performance of queries in the system as a whole (assuming that queries visit all system resources).

I consider that the DBMS is comprised only of CPU, I/O and Lock resources. It will be seen that these resources are the only ones of any relevance in the systems covered in this thesis, although other resources

could easily be considered as well. The primary measurements considered are the time that an average query spends executing at and waiting for the CPU(s) and I/O device(s), and the time waiting for Locks (including locks managed by the DBMS lock manager to ensure ACID properties are satisfied, but not spin locks/latches used to maintain internal DBMS data structures). The sum of these times is equal to the average query response time (the total time a query is in the DBMS).

I will show that for TPC-C workloads running against either commercial or research DBMS using 2PL concurrency control, Locks are almost always the bottleneck. This is a surprising fact, that queries (on average) spend the most time waiting for locks, despite the fact that other resources in the system may have high utilization (e.g. CPU). TPC-C workloads running against MVCC DBMS (in particular, PostgreSQL) will prove to be I/O-bottlenecked in most cases, although locks can become significant bottlenecks when concurrency is increased. TPC-W workloads will prove to be CPU-bound across the board.

Scheduling Algorithm Evaluation

For the scheduling algorithm evaluation, several scheduling algorithms are implemented in PostgreSQL (which uses MVCC) and Shore (which uses 2PL), and TPC-C and TPC-W are ran against these implementations. Prioritization is determined randomly: 10% of the queries are chosen to be high-priority, and the remainder are low-priority.

Given that the bottleneck analysis reveals that CPU and Locks are bottleneck resources, I only consider prioritization algorithms for these resources. In particular, the prioritization algorithms considered are:

1. `NP-LQ` — Lock prioritization, which non-preemptively prioritizes locks by reordering lock queues.
2. `NP-LQ-Inherit` — Lock prioritization, identical to NP-LQ, but also adds *priority inheritance.* Priority inheritance boosts the priority of low-priority queries whenever a high-priority query must wait (due to a lock dependency) on for them. The idea is to get low-priority that block high-priority queries out of the system faster, to let the high-priority queries proceed.
3. `P-LQ` — Lock prioritization, identical to NP-LQ, but adds *preemption.* Preemption lets high-priority queries kill low-priority queries that hold locks needed by the high-priority query (and force the high-priority query to wait). Killed low-priority queries are "rolled back" and restarted after the high-priority query is given the lock it needs.
4. `CPU-Prio` — CPU scheduling, which changes the UNIX-like priority of the operating system scheduler.
5. `CPU-Prio-Inherit` — CPU scheduling, identical to CPU-Prio, except that priority inheritance is added.
6. `P-CPU` — CPU (and Lock) scheduling, identical to CPU-Prio, but also kills (and restarts) low-priority queries when they prevent high-priority queries from acquiring locks.

I find that implementing scheduling precisely on the bottleneck resources has the strongest ability to provide performance isolation to high-priority queries. This is counter to the widely-believed idea that simply scheduling the CPU is enough to (indirectly) schedule other resources [2]. Furthermore, I find that priority inheritance is most effective on CPU-scheduling in CPU-bound workloads, such as TPC-W running

on PostgreSQL, in which `CPU-Prio-Inherit` is a factor of 3 better at isolating high-priority queries than `CPU-Prio`. For Lock-bound workloads, using lock scheduling with priority inheritance is still helpful, but, for instance, `NP-LQ-Inherit`'s isolating power is only 30% better than that of `NP-LQ`.

The most striking discovery is that for lock-bound workloads (OLTP TPC-C workloads running against DBMS using 2PL concurrency control), it is difficult to provide performance isolation to the high-priority queries.

Depending on the situation, either (i) the isolation that preemption gives to high-priority queries is too small (resulting in excessively large high-priority query response times), (ii) the isolation that preemption gives to high-priority queries is good, but the penalty caused to low-priority queries is too great (resulting in starvation of low-priority queries). Thus, it appears as if preemption is not very effective when used in lock scheduling. Chapter 3 of this thesis, however, addresses this problem, and shows how to make preemptive lock scheduling effective, without hurting low-priority queries too much.

1.8.2 Chapter 3: Lock Prioritization in OLTP Applications with POW

Chapter 3 is a continuation of the work done in Chapter 2, but focuses on improving the limitations observed in existing lock scheduling policies. Chapter 3 considers only the lock-bound workloads observed in Chapter 2: OLTP TPC-C workloads running against a DBMS using 2PL concurrency control.

The goal of Chapter 3 is to determine how to use preemption to improve lock scheduling to provide performance isolation for high-priority queries in OLTP TPC-C workloads. In particular, Chapter 3 tries to resolve the trade-off seen between preemptive and non-preemptive lock scheduling algorithms. Typical preemptive lock scheduling is good at improving high-priority query response times (more than non-preemptive lock scheduling), but effectively starves low-priority queries, resulting in extremely long low-priority query response times. On the other hand, non-preemptive lock scheduling is not as good at improving high-priority query response times as preemptive lock scheduling, but it does not starve low-priority queries. Chapter 3 attempts to get the best of both worlds: good high-priority query performance without starving low-priority queries.

Chapter 3 is broken into two halves: (i) a statistical analysis of the locking behavior observed in the systems under study with and without lock scheduling, and (ii) the development of a new algorithm, called Preempt-On-Wait, which selectively preempts low-priority queries to deliver better performance isolation to high-priority queries.

The primary questions addressed in Chapter 3 are:

(i) *What factors cause preemptive lock scheduling to starve low-priority queries?*

(ii) *What factors prevent non-preemptive lock scheduling from providing sufficient performance isolation to high-priority queries?*

(iii) *Can the use of selective preemption give good performance isolation to high-priority queries without starving low-priority queries?*

(iv) *What condition(s) should be used to decide when to preempt low-priority queries?*

The work in Chapter 3 is, in some ways, similar to the work done by Singhal and Smith [76], in which

several real-world DBMS workload implementations are analyzed to statistically characterize their performance and system behavior.

Statistical Analysis

The first half of Chapter 3 examines statistical properties about how queries hold and compete for locks in the DBMS. The first half also examines how common preemptive and non-preemptive lock scheduling policies affect these statistics, revealing why these policies are not as effective as desired. These statistics are collected by instrumenting the lock manager and tracing the execution of the TPC-C workload running against Shore, and verified (when possible) with IBM DB2.

The key observation made in this half of the work is that, most of the time, when a high-priority query is forced to wait (due to a lock conflict) behind a low-priority query, it does not wait very long. This is because the low-priority query typically finishes without having to wait for any locks (and since locks are the bottleneck resource, they are predominantly responsible for slow response times). Statistically speaking, the only time low-priority queries hurt high-priority queries is when those low-priority queries must wait to acquire another lock.

Preempt-On-Wait (POW)

The second half of Chapter 3 uses the statistical properties discovered in the first half to develop a new lock scheduling algorithm, called Preempt-On-Wait (POW). Furthermore, I evaluate the performance of POW, and show that it achieves the best-of-both worlds: (i) High-priority queries get the good performance isolation (and small response times) as found in preemptive lock scheduling, and (ii) Low-priority queries get good performance and do not starve, as found in non-preemptive lock scheduling.

The key idea for POW is to preempt low-priority queries *selectively*, and only when they most hurt high-priority query response times. POW preempts all low-priority queries which are both (i) being waited on by a high-priority query, and (ii) waiting to acquire another lock. The idea for this criteria is that when a low-priority query waits for a lock, it is likely to have a long response time, and the remaining time of the query is also long. Thus, a high-priority query that must wait on such a low-priority query will also have a long response time. This is due to the fact that the high-priority query has to wait for the low-priority query to finish before it can continue.

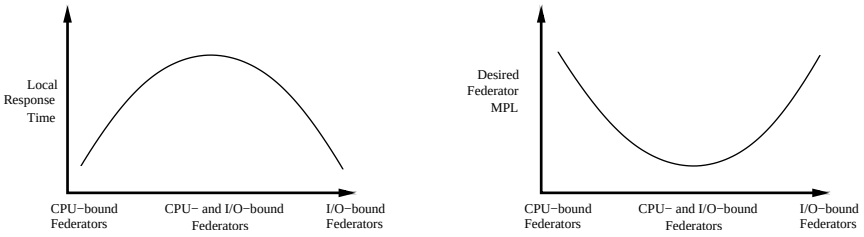
1.8.3 Chapter 4: Providing Isolation for Mixed DBMS Workloads (IDD)

Chapter 4 takes a turn away from the systems covered in Chapter 2 and Chapter 3. Instead of using internal prioritization, Chapter 4 considers how to configure admission control to provide performance isolation between two workloads running on the same DBMS.

To make the work tractable, Chapter 4 does not consider all possible combinations of workloads and DBMS. Instead, it focuses on systems with two different transactional web (TPC-W) workloads running against IBM DB2. The first workload will be CPU-bound, and called the **Local** workload (alternatively, the Locals). The second workload will vary from CPU-bound to I/O-bound by increasing the database size, and will be called the **Federator** workload (alternatively, the Federators). Throughout Chapter 4, the

Local workload will be high-priority, and the Federator workload will be low-priority, needing only best-effort service. Despite these DBMS and workload restrictions, none of the methods developed in Chapter 4 depend on them, and the methods are expected to be effective and accurate in a wide range of systems.

Chapter 4 has three major sections: (i) The discovery of *The Hump*, which is a surprising and non-intuitive performance trend found when mixing multiple workloads on the same DBMS, and (ii) A demonstration that existing queueing theoretic analysis does not predict the hump, and (iii) Introduction of a new queueing analysis technique, called *Isolated Demand Decomposition (IDD)*, which correctly predicts the performance of two workloads running on the same DBMS, especially as a function of admission control, based on how those workloads run on the DBMS by themselves (in isolation).



(a) The Hump (b) Federator MPL Policy

Figure 1.4: **Left:** Illustration of The Hump response time trend. Locals response times as a function of the I/O-boundedness of the Federator workload. Counter to intuition, response times for the CPU-bound Locals are good both when Federators are CPU-bound or Federators are I/O-bound. When Federators have simultaneously large CPU- and I/O-demands, Local response times are bad. **Right:** Illustration of the ideal Federator MPL policy as a function of the I/O-boundedness of the Federator workload. Counter to intuition, Federator MPL can be kept high when Federators are CPU- or I/O-bound, but must be low when Federators have simultaneously large CPU- and I/O-demands.

The primary questions addressed in Chapter 4 are:

(i) *What factors cause the Hump? Or, in other words, what factors cause Locals to receive good response times even with many Federators in the DBMS at the same time?*

(ii) *How do you model the DBMS and predict query performance when the DBMS is shared by Locals and Federators?*

(iii) *How do you determine how many Federator queries can be admitted into the DBMS so as to achieve specific performance goals for Locals?*

The Hump

As indicated above, Chapter 4 fixes the Local workload so that it is CPU-bound, and varies the Federator workload database size so that Federators shift from CPU-bound to I/O-bound. It is difficult to predict the performance effects on the Local workload as the Federator workload varies from CPU- to I/O-bound. Intuitively, computer scientists often come to one of two conclusions: (i) Performance is worst when both workloads are CPU-bound, since they compete for the same resource, or (ii) Performance is worst when the Federators are I/O-bound, since that slows down all I/O-requests, and both workloads (who both depend on I/O, though in different amounts) suffer. Either way, most people predict a *monotonic increase or decrease* in response times as the second workload is varied.

I show that the real-life performance trends are, in fact, often non-monotonic: Local (and overall) mean response times first rise, then fall as the Federator workload shifts from CPU- to I/O-bound. The non-monotonic trend, which I call *The Hump*, is illustrated in Figure 1.4(a), and understanding its causes and predicting it is at the heart of Chapter 4.

The Hump is a significant performance problem, and determines how admission control should be applied to the Federator workload to give performance isolation to the Locals. At the endpoints, Local (and overall) mean response times are low, even when there are many Federator queries running in the DBMS at the same time. At the peak of the Hump, however, running Locals with many Federators hurts Local (and overall) mean response times by nearly an order of magnitude (a factor of 7.5 in this study). To protect the Locals, while admission control can admit many Federators at the endpoints (CPU-bound Federators and I/O-bound Federators), it can only let in a few at the peak of the Hump. This type of policy is illustrated in Figure 1.4(a).

As indicated above, intuition does not help scientists to correctly predict the Hump. Usually, this is not due to incorrect understanding of systems issues, but due to the difficulty inherent in estimating the relative magnitude of their performance effects. In general, the systems issues that are most commonly considered are competition for CPU (when Federators are CPU-bound) and I/O resources (when Federators are I/O-bound). The expected result of these issues is that queries wait longer for those resources, or those queries demand more work from those resources. Table 1.6 summarizes the most common systems issues that arise when contemplating these DBMS performance issues.

The remainder of Chapter 4 is dedicated to understanding where the intuition breaks down. First, I determine exactly how important each of the issues listed in Table 1.6 are from a performance standpoint. Next, I demonstrate that those issues do not significantly contribute to cause the Hump. Finally, I determine what issues are, in fact, responsible for the Hump trend, and shows how to incorporate these into queueing theoretic models to accurately predict the hump.

Queueing Theory is Lacking

The second section of Chapter 4 shows the limitations of current queueing theoretic analysis techniques when applied to DBMS. Some queueing theoretic models have been used to model DBMS performance for years. Queueing theoretic models help fill in where intuition leaves off — they help to quantify and establish

	Resource/Effect	Federator Wkld	Issue
(i)	CPU wait time	CPU-bound	Increased competition for CPU, since queries spend less time in I/O
(ii)	I/O wait time	I/O-bound	Bigger databases need more I/O requests.
(iii)	I/O wait and exec time	I/O-bound	Bigger databases spread data further on disk, increasing seek times.
(iv)	CPU exec time	I/O-bound	Bigger databases have more data to process, needing more CPU instructions per query.

Table 1.6: The systems issues upon which many incorrect intuitive predictions of shared DBMS performance are based.

relative significance of the systems issues seen in Table 1.6 from a performance standpoint. Unfortunately, these models fail because they do not incorporate essential performance characteristics that are inherent to DBMS resources (particularly CPU).

Queueing theoretic modeling shows that one should expect that, as the second workload shifts from CPU- to I/O-bound, the response time should follow a plateau, then dip, and then rise exponentially. Many different queueing models all result in similar performance trends, which suggest that something critical is missing from these models. The final section of Chapter 4 will establish what, exactly, is missing, and how to update the models to properly incorporate them.

Isolated Demand Decomposition (IDD)

The final section of Chapter 4 focuses on the development of *Isolated Demand Decomposition (IDD)*. IDD's development depends on first determining what systems issues cause the Hump performance, and then determining how to incorporate these issues into queueing theoretic models.

To determine which systems issues cause the Hump, I profile the DBMS, and conduct an extensive statistical analysis to discover where queries spend their time. I discover that the Hump is largely due to two factors: the CPU becomes slower, and the queries execute more instructions than expected. The CPU slows down because of CPU data cache misses, which reduce the number of instructions per second the CPU can execute. Queries execute more instructions because data contention for internal DBMS data structures (spin-locking and latching) increase.

Chapter 2

Prioritization in OLTP and Transactional Web Applications

2.1 Background and Overview

In the introduction, we saw that DBMS are *shared* systems and have only *limited* resources. The consequence is that users will inevitably experience delays, when the DBMS or its components become overloaded. Fortunately, some users are more important than others, and as a result, we can make better use of the DBMS's limited resources by *prioritizing* the most important, high-priority queries in the DBMS. The goal of prioritization is to reduce the delays that high-priority queries experience in the DBMS.

DBMS are complex systems, comprised of many different physical devices (such as CPUs, disk drives, memory, etc) and logical resources (such as locks, queues, and so forth). DBMS queries are essentially small programs that execute at the DBMS, and need to use these devices and resources in turn. Queries all have different device and resource demands, the amount of work they require to be executed at each device or resource during their execution.

Queries experience delays in DBMS whenever they encounter delays when trying to use devices and resources within the DBMS. Physical devices within the DBMS are inherently *limited resources*, because each device can only do a limited amount of work in any period of time, and can usually only work on a single query (or at best, a few queries) at a time. Thus, when queries compete for those physical devices, they are forced to wait, and incur delays which slow them down. Logical resources, such as locks, are almost always used to control queries' access to data or physical devices within the DBMS. As a result, competition for logical resources also cause queries to wait and incur delays.

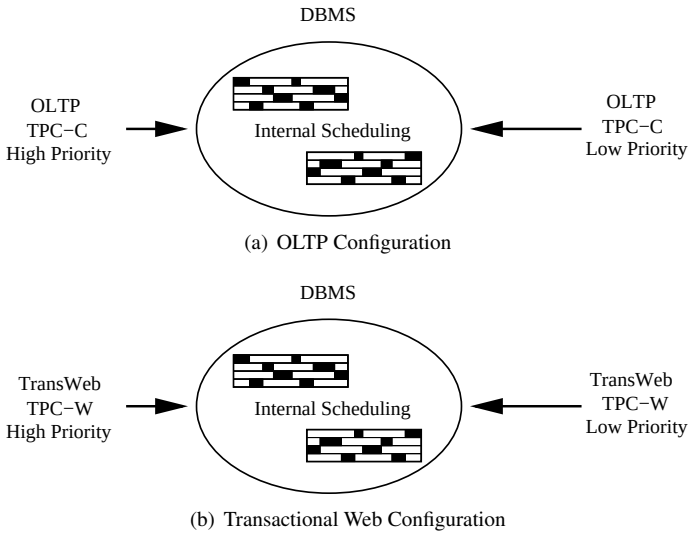


Figure 2.1: The system configurations considered in this chapter: OLTP and Transactional Web workloads with high- and low-priority queries, sharing a DBMS. High-priority queries are prioritized using internal prioritization.

The only way to effect query prioritization in DBMS is to prioritize queries' access to the devices and resources within the DBMS. One can accomplish that in two ways: (i) admit queries to the DBMS, and then use internal prioritization, to keep low-priority queries from using devices needed by high-priority queries,

or (ii) do not admit queries to the DBMS, and use external admission control to keep low-priority queries out of the DBMS, and from using devices needed by the high-priority queries. This chapter focuses on option (i), internal prioritization.

Implementing internal prioritization in a DBMS is a challenging problem. DBMS have many different physical devices, and many more logical resources. On *which devices and resources* should query scheduling be implemented? Implementing scheduling at *every device and resource* within the DBMS is *not a good idea* because (i) implementing scheduling algorithms is often difficult, expensive, and time-consuming for DBMS vendors and developers, especially when they affect mission-critical pieces of the DBMS source code, and (ii) scheduling queries incurs a run-time overhead, which means that implementing scheduling when it is not needed can actually hurt, and not help query performance.

Beyond the issue of what devices and resources should be scheduled, one has to decide *what scheduling policies* should be used. There are hundreds of different scheduling policies available, including processor sharing, first-come first-serve, last-come first-serve, and shortest job first, to name a few. Each one may result in different performance, and it is unclear which policies are best to implement for each resource.

The basic operation of DBMS further complicates the problem of choosing a scheduling policy, because they provide additional options for most scheduling policies. DBMS often have to *serialize* the execution of two queries to ensure data consistency (ACID properties), usually via the use of *locking*. This means that a high-priority query may be forced to wait for a low-priority query to completely finish before the high-priority query is able to get any work done. Serialization and locking introduce delay for high-priority queries, hurting high-priority performance isolation.

Most scheduling policies can be augmented using *preemption* or *priority inheritance* to minimize high-priority delays due to serialization and locking, drastically increasing the number of scheduling policies one needs to consider. Preemption lets high-priority queries terminate any low-priority queries that the high-priority queries must wait for. Priority-inheritance boosts the priority of any low-priority queries that the high-priority queries must wait for (such low-priority queries effectively become high-priority). While both preemption and priority-inheritance have potential performance benefits, they also have potential performance drawbacks, and it is unclear which dominate. Preemption kills queries, which can cause starvation and ruin overall DBMS throughput. Priority-inheritance can reduce the benefits seen by scheduling by introducing too many high-priority queries. Furthermore, both policies increase DBMS development times and costs.

In summary, when attempting to implement query prioritization by implementing query scheduling at devices and resources in the DBMS, four primary questions must be addressed: (i) Which devices and resources should be scheduled? (ii) Which scheduling policies should be used? (iii) Should preemption or priority-inheritance be used? (iv) How do changes to the DBMS and workload change the answers to questions (i) (ii) and (iii)?

The problem of prioritizing queries within DBMS is not new, and has been studied before in many contexts. We discuss prior approaches to implementing prioritization in Section 2.4. Unfortunately, the prior work does not provide much insight for the systems, DBMS, and workloads that we consider in this research, and do not help to answer the questions listed above. Existing research appears to come to contradictory conclusions of what resources to schedule and what scheduling policies are most effective. The differences of existing research appears to be due to the fact that each considers both (i) different DBMS implementations, some of which are not commercially relevant, and (ii) different workloads which do not resemble the OLTP and transactional web workloads considered in this chapter.

The key idea of this chapter revolves around identifying the *bottleneck resource*, and concentrating scheduling efforts on that resource. The bottleneck resource is the resource in which queries spend the bulk of their time, and is the biggest contributor to query response time.

This chapter will examine systems of the form illustrated in Figure 2.1, and described in detail in Section 1.7. A DBMS is shared between two OLTP (TPC-C) or two Transactional Web (TPC-W) query workloads, one high-priority and one low-priority. Internal prioritization is used to prioritize and provide performance isolation to the high-priority queries. This type of scenario arises in many commercial systems, but is of particular interest in the area of e-Commerce. It is extremely common for e-Commerce applications to have some users which are more important than others, either because (i) users pay for "gold service" and better performance, or (ii) the company recognizes that when certain users get better performance, they are more satisfied, and are more likely to spend more at their site.

This chapter consists of two primary contributions: (i) A comprehensive bottleneck analysis of real-world DBMS implementations and workloads, which identifies where queries spend their time in a DBMS, and (ii) A performance analysis of scheduling policies applied to the devices and resources in the DBMS.

2.1.1 Bottleneck Analysis

To determine how best to effect query prioritization, one must first determine which devices/resources must be scheduled. Implementing query scheduling on the wrong devices/resources is likely to result in both (i) undesirable performance overheads that slow down all queries, and (ii) unnecessary DBMS development and testing costs to implement scheduling on those devices (potentially resulting in bugs).

In the bottleneck analysis, we experiment with a variety of workloads running against a variety of DBMS implementations, and examine how the bottleneck changes as the workload changes. The workloads we consider are an OLTP TPC-C workload and a transactional web TPC-W workload. We vary the scale of these workloads in three ways: (i) scaling the number of clients (concurrency level), (ii) scaling the size of the database, or (iii) scaling the workload according to the benchmark specification, changing both the number of clients and the size of the database. The DBMS implementations considered are IBM DB2, Shore and PostgreSQL. IBM DB2 and Shore both use 2PL concurrency control, while PostgreSQL uses MVCC concurrency control.

Key Idea

The key observation is that, to effect DBMS query prioritization, *it is both sufficient and necessary to prioritize queries only at the bottleneck resources*. The bottleneck resources are those devices/resources in which queries spend most of their response time.

In particular, to improve query response times, one must consider: (i) consider all resources which can cause query delays, not just the obvious physical resources; this includes resources such as Locks, and (ii) consider the time that queries spend executing in and waiting for each resource, as a fraction of the overall query response time. This is in contrast to much conventional thinking in which one must simply consider resource utilization for the physical devices in the DBMS, such as CPU and I/O. Looking at where queries spend their time is essential, as resource utilization alone does not paint a sufficient enough picture to improve query response times. Likewise, DBMS queries can incur significant delays due to inter-query dependencies, such as through Locks, which are not seen in measurements made in primary physical devices

Concurrency Algorithm	Workload Class	Workload	Standard Bottleneck	Many Clients Bottleneck	Big DB Bottleneck
2PL	OLTP	TPC-C	Locks	Locks	I/O
MVCC	OLTP	TPC-C	I/O	Locks	I/O
2PL	TransWeb	TPC-W	CPU	CPU	CPU
MVCC	TransWeb	TPC-W	CPU	CPU	CPU

Table 2.1: Summary of bottleneck resources as a function of the DBMS workload and the DBMS concurrency control algorithm.

(such as CPU and I/O).

Summary of results

The primary observation is that the bottleneck resource changes in well-defined ways, based on the workload and the DBMS implementation. First, OLTP TPC-C workloads running on 2PL DBMS are almost always lock-bound. Second, OLTP TPC-C workloads running on MVCC workloads are predominantly I/O-bound but are occasionally lock-bound in certain circumstances (when the number of clients is increased but the database size is fixed). Third, transactional web TPC-W workloads are almost always CPU-bound.

Table 2.1 summarizes these results, indicating the bottleneck resource based on the workload and the concurrency level, and based on how the workload is scaled. The Standard Workload is scaled as described in the benchmark specification, scaling both the number of clients and the database size together. Many Clients is scaled by increasing the number of clients only, and Big DB is scaled by increasing the size of the database only.

2.1.2 Scheduling Algorithm Analysis

Once the bottleneck resource is determined, one must determine what scheduling policy should be used to schedule that resource. Many options are possible, though we focus on two in particular: (i) preemptive versus non-preemptive scheduling and (ii) scheduling with priority-inheritance versus without priority inheritance.

Preemption and priority-inheritance both try to minimize the amount of time that high-priority queries wait for low-priority queries, which can, in effect, improve the granularity of scheduling. Unfortunately both also have potential disadvantages, making it unclear whether either technique should be used. Preemption can cause starvation to low-priority queries, and can hurt overall system performance by introducing far extra work into the DBMS. Priority-inheritance can be more difficult to implement correctly (resulting in higher DBMS development costs, or the risk of bugs), but can also hurt high-priority queries, since high-priority queries may have to wait for low-priority queries (that have inherited high-priority status).

The scheduling policies that are considered, according to resource are as follows:

1. `NP-LQ` — Lock scheduling, which non-preemptively schedules locks by reordering lock queues.

2. `NP-LQ-Inherit` — Lock scheduling, identical to NP-LQ, except that priority inheritance is added.

3. `P-LQ` — Lock scheduling, which preemptively schedules locks, by reordering lock queues and killing (and restarting) low-priority queries when they prevent high-priority queries from acquiring locks.

4. `CPU-Prio` — CPU scheduling, which changes the UNIX-like priority of the operating system scheduler.

5. `CPU-Prio-Inherit` — CPU scheduling, identical to CPU-Prio, except that priority inheritance is added.

6. `P-CPU` — CPU (and Lock) scheduling, identical to CPU-Prio, but also kills (and restarts) low-priority queries when they prevent high-priority queries from acquiring locks.

Key Idea

To evaluate the effectiveness of different scheduling policies, we implement and experiment with the above scheduling policies on OLTP TPC-C workloads and transactional web TPC-W workloads running on Shore (with 2PL concurrency control) and PostgreSQL (with MVCC concurrency control). We examine the effectiveness of each policy as a function of the amount of load on the DBMS.

Summary of results

The first observation made is that scheduling the bottleneck resource is essential to effectively prioritize high-priority queries.

This result proves that pre-existing belief that prioritizing the CPU is good enough to prioritize high-priority queries [2] is wrong. The intuition for this incorrect belief is that queries can only use other resources after first running on the CPU, and thus scheduling the CPU implicitly schedules other resources as well. The basic premise of the intuition is true, but it turns out that scheduling the CPU alone is not effective (especially when locks are the bottleneck). This is due to the fact that when the CPU is not the bottleneck, high- and low-priority queries both (i) see few other queries at the CPU and (ii) experience small waiting times for the CPU, regardless of whether or not scheduling is applied. As a result, high-priority queries do not get access to resources much faster than low-priority queries.

Despite the fact that scheduling the bottleneck *directly* appears to be essential, CPU scheduling *does* appear to successfully prioritize high-priority queries in I/O-bound workloads, such as in OLTP TPC-C on PostgreSQL. This is likely due to the fact that I/O uses a complicated elevator scheduling policy which carries a lot of state, and a small reordering can result in significant performance differences. That said, the performance isolation that CPU scheduling gives to high-priority queries in an I/O-bound workload is not as strong as lock scheduling gives to lock-bound workloads nor as strong as CPU scheduling gives to CPU-bound workloads. This suggests that I/O-scheduling will be more effective at providing prioritization.

When locks are the bottleneck, in 2PL OLTP TPC-C workloads, CPU scheduling is almost completely ineffective. On the other hand, when lock scheduling is used (`NP-LQ`), high-priority response times can be reduced by a factor of nearly 4 (in our experiments; under higher loads this factor should increase). For

MVCC transactional web TPC-W workloads, where I/O is the bottleneck, CPU scheduling can be moderately useful, reducing high-priority response times by a factor of two.

Priority-inheritance is able to improve lock scheduling for 2PL OLTP TPC-C workloads by a factor of 30%, while it can improve CPU scheduling for MVCC transactional web TPC-W workloads by a factor of three.

Preemption can improve the effect that lock scheduling has for high-priority queries in OLTP TPC-C workloads by a factor of nearly 2, while it hurts low-priority queries drastically (hurting the overall average performance drastically). This penalty is due to the fact that preemption kills too many low-priority queries, which must be restarted and re-executed. Preemption is not useful for CPU scheduling in MVCC transactional web TPC-W workloads due to the fact that: (i) the CPU is scheduled using a fine-grained time-slicing (ii) when a query is preempted, any I/Os that it had in progress have already caused the drive heads to seek. Thus, simply waiting for those I/Os to complete is not very different from terminating them and issuing the next request.

2.2 Organization of this chapter

The remainder of this chapter proceeds as follows: Section 2.3 introduces and motivates the problem of prioritization in OLTP and transactional web DBMS workloads. Section 2.4 summarizes the prior work in the field of prioritizing queries in DBMS. Section 2.5 describes the experimental setup used throughout this chapter. Section 2.6 describes the process and the results of the bottleneck analysis for each of the workloads and DBMS considered in this chapter. Section 2.7 documents the evaluation process of various scheduling policies for each workload and DBMS considered. Section 2.8 summarizes the immediate results of the chapter, and Section 2.9 and Section 2.10 discuss the impact of the work along with future directions the work can be taken.

2.3 Introduction

Online transaction processing (OLTP) is a mainstay in modern commerce, banking, and Internet applications. For many OLTP applications, particularly e-commerce applications, clients require fast access times. Unfortunately, serving requests which involve database activity for dynamic query processing and data generation can be very slow — orders of magnitude slower than delivering static content. This slowness is exacerbated under heavy load and overload.

To alleviate the problem of costly database accesses, it can be extremely valuable to assign priorities to users and provide differing levels of performance. When both high- and low-priority clients share the database system, high-priority clients should complete more quickly on average than their low-priority counterparts. For example, an online merchant may make use of priorities to provide better performance to new prospective clients, or to big spenders expected to generate large profits. Alternatively, a web journal may provide improved responsiveness to "gold-customers" who pay higher subscription costs. Finally, point-of-sales systems may run long-running maintenance queries "in the background," at low-priority while customer purchases execute quickly at high-priority.

The goal of this research is to provide prioritization and differentiated performance classes within a traditional (general-purpose) relational database system running OLTP and transactional web workloads, including read/write transactions. This chapter provides a detailed resource utilization breakdown for OLTP workloads executing on a range of database platforms including IBM DB2[45], Shore[17], and PostgreSQL[52]. IBM DB2 and PostgreSQL are both widely used (commercial and noncommercial) database systems. Shore is an open source research prototype using traditional two-phase locking (2PL), the concurrency control used in DB2. PostgreSQL (like Oracle), on the other hand, uses multiversion concurrency control (MVCC) [12]. The chapter also implements several transaction prioritization policies within Shore and PostgreSQL. The prioritization policies studied include non-preemptive priorities, non-preemptive priorities with priority inheritance, and preemptive abort scheduling. Given the focus on web and complex transactional applications, we use the benchmark OLTP workloads TPC-C and TPC-W.

The primary contributions of this research are twofold:

1. Identification of bottleneck resource(s) across DBMS, workloads and concurrency levels.
2. Demonstration that simple priority scheduling inside the DBMS significantly improves high-priority transaction execution times without penalizing low-priority transactions.

With respect to bottleneck identification, we show that the bottleneck resource for TPC-C on IBM DB2 and Shore, both of which use 2PL, is lock waiting. By contrast, for the same TPC-C workload, PostgreSQL, which uses MVCC, exhibits an I/O synchronization bottleneck. For TPC-W on DB2 and PostgreSQL, we find that the bottleneck is always the CPU.

On the issue of scheduling policies, we find that scheduling of bottleneck resources results in improving high-priority transaction execution times considerably. For systems with lock bottlenecks (TPC-C on DB2 and Shore), CPU scheduling is ineffective, but lock scheduling can improve high-priority performance by a factor of 5.3. For systems with CPU bottleneck (TPC-W), lock scheduling is ineffective, while CPU scheduling improves high-priority performance by a factor of 4.5. For PostgreSQL, which has an I/O synchronization bottleneck, CPU scheduling with priority inheritance yields a factor of 6 improvement of high-priority transactions. Provided that the fraction of high-priority transactions is small, the penalty to the low-priority transactions is negligible as long as preemption is not used.

2.4 Prior Work

There is a wide range of well-known database research, including that of Abbott, Garcia-Molina, Stankovic, and others, studying different transaction scheduling policies and evaluating the effectiveness of each. Most existing implementation work is in the domain of real-time database systems (RTDBMS), where the goal is not improvement of mean execution times for classes of transactions, but rather meeting deadlines associated with each transaction. These RTDBMS are sufficiently different from the general-purpose DBMS studied in this chapter to warrant investigation as to whether results for RTDBMS apply to general-purpose DBMS as well. In addition to the existing implementation work in RTDBMS, there has also been work on simulation and analytical modeling of prioritization in DBMS and RTDBMS. Unfortunately, the simulation and analytical approaches have difficulty in capturing the complex interactions of CPU, I/O, and other resources in the database system.

In Section 2.4.1 we summarize the most relevant existing research on transaction prioritization within RTDBMS. In Section 2.4.2 we summarize the existing and ongoing work on prioritization in general-purpose DBMS.

2.4.1 Real-Time Databases

Real-time database systems (RTDBMS) have taken center stage in the field of database transaction scheduling for the past decade. These systems are useful for numerous important applications with intrinsic timing constraints, such as multimedia (*e.g.*, video-streaming), and industrial control systems. Traditional DBMS with transaction priorities differ from RTDBMS. In RTDBMS, each transaction is associated with time-dependent constraints (usually deadlines), which must be honored to maintain transactional semantics. The goal of minimizing the number of missed constraints (deadlines), requires maintaining time-cognizant protocols and various specialized data structures [79], unlike general-purpose DBMS. Scheduling issues such as priority inversion may have different costs for RTDBMS as compared to traditional DBMS: *i.e.*, a single priority inversion may cause a missed deadline while hardly affecting overall mean execution time. Lastly, RTDBMS workloads can differ substantially from traditional DBMS workloads.

Abbott and Garcia-Molina [2, 1, 3, 4, 5] extensively study scheduling RTDBMS in simulation, preemptively and non-preemptively scheduling the critical resources (CPU, locks and I/O) to meet real-time deadlines. On the question of which resource needs to be scheduled, Abbot and Garcia-Molina conclude that CPU scheduling is most important, as transactions only acquire resources when they have the CPU [5]. Additionally, they find scheduling of concurrency control resources also improves performance.

With respect to scheduling policies, both Abbott and Garcia-Molina [5] and Huang et. al. [42] examine priority inheritance and preemptive prioritization in RTDBMS that use 2PL, to address the priority-inversion problem. Abbott and Garcia-Molina find that priority inheritance is important when ensuring that deadlines are met, in particular when the database is small. In contrast, Huang et. al. find that standard priority inheritance is not very effective in RTDBMS.

Kang et. al. [48] differentiate between classes of real-time transactions, providing different classes with QoS guarantees on the rate of missed deadlines and data freshness. In that work, Kang et. al. focus on main memory databases.

Baccouche [10] develops the H/M/L scheduling policy for general RTDBMS, which uses admission control to prioritize real-time transactions in overload situations. They show that using admission control

to prioritize queries in RDBMS does not hurt overall performance (measured by the number of transactions missing their deadlines) much, while improving high-priority performance drastically.

Our results will differ from those above as follows: (i) CPU is not always the most important resource to schedule. For DBMS using 2PL and TPC-C workloads we see that scheduling locks is far more effective than CPU scheduling. (ii) Priority inheritance is not always necessary, and is ineffective for some workloads and DBMS.

We attribute these differences in results to the many differences between real-time and traditional DBMS and their workloads.

2.4.2 Priority Classes

Existing work to establish priority classes for mean performance (rather than meeting specific deadlines), can be divided into techniques which schedule transactions (i) outside the DBMS and (ii) inside the DBMS. External scheduling is typically implemented using admission control to prevent transactions from entering the DBMS. Internal prioritization, by contrast, prioritizes transactions as they execute within the database.

Recent work at IBM implements priority classes in admission control [28]. The approach makes admission control decisions based not only on the number of transactions in the DBMS, but also on transaction priorities, by limiting the number of low-priority transactions that are able to interfere with high-priority transactions. Such admission control reduces lock contention and also limits inefficiencies introduced when the system is under overload, such as virtual memory paging and thrashing. Consequently, high-priority transactions under overload can benefit significantly.

Despite the simplicity of admission control for prioritization, we believe that internal DBMS prioritization is more effective. Internal prioritization allows direct control of DBMS resources, and can utilize knowledge of query plans, transaction resource needs, and system resource availability (*e.g.* I/O requests and granted locks).

There is much room for further research in transaction scheduling internal to the DBMS. The most pertinent work, by Carey et. al. [18] is a simulation study of our same fundamental problem: evaluating priority scheduling policies within DBMS to improve high-priority transaction performance. They assume a read-only workload, but recommend mixed read/write workloads should also be examined in the future. In contrast, our work assumes mixed read/write workloads and our work uses fully implemented DBMS rather than a simulator.

Brown et. al. [15] address multi-class workloads with per-class response time goals. Again, this is a pure simulation study without experimental validation on a DBMS prototype. Moreover, it focuses on a single resource, memory, while in our work we analyze the resource breakdown for different DBMS and workloads and consider the different bottleneck resources.

Prioritization within traditional DBMS has not been a focus for academic research. As a testament to the importance of the problem, however, both IBM DB2 and Oracle provide prioritization tools (IBM DB2gov and QueryPatroller [45, 20] and Oracle DRM [67]), all of which focus on CPU scheduling. We have experimented extensively with IBM DB2gov, and find it does not provide nearly as large of a prioritization benefit for the lock-bound workloads discussed in this chapter. This chapter addresses a wider range of scheduling policies for both CPU and lock resources.

2.5 Experimental Setup

This section describes experimental setup details including the workloads, hardware, and software used.

2.5.1 Workloads

As representative workloads for OLTP and transactional web applications, we experiment with the TPC-C [22] and TPC-W [23] (TPC-W Shopping Mix) benchmarks.

The TPC-C workload implementation for DB2 and PostgreSQL is written and graciously donated by IBM. The TPC-C Shore implementation was written at CMU. TPC-C is modified to allow each client to access a different warehouse and district for each transaction, which produces more uniform access to the database. The TPC-W workload comes from the PHARM [55] project with minor improvements, such as an improved connection pooling algorithm.

2.5.2 Hardware and DBMS

All of the TPC-C experiments for DB2 and Shore are performed on a 2.2-GHz Pentium 4 with 1GB RAM, one 120GB IDE drive, and a 73GB SCSI drive. The TPC-C PostgreSQL experiments are conducted on a comparable machine with two 1-GHz processors and 2GB of RAM, allowing us to handle the larger memory requirements of PostgreSQL. The results for PostgreSQL on the dual-processor (two 1-GHz) machine are similar to those when performed on the single-processor 2.2-GHz machine used by DB2 and Shore. The TPC-W experiments are all conducted with the database running on the 2.2-GHz machine; the web server and Java servlet engine run on a Pentium III, 736Hz processor with 512MB of main memory; and the client applications run on two other machines. The operating system on all machines is Linux 2.4.

The DBMS we experiment with are IBM DB2 [45] version 7.1, PostgreSQL [52] version 7.3, and Shore [17] interim release 2. Several modifications are made to Shore, to improve its support for `SIX` locking modes, and to fix minor bugs experienced in transaction rollbacks.

2.6 The Bottleneck Resource

Central to this work is the idea that understanding a workload's resource utilization is essential for effective prioritization. In order to improve high-priority transaction execution times, the *bottleneck resource*, where transactions spend the bulk of their execution time, must be scheduled, either directly or indirectly. Given the complexity of modern database systems, predicting the bottleneck resource is non-trivial.

In this section, we derive resource utilization breakdowns and determine the bottlenecks for TPC-C on Shore, DB2, and PostgreSQL and for TPC-W on DB2 and PostgreSQL. First, we describe the model used for breaking down transaction resource utilization. Next, we examine how these resource breakdowns change under varying concurrency levels and database sizes.

2.6.1 DBMS Resources: CPU, I/O, Locks

Since the goal of this chapter is to improve individual transaction execution times, and not overall throughput, it is important to break down execution times from the point of view of a transaction. We focus on three core DBMS resources: CPU, I/O, and locks, chosen since they are under control of the database, and are believed to be important in performance [5].

We define the total execution time of a transaction, T_{Trans}, as the time from when the transaction is first submitted to when it completes. We break T_{Trans} into three components, $T_{Trans} = T_{CPU} + T_{IO} + T_{Lock}$, corresponding to CPU, I/O, and locks, respectively. These components consist of just the synchronous time in which the transaction is completely dedicated to either waiting for or consuming the corresponding resource. T_{CPU} consists of the time spent running on the processor and the time spent in the running state, waiting for the processor. T_{IO} consists of the time spent issuing and waiting for synchronous I/O to complete (although the cost of issuing an I/O operation is negligible). T_{Lock} is the time that a transaction spends waiting for database locks. Of course, time spent holding locks is accounted according to whether the transaction holding the lock is waiting for or consuming CPU or I/O or waiting for another lock.

Database locks are broken into "heavyweight" and "lightweight" locks. Heavyweight locks are used for logical database objects, to ensure the database ACID properties. Lightweight locks include spinlocks and mutexes used to protect data structures in the database engine (such as lock queues). We find that lightweight locking is not a significant component of transaction execution times in either Shore or IBM DB2. PostgreSQL, however, has significant lightweight lock waiting, due to an idiosyncrasy of the PostgreSQL implementation. We find almost all lightweight locking in PostgreSQL functions to serialize the I/O buffer pool and Write-Ahead-Logging activity (via the `WALInsert`, `WALWrite`, and `BufMgr` lightweight locks). As a result, we attribute all the lightweight lock waiting time for the above-listed locks to I/O. We use the term "locks" throughout the remainder of this chapter to refer exclusively to heavyweight locks.

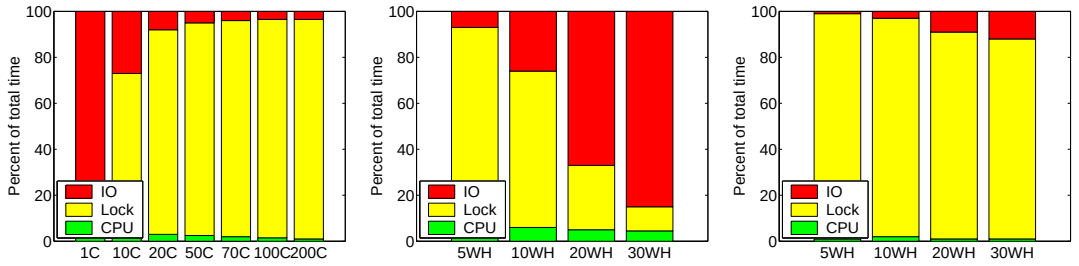
We use two different methods to obtain the desired resource breakdowns, depending on the DBMS used. For DB2, since its source code is unavailable, we rely on its built-in resource measurement facilities: snapshot and event monitoring [45]. For PostgreSQL and Shore, we implement custom measurement functionality by instrumenting the DBMS itself. We compute the total CPU, I/O and lock wait time over all transactions and then determine the fraction each component makes up of the sum of all execution times.

For DB2 and PostgreSQL, which use a process-based architecture, we verify the breakdowns at the operating system via the `vmstat` command, recording the fraction of time DBMS processes spend in the CPU run queue (`TASK_RUNNING`), blocked on I/O (`TASK_INTERRUPTIBLE`), or waiting for locks (`TASK_UNINTERRUPTABLE`). We also use a patch to the Linux kernel to accurately measure CPU wait times (not measured in Linux by default).

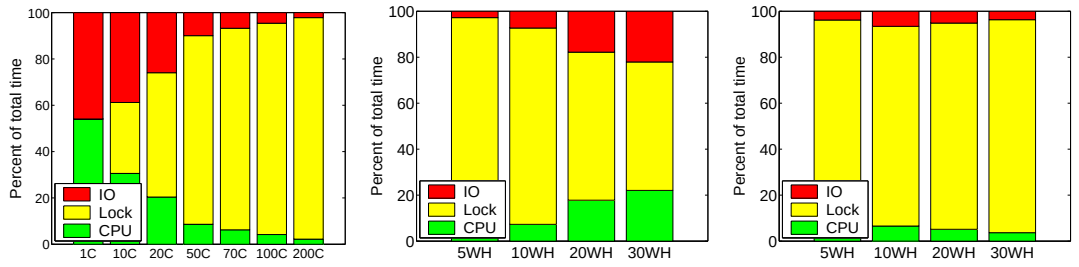
2.6.2 Breakdown Results

TPC-C. Figure 2.2 shows the resource breakdowns measured for TPC-C running on IBM DB2, PostgreSQL, and Shore. The graphs depict the average portions (indicated as percentages) of transaction execution time due to CPU, I/O, and lock resource usage.

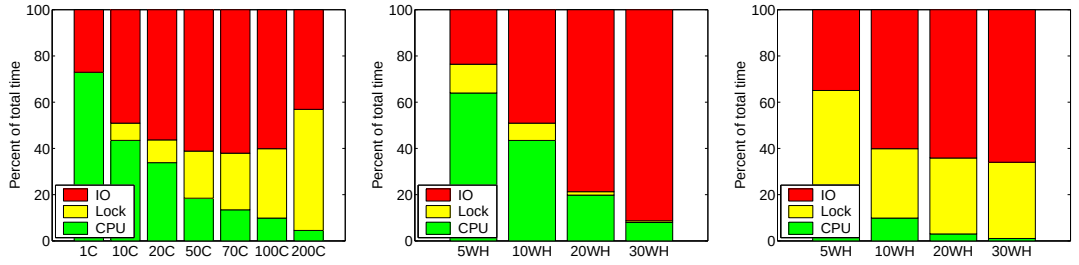
There are two sources of error in this data: (i) Measurement error and (ii) Sample error. (i) Measurement error is negligible in all systems except for IBM DB2, which exhibits an error of less than 10%, due to the low resolution of DB2's I/O and CPU measurements. All measurements are normalized to 100% for



(a) **DB2**: Varying Clients, 10 Warehouses (b) **DB2**: 10 Clients, Varying Warehouses (c) **DB2**: Standard Scaling (10 clients per WH)

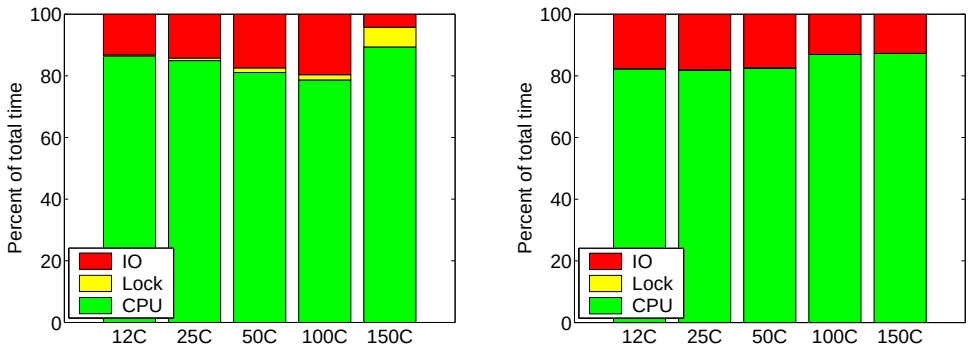


(d) **Shore**: Varying Clients, 10 Warehouses (e) **Shore**: 10 Clients, Varying Warehouses (f) **Shore**: Standard Scaling (10 clients per WH)



(g) **PostgreSQL**: Varying Clients, 10 Warehouses (h) **PostgreSQL**: 10 Clients, Varying Warehouses (i) **PostgreSQL**: Standard Scaling (10 clients per WH)

Figure 2.2: Resource breakdowns for TPC-C transactions under varying databases and configurations. The first row shows DB2; the second row shows Shore; and the third row shows PostgreSQL. The first column (Figures 2.2(a), 2.2(d), 2.2(g)) shows the impact of varying concurrency level by varying the number of clients. The second column (Figures 2.2(b), 2.2(e), 2.2(h)) shows the impact of varying the database size (number of warehouses) while holding the number of clients fixed. The third column (Figures 2.2(c), 2.2(f), 2.2(i)) shows the impact of varying both the number of clients and the database size according to the TPC-C specification (10 clients for each warehouse).



(a) **IBM DB2** (b) **PostgreSQL**

Figure 2.3: Resource breakdowns for TPC-W transactions running on IBM DB2 and PostgreSQL.

clarity. (ii) Sample error is limited by collecting enough samples, such that the standard error of the mean ($\frac{stddev}{\#samples}$) is less than 1%.

For each DBMS, Figure 2.2 presents three sets of results illustrating the most significant trends. In the first column, the database size is held constant at 10 warehouses (WH), and the number of clients connected to the database (concurrency) is varied. In the second column, the number of clients is held constant at 10, and the size of the database is varied by increasing the number of warehouses. In the third column, we vary the number of clients and warehouses together, always holding the number of clients at 10 times the number of warehouses, as specified by TPC-C, demonstrating breakdowns for standard *"realistic"* configurations. Throughout, the think times are fixed at zero.

The database sizes for TPC-C range from 500MB to 3GB, as the number of warehouses grows from 5 to 30 (100MB per WH). The buffer pool size is approximately 800MB for each DBMS, chosen to minimize transaction execution times.

The main result shown in Figure 2.2 is that locks are the bottleneck resource for both Shore and DB2 (rows 1 and 2), while I/O tends to be the bottleneck resource for PostgreSQL (row 3). We now discuss these in more detail.

We start with some obvious trends. First observe that as concurrency is increased while fixing the database size (column 1), lock contention increases. Also, as the database size grows, while the concurrency level is held constant (column 2), the I/O component grows, and the lock component decreases. When the database and concurrency level are scaled according to TPC-C specifications, the relative resource breakdowns remain fairly stable.

The resource breakdowns for Shore and DB2 (rows 1 and 2) are quite similar, and almost always depict lock bottlenecks. This may be surprising, since concurrency control was a very active area of research in the 1970's and 80's, and thus one might think that locking problems were all resolved at that time. Given our hardware limitations, we can only experiment with up to 30 WH. It is plausible that the bottleneck may shift to I/O as the database size increases. Alternatively, additional RAM and disks may hide the growing I/O for larger databases, leaving locks as the bottleneck resource.

The resource breakdowns for PostgreSQL (row 3) differ greatly from those for Shore and DB2: Post-

greSQL almost always exhibits an I/O bottleneck. As indicated earlier, PostgreSQL I/O time includes the time for both the actual I/O operation and the lightweight lock I/O synchronization. Almost all (80–95%) of the I/O time is due to I/O synchronization in the standard case (Figure 2.2(i)). While this suggests that I/O scheduling will be necessary for PostgreSQL prioritization, in Section 2.7, CPU scheduling will be used to indirectly schedule I/O.

Although not the bottleneck, locks are sometimes a non-trivial component for PostgreSQL. In particular, locks reach 50% when concurrency is increased while fixing the database size (Figure 2.2(g)), and reach 30% when standard TPC-C scaling is used (Figure 2.2(i)).

The fact that PostgreSQL's resource breakdowns differ from those for Shore and DB2 is due to differences in concurrency control in these systems: Shore and DB2 employ 2PL, while PostgreSQL uses MVCC. With MVCC, PostgreSQL transactions only have to wait for write-on-write conflicts. The result is fewer lock waits in PostgreSQL than in Shore and DB2, shifting its bottleneck to I/O.

Each breakdown presented in Figure 2.2 is an average computed over all transactions in an experimental run, and as such, may not be representative of any particular, or even most transactions. The breakdowns can be sharply skewed by a small fraction of exceptional transactions with extremely long execution times. Thus, the breakdowns are primarily an indicator of the relative importance of the resources when minimizing average transaction execution times.

TPC-W. Figure 2.3 shows resource breakdowns for TPC-W transactions running on IBM DB2 and PostgreSQL as a function of the number of clients connected to the database. The size of the database is held constant (150MB), and is representative of a database used by 10 clients according to the TPC-W specification. Increasing the number of clients to 150 models extremely high data contention. PostgreSQL sees almost no locking and DB2 sees very little, as TPC-W intrinsically has very little data contention. I/O costs are also low since the database is so small relative to main memory. Thus, CPU is the bottleneck resource for TPC-W [1].

2.7 Scheduling the Bottleneck

As seen in Section 2.6, the bottleneck resource for TPC-C on Shore and DB2 is locks, suggesting that lock prioritization will be effective. Figure 2.4 motivates this point, showing that transactions that do not wait for locks are almost 20 times faster than those that do.

The bottleneck resource for PostgreSQL is usually I/O. While I/O scheduling is outside the scope of this chapter, it is well-known that CPU scheduling may indirectly schedule other resources [5], such as I/O or locks. This is due to the fact that transactions need CPU resources to issue resource requests. Consequently, we investigate whether CPU scheduling is effective for PostgreSQL.

Throughout, we examine *both* lock and CPU scheduling for *both* TPC-C and TPC-W. We have reservations, however, about the TPC-W workload for two reasons: its transactions are (i) extremely simplistic, and (ii) need very little concurrency control. The TPC-C workload, with more complex transaction interactions, is in fact more representative of real-world applications. Note that we do not evaluate any of the

[1] We find that under extreme configurations, lock waiting can be significant for TPC-W as well. Since these configurations depart so much from the TPC-W specifications, we do not consider them here.

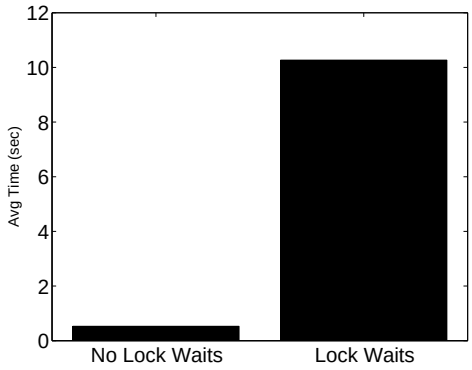


Figure 2.4: Average execution time for TPC-C Shore transactions that never wait for locks compared to those that do, with no prioritization. Think time is 1 second.

scheduling policies on IBM DB2, since it does not support such policies and the source code is unavailable for experimentation.

We begin by defining the specific scheduling policies that we will explore.

2.7.1 Prioritization Workload

Throughout this section, we use a representative 10 warehouse database for TPC-C (1GB) and a 10 client database for TPC-W (150MB). Priorities are assigned to each TPC-C and TPC-W transaction according to a Bernoulli trial with probability 10% of being a high-priority.

TPC-C and TPC-W are *closed loop systems*, where a fixed number of clients alternately wait and execute transactions against the database. The time spent waiting is known as *think time* and models interactive clients interpreting results. The concurrency level can be adjusted by either fixing the number of clients and varying think time, or fixing the think time and varying the number of clients. We find both methods yield similar results. Throughout our experiments we will fix think time at zero and vary the number of clients. The only exception will be for TPC-C experiments, where we will instead vary the think time and fix the number of clients at 300. We choose 300, because that allows us to use think time to vary the number of running clients both above and below the TPC-C-specified 100 clients. The reason that we vary think time for TPC-C prioritization is that the TPC-C clients can consume significant system resources, and thus using a constant number of clients helps reduce variability due to this overhead.

2.7.2 Definition of the Policies

Our scheduling policies are divided into lock scheduling and CPU scheduling policies:

Lock scheduling policies. We first consider non-preemptive lock scheduling policies, where lock holders are never forced to release their locks abnormally due to preemption. Subsequently, we consider preemptive

policies, in which high-priority transactions can preempt low-priority lock holders to acquire their locks. Preemption involves aborting, rolling back, and resubmitting the transaction, adding more work for the DBMS.

The simplest non-preemptive policy, `NP-LQ`, just reorders transactions waiting in the lock queue, and grants locks to high-priority transactions before those of low-priority. This policy has a problem: high-priority transactions moved to the front of the queue must wait for low-priority transactions already holding the lock to complete (known as "excess time" in queueing theory). The case where a high-priority transaction waits for a low-priority transaction is commonly known as priority inversion. Two techniques are commonly used to address the problem, *priority inheritance* [75] and *preemption*.

`NP-LQ-Inherit` is a non-preemptive policy that uses priority inheritance to reduce excess times. The policy is identical to `NP-LQ`, but the priority of each transaction is raised to the highest priority of any transaction that waits for it. For example, when a high-priority query waits for a low-priority query, the low-priority query is changed to high-priority. Thus, a transaction never waits for another transaction with a priority lower than its own. The intended result is that high-priority excess times are reduced, improving high-priority execution times.

`P-LQ` aims to reduce high-priority excess times by preempting transactions currently holding locks needed by high-priority transactions. The policy is identical to `NP-LQ`, but when a high-priority transaction needs a lock held by a low-priority transaction, the low-priority transaction is aborted (known as *preemptive abort*). In practice, two factors reduce the effectiveness of preemption. First, the preempting high-priority transaction must still wait for (part of) the low-priority transaction rollback to complete before continuing. Second, extra work created by preemption potentially slows down other transactions.

CPU scheduling policies. Each of the DBMS considered relies on approximations of (preemptive) generalized processor sharing (GPS), and as a result we do not distinguish preemptive or non-preemptive scheduling of the CPU device itself. We do, however, consider preemption of transactions due to lock conflicts while using CPU prioritization. We call CPU scheduling policies that preempt lock holders preemptive, and those that do not non-preemptive.

The simplest policy, `CPU-Prio`, is a non-preemptive policy that schedules the CPU using weighted GPS. It simply gives more weight to processes working on high-priority transactions. Specifically, for PostgreSQL, we assign UNIX priority nice level -20 to high-priority processes and $+20$ to low-priority processes. For Shore, high-priority threads get "time critical" priority, while low-priority transactions get "regular" priority.

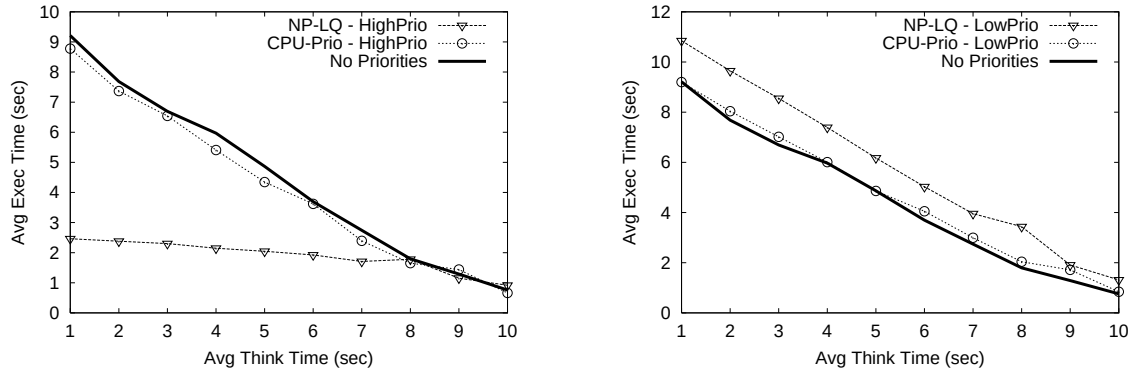
Although the `CPU-Prio` policy prioritizes CPU, the policy may suffer from priority inversions due to locks. A high-priority transaction with high CPU-priority cannot progress if it waits for a lock held by a low-priority transaction. `CPU-Prio-Inherit` is a non-preemptive policy that adds priority inheritance to the `CPU-Prio` policy. The priority of low-priority transactions that block high-priority transactions is raised, thus reducing high-priority excess times.

The `P-CPU` policy is a preemptive policy identical to `CPU-Prio` except that low-priority transactions that block high-priority transactions are preempted and rolled back.

Organization of remaining sections. In Section 2.7.3 we present results for simple scheduling policies without preemption or priority inheritance: `NP-LQ` and `CPU-Prio`, defined above. In Section 2.7.4, we ex-

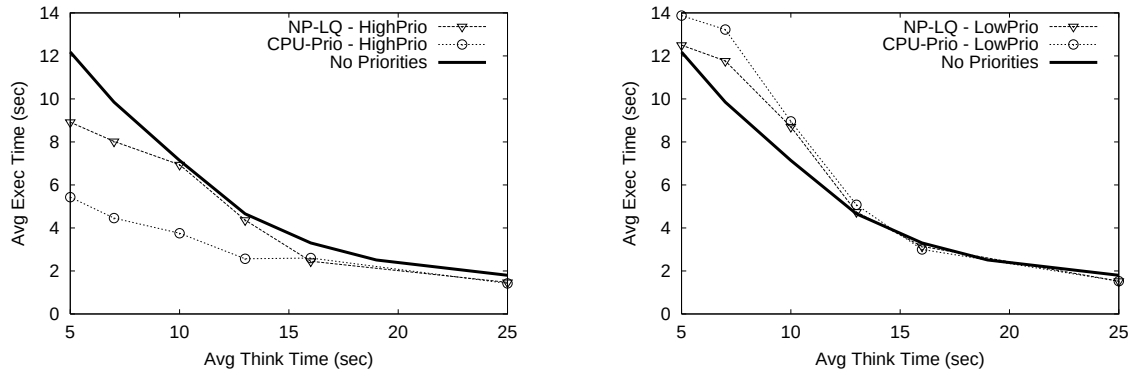
amine policies with priority inheritance: `NP-LQ-Inherit` and `CPU-Prio-Inherit`. In Section 2.7.5, we discuss the preemptive policies `P-LQ` and `P-CPU`.

2.7.3 Simple Scheduling



(a) **Shore** High-Priority

(b) **Shore** Low-Priority

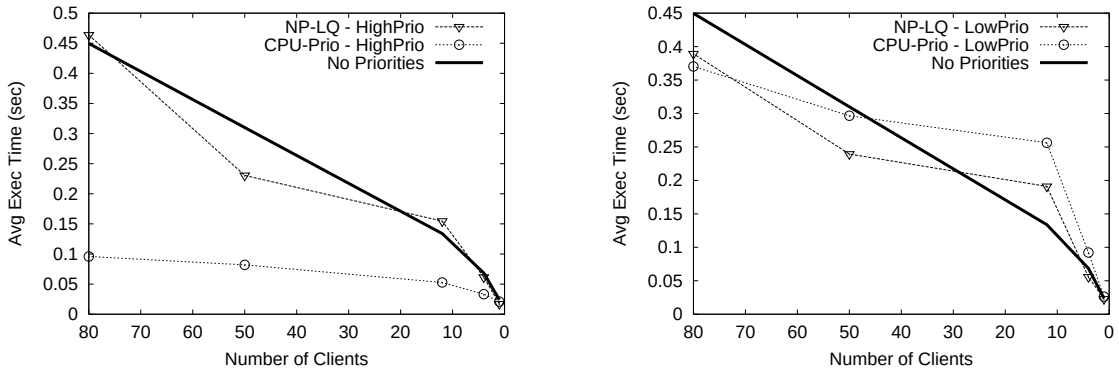


(c) **PostgreSQL** High-Priority

(d) **PostgreSQL** Low-Priority

Figure 2.5: Mean execution times for `NP-LQ` compared to `CPU-Prio` for Shore and PostgreSQL TPC-C with varying contention. Concurrency (load) increases to the left, as think time goes down.

The simple scheduling policies with no priority inheritance and no lock preemption, `NP-LQ` and `CPU-Prio`, exhibit striking differences depending on the workload and the DBMS. Figures 2.5 and 2.6 highlight these differences, showing the performance of high- and low-priority transactions using the policies for TPC-C



(a) **PostgreSQL** High-Priority

(b) **PostgreSQL** Low-Priority

Figure 2.6: Mean execution times for `NP-LQ` compared to `CPU-Prio` for PostgreSQL TPC-W with varying loads. As is the custom in this chapter, high-load (many clients) is on the left, and low-load (few clients) is on the right.

and TPC-W workloads respectively. In all results, the concurrency varies on the X-axis, from high levels of concurrency on the left to low concurrency on the right. Concurrency is controlled either by varying think time (for TPC-C) or, equivalently, by varying the number of clients (for TPC-W).

The best simple scheduling policy for TPC-C depends on the DBMS. For TPC-C running on Shore (see Figure 2.5(a)), `CPU-Prio` does not appreciably improve high-priority transaction execution times. `NP-LQ`, on the other hand, improves high-priority performance by 3.7 times. The penalty to low-priority transactions under both `NP-LQ` and `CPU-Prio` is small (less than 17% for `NP-LQ`) and tracks the "Default" no-priority setting (see Figure 2.5(b)). Lock scheduling is extremely effective for Shore because locks dominate transaction execution times under 2PL.

By contrast, for PostgreSQL, lock scheduling is not as effective as CPU scheduling (see Figure 2.5(c)). Under high loads, `NP-LQ` improves high-priority execution times by a factor of 1.3, whereas `CPU-Prio` improves them by a factor of two. With both policies, low-priority transactions are not significantly penalized (see Figure 2.5(d)). As the think time increases from 5 to 25 seconds, concurrency decreases from 200 to 20 running (non-thinking) clients on average, and the lock fraction of execution times becomes insignificant. As expected, the result is that lock scheduling (`NP-LQ`) is not very effective.

The effectiveness of `CPU-Prio` for TPC-C on PostgreSQL is surprising, given that I/O (I/O-related lightweight locks) is its bottleneck. Due to CPU prioritization, high-priority transactions are able to request I/O resources before low-priority transactions can. As a result, high-priority transactions wait fewer times (50–90% fewer) for I/O, and when they do wait, they wait behind fewer transactions (30% fewer). The fact that simple CPU prioritization is able to improve performance so significantly suggests that more complicated I/O scheduling is not always necessary.

For TPC-W, locks are never the bottleneck resource (see Figure 2.3), suggesting lock scheduling will

be ineffective. As confirmation, Figure 2.6 shows average execution times with NP−LQ and CPU−Prio for TPC-W as a function of the number of clients. As expected, NP−LQ does not significantly improve high-priority transactions. CPU−Prio, however, dramatically improves high-priority transaction times by a factor of up to 4.5 under high load (high number of clients) relative to a system with no priorities.

Low-priority transactions, on average, are not significantly penalized by either NP−LQ or CPU−Prio, for all DBMS and workloads studied. This result is important, and consistent with theoretical results: Performance of a small class of high-priority transactions can be improved without harming the overall low-priority performance.

2.7.4 Priority Inheritance

In this section we evaluate the two policies using priority inheritance: NP−LQ−Inherit and CPU−Prio−Inherit, which are extensions of the NP−LQ and CPU−Prio policies, respectively.

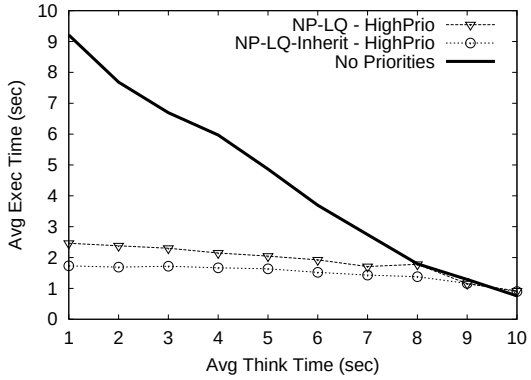


Figure 2.7: NP−LQ−Inherit compared to NP−LQ for Shore TPC-C.

Figure 2.7 compares the policies NP−LQ−Inherit and NP−LQ for TPC-C running on Shore for a range of concurrency levels. We find that adding priority inheritance to simple lock queue reordering (NP−LQ) improves performance by 30%. NP−LQ improves high-priority transaction execution times by a factor of 3.7 relative to a system without priorities, and NP−LQ−Inherit improves execution times by a factor of 5.3.

For TPC-C running on PostgreSQL, adding priority inheritance to NP−LQ offers no appreciable gain in performance, however, priority inheritance with CPU scheduling is beneficial. Figure 2.8 shows CPU priority inheritance improves high-priority transactions by a factor of 6, whereas CPU−Prio only helps by a factor of 2. The significant improvement in performance is due to the fact that the lock holder(s) are sped up, resulting in significantly smaller wait excesses.

Recall from Section 2.7.3 that CPU scheduling (CPU−Prio) is more effective than NP−LQ for TPC-W. Thus Figure 2.9 compares the policies CPU−Prio−Inherit to CPU−Prio for the TPC-W workload on

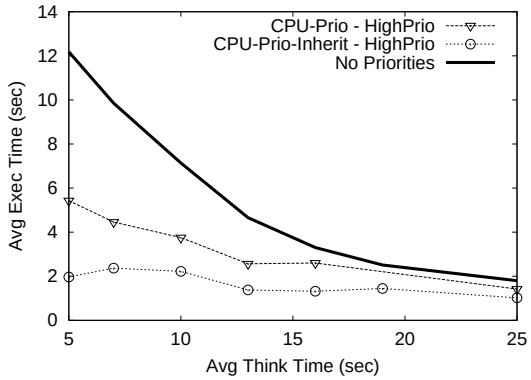


Figure 2.8: `CPU-Prio-Inherit` compared to `CPU-Prio` on PostgreSQL TPC-C.

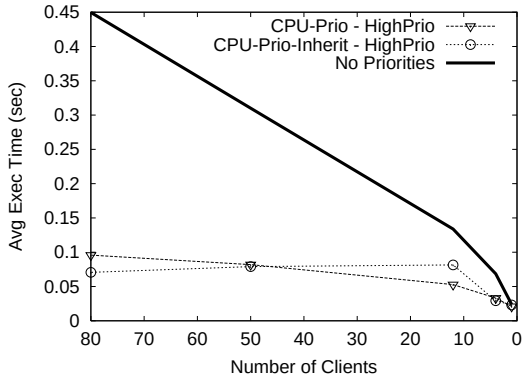


Figure 2.9: `CPU-Prio-Inherit` compared to `CPU-Prio` for TPC-W running on PostgreSQL.

PostgreSQL. We find that there is no improvement for `CPU-Prio-Inherit` over `CPU-Prio`. This is to be expected given the low data contention found in the TPC-W workload; priority inversions can only occur during data contention. Results for low-priority transactions are not shown, but as in Figure 2.5, low-priority transactions are only negligibly penalized on average.

2.7.5 Preemptive Scheduling

Non-preemptive scheduling already provides substantial performance improvements for high-priority TPC-C transactions, using lock scheduling for Shore and CPU scheduling for PostgreSQL. We now focus on whether preemption can provide further benefits. In particular, we evaluate whether `P-LQ` improves on `NP-LQ` for Shore and whether `P-CPU` improves on `CPU-Prio` for PostgreSQL.

With non-preemptive scheduling, high-priority transactions sometimes must wait on lock requests for locks currently held by low-priority transactions (the wait excess). The wait excess time is reduced, but not eliminated, with priority inheritance, which speeds up the low-priority transactions blocking high-priority transactions. Preemptive scheduling (`P-LQ` and `P-CPU`) attempts to eliminate the wait excess for high-priority transactions by preempting low-priority lock holders in the way of high-priority transactions.

We find that preemptive policies provide little benefit over non-preemptive policies. Figures 2.10(a) and 2.10(b) compare the average high- and low-priority execution times for `P-LQ` against `NP-LQ-Inherit` for TPC-C on Shore as a function of think time. High-priority transactions with `P-LQ` improve by a factor of 9.3 whereas `NP-LQ-Inherit` helps only by a factor of 5.3. Low-priority transactions, however, are slowed by a factor of 1.7, which is excessive, making this policy impractical. Figures 2.10(c) and 2.10(d) compare the performance of `P-CPU` to `CPU-Prio-Inherit` for TPC-C on PostgreSQL. Preemption seems to offer no significant benefit or penalty beyond `CPU-Prio-Inherit`.

TPC-W results for `P-LQ` and `P-CPU` are omitted as lock scheduling is ineffective since lock contention is low.

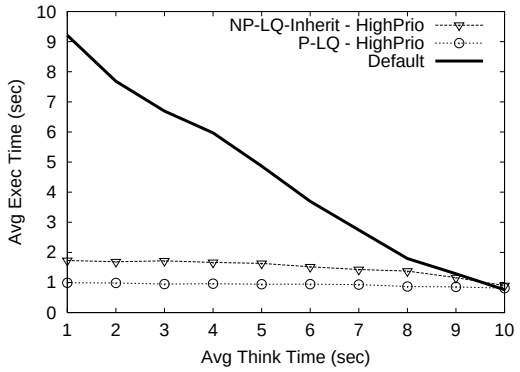
Future extensions to Preemptive Priorities. There are two problems with preemption that limit its effectiveness in our experiments. First, the penalty to low-priority transactions may be excessive. Second, the cost of waiting for a transaction to complete may be cheaper than preemption. As a result, there may be room for improvement in both `P-LQ` and `P-CPU`.

We explore a few other preemptive policies that are more selective about which transactions to preempt. These policies predict a victim transaction's remaining life expectancy and the cost of rolling back the victim to determine whether to preempt or wait. The first idea is to use the number of locks held by the victim to predict its remaining age. If it holds many, it is almost finished, but if it holds few, it is just starting. Second, we use the "wall-clock" age of the victim as a predictor. Although preliminary, we find these are both poor predictors of transaction life-expectancy. It is possible that better predictors can be invented.

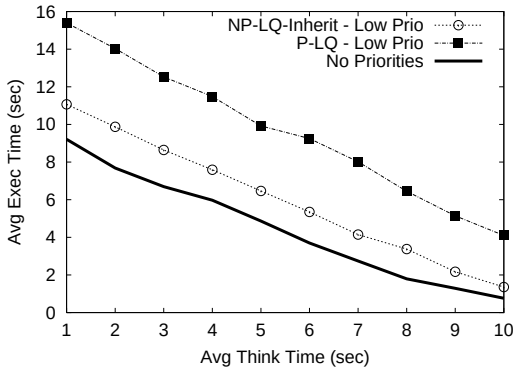
2.8 Conclusion

In this chapter, we develop and evaluate an implementation of transaction prioritization for differentiated performance classes for TPC-C or TPC-W workloads running on traditional relational DBMS.

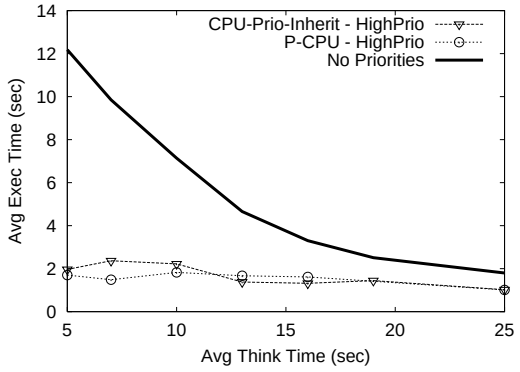
We first identify the bottleneck resource at which priority scheduling is most effective. We divide the lifetime of a transaction into three components: CPU, I/O, and lock wait times. The results are clearly differentiated by workload and concurrency control mechanism. Across a wide range of configurations, the bottleneck for TPC-C running on DBMS using 2PL (Shore and DB2) is *lock waiting*. By contrast, the bottleneck for TPC-C running on MVCC DBMS is *I/O synchronization* for low loads, although locking can dominate at extremely high concurrency levels. For TPC-W workloads, *CPU* is always the bottleneck.



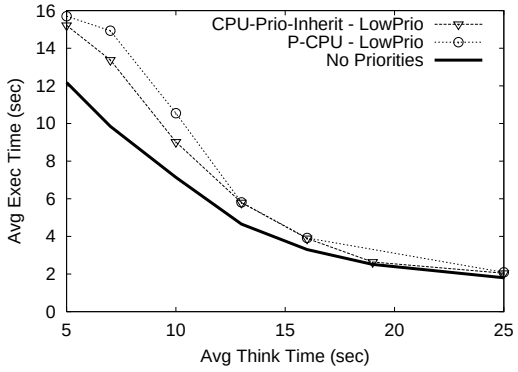
(a) **Shore** High-Priority



(b) **Shore** Low-Priority



(c) **PostgreSQL** High-Priority



(d) **PostgreSQL** Low-Priority

Figure 2.10: Preemptive policies `P-LQ` and `P-CPU` for Shore and PostgreSQL respectively, compared to the best non-preemptive policies for TPC-C.

This bottleneck analysis provides a roadmap for which resources must be scheduled to improve performance. In this chapter, we focus on lock and CPU scheduling to directly or indirectly schedule the bottleneck resource. We evaluate the effectiveness of simple prioritization, priority inheritance, and preemptive abort scheduling, and the results are broken down by workload and concurrency control mechanism.

For TPC-C on 2PL DBMS (Shore), non-preemptive lock scheduling with priority inheritance (`NP-LQ-Inherit`) is most effective. For Shore, high-priority transaction execution times improve 5.3 times, while low-priority transactions are hardly penalized. Priority inheritance and preemption do not appreciably help, and preemption excessively penalizes low-priority transactions. By extension, we believe that these results will hold for IBM DB2 since it has a similar resource breakdown to Shore.

For TPC-C on MVCC DBMS, and in particular PostgreSQL, CPU scheduling is most effective, due to its ability to indirectly schedule the I/O bottleneck. For TPC-C running on PostgreSQL, the simplest CPU scheduling policy (`CPU-Prio`) provides a factor of two improvement for high-priority transactions, while adding priority inheritance (`CPU-Prio-Inherit`) provides a factor of 6 improvement while hardly penalizing low-priority transactions. Preemption (`P-CPU`) provides no appreciable benefit over `CPU-Prio-Inherit`.

For TPC-W on all DBMS, we find that lock scheduling is largely ineffective since transactions rarely wait for locks. CPU scheduling, however, is extremely effective. For TPC-W running on PostgreSQL, we find that the simplest scheduling policy, `CPU-Prio`, is best, and improves performance for high-priority transactions by a factor of up to 4.5. Priority inheritance is not necessary since data contention for TPC-W is almost non-existent.

In conclusion, our results suggest that (i) knowledge of the bottleneck resources is important for determining the best scheduling policies, and (ii) priority scheduling at the bottleneck resource using simple policies can yield significant performance improvements for both TPC-C and TPC-W workloads on real general-purpose DBMS.

The impact to DBMS implementors is that CPU-only prioritization, as currently provided, is insufficient to provide overall transaction prioritization. DBMS implementors must implement more comprehensive prioritization implementations that incorporate other resources, in particular for lock-bound workloads. The next chapter of this thesis will discuss in detail how to best implement lock prioritization for lock-bound workloads.

2.9 Impact

This chapter consists of a performance evaluation of existing scheduling algorithms, and their ability to effect query prioritization and provide performance isolation to high-priority queries in a DBMS. Various scheduling policies are considered based on the device/resource they schedule, and whether they use priority inheritance or preemption. Each of these policies are evaluated for both OLTP TPC-C and transactional web TPC-W workloads running on commercial, open-source, and research DBMS.

This research is extremely applicable in the modern world of DBMS-based web and online services. Companies spend huge amounts of money on high-end DBMS hardware, software, and administration. Reducing this cost by even a small percentage can save companies significant amounts of money. Effective query prioritization is a powerful tool that can reduce hardware and performance tuning costs for DBMS. By prioritizing *important* queries, and giving only best-effort service to other queries, companies can get better performance *exactly where it's needed* with less expensive hardware. Furthermore, query prioritization can be used to selectively improve the responsiveness of time-critical portions of users' workflow, which should make it easier for users to maintain their flow of thought and minimize user frustration. As a result, users will be more satisfied with the service, increasing repeat business and word-of-mouth reputation, which usually translate into higher profits.

Concurrency control is a major performance problem facing modern DBMS-based web and online services. As seen in the bottleneck analysis in this chapter, as the number of clients is increased, workloads become increasingly lock-bound. Many online services have decided to reduce the amount of data-consistency given to users in order to improve performance [26]. Unfortunately, data-consistency problems are a source of frustration to users. For instance, users can purchase a book online, only to find out later that it is no longer available because all copies have been sold. The work on lock scheduling in this chapter sheds some light on the problems surrounding concurrency control and performance, and gives some insight into how scheduling can manage those problems. There is hope that this work can be used to provide users with more data consistency, without hurting performance where it is most important. If this goal can be met, then online services can provide better user satisfaction, which leads to increased profits. This problem will be further addressed in Chapter 3.

2.10 Future Directions

In this chapter, it is shown that scheduling the bottleneck resource is extremely effective at prioritizing high-priority queries in a DBMS workload. It is also shown that scheduling the CPU can be effective even when the bottleneck resource is I/O. The performance isolation that CPU scheduling gives to high-priority queries in I/O-bound workloads is, however, much weaker than when scheduling the bottleneck resource in CPU- and Lock-bound workloads. This suggests that I/O scheduling would be much more effective at scheduling I/O-bound workloads. Unfortunately, I/O scheduling is beyond the scope of this work, it is a field rich with existing scheduling algorithms. One future direction of this work is to evaluate existing I/O scheduling policies and determine which, if any, are effective for DBMS.

Understanding the bottleneck resource is central to understanding query response times, and even more critical to improving query response times. Measuring bottlenecks can have very little overhead, and can essentially be free if time-average statistics are all that is needed. Despite these facts, DBMS do not, in general, provide users with clear and convenient access to this type of data. At the same time, it can be

extremely difficult, if not impossible, for users to collect and verify bottleneck measurements on their own. Getting commercial DBMS to measure bottlenecks is essential for improving the state of the art in DBMS performance analysis.

The work outlined in this chapter focuses on providing two priority classes to DBMS workloads: high- and low-priority. In many real-world situations, additional priority classes, if not a complete continuum of query priorities may be necessary. For instance, administrators may identify high-priority queries from customers, medium-priority queries from employees, and low-priority queries from much less important background tasks. It is important to understand how the results of this performance evaluation may change as the number of priority classes increases.

This research can also be continued by considering both additional workloads and additional DBMS implementations:

Studying additional DBMS, such as Oracle and Microsoft SQL Server play important roles in many commercial systems, although they have not been considered in this performance evaluation so as to keep the scope manageable. It is, however, important to verify whether these DBMS follow the same performance trends observed in the DBMS in this research.

Studying additional workloads will help develop a taxonomy of the workload characteristics that affect DBMS prioritization and scheduling decisions. Each real-world system presents the DBMS with a different workload, and it is difficult to understand how these workloads relate to one another, and how they relate to industry standard benchmarks such as TPC-C and TPC-W. It is important to verify that real-world OLTP and transactional web workloads are comparable to the OLTP TPC-C and transactional web TPC-W workloads seen in this study. In particular, it is important to know whether they exhibit similar bottleneck resources and that the effects of scheduling are similar. Furthermore, it is important to examine other categories of workloads, such as data warehousing, decision support, or ETL workloads and determine their bottleneck and scheduling trends. Such knowledge will help DBMS administrators make better decisions to ensure that the DBMS provides necessary performance.

One of the limitations that becomes apparent when considering the wide range of DBMS and workloads that need to be considered is that experimental evaluation of so many different systems is time consuming and costly. Furthermore, it is difficult to discover the underlying trends which govern performance. In this vain, the DBMS community truly needs performance models which can predict bottlenecks and the effectiveness of various scheduling policies a priori. A small step in this direction is taken in Chapter 4, but the area of DBMS modeling is almost completely open.

Chapter 3

Lock Prioritization in OLTP Applications with POW

Bottleneck Resources		
	2PL	**MVCC**
OLTP TPC-C	Lock	I/O
TransWeb TPC-W	CPU	CPU
Scheduling Policies		
	Best Scheduling Policy	
CPU Bottleneck	CPU prioritization	
I/O Bottleneck	CPU prioritization	
Locks Bottleneck	**No ideal policy**	

Table 3.1: Summary of the key results from Chapter 2: CPU-scheduling provides great high-priority performance isolation when CPU is the bottleneck, and good isolation when I/O is the bottleneck. Existing scheduling policies provide poor isolation when Locks are the bottleneck.

3.1 Background and Overview

In Chapter 2, we studied how to prioritize queries in DBMS. We found that different systems have different bottleneck resources, including CPU, I/O, and Lock. We found that when either CPU or I/O is the bottleneck, simple CPU scheduling can provide significant high-priority query performance isolation and good high-priority query response times. Unfortunately, when Locks are the bottleneck, seen in all cases with OLTP TPC-C workloads running on DBMS using concurrency control based on two-phase locking (2PL), none of the scheduling policies considered in Chapter 2 were very effective (including the lock scheduling policies). These results are summarized in Table 3.1.

Locks are found in almost all DBMS. They are a synchronization element which ensures serialized execution of queries. The basic idea is that when a query needs to access or modify a piece of critical data, it acquires a lock. While that query holds the lock, it is guaranteed that the data remains consistent, and all other queries that want to use or modify that data must wait for the lock to be released. Actual DBMS locks are more sophisticated, and typically many queries can acquire the same lock, as long as their needs are *compatible*. Still, queries have to wait for locks and incur delays whenever their needs are *incompatible*.

In Chapter 2, we saw that a large part of query response times can be the delay experienced by waiting on locks. In the systems considered in this research, these delays can be on the order of seconds or tens of seconds, which is unbearable in commercial applications.

Lock waiting in DBMS is a significant performance problem, and a lot of work has been done to address it. Many people focus on eliminating locking. There are two typical approaches: (i) automatically eliminate locking in the DBMS, and (ii) reduce data consistency requirements at the application level.

Some DBMS, such as Oracle and PostgreSQL, take the first approach to eliminating lock waiting, and use a MultiVersioning Concurrency Control (MVCC) algorithm [11, 66] rather than typical 2PL concurrency control. MVCC makes the observation that DBMS use locking to make sure that, queries see the database as if they executed serially, even when they execute in parallel. Queries care less what serial order they experience, just so long as they see *some* serial order. MVCC eliminates locking by maintaining many copies (one for each update) of data in the database, so that instead of waiting on locks, queries can simply

"travel in time" to access data according to different serial orders. MVCC cannot, however, eliminate all locking in DBMS, and thus cannot eliminate all delays due to locking. Furthermore, there is overhead (in terms of storage space as well as performance) involved in maintaining multiple versions needed for MVCC which can hurt performance in some workloads. There are other approaches to automatically eliminate locking in DBMS, such as optimistic concurrency control [53]. Optimistic concurrency control is not often used, since it can cause significant performance degradation, especially when there are many users in the DBMS [6].

The second approach to eliminate locking is taken by companies and application designers. Companies have realized that they can allow users to access stale and inconsistent data in many situations. This allows companies to use less locking, and reduce delays due to locking [26]. This approach, however, comes with two primary drawbacks: (i) the approach cannot be easily generalized, and relies on significant domain- and application-specific knowledge about what data consistency can be relaxed, and (ii) users do encounter and notice data inconsistency, and this can be quite frustrating. For example, `Amazon.COM` can sell users a book or a product that is no longer in stock, but must be very aware that (i) it cannot double-bill a customer, and that (ii) customers are extremely frustrated when they have to wait an extra week or two to get their purchase.

It is clear that DBMS are stuck with lock delays. Neither of the above approaches can completely eliminate lock delays, and both have significant drawbacks that make them much less attractive than prioritization. Prioritization does not require significant domain-specific knowledge, does not require data consistency requirements to be relaxed, and usually has very little overhead.

There is a lot of research on using lock scheduling to prioritize queries in DBMS. While Chapter 2 found that existing lock scheduling policies are not effective for our systems, those policies *are* effective in other situations. In general, the difference is due to the fact that the workloads and DBMS in the research are very different from the real-world OLTP and transactional web workloads running on conventional DBMS considered in this thesis. The prior work on lock scheduling is summarized and discussed further in Section 3.4.

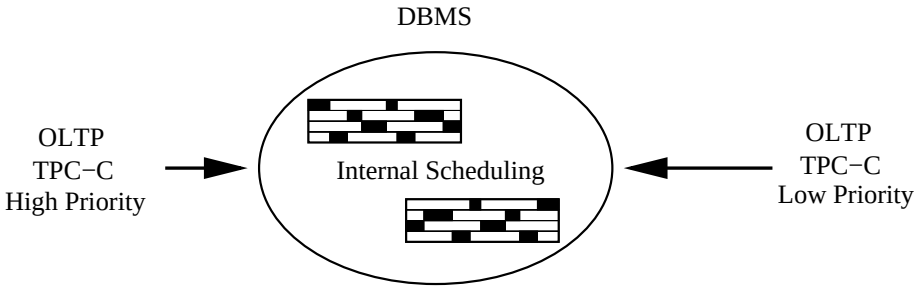


Figure 3.1: The system configurations considered in this chapter: OLTP Transactional Web workloads with high- and low-priority queries, sharing a DBMS. High-priority queries are prioritized using internal prioritization.

This chapter examines systems of the form illustrated in Figure 3.1, and described in detail in Section 1.7. A DBMS is shared between two OLTP (TPC-C) query workloads, one high- and one low-priority. Internal prioritization is used to prioritize and provide performance isolation to the high-priority queries. This scenario arises in many commercial systems, but is of particular interest in the area of e-Commerce. It

is extremely common for e-Commerce applications to have some users which are more important than others, either because (i) users pay for "gold service" and better performance, or (ii) the company recognizes that when certain users get better performance, they are more satisfied, and are more likely to spend more at their site.

The central issue addressed in this chapter is how to provide performance isolation to high-priority queries in a DBMS when the workload is lock-bound. This addresses the primary limitations of preemptive and non-preemptive lock scheduling seen previously in Chapter 2: (i) While preemptive lock scheduling gives good performance isolation to high-priority queries (they have low response times), it starves low-priority queries and hurts overall throughput. (ii) While non-preemptive lock scheduling does not starve low-priority queries and has good overall throughput, it does not provide good performance isolation to high-priority queries (they have high response times).

This chapter makes two main contributions: (i) An in-depth *statistical analysis* of the performance and locking behavior of lock-bound OLTP TPC-C workloads and preexisting lock scheduling algorithms. (ii) A new lock scheduling algorithm, called *Preempt-On-Wait (POW)* that uses preemption *selectively* to give high-priority queries good performance isolation (low response times) without excessively penalizing low-priority query performance or overall throughput. The development of POW is based on the preceding statistical analysis, by addressing the specific problems that arise in preexisting lock scheduling algorithms.

The primary questions that are answered in this chapter are as follows:

(i) *What factors cause preemptive lock scheduling to starve low-priority queries?*

(ii) *What factors prevent non-preemptive lock scheduling from providing sufficient performance isolation to high-priority queries?*

(iii) *Can the use of selective preemption give good performance isolation to high-priority queries without starving low-priority queries?*

(iv) *What condition(s) should be used to decide when to preempt low-priority queries?*

3.1.1 Statistical Analysis

Preexisting lock scheduling policies have serious limitations making them sub-optimal for providing performance isolation to high-priority queries in lock-bound OLTP TPC-C workloads. Preemptive policies hurt low-priority queries' response times too much, causing them to starve. Non-preemptive policies do not provide enough performance isolation to high-priority queries.

The first half of this chapter focuses on identifying the workload and system characteristics that cause preemptive and non-preemptive scheduling each to be sub-optimal. The analysis is done by instrumenting the Shore lock subsystem, and implementing the OLTP TPC-C workload and executing it on the Shore DBMS storage manager with different lock scheduling algorithms. Statistics are then collected describing how queries spend their time in the lock subsystem. Although IBM DB2 cannot provide comparable statistics, parallels are drawn between Shore and IBM DB2 that suggest that query execution in each system are comparable (the similarity between these systems is well-established in Chapter 2 and by others [7]).

Key Idea

The key idea used in the analysis of locking in OLTP TPC-C is to build a statistical model that describes both how queries wait for locks under non-preemptive lock scheduling policies and how queries are preempted under preemptive lock scheduling policies.

The key questions that need to be answered to understand how queries wait for locks are as follows:

(NP.i) *How many lock requests do transactions wait for?*

(NP.ii) *How long are lock waits?*

(NP.iii) *How long do queries wait for current lock holders, versus for other waiting queries?*

(NP.iv) *How much does waiting for a lock affect response time?*

The key questions that need to be answered to understand how `PAbort` preempts queries are as follows:

(P.i) *Are preemptions too expensive, due to rollback?*

(P.ii) *Are there too many preemptions?*

(P.iii) *How much work is lost due to preemptions?*

The DBMS is studied while running different lock scheduling policies to answer the above questions in each case. The lock scheduling policies that are considered are (1) no lock scheduling (`Standard` scheduling), (2) the naive preemptive lock scheduling algorithm that preempts all low-priority queries that block high-priority ones (`PAbort`), and (3) non-preemptive lock scheduling with priority inheritance (`NPrioinher`). Once the above questions are answered for the DBMS under each of these policies, one has enough information to make intelligent decisions regarding lock scheduling to provide performance isolation to high-priority queries.

Answering these questions requires only basic instrumentation of the lock manager, and requires very little overhead.

Summary of results

When analyzing `NPrioinher`, to determine why non-preemptive lock scheduling does not provide sufficient high-priority performance isolation, I find the following answers to the first set of key questions above:

(NP.i) Queries may wait for as few as 0 or as many as 550 lock requests, but over 99% of all queries wait for 2 or fewer lock requests, and this holds for all three lock scheduling polices.

(NP.ii) Lock waits themselves are rather long, making up 40%-50% of high-priority response times.

(NP.iii) When `NPrioinher` queries wait for a lock, the performance penalty almost always comes only from current lock holders, and not from waiting for other waiters in a lock queue.

(NP.iv) Query response time conditioned on the number of lock waits a query experiences reveals that if the query waits for one lock, it is 6 times slower than if it waits on none. If the query waits for more than one lock, then its response time is 17 times slower than if it waits on none. Thus, *queries that do not wait on locks are expected to complete very quickly.*

The key observation is that a query almost always only waits for a couple locks, and those waits will almost always be short. Statistically speaking, the only time those waits are long is when the current lock holders are themselves stuck waiting for another lock. This leads to the conclusion that it is OK for a high-priority query to wait for a low-priority query if and only if that low-priority query does not wait for a lock.

Analysis of `PAbort` and the reason it starves low-priority queries reveals the following answers to the second set of key questions above:

(P.i) Rollback costs in `PAbort` are relatively large (0.5 seconds) in comparison to the target *in isolation* high-priority query response time (1 to 2 seconds).

(P.ii) The number of times any individual low-priority query is preempted is relatively low. 80% are never preempted, 92% are preempted no more than once, and 97% are preempted no more than twice.

(P.iii) The amount of work lost when a query preempted is very large. A preempted query has already finished between 75% and 90% of its expected response time before being preempted.

The conclusion to draw from this is that the biggest reason that low-priority queries starve under `PAbort` is not that a query gets preempted too frequently, but that when a query is preempted, it loses a huge amount of already-completed work.

These results are used to develop the next major contribution, Preempt-On-Wait (POW).

3.1.2 Preempt-On-Wait (POW)

The second half of this chapter uses the results of the statistical analysis in the first half to develop a new lock scheduling algorithm called Preempt-On-Wait (POW). This algorithm is able to achieve "the best of both worlds": (i) the good performance isolation (low response times) for high-priority queries that preemptive lock scheduling can give, and (ii) ensuring that low-priority queries do not starve, and get relatively good performance, as non-preemptive lock scheduling gives.

Key Idea

The key idea for POW is that non-preemptive lock scheduling is relatively effective, except in a small number of cases in which preemption is truly necessary. Using *selective preemption* only when necessary can ensure that high-priority queries get the performance isolation that they need, while not penalizing low-priority queries too much.

POW relies on the fact that in lock-bound OLTP TPC-C workloads, high-priority queries that wait on low-priority queries do not always get stuck with high response times. If the low-priority query itself does not have to wait for a lock, that low-priority query is expected to complete very quickly, and will not hurt

the high-priority query much. On the other hand, if the low-priority query ever needs to wait for a lock, it is expected to take a very long time to complete. Since the high-priority query must wait for the low-priority query to complete before making progress, the high-priority response time will be hurt greatly.

Thus, the preemption condition used by POW is that a low-priority query Q is preempted if and only if both (i) Q is itself waiting to acquire a lock held by some other transaction, and (ii) there is a high-priority query that must wait to acquire a lock held by Q.

Summary of results

We compare `POW` to two policies: (i) `PAbort`, which naively preempts all low-priority queries on which high-priority queries are forced to wait, and (ii) `NPrioinher`, which uses non-preemptive lock scheduling with priority inheritance. `PAbort` provided the best high-priority performance isolation (without regard to low-priority performance), and `NPrioinher` provided the best high-priority performance isolation without starving low-priority queries out of all the lock scheduling algorithms in Chapter 2.

Experimentally, `POW` achieves the same level of performance isolation for high-priority queries that `PAbort` achieves: a factor of 5.45 times improvement for `PAbort` compared to a factor of 5.60 times improvement for `POW`. Likewise, POW achieves comparable performance for low-priority queries that `NPrioinher` achieves: a factor of 1.36 penalty for `NPrioinher` compared to a factor of 1.16 times for `POW`. Furthermore, `POW` is shown to provide significantly better (a factor of 2 times) performance isolation for high-priority queries than other lock scheduling policies that rely on selective preemption, such as Conditional Restart and Wait Depth Limited policies, can achieve.

`POW` is shown to be able to provide good high-priority performance isolation because it preempts low-priority queries whenever they are likely to drastically slow the high-priority query down. `POW` is shown to provide good low-priority performance because it only preempts a very small number of queries (1% of all low-priority queries, compared to `PAbort`, which preempts 20% of all low-priority queries).

3.2 Organization of this chapter

The remainder of this chapter proceeds as follows:

Section 3.3 introduces and motivates the problem of lock scheduling to achieve prioritization in OLTP and transactional web DBMS workloads. Section 3.4 summarizes the existing work and research on lock scheduling and query prioritization. Section 3.5 summarizes the results from Chapter 2 needed to understand this chapter. Section 3.6 is a performance evaluation of common preemptive and non-preemptive lock scheduling algorithms. These existing lock scheduling policies are each shown to be sub-optimal. Section 3.7 is a statistical analysis of the performance of the existing scheduling algorithms described in Section 3.6, which prevents them from working optimally.

Based on the statistical analysis in Section 3.6, Section 3.8 introduces and develops the Preempt-On-Wait (POW) lock scheduling algorithm, which addresses the limitations and failures of the existing policies. Section 3.8.2 evaluates the performance of POW and Section 3.8.4 explains how POW manages to be effective when other policies are not. Finally, Section 3.9 summarizes the results of the chapter, Section 3.10 discusses the impact of this research and Section 3.11 discusses directions for future work.

3.3 Introduction

Long delays and the accompanying unpredictably large response times [1] are a source of frustration in on-line transaction processing (OLTP) database systems. In many applications, consistently low response times are essential for users. Consider, for example, an online stock market with significant price volatility. A trader issues trade orders based on constantly varying market prices, and any delay creates potential for huge profit loss.

Minimizing delay and its unpredictability is much more valuable for some users than for others. A trader making thousands of large-volume trades a day may be willing to pay more for reduced delays on trades. On the other hand, a trader making only one trade a month may accept much more variable response times. Thus, we divide transactions into two classes: high- and low-priority, based on whether the transaction is issued by a high- or low-paying customer. Our primary goal is to *prioritize high-priority transactions* to execute as if *in isolation* of low-priority transactions, and ensure low-priority transactions do not delay high-priority transactions. Second, low-priority transactions must not be excessively penalized.

Transaction prioritization can be important in countless contexts. In commercial OLTP, for instance, customers who experience many excessive delays may become frustrated, and take their business elsewhere. Giving high-priority service to customers who routinely buy expensive merchandise will maximize the company's profits. As a testament to the importance of transaction prioritization, it is provided in most major commercial DBMS: DB2 offers db2gov and QueryPatroller[45, 20] and Oracle offers DRM [67]. We have previously shown that *CPU scheduling is ineffective* for prioritization in OLTP applications (such as TPC-C), while *lock scheduling is highly effective* [56]. Unfortunately, all the above commercial systems focus on CPU, not lock prioritization. Additionally, there is little research on lock scheduling in fully implemented general-purpose DBMS, as most are analytical or simulation studies, or focus on RTDBMS.

Many open questions remain for lock scheduling in general-purpose DBMS. Of these, we focus on whether the DBMS should use a *preemptive* or a *non-preemptive* scheduling policy. Each type of policy has advantages and disadvantages, and there is no consensus as to which is best. While preemptive policies allow high-priority transactions to reduce lock waiting time by killing other lock holders, rollbacks and re-executions may be too costly. Non-preemptive policies avoid these preemptive overheads, but high-priority transactions may wait for low-priority transactions to complete before making progress.

The first contribution of our chapter is a performance evaluation and in-depth statistical analysis of lock activity in TPC-C, for common scheduling policies. For non-preemptive policies, such as queue re-ordering (*NPrio*) and priority inheritance (*NPrioinher*), high-priority transactions are poorly isolated from low-priority transactions, resulting in variable and high response times. By contrast, preemptive policies (*PAbort*) yield good high-priority performance, but excessively penalize low-priority transactions.

To determine why non-preemptive policies fail to isolate high-priority transactions, we address four questions: (i) How many lock requests do transactions wait for? (ii) How long are lock waits? (iii) How long do transactions wait for current lock holders versus for other waiting transactions? (iv) How do lock waits affect response time? We show that the common policies primarily fail to eliminate *wait excess*: the time spent waiting for current lock holders to release locks. To determine why preemptive policies devastate low-priority transactions, we investigate potential reasons: (i) rollback costs (ii) preemption frequency, and (iii) wasted work per preemption. Surprisingly, we find that most of these issues are largely irrelevant, and wasted work per preemption dominates exclusively.

[1] Response time is defined as the time from when a transaction is submitted until it completes, including restarts.

The second contribution of our chapter is a demonstration that a little-known and unevaluated lock scheduling policy from the field of distributed databases, *Preempt-On-Wait* [68] (*POW*), excels over all the above policies. It combines the excellent high-priority performance of preemptive policies with the small penalty to low-priority transactions typical with non-preemptive policies.

The intuition behind *POW* is that if a high-priority transaction H needs a lock held by a low-priority transaction L, H should only preempt L if L will hold the lock a long time. We find that whether or not L waits in another lock queue is a highly accurate indicator of L's remaining holding time. Thus, *POW* only preempts low-priority transactions that both wait for a lock and block a high-priority transaction.

Our evaluation focuses on the TPC-C OLTP workload with Shore [17] (a modern prototype with transaction management, 2PL, and Aries-style recovery), and concentrates on improving high-priority transaction response times. Basic theory dictates that in all closed systems, like TPC-C, throughput is directly related to response-time.

This chapter presents the first major statistical analysis of locking with priority-scheduling in a fully implemented general-purpose DBMS, and thus incorporates complex system interactions, such as I/O. While Shore is noncommercial, it is important to note that (i) this evaluation could not be conducted using a commercial DBMS due to the lack of source code, and (ii) resource utilizations for Shore have been repeatedly shown to be remarkably similar to that of IBM DB2 [56, 7].

For prioritization to be most effective, the fraction of high-priority transactions should be low. Throughout the chapter, we randomly assign high-priority to 10% of the transactions, and low-priority to the remaining 90%. This is a pessimistically-realistic scenario, and results are similar when the ratio is varied.

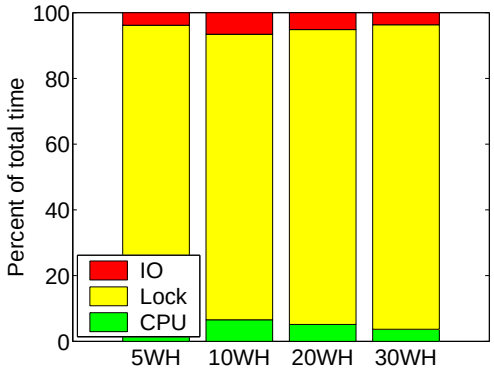
The chapter is organized as follows: In Section 3.4 we describe the prior work on priority scheduling. In Section 3.5 we review existing results showing that lock queues are the appropriate resource to schedule given general-purpose DBMS with lock-based concurrency control. In Section 3.6, we describe our evaluation of the common lock scheduling policies. In Section 3.7, we present the bulk of this work, a statistical profile of locking in Shore TPC-C with priorities. In Section 3.8, we present the *POW* algorithm and its performance analysis. Finally, we conclude in Section 3.9.

3.4 Prior Work

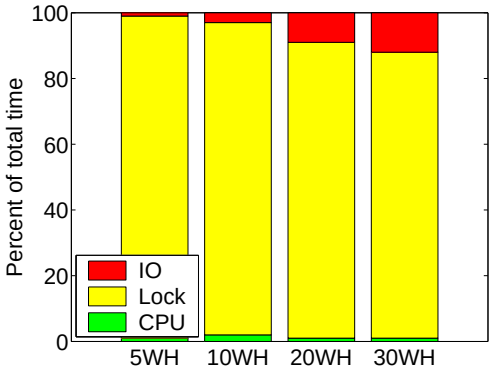
DBMS lock scheduling has been studied for decades, covering countless policies and systems. Most work concerning preemptive and non-preemptive lock scheduling focus on RTDBMS, and are primarily simulation or analytical studies. This is in stark contrast to our focus on general purpose DBMS, OLTP workloads, and full prototype evaluation.

NPrio [2], a non-preemptive policy that reorders lock queues according to priority, is one of the earliest policies considered. Without preemption, however, improvement to high-priority transactions is limited, since they must sometimes wait for low-priority transactions. This problem is known as *priority inversion*, and most other policies' goals are to address it.

NPrioinher, uses *priority-inheritance* [40, 74, 75] to reduce the cost of priority inversions. Low-priority transactions that block high-priority transactions become high-priority themselves. The idea is to reduce high-priority transaction wait times by speeding up the transactions they wait for. If those transactions do not wait on locks, or if too many transactions' priorities increase, the effectiveness becomes unclear. In simulation and prototypes, some research finds that *NPrioinher* is not as effective as *PAbort* in RTDBMS



(a) **Shore**



(b) **IBM DB2**

Figure 3.2: TPC-C Shore and DB2 average I/O, Lock, and CPU resource utilization, relative to total average transaction response time.

[44, 43]. In contrast, other simulation studies find that *NPrioinher* is, in fact, effective in RTDBMS as long as transaction arrivals are non-bursty [2].

PAbort (Preemptive Abort, or Wound-Wait) [40, 43, 68], preempts low-priority transactions that block a high-priority transaction. Since preempted transactions must be restarted, there may be significant extra work into the system, slowing transactions down. In simulation and RTDBMS testbeds, many researchers [40, 44, 43, 64] find that *NPinherit* is not as effective as *PAbort*. As indicated above, this contradicts the conclusions of others [2]. None of these studies consider general-purpose DBMS and workloads.

Much work has been done to improve preemptive policies by reducing the number of preemptions and extra work. In distributed databases, Rosenkrantz et. al. [68] mention *POW* (see Section 3.8) as a possible variation of *PAbort*, in which running transactions are not preempted, but do not implement the algorithm, nor analyze its performance. Conditional Restart (*CR*) and Conditional Priority Inheritance (*CPI*) [43, 44] in RTDBMS estimate the time until low-priority lock holders complete, and preempt if it take too long. We find common estimates, such as the number of locks held, do not work well for TPC-C type workloads.

Wait-depth-limited (*WDL*) [31, 32, 33, 82, 85] policies preempt transactions to keep chains of waiting transactions shorter than a given depth. Running Priority (*RP*) [31, 85] is a common *WDL* policy in which transactions wait only for transactions that are currently running. Though *RP* does not consider priorities, and *POW* is not *WDL*, the preemption conditions are similar (see Section 3.8).

Our work addresses three limitations in the literature:

- Neither preemptive nor non-preemptive policies are strictly superior, and it is difficult to predict which is best for OLTP workloads.
- Most work focuses on RTDBMS, rather than OLTP workloads and general-purpose DBMS. RTDBMS rely on specialized operating systems and workloads that result in different performance tradeoffs than in general DBMS.
- Only a few RTDBMS studies [44] examine locking in fully implemented systems, where complicated interactions can greatly affect performance.

Our prior work [56] is primarily a bottleneck analysis of TPC-C (summarized in Section 3.5), although we also observe the limitations of common non-preemptive and preemptive lock scheduling policies. This chapter supersedes that work, focusing on DBMS using 2PL, with an in-depth analysis identifying the reasons for these limitations. Further, we introduce the *POW* policy which does not suffer these limitations.

3.5 Bottleneck: Locks

Here, we review prior work [56], demonstrating that for TPC-C OLTP workloads on DBMS using 2PL, locks are almost always the bottleneck resource. Specifically, from the perspective of an individual transaction, its response time is dominated by time waiting to acquire locks. As a consequence, I/O and CPU scheduling will be *ineffective* for prioritization, so we focus exclusively on lock scheduling. It is important to note that overall system CPU and I/O utilization are high, as some transactions are always making progress.

We consider TPC-C type workloads on both commercial and non-commercial DBMS, namely IBM DB2 [30], PostgreSQL [52], and Shore [17]. Each of these systems is profiled, counting time transactions

spend waiting for locks and I/O and both consuming and waiting for CPU. (Lock time only accumulates when waiting for locks. After a lock is acquired, the time is spent in CPU, I/O, or waiting for other locks). For both IBM DB2 and Shore, which use traditional 2PL, our results show that, on average, transactions spend more than 80% of their lifetime waiting for locks. We find that this trend is present *over a wide range of configurations*. Only in the most unrealistic configurations are other resources be relevant.

Figure 3.2 shows the average resource breakdown for both Shore and IBM DB2 as a function of database size, measured in TPC-C warehouses. Each warehouse adds 100MB, and the buffer pool is 800MB. The number of concurrent clients is 10 times the number of warehouses, as specified by TPC-C. On average, waiting for locks accounts for most of the response times, and dominates even as the number of clients (load) or the size of the database are varied. While IBM DB2 I/O time increases as the database grows, it is not realistic to run more than 30 warehouses on our limited testbed hardware. In real applications, growing I/O cost is hidden by additional memory and disks.

3.6 Evaluating Lock Scheduling Policies

As seen in Section 3.4, the effectiveness of preemptive and non-preemptive lock scheduling policies cannot be easily predicted. In this section, we experimentally evaluate the behavior of the common policies, and seek to understand their performance trade-offs.

3.6.1 Experimental Setup and Methodology

We focus on the following lock scheduling policies, which are commonly used and referenced in the literature:

Standard: This is the baseline for comparison: transactions are not prioritized.

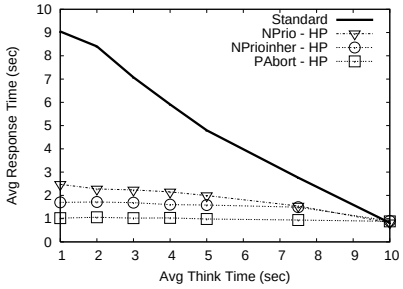
NPrio: Non-preemptive lock queue reordering. When locks are released, waiting compatible transactions are granted the lock in priority-order (from high- to low-priority).

NPrioinher: Non-preemptive lock queue reordering with priority inheritance. Locks are granted as in *NPrio*. Additionally, low-priority transactions that block high-priority transactions become high-priority (for the remainder of their lifetime) to release locks more quickly.

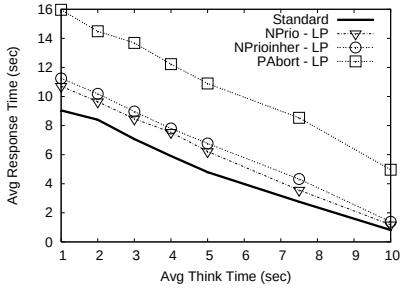
PAbort: Preemptive Abort. A low-priority transaction that blocks a high-priority transaction is always immediately preempted (aborted, rolled back, and restarted).

The above scheduling policies made an appearance in Chapter 2, but under different names. These names are changed here, since we focus on lock scheduling, and to make the terminology clearer. The mapping between names used in Chapter 2 and this chapter are summarized in Table 3.2.

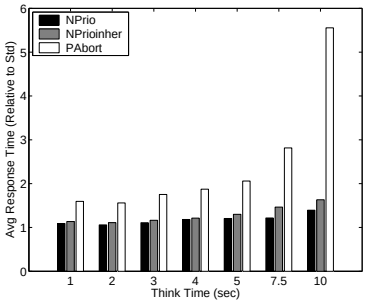
We implement the above lock scheduling policies in Shore and measure their effects on average high- and low-priority transaction response times in a TPC-C workload. The tests are run on a 2.2GHz Pentium 4 with two disks (one for data, one for log), 1GB of RAM and an 800MB buffer pool. Transactions run in serializable isolation level, given the critical nature of many OLTP applications (while weaker isolation levels will result in less locking, this issue is orthogonal to this work).



(a) **High-Priority**



(b) **Low-Priority**



(c) **Overhead**

Figure 3.3: Average TPC-C Shore response times for high- and low-priority transactions as a function of load for *NPrio*, *NPrioinher*, *PAbort*, and *Standard* policies (3.3(a) and 3.3(b)). Aggregate high- and low-priority response time relative to *Standard* (3.3(c)).

Chapter 2	This Chapter
`Standard`	`No Priorities`
`NPrio`	`NP-LQ`
`NPrioinher`	`NP-LQ-Inherit`
`PAbort`	`P-LQ`

Table 3.2: The names of lock scheduling policies used in Chapter 2 and this chapter.

Implementation of these policies in Shore involves sorting lock queues and minor modifications to lock acquire and wakeup functions. The biggest difficulty is forcing the deadlock detector to handle dynamically reordering lock queues. This is an artifact of the original Shore deadlock detection algorithm and should be less of an issue with an independent deadlock detection process, such as in DB2.

10% of the TPC-C transactions are independently and randomly assigned high-priority and the remaining 90% low-priority. The TPC-C code tells Shore the transaction priority, and retries deadlocked and preempted transactions. The database size is 10 Warehouses (1GB on disk), and is appropriate for our hardware limitations (the database size does not greatly affect the lock bottleneck [56]).

To vary concurrency (load), we change the arrival process to have 300 clients (rather than the TPC-C-specified 100 clients) and consider a range of client "think times" between submitting transactions. We vary the think time from 10 seconds ("low load") to 1 second ("high load"). This range of think time results in an average number of active clients in the database from about 25 to 250, allowing us to investigate concurrency levels both well below and above the 100 clients specified by TPC-C.

3.6.2 Performance Evaluation

Figure 3.3 depicts transaction response times under the common lock scheduling policies. Figure 3.3(a) shows mean response time for high-priority transactions and Figure 3.3(b) shows mean response time for low-priority transactions. Throughout the chapter, lower think time (left end) indicates higher load.

NPrio improves response times of high-priority transactions relative to *Standard* by a factor of 4 at high load. By comparison, *NPrioinher* improves response times of high-priority transactions by a factor of 5.3 at high load, and *PAbort* improves high-priority response times over *Standard* by a factor of 9. This significant improvement in high-priority response times further confirms that locks are the bottleneck resource. Under low loads, lock waiting time becomes less significant, and all the policies perform similarly.

The story is very different for low-priority transactions. *NPrio* and *NPrioinher* only slightly harm low-priority transactions as compared to *Standard*, increasing response time by a factor of 1.2 at high load. By comparison, *PAbort* drastically hurts low-priority performance, increasing response time by a factor of 1.8 at high load when compared with *Standard* and by much more at low load.

It is interesting to note that in Figure 3.3(a), the high-priority transaction response times increase as a function of load when no priorities are used (*Standard*), but remain relatively stable when using priority scheduling. This artifact is due to the TPC-C arrival process, which uses a fixed number of clients that submit transactions separated by exponential think times (i.e., a "closed system" in queueing theory). As each transaction has probability p of being high-priority, each client is expected to create one high-priority transaction for each $\lfloor 1/p \rfloor - 1$ low-priority transactions. Since low-priority transactions are an order of magnitude slower than high-priority transactions, the fraction of high-priority clients in the system is in fact much smaller than p. As a result, in Figure 3.3, the time-average fraction of high-priority transactions in the system ranges between 1.1% and 7.7% for *NPrio* (with absolute values ranging from 3 to 2.3 on the average), with similar numbers for the remaining priority policies. As the load increases, there are more and more low-priority transactions, but only a few high-priority transactions, resulting in relatively stable high-priority response times and degrading low-priority performance.

While *PAbort* appears to offer significant benefits to high-priority transactions (factor of 9 improvement) its penalty to low-priority transactions is too high, making it inappropriate for real DBMS. At the same time, while *NPrioinher* does well for both high- and low-priority transactions, its inability to do as well as *PAbort*

for high priorities is discouraging. The primary disadvantage of preemptive scheduling in *PAbort* is the fact that it introduces extra work into the system (rollbacks and re-execution of preempted transactions). It is important to understand exactly how much extra work is created.

One might think that prioritizing transactions does not affect the overall average response time (aggregated over high- and low-priority transactions), but simply provides better response time for high-priority transactions in exchange for worse response time for low-priority transactions. This is not necessarily true however for policies like *PAbort* which introduce significant overhead. Figure 3.3(c) studies the overhead incurred by all the common prioritization policies. Here the response times of the policies are shown normalized by the response time for *Standard* (that is they have been divided by *Standard*'s response time). An overhead of 1 (on the y-axis) indicates that the policy's average transaction response time (aggregate over high- and low-priority transactions) is the same as *Standard*, and the policy has not slowed the overall system down. The non-preemptive policies *NPrio* and *NPrioinher* have low overhead. However *PAbort* has overall average response times 1.5 to 6 times greater than *Standard*, indicating a huge overhead introduced due to preemption. The reason that preemption performs worse under low loads is due to the fact that transactions complete and release locks faster under lower loads, while rollback costs remain about constant (discussed in Section 3.7.2).

3.7 Statistical Profile of TPC-C Locking

In this section, we examine several hypotheses to explain the behavior of *PAbort* and *NPrioinher*, and test these hypotheses using empirical statistical measurements of the system. We will determine first why non-preemptive high-priority performance is not as good as in *PAbort*, and second why low-priority performance under *PAbort* deteriorates.

3.7.1 High-Priority Performance under Non-Preemptive Policies

There are four questions that must be answered to understand why preemptive policies are superior to non-preemptive policies in improving high-priority response times. (i) How many lock requests do high-priority transactions wait for? (ii) How long are the lock waits, and how do they contribute to high-priority response times? (iii) How much lock waiting is attributed to current lock holders? (iv) How much do lock waits contribute to response times?

(i) How many lock requests do high-priority transactions wait for? Shore TPC-C transactions make between 0 and 550 lock requests, depending on the type of the transaction (e.g.: New Order, Payment, etc). Understanding the fraction of these lock requests that are forced to wait will determine the flexibility available to lock scheduling policies.

Figure 3.4 shows the probability distribution on the number of times transactions wait for locks under each of the common scheduling policies. While the distribution is shown for high-priority transactions, the distributions for low-priority transactions is similar. Over 99% of the transactions wait for fewer than 3 lock requests, while fewer than 1% wait for 3 or more lock requests (Figure 3.4 truncated at 4 for clarity, as all other probabilities are practically zero).

Interestingly, the number of lock waits for non-preemptive policies does not change significantly as a function of the policy. Preemptive scheduling changes the distribution slightly, as preempting transactions

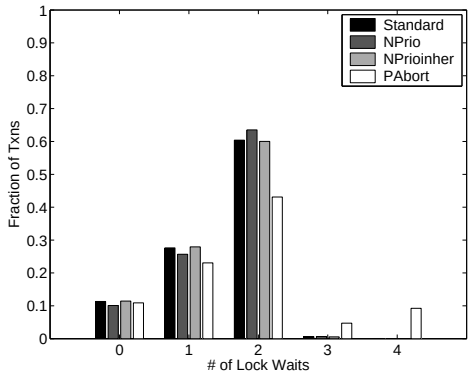


Figure 3.4: Distribution on the number of times that high-priority transactions wait for a lock under common lock scheduling policies (Similar for low-priority transactions). The probability of waiting for more than four locks is practically zero in all cases, and are not shown here for clarity.

reduces the expected number of locks held in the database, reducing contention. None of the policies try to directly reduce the number of times high-priority transactions wait on locks, which may involve knowledge of future lock requests. While reducing the number of lock waits may be an effective strategy to improve high-priority transactions, we only focus on reducing lock wait times once they occur.

(ii) How long are lock waits? The fact that high-priority transactions wait only for a few locks suggests an answer to our second question: individual lock waits are very long. For confirmation, we examine the average time that a transaction waits when it waits for a single lock request. We refer to this time as *QueueTime*, measured from when the transaction initiates the lock request until it is granted. Note that for preemptive policies, *QueueTime* includes preemptions, in which case it is made up of the time needed to preempt the transaction(s) holding the lock, until the preempted transaction(s) release the lock.

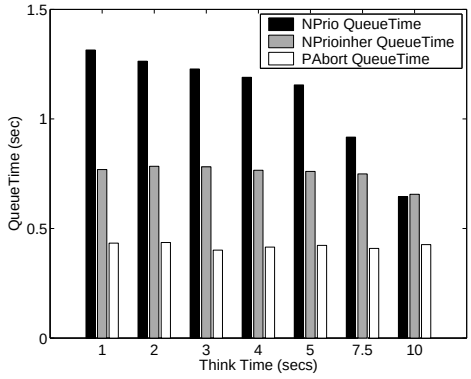


Figure 3.5: Average high-priority *QueueTime* for *NPrio*, *NPrioinher*, and *PAbort* as a function of load (think time).

Figure 3.5 depicts the *QueueTime* experienced by high-priority transactions for *NPrio*, *NPrioinher*, and *PAbort*. On average, high-priority *QueueTime* makes up 40 – 50 % of the the high-priority response time for all policies. Since 25% of transactions wait once and 40-60% wait twice, transactions are expected to include one or two *QueueTimes*, slowing the transactions considerably. Part of the reason that *PAbort* outperforms *NPrioinher* is that its *QueueTime* is only half as long.

(iii) How much lock waiting is attributed to current lock holders? It is important to understand *QueueTime* in more detail, because, as we have seen, long *QueueTimes* prevent non-preemptive policies

from sufficiently improving high-priority response times. Under non-preemptive policies, a transaction's *QueueTime* is comprised of two components: (i) *WaitExcess*, the time from when the lock request is made until the first transaction waiting for the lock is woken and acquires the lock, and (ii) *WaitRemainder*, the time from when the first waiter acquires the lock until the lock request is finally granted. Intuitively, *WaitExcess* is the time that a transaction waits for current holders to release the lock, and *WaitRemainder* is the time the transaction waits for other transactions in the queue with it. The question we want to address is which of *WaitExcess* or *WaitRemainder* is most responsible for high-priority *QueueTimes*.

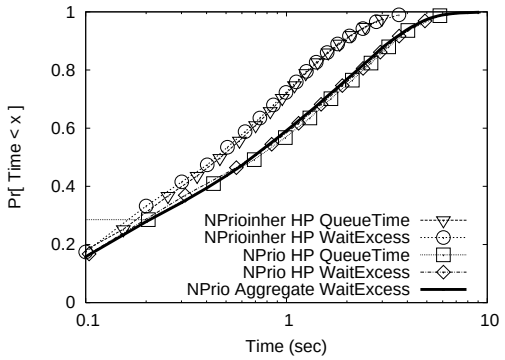


Figure 3.6: CDF of high-priority *QueueTime* and *WaitExcess* for *NPrio* and *NPrioinher* for high load along with aggregate high- and low-priority *WaitExcess* for *NPrio*.

Figure 3.6 compares the probability distributions for high-priority *QueueTime* and *WaitExcess* for *NPrio* and *NPrioinher* under high load. The two leftmost (upper) overlapping lines are *NPrioinher* high-priority *QueueTime* and high-priority *WaitExcess*. The three rightmost (lower) overlapping lines are *NPrio* high-priority *QueueTime*, high-priority *WaitExcess*, and overall average *NPrio WaitExcess*.

The fact that the high-priority *QueueTime* distribution is exactly the same as high-priority *WaitExcess* proves that high-priority transactions never wait behind other transactions in the queue, and only wait for the current lock holder. Figure 3.6 also demonstrates that priority inheritance (*NPrioinher*) reduces *WaitExcess* by boosting the priority of the current lock holders. Additionally, high-priority *NPrio WaitExcess* is identical to overall average *NPrio WaitExcess*, reflecting the fact that *NPrio* does not improve the response time of current lock holders.

Since high-priority *WaitRemainder* is effectively zero for the non-preemptive policies, the only remaining issue affecting high-priority performance is *WaitExcess*. While priority inheritance can help reduce *WaitExcess* by speeding up lock holders, there are limits to its effectiveness, and no clear way to extend the policy to improve its *QueueTimes* further.

(iv) How much do lock waits contribute to response times? Thus far we've seen that long response times can be attributed to waiting on a few locks with large *WaitExcess* times. We now ask how exactly the response time is correlated to the number of locks that a transaction waits on. Figure 3.7 depicts the average response time of a transaction as a function of the number of times the transaction waits for a lock request, for the *Standard* policy and high load (1 second think time). The average response time of transactions

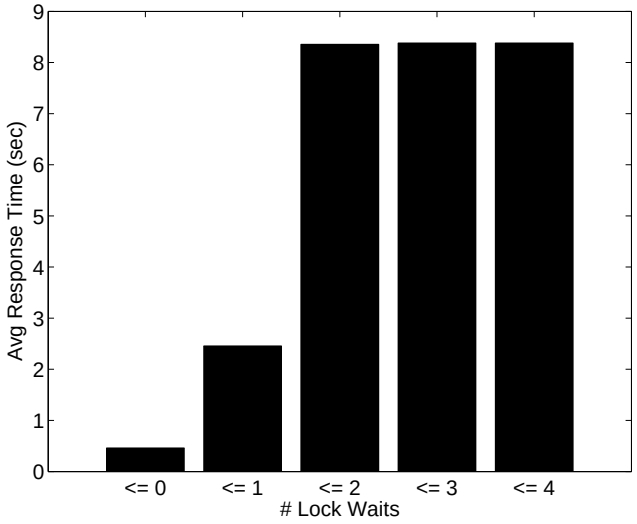


Figure 3.7: Average transaction response time as a function of the number of times a transaction waits under high load, when using the *Standard* policy.

that never wait for locks is a factor of 18 smaller than the overall average response time. In addition, these transactions complete faster than the mean high-priority response time under both *NPrioinher* and *PAbort* (a factor of 3.7 and 2.2 improvement respectively). *This statistic shows that an accurate predictor for the length of a transaction's remaining execution time is whether the transaction is about to wait for locks or not.* A desirable feature of this predictor is that it has low overhead, as it requires no history or bookkeeping in order to provide an estimation.

In conclusion, in this section we show that for TPC-C workloads, while high-priority transactions acquire numerous locks during their lifetime, they are forced to wait on very few (almost always less than 3 waits). This suggests, and we confirm, that the time spent waiting for these blocking lock requests comprises a large portion of high-priority transaction response times. We also show that high-priority transactions almost never wait for other transactions in the queue with them, and just wait for the current holders of the lock to release them. Finally, we show that those transactions which wait for one or fewer locks (40% of all transactions) have extremely short response times.

3.7.2 Low-Priority Performance under Preemptive Policies

While high-priority response times under *PAbort* are very promising, the effect of *PAbort* on low-priority transactions is disastrous. Our goal is to examine the statistical evidence to determine exactly the cause and significance of this problem.

In this section, we examine the well-known penalties for preemption that lead to poor performance: (i) the cost of rolling back transactions, (ii) the number of times transactions are preempted, and (iii) the work lost executing transactions that are subsequently preempted. We show that (iii) is almost the exclusive reason for poor low-priority performance.

Individual rollback costs in *PAbort* have two primary consequences. First, rollbacks delay both the preempting high-priority transaction and the preempted low-priority transaction(s). To ensure ACID properties, a high-priority transaction cannot immediately acquire the lock of a transaction it preempts, but must wait for the preempted transaction to rollback and release the needed lock. The high-priority transaction need not wait for the entire rollback, but only until the needed lock is released. Typically, low-priority transactions are not resubmitted until the rollback is complete. Second, rollbacks require DBMS CPU and I/O resources to clean up the preempted transaction which could otherwise be used for other transactions, potentially slowing down transactions overall.

We find that transaction rollback costs average about .5 seconds over all loads. This cost is nontrivial relative to the cost of a high-priority transaction. By contrast it is insignificant for low-priority transactions, which take between 5 to 16 seconds on average. It should be noted that in optimized commercial systems, rollbacks should be even less significant.

The next question is whether there are simply too many rollbacks, in which case the total cost of several small rollbacks may be significant. We find, however, that this is not the case. Figure 3.8 shows the probability distribution on the number of times a transaction is preempted by *PAbort* under high load (1 second think time). About 80% of transactions are never preempted. For those that are preempted, the trend is approximately geometric with about 12% being preempted once, 5% twice, 2.5% three times, etc. On average, the number of preemptions (and rollbacks) per transaction is less than 0.4, and the expected cost of rollbacks overall is not large relative to the average low-priority response time. Something else must hurt the low-priority response times.

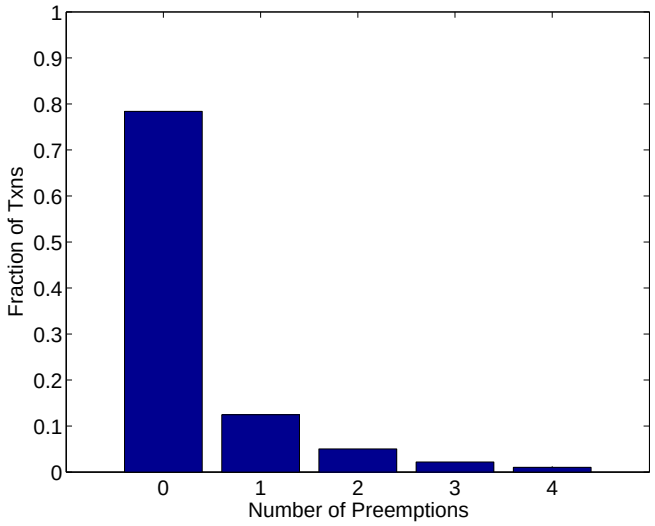


Figure 3.8: Probability distribution on the number of times a transaction is preempted by *PAbort* under high load (1 second think time).

Finally, we examine the amount of work wasted in processing transactions that are eventually preempted. For all loads, the age of a transaction when it is preempted is between 75% and 90% of the length of an average transaction. Thus, a preempted transaction essentially doubles its expected execution cost (assuming independence). The conclusion is that the work lost due to preemption is the most significant flaw of *PAbort*.

3.8 Preempt-On-Wait Scheduling

In Section 3.6 we conclude that high-priority transaction performance is hindered under *NPrioinher* because transactions wait too long for current lock holders. Similarly, low-priority transactions are hurt under *PAbort* because too many transactions (20%) are preempted after completing a significant amount of work. In this section, we describe and evaluate the *Preempt-On-Wait* (*POW*) lock scheduling policy, which combines the best of both worlds: *PAbort*'s good high-priority performance and *NPrioinher*'s good low-priority performance.

Section 3.8.1 describes the *POW* algorithm and its implementation in Shore. Section 3.8.2 demonstrates that *POW* achieves the best of *PAbort* and *NPrioinher*. Section 3.8.3 compares *POW* to several state-of-the-art preemptive policies. Finally, Section 3.8.4 provides a statistical analysis explaining why *POW* meets its performance goals.

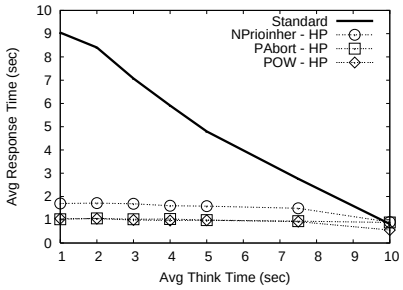
3.8.1 The POW Algorithm

POW is motivated by the following logic: consider a low-priority transaction L that blocks a high-priority transaction H. We have seen that if L is preempted, much work is lost, penalizing low-priority response times. If L is not preempted, its remaining time depends on whether it waits for a lock (in which case its remaining time is long) or doesn't wait for a lock (in which case its remaining time is very short).

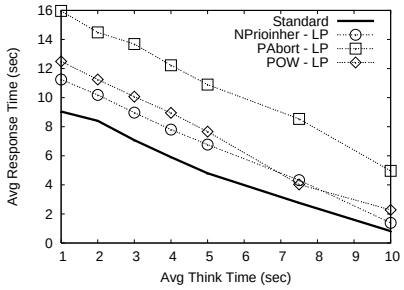
In *POW*, when a high-priority transaction H waits for a lock X_1 held by a low-priority transaction L, L is preempted if and only if L currently, or in the future, waits for some other lock X_2. Additionally, lock queues are reordered as in *NPrio* to ensure that high-priority transactions are first to get the lock when it is released.

The implementation of *POW* for Shore builds on the implementations of *NPrio* and *PAbort* as described in Section 3.6.1. The only additional state needed is a boolean flag $fpow$ for each transaction, which requires *almost no computational overhead*. If H must wait for L, and L is currently waiting for another lock, $fpow$ is set. On all blocking lock acquisitions, if $fpow$ is set, the transaction is aborted.

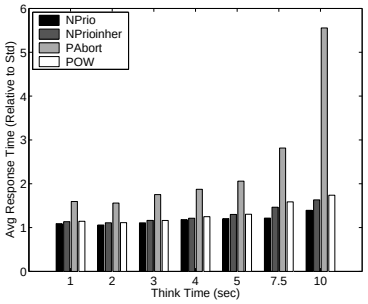
For example, consider that low-priority transactions L_1, L_2, and L_3 all hold a lock X in shared mode. L_1 is currently waiting to acquire another lock, L_2 will need to wait on another lock in the future before it completes, and L_3 will not wait for any more locks before it completes (though it may acquire several more). If high-priority transaction H requests an exclusive lock on X, it will immediately preempt L_1, and set the flag on L_2 and L_3. When L_2 makes its first lock request and is forced to wait, *POW* sees that L_2's flag is set, thus L_2 is aborted. Since L_3 does not block on any more locks, it completes. In this case, H acquires the lock X as soon as all of L_1, L_2, and L_3 have either completed or aborted.



(a) **High-Priority**



(b) **Low-Priority**



(c) **Overhead**

Figure 3.9: Average response time for high- and low-priority transactions for *POW*, *PAbort*, and *NPrioinher* as a function of load (3.9(a) and 3.9(b)). Aggregate high- and low-priority average response time relative to *Standard* (3.9(c)).

	NPrioinher	*PAbort*	*POW*
HP improvement	3.41x	5.45x	5.60x
LP penalty	1.36x	2.27x	1.16x

Table 3.3: High- and Low-priority response time speedup relative to *Standard* policy.

3.8.2 POW Performance Evaluation

Figure 3.9 compares the performance of *POW* with that of the common lock-scheduling policies, as a function of load. *POW* high-priority response times are nearly identical to those for *PAbort* for all loads. Simultaneously, *POW* low-priority response times are nearly identical to those for *NPrioinher*. Therefore, *POW* outperforms both *PAbort* and *NPrioinher* (and also *NPrio*). As the probability of a transaction being high-priority varies from 1% to 10%, the same trend holds.

Table 3.3 shows the high-priority improvement and low-priority penalty under *POW* and the common policies, averaged over the range of think times. *POW*'s improvement to high-priority transactions (a factor of 5.6 improvement over *Standard*) exceeds even that of *PAbort*. *POW*'s penalty to low-priority transactions (a factor of 1.16 above *Standard*) is even lower than that of *NPrioinher*.

As explained in Section 3.6.2, response times of high-priority transactions remain relatively constant as the load increases, and the number of high-priority transactions in the system at any time is relatively constant (between 1.1% and 5.2%).

Figure 3.9(c) depicts the overhead (overall average transaction response time relative to *Standard*) for *POW*. The overhead of *POW* is always comparable with *NPrioinher* and (to a lesser extent) *NPrio*. By comparison, the overhead of *PAbort* is disastrous (See Figure 3.3(c)).

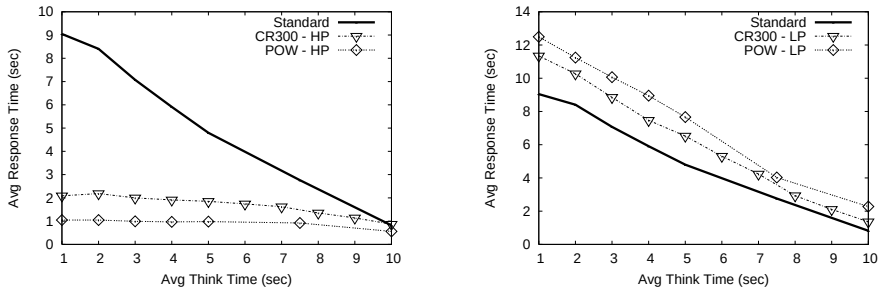
3.8.3 POW vs Other Preemptive Polices

In this section, we compare *POW* to other types of state of the art preemptive lock scheduling policies from the literature: *WDL* (wait-depth-limited) [32] and *CR* (conditional restart). Since *POW* allows long lock chains to form, it is not itself a *WDL* policy. Additionally, its preemption conditions differ from those employed in typical *CR* policies.

First, we consider *WDL1*, a simple wait-depth-limited policy without prioritization. This policy simply preempts transactions to ensure that a lock chain contains no more than 2 transactions. When three transactions wait in a chain $T_1 \rightarrow T_2 \rightarrow T_3$, T_2 is preempted. We find that *WDL1* performs poorly on our workload, particularly under high loads, since it has hot-spots and and high contention. Transactions are preempted too frequently (more than 50 times each) and make no forward progress. Our attempts to extend *WDL1* to respect priorities fail to resolve this problem.

We also consider a variation of the *CR* policy, *CR300*. *CR300* is identical to *PAbort*, except that transactions are given a reprieve time (300ms) to complete before being preempted. By contrast, *CR* preempts transactions immediately, but must make difficult predictions about transactions' remaining times. *CR300* safely avoids this issue, relying on the fact that high-priority transactions wait only for *WaitExcess*, and those *WaitExcesses* are very short (we find 30% are less than 300ms and 50% are less than 500ms). Varying the reprieve time from 100ms to 1000ms does not change performance significantly.

Figure 3.10 illustrates high- and low-priority transaction response times for each of the policies *Standard*, *CR300*, and *POW*. Invariably, the best policy for both high- and low-priority transactions is *POW*. *CR300* performs similarly to *NPrioinher* for both high- and low-priority transactions. *POW* manages to outperform *CR300* by preempting more transactions that greatly slow down high-priority transactions (*POW* preempts 4 times as many under high loads). It turns out that whether a transaction waits for a lock is a better predictor of its remaining time than whether it completes within its reprieve time.



(a) **High-Priority** (b) **Low-Priority**

Figure 3.10: Average response time for high- and low-priority transactions with preemptive policies *CR300* and *POW*.

3.8.4 Explaining POW Performance

In order to understand why *POW* improves high-priority transaction response times as much as *PAbort* without hurting low-priority response times, we conduct a statistical evaluation of TPC-C under *POW*.

Low-Priority Penalty. As determined in Section 3.6.2, the primary reason low-priority transactions suffer with *PAbort* is work lost to transactions later preempted. Two factors affect the amount of this wasted work: (i) the number of preempted transactions and (ii) the age of a transaction when preempted.

We find that *POW* preempts less than 1% of transactions ($\sim$ 265 preemptions), while *PAbort* preempts 20% of the transactions ($\sim$ 5000 preemptions). These figures are fairly constant over all loads. Thus, *POW* allows almost all low-priority transactions that block high-priority transactions to complete during their "reprieve." Thus, only a handful (1%) of transactions are penalized more with *POW* than with *NPrioinher*. The result is that low-priority response times and overhead for *POW* are similar to that of *NPrioinher*.

High-Priority Improvement. *POW* improves high-priority transaction response times because it significantly reduces high-priority *QueueTime*. We consider *QueueTime* in two cases: in the case where the lock holder completes (*QueueTime*|*Wait*) and in the case where the lock holder is preempted (*QueueTime*|*Preempt*).

Figure 3.11 compares the average high-priority *QueueTime* for *POW* and *PAbort*. While *POW*'s (*QueueTime*|*Preempt*) can be large, (*QueueTime*|*Wait*) is similar to *PAbort*'s *QueueTime*. Since *POW* preempts so few transactions, the overall average *POW QueueTime* is similar to the average *PAbort QueueTime*. The variability in *POW*'s (*QueueTime*|*Preempt*) in Figure 3.11 is a result of so few preemptions.

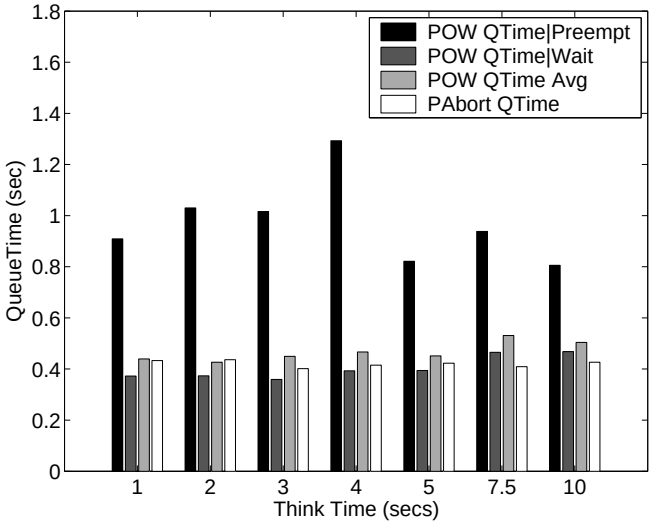


Figure 3.11: Average time for high-priority *QueueTime*, *QTime|Preempt*, and *QTime|Wait* as a function of load.

POW is as good for high-priority transactions as *PAbort* because whenever a high-priority transaction waits for a lock, it waits no longer than it would have if it preempted the current holder(s). In the few cases where *POW* preempts the lock holder(s), the waiting time will be extraordinarily large. Furthermore, the time lost waiting to determine whether to preempt the holders is not very long (2-3 times an average lock wait).

3.9 Conclusion

The goal of this work is to provide user priority classes for OLTP applications, such as TPC-C, using priority scheduling in the DBMS. As Shore (and similarly, IBM DB2) exhibits lock bottlenecks for these workloads, we consequently focus exclusively on lock scheduling. Experimental evaluation of common preemptive and non-preemptive lock scheduling policies in this environment reveals that no policy is clearly superior. The common policies have limited ability to improve high-priority response times without significantly hurting those for low-priority transactions.

Consequently, we formulate a novel and detailed statistical analysis of locking in TPC-C on Shore with the common lock scheduling policies. We draw two primary consequences from this analysis. *First,* with non-preemptive lock scheduling, *WaitExcess* dominates the delays experienced by high-priority transactions, and *WaitExcess* is not greatly reduced by techniques such as priority inheritance (Figures 3.5 and 3.6). Furthermore, if a transaction waits for a *WaitExcess*, its response time is 5-18 times longer than if it does not (Figure 3.7). As a result, while non-preemptive scheduling policies such as *NPrioinher* barely penalize low-priority transactions, they insufficiently improve high-priority transactions. *Second,* for preemptive lock scheduling, though preemption can introduce many overheads, the only relevant penalty is work wasted on transactions later preempted.

The above analysis suggests prioritization policies must exploit the tradeoff between wasted work and *WaitExcess*. To that end, we propose and implement the *POW* lock scheduling policy for TPC-C workloads on lock-based DBMS. *POW* exploits the statistical profile of locking in the workload, combining the excellent high-priority performance of *PAbort* with the good low-priority performance of *NPrioinher*. This is the first application and evaluation of *POW* for OLTP DBMS and workloads. Experimental results show that *POW* improves high-priority transaction response times by a factor of 5.6 on the average, while hurting low-priority transactions by only 16%. Thus, preemption can be effective with a low penalty in traditional DBMS and OLTP applications.

This work has several high-level impacts: First, *POW*-like policies can be used in online and commercial OLTP environments to increase profits, both by enabling service-level agreements and by ensuring good performance and satisfaction for high-profit customers. Second, analytical DBMS performance models may use our statistical profile of TPC-C locking to develop more accurate and tractable models for OLTP workloads. The dominating factors of *WaitExcess* and work lost in preempted transactions allow modelers to ignore irrelevant aspects of the system. Last, our analysis forms a basis for studying other workloads and building a taxonomy of workloads' statistical profiles, invaluable for DBMS algorithm development and tuning.

3.10 Impact

POW lock scheduling provides the ability to provide performance isolation to high-priority queries in DBMS workloads when those workloads are lock-bound. Lock-bound workloads arise in industry-standard OLTP benchmarks (TPC-C) designed to mimic and reflect the performance of real-world OLTP workloads, suggesting many real-world workloads may stand to benefit from POW.

While preexisting scheduling policies are able to provide good high-priority performance isolation in many situations, no policies are especially good with lock-bound workloads. With lock-bound workloads, preexisting policies are all sub-optimal, and are either not effective at providing high-priority performance isolation, or have too many disadvantages (for low-priority query performance or overall throughput). POW fills this vacuum, and provides excellent performance isolation for high-priority queries without excessively hurting the performance of low-priority queries. POW is the first lock scheduling policy to achieve this level of performance in lock-bound OLTP TPC-C workloads.

POW is an essential piece of technology necessary to provide comprehensive query prioritization in DBMS. Query prioritization, as discussed in Chapter 2, has many important ramifications. Most importantly, it can be used to both (i) provide better user satisfaction to important users and/or queries, and (ii) provide better performance *where it is important* with less expensive and less powerful hardware. Both of these accomplishments directly lead to improving companies' bottom lines. By providing better user satisfaction, as in (i), companies increase their market share and revenues by gaining repeat business. By spending less on hardware, as in (ii), companies save on the initial hardware costs, tuning costs, and most likely on energy and cooling costs.

Beyond simple cost savings, POW may be a first step to help improve online applications to provide better service and reduce user frustration. As mentioned in Section 2.9, concurrency control is a major problem facing modern DBMS-based web and online services. It is difficult to provide both good performance and data consistency, and many companies sacrifice data consistency to reduce locking and provide better performance [26]. The lack of data-consistency is frustrating to users, reducing customer satisfaction, which can hurt profits. POW gives companies the option to increase data consistency, and manage the performance penalty that arises due to increased locking and data contention. Paired with knowledge of users' workflow, companies can improve response times for portions of the workflow where users must maintain their flow of thought, which can reduce frustration. Likewise, companies could opt to improve the performance of important customers, paying customers, or customers likely to pay.

3.11 Future Directions

Evaluation of POW on different DBMS implementations and with different workloads will help determine how widely applicable it is. POW is strongly based on the statistical model of locking and preemption in the lock-bound OLTP TPC-C workloads developed in Section 3.6. It is expected that POW will be effective at providing high-priority performance isolation whenever the workload locking behavior is consistent with that statistical model. If the model does not fit the workload, however, it may provide insight into how POW should be changed to provide better performance isolation. Further research is necessary to identify other workload types for which POW is not effective, and to develop alternative scheduling policies to handle these cases.

The statistical model developed in Section 3.6 helps to determine whether or not POW will be effective.

The model is relatively simple, and the data needed for the model is relatively easy to collect by instrumenting the DBMS lock subsystem with little overhead. Thus, the DBMS may be able to automatically (without user intervention) determine whether POW will be effective at providing high-priority performance isolation. The DBMS may choose to use POW only when the workload fits the model described in this chapter, and switch to an alternative lock scheduling policy when it does not. Automatically tuning the DBMS scheduling policies satisfies a growing interest in self-tuning "knob-less" databases, which make DBMS easier to use and administer. Research is necessary to determine the appropriate control mechanism to determine when to switch scheduling policies in such a self-tuning system.

A final direction for continuing the work on POW is to develop a rigorous mathematical analysis of its performance based on queueing theory. Gaining theoretical insight into the performance of POW may lead to better predictions of POW's effectiveness, as well as better understanding of the benefits and drawbacks of POW scheduling. The key challenge here is in integrating the statistical model developed in Section 3.6 into queueing models. The statistical properties of locking in lock-bound OLTP TPC-C workloads is particularly non-Markovian and will be difficult to model using traditional queueing theoretic approaches. Using the statistical properties of OLTP locking measured in this chapter will provide a much easier means by which to model locking than to examine query data dependencies directly. Progress on this front will not only provide better insight into POW, and DBMS, but also help to analyze many other systems with locks.

Chapter 4

Providing Isolation for Mixed DBMS Workloads (IDD)

4.1 Background and Overview

In the previous chapters, we looked at how to use internal prioritization of DBMS devices and resources to effect query prioritization at the DBMS. The goal, of course, is to provide high-priority queries with good performance isolation, giving them good response times even when there are low-priority queries in the DBMS. At different times, we find that different devices and resources within the DBMS are more important for effecting high-priority performance isolation, depending on which resource is the *bottleneck resource*. Sometimes, CPU, I/O, or Locks are the bottleneck, depending on the DBMS implementation or workload. We show that implementing prioritization on the bottleneck resource usually yields the best high-priority performance isolation. Some resources, such as Locks, are particularly difficult to prioritize to good effect, and Chapter 3 addresses and solves that issue.

Internal prioritization is not always applicable. Unfortunately, internal prioritization cannot be used to effect prioritization in many DBMS applications. The primary reason this is so is because DBMS vendors rarely provide internal prioritization (and those that do do not provide comprehensive implementations that are needed). It is almost impossible to use internal prioritization on DBMS applications using legacy DBMS, since those DBMS cannot be easily upgraded. Applications that use modern (non-legacy) DBMS still have to wait for DBMS vendors to decide to implement internal prioritization.

Admission control works where internal prioritization does not. Without using internal prioritization, high-priority query performance isolation and query prioritization can be provided using *admission control*. Admission control (illustrated in Figure 4.1) conceptually sits between the DBMS and (either some, or all of) the users, and limits the number of those users in the DBMS at one time. The idea is to limit the number of low-priority queries in the DBMS, to limit the *performance impact* they have on high-priority queries. Admission control can easily be implemented independent of and *external* to the DBMS and the users. Thus, admission control is applicable even when the DBMS or the users are not specifically designed to support it.

Admission control is difficult to configure. Unfortunately, it is difficult to determine how to configure admission control to achieve the desired high-priority performance isolation. The primary question is how to set the low-priority MultiProgramming Level (MPL), defined as the number of low-priority queries allowed in the DBMS at any one time. Clearly, by limiting the number of low-priority queries in the DBMS to zero, one effects *complete* performance isolation to high-priority queries. The main drawback, however, is that the low-priority queries *starve* when none are admitted. As the number of low-priority queries increases, low-priority queries no longer starve (and low-priority performance improves), but one expects that the performance isolation for high-priority queries should decrease as a result (and high-priority performance should suffer). The key problem is that it is difficult to *predict* and *quantify* how changes to MPL affect the performance of either high- or low-priority queries (These problems are due to the factors discussed in the introduction Section 1.5).

Admission control is hard to configure because CPU performance is unpredictable. This chapter will address the issues that make configuring admission control to provide performance isolation difficult. It will be seen in this chapter that one of the underlying issues is that CPU performance can change drastically depending on the number of low-priority queries that are admitted into the DBMS. It turns out that two factors contribute to this issue: When the number of low-priority queries increases (the low-priority MPL increases), both (i) queries require more CPU service, and (ii) the CPU becomes less efficient, and it takes more time to provide queries with the same amount of service. The contribution of these factors can reduce the strength of the CPU(s) by a factor of 2 or more, and thus they are extremely significant. Neither of these factors are predicted by preexisting models (such as from queueing theory), and are *essential* to

understanding performance of the high- and low-priority queries.

No research addresses admission control for prioritization. While admission control is sometimes used in practice to limit the impact that one workload has on another, there is very little research that addresses that issue directly. Almost all research on admission control focuses on limiting the total number of all queries in the DBMS, so as to limit thrashing on specific system resources, such as locks (as seen by Thomasian et al. [83]). The primary goal of this chapter is specifically to determine how to use admission control to effect query prioritization, *quantifying* the effectiveness via both analysis and experimental performance evaluations with real-world systems.

Two key ideas are employed in this chapter. First, is that we evaluate the performance isolation provided by admission control on a wide range of real-world workloads (parameterized TPC-W workloads). This reveals interesting performance issues that would be difficult to discover without observing individual workloads. Second, is that, to model the DBMS performance, we start with simple queueing models for which analysis is tractable, and then we augment them to reflect the performance issues that are observed, maintaining tractability. This chapter combines these two ideas, one of performance evaluation, and the other of modeling and analysis, to produce the Isolated Demand Decomposition (IDD) approach, which accurately models DBMS performance with admission control.

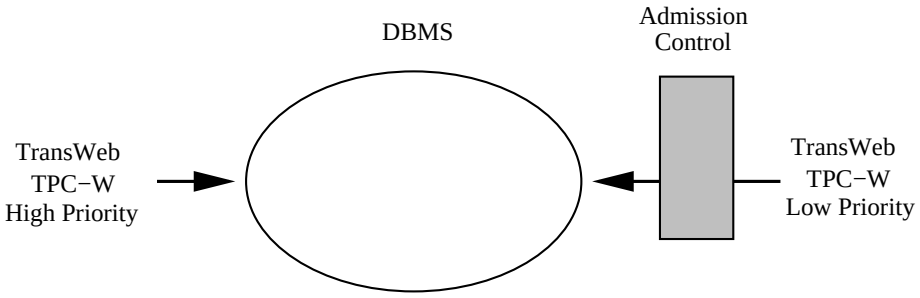


Figure 4.1: The system configurations considered in this chapter: Transactional Web workloads with high- and low-priority queries, sharing a DBMS. High-priority queries are prioritized by using admission control to limit the number of low-priority queries in the DBMS at any time.

This chapter examines systems of the form illustrated in Figure 4.1, and described in detail in Section 1.7. A DBMS is shared between two Transactional Web (TPC-W) query workloads, one high- and one low-priority. Admission control is used external to the DBMS in order to provide performance isolation to the high-priority queries. This chapter shows how to quantify and predict the performance isolation that admission control can give to the high-priority queries.

The above scenario arises in many commercial systems, but is of particular interest when companies have to mix different and independent workloads on the same DBMS. This is of particular interest when adding new applications, functionality, users, or services to an existing DBMS. Companies spend a lot of time and money setting up and running their DBMS application services. Changes to those systems that cause additional query workload to be sent to the DBMS risk hurting the performance of existing services, and hurting their primary, established business. Using admission control to prioritize existing workloads over new workloads can help reduce the risk that existing business will be hurt during such upgrades.

It should be observed that the scenario consisting of two Transactional Web TPC-W workloads sharing the DBMS is surprisingly rich and complex, and presents many unexpected performance-related issues. Restricting the scope to address TPC-W workloads does not significantly limit the impact of this work, but does make the evaluation more tractable. In fact, the analysis methods developed within this chapter are expected to be easily applied to many other scenarios and systems.

This chapter has three main contributions: (i) We establish that real-world systems can exhibit a non-intuitive performance problem, which we call *The Hump*, and which causes very large query response times. The hump is not only non-intuitive, but is also not predicted by conventional queueing-theoretic approaches to modeling DBMS performance. This is described below in Section 4.1.1. (ii) We perform a statistical analysis of DBMS performance, and identify the underlying systems issues that cause the Hump. This is described below in Section 4.1.2. (iii) I develop a modeling approach called IDD that adapts conventional queueing models to correctly incorporate the underlying causes of the Hump. IDD accurately predicts DBMS performance, and compensates for the fact that DBMS violates several key assumptions that queueing-theoretic models rely on. IDD is described further in Section 4.1.3.

The primary questions addressed in this chapter include:

(i) *What factors cause the Hump? Or, in other words, what factors cause Locals to receive good response times even with many Federators in the DBMS at the same time?*

(ii) *How do you model the DBMS and predict query performance when the DBMS is shared by Locals and Federators?*

(iii) *How do you determine how many Federator queries can be admitted into the DBMS so as to achieve specific performance goals for Locals?*

4.1.1 Performance Evaluation: The Hump

The first contribution of this section consists of a performance evaluation study of two TPC-W workloads running on the same DBMS: one high-priority, and one low-priority. The high-priority TPC-W workload (called the Local workload in the chapter) is fixed, and the low-priority TPC-W workload (called the Federator workload in the chapter) is varied by the size of its database. This allows us to examine how the two workloads share the DBMS (with various admission control settings) as the workloads' bottleneck resource ranges from CPU to I/O.

Clearly, this type of evaluation study is most directly representative of real-world systems in which two different, but relatively similar Transactional Web applications share a DBMS. For instance, an online retailer (such as `Amazon.COM`) may provide two different store fronts, one is the high-priority workload (e.g. `Amazon.COM`'s book-selling store front) and the other is a lower-priority workload (e.g. `Amazon.COM`'s shoe-selling store front, `Endless.COM`). More generally, however, the insight gathered into how workloads mix as a function of their bottleneck resources proves to be applicable for a much wider range of scenarios.

Key Idea

The key idea employed in th performance evaluation in this chapter is that first (i) parameterizing workloads and DBMS and second (ii) comprehensively evaluating performance as the parameters change is essential

to fully understand DBMS performance. Many critical performance issues in DBMS, however, are only evident under limited conditions, and are only revealed through fairly exhaustive performance evaluations.

Summary of results

The main result from this portion of the chapter is the discovery of an unintuitive, and significant performance trend, called The Hump. When admission control lets in many low-priority queries, sometimes query response times (both high- and low-priority) remain low, yielding good high-priority performance isolation, while other times query response times grow large, yielding poor high-priority performance isolation. This trend of large growth in (high-priority) query response times define The Hump, and is due to changes in the DBMS efficiency and service rate.

In particular, when the high-priority queries are CPU-bound, then the DBMS efficiency (service rate) is high whenever the low-priority queries are either strongly CPU-bound or I/O-bound. When the low-priority queries transition between CPU-bound and I/O-bound, however, the efficiency (service rate) drops dramatically, causing both high- and low-priority response times to grow significantly. The result is that high-priority queries experience poor performance isolation when admission control is high.

This trend is observed with two TPC-W workloads running on the same IBM DB2 DBMS with different tuning parameters. Furthermore, the trend is also observed with a TPC-H workload (an industry standard data warehousing workload) and a TPC-W workload running on the same PostgreSQL DBMS (either TPC-H or TPC-W can be the high-priority workload). Thus the data indicates this is a general trend that occurs in many real systems.

4.1.2 Statistical Analysis

The second contribution of this chapter is a statistical analysis of the performance issues that cause The Hump.

Key Idea

The key idea employed in this portion of the chapter is essentially a bottleneck analysis and profiling of the DBMS. This approach is similar to the bottleneck analysis performed in Chapter 2, and focuses on the major physical and logical devices in the DBMS, such as CPU, I/O and Locks. The key distinction is that the analysis in this chapter takes the analysis *inside* these devices, and in particular, looks at where queries spend their time, and in particular *inside the CPU* device.

Summary of results

The primary result from the statistical analysis portion of this chapter the identification of the underlying performance issues that cause The Hump. In particular, the analysis reveals that there are two factors that contribute to the The Hump: At the peak of the hump, (i) queries require far more CPU service than they do otherwise, and (ii) the CPU service rate (the efficiency of the CPU) decreases significantly. These factors are caused by the details of how DBMS queries share the CPU resources.

4.1.3 IDD

The final contribution of this chapter is an analysis method called Isolated Demand Decomposition (IDD) that can be used to predict the performance of two workloads that share the same DBMS when one workload is limited via admission control.

Key Idea

The key idea is to start with a simple queueing theoretic performance model for DBMS, which is easily understood and relatively easy to analyze via simulation or numerical methods. Then, we augment the model to reflect how (i) queries' CPU demands change, and (ii) the CPU service rate changes when two different workloads are mixed on the same DBMS. Predicting how queries' CPU demands change and CPU service rate changes are slightly more self-contained problems, and thus slightly easier than fully integrating them into the queueing theoretic model.

Summary of results

We show that preexisting queueing models for DBMS performance completely miss the underlying performance issues at the CPU that cause the Hump. The result is that the performance predictions from preexisting performance models can be arbitrarily bad in many real-world DBMS. Furthermore, they do not even provide indication of what the basic performance trends.

Our novel IDD modeling approach is able to both (i) predict the performance trends seen in real-world systems (such as The Hump), ad (ii) predict the performance of both high- and low-priority query response times with an average relative error of only 11.7% in a real-world system.

4.2 Organization of this chapter

The remainder of this chapter proceeds as follows: Section 4.3 introduces and motivates the problem of using admission control to achieve prioritization for DBMS workloads. Section 4.4 discusses common applications in which this problem arises. Section 4.5 is a performance evaluation of TPC-W workloads sharing a DBMS, and documents the discovery of The Hump performance trend, and demonstrates that preexisting modeling techniques do not predict The Hump.

Section 4.6 consists of the statistical analysis to identify the underlying causes of The Hump, and the development of the Isolated Demand Decomposition (IDD) modeling approach which incorporates these causes into simple performance models. Section 4.6 also consists of an analysis of IDD to ensure that it accurately predicts performance. Section 4.7 discusses future work to improve IDD by improving the accuracy of predictions for CPU cache miss penalties. Section 4.8 discusses prior research pertinent to this chapter, and Section 4.9 summarizes the research. Section 4.10 describes the impact of the research found in this chapter, and Section 4.11 discusses future directions in which the research can be taken.

4.3 Introduction

One might expect that different people in a company would want to share their DataBase Management Systems (DBMS) with all other users in the company, since they all work to benefit the company. In reality, we find that DBMS users are greedy, and rarely want to share a DBMS with other users, even when both users are in the same corporate "family." The reason is simple: DBMS performance is fragile and hard to predict, and users are afraid that others will hurt their performance.

Often, only a small set of DBMS users are performance-sensitive, and the rest are happy with best-effort service. Throughout this chapter, we focus on a specific scenario described in Section 4.4, in which there are two classes of users: Performance-sensitive *Local* users and Best-effort *Federator* users.

Often, *admission control* is used to help provide better performance to Local users, by limiting the Federator MultiProgramming Level (MPL). The *Federator MPL* is the maximum number of concurrent Federator queries allowed in the DBMS. Excess Federator queries (beyond the Federator MPL) are queued outside the DBMS until other Federator queries depart the DBMS. Intuitively, admission control works since the fewer Federator queries that are in the DBMS, the fewer resources they consume, and the less likely they are to hurt Local response times. There are two problems, however: First, it is difficult to tune admission control and choose the optimal Federator MPL. Second, the intuition for admission control does not always hold. Sometimes allowing more Federators will not hurt (and may even help) Local performance.

If the Federator MPL is too low, DBMS devices may be under-utilized, and Federator users' performance may suffer unnecessarily. If Federator MPL is too high, Local response times may be hurt too much. Sadly, administrators have little to no guidance in choosing Federator MPL to achieve Local performance isolation. Tuning the Federator MPL is currently a time-consuming process of trial-and-error, until a suitable level is found. Further complicating the situation, the optimal Federator MPL changes whenever system hardware or software components change, or when the exogenous workload changes. Thus, the administrator must constantly monitor performance and perform costly retuning.

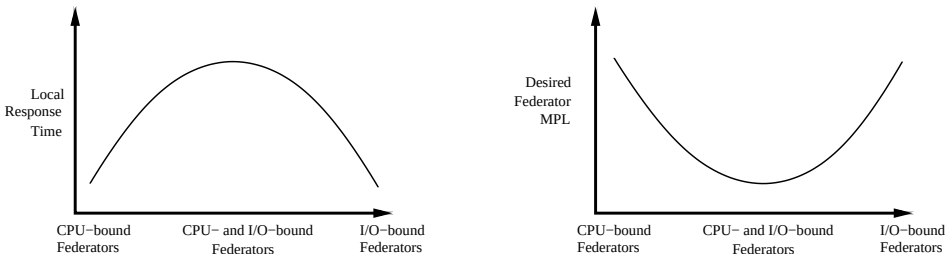
Our goal is to develop tools to reason about how Local and Federator users affect each other's performance (response times), as the Federator MPL is varied. Thus, administrators can answer "what-if" questions to help them configure Federator MPL to meet their performance requirements. As a basis for our analysis, we assume that we can make simple measurements of both the Local and Federator workloads running alone on the DBMS. Our approach uses simple, low-overhead measurements that do not require rewriting or replacing the DBMS, making our approach widely applicable. While we focus on admission control in this chapter, our work applies to any scenario with mixing DBMS workloads.

Throughout this chapter, we experiment with a commercial DBMS that serves two users' workloads, Local and Federator. Both the Locals and Federators TPC-W e-Commerce DBMS benchmark [21] workloads. We explore how response times for a CPU-bound Local workload change as the Federator workload varies from CPU-bound to I/O-bound, and as the Federator MPL varies from low to high.

We make a surprising discovery about the performance of mixed TPC-W workloads. Given that the Locals are CPU-bound, one might expect that the Locals are hurt the most by CPU-bound Federators (since they compete for the same resource) and are hurt the least by I/O-bound Federators (since they use different resources). Hence, one expects that Federator MPL must be kept low when Federators are CPU-bound, and can be higher when Federators are I/O-bound. We find this is not the case.

Instead, we find that CPU-bound Locals are not hurt much by either CPU-bound Federators (surprisingly) or I/O-bound Federators (expectedly). We also find (surprisingly) that CPU-bound Locals are, in fact,

significantly hurt when Federators have simultaneously high CPU- and I/O-demands. This trend is illustrated in Figure 4.2(a), whereby Local response times rise and then fall as the Federator workload shifts from CPU-bound to an I/O-bound workload (especially when Federator MPL is high). Throughout this chapter, we refer to this trend as "The Hump," (described in detail in Section 4.5), and we show it is not predicted by conventional queueing models.



(a) The Hump (b) Federator MPL Policy

Figure 4.2: Illustration of the observed trends. Counter to intuition, response times for CPU-bound Locals are good both when Federators are CPU-bound or Federators are I/O-bound, and in these cases, Federator MPL can be high. When Federators have simultaneously large CPU- and I/O-demands, Local response times are bad, and Federator MPL must be kept low.

The Hump leads us to conclude (non-intuitively) that Federator MPL can be kept high when Federators are CPU-bound or when Federators are I/O-bound. Likewise, Federator MPL must be kept low when Federators have both high CPU- and I/O-demands. Figure 4.2(b) illustrates the desired Federator MPL as a function of the Federator's bottleneck resource.

We provide an approach (described in Section 4.6), called Isolated Demand Decomposition (IDD), that predicts Local and Federator response times when run concurrently on the same DBMS based on how they run in isolation. This is important, as it is often undesirable to actually run the workloads together without knowing the performance ramifications. IDD's key idea is to measure each workload's device demands in isolation, and predict the demands in the mixed workload. IDD estimates Local and Federator response times using these demand estimates, with error between 11% and 17% in our experimental system. All measurements needed by IDD can be measured with minimal overhead without modifying the DBMS, which

makes IDD easy to deploy, even in legacy DBMS. We will also show that, unlike conventional queueing models, IDD accurately predicts The Hump.

4.4 Common Application

This work is motivated by problems witnessed in the real-world with Federated DBMS. Federated DBMS are used to unify several different physical DBMS, made by different vendors and holding different data, into a single logical DBMS. The Federated DBMS has a *Federator*, to which users can send queries that refer to data stored in any number of physical DBMS. The Federator will automatically (and efficiently) generate and send queries to all the physical DBMS as necessary and merge the results for the user.

Each physical DBMS in the Federation handles queries from both *Local Users*, and queries from *Federated Users*. Local users send queries directly to the physical DBMS, referring to data stored on only that physical DBMS. Federated users send queries to the *Federator*, which, in turn, sends queries to the physical DBMS.

Admission control is important in Federated DBMS since some Local users have sensitive performance needs, and have no interest in being part of the Federation of DBMS. Those Local users only need access to the data on their own Local DBMS, and are concerned that the extra work coming from other users' Federated queries will hurt their performance. Without understanding how the Federated workload will affect their performance, Locals often fight the deployment of the Federated DBMS, and succeed.

It is a simple process to install admission control to hold back Federator queries at a physical DBMS. Admission control can be installed externally to the DBMS, and does not require any modifications to the existing DBMS hardware or software, or to Local users' client software. All of these factors are critical in real-world systems, particularly those with legacy components or limited development and testing budgets.

4.5 The Hump

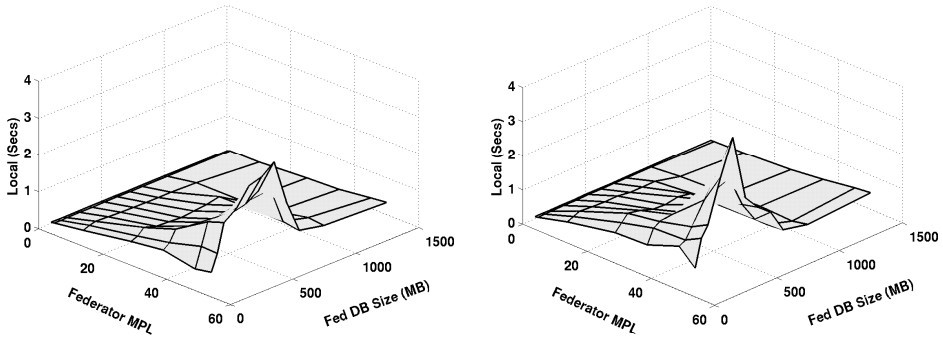
We are primarily interested in the effect Federators have on Local response times as Federator MPL is varied and as the Federator bottleneck resource varies from CPU-bound to I/O-bound. Throughout this chapter, we will vary the Federator DB size to vary the Federator bottleneck resource. When the Federator DB size is small (100MB), Federators are CPU-bound. As the Federator DB size increases, Federators consume increasingly more I/O and less CPU. When the Federator DB size is large (1500MB), Federators are I/O-bound. Thus, all experiments consider Local response times as a function of two axes: Federator MPL and Federator DB size.

Intuitively, one might expect that response times increase monotonically along each axis: (1) Increasing MPL slows Locals, due to the additional demand and contention for resources caused by the Locals. (2) Increasing Federator DB size slows Locals, since Federators create both more I/O requests (they have more buffer pool/cache misses) and slower I/O requests (their data is further apart on disk), which slows down Local I/Os. Both of these trends (1 and 2) are predicted by conventional queueing models of DBMS, such as the one presented in Section 4.5.3.

In practice, while we find that increasing Federator MPL does generally increase Local response times, increasing Federator DB size does *not* always increase Local response times. In fact, we find that as Fed-

erator DB size increases, Local response times rise, then fall, creating a "hump" trend, discussed in Section 4.5.2. The hump affects how admission control's Federator MPL should be configured. When the Federator DB size is within the hump region, increasing the Federator MPL hurts Local response times nearly an order of magnitude more than when the Federator DB size is outside the hump. Section 4.6 is dedicated to explaining the reasons for, and predicting, the hump.

4.5.1 Architecture and Experimental Setup



(a) Low Memory Config

(b) High Memory Config

Figure 4.3: The Hump: Local response times shown as a function of Federator MPL and Federator DB size. Local response times rise then fall as Federator DB size increases, as seen in many DBMS configurations.

Throughout this chapter, we consider a system with a single DBMS server, and two TPC-W workloads: the Locals, and the Federators. The DBMS holds two TPC-W databases, one for the Locals and one for the Federators. Each workload is generated in a "closed loop" whereby a fixed number of users repeatedly submit queries to the DBMS. We use 50 Local users and 50 Federator users.

Locals wait for an exponentially distributed "think time" between each DBMS query. We tune the Local think time so that the average number of Locals in the DBMS is one (when there are no Federators). When Federators are let into the DBMS, the number of Locals in the DBMS increases, since they slow down. This simulates a typical online arrival process, where there are usually few users, but there are sometimes many during periods of high-load. Federators have no think time, but the number of Federators in the DBMS is limited to the Federator MPL by admission control.

Throughout this chapter, we experimentally vary two axes: the *Federator MPL*, and the *Federator DB size*, and measure Local response time as these parameters change. We vary Federator MPL from 0 (no Federators) to 50 (all Federators) and Federator DB size from 100MB to 1500MB. This has the effect of shifting Federator queries from CPU-bound (100MB) to I/O-bound (1500MB). The Local DB size is always fixed at 100MB, making Locals CPU-bound. While both the Locals and Federators use different sets of data, both share the same disk.

Recall from Chapter 2, that TPC-W is almost always CPU-bound. To shift the workload from CPU-bound to I/O-bound, we increase the DB size outside of the parameters specified by TPC-W. It should be

observed that in all configurations, lock wait time is negligible.

Section 4.5.3 is exceptional since we vary Federator MPL and *Federator I/O rate* (rather than Federator DB size). This is because in Section 4.5.3, we formulate our problem as a queueing model, and changing the Federator I/O rate in the model most closely reflects changes to Federator DB size in the real system. As the Federator I/O rate decreases, the Federator I/O service times get longer, and individual Federator I/O requests slow down. Likewise, as Federator I/O rate increases, Federator I/O requests speed up. Note that Federator I/O rate and Federator DB size are inversely related. One might also consider changing Federator CPU rate to model changes to the Federator DB size, since the DBMS must process more data as the Federator DB size grows. We find, however, that Federator I/O rate is an order of magnitude more significant than Federator CPU rate, thus we concentrate on Federator I/O rate. We still, however, consider the effect of varying Federator CPU rate in Section 4.5.3 for completeness.

Our experimental setup uses two computers, one for the DBMS and one for the workload generators. The DBMS uses a dual-core 2.8-GHz Intel Xeon machine with 3GB of RAM and two SCSI disk drives, one for DBMS data, and one for DBMS logs. We use IBM DB2 as our DBMS. We spent a considerable amount of time tuning the DBMS to optimize its performance, with advice from IBM itself.

The workload-generation machine is a single-processor 2.4-GHz Intel Xeon machine, with 3GB of RAM and two IDE hard drives. The Local and Federator workloads are generated according to the industry-standard TPC-W e-Commerce benchmark [21]. Our implementation is in Java, and is based on the University of Wisconsin TPC-W implementation [55].

We use the TPC-W Browsing mix throughout this chapter. It is similar to the Shopping mix used in Chapter 2, but has a higher fraction of browse-related queries than buy-related queries. Both workload mixes are CPU-bound and are extremely similar.

For performance reasons, we make the following two primary changes to the Wisconsin TPC-W implementation:

First, we remove the need for a web server, so clients send DBMS requests directly to the DBMS. We do this since the web server should rarely be the bottleneck in online services, as web servers can typically be scaled and parallelized. Scaling DBMS performance is much more difficult.

Second, we modify the queries and architecture of the TPC-W workload to improve its efficiency. Several queries are changed to generate more efficient query plans and force the use of indexes when available. We make use of SQL `FOR UPDATE` clauses to improve the use of DBMS locks. Finally, unique identifiers and sequence numbers are generated more efficiently either at the application level or by DBMS sequence primitives.

4.5.2 Commercial DBMS in practice

We examine the mean Local response time as we vary the Federator MPL and the Federator DB size as described in Section 4.5.1. Since DBMS performance can depend on how the DBMS is configured, we consider a range of configurations. We find, however, that the trends in these configurations to be similar. We present two configurations here: a Low Memory Configuration (LMC) and a High Memory Configuration (HMC). The HMC differs from the LMC in two ways: (i) it uses more memory to store lock data structures and reduce spin locking, and (ii) it more aggressively cleans dirty data from the buffer pool. It should be noted that sometimes the LMC exhibits the best performance, whereas sometimes the HMC exhibits the best

performance.[1]

Figure 4.3 depicts the mean Local response time from these experiments. Figure 4.3(a), shows the LMC configuration and Figure 4.3(b) shows the HMC configuration. Both graphs show that Local response times exhibit a striking non-monotonic "hump" when the Federator DB size increases, corresponding to when the Federator workload shifts from CPU-bound to I/O-bound.

In general, Local response time increases monotonically with Federator MPL. The rate of increase, however, depends on whether the Federator DB size corresponds to a hump or non-hump region. Within the hump region, the penalty that increasing the Federator MPL has on Local response times is an order of magnitude higher than outside the hump region. In order to tune admission control, it is critical to understand how increases to Federator MPL change Local response times in a given system.

In particular, when the Federator DB size is small (100MB) or large (1500MB), increasing Federator MPL to 50 penalizes Local response times less than a factor of 10 (under 0.6 seconds per query). These cases correspond to when (i) Federator I/O requests are fast and Federators are CPU-bound (small Federator DB size) and (ii) Federator I/O requests are slow and Federators are I/O-bound (large Federator DB size). When the Federator DB size is "medium" (300MB to 800MB), and the Federators shift from CPU-bound to I/O-bound, increasing Federator MPL to 50 penalizes Local response times by a factor of 75 (3 to 4 seconds per query). We see this type of behavior under many DBMS configurations.

Our results indicate that common intuition about mixing workloads can be misguided. Intuition suggests that workloads that use different devices would mix well and have low response times, whereas workloads that use the same devices would compete and mix poorly. In our system, we find (as expected) that CPU-bound Locals and I/O-bound Federators (1500MB Federator DB size) do mix well. Surprisingly, we find that CPU-bound Locals and CPU-bound Federators (100MB Federator DB size) also mix well. Furthermore, it is hard to understand why we find that CPU-bound Locals mix poorly with Federators as Federators shift from CPU-bound to I/O-bound (300MB to 600MB Federator DB size).

All of Section 4.6 explains why the hump occurs, and describes how to use the IDD method to predict it. We will make the surprising discovery: the hump exists because Local queries' CPU demands increase when run together with Federators (relative to when run in isolation). In the next section (Section 4.5.3), we examine why conventional queueing models do not predict the hump.

4.5.3 Queueing Models are not enough

In an attempt to explain the hump seen in Section 4.5.2, we model the mixed Local and Federator DBMS with a queueing model. Our model is similar to those used by Thomasian and Ryu [71, 86], which have been used to successfully predict the effect MPL has on throughput (with a single class of users). We show that models of this type do not predict the hump seen in the real DBMS.

Our basic model is depicted in Figure 4.4, and consists of "device servers" for each important device in the DBMS: the CPU device, the primary I/O device, and a set of Think Time servers (which model the time Locals spend outside the DBMS). We consider many variations of this model, which consider different I/O scheduling policies, or incorporate the DBMS buffer pool or Log I/O device. We find that the Local response time trends are similar in each case.

We model Locals and Federators which run in a closed-loop, as in the experimental system. We model

[1]The HMC was reached after working with the DBMS vendor to improve performance at the peak of the hump (described below).

50 Local users and a number of Federator users equal to the Federator MPL (to model admission control). Queries move through the model as they would in a real DBMS: They start in the CPU and then either (i) complete or (ii) go to I/O. When Local queries complete, they depart the DBMS and wait in Think Time before returning to the DBMS. When Federator queries complete, they return immediately to CPU. All queries return to the CPU after serving at I/O.

Each server in the model (e.g. CPU, I/O, Think) processes queries at a fixed service rate, and service times are exponentially distributed. Service rates differ for Local and Federator queries, giving the Local service rate at server i: μ_i^{Loc} and the Federator service rate at server i: μ_i^{Fed}. We analyze the model using a Markov chain that tracks the number of Local queries in Think, CPU, and I/O (N_{Think}^{Loc}, N_{CPU}^{Loc}, and $N_{I/O}^{Loc}$), and the number of Federator queries in CPU and I/O (N_{CPU}^{Fed} and $N_{I/O}^{Fed}$). When we consider Random and FIFO I/O scheduling, we also track the number and order of queries in the I/O queue.

Transitions in the Markov chain are determined by the service rates and the scheduling policy at each server. The CPU uses processor-sharing (PS) scheduling. Given $N_{CPU} = N_{CPU}^{Loc} + N_{CPU}^{Fed}$ queries at the CPU, Locals leave the CPU with rate μ_{CPU}^{Loc}/N_{CPU} and Federators with rate μ_{CPU}^{Fed}/N_{CPU}. I/O uses either FIFO, Random, or PS scheduling. PS for I/O is handled similarly to the CPU case. For FIFO and Random, the first query in the I/O queue leaves at rate at rate μ_{IO}^{Loc} if it is Local or μ_{IO}^{Fed} if it is Federator.

One might think that this model is a closed classed Jackson queueing network, for which mean response time can be determined by closed-form formulas. This is not the case, however, since service rates depend on the class (Local or Federator) of the queries. As a result, we solve the Markov model using simulation.

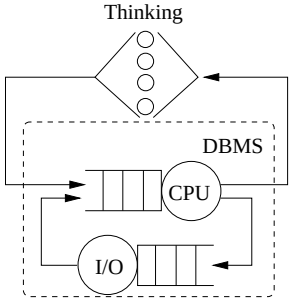


Figure 4.4: DBMS queueing model

As discussed in Section 4.5.1, we model the Federator DB size using the Federator I/O rate: $\mu_{I/O}^{Fed}$. The modeled Federator DB size is inversely proportional to $\mu_{I/O}^{Fed}$. For example, small modeled Federator DB size corresponds to a high Federator I/O rate.

We attempt to recreate the experiment from Figure 4.3(a) in Section 4.5.2 using the model, and measure Local response time as a function of Federator MPL and modeled Federator DB size. Our results are in Figure 4.5, and are representative of the trends produced by the model over a wide range of configurations. As Federator MPL increases, the model predicts monotonic increases to Local response time, as seen in the real DBMS. As Federator DB size increases, the real DBMS exhibits a hump, while the model exhibits a distinctive "dip." When Federator DB size is small, Local response times are constant. When Federator DB size grows, Local response times dip suddenly, then grow large monotonically. To understand the cause of the dip, we will further examine what causes the (i) constant and (ii) monotonic regions.

D_{CPU}^{Loc}	The per-job demand on the CPU by a Local job running in a Local-only workload.
D_{CPU}^{Fed}	The per-job demand on the CPU by a Federator job running in a Fed-only workload.
D_{Spin}^{Loc}	The portion of D_{CPU}^{Loc} spent spinning on spin locks in a Local-only workload.
D_{Spin}^{Fed}	The portion of D_{CPU}^{Fed} spent spinning on spin locks in a Federator-only workload.
D_{CPU}^{Mix}	The per-job demand on the CPU by all jobs running in the mixed Local and Federator workload.
$D_{CPU}^{Fed,Mix}$	The per-job demand on the CPU by a Federator job running in a Mixed workload.
$D_{CPU}^{Loc,Mix}$	The per-job demand on the CPU by a Local job running in a Mixed workload.
IPS^{Loc}	Number of instructions retired per second in the Local-only workload.
IPS^{Fed}	Number of instructions retired per second in the Federator-only workload.
IPS^{Mix}	Number of instructions retired per second in the Mixed workload.
$P\{Loc\}$	Fraction of Local jobs in the Mixed workload.
$P\{Fed\}$	Fraction of Local jobs in the Mixed workload.
T_{sys}^{Fed}	The measured system response time for a Federated job running in a Mixed workload.
T_{sys}^{Loc}	The measured system response time for a Local job running in a Mixed workload.
T_{sys}^{Mix}	The measured system response time for all jobs running in a Mixed workload.
T_{mod}^{Fed}	The response time for a Federated job running in a Mixed workload, as estimated by the model in Section 4.6.3.
T_{mod}^{Loc}	The response time for a Local job running in a Mixed workload, as estimated by the model in Section 4.6.3.
U_{CPU}^{Loc}	CPU utilization in a Local-only workload.
U_{CPU}^{Fed}	CPU utilization in a Federator-only workload.
X^{Loc}	Throughput (queries per second) in a Local-only workload.
X^{Fed}	Throughput (queries per second) in a Federator-only workload.

Table 4.1: Primary notation used for IDD parameters (Section 4.6). CPU can be replaced with I/O throughout the above.

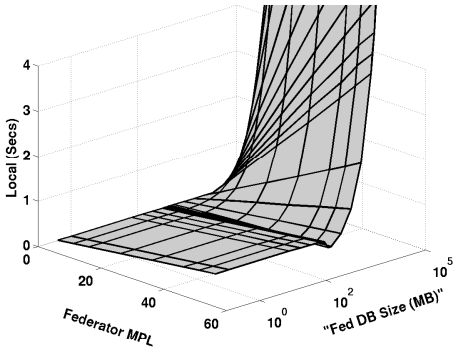


Figure 4.5: Local response times as a function of Federator MPL and modeled Federator DB size (Federator I/O rate). Conventional queueing models predict a Local response time dip, not the hump seen in real-world DBMS. Compare to Figure 4.3(a).

The constant region is caused because Federators are CPU-bound in the region. Federator I/O is faster than Federator CPU, and all Federators pile up in the CPU bottleneck. Locals are slowed down by a constant factor since the CPU uses PS scheduling and there are a fixed number of Federators in the DBMS. The monotonic region is caused because Federators are I/O-bound in the region and all Federators pile up in the I/O bottleneck. Local queries can often complete without needing I/O, but periodically do need I/O, and must wait behind many slow Federators to get it. As modeled Federator DB size grows, Federator I/O slows down, which increasingly hurts Local response times.

The dip in Figure 4.5 as Federator DB size increases is a discontinuity produced as the Federators shift from CPU-bound to I/O-bound. The dip corresponds to the case where all the Federators start to move from CPU to I/O so that both (i) Locals get more of the CPU which makes Locals faster, and (ii) Federator I/O is fast enough that it does not hurt Locals too much. The magnitude of the dip grows as the Federator CPU rate increases. We find that the model does not predict a hump over any combination of Federator I/O rates and Federator CPU rates.

We have considered modeling Federator DB size using other methods. One method models Federator DB size using Federator I/O rate. Another method sets Local CPU and I/O service rates as measured from the real Local-only system, and then sets Federator CPU and I/O service rates as measured from the real Federator-only system, for a given Federator DB size (this approach is similar to applying IDD without estimating mixed device demands). In all cases, the model is incapable of producing the response time hump.

The primary difficulty faced when modeling DBMS performance is that the DBMS processes queries differently when they are mixed with others. When by themselves, queries have one set of CPU and I/O service rates, and when run with others, the service rates are often very different. Models must be able to predict how service rates must change when workloads are mixed. The next section (Section 4.6) (i) explains why the service rates change when workloads are mixed, and (ii) outlines IDD, which predicts how service rates change in order to determine Local (and Federator) response times in the mixed workload.

4.6 Our Approach: IDD

We now introduce the main Isolated Demand Decomposition (IDD) procedure. Given measurements from Local and Federator workloads running in isolation, IDD models DBMS performance when the workloads run concurrently (mixed workload), and accurately estimates the Local response time and the Federator response time in the mixed workload.

IDD consists of four steps: (1) Run the Local and Federator workloads in isolation to measure the *isolated device demands* (Section 4.6.1); (2) Estimate the *mixed device demands* by modeling how isolated device demands change when the workloads are run together (Section 4.6.2); (3) Employ a queueing model which uses the estimated mixed device demands to estimate performance metrics for Locals and Federators (such as throughput, response time, etc.), when run together; (4) Use operational laws to improve upon the estimated performance metrics above, yielding a final estimate for Local (and Federator) response times (Section 4.6.4) for the mixed workload.

IDD relies intrinsically on *device demands*. The device can be either CPU or I/O. The CPU device demand represents the average amount of work needed to be done per query (or "job") on the CPU, i.e., the "CPU time" required per query in the system. Observe that the CPU demand excludes the time the query spends waiting or queueing for the CPU or time spent at other devices.

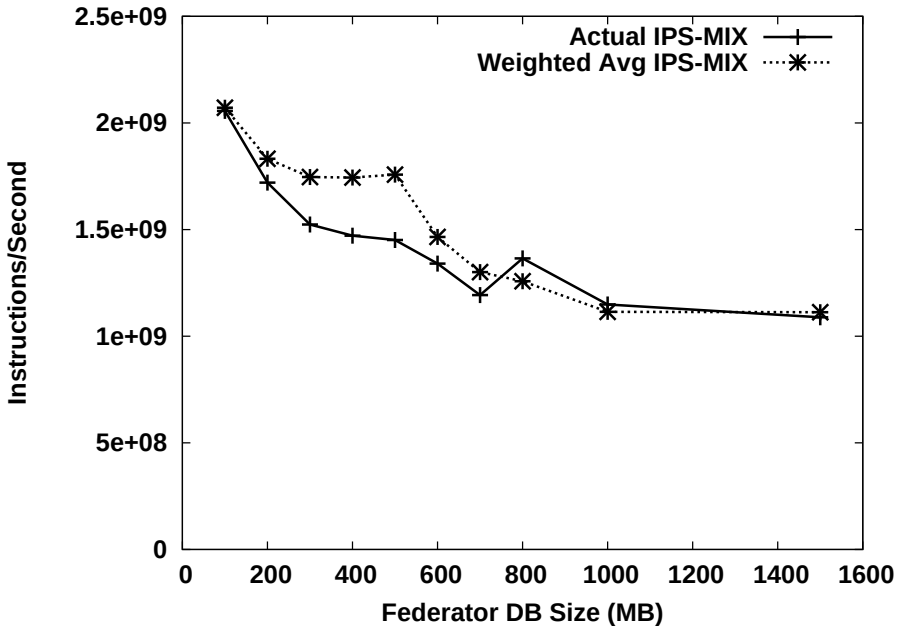
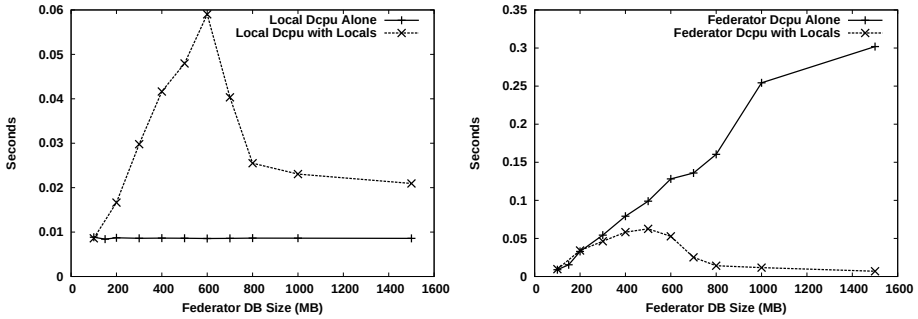


Figure 4.6: Actual and estimated IPS^{Mix} as a function of Federator DB size with Federator MPL set to 50. CPU stalls reduce CPU strength by a factor of 2, which is accurately estimated by IDD.

D_{CPU}^{Fed} denotes the CPU (per-job) demand for Federator jobs running in isolation on the DBMS. Likewise D_{CPU}^{Loc} denotes the CPU (per-job) demand for Local jobs running in isolation on the DBMS. An important



(a) Local Demands (b) Federator Demands

Figure 4.7: Local and Federator CPU demands as a function of Federator DB size, with Federator MPL set to 50. (a) $D_{CPU}^{Loc,Mix}$ differs from D_{CPU}^{Loc} and (b) $D_{CPU}^{Fed,Mix}$ differs from D_{CPU}^{Loc}, proving that demands change when workloads mix.

contribution of this chapter is that the device demands *change* when Locals and Federators are run concurrently. We call this a *mixed workload*. This fact necessitates defining additional terminology: $D_{CPU}^{Fed,Mix}$ will denote the CPU demand for Federator jobs running in a mixed workload on the DBMS (mix of Locals and Federators). The remaining similar notation is defined in Table 4.1. Note that response times are always defined for the case of mixed workload (Locals and Federators running concurrently).

IDD *measures* the *isolated device demands*, e.g., D_{CPU}^{Fed}, $D_{I/O}^{Fed}$, etc. from the real DBMS when running Locals and Federators in isolation. In contrast, IDD *estimates* the *mixed device demands*, e.g., $D_{CPU}^{Loc,Mix}$, $D_{CPU}^{Fed,Mix}$, etc.

When describing IDD, we focus on CPU demands, in particular when estimating mixed device demands. This is because (in our system) the CPU is the device with maximum demand, making it the bottleneck device, and hence the dominant component in response time. Figure 4.8 shows the CPU and I/O demands in our mixed Local and Federator workload, showing that CPU is almost always the bottleneck. IDD should also apply when I/O is the bottleneck resource. Here again the isolated I/O demands will differ from the mixed I/O demands.

4.6.1 Measure Isolated Device Demands

The first step of IDD is to measure the isolated device demands for each device i: D_i^{Loc} and D_i^{Fed}. We only consider CPU and I/O devices, since other device demands (e.g. locks) are negligible in our system (as seen in Chapter 2, TPC-W workloads have almost no lock waiting).

We rely on the fact that device demands are related to throughput (queries per second) and device utilization (the time-average fraction of time the device is busy working on any query) via the following operational laws:

$$\begin{aligned} D_i^{Fed} &= U_i^{Fed}/X^{Fed} \\ D_i^{Loc} &= U_i^{Loc}/X^{Loc} \end{aligned}$$

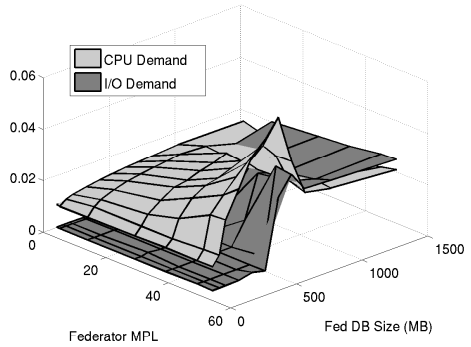


Figure 4.8: D_{CPU}^{Mix} and $D_{I/O}^{Mix}$ as a function of Federator MPL and Federator DB size. CPU is almost always the bottleneck, especially in the hump region.

where U_i^{Fed} is the device utilization for device i and X^{Fed} is the throughput, when Federators are run in isolation (and similarly for Locals).

It is easy to measure throughput and device utilizations on most DBMS and operating systems, with low overhead and without modifying the DBMS. For example, on Linux and UNIX, device utilizations can be measured using `iostat` and `vmstat`. Throughput can be measured by the application, the DBMS, or anywhere in the communication stream between them.

Device demands need to be measured for each device in the DBMS. We find this includes all CPUs and I/O devices, except for the DBMS log I/O device (which has negligible demands). We find that DBMS lock demands are negligible in our system, and thus, applying IDD to lock demands is beyond the scope of this chapter.

DBMS locks will not need to be considered in many large-scale commercial DBMS applications, due to the fact that many, such as Amazon [26], are designed to eliminate locking (and lock demands) as much as possible. In the event that lock demands are significant, and are responsible for significant fractions of query response times (as seen in TPC-C workloads in Chapter 2), IDD is still applicable as long as each workload accesses independent data, since lock demands are not expected to change. When workloads access the same data, however, demands will have to be adjusted according to the amount of incompatible data sharing. Approaches that model the amount of lock contention, such as those by Thomasian et al. [83] will need to be used.

4.6.2 Estimate Mixed Device Demands

When two workloads are mixed, we might expect that the device demands in the mixed workload are simply the weighted average of the device demands for the workloads running in isolation. In other words, for device i, we expect:

$$D_i^{Mix} = P\{Loc\}D_i^{Loc} + P\{Fed\}D_i^{Fed} \tag{4.1}$$

where $P\{Loc\}$ and $P\{Fed\}$ are the fraction of Local and Federator queries that complete in the mixed workload.

In measured DBMS performance, however, equation (4.1) *does not hold*. This is because the device demands for a query *change* when it is mixed with other queries. By definition, the following is true:

$$D_i^{Mix} = P\{Loc\}D_i^{Loc,Mix} + P\{Fed\}D_i^{Fed,Mix} \tag{4.2}$$

However, $D_i^{Loc,Mix} \neq D_i^{Loc}$ and $D_i^{Fed,Mix} \neq D_i^{Fed}$, hence (4.1) does not follow. To see this, we measure the Local and Federator CPU demands in the mixed workload, $D_{CPU}^{Loc,Mix}$ and $D_{CPU}^{Fed,Mix}$, and compare these with the measured isolated CPU demands D_{CPU}^{Loc} and D_{CPU}^{Fed}, as shown in Figure 4.7. Demands are shown as a function of Federator DB size with a fixed Federator MPL of 50.

In general, Local CPU demands are much *larger* in the mixed workload than in isolation, while the Federator CPU demands are much *lower* in the mixed workload than in isolation (Figure 4.7). Thus, Local and Federator CPU demands in the mixed workload are comparable. In addition, $P\{Loc\}$ dominates $P\{Fed\}$ in the mixed workload (seen later in Figure 4.10), especially as Federator DB size increases. Combining these facts means that $P\{Loc\}D_{CPU}^{Loc,Mix}$ dominates D_{CPU}^{Mix} in equation (4.2).

Given that CPU is almost always our bottleneck resource, and that $P\{Loc\}D_{CPU}^{Loc,Mix}$ dominates D_{CPU}^{Mix}, we focus on how to estimate $D_{CPU}^{Loc,Mix}$ in the remainder of this section.

The fact that $D_{CPU}^{Loc,Mix}$ is larger than D_{CPU}^{Loc} can be attributed to either: (1) CPU devices become slower or less efficient in the mix, or (2) queries perform more CPU work in the mix. We find that *both* of these reasons are responsible, via (1) CPU stalling (Section 4.6.2) and (2) spin locking (Section 4.6.2). It will turn out that these cause the experimental hump.

Our goal in Section 4.6.2 and Section 4.6.2 is to determine how to estimate $D_{CPU}^{Loc,Mix}$ and $D_{CPU}^{Fed,Mix}$ based on how CPU stalls and spin locks affect D_{CPU}^{Loc} and D_{CPU}^{Fed}.

The final estimates are as follows:

$$D_{CPU}^{Loc,Mix} \approx D_{CPU}^{Loc}\frac{IPS^{Loc}}{IPS^{Mix}} + D_{Spin}^{Fed}\frac{IPS^{Fed}}{IPS^{Mix}} \tag{4.3}$$

$$D_{CPU}^{Fed,Mix} \approx D_{CPU}^{Fed}\frac{IPS^{Fed}}{IPS^{Mix}} + D_{Spin}^{Loc}\frac{IPS^{Loc}}{IPS^{Mix}} \tag{4.4}$$

where IPS^{Loc} is the CPU instructions per second (IPS) measured in the Loc-only workload, IPS^{Fed} is the CPU IPS measured in the Fed-only workload, IPS^{Mix} is the CPU IPS estimated for the mixed workload, D_{Spin}^{Fed} is the spin lock demand for the Loc-only workload, and D_{Spin}^{Fed} is the spin lock demand for the Fed-only workload.

We will discuss how to estimate $P\{Loc\}$ and $P\{Fed\}$ in Section 4.6.3. Together these will give us all the inputs we need for equation (4.2), enabling us to get D_i^{Mix}, which will then be used in Section 4.6.4 to get the mixed workload response times.

CPU Stalling

It is well-known that DBMS performance suffers due to CPU stalling, particularly with modern CPUs [8]. CPU stalling decreases the number of instructions retired per second (IPS) that the CPU is able to execute. Since queries must execute a fixed number of instructions with or without stalling, stalling increases the CPU demand of the queries.

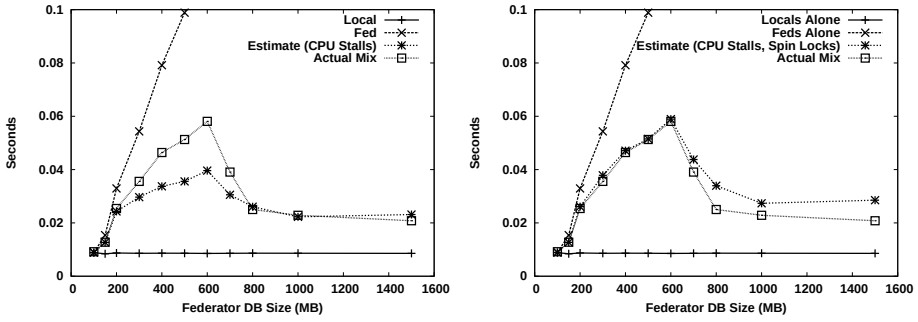
We find that TPC-W stalls are predominantly due to memory stalls. Several techniques, such as those from Zhang et al. [90] and Jacob et al. [46]), can be used to model memory stall penalties. These techniques use parameters describing the workload locality (e.g. Zhang et al.'s α and β), and costs associated with cache misses at each level of the memory hierarchy, to determine the amount of CPU stalling. Unfortunately, these techniques concentrate on modeling single workloads, and cannot be directly applied to predict miss rates when mixing multiple workloads. The primary difficulty is to determine the locality parameters for the mixed workload, based on the locality parameters of the workloads running in isolation. Given the difficulty required in determining these parameters, we choose not to take this approach, and largely leave it to future work.

Instead, we present a simpler method to estimate CPU stalling. We measure the number of instructions retired per second (IPS) for Locals alone and Federators alone, and average these together to predict the IPS in the mixed workload. This method ignores the fact that the locality of the mixed workload is worse than that of the workloads in isolation. Thus, in general, we predict lower miss rates and stall penalties than actually may occur in practice. Section 4.7 discusses how to address this problem.

Specifically, IDD models the effect that CPU stalling has on D_{CPU}^{Loc} and D_{CPU}^{Fed} using the following procedure: (1) Measure IPS^{Loc} and IPS^{Fed} from the isolated Local and Federator workloads; (2) Estimate IPS^{Mix} using the weighted average of IPS^{Loc} and IPS^{Fed}; (3) Scale D_{CPU}^{Loc} by the ratio: IPS^{Loc}/IPS^{Mix} and D_{CPU}^{Fed} by the ratio: IPS^{Fed}/IPS^{Mix}.

The above steps are described below, but first, we attempt to quantify the effect of CPU stalling in our mixed workload system. Figure 4.6 shows the Actual IPS^{Mix} as a function of Federator DB size when Federator MPL is 50. Over the experimental range, CPU IPS drops by approximately 48%, effectively cutting CPU strength in half and doubling CPU demands. If CPU is the bottleneck, doubling the CPU demands can approximately double response times.

Step (1). We measure IPS^{Loc} and IPS^{Fed} from the isolated Local and Federator workload systems. We rely on the Linux OProfile tool [54] to do this for us. OProfile uses the x86 hardware performance counters to measure the number of instructions executed (retired) in each workload, and does not require modifying the DBMS in any way. Similar tools are available on a wide range of CPU architectures and operating systems, and generally have negligible overhead.



(a) Stalling (b) Stalling + SpinLock

Figure 4.9: Estimates for D_{CPU}^{Mix} as a function of Federator DB size with Federator MPL set to 50. Estimates use measured D_{CPU}^{Loc} and D_{CPU}^{Fed}, and account for (a) CPU stalls alone, and (b) CPU stalls and spin locks.

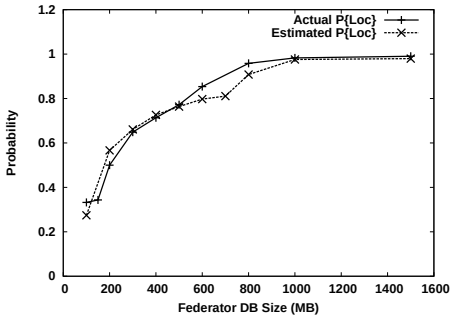


Figure 4.10: Actual and estimated $P\{Loc\}$ as a function of Federator DB size with Federator MPL set to 50. IDD's queueing model, correctly estimates $P\{Loc\}$ and $P\{Fed\}$.

Step (2). We estimate IPS^{Mix} using a simple weighted average of IPS^{Loc} and IPS^{Fed}:

$$IPS^{Mix} = \frac{N^{Loc} \cdot IPS^{Loc} + N^{Fed} \cdot IPS^{Fed}}{N^{Loc} + N^{Fed}}$$

where N^{Loc} and N^{Fed} are the number of Locals and Federators in the mixed workload. This estimate is quite crude, although it produces accurate results, especially when N^{Loc} and N^{Fed} are large. Figure 4.6 compares the actual IPS^{Mix} to the weighted average estimate, with 50 Locals and 50 Federators, which has an average relative error of only 9% over the experimental range. Section Section 4.7 discusses how to improve this estimate, particularly for other systems.

Step (3). To estimate $D_{CPU}^{Loc,Mix}$, we scale D_{CPU}^{Loc} by a factor $k^{Loc} = \frac{IPS^{Loc}}{IPS^{Mix}}$. k^{Loc} is actually a product of two terms:

$$k^{Loc} = \frac{IPS^{Loc}}{IPS^{Peak}} \cdot \frac{IPS^{Peak}}{IPS^{Mix}}$$

where IPS^{Peak} represents the peak number of instructions the CPU(s) can retire per second.

The first term in k^{Loc} is a scaling factor which erases the stalling found when running Locals in isolation, and the second term is a scaling factor to account for the stalling expected in the mixed workload. Similarly, $D_{CPU}^{Fed,Mix}$ is scaled by a factor k^{Fed}, where $k^{Fed} = \frac{IPS^{Fed}}{IPS^{Mix}}$.

Figure 4.9(a) compares the actual D_{CPU}^{Mix} with the estimated D_{CPU}^{Mix} using D_{CPU}^{Loc} and D_{CPU}^{Fed} adjusted for CPU stalls, as described above. The estimate's average relative error is 14% over the experimental range. We show next that this error is primarily due to spin locking.

Spin Locks

Spin locks are used in DBMS to protect internal DBMS data structures for short periods of time. To wait for a spin lock, a query "spins" by repeatedly trying to acquire the lock until the lock is released (by another query). As queries spin, they execute more CPU instructions, increasing CPU demand. IDD models the effect spin locking has on D_{CPU}^{Loc} and D_{CPU}^{Fed} using the following procedure:

Step (1). We measure D_{Spin}^{Loc} and D_{Spin}^{Fed}, the portions of D_{CPU}^{Loc} and D_{CPU}^{Fed} caused by spinning, by measuring (i) the mean number of instructions executed per query for Locals (I^{Loc}) and Federators (I^{Fed}), and (ii) the mean number of spin-lock instructions executed per query for Locals (I_{Spin}^{Loc}) and Federators (I_{Spin}^{Fed}). D_{Spin}^{Loc} and D_{Spin}^{Fed} are then:

$$D_{Spin}^{Loc} = \frac{I_{Spin}^{Loc}}{I^{Loc}}$$
$$D_{Spin}^{Fed} = \frac{I_{Spin}^{Fed}}{I^{Fed}}$$

I^{Loc}, I_{Spin}^{Loc}, I^{Fed}, and I_{Spin}^{Fed} are measured by OProfile, which (statistically) counts the number of instructions executed by each function in the DBMS. Thus, we must determine which DBMS functions correspond to spin locking, which is usually an easy task (otherwise, the DBMS vendor can help).

Step (2). We model the effect spin locking has on CPU demands by increasing D_{CPU}^{Loc} by D_{Spin}^{Fed} and D_{CPU}^{Fed} by D_{Spin}^{Loc}.

The intuition for this is as follows: Consider the system with Locals running in isolation. An average Local query spins for D_{Spin}^{Loc} seconds to acquire its spin locks, so those locks are held on average for D_{Spin}^{Loc} seconds. We assume that Federators need the same locks as the Locals. Thus, when Federators are mixed in, Federators spin for approximately the time Federators spin in isolation (included in D_{CPU}^{Fed}) plus the time Locals spin in isolation.

The final estimates for $D_{CPU}^{Loc,Mix}$ and $D_{CPU}^{Fed,Mix}$ are computed by first: increasing D_{CPU}^{Loc} and D_{CPU}^{Fed} to model spin locks as above, and then scaling by k^{Loc} and k^{Fed} to model CPU stalls (as described in Section 4.6.2). These estimates are summarized in Equation 4.4.

Figure 4.9(b) compares the actual D_{CPU}^{Mix} with the estimated D_{CPU}^{Mix} based on the final estimates of D_{CPU}^{Loc} and D_{CPU}^{Fed}, accounting for both CPU stalling and spin locking. The estimate's average relative error is only 10% over the experimental range. The error is an overestimate, which leads IDD to conservatively estimate response time. We find that, in our system, spin locking is only an issue within the hump region, and is caused by the Federators. Note when there is no spin locking, the hump will still exist, and IDD is even more accurate.

4.6.3 Solve a New Queueing Model

In this section, IDD builds a simple queueing model using the estimated mixed device demands estimated from the prior section, and then solves that model to estimate performance metrics for the mixed workload system. In particular, we estimate (i) $P\{Loc\}$ and $P\{Fed\}$, the fraction of Local and Federator queries that complete in a time period, and (ii) T_{mod}^{Loc} and T_{mod}^{Fed}, the Local and Federator response times.

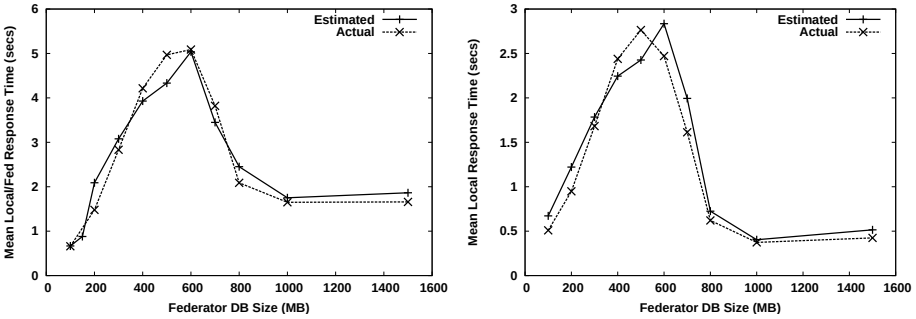
IDD uses a simple variant of the queueing models presented in Section 4.5.3. Our model has three servers: a processor-sharing CPU server, processor-sharing I/O server, and Local think time. Locals and Federators are routed probabilistically to I/O from the CPU device. The I/O-probability is set so that the mean number of I/O requests for Local and Federator queries is equal to the number measured when running each workload in isolation.

Since we model I/O using processor-sharing, states in the Markov chain are simple 5-tuples: N_{Fed}^{CPU}, $N_{Fed}^{I/O}$, N_{Loc}^{Think}, N_{Loc}^{CPU}, and $N_{Fed}^{I/O}$. The transitions are similar to those described in Section 4.5.3.

Each server has a separate Local and a Federator service rate. IDD sets the service rates in so that the device demands in the model are equal to IDD's estimated mixed device demands. For example, Local CPU service rate, μ_{CPU}^{Loc} is determined as follows: First, calculate the expected number of visits to the CPU device by Local queries (V_{CPU}^{Loc}), using the Local I/O routing probability. Then, $\mu_{CPU}^{Loc} = V_{CPU}^{Loc}/D_{CPU}^{Loc,Mix}$. Similar computation is used to determine all the other service rates.

We find that the model predicts $P\{Loc\}$ and $P\{Fed\}$ accurately. Figure 4.10 compares $P\{Loc\}$ estimated by the model with $P\{Loc\}$ actually measured in the mixed workload, as a function of Federator DB size with Federator MPL fixed at 50. The average relative error over the range of the estimate is 5.6%, and the maximum relative error is 16%. Given its simplicity, the Markov chain can be solved quickly using simulation. Results are within 0.06% error after simulating 10,000 Local queries, which takes less than a second.

Unfortunately, the model poorly predicts mean Local (and Federator) response times: T_{mod}^{Loc} and $T_{mod}^{Fed,mix}$, giving an average error of over 26%. In the next section, we will use operational laws to improve these response times, completing the IDD procedure.



(a) T^{Mix} Estimate (b) T^{Loc} Estimate

Figure 4.11: IDD's final estimates for (a) T_{sys}^{Mix} and (b) T_{sys}^{Loc} (and T_{sys}^{Fed}) as a function of Federator DB size with Federator MPL set to 50. IDD's estimates are accurate and correctly predict the hump.

It should be observed that estimating mixed device demands as described in Section 4.6.2 is essential. When unadjusted isolated device demands are used to configure the queueing model, (1) the relative error for Local response time is up to 93%, and (ii) the the ratio of T_{mod}^{Loc} to T_{mod}^{Fed} relied on in Section 4.6.4 is off by 60%. Due to these factors, without accurately estimating mixed device demands, we cannot predict the hump.

4.6.4 Improve Response Time Estimate

The final step of IDD produces an accurate estimates for T_{sys}^{Mix}, the mean Local response time in the mixed workload, T_{sys}^{Loc}, the mean Local response time in the mixed workload, and T_{sys}^{Fed}, the mean Federator re-

sponse time in the mixed workload. The estimates are made using statistics estimated earlier in the IDD procedure, based on measurements from the Local and Federator workloads running in isolation. In particular, we use (i) the estimated mixed device demands from Section 4.6.2 and (ii) the predictions from the queueing model in Section 4.6.3 for $P\{Loc\}$, $P\{Fed\}$, T_{mod}^{Loc} and T_{mod}^{Fed}.

First, we estimate T_{sys}^{Mix}. We rely on the following operational law (based on Little's law):

$$T \geq max\left(\sum D_i, N \cdot D_{max} - E[Z]\right) \tag{4.5}$$

where T is the mean response time, N is the number of users in the system, D_{max} is the demand of the device with highest demand, and $E[Z]$ is the mean think time. All variables on the right side of the bound are either known from the system (N and $E[Z]$) or have been estimated earlier in IDD (D_{max} from the estimated mixed device demands). When N is high, and the system is under high load, the $N \cdot D_{max} - E[Z]$ term dominates, and the bound is tight. We are most interested in performance under high load, so we use the bound itself as an estimate for the response time.

We use Equation (4.5) to estimate T_{sys}^{Mix}. We estimate D_{max} in the mixed workload using (i) the estimated mixed device demands from Section 4.6.2 and (ii) the estimated $P\{Loc\}$ and $P\{Fed\}$ from Section 4.6.3. Figure 4.11(a) compares the estimated T_{sys}^{Mix} to the actual T_{sys}^{Mix} as a function of Federator DB size with Federator MPL set to 50. We find that the estimate has average relative error of only 11.7% over the experimental range.

Next, we use T_{sys}^{Mix}, T_{mod}^{Loc} and T_{mod}^{Fed} to estimate T_{sys}^{Loc} and T_{sys}^{Fed}. In Section 4.6.3, we found that the estimates T_{mod}^{Loc} and T_{mod}^{Fed} were inaccurate. Given that the model estimates $P\{Loc\}$ and $P\{Fed\}$ correctly, we make the assumption that the ratio $T_{mod}^{Loc}/T_{mod}^{Fed}$ is an accurate estimate for $T_{sys}^{Loc}/T_{sys}^{Fed}$. We define the ratio $c_{mod} = T_{mod}^{Loc}/T_{mod}^{Fed}$. By definition, $T_{sys}^{Mix} = P\{Loc\} \cdot T_{sys}^{Loc} + P\{Fed\} \cdot T_{sys}^{Fed}$. Solving this for T_{sys}^{Loc} and substituting $T_{sys}^{Fed} = T_{sys}^{Loc}/c_{mod}$ gives T_{sys}^{Loc}:

$$T_{sys}^{Loc} = \frac{T_{sys}^{Mix} \cdot c_{mod}}{P\{Loc\} \cdot c_{mod} + P\{Fed\}}$$

Figure 4.11(b) depicts T_{sys}^{Loc} estimated as described above, as a function of Federator DB size with Federator MPL set to 50. We find that the estimate accurately predicts the response time hump, and has an average error of only 17% over the experimental range.

4.6.5 IDD Summary

- Run each workload in isolation and measure the isolated device demands (D_i^{Loc} and D_i^{Fed}) and OProfile measurements.
- Estimate mixed device demands ($D_i^{Loc,Mix}$ and $D_i^{Fed,Mix}$) from isolated device demands (D_i^{Loc} and D_i^{Fed}) so that Equation (4.1) holds.
- Build a simple queueing model using the estimated mixed device demands. Solve the model to predict performance statistics for the mixed workload system: $P\{Loc\}$, $P\{Fed\}$, T_{mod}^{Loc} and T_{mod}^{Fed}.
- Use operational laws (Equation (4.5)) to improve response time estimates for T_{sys}^{Mix}, T_{sys}^{Loc}, and T_{sys}^{Fed} from the queueing model.

4.7 Improving Cache Miss Penalty Prediction

Section 4.6.2 presents an estimate for the CPU stall penalty (due to cache misses) in a mixed workload. This estimate is a simple linear combination of the CPU stall penalty of each of the workloads in the mixed workload. Unfortunately, this estimate does not work in all systems. For instance, changing the think time for one of the workloads' arrival processes significantly can reduce the accuracy of this type of estimate by a factor of 30% to 50%. An error of this magnitude would greatly reduce the accuracy IDD's performance predictions, reducing its overall effectiveness.

It is a difficult problem to accurately estimate the CPU stall penalty seen by two workloads running together based on the CPU stall penalties each workload sees running independently of one another. Most existing research attempts to model CPU cache miss rates and stall penalties, and determine the magnitude of these penalties as a function of the cache size or cache design. This research all focuses on a single workload, and does not consider the effects of mixing two workloads on the same CPU, as we do in this chapter. Most such research does not constrain itself only to DBMS workloads, and focuses on the more general problem of running general programs.

4.7.1 Stack Depth Distributions

The primary approach to modeling cache performance is to statistically characterize the workload's pattern of memory references, estimate the probability of each memory access to miss in the cache (assuming independence), and then model system performance based on this probability. The predominant model for memory references is the stack-depth model [90, 78].

The stack-depth model conceptually keeps a stack of every memory access made by a program during its execution. The model assumes that every memory access is made by probabilistically choosing a *depth*, and accessing the datum located at that depth on the stack. Accessing the datum removes the datum from the conceptual stack, and pushes it onto the front of the stack.

The central idea is that more-recently accessed data is very likely to be accessed again, and less-recently accessed data is much less likely to be accessed again. In the stack depth model, a workload is characterized by the probability distribution of stack depths. The probability density function, $p(x)$ governs the probability of accessing a datum at depth x on the stack, and $P(x)$ is the cumulative probability distribution function. It should be observed that the relevance of $P(x)$ is that $1-P(x)$ gives the cache miss rate for a fully-associative LRU cache of size x.

In the stack depth model, $p(x)$ characterize the locality of the program, and can take various forms. We rely on the form used by Zhang et al. [90], which parameterizes $p(x)$ with two locality parameters $\alpha > 1$ and $\beta > 1$:

$$\begin{aligned} P(x) &= 1 - \frac{1}{(x/\beta + 1)^{\alpha-1}} \\ p(x) &= \frac{\beta^{\alpha-1}(\alpha - 1)}{(x+\beta)^{\alpha}} \end{aligned}$$

The form of the distribution function arises due to the "30% Rule" [78], which says that doubling the

cache size should reduce cache misses by 30%. Solving the recurrence relation $0.7f(x) = f(2x)$ yields a polynomial of the form $f(x) = \beta x^{\alpha}$. The form simply ensures that the function is a relatively simple probability distribution. Research has demonstrated that many real-world CPU workloads result in memory reference patterns that are compatible with the above stack distance distributions [46, 77, 78].

Mixed Stack Depth Distributions

When two programs that reference memory according to a stack depth model are executed together on a CPU (and interleaved using processor sharing or time slices), the resulting mixed workload (in general) does not follow a stack depth model. Thus, it can be difficult to predict and quantify the cache miss rate of the mixed workload, which makes it difficult to then predict the CPU stall penalty.

The cache miss rate of the mixed workload (of two other workloads) depends on (i) the stack-depth distribution of the first workload, (ii) the stack-depth distribution of the second workload, and (iii) the scheduling quantum: how long a query from one workload executes on the CPU before another query from another workload executes (because the first query completes, is preempted by the CPU scheduler, or is forced to wait for I/O or locks).

To understand how the above three factors affect the cache miss rate of the mixed workload, we implement a cache simulator, that simulates two programs and a unified fully-associative LRU CPU cache. The simulator alternates between executing the first workload and second workload, and a configurable *quantum size* determines how many memory accesses of each are made before switching between workloads. Each program has its own stack-depth distribution and accesses its own data (just as the workloads throughout this chapter access their own data). Throughout, we vary the stack depth distributions of each workload, as well as the quantum size. We simulate 5 million memory accesses for each experiment.

The first question we address is how α and β for each workload's stack depth interact to determine the cache miss rate for the mixed workload.

We start by exploring all possible combinations of α and β for the first workload (α_1 and β_1) and for the second workload (α_2 and β_2). α_1 and α_2 range between 1.1 and 10.0, while β_1 and β_2 range between 2.0 and 2000.0. Using these settings, the first and second workloads have hit rates (when run in isolation) that range between 2% and 99.999%, allowing us to explore all possible combinations of workloads with high and low hit rates.

We find that the individual α_1, α_2, β_1, and β_2 parameters chosen do not affect the hit rate of the mixed workload. The only factors that affect the hit rate of the mixed workload are the hit rates of the two input workloads. Thus, any two workloads with different α and β parameters but the same hit rate (when run in isolation) are interchangeable from a performance standpoint.

The second question we address is: *What is the hit rate of the mixed workload based on the hit rates of the component workloads in isolation?* To do this, we simulate and explore mixtures of two workloads, where each workload ranges from 2% hit rates to 99.999% hit rates, and measure the hit rate of the mixed workload from the simulation.

Figure 4.12 and Figure 4.13 depict the results of this experiment. The hit rate of the first workload is varied over each row of graphs in these figures. Each graph varies the hit rate of the second workload on the x-axis. Each graph shows the results using both the linear-combination "Average HR" estimate for cache penalties in the mixed workload (used in Section 4.6.2) as well as the simulation results labeled as "Simulation HR".

The left graphs depict the hit rate of the mixed workload, while the right graphs depict the modeled response time based on the hit rate depicted in the left graph.[2]

The left graphs in Figure 4.12 and Figure 4.13 compare the estimates for the hit rate in the mixed workload based on (a) Simulated HR: simulating the component workloads running together and measuring the simulated hit rate, and (b) Average HR: averaging the hit rates of the component workloads together. The graphs show that the simple average almost always overestimates the hit rate of the mixed workload, by up to 30%. The difference between these two estimates is smallest when both workloads have either an extremely large or small hit rate, and are somewhat small when either one of the workloads has either an extremely large or small hit rate. The difference is greatest when the workloads both have "mid-range" hit rates between 30% and 90%

It is difficult to determine whether the 30% error between the simulated and average estimates of mixed workload hit rates is significant or not. The right graphs in Figure 4.12 and Figure 4.13 use the estimated mixed workload hit rates for both simulation and the average estimates to model the expected query response time in the mixed workload. We find that the error between the estimates are, in fact, significant, and can lead to differences in response times up to a factor of 2 (this factor should be even larger when considering higher loaded systems). The error is particularly egregious when both the component workloads' hit rates are low. Increasing either one of the workloads' hit rates gradually reduces the error in estimating the response time, despite the fact that the hit rate estimates may have large error.

The data presented in Figure 4.12 and Figure 4.13 can be used to improve the estimates used by IDD. Using the measured miss rates of each of the component workloads in isolation, one can look up the hit rate of the mixed workload in these Figures, according to the simulation. Using knowledge of the cache-miss penalties of the CPU (which are easily measured), one can then model the cache miss penalty expected in the mixed workload. While this approach should lead to more accurate predictions by IDD, it has yet to be validated.

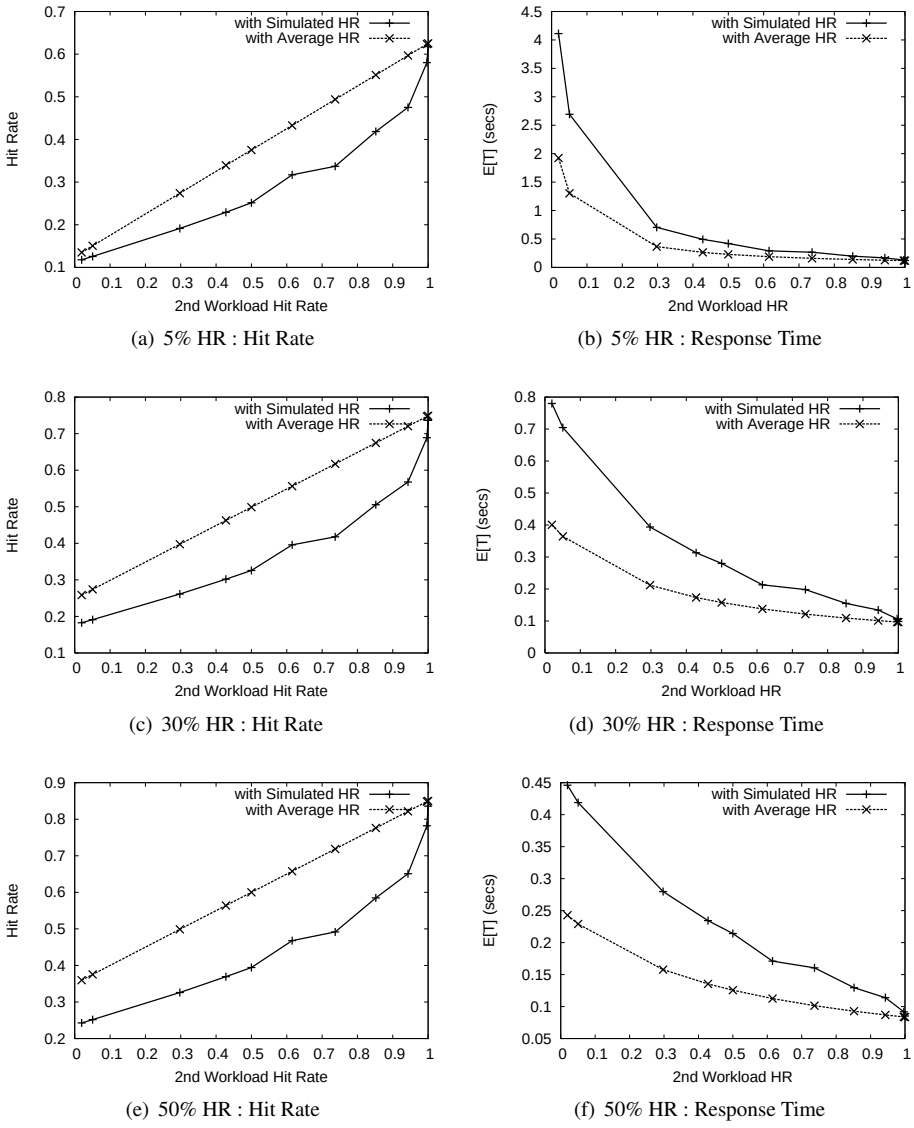
4.8 Prior Work

In practice, DBMS users are greatly concerned that their performance will degrade if they share their DBMS with second class of users. There is little research, however, on the magnitude of this degradation, and how to control this degradation using admission control.

Some studies [35, 14, 16] examine the performance effects of mixing two workloads on the same DBMS. These studies all use specialized micro-benchmarks that are easily parameterized for experimental purposes. Much of this work focuses on understanding issues of varying the amount of data shared between workloads. In contrast, we study much more complicated TPC [21] benchmark workloads, which we believe are more representative of real-world workloads. Our work does not consider the effects of data sharing between workloads, which can be important. Furthermore, while existing work does not directly model and predict the performance of mixed workloads, we develop the initial steps in this direction.

Most research on admission control [41, 47, 19, 49] addresses DBMS with only a single class of users, and attempts to choose the overall MPL to either maximize throughput or minimize overall mean query response time. It is difficult to apply the results of this research to our 2-class scenario.

[2] The modeled response time uses a simple Markov model with a single CPU server with a service rate of $HitRate \cdot CacheHitTime + (1 - HitRate) \cdot CacheMissTime$. $CacheHitTime$ and $CacheMissTime$ are measured on our DBMS hardware as being 0.0053 seconds and 0.0343 seconds, respectively.



(a) 5% HR : Hit Rate

(b) 5% HR : Response Time

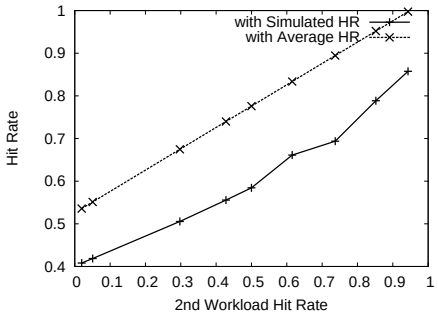
(c) 30% HR : Hit Rate

(d) 30% HR : Response Time

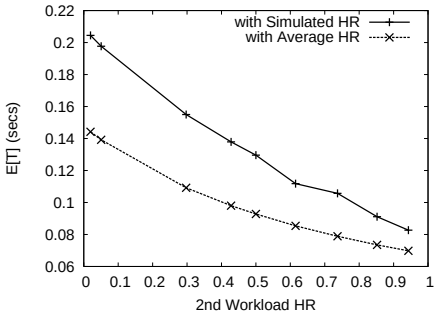
(e) 50% HR : Hit Rate

(f) 50% HR : Response Time

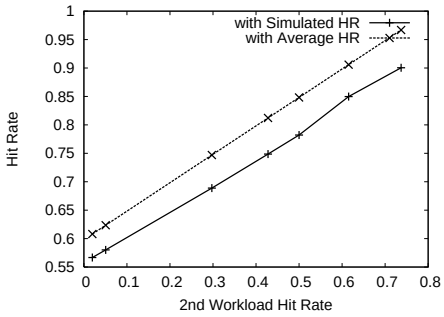
Figure 4.12: Simulated hit rates (left) and response times (right) for a mixed workload comprised of two stack-depth workloads as a function of the first workload hit rate, and the second workload hit rate. Each graph shows the results determined from simulating two stack-depth workloads ("Simulated HR") and using a simple average of the two workloads ("Average HR"). First workload hit rates range from 5% to 50%.



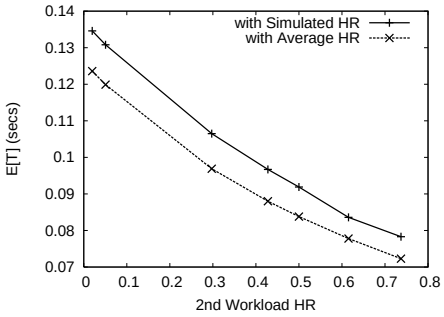
(a) 85% HR : Hit Rate



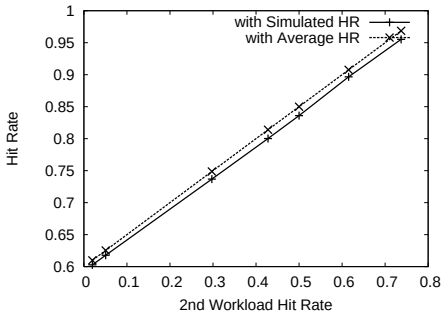
(b) 85% HR : Response Time



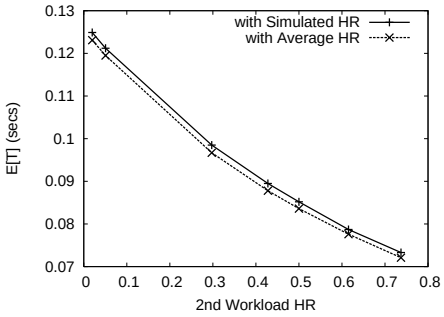
(c) 99.6% HR : Hit Rate



(d) 99.6% HR : Response Time



(e) 99.999% HR : Hit Rate



(f) 99.999% HR : Response Time

Figure 4.13: Continuation of Figure 4.12. First workload hit rates range from 50% to 100%. Simulated hit rates (left) and response times (right) for a mixed workload comprised of two stack-depth workloads as a function of the first workload hit rate, and the second workload hit rate. Each graph shows the results determined from simulating two stack-depth workloads ("Simulated HR") and using a simple average of the two workloads ("Average HR").

Some pioneering work on admission control [87, 83, 86, 70, 71, 33] uses queueing-theoretic models to predict DBMS performance. While these approaches also focus on understanding overall throughput with only a single class of users, the models can, in theory, be modified to incorporate two classes of users with admission control. Unfortunately, we show in Section 4.5.3 that these existing models do not sufficiently predict response times in real systems with multiple classes of users, due to effects such as spin locking and CPU stalling.

Thereska et al. [81, 80] and Narayanan et al. [60] instrument storage systems and DBMS to collect device demand statistics similar to our approach. They use these statistics to model system performance to answer "what-if" questions regarding changes to system configuration (such as buffer pool sizes, replication, etc). Their approach differs from ours because it requires modification to the DBMS to collect device demand statistics, which is impossible to do on legacy systems and can be very costly. In fact, we find that, with respect to performance in our commercial DBMS, collecting performance statistics can degrade performance by a factor of 3. Our approach, in contrast, requires only non-invasive statistics collection that does not hurt overall system performance.

Schroeder et al. [72] and McWherter et al. [56, 57] (see Chapter 2 and Chapter 3) focus on prioritizing one DBMS workload over another on the same DBMS. McWherter et al. [56, 57] (see Chapter 2 and Chapter 3) use priority scheduling inside the DBMS to prioritize one class of users over another. The approach here requires modifying the DBMS source code to implement prioritization internally. There is no admission control used whatsoever.

Schroeder et al. [72] combine admission control and priority scheduling inside the admission controller to prioritize high-priority users. Their approach uses a *single* MPL to limit the total number of users (both high- and low-priority). This may force high-priority users to wait outside the DBMS indefinitely, if low-priority jobs are long, which is very different from our goal of insulating the high-priority from low-priority users. The reason that we only place admission control on the low-priority users is that it is often politically or technically difficult to convince high-priority users to access their own DBMS through the admission controller.

4.9 Conclusion

This chapter studies sharing of DBMS between two sets of users, Locals and Federators, with the goal of limiting the effect of Federators on Locals, by appropriately setting the Federator MPL. The chapter makes two primary contributions:

First, we make the surprising discovery of the Local response time hump as the Federator DB size is increased. The hump determines how well admission control can isolate the performance of Local queries from Federator queries, and directly affects how Federator MPL should be chosen. Within the hump, even minute increases in the Federator MPL greatly degrade Local performance. We show that conventional queueing models for DBMS do not predict the existence of the hump.

Second, we introduce a new modeling approach, called Isolated Demand Decomposition (IDD), which uses demands computed from Locals and Federators running in isolation to predict the performance of Locals and Federators running together. IDD allows us to predict both the existence and the magnitude of the hump with unprecedented accuracy. We estimate the combined Local and Federator mean response time with error of 11.7%, and the Local response time with error of 17%, over the experimental range.

The direct impact of IDD is that DBMS administrators have, for the first time, a tool that predicts how Local response times change when a new Federator workload is added. Administrators will also be able to predict how setting Federator MPL with admission control affects Local (and Federator) response times.

The broader impact of this work is that it takes the first steps in predicting the performance of workloads when they mix. Variations of the IDD approach should apply to many scenarios with mixing workloads in systems ranging from DBMS, storage systems, and web servers. We also believe that IDD will also be useful in single-workload systems with background tasks, such as buffer pool cleaning or disk scrubbing. IDD should allow us to design background tasks so that they do not hurt the performance of the primary task too much.

4.10 Impact

The primary contributions of the research in this chapter are (i) discovery of a distinctive performance trend seen in DBMS: the Hump, (ii) conclusive proof of the underlying causes of the Hump, and (iii) the development of a new modeling approach, called Isolated Demand Decomposition (IDD), which can accurately predict the performance of mixed DBMS workloads where preexisting models fail.

The discovery of the Hump, and the identification of its underlying causes, has a significant impact for both (i) DBMS researchers as well as (ii) commercial users and administrators of DBMS. (i) DBMS researchers must make use of the fact that the microarchitectural design of modern CPU(s) can have a huge impact on DBMS performance, and can hurt performance by orders of magnitude. These factors can simply no longer be ignored if we want accurate performance models for modern systems. (ii) Commercial users and administrators of DBMS must be aware of the existence of the Hump when designing and tuning their DBMS. Understanding of the trend can help guide the design, development, and tuning of DBMS-based applications to help ensure these applications avoid the poor performance at the peak of the Hump. In particular, applications may be designed and tuned to improve locality and reduce the CPU cache pressure, minimizing the performance penalties that lead to the Hump. For instance, queries that operate on the same data can be scheduled together, so they are not interleaved with queries that operate on different data.

The IDD modeling approach has a direct impact on both (i) DBMS researchers and (ii) commercial DBMS users. (i) The direct impact from a research standpoint is that IDD pioneers the use of analysis to study admission control in DBMS shared by many users and in providing performance isolation to high-priority queries. IDD also demonstrates how to augment simple queueing models to incorporate the fundamental performance issues central to modeling the performance of modern CPU(s) in shared DBMS. This will lead to more accurate DBMS performance modeling, and also more accurate modeling of many other computer systems. (ii) The direct impact that IDD has for users and administrators of commercial DBMS is that, they have, for the first time, a tool that can advise how to configure the admission control MPL (how many low-priority queries should be admitted to the DBMS) to provide performance isolation to high-priority queries. Administrators will have accurate quantitative predictions of how the performance of high-priority queries will be hurt when adding additional low-priority queries to the DBMS. The primary result is that administrators will need to expend less time and effort in order to configure admission control in such situations, saving considerably on costs (since DBMS administrators' time is expensive).

The broader impact of this work is that it helps to understand the performance of running multiple workloads on the same system. Nothing ties IDD only to the study of DBMS: the techniques developed in this chapter do not depend at all on the internal design and architecture of DBMS (or only do so superficially). IDD should apply to most systems and workloads with both significant CPU and I/O components. Such systems are far-reaching and many, ranging from storage systems, to operating systems, to distributed systems, and so on. For example, storage systems often have to accommodate many background tasks, such as data scrubbing, which are low-priority since they should not interfere with the I/O requests from users. IDD may be used, for instance, to predict the performance isolation that the (high-priority) user I/O requests will receive when background tasks are run (and the number of concurrent background tasks), and can be used to determine when to run them. Thus, the performance of user I/O requests in storage systems may be improved.

Another major impact of IDD is that we can use it to help understand how to deploy systems and provision hardware. A common deployment question is whether to collocate two applications, such as a DBMS and a web server on the same hardware, or to put them on different hardware. We can use the IDD approach

to model the performance of running the applications on the same hardware or on separate hardware, and use that to make such decisions. In this case, IDD requires us to examine the DBMS workload in isolation, and the web server workload in isolation, and then we can use the same methodology to predict the performance of the mixed DBMS and web server workload.

4.11 Future Directions

The immediate future directions of this research are to evaluate and validate the accuracy of the IDD methods on (i) different workloads, (ii) different DBMS implementations, and (iii) sharing a DBMS among more than two workloads at a time. The scope of this chapter had to be restricted so as to make sufficient research progress, and focus on a finite set of performance issues. Despite this fact, there are many other DBMS implementations widely found in industry, and every different DBMS-based application produces a different DBMS workload. The variations between these DBMS implementations and workloads are poorly understood, and it is important to know whether IDD applies in all possible contexts (and if not, what needs to be addressed to improve the analysis). Likewise, it is important to understand whether there are factors which reduce or exaggerate the magnitude of the Hump.

One of the more involved directions for future research is to focus on evaluating and modeling the performance of shared DBMS which are I/O- or Lock-bound. Just as CPU devices have microarchitectural performance issues that can cause significant performance problems (the Hump) when workloads share a DBMS, similar issues may arise for I/O devices and Lock resources. While much research has been done on predicting the performance of locking in DBMS [71, 86], this research focuses on DBMS with a single workload (with no prioritization). Little research has focused on DBMS with multiple workloads of different priorities, as those considered in this research.

One of the keystones of IDD is the measurement of device demands throughout the DBMS. IDD relies on rather coarse estimates for device demands: time-average statistics collected for all the queries running together in the DBMS. This is due to the fact that we require the statistics to be collected without assistance from the DBMS (so that IDD is applicable in legacy DBMS). The primary consequence of this fact is that workloads have to be run separately to construct a device demand profile for the workload. If device demands could be collected on a per-query basis, IDD could construct a model of separate workloads even after they have been mixed on the DBMS. This would make IDD even more powerful, as we could reason about how different sets of queries contribute to performance. The primary difficulty is to determine how to measure and record per-query device demand statistics while minimally hurting overall performance. This requires collecting and storing huge amounts of data, which can be difficult to manage in most scenarios. Furthermore, it is an open problem to determine how to account for per-query device demands on certain devices. One such device is I/O, where an individual I/O request may be issued due to one or more queries. For example, when a query needs to access data, it may first have to evict a page from another query from the buffer pool, which may require writing that page to disk. It is unclear which query should take responsibility for this write.

Another open route for future research is how to more accurately predict CPU cache miss rates and penalties when two workloads share a DBMS. IDD relies on accurate predictions of the CPU cache miss penalty in the mixed workload, based on the cache miss penalties of each workload in the mixture. In particular, IDD uses a simple linear combination of the measured number of instructions per second (IPS) retired by the CPU(s) in each workload running in isolation to predict the IPS of the mixed workload. While

this is an effective predictor in the systems considered within this research, there are many other systems for which the predictor is *inaccurate* (by factors as large as 30% to 50%). We need more accurate estimates of CPU cache penalties to be truly applicable to many real-world systems. Section 4.7 formulates a strong basis of this work, by simulating CPU cache performance for workloads which run concurrently. Better prediction of CPU cache performance will improve the accuracy of IDD, as well as increase our ability to predict the performance in other computer systems.

Chapter 5

Conclusions

5.1 Conclusion

This thesis examines new methods for providing performance isolation to queries running in a DBMS. Two problems are considered:

First, it addresses how to prioritize high-priority queries within a single OLTP or transactional web workload comprised of high- and low-priority queries. The goal is to ensure high-priority queries get the same performance as if running alone, without low-priority queries.

Second, it addresses how to use admission control to protect the performance (query response times) of one transactional web workload (Locals) from a second workload (Federators) running on the same DBMS.

In the process of addressing the above problems, this thesis introduces and develops several *tools* and *analysis techniques*. Both the tools and techniques can be applied to a range of DBMS and other systems to provide performance isolation. The following Section 5.1.1 describes the key tools, and Section 5.1.2 outlines the key techniques put forth in this thesis.

5.1.1 Tools

The two primary tools developed for providing performance isolation in this thesis are Preempt-On-Wait (POW) lock scheduling (discussed in Part II), and Isolated Demand Decomposition (IDD) modeling (discussed in Part III).

Preempt-On-Wait (POW)

POW is a new scheduling policy that provides performance isolation for high-priority queries in a DBMS.

POW fills a void in the space of existing DBMS query prioritization algorithms, and is able to outperform existing scheduling policies. Many scheduling policies have difficulty isolating high-priority queries from low-priority queries in lock-bound OLTP workloads, which either hurts high- or low-priority queries too

much. POW is able to provide near-perfect isolation to high-priority queries, without hurting low-priority query response times too much.

POW is relatively easy to implement in a DBMS. Only two changes are required: (i) the DBMS must store and have a way to set the priority class of each query (some commercial systems support this already), (ii) the DBMS must change the order that it wakes up waiters for locks, (iii) the DBMS must set a flag on a query whenever a query of higher priority waits for it, and (iv) the DBMS must be able to abort and roll back lower-priority queries when they need to be preempted (most DBMS have the primitives to support this already, due to ACID compliance).

Isolated Demand Decomposition (IDD)

IDD is a new queueing modeling approach that models and predicts the performance of two workloads when they run together on the same DBMS. IDD predicts the performance of each workload when run together, based on statistics and measurements made when those workloads are run by themselves on the DBMS.

The key problem that IDD addresses is that many of the assumptions that queueing theory relies on are violated in DBMS, particularly those running multiple workloads. As a result, standard queueing models produce not only inaccurate predictions, but completely miss dominant performance trends. I show that existing models often fail due to their inability to model variable efficiency of the DBMS CPU. IDD augments queueing models to properly account for how DBMS CPU efficiency changes when workloads mix on the DBMS, to produce accurate performance predictions.

IDD is designed to be minimally invasive, so that the statistics and measurements it needs can be easily collected from standard operating system and CPU counters, and the DBMS does not need to be modified or instrumented to collect this data. As a result, IDD is widely applicable, even in the legacy systems found at many companies.

5.1.2 Analysis Techniques

In addition to developing the new tools described above, this thesis contributes a blueprint for both (i) analyzing DBMS performance, and (ii) identifying and resolving performance problems. The approach is based on results and models from queueing theory, especially bottleneck analysis, scheduling policies, operational laws, and Markov modeling.

This thesis (particularly in the analysis sections of Part II and Part III) details the process of instrumentation, measurement, and analysis necessary to understand DBMS performance issues. This process helps DBMS developers and administrators to make intelligent design and tuning decisions to more quickly and easily meet performance goals.

Without such strong analytical support, it would be extremely difficult to determine where to concentrate engineering effort, and what changes would need to be made to yield the desired effect. DBMS are extremely complex systems, comprised of many subsystems that interact in many ways. When DBMS are shared between multiple workloads, those interactions are even more complicated.

The analysis process developed in this thesis was essential to the development of the tools outlined above in Section 5.1.1. In Part I and Part II, for instance, while CPU and I/O utilization are often both high, a more detailed bottleneck analysis reveals that locks are the true source of performance problems. Likewise, in Part

III, isolating individual workload resource demands in a multi-workload DBMS reveals that CPU demands change for the worse when the workloads are mixed.

5.1.3 Impact

When a DBMS serves a workload comprised of both high- and low-priority queries, or when a DBMS serves multiple query workloads, many performance issues arise due to how queries share the DBMS and its resources. In general, users do not want queries to share the DBMS in an uncontrolled manner, because (i) users have specific performance goals that they want the DBMS to meet, and (ii) uncontrolled sharing can result in slow and/or unpredictable performance.

The primary impact of this work is to provide the effective tools necessary to *control* the way that queries share a DBMS. Specifically, it provides tools to ensure *performance isolation* to the high-priority users in a single DBMS workload, or to one of the workloads in a DBMS shared by multiple different workloads.

Cost Savings

The clearest impact of this work is that companies can use the tools and techniques developed here to save on costs. Cost savings are primarily achieved by helping companies to meet their performance goals while using less powerful and cheaper DBMS hardware.

DBMS hardware costs are often extraordinarily high because high-end DBMS applications absolutely need the performance and functionality provided by the highest-end hardware ("server grade hardware"). It is expensive to design and build high-end hardware, and at the same time, the market is relatively small. As a result, high-end hardware does not enjoy the economies of scale that make commodity hardware affordable.

Using less powerful and cheaper hardware leads to secondary cost savings: **First,** using less powerful hardware often results in lower power consumption and less power dissipation. As a result, companies spend less on energy and cooling costs at their data centers. **Second,** if a company can delay a hardware upgrade, they can save on the time-consuming work of tuning the DBMS to run on new hardware. Given that DBMS administrators' time is usually expensive, less tuning can result in huge cost savings.

POW lock scheduling and IDD, the key developments in this thesis, both help to achieve the cost savings described above.

POW lock scheduling helps realize cost savings by providing query prioritization when all users of a DBMS do not have the same performance requirements: some high-priority users need very low response times, while the other low-priority users need only best-effort response times. Without query prioritization, the DBMS can use hardware wastefully, to make *all* queries faster, and not just the important high-priority queries. With POW, high-priority queries run almost in isolation, meaning that less powerful hardware is needed to achieve the same level of performance.

IDD helps realize cost savings in two ways:

First, if two workloads share a DBMS, and one workload has stronger performance requirements than the other, IDD helps administrators use admission control to ensure that workload gets the necessary performance. This use is very similar to the use of query prioritization as described above for POW, but since it uses admission control instead of internal scheduling, it works on any DBMS. As a downside, admission control has less control over sharing after queries have been admitted to the DBMS.

Second, IDD can serve as an adviser to help administrators provision their systems. Often, DBMS need to be upgraded to handle increasing load, or additional functionality. It is typically extremely difficult to predict the performance of a DBMS workload on new hardware, and thus, difficult to know how much new hardware must be purchased. Using the models described in IDD, administrators can quickly and easily get a better understanding of the performance benefits of new hardware before making any purchases or performing costly performance testing.

User Satisfaction and Profit Maximization

Providing performance isolation to users in a DBMS can lead to increased revenues and profits due to increased user satisfaction. The reason is simple: without performance isolation, performance can be slow or unpredictable, which is frustrating for users and disrupts their flow of thought [61].

On the Internet, the Google maxim of "fast is better than slow" [36] is the rule of law. When users have two services that provide generally similar functionality, they usually prefer and choose the faster one [59, 69]. As a result, faster systems get more users, greater mind share, and greater market share, all of which lead to increased sales and revenues. Using the tools developed in this thesis, such as POW and IDD, DBMS administrators are better able to ensure that users receive acceptable performance.

Reliable Feature Deployment

Adding new tasks and functionality to existing DBMS applications, especially online services and e-Commerce sites, is a process fraught with difficulty. Accurately predicting the performance ramifications of adding a new feature is particularly challenging. Real-world systems often address this problem with incremental deployment of new features, which can introduce additional complexities to support the incremental upgrade, and cause user confusion (since different users see different versions of the service).

IDD improves administrators' ability to predict the DBMS performance of the upgraded workload, based on the performance of the original workload and basic properties of the additional features. IDD can be used in this way long before attempting to deploy new functionality, which gives developers, designers, and administrators additional time to prepare for the deployment of upgrades. IDD has the potential to have a huge impact on making upgrades smoother and reducing performance and other technical difficulties that may otherwise arise.

IDD applies because many upgrades and feature additions simply add an additional query workload to a DBMS processing an existing query workload. Thus, the existing workload can be considered the first (Local) workload and the additional features can be considered the second (Federator) workload as seen in IDD. IDD can predict the response times of the original workload queries, the new workload queries, and the aggregate workload queries when they run together on the same DBMS.

5.1.4 Lessons Learned

DBMS must collect more statistics

Many commercial DBMS collect many measurements and performance statistics that are necessary to conduct the analysis outlined in this thesis. They do not, however, collect all of the necessary data, making

it impossible to build a clear and complete picture of the system without doing significant work. Some of the data can be gathered from the operating system or CPU-level counters, but other data requires further instrumentation of the DBMS to acquire. This is impractical for production DBMS.

This reveals two problems that must be addressed: First, DBMS must provide support for collecting statistics from all major subsystems, including the lock manager. At the very least, the DBMS must provide hooks by which users can instrument the system themselves. Second, DBMS should collect all relevant data (e.g. from the operating system and CPU) and present it all as a unified picture so that administrators and users can better tune their systems. Doing this work *once* at the DBMS prevents it from being re-implemented and re-invented (potentially incorrectly) by every DBMS user.

Understand the bottleneck resource

Understanding the bottleneck resource is critical to understanding and improving system performance. Part I and Part II of this thesis show that implementing prioritization on non-bottleneck resources provides limited or no performance isolation for high-priority queries. Implementing prioritization at the bottleneck resource, on the other hand, yields much better performance. In fact, Part I and Part II show that scheduling CPU resources when locks are the bottleneck is no better than having no prioritization at all. Furthermore, scheduling locks when locks are the bottleneck can improve high-priority query response times by a factor of 3 over CPU scheduling (this factor can be arbitrarily high, as it is a function of system load).

As a result, the DBMS cannot (as widely believed [2]) simply implement CPU scheduling and rely on CPU may be prioritization to implicitly give queries prioritized access to other resources. At worst, this approach will have no effect (as I find), and at best, the prioritization will be sub-par. DBMS must implement query prioritization at all major system resources that may become the bottleneck, including the lock manager.

Understand Device Demands

As discussed previously, DBMS violate general assumptions that queueing theory tends to rely on. One of the major violations is that *DBMS are not work conserving*: the amount of work that each query brings into the system is not fixed, and instead depends on the state of the DBMS. That is, the *device demands* (for each device) that each query brings to the DBMS are variable. Ordinarily, queueing theory is unable to cope with systems that are non-work conserving. The result is typically a gross failure to accurately predict DBMS performance. This failure is exemplified in Part III of this thesis, in Section 4.5.3.

With the introduction of IDD, thesis makes new strides towards using queueing theory to correctly model DBMS performance, even in the face of variable device demands. The key idea is to measure device demands and predict how they change, and use that data into queueing theoretic models.

Preemption is powerful; Selective preemption is more so

Preemption is a powerful tool, and can eliminate many performance roadblocks in DBMS and other systems, and can drastically improve high-priority query response times. Preemption does, however, come with significant drawbacks, including the potential to starve low-priority queries. The problem is that preemption is indiscriminate, and kills many queries that have no significant effect on performance. *Selective preemption*

(or conditional preemption) can provide as much benefit to high-priority queries as unconditional preemption, while keeping low-priority queries from starving. This is demonstrated with lock scheduling in OLTP DBMS in Part II.

The key to selective preemption is what condition(s) low-priority queries should be preempted. Experimentation with existing conditional preemption algorithms in Part II reveals that using different conditions yields very different performance. While other researchers have indicated success with these other algorithms, it is clear that they are not as effective as Preempt-On-Wait (POW) in the OLTP workload and DBMS context.

A few queries can hurt everyone

The performance of a very small number of queries in a DBMS can greatly affect the performance of the rest of the queries in the DBMS. In Part II, it was seen in OLTP workloads that high-priority queries often get stuck waiting for a few low-priority queries to finish (their "excess") before being able to continue. Using preemption to kill (and then restart) these few low-priority queries, which account for two percent of the queries in the workload, is able to improve high-priority query response times by a factor of 3.

The primary cause for the above behavior is that some resources, such as locks, can be held for arbitrarily long periods of time once they are allocated (unlike, for instance, CPU, which is allocated only in small time slices, and shared between all queries). This, combined with large variability in query response times results in unpredictably large waiting times for those resources.

Micro-architectural performance is important

Often, when analyzing DBMS performance, researchers build queueing models that represent major system resources — CPU(s), disk drive(s), lock(s), etc. Unfortunately, the micro-architectural characteristics of these resources, particularly that of CPUs, are starting to have a major effect on DBMS performance.

The analysis for IDD in Part III of this thesis shows that the behavior of the CPU cache and CPU efficiency (instructions per second) can dominate and significantly hurt overall query performance. Queueing models that fail to capture these trends fail to accurately predict DBMS performance, and can completely mispredict the performance trends.

Monitoring CPU counters that measure the behavior of the CPU micro-architecture (such as cache events, instruction completion rate, etc) has relatively little overhead, but provides a wealth of data which can help identify and address performance problems. Stochastic sampling of these counters can also provide more comprehensive pictures of the CPU behavior (since only a few CPU counters can typically be active at any one time) and further reduce the performance overhead.

5.1.5 Limitations and Real-World Applicability

There are number of limiting assumptions are made in this thesis work. While these assumptions make the research tractable, they make it less clear how the research impacts real-world systems. Here, we discuss these limiting assumptions, and how they affect the application of this research to real-world systems.

The main limitations of this work are in (i) the number of DBMS implementations studied, (ii) the

number of workloads studied, (iii) the arrival processes studied, and the (iv) data distributions and data access patterns studied. These issues are discussed in turn below.

DBMS Implementations

Throughout this thesis, only a limited number of DBMS implementations are studied: IBM DB2, PostgreSQL, and Shore. Chapter 2 focuses on all three DBMS implementations to discover their individual bottlenecks, Chapter 3 focuses on developing new lock scheduling algorithms on IBM DB2 and Shore, while Chapter 4 focuses on modeling IBM DB2 with IDD.

There are many other DBMS implementations available, the most important of which are MySQL, Microsoft SQL Server, and Oracle. Furthermore, there are many versions available of each DBMS, and we consider only on a specific version of each. It is not immediately clear how the results in this thesis apply to different DBMS implementations and versions of DBMS implementations.

In regard to the bottleneck analysis of Chapter 3, we believe that the general trends in that chapter are representative of what would be seen in other DBMS: DBMS using 2PL concurrency control will be lock-bound for TPC-C, and those using MVCC (and variants) will be either CPU-bound or I/O-bound for TPC-C. This is due to the fact that the TPC-C workload has a significant amount of data contention, and only two methods are available to deal with this contention: waiting (in the case of 2PL) or versioning (in the case of MVCC). Likewise, we expect that TPC-W will almost always CPU-bound, regardless of the DBMS implementation, due to the fact that there is little lock contention and that TPC-W database sizes are typically very small, and the database is almost always resident in memory. When TPC-W database sizes are scaled dramatically, and do not fit in memory, the workload becomes I/O-bound. While some DBMS implementations may handle memory and I/O more or less efficiently than others, the implementations are not so different so as to expect radically different performance trends.

In regard to the implementation of Preempt-On-Wait (POW) lock prioritization in Chapter 3, we expect POW to be similarly effective on most DBMS, since DBMS lock subsystems are implemented similarly (at least conceptually). We believe that POW will only prioritize effectively when locks are the bottleneck, which means that it should work best on DBMS using 2PL concurrency control. DBMS using MVCC (and variants) should benefit whenever concurrency is very high and locking is significant, but even then, the impact should be weaker than for 2PL-based DBMS.

In regard to Isolated Demand Decomposition (IDD) modeling in Chapter 4, different DBMS implementations will change the device demands that queries bring into the DBMS and may change the way those device demands change when multiple workloads share the DBMS. The methodology of measuring performance data of each workload in isolation should remain unchanged, and the process of combining those measurements should be similar. There may be architectural issues that arise that affect how queries mix (such as spin locking and cache effects have done in IBM DB2) that will need to also be modeled. The framework should, however, remain unchanged.

In regard to The Hump in Chapter 4, the magnitude of each factor that gives rise to the hump will likely change as the DBMS implementation changes. Thus, we expect that as the DBMS implementation changes, the hump will appear in different regions of the configuration space, and will appear with different magnitudes. The underlying factors that drive the hump, especially the increase in pressure on CPU caches due to forcing increasing amounts of data through the CPU, however, should be present in all systems.

Workloads

This thesis only considers a limited number of workloads: TPC-C and TPC-W. Chapter 2 focuses on both workloads for bottleneck analysis, while Chapter 3 focuses only on lock prioritization in TPC-C and Chapter 4 focuses only on modeling the performance of TPC-W with IDD.

Ideally, we would test our methods on all real-world workloads, but this is infeasible, since every DBMS application generates a different DBMS workload, and real-world DBMS workloads are not available for academic research. While TPC-C and TPC-W are not real-world workloads, they have been designed to be representative of many of the workloads seen in real-world systems. The faith that industry places on these workloads suggests that our results should carry over to at least some real-world systems.

In regard to the bottleneck analysis done in Chapter 2, we expect that bottleneck trends for real OLTP and e-Commerce applications will follow trends similar to those seen by TPC-C and TPC-W, respectively. In particular, in 2PL-based DBMS, we expect to see many more data dependencies and more lock waiting in most OLTP applications than in e-Commerce applications. When the application is performance-critical, however, we expect that application designers will follow in the footsteps of Amazon.COM [26] and sacrifice application "correctness" (data consistency requirements) so as to reduce the need for locking. This will never be feasible for some applications, however, and even Amazon does not eschew data consistency in all areas of their application. In these applications (or areas), we expect to see lock bottlenecks arise again.

In regard to the POW prioritization done in Chapter 3, we expect that POW will only be effective for some lock-bound workloads. POW relies on specific statistical properties seen in the TPC-C workload, and to be effective on other workloads, these properties must also be found in those workloads. Without access to real-world workloads, however, it is impossible to judge whether these properties are common in the real-world. Fortunately, the statistical properties are easily measured. Thus, a DBMS engine could easily measure statistics (the same statistics we describe in Chapter 3) in the locking subsystem, and determine whether POW, or another prioritization algorithm would be most effective.

In regard to the IDD modeling done in Chapter 4, the essential aspects of the modeling approach are relatively workload-independent. In particular, once device demands have been attained, solving a queueing model using those demands should produce reasonably correct results. There are three potential problems when looking at other workloads, however: (i) predicting how two workloads' CPU demands change when they run together, (ii) mixing workloads that share data, and (iii) Lock-bound and I/O-bound workloads. (i) The techniques that were effective at predicting how CPU demands change when two TPC-W workloads mix may not be effective at predicting how CPU demands change for other workloads. Chapter 4 outlines the approach necessary to measure how device demands change, and how to validate such a prediction. The major issues affecting CPU demands (spin locking and CPU cache pressure) are, however, likely to be present in all workloads. (ii) When mixing workloads that share the same data, the potential to increase the amount of locking in the workload is also increased. There has been some research in characterizing the performance effects of different levels of data sharing [35, 14, 16], but little work on modeling and predicting the performance effects. (iii) Locks and I/O have very different performance characteristics than CPUs, and as a result, Lock-bound and I/O-bound workloads will likely have to be modeled differently than the largely CPU-bound (and occasionally I/O-bound) workloads considered in Chapter 4. Some prior work has modeled the effects of locking in DBMS [83], and may be used as a starting point to model locking within IDD. Unfortunately, all of these potential problems are non-trivial, and may need to be the focus of significant future work.

In regard to The Hump seen in Chapter 4, since the essential performance issues (spin locking and CPU

cache pressure) are nearly universal, we expect that hump-like trends will be seen in many workloads. The key difference, however, is that we expect that changing the DBMS workload will change both the region in which the hump occurs, as well as its magnitude.

Arrival Processes

Another major limitation in this thesis research is that we focus entirely on *closed systems*, where there are a fixed number of users who alternately think (for an exponentially distributed amount of time) and issue queries ad infinitum. While some DBMS are, in fact, closed systems (especially batch processing systems), many are not. Other DBMS may be *open systems*, where a potentially infinite number of users arrive to the DBMS according to some stochastic arrival process (often a Poisson process), issue a single query and depart (observe that closed systems approach open systems as the number of users in the closed system increases). Other DBMS may be *partly-open systems* [73], where an infinite number of users arrive to the DBMS and issue several sequential queries, and depart.

The performance differences between closed, open, and partly-open arrival processes can be significant.

The performance of a DBMS often depends on whether the system is closed, open, or partly-open. Open systems often have more performance variability than closed systems, because unlike closed systems, the number of potential users in the DBMS at any one time is *unbounded*. This can be a big problem especially when the DBMS is momentarily slowed down, as it can lead to incredibly large backlogs.

Furthermore, the performance of an open system depends on the exact nature of the arrival process. Some arrival processes are evenly spread out (e.g. a constant time between each query's arrival), while some are bursty (e.g. a dozen queries are likely to arrive at the same moment). In general, the burstier an arrival process is, the worse the DBMS performance, and the higher the performance variability.

It is important to understand the effect the arrival process has on DBMS performance, but it is a poorly understood issue, wide open for future work. We will discuss two issues: (i) *Openness*: whether the system is open or closed, and (ii) *Burstiness:* if it is an open system, how bursty the arrival process is.

We expect that in the bottleneck analysis done in Chapter 2, open systems and bursty open systems will both experience increased lock waiting, and be more likely to be lock-bound than closed systems. This is because it is well-known that lock wait times increases drastically with concurrency [84], and open systems can have (periodically) more concurrent clients than closed systems, and bursty open systems have even more.

If it is true that openness and burstiness lead to increased lock contention, it should make the impact of lock prioritization even greater. The POW lock prioritization described in Chapter 3 may be well suited to prioritize queries in the face of this increased contention, but it is difficult to determine whether it will be best. Fortunately, it is easy to measure the lock manager statistics in the that determine whether POW performs well or poorly, and enable POW only in the case that it performs well.

In regard to the IDD modeling done in Chapter 4, we believe that openness and burstiness in the Local and Federator workloads may have a wide range of effects on the results of IDD. The problem is that it is hard to tell whether the burstiness of each workload "lines up" (occurs at the same time), or is "staggered" (when Locals burst, Federators are idle, and when Federators burst, Locals are idle). The degree to which bursts overlapped must be characterized, and factored into the IDD modeling to properly understand how device demands change and how performance is affected.

Data Access Patterns

It is well-known that real-world data access patterns often follow power-law relationships, as is characterized by Zipf's Law [91]. The law specifies that the frequency of any item is inversely proportional to its rank in the frequency table. That is, $f_k = 1/k$, where f_k is the frequency of the k'th most popular item. In e-Commerce, this suggests that the most popular item is twice as likely to be accessed as the second most popular item, which itself is twice as likely to be accessed as the third most popular item, and so forth.

While the TPC-C benchmark does not use Zipf distributions per se, it does access data via a skewed non-uniform distribution (called `NURand`). Thus, we expect that using Zipf distributions for TPC-C will not significantly affect the results.

In contrast to TPC-C, TPC-W, treats all books in its database as if they were equally (uniformly) popular. Olston et al. [62] attempt to develop a more realistic workload by making books' popularity follow a Zipf distribution. In this Zipf TPC-W workload, books' popularities are set according to the relationship: $logQ = 10.526 - 0.871logR$, where R is the sales rank of a book and Q is the number of copies of the book sold within a short period of time.

We experimented with the Zipf TPC-W implementation to determine how Zipf distributions may effect the results of this thesis. We find that the biggest difference between Zipf TPC-W and the standard TPC-W implementation is that Zipf TPC-W is more CPU-bottlenecked than the standard TPC-W implementation. While Chapter 2 showed that CPU takes up 80% of queries' response times in standard TPC-W, CPU uses more than 93% of queries' response times in Zipf TPC-W. This is expected, since Zipf TPC-W is more likely to access a smaller set of the data, and is more likely to hit in the buffer pool, and avoid I/O.

In regard to The Hump and IDD of Chapter 4, Zipf TPC-W still exhibits the same type of Hump behavior seen in standard TPC-W. This result is, in fact, somewhat surprising. Since using a Zipf distribution reduces the amount of distinct data accesses, one would expect it to reduce the pressure on the CPU caches, and reduce CPU stalls. It appears, however, that in reality, this effect is not not strong enough to greatly reduce the impact of CPU stalling (possibly due to the fact that the CPU cache is still too small).

5.1.6 Future Directions

Device Demands

While IDD is able to get a good handle on the device demands from workloads running on a DBMS, it is limited in what it can accomplish. The biggest limitation is that IDD can only collect aggregate mean-value device demands for a query workload running by itself. If one had the ability to break down device demand measurements on a per-query basis (rather than in aggregate over all queries in the workload), it would open many directions for future analysis. For instance, IDD may be able to make predictions or help configure admission control without having to run workloads in isolation first; IDD could simply measure the device demands of each workload independently, and measure directly how their demands change as a function of time.

Unfortunately, DBMS generally do not provide per-query device demand measurements. When they do, those measurements are often inaccurate or taken at too low a resolution to be useful (for instance, per-query CPU demand measurements for transactional web workloads are reported by the DBMS as zero, even though other methods reveal that is not true). Two major challenges must be addressed to resolve this

problem. **First,** and foremost, DBMS vendor and user awareness of the importance of device demands must be raised. This thesis is a good step in that direction, but further progress is necessary. **Second,** it is difficult to do per-query device demand accounting at some devices. For instance, it is sometimes difficult what query is responsible for a given I/O request: when evicting dirty pages in the buffer pool, it is not clear whether to account those I/O requests to the query (or queries) who last modified the pages, or to the query (or queries) that need free space in the buffer pool. This second issue requires further study, to determine the correct measurement policies to make the measurements meaningful.

Other resources

Much of the work in this thesis has concentrated on CPU and Lock resources — focusing on CPU issues in Part III with admission control and Lock issues in Part I and Part II with query prioritization. The essential analysis techniques and approaches taken in this thesis should extend relatively easily to other resources, such as I/O devices, network, or memory.

As seen with CPU and Locks in this research, however, there are certainly performance-related details that will require the analysis to be fine-tuned. For example, Part II found that the structure of locking within the DBMS and lock holding times greatly affects the choice of scheduling policies. Likewise, Part III found that the behavior of the CPU at the micro-architectural level significantly affects the overall performance of the CPU and the DBMS.

I/O resources is likely to have micro-architectural performance issues which will need to be modeled in some way. Disk seek times are particularly interesting, as they can lead to significant response-time variability for I/O requests. Likewise, the "exotic" scheduling policies (e.g. elevator scheduling) used both in the operating system and the I/O devices themselves are likely to need special treatment, since they can result in hard-to-predict reorderings of queries' I/O requests.

Conventionally, memory is treated as a very different type of resource than those considered in this research. Instead of concentrating on the time memory requests take at the "memory resource," as is done with CPU, I/O, and Locks, researchers focus on how much memory to allocate to each query, query operator, or process. How much memory a query gets can significantly affect its performance [24], as the DBMS can choose more or less efficient algorithms given how much memory it has. The problem of allocating resources such as memory are not addressed in this thesis, and there are many open questions about how to do so in the context of this thesis. In particular, it will be important to understand how device demands change, depending on how much memory a query has (and thus, which algorithms are used).

In the future, however, it is possible (and even likely) that we will have to start modeling memory like the other resources in this research: as a "server" that queues and executes memory requests with some scheduling policy. This is for two reasons:

First, main memories are getting larger, and holding terabytes of data, if not the whole database in main memory, is increasingly realistic. Thus, memory allocation problems become much less significant.

Second, CPU architectures are moving towards multi-core packages, with many fast cores and fast on-chip memory within the package, all sharing a slow memory bus to access main memory. This memory bus is expected to become a performance bottleneck [8], especially as the number of cores grows. At the same time, on-chip memory must be much smaller than main memory, since they also must be much faster. Thus, data must often move between on-chip and main memory via memory requests. As these requests start queueing, the approaches used in this research become particularly suited to modeling memory system

performance.

CPU Cache behavior

One of the key components of the IDD analysis is to predict the rate of computation at the CPU(s) when two workloads mix on the DBMS. The CPU rate of computation depends on the CPU cache miss rate of the mixed workload. While the work in IDD uses a simple predictor of the CPU computation rate that works for the workloads considered in this thesis, it does not work in all situations, nor for all workloads.

Much research has been done to measure and predict the CPU cache miss rates of a single workload running on a CPU [78, 77, 34]. In general, this work does not focus on DBMS workloads, and is not easily extended to address multiple workloads.

The work in Section 4.7 begins to examine the nature of mixing workloads assuming that memory accesses and locality follow a particular "stack distance" distribution [78, 77]. In this thesis, much understanding has been gathered about how the locality of mixed workloads depend on the locality of each workload in the mix. Unfortunately, there is still a significant amount of work necessary to take this information and make it directly applicable to IDD. I believe that this is a tractable problem, however, and will result in IDD producing much more accurate predictions of performance.

Other workloads

Much of the research in this thesis focused on OLTP and transactional web DBMS workloads. Many other types of DBMS workloads are widely available, including but not limited to data warehousing, decision support, and ETL (Extract Transform Load). Furthermore, real-world workloads in each of these categories all differ from one another greatly. Even different benchmarks from each category can differ greatly.

The work in this thesis depends in varying degrees to the type of workload in consideration. Part I and Part II rather strongly focus on prioritization in OLTP and transactional web workloads. As a result, to determine the effectiveness of different scheduling algorithms on a new workload, it is necessary to reapply similar bottleneck and analysis techniques seen in Part I. Likewise, POW, described in Part II, will need to be evaluated in additional workloads, to determine whether its conditional preemption algorithm remains effective. Part III, which describes IDD, is a much more general framework, and should largely be independent of the workloads under study. Further evaluation is important, however, to ensure that its applicability is, indeed, as wide as believed.

Open versus closed workloads

One of the observations made while conducting experiments for this thesis is that the query arrival process had a large effect on performance.

All of the work in the thesis is done using a closed loop process: A fixed set of users, each of which repeatedly sends a query and then waits before sending the next query. An alternative would be an open loop process: Users arrive (from an infinite pool) at a steady stream according to a Poisson process (the time between arrivals fits an exponential distribution), submit a query, and never again return to the system.

In general, queueing theory predicts that a closed loop system approaches the behavior of an open loop

system as the number of users in the closed system goes to infinity (and the waiting time increases accordingly). Much of queueing theory, especially work that studies query scheduling, focuses on open arrival processes. This is largely due to the fact that these systems are typically easier to analyze: Markovian models of systems with open arrival processes usually have a much smaller state space than those with closed arrival processes.

When conducting experiments, I find that real-world DBMS do not handle open loop arrival processes very well. Consider a DBMS with a transactional web (TPC-W) workload, with queries arriving according to a Poisson process. If the arrival rate (average number of arrivals per second) is low, the load (the average number of concurrent queries in the DBMS) is low, then response times are good. Low load corresponds to an extremely under-utilized system, for instance, with CPU and I/O utilization under 50%. If the arrival rate is increased continuously, at first response times increase continuously (as expected), but then become *chaotic*, alternating from extremely high to low. It is interesting to observe that the use of admission control to limit the number of concurrent queries in the DBMS does not appear to eliminate the chaos, so it is not simply due to the fact that queries interfere with one another when there are many in the DBMS concurrently. It is also interesting to observe that this problem is often seen across different DBMS implementations (such as IBM DB2 and PostgreSQL).

The chaotic performance of DBMS with open arrival processes is a significant problem. There is debate over whether real-world arrival processes are more like open or closed arrival processes, particularly in online and e-Commerce DBMS systems [73]. If real-world DBMS arrival processes are open, it suggests that any system experiencing moderate to high loads (even transiently) will see significant performance problems and response time variability, which will hurt customer satisfaction.

Many companies often drastically over-provision their systems, so that they have enough hardware to almost always run at relatively low loads. While this approach helps ensure good performance, it is also extremely costly. Companies always want to reduce their operating costs, as long as sufficient performance can be maintained, and thus would prefer to run their DBMS at higher loads. Understanding the essential causes of chaotic performance for open arrival processes and developing tools and techniques to control it will yield huge cost savings for many applications.

The analysis approaches outlined in this thesis may be a good starting point for discovering the issues that cause chaotic performance problems. One significant difference, however, is that performance data will need to be analyzed and collected as time-varying processes, and simply looking at averages collected over an experimental run will not be sufficient. It is clear that the problems are transient, or time-dependent, as an experimental run can see good performance for a long period of time, and then see its performance fall apart.

Appendix A

Appendix: Workloads

A.1 TPC-W

TPC-W is of primary interest throughout the bottleneck analysis in Chapter 2 and the IDD modeling approach in Chapter 4.

TPC-W is an industry-standard benchmark designed by the Transaction Performance Council (TPC). The TPC consists of over 30 member companies, who helped to design the benchmark to better represent real-world system workloads.

TPC-W was designed by the TPC to model e-Commerce (Transactional Web) workloads, as seen in companies such as Amazon.COM. TPC-W simulates customers that browse and buy products (books) from an online retailer's web site, performing the same queries and tasks that would be done by a real customer: examining books, putting books into a "shopping cart," making purchases, searching for books, updating customer information, etc.

The TPC-W standard allows the relative amounts of browsing and buying in the workload to be adjusted, yielding different "mixes." Three basic mixes are standard: Shopping, Browsing, and Ordering. The difference between these mixes is simply the probability distribution of how simulated clients choose queries/tasks. For example, after performing a given interaction on a given web page, the client can choose an action to buy a product, or look for another product. In the Browsing mix, the client is more likely to view another product, while in the Ordering mix, the client is more likely to buy the product.

The benchmark specifies that simulated clients loop, alternately "thinking" and issuing queries. This is a closed-loop arrival process as seen in queueing theory, with the exception that the distribution of TPC-W think times is a truncated exponential, rather than the non-truncated exponential distribution typically used in queueing theory.

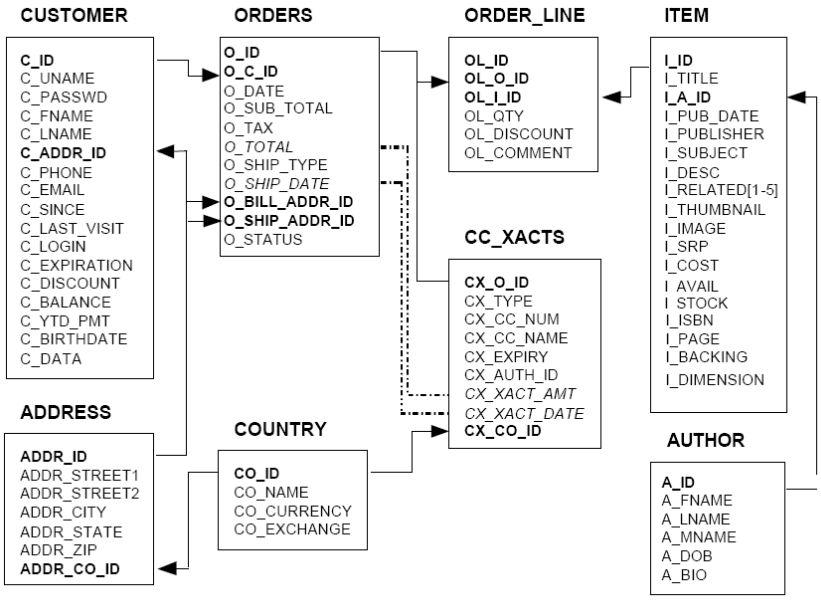


Figure A.1: The database schema for the TPC-W benchmark. Dotted lines represent one-to-one relationships. Arrows represent one-to-many relationships.

Figure A.1 depicts the database schema used for the TPC-W benchmark. The database consists of tables holding Customer, Orders, Order Line, Item, Credit Card Transaction, Country, Authors, and Address data. The size of the database can be scaled up and down based on the number of entries in the items table (the number of books in the database). The specification describes how to scale the sizes of the other tables for a given number of items.

In comparison to TPC-C, TPC-W is a CPU-bound workload with very little I/O and almost no lock contention. Chapter 2 uses to the TPC-W Shopping mix, with 80% browse queries and 20% buy queries. Chapter 4 uses the TPC-W Browsing mix, with 95% browse queries and 5% buy queries. We find that both workloads are relatively similar, especially with respect to the bottleneck resource, as both are almost always CPU-bound. In all the standard mixes, TPC-W never has a lock bottleneck even as the amount of buying is increased (and browsing is decreased).

A.2 TPC-C

TPC-C is of primary interest throughout the bottleneck analysis in Chapter 2 and the POW lock prioritization algorithm in Chapter 3.

TPC-C is an industry-standard benchmark designed by the Transaction Performance Council (TPC).

TPC-C was designed to model an OnLine Transaction Processing (OLTP) workload. In particular, the benchmark models the operations of a wholesale supplier, in which users manage the inventory at the warehouses of a company. The benchmark simulates a company which is comprised of warehouses, each of which is comprised of many districts, each of which is used by many users. Users can place new orders for items in the warehouses, request the status of orders, as well as enter payments, deliver orders, and monitor stock levels.

TPC-C consists of many different query transaction types, including New-Order, Payment, Order-Status, Delivery, and Stock-Level. The performance of New-Order transactions dominates the performance of the TPC-C workload, due to its frequency and complexity. New-Order is a frequent, mid-weight read-write transaction that simulates entering a new order for items in a warehouse. The specification dictates that New-Order must have low response times so as to ensure that users are satisfied. After New-Order, the remainder of the TPC-C workload is made up of Payment transactions (which must be at least 43% of all the queries), but they are so light-weight that they have minimal effect on the overall performance.

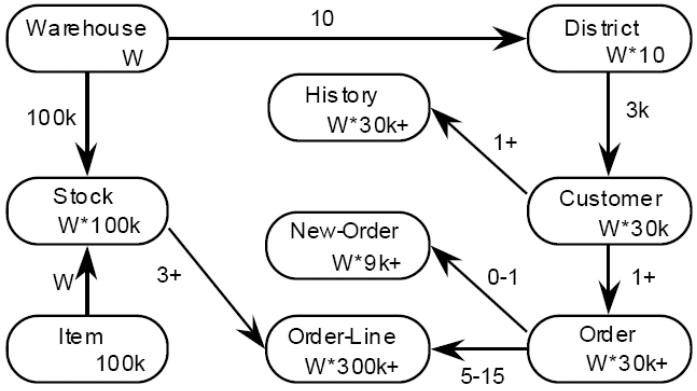


Figure A.2: The database schema for the TPC-C benchmark. Numbers in entity blocks represent the cardinality of the tables (number of rows), and are factored by W, the scale of the database (the number of Warehouses). Numbers next to relationship arrows represent the cardinality of the relationships.

Figure A.2 depicts the database schema used for the TPC-C benchmark. The TPC-C standard allows the database size to be scaled by changing the number of warehouses in the database. TPC-C specifies that the number of users ("terminals") should grow with the number of warehouses, so that each warehouse has 10 terminals. The number of users and warehouses is scaled according to this rule for some of the experiments

in Chapter 2, while the other experiments vary the number of users and warehouses independently.

In comparison to TPC-W, TPC-C is a workload with greater I/O requirements as well as more lock contention. TPC-C places much higher demands on the DBMS concurrency control subsystem, and requires strong isolation and ACID properties.

Bibliography

[1] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions. In *Proceedings of SIGMOD*, pages 71–81, 1988. 2.4.1

[2] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of Very Large Database Conference*, pages 1–12, 1988. 1.8.1, 2.1.2, 2.4.1, 3.4, 5.1.4

[3] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of Very Large Database Conference*, pages 385–396, 1989. 2.4.1

[4] R. K. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *IEEE Real-Time Systems Symposium*, pages 113–125, 1990. 2.4.1

[5] R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *Transactions on Database Systems*, 17(3):513–560, 1992. 2.4.1, 2.6.1, 2.7

[6] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987. 3.1

[7] A. Ailamaki, D. DeWitt, and M. Hill. Data page layouts for relational databases on deep memory hierarchies, 2002. 3.1.1, 3.3

[8] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMS on a modern processor: Where does time go? In *VLDB'99*, pages 266–277, 1999. 4.6.2, 5.1.6

[9] Charles Babcock. Data, data, everywhere. Technical report, InformationWeek, January 2006. 1.5.1

[10] L. Baccouche. Scheduling multi-class real-time transactions: A performance evaluation. In *PWASET 05: Proceedings of World Academy of Science, Engineering and Technology*, volume 6, June 2005. 2.4.1

[11] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981. 3.1

[12] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *TODS*, 8(4):465–483, 1983. 2.3

[13] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, 1990. 1.5.1

[14] Haran Boral and David J DeWitt. A methodology for database system performance evaluation. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 176–185, New York, NY, USA, 1984. ACM. 4.8, 5.1.5

[15] K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *Proceedings of Very Large Database Conference*, pages 328–341, 1993. 2.4.2

[16] Kurt P. Brown, Michael J. Carey, David J. DeWitt, Manish Mehta, and Jeffrey F Naughton. Resource allocation and scheduling for mixed database workloads. Technical Report TR-1095, University of Wisconsin Madison, 1992. 4.8, 5.1.5

[17] M. Carey, D. J. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In *Proc. of SIGMOD*, May 1994. 1.2, 2.3, 2.5.2, 3.3, 3.5

[18] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *Proceedings of Very Large Database Conference*, pages 397–410, 1989. 2.4.2

[19] Michael J. Carey, Sanjay Krishnamurthi, and Miron Livny. Load control for locking: The 'half-and-half' approach. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 72–84, 1990. 4.8

[20] IBM Corporation. IBM DB2 query patroller administration guide. 1.2, 2.4.2, 3.3

[21] Transaction Processing Performance Council. TPC benchmarks. `http://www.tpc.org`. 1.3, 4.3, 4.5.1, 4.8

[22] Transaction Processing Performance Council. Tpc benchmark(tm) c standard specification. Technical report, Transaction Processing Performance Council, February 2001. 1.3, 2.5.1

[23] Transaction Processing Performance Council. TPC benchmark W (web commerce). Number Revision 1.8, February 2002. 1.3, 2.5.1

[24] Benoît Dageville and Mohamed Zait. SQL memory management in oracle9i. In *Proceedings of the 28th VLDB Conference*, 2002. 5.1.6

[25] Neil Davies, Judy Holyer, and Peter Thompson. A queueing theory model that enables control of loss and delay at a network switch. Technical Report CSTR-99-011, University of Bristol Dept of Computer Science, 1 1999. 1.5.2

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardha n Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramani an, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07*, pages 205–220, 2007. 2.9, 3.1, 3.10, 4.6.1, 5.1.5

[27] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc. 1.5.1

[28] Sameh Elnitky, Erich M. Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in dynamic e-Commerce web sites. Unpublished Manuscript, May 2003. 2.4.2

[29] A. K. Erlang. Solution of some problems in the theory of probabilties of significance in automatic telephone exchanges. *Elektrotkeknikeren*, 13(513), 1917. 1.5.2

[30] IBM DB2 Product Family. `http://www.ibm.com/software/data/db2`. 3.5

[31] Peter Franaszek and John T. Robinson. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst.*, 10(1):1–28, 1985. 3.4

[32] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Wait depth limited concurrency control. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 92–101, Washington, DC, USA, 1991. IEEE Computer Society. 3.4, 3.8.3

[33] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian. Concurrency control for high contention environments. *ACM Trans. Database Syst.*, 17(2):304–345, 1992. 3.4, 4.8

[34] Davy Genbrugge, Lieven Eeckhout, and Koen De Bosschere. Microarchitecture-independent cache modeling for statistical simulation. Technical Report P106-114, ELIS, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium, 3 2003. 5.1.6

[35] Shahram Ghandeharizadeh and David J. DeWitt. Factors affecting the performance of multiuser database management systems. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 243–244, New York, NY, USA, 1990. ACM. 4.8, 5.1.5

[36] Google. Google corporate information - our philosophy. `www.google.com/corporate/tenthings.html`, 2008. 5.1.3

[37] Gartner Group/Dataquest. Server storage and RAID worldwide. Technical report, Gartner, Inc `www.gartner.com`, 1999. 1.2

[38] Varun Gupta, Mor Harchol-Balter, Karl Sigman, and Ward Whitt. Analysis of join-the-shortest-queue routing for web server farms. *Performance Evaluation*, 64(9–12), October 2007. 1.5.2

[39] Laura M. Haas, Patricia G. Selinger, Elisa Bertino, Dean Daniels, Bruce G. Lindsay, Guy M. Lohman, Yoshifumi Masunaga, C. Mohan, Pui Ng, Paul F. Wilms, and Robert A. Yost. R*: A research project on distributed relational DBMS. *IEEE Database Eng. Bull.*, 5(4):28 – 32, 1982. 1.5.1

[40] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Data access scheduling in firm real-time database systems. *Real-Time Systems*, 4(3):203–241, 1992. 3.4

[41] Hans-Ulrich Heiss and Roger Wagner. Adaptive load control in transaction processing systems. In *VLDB '91*, pages 47–54, 1991. 4.8

[42] J. Huang, J.A. Stankovic, K. Ramamritham, and D. F Towsley. On using priority inheritance in real-time databases. In *IEEE Real-Time Systems Symposium*, pages 210–221, 1991. 2.4.1

[43] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, Don Towsley, and Bhaskar Purimetla. Priority inheritance in soft real-time databases. *Real-Time Syst.*, 4(3):243–268, 1992. 3.4

[44] Jiandong Huang, John A. Stankovic, Krithi Ramamritham, and Donald F. Towsley. On using priority inheritance in real-time databases. In *IEEE Real-Time Systems Symposium*, pages 210–221, 1991. 3.4

[45] IBM. Ibm db2 universal database administration guide version 5. Document Number S10J-8157-00, 1992. 2.3, 2.4.2, 2.5.2, 2.6.1, 3.3

[46] Bruce L. Jacob, Peter M. Chen, Seth R. Silverman, and Trevor N. Mudge. An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, 45(10):1180–1194, 1996. 4.6.2, 4.7.1

[47] Abhinav Kamra, Vishal Misra, and Erich M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. In *IWQoS*, pages 47–56, 2004. 4.8

[48] K. D. Kang, Sang H. Son, and John A. Stankovic. Service differentiation in real-time main memory databases. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 29 2002. 2.4.1

[49] Naoki Katoh, Toshihide Ibaraki, and Tiko Kameda. Cautious transaction schedulers with admission control. *ACM Trans. Database Syst.*, 10(2):205–229, 1985. 4.8

[50] L. Kleinrock. Queueing systems, volume II: Computer applications, 1976. 1.5.2

[51] L. Kleinrock. Creating a mathematical theory of computer networks. *Operations Research*, 50(1), 2002. 1.5.2

[52] Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder, and Anastassia Ailamaki. PostgreSQL, chapter in Database System Concepts, by H. Korth, A. Sibershatz, and S. Sudarshan, McGraw Hill, 5th Edition. 1.2, 2.3, 2.5.2, 3.5

[53] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981. 3.1

[54] John Levon. OProfile - a system profiler for linux. `http://oprofile.sourceforge.net/`, 2002. 4.6.2

[55] Mikko H. Lipasti, Trey Cain, Milo Martin, Tim Heil, Eric Weglarz, and Todd Bezenek. Java TPC-W implementation. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2000. 2.5.1, 4.5.1

[56] David T. McWherter, Bianca Schroeder, and Anastassia Ailamaki an d Mor Harchol-Balter. Priority mechanisms for OLTP and transactional web applications. In *Proc. of ICDE*, 2004. 3.3, 3.4, 3.5, 3.6.1, 4.8

[57] David T. McWherter, Bianca Schroeder, and Anastassia Ailamaki an d Mor Harchol-Balter. Improving preemptive prioritization via statistical characterizatio n of OLTP locking. In *Proc. of ICDE*, 2005. 4.8

[58] J Moad. The real cost of storage. Technical report, eWeek `www.eweek.com`, October 2001. 1.2

[59] Fiona Fui-Hoon Nah. A study on tolerable waiting time: How long are web users willing to wait? *Behavior and Information Technology*, 23:153–163, 2004. 1.1, 5.1.3

[60] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting DBMS. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2005. 4.8

[61] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994. 1.1, 5.1.3

[62] Christopher Olston, Amit Manjhi, Charles Garrod, Anastassia Ailamaki, Bruce M. Maggs, and Todd C. Mowry. A scalability service for dynamic web applications. In *In Proc. CIDR*, pages 56–69, 2005. 5.1.5

[63] Takayuki Osogami. *Analysis of Multi-server Systems via Dimensionality Reduction of Markov Chains*. PhD in Computer Science, Carnegie Mellon University (CMU), Pittsburgh, PA, 15213, 2005. 1.5.2

[64] Özgür Ulusoy and Geneva G. Belford. Concurrency control in real-time database systems. In *CSC '92: Proceedings of the 1992 ACM annual conference on Communications*, pages 181–188, New York, NY, USA, 1992. ACM. 3.4

[65] Eileen Lin Piyush Gupta. DataJoiner: A practical approach to multidatabase access. In *PIDS '94: Proceedings of the 1994 Conference on Parallel and Distributed Information Systems*. IEEE, 1994. 1.6

[66] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978. 3.1

[67] Ann Rhee, Sumanta Chatterjee, and Tirthankar Lahiri. The Oracle database resource manager: Scheduling CPU resources at the application level. HPTS, 2001. 1.2, 2.4.2, 3.3

[68] Daniel J. Rosenkrantz, Richard E. Stearns, and II Philip M. Lewis. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, 1978. 3.3, 3.4

[69] Avi Rushinek and Sara F. Rushinek. What makes users happy? *Communications of the ACM*, 29(7):594–598, 1986. 1.1, 5.1.3

[70] In Kyung Ryu and Alexander Thomasian. Performance analysis of dynamic locking. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 698–708, 1986. 4.8

[71] In Kyung Ryu and Alexander Thomasian. Analysis of database performance with dynamic locking. *J. ACM*, 37(3):491–523, 1990. 4.5.3, 4.8, 4.11

[72] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman. How to determine a good multi-programming level for external scheduling. In *Proceedings of ICDE'06*, page 60, 2006. 4.8

[73] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association. 5.1.5, 5.1.6

[74] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report CMU-CS-98-181, Carnegie Mellon University, 1987. 3.4

[75] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990. 2.7.2, 3.4

[76] Vigyan Singhal and Alan Jay Smith. Analysis of locking behavior in three real database systems. *VLDB J.*, 6(1):40–52, 1997. 1.8.2

[77] A. J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Trans. Softw. Eng.*, 4(2):121–130, 1978. 4.7.1, 5.1.6

[78] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982. 4.7.1, 5.1.6

[79] John A. Stankovic, Sang Hyuk Son, and Jorgen Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, 1999. 2.4.1

[80] Eno Thereska. Enabling what-if explorations in systems. Carnegie Mellon University PhD Thesis, May 2007. 4.8

[81] Eno Thereska, Dushyanth Narayanan, and Gregory R. Ganger. Towards self-predicting systems: What if you could ask "what-if". *Knowl. Eng. Rev.*, 21(3):261–267, 2006. 4.8

[82] Alexander Thomasian. Performance analysis of locking policies with limited wait depth. *SIGMETRICS Perform. Eval. Rev.*, 20(1):115–127, 1992. 3.4

[83] Alexander Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Trans. Database Syst.*, 18(4):579–625, 1993. 4.1, 4.6.1, 4.8, 5.1.5

[84] Alexander Thomasian. On a more realistic lock contention model and its analysis. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 2–9, Washington, DC, USA, 1994. IEEE Computer Society. 5.1.5

[85] Alexander Thomasian. A performance comparison of locking methods with limited wait depth. *IEEE Trans. on Knowl. and Data Eng.*, 9(3):421–434, 1997. 3.4

[86] Alexander Thomasian. Concurrency control: methods, performance, and analysis. *ACM Comput. Surv.*, 30(1):70–119, 1998. 4.5.3, 4.8, 4.11

[87] Alexander Thomasian and In Kyung Ryu. A decomposition solution to the queueing network model of the centralized DBMS with static locking. In *ACM SIGMETRICS conference*, pages 82–92, 1983. 4.8

[88] William Wulf. Compilers and compiler architecture, July 1981. 1.5.1

[89] Erez Zadok, Jeffrey Osborn, Ariye Shater, Charles Wright, Kiran-Kumar Muniswamy-Reddy, and Jason Nieh. Reducing storage management costs via informed user-based policies. In *Conference on Mass Storage Systems and Technologies (MSST) '04 Proceedings*, pages 193–198. IEEE, April 2004. 1.2

[90] Xiaodong Zhang, Zhichun Zhu, and Xing Du. Analysis of commercial workload on SMP multiprocessors. *Proceedings of Performance 1999*, August 1999. 4.6.2, 4.7.1

[91] George K. Zipf. Human behavior and the principle of least-effort. page 573, 1949. 5.1.5